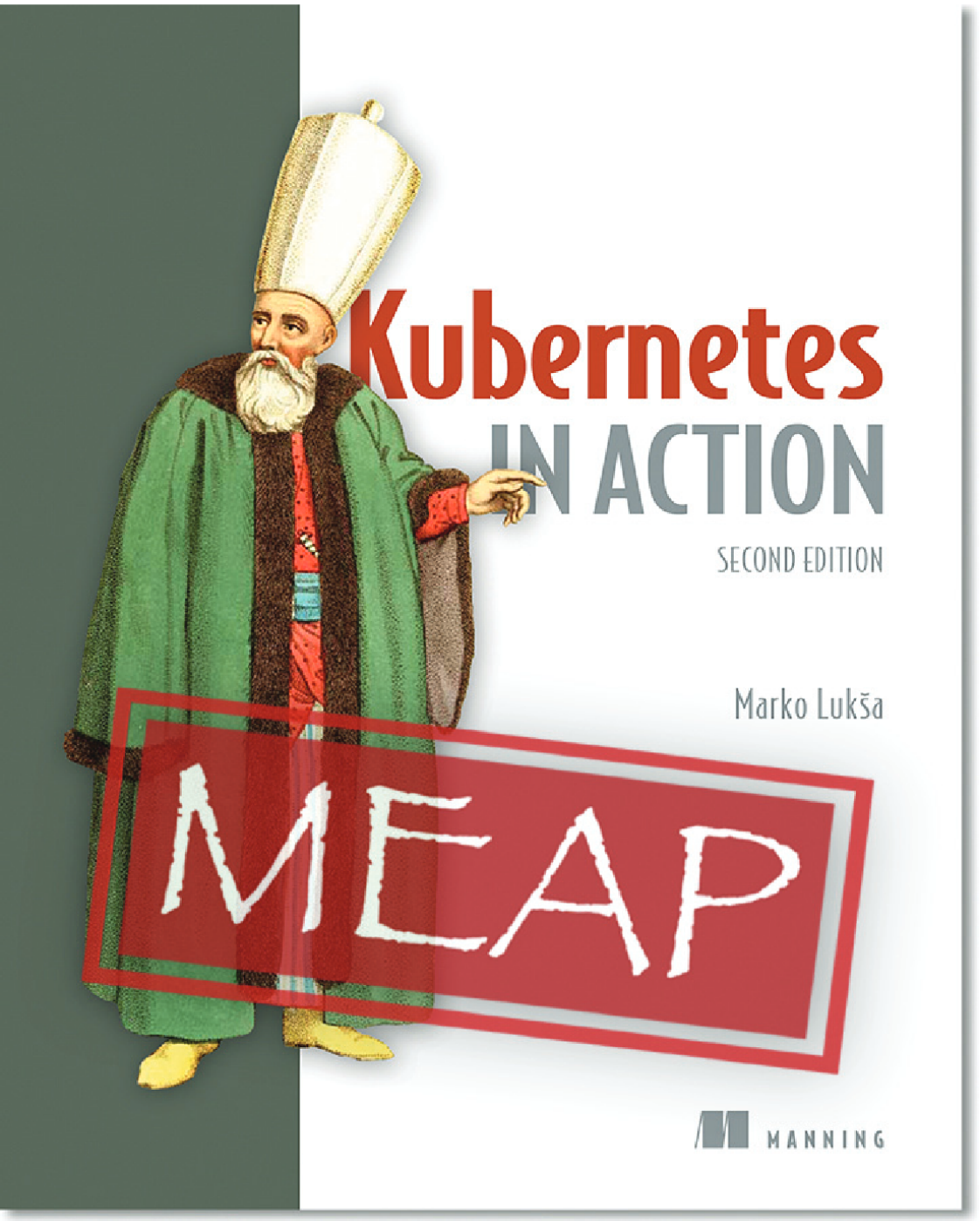

Kubernetes
IN ACTION
SECOND EDITION
Marko Lukša
MEAP
MANNING

MEAP Edition
Manning Early Access Program
Kubernetes in Action
Second edition
Version 5

For more information on this and other Manning titles go to
manning.com

MEAP版
曼宁早期访问计划
Kubernetes 的实际应用
第二版
版本5

有关此产品和曼宁其他产品的更多信息，请访问 manning.com

welcome

Thank you for purchasing the MEAP for Kubernetes in Action 2nd Edition.

As part of my work at Red Hat, I started using Kubernetes in 2014, even before version 1.0 was released. Those were interesting times. Not many people working in the software industry knew about Kubernetes, and there was no real community yet. There were hardly any blog posts about it and the documentation was still very basic. Kubernetes itself was ridden with bugs. When you combine all these facts, you can imagine that working with Kubernetes was extremely difficult.

In 2015 I was asked by Manning to write the first edition of this book. The originally planned 300-page book grew to over 600 pages full of information. The writing forced me to also research those parts of Kubernetes that I wouldn't have looked at more closely otherwise. I put most of what I learned into the book. Judging by their reviews and comments, readers love a detailed book like this.

The plan for the second edition of the book is to add even more information and to rearrange some of the existing content. The exercises in this book will take you from deploying a trivial application that initially uses only the basic features of Kubernetes to a full-fledged application that incorporates additional features as the book introduces them.

The book is divided into five parts. In the first part, after the introduction of Kubernetes and containers, you'll deploy the application in the simplest way. In the second part you'll learn the main concepts used to describe and deploy your application. After that you'll explore the inner workings of Kubernetes components. This will give you a good foundation to learn the difficult part - how to manage Kubernetes in production. In the last part of the book you'll learn about best practices and how to extend Kubernetes.

I hope you all like this second edition even better than the first, and if you're reading the book for the first time, your feedback will be even more valuable. If any part of the book is difficult to understand, please post your questions, comments or suggestions in the liveBook forum.

Thank you for helping me write the best book possible.

—Marko Lukša

欢迎

感谢您购买《MEAP for Kubernetes in Action》第二版。

作为我在 Red Hat 工作的一部分，我从 2014 年开始使用 Kubernetes，甚至在 1.0 版本发布之前。那是一段有趣的时光。软件行业的人了解 Kubernetes 的人并不多，也没有真正的社区。几乎没有任何关于它的博客文章，并且文档仍然非常基础。 Kubernetes 本身就充满了错误。当你结合所有这些事实时，你可以想象使用 Kubernetes 是极其困难的。

2015 年，曼宁邀请我撰写本书的第一版。原本的

原本计划 300 页的书后来增加到 600 多页，内容丰富。这篇文章迫使我也研究 Kubernetes 的那些部分，否则我不会更仔细地研究这些部分。我把学到的大部分知识都写进了书里。从他们的评论和评论来看，读者喜欢这样一本详细的书。

本书第二版的计划是添加更多信息并重新安排一些现有内容。本书中的练习将带您从部署一个最初仅使用 Kubernetes 基本功能的简单应用程序，到部署一个包含本书介绍的附加功能的成熟应用程序。

本书分为五个部分。在第一部分中，在介绍 Kubernetes 和容器之后，您将以最简单的方式部署应用程序。在第二部分中，您将学习用于描述和部署应用程序的主要概念。之后，您将探索 Kubernetes 组件的内部工作原理。这将为您学习困难的部分（如何在生产中管理 Kubernetes）奠定良好的基础。在本书的最后部分，您将了解最佳实践以及如何扩展 Kubernetes。

我希望你们都比第一版更喜欢第二版，如果您是第一次阅读这本书，您的反馈将更有价值。如果本书的任何部分难以理解，请在liveBook论坛中发表您的问题、评论或建议。

感谢您帮助我写出最好的书。

——马尔科·卢克萨

brief contents

简要内容

1

Introducing Kubernetes

This chapter covers

- **Introductory information about Kubernetes and its origins**
- **Why Kubernetes has seen such wide adoption**
- **How Kubernetes transforms your data center**
- **An overview of its architecture and operation**
- **How and if you should integrate Kubernetes into your own organization**

Before you can learn about the ins and outs of running applications with Kubernetes, you must first gain a basic understanding of the problems Kubernetes is designed to solve, how it came about, and its impact on application development and deployment. This first chapter is intended to give a general overview of these topics.

1.1 Introducing Kubernetes

The word *Kubernetes* is Greek for pilot or helmsman, the person who steers the ship - the person standing at the helm (the ship's wheel). A helmsman is not necessarily the same as a captain. A captain is responsible for the ship, while the helmsman is the one who steers it.

After learning more about what Kubernetes does, you'll find that the name hits the spot perfectly. A helmsman maintains the course of the ship, carries out the orders given by the captain and reports back the ship's heading. Kubernetes steers your applications and reports on their status while you - the captain - decide where you want the system to go.

1

介绍 Kubernetes

本章涵盖

- 有关 Kubernetes 及其起源的介绍信息
- 为什么 Kubernetes 得到如此广泛的采用
- Kubernetes 如何改造您的数据中心
- 其架构和操作概述
- 如何以及是否应该将 Kubernetes 集成到您自己的组织中

在了解使用 Kubernetes 运行应用程序的细节之前，您必须首先对 Kubernetes 旨在解决的问题、它是如何产生的以及它对应用程序开发和部署的影响有一个基本的了解。第一章旨在对这些主题进行总体概述。

1.1 Kubernetes 简介

Kubernetes 这个词在希腊语中是飞行员或舵手的意思，即掌舵的人（掌舵者）。舵手不一定与船长相同。船长负责掌管这艘船，而舵手则是掌舵的人。

在详细了解 Kubernetes 的用途后，您会发现这个名字非常切中要害。舵手维持船舶的航向，执行船长发出的命令并报告船舶的航向。 Kubernetes 引导您的应用程序并报告其状态，而您（船长）则决定您希望系统走向何处。

How to pronounce Kubernetes and what is k8s?

The correct Greek pronunciation of Kubernetes, which is *Kie-ver-nee-tees*, is different from the English pronunciation you normally hear in technical conversations. Most often it's *Koo-ber-netties* or *Koo-ber-nay'-tace*, but you may also hear *Koo-ber-nets*, although rarely.

In both written and oral conversations, it's also referred to as *Kube* or *K8s*, pronounced *Kates*, where the 8 signifies the number of letters omitted between the first and last letter.

1.1.1 Kubernetes in a nutshell

Kubernetes is a software system for automating the deployment and management of complex, large-scale application systems composed of computer processes running in containers. Let's learn what it does and how it does it.

ABSTRACTING AWAY THE INFRASTRUCTURE

When software developers or operators decide to deploy an application, they do this through Kubernetes instead of deploying the application to individual computers. Kubernetes provides an abstraction layer over the underlying hardware to both users and applications.

As you can see in the following figure, the underlying infrastructure, meaning the computers, the network and other components, is hidden from the applications, making it easier to develop and configure them.

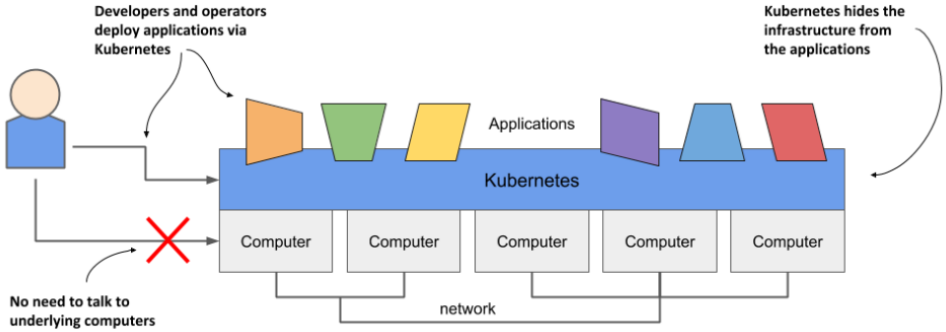


Figure 1.1 Infrastructure abstraction using Kubernetes

STANDARDIZING HOW WE DEPLOY APPLICATIONS

Because the details of the underlying infrastructure no longer affect the deployment of applications, you deploy applications to your corporate data center in the same way as you do in the cloud. A single manifest that describes the application can be used for local deployment and for deploying on any cloud provider. All differences in the underlying infrastructure are handled by Kubernetes, so you can focus on the application and the business logic it contains.

Kubernetes 如何发音以及什么是 k8s?

Kubernetes 的正确希腊语发音是 Kie-ver-nee-tees，与您通常在技术对话中听到的英语发音不同。最常见的是 Koo-ber-netties 或 Koo-ber-nay' -tace，但您也可能听到 Koo-ber-nets，尽管很少。

在书面和口头对话中，它也被称为 Kube 或 K8s，发音为 Kates，其中 8 表示第一个和最后一个字母之间省略的字母数。

1.1.1 Kubernetes 概述

Kubernetes 是一个软件系统，用于自动部署和管理由在容器中运行的计算机进程组成的复杂、大规模应用程序系统。让我们了解一下它的作用以及它是如何做的。

抽象化基础设施

当软件开发人员或运营商决定部署应用程序时，他们会通过 Kubernetes 来完成此操作，而不是将应用程序部署到单独的计算机上。 Kubernetes 为用户和应用程序提供了底层硬件的抽象层。

如下图所示，底层基础设施（即计算机、网络和其他组件）对应用程序是隐藏的，从而使开发和配置它们变得更加容易。

图 1.1 使用 Kubernetes 进行基础设施抽象

标准化我们部署应用程序的方式

由于底层基础设施的细节不再影响应用程序的部署，因此您可以像在云中一样将应用程序部署到企业数据中心。描述应用程序的单个清单可用于本地部署以及在任何云提供商上的部署。底层基础设施中的所有差异都由 Kubernetes 处理，因此您可以专注于应用程序及其包含的业务逻辑。

DEPLOYING APPLICATIONS DECLARATIVELY

Kubernetes uses a declarative model to define an application, as shown in the next figure. You describe the components that make up your application and Kubernetes turns this description into a running application. It then keeps the application healthy by restarting or recreating parts of it as needed.

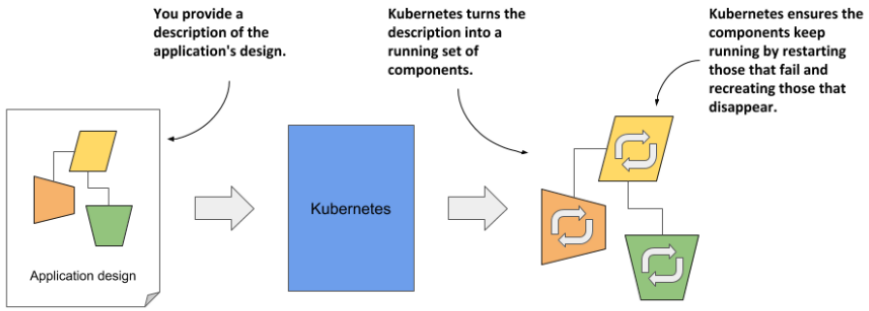


Figure 1.2 The declarative model of application deployment

Whenever you change the description, Kubernetes will take the necessary steps to reconfigure the running application to match the new description, as shown in the next figure.

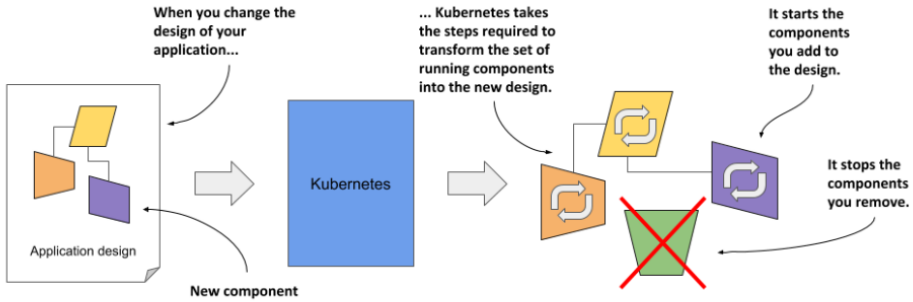


Figure 1.3 Changes in the description are reflected in the running application

TAKING ON THE DAILY MANAGEMENT OF APPLICATIONS

As soon as you deploy an application to Kubernetes, it takes over the daily management of the application. If the application fails, Kubernetes will automatically restart it. If the

以声明方式部署应用程序

Kubernetes 使用声明式模型来定义应用程序，如下图所示。您描述组成应用程序的组件，然后 Kubernetes 将此描述转化为正在运行的应用程序。然后，它通过根据需要重新启动或重新创建应用程序的一部分来保持应用程序的健康。

图1.2 应用程序部署的声明式模型

每当您更改描述时，Kubernetes 都会采取必要的步骤
重新配置正在运行的应用程序以匹配新的描述，如下图所示。

图1.3 描述的变化反映在正在运行的应用程序中

进行应用程序的日常管理

一旦您将应用程序部署到 Kubernetes，它就会接管该应用程序的日常管理。如果应用程序失败，Kubernetes会自动重启它

hardware fails or the infrastructure topology changes so that the application needs to be moved to other machines, Kubernetes does this all by itself. The engineers responsible for operating the system can focus on the big picture instead of wasting time on the details.

To circle back to the sailing analogy: the development and operations engineers are the ship's officers who make high-level decisions while sitting comfortably in their armchairs, and Kubernetes is the helmsman who takes care of the low-level tasks of steering the system through the rough waters your applications and infrastructure sail through.

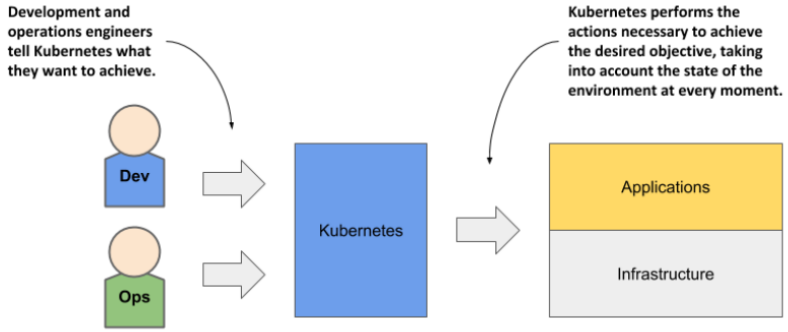


Figure 1.4 Kubernetes takes over the management of applications

Everything that Kubernetes does and all the advantages it brings requires a longer explanation, which we'll discuss later. Before we do that, it might help you to know how it all began and where the Kubernetes project currently stands.

1.1.2 About the Kubernetes project

Kubernetes was originally developed by Google. Google has practically always run applications in containers. As early as 2014, it was reported that they start two billion containers every week. That's over 3,000 containers per second, and the figure is much higher today. They run these containers on thousands of computers distributed across dozens of data centers around the world. Now imagine doing all this manually. It's clear that you need automation, and at this massive scale, it better be perfect.

About Borg and Omega - the predecessors of Kubernetes

The sheer scale of Google's workload has forced them to develop solutions to make the development and management of thousands of software components manageable and cost-effective. Over the years, Google developed an internal system called *Borg* (and later a new system called Omega) that helped both application developers and operators manage these thousands of applications and services.

硬件故障或基础设施拓扑发生变化，导致应用程序需要移动到其他机器，Kubernetes 会自行完成这一切。负责操作系统的工程师可以专注于大局，而不是在细节上浪费时间。

回到航海的比喻：开发和运营工程师是船上的高级官员，他们舒适地坐在扶手椅上做出高层决策，而 Kubernetes 是舵手，负责通过系统引导系统的低级任务。您的应用程序和基础设施将在波涛汹涌的大海中航行。

图1.4 Kubernetes接管应用程序的管理

Kubernetes 所做的一切以及它带来的所有优势都需要更长的解释，我们将在稍后讨论。在我们这样做之前，它可能会帮助您了解这一切是如何开始的以及 Kubernetes 项目目前的状况。

1.1.2 关于 Kubernetes 项目

Kubernetes 最初由 Google 开发。谷歌几乎一直在运行

容器中的应用。早在2014年，就有报道称他们每周启动20亿个集装箱。每秒处理超过 3,000 个容器，而今天这个数字要高得多。他们在分布在世界各地数十个数据中心的数千台计算机上运行这些容器。现在想象一下手动完成这一切。很明显，您需要自动化，而且在如此大规模的情况下，它最好是完美的。

关于博格和欧米茄 - Kubernetes 的前身

Google 工作量的巨大规模迫使他们开发解决方案，使数千个软件组件的开发和管理变得易于管理且具有成本效益。多年来，Google 开发了一个名为 Borg 的内部系统（后来又开发了一个名为 Omega 的新系统），帮助应用程序开发人员和运营商管理这数千个应用程序和服务。

In addition to simplifying development and management, these systems have also helped them to achieve better utilization of their infrastructure. This is important in any organization, but when you operate hundreds of thousands of machines, even tiny improvements in utilization mean savings in the millions, so the incentives for developing such a system are clear.

NOTE Data on Google's energy use suggests that they run around 900,000 servers.

Over time, your infrastructure grows and evolves. Every new data center is state-of-the-art. Its infrastructure differs from those built in the past. Despite the differences, the deployment of applications in one data center should not differ from deployment in another data center. This is especially important when you deploy your application across multiple zones or regions to reduce the likelihood that a regional failure will cause application downtime. To do this effectively, it's worth having a consistent method for deploying your applications.

About Kubernetes - the open-source project - and commercial products derived from it

Based on the experience they gained while developing Borg, Omega and other internal systems, in 2014 Google introduced Kubernetes, an open-source project that can now be used and further improved by everyone.

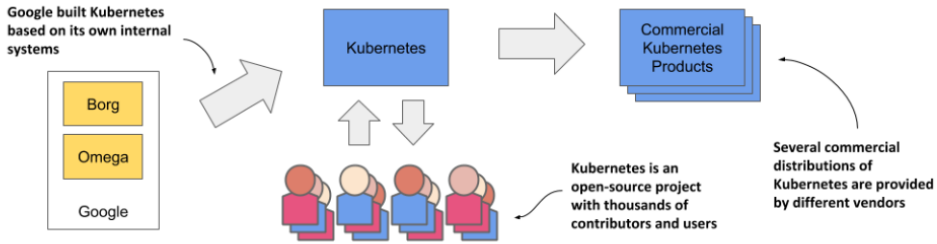


Figure 1.5 The origins and state of the Kubernetes open-source project

As soon as Kubernetes was announced, long before version 1.0 was officially released, other companies, such as Red Hat, who has always been at the forefront of open-source software, quickly stepped on board and helped develop the project. It eventually grew far beyond the expectations of its founders, and today is arguably one of the world's leading open-source projects, with dozens of organizations and thousands of individuals contributing to it.

Several companies are now offering enterprise-quality Kubernetes products that are built from the open-source project. These include Red Hat OpenShift, Pivotal Container Service, Rancher and many others.

除了简化开发和管理之外，这些系统还有助于
他们 到 达到 更好的 利用率 的 他们的 基础设施。 这 是 重要的 在 任何
组织，但是当 你 操作 数百 的 数千 机器的数量，甚至 微小的
利用率的提高意味着数百万美元的节省，因此开发这样一个系统的动机是明确的。

注意 Google 能源使用数据表明，他们运行着大约 900,000 台服务器。

随着时间的推移，您的基础设施不断增长和发展。每个新的数据中心都是最先进的。它的基础设施与过去建造的不同。尽管存在差异，一个数据中心中的应用程序部署不应与另一数据中心中的部署有所不同。当您跨多个区域或区域部署应用程序时，这一点尤其重要，以减少区域故障导致应用程序停机的可能性。为了有效地做到这一点，值得采用一致的方法来部署应用程序。

关于 KUBERNETES - 开源项目 - 以及由此衍生的商业产品

基于开发 Borg、Omega 等内部系统的经验，Google 在 2014 年推出了 Kubernetes 这个开源项目，现在每个人都可以使用并进一步改进。

图1.5 Kubernetes开源项目的起源和状态

Kubernetes一经宣布，早在1.0版本正式发布之前，其他公司，比如一直走在开源软件前沿的红帽，就迅速介入并帮助开发该项目。它最终的发展远远超出了其创始人的预期，今天可以说是世界领先的开源项目之一，有数十个组织和数千个人为其做出贡献。

现在有几家公司提供基于开源项目构建的企业级 Kubernetes 产品。其中包括红帽 OpenShift、Pivotal Container Service、Rancher 等。

How Kubernetes grew a whole new cloud-native eco-system

Kubernetes has also spawned many other related open-source projects, most of which are now under the umbrella of the *Cloud Native Computing Foundation* (CNCF), which is part of the *Linux Foundation*.

CNCF organizes several KubeCon - CloudNativeCon conferences per year - in North America, Europe and China. In 2019, the total number of attendees exceeded 23,000, with KubeCon North America reaching an overwhelming number of 12,000 participants. These figures show that Kubernetes has had an incredibly positive impact on the way companies around the world deploy applications today. It wouldn't have been so widely adopted if that wasn't the case.

1.1.3 Understanding why Kubernetes is so popular

In recent years, the way we develop applications has changed considerably. This has led to the development of new tools like Kubernetes, which in turn have fed back and fuelled further changes in application architecture and the way we develop them. Let's look at concrete examples of this.

Automating the management of microservices

In the past, most applications were large monoliths. The components of the application were tightly coupled, and they all ran in a single computer process. The application was developed as a unit by a large team of developers and the deployment of the application was straightforward. You installed it on a powerful computer and provided the little configuration it required. Scaling the application horizontally was rarely possible, so whenever you needed to increase the capacity of the application, you had to upgrade the hardware - in other words, scale the application vertically.

Then came the microservices paradigm. The monoliths were divided into dozens, sometimes hundreds, of separate processes, as shown in the following figure. This allowed organizations to divide their development departments into smaller teams where each team developed only a part of the entire system - just some of the microservices.

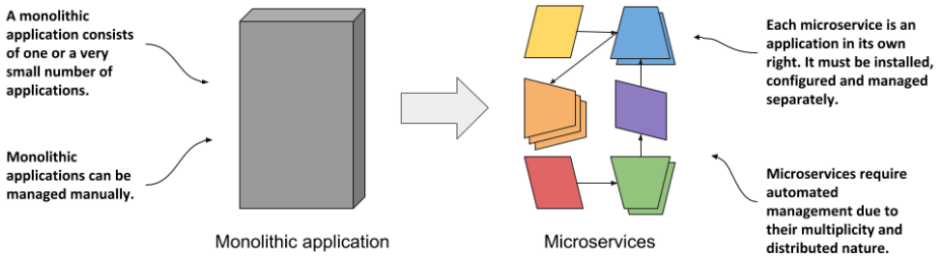


Figure 1.6 Comparing monolithic applications with microservices

Kubernetes 如何打造全新的云原生生态系统

Kubernetes 还催生了许多其他相关的开源项目，其中大部分现在都在云原生计算基金会 (CNCF) 的保护下，该基金会是 Linux 基金会的一部分。

CNCF 每年在北美、欧洲和中国组织多次 KubeCon - CloudNativeCon 会议。 2019年，参会总人数超过23,000人，其中KubeCon北美参会人数达到压倒性的12,000人。这些数字表明，Kubernetes 对当今世界各地的公司部署应用程序的方式产生了令人难以置信的积极影响。如果不是这样的话，它就不会被如此广泛地采用。

1.1.3 了解 Kubernetes 为何如此受欢迎

近年来，我们开发应用程序的方式发生了很大变化。这导致了 Kubernetes 等新工具的开发，而这些工具反过来又反馈并推动了应用程序架构和我们开发它们的方式的进一步变化。让我们看一下具体的例子。

自动化微服务管理

过去，大多数应用程序都是大型整体应用程序，该应用程序的组件紧密耦合，并且它们都在单个计算机进程中运行。该应用程序是由大型开发团队作为一个单元开发的，并且应用程序的部署非常简单。您将其安装在一台功能强大的计算机上并提供了所需的少量配置。水平扩展应用程序几乎是不可能的，因此每当您需要增加应用程序的容量时，就必须升级硬件 - 换句话说，垂直扩展应用程序。

然后是微服务范式。这些整体被分为数十个、有时数百个独立的进程，如下图所示。这使得组织可以将其开发部门划分为更小的团队，每个团队仅开发整个系统的一部分 - 只是一些微服务。

图1

Each microservice is now a separate application with its own development and release cycle. The dependencies of different microservices will inevitably diverge over time. One microservice requires one version of a library, while another microservice requires another, possibly incompatible, version of the same library. Running the two applications in the same operating system becomes difficult.

Fortunately, containers alone solve this problem where each microservice requires a different environment, but each microservice is now a separate application that must be managed individually. The increased number of applications makes this much more difficult.

Individual parts of the entire application no longer need to run on the same computer, which makes it easier to scale the entire system, but also means that the applications need to be configured to communicate with each other. For systems with only a handful of components, this can usually be done manually, but it's now common to see deployments with well over a hundred microservices.

When the system consists of many microservices, automated management is crucial. Kubernetes provides this automation. The features it offers make the task of managing hundreds of microservices almost trivial.

BRIDGING THE DEV AND OPS DIVIDE

Along with these changes in application architecture, we've also seen changes in the way teams develop and run software. It used to be normal for a development team to build the software in isolation and then throw the finished product over the wall to the operations team, who would then deploy it and manage it from there.

With the advent of the Dev-ops paradigm, the two teams now work much more closely together throughout the entire life of the software product. The development team is now much more involved in the daily management of the deployed software. But that means that they now need to know about the infrastructure on which it's running.

As a software developer, your primary focus is on implementing the business logic. You don't want to deal with the details of the underlying servers. Fortunately, Kubernetes hides these details.

STANDARDIZING THE CLOUD

Over the past decade or two, many organizations have moved their software from local servers to the cloud. The benefits of this seem to have outweighed the fear of being locked-in to a particular cloud provider, which is caused by relying on the provider's proprietary APIs to deploy and manage applications.

Any company that wants to be able to move its applications from one provider to another will have to make additional, initially unnecessary efforts to abstract the infrastructure and APIs of the underlying cloud provider from the applications. This requires resources that could otherwise be focused on building the primary business logic.

Kubernetes has also helped in this respect. The popularity of Kubernetes has forced all major cloud providers to integrate Kubernetes into their offerings. Customers can now deploy applications to any cloud provider through a standard set of APIs provided by Kubernetes.

每个微服务现在都是一个单独的应用程序，具有自己的开发和发布周期。这
依赖关系 的 不同的 微服务 将要 不可避免地 发散 超过 时间。 一
微服务需要一个库版本，而另一个微服务需要同一库的另一个版本（可能不兼容）。在同一操作系统中运行两个应用程序变得很困难。

幸运的是，容器单独解决了这个问题，其中每个微服务需要不同的环境，但每个微服务现在都是一个单独的应用程序，必须单独管理。申请数量的增加使得这变得更加困难。

整个应用程序的各个部分不再需要在同一台计算机上运行，这使得扩展整个系统变得更容易，但也意味着需要将应用程序配置为相互通信。对于只有少数组件的系统，这通常可以手动完成，但现在通常会看到超过一百个微服务的部署。

当系统由很多微服务组成时，自动化管理至关重要。 Kubernetes 提供了这种自动化。它提供的功能使管理数百个微服务的任务变得几乎微不足道。

弥合开发和运营鸿沟

除了应用程序架构的这些变化之外，我们还看到团队开发和运行软件的方式发生了变化。过去，开发团队通常会孤立地构建软件，然后将成品扔给运营团队，然后由运营团队进行部署和管理。

随着 Dev-ops 范式的出现，两个团队现在在软件产品的整个生命周期中更加紧密地合作。开发团队现在更多地参与已部署软件的日常管理。但这意味着他们现在需要了解其运行的基础设施。

作为软件开发人员，您的主要关注点是实现业务逻辑。您不想处理底层服务器的详细信息。幸运的是，Kubernetes 隐藏了这些细节。

云标准化

在过去的一两年里，许多组织已将其软件从本地服务器转移到云端。这样做的好处似乎超过了对被锁定到特定云提供商的担忧，这是由于依赖提供商的专有 API 来部署和管理应用程序而造成的。

任何希望能够将其应用程序从一个提供商转移到另一个提供商的公司都必须做出额外的、最初不必要的努力，将底层云提供商的基础设施和 API 从应用程序中抽象出来。这需要原本可以专注于构建主要业务逻辑的资源。

Kubernetes 在这方面也提供了帮助。 Kubernetes 的流行迫使所有主要云提供商将 Kubernetes 集成到他们的产品中。客户现在可以通过 Kubernetes 提供的一组标准 API 将应用程序部署到任何云提供商。

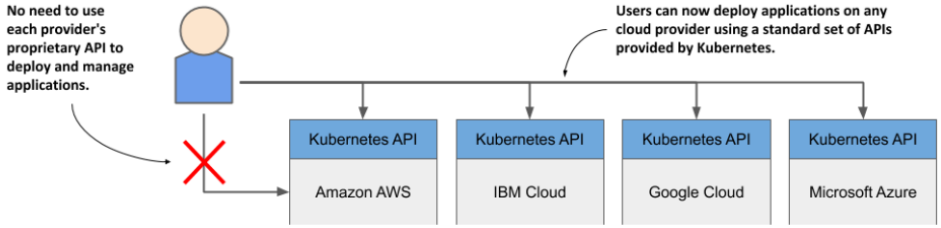


Figure 1.7 Kubernetes has standardized how you deploy applications on cloud providers

If the application is built on the APIs of Kubernetes instead of directly on the proprietary APIs of a specific cloud provider, it can be transferred relatively easily to any other provider.

1.2 Understanding Kubernetes

The previous section explained the origins of Kubernetes and the reasons for its wide adoption. In this section we'll take a closer look at what exactly Kubernetes is.

1.2.1 Understanding how Kubernetes transforms a computer cluster

Let's take a closer look at how the perception of the data center changes when you deploy Kubernetes on your servers.

KUBERNETES IS LIKE AN OPERATING SYSTEM FOR COMPUTER CLUSTERS

One can imagine Kubernetes as an operating system for the cluster. The next figure illustrates the analogies between an operating system running on a computer and Kubernetes running on a cluster of computers.

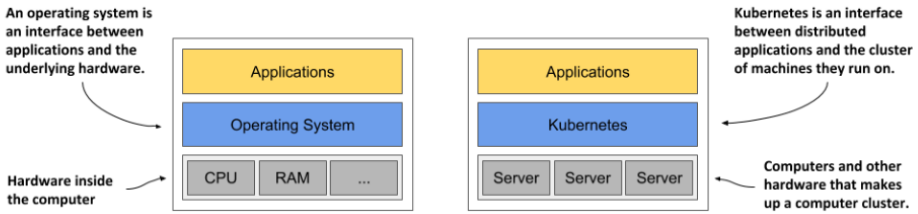


Figure 1.8 Kubernetes is to a computer cluster what an Operating System is to a computer

Just as an operating system supports the basic functions of a computer, such as scheduling processes onto its CPUs and acting as an interface between the application and the computer's hardware, Kubernetes schedules the components of a distributed application onto

图 1.7 Kubernetes 标准化了您在云提供商上部署应用程序的方式

如果应用程序是基于 Kubernetes 的 API 构建的，而不是直接基于特定云提供商的专有 API 构建的，那么它可以相对轻松地转移到任何其他提供商。

1.2 理解Kubernetes

上一节解释了 Kubernetes 的起源及其广泛采用的原因。在本节中，我们将仔细了解 Kubernetes 到底是什么。

1.2.1 了解 Kubernetes 如何改造计算机集群

让我们仔细看看当您在服务器上部署 Kubernetes 时，数据中心的认知会发生怎样的变化。

Kubernetes 就像计算机集群的操作系统

人们可以将 Kubernetes 想象为集群的操作系统。下图说明了
这 类比 之间 一个 操作 系统 跑步 在 a 电脑 和
Kubernetes 运行在计算机集群上。

图 1.8 Kubernetes 之于计算机集群就像操作系统之于计算机

正如操作系统支持计算机的基本功能一样，例如将进程调度到 CPU 上并充当应用程序和计算机硬件之间的接口

individual computers in the underlying computer cluster and acts as an interface between the application and the cluster.

It frees application developers from the need to implement infrastructure-related mechanisms in their applications; instead, they rely on Kubernetes to provide them. This includes things like:

- *service discovery* - a mechanism that allows applications to find other applications and use the services they provide,
- *horizontal scaling* - replicating your application to adjust to fluctuations in load,
- *load-balancing* - distributing load across all the application replicas,
- *self-healing* - keeping the system healthy by automatically restarting failed applications and moving them to healthy nodes after their nodes fail,
- *leader election* - a mechanism that decides which instance of the application should be active while the others remain idle but ready to take over if the active instance fails.

By relying on Kubernetes to provide these features, application developers can focus on implementing the core business logic instead of wasting time integrating applications with the infrastructure.

How Kubernetes fits into a computer cluster

To get a concrete example of how Kubernetes is deployed onto a cluster of computers, look at the following figure.

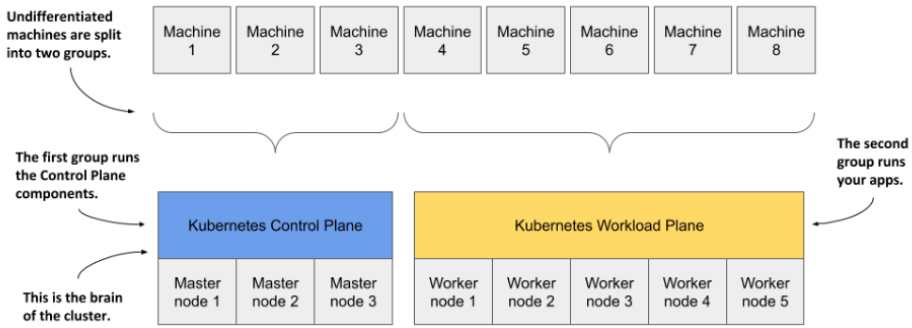


Figure 1.9 Computers in a Kubernetes cluster are divided into the Control Plane and the Workload Plane

You start with a fleet of machines that you divide into two groups - the master and the worker nodes. The master nodes will run the Kubernetes Control Plane, which represents the brain of your system and controls the cluster, while the rest will run your applications - your workloads - and will therefore represent the Workload Plane.

底层计算机集群中的各个计算机，充当应用程序和集群之间的接口。

它使应用程序开发人员无需 实施基础设施相关
其应用中的机制；相反，他们依赖 Kubernetes 来提供它们。这包括：

- 服务发现 - 一种允许应用程序查找其他应用程序并使用它们提供的服务的机制，
- 水平缩放 - 复制您的应用程序以适应负载波动，
- 负载平衡 - 在所有应用程序副本之间分配负载，
- 自愈 - 保持 这 系统 健康 经过 自动地 重新启动 失败的 应用程序并在节点发生故障后将它们移至健康节点，
- 领导者选举 - 一种机制，决定应用程序的哪个实例应处于活动状态，而其他实例保持空闲状态，但准备在活动实例发生故障时接管。

通过依靠 Kubernetes 提供这些功能，应用程序开发人员可以专注于实现核心业务逻辑，而不是浪费时间将应用程序与基础设施集成。

Kubernetes 如何融入计算机集群

要获取 Kubernetes 如何部署到计算机集群上的具体示例，请查看下图。

图1.9 Kubernetes集群中的计算机分为控制平面和工作负载平面

您从一组机器开始，将其分为两组 - 主节点和工作节点。主节点将运行 Kubernetes 控制平面，它代表系统的大脑并控制集群，而其余节点将运行您的应用程序（您的工作负载），因此代表工作负载平面。

NOTE The Workload Plane is sometimes referred to as the Data Plane, but this term could be confusing because the plane doesn't host data but applications. Don't be confused by the term "plane" either - in this context you can think of it as the "surface" the applications run on.

Non-production clusters can use a single master node, but highly available clusters use at least three physical master nodes to host the Control Plane. The number of worker nodes depends on the number of applications you'll deploy.

HOW ALL CLUSTER NODES BECOME ONE LARGE DEPLOYMENT AREA

After Kubernetes is installed on the computers, you no longer need to think about individual computers when deploying applications. Regardless of the number of worker nodes in your cluster, they all become a single space where you deploy your applications. You do this using the Kubernetes API, which is provided by the Kubernetes Control Plane.

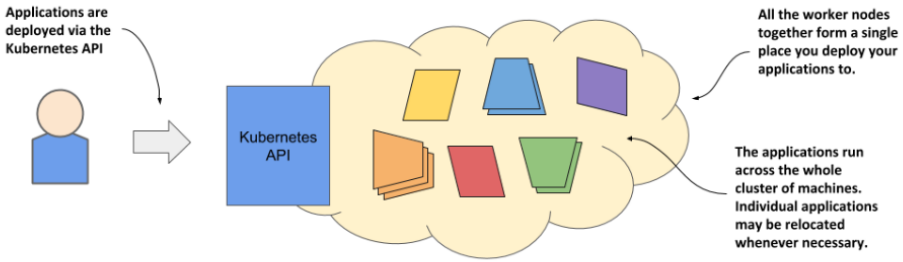


Figure 1.10 Kubernetes exposes the cluster as a uniform deployment area

When I say that all worker nodes become one space, I don't want you to think that you can deploy an extremely large application that is spread across several small machines. Kubernetes doesn't do magic tricks like this. Each application must be small enough to fit on one of the worker nodes.

What I meant was that when deploying applications, it doesn't matter which worker node they end up on. Kubernetes may later even move the application from one node to another. You may not even notice when that happens, and you shouldn't care.

1.2.2 The benefits of using Kubernetes

You've already learned why many organizations across the world have welcomed Kubernetes into their data centers. Now, let's take a closer look at the specific benefits it brings to both development and IT operations teams.

SELF-SERVICE DEPLOYMENT OF APPLICATIONS

Because Kubernetes presents all its worker nodes as a single deployment surface, it no longer matters which node you deploy your application to. This means that developers can

注意：工作负载平面有时被称为数据平面，但这个术语可能会令人困惑
因为飞机不承载数据，而是承载应用程序。也不要被“飞机”一词混淆 - 在这个
你可以将其视为应用程序运行的“表面”

非生产集群可以使用单个主节点，但高可用集群至少使用三个物理主节点来托管控制平面。工作节点的数量取决于您要部署的应用程序的数量。

所有集群节点如何成为一个大的部署区域

在计算机上安装 Kubernetes 后，部署应用程序时不再需要考虑单独的计算机。无论集群中的工作节点数量有多少，它们都会成为您部署应用程序的单个空间。您可以使用 Kubernetes API 来执行此操作，该 API 由 Kubernetes 控制平面提供。

图1.10 Kubernetes将集群暴露为统一部署区域

当我说所有工作节点成为一个空间时，我不想让你认为你可以部署一个分布在几台小机器上的非常大的应用程序。Kubernetes 不会做这样的魔术。每个应用程序必须足够小以适合其中一个工作节点。

我的意思是，部署应用程序时，它们最终位于哪个工作节点并不重要。 Kubernetes 稍后甚至可能将应用程序从一个节点移动到另一个节点。当这种情况发生时，你甚至可能没有注意到，你也不应该在意。

1.2.2 使用 Kubernetes 的好处

您已经了解了为什么世界各地的许多组织都欢迎 Kubernetes 进入其数据中心。现在，让我们仔细看看它为开发和 IT 运营团队带来的具体好处。

应用程序的自助部署

由于 Kubernetes 将其所有工作节点呈现为单个部署表面，因此将应用程序部署到哪个节点不再重要

now deploy applications on their own, even if they don't know anything about the number of nodes or the characteristics of each node.

In the past, the system administrators were the ones who decided where each application should be placed. This task is now left to Kubernetes. This allows a developer to deploy applications without having to rely on other people to do so. When a developer deploys an application, Kubernetes chooses the best node on which to run the application based on the resource requirements of the application and the resources available on each node.

REDUCING COSTS VIA BETTER INFRASTRUCTURE UTILIZATION

If you don't care which node your application lands on, it also means that it can be moved to any other node at any time without you having to worry about it. Kubernetes may need to do this to make room for a larger application that someone wants to deploy. This ability to move applications allows the applications to be packed tightly together so that the resources of the nodes can be utilized in the best possible way.

> **NOTE** In chapter 17 you'll learn more about how Kubernetes decides where to place each application and how you can influence the decision.

Finding optimal combinations can be challenging and time consuming, especially when the number of all possible options is huge, such as when you have many application components and many server nodes on which they can be deployed. Computers can perform this task much better and faster than humans. Kubernetes does it very well. By combining different applications on the same machines, Kubernetes improves the utilization of your hardware infrastructure so you can run more applications on fewer servers.

AUTOMATICALLY ADJUSTING TO CHANGING LOAD

Using Kubernetes to manage your deployed applications also means that the operations team doesn't have to constantly monitor the load of each application to respond to sudden load peaks. Kubernetes takes care of this also. It can monitor the resources consumed by each application and other metrics and adjust the number of running instances of each application to cope with increased load or resource usage.

When you run Kubernetes on cloud infrastructure, it can even increase the size of your cluster by provisioning additional nodes through the cloud provider's API. This way, you never run out of space to run additional instances of your applications.

KEEPING APPLICATIONS RUNNING SMOOTHLY

Kubernetes also makes every effort to ensure that your applications run smoothly. If your application crashes, Kubernetes will restart it automatically. So even if you have a broken application that runs out of memory after running for more than a few hours, Kubernetes will ensure that your application continues to provide the service to its users by automatically restarting it in this case.

Kubernetes is a self-healing system in that it deals with software errors like the one just described, but it also handles hardware failures. As clusters grow in size, the frequency of node failure also increases. For example, in a cluster with one hundred nodes and a MTBF

现在，即使他们不知道节点数量或每个节点的特征，也可以自己部署应用程序。

过去，系统管理员负责决定每个应用程序的放置位置。这个任务现在留给了 Kubernetes。这使得开发人员无需依赖其他人即可部署应用程序。当开发人员部署应用程序时，Kubernetes会根据应用程序的资源需求以及每个节点上的可用资源来选择运行应用程序的最佳节点。

通过更好地利用基础设施来降低成本

如果你不关心你的应用程序登陆哪个节点，这也意味着它可以随时移动到任何其他节点，而无需你担心。 Kubernetes 可能需要这样做，以便为某人想要部署的更大的应用程序腾出空间。这种移动应用程序的能力允许将应用程序紧密地打包在一起，以便可以以最佳方式利用节点的资源。

> 注意：在第 17 章中，您将详细了解 Kubernetes 如何决定每个应用程序的放置位置以及如何影响决策。

寻找最佳组合可能具有挑战性且耗时，特别是当所有可能选项的数量很大时，例如当您有许多应用程序组件和许多可以部署它们的服务器节点时。计算机可以比人类更好、更快地执行这项任务。 Kubernetes 在这方面做得非常好。通过在同一台机器上组合不同的应用程序，Kubernetes 提高了硬件基础设施的利用率，以便您可以在更少的服务器上运行更多的应用程序。

自动调整负载变化

使用 Kubernetes 来管理已部署的应用程序还意味着运维团队不必持续监控每个应用程序的负载以应对突然的负载峰值。 Kubernetes 也处理这个问题。它可以监控每个应用程序消耗的资源和其他指标，并调整每个应用程序的运行实例数量以应对增加的负载或资源使用情况。

当您在云基础设施上运行 Kubernetes 时，它甚至可以通过云提供商的 API 提供额外的节点来增加集群的大小。这样，您就不会耗尽空间来运行应用程序的其他实例。

保持应用程序平稳运行

Kubernetes 也尽一切努力确保您的应用程序顺利运行。如果您的应用程序崩溃，Kubernetes 将自动重新启动它。因此，即使您有一个损坏的应用程序，在运行几个小时以上后内存不足，Kubernetes 也会通过在这种情况下自动重新启动应用程序来确保您的应用程序继续为其用户提供服务。

Kubernetes 是一种自我修复系统，它可以处理如上所述的软件错误，但它也可以处理硬件故障。随着集群规模的增大，节点故障的频率也会增加。例如

(mean-time-between-failure) of 100 days for each node, you can expect one node to fail every day.

When a node fails, Kubernetes automatically moves applications to the remaining healthy nodes. The operations team no longer needs to manually move the application and can instead focus on repairing the node itself and returning it to the pool of available hardware resources.

If your infrastructure has enough free resources to allow normal system operation without the failed node, the operations team doesn't even have to react immediately to the failure. If it occurs in the middle of the night, no one from the operations team even has to wake up. They can sleep peacefully and deal with the failed node during regular working hours.

SIMPLIFYING APPLICATION DEVELOPMENT

The improvements described in the previous section mainly concern application deployment. But what about the process of application development? Does Kubernetes bring anything to their table? It definitely does.

As mentioned previously, Kubernetes offers infrastructure-related services that would otherwise have to be implemented in your applications. This includes the discovery of services and/or peers in a distributed application, leader election, centralized application configuration and others. Kubernetes provides this while keeping the application Kubernetes-agnostic, but when required, applications can also query the Kubernetes API to obtain detailed information about their environment. They can also use the API to change the environment.

1.2.3 The architecture of a Kubernetes cluster

As you've already learned, a Kubernetes cluster consists of nodes divided into two groups:

- A set of *master nodes* that host the *Control Plane* components, which are the brains of the system, since they control the entire cluster.
- A set of *worker nodes* that form the *Workload Plane*, which is where your workloads (or applications) run.

The following figure shows the two planes and the different nodes they consist of.

每个节点的（平均故障间隔时间）为 100 天，您可以预期每天有一个节点发生故障。

当某个节点发生故障时，Kubernetes 会自动将应用程序移动到剩余的健康节点。运营团队不再需要手动移动应用程序，而是可以专注于修复节点本身并将其返回到可用硬件资源池。

如果您的基础设施有足够的可用资源来允许系统在没有故障节点的情况下正常运行，那么运营团队甚至不必立即对故障做出反应。如果发生在半夜，运营团队的任何人都不必醒来。他们可以安心睡觉，并在正常工作时间处理故障节点。

简化应用程序开发

上一节中描述的改进主要涉及应用程序部署。但应用程序开发的过程又如何呢？ Kubernetes 能给他们带来什么吗？确实如此。

如前所述，Kubernetes 提供了与基础设施相关的服务，否则这些服务必须在您的应用程序中实现。这包括分布式应用程序中服务和/或对等点的发现、领导者选举、集中式应用程序配置等。 Kubernetes 提供了这一点，同时保持应用程序与 Kubernetes 无关，但在需要时，应用程序还可以查询 Kubernetes API 以获取有关其环境的详细信息。他们还可以使用 API 来更改环境。

1.2.3 Kubernetes集群的架构

正如您已经了解到的，Kubernetes 集群由分为两组的节点组成：

- 一组托管控制平面组件的主节点，它们是系统的大脑，因为它们控制整个集群。
- 形成工作负载平面的一组工作节点，这是您的工作负载（或应用程序）运行的位置。

下图显示了两个平面以及它们所组成的不同节点。

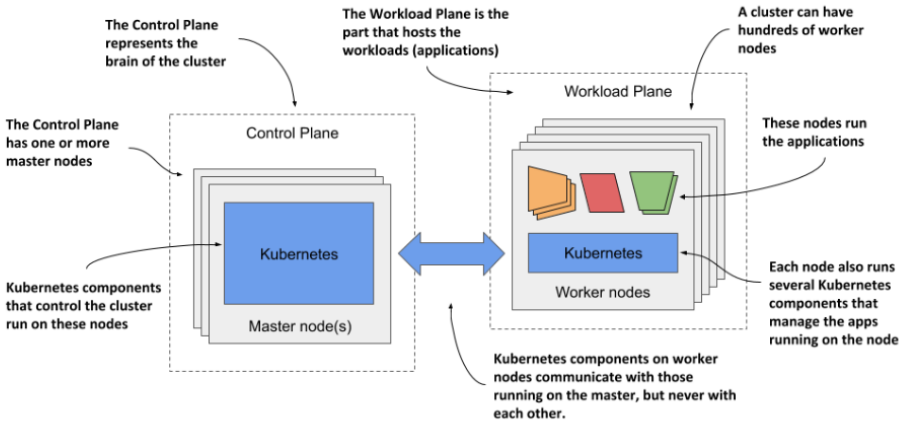


Figure 1.11 The two planes that make up a Kubernetes cluster

The two planes, and hence the two types of nodes, run different Kubernetes components. The next two sections of the book introduce them and summarize their functions without going into details. These components will be mentioned several times in the next part of the book where I explain the fundamental concepts of Kubernetes. An in-depth look at the components and their internals follows in the third part of the book.

CONTROL PLANE COMPONENTS

The Control Plane is what controls the cluster. It consists of several components that run on a single master node or are replicated across multiple master nodes to ensure high availability. The Control Plane's components are shown in the following figure.

图1.11 组成Kubernetes集群的两个平面

这两个平面以及两种类型的节点运行不同的 Kubernetes 组件。本书接下来的两节介绍它们并总结它们的功能，但不详细讨论。这些组件将在本书的下一部分中多次提到，我将在其中解释 Kubernetes 的基本概念。本书的第三部分将深入介绍这些组件及其内部结构。

控制平面组件

控制平面是控制集群的部分。它由多个组件组成，这些组件在单个主节点上运行或跨多个主节点复制以确保高可用性。控制平面的组件如下图所示。

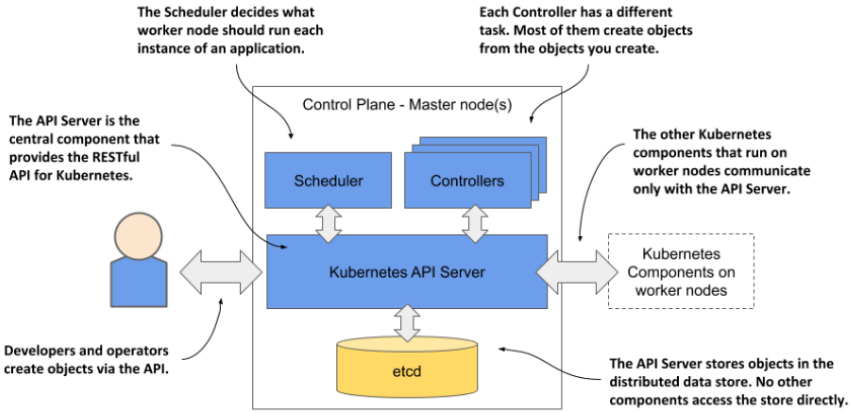


Figure 1.12 The components of the Kubernetes Control Plane

These are the components and their functions:

- The *Kubernetes API Server* exposes the RESTful Kubernetes API. Engineers using the cluster and other Kubernetes components create objects via this API.
- The *etcd* distributed datastore persists the objects you create through the API, since the API Server itself is stateless. The Server is the only component that talks to etcd.
- The *Scheduler* decides on which worker node each application instance should run.
- *Controllers* bring to life the objects you create through the API. Most of them simply create other objects, but some also communicate with external systems (for example, the cloud provider via its API).

The components of the Control Plane hold and control the state of the cluster, but they don't run your applications. This is done by the (worker) nodes.

WORKER NODE COMPONENTS

The worker nodes are the computers on which your applications run. They form the cluster's Workload Plane. In addition to applications, several Kubernetes components also run on these nodes. They perform the task of running, monitoring and providing connectivity between your applications. They are shown in the following figure.

图1.12 Kubernetes控制平面的组件

这些是组件及其功能：

- Kubernetes API 服务器公开 RESTful Kubernetes API。使用集群和其他 Kubernetes 组件的工程师通过此 API 创建对象。
- etcd 分布式数据存储会保留您通过 API 创建的对象，因为 API 服务器本身是无状态的。服务器是唯一与 etcd 通信的组件。
- 调度程序决定每个应用程序实例应在哪个工作节点上运行。
- 控制器使您通过 API 创建的对象栩栩如生。它们中的大多数只是创建其他对象，但有些还与外部系统进行通信（例如，云提供商通过其 API）。

控制平面的组件保存并控制集群的状态，但它们不运行您的应用程序。这是由（工作）节点完成的。

工作节点组件

工作节点是运行应用程序的计算机。它们形成集群的工作负载平面。除了应用程序之外，多个 Kubernetes 组件也在这些节点上运行。它们执行运行、监视和提供应用程序之间的连接的任务。它们如下图所示。

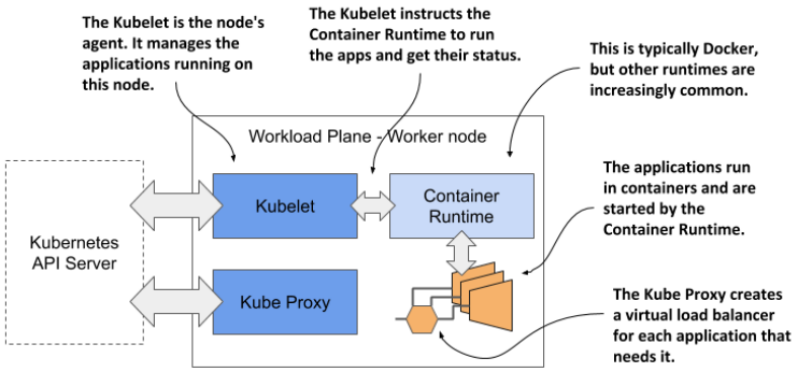


Figure 1.13 The Kubernetes components that run on each node

Each node runs the following set of components:

- The *Kubelet*, an agent that talks to the API server and manages the applications running on its node. It reports the status of these applications and the node via the API.
- The *Container Runtime*, which can be Docker or any other runtime compatible with Kubernetes. It runs your applications in containers as instructed by the Kubelet.
- The *Kubernetes Service Proxy (Kube Proxy)* load-balances network traffic between applications. Its name suggests that traffic flows through it, but that's no longer the case. You'll learn why in chapter 14.

ADD-ON COMPONENTS

Most Kubernetes clusters also contain several other components. This includes a DNS server, network plugins, logging agents and many others. They typically run on the worker nodes but can also be configured to run on the master.

GAINING A DEEPER UNDERSTANDING OF THE ARCHITECTURE

For now, I only expect you to be vaguely familiar with the names of these components and their function, as I'll mention them many times throughout the following chapters. You'll learn snippets about them in these chapters, but I'll explain them in more detail in chapter 14.

I'm not a fan of explaining how things work until I first explain *what* something does and teach you how to use it. It's like learning to drive. You don't want to know what's under the hood. At first, you just want to learn how to get from point A to B. Only then will you be interested in how the car makes this possible. Knowing what's under the hood may one day help you get your car moving again after it has broken down and you are stranded on the

了解引擎盖下的情况可能有一天会帮助您在汽车抛锚而被困在 15 号公路后重新启动。

图1.13 每个节点上运行的Kubernetes组件

每个节点运行以下组件集：

- Kubelet，一个与 API 服务器通信并管理在其节点上运行的应用程序的代理。它通过 API 报告这些应用程序和节点的状态。
- 容器运行时，可以是 Docker 或任何其他与 Kubernetes 兼容的运行时。它按照 Kubelet 的指示在容器中运行您的应用程序。
- Kubernetes 服务代理(Kube Proxy)对应用程序之间的网络流量进行负载平衡。它的名字表明交通流经它，但情况已不再如此。您将在第 14 章中了解原因。

附加组件

大多数 Kubernetes集群还包含其他几个组件。这包括 DNS 服务器、网络插件、日志代理等等。它们通常在工作节点上运行，但也可以配置为在主节点上运行。

更深入地了解架构

现在，我只希望您对这些组件的名称及其功能有一定的了解，因为我将在接下来的章节中多次提到它们。您将在这些章节中学习有关它们的片段，但我将在第 14 章中更详细地解释它们。

我不喜欢解释事物是如何工作的，直到我首先解释事物的作用并教你如何使用它。这就像学开车一样。你不想知道幕后是什么。一开始，您只想学习如何从 A 点到达 B 点。只有这样您才会对汽车如何实现这一点感兴趣

side of the road. I hate to say it, but you'll have many moments like this when dealing with Kubernetes due to its sheer complexity.

1.2.4 How Kubernetes runs an application

With a general overview of the components that make up Kubernetes, I can finally explain how to deploy an application in Kubernetes.

DEFINING YOUR APPLICATION

Everything in Kubernetes is represented by an object. You create and retrieve these objects via the Kubernetes API. Your application consists of several types of these objects - one type represents the application deployment as a whole, another represents a running instance of your application, another represents the service provided by a set of these instances and allows reaching them at a single IP address, and there are many others.

All these types are explained in detail in the second part of the book. At the moment, it's enough to know that you define your application through several types of objects. These objects are usually defined in one or more manifest files in either YAML or JSON format.

> **DEFINITION** YAML was initially said to mean "Yet Another Markup Language", but it was latter changed to the recursive acronym "YAML Ain't Markup Language". It's one of the ways to serialize an object into a human-readable text file.

> **DEFINITION** JSON is short for JavaScript Object Notation. It's a different way of serializing an object, but more suitable for exchanging data between applications.

The following figure shows an example of deploying an application by creating a manifest with two deployments exposed using two services.

路边。我不想这么说，但由于 Kubernetes的复杂性，在处理Kubernetes时你会遇到很多这样的时刻。

1.2.4 Kubernetes 如何运行应用程序

通过对构成 Kubernetes的组件的总体概述，我终于可以解释如何在 Kubernetes中部署应用程序。

定义您的应用程序

Kubernetes中的一切都由对象表示。您可以通过 Kubernetes API 创建和检索这些对象。您的应用程序由多种类型的对象组成 - 一种类型代表整个应用程序部署，另一种代表应用程序的运行实例，另一种代表由一组这些实例提供的服务，并允许通过单个 IP 地址访问它们，还有很多其他的。

所有这些类型都在本书的第二部分中详细解释。目前，只要知道您通过几种类型的对象定义应用程序就足够了。这些对象通常在一个或多个清单文件中定义，可以是 YAML 或 JSON 格式。

> 定义 YAML 最初被认为是 "Yet Another Markup Language"，但后来改为递归首字母缩写 "YAML Ain't Markup Language"。这是将对象序列化为对象的方法之一，可通过人类阅读的文本文件。

> 定义 JSON 是 JavaScript Object Notation 的缩写。这是序列化对象的不同方式，但更适合在应用程序之间交换数据。

下图显示了通过创建清单来部署应用程序的示例，该清单包含使用两项服务公开的两个部署。

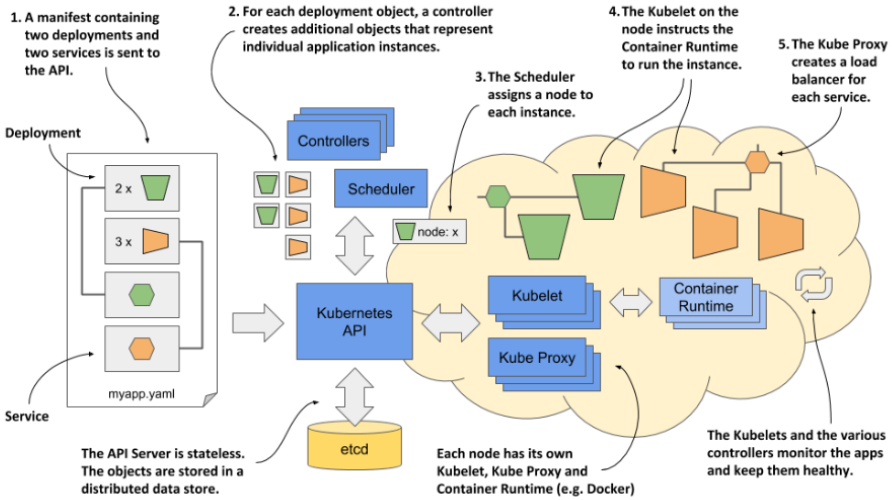


Figure 1.14 Deploying an application to Kubernetes

These actions take place when you deploy the application:

1. You submit the application manifest to the Kubernetes API. The API Server writes the objects defined in the manifest to etcd.
2. A controller notices the newly created objects and creates several new objects - one for each application instance.
3. The Scheduler assigns a node to each instance.
4. The Kubelet notices that an instance is assigned to the Kubelet's node. It runs the application instance via the Container Runtime.
5. The Kube Proxy notices that the application instances are ready to accept connections from clients and configures a load balancer for them.
6. The Kubelets and the Controllers monitor the system and keep the applications running.

The procedure is explained in more detail in the following sections, but the complete explanation is given in chapter 14, after you have familiarized yourself with all the objects and controllers involved.

17

图1.14 将应用程序部署到Kubernetes

这些操作在您部署应用程序时发生：

1. 将应用程序清单提交到 KubernetesAPI。 API Server将清单中定义的对象写入 etcd。
2. 控制器注意到新创建的对象并创建多个新对象 - 每个应用程序实例一个。
3. Schedule为每个实例分配一个节点。
4. Kubelet注意到一个实例已分配给 Kubelet的节点。它通过容器运行时运行应用程序实例。
5. Kube Proxy注意到应用程序实例已准备好接受来自客户端的连接，并为其配置负载均衡器。
6. Kubelet和控制器监控系统并保持应用程序运行。

以下各节将更详细地解释该过程，但在您熟悉所有涉及的对象和控制器后，第 14 章将给出完整的解释。

Submitting the application to the API

After you've created your YAML or JSON file(s), you submit the file to the API, usually via the Kubernetes command-line tool called *kubectl*.

> **NOTE** Kubectl is pronounced *kube-control*, but the softer souls in the community prefer to call it *kube-cuddle*. Some refer to it as *kube-C-T-L*.

Kubectl splits the file into individual objects and creates each of them by sending an HTTP PUT or POST request to the API, as is usually the case with RESTful APIs. The API Server validates the objects and stores them in the etcd datastore. In addition, it notifies all interested components that these objects have been created. Controllers, which are explained next, are one of these components.

About the controllers

Most object types have an associated controller. A controller is interested in a particular object type. It waits for the API server to notify it that a new object has been created, and then performs operations to bring that object to life. Typically, the controller just creates other objects via the same Kubernetes API. For example, the controller responsible for application deployments creates one or more objects that represent individual instances of the application. The number of objects created by the controller depends on the number of replicas specified in the application deployment object.

About the Scheduler

The scheduler is a special type of controller, whose only task is to schedule application instances onto worker nodes. It selects the best worker node for each new application instance object and assigns it to the instance - by modifying the object via the API.

About the Kubelet and the Container Runtime

The Kubelet that runs on each worker node is also a type of controller. Its task is to wait for application instances to be assigned to the node on which it is located and run the application. This is done by instructing the Container Runtime to start the application's container.

About the Kube Proxy

Because an application deployment can consist of multiple application instances, a load balancer is required to expose them at a single IP address. The Kube Proxy, another controller running alongside the Kubelet, is responsible for setting up the load balancer.

Keeping the applications healthy

Once the application is up and running, the Kubelet keeps the application healthy by restarting it when it terminates. It also reports the status of the application by updating the object that represents the application instance. The other controllers monitor these objects and ensure that applications are moved to healthy nodes if their nodes fail.

You're now roughly familiar with the architecture and functionality of Kubernetes. You don't need to understand or remember all the details at this moment, because internalizing

向 API 提交申请

创建 YAML 或 JSON 文件后，通常通过名为 kubectl 的 Kubernetes命令行工具将文件提交到 API。

> 注意：Kubectl 的发音是 kube-control，但社区中比较温和的人更喜欢称其为 kube-偎。有些人将其称为 kube-C-T-L。

Kubectl 将文件拆分为单独的对象，并通过向 API 发送 HTTP PUT 或 POST 请求来创建每个对象，这通常是 RESTfulAPI 的情况。 API 服务器验证对象并将它们存储在 etcd 数据存储中。此外，它还通知所有感兴趣的组件这些对象已被创建。接下来解释的控制器就是这些组件之一。

关于控制器

大多数对象类型都有一个关联的控制器。控制器对特定的对象类型感兴趣。它等待 API 服务器通知它已创建新对象，然后执行操作以便该对象生效。通常，控制器只是通过相同的 KubernetesAPI 创建其他对象。例如，负责应用程序部署的控制器创建一个或多个代表应用程序的各个实例的对象。控制器创建的对象数量取决于应用程序部署对象中指定的副本数量。

关于调度程序

调度程序是一种特殊类型的控制器，其唯一任务是将应用程序实例调度到工作节点上。它为每个新的应用程序实例对象选择最佳的工作节点，并将其分配给实例 - 通过 API 修改对象。

关于 kubelet 和容器运行时

每个工作节点上运行的 Kubelet 也是一种控制器。它的任务是等待应用程序实例被分配到它所在的节点并运行应用程序。这是通过指示容器运行时启动应用程序的容器来完成的。

关于 KUBE 代理

由于应用程序部署可能包含多个应用程序实例，因此需要负载均衡器将它们公开在单个 IP 地址上。 Kube Proxy 是与 Kubelet 一起运行的另一个控制器，负责设置负载均衡器。

保持应用程序健康

应用程序启动并运行后，Kubelet 通过在应用程序终止时重新启动应用程序来保持应用程序的健康。它还通过更新表示应用程序实例的对象来报告应用程序的状态。其他控制器监视这些对象，并确保在节点出现故障时将应用程序移至健康节点。

现在您已经大致熟悉了 Kubernetes的架构和功能。此时你不需要理解或记住所有细节

this information will be easier when you learn about each individual object types and the controllers that bring them to life in the second part of the book.

1.3 Introducing Kubernetes into your organization

To close this chapter, let's see what options are available to you if you decide to introduce Kubernetes in your own IT environment.

1.3.1 Running Kubernetes on-premises and in the cloud

If you want to run your applications on Kubernetes, you have to decide whether you want to run them locally, in your organization's own infrastructure (on-premises) or with one of the major cloud providers, or perhaps both - in a hybrid cloud solution.

RUNNING KUBERNETES ON-PREMISES

Running Kubernetes on your own infrastructure may be your only option if regulations require you to run applications on site. This usually means that you'll have to manage Kubernetes yourself, but we'll come to that later.

Kubernetes can run directly on your bare-metal machines or in virtual machines running in your data center. In either case, you won't be able to scale your cluster as easily as when you run it in virtual machines provided by a cloud provider.

DEPLOYING KUBERNETES IN THE CLOUD

If you have no on-premises infrastructure, you have no choice but to run Kubernetes in the cloud. This has the advantage that you can scale your cluster at any time at short notice if required. As mentioned earlier, Kubernetes itself can ask the cloud provider to provision additional virtual machines when the current size of the cluster is no longer sufficient to run all the applications you want to deploy.

When the number of workloads decreases and some worker nodes are left without running workloads, Kubernetes can ask the cloud provider to destroy the virtual machines of these nodes to reduce your operational costs. This elasticity of the cluster is certainly one of the main benefits of running Kubernetes in the cloud.

USING A HYBRID CLOUD SOLUTION

A more complex option is to run Kubernetes on-premises, but also allow it to spill over into the cloud. It's possible to configure Kubernetes to provision additional nodes in the cloud if you exceed the capacity of your own data center. This way, you get the best of both worlds. Most of the time, your applications run locally without the cost of virtual machine rental, but in short periods of peak load that may occur only a few times a year, your applications can handle the extra load by using the additional resources in the cloud.

If your use-case requires it, you can also run a Kubernetes cluster across multiple cloud providers or a combination of any of the options mentioned. This can be done using a single control plane or one control plane in each location.

当您在本书的第二部分中了解每个单独的对象类型以及使它们栩栩如生的控制器时，这些信息会更容易。

1.3 将 Kubernetes 引入您的组织

作为本章的结束语，我们来看看如果您决定在自己的 IT 环境中引入 Kubernetes，您可以选择哪些选项。

1.3.1 在本地和云端运行 Kubernetes

如果您想在 Kubernetes上运行应用程序，您必须决定是否要在本地运行它们、在组织自己的基础设施（本地）中运行它们，还是与主要云提供商之一一起运行，或者两者都在混合云解决方案中运行。

在本地运行 Kubernetes

如果法规要求您在现场运行应用程序，在您自己的基础设施上运行 Kubernetes可能是您唯一的选择。这通常意味着您必须自己管理 Kubernetes，但我们稍后会讨论这个问题。

Kubernetes可以直接在裸机上运行，也可以在数据中心运行的虚拟机上运行。无论哪种情况，您都无法像在云提供商提供的虚拟机中运行集群那样轻松地扩展集群。

在云中部署 Kubernetes

如果您没有本地基础设施，则别无选择，只能在云中运行 Kubernetes。这样做的优点是，如果需要，您可以在短时间内随时扩展集群。如前所述，当集群的当前大小不再足以运行您想要部署的所有应用程序时，Kubernetes本身可以要求云提供商提供额外的虚拟机。

当工作负载数量减少并且某些工作节点没有运行工作负载时，Kubernetes可以要求云提供商销毁这些节点的虚拟机，以降低您的运营成本。集群的这种弹性无疑是在云中运行 Kubernetes的主要好处之一。

使用混合云解决方案

一个更复杂的选择是在本地运行 Kubernetes，但也允许它溢出到云中。如果您超出了自己的数据中心的容量，可以配置Kubernetes在云中提供额外的节点。这样，您就可以两全其美。大多数时候，您的应用程序在本地运行，无需虚拟机租赁成本，但在一年可能只出现几次的短期峰值负载中，您的应用程序可以通过使用云中的额外资源来处理额外负载。

如果您的用例需要，您还可以跨多个云提供商或上述任何选项的组合运行 Kubernetes集群。这可以使用单个控制平面或每个位置的一个控制平面来完成。

1.3.2 To manage or not to manage Kubernetes yourself

If you are considering introducing Kubernetes in your organization, the most important question you need to answer is whether you'll manage Kubernetes yourself or use a Kubernetes-as-a-Service type offering where someone else manages it for you.

MANAGING KUBERNETES YOURSELF

If you already run applications on-premises and have enough hardware to run a production-ready Kubernetes cluster, your first instinct is probably to deploy and manage it yourself. If you ask anyone in the Kubernetes community if this is a good idea, you'll usually get a very definite "no".

Figure 1.14 was a very simplified representation of what happens in a Kubernetes cluster when you deploy an application. Even that figure should have scared you. Kubernetes brings with it an enormous amount of additional complexity. Anyone who wants to run a Kubernetes cluster must be intimately familiar with its inner workings.

The management of production-ready Kubernetes clusters is a multi-billion-dollar industry. Before you decide to manage one yourself, it's essential that you consult with engineers who have already done it to learn about the issues most teams run into. If you don't, you may be setting yourself up for failure. On the other hand, trying out Kubernetes for non-production use-cases or using a managed Kubernetes cluster is much less problematic.

USING A MANAGED KUBERNETES CLUSTER IN THE CLOUD

Using Kubernetes is ten times easier than managing it. Most major cloud providers now offer Kubernetes-as-a-Service. They take care of managing Kubernetes and its components while you simply use the Kubernetes API like any of the other APIs the cloud provider offers.

The top managed Kubernetes offerings include the following:

- Google Kubernetes Engine (GKE)
- Azure Kubernetes Service (AKS)
- Amazon Elastic Kubernetes Service (EKS)
- IBM Cloud Kubernetes Service
- Red Hat OpenShift Online and Dedicated
- VMware Cloud PKS
- Alibaba Cloud Container Service for Kubernetes (ACK)

The first half of this book focuses on just using Kubernetes. You'll run the exercises in a local development cluster and on a managed GKE cluster, as I find it's the easiest to use and offers the best user experience. The second part of the book gives you a solid foundation for managing Kubernetes, but to truly master it, you'll need to gain additional experience.

1.3.3 Using vanilla or extended Kubernetes

The final question is whether to use a vanilla open-source version of Kubernetes or an extended, enterprise-quality Kubernetes product.

1.3.2 自己管理或不管理Kubernetes

如果您正在考虑在您的组织中引入 Kubernetes，那么您需要回答的最重要的问题是您是否会自己管理 Kubernetes，还是使用由其他人为您管理的 Kubernetes即服务类型产品。

自行管理 Kubernetes

如果您已经在本地运行应用程序并且拥有足够的硬件来运行生产就绪的 Kubernetes集群，您的第一反应可能是自行部署和管理它。如果你问 Kubernetes社区中的任何人这是否是一个好主意，你通常会得到一个非常明确的"不"。

图 1.14非常简化地展示了部署应用程序时 Kubernetes集群中发生的情况。即使是这个数字也应该让你感到害怕。 Kubernetes带来了大量额外的复杂性。任何想要运行 Kubernetes集群的人都必须非常熟悉其内部工作原理。生产就绪 Kubernetes集群的管理是

价值数十亿美元的

行业。在决定自己管理之前，有必要咨询已经做过管理的工程师，以了解大多数团队遇到的问题。如果你不这样做，你可能会让自己陷入失败。另一方面，尝试将 Kubernetes用于非生产用例或使用

托管 Kubernetes集群要少得多

有问题的。

在云中使用托管 Kubernetes 集群

使用 Kubernetes比管理它容易十倍。大多数主要云提供商现在都提供 Kubernetes即服务。他们负责管理 Kubernetes及其组件，而您只需像云提供商提供的任何其他 API 一样使用 KubernetesAPI。顶级托管 Kubernetes产品包括以下内容：

- 谷歌 Kubernetes 引擎 (GKE)
- Azure Kubernetes 服务 (AKS)
- Amazon Elastic Kubernetes 服务 (EKS)
- IBM Cloud Kubernetes 服务
- 红帽 OpenShift 在线版和专用版
- VMware 云 PKS

本书的前半部分重点介绍 Kubernetes的使用。您将在本地开发集群和托管 GKE集群上运行练习，因为我发现它最容易使用并提供最佳的用户体验。本书的第二部分为您提供了管理 Kubernetes的坚实基础，但要真正掌握它，您需要获得额外的经验。

1.3.3 使用普通或扩展 Kubernetes

最后一个问题是是否使用 Kubernetes的普通开源版本或扩展的企业级 Kubernetes产品。

Using a vanilla version of Kubernetes

The open-source version of Kubernetes is maintained by the community and represents the cutting edge of Kubernetes development. This also means that it may not be as stable as the other options. It may also lack good security defaults. Deploying the vanilla version requires a lot of fine tuning to set everything up for production use.

Using enterprise-grade Kubernetes distributions

A better option for using Kubernetes in production is to use an enterprise-quality Kubernetes distribution such as OpenShift or Rancher. In addition to the increased security and performance provided by better defaults, they offer additional object types in addition to those provided in the upstream Kubernetes API. For example, vanilla Kubernetes does not contain object types that represent cluster users, whereas commercial distributions do. They also provide additional software tools for deploying and managing well-known third-party applications on Kubernetes.

Of course, extending and hardening Kubernetes takes time, so these commercial Kubernetes distributions usually lag one or two versions behind the upstream version of Kubernetes. It's not as bad as it sounds. The benefits usually outweigh the disadvantages.

1.3.4 Should you even use Kubernetes?

I hope this chapter has made you excited about Kubernetes and you can't wait to squeeze it into your IT stack. But to close this chapter properly, we need to say a word or two about when introducing Kubernetes is not a good idea.

Do your workloads require automated management?

The first thing you need to be honest about is whether you need to automate the management of your applications at all. If your application is a large monolith, you definitely don't need Kubernetes.

Even if you deploy microservices, using Kubernetes may not be the best option, especially if the number of your microservices is very small. It's difficult to provide an exact number when the scales tip over, since other factors also influence the decision. But if your system consists of less than five microservices, throwing Kubernetes into the mix is probably not a good idea. If your system has more than twenty microservices, you will most likely benefit from the integration of Kubernetes. If the number of your microservices falls somewhere in between, other factors, such as the ones described next, should be considered.

Can you afford to invest your engineers' time into learning Kubernetes?

Kubernetes is designed to allow applications to run without them knowing that they are running in Kubernetes. While the applications themselves don't need to be modified to run in Kubernetes, development engineers will inevitably spend a lot of time learning how to use Kubernetes, even though the operators are the only ones that actually need that knowledge.

It would be hard to tell your teams that you're switching to Kubernetes and expect only the operations team to start exploring it. Developers like shiny new things. At the time of writing, Kubernetes is still a very shiny thing.

使用 Kubernetes 的普通版本

Kubernetes的开源版本由社区维护，代表了 Kubernetes开发的最前沿。这也意味着它可能不如其他选项那么稳定。它还可能缺乏良好的安全默认设置。部署普通版本需要进行大量微调才能将所有内容设置为生产使用。

使用企业级 Kubernetes 发行版

在生产中使用 Kubernetes的更好选择是使用企业级 Kubernetes发行版，例如 OpenShift 或 Rancher。除了通过更好的默认设置提供更高的安全性和性能之外，除了上游 KubernetesAPI 中提供的对象类型之外，它们还提供其他对象类型。例如，vanillaKubernetes不包含代表集群用户的对象类型，而商业发行版则包含。他们还提供了额外的软件工具，用于在 Kubernetes上部署和管理知名的第三方应用程序。

当然，扩展和强化 Kubernetes需要时间，因此这些商业 Kubernetes发行版通常落后 Kubernetes上游版本一两个版本。这并不像听起来那么糟糕。好处通常大于坏处。

1.3.4 你是否应该使用 Kubernetes?

我希望本章能让您对 Kubernetes感到兴奋，并迫不及待地想将其融入您的 IT 堆栈中。但为了正确地结束本章，我们需要说一两句话，说明什么时候引入 Kubernetes不是一个好主意。

您的工作负载需要自动化管理吗?

您需要诚实的第一件事是您是否需要自动化应用程序的管理。如果您的应用程序是一个大型整体，那么您绝对不需要 Kubernetes。

即使您部署微服务，使用 Kubernetes也可能不是最佳选择，尤其是当您的微服务数量非常小时。当天平倾斜时，很难提供准确的数字，因为其他因素也会影响决策。但如果您的系统由少于五个微服务组成，那么将 Kubernetes加入其中可能不是一个好主意。如果您的系统有超过二十个微服务，您很可能会从 Kubernetes的集成中受益。如果微服务的数量介于两者之间，则应考虑其他因素，例如接下来描述的因素。

您有能力投入工程师的时间来学习 Kubernetes 吗?

Kubernetes旨在允许应用程序在不知道它们正在 Kubernetes中运行的情况下运行。虽然应用程序本身不需要修改即可在 Kubernetes中运行，但开发工程师将不可避免地花费大量时间学习如何使用 Kubernetes。尽管运维人员是唯一真正需要这些知识的人。

很难告诉您的团队您正在转向 Kubernetes，并期望只有运营团队开始探索它。开发人员喜欢闪亮的新事物。在撰写本文时，Kubernetes仍然是一个非常闪亮的东西。

ARE YOU PREPARED FOR INCREASED COSTS IN THE INTERIM?

While Kubernetes reduces long-term operational costs, introducing Kubernetes in your organization initially involves increased costs for training, hiring new engineers, building and purchasing new tools and possibly additional hardware. Kubernetes requires additional computing resources in addition to the resources that the applications use.

DON'T BELIEVE THE HYPE

Although Kubernetes has been around for several years at the time of writing this book, I can't say that the hype phase is over. The initial excitement has just begun to calm down, but many engineers may still be unable to make rational decisions about whether the integration of Kubernetes is as necessary as it seems.

1.4 Summary

In this introductory chapter, you've learned that:

- Kubernetes is Greek for helmsman. As a ship's captain oversees the ship while the helmsman steers it, you oversee your computer cluster, while Kubernetes performs the day-to-day management tasks.
- Kubernetes is pronounced *koo-ber-netties*. Kubectl, the Kubernetes command-line tool, is pronounced *kube-control*.
- Kubernetes is an open-source project built upon Google's vast experience in running applications on a global scale. Thousands of individuals now contribute to it.
- Kubernetes uses a declarative model to describe application deployments. After you provide a description of your application to Kubernetes, it brings it to life.
- Kubernetes is like an operating system for the cluster. It abstracts the infrastructure and presents all computers in a data center as one large, contiguous deployment area.
- Microservice-based applications are more difficult to manage than monolithic applications. The more microservices you have, the more you need to automate their management with a system like Kubernetes.
- Kubernetes helps both development and operations teams to do what they do best. It frees them from mundane tasks and introduces a standard way of deploying applications both on-premises and in any cloud.
- Using Kubernetes allows developers to deploy applications without the help of system administrators. It reduces operational costs through better utilization of existing hardware, automatically adjusts your system to load fluctuations, and heals itself and the applications running on it.
- A Kubernetes cluster consists of master and worker nodes. The master nodes run the *Control Plane*, which controls the entire cluster, while the worker nodes run the deployed applications or workloads, and therefore represent the *Workload Plane*.
- Using Kubernetes is simple, but managing it is hard. An inexperienced team should use a Kubernetes-as-a-Service offering instead of deploying Kubernetes by itself.

So far, you've only observed the ship from the pier. It's time to come aboard. But before you leave the docks, you should inspect the shipping containers it's carrying. You'll do this next.

您准备好应对在此期间增加的成本了吗？

虽然 Kubernetes降低了长期运营成本，但在组织中引入 Kubernetes最初会增加培训、雇用新工程师、构建和购买新工具以及可能的额外硬件的成本。除了应用程序使用的资源之外，Kubernetes还需要额外的计算资源。

不要相信炒作

尽管在撰写本书时 Kubernetes已经存在好几年了，但我不能说炒作阶段已经结束。最初的兴奋才刚刚开始平静下来，但很多工程师可能仍然无法对 Kubernetes的集成是否像看起来那样有必要做出理性的决定。

1.4 总结

在本介绍性章节中，您已经了解到：

- Kubernetes 在希腊语中是舵手的意思。当船长负责监督船舶而舵手负责驾驶时，您负责监督计算机集群，而 Kubernetes 则负责执行日常管理任务。
- Kubernetes 发音为koo-ber-netties。 Kubectl 是 Kubernetes 命令行工具，发音为 kube-control。
- Kubernetes 是一个开源项目，基于 Google 在全球范围内运行应用程序的丰富经验。现在有成千上万的人为此做出贡献。
- Kubernetes 使用声明性模型来描述应用程序部署。当您向 Kubernetes 提供应用程序的描述后，它就会变得栩栩如生。
- Kubernetes 就像集群的操作系统。它将基础设施抽象化，并将数据中心中的所有计算机呈现为一个大型、连续的部署区域。
- 基于微服务的应用程序比单体应用程序更难管理应用程序。拥有的微服务越多，就越需要使用 Kubernetes这样的系统来自动化管理它们。
- Kubernetes 帮助开发和运营团队做他们最擅长的事情。它将他们从平凡的任务中解放出来，并引入了在本地和任何云中部署应用程序的标准方法。
- 使用Kubernetes，开发人员无需系统管理员的帮助即可部署应用程序。它通过更好地利用现有硬件来降低运营成本，自动调整系统以适应负载波动，并修复自身及其上运行的应用程序。
- Kubernetes 集群由主节点和工作节点组成。主节点运行控制平面，控制整个集群，而工作节点运行已部署的应用程序或工作负载，因此代表工作负载平面。
- 使用Kubernetes 很简单，但管理它很困难。缺乏经验的团队应该使用 Kubernetes 即服务产品，而不是自行部署 Kubernetes。

到目前为止，您仅从码头观察了这艘船。是时候上船了。但在离开码头之前，您应该检查其携带的集装箱。接下来你将执行此操作。

2

Understanding containers

This chapter covers

- **Understanding what a container is**
- **Differences between containers and virtual machines**
- **Creating, running, and sharing a container image with Docker**
- **Linux kernel features that make containers possible**

Kubernetes primarily manages applications that run in containers - so before you start exploring Kubernetes, you need to have a good understanding of what a container is. This chapter explains the basics of Linux containers that a typical Kubernetes user needs to know.

2.1 Introducing containers

In Chapter 1 you learned how different microservices running in the same operating system may require different, potentially conflicting versions of dynamically linked libraries or have different environment requirements.

When a system consists of a small number of applications, it's okay to assign a dedicated virtual machine to each application and run each in its own operating system. But as the microservices become smaller and their numbers start to grow, you may not be able to afford to give each one its own VM if you want to keep your hardware costs low and not waste resources.

It's not just a matter of wasting hardware resources - each VM typically needs to be individually configured and managed, which means that running higher numbers of VMs also results in higher staffing requirements and the need for a better, often more complicated automation system. Due to the shift to microservice architectures, where systems consist of hundreds of deployed application instances, an alternative to VMs was needed. Containers are that alternative.

2

了解容器

本章涵盖

- 了解什么是容器
- 容器和虚拟机的区别
- 使用 Docker 创建、运行和共享容器映像
- 使容器成为可能的 Linux 内核功能

Kubernetes主要管理在容器中运行的应用程序 - 因此在开始探索 Kubernetes之前，您需要充分了解容器是什么。本章介绍了典型 Kubernetes用户需要了解的 Linux容器基础知识。

2.1 容器介绍

在第 1 章中，您了解了在同一操作系统中运行的不同微服务可能需要不同的、可能存在冲突的动态链接库版本，或者具有不同的环境要求。

当系统由少量应用程序组成时，可以为每个应用程序分配一个专用虚拟机并在其自己的操作系统中运行每个应用程序。但随着微服务变得越来越小并且数量开始增长，如果您想保持较低的硬件成本并且不浪费资源，您可能无法为每个微服务提供自己的虚拟机。

这不仅仅是浪费硬件资源的问题，每个虚拟机通常都需要单独配置和管理，这意味着运行更多数量的虚拟机也会导致更高的人员配置要求，并且需要更好、通常更复杂的自动化系统。由于向微服务架构的转变，其中系统由数百个已部署的应用程序实例组成，因此需要虚拟机的替代方案。容器就是这样的选择。

2.1.1 Comparing containers to virtual machines

Instead of using virtual machines to isolate the environments of individual microservices (or software processes in general), most development and operations teams now prefer to use containers. They allow you to run multiple services on the same host computer, while keeping them isolated from each other. Like VMs, but with much less overhead.

Unlike VMs, which each run a separate operating system with several system processes, a process running in a container runs within the existing host operating system. Because there is only one operating system, no duplicate system processes exist. Although all the application processes run in the same operating system, their environments are isolated, though not as well as when you run them in separate VMs. To the process in the container, this isolation makes it look like no other processes exist on the computer. You'll learn how this is possible in the next few sections, but first let's dive deeper into the differences between containers and virtual machines.

COMPARING THE OVERHEAD OF CONTAINERS AND VIRTUAL MACHINES

Compared to VMs, containers are much lighter, because they don't require a separate resource pool or any additional OS-level processes. While each VM usually runs its own set of system processes, which requires additional computing resources in addition to those consumed by the user application's own process, a container is nothing more than an isolated process running in the existing host OS that consumes only the resources the app consumes. They have virtually no overhead.

Figure 2.1 shows two bare metal computers, one running two virtual machines, and the other running containers instead. The latter has space for additional containers, as it runs only one operating system, while the first runs three – one host and two guest OSes.

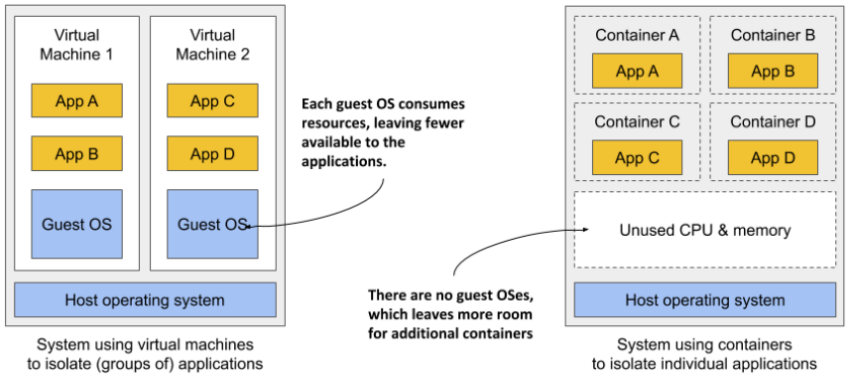


Figure 2.1 Using VMs to isolate groups of applications vs. isolating individual apps with containers

2.1.1 容器与虚拟机的比较

大多数开发和运营团队现在更喜欢使用容器，而不是使用虚拟机来隔离各个微服务（或一般软件流程）的环境。它们允许您在同一台主机上运行多个服务，同时使它们彼此隔离。与虚拟机类似，但开销要少得多。

与虚拟机不同的是，每个虚拟机都运行一个带有多个系统进程的单独操作系统，而容器中运行的进程则在现有主机操作系统中运行。由于只有一个操作系统，因此不存在重复的系统进程。尽管所有应用程序进程都在同一操作系统中运行，但它们的环境是隔离的，尽管不如在单独的虚拟机中运行它们时那样隔离。对于容器中的进程来说，这种隔离使得计算机上看起来就像不存在其他进程一样。您将在接下来的几节中了解这是如何实现的，但首先让我们更深入地了解容器和虚拟机之间的差异。

比较容器和虚拟机的开销

与虚拟机相比，容器要轻得多，因为它们不需要单独的资源池或任何额外的操作系统级进程。虽然每个虚拟机通常运行自己的一组系统进程，除了用户应用程序自己的进程消耗的资源之外，这还需要额外的计算资源，但容器只不过是在现有主机操作系统中运行的一个隔离进程，仅消耗应用程序所占用的资源。应用程序消耗。他们几乎没有任何开销。

图 2.1显示了两台裸机计算机，一台运行两个虚拟机，另一台运行容器。后者有空间容纳额外的容器，因为它只运行一个操作系统，而第一个运行三个操作系统——一个主机和两个客户操作系统。

图 2.1 使用虚拟机隔离应用程序组与

Because of the resource overhead of VMs, you often group multiple applications into each VM. You can't afford to dedicate a whole VM to each app. But containers introduce no overhead, which means you can afford to create a separate container for each application. In fact, you should never run multiple applications in the same container, as this makes managing the processes in the container much more difficult. Moreover, all existing software dealing with containers, including Kubernetes itself, is designed under the premise that there's only one application in a container.

COMPARING THE START-UP TIME OF CONTAINERS AND VIRTUAL MACHINES

In addition to the lower runtime overhead, containers also start the application faster, because only the application process itself needs to be started. No additional system processes need to be started first, as is the case when booting up a new virtual machine.

COMPARING THE ISOLATION OF CONTAINERS AND VIRTUAL MACHINES

You'll agree that containers are clearly better when it comes to the use of resources, but there's also a disadvantage. When you run applications in virtual machines, each VM runs its own operating system and kernel. Underneath those VMs is the hypervisor (and possibly an additional operating system), which splits the physical hardware resources into smaller sets of virtual resources that the operating system in each VM can use. As figure 2.2 shows, applications running in these VMs make system calls (*sys-calls*) to the guest OS kernel in the VM, and the machine instructions that the kernel then executes on the virtual CPUs are then forwarded to the host's physical CPU via the hypervisor.

25

由于虚拟机的资源开销，您通常会将多个应用程序分组到每个虚拟机中。您无法将整个虚拟机专用于每个应用程序。但容器不会带来任何开销，这意味着您可以为每个应用程序创建单独的容器。事实上，您永远不应该在同一个容器中运行多个应用程序，因为这会使管理容器中的进程变得更加困难。而且，现有的所有涉及容器的软件，包括 Kubernetes本身，都是在容器中只有一个应用程序的前提下设计的。

容器和虚拟机的启动时间比较

除了运行时开销较低之外，容器还可以更快地启动应用程序，因为只需要启动应用程序进程本身。不需要像启动新虚拟机时那样首先启动其他系统进程。

容器和虚拟机的隔离性比较

您会同意，在资源使用方面，容器显然更好，但也有一个缺点。当您在虚拟机中运行应用程序时，每个虚拟机都运行自己的操作系统和内核。这些虚拟机下面是虚拟机管理程序（可能还包括一个附加操作系统），它将物理硬件资源分割成每个虚拟机中的操作系统可以使用的更小的虚拟资源集。如图2.2所示，这些VM 中运行的应用程序对VM 中的客户操作系统内核进行系统调用（sys-calls），然后内核在虚拟CPU上执行的机器指令通过管理程序。

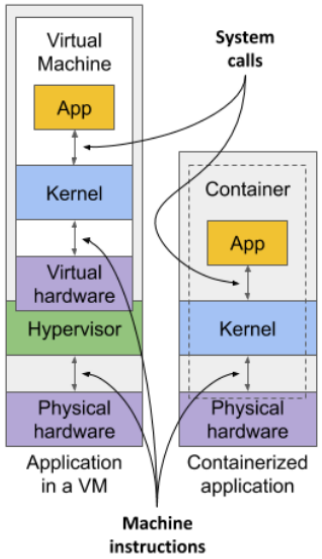


Figure 2.2 How apps use the hardware when running in a VM vs. in a container

NOTE Two types of hypervisors exist. Type 1 hypervisors don't require running a host OS, while type 2 hypervisors do.

Containers, on the other hand, all make system calls on the single kernel running in the host OS. This single kernel is the only one that executes instructions on the host's CPU. The CPU doesn't need to handle any kind of virtualization the way it does with VMs.

Examine figure 2.3 to see the difference between running three applications on bare metal, running them in two separate virtual machines, or running them in three containers.

26

图 2.2 应用程序在虚拟机和容器中运行时如何使用硬件

注意 存在两种类型的虚拟机管理程序。类型 1 虚拟机管理程序不需要运行主机操作系统，而类型 2 虚拟机管理程序可以。

另一方面，容器都在主机操作系统中运行的单个内核上进行系统调用。这个单一内核是唯一在主机 CPU 上执行指令的内核。 CPU 不需要像处理 VM 那样处理任何类型的虚拟化。

检查图 2.3，了解在裸机上运行三个应用程序、在两个单独的虚拟机中运行它们或在三个容器中运行它们之间的区别。

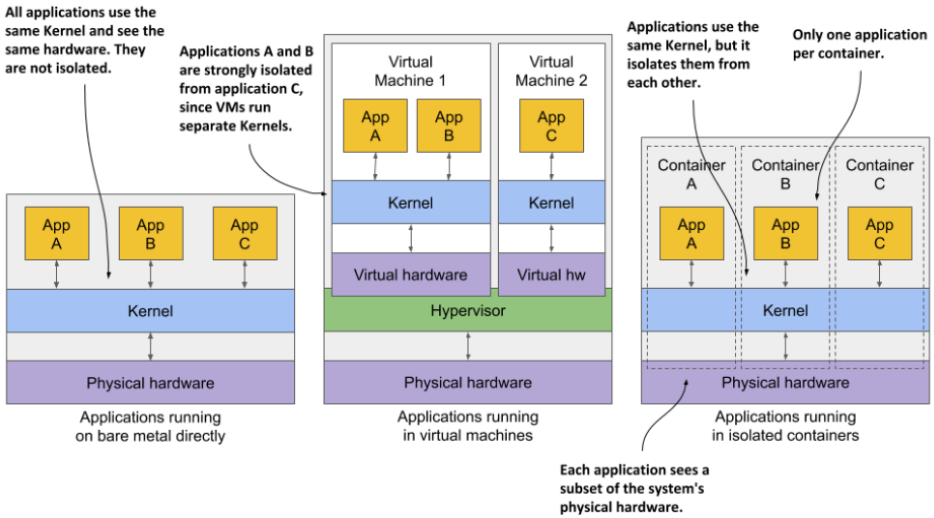


Figure 2.3 The difference between running applications on bare metal, in virtual machines, and in containers

In the first case, all three applications use the same kernel and aren't isolated at all. In the second case, applications A and B run in the same VM and thus share the kernel, while application C is completely isolated from the other two, since it uses its own kernel. It only shares the hardware with the first two.

The third case shows the same three applications running in containers. Although they all use the same kernel, they are isolated from each other and completely unaware of the others' existence. The isolation is provided by the kernel itself. Each application sees only a part of the physical hardware and sees itself as the only process running in the OS, although they all run in the same OS.

UNDERSTANDING THE SECURITY-IMPLICATIONS OF CONTAINER ISOLATION

The main advantage of using virtual machines over containers is the complete isolation they provide, since each VM has its own Linux kernel, while containers all use the same kernel. This can clearly pose a security risk. If there's a bug in the kernel, an application in one container might use it to read the memory of applications in other containers. If the apps run in different VMs and therefore share only the hardware, the probability of such attacks is much lower. Of course, complete isolation is only achieved by running applications on separate physical machines.

Additionally, containers share memory space, whereas each VM uses its own chunk of memory. Therefore, if you don't limit the amount of memory that a container can use, this

图2.3 在裸机、虚拟机和容器中运行应用程序的区别

在第一种情况下，所有三个应用程序都使用相同的内核并且根本不隔离。在第二种情况下，应用程序 A 和 B 运行在同一个 VM 中，因此共享内核，而应用程序 C 与其他两个完全隔离，因为它使用自己的内核。它仅与前两者共享硬件。

第三个案例显示了在容器中运行的相同三个应用程序。虽然它们都使用相同的内核，但它们彼此隔离，完全不知道其他的存在。隔离是由内核本身提供的。每个应用程序只看到物理硬件的一部分，并将自己视为操作系统中运行的唯一进程，尽管它们都运行在同一个操作系统中。

了解容器隔离的安全影响

与容器相比，使用虚拟机的主要优点是它们提供的完全隔离，因为每个虚拟机都有自己的 Linux 内核，而容器都使用相同的内核。这显然会带来安全风险。如果内核中存在错误，一个容器中的应用程序可能会使用它来读取其他容器中应用程序的内存。如果应用程序在不同的虚拟机中运行，因此仅共享硬件，则此类攻击的可能性会低得多。当然，完全隔离只有通过在单独的物理机器上运行应用程序才能实现。

此外，容器共享内存空间，而每个虚拟机使用自己的内存块。因此，如果不限制容器可以使用的内存量

could cause other containers to run out of memory or cause their data to be swapped out to disk.

NOTE This can't happen in Kubernetes, because it requires that swap is disabled on all the nodes.

UNDERSTANDING WHAT ENABLES CONTAINERS AND WHAT ENABLES VIRTUAL MACHINES

While virtual machines are enabled through virtualization support in the CPU and by virtualization software on the host, containers are enabled by the Linux kernel itself. You'll learn about container technologies later when you can try them out for yourself. You'll need to have Docker installed for that, so let's learn how it fits into the container story.

2.1.2 Introducing the Docker container platform

While container technologies have existed for a long time, they only became widely known with the rise of Docker. Docker was the first container system that made them easily portable across different computers. It simplified the process of packaging up the application and all its libraries and other dependencies - even the entire OS file system - into a simple, portable package that can be used to deploy the application on any computer running Docker.

INTRODUCING CONTAINERS, IMAGES AND REGISTRIES

Docker is a platform for packaging, distributing and running applications. As mentioned earlier, it allows you to package your application along with its entire environment. This can be just a few dynamically linked libraries required by the app, or all the files that are usually shipped with an operating system. Docker allows you to distribute this package via a public repository to any other Docker-enabled computer.

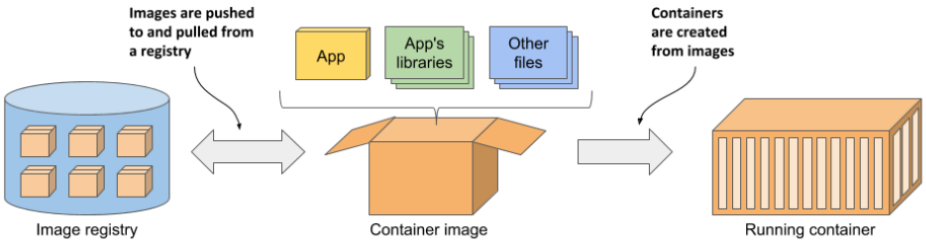


Figure 2.4 The three main Docker concepts are images, registries and containers

Figure 2.4 shows three main Docker concepts that appear in the process I've just described. Here's what each of them is:

- *Images*—A container image is something you package your application and its environment into. Like a zip file or a tarball. It contains the whole filesystem that the application will use and additional metadata, such as the path to the executable file to

可能会导致其他容器耗尽内存或导致其数据被换出到磁盘。

注意这在 Kubernetes 中不会发生，因为它要求在所有节点上禁用交换。

了解什么支持容器以及什么支持虚拟机

虚拟机是通过 CPU 中的虚拟化支持和主机上的虚拟化软件启用的，而容器是由 Linux 内核本身启用的。稍后当您可以亲自尝试时，您将了解容器技术。为此，您需要安装 Docker，所以让我们了解它如何融入容器故事。

2.1.2 Docker容器平台介绍

虽然容器技术已经存在很长时间了，但随着 Docker 的兴起，它们才变得广为人知。 Docker 是第一个使它们能够在不同计算机之间轻松移植的容器系统。它简化了将应用程序及其所有库和其他依赖项（甚至整个操作系统文件系统）打包成一个简单、可移植的包的过程，该包可用于在任何运行 Docker 的计算机上部署应用程序。

引入容器、镜像和注册表

Docker 是一个用于打包、分发和运行应用程序的平台。如前所述，它允许您将应用程序及其整个环境打包。这可以只是应用程序所需的几个动态链接库，也可以是操作系统通常附带的所有文件。Docker 允许您通过公共存储库将此包分发到任何其他启用 Docker 的计算机。

图 2.4 Docker 的三个主要概念是镜像、注册表和容器

图 2.4 显示了我刚才描述的过程中出现的三个主要 Docker 概念。以下是它们各自的含义：

• 映像—容器映像是您将应用程序及其环境打包到其中的东西。就像 zip 文件或 tarball 一样。它包含应用程序将使用的整个文件系统和附加元数据

run when the image is executed, the ports the application listens on, and other information about the image.

- *Registries*—A registry is a repository of container images that enables the exchange of images between different people and computers. After you build your image, you can either run it on the same computer, or *push* (upload) the image to a registry and then *pull* (download) it to another computer. Certain registries are public, allowing anyone to pull images from it, while others are private and only accessible to individuals, organizations or computers that have the required authentication credentials.
- *Containers*—A container is instantiated from a container image. A running container is a normal process running in the host operating system, but its environment is isolated from that of the host and the environments of other processes. The file system of the container originates from the container image, but additional file systems can also be mounted into the container. A container is usually resource-restricted, meaning it can only access and use the amount of resources such as CPU and memory that have been allocated to it.

BUILDING, DISTRIBUTING, AND RUNNING A CONTAINER IMAGE

To understand how containers, images and registries relate to each other, let's look at how to build a container image, distribute it through a registry and create a running container from the image. These three processes are shown in figures 2.5 to 2.7.

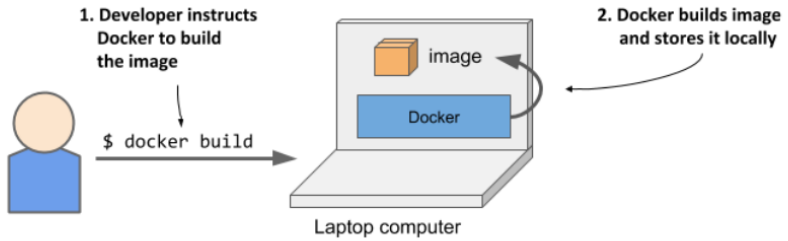


Figure 2.5 Building a container image

As shown in figure 2.5, the developer first builds an image, and then pushes it to a registry, as shown in figure 2.6. The image is now available to anyone who can access the registry.

29

执行映像时运行、应用程序侦听的端口以及有关映像的其他信息。

- 注册表——注册表是容器镜像的存储库，可以在不同的人和计算机之间交换镜像。构建映像后，您可以在同一台计算机上运行它，也可以将映像推送（上传）到注册表，然后将其拉取（下载）到另一台计算机。某些注册表是公开的，允许任何人从中提取图像，而其他注册表则是私有的，只有具有所需身份验证凭据的个人、组织或计算机才能访问。

- 容器——容器是从容器映像实例化的。运行中的容器是运行在宿主机操作系统中的一个正常进程，但其环境与宿主机及其他进程的环境是隔离的。容器的文件系统源自容器镜像，但也可以将其他文件系统挂载到容器中。容器通常是资源受限的，这意味着它只能访问和使用分配给它的资源量，例如 CPU 和内存。

构建、分发和运行容器镜像

为了了解容器、镜像和注册表之间的关系，我们来看看如何构建容器镜像、通过注册表分发它以及从镜像创建运行的容器。这三个过程如图2.5至2.7所示。

图2.5 构建容器镜像

如图2.5所示，开发者首先构建镜像，然后将其推送到注册表，如图2.6所示。现在，任何可以访问注册表的人都可以使用该映像。

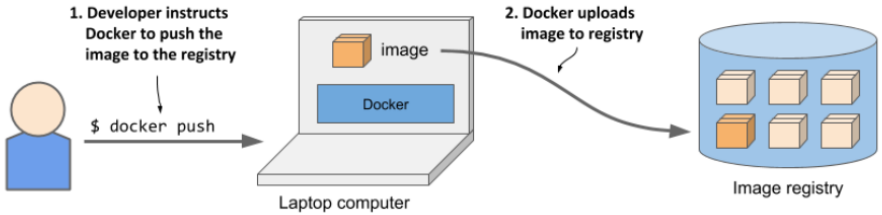


Figure 2.6 Uploading a container image to a registry

As the next figure shows, another person can now pull the image to any other computer running Docker and run it. Docker creates an isolated container based on the image and invokes the executable file specified in the image.

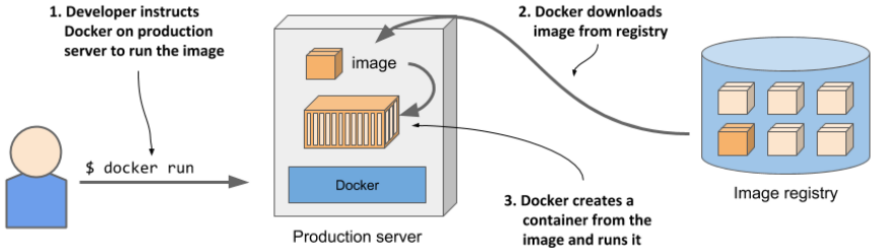


Figure 2.7 Running a container on a different computer

Running the application on any computer is made possible by the fact that the environment of the application is decoupled from the environment of the host.

UNDERSTANDING THE ENVIRONMENT THAT THE APPLICATION SEES

When you run an application in a container, it sees exactly the file system content you bundled into the container image, as well as any additional file systems you mount into the container. The application sees the same files whether it's running on your laptop or a full-fledged production server, even if the production server uses a completely different Linux distribution. The application typically has no access to the files in the host's operating system, so it doesn't matter if the server has a completely different set of installed libraries than your development computer.

For example, if you package your application with the files of the entire Red Hat Enterprise Linux (RHEL) operating system and then run it, the application will think it's running inside RHEL, whether you run it on your Fedora-based or a Debian-based computer.

30

图2.6 上传容器镜像到registry

如下图所示，另一个人现在可以将映像拉取到任何其他运行 Docker 的计算机并运行它。 Docker 根据镜像创建一个隔离的容器，并调用镜像中指定的可执行文件。

图2.7 在不同的计算机上运行容器

由于应用程序的环境与主机的环境分离，因此可以在任何计算机上运行应用程序。

了解应用程序所看到的环境

当您在容器中运行应用程序时，它会准确地看到您捆绑到容器映像中的文件系统内容，以及您挂载到容器中的任何其他文件系统。无论应用程序是在笔记本电脑上运行还是在成熟的生产服务器上运行，即使生产服务器使用完全不同的 Linux 发行版，应用程序都会看到相同的文件。应用程序通常无法访问主机操作系统中的文件，因此服务器是否具有与开发计算机完全不同的一组已安装库并不重要。

例如，如果您将应用程序与整个 Red Hat Enterprise Linux (RHEL) 操作系统的文件打包然后运行它，则应用程序将认为它在 RHEL 内部运行，无论您在基于 Fedora 的系统上还是在 Debian 上运行它 -基于计算机。

The Linux distribution installed on the host is irrelevant. The only thing that might be important is the kernel version and the kernel modules it loads. Later, I'll explain why.

This is similar to creating a VM image by creating a new VM, installing an operating system and your app in it, and then distributing the whole VM image so that other people can run it on different hosts. Docker achieves the same effect, but instead of using VMs for app isolation, it uses Linux container technologies to achieve (almost) the same level of isolation.

UNDERSTANDING IMAGE LAYERS

Unlike virtual machine images, which are big blobs of the entire filesystem required by the operating system installed in the VM, container images consist of layers that are usually much smaller. These layers can be shared and reused across multiple images. This means that only certain layers of an image need to be downloaded if the rest were already downloaded to the host as part of another image containing the same layers.

Layers make image distribution very efficient but also help to reduce the storage footprint of images. Docker stores each layer only once. As you can see in figure 2.8, two containers created from two images that contain the same layers use the same files.

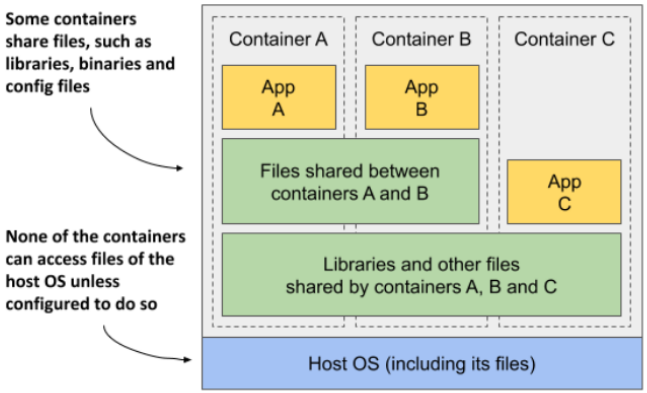


Figure 2.8 Containers can share image layers

The figure shows that containers A and B share an image layer, which means that applications A and B read some of the same files. In addition, they also share the underlying layer with container C. But if all three containers have access to the same files, how can they be completely isolated from each other? Are changes that application A makes to a file stored in the shared layer not visible to application B? They aren't. Here's why.

The filesystems are isolated by the Copy-on-Write (CoW) mechanism. The filesystem of a container consists of read-only layers from the container image and an additional read/write

主机上安装的 Linux 发行版无关。唯一可能重要的是内核版本及其加载的内核模块。稍后，我会解释原因。

这类似于通过创建新 VM、在其中安装操作系统和应用程序来创建 VM 映像，然后分发整个 VM 映像，以便其他人可以在不同主机上运行它。 Docker 实现了相同的效果，但它不是使用虚拟机进行应用程序隔离，而是使用 Linux 容器技术来实现（几乎）相同级别的隔离。

理解图像层

与虚拟机映像（虚拟机映像是虚拟机中安装的操作系统所需的整个文件系统的大块）不同，容器映像由通常小得多的层组成。这些层可以在多个图像之间共享和重用。这意味着，如果其余层已作为包含相同层的另一个图像的一部分下载到主机，则仅需要下载图像的某些层。

层使图像分发非常高效，但也有助于减少图像的存储占用空间。 Docker 每层仅存储一次。如图 2.8 所示，从包含相同层的两个镜像创建的两个容器使用相同的文件。

图2.8 容器可以共享镜像层

从图中可以看出，容器A和B共享一个镜像层，这意味着应用程序A和B读取了一些相同的文件。另外，它们还和容器C共享底层。但是如果三个容器都访问相同的文件，如何才能做到彼此完全隔离呢？应用程序 A 对存储在共享层中的文件所做的更改对应用程序 B 不可见吗？他们不是。原因如下。

文件系统通过写入时复制 (CoW) 机制进行隔离

layer stacked on top. When an application running in container A changes a file in one of the read-only layers, the entire file is copied into the container's read/write layer and the file contents are changed there. Since each container has its own writable layer, changes to shared files are not visible in any other container.

When you delete a file, it is only marked as deleted in the read/write layer, but it's still present in one or more of the layers below. What follows is that deleting files never reduces the size of the image.

> **WARNING** Even seemingly harmless operations such as changing permissions or ownership of a file result in a new copy of the entire file being created in the read/write layer. If you perform this type of operation on a large file or many files, the image size may swell significantly.

UNDERSTANDING THE PORTABILITY LIMITATIONS OF CONTAINER IMAGES

In theory, a Docker-based container image can be run on any Linux computer running Docker, but one small caveat exists, because containers don't have their own kernel. If a containerized application requires a particular kernel version, it may not work on every computer. If a computer is running a different version of the Linux kernel or doesn't load the required kernel modules, the app can't run on it. This scenario is illustrated in the following figure.

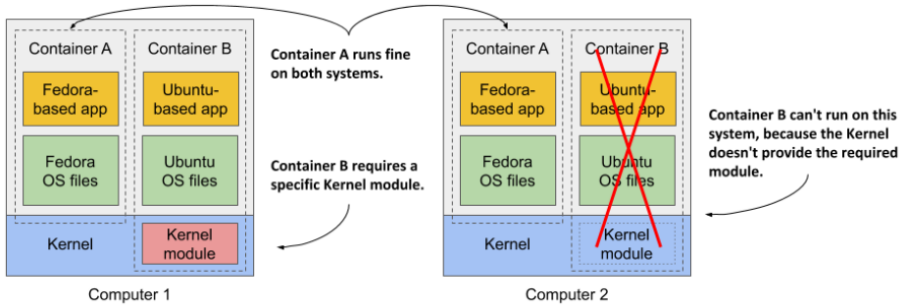


Figure 2.9 If a container requires specific kernel features or modules, it may not work everywhere

Container B requires a specific kernel module to run properly. This module is loaded in the kernel in the first computer, but not in the second. You can run the container image on the second computer, but it will break when it tries to use the missing module.

And it's not just about the kernel and its modules. It should also be clear that a containerized app built for a specific hardware architecture can only run on computers with the same architecture. You can't put an application compiled for the x86 CPU architecture into a container and expect to run it on an ARM-based computer just because Docker is available there. For this you would need a VM to emulate the x86 architecture.

32

层堆叠在顶部。当容器 A 中运行的应用程序更改只读层之一中的文件时，整个文件将被复制到容器的读/写层，并在那里更改文件内容。由于每个容器都有自己的可写层，因此对共享文件的更改在任何其他容器中都是不可见的。

当您删除文件时，它仅在读/写层中被标记为已删除，但它仍然存在于下面的一层或多层中。接下来是删除文件永远不会减少图像的大小。

> 警告 即使看似无害的操作（例如更改文件的权限或所有权）也会导致在读/写层中创建整个文件的新副本。如果您执行此类操作
> 大文件或多个文件时，图像大小可能会显着增大。

了解容器镜像的可移植性限制

理论上，基于 Docker 的容器镜像可以在任何运行 Docker 的 Linux 计算机上运行，但有一点需要注意，因为容器没有自己的内核。如果容器化应用程序需要特定的内核版本，它可能无法在每台计算机上运行。如果计算机运行不同版本的 Linux 内核或未加载所需的内核模块，则应用程序无法在其上运行。下图说明了这种情况。

图 2.9 如果容器需要特定的内核功能或模块，它可能无法在任何地方都工作

容器B需要特定的内核模块才能正常运行。该模块加载到第一台计算机的内核中，但不加载到第二台计算机中。您可以在第二台计算机上运行容器映像，但当它尝试使用缺少的模块时，它将中断。

这不仅仅是内核及其模块的问题。还应该清楚的是，为特定硬件架构构建的容器化应用程序只能在具有相同架构的计算机上运行。你不能仅仅因为 Docker 是为 x86 CPU 架构编译的应用程序放入容器中并期望在基于 ARM 的计算机上运行它

在那里可用。为此，您需要一个虚拟机来模拟 x86 架构。

2.1.3 Introducing Docker alternatives and the Open Container Initiative

Docker was the first container platform to make containers mainstream. I hope I've made it clear that Docker itself doesn't provide process isolation. The actual isolation of containers takes place at the Linux kernel level using the mechanisms it provides. Docker just makes it easy to use these mechanisms and allows you to distribute container images to different hosts.

Introducing the Open Container Initiative (OCI)

After the success of Docker, the Open Container Initiative (OCI) was born to create open industry standards around container formats and runtime. Docker is part of this initiative, as are other container runtimes and a number of organizations with interest in container technologies.

OCI members created the *OCI Image Format Specification*, which prescribes a standard format for container images, and the *OCI Runtime Specification*, which defines a standard interface for container runtimes with the aim of standardizing the creation, configuration and execution of containers.

Introducing the Container Runtime Interface (CRI) and its implementation (CRI-O)

This book focuses on using Docker as the container runtime for Kubernetes, as it was initially the only one supported by Kubernetes and is still the most widely used. But Kubernetes now supports many other container runtimes through the Container Runtime Interface (CRI).

One implementation of CRI is CRI-O, a lightweight alternative to Docker that allows you to leverage any OCI-compliant container runtime with Kubernetes. Examples of OCI-compliant runtimes include *rkt* (pronounced Rocket), *runC*, and *Kata Containers*.

2.2 Exploring containers hands-on

You now have a feel for what containers are, but I have yet to explain *how* they work. Before we get to that, you need to create an application, package it into a container image and run it. You'll need Docker, so let's install it and run a Hello world container first.

2.2.1 Installing Docker and running a Hello World container

Ideally, you'll install Docker directly on a Linux computer, so you won't have to deal with the additional complexity of running containers inside a VM running within your host OS. But, if you're using macOS or Windows and don't know how to set up a Linux VM, the Docker Desktop application will set it up for you. The Docker command-line (CLI) tool that you'll use to run containers will be installed in your host OS, but the Docker daemon will run inside the VM, as will all the containers it creates.

The Docker Platform consists of many components, but you only need to install Docker Engine to run containers. If you use macOS or Windows, install Docker Desktop. For details, follow the instructions at http://docs.docker.com/install.

NOTE Docker Desktop for Windows can run either Windows or Linux containers. Make sure that you configure it to use Linux containers.

2.1.3 介绍 Docker 替代方案和开放容器计划

Docker 是第一个使容器成为主流的容器平台。我希望我已经明确表示 Docker 本身不提供进程隔离。容器的实际隔离是在 Linux 内核级别使用其提供的机制进行的。 Docker 只是让这些机制的使用变得简单，并允许您将容器镜像分发到不同的主机。

开放容器倡议 (OCI) 简介

Docker 取得成功后，开放容器计划 (OCI) 诞生了，旨在围绕容器格式和运行时创建开放行业标准。 Docker 是该计划的一部分，其他容器运行时和许多对容器技术感兴趣的组织也是如此。

OCI 成员创建了 OCI 镜像格式规范（规定了容器镜像的标准格式）和 OCI 运行时规范（定义了容器运行时的标准接口，旨在标准化容器的创建、配置和执行）。

介绍容器运行时接口（CRI）及其实现（CRI-O）

本书重点介绍使用 Docker 作为 Kubernetes 的容器运行时，因为它最初是 Kubernetes 支持的唯一容器运行时，并且仍然是使用最广泛的容器运行时。但 Kubernetes 现在通过容器运行时接口 (CRI) 支持许多其他容器运行时。

CRI-O 是 CRI 的一个实现，它是 Docker 的轻量级替代品，允许您将任何符合 OCI 的容器运行时与 Kubernetes 结合使用。 OCI 兼容运行时的示例包括 rkt（发音为 Rocket）、runC 和 Kata Containers。

2.2 亲自探索容器

您现在已经了解了什么是容器，但我还没有解释它们是如何工作的。在此之前，您需要创建一个应用程序，将其打包到容器映像中并运行它。您需要 Docker，所以让我们先安装它并运行一个 Hello world 容器。

2.2.1 安装Docker并运行Hello World容器

理想情况下，您将直接在 Linux 计算机上安装 Docker，这样您就不必处理在主机操作系统中运行的虚拟机内运行容器的额外复杂性。但是，如果您使用的是 macOS 或 Windows 并且不知道如何设置 Linux VM，Docker Desktop 应用程序将为您进行设置。用于运行容器的 Docker 命令行 (CLI) 工具将安装在主机操作系统中，但 Docker 守护程序将在虚拟机内运行，它创建的所有容器也将在虚拟机内运行。

Docker Platform由许多组件组成，但您只需要安装Docker Engine即可运行容器。如果您使用 macOS 或 Windows，请安装 Docker Desktop。有关详细信息，请按照 http://docs.docker.com/install 中的说明进行操作。

注意 适用于 Windows 的 Docker Desktop 可以运行 Windows 或 Linux 容器。确保将其配置为使用 Linux 容器。

Running a Hello World container

After the installation is complete, you use the `docker` CLI tool to run Docker commands. First, let's try pulling and running an existing image from Docker Hub, the public image registry that contains ready-to-use container images for many well-known software packages. One of them is the `busybox` image, which you'll use to run a simple `echo "Hello world"` command in your first container.

If you're unfamiliar with `busybox`, it's a single executable file that combines many of the standard UNIX command-line tools, such as `echo`, `ls`, `gzip`, and so on. Instead of the `busybox` image, you could also use any other full-fledged OS container image like Fedora, Ubuntu, or any other image that contains the `echo` executable file.

You don't need to download or install anything to run the `busybox` image. You can do everything with a single `docker run` command, by specifying the image to download and the command to run in it. To run the simple Hello world container, execute the command shown in the following listing.

Listing 2.1 Running a Hello World container with Docker

```
$ docker run busybox echo "Hello World"
Unable to find image 'busybox:latest' locally          #A
latest: Pulling from library/busybox                   #A
7c9d20b9b6cd: Pull complete                            #A
Digest: sha256:fe301db49df08c384001ed752dff6d52b4…     #A
Status: Downloaded newer image for busybox:latest      #A
Hello World                                            #B
```

#A Docker downloads the container image
#B The output produced by the echo command

This doesn't look too impressive, but keep in mind that the entire "application" was downloaded and executed with a single command, without you having to install that application or any of its dependencies.

In your case, the app was just a single executable file, but it could have been an incredibly complex app with dozens of libraries and additional files. The entire process of setting up and running the app would be the same. What isn't obvious is that the app ran in a container, isolated from the other processes on the computer. You'll see that this is true in the exercises that follow.

Understanding what happens when you run a container

Figure 2.10 shows exactly what happened when you executed the `docker run` command.

运行 Hello World 容器

安装完成后，您可以使用docker CLI工具运行Docker命令。首先，让我们尝试从 Docker Hub 拉取并运行现有映像，Docker Hub 是公共映像注册表，其中包含许多知名软件的即用型容器映像包。其中一个是 busybox 映像，您将使用它在第一个容器中运行简单的 echo "Hello world" 命令。

如果您不熟悉 busybox，它是一个单一的可执行文件，结合了许多标准UNIX命令行工具，例如echo、ls、gzip等。除了 busybox 镜像之外，您还可以使用任何其他成熟的操作系统容器镜像，例如 Fedora，Ubuntu 或任何其他包含 echo 可执行文件的映像。

您无需下载或安装任何内容即可运行 busybox 映像。你可以做通过指定要下载的映像和在其中运行的命令，只需一个 docker run 命令即可完成所有操作。要运行简单的 Hello world 容器，请执行以下清单中所示的命令。

清单 2.1 使用 Docker 运行 Hello World 容器

```
$ docker run busybox echo "Hello World" 无法在本地找到图像 "busybox：最新" #A
最新：从图书馆/busybox #A 中提取
7c9d20b9b6cd：拉完成#A
摘要：sha256：
#A Docker 下载容器镜像 #B echo 命令产生的输出
```

这看起来不太令人印象深刻，但请记住，整个"应用程序"是通过单个命令下载并执行的，而无需安装该应用程序或其任何依赖项。

就您而言，该应用程序只是一个可执行文件，但它可能是一个极其复杂的应用程序，包含数十个库和其他文件。设置和运行应用程序的整个过程是相同的。不明显的是，该应用程序在容器中运行，与计算机上的其他进程隔离。在接下来的练习中您会发现这是正确的。

了解运行容器时会发生什么

图 2.10 显示了执行 docker run 命令时发生的情况。

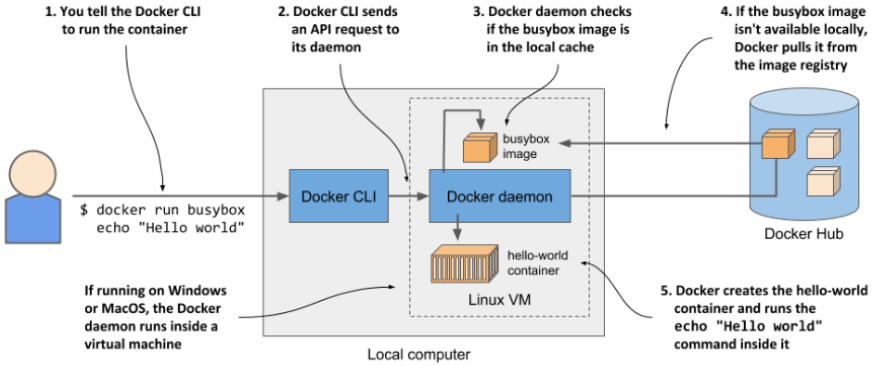


Figure 2.10 Running echo "Hello world" in a container based on the busybox container image

The `docker` CLI tool sent an instruction to run the container to the Docker daemon, which checked whether the `busybox` image was already present in its local image cache. It wasn't, so it pulled it from the Docker Hub registry.

After downloading the image to your computer, the Docker daemon created a container from that image and executed the `echo` command in it. The command printed the text to the standard output, the process then terminated and the container stopped.

If your local computer runs a Linux OS, the Docker CLI tool and the daemon both run in this OS. If it runs macOS or Windows, the daemon and the containers run in the Linux VM.

RUNNING OTHER IMAGES

Running other existing container images is much the same as running the `busybox` image. In fact, it's often even simpler, since you don't normally need to specify what command to execute, as with the echo command in the previous example. The command that should be executed is usually written in the image itself, but you can override it when you run it.

For example, if you want to run the Redis datastore, you can find the image name on http://hub.docker.com or another public registry. In the case of Redis, one of the images is called `redis:alpine`, so you'd run it like this:

```
$ docker run redis:alpine
```

To stop and exit the container, press Control-C (or Command-C on a Mac).

> **NOTE** If you want to run an image from a different registry, you must specify the registry along with the image name. For example, if you want to run an image from the Quay.io registry, which is another publicly accessible image registry, run it as follows: `docker run quay.io/some/image`.

图2.10 在基于busybox容器镜像的容器中运行echo "Hello world"

docker CLI工具向Docker守护进程发送运行容器的指令，该指令

检查 busybox 图像是否已存在于其本地图像缓存中。事实并非如此，所以它从 Docker Hub 注册表中提取了它。

将映像下载到计算机后，Docker 守护程序从该映像创建一个容器并在其中执行 echo 命令。该命令将文本打印到标准输出，然后进程终止并且容器停止。

如果您的本地计算机运行的是Linux操作系统，则Docker CLI工具和守护进程都在该操作系统中运行。如果它运行 macOS 或 Windows，则守护程序和容器在 Linux VM 中运行。
运行其他图像

运行其他现有容器映像与运行 busybox 映像非常相似。事实上，它通常更简单，因为您通常不需要指定要执行的命令，就像上一个示例中的 echo 命令一样。应执行的命令通常写在映像本身中，但您可以在运行时覆盖它。

例如，如果您想运行 Redis 数据存储，您可以在 http://hub.docker.com 或其他公共注册表上找到映像名称。对于 Redis，其中一个映像称为 redis:alpine，因此您可以像这样运行它：

```
$ docker run redis:alpine
```

要停止并退出容器，请按 Control-C（或 Mac 上的 Command-C）。

> 注意如果要运行来自不同注册表的映像，则必须指定注册表以及映像名称。例如，如果您想运行来自 Quay.io 注册表的映像，这是另一个公开的
> 可访问的镜像注册表，按如下方式运行：docker run quay.io/some/image。

UNDERSTANDING IMAGE TAGS

If you've searched for the Redis image on Docker Hub, you've noticed that there are many image *tags* you can choose from. For Redis, the tags are `latest`, `buster`, `alpine`, but also `5.0.7-buster`, `5.0.7-alpine`, and so on.

Docker allows you to have multiple versions or variants of the same image under the same name. Each variant has a unique tag. If you refer to images without explicitly specifying the tag, Docker assumes that you're referring to the special `latest` tag. When uploading a new version of an image, image authors usually tag it with both the actual version number and with `latest`. When you want to run the latest version of an image, use the `latest` tag instead of specifying the version.

> **NOTE** The `docker run` command only pulls the image if it hasn't already pulled it before. Using the `latest` tag ensures that you get the latest version when you first run the image. The locally cached image is used from that point on.

Even for a single version, there are usually several variants of an image. For Redis I mentioned `5.0.7-buster` and `5.0.7-alpine`. They both contain the same version of Redis, but differ in the base image they are built on. `5.0.7-buster` is based on Debian version "Buster", while `5.0.7-alpine` is based on the Alpine Linux base image, a very stripped-down image that is only 5MB in total – it contains only a small set of the installed binaries you see in a typical Linux distribution.

To run a specific version and/or variant of the image, specify the tag in the image name. For example, to run the `5.0.7-alpine` tag, you'd execute the following command:

```
$ docker run redis:5.0.7-alpine
```

2.2.2 Creating a containerized Node.js web application

Now that you have a working Docker setup, you'll create an app that you'll use throughout the book. You'll create a trivial Node.js web application and package it into a container image. The application will accept HTTP requests and respond with the hostname of the computer it's running on.

This way, you'll see that an app running in a container sees a different hostname and not that of the host computer, even though it runs on the host like any other process. This will be useful later, when you deploy the app on Kubernetes and scale it out (scale it horizontally; that is, run multiple instances of the app). You'll see your HTTP requests hitting different instances of the app.

The app consists of a single file called `app.js` whose contents are shown in the next listing.

Listing 2.2 A simple Node.js web application: app.js

```
const http = require('http');                                    #A
const os = require('os');                                        #A

const listenPort = 8080;                                         #B
```


理解图像标签

如果您在 Docker Hub 上搜索过 Redis 镜像，您会发现有很多镜像标签可供选择。对于Redis，标签有latest、buster、alpine，还有5.0.7-buster、5.0.7-alpine 等等。

Docker 允许您在同一名称下拥有同一映像的多个版本或变体。每个变体都有一个独特的标签。如果您在没有明确指定标签的情况下引用图像，Docker 会假设您引用的是特殊的最新标签。当上传图像的新版本时，图像作者通常会使用实际版本号和最新版本对其进行标记。当您想要运行最新版本的映像时，请使用最新标签而不是指定版本。

> 注意 docker run 命令仅在之前未拉取过镜像的情况下才拉取该镜像。使用最新标签可确保您在首次运行映像时获得最新版本。本地缓存的图像是
> 从那时起就使用了。

即使对于单个版本，图像通常也有多个变体。对于 Redis，我提到了 5.0.7-buster 和 5.0.7-alpine。它们都包含相同版本的 Redis，但构建的基础镜像有所不同。 5.0.7-buster 基于 Debian 版本 "Buster"，而 5.0.7-alpine 基于 Alpine Linux 基础镜像，这是一个非常精简的镜像，总共只有 5MB – 它只包含一小部分您在典型 Linux 发行版中看到的已安装二进制文件。

要运行图像的特定版本和/或变体，请在图像名称中指定标签。例如，要运行 5.0.7-alpine 标签，您需要执行以下命令：

```
$ docker 运行 redis:5.0.7-alpine
```

2.2.2 创建容器化 Node.js Web 应用程序

现在您已经有了一个可用的 Docker 设置，您将创建一个将在整本书中使用的应用程序。您将创建一个简单的 Node.js Web 应用程序并将其打包到容器映像中。应用程序将接受 HTTP 请求并使用其运行的计算机的主机名进行响应。

这样，您将看到在容器中运行的应用程序看到不同的主机名，而不是主机的主机名，即使它像任何其他进程一样在主机上运行。稍后，当您在 Kubernetes 上部署应用程序并将其扩展（水平扩展；即运行应用程序的多个实例）时，这将很有用。您将看到您的 HTTP 请求到达应用程序的不同实例。

该应用程序由一个名为 app.js 的文件组成，其内容如下所示
清单。

清单 2.2 一个简单的 Node.js Web 应用程序：app.js

```
const http = require('http'); #A
常量监听端口 = 8080; #B
```

```
console.log("Kubia server starting...");                                  #C
console.log("Local hostname is " + os.hostname());                        #C
console.log("Listening on port " + listenPort);                           #C

var handler = function(request, response) {                               #D
  let clientIP = request.connection.remoteAddress;                        #E
  console.log("Received request for "+request.url+" from "+clientIP);     #F
  response.writeHead(200);                                                #G
  response.write("Hey there, this is "+os.hostname()+". ");               #G
  response.write("Your IP is "+clientIP+". ");                            #G
  response.end("\n");                                                     #G
};

var server = http.createServer(handler);                                  #H
server.listen(listenPort);                                                #H
```

#A The http and os modules are loaded
#B Define the port the server listens on
#C Print server information to standard output
#D This function handles incoming HTTP requests
#E Get the client IP from the HTTP request
#F Log information about the request to the standard output
#G Create the HTTP response
#H Start the HTTP server

The code in the listing should be easy to understand. It starts an HTTP server on port `8080`. For each request, it logs the request information to standard output and sends a response with status code `200 OK` and the following text:

```
Hey there, this is <server-hostname>. Your IP is <client-IP>.
```

NOTE The hostname in the response is the server's actual hostname, not the one sent by the client in the request's `Host` header. This detail will be important later.

You could now download and install Node.js locally and test your app directly, but that's not necessary. It's easier to package it into a container image and run it with Docker. This enables you to run the app on any other Docker-enabled host without installing Node.js there either.

2.2.3 Creating a Dockerfile to build the container image

To package your app into an image, you must first create a file called `Dockerfile`, which contains a list of instructions that Docker should perform when building the image. Create the file in the same directory as the `app.js` file and make sure it contains the three directives in the following listing.

Listing 2.3 A minimal Dockerfile for building a container image for your app

```
FROM node:12                    #A
ADD app.js /app.js              #B
ENTRYPOINT ["node", "app.js"]   #C
```



```
console.log("Kubia服务器正在启动...");                                    #C
console.log("本地主机名是 " + os.hostname());                             #C
console.log("监听端口 " + listenPort);                                    #C

var handler = 函数(请求, 响应) {                                          #D
  让 clientIP = request.connection.remoteAddress;                         #E
  console.log("收到来自 "+clientIP+" 的 "+request.url+" 请求");           #F
  响应.writeHead(200);                                                    #G
  response.write("嘿，这是 "+os.hostname()+"。");                          #G
  response.write("您的IP是"+clientIP+"。");                               #G
  响应.end("\n");                                                         #G
};

var server = http.createServer(handler);                                  #H
服务器.listen(listenPort);                                                #H
```

#A 加载 http 和 os 模块 #B 定义服务器侦听的端口 #C 将服务器信息打印到标准输出 #D 该函数处理传入的 HTTP 请求 #E 从 HTTP 请求中获取客户端 IP

#F 将有关请求的信息记录到标准输出 #G 创建 HTTP 响应 #H 启动 HTTP 服务器

清单中的代码应该很容易理解。它在端口 8080 上启动 HTTP 服务器。对于每个请求，它将请求信息记录到标准输出，并发送状态代码 200 OK 和以下文本的响应：

```
嘿，这是<服务器主机名>。您的 IP 是 <客户端 IP>。
```

注意：响应中的主机名是服务器的实际主机名，而不是客户端在请求的 Host 标头中发送的主机名。这个细节稍后会很重要。

您现在可以在本地下载并安装 Node.js 并直接测试您的应用程序，但这不是必需的。将其打包成容器镜像并使用 Docker 运行会更容易。这使您能够在任何其他启用 Docker 的主机上运行该应用程序，而无需在那里安装 Node.js。

2.2.3 创建Dockerfile来构建容器镜像

要将应用程序打包到映像中，您必须首先创建一个名为 Dockerfile 的文件，其中包含 Docker 在构建映像时应执行的指令列表。在与 app.js 文件相同的目录中创建该文件，并确保它包含以下清单中的三个指令。

清单 2.3 用于为应用程序构建容器映像的最小 Dockerfile

```
来自节点：12 #A
添加 app.js /app.js #B
```

```
#A The base image to build upon
#B Adds the app.js file into the container image
#C Specifies the command to execute when the image is run
```

The `FROM` line defines the container image that you'll use as the starting point (the base image you're building on top of). In your case, you use the `node` container image, tag `12`. In the second line, you add the `app.js` file from your local directory into the root directory of the image, under the same name (`app.js`). Finally, in the third line, you specify the command that Docker should run when you execute the image. In your case, the command is `node app.js`.

Choosing a base image

You may wonder why use this specific image as your base. Because your app is a Node.js app, you need your image to contain the `node` binary file to run the app. You could have used any image containing this binary, or you could have even used a Linux distribution base image such as `fedora` or `ubuntu` and installed Node.js into the container when building the image. But since the `node` image already contains everything needed to run Node.js apps, it doesn't make sense to build the image from scratch. In some organizations, however, the use of a specific base image and adding software to it at build-time may be mandatory.

2.2.4 Building the container image

The `Dockerfile` and the `app.js` file are everything you need to build your image. You'll now build the image named `kubia:latest` using the command in the next listing:

Listing 2.4 Building the image

```
$ docker build -t kubia:latest .
Sending build context to Docker daemon  3.072kB
Step 1/3 : FROM node:12                                  #A
12: Pulling from library/node
092586df9206: Pull complete                              #B
ef599477fae0: Pull complete                              #B
…                                                        #B
89e674ac3af7: Pull complete                              #B
08df71ec9bb0: Pull complete                              #B
Digest: sha256:a919d679dd773a56acce15afa0f436055c9b9f20e1f28b4469a4bee69e0…
Status: Downloaded newer image for node:12
 ---> e498dabfee1c                                       #C
Step 2/3 : ADD app.js /app.js                            #D
 ---> 28d67701d6d9                                       #D
Step 3/3 : ENTRYPOINT ["node", "app.js"]                   #E
 ---> Running in a01d42eda116                            #E
Removing intermediate container a01d42eda116               #E
 ---> b0ecc49d7a1d                                       #E
Successfully built b0ecc49d7a1d                          #F
Successfully tagged kubia:latest                         #F
```

```
#A This corresponds to the first line of your Dockerfile
#B Docker is downloading individual layers of the node:12 image
#C This is the ID of image after the first build step is complete
```



```
#A 构建的基础镜像
#B 将 app.js 文件添加到容器镜像中
```

FROM 行定义您将用作起点的容器映像（您正在其上构建的基础映像）。在您的例子中，您使用节点容器映像，标记 12。在第二行中，您将本地目录中的 app.js 文件添加到映像的根目录中，并使用相同的名称 (app.js)。最后，在第三行中，指定执行映像时 Docker 应运行的命令。在您的例子中，命令是node app.js。

选择基础图像

您可能想知道为什么使用这个特定图像作为基础。因为您的应用程序是 Node.js 应用程序，所以您需要您的图像

包含运行应用程序的节点二进制文件。您可以使用包含此二进制文件的任何图像，或者您可以

甚至使用 Linux 发行版基础镜像（例如 fedora 或 ubuntu），并将 Node.js 安装到容器中

建立形象。但由于节点镜像已经包含运行 Node.js 应用程序所需的所有内容，因此从头开始构建镜像是没有意义的。然而，在某些组织中，使用特定的基础映像并在构建时向其中添加软件可能

2.2.4 构建容器镜像

Dockerfile 和 app.js 文件是构建映像所需的一切。你现在会

使用下一个清单中的命令构建名为 kubia:latest 的映像：

清单 2.4 构建镜像

```
$ docker build -t kubia:latest .将构建上下文发
送到 Docker 守护进程 3.072kB
步骤 1/3：来自节点：12 #A
12：从库/节点 092586df9206 拉取：拉取完成 #B
ef599477fae0：拉完成#B
…… #B
89e674ac3af7：拉动完成#B
08df71ec9bb0：拉动完成#B
步骤 3/3：ENTRYPOINT ["node"，"app.js"] #E
摘要：sha256:a919d679dd773a56acce15afa0f436055c9b9f20e1f28b4469a4bee69e0… #E
状态：为节点下载了较新的镜像：12 #E
 ---> e498dabfee1c #C
步骤 2/3：ADD app.js /app.js #D
 ---> 28d67701d6d9 #D
 ---> 在 a01d42eda116 #E 中运行
移除中间容器 a01d42eda116 #E
 ---> b0ecc49d7a1d #E
成功构建 b0ecc49d7a1d #F
```

```
#A 这对应于 Dockerfile 的第一行 #B
Docker 正在下载节点：12 镜像的各个层 #C
这是第一个构建步骤完成后镜像的 ID
```

#D The second step of the build and the resulting image ID
#E The final step of the build
#F The final image ID and its tag

The `-t` option specifies the desired image name and tag and the dot at the end specifies the path to the directory that contains the Dockerfile and the build context (all artefacts needed by the build process).

When the build process is complete, the newly created image is available in your computer's local image store. You can see it by listing local images, as in the following listing.

Listing 2.5 Listing locally stored images

```
$ docker images
REPOSITORY   TAG      IMAGE ID          CREATED           VIRTUAL SIZE
kubia        latest   b0ecc49d7a1d      1 minute ago      908 MB
…
```

UNDERSTANDING HOW THE IMAGE WAS BUILT

Figure 2.11 shows what happens during the build process. You tell Docker to build an image called `kubia` based on the contents of the current directory. Docker reads the `Dockerfile` in the directory and builds the image based on the directives in the file.

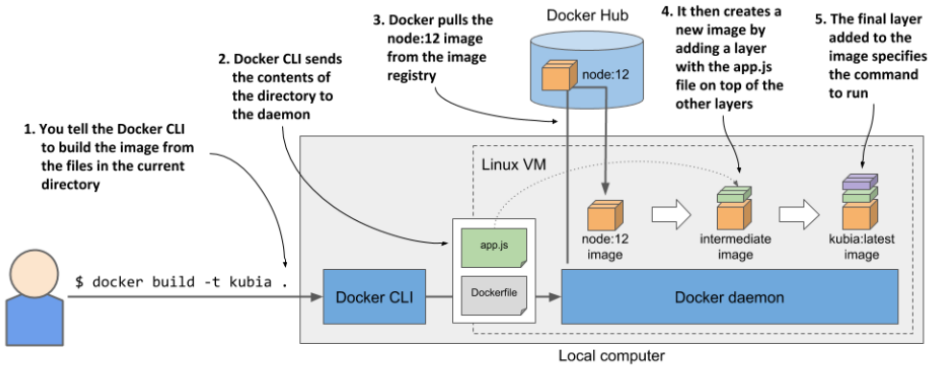


Figure 2.11 Building a new container image using a Dockerfile

The build itself isn't performed by the `docker` CLI tool. Instead, the contents of the entire directory are uploaded to the Docker daemon and the image is built by it. You've already learned that the CLI tool and the daemon aren't necessarily on the same computer. If you're using Docker on a non-Linux system such as macOS or Windows, the client is in your host OS, but the daemon runs inside a Linux VM. But it could also run on a remote computer.

#D 构建的第二步和生成的镜像 ID #E 构建的最后一步 #F 最终镜像 ID 及其标签

-t 选项指定所需的映像名称和标签，末尾的点指定包含 Dockerfile 和构建上下文（构建过程所需的所有工件）的目录的路径。

构建过程完成后，新创建的映像可在计算机的本地映像存储中使用。您可以通过列出本地图像来查看它，如下列表所示。

清单 2.5 列出本地存储的图像

```
$ docker images 存储库标签 图像 ID 创建的虚拟大小
kubia 最新 b0ecc49d7a1d 1 分钟前 908MB
…
```

了解图像是如何构建的

图 2.11 显示了构建过程中发生的情况。您告诉 Docker 根据当前目录的内容构建一个名为 kubia 的映像。 Docker 读取目录中的 Dockerfile，并根据文件中的指令构建镜像。

图2.11 使用Dockerfile构建新的容器镜像

构建本身不是由 docker CLI 工具执行的。相反，整个目录的内容都会上传到 Docker 守护进程，并由它构建镜像。您已经了解到 CLI 工具和守护程序不一定位于同一台计算机上。如果您在非 Linux 系统（例如 macOS 或 Windows）上使用 Docker，则客户端位于您的主机操作系统中，但守护进程在 Linux VM 内运行。但它也可以在远程计算机上运行。

TIP Don't add unnecessary files to the build directory, as they will slow down the build process—especially if the Docker daemon is located on a remote system.

To build the image, Docker first pulls the base image (`node:12`) from the public image repository (Docker Hub in this case), unless the image is already stored locally. It then creates a new container from the image and executes the next directive from the Dockerfile. The container's final state yields a new image with its own ID. The build process continues by processing the remaining directives in the Dockerfile. Each one creates a new image. The final image is then tagged with the tag you specified with the `-t` flag in the `docker build` command.

UNDERSTANDING WHAT THE LAYERS IN THE IMAGE ARE

Some pages ago, you learned that images consist of several layers. One might think that each image consists of only the layers of the base image and a single new layer on top, but that's not the case. When building an image, a new layer is created for each individual directive in the Dockerfile.

During the build of the kubia image, after it pulls all the layers of the base image, Docker creates a new layer and adds the `app.js` file into it. It then creates yet another layer that holds just the command to run when the image is executed. This last layer is then tagged as `kubia:latest`.

You can see the layers of an image and their size by running `docker history`, as shown in the following listing. The top layers are printed first.

Listing 2.6 Displaying the layers of a container image

```
$ docker history kubia:latest
IMAGE         CREATED     CREATED BY                                      SIZE
b0ecc49d7a1d  7 min ago   /bin/sh -c #(nop) ENTRYPOINT ["node"…   0B    #A
28d67701d6d9  7 min ago   /bin/sh -c #(nop) ADD file:2ed5d7753…   367B  #A
e498dabfee1c  2 days ago  /bin/sh -c #(nop) CMD ["node"]          0B    #B
<missing>     2 days ago  /bin/sh -c #(nop) ENTRYPOINT ["docke…   0B    #B
<missing>     2 days ago  /bin/sh -c #(nop) COPY file:23873730…   116B  #B
<missing>     2 days ago  /bin/sh -c set -ex && for key in 6A0…   5.4MB #B
<missing>     2 days ago  /bin/sh -c #(nop)  ENV YARN_VERSION=…   0B    #B
<missing>     2 days ago  /bin/sh -c ARCH= && dpkgArch="$(dpkg…   67MB  #B
<missing>     2 days ago  /bin/sh -c #(nop)  ENV NODE_VERSION=…   0B    #B
<missing>     3 weeks ago /bin/sh -c groupadd --gid 1000 node …   333kB #B
<missing>     3 weeks ago /bin/sh -c set -ex;  apt-get update;…   562MB #B
<missing>     3 weeks ago /bin/sh -c apt-get update && apt-get…   142MB #B
<missing>     3 weeks ago /bin/sh -c set -ex;  if ! command -v…   7.8MB #B
<missing>     3 weeks ago /bin/sh -c apt-get update && apt-get…   23.2MB#B
<missing>     3 weeks ago /bin/sh -c #(nop)  CMD ["bash"]         0B    #B
<missing>     3 weeks ago /bin/sh -c #(nop) ADD file:9788b61de…   101MB #B
```

#A The two layers that you added
#B The layers of the node:12 image and its base image(s)

Most of the layers you see come from the `node:12` image (they also include layers of that image's own base image). The two uppermost layers correspond to the second and third directives in the Dockerfile (`ADD` and `ENTRYPOINT`).

提示不要将不必要的文件添加到构建目录，因为它们会减慢构建过程 - 特别是在 Docker 守护进程位于远程系统上。

为了构建镜像，Docker 首先从公共镜像存储库（本例中为 Docker Hub）提取基础镜像 (node:12)，除非该镜像已存储在本地。然后，它从映像创建一个新容器，并执行 Dockerfile 中的下一个指令。容器的最终状态会生成一个具有自己 ID 的新映像。构建过程将继续处理 Dockerfile 中的剩余指令。每一个都创造了一个新的形象。然后，最终的映像将使用您在 docker build 命令中使用 -t 标志指定的标签进行标记。

了解图像中的各层

几页前，您了解到图像由多个层组成。人们可能认为每个图像仅由基础图像的层和顶部的单个新层组成，但事实并非如此。构建镜像时，会为 Dockerfile 中的每个单独指令创建一个新层。

在构建 kubia 镜像的过程中，在拉取基础镜像的所有层后，Docker 会创建一个新层并将 app.js 文件添加到其中。然后，它创建另一个层，仅保存执行图像时要运行的命令。最后一层被标记为 kubia:latest。

通过运行docker History可以看到镜像的层级和大小，如图
在下面的列表中。首先打印顶层。

清单 2.6 显示容器镜像的各层

```
$ docker 历史 kubia：最新创建的图像 按大小创建

b0ecc49d7a1d 7 分钟前 /bin/sh -c #(nop) ENTRYPOINT ["node"… 0B
#A

28d67701d6d9 7 分钟前 /bin/sh -c #(nop) 添加文件：2ed5d7753 …
367B #A

e498dabfee1c 2 天前 /bin/sh -c #(nop) CMD ["node"] 0B #B

2天前 /bin/sh -c #(nop) ENTRYPOINT ["docke... 0B #B

2 天前 /bin/sh -c #(nop) 复制文件：23873730 …116B #B

2天前 /bin/sh -c set -ex && for key in 6A0… 5.4MB #B

2 天前 /bin/sh -c #(nop) ENV YARN_VERSION =… 0B #B

2天前 /bin/sh -c ARCH= && dpkgArch="$(dpkg... 67MB #B
#A 您添加的两个层 #B 节点
的层 12 图像及其基础图像
2 天前 /bin/sh -c #(nop) ENV NODE_VERSION =… 0B #B
```

您看到的大多数图层都来自 node:12 图像（它们还包括该图像自己的基础图像的图层）。最上面的两个层对应于 Dockerfile 中的第二个和第三个指令（ADD 和 ENTRYPOINT）。

As you can see in the `CREATED BY` column, each layer is created by executing a command in the container. In addition to adding files with the `ADD` directive, you can also use other directives in the Dockerfile. For example, the `RUN` directive executes a command in the container during the build. In the listing above, you'll find a layer where the `apt-get update` and some additional `apt-get` commands were executed. `apt-get` is part of the Ubuntu package manager used to install software packages. The command shown in the listing installs some packages onto the image's filesystem.

To learn about RUN and other directives you can use in a Dockerfile, refer to the Dockerfile reference at https://docs.docker.com/engine/reference/builder/.

> **TIP** Each directive creates a new layer. I have already mentioned that when you delete a file, it is only marked as deleted in the new layer and is not removed from the layers below. Therefore, deleting a file with a subsequent directive won't reduce the size of the image. If you use the `RUN` directive, make sure that the command it executes deletes all temporary files it creates before it terminates.

2.2.5 Running the container image

With the image built and ready, you can now run the container with the following command:

```
$ docker run --name kubia-container -p 1234:8080 -d kubia
9d62e8a9c37e056a82bb1efad57789e947df58669f94adc2006c087a03c54e02
```

This tells Docker to run a new container called `kubia-container` from the `kubia` image. The container is detached from the console (`-d` flag) and runs in the background. Port `1234` on the host computer is mapped to port `8080` in the container (specified by the `-p 1234:8080` option), so you can access the app at http://localhost:1234.

The following figure should help you visualize how everything fits together. Note that the Linux VM exists only if you use macOS or Windows. If you use Linux directly, there is no VM and the box depicting port `1234` is at the edge of the local computer.

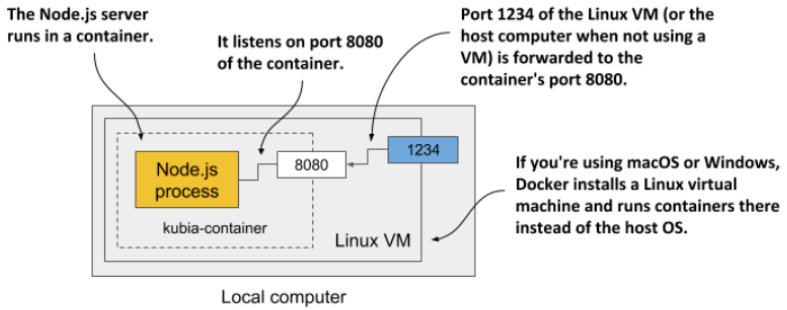


Figure 2.12 Visualizing your running container

正如您在 CREATED BY 列中看到的，每个层都是通过在容器中执行命令来创建的。除了使用 ADD 指令添加文件之外，您还可以在 Dockerfile 中使用其他指令。例如，RUN指令执行命令

构建期间的容器。在上面的列表中，您将找到执行 `apt-get update` 和一些附加 `apt-get` 命令的层。 `apt-get` 是 Ubuntu 包管理器的一部分，用于安装软件包。清单中显示的命令将一些软件包安装到映像的文件系统上。

要了解可在 Dockerfile 中使用的 RUN 和其他指令，请参阅

> 提示 每个指令都会创建一个新层。我已经提到过，当您删除文件时，它只会在新图层中标记为已删除，而不会从下面的图层中删除。因此，删除文件
> 后续指令不会减小图像的大小。如果您使用 RUN 指令，请确保
> 它执行的命令会在终止之前删除它创建的所有临时文件

2.2.5 运行容器镜像

构建并准备好映像后，您现在可以使用以下命令运行容器：

```
$ docker run --name kubia-container -p 1234:8080 -d
kubia
9d62e8a9c37e056a82bb1efad57789e947df58669f94adc2006
c087a03c54e02
```

这告诉 Docker 从 kubia 镜像运行一个名为 kubia-container 的新容器。这

容器与控制台分离（-d 标志）并在后台运行。端口 1234 开启

主机映射到容器中的端口 8080（由 `-p 1234:8080` 选项指定），因此您可以通过 `http://localhost:1234` 访问该应用程序。

下图应该可以帮助您直观地了解所有内容是如何组合在一起的。请注意，仅当您使用 macOS 或 Windows 时，Linux VM 才存在。如果直接使用Linux，则没有虚拟机，并且描绘端口1234的框位于本地计算机的边缘。

图 2

ACCESSING YOUR APP

Now access the application at http://localhost:1234 using `curl` or your internet browser:

```
$ curl localhost:1234
Hey there, this is 44d76963e8e1. Your IP is ::ffff:172.17.0.1.
```

NOTE If the Docker Daemon runs on a different machine, you must replace `localhost` with the IP of that machine. You can look it up in the `DOCKER_HOST` environment variable.

If all went well, you should see the response sent by the application. In my case, it returns `44d76963e8e1` as its hostname. In your case, you'll see a different hexadecimal number. This is the ID of the container that is displayed when you list them.

LISTING ALL RUNNING CONTAINERS

To list all the containers that are running on your computer, run the command that appears in the following listing. The output of the command has been edited to fit on the page—the last two lines of the output are the continuation of the first two.

Listing 2.6 Listing running containers

```
$ docker ps
CONTAINER ID    IMAGE           COMMAND          CREATED        ...
44d76963e8e1    kubia:latest    "node app.js"    6 minutes ago  ...

...  STATUS          PORTS                    NAMES
...  Up 6 minutes    0.0.0.0:1234->8080/tcp   kubia-container
```

For each container, Docker prints its ID and name, the image it uses, and the command it executes. It also shows when the container was created, what status it has, and which host ports are mapped to the container.

GETTING ADDITIONAL INFORMATION ABOUT A CONTAINER

The `docker ps` command shows the most basic information about the containers. To see additional information, you can use `docker inspect`:

```
$ docker inspect kubia-container
```

Docker prints a long JSON-formatted document containing a lot of information about the container, such as its state, config, and network settings, including its IP address.

INSPECTING THE APPLICATION LOG

Docker captures and stores everything the application writes to the standard output and error streams. This is typically the place where applications write their logs. You can use the `docker logs` command to see the output, as shown in the next listing.

Listing 2.7 Displaying the container's log

```
$ docker logs kubia-container
Kubia server starting...
```


42

访问您的应用程序

现在使用curl 或互联网浏览器访问http://localhost:1234 处的应用程序：

```
$ 卷曲本地主机：1234
```

注意如果 Docker Daemon 在不同的机器上运行，则必须将 localhost 替换为该机器的 IP 机器。您可以在 DOCKER_HOST 环境变量中查找它。

如果一切顺利，您应该会看到应用程序发送的响应。就我而言，它返回 44d76963e8e1 作为其主机名。在您的情况下，您会看到不同的十六进制数字。这

是列出它们时显示的容器的 ID。
列出所有正在运行的容器

要列出计算机上运行的所有容器，请运行以下列表中显示的命令。命令的输出已被编辑以适合页面 - 输出的最后两行是前两行的延续。

清单 2.6 列出正在运行的容器

```
$ docker ps 容器 ID 图像命令已创建
44d76963e8e1  kubia：最新的“节点app.js”   6 分钟前      ...
                                                       ...
... 状态端口名称
```

对于每个容器，Docker 都会打印它的 ID 和名称、它使用的镜像以及它执行的命令。它还显示容器的创建时间、容器的状态以及映射到容器的主机端口。

获取有关容器的附加信息

docker ps 命令显示有关容器的最基本信息。要查看更多信息，您可以使用 docker inform：

```
$ docker 检查 kubia 容器
```

Docker 打印一个长的 JSON 格式的文档，其中包含有关容器的大量信息，例如其状态、配置和网络设置（包括其 IP 地址）。

检查应用程序日志

Docker 捕获应用程序写入标准输出和错误流的所有内容并将其存储。这通常是应用程序写入日志的地方。您可以使用 docker log 命令查看输出，如下一个清单所示。

清单2.7 显示容器的日志

```
$ docker log kubia-
container Kubia 服务器正在
启动...
```

```
Local hostname is 44d76963e8e1
Listening on port 8080
Received request for / from ::ffff:172.17.0.1
```

You now know the basic commands for executing and inspecting an application in a container. Next, you'll learn how to distribute it.

2.2.6 Distributing container images

The image you've built is currently only available locally. To run it on other computers, you must first push it to an external image registry. Let's push it to the public Docker Hub registry, so that you don't need to set up a private one. You can also use other registries, such as Quay.io, which I've already mentioned, or the Google Container Registry.

Before you push the image, you must re-tag it according to Docker Hub's image naming schema. The image name must include your Docker Hub ID, which you choose when you register at http://hub.docker.com. I'll use my own ID (`luksa`) in the following examples, so remember to replace it with your ID when trying the commands yourself.

TAGGING AN IMAGE UNDER AN ADDITIONAL TAG

Once you have your ID, you're ready to add an additional tag `for` your image. Its current name is `kubia` and you'll now tag it also as `yourid/kubia:1.0` (replace `yourid` with your actual Docker Hub ID). This is the command I used:

```
$ docker tag kubia luksa/kubia:1.0
```

Confirm that your image now has two names by listing images again, as in the following listing.

Listing 2.7 A container image with multiple tags

```
$ docker images | head
REPOSITORY      TAG       IMAGE ID        CREATED              VIRTUAL SIZE
luksa/kubia     1.0       b0ecc49d7a1d    About an hour ago    908 MB
kubia           latest    b0ecc49d7a1d    About an hour ago    908 MB
node            12        e498dabfee1c    3 days ago           908 MB
```

As you can see, both `kubia` and `luksa/kubia:1.0` point to the same image ID, meaning that these aren't two images, but a single image with two names.

PUSHING THE IMAGE TO DOCKER HUB

Before you can push the image to Docker Hub, you must log in with your user ID using the `docker login` command as follows:

```
$ docker login -u yourid -p yourpassword docker.io
```

Once logged in, push the `yourid/kubia:1.0` image to Docker Hub with the following command:

```
$ docker push yourid/kubia:1.0
```



```
本地主机名是 44d76963e8e1 监听端口 8080
收到来自 ::ffff:172.17.0.1 的 / 请求
```

您现在已经了解了在容器中执行和检查应用程序的基本命令。接下来，您将学习如何分发它。

2.2.6 分发容器镜像

您构建的镜像目前仅在本地可用。要在其他计算机上运行它，您必须首先将其推送到外部映像注册表。让我们将其推送到公共 Docker Hub 注册表，这样您就无需设置私有注册表。您还可以使用其他注册表，例如我已经提到过的 Quay.io 或 Google 容器注册表。

在推送镜像之前，您必须根据 Docker Hub 的镜像命名架构重新标记它。镜像名称必须包含您在 http://hub.docker.com 注册时选择的 Docker Hub ID。我将在以下示例中使用我自己的 ID (luksa)，因此请记住在自己尝试命令时将其替换为您的 ID。

在附加标签下标记图像

获得 ID 后，您就可以为图像添加附加标签了。其目前的

名称是 kubia，现在您也可以将其标记为 yourid/kubia:1.0（将 yourid 替换为您的实际 Docker Hub ID）。这是我使用的命令：

```
$ docker 标签 kubia luksa/kubia:1.0
```

通过再次列出图像来确认您的图像现在有两个名称，如下列表所示。

清单 2.7 具有多个标签的容器镜像

```
$ docker 图像 |头 存储库标签 图像 ID 创建 虚拟尺寸

luksa/kubia 1.0 b0ecc49d7a1d 大约一小时前 908MB
kubia 最新 b0ecc49d7a1d 大约一小时前 908 MB
节点 12 e498dabfee1c 3天前 908MB
```

正如您所看到的，kubia 和 luksa/kubia:1.0 都指向相同的镜像 ID，这意味着它们不是两个镜像，而是具有两个名称的单个镜像。

将镜像推送到 Docker Hub

在将映像推送到 Docker Hub 之前，您必须使用 docker login 命令使用您的用户 ID 登录，如下所示：

```
$ docker 登录 -u 你的 ID -p 你的密码 docker.io
```

登录后，使用以下命令将 yourid/kubia:1.0 镜像推送到 Docker Hub：

```
$ docker Push yourid/kubia:1
```

RUNNING THE IMAGE ON OTHER HOSTS

When the push to Docker Hub is complete, the image is available to all. You can now run the image on any Docker-enabled host by running the following command:

```
$ docker run -p 1234:8080 -d luksa/kubia:1.0
```

If the container runs correctly on your computer, it should run on any other Linux computer, provided that the Node.js binary doesn't need any special Kernel features (it doesn't).

2.2.7 Stopping and deleting containers

If you've run the container on the other host, you can now terminate it, as you'll only need the one on your local computer for the exercises that follow.

STOPPING A CONTAINER

Instruct Docker to stop the container with this command:

```
$ docker stop kubia-container
```

This sends a termination signal to the main process in the container so that it can shut down gracefully. If the process doesn't respond to the termination signal or doesn't shut down in time, Docker kills it. When the top-level process in the container terminates, no other process runs in the container, so the container is stopped.

DELETING A CONTAINER

The container is no longer running, but it still exists. Docker keeps it around in case you decide to start it again. You can see stopped containers by running `docker ps -a`. The `-a` option prints all the containers - those running and those that have been stopped. As an exercise, you can start the container again by running `docker start kubia-container`.

You can safely delete the container on the other host, because you no longer need it. To delete it, run the following `docker rm` command:

```
$ docker rm kubia-container
```

This deletes the container. All its contents are removed and it can no longer be started. The image is still there, though. If you decide to create the container again, the image won't need to be downloaded again. If you also want to delete the image, use the `docker rmi` command:

```
$ docker rmi kubia:latest
```

To remove all dangling images, you can also use the `docker image prune` command.

2.3 Understanding what makes containers possible

You should keep the container running on your local computer so that you can use it in the following exercises, in which you'll examine how containers allow process isolation without

在其他主机上运行映像

当推送到 Docker Hub 完成后，该镜像就可供所有人使用。现在，您可以通过运行以下命令在任何启用 Docker 的主机上运行该映像：

```
$ docker run -p 1234:8080 -d luksa/kubia:1.0
```

如果容器在您的计算机上正确运行，它应该可以在任何其他 Linux 计算机上运行，前提是 Node.js 二进制文件不需要任何特殊的内核功能（它不需要）。

2.2.7 停止和删除容器

如果您已在另一台主机上运行容器，您现在可以终止它，因为您只需要本地计算机上的容器来进行后续练习。

停止容器

使用以下命令指示 Docker 停止容器：

```
$ docker stop kubia-container
```

这会向容器中的主进程发送终止信号，以便它可以正常关闭。如果进程没有响应终止信号或者没有及时关闭，Docker 就会杀死它。当容器中的顶层进程终止时，容器中没有其他进程运行，因此容器被停止。

删除容器

该容器不再运行，但仍然存在。 Docker 会保留它，以防您决定再次启动它。您可以通过运行 docker ps -a 来查看已停止的容器。 -a 选项打印所有容器 - 正在运行的容器和已停止的容器。作为练习，您可以通过运行 docker start kubia-container 再次启动容器。

您可以安全地删除其他主机上的容器，因为您不再需要它。到
删除它，运行以下 docker rm 命令：

```
$ docker rm kubia-container
```

这将删除容器。其所有内容均被删除，并且无法再启动。不过，图像仍然存在。如果您决定再次创建容器，则无需再次下载镜像。如果还想删除镜像，请使用 docker rmi 命令：

```
$ docker rmi kubia：最新
```

要删除所有悬空映像，您还可以使用 docker image prune 命令。

2.3 了解是什么让容器成为可能

您应该保持容器在本地计算机上运行，以便您可以在以下练习中使用它

using virtual machines. Several features of the Linux kernel make this possible and it's time to get to know them.

2.3.1 Using Namespaces to customize the environment of a process

The first feature called *Linux Namespaces* ensures that each process has its own view of the system. This means that a process running in a container will only see some of the files, processes and network interfaces on the system, as well as a different system hostname, just as if it were running in a separate virtual machine.

Initially, all the system resources available in a Linux OS, such as filesystems, process IDs, user IDs, network interfaces, and others, are all in the same bucket that all processes see and use. But the Kernel allows you to create additional buckets known as namespaces and move resources into them so that they are organized in smaller sets. This allows you to make each set visible only to one process or a group of processes. When you create a new process, you can specify which namespace it should use. The process only sees resources that are in this namespace and none in the other namespaces.

INTRODUCING THE AVAILABLE NAMESPACE TYPES

More specifically, there isn't just a single type of namespace. There are in fact several types – one for each resource type. A process thus uses not only one namespace, but one namespace for each type.

The following types of namespaces exist:

- The Mount namespace (mnt) isolates mount points (file systems).
- The Process ID namespace (pid) isolates process IDs.
- The Network namespace (net) isolates network devices, stacks, ports, etc.
- The Inter-process communication namespace (ipc) isolates the communication between processes (this includes isolating message queues, shared memory, and others).
- The UNIX Time-sharing System (UTS) namespace isolates the system hostname and the Network Information Service (NIS) domain name.
- The User ID namespace (user) isolates user and group IDs.
- The Cgroup namespace isolates the Control Groups root directory. You'll learn about cgroups later in this chapter.

USING NETWORK NAMESPACES TO GIVE A PROCESS A DEDICATED SET OF NETWORK INTERFACES

The network namespace in which a process runs determines what network interfaces the process can see. Each network interface belongs to exactly one namespace but can be moved from one namespace to another. If each container uses its own network namespace, each container sees its own set of network interfaces.

Examine figure 2.13 for a better overview of how network namespaces are used to create a container. Imagine you want to run a containerized process and provide it with a dedicated set of network interfaces that only that process can use.

使用虚拟机。 Linux 内核的几个功能使这成为可能，现在是时候了解它们了。

2.3.1 使用命名空间定制进程的环境

第一个称为 Linux 命名空间的功能可确保每个进程都有自己的系统视图。这意味着在容器中运行的进程只会看到系统上的部分文件、进程和网络接口，以及不同的系统主机名，就像在单独的虚拟机中运行一样。

最初，Linux 操作系统中可用的所有系统资源（例如文件系统、进程 ID、用户 ID、网络接口等）都位于所有进程看到和使用的同一个存储桶中。但内核允许您创建称为命名空间的附加存储桶，并将资源移入其中，以便将它们组织成更小的集合。这允许您使每个集合仅对一个进程或一组进程可见。创建新进程时，您可以指定它应使用哪个命名空间。该进程只能看到此命名空间中的资源，而看不到其他命名空间中的资源。

介绍可用的命名空间类型

更具体地说，不只有单一类型的命名空间。事实上有多种类型——每种资源类型都有一种类型。因此，进程不仅使用一个名称空间，而且还为每种类型使用一个名称空间。

存在以下类型的命名空间：

- 安装命名空间(mnt) 隔离安装点（文件系统）。
- 进程ID 命名空间(pid) 隔离进程ID。
- 网络命名空间(net) 隔离网络设备、堆栈、端口等。
  进程之间（这包括隔离消息队列、共享内存等）。
- UNIX 分时系统(UTS) 命名空间隔离系统主机名和网络信息服务(NIS) 域名。
- 用户ID 命名空间（用户）隔离用户ID 和组ID。
- Cgroup 命名空间隔离控制组根目录。您将在本章后面了解 cgroup。

使用网络命名空间为进程提供一组专用的网络接口

进程运行的网络命名空间决定了该进程可以看到哪些网络接口。每个网络接口都属于一个命名空间，但可以从一个命名空间移动到另一个命名空间。如果每个容器都使用自己的网络命名空间，则每个容器都会看到自己的一组网络接口。

查看图 2.13，可以更好地了解如何使用网络命名空间创建容器。想象一下，您想要运行一个容器化进程并为其提供一组专用的网络接口，只有该进程才能使用。

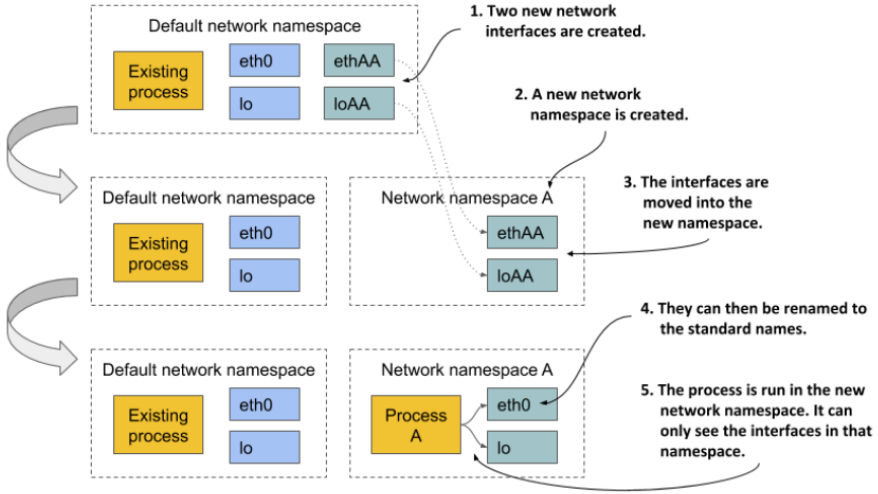


Figure 2.13 The network namespace limits which network interfaces a process uses

Initially, only the default network namespace exists. You then create two new network interfaces for the container and a new network namespace. The interfaces can then be moved from the default namespace to the new namespace. Once there, they can be renamed, because names must only be unique in each namespace. Finally, the process can be started in this network namespace, which allows it to only see the two interfaces that were created for it.

By looking solely at the available network interfaces, the process can't tell whether it's in a container or a VM or an OS running directly on a bare-metal machine.

Using the UTS namespace to give a process a dedicated hostname

Another example of how to make it look like the process is running on its own host is to use the UTS namespace. It determines what hostname and domain name the process running inside this namespace sees. By assigning two different UTS namespaces to two different processes, you can make them see different system hostnames. To the two processes, it looks as if they run on two different computers.

Understanding how namespaces isolate processes from each other

By creating a dedicated namespace instance for all available namespace types and assigning it to a process, you can make the process believe that it's running in its own OS. The main reason for this is that each process has its own environment. A process can only see and use the resources in its own namespaces. It can't use any in other namespaces. Likewise, other

图2.13 网络命名空间限制进程使用哪些网络接口

最初，仅存在默认网络命名空间。然后，您为容器创建两个新的网络接口和一个新的网络命名空间。然后可以将接口从默认命名空间移动到新命名空间。一旦到达那里，就可以对它们进行重命名，因为名称在每个名称空间中必须是唯一的。最后，可以在此网络命名空间中启动该进程，这使得它只能看到为其创建的两个接口。

仅通过查看可用的网络接口，该进程无法判断它是在容器中还是在虚拟机中，或者是直接在裸机上运行的操作系统中。

使用 UTS 命名空间为进程提供专用主机名

如何使进程看起来像是在自己的主机上运行的另一个示例是使用 UTS 命名空间。它确定在此命名空间内运行的进程看到的主机名和域名。通过将两个不同的 UTS 命名空间分配给两个不同的进程，您可以使它们看到不同的系统主机名。对于这两个进程来说，它们看起来好像运行在两台不同的计算机上。

了解命名空间如何相互隔离进程

通过为所有可用的命名空间类型创建专用命名空间实例并将其分配给进程，您可以使该进程相信它正在自己的操作系统中运行。主要原因是每个进程都有自己的环境。进程只能查看和使用自己命名空间中的资源。它不能在其他命名空间中使用任何内容。同样地

processes can't use its resources either. This is how containers isolate the environments of the processes that run within them.

SHARING NAMESPACES BETWEEN MULTIPLE PROCESSES

In the next chapter you'll learn that you don't always want to isolate the containers completely from each other. Related containers may want to share certain resources. The following figure shows an example of two processes that share the same network interfaces and the host and domain name of the system, but not the file system.

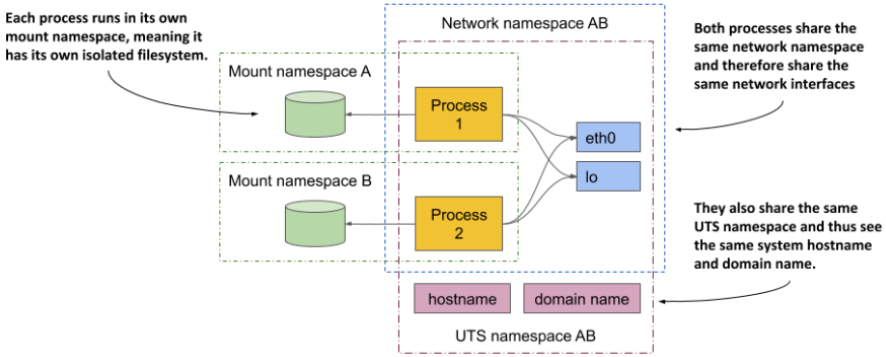


Figure 2.14 Each process is associated with multiple namespace types, some of which can be shared.

Concentrate on the shared network devices first. The two processes see and use the same two devices (`eth0` and `lo`) because they use the same network namespace. This allows them to bind to the same IP address and communicate through the loopback device, just as they could if they were running on a machine that doesn't use containers. The two processes also use the same UTS namespace and therefore see the same system host name. In contrast, they each use their own mount namespace, which means they have separate file systems.

In summary, processes may want to share some resources but not others. This is possible because separate namespace *types* exist. A process has an associated namespace for each type.

In view of all this, one might ask what is a container at all? A process that runs "in a container" doesn't run in something that resembles a real enclosure like a VM. It's only a process to which seven namespaces (one for each type) are assigned. Some are shared with other processes, while others are not. This means that the boundaries between the processes do not all fall on the same line.

In a later chapter, you'll learn how to debug a container by running a new process directly on the host OS, but using the network namespace of an existing container, while using the host's default namespaces for everything else. This will allow you to debug the

进程也不能使用其资源。这就是容器隔离在其中运行的进程的环境的方式。

在多个进程之间共享命名空间

在下一章中，您将了解到您并不总是希望将容器彼此完全隔离。相关容器可能想要共享某些资源。下图显示了两个进程共享相同网络接口以及系统主机名和域名但不共享文件系统的示例。

图2.14 每个进程都与多个命名空间类型相关联，其中一些命名空间类型可以共享。

首先关注共享网络设备。这两个进程看到并使用相同的两个设备（eth0 和 lo），因为它们使用相同的网络命名空间。这允许它们绑定到相同的 IP 地址并通过环回设备进行通信，就像它们在不使用容器的计算机上运行一样。这两个进程还使用相同的 UTS 命名空间，因此看到相同的系统主机名。相反，它们各自使用自己的挂载命名空间，这意味着它们具有单独的文件系统。

总之，进程可能希望共享一些资源，但不想共享其他资源。这是可能的，因为存在单独的命名空间类型。进程的每种类型都有一个关联的命名空间。

鉴于这一切，人们可能会问容器到底是什么？ "在容器中"运行的进程不会在类似于虚拟机等真实外壳的东西中运行。它只是一个分配了七个命名空间（每种类型一个）的进程。有些与其他进程共享，而另一些则不然。这意味着进程之间的边界并不都落在同一条线上。

在后面的章节中，您将学习如何通过直接在主机操作系统上运行新进程来调试容器，但使用现有容器的网络命名空间，同时使用主机的默认命名空间进行其他操作

container's networking system with tools available on the host that may not be available in the container.

2.3.2 Exploring the environment of a running container

What if you want to see what the environment inside the container looks like? What is the system host name, what is the local IP address, what binaries and libraries are available on the file system, and so on?

To explore these features in the case of a VM, you typically connect to it remotely via ssh and use a shell to execute commands. This process is very similar for containers. You run a shell inside the container.

> **NOTE** The shell's executable file must be available in the container's file system. This isn't always the case with containers running in production.

RUNNING A SHELL INSIDE AN EXISTING CONTAINER

The Node.js image on which your image is based provides the bash shell, meaning you can run it in the container with the following command:

```
$ docker exec -it kubia-container bash
root@44d76963e8e1:/#                #A
```

#A This is the shell's command prompt

This command runs `bash` as an additional process in the existing `kubia-container` container. The process has the same Linux namespaces as the main container process (the running Node.js server). This way you can explore the container from within and see how Node.js and your app see the system when running in the container. The `-it` option is shorthand for two options:

- `-i` tells Docker to run the command in interactive mode.
- `-t` tells it to allocate a pseudo terminal (TTY) so you can use the shell properly.

You need both if you want to use the shell the way you're used to. If you omit the first, you can't execute any commands, and if you omit the second, the command prompt doesn't appear and some commands may complain that the `TERM` variable is not set.

LISTING RUNNING PROCESSES IN A CONTAINER

Let's list the processes running in the container by executing the `ps aux` command inside the shell you ran in the container. The following listing shows the command's output.

Listing 2.8 Listing processes running in the container

```
root@44d76963e8e1:/# ps aux
USER  PID %CPU %MEM    VSZ   RSS TTY STAT START TIME COMMAND
root    1  0.0  0.1 676380 16504 ?   Sl   12:31 0:00 node app.js
root   10  0.0  0.0  20216  1924 ?   Ss   12:31 0:00 bash
root   19  0.0  0.0  17492  1136 ?   R+   12:38 0:00 ps aux
```


容器的网络系统，具有主机上可用但容器中可能不可用的工具。

2.3.2 探索运行容器的环境

如果你想看看容器内部的环境是什么样的呢？系统主机名是什么，本地 IP 地址是什么，文件系统上有哪些可用的二进制文件和库，等等？

要在虚拟机的情况下探索这些功能，您通常通过 ssh 远程连接到它并使用 shell 执行命令。这个过程与容器非常相似。您在容器内运行一个 shell。

> 注意 shell 的可执行文件必须在容器的文件系统中可用。在生产环境中运行的容器并非总是如

在现有容器内运行 shell

您的映像所基于的 Node.js 映像提供了 bash shell，这意味着您可以使用以下命令在容器中运行它：

```
$ docker exec -it kubia-container
bash root@44d76963e8e1 :/# #A
```

#A 这是 shell 的命令提示符

此命令将 bash 作为现有 kubia-container 容器中的附加进程运行。该进程与主容器进程（正在运行的 Node.js 服务器）具有相同的 Linux 命名空间。通过这种方式，您可以从内部探索容器，并了解 Node.js 和您的应用程序在容器中运行时如何看待系统。 -it 选项是两个选项的简写：

- -i 告诉Docker 以交互模式运行命令。
- -t 告诉它分配一个伪终端(TTY)，以便您可以正确使用shell

如果您想按照习惯的方式使用 shell，则两者都需要。如果省略第一个，则无法执行任何命令，如果省略第二个，则不会出现命令提示符，并且某些命令可能会抱怨未设置 TERM 变量。

列出容器中正在运行的进程

让我们通过在容器中运行的 shell 中执行 ps aux 命令来列出容器中运行的进程。以下清单显示了该命令的输出。

清单 2.8 列出容器中运行的进程

```
root@44d76963e8e1 :/# ps aux USER PID %CPU %MEM VSZ
RSS TTY STAT 启动时间命令
根 1 0.0 0.1 676380 16504 ? 斯尔 12:31 0:00 节点 app.js
根 10 0.0 0.0 20216 1924 ? SS 12:31 0:00 狂欢
根 19 0.0 0.0 17492 第1136章右+ 12:38 0:00 ps 辅助
```

The list shows only three processes. These are the only ones that run in the container. You can't see the other processes that run in the host OS or in other containers because the container runs in its own Process ID namespace.

SEEING CONTAINER PROCESSES IN THE HOST'S LIST OF PROCESSES

If you now open another terminal and list the processes in the host OS itself, you *will* also see the processes that run in the container. This confirms that the processes in the container are in fact regular processes that run in the host OS, as you can see in the following listing.

Listing 2.9 A container's processes run in the host OS

```
$ ps aux | grep app.js
USER  PID %CPU %MEM    VSZ   RSS TTY STAT START TIME COMMAND
root  382  0.0  0.1 676380 16504 ?   Sl   12:31 0:00 node app.js
```

NOTE If you use macOS or Windows, you must list the processes in the VM that hosts the Docker daemon, as that's where your containers run. In Docker Desktop, you can enter the VM using the following command:

```
docker run --net=host --ipc=host --uts=host --pid=host -it --security-
opt=seccomp=unconfined --privileged --rm -v /:/host alpine chroot /host
```

If you have a sharp eye, you may notice that the process IDs in the container are different from those on the host. Because the container uses its own Process ID namespace it has its own process tree with its own ID number sequence. As the next figure shows, the tree is a subtree of the host's full process tree. Each process thus has two IDs.

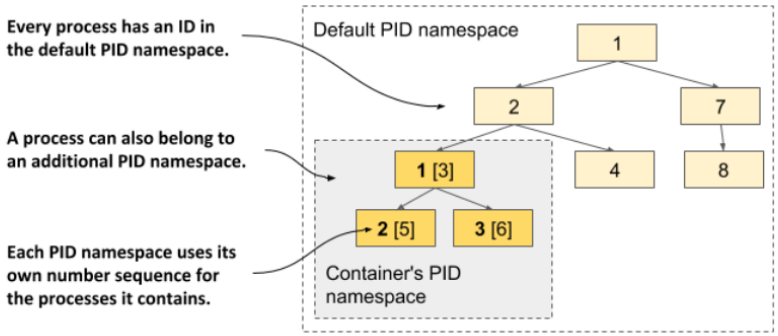


Figure 2.15 The PID namespace makes a process sub-tree appear as a separate process tree with its own numbering sequence

该列表仅显示三个进程。这些是唯一在容器中运行的。您无法看到在主机操作系统或其他容器中运行的其他进程，因为容器在其自己的进程 ID 命名空间中运行。

在主机进程列表中查看容器进程

如果您现在打开另一个终端并列出主机操作系统本身中的进程，您还将看到容器中运行的进程。这证实了容器中的进程实际上是在主机操作系统中运行的常规进程，如下面的清单所示。

清单 2.9 容器的进程在主机操作系统中运行

```
$ ps 辅助 | grep app.js USER PID %CPU %MEM VSZ RSS
TTY STAT 开始时间命令
根 382 0.0 0.1 676380 16504 ？斯尔 12:31 0:00 节点 app.js
```

注意如果您使用 macOS 或 Windows，则必须列出托管 Docker 守护程序的虚拟机中的进程，因为容器就是在其中运行的。在 Docker Desktop 中，可以使用以下命令进入虚拟机：

```
docker run --net=主机 --ipc=主机 --uts=主机 --pid=主机 -it --security-
opt=seccomp=unconfined --privileged --rm -v /:/host alpine chroot /host
```

如果你眼光敏锐，你可能会注意到容器中的进程ID与主机上的进程ID不同。因为容器使用自己的进程 ID 命名空间，所以它有自己的进程树和自己的 ID 编号序列。如下图所示，该树是主机完整进程树的子树。因此每个进程都有两个 ID。

图 2

THE CONTAINER'S FILESYSTEM IS ISOLATED FROM THE HOST AND OTHER CONTAINERS

As with an isolated process tree, each container also has an isolated filesystem. If you list the contents of the container's root directory, only the files in the container are displayed. This includes files from the container image and files created during container operation, such as log files. The next listing shows the files in your kubia container.

Listing 2.10 A container has its own filesystem

```
root@44d76963e8e1:/# ls /
app.js  boot  etc   lib    media  opt   root  sbin  sys  usr
bin     dev   home  lib64  mnt    proc  run   srv   tmp  var
```

It contains the `app.js` file and other system directories that are part of the `node:12` base image. You are welcome to browse the container's filesystem. You'll see that there is no way to view files from the host's filesystem. This is great, because it prevents a potential attacker from gaining access to them through vulnerabilities in the Node.js server.

To leave the container, leave the shell by running the `exit` command or pressing Control-D and you'll be returned to your host computer (similar to logging out from an `ssh` session).

> **TIP** Entering a running container like this is useful when debugging an app running in a container. When something breaks, the first thing you'll want to investigate is the actual state of the system your application sees.

2.3.3 Limiting a process' resource usage with Linux Control Groups

Linux Namespaces make it possible for processes to access only some of the host's resources, but they don't limit how much of a single resource each process can consume. For example, you can use namespaces to allow a process to access only a particular network interface, but you can't limit the network bandwidth the process consumes. Likewise, you can't use namespaces to limit the CPU time or memory available to a process. You may want to do that to prevent one process from consuming all the CPU time and preventing critical system processes from running properly. For that, we need another feature of the Linux kernel.

INTRODUCING CGROUPS

The second Linux kernel feature that makes containers possible is called *Linux Control Groups (cgroups)*. It limits, accounts for and isolates system resources such as CPU, memory and disk or network bandwidth. When using cgroups, a process or group of processes can only use the allotted CPU time, memory, and network bandwidth for example. This way, processes cannot occupy resources that are reserved for other processes.

At this point, you don't need to know how Control Groups do all this, but it may be worth seeing how you can ask Docker to limit the amount of CPU and memory a container can use.

LIMITING A CONTAINER'S USE OF THE CPU

If you don't impose any restrictions on the container's use of the CPU, it has unrestricted access to all CPU cores on the host. You can explicitly specify which cores a container can

容器的文件系统与主机和其他容器隔离

与隔离的进程树一样，每个容器也有一个隔离的文件系统。如果列出容器根目录的内容，则仅显示容器中的文件。这包括容器映像中的文件以及容器操作期间创建的文件，例如日志文件。下一个列表显示了 kubia 容器中的文件。

清单 2.10 容器有自己的文件系统

```
root@44d76963e8e1 :/# ls/app.js boot etc lib media
opt root sbin sys usr
```

它包含 app.js 文件和属于 node:12 基础映像一部分的其他系统目录。欢迎您浏览容器的文件系统。您会发现无法从主机的文件系统中查看文件。这很棒，因为它可以防止潜在的攻击者通过 Node.js 服务器中的漏洞访问它们。

要离开容器，请通过运行 exit 命令或按 Control- 离开 shell D，您将返回到主机（类似于从 ssh 会话注销）。

提示 在调试容器中运行的应用程序时，像这样输入正在运行的容器非常有用。当出现问题时，您首先要调查的是应用程序系统的实际状态
看到了。

2.3.3 使用 Linux 控制组限制进程的资源使用

Linux 命名空间使进程可以仅访问主机的部分资源，但它们并不限制每个进程可以消耗的单个资源的数量。例如，您可以使用命名空间来允许进程仅访问特定的网络接口，但无法限制进程消耗的网络带宽。同样，您不能使用命名空间来限制进程可用的 CPU 时间或内存。您可能希望这样做是为了防止一个进程消耗所有 CPU 时间并阻止关键系统进程正常运行。为此，我们需要 Linux 内核的另一个功能。

集团简介

使容器成为可能的第二个 Linux 内核功能称为 Linux 控制组 (cgroups)。它限制、考虑和隔离系统资源，例如 CPU、内存和磁盘或网络带宽。例如，使用 cgroup 时，一个进程或一组进程只能使用分配的 CPU 时间、内存和网络带宽。这样，进程就不能占用为其他进程保留的资源。

此时，您不需要知道控制组如何完成所有这些，但可能值得看看如何要求 Docker 限制容器可以使用的 CPU 和内存量。

限制容器对 CPU 的使用

如果不对容器对CPU的使用施加任何限制，则它可以不受限制地访问主机上的所有CPU核心

use with Docker's `--cpuset-cpus` option. For example, to allow the container to only use cores one and two, you can run the container with the following option:

```
$ docker run --cpuset-cpus="1,2" ...
```

You can also limit the available CPU time using options `--cpus`, `--cpu-period`, `--cpu-quota` and `--cpu-shares`. For example, to allow the container to use only half of a CPU core, run the container as follows:

```
$ docker run --cpus="0.5" ...
```

LIMITING A CONTAINER'S USE OF MEMORY

As with CPU, a container can use all the available system memory, just like any regular OS process, but you may want to limit this. Docker provides the following options to limit container memory and swap usage: `--memory`, `--memory-reservation`, `--kernel-memory`, `--memory-swap`, and `--memory-swappiness`.

For example, to set the maximum memory size available in the container to 100MB, run the container as follows (`m` stands for megabyte):

```
$ docker run --memory="100m" ...
```

Behind the scenes, all these Docker options merely configure the cgroups of the process. It's the Kernel that takes care of limiting the resources available to the process. See the Docker documentation for more information about the other memory and CPU limit options.

2.3.4 Strengthening isolation between containers

Linux Namespaces and Cgroups separate the containers' environments and prevent one container from starving the other containers of compute resources. But the processes in these containers use the same system kernel, so we can't say that they are really isolated. A rogue container could make malicious system calls that would affect its neighbours.

Imagine a Kubernetes node on which several containers run. Each has its own network devices and files and can only consume a limited amount of CPU and memory. At first glance, a rogue program in one of these containers can't cause damage to the other containers. But what if the rogue program modifies the system clock that is shared by all containers?

Depending on the application, changing the time may not be too much of a problem, but allowing programs to make any system call to the kernel allows them to do virtually anything. Sys-calls allow them to modify the kernel memory, add or remove kernel modules, and many other things that regular containers aren't supposed to do.

This brings us to the third set of technologies that make containers possible. Explaining them fully is outside the scope of this book, so please refer to other resources that focus specifically on containers or the technologies used to secure them. This section provides a brief introduction to these technologies.

与 Docker 的 --cpuset-cpus 选项一起使用。例如，要允许容器仅使用核心一和二，您可以使用以下选项运行容器：

```
$ docker run --cpuset-cpus="1,2" ...
```

您还可以使用选项 --cpus、--cpu-period、--cpu-quota 和 --cpu-shares 限制可用 CPU 时间。例如，要允许容器仅使用一半的 CPU 核心，请按如下方式运行容器：

```
$ docker run --cpus="0.5" ...
```

限制容器的内存使用

与 CPU 一样，容器可以使用所有可用的系统内存，就像任何常规操作系统进程一样，但您可能希望限制这一点。 Docker 提供以下选项来限制容器内存和交换使用：--memory、--memory-reservation、--kernel-memory、-memory-swap 和 --memory-swappiness。

例如，要将容器中可用的最大内存大小设置为 100MB，请运行容器如下（m代表兆字节）：

```
$ docker run --内存="100m" ...
```

在幕后，所有这些 Docker 选项仅配置进程的 cgroup。内核负责限制进程可用的资源。有关其他内存和 CPU 限制选项的更多信息，请参阅 Docker 文档。

2.3.4 加强容器之间的隔离

Linux 命名空间和 Cgroup 分隔了容器的环境，并防止一个容器导致其他容器的计算资源不足。但这些容器中的进程使用相同的系统内核，所以我们不能说它们是真正隔离的。恶意容器可能会发出恶意系统调用，从而影响其邻居。

想象一个运行多个容器的 Kubernetes 节点。每个都有自己的网络设备和文件，并且只能消耗有限的CPU和内存。乍一看，其中一个容器中的恶意程序不会对其他容器造成损坏。但是如果流氓程序修改了所有容器共享的系统时钟怎么办？

根据应用程序的不同，更改时间可能不会有太大问题，但允许程序对内核进行任何系统调用实际上允许它们执行任何操作。系统调用允许它们修改内核内存、添加或删除内核模块以及许多其他常规容器不应该执行的操作。

这给我们带来了使容器成为可能的第三组技术。全面解释它们超出了本书的范围，因此请参阅专门关注容器或用于保护容器的技术的其他资源。本节简要介绍这些技术。

GIVING CONTAINERS FULL PRIVILEGES TO THE SYSTEM

The operating system kernel provides a set of *sys-calls* that programs use to interact with the operating system and underlying hardware. These includes calls to create processes, manipulate files and devices, establish communication channels between applications, and so on.

Some of these sys-calls are fairly safe and available to any process, but others are reserved for processes with elevated privileges only. If you look at the example presented earlier, applications running on the Kubernetes node should be allowed to open their local files, but not change the system clock or modify the kernel in a way that breaks the other containers.

Most containers should run without elevated privileges. Only those programs that you trust and that actually need the additional privileges should run in privileged containers.

> **NOTE** With Docker you create a privileged container by using the `--privileged` flag.

USING CAPABILITIES TO GIVE CONTAINERS A SUBSET OF ALL PRIVILEGES

If an application only needs to invoke some of the sys-calls that require elevated privileges, creating a container with full privileges is not ideal. Fortunately, the Linux kernel also divides privileges into units called *capabilities*. Examples of capabilities are:

- `CAP_NET_ADMIN` allows the process to perform network-related operations,
- `CAP_NET_BIND_SERVICE` allows it to bind to port numbers less than 1024,
- `CAP_SYS_TIME` allows it to modify the system clock, and so on.

Capabilities can be added or removed (*dropped*) from a container when you create it. Each capability represents a set of privileges available to the processes in the container. Docker and Kubernetes drop all capabilities except those required by typical applications, but users can add or drop other capabilities if authorized to do so.

> **NOTE** Always follow the *principle of least privilege* when running containers. Don't give them any capabilities that they don't need. This prevents attackers from using them to gain access to your operating system.

USING SECCOMP PROFILES TO FILTER INDIVIDUAL SYS-CALLS

If you need even finer control over what sys-calls a program can make, you can use *seccomp* (Secure Computing Mode). You can create a custom seccomp profile by creating a JSON file that lists the sys-calls that the container using the profile is allowed to make. You then provide the file to Docker when you create the container.

HARDENING CONTAINERS USING APPARMOR AND SELINUX

And as if the technologies discussed so far weren't enough, containers can also be secured with two additional mandatory access control (MAC) mechanisms: SELinux (Security-Enhanced Linux) and AppArmor (Application Armor).

52

赋予容器系统完整的权限

操作系统内核提供了一组系统调用，程序使用它们与操作系统和底层硬件进行交互。其中包括创建进程、操作文件和设备、在应用程序之间建立通信通道等的调用。

其中一些系统调用相当安全并且可供任何进程使用，但其他系统调用仅为具有提升权限的进程保留。如果您查看前面提供的示例，应该允许在 Kubernetes 节点上运行的应用程序打开其本地文件，但不得更改系统时钟或以破坏其他容器的方式修改内核。

大多数容器应该在没有提升权限的情况下运行。只有那些您信任且实际需要额外权限的程序才应在特权容器中运行。

注意使用 Docker，您可以使用 --privileged 标志创建特权容器。

使用功能为容器提供所有权限的子集

如果应用程序只需要调用一些需要提升权限的系统调用，那么创建具有完整权限的容器并不理想。幸运的是，Linux 内核还将权限划分为称为功能的单元。能力的例子有：

- CAP_NET_ADMIN 允许进程执行与网络相关的操作，
- CAP_NET_BIND_SERVICE 允许其绑定到小于 1024 的端口号，
- CAP_SYS_TIME 允许修改系统时钟等

创建容器时可以添加或删除（删除）容器的功能。每个功能代表容器中进程可用的一组权限。Docker 和 Kubernetes 删除了除典型应用程序所需的所有功能，但用户可以在获得授权的情况下添加或删除其他功能。

注意 运行容器时始终遵循最小权限原则。不要给他们任何他们不需要的能力。这可以防止攻击者使用它们来访问您的操作
系统。

使用 SECCOMP 配置文件过滤单个系统调用

如果您需要更好地控制程序可以进行的系统调用，您可以使用 seccomp（安全计算模式）。您可以通过创建一个 JSON 文件来创建自定义 seccomp 配置文件，该文件列出了允许使用该配置文件的容器进行的系统调用。然后，您在创建容器时将该文件提供给 Docker。

使用 Apparmor 和 SELINUX 强化容器

好像到目前为止讨论的技术还不够，容器还可以通过两个额外的强制访问控制 (MAC) 机制来保护：SELinux (SecurityEnhanced Linux) 和 AppArmor (Application Armor)。

With SELinux, you attach labels to files and system resources, as well as to users and processes. A user or process can only access a file or resource if the labels of all subjects and objects involved match a set of policies. AppArmor is similar but uses file paths instead of labels and focuses on processes rather than users.

Both SELinux and AppArmor considerably improve the security of an operating system, but don't worry if you are overwhelmed by all these security-related mechanisms. The aim of this section was to shed light on everything involved in the proper isolation of containers, but a basic understanding of namespaces should be more than sufficient for the moment.

2.4 Summary

If you were new to containers before reading this chapter, you should now understand what they are, why we use them, and what features of the Linux kernel make them possible. If you have previously used containers, I hope this chapter has helped to clarify your uncertainties about how containers work, and you now understand that they're nothing more than regular OS processes that the Linux kernel isolates from other processes.

After reading this chapter, you should know that:

- Containers are regular processes, but isolated from each other and the other processes running in the host OS.
- Containers are much lighter than virtual machines, but because they use the same Linux kernel, they aren't as isolated as VMs.
- Docker was the first container platform to make containers popular and the first container runtime supported by Kubernetes. Now, others are supported through the Container Runtime Interface (CRI).
- A container image contains the user application and all its dependencies. It is distributed through a container registry and used to create running containers.
- Containers can be downloaded and executed with a single `docker run` command.
- Docker builds an image from a `Dockerfile` that contains commands that Docker should execute during the build process. Images consist of layers that can be shared between multiple images. Each layer only needs to be transmitted and stored once.
- Containers are isolated by Linux kernel features called Namespaces, Control groups, Capabilities, seccomp, AppArmor and/or SELinux. Namespaces ensure that a container sees only a part of the resources available on the host, Control groups limit the amount of a resource it can use, while other features strengthen the isolation between containers.

After inspecting the containers on this ship, you're now ready to raise the anchor and sail into the next chapter, where you'll learn about running containers with Kubernetes.

53

使用 SELinux，您可以将标签附加到文件和系统资源以及用户和进程。仅当涉及的所有主题和客体的标签与一组策略匹配时，用户或进程才能访问文件或资源。 AppArmor 类似，但使用文件路径而不是标签，并且关注进程而不是用户。

SELinux 和 AppArmor 都显着提高了操作系统的安全性，但如果您对所有这些安全相关机制感到不知所措，请不要担心。本节的目的是阐明容器正确隔离所涉及的所有内容，但目前对命名空间的基本了解应该足够了。

2.4 总结

如果您在阅读本章之前对容器不熟悉，那么您现在应该了解它们是什么、我们为什么使用它们以及 Linux 内核的哪些功能使它们成为可能。如果您以前使用过容器，我希望本章能够帮助您澄清对容器如何工作的不确定性，并且您现在明白它们只不过是 Linux 内核与其他进程隔离的常规操作系统进程。读完本章后，您应该知道：

- 容器是常规进程，但彼此之间以及与主机操作系统中运行的其他进程相互隔离。
- 容器比虚拟机轻得多，但由于它们使用相同的Linux 内核，因此它们不像虚拟机那样隔离。
- Docker 是第一个使容器流行的容器平台，也是第一个受 Kubernetes 支持的容器运行时。现在，其他功能是通过容器运行时接口 (CRI) 支持的。
- 容器映像包含用户应用程序及其所有依赖项。它通过容器注册表进行分发并用于创建正在运行的容器。
- 可以使用单个docker run 命令下载并执行容器。
- Docker 从Dockerfile 构建映像，其中包含Docker 在构建过程中应执行的命令。图像由可以在多个图像之间共享的层组成。每层只需要传输和存储一次。
- 容器通过称为命名空间、控制组、功能、seccomp、AppArmor 和/或SELinux 的Linux 内核功能进行隔离。命名空间确保 a 容器只能看到主机上可用的部分资源，控制组限制了它可以使用的资源量，而其他功能则加强了容器之间的隔离。

检查完这艘船上的容器后，您现在可以起锚并航行到下一章，您将在其中了解如何使用 Kubernetes 运行容器。

3

Deploying your first application

This chapter covers

- **Running a single-node Kubernetes cluster on your laptop**
- **Setting up a Kubernetes cluster on Google Kubernetes Engine**
- **Setting up and using the `kubectl` command-line tool**
- **Deploying an application in Kubernetes and making it available across the globe**
- **Horizontally scaling the application**

The goal of this chapter is to show you how to run a local single-node development Kubernetes cluster or set up a proper, managed multi-node cluster in the cloud. Once your cluster is running, you'll use it to run the container you created in the previous chapter.

3.1 Deploying a Kubernetes cluster

Setting up a full-fledged, multi-node Kubernetes cluster isn't a simple task, especially if you're not familiar with Linux and network administration. A proper Kubernetes installation spans multiple physical or virtual machines and requires proper network setup to allow all containers in the cluster to communicate with each other.

You can install Kubernetes on your laptop computer, on your organization's infrastructure, or on virtual machines provided by cloud providers (Google Compute Engine, Amazon EC2, Microsoft Azure, and so on). Alternatively, most cloud providers now offer managed Kubernetes services, saving you from the hassle of installation and management. Here's a short overview of what the largest cloud providers offer:

- Google offers GKE - Google Kubernetes Engine,
- Amazon has EKS - Amazon Elastic Kubernetes Service,
- Microsoft has AKS – Azure Kubernetes Service,
- IBM has IBM Cloud Kubernetes Service,

3

部署您的第一个应用程序

本章涵盖

- 在笔记本电脑上运行单节点 Kubernetes 集群
- 在 Google Kubernetes Engine 上设置 Kubernetes 集群
- 设置和使用 kubectl 命令行工具
- 在 Kubernetes 中部署应用程序并使其在全球范围内可用
- 水平扩展应用程序

本章的目标是向您展示如何运行本地单节点开发 Kubernetes 集群或在云中设置适当的、托管的多节点集群。集群运行后，您将使用它来运行在上一章中创建的容器。

3.1 部署Kubernetes集群

设置成熟的多节点 Kubernetes 集群并不是一项简单的任务，特别是如果您不熟悉 Linux 和网络管理的话。正确的 Kubernetes 安装跨越多个物理机或虚拟机，并且需要正确的网络设置以允许集群中的所有容器相互通信。您可以将 Kubernetes 安装在您组织的笔记本电脑上

基础设施或由云提供商（Google Compute Engine、Amazon EC2、Microsoft Azure 等）提供的虚拟机上。另外，大多数云提供商现在都提供托管 Kubernetes 服务，让您免去安装和管理的麻烦。以下是最大的云提供商提供的服务的简短概述：

- Google 提供 GKE - Google Kubernetes Engine，
- Amazon 有 EKS - Amazon Elastic Kubernetes 服务，
- Microsoft 有 AKS – Azure Kubernetes 服务，

- Alibaba provides the Alibaba Cloud Container Service.

Installing and managing Kubernetes is much more difficult than just using it, especially until you're intimately familiar with its architecture and operation. For this reason, we'll start with the easiest ways to obtain a working Kubernetes cluster. You'll learn several ways to run a single-node Kubernetes cluster on your local computer and how to use a hosted cluster running on Google Kubernetes Engine (GKE).

A third option, which involves installing a cluster using the `kubeadm` tool, is explained in Appendix B. The tutorial there will show you how to set up a three-node Kubernetes cluster using virtual machines. But you may want to try that only after you've become familiar with using Kubernetes. Many other options also exist, but they are beyond the scope of this book. Refer to the kubernetes.io website to learn more.

If you've been granted access to an existing cluster deployed by someone else, you can skip this section and go on to section 3.2 where you'll learn how to interact with Kubernetes clusters.

3.1.1 Using the built-in Kubernetes cluster in Docker Desktop

If you use macOS or Windows, you've most likely installed Docker Desktop to run the exercises in the previous chapter. It contains a single-node Kubernetes cluster that you can enable via its Settings dialog box. This may be the easiest way for you to start your Kubernetes journey, but keep in mind that the version of Kubernetes may not be as recent as when using the alternative options described in the next sections.

> **NOTE** Although technically not a cluster, the single-node Kubernetes system provided by Docker Desktop should be enough to explore most of the topics discussed in this book. When an exercise requires a multi-node cluster, I will point this out.

Enabling Kubernetes in Docker Desktop

Assuming Docker Desktop is already installed on your computer, you can start the Kubernetes cluster by clicking the whale icon in the system tray and opening the Settings dialog box. Click the *Kubernetes* tab and make sure the *Enable Kubernetes* checkbox is selected. The components that make up the Control Plane run as Docker containers, but they aren't displayed in the list of running containers when you invoke the `docker ps` command. To display them, select the *Show system containers* checkbox.

> **NOTE** The initial installation of the cluster takes several minutes, as all container images for the Kubernetes components must be downloaded.

- 阿里巴巴提供阿里云容器服务。

安装和管理 Kubernetes 比仅仅使用它要困难得多，尤其是在您非常熟悉它的架构和操作之前。因此，我们将从最简单的方法开始获取可用的 Kubernetes 集群。您将了解在本地计算机上运行单节点 Kubernetes 集群的多种方法，以及如何使用在 Google Kubernetes Engine (GKE) 上运行的托管集群。

第三个选项涉及使用 kubeadm 工具安装集群，附录 B 中对此进行了说明。其中的教程将向您展示如何使用虚拟机设置三节点 Kubernetes 集群。但您可能只想在熟悉使用 Kubernetes 后才尝试这样做。还存在许多其他选项，但它们超出了本书的范围。请参阅 kubernetes.io 网站了解更多信息。

如果您已被授予访问其他人部署的现有集群的权限，则可以跳过本节并继续进行第 3.2 节，您将在其中学习如何与 Kubernetes 集群交互。

3.1.1 使用Docker Desktop内置的Kubernetes集群

如果您使用 macOS 或 Windows，您很可能已经安装了 Docker Desktop 来运行上一章中的练习。它包含一个单节点 Kubernetes 集群，您可以通过其“设置”对话框启用该集群。这可能是您开始 Kubernetes 之旅的最简单方法，但请记住，Kubernetes 的版本可能不像使用下一节中描述的替代选项时那么新。

> 注意 虽然技术上不是集群，但 Docker Desktop 提供的单节点 Kubernetes 系统应该足以探索本书中讨论的大部分主题。当一项练习需要多项
> 节点集群，我会指出这一点。

在 Docker Desktop 中启用 Kubernetes

假设您的计算机上已安装 Docker Desktop，您可以通过单击系统托盘中的鲸鱼图标并打开“设置”对话框来启动 Kubernetes 集群。单击 Kubernetes 选项卡并确保选中启用 Kubernetes 复选框。组成控制平面的组件作为 Docker 容器运行，但当您调用 docker ps 命令时，它们不会显示在正在运行的容器列表中。要显示它们，请选中显示系统容器复选框。

> 注意 集群的初始安装需要几
> 分钟时间，因为必须下载
> Kubernetes 组件的所有容器
> 映像。

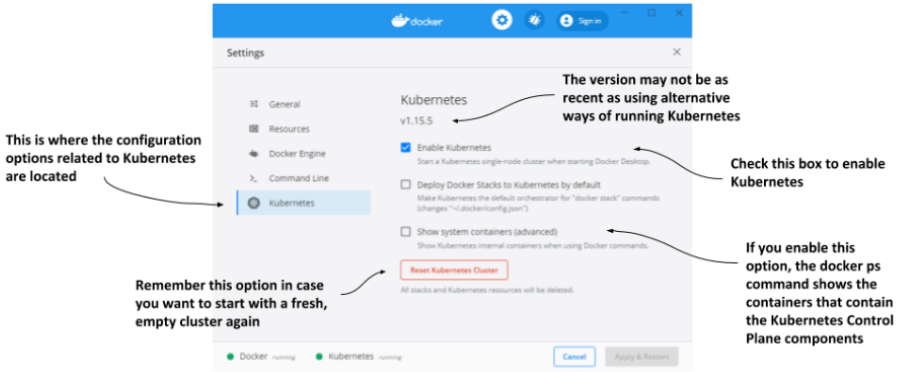


Figure 3.1 The Settings dialog box in Docker Desktop for Windows

Remember the *Reset Kubernetes Cluster* button if you ever want to reset the cluster to remove all the objects you've deployed in it.

VISUALIZING THE SYSTEM

To understand where the various components that make up the Kubernetes cluster run in Docker Desktop, look at the following figure.

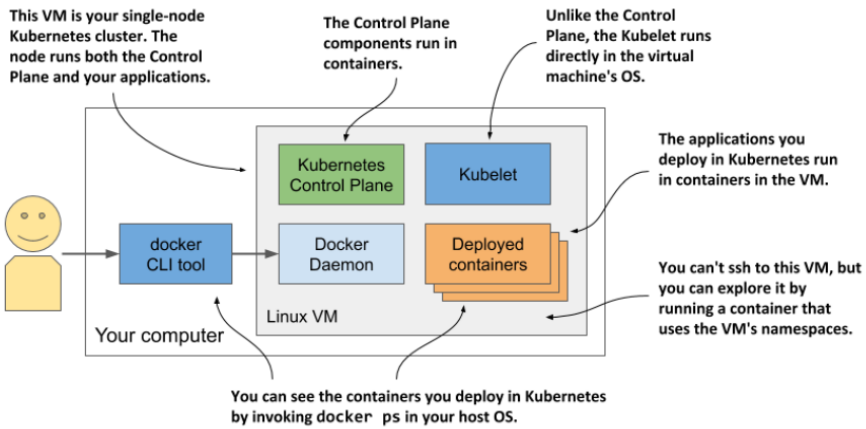


Figure 3.2 Kubernetes running in Docker Desktop

图 3.1 Windows 版 Docker Desktop 中的“设置”对话框

如果您想要重置集群以删除已部署在其中的所有对象，请记住重置 Kubernetes 集群按钮。

系统可视化

要了解组成 Kubernetes 集群的各个组件在 Docker Desktop 中运行的位置，请查看下图。

图3.2 Docker Desktop中运行的Kubernetes

Docker Desktop sets up a Linux virtual machine that hosts the Docker Daemon and all the containers. This VM also runs the Kubelet - the Kubernetes agent that manages the node. The components of the Control Plane run in containers, as do all the applications you deploy.

To list the running containers, you don't need to log on to the VM because the docker CLI tool available in your host OS displays them.

EXPLORING THE VIRTUAL MACHINE FROM THE INSIDE

At the time of writing, Docker Desktop provides no command to log into the VM if you want to explore it from the inside. However, you can run a special container configured to use the VM's namespaces to run a remote shell, which is virtually identical to using SSH to access a remote server. To run the container, execute the following command:

```
$ docker run --net=host --ipc=host --uts=host --pid=host --privileged \
  --security-opt=seccomp=unconfined -it --rm -v /:/host alpine chroot /host
```

This long command requires explanation:

- The container is created from the `alpine` image.
- The `--net`, `--ipc`, `--uts` and `--pid` flags make the container use the host's namespaces instead of being sandboxed, and the `--privileged` and `--security-opt` flags give the container unrestricted access to all sys-calls.
- The `-it` flag runs the container interactive mode and the `--rm` flags ensures the container is deleted when it terminates.
- The `-v` flag mounts the host's root directory to the `/host` directory in the container. The `chroot /host` command then makes this directory the root directory in the container.

After you run the command, you are in a shell that's effectively the same as if you had used SSH to enter the VM. Use this shell to explore the VM - try listing processes by executing the `ps aux` command or explore the network interfaces by running `ip addr`.

3.1.2 Running a local cluster using Minikube

Another way to create a Kubernetes cluster is to use *Minikube*, a tool maintained by the Kubernetes community. The version of Kubernetes that Minikube deploys is usually more recent than the version deployed by Docker Desktop. The cluster consists of a single node and is suitable for both testing Kubernetes and developing applications locally. It normally runs Kubernetes in a Linux VM, but if your computer is Linux-based, it can also deploy Kubernetes directly in your host OS via Docker.

> **NOTE** If you configure Minikube to use a VM, you don't need Docker, but you do need a hypervisor like VirtualBox. In the other case you need Docker, but not the hypervisor.

INSTALLING MINIKUBE

Minikube supports macOS, Linux, and Windows. It has a single binary executable file, which you'll find in the Minikube repository on GitHub (http://github.com/kubernetes/minikube).

Docker Desktop 设置一个 Linux 虚拟机来托管 Docker Daemon 和所有容器。该虚拟机还运行 Kubelet——管理节点的 Kubernetes 代理。控制平面的组件在容器中运行，您部署的所有应用程序也是如此。

要列出正在运行的容器，您不需要登录到虚拟机，因为 docker CLI 主机操作系统中可用的工具会显示它们。

从内部探索虚拟机

在撰写本文时，如果您想从内部探索它，Docker Desktop 没有提供登录虚拟机的命令。但是，您可以运行一个特殊的容器，配置为使用虚拟机的命名空间来运行远程 shell，这实际上与使用 SSH 访问远程服务器相同。要运行容器，请执行以下命令：

```
$ docker run --net=主机 --ipc=主机 --uts=主机 --pid=主机 --privileged \
  --security-opt=seccomp=unconfined -it --rm -v /:/host alpine chroot /host
```

这个长命令需要解释一下：

- 容器是根据alpine 映像创建的。
- --net、--ipc、--uts 和 --pid 标志使容器使用主机的命名空间而不是沙箱，而 --privileged 和 --security-opt 标志使容器可以不受限制地访问所有系统-打电话。
- -it 标志运行容器交互模式，--rm 标志确保容器在终止时被删除。
- -v 标志将主机的根目录安装到容器中的/host 目录。然后 chroot /host 命令使该目录成为根目录

容器。

运行该命令后，您将进入一个 shell，该 shell 实际上与使用 SSH 进入虚拟机相同。使用此 shell 探索虚拟机 - 尝试通过执行 ps aux 命令列出进程或通过运行 ip addr 探索网络接口。

3.1.2 使用Minikube运行本地集群

创建 Kubernetes 集群的另一种方法是使用 Minikube，这是 Kubernetes 社区维护的工具。Minikube 部署的 Kubernetes 版本通常比 Docker Desktop 部署的版本更新。该集群由单个节点组成，适合测试 Kubernetes 和本地开发应用程序。它通常在 Linux VM 中运行 Kubernetes，但如果您的计算机基于 Linux，它也可以通过 Docker 直接在主机操作系统中部署 Kubernetes。

> 注意如果您将 Minikube 配置为使用 VM，则不需要 Docker，但确实需要 VirtualBox 等虚拟机管理程序。在另一种情况下，您需要 Docker，但不需要虚拟机管理程序。

安装 MINIKUBE

Minikube 支持 macOS、Linux 和 Windows。它有一个二进制可执行文件，您可以在 GitHub (http://github.com/kubernetes/minikube) 上的 Minikube 存储库中找到该文件。

It's best to follow the current installation instructions published there, but roughly speaking, you install it as follows.

On macOS you can install it using the Brew Package Manager, on Windows there's an installer that you can download, and on Linux you can either download a .deb or .rpm package or simply download the binary file and make it executable with the following command:

```
$ curl -LO https://storage.googleapis.com/minikube/releases/latest/minikube-linux-
amd64 && sudo install minikube-linux-amd64 /usr/local/bin/minikube
```

For details on your specific OS, please refer to the installation guide online.

STARTING A KUBERNETES CLUSTER WITH MINIKUBE

After Minikube is installed, start the Kubernetes cluster as indicated in the following listing.

Listing 3.1 Starting Kubernetes with Minikube

```
$ minikube start
minikube v1.11.0 on Fedora 31
Using the virtualbox driver based on user configuration
Downloading VM boot image ...
> minikube-v1.11.0.iso.sha256: 65 B / 65 B [-------------] 100.00% ? p/s 0s
> minikube-v1.11.0.iso: 174.99 MiB / 174.99 MiB [] 100.00% 50.16 MiB p/s 4s
Starting control plane node minikube in cluster minikube
Downloading Kubernetes v1.18.3 preload ...
> preloaded-images-k8s-v3-v1.18.3-docker-overlay2-amd64.tar.lz4: 526.01 MiB
Creating virtualbox VM (CPUs=2, Memory=6000MB, Disk=20000MB) ...
Preparing Kubernetes v1.18.3 on Docker 19.03.8 ...
Verifying Kubernetes components...
Enabled addons: default-storageclass, storage-provisioner
Done! kubectl is now configured to use "minikube"
```

The process may take several minutes, because the VM image and the container images of the Kubernetes components must be downloaded.

TIP If you use Linux, you can reduce the resources required by Minikube by creating the cluster without a VM. Use this command: `minikube start --vm-driver none`

CHECKING MINIKUBE'S STATUS

When the `minikube start` command is complete, you can check the status of the cluster by running `minikube status`, as shown in the following listing.

Listing 3.2 Checking Minikube's status

```
$ minikube status
host: Running
kubelet: Running
apiserver: Running
kubeconfig: Configured
```


58

最好遵循那里发布的当前安装说明，但粗略地说，您可以按如下方式安装它。

在 macOS 上，您可以使用 Brew Package Manager 安装它，在 Windows 上有一个
您可以下载安装程序，在 Linux 上您可以下载 .deb 或 .rpm
打包或直接下载二进制文件并使用以下命令使其可执行： amd64 && sudo install minikube-
linux-amd64 /usr/local/bin/minikube

有关您的特定操作系统的详细信息，请参阅在线安装指南。

使用 MINIKUBE 启动 Kubernetes 集群

安装 Minikube 后，启动 Kubernetes 集群，如下清单所示。

清单 3.1 使用 Minikube 启动 Kubernetes

```
$ minikube 启动
Fedora 31 上的 minikube v1.11.0
根据用户配置使用 virtualbox 驱动程序正在下载 VM 启动映像...
> minikube-v1.11.0.iso.sha256: 65 B / 65 B [-------------]
100.00% ？秒/秒 0秒
> minikube-v1.11.0.iso: 174.99 MiB / 174.99 MiB [] 100.00% 50.16
MiB p/s 4s 启动集群 minikube 中的控制平面节点 minikube
正在下载 Kubernetes v1.18.3 预加载...
> preloaded-images-k8s-v3-v1.18.3-docker-overlay2-amd64.tar.lz4:
526.01 MiB 创建 virtualbox VM (CPU=2, 内存=6000MB, 磁盘=20000MB)
```

该过程可能需要几分钟，因为必须下载 VM 映像和 Kubernetes 组件的容器映像。

提示如果您使用 Linux，则可以通过创建没有 VM 的集群来减少 Minikube 所需的资源。
使用此命令：minikube start --vm-driver none

检查 MINIKUBE 的状态

当minikube启动命令完成后，您可以通过以下方式检查集群的状态

运行 minikube 状态，如下清单所示。

清单 3.2 检查 Minikube 的状态

```
$ minikube
status 主机：运
行 kubelet：运行
apiserver：运行
kubeconfig：已配
```

The output of the command shows that the Kubernetes host (the VM that hosts Kubernetes) is running, and so are the Kubelet – the agent responsible for managing the node – and the Kubernetes API server. The last line shows that the kubectl command-line tool (CLI) is configured to use the Kubernetes cluster that Minikube has provided. Minikube doesn't install the CLI tool, but it does create its configuration file. Installation of the CLI tool is explained in section 3.2.

VISUALIZING THE SYSTEM

The architecture of the system, which is shown in the next figure, is practically identical to the one in Docker Desktop.

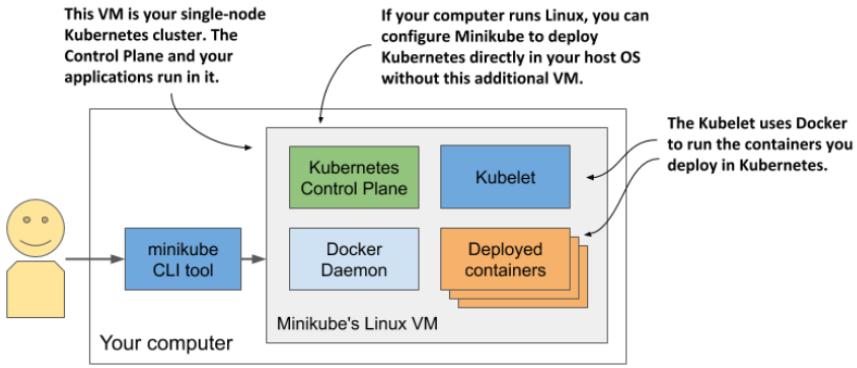


Figure 3.3 Running a single-node Kubernetes cluster using Minikube

The Control Plane components run in containers in the VM or directly in your host OS if you used the `--vm-driver none` option to create the cluster. The Kubelet runs directly in the VM's or your host's operating system. It runs the applications you deploy in the cluster via the Docker Daemon.

You can run `minikube ssh` to log into the Minikube VM and explore it from inside. For example, you can see what's running in the VM by running `ps aux` to list running processes or `docker ps` to list running containers.

> **TIP** If you want to list containers using your local docker CLI instance, as in the case of Docker Desktop, run the following command: `eval $(minikube docker-env)`

3.1.3 Running a local cluster using kind (Kubernetes in Docker)

An alternative to Minikube, although not as mature, is *kind* (Kubernetes-in-Docker). Instead of running Kubernetes in a virtual machine or directly on the host, kind runs each Kubernetes

该命令的输出显示 Kubernetes 主机（托管 Kubernetes 的虚拟机）正在运行，Kubelet（负责管理节点的代理）和 Kubernetes API 服务器也在运行。最后一行显示 kubectl 命令行工具 (CLI) 已配置为使用 Minikube 提供的 Kubernetes 集群。 Minikube 不安装 CLI 工具，但它会创建其配置文件。 CLI 工具的安装在 3.2 节中进行了说明。

系统可视化

该系统的架构如下图所示，实际上与 Docker Desktop 中的架构相同。

图3.3 使用Minikube运行单节点Kubernetes集群

控制平面组件在虚拟机中的容器中运行，或者如果您使用 --vm-driver none 选项创建集群，则直接在主机操作系统中运行。 Kubelet 直接在虚拟机或主机的操作系统中运行。它通过 Docker Daemon 运行您在集群中部署的应用程序。

您可以运行 minikube ssh 登录 Minikube VM 并从内部探索它。例如，您可以通过运行 ps aux 列出正在运行的进程来查看虚拟机中正在运行的内容

或 docker ps 列出正在运行的容器。

> 提示如果您想使用本地 docker CLI 实例列出容器
> （如 Docker Desktop 的情况），请运行以下命令：
> eval $(minikube docker-env)

3.1.3 使用kind运行本地集群（Docker中的Kubernetes）

Minikube 的替代方案虽然不那么成熟，但很不错（Kubernetes-in-Docker）。而不是在虚拟机中或直接在主机上运行 Kubernetes

cluster node inside a container. Unlike Minikube, this allows it to create multi-node clusters by starting several containers. The actual application containers that you deploy to Kubernetes then run within these node containers. The system is shown in the next figure.

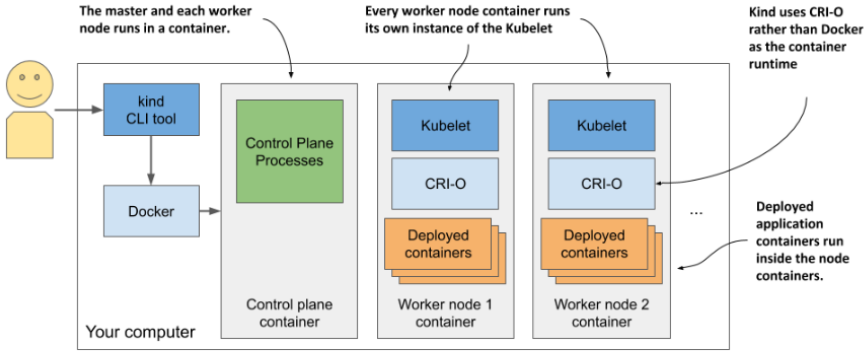


Figure 3.4 Running a multi-node Kubernetes cluster using kind

In the previous chapter I mentioned that a process that runs in a container actually runs in the host OS. This means that when you run Kubernetes using kind, all Kubernetes components run in your host OS. The applications you deploy to the Kubernetes cluster also run in your host OS.

This makes kind the perfect tool for development and testing, as everything runs locally and you can debug running processes as easily as when you run them outside of a container. I prefer to use this approach when I develop apps on Kubernetes, as it allows me to do magical things like run network traffic analysis tools such as Wireshark or even my web browser inside the containers that run my applications. I use a tool called `nsenter` that allows me to run these tools in the network or other namespaces of the container.

If you're new to Kubernetes, the safest bet is to start with Minikube, but if you're feeling adventurous, here's how to get started with kind.

INSTALLING KIND

Just like Minikube, kind consists of a single binary executable file. To install it, refer to the installation instructions at https://kind.sigs.k8s.io/docs/user/quick-start/. On macOS and Linux, the command to install it is as follows:

```
$ curl -Lo ./kind https://github.com/kubernetes-
sigs/kind/releases/download/v0.7.0/kind-$(uname)-amd64 && \chmod +x ./kind && \mv
./kind /some-dir-in-your-PATH/kind
```


容器内的集群节点。与 Minikube 不同的是，这允许它通过启动多个容器来创建多节点集群。然后，您部署到 Kubernetes 的实际应用程序容器在这些节点容器中运行。该系统如下图所示。

图3.4 使用kind运行多节点Kubernetes集群

在上一章中我提到，在容器中运行的进程实际上是在主机操作系统中运行。这意味着当您使用 kind 运行 Kubernetes 时，所有 Kubernetes 组件都在您的主机操作系统中运行。您部署到 Kubernetes 集群的应用程序也在您的主机操作系统中运行。

这使得 kind 成为开发和测试的完美工具，因为一切都在本地运行，并且您可以像在容器外部运行它们一样轻松地调试正在运行的进程。当我在 Kubernetes 上开发应用程序时，我更喜欢使用这种方法，因为它允许我做一些神奇的事情，比如运行 Wireshark 等网络流量分析工具，甚至在运行应用程序的容器内运行我的 Web 浏览器。我使用一个名为 nsenter 的工具，它允许我在网络或容器的其他命名空间中运行这些工具。

如果您是 Kubernetes 新手，最安全的选择是从 Minikube 开始，但如果您感觉喜欢冒险，以下是如何开始使用 kind。

安装方式

就像 Minikube 一样，kind 由单个二进制可执行文件组成。要安装它，请参阅 https://kind.sigs.k8s.io/docs/user/quick-start/ 上的安装说明。在 macOS 和 Linux 上，安装命令如下： sigs/kind/releases/download/v0.7.0/kind-$(uname)-amd64 && \chmod +x ./kind && \mv

Check the documentation to see what the latest version is and use it instead of v0.7.0 in the above example. Also, replace `/some-dir-in-your-PATH/` with an actual directory in your path.

NOTE Docker must be installed on your system to use kind.

STARTING A KUBERNETES CLUSTER WITH KIND

Starting a new cluster is as easy as it is with Minikube. Execute the following command:

```
$ kind create cluster
```

Like Minikube, kind configures kubectl to use the cluster that it creates.

STARTING A MULTI-NODE CLUSTER WITH KIND

Kind runs a single-node cluster by default. If you want to run a cluster with multiple worker nodes, you must first create a configuration file named `kind-multi-node.yaml` with the following content (you can find the file in the book's code archive, directory `Chapter03/`):

Listing 3.3 Config file for running a three-node cluster with kind

```
kind: Cluster
apiVersion: kind.sigs.k8s.io/v1alpha3
nodes:
- role: control-plane
- role: worker
- role: worker
```

With the file in place, create the cluster using the following command:

```
$ kind create cluster --config kind-multi-node.yaml
```

LISTING WORKER NODES

At the time of this writing, kind doesn't provide a command to check the status of the cluster, but you can list cluster nodes using `kind get nodes`, as shown in the next listing.

Listing 3.4 Listing nodes using kind get nodes

```
$ kind get nodes
kind-worker2
kind-worker
kind-control-plane
```

NOTE Due to width restrictions, the node names `control-plane`, `worker1`, and `worker2` are used instead of the actual node names throughout this book.

Since each node runs as a container, you can also see the nodes by listing the running containers using `docker ps`, as the next listing shows.

检查文档以了解最新版本是什么，并在上面的示例中使用它而不是 v0.7.0。另外，将 /some-dir-in-your-PATH/ 替换为路径中的实际目录。

注意 Docker 必须安装在您的系统上才能使用 kind。

使用 Kind 启动 Kubernetes 集群

启动新集群就像使用 Minikube 一样简单。执行以下命令：

```
$ kind 创建集群
```

与 Minikube 一样，kind 配置 kubectl 以使用它创建的集群。

使用 Kind 启动多节点集群

Kind 默认运行单节点集群。如果要运行具有多个工作节点的集群，则必须首先创建一个名为 kind-multi-node.yaml 的配置文件，其中包含

以下内容（您可以在本书的代码存档中找到该文件，目录Chapter03/）：

清单 3.3 用于运行带有 kind 的三节点集群的配置文件

```
种类：簇
api 版本：
kind.sigs.k8s.io/v1alpha3 节点：
- 角色：控制平面 - 角色：工作者
- 角色：工作者
```

文件就位后，使用以下命令创建集群：

```
$ kind 创建集群 --config kind-multi-node.yaml
```

列出工作节点

在撰写本文时，kind 尚未提供检查集群状态的命令，但您可以使用 kind get Nodes 列出集群节点，如下列表所示。

清单 3.4 使用 kind get 节点列出节点

```
$ kind 获取节点
kind-worker2
kind-worker
```

注意 由于宽度限制，使用节点名称 control-plane、worker1 和worker2 而不是本书中实际的节点名称。

由于每个节点都作为容器运行，因此您还可以通过使用 docker ps 列出正在运行的容器来查看节点，如下列表所示。

Listing 3.5 Displaying kind nodes running as containers

```
$ docker ps
CONTAINER ID    IMAGE                  ...    NAMES
45d0f712eac0    kindest/node:v1.18.2   ...    kind-worker2
d1e88e98e3ae    kindest/node:v1.18.2   ...    kind-worker
4b7751144ca4    kindest/node:v1.18.2   ...    kind-control-plane
```

LOGGING INTO CLUSTER NODES PROVISIONED BY KIND

Unlike Minikube, where you use `minikube ssh` to log into the node if you want to explore the processes running inside it, with kind you use `docker exec`. For example, to enter the node called `kind-control-plane`, run:

```
$ docker exec -it kind-control-plane bash
```

Instead of using Docker to run containers, nodes created by kind use the CRI-O container runtime, which I mentioned in the previous chapter as a lightweight alternative to Docker. The `crictl` CLI tool is used to interact with CRI-O. Its use is very similar to that of the `docker` tool. After logging into the node, list the containers running in it by running `crictl ps` instead of `docker ps`. An example of the command's output is shown in the next listing:

Listing 3.6 Listing containers inside a cluster node provisioned with kind

```
root@kind-control-plane:/# crictl ps
CONTAINER ID    IMAGE           CREATED       STATE     NAME
c7f44d171fb72   eb516548c180f   15 min ago    Running   coredns         ...
cce9c0261854c   eb516548c180f   15 min ago    Running   coredns         ...
e6522aae66fcc   d428039608992   16 min ago    Running   kube-proxy      ...
6b2dc4bbfee0c   ef97cccdfdb50   16 min ago    Running   kindnet-cni     ...
c3e66dfe44deb   be321f2ded3f3   16 min ago    Running   kube-apiserver  ...
```

3.1.4 Creating a managed cluster with Google Kubernetes Engine

If you want to use a full-fledged multi-node Kubernetes cluster instead of a local one, you can use a managed cluster, such as the one provided by Google Kubernetes Engine (GKE). This way, you don't have to manually set up all the cluster nodes and networking, which is usually too hard for someone taking their first steps with Kubernetes. Using a managed solution such as GKE ensures that you don't end up with an incorrectly configured cluster.

SETTING UP GOOGLE CLOUD AND INSTALLING THE GCLOUD CLIENT BINARY

Before you can set up a new Kubernetes cluster, you must set up your GKE environment. The process may change in the future, so I'll only give you a few general instructions here. For complete instructions, refer to https://cloud.google.com/container-engine/docs/before-you-begin.

Roughly, the whole procedure includes

1. Signing up for a Google account if you don't have one already.
2. Creating a project in the Google Cloud Platform Console.

清单 3.5 显示作为容器运行的种类节点

```
$ docker
ps 容器 ID      图像                    ...    名称
45d0f712eac0    最善良/节点：v1.18.2 ...    善良的工人2
d1e88e98e3ae    最善良/节点：v1.18.2 ...    善良的工人
4b7751144ca4    最善良/节点：v1.18.2 ...    种类控制平面
```

登录按种类配置的集群节点

与 Minikube 不同的是，如果您想探索节点内部运行的进程，则可以使用 minikube ssh 登录节点，而使用 docker exec 则可以。例如，输入节点

称为 kind-control-plane，运行：

```
$ docker exec -it kind-control-plane bash
```

类创建的节点不使用 Docker 来运行容器，而是使用 CRI-O 容器运行时，我在上一章中提到它是 Docker 的轻量级替代方案。 crictl CLI 工具用于与 CRI-O 交互。它的使用与docker工具非常相似。登录节点后，通过运行 crictl ps 而不是 docker ps 列出在该节点中运行的容器。该命令的输出示例如下所示：

清单 3.6 列出配置有 kind 的集群节点内的容器

```
root@kind-control-plane:/# crictl ps 容器 ID 图像创
建状态名称
c7f44d171fb72  eb516548c180f  15 分钟前 运行 coredns          ...
cce9c0261854c  eb516548c180f  15 分钟前 运行 coredns          ...
e6522aae66fcc  d428039608992  16 分钟前 运行 kube-proxy       ...
6b2dc4bbfee0c  ef97cccdfdb50  16 分钟前 运行 kindnet-cni      ...
c3e66dfe44deb  be321f2ded3f3  16 分钟前 运行 kube-apiserver ...
```

3.1.4 使用 Google Kubernetes Engine 创建托管集群

如果您想使用成熟的多节点 Kubernetes 集群而不是本地集群，可以使用托管集群，例如 Google Kubernetes Engine (GKE) 提供的集群。这样，您就不必手动设置所有集群节点和网络，这对于刚开始使用 Kubernetes 的人来说通常太难了。使用 GKE 等托管解决方案可确保您不会得到错误配置的集群。

设置 GOOGLE Cloud 并安装 GCloud 客户端二进制文件

在设置新的 Kubernetes 集群之前，您必须设置 GKE 环境。该过程将来可能会发生变化，所以我在这里只给您一些一般性说明。从你开始。

整个流程大致包括

1. 如果您还没有 Google 帐户，请注册一个。
2. 在 Google Cloud Platform Console 中创建项目。

3. Enabling billing. This does require your credit card info, but Google provides a 12-month free trial with a free $300 credit. And they don't start charging automatically after the free trial is over.
4. Downloading and installing the Google Cloud SDK, which includes the `gcloud` tool.
5. Creating the cluster using the `gcloud` command-line tool.

NOTE Certain operations (the one in step 2, for example) may take a few minutes to complete, so relax and grab a coffee in the meantime.

CREATING A GKE KUBERNETES CLUSTER WITH THREE NODES

Before you create your cluster, you must decide in which geographical region and zone it should be created. Refer to https://cloud.google.com/compute/docs/regions-zones to see the list of available locations. In the following examples, I use the europe-west3 region based in Frankfurt, Germany. It has three different zones - I'll use the zone europe-west3-c. The default zone for all gcloud operations can be set with the following command:

```
$ gcloud config set compute/zone europe-west3-c
```

Create the Kubernetes cluster using the command shown in the next listing. You can choose a name other than `kubia` if you wish.

Listing 3.7 Creating a three-node cluster in GKE

```
$ gcloud container clusters create kubia --num-nodes 3
Creating cluster kubia in europe-west3-c...
...
kubeconfig entry generated for kubia.
NAME    LOCAT.    MASTER_VER  MASTER_IP    MACH_TYPE      ... NODES
STATUS
kubia   eu-w3-c   1.13.11...  5.24.21.22   n1-standard-1  ... 3
RUNNING
```

NOTE I'm creating all three worker nodes in the same zone, but you can also spread them across all zones in the region by setting the `compute/zone` config value to an entire region instead of a single zone. If you do so, note that `--num-nodes` indicates the number of nodes *per zone*. If the region contains three zones and you only want three nodes, you must set `--num-nodes` to 1.

You should now have a running Kubernetes cluster with three worker nodes. Each node is a virtual machine provided by the Google Compute Engine (GCE) infrastructure-as-a-service platform. You can list GCE virtual machines using the command in the next listing.

Listing 3.8 Listing GCE virtual machines

```
$ gcloud compute instances list
NAME      ZONE         MACHINE_TYPE   INTERNAL_IP  EXTERNAL_IP     STATUS
...-ctlk  eu-west3-c   n1-standard-1  10.156.0.16  34.89.238.55    RUNNING
...-gj1f  eu-west3-c   n1-standard-1  10.156.0.14  35.242.223.97   RUNNING
```


63

3. 启用计费。这确实需要您的信用卡信息，但 Google 提供了 12-一个月免费试用，并获得 300 美元的免费赠金。免费试用结束后，它们不会自动开始充电。
4. 下载并安装 Google Cloud SDK，其中包括 gcloud 工具。
5. 使用 gcloud 命令行工具创建集群。

注意 某些操作（例如步骤 2 中的操作）可能需要几分钟才能完成，因此请放心同时喝杯咖啡。

创建具有三个节点的 GKE Kubernetes 集群

在创建集群之前，您必须决定应在哪个地理区域和区域中创建集群。请参阅 https://cloud.google.com/compute/docs/regions-zones 查看可用位置的列表。在以下示例中，我使用位于德国法兰克福的 europe-west3 区域。它有三个不同的区域 - 我将使用区域 europe-west3-c。可以使用以下命令设置所有 gcloud 操作的默认区域：

```
$ gcloud 配置集计算/区域 europe-west3-c
```

使用下一个清单中所示的命令创建 Kubernetes 集群。如果您愿意，您可以选择 kubia 以外的名称。

清单 3.7 在 GKE 中创建一个三节点集群

```
$ gcloud 容器集群 create kubia --num-nodes 3 在
europe-west3-c 创建集群 kubia...
...为 kubia 生成的 kubeconfig 条目。名称 定位。 MASTER_VER MASTER_IP
MACH_TYPE ... 节点

状态 kubia eu-w3-c 1.13.11... 5.24.21.22 n1-standard-1 ... 3

跑步
```

注意我在同一区域中创建所有三个工作节点，但您也可以通过将计算/区域配置值设置为整个区域而不是单个区域，将它们分布在该区域的所有区域中。如果你
这样做时，请注意 --num-nodes 表示每个区域的节点数。如果该区域包含三个区域
如果您只需要三个节点，则必须将 --num-nodes 设置为 1

您现在应该拥有一个正在运行的 Kubernetes 集群，其中包含三个工作节点。每个节点都是由 Google 计算引擎 (GCE) 基础设施即服务平台提供的虚拟机。您可以使用下一个列表中的命令列出 GCE 虚拟机。

清单 3.8 列出 GCE 虚拟机

```
$ gcloud 计算实例列表 NAME ZONE MACHINE_TYPE INTERNAL_IP
EXTERNAL_IP STATUS

...-ctlk eu-west3-c n1-standard-1 10.156.0.16 34.89.238.55 正在
```

```
...-r01z  eu-west3-c  n1-standard-1  10.156.0.15  35.198.191.189  RUNNING
```

TIP Each VM incurs costs. To reduce the cost of your cluster, you can reduce the number of nodes to one, or even to zero while not using it. See next section for details.

The system is shown in the next figure. Note that only your worker nodes run in GCE virtual machines. The control plane runs elsewhere - you can't access the machines hosting it.

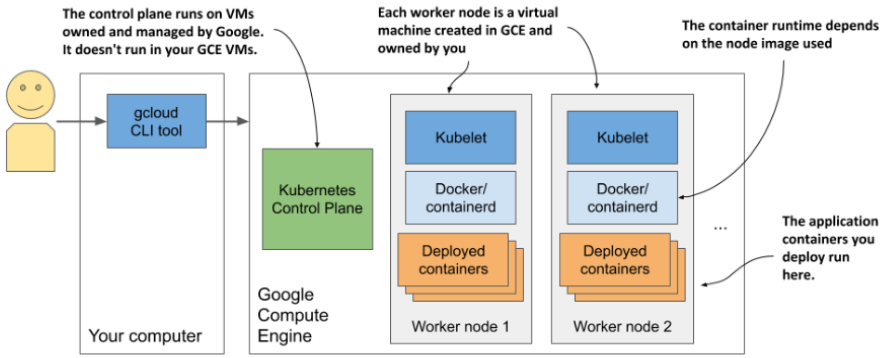


Figure 3.5 Your Kubernetes cluster in Google Kubernetes Engine

SCALING THE NUMBER OF NODES

Google allows you to easily increase or decrease the number of nodes in your cluster. For most exercises in this book you can scale it down to just one node if you want to save money. You can even scale it down to zero so that your cluster doesn't incur any costs.

To scale the cluster to zero, use the following command:

```
$ gcloud container clusters resize kubia --size 0
```

The nice thing about scaling to zero is that none of the objects you create in your Kubernetes cluster, including the applications you deploy, are deleted. Granted, if you scale down to zero, the applications will have no nodes to run on, so they won't run. But as soon as you scale the cluster back up, they will be redeployed. And even with no worker nodes you can still interact with the Kubernetes API (you can create, update, and delete objects).

INSPECTING A GKE WORKER NODE

If you're interested in what's running on your nodes, you can log into them with the following command (use one of the node names from the output of the previous command):

```
$ gcloud compute ssh gke-kubia-default-pool-9bba9b18-4glf
```



```
...-r01z 欧盟-west3-rd-标准-1     10.156.0.15 35.198.191.189 跑步
```

提示每个虚拟机都会产生成本。要降低集群成本，您可以将节点数量减少到 1 个，或者甚至在不使用时为零。有关详细信息，请参阅下一节。

该系统如下图所示。请注意，只有您的工作节点在 GCE 虚拟机中运行。控制平面在其他地方运行 - 您无法访问托管它的计算机。

图 3.5 Google Kubernetes Engine 中的 Kubernetes 集群

扩展节点数量

Google 允许您轻松增加或减少集群中的节点数量。对于本书中的大多数练习，如果您想省钱，可以将其缩小到只有一个节点。您甚至可以将其缩小到零，这样您的集群就不会产生任何成本。要将集群扩展到零，请使用以下命令：

```
$ gcloud 容器集群调整 kubia 大小 --size 0
```

扩展到零的好处是，您在 Kubernetes 集群中创建的任何对象（包括您部署的应用程序）都不会被删除。当然，如果缩小到零，应用程序将没有可运行的节点，因此它们将无法运行。但是，一旦您重新扩展集群，它们就会被重新部署。即使没有工作节点，您仍然可以与 Kubernetes API 交互（您可以创建、更新和删除对象）。

检查 GKE 工作节点

如果您对节点上运行的内容感兴趣，可以使用以下命令登录它们（使用上一个命令输出中的节点名称之一）：

While logged into the node, you can try to list all running containers with `docker ps`. You haven't run any applications yet, so you'll only see Kubernetes system containers. What they are isn't important right now, but you'll learn about them in later chapters.

3.1.5 Creating a cluster using Amazon Elastic Kubernetes Service

If you prefer to use Amazon instead of Google to deploy your Kubernetes cluster in the cloud, you can try the Amazon Elastic Kubernetes Service (EKS). Let's go over the basics.

First, you have to install the `eksctl` command-line tool by following the instructions at https://docs.aws.amazon.com/eks/latest/userguide/getting-started-eksctl.html.

CREATING AN EKS KUBERNETES CLUSTER

Creating an EKS Kubernetes cluster using `eksctl` does not differ significantly from how you create a cluster in GKE. All you must do is run the following command:

```
$ eksctl create cluster --name kubia --region eu-central-1 --nodes 3 --ssh-access
```

This command creates a three-node cluster in the `eu-central-1` region. The regions are listed at https://aws.amazon.com/about-aws/global-infrastructure/regional-product-services/.

INSPECTING AN EKS WORKER NODE

If you're interested in what's running on those nodes, you can use SSH to connect to them. The `--ssh-access` flag used in the command that creates the cluster ensures that your SSH public key is imported to the node.

As with GKE and Minikube, once you've logged into the node, you can try to list all running containers with `docker ps`. You can expect to see similar containers as in the clusters we covered earlier.

3.1.6 Deploying a multi-node cluster from scratch

Until you get a deeper understanding of Kubernetes, I strongly recommend that you don't try to install a multi-node cluster from scratch. If you are an experienced systems administrator, you may be able to do it without much pain and suffering, but most people may want to try one of the methods described in the previous sections first. Proper management of Kubernetes clusters is incredibly difficult. The installation alone is a task not to be underestimated.

If you still feel adventurous, you can start with the instructions in Appendix B, which explain how to create VMs with VirtualBox and install Kubernetes using the kubeadm tool. You can also use those instructions to install Kubernetes on your bare-metal machines or in VMs running in the cloud.

Once you've successfully deployed one or two clusters using kubeadm, you can then try to deploy it completely manually, by following Kelsey Hightower's *Kubernetes the Hard Way* tutorial at github.com/kelseyhightower/Kubernetes-the-hard-way. Though you may run into several problems, figuring out how to solve them can be a great learning experience.

65

登录节点后，您可以尝试使用 docker ps 列出所有正在运行的容器。您还没有运行任何应用程序，因此您只会看到 Kubernetes 系统容器。它们是什么现在并不重要，但您将在后面的章节中了解它们。

3.1.5 使用 Amazon Elastic Kubernetes Service 创建集群

如果您更喜欢使用 Amazon 而不是 Google 在云中部署 Kubernetes 集群，您可以尝试 Amazon Elastic Kubernetes Service (EKS)。让我们回顾一下基础知识。

首先，您必须按照以下网址的说明安装 eksctl 命令行工具

创建 EKS Kubernetes 集群

使用 eksctl 创建 EKS Kubernetes 集群与在 GKE 中创建集群的方式没有显着差异。您所要做的就是运行以下命令：

```
$ eksctl 创建集群 --name kubia --region eu-central-1 --nodes 3 --ssh-access
```

此命令在 eu-central-1 区域中创建一个三节点集群。这些地区列于

服务/。

检查 EKS 工作节点

如果您对这些节点上运行的内容感兴趣，可以使用 SSH 连接到它们。创建集群的命令中使用的 --ssh-access 标志可确保您的 SSH 公钥导入到节点。

与 GKE 和 Minikube 一样，登录节点后，您可以尝试使用 docker ps 列出所有正在运行的容器。您可以期望看到与我们之前介绍的集群中类似的容器。

3.1.6 从头开始部署多节点集群

在您对 Kubernetes 有了更深入的了解之前，我强烈建议您不要尝试从头开始安装多节点集群。如果您是一位经验丰富的系统管理员，您可能可以在没有太多痛苦的情况下完成此操作，但大多数人可能想首先尝试前面几节中描述的方法之一。正确管理 Kubernetes 集群非常困难。仅安装一项就是一项不可低估的任务。

如果您仍然喜欢冒险，可以从附录 B 中的说明开始，其中解释了如何使用 VirtualBox 创建虚拟机并使用 kubeadm 工具安装 Kubernetes。您还可以使用这些说明在裸机或在云中运行的虚拟机上安装 Kubernetes。

使用 kubeadm 成功部署一两个集群后，您可以按照 Kelsey Hightower 的 Kubernetes the Hard Way 教程（位于 github.com/kelseyhightower/Kubernetes-the-hard-way）尝试完全手动部署。尽管您可能会遇到几个问题，但弄清楚如何解决它们可能是一次很好的学习经历。

3.2 Interacting with Kubernetes

You've now learned about several possible methods to deploy a Kubernetes cluster. Now's the time to learn how to use the cluster. To interact with Kubernetes, you use a command-line tool called `kubectl`, pronounced *kube-control*, *kube-C-T-L* or *kube-cuddle*.

As the next figure shows, the tool communicates with the Kubernetes API server, which is part of the Kubernetes Control Plane. The control plane then triggers the other components to do whatever needs to be done based on the changes you made via the API.

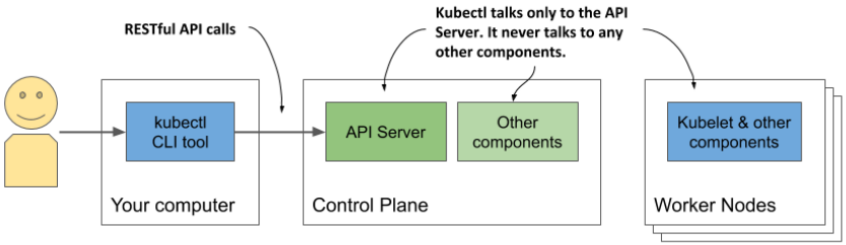


Figure 3.6 How you interact with a Kubernetes cluster

3.2.1 Setting up kubectl - the Kubernetes command-line client

Kubectl is a single executable file that you must download to your computer and place into your path. It loads its configuration from a configuration file called *kubeconfig*. To use kubectl, you must both install it and prepare the kubeconfig file so kubectl knows what cluster to talk to.

Downloading and Installing kubectl

The latest stable release for Linux can be downloaded and installed with the following command:

```
$ curl -LO https://storage.googleapis.com/kubernetes-release/release/$(curl -s
https://storage.googleapis.com/kubernetes-
release/release/stable.txt)/bin/linux/amd64/kubectl && chmod +x kubectl && sudo mv
kubectl /usr/local/bin/
```

To install `kubectl` on macOS, you can either run the same command, but replace `linux` in the URL with `darwin`, or install the tool via Homebrew by running `brew install kubectl`.

On Windows, download `kubectl.exe` from https://storage.googleapis.com/kubernetes-release/release/v1.18.2/bin/windows/amd64/kubectl.exe. To download the latest version, first go to https://storage.googleapis.com/kubernetes-release/release/stable.txt to see what the latest stable version is and then replace the version number in the first URL with this version. To check if you've installed it correctly, run `kubectl --help`. Note that kubectl may

3.2 与 Kubernetes 交互

您现在已经了解了部署 Kubernetes 集群的几种可能的方法。现在是学习如何使用集群的时候了。要与 Kubernetes 交互，您可以使用名为 kubectl 的命令行工具，发音为 kube-control、kube-C-T-L 或 kube-cuddle。

如下图所示，该工具与 Kubernetes API 服务器进行通信，该服务器是 Kubernetes 控制平面的一部分。然后，控制平面根据您通过 API 所做的更改触发其他组件执行需要执行的操作。

图 3.6 如何与 Kubernetes 集群交互

3.2.1 设置 kubectl - Kubernetes 命令行客户端

Kubectl 是一个可执行文件，您必须将其下载到您的计算机并放入您的路径中。它从名为 kubeconfig 的配置文件加载其配置。要使用 kubectl，您必须安装它并准备 kubeconfig 文件，以便 kubectl 知道要与哪个集群通信。

下载并安装 KUBECTL

可以使用以下命令下载并安装 Linux 的最新稳定版本：

```
$curl -LO https://storage.googleapis.com/kubernetes-
release/release/$(curl -s
release/release/stable.txt)/bin/linux/amd64/kubectl && chmod +x
kubectl && sudo mv kubectl /usr/local/bin/
```

要在 macOS 上安装 kubectl，您可以运行相同的命令，但将 linux 替换为

使用 darwin 获取 URL，或者通过运行brew install kubectl 通过 Homebrew 安装该工具。发布/release/v1.18.2/bin/windows/amd64/kubectl.exe。要下载最新版本，请先访问 https://storage.googleapis.com/kubernetes-release/release/stable.txt 查看最新的稳定版本是什么，然后将第一个 URL 中的版本号替换为该版本。要检查是否已正确安装，请运行 kubectl --help

or may not yet be configured to talk to your Kubernetes cluster, which means most commands may not work yet.

> **TIP** You can always append `--help` to any `kubectl` command to get information on what it does and how to use it.

SETTING UP A SHORT ALIAS FOR KUBECTL

You'll use `kubectl` often. Having to type the full command every time is needlessly time-consuming, but you can speed things up by setting up an alias and tab completion for it.

Most users of Kubernetes use `k` as the alias for kubectl. If you haven't used aliases yet, here's how to define it in Linux and macOS. Add the following line to your `~/.bashrc` or equivalent file:

```
alias k=kubectl
```

On Windows, if you use the Command Prompt, define the alias by executing `doskey k=kubectl $*`. If you use PowerShell, execute `set-alias -name k -value kubectl`.

> **NOTE** You may not need an alias if you used `gcloud` to set up the cluster. It installs the `k` binary in addition to `kubectl`.

CONFIGURING TAB COMPLETION FOR KUBECTL

Even with a short alias like `k`, you'll still have to type a lot. Fortunately, the `kubectl` command can also output shell completion code for both the bash and the zsh shell. It enables tab completion of not only command names but also the object names. For example, later you'll learn how to view details of a particular cluster node by executing the following command:

```
$ kubectl describe node gke-kubia-default-pool-9bba9b18-4glf
```

That's a lot of typing that you'll repeat all the time. With tab completion, things are much easier. You just press TAB after typing the first few characters of each token:

```
$ kubectl desc<TAB> no<TAB> gke-ku<TAB>
```

To enable tab completion in bash, you must first install a package called `bash-completion` and then run the following command (you can also add it to `~/.bashrc` or equivalent):

```
$ source <(kubectl completion bash)
```

But there's one caveat. This will only complete your commands when you use the full `kubectl` command name. It won't work when you use the `k` alias. To make it work with the alias, you must transform the output of the `kubectl completion` command using the `sed` tool:

```
$ source <(kubectl completion bash | sed s/kubectl/k/g)
```


或者可能尚未配置为与您的 Kubernetes 集群通信，这意味着大多数命令可能尚无法运行。

> 提示您始终可以将 --help 附加到任何 kubectl 命令以获取有关其功能和操作方式的信息使用它。

为 KUBECTL 设置短别名

您会经常使用 kubectl。每次都必须输入完整的命令是不必要的耗时，但是您可以通过为其设置别名和制表符完成来加快速度。

Kubernetes 的大多数用户使用 k 作为 kubectl 的别名。如果您还没有使用过别名，请参阅以下在 Linux 和 macOS 中定义别名的方法。将以下行添加到 ~/.bashrc 或等效文件中：

```
别名 k=kubectl
```

在 Windows 上，如果使用命令提示符，请通过执行 doskey k=kubectl $* 来定义别名。如果使用 PowerShell，请执行 set-alias -name k -value kubectl。

> 注意 如果您使用
> gcloud 设置集
> 群，则可能不需要
> 别名。它安装
> kubectl 之外，它
> 还安装 k 二进制文
> 件。

为 KUBECTL 配置制表符补全

即使使用像 k 这样的短别名，您仍然需要输入很多内容。幸运的是，kubectl 命令还可以输出 bash 和 zsh shell 的 shell 完成代码。它不仅可以完成命令名称，还可以完成对象名称。例如，稍后您将学习如何通过执行以下命令来查看特定集群节点的详细信息：

```
$ kubectl 描述节点 gke-kubia-default-pool-9bba9b18-4glf
```

您会一直重复输入大量内容。通过制表符补全，事情就变得容易多了。您只需在输入每个标记的前几个字符后按 TAB 键即可：

```
$ kubectl desc no gke-ku
```

要在 bash 中启用制表符补全，您必须首先安装一个名为 bash-completion 的包，然后运行以下命令（您也可以将其添加到 ~/.bashrc 或等效命令）：

```
$ source <(kubectl 完成 bash)
```

但有一点需要注意。仅当您使用完整的 kubectl 命令名称时，这才会完成您的命令。当您使用 k 别名时它将不起作用。为了使其与

别名，您必须使用 sed 工具转换 kubectl 完成命令的输出：

3.2.2 Configuring kubectl to use a specific Kubernetes cluster

The kubeconfig configuration file is located at `~/.kube/config`. If you deployed your cluster using Docker Desktop, Minikube or GKE, the file was created for you. If you've been given access to an existing cluster, you should have received the file. Other tools, such as kind, may have written the file to a different location. Instead of moving the file to the default location, you can also point kubectl to it by setting the `KUBECONFIG` environment variable as follows:

```
$ export KUBECONFIG=/path/to/custom/kubeconfig
```

To learn more about how to manage kubectl's configuration and create a config file from scratch, refer to appendix A.

NOTE If you want to use several Kubernetes clusters (for example, both Minikube and GKE), see appendix A for information on switching between different `kubectl` contexts.

3.2.3 Using kubectl

Assuming you've installed and configured kubectl, you can now use it to talk to your cluster.

VERIFYING IF THE CLUSTER IS UP AND KUBECTL CAN TALK TO IT

To verify that your cluster is working, use the `kubectl cluster-info` command shown in the following listing.

Listing 3.9 Displaying cluster information

```
$ kubectl cluster-info
Kubernetes master is running at https://192.168.99.101:8443
KubeDNS is running at https://192.168.99.101:8443/api/v1/namespaces/...
```

This indicates that the API server is active and responding to requests. The output lists the URLs of the various Kubernetes cluster services running in your cluster. The above example shows that besides the API server, the KubeDNS service, which provides domain-name sevices within the cluster, is another service that runs in the cluster.

LISTING CLUSTER NODES

Now use the `kubectl` command to list all nodes in your cluster. The following listing shows the output that is generated when executing kubectl against a GKE cluster with three nodes.

Listing 3.10 Listing cluster nodes with `kubectl`

```
$ kubectl get nodes
NAME            STATUS   ROLES    AGE   VERSION
control-plane   Ready    <none>   12m   v1.18.2
worker1         Ready    <none>   12m   v1.18.2
worker2         Ready    <none>   12m   v1.18.2
```


3.2.2 配置kubectl使用特定的Kubernetes集群

kubeconfig 配置文件位于 ~/.kube/config。如果您使用 Docker Desktop、Minikube 或 GKE 部署集群，则会为您创建该文件。如果您已获得对现有集群的访问权限，则您应该已经收到该文件。其他工具（例如 kind）可能已将文件写入到不同的位置。您还可以通过设置 KUBECONFIG 环境变量将 kubectl 指向该文件，而不是将文件移动到默认位置，如下所示：

```
$ 导出 KUBECONFIG =/path/to/custom/kubeconfig
```

要了解有关如何管理 kubectl 配置并从头开始创建配置文件的更多信息，请参阅附录 A。

注意 如果您想使用多个 Kubernetes 集群（例如 Minikube 和 GKE），请参阅附录 A 了解有关在不同 kubectl 上下文之间切换的信息。

3.2.3 使用kubectl

假设您已经安装并配置了 kubectl，您现在可以使用它与您的集群进行通信。

验证集群是否已启动并且 KUBECTL 是否可以与其通信

要验证集群是否正常工作，请使用以下清单中所示的 kubectl cluster-info 命令。

清单 3.9 显示集群信息

```
$ kubectl 集群信息
```

这表明 API 服务器处于活动状态并正在响应请求。输出列出了集群中运行的各种 Kubernetes 集群服务的 URL。从上面的例子可以看出，除了API Server之外，集群内运行的另一个服务是KubeDNS服务，它在集群内提供域名服务。

列出集群节点

现在使用 kubectl 命令列出集群中的所有节点。以下列表显示了对具有三个节点的 GKE 集群执行 kubectl 时生成的输出。

清单 3.10 使用 kubectl 列出集群节点

```
$ kubectl get 节点名称 状态 角色 年龄 版本
控制平面 就绪 <无> 12m v1.18.2
worker1 就绪 <无> 12m v1.18.2
```

Everything in Kubernetes is represented by an object and can be retrieved and manipulated via the RESTful API. The `kubectl get` command retrieves a list of objects of the specified type from the API. You'll use this command all the time, but it only displays summary information about the listed objects.

RETRIEVING ADDITIONAL DETAILS OF AN OBJECT

To see more detailed information about an object, you use the `kubectl describe` command, which shows much more:

```
$ kubectl describe node gke-kubia-85f6-node-0rrx
```

I omit the actual output of the `describe` command because it's quite wide and would be completely unreadable here in the book. If you run the command yourself, you'll see that it displays the status of the node, information about its CPU and memory usage, system information, containers running on the node, and much more.

If you run the `kubectl describe` command without specifying the resource name, information of all nodes will be printed.

> **TIP** Executing the `describe` command without specifying the object name is useful when only one object of a certain type exists. You don't have to type or copy/paste the object name.

You'll learn more about the numerous other kubectl commands throughout the book.

3.2.4 Interacting with Kubernetes through web dashboards

If you prefer using graphical web user interfaces, you'll be happy to hear that Kubernetes also comes with a nice web dashboard. Note, however, that the functionality of the dashboard may lag significantly behind kubectl, which is the primary tool for interacting with Kubernetes.

Nevertheless, the dashboard shows different resources in context and can be a good start to get a feel for what the main resource types in Kubernetes are and how they relate to each other. The dashboard also offers the possibility to modify the deployed objects and displays the equivalent kubectl command for each action - a feature most beginners will appreciate.

Figure 3.7 shows the dashboard with two workloads deployed in the cluster.

69

Kubernetes 中的所有内容都由对象表示，并且可以通过 RESTful API 进行检索和操作。 kubectl get 命令从 API 检索指定类型的对象列表。您将一直使用此命令，但它仅显示有关列出的对象的摘要信息。

检索对象的附加详细信息

要查看有关对象的更多详细信息，您可以使用 kubectl describe 命令，该命令会显示更多信息：

```
$ kubectl 描述节点 gke-kubia-85f6-node-0rrx
```

我省略了描述命令的实际输出，因为它非常宽，在本书中完全无法阅读。如果您自己运行该命令，您将看到它显示节点的状态、有关其 CPU 和内存使用情况的信息、系统信息、节点上运行的容器等等。

如果运行 kubectl describe 命令时未指定资源名称，
将打印所有节点的信息。

> 提示 当某种类型的对象仅存在一个时，执行描述命令而不指定对象名称非常有用。您不必键入或复制/粘贴对象名称。

您将在本书中了解更多有关许多其他 kubectl 命令的信息。

3.2.4 通过Web仪表板与Kubernetes交互

如果您更喜欢使用图形 Web 用户界面，您会很高兴听到 Kubernetes 还附带了一个漂亮的 Web 仪表板。但请注意，仪表板的功能可能明显落后于 kubectl，而 kubectl 是与 Kubernetes 交互的主要工具。

尽管如此，仪表板在上下文中显示了不同的资源，并且可以作为了解 Kubernetes 中的主要资源类型以及它们之间的关系的良好开端。仪表板还提供了修改已部署对象的可能性，并为每个操作显示等效的 kubectl 命令 - 这是大多数初学者都会欣赏的功能。图 3.7 显示了集群中部署了两个工作负载的仪表板。

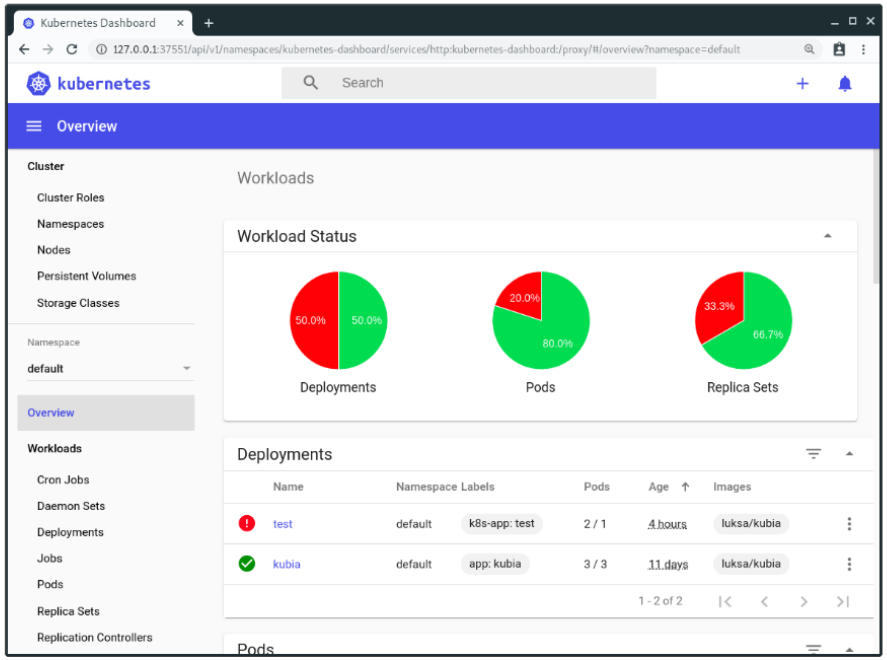


Figure 3.7 Screenshot of the Kubernetes web-based dashboard

Although you won't use the dashboard in this book, you can always open it to quickly see a graphical view of the objects deployed in your cluster after you create them via kubectl.

Accessing the dashboard in Docker Desktop

Unfortunately, Docker Desktop does not install the Kubernetes dashboard by default. Accessing it is also not trivial, but here's how. First, you need to install it using the following command:

```
$ kubectl apply -f https://raw.githubusercontent.com/kubernetes/dashboard/v2.0.0-
rc5/aio/deploy/recommended.yaml
```

Refer to github.com/kubernetes/dashboard to find the latest version number. After installing the dashboard, the next command you must run is:

```
$ kubectl proxy
```


图 3.7 Kubernetes 基于 Web 的仪表板的屏幕截图

尽管您不会使用本书中的仪表板，但在通过 kubectl 创建集群中部署的对象后，您始终可以打开它来快速查看集群中部署的对象的图形视图。

访问 Docker Desktop 中的仪表板

不幸的是，Docker Desktop 默认情况下不安装 Kubernetes 仪表板。访问它也并非易事，但方法如下。首先，您需要使用以下命令安装它：rc5/aio/deploy/recommended.yaml

请参阅 github.com/kubernetes/dashboard 查找最新版本号。安装仪表板后，您必须运行的下一个命令是：

This command runs a local proxy to the API server, allowing you to access the services through it. Let the proxy process run and use the browser to open the dashboard at the following URL:

http://localhost:8001/api/v1/namespaces/kubernetes-dashboard/services/https:kubernetes-dashboard:/proxy/

You'll be presented with an authentication page. You must then run the following command to retrieve an authentication token.

```
$ kubectl -n kubernetes-dashboard describe secret $(kubectl -n kubernetes-dashboard
get secret | sls admin-user | ForEach-Object { $_ -Split '\s+' } | Select -First 1)
```

NOTE This command must be run in Windows PowerShell.

Find the token listed under `kubernetes-dashboard-token-xyz` and paste it into the token field on the authentication page shown in your browser. After you do this, you should be able to use the dashboard. When you're finished using it, terminate the `kubectl proxy` process using Control-C.

ACCESSING THE DASHBOARD WHEN USING MINIKUBE

If you're using Minikube, accessing the dashboard is much easier. Run the following command and the dashboard will open in your default browser:

```
$ minikube dashboard
```

ACCESSING THE DASHBOARD WHEN RUNNING KUBERNETES ELSEWHERE

The Google Kubernetes Engine no longer provides access to the open source Kubernetes Dashboard, but it offers an alternative web-based console. The same applies to other cloud providers. For information on how to access the dashboard, please refer to the documentation of the respective provider.

If your cluster runs on your own infrastructure, you can deploy the dashboard by following the guide at kubernetes.io/docs/tasks/access-application-cluster/web-ui-dashboard.

3.3 Running your first application on Kubernetes

Now is the time to finally deploy something to your cluster. Usually, to deploy an application, you'd prepare a JSON or YAML file describing all the components that your application consists of and apply that file to your cluster. This would be the declarative approach.

Since this may be your first time deploying an application to Kubernetes, let's choose an easier way to do this. We'll use simple, one-line imperative commands to deploy your application.

3.3.1 Deploying your application

The imperative way to deploy an application is to use the `kubectl create deployment` command. As the command itself suggests, it creates a *Deployment* object, which represents

此命令运行 API 服务器的本地代理，允许您通过它访问服务。让代理进程运行并使用浏览器打开以下 URL 的仪表板：

http://localhost:8001/api/v1/namespaces/kubernetesdashboard/services/https:kubernetes-dashboard:/proxy/

您将看到一个身份验证页面。然后您必须运行以下命令
命令检索身份验证令牌。

```
$ kubectl -n kubernetes-dashboard 描述秘密 $(kubectl -n kubernetes-
dashboard 获取秘密 | sls admin-user | ForEach-Object { $_ -Split '\s+' }
| Select -First 1)
```

注意 此命令必须在 Windows PowerShell 中运行。

找到 kubernetes-dashboard-token-xyz 下列出的令牌，并将其粘贴到浏览器中显示的身份验证页面上的令牌字段中。执行此操作后，您应该能够使用仪表板。使用完毕后，使用 Control-C 终止 kubectl 代理进程。

使用 MINIKUBE 时访问仪表板

如果您使用 Minikube，访问仪表板会更容易。运行以下命令，仪表板将在默认浏览器中打开：

```
$ minikube 仪表板
```

在其他地方运行 Kubernetes 时访问仪表板

Google Kubernetes Engine 不再提供对开源 Kubernetes 仪表板的访问，但它提供了替代的基于 Web 的控制台。这同样适用于其他云提供商。有关如何访问仪表板的信息，请参阅

相应提供商的文档。

如果您的集群在您自己的基础设施上运行，您可以按照 kubernetes.io/docs/tasks/access-application-cluster/web-ui-dashboard 上的指南部署仪表板。

3.3 在 Kubernetes 上运行您的第一个应用程序

现在是最终将某些内容部署到集群的时候了。通常，要部署应用程序，您需要准备一个 JSON 或 YAML 文件来描述应用程序包含的所有组件，并将该文件应用到集群。这将是声明性方法。

由于这可能是您第一次将应用程序部署到 Kubernetes，因此我们选择一种更简单的方法来执行此操作。我们将使用简单的一行命令式命令来部署您的应用程序。

3.3.1 部署应用程序

部署应用程序的必要方法是使用 `kubectl create` 部署命令。正如命令本身所示，它创建了一个 `Deployment` 对象

an application deployed in the cluster. By using the imperative command, you avoid the need to know the structure of Deployment objects as when you write YAML or JSON manifests.

Creating a Deployment

In the previous chapter, you created a Node.js application that you packaged into a container image and pushed to Docker Hub to make it easily distributable to any computer. Let's deploy that application to your Kubernetes cluster. Here's the command you need to execute:

```
$ kubectl create deployment kubia --image=luksa/kubia:1.0
deployment.apps/kubia created
```

You've specified three things here:

- You want to create a `deployment` object.
- You want the object to be called `kubia`.
- You want the deployment to use the container image `luksa/kubia:1.0`.

By default, the image is pulled from Docker Hub, but you can also specify the image registry in the image name (for example, `quay.io/luksa/kubia:1.0`).

NOTE Make sure that the image is stored in a public registry and can be pulled without access authorization. You'll learn how to provide credentials for pulling private images in chapter 8.

The Deployment object is now stored in the Kubernetes API. The existence of this object tells Kubernetes that the `luksa/kubia:1.0` container must run in your cluster. You've stated your *desired* state. Kubernetes must now ensure that the *actual* state reflects your wishes.

Listing deployments

The interaction with Kubernetes consists mainly of the creation and manipulation of objects via its API. Kubernetes stores these objects and then performs operations to bring them to life. For example, when you create a Deployment object, Kubernetes runs an application. Kubernetes then keeps you informed about the current state of the application by writing the status to the same Deployment object. You can view the status by reading back the object. One way to do this is to list all Deployment objects as follows:

```
$ kubectl get deployments
NAME    READY   UP-TO-DATE   AVAILABLE   AGE
kubia   0/1     1            0           6s
```

The `kubectl get deployments` command lists all Deployment objects that currently exist in the cluster. You have only one Deployment in your cluster. It runs one instance of your application as shown in the `UP-TO-DATE` column, but the `AVAILABLE` column indicates that the application is not yet available. That's because the container isn't ready, as shown in the `READY` column. You can see that zero of a total of one container are ready.

You may wonder if you can ask Kubernetes to list all the running containers by running `kubectl get containers`. Let's try this.

72

部署在集群中的应用程序。通过使用命令式命令，您无需像编写 YAML 或 JSON 清单时那样了解 Deployment 对象的结构。

创建部署

在上一章中，您创建了一个 Node.js 应用程序，将其打包到容器映像中并推送到 Docker Hub，以使其可以轻松分发到任何计算机。让我们将该应用程序部署到您的 Kubernetes 集群。这是您需要执行的命令：

```
$ kubectl 创建部署 kubia --image=luksa/kubia:1.0 部
署.apps/kubia 创建
```

您在这里指定了三件事：

- 您想要创建部署对象。
- 您希望该对象被称为kubia。
- 您希望部署使用容器映像luksa/kubia:1.0

默认情况下，映像是从 Docker Hub 拉取的，但您也可以在映像名称中指定映像注册表（例如 quay.io/luksa/kubia:1.0）。

注意 确保镜像存储在公共注册表中，并且无需访问授权即可提取。您将在第 8 章中了解如何提供用于拉取私有映像的凭据。

Deployment 对象现在存储在 Kubernetes API 中。该对象的存在告诉 Kubernetes luksa/kubia:1.0 容器必须在您的集群中运行。你已经说出了你想要的状态。 Kubernetes 现在必须确保实际状态反映您的意愿。

列出部署

与 Kubernetes 的交互主要包括通过其 API 创建和操作对象。 Kubernetes 存储这些对象，然后执行操作使它们栩栩如生。例如，当您创建 Deployment 对象时，Kubernetes 会运行一个应用程序。然后，Kubernetes 通过将状态写入同一个 Deployment 对象来让您了解应用程序的当前状态。您可以通过读回对象来查看状态。执行此操作的一种方法是列出所有 Deployment 对象，如下所示：

```
$ kubectl get 部署名称 READY UP-TO-DATE
AVAILABLE AGE
库比亚 0/1 1 0 6秒
```

kubectl getDeployments 命令列出集群中当前存在的所有 Deployment 对象。您的集群中只有一个 Deployment。它运行应用程序的一个实例，如“UP-TO-DATE”列中所示，但“AVAILABLE”列表明该应用程序尚不可用。这是因为容器尚未准备好，如 READY 列中所示。您可以看到总共 1 个容器中有 0 个已准备就绪。

您可能想知道是否可以通过运行来要求 Kubernetes 列出所有正在运行的容器 kubectl 获取容器。让我们试试这个。

```
$ kubectl get containers
error: the server doesn't have a resource type "containers"
```

The command fails because Kubernetes doesn't have a "Container" object type. This may seem odd, since Kubernetes is all about running containers, but there's a twist. A container is not the smallest unit of deployment in Kubernetes. So, what is?

Introducing Pods

In Kubernetes, instead of deploying individual containers, you deploy groups of co-located containers – so-called *pods*. You know, as in *pod of whales*, or a *pea pod*.

A pod is a group of one or more closely related containers (not unlike peas in a pod) that run together on the same worker node and need to share certain Linux namespaces, so that they can interact more closely than with other pods.

In the previous chapter I showed an example where two processes use the same namespaces. By sharing the network namespace, both processes use the same network interfaces, share the same IP address and port space. By sharing the UTS namespace, both see the same system hostname. This is exactly what happens when you run containers in the same pod. They use the same network and UTS namespaces, as well as others, depending on the pod's spec.

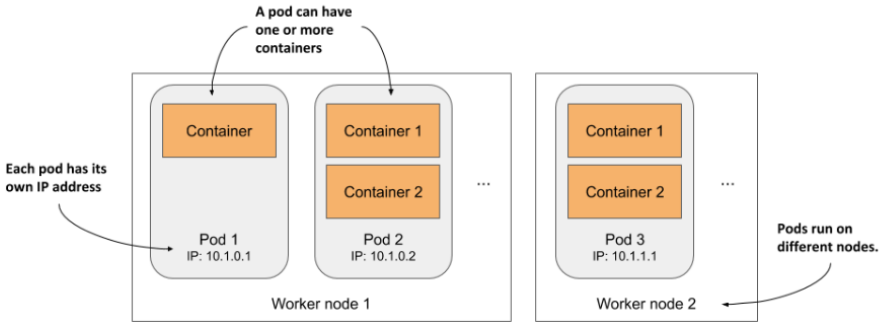


Figure 3.8 The relationship between containers, pods, and worker nodes

As illustrated in figure 3.8, you can think of each pod as a separate logical computer that contains one application. The application can consist of a single process running in a container, or a main application process and additional supporting processes, each running in a separate container. Pods are distributed across all the worker nodes of the cluster.

Each pod has its own IP, hostname, processes, network interfaces and other resources. Containers that are part of the same pod think that they're the only ones running on the computer. They don't see the processes of any other pod, even if located on the same node.

73

```
$ kubectl 获取容器
```

该命令失败，因为 Kubernetes 没有“Container”对象类型。这可能看起来很奇怪，因为 Kubernetes 就是运行容器，但有一个转折点。容器并不是 Kubernetes 中最小的部署单元。那么，什么是?

Pod 简介

在 Kubernetes 中，您不是部署单个容器，而是部署一组共置容器（即所谓的 Pod）。你知道，就像鲸鱼群或豌豆荚一样。

Pod 是一组一个或多个紧密相关的容器（与 Pod 中的豌豆不同），它们在同一工作节点上一起运行，并且需要共享某些 Linux 命名空间，以便它们可以比与其他 Pod 更紧密地交互。

在上一章中，我展示了一个示例，其中两个进程使用相同的命名空间。通过共享网络命名空间，两个进程使用相同的网络接口、共享相同的 IP 地址和端口空间。通过共享 UTS 命名空间，两者可以看到相同的系统主机名。这正是在同一个 Pod 中运行容器时发生的情况。它们使用相同的网络和 UTS 命名空间以及其他命名空间，具体取决于 Pod 的规格。

图3.8 容器、pod、worker节点之间的关系

如图 3.8 所示，您可以将每个 Pod 视为包含一个应用程序的单独逻辑计算机。应用程序可以由在容器中运行的单个进程组成，也可以由主应用程序进程和其他支持进程组成，每个进程都在单独的容器中运行。 Pod 分布在集群的所有工作节点上。

每个 Pod 都有自己的 IP、主机名、进程、网络接口和其他资源。属于同一 Pod 的容器认为它们是唯一在计算机上运行的容器。他们看不到任何其他 Pod 的进程，即使位于同一节点上也是如此。

LISTING PODS

Since containers aren't a top-level Kubernetes object, you can't list them. But you can list pods. As the next listing shows, by creating the Deployment object, you've deployed one pod.

Listing 3.11 Listing pods

```
$ kubectl get pods
NAME                     READY     STATUS    RESTARTS   AGE
kubia-9d785b578-p449x    0/1       Pending   0          1m
```

This is the pod that houses the container running your application. To be precise, since the status is still `Pending`, the application, or rather the container, isn't running yet. This is also expressed in the `READY` column, which indicates that the pod has a single container that's not ready.

The reason the pod is pending is because the worker node to which the pod has been assigned must first download the container image before it can run it. When the download is complete, the pod's container is created and the pod enters the `Running` state.

If Kubernetes can't pull the image from the registry, the `kubectl get pods` command will indicate this in the `STATUS` column. If you're using your own image, ensure it's marked as public on Docker Hub. Try pulling the image manually with the `docker pull` command on another computer.

If another issue is causing your pod not to run, or if you simply want to see more information about the pod, you can also use the `kubectl describe pod` command, as you did earlier to see the details of a worker node. If there are any issues with the pod, they should be displayed by this command. Look at the events shown at the bottom of its output. For a running pod, they should be similar to the following listing.

Listing 3.12 The events displayed by kubectl describe pod

```
Events:
Type    Reason     Age   From               Message
----    ------     ----  ----               -------
Normal  Scheduled  25s   default-scheduler  Successfully assigned
                                            default/kubia-9d785b578-p449x
                                            to worker2
Normal  Pulling    23s   kubelet, worker2   Pulling image "luksa/kubia:1.0"
Normal  Pulled     21s   kubelet, worker2   Successfully pulled image
Normal  Created    21s   kubelet, worker2   Created container kubia
Normal  Started    21s   kubelet, worker2   Started container kubia
```

UNDERSTANDING WHAT HAPPENS BEHIND THE SCENES

To help you visualize what happened when you created the Deployment, see figure 3.9.

74

列表 Pod

由于容器不是顶级 Kubernetes 对象，因此您无法列出它们。但您可以列出 pod。如下清单所示，通过创建 Deployment 对象，您已经部署了一个 Pod。

清单 3.11 列出 pod

```
$ kubectl  get pods NAME READY  STATUS RESTARTS  AGE
kubia-9d785b578-p449x 0/1 待定          0 1米
```

这是容纳运行应用程序的容器的 Pod。准确地说，由于状态仍为 Pending，因此应用程序（或者更确切地说容器）尚未运行。这也体现在 READY 列中，这表明 pod 有一个容器尚未准备好。

Pod 处于挂起状态的原因是，已分配 Pod 的工作节点必须先下载容器镜像，然后才能运行它。下载完成后，Pod的容器被创建，Pod进入Running状态。

如果 Kubernetes 无法从注册表中提取映像，kubectl get pods 命令将在 STATUS 列中指示这一点。如果您使用自己的图像，请确保将其标记为

在 Docker Hub 上公开。尝试在另一台计算机上使用 docker pull 命令手动拉取映像。

如果另一个问题导致您的 pod 无法运行，或者您只是想查看有关 pod 的更多信息，您还可以使用 kubectl describe pod 命令，就像您之前查看工作节点的详细信息一样。如果 Pod 存在任何问题，应通过此命令显示这些问题。查看其输出底部显示的事件。对于正在运行的 Pod，它们应该类似于以下清单。

清单 3.12 kubectl 描述 pod 显示的事件

```
事件：类型  原因  消息 年龄
____   ______    ____ ____           _______
正常 调度 25秒 default-scheduler 成功 分配 default/kubia-9d785b578-
p449x 给 worker2
正常 拉取  23s  kubelet 、worker2  拉取镜像 “luksa/kubia:1.0”
正常 拉取 21s  kubelet , worker2 拉取镜像成功
正常 创建了 21s  kubelet ， worker2 创建了容器 kubia
```

了解幕后发生的事情

为了帮助您可视化创建部署时发生的情况，请参见图 3.9。

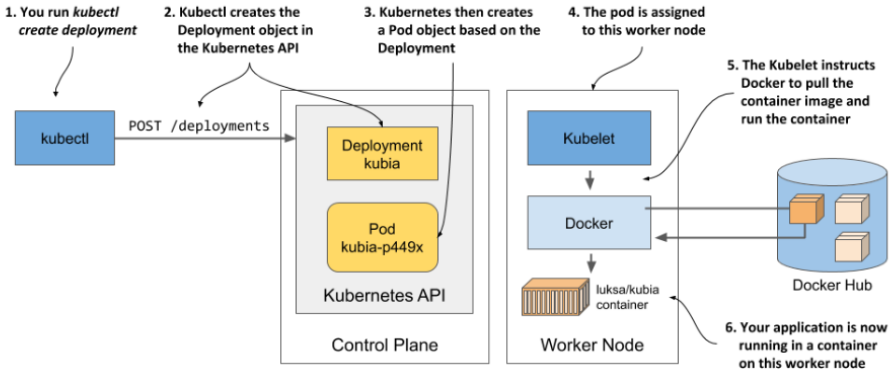


Figure 3.9 How creating a Deployment object results in a running application container

When you ran the `kubectl create` command, it created a new Deployment object in the cluster by sending an HTTP request to the Kubernetes API server. Kubernetes then created a new Pod object, which was then assigned or *scheduled* to one of the worker nodes. The Kubernetes agent on the worker node (the Kubelet) became aware of the newly created Pod object, saw that it was scheduled to its node, and instructed Docker to pull the specified image from the registry, create a container from the image, and execute it.

> **DEFINITION** The term scheduling refers to the assignment of the pod to a node. The pod runs immediately, not at some point in the future. Just like how the CPU scheduler in an operating system selects what CPU to run a process on, the scheduler in Kubernetes decides what worker node should execute each container. Unlike an OS process, once a pod is assigned to a node, it runs only on that node. Even if it fails, this instance of the pod is never moved to other nodes, as is the case with CPU processes, but a new pod instance may be created to replace it.

Depending on what you use to run your Kubernetes cluster, the number of worker nodes in your cluster may vary. The figure shows only the worker node that the pod was scheduled to. In a multi-node cluster, none of the other worker nodes are involved in the process.

3.3.2 Exposing your application to the world

Your application is now running, so the next question to answer is how to access it. I mentioned that each pod gets its own IP address, but this address is internal to the cluster and not accessible from the outside. To make the pod accessible externally, you'll *expose* it by creating a Service object.

Several types of Service objects exist. You decide what type you need. Some expose pods only within the cluster, while others expose them externally. A service with the type

图 3.9 创建 Deployment 对象如何导致运行应用程序容器

当您运行 kubectl create 命令时，它通过向 Kubernetes API 服务器发送 HTTP 请求在集群中创建一个新的 Deployment 对象。然后，Kubernetes 创建一个新的 Pod 对象，然后将其分配或调度到其中一个工作节点。工作节点（Kubelet）上的 Kubernetes 代理意识到新创建的 Pod 对象，发现它被调度到其节点，并指示 Docker 从注册表中提取指定的镜像，从该镜像创建一个容器，并执行它。

> 定义 术语“调度”是指将 pod 分配给节点。 Pod 会立即运行，而不是在将来的某个时刻运行。就像操作系统中CPU调度器如何选择一样
> 在哪个 CPU 上运行进程，Kubernetes 中的调度程序决定哪个工作节点应该执行每个进程
> 容器。与操作系统进程不同，一旦将 Pod 分配给节点，它就仅在该节点上运行。就算失败了，
> 该 Pod 实例永远不会像 CPU 进程那样移动到其他节点，而是一个新的 Pod

根据运行 Kubernetes 集群所使用的内容，集群中的工作节点数量可能会有所不同。图中仅显示了 pod 被调度到的工作节点。在多节点集群中，其他工作节点都不参与该过程。

3.3.2 向全世界公开您的应用程序

您的应用程序现在正在运行，因此下一个要回答的问题是如何访问它。我提到每个 Pod 都有自己的 IP 地址，但该地址是集群内部的，无法从外部访问。为了使 Pod 可以从外部访问，您将通过创建一个 Service 对象来公开它。

存在多种类型的服务对象。您决定需要什么类型。有些仅在集群内公开 pod，而另一些则在外部公开

LoadBalancer provisions an external load balancer, which makes the service accessible via a public IP. This is the type of service you'll create now.

CREATING A SERVICE

The easiest way to create the service is to use the following imperative command:

```
$ kubectl expose deployment kubia --type=LoadBalancer --port 8080
service/kubia exposed
```

The `create deployment` command that you ran previously created a Deployment object, whereas the `expose deployment` command creates a Service object. This is what running the above command tells Kubernetes:

- You want to expose all pods that belong to the kubia Deployment as a new service.
- You want the pods to be accessible from outside the cluster via a load balancer.
- The application listens on port 8080, so you want to access it via that port.

You didn't specify a name for the Service object, so it inherits the name of the Deployment.

LISTING SERVICES

Services are API objects, just like Pods, Deployments, Nodes and virtually everything else in Kubernetes, so you can list them by executing `kubectl get services`, as in the next listing.

Listing 3.13 Listing Services

```
$ kubectl get svc
NAME         TYPE           CLUSTER-IP     EXTERNAL-IP   PORT(S)          AGE
kubernetes   ClusterIP      10.19.240.1    <none>        443/TCP          34m
kubia        LoadBalancer   10.19.243.17   <pending>     8080:30838/TCP   4s
```

NOTE Notice the use of the abbreviation `svc` instead of `services`. Most resource types have a short name that you can use instead of the full object type (for example, `po` is short for `pods`, `no` for `nodes` and `deploy` for `deployments`).

The list shows two services with their types, IPs and the ports they expose. Ignore the `kubernetes` service for now and take a close look at the `kubia` service. It doesn't yet have an external IP address. Whether it gets one depends on how you've deployed the cluster.

Listing the available object types with kubectl api-resources

You've used the `kubectl get` command to list various things in your cluster: Nodes, Deployments, Pods and now Services. These are all Kubernetes object types. You can display a list of all supported types by running `kubectl api-resources`. The list also shows the short name for each type and some other information you need to define objects in JSON/YAML files, which you'll learn in the following chapters.

76

LoadBalancer 提供外部负载均衡器，使服务可以通过公共 IP 访问。这就是您现在要创建的服务类型。

创建服务

创建服务的最简单方法是使用以下命令式命令：

```
$ kubectl 暴露部署 kubia --type=LoadBalancer --port 8080
服务/kubia 暴露
```

您之前运行的创建部署命令创建了一个部署对象，而公开部署命令创建了一个服务对象。运行上述命令会告诉 Kubernetes：

- 您希望将属于kubia 部署的所有pod 作为新服务公开。
- 您希望可以通过负载均衡器从集群外部访问 Pod。

您没有为 Service 对象指定名称，因此它继承了 Deployment 的名称。

上市服务

服务是 API 对象，就像 Pod、部署、节点以及 Kubernetes 中的几乎所有其他对象一样，因此您可以通过执行 kubectl get services 来列出它们，如下一个清单所示。

列表 3.13 列表服务

```
$ kubectl get svc 名称类型集群-IP 外部-IP 端口年龄
kubernetes ClusterIP 10.19.240.1 <无> 443/TCP 34m
kubia LoadBalancer 10.19.243.17 <待定> 8080:30838/TCP 4s
```

注意 请注意使用缩写 svc 而不是 services。大多数资源类型都有一个简短的名称，您可以使用它来代替完整的对象类型（例如，po 是 pod 的缩写，no 是节点的缩写，部署进行部署）。

该列表显示了两个服务及其类型、IP 和它们公开的端口。暂时忽略 kubernetes 服务，仔细看看 kubia 服务。目前还没有

外部 IP 地址。是否获得它取决于您如何部署集群。

使用 kubectl api-resources 列出可用的对象类型

您已经使用 kubectl get 命令列出了集群中的各种内容：节点、部署、Pod，现在

服务。这些都是 Kubernetes 对象类型。您可以通过运行 kubectl 显示所有支持类型的列表

api 资源。该列表还显示了每种类型的短名称以及在 JSON/YAML 文件中定义对象所需的一些

UNDERSTANDING LOAD BALANCER SERVICES

While Kubernetes allows you to create so-called LoadBalancer services, it doesn't provide the load balancer itself. If your cluster is deployed in the cloud, Kubernetes can ask the cloud infrastructure to provision a load balancer and configure it to forward traffic into your cluster. The infrastructure tells Kubernetes the IP address of the load balancer and this becomes the external address of your service.

The process of creating the Service object, provisioning the load balancer and how it forwards connections into the cluster is shown in the next figure.

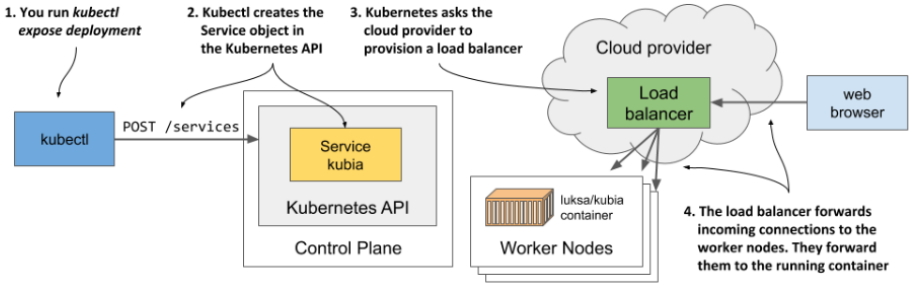


Figure 3.10 How creating a Service object with the type LoadBalancer works

Provisioning of the load balancer takes some time, so let's wait a few more seconds and check again whether the IP address is already assigned. This time, instead of listing all services, you'll only display the kubia service by using its name, as shown in the next listing.

Listing 3.14 Getting a single service

```
$ kubectl get svc kubia
NAME         TYPE           CLUSTER-IP     EXTERNAL-IP     PORT(S)          AGE
kubia        LoadBalancer   10.19.243.17   35.246.179.22   8080:30838/TCP   82s
```

The external IP is now displayed. This means that the load balancer is ready to forward requests to your application for clients around the world.

> **NOTE** If you deployed your cluster with Docker Desktop, the load balancer's IP address is shown as `localhost`, referring to your Windows or macOS machine, not the VM where Kubernetes and the application runs. If you use Minikube to create the cluster, no load balancer is created, but you can access the service in another way. More on this later.

ACCESSING YOUR APPLICATION THROUGH THE LOAD BALANCER

You can now send requests to your application through the external IP and port of the service:

77

了解负载均衡器服务

虽然 Kubernetes 允许您创建所谓的 LoadBalancer 服务，但它本身不提供负载均衡器。如果您的集群部署在云中，Kubernetes 可以要求云基础设施配置负载均衡器并将其配置为将流量转发到您的集群中。基础设施告诉 Kubernetes 负载均衡器的 IP 地址，这将成为您服务的外部地址。

创建服务对象、配置负载均衡器的过程以及如何配置
将连接转发到集群中，如下图所示。

图 3.10 如何创建类型为 LoadBalancer 的 Service 对象

负载均衡器的配置需要一些时间，因此我们再等待几秒钟，然后再次检查 IP 地址是否已分配。这次，您将仅使用其名称来显示 kubia 服务，而不是列出所有服务，如下一个清单所示。

清单 3.14 获取单个服务

```
$ kubectl  get svc kubia  名称 类型 集群-IP 外部-IP 端口 年龄
库比亚 负载均衡器 10.19.243.17  35.246.179.22  8080:30838/TCP  82s
```

现在显示外部 IP。这意味着负载均衡器已准备好将世界各地客户端的请求转发到您的应用程序。

注意如果您使用 Docker Desktop 部署集群，负载均衡器的 IP 地址将显示为 localhost，指的是您的 Windows 或 macOS 计算机，而不是 Kubernetes 和
应用程序运行。如果使用Minikube创建集群，不会创建负载均衡器，但可以访问
以另一种方式提供服务。稍后会详细介绍这一点

通过负载均衡器访问您的应用程序

您现在可以通过服务的外部 IP 和端口向您的应用程序发送请求：

```
$ curl 35.246.179.22:8080
Hey there, this is kubia-9d785b578-p449x. Your IP is ::ffff:1.2.3.4.
```

NOTE If you use Docker Desktop, the service is available at `localhost:8080` from within your host operating system. Use curl or your browser to access it.

Congratulations! If you use Google Kubernetes Engine, you've successfully published your application to users across the globe. Anyone who knows its IP and port can now access it. If you don't count the steps needed to deploy the cluster itself, only two simple commands were needed to deploy your application:

- `kubectl create deployment` and
- `kubectl expose deployment`.

ACCESSING YOUR APPLICATION WHEN A LOAD BALANCER ISN'T AVAILABLE

Not all Kubernetes clusters have mechanisms to provide a load balancer. The cluster provided by Minikube is one of them. If you create a service of type LoadBalancer, the service itself works, but there is no load balancer. Kubectl always shows the external IP as `<pending>` and you must use a different method to access the service.

Several methods of accessing services exist. You can even bypass the service and access individual pods directly, but this is mostly used for troubleshooting. You'll learn how to do this in chapter 5. For now, let's explore the next easier way to access your service if no load balancer is available.

As the next listing shows, Minikube can tell you where to access the service:

Listing 3.15 Getting the service URL when using Minikube

```
$ minikube service kubia --url
http://192.168.99.102:30838
```

The command prints out the URL of the service. You can now point `curl` or your browser to that URL to access your application:

```
$ curl http://192.168.99.102:30838
Hey there, this is kubia-9d785b578-p449x. Your IP is ::ffff:172.17.0.1.
```

TIP If you omit the `--url` option when running the `minikube service` command, your browser opens and loads the service URL.

You may wonder where this IP address and port come from. This is the IP of the Minikube virtual machine. You can confirm this by executing the `minikube ip` command. The Minikube VM is also your single worker node. The port `30838` is the so-called *node port*. It's the port on the worker node that forwards connections to your service. You may have noticed the port in the service's port list when you ran the `kubectl get svc` command:

```
$ kubectl get svc kubia
NAME         TYPE           CLUSTER-IP    EXTERNAL-IP   PORT(S)          AGE
kubia        LoadBalancer   10.19.243.17  <pending>     8080:30838/TCP   82s
```


78

$卷曲35.246.179.22 :8080

注意如果您使用 Docker Desktop，则可以在主机内的 localhost:8080 上使用该服务操作系统。使用curl 或您的浏览器来访问它。

恭喜！如果您使用 Google Kubernetes Engine，则您已成功向全球用户发布了您的应用程序。任何知道其 IP 和端口的人现在都可以访问它。如果不计算部署集群本身所需的步骤，则部署应用程序只需要两个简单的命令：

- kubectl 创建部署并
- kubectl 公开部署

当负载均衡器不可用时访问您的应用程序

并非所有 Kubernetes 集群都有提供负载均衡器的机制。 Minikube提供的集群就是其中之一。如果您创建 LoadBalancer 类型的服务，该服务本身可以工作，但没有负载均衡器。 Kubectl 始终将外部 IP 显示为 ，您必须使用不同的方法来访问该服务。

存在多种访问服务的方法。您甚至可以绕过该服务并直接访问各个 Pod，但这主要用于故障排除。您将在第 5 章中了解如何执行此操作。现在，让我们探索在没有可用负载均衡器的情况下访问服务的下一个更简单的方法。

如下列表所示，Minikube 可以告诉您在哪里访问该服务：

清单 3.15 使用 Minikube 时获取服务 URL

```
$ minikube 服务 kubia --url
http://192.168.99.102 :30838
```

该命令打印出服务的 URL。现在，您可以将 curl 或浏览器指向该 URL 来访问您的应用程序：

```
$curl
http://192.168.99.102 :30838
嘿，这是 kubia-9d785b578-
提示如果您的IP
minikube服务命令时
省略 --url 选项，您的
```

您可能想知道这些打开并加载服务 URL 来自哪里。这是 Minikube 虚拟机的 IP。您可以通过执行 minikube IP 命令来确认这一点。 Minikube VM 也是您的单个工作节点。 30838端口就是所谓的节点端口。它是工作节点上的端口，用于将连接转发到您的服务。当您运行 kubectl get svc 命令时，您可能已经注意到服务端口列表中的端口：

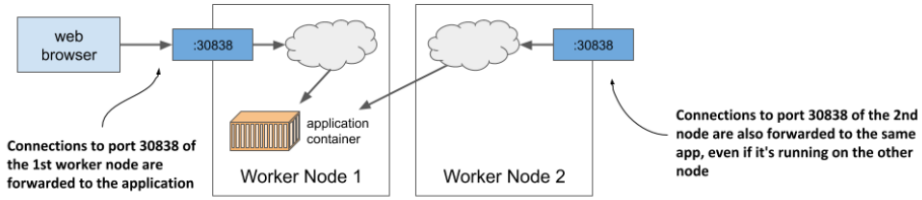
```
$ kubectl get svc kubia 名称类型集群-IP 外部-IP 端口年龄
kubia LoadBalancer 10.19.243.17 〈待定〉8080:30838/TCP 82s
```

Your service is accessible via this port number on all your worker nodes, regardless of whether you're using Minikube or any other Kubernetes cluster.

> **NOTE** If you use Docker Desktop, the VM running Kubernetes can't be reached from your host OS through the VM's IP. You can access the service through the node port only within the VM by logging into it using the special container as described in section 3.1.1.

If you know the IP of at least one of your worker nodes, you should be able to access your service through this `IP:port` combination, provided that firewall rules do not prevent you from accessing the port.

The next figure shows how external clients access the application via the node ports.



Figgure 3.11 Connection routing through a service's node port

To connect this to what I mentioned earlier about the load balancer forwarding connections to the nodes and the nodes then forwarding them to the containers: the node ports are exactly where the load balancer sends incoming requests to. Kubernetes then ensures that they are forwarded to the application running in the container. You'll learn how it does this in chapter 10, as we delve deeper into services. Don't lose too much time thinking about it until then. Instead, let's play a little more with our cluster to see what else Kubernetes can do.

3.3.3 Horizontally scaling the application

You now have a running application that is represented by a Deployment and exposed to the world by a Service object. Now let's create some additional magic.

One of the major benefits of running applications in containers is the ease with which you can scale your application deployments. You're currently running a single instance of your application. Imagine you suddenly see many more users using your application. The single instance can no longer handle the load. You need to run additional instances to distribute the load and provide service to your users. This is known as *scaling out*. With Kubernetes, it's trivial to do.

无论您使用的是 Minikube 还是任何其他 Kubernetes 集群，您的服务都可以通过所有工作节点上的此端口号访问。

> 注意如果您使用 Docker Desktop，则无法通过主机操作系统访问运行 Kubernetes 的虚拟机虚拟机的IP。您只能在虚拟机内通过节点端口访问该服务，方法是使用以下命令登录虚拟机：

如果您知道至少一个工作节点的 IP，则您应该能够通过此 IP:端口组合访问您的服务，前提是防火墙规则不会阻止您访问该端口。

下图显示了外部客户端如何通过节点端口访问应用程序。

图 3.11 通过服务节点端口的连接路由

要将其与我之前提到的有关负载均衡器将连接转发到节点以及节点然后将它们转发到容器的内容联系起来：节点端口正是负载均衡器将传入请求发送到的位置。然后，Kubernetes 确保将它们转发到容器中运行的应用程序。当我们深入研究服务时，您将在第 10 章中了解它是如何做到这一点的。在那之前不要浪费太多时间思考它。相反，让我们多尝试一下我们的集群，看看 Kubernetes 还能做什么。

3.3.3 水平扩展应用程序

您现在拥有一个正在运行的应用程序，该应用程序由 Deployment 表示并通过 Service 对象向外界公开。现在让我们创造一些额外的魔法。

在容器中运行应用程序的主要好处之一是可以轻松扩展应用程序部署。您当前正在运行应用程序的单个实例。想象一下，您突然看到更多的用户使用您的应用程序。单个实例无法再处理负载。您需要运行额外的实例来分配负载并向用户提供服务。这称为横向扩展。使用 Kubernetes，这一切都很简单。

INCREASING THE NUMBER OF RUNNING APPLICATION INSTANCES

To deploy your application, you've created a Deployment object. By default, it runs a single instance of your application. To run additional instances, you only need to scale the Deployment object with the following command:

```
$ kubectl scale deployment kubia --replicas=3
deployment.apps/kubia scaled
```

You've now told Kubernetes that you want to run three exact copies or *replicas* of your pod. Note that you haven't instructed Kubernetes what to do. You haven't told it to add two more pods. You just set the new desired number of replicas and let Kubernetes determine what action it must take to reach the new desired state.

This is one of the most fundamental principles in Kubernetes. Instead of telling Kubernetes what to do, you simply set a new desired state of the system and let Kubernetes achieve it. To do this, it examines the current state, compares it with the desired state, identifies the differences and determines what it must do to reconcile them.

SEEING THE RESULTS OF THE SCALE-OUT

Although it's true that the `kubectl scale deployment` command seems imperative, since it apparently tells Kubernetes to scale your application, what the command actually does is modify the specified Deployment object. As you'll see in a later chapter, you could have simply edited the object instead of giving the imperative command. Let's view the Deployment object again to see how the scale command has affected it:

```
$ kubectl get deploy
NAME    READY   UP-TO-DATE   AVAILABLE   AGE
kubia   3/3     3            3           18m
```

Three instances are now up to date and available and three of three containers are ready. This isn't clear from the command output, but the three containers are not part of the same pod instance. There are three pods with one container each. You can confirm this by listing pods:

```
$ kubectl get pods
NAME                    READY   STATUS    RESTARTS   AGE
kubia-9d785b578-58vhc   1/1     Running   0          17s
kubia-9d785b578-jmnj8   1/1     Running   0          17s
kubia-9d785b578-p449x   1/1     Running   0          18m
```

As you can see, three pods now exist. As indicated in the `READY` column, each has a single container, and all the containers are ready. All the pods are `Running`.

DISPLAYING THE PODS' HOST NODE WHEN LISTING PODS

If you use a single-node cluster, all your pods run on the same node. But in a multi-node cluster, the three pods should be distributed throughout the cluster. To see which nodes the pods were scheduled to, you can use the `-o wide` option to display a more detailed pod list:

```
$ kubectl get pods -o wide
NAME                    ...  IP          NODE
```


增加正在运行的应用程序实例的数量

为了部署您的应用程序，您创建了一个 Deployment 对象。默认情况下，它运行应用程序的单个实例。要运行其他实例，您只需使用以下命令扩展 Deployment 对象：

```
$ kubectl 规模部署 kubia --replicas=3 部
署.apps/kubia 规模化
```

现在，您已经告诉 Kubernetes 您想要运行 pod 的三个精确副本或副本。请注意，您尚未指示 Kubernetes 做什么。您还没有告诉它再添加两个 pod。您只需设置新的所需副本数量，然后让 Kubernetes 确定必须采取什么操作才能达到新的所需状态。

这是 Kubernetes 最基本的原则之一。您无需告诉 Kubernetes 要做什么，只需设置系统的新所需状态并让 Kubernetes 实现它。为此，它会检查当前状态，将其与所需状态进行比较，识别差异并确定必须采取哪些措施来协调它们。

查看横向扩展的结果

尽管 kubectl scale 部署命令看起来确实势在必行，因为它显然告诉 Kubernetes 扩展您的应用程序，但该命令实际上所做的是修改指定的 Deployment 对象。正如您将在后面的章节中看到的，您可以简单地编辑对象，而不是发出命令性命令。让我们再次查看 Deployment 对象，看看缩放命令如何影响它：

```
$ kubectl get 部署名称已准备好，最新可
用年龄
库比亚3/3     3           3 18米
```

三个实例现已更新并可用，三个容器中的三个已准备就绪。从命令输出中尚不清楚这一点，但这三个容器不是同一 Pod 实例的一部分。一共有三个 Pod，每个 Pod 一个容器。您可以通过列出 pod 来确认这一点：

```
$ kubectl get pods NAME READY STATUS RESTARTS AGE

kubia-9d785b578-58vhc 1/1 运行 0 17秒
kubia-9d785b578-jmnj8 1/1 运行 0 17秒
kubia-9d785b578-p449x 1/1 运行 0 18米
```

如您所见，现在存在三个 Pod。如 READY 列所示，每个容器都有一个容器，并且所有容器都已准备就绪。所有 Pod 都在运行。

列出 Pod 时显示 Pod 的主机节点

如果您使用单节点集群，则所有 Pod 都在同一节点上运行。但在多节点集群中，三个 pod 应该分布在整个集群中。要查看 Pod 被调度到哪些节点，可以使用 -o Wide 选项显示更详细的 Pod 列表：

```
$ kubectl get pods -o
宽名称              ... IP 节点
```

```
kubia-9d785b578-58vhc  ...  10.244.1.5  worker1 #A
kubia-9d785b578-jmnj8  ...  10.244.2.4  worker2 #B
kubia-9d785b578-p449x  ...  10.244.2.3  worker2 #B
```

#A Pod scheduled to one node
#B Two pods scheduled to another node

NOTE You can also use the `-o wide` output option to see additional information when listing other object types.

The wide output shows that one pod was scheduled to one node, whereas the other two were both scheduled to a different node. The Scheduler usually distributes pods evenly, but it depends on how it's configured. You'll learn more about scheduling in chapter 21.

Understanding why the worker node a pod is scheduled to is not important

Regardless of the node they run on, all instances of your application have an identical OS environment, because they run in containers created from the same container image. You may remember from the previous chapter that the only thing that might be different is the OS kernel, but this only happens when different nodes use different kernel versions or load different kernel modules.

In addition, each pod gets its own IP and can communicate in the same way with any other pod - it doesn't matter if the other pod is on the same worker node, another node located in the same server rack or even a completely different data center.

So far, you've set no resource requirements for the pods, but if you had, each pod would have been allocated the requested amount of compute resources. It shouldn't matter to the pod which node provides these resources, as long as the pod's requirements are met.

Therefore, you shouldn't care where a pod is scheduled to. It's also why the default `kubectl get pods` command doesn't display information about the worker nodes for the listed pods. In the world of Kubernetes, it's just not that important.

As you can see, scaling an application is incredibly easy. Once your application is in production and there is a need to scale it, you can add additional instances with a single command without having to manually install, configure and run additional copies.

NOTE The app itself must support horizontal scaling. Kubernetes doesn't magically make your app scalable; it merely makes it trivial to replicate it.

OBSERVING REQUESTS HITTING ALL THREE PODS WHEN USING THE SERVICE

Now that multiple instances of your app are running, let's see what happens when you hit the service URL again. Will the response come from the same instance every time? The next listing shows what happens.

Listing 3.16 Requests sent to the service are spread across all the pods

```
$ curl 35.246.179.22:8080
Hey there, this is kubia-9d785b578-58vhc. Your IP is ::ffff:1.2.3.4.    #A
$ curl 35.246.179.22:8080
```



```
kubia-9d785b578-58vhc..  10.244.1.5  工人1#A
kubia-9d785b578-jmnj8..  10.244.2.4  工人2 #B
kubia-9d785b578-p449x..  10.244.2.3  工人2 #B
```

#一个Pod调度到一个节点

注意 在列出其他对象时，您还可以使用 -o Wide 输出选项来查看附加信息类型。

宽输出显示一个 pod 被调度到一个节点，而另外两个 pod 都被调度到不同的节点。 Scheduler 通常会均匀地分配 Pod，但这取决于它的配置方式。您将在第 21 章中了解有关日程安排的更多信息。

了解为什么 Pod 被调度到工作节点并不重要

无论它们在哪个节点上运行，应用程序的所有实例都具有相同的操作系统环境，因为它们在从同一容器映像创建的容器中运行。您可能还记得上一章中唯一可能不同的是操作系统内核，但这仅在不同节点使用不同内核版本或加载不同内核模块时才会发生。

此外，每个 Pod 都有自己的 IP，并且可以以相同的方式与任何其他 Pod 进行通信 - 无论其他 Pod 位于同一工作节点、位于同一服务器机架中的另一个节点，甚至是完全不同的节点，都没有关系。数据中心。

到目前为止，您尚未为 Pod 设置资源要求，但如果设置了，每个 Pod 都会分配到请求的计算资源量。对于 pod 来说，哪个节点提供这些资源并不重要，只要满足 pod 的要求即可。

因此，您不应该关心 pod 被安排在哪里。这也是为什么默认的 kubectl get pods 命令不显示有关列出的 pod 的工作节点的信息。在 Kubernetes 的世界里，这并不那么重要。

正如您所看到的，扩展应用程序非常容易。一旦您的应用程序投入生产并且需要对其进行扩展，您可以使用单个命令添加其他实例，而无需手动安装、配置和运行其他副本。

注意应用程序本身必须支持水平缩放。 Kubernetes 不会神奇地使您的应用程序具有可扩展性；使用服务时，它只会重新配置负载均衡[illegible]请求

现在您的应用程序的多个实例正在运行，让我们看看当您再次点击服务 URL 时会发生什么。每次响应都会来自同一个实例吗？下一个清单显示了会发生什么。

清单 3.16 发送到服务的请求分布在所有 Pod 中

```
$curl 35.246.179.22 :8080 嘿，这是 kubia-9d785b578-58vhc。您的 IP 是::ffff:1.2.3.4。  #A
```

```
Hey there, this is kubia-9d785b578-p449x. Your IP is ::ffff:1.2.3.4.    #B
$ curl 35.246.179.22:8080
Hey there, this is kubia-9d785b578-jmnj8. Your IP is ::ffff:1.2.3.4.    #C
$ curl 35.246.179.22:8080
Hey there, this is kubia-9d785b578-p449x. Your IP is ::ffff:1.2.3.4.    #D
```

#A Request reaches the first pod
#B Request reaches the third pod
#C Request reaches the second pod
#D Request reaches the third pod again

If you look closely at the responses, you'll see that they correspond to the names of the pods. Each request arrives at a different pod in random order. This is what services in Kubernetes do when more than one pod instance is behind them. They act as load balancers in front of the pods. Let's visualize the system using the following figure.

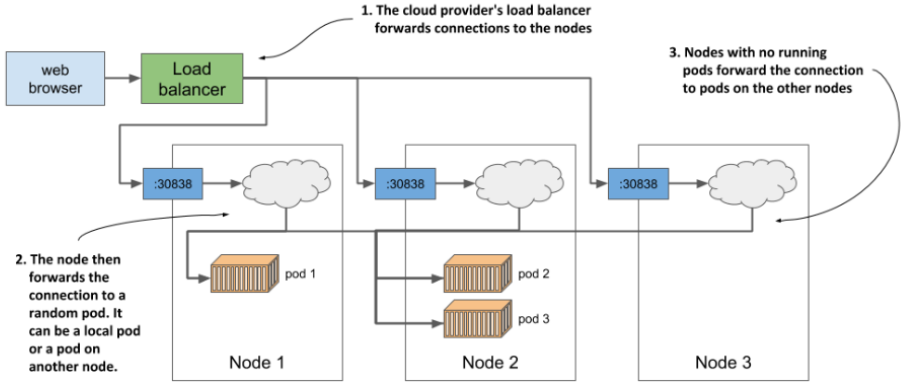


Figure 3.12 Load balancing across multiple pods backing the same service

As the figure shows, you shouldn't confuse this load balancing mechanism, which is provided by the Kubernetes service itself, with the additional load balancer provided by the infrastructure when running in GKE or another cluster running in the cloud. Even if you use Minikube and have no external load balancer, your requests are still distributed across the three pods by the service itself. If you use GKE, there are actually *two* load balancers in play. The figure shows that the load balancer provided by the infrastructure distributes requests across the nodes, and the service then distributes requests across the pods.

I know this may be very confusing right now, but it should all become clear in chapter 10.


```
嘿，这是 kubia-9d785b578-p449x。您的 IP 是::ffff:1.2.3.4。      #B
$卷曲35.246.179.22 :8080
嘿，这是 kubia-9d785b578-jmnj8。您的 IP 是::ffff:1.2.3.4。      #C
$卷曲35.246.179.22 :8080
嘿，这是 kubia-9d785b578-p449x。您的 IP 是::ffff:1.2.3.4。      #D
```

#A 请求到达第一个 pod #B
请求到达第三个 pod #C 请
求到达第二个 pod #D 请求
再次到达第三个 pod

如果您仔细查看响应，您会发现它们与 Pod 的名称相对应。每个请求以随机顺序到达不同的 Pod。这就是 Kubernetes 中的服务在多个 Pod 实例背后时所做的事情。它们充当 Pod 前面的负载平衡器。让我们使用下图来可视化该系统。

图 3.12 支持同一服务的多个 Pod 之间的负载均衡

如图所示，您不应将 Kubernetes 服务本身提供的负载均衡机制与在 GKE 或云中运行的其他集群中运行时基础设施提供的附加负载均衡器混淆。即使您使用 Minikube 并且没有外部负载均衡器，您的请求仍然由服务本身分布在三个 Pod 上。如果您使用 GKE，实际上有两个负载均衡器在起作用。该图显示基础设施提供的负载均衡器在节点之间分发请求，然后服务在 Pod 之间分发请求。

我知道现在这可能很令人困惑，但在第 10 章中一切都会变得清晰。

3.3.4 Understanding the deployed application

To conclude this chapter, let's review what your system consists of. There are two ways to look at your system – the logical and the physical view. You've just seen the physical view in figure 3.12. There are three running containers that are deployed on three worker nodes (a single node when using Minikube). If you run Kubernetes in the cloud, the cloud infrastructure has also created a load balancer for you. Docker Desktop also creates a type of local load balancer. Minikube doesn't create a load balancer, but you can access your service directly through the node port.

While differences in the physical view of the system in different clusters exist, the logical view is always the same, whether you use a small development cluster or a large production cluster with thousands of nodes. If you're not the one who manages the cluster, you don't even need to worry about the physical view of the cluster. If everything works as expected, the logical view is all you need to worry about. Let's take a closer look at this view.

UNDERSTANDING THE API OBJECTS REPRESENTING YOUR APPLICATION

The logical view consists of the objects you've created in the Kubernetes API – either directly or indirectly. The following figure shows how the objects relate to each other.

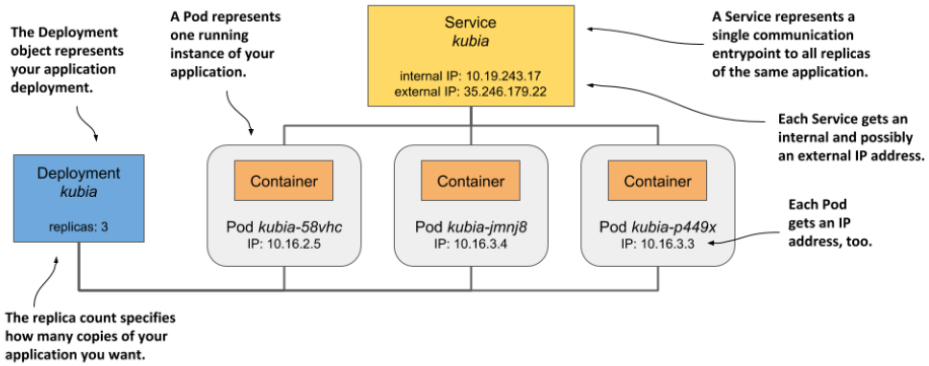


Figure 3.13 Your deployed application consists of a Deployment, several Pods, and a Service.

The objects are as follows:

- the Deployment object you created,
- the Pod objects that were automatically created based on the Deployment, and
- the Service object you created manually.

There are other objects between the three just mentioned, but you don't need to know them yet. You'll learn about them in the following chapters.

83

3.3.4 了解已部署的应用程序

作为本章的总结，让我们回顾一下您的系统的组成部分。有两种查看系统的方法——逻辑视图和物理视图。您刚刚看到了图 3.12 中的物理视图。三个正在运行的容器部署在三个工作节点（使用 Minikube 时为单个节点）上。如果您在云中运行 Kubernetes，那么云

基础设施还为您创建了一个负载均衡器。 Docker Desktop 还创建了一种本地负载均衡器。 Minikube 不会创建负载均衡器，但您可以直接通过节点端口访问您的服务。

虽然不同集群中系统的物理视图存在差异，但无论您使用小型开发集群还是具有数千个节点的大型生产集群，逻辑视图始终相同。如果您不是集群的管理者，您甚至不需要担心集群的物理视图。如果一切都按预期进行，那么您只需担心逻辑视图。让我们仔细看看这个观点。

了解代表您的应用程序的 API 对象

逻辑视图由您在 Kubernetes API 中直接或间接创建的对象组成。下图显示了对象之间的关系。

图 3.13 您部署的应用程序由一个 Deployment、几个 Pod 和一个 Service 组成。

对象如下：

- 您创建的部署对象，
- 基于Deployment自动创建的Pod对象，以及

在刚才提到的三个对象之间还有其他对象，但您还不需要了解它们。您将在接下来的章节中了解它们。

Remember when I explained in chapter 1 that Kubernetes abstracts the infrastructure? The logical view of your application is a great example of this. There are no nodes, no complex network topology, no physical load balancers. Just a simple view that only contains your applications and the supporting objects. Let's look at how these objects fit together and what role they play in your small setup.

The Deployment object represents an application deployment. It specifies which container image contains your application and how many replicas of the application Kubernetes should run. Each replica is represented by a Pod object. The Service object represents a single communication entry point to these replicas.

UNDERSTANDING THE PODS

The essential and most important part of your system are the pods. Each pod definition contains one or more containers that make up the pod. When Kubernetes brings a pod to life, it runs all the containers specified in its definition. As long as a Pod object exists, Kubernetes will do its best to ensure that its containers keep running. It only shuts them down when the Pod object is deleted.

UNDERSTANDING THE ROLE OF THE DEPLOYMENT

When you first created the Deployment object, only a single Pod object was created. But when you increased the desired number of replicas on the Deployment, Kubernetes created additional replicas. Kubernetes ensures that the actual number of pods always matches the desired number.

If one or more pods disappear or their status is unknown, Kubernetes replaces them to bring the actual number of pods back to the desired number of replicas. A pod disappears when someone or something deletes it, whereas a pod's status is unknown when the node it is running on no longer reports its status due to a network or node failure.

Strictly speaking, a Deployment results in nothing more than the creation of a certain number of Pod objects. You may wonder if you can create Pods directly instead of having the Deployment create them for you. You can certainly do this, but if you wanted to run multiple replicas, you'd have to manually create each pod individually and make sure you give each one a unique name. You'd then also have to keep a constant eye on your pods to replace them if they suddenly disappear or the node on which they run fails. And that's exactly why you almost never create pods directly but use a Deployment instead.

UNDERSTANDING WHY YOU NEED A SERVICE

The third component of your system is the Service object. By creating it, you tell Kubernetes that you need a single communication entry point to your pods. The service gives you a single IP address to talk to your pods, regardless of how many replicas are currently deployed. If the service is backed by multiple pods, it acts as a load balancer. But even there is only one pod, you still want to expose it through a service. To understand why, you need to learn an important detail about pods.

Pods are ephemeral. A pod may disappear at any time. This can happen when its host node fails, when someone inadvertently deletes the pod, or when the pod is evicted from an otherwise healthy node to make room for other, more important pods. As explained in the previous section, when pods are created through a Deployment, a missing pod is

还记得我在第 1 章中解释过 Kubernetes 抽象了基础设施吗？应用程序的逻辑视图就是一个很好的例子。没有节点，没有复杂的网络拓扑，没有物理负载平衡器。只是一个简单的视图，仅包含您的应用程序和支持对象。让我们看看这些对象如何组合在一起以及它们在您的小型设置中扮演什么角色。

Deployment 对象代表应用程序部署。它指定哪个容器映像包含您的应用程序以及 Kubernetes 应运行多少个应用程序副本。每个副本都由一个 Pod 对象表示。服务对象代表这些副本的单个通信入口点。

了解 Pod

系统中最重要也是最重要的部分是 Pod。每个 Pod 定义都包含一个或多个构成 Pod 的容器。当 Kubernetes 启动 pod 时，它会运行其定义中指定的所有容器。只要 Pod 对象存在，Kubernetes 就会尽力确保其容器保持运行。仅当 Pod 对象被删除时才会关闭它们。

了解部署的作用

当您第一次创建 Deployment 对象时，仅创建了一个 Pod 对象。但是，当您增加部署上所需的副本数量时，Kubernetes 会创建额外的副本。 Kubernetes 确保 Pod 的实际数量始终与所需数量匹配。

如果一个或多个 Pod 消失或者状态未知，Kubernetes 会替换它们，使实际 Pod 数量恢复到所需的副本数量。当某人或某物删除 Pod 时，Pod 就会消失，而当运行 Pod 的节点由于网络或节点故障而不再报告其状态时，Pod 的状态是未知的。

严格来说，Deployment 只不过是创建一定数量的 Pod 对象。您可能想知道是否可以直接创建 Pod，而不是让 Deployment 为您创建它们。您当然可以这样做，但如果您想运行多个副本，则必须单独手动创建每个 pod，并确保为每个 pod 指定一个唯一的名称。然后，您还必须时刻关注您的 Pod，以便在它们突然消失或它们运行的节点发生故障时进行更换。这正是为什么您几乎从不直接创建 Pod 而是使用 Deployment 的原因。

了解您为何需要服务

系统的第三个组件是服务对象。通过创建它，您可以告诉 Kubernetes 您的 Pod 需要一个通信入口点。该服务为您提供一个 IP 地址来与您的 pod 进行通信，无论当前部署了多少个副本。如果服务由多个 Pod 支持，则它充当负载均衡器。但即使只有一个 pod，您仍然希望通过服务公开它。要了解原因，您需要了解有关 pod 的重要细节。

Pod 是短暂的。 pod 可能随时消失。当其主机节点发生故障、有人无意中删除了 pod 时，或者当 pod 被从原本健康的节点中逐出以便为其他更重要的 pod 腾出空间时，就会发生这种情况。正如上一节中所解释的，当通过以下方式创建 pod 部署，缺少的 pod 是时

immediately replaced with a new one. This new pod is not the same as the one it replaces. It's a completely new pod, with a new IP address.

If you weren't using a service and had configured your clients to connect directly to the IP of the original pod, you would now need to reconfigure all these clients to connect to the IP of the new pod. This is not necessary when using a service. Unlike pods, services aren't ephemeral. When you create a service, it is assigned a static IP address that never changes during lifetime of the service.

Instead of connecting directly to the pod, clients should connect to the IP of the service. This ensures that their connections are always routed to a healthy pod, even if the set of pods behind the service is constantly changing. It also ensures that the load is distributed evenly across all pods should you decide to scale the deployment horizontally.

3.4 Summary

In this hands-on chapter, you've learned:

- Virtually all cloud providers offer a managed Kubernetes option. They take on the burden of maintaining your Kubernetes cluster, while you just use its API to deploy your applications.
- You can also install Kubernetes in the cloud yourself, but this has often proven not to be the best idea until you master all aspects of managing Kubernetes.
- You can install Kubernetes locally, even on your laptop, using tools such as Docker Desktop or Minikube, which run Kubernetes in a Linux VM, or kind, which runs the master and worker nodes as Docker containers and the application containers inside those containers.
- Kubectl, the command-line tool, is the usual way you interact with Kubernetes. A web-based dashboard also exists but is not as stable and up to date as the CLI tool.
- To work faster with kubectl, it is useful to define a short alias for it and enable shell completion.
- An application can be deployed using `kubectl create deployment`. It can then be exposed to clients by running `kubectl expose deployment`. Horizontally scaling the application is trivial: `kubectl scale deployment` instructs Kubernetes to add new replicas or removes existing ones to reach the number of replicas you specify.
- The basic unit of deployment is not a container, but a pod, which can contain one or more related containers.
- Deployments, Services, Pods and Nodes are Kubernetes objects/resources. You can list them with `kubectl get` and inspect them with `kubectl describe`.
- The Deployment object deploys the desired number of Pods, while the Service object makes them accessible under a single, stable IP address.
- Each service provides internal load balancing in the cluster, but if you set the type of service to `LoadBalancer`, Kubernetes will ask the cloud infrastructure it runs in for an additional load balancer to make your application available at a publicly accessible address.

You've now completed your first guided tour around the bay. Now it's time to start learning the ropes, so that you'll be able to sail independently. The next part of the book focuses on

立即更换新的。这个新吊舱与它所取代的吊舱不同。这是一个全新的 Pod，具有新的 IP 地址。

如果您没有使用服务并且已将客户端配置为直接连接到原始 Pod 的 IP，那么您现在需要重新配置所有这些客户端以连接到新 Pod 的 IP。使用服务时不需要这样做。与 Pod 不同，服务不是短暂的。创建服务时，会为其分配一个静态 IP 地址，该地址在服务的生命周期内不会更改。

客户端不应直接连接到 pod，而是应连接到服务的 IP。这可以确保它们的连接始终路由到健康的 Pod，即使服务背后的 Pod 集不断变化也是如此。如果您决定水平扩展部署，它还可以确保负载均匀分布在所有 Pod 上。

3.4 总结

在本实践章节中，您学习了：

- 几乎所有云提供商都提供托管 Kubernetes 选项。他们承担了维护 Kubernetes 集群的负担，而您只需使用其 API 来部署应用程序。
- 您还可以自己在云中安装 Kubernetes，但事实证明这通常不是最好的主意，除非您掌握了管理 Kubernetes 的所有方面。
- 您可以使用 Docker Desktop 或 Minikube 等工具在本地安装 Kubernetes，甚至可以在笔记本电脑上安装，这些工具在 Linux 虚拟机或其他虚拟机中运行 Kubernetes，将主节点和工作节点作为 Docker 容器以及这些容器内的应用程序容器运行。
- Kubectl，命令行工具，是与 Kubernetes 交互的常用方式。基于 Web 的仪表板也存在，但不如 CLI 工具稳定和最新。
- 为了更快地使用kubectl，为其定义一个短别名并启用shell 完成非常有用。
- 可以使用kubectl create deployment 来部署应用程序。然后可以通过运行 kubectl公开部署将其公开给客户端。水平扩展应用程序很简单：kubectl 扩展部署指示 Kubernetes 添加新副本或删除现有副本以达到您指定的副本数量。
- 部署的基本单位不是容器，而是Pod，Pod 中可以包含一个或多个相关容器。
- 部署、服务、Pod 和节点是 Kubernetes 对象/资源。您可以使用 kubectl get 列出它们并使用 kubectl describe 检查它们。
- Deployment 对象部署所需数量的Pod，而Service 对象使它们可以在单个稳定的IP 地址下进行访问。
- 每个服务都在集群中提供内部负载平衡，但如果您将服务类型设置为LoadBalancer，Kubernetes 将要求其运行的云基础设施提供额外的负载平衡器，以使您的应用程序可在可公开访问的地址上使用。

您现在已经完成了第一次海湾导游之旅。现在是时候开始学习诀窍了，这样您就能够独立航行

the different Kubernetes objects and how/when to use them. You'll start with the most important one – the Pod.

86

不同的 Kubernetes 对象以及如何/何时使用它们。您将从最重要的一个开始——Pod。

4

Introducing the Kubernetes API objects

This chapter covers

- **Managing a Kubernetes cluster and the applications it hosts via its API**
- **Understanding the structure of Kubernetes API objects**
- **Retrieving and understanding an object's YAML or JSON manifest**
- **Inspecting the status of cluster nodes via Node objects**
- **Inspecting cluster events through Event objects**

The previous chapter introduced three fundamental objects that make up a deployed application. You created a Deployment object that spawned multiple Pod objects representing individual instances of your application and exposed them to the world by creating a Service object that deployed a load balancer in front of them.

The chapters in the second part of this book explain these and other object types in detail. In this chapter, the common features of Kubernetes objects are presented using the example of Node and Event objects.

4.1 Getting familiar with the Kubernetes API

In a Kubernetes cluster, both users and Kubernetes components interact with the cluster by manipulating objects through the Kubernetes API, as shown in figure 4.1.

These objects represent the configuration of the entire cluster. They include the applications running in the cluster, their configuration, the load balancers through which they are exposed within the cluster or externally, the underlying servers and the storage used by these applications, the security privileges of users and applications, and many other details of the infrastructure.

4

介绍 Kubernetes API 物体

本章涵盖

- 通过 API 管理 Kubernetes 集群及其托管的应用程序
- 了解 Kubernetes API 对象的结构
- 检索和理解对象的 YAML 或 JSON 清单
- 通过 Node 对象检查集群节点的状态
- 通过Event对象检查集群事件

上一章介绍了构成已部署应用程序的三个基本对象。您创建了一个 Deployment 对象，该对象生成多个代表应用程序各个实例的 Pod 对象，并通过创建一个在它们前面部署负载均衡器的 Service 对象将它们公开给外界。

本书第二部分的章节详细解释了这些和其他对象类型。本章以 Node 和 Event 对象为例介绍 Kubernetes 对象的共同特征。

4.1 熟悉 Kubernetes API

在 Kubernetes 集群中，用户和 Kubernetes 组件都通过 Kubernetes API 操作对象来与集群进行交互，如图 4.1 所示。

这些对象代表了整个集群的配置。它们包括集群中运行的应用程序、它们的配置、它们在集群内部或外部公开的负载均衡器、这些应用程序使用的底层服务器和存储、用户和应用程序的安全权限以及许多其他详细信息。基础设施。

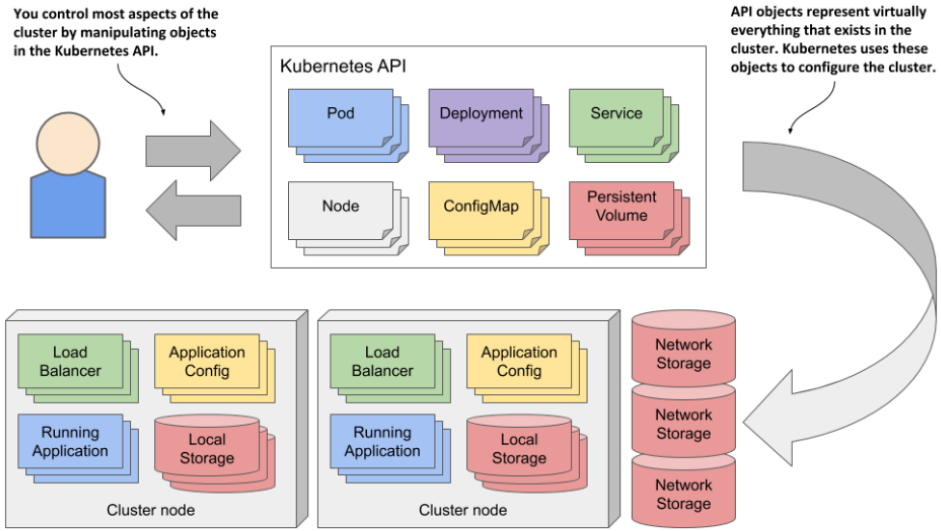


Figure 4.1 A Kubernetes cluster is configured by manipulating objects in the Kubernetes API

4.1.1 Introducing the API

The Kubernetes API is the central point of interaction with the cluster, so much of this book is dedicated to explaining this API. The most important API objects are described in the following chapters, but a basic introduction to the API is presented here.

UNDERSTANDING THE ARCHITECTURAL STYLE OF THE API

The Kubernetes API is an HTTP-based RESTful API where the state is represented by *resources* on which you perform CRUD operations (Create, Read, Update, Delete) using standard HTTP methods such as `POST`, `GET`, `PUT`/`PATCH` or `DELETE`.

> **DEFINITION** REST is Representational State Transfer, an architectural style for implementing interoperability between computer systems via web services using stateless operations, described by Roy Thomas Fielding in his doctoral dissertation. To learn more, read the dissertation at https://www.ics.uci.edu/~fielding/pubs/dissertation/top.htm.

It is these resources (or objects) that represent the configuration of the cluster. Cluster administrators and engineers who deploy applications into the cluster therefore influence the configuration by manipulating these objects.

88

图 4.1 通过操作 Kubernetes API 中的对象来配置 Kubernetes 集群

4.1.1 API介绍

Kubernetes API 是与集群交互的中心点，因此本书的大部分内容都致力于解释这个 API。最重要的 API 对象将在以下章节中描述，但此处仅介绍 API 的基本介绍。

了解 API 的架构风格

Kubernetes API 是基于 HTTP 的 RESTful API，其中状态由您使用标准 HTTP 方法（例如 POST、GET、PUT/PATCH 或 DELETE）执行 CRUD 操作（创建、读取、更新、删除）的资源来表示。

> 定义 REST 是表述性状态传输 (Representational State Transfer)，一种使用无状态操作通过 Web 服务实现计算机系统之间互操作性的架构风格，由 Roy 描述
> 托马斯·菲尔丁在他的博士论文中。要了解更多信息，请阅读论文：

正是这些资源（或对象）代表了集群的配置。因此，将应用程序部署到集群中的集群管理员和工程师通过操作这些对象来影响配置。

In the Kubernetes community, the terms "resource" and "object" are used interchangeably, but there are subtle differences that warrant an explanation.

UNDERSTANDING THE DIFFERENCE BETWEEN RESOURCES AND OBJECTS

The essential concept in RESTful APIs is the resource, and each resource is assigned a URI or Uniform Resource Identifier that uniquely identifies it. For example, in the Kubernetes API, application deployments are represented by deployment resources.

The collection of all deployments in the cluster is a REST resource exposed at `/api/v1/deployments`. When you use the `GET` method to send an HTTP request to this URI, you receive a response that lists all deployment instances in the cluster.

Each individual deployment instance also has its own unique URI through which it can be manipulated. The individual deployment is thus exposed as another REST resource. You can retrieve information about the deployment by sending a `GET` request to the resource URI and you can modify it using a `PUT` request.

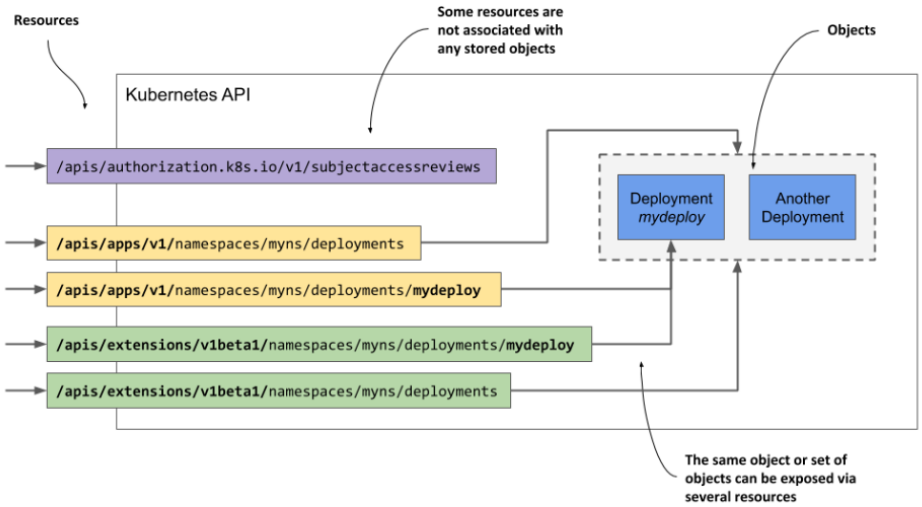


Figure 4.2 A single object can be exposed by two or more resources

An object can therefore be exposed through more than one resource. As shown in figure 4.2, the Deployment object instance named `mydeploy` is returned both as an element of a collection when you query the `deployments` resource and as a single object when you query the individual resource URI directly.

In addition, a single object instance can also be exposed via multiple resources if multiple API versions exist for an object type. Up to Kubernetes version 1.15, two different

在这 库伯内斯 社区， 这 条款 "资源" 和 "目的" 是 用过的可以互换，但有一些细微的差异需要解释。

了解资源和对象之间的差异

RESTful API 中的基本概念是资源，每个资源都分配有一个唯一标识它的 URI 或统一资源标识符。例如，在 Kubernetes API 中，应用程序部署由部署资源表示。

集群中所有部署的集合是在 /api/v1/deployments 中公开的 REST 资源。当您使用 GET 方法向此 URI 发送 HTTP 请求时，

您会收到一条响应，其中列出了集群中的所有部署实例。

每个单独的部署实例还具有其自己唯一的 URI，可以通过该 URI 对其进行操作。因此，单独的部署作为另一个 REST 资源公开。您可以通过向资源 URI 发送 GET 请求来检索有关部署的信息

您可以使用 PUT 请求修改它。

图 4.2 单个对象可以由两个或多个资源公开

因此，一个对象可以通过多个资源公开。如图 4.2 所示，名为 mydeploy 的 Deployment 对象实例在查询部署资源时作为集合元素返回，在直接查询单个资源 URI 时作为单个对象返回。

此外，如果某个对象类型存在多个 API 版本，则还可以通过多个资源公开单个对象实例。直至 Kubernetes 版本 1.15

representations of Deployment objects were exposed by the API. In addition to the `apps/v1` version, exposed at `/apis/apps/v1/deployments`, an older version, `extensions/v1beta1`, exposed at `/apis/extensions/v1beta1/deployments` was available in the API. These two resources didn't represent two different sets of Deployment objects, but a single set that was represented in two different ways - with small differences in the object schema. You could create an instance of a Deployment object via the first URI and then read it back using the second.

In some cases, a resource doesn't represent any object at all. An example of this is the way the Kubernetes API allows clients to verify whether a subject (a person or a service) is authorized to perform an API operation. This is done by submitting a `POST` request to the `/apis/authorization.k8s.io/v1/subjectaccessreviews` resource. The response indicates whether the subject is authorized to perform the operation specified in the request body. The key thing here is that no object is created by the `POST` request.

The examples described above show that a resource isn't the same as an object. If you are familiar with relational database systems, you can compare resources and object types with views and tables. Resources are views through which you interact with objects.

> **NOTE** Because the term "resource" can also refer to compute resources, such as CPU and memory, to reduce confusion, the term "objects" is used in this book to refer to API resources.

UNDERSTANDING HOW OBJECTS ARE REPRESENTED

When you make a `GET` request for a resource, the Kubernetes API server returns the object in structured text form. The default data model is JSON, but you can also tell the server to return YAML instead. When you update the object using a `POST` or `PUT` request, you also specify the new state with either JSON or YAML.

The individual fields in an object's manifest depend on the object type, but the general structure and many fields are shared by all Kubernetes API objects. You'll learn about them next.

4.1.2 Understanding the structure of an object manifest

Before you are confronted with the complete manifest of a Kubernetes object, let me first explain its major parts, because this will help you to find your way through the sometimes hundreds of lines it is composed of.

INTRODUCING THE MAIN PARTS OF AN OBJECT

The manifest of most Kubernetes API objects consists of the following four sections:

- *Type Metadata* contains information about the type of object this manifest describes. It specifies the object type, the group to which the type belongs, and the API version.
- *Object Metadata* holds the basic information about the object instance, including its name, time of creation, owner of the object, and other identifying information. The fields in the Object Metadata are the same for all object types.
- *Spec* is the part in which you specify the desired state of the object. Its fields differ between different object types. For pods, this is the part that specifies the pod's

Deployment 对象的表示由 API 公开。除了在 /apis/apps/v1/deployments 中公开的 apps/v1 版本之外，API 中还提供了在 /apis/extensions/v1beta1/deployments 中公开的旧版本 extensions/v1beta1。这两个资源并不代表两组不同的 Deployment 对象，而是以两种不同方式表示的一组对象 – 对象模式存在细微差别。您可以通过第一个 URI 创建 Deployment 对象的实例，然后使用第二个 URI 读回它。

在某些情况下，资源根本不代表任何对象。一个例子是 Kubernetes API 允许客户端验证主体（个人或服务）是否有权执行 API 操作。这是通过向 /apis/authorization.k8s.io/v1/subjectaccessreviews 资源提交 POST 请求来完成的。响应表明主体是否有权执行请求正文中指定的操作。这里的关键是 POST 请求没有创建任何对象。

上面描述的示例表明资源与对象不同。如果您熟悉关系数据库系统，则可以将资源和对象类型与视图和表进行比较。资源是您与对象交互的视图。

> 注意：由于术语“资源”也可以指代计算资源，例如 CPU 和内存，为了减少混淆，本书中使用术语“对象”来指代 API 资源。

理解对象的表示方式

当您对资源发出 GET 请求时，Kubernetes API 服务器会以结构化文本形式返回对象。默认数据模型是 JSON，但您也可以告诉服务器返回 YAML。当您使用 POST 或 PUT 请求更新对象时，您还可以使用 JSON 或 YAML 指定新状态。

对象清单中的各个字段取决于对象类型，但一般结构和许多字段由所有 Kubernetes API 对象共享。接下来您将了解它们。

4.1.2 理解对象清单的结构

在您面对 Kubernetes 对象的完整清单之前，让我首先解释一下它的主要部分，因为这将帮助您找到它由数百行组成的方法。

介绍物体的主要部分

大多数 Kubernetes API 对象的清单由以下四个部分组成：

- 类型元数据包含有关此清单描述的对象类型的信息。它指定对象类型、该类型所属的组以及 API 版本。
- 对象元数据保存有关对象实例的基本信息，包括其名称、创建时间、对象所有者以及其他标识信息。对象元数据中的字段对于所有对象类型都是相同的。
- Spec 是您指定对象所需状态的部分。不同的对象类型其字段有所不同。对于豆荚

containers, storage volumes and other information related to its operation.

- *Status* contains the current actual state of the object. For a pod, it tells you the condition of the pod, the status of each of its containers, its IP address, the node it's running on, and other information that reveals what's happening to your pod.

A visual representation of an object manifest and its four sections is shown in the next figure.

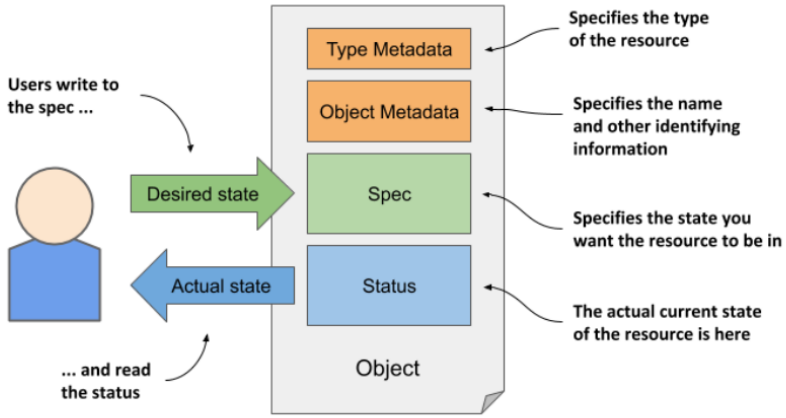


Figure 4.3 The main sections of a Kubernetes API object.

> **NOTE** Although the figure shows that users write to the object's Spec section and read its Status, the API server always returns the entire object when you perform a GET request; to update the object, you also send the entire object in the PUT request.

You'll see an example later to see which fields exist in these sections but let me first explain the Spec and Status sections, as they represent the flesh of the object.

UNDERSTANDING THE SPEC AND STATUS SECTIONS

As you may have noticed in the previous figure, the two most important parts of an object are the Spec and Status sections. You use the Spec to specify the desired state of the object and read the actual state of the object from the Status section. So, you are the one who writes the Spec and reads the Status, but who or what reads the Spec and writes the Status?

The Kubernetes Control Plane runs several components called *controllers* that manage the objects you create. Each controller is usually only responsible for one object type. For example, the *Deployment controller* manages Deployment objects.

91

容器、存储卷以及与其操作相关的其他信息。

- Status 包含对象当前的实际状态。对于 Pod，它会告诉您 Pod 的状况、每个容器的状态、IP 地址、运行的节点以及揭示 Pod 发生情况的其他信息。

下图显示了对象清单及其四个部分的直观表示。

图 4.3 Kubernetes API 对象的主要部分。

> 注意 虽然图中显示用户写入对象的 Spec 部分并读取其 Status，但 API 当您执行 GET 请求时，服务器始终返回整个对象；要更新对象，您还可以发送 PUT 请求中的整个对象

稍后您将看到一个示例，了解这些部分中存在哪些字段，但让我首先解释一下 Spec 和 Status 部分，因为它们代表了对象的实质。

了解规范和状态部分

正如您在上图中可能已经注意到的，对象的两个最重要的部分是 Spec 和 Status 部分。您可以使用 Spec 来指定对象的所需状态，并从 Status 部分读取对象的实际状态。那么，您是编写规范并读取状态的人，但是谁或什么读取规范并编写状态呢？

Kubernetes 控制平面运行多个称为控制器的组件，用于管理您创建的对象。每个控制器通常只负责一种对象类型。例如，部署控制器管理部署对象。

As shown in figure 4.4, the task of a controller is to read the desired object state from the object's Spec section, perform the actions required to achieve this state, and report back the actual state of the object by writing to its Status section.

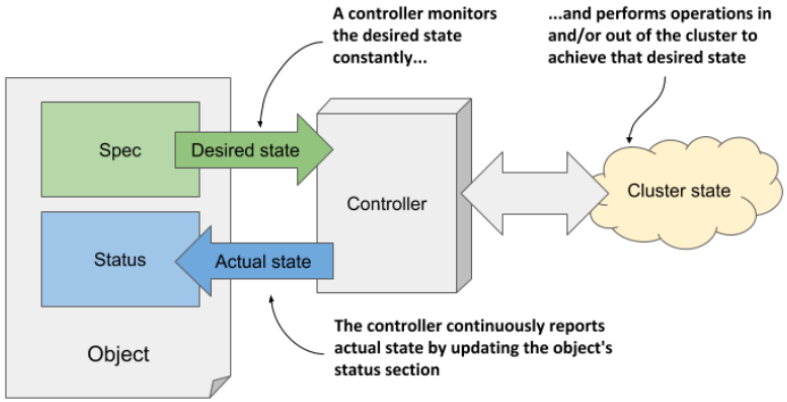


Figure 4.4 How a controller manages an object

Essentially, you tell Kubernetes what it has to do by creating and updating API objects. Kubernetes controllers use the same API objects to tell you what they have done and what the status of their work is.

You'll learn more about the individual controllers and their responsibilities in chapter 13. For now, just remember that a controller is associated with most object types and that the controller is the thing that reads the Spec and writes the Status of the object.

Not all objects have the spec and status sections

All Kubernetes API objects contain the two metadata sections, but not all have the Spec and Status sections. Those that don't, typically contain just static data and don't have a corresponding controller, so it is not necessary to distinguish between the desired and the actual state of the object.

An example of such an object is the Event object, which is created by various controllers to provide additional information about what is happening with an object that the controller is managing. The Event object is explained in section 4.3.

You now understand the general outline of an object, so the next section of this chapter can finally explore the individual fields of an object.

92

如图 4.4 所示，控制器的任务是从对象的 Spec 部分读取所需的对象状态，执行实现此状态所需的操作，并通过写入其 Status 部分来报告对象的实际状态。

图 4.4 控制器如何管理对象

本质上，您通过创建和更新 API 对象来告诉 Kubernetes 它必须做什么。 Kubernetes 控制器使用相同的 API 对象来告诉您他们做了什么以及他们的工作状态是什么。

您将在第 13 章中了解有关各个控制器及其职责的更多信息。现在，只需记住控制器与大多数对象类型相关联，并且控制器是读取 Spec 并写入对象状态的东西。

并非所有对象都有规格和状态部分

所有 Kubernetes API 对象都包含两个元数据部分，但并非所有对象都包含 Spec 和 Status 部分。那些不包含静态数据的对象通常只包含静态数据，并且没有相应的控制器，因此没有必要区分对象的所需状态和实际状态。

此类对象的一个示例是事件对象，它由各种控制器创建，以提供有关控制器正在管理的对象所发生情况的附加信息。 Event 对象在 4.3 节中进行了解释。

现在您已经了解了对象的总体轮廓，因此本章的下一节终于可以探索对象的各个字段了。

4.2 Examining an object's individual properties

To examine Kubernetes API objects up close, we'll need a concrete example. Let's take the Node object, which should be easy to understand because it represents something you might be relatively familiar with - a computer in the cluster.

My Kubernetes cluster provisioned by the kind tool has three nodes - one master and two workers. They are represented by three Node objects in the API. I can query the API and list these objects using `kubectl get nodes`, as shown in the next listing.

Listing 4.1 Listing Node objects

```
$ kubectl get nodes
NAME                 STATUS   ROLES    AGE    VERSION
kind-control-plane   Ready    master   1h     v1.18.2
kind-worker          Ready    <none>   1h     v1.18.2
kind-worker2         Ready    <none>   1h     v1.18.2
```

The following figure shows the three Node objects and the actual cluster machines that make up the cluster. Each Node object instance represents one host. In each instance, the Spec section contains (part of) the configuration of the host, and the Status section contains the state of the host.

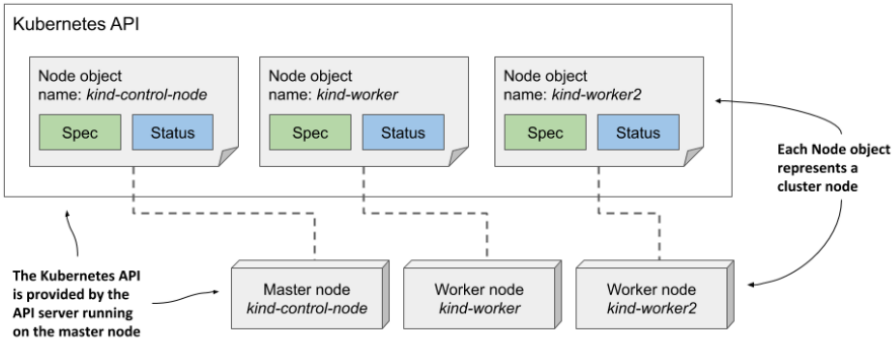


Figure 4.5 Cluster nodes are represented by Node objects

NOTE Node objects are slightly different from other objects because they are usually created by the Kubelet - the node agent running on the cluster node - rather than by users. When you add a machine to the cluster, the Kubelet registers the node by creating a Node object that represents the host. Users can then edit (some of) the fields in the Spec section.

4.2 检查对象的单独属性

为了近距离检查 Kubernetes API 对象，我们需要一个具体的例子。我们以 Node 对象为例，它应该很容易理解，因为它代表了你可能相对熟悉的东西——集群中的一台计算机。

我的 Kubernetes 集群由 kind 工具配置，具有三个节点 - 一个主节点和两个工作节点。它们由 API 中的三个 Node 对象表示。我可以查询 API 并使用 kubectl get 节点列出这些对象，如下一个清单所示。

清单 4.1 列出 Node 对象

```
$ kubectl  get  节点名称 状态 角色 年龄 版本
kind-control-plane  Ready  master  1h  v1.18.2
kind-worker  就绪  <无>  1h  v1.18.2
```

下图显示了组成集群的三个 Node 对象和实际的集群机器。每个 Node 对象实例代表一台主机。在每个实例中，Spec 部分包含主机的（部分）配置，Status 部分包含主机的状态。

图4.5 集群节点由Node对象表示

注意 节点对象与其他对象略有不同，因为它们通常由 Kubelet（在集群节点上运行的节点代理）创建，而不是由用户创建。当您将一台机器添加到集群时，
Kubelet 通过创建代表主机的 Node 对象来注册节点。然后用户可以编辑（某些
of) 规格部分中的字段

4.2.1 Exploring the full manifest of a Node object

Let's take a close look at one of the Node objects. List all Node objects in your cluster by running the `kubectl get nodes` command and select one you want to inspect. Then, execute the `kubectl get node <node-name> -o yaml` command, where you replace `<node-name>` with the name of the node, as shown in the following listing.

Listing 4.2 Displaying the complete YAML manifest of an object

```
$ kubectl get node kind-control-plane -o yaml
apiVersion: v1            #A
kind: Node                #A
metadata:                                                  #B
  annotations: ...                                         #B
  creationTimestamp: "2020-05-03T15:09:17Z"                #B
  labels: ...                                              #B
  managedFields: ...                                       #B
  name: kind-control-plane                           #C    #B
  resourceVersion: "3220054"                               #B
  selfLink: /api/v1/nodes/kind-control-plane               #B
  uid: 16dc1e0b-8d34-4cfb-8ade-3b0e91ec838b                #B
spec:                                                      #D
  podCIDR: 10.244.0.0/24                             #E    #D
  podCIDRs:                                          #E    #D
  - 10.244.0.0/24                                    #E    #D
  taints:                                                  #D
  - effect: NoSchedule                                     #D
    key: node-role.kubernetes.io/master                    #D
status:                                                    #F
  addresses:                                         #G    #F
  - address: 172.18.0.2                              #G    #F
    type: InternalIP                                 #G    #F
  - address: kind-control-plane                      #G    #F
    type: Hostname                                   #G    #F
  allocatable: ...                                         #F
  capacity:                                          #H    #F
    cpu: "8"                                         #H    #F
    ephemeral-storage: 401520944Ki                   #H    #F
    hugepages-1Gi: "0"                               #H    #F
    hugepages-2Mi: "0"                               #H    #F
    memory: 32720824Ki                               #H    #F
    pods: "110"                                      #H    #F
  conditions:                                              #F
  - lastHeartbeatTime: "2020-05-17T12:28:41Z"              #F
    lastTransitionTime: "2020-05-03T15:09:17Z"             #F
    message: kubelet has sufficient memory available       #F
    reason: KubeletHasSufficientMemory                     #F
    status: "False"                                        #F
    type: MemoryPressure                                   #F
    ...                                                    #F
  daemonEndpoints:                                         #F
    kubeletEndpoint:                                       #F
      Port: 10250                                          #F
  images:                                            #I    #F
  - names:                                           #I    #F
    - k8s.gcr.io/etcd:3.4.3-0                        #I    #F
    sizeBytes: 289997247                             #I    #F
```


94

4.2.1 探索 Node 对象的完整清单

让我们仔细看看其中一个 Node 对象。通过运行 kubectl 列出集群中的所有 Node 对象得到 节点命令并选择您要检查的节点。然后，
执行 kubectl get node -o yaml 命令，将 替换为节点的名称，如以下清单所示。

清单 4.2 显示对象的完整 YAML 清单

```
$ kubectl 获取节点 kind-control-plane -o
yaml apiVersion：v1     #A
种类：节点              #A
元数据：                                                   #B
  注释：...                                                #B
  创建时间戳：“2020-05-03T15:09:17Z”                       #B
  标签：...                                                #B
  管理字段：...                                            #B
  名称：种类控制平面                                 #C    #B
  资源版本：“3220054”                                      #B
  自链接：/api/v1/nodes/kind-control-plane                 #B
  uid：16dc1e0b-8d34-4cfb-8ade-3b0e91ec838b                #B
规格：                                                     #D
  podCIDR：10.244.0.0/24                             #E    #D
  podCIDR：                                          #E    #D
  - 10.244.0.0/24                                    #E    #D
  污点：                                                   #D
  - 效果：无时间表                                         #D
    密钥：node-role.kubernetes.io/master                   #D
地位：                                                     #F
  地址：                                             #G    #F
  - 地址：172.18.0.2                                 #G    #F
    类型：内部IP                                     #G    #F
  - 地址：种类控制平面                               #G    #F
    类型：主机名                                     #G    #F
  可分配：...                                              #F
  容量：                                             #H    #F
    中央处理器：“8”                                  #H    #F
    临时存储：401520944Ki                            #H    #F
    巨大的页面-1Gi：“0”                              #H    #F
    巨大的页面-2Mi：“0”                              #H    #F
    内存：32720824Ki                                 #H    #F
    豆荚：“110”                                      #H    #F
  状况：                                                   #F
  - 最后心跳时间：“2020-05-17T12:28:41Z”                   #F
    最后转换时间：“2020-05-03T15:09:17Z”                   #F
    消息：kubelet 有足够的可用内存                         #F
    原因：Kubelet有足够的内存                              #F
    状态：“假”                                             #F
    类型：内存压力                                         #F
    ...                                                    #F
  守护进程端点：                                           #F
    kubelet端点：                                          #F
      端口：10250                                          #F
  图片：                                             #我   #F
  - 姓名：                                           #我   #F
    - k8s.gcr.io/etcd:3.4.3-0                        #我   #F
    大小字节：289997247                              #我   #F
```

```
  ...                                                    #I      #F
nodeInfo:                                                #J      #F
  architecture: amd64                                    #J      #F
  bootID: 233a359f-5897-4860-863d-06546130e1ff           #J      #F
  containerRuntimeVersion: containerd://1.3.3-14-g449e9269 #J    #F
  kernelVersion: 5.5.10-200.fc31.x86_64                  #J      #F
  kubeProxyVersion: v1.18.2                              #J      #F
  kubeletVersion: v1.18.2                                #J      #F
  machineID: 74b74e389bb246e99abdf731d145142d            #J      #F
  operatingSystem: linux                                 #J      #F
  osImage: Ubuntu 19.10                                  #J      #F
  systemUUID: 8749f818-8269-4a02-bdc2-84bf5fa21700       #J      #F
```

#A The Type Metadata specifies the type of object and the API version of this object manifest.
#B The Object Metadata section
#C The object name (the node's name)
#D The node's desired state is specified here
#E The IP range reserved for the pods on this node
#F The node's actual state is shown here
#G The IP(s) and hostname of the node
#H The nodes capacity (the amount of compute resources it has)
#I The list of cached container images on this node
#J Information about the node's operating system and the Kubernetes components running on it

NOTE Use the `-o json` option to display the object in JSON instead of YAML.

In the listing, the four main sections of the object definition and the more important properties of the node are annotated to help you distinguish between the more and less important fields. Some lines have been omitted to reduce the length of the manifest.

Accessing the API directly

You may be interested in trying to access the API directly instead of through kubectl. As explained earlier, the Kubernetes API is web based, so you can use a web browser or the curl command to perform API operations, but the API server uses TLS and you typically need a client certificate or token for authentication. Fortunately, kubectl provides a special proxy that takes care of this, allowing you to talk to the API through the proxy using plain HTTP.

To run the proxy, execute the command:

```
$ kubectl proxy
Starting to serve on 127.0.0.1:8001
```

You can now access the API using HTTP at 127.0.0.1:8001. For example, to retrieve the node object, open the URL http://127.0.0.1:8001/api/v1/nodes/kind-control-plane (replace kind-control-plane with one of your nodes' names).

Now let's take a closer look at the fields in each of the four main sections.

The Type Metadata fields

As you can see, the listing starts with the `apiVersion` and `kind` fields, which specify the API version and type of the object that this object manifest specifies. The API version is the schema used to describe this object. As mentioned before, an object type can be associated


```
...                                                    #我      #F
节点信息：                                               #J      #F
  架构：amd64                                          #J      #F
  启动ID：233a359f-5897-4860-863d-06546130e1ff         #J      #F
  容器运行时版本：containerd://1.3.3-14-g449e9269      #J      #F
  内核版本：5.5.10-200.fc31.x86_64                     #J      #F
  kubeProxy 版本：v1.18.2                              #J      #F
  kubelet 版本：v1.18.2                                #J      #F
  机器ID：74b74e389bb246e99abdf731d145142d             #J      #F
  操作系统：Linux                                      #J      #F
  osImage：Ubuntu 19.10                                #J      #F
  系统UUID：8749f818-8269-4a02-bdc2-84bf5fa21700       #J      #F
```

#A 类型元数据指定对象的类型以及该对象清单的 API 版本。 #B 对象元数据部分 #C 对象名称（节点名称） #D 此处指定节点的所需状态 #E 为该节点上的 pod 保留的 IP 范围 #F 此处显示节点的实际状态 #G IP (s) 和节点的主机名

#H 节点容量（拥有的计算资源量） #I 该节点上缓存的容器镜像列表

#J 有关节点操作系统及其上运行的 Kubernetes 组件的信息

注意 使用 -o json 选项以 JSON 而不是 YAML 形式显示对象。

在列表中，对象定义的四个主要部分和节点的更重要属性都进行了注释，以帮助您区分较重要和不太重要的字段。为了减少清单的长度，省略了一些行。

直接访问API

您可能有兴趣尝试直接访问 API，而不是通过 kubectl。如前所述，Kubernetes API 是基于 Web 的，因此您可以使用 Web 浏览器或curl 命令来执行 API 操作，但 API 服务器使用 TLS，您通常需要客户端证书或令牌进行身份验证。幸运的是，kubectl 提供了一个特殊的代理来处理这个问题，允许您使用纯 HTTP 通过代理与 API 进行通信。要运行代理，请执行以下命令：

```
$ kubectl 代理
开始在 127.0.0.1:8001 上提供服
```

您现在可以使用 HTTP（地址为 127.0.0.1:8001）访问 API。例如，要检索节点对象，请打开 URL http://127.0.0.1:8001/api/v1/nodes/kind-control-plane（将 kind-control-plane 替换为您的节点名称之一）。

现在让我们仔细看看四个主要部分中每个部分的字段。

类型元数据字段

如您所见，列表以 apiVersion 和 kind 字段开头，它们指定此对象清单指定的对象的 API 版本和类型。 API 版本是用于描述该对象的架构。正如之前所提

with more than one schema, with different fields in each schema being used to describe the object. However, usually only one schema exists for each type.

The `apiVersion` in the previous listing is simply `v1`, but you'll see in the following chapters that the `apiVersion` in other object types contains more than just the version number. For Deployment objects, for example, the `apiVersion` is `apps/v1`. Whereas the field was originally used only to specify the API version, it is now also used to specify the API group to which the resource belongs. Node objects belong to the core API group, which is conventionally omitted from the `apiVersion` field.

The type of object defined in the manifest is specified by the field `kind`. The object kind in the previous listing is Node, and so far in this book you've also dealt with the following kinds: Deployment, Service, and Pod.

FIELDS IN THE OBJECT METADATA SECTION

The `metadata` section contains the metadata of this object instance. It contains the `name` of the instance, along with additional attributes such as `labels` and `annotations`, which are explained in chapter 9, and fields such as `resourceVersion`, `managedFields`, and other low-level fields, which are explained at depth in chapter 12.

FIELDS IN THE SPEC SECTION

Next comes the `spec` section, which is specific to each object kind. It is relatively short for Node objects compared to what you find for other object kinds. The `podCIDR` fields specify the pod IP range assigned to the node. Pods running on this node are assigned IPs from this range. The `taints` field is not important at this point, but you'll learn about it in chapter 18.

Typically, an object's spec section contains many more fields that you use to configure the object.

FIELDS IN THE STATUS SECTION

The `status` section also differs between the different kinds of object, but its purpose is always the same - it contains the last observed state of the thing the object represents. For Node objects, the status reveals the node's IP address(es), host name, capacity to provide compute resources, the current conditions of the node, the container images it has already downloaded and which are now cached locally, and information about its operating system and the version of Kubernetes components running on it.

4.2.2 Understanding individual object fields

To learn more about individual fields in the manifest, you can refer to the API reference documentation at http://kubernetes.io/docs/reference/ or use the `kubectl explain` command as described next.

USING KUBECTL EXPLAIN TO EXPLORE API OBJECT FIELDS

The kubectl tool has a nice feature that allows you to look up the explanation of each field for each object type (kind) from the command line. Usually, you start by asking it to provide the basic description of the object kind by running `kubectl explain <kind>`, as shown here:

具有多个模式，每个模式中的不同字段用于描述对象。然而，通常每种类型只存在一个模式。

上一个清单中的 apiVersion 只是 v1，但您将在下面看到
其他对象类型中的 apiVersion 不仅仅包含版本号。例如，对于 Deployment 对象，apiVersion 是 apps/v1。该字段最初仅用于指定 API 版本，现在也用于指定资源所属的 API 组。节点对象属于核心 API 组，通常从 apiVersion 字段中省略。

清单中定义的对象类型由字段 kind 指定。前面清单中的对象类型是 Node，到目前为止，在本书中您还处理了以下类型：Deployment、Service 和 Pod。

对象元数据部分中的字段

元数据部分包含该对象实例的元数据。它包含实例的名称，以及诸如标签和注释之类的附加属性，这些属性将在第 9 章中进行解释，以及诸如资源版本、托管字段和其他低级字段之类的字段，这些字段将在第 12 章中进行深入解释。

规范部分中的字段

接下来是规格部分，它特定于每种对象类型。与其他对象类型相比，Node 对象的长度相对较短。podCIDR 字段指定分配给节点的 Pod IP 范围。在此节点上运行的 Pod 会被分配此范围内的 IP。此时，污点字段并不重要，但您将在第 18 章中了解它。

通常，对象的规范部分包含更多用于配置的字段
物体。

状态部分中的字段

不同类型的对象之间的状态部分也有所不同，但其目的始终相同 - 它包含对象所代表的事物的最后观察到的状态。对于 Node 对象，状态显示节点的 IP 地址、主机名、提供计算资源的能力、节点的当前状况、已下载且现在缓存在本地的容器映像以及有关其运行的信息系统及其上运行的 Kubernetes 组件的版本。

4.2.2 了解各个对象字段

要了解有关清单中各个字段的更多信息，您可以参阅 http://kubernetes.io/docs/reference/ 上的 API 参考文档或使用 kubectlexplain

命令如下所述。
使用 KUBECTL EXPLAIN 探索 API 对象字段

kubectl 工具有一个很好的功能，允许您从命令行查找每种对象类型（种类）的每个字段的解释。通常，您首先通过运行 kubectlexplain 要求它提供对象类型的基本描述

Listing 4.3 Using kubectl explain to learn about an object kind

```
$ kubectl explain nodes
KIND:     Node
VERSION:  v1

DESCRIPTION:
     Node is a worker node in Kubernetes. Each node will have a unique
     identifier in the cache (i.e. in etcd).

FIELDS:
   apiVersion   <string>
     APIVersion defines the versioned schema of this representation of an
     object. Servers should convert recognized schemas to the latest...

   kind <string>
     Kind is a string value representing the REST resource this object
     represents. Servers may infer this from the endpoint the client...

   metadata     <Object>
     Standard object's metadata. More info: ...

   spec <Object>
     Spec defines the behavior of a node...

   status       <Object>
     Most recently observed status of the node. Populated by the system.
     Read-only. More info: ...
```

The command prints the explanation of the object and lists the top-level fields that the object can contain.

DRILLING DEEPER INTO AN API OBJECT'S STRUCTURE

You can then drill deeper to find subfields under each specific field. For example, you can use the following command to explain the node's `spec` field:

Listing 4.4 Using kubectl explain to learn about a specific object field and sub-fields

```
$ kubectl explain node.spec
KIND:     Node
VERSION:  v1

RESOURCE: spec <Object>

DESCRIPTION:
     Spec defines the behavior of a node.

     NodeSpec describes the attributes that a node is created with.

FIELDS:
   configSource <Object>
     If specified, the source to get node configuration from The
     DynamicKubeletConfig feature gate must be enabled for the Kubelet...

   externalID   <string>
     Deprecated. Not all kubelets will set this field...
```


清单 4.3 使用 kubectlexplain 来了解对象类型

```
$ kubectl  解释节点种
类：       节点
版本：     v1

描述：
     Node是Kubernetes 中的工作节点。每个节点在缓存中（即在 etcd
     中）都有一个唯一的标识符。

领域：
   api版本      <字符串>
     APIVersion 定义了对象表示的版本化架构。服务器应该将识别的模
     式转换为最新的...

   种类<字符串>
     Kind 是一个字符串值，表示该对象所代表的 REST 资源。服务器
     可以从客户端的端点推断出这一点.....

   元数据       <对象>
```

```
  podCIDR      <string>
    PodCIDR represents the pod IP range assigned to the node.

  ...
```

Please note the API version given at the top. As explained earlier, multiple versions of the same kind can exist. Different versions can have different fields or default values. If you want to display a different version, specify it with the `--api-version` option.

NOTE If you want to see the complete structure of an object (the complete hierarchical list of fields without the descriptions), try `kubectl explain pods --recursive`.

4.2.3 Understanding an object's status conditions

The set of fields in both the `spec` and `status` sections is different for each object kind, but the `conditions` field is found in many of them. It gives a list of conditions the object is currently in. They are very useful when you need to troubleshoot an object, so let's examine them more closely. Since the Node object is used as an example, this section also teaches you how to easily identify problems with a cluster node.

INTRODUCING THE NODE'S STATUS CONDITIONS

Let's print out the YAML manifest of the one of the node objects again, but this time we'll only focus on the `conditions` field in the object's `status`, as shown in the following listing.

Listing 4.5 The current status conditions in a Node object

```
$ kubectl get node kind-control-plane -o yaml
...
status:
  ...
  conditions:
  - lastHeartbeatTime: "2020-05-17T13:03:42Z"
    lastTransitionTime: "2020-05-03T15:09:17Z"
    message: kubelet has sufficient memory available
    reason: KubeletHasSufficientMemory
    status: "False"                                    #A
    type: MemoryPressure                               #A
  - lastHeartbeatTime: "2020-05-17T13:03:42Z"
    lastTransitionTime: "2020-05-03T15:09:17Z"
    message: kubelet has no disk pressure
    reason: KubeletHasNoDiskPressure
    status: "False"                                    #B
    type: DiskPressure                                 #B
  - lastHeartbeatTime: "2020-05-17T13:03:42Z"
    lastTransitionTime: "2020-05-03T15:09:17Z"
    message: kubelet has sufficient PID available
    reason: KubeletHasSufficientPID
    status: "False"                                    #C
    type: PIDPressure                                  #C
  - lastHeartbeatTime: "2020-05-17T13:03:42Z"
    lastTransitionTime: "2020-05-03T15:10:15Z"
    message: kubelet is posting ready status
```



```
podCIDR      <字符串>
  PodCIDR 表示分配给节点的 Pod IP 范围。

...
```

请注意顶部给出的 API 版本。如前所述，同一类型可以存在多个版本。不同的版本可以有不同的字段或默认值。如果要显示不同的版本，请使用 --api-version 选项指定。

注意如果您想查看对象的完整结构（没有描述的完整字段层次列表），请尝试 kubectlexplain pods --recursive。

4.2.3 了解对象的状态条件

对于每种对象类型，规范和状态部分中的字段集都不同，但是

其中许多都包含条件字段。它提供了对象当前所处条件的列表。当您需要对对象进行故障排除时，它们非常有用，因此让我们更仔细地检查它们。由于以 Node 对象为例，本节还教您如何轻松识别集群节点的问题。

节点状态介绍

让我们再次打印其中一个节点对象的 YAML 清单，但这一次我们只关注对象状态中的条件字段，如以下清单所示。

清单 4.5 Node 对象中的当前状态条件

```
$ kubectl 获取节点种类控制平面 -o yaml

... 地位：
  ...
  条
  件lastTransitionTime ：“2020-05-03T15：09：17Z”消
    息：kubelet 有足够的可用内存原因：
    KubeletHasSufficientMemory  状态：“False”#A

  类型：MemoryPressure #A
- 最后心跳时间：“2020-05-17T13:03:42Z”
  lastTransitionTime ：“2020-05-03T15：09：17Z”消
  息：kubelet 没有磁盘压力原因：
  KubeletHasNoDiskPressure 状态：“False”#B

  类型：DiskPressure #B
- 最后心跳时间：“2020-05-17T13:03:42Z”
  lastTransitionTime ：“2020-05-03T15：09：17Z”消
  息：kubelet 有足够的 PID 可用原因：
  KubeletHasSufficientPID  状态：“False”#C

  类型：PID压力#C
- 最后心跳时间：“2020-05-17T13:03:42Z”
  lastTransitionTime ：“2020-05-03T15：
  10：15Z”消息：kubelet 正在发布就绪状
  态
```

```
    reason: KubeletReady
    status: "True"                          #D
    type: Ready                             #D
```

#A Node is not running out of memory
#B Node is not running out of disk space
#C Node has not run out of unused process Ids
#D Node is ready

TIP The `jq` tool is very handy if you want to see only a part of the object's structure. For example, to display the node's status conditions, you can run `kubectl get node <name> -o json | jq .status.conditions`. The equivalent tool for YAML is `yq`.

There are four conditions that reveal the state of the node. Each condition has a `type` and a `status` field, which can be `True`, `False` or `Unknown`, as shown in the figure 4.6. A condition can also specify a machine-facing `reason` for the last transition of the condition and a human-facing `message` with details about the transition. The `lastTransitionTime` field indicates when the condition moved from one status to another, whereas the `lastHeartbeatTime` field reveals the last time the controller received an update on the given condition.

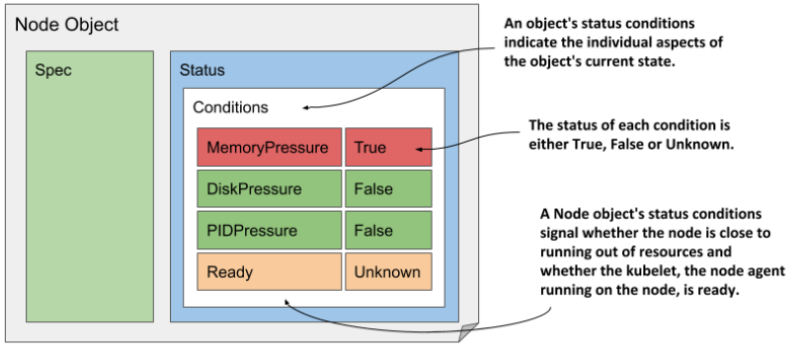


Figure 4.6 The status conditions indicating the state of a Node object

Although it's the last condition in the list, the `Ready` condition is probably the most important, as it signals whether the node is ready to accept new workloads (pods). The other conditions (`MemoryPressure`, `DiskPressure` and `PIDPressure`) signal whether the node is running out of resources. Remember to check these conditions if a node starts to behave strangely - for example, if the applications running on it start running out of resources and/or crash.

99

```
原因：KubeletReady
状态：“True”                            #D
类型：准备就绪                          #D
```

#A 节点未耗尽内存 #B 节点未耗尽磁盘空间 #C 节点未耗尽未使用的进程 ID #D 节点已准备就绪

提示如果您只想查看对象结构的一部分，jq 工具非常方便。例如，要显示节点的状态条件，可以运行kubectl 得到 节点 〈名字〉 -o json | 杰克 .状态.条件。 YAML 的等效工具是 yq。

有四个条件可以揭示节点的状态。每个条件都有一个类型和一个状态字段，可以是 True、False 或 Unknown，如图 4.6 所示。一个条件

还可以指定最后一次条件转换的面向机器的原因以及包含转换详细信息的面向人的消息。LastTransitionTime 字段指示条件从一种状态转变为另一种状态的时间，而

lastHeartbeatTime 字段显示控制器上次收到给定更新的时间

健康）状况。

图 4.6 指示 Node 对象状态的状态条件

尽管这是列表中的最后一个条件，但“就绪”条件可能是最重要的，因为它表示节点是否已准备好接受新的工作负载 (pod)。其他条件（MemoryPressure、DiskPressure 和 PIDPressure）指示节点是否耗尽资源。如果节点开始表现异常，请记住检查这些条件 - 例如，如果在其上运行的应用程序开始耗尽资源和/或崩溃。

UNDERSTANDING CONDITIONS IN OTHER OBJECT KINDS

A condition list such as that in Node objects is also used in many other object kinds. The conditions explained earlier are a good example of why the state of most objects is represented by multiple conditions instead of a single field.

> **NOTE** Conditions are usually orthogonal, meaning that they represent unrelated aspects of the object.

If the state of an object were represented as a single field, it would be very difficult to subsequently extend it with new values, as this would require updating all clients that monitor the state of the object and perform actions based on it. Some object kinds originally used such a single field, and some still do, but most now use a list of conditions instead.

Since the focus of this chapter is to introduce the common features of the Kubernetes API objects, we've focused only on the `conditions` field, but it is far from being the only field in the status of the Node object. To explore the others, use the `kubectl explain` command as described in the previous sidebar. The fields that are not immediately easy for you to understand should become clear to you after reading the remaining chapters in this part of the book.

> **NOTE** As an exercise, use the command `kubectl get <kind> <name> -o yaml` to explore the other objects you've created so far (deployments, services, and pods).

4.2.4 Inspecting objects using the kubectl describe command

To give you a correct impression of the entire structure of the Kubernetes API objects, it was necessary to show you the complete YAML manifest of an object. While I personally often use this method to inspect an object, a more user-friendly way to inspect an object is the `kubectl describe` command, which typically displays the same information or sometimes even more.

UNDERSTANDING THE KUBECTL DESCRIBE OUTPUT FOR A NODE OBJECT

Let's try running the `kubectl describe` command on a Node object. To keep things interesting, let's now take one of the worker nodes instead of the master. The following listing shows what the `kubectl describe` command displays for one of my two worker nodes.

Listing 4.6 Inspecting a Node object with kubectl describe

```
$ kubectl describe node kind-worker-2
Name:               kind-worker2
Roles:              <none>
Labels:             beta.kubernetes.io/arch=amd64
                    beta.kubernetes.io/os=linux
                    kubernetes.io/arch=amd64
                    kubernetes.io/hostname=kind-worker2
                    kubernetes.io/os=linux
Annotations:        kubeadm.alpha.kubernetes.io/cri-socket: /run/contain...
                    node.alpha.kubernetes.io/ttl: 0
```


了解其他物体种类的条件

诸如 Node 对象中的条件列表也可用于许多其他对象类型。前面解释的条件是为什么大多数对象的状态由多个条件而不是单个字段表示的一个很好的例子。

注：条件通常是正交的，这意味着它们代表对象的不相关方面。

如果对象的状态表示为单个字段，则随后用新值扩展它会非常困难，因为这将需要更新监视对象状态并基于该状态执行操作的所有客户端。某些对象类型最初使用这样的单个字段，有些对象仍然这样做，但现在大多数使用条件列表。

由于本章的重点是介绍 Kubernetes API 对象的共同特征，因此我们只关注条件字段，但它远不是唯一的字段。

Node 对象的状态。要探索其他命令，请使用 kubectlexplain 命令，如上边栏中所述。在阅读了本书这一部分的其余章节后，对于您来说不太容易理解的领域应该会变得清晰。

注意作为练习，使用命令 kubectl get -o yaml 来探索您迄今为止创建的其他对象（部署、服务和 Pod）。

4.2.4 使用 kubectl describe 命令检查对象

为了让您对 Kubernetes API 对象的整个结构有一个正确的印象，有必要向您展示对象的完整 YAML 清单。虽然我个人经常使用这种方法来检查对象，但检查对象的一种更用户友好的方法是 kubectl describe 命令，它通常显示相同的信息，或者有时显示相同的信息更。

了解 KUBECTL 描述节点对象的输出

让我们尝试在 Node 对象上运行 kubectl describe 命令。为了保留东西有趣的是，现在让我们采用一个工作节点而不是主节点。以下列表显示了 kubectldescribe 命令为我的两个工作节点之一显示的内容。

清单 4.6 使用 kubectl describe 检查 Node 对象

```
$ kubectl 描述节点 kind-worker-2 名称：kind-worker2
角色：<无>
标签： beta.kubernetes.io/arch=amd64 beta.kubernetes.io/os=linux
kubernetes.io/arch=amd64 kubernetes.io/hostname=kind-worker2
kubernetes.io/os=linux
注释： kubeadm.alpha.kubernetes.io/cri-socket：
/run/contain...node.alpha.kubernetes
```

```
                    volumes.kubernetes.io/controller-managed-attach-deta...
CreationTimestamp:  Sun, 03 May 2020 17:09:48 +0200
Taints:             <none>
Unschedulable:      false
Lease:
  HolderIdentity:  kind-worker2
  AcquireTime:     <unset>
  RenewTime:       Sun, 17 May 2020 16:15:03 +0200
Conditions:
  Type             Status  ...  Reason                       Message
  ----             ------  ---  ------                       -------
  MemoryPressure   False   ...  KubeletHasSufficientMemory   ...
  DiskPressure     False   ...  KubeletHasNoDiskPressure     ...
  PIDPressure      False   ...  KubeletHasSufficientPID      ...
  Ready            True    ...  KubeletReady                 ...
Addresses:
  InternalIP:  172.18.0.4
  Hostname:    kind-worker2
Capacity:
  cpu:                8
  ephemeral-storage:  401520944Ki
  hugepages-1Gi:      0
  hugepages-2Mi:      0
  memory:             32720824Ki
  pods:               110
Allocatable:
  ...
System Info:
  ...
PodCIDR:                      10.244.1.0/24
PodCIDRs:                     10.244.1.0/24
Non-terminated Pods:          (2 in total)
  Namespace     Name               CPU Requests  CPU Limits  ...  AGE
  ---------     ----               ------------  ----------  ...  ---
  kube-system   kindnet-4xmjh      100m (1%)     100m (1%)   ...  13d
  kube-system   kube-proxy-dgkfm   0 (0%)        0 (0%)      ...  13d
Allocated resources:
  (Total limits may be over 100 percent, i.e., overcommitted.)
  Resource           Requests   Limits
  --------           --------   ------
  cpu                100m (1%)  100m (1%)
  memory             50Mi (0%)  50Mi (0%)
  ephemeral-storage  0 (0%)     0 (0%)
  hugepages-1Gi      0 (0%)     0 (0%)
  hugepages-2Mi      0 (0%)     0 (0%)
Events:
  Type    Reason                   Age    From                     Message
  ----    ------                   ----   ----                     -------
  Normal  Starting                 3m50s  kubelet, kind-worker2    ...
  Normal  NodeAllocatableEnforced  3m50s  kubelet, kind-worker2    ...
  Normal  NodeHasSufficientMemory  3m50s  kubelet, kind-worker2    ...
  Normal  NodeHasNoDiskPressure    3m50s  kubelet, kind-worker2    ...
  Normal  NodeHasSufficientPID     3m50s  kubelet, kind-worker2    ...
  Normal  Starting                 3m49s  kube-proxy, kind-worker2 ...
```

As you can see, the `kubectl describe` command displays all the information you previously found in the YAML manifest of the Node object, but in a more readable form. You can see the


```
                    Volumes.kubernetes.io/controller-management-attach-deta...
创建时间戳：        2020 年 5 月 3 日星期日 17:09:48 +0200
污点：              <无>
无法安排：          错误的
租：
  持有人身份：      善良的工人2
  获取时间：        <未设置>
  续订时间：        2020 年 5 月 17 日星期日 16:15:03 +0200
条件：类型
                    地位    ...  原因                       信息
  ----              ------  ---  ------                     -------
  内存压力          错误的  ...  Kubelet有足够的内存        ...
  磁盘压力          错误的  ...  Kubelet没有磁盘压力        ...
  PID压力           错误的  ...  Kubelet有足够的PID         ...
  准备好            真的    ...  Kubelet就绪                ...
地址：
  内部IP:     172.18.0.4
  主机名：    善良的工人2
容量：  中
央处理                8
器临时存储：          401520944Ki
  巨大页面-1Gi：      0
  巨大的页面-2Mi：    0
  记忆：              32720824Ki
  豆荚：              110
可分配：
  ...
系统信息：
  ...
PodCIDR：                     10.244.1.0/24
PodCIDR：                     10.244.1.0/24
非终止 Pod：                  (共2个)
  命名空间    姓名              CPU 请求    CPU 限制    ...  年龄
  ---------   ----              ----------  ----------  ...  ---
  kube 系统   kindnet-4xmjh     100m (1%)   100m (1%)   ...  13天
  kube 系统   kube-代理-dgkfm   0 (0%)      0 (0%)      ...  13天
分配的资源：
  （总限制可能超过 100%，即过度使用。）
                    要求       限制
  --------          --------   ------
  中央处理器        100m (1%)  100m (1%)
  记忆              50米 (0%)  50米 (0%)
  临时存储          0 (0%)     0 (0%)
  大页-1Gi          0 (0%)     0 (0%)
  大页-2Mi          0 (0%)     0 (0%)
事件：
类型    原因                     年龄  从                       信息
  ----  ------                   ----  ----                     -------
  普通的 开始                    3分50秒kubelet，善良的工人2   ...
  普通的 NodeAllocatableEnforced 3分50秒kubelet，善良的工人2   ...
  普通的 节点有足够的内存        3分50秒kubelet，善良的工人2   ...
  普通的 节点没有磁盘压力        3分50秒kubelet，善良的工人2   ...
  普通的 节点有足够的PID         3分50秒kubelet，善良的工人2   ...
  普通的 开始                    3分49秒kube-proxy、kind-worker2..
```

如您所见，kubectl describe 命令显示了您之前在 Node 对象的 YAML 清单中找到的所有信息，但以更易读的形式。您可以看到

name, IP address, and hostname, as well as the conditions and available capacity of the node.

INSPECTING OTHER OBJECTS RELATED TO THE NODE

In addition to the information stored in the Node object itself, the `kubectl describe` command also displays the pods running on the node and the total amount of compute resources allocated to them. Below is also a list of events related to the node.

This additional information isn't found in the Node object itself, but is collected by the kubectl tool from other API objects. For example, the list of pods running on the node is obtained by retrieving Pod objects via the `pods` resource.

If you run the `describe` command yourself, no events may be displayed. This is because only events that have occurred recently are shown. For Node objects, unless the node has resource capacity issues, you'll only see events if you've recently (re)started the node.

Virtually every API object kind has events associated with it. Since they are crucial for debugging a cluster, they warrant a closer look before you start exploring other objects.

4.3 Observing cluster events via Event objects

As controllers perform their task of reconciling the actual state of an object with the desired state, as specified in the object's `spec` field, they generate events to reveal what they have done. Two types of events exist: Normal and Warning. Events of the latter type are usually generated by controllers when something prevents them from reconciling the object. By monitoring this type of events, you can be quickly informed of any problems that the cluster encounters.

4.3.1 Introducing the Event object

Like everything else in Kubernetes, events are represented by Event objects that are created and read via the Kubernetes API. As the following figure shows, they contain information about what happened to the object and what the source of the event was. Unlike other objects, each Event object is deleted one hour after its creation to reduce the burden on etcd, the data store for Kubernetes API objects.

102

名称、IP 地址和主机名，以及节点的条件和可用容量。

检查与节点相关的其他对象

除了存储在 Node 对象本身中的信息之外，kubectl describe 命令还显示节点上运行的 pod 以及分配给它们的计算资源总量。下面也是与该节点相关的事件列表。

这些附加信息在 Node 对象本身中找不到，而是由 kubectl 工具从其他 API 对象收集。例如，通过 pods 资源检索 Pod 对象来获取节点上运行的 pod 列表。

如果您自己运行describe 命令，则可能不会显示任何事件。这是因为仅显示最近发生的事件。对于 Node 对象，除非节点存在资源容量问题，否则只有最近（重新）启动该节点时您才会看到事件。

事实上，每种 API 对象类型都有与其关联的事件。由于它们对于调试集群至关重要，因此在开始探索其他对象之前需要仔细查看它们。

4.3 通过Event对象观察集群事件

当控制器执行协调对象的实际状态与所需状态（如对象的规范字段中指定）的任务时，它们会生成事件来揭示它们所做的事情。存在两种类型的事件：正常和警告。当控制器无法协调对象时，通常会生成后一种类型的事件。通过监控此类事件，您可以快速获悉集群遇到的任何问题。

4.3.1 介绍Event对象

与 Kubernetes 中的其他所有内容一样，事件由通过 Kubernetes API 创建和读取的事件对象表示。如下图所示，它们包含有关对象发生的情况以及事件来源的信息。与其他对象不同，每个 Event 对象在创建一小时后就会被删除，以减轻 Kubernetes API 对象的数据存储 etcd 的负担。

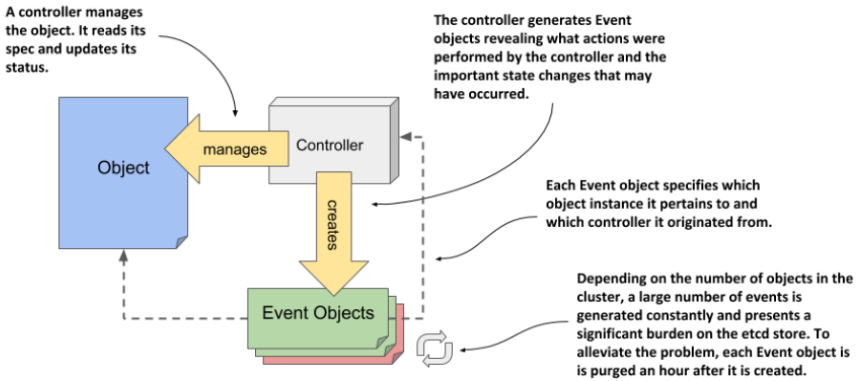


Figure 4.7 The relationship between Event objects, controllers, and other API objects.

NOTE The amount of time to retain events is configurable via the API server's command-line options.

LISTING EVENTS USING KUBECTL GET EVENTS

The events displayed by `kubectl describe` refer to the object you specify as the argument to the command. Due to their nature and the fact that many events can be created for an object in a short time, they aren't part of the object itself. You won't find them in the object's YAML manifest, as they exist on their own, just like Nodes and the other objects you've seen so far.

NOTE If you want to follow the exercises in this section in your own cluster, you may need to restart one of the nodes to ensure that the events are recent enough to still be present in etcd. If you can't do this, don't worry, and just skip doing these exercises yourself, as you'll also be generating and inspecting events in the exercises in the next chapter.

Because Events are standalone objects, you can list them using `kubectl get events`, as shown in the next listing.

Listing 4.7 Listing events using kubectl get events

```
$ kubectl get ev
LAST
SEEN  TYPE    REASON                   OBJECT             MESSAGE
48s   Normal  Starting                 node/kind-worker2  Starting kubelet.
48s   Normal  NodeAllocatableEnforced  node/kind-worker2  Updated Node A...
48s   Normal  NodeHasSufficientMemory  node/kind-worker2  Node kind-work...
48s   Normal  NodeHasNoDiskPressure    node/kind-worker2  Node kind-work...
48s   Normal  NodeHasSufficientPID     node/kind-worker2  Node kind-work...
47s   Normal  Starting                 node/kind-worker2  Starting kube-...
```


103

图 4.7 事件对象、控制器和其他 API 对象之间的关系。

注意 保留事件的时间量可通过 API 服务器的命令行选项进行配置。

使用 KUBECTL GET EVENTS 列出事件

kubectl describe 显示的事件引用您指定为命令参数的对象。由于它们的性质以及可以在短时间内为对象创建许多事件的事实，它们不是对象本身的一部分。您不会在对象的 YAML 清单中找到它们，因为它们独立存在，就像节点和您到目前为止看到的其他对象一样。

注意如果您想在自己的集群中执行本节中的练习，您可能需要重新启动其中一个节点，以确保事件足够新以仍然存在于 etcd 中。如果你做不到，就不要这样做
不用担心，只需跳过自己做这些练习，因为您还将生成和检查中的事件
下一章的练习

由于事件是独立的对象，因此您可以使用 kubectl get events 列出它们，如下一个清单所示。

清单 4.7 使用 kubectl get events 列出事件

```
$ kubectl  get ev 最后看到的类型 原因 对象 消息
48s 正常 启动node/kind-worker2 启动kubelet 。
48s 正常 NodeAllocatableEnforced  node/kind-worker2 更新节点 A...
48s 正常 NodeHasSufficientMemory  节点/kind-worker2 节点 kind-
work...
48s 正常 NodeHasNoDiskPressure  节点/kind-worker2 节点 kind-
work
```

NOTE The previous listing uses the short name `ev` in place of `events`.

You'll notice that some events displayed in the listing match the status conditions of the Node. This is often the case, but you'll also find additional events. The two events with the reason `Starting` are two such examples. In the case at hand, they signal that the Kubelet and the Kube Proxy components have been started on the node. You don't need to worry about these components yet. They are explained in the third part of the book.

UNDERSTANDING WHAT'S IN AN EVENT OBJECT

As with other objects, the `kubectl get` command only outputs the most important object data. To display additional information, you can enable additional columns by executing the command with the `-o wide` option:

```
$ kubectl get ev -o wide
```

The output of this command is extremely wide and is not listed here in the book. Instead, the information that is displayed is explained in the following table.

Property	Description
Name	The name of this Event object instance. Useful only if you want to retrieve the given object from the API.
Type	The type of the event. Either `Normal` or `Warning`.
Reason	The machine-facing description why the event occurred.
Source	The component that reported this event. This is usually a controller.
Object	The object instance to which the event refers. For example, node/xyz.
Sub-object	The sub-object to which the event refers. For example, what container of the pod.
Message	The human-facing description of the event.
First seen	The first time this event occurred. Remember that each Event object is deleted after a while, so this may not be the first time that the event actually occurred.
Last seen	Events often occur repeatedly. This field indicates when this event last occurred.
Count	The number of times this event has occurred.

Table 4.1 Properties of the Event object

TIP As you complete the exercises throughout this book, you may find it useful to run the `kubectl get events` command each time you make changes to one of your objects. This will help you learn what happens beneath the surface.

DISPLAYING ONLY WARNING EVENTS

Unlike the `kubectl describe` command, which only displays events related to the object you're describing, the `kubectl get events` command displays all events. This is useful if

注意 前面的列表使用短名称 ev 代替事件。

您会注意到列表中显示的某些事件与节点的状态条件匹配。这种情况很常见，但您还会发现其他事件。原因为“开始”的两个事件就是两个此类示例。在本例中，它们表示 Kubelet 和 Kube Proxy 组件已在节点上启动。您还不需要担心这些组件。本书的第三部分对此进行了解释。

了解事件对象中的内容

与其他对象一样，kubectl get 命令仅输出最重要的对象数据。要显示附加信息，您可以通过执行带有 -o Wide 选项的命令来启用附加列：

```
$ kubectl 获取 ev -o 宽
```

该命令的输出非常广泛，本书中没有列出。相反，下表对显示的信息进行了说明。

属性说明

名称 此事件对象实例的名称。仅当您想从 API 检索给定对象时才有用。

类型 事件的类型。正常或警告。

原因 面向机器的事件发生原因的描述。

源 报告此事件的组件。这通常是一个控制器。

对象 事件引用的对象实例。例如，节点/xyz。

子对象 事件引用的子对象。比如pod是什么容器。

消息 事件的人性化描述。

首次出现 此事件第一次发生。请记住，每个 Event 对象都会在一段时间后被删除，因此这可能不是该事件第一次实际发生。

最后见到的事件经常重复发生。该字段指示该事件上次发生的时间。

计数 此事件发生的次数。

表 4.1 Event 对象的属性

提示 当您完成本书中的练习时，您可能会发现每次对某个对象进行更改时运行 kubectl get events 命令很有用。这将帮助您了解什么
发生在表面之下。

仅显示警告事件

与 kubectl describe 命令仅显示与您所描述的对象相关的事件不同，kubectl get events 命令显示所有事件

you want to check if there are events that you should be concerned about. You may want to ignore events of type `Normal` and focus only on those of type `Warning`.

The API provides a way to filter objects through a mechanism called field selectors. Only objects where the specified field matches the specified selector value are returned. You can use this to display only Warning events. The `kubectl get` command allows you to specify the field selector with the `--field-selector` option. To list only events that represent warnings, you execute the following command:

```
$ kubectl get ev --field-selector type=Warning
No resources found in default namespace.
```

If the command does not print any events, as in the above case, no warnings have been recorded in your cluster recently.

You may wonder how I knew the exact name of the field to be used in the field selector and what its exact value should be (perhaps it should have been lower case, for example). Hats off if you guessed that this information is provided by the `kubectl explain events` command. Since events are regular API objects, you can use it to look up documentation on the event objects' structure. There you'll learn that the `type` field can have two values: either `Normal` or `Warning`.

4.3.2 Examining the YAML of the Event object

To inspect the events in your cluster, the commands `kubectl describe` and `kubectl get events` should be sufficient. Unlike other objects, you'll probably never have to display the complete YAML of an Event object. But I'd like to take this opportunity to show you an annoying thing about Kubernetes object manifests that the API returns.

EVENT OBJECTS HAVE NO SPEC AND STATUS SECTIONS

If you use the `kubectl explain` to explore the structure of the Event object, you'll notice that it has no `spec` or `status` sections. Unfortunately, this means that its fields are not as nicely organized as in the Node object, for example.

Inspect the following listing and see if you can easily find the object kind, metadata and other fields.

Listing 4.8 The YAML manifest of an Event object

```
apiVersion: v1                                          #A
count: 1
eventTime: null
firstTimestamp: "2020-05-17T18:16:40Z"
involvedObject:
  kind: Node
  name: kind-worker2
  uid: kind-worker2
kind: Event                                             #B
lastTimestamp: "2020-05-17T18:16:40Z"
message: Starting kubelet.
metadata:                                               #C
  creationTimestamp: "2020-05-17T18:16:40Z"             #C
```


您想要检查是否有您应该关注的事件。您可能希望忽略“正常”类型的事件，而仅关注“警告”类型的事件。

API 提供了一种通过称为字段选择器的机制来过滤对象的方法。仅返回指定字段与指定选择器值匹配的对象。您可以使用它来仅显示警告事件。 kubectl get 命令允许您指定

带有 --field-selector 选项的字段选择器。要仅列出代表警告的事件，请执行以下命令：

```
$ kubectl get ev --field-selector type=警
告 在默认命名空间中找不到资源。
```

如果该命令没有打印任何事件，如上述情况，则说明您的集群最近没有记录任何警告。

您可能想知道我如何知道字段选择器中要使用的字段的确切名称以及其确切值应该是什么（例如，也许应该是小写）。如果您猜到此信息是由 kubectlexplain events 命令提供的，那么请向您致敬。由于事件是常规 API 对象，因此您可以使用它来查找有关事件对象结构的文档。在那里您将了解到类型字段可以有两个值：

正常或警告。

4.3.2 检查Event对象的YAML

要检查集群中的事件，命令 kubectl describe 和 kubectl get events 应该足够了。与其他对象不同，您可能永远不需要显示

事件对象的完整 YAML。但我想借此机会向您展示有关 API 返回的 Kubernetes 对象清单的一件烦人的事情。
事件对象没有规格和状态部分

如果您使用 kubectlexplain 来探索 Event 对象的结构，您会注意到它没有规范或状态部分。不幸的是，这意味着它的字段不像 Node 对象那样组织得很好。

检查以下列表，看看是否可以轻松找到对象类型、元数据和其他领域。

清单 4.8 Event 对象的 YAML 清单

api 版本：v1 #A

计数：1 eventTime：null firstTimestamp：“2020-05-17T18：16：40Z” 涉及对象：kind：节点名称：kind-worker2 uid：kind-worker2

种类：事件 #B

lastTimestamp：″2020-05-17T18:16:40Z″ 消息：启动 kubelet。元数据：#C

创建时间戳：“2020-05-17T18:16:40Z”#C

```
  managedFields:                                                    #C
  - ...                                                             #C
  name: kind-worker2.160fe38fc0bc3703             #D                #C
  namespace: default                                                #C
  resourceVersion: "3528471"                                        #C
  selfLink: /api/v1/namespaces/default/events/kind-worker2.160f...  #C
  uid: da97e812-d89e-4890-9663-091fd1ec5e2d                         #C
reason: Starting
reportingComponent: ""
reportingInstance: ""
source:
  component: kubelet
  host: kind-worker2
type: Normal
```

#A The apiVersion field is easy to spot
#B The kind field is hard to find
#C The object's metadata is here
#D The object's name is hidden here

You will surely agree that the YAML manifest in the listing is disorganized. The fields are listed alphabetically instead of being organized into coherent groups. This makes it difficult for us humans to read. It looks so chaotic that it's no wonder that many people hate to deal with Kubernetes YAML or JSON manifests, since both suffer from this problem.

In contrast, the earlier YAML manifest of the Node object was relatively easy to read, because the order of the top-level fields is what one would expect: `apiVersion`, `kind`, `metadata`, `spec`, and `status`. You'll notice that this is simply because the alphabetical order of the five fields just happens to make sense. But the fields under those fields suffer from the same problem, as they are also sorted alphabetically.

YAML is supposed to be easy for people to read, but the alphabetical field order in Kubernetes YAML breaks this. Fortunately, most objects contain the `spec` and `status` sections, so at least the top-level fields in these objects are well organized. As for the rest, you'll just have to accept this unfortunate aspect of dealing with Kubernetes manifests.

4.4 Summary

In this chapter, you've learned:

- Kubernetes provides a RESTful API for interaction with a cluster. API Objects map to actual components that make up the cluster, including applications, load balancers, nodes, storage volumes, and many others.
- An object instance can be represented by many resources. A single object type can be exposed through several resources that are just different representations of the same thing.
- Kubernetes API objects are described in YAML or JSON manifests. Objects are created by posting a manifest to the API. The status of the object is stored in the object itself and can be retrieved by requesting the object from the API with a `GET` request.
- All Kubernetes API objects contain Type and Object Metadata, and most have a `spec` and `status` sections. A few object types don't have these two sections, because they


```
  管理字段：                                                    #C
  - ...                                                         #C
  名称：kind-worker2.160fe38fc0bc3703          #D               #C
  命名空间：默认                                                #C
  资源版本："3528471"                                           #C
  自链接：/api/v1/namespaces/default/events/kind-worker2.160f#C.
  uid：da97e812-d89e-4890-9663-091fd1ec5e2d                     #C
原因：开始

reportingComponent :
""
  组件：kubelet 主
  机：kind-worker2
类型：普通
```

#A apiVersion 字段很容易找到
#B kind 字段很难找到 #C 对象
的元数据在这里 #D 对象的名称
隐藏在这里

您肯定会同意列表中的 YAML 清单是杂乱无章的。这些字段按字母顺序列出，而不是组织成连贯的组。这使得我们人类阅读变得困难。它看起来如此混乱，难怪很多人讨厌处理 Kubernetes YAML 或 JSON 清单，因为两者都面临这个问题。

相比之下，Node 对象的早期 YAML 清单相对容易阅读，因为顶级字段的顺序符合预期：apiVersion、kind、metadata、spec 和 status。你会注意到这只是因为字母顺序

这五个字段恰好是有意义的。但这些字段下的字段也遇到同样的问题，因为它们也是按字母顺序排序的。

YAML 应该易于人们阅读，但 Kubernetes YAML 中的字母字段顺序打破了这一点。幸运的是，大多数对象都包含规范和状态部分，因此至少这些对象中的顶级字段组织良好。至于其余的，您只需要接受处理 Kubernetes 清单的这个不幸的方面。

4.4 总结

在本章中，您学习了：

- Kubernetes 提供RESTful API 用于与集群交互。 API 对象映射到组成集群的实际组件，包括应用程序、负载均衡器、节点、存储卷等。
- 一个对象实例可以由许多资源表示。单个对象类型可以通过多个资源公开，这些资源只是同一事物的不同表示。
- Kubernetes API 对象在YAML 或JSON 清单中描述。对象是通过将清单发布到 API 来创建的。对象的状态存储在对象本身中，可以通过使用 GET 请求从 API 请求对象来检索。
- 所有Kubernetes API 对象都包含类型和对象元数据，并且大多数都有规范和状态部分。一些对象类型没有这两个部分

only contain static data.

- Controllers bring objects to life by constantly watching for changes in their `spec`, updating the cluster state and reporting the current state via the object's `status` field.
- As controllers manage Kubernetes API objects, they emit events to reveal what actions they have performed. Like everything else, events are represented by Event objects and can be retrieved through the API. Events signal what is happening to a Node or other object. They show what has recently happened to the object and can provide clues as to why it is broken.
- The `kubectl explain` command provides a quick way to look up documentation on a specific object kind and its fields from the command line.
- The status in a Node object contains information about the node's IP address and hostname, its resource capacity, conditions, cached container images and other information about the node. Which pods are running on the node is not part of the node's status, but the `kubectl describe node` commands gets this information from the `pods` resource.
- Many object types use status conditions to signal the state of the component that the object represents. For nodes, these conditions are `MemoryPressure`, `DiskPressure` and `PIDPressure`. Each condition is either `True`, `False`, or `Unknown` and has an associated `reason` and `message` that explain why the condition is in the specified state.

You should now be familiar with the general structure of the Kubernetes API objects. In the next chapter, you'll learn about the Pod object, the fundamental building block which represents one running instance of your application.

107

仅包含静态数据。

- 控制器通过不断监视其规范的变化、更新集群状态并通过对象的状态字段报告当前状态来使对象栩栩如生。

- 当控制器管理Kubernetes API 对象时，它们会发出事件来显示它们执行了哪些操作。与其他事物一样，事件由 Event 对象表示，并且可以通过 API 检索。事件表示节点或其他对象发生的情况。它们显示了物体最近发生的情况，并可以提供有关物体损坏原因的线索。

- kubectl解释命令提供了一种从命令行查找有关特定对象类型及其字段的文档的快速方法。

- Node 对象中的状态包含有关节点的 IP 地址和主机名、其资源容量、条件、缓存的容器映像以及有关节点的其他信息的信息。节点上运行的 pod 不是节点状态的一部分，但 kubectl 描述节点命令从 pod 资源获取此信息。

- 许多对象类型使用状态条件来表示对象所代表的组件的状态。对于节点，这些条件是 MemoryPressure、DiskPressure 和 PIDPressure。每个条件要么是 True、False 要么是 Unknown，并且具有关联的原因和消息，用于解释该条件为何处于指定状态。

您现在应该熟悉 Kubernetes API 对象的一般结构。在下一章中，您将了解 Pod 对象，它是代表应用程序的一个运行实例的基本构建块。

5

Running applications in Pods

This chapter covers

- Understanding how and when to group containers
- Running an application by creating a Pod object from a YAML file
- Communicating with an application, viewing its logs, and exploring its environment
- Adding a sidecar container to extend the pod's main container
- Initializing pods by running init containers at pod startup

Let me refresh your memory with a diagram that shows the three types of objects you created in chapter 3 to deploy a minimal application on Kubernetes. Figure 5.1 shows how they relate to each other and what functions they have in the system.

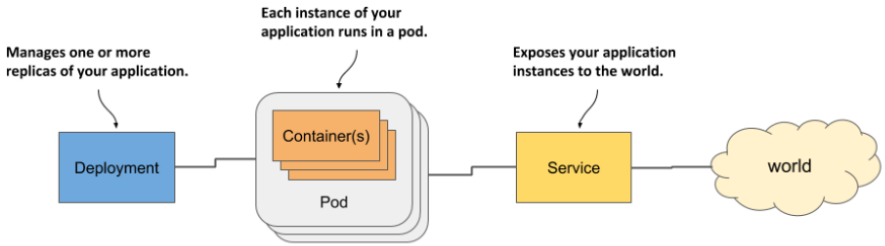


Figure 5.1 Three basic object types comprising a deployed application

You now have a basic understanding of how these objects are exposed via the Kubernetes API. In this and the following chapters, you'll learn about the specifics of each of them and

5

在 Pod 中运行应用程序

本章涵盖

- 了解如何以及何时对容器进行分组
- 通过从 YAML 文件创建 Pod 对象来运行应用程序
- 与应用程序通信、查看其日志并探索其环境
- 添加 sidecar 容器来扩展 Pod 的主容器
- 通过在 Pod 启动时运行 init 容器来初始化 Pod

让我用一个图表来刷新您的记忆，该图表显示了您在第 3 章中创建的三种类型的对象，以在 Kubernetes 上部署最小应用程序。图 5.1 显示了它们之间的关系以及它们在系统中的功能。

图 5.1 构成已部署应用程序的三种基本对象类型

您现在已经基本了解了如何通过 Kubernetes API 公开这些对象。在本章和后续章节中

many others that are typically used to deploy a full application. Let's start with the Pod object, as it represents the central, most important concept in Kubernetes - a running instance of your application.

5.1 Understanding pods

You've already learned that a pod is a co-located group of containers and the basic building block in Kubernetes. Instead of deploying containers individually, you deploy and manage a group of containers as a single unit — a pod. Although pods may contain several, it's not uncommon for a pod to contain just a single container. When a pod has multiple containers, all of them run on the same worker node — a single pod instance never spans multiple nodes. Figure 5.2 will help you visualize this information.

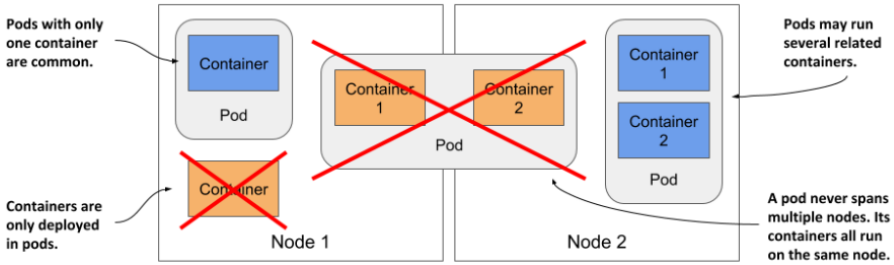


Figure 5.2 All containers of a pod run on the same node. A pod never spans multiple nodes.

5.1.1 Understanding why we need pods

Let's discuss why we need to run multiple containers together, as opposed to, for example, running multiple processes in the same container.

UNDERSTANDING WHY ONE CONTAINER SHOULDN'T CONTAIN MULTIPLE PROCESSES

Imagine an application that consists of several processes that communicate with each other via *IPC* (Inter-Process Communication) or shared files, which requires them to run on the same computer. In chapter 2, you learned that each container is like an isolated computer or virtual machine. A computer typically runs several processes; containers can also do this. You can run all the processes that make up an application in just one container, but that makes the container very difficult to manage.

Containers are *designed* to run only a single process, not counting any child processes that it spawns. Both container tooling and Kubernetes were developed around this fact. For example, a process running in a container is expected to write its logs to standard output. Docker and Kubernetes commands that you use to display the logs only show what has been captured from this output. If a single process is running in the container, it's the only writer, but if you run multiple processes in the container, they all write to the same output. Their

许多其他通常用于部署完整应用程序的方法。让我们从 Pod 对象开始，因为它代表 Kubernetes 中最核心、最重要的概念——应用程序的运行实例。

5.1 了解 Pod

您已经了解到，Pod 是一组位于同一位置的容器，是 Kubernetes 中的基本构建块。您无需单独部署容器，而是将一组容器作为一个单元（即 Pod）进行部署和管理。尽管 Pod 可能包含多个容器，但一个 Pod 仅包含一个容器的情况并不罕见。当一个 Pod 具有多个容器时，所有容器都在同一个工作节点上运行 - 单个 Pod 实例永远不会跨越多个节点。图 5.2 将帮助您可视化此信息。

图5.2 一个pod的所有容器都运行在同一个节点上。一个 Pod 永远不会跨越多个节点。

5.1.1 理解为什么我们需要 pod

让我们讨论一下为什么我们需要一起运行多个容器，而不是在同一个容器中运行多个进程。

了解为什么一个容器不应包含多个进程

想象一个应用程序由多个进程组成，这些进程通过 IPC（进程间通信）或共享文件相互通信，这要求它们运行在同一台计算机上。在第 2 章中，您了解到每个容器就像一台独立的计算机或虚拟机。一台计算机通常运行多个进程；容器也可以做到这一点。您可以在一个容器中运行组成应用程序的所有进程，但这使得容器非常难以管理。

容器被设计为仅运行单个进程，不计算它产生的任何子进程。容器工具和 Kubernetes 都是围绕这一事实开发的。例如，容器中运行的进程应将其日志写入标准输出。用于显示日志的 Docker 和 Kubernetes 命令仅显示从此输出中捕获的内容。如果容器中运行单个进程，则它是唯一的写入器，但如果容器中运行多个进程，它们都会写入相同的输出

logs are therefore intertwined, and it is difficult to tell which process each logged line belongs to.

Another indication that containers should only run a single process is the fact that the container runtime only restarts the container when the container's root process dies. It doesn't care about any child processes created by this root process. If it spawns child processes, it alone is responsible for keeping all these processes running.

To take full advantage of the features provided by the container runtime, you should consider running only one process in each container.

UNDERSTANDING HOW A POD COMBINES MULTIPLE CONTAINERS

Since you shouldn't run multiple processes in a single container, it's evident you need another higher-level construct that allows you to run related processes together even when divided into multiple containers. These processes must be able to communicate with each other like processes in a normal computer. And that is why pods were introduced.

With a pod, you can run closely related processes together, giving them (almost) the same environment as if they were all running in a single container. These processes are somewhat isolated, but not completely - they share some resources. This gives you the best of both worlds. You can use all the features that containers offer, but also allow processes to work together. A pod makes these interconnected containers manageable as one unit.

In the second chapter, you learned that a container uses its own set of Linux namespaces, but it can also share some with other containers. This sharing of namespaces is exactly how Kubernetes and the container runtime combine containers into pods.

As shown in figure 5.3, all containers in a pod share the same Network namespace and thus the network interfaces, IP address(es) and port space that belong to it.

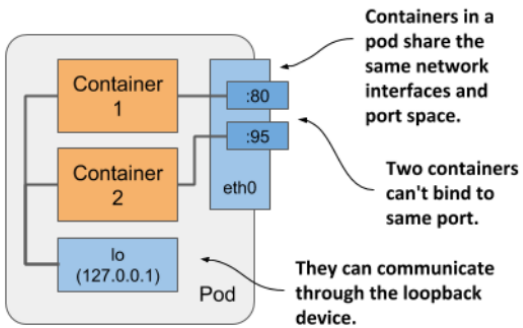


Figure 5.3 Containers in a pod share the same network interfaces

因此，日志相互交织，很难判断每条记录的行属于哪个进程。

容器应该只运行单个进程的另一个迹象是，容器运行时仅在容器的根进程终止时重新启动容器。它不关心该根进程创建的任何子进程。如果它生成子进程，则它独自负责保持所有这些进程的运行。

要充分利用容器运行时提供的功能，您应该考虑在每个容器中仅运行一个进程。

了解 Pod 如何组合多个容器

由于您不应该在单个容器中运行多个进程，因此显然您需要另一个更高级别的构造，即使在划分到多个容器中时也允许您一起运行相关进程。这些进程必须能够像普通计算机中的进程一样相互通信。这就是引入 pod 的原因。

使用 Pod，您可以一起运行密切相关的进程，为它们提供（几乎）相同的环境，就好像它们都在单个容器中运行一样。这些进程在某种程度上是孤立的，但并不完全 - 它们共享一些资源。这让您两全其美。您可以使用容器提供的所有功能，还可以允许进程协同工作。 Pod 使这些互连的容器可以作为一个单元进行管理。

在第二章中，您了解到容器使用自己的一组 Linux 命名空间，但它也可以与其他容器共享一些命名空间。这种命名空间的共享正是 Kubernetes 和容器运行时将容器组合到 Pod 中的方式。

如图 5.3 所示，Pod 中的所有容器共享相同的 Network 命名空间，并且因此属于它的网络接口、IP 地址和端口空间。

图5

Because of the shared port space, processes running in containers of the same pod can't be bound to the same port numbers, whereas processes in other pods have their own network interfaces and port spaces, eliminating port conflicts between different pods.

All the containers in a pod also see the same system hostname, because they share the UTS namespace, and can communicate through the usual IPC mechanisms because they share the IPC namespace. A pod can also be configured to use a single PID namespace for all its containers, which makes them share a single process tree, but you must explicitly enable this for each pod individually.

> **NOTE** When containers of the same pod use separate PID namespaces, they can't see each other or send process signals like `SIGTERM` or `SIGINT` between them.

It's this sharing of certain namespaces that gives the processes running in a pod the impression that they run together, even though they run in separate containers.

In contrast, each container always has its own Mount namespace, giving it its own file system, but when two containers must share a part of the file system, you can add a *volume* to the pod and mount it into both containers. The two containers still use two separate Mount namespaces, but the shared volume is mounted into both. You'll learn more about volumes in chapter 7.

5.1.2 Organizing containers into pods

You can think of each pod as a separate computer. Unlike virtual machines, which typically host multiple applications, you typically run only one application in each pod. You never need to combine multiple applications in a single pod, as pods have almost no resource overhead. You can have as many pods as you need, so instead of stuffing all your applications into a single pod, you should divide them so that each pod runs only closely related application processes.

Let me illustrate this with a concrete example.

SPLITTING A MULTI-TIER APPLICATION STACK INTO MULTIPLE PODS

Imagine a simple system composed of a front-end web server and a back-end database. I've already explained that the front-end server and the database shouldn't run in the same container, as all the features built into containers were designed around the expectation that not more than one process runs in a container. If not in a single container, should you then run them in separate containers that are all in the same pod?

Although nothing prevents you from running both the front-end server and the database in a single pod, this isn't the best approach. I've explained that all containers of a pod always run co-located, but do the web server and the database have to run on the same computer? The answer is obviously no, as they can easily communicate over the network. Therefore you shouldn't run them in the same pod.

If both the front-end and the back-end are in the same pod, both run on the same cluster node. If you have a two-node cluster and only create this one pod, you are using only a single worker node and aren't taking advantage of the computing resources available on the second node. This means wasted CPU, memory, disk storage and bandwidth. Splitting the

由于共享端口空间，同一 Pod 的容器中运行的进程无法绑定到相同的端口号，而其他 Pod 中的进程则拥有自己的网络接口和端口空间，从而消除了不同 Pod 之间的端口冲突。

Pod 中的所有容器也会看到相同的系统主机名，因为它们共享 UTS 命名空间，并且可以通过通常的 IPC 机制进行通信，因为它们共享 IPC 命名空间。还可以将 Pod 配置为对其所有容器使用单个 PID 命名空间，这使它们共享单个进程树，但您必须分别为每个 Pod 显式启用此功能。

> 注意：当同一 Pod 的容器使用单独的 PID 命名空间时，它们无法看到彼此，也无法在它们之间发送 SIGTERM 或 SIGINT 等进程信号。

正是这种共享某些命名空间的方式，让 Pod 中运行的进程给人一种它们一起运行的印象，尽管它们运行在单独的容器中。

相比之下，每个容器总是有自己的 Mount 命名空间，赋予它自己的文件系统，但是当两个容器必须共享文件系统的一部分时，您可以向 pod 添加一个卷并将其挂载到两个容器中。这两个容器仍然使用两个独立的 Mount 命名空间，但共享卷被安装到两个容器中。您将在第 7 章中了解有关卷的更多信息。

5.1.2 将容器组织成 pod

您可以将每个 Pod 视为一台单独的计算机。与通常托管多个应用程序的虚拟机不同，您通常在每个 Pod 中仅运行一个应用程序。您永远不需要将多个应用程序组合在一个 pod 中，因为 pod 几乎没有资源开销。您可以根据需要拥有任意数量的 pod，因此不应将所有应用程序填充到一个 pod 中，而是应该将它们分开，以便每个 pod 只运行密切相关的应用程序进程。

让我用一个具体的例子来说明这一点。

将多层应用程序堆栈拆分为多个 Pod

想象一个由前端 Web 服务器和后端数据库组成的简单系统。我已经解释过，前端服务器和数据库不应该在同一个容器中运行，因为容器中内置的所有功能都是围绕容器中运行不超过一个进程这一预期而设计的。如果不在单个容器中，您是否应该在同一个 pod 中的单独容器中运行它们？

尽管没有什么可以阻止您在单个 Pod 中运行前端服务器和数据库，但这并不是最好的方法。我已经解释过，Pod 的所有容器始终在同一位置运行，但是 Web 服务器和数据库必须在同一台计算机上运行吗？答案显然是否定的，因为他们可以轻松地通过网络进行通信。因此，您不应该在同一个 Pod 中运行它们。

如果前端和后端都在同一个 pod 中，则两者都运行在同一个集群节点上。如果您有一个双节点集群并且仅创建一个 Pod，则您仅使用单个工作节点，并且没有利用第二个节点上可用的计算资源。这意味着 CPU、内存、磁盘存储和带宽的浪费

containers into two pods allows Kubernetes to place the front-end pod on one node and the back-end pod on the other, thereby improving the utilization of your hardware.

SPLITTING INTO MULTIPLE PODS TO ENABLE INDIVIDUAL SCALING

Another reason not to use a single pod has to do with horizontal scaling. A pod is not only the basic unit of deployment, but also the basic unit of scaling. In chapter 2 you scaled the Deployment object and Kubernetes created additional pods – additional replicas of your application. Kubernetes doesn't replicate containers within a pod. It replicates the entire pod.

Front-end components usually have different scaling requirements than back-end components, so we typically scale them individually. When your pod contains both the front-end and back-end containers and Kubernetes replicates it, you end up with multiple instances of both the front-end and back-end containers, which isn't always what you want. Stateful back-ends, such as databases, usually can't be scaled. At least not as easily as stateless front ends. If a container has to be scaled separately from the other components, this is a clear indication that it must be deployed in a separate pod.

The following figure illustrates what was just explained.

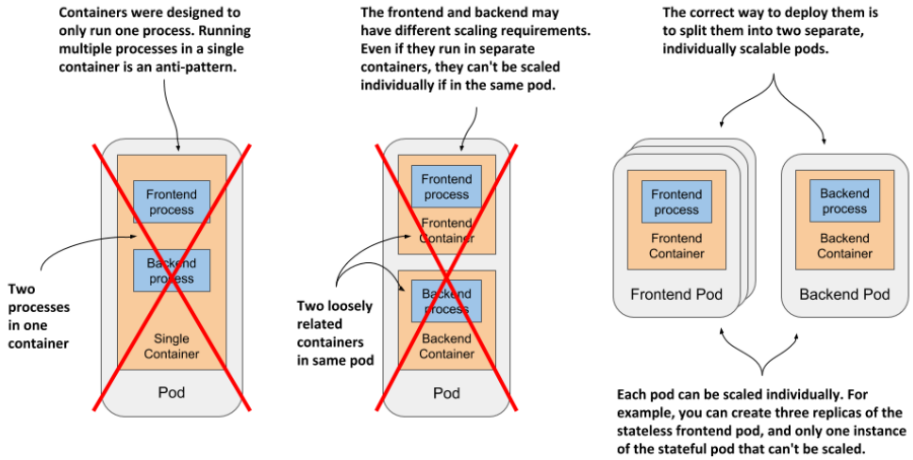


Figure 5.4 Splitting an application stack into pods

Splitting application stacks into multiple pods is the correct approach. But then, when does one run multiple containers in the same pod?

INTRODUCING SIDECAR CONTAINERS

Placing several containers in a single pod is only appropriate if the application consists of a primary process and one or more processes that complement the operation of the primary

将容器分成两个 Pod 允许 Kubernetes 将前端 Pod 放置在一个节点上，将后端 Pod 放置在另一个节点上，从而提高硬件的利用率。

分成多个 Pod 以实现单独扩展

不使用单个 Pod 的另一个原因与水平扩展有关。 Pod 不仅是部署的基本单位，也是伸缩的基本单位。在第 2 章中，您扩展了 Deployment 对象，并且 Kubernetes 创建了额外的 Pod——应用程序的额外副本。 Kubernetes 不会在 Pod 内复制容器。它复制整个 Pod。前端组件通常具有与后端不同的扩展要求

组件，因此我们通常单独缩放它们。当您的 Pod 包含前端和后端容器并且 Kubernetes 复制它时，您最终会获得前端和后端容器的多个实例，这并不总是您想要的。有状态的后端，例如数据库，通常无法扩展。至少不像无状态前端那么容易。如果容器必须与其他组件分开扩展，则明确表明它必须部署在单独的 Pod 中。下图说明了刚刚解释的内容。

图 5.4 将应用程序堆栈拆分为 Pod

将应用程序堆栈拆分为多个 Pod 是正确的方法。但是，什么时候可以在同一个 Pod 中运行多个容器呢?

引入 Sidecar 容器

仅当应用程序由一个主进程和一个或多个补充主进程操作的进程组成时，才适合将多个容器放置在一个 Pod 中。

process. The container in which the complementary process runs is called a *sidecar container* because it's analogous to a motorcycle sidecar, which makes the motorcycle more stable and offers the possibility of carrying an additional passenger. But unlike motorcycles, a pod can have more than one sidecar, as shown in figure 5.5.

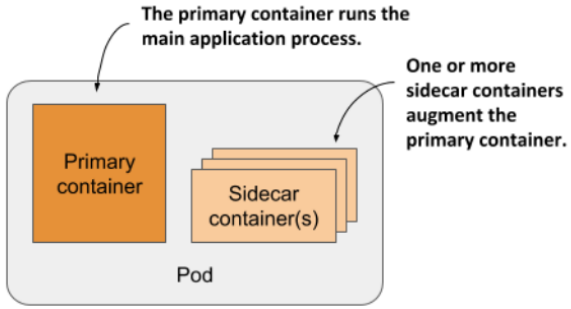


Figure 5.5 A pod with a primary and sidecar container(s)

It's difficult to imagine what constitutes a complementary process, so I'll give you some examples. In chapter 2, you deployed pods with one container that runs a Node.js application. The Node.js application only supports the HTTP protocol. To make it support HTTPS, we could add a bit more JavaScript code, but we can also do it without changing the existing application at all - by adding an additional container to the pod – a reverse proxy that converts HTTPS traffic to HTTP and forwards it to the Node.js container. The Node.js container is thus the primary container, whereas the container running the proxy is the sidecar container. Figure 5.6 shows this example.

113

过程。运行补充过程的容器称为边车容器，因为它类似于摩托车边车，这使得摩托车更稳定，并提供了承载额外乘客的可能性。但与摩托车不同的是，一个 pod 可以有多个边车，如图 5.5 所示。

图 5.5 带有主容器和边车容器的 Pod

很难想象什么是互补过程，所以我给你举一些例子。在第 2 章中，您使用一个运行 Node.js 应用程序的容器部署了 pod。 Node.js 应用程序仅支持 HTTP 协议。为了使其支持 HTTPS，我们可以添加更多的 JavaScript 代码，但我们也可以在不更改现有应用程序的情况下实现这一点 - 通过向 pod 添加一个额外的容器 - 将 HTTPS 流量转换为 HTTP 并转发它的反向代理到 Node.js 容器。因此，Node.js 容器是主容器，而运行代理的容器是 sidecar 容器。图 5.6 显示了这个示例。

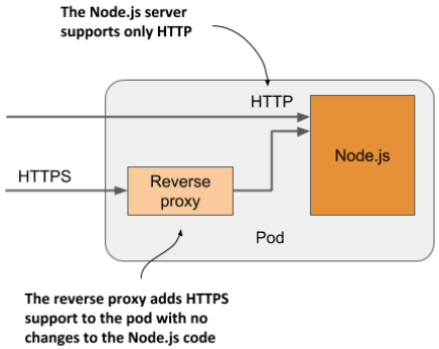


Figure 5.6 A sidecar container that converts HTTPS traffic to HTTP

NOTE You'll create this pod in section 5.4.

Another example, shown in figure 5.7, is a pod where the primary container runs a web server that serves files from its webroot directory. The other container in the pod is an agent that periodically downloads content from an external source and stores it in the web server's webroot directory. As I mentioned earlier, two containers can share files by sharing a volume. The webroot directory would be located on this volume.

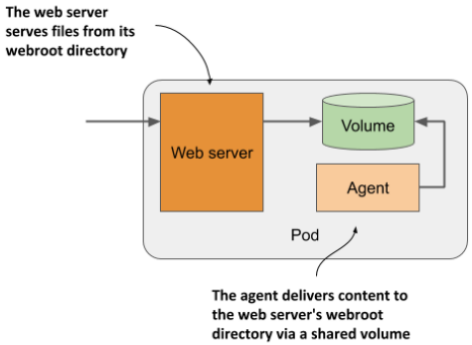


Figure 5.7 A sidecar container that delivers content to the web server container via a volume

NOTE You'll create this pod in the chapter 7.

114

图5.6 将HTTPS流量转换为HTTP的sidecar容器

注意您将在第 5.4 节中创建此 pod。

另一个示例（如图 5.7 所示）是一个 pod，其中主容器运行一个 Web 服务器，该服务器从其 webroot 目录提供文件。 Pod 中的另一个容器是一个代理，它定期从外部源下载内容并将其存储在 Web 服务器的 webroot 目录中。正如我之前提到的，两个容器可以通过共享卷来共享文件。webroot 目录将位于该卷上。

图 5.7 通过卷将内容传送到 Web 服务器容器的 sidecar 容器

注意您将在第 7 章中创建此 pod。

Other examples of sidecar containers are log rotators and collectors, data processors, communication adapters, and others.

Unlike changing the application's existing code, adding a sidecar increases the pod's resources requirements because an additional process must run in the pod. But keep in mind that adding code to legacy applications can be very difficult. This could be because its code is difficult to modify, it's difficult to set up the build environment, or the source code itself is no longer available. Extending the application by adding an additional process is sometimes a cheaper and faster option.

HOW TO DECIDE WHETHER TO SPLIT CONTAINERS INTO MULTIPLE PODS

When deciding whether to use the sidecar pattern and place containers in a single pod, or to place them in separate pods, ask yourself the following questions:

- Do these containers have to run on the same host?
- Do I want to manage them as a single unit?
- Do they form a unified whole instead of being independent components?
- Do they have to be scaled together?
- Can a single node meet their combined resource needs?

If the answer to all these questions is yes, put them all in the same pod. As a rule of thumb, always place containers in separate pods unless a specific reason requires them to be part of the same pod.

5.2 Creating pods from YAML or JSON files

With the information you learned in the previous sections, you can now start creating pods. In chapter 3, you created them using the imperative command `kubectl create`, but pods and other Kubernetes objects are usually created by creating a JSON or YAML manifest file and posting it to the Kubernetes API, as you've already learned in the previous chapter.

> **NOTE** The decision whether to use YAML or JSON to define your objects is yours. Most people prefer to use YAML because it's slightly more human-friendly and allows you to add comments to the object definition.

By using YAML files to define the structure of your application, you don't need shell scripts to make the process of deploying your applications repeatable, and you can keep a history of all changes by storing these files in a VCS (Version Control System). Just like you store code.

In fact, the application manifests of the exercises in this book are all stored in a VCS. You can find them on GitHub at github.com/luksa/kubernetes-in-action-2ed.

5.2.1 Creating a YAML manifest for a pod

In the previous chapter you learned how to retrieve and examine the YAML manifests of existing API objects. Now you'll create an object manifest from scratch.

You'll start by creating a file called `kubia.yaml` on your computer, in a file directory of your choice. You can also find the file in the book's code archive, available on GitHub. The file is in the `Chapter04/` directory. The following listing shows its contents.

边车容器的其他示例包括日志旋转器和收集器、数据处理器、通信适配器等。

与更改应用程序的现有代码不同，添加 sidecar 会增加 pod 的资源需求，因为 pod 中必须运行额外的进程。但请记住，向遗留应用程序添加代码可能非常困难。这可能是因为它的代码很难修改，很难设置构建环境，或者源代码本身不再可用。通过添加额外的进程来扩展应用程序有时是一种更便宜且更快的选择。

如何决定是否将容器拆分为多个 Pod

在决定是使用 sidecar 模式并将容器放置在单个 Pod 中，还是将它们放置在单独的 Pod 中时，请问自己以下问题：

- 这些容器是否必须在同一主机上运行?
- 我想将它们作为一个整体进行管理吗?
- 它们是否形成一个统一的整体而不是独立的组成部分?
- 它们是否必须一起缩放?

如果所有这些问题的答案都是肯定的，请将它们全部放在同一个容器中。根据经验，应始终将容器放置在单独的 Pod 中，除非出于特定原因要求它们属于同一 Pod。

5.2 从 YAML 或 JSON 文件创建 pod

根据您在前面部分中学到的信息，您现在可以开始创建 Pod。在第 3 章中，您使用命令式命令 kubectl create 创建了它们，但 Pod 和其他 Kubernetes 对象通常是通过创建 JSON 或 YAML 清单文件并将其发布到 Kubernetes API 来创建的，正如您在上一章中已经了解的那样。

> 注意 使用 YAML 还是 JSON 来定义对象的决定权在于您。大多数人更喜欢使用 YAML，因为它更加人性化，并且允许您向对象定义添加注释。

通过使用 YAML 文件定义应用程序的结构，您不需要 shell 脚本来使应用程序的部署过程可重复，并且您可以通过将这些文件存储在 VCS（版本控制系统）中来保留所有更改的历史记录。就像你存储代码一样。

事实上，本书中练习的应用清单都存储在VCS中。你可以在 GitHub 上找到它们：github.com/luksa/kubernetes-in-action-2ed。

5.2.1 为 pod 创建 YAML 清单

在上一章中，您学习了如何检索和检查现有 API 对象的 YAML 清单。现在您将从头开始创建一个对象清单。

首先，您将在计算机上选择的文件目录中创建一个名为 kubia.yaml 的文件。您还可以在本书的代码存档（可在 GitHub 上找到）中找到该文件。该文件位于 Chapter04/ 目录中。以下清单显示了其内容。

Listing 5.1 A basic pod manifest: kubia.yaml

```
apiVersion: v1                #A
kind: Pod                     #B
metadata:
  name: kubia                 #C
spec:
  containers:
  - name: kubia               #D
    image: luksa/kubia:1.0    #E
    ports:
    - containerPort: 8080     #F
```

#A This manifest uses the v1 API version to define the object
#B The object specified in this manifest is a pod
#C The name of the pod
#D The name of the container
#E Container image to create the container from
#F The port the app is listening on

I'm sure you'll agree that this pod manifest is much easier to understand than the mammoth of a manifest representing the Node object, which you saw in the previous chapter. But once you post this pod object manifest to the API and then read it back, it won't be much different.

The manifest in listing 5.1 is short only because it does not yet contain all the fields that a pod object gets after it is created through the API. For example, you'll notice that the `metadata` section contains only a single field and that the `status` section is completely missing. Once you create the object from this manifest, this will no longer be the case. But we'll get to that later.

Before you create the object, let's examine the manifest in detail. It uses version `v1` of the Kubernetes API to describe the object. The object is a `Pod` called `kubia`. The pod consists of a single container called `kubia`, based on the `luksa/kubia:1.0` image. The pod definition also specifies that the application in the container listens on port `8080`.

> **TIP** Whenever you want to create a pod manifest from scratch, you can also use the following command to create the file and then edit it to add more fields: `kubectl run kubia --image=luksa/kubia:1.0 --dry-run=client -o yaml > mypod.yaml`. The `--dry-run=client` flag tells kubectl to output the definition instead of actually creating the object via the API.

The fields in the YAML file are self-explanatory, but if you want more information about each field or want to know what additional fields you can add, remember to use the `kubectl explain pods` command.

5.2.2 Creating the Pod object from the YAML file

After you've prepared the manifest file for your pod, you can now create the object by posting the file to the Kubernetes API.

清单 5.1 基本的 pod 清单：kubia.yaml

```
api版本：v1                #A
种类：豆荚                 #B
元数据：
  姓名：库比亚             #C
规格：
  容器：  -
  名称：                   #D
    图片：luksa/kubia:1.0 #E
  kubia
    端口：
    - 集装箱端口：8080     #F
```

#A 此清单使用 v1 API 版本来定义对象 #B 此清单中指定的对象是一个 pod #C pod 的名称 #D 容器的名称

#E 用于从 #F 应用程序正在侦听的端口创建容器的容器映像

我相信您会同意这个 pod 清单比您在上一章中看到的代表 Node 对象的庞大清单更容易理解。但是，一旦您将此 pod 对象清单发布到 API，然后将其读回，情况就不会有太大不同。

清单 5.1 中的清单很短，只是因为它尚未包含 pod 对象通过 API 创建后获取的所有字段。例如，您会注意到元数据部分仅包含一个字段，而状态部分完全包含

丢失的。一旦您从此清单创建对象，情况将不再如此。但我们稍后会讨论这个问题。

在创建对象之前，让我们详细检查一下清单。它使用 Kubernetes API v1 版本来描述对象。该对象是一个名为 kubia 的 Pod。该吊舱由

名为 kubia 的单个容器，基于 luksa/kubia:1.0 镜像。 Pod 定义还指定容器中的应用程序侦听端口 8080。

> 提示每当您想要从头开始创建 pod 清单时，您还可以使用以下命令创建文件，然后编辑它以添加更多字段： kubectl run kubia --image=luksa/kubia:1.0 --dry-run=client -o yaml > mypod.yaml。 --dry-run=client 标志告诉 kubectl 输出定义而不是通过 API 实际创建对象

YAML 文件中的字段是不言自明的，但如果您想了解有关每个字段的更多信息或想知道可以添加哪些其他字段，请记住使用 kubectlexplain pods 命令。

5.2.2 从YAML文件创建Pod对象

为 pod 准备好清单文件后，您现在可以通过将文件发布到 Kubernetes API 来创建对象。

CREATING OBJECTS BY APPLYING THE MANIFEST FILE TO THE CLUSTER

When you post the manifest to the API, you are directing Kubernetes to *apply* the manifest to the cluster. That's why the `kubectl` sub-command that does this is called `apply`. Let's use it to create the pod:

```
$ kubectl apply -f kubia.yaml
pod "kubia" created
```

UPDATING OBJECTS BY MODIFYING THE MANIFEST FILE AND RE-APPLYING IT

The `kubectl apply` command is used for creating objects as well as for making changes to existing objects. If you later decide to make changes to your pod object, you can simply edit the `kubia.yaml` file and run the apply command again. Some of the pod's fields aren't mutable, so the update may fail, but you can always delete the pod and then create it again. You'll learn how to delete pods and other objects at the end of this chapter.

Retrieving the full manifest of a running pod

The pod object is now part of the cluster configuration. You can now read it back from the API to see the full object manifest with the following command:

```
$ kubectl get po kubia -o yaml
```

If you run this command, you'll notice that the manifest has grown considerably compared to the one in the `kubia.yaml` file. You'll see that the `metadata` section is now much bigger, and the object now has a `status` section. The `spec` section has also grown by several fields. You can use `kubectl explain` to learn more about these new fields, but most of them will be explained in this and the following chapters.

5.2.3 Checking the newly created pod

Let's use the basic `kubectl` commands to see how the pod is doing before we start interacting with the application running inside it.

QUICKLY CHECKING THE STATUS OF A POD

Your Pod object has been created, but how do you know if the container in the pod is actually running? You can use the `kubectl get` command to see a summary of the pod:

```
$ kubectl get pod kubia
NAME      READY   STATUS    RESTARTS   AGE
kubia     1/1     Running   0          32s
```

You can see that the pod is running, but not much else. To see more, you can try the `kubectl get pod -o wide` or the `kubectl describe` command that you learned in the previous chapter.

USING KUBECTL DESCRIBE TO SEE POD DETAILS

To display a more detailed view of the pod, use the `kubectl describe` command:

通过将清单文件应用到集群来创建对象

当您将清单发布到 API 时，您是在指示 Kubernetes 将清单应用到集群。这就是为什么执行此操作的 kubectl 子命令称为 apply。让我们用它来创建 pod：

```
$ kubectl apply -f
kubia.yaml pod "kubia" 已
创建
```

通过修改清单文件并重新应用它来更新对象

kubectl apply 命令用于创建对象以及对现有对象进行更改。如果您稍后决定对 pod 对象进行更改，您只需编辑 kubia.yaml 文件并再次运行 apply 命令即可。 Pod 的某些字段不可变，因此更新可能会失败，但您始终可以删除 Pod，然后重新创建它。在本章末尾，您将学习如何删除 Pod 和其他对象。

检索正在运行的 Pod 的完整清单

Pod 对象现在是集群配置的一部分。现在，您可以使用以下命令从 API 读回它以查看完整的对象清单：

```
$ kubectl 获取 po kubia -o yaml
```

如果运行此命令，您会注意到清单与清单中的清单相比已显着增长

kubia.yaml 文件。您会看到元数据部分现在更大了，并且对象现在具有状态

部分。规格部分也增加了多个领域。您可以使用 kubectlexplain 来了解有关这些新字段的更多信息，但其中大部分将在本章和后续章节中进行解释

5.2.3 检查新创建的Pod

在开始与 pod 中运行的应用程序交互之前，让我们使用基本的 kubectl 命令来查看 pod 的运行情况。

快速查看Pod状态

你的 Pod 对象已经创建了，但是你如何知道 pod 中的容器是否真正在运行呢？您可以使用 kubectl get 命令查看 pod 的摘要：

```
$ kubectl get pod kubia 名称 就绪 状态
重新启动年龄
库比亚 1/1 跑步 0 32秒
```

您可以看到 Pod 正在运行，但除此之外就没有其他信息了。要查看更多信息，您可以尝试 kubectl get pod -o Wide 或您在

前一章。
使用 KUBECTL DESCRIBE 查看 Pod 详细信息

显示 Pod 的更详细视图

Listing 5.2 Using kubectl describe pod to inspect a pod

```
$ kubectl describe pod kubia
Name:         kubia
Namespace:    default
Priority:     0
Node:         worker2/172.18.0.4
Start Time:   Mon, 27 Jan 2020 12:53:28 +0100
...
```

The listing doesn't show the entire output, but if you run the command yourself, you'll see virtually all information that you'd see if you print the complete object manifest using the `kubectl get -o yaml` command.

INSPECTING EVENTS TO SEE WHAT HAPPENS BENEATH THE SURFACE

As in the previous chapter where you used the `describe node` command to inspect a Node object, the `describe pod` command should display several events related to the pod at the bottom of the output.

If you remember, these events aren't part of the object itself, but are separate objects. Let's print them to learn more about what happens when you create the pod object. The following listing shows all the events that were logged after creating the pod.

Listing 5.3 Events recorded after deploying a Pod object

```
$ kubectl get events
LAST SEEN   TYPE     REASON      OBJECT      MESSAGE
<unknown>   Normal   Scheduled   pod/kubia   Successfully assigned default/
                                             kubia to worker2
5m          Normal   Pulling     pod/kubia   Pulling image luksa/kubia:1.0
5m          Normal   Pulled      pod/kubia   Successfully pulled image
5m          Normal   Created     pod/kubia   Created container kubia
5m          Normal   Started     pod/kubia   Started container kubia
```

These events are printed in chronological order. The most recent event is at the bottom. You see that the pod was first assigned to one of the worker nodes, then the container image was pulled, then the container was created and finally started.

No warning events are displayed, so everything seems to be fine. If this is not the case in your cluster, you should read section 5.4 to learn how to troubleshoot pod failures.

5.3 Interacting with the application and the pod

Your container is now running. In this section, you'll learn how to communicate with the application, inspect its logs, and execute commands in the container to explore the application's environment. Let's confirm that the application running in the container responds to your requests.

5.3.1 Sending requests to the application in the pod

In chapter 2, you used the `kubectl expose` command to create a service that provisioned a load balancer so you could talk to the application running in your pod(s). You'll now take a

清单 5.2 使用 kubectl describe pod 来检查 pod

```
$ kubectl 描述 pod kubia
名称：         库比亚
命名空间：     默认
优先事项：     0
节点：         工人2/172.18.0.4
开始时间：     2020 年 1 月 27 日星期一 12:53:28 +0100
...
```

该列表并未显示完整的输出，但如果您自己运行该命令，您几乎会看到使用 kubectl get -o yaml 命令打印完整对象清单时看到的所有信息。

检查事件以了解表面下发生的情况

正如在上一章中使用describe node 命令检查 Node 对象一样，describe pod 命令应在输出底部显示与 pod 相关的多个事件。

如果您还记得的话，这些事件不是对象本身的一部分，而是单独的对象。让我们打印它们以了解有关创建 pod 对象时会发生什么的更多信息。以下列表显示了创建 Pod 后记录的所有事件。

清单 5.3 部署 Pod 对象后记录的事件

```
$ kubectl get events LAST SEEN TYPE REASON OBJECT MESSAGE
<未知> 正常调度的 pod/kubia 已成功将 default/ kubia 分配给worker2
5m 正常 拉 pod/kubia 拉图像 luksa/kubia:1.0
5m 正常拉取pod/kubia 成功拉取镜像
5m 正常 创建 pod/kubia 创建容器 kubia
```

这些事件按时间顺序打印。最近的事件位于底部。您会看到，pod 首先被分配给其中一个工作节点，然后拉取容器映像，然后创建容器，最后启动。

没有显示警告事件，所以一切似乎都很好。如果您的集群不是这种情况，您应该阅读第 5.4 节以了解如何排除 pod 故障。

5.3 与应用程序和 Pod 交互

您的容器现在正在运行。在本节中，您将学习如何与应用程序通信、检查其日志以及在容器中执行命令以探索应用程序的环境。让我们确认容器中运行的应用程序响应您的请求。

5.3.1 向Pod中的应用程序发送请求

在第 2 章中，您使用 kubectl公开命令创建了一个配置负载均衡器的服务，以便您可以与 Pod 中运行的应用程序进行通信

different approach. For development, testing and debugging purposes, you may want to communicate directly with a specific pod, rather than using a service that forwards connections to randomly selected pods.

You've learned that each pod is assigned its own IP address where it can be accessed by every other pod in the cluster. This IP address is typically internal to the cluster. You can't access it from your local computer, except when Kubernetes is deployed in a specific way – for example, when using kind or Minikube without a VM to create the cluster.

In general, to access pods, you must use one of the methods described in the following sections. First, let's determine the pod's IP address.

GETTING THE POD'S IP ADDRESS

You can get the pod's IP address by retrieving the pod's full YAML and searching for the `podIP` field in the `status` section. Alternatively, you can display the IP with `kubectl describe`, but the easiest way is to use `kubectl get` with the `wide` output option:

```
$ kubectl get pod kubia -o wide
NAME    READY   STATUS    RESTARTS   AGE   IP           NODE     ...
kubia   1/1     Running   0          35m   10.244.2.4   worker2  ...
```

As indicated in the IP column, my pod's IP is `10.244.2.4`. Now I need to determine the port number the application is listening on.

GETTING THE PORT THE APPLICATION IS BOUND TO

If I wasn't the author of the application, it would be difficult for me to find out which port the application listens on. I could inspect its source code or the Dockerfile of the container image, as the port is usually specified there, but I might not have access to either. If someone else had created the pod, how would I know which port it was listening on?

Fortunately, you can specify a list of ports in the pod definition itself. It isn't necessary to specify any ports, but it is a good idea to always do so. See sidebar for details.

Why specify container ports in pod definitions

Specifying ports in the pod definition is purely informative. Their omission has no effect on whether clients can connect to the pod's port. If the container accepts connections through a port bound to its IP address, anyone can connect to it, even if the port isn't explicitly specified in the pod spec or if you specify an incorrect port number. Despite this, it's a good idea to always specify the ports so that anyone who has access to your cluster can see which ports each pod exposes. By explicitly defining ports, you can also assign a name to each port, which is very useful when you expose pods via services.

The pod manifest says that the container uses port 8080, so you now have everything you need to talk to the application.

CONNECTING TO THE POD FROM THE WORKER NODES

The Kubernetes network model dictates that each pod is accessible from any other pod and that each *node* can reach any pod on any node in the cluster.

119

不同的方法。出于开发、测试和调试目的，您可能希望直接与特定 Pod 通信，而不是使用将连接转发到随机选择的 Pod 的服务。

您已经了解到，每个 Pod 都分配有自己的 IP 地址，集群中的每个其他 Pod 都可以访问它。此 IP 地址通常位于集群内部。您无法从本地计算机访问它，除非 Kubernetes 以特定方式部署 - 例如，在没有 VM 的情况下使用 kind 或 Minikube 创建集群时。

一般来说，要访问 pod，您必须使用以下描述的方法之一
部分。首先，我们确定 pod 的 IP 地址。

获取 Pod 的 IP 地址

您可以通过检索 pod 的完整 YAML 并在状态部分中搜索 podIP 字段来获取 pod 的 IP 地址。或者，您可以使用 kubectl 显示 IP

描述，但最简单的方法是使用 kubectl get 和宽输出选项：

```
$ kubectl get pod kubia -o Wide 名称就绪状态重新启动年龄 IP
节点                                                    ...
库比亚 1/1 跑步         0 35m 10.244.2.4 工人2          ...
```

如 IP 列所示，我的 Pod 的 IP 是 10.244.2.4。现在我需要确定应用程序正在侦听的端口号。

获取应用程序绑定的端口

如果我不是应用程序的作者，我将很难找出应用程序侦听哪个端口。我可以检查其源代码或容器映像的 Dockerfile，因为端口通常在那里指定，但我可能无法访问其中任何一个。如果其他人创建了 pod，我怎么知道它正在侦听哪个端口？

幸运的是，您可以在 pod 定义本身中指定端口列表。没有必要
指定任何端口，但最好始终这样做。有关详细信息，请参阅侧边栏。

为什么在 Pod 定义中指定容器端口 在 Pod 定义中指定端口纯粹是为了提供信息。它们的省略不会影响客户端是否可以连接到 Pod 的端口。如果容器通过绑定到其 IP 地址的端口接受连接，则任何人都可以连接到它，即使 pod 规范中没有明确指定该端口或者您指定了不正确的端口号。尽管如此，最好始终指定端口，以便有权访问集群的任何人都可以看到每个 Pod 公开的端口。通过显式定义端口，您还可以为每个端口分配一个名称，这在您通过服务公开 pod 时非常有用。

Pod 清单表明容器使用端口 8080，因此您现在拥有与应用程序通信所需的一切。

从工作节点连接到 Pod

Kubernetes 网络模型规定每个 Pod 都可以从任何其他 Pod 访问，并且每个节点都可以访问集群中任何节点上的任何 Pod。

Because of this, one way to communicate with your pod is to log into one of your worker nodes and talk to the pod from there. You've already learned that the way you log on to a node depends on what you used to deploy your cluster. In Minikube, you can run `minikube ssh` to log in to your single node. On GKE use the `gcloud compute ssh` command. For other clusters refer to their documentation.

Once you have logged into the node, use the `curl` command with the pod's IP and port to access your application. My pod's IP is 10.244.2.4 and the port is 8080, so I run the following command:

```
$ curl 10.244.2.4:8080
Hey there, this is kubia. Your IP is ::ffff:10.244.2.1.
```

Normally you don't use this method to talk to your pods, but you may need to use it if there are communication issues and you want to find the cause by first trying the shortest possible communication route. In this case, it's best to log into the node where the pod is located and run `curl` from there. The communication between it and the pod takes place locally, so this method always has the highest chances of success.

CONNECTING FROM A ONE-OFF CLIENT POD

The second way to test the connectivity of your application is to run `curl` in another pod that you create specifically for this task. Use this method to test if other pods will be able to access your pod. Even if the network works perfectly, this may not be the case. In chapter 24, you'll learn how to lock down the network by isolating pods from each other. In such a system, a pod can only talk to the pods it's allowed to.

To run `curl` in a one-off pod, use the following command:

```
$ kubectl run --image=tutum/curl -it --restart=Never --rm client-pod curl
10.244.2.4:8080
Hey there, this is kubia. Your IP is ::ffff:10.244.2.5.
pod "client-pod" deleted
```

This command runs a pod with a single container created from the `tutum/curl` image. You can also use any other image that provides the `curl` binary executable. The `-it` option attaches your console to the container's standard input and output, the `--restart=Never` option ensures that the pod is considered Completed when the `curl` command and its container terminate, and the `--rm` options removes the pod at the end. The name of the pod is `client-pod` and the command executed in its container is `curl 10.244.2.4:8080`.

> **NOTE** You can also modify the command to run the `bash` shell in the client pod and then run `curl` from the shell.

Creating a pod just to see if it can access another pod is useful when you're specifically testing pod-to-pod connectivity. If you only want to know if your pod is responding to requests, you can also use the method explained in the next section.

因此，与 Pod 通信的一种方法是登录到其中一个工作节点并从那里与 Pod 进行通信。您已经了解到，登录节点的方式取决于您用于部署集群的方式。在Minikube中，您可以运行 minikube ssh来登录到您的单节点。在 GKE 上使用 gcloudcompute ssh 命令。对于他人

集群请参阅其文档。

登录节点后，使用带有 pod 的 IP 和端口的curl 命令来访问您的应用程序。我的 Pod 的 IP 是 10.244.2.4，端口是 8080，所以我运行以下命令：

```
$curl
10.244.2.4:8080 嘿，
这是 kubia。您的 IP
是：：ffff:10.244.2.1。
```

通常，您不会使用此方法与 pod 进行通信，但如果出现通信问题并且您希望通过首先尝试最短的通信路线来查找原因，则可能需要使用它。在这种情况下，最好登录 pod 所在的节点并从那里运行curl。它和 Pod 之间的通信在本地进行，因此这种方法总是有最高的成功机会。

从一次性客户端 Pod 进行连接

测试应用程序连接性的第二种方法是在专门为此任务创建的另一个 Pod 中运行curl。使用此方法测试其他 pod 是否能够访问您的 pod。即使网络运行完美，情况也可能并非如此。在第 24 章中，您将学习如何通过将 Pod 彼此隔离来锁定网络。在这样的系统中，Pod 只能与它允许的 Pod 进行通信。

要在一次性 pod 中运行curl，请使用以下命令：

```
$ kubectl run --image=tutum/curl -it --restart=Never --rm client-
pod curl 10.244.2.4:8080 嘿，这里是 kubia。您的 IP
是::ffff:10.244.2.5。 pod“client-pod”已删除
```

此命令运行一个带有从 tutum/curl 镜像创建的单个容器的 pod。您还可以使用提供curl 二进制可执行文件的任何其他映像。 -it 选项

将控制台附加到容器的标准输入和输出，--restart=Never 选项确保当curl 命令及其容器终止时，pod 被视为已完成，--rm 选项在最后删除 pod。 Pod 的名称为 client-pod，在其容器中执行的命令为curl 10.244.2.4:8080。

注意 您还可以删除 pod 以在客户端 pod 中运行 bash shell

当您专门测试 Pod 到 Pod 的连接时，创建一个 Pod 来查看它是否可以访问另一个 Pod 非常有用。如果您只想知道您的 pod 是否正在响应请求，您也可以使用下一节中介绍的方法。

CONNECTING TO PODS VIA KUBECTL PORT FORWARDING

During development, the easiest way to talk to applications running in your pods is to use the `kubectl port-forward` command, which allows you to communicate with a specific pod through a proxy bound to a network port on your local computer, as shown in the next figure.

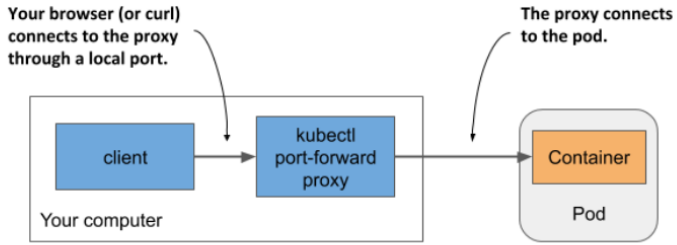


Figure 5.8 Connecting to a pod through the kubectl port-forward proxy

To open a communication path with a pod, you don't even need to look up the pod's IP, as you only need to specify its name and the port. The following command starts a proxy that forwards your computer's local port `8080` to the `kubia` pod's port `8080`:

```
$ kubectl port-forward kubia 8080
... Forwarding from 127.0.0.1:8080 -> 8080
... Forwarding from [::1]:8080 -> 8080
```

The proxy now waits for incoming connections. Run the following `curl` command in another terminal:

```
$ curl localhost:8080
Hey there, this is kubia. Your IP is ::ffff:127.0.0.1.
```

As you can see, `curl` has connected to the local proxy and received the response from the pod. While the `port-forward` command is the easiest method for communicating with a specific pod during development and troubleshooting, it's also the most complex method in terms of what happens underneath. Communication passes through several components, so if anything is broken in the communication path, you won't be able to talk to the pod, even if the pod itself is accessible via regular communication channels.

NOTE The `kubectl port-forward` command can also forward connections to services instead of pods and has several other useful features. Run `kubectl port-forward --help` to learn more.

121

通过 KUBECTL 端口转发连接到 Pod

在开发过程中，与 Pod 中运行的应用程序通信的最简单方法是使用 kubectl port-forward 命令，该命令允许您通过绑定到本地计算机上网络端口的代理与特定 Pod 进行通信，如下所示下图。

图 5.8 通过 kubectl port-forward 代理连接到 pod

要打开与 Pod 的通信路径，您甚至不需要查找 Pod 的 IP，因为您只需要指定其名称和端口。以下命令启动一个代理，将计算机的本地端口 8080 转发到 kubia pod 的端口 8080：

```
$ kubectl port-forward kubia 8080
... 从 127.0.0.1 :8080 -> 8080 转
发
```

代理现在等待传入连接。在另一个终端中运行以下curl命令：

```
$curl
localhost :8080 嘿，
这是 kubia。您的 IP
是 ::ffff:127.0.0.1。
```

如您所见，curl已连接到本地代理并收到了来自 pod 的响应。虽然 port-forward 命令是在开发和故障排除期间与特定 Pod 通信的最简单方法，但就底层发生的情况而言，它也是最复杂的方法。通信通过多个组件，因此如果通信路径中出现任何问题，您将无法与 Pod 通信，即使 Pod 本身可以通过常规通信通道进行访问。

注意 kubectl port-forward 命令还可以将连接转发到服务而不是 pod，并且具有其他一些有用的功能。运行 kubectl port-forward --help 以了解更多信息。

Figure 5.9 shows how the network packets flow from the `curl` process to your application and back.

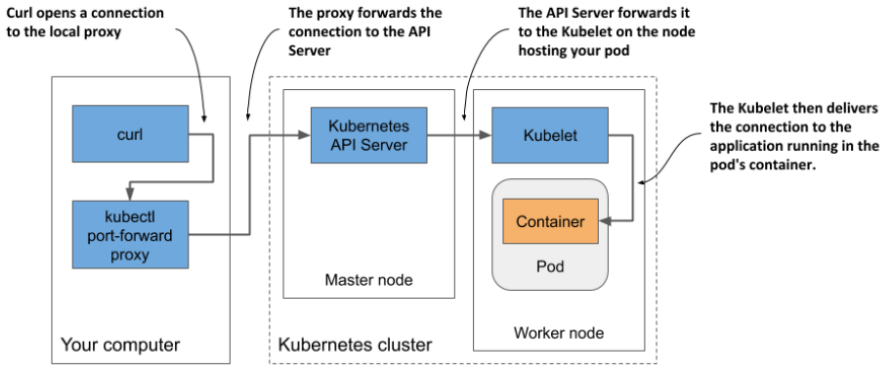


Figure 5.9 The long communication path between curl and the container when using port forwarding

As shown in the figure, the `curl` process connects to the proxy, which connects to the API server, which then connects to the Kubelet on the node that hosts the pod, and the Kubelet then connects to the container through the pod's loopback device (in other words, through the localhost address). I'm sure you'll agree that the communication path is exceptionally long.

> **NOTE** The application in the container must be bound to a port on the loopback device for the Kubelet to reach it. If it listens only on the pod's `eth0` network interface, you won't be able to reach it with the `kubectl port-forward` command.

5.3.2 Viewing application logs

Your Node.js application writes its log to the standard output stream. Instead of writing the log to a file, containerized applications usually log to the standard output (*stdout*) and standard error streams (*stderr*). This allows the container runtime to intercept the output, store it in a consistent location (usually `/var/log/containers`) and provide access to the log without having to know where each application stores its log files.

When you run an application in a container using Docker, you can display its log with `docker logs <container-id>`. When you run your application in Kubernetes, you could log into the node that hosts the pod and display its log using `docker logs`, but Kubernetes provides an easier way to do this with the `kubectl logs` command.

图 5.9 显示了网络数据包如何从curl 进程流向您的应用程序并返回。

图5.9 使用端口转发时curl与容器之间的长通信路径

如图所示，curl进程连接到代理，代理连接到API服务器，然后API服务器连接到托管pod的节点上的Kubelet，然后Kubelet通过pod的环回设备连接到容器（在换句话说，通过本地主机地址）。我相信你会同意沟通路径非常长。

> 注意容器中的应用程序必须绑定到环回设备上的端口，Kubelet 才能访问它。如果它仅侦听 pod 的 eth0 网络接口，您将无法使用
> kubectl 端口转发命令。

5.3.2 查看应用程序日志

您的 Node.js 应用程序将其日志写入标准输出流。容器化应用程序通常不会将日志写入文件，而是将日志记录到标准输出 (stdout) 和标准错误流 (stderr)。这允许容器运行时拦截输出，将其存储在一致的位置（通常是 /var/log/containers）并提供对日志的访问，而无需知道每个应用程序存储其日志文件的位置。

当您使用 Docker 在容器中运行应用程序时，可以使用 docker log 显示其日志。当您在 Kubernetes 中运行应用程序时，您可以记录

进入托管 pod 的节点并使用 docker log 显示其日志，但 Kubernetes

使用 kubectl logs 命令提供了一种更简单的方法来执行此操作。

RETRIEVING A POD'S LOG WITH KUBECTL LOGS

To view the log of your pod (more specifically, the container's log), run the command shown in the following listing on your local computer:

Listing 5.4 Displaying a pod's log

```
$ kubectl logs kubia
Kubia server starting...
Local hostname is kubia
Listening on port 8080
Received request for / from ::ffff:10.244.2.1    #A
Received request for / from ::ffff:10.244.2.5    #B
Received request for / from ::ffff:127.0.0.1     #C
```

#A Request you sent from within the node
#B Request from the one-off client pod
#C Request sent through port forwarding

STREAMING LOGS USING KUBECTL LOGS -F

If you want to stream the application log in real-time to see each request as it comes in, you can run the command with the `--follow` option (or the shorter version `-f`):

```
$ kubectl logs kubia -f
```

Now send some additional requests to the application and have a look at the log. Press ctrl-C to stop streaming the log when you're done.

DISPLAYING THE TIMESTAMP OF EACH LOGGED LINE

You may have noticed that we forgot to include the timestamp in the log statement. Logs without timestamps have limited usability. Fortunately, the container runtime attaches the current timestamp to every line produced by the application. You can display these timestamps by using the `--timestamps=true` option, as shown in the next listing.

Listing 5.5 Displaying the timestamp of each log line

```
$ kubectl logs kubia --timestamps=true
2020-02-01T09:44:40.954641934Z Kubia server starting...
2020-02-01T09:44:40.955123432Z Local hostname is kubia
2020-02-01T09:44:40.956435431Z Listening on port 8080
2020-02-01T09:50:04.9780430897Z Received request for / from ...
2020-02-01T09:50:33.6408973787Z Received request for / from ...
2020-02-01T09:50:44.7814732567Z Received request for / from ...
```

TIP You can display timestamps by only typing `--timestamps` without the value. For boolean options, merely specifying the option name sets the option to `true`. This applies to all kubectl options that take a Boolean value and default to `false`.

使用 KUBECTL LOGS 检索 Pod 的日志

要查看 pod 的日志（更具体地说，容器的日志），请在本地计算机上运行以下列表中所示的命令：

清单 5.4 显示 pod 的日志

```
$ kubectl 记录
kubia Kubia 服务器
正在启动...本地主
机名是 kubia 正在
收到来自 ::ffff:10.244.2.1 的 / 请求        #A
收到来自 ::ffff:10.244.2.5 的 / 请求        #B
收到来自 ::ffff:127.0.0.1 的 / 请求         #C
```

#A 从节点内部发送的请求 #B 来自
一次性客户端 Pod 的请求 #C 通过
端口转发发送的请求

使用 KUBECTL LOGS -F 流式传输日志

如果您想实时流式传输应用程序日志以查看每个请求的传入，您可以使用 --follow 选项（或更短的版本 -f）运行该命令：

```
$ kubectl 日志 kubia -f
```

现在向应用程序发送一些额外的请求并查看日志。完成后按 ctrl-C 停止流式传输日志。

显示每条记录行的时间戳

您可能已经注意到，我们忘记在日志语句中包含时间戳。没有时间戳的日志可用性有限。幸运的是，容器运行时将当前时间戳附加到应用程序生成的每一行。您可以使用 --timestamps=true 选项显示这些时间戳，如下一个清单所示。

清单 5.5 显示每个日志行的时间戳

```
$ kubectl 记录 kubia --
timestamps=true 2020-02-
01T09:44:40.954641934Z Kubia 服务
器正在启动... 2020-02-
01T09:44:40.955123432Z 本地主机名
是 kubia 2020-02-
01T09:44:40.956435431Z 听力在端口
```

提示 您可以通过仅键入 --timestamps 而不输入值来显示时间戳。对于布尔选项，只需指定选项名称即可将选项设置为 true。这适用于所有采用
布尔值，默认为 false。

DISPLAYING RECENT LOGS

The previous feature is great if you run third-party applications that don't include the timestamp in their log output, but the fact that each line is timestamped brings us another benefit: filtering log lines by time. Kubectl provides two ways of filtering the logs by time.

The first option is when you want to only display logs from the past several seconds, minutes or hours. For example, to see the logs produced in the last two minutes, run:

```
$ kubectl logs kubia --since=2m
```

The other option is to display logs produced after a specific date and time using the `--since-time` option. The time format to be used is RFC3339. For example, the following command is used to print logs produced after February 1st, 2020 at 9:50 a.m.:

```
$ kubectl logs kubia --since-time=2020-02-01T09:50:00Z
```

DISPLAYING THE LAST SEVERAL LINES OF THE LOG

Instead of using time to constrain the output, you can also specify how many lines from the end of the log you want to display. To display the last ten lines, try:

```
$ kubectl logs kubia --tail=10
```

NOTE Kubectl options that take a value can be specified with an equal sign or with a space. Instead of `--tail=10`, you can also type `--tail 10`.

UNDERSTANDING THE AVAILABILITY OF THE POD'S LOGS

Kubernetes keeps a separate log file for each container. They are usually stored in `/var/log/containers` on the node that runs the container. A separate file is created for each container. If the container is restarted, its logs are written to a new file. Because of this, if the container is restarted while you're following its log with `kubectl logs -f`, the command will terminate, and you'll need to run it again to stream the new container's logs.

The `kubectl logs` command displays only the logs of the current container. To view the logs from the previous container, use the `--previous` (or `-p`) option.

NOTE Depending on your cluster configuration, the log files may also be rotated when they reach a certain size. In this case, `kubectl logs` will only display the new log file. When streaming the logs, you must restart the command to switch to the new file.

When you delete a pod, all its log files are also deleted. To make pods' logs available permanently, you need to set up a central, cluster-wide logging system. Chapter 23 explains how.

WHAT ABOUT APPLICATIONS THAT WRITE THEIR LOGS TO FILES?

If your application writes its logs to a file instead of stdout, you may be wondering how to access that file. Ideally, you'd configure the centralized logging system to collect the logs so you can view them in a central location, but sometimes you just want to keep things simple

显示最近的日志

如果您运行的第三方应用程序在日志输出中不包含时间戳，则前面的功能非常有用，但每行都带有时间戳的事实给我们带来了另一个好处：按时间过滤日志行。 Kubectl 提供了两种按时间过滤日志的方法。

第一个选项是当您只想显示过去几秒、几分钟或几小时的日志时。例如，要查看最后两分钟内生成的日志，请运行：

```
$ kubectl 日志 kubia --since=2m
```

另一个选项是使用 `-since-time` 选项显示在特定日期和时间之后生成的日志。要使用的时间格式是 RFC3339。例如，以下

命令用于打印2020年2月1日上午9点50分之后产生的日志：

```
$ kubectl 记录 kubia --since-time=2020-02-01T09:50:00Z
```

显示日志的最后几行

您还可以指定要显示的日志末尾的行数，而不是使用时间来限制输出。要显示最后十行，请尝试：

```
$ kubectl 日志 kubia --tail=10
```

注意 可以使用等号或空格来指定采用值的 Kubectl 选项。您还可以键入 `--tail 10`，而不是 `-`

了解 Pod 日志的可用性

Kubernetes 为每个容器保留一个单独的日志文件。它们通常存储在运行容器的节点上的 /var/log/containers 中。为以下内容创建一个单独的文件

每个容器。如果容器重新启动，其日志将写入新文件。因此，如果在您使用 kubectl log -f 跟踪其日志时容器重新启动，该命令将终止，您需要再次运行它以流式传输新容器的日志。

kubectl messages命令仅显示当前容器的日志。要查看来自上一个容器的日志，请使用 --previous （或 -p）选项。

注意 根据您的集群配置，日志文件在达到一定大小时也可能会轮换。在这种情况下，kubectl log 将仅显示新的日志文件。流式传输日志时，您必须重新启动命令以切换到新文件。

当您删除 Pod 时，它的所有日志文件也会被删除。为了使 Pod 的日志永久可用，您需要设置一个中央的、集群范围的日志系统。第 23 章解释了如何进行。

将日志写入文件的应用程序怎么样？

如果您的应用程序将其日志写入文件而不是 stdout，您可能想知道如何访问该文件。理想情况下，您应该配置集中式日志记录系统来收集日志，以便您可以在中央位置查看它们

and don't mind accessing the logs manually. In the next two sections, you'll learn how to copy log and other files from the container to your computer and in the opposite direction, and how to run commands in running containers. You can use either method to display the log files or any other file inside the container.

5.3.3 Copying files to and from containers

Sometimes you may want to add a file to a running container or retrieve a file from it. Modifying files in running containers isn't something you normally do - at least not in production - but it can be useful during development.

Kubectl offers the `cp` command to copy files or directories from your local computer to a container of any pod or from the container to your computer. For example, to copy the `/etc/hosts` file from the container of the `kubia` pod to the `/tmp` directory on your local file system, run the following command:

```
$ kubectl cp kubia:/etc/hosts /tmp/kubia-hosts
```

To copy a file from your local file system to the container, run the following command:

```
$ kubectl cp /path/to/local/file kubia:path/in/container
```

NOTE The `kubectl cp` command requires the `tar` binary to be present in your container, but this requirement may change in the future.

5.3.4 Executing commands in running containers

When debugging an application running in a container, it may be necessary to examine the container and its environment from the inside. Kubectl provides this functionality, too. You can execute any binary file present in the container's file system using the `kubectl exec` command.

INVOKING A SINGLE COMMAND IN THE CONTAINER

For example, you can list the processes running in the container in the `kubia` pod by running the following command:

Listing 5.6 Processes running inside a pod container

```
$ kubectl exec kubia -- ps aux
USER  PID %CPU %MEM    VSZ   RSS TTY STAT START TIME COMMAND
root    1  0.0  1.3 812860 27356 ?   Ssl  11:54 0:00 node app.js #A
root  120  0.0  0.1  17500  2128 ?   Rs   12:22 0:00 ps aux      #B
```

#A The Node.js server
#B The command you've just invoked

This is the Kubernetes equivalent of the Docker command you used to explore the processes in a running container in chapter 2. It allows you to remotely run a command in any pod

并且不介意手动访问日志。在接下来的两节中，您将学习如何将日志和其他文件从容器复制到计算机以及相反的方向，以及如何在正在运行的容器中运行命令。您可以使用任一方法来显示日志文件或容器内的任何其他文件。

5.3.3 将文件复制到容器或从容器复制文件

有时您可能想要将文件添加到正在运行的容器或从中检索文件。在运行的容器中修改文件不是您通常会做的事情 - 至少在生产中不会 - 但它在开发过程中可能很有用。

Kubectl 提供 cp 命令，用于将文件或目录从本地计算机复制到任何 pod 的容器或从容器复制到您的计算机。例如，将 kubia pod 容器中的 /etc/hosts 文件复制到本地文件上的 /tmp 目录系统，运行以下命令：

```
$ kubectl cp kubia:/etc/hosts /tmp/kubia-hosts
```

要将文件从本地文件系统复制到容器，请运行以下命令：

```
$ kubectl cp /path/to/local/file kubia:path/in/container
```

注意 kubectl cp 命令要求容器中
存在 tar 二进制文件，但此要求
将来可能会发生变化。

5.3.4 在运行的容器中执行命令

当调试在容器中运行的应用程序时，可能需要从内部检查容器及其环境。 Kubectl 也提供了这个功能。您可以使用 kubectl exec 命令执行容器文件系统中存在的任何二进制文件。

在容器中调用单个命令

例如，您可以通过运行以下命令列出 kubia pod 中容器中运行的进程：

清单 5.6 在 Pod 容器内运行的进程

```
$ kubectl exec kubia -- ps aux USER PID %CPU %MEM VSZ
RSS TTY STAT 启动时间命令
根 1 0.0 1.3 812860 27356 ？安全套接字层:00 节点 app.js #A
根 120 0.0 0.1 17500 第2128章卢比 12:22 0:00 ps 辅助 #B
```

#A Node.js 服务
器 #B 您刚刚调用
的命令

这相当于您在第 2 章中用来探索正在运行的容器中的进程的 Docker 命令的 Kubernetes

without having to log in to the node that hosts the pod. If you've used `ssh` to execute commands on a remote system, you'll see that `kubectl exec` is not much different.

In section 5.3.1 you executed the `curl` command in a one-off client pod to send a request to your application, but you can also run the command inside the `kubia` pod itself:

```
$ kubectl exec kubia -- curl -s localhost:8080
Hey there, this is kubia. Your IP is ::1.
```

Why use a double dash in the kubectl exec command?

The double dash (`--`) in the command delimits kubectl arguments from the command to be executed in the container. The use of the double dash isn't necessary if the command has no arguments that begin with a dash. If you omit the double dash in the previous example, the `-s` option is interpreted as an option for `kubectl exec` and results in the following misleading error:

```
$ kubectl exec kubia curl -s localhost:8080
The connection to the server localhost:8080 was refused - did you specify the right host or port?
```

This may look like the Node.js server is refusing to accept the connection, but the issue lies elsewhere. The curl command is never executed. The error is reported by `kubectl` itself when it tries to talk to the Kubernetes API server at `localhost:8080`, which isn't where the server is. If you run the `kubectl options` command, you'll see that the `-s` option can be used to specify the address and port of the Kubernetes API server. Instead of passing that option to curl, kubectl adopted it as its own. Adding the double dash prevents this.
Fortunately, to prevent scenarios like this, newer versions of kubectl are set to return an error if you forget the double dash.

RUNNING AN INTERACTIVE SHELL IN THE CONTAINER

The two previous examples showed how a single command can be executed in the container. When the command completes, you are returned to your shell. If you want to run several commands in the container, you can run a shell in the container as follows:

```
$ kubectl exec -it kubia -- bash
root@kubia:/#         #A
```

#A The command prompt of the shell running in the container

The `-it` is short for two options: `-i` and `-t`, which indicate that you want to execute the `bash` command interactively by passing the standard input to the container and marking it as a terminal (TTY).

You can now explore the inside of the container by executing commands in the shell. For example, you can view the files in the container by running `ls -la`, view its network interfaces with `ip link`, or test its connectivity with `ping`. You can run any tool available in the container.

无需登录托管 pod 的节点。如果您使用 ssh 在远程系统上执行命令，您会发现 kubectl exec 没有太大不同。

在第 5.3.1 节中，您在一次性客户端 pod 中执行了curl命令以发送向您的应用程序发出请求，但您也可以在 kubia pod 本身内部运行该命令：

```
$ kubectl exec kubia --curl -s
localhost:8080 嘿，这是 kubia。您的 IP
是::1。
```

为什么在 kubectl exec 命令中使用双破折号？

命令中的双破折号 (--) 将 kubectl 参数与要在容器中执行的命令分隔开。如果命令没有以破折号开头的参数，则不需要使用双破折号。如果你

省略前面示例中的双破折号，-s 选项将被解释为 kubectl exec 的选项，并导致以下误导性错误：

```
$ kubectl 执行 kubia curl -s 本地主机:8080
与服务器 localhost:8080 的连接被拒绝 - 您指定了正确的主机或端口吗？
```

这可能看起来像 Node.js 服务器拒绝接受连接，但问题出在其他地方。卷曲

命令永远不会被执行。当 kubectl 尝试与 Kubernetes API 服务器通信时，它本身会报告该错误

位于 localhost:8080，这不是服务器所在的位置。如果运行 kubectl options 命令，您会看到

-s 选项可用于指定 Kubernetes API 服务器的地址和端口。 kubectl 没有将该选项传递给curl，而是将其作为自己的选项。添加双破折号可以防止这种情况。

幸运的是，为了防止出现这种情况，较新版本的 kubectl 设置为在您忘记双破折号时返回错误。

在容器中运行交互式 shell

前面的两个示例展示了如何在容器中执行单个命令。命令完成后，您将返回到 shell。如果要在容器中运行多个命令，可以在容器中运行 shell，如下所示：

```
$ kubectl exec -it
kubia -- bash
```

#A 容器中运行的shell的命令提示符

-it 是两个选项的缩写：-i 和 -t，这表明您希望通过将标准输入传递到容器并将其标记为以交互方式执行 bash 命令

终端(TTY)。

现在，您可以通过在 shell 中执行命令来探索容器的内部。为了例如，可以通过运行 ls -la 查看容器中的文件，查看其网络

与 ip link 接口，或使用 ping 测试其连通性。您可以运行容器中可用的任何工具。

NOT ALL CONTAINERS ALLOW YOU TO RUN SHELLS

The container image of your application contains many important debugging tools, but this isn't the case with every container image. To keep images small and improve security in the container, most containers used in production don't contain any binary files other than those required for the container's primary process. This significantly reduces the attack surface, but also means that you can't run shells or other tools in production containers. Fortunately, a new Kubernetes feature called *ephemeral containers* allows you to debug running containers by attaching a debug container to them.

> **NOTE TO MEAP READERS** Ephemeral containers are currently an alpha feature, which means they may change or even be removed at any time. This is also why they are currently not explained in this book. If they graduate to beta before the book goes into production, a section explaining them will be added.

5.3.5 Attaching to a running container

The `kubectl attach` command is another way to interact with a running container. It attaches itself to the standard input, output and error streams of the main process running in the container. Normally, you only use it to interact with applications that read from the standard input.

USING KUBECTL ATTACH TO SEE WHAT THE APPLICATION PRINTS TO STANDARD OUTPUT

If the application doesn't read from standard input, the `kubectl attach` command is only an alternative way to stream the application logs, as these are typically written to the standard output and error streams, and the `attach` command streams them just like the `kubectl logs -f` command does.

Attach to your `kubia` pod by running the following command:

```
$ kubectl attach kubia
Defaulting container name to kubia.
Use 'kubectl describe pod/kubia -n default' to see all of the containers in this
pod.
If you don't see a command prompt, try pressing enter.
```

Now, when you send new HTTP requests to the application using `curl` in another terminal, you'll see the lines that the application logs to standard output also printed in the terminal where the `kubectl attach` command is executed.

USING KUBECTL ATTACH TO WRITE TO THE APPLICATION'S STANDARD INPUT

The kubia application doesn't read from the standard input stream, but you'll find another version of the application that does this in the book's code archive. You can change the greeting with which the application responds to HTTP requests by writing it to the standard input stream of the application. Let's deploy this version of the application in a new pod and use the `kubectl attach` command to change the greeting.

You can find the artifacts required to build the image in the `kubia-stdin-image/` directory, or you can use the pre-built image `docker.io/luksa/kubia:1.0-stdin`. The pod manifest is in the file `kubia-stdin.yaml`. It is only slightly different from the pod manifest in

并非所有容器都允许您运行 shell

应用程序的容器映像包含许多重要的调试工具，但并非每个容器映像都是如此。为了保持镜像较小并提高容器中的安全性，生产中使用的大多数容器不包含除容器主进程所需的二进制文件之外的任何二进制文件。这显着减少了攻击面，但也意味着您无法在生产容器中运行 shell 或其他工具。幸运的是，一个称为临时容器的新 Kubernetes 功能允许您通过将调试容器附加到正在运行的容器来调试它们。

> MEAP 读者注意事项 临时容器目前是一个 alpha 功能，这意味着它们可能随时更改甚至被删除。这也是本书目前未对它们进行解释的原因。如果他们
> 在本书投入生产之前升级到测试版，将添加一个解释它们的部分。

5.3.5 附加到正在运行的容器

kubectl Attach 命令是与正在运行的容器交互的另一种方式。它

将自身附加到容器中运行的主进程的标准输入、输出和错误流。通常，您仅使用它与从标准输入读取数据的应用程序进行交互。

使用 KUBECTL ATTACH 查看应用程序打印到标准输出的内容

如果应用程序不从标准输入读取，则 kubectl Attach 命令只是流式传输应用程序日志的另一种方法，因为这些日志通常写入标准输出和错误流，并且 Attach 命令就像 kubectl 日志一样流式传输它们-f 命令可以。

通过运行以下命令连接到您的 kubia pod：

```
$ kubectl Attach
kubia 将容器名称默认
为 kubia。使
用“kubectl describe
pod/kubia -n
```

现在，当您在另一个终端中使用curl向应用程序发送新的HTTP请求时，您将看到应用程序记录到标准输出的行也打印在执行kubectl Attach命令的终端中。

使用 KUBECTL ATTACH 写入应用程序的标准输入

kubia 应用程序不会从标准输入流中读取数据，但您会在本书的代码存档中找到执行此操作的另一个版本的应用程序。您可以通过将应用程序响应 HTTP 请求的问候语写入应用程序的标准输入流来更改该问候语。让我们在新的 Pod 中部署此版本的应用程序，并使用 kubectl Attach 命令更改问候语。

您可以在 kubia-stdin-image/ 中找到构建映像所需的工件
目录，或者您可以使用预构建的映像 docker.io/luksa/kubia:1.0-stdin。 Pod 清单位于文件 kubia-stdin.yaml 中

kubia.yaml that you deployed previously. The differences are highlighted in the following listing.

Listing 5.7 Enabling standard input for a container

```
apiVersion: v1
kind: Pod
metadata:
  name: kubia-stdin                   #A
spec:
  containers:
  - name: kubia
    image: luksa/kubia:1.0-stdin      #B
    stdin: true                       #C
    ports:
    - containerPort: 8080
```

#A This pod is named kubia-stdin
#B It uses the special version of the kubia app
#C The application needs to read from the standard input stream

As you can see in the listing, if the application running in a pod wants to read from standard input, you must indicate this in the pod manifest by setting the stdin field in the container definition to true. This tells Kubernetes to allocate a buffer for the standard input stream, otherwise the application will always receive an EOF when it tries to read from it.

Create the pod from this manifest file with the kubectl apply command:

```
$ kubectl apply -f kubia-stdin.yaml
pod/kubia-stdin created
```

To enable communication with the application, use the kubectl port-forward command again, but because the local port 8080 is still being used by the previously executed port-forward command, you must either terminate it or choose a different local port to forward to the new pod. You can do this as follows:

```
$ kubectl port-forward kubia-stdin 8888:8080
Forwarding from 127.0.0.1:8888 -> 8080
Forwarding from [::1]:8888 -> 8080
```

The command-line argument 8888:8080 instructs the command to forward local port 8888 to the pod's port 8080.

You can now reach the application at http://localhost:8888:

```
$ curl localhost:8888
Hey there, this is kubia-stdin. Your IP is ::ffff:127.0.0.1.
```

Let's change the greeting from "Hey there" to "Howdy" by using kubectl attach to write to the standard input stream of the application. Run the following command:

```
$ kubectl attach -i kubia-stdin
```


您之前部署的 kubia.yaml。差异点如下所示

清单。

清单 5.7 为容器启用标准输入

```
api版本：v1
种类：Pod 元
数据
名称：kubia-stdin                #A
规格：
  容器： -
  名称：
  kubia
    图片：luksa/kubia:1.0-stdin  #B
    标准输入：真                 #C
    端口：
```

#A 这个 Pod 名为 kubia-stdin

#B 它使用 kubia 应用程序的特殊版本

正如您在清单中看到的，如果 pod 中运行的应用程序想要从标准输入读取，则必须通过将容器定义中的 stdin 字段设置为 true 来在 pod 清单中指示这一点。这告诉 Kubernetes 为标准输入流分配一个缓冲区，

否则应用程序在尝试读取时将始终收到 EOF。使用 kubectl apply 命令从此清单文件创建 pod：

```
$ kubectl  apply -f kubia-
stdin.yaml pod/kubia-stdin 创建
```

要启用与应用程序的通信，请再次使用 kubectl port-forward 命令，但由于之前执行的 portforward 命令仍在使用本地端口 8080，因此您必须终止它或选择其他本地端口转发到

新吊舱。您可以按如下方式执行此操作：

```
$ kubectl  port-forward  kubia-stdin
8888:8080 从 127.0.0.1 :8888 -> 8080 转
发
```

命令行参数 8888:8080 指示命令将本地端口 8888 转发到 Pod 的端口 8080。

您现在可以通过 http://localhost:8888 访问该应用程序：

```
$curl
localhost :8888 嘿，
这是 kubia-stdin。
您的 IP 是 ::ffff:127.0.0.1
。
```

让我们使用 kubectl Attach 写入应用程序的标准输入流，将问候语从“Hey There”更改为“Howdy”。运行以下命令：

Note the use of the additional option `-i` in the command. It instructs `kubectl` to pass its standard input to the container.

> **NOTE** Like the `kubectl exec` command, `kubectl attach` also supports the `--tty` or `-t` option, which indicates that the standard input is a terminal (TTY), but the container must be configured to allocate a terminal through the `tty` field in the container definition.

You can now enter the new greeting into the terminal and press the ENTER key. The application should then respond with the new greeting:

```
Howdy                          #A
Greeting set to: Howdy         #B
```

#A Type the greeting and press <ENTER>
#B This is the application's response

To see if the application now responds to HTTP requests with the new greeting, re-execute the `curl` command or refresh the page in your web browser:

```
$ curl localhost:8888
Howdy, this is kubia-stdin. Your IP is ::ffff:127.0.0.1.
```

There's the new greeting. You can change it again by typing another line in the terminal with the `kubectl attach` command. To exit the `attach` command, press Control-C or the equivalent key.

> **NOTE** An additional field in the container definition, `stdinOnce`, determines whether the standard input channel is closed when the attach session ends. It's set to `false` by default, which allows you to use the standard input in every `kubectl attach` session. If you set it to `true`, standard input remains open only during the first session.

5.4 Running multiple containers in a pod

The kubia application you deployed in section 5.2 only supports HTTP. Let's add TLS support so it can also serve clients over HTTPS. You could do this by writing additional code, but an easier option exists where you don't need to touch the code at all.

You can run a reverse proxy alongside the Node.js application in a sidecar container, as explained in section 5.1.2, and let it handle HTTPS requests on behalf of the application. A very popular software package that can provide this functionality is *Envoy*. The Envoy proxy is a high-performance open source service proxy originally built by Lyft that has since been contributed to the Cloud Native Computing Foundation. Let's add it to your pod.

5.4.1 Extending the kubia Node.js application using the Envoy proxy

Let me briefly explain what the new architecture of the application will look like. As shown in the next figure, the pod will have two containers - the Node.js and the new Envoy container. The Node.js container will continue to handle HTTP requests directly, but the HTTPS requests

请注意命令中附加选项 -i 的使用。它指示 kubectl 将其标准输入传递给容器。

> 注意 与 kubectl exec 命令一样，kubectl Attach 也支持 --tty 或 -t 选项，即
> 表示标准输入是终端（TTY），但容器必须配置为分配一个
> 终端通过容器定义中的 tty 字段

您现在可以在终端中输入新的问候语并按 ENTER 键。然后应用程序应使用新的问候语进行响应：

```
你好                          #A
问候语设置为：你好            #B
```

#A 输入问候语并按 #B 这是应用程
序的响应

要查看应用程序现在是否使用新问候语响应 HTTP 请求，请重新执行 curl 命令或刷新 Web 浏览器中的页面：

```
$ 卷曲本地主机：8888
```

有新的问候语。您可以通过使用 kubectl 在终端中键入另一行来再次更改它
附加命令。要退出附加命令，请按 Control-C 或
等效密钥。

> 注意容器定义中的附加字段 stdinOnce 确定标准输入是否
> 当附加会话结束时，通道将关闭。默认情况下它设置为 false，这允许您使用
> 每个 kubectl Attach 会话中的标准输入。如果将其设置为 true，则标准输入仅保持打开
> 状态

5.4 在一个pod中运行多个容器

您在 5.2 节中部署的 kubia 应用程序仅支持 HTTP。让我们添加 TLS 支持，以便它也可以通过 HTTPS 为客户端提供服务。您可以通过编写额外的代码来做到这一点，但还有一个更简单的选择，您根本不需要接触代码。

您可以在 sidecar 容器中与 Node.js 应用程序一起运行反向代理，如第 5.1.2 节中所述，并让它代表应用程序处理 HTTPS 请求。可以提供此功能的一个非常流行的软件包是 Envoy。 Envoy 代理是一个高性能开源服务代理，最初由 Lyft 构建，后来贡献给云原生计算基金会。让我们将其添加到您的 pod 中。

5.4.1 使用 Envoy 代理扩展 kubia Node.js 应用程序

让我简要解释一下应用程序的新架构会是什么样子。如下图所示，pod 将有两个容器 - Node.js 和新的 Envoy 容器。 Node.js 容器将继续直接处理 HTTP 请求

will be handled by Envoy. For each incoming HTTPS request, Envoy will create a new HTTP request that it will then send to the Node.js application via the local loopback device (via the localhost IP address).

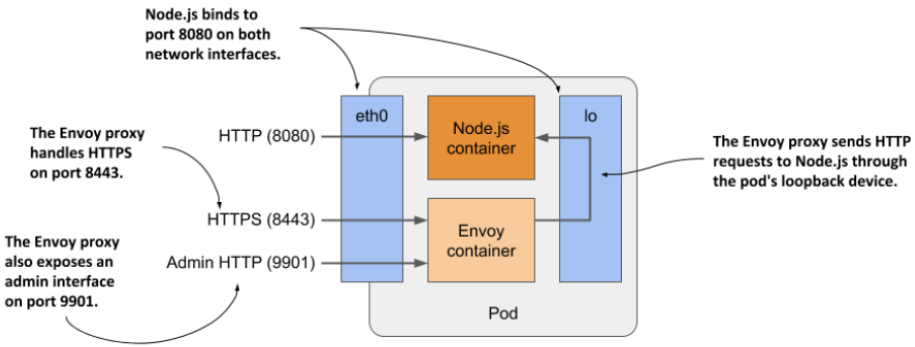


Figure 5.10 Detailed view of the pod's containers and network interfaces

Envoy also provides a web-based administration interface that will prove handy in some of the exercises in the next chapter.

It's obvious that if you implement TLS support within the Node.js application itself, the application will consume less computing resources and have lower latency because no additional network hop is required, but adding the Envoy proxy could be a faster and easier solution. It also provides a good starting point from which you can add many other features provided by Envoy that you would probably never implement in the application code itself. Refer to the Envoy proxy documentation at envoyproxy.io to learn more.

5.4.2 Adding Envoy proxy to the pod

You'll create a new pod with two containers. You've already got the Node.js container, but you also need a container that will run Envoy.

CREATING THE ENVOY CONTAINER IMAGE

The authors of the proxy have published the official Envoy proxy container image at Docker Hub. You could use this image directly, but you would need to somehow provide the configuration, certificate, and private key files to the Envoy process in the container. You'll learn how to do this in chapter 7. For now, you'll use an image that already contains all three files.

I've already created the image and made it available at `docker.io/luksa/kubia-ssl-proxy:1.0`, but if you want to build it yourself, you can find the files in the `kubia-ssl-proxy-image` directory in the book's code archive.

将由 Envoy 处理。对于每个传入的 HTTPS 请求，Envoy 将创建一个新的 HTTP 请求，然后通过本地环回设备（通过本地主机 IP 地址）将其发送到 Node.js 应用程序。

图 5.10 Pod 的容器和网络接口的详细视图

Envoy 还提供了一个基于 Web 的管理界面，该界面将在下一章的一些练习中证明很方便。

很明显，如果您在 Node.js 应用程序本身中实现 TLS 支持，则该应用程序将消耗更少的计算资源并具有更低的延迟，因为不需要额外的网络跃点，但添加 Envoy 代理可能是一个更快、更简单的解决方案。它还提供了一个很好的起点，您可以从中添加 Envoy 提供的许多其他功能，而您可能永远不会在应用程序代码本身中实现这些功能。请参阅 envoyproxy.io 上的 Envoy 代理文档了解更多信息。

5.4.2 将 Envoy 代理添加到 pod 中

您将创建一个包含两个容器的新 Pod。您已经有了 Node.js 容器，但还需要一个运行 Envoy 的容器。

创建 Envoy 容器镜像

该代理的作者已在 Docker Hub 上发布了官方 Envoy 代理容器镜像。您可以直接使用此映像，但需要以某种方式向容器中的 Envoy 进程提供配置、证书和私钥文件。您将在第 7 章中学习如何执行此操作。现在，您将使用已包含所有三个文件的图像。

我已经创建了该映像并在 `docker.io/luksa/kubia-sslproxy:1.0` 上提供了它，但是如果您想自己构建它，您可以在本书代码中的 `kubia-sslproxy-image` 目录中找到这些文件档案。

The directory contains the `Dockerfile`, as well as the private key and certificate that the proxy will use to serve HTTPS. It also contains the `envoy.conf` config file. In it, you'll see that the proxy is configured to listen on port `8443`, terminate TLS, and forward requests to port `8080` on `localhost`, which is where the Node.js application is listening. The proxy is also configured to provide an administration interface on port `9901`, as explained earlier.

CREATING THE POD MANIFEST

After building the image, you must create the manifest for the new pod, as shown in the following listing.

Listing 5.8 Manifest of pod kubia-ssl (kubia-ssl.yaml)

```
apiVersion: v1
kind: Pod
metadata:
  name: kubia-ssl
spec:
  containers:
  - name: kubia                                  #A
    image: luksa/kubia:1.0                       #A
    ports:                                       #A
    - name: http                #B               #A
      containerPort: 8080       #B               #A
  - name: envoy                                  #C
    image: luksa/kubia-ssl-proxy:1.0             #C
    ports:                                       #C
    - name: https               #D               #C
      containerPort: 8443       #D               #C
    - name: admin               #E               #C
      containerPort: 9901       #E               #C
```

#A The container running the Node.js app
#B Node.js listens on port 8080
#C The container running the Envoy proxy
#D The proxy's HTTPS port
#E The proxy's admin interface port

The name of this pod is `kubia-ssl`. It has two containers: `kubia` and `envoy`. The manifest is only slightly more complex than the manifest in section 5.2.1. The only new fields are the port names, which are included so that anyone reading the manifest can understand what each port number stands for.

CREATING THE POD

Create the pod from the manifest using the command `kubectl apply -f kubia-ssl.yaml`. Then use the `kubectl get` and `kubectl describe` commands to confirm that the pod's containers were successfully launched.

5.4.3 Interacting with the two-container pod

When the pod starts, you can start using the application in the pod, inspect its logs and explore the containers from within.

该目录包含 Dockerfile，以及代理将用于提供 HTTPS 服务的私钥和证书。它还包含 envoy.conf 配置文件。在其中，您将看到代理配置为侦听端口 8443、终止 TLS 并将请求转发到

本地主机上的端口 8080，这是 Node.js 应用程序正在侦听的位置。代理还配置为在端口 9901 上提供管理界面，如前所述。

创建 Pod 清单

构建映像后，您必须为新 Pod 创建清单，如以下清单所示。

清单 5.8 pod kubia-ssl 清单 (kubia-ssl.yaml)

```
api 版本：v1
种类：Pod 元
数据
  名称：kubia-ssl
规格：
  容器：  - 名称：kubia  #A

    图片：luksa/kubia:1.0  #A

    端口：#A
      容器端口：8080  #B #A
  - 姓名：特使#C
    图片：luksa/kubia-ssl-proxy:1.0  #C

    端口：#C
      容器端口：8443  #D #C
    - 姓名：管理员#E  #C
      容器端口：9901  #E #C
```

#A 运行 Node.js 应用程序的容器
#B Node.js 侦听端口 8080 #C 运行 Envoy 代理的容器 #D 代理的 HTTPS 端口
#E 代理的管理接口端口

该 Pod 的名称是 kubia-ssl。它有两个容器：kubia 和 envoy。该清单仅比第 5.2.1 节中的清单稍微复杂一些。唯一的新字段是端口名称，这样任何阅读清单的人都可以理解每个端口号代表什么。

创建 Pod

使用命令 kubectl apply -f kubia-ssl.yaml 从清单创建 pod。

然后使用 kubectl get 和 kubectl describe 命令确认 pod 的容器已成功启动。

5.4.3 与二容器 Pod 交互

当 Pod 启动时，您可以开始使用 Pod 中的应用程序、检查其日志并从内部探索容器。

COMMUNICATING WITH THE APPLICATION

As before, you can use the `kubectl port-forward` to enable communication with the application in the pod. Because it exposes three different ports, you enable forwarding to all three ports as follows:

```
$ kubectl port-forward kubia-ssl 8080 8443 9901
Forwarding from 127.0.0.1:8080 -> 8080
Forwarding from [::1]:8080 -> 8080
Forwarding from 127.0.0.1:8443 -> 8443
Forwarding from [::1]:8443 -> 8443
Forwarding from 127.0.0.1:9901 -> 9901
Forwarding from [::1]:9901 -> 9901
```

First, confirm that you can communicate with the application via HTTP by opening the URL http://localhost:8080 in your browser or by using `curl`:

```
$ curl localhost:8080
Hey there, this is kubia-ssl. Your IP is ::ffff:127.0.0.1.
```

If this works, you can also try to access the application over HTTPS at https://localhost:8443. With `curl` you can do this as follows:

```
$ curl https://localhost:8443 --insecure
Hey there, this is kubia-ssl. Your IP is ::ffff:127.0.0.1.
```

Success! The Envoy proxy handles the task perfectly. Your application now supports HTTPS using a sidecar container.

Why it is necessary to use the --insecure option

There are two reasons why you must use the `--insecure` option when accessing the service. The certificate used by the Envoy proxy is self-signed and was issued for the domain name `example.com`. You access the service via the local `kubectl proxy` and use `localhost` as the domain name in the URL, which means that it doesn't match the name in the server certificate. To make it match, you'd have to use the following command:

```
$ curl https://example.com:8443 --resolve example.com:8443:127.0.0.1
```

This ensures that the certificate matches the requested URL, but because the certificate is self-signed, `curl` still can't verify the legitimacy of the server. You must either replace the server's certificate with a certificate signed by a trusted authority or use the `--insecure` flag anyway; in this case, you also don't need to bother with using the `--resolve` flag.

DISPLAYING LOGS OF PODS WITH MULTIPLE CONTAINERS

The `kubia-ssl` pod contains two containers, so if you want to display the logs, you must specify the name of the container using the `--container` or `-c` option. For example, to view the logs of the `kubia` container, run the following command:

```
$ kubectl logs kubia-ssl -c kubia
```


132

与应用程序通信

和以前一样，您可以使用 kubectl 端口转发以启用与
pod 中的应用程序。由于它公开三个不同的端口，因此您可以按如下方式启用到所有三个端口的转发：

```
$ kubectl 端口转发 kubia-ssl 8080 8443
9901
转发自 127.0.0.1 :8080 -> 8080
转发自 [::1]:8080 -> 8080
转发自 127.0.0.1 :8443 -> 8443
```

首先，通过在浏览器中打开 URL http://localhost:8080 或使用 curl 来确认您可以通过 HTTP 与应用程序进行通信：

```
$ 卷曲本地主机：8080
```

如果这 作品， 你 能 还 尝试到 使用权 这 应用 超过 HTTPS 在 https://localhost:8443。使用curl，您可以按如下方式执行此操作：

```
$curl https://localhost :8443 --不安全
```

成功！ Envoy 代理完美地处理了该任务。您的应用程序现在使用 sidecar 容器支持 HTTPS。

为什么需要使用--insecure选项
访问服务时必须使用 --insecure 选项有两个原因。使用的证书

Envoy 代理是自签名的，是为域名 example.com 颁发的。您通过以下方式访问该服务

本地 kubectl 代理并使用 localhost 作为 URL 中的域名，这意味着它与服务器证书中的名称不匹配。要使其匹配，您必须使用以下命令：

```
$curl https://example.com:8443 --resolve example.com:8443:127.0.0.1
```

这确保了证书与请求的 URL 匹配，但由于证书是自签名的，curl 仍然无法验证服务器的合法性。您必须将服务器的证书替换为受信任的证书签名的证书

无论如何，授权或使用 --insecure 标志；在这种情况下，您也不需要费心使用 --resolve 标志。

显示具有多个容器的 Pod 日志

kubia-ssl pod包含两个容器，因此如果要显示日志，则必须

使用 --container 或 -c 选项指定容器的名称。例如，要查看kubia容器的日志，请运行以下命令：

```
$ kubectl 日志 kubia-ssl -c kubia
```

The Envoy proxy runs in the container named `envoy`, so you display its logs as follows:

```
$ kubectl logs kubia-ssl -c envoy
```

Alternatively, you can display the logs of both containers with the `--all-containers` option:

```
$ kubectl logs kubia-ssl --all-containers
```

You can also combine these commands with the other options explained in section 5.3.2.

RUNNING COMMANDS IN CONTAINERS OF MULTI-CONTAINER PODS

If you'd like to run a shell or another command in one of the pod's containers using the `kubectl exec` command, you also specify the container name using the `--container` or `-c` option. For example, to run a shell inside the envoy container, run the following command:

```
$ kubectl exec -it kubia-ssl -c envoy -- bash
```

NOTE If you don't provide the name, `kubectl exec` defaults to the first container specified in the pod manifest.

5.5 Running additional containers at pod startup

When a pod contains more than one container, all the containers are started in parallel. Kubernetes doesn't yet provide a mechanism to specify whether a container depends on another container, which would allow you to ensure that one is started before the other. However, Kubernetes allows you to run a sequence of containers to initialize the pod before its main containers start. This special type of container is explained in this section.

5.5.1 Introducing init containers

A pod manifest can specify a list of containers to run when the pod starts and before the pod's normal containers are started. These containers are intended to initialize the pod and are appropriately called *init containers*. As the following figure shows, they run one after the other and must all finish successfully before the main containers of the pod are started.

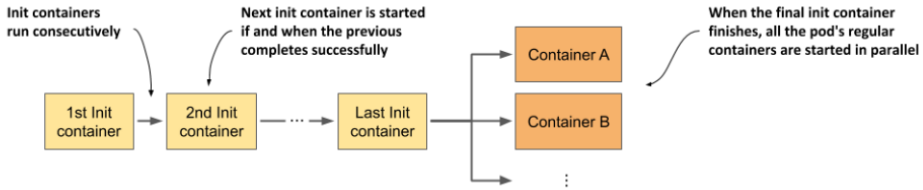


Figure 5.11 Time sequence showing how a pod's init and regular containers are started

Envoy 代理在名为 envoy 的容器中运行，因此您显示其日志如下：

```
$ kubectl 日志 kubia-ssl -c envoy
```

或者，您可以使用 --all-containers 选项显示两个容器的日志：

```
$ kubectl 日志 kubia-ssl --all-containers
```

您还可以将这些命令与第 5.3.2 节中解释的其他选项结合起来。

在多容器 Pod 的容器中运行命令

如果您想使用 kubectl exec 命令在 Pod 的容器之一中运行 shell 或其他命令，您还可以使用 --container 或 -c 指定容器名称

选项。例如，要在 Envoy 容器内运行 shell，请运行以下命令：

```
$ kubectl exec -it kubia-ssl -c envoy -- bash
```

注意：如果不提供名称，kubectl exec 默认为 pod 中指定的第一个容器显现。

5.5 在 Pod 启动时运行附加容器

当一个 Pod 包含多个容器时，所有容器都会并行启动。 Kubernetes 尚未提供一种机制来指定一个容器是否依赖于另一个容器，这将允许您确保一个容器先于另一个容器启动。但是，Kubernetes 允许您运行一系列容器来在 pod 的主容器启动之前对其进行初始化。本节将介绍这种特殊类型的容器。

5.5.1 引入init容器

Pod 清单可以指定在 Pod 启动时以及 Pod 的正常容器启动之前要运行的容器列表。这些容器用于初始化 pod，适当地称为 init 容器。如下图所示，它们依次运行，并且必须全部成功完成才能启动 Pod 的主容器。

图5

Init containers are like the pod's regular containers, but they don't run in parallel - only one init container runs at a time.

UNDERSTANDING WHAT INIT CONTAINERS CAN DO

Init containers are typically added to pods to achieve the following:

- Initialize files in the volumes used by the pod's main containers. This includes retrieving certificates and private keys used by the main container from secure certificate stores, generating config files, downloading data, and so on.
- Initialize the pod's networking system. Because all containers of the pod share the same network namespaces, and thus the network interfaces and configuration, any changes made to it by an init container also affect the main container.
- Delay the start of the pod's main containers until a precondition is met. For example, if the main container relies on another service being available before the container is started, an init container can block until this service is ready.
- Notify an external service that the pod is about to start running. In special cases where an external system must be notified when a new instance of the application is started, an init container can be used to deliver this notification.

You could perform these operations in the main container itself but using an init container is sometimes a better option and can have other advantages. Let's see why.

UNDERSTANDING WHEN MOVING INITIALIZATION CODE TO INIT CONTAINERS MAKES SENSE

Using an init container to perform initialization tasks doesn't require the main container image to be rebuilt and allows a single init container image to be reused with many different applications. This is especially useful if you want to inject the same infrastructure-specific initialization code into all your pods. Using an init container also ensures that this initialization is complete before any of the (possibly multiple) main containers start.

Another important reason is security. By moving tools or data that could be used by an attacker to compromise your cluster from the main container to an init container, you reduce the pod's attack surface.

For example, imagine that the pod must be registered with an external system. The pod needs some sort of secret token to authenticate against this system. If the registration procedure is performed by the main container, this secret token must be present in its filesystem. If the application running in the main container has a vulnerability that allows an attacker to read arbitrary files on the filesystem, the attacker may be able to obtain this token. By performing the registration from an init container, the token must be available only in the filesystem of the init container, which an attacker can't easily compromise.

5.5.2 Adding init containers to a pod

In a pod manifest, init containers are defined in the `initContainers` field in the spec section, just as regular containers are defined in its `containers` field.

Init 容器就像 pod 的常规容器一样，但它们不并行运行 - 一次只有一个 init 容器运行。

了解 INIT 容器可以做什么

Init 容器通常添加到 Pod 中以实现以下目的：

- 初始化 pod 主容器使用的卷中的文件。这包括从安全证书存储中检索主容器使用的证书和私钥、生成配置文件、下载数据等。
- 初始化容器的网络系统。由于 Pod 的所有容器共享相同的网络命名空间，从而共享网络接口和配置，因此 init 容器对其所做的任何更改也会影响主容器。
- 延迟启动 Pod 的主容器，直到满足前提条件。例如，如果主容器在启动容器之前依赖于另一个可用的服务，则 init 容器可能会阻塞，直到该服务准备就绪。
- 通知外部服务Pod 即将开始运行。在启动应用程序的新实例时必须通知外部系统的特殊情况下，可以使用 init 容器来传递此通知。

您可以在主容器本身中执行这些操作，但使用 init 容器有时是更好的选择，并且可以具有其他优点。让我们看看为什么。

了解何时将初始化代码移动到初始化容器才有意义

使用 init 容器执行初始化任务不需要重建主容器映像，并且允许单个 init 容器映像在许多不同的应用程序中重复使用。如果您想将相同的特定于基础设施的初始化代码注入到所有 Pod 中，这尤其有用。使用 init 容器还可以确保在任何（可能是多个）主容器启动之前完成初始化。

另一个重要原因是安全。通过将攻击者可用来危害集群的工具或数据从主容器移动到 init 容器，可以减少 Pod 的攻击面。

例如，假设 pod 必须向外部系统注册。 Pod 需要某种秘密令牌来针对该系统进行身份验证。如果注册过程由主容器执行，则该秘密令牌必须存在于其文件系统中。如果在主容器中运行的应用程序存在允许攻击者读取文件系统上的任意文件的漏洞，则攻击者可能能够获取此令牌。通过从 init 容器执行注册，令牌必须仅在 init 容器的文件系统中可用，攻击者无法轻易破解。

5.5.2 将init容器添加到pod中

在 pod 清单中，init 容器在规范的 initContainers 字段中定义

部分，就像在其Containers字段中定义常规容器一样。

ADDING TWO CONTAINERS TO THE KUBIA-SSL POD

Let's look at an example of adding two init containers to the kubia pod. The first init container emulates an initialization procedure. It runs for 5 seconds, while printing a few lines of text to standard output.

The second init container performs a network connectivity test by using the `ping` command to check if a specific IP address is reachable from within the pod. If the IP address is not specified, the address 1.1.1.1 is used. You'll find the `Dockerfiles` and other artifacts for both images in the book's code archive, if you want to build them yourself. Alternatively, you can use the pre-build images specified in the following listing.

A pod manifest containing these two init containers is in the `kubia-init.yaml` file. The following listing shows how the init containers are defined.

Listing 5.9 Defining init containers in a pod manifest: kubia-init.yaml

```
apiVersion: v1
kind: Pod
metadata:
  name: kubia-init
spec:
  initContainers:                                    #A
  - name: init-demo                                  #B
    image: luksa/init-demo:1.0                       #B
  - name: network-check                              #C
    image: luksa/network-connectivity-checker:1.0    #C
  containers:                                        #D
  - name: kubia                                      #D
    image: luksa/kubia:1.0                           #D
    ports:                                           #D
    - name: http                                     #D
      containerPort: 8080                            #D
  - name: envoy                                      #D
    image: luksa/kubia-ssl-proxy:1.0                 #D
    ports:                                           #D
    - name: https                                    #D
      containerPort: 8443                            #D
    - name: admin                                    #D
      containerPort: 9901                            #D
```

#A Init containers are specified in the initContainers field
#B This container runs first
#C This container runs after the first one completes
#D These are the pod's regular containers. They run at the same time.

As you can see, the definition of an init container is almost trivial. It's sufficient to specify only the `name` and `image` for each container.

NOTE Container names must be unique within the union of all init and regular containers.

135

将两个容器添加到 KUBIA-SSL POD

让我们看一个向 kubia pod 添加两个 init 容器的示例。第一个 init 容器模拟初始化过程。它运行 5 秒，同时将几行文本打印到标准输出。

第二个 `init` 容器使用 `ping` 命令执行网络连接测试，以检查是否可以从 `pod` 内访问特定 IP 地址。如果未指定 IP 地址，则使用地址 1.1.1.1。如果您想自己构建这两个映像，您可以在本书的代码存档中找到 `Dockerfile` 和其他工件。或者，您可以使用以下列表中指定的预构建映像。

包含这两个 init 容器的 pod 清单位于 kubia-init.yaml 文件中。这以下清单显示了如何定义 init 容器。

清单 5.9 在 pod 清单中定义 init 容器：kubia-init.yaml

```
api版本：v1
种类：Pod 元
数据
  名称：kubia-init
规格：
  初始化容器：#A

    图片：luksa/init-demo:1.0 #B
  - 名称：网络检查#C
    图片：luksa/network-connectivity-checker:1.0 #C
  容器：#D

    图片：luksa/kubia:1.0 #D

    端口：#D
      容器端口：8080 #D
  - 姓名：特使#D
    图片：luksa/kubia-ssl-proxy:1.0 #D

    端口：#D
      容器端口：8443 #D
    - 姓名：管理员#D
      容器端口：9901 #D
```

#A Init 容器在 initContainers 字段中指定 #B 该容器首先运行 #C 该容器在第一个容器完成后运行 #D 这些是 Pod 的常规容器。他们同时运行。

正如您所看到的，init 容器的定义几乎是微不足道的。只需指定每个容器的名称和图像就足够了。

注意：容器名称在所有 init 容器和常规容器的联合体中必须是唯一的。

DEPLOYING A POD WITH INIT CONTAINERS

Before you create the pod from the manifest file, run the following command in a separate terminal so you can see how the pod's status changes as the init and regular containers start:

```
$ kubectl get pods -w
```

You'll also want to watch events in another terminal using the following command:

```
$ kubectl get events -w
```

When ready, create the pod by running the apply command:

```
$ kubectl apply -f kubia-init.yaml
```

INSPECTING THE STARTUP OF A POD WITH INIT CONTAINERS

As the pod starts up, inspect the events that the `kubectl get events -w` command prints. The following listing shows what you should see.

Listing 5.10 Pod events showing how the execution of init containers

```
TYPE     REASON      MESSAGE
Normal   Scheduled   Successfully assigned pod to worker2
Normal   Pulling     Pulling image "luksa/init-demo:1.0"       #A
Normal   Pulled      Successfully pulled image                 #A
Normal   Created     Created container init-demo               #A
Normal   Started     Started container init-demo               #A
Normal   Pulling     Pulling image "luksa/network-connec...    #B
Normal   Pulled      Successfully pulled image                 #B
Normal   Created     Created container network-check           #B
Normal   Started     Started container network-check           #B
Normal   Pulled      Container image "luksa/kubia:1.0"         #C
                     already present on machine                #C
Normal   Created     Created container kubia                   #C
Normal   Started     Started container kubia                   #C
Normal   Pulled      Container image "luksa/kubia-ssl-         #C
                     proxy:1.0" already present on machine     #C
Normal   Created     Created container envoy                   #C
Normal   Started     Started container envoy                   #C
```

#A The first init container's image is pulled and the container is started
#B After the first init container completes, the second is started
#C The pod's two main containers are then started in parallel

The listing shows the order in which the containers are started. The `init-demo` container is started first. When it completes, the `network-check` container is started, and when it completes, the two main containers, `kubia` and `envoy`, are started.

Now inspect the transitions of the pod's status in the other terminal. They are shown in the next listing.

136

使用 Init 容器部署 Pod

在从清单文件创建 pod 之前，请在单独的终端中运行以下命令，以便您可以看到 pod 的状态在 init 和常规容器启动时如何变化：

```
$ kubectl 获取 pod -w
```

您还需要使用以下命令在另一个终端中观看事件：

```
$ kubectl 获取事件 -w
```

准备就绪后，通过运行 apply 命令创建 pod：

```
$ kubectl apply -f kubia-init.yaml
```

使用 INIT 容器检查 Pod 的启动情况

当 Pod 启动时，检查 kubectl get events -w 命令打印的事件。以下列表显示了您应该看到的内容。

清单 5.10 Pod 事件显示了 init 容器是如何执行的

```
类型  原因  消息
正常 调度 成功 将pod分配给worker2
正常拉动  拉动镜像“luksa/init-demo:1.0”#A
正常拉取成功拉取图像#A
正常创建已创建容器 init-demo #A
正常启动  已启动容器 init-demo #A
正常拉动  拉动图像“luksa/network-connec...#B
正常创建  已创建容器 kubia #C
正常启动  已启动容器 kubia #C
正常创建已创建容器特使#C
```

已经存在于机器#C上

proxy:1.0″ 已存在于机器 #C 上

#A 第一个 init 容器的镜像被拉取并启动容器 #B 第一个 init 容器完成后，启动第二个 #C 然后 pod 的两个主容器并行启动

该列表显示了容器的启动顺序。首先启动 init-demo 容器。完成后，网络检查容器将启动，完成后，两个主容器 kubia 和 envoy 也会启动。

现在检查另一个终端中 pod 状态的转换。它们显示在下一个列表。

Listing 5.11 Pod status changes during startup involving init containers

```
NAME          READY   STATUS            RESTARTS   AGE
kubia-init    0/2     Pending           0          0s
kubia-init    0/2     Pending           0          0s
kubia-init    0/2     Init:0/2          0          0s          #A
kubia-init    0/2     Init:0/2          0          1s          #A
kubia-init    0/2     Init:1/2          0          6s          #B
kubia-init    0/2     PodInitializing   0          7s          #C
kubia-init    2/2     Running           0          8s          #D
```

#A The first init container is running
#B The first init container is complete, the second is now running
#C All init containers have completed successfully
#D The pod's main containers are running

As the listing shows, when the init containers run, the pod's status shows the number of init containers that have completed and the total number. When all init containers are done, the pod's status is displayed as `PodInitializing`. At this point, the images of the main containers are pulled. When the containers start, the status changes to `Running`.

5.5.3 Inspecting init containers

While the init containers run and after they have finished, you can display their logs and enter the running container, just as you can with regular containers.

DISPLAYING THE LOGS OF AN INIT CONTAINER

The standard and error output, into which each init container can write, are captured exactly as they are for regular containers. The logs of an init container can be displayed using the `kubectl logs` command by specifying the name of the container with the `-c` option. To display the logs of the `network-check` container in the `kubia-init` pod, run the command shown in the following listing.

Listing 5.12 Displaying the logs of an init container

```
$ kubectl logs kubia-init -c network-check
Checking network connectivity to 1.1.1.1 ...
Host appears to be reachable
```

The logs show that the `network-check` init container ran without errors. In the next chapter, you'll see what happens if an init container fails.

ENTERING A RUNNING INIT CONTAINER

You can use the `kubectl exec` command to run a shell or a different command inside an init container the same way you can with regular containers, but you can only do this before the init container terminates. If you'd like to try this yourself, create a pod from the `kubia-init-slow.yaml` file, which makes the `init-demo` container run for 60 seconds. When the pod starts, run a shell in the container with the following command:

```
$ kubectl exec -it kubia-init-slow -c init-demo -- sh
```


清单 5.11 启动期间涉及 init 容器的 Pod 状态变化

```
姓名          准备好 地位              重启      年龄
kubia初始化 0/2    待办的              0         0秒
kubia初始化 0/2    待办的              0         0秒
kubia初始化 0/2    初始：0/2           0         0秒         #A
kubia初始化 0/2    初始：0/2           0         1秒         #A
kubia初始化 0/2    初始：1/2           0         6秒         #B
kubia初始化 0/2    Pod初始化中         0         7秒         #C
kubia初始化 2/2    跑步                0         8秒         #D
```

#A 第一个 init 容器正在运行

#B 第一个 init 容器已完成，第二个正在运行 #C 所有 init 容器已成功完成 #D pod 的主容器正在运行

如清单所示，当 init 容器运行时，pod 的状态显示已完成的 init 容器的数量和总数。当所有初始化容器完成后，Pod 的状态显示为 PodInitializing。至此，主要容器的镜像就被拉取了。当容器启动时，状态更改为正在运行。

5.5.3 检查初始化容器

当初始化容器运行时以及完成后，您可以显示其日志并进入正在运行的容器，就像使用常规容器一样。

显示初始化容器的日志

每个 init 容器可以写入的标准输出和错误输出的捕获方式与常规容器的捕获方式完全一样。可以使用 kubectl messages 命令通过 -c 选项指定容器名称来显示 init 容器的日志。到

要显示 kubia-init pod 中网络检查容器的日志，请运行以下列表中所示的命令。

清单 5.12 显示 init 容器的日志

```
$ kubectl logs kubia-init -c network-
check 检查 1.1.1.1 的网络连接 ... 主
机似乎可以访问
```

日志显示网络检查 init 容器运行时没有错误。在下一章中，您将看到如果初始化容器失败会发生什么。

进入正在运行的初始化容器

您可以使用 kubectl exec 命令在 init 容器内运行 shell 或其他命令，就像使用常规容器一样，但只能在 init 容器终止之前执行此操作。如果您想自己尝试一下，请从 kubiainit-slow.yaml 文件创建一个 pod，这会使 init-demo 容器运行 60 秒。当。。。的时候

pod 启动后，使用以下命令在容器中运行 shell：

You can use the shell to explore the container from the inside, but not for long. When the container's main process exits after 60 seconds, the shell process is also terminated.

You typically enter a running init container only when it fails to complete in time, and you want to find the cause. During normal operation, the init container terminates before you can run the `kubectl exec` command.

5.6 Deleting pods and other objects

If you've tried the exercises in this chapter and in chapter 2, several pods and other objects now exist in your cluster. To close this chapter, you'll learn various ways to delete them. Deleting a pod will terminate its containers and remove them from the node. Deleting a Deployment object causes the deletion of its pods, whereas deleting a LoadBalancer-typed Service deprovisions the load balancer if one was provisioned.

5.6.1 Deleting a pod by name

The easiest way to delete an object is to delete it by name.

DELETING A SINGLE POD

Use the following command to remove the `kubia` pod from your cluster:

```
$ kubectl delete po kubia
pod "kubia" deleted
```

By deleting a pod, you state that you no longer want the pod or its containers to exist. The Kubelet shuts down the pod's containers, removes all associated resources, such as log files, and notifies the API server after this process is complete. The Pod object is then removed.

TIP By default, the `kubectl delete` command waits until the object no longer exists. To skip the wait, run the command with the `--wait=false` option.

While the pod is in the process of shutting down, its status changes to `Terminating`:

```
$ kubectl get po kubia
NAME    READY   STATUS        RESTARTS   AGE
kubia   1/1     Terminating   0          35m
```

Knowing exactly how containers are shut down is important if you want your application to provide a good experience for its clients. This is explained in the next chapter, where we dive deeper into the life cycle of the pod and its containers.

NOTE If you're familiar with Docker, you may wonder if you can stop a pod and start it again later, as you can with Docker containers. The answer is no. With Kubernetes, you can only remove a pod completely and create it again later.

您可以使用 shell 从内部探索容器，但时间不会太长。当容器的主进程在60秒后退出时，shell进程也会终止。

通常，只有当 init 容器未能及时完成并且您想要查找原因时，您才会进入正在运行的 init 容器。在正常操作期间，init 容器会在您运行 kubectl exec 命令之前终止。

5.6 删除 pod 和其他对象

如果您尝试过本章和第 2 章中的练习，您的集群中现在存在多个 Pod 和其他对象。为了结束本章，您将学习删除它们的各种方法。删除 Pod 将终止其容器并将其从节点中删除。删除 Deployment 对象会导致删除其 pod，而删除 LoadBalancer 类型的 Service 会取消配置负载均衡器（如果已配置）。

5.6.1 根据名称删除 Pod

删除对象的最简单方法是按名称删除它。

删除单个 Pod

使用以下命令从集群中删除 kubia pod：

```
$ kubectl delete po
kubia pod "kubia" 已删除
```

通过删除 pod，您表明您不再希望该 pod 或其容器存在。 Kubelet 关闭 pod 的容器，删除所有关联的资源（例如日志文件），并在此过程完成后通知 API 服务器。然后 Pod 对象被删除。

提示 默认情况下，kubectl delete 命令会等待，直到对象不再存在。要跳过等待，请使用 --wait=false 选项运行命令

当 Pod 处于关闭过程中时，其状态会更改为 Termination：

```
$ kubectl get po kubia 名称就绪状态重新启动年龄
kubia 1/1 终止 0 35m
```

如果您希望应用程序为其客户提供良好的体验，那么准确了解容器如何关闭非常重要。这将在下一章中进行解释，我们将更深入地了解 Pod 及其容器的生命周期。

注意如果您熟悉 Docker，您可能想知道是否可以停止 pod 并稍后重新启动它，就像使用 Docker 容器一样。答案是不。使用 Kubernetes，您只能完全删除 pod，并且稍后再次创建。

DELETING MULTIPLE PODS WITH A SINGLE COMMAND

You can also delete multiple pods with a single command. If you ran the `kubia-init` and the `kubia-init-slow` pods, you can delete them both by specifying their names separated by a space, as follows:

```
$ kubectl delete po kubia-init kubia-init-slow
pod "kubia-init" deleted
pod "kubia-init-slow" deleted
```

5.6.2 Deleting objects defined in manifest files

Whenever you create objects from a file, you can also delete them by passing the file to the `delete` command instead of specifying the name of the pod.

DELETING OBJECTS BY SPECIFYING THE MANIFEST FILE

You can delete the `kubia-ssl` pod, which you created from the `kubia-ssl.yaml` file, with the following command:

```
$ kubectl delete -f kubia-ssl.yaml
pod "kubia-ssl" deleted
```

In your case, the file contains only a single pod object, but you'll typically come across files that contain several objects of different types that represent a complete application. This makes deploying and removing the application as easy as executing `kubectl apply -f app.yaml` and `kubectl delete -f app.yaml`, respectively.

DELETING OBJECTS FROM MULTIPLE MANIFEST FILES

Sometimes, an application is defined in several manifest files. You can specify multiple files by separating them with a comma. For example:

```
$ kubectl delete -f kubia.yaml,kubia-ssl.yaml
```

NOTE You can also apply several files at the same time using this syntax (for example: `kubectl apply -f kubia.yaml,kubia-ssl.yaml`).

I've never actually used this approach in the many years I've been using Kubernetes, but I often deploy all the manifest files from a file directory by specifying the directory name instead of the names of individual files. For example, you can deploy all the pods you created in this chapter again by running the following command in the base directory of this book's code archive:

```
$ kubectl apply -f Chapter05/
```

This applies all files in the directory that have the correct file extension (`.yaml`, `.json`, and others). You can then delete the pods using the same method:

```
$ kubectl delete -f Chapter05/
```


使用单个命令删除多个 Pod

您还可以使用单个命令删除多个 Pod。如果您运行了 kubia-init 和 kubia-init-slow pod，则可以通过指定以 分隔的名称来删除它们

空间，如下：

```
$ kubectl delete po kubia-init kubia-
init-slow pod“kubia-init”已删除
pod“kubia-init-slow”已删除
```

5.6.2 删除清单文件中定义的对象

每当您从文件创建对象时，您还可以通过将文件传递给删除命令来删除它们，而不是指定 pod 的名称。

通过指定清单文件删除对象

您可以使用以下命令删除从 kubia-ssl.yaml 文件创建的 kubia-ssl pod：

```
$ kubectl delete -f kubia-
ssl.yaml pod“kubia-ssl”已删
除
```

在您的例子中，该文件仅包含一个 pod 对象，但您通常会遇到包含多个代表完整应用程序的不同类型对象的文件。这使得部署和删除应用程序就像分别执行 kubectl apply -f app.yaml 和 kubectl delete -f app.yaml 一样简单。

从多个清单文件中删除对象

有时，一个应用程序是在多个清单文件中定义的。您可以通过用逗号分隔来指定多个文件。例如：

```
$ kubectl 删除 -f kubia.yaml,kubia-ssl.yaml

注意 您还可以使用此语法同时
应用多个文件（例如：kubectl
apply f kubia.yaml,kubia-
ssl.yaml）。
```

在使用 Kubernetes 的很多年里，我从未真正使用过这种方法，但我经常通过指定目录名称而不是单个文件的名称来部署文件目录中的所有清单文件。例如，您可以通过在本书代码存档的基目录中运行以下命令来再次部署您在本章中创建的所有 Pod：

```
$ kubectl apply -f Chapter05/
```

这适用于目录中具有正确文件扩展名（.yaml、.json 等）的所有文件。然后您可以使用相同的方法删除 Pod：

NOTE Use the `--recursive` flag to also scan subdirectories.

5.6.3 Deleting all pods

You've now removed all pods except `kubia-stdin` and the pods you created in chapter 3 using the `kubectl create deployment` command. Depending on how you've scaled the deployment, some of these pods should still be running:

```
$ kubectl get pods
NAME                  READY   STATUS    RESTARTS   AGE
kubia-stdin           1/1     Running   0          10m
kubia-9d785b578-58vhc 1/1     Running   0          1d
kubia-9d785b578-jmnj8 1/1     Running   0          1d
```

Instead of deleting these pods by name, we can delete them all using the `--all` option:

```
$ kubectl delete po --all
pod "kubia-stdin" deleted
pod "kubia-9d785b578-58vhc" deleted
pod "kubia-9d785b578-jmnj8" deleted
```

Now confirm that no pods exist by executing the `kubectl get pods` command again:

```
$ kubectl get po
NAME                  READY   STATUS    RESTARTS   AGE
kubia-9d785b578-cc6tk 1/1     Running   0          13s
kubia-9d785b578-h4gml 1/1     Running   0          13s
```

That was unexpected! Two pods are still running. If you look closely at their names, you'll see that these aren't the two you've just deleted. The `AGE` column also indicates that these are *new* pods. You can try to delete them as well, but you'll see that no matter how often you delete them, new pods are created to replace them.

The reason why these pods keep popping up is because of the Deployment object. The controller responsible for bringing Deployment objects to life must ensure that the number of pods always matches the desired number of replicas specified in the object. When you delete a pod associated with the Deployment, the controller immediately creates a replacement pod.

To delete these pods, you must either scale the Deployment to zero or delete the object altogether. This would indicate that you no longer want this deployment or its pods to exist in your cluster.

5.6.4 Deleting objects of most kinds

You can delete everything you've created so far - including the deployment, its pods, and the service - with the command shown in the next listing.

Listing 5.13 Deleting most objects regardless of type

```
$ kubectl delete all --all
pod "kubia-9d785b578-cc6tk" deleted
pod "kubia-9d785b578-h4gml" deleted
```


注意 使用 --recursive 标志还可以扫描子目录。

5.6.3 删除所有 Pod

现在，您已经删除了除 kubia-stdin 和您在第 3 章中使用 kubectl create 创建的 pod 之外的所有 pod 部署命令。取决于你如何扩展
部署时，其中一些 pod 应该仍在运行：

```
$ kubectl 获取
pod 名称                准备好 地位      重启       年龄
kubia 标准输入          1/1    跑步      0          10m
kubia-9d785b578-58vhc 1/1    跑步      0          1天
kubia-9d785b578-jmnj8 1/1    跑步      0          1天
```

我们可以使用 --all 选项将它们全部删除，而不是按名称删除这些 pod：

```
$ kubectl delete po --
all pod “kubia-
stdin” 已删除
pod “kubia-9d785b578-
```

现在再次执行 kubectl get pods 命令确认不存在 pod：

```
$ kubectl 获取
po NAME                 准备好 地位      重启       年龄
kubia-9d785b578-cc6tk 1/1    跑步      0          13秒
kubia-9d785b578-h4gml 1/1    跑步      0          13秒
```

这是出乎意料的！两个 Pod 仍在运行。如果你仔细观察他们的名字，你会发现这不是你刚刚删除的两个。 AGE 列还表明这些是新的 pod。您也可以尝试删除它们，但您会发现无论您删除它们的频率如何，都会创建新的 Pod 来替换它们。

这些 pod 不断弹出的原因是 Deployment 对象。负责使 Deployment 对象生效的控制器必须确保 Pod 数量始终与对象中指定的所需副本数量相匹配。当您删除与 Deployment 关联的 Pod 时，控制器会立即创建一个替换 Pod。

要删除这些 Pod，您必须将 Deployment 缩放为零或完全删除该对象。这表明您不再希望此部署或其 Pod 存在于集群中。

5.6.4 删除大多数类型的对象

您可以使用下一个清单中显示的命令删除迄今为止创建的所有内容 - 包括部署、其 Pod 和服务。

清单 5.13 删除大多数对象，无论其类型如何

```
$ kubectl delete all --
all pod “kubia-
9d785b578-cc6tk” 已删除
```

```
service "kubernetes" deleted
service "kubia" deleted
deployment.apps "kubia" deleted
replicaset.apps "kubia-9d785b578" deleted
```

The first `all` in the command indicates that you want to delete objects of all types. The `--all` option indicates that you want to delete all instances of each object type. You used this option in the previous section when you tried to delete all pods.

When deleting objects, `kubectl` prints the type and name of each deleted object. In the previous listing, you should see that it deleted the pods, the deployment, and the service, but also a so-called replica set object. You'll learn what this is in chapter 11, where we take a closer look at deployments.

You'll notice that the delete command also deletes the built-in `kubernetes` service. Don't worry about this, as the service is automatically recreated after a few moments.

Certain objects aren't deleted when using this method, because the keyword `all` does not include all object kinds. This is a precaution to prevent you from accidentally deleting objects that contain important information. The Event object kind is one example of this.

> **NOTE** You can specify multiple object types in the `delete` command. For example, you can use `kubectl delete events,all --all` to delete events along with all object kinds included in `all`.

5.7 Summary

In this chapter, you've learned:

- Pods run one or more containers as a co-located group. They are the unit of deployment and horizontal scaling. A typical container runs only one process. Sidecar containers complement the primary container in the pod.
- Containers should only be part of the same pod if they must run together. A frontend and a backend process should run in separate pods. This allows them to be scaled individually.
- When a pod starts, its init containers run one after the other. When the last init container completes, the pod's main containers are started. You can use an init container to configure the pod from within, delay startup of its main containers until a precondition is met, or notify an external service that the pod is about to start running.
- The `kubectl` tool is used to create pods, view their logs, copy files to/from their containers, execute commands in those containers and enable communication with individual pods during development.

In the next chapter, you'll learn about the lifecycle of the pod and its containers.

服务“kubernetes”已删除 服务“kubia”已删除

已删除deployment.apps“kubia”

命令中的第一个 all 表示要删除所有类型的对象。 -all 选项表示您要删除每种对象类型的所有实例。你用过这个

当您尝试删除所有 Pod 时，请选择上一节中的选项。

删除对象时，kubectl 会打印每个删除的对象的类型和名称。在前面的清单中，您应该看到它删除了 Pod、部署和服务，还删除了所谓的副本集对象。您将在第 11 章中了解这是什么，我们将在其中仔细研究部署。

您会注意到删除命令还会删除内置的 kubernetes 服务。不请担心这一点，因为该服务会在几分钟后自动重新创建。

使用此方法时，某些对象不会被删除，因为关键字 all 不包括所有对象类型。这是一项预防措施，可防止您意外删除包含重要信息的对象。 Event 对象类型就是这样的一个示例。

> 说明 删除命令中可以指定多种对象类型。例如，您可以使用 kubectl delete events,all --all 来删除事件以及 all 中包含的所有对象类型。

5.7 总结

在本章中，您学习了：

- Pod 作为同一位置组运行一个或多个容器。它们是部署和水平扩展的单位。典型的容器只运行一个进程。 Sidecar 容器是 Pod 中主容器的补充。
- 如果容器必须一起运行，则它们只能是同一 Pod 的一部分。前端和后端进程应该在单独的 Pod 中运行。这使得它们可以单独缩放。
- 当 Pod 启动时，其初始化容器会依次运行。当最后一个 init 容器完成时，pod 的主容器就会启动。您可以使用 init 容器从内部配置 pod，延迟启动其主容器直到满足前提条件，或者通知外部服务 pod 即将开始运行。
- kubectl 工具用于创建pod、查看其日志、将文件复制到容器或从其容器复制文件、在这些容器中执行命令以及在开发过程中启用与各个pod 的通信。

在下一章中，您将了解 Pod 及其容器的生命周期。

6

Managing the lifecycle of the Pod's containers

This chapter covers

- **Inspecting the pod's status**
- **Keeping containers healthy using liveness probes**
- **Using lifecycle hooks to perform actions at container startup and shutdown**
- **Understanding the complete lifecycle of the pod and its containers**

After reading the previous chapter, you should be able to deploy, inspect and communicate with pods containing one or more containers. In this chapter, you'll gain a much deeper understanding of how the pod and its containers operate.

6.1 Understanding the pod's status

After you create a pod object and it runs, you can see what's going on with the pod by reading the pod object back from the API. As you've learned in chapter 4, the pod object manifest, as well as the manifests of most other kinds of objects, contain a section, which provides the status of the object. A pod's `status` section contains the following information:

- the IP addresses of the pod and the worker node that hosts it
- when the pod was started
- the pod's quality-of-service (QoS) class
- what phase the pod is in,
- the conditions of the pod, and
- the state of its individual containers.

6

管理 Pod 的生命周期

容器

本章涵盖

- 检查 pod 的状态
- 使用活性探针保持容器健康
- 使用生命周期钩子在容器启动和关闭时执行操作
- 了解 Pod 及其容器的完整生命周期

阅读上一章后，您应该能够部署、检查包含一个或多个容器的 Pod 并与之通信。在本章中，您将更深入地了解 Pod 及其容器的运行方式。

6.1 了解 Pod 的状态

创建 pod 对象并运行后，您可以通过从 API 读回 pod 对象来查看 pod 发生了什么。正如您在第 4 章中了解到的，pod 对象清单以及大多数其他类型的对象的清单都包含一个部分，其中提供了对象的状态。 Pod 的状态部分包含以下信息：

- Pod 和托管它的工作节点的 IP 地址
- Pod 何时启动
- Pod 的服务质量 (QoS) 等级
- pod 处于哪个阶段，
- 集群的条件，以及

The IP addresses and the start time don't need any further explanation, and the QoS class isn't relevant now - you'll learn about it in chapter 19. However, the phase and conditions of the pod, as well as the states of its containers are important for you to understand the pod lifecycle.

6.1.1 Understanding the pod phase

In any moment of the pod's life, it's in one of the five phases shown in the following figure.

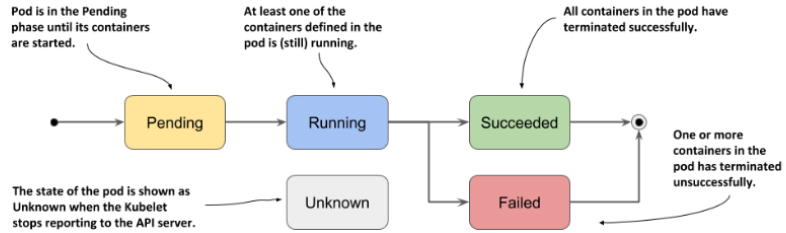


Figure 6.1 The phases of a Kubernetes pod

The meaning of each phase is explained in the following table.

Table 6.1 List of phases a pod can be in

Pod Phase	Description
`Pending`	After you create the Pod object, this is its initial phase. Until the pod is scheduled to a node and the images of its containers are pulled and started, it remains in this phase.
`Running`	At least one of the pod's containers is running.
`Succeeded`	Pods that aren't intended to run indefinitely are marked as `Succeeded` when all their containers complete successfully.
`Failed`	When a pod is not configured to run indefinitely and at least one of its containers terminates unsuccessfully, the pod is marked as Failed.
`Unknown`	The state of the pod is unknown because the Kubelet has stopped reporting communicating with the API server. Possibly the worker node has failed or has disconnected from the network.

The pod's phase provides a quick summary of what's happening with the pod. Let's deploy the `kubia` pod again and inspect its phase. Create the pod by applying the `kubia.yaml` manifest to your cluster again, as in the previous chapter:

```
$ kubectl apply -f kubia.yaml
```


IP 地址和开始时间不需要任何进一步的解释，并且 QoS 类别现在不相关 - 您将在第 19 章中了解它。但是，Pod 的阶段和条件以及状态它的容器对于您了解 Pod 生命周期非常重要。

6.1.1 了解 Pod 阶段

在 Pod 生命周期的任何时刻，它都处于下图所示的五个阶段之一。

图 6.1 Kubernetes Pod 的阶段

各阶段的含义如下表所示。

表 6.1 pod 可以处于的阶段列表

Pod 阶段	描述
`Pending`	创建 `Pod` 对象后，这是其初始阶段。在 `pod` 被调度到节点并且拉取并启动其容器的镜像之前，它一直处于此阶段。
`Running`	运行 `Pod` 的至少一个容器正在运行。
`Succeeded`	当所有容器成功完成时，不打算无限期运行的成功 `Pod` 会被标记为成功。
`Failed`	失败 当某个 `Pod` 未配置为无限期运行并且其至少一个容器未成功终止时，该 `Pod` 会被标记为“失败”。
`Unknown`	未知 `Pod` 的状态未知，因为 `Kubelet` 已停止报告与 `API` 服务器的通信。工作节点可能出现故障或与网络断开连接。

Pod 的阶段提供了 Pod 所发生情况的快速摘要。让我们再次部署 kubia pod 并检查其阶段。通过再次将 kubia.yaml 清单应用到集群来创建 pod，如上一章所示：

```
$ kubectl 应用 -f kubia
```

DISPLAYING A POD'S PHASE

The pod's phase is one of the fields in the pod object's `status` section. You can see it by displaying its manifest and optionally grepping the output to search for the field:

```
$ kubectl get po kubia -o yaml | grep phase
phase: Running
```

TIP Remember the `jq` tool? You can use it instead to print out the value of the `phase` field like this:

```
kubectl get po kubia -o json | jq .status.phase
```

You can also see the pod's phase using `kubectl describe`:

```
$ kubectl describe po kubia | grep Status:
Status:       Running
```

Although it may appear that the `STATUS` column displayed by `kubectl get pods` also shows the phase, this is only true for pods that are healthy:

```
$ kubectl get po kubia
NAME    READY   STATUS    RESTARTS   AGE
kubia   1/1     Running   0          40m
```

For unhealthy pods, the `STATUS` column indicates what's wrong with the pod. You'll see this later in this chapter.

6.1.2 Understanding pod conditions

The phase of a pod says little about the condition of the pod. You can learn more by looking at the pod's list of conditions, just as you did for the node object in the chapter 4. A pod's conditions indicate whether a pod has reached a certain state or not, and why that's the case.

In contrast to the phase, a pod has several conditions at the same time. Four condition *types* are known at the time of writing. They are explained in the following table.

Table 6.2 List of pod conditions

Pod Condition	Description
`PodScheduled`	Indicates whether or not the pod has been scheduled to a node.
`Initialized`	The pod's init containers have all completed successfully.
`ContainersReady`	All containers in the pod indicate that they are ready. This is a necessary but not sufficient condition for the entire pod to be ready.
`Ready`	The pod is ready to provide services to its clients. The containers in the pod and the pod's readiness gates are all reporting that they are ready. Note: this is explained in chapter 10.

显示 Pod 的相位

Pod 的阶段是 Pod 对象状态部分中的字段之一。您可以通过显示其清单并选择 grep 输出来搜索该字段来查看它：

```
$ kubectl 获取 po kubia -o yaml | grep
阶段 阶段： 运行
```

提示还记得 jq 工具吗？您可以使用它来打印相位字段的值，如下所示：

```
kubectl 获取 po kubia -o json | jq.状态.阶段
```

您还可以使用 kubectl describe 查看 pod 的阶段：

```
$ kubectl 描述 po kubia | grep 状态：
状态：       跑步
```

尽管 kubectl get pods 显示的 STATUS 列似乎也显示了阶段，但这仅适用于健康的 pod：

```
$ kubectl 获取 po
kubia 名称 准备好 地位     重启      年龄
库比亚 1/1   跑步     0         40m
```

对于不健康的 pod，STATUS 列会指示 pod 出现的问题。您将在本章后面看到这一点。

6.1.2 了解 Pod 状况

豆荚的阶段几乎不能说明豆荚的状况。您可以通过查看 pod 的条件列表来了解更多信息，就像在第 4 章中对节点对象所做的那样。pod 的条件指示 pod 是否已达到某种状态，以及为什么会出现这种情况。

与阶段相反，Pod 同时具有多个条件。四个条件
在撰写本文时类型是已知的。下表对它们进行了解释。

表 6.2 Pod 条件列表

Pod 状况	说明
`PodScheduled`	指示 pod 是否已调度到节点。
已初始化	Pod 的初始化容器已全部成功完成。
`ContainersReady`	Pod 中的所有容器都表明它们已准备就绪。这是整个 Pod 准备就绪的必要但非充分条件。
就绪	该 Pod 已准备好为其客户提供服务。 Pod 中的容器和 Pod 的就绪门都报告它们已准备就绪。注意：第 10 章对此进行了解释。

Each condition is either fulfilled or not. As you can see in the following figure, the `PodScheduled` and `Initialized` conditions start as unfulfilled, but are soon fulfilled and remain so throughout the life of the pod. In contrast, the `Ready` and `ContainersReady` conditions can change many times during the pod's lifetime.

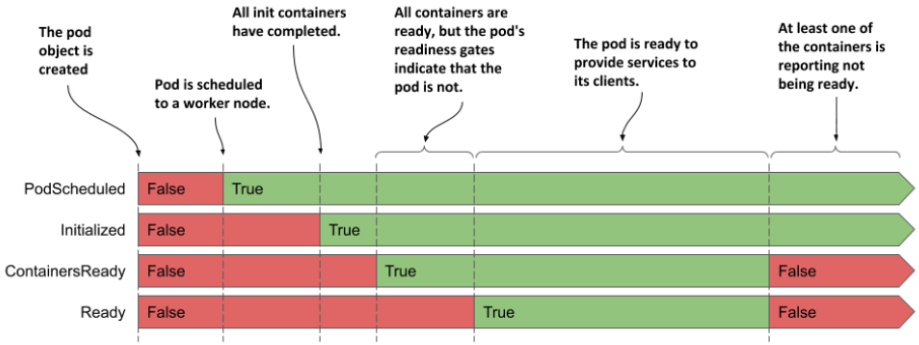


Figure 6.2 The transitions of the pod's conditions during its lifecycle

Do you remember the conditions you can find in a node object? They are `MemoryPressure`, `DiskPressure`, `PIDPressure` and `Ready`. As you can see, each object has its own set of condition types, but many contain the generic `Ready` condition, which typically indicates whether everything is fine with the object.

INSPECTING THE POD'S CONDITIONS

To see the conditions of a pod, you can use `kubectl describe` as in the next listing:

Listing 6.1 Displaying a pod's conditions using kubectl describe

```
$ kubectl describe po kubia | grep Conditions: -A5
Conditions:
  Type              Status
  Initialized       True        #A
  Ready             True        #B
  ContainersReady   True        #B
  PodScheduled      True        #C
```

#A The pod has been initialized
#B The pod and its containers are ready
#C The pod has been scheduled to a node

The `kubectl describe` command only shows whether each condition is true or not. To find out why a condition is false, you must inspect the pod manifest, as shown in the next listing.

每个条件要么满足，要么不满足。如下图所示，PodScheduled 和 Initialized 条件一开始未满足，但很快就满足并且

在 Pod 的整个生命周期中都保持这种状态。相比之下，Ready 和 ContainersReady 条件在 Pod 的生命周期内可能会发生多次变化。

图6.2 Pod生命周期内状态变化

您还记得可以在节点对象中找到的条件吗？它们是 MemoryPressure、DiskPressure、PIDPressure 和 Ready。正如你所看到的，每个对象都有自己的一组

条件类型，但许多都包含通用的就绪条件，该条件通常指示对象是否一切正常。

检查 Pod 的状况

要查看 pod 的状况，您可以使用 kubectl描述，如下列表所示：

清单 6.1 使用 kubectl describe 显示 pod 的状况

```
$ kubectl 描述 po kubia | grep 条件：-A5 条
件：类型
                  地位
  已初始化        真的        #A
  准备好          真的        #B
  容器就绪        真的        #B
  Pod调度         真的        #C
```

#A Pod已经初始化

#B pod 及其容器已准备就绪 #C

kubectl describe 命令仅显示每个条件是否为真。要找出条件为假的原因，您必须检查 pod 清单，如下一个清单所示。

Listing 6.2 Displaying a pod's conditions using kubectl and jq

```
$ kubectl get po kubia -o json | jq .status.conditions
[
  {
    "lastProbeTime": null,
    "lastTransitionTime": "2020-02-02T11:42:59Z",
    "status": "True",
    "type": "Initialized"
  },
  ...
```

Each condition has a `status` field that indicates whether the condition is `True`, `False` or `Unknown`. In the case of the `kubia` pod, the status of all conditions is `True`, which means they are all fulfilled. The condition can also contain a `reason` field that specifies a machine-facing reason for the last change of the condition's status, and a `message` field that explains the change in detail. The `lastTransitionTime` field shows when the change occurred, while the `lastProbeTime` indicates when this condition was last checked.

6.1.3 Understanding the status of the containers

Also contained in the status of the pod is the status of each of its containers. Inspecting the status provides better insight into the operation of each individual container.

The status contains several fields. The `state` field indicates the container's current state, whereas the `lastState` field shows the state of the previous container after it has terminated. The container status also indicates the internal ID of the container (`containerID`), the `image` and `imageID` the container is running, whether the container is `ready` or not and how often it has been restarted (`restartCount`).

UNDERSTANDING THE CONTAINER STATE

The most important part of a container's status is its `state`. A container can be in one of the states shown in the following figure.

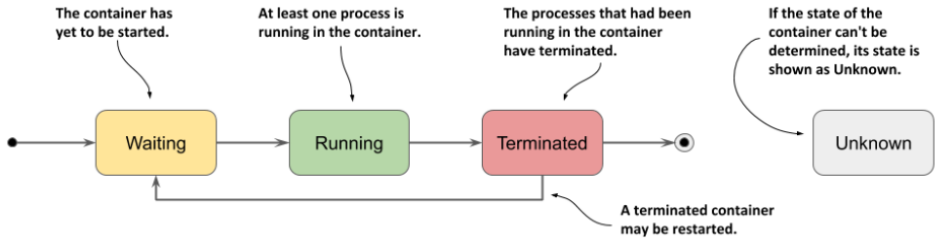


Figure 6.3 The possible states of a container

Individual states are explained in the following table.

清单 6.2 使用 kubectl 和 jq 显示 pod 的状况

```
$ kubectl 获取 po kubia -o json | jq .状态.条件
[
  {
    “最后探测时间”：空，
    “lastTransitionTime ”：“2020-02-
  02T11：42：59Z”，“状
],
```

每个条件都有一个状态字段，指示条件是 True、False 还是未知。对于 kubia pod，所有条件的状态都是 True，这意味着它们

都满足了。该条件还可以包含一个原因字段，指定面向机器的

条件状态上次更改的原因，以及详细解释更改的消息字段。 lastTransitionTime 字段显示更改发生的时间，而lastProbeTime 则指示上次检查此条件的时间。

6.1.3 了解容器的状态

Pod 的状态还包含其每个容器的状态。检查状态可以更好地了解每个容器的运行情况。

状态包含多个字段。 state 字段指示容器的当前状态，而 lastState 字段显示前一个容器终止后的状态。容器状态还指示容器的内部ID

(containerID)、容器正在运行的镜像和镜像ID、容器是否准备就绪以及重新启动的频率(restartCount)。

了解容器状态

容器状态最重要的部分是它的状态。容器可以处于下图所示的状态之一。

图6.3 容器可能的状态

下表解释了各个状态。

Table 6.3 Possible container states

Container State	Description
`Waiting`	The container is waiting to be started. The `reason` and `message` fields indicate why the container is in this state.
`Running`	The container has been created and processes are running in it. The `startedAt` field indicates the time at which this container was started.
`Terminated`	The processes that had been running in the container have terminated. The `startedAt` and `finishedAt` fields indicate when the container was started and when it terminated. The exit code with which the main process terminated is in the `exitCode` field.
`Unknown`	The state of the container couldn't be determined.

DISPLAYING THE STATUS OF THE POD'S CONTAINERS

The pod list displayed by `kubectl get pods` shows only the number of containers in each pod and how many of them are ready. To see the status of individual containers, you must use `kubectl describe`, as shown in the following listing.

Listing 6.3 Inspecting a container's status using kubectl describe

```
$ kubectl describe po kubia | grep Containers: -A15
Containers:
  kubia:
    Container ID:   docker://c64944a684d57faacfced0be1af44686...
    Image:          luksa/kubia:1.0
    Image ID:       docker-pullable://luksa/kubia@sha256:3f28...
    Port:           8080/TCP
    Host Port:      0/TCP
    State:          Running                          #A
      Started:      Sun, 02 Feb 2020 12:43:03 +0100   #A
    Ready:          True                             #B
    Restart Count:  0                                #C
    Environment:    <none>
    Mounts:
      ...
```

#A The current state of the container and when it was started
#B Whether the container is ready to provide its services
#C How many times the container has been restarted

Focus on the annotated lines in the listing, as they indicate whether the container is healthy. The `kubia` container is `Running` and is `Ready`. It has never been restarted.

TIP You can also display the container status(es) using `jq` like this: `kubectl get po kubia -o json | jq .status.containerStatuses`

表 6.3 可能的容器状态

容器状态	描述
等待	容器正在等待启动。原因和消息字段指示容器处于此状态的原因。
跑步	容器已创建并且进程正在其中运行。 `startedAt` 字段指示该容器启动的时间。
终止	容器中运行的进程已终止。 startedAt 和finishedAt 字段指示容器何时启动以及何时终止。主进程终止的退出代码位于 exitCode 字段中。
未知	无法确定容器的状态。

显示 Pod 容器的状态

kubectl get pods 显示的 pod 列表仅显示每个 pod 中的容器数量以及其中有多少容器已就绪。要查看各个容器的状态，您必须使用 kubectl describe，如以下清单所示。

清单 6.3 使用 kubectl describe 检查容器的状态

```
$ kubectl 描述 po kubia | grep 容器：-A15 容
器：kubia：
    容器 ID：
    docker：//c64944a684d57faacfced0be1af44686 ...
    图片：luksa/kubia：1.0
    图片 ID：docker-pullable：//luksa/kubia@sha256：3f28...
    开始时间：2020 年 2 月 2 日星期日 12:43:03
    准备好：正确#B
    重新启动计数：0#C
    环境：<无>
    ...
```

#A 容器的当前状态以及启动时间 #B 容器是否准备好提供服务 #C 容器已重启了多少次

重点关注列表中带注释的行，因为它们表明容器是否正常。 kubia 容器正在运行并且已准备就绪。它从未被重新启动过。

提示您还可以使用 jq 显示容器状态，如下所示： `kubectl get po kubia -o json | jq .status.containerStatuses`

INSPECTING THE STATUS OF AN INIT CONTAINER

In the previous chapter, you learned that in addition to regular containers, a pod can also have init containers that run when the pod starts. As with regular containers, the status of these containers is available in the `status` section of the pod object manifest, but in the `initContainerStatuses` field.

Inspecting the status of the kubia-init pod

As an additional exercise you can try on your own, create the pod defined in the `kubia-init.yaml` file from the previous chapter and inspect its phase, conditions and the status of its two regular and two init containers using `kubectl describe` and by retrieving the pod manifest using the `kubectl get po kubia-init -o json | jq .status` command.

6.2 Keeping containers healthy

The pods you created in the previous chapter ran without any problems. But what if one of the containers dies? What if all the containers in a pod die? How do you keep the pods healthy and their containers running? That's the focus of this section.

6.2.1 Understanding container auto-restart

When a pod is scheduled to a node, the Kubelet on that node starts its containers and from then on keeps them running for as long as the pod object exists. If the main process in the container terminates for any reason, the Kubelet restarts the container. If an error in your application causes it to crash, Kubernetes automatically restarts it, so even without doing anything special in the application itself, running it in Kubernetes automatically gives it the ability to heal itself. Let's see this in action.

OBSERVING A CONTAINER FAILURE

In the previous chapter, you created the `kubia-ssl` pod, which contains the Node.js and the Envoy containers. Create the pod again and enable communication with the pod by running the following two commands:

```
$ kubectl apply -f kubia-ssl.yaml
$ kubectl port-forward kubia-ssl 8080 8443 9901
```

You'll now cause the Envoy container to terminate to see how Kubernetes deals with the situation. Run the following command in a separate terminal so you can see how the pod's status changes when one of its containers terminates:

```
$ kubectl get pods -w
```

You'll also want to watch events in another terminal using the following command:

```
$ kubectl get events -w
```


检查 INIT 容器的状态

在上一章中，您了解到除了常规容器之外，Pod 还可以具有在 Pod 启动时运行的 init 容器。与常规容器一样，这些容器的状态可在 pod 对象清单的状态部分中找到，但在 initContainerStatuses 字段中可用。

检查 kubia-init pod 的状态

作为附加练习，您可以自己尝试，创建上一章中 kubia-init.yaml 文件中定义的 pod，并使用以下命令检查其阶段、条件以及两个常规容器和两个 init 容器的状态

kubectl 描述并使用 kubectl get po kubia-init -o json | 检索 pod 清单

jq .status 命令

6.2 保持容器健康

您在上一章中创建的 Pod 运行没有任何问题。但如果其中一个容器死掉了怎么办？如果 pod 中的所有容器都死掉了怎么办？如何保持 Pod 健康及其容器运行？这是本节的重点。

6.2.1 了解容器自动重启

当 pod 被调度到节点时，该节点上的 Kubelet 会启动其容器，并从那时起，只要 pod 对象存在，它们就会保持运行。如果容器中的主进程因任何原因终止，Kubelet 会重新启动容器。如果应用程序中的错误导致其崩溃，Kubernetes 会自动重新启动它，因此即使应用程序本身没有执行任何特殊操作，在 Kubernetes 中运行它也会自动赋予其自我修复的能力。让我们看看实际效果。

观察容器故障

在上一章中，您创建了 kubia-ssl pod，其中包含 Node.js 和 Envoy 容器。再次创建 Pod，并通过运行以下两个命令启用与 Pod 的通信：

```
$ kubectl apply -f kubia-
ssl.yaml $ kubectl 端口转发
kubia-ssl 8080 8443 9901
```

现在，您将导致 Envoy 容器终止，以查看 Kubernetes 如何处理这种情况。在单独的终端中运行以下命令，以便您可以看到当其中一个容器终止时 pod 的状态如何变化：

```
$ kubectl 获取 pod -w
```

您还需要使用以下命令在另一个终端中观看事件：

You could emulate a crash of the container's main process by sending it the `KILL` signal, but you can't do this from inside the container because the Linux Kernel doesn't let you kill the root process (the process with PID 1). You would have to SSH to the pod's host node and kill the process from there. Fortunately, Envoy's administration interface allows you to stop the process via its HTTP API.

To terminate the `envoy` container, open the URL http://localhost:9901 in your browser and click the *quitquitquit* button or run the following `curl` command in another terminal:

```
$ curl -X POST http://localhost:9901/quitquitquit
OK
```

To see what happens with the container and the pod it belongs to, examine the output of the `kubectl get pods -w` command you ran earlier. It's shown in the next listing.

Listing 6.4 Pod state transitions when a container terminates

```
$ kubectl get po -w
NAME          READY   STATUS    RESTARTS   AGE
kubia-ssl     2/2     Running   0          1s
kubia-ssl     1/2     NotReady  0          9m33s
kubia-ssl     2/2     Running   1          9m34s
```

The listing shows that the pod's `STATUS` changes from `Running` to `NotReady`, while the `READY` column indicates that only one of the two containers is ready. Immediately thereafter, Kubernetes restarts the container and the pod's status returns to `Running`. The `RESTARTS` column indicates that one container has been restarted.

NOTE If one of the pod's containers fails, the other containers continue to run.

Now examine the output of the `kubectl get events -w` command you ran earlier. It is shown in the next listing.

Listing 6.6 Events emitted when a container terminates

```
$ kubectl get ev -w
LAST SEEN   TYPE      REASON      OBJECT           MESSAGE
0s          Normal    Pulled      pod/kubia-ssl    Container image already
                                                   present on machine
0s          Normal    Created     pod/kubia-ssl    Created container envoy
0s          Normal    Started     pod/kubia-ssl    Started container envoy
```

The events show that the new `envoy` container has been started. You should be able to access the application via HTTPS again. Please confirm with your browser or `curl`.

The events in the listing also expose an important detail about how Kubernetes restarts containers. The second event indicates that the entire `envoy` container has been recreated. Kubernetes never restarts a container, but instead discards it and creates a new container. Regardless, we call this *restarting* a container.

149

您可以通过向容器的主进程发送 KILL 信号来模拟容器主进程的崩溃，但您不能从容器内部执行此操作，因为 Linux 内核不允许您终止根进程（PID 为 1 的进程）。您必须通过 SSH 连接到 pod 的主机节点并从那里终止进程。幸运的是，Envoy 的管理界面允许您通过其 HTTP API 停止该进程。

要终止 Envoy 容器，请在浏览器中打开 URL http://localhost:9901 并单击 quitquitquit 按钮或在另一个终端中运行以下curl 命令：

```
$curl -X POST
http://localhost:9901/quitquitquit 确定
```

要查看容器及其所属 pod 发生的情况，请检查您之前运行的 kubectl get pods -w 命令的输出。它显示在下一个列表中。

清单 6.4 容器终止时 Pod 状态转换

```
$ kubectl get po -w 名称就绪状态重新启动年
龄
kubia-ssl 2/2 运行            0 1秒
kubia-ssl 1/2 未就绪          0 9分33秒
kubia-ssl 2/2 运行            1 9分34秒
```

该列表显示 Pod 的 STATUS 从 Running 更改为 NotReady，而 READY 列表明两个容器中只有一个已就绪。此后，Kubernetes 立即重新启动容器，Pod 的状态恢复为 Running。RESTARTS 列表示一个容器已重新启动。

注意：如果 pod 的一个容器发生故障，其他容器将继续运行。

现在检查您之前运行的 kubectl get events -w 命令的输出。它显示在下一个列表中。

清单 6.6 容器终止时发出的事件

```
$ kubectl get ev -w 最后看到的类型原因对象消息
0s 正常 拉取 pod/kubia-ssl 容器镜像已存在于机器上
0s 正常 创建 pod/kubia-ssl 创建容器 envoy
0s 正常启动 pod/kubia-ssl 启动容器 envoy
```

这些事件表明新的 Envoy 容器已启动。您应该能够再次通过 HTTPS 访问该应用程序。请使用浏览器或curl确认。

列表中的事件还揭示了有关 Kubernetes 如何重新启动容器的重要细节。第二个事件表明整个 Envoy 容器已被重新创建。 Kubernetes 永远不会重新启动容器，而是丢弃它并创建一个新容器。无论如何，我们称之为重新启动容器。

NOTE Any data that the process writes to the container's filesystem is lost when the container is recreated. This behavior is sometimes undesirable. To persist data, you must add a storage volume to the pod, as explained in the next chapter.

NOTE If init containers are defined in the pod and one of the pod's regular containers is restarted, the init containers are not executed again.

CONFIGURING THE POD'S RESTART POLICY

By default, Kubernetes restarts the container regardless of whether the process in the container exits with a zero or non-zero exit code - in other words, whether the container completes successfully or fails. This behavior can be changed by setting the `restartPolicy` field in the pod's `spec`.

Three restart policies exist. They are explained in the following figure.

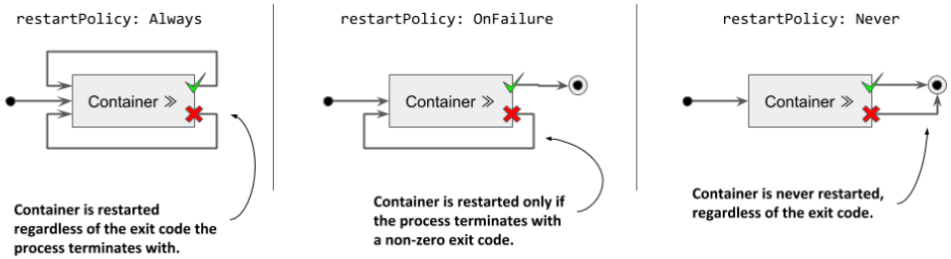


Figure 6.4 The pod's restartPolicy determines whether its containers are restarted or not

The following table describes the three restart policies.

Table 6.4 Pod restart policies

Restart Policy	Description
`Always`	Container is restarted regardless of the exit code the process in the container terminates with. This is the default restart policy.
`OnFailure`	The container is restarted only if the process terminates with a non-zero exit code, which by convention indicates failure.
`Never`	The container is never restarted - not even when it fails.

NOTE Surprisingly, the restart policy is configured at the pod level and applies to all its containers. It can't be configured for each container individually.

注意重新创建容器时，进程写入容器文件系统的任何数据都会丢失。
这种行为有时是不可取的。要持久保存数据，您必须向 Pod 添加存储卷，如下所示
在下一章中解释

注意：如果 Pod 中定义了 init 容器，并且重新启动了 pod 的常规容器之一，则 init 容器不会再次执行。

配置 Pod 的重启策略

默认情况下，无论容器中的进程是否以零或非零退出代码退出，Kubernetes 都会重新启动容器 - 换句话说，无论容器成功完成还是失败。可以通过在 Pod 规范中设置 restartPolicy 字段来更改此行为。

存在三种重启策略。下图对其进行了解释。

图6.4 Pod的restartPolicy决定其容器是否重启

下表描述了三种重启策略。

表 6.4 Pod 重启策略

重启策略说明
始终 重新启动容器，无论容器中的进程以什么退出代码终止。这是默认的重启策略。

`OnFailure` 仅当进程以非零退出代码终止时，容器才会重新启动，按照惯例，该退出代码表示失败。
从不 容器永远不会重新启动 – 即使它失败了。

注意 令人惊讶的是，重启策略是在 Pod 级别配置的，并适用于其所有容器。不能为每个容器单独配置。

UNDERSTANDING THE TIME DELAY INSERTED BEFORE A CONTAINER IS RESTARTED

If you call Envoy's `/quitquitquit` endpoint several times, you'll notice that each time it takes longer to restart the container after it terminates. The pod's status is displayed as either `NotReady` or `CrashLoopBackOff`. Here's what it means.

As shown in the following figure, the first time a container terminates, it is restarted immediately. The next time, however, Kubernetes waits ten seconds before restarting it again. This delay is then doubled to 20, 40, 80 and then to 160 seconds after each subsequent termination. From then on, the delay is kept at five minutes. This delay that doubles between attempts is called exponential back-off.

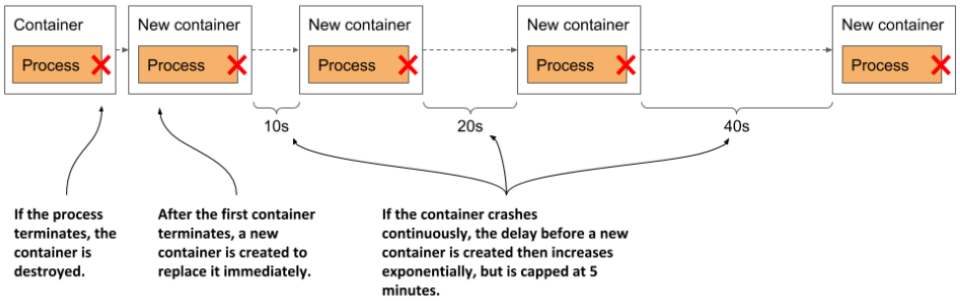


Figure 6.5 Exponential back-off between container restarts

In the worst case, a container can therefore be prevented from starting for up to five minutes.

> **NOTE** The delay is reset to zero when the container has run successfully for 10 minutes. If the container must be restarted later, it is restarted immediately.

As you can see in the following listing, the container is in the `Waiting` state while it waits to be restarted, and the `reason` is shown as `CrashLoopBackOff`. The `message` field indicates how long it will take for the container to be restarted.

Listing 6.7 The state of a container that's waiting to be restarted

```
$ kubectl get po kubia-ssl -o json | jq .status.containerStatuses
...
"state": {
  "waiting": {
    "message": "back-off 40s restarting failed container=envoy ...",
    "reason": "CrashLoopBackOff"
```

了解容器重新启动之前插入的时间延迟

如果您多次调用 Envoy 的 /quitquitquit 端点，您会注意到每次容器终止后重新启动都需要更长的时间。 Pod 的状态显示为 NotReady 或 CrashLoopBackOff。这就是它的意思。

如下图，容器第一次终止时，会重新启动
立即地。然而，下一次，Kubernetes 会等待十秒然后重新启动它

再次。然后，在每次后续终止后，此延迟会加倍到 20、40、80 秒，然后是 160 秒。从此以后，延迟时间保持在五分钟。这种在尝试之间加倍的延迟称为指数退避。

图 6.5 容器重新启动之间的指数退避

在最坏的情况下，容器可能会被阻止启动长达五分钟。

注意 当容器成功运行 10 分钟后，延迟将重
置为零。如果稍后必须重新启动容器，则会
立即重新启动。

正如您在下面的清单中看到的，容器在等待时处于等待状态

重启，原因显示为CrashLoopBackOff。消息字段指示容器重新启动需要多长时间。

清单 6.7 等待重新启动的容器的状态

```
$ kubectl 获取 po kubia-ssl -o json | jq
.status.containerStatuses
  "等待": {
    "message": "后退 40 秒重新启动失败的容器=envoy ...
```

NOTE When you tell Envoy to terminate, it terminates with exit code zero, which means it hasn't crashed. The `CrashLoopBackOff` status can therefore be misleading.

6.2.2 Checking the container's health using liveness probes

In the previous section, you learned that Kubernetes keeps your application healthy by restarting it when its process terminates. But applications can also become unresponsive without terminating. For example, a Java application with a memory leak eventually starts spewing out OutOfMemoryErrors, but its JVM process continues to run. Ideally, Kubernetes should detect this kind of error and restart the container.

The application could catch these errors by itself and immediately terminate, but what about the situations where your application stops responding because it gets into an infinite loop or deadlock? What if the application can't detect this? To ensure that the application is restarted in such cases, it may be necessary to check its state from the outside.

INTRODUCING LIVENESS PROBES

Kubernetes can be configured to check whether an application is still alive by defining a *liveness probe*. You can specify a liveness probe for each container in the pod. Kubernetes runs the probe periodically to ask the application if it's still alive and well. If the application doesn't respond, an error occurs, or the response is negative, the container is considered unhealthy and is terminated. The container is then restarted if the restart policy allows it.

NOTE Liveness probes can only be used in the pod's regular containers. They can't be defined in init containers.

TYPES OF LIVENESS PROBES

Kubernetes can probe a container with one of the following three mechanisms:

- An *HTTP GET* probe sends a GET request to the container's IP address, on the network port and path you specify. If the probe receives a response, and the response code doesn't represent an error (in other words, if the HTTP response code is `2xx` or `3xx`), the probe is considered successful. If the server returns an error response code, or if it doesn't respond in time, the probe is considered to have failed.
- A *TCP Socket* probe attempts to open a TCP connection to the specified port of the container. If the connection is successfully established, the probe is considered successful. If the connection can't be established in time, the probe is considered failed.
- An *Exec* probe executes a command inside the container and checks the exit code it terminates with. If the exit code is zero, the probe is successful. A non-zero exit code is considered a failure. The probe is also considered to have failed if the command fails to terminate in time.

NOTE In addition to a liveness probe, a container may also have a *startup* probe, which is discussed in section 6.2.6, and a *readiness* probe, which is explained in chapter 10.

注意当您告诉 Envoy 终止时，它会以退出代码零终止，这意味着它没有崩溃。因此，CrashLoopBackOff 状态可能会产生误导。

6.2.2 使用活性探针检查容器的健康状况

在上一节中，您了解到 Kubernetes 通过在进程终止时重新启动应用程序来保持应用程序的健康。但应用程序也可能变得无响应而不终止。例如，存在内存泄漏的 Java 应用程序最终开始抛出 OutOfMemoryErrors，但其 JVM 进程仍继续运行。理想情况下，Kubernetes 应该检测到这种错误并重新启动容器。

应用程序可以自行捕获这些错误并立即终止，但是应用程序由于陷入无限循环或死锁而停止响应的情况该怎么办？如果应用程序无法检测到这一点怎么办？为了确保应用程序在这种情况下重新启动，可能需要从外部检查其状态。

介绍活性探针

可以将 Kubernetes 配置为通过定义活性探针来检查应用程序是否仍然存在。您可以为 pod 中的每个容器指定一个活性探针。 Kubernetes 定期运行探测器来询问应用程序是否仍然存在且运行良好。如果应用程序没有响应、发生错误或响应是否定的，则容器被认为不健康并被终止。如果重启策略允许，容器就会重新启动。

注意 活性探针只能在 Pod 的常规容器中使用。它们不能在 init 容器中定义。

活性探针的类型

Kubernetes 可以使用以下三种机制之一来探测容器：

- HTTP GET 探针在您指定的网络端口和路径上向容器的 IP 地址发送 GET 请求。如果探针收到响应，并且响应代码不代表错误（换句话说，如果 HTTP 响应代码为 2xx 或 3xx），则认为探针成功。如果服务器返回错误响应码，或者没有及时响应，则认为探测失败。
- TCP 套接字探测尝试打开到容器指定端口的TCP 连接。如果连接成功建立，则认为探测成功。如果不能及时建立连接，则认为探测失败。
- Exec 探针在容器内执行命令并检查其终止的退出代码。如果退出代码为零，则探测成功。非零退出代码被视为失败。如果命令未能及时终止，则探测也被视为失败。

注：除了活性探针之外，容器还可能具有启动探针（第 6.2.6 节中讨论）和就绪探针（第 10 章中说明）。

6.2.3 Creating an HTTP GET liveness probe

Let's look at how to add a liveness probe to each of the containers in the `kubia-ssl` pod. Because they both run applications that understand HTTP, it makes sense to use an HTTP GET probe in each of them. The Node.js application doesn't provide any endpoints to explicitly check the health of the application, but the Envoy proxy does. In real-world applications, you'll encounter both cases.

DEFINING LIVENESS PROBES IN THE POD MANIFEST

The following listing shows an updated manifest for the pod, which defines a liveness probe for each of the two containers, with different levels of configuration.

Listing 6.8 Adding a liveness probe to a pod: kubia-liveness.yaml

```
apiVersion: v1
kind: Pod
metadata:
  name: kubia-liveness
spec:
  containers:
  - name: kubia
    image: luksa/kubia:1.0
    ports:
    - name: http
      containerPort: 8080
    livenessProbe:                          #A
      httpGet:                              #A
        path: /                             #A
        port: 8080                          #A
  - name: envoy
    image: luksa/kubia-ssl-proxy:1.0
    ports:
    - name: https
      containerPort: 8443
    - name: admin
      containerPort: 9901
    livenessProbe:                          #B
      httpGet:                              #B
        path: /ready                        #B
        port: admin                         #B
      initialDelaySeconds: 10               #B
      periodSeconds: 5                      #B
      timeoutSeconds: 2                     #B
      failureThreshold: 3                   #B
```

#A The liveness probe definition for the container running Node.js
#B The liveness probe for the Envoy proxy

These liveness probes are explained in the next two sections.

DEFINING A LIVENESS PROBE USING THE MINIMUM REQUIRED CONFIGURATION

The liveness probe for the `kubia` container is the simplest version of a probe for HTTP-based applications. The probe simply sends an HTTP `GET` request for the path `/` on port `8080` to

6.2.3 创建 HTTP GET 活跃度探针

让我们看看如何向 kubia-ssl pod 中的每个容器添加活性探针。因为它们都运行理解 HTTP 的应用程序，所以在每个应用程序中使用 HTTP GET 探针是有意义的。 Node.js 应用程序不提供任何端点来显式检查应用程序的运行状况，但 Envoy 代理提供。在实际应用中，您会遇到这两种情况。

在 POD 清单中定义活性探针

以下清单显示了 Pod 的更新清单，它为两个容器中的每一个定义了一个具有不同配置级别的活动探针。

清单 6.8 向 pod 添加活性探针：kubia-liveness.yaml

```
api 版本：v1
种类：Pod 元
数据
  名称：kubia-liveness
规格：
  容器：  -
  名称：
    图片：
kubia
    luksa/kubia:1.0 端
    口 - 名称：http
    容器端口：8080
    活性探针：#A
      http获取：#A
        路径：/#A

  - 姓名：特使
    图片：luksa/kubia-ssl-
    proxy:1.0 端口：-名称：https

      集装箱端口：8443
    - 姓名：管理员
      集装箱端口：9901
    活性探针：#B
      http获取：#B
        路径：/准备#B

      初始延迟秒：10 #B

      periodSeconds：5 #B

      超时秒数：2 #B
```

#A 运行 Node.js 的容器的活性探针定义 #B Envoy 代理的活性探针

这些活性探针将在接下来的两节中进行解释。

使用所需的最低配置定义活性探针

kubia 容器的活性探针是基于 HTTP 的应用程序探针的最简单版本

determine if the container can still serve requests. If the application responds with an HTTP status between `200` and `399`, the application is considered healthy.

The probe doesn't specify any other fields, so the default settings are used. The first request is sent 10s after the container starts and is repeated every 10s. If the application doesn't respond within one second, the probe attempt is considered failed. If it fails three times in a row, the container is considered unhealthy and is terminated.

UNDERSTANDING LIVENESS PROBE CONFIGURATION OPTIONS

The administration interface of the Envoy proxy provides the special endpoint `/ready` through which it exposes its health status. Instead of targeting port `8443`, which is the port through which Envoy forwards HTTPS requests to Node.js, the liveness probe for the `envoy` container targets this special endpoint on the `admin` port, which is port number `9901`.

> **NOTE** As you can see in the `envoy` container's liveness probe, you can specify the probe's target port by name instead of by number.

The liveness probe for the `envoy` container also contains additional fields. These are best explained with the following figure.

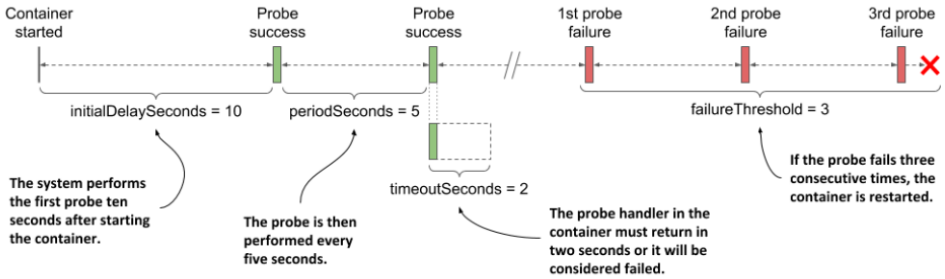


Figure 6.6 The configuration and operation of a liveness probe

The parameter `initialDelaySeconds` determines how long Kubernetes should delay the execution of the first probe after starting the container. The `periodSeconds` field specifies the amount of time between the execution of two consecutive probes, whereas the `timeoutSeconds` field specifies how long to wait for a response before the probe attempt counts as failed. The `failureThreshold` field specifies how many times the probe must fail for the container to be considered unhealthy and potentially restarted.

6.2.4 Observing the liveness probe in action

To see Kubernetes restart a container when its liveness probe fails, create the pod from the `kubia-liveness.yaml` manifest file using `kubectl apply`, and run `kubectl port-forward`

确定容器是否仍然可以服务请求。如果应用程序响应的 HTTP 状态在 200 到 399 之间，则该应用程序被认为是健康的。

探测器未指定任何其他字段，因此使用默认设置。第一个请求在容器启动后 10 秒发送，每 10 秒重复一次。如果应用程序在一秒内没有响应，则探测尝试被视为失败。如果连续失败 3 次，则容器被认为不健康并被终止。

了解活性探针配置选项

Envoy 代理的管理界面提供特殊端点 /ready

通过它暴露其健康状况。 Envoy 的活性探测不是以端口 8443（Envoy 将 HTTPS 请求转发到 Node.js）为目标

容器以管理端口（端口号 9901）上的此特殊端点为目标。

注意 正如您在 Envoy 容器的 liveness 探针中看到的，您可以按名称而不是按数字指定探针的目标端口。

Envoy 容器的活性探针还包含其他字段。下图可以很好地解释这些内容。

图 6.6 活性探针的配置和操作

参数initialDelaySeconds决定了Kubernetes在启动容器后应该延迟多长时间执行第一个探测。periodSeconds 字段指定两次连续探测执行之间的时间量，而 timeoutSeconds 字段指定在尝试探测之前等待响应的时间

算作失败。 failureThreshold 字段指定探测器必须失败多少次才能将容器视为不健康并可能重新启动。

6.2.4 观察活动探针的运行情况

要查看 Kubernetes 在活动探测失败时重新启动容器，请使用 kubectl apply 从 kubia-liveness.yaml 清单文件创建 pod

to enable communication with the pod. You'll need to stop the `kubectl port-forward` command still running from the previous exercise. Confirm that the pod is running and is responding to HTTP requests.

OBSERVING A SUCCESSFUL LIVENESS PROBE

The liveness probes for the pod's containers starts firing soon after the start of each individual container. Since the processes in both containers are healthy, the probes continuously report success. As this is the normal state, the fact that the probes are successful is not explicitly indicated anywhere in the status of the pod nor in its events.

The only indication that Kubernetes is executing the probe is found in the container logs. The Node.js application in the `kubia` container prints a line to the standard output every time it handles an HTTP request. This includes the liveness probe requests, so you can display them using the following command:

```
$ kubectl logs kubia-liveness -c kubia -f
```

The liveness probe for the `envoy` container is configured to send HTTP requests to Envoy's administration interface, which doesn't log HTTP requests to the standard output, but to the file `/var/log/envoy.admin.log` in the container's filesystem. To display the log file, you use the following command:

```
$ kubectl exec kubia-liveness -c envoy -- tail -f /var/log/envoy.admin.log
```

OBSERVING THE LIVENESS PROBE FAIL

A successful liveness probe isn't interesting, so let's cause Envoy's liveness probe to fail. To see what will happen behind the scenes, start watching events by executing the following command in a separate terminal:

```
$ kubectl get events -w
```

Using Envoy's administration interface, you can configure its health check endpoint to succeed or fail. To make it fail, open URL http://localhost:9901 in your browser and click the *healthcheck/fail* button, or use the following `curl` command:

```
$ curl -X POST localhost:9901/healthcheck/fail
```

Immediately after executing the command, observe the events that are displayed in the other terminal. When the probe fails, a `Warning` event is recorded, indicating the error and the HTTP status code returned:

```
Warning  Unhealthy  Liveness probe failed: HTTP probe failed with code 503
```

Because the probe's `failureThreshold` is set to three, a single failure is not enough to consider the container unhealthy, so it continues to run. You can make the liveness probe succeed again by clicking the *healthcheck/ok* button in Envoy's admin interface, or by using `curl` as follows:

```
$ curl -X POST localhost:9901/healthcheck/ok
```


启用与 Pod 的通信。您需要停止 kubectl 端口转发命令仍在运行上一个练习中的命令。确认 Pod 正在运行并且正在响应 HTTP 请求。

观察成功的活性探测

Pod 容器的活性探测器在每个容器启动后不久就会开始触发。由于两个容器中的进程均正常，因此探测器会不断报告成功。由于这是正常状态，因此 Pod 状态及其事件中的任何位置都没有明确指示探测成功的事实。

Kubernetes 正在执行探测的唯一指示是在容器日志中找到的。 kubia 容器中的 Node.js 应用程序每次处理 HTTP 请求时都会将一行打印到标准输出。这包括活性探测请求，因此您可以使用以下命令显示它们：

```
$ kubectl 日志 kubia-liveness -c kubia -f
```

Envoy 容器的 liveness 探针配置为将 HTTP 请求发送到 Envoy 的管理界面，该界面不会将 HTTP 请求记录到标准输出，而是记录到容器文件系统中的 /var/log/envoy.admin.log 文件。要显示日志文件，请使用以下命令：

```
$ kubectl exec kubia-liveness -c envoy -- tail -f /var/log/envoy.admin.log
```

观察活性探针失败

成功的活跃度探测并不有趣，所以让我们让 Envoy 的活跃度探测失败。要查看幕后会发生什么，请通过在单独的终端中执行以下命令来开始观察事件：

```
$ kubectl 获取事件 -w
```

使用 Envoy 的管理界面，您可以将其健康检查端点配置为成功或失败。要使其失败，请在浏览器中打开 URL http://localhost:9901 并单击 healthcheck/fail 按钮，或使用以下curl 命令：

```
$curl -X POST localhost :9901/healthcheck /fail
```

执行命令后，立即观察另一个终端中显示的事件。当探测失败时，会记录一个警告事件，指示错误并返回 HTTP 状态代码：

```
警告 不健康 的活动探测失败：HTTP 探测失败，代码为 503
```

由于探测器的 failureThreshold 设置为 3，单个故障不足以认为容器不健康，因此它会继续运行。您可以通过单击 Envoy 管理界面中的 healthcheck/ok 按钮或使用curl 使活性探测再次成功，如下所示：

If you are fast enough, the container won't be restarted.

OBSERVING THE LIVENESS PROBE REACH THE FAILURE THRESHOLD

If you let the liveness probe fail multiple times, you should see events like the ones in the next listing (note that some columns are omitted due to page width constraints).

Listing 6.9 Events recorded when a liveness probe fails

```
$ kubectl get events -w
TYPE     REASON     MESSAGE
Warning  Unhealthy  Liveness probe failed: HTTP probe failed with code 503
Warning  Unhealthy  Liveness probe failed: HTTP probe failed with code 503
Warning  Unhealthy  Liveness probe failed: HTTP probe failed with code 503
Normal   Killing    Container envoy failed liveness probe, will be
                    restarted
Normal   Pulled     Container image already present on machine
Normal   Created    Created container envoy
Normal   Started    Started container envoy
```

Remember that the probe failure threshold is set to three, so when the probe fails three times in a row, the container is stopped and restarted. This is indicated by the events in the listing.

The `kubectl get pods` command shows that the container has been restarted:

```
$ kubectl get po kubia-liveness
NAME             READY   STATUS    RESTARTS   AGE
kubia-liveness   2/2     Running   1          5m
```

The `RESTARTS` column shows that one container restart has taken place in the pod.

UNDERSTANDING HOW A CONTAINER THAT FAILS ITS LIVENESS PROBE IS RESTARTED

If you're wondering whether the main process in the container was gracefully stopped or killed forcibly, you can check the pod's status by retrieving the full manifest using `kubectl get` or using `kubectl describe` as shown in the following listing.

Listing 6.10 Inspecting the restarted container's last state with kubectl describe

```
$ kubectl describe po kubia-liveness
Name:           kubia-liveness
...
Containers:
  ...
  envoy:
    ...
    State:          Running                                 #A
      Started:      Sun, 31 May 2020 21:33:13 +0200         #A
    Last State:     Terminated                              #B
      Reason:       Completed                               #B
      Exit Code:    0                                       #B
      Started:      Sun, 31 May 2020 21:16:43 +0200         #B
      Finished:     Sun, 31 May 2020 21:33:13 +0200         #B
    ...
```


如果你足够快，容器将不会重新启动。

观察活性探针是否达到故障阈值

如果让活性探针多次失败，您应该会看到类似下一个清单中的事件（请注意，由于页面宽度限制，某些列被省略）。

清单 6.9 活性探测失败时记录的事件

```
$ kubectl 获取事件 -w
TYPE     原因       信息
警告     不良       活动探测失败：HTTP 探测失败，代码为 503
警告     不良       活动探测失败：HTTP 探测失败，代码为 503
警告     不良       活动探测失败：HTTP 探测失败，代码为 503
普通的   杀戮       容器特使活动探测失败，将重新启动

普通的   拉动       容器镜像已经存在于机器上
普通的   已创建     创建容器特使
普通的   开始       启动容器特使
```

请记住，探测失败阈值设置为 3，因此当探测连续失败 3 次时，容器将停止并重新启动。列表中的事件表明了这一点。

kubectl get pods 命令显示容器已重新启动：

```
$ kubectl get po kubia-liveness 名称就绪状态
重新启动年龄
kubia-liveness 2/2 运行        1 5m
```

RESTARTS 列显示 pod 中已发生一次容器重启。

了解活性探针失败的容器如何重新启动

如果您想知道容器中的主进程是否被正常停止或被强制终止，您可以通过使用 kubectl get 或使用 kubectl describe 检索完整清单来检查 pod 的状态，如以下列表所示。

清单 6.10 使用 kubectl describe 检查重新启动的容器的最后状态

```
$ kubectl 描述 po kubia-
liveness 名称：kubia-
liveness
..容器
特
使..状态：运行 #A
：  开始时间：2020 年 5 月 31 日星期日 21:33:13 +0200
  最后状态：终止 #B
    原因：已完成 #B
    退出代码：   0#B
    开始时间：2020 年 5 月 31 日星期日 21:16:43
    +0200 #B
  ...
```

#A This is the state of the new container.
#B The previous container terminated with exit code 0.

The exit code zero shown in the listing implies that the application process gracefully exited on its own. If it had been killed, the exit code would have been 137.

NOTE Exit code `128+n` indicates that the process exited due to external signal `n`. Exit code `137` is `128+9`, where `9` represents the `KILL` signal. You'll see this exit code whenever the container is killed. Exit code `143` is `128+15`, where 15 is the `TERM` signal. You'll typically see this exit code when the container runs a shell that has terminated gracefully.

Let's examine Envoy's log to confirm that it caught the `TERM` signal and has terminated by itself. You must use the `kubectl logs` command with the `--container` or the shorter `-c` option to specify what container you're interested in.

Also, because the container has been replaced with a new one due to the restart, you must request the log of the previous container using the `--previous` or `-p` flag. The next listing shows the full command and the last four lines of its output.

Listing 6.11 The last few lines of Envoy's log when killed due to a failed liveness probe

```
$ kubectl logs kubia-liveness -c envoy -p
...
...[warning][main] [source/server/server.cc:493] caught SIGTERM
...[info][main] [source/server/server.cc:613] shutting down server instance
...[info][main] [source/server/server.cc:560] main dispatch loop exited
...[info][main] [source/server/server.cc:606] exiting
```

The log confirms that Kubernetes sent the `TERM` signal to the process, allowing it to shut down gracefully. Had it not terminated by itself, Kubernetes would have killed it forcibly.

After the container is restarted, its health check endpoint responds with HTTP status `200 OK` again, indicating that the container is healthy.

6.2.5 Using the exec and the tcpSocket liveness probe types

For applications that don't expose HTTP health-check endpoints, the `tcpSocket` or the `exec` liveness probes should be used.

ADDING A TCPSOCKET LIVENESS PROBE

For applications that accept non-HTTP TCP connections, a `tcpSocket` liveness probe can be configured. Kubernetes tries to open a socket to the TCP port and if the connection is established, the probe is considered a success, otherwise it's considered a failure.

An example of a `tcpSocket` liveness probe is shown in the following listing.

Listing 6.12 An example of a tcpSocket liveness probe

```
livenessProbe:
  tcpSocket:              #A
    port: 1234            #A
```


#A 这是新容器的状态。

列表中显示的退出代码零意味着应用程序进程自行正常退出。如果它被杀死，退出代码将为 137。

注意 退出代码 128+n 表示进程由于外部信号 n 而退出。退出代码 137 是 128+9，其中 9 代表 KILL 信号。每当容器被终止时，您都会看到此退出代码。退出代码 143

是 128+15，其中 15 是 TERM 信号。当容器运行 shell 时，您通常会看到此退出代码

让我们检查 Envoy 的日志以确认它捕获了 TERM 信号并已终止

本身。您必须使用带有 --container 或更短的 -c 选项的 kubectl logs 命令来指定您感兴趣的容器。

另外，由于容器因重新启动而被新容器替换，因此您必须使用 --previous 或 -p 标志请求前一个容器的日志。下一个清单显示了完整的命令及其输出的最后四行。

清单 6.11 由于活跃度探测失败而被杀死时 Envoy 日志的最后几行

```
$ kubectl 日志 kubia-liveness -c envoy
-p

... ...[警告][main]
[source/server/server.cc:493] 捕获
SIGTERM ...[info][main]
```

日志确认 Kubernetes 向进程发送了 TERM 信号，允许其正常关闭。如果它不是自己终止的话，Kubernetes 就会强行杀死它。

容器重新启动后，其健康检查端点响应 HTTP 状态 200 再次OK，说明容器是健康的。

6.2.5 使用 exec 和 tcpSocket 活性探测类型

对于不公开 HTTP 运行状况检查端点的应用程序，应使用 tcpSocket 或 exec liveness 探针。

添加 TCPSocket 活跃度探针

对于接受非 HTTP TCP 连接的应用程序，可以配置 tcpSocket liveness 探针。 Kubernetes 尝试打开一个到 TCP 端口的套接字，如果连接建立，则探测被认为是成功的，否则被认为是失败的。下面的清单显示了 tcpSocket 活动探针的示例。

清单 6.12 tcpSocket 活跃度探测的示例

```
livenessProbe : tcpSocket :
#A
    端口：1234#A
```

```
    periodSeconds: 2        #B
    failureThreshold: 1     #C
```

#A This tcpSocket probe uses TCP port 1234
#B The probe runs every 2s
#C A single probe failure is enough to restart the container

The probe in the listing is configured to check if the container's network port `1234` is open. An attempt to establish a connection is made every two seconds and a single failed attempt is enough to consider the container as unhealthy.

ADDING AN EXEC LIVENESS PROBE

Applications that do not accept TCP connections may provide a command to check their status. For these applications, an `exec` liveness probe is used. As shown in the next figure, the command is executed inside the container and must therefore be available on the container's file system.

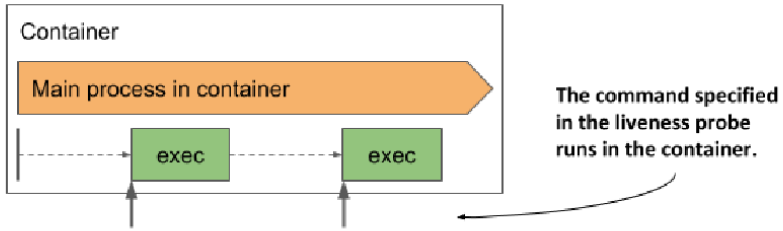


Figure 6.7 The exec liveness probe runs the command inside the container

The following listing shows an example of a probe that runs `/usr/bin/healthcheck` every two seconds to determine if the application running in the container is still alive.

Listing 6.13 An example of an exec liveness probe

```
  livenessProbe:
    exec:
      command:                  #A
      - /usr/bin/healthcheck    #A
    periodSeconds: 2            #B
    timeoutSeconds: 1           #C
    failureThreshold: 1         #D
```

#A The command to run and its arguments
#B The probe runs every second
#C The command must return within one second
#D A single probe failure is enough to restart the container


```
周期秒：2          #B
失败阈值：1        #C
```

#A 此 tcpSocket 探测使用 TCP 端口 1234 #B
探测每 2 秒运行一次

列表中的探测器配置为检查容器的网络端口 1234 是否打开。每两秒就会尝试建立一次连接，一次失败的尝试就足以将容器视为不健康。

添加 EXEC 活动探针

不接受 TCP 连接的应用程序可以提供命令来检查其状态。对于这些应用程序，使用 exec liveness 探针。如下图所示，该命令在容器内执行，因此必须在容器的文件系统上可用。

图6.7 exec liveness探针在容器内运行命令

以下清单显示了一个探测器示例，该探测器每两秒运行一次 /usr/bin/healthcheck 以确定容器中运行的应用程序是否仍处于活动状态。

清单 6.13 exec 活跃度探测的示例

```
livenessProbe
: 执行：
    命令：#A

  periodSeconds: 2 #B

  超时秒数：1 #C
```

#A 要运行的命令及其参数 #B 探针
每秒运行一次 #C 命令必须在一秒内
返回 #D 单个探针失败足以重新启动
容器

If the command returns exit code zero, the container is considered healthy. If it returns a non-zero exit code or fails to complete within one second as specified in the `timeoutSeconds` field, the container is terminated immediately, as configured in the `failureThreshold` field, which indicates that a single probe failure is sufficient to consider the container as unhealthy.

6.2.6 Using a startup probe when an application is slow to start

The default liveness probe settings give the application between 20 and 30 seconds to start responding to liveness probe requests. If the application takes longer to start, it is restarted and must start again. If the second start also takes as long, it is restarted again. If this continues, the container never reaches the state where the liveness probe succeeds and gets stuck in an endless restart loop.

To prevent this, you can increase the `initialDelaySeconds`, `periodSeconds` or `failureThreshold` settings to account for the long start time, but this will have a negative effect on the normal operation of the application. The higher the result of `periodSeconds * failureThreshold`, the longer it takes to restart the application if it becomes unhealthy. For applications that take minutes to start, increasing these parameters enough to prevent the application from being restarted prematurely may not be a viable option.

INTRODUCING STARTUP PROBES

To deal with the discrepancy between the start and the steady-state operation of an application, Kubernetes also provides *startup probes*.

If a startup probe is defined for a container, only the startup probe is executed when the container is started. The startup probe can be configured to take into account the slow start of the application. When the startup probe succeeds, Kubernetes switches to using the liveness probe, which is configured to quickly detect when the application becomes unhealthy.

ADDING A STARTUP PROBE TO A POD'S MANIFEST

Imagine that the kubia Node.js application needs more than a minute to warm up, but you want it to be restarted within 10 seconds after it has become unhealthy during normal operation. The following listing shows how you'd configure the startup and the liveness probes.

Listing 6.14 Using a combination of startup and liveness probes

```
containers:
- name: kubia
  image: luksa/kubia:1.0
  ports:
  - name: http
    containerPort: 8080
  startupProbe:
    httpGet:
      path: /                   #A
      port: http                #A
    periodSeconds: 10           #B
    failureThreshold:  12       #B
```


如果该命令返回退出代码零，则容器被认为是健康的。如果它返回非零退出代码或未能在 timeoutSeconds 字段中指定的一秒内完成，则容器将立即终止，如 failureThreshold 字段中的配置，这表明单个探测失败足以将容器视为不良。

6.2.6 当应用程序启动缓慢时使用启动探针

默认的活性探测设置为应用程序提供了 20 到 30 秒的时间来开始响应活性探测请求。如果应用程序启动时间较长，则会重新启动并且必须重新启动。如果第二次启动也需要同样长的时间，则会再次重新启动。如果这种情况持续下去，容器将永远不会达到活性探测成功的状态，并陷入无限的重启循环。为了防止这种情况，您可以增加initialDelaySeconds、periodSeconds或

failureThreshold 设置要考虑到较长的启动时间，但这会产生负数

影响应用程序的正常运行。 periodSeconds * failureThreshold 的结果越高，如果应用程序变得不健康，重新启动应用程序所需的时间就越长。为了

对于需要几分钟才能启动的应用程序，增加这些参数足以防止应用程序过早重新启动可能不是一个可行的选择。

启动探针简介

为了处理应用程序的启动和稳态运行之间的差异，Kubernetes 还提供了启动探针。

如果为容器定义了启动探针，则容器启动时仅执行启动探针。可以配置启动探针以考虑应用程序的缓慢启动。当启动探针成功时，Kubernetes 会切换到使用活性探针，该探针配置为快速检测应用程序何时变得不健康。

将启动探针添加到 Pod 的清单中

想象一下，kubia Node.js 应用程序需要一分多钟的时间来预热，但您希望它在正常运行期间变得不健康后在 10 秒内重新启动。以下列表显示了如何配置启动和活动探针。

清单 6.14 结合使用启动和活跃探针

```
容器： -
名称：
图片：
kubia
  luksa/kubia:1.0 端
  口 名称：http
  容器端口：8080
  启动探针：
  httpGet：
      路径：/#A

    周期秒：10 #B
```

```
    livenessProbe:
      httpGet:
        path: /                 #A
        port: http              #A
      periodSeconds: 5          #C
      failureThreshold: 2       #C
```

#A The startup and the liveness probes typically use the same endpoint
#B The application gets 120 seconds to start
#C After startup, the application's health is checked every 5 seconds, and is restarted when it fails the liveness probe twice

When the container defined in the listing starts, the application has 120 seconds to start responding to requests. Kubernetes performs the startup probe every 10 seconds and makes a maximum of 12 attempts.

As shown in the following figure, unlike liveness probes, it's perfectly normal for a startup probe to fail. A failure only indicates that the application hasn't yet been completely started. A successful startup probe indicates that the application has started successfully, and Kubernetes should switch to the liveness probe. The liveness probe is then typically executed using a shorter period of time, which allows for faster detection of non-responsive applications.

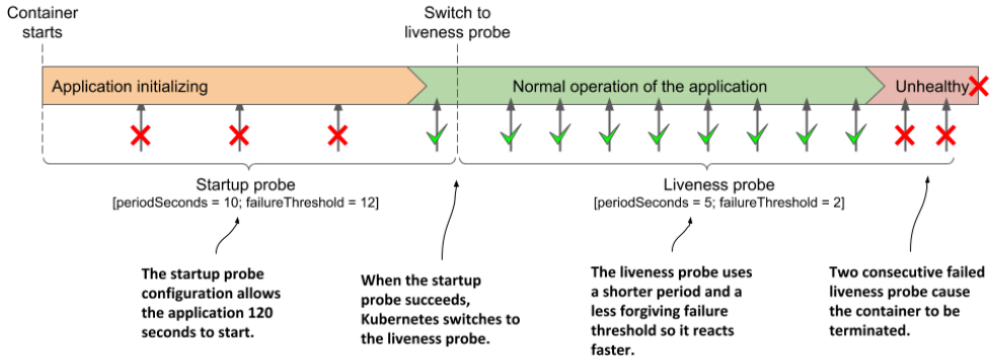


Figure 6.8 Fast detection of application health problems using a combination of startup and liveness probe

NOTE If the startup probe fails often enough to reach the `failureThreshold`, the container is terminated as it the liveness probe had failed.

Usually, the startup and liveness probes are configured to use the same HTTP endpoint, but different endpoints can be used. You can also configure the startup probe as an `exec` or `tcpSocket` probe instead of an `httpGet` probe.


```
livenessProbe
: httpGet：
    小路： /                #A
    端口：http              #A
  周期秒：5                 #C
  失败阈值：2               #C
```

#A 启动和活性探针通常使用相同的端点 #B 应用程序有 120 秒的时间启动

#C 启动后，每5秒检查一次应用程序的健康状况，当两次liveness探针失败时重新启动

当清单中定义的容器启动时，应用程序有 120 秒的时间启动

响应请求。 Kubernetes 每 10 秒执行一次启动探测，并进行

最多 12 次尝试。

如下图所示，与活性探针不同，启动探针失败是完全正常的。失败仅表明应用程序尚未完全启动。成功的启动探针表明应用程序已成功启动，Kubernetes 应该切换到活性探针。然后，通常会使用更短的时间执行活性探测，这样可以更快地检测无响应的应用程序。

图 6.8 使用启动和活跃探针组合快速检测应用程序健康问题

注意：如果启动探测失败的频
率足以达到
failureThreshold，则容器将

通常，启动探针和活动探针配置为使用相同的 HTTP 端点，但也可以使用不同的端点。您还可以将启动探针配置为 exec 或 tcpSocket 探针，而不是 httpGet 探针。

6.2.7 Creating effective liveness probe handlers

You should define a liveness probe for all your pods. Without one, Kubernetes has no way of knowing whether your app is still alive or not, apart from checking whether the application process has terminated.

Causing unnecessary restarts with badly implemented liveness probe handlers

When you implement a handler for the liveness probe, either as an HTTP endpoint in your application or as an additional executable command, be very careful to implement it correctly. If a poorly implemented probe returns a negative response even though the application is healthy, the application will be restarted unnecessarily. Many Kubernetes users learn this the hard way. If you can make sure that the application process terminates by itself when it becomes unhealthy, it may be safer not to define a liveness probe.

What a liveness probe should check

The liveness probe for the `kubia` container isn't configured to call an actual health-check endpoint, but only checks that the Node.js server responds to simple HTTP requests for the root URI. This may seem overly simple, but even such a liveness probe works wonders, because it causes a restart of the container if the server no longer responds to HTTP requests, which is its main task. If no liveness probe was defined, the pod would remain in an unhealthy state where it doesn't respond to any requests and would have to be restarted manually. A simple liveness probe like this is better than nothing.

To provide a better liveness check, web applications typically expose a specific health-check endpoint, such as `/healthz`. When this endpoint is called, the application performs an internal status check of all the major components running within the application to ensure that none of them have died or are no longer doing what they should.

> **TIP** Make sure that the `/healthz` HTTP endpoint doesn't require authentication; otherwise the probe will always fail, causing your container to be restarted indefinitely.

Make sure that the application checks only the operation of its internal components and nothing that is influenced by an external factor. For example, the health-check endpoint of a frontend service should never respond with failure when it can't connect to a backend service. If the backend service fails, restarting the frontend will not solve the problem. Such a liveness probe will fail again after the restart, so the container will be restarted repeatedly until the backend is repaired. If many services are interdependent in this way, the failure of a single service can result in cascading failures across the entire system.

Keeping probes light

The handler invoked by a liveness probe shouldn't use too much computing resources and shouldn't take too long to complete. By default, probes are executed relatively often and only given one second to complete.

Using a handler that consumes a lot of CPU or memory can seriously affect the main process of your container. Later in the book you'll learn how to limit the CPU time and total memory available to a container. The CPU and memory consumed by the probe handler

6.2.7 创建有效的活性探针处理程序

您应该为所有 pod 定义一个活性探针。如果没有它，Kubernetes 除了检查应用程序进程是否已终止之外，无法知道您的应用程序是否仍然存在。

活动探针处理程序实施不当导致不必要的重启

当您为活性探针实现处理程序时，无论是作为应用程序中的 HTTP 端点还是作为附加的可执行命令，都要非常小心地正确实现它。如果实施不当的探测器即使应用程序运行正常也返回否定响应，则应用程序将不必要地重新启动。许多 Kubernetes 用户都经历了惨痛的教训才明白这一点。如果您可以确保应用程序进程在变得不健康时自行终止，那么不定义活性探针可能会更安全。

活性探针应该检查什么

kubia 容器的活性探针未配置为调用实际的运行状况检查端点，而仅检查 Node.js 服务器是否响应根 URI 的简单 HTTP 请求。这可能看起来过于简单，但即使这样的活性探针也能产生奇迹，因为如果服务器不再响应 HTTP 请求（这是其主要任务），它会导致容器重新启动。如果未定义活性探针，Pod 将保持不健康状态，不响应任何请求，并且必须手动重新启动。像这样简单的活性探针总比没有好。

为了提供更好的活性检查，Web 应用程序通常会公开特定的运行状况检查端点，例如 /healthz。调用此端点时，应用程序会对应用程序中运行的所有主要组件执行内部状态检查，以确保它们都没有死亡或不再执行其应做的事情。

> 提示确保 /healthz HTTP 端点不需要身份验证；否则探测器将始终失败，导致容器无限期地重新启动。

确保应用程序仅检查其内部组件的操作，而不检查受外部因素影响的情况。例如，当前端服务的运行状况检查端点无法连接到后端服务时，它永远不应该响应失败。如果后端服务出现故障，重启前端也无法解决问题。这样的liveness探针重启后会再次失败，因此容器会反复重启，直到后端修复。如果许多服务以这种方式相互依赖，那么单个服务的故障可能会导致整个系统的级联故障。

保持探头轻便

活性探针调用的处理程序不应使用过多的计算资源，也不应花费太长时间才能完成。默认情况下，探测相对频繁地执行，并且只给一秒钟的时间来完成。

使用消耗大量 CPU 或内存的处理程序可能会严重影响容器的主进程。在本书后面，您将学习如何限制容器可用的 CPU 时间和总内存

invocation count towards the resource quota of the container, so using a resource-intensive handler will reduce the CPU time available to the main process of the application.

> **TIP** When running a Java application in your container, you may want to use an HTTP GET probe instead of an exec liveness probe that starts an entire JVM. The same applies to commands that require considerable computing resources.

AVOIDING RETRY LOOPS IN YOUR PROBE HANDLERS

You've learned that the failure threshold for the probe is configurable. Instead of implementing a retry loop in your probe handlers, keep it simple and instead set the `failureThreshold` field to a higher value so that the probe must fail several times before the application is considered unhealthy. Implementing your own retry mechanism in the handler is a waste of effort and represents another potential point of failure.

6.3 Executing actions at container start-up and shutdown

In the previous chapter you learned that you can use init containers to run containers at the start of the pod lifecycle. You may also want to run additional processes every time a container starts and just before it stops. You can do this by adding *lifecycle hooks* to the container. Two types of hooks are currently supported:

- *Post-start* hooks, which are executed when the container is started, and
- *Pre-stop* hooks, which are executed shortly before the container stops.

These lifecycle hooks are specified per container, as opposed to init containers, which are specified at the pod level. The next figure should help you visualize how lifecycle hooks fit into the lifecycle of a container.

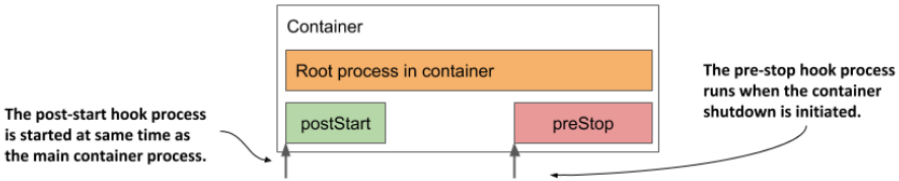


Figure 6.9 How the post-start and pre-stop hook fit into the container's lifecycle

Like liveness probes, lifecycle hooks can be used to either

- execute a command inside the container, or
- send an HTTP GET request to the application in the container.

> **NOTE** The same as with liveness probes, lifecycle hooks can only be applied to regular containers and not to init containers. Unlike probes, lifecycle hooks do not support `tcpSocket` handlers.

162

调用计数计入容器的资源配额，因此使用资源密集型处理程序将减少应用程序主进程可用的 CPU 时间。

> 提示在容器中运行 Java 应用程序时，您可能需要使用 HTTP GET 探针而不是启动整个 JVM 的 exec liveness 探针。这同样适用于需要相当多的命令
>
> 计算资源

避免探针处理程序中的重试循环

你已经 学到了 那 这 失败 临界点 为了这 探测 是 可配置。 反而 的 在探针处理程序中实现重试循环，保持简单，并将 failureThreshold 字段设置为更高的值，以便探针必须先失败几次

该应用程序被认为是不健康的。在处理程序中实现您自己的重试机制是浪费精力，并且代表另一个潜在的失败点。

6.3 容器启动和关闭时执行操作

在上一章中，您了解到可以使用 init 容器在 pod 生命周期开始时运行容器。您可能还希望在容器每次启动时和停止之前运行其他进程。您可以通过向容器添加生命周期挂钩来做到这一点。目前支持两种类型的钩子：

- 启动后挂钩，在容器启动时执行，以及
- 预停止挂钩，在容器停止前不久执行

这些生命周期挂钩是按容器指定的，而不是在 pod 级别指定的 init 容器。下图应该可以帮助您直观地了解生命周期挂钩如何融入容器的生命周期。

图 6.9 post-start 和 pre-stop 钩子如何适应容器的生命周期

与活性探针一样，生命周期钩子可用于

- 在容器内执行命令，或者
- 向容器中的应用程序发送HTTP GET 请求

> 注意与活性探针相同，生命周期钩子只能应用于常规容器，而不能应用于初始化容器。与探针不同，生命周期挂钩不支持 tcpSocket 处理程序。

Let's look at the two types of hooks individually to see what you can use them for.

6.3.1 Using post-start hooks to perform actions when the container starts

The post-start lifecycle hook is invoked immediately after the container is created. You can use the `exec` type of the hook to execute an additional process as the main process starts, or you can use the `httpGet` hook to send an HTTP request to the application running in the container to perform some type of initialization or warm-up procedure.

If you're the author of the application, you could perform the same operation within the application code itself, but if you need to add it to an existing application that you didn't create yourself, you may not be able to do so. A post-start hook provides a simple alternative that doesn't require you to change the application or its container image.

Let's look at an example of how a post-start hook can be used in a new service you'll create.

USING A POST-START CONTAINER LIFECYCLE HOOK TO RUN A COMMAND IN THE CONTAINER

During my first steps with Unix in the 1990s, one of the things I found amusing was the random, sometimes funny message that the `fortune` command displayed every time I logged into our high school's server, which was running the Solaris OS at the time. Nowadays, you'll rarely see the `fortune` command installed on Unix/Linux systems anymore, but you can still install it and run it whenever you're bored. Here's an example of what it may display:

```
$ fortune
Dinner is ready when the smoke alarm goes off.
```

In the following exercise, you'll combine the `fortune` program with the Nginx web server to create a web-based fortune service.

For the first version of the service, the `fortune` command will write the message to a file, which will then be served by Nginx. Although this means that the same message is returned in every request, this is a perfectly good start. You'll later improve the service iteratively.

The Nginx web server is available as a container image, so let's use it. Because the fortune command is not available in the image, you'd normally build a new image based on the Nginx image and install the fortune package during the container build process.

Let's keep things ultra simple for now and both install and run the `fortune` command when the container starts. Although this is not something you should normally do, a post-start hook can be used for this. The following listing shows the pod manifest that defines a post-start hook that does this.

Listing 6.15 Pod with a post-start lifecycle hook

```
apiVersion: v1
kind: Pod
metadata:
  name: fortune-poststart                    #A
spec:
  containers:
  - name: nginx                              #B
```


让我们分别看看这两种类型的钩子，看看它们可以用来做什么。

6.3.1 使用启动后钩子在容器启动时执行操作

创建容器后立即调用启动后生命周期挂钩。您可以使用钩子的 exec 类型在主进程启动时执行附加进程，或者

您可以使用 httpGet 挂钩向容器中运行的应用程序发送 HTTP 请求，以执行某种类型的初始化或预热过程。

如果您是应用程序的作者，则可以在应用程序代码本身中执行相同的操作，但如果您需要将其添加到不是您自己创建的现有应用程序中，则可能无法这样做。启动后挂钩提供了一种简单的替代方案，不需要您更改应用程序或其容器映像。

让我们看一个示例，了解如何在您将要使用的新服务中使用启动后挂钩
创造。

使用启动后容器生命周期挂钩在容器中运行命令

在 20 世纪 90 年代我第一次接触 Unix 时，我发现有趣的一件事是每次我登录我们高中的服务器时，fortune 命令都会显示随机的、有时甚至是有趣的消息，该服务器当时运行的是 Solaris 操作系统。如今，你很少会看到在 Unix/Linux 系统上安装 Fortune 命令，但你仍然可以安装它并在无聊时运行它。以下是它可能显示的内容的示例：

```
$ Fortune
当烟雾警
报器响起
时，晚餐
就准备好
```

在下面的练习中，您将把 Fortune 程序与 Nginx Web 服务器结合起来，创建一个基于 Web 的 Fortune 服务。

对于该服务的第一个版本，fortune 命令会将消息写入文件，然后由 Nginx 提供服务。尽管这意味着每个请求都会返回相同的消息，但这是一个完美的开始。您稍后将迭代地改进服务。

Nginx Web 服务器可作为容器映像使用，所以让我们使用它。由于镜像中没有 Fortune 命令，因此您通常需要基于 Nginx 镜像构建一个新镜像，并在容器构建过程中安装 Fortune 包。

现在让事情变得非常简单，在容器启动时安装并运行 Fortune 命令。尽管这不是您通常应该做的事情，但可以使用启动后挂钩来实现此目的。以下清单显示了定义执行此操作的启动后挂钩的 pod 清单。

清单 6.15 具有启动后生命周期挂钩的 Pod

```
api 版本：v1
种类：Pod 元
数据
  名称： Fortune-Poststart #A
规格：
  容器：-名称：nginx #B
```

```
    image: nginx:alpine                              #B
    lifecycle:                                       #C
      postStart:                                     #C
        exec:                                        #C
          command:                                   #C
          - sh                                                          #D
          - -c                                                          #D
          - "apk add fortune && fortune > /usr/share/nginx/html/quote"  #D
    ports:                                           #E
    - name: http                                     #E
      containerPort: 80                              #E
```

#A Let's name this pod fortune-poststart
#B The nginx:alpine container image is used in this single-container pod
#C A post-start lifecycle hook is used to run a command when the container starts
#D This is the shell command that is executed
#E The Nginx server runs on port 80

The pod is named `fortune-poststart` and contains a single container based on the `nginx:alpine` image. A `postStart` lifecycle hook is defined for the container. It executes the following command when the Nginx server is started:

```
sh -c "apk add fortune && fortune > /usr/share/nginx/html/quote"
```

Let me break the command down for you. It executes the `sh` shell that runs the following two commands:

1. The `apk add fortune` command installs the `fortune` package.
2. The `fortune` command is executed, and its output is redirected to the `/usr/share/nginx/html/quote` file.

The command runs parallel to the main process. The `postStart` name is somewhat misleading, because the hook isn't called after the main process is fully started, but as soon as the container is created, at roughly the same time as when the main process starts. When the `postStart` hook is finished, the quote produced by the `fortune` command is stored in the file and is ready to be served by Nginx.

Use the `kubectl apply` command to create the pod from the `fortune-poststart.yaml` file, and you should then be able to use `curl` or your browser to get the quote at URI `/quote` on port `80` of the `fortune-poststart` pod. You've already learned how to do this, but you may want to refer to the sidebar because there is a caveat.

Accessing the fortune-poststart pod

To retrieve the quote from the `fortune-poststart` pod, you must first run the `kubectl port-forward` command, which may fail as shown here:

```
$ kubectl port-forward fortune-poststart 80
Unable to listen on port 80: Listeners failed to create with the following errors: [unable
to create listener: Error listen tcp4 127.0.0.1:80: bind: permission denied unable to
create listener: Error listen tcp6 [::1]:80: bind: permission denied]
```



```
图片：nginx:alpine                          #B
生命周期：                                  #C
  开始后：                                  #C
    执行：                                  #C
      命令：                                #C
      - 嘘                                                  #D
      - -C                                                  #D
      - “apk 添加财富 && 财富 > /usr/share/nginx/html/quote”#D
端口：                                      #E
- 名称：http                                #E
  集装箱端口：80                            #E
```

#A 让我们将这个 Pod 命名为 Fortune-Poststart

#B 在这个单容器 Pod 中使用 nginx:alpine 容器镜像 #C 启动后生命周期钩子用于在容器启动时运行命令 #D 这是执行的 shell 命令 #E Nginx 服务器运行于端口80

该 Pod 名为 Fortune-Poststart，包含一个基于 nginx:alpine 镜像的容器。为容器定义了 postStart 生命周期钩子。它执行

Nginx服务器启动时执行以下命令：

```
sh -c "apk 添加财富 && 财富 > /usr/share/nginx/html/quote"
```

让我为您分解该命令。它执行 sh shell，该 shell 运行以下两个命令：

1. apk add Fortune命令安装fortune包。
2. 执行fortune命令，其输出被重定向到/usr/share/nginx/html/quote文件。

该命令与主进程并行运行。 postStart 名称有些误导，因为钩子不会在主进程完全启动后调用，而是在容器创建后立即调用，与主进程启动的时间大致相同。当 postStart 钩子完成时，fortune 命令生成的报价将存储在文件中，并准备好由 Nginx 提供服务。

使用 kubectl apply 命令从 Fortune-poststart.yaml 创建 pod 文件，然后您应该能够使用curl或您的浏览器在fortune-poststart pod的端口80上的URI /quote处获取报价。您已经了解了如何执行此操作，但您可能需要参考侧边栏，因为有一个警告。

访问 Fortune-Poststart pod

要从 Fortune-poststart pod 中检索报价，您必须首先运行 kubectl port-forward 命令，该命令可能会失败，如下所示：

```
$ kubectl 端口转发 Fortune-poststart 80
```

无法侦听端口 80：侦听器无法创建，并出现以下错误：[无法创建侦听器：错误侦听 tcp4 127.0.0.1:80：绑定：权限被拒绝无法

```
error: unable to listen on any of the requested ports: [{80 80}]
```

The command fails if your operating system doesn't allow you to run processes that bind to port numbers 0-1023. To fix this, you must use a higher local port number as follows:

```
$ kubectl port-forward fortune-poststart 1080:80
```

The last argument tells `kubectl` to use port `1080` locally and forward it to port `80` of the pod.
You can now access the fortune service at http://localhost:1080/quote.

UNDERSTANDING HOW A POST-START HOOK AFFECTS THE CONTAINER

Although the post-start hook runs asynchronously with the main container process, it affects the container in two ways.

First, the container remains in the `Waiting` state with the reason `ContainerCreating` until the hook invocation is completed. The phase of the pod is `Pending`. If you run the `kubectl logs` command at this point, it refuses to show the logs, even though the container is running. The `kubectl port-forward` command also refuses to forward ports to the pod.

If you want to see this for yourself, deploy the `fortune-poststart-slow.yaml` pod manifest file that you can find in the code archive of the book. It defines a post-start hook that takes 60 seconds to complete. Immediately after the pod is created, inspect its state, and display the logs with the following command:

```
$ kubectl logs fortune-poststart-slow
Error from server (BadRequest): container "nginx" in pod "fortune-poststart-slow" is
waiting to start: ContainerCreating
```

The error message returned implies that the container hasn't started yet, which isn't the case. To prove this, use the command in the following listing to list processes in the container:

Listing 6.16 Processes running in a container while the post-start hooks runs

```
$ kubectl exec fortune-poststart-slow -- ps x
PID   USER     TIME  COMMAND
  1 root      0:00 nginx: master process nginx -g daemon off;          #A
  7 root      0:00 sh -c sleep 60 && apk add fortune && fortune > ...  #B
 13 nginx     0:00 nginx: worker process                               #A
...                                                                    #A
 20 nginx     0:00 nginx: worker process                               #A
 21 root      0:00 sleep 60                                            #B
 22 root      0:00 ps x
```

#A Nginx is running
#B The post-start hook processes

The other way a post-start hook can affect the container is if the command used in the hook can't be executed or returns a non-zero exit code. If this happens, the entire container is

错误：无法侦听任何请求的端口：[{80 80}]

如果您的操作系统不允许您运行绑定到端口号 0-1023 的进程，则该命令将失败。要解决此问题，您必须使用更高的本地端口号，如下所示：

$ kubectl 端口转发 Fortune-poststart 1080:80

最后一个参数告诉 kubectl 在本地使用端口 1080 并将其转发到 pod 的端口 80。您现在可以通过 http://localhost:1080/quote 访问 Fortune 服务。

了解启动后钩子如何影响容器

尽管启动后钩子与主容器进程异步运行，但它以两种方式影响容器。

首先，容器仍处于 Waiting 状态，原因是 ContainerCreating 直到钩子调用完成。 Pod 的阶段为 Pending。如果此时运行 kubectl logs 命令，它会拒绝显示日志，即使容器

在跑。 kubectl port-forward 命令也拒绝将端口转发到 pod。

如果您想亲自查看这一点，请部署 Fortune-poststart-slow.yaml pod 清单文件，您可以在本书的代码存档中找到该文件。它定义了一个需要 60 秒才能完成的启动后挂钩。创建 Pod 后，立即检查其状态，并使用以下命令显示日志：

```
$ kubectl 日志 Fortune-poststart-slow
来自服务器的错误（BadRequest）：pod“fortune-poststart-slow”中的容
器“nginx”正在等待启动：ContainerCreating
```

返回的错误消息意味着容器尚未启动，但事实并非如此。为了证明这一点，请使用以下清单中的命令列出容器中的进程：

清单 6.16 当启动后挂钩运行时，进程在容器中运行

```
$ kubectl exec Fortune-
poststart-slow -- ps x
  1根         0:00 nginx：主进程 nginx -g 守护进程关闭; #A
  7根         0:00 sh -c sleep 60 && apk 添加财富 && 财富 > ... #B
 13 nginx     0:00 nginx：工作进程#A
... #A
 20个nginx    0:00 nginx：工作进程#A
 21根         0:00 睡眠 60 #B
 22根         0:00 ps x
```

#A Nginx 正在
运行 #B 启动后
钩子进程

启动后挂钩影响容器的另一种方式是挂钩中使用的命令无法执行或返回非零退出代码。如果发生这种情况

restarted. To see an example of a post-start hook that fails, deploy the pod manifest `fortune-poststart-fail.yaml`.

If you watch the pod's status using kubectl get pods -w, you'll see the following status:

```
fortune-poststart-fail   0/1     PostStartHookError: command 'sh -c echo 'Emulating
a post-start hook failure'; exit 1' exited with 1:
```

It shows the command that was executed and the code with which it terminated. When you review the pod events shown in the following listing, you'll see a `FailedPostStartHook` warning event that indicates the exit code and what the command printed to the standard or error output.

Listing 6.17 Event showing the exit code and output of a failed postStart hook

```
Warning  FailedPostStartHook  Exec lifecycle hook ([sh -c ...]) for
                              Container "nginx" in Pod "fortune-poststart-
                              fail_default(...)" failed - error: command
                                '...' exited with 1: , message: "Emulating a
                                post-start hook failure\n"
```

The same information is also contained in the `containerStatuses` field in the pod's `status` field, but only for a short time, as the container status changes to `CrashLoopBackOff` shortly afterwards.

> **TIP** Because the state of a pod can change quickly, inspecting just its status may not tell you everything you need to know. Rather than inspecting the state at a particular moment in time, reviewing the pod's events is usually a better way to get the full picture.

CAPTURING THE OUTPUT PRODUCED BY THE PROCESS INVOKED VIA A POST-START HOOK

As you've just learned, the output of the command defined in the post-start hook can be inspected if it fails. In cases where the command completes successfully, the output of the command is not logged anywhere. To see the output, the command must log to a file instead of the standard or error output. You can then view the contents of the file with a command like the following:

```
$ kubectl exec my-pod cat logfile.txt
```

USING AN HTTP GET POST-START HOOK

In the previous example, you configured the post-start hook to invoke a command inside the container. Alternatively, you can have Kubernetes send an HTTP GET request when it starts the container by using an `httpGet` post-start hook.

> **NOTE** You can't specify both an `exec` and an `httpGet` post-start hook for a container. You can only specify one.

For example, when starting a web application inside a container, you may want to use a post-start hook to send an initial request to the application so that it can initialize its caches

重新启动。要查看启动后挂钩失败的示例，请部署 Pod 清单 Fortune-poststart-fail.yaml。

如果您使用 kubectl get pods -w 查看 pod 的状态，您将看到以下状态：

```
启动后的财富-失败      0/1     PostStartHookError ：命令'sh -c echo'模拟
启动后挂钩失败'； exit 1' 以 1 退出：
```

它显示了已执行的命令及其终止的代码。当您查看以下列表中显示的 pod 事件时，您将看到 FailedPostStartHook 警告事件，该事件指示退出代码以及命令打印到标准或错误输出的内容。

清单 6.17 显示失败的 postStart 挂钩的退出代码和输出的事件

```
警告 FailedPostStartHook  Exec 生命周期挂钩 ([sh -c ...])
                          Pod "fortune-
                          poststartfail_default (...)" 中的容
                          器 "nginx" 退出并显示 "1"：消息："模拟启动后挂
                          钩失败\n"
```

Pod 状态的 containerStatuses 字段中也包含相同的信息

字段，但只持续很短的时间，因为容器状态很快就会更改为 CrashLoopBackOff。

> 提示 由于 Pod 的状态可能会快速更改，因此仅检查其状态可能无法告诉您需要了解的所有信息。检查 pod 的事件不是检查特定时刻的状态，而是
> 通常是了解全貌的更好方法。

通过启动后挂钩捕获调用的进程产生的输出

正如您刚刚了解到的，如果启动后挂钩中定义的命令失败，则可以检查该命令的输出。如果命令成功完成，则命令的输出不会记录在任何地方。要查看输出，该命令必须记录到文件而不是标准或错误输出。然后，您可以使用如下命令查看文件的内容：

```
$ kubectl exec my-pod cat logfile.txt
```

使用 HTTP GET 启动后挂钩

在前面的示例中，您配置了启动后挂钩以调用容器内的命令。或者，您可以让 Kubernetes 在启动容器时使用 httpGet 启动后挂钩发送 HTTP GET 请求。

> 注意 您不能为容器同时指定 exec 和 httpGet

例如，当在容器内启动 Web 应用程序时

or warm up its other internal components, which allows the first request sent by an actual client to be processed more quickly.

The following listing shows an example of a post-start hook definition that does this:

Listing 6.18 Using an httpGet post-start hook to warm up a web server

```
  lifecycle:                #A
    postStart:              #A
      httpGet:              #A
        port: 80            #B
        path: /warmup       #C
```

#A This is a post-start lifecycle hook that performs an HTTP GET request
#B The request is sent to port 80 of the pod
#C The URI requested in the HTTP GET request

The example in the listing shows an `httpGet` post-start hook that calls the `/warmup` URI on port `80` of the pod. You can find this example in the `poststart-httpget.yaml` file in the book's code archive.

In addition to the `port` and `path` shown in the listing, you can also specify the `scheme` (`HTTP` or `HTTPS`) and `host` fields, as well as the `httpHeaders` to be sent in the request. The `host` field defaults to the pod IP. Be careful not to set it to `localhost`, because `localhost` refers to the node hosting the pod, not the pod itself.

As with command-based post-start hooks, the HTTP GET post-start hook is executed immediately when the container's main process starts. This can be problematic if the process doesn't start up immediately. As you've already learned, the container is restarted if the post-start hook fails. To see this for yourself, try creating the pod defined in `poststart-httpget-slow.yaml`.

> **WARNING** Using an HTTP GET post-start hook with applications that don't immediately start accepting connections can cause the container to enter an endless restart loop.

Interestingly, Kubernetes doesn't treat the hook as failed if the HTTP server responds with an HTTP error code like `404 Not Found`. Make sure you specify the correct URI in your HTTP GET hook, otherwise you might not even notice that the post-start hook does nothing.

6.3.2 Running a process just before the container terminates

Besides executing a command or sending an HTTP request at container startup, Kubernetes also allows the definition of a *pre-stop* hook in your containers.

A pre-stop hook is executed immediately before a container is terminated. To terminate a process, the `TERM` signal is usually sent to it. This tells the application to finish what it's doing and shut down. The same happens with containers. Whenever a container needs to be stopped or restarted, the `TERM` signal is sent to the main process in the container. Before this happens, however, Kubernetes first executes the pre-stop hook, if one is configured for the container. The `TERM` signal is not sent until the pre-stop hook completes unless the process has already terminated due to the invocation of the pre-stop hook handler itself.

或者预热其其他内部组件，这使得可以更快地处理实际客户端发送的第一个请求。

以下清单显示了执行此操作的启动后挂钩定义的示例：

清单 6.18 使用 httpGet 启动后挂钩来预热 Web 服务器

```
生命周期：                #A
  开始后：                #A
    http获取：            #A
      端口：80            #B
      路径：/热身         #C
```

#A 这是一个启动后生命周期钩子，执行 HTTP GET 请求 #B 请求发送到 Pod 的 80 端口 #C HTTP GET 请求中请求的 URI

清单中的示例显示了一个 httpGet 启动后挂钩，该挂钩在 pod 的端口 80 上调用 /warmup URI。您可以在本书代码存档的 poststart-httpget.yaml 文件中找到此示例。

除了列表中显示的端口和路径之外，您还可以指定方案（HTTP 或 HTTPS）和主机字段，以及要在请求中发送的 httpHeaders。主机字段默认为 Pod IP。注意不要设置为localhost，因为localhost

指托管 Pod 的节点，而不是 Pod 本身。

与基于命令的启动后挂钩一样，HTTP GET 启动后挂钩在容器的主进程启动时立即执行。如果进程没有立即启动，这可能会出现问题。正如您已经了解到的，如果启动后挂钩失败，容器将重新启动。要亲自查看这一点，请尝试创建 poststarthttpget-slow.yaml 中定义的 pod。

> 警告 对不立即开始接受连接的应用程序使用 HTTP GET 启动后挂钩可能会导致容器进入无休止的重新启动循环。

有趣的是，如果 HTTP 服务器响应像 404 Not Found 这样的 HTTP 错误代码，Kubernetes 不会将钩子视为失败。确保在 HTTP GET 挂钩中指定正确的 URI，否则您可能甚至不会注意到启动后挂钩不执行任何操作。

6.3.2 在容器终止之前运行进程

除了在容器启动时执行命令或发送 HTTP 请求之外，Kubernetes 还允许在容器中定义预停止钩子。

预停止钩子在容器终止之前立即执行。要终止进程，通常会向该进程发送 TERM 信号。这告诉应用程序完成它正在做的事情并关闭。容器也会发生同样的情况。每当容器需要停止或重新启动时，都会向容器中的主进程发送 TERM 信号。然而，在此之前，Kubernetes 首先执行预停止挂钩（如果为容器配置了预停止挂钩）。在预停止挂钩完成之前不会发送 TERM 信号，除非进程由于调用预停止挂钩处理程序本身而终止。

NOTE When container termination is initiated, the liveness and other probes are no longer invoked.

A pre-stop hook can be used to initiate a graceful shutdown of the container or to perform additional operations without having to implement them in the application itself. As with post-start hooks, you can either execute a command within the container or send an HTTP request to the application running in it.

USING A PRE-STOP LIFECYCLE HOOK TO SHUT DOWN A CONTAINER GRACEFULLY

The Nginx web server used in the fortune pod responds to the `TERM` signal by immediately closing all open connections and terminating the process. This is not ideal, as the client requests that are being processed at this time aren't allowed to complete.

Fortunately, you can instruct Nginx to shut down gracefully by running the command `nginx -s quit`. When you run this command, the server stops accepting new connections, waits until all in-flight requests have been processed, and then quits.

When you run Nginx in a Kubernetes pod, you can use a pre-stop lifecycle hook to run this command and ensure that the pod shuts down gracefully. The following listing shows the definition of this pre-stop hook. You can find it in the `fortune-prestop.yaml` pod manifest.

Listing 6.19 Defining a pre-stop hook for Nginx

```
    lifecycle:              #A
      preStop:              #A
        exec:               #B
          command:          #B
          - nginx           #C
          - -s              #C
          - quit            #C
```

#A This is a pre-stop lifecycle hook
#B It executes a command
#C This is the command that gets executed

Whenever a container using this pre-stop hook is terminated, the command `nginx -s quit` is executed in the container before the main process of the container receives the `TERM` signal.

Unlike the post-start hook, the container is terminated regardless of the result of the pre-stop hook - a failure to execute the command or a non-zero exit code does not prevent the container from being terminated. If the pre-stop hook fails, you'll see a `FailedPreStopHook` warning event among the pod events, but you might not see any indication of the failure if you are only monitoring the status of the pod.

TIP If successful completion of the pre-stop hook is critical to the proper operation of your system, make sure that it runs successfully. I've experienced situations where the pre-stop hook didn't run at all, but the engineers weren't even aware of it.

注意：当启动容器终止时，不再调用活性和其他探测器。

预停止挂钩可用于启动容器的正常关闭或执行其他操作，而无需在应用程序本身中实现它们。与启动后挂钩一样，您可以在容器内执行命令，也可以向在容器中运行的应用程序发送 HTTP 请求。

使用预停止生命周期钩子优雅地关闭容器

Fortune Pod 中使用的 Nginx Web 服务器通过立即关闭所有打开的连接并终止进程来响应 TERM 信号。这并不理想，因为此时正在处理的客户端请求不允许完成。

幸运的是，您可以通过运行命令 nginx -s quit 来指示 Nginx 正常关闭。当您运行此命令时，服务器停止接受新连接，

等待所有正在进行的请求都已处理，然后退出。

当您在 Kubernetes Pod 中运行 Nginx 时，可以使用预停止生命周期挂钩来运行此命令并确保 Pod 正常关闭。以下清单显示了此预停止挂钩的定义。您可以在 Fortune-prestop.yaml pod 清单中找到它。

清单 6.19 为 Nginx 定义预停止钩子

```
生命周期：#A
  停止前：#A
    执行：#B
      命令：#B

      - nginx #C

      - -s #C
```

#A 这是一个预停止生命周期
钩子 #B 它执行一个命令 #C
这是要执行的命令

每当使用此预停止挂钩的容器终止时，命令 `nginx -s quit`

在容器的主进程收到 TERM 信号之前在容器中执行。

与 post-start 挂钩不同，无论 prestop 挂钩的结果如何，容器都会终止 - 执行命令失败或非零退出代码不会阻止容器被终止。如果预停止挂钩失败，您将在 Pod 事件中看到 FailedPreStopHook 警告事件，但如果您仅监控 Pod 的状态，则可能看不到任何失败指示。

提示 如果成功完成预停止挂钩对于系统的正常运行至关重要，请确保它成功运行。我经历过预停止挂钩根本不运行的情况，但是
工程师甚至没有意识到这一点。

Like post-start hooks, you can also configure the pre-stop hook to send an HTTP GET request to your application instead of executing commands. The configuration of the HTTP GET pre-stop hook is the same as for a post-start hook. For more information, see section 6.3.1.

Application not receiving the TERM signal

Many developers make the mistake of defining a pre-stop hook just to send a `TERM` signal to their applications in the pre-stop hook. They do this when they find that their application never receives the `TERM` signal. The root cause is usually not that the signal is never sent, but that it is swallowed by something inside the container. This typically happens when you use the *shell* form of the `ENTRYPOINT` or the `CMD` directive in your Dockerfile. Two forms of these directives exist.

The *exec* form is: `ENTRYPOINT ["/myexecutable", "1st-arg", "2nd-arg"]`

The *shell* form is: `ENTRYPOINT /myexecutable 1st-arg 2nd-arg`

When you use the exec form, the executable file is called directly. The process it starts becomes the root process of the container. When you use the shell form, a shell runs as the root process, and the shell runs the executable as its child process. In this case, the shell process is the one that receives the `TERM` signal. Unfortunately, it doesn't pass this signal to the child process.

In such cases, instead of adding a pre-stop hook to send the `TERM` signal to your app, the correct solution is to use the exec form of `ENTRYPOINT` or `CMD`.

Note that the same problem occurs if you use a shell script in your container to run the application. In this case, you must either intercept and pass signals to the application or use the `exec` shell command to run the application in your script.

Pre-stop hooks are only invoked when the container is requested to terminate, either because it has failed its liveness probe or because the pod has to shut down. They are not called when the process running in the container terminates by itself.

UNDERSTANDING THAT LIFECYCLE HOOKS TARGET CONTAINERS, NOT PODS

As a final consideration on the post-start and pre-stop hooks, I would like to emphasize that these lifecycle hooks apply to containers and not to pods. You shouldn't use a pre-stop hook to perform an action that needs to be performed when the entire pod is shut down, because pre-stop hooks run every time the container needs to terminate. This can happen several times during the pod's lifetime, not just when the pod shuts down.

6.4 Understanding the pod lifecycle

So far in this chapter you've learned a lot about how the containers in a pod run. Now let's take a closer look at the entire lifecycle of a pod and its containers.

When you create a pod object, Kubernetes schedules it to a worker node that then runs its containers. The pod's lifecycle is divided into the three stages shown in the next figure:

与启动后挂钩一样，您还可以配置预停止挂钩以向应用程序发送 HTTP GET 请求，而不是执行命令。 HTTP GET prestop 挂钩的配置与 post-start 挂钩的配置相同。有关详细信息，请参阅第 6.3.1 节。

应用程序未收到 TERM 信号

许多开发人员会犯这样的错误：定义预停止钩子只是为了向其应用程序发送 TERM 信号

预停钩。当他们发现他们的应用程序从未收到 TERM 信号时，他们就会这样做。根本原因通常不是信号从未发送，而是信号被容器内的某些东西吞噬。这通常

当您在 Dockerfile 中使用 ENTRYPOINT 的 shell 形式或 CMD 指令时，就会发生这种情况。这些指令有两种形式。

exec 形式为： ENTRYPOINT ["/myexecutable", "1st-arg", "2nd-arg"] shell 形式为：ENTRYPOINT /myexecutable 1st-arg 2nd-arg 当使用 exec 形式时，调用可执行文件直接地。它启动的进程成为容器的根进程。当您使用 shell 形式时，shell 作为根进程运行，并且 shell 运行可执行文件

它的子进程。在这种情况下，正确的解决方案是接收 TERM 信号的进程。不幸的是，它不会将此信号传递给子进程。或 CMD 的执行形式。

请注意，如果您在容器中使用 shell 脚本来运行应用程序，也会出现同样的问题。在这种情况下，你

必须拦截信号并将其传递给应用程序，或者使用 exec shell 命令在脚本中运行应用程序

仅当容器被请求终止时才会调用预停止挂钩，因为容器的活动探测失败或 Pod 必须关闭。当容器中运行的进程自行终止时，不会调用它们。

了解生命周期挂钩目标容器，而不是 Pod

作为启动后和停止前钩子的最后考虑因素，我想强调这些生命周期钩子适用于容器而不是 Pod。您不应该使用 pre-stop 挂钩来执行整个 pod 关闭时需要执行的操作，因为每次容器需要终止时都会运行 pre-stop 挂钩。在 Pod 的生命周期中，这种情况可能会发生多次，而不仅仅是在 Pod 关闭时发生。

6.4 了解 Pod 生命周期

到目前为止，在本章中，您已经了解了很多有关 pod 中的容器如何运行的知识。现在让我们仔细看看 Pod 及其容器的整个生命周期。

当您创建 pod 对象时，Kubernetes 会将其调度到工作节点，然后运行其容器

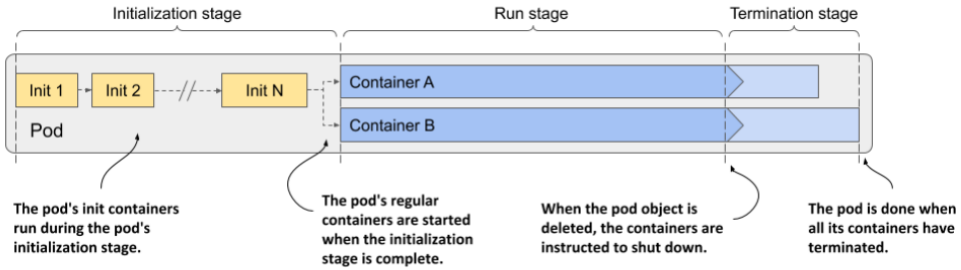


Figure 6.10 The three stages of the pod's lifecycle

The three stages of the pod's lifecycle are:

1. The initialization stage, during which the pod's init containers run.
2. The run stage, in which the regular containers of the pod run.
3. The termination stage, in which the pod's containers are terminated.

Let's see what happens in each of these stages.

6.4.1 Understanding the initialization stage

As you've already learned, the pod's init containers run first. They run in the order specified in the `initContainers` field in the pod's `spec`. Let me explain everything that unfolds.

PULLING THE CONTAINER IMAGE

Before each init container is started, its container image is pulled to the worker node. The `imagePullPolicy` field in the container definition in the pod specification determines whether the image is pulled every time, only the first time, or never.

Image pull policy	Description
Not specified	If the `imagePullPolicy` is not explicitly specified, it defaults to `Always` if the `:latest` tag is used in the image. For other image tags, it defaults to `IfNotPresent`.
`Always`	The image is pulled every time the container is (re)started. If the locally cached image matches the one in the registry, it is not downloaded again, but the registry still needs to be contacted.
`Never`	The container image is never pulled from the registry. It must exist on the worker node beforehand. Either it was stored locally when another container with the same image was deployed, or it was built on the node itself, or simply downloaded by someone or something else.
`IfNotPresent`	Image is pulled if it is not already present on the worker node. This ensures that the

图6.10 Pod生命周期的三个阶段

Pod 生命周期的三个阶段是：

1. 初始化阶段，Pod 的 init 容器运行。
2. 运行阶段，Pod 的常规容器在该阶段运行。
3. 终止阶段，Pod 的容器被终止。

让我们看看每个阶段会发生什么。

6.4.1 了解初始化阶段

正如您已经了解到的，Pod 的初始化容器首先运行。它们按照 Pod 规范中 initContainers 字段中指定的顺序运行。让我解释一下所发生的一切。

拉取容器镜像

每个init容器启动之前，其容器镜像都会被拉取到worker节点。 pod规范中容器定义中的imagePullPolicy字段决定是否

每次都会拉取图像，仅在第一次，或者从不拉取。

图片拉取政策	描述
未指定	如果未显式指定 imagePullPolicy，则在图像中使用 :latest 标签时，它默认为“Always”。对于其他图像标签，默认为 IfNotPresent。
总是	每次（重新）启动容器时都会拉取图像。如果本地缓存的图像与注册表中的图像匹配，则不会再次下载，但仍需要联系注册表。
绝不	容器镜像永远不会从注册表中提取。它必须预先存在于工作节点上。它要么是在部署具有相同映像的另一个容器时存储在本地，要么是在节点本身上构建，或者只是由某人或其他东西下载。
如果不存在	如果工作节点上尚不存在图像，则会拉取该图像。这确保了

image is only pulled the first time it's required.

Table 6.5 List of image-pull policies

The image-pull policy is also applied every time the container is restarted, so a closer look is warranted. Examine the following figure to understand the behavior of these three policies.

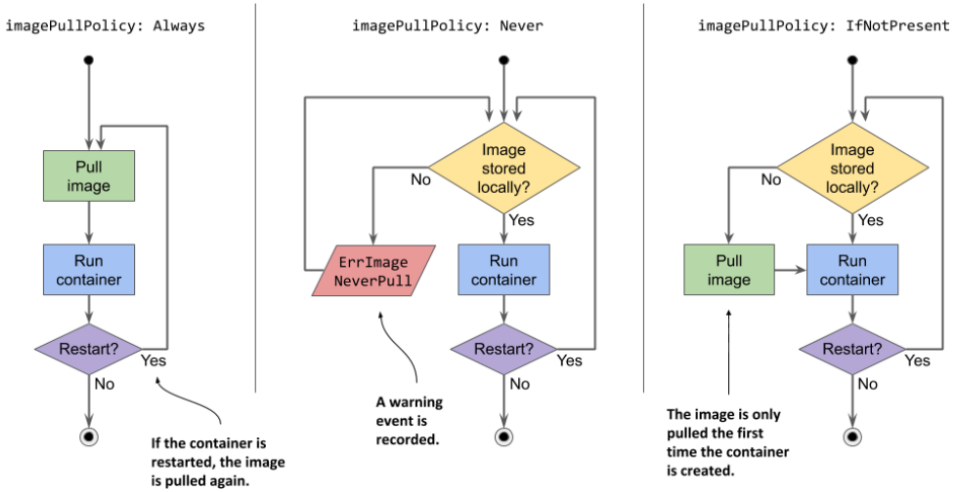


Figure 6.11 An overview of the three different image-pull policies

WARNING If the `imagePullPolicy` is set to `Always` and the image registry is offline, the container will not run even if the same image is already stored locally. A registry that is unavailable may therefore prevent your application from (re)starting.

RUNNING THE CONTAINERS

When the first container image is downloaded to the node, the container is started. When the first init container is complete, the image for the next init container is pulled and the container is started. This process is repeated until all init containers are successfully completed. Containers that fail might be restarted, as shown in the following figure.

171

图像仅在第一次需要时才被拉取。

表 6.5 镜像拉取策略列表

每次重新启动容器时也会应用映像拉取策略，因此有必要仔细检查。检查下图以了解这三个策略的行为。

图 6.11 三种不同的镜像拉取策略概览

警告 如果 imagePullPolicy 设置为 Always 并且镜像注册表处于脱机状态，容器将即使相同的图像已存储在本地，也不会运行。因此，不可用的注册表可能会阻止

你的应用程序从（重新）开始

运行容器

当第一个容器镜像下载到节点时，容器就会启动。当第一个初始化容器完成时，将拉取下一个初始化容器的映像并启动该容器。重复此过程，直到所有 init 容器都成功完成。失败的容器可能会重新启动，如下图所示。

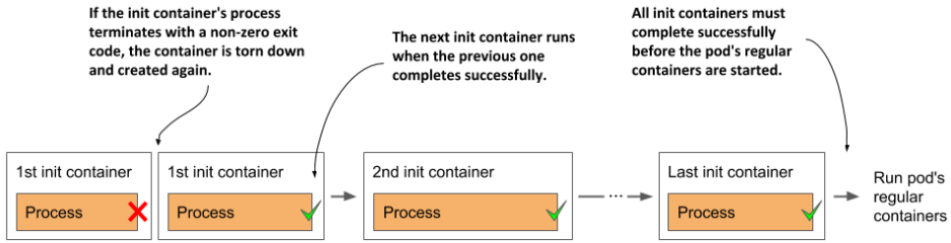


Figure 6.12 All init containers must run to completion before the regular containers can start

RESTARTING FAILED INIT CONTAINERS

If an init container terminates with an error and the pod's restart policy is set to `Always` or `OnFailure`, the failed init container is restarted. If the policy is set to `Never`, the subsequent init containers and the pod's regular containers are never started. The pod's status is displayed as `Init:Error` indefinitely. You must then delete and recreate the pod object to restart the application. For an example of such a pod, see the `fortune-init-fail-norestart.yaml` file in the book's code archive.

> **NOTE** If the container needs to be restarted and `imagePullPolicy` is set to `Always`, the container image is pulled again. If the container had terminated due to an error and you push a new image with the same tag that fixes the error, you don't need to recreate the pod, as the updated container image will be pulled before the container is restarted.

RE-EXECUTING THE POD'S INIT CONTAINERS

Init containers are normally only executed once. Even if one of the pod's main containers is terminated later, the pod's init containers are not re-executed. However, in exceptional cases, such as when Kubernetes must restart the entire pod, the pod's init containers might be executed again. This means that the operations performed by your init containers must be idempotent.

6.4.2 Understanding the run stage

When all init containers are successfully completed, the pod's regular containers are all created in parallel. In theory, the lifecycle of each container should be independent of the other containers in the pod, but this is not quite true. See sidebar for more information.

A container's post-start hook blocks the creation of the next container

The Kubelet doesn't start all containers of the pod at the same time. It creates and starts the containers synchronously in the order they are defined in the pod's `spec`. If a post-start hook is defined for a container, it runs

图 6.12 所有 init 容器必须运行完成后常规容器才能启动

重新启动失败的初始化容器

如果 init 容器因错误而终止，并且 pod 的重启策略设置为 Always 或 OnFailure，则失败的 init 容器将重新启动。如果策略设置为从不，则后续

init 容器和 pod 的常规容器永远不会启动。 Pod 的状态无限期地显示为 Init:Error。然后，您必须删除并重新创建 pod 对象才能重新启动应用程序。有关此类 pod 的示例，请参阅本书代码存档中的 Fortune-init-failnorestart.yaml 文件。

> 说明 如果需要重启容器，且imagePullPolicy设置为Always，则会再次拉取容器镜像。如果容器由于错误而终止，并且您推送具有相同标签的新映像
> 这修复了错误，您不需要重新创建 Pod，因为更新的容器映像将在之前拉取
> 容器重新启动

重新执行 Pod 的初始化容器

Init 容器通常只执行一次。即使 pod 的主容器之一稍后终止，pod 的 init 容器也不会重新执行。但是，在特殊情况下，例如 Kubernetes 必须重新启动整个 pod 时，pod 的 init 容器可能会再次执行。这意味着初始化容器执行的操作必须是幂等的。

6.4.2 了解运行阶段

当所有 init 容器成功完成后，pod 的常规容器将全部并行创建。理论上，每个容器的生命周期应该独立于 Pod 中的其他容器，但事实并非如此。请参阅侧边栏了解更多信息。

容器的启动后钩子会阻止下一个容器的创建

Kubelet 不会同时启动 pod 中的所有容器。它创建并启动容器

按照 pod 规范中定义的顺序同步进行。如果为容器定义了启动后钩子

asynchronously with the main container process, but the execution of the post-start hook handler blocks the creation and start of the subsequent containers.

This is an implementation detail that might change in the future.

In contrast, the termination of containers is performed in parallel. A long-running pre-stop hook does block the shutdown of the container in which it is defined, but it does not block the shutdown of other containers. The pre-stop hooks of the containers are all invoked at the same time.

The following sequence runs independently for each container. First, the container image is pulled, and the container is started. When the container terminates, it is restarted, if this is provided for in the pod's restart policy. The container continues to run until the termination of the pod is initiated. A more detailed explanation of this sequence is presented next.

PULLING THE CONTAINER IMAGE

Before the container is created, its image is pulled from the image registry, following the pod's `imagePullPolicy`. Once the image is pulled, the container is created.

> **NOTE** If one of the container images can't be pulled, the other containers run anyway.

> **WARNING** Containers don't necessarily start at the same moment. If pulling the image takes a long time, the container may start well after all the others have already started. Bear this in mind if one of your containers depends on another.

RUNNING THE CONTAINER

The container starts when the main container process starts. If a post-start hook is defined in the container, it is invoked in parallel with the main container process. The post-start hook runs asynchronously and must be successful for the container to continue running.

Together with the main container and the potential post-start hook process, the startup probe, if defined for the container, is started. When the startup probe is successful, or if the startup probe is not configured, the liveness probe is started.

TERMINATING AND RESTARTING THE CONTAINER ON FAILURES

If the startup or the liveness probe fails so often that it reaches the configured failure threshold, the container is terminated. As with init containers, the pod's `restartPolicy` determines whether the container is then restarted or not.

Perhaps surprisingly, if the restart policy is set to `Never` and the startup hook fails, the pod's status is shown as `Completed` even though the post-start hook failed. You can see this for yourself by creating the pod in the `fortune-poststart-fail-norestart.yaml` file.

INTRODUCING THE TERMINATION GRACE PERIOD

If a container must be terminated, the container's pre-stop hook is called so that the application can shut down gracefully. When the pre-stop hook is completed, or if no pre-stop hook is defined, the `TERM` signal is sent to the main container process. This is another hint to the application that it should shut down.

173

与主容器进程异步，但启动后钩子处理程序的执行会阻止后续容器的创建和启动。

这是一个将来可能会改变的实现细节。
相反，容器的终止是并行执行的。长时间运行的预停止钩子确实会阻止定义它的容器的关闭，但不会阻止其他容器的关闭。容器的预停止钩子全部同时调用。

以下序列针对每个容器独立运行。首先，拉取容器镜像，并启动容器。当容器终止时，它会重新启动（如果 Pod 的重新启动策略中提供了此规定）。容器将继续运行，直到启动 pod 终止。接下来将对此序列进行更详细的解释。

拉取容器镜像

在创建容器之前，会按照 pod 的 imagePullPolicy 从镜像注册表中提取其镜像。拉取镜像后，就会创建容器。

> 注意：如果无法拉取其中一个容器映像，其他容器仍会运行。

> 警告容器不一定同时启动。如果拉取镜像需要很长时间，则容器可能会在所有其他容器都启动之后才启动。如果您的其中一个人请记住这一点
> 容器依赖于另一个容器。

运行容器

容器在主容器进程启动时启动。如果在容器中定义了启动后钩子，则它会与主容器进程并行调用。启动后挂钩异步运行，必须成功容器才能继续运行。

启动探针（如果为容器定义）将与主容器和潜在的启动后挂钩进程一起启动。当启动探针成功时，或者如果未配置启动探针，则启动活性探针。

发生故障时终止并重新启动容器

如果启动或活性探针频繁失败以致达到配置的失败阈值，则容器将被终止。与 init 容器一样，pod 的 restartPolicy 决定容器是否重新启动。

也许令人惊讶的是，如果重新启动策略设置为“从不”并且启动挂钩失败，即使启动后挂钩失败，Pod 的状态也会显示为“已完成”。您可以通过在 Fortune-poststart-fail-norestart.yaml 文件中创建 pod 来亲自查看这一点。

介绍终止宽限期

如果必须终止容器，则会调用容器的预停止钩子，以便应用程序可以正常关闭。当预停止挂钩完成时，或者如果未定义预停止挂钩，则将 TERM 信号发送到主容器进程。这是应用程序应该关闭的另一个提示。

The application is given a certain amount of time to terminate. This time can be configured using the `terminationGracePeriodSeconds` field in the pod's `spec` and defaults to 30 seconds. The timer starts when the pre-stop hook is called or when the `TERM` signal is sent if no hook is defined. If the process is still running after the termination grace period has expired, it's terminated by force via the `KILL` signal. This terminates the container.

The following figure illustrates the container termination sequence.

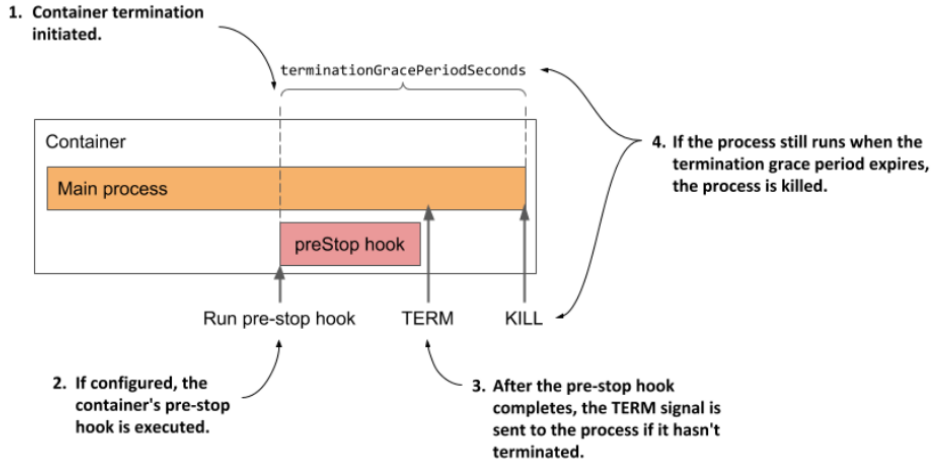


Figure 6.13 A container's termination sequence

After the container has terminated, it will be restarted if the pod's restart policy allows it. If not, the container will remain in the `Terminated` state, but the other containers will continue running until the entire pod is shut down or until they fail as well.

6.4.3 Understanding the termination stage

The pod's containers continue to run until you finally delete the pod object. When this happens, termination of all containers in the pod is initiated and its status is changed to `Terminating`.

INTRODUCING THE DELETION GRACE PERIOD

The termination of each container at pod shutdown follows the same sequence as when the container is terminated because it has failed its liveness probe, except that instead of the termination grace period, the pod's *deletion grace period* determines how much time is available to the containers to shut down on their own.

174

该应用程序有一定的时间来终止。该时间可以使用 Pod 规范中的 TerminationGracePeriodSeconds 字段进行配置，默认为 30 秒。当调用预停止钩子时或者当发送 TERM 信号（如果没有定义钩子）时，计时器启动。如果终止宽限期过后进程仍在运行，则会通过 KILL 信号强制终止它。这将终止容器。下图说明了容器终止顺序。

图 6.13 容器的终止序列

容器终止后，如果 pod 的重启策略允许，它将重新启动。如果没有，容器将保持在 Termerated 状态，但其他容器将继续运行，直到整个 Pod 关闭或它们也发生故障。

6.4.3 了解终止阶段

Pod 的容器会继续运行，直到您最终删除 Pod 对象。发生这种情况时，将启动 pod 中所有容器的终止，并将其状态更改为“正在终止”。

删除宽限期简介

Pod 关闭时每个容器的终止遵循与容器因活动探测失败而终止时相同的顺序，只不过 Pod 的删除宽限期不是终止宽限期，而是决定了容器可以使用多少时间来执行任务。自行关闭。

This grace period is defined in the pod's `metadata.deletionGracePeriodSeconds` field, which gets initialized when you delete the pod. By default, it gets its value from the `spec.terminationGracePeriodSeconds` field, but you can specify a different value in the `kubectl delete` command. You'll see how to do this later.

UNDERSTANDING HOW THE POD'S CONTAINERS ARE TERMINATED

As shown in the next figure, the pod's containers are terminated in parallel. For each of the pod's containers, the container's pre-stop hook is called, the `TERM` signal is then sent to the main container process, and finally the process is terminated using the `KILL` signal if the deletion grace period expires before the process stops by itself. After all the containers in the pod have stopped running, the pod object is deleted.

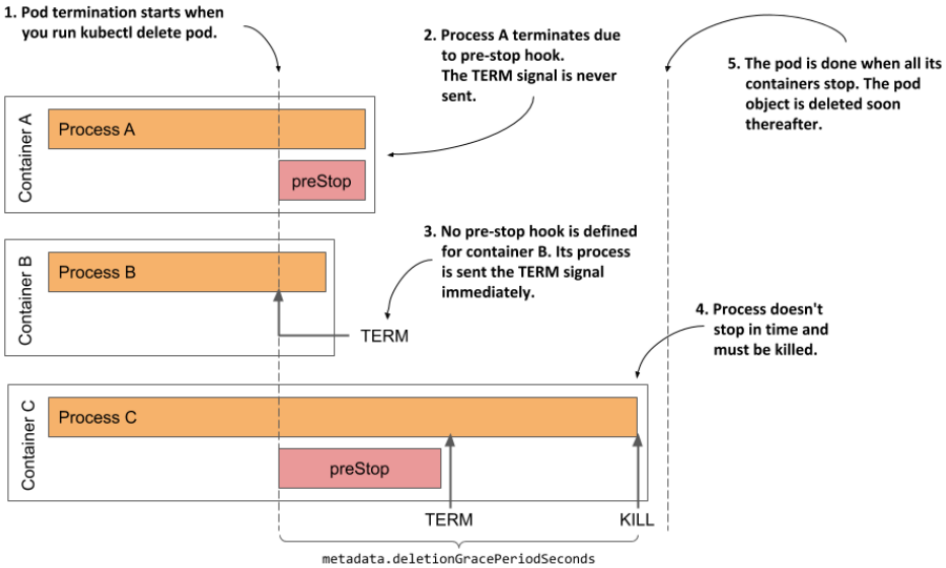


Figure 6.14 The termination sequence inside a pod

INSPECTING THE SLOW SHUTDOWN OF A POD

Let's look at this last stage of the pod's life on one of the pods you created previously. If the `kubia-ssl` pod doesn't run in your cluster, please create it again. Now delete the pod by running `kubectl delete kubia-ssl`.

175

此宽限期在 pod 的metadata.deletionGracePeriodSeconds 字段中定义，该字段会在您删除 pod 时初始化。默认情况下，它从 spec.terminationGracePeriodSeconds 字段获取值，但您可以在

kubectl 删除命令。稍后您将看到如何执行此操作。
了解 Pod 的容器如何终止

如下图所示，Pod 的容器并行终止。对于每个 pod 的容器，调用容器的 pre-stop 钩子，然后将 TERM 信号发送到主容器进程，最后，如果在进程自行停止之前删除宽限期到期，则使用 KILL 信号终止该进程。当 Pod 中的所有容器停止运行后，Pod 对象将被删除。

图 6.14 pod 内的终止序列

检查 Pod 缓慢关闭情况

让我们在您之前创建的一个 pod 上看看 pod 生命周期的最后阶段。如果 kubia-ssl pod 未在您的集群中运行，请重新创建它。现在删除 pod

运行 kubectl 删除 kubia-ssl。

It takes surprisingly long to delete the pod, doesn't it? I counted at least 30 seconds. This is neither normal nor acceptable, so let's fix it.

Considering what you've learned in this section, you may already know what's causing the pod to take so long to finish. If not, let me help you analyze the situation.

The `kubia-ssl` pod has two containers. Both must stop before the pod object can be deleted. Neither container has a pre-stop hook defined, so both containers should receive the `TERM` signal immediately when you delete the pod. The 30s I mentioned earlier match the default termination grace period value, so it looks like one of the containers, if not both, doesn't stop when it receives the `TERM` signal, and is killed after the grace period expires.

CHANGING THE TERMINATION GRACE PERIOD

You could set the pod's `terminationGracePeriodSeconds` field to a lower value, as shown in the following listing, and see if the pod shuts down faster.

Listing 6.20 Setting a lower terminationGracePeriodSeconds for faster pod shutdown

```
apiVersion: v1
kind: Pod
metadata:
  name: kubia-ssl-shortgraceperiod
spec:
  terminationGracePeriodSeconds: 5              #A
  containers:
  ...
```

#A This pod's containers have 5 seconds to terminate after receiving the TERM signal or they will be killed

In the listing above, the pod's `terminationGracePeriodSeconds` is set to `5`. If you create and then delete this pod, you'll see that its containers are terminated within 5s of receiving the `TERM` signal.

> **TIP** A reduction of the termination grace period is rarely necessary. However, it is advisable to extend it if the application usually needs more time to shut down gracefully.

SPECIFYING THE DELETION GRACE PERIOD WHEN DELETING THE POD

Any time you delete a pod, the pod's `terminationGracePeriodSeconds` determines the amount of time the pod is given to shut down, but you can override this time when you execute the `kubectl delete` command using the `--grace-period` command line option.

For example, to give the pod 10s to shut down, you run the following command:

```
$ kubectl delete po kubia-ssl --grace-period 10
```

> **NOTE** If you set this grace period to zero, the pod's pre-stop hooks are not executed.

FIXING THE SHUTDOWN BEHAVIOR OF THE KUBIA APPLICATION

Considering that the shortening of the grace period leads to a faster shutdown of the pod, it's clear that at least one of the two containers doesn't terminate by itself after it receives the

删除 Pod 需要花费很长的时间，不是吗？我数了数，至少有30秒。这
这既不正常也不可接受，所以让我们解决它。

考虑到您在本节中学到的内容，您可能已经知道是什么原因造成的
吊舱需要很长时间才能完成。如果没有的话，我来帮你分析一下情况。

kubia-ssl pod 有两个容器。两者都必须停止，然后才能删除 pod 对象。这两个容器都没有定义预停止钩子，因此当您删除 Pod 时，两个容器都应该立即收到 TERM 信号。我之前提到的 30 年代与

默认终止宽限期值，因此看起来容器之一（如果不是两个）在收到 TERM 信号时不会停止，并在宽限期到期后被终止。

更改终止宽限期

您可以将 pod 的终止GracePeriodSeconds 字段设置为较低的值，如以下列表所示，并查看 pod 是否关闭得更快。

清单 6.20 设置较低的终止GracePeriodSeconds 以加快 pod 关闭速度

```
api版本：v1
种类：Pod 元
数据
  名称：kubia-ssl-shortgraceperiod
规格：
  终止宽限期秒：5 #A

  容器：
```

#A 该 pod 的容器在收到 TERM 信号后有 5 秒的时间终止，否则将被杀死

在上面的列表中，pod 的终止GracePeriodSeconds 设置为 5。如果您创建然后删除此 pod，您将看到其容器在收到 TERM 信号后 5 秒内终止。

提示 很少需要缩短终止宽限期。但是，如果应用程序通常需要更多时间才能正常关闭，则建议延长它。

删除 Pod 时指定删除宽限期

任何时候删除 pod 时，pod 的终止GracePeriodSeconds 都会确定 pod 关闭的时间，但您可以在使用 --grace-period 命令行选项执行 kubectl delete 命令时覆盖此时间。例如，要让 pod 在 10 秒内关闭，请运行以下命令：

```
$ kubectl  删除  po kubia-ssl --grace-period 10
```

注意如果将此宽限期设置为零，则不会执行 Pod 的预停止挂钩。

修复 KUBIA 应用程序的关闭行为

考虑到宽限期的缩短会导致 pod 更快关闭

TERM signal. To see which one, recreate the pod, then run the following commands to stream the logs of each container before deleting the pod again:

```
$ kubectl logs kubia-ssl -c kubia -f
$ kubectl logs kubia-ssl -c envoy -f
```

The logs show that the Envoy proxy catches the signal and immediately terminates, whereas the Node.js application doesn't seem to respond to the signal. To fix this, you need to add the code shown in the following listing to the end of the app.js file. You can find the modified file in the Chapter06/kubia-v2-image directory in the code archive of the book.

Listing 6.21 Handling the TERM signal in the kubia application

```
process.on('SIGTERM', function () {
  console.log("Received SIGTERM. Server shutting down...");
  server.close(function () {
    process.exit(0);
  });
});
```

After you make the change to the code, create a new container image with the tag :1.1, push it to your image registry, and deploy a new pod that uses the new image. If you don't want to bother to create the image, you can also use the image luksa/kubia:1.1, which is published on Docker Hub. To create the pod, apply the manifest in the file kubia-ssl-v1-1.yaml, which you can find in the book's code archive.

If you delete this new pod, you'll see that it shuts down considerably faster. From the logs of the kubia container, you can see that it begins to shut down as soon as it receives the TERM signal.

TIP You should make sure that your init containers also handle the TERM signal so that they shut down immediately if you delete the pod object while it's still being initialized.

6.4.4 Visualizing the full lifecycle of the pod's containers

To conclude this chapter on what goes on in a pod, I present a final overview of everything that happens during the life of a pod. The following two figures summarize everything that has been explained in this chapter. The initialization of the pod is shown in the next figure.

术语信号。要查看是哪一个，请重新创建 pod，然后运行以下命令进行流式传输

再次删除 Pod 之前每个容器的日志：

```
$ kubectl 日志 kubia-ssl -c
kubia -f $ kubectl 日志 kubia-
ssl -c envoy -f
```

日志显示 Envoy 代理捕获了信号并立即终止，而 Node.js 应用程序似乎没有响应该信号。要解决此问题，您需要将以下清单中所示的代码添加到 app.js 文件的末尾。您可以找到

修改后的文件位于本书代码存档的Chapter06/kubia-v2-image目录中。

清单 6.21 在 kubia 应用程序中处理 TERM 信号

```
process.on('SIGTERM', 函数 () {
  console.log("收到 SIGTERM。服务器正在关闭...");
  server.close(function () { process.exit(0);
});
});
```

对代码进行更改后，创建一个带有标签 :1.1 的新容器映像，将其推送到映像注册表，然后部署一个使用新映像的新 pod。如果你不想费心创建镜像，也可以使用镜像luksa/kubia:1.1，即

发布在 Docker Hub 上。要创建 pod，请应用文件 kubia-ssl-v11.yaml 中的清单，您可以在本书的代码存档中找到该文件。

如果删除这个新 Pod，您会发现它关闭的速度要快得多。从kubia容器的日志中可以看到，它一收到就开始关闭

术语信号。

提示 您应该确保您的 init 容器也处理 TERM 信号，以便在
您删除仍在初始化的 pod 对象时它们会立即关闭。

6.4.4 可视化 Pod 容器的完整生命周期

为了结束关于 Pod 中发生的事情的本章，我对 Pod 生命周期中发生的所有事情进行了最后的概述。下面两张图总结了本章中解释的所有内容。 pod的初始化如下图所示。

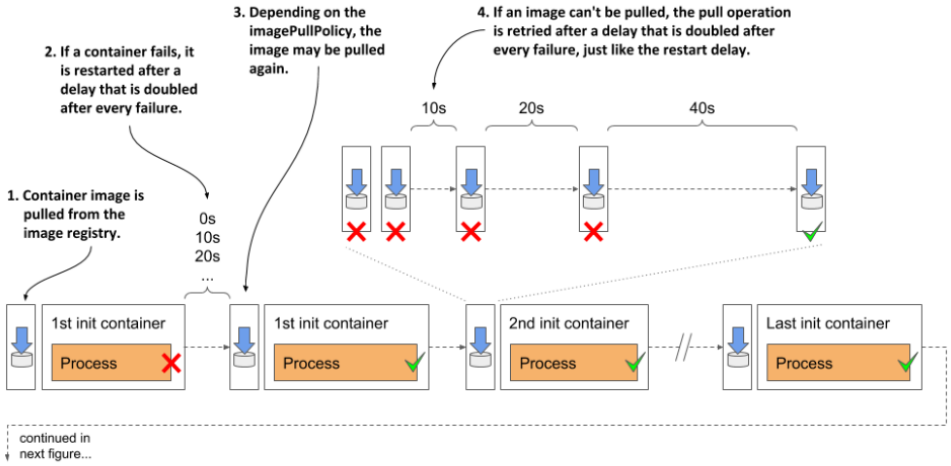


Figure 6.15 Complete overview of the pod's initialization stage

When initialization is complete, normal operation of the pod's containers begins. This is shown in the next figure.

178

图 6.15 Pod 初始化阶段的完整概述

初始化完成后，Pod 容器开始正常运行。如下图所示。

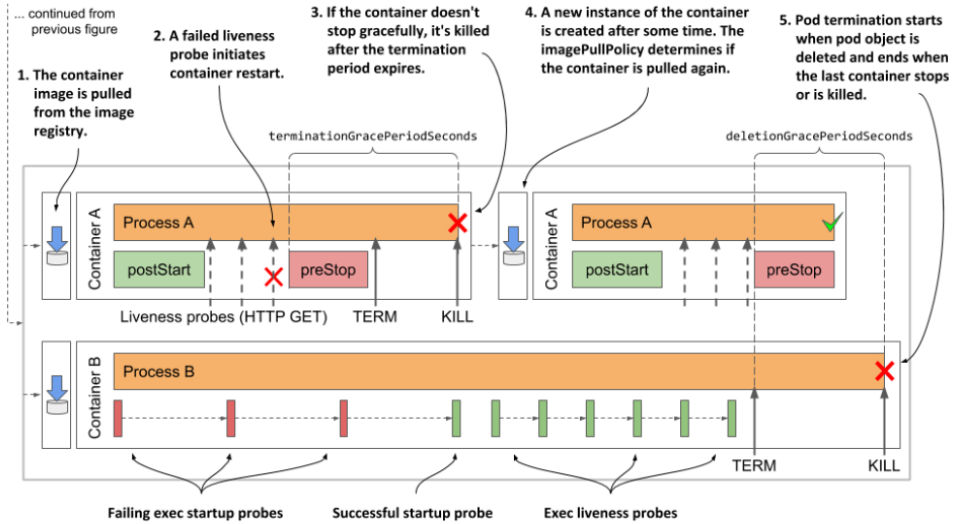


Figure 6.16 Complete overview of the pod's normal operation

6.5 Summary

In this chapter, you've learned:

- The status of the pod contains information about the phase of the pod, its conditions, and the status of each of its containers. You can view the status by running the `kubectl describe` command or by retrieving the full pod manifest using the command `kubectl get -o yaml`.
- Depending on the pod's restart policy, its containers can be restarted after they are terminated. In reality, a container is never actually restarted. Instead, the old container is destroyed, and a new container is created in its place.
- If a container is repeatedly terminated, an exponentially increasing delay is inserted before each restart. There is no delay for the first restart, then the delay is 10 seconds and then doubles before each subsequent restart. The maximum delay is 5 minutes and is reset to zero when the container has been running properly for at least twice this time.
- An exponentially increasing delay is also used after each failed attempt to download a container image.
- Adding a liveness probe to a container ensures that the container is restarted when it stops responding. The liveness probe checks the state of the application via an HTTP GET request, by executing a command in the container, or opening a TCP connection

图 6.16 Pod 正常运行的完整概览

6.5 总结

在本章中，您学习了：

- Pod 的状态包含有关 Pod 的阶段、其条件及其每个容器的状态的信息。您可以通过运行 kubectl describe 命令或使用以下命令检索完整的 pod 清单来查看状态：

  命令 kubectl get -o yaml。

- 根据 Pod 的重启策略，其容器可以在终止后重新启动。实际上，容器永远不会真正重新启动。相反，旧容器将被销毁，并在其位置创建新容器。

- 如果容器被重复终止，则在每次重新启动之前都会插入指数增加的延迟。第一次重新启动没有延迟，然后延迟 10 秒，然后在后续每次重新启动之前延迟加倍。最大延迟时间为 5 分钟，当容器正常运行至少两次后，延迟时间会重置为零。

- 每次尝试下载容器映像失败后，也会使用呈指数增加的延迟。

- 将活性探测器添加到容器可确保容器在停止响应时重新启动。 liveness 探针通过 HTTP GET 请求检查应用程序的状态，方法是在容器中执行命令

to one of the network ports of the container.

- If the application needs a long time to start, a startup probe can be defined with settings that are more forgiving than those in the liveness probe to prevent premature restarting of the container.
- You can define lifecycle hooks for each of the pod's main containers. A post-start hook is invoked when the container starts, whereas a pre-stop hook is invoked when the container must shut down. A lifecycle hook is configured to either send an HTTP GET request or execute a command within the container.
- If a pre-stop hook is defined in the container and the container must terminate, the hook is invoked first. The `TERM` signal is then sent to the main process in the container. If the process doesn't stop within `terminationGracePeriodSeconds` after the start of the termination sequence, the process is killed.
- When you delete a pod object, all its containers are terminated in parallel. The pod's `deletionGracePeriodSeconds` is the time given to the containers to shut down. By default, it's set to the termination grace period, but can be overridden with the `kubectl delete` command.
- If shutting down a pod takes a long time, it is likely that one of the processes running in it doesn't handle the `TERM` signal. Adding a `TERM` signal handler is a better solution than shortening the termination or deletion grace period.

You now understand everything about the operation of containers in pods. In the next chapter you'll learn about the other important component of pods - storage volumes.

180

到容器的网络端口之一。

- 如果应用程序需要很长时间才能启动，则可以使用比活性探针中的设置更宽松的设置来定义启动探针，以防止容器过早重新启动。

• 您可以为每个 Pod 的主容器定义生命周期挂钩。启动后钩子在容器启动时调用，而停止前钩子在容器必须关闭时调用。生命周期挂钩配置为发送 HTTP GET 请求或在容器内执行命令。

• 如果在容器中定义了预停止钩子并且容器必须终止，则首先调用该钩子。然后 TERM 信号被发送到容器中的主进程。如果进程在终止序列开始后的terminationGracePeriodSeconds内没有停止，则进程将被终止。

• 当您删除Pod 对象时，其所有容器将同时终止。 Pod 的删除GracePeriodSeconds 是给予容器关闭的时间。经过

默认情况下，它设置为终止宽限期，但可以使用 kubectl delete 命令覆盖。

• 如果关闭 pod 需要很长时间，则可能是其中运行的进程之一无法处理 TERM 信号。添加 TERM 信号处理程序是比缩短终止或删除宽限期更好的解决方案。

您现在已经了解了有关 Pod 中容器操作的所有内容。在下一章中，您将了解 Pod 的另一个重要组件 - 存储卷。

7

Mounting storage volumes into the Pod's containers

This chapter covers

- **Persisting files across container restarts**
- **Sharing files between containers of the same pod**
- **Sharing files between pods**
- **Attaching network storage to pods**
- **Accessing the host node filesystem from within a pod**

The previous two chapters focused on the pod's containers, but they are only half of what a pod typically contains. They are typically accompanied by storage volumes that allow a pod's containers to store data for the lifetime of the pod or beyond, or to share files with the other containers of the pod. This is the focus of this chapter.

7.1 Introducing volumes

A pod is like a small logical computer that runs a single application. This application can consist of one or more containers that run the application processes. These processes share computing resources such as CPU, RAM, network interfaces, and others. In a typical computer, the processes use the same filesystem, but this isn't the case with containers. Instead, each container has its own isolated filesystem provided by the container image.

When a container starts, the files in its filesystem are those that were added to its container image during build time. The process running in the container can then modify those files or create new ones. When the container is terminated and restarted, all changes it made to its files are lost, because the previous container is not really restarted, but completely replaced, as

将存储卷安装到 Pod 的容器

本章涵盖

- 跨容器重启保留文件
- 同一 Pod 的容器之间共享文件
- 在 Pod 之间共享文件
- 将网络存储附加到 Pod
- 从 Pod 内访问主机节点文件系统

前两章重点介绍了 pod 的容器，但它们只是 pod 的一半

通常包含。它们通常附带存储卷，允许 Pod 的容器在 Pod 的生命周期内或更长时间内存储数据，或与其他容器共享文件

Pod 的容器。这是本章的重点。

7.1 介绍卷

Pod 就像一台运行单个应用程序的小型逻辑计算机。该应用程序可以包括

运行应用程序进程的一个或多个容器。这些进程共享计算资源，例如 CPU、RAM、网络接口等。在典型的计算机中，

进程使用相同的文件系统，但容器的情况并非如此。相反，每个容器都有其自己的由容器映像提供的独立文件系统。

当容器启动时，其文件系统中的文件是在构建期间添加到其容器映像中的文件。然后，容器中运行的进程可以修改这些文件或

创建新的。当容器终止并重新启动时，它对其文件所做的所有更改

都丢失了，因为之前的容器并不是真正重启，而是完全替换掉了。如

explained in the previous chapter. Therefore, when a containerized application is restarted, it can't continue from the point where it was when it stopped. Although this may be okay for some types of applications, others may need the filesystem or at least part of it to be preserved on restart.

This is achieved by adding a *volume* to the pod and *mounting* it into the container.

DEFINITION *Mounting* is the act of attaching the filesystem of some storage device or volume into a specific location in the operating system's file tree, as shown in figure 7.1. The contents of the volume are then available at that location.

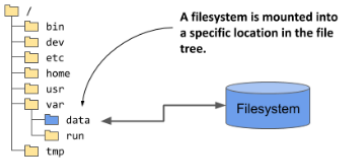


Figure 7.1 Mounting a filesystem into the file tree

7.1.1 Understanding how volumes fit into pods

Like containers, volumes aren't top-level resources like pods or nodes, but are a component within the pod and thus share its lifecycle. As the following figure shows, a volume is defined at the pod level and then mounted at the desired location in the container.

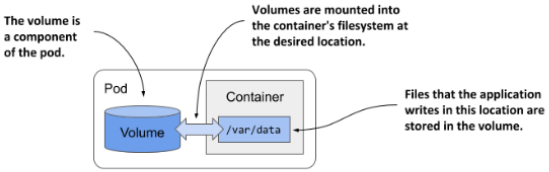


Figure 7.2 Volumes are defined at the pod level and mounted in the pod's containers

The lifecycle of a volume is tied to the lifecycle of the entire pod and is independent of the lifecycle of the container in which it is mounted. Due to this fact, volumes are also used to persist data across container restarts.

PERSISTING FILES ACROSS CONTAINER RESTARTS

All volumes in a pod are created when the pod is set up - before any of its containers are started. They are torn down when the pod is shut down.

182

前一章已解释过。因此，当容器化应用程序重新启动时，

无法从停止时的位置继续。虽然这对某些人来说可能没问题

类型的应用程序，其他人可能需要文件系统或至少部分保存在

重启。这是通过向 pod 添加卷并将其安装到容器中来实现的。

定义 挂载是将某些存储设备或卷的文件系统附加到特定位置的行为
在操作系统文件树中的位置，如图7.1所示。然后该卷的内容就可用了
在那个位置

图 7.1 将文件系统挂载到文件树中

7.1.1 了解卷如何适合 Pod

与容器一样，卷不是像 pod 或节点这样的顶级资源，而是一个组件

在 Pod 内，从而共享其生命周期。如下图所示，在 Pod 级别定义一个卷，然后将其挂载到容器中所需的位置。

图 7.2 卷在 Pod 级别定义并安装在 Pod 的容器中

卷的生命周期与整个 Pod 的生命周期相关，并且独立于

它所安装的容器的生命周期。因此，卷还用于在容器重新启动时保存数据。

跨容器重启保留文件

Pod 中的所有卷都是在 Pod 设置时创建的 - 在其任何容器启动之前。

当 Pod 关闭时，它们会被拆除。

Each time a container is (re)started, the volumes that the container is configured to use are mounted in the container's filesystem. The application running in the container can read from the volume and write to it if the volume and mount are configured to be writable.

A typical reason for adding a volume to a pod is to the persistence of data across container restarts. If no volume is mounted in the container, the entire filesystem of the container is ephemeral. Since a container restart replaces the entire container, its filesystem is also re-created from the container image. As a result, all files written by the application are lost.

If, on the other hand, the application writes data to a volume mounted inside the container, as shown in the following figure, the application process in the new container can access the same data after the container is restarted.

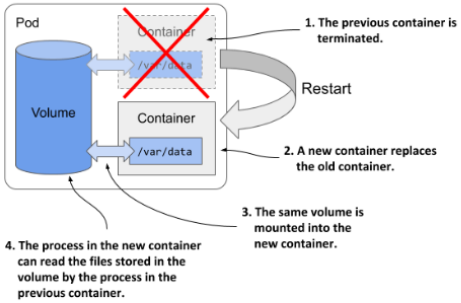


Figure 7.3 Volumes ensure that part of the container's filesystem is persisted across restarts

It is up to the author of the application to determine which files must be retained on restart. Normally you want to preserve data representing the application's state, but you may not want to preserve files that contain the application's locally cached data, as this prevents the container from starting fresh when it's restarted. Starting fresh every time may allow the application to heal itself when corruption of the local cache causes it to crash. Just restarting the container and using the same corrupted files could result in an endless crash loop.

> **TIP** Before you mount a volume in a container to preserve files across container restarts, consider how this affects the container's self-healing capability.

MOUNTING MULTIPLE VOLUMES IN A CONTAINER

A pod can have multiple volumes and each container can mount zero or more of these volumes in different locations, as shown in the following figure.

183

每次（重新）启动容器时，容器配置使用的卷是
安装在容器的文件系统中。容器中运行的应用程序可以读取

如果卷和安装配置为可写，则写入该卷。

将卷添加到 pod 的一个典型原因是为了跨容器保存数据
重新启动。如果容器中没有挂载卷，则容器的整个文件系统

短暂的。由于容器重新启动会替换整个容器，因此其文件系统也会重新启动

从容器镜像创建。应用（应用程序写入）的所有文件都会丢失

另一方面，如果应用程序将数据写入容器内安装的卷，
如下图所示，容器重启后，新容器中的应用进程可以访问相同的数据。

图 7.3 卷确保容器文件系统的一部分在重新启动后仍保留

由应用程序的作者决定重新启动时必须保留哪些文件。

通常，您希望保留代表应用程序状态的数据，但您可能不想保留包含应用程序本地缓存数据的文件，因为这会阻止容器

重新启动时从新开始。当本地缓存损坏导致应用程序崩溃时，每次重新启动都可以让应用程序自行修复。只需重启容器即可

使用相同的损坏文件可能会导致无限的崩溃循环。

提示 在将卷挂载到容器中以在容器重新启动后保留文件之前，请考虑如何这样做
影响容器的自愈能力。

在容器中安装多个卷

一个 Pod 可以有多个卷，每个容器可以挂载零个或多个这些卷

在不同的位置，如下图所示。

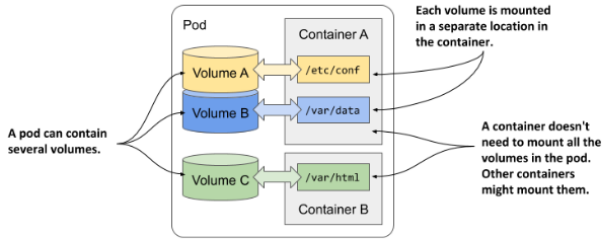


Figure 7.4 A pod can contain multiple volumes and a container can mount multiple volumes

The reason why you might want to mount multiple volumes in one container is that these volumes may serve different purposes and can be of different types with different performance characteristics.

In pods with more than one container, some volumes can be mounted in some containers but not in others. This is especially useful when a volume contains sensitive information that should only be accessible to some containers.

SHARING FILES BETWEEN MULTIPLE CONTAINERS

A volume can be mounted in more than one container so that applications running in these containers can share files. As discussed in chapter 5, a pod can combine a main application container with sidecar containers that extend the behavior of the main application. In some cases, the containers must read or write the same files.

For example, you could create a pod that combines a web server running in one container with a content-producing agent running in another container. The content agent container generates the static content that the web server then delivers to its clients. Each of the two containers performs a single task that has no real value on its own. However, as the next figure shows, if you add a volume to the pod and mount it in both containers, you enable these containers to become a complete system that provides a valuable service and is more than the sum of its parts.

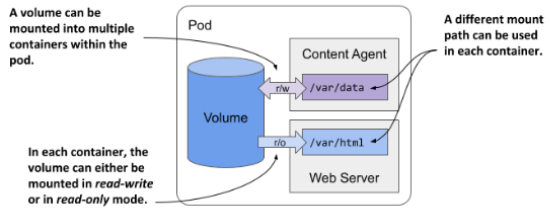


Figure 7.5 A volume can be mounted into more than one container

图7.4 一个pod可以包含多个卷，一个容器可以挂载多个卷

您可能想要在一个容器中安装多个卷的原因是这些

卷可能有不同的用途，并且可以是具有不同性能的不同类型

特征在具有多个容器的 Pod 中，某些卷可以安装在某些容器中
但其他人则不然。当卷包含只能由某些容器访问的敏感信息时，这尤其有用。

在多个容器之间共享文件

一个卷可以安装在多个容器中，以便应用程序在这些容器中运行

容器可以共享文件。正如第 5 章中所讨论的，pod 可以将主应用程序容器与扩展主应用程序行为的 sidecar 容器结合起来。在一些

在这种情况下，容器必须读取或写入相同的文件

例如，您可以创建一个 pod，将运行在一个容器中的 Web 服务器组合在一起
内容生成代理在另一个容器中运行。内容代理容器生成静态内容，然后 Web 服务器将其传送到其客户端。两者中的每一个

容器执行的任务本身没有实际价值。但是，如下图所示，如果您向 pod 添加卷并将其挂载到两个容器中，则可以启用这些

容器成为一个完整的系统，提供有价值的服务，并且不仅仅是其各个部分的总和。

图 7.5 一个卷可以挂载到多个容器中

The same volume can be mounted at different places in each container, depending on the needs of the container itself. If the content agent writes content to `/var/data`, it makes sense to mount the volume there. Since the web server expects the content to be in `/var/html`, the container running it has the volume mounted at this location.

In the figure you'll also notice that the volume mount in each container can be configured either as read/write or as read-only. Because the content agent needs to write to the volume whereas the web server only reads from it, the two mounts are configured differently. In the interest of security, it's advisable to prevent the web server from writing to the volume, since this could allow an attacker to compromise the system if the web server software has a vulnerability that allows attackers to write arbitrary files to the filesystem and execute them.

Other examples of using a single volume in two containers are cases where a sidecar container runs a tool that processes or rotates the web server logs or when an init container creates configuration files for the main application container.

PERSISTING DATA ACROSS POD RESTARTS

A volume is tied to the lifecycle of the pod and only exists for as long as the pod exists, but depending on the volume type, the files in the volume can remain intact after the pod and volume disappear and can later be mounted into a new volume.

As the following figure shows, a pod volume can map to persistent storage outside the pod. In this case, the file directory representing the volume isn't a local file directory that persists data only for the duration of the pod, but is instead a mount to an existing, typically network-attached storage volume (NAS) whose lifecycle isn't tied to any pod. The data stored in the volume is thus persistent and can be used by the application even after the pod it runs in is replaced with a new pod running on a different worker node.

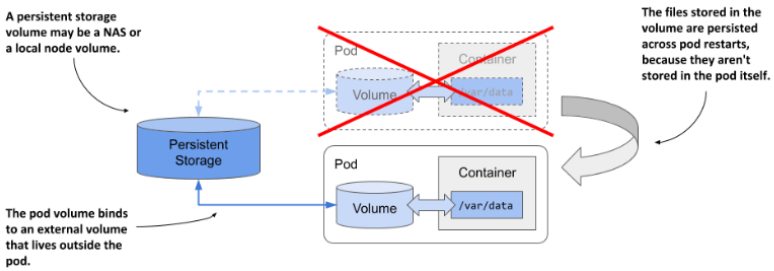


Figure 7.6 Pod volumes can also map to storage volumes that persist across pod restarts

If the pod is deleted and a new pod is created to replace it, the same network-attached storage volume can be attached to the new pod instance so that it can access the data stored there by the previous instance.

185

相同的体积可以安装在每个容器的不同位置，具体取决于
容器本身的需求。如果内容代理将内容写入/var/data，则有意义

在那里安装卷。由于 Web 服务器期望内容位于 /var/html 中，因此

运行它的容器将卷安装在该位置。

在图中，您还会注意到每个容器中的卷安装都可以配置
作为读/写或只读。因为内容代理需要写入卷

虽然 Web 服务器仅从中读取，但两个挂载的配置不同。在里面

出于安全考虑，建议阻止 Web 服务器写入卷，因为

如果 Web 服务器软件存在允许攻击者将任意文件写入文件系统并执行它们的漏洞，则攻击者可能会破坏系统。

在两个容器中使用单个卷的其他示例包括 sidecar 容器运行处理或轮换 Web 服务器日志的工具，或者 init 容器时

为主应用程序容器创建配置文件。

卷与 pod 的生命周期相关联，并且仅在 pod 存在时存在，但根据卷类型，卷中的文件在 pod 退出后仍可保持完整。

卷消失，稍后可以安装到新卷中。

如下图所示，Pod 卷可以映射到 Pod 外部的持久存储。

在这种情况下，代表卷的文件目录不是仅在 pod 持续时间内保留数据的本地文件目录，而是对现有（通常是网络）的挂载。

附加存储卷 (NAS)，其生命周期不与任何 Pod 绑定。因此，存储在卷中的数据是持久的，即使应用程序运行的 pod 已关闭，应用程序也可以使用该数据。

替换为在不同工作节点上运行的新 Pod。

图 7.6 Pod 卷还可以映射到在 Pod 重新启动后持续存在的存储卷

如果删除 Pod 并创建新的 Pod 来替换它，则可以将相同的网络附加存储卷附加到新的 Pod 实例，以便它可以访问存储的数据

之前的实例。

SHARING DATA BETWEEN PODS

Depending on the technology that provides the external storage volume, the same external volume can be attached to multiple pods simultaneously, allowing them to share data. The following figure shows a scenario where three pods each define a volume that is mapped to the same external persistent storage volume.

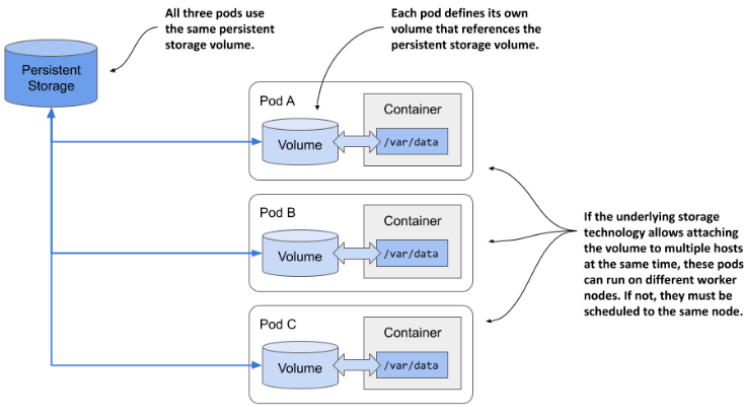


Figure 7.7 Using volumes to share data between pods

In the simplest case, the persistent storage volume could be a simple local directory on the worker node's filesystem, and the three pods have volumes that map to that directory. If all three pods are running on the same node, they can share files through this directory.

If the persistent storage is a network-attached storage volume, the pods may be able to use it even when they are deployed to different nodes. However, this depends on whether the underlying storage technology supports attaching the network volume to more than one computer simultaneously.

While technologies such as Network File System (NFS) allow attaching the volume in read/write mode on multiple computers, other technologies typically available in cloud environments, such as the Google Compute Engine Persistent Disk, allow the volume to be used either in read/write mode on a single cluster node, or in read-only mode on many nodes.

7.1.2 Introducing volume types

When you add a volume to a pod, you must specify the volume type. A wide range of volume types is available. Some are generic, while others are specific to the storage technologies used underneath. Here's a non-exhaustive list of the supported volume types:

在 Pod 之间共享数据

根据提供外部存储卷的技术，相同的外部存储卷

卷可以同时附加到多个 Pod，从而允许它们共享数据。这

下图显示了一个场景，其中三个 Pod 每个定义一个卷，该卷映射到

相同的外部持久存储卷

图7.7 使用卷在Pod之间共享数据

在最简单的情况下，持久存储卷可以是工作节点文件系统上的简单本地目录，并且三个 Pod 具有映射到该目录的卷。我摔倒

三个pod运行在同一个节点上，它们可以通过这个目录共享文件。

如果持久存储是网络附加存储卷，则 Pod 可能能够使用

即使它们部署到不同的节点也是如此。然而，这取决于底层存储技术是否支持将网络卷附加到多个

电脑同时进行。

虽然网络文件系统 (NFS) 等技术允许将卷附加到

多台计算机上的读/写模式，云环境中通常可用的其他技术（例如 Google 计算引擎持久磁盘）允许使用卷

在单个集群节点上处于读/写模式，或者在许多节点上处于只读模式。

7.1.2 介绍卷类型

将卷添加到 Pod 时，必须指定卷类型。有多种卷类型可供选择。有些是通用的，而另一些则特定于所使用的存储技术

下。以下是支持的卷类型的非详尽列表：

- `emptyDir`—A simple directory that allows the pod to store data for the duration of its life cycle. The directory is created just before the pod starts and is initially empty - hence the name. The `gitRepo` volume, which is now deprecated, is similar, but is initialized by cloning a Git repository. Instead of using a `gitRepo` volume, it is recommended to use an `emptyDir` volume and initialize it using an init container.
- `hostPath`—Used for mounting files from the worker node's filesystem into the pod.
- `nfs`—An NFS share mounted into the pod.
- `gcePersistentDisk` (Google Compute Engine Persistent Disk), `awsElasticBlockStore` (Amazon Web Services Elastic Block Store), `azureFile` (Microsoft Azure File Service), `azureDisk` (Microsoft Azure Data Disk)—Used for mounting cloud provider-specific storage.
- `cephfs`, `cinder`, `fc`, `flexVolume`, `flocker`, `glusterfs`, `iscsi`, `portworxVolume`, `quobyte`, `rbd`, `scaleIO`, `storageos`, `photonPersistentDisk`, `vsphereVolume`—Used for mounting other types of network storage.
- `configMap`, `secret`, `downwardAPI`, and the `projected` volume type—Special types of volumes used to expose information about the pod and other Kubernetes objects through files. They are typically used to configure the application running in the pod. You'll learn about them in chapter 9.
- `persistentVolumeClaim`—A portable way to integrate external storage into pods. Instead of pointing directly to an external storage volume, this volume type points to a `PersistentVolumeClaim` object that points to a `PersistentVolume` object that finally references the actual storage. This volume type requires a separate explanation, which you'll find in the next chapter.
- `csi`—A pluggable way of adding storage via the Container Storage Interface. This volume type allows anyone to implement their own storage driver that is then referenced in the `csi` volume definition. During pod setup, the CSI driver is called to attach the volume to the pod.

These volume types serve different purposes. The following sections cover the most representative volume types and help you to gain a general understanding of volumes.

7.2 Using volumes

The simplest volume type is `emptyDir`. This is the volume type you normally use when you need to persist files across container restarts, as explained in section 7.1.1.

7.2.1 Using an emptyDir volume to persist files across container restarts

Remember the fortune pod from the previous chapter? It uses a post-start hook to write a fortune quote to a file that is then served by the Nginx web server. This file is stored in the container's filesystem, which means that it's lost whenever the container is restarted.

•

emptyDir——一个简单的目录，允许 Pod 在其生命周期内存储数据

循环。该目录是在 Pod 启动之前创建的，最初是空的 - 因此

名字。现已弃用的 gitRepo 卷与之类似，但由以下方式初始化

• hostPath——用于将文件从工作节点的文件系统挂载到 Pod 中。
• nfs - 安装到 Pod 中的 NFS 共享。
•

gcePersistentDisk （Google 计算引擎持久磁盘）、awsElasticBlockStore

（Amazon Web Services 弹性块存储）、azureFile（Microsoft Azure 文件服务）、
azureDisk（Microsoft Azure 数据磁盘）— 用于安装特定于云提供商的

•

cephfs、cinder、fc、flexVolume、flocker、glusterfs、iscsi、portworxVolume、

quobyte、rbd、scaleIO、storageos、photonPersistentDisk、vsphereVolume — 用于

•

configMap、secret、downAPI 和投影卷类型 - 特殊类型

卷用于通过以下方式公开有关 pod 和其他 Kubernetes 对象的信息

文件，它们通常用于配置 Pod 中运行的应用程序，你将在第 9 章中了解它们

permanentVolumeClaim——一种将外部存储集成到 Pod 中的便携式方法。

该卷类型不是直接指向外部存储卷，而是指向一个 PersistentVolumeClaim 对象，该对象
又指向一个最终存储在该对象中的 PersistentVolume 对象。

参考实际存储。这种卷类型需要单独解释，其中

csi - 一种通过容器存储接口添加存储的可插入方式。本卷

type 允许任何人实现自己的存储驱动程序，然后在 csi 卷定义中引用该驱动程序。在 Pod
设置期间，会调用 CSI 驱动程序将卷附加到

这些 卷类型服务 不同的目的。这 下列的 部分 覆盖 最多
具有代表性的体积类型并帮助您对体积有一个总体的了解。

7.2 使用卷

最简单的卷类型是emptyDir。这是您需要时通常使用的卷类型

在容器重新启动时保留文件，如第 7.1.1 节中所述。

7.2.1 使用emptyDir卷在容器重新启动后保留文件

还记得上一章的财富荚吗？它使用启动后钩子来编写

Fortune 引用一个文件，然后由 Nginx Web 服务器提供服务。该文件存储在

容器的文件系统，这意味着每当容器重新启动时它都会丢失

You can confirm this by deploying the pod in the `fortune-no-volume.yaml` file, retrieving the quote, and then restarting the container by telling Nginx to stop. The following listing shows the commands you need to run.

Listing 7.1 Testing the behavior of the fortune-no-volume pod

```
$ kubectl apply -f fortune-no-volume.yaml            #A
pod/fortune-no-volume created                        #A

$ kubectl exec fortune-no-volume -- cat /usr/share/nginx/html/quote #B
Quick!! Act as if nothing has happened!              #B

$ kubectl exec fortune-no-volume -- nginx -s stop        #C
[notice] 71#71: signal process started                   #C

$ kubectl exec fortune-no-volume -- cat /usr/share/nginx/html/quote #D
Hindsight is an exact science.                       #D
```

#A Create the pod
#B Fetch the quote
#C Stop Nginx and cause container to restart
#D Fetch the quote again to see that is has changed

When you retrieve the quote after restarting the container, you'll see that it has changed. If you want the pod to serve the same quote no matter how often its container is restarted, you must ensure the file in stored in a volume. An `emptyDir` volume is perfect for this.

INTRODUCING THE EMPTYDIR VOLUME TYPE

The `emptyDir` volume is the simplest type of volume. As the name suggests, it starts as an empty directory. When this type of volume is mounted in a container, files written by the application to the path where the volume is mounted are preserved for the duration of the pod's existence.

This volume type is used in single-container pods when data must be preserved even if the container is restarted. It's also used when the container's filesystem is marked read-only and you want to make only part of it writable. In pods with two or more containers, an `emptyDir` volume is used to exchange data between them.

ADDING AN EMPTYDIR VOLUME TO THE FORTUNE POD

Let's change the definition of the fortune pod so that the post-start hook writes the file to the volume instead of to the ephemeral filesystem of the container. Two changes to the pod manifest are required to achieve this:

1. An `emptyDir` volume must be added to the pod.
2. The volume must be mounted into the container.

However, since the post-start hook is executed every time the container is started, you also need to change the command that the post-start hook executes if you want to prevent it from

188

您可以通过在 Fortune-no-volume.yaml 文件中部署 pod 并检索来确认这一点
引用，然后通过告诉 Nginx 停止来重新启动容器。以下列表显示

您要要运行的命令.

清单 7.1 测试 Fortune-No Volume pod 的行为

```
$ kubectl  apply  -f Fortune-no-volume.yaml          #A
已创建 pod/fortune-no-volume          #A

$ kubectl exec Fortune-no-volume -- cat /usr/share/nginx/html/quote #B
快！！就当什么都没发生过一样！              #B

$ kubectl  exec  Fortune-no-volume  --  nginx  -s  stop      #C
[通知]71#71：信号进程开始                #C

$ kubectl exec Fortune-no-volume -- cat /usr/share/nginx/html/quote #D
事后诸葛亮是一门精确的科学。          #D
```

#A 创建 Pod #B 获取报价

#C 停止 Nginx 并导致容器重新启动
#D 再次获取引用以查看其已更改

当您在重新启动容器后检索报价时，您会看到它已更改。如果您希望 pod 提供相同的报价，无论其容器重新启动的频率如何，您必须

确保文件存储在卷中。 emptyDir 卷非常适合此目的。

介绍 EMPTYDIR 卷类型

emptyDir 卷是最简单的卷类型。顾名思义，它以空目录开始。当这种类型的卷安装在容器中时，由该卷写入的文件

在 Pod 存在期间，将保留对挂载卷的路径的应用程序。

当即使容器重新启动也必须保留数据时，在单容器 Pod 中使用此卷类型。当容器的文件系统标记为只读并且

你只想让它的一部分可写。在具有两个或多个容器的 Pod 中，使用 emptyDir 卷在它们之间交换数据。

向 Fortune POD 添加空目录卷

让我们更改 Fortune pod 的定义，以便 post-start 挂钩将文件写入

卷而不是容器的临时文件系统。要实现此目的，需要对 pod 清单进行两处更改：

1. 必须将一个emptyDir卷添加到pod中。
2. 卷必须安装到容器中。

但是，由于每次启动容器时都会执行 post-start hook，因此如果您想阻止它，还需要更改 post-start hook 执行的命令

overwriting the file if it already exists. The operation of the new command is explained in the sidebar.

Understanding the new command in the fortune post-start hook

The post-start hook executes the following shell command:

ls /usr/share/nginx/html/quote || (apk add fortune &&
[CA] fortune > /usr/share/nginx/html/quote)

If you're not familiar with the Linux shell, it may not be obvious what this command does. Let me explain.

First, the `ls` command, which is typically used to list directory contents, is used in an untypical way to check whether the `quote` file exists. What you may not know is that the `ls` command succeeds (returns exit code zero) if the specified file exists, and fails (returns a non-zero exit code) if it doesn't.

Since in our case the entire command is an *or* expression, as denoted by the double pipe character, the right side of the expression is never evaluated if the left side of the expression evaluates to true (which is when the `ls` command succeeds). When this happens, the `apk` and `fortune` commands are never executed. Therefore, if the `quote` file already exists, it's not overwritten.

If the file doesn't exist, the `ls` command fails, causing the left side of the expression to be false, which means that the right side must be evaluated to resolve the expression. The `apk` and `fortune` commands are therefore executed and they generate the `quote` file.

This new command ensures that the quote file is only written once - at the first start of the container, not at subsequent restarts

The following listing shows the new pod manifest that incorporates all the changes.

Listing 7.2 A pod with an emptyDir volume: fortune-emptydir.yaml

```
kind: Pod
metadata:
  name: fortune-emptydir
spec:
  volumes:                          #A
  - name: content                     #A
    emptyDir: {}                      #A
  containers:
  - name: nginx
    image: nginx:alpine
    volumeMounts:                     #B
    - name: content                   #B
      mountPath: /usr/share/nginx/html      #B
    lifecycle:
      postStart:
        exec:
          command:
          - sh
          - -c
          - "ls /usr/share/nginx/html/quote || (apk add fortune &&  #C
[CA] fortune > /usr/share/nginx/html/quote)"                        #C
    ports:
    - name: http
      containerPort: 80
```


如果文件已存在，则覆盖该文件。新命令的操作在

侧边栏。

了解 Fortune 启动后挂钩中的新命令
post-start 挂钩执行以下 shell 命令： ls /usr/share/nginx/html/quote || (apk添加财富&&

[CA] 财富 > /usr/share/nginx/html/quote)

如果您不熟悉 Linux shell，可能不太清楚该命令的作用。让我解释。首先，通常用于列出目录内容的 ls 命令以一种非典型的方式使用来检查是否

报价文件存在。您可能不知道的是，如果指定的文件存在，则 ls 命令会成功（返回退出代码零），如果不存在，则会失败（返回非零退出代码）。

由于在我们的例子中，整个命令是一个 or 表达式，如双管道字符所示，因此如果表达式左侧的计算结果为 true（即 ls 命令成功时），则永远不会计算表达式的右侧。发生这种情况时，apk 和 Fortune 命令永远不会执行。因此，如果报价文件已经存在，则不会被覆盖。

如果文件不存在，则 ls 命令失败，导致表达式左侧为 false，这意味着必须计算右侧才能解析表达式。因此执行 apk 和 Fortune 命令并

他们生成报价文件。

这个新命令确保引用文件仅写入一次 - 在容器第一次启动时，而不是在后续重新启动时

以下清单显示了包含所有更改的新 pod 清单。

清单 7.2 具有emptyDir卷的pod：fortune-emptydir.yaml

种类：Pod 元数
据：

名称：fortune-
卷：
- 名称：内容 # A
空目录：{} # A
容器：-名
称：
图片：
nginx:alpine 卷 #B
- 名称：内容 #B
挂载路径：/usr/share/nginx/html
生命周
期：
postSta
命令：-sh--c

- "ls /usr/share/nginx/html/quote || (apk 添
加财富 && #C
[CA] 财富 > /usr/share/nginx/html/quote)" # C
端口：

- 名称：http

#A The volume is defined here
#B The volume is mounted into the container here
#C The new post-start command

The listing shows that an `emptyDir` volume named `content` is defined in the `spec.volumes` array of the pod manifest.

CONFIGURING THE VOLUME

In general, each volume definition must include a `name` and a type, which is indicated by the name of the nested field (for example: `emptyDir`, `gcePersistentDisk`, `nfs`, and so on). This field is typically an object with additional sub-fields that allow you to configure the volume and are specific to each volume type.

For example, the `emptyDir` volume type supports two fields for configuring the volume. They are explained in the following table.

Field	Description
medium	The type of storage medium to use for the directory. If left empty, the default medium of the host node is used (the directory is created on one of the node's disks). The only other supported option is `Memory`, which causes the volume to use `tmpfs`, a virtual memory filesystem where the files are kept in memory instead of on the hard disk.
sizeLimit	The total amount of local storage required for the directory, whether on disk or in memory. For example, to set the maximum size to ten mebibytes, you set this field to `10Mi`.

Table 7.1 Configuration options for an emptyDir volume

NOTE The `emptyDir` field in the volume definition defines neither of these properties. The curly braces `{}` have been added to indicate this explicitly, but they can be omitted.

MOUNTING THE VOLUME IN THE CONTAINER

Defining a volume in the pod is only half of what you need to do to make it available in a container. The volume must also be mounted in the container. This is done by referencing the volume by name in the `volumeMounts` array within the container definition.

In addition to the `name`, a volume mount definition must also include the `mountPath` - the path within the container where the volume should be mounted. In the example, the volume is mounted at `/usr/share/nginx/html` because that's where the post-start hook writes the `quote` file.

Since the `quote` file is now written to the volume, which exists for the duration of the pod, the pod should always serve the same quote, even when the container is restarted.

#A 体积在这里定义

#B 卷已安装到此处的容器中 #C 新的启

该清单显示在spec.volumes 中定义了一个名为content 的emptyDir 卷

Pod 清单的数组。

配置音量

一般来说，每个卷定义必须包含名称和类型，由嵌套字段的名称来指示（例如：emptyDir、gcePersistentDisk、nfs 等）。这

字段通常是具有附加子字段的对象，这些子字段允许您配置卷并且特定于每种卷类型。

例如，emptyDir 卷类型支持两个用于配置卷的字段。他们
解释如下表。

场地	描述
中等的	用于目录的存储介质的类型。如果留空，则默认媒体 使用主机节点（目录在节点的磁盘之一上创建）。唯一受支持的其他 选项是内存，这会导致卷使用 tmpfs（一种虚拟内存
尺寸限制	目录所需的本地存储总量，无论是在磁盘上还是在内存中。例如，要将最大大小设置为 10 MB，请将此字段设置为 10Mi。

表 7.1 emptyDir 卷的配置选项

注意卷定义中的emptyDir 字段未定义这些属性。大括号 {}
已添加以明确表明这一点，但可以省略它们。

将体积安装到容器中

在 Pod 中定义卷只是使其在容器中可用所需要做的一半。该卷还必须安装在容器中。这是通过引用来完成的

容器定义中的volumeMounts数组中的卷名称。
除了名称之外，卷安装定义还必须包括安装路径 -
容器内应安装卷的路径。在示例中，卷安装在 /usr/share/nginx/html 中，因为这是启动后挂钩写入引用的位置

文件。
由于报价文件现已写入卷，该卷在 Pod 运行期间一直存在，
即使容器重新启动，pod 也应始终提供相同的报价。

OBSERVING THE EMPTYDIR VOLUME IN ACTION

If you deploy the pod in the listing, you'll notice that the quote will remain the same throughout the life of the pod, regardless of how often the container is restarted, because the `/usr/share/nginx/html` directory where the file is stored is mounted from somewhere else. You can tell this by listing the mount points in the container. Run the `mount` command to see that the directory is mounted into the container from elsewhere:

```
$ kubectl exec fortune-emptydir -- mount --list | grep nginx/html
/dev/mapper/fedora_t580-home on /usr/share/nginx/html type ext4 (rw)
```

7.2.2 Using an emptyDir volume to share files between containers

An `emptyDir` volume is also useful for sharing files between containers of the same pod (it can't be used to share files between containers of different pods). Let's see how this is done.

The fortune pod currently serves the same quote throughout the lifetime of the pod. This isn't so interesting. Let's build a new version of the pod, where the quote changes every 30 seconds.

You'll retain Nginx as the web server, but will replace the post-start hook with a container that periodically runs the `fortune` command to update the quote stored in the file. The container is available at `docker.io/luksa/fortune:1.0`, but you can also build it yourself by following the instructions in the sidebar.

Building the fortune container image

You need two files to build the image. First is the `docker_entrypoint.sh` script that has the following contents:

```
#!/bin/sh
trap "exit" INT

INTERVAL=${INTERVAL:-30}
OUTPUT_FILE=${1:-/var/local/output/quote}

echo "Fortune Writer 1.0"
echo "Configured to write fortune to $OUTPUT_FILE every $INTERVAL seconds"

while :
do
  echo "$(date) Writing fortune..."
  fortune > $OUTPUT_FILE
  sleep $INTERVAL
done
```

This is the script that is executed when you run this container. To build the container image, you'll also need a `Dockerfile` with the following contents:

```
FROM alpine
RUN apk add fortune
```


191

观察运行中的 EMPTYDIR 卷

如果您在列表中部署 Pod，您会注意到报价将始终保持不变

pod 的生命周期，无论容器重新启动的频率如何，因为存储文件的 /usr/share/nginx/html 目录是从其他地方挂载的。

您可以通过列出容器中的安装点来判断这一点。运行mount命令可以看到

该目录从其他地方安装到容器中：

```
$ kubectl exec Fortune-emptydir --mount --list | kubectl exec Fortune-emptydir --
mount --list | grep nginx/html /dev/mapper/fedora_t580-home 位于
```

7.2.2 使用emptyDir卷在容器之间共享文件

emptyDir 卷对于在同一 pod 的容器之间共享文件也很有用（它不能用于在不同 pod 的容器之间共享文件）。让我们看看这是如何完成的。

目前，财富英在其整个生命周期内提供相同的报价。这不太有趣。让我们构建一个新版本的 Pod，其中报价每 30 更改一次

秒。您将保留 Nginx 作为 Web 服务器，但将用容器替换启动后钩子
定期运行 Fortune 命令来更新文件中存储的报价。该容器可在 docker.io/luksa/fortune:1.0 上找到，但您也可以通过以下方式自己构建它

侧栏中的说明。

构建 Fortune 容器镜像
您需要两个文件来构建映像。首先是 docker_entrypoint.sh 脚本，其内容如下：#!/bin/sh

陷阱“退出”INT

间隔=${间隔:-30}

OUTPUT_FILE=${1:-/var/local/o

echo "财富作家1.0"

echo "配置为每 $INTERVAL 秒将财富写入 $OUTPUT_FILE"

尽管：

做

echo "$(date) 写算命..."

财富 > $OUTPUT_FILE

睡眠 $INTERVAL

完毕

这是运行此容器时执行的脚本。要构建容器映像，您还需要一个包含以下内容的 Dockerfile：FROM alpine

RUN apk 添加财富

```
COPY docker_entrypoint.sh /docker_entrypoint.sh
VOLUME /var/local/output
ENTRYPOINT ["/docker_entrypoint.sh"]
```

You can build the image and push it to Docker Hub by running the following two commands (replace `luksa` with your own Docker Hub user ID):

```
$ docker build -t luksa/fortune-writer:1.0 .
$ docker push luksa/fortune-writer:1.0
```

If you've downloaded the code from the book's code repository at GitHub, you'll find both files in the `Chapter07/fortune-writer-image` directory. To build and push the image you can also run the following command in the `Chapter07/` directory:

```
$ PREFIX=luksa/ PUSH=true ./build-fortune-writer-image.sh
```

Again, replace `luksa/` with your Docker Hub user ID.

CREATING THE POD

Now that you have both images required to run the pod, create the pod manifest. Its contents are shown in the following listing.

Listing 7.3 Two containers sharing the same volume: fortune.yaml

```
apiVersion: v1
kind: Pod
metadata:
  name: fortune
spec:
  volumes:                          #A
  - name: content                     #A
    emptyDir: {}                      #A
  containers:
  - name: fortune                     #B
    image: luksa/fortune-writer:1.0         #B
    volumeMounts:                     #B
    - name: content                   #B
      mountPath: /var/local/output    #C    #B
  - name: nginx
    image: nginx:alpine
    volumeMounts:
    - name: content                   #D
      mountPath: /usr/share/nginx/html      #D
      readOnly: true                  #D
    ports:
    - name: http
      containerPort: 80
```

#A The emptyDir volume shared by the two containers
#B The container that writes the fortune to the volume
#C In this container, the volume is mounted at this location
#D In the nginx container, the volume is mounted as read-only in a different location

The pod consists of two containers and a single volume, which is mounted in both containers, but at different locations within each container. The reason for this is that the `fortune` container

复制 docker_entrypoint.sh /docker_entrypoint.sh

音量 /var/local/输出

入口点["/docker_entrypoint.sh"]

您可以通过运行以下两个命令来构建映像并将其推送到 Docker Hub（将 luksa 替换为您自己的 Docker Hub 用户 ID）：

```
$ docker build -t luksa/fortune-writer:1.0 。
$ docker Push Luksa/fortune-writer:1.0
```

如果您从本书的 GitHub 代码存储库下载了代码，您将在 Chapter07/fortune-writer-image 目录中找到这两个文件。要构建并推送映像，您还可以在 Chapter07/ 目录中运行以下命令：

```
$ PREFIX=luksa/ PUSH=true ./build-fortune-writer-image.sh
```

再次将 luksa/ 替换为您的 Docker Hub 用户 ID。

创建 Pod

现在您已拥有运行 pod 所需的两个映像，请创建 pod 清单。其内容如下列表所示。

清单 7.3 共享相同卷的两个容器：fortune.yaml

```
api版本：
v1 种类：
Pod 元数
据：
卷：                              # A
 - 名称：内容                       # A
  空目录：{}                       # A
容器：
 - 名称：财富                       #B
  图片：luksa/fortune-writer:1.0  #B
  体积安装：                        #B
  - 名称：内容                      #B
    挂载路径：/var/local/output  #C #B
-名称：nginx
 图片：
  nginx:alpine
  volumeMounts                    #D
    挂载路径：/usr/share/nginx/html #D
    只读：真                        #D
 端口：
  - 名称：http
```

#A 两个容器共享的emptyDir卷 #B 向卷写入财富的容器 #C 在这个容器中，卷挂载在这个位置

#D 在 nginx 容器中，卷以只读方式安装在不同的位置

Pod 由两个容器和一个安装在两个容器中的卷组成，但在每个容器内的不同位置

writes the `quote` file into the `/var/local/output` directory, whereas the `nginx` container serves files from the `/usr/share/nginx/html` directory.

RUNNING THE POD

When you create the pod from the manifest, the two containers start and continue running until the pod is deleted. The `fortune` container writes a new quote to the file every 30 seconds, and the `nginx` container serves this file. After you create the pod, use the `kubectl port-forward` command to open a communication tunnel to the pod and test if the server responds with a different quote every 30 seconds.

Alternatively, you can also display the contents of the file using either of the following two commands:

```
$ kubectl exec fortune -c fortune -- cat /var/local/output/quote
$ kubectl exec fortune -c nginx -- cat /usr/share/nginx/html/quote
```

As you can see, one of them prints the contents of the file from within the `fortune` container, whereas the other command prints the contents from within the `nginx` container. Because the two paths point to the same `quote` file on the shared volume, the output of the commands is identical.

SPECIFYING THE STORAGE MEDIUM TO USE IN THE EMPTYDIR VOLUME

The `emptyDir` volume in the previous example created a directory on the actual drive of the worker node hosting your pod, so its performance depends on the type of drive installed on the node. If you need to perform the I/O operations on the volume as quickly as possible, you can instruct Kubernetes to create the volume using the *tmpfs* filesystem, which keeps its files in memory. To do this, set the `emptyDir`'s `medium` field to `Memory` as in the following snippet:

```
volumes:
  - name: content
    emptyDir:
      medium: Memory   #A
```

#A This directory should be stored in memory.

Creating the `emptyDir` volume in memory is also a good idea whenever it's used to store sensitive data. Because the data is not written to disk, there is less chance that the data will be compromised and persisted longer than desired. As you'll learn in chapter 9, Kubernetes uses the same in-memory approach when data stored in the *Secret* API object type needs to be exposed to the application in the container.

SPECIFYING THE SIZE LIMIT FOR THE EMPTYDIR VOLUME

The size of an `emptyDir` volume can be limited by setting the `sizeLimit` field. Setting this field is especially important for in-memory volumes when the overall memory usage of the pod is limited by so-called *resource limits*. You'll learn about this in chapter 20.

193

将引用文件写入 /var/local/output 目录，而 nginx 容器提供服务

/usr/share/nginx/html 目录中的文件。

运行 Pod

当您从清单创建 Pod 时，两个容器将启动并继续运行，直到

该 Pod 已被删除。 Fortune 容器每 30 秒向该文件写入一个新的报价，并且 nginx 容器为该文件提供服务。创建 pod 后，使用 kubectl port-forward

命令打开到 pod 的通信隧道并测试服务器是否每 30 秒响应一次不同的引用。

或者，您也可以使用以下两种方法之一显示文件的内容
命令：

$ kubectl exec Fortune -c Fortune -- cat
/var/local/output/quote $ kubectl exec Fortune -c nginx --

正如您所看到的，其中一个命令从 Fortune 容器中打印文件的内容，而另一个命令从 nginx 容器中打印文件的内容。因为

两个路径指向共享卷上的同一个引用文件，命令的输出是相同的。

指定 EMPTYDIR 卷中使用的存储介质

这 空目录 上一个示例中的卷在工作器的实际驱动器上创建了一个目录
托管 Pod 的节点，因此其性能取决于节点上安装的驱动器类型。如果您需要尽快在卷上执行 I/O 操作，您可以指示

Kubernetes 使用 tmpfs 文件系统创建卷，该系统将其文件保存在内存中。为此，请设置
空目录 的中字段到内存，如以下代码片段所示：

卷：

- 名称：内
  介质：内存 #A

#A 该目录应该存储在内存中。

每当用于存储时，在内存中创建emptyDir卷也是一个好主意

敏感数据。由于数据不会写入磁盘，因此数据受到损害和保留时间超过预期的可能性较小。正如您将在第 9 章中学到的，Kubernetes 使用

当存储在 Secret API 对象类型中的数据需要向容器中的应用程序公开时，采用相同的内存方法。

指定 EMPTYDIR 卷的大小限制

可以通过设置 sizeLimit 字段来限制 emptyDir 卷的大小。设置该字段

当 Pod 的总体内存使用量受到所谓的资源限制时，对于内存卷尤其重要。您将在第 20 章中了解这一点。

7.2.3 Specifying how a volume is to be mounted in the container

In the previous section, the focus was mainly on the volume type. The volume was mounted in the container with the minimum configuration required, as shown in the following listing.

Listing 7.4 The minimal configuration of a volumeMount

```
containers:
- name: my-container
  ...
  volumeMounts:
  - name: my-volume              #A
    mountPath: /path/in/container    #B
```

#A The name of the volume to mount into this container
#B Where in the container's filesystem to mount the volume

As you can see in the listing, the absolute minimum is to specify the name of the volume and the path where it should be mounted. But other configuration options for mounting volumes exist. The full list of volume mount options is shown in the following table.

Field	Description
name	The name of the volume to mount. This must match one of the volumes defined in the pod.
mountPath	The path within the container at which to mount the volume.
readOnly	Whether to mount the volume as read-only. Defaults to false.
mountPropagation	Specifies what should happen if additional filesystem volumes are mounted inside the volume. Defaults to None, which means that the container won't receive any mounts that are mounted by the host, and the host won't receive any mounts that are mounted by the container. HostToContainer means that the container will receive all mounts that are mounted into this volume by the host, but not the other way around. Bidirectional means that the container will receive mounts added by the host, and the host will receive mounts added by the container.
subPath	Defaults to "" which means that the entire volume is mounted into the container. When set to a non-empty string, only the specified subPath within the volume is mounted into the container.
subPathExpr	Just like subPath but can have environment variable references using the syntax $(ENV_VAR_NAME). Only environment variables that are explicitly defined in the container definition are applicable. Implicit variables such as HOSTNAME will not be resolved. You'll learn how to specify environment variables in chapter 9.

7.2.3 指定卷如何挂载到容器中

上一节主要关注的是卷类型。该卷安装在

具有所需最低配置的容器，如以下清单所示。

清单 7.4 VolumeMount 的最小配置

```
容器：

- 名称：我的容
  "VolumeMou
  nts：-名称：                # A
    挂载路径：/path/in/contai#Ber
    我的卷
```

#A 要挂载到此容器中的卷的名称 #B 在容器的
文件系统中挂载该卷的位置

正如您在清单中看到的，绝对最少的是指定卷的名称
以及它应该安装的路径。但用于安装卷的其他配置选项

存在。卷安装选项的完整列表如下表所示。

场地	描述
姓名	
挂载路径	容器内用于安装卷的路径。
只读	是否将卷安装为只读。默认为 false。
挂载传播	指定如果内部安装了额外的文件系统卷会发生什么 音量。 默认为 None，这意味着容器不会接收任何安装 由主机挂载，并且主机不会收到由主机挂载的任何挂载 容器。 HostToContainer 表示容器将接收所有已挂载的挂载 由主机放入此卷。但反之则不然。
子路径	默认为 ""，这意味着整个卷都安装到容器中。
子路径表达式	就像 subPath 一样，但可以使用语法 $(ENV_VAR_NAME) 来引用环境变量。仅在显式定义的环境变量 容器定义适用。诸如 HOSTNAME 之类的隐式变量将不会被 解决。您将在第 9 章中学习如何指定环境变量。

Table 7.2 Configuration options for a volume mount

In most cases, you only specify the `name`, `mountPath` and whether the mount should be `readOnly`. The `mountPropagation` option comes into play for advanced use-cases where additional mounts are added to the volume's file tree later, either from the host or from the container. The `subPath` and `subPathExpr` options are useful when you want to use a single volume with multiple directories that you want to mount to different containers instead of using multiple volumes.

The `subPathExpr` option is also used when a volume is shared by multiple pod replicas. In chapter 9, you'll learn how to use the Downward API to inject the name of the pod into an environment variable. By referencing this variable in `subPathExpr`, you can configure each replica to use its own subdirectory based on its name.

7.3 Integrating external storage into pods

An `emptyDir` volume is basically a dedicated directory created specifically for and used exclusively by the pod that contains the volume. When the pod is deleted, the volume and its contents are deleted. However, other types of volumes don't create a new directory, but instead mount an existing external directory in the filesystem of the container. The contents of this volume can survive multiple instantiations of the same pod and can even be shared by multiple pods. These are the types of volumes we'll explore next.

To learn how external storage is used in a pod, you'll create a pod that runs the document-oriented database MongoDB. To ensure that the data stored in the database is persisted, you'll add a volume to the pod and mount it in the container at the location where MongoDB writes its data files.

The tricky part of this exercise is that the type of persistent volumes available in your cluster depends on the environment in which the cluster is running. At the beginning of this book, you learned that Kubernetes can reschedule a pod to another node at will. To ensure that the MongoDB pod can still access its data, it should use some kind of network-attached storage instead of the worker node's local drive.

Ideally, you should use a proper Kubernetes cluster, such as GKE, for the following exercises. Unfortunately, clusters provisioned with Minikube or kind don't provide any kind of network storage volume out of the box. So, if you're using either of these tools, you'll need to resort to using node-local storage provided by the so-called `hostPath` volume type, even though this volume type is explained later, in section 7.4.

7.3.1 Using a Google Compute Engine Persistent Disk in a pod volume

If you use Google Kubernetes Engine to run the exercises in this book, your cluster nodes run on Google Compute Engine (GCE). In GCE, persistent storage is provided via GCE Persistent Disks. Kubernetes supports adding the to your pods via the `gcePersistentDisk` volume type.

表 7.2 卷安装的配置选项

在大多数情况下，您只需指定名称、安装路径以及安装是否应为只读。 mountPropagation 选项适用于高级用例，其中

稍后从主机或容器将额外的安装添加到卷的文件树中。当您想使用单个路径时， subPath 和 subPathExpr 选项非常有用

具有多个目录的卷，您希望将其挂载到不同的容器而不是使用

多卷。

当卷由多个 Pod 副本共享时，也会使用 subPathExpr 选项。在第 9 章中，您将学习如何使用 Downward API 将 pod 的名称注入到

环境变量。通过在 subPathExpr 中引用此变量，您可以将每个副本配置为根据其名称使用自己的子目录。

7.3 将外部存储集成到 Pod 中

一个emptyDir卷基本上是一个专门为包含该卷的pod创建并专门使用的专用目录。当 pod 被删除时，该卷及其

内容被删除。但是，其他类型的卷不会创建新目录，而是在容器的文件系统中挂载现有的外部目录。本篇的内容

Volume 可以在同一个 Pod 的多个实例化中幸存下来，甚至可以由多个 Pod 共享。这些是我们接下来要探讨的卷类型。

要了解如何在 Pod 中使用外部存储，您将创建一个运行面向文档的数据库 MongoDB 的 Pod。为了确保存储在数据库中的数据被持久化，您将

向 pod 添加一个卷，并将其挂载到容器中 MongoDB 写入数据文件的位置。

此练习的棘手部分是集群中可用的持久卷的类型取决于集群运行的环境。在本书的开头，你

了解到 Kubernetes 可以随意将 pod 重新调度到另一个节点。为了确保 MongoDB Pod 仍然可以访问其数据，它应该使用某种网络附加存储

而不是工作节点的本地驱动器。

理想情况下，您应该使用适当的 Kubernetes 集群（例如 GKE）来进行以下练习。不幸的是，使用 Minikube 或其他类型配置的集群不提供任何类型的开箱即用的网络存储卷。因此，如果您正在使用这些工具中的任何一个，则需要求助于

使用由所谓的 hostPath 卷类型提供的节点本地存储，尽管此卷类型将在稍后的 7.4 节中进行解释。

7.3.1 在 Pod 卷中使用 Google Compute Engine 永久磁盘

如果您使用 Google Kubernetes Engine 运行本书中的练习，您的集群节点将运行

在谷歌计算引擎（GCE）上。在 GCE 中，持久存储是通过 GCE 持久磁盘提供的。 Kubernetes 支持通过 gcePersistentDisk 卷类型将其添加到 pod。

NOTE To adapt this exercise for use with other cloud providers, use the appropriate volume type supported by the cloud provider. Consult the documentation provided by the cloud provider to determine how to create the storage volume and how to mount it into the pod.

CREATING A GCE PERSISTENT DISK

Before you can use the GCE persistent disk volume in your pod, you must create the disk itself. It must reside in the same zone as your Kubernetes cluster. If you don't remember in which zone you created the cluster, you can see it by listing your Kubernetes clusters using the `gcloud` command as follows:

```
$ gcloud container clusters list
NAME  ZONE         MASTER_VERSION MASTER_IP     ...
kubia europe-west3-c 1.14.10-gke.42 104.155.84.137 ...
```

In my case, this shows that the cluster is located in the zone `europe-west3-c`, so I have to create the GCE persistent disk in the same zone as well. The zone must be specified when creating the disk as follows:

```
$ gcloud compute disks create --size=1GiB --zone=europe-west3-c mongodb
WARNING: You have selected a disk size of under [200GB]. This may result in poor I/O performance. For more information,
         see: https://developers.google.com/compute/docs/disks#pdperformance.
Created [https://www.googleapis.com/compute/v1/projects/rapid-pivot-136513/zones/europe-west3-c/disks/mongodb].
NAME    ZONE         SIZE_GB TYPE       STATUS
mongodb europe-west3-c 1       pd-standard READY
```

This command creates a GCE persistent disk called `mongodb` with 1GiB of space. You can freely ignore the disk size warning, because it doesn't affect the exercises you're about to run. You may also see an additional warning that the disk is not yet formatted. You can ignore that, too, because formatting is done automatically when you use the disk in your pod.

CREATING A POD WITH A GCEPERSISTENTDISK VOLUME

Now that you have set up your physical storage, you can use it in a volume inside your MongoDB pod. You'll create the pod from the YAML shown in the following listing.

Listing 7.5 A pod using a gcePersistentDisk volume: mongodb-pod-gcepd.yaml

```
apiVersion: v1
kind: Pod
metadata:
  name: mongodb
spec:
  volumes:
  - name: mongodb-data        #A
    gcePersistentDisk:        #B
      pdName: mongodb         #C
      fsType: ext4            #D
  containers:
  - image: mongo
    name: mongodb
```


注意 要调整此练习以与其他云提供商一起使用，请使用以下支持的适当卷类型：
云提供商。请参阅云提供商提供的文档以确定如何创建
存储卷以及如何将其安装到 Pod 中

创建GCE持久磁盘

您必须先创建磁盘本身，然后才能在 Pod 中使用 GCE 永久性磁盘卷。

它必须与您的 Kubernetes 集群位于同一区域。如果您不记得在哪个区域创建了集群，可以通过使用 gcloud 列出 Kubernetes 集群来查看它

命令如下：

```
$ gcloud 容器集群列表 NAME ZONE
                MASTER_VERSION MASTER_IP
库比亚欧洲-west3-c 1.14.10-gke.42 104.155.84.137 ...
```

就我而言，这表明该集群位于 europe-west3-c 区域，因此我必须

也在同一区域中创建 GCE 永久磁盘。当以下情况时必须指定区域：

创建磁盘如下：

```
$ gcloud Compute disks create --size=1GiB --zone=europe-west3-c
mongodb 警告：您选择的磁盘大小小于 [200GB]。这可能会导致 I/O 性能不
佳。欲了解更多信息，请输入姓名
        区         SIZE_GB 类型 地位
mongodb 欧洲-west3-c PD 标准就绪
```

此命令创建一个名为 mongodb 且拥有 1GiB 空间的 GCE 永久磁盘。你可以自由地

忽略磁盘大小警告，因为它不会影响您即将运行的练习。您可能还会看到磁盘尚未格式化的附加警告。你也可以忽略它

因为当您在 Pod 中使用磁盘时，格式化会自动完成。
创建具有 GCEPERSISTENTDISK 卷的 Pod

现在您已经设置了物理存储，您可以在 MongoDB 内的卷中使用它

荣，您将根据以下清单中所示的 YAML 创建 pod。

清单 7.5 使用 gcePersistentDisk 卷的 pod：mongodb-pod-gcepd.yaml

```
api版本：v1
种类：Pod
元数据：

名称：
mongodb
卷：
  - 名称：mongodb-数据A
    gce持久磁盘：      #B
    pd名称：mongodb#C
    文件系统类型：ext4#D
容器：

  - 图片：
```

```
  volumeMounts:
  - name: mongodb-data      #A
    mountPath: /data/db     #E
  ports:
  - containerPort: 27017
    protocol: TCP
```

#A The name of the volume (also referenced in the volumeMounts section below)
#B The volume type is GCE Persistent Disk.
#C The name of the persistent disk must match the actual disk you created earlier.
#D The filesystem type is ext4.
#E The path where MongoDB stores its data

> **NOTE** If you're using Minikube or kind, you can't use a GCE Persistent Disk, but you can deploy `mongodb-pod-hostpath.yaml`, which uses a `hostPath` volume instead of a GCE PD. This uses node-local instead of network storage, so you must ensure that the pod is always deployed to the same node. This is always true in Minikube because it creates a single node cluster. If you're using kind, create the pod from the file `mongodb-pod-hostpath-kind.yaml`, which ensures that the pod is always deployed on the same node.

The pod is visualized in the following figure. It consists of a single container and a single volume backed by the GCE Persistent Disk you created earlier. Within the container, the volume is mounted at `/data/db`, since this is the place where MongoDB stores its data. This ensures that the data will be written to the persistent disk.

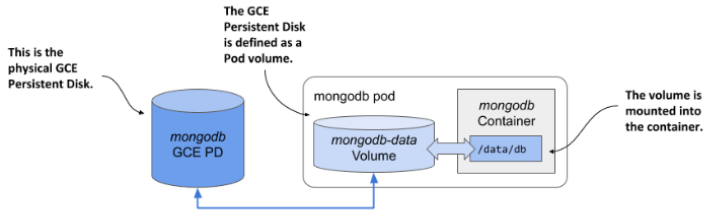


Figure 7.8 A GCE Persistent Disk used as a pod Volume and mounted into one of its containers

WRITING TO THE VOLUME

After you create the pod, run the MongoDB client inside the container and add some data to the database. To do this, run the client as shown in the following listing.

Listing 7.6 Entering the MongoDB shell inside the mongodb pod

```
$ kubectl exec -it mongodb -- mongo
MongoDB shell version v4.4.0
connecting to: mongodb://127.0.0.1:27017/...
Implicit session: session { "id" : UUID("42671520-0cf7-...") }
MongoDB server version: 4.4.0
...
```


197

```
体积安装：
- 名称：mongodb-数据A
 挂载路径：/data/db #E
端口：

- 容器端口：
```

#A 卷的名称（也在下面的volumeMounts 部分中引用） #B 卷类型是GCE 持久磁盘。

#C 永久磁盘的名称必须与您之前创建的实际磁盘匹配。 #D 文件系统类型是 ext4。

> 笔记 如果您使用 Minikube 或 kind，则不能使用 GCE 持久磁盘，但可以部署 mongodb-pod-hostpath.yaml，它使用 hostPath 卷而不是 GCE PD。这使用节点本地而不是网络存储，因此必须保证Pod始终部署到同一个节点。这始终是正确的 Minikube 因为它创建单节点集群。如果您使用 kind，请从文件 mongodb- 创建 pod

该 Pod 如下图所示。它由一个容器和一个由您之前创建的 GCE 持久磁盘支持的卷组成。容器内的体积为

安装在/data/db，因为这是 MongoDB 存储数据的地方。这可确保数据将写入永久磁盘。

图 7.8 用作 Pod 卷并安装到其容器之一的 GCE 持久磁盘

写入卷

创建 Pod 后，在容器内运行 MongoDB 客户端并向数据库添加一些数据。为此，请运行客户端，如以下清单所示。

清单 7.6 在 mongodb pod 内进入 MongoDB shell

```
$ kubectl exec -it mongodb -- mongo
MongoDB shell 版本 v4.4.0
```

连接到： mongodb://127.0.0.1:27017/...

隐式会话： session { "id" : UUID("42671520-0cf7-...") } MongoDB 服

```
>
```

To insert a document into the database, enter the following commands:

```
> use mystore
switched to db mystore
> db.foo.insert({name:'foo'})
WriteResult({ "nInserted" : 1 })
```

This inserts a document with a single property called `name`. You can also specify additional properties in the document if you wish, or add additional documents. Now, use the `find()` command to retrieve the document you inserted:

```
> db.foo.find()
{ "_id" : ObjectId("57a61eb9de0cfd512374cc75"), "name" : "foo" }
```

This document should now be stored in MongoDB's data files, which are located in the `/data/db` directory. Since this is where you mounted the GCE Persistent Disk, the document should be stored permanently.

RE-CREATING THE POD AND VERIFYING THAT IT CAN READ THE DATA PERSISTED BY THE PREVIOUS POD

You can now exit the `mongodb` client (type `exit` and press Enter), after which you can delete the pod and recreate it:

```
$ kubectl delete pod mongodb
pod "mongodb" deleted
$ kubectl create -f mongodb-pod-gcepd.yaml
pod "mongodb" created
```

Since the new pod is an exact replica of the previous, it points to the same GCE persistent disk as the previous pod, so the MongoDB container running inside it should see the exact same files, even if the new pod is scheduled to another node.

TIP You can see what node a pod is scheduled to by running `kubectl get po -o wide`.

Once the container starts up, you can run the MongoDB client again and check if the previously stored document(s) can still be retrieved, as shown in the following listing.

Listing 7.7 Retrieving MongoDB's persisted data in a new pod

```
$ kubectl exec -it mongodb mongo
...
> use mystore
switched to db mystore
> db.foo.find()
{ "_id" : ObjectId("57a61eb9de0cfd512374cc75"), "name" : "foo" }
```

As expected, the data still exists even though you deleted and recreated the pod. This confirms that you can use a GCE persistent disk to persist data across multiple instantiations of the same


```
>
```

要将文档插入数据库，请输入以下命令：

> 使用我的商店

切换到 db mystore

> db.foo.insert({name:'foo'})

这将插入一个具有名为 name 的单个属性的文档。您还可以指定额外的

如果您愿意，可以在文档中添加属性，或者添加其他文档。现在，使用 find()

命令检索您插入的文档：

> db.foo.find()

该文档现在应该存储在 MongoDB 的数据文件中，这些文件位于 /data/db 目录中。由于这是您安装 GCE 永久磁盘的位置，因此该文档应该是

永久保存。

重新创建 Pod 并验证它是否可以读取先前 Pod 保存的数据

您现在可以退出 mongodb 客户端（输入 exit 并按 Enter），之后您可以删除

pod 并重新创建它：

$ kubectl 删除 pod mongodb

pod "mongodb" 已删除

$ kubectl create -f mongodb-pod-gcepd.yaml pod "mongodb" 已创建

由于新 pod 是前一个 pod 的精确副本，因此它指向与前一个 pod 相同的 GCE 持久磁盘，因此在其中运行的 MongoDB 容器应该看到完全相同的文件，

即使新的 Pod 被调度到另一个节点。

提示 您可以通过运行 kubectl get po -o Wide 来查看 pod 被调度到哪个节点。

容器启动后，您可以再次运行MongoDB客户端，检查之前是否有

仍然可以检索存储的文档，如下列表所示。

清单 7.7 在新的 Pod 中检索 MongoDB 的持久数据

$ kubectl exec -it mongodb mongo

...

> 使用我的商店

切换到 db mystore

正如预期的那样，即使您删除并重新创建了 pod，数据仍然存在。这证实了

您可以使用 GCE 永久性磁盘在同一磁盘的多个实例中保存数据

pod. To be more precise, it isn't the same pod. They are two different pods pointing to the same underlying persistent storage.

You might wonder if you can use the same persistent disk in two or more pods at the same time. The answer to this question is not trivial, because it requires that you understand exactly how external volumes are mounted in pods. I'll explain this in section 7.3.3. Before I do, I need to explain how to mount external storage when your cluster isn't running on Google's infrastructure.

7.3.2 Using other persistent volume types

In the previous exercise, I explained how to add persistent storage to a pod running in Google Kubernetes Engine. If you are running your cluster elsewhere, you should use whatever volume type is supported by the underlying infrastructure.

For example, if your Kubernetes cluster runs on Amazon's AWS EC2, you can use an `awsElasticBlockStore` volume. If your cluster runs on Microsoft Azure, you can use the `azureFile` or the `azureDisk` volume. I won't go into detail about how to do this, but it's practically the same as in the previous example. You first need to create the actual underlying storage and then set the right fields in the volume definition.

USING AN AWS ELASTIC BLOCK STORE VOLUME

For example, if you want to use an AWS elastic block store volume instead of the GCE Persistent Disk, you only need to change the volume definition as shown in the following listing.

Listing 7.8 A pod using an awsElasticBlockStore volume: mongodb-pod-aws.yaml

```
apiVersion: v1
kind: Pod
metadata:
  name: mongodb
spec:
  volumes:
  - name: mongodb-data
      awsElasticBlockStore:          #A
        volumeId: mongodb            #B
        fsType: ext4                 #C
  containers:
  - ...
```

#A Using awsElasticBlockStore instead of gcePersistentDisk
#B Specify the ID of the EBS volume you created.
#C The filesystem type is ext4.

USING AN NFS VOLUME

If your cluster runs on your own servers, you have a range of other supported options for adding external storage to your pods. For example, to mount a simple NFS share, you only need to specify the NFS server and the path exported by the server, as shown in the following listing.

英。更准确地说，这不是同一个 Pod。它们是两个不同的 pod，指向同一个

底层持久存储

您可能想知道是否可以同时在两个或多个 Pod 中使用同一个永久磁盘
时间。这个问题的答案并不简单，因为它要求你准确理解

如何在 Pod 中安装外部卷。我将在第 7.3.3 节中对此进行解释。在我这样做之前，我需要

解释当您的集群未在 Google 上运行时如何安装外部存储

基础设施。

7.3.2 使用其他持久卷类型

在上一个练习中，我解释了如何向在 Google 中运行的 pod 添加持久存储

Kubernetes 引擎。如果您在其他地方运行集群，则应该使用任何卷

类型由底层基础设施支持。

例如，如果您的 Kubernetes 集群在 Amazon 的 AWS EC2 上运行，您可以使用
awsElasticBlockStore 卷。如果您的集群在 Microsoft Azure 上运行，您可以使用

azureFile 或 azureDisk 卷。我不会详细说明如何做到这一点，但它是

实际上与前面的示例相同。您首先需要创建实际的底层存储，然后在卷定义中设置正确的字段。
使用 AWS Elastic Block Store 卷

例如，如果您想使用 AWS 弹性块存储卷而不是 GCE 持久存储卷

磁盘，您只需更改卷定义，如下列表所示。

清单 7.8 使用 awsElasticBlockStore 卷的 pod：mongodb-pod-aws.yaml

api版本：v1 种
类：Pod 元数
据：

名称：mongodb
规格：

```
卷：  awsElasticBlockStore :          #A
        卷 ID：mongodb                #B
        文件系统类型：ext4            #C
 容器：  -
 ...
```

#A 使用 awsElasticBlockStore 而不是
gcePersistentDisk #B 指定您创建的 EBS 卷的
ID。 #C 文件系统类型是 ext4。

使用 NFS 卷

如果您的集群在您自己的服务器上运行，您可以使用一系列其他受支持的选项来向 Pod 添加外部存储。例如挂载一个简单的NFS共享，只需要

指定 NFS 服务器以及服务器导出的路径，如下清单所示。

Listing 7.9 A pod using an nfs volume: mongodb-pod-nfs.yaml

```
volumes:
- name: mongodb-data
    nfs:                          #A
      server: 1.2.3.4             #B
      path: /some/path            #C
```

#A This volume is backed by an NFS share.
#B The IP of the NFS server
#C The path exported by the server

> **NOTE** Although Kubernetes supports nfs volumes, the operating system running on the worker nodes provisioned by Minikube or kind might not support mounting nfs volumes.

USING OTHER STORAGE TECHNOLOGIES

Other supported options are `iscsi` for mounting an ISCSI disk resource, `glusterfs` for a GlusterFS mount, `rbd` for a RADOS Block Device, `flexVolume`, `cinder`, `cephfs`, `flocker`, `fc` (Fibre Channel), and others. You don't need to understand all these technologies. They're mentioned here to show you that Kubernetes supports a wide range of these technologies, and you can use the technologies that are available in your environment or that you prefer.

For details on what properties you need to set for each of these volume types, you can either refer to the Kubernetes API definitions in the Kubernetes API reference or look up the information by running `kubectl explain pod.spec.volumes`. If you're already familiar with a particular storage technology, you should be able to use the `explain` command to easily find out how to configure the correct volume type.

WHY DOES KUBERNETES FORCE SOFTWARE DEVELOPERS TO UNDERSTAND LOW-LEVEL STORAGE?

If you're a software developer and not a system administrator, you might wonder if you really need to know all this low-level information about storage volumes? As a developer, should you have to deal with infrastructure-related storage details when writing the pod definition, or should this be left to the cluster administrator?

At the beginning of this book, I explained that Kubernetes abstracts away the underlying infrastructure. The configuration of storage volumes explained earlier clearly contradicts this. Furthermore, including infrastructure-related information, such as the NFS server hostname directly in a pod manifest means that this manifest is tied to this particular Kubernetes cluster. You can't use the same manifest without modification to deploy the pod in another cluster.

Fortunately, Kubernetes offers another way to add external storage to your pods. One that divides the responsibility for configuring and using the external storage volume into two parts. The low-level part is managed by cluster administrators, while software developers only specify the high-level storage requirements for their applications. Kubernetes then connects the two parts.

200

清单 7.9 使用 nfs 卷的 pod：mongodb-pod-nfs.yaml

卷：

```
  ~网络文件系统：                #A
    服务器：1.2.3.4              #B
    路径：/一些/路径             #C
```

#A 该卷由 NFS 共享支持。 #B
NFS服务器的IP

> 注意：虽然 Kubernetes 支持 nfs 卷，但运行在工作节点上的操作系统
> Minikube 或 kind 提供的可能不支持挂载 nfs 卷。

使用其他存储技术

其他支持的选项包括用于挂载 ISCSI 磁盘资源的 iscsi、用于挂载 ISCSI 磁盘资源的 glusterfs。

GlusterFS 安装、用于 RADOS 块设备的 rbd、flexVolume、cinder、cephfs、flocker、fc

（光纤通道）等。您不需要了解所有这些技术。这里提到它们是为了向您展示 Kubernetes 支持广泛的这些技术，并且

您可以[illegible]信息，您可以
请参阅 Kubernetes API 参考中的 Kubernetes API 定义或通过运行 kubectlexplain pod.spec.volumes 查找信息。如果您已经熟悉

对于特定的存储技术，您应该能够使用说明命令轻松了解如何配置正确的卷类型。

为什么 Kubernetes 迫使软件开发人员了解底层存储？

如果您是软件开发人员而不是系统管理员，您可能想知道是否真的需要了解有关存储卷的所有这些低级信息？作为一名开发人员，您是否应该

在编写 pod 定义时必须处理与基础设施相关的存储细节，还是应该将其留给集群管理员？

在本书的开头，我解释了 Kubernetes 抽象了底层基础设施。前面解释的存储卷的配置显然与此相矛盾。

此外，包含基础设施相关信息（例如直接在 pod 清单中的 NFS 服务器主机名）意味着该清单与该特定 Kubernetes 集群相关联。

您无法在不进行修改的情况下使用相同的清单将 pod 部署到另一个集群中。
幸运的是，Kubernetes 提供了另一种向 Pod 添加外部存储的方法。一个那个
将配置和使用外部存储卷的职责分为两部分。底层部分由集群管理员管理，而软件开发人员只指定

其应用程序的高级存储要求。然后 Kubernetes 将这两个部分连接起来。

You'll learn about this in the next chapter, but first you need a basic understanding of pod volumes. You've already learned most of it, but I still need to explain some details.

7.3.3 Understanding how external volumes are mounted

To understand the limitations of using external volumes in your pods, whether a pod references the volume directly or indirectly, as explained in the next chapter, you must be aware of the caveats associated with the way network storage volumes are actually attached to the pods.

Let's return to the issue of using the same network storage volume in multiple pods at the same time. What happens if you create a second pod and point it to the same GCE Persistent Disk?

I've prepared a manifest for a second MongoDB pod that uses the same GCE Persistent Disk. The manifest can be found in the file `mongodb2-pod-gcepd.yaml`. If you use it to create the second pod, you'll notice that it never runs. Even after a few minutes, its status is still shown as `ContainerCreating`:

```
$ kubectl get po
NAME      READY  STATUS             RESTARTS  AGE
mongodb   1/1    Running            0         10m
mongodb2  0/1    ContainerCreating  0         2m15s
```

You can see why this is the case with the `kubectl describe` command. At the very bottom, you see a `FailedAttachVolume` event generated by the `attachdetach-controller`. The event has the following message:

```
AttachVolume.Attach failed for volume "mongodb-data" : googleapi: Error 400:
      RESOURCE_IN_USE_BY_ANOTHER_RESOURCE - The disk resource 'projects/-xyz/zones/europe-west3-
      c/disks/mongodb' is already being used by 'projects/ xyz/zones/europe-west3-
      c/instances/gke-kubia-default-pool-xyz-1b27'
```

The message indicates that the node hosting the `mongodb2` pod can't attach the external volume because it's already in use by another node. If you check where the two pods are scheduled, you'll see that they are not on the same node:

```
$ kubectl get po -o wide
NAME      READY  STATUS             ... NODE
mongodb   1/1    Running            ... gke-kubia-default-pool-xyz-1b27
mongodb2  0/1    ContainerCreating  ... gke-kubia-default-pool-xyz-gqbj
```

The `mongodb` pod is running on node `1b27`, while the `mongodb2` pod is scheduled to node `gqbj`. As is typically the case in cloud environments, you can't mount the same GCE Persistent Disk on multiple hosts simultaneously in read/write mode. You can only mount it on multiple hosts in read-only mode.

Interestingly, the error message doesn't say that the disk is being used by the `mongodb` pod, but by the node hosting the pod. And this is a very important detail about how external volumes are mounted into pods.

201

您将在下一章中了解这一点，但首先您需要对 pod 有基本的了解
卷。你已经了解了大部分内容，但我仍然需要解释一些细节。

7.3.3 了解外部卷的挂载方式

要了解在 Pod 中使用外部卷的限制，Pod 是否引用

正如下一章所述，您必须了解与网络存储卷实际附加到 Pod 的方式相关的注意事项。

让我们回到在多个 Pod 中同时使用相同网络存储卷的问题。如果您创建第二个 pod 并将其指向同一个 GCE 持久性，会发生什么

磁盘？

我已经为使用相同 GCE 持久磁盘的第二个 MongoDB Pod 准备了清单。
清单可以在文件 mongodb2-pod-gcepd.yaml 中找到。如果您使用它来创建第二个 Pod，您会发现它永远不会运行。即使几分钟后，其状态仍然显示

作为容器创建：

```
$ kubectl 获取 po          重新开始时代
姓名 准备状态 0 10m
蒙古数据库 1/1 跑步 0 10m
mongodb2 0/1 容器创建 0 2分15秒
```

您可以使用 kubectl describe 命令了解为什么会出现这种情况。在最底部，您可以看到
Attachdetach-controller 生成的 FailedAttachVolume 事件。事件

有以下消息：

```
AttachVolume.Attach 卷“mongodb-data”失败：googleapi：错误 400：
      RESOURCE_IN_USE_BY_ANOTHER_RESOURCE - 磁盘资源“projects/-
      xyz/zones/europe-west3c/disks/mongodb”已被“projects/
      xyz/zones/europe-west3c/instances/gke-kubia-default-pool-xyz-1b27”使用'
```

该消息表明托管 mongodb2 pod 的节点无法附加外部卷，因为它已被另一个节点使用。如果
你检查两个 Pod 的位置

安排好后，您会看到它们不在同一节点上：

```
$ kubectl 获取 po -o 宽 节点
姓名 准备状态 ... gke-kubia-default-pool-xyz-1b27
蒙古数据库 1/1 跑步 ... gke-kubia-default-pool-xyz-gqbj
mongodb2 0/ContainerCreating ... gke-kubia-default-pool-xyz-gqbj
```

mongodb pod 运行在节点 1b27 上，而 mongodb2 pod 调度到节点 gqbj 上。与云环境中的常见情况一样，您无法挂载相同的 GCE 持久性

多个主机上的磁盘同时处于读/写模式。您只能将其安装在多个

主机处于只读模式。

有趣的是，错误消息并没有说磁盘正在被 mongodb pod 使用，
而是由托管 pod 的节点执行。这是关于外部卷如何进行的一个非常重要的细节

被安装到吊舱中。

As the following figure shows, a network volume is mounted by the host node, and then the pod is given access to the mount point. Typically, the underlying storage technology doesn't allow a volume to be attached to more than one node at a time in read/write mode, but multiple pods on the same node *can* all use the volume in read/write mode.

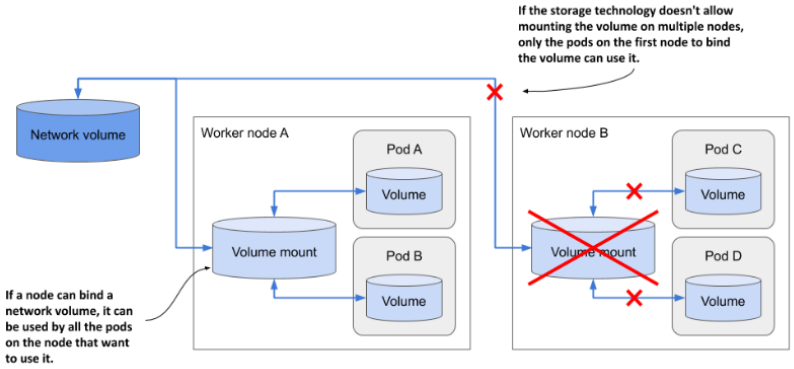


Figure 7.9 Network volumes are mounted by the host node and then exposed in pods

For most storage technologies available in the cloud, the only way to use the same network volume on multiple nodes simultaneously is to mount them in read-only mode. For example, pods scheduled to different nodes can use the same GCE Persistent Disk if it is mounted in read-only mode, as shown in the next listing:

Listing 7.10 Mounting a GCE Persistent Disk in read-only mode

```
kind: Pod
spec:
  volumes:
  - name: my-volume
    gcePersistentDisk:
      pdName: my-volume
      fsType: ext4
      readOnly: true    #A
```

#A This GCE Persistent Disk is mounted in read-only mode

It is important to consider this network storage limitation when designing the architecture of your distributed application. Replicas of the same pod typically can't use the same network volume in read/write mode. Fortunately, Kubernetes takes care of this, too. In chapter 13, you'll learn how to deploy stateful applications, where each pod instance gets its own network storage volume.

202

如下图所示，主机节点挂载了一个网络卷，然后
pod 有权访问挂载点。通常，底层存储技术不会

允许一个卷在读/写模式下一次附加到多个节点，但可以是多个

同一节点上的 Pod 都可以以读/写模式使用该卷

图7.9 网络卷由主机节点挂载，然后暴露在pod中

对于云中可用的大多数存储技术来说，使用同一网络的唯一方法

同时在多个节点上创建卷是以只读模式挂载它们。例如，调度到不同节点的 Pod 可以使用相同的 GCE 持久磁盘（如果它以只读方式挂载）

only 模式，如下清单所示：
清单 7.10 以只读模式挂载 GCE 持久磁盘

种类：
Pod 规
格：名称：我的卷
gcePersistentD
isk：
pdName：我的# A
卷 fsType：
#A 此 GCE 持久磁盘以只读模式挂载

在设计分布式应用程序的体系结构时，考虑这种网络存储限制非常重要。同一 Pod 的副本通常不能使用同一网络

卷处于读/写模式。幸运的是，Kubernetes 也解决了这个问题。在第 13 章中，您将学习如何部署有状态应用程序，其中每个 Pod 实例都有自己的网络存储

体积。

You're now done playing with the MongoDB pods, so delete them both, but hold off on deleting the underlying GCE persistent disk. You'll use it again in the next chapter.

7.4 Accessing files on the worker node's filesystem

Most pods shouldn't care which host node they are running on, and they shouldn't access any files on the node's filesystem. System-level pods are the exception. They may need to read the node's files or use the node's filesystem to access the node's devices or other components via the filesystem. Kubernetes makes this possible through the `hostPath` volume type. I already mentioned it in the previous section, but this is where you'll learn when to actually use it.

7.4.1 Introducing the hostPath volume

A `hostPath` volume points to a specific file or directory in the filesystem of the host node, as shown in the next figure. Pods running on the same node and using the same path in their `hostPath` volume see the same files.

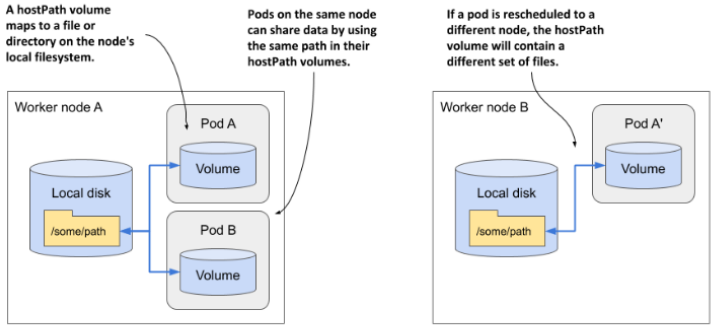


Figure 7.10 A `hostPath` volume mounts a file or directory from the worker node's filesystem into the container.

A `hostPath` volume is not a good place to store the data of a database. Because the contents of the volume are stored on the filesystem of a specific node, the database pod will not be able to see the data if it gets rescheduled to another node.

Typically, a `hostPath` volume is used in cases where the pod actually needs to read or write files written or read by the node itself, such as system-level logs.

The `hostPath` volume type is one of the most dangerous volume types in Kubernetes and is usually reserved for use in privileged pods only. If you allow unrestricted use of the `hostPath` volume, users of the cluster can do anything they want on the node. For example, they can use it to mount the Docker socket file (typically `/var/run/docker.sock`) in their container and then

您现在已经完成了 MongoDB Pod 的操作，因此将它们都删除，但请先缓一缓
删除底层 GCE 永久磁盘。您将在下一章中再次使用它。

7.4 访问工作节点文件系统上的文件

大多数 Pod 不应该关心它们在哪个主机节点上运行，并且它们不应该访问节点文件系统上的任何文件。系统级 Pod 是例外。他们可能需要阅读

节点的文件或使用节点的文件系统通过以下方式访问节点的设备或其他组件

文件系统。 Kubernetes 通过 hostPath 卷类型使这成为可能。我已经

在上一节中提到过，但这是您将了解何时实际使用它的地方。

7.4.1 引入hostPath卷

HostPath卷指向主机节点文件系统中的特定文件或目录，如下图所示。在同一节点上运行并在其 hostPath 卷中使用相同路径的 Pod 会看到相同的文件。

图 7.10 hostPath 卷将文件或目录从工作节点的文件系统挂载到容器中。

HostPath 卷不是存储数据库数据的好地方。由于卷的内容存储在特定节点的文件系统上，因此数据库 pod 将无法

查看数据是否被重新安排到另一个节点。

通常，hostPath 卷用于 pod 实际需要读取或写入的情况

节点本身写入或读取的文件，例如系统级日志。

hostPath 卷类型是 Kubernetes 中最危险的卷类型之一，
通常保留仅供特权 Pod 使用。如果您允许不受限制地使用 hostPath 卷，则集群的用户可以在节点上执行任何操作。例如，他们可以使用

它用于挂载 Docker 套接字文件（通常为 /var/run/docker

run the Docker client within the container to run any command on the host node as the root user. You'll learn how to prevent this in chapter 24.

7.4.2 Using a hostPath volume

To demonstrate how dangerous `hostPath` volumes are, let's deploy a pod that allows you to explore the entire filesystem of the host node from within the pod. The pod manifest is shown in the following listing.

Listing 7.11 Using a hostPath volume to gain access to the host node's filesystem

```
apiVersion: v1
kind: Pod
metadata:
  name: node-explorer
spec:
  volumes:
  - name: host-root                #A
    hostPath:                      #A
      path: /                      #A
  containers:
  - name: node-explorer
    image: alpine
    command: ["sleep", "9999999999"]
    volumeMounts:                  #B
    - name: host-root              #B
      mountPath: /host             #B
```

#A The hostPath volume points to the root directory on the node's filesystem
#B The volume is mounted in the container at /host

As you can see in the listing, a `hostPath` volume must specify the `path` on the host that it wants to mount. The volume in the listing will point to the root directory on the node's filesystem, providing access to the entire filesystem of the node the pod is scheduled to.

After creating the pod from this manifest using kubectl apply, run a shell in the pod with the following command:

```
$ kubectl exec -it node-explorer -- sh
```

You can now navigate to the root directory of the node's filesystem by running the following command:

```
/ # cd /host
```

From here, you can explore the files on the host node. Since the container and the shell command are running as root, you can modify any file on the worker node. Be careful not to break anything.

Now imagine you're an attacker that has gained access to the Kubernetes API and are able to deploy this type of pod in a production cluster. Unfortunately, at the time of writing, Kubernetes doesn't prevent regular users from using `hostPath` volumes in their pods and is

在容器内运行 Docker 客户端，以 root 身份在主机节点上运行任何命令
用户。您将在第 24 章中了解如何防止这种情况发生。

7.4.2 使用 hostPath 卷

为了演示 hostPath 卷有多么危险，让我们部署一个 pod，它允许您
从 pod 内探索主机节点的整个文件系统。 Pod 清单如以下清单所示。

清单 7.11 使用 hostPath 卷来访问主机节点的文件系统

```
api版本：v1 种
类：Pod 元数
据：

名称：节点资源
管理器规格：
  名称：主机根          # A
  主机路径：            # A
  小路：/              # A
容器：

 - 名称：node-
  命令：[ "睡'
  眠"      "9999999999"#B
卷安装：vol
  名称：主机根          #B
  挂载路径：/host       #B
```

#A hostPath 卷指向节点文件系统上的根目录 #B 该卷安装在容
器中的 /host

正如您在清单中所看到的，hostPath 卷必须指定它所在的主机上的路径。
想要安装。列表中的卷将指向节点文件系统上的根目录，提供对 pod 计划到的节点的整个文件系统
的访问。

使用 kubectl apply 从此清单创建 pod 后，使用以下命令在 pod 中运行 shell
以下命令：

```
$ kubectl exec -it 节点资源
```

您现在可以通过运行以下命令导航到节点文件系统的根目录
命令：

```
/# cd /主
```

从这里，您可以浏览主机节点上的文件。由于容器和外壳
命令以 root 身份运行，您可以修改工作节点上的任何文件。小心不要弄坏任何东西。

现在假设您是一名攻击者，已经获得了 Kubernetes API 的访问权限，并且能够在生产集群中部署
这种类型的 Pod。不幸的是，在撰写本文时，

Kubernetes 不会阻止普通用户在其 pod 中使用 hostPath 卷，并且

therefore totally unsecure. As already mentioned, you'll learn how to secure the cluster from this type of attack in chapter 24.

SPECIFYING THE TYPE FOR A HOSTPATH VOLUME

In the previous example, you only specified the path for the `hostPath` volume, but you can also specify the `type` to ensure that the path represents what that the process in the container expects (a file, a directory, or something else).

The following table explains the supported `hostPath` types:

Type	Description
<empty>	Kubernetes performs no checks before it mounts the volume.
Directory	Kubernetes checks if a directory exists at the specified path. You use this type if you want to mount a pre-existing directory into the pod and want to prevent the pod from running if the directory doesn't exist.
DirectoryOrCreate	Same as Directory, but if nothing exists at the specified path, an empty directory is created.
File	The specified path must be a file.
FileOrCreate	Same as File, but if nothing exists at the specified path, an empty file is created.
BlockDevice	The specified path must be a block device.
CharDevice	The specified path must be a character device.
Socket	The specified path must be a UNIX socket.

Table 7.3 Supported hostPath volume types

If the specified path doesn't match the type, the pod's containers don't run. The pod's events explain why the hostPath type check failed.

> **NOTE** When the type is `FileOrCreate` or `DirectoryOrCreate` and Kubernetes needs to create the file/directory, its file permissions are set to `644` (`rw-r--r--`) and `755` (`rwxr-xr-x`), respectively. In either case, the file/directory is owned by the user and group used to run the Kubelet.

7.5 Summary

This chapter has explained the basics of adding volumes to pods, but this was only the beginning. You'll learn more about this topic in the next chapter. So far you've learned this:

- Pods consist of containers and volumes. Each volume can be mounted at the desired location in the container's filesystem.
- Volumes are used to persist data across container restarts, share data between the containers in the pod, and even share data between the pods.
- Many volume types exist. Some are generic and can be used in any cluster regardless of the cluster environment, while others, such as the `gcePersistentDisk`, can only be used if the cluster runs on a particular cloud provider infrastructure.

205

因此完全不安全。如前所述，您将学习如何保护集群免受

第 24 章中介绍了这种类型的攻击。

指定主机路径卷的类型

在前面的示例中，您仅指定了 hostPath 卷的路径，但您也可以

指定类型以确保路径代表容器中的进程所期望的内容（文件、目录或其他内容）。

下表说明了支持的 hostPath 类型：

类型	描述
<空>	Kubernetes 在挂载卷之前不执行任何检查。
目录	Kubernetes 检查指定路径中是否存在目录。如果您使用此类型
目录或创建	
文件	指定的路径必须是一个文件。
文件或创建	与 File 相同，但如果指定路径中不存在任何内容，则会创建一个空文件。
块设备	指定的路径必须是块设备。
字符设备	指定的路径必须是字符设备。
插座	指定的路径必须是 UNIX 套接字。

表 7.3 支持的 hostPath 卷类型

如果指定的路径与类型不匹配，Pod 的容器将不会运行。 Pod 的事件
解释主机路径类型检查失败的原因。

> 说明 当类型为FileOrCreate或DirectoryOrCreate时，Kubernetes需要创建
> 文件/目录，其文件权限分别设置为644（rw-r--r--）和755（rwxr-xr-x）。在任一

7.5 概括

本章解释了向 pod 添加卷的基础知识，但这仅仅是开始。

您将在下一章中了解有关此主题的更多信息。到目前为止，您已经了解了这一点：

- Pod 由容器和卷组成。每个卷都可以
- 卷用于在容器重新启动时保留数据，在容器之间共享数据
- 存在许多卷类型。有些是通用的，可以在任何集群中使用，无论

- An `emptyDir` volume is used to store data for the duration of the pod. It starts as an empty directory just before the pod's containers are started and is deleted when the pod terminates.
- The `gitRepo` volume is a deprecated volume type that is initialized by cloning a Git repository. Alternatively, an `emptyDir` volume can be used in combination with an init container that initializes the volume from Git or any other source.
- Network volumes are typically mounted by the host node and then exposed to the pod(s) on that node.
- Depending on the underlying storage technology, you may or may not be able to mount a network storage volume in read/write mode on multiple nodes simultaneously.
- By using a proprietary volume type in a pod manifest, the pod manifest is tied to a specific Kubernetes cluster. The manifest must be modified before it can be used in another cluster. Chapter 8 explains how to avoid this issue.
- The `hostPath` volume allows a pod to access any path in filesystem of the worker node. This volume type is dangerous because it allows users to make changes to the configuration of the worker node and run any process they want on the node.

In the next chapter, you'll learn how to abstract the underlying storage technology away from the pod manifest and make the manifest portable to any other Kubernetes cluster.

206

- 一个emptyDir卷用于存储pod运行期间的数据。它在 pod 的容器启动之前作为一个空目录启动，并在 pod 启动时被删除

  终止。
-

gitRepo 卷是一种已弃用的卷类型，通过克隆 Git 来初始化

存储库，可卷。emptyDir 卷可以与 init 容器结合使用，该容器从 Git 或任何其他源初始化
- 网络卷通常由主机节点安装，然后暴露给该节点上的 pod。
-

根据底层存储技术，你可能能够也可能无法安装
- 通过在 pod 清单中使用专有卷类型，pod 清单将绑定到特定的 Kubernetes 集群。清单必须先修改才能用于

  另一个集群。第 8 章解释了如何避免这个问题。
-

hostPath 卷允许 Pod 访问工作节点文件系统中的任何路径。

这种卷类型很危险，因为它允许用户更改

在下一章中，您将学习如何从 pod 清单中抽象出底层存储技术，并使清单可移植到任何其他 Kubernetes 集群。