O'REILLY®

HZ BOOKS
华章IT

Helm
学习指南

Kubernetes上的应用程序管理

Learning Helm

[美] Matt Butcher Matt Farina
Josh Dolitsky 著
卢涛 译

机械工业出版社
China Machine Press

版权信息
COPYRIGHT INFORMATION

书名：Helm学习指南:Kubernetes上的应用程序管理
作者：(美)马特·布彻(Matt Butcher);(美)马特·法里纳(Matt Farina);(美)乔什·多利茨基(Josh Dolitsky)
排版：skip
出版社：机械工业出版社
出版时间：2021-09-01
ISBN：9787111689959

关于作者

Matt Butcher是Helm项目的联合创始人，并在微软Azure领导一个开源工程师团队。Matt与云原生计算基金会的Karen Chu合著了The Illustrated Children’s Guide to Kubernetes，并撰写了另外8本书（其中与Matt Farina合著两本）。他还拥有哲学博士学位，平时喜欢喝咖啡和徒步旅行。

Matt Farina是Helm项目的维护者，已经为开源项目贡献了超过15年的时间。他参与创建并主持了Kubernetes应用程序特别兴趣小组（SIG），该小组专注于在Kubernetes上运行工作负载。Matt在SUSE担任软件架构师，负责Kubernetes和工具开发。解决重大问题和帮助他人是Matt从事软件工作的两大驱动力。

Josh Dolitsky是Helm项目的维护者和ChartMuseum项目的创始人。他是Blood Orange的所有者兼首席工程师，Blood Orange是一家专门帮助人们使用DevOps、CI/CD和Kubernetes的软件咨询公司。Josh喜欢承担大型且有潜力的软件项目。

关于封面

本书封面上的动物是一种小䴙䴘（学名tachybaptus ruficollis），也被称为dabchick，它是一种分布在欧洲、非洲和南亚的小水鸟。

小䴙䴘有一个尖喙，在它成熟后喙的颜色由黄变黑，周围有白色的斑纹。它的羽毛大部分是深色的，从背部一直延伸到圆钝的尾部，它的颈部羽毛呈铁锈色，腹部的颜色较浅。小䴙䴘繁殖时的叫声是一种颤音，有点像马的嘶鸣声。

就像所有的䴙䴘一样，它的腿向后伸展，虽然它在陆地上行走很困难，但它是游泳和潜水健将。它以昆虫、软体动物、蝌蚪和小鱼为食，并在水边筑巢。小䴙䴘幼时被父母用羽毛喂养，以形成柔软的胃黏膜，可防止鱼骨和贝壳造成胃损伤。

目前小䴙䴘的保护状况被列为Least Concern，但O’Reilly系列丛书封面上的许多动物都濒临灭绝，它们对世界都很重要。

封面插图由Karen Montgomery根据Elements of Ornithology中的黑白版画制作而成。

O’Reilly Media, Inc.介绍

O’Reilly以“分享创新知识，改变世界”为己任。40多年来我们一直向企业、个人提供成功所必需之技能及思想，激励他们创新并做得更好。

O’Reilly业务的核心是独特的专家及创新者网络，众多专家及创新者通过我们分享知识。我们的在线学习（Online Learning）平台提供独家的直播培训、图书及视频，使客户更容易获取业务成功所需的专业知识。几十年来O’Reilly图书一直被视为学习开创未来之技术的权威资料。我们每年举办的诸多会议是活跃的技术聚会场所，来自各领域的专业人士在此建立联系，讨论最佳实践并发现可能影响技术行业未来的新趋势。

我们的客户渴望做出推动世界前进的创新之举，我们希望能助他们一臂之力。

业界评论

“O’Reilly Radar博客有口皆碑。”

——Wired

“O’Reilly凭借一系列非凡想法（真希望当初我也想到了）建立了数百万美元的业务。”

——Business 2.0

“O’Reilly Conference是聚集关键思想领袖的绝对典范。”

——CRN

“一本O’Reilly的书就代表一个有用、有前途、需要学习的主题。”

——Irish Times

“Tim是位特立独行的商人，他不光放眼于最长远、最广阔的领域，并且切实地按照Yogi Berra的建议——如果你在路上遇到岔路口，那就走小路——去做了。回顾过去，Tim似乎每一次都选择了小路，而且有几次都是一闪即逝的机会，尽管大路也不错。”

——Linux Journal

前言

Helm是用于流行的开源容器管理平台Kubernetes的软件包管理器。

软件包管理器使平台更易于访问。使用Kubernetes等平台，你需要在上面运行软件，而且上面的大部分软件都是现成的或共享的。Helm这一软件包管理器以易于使用的方式对软件进行了打包，使你能够快速安装并使用某个软件。

使用软件包管理器，你可以很容易地与其他人共享某个软件。当某个平台上有各种各样的软件在运行时，此平台会更有用。开源项目和公司都喜欢让它们的软件在其运行的平台上易于安装，而Helm使Kubernetes做到了这一点。

软件包管理器不仅可以用来共享和使用他人的软件，它通常也是其他系统（如DevOps工具）不可分割的一部分，并被用作基础构件。

几乎每个现代平台都有一个软件包管理器。各种操作系统、编程语言和云平台都有某种形式的软件包管理器。

在本书中，你将了解Helm，它为Kubernetes提供了现代的软件包管理，以及你可以使用的软件包（称为chart）。你将学习如何使用Helm，如何创建软件包，以及如何与其他平台共享软件包。

本书受众

如果你是Kubernetes的新手，或者想学习如何安装现成的应用程序，本书将帮助你学习如何使用Helm达成目标。通过Helm安装应用程序要比通过Kubernetes手工安装应用程序容易得多，也快得多。

如果你为一家公司（或某个项目）工作，希望以一种简便的方式将应用程序分发给Kubernetes用户，那么这本书将教你如何使用Helm做到这一点。Helm能够帮助你快速安装应用程序，从而使项目启动更容易。

本书也是为DevOps专业人士准备的，通过学习本书，他们可以学会将Kubernetes软件包管理作为DevOps工具链的一部分来使用。Helm提供了强大的高级功能，可以作为其他自动化的基础构件。这些功能已经被用于在Kubernetes上部署复杂的大型应用程序，本书将教你如何利用这些功能。

为什么写作本书

我们不仅想提供文档中经常可以找到的技术细节，还想提供有关Helm能做什么以及为什么要这样做的背景和见解。

本书内容结构

前三章介绍Helm并展示如何使用Helm客户端。第1章概述了Helm在云原生生态系统中的位置及其架构。第2章和第3章介绍如何使用Helm客户端——从安装逐步过渡到高级用法。

第4～6章介绍了如何为Helm创建软件包。该部分从如何创建包（第4章）开始，然后学习模板语法（第5章），最后学习高级功能（第6章）。

第7章介绍了共享软件包，包括它们各自的发布版本。如果你正在使用DevOps进程将软件分发给其他人或在系统之间共享软件，则共享非常重要。

第8章介绍了Helm的扩展。可以在无须对其建立分支或增加功能的前提下，对Helm进行定制。

两个附录提供了参考资料。附录A概述了当前软件包和遗留软件包之间的区别，附录B介绍了用于共享包的存储库API。

排版约定

本书中使用以下排版约定：

斜体（Italic）

表示新的术语、URL、电子邮件地址、文件名和文件扩展名。

等宽字体（Constant width）

用于程序清单，以及段落中的程序元素，例如变量名、函数名、数据库、数据类型、环境变量、语句以及关键字。

等宽粗体（Constant width bold）

表示应由用户直接输入的命令或其他文本。

等宽斜体（Constant width italic）

表示应由用户提供的值或由上下文确定的值替换的文本。

 该图示表示提示或建议。

 该图示表示一般性说明。

 该图示表示警告或注意。

示例代码

可以从https://github.com/masterminds/learning-helm下载补充材料（示例代码、练习、勘误等）。

这里的代码是为了帮助你更好地理解本书的内容。通常，可以在程序或文档中使用本书中的代码，而不需要联系O’Reilly获得许可，除非需要大段地复制代码。例如，使用本书中所提供的几个代码片段来编写一个程序不需要得到我们的许可，但销售或发布O’Reilly书籍中的示例代码需要获得许可。引用本书的示例代码来回答问题也不需要许可，将本书中的很大一部分示例代码放到自己的产品文档中则需要获得许可。

非常欢迎读者使用本书中的代码，希望（但不强制）注明出处。注明出处的形式包含书名、作者、出版社和ISBN，例如：

Learning Helm：Managing Apps on Kubernetes，作者Matt Butcher、Matt Farina和Josh Dolitsky，由O’Reilly出版，书号978-1-492-08365-8

如果读者觉得对示例代码的使用超出了上面所给出的许可范围，欢迎通过permissions@oreilly.com联系我们。

O’Reilly在线学习平台（O’Reilly Online Learning）

40多年来，O’Reilly Media致力于提供技术和商业培训、知识和卓越见解，来帮助众多公司取得成功。我们拥有独一无二的专家和革新者组成的庞大网络，他们通过图书、文章、会议和我们的在线学习平台分享他们的知识和经验。O’Reilly的在线学习平台允许你按需访问现场培训课程、深入的学习路径、交互式编程环境，以及O’Reilly和200多家其他出版商提供的大量文本和视频资源。有关的更多信息，请访问http://oreilly.com。

如何联系我们

对于本书，如果有任何意见或疑问，请按照以下地址联系本书出版商。

美国：

O’Reilly Media，Inc.

1005 Gravenstein Highway North

Sebastopol，CA 95472

中国：

北京市西城区西直门南大街2号成铭大厦C座807室（100035）

奥莱利技术咨询（北京）有限公司

要询问技术问题或对本书提出建议，请发送电子邮件至bookquestions@oreilly.com。

本书配套网站https://oreil.ly/learning-helm上列出了勘误表、示例以及其他信息。

关于书籍、课程、会议和新闻的更多信息，请访问我们的网站http://oreilly.com。

我们在Facebook上的地址：http://facebook.com/oreilly

我们在Twitter上的地址：http://twitter.com/oreillymedia

我们在YouTube上的地址：http://www.youtube.com/oreillymedia

致谢

感谢本书的技术审校者：Taylor Thomas、Jonathan Johnson和Michael Hausenblas。

感谢O’Reilly每一位帮助我们完成本书的人，尤其是John Devins和Jeff Bleiel。撰写本书的过程令人愉快。

Helm生态系统是来自世界各地的众多贡献者共同创建的。个人、非政府组织和公司已经合作开发出一种能够满足广泛需求的技术。从构建chart到提供修复程序，再到帮助其他人学习Helm，每个人都投入了时间和精力来优化代码。我们非常感谢他们。

最重要的是，我们要感谢我们各自的妻子和孩子在整个写作过程中付出的耐心和爱。

第1章

Helm简介

Helm是Kubernetes的软件包管理器。自从Helm第一次被提交到Git存储库以来，Helm开发人员就是这样描述它的。这句话是本章的主题。

在本章中，我们首先从概念上审视云原生生态系统，其中Kubernetes是它的一项关键技术。我们将重新审视Kubernetes必须提供什么来为描述Helm奠定基础。

接下来，我们将看看Helm着手解决的问题。在该节中，我们将研究软件包管理的概念，以及为什么以这种方式对Helm进行设计。我们还将了解将软件包安装到Kubernetes等集群管理工具中有哪些独特的方面。

最后，我们将介绍Helm的高级架构，重点介绍chart、模板和发布版本的概念。在本章结束时，你将了解Helm如何适应更广阔的工具生态系统，并熟悉有关Helm的术语和概念。

1.1 云原生生态系统

云技术的出现显然改变了业界对硬件、系统管理、物理网络等的看法。虚拟机取代了物理服务器，存储服务取代了硬盘驱动器，自动化工具的地位日益突出。这也许是业界对云的概念化方式的早期改变。但是，随着这种新方法的优点和缺点日益清晰，设计应用程序和服务的实践也开始发生变化。

开发人员和运营人员开始质疑在坚固的硬件上构建大型单体二进制应用程序的做法。他们认识到了在保持数据完整性的同时跨不同应用程序共享数据的困难性。分布式锁定、存储和缓存成为主流问题，而不再只是学术界关注的焦点。大型软件包被分解成更小的离散可执行文件。正如Kubernetes创始人Brendan Burns经常说的那样，“分布式计算从一个高级主题发展为计算机科学入门课程”。

云原生抓住了这种认知上的转变，我们可以称之为云的架构视角。当我们围绕云的功能和约束设计系统时，设计的就是云原生系统。

1.1.1 容器和微服务

云原生计算的核心是这样一种哲学观点，即较小的离散独立服务（smaller discrete standalone service）比处理一切事务的大型单体服务（large monolithic service）更可取。云原生方法不是编写一个大型应用程序来处理从生成用户界面到处理任务队列再到与数据库和缓存交互的所有事务，而是编写一系列较小的服务，每个服务都有相对特定的用途，然后将这些服务连接到一起以用于更高级别的用途。在这种模型中，其中一个服务可能是关系数据库的独占用户。希望访问数据的服务将（通常）通过表征状态转移（REST）API与该服务联系。而且，通过HTTP使用JavaScript对象表示法（JSON），这些其他的服务将查询和更新数据。

这种细分允许开发人员隐藏底层实现，提供针对更广泛应用程序的特定业务逻辑的一组特性。

微服务

如果一个应用程序由完成所有工作的单个可执行文件组成，则云原生应用程序就是分布式应用程序。虽然独立的程序各自负责一个或两个独立的任务，但这些程序一起构成了一个逻辑上单一的应用程序。

结合这些理论，下面我们举一个简单的例子来具体解释。想象一个电子商务网站。这类网站一般由几个任务共同组成。网站上有一个产品目录、用户账户和购物车、一个支付处理器（处理对安全敏感的货币交易过程），以及一个前端（客户可以查看商品并进行选择和购买）。还有一个管理界面，店主在这里管理库存和完成订单。

历史上，像这样的应用程序曾经作为一个单独的程序构建。负责每个工作单元的代码都被编译进一个大的可执行文件，通常在一台庞大的硬件设备上运行此文件。

然而，编写这种应用程序的云原生方法是将这个电子商务应用程序分成多个部分：一个处理支付交易，一个跟踪产品目录，还有一个提供管理功能，等等。这些服务使用定义良好的REST API通过网络相互通信。

极端地说，应用程序被分解成最小的基础构件，每个部分都是一个程序。这就是微服务架构。与单体应用程序相反，微服务仅负责处理整个应用程序的处理流程的一小部分。

微服务概念对云计算的发展产生了巨大的影响。这一点在容器计算（container computing）的出现中最为明显。

容器

容器常常被拿来和虚拟机进行比较。虚拟机在主机上的隔离环境中运行整个操作系统。相反，容器有自己的文件系统，但与主机在同一操作系统内核中执行。

还有第二种描绘容器的方法可能对目前的讨论更有利。顾名思义，容器为打包单个程序的运行时环境提供了一种有用的方法，从而确保可执行文件在从一个主机移动到另一个主机时满足其所有依赖项。

这是一种更具哲理的方法，也许是因为它对容器施加了一些非技术性的限制。例如，可以将十几个不同的程序打包在一个容器中，并同时执行它们。但容器，至少按Docker的要求设计的容器，是一个顶层程序的载体。

 当我们在这里谈论程序时，实际上是在思考一个比“二进制文件”更高层次的抽象。大多数Docker容器至少有几个可执行文件，它们只是用来辅助主程序的。但这些可执行文件都是容器主要功能的辅助文件。例如，Web服务器可能需要一些其他本地实用程序来启动或执行低级任务（例如，Apache有用于模块的工具），但主程序是Web服务器本身。

容器和微服务在设计上是完美的搭配。小的离散程序可以连同它们的所有依赖项一起打包到小巧的容器中。这些容器可以在主机之间移动。在执行容器时，主机不必拥有执行程序所需的所有工具，因为所有工具都打包在容器中了。主机只需具备运行容器的能力即可。

例如，如果一个程序是用Python 3构建的，那么主机不需要安装和配置Python并安装该程序所需的所有库，因为所有这些都被打包在容器里。当主机执行此容器时，容器中已经存储了正确版本的Python 3和每个必需的库。

更进一步，主机可以自由地执行具有互相冲突的需求的容器。容器化的Python 2程序可以与容器化的Python 3需求在同一主机上运行，并且主机的管理员无须做任何特殊的工作来配置这些互相冲突的需求！

这些例子说明了云原生生态系统的一个特性：管理员、运营人员和站点可靠性工程师（SRE）不再从事管理程序依赖项的工作。相反，他们可以把精力集中在更高层次的资源分配上。运营人员不必担心Python、Ruby和Node的哪个版本在不同的服务器上运行，而可以关注是否为这些容器化的工作负载正确分配了网络、存储和CPU资源。

在完全隔离的环境中运行程序有时是有用的。但更多的时候，我们希望将这个容器的某些方面暴露给外部世界。我们想让它能够访问存储，能够回答网络连接，能够根据我们目前的需求向容器中注入一些配置。所有这些任务（甚至更多）都是由容器运行时提供的。当容器声明它有一个服务在端口8080上进行内部监听时，容器运行时可以授予它在主机端口8000上的访问权限。因此，当主机在端口8000上收到网络请求时，容器将此视为其端口8080上的请求。同样，主机可以将文件系统装载到容器中，或者在容器中设置特定的环境变量。通过这种方式，容器可以参与到它周围更广泛的环境中——不仅包括该主机上的其他容器，还包括本地网络甚至Internet上的远程服务。

容器镜像和登记站

容器技术本身就是一个复杂而迷人的空间。但就我们的目的而言，在进入云原生堆栈的下一层之前，我们只需要了解更多关于容器如何工作的内容就够了。

如前所述，容器是一个程序及其依赖项和环境。整个过程可以打包成一个可移植的表示形式，称为容器镜像（通常简称为镜像）。镜像不是被打包成一个大的二进制文件，而是被打包成离散的层，每个层都有自己的唯一标识符。当镜像四处移动时，它们作为层的集合移动，这提供了巨大的优势。如果一个主机具有五层镜像，而另一个主机需要相同的镜像，那么它只需要获取它还没有的层。例如，如果它已经有五层中的两个，则只需要获取另外三层就可以重建整个容器。

有一项关键的技术提供了移动容器镜像的能力，即镜像登记站（image registry），这是一种专门的块存储技术，它容纳容器，使其可供主机使用。主机可以将容器镜像推送（push）到登记站，该动作会将各层传输到登记站。然后另一台主机可以将镜像从登记站拉取（pull）到该主机的环境中，如此一来该主机就可以执行此容器了。

登记站负责管理各个层。当一个主机请求某个镜像时，登记站会让该主机知道哪些层构成了该镜像。然后，主机可以确定自己缺少哪些层（如果有缺少的话），然后从登记站下载这些层。

登记站最多使用三条信息来标识特定的镜像：

名称

镜像名称可以简单也可以复杂，具体取决于存储镜像的登记站：nginx、servers/nginx或example.com/servers/nginx都是镜像名称。

标签

标签通常是指安装的软件版本（v1.2.3），尽管标签实际上只是任意字符串。标签latest和stable通常分别用于表示“最新版本”和“最新生产就绪版本”。

摘要

有时必须拉取一个非常具体的镜像版本。由于标签是可变的，因此不能保证在任何给定的时间，标签都确切地引用软件的此特定版本。因此，登记站支持通过摘要（镜像层信息的SHA-256或SHA-512摘要）获取镜像。

在本书中，我们将看到使用前面三条信息引用的镜像。组合这些的标准格式是name:tag@digest，其中只有name是必需的。因此，example.com/servers/nginx:latest表示“给我名为example.com/servers/nginx，标签为latest的映像”，而

```
example.com/my/app@sha256:
a428de44a9059feee59237a5881c2d2cffa93757d99026156e4ea544577ab7f3
```

表示“给我与此处的摘要完全一致的example.com/my/app”。

虽然关于镜像和容器还有很多知识需要学习，但是我们现在已经有足够的知识来继续下一个重要的主题了。下面我们将探索调度器和Kubernetes。

1.1.2 调度器和Kubernetes

在上一节中，我们看到了容器如何封装各个程序及其所需的环境。容器可以在工作站上本地执行，也可以在服务器上远程执行。

随着开发人员开始将应用程序打包到容器中，运营人员也开始使用容器作为部署的工件，于是出现了一组新的问题。如何最好地执行大量容器呢？如何最好地促进一个需要大量容器协同工作的微服务架构呢？如何明智地共享对网络连接存储、负载均衡器和网关等的访问呢？如何设法将配置信息注入许多容器中呢？也许最重要的是，如何管理内存、CPU、网络带宽和存储空间等资源呢？

甚至更进一步，人们（基于他们对虚拟机的经验）开始询问如何管理跨多个主机分发容器，以及在合理使用资源的同时公平地分配负载呢。即，如何在运行尽可能多的容器的同时运行尽可能少的主机呢？

2015年，时机已经成熟：Docker容器正在向企业进军。显然，此时需要一种工具来管理跨主机的容器调度和资源管理。多种技术相继登场：Mesos引入了Marathon；Docker创建了Swarm；Hashicorp发布了Nomad；Google创建了其内部Borg平台的开源兄弟，并将这项技术命名为Kubernetes（希腊语中的船长一词）。

所有这些项目都提供了一个集群容器管理系统的实现，该系统可以调度容器并将它们连接起来，以托管复杂的类似微服务的分布式应用程序。

每一个调度器都有其优点和缺点。但是Kubernetes引入了两个概念，使它与众不同：声明性基础设施（declarative infrastructure）和协调循环（reconciliation loop）。

声明性基础设施

考虑部署容器的情况。我们可以这样处理部署容器的过程：创建容器；打开一个端口让它监听，然后在文件系统的这个特定位置附加一些存储；等待所有的东西被初始化；测试它，看看容器是否准备好了；将其标记为可用。

在这种方法中，我们通过关注设置容器的流程来按过程进行思考。但Kubernetes的设计是我们以声明的方式思考。我们告诉调度器（Kubernetes）我们想要的状态是什么，Kubernetes负责将声明性语句转换成它自己的内部过程。

在Kubernetes上安装一个容器更像是在说，“我希望这个容器在这个端口上运行，使用一定量的CPU并在文件系统的这个位置上安装一些存储”。Kubernetes在后台工作，根据我们对所需内容的声明来连接所有内容。

协调循环

Kubernetes是如何在幕后完成这一切的？当我们按过程看问题时，会有一定的运作顺序。Kubernetes是怎么知道顺序的呢？这就是协调循环思想的用武之地。

在协调循环中，调度程序说：“这是用户所需的状态。以下是当前状态。它们不一样，所以我将采取步骤来协调它们。”用户希望为容器提供存储空间。当前没有附加存储。因此Kubernetes创建了一个存储单元并将其附加到容器中。容器需要公共网络地址。现在也不存在。所以一个新的地址被附加到容器上。Kubernetes中的不同子系统开展工作以实现用户对所需状态的整体声明的各个部分。

最终，Kubernetes要么成功地创建了用户想要的环境，要么得出了无法实现用户需求的结论。同时，用户只能被动地观察Kubernetes集群并等待它成功完成或将安装标记为失败。

从容器到pod、服务、部署等

前面的例子虽然简洁，但有点误导。Kubernetes不一定把容器当作工作单元。相反，Kubernetes引入了一个更高层次的抽象，称为pod。pod是描述离散工作单元的抽象信封。pod描述的不仅是一个容器，它还描述一个或多个容器（以及它们的配置和需求），这些容器结合在一起执行一个工作单元：

```
apiVersion: v1 ❶
kind: Pod
metadata:
    name: example-pod
spec:
    containers: ❷
    - image: "nginx:latest"
      name: example-nginx
```

❶ 前两行定义了Kubernetes类型（v1 Pod）。

❷ 一个pod可以有一个或多个容器。

一般一个pod只有一个容器。但有时有一些用于为主容器做预配置并在主容器联机之前退出的容器，这些被称为初始化容器（init container）。另外有一些与主容器同时运行并提供辅助服务的容器，这些被称为边车容器（sidecar container）。这些都被认为是同一个pod的一部分。

 在前面的代码中，我们编写了Kubernetes Pod资源的定义。当用YAML或JSON表示时，这些定义称为清单（manifest）。清单可以包含一个或多个Kubernetes资源（资源也称为对象或资源定义）。每个资源都与一种Kubernetes类型相关联，例如Pod或Deployment。在本书中，我们通常使用“资源”一词，因为“对象”这个词还有其他含义：YAML将对象定义为一个命名的键/值结构。

Pod描述容器所需的配置（如网络端口或文件系统装载点）。Kubernetes中的配置信息可以存储在ConfigMap中，对于敏感信息，可以存储在Secret中。Pod的定义可以将这些ConfigMap和Secret与每个容器中的环境变量或文件联系起来。当Kubernetes看到这些关系时，它将尝试附加和配置Pod定义中描述的配置数据：

```
apiVersion: v1 ❶
kind: ConfigMap
metadata:
    name: configuration-data
data: ❷
    backgroundColor: blue
    title: Learning Helm
```

❶ 在本例中，我们声明了一个v1 ConfigMap对象。

❷ 在data内部，我们声明了一些任意的名/值对。

Secret在结构上类似于ConfigMap，只是data部分中的值必须是Base64编码的。

Pod使用卷（volume）链接到配置对象（如ConfigMap或Secret）。在本例中，我们采用前面的Pod示例并附加上述Secret：

```
apiVersion: v1
kind: Pod
metadata:
    name: example-pod
spec:
    volumes: ❶
    - name: my-configuration
      configMap:
        name: configuration-data ❷
    containers:
    - image: "nginx:latest"
      name: example-nginx
      env: ❸
        - name: BACKGROUND_COLOR ❹
          valueFrom:
            configMapKeyRef:
                name: configuration-data ❺
                key: backgroundColor ❻
```

❶ volumes部分告诉Kubernetes这个pod需要哪些存储源。

❷ configuration-data是我们在前面的示例中创建的ConfigMap的名称。

❸ env部分将环境变量注入容器中。

❹ 环境变量将在容器内命名为BACKGROUND_COLOR。

❺ 这是它将使用的ConfigMap的名称。如果要将此映射用作文件系统卷，该映射必须在volumes中。

❻ 这是ConfigMap的data部分中的键的名称。

pod是可运行工作单元的“原始”描述，容器是pod的一部分。但是Kubernetes引入了更高阶的概念。

考虑一个Web应用程序。我们可能不想只运行此Web应用程序的一个实例。如果我们只运行了一个实例，且它失败了，那么网站就会崩溃。如果我们想升级它，必须想办法在不破坏整个网站的情况下进行升级。因此，Kubernetes引入了部署（Deployment）的概念。Deployment将应用程序描述为相同pod的集合。Deployment由一些顶级配置数据以及构建副本pod的模板组成。

通过Deployment，我们可以告诉Kubernetes使用单个pod创建应用程序。然后我们可以扩大到使用5个pod。再减少到3个pod。我们可以附加一个Horizontal PodAutoscaler（另一种Kubernetes类型）并配置它来根据资源使用情况扩展pod。当升级应用程序时，Deployment可以采用各种策略来增量升级单个pod，而不必关闭整个应用程序：

```
apiVersion: apps/v1 ❶
kind: Deployment
metadata:
    name: example-deployment
    labels:
        app: my-deployment
spec:
    replicas: 3 ❷
    selector:
        matchLabels:
            app: my-deployment
    template: ❸
        metadata:
            labels:
                app: my-deployment
        spec:
            containers:
            - image: "nginx:latest"
              name: example-nginx
```

❶ 这是一个apps/v1 Deployment对象。

❷ 在spec中，我们要求提供以下template的三个副本。

❸ template指定每个副本pod的规格。

当涉及将Kubernetes应用程序附加到网络上的其他东西时，Kubernetes提供了服务（Service）的定义。Service是一种持久的网络资源（有点像静态IP），即使连接到它的一个或多个pod消失，它也会持久存在。这样，Kubernetes Pod可以来去自如，而网络层可以继续将流量路由到同一Service端点。虽然Service是一个抽象的Kubernetes概念，但在幕后它可以实现为从路由规则到外部负载均衡器的任何东西：

```
apiVersion: v1 ❶
kind: Service
metadata:
  name: example-service
spec:
  selector:
    app: my-deployment ❷
  ports:
    - protocol: TCP ❸
      port: 80
      targetPort: 8080
```

❶ 类型是v1 Service。

❷ 此Service将通过app:my-deployment标签路由到pod。

❸ 此Service的80端口的TCP流量将路由到与app:my-deployment标签匹配的pod上的8080端口。

上述Service将把流量路由到我们之前创建的Deployment。

我们已经介绍了几种类型的Kubernetes。还有另外几十种，但到目前为止最常用的是Pod、Deployment、ConfigMap、Secret和Service。在下一章中，我们将更直接地介绍这些概念。但现在，我们可以介绍Helm了。

1.2 Helm的目标

到目前为止，我们关注的是更广泛的云原生生态系统以及Kubernetes在该生态系统中的角色。在本节中，我们将把焦点转移到Helm。

在上一节中，我们看到了几个不同的Kubernetes资源：Pod、ConfigMap、Deployment和Service。每一个资源都扮演着一些独立的角色。但是一个应用程序通常需要不止一个资源。

例如，WordPress CMS系统可以在Kubernetes内部运行。但通常情况下，它至少需要一个Deployment（用于WordPress服务器）、一个ConfigMap（用于配置）、一个Secret（用于保存密码）、几个Service对象、一个StatefulSet（用于运行数据库）和几个基于角色的访问控制（RBAC）规则。Kubernetes对基本WordPress站点的描述已经超过了数千行YAML。Helm的核心思想是，所有这些对象都可以打包在一起进行安装、更新和删除。

编写Helm有三个主要目标：

1. 轻松地实现从“从零到Kubernetes”；

2. 提供与操作系统类似的软件包管理系统；

3. 强调将应用程序部署到Kubernetes的安全性和可配置性。

接下来一一介绍，然后看看Helm用法的另一个方面：它如何参与生命周期管理。

1.2.1 从零到Kubernetes

Helm项目始于2015年，也就是KubeCon创始前几个月。Kubernetes过去很难设置，通常需要新用户编译Kubernetes源代码，然后使用一些shell脚本来运行Kubernetes。一旦集群启动，新用户就需要从头开始编写YAML。

我们希望颠倒学习周期：不要求用户从基本示例开始，尝试构建自己的应用程序，而是希望为用户提供现成的生产就绪示例。用户可以安装这些示例，查看它们的实际操作，然后了解Kubernetes是如何工作的。

这曾经是（至今仍然是）我们编写Helm的第一要务：让Kubernetes更易于使用。在我们看来，拥有现有Kubernetes集群的新Helm用户应该能够在5分钟甚至更短的时间内完成从下载到安装应用程序的步骤。

Helm不仅仅是一个学习工具，它还是一个软件包管理器。

1.2.2 软件包管理

Kubernetes就像一个操作系统。在它的基础上，操作系统提供了执行程序的环境。它提供了存储、执行和监视程序生命周期所需的工具。

它不执行程序，而是执行容器。但与操作系统类似，它提供了存储、执行和监视这些容器所需的工具。

大多数操作系统都由软件包管理器支持。软件包管理器的任务是使在操作系统上查找、安装、升级和删除程序变得容易。软件包管理器提供了将程序捆绑到可安装的应用程序中的语义，并提供了存储和检索软件包以及安装和管理软件包的方案。

当我们将Kubernetes设想为一个操作系统时，很快就看到了对Kubernetes软件包管理器的需求。从第一次提交到Helm源代码库，我们就一直将软件包管理比喻应用到Helm上：

· Helm提供软件包存储库和搜索功能，以查找可用的Kubernetes应用程序。

· Helm拥有我们熟悉的安装、升级和删除命令。

· Helm定义了在安装软件包之前配置软件包的方法。

· Helm提供了查看已安装内容和配置方式的工具。

我们最初借鉴Homebrew（macOS的软件包管理器）和Apt（Debian的软件包管理器）对Helm进行设计。但随着Helm的成熟，我们已经尽可能多地借鉴不同的软件包管理器。

典型的操作系统和Kubernetes之间有一些区别。其中之一是Kubernetes支持运行同一应用程序的多个实例。虽然我可能只在工作站上安装了一次数据库MariaDB，但是Kubernetes集群可能会运行数十、数百甚至数千个MariaDB安装，其中每个安装都有不同的配置，甚至是不同的版本。

另一个在典型操作系统中很少见的概念是命名空间（namespace），但它是Kubernetes的核心。在Kubernetes中，命名空间是一种任意的分组机制，它定义了命名空间内部和外部事物之间的边界。使用命名空间组织资源有许多不同的方法，但通常它们被用作附加安全性的固定装置。例如，可能只有特定用户才能访问命名空间内的资源。

这些只是Kubernetes与传统操作系统不同的几个方面。这些差异和其他差异对Helm的设计提出了挑战。我们不得不构建Helm来充分利用这些差异，但同时又不放弃关于我们的软件包管理比喻。

例如，Helm安装命令不仅需要软件包的名称，还需要用户提供的名称，通过用户提供的该名称可以引用该软件包的已安装版本。在下一章中，我们将看到这样的例子。

同样，Helm中的操作也是区分命名空间的。可以将同一个应用程序安装到两个不同的命名空间中，Helm提供了管理这些不同应用程序的实例的工具。

不过，最终，Helm仍然稳居软件包管理类工具之列。

1.2.3 安全性、可重用性和可配置性

我们设计Helm的第三个目标是关注管理集群中应用程序的三个“必备条件”：

1. 安全性

2. 可重用性

3. 可配置性

简而言之，我们想让Helm充分了解这些原则，以便Helm用户对他们使用的软件包有信心。用户应该验证（verify）一个软件包是否有可靠的来源（并且没有被篡改），多次重用（reuse）同一个软件包，并配置（configure）该软件包以满足他们的需求。

尽管Helm的开发人员可以直接控制前两个设计目标（易用和软件包管理），但第三个目标不一样：Helm只能为软件包作者提供正确的工具，以供这些创建者选择来实现这三个“必备条件”。

安全性

安全性是一个广泛的范畴。不过此处我们指的是这样一种想法，当用户检查某个软件包时，用户有能力验证软件包的某些内容：

- 软件包来源可靠。
- 拉取软件包的网络连接是安全的。
- 软件包未被篡改。
- 软件包易于检查，因此用户可以看到它的功能。
- 用户可以看到软件包的配置，以及不同的输入如何影响软件包的输出。

在本书中，特别是在第6章中，我们将更详细地介绍安全性。但我们相信Helm已经提供了这5种能力。

Helm提供了一个provenance特性来验证软件包的来源、作者和完整性。Helm支持安全套接字层/传输层安全性（SSL/TLS），用于通过网络安全地发送数据。Helm提供了dry-run、template和linting（试运行、模板和代码校验）命令来检查包及其可能的排列。

可重用性

软件包管理的一个优点是它能够重复和可预测地安装相同的东西。在Helm中，这个想法得到了扩展：我们甚至可能希望将相同的东西（重复地和可预测地）安装到同一个集群中，甚至是集群中的同一个命名空间中。

Helm chart是可重用性的关键。chart提供了一种模式来生成相同的Kubernetes清单。但是chart也允许用户提供额外的配置（见第2章）。所以Helm提供了存储配置的模式，使得chart加上它的配置的组合甚至可以重复进行。

通过这种方式，Helm鼓励Kubernetes用户将他们的YAML打包成chart，以便这些描述可以重用。

在Linux世界中，每个Linux发行版都有自己的软件包管理器和存储库。Kubernetes的世界并非如此。Helm的构造使得所有Kubernetes发行版都可以共享相同的软件包管理器，并且（除了极少数例外）也可以共享相同的软件包。当两个不同的Kubernetes发行版之间存在差异时，chart可以使用模板（见第5章）和配置来适应这种情况。

可配置性

Helm提供了一个Helm chart模式，并提供了一些额外的配置。例如，我可能安装了一个带有Helm的网站，但希望（在安装时）设置该网站的名称。Helm提供了在安装时配置软件包的工具，以及在升级期间重新配置安装的工具。但要注意的一点是按照顺序。

Helm是一个软件包管理器。另一类软件处理配置管理。这类软件以Puppet、Ansible和Chef为代表，主要关注给定的软件（通常已打包）是如何针对其宿主环境进行特定配置的。它的职责是随时间的变化管理配置更改。

Helm并未被设计成配置管理工具，尽管软件包管理和配置管理之间有一些重叠。

软件包管理通常局限于实现三个动词：安装、升级和删除。配置管理是一个更高层次的概念，旨在随着时间的推移管理一个或多个应用程序。这有时被称为“第二天行动”（day-two ops）。

虽然Helm并不是一个配置管理工具，但有时会被用作配置管理工具。组织不仅依赖Helm来安装、升级和删除应用程序，还依赖Helm来跟踪随时间的变化、跟踪配置以及确定整个应用程序是否正在运行。Helm可以这样扩展，但是如果你想要一个强大的配置管理解决方案，可能需要利用Helm生态系统中的其他工具。许多工具（如Helmfile、Flux和Reckoner）都在更大的配置管理事项中填补了细节。

 Helm社区创建了大量与Helm互操作或增强Helm的工具。Helm项目在官方文档（https://oreil.ly/hOqca）中保留了这些工具的清单。

在Helm chart中你会注意到的一个常见主题是，配置选项通常是设置好的，这样你就可以将同一个chart的最小版本发布到你的开发环境中，或者使用不同的配置选项将一个复杂的版本发布到你的生产环境中。

1.3 Helm架构

在本节中，我们将简要介绍Helm的高级架构。除了对云原生Kubernetes应用程序和软件包管理的概念性讨论外，本节还为第2章铺平了道路，在第2章中，我们将深入讨论如何使用Helm。

1.3.1 Kubernetes资源

我们已经看过了几种Kubernetes资源：Pod、ConfigMap、Deployment和Service。Kubernetes还提供了几十种。你甚至可以使用自定义资源定义（CRD）来定义自己的自定义资源类型。主要的Kubernetes文档提供了关于每种类型的可访问指南和详细的API文档。

在本书中，我们将使用许多不同的Kubernetes资源类型。当我们在上下文中讨论它们时，你可能会发现在浏览新的资源定义时浏览主要的Kubernetes文档是有益的。

如前所述，资源定义是声明性的。用户为Kubernetes描述所需的资源状态。例如，你可以将我们在本章前面创建的Pod定义理解为“我希望Kubernetes为我制作一个具有这些特性的Pod”的语句，这取决于Kubernetes如何根据你的规范配置和运行Pod。

所有Kubernetes资源定义共享一个共同的元素子集。下面的清单使用Deployment来说明资源定义的主要结构元素：

```
apiVersion: apps/v1 ❶
kind: Deployment ❷
metadata: ❸
    name: example-deployment ❹
    labels: ❺
        some-name: some-value
    annotations: ❻
        some-name: some-value
# resource-specific YAML
```

❶ 此资源的API系列和版本。

❷ 资源的种类。结合apiVersion，得到资源类型。

❸ metadata（元数据）部分包含关于资源的顶级数据。

❹ 几乎每种资源类型都需要一个name（名称）。

❺ 标签（labels）用于为你的资源提供Kubernetes可查询的“句柄”。

❻ 注解（annotations）为作者提供了一种将自己的键和值附加到资源的方法。

特别要注意的是，Kubernetes中的资源类型由三块信息组成：

API组（或家族）

一些基本资源类型（如Pod和ConfigMap）忽略了此名称。

API版本

表示为v，后跟主要版本和可选的稳定性标记。例如，v1是一个稳定的“version 1”，而v1alpha表示一个不稳定的“version 1 alpha 1”。

资源类型

API组中特定资源的（大写）名称。

 虽然完整的资源类型名称类似于apps/v1 Deployment或v1 Pod（对于核心类型），但Kubernetes用户在谈论或编写已知类型时通常会忽略组和版本。例如，在本书中，我们只是写Deployment，而不是apps/v1 Deployment。在指定确切的版本或讨论CRD中定义的资源类型时，将使用完全限定名。

因此，apps/v1 Deployment表示API组“apps”有一个名为“Deployment”的“version 1”（稳定）资源类型。

Kubernetes支持两种主要格式来声明所需的资源：JSON和YAML。严格地说，YAML是JSON的超集（superset）。所有JSON文档都是有效的YAML，但是YAML添加了许多附加特性。

在这本书中，我们坚持使用YAML格式。它更容易读写，几乎所有Helm用户都选择YAML而不是JSON。但是，如果你的偏好与此不同，Kubernetes和Helm也都支持纯JSON。

早些时候，我们引入了术语清单。清单只是序列化为JSON或YAML格式的Kubernetes资源。我们可以将前面的Pod、ConfigMap、Deployment和Service示例分别称为一个Kubernetes清单，因为它们是用YAML表示的资源。

1.3.2 chart

我们已经在本章中讨论了Helm软件包。在Helm的词汇表中，有一个软件包叫作chart（航海图）。这个名字是对Kubernetes（在希腊语中是“船长”的意思）和Helm（舵，船舶的操纵机构）的航海性质的一种诠释。chart描绘了Kubernetes应用程序的安装方式。

chart是遵循chart规范的一组文件和目录，用于描述要安装到Kubernetes中的资源。第4章详细解释了chart结构，但这里我们将介绍一些高级概念。

chart包含一个名为chart.yaml的文件，它描述了该chart，包含有关chart版本、chart名称和说明以及chart作者的信息。

chart也包含模板。这些也是Kubernetes清单，可能用模板指令进行注解。详见第5章。

chart也可以包含提供默认配置的values.yaml文件。此文件包含安装和升级期间可以覆盖的参数。

这些是你将在Helm chart中找到的基本文件，更多文件参见第4章。不过，当你看到一个Helm chart时，它可能是以未打包或打包的形式呈现的。

一个未打包的Helm chart只是一个目录。在里面，它会有一个Chart.yaml、一个values.yaml、一个templates/目录等。打包的Helm chart包含与未打包的Helm chart相同的信息，但它被tar打包并用gzip压缩到单个文件中。

未打包的chart由一个带有chart名称的目录表示。例如，名为mychart的chart将被解压到名为mychart/的目录中。相比之下，打包的chart有chart的名称和版本，以及tgz后缀：mychart-1.2.3.tgz。

chart存储在chart存储库中，我们将在第7章中介绍它们。Helm知道如何从存储库下载和安装chart。

1.3.3 资源、安装和发布

为了将本节中介绍的术语联系起来，当某个Helm chart被安装到Kubernetes中时，会发生以下情况：

1. Helm读取chart（必要时下载）。

2. 它将值发送到模板中，生成Kubernetes清单。

3. 清单被送到Kubernetes。

4. Kubernetes在集群内创建请求的资源。

当一个Helm chart被安装时，Helm将根据需要生成尽可能多的资源定义。有些chart可能创造一个或两个定义，其他的可能创造数百个定义。当Kubernetes收到这些定义时，将为它们创建资源。

Helm chart可能有许多资源定义。Kubernetes认为每个资源定义都是独立的。但在Helm看来，chart定义的所有资源都是相关的。例如，我的WordPress应用程序可能有一个Deployment、一个ConfigMap、一个Service等。但它们都是一个chart的一部分。安装它们时，它们都是同一个安装的一部分。同一chart可以安装多次（每次使用不同的名称）。因此，我可能拥有相同chart的多个安装，就像我可能拥有相同Kubernetes资源类型的多个资源一样。

这带来了最后一个术语。一旦安装了WordPress chart，我们就拥有了一个该chart的安装。然后我们用helm upgrade来升级该chart。现在，这个安装有两个版本。每次使用Helm修改安装时，都会创建一个新的安装版本。

当安装新版本的WordPress时，会创建一个版本。但是，当我们仅仅更改安装的配置或者回滚安装时，也会创建一个版本。这是Helm的一个重要特征，我们将在第7章中再次看到。

1.3.4 关于Helm 2的简要说明

熟悉Helm 2的人可能会注意到本书中缺少某些概念：Tiller或gRPC。这些东西已从Helm 3中删除，而本书的主题是讲述Helm 3。此外，本书着重讲述第2版Helm chart。因为Helm chart版本与Helm版本的递增是不一致的。所以Helm v2使用Helm Charts v1，Helm v3使用Helm Charts v2。它们在一些重要的方面与第1版的Helm Charts不同，最显著的是在声明依赖项的方式上。Helm 2和Helm Charts v1被认为已弃用。

1.4 结论

本章的内容是其他章节的基础。只有使Kubernetes对新用户和长期使用Helm的运营团队和SRE都更有用，Helm才能成功。本书的其余部分致力于（用大量的例子）解释如何最大限度地发挥Helm的作用，以及如何安全地、习惯地做到这一点。

第2章

使用Helm

Helm提供了一个名为helm的命令行工具，它提供了使用Helm chart所需的所有特性。在本章中，我们将探索helm客户端的主要特性，并了解Helm如何与Kubernetes交互。

我们将从了解如何安装和配置Helm开始，并通过Helm中的主要命令组进行工作。然后我们将介绍如何查找和学习软件包，以及如何安装、升级和删除它们。

2.1 安装和配置Helm客户端

Helm提供了一个能够执行所有主要Helm任务的命令行客户端：helm。虽然还有许多其他工具可以使用Helm chart，但这是由Helm核心维护人员维护的官方通用工具，也是本章和下一章的主题。

helm客户端是用Go语言编写的。与Python、JavaScript和Ruby不同，Go是一种编译语言。一旦编译了Go程序，就不需要任何Go工具来运行或以其他方式处理二进制文件。

因此，我们将首先介绍下载和安装静态二进制文件，然后简要介绍获取和编译Go源代码的过程。

2.1.1 安装预构建的二进制文件

每次Helm维护人员发布新版本的helm时，项目都会为许多常见的操作系统和架构提供新的helm签名二进制版本。在撰写本书时，可以在64位Intel/AMD、ARM、s390和PPC等架构上为Linux、Windows和macOS操作系统提供预构建版本的Helm。这意味着你可以在从树莓派到超级计算机的任何机器上运行Helm。

Helm发布的最终列表在Helm发布页面（https://oreil.ly/L_My5）。发布页面将显示按时间顺序排列的发布列表，最新版本位于顶部。

使用软件包管理器安装

许多操作系统软件包管理器，包括适用于macOS的Homebrew、适用于Linux的Snap和适用于Windows的Chocolate，都可以安装Helm。我们是软件包管理的忠实爱好者。软件包管理器使安装、更新和删除软件变得容易，因此我们鼓励你让你的操作系统软件包管理器安装

Helm。但通常明智的做法是检查所选软件包管理器中的版本是否与当前在Helm站点上标记为stable的版本相同。

关于Helm版本号

直到2020年11月，两个不同的主要版本的Helm都被积极维护。目前稳定的是Helm 3。当你访问Helm下载页面时，可能会看到两个版本都可以下载。因为版本是按时间顺序列出的，所以一个Helm 2版本甚至有可能比最新的Helm 3版本还新。但你应该使用Helm 3。

Helm遵循一种称为语义版本管理（SemVer）（https://semver.org）的版本管理约定。在语义版本管理中，版本号表示你能期望在发行版中获得的内容。因为Helm遵循该规范，所以用户只需仔细阅读版本号就可以从发行版中获得特定的东西。

语义版本的核心是三个数字部分和一个可选的稳定性标记（stability marker）（用于alpha、beta和候选版本）。以下是一些示例：

- v1.0.0
- v3.3.2
- v2.4.22-alpha.2

我们先来讨论数字部分。

我们经常将这种格式概括为X.Y.Z，其中X是主版本（major version），Y是次要版本（minor version），Z是补丁版本（patch release）：

- 主版本号往往很少增加。它表明对Helm进行了重大更改，其中一些更改可能会破坏与以前版本的兼容性。Helm 2和Helm 3之间的差别很大，需要在两个版本之间进行迁移。

- 次要版本号表示新增功能。3.2.0和3.3.0之间的区别可能是添加了一些小的新特性。但是，版本之间没有突破性的修改（breaking

change）。有一个警告：一个安全修复可能需要一个突破性的修改，但在这种情况下，我们会强调指出。

· 补丁版本号表示在这个版本和上一个版本之间只进行了向后兼容的bug修复。建议始终保持最新的补丁版本。

当你看到一个版本在版本号后面附加了一个稳定性标记（如alpha.1、beta.4或rc.2）时，这意味着该版本被认为是一个预发布版本，还没有准备好用于主流生产。特别是，Helm经常在主要或次要更新之前发布候选版本。在发布最终版本之前，社区有机会就稳定性、兼容性和新特性向我们提供一些反馈。

记住这一点，我们准备继续实际的安装。

下载二进制文件

从存储库安装Helm最简单的方法就是进入发布页面（https://oreil.ly/L_My5）并下载最新的Helm 3版本。

在Windows上，下载文件是包含README.md文本文件、LICENSE文本文件和helm.exe的ZIP压缩包。

在macOS和Linux上，下载的是一个gzip tar压缩包（.tar.gz），它可以用tar-zxf命令提取。与Windows版本一样，它将包含README.md文本文件、LICENSE文本文件和helm二进制文件。如果你使用的是Windows Subsystem for Linux（WSL），那么应该将Linux AMD64版本安装到WSL实例中。

无论你使用哪种操作系统，二进制文件都是运行Helm所需的唯一文件，你可以将其放置在系统中你喜欢的任何位置。它应该被预先标记为可执行文件，但是在类似UNIX的环境中，在很少的情况下，你可能还需要运行chmod helm+x命令来将Helm设置为可执行文件。

 当使用Homebrew（macOS）、Snap（Linux）或Chocooley（Windows）等软件包管理器进行安装时，helm将安装在标准位置，并能通过命令行立即使用。

安装helm后，就应该能够运行helm help命令并查看Helm帮助文本。

使用get脚本安装

在macOS和Linux上，你可能更喜欢运行shell脚本，该脚本将决定要安装哪个版本的Helm，并自动为你完成安装。

以这种方式安装的命令序列通常如下所示：

```
$ curl -fsSL -o get_helm.sh \
https://raw.githubusercontent.com/helm/helm/master/scripts/get-helm-3
$ chmod 700 get_helm.sh
$ ./get_helm.sh
```

前面的命令获取最新版本的get_helm.sh脚本，然后使用该脚本查找并安装最新版本的Helm 3。

对于自动安装Helm的系统，例如持续集成（CI）系统，如果需要始终使用最新的Helm版本，我们建议使用此方法安装。

2.1.2 从源代码构建的指南

除非你已经熟悉Go开发，否则从源代码构建Helm可能是一项艰巨的任务。你将需要make命令的一个版本。因为Makefile样式的构建脚本不遵循单一的标准，所以并非make的所有版本都能用于构建Helm。Gnu Make是Linux和Mac上最常用的一个，也是大多数Helm核心开发人员最常用的，所以这是一个安全的选择。你还需要gcc编译器和整个Go工具链。

除此之外，Helm还需要一些辅助工具。幸运的是，当你第一次运行make时，它将尝试安装缺少的任何其他工具。

虽然git工具和kubectl命令不是绝对必要的，但是你可能还需要它们。git工具将允许你直接使用Helm源代码库，而不是下载源代码包。kubectl用来与Kubernetes集群进行交互。虽然这对于构建Helm不是必需的，但是在检查Helm是否执行了你希望它执行的操作时，它确实是必需的。

安装并配置了这些工具后，就可以简单地更改包含Helm源代码的文件夹中的目录（包含README.md和Makefile文件）并运行make build。第一次运行此命令时，至少需要几分钟。构建系统必须获取大量依赖项，包括许多Kubernetes源代码，并对其进行编译。

 第一次编译Helm可能会让人望而生畏，特别是如果你对Go编程语言还不熟悉的话。Kubernetes是一个复杂的平台，因此Helm的源代码量很大，很难构建。一个新的环境至少要花一两个小时来准备安装Helm。

要验证Helm是否正常工作（特别是在修改了源代码的情况下），可以运行make test。这将构建代码，运行各种检查器（checker）和代码校验器（linter），然后运行Helm的单元测试。如果你计划对Helm进行任何更改，在Helm的核心维护人员查看你请求的更改之前，这个命令必须通过。

Helm编译完成时，它将与源代码一起位于名为bin/的子目录中。它不会自动添加到可执行路径中，因此要执行刚刚构建的版本，可能需要指定相对或精确的路径（例如，./bin/helm或$GOPATH/src/helm.sh/helm/bin/helm）。

如果命令helm version正确执行，你可以确信已经正确编译了Helm。

从这里，你可以按照详细的开发人员指南（https://oreil.ly/X9-Ii）了解更多信息。一如既往，如果遇到问题，Kubernetes Slack服务器上的helm-users频道是寻求帮助的好地方。

2.1.3 使用Kubernetes集群

Helm直接与kubernetes API服务器交互。因此，Helm需要能够连接到Kubernetes集群。Helm试图通过读取kubectl（主Kubernetes命令行客户端）使用的相同配置文件来自动完成这一任务。

Helm将尝试通过读取环境变量$KUBECONFIG来查找此信息。如果没有设置此环境变量，它将在与kubectl相同的默认位置中查找（例如，在UNIX、Linux和macOS上，$HOME/.kube/config）。

你还可以使用环境变量（HELM_KUBECONTEXT）和命令行标志（--kube-context）覆盖这些设置。运行helm help可以看到环境变量和标志的列表。

Helm维护人员建议使用kubectl来管理Kubernetes凭据，让Helm只自动检测这些设置。如果你还没有安装kubectl，最好从Kubernetes的官方安装文档（https://oreil.ly/pHZIh）开始。

2.1.4 Helm入门

无论你是从源代码构建Helm，还是使用上述方法之一安装Helm，此时你的系统上都应该有helm命令了。从这里开始，我们将假设Helm可以通过helm命令执行（而不是上一节讨论的完整路径或相对路径）。

在下面的内容中，我们将了解从Helm开始的最常见的工作流程：

1. 添加chart存储库。

2. 查找要安装的chart。

3. 安装Helm chart。

4. 查看安装内容列表。

5. 升级安装。

6. 删除安装。

在下一章中，我们将深入了解Helm的一些附加功能，并在此过程中进一步了解Helm的工作原理。

2.2 添加chart存储库

chart存储库本身就是一个宏大的主题，在第7章中我们将详细研究它们。但是任何使用Helm的人都必须知道一些关于chart存储库的基本知识。

Helm chart是一个单独的软件包，可以安装到Kubernetes集群中。在chart开发过程中，通常只使用存储在本地文件系统中的chart。

但当谈到共享chart时，Helm描述了一种标准格式，用于索引和共享有关Helm chart的信息。Helm chart存储库只是一组可通过网络访问、符合Helm规范的文件。

Helm 3引入了一个实验特性，用于将Helm chart存储在不同类型的存储库中：Open Container Initiative（OCI）注册中心（有时称为Docker注册中心）。在这个后端，一个Helm chart可以存储在Docker镜像旁边。虽然这个功能还没有得到广泛的支持，但它可能会成为Helm软件包存储的未来。详见第7章。

在因特网上有许多（可能数千个）chart存储库。找到流行存储库的最简单方法是使用Web浏览器导航到Artifact Hub（https://artifacthub.io）。在那里你会发现数以千计的Helm chart，每一个都托管在一个适当的存储库中。

首先，我们将安装流行的Drupal内容管理系统（https://www.drupal.org）。这是一个很好的示例chart，因为它使用了Kubernetes的许多类型，包括Deployment、Service、Ingress和ConfigMap（部署、服务、入口和配置映射）。

Helm 2默认安装了一个Helm存储库。stable chart存储库曾一度是生产就绪的Helm chart的官方来源。但我们意识到，将chart集中到一个存储库对一小群维护人员来说任务过于繁重，而对chart贡献者来说则令人沮丧。

因此在Helm 3中没有默认的存储库。鼓励用户使用Artifact Hub来查找他们要查找的内容，然后添加他们首选的存储库。

Drupal的Helm chart位于一个最完善的chart库中：Bitnami的官方Helm chart。你可以查看Artifact Hub的Drupal chart（https://oreil.ly/baxxf）条目以获取更多信息。


一些Bitnami开发人员是设计Helm存储库系统的核心贡献者。他们为建立Helm chart开发的最佳实践做出了贡献，并编写了许多应用最广泛的chart。

使用helm repo add命令添加Helm chart。几个Helm存储库命令被划分在helm repo命令组下：

```
$ helm repo add bitnami https://charts.bitnami.com/bitnami
"bitnami" has been added to your repositories
```

helm repo add命令将添加一个名为bitnami的存储库，该存储库指向URLhttps://charts.bitnami.com/bitnami。

现在，我们可以通过运行第二个repo命令来验证Bitnami存储库是否存在：

```
$ helm repo list
NAME    URL
bitnami https://charts.bitnami.com/bitnami
```

这个命令显示了为Helm安装的所有存储库。现在，我们只看到刚才添加的Bitnami存储库。

一旦我们添加了一个存储库，它的索引就将被本地缓存，直到下次更新它（见第7章）。我们现在可以做的一件重要的事情就是搜索存储库。

2.3 搜索chart存储库

尽管我们已经了解了Artifact Hub，知道Drupal chart存在于这个存储库中，但是从命令行搜索它仍然很有用。通常，搜索是一种有用的方法，不仅可以找到可以安装的chart，而且可以找到可用的版本。

首先，搜索Drupal chart：

```
$ helm search repo drupal
NAME            CHART VERSION   APP VERSION     DESCRIPTION
bitnami/drupal  7.0.0           9.0.0           One of the most versatile open...
```

我们对drupal这个词做了一个简单的搜索。Helm不仅会搜索包名，还会搜索标签和描述等其他字段。因此，我们可以搜索content（内容）并在那里看到Drupal，因为它是一个内容管理系统：

```
$ helm search repo content
NAME                    CHART VERSION   APP VERSION     DESCRIPTION
bitnami/drupal          7.0.0           9.0.0           One of the most versa...
bitnami/harbor          6.0.1           2.0.0           Harbor is an an open...
bitnami/joomla          7.1.18          3.9.19          PHP content managemen...
bitnami/mongodb         7.14.6          4.2.8           NoSQL document-orient...
bitnami/mongodb-sharded 1.4.2           4.2.8           NoSQL document-orient...
```

虽然Drupal是第一个结果，但请注意，还有许多其他chart，它们在描述性文本中的某个地方，包含单词content。

默认情况下，Helm尝试安装chart的最新稳定版本，但是你可以覆盖此行为并安装chart的特定版本。如此，我们不仅可以查看chart的摘要信息，还可以查看chart的确切版本：

```
$ helm search repo drupal --versions
NAME            CHART VERSION   APP VERSION     DESCRIPTION
bitnami/drupal  7.0.0           9.0.0           One of the most versatile op...
bitnami/drupal  6.2.22          8.9.0           One of the most versatile op...
bitnami/drupal  6.2.21          8.8.6           One of the most versatile op...
bitnami/drupal  6.2.20          8.8.5           One of the most versatile op...
bitnami/drupal  6.2.19          8.8.5           One of the most versatile op...
...
```

Drupal chart有几十个版本。前面的示例已被截断，仅显示前几个版本。

chart版本和应用程序版本

chart版本是Helm chart的版本。应用程序版本是打包在chart中的应用程序的版本。Helm使用chart版本来进行版本控制决策，比如哪个软件包是最新的。正如我们在前面的示例中看到的，多个chart版本可能包含相同的应用程序版本。

2.4 安装程序包

在下一章中，我们将深入探讨如何在Helm中安装软件包的细节。不过，在本节中，我们将了解安装Helm chart的基本机制。

在Helm中安装chart至少需要两条信息：安装的名称和要安装的chart。

回想一下，在上一章中，我们区分了安装（installation）和chart。这是安装和升级期间的一个重要区别。在操作系统软件包管理器中，我们可以请求它安装一个软件。但在一个操作系统上，我们需要多次安装同一个软件包的情况极为罕见。Kubernetes集群是不同的。在Kubernetes中说“我想为应用程序A安装一个MySQL数据库，为应用程序B安装第二个MySQL数据库”是完全有意义的。即使这两个数据库的版本完全相同，配置也完全相同，为了适当地管理应用程序，我们可能希望有两个实例运行。

因此，Helm需要一种方法来区分同一chart的不同实例。因此，chart的installation是chart的一个特定实例。一个chart可能有许多installation。当我们运行helm install命令时，我们需要给它指定一个installation名和chart名。所以最基本的installation命令如下：

```
$ helm install mysite bitnami/drupal
NAME: mysite
LAST DEPLOYED: Sun Jun 14 14:46:51 2020
NAMESPACE: default
STATUS: deployed
REVISION: 1
NOTES:
******************************************************************
*** PLEASE BE PATIENT: Drupal may take a few minutes to install ***
******************************************************************
1. Get the Drupal URL:

  You should be able to access your new Drupal installation through

  http://drupal.local/

2. Login with the following credentials

  echo Username: user
  echo Password: $(kubectl get secret --namespace default mysite-drupal...
```

前面的代码将创建bitnami/drupal chart的一个实例，并将这个实例命名为mysite。

在install命令运行时，它将返回大量信息，包括有关开始使用Drupal的面向用户的说明。

 在以后的helm install示例中，为了简洁起见，我们将省略返回的输出。但是，当使用Helm时，你将看到每个安装的输出。在下一章中，我们还将看到如何使用helm get命令再次查看该输出。

此时，集群中现在有一个名为mysite的实例。如果我们尝试重新运行前面的命令，就不会得到第二个实例。相反，我们会得到一个错误，因为名称mysite已经被使用：

```
$ helm install mysite bitnami/drupal
Error: cannot re-use a name that is still in use
```

需要进一步澄清。在Helm 2中，实例名是集群范围的。每个集群只能有一个名为mysite的实例。在Helm 3中，命名规则已经改变。现在实例名称的作用域是Kubernetes命名空间。我们可以安装两个名为mysite的实例，只要它们各自位于不同的命名空间中即可。

例如，以下内容在Helm 3中是完全合法的，尽管它会在Helm 2中引发致命错误：

```
$ kubectl create ns first
$ kubectl create ns second
$ helm install --namespace first mysite bitnami/drupal
$ helm install --namespace second mysite bitnami/drupal
```

这将在第一个命名空间中安装一个名为mysite的Drupal站点，在第二个命名空间中安装一个相同配置的名为mysite的实例。这一点一开始可能会让人困惑，但当我们将命名空间看作名称的前缀（prefix on a name）时，情况就更清楚了。从这个意义上说，我们有一个名为“first mysite”的网站和另一个名为“second mysite”的网站。

 在整个Helm中使用命名空间标志

使用命名空间和Helm时，可以使用--namespace或-n标志指定所需的命名空间。

2.4.1 安装时的配置

在前面的示例中，我们以几种不同的方式安装了相同的chart。在所有情况下，它们的配置都是相同的。虽然使用默认配置有时是不错的，但更多的时候我们希望将自己的配置传递给chart。

许多chart允许你提供配置值。如果我们看看Drupal的Artifact Hub页面（https://oreil.ly/baxxf），就会看到一长串可配置参数。例如，我们可以通过设置drupalUsername值来配置Drupal管理员账户的用户名。

 在下一章中，我们将学习如何使用helm命令获取这些信息。

有几种方法可以告诉Helm要配置哪些值。最好的方法是创建一个包含所有配置覆盖值的YAML文件。例如，我们可以创建一个文件来设置drupalUsername和drupalEmail的值：

```
drupalUsername: admin
drupalEmail: admin@example.com
```

现在我们有一个文件（按惯例命名为values.yaml），它拥有我们所有的配置。因为它在一个文件中，所以很容易复制相同的安装。你还可以将此文件签入版本控制系统，以跟踪随时间变化的值。Helm核心维护人员认为将配置值保存在YAML文件中是一种很好的做法。但是，请务必记住，如果配置文件中包含敏感信息（如口令或身份验证令牌），则应采取措施确保这些信息不会泄漏。

helm install和helm upgrade都提供了一个--values标志，它指向具有覆盖值的YAML文件：

```
$ helm install mysite bitnami/drupal --values values.yaml
NAME: mysite
LAST DEPLOYED: Sun Jun 14 14:56:15 2020
NAMESPACE: default
STATUS: deployed
REVISION: 1
NOTES:
**********************************************************************
*** PLEASE BE PATIENT: Drupal may take a few minutes to install ***
**********************************************************************

1. Get the Drupal URL:

  You should be able to access your new Drupal installation through

  http://drupal.local/

2. Login with the following credentials

  echo Username: admin
  echo Password: $(kubectl get secret --namespace default mysite-drupal -o js...
```

注意，在前面的输出中，Username（用户名）现在是admin而不是user。Helm的一个很好的特性是甚至可以使用你提供的值来更新帮助文本。

 可以多次指定--values标志。有些人使用此功能以便在一个文件中具有“公共”覆盖，而在另一个文件中具有特定覆盖。

第二个标志可用于向安装或升级添加单个参数。--set标志直接接受一个或多个值。它们不需要存储在YAML文件中：

```
$ helm install mysite bitnami/drupal --set drupalUsername=admin
```

这只设置了drupalUsername这一个参数。此标志使用简单的key=value格式。

配置参数可以结构化。也就是说，一个配置文件可以有多个部分。例如，Drupal chart具有特定于MariaDB数据库的配置。这些参数都分组到mariadb部分中。在前面的示例的基础上，我们可以覆盖MariaDB数据库名称，如下所示：

```
drupalUsername: admin
drupalEmail: admin@example.com
mariadb:
  db:
    name: "my-database"
```

当使用--set标志时，子部分要复杂一些。你将需要使用虚线表示法：--set mariadb.db.name=my-database。当设置多个值时，这可能会变得冗长。

一般来说，Helm核心维护人员建议将配置存储在values.yaml文件中（注意，文件名不一定是“values”），仅在绝对必要时才使用--set。通过这种方式，你可以很容易地访问在操作期间使用的值（并且可以随着时间的推移跟踪这些值），同时还可以缩短Helm命令。使用文件还意味着不必像在命令行上设置内容时那样转义那么多字符。

在继续介绍升级之前，我们先快速了解一条最有用的Helm命令。

2.5 列出你的安装

到目前为止，我们已经看到，Helm可以在同一个集群中安装很多东西，甚至可以在同一个chart中安装多个实例。对于集群上的多个用户，不同的人可能会在集群上的同一命名空间中安装所需内容。

helm list命令是一个简单的工具，可以帮助你查看安装并了解这些安装：

```
$ helm list
NAME    NAMESPACE  REVISION  UPDATED       STATUS    CHART          APP VERSION
mysite  default    1         2020-06-14... deployed  drupal-7.0.0   9.0.0
```

此命令将为你提供许多有用的信息，包括版本的名称和命名空间、当前版本号（在第1章中讨论过，在下一节中将进行更深入的讨论）、上次更新的时间、安装状态以及chart和应用程序的版本。

与其他命令一样，helm list是区分命名空间的。默认情况下，Helm将Kubernetes配置文件设置的命名空间作为默认命名空间（通常名为default）。前面，我们在命名空间first中安装了一个Drupal实例。可以在helm list__namespace first中看到这一点。

列出所有发行版时，一个有用的标志是--all namespaces，它将查询你有权访问的所有Kubernetes命名空间，并返回找到的所有发行版：

```
$ helm list --all-namespaces
NAME    NAMESPACE  REVISION  UPDATED        STATUS    CHART         APP VERSION
mysite  default    1         2020-06-14...  deployed  drupal-7.0.0  9.0.0
mysite  first      1         2020-06-14...  deployed  drupal-7.0.0  9.0.0
mysite  second     1         2020-06-14...  deployed  drupal-7.0.0  9.0.0
```

2.6 升级安装

当我们在Helm中谈论升级时，谈论的是升级一个安装，而不是一个chart。安装是集群中某个chart的特定实例。运行helm install时，它将创建这个安装。要修改该安装，需使用helm upgrade。

目前，这是一个重要的区别，因为升级安装可能包括两种不同的更改：

- 升级chart的版本
- 升级此安装的配置

这两者不是相互排斥的，可以同时进行。但这确实引入了Helm用户在谈论他们的系统时提到的一个新术语：版本是安装的配置和chart版本的特定组合。

当我们第一次安装chart时，我们会创建安装的初始版本，称之为版本1。当我们执行升级时，正在创建同一安装的新版本：版本2。当再次升级时，我们将创建版本3（在下一章中，我们将看到回滚如何创建版本）。

在升级过程中，我们可以创建一个具有新配置、新chart版本或两者兼有的版本。

例如，假设我们在ingress（入口）关闭的情况下安装Drupal chart（这将有效地防止流量从集群外部路由到Drupal实例）。

注意，我们使用--set标志来保持示例的紧凑性，但在常规场景中建议使用values.yaml文件：

```
$ helm install mysite bitnami/drupal --set ingress.enabled=false
```

在ingress关闭的情况下，我们可以根据自己的喜好设置网站。准备好之后，我们可以创建一个新的版本来启用ingress特性：

```
$ helm upgrade mysite bitnami/drupal --set ingress.enabled=true
```

在这种情况下，我们正在运行的upgrade只会更改配置。

在后台，Helm将加载chart，生成该chart中的所有Kubernetes对象，然后查看这些对象与已安装的chart的版本有何不同。它只会给Kubernetes发送需要更改的东西。换言之，Helm只会试图改变最少量的东西。

前面的示例只会更改ingress配置。数据库没有任何变化，甚至运行Drupal的Web服务器也没有任何变化。因此，不会重新启动或删除并重新创建任何内容。这有时会让Helm的新用户感到困惑，但这是特意设计的。Kubernetes的哲学是以最精简的方式进行更改，Helm试图遵循这一哲学。

有时，你可能需要强制某个服务重新启动。你不需要用Helm去实现，可以简单地使用kubectl本身来重启它。对于操作系统的软件包管理器，你不能使用软件包管理器重新启动程序。同样，你不需要使用Helm来重新启动Web服务器或数据库。

当chart的新版本出现时，你可能需要升级现有安装以使用新的chart版本。在很大程度上，Helm试图让这变得简单：

```
$ helm repo update ❶
Hang tight while we grab the latest from your chart repositories...
...Successfully got an update from the "bitnami" chart repository
Update Complete. ⎈ Happy Helming!⎈

$ helm upgrade mysite bitnami/drupal ❷
```

❶ 从chart存储库获取最新的软件包。

❷ 升级mysite版本以使用最新版本的bitnami/drupal。

如你所见，Helm的默认策略是尝试使用最新版本的chart。如果你希望保留chart的特定版本，可以显式声明：

```
$ helm upgrade mysite bitnami/drupal --version 6.2.22
```

在这种情况下，即使发布了更新的版本，也只会安装bitnaim/drupal版本6.2.22。

2.6.1 配置值和升级

关于Helm的安装和升级，需要了解的最重要的一点是，在每个版本中都会应用新的配置。下面是一个简单的例子：

```
$ helm install mysite bitnami/drupal --values values.yaml ❶
$ helm upgrade mysite bitnami/drupal ❷
```

❶ 使用配置文件安装。

❷ 不带配置文件的升级。

这一对操作的结果是什么？安装将使用values.yaml中提供的所有配置数据，但升级不会。因此，某些设置可能会更改回其默认值。这通常不是你想要的。

检查值

在下一章中，我们将研究helm get命令。你可以使用helm get values mysite查看上次helm install或helm upgrade操作中使用的值。

Helm核心维护人员建议你在每次安装和升级时提供一致的配置。要对两个发布版本应用相同的配置，可在每个操作上都提供值：

```
$ helm install mysite bitnami/drupal --values values.yaml ❶
$ helm upgrade mysite bitnami/drupal --values values.yaml ❷
```

❶ 使用配置文件安装。

❷ 使用相同的配置文件升级。

我们建议将配置存储在一个values.yaml文件中，以便很容易复制这个模式。想象一下，如果你使用--set来设置三个或四个配置参数，这些命令会有多麻烦！对于每个版本，你必须准确地记住要设置哪些内容。

虽然我们强烈建议使用这里讨论的模式，并每次指定--values，但是有一个升级快捷方式可以重用你发送的最后一组值：

```
$ helm upgrade mysite bitnami/drupal --reuse-values
```

--reuse values标志将告诉Helm重新加载最后一组值的服务器端副本，然后使用这些值生成升级。如果你总是重复使用相同的值，那么这个方法是可以的。但是，Helm维护人员强烈建议不要尝试将--reuse-values与其他--set或--values选项混合使用。这样做会使故障排除变得复杂，并很快导致无法维护的安装，因为没有人知道这个安装的某些配置参数是如何设置的。虽然Helm确实保留了一些状态信息，但它不是一个配置管理工具。建议用户使用自己的工具管理配置，并在每次调用中显式地将配置传递给Helm。

现在，我们已经学习了如何安装、列出和升级安装。在本章的最后一节中，我们将删除一个安装。

2.7 卸载安装

要删除某个Helm安装，可使用helm uninstall命令：

```
$ helm uninstall mysite
```

注意，这个命令不需要chart名（bitnami/drupal）或任何配置文件。它只需要安装的名称。在本节中，我们将了解删除是如何工作的，并简要介绍一下Helm 2和Helm 3之间的重大变化。

与install、list和upgrade类似，你可以提供--namespace标志来指定要从特定命名空间中删除安装：

```
$ helm uninstall mysite --namespace first
```

前面的命令将删除我们在本章前面的first命名空间中创建的站点。注意，没有命名可以删除多个应用程序，要想删除多个应用程序，必须卸载特定安装。

删除需要时间。大型应用程序可能需要几分钟甚至更长时间，因为Kubernetes会清理所有资源。在此期间，你将无法使用相同的名称重新安装。

2.7.1 Helm如何存储发布信息

Helm 3的一个重大变化是删除Helm自己关于安装的数据的方式。本节简要描述了如何跟踪安装，然后解释了Helm 2和Helm 3之间发生的更改以及更改的原因。

当第一次用Helm安装chart时（比如用helm install mysite bitnami/drupal），我们创建了Drupal应用程序实例，并且还创建了一个包含发布版本信息的特殊记录。默认情况下，Helm将这些记录存储为Kubernetes Secret（尽管还有其他受支持的存储后端）。

我们可以用kubectl get secret看到这些记录：

```
$ kubectl get secret
NAME                              TYPE                                  DATA   AGE
default-token-vjhx2               kubernetes.io/service-account-token   3      58m
mysite-drupal                     Opaque                                1      13m
mysite-mariadb                    Opaque                                2      13m
sh.helm.release.v1.mysite.v1      helm.sh/release.v1                    1      13m
sh.helm.release.v1.mysite.v2      helm.sh/release.v1                    1      13m
sh.helm.release.v1.mysite.v3      helm.sh/release.v1                    1      7m53s
sh.helm.release.v1.mysite.v4      helm.sh/release.v1                    1      5m30s
```

我们可以在列表底部看到多个版本记录，每个版本一个。如你所见，我们通过运行install和upgrade操作创建了mysite的4个修订版本。

在下一章中，我们将看到如何使用这些扩展记录回滚到以前的安装版本。但现在，我们指出这一点是为了说明helm uninstall是如何工作的。

运行helm uninstall mysite命令将加载mysite安装的最新版本记录。从该记录中，它将汇集一个应该从Kubernetes中移除的对象列表。然后Helm将删除所有这些内容，最后返回并删除4个版本的记录：

```
$ helm uninstall mysite
release "mysite" uninstalled
```

helm list命令将不再显示mysite：

```
$ helm list
NAME    NAMESPACE       REVISION        UPDATED STATUS  CHART   APP VERSION
```

我们现在没有安装了。如果重新运行kubectl get secrets命令，还会看到mysite的所有记录都被清除了：

```
$ kubectl get secrets
NAME                  TYPE                                  DATA   AGE
default-token-vjhx2   kubernetes.io/service-account-token   3      65m
```

从这个输出中我们可以看到，不仅Drupal chart创建的两个Secret被删除了，而且4个发布记录也被删除了。

在下一章中，我们将看到helm rollback命令。前面的解释应该给了你一些提示，说明为什么在默认情况下你不能回滚一个卸载。不过，你可以删除应用程序，但会保留发布记录：

```
$ helm uninstall --keep-history
```

在Helm 2中，历史记录被默认保留。在Helm 3中，默认值更改为删除历史记录。不同的组织喜欢不同的策略，但是核心维护人员发现，大多数人在执行卸载时希望所有安装痕迹都被销毁。

2.8 结论

在本章中，我们介绍了安装和使用Helm的基础知识。在研究了安装和配置Helm的流行方法之后，我们添加了一个chart存储库，并学习了如何搜索chart。然后我们安装、列出、升级并最终卸载了bitnami/drupal chart。

在本章中，我们学到了一些重要的概念。我们了解了安装和发布版本。我们初步了解了chart存储库，这将在第7章中详细介绍。在本章的结尾，我们学习了一些关于Helm如何存储有关安装的信息。

在下一章中，我们将回到helm命令上，学习Helm工具可以做的其他事情。

第3章

Helm的高级功能

在上一章中，我们研究了最常用的Helm命令。在本章中，我们将探讨Helm工具提供的其他功能。我们将深入研究提供有关发布、测试安装和跟踪历史等信息的命令。最后，我们将重新讨论安装和升级，这次将讨论高级案例。

我们还将开始使用一些有助于故障排除和调试的工具。

3.1 模板和试运行

当Helm安装一个发行版时，程序会经历几个阶段。它加载chart，解析传递给程序的值，读取chart元数据，等等。大约在这个过程的中间，Helm编译chart中的所有模板（一次完成），然后通过传递值来呈现它们（就像我们在上一章中看到的那样）。在这个中间部分，它执行所有的模板指令。一旦模板被呈现到YAML中，Helm通过将其解析为Kubernetes对象来验证YAML的结构。最后，Helm序列化这些对象并将它们发送到kubernetes API服务器。

过程如下：

1. 加载整个chart，包括其依赖项。

2. 解析值。

3. 执行模板，生成YAML。

4. 将YAML解析为Kubernetes对象以验证数据。

5. 将它发送给Kubernetes。

例如，让我们看看上一章中发出的如下命令：

```
$ helm install mysite bitnami/drupal --set drupalUsername=admin
```

在第一阶段，Helm将定位名为bitnami/drupal的chart并加载该chart。如果chart是本机的，它将从磁盘中读取。如果给定了一个URL，它将从远程位置获取（可能使用插件来帮助获取chart）。

然后它将把--set drupalUsername=admin转换为一个可以注入模板的值。此值将与chart的values.yaml文件中的默认值合并。Helm根据数据做了一些基本检查。如果它在解析用户输入时遇到问题，或者默认值已损坏，它将退出并返回一个错误。否则，它将构建一个大值对象，模板引擎可以使用该对象进行替换。

生成的值对象通过以下三个步骤来创建：加载chart文件的所有值；用从文件加载的任何值（即，使用-f标志）覆盖；使用--set标志设置的任何值覆盖。换句话说，--set值覆盖传入的值文件中的设置，而传入的值文件又覆盖chart的默认values.yaml文件中的任何内容。

此时，Helm将读取Drupal chart中的所有模板并执行这些模板，然后将合并的值传递给模板引擎。格式错误的模板将导致错误。但也有很多其他情况可能会导致失败。例如，如果缺少必需的值，则在此阶段返回错误。

需要注意的是，在执行时，一些Helm模板需要关于Kubernetes的信息。因此，在模板渲染期间，Helm可能会联系Kubernetes API服务器。这是我们稍后将讨论的一个重要主题。

然后将前面步骤的输出从YAML解析为Kubernetes对象。Helm将在此时执行一些模式级验证，确保对象的格式良好。然后将它们序列化为Kubernetes的最终YAML格式。

在最后一个阶段，Helm将YAML数据发送到Kubernetes API服务器。这是kubectl和其他Kubernetes工具与之交互的服务器。

API服务器将对提交的YAML运行一系列检查。如果Kubernetes接受YAML数据，Helm将认为部署成功。但是如果Kubernetes拒绝了YAML，Helm将以一个错误退出。

稍后，我们将详细讨论将对象发送到Kubernetes之后会发生什么。特别地，我们将介绍Helm如何将前面描述的过程与安装和修改相关联。但是现在，我们有足够的关于工作流的信息来理解两个相关的Helm特性：--dry-run标志和helm template命令。

3.1.1 --dry-run标志

helm install和helm upgrade等命令提供了一个名为--dry-run的标志。当在命令行中添加此标志时，将导致Helm逐步完成前四个阶段（加载chart、确定值、渲染模板、格式化为YAML）。但是当第四阶段结束时，Helm会将大量信息转储到标准输出，包括所有渲染的模板。然后它将退出，而不将对象发送给Kubernetes，也不创建任何发布记录。

例如，这里是以前的Drupal安装的一个版本，附加了--dry run标志：

```
$ helm install mysite bitnami/drupal --values values.yaml --set \
drupalEmail=foo@example.com --dry-run
```

在输出的最前面，它将打印有关此发布版本的信息：

```
NAME: mysite
LAST DEPLOYED: Tue Aug 11 11:42:05 2020
NAMESPACE: default
STATUS: pending-install
REVISION: 1
HOOKS:
```

前面的内容显示了安装的名称、上次的部署时间（在本例中是当前日期和时间）、它将部署到哪个命名空间、它处于版本的哪个阶段（pending-install）以及版本号。因为这是一个安装，所以修订版本是1。升级时，它将是2或更大的数字。

最后，如果这个chart声明了任何钩子，它们将在这里被枚举。有关钩子的更多信息，参见第6章和第7章。

乍一看，这个元数据条目似乎有很多不必要的数据。毕竟，如果我们没有实际安装，LAST DEPLOYED有什么用处呢？事实上，这部分信息是整个Helm中使用的标准集。它是发布记录的一部分：关于发布的一组信息。helm get等命令会使用这些相同的字段。

接下来，在信息块之后，所有渲染的模板都被转储到标准输出：

```
# Source: drupal/charts/mariadb/templates/test-runner.yaml
apiVersion: v1
kind: Pod
metadata:
  name: "mysite-mariadb-test-afv3u"
  annotations:
    "helm.sh/hook": test-success
spec:
  initContainers:
    - name: "test-framework"
      image: docker.io/dduportal/bats:0.4.0
...
```

渲染的Drupal chart有数千行，因此前面只显示了输出的前几行。

在试运行输出的最后，Helm打印面向用户的发行版本说明：

```
NOTES:
******************************************************************
*** PLEASE BE PATIENT: Drupal may take a few minutes to install ***
******************************************************************

1. Get the Drupal URL:

  You should be able to access your new Drupal installation through

  http://drupal.local/
...
```

为了简洁起见，这个例子被截断了。

这个试运行（dry-run）特性为Helm用户提供了一种在chart把输出发送到Kubernetes之前调试它的方法。通过渲染所有模板，你可以准确地检查提交给集群的内容。通过发布版本数据，你可以验证是否按照预期创建了发布版本。

--dry-run标志的主要目的是让人们有机会在将输出发送到Kubernetes之前检查和调试输出。但在它被引入后不久，Helm维护人员就注意到了用户中的一种趋势：人们希望使用--dry-run将Helm用作模板引擎，然后使用其他工具（如kubectl）将渲染的输出发送到Kubernetes。

但是编写--dry-run时没有考虑到这个用例，这导致了一些问题：

1.--dry-run将非YAML信息与渲染的模板混合。这意味着在将数据发送到kubectl等工具之前，必须对其进行清理。

2. --dry-run在升级时可能会产生与其安装时不同的YAML输出，这可能会让人困惑。

3. 它会联系Kubernetes API服务器进行验证，这意味着Helm必须拥有Kubernetes凭据，即使它只是用来试运行一个发行版本。

4. 它还将特定于集群的信息插入模板引擎。因此，某些渲染过程的输出可能是特定于集群的。

为了解决这些问题，Helm维护人员引入了一个完全独立的命令：helm template。

3.1.2 helm template命令

虽然--dry-run是为调试而设计的，但helm template是为了将Helm的模板渲染过程与安装或升级逻辑隔离开来。

之前，我们研究了Helm安装或升级的5个阶段。template命令执行前4个阶段（加载chart、解析值、渲染模板、格式化为YAML）。但它也有一些额外的注意事项：

· 在helm template执行期间，Helm从不联系远程Kubernetes服务器。

· template命令的作用始终类似于安装。

· 通常需要联系Kubernetes服务器的模板函数和指令将只返回默认数据。

· chart只能访问默认的Kubernetes类型。

关于最后一项，helm template做了一个显著的简化假设。Kubernetes服务器支持内置种类（Pod、Service、ConfigMap等）以及由CRD生成的自定义类型。当运行安装或升级时，Helm在处理chart之前从Kubernetes服务器获取这些类型。

但是，helm template执行此步骤的方式不同。在编译Helm时，它是针对Kubernetes的特定版本编译的。Kubernetes库包含该版本的内

置类型列表。Helm使用这个内置列表，而不是它从API服务器获取的列表。因此，Helm在helm template运行期间不能访问任何CRD，因为CRD安装在集群上，并且不包含在Kubernetes库中。

 如果针对使用新类型或新版本的chart运行旧版本的Helm，会在helm template运行期间产生错误，因为Helm不会把最新类型或版本编译进去。

作为这些决策的结果，helm template在运行之后生成一致的输出。更重要的是，它可以在不能访问Kubernetes集群的环境中运行，比如持续集成（CI）管道。

输出也不同于--dry-run。下面是一个命令示例：

```
$ helm template mysite bitnami/drupal --values values.yaml --set \
drupalEmail=foo@example.com
 ---
# Source: drupal/charts/mariadb/templates/secrets.yaml
apiVersion: v1
kind: Secret
metadata:
  name: mysite-mariadb
  labels:
    app: "mariadb"
    chart: "mariadb-7.5.1"
    release: "mysite"
    heritage: "Helm"
type: Opaque
# ... LOTS removed from here
  volumes:
    - name: tests
      configMap:
        name: mysite-mariadb-tests
    - name: tools
      emptyDir: {}
  restartPolicy: Never
```

前面是一个大大简化了的输出版本，只显示了命令、开始数据和结束数据的示例。不过，需要重点注意的是，默认情况下只打印YAML格式的Kubernetes清单。

因为Helm在helm template运行期间不联系Kubernetes集群，所以它不会对输出进行完全验证。在这种情况下，Helm可能不会发现一些错误。如果你想要这种行为，可以选择使用--validate标志，但是在这种情况下，Helm需要一个有效的kubeconfig文件，其中包含集群的凭据。

helm template命令有大量的标志，这些标志对应于helm install中的标志。因此，在许多情况下，你可以像执行helm install一样执行helm template命令，随后捕获YAML并将其与其他工具一起使用。

使用后期渲染而不是Helm模板

有时你需要截取YAML，用自己的工具修改它，然后将其加载到Kubernetes中。Helm提供了一种执行这个外部工具的方法，而不必使用helm template。install、upgrade、rollback和template上的--post-renderer标记会导致Helm将YAML数据发送到命令，然后将结果读回Helm。这是使用Kustomize等工具的好方法。

总而言之，helm template是一个用于将Helm chart渲染到YAML中的工具，--dry-run标志是一个用于调试安装和升级命令，而无须将数据加载到Kubernetes中的工具。

3.2 了解发布版本信息

在上一章中，我们看到了helm get命令。本节将更深入地研究该命令以及其他可以提供有关Helm发布版本信息的命令。

首先回顾上一节中的5个Helm安装阶段：

1. 加载chart。

2. 解析值。

3. 执行模板。

4. 渲染YAML。

5. 把它发送到Kubernetes。

前4个阶段主要涉及数据的局部表示。即，Helm在运行helm命令的同一台计算机上执行所有处理。

不过，在最后一个阶段，Helm将数据发送给Kubernetes。然后二者“来回沟通”，直到发布版本被接受或拒绝。

在第5阶段，Helm必须监控发布状态。此外，由于许多人可能正在处理该特定应用程序安装的同一副本，因此Helm需要以多个用户可以看到该信息的方式监控状态。

Helm为这个特性提供了发布记录。

3.2.1 发布记录

当我们（使用helm install）安装一个Helm chart时，新的安装将在你指定的命名空间或默认命名空间中创建（在第2章中讨论过）。

在第2章的结尾，我们还看到了helm install如何创建一种特殊类型的Kubernetes Secret来保存发布信息。我们看到了如何与kubectl一起检查这些Secret：

```
$ kubectl get secret
NAME                            TYPE                                  DATA   AGE
default-token-g777k             kubernetes.io/service-account-token   3      6m
mysite-drupal                   Opaque                                1      2m20s
mysite-mariadb                  Opaque                                2      2m20s
sh.helm.release.v1.mysite.v1    helm.sh/release.v1                    1      2m20s
```

特别值得注意的是最后一个Secret，即sh.helm.release.v1.mysite.v1。注意，它使用了一个特殊类型（helm.sh/release.v1）来表示这是一个Helm Secret。Helm自动生成这个Secret来跟踪mysite安装的版本1（这是一个Drupal站点）。

每次升级mysite安装时，都会创建一个新的Secret来跟踪每个版本。换句话说，发布记录跟踪安装的每个修订版：

```
$ helm upgrade mysite bitnami/drupal
# Output omitted
$ helm upgrade mysite bitnami/drupal
# Output omitted
$ kubectl get secrets
NAME                            TYPE                                  DATA   AGE
default-token-g777k             kubernetes.io/service-account-token   3      8m43s
mysite-drupal                   Opaque                                1      5m3s
mysite-mariadb                  Opaque                                2      5m3s
sh.helm.release.v1.mysite.v1    helm.sh/release.v1                    1      5m3s
sh.helm.release.v1.mysite.v2    helm.sh/release.v1                    1      20s
sh.helm.release.v1.mysite.v3    helm.sh/release.v1                    1      8s
```

在前面的例子中，我们已经升级了几次，现在是mysite的v3。默认情况下，Helm最多跟踪每个安装的10个修订版。一旦安装超过10个发布版本，Helm将删除最早的发布记录，直到剩余的数量不超过最大允许值。

每个发布记录都包含足够的信息，可以为该修订版重新创建Kubernetes对象（这对于helm rollback来说很重要）。它还包含有关发布的元数据。

例如，如果我们使用kubectl查看此发布版本，会看到如下内容：

```
apiVersion: v1
data:
  release: SDRzSUFBQU... # Lots of Base64-encoded data removed
kind: Secret
metadata:
  creationTimestamp: "2020-08-11T18:37:26Z"
  labels: ❶
    modifiedAt: "1597171046"
    name: mysite
    owner: helm
    status: deployed
    version: "3"
  name: sh.helm.release.v1.mysite.v3
  namespace: default
  resourceVersion: "1991"
  selfLink: /api/v1/namespaces/default/secrets/sh.helm.release.v1.mysite.v3
  uid: cbb8b457-e331-467b-aa78-1e20360b5be6
type: helm.sh/release.v1
```

❶ 标签包含Helm元数据。

在本例中，数量巨大的Base64编码数据与其他一些不重要的字段一起被删除。该blob包含chart和版本的gzip表示。但重要的是，Kubernetes元数据的labels部分包含有关此版本的信息。

例如，我们可以看到，这个数据描述了名为mysite的版本，它的当前修订号是3，这个版本被标记为deployed。如果我们看第2版，会看到发布状态（status）是superseded，这意味着它已经被更新的版本所取代。

简而言之，这个Secret存储在Kubernetes中，以便同一集群的不同用户可以访问相同的发布信息。

在某个发布版的生命周期中，它可以经历几个不同的状态。在这里，大致按照你可能看到的顺序排列它们：

pending-install（挂起-安装）

在将清单发送给Kubernetes之前，Helm通过创建一个状态设置为pending-install的版本（标记为version 1）来声明安装。

deployed（已部署）

一旦Kubernetes接受Helm的清单，Helm就会更新发布记录，将其标记为已部署（deployed）。

pending-upgrad（挂起-升级）

当Helm升级开始时，会为安装创建一个新的发布版本（例如v2），其状态设置为pending-upgrade。

superseded（已取代）

在运行升级时，将更新上次部署的发布版本，并将其标记为已取代（superseded），新升级的版本将从pending-upgrade更改为deployed。

pending-rollback（挂起-回滚）

如果创建了回滚，则会创建一个新的发布版本（例如v3），其状态设置为pending-rollback，直到Kubernetes接受该发布版本清单。然后它被标记为deployed，而前一个发布版本被标记为superseded。

uninstalling（正在卸载）

执行helm uninstall时，将读取最新发布版本，然后将其状态更改为uninstalling。

uninstalled（已卸载）

如果在删除过程中保留了历史记录，那么当helm uninstall完成时，前一个发布版本的状态将更改为uninstalled。

failed（失败）

最后，如果在任何操作中，Kubernetes拒绝了Helm提交的清单，Helm都会将该发布版本标记为failed。

3.2.2 列出发布版本

状态信息在许多Helm命令中都会显示。我们已经看到了pending-install是如何在一个--dry-run中出现的。在本节和下一节中，我们将看到更多这种情况。

在上一章中，我们使用helm list查看安装的chart。鉴于我们对各种状态的知识，值得再回顾一下helm list。list命令是快速检查版本状态的最佳工具。

例如，假设我们有一个同时安装了drupal和wordpress chart的集群。以下是helm list的输出：

```
NAME      NAMESPACE REVISION  UPDATED       STATUS   CHART           APP V...
mysite    default   3         2020-08-11... deployed  drupal-7.0.0     9.0.0
wordpress default   2         2020-08-12... deployed  wordpress-9.3.11 5.4.2
```

不过，为了显示失败的结果，我们可以运行一个升级命令，我们知道该命令将中断：

```
$ helm upgrade wordpress bitnami/wordpress --set image.pullPolicy=NoSuchPolicy
Error: UPGRADE FAILED: cannot patch "wordpress" with kind Deployment:
Deployment.apps "wordpress" is invalid:
spec.template.spec.containers[0].imagePullPolicy: Unsupported value:
"NoSuchPolicy": supported values: "Always", "IfNotPresent", "Never"
```

如错误消息所示，拉取（pull）策略不能设置为NoSuchPolicy。这个错误来自Kubernetes API服务器，这意味着Helm提交了清单，而Kubernetes拒绝了它。所以我们的发布版本应该是失败的。

我们可以再次运行helm ls来验证这一点：

```
$ helm ls
NAME      NAMESPACE REVISION  UPDATED     STATUS    CHART           APP VER...
mysite    default   3         2020-08-11  deployed  drupal-7.0.0    9.0.0
wordpress default   3         2020-08-12  failed    wordpress-9.3.11 5.4.2
```

值得再次注意的是，我们最近失败的wordpress安装的REVISION字段已经从2增加到了3。即使是失败的版本也附带了修订号。我们将在3.3节中了解为什么这很重要。

3.2.3 使用helm get查找发布的详细信息

虽然helm list提供了安装的摘要视图，但helm get命令集提供了有关特定发布版本的更深入的信息。

helm get有5个子命令（hooks、manifests、notes、values和all）。每个子命令都会检索某个版本的Helm跟踪信息的某些部分。

使用helm get notes

helm get notes子命令打印发布说明：

```
$ helm get notes mysite
NOTES:

******************************************************************
*** PLEASE BE PATIENT: Drupal may take a few minutes to install ***
******************************************************************

1. Get the Drupal URL:

  You should be able to access your new Drupal installation through

  http://drupal.local/
...
```

这个输出看起来应该很熟悉，因为helm install和helm upgrade都会在成功操作结束时打印发布说明。但是helm get notes提供了一种方便的方式来按需获取这些说明。如果你忘记了Drupal站点的URL是什么，这将非常有用。

使用helm get values

另一个有用的子命令是values。你可以使用它来查看上一个版本提供了哪些值。在上一节中，我们升级了WordPress安装并导致它失败。我们可以使用helm get values来查看是什么值导致它失败：

```
$ helm get values wordpress
USER-SUPPLIED VALUES:
image:
  pullPolicy: NoSuchPolicy
```

我们知道第二次修订是成功的，但是第三次修订失败了。所以我们可以看看前面的值有什么变化：

```
$ helm get values wordpress --revision 2
USER-SUPPLIED VALUES:
image:
  tag: latest
```

这样，我们可以看到一个值被删除，一个值被添加。类似这样的特性旨在使Helm用户更容易识别错误的来源。

此命令对于了解版本配置的总体状态也很有用。我们可以使用helm get values来查看当前为该版本设置的所有值。为此，我们使用--all标志：

```
$ helm get values wordpress --all
COMPUTED VALUES:
affinity: {}
allowEmptyPassword: true
allowOverrideNone: false
customHTAccessCM: null
customLivenessProbe: {}
customReadinessProbe: {}
externalDatabase:
  database: bitnami_wordpress
  host: localhost
  password: ""
  port: 3306
...
```

当指定--all标志时，Helm将获得按字母顺序排序的完整计算值集。这是一个很好的工具，可以查看发行版的确切配置状态。

查看默认值

尽管helm get values没有显示默认值的方法，但是你可以看到那些带有helm inspect values CHARTNAME的值。这将检查chart本身（而不是发布版本）并打印出记录的默认values.yaml文件。因此，我们可以使用helm inspect values bitnami/wordpress来查看WordPress chart的默认配置。

使用helm get manifest

我们将讨论的最后一个helm get子命令是helm get manifest。此子命令获取Helm使用Chart模板生成的确切YAML清单：

```
$ helm get manifest wordpress
# Source: wordpress/charts/mariadb/templates/secrets.yaml
apiVersion: v1
kind: Secret
metadata:
  name: wordpress-mariadb
  labels:
    app: "mariadb"
    chart: "mariadb-7.5.1"
    release: "wordpress"
    heritage: "Helm"
type: Opaque
...
```

关于这个命令的一个重要细节是，它不会返回所有资源的当前状态。它返回从模板生成的清单。在前面的示例中，我们看到了一个名为wordpress-mariadb的Secret。如果我们使用kubectl查询这个Secret，则元数据部分如下：

```
$ kubectl get secret wordpress-mariadb -o yaml
apiVersion: v1
kind: Secret
metadata:
  annotations:
    meta.helm.sh/release-name: wordpress
    meta.helm.sh/release-namespace: default
  creationTimestamp: "2020-08-12T16:45:00Z"
  labels:

    app: mariadb
    app.kubernetes.io/managed-by: Helm
    chart: mariadb-7.5.1
    heritage: Helm
    release: wordpress
 managedFields:
 - apiVersion: v1
   fieldsType: FieldsV1
...
```

kubectl的输出包含当前存在于Kubernetes中的记录。自模板输出后添加了几个字段。一些（如注解）由Helm自己管理，而另一些（如

managedFields和creationTimestamp）由Kubernetes管理。

Helm再次提供了简化调试的工具。在helm get manifest和kubectl get之间，有一些工具可以用来比较Kubernetes认为当前的对象是什么，以及chart产生了什么。当应该由Helm管理的资源在Helm外部（例如，使用kubectl edit）手动编辑时，这尤其有用。

有了helm get，我们可以仔细检查一个单独的发布版本。但我们将介绍的下一个工具提供了一个有关发布进程的视图。在下一节中，我们将了解helm history和helm rollback。

3.3 历史记录和回滚

在本书中，我们区分了安装和修订。在本章中，我们使用了两个安装：mysite和wordpress。当早些时候运行helm list时，可以看到每个安装都有三个版本。此外，可以看到wordpress处于失败状态：

```
$ helm list
NAME      NAMESPACE REVISION  UPDATED     STATUS    CHART           APP VER...
mysite    default   3         2020-08-11  deployed  drupal-7.0.0    9.0.0
wordpress default   3         2020-08-12  failed    wordpress-9.3.11 5.4.2
```

我们可以调查WordPress的发布历史，看看发生了什么。为此，我们将使用helm history：

```
$ helm history wordpress
REVISION UPDATED       STATUS     CHART           APP VER  DESCRIPTION
1        Wed Aug 12... superseded wordpress-9.3.11 5.4.2    Install complete
```

```
2       Wed Aug 12... deployed   wordpress-9.3.11  5.4.2    Upgrade complete
3       Wed Aug 12... failed     wordpress-9.3.11  5.4.2    Upgrade \
  "wordpress" failed: cannot patch "wordpress" with kind Deployment: \
  Deployment.apps "wordpress" is invalid: \
  spec.template.spec.containers[0].imagePullPolicy: Unsupported value: \
  "NoSuchPolicy": supported values: "Always", "IfNotPresent", "Never"
```

这个命令的输出提供了wordpress发布的良好历史记录。首先安装，然后升级并标记为已部署（这意味着升级成功）。但当它再次升级时，升级失败了。helm history命令甚至给出了Kubernetes在标记发布失败时返回的错误消息。

从错误消息中，我们知道发布失败是因为我们提供了无效的镜像拉取策略。所以我们当然可以通过运行另一个helm upgrade来纠正这个问题。但设想一下并不容易查到错误原因的情况。与其在诊断问题时让应用程序处于失败状态，不如简单地恢复到以前的版本。

这就是helm rollback的用途：

```
$ helm rollback wordpress 2
Rollback was a success! Happy Helming!
```

这个命令告诉Helm获取wordpress version 2版本，并将清单重新提交给Kubernetes。回滚不会还原到集群的上一个快照。Helm没有追踪足够的信息来做到这一点。它会重新提交以前的配置，而Kubernetes尝试重置资源以匹配。

现在我们可以再次使用helm history来查看发生了什么：

```
REVISION  UPDATED      STATUS     CHART            APP VER  DESCRIPTION
1         Wed Aug 12... superseded wordpress-9.3.11 5.4.2    Install complete
2         Wed Aug 12... superseded wordpress-9.3.11 5.4.2    Upgrade complete
3         Wed Aug 12... failed     wordpress-9.3.11 5.4.2    Upgrade \
  "wordpress" failed: cannot patch "wordpress" with kind Deployment: \
  Deployment.apps "wordpress" is invalid: \
  spec.template.spec.containers[0].imagePullPolicy: Unsupported value: \
  "NoSuchPolicy": supported values: "Always", "IfNotPresent", "Never"
4         Wed Aug 12... deployed   wordpress-9.3.11 5.4.2    Rollback to 2
```

回滚操作创建了一个新版本（4）。由于回滚是成功的（并且Kubernetes接受了修改），所以这个版本被标记为deployed。注意，修订版2现在被标记为已取代（superseded），而失败的第3版仍然被标记为失败（failed）。

因为Helm保存了历史记录，所以在回滚到已知的良好配置之后，仍然可以检查失败的版本。

在大多数情况下，helm rollback是从灾难中恢复过来的良好方法。但是，如果手工编辑由Helm管理的资源，可能会出现一个有趣的问题。回滚有时会导致一些意外的行为，特别是在Kubernetes资源已经被用户手工编辑的情况下。如果这些手工编辑的内容与回滚不冲突，Helm和Kubernetes将尝试保留它们。从本质上讲，回滚将在资源的当前状态、失败的Helm版本和回滚到的Helm版本之间产生一个三向差异。在某些情况下，生成的差异可能导致回滚手工编辑的内容，而在其他情况下，这些差异将被合并。在最坏的情况下，某些手工编辑的内容可能会被覆盖，而其他相关的编辑内容则会被合并，从而导致配置不一致。这是Helm核心维护人员建议不要手工编辑资源的众多原因之一。如果所有的编辑都是通过Helm进行的，那么你就可以有效地使用Helm工具，而无须猜测。

3.3.1 保留历史和回滚

在上一章中，我们看到helm uninstall命令有一个名为--keep history的标志。通常，删除事件将销毁与该安装相关联的所有发布记录。但是当指定--keep history时，即使删除了安装，你也可以看到它的历史记录：

```
$ helm uninstall wordpress --keep-history
release "wordpress" uninstalled
$ helm history wordpress
REVISION UPDATED      STATUS     CHART            APP V  DESCRIPTION
1        Wed Aug 12... superseded wordpress-9.3.11 5.4.2  Install complete
2        Wed Aug 12... superseded wordpress-9.3.11 5.4.2  Upgrade complete
3        Wed Aug 12... failed     wordpress-9.3.11 5.4.2  Upgrade \
  "wordpress" failed: cannot patch "wordpress" with kind Deployment: \
  Deployment.apps "wordpress" is invalid: \
  spec.template.spec.containers[0].imagePullPolicy: Unsupported value: \
  "NoSuchPolicy": supported values: "Always", "IfNotPresent", "Never"
4        Wed Aug 12... uninstalled  wordpress-9.3.11  5.4.2  Uninstall complete
```

注意，最后一个版本现在标记为已卸载（uninstalled）。当历史记录被保留时，你可以回滚已删除的安装：

```
$ helm rollback wordpress 4
Rollback was a success! Happy Helming!
```

现在我们可以看到新部署的版本5：

```
$ helm history wordpress
REVISION UPDATED    STATUS      CHART            APP VER  DESCRIPTION
1        Wed Aug... superseded  wordpress-9.3.11       5.4.2    Install complete
2        Wed Aug... superseded  wordpress-9.3.11       5.4.2    Upgrade complete
3        Wed Aug... failed      wordpress-9.3.11       5.4.2    Upgrade \
  "wordpress" failed: cannot patch "wordpress" with kind Deployment: \
  Deployment.apps "wordpress" is invalid: \
  spec.template.spec.containers[0].imagePullPolicy: Unsupported value: \
  "NoSuchPolicy": supported values: "Always", "IfNotPresent", "Never"
4        Wed Aug... uninstalled wordpress-9.3.11  5.4.2    Uninstall complete
5        Wed Aug... deployed    wordpress-9.3.11  5.4.2    Rollback to 4
```

如果没有--keep history标志，这将不起作用：

```
$ helm uninstall wordpress
release "wordpress" uninstalled
$ helm history wordpress
Error: release: not found
$ helm rollback wordpress 4
Error: release: not found
```

3.4 深入了解安装和升级

在第2章中，我们首先介绍了如何安装和升级Helm包，在本章中，我们还介绍了帮助我们使用Helm安装的工具。在本节中，我们将再次回到安装和升级，并查看一些高级功能。

3.4.1 --generate-name和--name-template标志

Kubernetes的工作方式有一个与命名有关的微妙危险。Kubernetes假设名称将具有某些唯一性属性。例如，Deployment对象的命名空间中必须具有唯一的名称。也就是说，在命名空间mynamespace中不能有两个名为myapp的Deployment。但可以有一个Deployment和一个Pod，它们都叫myapp。

这使得某些任务变得更加复杂。例如，自动部署内容的CI系统必须确保这些内容的名称在命名空间中是唯一的。处理这个问题的一种方法是Helm提供一个工具来生成唯一的名称（另一种方法是，如果名称已经存在，则始终覆盖该名称。有关该方法，请参见下一节）。

Helm为helm install提供--generate name标志：

```
$ helm install bitnami/wordpress --generate-name
NAME: wordpress-1597689085
LAST DEPLOYED: Mon Aug 17 11:31:27 2020
NAMESPACE: default
STATUS: deployed
REVISION: 1
```

使用--generate name标志，我们不再需要提供名称作为helm install的第一个参数。Helm根据chart名称和时间戳的组合生成名称。在前面的输出中，我们可以看到生成的名称：wordpress-1597689085。

在Helm 2中，“友好的名字”是用形容词和动物的名字产生的。因为有人抱怨发布的名字不专业，所以在Helm 3中删除了这种命名模式。目前无法重新启用此功能。

但是，还有一个额外的标志允许你指定命名模板。--name-template标志允许你执行以下操作：

```
$ helm install bitnami/wordpress --generate-name \
  --name-template "foo-{{ randAlpha 9 | lower }}"
NAME: foo-yejpiyjmp
LAST DEPLOYED: Mon Aug 17 11:46:04 2020
NAMESPACE: default
```

在本例中，我们使用了名称模板foo-{{randAlpha 9|lower}}。这将使用Helm模板引擎生成一个名称。在接下来的几章中，我们将介绍Helm模板引擎。但是名称模板的作用是：{{和}}划分模板的开始和结束。在该模板中，我们调用randAlpha函数，从a-z、A-Z字符范围中请求一个9个字符的随机字符串。然后我们通过第二个函数（lower）将结果“管道化”，该函数将所有内容都小写化。

查看前面示例的输出，{{randAlpha 9|lower}}的结果是yejpiyjmp。所以整个名称模板的结果是foo-yejpiyjmp。

3.4.2 --create-namespace标志

Kubernetes中命名的另一个考虑因素与命名空间有关。前面，我们看到同一命名空间中的同一类型的两个对象不能有相同的名称。但Kubernetes也有一个全局名称的概念。CRD和命名空间都有全局名称。

因此，命名空间必须是集群范围内唯一的。

每当Helm遇到全局唯一的名字，它就会采取防御的姿态。在后面的章节中，我们将看到chart如何处理全局唯一的名称。但这里值得指出的是，Helm 3默认情况下假定，如果你尝试将chart部署到命名空间中，则该命名空间已经创建。

例如，在一个新集群上，这将失败：

```
$ helm install drupal bitnami/drupal --namespace mynamespace
Error: create: failed to create: namespaces "mynamespace" not found
```

它失败了，因为mynamespace尚未创建，Helm不会自动创建命名空间。它不会创建空间，因为命名空间是全局的。安全的假设是，在创建命名空间时，它可能需要访问控制（如RBAC）和分配给它的其他东西，然后才能在生产中安全地使用。简而言之，它将静默地创建名称空间视为无意中创建安全漏洞的机会。

但是，Helm确实允许你通过显式地声明要创建命名空间来覆盖此考虑：

```
$ helm install drupal bitnami/drupal --namespace mynamespace --create-namespace
NAME: drupal
LAST DEPLOYED: Mon Aug 17 11:59:29 2020
NAMESPACE: mynamespace
STATUS: deployed
```

通过添加--create-namespace，我们已经向Helm表明，我们知道可能还没有具有该名称的命名空间，只希望创建一个。当然，请确保如果在生产实例上使用此标志，会有其他机制来强制执行此新命名空间的安全性。

在helm uninstall上没有类似的--delete-namespace。原因在于Helm对全局对象的防御性。创建命名空间后，可以将任意数量的对象放入其中，而它们不一定全都是由Helm管理的。当一个命名空间被删除时，该命名空间内的所有对象也都被删除。所以Helm不会自动删除用--create-namespace创建的命名空间。要删除命名空间，请使用kubectl delete namespace（当然，在确保该命名空间中不存在重要对象之后再删除）。

3.4.3 使用helm upgrade--install

有些系统（如CI管道）用于在每次发生重大事件时自动安装或升级chart。例如，许多组织都有管道，每当新代码上传到Git等版本控制系统（VCS）时就会触发。GitHub是一种流行的Git托管服务，它甚至提供了在合并代码更改时自动部署的工具。

这样的系统通常在无状态平台上运行基本脚本，而无状态平台没有查询Kubernetes的方法。这类系统的用户要求Helm提供一个特性，允许在一个命令中支持“安装或升级”。

为了方便这种行为，Helm维护人员在helm upgrade命令中添加了--install标志。helm upgrade--install命令将安装一个尚不存在的版本，或者升级一个以该名称命名的版本。在底层，它通过查询

Kubernetes以获得具有给定名称的发布版。如果该版本不存在，它将从升级逻辑切换到安装逻辑。

例如，我们可以使用完全相同的命令依次运行安装和升级：

```
$ helm upgrade --install wordpress bitnami/wordpress
Release "wordpress" does not exist. Installing it now.
NAME: wordpress
LAST DEPLOYED: Mon Aug 17 13:18:14 2020
NAMESPACE: default
STATUS: deployed

...
$ helm upgrade --install wordpress bitnami/wordpress
Release "wordpress" has been upgraded. Happy Helming!
NAME: wordpress
LAST DEPLOYED: Mon Aug 17 13:18:43 2020
NAMESPACE: default
STATUS: deployed
```

正如我们在第一行输出中看到的，命令的第一次运行导致安装，而第二次运行导致升级。

不过，这个命令确实带来了一些危险。Helm无法确定你提供给helm upgrade--install的安装名称是属于你要升级的发布版本，还是恰好与你要安装的名称相同。不小心使用此命令可能会导致用另一个安装覆盖某个安装。这就是它不是Helm的默认行为的原因。

3.4.4 --wait和--atomic标志

helm install和helm upgrade的另外两个重要标志修改了Helm操作的成功标准。它们是--wait和--atomic标志。

--wait标志有几种方式可以修改Helm客户端的行为。首先，当Helm运行一个安装时，它会在一段时间内保持活动状态（可以使用--timeout标志进行修改），在此期间它会监视Kubernetes。它轮询Kubernetes API服务器，以获取有关由chart创建的所有pod运行对象的信息。例如，DaemonSet、Deployment和StatefulSet（守护程序集、部署程序集和状态程序集）都创建pod。因此，使用--wait的Helm将一直跟踪这些对象，直到它们创建的pod被标记为由Kubernetes运行为止。

在正常的安装或升级中，只要Kubernetes API服务器接受清单，Helm就会将发布标记为成功。这类似于软件包管理器，它认为只要软件包内容写入正确的存储位置，就算成功地安装了软件包。

但是使用--wait，安装成功的标准被修改了。除非Kubernetes API服务器接受清单，并且由chart创建的所有pod都在Helm超时过期之前达到运行状态，否则chart不会被视为已成功安装。

因此，带有--wait的安装可能会由于各种原因而失败，包括网络延迟、调度程序速度慢、节点忙、镜像拉取速度慢，以及容器完全无法启动。

这种行为被视为一种理想的结果，操作员使用helm install--wait来确保不仅成功安装了chart，而且正确启动了生成的应用程序。但是，在排除故障时，它确实会引入一些复杂的因素。短暂停机可能会导致Helm故障（稍后由Kubernetes解决该故障）。例如，镜像拉取延迟可能会导致某个Helm发布版本被标记为失败，即使几分钟后镜像拉取可以完成并且应用程序可以启动。

不过，考虑到这一点，helm install--wait是一个很好的工具，可以确保发布版本一直在运行。但在自动化系统（如CI）中使用时，可能会导致虚假故障。在CI中使用--wait的一个建议是使用长--timeout（5或10分钟），以确保Kubernetes有时间解决任何瞬时故障。

第二种策略是使用--atomic标志而不是--wait标志。除非发布失败，否则此标志导致的行为与--wait相同。然后，它将自动回滚到最后一个成功的版本，而不是将该版本标记为失败（failed）并退出。在自动化系统中，--atomic标志更能抵抗中断，因为它的最终结果不太可能出现故障（不过，请记住，不保证回滚一定会成功）。

正如--wait可以出于Kubernetes本身可以解决的过渡原因将发布版本标记为失败一样，--atomic可能会出于同样的原因而触发不必要的回滚。因此，建议使用更长的--atomic超时时间（--timeout），特别是在与CI系统一起使用时。

3.4.5 使用--force和--cleanup-on-fail升级

我们下面看到的最后两个标志修改了Helm处理升级的细微差别的方式。

--force标志在升级用于管理pod（如Pod、Deployment和StatefulSet）的资源时修改Helm的行为。通常，当Kubernetes收到修改此类对象的请求时，它会确定是否需要重新启动此资源管理的pod。例如，某个Deployment可以运行一个pod的五个副本。但是，如果Kubernetes接收到Deployment对象的更新，它只会在某些字段被修改时重新启动这些pod。

不过，有时Helm用户希望确保pod重启。这时就用到了--force标记。它将删除并重新创建Deployment，而不是修改Deployment（或类似对象）。这迫使Kubernetes删除旧pod并创建新pod。根据设计，使用--force会导致停机（通常只有几秒钟的停机时间）。建议仅在明确需要的情况使用--force，而不是作为默认选项。例如，核心维护人员不建议在部署到生产环境的CI管道中使用--force。

修改升级行为的另一种方法是使用--cleanup on fail标志。类似于--force，这个标志指示Helm做额外的工作。

考虑这样一种情况：安装一个创建Kubernetes Secret的chart。一个新版本的chart被创建，它创建了第二个Secret。但在安装过程中，Helm遇到了一个错误，并将发布标记为失败。第二个Secret有可能被挂起。如果使用--wait或--atomic，则更可能出现这种情况，因为在Kubernetes接受清单并创建资源之后，这些操作可能会失败。

--cleanup-on-fail标志将尝试修复此情况。如果失败，它将请求删除升级期间新创建的每个对象。使用它可能会使调试变得更加困难（特别是如果失败是由新创建的对象造成的），但是如果你不想冒险在失败后挂起未使用的对象，那么它是很有用的。

3.5 结论

Helm命令行工具提供了许多有用的命令。第2章介绍了基本命令，本章重点介绍了Helm中其他一些有用的命令。本章末尾，我们还重新复习了安装和升级命令，体验了使用这些命令的一些更复杂的功能。

但是，这里并未讨论所有的命令。在接下来的章节中，我们将了解用于创建和打包chart的命令、用于签名和验证软件包的命令，以及更多用于使用存储库的命令。

第4章

构建chart

chart是Helm的核心。除了将它们安装到Kubernetes集群中或管理已安装的chart实例之外，还可以构建新chart或更改现有chart。在接下来的三章中，我们将介绍很多关于chart的细节，包括创建chart、chart中的元素、模板化Kubernetes清单、测试chart、依赖项等。

在本章中，你将学习如何创建新chart，并了解chart的许多部分，包括几个内置命令的用法（它们可以在chart开发过程中提供帮助）。

chart是Helm使用的软件包。它们在概念上类似于APT使用的Debian软件包或macOS Homebrew使用的Formula。但它们的相似性仅限于概念上的相似性。chart的设计目标是把Kubernetes作为一个有自己独特风格的平台。chart的核心是模板，该模板用于生成可以在集群中安装和管理的Kubernetes清单。

在深入研究第5章中的模板之前，让我们先创建一个基本的全功能chart。为此，我们将介绍一个名为anvil的示例chart。利用这个chart，你将了解如何使用Helm生成chart、chart和其中的文件的结构、打包chart和校验chart代码。请参阅此chart的在线源代码：https://github.com/Masterminds/learning-helm/tree/main/chapter4/anvil。

4.1 chart创建命令

Helm包含create命令，使你可以轻松地创建自己的chart，这是一个很好的开始方式。这个命令创建一个新的Nginx chart——使用你选择的名称，遵循chart布局的最佳实践。由于Kubernetes集群有不同的方法来公开应用程序，这个chart使Nginx公开于网络流量的方式可配置，因此它可以在各种集群中被公开。

create命令为你创建一个chart，其中包含所有必需的chart结构和文件。这些文件具备良好的文档，这有助于你了解所需内容。chart提供的模板展示了多个Kubernetes清单协同工作以部署应用程序的示例。此外，你可以直接安装和测试此chart。

在本章中，我们将介绍一个名为anvil的示例应用程序。这是一个简单的应用程序，它将向你展示chart的结构，并为你提供针对不同的应用程序更改此chart的机会。要创建新chart，请在命令提示符下运行以下命令：

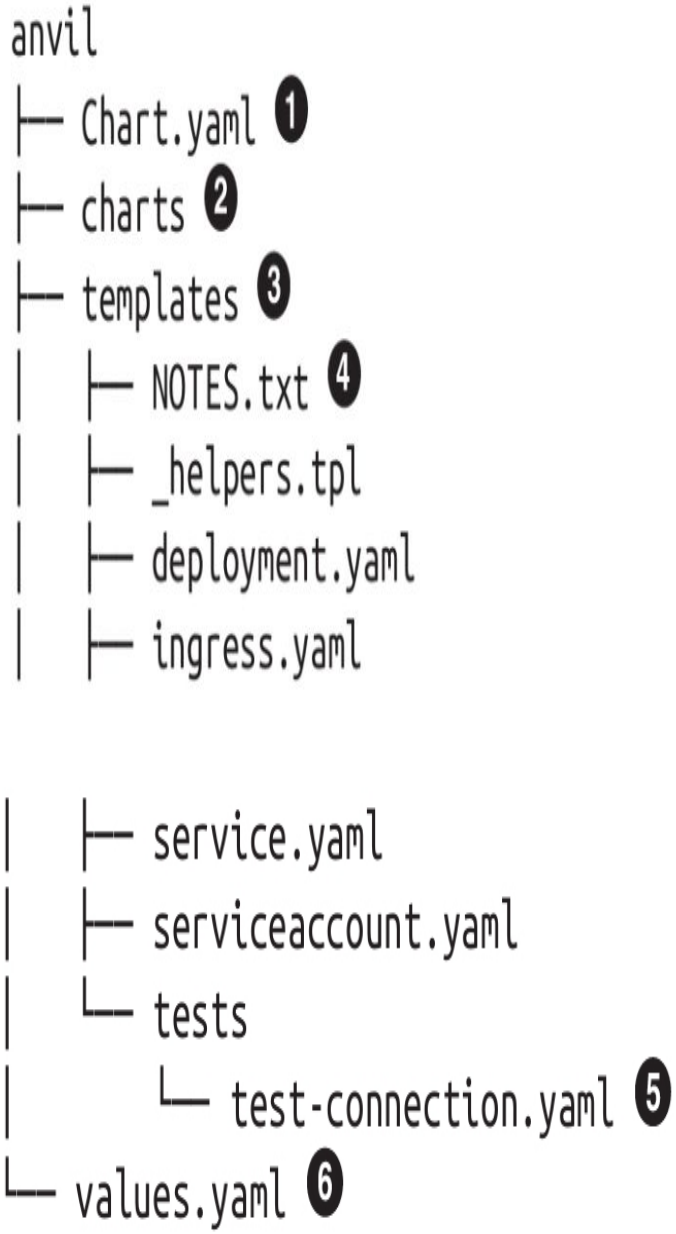
```
$ helm create anvil
```

这将创建一个名为anvil的新chart作为当前目录的子目录。

不同的起点

Nginx是展示chart部分和基本无状态服务的良好起点。但是，如果你经常创建不遵循Nginx模型的chart，那么使用不同的起点会更有帮助。Helm有一个名为starter packs的特性，helm create可以利用这个特性提供一个不同的起点来生成chart。详见第6章。

新chart是一个包含许多文件和文件夹的目录。这里并未包括每个文件和文件夹，在接下来的几章中，你会发现更多的内容。这些都是正常运行chart所需要的基本要素：



❶ 这个Chart.yaml文件包含元数据和chart的一些功能控件。

❷ 依赖的chart可以选择保存在charts目录中。第6章介绍了chart依赖项。目前这将是一个空目录

❸ 用于生成Kubernetes清单的模板存储在templates目录中

❹ 这个NOTES.txt文件是一个特殊的模板。安装chart时，NOTES.txt文件模板是被渲染和显示到（而不是被安装到）集群中。

❺ 模板可以包含未作为install或upgrade命令的一部分安装的测试。此chart包括由helm test命令使用的测试。测试详见第6章

❻ 当Helm渲染清单时传递给模板的默认值位于values.yaml文件中。实例化chart时，可以覆盖这些值。

你可以通过运行以下命令来安装这个新创建的chart，无须任何修改：

```
$ helm install myapp anvil
```

运行此命令时，Helm将创建一个在集群中运行的chart实例，名称为myapp。Helm将使用当前配置的连接和用于Kubernetes的上下文来安装此实例。Helm使用的配置与使用Kubernetes的命令行应用程序kubectl时使用的配置相同。在该命令中，它的最后一个参数anvil是chart所在的目录。

此命令的输出包括通过渲染NOTES.txt模板生成的内容，如下所示：

```
NAME: myapp
LAST DEPLOYED: Sun Apr  5 08:12:59 2020
NAMESPACE: default
STATUS: deployed
REVISION: 1
NOTES:
1. Get the application URL by running these commands:
  export POD_NAME=$(kubectl get pods --namespace default ↵
    -l "app.kubernetes.io/name=anvil,app.kubernetes.io/instance=myapp" ↵
    -o jsonpath="{.items[0].metadata.name}")
  echo "Visit http://127.0.0.1:8080 to use your application"
  kubectl --namespace default port-forward $POD_NAME 8080:80
```

“注释”（NOTES）部分包含有关连接到应用程序的信息。根据实例化时传递到chart中的值，此信息可能会有很大的差别。此chart可以配置为使用ClusterIP、NodePort、LoadBalancer和Ingress来公开应用程序。默认情况下，使用ClusterIP。

如果你按照注释中的说明操作，将看到默认的Nginx网页，显示它正在运行，如图4-1所示。

图4-1：访问正在运行的应用程序时的默认Nginx网页

公开应用程序的方法被绑定到内置的Kubernetes资源类型，而不是应用程序的特性。这使得它们可以移植到你的自定义应用程序中。公开应用程序的方法包括：

ClusterIP（集群IP）

Kubernetes Service资源类型上的配置选项，用于在集群级内部IP地址上公开服务。

NodePort（节点端口）

Kubernetes Service资源的另一种选项，它在每个节点的静态端口上公开服务。也将自动创建一个ClusterIP。

LoadBalancer（负载均衡器）

一个Kubernetes Service配置选项，它使用主机提供商提供的负载均衡器在外部公开应用程序。

Ingress（入口）

入口资源是Service的附加资源，它通过HTTP和HTTPS公开服务。它需要一个入口控制器（如ingress-nginx）才能工作。

如果已将此chart安装到集群中进行测试，则可以通过运行以下命令从集群中删除该实例：

```
$ helm delete myapp
```

 安装chart时，默认情况下，用于Nginx的镜像是Docker官方镜像（https://oreil.ly/YghQP）的最新版本。如果你使用的Kubernetes集群没有访问hub.docker.com的权限，则你将无法安装镜像。你需要将镜像设置为集群可以访问。

现在已经构建了一个可以工作的chart，让我们看看其中的内容，并针对Anvil应用程序修改它。

4.2 Chart.yaml文件

在anvil目录中有一个名为Chart.yaml的文件。这个Chart.yaml文件告诉Helm和其他工具有关此chart的信息。其他工具包括Kubeapps（一个内部目录和应用程序安装程序）、Artifact Hub（一个云本地工件的列表）和许多其他工具。

当你打开Chart.yaml文件时，将看到示例4-1中所示的内容。

示例4-1： 生成的Chart.yaml文件

```
apiVersion: v2 ❶
name: anvil ❷
description: A Helm chart for Kubernetes

# A chart can be either an 'application' or a 'library' chart.
#
# Application charts are a collection of templates that can be packaged into ↵
  versioned archives
# to be deployed.
#
```

```
# Library charts provide useful utilities or functions for the chart developer.
  They're included as
# a dependency of application charts to inject those utilities and functions
  into the rendering
# pipeline. Library charts do not define any templates and therefore cannot be
  deployed.
type: application

# This is the chart version. This version number should be incremented each
  time you make changes
# to the chart and its templates, including the app version.
version: 0.1.0 ❸

# This is the version number of the application being deployed. This version
  number should be
# incremented each time you make changes to the application. Versions are not
  expected to
# follow Semantic Versioning. They should reflect the version the application
  is using.
appVersion: 1.16.0
```

❶ apiVersion告诉Helm此chart使用的是什么结构。apiVerison v2是为Helm v3设计的。

❷ 该名称用于标识不同位置的chart。

❸ chart可以有许多版本。Helm使用版本信息来排序和识别chart。

这个Chart.yaml文件包含许多键，其中只有三个键是必需的。apiVersion属性告诉Helm的chart版本。Helm v3可以处理apiVersion为v1或v2的chart。v1 chart是那些设计用来与以前版本的Helm一起工作的chart。如果你的chart是针对Helm v3或更新的版本设计的，则应该将它设置为v2。name的值通常用作Kubernetes资源名称的一部分。这意味着名称仅限于小写字母数字、“-”和“.”字符，并必须以字母或数字字符开头和结尾。名称通常是小写字母和数字字符。最后一个必需的键是version，它包含chart的版本。版本号应该遵循语义版本控制，这在第2章中有介绍。

你可能会注意到Chart.yaml文件的样式与Kubernetes的文件相似，但略有不同。Chart.yaml文件与自定义资源的格式不同，但确实包含一些相同的属性。原版Chart.yaml文件的设计要追溯到2015年，早在Kubernetes自定义资源定义存在之前。虽然随着时间的推移，Helm的主版本递进了，但它保持了一定的向后兼容性，不会对用户造成太大的干扰。这导致了Chart.yaml和Kubernetes清单文件格式之间的差别。

Chart.yaml文件还包含描述性信息，这些信息在用户界面中显示时非常有用。示例4-1中的description字段就是这样一个字段，但你可以添加其他字段，例如：

· home是指向chart或项目主页的URL。

· icon是URL形式的图像（例如PNG或SVG文件）。

· maintainers包含维护人员列表。列表中的每个维护人员都可以有一个名称、电子邮件和URL。

· keywords可以保存有关项目的关键字列表。

· sources是指向项目或chart源代码的URL列表。

Chart.yaml文件中属性的完整描述见附录A，供参考。

生成的Chart.yaml文件可以针对Anvil应用程序做修改。以下修改更新了所需字段、添加了一些描述性文件并删除了注释：

```
apiVersion: v2
name: anvil
description: A surprise to catch something speedy.
version: 0.1.0
appVersion: 9.17.49
icon: https://wile.example.com/anvil.svg
keywords:
  - road runner
  - anvil
home: https://wile.example.com/
sources:
  - https://github.com/Masterminds/learning-helm/tree/main/chapter4/anvil
maintainers:
  - name: ACME Corp
    email: maintainers@example.com
  - name: Wile E. Coyote
    email: wile@example.com
```

有一个属性tyep在生成的Chart.yaml文件中，但不在Anvil的那个文件中。Anvil是一个应用程序，它是type字段的默认值，因此type字段是不必要的。另一种类型的chart是库chart，这将在第7章中介绍。

appVersion属性是唯一的。它既是描述性的，也是模板中经常使用的。appVersion属性表示主应用程序或组合应用程序的版本。例如，如果被打包的应用程序是WordPress，那么它就是WordPress的版本。

 icon属性是一个URL，可以包含数据URL。数据URL（https://oreil.ly/1gj45）使你能够以URL形式嵌入小文件。如果徽标（logo）是一个小的SVG文件，这一点尤其有用。如果chart可能会在内部环境中运行，或者你不希望用户界面不断地从Web服务器下载文件，那么数据URL是一个有用的选择。

4.3 修改模板

要为Anvil应用程序或你自己的自定义应用程序修改此chart，你需要了解并修改模板。helm create命令创建的模板可以作为无状态应用程序运行Nginx。在我们正在处理的示例中，Nginx需要用Anvil替换。

Helm是用Go编程语言编写的，Go包含模板包。Helm利用文本模板包作为模板的基础。此模板语言与其他模板语言类似，包括循环、if/then逻辑、函数等。YAML文件的示例模板如下：

```
product: {{ .Values.product | default "rocket" | quote }}
```

在这个YAML文件中有一个键名product。该值是使用模板生成的。{{和}}是进入和退出模板逻辑的左括号和右括号。模板逻辑有三部分，用|分隔。这称为管道，其工作方式与基于Unix的系统中的管道相同。左侧函数的值或输出作为最后一个参数传入管道中的下一项。在本例中，管道从.Values.product中属性的值开始。这来自呈渲染模板时传入的数据对象。此数据的值作为default函数（Helm提供的函数）的最后一个参数通过管道传递。如果传入的值为空，则default函数使用默认值"rocket"，以确保存在值。然后将其发送到quote函数，该函数确保在将字符串写入模板之前将其用引号括起来。

位于.Values.product开头的“.”很重要。这被认为是当前作用域中的根对象。.Values是根对象的属性。

4.3.1 部署

Helm chart可以为你可能使用的任何Kubernetes资源类型保存模板。这些资源类型包括StatefulSet、Job、PersistentVolumeClaim、

Service（状态集、作业、持久卷目标、服务）等。使用helm create创建的chart是为了作为Kubernetes Deployment来运行无状态服务。我们在这里为Anvil使用的示例应用程序是一个无状态应用程序，这意味着它可以很好地作为一个部署来使用。

为了理解Deployment模板，我们可以看看在chart的templates目录中的deployment.yaml文件。以下是直到spec部分之前的Deployment模板版本：

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: {{ include "anvil.fullname" . }}
  labels:
    {{- include "anvil.labels" . | nindent 4 }}
```

这看起来非常类似于Kubernetes清单的开头包含apiVersion、kind、meta-data（API版本、种类和元数据）。进入metadata，你会注意到模板开始了。

 如果你不熟悉Kubernetes Deployment的结构，可以在Kubernetes文档（https://oreil.ly/aluIE）中阅读。

include模板函数允许将一个模板的输出包含在另一个模板中，这在管道中也能工作。include函数的第一个参数是要使用的模板的名称。作为第二个参数传入的“.”是根对象。因为根对象被传入，所以根对象上的属性和函数可以在被调用的模板中使用。

anvil.fullname以及anvil.labels是通过_helpers.tpl文件引入的两个可重用模板（名称开头的“_”会使它上升到目录列表的顶部，以便你可以在模板中轻松找到它。Helm不会将它们渲染到Kubernetes清单中，但确实会使其中的模板可用）。anvil.fullname根据实例化

chart时选择的名称提供名称，而anvil.labels提供遵循Kubernetes最佳实践的标签。第5章将更深入地介绍这些功能。

模板的metadata部分之后是spec部分，其内容如下：

```
spec:
  replicas: {{ .Values.replicaCount }}
  selector:
    matchLabels:
      {{- include "anvil.selectorLabels" . | nindent 6 }}
  template:
    metadata:
      labels:
        {{- include "anvil.selectorLabels" . | nindent 8 }}
    spec:
    {{- with .Values.imagePullSecrets }}
      imagePullSecrets:
        {{- toYaml . | nindent 8 }}
    {{- end }}
      serviceAccountName: {{ include "anvil.serviceAccountName" . }}
      securityContext:
        {{- toYaml .Values.podSecurityContext | nindent 8 }}
      containers:
        - name: {{ .Chart.Name }}
          securityContext:
            {{- toYaml .Values.securityContext | nindent 12 }}
          image: "{{ .Values.image.repository }}:{{ .Values.image.tag | default↵
            .Chart.AppVersion }}" ❶
        imagePullPolicy: {{ .Values.image.pullPolicy }}
        ports:
          - name: http
            containerPort: 80
            protocol: TCP
        livenessProbe:
          httpGet:
            path: /
            port: http
        readinessProbe:
          httpGet:
            path: /
            port: http
        resources:
          {{- toYaml .Values.resources | nindent 12 }}
    {{- with .Values.nodeSelector }}
    nodeSelector:
      {{- toYaml . | nindent 8 }}
    {{- end }}
  {{- with .Values.affinity }}
    affinity:
      {{- toYaml . | nindent 8 }}
```

```
{{- end }}
{{- with .Values.tolerations }}
  tolerations:
    {{- toYaml . | nindent 8 }}
{{- end }}
```

❶ 容器镜像的位置和版本可以通过值进行配置。

spec部分完成部署。本节的大部分内容是用.Values上的属性填充数据。有一些元素是硬编码的，例如用于公开应用程序的端口。Anvil在端口80上通过HTTP公开，所以我们不需要更改端口。如果容器公开在不同的端口上，则需要在此处进行更改。

容器的image值是使用值设置的。你在这里找不到硬编码镜像的位置。这对于在实例化chart时需要将镜像位置设置为其他位置的情况非常有用。这意味着我们需要更改默认值中的位置。

.Values上的属性是基于许多因素计算的。默认值和起点基于在chart的values.yaml文件（该文件见4.4节）中的输入。这些值可以被实例化chart时传入的值覆盖。helm CLI具有直接传入值的标志（即--set、--set-file和--set-string）或传递带有值（即，-f或--values）的文件的标志。这些值被合并在一起，后传入的值优先采用。

模板是一个很宏大的主题，通常占chart的大部分。第5章专门讨论模板。

4.4 使用values文件

当有人从chart实例化Kubernetes集群中的应用程序时，他们不需要提供模板中使用的所有值。但确实需要这样做的话将会是一个糟糕的用户体验。这就是values.yaml文件的用途所在。

chart在存放Chart.yaml文件的根目录中包括一个values.yaml文件。这个values.yaml文件包含chart使用的默认值，它是可以传递到chart中的自定义值的文档形式。

values.yaml是一个非结构化的YAML文件。稍后将介绍一些常见的和有用的实践，但是YAML的格式中不需要任何东西。这使得chart创建者能够提供适合他们的结构和信息。values.yaml文件可以包含许多内容，从对Kubernetes清单属性的简单替换到特定于应用程序的业务逻辑所需的元素。

4.4.1 容器镜像

helm create创建的values.yaml文件的开口部分包含镜像信息以及一些打开的文档和副本信息：

```
# Default values for anvil.
# This is a YAML-formatted file.
# Declare variables to be passed into your templates.

replicaCount: 1

image:
  repository: ghcr.io/masterminds/learning-helm/anvil-app ❶
  pullPolicy: IfNotPresent ❷
  # Overrides the image tag whose default is the chart version.
  tag: "" ❸

imagePullSecrets: [] ❹
```

❶ 镜像的位置。它已经被更新用来反映Anvil的位置。

❷ IfNotPresent策略意味着镜像将按所使用的版本缓存在Kubernetes集群中。Always是另一个绕过缓存并始终从存储库下载的选项。

❸ 默认情况下，此chart使用appVersion作为标记。如果指定了镜像标记，则使用它而不是appVersion。

❹ 当需要凭据来访问受用户名和密码保护的容器登记站位置时，将使用pull secret列表。

此chart和值表示绑定为单个镜像的应用程序。values.yaml文件中使用的模式的设计考虑到了这一点。例如，只有一个镜像位置。如果你的应用程序有多个镜像，则每个镜像都会有一个包含此处大部分信息的部分。这包括replicaCount，它是Kubernetes在创建Deployment时将使用的副本（replicas）数。

image部分包含有关镜像的详细信息。repository包含要使用的镜像的位置，而pullPolicy告诉Kubernetes获取或缓存镜像的频率。如果使用了移动标记（如stable），则pullPolicy应设置为Always，以便获取修改的镜像。由于正在使用某个版本，因此默认的pullPolicy设置为IfNotPresent，以便可以使用缓存的版本（如果可用）。tag属性提供了一个机会，可以设置一个不同于Chart.yaml文件中设置的appVersion的标记。

你可能注意到在获取镜像时没有方法来设置摘要。当镜像位于不同的存储库中时，其摘要可能不同。例如，如果将Anvil镜像从Docker Hub复制到另一个镜像存储库Quay中，那么即使标签和内容保持不变，对于同一个镜像，摘要也会发生变化。第5章提供了一个示例，如果需要的话，可以添加对chart摘要的支持。

如果需要从带有访问控制的容器登记站中拉取（pull）镜像，则Kubernetes需要知道如何做到这一点。这是通过使用pull secret来实现的。imagePullSecrets允许你列出访问私有登记站的pull secret的名称。请参考文档（https://oreil.ly/BL-VO）以创建pull secret。

生成的chart内置了一些安全考虑事项，可以启用或以其他方式进行配置。默认情况下，会为chart实例创建一个服务账户，而其他选项是opt-in。以下是helm create命令生成的内容：

```
serviceAccount:
  # Specifies whether a service account should be created
  create: true
  # Annotations to add to the service account
  annotations: {}
  # The name of the service account to use.
  # If not set and create is true, a name is generated using the fullname ↵
    template
  name:

podSecurityContext: {}
  # fsGroup: 2000
securityContext: {}
  # capabilities:
  #   drop:
  #   - ALL
  # readOnlyRootFilesystem: true
  # runAsNonRoot: true
  # runAsUser: 1000
```

你将注意到，配置中的大多数属性都是注释，并且处于非活动状态。当值作为注释渲染chart时，这些属性没有值。值为空。通过将结构和值作为注释，chart记录了可以使用的结构和默认值，但没有启用这些功能。

4.4.2 公开服务

values.yaml文件的下一部分负责将应用程序公开给其他人使用：

```
service:
  type: ClusterIP
  port: 80

ingress:
  enabled: false
  annotations: {}
    # kubernetes.io/ingress.class: nginx
    # kubernetes.io/tls-acme: "true"
  hosts:
    - host: chart-example.local
      paths: []
  tls: []
  #  - secretName: chart-example-tls
  #    hosts:
  #      - chart-example.local
```

在Kubernetes中，有两个内置对象可用于公开应用程序。首先是Service。service属性允许你选择正在使用的Service的类型。虽然默认情况下使用ClusterIP，但也可以使用NodePort和LoadBalancer等其

他选项。service部分的几行YAML与生成的service.yaml模板配对，以创建完整的服务清单来上传到Kubernetes。

第二个内置对象是Ingress清单，它可以与Service配对，并且chart能够生成它们。Ingress配置提供了一种显示chart中常见模式的方法：使用enabled属性来打开和关闭功能。在本例中，ingress.enabled设置为false。当Helm渲染模板并看到值为false时，将跳过Ingress清单。这是由于在生成的ingress.yaml文件的Ingress模板中使用了一个if逻辑语句。

入口控制器

对于功能性入口设置，你需要的不仅仅是Kubernetes中的Ingress资源。可以包含在chart中的Ingress资源将入口控制器连接到一个Service。你需要在集群中运行一个入口控制器，因为默认情况下不包括入口控制器（Ingress Controller）。Kubernetes社区提供Nginx Ingress Controller（https://oreil.ly/vc3ed），这是一个很好的默认选项。

4.4.3 资源限制

在生产环境中运行应用程序时，最好设置资源限制。例如，这可以防止一个容器中的内存泄漏中断其他容器。当chart作者创建其他人将要使用的chart时，他们可能不知道该chart将安装在何处以及有多少资源可用。开发人员或者测试chart的人可以把它安装在笔记本电脑上吗？它是否可以安装在大型生产服务器上？为了处理环境中的这种差异，我们建议设置资源限制，然后将它们转换为注释。这可以在values.yaml文件的下一部分中找到：

```
resources: {}
  # We usually recommend not to specify default resources and to leave this as
  # a conscious choice for the user. This also increases chances charts run on
  # environments with little resources, such as Minikube. If you do want to
  # specify resources, uncomment the following lines, adjust them as necessary,
  # and remove the curly braces after 'resources:'.
  # limits:
  #   cpu: 100m
  #   memory: 128Mi
  # requests:
  #   cpu: 100m
  #   memory: 128Mi
```

安装应用程序的人在实例化chart时会参考这些数字。这些数字是在生成Nginx时为其设置的默认值。它们适用于Anvil应用程序。如果应用程序需要不同的值，则需要更新这些值。

工作负载能够通过设置节点选择器、容错性和关联性来指定在集群中执行它们的具体位置。尽管这些更高级的特性通常不被使用，但是将它们包含在chart中对于需要它们的人来说是一个好主意。生成的values.yaml文件和模板会考虑到这一点。下面的示例为这些高级功能生成了YAML密钥。默认情况下，这些值为空，需要安装chart的人员会为其安装设置适当的值：

```
nodeSelector: {}

tolerations: []

affinity: {}
```

4.5 打包chart

你可以将chart的文件和目录打包到单个归档文件中。这很有用，原因很多，包括：

· 分发给他人。软件包管理器的一个强大方面是，懂得运行应用程序的人可以将其打包，以便其他对平台或应用程序不熟悉的人可以运行它。

· 当应用程序的一个版本需要通过一个多环境测试过程时。多环境测试过程的一个例子：有开发、质量保证（QA）和生产环境，应用程序在投入生产之前需要通过QA。

· 开发多服务应用程序时，开发人员需要运行由他人构建或以其他方式处理的服务，作为其开发设置的一部分。

在上述几种情况下，为chart传递单个文件通常比传递目录结构更简单。

chart版本给你发布和使用chart的方式带来了另一个麻烦。你或使用你的chart的人可能需要不同版本的chart。这就是使用chart存储库或开放容器倡议（OCI）登记站来存储和共享不同版本的原因（见第7章）。在这些环境中，在每个版本的目录结构集合中存储和共享许多文件绝非易事。

Helm有能力建立一个chart归档文件。每个chart归档都是一个扩展名为.tgz的gzip TAR文件。任何可以创建、提取和处理gzip TAR文件的工具都可以使用Helm的chart归档文件。

当Helm生成归档文件时，它们将使用chart name-version.tgz的模式进行命名，Helm在使用它们时也会期待有同样的模式。chart name是你将在Chart.yaml文件中找到的名称，而version是chart版

本。这使得同一个chart的多个版本可以彼此并排存储。你可以通过运行以下命令将Anvil打包为归档文件：

```
$ helm package anvil
```

在本例中，anvil是指向anvil chart源所在位置的路径。默认情况下，helm package命令将把归档文件放在运行该命令时所在的目录中。

在打包chart文件时，可以使用一些有用的标志：

--dependency-update(-u)

告诉Helm在创建存档文件之前更新依赖的chart。这将更新Chart.lock文件并将依赖的chart的副本归档并放置在chart目录中。第6章更详细地介绍了依赖项。

--destination(-d)

允许你设置用于放置chart归档文件的位置（如果它与当前工作目录不同）。

--app-version

可用于设置Chart.yaml文件的appVersion属性。如果为容器中运行的应用程序的每个新版本都创建chart的新版本，并且对chart没有其他更改，则这一点特别有用。自动化可以在构建新版本的过程中使用这样的标志。

--version

更新chart的版本。如果将命令行用作打包chart的过程的一部分来更新appVersion，那么这将非常有用。

Pretty Good Privacy（PGP）签名chart标志

Helm chart可以被加密签名和验证。package命令具有对过程部分进行签名的标志，而install和upgrade等命令具有对过程部分进行验证的标志。第6章介绍了这个过程。

有时，chart目录中有一些不希望包含在chart归档文件中的文件。在chart目录中可以有一个可选的.helmignore文件。这类似于Git的.gitinore文件。前面使用的helm create命令创建了一个包含以下内容的命令：

```
# Patterns to ignore when building packages.
# This supports shell glob matching, relative path matching, and
# negation (prefixed with !). Only one pattern per line.
.DS_Store
# Common VCS dirs
.git/
.gitignore
.bzr/
.bzrignore
.hg/
.hgignore
.svn/
# Common backup files
*.swp
*.bak
*.tmp
*.orig
*~
# Various IDEs
.project
.idea/
*.tmproj
.vscode/
```

其中许多扩展和模式看起来都很熟悉，因为它们来自各种版本控制系统和代码编辑器。

创建chart归档文件时，通常不希望包含版本控制系统数据之类的元素。

.helmignore文件提供了一个位置来指定要跳过的内容。此文件必须位于chart的最上层。

Helm被设计用于处理归档文件，就像处理目录结构一样。helm install和helm lint等命令可以像传递目录一样传递归档文件，很快就会介绍这一点。

4.6 校验chart代码

在开发chart时，尤其是使用YAML模板时，很容易出错或遗漏某些内容。为了帮助你捕捉错误、bug、样式问题和其他可疑元素，Helm客户端包括了一个代码校验器（linter）。这种代码校验器可以在chart开发过程中使用，也可以作为任何测试过程的一部分来使用。

要使用代码校验器，请将chart上的lint命令用于chart目录或打包的归档文件：

```
$ helm lint anvil
==> Linting anvil


1 chart(s) linted, 0 chart(s) failed
```

第一行是你运行的命令，而下面的行是Helm的输出。在这个例子中没有查出问题。你可以对归档文件使用此命令，就像上一节中一样。为此，将anvil参数（设置为chart的目录位置）更改为归档文件anvil-0.1.0.tgz。

此命令可以在一个命令中校验多个chart。例如，如果你有第二个名为mychart的chart，并希望将其与anvil一起校验，则可以运行以下命令：

```
$ helm lint anvil mychart
```

Helm提供的有关chart的可操作反馈的三个级别是信息、警告和错误。信息级反馈是信息性的。chart可以与信息级反馈一起安装。信息级反馈使Helm的退出代码为0。错误级反馈表示chart有问题。如果chart为Kubernetes生成了无效的清单，例如YAML无效，则Helm将生成一个错误。错误导致Helm有一个非零的退出代码，这有助于在自动化的测试工具中捕获问题。中间级别是警告信息。这些消息涉及可能导致问题的发现。默认情况下，警告消息使Helm的退出代码为0，但Helm添加了一个--strict标志，使退出代码不为零。你可以选择如何在自动化工具中处理这些反馈。

退出代码

当应用程序退出时，它向执行它的父级提供代码或状态（https://oreil.ly/zHz8g）。运行Helm时，执行它的父级通常是操作系统、命令提示符或shell。退出状态为零意味着应用程序退出时没有任何问题。非零退出状态意味着有问题。自动化的测试系统通常使用退出代码来知道何时继续或停止。通常，当自动化的测试中使用的应用程序返回一个非零退出代码时，自动化的过程会结束，并向人们通知出现了错误。

在本例中，没有发现anvil chart的问题。由helm create创建的默认chart将有一条关于Chart.yaml文件中缺少icon属性的信息级消息。这是一个信息级别的通知，以便人们知道它丢失了。缺少的图标不会影响chart的操作，但会影响它在用户界面中的显示方式。

4.7 结论

使用helm create命令为应用程序创建一个简单的chart非常简单。即使你的应用程序更复杂，chart的结构也可以容纳它们，helm create命令可以帮助你。利用本章中介绍的内容，只需做一些小的修改就可以使用helm install安装Anvil chart，并查看在集群中运行的自定义应用程序。你可以使用相同的流程来创建自己的chart。

在下一章中，你将学习如何创建模板，重点介绍模板语言的工作原理以及如何将其应用于存储在chart中的Kubernetes模板。模板通常是chart中花费时间最多的部分。创建模板时，了解你可以使用什么将使开发过程更快、更容易。

第5章

开发模板

模板是Helm chart的核心，它们构成了chart的大部分文件和内容。这些是位于templates目录中的文件。当你运行helm install和helm upgrade等命令时，Helm将渲染模板并将它们发送给Kubernetes。如果使用helm template命令，模板将被渲染并显示为输出（即，发送到标准输出）。

模板引擎支持多种方法来构建模板。在简单的情况下，你可以采用用户传入的值或来自values.yaml文件的值替换Kubernetes清单YAML文件中的值。在更复杂的情况下，可以将逻辑构建到模板中，以简化chart使用者需要输入的内容。或者，你可以构建能配置应用程序本身的功能。

在本章中，你将学习如何开发模板，并了解模板语法的工作原理。我们还将介绍Helm添加到模板中的一些很酷的特性，这些特性使你能够使用YAML和Kubernetes进行交互。在此过程中，我们将研究一些可以应用于自己的模板的模式。

5.1 模板语法

Helm使用Go标准库提供的Go文本模板引擎。该语法用于kubectl（Kubernetes的命令行应用程序）模板、Hugo（静态站点生成器）和许多其他内置于Go中的应用程序。Helm中使用的模板引擎用于处理各种类型的文本文件。

开发模板不需要了解Go编程语言。模板引擎中有一些Go元素，但是如果你不了解Go，可以将它们视为模板语言的细微差别。当学习开发模板时，我们将调用它们。

为什么使用Go的模板引擎？

当Helm在开发过程中需要一个模板引擎时，Go标准库中提供的模板引擎是最成熟、最稳定的选择。此外，这个模板引擎有一个安全模型，由Google使用安全策略进行维护。这是最好的选择。

从那时起，更多的通用模板引擎被提供给Go。Helm项目也支持其他模板引擎，并且几年来一直有一个代码扩展点，可以在那里添加它们。在那段时间里，人们对其他模板系统的兴趣微乎其微，几乎没有人支持模板系统。

Go模板语法与其他系统的语法相似，能够满足Helm用户的需求。

5.1.1 动作

逻辑、控制结构和数据计算用{{和}}包装。这些被称为动作（action）。动作之外的任何内容都将被复制到输出。

当花括号用于启动和停止动作时，可以用-来删除前导或尾随空格。下面的示例说明了这一点：

```
{{ "Hello" -}} , {{- "World" }}
```

生成的输出是“Hello, World”。空格已经从带有-的那一方删除，直到下一个非空格字符为止。在（用于删除空格的）-和动作的其余部分之间需要有一个ASCII空格。例如，{{-12}}的计算结果是-12，因为-被认为是数字的一部分，而不是括号的一部分。

在动作中，你可以利用各种各样的特性，包括管道、if/else语句、循环、变量、子模板和函数。结合使用这些工具提供了编写模板的强大方法。

5.1.2 Helm传递到模板的信息

当Helm渲染一个模板时，它会将一个数据对象传递给模板，其中包含你可以访问的信息。在模板内部，此对象表示为“.”（即，一个句点）。它被称为点。这个对象有各种各样的可用信息。

在第4章中，你已经看到了如何把values.yaml文件中的值作为.Values上的属性使用。.Values上的属性特定于每个chart，完全基于values.yaml文件和传递到chart中的值。.Values上的属性没有模式（schema），并且因chart而异。

除了这些值之外，有关发布的信息（如第2章所述）还可以作为.release的属性访问。此信息包括：

.Release.Name

发布版本的名称。

.Release.Namespace

包含将chart发布到的命名空间。

.Release.IsInstall

当发布版本是一个正在安装的工作负载时，设置为true。

.Release.IsUpgrade

当发布版本是一个升级或回滚时，设置为true。

.Release.Service

列出执行发布的服务。当Helm安装chart时，此值设置为"Helm"。建立在Helm之上的不同的应用程序可以将其设置为自己的值。

Chart.yaml文件中的信息也可以在.Chart的数据对象上找到。此信息确实遵循Chart.yaml文件的模式。包括：

.Chart.Name

包含chart的名称。

.Chart.Version

chart的版本。

.Chart.AppVersion

应用程序版本（如果已设置）。

.Chart.Annotations

包含注解的键/值列表。

Chart.yaml文件中的每个属性都是可访问的。名字的不同之处在于它们在Chart.yaml中以小写字母开头，但当它们是.Chart对象的属性时，以大写字母开头。

如果要将Chart.yaml文件中的自定义信息传递到模板，需要使用注解。.Chart对象只包含模式中的Chart.yaml文件中的字段。不能添加新字段来传递它们，但可以将自定义信息添加到注解中。

大写属性名

传递到模板中的数据对象的属性名以大写字母开头。这是Helm用Go编程语言编写的产物。在Go中，公共属性以大写字母开头，私有属性以小写字母开头。当访问数据对象时，你只需要记住第一个字母是大写的。

不同的Kubernetes集群可以具有不同的功能。这可能取决于你使用的Kubernetes的版本，或者是否安装了CRD。Helm提供了一些关于集群能力的数据作为.Capabilities的属性。Helm询问你要将应用程序部署到的集群以获取此类信息。这些信息包括：

.Capabilities.APIVersions

包含集群中可用的API版本和资源类型。

.Capabilities.KubeVersion.Version

完整的Kubernetes版本。

.Capabilities.KubeVersion.Major

包含Kubernetes的主版本。因为Kubernetes没有增加主版本，所以设置为1。

.Capabilities.KubeVersion.Minor

集群中使用的Kubernetes的次要版本。

当使用helm template时，Helm不会像使用helm install或helm upgrade时那样询问集群。运行helm template时提供给正在处理的模板的功能信息是Helm已经知道的关于兼容Kubernetes集群的默认信息。Helm之所以这样工作，是因为template命令只用于处理模板，并且这样做不会意外地从配置的集群泄露信息。

chart可以包含自定义文件。例如，你可以将配置文件作为chart中的文件，通过ConfigMap或Secret传递给应用程序。chart中未在.helmignore文件中列出的非特殊文件可以在模板的.Files中找到。这将不允许你访问模板文件。

.helmignore文件

你可以在chart目录中包含不希望打包到chart归档中的文件，以及包含不希望被Helm或chart使用的文件。可以在chart中的Chart.yaml文件所在的根目录下的.helmignore文件中列出这些文件。

.helmignore文件类似于源代码管理系统Git中的.gitignore文件。可以列出要忽略的单个文件、目录和文件模式。运行helm create生成新chart时，它包含一个.helmignore文件，该文件将忽略常见的源代码管理系统和编辑器文件。

传递到模板中的最后一块数据是有关当前正在执行的模板的详细信息。Helm传入：

.Template.Name

包含模板的命名空间文件路径。例如，在第4章的anvil chart中，路径是anvil/templates/deployment.yaml。

.Template.BasePath

当前chart的templates目录的命名空间路径（例如，anvil/templates）。

在本章后面，你将学习在某些情况下如何更改“.”的作用域（scope）。当作用域发生更改时，一些属性（如.Capabilities.KubeVersion.Minor）在该位置将是不可访问的。当模板执行开始时，“.”被映射到$，并且$不变。即使作用域发生变化，$.Capabilities.KubeVersion.Minor和其他传入的数据仍然是可访问的。你会发现$通常只在作用域发生更改时使用。

现在你已经了解了传递到模板中的数据，我们将研究如何在模板中使用和操作这些数据。

5.1.3 管道

管道是一系列链接在一起的命令、函数和变量。变量的值或函数的输出用作管道中下一个函数的输入。管道的最后一个元素的输出就是管道的输出。下面是一个简单的管道：

```
character: {{ .Values.character | default "Sylvester" | quote }}
```

该管道有三部分，每个部分之间用|隔开。第一部分是.Values.character，它是character的计算值。这是values.yaml文件或helm install、helm upgrade或helm template渲染chart时传入的值。此值作为最后一个参数传递给default函数。如果该值为空，default将使用值“Sylvester”代替它。default的输出作为一个输入传递给quote，用于将值用引号括起来。quote的输出是从动作返回的。

管道是一个强大的工具，可以用来转换模板中所需的数据。它们可以用于多种目的，从创建强大的转换到防止简单的bug。你能在下面的YAML输出中发现bug吗？

```
id: 12345e2
```

id的值看起来像字符串，但实际上不是。唯一的字母是e，其余的都是数字。YAML解析器（包括Kubernetes使用的解析器）将把它解释为科学记数法中的一个数字。这将导致错误。当你从Git获得摘要或提交ID的缩短版本时，像这样的短字符串是常见的输出。一个简单的解决方法是将值包含在引号中：

```
id: "12345e2"
```

当值用引号括起来时，YAML解析器将把它解释为一个字符串。在这种情况下，在管道末端使用quote函数可以修复或避免bug。

Unix管道

在Unix和类Unix系统（如Linux）中，管道是一个应用程序的输出被用作下一个应用程序的输入的地方。多个各自做一件事的应用程序可以使用它们的输入和输出作为接口链接在一起。

管道源于Douglas McIlroy，后来被Ken Thompson纳入Unix哲学，Ken Thompson致力于设计和实现原始Unix操作系统。Unix哲学中的两个原则包括“使每个程序都能很好地完成一件事”和“期望每个程序的输出成为另一个（目前还不知道的）程序的输入”。

Ken Thompson和Unix团队的另一名成员Rob Pike是Go编程语言原创者中的两位。

5.1.4 模板函数

在动作和管道中可以使用模板函数。本章前面已经介绍过default和quote函数。函数提供了一种方法，可以将数据转换为需要渲染的格式，或者在不存在数据的情况下生成数据。

大多数函数都是由Helm提供的，在构建chart时非常有用。这些函数包括简单的函数（如用于缩进输出的indent和nindent函数）和复杂的函数（如能够深入集群并获取有关当前资源和资源类型的信息的函数）。

Sprig库

Helm模板中的许多函数都是由库Sprig（https://oreil.ly/fBfwm）提供的。Helm的作者将这些函数与Helm一起开发，并考虑了chart用例。它们被放在一个单独的库中，其他应用程序也可以使用它们。

如果你的基于Go的应用程序需要适用的函数，需要在函数中查找问题并希望报告或修复它，或者希望将自己的函数贡献给Helm，那么这个库将非常有用。

为了说明函数的作用，我们可以看看chart中用于提高可读性的常用模式。当helm create运行时，正如你在第4章中看到的，

Kubernetes Deployment模板将作为chart的一部分创建。Deployment模板包括一个安全上下文（security context）部分：

```
securityContext:
  {{- toYaml .Values.podSecurityContext | nindent 8 }}
```

 第4章的完整chart参见https://github.com/Masterminds/learning-helm/tree/main/chapter4/anvil。

在values.yaml文件中，有一个用于podSecurityContext的YAML条目。这就是securityContext的Deployment的template部分中传递的确切YAML。在内部，values.yaml文件中的模板信息不再是YAML。相反，它是一个数据对象。toYaml函数可将数据转换为YAML。

securityContext下面的YAML需要正确缩进，否则部署的清单将由于某个部分没有正确缩进而出现YAML错误。缩进是通过使用两个函数来实现的。在toYaml的左边，-与{{一起使用，以删除前一行中到:为止的所有空格。toYaml的输出被传递给nindent。此函数在接收到的文本的开头添加一个新行，然后缩进每一行。

为了便于阅读，使用nindent代替indent函数。indent函数不会在开头添加换行符。使用nindent可以使securityContext下的YAML位于新行上。这是模板中的另一个常见模式。

 除了toYaml之外，Helm还有一些函数。例如，toJson和toToml可以将数据分别转换为JSON和TOML。toYaml通常用于创建Kubernetes清单，而toJson和toToml更常用于创建配置文件，它们通过Secret和ConfigMap传递给应用程序。

传入函数的参数的顺序是有意安排的。使用管道时，一个函数的输出作为最后一个参数传递给管道中的下一个函数。在上一个示例

中，toYaml的输出作为最后一个参数传递给nindent，后者接受两个参数。函数的参数顺序是为常见的管道用例设计的。

在模板中有一百多个函数（https://oreil.ly/Xtoya）可供使用。这些函数包括处理数学、字典和列表、反射、散列生成、日期函数等。

5.1.5 方法

到目前为止，你已经看到了模板函数。Helm还包括检测Kubernetes集群功能的函数和处理文件的方法。

.Capabilities对象具有方法.Capabilities.APIVersions.Has，它接受一个参数，此参数为要检查其存在性的Kubernetes API或类型。它返回true或false来显示该资源在集群中是否可用。你可以检查组和版本（如batch/v1）或资源类型（如apps/v1/Deployment）。

 在处理自定义资源定义和Kubernetes资源类型的多个版本时，检查资源和API组是否存在非常有用。当Kubernetes API版本从alpha、beta版本到发布版本时，你希望使用资源类型的最新版本，因为alpha和beta版本已被弃用并从Kubernetes中删除。如果你的应用程序将安装在各种Kubernetes版本上，那么在所有这些集群中支持API版本是非常有用的。

当使用helm template时，Helm将为兼容的Kubernetes集群使用一组默认的API版本，而不是与集群交互来生成已知的功能。

.Files中也包含一些方法来帮助你使用文件：

.Files.Get name

提供以字符串形式获取文件内容的方法。在本例中，name是包含chart根目录下的文件路径的名称。

.Files.GetBytes

类似于.Files.Get，但是文件不是以字符串形式返回，而是以字节数组的形式返回。在Go术语中，这是一个字节分片（即[]byte）。

.Files.Glob

接受glob模式并返回另一个files对象，该对象仅包含其名称与该模式匹配的文件。

.Files.AsConfig

获取一个文件组，并将其作为适合包含在Kubernetes ConfigMap清单的data部分的纯YAML返回。这在与.Files.Glob匹配时非常有用。

.Files.AsSecrets

类似于.Files.AsConfig.它不返回纯YAML，而是以一种可以包含在Kubernetes Secret清单的data部分的格式返回数据。它是Base64编码的。这在与.Files.Glob匹配时非常有用。例如，{{.Files.Glob("mysecrets/**").AsSecrets}}。

.Files.Lines

有一个文件名参数，并将文件内容作为一个按换行符（即\n）拆分的数组返回。

为了说明这些方法的用法，下面的模板来自一个example chart。它读取chart的config子目录中的所有文件，并将这些文件都嵌入一个Secret中：

```
apiVersion: v1
kind: Secret
metadata:
  name: {{ include "example.fullname" . }}
type: Opaque
data:
{{ (.Files.Glob "config/*").AsSecrets | indent 2 }}
```

如以下来自Helm的输出示例所示，在此文件中，config子目录中的每个文件都可以在其对应的键的位置找到：

```
apiVersion: v1
kind: Secret

metadata:
  name: myapp
type: Opaque
data:
  jetpack.ini: ZW5hYmxlZCA9IHRydWU=
  rocket.yaml: ZW5hYmxlZDogdHJ1ZQ==
```

5.1.6 在chart中查询Kubernetes资源

Helm包含一个模板函数，使你能够在Kubernetes集群中查找资源。lookup模板函数可以返回单个对象或对象列表。当执行与集群不交互的命令时，此函数返回空响应。

以下示例在anvil命名空间中查找名为runner的Deployment，并使元数据注解可用：

```
{{ (lookup "apps/v1" "Deployment" "anvil" "runner").metadata.annotations }}
```

lookup函数可传递四个参数：

API版本

这是任意对象的版本，无论是包含在Kubernetes中还是作为附加组件的一部分安装。例如，"v1"和"apps/v1"。

对象的种类

这可以是任意资源类型。

从中查找对象的命名空间

可以将其留空以查找你有权访问的所有命名空间或全局资源，例如，Namespace。

要查找的资源名称

可以将其留空以返回资源列表，而不是特定的资源列表。

当返回一个资源列表时，你需要遍历结果以访问每个对象上的数据。如果查找对象则返回dict，查找对象列表则返回list。这是Helm提供的两种不同类型的模板。

返回列表时，对象位于items属性上：

```
{{ (lookup "v1" "ConfigMap" "anvil" "").items }}
```

这些条目可以使用循环进行迭代，详见本章后面。本例返回anvil命名空间中的所有ConfigMaps，假设你可以访问该命名空间。

使用这个函数时要小心。例如，作为试运行的一部分而不是运行升级使用时，它将返回不同的结果。试运行不会与集群交互，因此该函数不会返回任何结果。当运行升级时，它将返回结果。

在不同的集群中安装或升级时返回的结果也可能不同。例如，在开发环境和生产环境中，安装在集群中的资源将存在差异，这也可能导致不同的响应。

5.1.7 if/else/with

Go模板中包含if语句、else语句以及with语句。if和else的工作方式与大多数编程语言相同。为了说明if语句，我们可以查看使用第4章中介绍的helm create命令生成的chart中的模式。在那个chart里，values.yaml文件包含一个启用属性的ingress部分。它看起来如下所示：

```
ingress:
  enabled: false
```

在为Kubernetes创建Ingress资源的ingress.yaml文件中，第一行和最后一行是实现这一功能的if语句：

```
{{- if .Values.ingress.enabled -}}
...
{{- end }}
```

在这种情况下，if语句计算if语句后面的管道输出是true还是false。如果它是true，if语句包含的内容将被计算。为了知道块的结尾在哪里，需要一个end语句。这一点很重要，因为缩进或更典型的括号可能是你要渲染的材料的一部分。

使用if语句是实现通用的enabled模式的典型方式。

如果if语句的计算结果为false，则可以执行if语句中的else语句。下面的示例在未启用Ingress时打印YAML注释：

```
{{- if .Values.ingress.enabled -}}
...
{{- else -}}
# Ingress not enabled
{{- end }}
```

有时，你需要在if语句中通过将多个元素与and或or语句组合来计算这些元素。在模板中，这可能与你的习惯用法稍有不同。考虑模板中的以下片段：

```
{{- if and .Values.characters .Values.products -}}
...
{{- end }}
```

在本例中，and被实现为具有两个参数的函数。这意味着and在两个条目之前使用。与and一样，or也作为一个函数实现。

当与and或or一起使用的元素之一是函数或管道时，可以使用括号。在下面的例子中，or函数有一个参数是一个相等检查：

```
{{- if or (eq .Values.character "Wile E. Coyote") .Values.products -}}
...
{{- end }}
```

使用eq函数实现的相等检查的输出作为第一个参数传递给or。圆括号使你能够将元素分组在一起以构建更复杂的逻辑。

with类似于if，注意with块中的作用域发生了更改。为了继续使用Ingress的示例，下面的块显示了作用域更改：

```
{{- with .Values.ingress.annotations }}
annotations:
  {{- toYaml . | nindent 4 }}
{{- end }}
```

如果传入with的值为空，则跳过该块。如果该值不为空，则执行该块并返回该值。块中的“.”的值
是.Values.ingress.annotations，在这种情况下，块中的作用域已更改为用with检查的值。

使用with检查值然后使用toYaml和nindent函数将其发送到输出的模式，对于values.yaml文件中要直接输出到模板中的元素很常见。这经常用于镜像pull secret、节点选择器等。

就像if语句一样，with也可以附带一个else块，当值为空时可以使用它。

5.1.8 变量

在模板中，你可以创建自己的变量，并使用它们作为参数传递给函数、打印输出等。变量以$开头并具有类型。一旦创建了某种类型的变量（如字符串），就不能将该值设置为另一种类型（如整数）。

创建和初始化变量具有特殊的语法：使用:=符号，如下面的示例所示：

```
{{ $var := .Values.character }}
```

在本例中，将创建一个新变量$var，并将.Values.character分配给它。此变量可用于其他地方，例如：

```
character: {{ $var | default "Sylvester" | quote }}
```

$var的值被传递给default。这与在本章前面传
递.Values.character的方式相同。

创建具有初始值的变量的方法与更改现有变量的值的方法不同。将新值赋给现有变量时需使用=符号。例如：

```
{{ $var := .Values.character }}
{{ $var = "Tweety" }}
```

在本例中，变量在另一个动作中被更改。变量在模板执行的生命周期内一直存在，并且在模板后面的相同动作或不同动作中可用。

 变量处理反映了Go编程语言中使用的语法和样式。通过使用:=、=和类型化，它遵循相同的语义。

5.1.9 循环

使用循环是简化用户与chart交互的常用方法。例如，可以使用循环传值得到公开Web应用程序时要使用的主机列表，然后循环遍历该列表以创建更复杂的Kubernetes Ingress资源。

模板中的循环语法与许多编程语言中的循环语法略有不同。与for循环不同的是，range循环可用于迭代dict（也称为map）和list。

下面的示例演示了dict和list：

```
# An example list in YAML
characters:
  - Sylvester
  - Tweety
  - Road Runner
  - Wile E. Coyote

# An example map in YAML
products:
  anvil: They ring like a bell
  grease: 50% slippery
  boomerang: Guaranteed to return
```

你可以将列表看作一个数组，而带有键名和值的映射类似于Python中的字典或Java中的HashMap。在Helm模板中，你可以使用dict和list函数创建自己的词典和列表。

使用range函数的方法有两种。下面的示例在更改作用域（即“.”的值）时对characters进行迭代：

```
characters:
{{- range .Values.characters }}
  - {{ . | quote }}
{{- end }}
```

在本例中，range遍历列表中的每一项，当Helm对列表中的每一项进行迭代时，它将“.”的值设置为每项的值。在本例中，该值被传递给管道中的quote。在直到end的块中，“.”的作用域都被更改，end充当循环的结束（右）括号或语句。

此代码段的输出为：

```
characters:
  - "Sylvester"
  - "Tweety"
  - "Road Runner"
  - "Wile E. Coyote"
```

使用range的另一种方法是让它为键和值创建新的变量。这将适用于list和dict。下一个示例创建可以在块中使用的变量：

```
products:
{{- range $key, $value := .Values.products }}
  - {{ $key }}: {{ $value | quote }}
{{- end }}
```

$key变量包含map或dict中的键和list中的数字。$value包含值。如果这个值是一个复杂类型（例如另一个dict），那么它将作为$value提供。新变量的作用域一直到range块的末尾，range块由相应的end动作表示。本例的输出为：

```
products:
  - anvil: "They ring like a bell"
  - boomerang: "Guaranteed to return"
  - grease: "50% slippery"
```

dict和list的底层实现

在Go语言中，list表示为由数组支持的分片。值是一个接口，因此它可以是多种类型的。例如，如果使用list函数在模板中创建列表，则返回的值将被类型化为[]interface{}。当对值执行动作时，用反射来确定类型以及如何对该类型执行动作。

map或dict的表示方式稍有不同。它们通常表示为map[string]interface{}。这是dict函数返回的类型，可以在模板中使用。与list一样，它们在对值执行动作时，使用反射来计算值类型。

5.2 命名模板

有时，你需要创建一个模板，以便从Kubernetes清单的模板中调用它。例如，当你有一个由某些复杂逻辑生成的值时，或者当你有一个在多个Kubernetes清单中重复的部分时，可以创建自己的模板（Helm不会自动渲染这些模板），并在Kubernetes清单的模板中使用它们。

运行helm create生成chart时可以找到一个例子。默认情况下，Helm使用一些共享元素（如标签）创建几个Kubernetes清单。为了保持标签的一致性，使它们只需要在一个地方更新，Helm会生成一个模板，然后在每次需要这些标签时调用该模板。

模板中使用了两种类型的标签。有用于更高级别资源（如Deployment）的标签，还有与用于更新的选择器配对的规范中使用的标签。这些标签需要区别对待，因为规范中和选择器上使用的标签通常是不可变的。这意味着你不希望它们包含应用程序版本等元素，因为这些元素会随着应用程序的升级而更改，但是规范和选择器不能用新版本来更新。

以下模板选择包含用于在生成的模板中生成规范和选择器部分的选择器标签。名称anvil来自第4章中生成的chart：

```
{{/*
Selector labels ❶
*/}}
{{- define "anvil.selectorLabels" -}} ❷
app.kubernetes.io/name: {{ include "anvil.name" . }} ❸
app.kubernetes.io/instance: {{ .Release.Name }}
{{- end -}} ❹
```

❶ 定义函数之前的注释。动作中的注释以/*开始，以*/结束。

❷ 你可以使用define语句和模板名称来定义模板。

❸ 模板的内容与其他模板的内容一样。

❹ 模板的定义通过与define语句匹配的end语句结束。

此模板包括一些有用的内容，你应该考虑在自己的模板中使用这些内容：

1. 描述模板的注释。当渲染模板时，它会被忽略，但它与代码注释具有相同的作用。

2. 名称是包含命名空间的，使用“.”作为分隔符，以包含chart名称。在第6章中，你将学习库chart和依赖chart。在模板名称上使用命名空间可以使用库chart，并避免在依赖chart上发生冲突。

3. define和end调用删除前后空格的use动作，以便它们的使用不会向最终输出YAML添加额外的行。

这个模板在资源的spec部分调用，比如anvil chart中的Deployment：

```
spec:
  replicas: {{ .Values.replicaCount }}
  selector:
    matchLabels:
      {{- include "anvil.selectorLabels" . | nindent 6 }}
  template:
    metadata:
      labels:
        {{- include "anvil.selectorLabels" . | nindent 8 }}
```

这里的matchLabels部分是不可变的，因此不能更改，它会在template部分中查找labels。

有两个函数可用于在模板中包含另一个模板：不能用于管道的template函数和可以在管道中使用的include函数。在前面的示例中，include用于调用另一个模板，并将该模板的输出传递给nindent，以确保输出具有适当的缩进级别。由于每个调用的输出都有不同的缩进级别，因此不能将缩进级别作为定义它的模板的一部分。

include函数有两个参数。第一个是要调用的模板的名称，它需要是包含任何命名空间的全名。第二个是要传递的数据对象。它可以是你自己使用dict函数创建的对象，也可以是模板中使用的全局对象的全部或部分。在本例中，将传入整个全局对象。

Helm模板函数用于生成更广泛的标签选择，用在标签可变的更高级别资源的标签上，它既添加标签，也包括选择器标签。它具有调用其他用户定义模板的用户定义模板：

```
{{/*
Common labels
*/}}

{{- define "anvil.labels" -}}
helm.sh/chart: {{ include "anvil.chart" . }}
{{ include "anvil.selectorLabels" . }}
{{- if .Chart.AppVersion }}
app.kubernetes.io/version: {{ .Chart.AppVersion | quote }}
{{- end }}
app.kubernetes.io/managed-by: {{ .Release.Service }}
{{- end -}}
```

因为这些标签是可变的，所以这里包含了一些有用的标签，它们会因各种原因而改变。为了避免重复选择器使用的标签（这里它也很有用），通过调用生成这些标签的函数来包含这些标签。

Kubernetes推荐的标签

Kubernetes文档推荐了一组可以应用于工作负载清单的通用标签。helm create生成的chart包含生成这些标签的模板。

标签以前缀app.kubernetes.io开头，后跟/作为分隔符。Kubernetes的标签文档提到，任何自动生成的标签都应该使用前缀，没有前缀的标签是用户专用的。

这些标签包括应用程序的名称、应用程序的实例（你可以在集群中甚至在单个命名空间中多次运行应用程序）、应用程序的版本、用于显示它在更大的应用程序中的位置的组件类型、应用程序的组成部分，以及用于管理应用程序生命周期的工具的名称（例如，Helm）。在将应用程序链接在一起、在用户界面中显示元数据以及在Kubernetes API中查询信息时，这些标签非常有用。

你可以在Kubernetes文档（https://oreil.ly/uAFIm）中了解有关标签的更多信息和示例。

另一种情况是，当你想要封装复杂的逻辑时，命名模板可能会很有用。为了说明这一想法，请考虑一个chart，在该chart中，你希望能够将容器版本作为标记或摘要传入，或者将应用程序版本作为默认值返回。Pod规范中接受容器镜像（包括版本）的部分是一行代码。要提供全部这三个选项，你需要许多逻辑行：

```
{{- define "anvil.getImage" -}}
{{- if .Values.image.digest -}}
{{ .Values.image.repository }}@{{ .Values.image.digest }}
{{- else -}}
{{ .Values.image.repository }}:
{{- .Values.image.tag | default .Chart.AppVersion }}
{{- end -}}
{{- end -}}
```

这个新的getImage模板能够处理摘要、标记和应用程序版本的默认值（如果其他两个都不存在）。首先，检查并使用摘要。摘要是不可变的，它是指定要使用的镜像修订版本的最精确的方法。如果没有传入摘要，则选中标记。标记是指向摘要的指针，可以更改。如果找不到标记，则将AppVersion用作标记。

这个函数的目标是在第4章创建的anvil chart的结构。镜像的细节应该在chart的结构及其values.yaml文件中。

在Deployment的模板中，将使用如下新的函数来引用镜像：

```
image: "{{ include "anvil.getImage" . }}"
```

模板的作用类似于软件程序中的函数。模板可以分解复杂的逻辑并共享功能。

5.3 为可维护性构建模板

在templates目录中的模板上执行的结构有限。多个Kubernetes清单可以在同一个YAML文件中，这意味着多个Kubernetes清单的模板也可以在同一个文件中。命名模板可以存在于任何模板文件中，也可以在其他模板文件中引用。NOTES.txt文件模板是向用户显示的特殊文件，测试（详见第6章）以特殊的方式处理。除此之外，它还是一个空白画布，供你创建模板。

为了帮助创建易于浏览的可维护模板，Helm维护人员推荐了几种模式。推荐这些模式有以下几个原因：

- 你可能有很长时间不对chart中的模板进行结构修改，然后再回过头来修改。能够快速重新发现布局将使修改过程更快。

- 其他人将查看chart中的模板。这里的其他人可能是创建chart或使用chart的团队成员。使用者可以（有时也确实会）在安装chart之前先打开它进行检查，或者将检查作为分叉过程的一部分来执行。

- 调试chart（见5.4节）时，使用模板中的某些结构更容易调试。

第一条准则是每个Kubernetes清单应该在它自己的模板文件中，并且该文件应该有一个描述性的名称。例如，如果只有一个部署，则把模板命名为deployment.yaml。如果你有相同类型的多个清单的情况，例如使用主库和副本库部署数据库的情况，则命名为statefulset-primary.yaml和statefulset-replica.yaml。

第二条准则是将（包含在自己的模板中的）命名模板放入名为_helpers.tpl的文件中，因为这些基本上是其他模板的辅助模板，所以名称是描述性的。如前所述，名称开头的“_”会使它上升到目录列表的顶部，因此你可以在模板中轻松找到它。

当你使用helm create命令启动一个新chart时，默认情况下，它启动的模板的内容将已经遵守这些准则。

5.4 调试模板

在开发模板时，对模板进行调试很有用。Helm提供了三个特性，你可以在开发工作流中使用它们来查找问题。这些都是用来补充测试的，将在第6章中介绍。

5.4.1 试运行

用于安装、升级、回滚和卸载Helm chart的命令都有一个标志，用于启动试运行和模拟进程，但没有在该进程上完全执行。这是通过在这些命令上使用--dry-run标志来完成的。例如，如果在anvil chart上的install命令上使用--dry-run标志，则可以使用命令helm install myanvil anvil--dry-run。Helm将渲染模板，检查模板以确保发送给Kubernetes的内容格式是正确的，然后将其发送到输出。输出看起来与正常安装时的输出类似，但会有两个额外的部分：

```
NAME: myanvil
LAST DEPLOYED: Tue Jun  9 06:58:58 2020
NAMESPACE: default
STATUS: pending-install
REVISION: 1
HOOKS:
...
MANIFEST:
...
NOTES:
1. Get the application URL by running these commands:
  export POD_NAME=$(kubectl get pods --namespace default ↵
    -l "app.kubernetes.io/name=anvil,app.kubernetes.io/instance=myanvil" ↵
    -o jsonpath="{.items[0].metadata.name}")
  echo "Visit http://127.0.0.1:8080 to use your application"
  kubectl --namespace default port-forward $POD_NAME 8080:80
```

两个新部分是HOOKS和MANIFEST，它们将包含Helm通常会传递给Kubernetes的YAML。不过，它被发送到输出。为了简洁起见，不包括完全生成的YAML，因为那将有一页。

如果模板中有问题，则响应将完全不同。为了说明这一点，请尝试从anvil chart的deployment.yaml文件中删除第一个“}”，并再次执行一个试运行安装。删除“}”将导致解析模板中的动作（action）时出错。Helm不会输出状态，而是输出一个错误，例如：

```
Error: parse error at (anvil/templates/deployment.yaml:4): unexpected "}" in
operand
```

这里的信息概述了在哪里寻找问题的提示，包括：

· 发生错误的文件。在本例中是anvil/templates/deployment.yaml。

· 发生错误的文件中的行号。这里是第4行。

· 带有问题提示的错误消息。错误消息通常不会显示问题是什么，而是显示解析器出现问题的位置。在本例中，单个“}”是意外的。

Helm将检查模板语法中的错误和输出的语法。为了说明这一点，在deployment.yaml文件的开头删除apiVersion:，并确保添加缺少的“}”，这样此动作就修复了。文件的开头现在看起来如下所示：

```
apps/v1
kind: Deployment
```

执行试运行安装将产生以下输出：

```
Error: YAML parse error on anvil/templates/deployment.yaml: error converting
YAML to JSON: yaml: line 2: mapping values are not allowed in this context
```

你可能想知道为什么在YAML和JSON之间转换时会出错。这是Helm和Kubernetes使用的YAML解析库的产物。错误消息的有用部分是从第2行开始的部分。第一行是不完整的，尽管第二行的格式良好，它也处于错误的上下文中。此文件不是有效的YAML，Helm将告诉你从何处开

始查找问题。如果你使用YAML的同一部分并在一个在线YAML验证器中测试它，将得到相同的错误。

Helm还能验证Kubernetes资源的模式。这是因为Kubernetes为其清单提供了模式定义。为了说明这一点，在deployment.yaml中把apiVersion改成foo：

```
foo: apps/v1
kind: Deployment
```

执行试运行安装将产生以下输出：

```
Error: unable to build kubernetes objects from release manifest: error
validating "": error validating data: apiVersion not set
```

部署不再有效，Helm能够就所缺少的东西提供具体的反馈。在本例中，未设置apiVersion属性。

利用试运行并不是使用此功能的唯一方法。helm template命令提供了类似的体验，但没有完整的调试功能集。template命令会将template命令转换为YAML。此时，如果无法解析生成的YAML，它将提供一个错误。它不会根据Kubernetes模式验证YAML。如果apiVersion被转换为foo，template命令不会发出警告。这是因为在使用template命令时，Helm没有与Kubernetes集群通信以获取要验证的模式。

5.4.2 获取已安装清单

有时，你会将应用程序安装到集群中，然后其他内容会更改清单。这会导致所声明的内容和所运行的内容之间存在差异。例如，当一个服务网格自动添加一个sidecar（边车）容器到你的Helm chart创建的Pod中时。

服务网格

服务网格是用于管理“服务到服务”通信的基础结构层。在Kubernetes中，服务网格使用添加到Pod的sidecar代理容器来处理通信。许多服务网格平台都提供了通过改变清单配置来自动注入sidecar代理的能力。

你可以使用helm get manifest命令获取Helm部署的原始清单。此命令将检索发行版本的清单，就像Helm安装发行版本时一样。它能够检索历史中仍然可用的发布版本的任何修订版的信息，与使用helm history命令所发现的一样。

继续myanvil示例，检索anvil chart实例的清单，运行：

```
$ helm get manifest myanvil
```

输出将包括所有的清单，在每个新清单的开始处都带有---。以下是输出中的前15行：

```
---
# Source: anvil/templates/serviceaccount.yaml
apiVersion: v1
kind: ServiceAccount
metadata:
  name: myanvil-anvil
  labels:
    helm.sh/chart: anvil-0.1.0
    app.kubernetes.io/name: anvil
    app.kubernetes.io/instance: myanvil
    app.kubernetes.io/version: "9.17.49"
    app.kubernetes.io/managed-by: Helm
---
# Source: anvil/templates/service.yaml
apiVersion: v1
kind: Service
...
```

---用作YAML文档之间的分隔符。除此之外，Helm还添加了一个YAML注释和用于生成清单的源模板。

5.4.3 校验chart代码

你遇到的一些问题不会显示为违反API规范，也不是模板中的问题。例如，Kubernetes资源需要有可以用作域名的一部分的名称。这限制了名称中可以使用的字符及其长度。Kubernetes提供的OpenAPI模式没有提供足够的信息来检测发送到Kubernetes时将失败的名称。第4章中介绍过的lint命令能够检测到这样的问题并告诉你问题在哪里。

为了说明这一点，你可以修改anvil chart，将Wile添加到deployment.yaml中部署名称的末尾：

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: {{ include "anvil.fullname" . }}-Wile
```

运行helm lint anvil将产生一个错误，通知你该问题：

```
$ helm lint anvil
==> Linting anvil
[ERROR] templates/deployment.yaml: object name does not conform to Kubernetes
naming requirements: "test-release-anvil-Wile"

Error: 1 chart(s) linted, 1 chart(s) failed
```

在本例中，helm lint将向你指出一个问题，并告诉你问题发生在哪里。

5.5 结论

chart中包含的模板提供了在Kubernetes中创建资源的强大功能。它的编写规则类似于围绕模板的编程语言。模板系统具有逻辑、内置函数、自定义模板和调试等功能。这意味着你可以通过值收集所需的输入，并生成所需的Kubernetes清单。

chart还有更多的内容，包括依赖项、测试、值文件的模式等。第6章将详细介绍chart的高级功能。

第6章

chart的高级功能

除了chart元数据和模板集合之外，关于chart还有更多的内容。chart可以有依赖项，值可以有模式，Helm有不同生命周期的钩子，你可以签署chart，等等。在本章中，除了模板之外你还将学习chart的其他元素。

这些特性为构建软件包时出现的常见问题提供了强大的解决方案。本章首先介绍依赖项。依赖项实际上是每个软件包管理解决方案的关键部分，因为它们允许你在解决方案中利用现有的软件包，并基于其他人的工作进行构建。然后讨论了模式和验证，用于帮助chart用户避免问题。再然后讨论了如何用钩子连接Helm执行的流程以执行自定义操作。本章还介绍了测试，测试在开发中非常重要，因为它们可以确保你的软件按预期运行。Helm提供了一些安全特性，这些特性有助于缓解一些常见的威胁途径。本章最后介绍了如何使用chart扩展Kubernetes API。

在本章中，你可以访问网址https://github.com/masterminds/learning-helm/blob/main/chapter6参考作为示例的chart。它们展示了本章中介绍的不同功能以及Helm存储库。

6.1 chart依赖项

依赖项是软件包管理器及其软件包的常见元素。chart可以依赖于其他chart。这使得可以把服务封装在一个chart中，可以重用chart，也可以同时使用多个chart。

我们以一个安装流行博客软件WordPress的chart为例来说明依赖项。WordPress依赖于一个兼容MySQL的数据库来存储博客内容、用户信息和其他配置。兼容MySQL的数据库可以被其他应用程序使用，也可以作为服务使用。将MySQL与WordPress结合使用的一种处理方法是将其清单放在WordPress chart中。另一种处理方法是使用一个独立的MySQL chart，而WordPress chart依赖于它。将一个MySQL兼容的数据库作为一个独立的chart使它能够被多个应用程序使用，并且可以独立地构建和测试数据库。

依赖项在Chart.yaml文件中指定。以下是chart中（该chart名为rocket）Chart.yaml文件的依赖项部分：

```
dependencies:
  - name: booster ❶
    version: ^1.0.0 ❷
    repository: https://raw.githubusercontent.com/Masterminds/learning-helm/main/
      chapter6/repository/ ❸
```

❶ 存储库中依赖chart的名称。

❷ chart的版本范围字符串。

❸ 要从中检索chart的存储库。

Helm chart使用语义版本作为其版本控制方案。用于依赖项的version字段接受一个版本范围，这些范围有一些速记语法。例如，^1.2.3是“>=1.2.3，<2.0.0”的缩写。Helm支持包括=、!=、<、<=、>、>=、^、~、和-的范围。可以使用空格或逗号将不同的范围组合在一起以支持逻辑“与”组合，用|支持逻辑“或”组合。Helm还支持使用通配符X或*。如果你省略了版本的一部分，例如省略了补丁程序部分，Helm将假定缺少的部分是通配符。

范围是指定所需版本的首选方式。稍后，你将学习如何从指定的范围锁定到特定的依赖项版本。通过指定范围，可以使用Helm命令自动更新到该范围内的最新版本。如果你想将错误修复或安全更新拉入依赖项，这非常有用。

速记范围语法

虽然语义版本是从规范定义的，但是用于指定语义版本范围的范围语法没有规范。不同的工具将对相同的^和~使用不同的算法。Helm遵循JavaScript在npm中和Rust在Cargo中使用的相同语法。

对于大于0的主版本，当你使用^时，它产生的范围大于或等于你设置的数字，小于下一个主版本。当主版本小于1时，Helm通常将次要版本视为其工作的范围，而不去处理主版本。以下是范围和等效含义的示例：

· ^1.2.3等价于>=1.2.3<2.0.0

· ^1.2.x等价于>=1.2.0<2.0.0

· ^2.3等价于>=2.3<3

· ^2.x等价于>=2.0.0<3

· ^0.2.3等价于>=0.2.3<0.3.0

· ^0.2等价于>=0.2.0<0.3.0

· ^0.0.3等价于>=0.0.3<0.0.4

· ^0.0等价于>=0.0.0<0.1.0

· ^0等价于>=0.0.0<1.0.0

~用于指定补丁程序范围。其中，^通常舍入到主版本范围内的最新版本，只要指定了次要版本，~就在次要版本范围内舍入。以下示例说明~的含义：

· ~1.2.3等价于>=1.2.3<1.3.0

· ~1等价于>=1<2

· ~2.3等价于>=2.3<2.4

· ~1.2.x等价于>=1.2.0<1.3.0

· ~1.x等价于>=1<2

repository字段用于指定要从中提取依赖项的chart存储库位置。它可以通过以下两种方式之一指定：

· 指向Helm存储库的URL。

· 使用helm repo add命令设置的存储库的名称。此名称前面必须加@，并用引号括起来（例如，"@myrepo"）。

通常用完整的URL来指定位置。这将确保在使用chart的每个环境中检索的依赖项是相同的。

一旦指定了依赖项及其请求的版本范围，就需要使用Helm将这些依赖项锁定到特定版本并检索依赖项。如果要将chart打包为第4章中介绍的chart归档文件，则需要在打包之前锁定并获取依赖项。

要在指定范围内解析依赖项的最新版本并检索它，可以使用以下命令：

```
$ helm dependency update .
```

运行命令后，你将看到以下输出：

```
Saving 1 charts
Downloading booster from repo https://raw.githubusercontent.com/Masterminds/
  learning-helm/main/chapter6/repository/
Deleting outdated charts
```

运行以上命令导致了一些步骤。

首先，Helm解析了最新版本的booster chart。它使用存储库中的元数据来知道哪些版本的chart可用。从元数据和指定的版本范围中，Helm找到了最佳匹配。

解析的信息将写入Chart.lock文件。在Chart.lock文件中包含的不是版本范围，而是要使用的依赖项的特定版本。这对再现性很重要。Chart.lock文件由Helm管理。下一次运行helm dep up（简写语法）时，来自用户的更改将被覆盖。这类似于其他平台上的依赖项管理器的锁定文件操作。

一旦Helm知道要使用的特定版本，它就会下载相关chart并将其放入charts子目录。重要的是依赖chart必须在charts目录中，因为这是Helm从中获取内容以渲染模板的地方。chart可以以归档文件或目录形式存在于chart目录中。当Helm从存储库下载它们时，会将它们存储在它们的归档形式中。

如果你有Chart.lock但是charts目录中没有内容，可以通过运行命令helm dependency build来重建charts目录。这将使用锁文件检索

已确定版本的依赖项。

一旦具备了依赖项，Helm将在运行helm install或helm upgrade等命令时渲染它们的资源。

指定依赖项时，可能还需要将配置从父chart或主chart传递到依赖chart。如果我们回顾一下WordPress的例子，会发现这可以用来设置要使用的数据库的名称。Helm提供了在父chart的值内执行此操作的方法。

在主chart的values.yaml文件中，可以使用依赖chart的名称创建一个新部分。在此部分中，你可以设置要传入的值。只需设置要更改的chart，因为values.yaml文件中包含的依赖chart将用作默认值。

在rocket chart的values.yaml文件中有一个如下部分：

```
booster:
  image:
    tag: 9.17.49
```

Helm知道该部分是用于booster chart的。在本例中，它将镜像标记设置为特定值。依赖chart中的任何值都可以这样设置。运行helm install等命令时，可以使用标志来设置（例如，--set）依赖项的值以及主chart的值。

如果同一chart上有两个依赖项，则可以选择在Chart.yaml文件上使用别名（alias）属性。此属性适用于要在name、version和其他属性旁使用替代名称的每个依赖项。使用alias，你可以为每个依赖项指定一个唯一的名称，以便在其他地方（如values.yaml文件）引用。

紧耦合依赖项与松耦合依赖项

当你有依赖项时，可以将它们紧耦合或松耦合。使用Chart.yaml文件来指定依赖项会导致chart之间的紧耦合。你可以从升级的工作方式中看到这一点。要升级一个chart，必须升级整个组。紧耦合有很多好处，比如一个Helm命令可以安装整个chart集合。当你希望将chart分发给公司或机构外部的其他人时，紧耦合非常有用。

在松耦合的情况下，你可以独立于其他chart安装每个chart。每个chart都将作为自己的实例运行。在这种设置中，每个实例都充当其他服务可以连接的服务。通过松耦合，你可以独立于其他chart更改和升级每个chart。当你在自己的机构中创建和运行chart时，有时会使用此方法。

6.1.1 用于启用依赖项的条件标志

Helm提供了通过配置启用或禁用依赖项的能力。为了说明这个功能，请考虑这样一种情况：你希望提供一个WordPress博客解决方案，但允许安装WordPress的人员将数据库当作服务使用或当作包含的数据库使用。如果安装chart的人选择将数据库当作服务使用，他们将提供指向该服务的URL，而不需要安装数据库。这可以通过两种不同的方式进行配置来实现。

如果要控制是否通过依赖项来启用或禁用单个功能，可以对依赖项使用condition属性。为了说明这一点，我们将查看带条件的chart的chart.yaml文件中的dependencies部分：

```
dependencies:
  - name: booster
    version: ^1.0.0
    condition: booster.enabled
    repository: https://raw.githubusercontent.com/Masterminds/learning-helm/main/
      chapter6/repository/
```

依赖项有一个带有值的condition键，此值告诉Helm在何处查找这些值，以知道是否应该启用或禁用依赖项。在values.yaml文件中，对应部分为：

```
booster:
  enabled: false
```

在本例中，默认值是禁用依赖项。当有人安装chart时，他们可以通过传入一个值来启用依赖项。

如果要启用或禁用多个涉及依赖项的功能，可以使用tags属性。与条件类似，在描述依赖项时，此属性与name和version并列。它包含依赖项的标记列表。为了说明这一点，我们可以查看另一个名为tag的chart的依赖项：

```
dependencies:
  - name: booster
    tags:
      - faster
    version: ^1.0.0
    repository: https://raw.githubusercontent.com/Masterminds/learning-helm/main/
      chapter6/repository/
  - name: rocket
    tags:
      - faster
    version: ^1.0.0
    repository: https://raw.githubusercontent.com/Masterminds/learning-helm/main/
      chapter6/repository/
```

在这里，你将看到两个带有tags部分的依赖项。tags是相关标签的列表。在chart的values.yaml文件中，你使用了tags属性：

```
tags:
  faster: false
```

tags是一种具有特殊意义的属性。这里的值告诉Helm在默认情况下禁用带有faster标记的依赖项。在安装或升级chart时，chart的用户可以将true值传入chart来启用它们。

6.1.2 将值从子chart导入父chart

有时你可能需要将值从子chart导入或拉取到父chart中。Helm为此提供了两种方法。一种是子chart显式导出由父chart导入的值的情况，另一种是子chart不导出值的情况。

exports属性

exports属性是values.yaml文件中的特殊顶级属性。当子chart声明了export属性时，其内容可以直接导入父chart。

例如，子chart的values.yaml文件中包含如下内容：

```
exports:
  types:
    foghorn: rooster
```

当父chart将子chart声明为依赖项时，它可以从导出的内容导入，如下所示：

```
dependencies:
  - name: example-child
    version: ^1.0.0
    repository: https://charts.example.com/
    import-values:
      - types
```

在父类的计算值中，现在可以在顶层访问这些类型。在YAML中，这相当于：

```
foghorn: rooster
```

子-父格式

当父chart想要从子chart导入值，但子chart尚未导出值时，有一种方法可以告诉Helm将子值拉取到父chart中。

为了说明这一点，请考虑一个子chart，它的values.yaml文件如下所示：

```
types:
  foghorn: rooster
```

这些值没有被导出，但父chart仍然可以导入它们。当在父类中声明依赖项时，它可以使用child和parent文件导入值，如以下示例所示：

```
dependencies:
  - name: example-child
    version: ^1.0.0
    repository: https://charts.example.com/
    import-values:
      - child: types
        parent: characters
```

在这两种导入方法中，依赖项上使用的都是import-values属性。Helm知道如何区分不同的格式，你可以将两者混合使用。

在子chart中，types的顶级属性在父chart的计算值中的characters的顶级属性下将不可用。这在YAML中将表示为：

```
characters:
  foghorn: rooster
```

使用句点作为分隔符，此格式除了访问顶级属性外，还允许访问嵌套值。例如，如果子chart具有以下格式，则import-values的child属性可以读作data.types：

```
data:
  types:
    foghorn: rooster
```

6.2 库chart

你可能会遇到这样的情况：你正在创建多个共享许多相同模板的类似chart。对于这些情况，可以使用库chart。

库chart在概念上类似于软件库。它们提供了可重用的功能，这些功能可以被导入并由其他chart使用，但它们本身不能被安装。

使用helm create创建新的库chart时，第一步是删除templates目录和values.yaml文件，因为二者都不会被使用。然后，你需要告诉Helm这是一个库chart。在Chart.yaml文件中将type设置为library。为了说明这一点，下面介绍一个名为mylib的chart中的Chart.yaml：

```
apiVersion: v2
name: mylib
type: library
description: an example library chart
version: 0.1.0
```

未设置type的值时，它的默认值为application。只有在chart是库时，才需要设置它。

templates目录中以下划线（即_）开头的文件不应渲染要发送给Kubernetes的清单。按照惯例，辅助模板和代码段位于_*.tpl和_*.yaml文件中。

为了说明可重用模板的工作原理，以下模板的作用是在mylib chart名为_configmap.yaml的文件中创建一个ConfigMap：

```
{{- define "mylib.configmap.tpl" -}}
apiVersion: v1
kind: ConfigMap

metadata:
  name: {{ include "mylib.fullname" . }} ❶
  labels:
    {{- include "mylib.labels" . | nindent 4 }} ❷
data: {}
{{- end -}}
{{- define "mylib.configmap" -}} ❸
{{- template "mylib.util.merge" (append . "mylib.configmap.tpl") -}}
{{- end -}}
```

❶ fullname函数与helm create生成的函数相同。

❷ labels函数生成Helm建议在chart中使用的常用标签。

❸ 定义了一个特殊的模板，它知道如何将模板合并在一起。

这个定义的大部分内容看起来与你将放入templates目录的其他模板类似。define是用于定义在其他地方使用的模板的函数。此文件中定义了两个模板。mylib.configmap.tpl文件包含资源的模板。这看起

来与其他模板类似。它提供了一个蓝图，该蓝图将由包含此库的chart中的调用者重写。mylib.configmap文件是另一个chart将使用的特殊模板。它需要mylib.configmap.tpl文件以及另一个包含覆盖值的模板（尚未定义），并将它们合并到一个输出中。mylib.configmap文件使用一个实用函数来处理合并，并且易于重用。这个函数是：

```
{{- /*
mylib.util.merge will merge two YAML templates and output the result.
This takes an array of three values:
- the top context
- the template name of the overrides (destination)
- the template name of the base (source)
*/ -}}
{{- define "mylib.util.merge" -}}
{{- $top := first . -}}
{{- $overrides := fromYaml (include (index . 1) $top) | default (dict ) -}}
{{- $tpl := fromYaml (include (index . 2) $top) | default (dict ) -}}
{{- toYaml (merge $overrides $tpl) -}}
{{- end -}}
```

这个函数需要一个上下文（考虑第5章包含的.data）、一个包含覆盖值的模板，以及要被覆盖的基础模板作为参数。当你看到如何使用该函数时，这将变得更加清晰。

库chart的概念是在正式纳入Helm之前发展起来的。merge函数是由Adnan Abdulhussein创建的，这是他通过一个名为Common的chart来开发这个概念的工作的一部分。

为了说明如何使用这个库函数，下面的模板来自另一个名为mychart的chart。在使用它定义的资源之前，需要将其作为依赖项添加，就像其他任何依赖项一样。mychart中包含一个模板以创建ConfigMap：

```
{{- include "mylib.configmap" (list . "mychart.configmap") -}} ❶
{{- define "mychart.configmap" -}} ❷
data: ❸
  myvalue: "Hello Bosko"
{{- end -}}
```

❶ 为ConfigMap包含并使用库chart中的函数。

❷ 定义一个新的模板，只包含用来覆盖库提供的模板的部分。

❸ data部分用于在ConfigMap中使用。

这个模板一开始可能看起来很混乱，因为有很多事情要做。

第一行包含库chart中的ConfigMap模板。将一个新的列表与两个项目一起传递给它。第一个项目是当前的数据对象，第二个项目是另一个模板的名称，该模板包含要覆盖库chart提供的元素。

文件的其余部分是包含覆盖值的模板。在库chart提供的模板中，没有为data部分提供任何内容。函数mychart.configmap负责提供data部分。

Helm从此模板中渲染的输出为：

```
apiVersion: v1
kind: ConfigMap
metadata:
  labels:
    app.kubernetes.io/instance: example
    app.kubernetes.io/managed-by: Helm
    app.kubernetes.io/name: mychart
    helm.sh/chart: mychart-0.1.0
  name: example-mychart
data:
  myvalue: Hello Bosko
```

此输出是来自库和使用此库的chart的合并输出。同样的概念也可以扩展到其他资源，包括那些更长、更复杂的资源。

6.3 模式化值文件

由values.yaml文件定义的值是无模式的。values.yaml文件没有固定的结构需要遵循。不同的chart有不同的结构。这使你能够为正在用chart部署的应用程序或工作负载构造值。

模式提供了许多有用的好处，包括验证内容的能力，以及你可以从中生成用户界面。

Helm为每个chart都提供了可选的功能，可以使用JSON Schema（https://json-schema.org）为其值提供自己的模式。JSON Schema提供了描述JSON文件的词汇表。YAML是JSON的超集，你可以在两种文件格式之间转换内容。这使得使用JSON Schema验证YAML文件的内容成为可能。

运行helm install、helm upgrade、helm lint和helm template命令时，Helm将根据values.schema.json文件中的内容来对值进行验证。Helm验证的值是计算值，包括chart提供的值以及安装chart的人员传入的值。values.schema.json文件位于chart的根目录中，与values.yaml文件并列。该文件可以描述全部或部分值。

考虑以下来自values.yaml文件的部分：

```
image:
  repository: ghcr.io/masterminds/learning-helm/anvil-app
  pullPolicy: IfNotPresent
  tag: ""
```

用来检查该部分的JSON Schema是：

```
{
    "$schema": "http://json-schema.org/schema#",
    "type": "object",
    "properties": {
        "image": {
```

```
        "type": "object", ❶
        "properties": {
            "pullPolicy": {
                "type": "string", ❷
                "enum": ["Always", "IfNotPresent"] ❸
            },
            "repository": {
                "type": "string"
            },
            "tag": {
                "type": "string"
            }
        }
    }
}
```

❶ image是一个对象。如果image不是作为对象传递给Helm，则会抛出一个错误。

❷ pullPolicy是一个字符串。当传入其他类型（如整数）时，将抛出错误。这可以捕捉到细微的问题。

❸ pullPolicy必须是列出的值之一。当另一个值（甚至是一个拼写错误的值）被传入Helm时，就会抛出一个错误。

为了说明这一点，我们可以使用booster chart来举例。如果从chart的根目录运行该命令，你将看到一个错误：

```
$ helm lint . --set image.pullPolicy=foo
```

以下错误告诉你值与模式不匹配的位置：

```
==> Linting .
[ERROR] templates/: values don't meet the specifications of the schema(s) in the
following chart(s):
booster:
- image.pullPolicy: image.pullPolicy must be one of the following: "Always",
  "IfNotPresent"


Error: 1 chart(s) linted, 1 chart(s) failed
```

JSON Schema提供了几种描述属性的方法。最灵活的方法（catch all）是对字符串使用正则表达式。例如，可以使用^(Always|IfNotPresent)$模式来代替枚举。这样的模式就没有描述性了。错误消息可能会指出该值不符合模式。当没有其他方法来描述属性的值时，非常适合使用模式。

模式是对chart的一个有用的补充，它可以捕捉和修复安装chart时可能遇到的细微问题。

6.4 钩子

Helm提供了一种在发布过程中钩住事件并采取行动的方法。如果你希望将一些动作捆绑为发布的一部分，这将非常有用。例如，在升级过程中对数据库进行备份的能力，同时确保在升级Kubernetes资源之前进行备份。

钩子类似于常规模板，它们封装的功能是通过运行在Kubernetes集群中的容器以及应用程序的其他资源提供的。钩子与其他资源的区别在于设置了特殊注解。当Helm看到helm.sh/hook注解时，它将资源用作钩子。表6-1列出了各种钩子及其执行时间。

表6-1：Helm钩子

注解值	说明
pre-install	在渲染资源之后，但在资源被上传到 Kubernetes 之前执行
post-install	在资源上传到 Kubernetes 之后执行
pre-delete	在收到删除请求之后，但从 Kubernetes 中删除任何资源之前执行
post-delete	从 Kubernetes 中删除所有资源之后执行
pre-upgrade	在渲染资源之后，但在 Kubernetes 更新资源之前执行
post-upgrade	在 Kubernetes 中升级资源之后执行
pre-rollback	在渲染资源之后，但在回滚 Kubernetes 中的任何资源之前执行
post-rollback	在 Kubernetes 中回滚资源之后执行
test	运行 helm test 命令时执行

一个资源可以通过以逗号分隔的列表列出多个钩子值来实现多个钩子。例如：

```
annotations:
  "helm.sh/hook": pre-install,pre-upgrade
```

钩子可以加权，并在资源运行后为其指定删除策略。权重允许为同一事件指定多个钩子，同时指定钩子的运行顺序。这使你能够确保确定的顺序。因为Kubernetes资源用于执行钩子，所以即使在执行完

成之后，这些资源也存储在Kubernetes中。删除策略为你提供了一些关于何时从Kubernetes中删除这些资源的附加控制。

下面的代码提供了示例注解，用于指定所有三个值：

```
annotations:
  "helm.sh/hook": pre-install,pre-upgrade
  "helm.sh/hook-weight": "1"
  "helm.sh/hook-delete-policy": before-hook-creation,hook-succeeded
```

权重由helm.sh/hook-weight注解键指定，是字符串形式的数字。它应该永远是一个字符串。权重可以是正数或负数，默认值为0。在执行钩子之前，Helm会按升序排列它们。

删除策略使用helm.sh/hook-delete-policy注解键设置，是以逗号分隔的策略选项列表。表6-2列出了三种删除策略。

表6-2：Helm钩子删除策略

策略值	说明
before-hook-creation	在启动此钩子的新实例之前，将删除以前的资源。这是默认值
hook-succeeded	在钩子成功运行后删除 Kubernetes 资源
hook-failed	如果钩子在执行时失败，删除 Kubernetes 资源

默认情况下，Helm保留用于钩子的Kubernetes资源，直到钩子再次运行。这提供了在钩子运行后检查日志或查看其他钩子信息的能力。上一个示例中使用的策略是设置的通用策略。除非钩子资源成功

完成，否则策略将保留钩子资源。当钩子失败时，资源及其日志仍可用于检查，否则它们将被删除。

下面的Pod是一个安装后运行的钩子示例：

```
apiVersion: v1
kind: Pod
metadata:
  name: "{{ include "mychart.fullname" . }}-post-install"
  labels:
    {{- include "mychart.labels" . | nindent 4 }}
  annotations:
    "helm.sh/hook": post-install

    "helm.sh/hook-weight": "-1"
    "helm.sh/hook-delete-policy": before-hook-creation,hook-succeeded
spec:
  containers:
    - name: wget
      image: busybox
      command: ["/bin/sleep","{{ default "10" .Values.sleepTime }}"]
  restartPolicy: Never
```

如果正在运行Helm命令（如helm install），并且希望跳过正在运行的钩子，则可以使用--no hooks标志。钩子是一种可以选择跳过

的功能。

6.5 向chart中添加测试

测试是软件开发不可或缺的一部分，Helm通过使用test钩子和Kubernetes资源提供了测试chart的能力。这意味着测试在Kubernetes集群中与工作负载（可以访问chart安装的组件）一起运行。除了Helm中内置的chart测试之外，Helm项目还提供了一个名为Chart Testing的测试工具。由于Chart Testing建立在Helm客户端的特性之上，因此我们将首先查看Helm客户端中内置的功能。

6.5.1 Helm测试

Helm有一个helm test命令，该命令在运行的chart实例上执行test钩子。实现这些钩子的资源可以用来检查数据库能否访问、数据库模式是否正确就位、工作负载之间的连接是否工作以及其他操作细节。

如果测试失败，Helm将使用非零的退出代码退出，并向你提供失败的Kubernetes资源的名称。非零的退出代码在与某些自动测试系统（以这种方式检测故障）配对时非常有用。当你拥有Kubernetes资源的名称时，可以查看日志以查看失败的内容。

测试通常位于templates目录的tests子目录中。将测试放在这个目录中提供了一个有用的分离。这是一个约定，不是运行测试所必需的。

为了演示一个测试，我们将看看booster chart（https://oreil.ly/COJ7w）。在templates/tests目录中，test-connection.yaml文件中有一个测试，它包含以下测试钩子：

```
apiVersion: v1
kind: Pod
metadata:
  name: "{{ include "booster.fullname" . }}-test-connection"
  labels:
    {{- include "booster.labels" . | nindent 4 }}
  annotations:
    "helm.sh/hook": test
spec:
  containers:
    - name: wget
      image: busybox
      command: ['wget']
      args: ['{{ include "booster.fullname" . }}:{{ .Values.service.port }}']
  restartPolicy: Never
```

这个测试是在运行helm create时默认为Nginx创建的。它碰巧也可以对与booster应用程序的连接进行测试。这个简单的测试说明了测试的结构。

如果你查看一些现有chart中的测试，可能会发现它们使用的钩子是test-success而不是test。在Helm 2中，有一个名为test-success的钩子用于运行测试。helm 3提供了向后兼容性，并将此钩子名称作为测试钩子来运行。

运行测试有两个步骤。第一步是安装chart，使其实例正在运行。可以使用helm install命令执行此操作。下面的命令安装booster

chart，并假设你正在从chart的根目录运行它：

```
$ helm install boost .
```

chart实例运行后，就可以运行helm test命令执行测试：

```
$ helm test boost
```

Helm将在测试执行时输出测试的状态，然后在完成时输出有关测试和发布的信息。在上一次测试中，将会返回：

```
Pod boost-booster-test-connection pending
Pod boost-booster-test-connection pending
Pod boost-booster-test-connection running
```

```
Pod boost-booster-test-connection succeeded
NAME: boost
LAST DEPLOYED: Tue Jul 21 06:47:05 2020
NAMESPACE: default
STATUS: deployed
REVISION: 1
TEST SUITE:     boost-booster-test-connection
Last Started:   Tue Jul 21 06:47:12 2020
Last Completed: Tue Jul 21 06:47:17 2020
Phase:          Succeeded
NOTES:
1. Get the application URL by running these commands:
  export POD_NAME=$(kubectl get pods --namespace default -l
    "app.kubernetes.io/name=booster,app.kubernetes.io/instance=boost"
    -o jsonpath="{.items[0].metadata.name}")
  echo "Visit http://127.0.0.1:8080 to use your application"
  kubectl --namespace default port-forward $POD_NAME 8080:80
```

当chart具有包含测试的依赖项时，这些依赖项也将运行测试。例如，如果本章前面使用的rocket chart中的测试运行，则将运行booster chart测试和rocket chart测试。

如果需要在测试中安装配置文件，可以将测试钩子放在Kubernetes Secret或ConfigMap上，以便将其与其他测试资源一起安装。

测试chart是一种很好的方法，可以确保chart的内容能够在Kubernetes中运行工作负载，并捕获可能破坏工作负载的更改。

6.5.2 Chart Testing工具

Helm项目提供了一个附加的测试工具，它建立在helm test的基础上，提供了更高级的测试能力。它包含以下附加功能：

· 能够在安装chart时测试不同的（互斥）配置选项。

· 包含自定义模式规则的Chart.yaml模式验证。

· 额外的YAML源代码校验，包括可配置的规则。例如，你可以确保YAML文件中的缩进是一致的。

· 当源代码存储在Git中时，能够检查chart.yaml文件中的version属性是否已正确递增。

· 能够处理chart集合并仅测试已更改的chart。

Chart Testing工具被设计用于持续集成系统工作流中，其中一些特性直接针对这种情况设计。

Chart Testing的历史

在Helm发展的早期，项目维护人员创建了一个存储库，在其中存储了一些chart并展示了它们的用途。Helm存储库从一开始就被设计为分布式的，不同的组织运行自己的存储库，Helm项目提供了一个如何做到这一点的示例。

这个chart存储库已发展到有许多chart，并成为一种中央存储库。为了帮助维护多个chart，维护人员创建了自动化脚本，以帮助自动对向chart提出的拉取请求提供反馈。

事实证明，自动化脚本不仅仅对Helm项目有用。为了使其他人托管的chart存储库具有相同的测试功能，Helm项目使用的脚本被分解成

一个单独的工具，并在考虑可移植性的情况下重写。

Chart Testing工具现在被许多公司和组织用来帮助测试其托管的chart。

Chart Testing需要了解不同的、互斥的配置，才能使用这些配置来测试chart。它们被捆绑在chart的ci目录中。

在ci目录中，你可以为每个测试创建一个值文件。命名每个文件时，你需要使用glob命名模式：*-values.yaml。例如，可以使用minimal-values.yaml和full-values.yaml等文件名。

Chart Testing将分别测试这些配置。例如，在对chart进行源代码校验时，将分别校验每个案例。自定义值将使用--values标志传递给helm lint。在对chart进行运行时测试时，同样的思想和标志也适用。这些值通过--values标志传递给Helm，因为这就是安装chart的最终用户提供自定义配置的方式。

如果你想使用各种配置进行测试，但不想将这些配置作为chart归档文件的一部分提供，可以将ci目录放在.helmignore文件中。当Helm打包chart时，ci目录将被忽略。

Chart Testing可以用各种方式安装和使用。例如，你可以将其用作开发系统上的二进制应用程序或持续集成系统中的容器中的二进制应用程序。在项目页面（https://oreil.ly/sJXpR）上可了解有关使用和设置它的更多信息。

6.6 安全注意事项

软件更新操作已经使一些最大和最受信任的技术组织的用户受到攻击。软件更改和用于更新甚至安装软件的机制都为此类攻击提供了攻击渠道。

Helm提供了一种可选的方法来检查chart的来源和完整性。出处（prove-nance）提供了一种方法来验证chart的来源（如公司或个人）。而完整性（integrity）提供了一种方法来检查你收到了期望的、未被修改的东西。此功能使你和使用chart的人员能够验证chart的来源以及内容未被修改。

为了实现这一点，Helm使用了Pretty Good Privacy（PGP）、散列和一个与chart归档文件并列存放的provenance文件。例如，如果你有一个名为mylib-1.0.0.tgz的chart归档文件，则可以有一个名为mylib-1.0.0.tgz.prov的provenance文件。此文件包含一条PGP消息，其中包含Chart.yaml文件的内容以及chart归档文件的散列值。Helm可以为你生成这些文件。以下示例是mylib-1.0.0.tgz的provenance文件：

```
-----BEGIN PGP SIGNED MESSAGE-----
Hash: SHA512

apiVersion: v2
description: an example library chart
name: mylib
type: library
version: 0.1.0

...
files:
  mylib-0.1.0.tgz: sha256:d312aea39acf7026f39edebd315a11a52d29a9
                          6a8d68737ead110ac0c0a1163d
```

```
-----BEGIN PGP SIGNATURE-----

iQIzBAEBCgAdFiEEcR8o1RDh4Ly9X2v+lDboC/ukaQkFAl8yiesACgkQlDboC/uk
aQkG2BAAlIEgGI7uu9Kr8j4ZIxDseLmgphhPM1kgnIMPriLieBxFXSJQxciN3+dx
OQpIfdsFQvW98EnJ4781Pm+leHY2iI/L08O1cQWUtzKhfPEWC65YQJPXkTKpHnC2
wXYVUVYWvhx6BJ77RiS/f+hoXiC+i1aBqqS0TAG+AqXuwARO2tY/L7cF6EHjsUwD
pPuTNpYZ/OEWqh1KEYZYVDvLm6uN6QjV4pNTFfAgnvMckfoDLQ+kOPQVqCeUWG3F
tZO3sBzUg+Ak2dDviSTOFQ7TCifc3tOOaWS1XtcooSOkUENmTeeWV56jZnhK1rT4
yaIGT16zXZIdmkZ1t5o9VccuAhQ1Us2FhipdGqpD8yDoJABVz/ee9d2zoX8anfR7
LZ7fwecgQ/THnj54RroyQlzf2aottFiL9ZV4MjUqs0CSoA9+SZ/CcJDd/rxBGI8C
yxRqo0VoNdjT8Kr9hha13krfwD8IpLH8bv4kWt3Ckh6rgphjUL19xyTHJY7w2toY
bAeZMl3Y05Ca76EA7XDdoltE57SUS1Zzd+wDRzRD0IZO8KVk+Z5/PzzvV4l9lnDJ
X63fptInbJpyk0xYKLMFquOY7Yy5mlI9de7424CScePo9Nua3GAakfi4zk3i4Auz
2eaoU/S5uXt605OydkSLLz99BAyJwmazzf/qPyYcPWMw/b+gHxw=
=pRcC
-----END PGP SIGNATURE-----
```

provenance文件是消息中具有特定结构的PGP签名消息。Helm使用消息中的散列值来验证完整性，PGP签名用于验证消息来自谁。

使用provenance文件有两个步骤。首先，你需要生成它们。为此，你需要有一个PGP密钥对。

使用helm package命令创建软件包时，可以告诉Helm对软件包进行签名：

```
$ helm package --sign --key 'bugs@acme.example.com' \
  --keyring path/to/keyring mychart
```

附加的标志将告诉Helm创建provenance文件。--sign标志选择进行签名，--key标志指定要使用的私钥的名称，--keyring标志指定要使用的密钥环的位置，该密钥环包含用于签名的私钥。当Helm创建chart的归档文件时，还会同时创建对应的.prov文件。

provenance文件应与chart归档文件一起上传，并可从chart存储库下载。

验证以相反的方式进行，并内置在helm install、helm upgrade和helm pull等命令中，同时在helm verify命令中可用。

Helm可以处理这样两种情况：你在本机同时拥有归档文件和provenance文件；你在远程存储库中拥有chart。

为了说明两个文件都在本机的情况，我们可以使用helm verify命令：

```
$ helm verify --keyring path/to/keyring mychart-0.1.0.tgz
```

verify命令将告诉Helm检查散列值和签名。--keyring标志告诉Helm PGP密钥环存在于何处，其公钥与签名chart的私钥相匹配。这可以是密钥环，也可以是公钥的非ASCII封装版本。Helm将寻找mychart-0.1.0.tgz.prov文件并使用它执行检查。

在mylib chart上运行verify命令的方法如下：

```
$ helm verify mylib-0.1.0.tgz --keyring public.key
```

输出：

```
Signed by: Matthew Farina
Using Key With Fingerprint: 672C657BE06B4B30969C4A57461449C25E36B98E
Chart Hash Verified: sha256:d312aea39acf7026f39edebd315a11a52d29a96a8d68737ead11
                                0ac0c0a1163d
```

如果你在某个Helm存储库中有一个chart，Helm将在下载此chart时下载provenance文件。例如：

```
$ helm install --verify --keyring public.key myrepo/mychart
```

当Helm获取chart归档文件时，它还将下载provenance文件，验证签名，并验证散列值。

 公钥应该通过与chart和provenance文件不同的通道共享。

如果在验证过程中出现问题，Helm将报告一个错误并使用非零退出代码退出。

GNU隐私保护

从GNU Privacy Guard（GPG）2.1开始，密钥以新的keybox格式存储。此新格式与PGP规范和格式不兼容。这意味着，如果使用GPG，使用密钥还有一些额外的步骤。以下命令提供了使用GPG时可以使用的参考。

你可以用如下命令将密钥从GPG导出为PGP格式：

```
gpg --export-secret-keys > secring.gpg
```

你可以用如下命令将公钥从GPG导出为PGP格式：

```
gpg --export > pubring.gpg
```

你可以用如下命令将ASCII格式的公钥转换为二进制格式：

```
gpg --dearmor < pgp_key.asc > public.key
```

pgp_key.asc是ASCII封装密钥文件的名称，而public.key是相同密钥的二进制格式名称。这个public.key文件可以作为密钥环传递给Helm进行验证。

如果将密码或硬件安全设备与GPG配合使用，则可能无法导出私钥。在这种情况下，就需要利用Helm GPG插件。它提供了通过GPG直接处理provenance文件的命令和方法。第8章将更详细地介绍插件。

验证一个chart是否来自你期望的来源，并且内容没有改变，这是确保软件供应链安全的一个有用步骤。

6.7 自定义资源定义

Kubernetes自定义资源定义（CRD）提供了一种扩展Kubernetes API的方法，Helm提供了将其作为chart的一部分进行安装的方法。

自定义资源定义、Kubernetes API和一些陷阱

CRD提供了一种方法来为集群的所有用户扩展Kubernetes API。它添加了新的资源类型，称为自定义资源，这些资源可以与Kubernetes附带的资源类型一起上传到集群中。CRD提供了一个模式，可以描述同一资源的多个版本。它们是共享的全局资源。

可以更新集群中的CRD以更改API。可以是更改或更新API现有版本的模式，也可以是向API添加新版本。Kubernetes社区建议，只要API有突破性的更改，就应该增加API版本号，但是向后兼容的更改是可以接受的。Kubernetes没有强制执行这一建议。例如，将可选字段添加到API是向后兼容的，但新的强制字段不向后兼容。更改CRD会影响集群的所有用户。

当从集群中删除CRD时，基于它的所有自定义资源也将被删除。这适用于集群的所有用户，因为CRD是集群范围内的资源。在多租户集群中，当一个租户删除一个CRD时，该CRD为所有租户描述的自定义资源都将被删除。

CRD之所以用这样的方式工作，是因为它被设计成集群级扩展。在一次关于CRD的回顾中，Kubernetes的创始人之一Brendan Burns描述了他们的三个目标：

1.向Kubernetes中动态添加新API类型的简单方法。

2.实现API可扩展性，且不给Kubernetes集群的操作员增加额外的负担。

3.为最终用户集群提供增值扩展的生态系统。

在开发Kubernetes API扩展时，开发了另一种扩展API的方法，但这需要集群操作员和开发人员做更多的工作。CRD简化了这种体验。

CRD在概念上类似于操作系统中的内核模块和扩展。

有两种基于Helm的方法来管理chart所使用的CRD。选择的方法通常取决于安装chart的人的需求和环境配置。

首先，crds目录是一个特殊的目录，你可以将它添加到chart中以保存CRD。Helm将在安装其他资源之前安装CRD。这样可以确保CRD可用于任何可能在chart中利用它的自定义资源或控制器。

crds目录中的CRD与Helm安装的其他资源不同。这些文件没有被模板化。这对于我们稍后将介绍的CRD管理工作流非常有用。Helm不会像升级或删除其他资源一样升级或删除CRD。升级CRD会更改集群中所有自定义资源实例的API，删除CRD会删除所有用户的所有自定义资源。当涉及处理这些集群范围的更改时，你需要使用一个配套工具，比如kubectl（Kubernetes的命令行工具）。

因为CRD更改了Kubernetes API，所以安装你的chart的人可能没有安装、升级或删除它的权限。如果要将应用程序捆绑分发给其他公司或公众，则会出现这种情况。一些集群管理员限制对这些功能的访问，作为其安全访问控制的一部分。

crds目录中的CRD可以从chart中提取，并直接与kubectl等工具一起使用。这样，如果安装chart的人没有权限，就可以将CRD传递给有权安装它的人。提取的CRD还可以用于使用其他工具升级集群中的CRD。

第二种管理CRD的方法是使用第二个保存CRD的chart，这种方法基于Helm，同时在通过自定义资源使用CRD之前提供了安装CRD的顺序。这种方法通过Helm提供了更细致的控制。

使用第二个chart：

1.可以为CRD使用Helm模板和普通的templates目录。

2.Helm将管理CRD的生命周期。这包括卸载和升级。如果要在卸载chart后继续安装CRD，可以设置注解"helm.sh/resource-policy":keep告诉Helm跳过卸载资源的步骤。

3.如果应用程序出现问题，并使用卸载和重新安装的方法尝试修复问题，则不会删除单独的chart中的CRD。

第二个chart可以以松耦合的方式安装（指示人们先安装），也可以以紧耦合的方式安装（设置为依赖项）。如果保存CRD的chart被设置为依赖项，那么其用例应该在设置集群范围的资源时只安装一次。

当Helm管理CRD时，需要特别注意升级用例和删除用例的处理方式。例如，如果安装了两个版本的CRD安装chart，则需要确保旧版本不会覆盖新版本，并且新版本不会破坏集群中使用旧版本的其他人的功能。如果两个人安装了不同版本的安装CRD的chart，就会发生这种情况。在多租户集群中，集群的不同用户之间可能彼此不了解，确保集群的一个用户不会破坏集群的另一个用户的工作负载是很重要的。

安装和使用CRD时，Helm开发人员建议在所有生命周期步骤中都要特别小心，以确保chart用户不会遇到意外破坏工作负载的情况。

6.8 结论

Helm chart不仅仅是模板的集合。它还处理依赖项，可以包含模式，提供事件钩子机制，可以包含测试，并具有安全特性。这些特性是使Helm成为健壮可靠的软件包管理问题解决方案的重要因素。

第7章

chart存储库

如果没有共享和分发软件包的方法，软件包管理器的功能就不完整。组织和供应商必须有一种发布软件包的方法，供最终用户下载和使用。同样，最终用户必须有一种可以从各种来源获取软件包的通用方法。

Helm支持通过一个名为chart存储库的系统来分发软件包。chart存储库是简单的HTTP（S）Web服务，用户可以从中查找和下载可用的chart。从概念上讲，chart存储库在设计上类似于Debian软件包存储库、Fedora软件包数据库或综合Perl归档网络（Comprehensive Perl Archive Network,CPAN）。

在本章中，我们将首先深入研究chart存储库的内部结构。我们将讨论存储库索引以及如何用新的chart版本更新它。之后，我们将演示如何从头开始设置chart存储库，如何保护一个chart存储库，还将演示如何使用GitHub页面为开源项目托管公共chart存储库的真实示例。之后，我们将介绍各种helm repo命令以及如何有效地使用它们。

最后，我们将介绍使用Helm的实验性开放容器计划（Open Container Initiative,OCI）支持的下一代chart存储库。Helm 3中添加的这一前沿功能允许用户将Helm chart与容器镜像一起存储在容器登记站中。

最后，我们将简要描述Helm生态系统中与chart存储库相关的一些项目。

7.1 存储库索引

所有chart存储库都包含一个名为index.yaml的索引文件，其中列出了所有可用的chart及其各自的下载位置。

 index.yaml格式的更多详细信息参见附录B。

下面是一个非常基本的index.yaml文件：

```
apiVersion: v1
entries:
  superapp:
  - apiVersion: v2
    appVersion: 1.16.0
    created: "2020-04-27T17:46:52.60919-05:00"
    description: A Helm chart for Kubernetes
    digest: cd1f8d949aeb6a7a3c6720bfe71688d4add794881b78ad9715017581f7867db4
    name: superapp
    type: application
    urls:
    - superapp-0.1.0.tgz
    version: 0.1.0
generated: "2020-04-27T17:46:52.607943-05:00"
```

注意entries部分，它列出了所有chart和chart版本。这个index.yaml示例仅列出了一个chart，即superapp，其版本为0.1.0。

7.1.1 chart存储库索引示例

通常，chart存储库列出许多chart及其所有可用版本。这允许用户下载他们希望安装的特定版本的chart。下面是一个更详细的chart存储库索引示例，其中包含多个chart和chart版本：

```
apiVersion: v1
entries:
  cert-manager:
  - apiVersion: v1
    appVersion: v0.14.2
    created: "2020-04-08T11:38:26.281Z"
    description: A Helm chart for cert-manager
```

```
    digest: 160e1bd4906855b91c8ba42afe10af2d0443b184916e4534175890b1a7278f4e
    home: https://github.com/jetstack/cert-manager
    icon: https://raw.githubusercontent.com/jetstack/cert-manager/master/logo/
          logo.png
    keywords:
    - cert-manager
    - kube-lego
    - letsencrypt
    - tls
    maintainers:
    - email: dev@jetstack.io
      name: jetstack-dev
    name: cert-manager
    sources:
    - https://github.com/jetstack/cert-manager
    urls:
    - charts/cert-manager-v0.14.2.tgz
    version: v0.14.2
  - apiVersion: v1
    appVersion: v0.14.1
    created: "2020-03-25T18:30:16.354Z"
    description: A Helm chart for cert-manager
    digest: 629150400487df41af6c7acf2a3bfd8e691f657a930bc81e1dcf3b9d23329baf
    home: https://github.com/jetstack/cert-manager
    icon: https://raw.githubusercontent.com/jetstack/cert-manager/master/logo/
          logo.png
    keywords:
    - cert-manager
    - kube-lego
    - letsencrypt
    - tls
    maintainers:
    - email: dev@jetstack.io
      name: jetstack-dev
    name: cert-manager
    sources:
    - https://github.com/jetstack/cert-manager
    urls:
    - charts/cert-manager-v0.14.1.tgz
    version: v0.14.1
  tor-proxy:
  - apiVersion: v1
    created: "2018-11-16T09:23:13.538Z"
    description: A Helm chart for Kubernetes
    digest: 1d2fd11e22ba58bf0a263c39777f0f18855368b099aed7b03123ca91e55343e4
    name: tor-proxy
    urls:
    - charts/tor-proxy-0.1.1.tgz
    version: 0.1.1
generated: "2020-04-23T17:43:41Z"
```

前面的示例显示了两个可用的chart：cert-manager和tor-proxy。共有三个可用的chart版本：cert-manager v0.14.1、cert-manager v0.14.2（最新版本）和tor-proxy 0.1.1（最新版本）。在运行helm search时，会显示repo中每个chart的最新版本。

通常，chart归档文件（.tgz文件）本身的存放位置与存储库索引相同，但索引也可能链接到完全不同的域上的远程位置。下面的index.yaml片断引用了位于分离的域上的chart归档文件（请注意绝对URL）：

```
- appVersion: 2.10.1
  created: 2019-01-14T23:25:37.125126859Z
  description: A simple, powerful publishing platform that allows you to share
    your stories with the world
  digest: dcadf39f81253a9a016fcab1b74aba1d470e015197152affdaeb1b337221cc5c
  engine: gotpl
  home: http://www.ghost.org/
  icon: https://bitnami.com/assets/stacks/ghost/img/ghost-stack-220x234.png
  keywords:
  - ghost
  - blog
  - http
  - web
  - application
  - nodejs
  - javascript
  maintainers:
  - email: containers@bitnami.com
    name: Bitnami
  name: ghost
  sources:
  - https://github.com/bitnami/bitnami-docker-ghost
  urls:
  - https://charts.example.com/ghost-6.2.3.tgz ❶
  version: 6.2.3
...
```

❶ 绝对chart URL。

每个条目中包含的其他字段包括chart的元数据（它们在Chart.yaml中描述，比如description），以及添加的digest（摘要）字段，其中包含chart归档文件的安全散列算法（SHA-256）校验和。在第4章中，我们详细讨论了chart元数据和Chart.yaml。

此外，在最上层是一个generated字段（描述索引创建的时间（使用RFC 3339格式）），以及一个apiVersion（描述索引的API版本）。在撰写本书时，chart存储库只有一个API版本。此字段应始终为v1。

7.1.2 生成索引

存储库索引可以由自定义程序生成，也可以手动键入。Helm还提供了生成存储库索引的内置功能。

让我们创建一个空目录charts/，它将作为chart存储库的根目录：

```
$ mkdir -p charts/
```

要在charts/目录中生成存储库索引，请运行以下命令：

```
$ helm repo index charts/
```

这将在chart/index.yaml目录下创建一个文件：

```
$ cat charts/index.yaml
apiVersion: v1
entries: {}
generated: "2020-04-28T09:55:29.517285-05:00"
```

你会注意到entries是空的。这是意料之中的，因为charts/目录中还没有任何chart。

让我们创建一个示例chart，并将其打包到charts/目录中：

```
$ helm create superapp
Creating superapp
$ helm package superapp/ --destination charts/
Successfully packaged chart and saved it to: charts/superapp-0.1.0.tgz
```

现在，让我们再次尝试生成索引：

```
$ helm repo index charts/
$ cat charts/index.yaml
apiVersion: v1
  entries:
    superapp:
    - apiVersion: v2
      appVersion: 1.16.0

      created: "2020-04-28T10:12:22.507943-05:00"
      description: A Helm chart for Kubernetes
      digest: 46f9ddeca12ec0bc257a702dac7d069af018aed2a87314d86b230454ac033672
      name: superapp
      type: application
      urls:
      - superapp-0.1.0.tgz
      version: 0.1.0
generated: "2020-04-28T10:12:22.507289-05:00"
```

现在我们看到entries部分中列出了这个chart。

7.1.3 添加到现有索引

在某些场景中（例如，持续集成/持续部署（CI/CD）），你可能只能访问现有的index.yaml文件和新打包的chart归档文件。Helm使用--merge选项提供了一种基于现有索引内容的构建机制。

让我们来模拟这个场景。创建一个名为workspace/的新目录，它将表示CI/CD流水线中的一个新工作目录：

```
$ mkdir -p workspace/
```

将现有的索引文件用新名称（如index-old.yaml）复制到workspace/目录中：

```
$ cp charts/index.yaml workspace/index-old.yaml
```

在真实场景中，你可能从某个远程位置（例如，Amazon S3）获取现有的索引文件。

接下来创建另一个Helm chart并将其打包到workspace/目录中：

```
$ helm create duperapp
Creating duperapp
$ helm package duperapp/ --destination workspace/
Successfully packaged chart and saved it to: workspace/duperapp-0.1.0.tgz
```

运行以下命令，它将基于在index-old.yaml索引中找到的现有条目以及workspace/目录中的任何.tgz文件的组合来创建一个新的index.yaml文件：

```
$ helm repo index workspace/ --merge workspace/index-old.yaml
```

最后，将文件从workspace/目录移到charts/目录中，用新的索引文件覆盖旧的索引文件

```
$ mv workspace/duperapp-0.1.0.tgz charts/
$ mv workspace/index.yaml charts/
```

索引文件的新版本现在应包含两个chart的条目：

```
$ cat charts/index.yaml
apiVersion: v1
entries:
  duperapp:
  - apiVersion: v2
    appVersion: 1.16.0
    created: "2020-04-28T11:34:26.780267-05:00"
    description: A Helm chart for Kubernetes
    digest: 30ea14a4ce92e0d1aea7626cb30dfbac68a87dca360d0d76a55460b004d62f52
    name: duperapp
    type: application
    urls:
    - duperapp-0.1.0.tgz
    version: 0.1.0
  superapp:
  - apiVersion: v2
    appVersion: 1.16.0
    created: "2020-04-28T10:12:22.507943-05:00"
    description: A Helm chart for Kubernetes
    digest: 46f9ddeca12ec0bc257a702dac7d069af018aed2a87314d86b230454ac033672
    name: superapp
    type: application
    urls:
    - superapp-0.1.0.tgz
    version: 0.1.0
generated: "2020-04-28T11:34:26.779758-05:00"
```

在你不一定能访问包含所有chart归档文件的目录的环境中，此方法非常有用。

但是请记住，如果这种合并同时发生在多个系统上，你可能会遇到冲突的情况，导致索引中会缺少一个或多个chart。通过确保此过程仅同步执行（例如，一个单独的CI作业负责创建用于存储库的index.yaml）可以缓解此问题。解决此问题的另一种方法是使用一个动态Web服务器负责生成index.yaml的内容，ChartMuseum项目（见7.5节）就是可以用于此用途的动态chart存储库服务器的一个示例。

7.2 设置chart存储库

chart存储库的好处之一是它们可以是完全静态的，这意味着你可以将文件放在简单的Web服务器（如Apache或Nginx）后面，并按原样提供服务。你甚至可以使用对象存储提供程序，如Amazon S3。当客户端请求index.yaml时，服务器端不需要进行任何重要的计算。例如，静态Web服务器只打开文件系统中存在的文件，并将原始内容发送回客户端。

7.2.1 基于Python的简单chart存储库

在本例中，我们将使用Python内置的静态Web服务器来建立本地测试存储库。注意，几乎所有编程语言的标准库都支持启动Web服务器并提供静态文件。之所以选择Python，仅仅是因为它预先安装在大多数基于Unix的系统上，而且提供了一个简单的命令行来启动静态Web服务器。

按照7.1.2节中的说明，创建chart/目录，其中包含index.yaml，superapp-0.1.0.tgz和duperapp-0.1.0.tgz文件。运行以下命令之一，在http://localhost:8080上启动本地Web服务器。

使用Python 3（首先尝试）：

```
$ ( cd charts/ && python3 -m http.server --bind 127.0.0.1 8080 )
```

使用Python 2：

```
$ ( ch charts/ && python -m SimpleHTTPServer 8080 )
```

 Python2的这行命令监听所有接口（0.0.0.0），而不只监听环回接口（127.0.0.1）。这将允许网络上的其他设备连接到你的机器，具体取决于你的系统。在运行此命令之前，请注意charts/目录中存在哪些文件。

现在，在另一个终端窗口中，尝试使用curl获取index.yaml：

```
$ curl -sO http://localhost:8080/index.yaml
$ ls *.yaml
index.yaml
```

验证一下是否可以获取chart归档文件：

```
$ curl -sO http://localhost:8080/superapp-0.1.0.tgz
$ curl -sO http://localhost:8080/duperapp-0.1.0.tgz
$ ls *.tgz
duperapp-0.1.0.tgz      superapp-0.1.0.tgz
```

如果curl命令执行成功，你的chart库就可以用于Helm了。

7.2.2 保护chart存储库

在许多情况下，你可能希望限制对chart存储库的访问，或者维护对哪些用户正在访问哪些资源的审计跟踪日志。Helm对此提供了内置的支持，允许用户针对通过basic auth或mTLS保护的chart存储库对自己进行身份验证。

基本身份验证

chart存储库可以通过基本访问验证或基本身份验证（basic auth）进行保护。这要求用户提供有效的用户名/口令的组合来访问服务器上的资源。

服务器可以通过在处理请求之前首先检查Authorization头来实现基本身份验证。传入的基本身份验证头类似于以下内容：

```
Authorization: Basic bXl1c2VyOm15cGFzcw== ❶
```

❶ 这里的隐蔽字符串是username+":"+password的Base64编码。

 Authorization头的内容没有加密，因此强烈建议你在提供基本身份验证凭据时也使用HTTPS。

首次添加存储库时，可以在命令行中提供用户名和口令组合，这将指示Helm在对此存储库发出请求时使用基本访问验证：

```
$ helm repo add mycharts http://localhost:8080 --username myuser \
    --password mypass
"mycharts" has been added to your repositories
```

客户端证书

大多数通过HTTPS进行的客户端-服务器通信都允许客户端基于服务器提供的SSL证书来验证服务器的身份。通过相互TLS身份验证（mTLS），服务器还可以基于客户端在TLS握手期间提供的单独SSL证书来验证客户端的身份。

下面是一个简单的Nginx服务器配置，支持chart存储库的mTLS，假设静态文件（例如，index.yaml和.tgz文件）位于服务器上的/chartrepo目录中：

```
events { }
http {
    server {
        root /chartrepo;
        listen 443 ssl;
        server_name charts.example.com;
        ssl_certificate /certs/server.crt; ❶
        ssl_certificate_key /certs/server.key; ❷
        ssl_client_certificate /certs/client-ca.pem; ❸
        ssl_verify_client on;
        proxy_set_header SSL_CLIENT_CERT $ssl_client_cert;
    }
}
```

❶ 服务器的SSL证书。

❷ 服务器的私钥。

❸ 用于客户端身份验证的证书颁发机构（CA）仅接受来自具有此CA签名的证书的客户端的请求。

获取客户端证书的第一步是生成新的私钥和证书签名请求（CSR）：

```
$ mkdir -p ~/client-certs/
$ cd ~/client-certs/
$ openssl genrsa -out francis.key 4096
$ openssl req -new -key francis.key -out francis.csr
```

当生成CSR时如果提示输入“通用名称”（Common Name），则必须输入一个值。使用能识别客户端的输入（例如，“francis”）。从技术上讲，其他字段可以留空，但鼓励你填写。

接下来，使用服务器上配置的证书颁发机构（client-ca.pem）以及相关的私钥（client-ca.key），从CSR生成新的客户端证书：

```
$ openssl x509 -req -in francis.csr \
    -CA /certs/client-ca.pem -CAkey /certs/client-ca.key \
    -out francis.crt -sha256
```

现在，你可以通过在添加新的chart存储库时指定--cert-file和--key-file选项来使用此证书进行身份验证：

```
$ helm repo add client-cert-repo https://charts.example.com \
    --cert-file ~/client-certs/francis.crt --key-file ~/client-certs/francis.key
"client-cert-repo" has been added to your repositories
```

在服务器使用自签名证书的情况下，还可以指定--ca-file选项，指向受信任的证书或证书包：

```
$ helm repo add client-cert-repo-selfsigned https://charts.example.com \
    --cert-file ~/client-certs/francis.crt --key-file ~/client-certs/francis.key
    --ca-file /certs/server.crt
"client-cert-repo-selfsigned" has been added to your repositories
```

 用于--cert-file、--key-file和--ca-file的路径都存储在绑定到存储库的Helm缓存中。千万记住，不要移动这些文件。否则，由于缺少对客户端进行身份验证所需的文件，将来对存储库的请求将会失败。

有关mTLS的更多信息，请参阅Internet Engineering Task Force(IETF)RFC 8446, “The Transport Layer Security(TLS)Protocol Version 1.3”。

7.2.3 真实示例：使用GitHub页面

GitHub有一个免费的静态托管解决方案，称为GitHub Pages。如果你不介意向全世界公开你的chart，GitHub Pages是托管chart存储库的一个很好的选择。

此外，GitHub Pages允许你使用指向GitHub Pages站点的自定义域名。在本节中，我们将展示如何使用GitHub Pages轻松地设置公共Helm chart存储库。

GitHub Pages有一些限制（如带宽），因此在使用此方法之前，请先列举chart存储库的性能要求，并与GitHub的GitHub Pages功能文档进行比较。

创建一个新Git repo

第一步是在GitHub上创建一个专门用于chart存储库的全新Git repo。从技术上讲，你可以将chart repo与其他内容一起托管，但为

了简单起见，我们将使用专用的Git repo。图7-1显示了如何设置新的存储库。

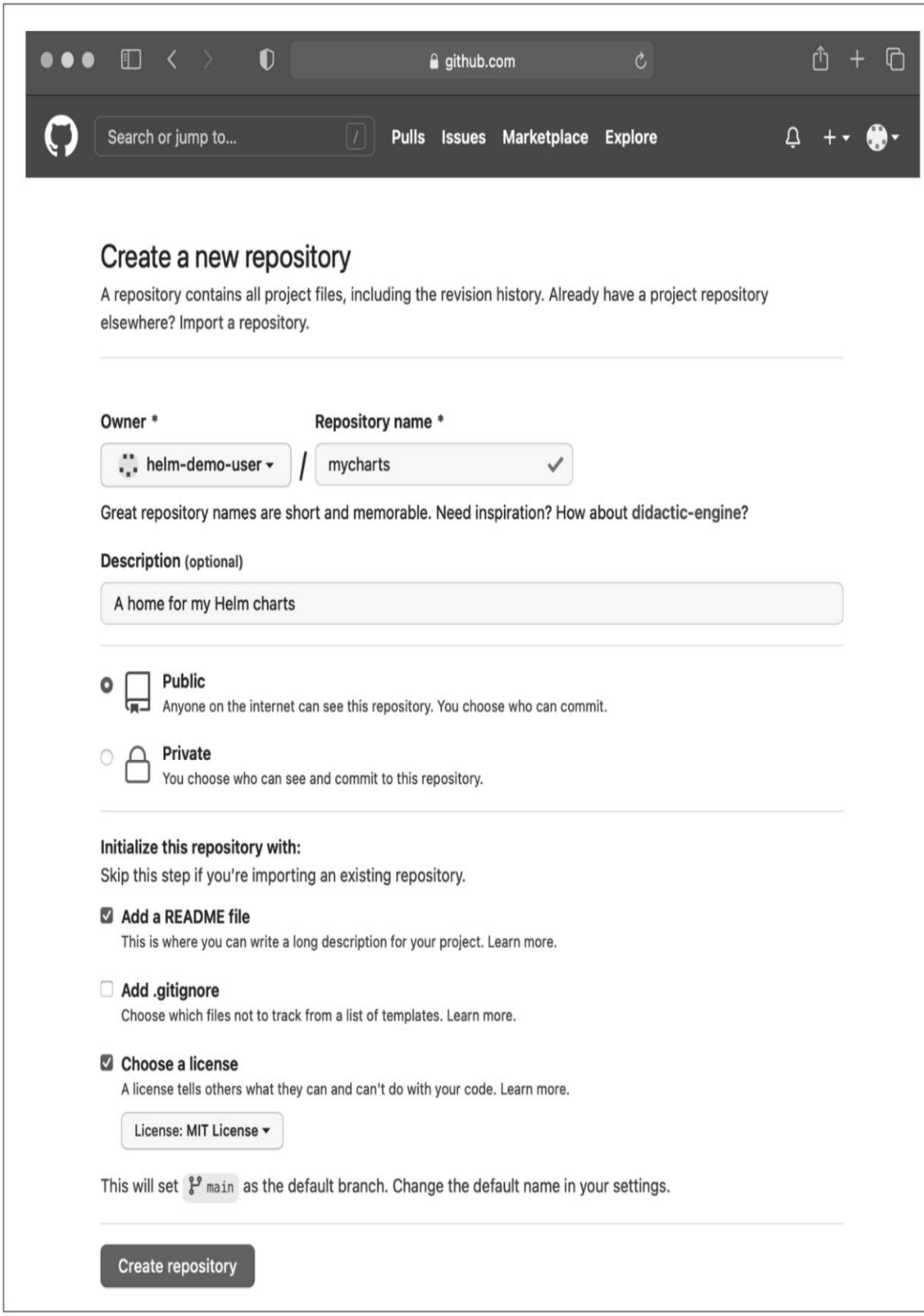


图7-1：在GitHub中创建新的公共存储库

登录到GitHub后，单击屏幕右上角并选择“New repository”（新建存储库）。你可以对Git repo任意命名。在这个例子中，我们将其命名为mycharts。确保选择了将存储库标记为“Public”的选项，这是使用GitHub Pages的先决条件。选择“Initialize this repository with a README”复选框，这将允许我们立即克隆repo。请任意选择一个许可证，例如“MIT License”，以表明此repo中的源代码可以自由使用和重新调整用途。最后，单击“Create repository”（创建存储库）。

 在此上下文中，需要注意Helm repo（或chart存储库）和GitHub上托管的Git repo（用于版本控制）之间的区别。

启用GitHub页面

定位到存储库上的“Settings”面板。在主设置中，向下滚动到标题为GitHub Pages的部分（参见图7-2）。对于Source选项，选择“main branch”。这将导致GitHub在你每次对主分支进行新的提交时重新部署GitHub Pages站点。单击“Save”。

图7-2：在存储库中启用GitHub页面

可选：使用自定义域

默认情况下，GitHub Pages上的站点驻留为github.io域下面的一个子域。例如，你的网站的URL类似于

https://yourusername.github.io/mycharts/。

如果要使用自定义域名，请在你注册者的Web控制台（或者，在你为权威的域名服务器设置的服务控制台）中，创建指向yourusername.github.io的新DNS记录，如果使用根域，请使用ALIAS记录类型；如果使用子域，请使用CNAME记录类型。

返回GitHub中的存储库设置。如图7-3所示，在“Custom domain”输入框中，输入为其设置DNS记录的域名。

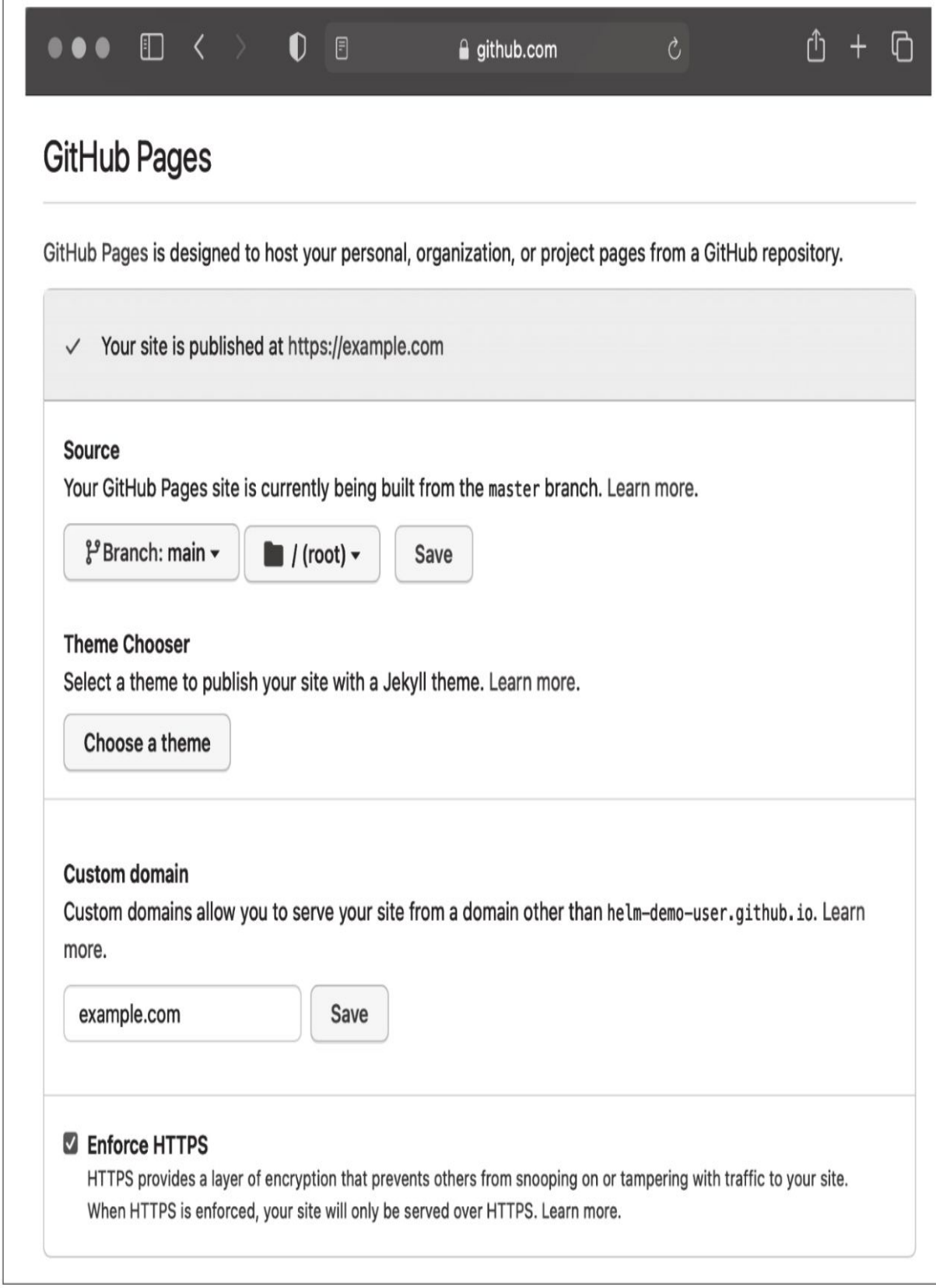


图7-3：为GitHub Pages使用自定义域

GitHub可能需要一个小时才能为你的域生成TLS证书。一旦准备就绪，你应该会看到一些文本显示在设置中，如“Your site is published at https://example.com”。一旦你看到这一点，请确保启用Enforce HTTPS（强制HTTPS）选项，这样你的站点就只能通过HTTPS而不是普通的HTTP进行访问。

添加chart存储库文件

在GitHub UI中找到repo的克隆URL（通常在屏幕的右侧）。将新的GitHub存储库克隆到本地系统，这样我们就可以添加一些文件，将其转换为真正的chart存储库：

```
$ git clone git@github.com:youruser/mycharts.git
Cloning into 'mycharts'...
remote: Enumerating objects: 7, done.
remote: Counting objects: 100% (7/7), done.
remote: Compressing objects: 100% (6/6), done.
remote: Total 7 (delta 0), reused 0 (delta 0), pack-reused 0
Receiving objects: 100% (7/7), done.
```

输入Git存储库的目录：

```
$ cd mycharts/
```

接下来，在一个新的src/目录中创建一个名为spineapple的chart，将其打包到repo根目录中的归档文件中，并创建一个index.yaml文件：

```
$ mkdir -p src/
$ helm create src/pineapple
Creating src/pineapple
$ helm package src/pineapple/
Successfully packaged chart and saved it to: /home/user/mycharts/
  pineapple-0.1.0.tgz
$ helm repo index .
```

完成后，提交并将所有这些新文件推回到GitHub中：

```
$ git add .

$ git commit -m "Add pineapple chart v0.1.0"
[main 9bba19d] Add pineapple chart v0.1.0
 13 files changed, 395 insertions(+)
 create mode 100644 index.yaml
 create mode 100644 pineapple-0.1.0.tgz
 create mode 100644 src/pineapple/.helmignore
 create mode 100644 src/pineapple/Chart.yaml
 create mode 100644 src/pineapple/templates/NOTES.txt
```

```
 create mode 100644 src/pineapple/templates/_helpers.tpl
 create mode 100644 src/pineapple/templates/deployment.yaml
 create mode 100644 src/pineapple/templates/hpa.yaml
 create mode 100644 src/pineapple/templates/ingress.yaml
 create mode 100644 src/pineapple/templates/service.yaml
 create mode 100644 src/pineapple/templates/serviceaccount.yaml
 create mode 100644 src/pineapple/templates/tests/test-connection.yaml
 create mode 100644 src/pineapple/values.yaml

$ git push origin main
Enumerating objects: 20, done.
Counting objects: 100% (20/20), done.
Delta compression using up to 12 threads
Compressing objects: 100% (17/17), done.
Writing objects: 100% (19/19), 9.29 KiB | 4.64 MiB/s, done.
Total 19 (delta 0), reused 0 (delta 0)
To github.com:youruser/mycharts.git
   4964b76..9bba19d  main -> main
```

在浏览器中返回GitHub。在推送更改和这些更改在GitHub Pages站点上生效之间有一个小的延迟。单击右侧边栏中的“Environments”项将显示上次部署站点的时间。如果你看到一个对你刚刚推送的提交（上例中的9bba19d）的引用，那么你的GitHub Pages站点就可以使用了。

将GitHub Pages站点用作chart存储库

一旦你推送了一个index.yaml文件到你的Git repo，并且该站点带有最新提交，则你可以像使用任何其他chart存储库一样使用它。

将GitHub Pages chart存储库添加到本地存储库：

```
$ helm repo add gh-pages https://yourusername.github.io/mycharts/
```

或者，如果你使用的是自定义域：

```
$ helm repo add gh-pages https://example.com
```

7.3 使用chart存储库

一旦有了一个能工作的chart存储库，就可以使用Helm CLI来利用它。

顶级helm repo子命令下有几个命令可用于处理chart存储库。本节将重点介绍如何有效地使用这些命令。

7.3.1 添加存储库

使用chart存储库的第一步是为它指定一个唯一的名称（比如mycharts），并将它添加到Helm已知的存储库列表中。当你第一次添加存储库时，Helm从提供的URL中获取index.yaml，并将其存储在本地。

使用helm repo add命令添加存储库：

```
$ helm repo add mycharts http://localhost:8080
"mycharts" has been added to your repositories
```

如果你正在运行Python示例，请查看chart存储库的日志，你应该会看到传入的GET/index.yaml请求：

```
127.0.0.1 - - [06/May/2020 15:31:07] "GET /index.yaml HTTP/1.1" 200 -
```

7.3.2 下载chart

要直接从存储库下载chart，可使用helm pull命令：

```
$ helm pull mycharts/superapp
```

Helm将根据语义版本控制自动查找最新版本。你还可以指定一个版本：

```
$ helm pull mycharts/superapp --version 0.1.0
```

这将在你的工作区中生成一个新的chart存档（.tgz文件）：

```
$ ls *.tgz
superapp-0.1.0.tgz
```

然后可以直接安装此归档文件：

```
$ helm install superapp-dev superapp-0.1.0.tgz
```

你还可以直接从添加的存储库安装chart：

```
$ helm install superapp-dev mycharts/superapp
```

7.3.3 列出存储库

了解哪些chart存储库已经添加到你的系统中通常是有帮助的，这样我们可以决定是使用其中一个来下载chart，还是从系统中完全删除其中的一个。

使用helm repo list命令可以列出添加到系统中的所有chart存储库：

```
$ helm repo list
NAME    URL
mycharts        http://localhost:8080
```

如果需要的话，你还可以利用--output/-o选项以机器可读的格式获取该文件。

通过添加-o yaml以YAML的形式获取列表：

```
$ helm repo list -o yaml
- name: mycharts
  url: http://localhost:8080
```

通过添加-o json以JSON的形式获取列表：

```
$ helm repo list -o json
[{"name":"mycharts","url":"http://localhost:8080"}]
```

7.3.4 更新存储库

新chart版本发布后，存储库所有者将.tgz包添加到repo存储，并用一个新条目更新index.yaml。

为了获取存储库索引的最新版本，可使用helm repo update命令：

```
$ helm repo update
Hang tight while we grab the latest from your chart repositories...
...Successfully got an update from the "mycharts" chart repository
Update Complete. Happy Helming!
```

如果再次运行Python示例，你应该能在你的chart存储库的输出日志中注意到一个传入的GET/index.yaml请求：

```
127.0.0.1 - - [06/May/2020 15:31:07] "GET /index.yaml HTTP/1.1" 200 -
```

无论存储库索引是否更改了内容（我们没有向myrepo添加更多的chart），此文件都会被提取并下载到缓存中，覆盖先前保存的版本。

7.3.5 删除存储库

可以使用helm repo remove命令删除存储库：

```
$ helm repo remove mycharts
"mycharts" has been removed from your repositories
```

这将删除存储在Helm缓存中的对此存储库的所有引用。

7.4 实验性OCI支持

Helm的OCI支持（https://oreil.ly/eH4KE）仍然被认为是高度实验性的。虽然这方面的开发仍在进行中，但本节所述的语法可能很快就会过时。

chart存储库系统的设计意图是为了易于使用。在大多数情况下，这一系统已被证明足以使全球各地的组织分享和分发其Helm chart。

然而，chart存储库确实存在一些关键挑战：

- 它们没有命名空间的概念。一个repo的所有chart都列在一个索引中。
- 它们没有细粒度的访问控制。你要么可以访问repo中的所有chart，要么不能访问任何chart。
- 具有不同名称但原始内容完全相同的chart包被存储了两次。
- 存储库索引可能变得非常大，导致Helm消耗大量内存。

与其尝试添加特性来解决当前chart存储库模型中的这些问题，不如在符合OCI分发规范（OCI Distribution Specification）的登记站之上构建下一代chart存储库。

OCI代表开放式容器倡议（Open Container Initiative）。摘自网站https://opencontainers.org，OCI的定义如下：

> 一个开放的治理结构，其明确目的是围绕容器格式和运行时创建开放的行业标准。

OCI定义的标准之一是分发规范（distribution specification）。这个规范描述了一个用于分发容器镜像的HTTP API。有趣的是，这个API是通用的，可以应用于除了容器镜像之外的所有程序，比如Helm chart！

从Helm 3.0.0开始，添加了基于OCI的容器登记站的推送、拉取chart的实验支持。

OCI分发规范的历史

Docker在2013年推出了自己的容器引擎，2014年CoreOS推出了rkt（一种具有开放标准的替代引擎），对其提出了挑战。有趣的事实是：Kubernetes的Pod概念直接来自rkt。为了弥合这一分歧，2015年成立了开放式容器倡议，就容器运行时和镜像的开放标准展开合作。

同时，Docker也在开发其登记站API的v2版本。如果你曾经使用过docker pull或docker push，那么底层HTTP调用将基于此API。Docker登记站开始被大规模采用，诞生了Quay.io（https://www.quay.io）等公司。AWS等云服务提供商开始提供自己的托管选择。

2018年，Docker将其登记站v2规范以分发规范（https://oreil.ly/JWYFv）的名义捐赠给OCI，使更广泛的社区能够在Docker的工作基础上继续发展。该规范今天继续发展，已成为一个具有强大API的通用存储解决方案。

7.4.1 启用OCI支持

在撰写本书时，Helm的OCI支持仍被认为是实验性的。

现在，在你的环境中设置以下内容以启用OCI支持：

```
$ export HELM_EXPERIMENTAL_OCI=1
```

7.4.2 运行本地登记站

Docker Distribution项目（也称为Docker登记站）是Docker’s Registry v2 API的原始实现。它开箱即用地支持Helm chart。

如果安装了docker，则可以使用以下命令轻松地在端口5000上的容器中运行本地登记站：

```
$ docker run -d --name oci-registry -p 5000:5000 registry
```

要跟踪你的登记站的日志，可运行以下命令（按Ctrl-C退出）：

```
$ docker logs -f oci-registry
```

要停止登记站，可运行以下命令：

```
$ docker rm -f oci-registry
```

Docker登记站有几个与身份验证、存储等相关的配置选项（https://oreil.ly/CBbOF）。

如果你想用一个用户名-口令的组合来配置基本身份验证，首先要创建一个.htpasswd文件：

```
$ htpasswd -cB -b auth.htpasswd myuser mypass
```

然后启动登记站，挂载.htpasswd文件并设置REGISTRY_AUTH环境变量：

```
$ docker run -d --name oci-registry -p 5000:5000 \
    -v $(pwd)/auth.htpasswd:/etc/docker/registry/auth.htpasswd \
    -e REGISTRY_AUTH="{htpasswd: {realm: localhost, path: /etc/docker/registry \
    auth.htpasswd}}" registry
```

有关Docker Distribution的更多信息，请访问项目GitHub页面（https://oreil.ly/Q8Omf）。

7.4.3 登录到登记站

为了对登记站进行身份验证，可使用helm registry login命令（将提示你手动输入口令）：

```
$ helm registry login -u myuser localhost:5000
Password:
Login succeeded
```

这将使用凭据向登记站上的路径/v2/发出一个简单的GET请求，以确定它们是否有效。如果有效，凭据将存储在Helm配置文件中。如果你启用了任何Docker凭据存储（例如macOS上的osxkeychain），则用户名和口令将安全地存储在那里。

 在localhost:5000上运行本地登记站的示例不使用身份验证。如果你尚未在登记站上启用身份验证，则任何登录凭据组合都将被接受。

7.4.4 注销登记站

要从系统中删除给定登记站的凭据，可使用helm registry logout命令：

```
$ helm registry logout localhost:5000
Logout succeeded
```

7.4.5 在缓存中存储chart

在将chart上传到登记站之前，必须先将其保存到缓存中。这会将chart从其正常状态转换为内容可寻址的blob，并为其提供唯一标识符。

使用helm chart save在缓存中存储chart：

```
$ helm chart save mychart/ localhost:5000/myrepo/mychart
ref:     localhost:5000/myrepo/mychart:2.7.0
digest:  1b251d38cfe948dfc0a5745b7af5ca574ecb61e52aed10b19039db39af6e1617
size:    2.4 KiB
name:    mychart
version: 2.7.0
2.7.0: saved
```

注意，chart引用上使用的标记基于Chart.yaml中chart的版本（2.7.0）。

你还可以使用自定义标记，例如stable，方法是在chart引用的冒号（:）后面指定它：

```
$ helm chart save mychart/ localhost:5000/myrepo/mychart:stable
ref:     localhost:5000/myrepo/mychart:stable
digest:  1b251d38cfe948dfc0a5745b7af5ca574ecb61e52aed10b19039db39af6e1617
size:    2.4 KiB
name:    mychart
version: 2.7.0
stable: saved
```

7.4.6 列出缓存中的chart

使用helm chart list可显示当前存储在缓存中的所有chart：

```
$ helm chart list
REF                                                VERSION DIGEST  SIZE
localhost:5000/myrepo/mychart:2.7.0                2.7.0   84059d7 454 B
localhost:5000/stable/acs-engine-autoscaler:2.2.2 2.2.2   d8d6762 4.3 KiB
localhost:5000/stable/aerospike:0.2.1              0.2.1   4aff638 3.7 KiB
localhost:5000/stable/airflow:0.13.0               0.13.0  c46cc43 28.1 KiB
localhost:5000/stable/anchore-engine:0.10.0        0.10.0  3f3dcd7 34.3 KiB
```

7.4.7 从缓存中导出chart

如果要在缓存中提取chart的源文件，必须首先将其导出到本地目录中。使用helm chart export命令可以导出chart：

```
$ helm chart export localhost:5000/myrepo/mychart:2.7.0
ref:     localhost:5000/myrepo/mychart:2.7.0
digest:  1b251d38cfe948dfc0a5745b7af5ca574ecb61e52aed10b19039db39af6e1617
size:    2.4 KiB
name:    mychart
version: 2.7.0
Exported chart to mychart/
```

chart的名称将用于确定输出路径（例如，mychart/）。

7.4.8 将chart推送到登记站

把一个chart推送（即上传）到登记站中，其他人就可以使用它。一旦你已经登录到登记站并且要推送的chart已保存到缓存中，可以使用helm chart push命令推送chart：

```
$ helm chart push localhost:5000/myrepo/mychart:2.7.0
The push refers to repository [localhost:5000/myrepo/mychart]
ref:     localhost:5000/myrepo/mychart:2.7.0
digest:  1b251d38cfe948dfc0a5745b7af5ca574ecb61e52aed10b19039db39af6e1617
size:    2.4 KiB
name:    mychart
version: 2.7.0
2.7.0: pushed to remote (1 layer, 2.4 KiB total)
```

7.4.9 从登记站中拉取chart

一旦chart被推送到登记站，其他用户就可以拉取（即下载）它们。从登记站中拉取chart会将它们放入本地缓存。要从登记站中拉取现有chart，可使用helm chart pull命令：

```
$ helm chart pull localhost:5000/myrepo/mychart:2.7.0
2.7.0: Pulling from localhost:5000/myrepo/mychart
ref:     localhost:5000/myrepo/mychart:2.7.0
digest:  1b251d38cfe948dfc0a5745b7af5ca574ecb61e52aed10b19039db39af6e1617
size:    2.4 KiB
name:    mychart
version: 2.7.0
Status: Downloaded newer chart for localhost:5000/myrepo/mychart:2.7.0
```

7.4.10 从缓存中删除chart

要从本地缓存中删除chart，可使用helm chart remove命令：

```
$ helm chart remove localhost:5000/myrepo/mychart:2.7.0
2.7.0: removed
```

7.5 相关项目

Helm的chart存储库系统产生了一系列开源工具来进一步增强这种体验。下面介绍与chart存储库相关的一些项目。

7.5.1 ChartMuseum

项目主页：https://github.com/helm/chartmuseum

ChartMuseum是一个简单的chart存储库Web服务器。将其配置为指向包含chart包的存储位置，它将动态生成index.yaml。它还公开了一个用于上传、查询和删除chart包的HTTP API。此外，它还有许多其他身份验证、多租户和缓存的配置设置，这使得它成为用户托管私有或内部chart存储库的流行选择。

ChartMuseum支持的后端

ChartMuseum支持多种存储后端，包括：

- Alibaba Cloud OSS Storage
- Amazon S3
- Baidu Cloud BOS Storage
- DigitalOcean Spaces
- etcd
- Google Cloud Storage
- Local filesystem

- Microsoft Azure Blob Storage
- Minio
- Netease Cloud NOS Storage
- Openstack Object Storage
- Oracle Cloud Infrastructure Object Storage
- Tencent Cloud Object Storage

7.5.2 Harbor

项目主页：https://github.com/goharbor/harbor

Harbor是一个功能齐全的登记站，具有附加的安全和管理功能。它为Helm chart提供了一个UI，并利用后端的ChartMuseum作为多租户chart存储库。它还为Helm的实验性OCI特性集提供支持。

与Helm类似，Harbor是一个优秀的顶级CNCF项目。

7.5.3 Chart Releaser

项目主页：https://github.com/helm/chart-releaser

Chart Releaser（简称为cr）是一个命令行工具，它利用GitHub发行版来托管chart包。它能够检测Git repo中的chart，对它们进行打包，并将它们作为以独特的chart版本命名的工件上传到GitHub发行版中。

使用cr上传chart后，还可以使用该工具基于GitHub版本和相关工件的内容生成index.yaml文件。这个存储库索引可以静态地托管在GitHub Pages或其他地方。

7.5.4 S3插件

项目主页：https://github.com/hypnoglow/helm-s3

S3插件是一个Helm插件，它允许你使用私有的Amazon S3存储桶作为chart存储库。

7.5.5 GCS插件

项目主页：https://github.com/hayorov/helm-gcs

GCS插件是一个Helm插件，它允许你使用私有的Google Cloud Storage存储桶作为chart存储库。

7.5.6 Git插件

项目主页：https://github.com/aslafy-z/helm-git

Git插件是一个Helm插件，它允许你使用包含chart源文件的Git存储库作为chart存储库。它支持子路径、自定义引用以及HTTPS和SSH Git URL。

第8章

Helm插件和启动程序

正如我们在本书中所看到的，Helm有很多特性和方法可以帮助我们在Kubernetes上交付应用程序。此外，也可以对Helm提供的功能进行自定义和扩展。

在本章中，我们将讨论两种进一步增强和定制Helm的用途的方法：插件和启动程序。

插件允许你向Helm添加额外的功能，并与CLI无缝集成，这使它们成为具有独特工作流需求的用户的热门选择。网上有许多第三方插件可用于常见用例，如secret管理。此外，非常容易针对独特的一次性任务来构建插件。

启动程序扩展了使用helm create为不同类型的应用程序生成新的Helm chart的可能性。例如，你可能有一个为内部微服务构建的Helm chart，它非常适合作为未来微服务的示例。那么你可以将chart转换为启动程序，然后每次出现具有类似需求的新项目时都可以使用它。

通过利用插件和启动程序，我们可以在Helm的开箱即用功能的基础上进行构建，以简化和自动化日常的工作流任务。

8.1 插件

Helm插件是可以直接从Helm CLI访问的外部工具。它们允许你向Helm添加自定义子命令，而无须对Helm的Go源代码进行任何修改。这在设计上类似于在其他工具中实现插件系统的方法，比如kubectl（Kubernetes CLI）。

此外，downloader（下载器）插件允许你指定与chart存储库通信的自定义协议。如果你有一些自定义的身份验证方法，或者你需要以某种方式修改Helm从存储库获取chart的方法，那么这将非常有用。

8.1.1 安装第三方插件

许多第三方插件都是开源的，可以在GitHub上公开获得。许多插件使用“helm-plugin”这样的标签使它们易于查找。请参阅GitHub上的Helm插件文档（https://oreil.ly/3KwNb）。

示例公共可用插件

下面是一些GitHub上的Helm插件：

helm/helm-2to3

将Helm 2版本转换为Helm 3版本的插件。

jkroepke/helm-secrets

在YAML格式中有效管理secret的插件。

maorfr/helm-backup

用文本文件备份/恢复Helm版本的插件。

karuppiah7890/helm-schema-gen

基于values.yaml生成values.schema.json的插件（参见第6章以了解更多关于模式化值的信息）。

hickeyma/helm-mapkubeapis

更新包含弃用的Kubernetes API的Helm发布版本元数据的插件。

找到要安装的插件后，获取其版本控制URL。这是获取正确版本的plugin.yaml以及其余的插件源代码的方法。

目前支持Git、SVN、Bazaar（Bzr）和Mercurial（Hg）URL。对于Git，版本控制URL类似于https://example.com/myorg/myrepo.git。

例如，有一个简单的用于管理启动程序的插件，位于https://github.com/salesforce/helm-starter。此插件的版本控制URL为https://github.com/salesforce/helm-starter.git。

要安装此插件，可运行helm plugin install，并将版本控制URL作为第一个参数传入：

```
$ helm plugin install https://github.com/salesforce/helm-starter.git
Installed plugin: starter
```

如果安装成功，你可以继续使用这个插件：

```
$ helm starter --help
Fetch, list, and delete helm starters from github.

Available Commands:
    helm starter fetch GITURL       Install a bare Helm starter from Github
                                      (e.g., git clone)
    helm starter list               List installed Helm starters
    helm starter delete NAME        Delete an installed Helm starter
    --help                          Display this text
```

要列出所有已安装的插件，可以使用helm plugin list命令：

```
$ helm plugin list
NAME    VERSION DESCRIPTION
starter 1.0.0   This plugin fetches, lists, and deletes helm starters from github
```

要尝试更新插件，可以使用helm plugin update命令：

```
$ helm plugin update starter
Updated plugin: starter
```

如果希望从系统中卸载插件，可以使用helm plugin remove命令：

```
$ helm plugin remove starter
Uninstalled plugin: starter
```

除非另有规定，否则Helm将使用plugin.yaml以及安装插件时位于Git repo的默认分支上的源代码。如果希望指定要使用的Git标记，可在安装时使用--version标志：

```
$ helm plugin install https://github.com/databus23/helm-diff.git --version v3.1.0
```

也可以直接从tar压缩包的URL安装插件。Helm将下载tar压缩包并将其解压到插件目录：

```
$ helm plugin install https://example.com/archives/myplugin-0.6.0.tar.gz
```

此外，还可以从本地目录安装插件：

```
$ helm plugin install /path/to/myplugin
```

Helm不会复制这个文件，而是创建指向原始文件的符号链接：

```
$ ls -la "$(helm env HELM_PLUGINS)"
total 8
drwxrwxr-x 2 myuser myuser 4096 Jul  3 21:49 .
drwxrwxr-x 4 myuser myuser 4096 Jul  1 21:38 ..
lrwxrwxrwx 1 myuser myuser   21 Jul  3 21:49 myplugin -> /path/to/myplugin
```

这个功能可能是有用的。例如，如果你正在积极开发一个插件，当调用符号链接的插件时，对plugin.yaml和其他源文件的更改将立即被识别。

8.1.2 自定义子命令

插件有许多有用的特性，可以实现与现有Helm用户体验的无缝集成。Helm插件最显著的特点是每个插件都为Helm提供了一个自定义的顶级子命令。这些子命令甚至可以利用shell完成（本章后面将介绍）。

Helm插件的历史

Helm插件的一个原始特性是为它们提供了连接到Tiller的环境设置，Tiller是Helm 2中弃用的Helm服务器端组件。对于需要与Helm紧密集成的插件子命令来说，这是一个重要的概念，因为任何涉及Helm发布版本的东西都需要经过Tiller。

在Helm 3中，Tiller已被移除了，所有与Kubernetes API的通信都由Helm客户端自己执行。然而，插件系统仍然存在。

安装一个插件后，一个新的命令就将提供给你使用，它按照插件的名称来命名。这个新的命令直接与Helm集成，甚至会出现在helm help中。

例如，假设我们安装了一个名为inspect-templates的插件，它提供了有关在chart中找到的YAML模板的额外信息。这个插件将为你提供一个额外的Helm命令：

```
$ helm inspect-templates [args]
```

这将执行inspect templates插件，并把提供的任何参数或标志传递给插件被调用时执行的底层工具。插件的作者指定了一些命令，Helm应该在每次调用插件时将其作为子进程运行（关于如何指定此命令的更多信息见8.1.3节）。

插件提供了一个很好的替代方案来扩充Helm现有的特性集，而无须对Helm本身进行任何修改。

8.1.3 构建插件

构建Helm插件是一个相当简单的过程。根据插件的需求和总体复杂性，这可能需要一些编程知识。然而，许多插件只运行一个基本的shell命令。

底层实现

考虑下面的Bash脚本inspect-templates.sh，我们的示例inspect-templates插件的底层实现为：

```
#!/usr/bin/env bash
set -e

# First argument on the command line, a relative path to a chart directory
CHART_DIRECTORY="${1}"

# Fail if no chart directory provided or is invalid
if [[ "${CHART_DIRECTORY}" == "" ]]; then
    echo "Usage: helm inspect-templates <chart_directory>"
    exit 1
elif [[ ! -d "${CHART_DIRECTORY}" ]]; then
    echo "Invalid chart directory provided: ${CHART_DIRECTORY}"
    exit 1
fi

# Print a summary of the chart's templates
cd "${CHART_DIRECTORY}"
```

```
cd templates/
echo "--------------------"
echo "Chart template summary"
echo "--------------------"
echo ""
total="$(find . -type f -name '*.yaml' -maxdepth 1 | wc -l | tr -d '[:space:]')"
echo " Total number: ${total}"
echo ""
echo " List of templates:"
for filename in $(find . -type f -name '*.yaml' -maxdepth 1 | sed 's|^\./||'); do
    kind=$(cat "${filename}" | grep kind: | head -1 | awk '{print $2}')
    echo "  - ${filename} (${kind})"
done
echo ""
```

当用户运行helm inspect-templates时，Helm将在后台执行此脚本。

 底层插件实现不需要用Bash、Go或任何特定的编程语言编写。对于这个插件的最终用户来说，它应该只是Helm CLI的另一部分。

插件清单

每个插件都由一个名为plugin.yaml的YAML文件定义，此文件包含插件元数据和有关调用插件时要运行的命令的信息。

下面是一个用于inspect-templates插件的plugin.yaml示例：

```
name: inspect-templates ❶
version: 0.1.0 ❷
description: get a summary of a chart's templates ❸
command: "${HELM_PLUGIN_DIR}/inspect-templates.sh" ❹
```

❶ 插件的名称。

❷ 插件的版本。

❸ 插件的基本描述。

❹ 调用此插件时要运行的命令。

手动安装

首先检查插件存储根目录的配置路径：

```
$ HELM_PLUGINS="$(helm env HELM_PLUGINS)"
$ echo "${HELM_PLUGINS}"
/home/myuser/.local/share/helm/plugins
```

对插件使用自定义根目录

通过为当前环境中的HELM_PLUGINS环境变量提供自定义路径，可以覆盖插件的根目录。

在插件存储根目录中创建一个与插件名称（inspect-templates）匹配的目录：

```
$ PLUGIN_ROOT="${HELM_PLUGINS}/inspect-templates"
$ mkdir -p "${PLUGIN_ROOT}"
```

下一步，将plugin.yaml和inspect-templates.sh复制到新目录，并确保脚本是可执行的：

```
$ cp plugin.yaml "${PLUGIN_ROOT}"
$ cp inspect-templates.sh "${PLUGIN_ROOT}"
$ chmod +x "${PLUGIN_ROOT}/inspect-templates.sh"
```

最终结果

下面是inspect templates插件的运行结果：

```
$ helm inspect-templates
Usage: helm inspect-templates <chart_directory>
Error: plugin "inspect-templates" exited with error

$ helm inspect-templates nonexistant/
Invalid chart directory provided: nonexistant/
Error: plugin "inspect-templates" exited with error

$ helm create mychart
Creating mychart

$ helm inspect-templates mychart/
---------------------
Chart template summary
---------------------

 Total number: 5

 List of templates:
  - serviceaccount.yaml (ServiceAccount)
  - deployment.yaml (Deployment)
```

- service.yaml (Service)
- hpa.yaml (HorizontalPodAutoscaler)
- ingress.yaml (Ingress)

注意所提供的命令行参数（即mychart/）是如何直接传递给脚本的。这使得插件作者可以轻松地构建接受任意数量的参数或自定义标志的独立工具。

8.1.4 plugin.yaml

plugin.yaml是描述插件、插件的调用命令和其他重要细节的插件清单文件的名称。

下面是一个plugin.yaml示例，包含所有你可能为插件指定的选项：

```
name: myplugin ❶
version: 0.3.0 ❷
usage: "helm myplugin --help" ❸
description "a plugin that belongs to me" ❹
platformCommand: ❺
  - os: windows
    arch: amd64
    command: "bin/myplugin.exe"
command: "bin/myplugin" ❻
ignoreFlags: false ❼
hooks: ❽
  install: "scripts/install-hook.sh"
  update: "scripts/update-hook.sh"
  delete: "scripts/delete-hook.sh"
downloaders: ❾
  - command: "bin/myplugin-myp-downloader"
    protocols:
      - "myp"
      - "myps"
```

❶ 插件的名称。

❷ 插件的版本。

❸ 插件的使用说明。

❹ 插件的描述。

❺ 平台特定命令。如果客户端匹配某os/arch组合，则运行该命令，而不是运行默认的命令。

❻ 调用此插件时运行的命令。

❼ 当作为参数传递给插件时，是否抑制传递的Helm全局标志（例如，--debug）。

⑧ 插件钩子（参见8.1.5节）。

⑨ 下载程序和相关协议（参见8.1.6节）。

插件的名称（name）将是用于从Helm CLI调用该插件的子命令（例如，helm myplugin）。因此，插件名称不应与任何现有的Helm子命令（install、repo等）重名。而且名称只能包含字符a－z、A－Z、0－9、_和-。

插件的version应该是有效的SemVer 2版本。

运行helm help和helm help myplugin将显示插件的用法（usage）和说明（description）。但是，插件本身必须处理自己的标志解析，比如helm myplugin--help。

command是当这个插件被调用时，Helm将在子进程中执行的命令。如果定义了platformCommands部分，Helm将首先检查系统是否与提供的os（操作系统）和arch（架构）匹配，如果匹配，Helm将使用匹配条目中定义的命令（command）。arch字段是可选的，如果缺少，则只检查os。

下面是Helm根据插件的plugin.yaml内容以及运行时环境确定在调用插件时运行哪个命令的确切顺序：

1. 如果platformCommand存在，将首先搜索它。

2. 如果os和arch都匹配当前的平台，搜索将停止并执行特定于平台的命令。

3. 如果os匹配，并且没有更具体的匹配，则将执行特定于平台的命令。

4. 如果找不到匹配的os/arch，将执行默认的顶级命令。

5. 如果没有顶级命令，并且在platformCommand中找不到匹配项，Helm将退出并返回错误。

8.1.5 钩子

插件钩子允许你在安装、更新或删除插件时执行其他操作。

例如，插件的底层实现可能是必须从Internet下载的特定于平台的二进制文件。二进制文件的URL因用户的操作系统而异。

基于操作系统处理此逻辑的脚本可能如下所示：

```
#!/usr/bin/env bash

set -e

URL=""
EXTRACT_TO=""

if [[ "$(uname)" = "Darwin" ]]; then
    URL="https://example.com/releases/myplugin-mac"
    EXTRACT_TO="myplugin"
elif [[ "$(uname)" = "Linux" ]]; then
    URL="https://example.com/releases/myplugin-linux"
    EXTRACT_TO="myplugin"
else
    URL="https://example.com/releases/myplugin-windows"
    EXTRACT_TO="myplugin.exe"
fi

mkdir -p bin/
curl -sSL "${URL}" -o "bin/${EXTRACT_TO}"
```

通过为我们的插件定义一个安装钩子，可以让它在用户安装这个插件时运行这个脚本。

要定义钩子，可以将hooks部分添加到plugin.yaml中，为要响应的每个事件定义命令：

```
...
hooks:
  install: "scripts/install-hook.sh" ❶
  update: "scripts/update-hook.sh" ❷
  delete: "scripts/delete-hook.sh" ❸
```

❶ 在helm plugin install上运行的命令。

❷ 在helm plugin update上运行的命令。

❸ 在helm plugin remove上运行的命令。

8.1.6 下载程序插件

有些插件有特殊的功能，可以作为下载chart的替代品。

如果你以不同于纯chart存储库的方式存储chart，或者你的chart存储库实现有额外的要求，那么这将非常有用。

下载程序插件（downloader plugin）定义了一个（或多个）协议，如果在命令行中检测到，它将指示Helm使用插件而不是Helm的内部下载机制来下载index.yaml或者chart.tgz包。

下面是一个名为“super-secure”的下载程序插件的plugin.yaml示例，它注册了ss://协议：

```
name: super-secure
version: 0.1.0
description: a super secure chart downloader
command: "${HELM_PLUGIN_DIR}/super-secure.sh"
downloaders:
  - command: "super-secure-downloader.sh" ❶
    protocols: ❷
      - "ss"
```

❶ 此插件作为下载程序调用时要运行的命令。

❷ 此插件声明的自定义协议。

 记住，所有插件（包括下载程序插件）都定义了一个自定义的顶级命令（即helm super-secure）。用于下载程序插件的命令可以与command字段相同，但要注意，如果你希望将该插件同时用作标准插件和下载程序，那么确定它的使用方式可能会变得很困难。确定插件是否用作下载程序的一种方法是检查是否使用四个命令行参数调用该命令。

下载程序命令总是使用以下参数调用：

```
<command> certFile keyFile caFile full-URL
```

certFile、keyFile和caFile参数派生自YAML配置文件中的条目，其路径由$(helm env helm_REPOSITORY_CONFIG)返回，并在使用helm repo add添加存储库时设置（更多信息请参阅第7章）。full-URL参数

是正在下载的资源的完整URL，可以是index.yaml或chart.tgz/.prov文件。

让我们看看super-secure插件定义的ss://协议下载程序的具体实现：

```
#!/usr/bin/env bash
set -e

# The fourth argument is the URL to the resource to download from the repo
URL="${4}"

# Replace "ss://" with "https://"
URL="$(echo ${URL} | sed 's/ss:/https:/')"

# Request the resource using the token, outputting contents to stdout
echo "Downloading $(basename ${URL}) using super-secure plugin..." 1>&2
curl -sL -H "Authorization: Bearer ${SUPER_SECURE_TOKEN}" "${URL}"
```

此下载程序允许我们使用受令牌/承载身份验证保护的chart存储库。它期望设置环境变量SUPER_SECURE_TOKEN，该环境变量将用于制定从chart存储库请求资源时使用的头Authorization:Bearer<token>。

 虽然super-secure插件是一个很好的例子，但Helm的未来版本实际上可能支持bearer token auth开箱即用。

下载程序插件需要将资源的内容输出到stdout，因此任何额外的日志等都应该打印到stderr。这就是在以echo开头的行中，我们使用1>&2将此消息重定向到stderr的原因。

安装了这个插件后，我们将添加一个受token auth保护的chart存储库：

```
$ export SUPER_SECURE_TOKEN="abc123"
$ helm repo add my-secure-repo ss://secure.example.com
Downloading index.yaml using super-secure plugin...
"my-secure-repo" has been added to your repositories
```

此存储库URL现在将显示在本地存储库列表中，其中包含ss://协议：

```
$ helm repo list
NAME          URL
my-secure-repo  ss://secure.example.com
```

现在，此存储库可以像其他存储库一样用于下载远程chart包：

```
$ export SUPER_SECURE_TOKEN="abc123"
$ helm pull my-secure-repo/superapp
Downloading superapp-0.1.0.tgz using super-secure plugin...
$ ls
superapp-0.1.0.tgz
```

下载程序插件为Helm用户提供了一种方法，通过定义自定义协议来扩展使用chart存储库的传输机制。当Helm检测到正在使用一个自定义协议时，它将尝试定位一个可以处理它的已安装插件，然后将资源请求推送到该插件中。

8.1.7 执行环境

由于插件是为了扩展Helm的功能，它们可能需要访问Helm的一些内部配置文件，或者命令行上提供的全局标志。

为了让插件访问此类信息，在运行时，Helm向插件提供了一系列已知的环境变量。

以下是插件可用的所有环境变量的最新列表，按字母顺序排列：

HELM_BIN

正在执行的Helm命令的路径。

HELM_DEBUG

为全局布尔--debug选项（“true”或“false”）设置的值。

HELM_KUBECONTEXT

为全局--kube-context<context>选项设置的值。

HELM_NAMESPACE

为全局--namespace<namespace>选项设置的值。

HELM_PLUGIN_DIR

当前插件的根目录。

HELM_PLUGIN_NAME

当前插件的名称。

HELM_PLUGINS

包含所有插件的顶级目录。

HELM_REGISTRY_CONFIG

登记站配置的根目录。

HELM_REPOSITORY_CACHE

存储库的根目录。

HELM_REPOSITORY_CONFIG

存储库配置的根目录。

8.1.8 Shell完成

Helm内置了对Bash和Z shell（Zsh）的shell自动完成的支持（参见helm completion--help）。当你忘记要使用的子命令或标志的名称时，这会很有帮助。

插件还可以使用两种方法之一提供自己的自定义shell完成：静态自动完成和动态完成。

静态自动完成

通过在插件目录的根目录中包含一个名为completion.yaml的文件，Helm插件可以静态地指定插件可用的所有预期标志和命令。

下面是一个用于虚构的zoo插件的示例completion.yaml：

```
name: zoo ❶
flags: ❷
  - disable-smells
  - disable-snacks
commands: ❸
  - name: price ❹
```

```
  flags:
    - kids-discount
- name: animals
  commands:
    - name: list
      validArgs: ❺
        - birds
        - reptiles
        - cats
    - name: describe
      flags:
        - format-json
      validArgs:
        - birds
        - reptiles
        - cats
```

❶ 此完成文件使用的插件的名称。

❷ 可用标志的列表（注意，这些不应包括-或--前缀）。

❸ 可用的子命令列表。

❹ 单个子命令的名称。

❺ 子命令后面第一个参数的有效选项列表。

在顶层commands部分下面，可以为嵌套的子命令指定另一个commands部分（并根据需要递归多次）。commands部分中的每个命令都可以包含自己的标志（flags）和有效值（validArgs）的列表。

Helm的全局标志（如--debug或--namespace）已经由Helm的内置shell完成，因此不必在flags下列出这些标志。

如果我们开始尝试运行示例zoo插件，然后按Tab键，它将显示所有可用的子命令：

```
$ helm zoo  # (click tab)
animals  price
```

现在，如果我们做同样的操作，但是在按下Tab键之前添加--disable-s前缀（原文后缀错），则应该看到标志如下：

```
$ helm zoo --disables-s  # (click tab)
--disable-smells  --disable-snacks
```

使用静态完成，我们能够实现与Helm的现有shell完成对等的功能，使插件感觉与Helm用户体验更紧密地集成在一起。

 如果你正在开发一个插件，则必须打开一个新的终端窗口，以便刷新静态shell完成。

或者，你可以运行以下命令之一以获取当前终端中的最新完成：

```
source <(helm completion bash) # for Bash
source <(helm completion zsh)  # for Z shell
```

动态完成

在某些情况下，可能无法提前知道给定命令的有效参数。例如，你可能希望在集群中提供一个Helm发布版本名称列表来作为插件的有效参数。这可以通过使用动态完成来实现。

要启用动态完成，可以在插件目录的根目录中包含一个名为plugin.complete的文件。此文件可以是任何类型的可执行文件。例如，shell脚本或二进制文件。

对于包含plugin.complete文件的插件，当请求完成时（即，按Tab键），Helm将运行此可执行文件，并将需要完成的文本作为第一个参数传递。然后，这个程序应该返回一个可能结果的列表，每个结果占一行，并成功退出（即返回代码0）。

你甚至可以决定将此完成功能作为主插件程序的一部分来提供，具体可以使用一个简单的包装器脚本，然后通过使用--complete等标志来触发它。下面是一个基本的plugin.complete可执行文件，它执行以下操作：

```
#!/usr/bin/env sh
$HELM_PLUGIN_DIR/my-plugin-program --complete "$@"
```

在zoo插件示例的基础上，假设可用动物类别的列表不断变化，并存储在用户主目录的一个名为animals.txt的文件中。以下是animals.txt可能的内容：

```
birds
reptiles
cats
```

我们希望能够基于此文件的内容动态地提供完成。下面是一个可用于提供动态完成的plugin.complete可执行文件（Bash脚本）：

```
#!/usr/bin/env bash
set -e
INPUT="${@}"
if [[ "${INPUT}" == "animals list"* ]]; then
    INPUT="$(echo "${INPUT}" | sed -e 's/^animals list //')"
    for flag in $(cat "${HOME}/animals.txt"); do
        if [[ "${flag}" == "${INPUT}"* ]]; then
            echo "${flag}"
        fi
    done
fi
```

现在，如果我们运行这个插件并键入animals list，然后按Tab键，它将显示所有可用动物类别的列表：

```
$ helm zoo animals list  # (press Tab key)
birds     cats      reptiles
```

为了确保它是动态的，让我们在animals.txt中添加一个种类“monkeys”并再试一次：

```
$ echo "monkeys" >> "${HOME}/animals.txt"
$ helm zoo animals list  # (press Tab key)
birds     cats      monkeys   reptiles
```

成功了！

这只是一个使用动态完成的简单示例，但是请记住，你还可以查询一些远程的东西，例如Kubernetes集群中的资源，这使它成为插件的一个强大功能。

 如果你已经在使用completion.yaml文件，则不能使用动态完成，即使plugin.complete可执行文件存在于插件的根目录中。

8.2 启动程序

启动程序或启动程序包类似于Helm chart，只不过它们专门用来作为新chart的模板。

当你使用helm create命令创建一个新chart时，这将使用Helm的内置启动程序生成一个新chart，这是一个使用最佳实践的通用chart。

要指定自定义启动程序，可以在创建新chart时使用--starter选项：

```
$ helm create --starter basic-webapp superapp
```

使用启动程序允许我们利用以前为具有类似用途的应用程序构建的chart。这对于引导具有类似需求的新项目快速部署到Kubernetes环境非常有用。

8.2.1 将chart转换为启动程序

任何Helm chart都可以被转换成启动程序。唯一能区分启动程序与标准chart的是启动程序的模板中存在对chart名称的动态引用。

要将一个标准chart转换为启动程序，需要用字符串<CHARTNAME>替换对chart名称的任何硬编码引用。

为了演示，让我们从一个名为mychart的chart中选取一个简单的ConfigMap模板：

```
apiVersion: v1
kind: ConfigMap
metadata:
  name: {{ include "mychart.fullname" . }}
  labels:
    {{- include "mychart.labels" . | nindent 4 }}
data:
  hello: {{ .Values.hello | quote }}
```

下面是启动程序中的模板：

```
apiVersion: v1
kind: ConfigMap
metadata:
  name: {{ include "<CHARTNAME>.fullname" . }}
  labels:
    {{- include "<CHARTNAME>.labels" . | nindent 4 }}
data:
  hello: {{ .Values.hello | quote }}
```

 此chart必须仍然包含Chart.yaml才能工作，但是，它将被生成器覆盖。

8.2.2 使启动程序可供Helm使用

在使用启动程序之前，必须首先确定它的唯一名称。例如，将包含用于部署基本Web应用程序的样板模板的启动程序命名为“basic-webapp”。

要使此启动程序成为在命令行上指定--starter标志时使用的有效选项，它必须作为文件路径$(helm env HELM_DATA_HOME)/starters下的目录存在。

如果这是你要添加的第一个启动程序，请确保顶层starters目录首先存在：

```
$ export HELM_STARTERS="$(helm env HELM_DATA_HOME)/starters"
$ mkdir -p "${HELM_STARTERS}"
```

然后将整个basic-webapp目录复制到顶层目录：

```
cp -r basic-webapp "${HELM_STARTERS}"
```

8.2.3 使用启动程序

一旦启动程序可用，你就可以在命令行中引用其名称来生成新的chart：

```
$ helm create --starter basic-webapp superapp
Creating superapp
```

新生成的chart的结构将与启动程序的结构相同。启动程序的模板中对<CHARTNAME>的所有引用都将被替换为新chart的名称（即superapp）。

下面是基于启动程序生成的chart的示例目录结构，该启动程序只定义了deployment.yaml和service.yaml这两个模板：

```
$ tree superapp/
superapp/
├── Chart.yaml
├── templates
│   ├── _helpers.tpl
│   ├── deployment.yaml
│   └── service.yaml
└── values.yaml
```

从这里，你可以将这个新chart签入版本控制并开始进行更改，以便为给定的应用程序定制它。

8.3 进一步扩展Helm

在本章中，我们讨论了如何使用插件和启动程序来扩展Helm。然而，还有一种方法可以扩展Helm：通过开源贡献。

本书中的所有内容都反映了对Helm项目的数千个开源贡献。虽然这项工作大部分是由维护人员完成的，但大部分贡献来自世界各地的个人。这不仅包括对Go源代码的更改，还包括测试和文档更新。

欢迎大家到Helm社区登录页（https://oreil.ly/9TloH）了解更多信息！

附录A
chart API版本

本附录介绍了chart API的第2版和第1版（旧版）之间的差异。

chart API版本在每个chart的Chart.yaml文件中指定，并由Helm用于确定如何解析chart以及哪些功能集可用。

对于新chart，通常应使用第2版的API。然而，许多公开可用的chart是在API第2版出现之前创建的，并且使用的是旧版本，即第1版的API。在这里，我们将详细介绍这两个API版本以及它们的不同之处。

API第2版

chart API第2版是Helm 3中引入的当前API版本。这是使用helm create创建新chart时使用的默认API版本。

使用第2版API的chart被Helm 3支持，但不一定被Helm 2支持。如果你只打算支持Helm 3及更高版本，建议你只使用第2版API。

Chart.yaml文件

下面是一个使用API第2版的chart的Chart.yaml文件：

```
apiVersion: v2 ❶
name: lemon
version: 1.2.3
```

```
type: application
description: When life gives you lemons, do the DevOps
appVersion: 2.0.0
home: https://example.com
icon: https://example.com/img/lemon.png
sources:
  - https://github.com/myorg/mychart
keywords:
  - fruit
  - citrus
maintainers:
  - name: Carly Jenkins
    email: carly@mail.cj.example.com
    url: https://cj.example.com
  - name: William James Spode
    email: william.j@mail.wjs.example.com
    url: https://wjs.example.com
deprecated: false
annotations:
  sour: 1
kubeVersion: ">=1.14.0"
dependencies:
  - name: redis
    version: ~10.5.7
    repository: https://kubernetes-charts.storage.example.com/
    condition: useCache,redis.enabled
  - name: postgresql
    version: 8.6.4
    repository: @myrepo
    tags:
      - database
      - backend
```

❶ 表示chart API第2版。

下面将详细描述此文件中的每个顶级字段。

字段：apiVersion

必需的

此chart的API版本。

此字段应始终设置为v2。

字段：name

必需的

chart的名称。

在大多数情况下，该字段与应用程序的名称（即lemon）一一对应。如果你的应用程序被分解成多个可安装的组件，通常会在这个名称后面加上组件的描述。例如，lemon-frontend、lemon-backend等。

chart名称必须由小写字母、数字和连字符（-）组成。

字段：version

必需的

chart的当前版本，严格使用Semantic Versioning 2（语义版本控制2）进行格式化。

对Helm chart进行版本控制

良好的语义版本控制非常有帮助。它让操作人员知道在软件升级过程中会发生什么。你可能已经看到并使用了语义版本。它们由三个数字组成，以句点（.）分隔，如3.1.2。语义版本遵循格式

MAJOR.MINOR.PATCH（主版本.次要版本.补丁版本），它们各自版本号增加的规则如下：

主版本

表示所做的更改具有突破性，不向后兼容（例如，3.1.2→4.0.0）

次要版本

表示存在向后兼容的新功能（例如，3.1.2→3.2.0）

补丁版本

表示已修复一个或多个错误，并且所做的更改使chart按最初的预期工作，不引入任何新功能（例如，3.1.2→3.1.3）

Helm chart版本的正确设置可能很有挑战性，因为它们不是那种典型的软件包（其中包含具有特定功能的应用程序）。根据经验，对chart版本的大多数更新应该是对次要版本的增量更新。

例如，当你添加在模板中使用的新键-值对时，这可以被视为一个新的“特性”，因为配置选项已经被扩展了。另一方面，如果要修改模板中某个值的使用方式，或者完全删除某个配置设置，则可以视为一个突破性的更改，应该增加主版本号。最后，如果你只是对一个Helm chart做了一个修复，用来解决它对于期望的输入没有正确地模板化，这将被认为是一个bug修复，应该增加补丁版本号。

对于chart的初始版本，请选择0.1.0或1.0.0。当主版本号为0（例如，0.1.0）时，从技术上讲，这表示对在次要版本升级和补丁版本升级之间进行突破性更改不做承诺。1.0.0和更高版本表示一定程度的稳定性和严格遵守SemVer 2。

字段：type

必需的

指定chart类型，可以是以下两种类型中的一种：

application

典型的、可安装的chart。

library

包含通用定义的不可安装的chart，意味着作为依赖项chart被包含。

此字段对于API 2是唯一的。在API 1中，所有chart都被认为是application chart。有关library chart的更多信息，参见第6章。

字段：description

一个简单的、一句话的chart描述。

字段：appVersion

chart表示的应用程序版本。

此字段应与正在部署的软件版本匹配，而不是与chart本身的版本匹配。例如，如果你正在创建一个新的内部chart来部署一个自定义配置的Nginx 1.18.0，appVersion字段将是1.18.0，而version字段将可能是0.1.0（初始版本）。

字段：home

chart或应用程序主页的绝对URL。

字段：icon

可以用作此chart图标的图像的绝对URL。

此字段通常由Artifact Hub等服务来对可供下载的chart显示适当的图像。

字段：sources

指向chart源代码的一个或多个绝对URL（如果可用）。

字段：keywords

chart表示的关键字或主题的列表。

这些关键字由服务（如Artifact Hub）使用，用于按类别对chart进行分组，或进一步增强搜索能力。

字段：maintainers

chart的维护人员的姓名/电子邮件/URL组合的列表。

字段：deprecated

此chart是否已被弃用。

此字段由Artifact Hub等服务用于确定何时删除chart列表。

字段：annotations

Helm未解释的chart的附加映射，可供其他应用程序检查。

注意：此字段没有以任何有意义的方式链接到Kubernetes注解。但是，你可以选择将它们定义为Kubernetes特定的注解，具体取决于你如何使用此字段。

字段：kubeVersion

SemVer约束，指定正确安装chart所需的最低Kubernetes版本。

某些chart可能使用了仅在某些特定Kubernetes版本上可用的Kubernetes资源类型和API组。如果操作人员试图安装与目标集群的kubeVersion不兼容的chart，则在配置Kubernetes资源之前将发生错误。

字段：dependencies

chart的依赖项列表。

当运行helm dependency update时，这里列出的chart依赖项将被协商并适当地放置到chart/子目录中。

dependencies块下的每个条目至少应包含一个name子字段，以及一个repository或alias子字段。repository应该是有效chart存储库的绝对URL（提供/index.yaml)。alias（别名）应该是字符“@”，后跟先前添加的chart存储库的名称（例如，@myrepo）。

有关如何使用chart依赖项的详细信息，参见6.1节。

Chart.lock文件

当chart在Chart.yaml的dependencies字段下列出依赖项时，每次运行命令helm dependency update时都会生成和更新一个名为Chart.lock的特殊文件。当chart包含Chart.lock文件时，操作人员可以运行helm dependency build来生成chart/目录，而无须重新协商依赖项。

下面是一个基于Chart.yaml中指定的依赖项生成的Chart.lock文件：

```
dependencies:
- name: redis
  repository: https://kubernetes-charts.storage.example.com/
  version: 10.5.7
- name: postgresql
  repository: https://charts.example.com/
  version: 8.6.4
digest: sha256:529608876e9f959460d0521eee3f3d7be67a298a4c9385049914f44bd75ac9a9
generated: "2020-07-17T11:10:34.023896-05:00"
```

动态字段（如conditions和tags）被剥离掉了，该文件只包含更新期间为每个依赖项解析的repository、name和version，以及digest（SHA-256）和generated时间戳。

注意，PostgreSQL依赖项的alias:"@myrepo"设置已被转换为repository:https://charts.example.com/。这意味着在更新依赖项之前，使用以下命令添加了一个chart存储库：

```
$ helm repo add myrepo https://charts.example.com/
```

API第1版（旧版）

chart API第1版是原始的API版本，也是Helm 2唯一认可的版本。Chart.yaml中的apiVersion字段最初是在Helm 3中引入的，Helm 2不认可。使用Helm 2，则假设所有chart都遵循API第1版。在Helm 3中，apiVersion是严格必需的。

使用API第1版的chart同时到Helm 2和Helm 3的支持，但可能无法支持某些将来只有Helm 3才能使用的特性。

chart.yaml文件

对于chart的Chart.yaml文件格式而言，使用API第1版与API第2版几乎没有差别，只有几个重要的区别。

下面是一个使用第1版API的chart的Chart.yaml文件：

```
apiVersion: v1 ❶
name: lemon
version: 1.2.3
description: When life gives you lemons, do the DevOps
appVersion: 2.0.0
home: https://example.com
icon: https://example.com/img/lemon.png
sources:
  - https://github.com/myorg/mychart
keywords:
  - fruit
  - citrus
maintainers:
  - name: Carly Jenkins
```

```
    email: carly@mail.cj.example.com
    url: https://cj.example.com
  - name: William James Spode
    email: william.j@mail.wjs.example.com
    url: https://wjs.example.com
deprecated: false
annotations:
  sour: 1
kubeVersion: ">=1.14.0"
tillerVersion: ">=2.12.0"
engine: gotpl
```

❶ 表示chart API第1版的字段。

第1版与第2版的差异

API第1版与第2版相比，有一些细微的区别：

· apiVersion字段设置为v1（在Helm 2中，此字段不是严格必需的）。

· 缺少type字段。API第1版中没有库chart的概念。

· 缺少dependencies字段。在API第1版中，chart依赖项是在一个名为requirements.yaml的专用文件中指定的（如本节后面所述）。

· 还有两个字段：tillerVersion和engine。

 在许多方面，这两个chart API版本基本上可以被认为是Helm 2 chart（v1）和Helm 3 chart（v2）。这一点尤其正确，因为chart API第2版是在Helm 3发布的同一时间引入的。

这些版本之所以没有被命名为v2和v3（表示Helm版本），是因为用于chart的API独立于用于Helm CLI的API。

例如，如果Helm 4发布了，那么chart API第2版可能仍然会被使用。同样地，如果chart API第2版后来由于一些原因被确定为不能胜任，那么可以在发布另一个主要Helm版本之前引入一个新的chart API第3版。

字段：tillerVision（旧版）

SemVer约束指定正确安装chart所需的Tiller版本。

Tiller是一个旧版的Helm服务器端组件，只在Helm 2中使用。当使用Helm 3时，此字段被完全忽略。

字段：engine（旧版）

要使用的模板引擎的名称，默认为gotpl。

requirements.yaml文件（旧版）

在API第1版中，还有一个名为requirements.yaml的文件用于指定chart的依赖项。此文件的格式与API第2版中定义的dependencie字段相同。

下面是一个独立的requirements.yaml文件：

```
dependencies:
  - name: redis
    version: ~10.5.7
    repository: https://kubernetes-charts.storage.example.com/
    condition: redis.enabled
  - name: postgresql
    version: 8.6.4
    repository: "@myrepo"
    tags:
      - database
      - backend
```

有关每个子字段的详细说明，参见API第2版下标题为“字段：dependencies”的小节。在API第2版中，此文件的内容直接在Chart.yaml中定义。

requirements.lock文件（旧版）

在API第1版中，chart依赖项lock文件的名称为requirements.lock。此文件的格式和用途与API第2版描述的Chart.lock文件相同，只是名称不同。有关更多信息，参见API第2版下标题为“Chart.lock文件”的小节。

附录B
chart存储库API

在第7章中，我们讨论了chart存储库。本附录简要介绍了chart存储库API，这是一个使Helm能够使用chart存储库的底层规范。

chart存储库API是轻量级的，因为它只有一个必需的HTTP端点：GET/index.yaml。

在99%的情况下，chart存储库还提供chart软件包tar压缩文件（.tgz）和任何相关的provenance文件（.prov）。但是，也可以将这些文件托管在单独的域中。

如第7章所述，index.yaml表示存储库索引，它包含存储库中所有可用chart版本的完整列表。这个文件的格式是特定于Helm的，目前只有一个API版本（1）。

index.yaml

实现chart存储库API时，你的服务必须提供一个相对于提供的存储库URL的HTTP GET/index.yaml路由。此请求的响应必须返回状态代码200 OK，并且响应的正文必须是如下所述的有效index.yaml。

 GET/index.yaml端点不一定要位于URL路径的根目录下。例如，如果提供的存储库URL位于https://example.com/charts，那么GET/index.yaml路由必须能在https://example.com/charts/index.yaml上访问。

index.yaml格式

以下是一个简单有效的使用单一chart版本（superapp-0.1.0）的index.yaml：

```
apiVersion: v1 ❶
entries: ❷
  superapp:
  - apiVersion: v2
    appVersion: 1.16.0
    created: "2020-04-28T10:12:22.507943-05:00" ❸
    description: A Helm chart for Kubernetes
    digest: 46f9ddeca12ec0bc257a702dac7d069af018aed2a87314d86b230454ac033672 ❹
    name: superapp
    type: application
    urls: ❺
    - superapp-0.1.0.tgz
    version: 0.1.0
generated: "2020-04-28T11:34:26.779758-05:00" ❻
```

❶ 存储库API版本（必须始终为v1）。

❷ 存储库中唯一chart名称到所有可用版本列表的映射。

❸ 使用helm package创建tar压缩文件的时间戳。

❹ tar压缩文件的SHA-256摘要。

❺ 可以下载chart的URL列表。这些URL可以是绝对路径，甚至可以托管在不同的域上。如果提供了相对路径，则认为它是相对于index.yaml的。通常每个chart版本只提供一个URL条目，但是列表中可以提供多个URL条目，如果上一个条目不可访问，Helm将尝试下载列表中的下一个条目。

❻ index.yaml文件生成时的时间戳，以RFC 3339格式生成。

除了created、digest和urls字段之外，每个chart版本上的所有字段（name、version等）都由chart API定义。更多信息参见附录A。

何时下载index.yaml

当Helm下载或重新下载存储库索引时，有5种值得注意的场景：

1.最初添加chart存储库时：

```
$ helm repo add myrepo https://charts.example.com
```

2.更新所有chart存储库时：

```
$ helm repo update
```

3.更新依赖项时（使用--skip-refresh标志来禁用）：

```
$ helm dependency update
```

4.从lock文件生成依赖项时（使用--skip-refresh标志禁用）：

```
$ helm dependency build
```

5.使用--dependency-update标志安装具有依赖项的本机chart时：

```
$ helm install myapp . --dependency-update
```

何时使用index.yaml的缓存版本

一旦index.yaml已下载，它就存储到本机缓存中，并在引用与存储库关联的唯一名称时使用（例如，“myrepo”）。

当Helm使用本地缓存的存储库索引时，有5种值得注意的场景：

1.从repo中拉取chart时：

```
helm pull myrepo/mychart
```

2.从repo安装chart时：

```
helm install myapp myrepo/mychart
```

3.从repo中基于chart升级版本时：

```
helm upgrade myapp myrepo/mychart
```

4.搜索要使用的chart时：

```
helm search repo myrepo/
```

5.使用--skip-refresh标志（并且依赖项包含alias子字段，如"@myrepo"）更新依赖项时：

```
helm dependency update --skip-refresh
```

.tgz文件

存储库中的.tgz文件表示单独的chart版本，打包为压缩的tar文件。

由于这些文件托管在存储库中，因此不需要URL路径。但是，当Helm请求它们时，必须能够下载它们。响应的状态码必须是200 OK，响应的正文应该是二进制形式的.tgz内容。

何时下载.tgz文件

当Helm下载chart软件包.tgz文件时，有3个值得注意的场景：

1.从repo中拉取chart时：

```
helm pull myrepo/mychart
```

2.从repo安装chart时：

```
helm install myapp myrepo/mychart
```

3.从repo中基于chart升级版本时：

```
helm upgrade myapp myrepo/mychart
```

.prov文件

存储库中的.prov文件表示用GNU Privacy Guard签名的chart版本签名文件。这些文件是可选的，用于验证的目的。

与.tgz文件不同，.prov文件具有独特的URL路径要求。它们必须可以在相关.tgz文件（添加.prov后缀）的路径上访问。例如，如果.tgz文件位于https://charts.example.com/superapp-0.1.0.tgz，则.prov文件必须位于https://charts.example.com/superapp-0.1.0.tgz.prov。

响应的状态码必须是200 OK，响应的正文应该是二进制形式的.prov内容。

何时下载.prov文件

当Helm下载chart签名.prov文件时，有3个值得注意的场景：

1.使用--verify标志从repo中拉取chart时：

```
helm pull myrepo/mychart --verify
```

2.使用--verify标志从repo安装chart时：

```
helm install myapp myrepo/mychart --verify
```

3.使用--verify标志从repo中基于chart升级版本时：

```
helm upgrade myapp myrepo/mychart --verify
```