

OpenShift 开发指南中文版 (alpha)

O'REILLY®

Compliments of
RED HAT
OPENSIFT



Deploying to OpenShift

A GUIDE FOR BUSY DEVELOPERS

Graham Dumpleton

OREILLY

Deploying to Openshift

深入了解 Open Shift, Red Hat 基于容器的软件部署和管理平台, 为企业提供安全的多租户环境。本实用指南详细描述了如何在 Kubernetes 上构建的 Openshift 使您能够自动化在集装箱环境中创建, 运输和运行应用程序的方式。

作者 Graham Dumbleton 提供了您需要的知识, 以便充分利用 Open shift 容器平台, 不仅可以部署云本机应用程序, 还可以部署更传统的有状态应用程序

开发人员和管理员将学习如何在 Openshift 中运行, 访问和管理容器, 包括如何协调规模

本书是 Openshift for Developers 的完美后续: 写给初学者的指南 (Oreilly)

- 创建一个项目并部署预先存在的应用程序容器映像
- 从源代码构建应用程序容器映像并进行部署
- 实现和扩展应用程序映像构建器
- 使用增量式和链式构建来加速构建时间
- 通过使用 webhook 将 Open Shift 链接到 Git 存储库来自动构建
- 作为项目资源添加配置和秘密到容器
- 在 Open Shift 群集外部使应用程序可见
- 管理 Openshift 容器内的永久存储
- 监视应用程序运行状况并管理应用程序生命周

Graham Dumbleton 是红帽开放转移的开发者倡导者。他是 Python 软件开发人员社区的活跃成员, 也是 `mod_wsgi` 的作者, `mod_wsgi` 是用于与 Apache HTTPD Web 服务器一起托管 Python Web 应用程序的流行模块。格雷厄姆对 Docker 和平台即服务 (Paas) 技术也有着浓厚的兴趣

部署到 OpenShift—高晓雪译(snow@polaristech.io)

Graham Dumpleton 作

版权所有©2018 Red Hat, Inc. 保留所有权利。

美利坚合众国印刷。

由 O'Reilly Media, Inc., 1005 Gravenstein Highway North, Sebastopol, CA 95472 出版。

O'Reilly 图书可以用于教育, 商业或促销用途。在线版

(<http://oreilly.com/safari>) 也可以应用于上述用途。欲了解更多信息, 请联系我们的公司/机构。销售部门: [800-998-9938](tel:800-998-9938) 或 corporate@oreilly.com。

编辑: 弗吉尼亚威尔逊和尼基麦克唐纳

制作编辑: Melanie Yarbrough

Copyeditor: 德怀特拉姆齐

校对员: 雷切尔头

索引: Judy McConville

室内设计师: David Futato

封面设计师: 凯伦蒙哥马利

插画: 丽贝卡德马雷斯

2018 年 4 月: 第一版

第一版的修订历史

2018-03-05: 第一版

这项工作是 O'Reilly 和红帽合作的一部分。请参阅我们的[编辑独立声明](#)。

O'Reilly 标识是 O'Reilly Media, Inc. 的注册商标。部署到 OpenShift 封面图片和相关商业外观是 O'Reilly Media, Inc. 的商标。

虽然出版商和作者已经使用诚信努力确保信息和

包含在这项工作中的指示是准确的, 出版商和作者不承担任何责任

对于错误或遗漏, 包括但不限于由于使用而导致的损害

或依赖这项工作。使用本工作中包含的信息和说明是您自己的

风险。如果此工作包含或描述的任何代码示例或其他技术受制于开源

许可证或他人的知识产权, 您有责任确保您的使用

符合这些许可和/或权利。

目录

前言	7
第 1 章.OpenShift 容器平台	9
1.1 容器的作用	9
1.2 组织规模	10
1.3 容器即服务	11
1.4 平台即服务	11
1.5 部署您的应用程序	12
第 2 章.运行 OpenShift 群集	12
2.2 在线使用 OpenShift	13
2.3 安装 OpenShift Origin	13
2.4 启动使用 Minishift	13
2.5 运行 oc 群集	15
2.6 总结	16
第 3 章.访问 OpenShift 群集	16
3.1 使用 Web 控制台	16
3.2 使用命令行	17
3.3 使用 OpenShift REST API	19
3.4 总结	20
第 4 章.应用程序添加到项目	20
4.1 一个项目的作用	20
4.2 创建一个项目	21
4.3 添加合作者	22
4.4 部署应用程序	23
4.5 从目录中部署	23
4.6 部署映像	25
4.7 部署一组资源	25
4.8 总结	26
第 5 章 从映像部署应用程序	26
5.1 部署您的第一个人镜像	27
5.2 扩展应用程序	29
5.3 运行时间配置	29
5.4 删除应用程序	30
5.5 使用 Web 控制台进行部署	30
5.6 导入镜像	31
5.7 推送到注册表	31
5.8 图片和安全	32
5.9 总结	33
第 6 章从源代码构建和部署	33
6.1 源代码构建策略	33
6.2 从源部署	34
6.3 创建一个单独的版本	35
6.4 触发新构建	36

6.5 从本地来源构建	36
6.6 二进制输入构建	37
6.7 测试容器镜像	38
6.8 构建和运行时间配置	38
6.9 总结	39
第 7 章. 从 Dockerfile 构建一个镜像	39
7.1 Docker 构建策略	39
7.2 安全和 Docker 构建	39
7.3 创建 Build	40
7.4 部署镜像	40
7.5 构建和运行时间配置	41
7.6 使用内联 Dockerfile	42
7.7 总结	43
第 8 章. 了解源到镜像构建器	43
8.1 Source-to-Image 项目	43
8.2 构建应用程序镜像	43
8.3 汇编源代码	44
8.4 创建 S2I 生成器镜像	45
8.5 构建 S2I 生成器镜像	46
8.6 在 OpenShift 中使用 S2I Builder	46
8.7 将 S2I 生成器添加到目录	47
8.8 总结	48
第 9 章. 自定义 Source-to-Image 构建	48
9.1 使用环境变量	48
9.2 覆盖 Builder 脚本	49
9.3 只读代码存储库	50
9.4 覆盖运行时镜像	50
9.5 更新镜像元数据	51
9.6 总结	52
第 10 章. 使用增量和链接构建	52
10.1 使用缓存快速构建	52
10.2 使用增量构建	53
10.3 从构建中保存开发	53
10.4 恢复构建开发	54
10.5 启用增量构建	54
10.6 使用链式构建	55
10.7 总结	56
第 11 章. Webhooks 和构建自动化	56
11.1 使用 Hosted Git Repository	56
11.2 访问私有 Git 存储库	57
11.3 添加 Webhook 存储库	58
11.4 定制构建触发器	59
11.5 总结	59
第 12 章 配置和隐私	59

12.1 传递环境变量	60
12.2 使用配置文件	60
12.3 处理保密信息	62
12.4 删除配置和隐私	64
12.5 总结	64
第 13 章.服务，网络和路由	64
13.1 容器和集群	65
13.2 服务和端点	65
13.3 连接项目	66
13.4 创建外部路由	67
13.5 使用安全连接	68
13.6 内部和外部端口	68
13.7 公开非 HTTP 服务	69
13.8 本地端口转发	69
13.9 总结	70
第 14 章.使用持久存储	70
14.1 持久存储的类型	70
14.2 声明持久卷	71
14.3 卸载持久卷	72
14.4 重新使用持久卷声明	72
14.5 在应用程序间共享	72
14.6 在容器之间共享	73
14.7 删除持久卷	73
14.8 将数据复制到卷	73
14.9 总结	74
第 15 章.资源配额和限制	74
15.1 什么是配额管理	74
15.2 配额与限制范围	75
15.3 请求与限制	76
15.4 资源需求	76
15.5 重写构建资源	77
15.6 总结	77
第 16 章.监控应用程序健康	78
16.1 准备探测器的作用	78
16.2 活力探测器的作用	78
16.3 使用 HTTP 请求	79
16.4 使用容器命令	79
16.5 使用套接字连接	80
16.6 探测频率和超时	80
16.7 总结	81
第 17 章.应用程序生命周期管理	81
17.1 部署策略	81
17.2 滚动部署	82
17.3 重新创建部署	82

17.4 自定义部署	83
17.5 容器运行时钩子	83
17.6 初始容器	84
17.7 总结	85
第 18 章.记录, 监视和调试	85
18.1 查看构建日志	86
18.2 查看应用程序日志	86
18.3 监视资源对象	87
18.4 监视系统事件	88
18.5 查看容器度量标准	88
18.6 运行交互式 Shell	89
18.7 调试启动失败	89
18.8 总结	90

前言

OpenShift 为 Web 应用程序的部署实施了多语言平台和服务。它将容器与安全增强型 Linux (SELinux) 环境结合使用, 以实现适合企业的安全多租户环境。您可以在自己的基础设施或公共云中部署 OpenShift, 也可以使用 Red Hat 的基于云的托管服务 OpenShift Online。

最新版本的 OpenShift 使用来自 [云本机计算基金会 \(CNCF\)](#) 行业标准的 Kubernetes 平台, 用于管理和运行容器内的应用程序。通过遵守 [Open Container Initiative \(OCI\)](#) 中的映像和运行时规范来确保运行任何应用程序映像的能力。

OpenShift 为您提供了直接轻松部署 Web 应用程序代码的能力, 使用预定义的图像构建器库, 或者您可以携带自己的容器镜像。通过在 OpenShift 中支持持久性卷等功能, 您可以不仅限于运行无状态的 12 因子或云本地应用程序。使用 OpenShift, 您还可以部署数据库和许多传统应用程序, 否则这些应用程序无法在传统的平台即服务 (PaaS) 产品上运行。

OpenShift 是一个完整的容器应用程序平台。这是对现有应用程序可以使用的传统 PaaS 的现代化应用, 但也提供了满足未来需求的功能和灵活性。

谁适用于这本书

本书面向已经开发 OpenShift 的开发人员, 还有决定使用它并且需要对用于部署应用程序的 OpenShift 的核心功能有更深入的了解的人。管理 OpenShift 集群的管理员也需要为使用该平台的开发人员提供帮助。

本书是 Red Hat 有关 OpenShift 最新版本的系列丛书中的第三本。以前的系列丛书是:

- [OpenShift for Developers: 初学者指南](#)
- [使用 OpenShift 的 DevOps: 轻松实现云部署](#)

为什么写这本书

本书 OpenShift 开发指南, 与 Grant Shipley 合作编写, 旨在通过跳过许多细节尽快启动。但是, 这些细节非常重要, 当您想要充分利用 OpenShift 时。在本书中, 我想填补一些空缺, 以便让您更深入地了解 OpenShift, 以及它如何使您更轻松地将应用程序部署到云中。

我为这本书精选的主题很多都是关于公共社区论坛上 (如 Stack Overflow 和 Google

Groups) 以及在会议上, 在 Red Hat 担任 OpenShift 开发人员的角色时, 遇到的必须解决的问题

写这本书目的是将作为一个快速参考指南, 你可以继续, 以便在常用的模式中刷新你的记忆, 或学习更多关于您继续使用 OpenShift 的其他主题。

在线资源

与许多新技术一样, OpenShift 在适应其应用的广泛用例的情况下仍在不断发展。 当你阅读本书时, 某些信息可能无法提供最新的图片。这就是为什么我们鼓励您查看在线资源以了解 OpenShift 的最新细节以及如何使用它。

[OpenShift 文档](#)是从 OpenShift Online 到 Red Hat 的企业产品寻找有关 OpenShift 的信息时的首选。

要在虚拟机中的本地计算机上本地运行 OpenShift Origin, 您可以使用 [Minishift](#)。

如果您对 OpenShift 的源代码感兴趣, 可以通过 [OpenShift 起源项目](#)获取。

OpenShift Origin 是用于创建红帽 OpenShift 产品系列上游开源项目,。OpenShift Origin 将始终包含所有最新特性, 包括实验性功能, 并由 OpenShift 社区提供支持。我们热烈邀请您克隆 OpenShift Origin 项目代码, 发送您的 **contributions**, 或者打开 issue 报告您发现的任何问题。

OpenShift 产品版本是作为 OpenShift 的常规快照创建的起源项目。产品版本并不总是启用最新功能, 但如果您有红帽订阅, 则产品版本包括来自红帽的支持。

如果您想尝试 OpenShift 企业产品, 有许多选项可用。

首先是注册[红帽开发者计划](#)。这是一个免费的程序, 允许您在自己的计算机上访问红帽产品的个人使用版本。通过该计划提供的产品之一是[红帽容器开发工具包](#)。 这包括 OpenShift 的一个版本, 您可以在自己的计算机上的虚拟机中运行, 但基于 [OpenShift Container Platform](#) 产品而不是 OpenShift Origin。

尝试 OpenShift 容器平台的第二种方法是通过[免费试用版](#)领先的云服务提供商。这将为您设置一个 OpenShift 环境, 该环境跨所选云提供商的多节点群集运行。

就像你所看到的, 只是想让你的网站在那里被使用, 而不需要建立和运行你自己的 OpenShift 集群? 查看 Red Hat 的基于公共云的托管服务 OpenShift Online。

想要了解 OpenShift 社区中的其他人如何使用 OpenShift 或希望分享你自己的经验? 你可以加入 [OpenShift Commons](#)。

除了听取 OpenShift 社区成员的意见外, 还请检查 [OpenShift 博客](#), 来自 Red Hat OpenShift 团队成员的常规文章发布。

如果您对 OpenShift 的开发有疑问, 您可以通过 [OpenShift 邮件列表](#)或#openshiftdev 访问 OpenShift 开发团队频道在 IRC 的 Freenode 网络上。 社区对 OpenShift Online 的支持可以在 [Google 网上论坛](#)或 [Stack Overflow](#)中找到。

O'Reilly Safari

Safari (以前称为 Safari Books Online) 是一个针对企业, 政府, 教育工作者和个人的基于会员的培训和参考平台。

会员可以访问 250 多家发布商的数千本图书, 培训视频, 学习路径, 互动教程和策划播放列表, 其中包括 O'Reilly 媒体, 哈佛商业评论, 普伦蒂斯霍尔专业, 艾迪生 - 韦斯利专业, Microsoft Press, Sams, Que, Peachpit Press, Adobe, Focal Press, Cisco Press,

John Wiley&Sons, Syngress, Morgan Kaufmann, IBM 红皮书, Packt, Adobe Press, FT Press, Apress, Manning, New Riders, McGraw-Hill, Jones&Bartlett, 和课程技术等。

欲了解更多信息, 请访问 <http://oreilly.com/safari><http://oreilly.com/safari>。

如何联系我们

请向出版商发表有关本书的评论和问题:

O'Reilly 媒体公司
1005 Gravenstein 公路北部
Sebastopol, CA 95472
800-998-9938 (在美国或加拿大)
707-829-0515 (国际或本地)
707-829-0104 (传真)

要评论或询问关于本书的技术问题, 请发送电子邮件至 [bookquestions @ oreilly.com](mailto:bookquestions@oreilly.com)。

有关我们的书籍, 课程, 会议和新闻的更多信息, 请参阅我们的网站在 <http://www.oreilly.com>。

在 Facebook 上找到我们: <http://facebook.com/oreilly>

在 Twitter 上关注我们: <http://twitter.com/oreillymedia>

在 YouTube 上观看我们的视频: <http://www.youtube.com/oreillymedia>

第1章. OpenShift 容器平台

OpenShift 平台于 2011 年 5 月推出。源代码通过一个开源项目提供, 任何人都可以下载并使用它。红帽还提供了用于企业部署的受支持版本的 OpenShift, 以及名为 OpenShift Online 的托管服务。

OpenShift 一直在容器之上实现, 但技术是总是在不断发展。2013 年 6 月, 一次重大改写开始重新实现 OpenShift 容器领域最新技术的领先者。

基于 [Kubernetes](#) 和 Docker 容器运行时的 OpenShift Origin 1.0 版已于 2015 年 6 月发布。

在撰写本书时, OpenShift 3.6 是最新版本, 3.7 版即将发布, 并且每季度推出新版本。

OpenShift 究竟是什么?

简而言之, 它是一个平台, 可帮助您在多个主机上开发并部署应用程序。这些可以是面向公众的 Web 应用程序, 也可以是后端应用程序, 包括微服务或数据库。应用程序可以用您选择的任何编程语言来实现。唯一的要求是应用程序可以在容器中运行。

OpenShift 可以运行在任何可以运行红帽企业 Linux (RHEL), CentOS, 或 Fedora。这可以在公共或私有云基础架构上, 直接在物理上硬件或使用虚拟机。

在第一章中, 您将学习更多关于 OpenShift 使用的技术, 以及它适合云计算生态系统的地方。

1.1 容器的作用

美国国家标准与技术研究院 (NIST) 将云计算定义的一部分定义为提供 [云计算](#) 服务的三种标准服务模型:

1. 软件即服务 (SaaS)

提供给消费者的功能是使用在云基础架构上运行的提供商的应用程序。应用程序可以通过瘦客户机接口（例如网络浏览器（例如，基于 web 的电子邮件））或程序接口从各种客户机设备访问。消费者不管理或控制底层云基础架构，包括网络，服务器，操作系统，存储或甚至是单个应用程序功能，但有限的用户特定应用程序配置设置可能例外。

2. 平台即服务 (PaaS)

提供给消费者的功能是部署到云基础设施上，使用由供应商支持的编程语言，库，服务和工具创建的消费者创建或获取的应用程序。消费者不管理或控制包括网络，服务器，操作系统或存储在内的底层云基础架构，但可以控制应用程序托管环境的已部署应用程序和可能的配置设置。

3. 基础设施即服务 (IaaS)

提供给消费者的能力是提供消费者能够部署和运行任意软件（包括操作系统和应用程序）的处理，存储，网络和其他基本计算资源。消费者不管理或控制底层云基础架构，但可以控制操作系统，存储和部署的应用程序；并且可能限制对选定网络组件（例如，主机防火墙）的控制。

在这些云服务计算模型的传统定义下，OpenShift 将被归类为 PaaS。在 PaaS 和 SaaS 模式中，集装箱化往往是用于将应用程序彼此分离并从不同的用户分离。

容器作为一项技术有着悠久的历史，先行者是 FreeBSD jail 和 Solaris Zones。在 Linux 中，对容器的支持围绕着 Linux 容器项目 (LXC)。这为 Linux 内核功能（如 cgroups 和命名空间）带来了用户空间工具，并增加了来自 Seccomp 和 SELinux 的安全性。LXC 工具可以在 Linux 中使用容器，但在容器中设置和运行应用程序仍然是一个繁琐的过程。

2013 年，一家名为 dotCloud 的公司，一家 PaaS 提供商，在 PyCon 美国闪电发布了一款名为 Docker 的工具。该工具是 dotCloud 用于帮助在容器中运行应用程序的专有技术的延伸；它提供了一个包装器，使用 LXC 在容器中启动应用程序变得更加容易。

Docker 工具很快就被开发人员所接受，因为它解决了两个关键问题。第一个是图像通用打包格式的定义，其中包含一个应用程序及其所需的所有依赖项，包括操作系统库和程序。第二个是制作这个图像的工具。

这些功能一起使得创建应用程序映像成为可能，这些应用程序映像可以在不同的系统之间轻松移动，然后在容器中运行，并且更有信心可以开箱即用。

围绕 Docker 工具的技术与 dotCloud 公司单独拆分出来，并创建了一家新公司 Docker Inc.，以管理该技术的开发。由于对该技术的兴趣日益浓厚，开放容器倡议组织 (OCI) 后来成立，旨在为围绕容器格式和运行时创建开放行业标准的明确目的提供开放式管理结构。这个过程是从 Docker 工具派生的规范中播种的。OCI 目前担任两种规格的管家：运行时规范（运行时规格）和图像规格（图像规格）。

1.2 组织规模

Docker 工具使开发人员可以更轻松地构建应用程序映像，并在单个主机上的容器中运行单个应用程序。但是，扩大应用程序以使多个实例在同一主机上或跨多个主机运行时，需要额外的软件组件来帮助协调应用程序的部署和运行，以及路由器负载平衡流量到每个实例应用。

在 Docker 采用的初始阶段，业务流程和路由层不存在开箱即用的解决方案，从而导致

用户手工制作自主开发的解决方案。

在 2014 年年中，Google 宣布了 Kubernetes 项目，这是一个用于自动化集装箱应用程序的部署，扩展和管理的开源系统。这提供了尝试处理大规模运行容器时所需的缺失组件之一。

当时，红帽已经进入了一个项目，围绕 Docker 重新实现了 OpenShift，但一直在实施自己的编排层。随着 Kubernetes 的宣布，红帽决定放弃自己的努力，采用 Kubernetes 并成为该项目的主要贡献者。

Kubernetes 随后于 2015 年 7 月以 1.0 发布，并将该项目捐赠给云计算本机计算基金会（CNCF）。它已经成为集装箱业务的事实标准。

1.3 容器即服务

Kubernetes 不适合任何现有的云计算服务模型分类。这导致了新的名称“集装箱即服务”（CaaS）的诞生。此服务模型可以看作与 IaaS 类似，不同之处在于不提供虚拟机，而是提供一个容器。

图 1-1 显示了 CaaS 在其他服务模型中的适用位置。

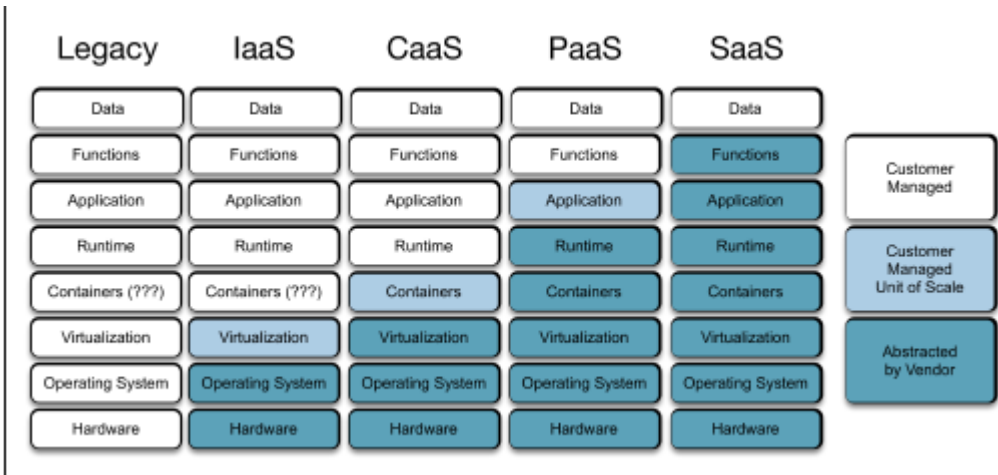


图 1-1 云服务

要在 CaaS 中运行应用程序，您需要提供您构建的应用程序映像，其中包含您的应用程序以及您需要的任何操作系统库和程序。

尽管应用程序映像包含这些操作系统库和程序的副本，但它只是您运行的应用程序进程。

1.4 平台即服务

使用 Kubernetes 的 CaaS 功能，OpenShift 能够从存储在任何图像注册表中的容器映像部署应用程序。Kubernetes 本身不提供任何支持来构建容器映像。您需要运行构建工具以在单独的系统中创建应用程序映像，并将其推送到映像注册表。从中可以部署它。这是因为 CaaS 专注于运行容器，缺乏 PaaS 的功能，在这里您可以提供源代码，平台将解决如何在容器中运行这些代码。

为了提供 PaaS 能够获取源代码并进行部署的能力，OpenShift 增加了在 Kubernetes 之上执行构建的自动化。OpenShift 支持两种从源代码构建的主要策略。

按照传统 PaaS 的风格，OpenShift 可以获取应用程序源代码，并使用您正在使用的编程

语言的构建器创建应用程序映像。您作为开发人员不需要提供有关如何创建容器图像的说明;该平台将为您做到这一点。

另外, OpenShift 还可以接受包含一组指令的源代码, 以在 Dockerfile 中创建应用程序映像。如果您使用 Docker 自己构建容器图像, 则需要执行此操作, 但 OpenShift 会在 OpenShift 平台内为您执行此操作。

在这两种情况下, OpenShift 都会将应用程序映像缓存在它提供的映像注册表中。应用程序映像将从此内部映像注册表部署。

使用 OpenShift 从 Dockerfile 构建应用程序图像意味着您不需要在自己的系统上安装 Docker 工具, 也不需要单独使用单独的图像注册表来保存图像。

对于这两种构建类型, OpenShift 都可以从托管的 Git 存储库中获取构建的源代码。如果您使用的是 GitHub, GitLab 或 Bitbucket 等服务, 则只要将更改后的代码推送回托管的 Git 存储库, 就可以配置 Git 存储库托管服务以通知 OpenShift。此通知可以触发您的应用程序的新构建和部署。

这种自动化意味着, 一旦您在 OpenShift 中设置了您的应用程序, 您就不需要在继续进行应用程序开发时直接与 OpenShift 进行交互。只要将更改推回到托管的 Git 存储库, OpenShift 就会知道并可以自动构建和重新部署应用程序。

1.5 部署您的应用程序

通过在 Kubernetes 上游构建, 为构建和部署添加自己的自动化, OpenShift 既可以作为 CaaS 又可以作为 PaaS 运行。换句话说, OpenShift 实现了一个通用的容器平台。您可以部署自己的定制应用程序, 也可以导入第三方应用程序(如数据库, 消息传递系统或其他业务套件)以支持组织的业务流程。

在本书中, 您将了解可以将应用程序部署到 OpenShift 的不同方式。这将包括使用 OpenShift 的 CaaS 功能从预构建的应用程序映像进行部署, 并以 PaaS 的方式从源代码构建。

您还将学习如何将您的应用程序集成到 OpenShift 平台中, 如何通过 OpenShift 进行配置, 如何挂载持久卷, 如何将其公开以便用户可以访问它以及如何监视和调试应用程序。

要与 OpenShift 交互, 您可以使用 Web 控制台或命令行客户端。本书将重点介绍如何使用命令行客户端。

整本书将使用的主要应用程序示例是一个使用 Django Web 框架实现的 Python 应用程序。这是 [OpenShift Interactive Learning Portal](#) 上许多教程中使用的示例应用程序。除了阅读本书外, 您还可以使用这些教程进一步调查我们涵盖的许多主题。

示例应用程序的源代码可以在 [GitHub](#) 上找到。

如果您希望自己完成本书中的示例, 则可以使用 OpenShift Interactive Learning Portal 上的操场环境。游乐场不遵循设定的教程, 并且您可以随意尝试任何您想要的。

您也可以自己安装 OpenShift, 或者使用托管的 OpenShift 服务。下一章将讨论运行 OpenShift 的选项, 本书的其余部分将向您展示如何使用它, 以便您可以学习部署自己的应用程序。

第2章. 运行 OpenShift 集群

开始使用 OpenShift 的最简单方法是访问托管版本, 例如 Red Hat 提供的 OpenShift Online。

要自己安装并运行 OpenShift，可以从上游的 OpenShift Origin 项目下载源代码。
要在您自己的计算机上快速运行 OpenShift，您可以启动预置版本的 OpenShift。
在本章中，您将了解这些不同的选项，包括有关如何使用 Minishift 和 oc 群集在本地运行 OpenShift 的更多详细信息。

2.2 在线使用 OpenShift

[OpenShift Online](#) 是 Red Hat 公开托管的 OpenShift 服务。如果你不想安装和管理你自己的 OpenShift 集群，这是你的选择。

OpenShift Online 提供免费的入门级，用于实验，测试或开发。当您准备将应用程序转移到生产环境并使其他人可以使用时，或者您需要除免费层提供的资源以外的其他资源时，可以升级到付费层。

超出 OpenShift Online，但仍然不想自己安装和管理 OpenShift？红帽还提供 [OpenShift Dedicated](#)。此服务与 OpenShift Online 类似，但 OpenShift 群集保留供您使用，并且不由组织外的其他用户共享。

2.3 安装 OpenShift Origin

如果您更喜欢自己安装并运行 OpenShift，则可以安装 [OpenShift Origin](#)。推荐的 OpenShift Origin 安装方法使用一组 Ansible playbooks。提供有关您的环境和机器节点的详细信息，运行 Ansible，并为您设置群集。

从头开始配置和安装 OpenShift 集群并不是本书将要讨论的内容。您可以在 [OpenShift Origin 网站的高级安装文档](#) 中找到说明。

OpenShift Origin 是由社区支持的 OpenShift 发行版。如果您希望运行自己的 OpenShift 集群，但有权访问专业支持服务，则可以从 Red Hat 获得 [OpenShift Container Platform](#) 的订阅。

2.4 启动使用 Minishift

为了在本地快速启动 OpenShift 集群，使用 OpenShift Origin 构建集群的另一种方法是在虚拟机（VM）内启动预先构建的系统。

要在 VM 内启动 OpenShift，可以使用 [Minishift](#)。这本身并不是一个 OpenShift 发行版，而是一个可以运行的工具，用于创建包含容器服务的最小 VM。Minishift 然后使用 OpenShift 客户机 oc 通过下载并启动包含 OpenShift 的预格式化容器映像来启动 OpenShift 群集。

运行 Minishift 需要管理程序运行包含 OpenShift 的虚拟机。根据您的主机操作系统，您可以选择以下虚拟机管理程序：

- macOS: xhyve（默认），VirtualBox
- GNU / Linux: KVM（默认），VirtualBox
- Windows: Hyper-V（默认），VirtualBox

要下载 Minishift 最新版本并查看任何发行说明，请访问 [Minishift 发布页面](#)。如果使用 macOS，您也可以使用 [Homebrew](#) 安装 Minishift。

在使用 Minishift 之前，请确保您查看[安装说明](#)，了解您的系统必须满足的任何先决条件。

安装 Minishift 程序后，您可以开始安装并通过运行 `minishift start` 命令来启动 OpenShift 集群：

```
$ minishift start
Starting local OpenShift cluster using 'kvm' hypervisor...
...
OpenShift server started.
The server is accessible via web console at:
https://192.168.99.128:8443
You are logged in as:
User: developer
Password: developer
To login as administrator:
oc login -u system:admin
```

默认情况下，Minishift 会尝试使用操作系统本地的虚拟驱动程序。要使用不同的驱动程序，请在运行此命令时设置 `--vm-driver` 标志。例如，要使用 VirtualBox，请运行 `minishift start --vm-driver = virtualbox`。有关可用选项的更多信息，请运行 `minishift start --help`。

当 Minishift 首次运行时，它会自动下载并安装特定于您的操作系统的 OpenShift oc 客户端二进制文件。这将安装在主目录中的缓存目录中，并以正在运行的 OpenShift 版本为关键字。运行命令 `minishift oc-env` 输出关于如何设置 shell 环境的指令，以便它可以找到 oc 程序：

```
$ minishift oc-env
export PATH="/Users/graham/.minishift/cache/oc/v1.5.0:$PATH"
# Run this command to configure your shell:
# eval $(minishift oc-env)
```

要访问 OpenShift 的 Web 控制台，请运行命令 `minishift console`。这将在 Web 控制台中为您打开一个浏览器客户端。

要确定 OpenShift 群集的 URL，以便从命令行登录或从浏览器访问 Web 控制台时使用，请运行命令 `minishift console --url`。您应该与 Minishift 创建的 OpenShift 群集一起使用的登录凭证是：

- 用户名：developer
- 密码：developer

要关闭 OpenShift 集群，请运行命令 `minishift stop`。您可以使用 `minishift start` 重新启动它。您的所有工作将在重新启动时恢复。要删除 OpenShift 集群，请运行命令 `minishift delete`。

当 Minishift 创建集群时，它将使用 CPU 的数量，可用内存以及 VM 磁盘的大小的默认设置。建议您使用能够更好地匹配可用资源或需求的值来覆盖这些值。这可以通过使用选项在第一次创建 OpenShift 群集时使用 `minishift start` 来完成。您可以使用 `minishift config` 命令覆盖这些资源的默认值以及使用的 VM 驱动程序。

Minishift 使用 OpenShift Origin。对于使用 Red Hat 的 OpenShift Container Platform 的

Minishift 版本，请参阅[红帽容器开发工具包](#)。

2.5 运行 oc 群集

Minishift 提供的主要功能是创建虚拟机。为了在该 VM 中设置和启动 OpenShift 集群，Minishift 将控制委派给命令 `oc` 集群。

`oc` 程序是 OpenShift 的标准命令行客户端。如果您的计算机上已经有本地容器服务运行，则可以使用已有的容器服务实例运行它，而不是使用 Minishift 并在 VM 内部运行 OpenShift。

您可以从 OpenShift Origin 的[发布页面](#)下载包含适用于您的操作系统的 `oc` 程序的存档文件。

在运行 `oc` 群集之前，请确保检查[安装说明](#)，因为您必须执行许多必备步骤来配置本地系统。

准备就绪后，要启动 OpenShift，请运行以下命令：

```
$ oc cluster up
-- Checking OpenShift client ... OK
-- Checking Docker client ... OK
-- Checking Docker version ... OK
-- Checking for existing OpenShift container ... OK
-- Checking for openshift/origin:v1.5.0 image ...
Pulling image openshift/origin:v1.5.0
....
-- Server Information ...
OpenShift server started.
The server is accessible via web console at:
https://127.0.0.1:8443
You are logged in as:
User: developer
Password: developer
To login as administrator:
oc login -u system:admin
```

要查找 OpenShift Web 控制台的 URL，您可以运行 `oc whoami --showserver`。要关闭 OpenShift 群集，请运行 `oc` 群集。

`oc cluster up` 命令旨在用于本地开发和测试，而不是用于生产用途。

默认情况下，当您运行 `oc` 集群时，您在 OpenShift 集群中执行的任何操作都不会持久化。也就是说，当你运行 `oc` 集群时，你将失去所有的工作。

为了保存您的工作，您必须为 `oc` 集群提供其他命令行选项。关机后重新启动时必须提供相同的选项：

```
$ oc cluster up --use-existing-config \
--host-config-dir $HOME/.oc-cluster-up/config \
--host-data-dir $HOME/.oc-cluster-up/data
```

运行 `oc cluster up --help` 找出可用的选项，并查看安装说明以获取有关其使用的更多信息。

2.6 总结

您可以通过多种方式安装 OpenShift - 直接连接到物理机器，虚拟机或私有云或公共云。OpenShift 通常会安装在一组机器上，以提供大规模运行应用程序的能力。

即使您正在使用独立的托管 OpenShift 环境，作为开发人员，在您自己的本地计算机上安装 Minishift 仍然是一个好主意。Minishift 的本地安装可用于试验 OpenShift 或尝试更新的版本。您也可以将 Minishift 集成到您的开发过程中，在将代码迁移到应用程序映像之前在本地处理您的应用程序到您的生产环境中。

如果您只想尝试 OpenShift，则可以在 OpenShift Interactive Learning Portal 上创建一个操场。这将为您提供一个可以通过浏览器使用的临时环境，而无需在本地计算机上安装任何东西。

第3章. 访问 OpenShift 群集

此时，您应该对 OpenShift 平台有一个基本的了解，以及它可以用于什么。要将您自己的应用程序部署到 OpenShift 群集，您可以使用 OpenShift Web 控制台或 `oc` 命令行客户端。然而，在你做任何事情之前，你首先需要登录到 OpenShift 集群。由于 OpenShift 是一个多租户环境，您将使用自己的帐户。

如果您已在自己的计算机上安装了 Minishift 或运行 `oc` 群集，则会为您预先配置一个帐户。该帐户的用户名将是开发人员。Minishift 和 `oc` 集群将接受密码的任何值。如果您使用其他用户名登录，它将自动配置一个新帐户。

如果您使用的是托管 OpenShift 环境（例如 OpenShift Online），则登录凭证将是您注册该服务的凭证，或者您可能需要使用第三方身份提供程序登录。

本章将介绍如何通过 Web 控制台和 `oc` 命令行客户端登录到 OpenShift。它还将说明如何为活动登录会话检索访问令牌，该访问令牌可用于在使用 REST API 时访问 OpenShift 群集。

3.1 使用 Web 控制台

访问 OpenShift 并与之交互的最简单方法是通过 Web 控制台。Web 控制台的 URL 将由作为 OpenShift 集群的公用 URL 指定的内容决定。一旦控制台被访问，您如何登录将取决于配置的身份提供者。

在简单情况下，Web 控制台登录页面会询问您的用户名和密码，如图 3-1 所示。

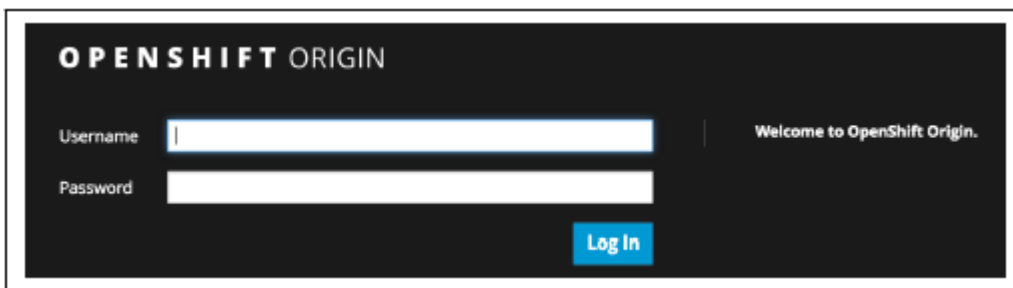


图 3-1 Web 控制台登录

如果使用外部 OpenID Connect 或 OAuth 身份验证服务提供程序，则需要通过外部服务登录（图 3-2）。

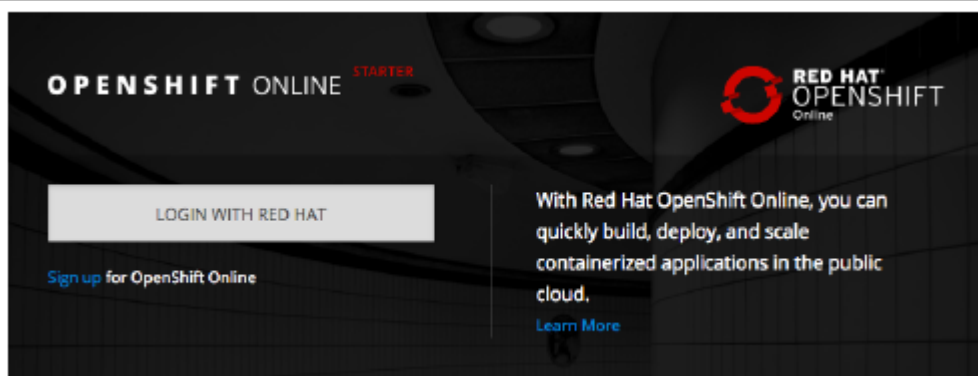


图 3-2 通过 Red Hat 进行 Web 控制台登录

对于新用户，一旦登录，您应该看到“欢迎使用 OpenShift”屏幕并且可以选择创建新项目（图 3-3）。

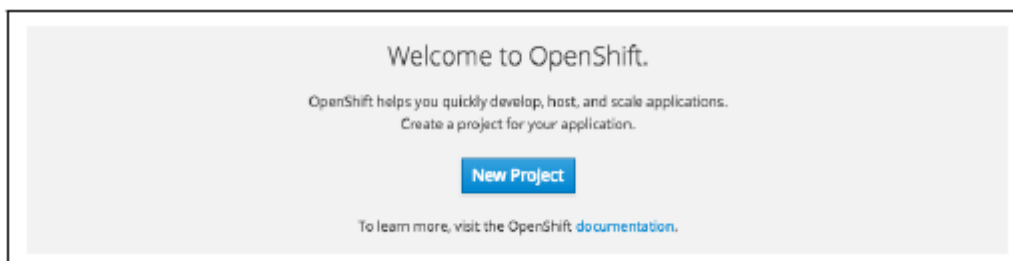


图 3-3 Web 控制台欢迎

3.2 使用命令行

Web 控制台提供了一种快速与 OpenShift 进行交互并查看已部署应用程序状态的便捷方法。但并非所有您可能想要做的事情，特别是集群或项目管理员角色都可以通过 Web 控制台完成。因此，您还需要熟悉使用 OpenShift 命令行工具 oc。

始终使用与您使用的 OpenShift 环境版本相匹配的 oc 命令行工具版本。如果使用旧版本的 oc 命令行客户端，则无法访问较新版本的 OpenShift 的所有功能。

很显然，这种情况下，功能依赖于 oc 命令行工具中实现的新命令类型或命令行选项。在使用较旧的客户端时，也可能会限制您查询由 OpenShift 实施的新类型资源对象的能力。

如果您还没有 oc 命令行工具，则可以按照图 3-4 中的步骤从 Web 控制台下载与您正在使用的 OpenShift 集群相对应的版本。同一页面还为您提供运行登录所需的命令。

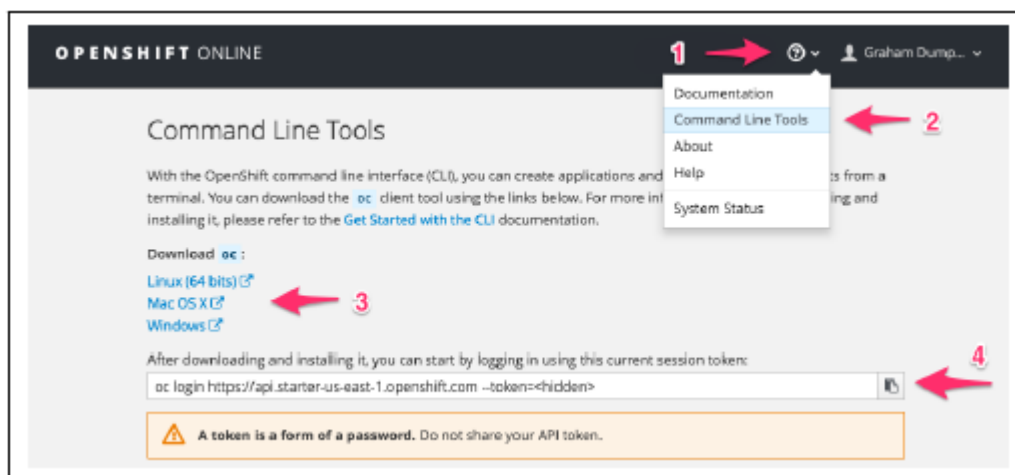


图 3-4 下载命令行工具

1. 从导航栏中选择 “？” 下拉菜单。
2. 选择命令行工具菜单选项。
3. 为您的平台选择下载链接。
4. 单击复制图标以捕获登录命令，包括登录令牌，然后将其粘贴到终端窗口并运行。

使用令牌的登录命令的形式是：

oc login https://api.starter-us-east-1.openshift.com --token=Sbqw....T3UU

登录时使用的令牌将定期过期，并且您需要在登录时再次登录。发生这种情况时，您可以运行不带选项的 oc 登录：

```
$ oc login
```

Login failed (401 Unauthorized)

You must obtain an API token by visiting

https://api.starter-us-east-1.openshift.com/oauth/token/request

这将引导您进入备用页面，如图 3-5 所示，您可以在其中获得一个新的令牌。

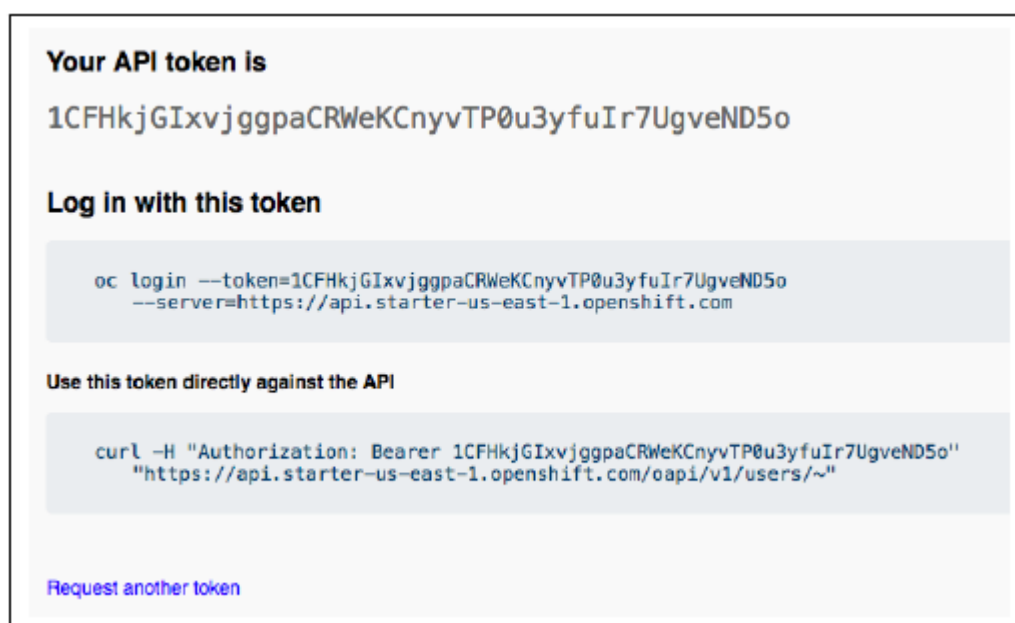


图 3-5 获取新的访问令牌

从命令行登录时使用访问令牌是首选机制，因为它确保会话将定期过期。当 OpenShift 群集不使用外部身份提供程序时，它也可能允许使用用户名和密码从命令行登录。如果是这种情况，运行没有选项的 `oc` 登录会提示您输入凭据：

```
$ oc login
Authentication required for https://127.0.0.1:8443 (openshift)
Username: developer
Password:
Login successful.
You don't have any projects. You can try to create a new project, by running
oc new-project <projectname>
```

有关 `oc` 登录所接受的特定选项或任何其他命令的帮助，请运行该命令但带有 `--help` 选项：

```
$ oc login --help
```

要获得有关 `oc` 命令的一般信息，请不带任何选项运行它。要获得 `oc` 接受的所有命令的列表，请使用 `help` 命令运行它：

```
$ oc help
```

有关所有命令接受的常用选项的详细信息，请使用选项运行它
命令：

```
$ oc options
```

许多命令将接受 `-` 干运行作为一个选项。这可以用来验证您传递命令的选项组合是否正确，而不进行任何更改。

3.3 使用 OpenShift REST API

当您使用 Web 控制台或 `oc` 命令行工具时，它使用 REST API 端点与 OpenShift 进行通信。您还可以直接使用简单的 HTTP 客户端访问此 REST API，或者使用自定义客户端获取从 OpenShift 的 Swagger API 规范生成的特定编程语言。

在针对 REST API 进行 HTTP 调用时，可以使用从命令行登录时使用的相同访问令牌。该标记应作为与 HTTP 请求一起发送的 `Authorization` 头的值的一部分。

在用于获取新令牌的页面中给出了一个用于通过 REST API 请求用户详细信息的 `curl` 命令示例（图 3-5）：

```
curl -H "Authorization: Bearer 1CFH...ND5o" \
https://api.starter-us-east-1.openshift.com/oapi/v1/users/~
```

如果您已从命令行登录并需要令牌，则还可以通过运行 `oc whoami --token` 命令来获取它。在发出请求之前，可以在脚本中使用此命令来获取令牌：

```
#!/bin/sh
SERVER=`oc whoami --show-server`
TOKEN=`oc whoami --token`
URL="$SERVER/oapi/v1/users/~"
curl -H "Authorization: Bearer $TOKEN" $URL
```

与使用 `oc` 登录登录时一样，此令牌将过期并需要更新。

REST API 可用于管理最终用户应用程序，群集和群集用户。本书不会深入研究如何使用 REST API；有关更多信息，请参阅 [OpenShift REST API 文档](#)。

学习如何使用 REST API 的另一个选项是查看 `oc` 命令行工具在使用时的功能。要查看 REST API 调用 `oc` 客户端工具执行命令所执行的操作，请运行该命令并将 `--loglevel 9` 作为选项传递。这将显示有关 `oc` 在做什么的详细消息，包括 REST API 调用的详细信息和内容。

3.4 总结

所有对 OpenShift 的访问都是通过 REST API 进行的。如果您需要创建自己的工具来控制或使用 OpenShift，则可以直接与 REST API 交互，但通常您可以使用 Web 控制台或 `oc` 命令行工具。如果您是开发人员，尤其是高级用户，则主要使用命令行。

使用命令行使用 OpenShift 时，可以在两个级别上进行操作。您可以在 OpenShift 中创建或编辑原始资源定义以控制应用程序的部署方式，或者您可以使用由 `oc` 命令行工具实现的命令和选项为您进行更改。

在本书中，重点将放在使用 `oc` 以及它提供的命令和选项上。与使用原始资源定义相比，这是使用 OpenShift 和 Kubernetes 功能的更简单的途径。

第4章. 应用程序添加到项目

既然您已经知道如何访问您的 OpenShift 集群并可以登录，您几乎可以开始部署第一个应用程序了。

应用程序可以从您在 OpenShift 群集之外构建的现有容器映像进行部署，或者由第三方提供。

或者，如果您有应用程序的源代码，则可以让 OpenShift 为您构建映像。OpenShift 可以根据 Docker 文件提供的指令构建图像，也可以通过 Source-to-Image (S2I) 构建器运行源代码以生成容器图像。

OpenShift 还包括针对流行数据库产品（如 PostgreSQL，MySQL，MongoDB 和 Redis）的现成容器映像。

但是，在部署任何应用程序之前，首先需要在 OpenShift 群集中创建一个项目来包含应用程序。

在本章中，您将了解哪些项目用于以及如何创建它们。您将得到一个快速浏览，了解如何找出 OpenShift 提供的准备运行的容器映像，通过 S2I 构建器支持哪些语言，以及从映像或一组资源定义中部署应用程序的方法。

4.1 一个项目的作用

每当您使用 OpenShift 时，您都将在项目的上下文中工作。这是一个用于容纳与一组应用程序相关的所有内容的有名的命名空间。

当您创建项目时，您是项目的拥有者，并且您是该项目的管理员。您在项目中部署的任何应用程序只有在同一项目中运行的其他应用程序才能看到，除非您选择在 OpenShift 群集之外公开其他应用程序。

您可以将多个应用程序部署到单个项目中。如果他们有紧密的联系，你通常会这样做。或者，您可以选择始终为每个应用程序创建一个单独的项目，并在需要彼此通信的项目之间选择性地设置项目之间的访问权限。

4.2 创建一个项目

当您第一次访问 OpenShift 群集时，您将需要创建一个项目。这是一个例外，当您使用 Minishift 或 oc 集群。因为这些是为本地测试和开发而设计的，为了方便起见，他们将为您设置一个初始项目。

如果您使用 Web 控制台访问 OpenShift 集群并且没有任何项目，您将看到添加新项目的选项（请参见图 4-1）。

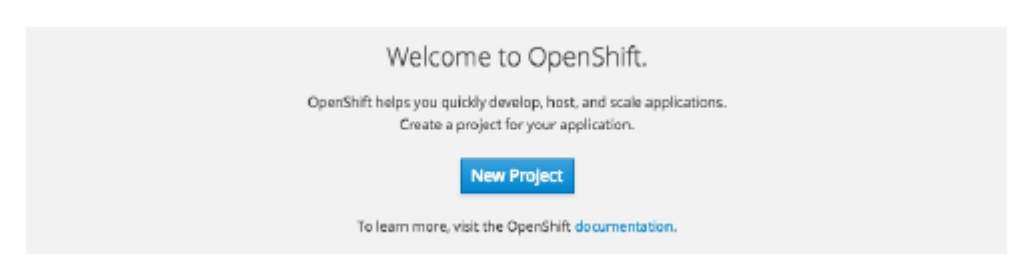


图 4-1 添加一个项目

点击 **New Project** 将弹出图 4-2 中的表格。输入项目的名称并可选择提供显示名称和说明。

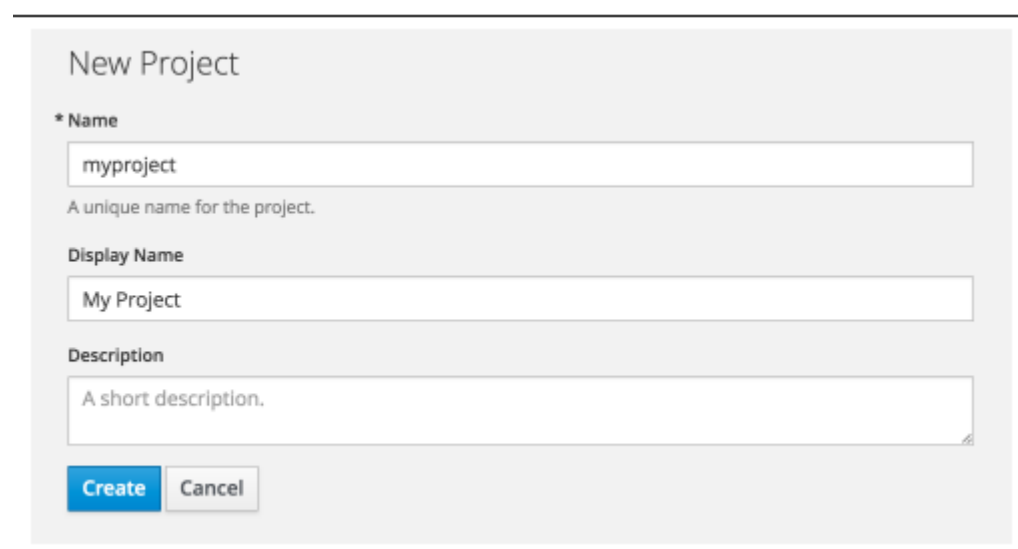
The image shows a 'New Project' form. It has three input fields: 'Name' (with 'myproject' entered), 'Display Name' (with 'My Project' entered), and 'Description' (with 'A short description.' entered). Below the fields are 'Create' and 'Cancel' buttons. The form is titled 'New Project' and has a subtitle '* Name'.

图 4-2。创建一个新项目

当您为项目指定名称时，它需要满足一些要求。

第一个要求是您选择的名称在整个 OpenShift 集群中必须是唯一的。这意味着您不能使用其他用户已在使用的项目名称。

第二个要求是该名称只能包含小写字母，数字和破折号字符。这是必要的，因为项目名称在分配给应用程序的主机名中用作组件时，它在 Open-Shift 群集之外变得可见时使用。

一旦你创建了一个项目并且在 Overview_页面上，你可以点击主页图标跳回到项目列表（图 4-3）。



图 4-3 项目列表

您还可以使用任何项目顶部横幅中的“项目”下拉菜单在项目之间跳转。
通过使用 `oc` 新项目，也可以从命令行创建项目

命令：

```
$ oc new-project myproject --display-name 'My Project'
```

已经在服务器 “`https://localhost:8443`” 上的项目 “myproject” 上。
您可以使用 `'new-app'` 命令将应用程序添加到此项目中。

例如，尝试：

```
oc new-app centos / ruby-22-centos7~https://github.com/openshift/ruby-ex.git
```

在 Ruby 中构建一个新的示例应用程序。

您可以使用 `oc projects` 命令列出您可以访问的所有项目：

```
$ oc project
```

您在此服务器上有一个项目：“我的项目（myproject）”。

在服务器 “`https://localhost:8443`” 上使用项目 “myproject”。

可以通过运行命令 `oc project` 来确定当前项目的名称，您的命令将应用于此项目：

```
$ oc project
```

在服务器 “`https://localhost:8443`” 上使用项目 “myproject”。

当您有权访问多个项目时，可以通过运行 `oc project` 并指定项目名称来设置当前项目：

```
$ oc project myproject
```

现在在服务器 “`https://localhost:8443`” 上使用项目 “myproject”。

当您使用 `oc new-project` 创建新项目时，新项目将自动设置为当前项目。

如果您需要针对不同项目运行单个命令，则可以使用 `--namespace` 选项将项目名称传递给在项目上运行的任何命令：

```
$ oc get templates --namespace openshift
```

`openshift` 项目是一个特殊项目，充当 OpenShift 集群中每个人都可以使用的图像和模板的存储库。虽然它没有出现在您自己的项目列表中，但您仍然可以查询它以获取特定信息。

4.3 添加合作者

作为一个项目的拥有者，最初你是唯一可以访问它并在其中工作的人。如果您需要与其他用户进行项目协作，则可以将其他成员添加到项目中。将用户添加到项目中时，可以将其添加到以下三个主要角色之一中：

管理

一名项目经理。用户将有权查看项目中的任何资源并修改项目中除配额之外的任何资源。具有该项目角色的用户将能够删除该项目。

编辑

用户可以修改项目中的大多数对象，但无权查看或修改角色或绑定。具有此角色的用户可以在项目中创建和删除应用程序。

视图

无法进行任何修改的用户，但可以查看项目中的大多数对象。

要将具有编辑角色的其他用户添加到项目中，以便他们可以创建和删除应用程序，则需要使用 `oc adm` 策略命令。您必须在项目中

当您运行这个命令时：

```
$ oc adm policy add-role-to-user edit <collaborator>
```

用由用户运行时的 `oc whoami` 命令显示的用户名替换 `<collaborator>`。

要从项目中删除用户，请运行：

```
$ oc adm policy remove-role-from-user edit <collaborator>
```

要获取有权访问项目及其角色的用户列表，项目经理可以运行 `oc get rolebindings` 命令。

项目的成员资格还可以通过转到项目列表，单击项目的三点菜单图标，然后选择查看成员资格，从 Web 控制台进行编辑。

4.4 部署应用程序

使用 Web 控制台和命令行 `oc` 客户端，可以通过多种不同的方式将应用程序部署到 OpenShift。

部署应用程序的主要方法是：

- 位于 OpenShift 集群外部的映像注册表中托管的现有容器映像。
- 从已导入到在 OpenShift 群集内运行的映像注册表中的现有容器映像。
- 从 Git 存储库托管服务中的应用程序源代码。应用程序源代码将使用 S2I 构建器构建到 OpenShift 内部的映像中。
- 从 Git 存储库托管服务中的图像源代码。使用 Dockerfile 中提供的指令将图像源代码构建到 OpenShift 内部的图像中。
- 使用命令行 `oc` 客户端从本地文件系统将应用程序源代码压入 OpenShift。应用程序源代码将使用 S2I 构建器构建到 OpenShift 内部的图像中。
- 使用命令行 `oc` 客户端从本地文件系统推送到 OpenShift 的图像源代码。使用 Dockerfile 中提供的指令将图像源代码构建到 OpenShift 内部的图像中。

为了简化具有多个组件部分的应用程序的部署，或者需要在创建应用程序时提供配置，OpenShift 提供了一种定义模板的机制。可以使用模板使用列出的任何方法同时设置一个或多个应用程序的部署。可以在创建应用程序时将参数提供给模板，其中的值用于填充由模板定义的资源对象中的任何配置。

为了获得最大的可配置性和控制能力，还可以使用要创建的资源对象列表直接描述应用程序部署。这些列表可以作为 YAML 或 JSON 提供。

4.5 从目录中部署

当您在 Web 控制台中处于空白项目时，您将看到如图 4-4 所示的选项，以将新应用程序添加到项目中。

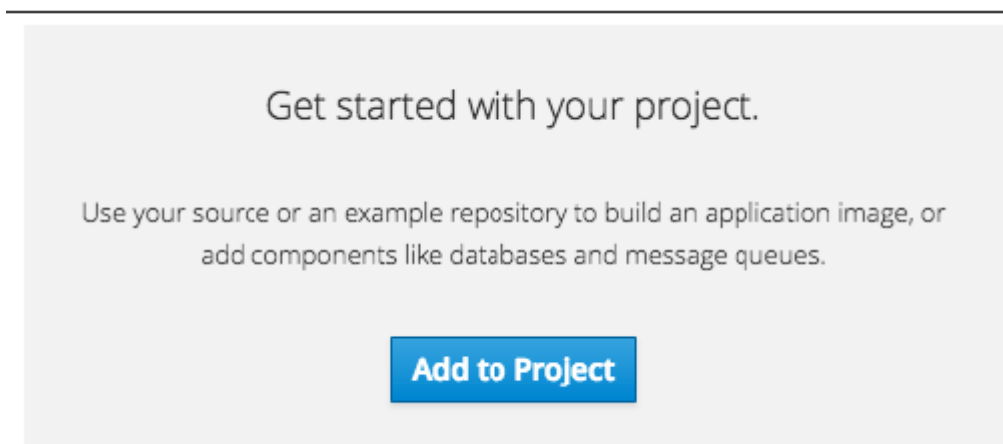


图 4-4 将应用程序添加到项目

单击“添加到项目”将会带您进入应用程序模板和 S2I 构建器的目录，如图 4-5 所示，这些已经预先安装到 OpenShift 集群中。您也可以通过点击任何项目顶部横幅中的“添加到项目”并选择浏览目录来访问此页面。

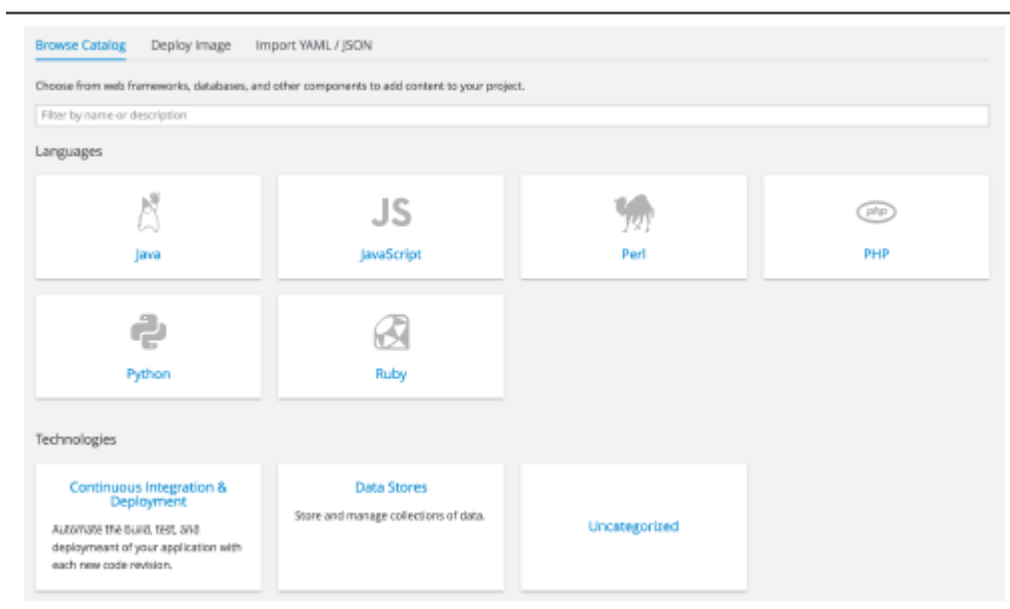


图 4-5 浏览目录

目录是从许多来源自动构建的。

您可以使用的 S2I 构建器的列表是通过在当前项目中查找图像以及标记为构建器图像的 openshift 项目得出的。

应用程序模板列表是通过查询当前项目和 openshift 项目中的模板定义列表生成的。

openshift 项目充当了构建器映像和模板的全局存储库。如果管理员想要将整个构建器映像或应用程序模板提供给整个 OpenShift 集群，则这是他们应该添加它们的位置。

因为 openshift 项目中包含的内容由 OpenShift 群集的管理员控制，所以在 Open-Shift 群集中您在目录中列出的内容可能会有所不同。根据是否使用 OpenShift Origin 或 Red Hat OpenShift Container Platform 产品，出现的内容也可能有所不同。

从命令行可以使用 `oc` 获取模板命令获取应用程序模板列表。这不会为空项目返回任何内容，也不会将任何模板添加到项目中。要列出 openshift 项目中可用的模板，请提供--namespace openshift 选项。

要获取可用图像的列表，请使用命令 `oc get imagestreams`。应再次提供--namespace

openshift 选项以列出打开的班次项目中的那些选项。

当您运行此命令时，并非列出的所有映像都可能对应于构建器映像。这是因为图像也是为通过运行 S2I 构建器而创建的应用程序图像构建的。

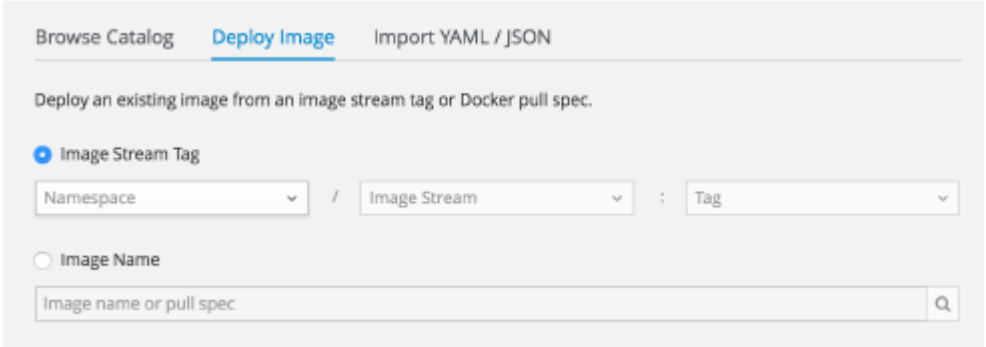
从命令行查看可用的应用程序模板和构建器映像的更好方法是运行 `oc new-app -L` 命令。这会产生类似于浏览目录页面可用的结果，为当前项目和输出中的开放移植项目组合应用程序模板和构建器映像。

要在可用的应用程序模板和构建器映像中搜索，请使用目录浏览器中的过滤器字段。在命令行中，您可以使用 `oc new-app -S` 命令，提供要搜索的关键字。

当您找到符合您需要的条目时，可以从 Web 控制台中选择它。这会通过基于表单的工作流程发送给您，以部署应用程序。在命令行中，您将使用 `oc new-app`。

4.6 部署映像

要部署现有容器映像，请切换到图 4-6 中的“部署映像”选项卡。



The screenshot shows the 'Deploy Image' tab in the OpenShift web console. It has three tabs: 'Browse Catalog', 'Deploy Image' (active), and 'Import YAML / JSON'. Below the tabs, there is a heading 'Deploy an existing image from an image stream tag or Docker pull spec.' and two radio buttons. The 'Image Stream Tag' radio button is selected. Below it, there are three dropdown menus: 'Namespace', 'Image Stream', and 'Tag', separated by slashes and a colon. The 'Image Name' radio button is unselected. Below it, there is a text input field labeled 'Image name or pull spec' with a search icon.

图 4-6 部署图像

要使用驻留在 OpenShift 群集中的图像，请选择图像流标记，然后选择图像所属的项目，图像和标记。您将只能看到您是所有者的项目，您已明确授予访问权限的其他项目以及 openshift 项目。

要使用托管在 OpenShift 集群外部的映像注册表中的映像，请选择映像名称并输入映像的名称，包括使用除 Docker Hub 之外的映像注册表时的映像注册表的主机名。

如果使用命令行，则可以使用 `oc new-app` 命令来部署托管在任何映像注册表上的映像。

4.7 部署一组资源

要从资源对象定义列表中部署应用程序，请切换到如图 4-7 所示的 Import YAML / JSON 选项卡。

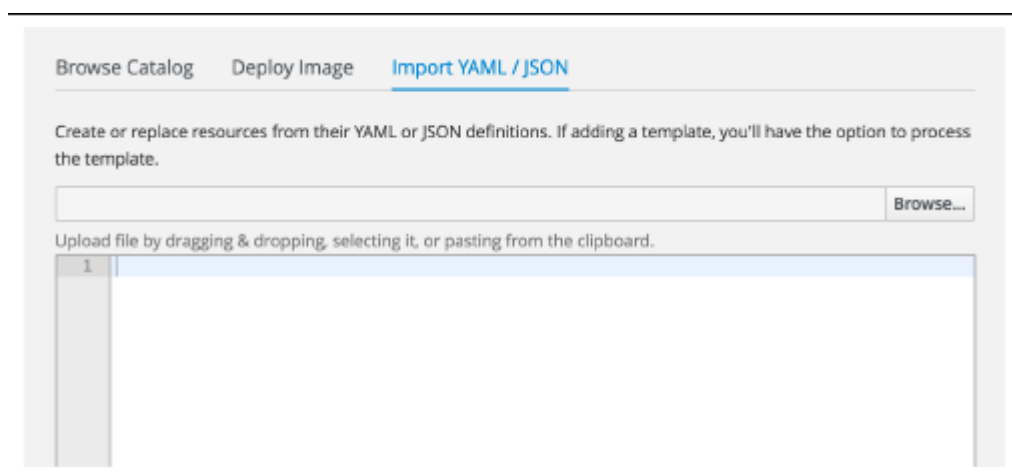


图 4-7 导入 YAML / JSON 定义

YAML 或 JSON 定义可以从本地计算机上传或直接输入到网页中。

如果您提供模板，则会询问您是否希望立即应用模板来部署应用程序或加载它，从而导致可以从浏览器中进行选择，并且可以从 `oc` 命令行客户端使用它来部署应用。

如果使用命令行，可以使用 `oc new-app` 或 `oc create` 从一组资源或模板创建应用程序。

本书不会详细讨论如何创建原始资源定义或模板，或者如何使用它们部署应用程序。有关更多信息，请参阅 [OpenShift 开发人员指南](#)。

4.8 总结

项目为您提供了一个可以部署应用程序的空间。您可以选择在单个项目中执行所有操作，也可以使用多个项目并选择性地在项目之间启用访问，以便不同的应用程序组件可以相互通信。

在您自己的项目之间提供隔离的相同功能是为了在多租户环境中分隔不同用户的功能。

当您使用 **OpenShift** 部署应用程序时，您可以使用自己的用户帐户，并控制您可以执行的操作。作为开发人员，您不会使用管理员帐户。如果其他人需要在应用程序上与您一起工作，您可以授予他们对运行特定应用程序的项目的必要访问权限。

用户帐户以及由 **OpenShift** 多租户功能提供的名称空间之间的额外隔离级别是将 **OpenShift** 与标准 **Kubernetes** 环境的工作方式区分开来的关键特性。这些功能是使 **OpenShift** 更安全的一部分，也是企业环境的更好选择。

第 5 章 从映像部署应用程序

现在您已经创建了一个项目，您可以继续部署应用程序。

在本章中，您将首先从托管在外部映像注册表上的预先存在的容器映像部署应用程序。

如果您为 **OpenShift** 集群外部的应用程序创建了映像，或者第三方提供了该映像，则可以使用此方法。

一旦你部署了应用程序，你将公开它，以使用户可以访问它。然后，您将使用环境变量重新配置正在运行的应用程序，并且扩大应用程序的实例数量以处理越来越多的流量。您还将看到如何删除应用程序。

5.1 部署您的第一个人镜像

OpenShift 支持部署托管在任何可从 OpenShift 集群访问的映像注册表上的容器映像。您将部署的第一个映像存储在 Docker Hub 上，名为 `openshiftkatacoda / blog-django-py`。图像中的应用程序实现了一个简单的博客站点。

使用的图像的全名是 `docker.io/openshiftkatacoda/blog-django-py`。当您离开映像注册表的主机名时，OpenShift 将默认首先在群集配置中指定的任何全局映像注册表上查找映像。Docker Hub 映像注册表通常包含在该列表中。公司图像注册表或红帽容器注册表也可能包含在内。

要部署容器图像，请使用 `oc new-app` 命令，为它提供图像的位置。`--name` 选项是为已部署的应用程序设置名称。如果没有提供名称，它将默认为图像名称的最后一部分。我们将使用 `name block`：

```
$ oc new-app openshiftkatacoda / blog-django-py --name blog
```

->找到 Docker Hub for “`openshiftkatacoda / blog-django-py`” 的 Docker image 0f405dd (5 天前)

...

- 图像流将被创建为“博客：最新”，将跟踪这个图片
- 此图片将部署在配置配置“博客”
- 端口 8080 / TCP 将通过服务“bl”进行负载平衡
- 其他容器可以通过主机名“博客”访问此服务

-->创建资源...

imagestream “博客” 创建

部署配置 “博客” 创建

服务 “博客” 创建

-->成功

运行'oc 状态'查看您的应用程序。

OpenShift 需要做什么。在这种情况下，`imagestream`，`deploymentconfig` 和服务已创建。

`imagestream` 是您想要部署的映像的记录。`deploymentconfig` 捕获部署应该如何完成的细节。该服务维护一个到应用程序实例的映射，以便可以访问它。

某些资源对象类型具有可用于命令或在命令输出中出现的别名。例如，可以使用 `svc` 代替服务，使用 `dc` 代替 `deploymentconfig`，并且代替图像流。您可以通过运行 `oc get` 来查看所有资源对象类型和任何名称别名的列表。您可以通过运行 `oc` 类型来查看主要资源对象类型的描述。

使用 `oc new-app` 从命令行部署容器映像时，正在运行的容器在 OpenShift 集群外部不可见。对于 Web 应用程序，您可以通过公开该服务来使其可见。这是使用 `oc expose` 命令完成的：

```
$ oc expose service/blog
```

```
route "blog" exposed
```

这将创建一个名为路由的资源对象。

通过部署并显示应用程序，您可以使用 `oc status` 命令检查整个项目的状态：

```
$ oc status
```

在项目我的项目 (myproject) 在服务器 `https://127.0.0.1:8443`

http://blog-myproject.127.0.0.1.nip.io 到 pod 端口 8080-tcp (svc / blog)

dc/blog deploys istag/blog:latest

deployment #1 deployed 1 minute ago - 1 pod

'oc describe <resource> / <name>'查看详细信息，或者用'oc get all'列出所有内容。

要获得已部署应用程序实例的列表，您可以使用命令 `oc get pods`:

\$ oc get pods

NAME	READY	STATUS	RESTARTS	AGE
blog-1-36f46	1/1	Running	0	1m

此时将只有一个。要获得为应用程序创建的资源对象列表，可以使用 `oc get all` 命令:

\$ oc get all -o name --selector app=blog

imagestreams/blog

deploymentconfigs/blog

replicationcontrollers/blog-1

routes/blog

services/blog

pods/blog-1-36f46

为了确保只显示部署的应用程序的资源，应该在标签 `app = blog` 上使用选择器并匹配，该标签已添加到由 `oc new-app` 和 `oc expose` 命令创建的所有资源对象。

`pod` 资源对象表示一组一个或多个容器，但在大多数情况下，一个容器只能容纳一个容器。您将看到每个应用程序实例的唯一容器。容器中的容器一起部署，并作为一个组来启动，停止和复制。

复制控制器是从 `deploymentconfig` 创建的。

它记录了应用程序在任何时间点应该运行多少个实例的计数。OpenShift 将根据需要创建尽可能多的窗格以匹配所需的计数。

如图 5-1 所示，OpenShift Web 控制台也可用于查看应用程序。

您将在项目总览中找到已部署应用程序的摘要。

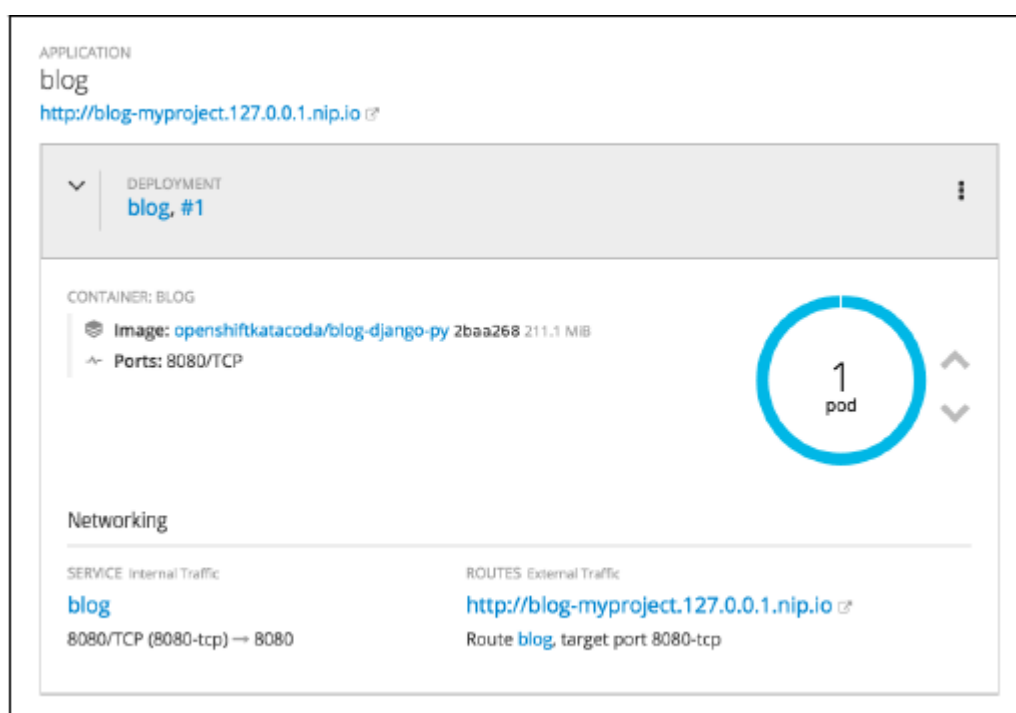


图 5-1 带路由的博客站点概述使用 Web 浏览器访问应用程序摘要中显示的 URL，您将看到博客站点的主页（图 5-2）。

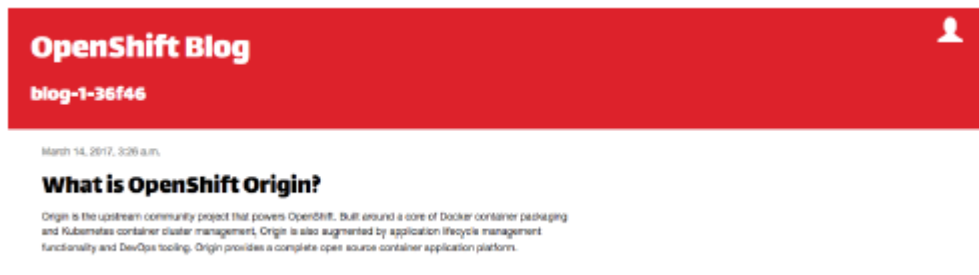


图 5-2 博客网站主页您还可以运行 `oc` 获取路线命令以查看已公开的任何服务的详细信息。该命令将显示 OpenShift 分配给应用程序的主机名。

5.2 扩展应用程序

使用 `oc new-app` 从容器映像部署应用程序时，只会启动应用程序的一个实例。如果您需要运行多个实例才能处理预期流量，则可以通过针对部署配置运行 `oc scale` 命令来扩展实例数量：

```
$ oc scale --replicas=3 dc/blog
deploymentconfig "blog" scaled
```

当 Web 应用程序扩大时，OpenShift 会自动重新配置公开的路由器，以便在应用程序的所有实例之间进行负载平衡。

如果您再次运行 `oc get pod`，现在应该可以看到应用程序的三个实例。您可以根据为应用程序收集的度量标准启用自动缩放，而不是手动缩放实例数量。有关更多信息，请查看 [pod autoscaling](#) 上的 OpenShift 文档。

5.3 运行时间配置

应用程序的配置可以通过在容器中设置环境变量或将配置文件挂载到容器中来提供。

运行 `oc new-app` 命令时，使用 `--env` 选项设置所需的环境变量：

```
$ oc new-app openshiftkatacoda/blog-django-py --name blog \
--env BLOG_BANNER_COLOR=green
```

可以稍后通过针对部署配置运行 `oc set env` 命令来设置可选的环境变量：

```
$ oc set env dc/blog BLOG_BANNER_COLOR=green
deploymentconfig "blog" updated
```

当使用 `oc set env` 更新环境变量时，将使用新配置自动重新部署应用程序。如果您想查看容器中将设置哪些环境变量，则可以使用 `oc set env` 和 `-list` 选项：

```
$ oc set env dc/blog --list
# deploymentconfigs blog, container blog
BLOG_BANNER_COLOR=green
```

配置和环境变量的主题将在 [第 12 章](#) 中详细介绍。

5.4 删除应用程序

当您不再需要该应用程序时，可以使用 `oc delete` 命令将其删除。

这样做时，您需要选择删除哪些资源对象，以确保只删除那些特定应用程序的对象。这可以通过使用由 `oc new-app` 和 `oc expose` 命令应用于创建的资源对象的标签来实现：

```
$ oc delete all --selector app=blog
imagestream "blog" deleted
deploymentconfig "blog" deleted
route "blog" deleted
service "blog" deleted
```

`pod` 和 `replicationcontroller` 可能不会在运行 `oc` 删除的输出中列出，但它们将被删除。这是因为它们被删除，作为删除应用程序的 `deploymentconfig` 的副作用。

5.5 使用 Web 控制台进行部署

要使用 Web 控制台部署预先存在的容器图像，请从项目中选择添加到项目，然后按照图 5-3 中列出的步骤进行操作。

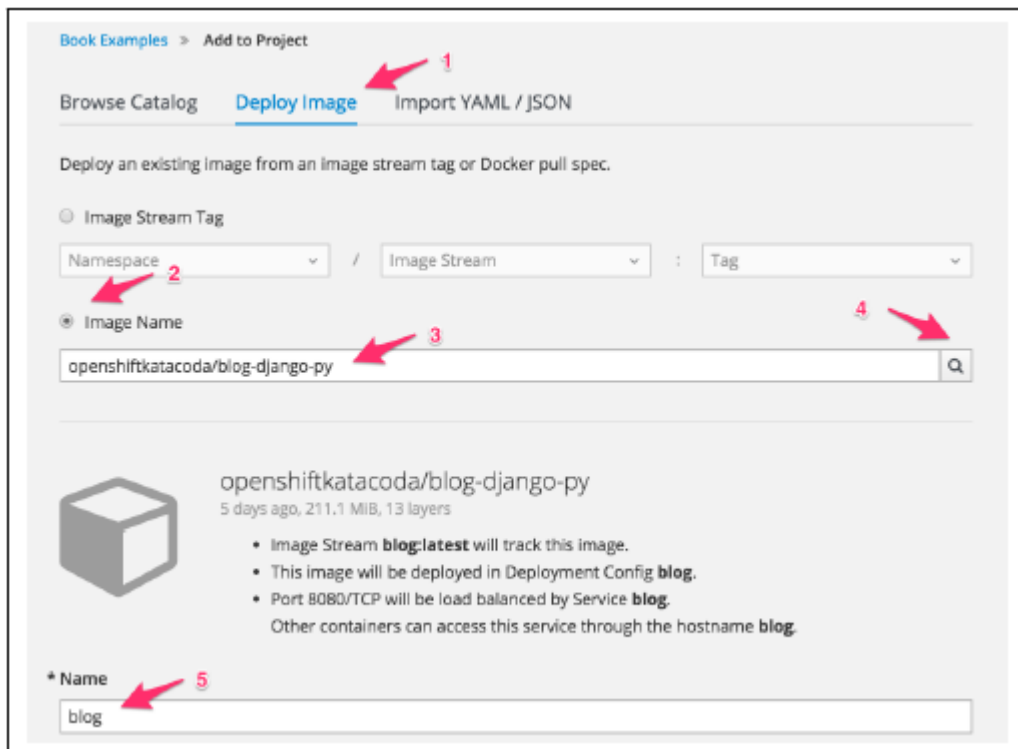


图 5-3。部署应用程序映像

1. 点击 `Deploy Image` 标签。
2. 选择图像名称以使用存储在外部图像注册表中的图像。
3. 输入 `openshiftkatacoda / blog-django-py` 作为图像名称的值。
4. 按 `Enter` 键，或点击查询图标以下拉图像的详情。
5. 将应用程序的名称更改为博客。可以通过单击页面底部的创建来创建图像的

部署。

通过选择 **Create Route**（图 5-4），可以从项目的概览页面创建在 OpenShift 集群外公开服务的路径。

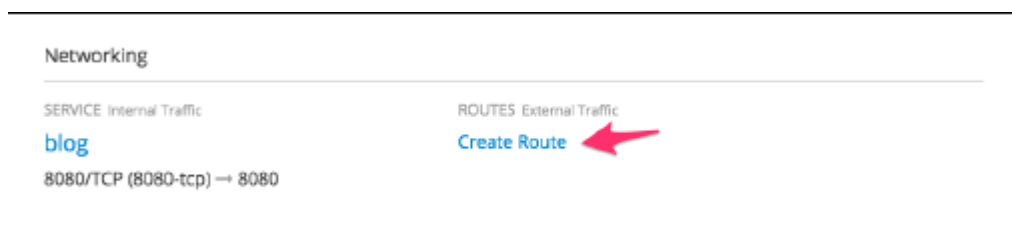


图 5-4 创建路线链接

服务，网络和路线将在第 13 章中详细介绍。

概览页面还提供了扩展或缩减部署的 Web 应用程序实例数量的功能。使用圆圈右侧的向上和向下箭头显示正在运行的窗格的数量和状态。

5.6 导入镜像

当您从托管在外部映像注册表中的现有容器映像部署应用程序时，会下载映像副本并将其存储到 OpenShift 内部的映像注册表中。然后将映像从那里复制到运行应用程序的集群中的每个节点。

为了跟踪下载的图像，创建图像流定义。要查看图像流定义的列表，请运行 `oc get`：

```
$ oc get is
```

NAME	DOCKER REPO	TAGS	UPDATED
blog	172.30.118.67:5000/book/blog	latest	About a minute ago

在 DOCKER REPO 下显示的 <IP>: <PORT> 是内部映像注册表的地址。

由于该图像正在作为应用程序部署的一部分使用，因此它使用应用程序的应用程序标签进行了标记。如果使用标签删除应用程序，这也会删除图像流和图像。

如果您需要从一个映像部署多个单独的应用程序，则应先使用 `oc import-image` 将映像导入 OpenShift：

```
$ oc import-image openshiftkatacoda/blog-django-py --confirm
```

```
The import completed successfully.
```

```
Name: blog-django-py
```

```
...
```

然后，您可以从导入的映像中部署应用程序：

```
$ oc new-app blog-django-py --name blog
```

在 Web 控制台中，您可以使用“图像流标记”选项，而不是“部署映像”页面上的图像名称选项。

由于映像是在部署应用程序之前导入的，因此不会使用特定部署的应用程序标签进行标记。当您使用标签删除任何应用程序时，您将不会删除图像流，而是将其保留在依赖于它的其他应用程序中。

5.7 推送到注册表

到目前为止描述的从图像部署的方法依赖于能够从外部图像注册表中拉出图像。如果

您在自己的本地计算机上使用工具来构建图像，则可以绕过首先将图像推送到外部图像注册表的需要，而直接将其推送到 OpenShift 的内部图像注册表。

要使用内部映像注册表，您需要知道访问它的地址。无法从 OpenShift Web 控制台或 oc 命令行客户端获得此信息，因此，如果可以访问内部映像注册表以及使用哪个主机名和端口，则需要向管理员询问 OpenShift 群集。如果使用托管的 OpenShift 服务，请检查其文档。对于 OpenShift Online，可以使用以下格式的地址访问内部映像注册表：

注册表<群集名称>.openshift.com: 443

要使用 docker 命令登录，请运行：

```
$ docker login -u`oc whoami` -p`oc whoami --show-token` \
registry.pro-us-east-1.openshift.com:443
登录成功
```

在您推送图像之前，您需要使用 oc create imagestream 为其创建一个空图像流。

```
$ oc create imagestream blog-django-py
```

接下来，使用图像注册表的详细信息，OpenShift 中的项目以及图像流和图像版本标记的名称标记要推送的本地图像：

```
$ docker tag blog-django-py \
registry.pro-us-east-1.openshift.com:443/book/blog-django-py:latest
```

然后准备将图像推送到 OpenShift 内部图像注册表：

```
$ docker push registry.pro-us-east-1.openshift.com:443/book/blog-django-py
```

然后可以使用图像流名称来部署应用程序。

5.8 图片和安全

当您的应用程序部署到 OpenShift 时，默认安全模型将强制执行，使用您要部署到的项目特有的分配的 Unix 用户标识运行它。此行为是作为 OpenShift 的多租户功能的一部分实现的，但也是为了防止以 Unix root 用户身份运行映像。

虽然容器提供了一个旨在防止应用程序能够与底层主机操作系统交互的 sandbox 环境，但如果容器运行时出现安全漏洞，允许应用程序脱离 sandbox 并且应用程序以 root 身份运行，它可能成为底层主机的根。

图像的最佳实践是将它们设计为可以像任何 Unix 用户标识一样运行。公共图像注册中心提供的许多图像不符合这种做法，即使它们不需要提升特权，也需要以 root 身份运行。这意味着在典型的 OpenShift 环境中，并非您在公共映像注册表中找到的所有映像都能正常工作。

如果它是你自己的图像，你应该重新设计图像，所以它不必以 root 身份运行。如果它确实需要以 root 身份运行，那么只有集群管理员才能授予这种能力。

为了允许在项目中部署的任何应用程序能够以容器映像指定的用户（包括 root）运行，集群管理员可以针对项目运行以下命令：

```
$ oadm policy add-scc-to-user anyuid -z default
```

集群管理员只会与作为根用户运行映像相关的风险进行评估之后才允许这样做。从公共图像注册表中获取根任意图像运行永远不是好习惯。

有关此主题的更多信息，请参阅有关[管理安全上下文约束](#)的 OpenShift 文档以及有关[创建映像的指导原则](#)。

5.9 总结

由于 Kubernetes 提供的容器即服务（CaaS）功能，因此可以在 OpenShift 中部署预先存在的容器图像。使用标准 Kubernetes 安装时，图像需要从外部图像注册表中提取。

在使用 OpenShift 时，您仍然可以在部署应用程序时直接从外部图像注册表中提取图像，但 OpenShift 还提供图像注册表作为环境的一部分。您可以将图像推送到内部图像注册表中，或者可以设置图像流定义，以便在部署图像时自动将其拉入并缓存在内部图像注册表中。从 Web 控制台或命令行部署图像时，OpenShift 会自动为您设置此图像流。

托管 OpenShift 内部的映像注册表以缓存映像可加快部署应用程序的速度。在将应用程序部署到集群中的节点时，只需从本地内部映像注册表中提取映像，而不必连接到外部映像注册表。

尽管许多来自外部源的图像都可以在 OpenShift 中使用，但是已经存在的防止应用程序作为 Unix root 用户运行的默认限制可能意味着图像需要修改才能正常工作。或者，可以放宽安全性以允许图像按照图像指定的用户运行。最佳做法是确保图像可以作为任意分配的 Unix 用户标识运行，而不是作为特定的 Unix 用户标识运行。

第 6 章从源代码构建和部署

在部署预先存在的容器图像时，如果这是您自己的图像，这意味着您需要有单独的工具来构建该图像。您还必须将图像上载到 OpenShift 可以将其拉下的图像注册表中，或将图像推送到 OpenShift 的内部图像注册表中。

为了简化应用程序的发布管理流程，OpenShift 提供了为您构建映像的功能。当您想要自动执行完整的工作流程时，您可以使用它，包括构建映像，对映像进行任何测试，然后进行部署。

OpenShift 提供了四种不同的构建策略：

资源

这使用 Source-to-Image 通过将应用程序源（或其他资源）注入构建器映像来生成可立即运行的映像。

搬运工人

这使用 docker build 来获取 Dockerfile 和相关的源文件并创建一个可运行的图像。

管道

这使用 Jenkins 和由 Jenkinsfile 定义的工作流来创建用于构建可运行图像的管道。

习惯

这会使用您自己的自定义图像来控制用于创建可运行图像的构建过程。

在本章中，您将学习如何使用 Source 构建策略从托管 Git 存储库中的源代码构建和部署应用程序，或从本地计算机将源代码压入 OpenShift。

有关如何使用 Docker 构建策略的详细信息将在[第 7 章](#)中介绍。但是，本书不会介绍最后两种构建策略。有关[管道](#)和自定义[构建策略](#)的更多详细信息，请参阅 OpenShift 文档。

6.1 源代码构建策略

Source 构建策略使用 [Source-to-Image \(S2I\)](#) 工具从应用程序源代码构建可运行图像。

要从应用程序源构建，需要包含源文件的托管 Git 存储库。此 Git 存储库必须可供您执行构建的 OpenShift 群集访问。

OpenShift 为通用编程语言（包括 Java，NodeJS，Perl，PHP，Python 和 Ruby）提供了 S2I 构建器。构建者将获取您的应用程序源代码，必要时进行编译，并将其与构建器映像提供的应用程序服务器堆栈进行集成。容器运行时，服务器将启动并运行您的应用程序代码。

Builder 图像不限于用于从源代码构建应用程序。任何形式的输入数据都可以与构建器图像结合使用以创建可运行的图像。

为了说明 Source 构建策略，您将部署与前一章中相同的 Web 应用程序。而在上一章中，您使用预先存在的容器映像部署它，这次您将从应用程序源代码部署它。有问题的 Web 应用程序是使用 Python 编程语言实现的，因此您将使用 Python S2I 构建器。

6.2 从源部署

在上一章中，要部署博客站点容器映像，请使用以下命令：

```
$ oc new-app openshiftkatacoda / blog-django-py --name blog
```

从 Docker Hub 获取预先存在的容器镜像 openshiftkatacoda / blog-django-py，部署它并启动 Web 应用程序。

要从应用程序源代码进行部署，这次使用以下命令：

```
$ oc new-app --name blog \  
python:3.5~https://github.com/openshift-katacoda/blog-django-py
```

代替图像名称，使用了 S2I 构建器的名称和包含 Web 应用程序源文件的存储库的 URL。通过在这两个值之间插入~来将这些组合成命令的单个参数。oc new-app 命令将解释此特殊组合，表示应使用 Source 构建策略。

运行此命令的结果是：

```
--> Found image 956e2bd (5 days old) in image stream "openshift/python"  
under tag "3.5" for "python"  
Python 3.5  
-----  
Platform for building and running Python 3.5 applications  
Tags: builder, python, python35, rh-python35  
* A source build using source code from  
https://github.com/openshift-katacoda/blog-django-py will be created  
* The resulting image will be pushed to image stream "blog:latest"  
* Use 'start-build' to trigger a new build  
* This image will be deployed in deployment config "blog"  
* Port 8080/tcp will be load balanced by service "blog"  
* Other containers can access this service through the hostname "blog"  
--> Creating resources ...  
imagestream "blog" created  
buildconfig "blog" created  
deploymentconfig "blog" created  
service "blog" created  
--> Success  
Build scheduled, use 'oc logs -f bc/blog' to track its progress.
```

Run 'oc status' to view your app.

除了从映像部署时创建的资源对象之外，还创建了一个 `buildconfig`。这捕获了关于如何在 OpenShift 中构建图像的细节。

`buildconfig` 也可以使用 `bc` 别名来引用。

要监视应用程序映像发生时的构建，可以运行以下命令：

```
$ oc logs -f bc / blog
```

您可以通过运行以下命令在 OpenShift 集群外部公开该服务：

```
$ oc expose svc/blog
```

使用 Web 控制台查看访问 Web 应用程序的 URL，或运行 `oc get routes / blog` 命令以确定分配给它的唯一主机名。如果您访问该网站，则应该再次访问博客网站主页。

6.3 创建一个单独的版本

当源构建策略由 `oc new-app` 调用时，它将设置两个步骤。第一步是使用 S2I 运行构建，将源文件与构建器映像相结合以创建可运行映像。第二步是部署可运行图像并启动 Web 应用程序。

您可以通过运行 `oc new-build` 命令而不是 `oc new-app` 命令分别执行构建步骤：

```
$ oc new-build --name blog \  
python:3.5~https://github.com/openshift-katacoda/blog-django-py  
--> Found image 956e2bd (5 days old) in image stream "openshift/python"  
under tag "3.5" for "python"  
Python 3.5  
-----  
Platform for building and running Python 3.5 applications  
Tags: builder, python, python35, rh-python35  
* A source build using source code from  
https://github.com/openshift-katacoda/blog-django-py will be created  
* The resulting image will be pushed to image stream "blog:latest"  
* Use 'start-build' to trigger a new build  
--> Creating resources with label build=blog ...  
imagestream "blog" created  
buildconfig "blog" created  
--> Success  
Build configuration "blog" created and build triggered.  
Run 'oc logs -f bc/blog' to stream the build progress.
```

输出看起来很相似，但不会创建 `deploymentconfig` 和 `service` 资源对象。

构建完成后，创建的可运行图像将被保存为称为博客的图像流。要部署该映像，使用 `oc new-app` 命令，但是这次提供了由构建创建的图像流的名称，而不是构建器映像和存储库 URL 的详细信息：

```
$ oc new-app blog  
--> Found image 6792c9e (32 seconds old) in image stream "myproject/blog"  
under tag "latest" for "blog"  
myproject/blog-1:0b39e4f7  
-----
```

```
Platform for building and running Python 3.5 applications
Tags: builder, python, python35, rh-python35
* This image will be deployed in deployment config "blog"
* Port 8080/tcp will be load balanced by service "blog"
* Other containers can access this service through the hostname "blog"
--> Creating resources ...
deploymentconfig "blog" created
service "blog" created
--> Success
Run 'oc status' to view your app.
```

该服务再次可以暴露使用 `oc expose`。

6.4 触发新构建

如果用作 Source 构建策略输入的源文件已更改，则可以使用 `oc start-build` 命令触发新构建：

```
$ oc get bc
NAME      TYPE      FROM      LATEST
blog      Source    Git       1
$ oc start-build bc/blog
build "blog-2" started
```

即使您创建了与设置部署分开的构建，但在构建完成并更新了图像流后，重新部署将自动触发。发生这种情况是因为 `oc new-app` 会在部署配置中自动设置图像更改触发器。

触发器也被定义为构建配置的一部分。其中第一个是另一个图像更改触发器。如果更新了 S2I 构建器图像 `python: 3.5`，此触发器将导致重建。

这是构建过程的一个重要特征。当使用 S2I 构建器从不同的源文件创建多个不同的应用程序时，如果有安全修补程序可用，则更新构建器映像将自动触发重建和重新部署使用它的所有应用程序。这使得在需要修补它们使用的图像时可以快速更新应用程序。

当提供 S2I 构建器的名称时，如果您未指定版本标签，它将默认使用最新的标签。在编写本书时，Python S2I 构建器的情况下，最新映射到 3.5。如果以后安装了用于 Python 3.6 的 S2I 构建器，并且将最新的标记重新映射以引用它，则会导致应用程序被更新版本的 Python 重新构建。如果您的应用程序代码尚未准备好用于 Python 3.6，则可能会导致构建失败或应用程序无法正常运行。建议在 S2I 构建器上使用版本标签时，指定您想要的确切版本并避免使用最新版本。

其他触发器也在构建配置中定义，以跟踪源代码更改。您可以为您的源代码存储库配置托管服务，以在代码更改推送到您的存储库时通知 OpenShift。然后，源代码触发器将确保最新的代码被下拉，并且 S2I 构建过程针对它运行以生成更新的应用程序映像，并且因此应用程序也被重新部署。[第 11 章](#)将进一步讨论源代码更改时自动构建的主题。

6.5 从本地来源构建

构建的构建配置使用托管在源代码库中的源文件。要重建应用程序，您的代码更改必须推送到存储库。虽然构建链接到托管的源代码存储库，但一次构建可绕过存储库并使用

本地文件系统上的源文件。

使用 `oc start-build` 触发从本地源构建，使用 `--from-dir` 选项指定本地源目录的位置：

```
$ oc start-build bc/blog --from-dir=.  
Uploading directory "." as binary input for the build ...  
build "blog-3" started
```

要恢复使用托管代码存储库中的源文件，请启动新的版本，而不指定文件的输入源：

```
$ oc start-build bc/blog  
build "blog-4" started
```

6.6 二进制输入构建

二进制输入源构建。这在开发应用程序时很有用，因为您可以迭代更改而无需提交并将更改推送到存储库。只有在检查结果并完成后才提交并推送更改。

构建配置可以从一开始就设置为二进制输入源构建。在此配置中，它不会链接到托管的源代码库，并且所有的构建都需要手动触发并提供源文件。

要创建二进制输入构建配置，请使用 `oc new-build` 并提供 `--binary` 选项：

```
$ oc new-build --name blog --binary --strategy=source --image-stream python:3.5  
--> Found image 440f01a (6 days old) in image stream "openshift/python"  
under tag "3.5" for "python"  
Python 3.5  
-----  
构建和运行 Python 3.5 应用程序的平台  
Tags: builder, python, python35, rh-python35  
* A source build using binary input will be created  
* The resulting image will be pushed to image stream "blog:latest"  
* A binary build was created, use 'start-build --from-dir' to trigger  
a new build  
--> Creating resources with label build=blog ...  
imagestream "blog" created  
buildconfig "blog" created  
--> Success
```

使用 `oc start-build` 触发初始构建和后续构建，并提供 `from-dir` 选项以使用本地目录中的源文件：

```
$ oc start-build blog --from-dir=.  
Uploading directory "." as binary input for the build ...  
build "blog-1" started
```

由于构建配置未链接到源代码存储库，并且 `oc start-build` 必须每次都手动运行，所以如果 S2I 构建器映像发生更改，则无法自动重建和重新部署应用程序。这是因为输入源不能用于构建。

当你有一个现有的工具链来创建应用程序二进制文件或要包含在图像中的组件时，二进制输入构建很有用。一个例子是创建一个 Java WAR 文件，然后使用 S2I 构建器将其注入到包含 Java servlet 容器运行时的基本映像中。

6.7 测试容器镜像

构建过程创建的映像部署时将不会被测试。如果您希望能够对构建中使用的应用程序源代码运行单元测试，或者在将图像推入内部映像注册表之前对其进行验证，则可以在构建中使用后提交挂钩。

测试通过启动一个新的容器和最近构建的镜像并在容器中运行 `post-commit hook` 命令来运行。如果由构建钩子运行的命令返回非零退出代码，则生成的图像将不会被推送到注册表，并且构建将被标记为失败。

要指定要作为提交后挂接运行的命令，请运行 `oc set buildhook` 命令。例如：

```
$ oc set build-hook bc / blog --post-commit --script "powershift image verify"
```

当使用 `--script` 选项指定命令时，原始图像入口点将保持原样，同时覆盖图像命令。如果使用 `--command` 选项指定命令，它将用于替换原始图像入口点。如果使用 `-`，则可以提供传递给原始图像命令的参数。

要移除构建钩子，可以使用 `--remove` 选项。

从后提交挂钩运行单元测试时，应避免联系其他服务，因为容器将与部署的应用程序在同一项目中运行。这是为了避免对您的生产服务意外运行测试。如果测试需要数据库，请运行本地基于文件系统的数据库，如 SQLite。

对于更复杂的端到端集成测试，请使用生产环境中的单独项目来运行构建和测试。当图像通过测试后，您可以将其提升为用于生产的项目。使用集成 Jenkins 安装的管道可用于管理高级构建。有关更多详细信息，请参阅[管道](#)上的 OpenShift 文档。

6.8 构建和运行时间配置

与从容器映像部署应用程序时类似，运行容器时要设置的环境变量可以使用 `--env` 选项指定，以指定新应用程序。这可以在直接从源代码部署时，或者使用 `oc new-app` 部署由独立构建步骤创建的映像时完成。

如果需要为构建步骤设置环境变量，并且您正在从源代码直接部署，请使用 `--build-env` 选项来创建新应用程序：

```
$ oc new-app --name blog --build-env UPGRADE_PIP_TO_LATEST=1 \
python:3.5~https://github.com/openshift-katacoda/blog-django-py
```

如果使用调用 `oc new-build` 和 `oc new-app` 的两步方法，则应使用 `--env` 选项将特定于构建步骤的环境变量传递给 `oc new-build`：

```
$ oc new-build --name blog --env UPGRADE_PIP_TO_LATEST=1 \
python:3.5~https://github.com/openshift-katacoda/blog-django-py
```

如果在创建构建配置之后需要添加环境变量，则可以使用 `oc set env` 命令：

```
$ oc set env bc/blog UPGRADE_PIP_TO_LATEST=1
buildconfig "blog" updated
```

可能需要构建时环境变量来设置代理详细信息或自定义由 S2I 构建器实现的构建过程。您可以通过使用 `oc set env` 和 `--list` 选项来查看为构建设置的环境变量。

```
$ oc set env bc/blog --list
# buildconfigs blog
UPGRADE_PIP_TO_LATEST=1
```

虽然环境变量是在构建配置中指定的，但它们也将在创建的映像中设置，并且在部署时应用程序可见。

6.9 总结

OpenShift 为您创建应用程序映像的能力是源代码部署应用程序的能力，是 OpenShift 提供的平台即服务（PaaS）功能的一项功能。这是建立在 Kubernetes 之上的另一层功能。

Source 构建策略使用 Source-to-Image 工具从应用程序源代码构建可运行图像。OpenShift 为许多常用编程语言提供了 S2I 构建器。如果您需要支持自定义应用程序堆栈，您还可以创建自己的 S2I 构建器映像。

自动构建和部署机制意味着当构建完成并且任何测试成功运行时，应用程序映像将被部署。

第 7 章. 从 Dockerfile 构建一个镜像

使用 S2I 构建器创建应用程序映像可以简化构建过程，因为构建器会为您完成所有艰苦的工作。为了简单起见，S2I 构建器的作者将事先做出关于使用哪个应用服务器堆栈，如何配置以及如何构建应用程序源代码的决定。

尽管您可以覆盖应用程序的 S2I 构建和部署过程，但您可以制作的自定义范围受到限制。例如，您不能安装其他操作系统软件包或运行任何需要 root 权限的操作。

为了完全控制图像的构建和应用程序的运行，您需要使用 Docker 构建策略。

在本章中，您将学习如何使用 Docker 构建策略从 Dockerfile 在 OpenShift 中构建容器图像。生成的容器图像可以是应用程序图像或您自己的自定义 S2I 构建器图像。

7.1 Docker 构建策略

Docker 构建策略从 Dockerfile 获取指令并使用它们构建容器映像。构建映像所需的 Dockerfile 和相关文件需要位于您执行构建的 OpenShift 集群可访问的托管源代码库中。

该策略可用于构建应用程序映像或 S2I 构建器映像。通过让 OpenShift 执行构建，您可以避免使用单独的基础结构来创建映像。图像的构建也可以通过图像触发器链接到依赖于图像的任何构建或部署。因此，对图像的更新将触发后续的构建和部署，使您的工作流程实现自动化。

为了说明如何从 Dockerfile 设置构建，我们将构建与[第 5 章](#)中用于部署我们的博客站点相同的映像，但在 OpenShift 中执行。然后，我们将快速回顾一下如何部署映像。这次我们将使用本地构建的映像，而不是从外部映像注册表中删除的映像。

7.2 安全和 Docker 构建

在使用此构建策略之前，了解使用它的安全性意义非常重要。

构建策略通过使用 `docker build` 命令来处理 Dockerfile 中包含的指令。虽然构建过程是

从容器启动的，但它有必要以 root 用户身份运行。这样就可以拥有与 Docker 守护进程交互的适当权限。Docker 守护进程生成的映像也以 root 用户身份运行，以便执行诸如安装映像所需的系统软件包等操作。

由于存在以 root 身份运行时存在的风险，即使在容器内运行，也可以在 OpenShift 集群中禁用以 root 身份运行任何容器的功能。因此，使用此构建策略的能力可能不会出现在您所使用的 OpenShift 集群中。

7.3 创建 Build

对于这个构建，您将使用与 Source 构建策略相同的源代码库，但直接使用 new-app 来运行 Docker 构建策略的构建。

这通过传递--strategy = docker 选项以及存储库的 URL 来完成：

```
$ oc new-build --name blog --strategy=docker \
https://github.com/openshift-katacoda/blog-django-py
--> Found Docker image 956e2bd (5 days old) from Docker Hub
for "centos/python-35-centos7:latest"
...
* An image stream will be created as "python-35-centos7:latest"
that will track the source image
* A Docker build using source code from
https://github.com/openshift-katacoda/blog-django-py will be
created
* The resulting image will be pushed to image stream "blog:latest"
* Every time "python-35-centos7:latest" changes a new build will
be triggered
--> Creating resources with label build=blog ...
imagestream "python-35-centos7" created
imagestream "blog" created
buildconfig "blog" created
--> Success
Build configuration "blog" created and build triggered.
Run 'oc logs -f bc/blog' to stream the build progress.
```

该版本在运行时将使用存储库中的 Dockerfile 作为如何构建映像的说明源。

7.4 部署镜像

生成完成后，生成的图像作为名为 blog 的图像流存储在项目中。要将图像作为应用程序部署，请使用 oc new-app 命令：

```
$ oc new-app blog
--> Found image 1f6debb (5 minutes old) in image stream "myproject/blog"
under tag "latest" for "blog"
...
* This image will be deployed in deployment config "blog"
```



```

* Port 8080/tcp will be load balanced by service "blog"
* Other containers can access this service through the
hostname "blog"
--> Creating resources ...
deploymentconfig "blog" created
service "blog" created
--> Success

Run 'oc status' to view your app.

```

在这种情况下，因为您继续将该映像作为应用程序进行部署，您可以使用 `oc new-app` 命令并直接从源文件转到部署的应用程序：

```

$ oc new-app --name blog --strategy = docker \
https://github.com/openshift-katacoda/blog-django-py

```

从 Dockerfile 创建 S2I 构建器映像时，您将使用 `oc new-build` 而不是 `oc new-app`。在这种情况下，最终的图像将无法作为应用程序进行部署。

7.5 构建和运行时间配置

与从容器映像部署应用程序时类似，运行容器时要设置的环境变量可以使用 `--env` 选项指定，以指定新应用程序。这可以在直接从源代码部署时，或者使用 `oc new-app` 部署由独立构建步骤创建的映像时完成。

如果需要为构建步骤设置环境变量，并且您正在从源代码直接部署，请使用 `--build-env` 选项来创建新应用程序：

```

$ oc new-app --name blog --strategy = docker \
--build-env UPGRADE_PIP_TO_LATEST = 1 \
https://github.com/openshift-katacoda/blog-django-py

```

如果使用调用 `oc new-build` 和 `oc new-app` 的两步方法，则应使用 `--env` 选项将特定于构建步骤的环境变量传递给 `oc new-build`：

```

$ oc new-build --name blog --strategy = docker \
--env UPGRADE_PIP_TO_LATEST = 1 \
https://github.com/openshift-katacoda/blog-django-py

```

如果在构建配置完成后需要添加环境变量创建时，可以使用 `oc set env` 命令：

```

$ oc set env bc/blog UPGRADE_PIP_TO_LATEST=1
buildconfig "blog" updated

```

在构建映像时可能需要构建时环境变量来自定义构建过程。您可以通过使用 `oc set env` 和 `--list` 选项来查看为构建设置的环境变量：

```

$ oc set env bc/blog --list
# buildconfigs blog
UPGRADE_PIP_TO_LATEST=1

```

虽然环境变量是在构建配置中指定的，但它们也将在创建的映像中设置，并且在部署时应用程序可见。

从 Dockerfile 构建图像时，也可以提供构建参数。

这些行为与环境变量相似，只是在映像运行时不会在容器中设置它们，仅在构建过程中设置。

使用 `oc new-build` 时，可以使用 `--build-arg` 选项设置构建参数：

```
$ oc new-build --name blog --strategy = docker \
--build-arg HTTP_PROXY = https://proxy.example.com \
https://github.com/openshift-katacoda/blog-django-py
```

要使用构建参数，必须在 `Dockerfile` 中定义适当的 `ARG` 指令；否则，他们将被忽略。

虽然在运行图像时不会在容器中设置与构建参数相对应的环境变量，但如果您能够直接从图像注册表访问图像，您将能够使用图像注册表查看与构建参数相关的值 `docker` 历史或 `docker` 检查。

7.6 使用内联 Dockerfile

可以通过创建从现有图像派生的新图像来自定义图像。然后添加其他层，其中包括更多软件包或配置。这可以使用 `Docker` 构建策略完成，`Dockerfile` 和相关文件托管在源代码库中。还可以使用二进制输入构建，使文件可以保存在本地文件系统中，并注入构建过程。

对于可以在 `Docker` 文件中描述所有步骤的简单映像自定义，不需要将其他文件与 `Dockerfile` 一起提供，`Dockerfile` 可以被定义为构建配置的一部分。当您需要安装使用 `S2I` 构建器构建的应用程序所需的其他操作系统软件包或实用程序时，可以使用此方法。

为了说明这种情况，使用其他步骤创建一个 `Dockerfile`：

```
FROM openshift/python:3.5
USER root
RUN yum install -y wget
USER 1001
```

然后，您可以使用 `Docker` 构建，使用 `Dockerfile` 作为输入，通过使用 `--dockerfile` 选项运行 `oc new-build` 来创建新映像：

```
$ cat Dockerfile | oc new-build --name python-plus --dockerfile=-
--> Found image a080357 (2 days old) in image stream "openshift/python" under
tag "3.5" for "openshift/python:3.5"
...
* A Docker build using a predefined Dockerfile will be created
* The resulting image will be pushed to image stream "python-plus:latest"
* Use 'start-build' to trigger a new build
--> Creating resources with label build=python-plus ...
imagestream "python-plus" created
buildconfig "python-plus" created
--> Success
Build configuration "python-plus" created and build triggered.
Run 'oc logs -f bc/python-plus' to stream the build progress.
```

`--dockerfile` 选项的参数表示通过管道从标准输入中读取 `Dockerfile` 内容 - 否则，需要将 `Dockerfile` 的内容作为选项的值提供。无法提供指定文件系统中 `Dockerfile` 位置的路径。

构建完成后，通过使用替代原始构建器名称创建的图像流的名称，可以使用自定义图像来代替原始 `S2I` 构建器图像。

7.7 总结

作为传统 PaaS 功能的扩展，可以通过平台上的应用程序源代码来构建令人担忧的细节，OpenShift 还支持从 Dockerfile 中的指令构建容器映像。

这为控制容器映像的构建和运行方式提供了最大的灵活性，并且可以将其他操作系统软件包安装到容器映像中。

因此构建策略要求构建以 Unix root 用户身份运行，所以可能不会为您正在使用的 OpenShift 集群启用它。

第 8 章.了解源到镜像构建器

Docker 构建策略提供了对如何构建映像的最佳控制。由于允许 Docker 在共享 OpenShift 集群中构建的潜在安全风险，使用 Docker 构建策略的能力通常仅限于受信任的开发人员。

因此，建设应用程序的最常用方法是使用 S2I 构建器。为了充分利用 S2I 的建设者，了解他们的工作方式很有帮助。

在本章中，您将深入了解 S2I 构建器如何工作以及如何实现简单的 S2I 构建器映像。您还将看到如何向自定义 S2I 构建器的图像添加注释，以便可以从 Open-Shift Web 控制台中选择和使用它。

8.1 Source-to-Image 项目

在 OpenShift 中从应用程序源代码构建图像的过程利用了源于开源 [Source-to-Image](#) 项目的工具包和 workflows。

S2I 工具包通过将源文件注入构建器基础映像的运行实例中，生成可立即运行的映像，并使用构建器映像中的脚本将该源代码转换为可运行的应用程序。从容器中运行构建过程，然后创建可运行的应用程序映像。

驱动 S2I 构建过程的命令行工具称为 s2i。当您使用 S2I 在 OpenShift 的源代码中部署应用程序时，所有涉及运行 s2i 命令行工具的步骤均可为您完成。但是，可以在您自己的系统上使用此工具创建独立于 OpenShift 的容器图像。

为了说明 S2I 构建过程如何工作，我们将直接使用 s2i 命令行工具。

8.2 构建应用程序镜像

要使用 s2i 命令行工具构建应用程序映像，需要两个输入。首先是您的应用程序源代码。第二个是支持您的应用程序实现的编程语言和服务堆栈的 S2I 构建器映像。

S2I 构建器映像是正常的容器映像，在 Docker Hub 映像注册表中有许多可用的映像。常用编程语言或应用程序服务器堆栈的图像示例如下：

Node.js 6 centos / nodejs-6-centos7

Ruby 2.3 centos / ruby-23-centos7

Perl 5.24 centos / perl-524-centos7

PHP 7.0 centos / php-70-centos7

Python 3.5 centos / python-35-centos7

Wildfly 10.1 openshift / wildfly-101-centos7

应用程序源代码可以托管在 Git 存储库托管服务上，也可以保存在本地文件系统目录中。

要使用 `s2i` 命令行工具创建可运行的容器映像，还需要运行本地容器服务。

要构建容器映像，请运行 `s2i build`，为其提供应用程序源代码的位置，`S2I` 构建器映像的名称以及要为创建的应用程序映像提供的名称：

```
$ s2i build https://github.com/openshift-katacoda/blog-django-py \
centos/python-35-centos7 blog
```

生成的图像的详细信息可以通过运行 `docker` 图像查看：

```
$ docker images
```

REPOSITORY	TAG	IMAGE ID	CREATED	SIZE
blog	latest	ec50f4d34a83	About a minute ago	675MB

使用 `s2i`，您可以创建应用程序映像，而无需知道如何创建 `Dockerfile`，也无需知道如何运行 `Docker` 构建。

要运行应用程序镜像，请使用 `docker run`：

```
$ docker run --rm -p 8080:8080 blog
```

尽管生成应用程序图像的过程看起来很神奇，但生成的图像并不特殊。您可以使用 `s2i` 在 `OpenShift` 之外构建图像，并使用本地容器服务运行它们，通过将它们推送到图像注册表或将它们部署到 `OpenShift` 来与其他人共享。

然而，在 `OpenShift` 中使用 `S2I` 支持的好处在于绑定到自动构建和部署机制。[第 11 章](#)将更多关注使用 `OpenShift` 自动构建和部署。

8.3 汇编源代码

当运行 `s2i build` 命令时，会执行多个步骤来创建应用程序映像。

第一步是将应用程序源代码打包到一个存档中。

在我们的例子中，`s2i build` 命令在 `Git` 存储库托管服务中提供了应用程序源代码的 `URL`。`s2i` 命令将下载最新版本的源代码。另一个版本可以通过提供标签名称，提交引用或分支名称来提名，而不是远程 `Git` 存储库，可以提供包含应用程序源代码的本地目录的路径。

第二步是 `s2i` 命令将使用 `docker run` 命令从 `S2I` 构建器映像启动容器。同时，它会将包含应用程序源代码的存档注入到容器中。

由 `s2i` 命令在容器中运行的命令将解压包含应用程序源代码的存档。然后运行由 `S2I` 构建器映像提供的汇编脚本。

正是这个汇编脚本需要应用程序源代码并将其移动到正确的位置，或编译源代码以生成可运行的应用程序。作为此构建过程的一部分，汇编脚本还可以下载应用程序运行所需的任何特定于语言的程序包，然后进行安装。

汇编脚本完成之后，执行将停止的容器的快照转换为可运行映像的第三步也是最后一步。完成此操作后，`s2i` 命令还会将图像运行时要执行的程序设置为由 `S2I` 构建器映像提供的运行脚本。

这是在 `docker run` 命令用于运行映像时执行的运行脚本。运行脚本将执行启动由汇编脚本准备的应用程序所需的任何步骤。

8.4 创建 S2I 生成器镜像

由于 S2I 构建器是容器映像，因此您可以创建自己的自定义构建器映像。这是通过从 Dockerfile 构建图像完成的。OpenShift 项目提供了一个方便的基础映像，您可以将其用作自定义 S2I 构建器的起点。

为了说明一个简单的 S2I 构建器，您可以创建一个运行 Web 服务器来构建静态文件的构建器。运行 S2I 进程时向构建器提供的源文件将成为 Web 服务器托管并提供的內容。

该构建器的 Dockerfile 将包含：

```
FROM openshift/base-centos7
LABEL io.k8s.description="Simple HTTP Server" \
io.k8s.display-name="Simple HTTP Server" \
io.openshift.s2i.scripts-url="image:///usr/libexec/s2i" \
io.openshift.expose-services="8080:http" \
io.openshift.tags="builder,http"
COPY assemble /usr/libexec/s2i/
COPY run /usr/libexec/s2i/
COPY usage /usr/libexec/s2i/
EXPOSE 8080
USER 1001
CMD [ "/usr/libexec/s2i/usage" ]
```

Dockerfile 需要将 io.openshift.s2i.scripts-url 标签定义为图像中汇编和运行脚本的位置，并将脚本复制到该位置。

io.openshift.expose-services 标签和 EXPOSE 语句一起告诉 OpenShift 构建器生成的应用程序将使用哪些端口。

Dockerfile 可以包含在生成构建器映像时安装附加包所需的任何 RUN 语句。

USER 语句必须在 Dockerfile 的末尾设置为用户 ID 1001，该用户 ID 将作为构建过程的运行用户。

映像中包含的使用脚本被配置为运行映像时执行的默认命令。它有以下内容：

```
#!/bin/bash
cat <<EOF
This is a S2I builder image for running a simple HTTP server.
For instructions on using the S2I builder image see:
* https://github.com/openshift-katacoda/simple-http-server
EOF
```

使用脚本作为用户查找有关如何使用构建器映像的信息的方式提供。

准备图像的主要工作由汇编脚本完成，如下所示：

```
#!/bin/bash
echo " -----> Move HTTP server files into place."
mv /tmp/src/* /opt/app-root/src/
```

汇编脚本应该将 s2i 构建过程中放置在 /tmp/src 目录中的任何源文件复制到所需的位置，或将它们编译为可执行应用程序。

汇编脚本还可以下载并安装应用程序所需的其他特定于语言的程序包，这些程序包列在源文件包含的程序包依赖关系文件中。

对于使用的 openshift / base-centos7 映像，运行构建和应用程序时的主目录设置为 / opt / app-root / src。应用程序可以写入 / opt / app-root 下的任何位置。

尽管在本例中不需要，但如果应用程序在运行时需要写入文件系统，则需要确保在运行任何自定义构建步骤后，修复 / opt / app-root 目录树上的权限。基础构建器映像为此提供修复权限脚本。运行时，修复权限脚本将确保作为参数传递的目录中的所有目录和文件都可由同一组中的任何用户写入。如果需要执行此步骤，请将其添加到汇编脚本的末尾：

```
fix-permissions /opt/app-root
```

运行脚本在 S2I 构建过程生成的应用程序映像运行时执行。它看起来像这样：

```
#!/bin/bash  
echo " -----> Run HTTP server."  
exec python -m SimpleHTTPServer 8080
```

你从这个脚本运行的应用程序应该在前台运行，执行它时应该使用 **exec** 语句。这确保它继承进程

容器的 ID 1，允许应用程序接收发送到容器的任何信号以触发关闭。

使用汇编脚本和运行脚本，如果脚本中的任何步骤都失败，则脚本作为整体应该立即失败。无需检查每个命令运行的结果，确保这一点的最简单方法是在每个脚本的开始处添加以下行：

```
set -eo pipefail
```

此命令指示 **bash** 解释程序在任何命令运行返回指示失败的结果时立即退出。

8.5 构建 S2I 生成器镜像

如果在 OpenShift 之外使用 **s2i** 命令行工具，要从这些文件构建 S2I 构建器映像，您应该运行：

```
$ docker build -t simple-http-server。
```

这将产生一个名为 **simple-http-server** 的图像。要使用它来构建应用程序图像，您可以运行：

```
$ s2i build \  
https://github.com/example/static-web-site \  
simple-http-server \  
static-web-site
```

此构建器中的运行脚本启动简单的 HTTP 服务器来托管复制到映像中的文件。要运行应用程序映像，请使用以下命令：

```
$ docker run --rm -p 8080: 8080 static-web-site
```

8.6 在 OpenShift 中使用 S2I Builder

要在 OpenShift 中使用您的 S2I 构建器图像，您有三个选项。首先是在 OpenShift 之外构建它，如前所示，并将其上传到 Docker Hub 等外部图像注册中心。此示例已作为 **openshiftkatacoda / simple-http-server** 放置在 Docker Hub 上。您可以使用以下命令将其导入 OpenShift：

```
$ oc import-image openshiftkatacoda / simple-http-server --confirm
```

这将创建一个名为 **simple-http-server** 的图像流，您可以通过运行以下命令来使用它：

```
$ oc new-app simple-http-server~https://github.com/example/static-web-site
```

第二种选择是，而不是将图像上传到外部图像注册表，将其推送到 OpenShift 的内部图像注册表。[第 5 章](#)介绍了如何做到这一点。

最后的选择是在 OpenShift 中构建 S2I 构建器映像。这个例子的源文件可以在 GitHub 上找到。要在 OpenShift 中构建它，您需要运行：

```
$ oc new-build --name simple-http-server --strategy=docker \
--code https://github.com/openshift-katacoda/simple-http-server
$ oc start-build simple-http-server
```

这将再次创建一个名为 simple-http-server 的图像流，如前所示。

8.7 将 S2I 生成器添加到目录

为了让您的 S2I 构建器可以在 Web 控制台的目录中进行选择，还需要为 S2I 构建器映像的图像流定义添加注释。这可以通过使用以下命令编辑图像流来完成：

```
$ oc edit / simple-http-server
```

应该添加带有值构建器的标签注释，得出：

```
{
  "kind": "ImageStream",
  "apiVersion": "v1",
  "metadata": {
    "name": "simple-http-server"
  },
  "spec": {
    "tags": [
      {
        "name": "latest",
        "annotations": {
          "tags": "builder"
        },
        "from": {
          "kind": "DockerImage",
          "name": "openshiftkatacoda/simple-http-server:latest"
        }
      }
    ]
  }
}
```

在编写本书时，必须为每个图像版本标签添加标签注释。有一种方案允许将其定义为图像流元数据中的注释，并且在您阅读本书时可能会受到支持。

在 Web 控制台的目录中，您现在可以搜索 simple-http-server 并使用浏览器中的构建器创建您的应用程序。

可以将其他注释添加到图像流定义中以提供显示名称，说明和类别。有关附加信息，请参阅关于为 S2I 构建器编写图像流的 OpenShift 文档。

如果您希望让其他人使用 S2I 构建器变得简单，并且图像托管在 Docker Hub 上，则可

以使用任何 Web 服务器托管图像流定义。然后用户可以使用 `oc create` 来加载图像流定义，而不是使用 `oc` 导入图像。对于此示例构建器，您可以运行：

```
$ oc create -f https://raw.githubusercontent.com \
/openshift-katacoda/simple-http-server/master/imagestream.json
```

8.8 总结

`Source-to-Image` 工具实现了一种机制来获取应用程序源代码并将其构建到容器图像中。该工具通过使用 `S2I` 构建器映像启动容器，将应用程序源代码注入容器并运行汇编脚本来设置映像内容来工作。

`S2I` 工具是一个独立的应用程序，您可以在自己的本地系统上使用，独立于将应用程序部署到容器的任何平台。为了便于使用，`OpenShift` 为该工具提供了集成支持，该工具构成了 `OpenShift` 的 `PaaS` 功能的核心。

`OpenShift` 提供了一系列您可以使用的 `S2I` 构建器，或者您可以轻松创建自己的自定义 `S2I` 构建器并将它们集成到 `OpenShift` 中，以便您可以从 `OpenShift Web` 控制台中选择它们。

第 9 章.自定义 Source-to-Image 构建

更典型的是，对于特定应用程序，您只需要对 `S2I` 构建器映像的行为进行较小的自定义设置而不是从头开始创建 `S2I` 构建器，。

设计良好的 `S2I` 构建器映像应该能够使用在构建期间或部署期间设置的环境变量来定制常见用例的行为。

当 `S2I` 构建器不执行此操作或您的用例需要运行其他步骤时，您将需要覆盖默认汇编和运行脚本的行为。

在本章中，您将了解 `S2I` 汇编和运行脚本的不同方式可以被覆盖或扩展。这可以通过在应用程序源代码中包含不同版本的脚本来完成;通过将它们托管在单独的 `Web` 服务器上并从构建配置引用它们;或者，如果您只希望覆盖运行脚本，请将其从配置映射或持久卷挂载到容器中。

您还将学习如何通过使用 `Source` 构建策略构建映像而不是 `Docker` 构建策略来创建修改后的 `S2I` 构建器。

9.1 使用环境变量

当使用现有的 `S2I` 构建器时，希望构建器的作者能够灵活设计它 - 也就是说，他们已经允许通过配置来定制 `S2I` 构建器的行为，无论是在构建器运行时创建应用程序映像以及该应用程序映像正在运行时。

可以使用环境变量提供构建器的配置。您在第 6 章中看到了如何为部署应用程序设置环境变量，或者何时使用 `S2I` 构建器构建应用程序。

在之前的例子中，环境变量被设置为 `OpenShift` 中构建和部署配置的一部分。在 `OpenShift` 中将环境变量定义为配置的一个问题是它与应用程序的源代码是分开的，而不是版本控制。

如果将这些环境变量与源代码包含在一起更有意义，那么在使用 `S2I` 构建器时，它们可

以作为源代码的一部分添加到文件.s2i / environment 中。

您可以创建.s2i / environment 文件并将其条目放置为 **key = value** 形式，而不是使用--build-env 选项来创建 new-app 或 oc new-build。例如：

```
UPGRADE_PIP_TO_LATEST = 1
```

只有静态值可以在这个文件中指定。无法为从其他值动态计算的环境变量或运行命令添加值。

在此文件中设置的环境变量将成为应用程序映像的一部分，并将用于构建应用程序映像以及部署应用程序映像。

9.2 覆盖 Builder 脚本

当运行 S2I 构建器以构建应用程序映像时，将运行映像中包含的汇编脚本以获取源文件并生成可运行应用程序。稍后运行应用程序映像时，映像中包含的运行脚本将用于运行应用程序。

通过在应用程序源代码的.s2i / bin 目录中提供您自己的脚本，可以重写汇编脚本和运行脚本。如果存在，这些将在 S2I 构建过程中被复制到容器中，组装脚本将在默认脚本的位置运行。同样，如果提供替换运行脚本，则将在应用程序映像运行时使用它。

尽管这些脚本可以是独立的，并可以完全替代原始脚本，但更典型的用例是执行某些操作，然后执行原始脚本。在汇编脚本的情况下，在执行原始汇编脚本之后，您可能执行其他操作。

添加为.s2i / bin / assemble 的自定义组装脚本将采用以下格式：

```
#!/bin/bash
set -eo pipefail
# Set environment variables.
# ...
# Execute original assemble script.
/usr/libexec/s2i/assemble
# Run additional build steps.
# ...
```

确保脚本是可执行的。作为一个 shell 脚本，您可以在其中包含任何 shell 代码，包括动态设置环境变量的代码。

请记住，它是原始的汇编脚本，它将复制源文件，安装其他软件包或将源文件编译为应用程序可执行文件。如果您在运行原始汇编脚本之前需要做的不仅仅是设置和导出环境变量，那么复制到源代码中的源文件将位于 /tmp / src 目录中。

如果您通过添加.s2i / bin / run 来覆盖运行脚本，它将采用以下形式：

```
#!/bin/bash
set -eo pipefail
# Set environment variables.
# ...
# Run additional deployment steps.
# ...
# Execute original run script.
exec /usr/libexec/s2i/run
```

运行原始运行脚本时，您必须使用 **exec**。这确保了原始脚本在容器中作为进程 ID 1 运

行，并且能够接收发送到容器的信号以关闭应用程序。

由于原始运行脚本取代了此脚本的执行，因此无法运行任何部署后步骤。如果您需要在部署新应用程序的新版本后运行操作，则应该查看使用生命周期挂钩。生命周期钩子将在[第 17 章](#)中详细介绍。

9.3 只读代码存储库

只有拥有原始源代码，或者应用程序源代码使用托管 Git 存储库并且已分叉原始代码存储库时，才能将组合和运行脚本添加到应用程序源文件。

如果您不能添加到原始源文件或不想，您可以将汇编和运行脚本托管在单独的代码库中，并设置构建配置以从该位置下载它们。为此，您需要编辑构建配置并将 `spec.strategy.sourceStrategy.scripts` 属性设置为可从中下载汇编和运行脚本的 Web 服务器上的目录的 URL。

这不能在使用 `oc new-app` 创建应用程序时或从 Web 控制台创建应用程序时完成，除非您要为构建和部署提供原始资源对象定义。

要为已经创建的构建配置设置此属性，可以使用 Web 控制台或 `oc` 编辑，并直接更改构建配置的 YAML / JSON 定义。

你也可以使用 `oc` 补丁从命令行编辑它：

```
$ oc patch bc/blog --type=json --patch \
'[{ "op": "add",
  "path": "/spec/strategy/sourceStrategy/scripts",
  "value": "https://raw.githubusercontent.com/example/test/master"}]'
```

如果使用此方法，并且应用程序源代码还提供了在 `.s2i / bin` 目录中组装和运行脚本，则应用程序源代码中的文件将被忽略，并且不会被复制到镜像中。这意味着使用 URL 从构建配置中指定的脚本版本无法调用应用程序源代码中存在的现有汇编和运行脚本。但是，这些脚本仍然可以执行 S2I 构建器映像提供的原始汇编和运行脚本。

9.4 覆盖运行时镜像

先前描述的方法会覆盖应用程序映像构建时的汇编和运行脚本。如果您有现有映像并需要覆盖运行脚本，但不想重新构建映像，则可以覆盖容器启动时使用的命令。这可以通过编辑从映像部署的应用程序的部署配置来完成。

在这种情况下，替换运行脚本需要存储在持久卷中，使用配置映射项目作为文件挂载到容器中，或者使用 `init` 容器放置在临时卷中。如果使用配置图，请先创建它：

```
$ oc create configmap blog-run-script --from-file = run
```

接下来，将配置映射挂载到容器中：

```
$ oc set volume dc/blog --add --type=configmap \
--configmap-name=blog-run-script \
--mount-path=/opt/app-root/scripts
```

然后更新部署配置以在启动容器时执行此脚本：

```
$ oc patch dc/blog --type=json --patch \
'[{ "op": "add",
```

```
"path":"/spec/template/spec/containers/0/command",  
"value":["bash","/opt/app-root/scripts/run"]}]'
```

由于您使用的是配置映射，因此运行脚本不会被标记为可执行文件。因此有必要使用 `bash` 来执行运行脚本。

如果使用挂载的持久性卷，请使用 `oc rsync` 将运行脚本复制到持久性卷中，或使用 **Web 控制台** 或 `oc rsh` 在容器中创建交互式终端会话并编辑运行脚本。将运行脚本存储在持久卷中时，可以将其标记为可执行文件，并将其作为部署配置中的命令直接执行。

要使用 `init` 容器，你需要在 `init` 容器中挂载一个类型为 `emptyDir` 的卷，并在其中放置运行脚本。主应用程序容器将安装相同的卷并从中运行脚本。您可以在[第 17 章](#)中找到有关使用卷的更多信息，以及[第 14 章](#)中的 `init` 容器。

9.5 更新镜像元数据

在这里描述的情况下，汇编脚本和运行脚本被新版本覆盖，这些新版本位于应用程序源代码的 `.s2i / bin` 目录中，或者通过指定可从中下载脚本的 **Web 服务器** 来提供，这些新版本仅适用于影响正在创建的特定图像。如果要从 `S2I` 构建过程创建映像并尝试将其用作 `S2I` 构建器映像，那么后一个构建将恢复为使用第一个构建器映像中提供的原始脚本。为了使用 `S2I` 构建创建一个行为不同的新 `S2I` 构建器映像，必须覆盖所创建映像的元数据。这可以通过创建文件 `/tmp/.s2i/image_metadata.json` 并覆盖映像的标签来指定汇编脚本和运行脚本的位置，从汇编脚本完成。从 `Dockerfile` 中[第 8 章](#)创建的 `simple-http-server` 映像的源文件开始，添加文件 `.s2i / bin / assemble`，其中包含：

```
#!/bin/bash  
set -eo pipefail  
# Move assemble/run scripts to new location.  
mkdir /opt/app-root/s2i  
mv /tmp/src/* /opt/app-root/s2i  
# Override image metadata for builder image.  
mkdir -p /tmp/.s2i  
cat > /tmp/.s2i/image_metadata.json << EOF  
{  
  "labels": [  
    {"io.openshift.s2i.scripts-url": "image:///opt/app-root/s2i"}  
  ]  
}  
EOF
```

该脚本将汇编，运行和使用脚本从以前的版本移动到映像的 `/opt / approot / s2i` 目录中。然后创建 `/tmp/.s2i/image_metadata.json` 文件，将 `io.openshift.s2i.scripts-url` 标签设置为该目录。

当在这些源文件上运行 `S2I` 构建过程时，`image_metadata.json` 文件中的标签信息将用于更新所创建的最终图像上的标签。

除 `.s2i / bin / assemble` 脚本外，我们还添加 `.s2i / bin / run`。在此我们添加：

```
#!/bin/bash  
exec /opt/app-root/s2i/run
```

这将被设置为图像运行时使用的CMD。如果遵循S2I生成器的正常模式，则会调用/opt / approot / s2i / usage。在这种情况下，运行脚本正在被调用。这意味着生成的图像既可以作为S2I构建器运行，也可以作为独立的应用程序映像运行，并将静态文件从持久卷装载到容器中。

要构建自定义S2I图像，您可以运行：

```
$ oc new-build --strategy=source --name simple-http-server \
--code https://github.com/openshift-katacoda/simple-http-server \
--image-stream python:2.7
```

使用S2I构建器并以这种方式更新图像元数据可让您创建新的S2I构建器映像，而无需从Dockerfile构建映像。

唯一的限制是构建以非root用户身份运行，因此无法安装其他操作系统软件包。

9.6 总结

“Source-to-Image”构建器通常可以通过将环境变量传递到构建或随后部署创建的应用程序映像来进行配置。

如果您需要对构建进行更多控制，则可以通过在应用程序源代码中提供自己的脚本来重写汇编和运行脚本。对于使用S2I创建的应用程序的特定部署，也可以从部署配置中覆盖运行脚本。

当为应用程序源代码提供汇编脚本时，可以更新所创建的容器图像的图像元数据，以便如果创建的图像用作S2I构建器，它将使用与原始图像不同的一组S2I脚本。

以这种方式更新映像元数据可以创建自己的自定义S2I构建器映像，而无需使用Docker构建策略，而是使用源构建策略。

第10章.使用增量和链接构建

使用“Source-to-Image”构建器时，构建阶段由汇编脚本在一个步骤中执行。与此有关的一个问题是每个构建都是不同的。这意味着由一个构建生成的构建工件不能用于后续构建。由于每次都需要重新创建，这会减慢构建时间。

部分原因可能是速度慢，对于每个版本，都需要通过互联网从远程软件包存储库中下载第三方软件包。加速这一点的一种方法是部署本地高速缓存代理服务器，通过该服务器路由下载。软件包的下载将被加速，因为软件包可以由缓存提供，从而避免每次都需要从远程软件包索引下载它。

然而，使用本地缓存代理服务器不会消除重新编译仅作为源代码提供的包的需求，除非缓存还支持将预编译包上传到构建过程可以使用的缓存。

使用本地高速缓存代理服务器的另一种方法是在OpenShift中保存来自构建的构建工件，并将它们复制到后续构建中，以便可以重用它们。在本章中，您将学习如何通过重用先前构建的构建工件来使用增量式和链式构建来加快构建时间。

10.1 使用缓存快速构建

当您在自己的计算机系统上本地开发和部署应用程序时，您可以通过重复使用以前版

本的应用程序中的工件来缩短构建时间。

例如，您不需要在每次更改后部署应用程序时下载并安装应用程序需要的所有第三方软件包。对于已编译的编程语言，构建工具通常足够聪明，可以知道某些代码文件不需要重新编译，因为代码更改不会影响它们。

但是，在使用 S2I 构建器时，每次构建应用程序可能需要的所有内容都必须完成。这是因为每个构建都在一个新的容器中运行，只从构建器基础映像开始。

在需要从互联网下载第三方软件包的情况下，通过使用本地高速缓存代理服务器可以节省一些时间。根据所使用语言的打包系统，这可能是一个普通的缓存 HTTP 代理，也可能是一个特殊用途的代理缓存。对于 Java，这可能是 Nexus 或 Artifactory，对于 Python，它可能是 Python Package Index (PyPi) 镜像或代理，如 devpi-server。但是，这种本地缓存代理可能不会消除每次重新编译包的代码的需要。

OpenShift 提供了两种机制，旨在减少运行应用程序的 S2I 构建所花费的时间。这些可用于实现自定义缓存解决方案，从一个构建到下一个构建实现构建工件的遗留或为预编译构建工件提供本地缓存服务器。这两个机制是增量构建和链式构建。

10.2 使用增量构建

在 [第 8 章](#) 中，您了解了 S2I 构建过程的工作原理。该过程的第一步是使用构建器映像启动容器，并将应用程序源代码注入正在运行的容器中。增量构建的基础是，除了应用程序源代码之外，还可以同时将一组从前一版本恢复的构建工件注入到容器中。

对于支持增量构建的 S2I 构建器，它必须满足三个要求。

其中第一个原因是它不能丢弃可以转移到后续版本的构建工件。这可能是一个问题，因为构建应用程序映像时的一个目标是使映像尽可能小。通常可以用来加速后续构建的中间构建工件将被丢弃。

第二个是 S2I 构建器必须提供一个脚本来从先前构建的图像中提取构建工件。

第三个也是最后一个要求是，S2I 构建器的汇编脚本必须知道要查找和使用注入到容器中的任何构建工件以用于新构建。

10.3 从构建中保存开发

为了从构建生成的以前的图像中提取构建工件，S2I 构建器必须提供一个保存构件脚本。这应该放在与汇编和运行脚本相同的目录中。通用保存工件脚本的示例如下所示：

```
#!/bin/bash
mkdir -p /opt/app-root/cache
echo "Incremental Build" > /opt/app-root/cache/README.txt
tar -C /opt/app-root -cf - cache save-artifacts
```

脚本的输出应该是构建工件的 tar 归档流。在这个例子中，脚本希望将任何构建工件都转移到后续的构建中，这些构件已经被组脚本放置到目录 `/opt/app-root/cache` 中。

如果 `/opt/app-root/cache` 目录不存在，则脚本首先创建它。这样做是为了确保我们总是从脚本中返回一些东西。 `README.txt` 文件是作为标记文件创建的，因为如果它包含的是没有文件的空目录，S2I 构建过程会丢弃输出。

10.4 恢复构建开发

当运行 S2I 构建过程并启用增量构建支持时，将从前一构建生成的映像启动临时容器。在此临时容器中运行的命令将是保存工件脚本。输出将被捕获并注入到新版本的容器中，以及应用程序源代码。构建工件将被放置在目录 `/tmp/artifacts` 中的容器中。要恢复构建工件，汇编脚本将检查 `/tmp/artifacts` 下高速缓存目录的存在并将其移动到后续构建步骤可以使用它的位置：

```
if [ -d /tmp/artifacts/cache ]; then
    echo " ----> Restoring cache directory from incremental build."
    mv /tmp/artifacts/cache /opt/app-root/
fi
```

如果 S2I 构建器通常丢弃构建工件，则保存构件脚本将无法保存。在这种情况下，汇编脚本可以使用 `/tmp/artifacts` 目录的存在来启用构建工件的保留。

10.5 启用增量构建

要为现有构建配置启用增量构建，必须将构建配置的 `spec.strategy.sourceStrategy.incremental` 属性设置为 `true`。

要在第 6 章的示例博客网站应用程序上启用增量构建，可以使用 `oc patch` 命令：

```
$ oc patch bc/blog --type=json --patch \
'[{ "op": "add",
  "path": "/spec/strategy/sourceStrategy/incremental",
  "value": true } ]'
```

然后应该触发一个新的构建：

```
$ oc start-build blog
```

在博客应用程序中，使用了 Python S2I 构建器，它不支持增量构建。增量构建支持是通过向应用程序源代码添加合适的保存构件和组装脚本包装来添加的。

由于 Python S2I 构建器在下载和构建时通常不会缓存第三方包，因此有必要启用它。这是通过在第一次增量构建完成时触发创建缓存目录来完成的。结果是第一个增量构建仍然花费相同的时间，因为没有使用缓存的构件。只有在后续的构建中才实现了加速：

```
$ oc get builds
NAME TYPE FROM STATUS STARTED DURATION
blog-1 Source Git@832a277 Complete 5 minutes ago 38s
blog-2 Source Git@832a277 Complete 4 minutes ago 41s
blog-3 Source Git@832a277 Complete 3 minutes ago 23s
blog-4 Source Git@832a277 Complete 2 minutes ago 24s
```

在使用缓存构建工件的第一个构建是 `blog-3` 之后，在 `blog-2` 之前启用了增量构建之后。当构建工件可用时，构建时间几乎减半。您将看到多少改进取决于正在使用的 S2I 构建器，应用程序以及跨构建缓存的构建工件类型。

因为增量构建依赖于包括支持的 S2I 构建器组装脚本，所以如果在未添加支持时启用增量构建，则将看不到任何益处。如果使用 OpenShift 附带的一个 S2I 构建器或第三方的 S2I 构建器，请确保检查任何文档以查看是否包含增量构建支持以及是否需要更改应用程序代码以使其可用。

10.6 使用链式构建

链式构建与增量构建类似，即从现有映像复制文件。然而，使用链式构建时有两个关键的区别。

第一个是，不是从同一图像的以前版本复制文件，而是从单独的图像复制它们。这可以是导入到群集中的映像或已在群集中构建的映像。也可以同时复制来自多个图像的文件。

第二种方法是，将文件复制到与应用程序源文件分开的映像中的固定位置，而不是将它们与应用程序源文件合并到您选择的位置。

使用链式构建可以设置单独的构建配置，以将构件预先构建到图像中，然后在多个应用程序图像中使用该构件。链式构建还允许使用特殊的构建时图像来生成可执行应用程序，然后将可执行应用程序复制到较小的运行时映像中，而无需构建工具和编译器。

我们一直用于例子的博客应用程序是用 Python 编程语言实现的。Python 中用于加速构建时间的一种机制是将 Python 包预编译为 Python 轮子。这些文件包含已安装版本的 Python 软件包，包括任何已编译的 C 扩展模块。

尽管 Python 轮子可以上传到 PyPi，但是当使用 C 扩展模块时，程序包可能无法为您的平台提供它们。

要下载并预编译一组软件包到 Python 轮子，可以使用 S2I 构建器创建一个 Python 控制台。

运行我们的博客应用程序的 Python 操舵室 S2I 生成器可以通过运行：

```
$ oc import-image openshiftkatacoda/python-wheelhouse --confirm  
This can then be used to prebuild the Python wheels into an image:  
$ oc new-build --name blog-wheelhouse \  
--image-stream python-wheelhouse \  
--code https://github.com/openshift-katacoda/blog-django-py
```

在这种情况下，我们使用与博客应用程序源代码所在位置相同的源代码库。从存储库中使用的所有内容都是来自 requirements.txt 文件的软件包列表。我们可以使用一个单独的源代码库来构建一套更大的软件包，而不仅仅是博客应用程序所需的软件包。

这可以用来代替本地代理服务器或缓存，如 devpserver。

要创建链式构建，我们需要使用 oc new-build 来首先为应用程序创建构建。--source-image 选项指定包含预构建工件的图像的名称。--source-image-path 选项指示源图像中的哪个目录在构建时应该复制到应用程序图像中，以及应用程序源代码中的相对位置：

```
$ oc new-build --name blog \  
--image-stream python:3.5 \  
--code https://github.com/openshift-katacoda/blog-django-py \  
--source-image blog-wheelhouse \  
--source-image-path /opt/app-root/wheelhouse/..s2i/wheelhouse
```

构建完成后，然后部署映像：

```
$ oc new-app --image-stream blog  
oc expose svc/blog
```

在使用链式构建时，用于构建应用程序映像的 S2I 构建器（或应用程序源代码中的自定义组装脚本）需要支持复制的预构建构件。对于博客应用程序，Python S2I 构建器不提供对此的任何支持以及自定义组装脚本处理设置配置。

现在，将使用从操舵室图像复制的 Python 轮子，而不是下载和重新编译 Python 程序包。

当使用链式构建而不是增量构建时，博客应用程序的结果是构建时间的立即加速：

```
$ oc get builds
```

NAME	TYPE	FROM	STATUS	STARTED	DURATION
blog-1	Source	Git@04599b1	Complete 3	minutes ago	24s
blog-2	Source	Git@04599b1	Complete 2	minutes ago	22s

只有当需要修改 Python 包列表或需要更新版本的包时，才需要重新构建博客应用程序使用的操舵室图像。

如果操舵室图像更新，则在构建配置中定义的图像触发器将自动导致重建博客应用程序。

10.7 总结

由于 Source-to-Image 构建器每次都会启动，因此构建应用程序映像可能会很慢。这可能是由于需要从外部软件包存储库下载第三方软件包，或者每次构建时都要编译源代码，即使源代码没有改变。

为了加速 S2I 构建，S2I 构建者可以实现对增量或链接构建的支持。这些允许来自先前的图像构建的预编译的构件或来自完全独立的构建的构件在新构建中使用。

第 11 章. Webhooks 和构建自动化

OpenShift 为 Docker 和 Source 构建策略提供的支持简化了构建和部署过程，因为 OpenShift 会为您提供运行步骤的详细信息。

当源代码包含在托管 Git 存储库中时，当 S2I 构建器映像或用于 Docker 构建的基础映像已更新时，OpenShift 还可以自动触发重建和新部署。这是因为它可以在需要时从 Git 存储库中提取上次使用的源代码。

通过您迄今为止学到的知识，无论您何时更改源代码，您仍然需要手动触发具有最新源代码的新版本。

为了使您的开发工作流程完全自动化，在本章中，您将学习如何使用 webhook 将您的 Git 存储库链接到 OpenShift。这将允许您在代码更改时自动启动新构建，每次提交并将这些更改推送到您的 Git 存储库。

因为您可能会希望使用私有托管 Git 存储库，只能通过首先提供适当的访问凭据来访问源代码，本章还将介绍如何将 OpenShift 与私有托管 Git 存储库一起使用。

11.1 使用 Hosted Git Repository

您已经了解了如何使用 Web 控制台或 oc 命令行工具从源代码设置应用程序的构建和部署。构建可以使用 Source-to-Image 构建器，或者可以从 Dockerfile 构建应用程序。

您现在还可以从 Web 控制台中的构建配置描述或通过从命令行运行 oc start-build 命令来手动触发新构建的应用程序。

除了使用二进制输入源构建外，应用程序源代码将从单独的 Git 存储库托管服务中的主副本或分支下拉。当 OpenShift 以这种方式拉取应用程序源代码时，这意味着它可以在任何时候重建，而无需用户执行任何操作。这使我们能够开始自动执行端到端的构建和部署过程。

11.2 访问私有 Git 存储库

迄今为止的示例使用了公用 Git 存储库中托管的应用程序源代码。对于您自己的应用程序，您可能需要使用私人 Git 存储库。

使用私有 Git 存储库时，使用 SSH 或 HTTPS 连接可以实现安全访问，并且您需要提供访问凭据。

由于访问凭证是您的 Git 存储库的关键，并且需要存储在 OpenShift 中，因此使用具有有限权限的用户帐户的凭证非常重要，该权限可用于 Git 存储库，并且不会用于任何其他目的。对于 SSH 连接，您不应使用您的主要身份密钥。

要在访问存储库时创建供 OpenShift 使用的单独 SSH 密钥对，请运行 `ssh-keygen` 命令。将结果保存到一个名为它将用于的存储库的文件中：

```
$ ssh-keygen -f ~/.ssh/github-blog-sshauth
```

当被要求提供密码时，由于 OpenShift 无法使用需要密码的 SSH 密钥，请将其保留为空。

`ssh-keygen` 命令将创建两个文件。私钥文件将使用提供的名称，并且公钥文件具有相同的基本名称，但在末尾添加了 `.pub` 扩展名。

您应该配置您的 Git 存储库托管服务以使用公钥。如果使用 GitHub 或 GitLab，这称为部署密钥。如果使用 Bitbucket，它被称为访问密钥。确保何时添加授予它的密钥对存储库的只读访问权。这意味着 OpenShift 将能够下拉源代码来完成构建，但不能更改托管的 Git 存储库。

使用公钥设置 Git 存储库托管服务后，您就可以将私钥添加到 OpenShift 中，并配置 OpenShift 以将其用于您的应用程序。

在 OpenShift 中创建一个秘密来保存私钥：

```
$ oc secrets new-sshauth github-blog-sshauth \  
--ssh-privatekey=$HOME/.ssh/github-blog-sshauth
```

授予构建器服务帐户访问秘密的权限，以便在下拉源代码以构建应用程序时使用它：

```
$ oc secrets link builder github-blog-sshauth
```

最后，为秘密添加一个注释以标识它的源代码存储库：

```
$ oc annotate secret/github-blog-sshauth \  
'build.openshift.io/source-secret-match-uri-1=\  
ssh://github.com/openshift-katacoda/blog-django-py.git'
```

要创建应用程序，请使用 SSH URI 而不是 HTTPS URI：

```
$ oc new-app --name blog --image-stream python \  
--code git@github.com: openshift-katacoda / blog-django-py.git
```

由于注释已添加到秘密中，因此构建会自动将存储库的使用与该秘密关联，并在下拉应用程序源代码时使用访问凭据。

如果正在使用的 Git 存储库托管服务不支持设置 SSH 访问密钥，则可以使用基于 HTTPS 的基本身份验证。如果 Git 存储库托管服务支持，则可以使用应用程序访问令牌。

如果您打算使用 HTTPS 连接与您的 Git 存储库交互，为您的凭证创建一个秘密，请使用以下命令：

```
$ oc secrets new-basicauth github-blog-basicauth --username <username> --prompt
```

将 `<username>` 替换为访问 Git 存储库时使用的用户名。 `--prompt` 选项确保提示您输入用户密码或应用程序访问令牌。

授予构建器服务帐户对秘密的访问权限，并使用要访问的存储库的详细信息对其进行注释。

向秘密添加注释可以将访问凭证自动链接到构建。如有必要，您还可以编辑现有的构建配置，并将 `spec.source.sourceSecret.name` 属性设置为包含访问凭证的秘密的名称。

如果使用 OpenShift 3.7 或更高版本，则可以在运行时将私有 Git 存储库的源秘密的名称提供给 `oc new-app` 或 `oc new-build` 命令，而不是将注释添加到秘密中 `---source-secret` 选项。

11.3 添加 Webhook 存储库

webhook（也称为用户定义的 HTTP 回调）是应用程序通知其他应用程序发生变化的一种方式。

当一组新的源代码更改被推送到托管的 Git 存储库时，所有主要的 Git 存储库托管服务都支持通过 webhook 生成回调。OpenShift 通过 webhook 接受回调作为触发新构建的手段。

通过配置 Git 存储库托管服务以及回调的 URL OpenShift 接受的详细信息，您可以完全自动化您的开发工作流程，从而将提交并推送回托管的 Git 存储库将启动新的构建和部署。

webhook 只能用于 Git 存储库托管服务可见的 OpenShift 集群。在使用 Minishift 或 oc 集群时不能使用 webhook，因为这些只能从运行它们的计算机系统中看到。

要确定 webhook 回调的 URL，请在构建配置上运行 `oc describe` 命令：

```
$ oc describe bc/blog
Name: blog
Namespace: myproject
Created: 29 hours ago
Labels: app=blog
Annotations: openshift.io/generated-by=OpenShiftNewApp
Latest Version: 1
Strategy: Source
URL: https://github.com/openshift-katacoda/blog-django-py
From Image: ImageStreamTag openshift/python:3.5
Output to: ImageStreamTag blog:latest
Build Run Policy: Serial
Triggered by: Config, ImageChange
Webhook GitHub:
URL: https://api.pro-us-east-1.openshift.com:443/oapi/v1/namespaces/
myproject/buildconfigs/blog/webhooks/3dhkEZGRIHD18XKbK_0e/github
Build Status Duration Creation Time
blog-1 complete 47s 2017-10-23 16:23:47 +1100 AEDT
Events: <none>
```

不同的 URL 用于 GitHub，GitLab 和 Bitbucket。如果您需要使用新的秘密令牌重新生成 webhook URL，或者您所使用的 Git 托管服务没有显示 webhook URL，则可以使用生成配置名称作为参数运行命令 `oc set 触发器`，然后通过选项 `- fromgithub`，`- from-gitlab` 或 `--from-bitbucket`（如适用）。重新运行 `oc describe` 构建配置以查看新的 webhook URL。

在配置 Git 存储库托管服务时使用 URL 来在您的 Git 存储库发生推送事件时触发 webhook。将 webhook 回调添加到 Git 存储库托管服务时，请确保生成的 HTTP 回调的内容类型是 `application / json`。

11.4 定制构建触发器

除了支持主要 Git 存储库托管服务的特殊 `webhook` 回调外，还可以使用通用 `webhook` 回调来触发新的构建。

这可以被任何能够发出 Web 请求的服务使用。例如，可以通过成功完成由第三方在应用程序源代码上运行的测试来触发回调。

换句话说，Git 仓库托管服务不是触发 OpenShift 运行构建，而是通知第三方测试服务首先对代码运行测试。如果测试通过，OpenShift 将继续构建和部署代码。

通用 `webhook` 的 URL 可以通过在生成配置上运行 `oc` 描述找到。如果尚未为构建配置设置通用 `webhook` URL，请使用 `--from-webhook` 选项在构建配置上运行 `oc set` 触发器。

当您使用通用 `webhook` 时，还支持将其他构建时环境变量作为 `webhook` 请求主体的一部分传递。这种方法可以用来定制应用程序的构建方式。有关传递环境变量的更多详细信息，请参阅有关通用 `webhook` 的 OpenShift 文档。

[第 6 章](#)研究了将测试集成到工作流中的另一种方法，即在应用程序映像生成后运行测试。只有那些测试通过，构建才会被标记为成功。

11.5 总结

OpenShift 基于 Kubernetes 构建一个实现 PaaS 层的构建系统。

这个构建系统可用于使用 `Sourceto-Image` 构建器或 `Dockerfile` 中提供的一组指令从源代码构建应用程序。

从应用程序源代码构建时，可以从托管的 Git 存储库中提取文件。这可能是一个公共或私人的 Git 存储库。构建的输入也可以从您自己的系统上的本地目录上传一次。

当使用托管 Git 存储库时，只要将代码更改推送到 Git 存储库，就可以配置 `webhook` 回调。这可以用来在 OpenShift 中触发新应用程序的构建和部署。如果使用单独的测试服务器，`webhook` 可能会触发测试套件首先运行，测试的成功将使用单独的 `webhook` 触发构建和部署。

第 12 章配置和隐私

开发应用程序时的最佳做法是将源代码保存在 Git 等版本控制系统中。这可确保您可以回滚到以前版本的源代码。尽管默认的配置设置最好也保存在使用应用程序源代码的版本控制下，但是应该将诸如数据库访问凭证或 SSH 密钥等秘密信息分开存储。这些秘密信息以及用于自定义应用程序默认值的单独配置设置不属于应用程序源代码的一部分，但在应用程序在容器中运行时需要提供给应用程序。为了实现这一点，Open-Shift 可以将配置和秘密存储为项目中的资源，并在部署发生时将它们添加到容器中。

存储单独配置的最简单方法是添加要在容器中设置的环境变量的定义，作为构建或部署配置的一部分。

以前的章节展示了以这种方式设置环境变量的示例，其中展示了从现有图像和源代码中部署应用程序。

在本章中，您将学习更多关于使用环境变量以及秘密的存储和使用。这将包括如何使用配置映射使配置能够在一个地方定义并在多个构建或部署配置中引用以避免重复。除了

将配置和秘密作为环境变量使用之外，如何将这些配置和秘密映射到容器中以使它们可用作文件也将被覆盖。

12.1 传递环境变量

从源代码构建映像时或在运行容器的任何情况下，都可以定义环境变量。

查询，添加或删除构建或部署配置中的环境变量由 `oc set env` 命令处理。

要列出环境变量的名称和值，请运行 `oc set env` 命令，将资源对象的名称作为参数传递并使用 `--list` 选项：

```
$ oc set env dc/blog --list
# deploymentconfigs blog, container blog
DATABASE_URL=postgres://user145c30ca:EbAYDR1sJsvW@blog-db:5432/blog
```

要添加新的环境变量传递环境变量及其值，通过的;分离的名称：

```
$ oc set env dc/blog BLOG_BANNER_COLOR=blue
```

如果环境变量已经设置，这将覆盖任何现有的值。如果环境变量存在但值不同，则可以使用 `--overwrite = false` 使更新失败。

要同时设置多个环境变量，请逐一列出它们之间的空格：

```
$ oc set env dc/blog BLOG_BANNER_COLOR=blue BLOG_SITE_NAME="My Blog"
```

如果您的环境变量要在文件中设置，或者想要从本地环境中设置它们，可以将它们传送到 `oc set env` 命令中，传递 `-` 以表明它应该从管道读取它们：

```
$ env | grep '^ AWS_' | oc set env dc / blog -
```

无论何时您设置环境变量的值，如果您需要从已设置的其他环境变量中编写值，则可以在该值中使用 `$(<VARNAME>)`。在从命令行设置时，确保使用单引号括住参数，以避免本地 shell 试图解释该值：

```
$ oc set env dc/blog \
DATABASE_USERNAME=user145c30ca \
DATABASE_PASSWORD=EbAYDR1sJsvW \
DATABASE_URL='postgres://$(DATABASE_USERNAME):$(DATABASE_PASSWORD)@blog-db:5432/blog'
```

如果需要值包含 `$(<VARNAME>)` 形式的文字字符串，请使用 `$$(<VARNAME>)` 来防止解释它。结果将以 `$(<VARNAME>)` 的形式传递。

要删除一个环境变量，而不是使用变量名称，后跟 `=` 和值，请使用变量名称，后跟 `-`：

```
$ oc set env dc / blog BLOG_BANNER_COLOR-
```

当更新部署配置，默认情况下这些命令将被应用到 `pod` 定义中的所有容器。如果您只想将操作应用于单个容器，则可以使用 `--container` 选项来命名该容器。

12.2 使用配置文件

环境变量是用于将配置信息注入容器的最简单的机制。但是，使用环境变量传递的配置仅限于简单键/值对的形式。此方法不适合将更复杂的结构化数据传递给应用程序，如 JSON，YAML，XML 或 INI 格式的配置文件。

为了处理更复杂的数据，OpenShift 提供了 `configmap` 资源类型。这也提供了存储键控数据的功能，但数据值可能更复杂。除了可以通过环境变量传递到容器之外，配置映射中

保存的配置可以作为文件在容器中提供，使得这种方法适合与传统配置文件一起使用。

要创建一个配置映射，您可以使用 `oc` 创建 `configmap`，或者使用 JSON / YAML 资源定义为配置映射创建 `oc` 创建。

如果您只需要存储简单的键/值对，则可以通过运行 `oc create configmap` 并传递 `--from-literal` 选项以及设置的名称和值来创建配置映射：

```
oc create configmap blog-settings \  
--from-literal BLOG_BANNER_COLOR=blue \  
--from-literal BLOG_SITE_NAME="My Blog"
```

您可以通过查询使用 `oc get -o json` 创建的资源来查看配置映射的定义。需要重现它的定义的关键部分是：

```
{  
  "apiVersion": "v1",  
  "kind": "ConfigMap",  
  "metadata": {  
    "name": "blog-settings"  
  },  
  "data": {  
    "BLOG_BANNER_COLOR": "blue",  
    "BLOG_SITE_NAME": "My Blog"  
  }  
}
```

如果您要采用此定义并将其另存为 `blog-settings-configmap.json`，则还可以通过运行以下命令来加载它：

```
$ oc create -f blog-settings-configmap.json
```

创建配置图时，它不与任何应用程序关联。要在部署配置中将此配置映射中的设置作为环境变量传递，您需要运行额外的步骤：

```
$ oc set env dc / blog --from configmap / blog-settings
```

使用 `--list` 选项运行 `oc set env` 对部署配置的结果现在将是：

```
# deploymentconfigs blog, container blog  
# BLOG_BANNER_COLOR from configmap blog-settings, key BLOG_BANNER_COLOR  
# BLOG_SITE_NAME from configmap blog-settings, key BLOG_SITE_NAME
```

环境变量将像以前一样设置，但不是存储在部署配置中，而是从配置映射引用它们。当容器启动时，每个环境变量的值将从配置映射中复制。

您可以将相同的配置图与任何需要它的应用程序相关联。这意味着您可以使用配置映射将配置保留在一个位置，而不需要在每个构建或部署配置中复制配置。对于更复杂的数据，您可以使用文件作为输入来创建配置图。创建一个名为 `blog.json` 的文件，其中包含：

```
{  
  "BLOG_BANNER_COLOR": "blue",  
  "BLOG_SITE_NAME": "My Blog"  
}
```

要创建配置图，而不是使用 `--from-literal`，请使用 `--from-file`：

```
$ oc create configmap blog-settings-file --from-file blog.json
```

运行 `oc` 描述在创建的配置图上，结果是：

```
$ oc describe configmap/blog-settings-file
```

```

Name:      blog-settings-file
Namespace:  myproject
Labels:     <none>
Annotations: <none>
Data
====
blog.json:
----
{
  "BLOG_BANNER_COLOR": "blue",
  "BLOG_SITE_NAME": "My Blog"
}

```

该文件的名称被用作密钥。如果密钥与用作输入的文件名称不同，您可以使用<key> = blog.json 作为--from-file 选项的参数，将密钥替换为您想要的名称使用。要将配置映射作为一组文件挂载到容器中，请运行：

```

$ oc set volume dc/blog --add --configmap-name blog-settings-file \
  --mount-path=/opt/app-root/src/settings

```

这将导致在由--mount-path 选项指定的目录中创建文件，其中创建的文件名称与键相对应，文件的内容是与这些键相关的值。

如果由--mount-path 指定的路径是包含现有文件的目录，则这些文件将从视图中隐藏并且不再可访问。

如果您有多个配置文件，您可以将它们全部添加到一个配置映射中。这可以通过将多个--from-file 选项传递给 oc 创建配置映射来完成。或者，如果这些文件本身全部位于同一个目录中，请将目录的路径传递给--from-file。目录中的每个文件都将被添加到一个单独的密钥下。

要更改配置映射，您可以使用 oc 编辑，也可以使用 oc get -o json 或 oc get -o yaml 将当前资源定义保存到文件中，编辑文件中的定义，然后通过运行替换原始文件 oc 用文件作为参数替换-f。

当您编辑配置图的定义时，更改将自动反映在从任何正在运行的容器中的配置图挂载的文件中。这不是即时的，根据 OpenShift 群集的配置情况，这些更改需要一分钟或更长时间才能显示。如果应用程序自动检测到配置文件的更改并重新读取它，则会立即使用这些更改，而无需重新部署应用程序。

如果使用配置映射来设置环境变量，则应用程序将不会自动重新部署。您需要通过在部署配置上运行 oc deploy --latest 来强制重新部署更新的值。

12.3 处理保密信息

数据库凭证等秘密可以存储在配置映射中，并使用环境变量或配置文件传递给应用程序。由于秘密的安全性很重要，OpenShift 提供了一种处理被称为秘密的秘密数据的替代资源类型。

通用秘密与配置图一样工作，但 OpenShift 以更安全的方式在内部管理它们。这包括由临时文件存储设施（tmpfs）支持的秘密的数据卷，并且永远不会停留在节点上。秘密也只能由需要它们的服务帐户访问，或者明确授予访问权限。

要创建一个通用的秘密，您可以使用 oc 创建秘密泛型，或 oc 创建-f 与秘密的 JSON /

YAML 资源定义。

如果您只需要存储简单的键/值对，则可以通过运行 `oc create secret generic` 并传递 `--from-literal` 选项以及设置的名称和值来创建秘密：

```
$ oc create secret generic blog-secrets \  
--from-literal DATABASE_USERNAME=user145c30ca \  
--from-literal DATABASE_PASSWORD=EbAYDR1sJsvW
```

您可以通过查询使用 `oc get -o json` 创建的资源来查看秘密的定义。需要重现它的定义的关键部分是：

```
{  
  "apiVersion": "v1",  
  "kind": "Secret",  
  "metadata": {  
    "name": "blog-secrets"  
  },  
  "data": {  
    "DATABASE_USERNAME": "dXNlcjE0NWZMGNh",  
    "DATABASE_PASSWORD": "RWJBWURSMXNkc3ZX"  
  }  
}
```

您会注意到，数据下的每个键的值已通过应用 `base64` 编码而被混淆。如果您要自己创建资源定义，则在添加值时需要自己进行编码。您可以使用 `Unix base64` 命令创建混淆值。

为了方便起见，特别是在模板定义中使用秘密时，只要将它们添加到 `stringData` 字段而不是数据字段中，就可以将它们作为明文提供：

```
{  
  "apiVersion": "v1",  
  "kind": "Secret",  
  "metadata": {  
    "name": "blog-secrets"  
  },  
  "stringData": {  
    "DATABASE_USERNAME": "user145c30ca",  
    "DATABASE_PASSWORD": "EbAYDR1sJsvW"  
  }  
}
```

即使以这种方式创建，在查询时，秘密将始终显示数据字段的值被混淆。可以使用 `Unix base64` 命令对值进行反混淆处理，如果通过 Web 控制台查看秘密，还可以使用一个选项来显示反混淆值。

要在部署配置中将此秘密中的设置作为环境变量传递，您需要运行额外的步骤：

```
$ oc set env dc/blog --from secret/blog-secrets
```

使用 `--list` 选项运行 `oc set env` 对部署配置的结果现在将是：

```
# deploymentconfigs blog, container blog  
# DATABASE_USERNAME from secret blog-secrets, key DATABASE_USERNAME  
# DATABASE_PASSWORD from secret blog-secrets, key DATABASE_PASSWORD
```

要使用文件创建密钥，而不是使用 `--from-literal` 使用 `--from-file`，必要时覆盖用于该值的密钥：

```
$ oc create secret generic blog-webdav-users
--from-file .htdigest=webdav.htdigest
```

要安装该密码，请使用 `oc set volume`，并使用 `--secret-name` 选项标识要使用的密码：

```
$ oc set volume dc/blog --add --secret-name blog-webdav-users \
--mount-path=/opt/app-root/secrets/webdav
```

要更改密码，您可以使用 `oc` 编辑，也可以使用 `oc get -o json` 或 `oc get -o yaml` 将当前资源定义保存到文件中，编辑文件中的定义，并通过运行 `oc` 替换原始文件以文件作为参数应用 `-f`。

编辑秘密的定义时，您需要触发新的部署以供使用。作为文件安装时，秘密的值不会自动更新，就像使用配置映射时一样。

12.4 删除配置和隐私

当您创建配置映射或秘密时，它们将独立于任何现有的构建或部署配置而创建，并且与特定应用程序无关。

如果您想将它们与特定应用程序相关联以便跟踪它们，请使用以下命令：

```
$ oc label secrets / blog-secrets app = blog
```

当您使用 `oc` 删除一个应用程序删除全部和一个标签选择器时，该标签已经应用到的任何配置映射或秘密都不会被删除。这是因为所有选项中都不包含类型为 `configmap` 和 `secret` 的资源对象。

要使用标签选择器（包括 `secret` 和 `configmap` 对象类型）删除应用程序的所有资源对象，请使用：

```
$ oc delete all,configmap,secret --selector app=blog
```

12.5 总结

可以使用环境变量将配置传递给构建或运行的容器。环境变量在构建或部署配置中设置。

在部署的情况下，也可以定义配置图或秘密。这些可以包含可用于在部署配置中填充环境变量的键/值对，或者作为文件挂载到正在运行的容器中。

秘密的工作方式与配置地图相同，但为 `OpenShift` 如何管理提供了额外的保证。这包括控制谁可以访问秘密并确保秘密永远不会存储到运行应用程序的节点上的磁盘上。

第 13 章.服务，网络和路由

在部署应用程序时，无论它是 `Web` 应用程序还是数据库，都需要它可访问，以便其他应用程序组件或用户可以访问它。但是，您希望能够控制谁或可以访问它。

在大多数情况下，当您在 `OpenShift` 中部署应用程序时，只能在同一项目中运行的其他应用程序组件访问它。为了使 `Web` 应用程序可见，以便 `OpenShift` 集群外的用户可以访问它，您需要创建一个路线。创建路线会为您的 `Web` 应用程序提供一个公共 `URL`，用户可以通过该 `URL` 访问它。

在本章中，您将了解容器与容器之间的关系，应用程序如何访问运行在相同或不同项

目中的其他应用程序，以及如何将应用程序公开给外部用户。

13.1 容器和集群

您的应用程序部署后，将在容器中运行。该容器将您的应用程序与其他应用程序隔离。在容器中，应用程序只能看到属于同一已部署应用程序的一部分进程。它无法看到在其他容器中运行的应用程序的进程。

当容器直接在主机上运行时，尽管应用程序彼此隔离，但所有容器通常会共享相同的 IP 地址和端口名称空间。

这意味着如果您想运行同一个 Web 应用程序的多个实例，并且都希望使用相同的侦听器端口来接受 Web 请求，它们将会相互冲突。

出于这个原因，当您的应用程序在 OpenShift 中运行时，容器被进一步封装在所谓的 pod 中。

一个容器是一组具有共享存储和网络资源的容器。容器中的容器始终位于同一位置并进行共同调度，并在共享环境中运行。

容器中的容器共享 IP 地址和端口名称空间，并且可以通过本地主机查找对方。它们还可以使用 SystemV 信号量或 POSIX 共享内存等本地进程间通信（IPC）机制相互通信。不同容器中的容器具有不同的 IP 地址，并且不能使用本地 IPC 机制进行通信。

每个具有自己的 IP 地址和端口名称空间的窗格意味着应用程序的多个实例（都希望使用相同的侦听器端口来接受 Web 请求）可以在同一主机上运行，而无需重写每个正在使用的端口。在这方面，一个吊舱表现得好像是它自己的主机一样。

要查看项目中所有窗格的列表，可以运行 `oc get pods` 命令。如果希望在一个窗格中看到更多信息，包括窗格的 IP 地址和在窗格中运行的每个容器的详细信息，请运行 `oc describe pod`，将该窗格的名称作为参数传递。从容器外部无法查看正在运行的进程。为了查看容器中正在运行的内容，可以使用 `oc rsh` 在容器内启动交互式终端会话。如果应用程序映像捆绑了所需的 Unix 命令，则可以使用终端会话中的进程进行交互，就像在自己的主机上运行一样。您可以使用 Unix 命令（如 `ps` 或 `top`）列出正在运行的进程。

13.2 服务和端点

每个吊舱都有一个独特的 IP 地址。从在同一项目中运行的任何应用程序，您可以使用该窗格中正在使用的应用程序的端口上的 IP 地址连接到另一个窗格。

为了能够从 pod 外部的端口接受连接，应用程序应绑定到网络地址 0.0.0.0 而不是 127.0.0.1 或 localhost。

不建议在连接时使用吊舱的直接 IP 地址。这是因为 IP 地址不是永久性的，并且如果一个 pod 被杀死并且被在一个新的 pod 中运行的应用程序的实例取代，那么 IP 地址可能会改变。

如果您有多个应用程序实例，则它们将分别运行在一个单独的窗格中。这些可能位于同一节点上，也可能位于 OpenShift 群集中的不同节点上。每个实例都有一个单独的 IP 地址。

为了有一个永久 IP 地址可用于连接到应用程序的任何实例并负载平衡应用程序实例之间的连接，OpenShift 提供了服务抽象。如果使用 `oc new-app` 或 Web 控制台从预先存在的映像部署应用程序，或者从源代码构建应用程序，则会自动为您创建服务资源对象。

要查看应用程序的 pod 和服务的列表，您可以运行 oc 获取窗格，服务并提供与应用程序使用的匹配的标签选择器：

```
$ oc get pods,services --selector app=blog
```

NAME	READY	STATUS	RESTARTS	AGE
po/blog-2-sxbpd	1/1	Running	0	1m
po/blog-2-ww5ck	1/1	Running	0	1m

NAME	CLUSTER-IP	EXTERNAL-IP	PORT(S)	AGE
svc/blog	172.30.229.46	<none>	8080/TCP	2m

服务条目将显示可用于连接到应用程序的永久 IP 地址。当应用程序有多个实例时，您连接的实例将是随机的。要查看与服务关联的 pod IP 地址列表，可以使用 oc get endpoints 命令：

```
$ oc get endpoints blog
```

NAME	ENDPOINTS	AGE
blog	172.17.0.10:8080,172.17.0.8:8080	2m

虽然服务的 IP 地址在应用程序的整个生命周期中保持不变，但您无法事先确定要使用的 IP 地址。

如果您直接将应用程序部署到主机，为了避免应用程序的用户知道 IP 地址，您应该在 DNS 服务器中针对主机名注册 IP 地址。当您使用 OpenShift 并创建服务时，将为您完成此操作，IP 地址在 OpenShift 群集内部的 DNS 服务器中注册，并使用与服务名称匹配的主机名。

在这个示例中，不是在同一项目中运行的客户端应用程序需要使用 IP 地址，而是可以使用主机名博客连接到应用程序。

在 DNS 服务器中注册 IP 地址意味着客户端应用程序可以编码为使用固定的主机名，并且部署的后端应用程序或数据库在创建时只需使用相同的名称。或者，您可以使用环境变量将后端应用程序或数据库的主机名传递给客户端应用程序，或者使用配置映射或秘密传递给客户端应用程序的容器中的配置文件。

13.3 连接项目

在将应用程序的服务的非限定名称用作主机名时，只能从部署应用程序的项目中解析名称。如果您需要能够访问不同项目中的后端应用程序或数据库，则需要使用包含项目名称的合格主机名。

全限定主机名的格式是：

```
<service-name>.<project-name>.svc.cluster.local
```

例如，如果服务名称是博客，并且它已部署在项目 myproject 中，则完全限定的主机名将为 blog.myproject.svc.cluster.local。

由于 OpenShift 的安装默认设置为多租户，并且一个项目中的应用程序无法看到任何部署在不同项目中的应用程序，因此在尝试直接从一个项目到另一个项目建立连接之前，需要启用访问。要打开项目之间的访问权限，可以使用 oc adm pod-network 命令。

如果您正在使用 Minishift 或 oc 群集，则这些功能不会启用多租户网络覆盖。这意味着您实际上可以跨项目建立连接，而无需执行任何操作。

如果您需要此功能，请查看关于[管理网络的 OpenShift 文档](#)。

13.4 创建外部路由

无论是 pod 或服务的直接 IP 地址还是服务的主机名都不能用于从 OpenShift 集群外部访问应用程序。为了使 Web 应用程序公开并可从 OpenShift 集群外部访问，您需要创建一个路径。

要为 Web 应用程序公开服务，以便用户可以从外部访问服务，可以运行 `oc expose service` 命令，并将该服务的名称作为附加参数传递：

```
$ oc expose service/blog  
route "blog" exposed
```

创建路由时，OpenShift 将默认为您的应用程序分配一个唯一的主机名，通过它可以从 OpenShift 集群外访问它。您可以看到通过在路由上运行 `oc` 描述创建的路由的详细信息：

```
$ oc describe route/blog  
Name:                blog  
Namespace:           myproject  
Created:             5 minutes ago  
Labels:              app=blog  
Annotations:         openshift.io/host.generated=true  
Requested Host:      blog-myproject.b9ad.pro-us-east-1.openshiftapps.com  
                     exposed on router router 5 minutes ago  
Path:                <none>  
TLS Termination:     <none>  
Insecure Policy:     <none>  
Endpoint Port:       8080-tcp  
Service: blog  
Weight:              100 (100%)  
Endpoints:           172.17.0.4:8080
```

如果您拥有自己的自定义主机名，则可以使用它，而不是依赖分配的主机名。这可以通过使用 `--hostname` 选项将主机名传递给 `oc` 来完成。

使用自定义主机名时，您需要控制域名的 DNS 服务器。然后您需要在主机名的 DNS 服务器配置中创建一个 `CNAME` 记录，并将其指向 OpenShift 集群入站路由器的主机名。

如果无法确定入站路由器的主机名，则通常可以将 `CNAME` 的任何主机名值用作 OpenShift 集群用于生成主机名的通配符子域内的目标。

对于前面的示例，分配的主机名是：

```
blog-myproject.b9ad.pro-us-east-1.openshiftapps.com
```

作为 `CNAME` 记录的目标，您可以使用：

```
myproject.b9ad.pro-us-east-1.openshiftapps.com
```

如果您有多个应用程序实例并且创建了路线，则 OpenShift 将自动配置路由，以便在实例之间分配流量。

将使用粘滞会话，以便来自同一客户端的流量将优先路由回同一个实例。

13.5 使用安全连接

使用 `oc` 创建的路由只公开支持使用 HTTP 协议的请求。它不能用于使用非 HTTP 协议公开数据库服务。

在 HTTP 通信的情况下，如果您希望客户端使用安全连接，则需要使用 `oc create route` 创建路由，而不是使用 `oc expose`。

支持三种安全路由：

边缘

安全连接由路由器终止，路由器使用内部群集网络上的不安全连接代理与应用程序的连接。如果未提供您自己的主机名和 SSL 证书，则将使用 OpenShift 群集的 SSL 证书。尽管内部使用了不安全的连接，但群集的其他用户不可见流量。

直通

路由器将直接通过应用程序代理安全连接。

该应用程序必须能够接受安全连接并配置相应的 SSL 证书。假设客户端支持通过传输层安全（TLS）连接的服务器名称标识（SNI），则可以使用该标识符来允许使用非 HTTP 协议访问应用程序。

重新加密

安全连接由路由器终止，路由器在代理到应用程序的连接时重新加密流量。如果未提供自己的主机名和 SSL 证书，则 OpenShift 群集的 SSL 证书将用于初始入站连接。对于从路由器到应用程序的连接，应用程序必须能够接受安全连接并配置有用于重新加密连接的适当 SSL 证书。

要使用边缘终止创建安全连接，请使用由 OpenShift 分配的主机名，然后运行以下命令：

```
$ oc create route edge blog --service blog
```

如果您已使用 `oc expose` 为不安全连接创建路由，则为路由提供的名称必须不同。在这个例子中，路由的名称是作为博客安全的。

不要为不安全和安全的连接创建单独的路由，您可以通过指定应如何处理不安全的连接来创建一个涵盖两种情况的单个路由。

如果您想允许 Web 应用程序通过不安全和安全的连接接受 HTTP 请求，请运行：

```
$ oc create route edge blog --service blog --insecure-policy Allow
```

如果您希望尝试使用不安全连接的用户重定向，以便使用安全连接，请运行：

```
$ oc create route edge blog --service blog --insecure-policy Redirect
```

如果使用自己的自定义主机名，则可以使用 `--hostname` 选项提供它。

您将需要使用 `--key`、`--cert` 和 `--ca-cert` 选项提供 SSL 证书文件。

有关 OpenShift 提供的所有路由功能（包括使用传递和重新加密安全路由）的更深入讨论，请参阅[路由](#)上的 OpenShift 体系结构文档。

13.6 内部和外部端口

当使用 OpenShift 托管应用程序时，容器运行的用户标识将根据其运行的项目进行分配。默认情况下，容器不允许以 `root` 用户身份运行。由于容器不以 `root` 用户身份运行，因此它们将无法使用低于 1024 的特权端口号。例如，Web 应用程序不能使用用于 HTTP 的标准端口 80。

设计为不需要以 `root` 身份运行的映像将使用更高的端口号。对于 Web 应用程序，通常

使用端口 8080 而不是端口 80。通过运行 `oc get` 服务，您可以看到服务被宣布为使用的端口。在 OpenShift 群集中，当您直接通过 IP 地址或通过使用 IP 地址或主机名的服务访问某个群时，您可以使用此端口号。

如果您使用路由在外部公开外部服务，则外部用户将使用标准端口 80 进行 HTTP 请求，或使用端口 443 进行安全连接的 HTTP 请求。路由层将始终通过标准端口接受请求，并在向应用程序发出请求代理时将这些请求映射到内部端口号。

路由 HTTP 请求时使用的端口取决于映像使用何种端口进行广告，以及哪些端口已添加到服务中。如果图像广告的端口不止一个，则使用第一个端口。如果这是错误的端口，则可以通过将 `--port` 选项传递给 `oc` 公开服务或 `oc` 创建路由来覆盖将使用的端口。

13.7 公开非 HTTP 服务

如果您正在运行基于 HTTP 的服务或可终止安全连接的服务，并且客户端支持使用 TLS 的安全连接的 SNI，则通过路由公开服务仅适用。

如果你想公开不同类型的服务，你有两种选择。

首先，您可以为该服务专用一个新的公共 IP 地址，并配置网络路由以将该地址上的任何端口的连接传递到内部服务的 IP 地址。如果您有多个使用相同端口的服务，则需要为每个端口分配一个单独的公用 IP 地址。

或者，可以将网关主机上的专用端口分配给该服务。此端口上的任何连接都将路由到内部服务上的端口。为此类服务分配一个特定的端口将导致端口在集群中的每个节点上被保留，即使它一次只能在一个节点上使用。如果它是一个标准端口，那么您将无法将其用于任何其他服务。

这两种方法都需要集群管理员执行其他设置。

有关更多信息，请查看 OpenShift 文档，了解如何将流量导入 [非标准端口](#) 上的群集。

13.8 本地端口转发

如果您想公开非基于 HTTP 的服务的原因是允许临时访问以允许调试应用程序，加载数据或管理，则可以使用端口转发将服务暂时暴露给本地计算机。

要使用端口转发，您需要标识要与之通信的特定窗格。您不能使用端口转发通过服务公开应用程序。使用 `oc` 获取豆荚与适当的标签选择器来标识该应用程序的豆荚：

```
$ oc get pods --selector name=postgresql
```

NAME	READY	STATUS	RESTARTS	AGE
postgresql-1-8cng2	1/1	Running	0	5m

然后，您可以使用容器的名称和要连接的容器上的端口运行 `oc` 端口转发：

```
$ oc port-forward postgresql-1-8cng2 5432
```

```
Forwarding from 127.0.0.1:5432 -> 5432
```

```
Forwarding from [::1]:5432 -> 5432
```

远程端口将使用相同的端口号在本地公开。如果端口号已在本地机器上使用，您可以指定一个不同的本地端口来使用：

```
$ oc port-forward postgresql-1-8cng2 15432:5432
```

```
Forwarding from 127.0.0.1:15432 -> 5432
```

```
Forwarding from [::1]:15432 -> 5432
```

You can also have oc port-forward select a local port for you:

```
$ oc port-forward postgresql-1-8cng2 :5432
```

```
Forwarding from 127.0.0.1:48888 -> 5432
```

```
Forwarding from [::1]:48888 -> 5432
```

oc 端口转发命令将在前台运行，直到它被终止或连接丢失。在连接处于活动状态时，您可以在本地运行客户端程序，连接到 127.0.0.1 上的转发端口，并将连接代理到远程应用程序：

```
$ psql sampledб username --host=127.0.0.1 --port=48888
```

```
Handling connection for 5432
```

```
psql (9.2.18, server 9.5.4)
```

```
Type "help" for help.
```

```
sampledb=>
```

13.9 总结

将应用程序部署到 OpenShift 时，默认情况下，它只能在 OpenShift 集群中访问。您可以使用从应用程序的服务名称派生的内部主机名从群集内访问应用程序。

当应用程序有多个实例时，连接到应用程序将导致连接被路由到运行应用程序的其中一个窗格。

如果您需要为 OpenShift 集群外的用户提供 HTTP Web 服务，则可以通过创建路由来公开它。创建路由将导致 OpenShift 自动为您重新配置路由层。您可以通过 HTTP 协议公开 Web 应用程序，也可以通过安全连接将其公开为 HTTPS。OpenShift 可以为您提供外部主机名，或者您可以使用自己的自定义主机名。

服务也可以暴露在专用的 IP 地址或端口上，或者使用端口转发暂时发送到您自己的本地系统。

第 14 章.使用持久存储

当应用程序在容器中运行时，它可以访问它自己的文件系统。它包含映像中的操作系统文件，任何应用程序服务器或语言运行时文件的副本，以及正在运行的应用程序的源代码或编译二进制文件。

当应用程序运行时，它可以将文件写入有权写入的文件系统的任何部分，但是当容器停止时，所做的任何更改都将丢失。这是因为本地容器文件系统是短暂的。

为了保存应用程序在应用程序重新启动时创建的数据，或在应用程序实例之间共享动态数据，需要持久存储。这可能是直接附加到运行应用程序的容器的持久性存储，也可能是连接到应用程序正在使用的 OpenShift 中运行的单独数据库的持久性存储。

在本章中，您将学习 OpenShift 提供的持久性存储，如何制作持久性卷声明以及如何将持久性卷挂载到应用程序的容器中。

14.1 持久存储的类型

在设置 OpenShift 群集时，群集管理员会将其配置为持久存储。可用的持久卷可以从固

定存储器预先分配，或者可以设置动态存储供应器，持久存储提供者可以根据需要分配持久卷。

OpenShift 支持许多底层存储技术，包括 NFS，GlusterFS，Ceph RBD，OpenStack Cinder，AWS Elastic Block Storage，GCE 永久存储，Azure 磁盘，Azure 文件，iSCSI，光纤通道和 VMware vSphere。

在声明卷时，它们可以与指示所用技术类型或其他属性（例如磁盘性能）的存储类关联。

如果需要，可以在请求存储时指示需要具有特定存储级别的持久性卷。

所使用的存储技术类型也将决定支持哪些访问模式以及如何使用持久性卷。

持久存储的访问模式是：

ReadWriteOnce (RWO)

卷可以作为单个节点的读/写进行装载。

ReadOnlyMany (ROX)

卷可以作为只读由许多节点安装。

ReadWriteMany (RWX)

卷可以作为许多节点的读/写装入。

有关更多信息，请参阅 OpenShift 文档了解[存储类别](#)和[访问模式](#)。

如果您是群集管理员，则可以通过运行 `oc get pv` 命令来查看预定义的持久性卷，包括访问模式和存储类。OpenShift 群集的非管理员用户无法访问此信息，因此您需要向群集管理员询问可用的功能，或者如果使用托管的 OpenShift 服务，请检查服务提供商提供的文档。

OpenShift 群集中可用的持久存储可能不支持所有访问模式。如果唯一受支持的访问模式是 ReadWriteOnce，这将限制您如何使用持久性存储。

在使用 ReadWriteOnce 访问模式的持久存储的情况下，您不能将其与缩放的应用程序一起使用，您将无法使用滚动部署。

这是因为一次只支持 OpenShift 集群中的一个节点的仅支持该访问模式的持久卷。扩展应用程序时，不保证所有实例都将在同一节点上运行。在滚动部署的情况下，即使副本计数设置为一个实例，应用程序的新实例也将在现有关闭实例关闭之前启动。这提出了与应用程序扩展时相同的问题。

[第 17 章](#)介绍部署策略的主题以及如何更改部署策略。

14.2 声明持久卷

当您需要应用程序的持久性存储时，您需要提出持久性卷声明。进行索赔时，您必须指定所需持久卷的大小。您也可以选择指定您需要持久卷支持的访问模式。如果未提供，访问模式将默认为 ReadWriteOnce。

要制作持久性卷声明并根据应用程序的每个实例安装持久性卷，可以使用 `oc set volume --add` 命令。您必须提供目录路径以用作容器中持久卷的装入路径。您可以选择指定持久卷声明的名称和名称以标识部署配置中的卷装入。例如：

```
$ oc set volume dc/blog --add \
--type=pvc --claim-size=1Gi --claim-mode=ReadWriteOnce \
--claim-name blog-data --name data --mount-path /opt/app-root/src/media
persistentvolumeclaims/blog-data
deploymentconfig "blog" updated
```

如果您需要具有特定存储类的持久性卷，则可以使用`--claim-class` 选项。

使用 `oc set volume` 将持久性卷添加到部署配置时，应用程序将自动重新部署。一个持久卷将被装入每个容器中，用于应用程序的所有实例。如果某个容器重新启动，则替换容器将使用相同的持久容量。因此，在持久卷中所做的任何更改都将在所有实例之间共享，并将在重新启动应用程序时保留。要列出已针对应用程序添加的持久性卷，请针对部署配置运行 `oc set volume`，而不使用任何其他参数：

```
$ oc set volume dc/blog
deploymentconfigs/blog
pvc/blog-data (allocated 5GiB) as data
mounted at /opt/app-root/src/media
```

这将显示正在使用的持久性卷的实际大小。该尺寸可能比请求的尺寸要大，因为持续卷是用设定尺寸定义的。OpenShift 将使用满足您请求的最小卷大小。永久卷容量的限制取决于容量的大小，而不是请求的大小。要查看项目中当前的持续量声明，请运行 `oc get pvc`：

```
$ oc get pvc
NAME STATUS VOLUME CAPACITY ACCESSMODES STORAGECLASS AGE
blog-data Bound pv0088 5Gi RWO,ROX,RWX 3m
```

这也将显示您分配的持续卷支持的所有访问模式。

14.3 卸载持久卷

要停止在应用程序中使用持久性卷，可以使用 `oc set volume --remove` 命令。您必须提供用于在部署配置中标识卷装入的名称：

```
$ oc set volume dc/blog --remove --name data
deploymentconfig "blog" updated
```

该命令只会从应用程序的容器中删除卷装入。它不会删除持久卷。运行 `oc` 获得 `pvc`，您仍然应该看到列出的持久性卷索赔。

14.4 重新使用持久卷声明

如果您有一个持久性卷索赔，说明您之前已从应用程序卸载，则 `oc set volume --add` 命令可用于将其添加回应用程序：

```
$ oc set volume dc/blog --add \
--claim-name blog-data --name data --mount-path /opt/app-root/src/media
deploymentconfig "blog" updated
```

由于持久性卷声明已存在，因此您无需指定卷类型，所需的卷大小或所需的访问模式。传递给`--claim-name` 选项的名称必须与现有的持久卷索赔相匹配。

14.5 在应用程序间共享

如果持久卷的访问模式为 `ReadWriteMany` 或 `ReadOnlyMany`，则可以同时安全地将该持久卷安装在多个应用程序中。这将允许您使用单个持久卷在应用程序之间共享数据。

要在多个应用程序之间共享一个持久性卷，但只能看到存储在持久性卷中的一部分内

容，则可以将`--sub-path` 选项传递给 `oc set volume --add`，指定该子目录的子目录 持久卷应该被安装到应用程序的容器中。

14.6 在容器之间共享

如果多个容器在一个容器中运行，如果他们需要查看相同的文件，则它们都可以共享一个持久容量。但是，相同的文件也可以与运行相同应用程序的所有其他 Pod 共享。

如果共享文件的要求只在该容器中的容器之间，并且文件不需要在重新启动容器时保持不变，则可以使用名为 `emptyDir` 的特殊容量类型。

这种类型的卷是从每个节点上的本地存储分配的，并专用于该容器。当吊舱重新启动时，存储被删除。

除了可以用来保存普通文件外，还可以在卷中创建特殊文件类型（如 Unix 套接字），或者可以将卷挂载到 `/dev/shm` 以启用共享内存访问。这样，不同容器中的应用程序就可以相互通信。

如果您正在使用 `init` 容器，则此类型的卷也可用于将由 `init` 容器生成的文件映射到应用程序容器中。这些可能是数据文件，配置文件或应用程序的自定义启动脚本。

使用 `oc set volume` 添加此类型的卷时，请使用 `--type = emptyDir` 选项。默认情况下，卷将挂载到容器中的所有容器。如果您需要指定容器的子集，请使用 `--container` 选项来命名它们。

14.7 删除持久卷

从所有应用程序卸载持久性卷时，持久性卷声明以及持久性卷将仍然存在。您添加到持久卷的任何现有数据都将保留在那里。

如果您不再需要持久性卷，则可以通过在持久性卷声明中运行 `oc delete pvc` 命令来释放它：

```
$ oc delete pvc/blog-data
```

```
persistentvolumeclaim "blog-data" deleted
```

持久卷的回收策略通常是“回收”，这意味着只要删除持久卷，其内容就会被删除，持久卷将返回可用持久卷池。

为防止意外删除持久卷，群集管理员可以选择将持久卷上的回收策略设置为保留。在这种情况下，如果意外删除了一个持久性卷，它将不会自动回收。如果您知道正在使用此回收策略，则可以联系您的集群管理员以查看是否可以使用相同的持久性卷恢复持久性卷索赔。您需要知道持续卷声明的原始卷名是什么。您可以从运行 `oc get pvc` 的输出获取当前的持续卷声明。

14.8 将数据复制到卷

如果您的应用程序正在运行并且挂载了持久性卷，则可以使用 `oc rsync` 将目录从本地系统复制到持久性卷中。首先确定装载持久性卷的应用程序的 pod 名称：

```
$ oc get pods --selector app=blog
```

NAME	READY	STATUS	RESTARTS	AGE
blog-1-5m3q6	1/1	Running	0	2m

然后可以运行 `oc rsync` 来复制目录。

```
$ oc rsync / tmp / images blog-1-5m3q6: / opt / app-root / src / media --no-perms
```

`--no-perms` 选项告诉 `oc rsync` 不要尝试保留目录和文件的权限。在将文件复制到本地容器文件系统时，这是必要的，并且正在复制文件的目录不属于容器运行的用户标识，而是由运行 `S2I` 构建器的用户拥有。

如果没有此选项，`oc rsync` 在尝试更改目录权限时会失败。

`oc rsync` 命令也可用于将目录或文件从正在运行的容器复制回本地系统。在任一方向复制都可以作为一次性事件运行，或者您可以让 `oc rsync` 持续监视更改并在每次更改时复制文件。

14.9 总结

`OpenShift` 能够支持的不仅仅是无状态的 12 因素或云本地应用程序。使用将持久卷挂载到容器中的能力，应用程序可以保存需要在实例之间共享的数据或在重新启动后保留的数据。

持久性存储可以根据特定的存储类型或类别以及所需的持久性卷的大小来声明。`OpenShift` 会将持久卷挂载到一个容器中，即使应用程序移动到 `OpenShift` 群集中的其他节点，存储也会自动跟随应用程序。

第 15 章.资源配额和限制

每当您与 `OpenShift` 进行交互时，您都将以特定用户身份执行操作。如果您正在部署应用程序，则您的操作将在项目的上下文中执行。不过，你不能只做你喜欢的任何事情。控件存在于你可以做什么，以及你可以消耗多少资源。

资源对象上的配额控制您创建的项目数量，可以部署的应用程序数量或可以使用的持续卷数量。还可以指定配额，以限制您可以在所有应用程序中使用的最大内存量或 `CPU` 数量。限制范围可以通过指定单个应用程序可以使用多少来进一步控制如何使用内存或 `CPU`。

在定义的任何限制范围内，您可以指定应用程序实际需要多少内存或 `CPU`。这使您可以划分整体内存和 `CPU` 配额，以充分利用可用资源来尽可能多地部署不同的应用程序或实例。定义您的应用程序需要多少内存和 `CPU` 也有助于 `OpenShift` 调度程序找出运行应用程序的最佳位置。

在本章中，您将了解控制项目和应用程序可以消耗多少资源的配额和限制。您还将了解如何指定您的应用程序需要多少内存和 `CPU`。

15.1 什么是配额管理

配额用于管理两类资源。第一类是资源对象 - 也就是说，可以为特定类型创建多少个对象。这类资源上的配额可能会控制您可以创建多少个项目，您可以部署多少个应用程序，可以运行多少个应用程序实例，或者可以使用多少个持久性卷。

第二类是计算资源。此类别中的配额可以设置可用于所有应用程序的 `CPU` 和内存资源总量。

计算资源上的配额可以根据工作负载的类型来定义。非终止资源上的配额将应用于您永久运行的应用程序。终止资源的配额将应用于临时工作负载，例如从 OpenShift 的源代码构建映像或运行作业。

您正在处理的项目是否受限于资源配额取决于 OpenShift 群集的配置方式。有几种方法可以确定是否将配额应用于项目。

当您使用 Minishift 或 oc 群集时，配额和限制范围不会被预定义。在使用这些资源时，您可以使用多少资源取决于底层主机的可用数量。

在 Web 控制台中，通过从左侧导航栏的 Resources 菜单中选择 Quota（图 15-1），可以查看任何正在应用于项目的配额。

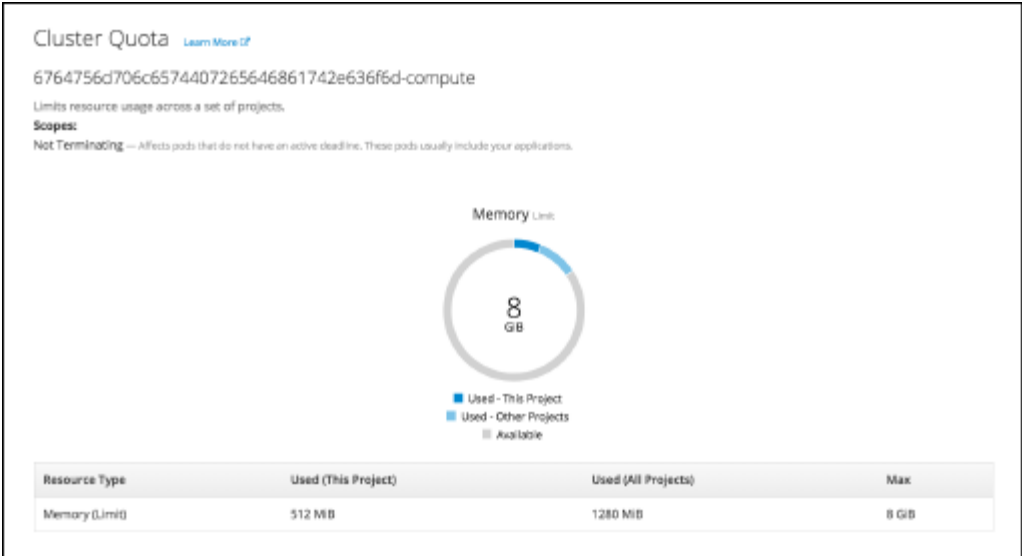


图 15-1 资源配额

在命令行中，如何确定正在应用的配额将取决于它们是以每个项目为基础应用还是应用于您创建的所有项目。

如果按每个项目应用配额，则可以使用 `oc describe quota` 命令查看它们。如果配额涉及多个项目，则您将使用 `oc describe appliedclusterresourcequota` 命令查看它们。

正在应用的任何配额将由群集管理员事先设置。作为用户，您无法通过 OpenShift 更改配额。

因为定义配额的方式有很大的灵活性，所以不同的 OpenShift 群集可能会以不同的方式应用它们。使用典型的 OpenShift 群集配置，您将看到单独的 CPU 和内存配额。也有可能你只能看到一个配额用于记忆。这是因为群集管理员可以配置群集，使得 CPU 与分配给容器的内存量的比例有限。对容器使用 1 Gi 内存可能会转换为 2 个 CPU 内核。在此场景中将分配给容器的内存减少到 512 M 将会降低单个 CPU 内核可用 CPU 数量的限制。

有关此主题的更多信息，请参阅关于 [资源过度使用](#) 的 OpenShift 文档。

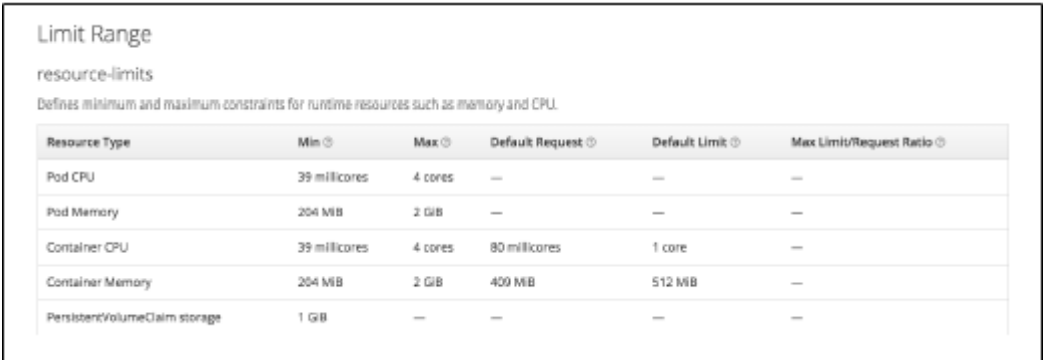
15.2 配额与限制范围

配额设置了所有应用程序可以使用的资源总量的上限。配额不能确定在部署多个应用程序时，单个应用程序可以消耗多少计算资源。在这种情况下唯一的约束是所有应用程序消耗的总计算资源不能超过配额。

单个应用程序可以消耗的计算资源数量受限制范围控制。限制范围适用于每个项目。

您可以在 **Web** 控制台的配额页底部找到项目的限制范围，如图 15-2 所示。

限制范围详细信息位于显示群集配额的相同配额页的底部，如上一节中所述。在左侧菜单中导航到资源→配额。请注意，配额页面非常长，并且没有显示其他详细信息。



Resource Type	Min	Max	Default Request	Default Limit	Max Limit/Request Ratio
Pod CPU	39 millicores	4 cores	—	—	—
Pod Memory	204 MiB	2 GiB	—	—	—
Container CPU	39 millicores	4 cores	80 millicores	1 core	—
Container Memory	204 MiB	2 GiB	400 MiB	512 MiB	—
PersistentVolumeClaim storage	1 GiB	—	—	—	—

图 15-2 限制范围

限制范围也可以使用 `oc` 描述限制从命令行查看。

如果已经为内存和 **CPU** 使用量定义了配额，则还会为项目定义这些计算资源的限制范围。这是必要的，这样当应用程序没有定义它需要的内存或 **CPU** 的数量时，可以使用默认值。

15.3 请求与限制

除了定义内存和 **CPU** 的数量之外，应用程序还需要考虑配额计算，这也是 **Open-Shift** 调度程序如何解决应用程序部署问题的主要因素。当调度程序知道您需要的最小内存量和 **CPU** 时，它可以确保您的应用程序在有足够的计算资源可用于满足您的请求的节点上运行。

即使配额尚未定义，群集管理员通常仍会将限制范围定义添加到项目。这将确保应用程序总是对它们所需的计算资源进行一些估计，并且调度程序将能够更好地管理应用程序在节点上的放置。

定义所需的计算资源时，可以为每个容器定义 **CPU** 和内存的请求和限制值。请求值指示在容器中运行的应用程序需要的最小计算资源量。限制是它可以增长消耗的最大值。

如果容器未指定请求或限制值，并且限制范围为该计算资源定义了默认值，则使用默认值。

根据是否设置了请求和限制值及其值，应用程序将被分配到服务质量层。这会影响应用程序在诸如节点上可用资源较少的情况下如何处理。标记服务质量较低的应用程序更有可能从节点中逐出并在不同节点上重新启动，这可能会影响应用程序的可用性。

有关如何应用配额和限制（包括服务质量等级）的更多详细信息，请查看 **OpenShift** 有关[配额和限制范围](#)的文档。

15.4 资源需求

要查看正在应用于应用程序的资源限制，可以在应用程序的其中一个窗格上运行 `oc describe`。这将显示对于容器中的每个容器，**CPU** 和内存的请求和限制值：

```
Limits:
  cpu: 500m
  memory: 256Mi
```

Requests:

cpu: 40m

memory: 204Mi

这些是在从部署配置中获取定义值并从项目默认限制中填入缺失值之后的值。

要查看部署配置指定的值，请运行 `oc` 描述：

Limits:

memory: 256Mi

在这个例子中，只指定了内存的限制值。因此，请求值是从项目默认限制中填入的。由于没有定义 CPU 资源需求，因此也从项目缺省值中填充这些需求。

要修改应用程序的资源需求，可以运行 `oc set resources` 命令：

```
$ oc set resources dc/blog --limits memory=512Mi
```

```
deploymentconfig "blog" resource requirements updated
```

对于典型的 OpenShift 集群，您将能够为 CPU 和内存设置请求和限制值。提供的任何值必须介于资源限制范围指定的最小值和最大值之间，并且请求不得超过限制。

如果没有提供 CPU 配额，并且可用 CPU 按内存的比例计算，则只能设置内存的资源要求。对于某些 OpenShift 群集配置，也可以忽略任何请求值，并将其作为限制的百分比进行计算。询问您的集群管理员如何配置配额，或者查看他们的文档以获取更多详细信息，如果您使用的是 OpenShift 服务提供商。

15.5 重写构建资源

限制范围不适用于在从源代码构建 OpenShift 中的图像时运行的容器。CPU 和内存上的构建请求和限制值全局定义为整个群集。构建内存的典型限制是 512M。当使用 `nodejs` 或 `python S2I` 构建器时，在将某些软件包作为 S2I 构建的一部分进行安装时，这可能是不够的。

这些 CPU 和内存的值可以被覆盖，但是在编写本书时，`oc set resources` 命令不能用于更新构建配置。在阅读本文时，这种情况可能会发生变化。要在构建配置中更改内存限制值，可以运行：

```
$ oc set resources bc/blog --limits memory=1Gi
```

```
buildconfig "blog" resource requirements updated
```

如果 `oc` 集资源不适用于您正在使用的 OpenShift 版本，则可以使用 `oc` 修补程序更新构建配置：

```
$ oc patch bc/blog --patch '{"spec":{"resources":{"limits":{"memory":"1Gi"}}}}'
```

```
buildconfig "blog" patched
```

在为内部版本设置内存和 CPU 限制时，这些值需要小于终止资源的配额。

15.6 总结

当您创建一个新项目来保存您的应用程序时，配额会指定这些应用程序可以使用多少个 CPU 和内存资源。单独配额存在非终结和终止工作量。非终结配额适用于您的应用程序。终止配额是适用于构建和作业的内容。

如果您没有明确说明您的应用程序需要多少资源，则会为请求的资源 and 最大资源应用默认值。指示您需要的资源量很重要，因为它使 OpenShift 能够将工作负载安排到资源可用的位置。

第 16 章.监控应用程序健康

在部署应用程序时，OpenShift 需要知道它是否在发送流量之前正确启动。即使在启动后，如果它无法正常工作，也希望它重新启动。

要监视应用程序的运行状况，可以定义一个准备就绪探测器和一个活动探测器。OpenShift 将定期运行探针来监视您的应用程序。准备就绪探针用于确定您的应用程序是否处于其他应用程序或外部用户可以与之通信的状态。活跃度探测器用于确定您的应用程序是否仍然正常运行。

在本章中，您将更多地了解每种探针的用途以及成功或失败时采取的行动。您还将了解可以实施探针的不同方式。

16.1 准备探测器的作用

准备就绪探测器检查应用程序是否准备好为请求提供服务。当为应用程序创建新的应用程序时，如果您提供了准备就绪探测器，它将用于定期检查在该应用程序中运行的应用程序实例是否已准备就绪处理请求。当探针成功完成新的 pod 时，该 pod 的 IP 地址将被添加到与该服务关联的端点列表中。

将 Pod 的 IP 地址添加到活动端点列表后，其他应用程序将能够通过服务的 IP 地址或内部主机名与其进行通信。如果已经针对服务创建了路由，则路由层将被自动重新配置，以便外部流量可以到达在新的 pod 中运行的应用程序。

如果探头在启动时持续发生故障，则认为该探测器的部署失败。这将导致整个部署或扩展事件失败。

如果探测成功并且 IP 地址已添加到端点，则探测仍将被定期用于检查应用程序是否能够继续接受请求。如果探测随后开始失败，则 IP 地址将从与该服务关联的端点列表中删除，但该窗格将保留运行。

应用程序成功启动后，探测器的这种故障可以由应用程序使用，以控制在处理请求的应用程序队列为该实例填满的情况下，通信量在哪里流动。当请求积压已被清除时，应用程序应再次通过准备就绪检查，并且 IP 地址将被添加回与服务关联的端点列表。

如果未提供准备就绪探针，则会假定该容器始终处于准备就绪状态，只要该容器启动，IP 地址就会添加到与该服务关联的端点列表中，并且只有在关闭容器时才会将其删除。

如果使用滚动部署策略，建议始终使用准备就绪探针。这是因为它确保只有在准备就绪的情况下，新 Pod 才会提出请求，确保在部署新版本应用程序时停机时间为零。部署策略和滚动部署的主题将在[第 17 章](#)中详细介绍。

16.2 活力探测器的作用

生存性探测器检查应用程序是否仍在工作。

当您提供活跃性探测时，它将用于定期检查运行在窗格中的应用程序实例是否仍在运行以及它是否也正常运行。

如果探头一直不能工作，那么吊舱将关闭，并启动一个新的吊舱来更换它。

16.3 使用 HTTP 请求

探针可以实现的第Ⓐ种机制是使用 HTTP GET 请求。如果应用程序为请求返回的 HTTP 响应代码介于 200 和 399 之间，则该检查被视为成功。

要将 HTTP 请求注册为探针，请针对部署配置运行 `oc set probe` 命令。根据您要设置的探针类型，使用 `--readiness` 和/或 `--liveness` 选项。然后使用 `--get-url` 选项为执行探针的处理程序提供 URL：

```
$ oc set probe dc / blog --readiness --get-url = http: //: 8080 / healthz / ready
```

指定 URL 时，请忽略主机名部分。主机名将自动填入探测器用于检查的吊舱的 IP 地址。必须指定端口号，并且应该是 pod 中运行的应用程序用来接受 HTTP 请求连接的端口。

虽然您可以使用 Web 应用程序处理的现有 URL，但建议您为每种类型的探针创建专用处理程序。通过这种方式，您可以定制每个处理程序来执行特定于探针类型的检查。

要删除探针，可以使用 `--remove` 选项来设置探针：

```
$ oc set probe dc/blog --readiness --liveness --remove
```

16.4 使用容器命令

依赖 HTTP 请求的探测器要求应用程序是一个 Web 应用程序，或者将一个单独的 Web 服务器部署在同一个容器或边车容器中，以处理请求并执行所需的检查。HTTP 请求不适用于传统数据库。

探针可以实现的下一个机制是使用在运行应用程序的容器内部执行的命令。这可以是任何可以在容器中的命令行运行的命令。如果命令的退出状态为 0，则探测将被视为成功。

要将容器命令注册为探针，可以针对部署配置运行 `oc set` 探针命令。根据您要设置的探针类型，使用 `--readiness` 和/或 `--liveness` 选项。

然后提供 - 后面的命令来运行：

```
$ oc set probe dc/blog --liveness -- powershift image alive
```

建议不要让命令太复杂。让命令运行随应用程序提供的脚本，并在该脚本中嵌入任何检查。这是因为如果检查的细节是命令的一部分（即 OpenShift 中部署配置的一部分），那么如果您使用的是 Git，那么它将不会与您的应用程序源代码一起版本化。

在应用程序源代码中使用脚本还可以更轻松地更改检查的功能，而无需在部署应用程序的新版本的同时更新部署配置。

对于数据库应用程序，探测脚本可以运行数据库客户端命令来检查数据库是否准备就绪。对于 Web 应用程序，探测脚本可以通过与 `$HOSTNAME: 8080` 建立连接来检查 Web 应用程序是否正常。`$HOSTNAME` 环境变量将自动设置为容器作为该容器的主机名。该端口将需要 Web 应用程序正在接受 HTTP 请求连接的端口。

如果在应用程序映像中没有包含合适的探针脚本，并且所需的检查过于复杂而无法在实际命令中提供，则可以使用配置映射来保存脚本，并通过卷将其挂载到容器中。该命令然后可以从卷执行脚本。

16.5 使用套接字连接

可以实现探测的最终机制是仅检查应用程序是否接受新的套接字连接。没有实际的数据通过连接发送。这仅适用于应用程序只有在启动成功时才开始监听连接，并且接受新连接足以指示应用程序正常运行的情况。

要将套接字连接的成功注册为探针，可以针对部署配置运行 `oc set probe` 命令。根据您要设置的探针类型，使用 `--readiness` 和/或 `--liveness` 选项。然后提供带端口的 `--open-tcp` 选项作为参数：

```
$ oc set probe dc / blog --readiness --liveness --open-tcp 8080
```

16.6 探测频率和超时

要查看任何就绪或活性探测的详细信息，可以在部署配置上运行 `oc` 描述。将显示检查运行时间和容错情况的详细信息：

```
Liveness: exec [powershift image alive] delay=0s timeout=1s period=10s
```

```
#success=1 #failure=3
```

```
Readiness: http-get http://:8080/healthz/ready delay=0s timeout=1s period=10s
```

```
#success=1 #failure=3
```

每个探针的设置是：

（`delay`）延迟

在使用探头进行第一次检查之前，吊舱已经启动多久

（`timeout`）时间到

探测器在被认为失败之前需要多长时间才能得到结果

（`period`）期

在下次检查运行之前，使用探针进行等待的前一次检查需要多长时间

`#(success)`成功

探针被视为已通过所需的成功检查数量

`#(failure)`失败

探针被视为失败所需的连续检查失败次数

这些默认设置可能不适用于所有应用程序。在添加探针时要覆盖默认设置，可以使用其他选项

传递给 `oc set` 探针。例如，要设置延迟，可以使用 `--initial-delay-seconds` 选项：

```
$ oc set probe dc / blog --readiness --get-url = http://: 8080 / healthz / ready \
--initial-delay-seconds 10
```

在启动期间，尽管应用程序可能接受新的请求连接，但可能无法立即开始处理实时请求，并且将返回 HTTP 错误响应，直到准备就绪，则需要设置初始延迟。

超时值可通过传递 `--timeout-seconds` 选项来覆盖，如果应用程序在默认的一秒内不能始终可靠地响应成功，则需要覆盖超时值。

如果 OpenShift 使用 `containerd` 在容器中运行图像，则使用容器命令时探测器的超时未实现。这是由于 `containerd` 和 `docker` 的实现受到限制。对于容器命令，建议探测器脚本在设定时间后执行自己的探测器失败机制。如果没有完成并且探针永不返回，则探针不会失败，后续探针也不会运行。

16.7 总结

健康探测器使您能够让 OpenShift 监控您的应用程序的健康状况。探针有两种形式，准备探针和活性探针。

准备就绪探测器确定您的应用程序实例是否准备好处理流量，并且应该包含在您的服务的活动实例列表中。当您的应用程序的新实例启动时，准备就绪探测还用于最初确定应用程序实例是否正确启动以及部署是成功还是失败。

活力探测器确定您的应用程序是否正常运行。如果应用程序的活动探测失败，那么该实例将被关闭并在其位置创建一个新实例。

第 17 章.应用程序生命周期管理

当您的应用程序正在部署，扩展，重新部署或关闭时，将按照一组规定的步骤完成。正如您在上一章中看到的那样，可以使用运行状况探测来确定应用程序是否成功启动，是否准备好接受请求并继续正常运行。不过，这些只是正在使用的部署策略所规定的更大流程的一部分。

OpenShift 实现了两个基本的部署策略。默认的部署策略是 Rolling 部署。此部署策略的目标是在向应用程序推出更新时启用零宕机时间部署。

第二种部署策略是重新创建。当您不能同时运行多个应用程序的实例或版本时，可以使用它。当使用一次只能绑定到群集中的单个节点的持久性存储时，也需要使用此部署策略。

在本章中，您将学习更多关于这些部署策略的知识，以及如何定义生命周期钩子，以及在部署中设定点执行的特殊命令。

您还将了解可能与单个 Pod 的启动和关闭相关的挂钩。

17.1 部署策略

部署策略定义了启动应用程序的新版本并关闭现有实例的过程。

要查看部署正在使用的策略，请在部署配置上运行 `oc describe` 命令：

策略：滚动

在编写本书时提供的 OpenShift 版本没有提供专门的命令来改变部署策略。在较早版本的 Open-Shift 中，您需要使用 `oc` 编辑来编辑部署配置，或者运行 `oc` 修补程序来修补部署配置：

```
$ oc patch dc / blog --patch '{ "spec": { "strategy": { "type": "Recreate" } } }
```

新的命令 `oc set deployment-strategy` 正在计划用于较新版本的 OpenShift，它将允许您将部署策略设置为通过运行重新创建：

```
$ oc set deployment-strategy dc / blog --recreate
```

要使用此命令将部署策略设置为滚动，您可以使用 - 滚动选项。

17.2 滚动部署

滚动部署使用新版本应用程序的实例慢慢替换了先前版本的应用程序的实例。在滚动部署中，在将其 IP 地址添加到服务端点列表并缩减旧实例之前，将针对新实例运行检查（使用准备就绪探针）以确定是否准备好接受请求。如果发生重大问题，可以中止滚动部署。

这是 OpenShift 中的默认部署策略。因为它可以并行运行应用程序的新旧实例，并在新实例部署和旧实例关闭时平衡流量，从而实现无需停机的更新。

在以下情况下不应该使用滚动部署：

- 当新版本的应用程序代码不能与单独数据库的现有模式一起工作时，首先需要进行数据库迁移
- 由于除了依赖单独的数据库模式版本之外的原因，新旧应用程序代码不能同时运行
- 如果同时运行多个应用程序实例并不安全，即使是同一版本

在这些情况下，应该使用重新创建部署策略。

要在部署进行时启用其他操作，您可以定义两种类型的生命周期挂钩：

Pre

此钩子在为新部署创建应用程序的第一个新实例之前执行。

Post

此钩子在您的应用程序的所有实例已成功启动并且旧实例已关闭后执行。

将使用新版本的图像创建容器，并为每个钩子指定的命令将在其中运行。

预生命周期钩子中的命令的一个示例是启用数据库标志以将应用程序置于只读模式。部署成功完成后，后期生命周期挂钩可以禁用该标志。

在部署失败的情况下，后期生命周期挂钩将不会运行。在这个例子中，这意味着网站将保持只读模式，直到已经采取手动操作来确定失败和标志被禁用。

要添加生命周期钩子，请在部署配置中运行 `oc set deployment-hook` 命令。传递 `--pre` 或 `--post` 选项来指示应该添加哪个生命周期钩子。命令应该遵循 -：

```
$ oc set deployment-hook dc / blog --pre - powershift image jobs pre-deployment
```

```
$ oc set deployment-hook dc / blog --post - powershift image jobs post-deployment
```

默认情况下，不会将卷装入运行生命周期钩子的容器中。要指定要从部署配置装入的卷，请使用 `--volumes` 选项并提供卷的名称。

为部署配置中的容器指定的任何环境变量都将传递到用于运行挂接命令的容器。如果您需要为钩子设置其他环境变量，请使用 `--environment` 选项。

要移除生命周期挂钩，请使用 `--remove` 选项：

```
$ set deployment-hook dc / blog --remove --pre --post
```

17.3 重新创建部署

与其在运行旧实例时启动应用程序的新实例，当使用重新创建策略时，应用程序的现有运行实例将首先关闭。只有在应用程序的所有旧实例都关闭后，才会启动新实例。

如果您不能同时运行多个应用程序实例，或者无法同时使用旧代码和新代码运行实例，则应该使用此部署策略。

它也应该用于您的应用程序使用持久性卷并且持久性卷的类型为 `ReadWriteOnce` 的情

况。这是必需的，因为此类持久性卷一次只能连接到 OpenShift 群集中的一个节点。

如果您在使用 Rolling 部署策略的同时使用 ReadWriteOnce 类型的持久卷，导致部署停滞，请在切换到重新创建部署之前将应用程序的副本数减少到零。一旦应用程序停止并且部署策略发生变化，请将副本数恢复为 1。这将避免需要等待卡住的部署超时。

有了这种策略，由于在使用新代码部署实例之前应用程序的所有旧实例都将停止，因此可能会有一段时间您的应用程序不可用。除非您已采取措施将流量首先路由到显示维护页面的临时应用程序，否则用户将看到 HTTP 503 服务不可用错误响应。

使用此部署策略时可以定义生命周期前期和后期挂钩，以及以下附加挂钩：

Mid

在应用程序的所有旧实例已关闭之后，但在您的应用程序的任何新实例已启动之前执行此挂钩。

Mid 挂钩可用于安全地运行任何数据库迁移，因为您的应用程序不会在此时运行。

要添加中间生命周期挂钩，请使用 `--mid` 选项在部署配置上运行 `oc set deployment-hook` 命令：

```
$ oc set deployment-hook dc/blog --mid -- powershift image migrate
```

17.4 自定义部署

除了这两种基本的部署策略之外，还可以实施其他各种更复杂的策略，包括蓝绿或 A / B 部署策略。这些可以通过使用多个部署配置并重新配置应用程序的哪个窗格与使用标签的服务或路线相关联来实现。可以手动设置这些自定义策略，也可以提供嵌入处理部署逻辑的映像，并通过 REST API 与 OpenShift 交互以对与应用程序相关的资源进行更改。有关更多信息，请查看关于 [部署策略](#) 的 OpenShift 文档。

17.5 容器运行时钩子

无论您的应用程序的副本数量多少，pre，mid 和 post 钩子都只针对每个部署执行一次。它们只能用于运行与整体部署相关的其他操作，而不能用于特定的 Pod。

要对每个吊舱执行特殊操作，可以定义以下吊钩：

poststart

该钩子在创建容器后立即执行。如果失败，集装箱将根据其重新启动策略终止并重新启动。

prestop

该钩子在容器关闭之前立即被调用。挂钩完成后，容器关闭。无论钩子的结果如何，容器仍将被关闭。

钩子可用于触发将数据预加载到应用程序特定实例使用的缓存中。钩子可以用来表示应用程序正常关闭的方式，允许更多地控制应用程序完成当前请求的时间。

这些钩子可以被实现为在应用程序的容器中运行的命令，或者作为针对在 pod 中运行的应用程序的 HTTP GET 请求，或者只是创建到应用程序的套接字连接的动作可以用于触发某个动作。

在编写本书时可用的 OpenShift 版本没有提供专门的命令来设置容器挂钩。有必要在部署配置上使用 `oc` 编辑，或者使用 `oc patch` 命令使用 JSON 样式的修补程序。

要使用 `oc patch` 为 preStop 挂钩设置要执行的命令，请创建一个 patch 文件 `pre-stop.json`，

其中包含：

```
[
{
  "op": "add",
  "path": "/ spec / template / spec / containers / 0 / lifecycle / preStop / exec / command",
  "value": [ "powershift", "image", "jobs", "graceful-shutdown" ]
}]
```

然后运行 `oc patch` 命令，使用补丁文件作为 `--patch` 选项的输入：

```
$ oc patch dc / blog --type = json --patch "`cat pre-stop.json`"
```

补丁文件中的路径属性指示部署配置中正在设置的值。容器后的 `0` 表示为部署配置定义的第一个容器。如果在容器中运行多个容器，则可能需要更改此号码。

17.6 初始容器

容器挂钩允许为每个 `pod` 实例执行动作，但它们只能在启动后或关闭之前立即运行在已运行的 `pod` 上。对于希望在应用程序启动之前执行操作的情况，您可以使用 `init` 容器。

`init` 容器就像一个容器中的常规容器，只是它们按顺序运行，并且必须在下一个 `init` 容器运行之前成功完成，最后启动应用程序。可以使用 `init` 容器来使用用于该应用程序的相同图像来运行命令，或者可以使用不同的图像。

如何使用 `init` 容器的一个示例是更新应用程序映像中的目录中持久卷中的文件，并将持久卷安装在同一目录之上。

这将允许持久卷使用来自映像的文件填充，但也允许通过应用程序或手动将其他文件添加到目录中，这会在重新启动应用程序时持续存在。其他文件可能会在以后的日期被合并到图像中，以便它们可用于全新的部署。

对于这种用例，您首先会将持久性卷挂载到应用程序容器中：

```
$ oc set volume dc / blog --add \
--type = pvc --claim-size = 1G --claim-mode = ReadWriteOnce \
--claim-name blog-htdocs --name htdocs --mount-path / opt / app-root / src / htdocs
```

此时，持久卷将隐藏最初包含在挂载持久卷的映像中的文件，并且该目录将显示为空。

要使用 `init` 容器将应用程序映像中的文件复制到永久卷中，请创建一个名为 `init-containers.json` 的修补程序文件，其中包含：

```
[
{
  "op": "add",
  "path": "/spec/template/spec/initContainers",
  "value": [
    {
      "name": "blog-htdocs-init",
      "image": "blog",
      "volumeMounts": [
        {
          "mountPath": "/mnt",
          "name": "htdocs"
        }
      ]
    }
  ]
}]
```

```

        }
    ],
    "command": [ "rsync",
        "--archive",
        "--no-perms",
        "--no-times",
        "/opt/app-root/src/htdocs/",
        "/mnt/"
    ]
}
]
}
]

```

然后运行 `oc patch` 命令，使用补丁文件作为 `--patch` 选项的输入：

```
$ oc patch dc / blog --type = json --patch “`cat init-containers.json”
```

`init` 容器将挂载持久性卷，但它将暂时挂载到目录 `/mnt` 上。然后可以将文件从应用程序映像复制到持久卷。当主应用程序容器运行时，持久卷将被挂载到从中复制文件的目录。在完成之前需要运行一个额外命令是：

```
$ oc set image-lookup dc / blog
```

这是必要的，因为，与容器中的常规容器不同，当在映像字段中使用包含图像流名称的 `init` 容器时，不会自动导致对照图像流解析该字段。`oc set image-lookup` 命令启用本地查找，因此名称将针对应用程序映像的图像流进行解析。

17.7 总结

部署策略定义启动应用程序的新版本并关闭现有实例的过程。两种主要的部署策略是滚动和重新创建。

对于 **Rolling** 部署，对于应用程序的每个实例，将在旧实例关闭之前启动新实例。这确保了应用程序总是有一个实例在运行，并且不会发生停机。

对于重新创建，应用程序的所有实例将在任何新实例创建之前关闭。部署发生时，您的应用程序的用户将看到它在一段时间内不可用。

可以使用哪种部署策略取决于应用程序的实现方式以及应用程序使用的持久性存储的类型。

在整个部署流程中，可以定义挂钩，以便在部署之前，部署期间或之后运行其他操作。还可以为应用程序的每个实例运行其他操作。

第 18 章.记录，监视和调试

开发和部署应用程序可能并不总是顺利。因此了解 **OpenShift** 提供哪些工具可帮助您调试问题非常重要。

计算出发生的主要方法是在构建，部署和运行应用程序时生成的日志。这些由 **OpenShift** 生成的系统事件补充。您还可以监视资源对象的更改。

要调试正在运行的应用程序，可以启动在应用程序容器内运行的交互式终端会话。为

了调试你的应用程序可能没有启动的原因，你可以用你的应用程序映像创建一个特殊的容器，而不是自动运行应用程序，你会得到一个交互式 `shell`，你可以自己启动它。

在本章中，您将学习如何访问日志，如何使用交互式 `shell` 与应用程序进行交互，以及用于监视应用程序的其他方法。

18.1 查看构建日志

当您在 OpenShift 中使用源代码构建应用程序时，无论是使用 `S2I` 构建器还是从 `Dockerfile` 构建应用程序，构建过程都由构建窗格协调。构建过程中的任何日志输出都将在此构建窗格中捕获。

要查看上次构建运行的构建日志，可以针对应用程序的构建配置运行 `oc` 日志命令：

```
$ oc logs bc/blog Receiving source from STDIN as archive ... Pulling image
"registry.access.redhat.com/rhsccl/python-35-rhel7@sha256:..." ... Collecting
powershift-cli[image]
```

```
Downloading powershift-cli-1.2.5.tar.gz
```

```
....
```

如果构建仍在运行，您可以提供 `--follow` 选项以使构建完成。

要调试构建过程本身，可以为构建配置定义 `BUILD_LOGLEVEL` 环境变量。这将导致 OpenShift 记录它正在做什么的消息：

```
$ oc set env bc / blog BUILD_LOGLEVEL = 9
```

为了查看先前构建的日志，首先使用 `oc get build` 来获取构建的列表，如果将多个应用程序部署到同一个项目，则提供一个标签选择器：

```
$ oc get builds --selector app = blog
$ oc get builds --selector app=blog
```

NAME	TYPE	FROM	STATUS	STARTED	DURATION
blog-2	Source	Git@9d745d3	Complete	2 minutes ago	59s
blog-1	Source	Git@b6e9504	Complete	5 minutes ago	1m37s

然后，您可以针对特定版本运行 `oc` 日志：

```
$ oc logs build / blog-1
```

您还可以运行 `oc` 获取窗格以查看构建窗格中所有窗格的列表并运行 `oc` 日志：

```
$ oc logs build/blog-1 --build
```

因为 OpenShift 会清理旧的容器和构建，所以你不会有完整的构建历史。对于旧版本，虽然构建记录可能仍然存在，但容器和日志可能已被清除，并且尝试查看日志将导致错误。

18.2 查看应用程序日志

要使用 `oc` 日志查看应用程序日志，您需要为应用程序的每个实例标识 `pod` 并在每个实例上运行 `oc` 日志：

```
$ oc get pods --selector app=blog
```

NAME	READY
------	-------

```
blog-2-116wg 1/1
blog-db-1-1bss8 1/1
$ oc logs pod/blog-5-116wg ...
```

```
STATUS  RESTARTS  AGE
Running 0      5m
Running 0      30m
```

您可以使用--follow 选项来记录日志以持续监视该日志的日志。

默认情况下，除非应用程序本身添加日志消息，否则不会显示单个日志消息的时间戳。要在日志消息旁边显示由 OpenShift 捕获的消息时间，请使用--timestamps 选项。在运行新部署时，您可以通过在应用程序的部署配置上运行 oc 日志来查看其进度：

```
$ oc logs --follow dc/blog
--> Scaling blog-2 down to zero
--> Scaling blog-3 to 1 before performing acceptance check
--> Waiting up to 10m0s for pods in rc blog-3 to become ready
--> Success
```

在其他时间，在部署配置上运行 oc 日志将导致显示最后一个 pod 的日志。

也可以在 Web 控制台中查看每个群集的日志，并且群集管理员为 OpenShift 群集启用了聚合日志记录时，您可以一起查看它们，并对日志执行查询。有关更多信息，请参阅关于[配置聚合日志记录](#)的 OpenShift 文档。

18.3 监视资源对象

无论是构建映像，执行新部署还是运行应用程序，OpenShift 中的所有工作通常都在一个 pod 中运行。您可以通过运行 oc get pod 来查看 pod 列表，但这只是当时存在的 pod 列表。如果您想要随时监控一组资源对象，则可以将--watch 选项传递给 oc 获取。这将允许您观看创建并随后终止的 Pod：

```
$ oc get pods --watch
```

NAME	READY	STATUS	RESTARTS	AGE
blog-3-5jkg0	1/1	Running	0	1m
blog-2-build	0/1	Pending	0	0s
blog-2-build	0/1	ContainerCreating	0	0s
blog-2-build	1/1	Running	0	6s
blog-3-deploy	0/1	Pending	0	0s
blog-3-deploy	0/1	ContainerCreating	0	0s
blog-2-build	0/1	Completed	0	1S
blog-3-deploy	1/1	Running	0	5S
blog-3-5jkg0	1/1	Terminating	0	2S

blog-3-5jkg0	0/1	Terminating	0	2S
blog-4-d6xbx	0/1	Pending	0	0S
blog-4-d6xbx	0/1	ContainerCreating	0	0s
blog-4-d6xbx	0/1	Running	0	8S
blog-4-d6xbx	1/1	Running	0	10S
blog-3-deploy	0/1	Completed	0	32S
blog-3-deploy	0/1	Terminating	0	32S

18.4 监视系统事件

观看窗格可以让您更好地了解新建构建和部署发生时系统正在发生的变化。您还可以查看运行应用程序实例的窗口何时终止并重新启动。监控吊舱不会告诉您什么是发生了什么事。

同样，日志可以告诉您构建或应用程序中发生了什么，包括错误，但它们不会告诉您 OpenShift 本身可能遇到的错误，或 OpenShift 为什么会采取特定操作。

为了监视 OpenShift 为项目中运行的应用程序执行的操作，可以使用 `oc` 获取事件，并通过 `--watch` 选项来监视发生的新事件。可以生成许多类型的事件。这些包括何时运行新版本，何时在容器停止和启动时运行部署，以及诸如探测失败或使用新部署超出资源配额等错误。

18.5 查看容器度量标准

要确定您使用的资源配额有多少，可以使用 Web 控制台中的配额页面。这里的信息会告诉你分配给应用程序的分配有多少，但它不是衡量应用程序实际使用的资源数量。

要确定应用程序的使用量，可以在 Web 控制台的项目概览中找到度量图表。这显示了您的应用程序的所有实例的平均使用情况（请参见图 18-1）。

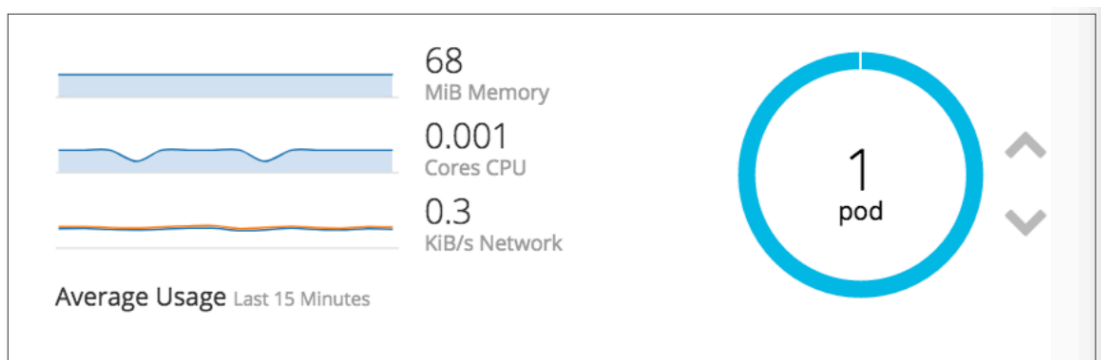


图 18-1 容器指标

单个窗格的度量标准可以在窗格的 Web 控制台页面上找到，也可以在项目概览的左侧导航栏中选择的“监视”页面下找到。

通过了解应用程序使用多少 CPU 或内存的知识，可以增加资源分配以确保资源有足够的资源，或者如果资源分配超过需求，可以减少资源分配，从而允许将资源分配给其他应用程序。

18.6 运行交互式 Shell

您的应用程序的每个实例都运行在一个容器中，从其他所有内容中删除。在容器中运行的唯一东西就是应用程序进程。

为了使用应用程序调试容器内发生的事情，可以访问容器并运行交互式 shell。为此，请针对 pod 的名称运行 `oc rsh`：

```
$ oc rsh blog-4-d6xbx
(app-root)sh-4.2$
```

这将适用于图像中包含 `/bin/sh` 的任何容器图像。调试应用程序时可以访问的 Unix 工具取决于映像中包含的内容。

在图像中包含诸如 `vi`，`curl`，`ps` 和 `top` 之类的工具总是一个好主意，以帮助更轻松地调试应用程序，检查或修改容器中的文件或附加的持久卷，或查看 CPU 和内存中运行的容器中的单个进程正在使用。

如果不是交互式 shell，而只想运行一个不需要输入的单个命令，请使用 `oc exec`：

```
$ oc exec blog-4-d6xbx env | grep
HOSTNAME HOSTNAME = blog-4-d6xbx
```

18.7 调试启动失败

如果您的应用程序在启动时失败，并且容器也被终止，OpenShift 将继续尝试重新启动它。如果这种情况持续发生，则部署将失败。OpenShift 会通过将该窗格的状态设置为 `CrashLoopBack Off` 来指示此情况。

要调试无法启动的容器，可以使用 `oc debug` 命令，根据应用程序的部署配置运行它：

```
$ oc debug dc / blog
```

使用 `pod / blog-debug` 进行调试，原始命令：

```
container-entrypoint /tmp/scripts/run
Waiting for pod to start ...
Pod IP: 10.131.1.193
```

如果您看不到命令提示符，请尝试按 `Enter` 键。

```
(app-root)sh-4.2$
```

与其启动应用程序，不会启动交互式 shell 会话。`oc` 调试运行时的启动消息将显示原始命令是否已经为容器运行。

在 shell 中，您可以验证任何环境变量或配置文件，如有必要，更改它们，然后运行原始命令以启动应用程序。您的命令的任何输出都将显示在终端中，以便您可以查看启动时可能发生的错误。

当您的应用程序以 `oc` 调试开始时，将无法使用其服务名称从任何其他 Pod 连接到该应用程序，也不会通过创建应用程序的路由公开它。如果您需要发送请求，请使用 `oc` 获取窗格获取为调试会话创建的窗格的名称，然后使用独立终端的 `oc rsh` 在容器中获取第二个交互式 shell。然后，您可以在容器内对应用程序运行一个命令，例如 `curl`：

```
$ oc get pods
NAME          READY   STATUS    RESTARTS  AGE
blog-4-d6xbx  1/1     Running   0          1h
blog-debug    1/1     Running   0          6m
```

```
$ oc rsh blog-debug
(app-root)sh-4.2$ curl $HOSTNAME:8080
```

...

由于在部署配置中定义的任何持久卷也将被挂载，所以如果您使用的是持久卷类型的 `ReadWriteOnce`，则首先需要通过运行 `oc 缩放--replicas = 0`；否则，调试容器将无法同时挂载持久性卷，并且无法启动。

18.8 总结

OpenShift 捕获并记录应用程序的每个实例的输出。要直接与应用程序实例交互，可以在运行应用程序的容器中启动交互式终端会话。可以使用它来检查环境变量的设置，查看配置文件，查看已装入持久卷中文件的内容，或直接与组成正在运行的应用程序的进程进行交互。有关使用的资源的详细信息可以使用容器中的进程监视工具来确定，也可以通过查看由 **OpenShift** 收集的用于总体 **Pod** 使用情况的指标来确定。

日志也可用于涵盖构建应用程序映像并进行部署的步骤。您可以通过监视 **Pod** 来监视部署的进度

通过观察 **OpenShift** 产生的事件以及发生的错误来创建和销毁。启动应用程序时发生的错误可以通过在与您的应用程序具有相同环境启动的特殊 **Pod** 中启动调试会话来进行调查，但是提供了交互式 **shell** 而不是启动应用程序。