




## 基 础 篇



- 第 1 章 开源容器云概述
  - 第 2 章 初探 OpenShift 容器云
  - 第 3 章 OpenShift 架构探秘
  - 第 4 章 OpenShift 企业部署
- 

# 开源容器云概述

## 1.1 容器时代的 IT

进入 21 世纪，我们的社会和经济发生了巨大的变化，社会对各行业服务的要求越来越高、越来越细致。新的需求如洪水一样滔滔不绝地从市场的第一线喷涌到企业的产品部门和 IT 部门。企业想要在竞争中获取优势，就必须比竞争对手更快地把产品推出市场。为了缩短产品从概念到上市的时间，企业的各个流程和流程中的各个环节都要升级优化。企业在变革，企业 IT 自然不能独善其身，必须要跟上市场的节奏响应市场的需求。目标是明确的，就是速度要更快，成本要更低、质量要更好。但是现实的问题是，如何做到？

为了满足业务的要求，企业 IT 在不断地变革，而且从未停步。从客户端 / 服务器模型，变革为浏览器 / 服务端模型，从庞大的信息孤岛，变革为基于服务的架构（Service Oriented Architecture, SOA），从物理机，到虚拟化，再到基础架构云（Infrastructure as a Service, IaaS）和应用云（Platform as a Service, PaaS）。对比十几年前，如今 IT 的效率得到了极大的提升，尤其是进入云时代后，一切资源变得触手可及。以往应用上线需要的资源，从提出申请，到审批，到采购，到安装，再到部署往往需要至少几十天的时间。在云时代，这些事情往往在几天或几小时便可以准备到位。业界在还没有来得及对云的怀疑之声做出反击回应之前，云就已经征服了整个 IT 世界。

通过这些年云化的推进，大多数有一定规模的企业已经实现了基础架构资源的云化和池化，这里的资源指的是诸如虚拟机、数据库、网络、存储。用户可以用很短的时间获取业务应用所需的机器、存储和数据库。基础架构资源云化其实并不是目的，而是手段。最终的目标是让承载业务的应用可以更快地上线。但现实是，通过 IaaS 获取的大量基础架构资源并不能被我们的最终业务应用直接消费。应用还必须进行或繁或简的部署和配置，才可能运行

在云化的虚拟机之上。部署涉及操作系统配置的修改、编程语言运行环境的安装配置以及中间件的安装配置等。部署的过程在一些企业仍然是通过手工完成，低效且容易出错。有的企业则是通过简单的自动化方式完成，提高了效率，但是满足不了后期更高级别的需求，如动态扩容，持续部署。即使勉强通过简单的自动化实现，后期随着部署平台类型的增多及复杂化，维护的难度将会陡然提高，无法真正做到随时随地持续交付、部署。

基于这个背景，业界需要有一种手段来填充业务应用和基础架构资源的这道鸿沟。让应用可以做到“一键式”快速地在基础架构资源上运行。不管底层的基础架构资源是物理机、虚拟化平台、OpenStack、Amazon Web service，还是 Microsoft Azure，都能实现快速、顺畅地部署交付。为了实现这个目标，业界出现了多种不同的平台，即服务云的容器方案。最终命运之神的棒槌砸到了一个叫 Docker 的开源项目上。Docker 通过对 Linux 内核已有机能的整合和强化，为业务应用提供了一个可靠的隔离环境。此外，层叠式的 Docker 镜像为应用环境的复用提供了一个绝妙的方案。最后其简单易用的用户命令行，让 Docker 快速地获取了巨大的用户基础，也成就了今日其在容器界的地位。

通过容器这个手段，下一步就是实现应用在大规模云环境进行应用部署。在以往的软件业中，软件的交付件往往是软件的二进制安装部署包，比如 Java 的 WAR 包、Windows 的 EXE、Linux 的 RPM 包等。在容器时代，不难想象，未来软件的交付件将会以容器镜像作为载体。容器镜像中包含了软件应用本身，应用所依赖的操作系统配置、基础软件、中间件及配置。同时这些镜像将会设计得非常智能，能够自动获取依赖服务的相关信息，如网络 IP 地址、用户名、密钥等。在云的环境中部署这些应用，需要做的只是简单地启动容器镜像，实例化出相应的容器，然后业务应用快速启动，向最终用户提供服务。目前大量的企业正处于这个变革和转型的过程中。

随着容器成为了部署交付件的标准，大量的业务应用将会需要运行在容器环境中，或者换句话说，未来容器将会成为应用的标准运行环境。那么下一个问题就是，应用如何在容器的环境中运行得更高效、更稳定？为了更好地运行在容器环境中，应用的架构也必然要发生变化，变得契合容器的特性。正因为这个背景，最近，业界在热烈地探讨容器之余，也非常关注应用的微服务化。

如同第二次工业革命蒸汽机带来的冲击一样，容器给 IT 业界带来了巨大的冲击。面对这场变革，企业 IT 要做的不仅仅是技术的决策，而且是一个战略性的决策。企业要么主动拥抱它，要么等待来自竞争对手的压力后，再被动地接受并追赶其他的先行者。

## 1.2 开源容器云

如前文所述，为了响应快速变化的业务需求，IT 业界正在进行一场变革，在这场变革中，用户通过容器作为手段，在应用程序开发、测试、部署，在 IT 运维的各个环节进行方方面面的改进和提升。如同第二次工业革命，新技术的应用带来了生产效率和生产力的提升，意味着顺应变革的企业会有更强的竞争能力。它们的应用能更快地上线，想法能更快地变成现实，

变成企业的现金收入。相反，没有拥抱新技术和变革的企业竞争能力将会快速下滑。通过对社区及国内的几场大型容器会议的观察，可以明显感觉到经过了这些年的发展，目前容器技术的使用已经是不可逆转的趋势。企业现在的关注重点已经不再停留于容器技术不可用，而转变到了如何使用容器，如何用好容器来提升自己 IT 的效率，提升企业的竞争能力。


既然决定要投入这场变革，拥抱新的技术，那么下一个问题就是：应该怎么做？通过 Docker 启动一个容器很简单，但是要管理好千千万万个容器，需要的不仅仅是热情和勇气。我们需要回答许多问题，如容器镜像从哪里来？怎么保证容器运行环境的安全？如何进行容器的调度？多主机上的容器如何通信？容器的持久化数据怎么解决？处理好这些问题，需要有切切实实可以落地的方案。一个企业要自行解决所有的这些问题，可以说是不可能完成的任务，其需要投入的人力、物力和时间成本，不是单纯一个企业可以接受的。通过现有的技术或平台快速构建企业自有的容器平台，从经济成本及技术难度角度考量，可以说是更为符合现状的合理选择。

现代容器技术的根据地是开源社区。开源社区提供了一个活跃的舞台，这个舞台凝聚来自世界各地的企业、团队及个人。可以说目前开源社区是 IT 行业创新发生最高度密集的地方。开源软件目前被应用在 IT 行业的方方面面，如我们的开发工具、编程语言、编程框架、中间件、数据库、操作系统、储存、网络、云等。通过开源社区的技术，完全可以构建出一个稳定可靠的企业 IT 技术堆栈。现今企业要基于已有的解决方案构建自有的容器云平台，我认为，开源的容器云平台是一个必然的选择。

### 1.3 OpenShift



图 1-1 容器云 OpenShift 开源项目主页

 提示 OpenShift Origin 项目主页：<http://www.openshift.org>。

OpenShift 是一个开源容器云平台，是一个基于主流的容器技术 Docker 及 Kubernetes 构建的云平台。作为一个开源项目，OpenShift 已有 5 年的发展历史，其最早的定位是一个应用云平台（Platform as a Service, PaaS）。在 Docker 时代来临之前，各个厂商和社区项目倾向构建自己的容器标准，如 CloudFoundry 的 Warden、OpenShift 的 Gear，但是在 Docker 成为主流及社区的技术发展方向后，OpenShift 快速地拥抱了 Docker，并推出了市场上第一个基于 Docker 及 Kubernetes 的容器 PaaS 解决方案。OpenShift 对 Docker 及 Kubernetes 的整合和 OpenShift 项目最大的贡献方红帽公司（Red Hat Inc.）有着很大的关系。Red Hat 对于 Linux 和开源爱好者而言不用过多的介绍，在某个时代，Red Hat 几乎成为了 Linux 的代名词，它是目前世界上最大的开源软件公司，是开源社区的领导者。Red Hat 是 OpenShift 项目最大的贡献者，同时也是 Docker 和 Kubernetes 项目重要的贡献方。正是 Red Hat 对社区技术发展的敏锐触觉促成了 OpenShift 与 Docker 及 Kubernetes 的整合。事实证明这个决定非常明智。OpenShift 前几年在容器和 PaaS 领域的经验积累，叠加上 Docker 和 Kubernetes 容器及容器编排上的特性，一经推出就受到了广泛的关注和好评，连续两年获得 InfoWorld 年度技术创新大奖。

通过 OpenShift 这个平台，企业可以快速在内部网络中构建出一个多租户的云平台，在这朵云上提供应用开发、测试、部署、运维的各项服务（如图 1-2 所示）。OpenShift 在一个平台上贯通开发、测试、部署、运维的流程，实现高度的自动化，满足应用持续集成及持续交付和部署的需求；满足企业及组织对容器管理、容器编排的需求。通过 OpenShift 的灵活架构，企业可以以 OpenShift 作为核心，在其上搭建一个企业的 DevOps 引擎，推动企业的 DevOps 变革和转型。



图 1-2 OpenShift 上运行的容器应用



## 1.4 Docker、Kubernetes 与 OpenShift

许多刚接触 OpenShift 的朋友会有这样一个疑问：“Open-Shift 与 Docker 及 Kubernetes 的关系究竟是什么？”OpenShift 是基于容器技术构建的一个云平台。这里所指的容器技术即包含 Docker 及 Kubernetes。如图 1-3 所示，OpenShift 底层以 Docker 作为容器引擎驱动，以 Kubernetes 作为容器编排引擎组件。OpenShift 提供了开发语言、中间件、自动化流程工具及界面等元素，提供了一套完整的基于容器的应用云平台。



图 1-3 OpenShift 的技术堆栈

### 1.4.1 容器引擎

Docker 的优势在于它可以构建一个隔离的、稳定的、安全的、高性能的容器运行环境。目前，OpenShift 使用原生的 Docker 作为平台的容器引擎，为上层组件及用户应用提供可靠安全的运行环境具有十分重要的价值：

- ❑ Docker 有非常大的用户基础。以 Docker 为基础引擎，降低了用户学习的成本。熟悉 Docker 的用户可以非常容易地上手。
- ❑ Docker Hub 上有海量的镜像资源。我们日常使用的绝大部分软件，都可以在 Docker-Hub 上找到官方的或社区贡献的镜像。所有的这些镜像都可以无缝地运行在 OpenShift 平台上。
- ❑ Red Hat 本身就是 Docker 的一个主要贡献者，它们对社区有着很强的影响力，对这个技术的发展也有着很强的领导力。这一点对企业用户来说非常关键，因为谁也不想投资在一个没有前景或过时的技术上。

这里值得关注的一点是 OpenShift 使用的 Docker 是原生的 Docker，没有任何闭源的修改。因为历史的原因，有些应用云平台如 CloudFoundry 的选择是兼容 Docker 的。通过拷贝 Docker 的部分源代码加入它们的容器引擎中，以读取 Docker 镜像的内容，然后启动一个非 Docker 的容器实例。这种兼容的做法，我个人认为是值得商榷的。这让我联想起了当年安卓崛起后，为了挽回颓势，黑莓手机 (BlackBerry) 推出在自家系统中兼容运行安卓应用的做法。作为曾经黑莓 Z10 的用户，我非常喜欢那款精致的手机，但不得不说，在黑莓系统上运行安卓应用，简直就是一个噩梦。

### 1.4.2 容器编排

Docker 的流行使得当下每每提起容器时，大家更容易想到的是 Docker，甚至说是只有 Docker。但是现实是，Docker 其实只是容器技术中的一个点。Docker 是一款非常优秀和受欢迎的容器引擎，但是当企业或者某一个组织要大规模地将容器技术应用到生产中时，除了

有优秀的容器引擎提供稳定可靠及高效的运行环境之外，还需要考虑集群管理、高可用、安全、持续集成等方方面面的问题。单凭一个容器引擎，并不能满足容器技术在生产环境中的需求，尤其是规模较大的生产环境。

在大规模的容器部署环境中，往往涉及成百上千台物理机或者运行于 IaaS 之上的虚拟机。面对数量庞大的机器集群，用户面临着巨大的管理挑战。举个简单的例子，假设我们需要在 100 台机器上启动 100 个容器实例，通过手工的方式在 100 台机器上执行 `docker run` 命令将会是一件疯狂的事情。又比如，我们希望 20 个容器部署在美国机房、20 个容器部署在上海机房、20 个容器部署在深圳机房有 SSD 的服务器上，20 个容器部署在深圳机房带万兆网卡的机器上，通过人工或者传统的自动化工具来实现复杂的部署需求将会十分低效。现实是，为了满足容器集群所需的调度、网络、储存、性能及安全的需求，我们必须有专业的工具和平台。这些关于容器集群管理的问题，其实就是容器编排的问题，即 Kubernetes 要解决的问题。

Kubernetes 是 Google 十多年容器使用经验的总结，虽然 Google 使用的容器是 Docker 时代之前的容器，但是业务应用对安全、性能、隔离、网络、储存及调度方面的需求，在最原始的本质其实并没有发生变化。Google 选择和 Red Hat 一同开源了 Kubernetes，且目前在 GitHub 上的关注程度远远高于其他同类的平台，未来非常可能在容器编排领域成为类似 Docker 一样的“事实标准”。



提示 Kubernetes 项目 GitHub 仓库：<https://github.com/kubernetes>。

OpenShift 集成了原生的 Kubernetes 作为容器编排组件。OpenShift 通过 Kubernetes 来管理容器集群中的机器节点及容器，为业务应用提供：

- ❑ 容器调度：按业务的要求快速部署容器至指定的目标。
- ❑ 弹性伸缩：按业务的需要快速扩展或收缩容器的运行实例数量。
- ❑ 异常自愈：当容器实例发生异常，集群能自动感知、处理并恢复服务状态。
- ❑ 持久化卷：为散布在集群不同机器上的容器提供持久化卷的智能对接。
- ❑ 服务发现：为业务微服务化提供服务发现及负载均衡等功能。
- ❑ 配置管理：为业务应用提供灵活的配置管理及分发规则。

### 1.4.3 容器应用云

前文谈到了容器引擎及容器编排，这两项是容器技术的重要基石。掌握这两个基石，用户就具备了运维大规模容器集群的能力。现实中用户考虑使用容器应用平台的一个最终的目的就是提高生产效率。容器引擎及容器编排组件是两项关键的技术，但是光有技术还不能满足生产效率的要求。在这些技术及框架的基础上，必须有更丰富的内容以及更友好的用户接入方式，把这些技术转化成实实在在的生产力。

OpenShift 在 Docker 和 Kubernetes 的基础上提供了各种功能，以满足业务应用、研发用户及运维用户在生产效率上的诉求。

- ❑ 应用开发框架及中间件。OpenShift 提供了丰富的开箱即用的编程开发框架及中间件，如 Java、PHP、Ruby、Python、JBoss EAP、Tomcat、MySQL、MongoDB 及 JBoss 系列中间件等。
- ❑ 应用及服务目录。OpenShift 提供了如软件市场式的服务及应用目录，可以实现用户一键部署各类应用及服务，比如一键部署 Hadoop 集群和 Spark 集群。
- ❑ 自动化流程及工具。OpenShift 内置了自动化流程工具 S2I (Source to Image)，帮助用户自动化完成代码的编译、构建及镜像发布。
- ❑ 软件自定义网络。通过 OpenVSwitch，OpenShift 为用户提供了灵活强健的软件定义网络。实现跨主机共享网络及多租户隔离网络模式。
- ❑ 性能监控及日志管理。OpenShift 提供了开箱可用的性能监控及日志管理的组件。通过平台，业务能快速获取运行状态指标，对业务日志进行收集及分析。
- ❑ 多用户接口。OpenShift 提供了友好的 Web 用户界面、命令行工具及 RESTful API。
- ❑ 自动化集群部署及管理。OpenShift 通过 Ansible 实现了集群的自动化部署，为集群的自动化扩容提供了接口。

通过前面的介绍，我们可以了解到 OpenShift 在 Docker 及 Kubernetes 的基础上做了方方面面的创新，最终目的就是为用户及业务应用提供一个高效、高生产力的平台。

## 1.5 OpenShift 社区版与企业版

OpenShift 是一个开源项目，所有的源代码都可以在 GitHub 仓库上查阅及下载。企业和个人都可以免费下载和使用 OpenShift 构建属于自己的容器云平台。我们也可以加入 OpenShift 的社区成为一名光荣的 OpenShift 社区贡献者。



提示

❑ OpenShift 项目主页：<https://www.openshift.org>。

❑ OpenShift GitHub 仓库：<https://github.com/openshift>。

开源软件的一大好处在于，用户可以自由选择 and 免费使用。缺点是没有会对软件的使用提供支持保障。对于个人用户来说，这不是问题。但是对于企业来说，更多是希望有人能在出现问题的时候提供专业的支持和保障。这种需求给一些公司提供了机会，他们在开源软件的基础上进行定制、测试、修复及优化，推出企业版本，并对之进行支持。这种软件称为开源商业软件。Red Hat 就是开源软件商业模式的奠基人，而且是目前世界上最大的开源软件公司。OpenShift 的开源社区版本叫 OpenShift Origin，Red Hat 在 OpenShift Origin 的基础上推出了 OpenShift 的企业版本，其中包含了公有云服务 OpenShift Online 及私有云产品



OpenShift Container Platform (以前也称为 OpenShift Enterprise)。更多关于 OpenShift 企业版的信息可以访问 OpenShift 企业版的主页：<http://www.openshift.com>。

OpenShift 的企业版和社区版在代码上十分相似，功能上可以说是基本一致。企业版是基于某个社区版本产生的。作为一个开源软件公司，Red Hat 所有产品的企业版的源代码也是完全公开的。就我个人的经验而言，企业版往往会更稳定，因为社区版的代码变化会更频繁。

经常会被问到这样一个问题：“究竟我们是使用社区版还是企业版比较好？”这个问题没有唯一的答案，要视用户所处的使用场景而言。对于个人用户的开发测试及出于研究目的而言，社区版会是一个不错的选择，但是其实 OpenShift 的企业版也对个人用户免费开放。对于企业的关键业务应用的部署而言，企业版自然会是更好的选择，企业版除了稳定以外，还有专业的售后支持。

目前一些 OpenShift 的企业客户，在使用 OpenShift 企业版的同时，也会将他们的需求以提案或代码的方式提交到社区，在被社区评审接纳以后融入成为 OpenShift 产品未来版本的核心特性。这样做的好处是，企业所需的功能往后就由社区进行维护，不存在如自定义的修改在未来版本还需要进行测试及匹配，从而带来不可预知的工作量和风险。企业用户参与到开源社区，可以对产品的发展方向发表自己的看法。从而避免在传统的闭源商业软件时代用户只能被厂商牵着鼻子走的窘境。

本书的内容将以 OpenShift 社区版 OpenShift Origin 进行探讨及讲解。相关的经验也适用于 OpenShift 的企业版 OpenShift Container Platform。在一般情况下，本书将直接使用 OpenShift 来指代 OpenShift 的社区版和企业版。在企业版和社区版存在差异的情况下，将会特别标注。

接下来让我们扬帆起航，开启一段通往企业容器云的旅程。

## 初探 OpenShift 容器云

在前面的章节我们一起探讨了容器云的概念以及 OpenShift 容器云项目的情况。理论联系实际，为了对 OpenShift 容器云有更直接的认识，本章将会帮助你快速地在自己的个人电脑上搭建一个可用的 OpenShift 环境，并在这个环境上运行我们的第一个容器应用。这里假设你对 Linux 及 Docker 已经有了基本的了解，并掌握了基本的使用命令。对 Docker 不熟悉的读者，请参考 dockone.io 的 Docker 入门教程 (<http://dockone.io/article/111>)。

### 2.1 启动 OpenShift Origin

OpenShift 支持运行在基础架构之上，同时支持多种安装方式。

- ❑ 手工安装。用户下载 OpenShift 的二进制包，手动进行配置和启动。
- ❑ 快速安装。通过 OpenShift 提供的交互式 Installer 进行安装。
- ❑ 高级安装。在多节点集群的环境中，OpenShift 可通过 Ansible 对多台集群主机进行自动化安装和配置。
- ❑ Docker 镜像。通过运行 OpenShift 的 Docker 镜像启动一个 All-in-One 的 OpenShift 容器实例。这适合开发测试人员快速部署和验证。



**提示** OpenShift Origin 1.3.0 提供了一个全新的命令 `oc cluster up` 帮助开发用户快速启动一个可用的 OpenShift 集群。

为了尽可能了解 OpenShift 的细节，这里使用手动安装方式快速启动一个可用的 Open-

Shift Origin 实例，这个方法也适用于开发和测试。在实际的多节点集群环境中，OpenShift 的安装一般会通过高级安装完成，即通过 Ansible 完成。关于安装的更多信息可以参考 OpenShift Origin 的安装文档。

 **提示** OpenShift Origin 的安装文档：[https://docs.openshift.org/latest/install\\_config/index.html](https://docs.openshift.org/latest/install_config/index.html)。

2.1.1 准备主机


为了运行 OpenShift Origin，需要一台运行 Linux 操作系统的主机，可以是物理机或是虚拟机。为了试验方便，推荐使用虚拟机。KVM、VMWare 或 VirtualBox 的虚拟机均可。表 2-1 是推荐的最低配置。

表 2-1 主机硬件配置

CPU	内存	磁盘	网络
X86-64 1 核	2 GB	20 GB	IPV4

2.1.2 准备操作系统

请至 CentOS Linux 主页下载 CentOS Linux 7.2 的 ISO 镜像。如果镜像下载的网速太慢，也可从中国科技大学提供的开源镜像站点下载。

 **提示** ☐ CentOS 主页：<https://www.centos.org/download/>。  
☐ 中国科技大学镜像站点：[https://mirrors.ustc.edu.cn/centos/7.2.1511/isos/x86\\_64/](https://mirrors.ustc.edu.cn/centos/7.2.1511/isos/x86_64/)。

按提示为主机安装 CentOS 操作系统。安装时选择最小安装模式（Minimal 模式）。

2.1.3 操作系统配置

操作系统安装完毕后，以 root 用户登录系统。请确认主机已经获取了 IP 地址。本例中，笔者使用的主机的 IP 地址为 192.168.172.167。

```
[root@master ~]# ip a
1: lo: <LOOPBACK,UP,LOWER_UP>mtu 65536 qdiscnoqueue state UNKNOWN
link/loopback 00:00:00:00:00:00 brd 00:00:00:00:00:00
inet 127.0.0.1/8 scope host lo
valid_lft forever preferred_lft forever
inet6 ::1/128 scope host
valid_lft forever preferred_lft forever
2: eno16777736: <BROADCAST,MULTICAST,UP,LOWER_UP>mtu 1500 qdiscpfifo_fast state UP qlen 1000
link/ether 00:0c:29:df:3e:cb brdff:ff:ff:ff:ff:ff
inet 192.168.172.167/24 brd 192.168.172.255 scope global dynamic eno16777736
```

```
valid_lft 1286sec preferred_lft 1286sec
inet6 fe80::20c:29ff:fedf:3ecb/64 scope link
```

OpenShift 集群的正常运行需要一个可被解析的主机名。本例配置的主机名为 `master.example.com`。

```
[root@master ~]# hostnamectl set-hostname master.example.com
```

确认主机名是否能被正常解析。如不能解析，请修改 `/etc/hosts` 文件添加主机名的解析，将主机名指向实验主机的 IP 地址。

### 2.1.4 安装 Docker

OpenShift 平台使用的容器引擎为 Docker，因此需要安装 Docker 软件包。

```
[root@master ~]# yum install -y docker
```

安装完毕后启动 Docker 服务，并配置为开机自启动。

```
[root@master ~]# systemctl start docker
[root@master ~]# systemctl enable docker
```

由于国内访问 DockerHub 下载镜像的速度过于缓慢，可以使用中国科技大学的 Docker-Hub 镜像服务器进行加速。编辑 `/etc/sysconfig/docker` 文件，为 `DOCKER_OPTS` 变量追加参数 `--registry-mirror=https://docker.mirrors.ustc.edu.cn`。修改后变量值大致如下：

```
OPTIONS='--selinux-enabled --log-driver=journald --registry-mirror=https://docker.mirrors.ustc.edu.cn'
```

修改完 Docker 配置文件后，重启 Docker 进程使修改的配置生效。

```
[root@master ~]# systemctl restart docker
```

此时可以尝试运行一个 DockerHub 上的镜像，测试 Docker 是否正常工作。在下面的例子中，笔者运行了一个名为 `hello-openshift` 的镜像。这个镜像中运行了一个用 go 语言编写的小程序。如果一切正常，容器将成功启动，并监听 8080 及 8888 端口。

```
[root@master ~]# docker run -it openshift/hello-openshift
Unable to find image 'openshift/hello-openshift:latest' locally
Trying to pull repository docker.io/openshift/hello-openshift ...
latest: Pulling from docker.io/openshift/hello-openshift
a3ed95caeb02: Pull complete
a8a87b6280f5: Pull complete
Digest: sha256:fe89d47f566947617019a15eef50d97c8c20d6c9a5aba0d3cb45f84d2085e4e3
Status: Downloaded newer image for docker.io/openshift/hello-openshift:latest
serving on 8888
serving on 8080
```

测试完毕后，按 `Ctrl+c` 组合键停止运行中的容器。如果 Docker 无法找到容器镜像或

者出现了其他错误，则检查前序执行的安装配置步骤。

### 2.1.5 下载 OpenShift Origin 安装包

从 OpenShift Origin 的 GitHub 仓库中下载 OpenShift Origin 的二进制执行文件。本书示例所用的 OpenShift Origin 版本为 1.3.0，文件为

```
openshift-origin-server-v1.3.0-3ab7af3d097b57f933eccef684a714f2368804e7-linux-64bit.tar.gz
```

将下载好的 OpenShift 二进制包拷贝到主机的 /opt 目录下。



本例中使用的 OpenShift Origin 二进制执行文件包的下载地址如下：<https://github.com/openshift/origin/releases/download/v1.3.0/openshift-origin-server-v1.3.0-3ab7af3d097b57f933eccef684a714f2368804e7-linux-64bit.tar.gz>。

### 2.1.6 安装及启动 OpenShift Origin

进入 /opt 目录，解压下载好的 OpenShift Origin 二进制安装包。

```
[root@master /]# cd /opt/
[root@master opt]# tar zxvf openshift-origin-server-v1.3.0-3ab7af3d097b57f933eccef684a714f2368804e7-linux-64bit.tar.gz
[root@master opt]# ln -s openshift-origin-server-v1.3.0-3ab7af3d097b57f933eccef684a714f2368804e7-linux-64bit /opt/openshift
```

将 OpenShift 的相关命令追加至系统的 PATH 环境变量中。编辑 /etc/profile 文件，添加如下文本内容至文件末尾。

```
PATH=$PATH:/opt/openshift/
```

执行 source 命令使修改的配置生效。

```
[root@master opt]# source /etc/profile
```

修改完毕后，可以测试 Shell 能否找到 openshift 命令。执行 openshift version 命令查看当前 OpenShift 的版本。通过下面的输出可以看到当前使用的 OpenShift 版本是 1.3.0，搭配的 Kubernetes 的版本为 1.3.0，etcd 为 2.3.0。

```
[root@masteropenshift]# openshift version
openshift v1.3.0
kubernetes v1.3.0+52492b4
etcd 2.3.0+git
```

进入 /opt/openshift 目录。执行 openshift start 命令启动 OpenShift Origin。

```
[root@master opt]# cd /opt/openshift
[root@masteropenshift]# openshift start
```



命令执行后控制台将有日志输出，当日志输出停止后即表示 OpenShift 服务启动完毕。

### 2.1.7 登录 OpenShift Origin 控制台

OpenShift Origin 启动完毕后，在浏览器中访问网址 <https://master.example.com:8443>，即可看见 OpenShift 的 Web 控制台，如图 2-1 所示。

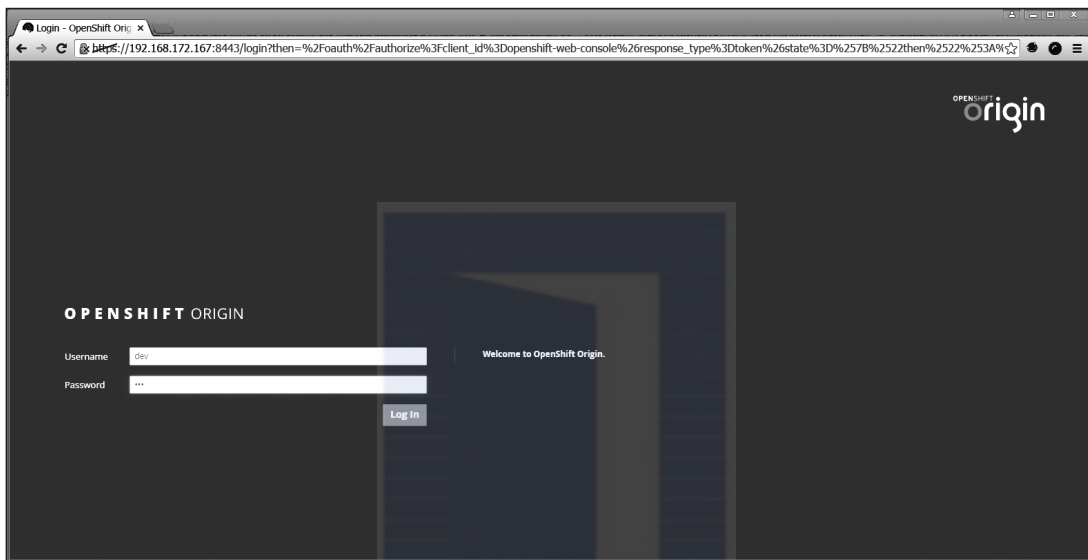


图 2-1 OpenShift Web 控制台登录界面



注意

如浏览器提示证书不可信，请忽略此告警并继续。推荐使用 Firefox 及 Chrome 浏览器访问 Web 控制台。

请使用系统用户 `dev` 登录，用户密码为 `dev`。成功登录后将可以看到 OpenShift 的欢迎页面，如图 2-2 所示。

恭喜您！您已经成功踏入了容器云的世界！

## 2.2 运行第一个容器应用

OpenShift 服务成功启动后，现在可以尝试运行你的第一个容器应用了！

### 2.2.1 创建项目

在部署应用前，先要为应用创建一个项目，即



图 2-2 OpenShift Origin Web 控制台欢迎  
页面

Project 对象。项目是 OpenShift 中的一种资源组织方式。对一般用户而言，不同类型的相关资源可以被归属到某一个项目中进行统一管理。对管理员来说，项目是配额管理和网络隔离的基本单位。

以 dev 用户登录 OpenShift 的 Web 控制台。单击页面中的 New Project 按钮创建一个新的项目。在创建项目页面输入项目名 hello-world，展示名称填入 Hello World。单击 Create 按钮创建项目，如图 2-3 所示。

图 2-3 创建 Hello World 项目

### 2.2.2 部署 Docker 镜像

现在马上可以部署你的第一个容器应用了。前文曾介绍到 OpenShift 是以原生的 Docker 作为平台的容器引擎，因此只要是有效的 Docker 镜像，均可以运行于 OpenShift 容器云平台之上。



**提示** Docker 默认允许容器以 root 用户的身份执行容器内的程序。OpenShift 对容器的安全比 Docker 有更谨慎的态度。OpenShift 默认在启动容器应用时使用非 root 用户。这可能会导致一些 Docker 镜像在 OpenShift 平台上启动时报出 Permission denied 的错误。别担心，其实只需要稍稍修改 OpenShift 的安全配置，即可解决这个问题。具体修改我们会在后面的章节介绍。但是请记住，在制作自己的 Docker 镜像时，建议避免使用 root 用户启动容器内的应用，以降低安全风险。

下面在 OpenShift 上运行 DockerHub 上的 hello-openshift 镜像。单击页面上方的 Deploy Image 页签，如图 2-4 所示。

在部署镜像页面，单击 Image Name 单选按钮，并输入镜像名称 openshift/hello-openshift，然后单击放大镜按钮，如图 2-5 所示。单击按钮后 OpenShift 将根据输入的镜像名称在 DockerHub 及配置了的镜像仓库中查找该名称的容器镜像。

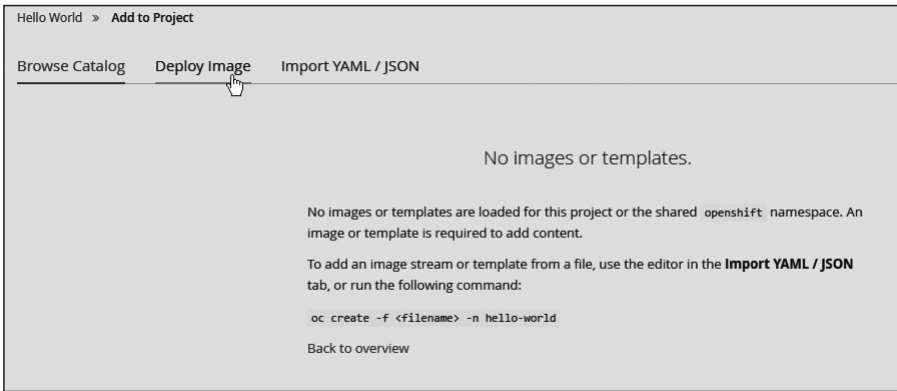


图 2-4 部署容器镜像

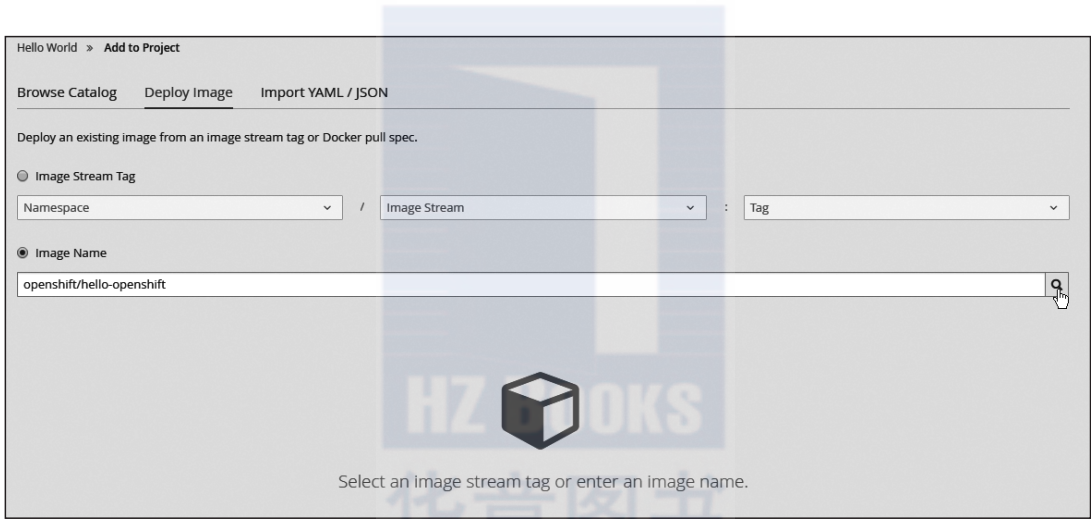


图 2-5 输入容器镜像名称



**注意** 请保证实验用的虚拟机能连接上互联网，以访问 DockerHub 仓库下载所需镜像。

片刻之后，OpenShift 将找到我们指定的镜像，并加载镜像的信息。浏览信息后，单击页面下方的 **Create** 按钮进行部署，如图 2-6 所示。此时 OpenShift 将会后台创建部署此容器镜像的相关对象。

确认部署后，页面将转跳到一个部署完成页面，如图 2-7 所示。单击页面上的 **Continue to overview** 链接转跳到 Hello World 项目的主页。

在 Hello World 项目的主页，你会看到界面上有一个空心的圆圈，圆圈中间有一个大大的数字 0，如图 2-8 所示。这表示 OpenShift 正在部署容器镜像并实例化容器，当前就绪的容器数量为 0。

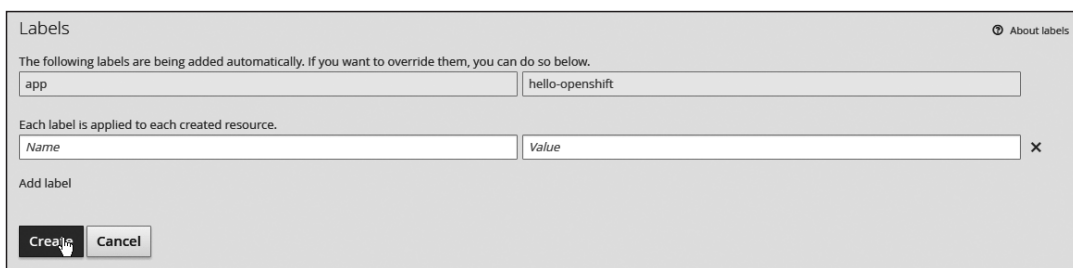


图 2-6 确认部署容器镜像

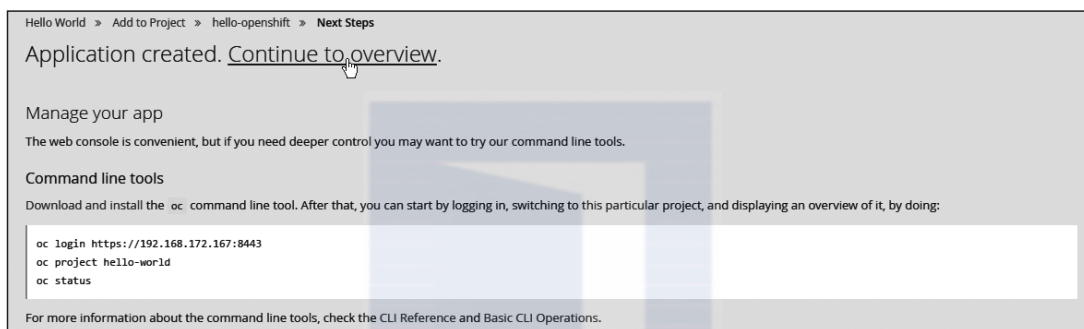


图 2-7 部署完成确认页面

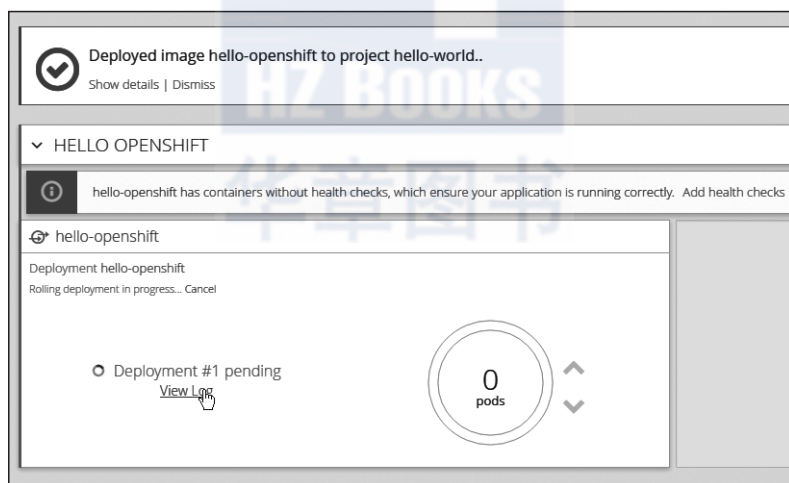


图 2-8 Hello World 项目主页

当第一次部署某个容器应用时，由于需要到 DockerHub 上下载镜像文件，所以需要等待一定的时间。所需时间视实验主机所在网络的网速而定。如果在创建容器应用的过程中出现了 Image Pull Error 的状态，可以尝试手工下载镜像。检查 Docker 能否正常连接上 DockerHub 及其镜像站点。

```
docker pull docker.io/openshift/hello-openshift
docker pull docker.io/openshift/origin-deployer:v1.3.0
docker pull docker.io/openshift/origin-pod:v1.3.0
```

稍等片刻后，hello-openshift 容器会成功启动。可以看到项目主页上的圆圈变成了蓝色，容器计数从“0”变成了“1”，如图 2-9 所示。这说明容器已经成功启动了，当前有“1”个在运行的实例。

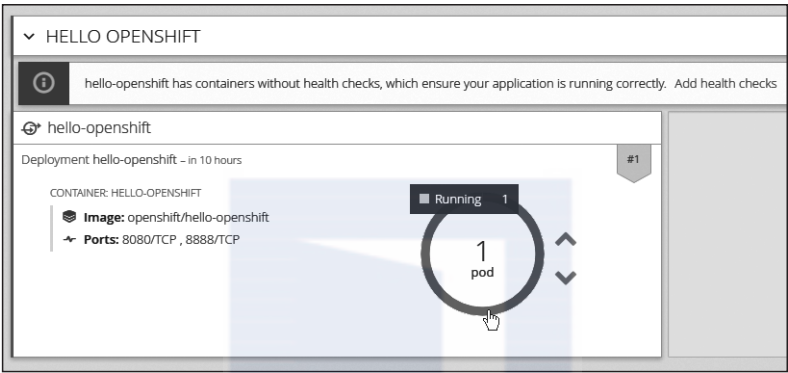


图 2-9 hello-openshift 容器成功启动

恭喜，您已经成功在 OpenShift 上运行了您的第一个容器应用！

2.2.3 访问容器应用

容器启动后，用户就可以尝试访问这个容器实例中运行的应用服务了。当容器启动后，每个容器实例都会被赋予一个内部的 IP 地址，用户可以通过这个地址访问容器。

单击界面上的圆圈将转跳到 hello-openshift 容器的实例列表，单击列表中的容器进入容器详情页面。在详情页面可以看到当前的容器被分配了一个 IP 地址，如图 2-10 中的 172.17.0.3。

回到实验的主机上，执行下面的命令就可以访问 hello-openshift 容器提供的服务。hello-openshift 中运行着一个简单的用 Go 语言编写的应用。其监听在 8080 端口，并为所有请求返回字符串 Hello OpenShift!。

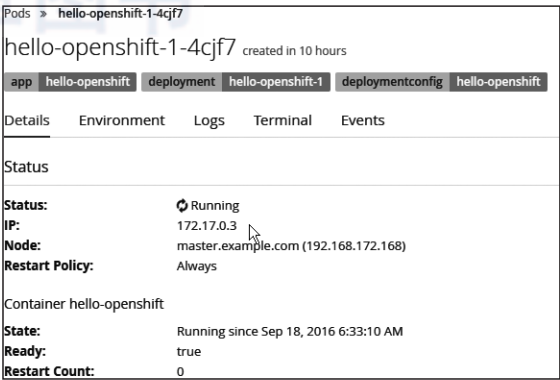


图 2-10 hello-openshift 容器的详情页面

```
[root@masteropenshift]# curl 172.17.0.3:8080
Hello OpenShift!
```



如上面的输出所示，容器应用成功返回了 `Hello OpenShift!`。这表明，我们的容器应用工作正常。

### 2.2.4 一些疑问

在实验的主机上，我们通过命令 `curl 172.17.0.3:8080` 成功访问了应用。但是如果在另一台主机上执行相同的命令，就会发现无法访问到这个服务。别着急，这是因为 `172.17.0.3` 是一个内部的 IP 地址，只存在于 OpenShift 集群当中。OpenShift 集群之外的机器将无法识别这个 IP 地址。那么，集群外的机器该如何访问我们的容器服务呢？这里先买个关子，在后面的章节里，我们将会慢慢揭晓答案。

此外，本例的安装只是针对一台主机，这样的环境适合作为开发环境使用。对于多主机集群的安装，请参考第 11 章。

至此，我们安装了一个单节点的 OpenShift 集群，并运行了一个名为 `hello-openshift` 的容器镜像。`hello-openshift` 是一个非常简单的容器应用。在 `hello-openshift` 的容器启动时会运行一个用 Go 语言编写的程序，这个程序将会持续监听在 `8080` 端口，响应任何输入请求并返回字符串 `"Hello OpenShift!"`。接下来，我们将部署一个更加复杂且有趣的容器应用，并一起探索 OpenShift 为部署的容器应用提供了哪些后台支持。

## 2.3 完善 OpenShift 集群

在部署更复杂的应用之前，有一项重要的任务需要完成，那就是完善 OpenShift 集群。在上一章中，通过二进制的安装包，我们快速完成了 OpenShift 集群的安装，但是这个集群还只是一个“空”的集群。面对复杂的应用，OpenShift 需要更多组件的支持。而这些组件并没有在上一章的安装中完成。同时，OpenShift 作为一个容器云平台，默认提供了一系列用户开箱即用、一键部署的应用和服务，这些应用和服务的信息也需要在系统中注册，以便用户在类似软件市场（App Store）的服务目录中选用。现在让我们一起完善这个 OpenShift 实例。

### 2.3.1 命令行工具

在上一章中，我们使用 OpenShift 的 Web 控制台部署了第一个容器应用。OpenShift 的 Web 控制台的用户体验非常好，通过图形界面，用户可以高效快速地完成操作。除了 Web 控制台外，OpenShift 还提供了一系列命令行工具。

`oc` 是 OpenShift 中一个重要的命令行客户端。OpenShift Web 控制台能完成的事情，通过 `oc` 命令也能完成。在进行自动化及重复性的操作时，命令行工具比图形界面更加高效。为了方便读者进行实验操作，本书后续的示例将以命令行进行操作。

可以尝试执行 `oc version` 命令查看 OpenShift 的集群版本信息，测试 `oc` 命令是否正常工作。

```
[root@masteropenshift]# oc version
oc v1.3.0
kubernetes v1.3.0+52492b4
features: Basic-Auth GSSAPI Kerberos SPNEGO
```

可以看到命令输出了 OpenShift 及其使用的 Kubernetes 的版本信息。

因为 oc 命令是带有权限管控的，所以在使用 oc 命令进行实际的操作前，需要先通过 oc login 命令登录。如下例所示，通过 oc login 命令，以 dev 用户的身份登录。

```
[root@master ~]# oc login -u dev https://192.168.172.167:8443
The server uses a certificate signed by an unknown authority.
You can bypass the certificate check, but any data you send to the server could be intercepted
by others.
Use insecure connections? (y/n): y

Authentication required for https://192.168.172.167:8443 (openshift)
Username: dev
Password:
Login successful.

You have access to the following projects and can switch between them with 'oc project
<projectname>':

    * hello-world

Using project "hello-world".
Welcome! See 'oc help' to get started.
[root@master ~]#
```

通过 oc new-project 命令创建一个新项目 hello-world-oc。

```
[root@master ~]# oc new-project hello-world-oc
Now using project "hello-world-oc" on server "https://192.168.172.167:8443".

You can add applications to this project with the 'new-app' command. For example, try:

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

to build a new example application in Ruby.
```

前文我们通过 Web 控制台部署了 hello-openshift 镜像。在命令行可以通过 oc new-app 命令方便地部署 DockerHub 等 Docker 镜像仓库的镜像。

```
[root@master ~]# oc new-app openshift/hello-openshift
warning: Cannot find git. Ensure that it is installed and in your path. Git is required
to work with git repositories.
--> Found Docker image 17b78a4 (28 hours old) from Docker Hub for "openshift/hello-openshift"

* An image stream will be created as "hello-openshift:latest" that will track this image
* This image will be deployed in deployment config "hello-openshift"
```

```

* Ports 8080/tcp, 8888/tcp will be load balanced by service "hello-openshift"
* Other containers can access this service through the hostname "hello-openshift"
* WARNING: Image "openshift/hello-openshift" runs as the 'root' user which may not
be permitted by your cluster administrator

--> Creating resources with label app=hello-openshift ...
imagestream "hello-openshift" created
deploymentconfig "hello-openshift" created
service "hello-openshift" created
--> Success
Run 'oc status' to view your app.

```

执行 `oc get pod` 命令可以查看当前项目的容器的列表。和在 Kubernetes 一样，在 OpenShift 中，所有的 Docker 容器都是被“包裹”在一种称为 Pod 的容器内部。用户可以近似地认为 Pod 就是我们要运行的 Docker 容器本身。

```

[root@master ~]# oc get pod
NAME                READY    STATUS    RESTARTS   AGE
hello-openshift-1-8gv1i  1/1      Running   0           19s

```

执行 `oc describe pod` 命令可以查看 Pod 的详细配置和状态信息。下例为 Pod `hello-openshift-1-8gv1i` 的详细信息，包含容器的名称、状态、所处的命名空间（项目）、标签、IP 地址等。

```

[root@master ~]# oc describe pod hello-openshift-1-8gv1i
Name:                hello-openshift-1-8gv1i
Namespace:           hello-world-oc
Security Policy:      restricted
Node:                master.example.com/192.168.172.167
Start Time:          Sat, 17 Sep 2016 21:33:06 -0400
Labels:              app=hello-openshift
                    deployment=hello-openshift-1
                    deploymentconfig=hello-openshift
Status:              Running
IP:                  172.17.0.7
Controllers:         ReplicationController/hello-openshift-1
Containers:
  hello-openshift:
    Container ID:     docker://15263bbec6053b215b5dade08c839d5b07530b4cdf3b87776
    .....

```

在后续的介绍中，我们会使用 `oc` 命令进行大量的操作，相信你很快就会熟悉它的使用方法。

### 2.3.2 以集群管理员登录

在安装组件之前，我们需要以集群管理员的角色登录。在 OpenShift 中，默认的集群管理员是 `system:admin`。`system:admin` 这个用户拥有最高的权限。有意思的是，和其他

用户不同，system:admin 用户并没有密码！system:admin 的登录依赖于证书密钥。以下是登录的方法。

1) 拷贝登录配置文件。如果提示文件已存在，请选择覆盖。

```
[root@master ~]# mkdir -p ~/.kube
[root@master ~]# cp /opt/openshift/openshift.local.config/master/admin.kubeconfig
~/.kube/config
[root@master ~]# cp: overwrite '/root/.kube/config'? y
```

2) 通过 oc login 命令登录。

```
[root@masteropenshift]# oc login -u system:admin
Logged into "https://192.168.172.167:8443" as "system:admin" using existing credentials.
```

```
You have access to the following projects and can switch between them with 'oc project
<projectname>':
```

```
    * default
hello-world
hello-world-oc
kube-system
openshift
openshift-infra
```

```
Using project "default".
```

3) 执行 ocwhoami 命令，即可见当前登录用户为 system:admin。

```
[root@master ~]# ocwhoami
system:admin
```

可以尝试执行 oc get node 命令查看集群节点信息。只有集群管理员才有权限查看集群的节点信息。

```
[root@master ~]# oc get node
NAME                STATUS    AGE
master.example.com  Ready    2h
```

可以看到我们的机器中有且只有一个节点 master.example.com，它的状态是就绪 (Ready) 的。在实际的生产环境中，集群中将会有许多节点，这会是一个庞大的列表。

### 2.3.3 添加 Router

首先，为集群添加一个 Router 组件。Router 是 OpenShift 集群中一个重要的组件，它是外界访问集群容器应用的入口。集群外部的请求都会到达 Router，并由 Router 分发到具体的容器中。关于 Router 的详细信息我们会在后续的章节详细探讨。

切换到 default 项目。

```
[root@master ~]# oc project default
```

Router 组件需要读取集群的信息，因此它关联一个系统账号 Service Account，并为这个账号赋权。Service Account 是 OpenShift 中专门供程序和组件使用的账号。OpenShift 中有严格的权限和安全保障机制。不同的用户会关联到不同的安全上下文（Security Context Constraint, SCC）。同时，用户或组也会关联到不同的系统角色（Role）。

```
[root@master ~]# oadm policy add-scc-to-user privileged system:serviceaccount:default:router
```

执行 `oadm router` 命令创建 Router 实例。

```
[root@master ~]# oadm router router --replicas=1 --service-account=router
info: password for stats user admin has been set to EhEVZXbjAn
--> Creating router router ...
serviceaccount "router" created
clusterrolebinding "router-router-role" created
deploymentconfig "router" created
service "router" created
--> Success
```

`oadm` 命令是 `oc` 命令的好搭档。`oc` 命令更多地是面向一般用户，而 `oadm` 命令是面向集群管理员，可以进行集群的管理和配置。在上面的命令中，我们指定创建一个名为 `router` 的 Router。

参数 `--replicas=1` 表明，我们只想创建一个实例。在实际的生产中，为了达到高可用的效果，可以创建多个 Router 实例实现负载均衡并防止单点失效。

执行片刻之后，通过 `oc get pod -n default` 命令可以查看 Router 容器的状态。

```
[root@master ~]# oc get pod -n default
NAME          READY   STATUS    RESTARTS   AGE
router-1-e95qa 1/1     Running   0           3m
```

上面的输出显示 Router 容器的状态是 `Running`。如果此时检查实验主机上的端口监听状态，可以发现主机的端口 80、443 正在被 Haproxy 监听。

```
[root@master ~]# ss -ltn|egrep -w "80|443"
LISTEN        0        128      *:80        *:80
LISTEN        0        128      *:443       *:443
```

其实，从技术上来说，Router 就是一个运行在容器中的 Haproxy，当然这个 Haproxy 经过了特别的配置来实现特殊的功能。这些我们在后面再详细讨论。

至此，Router 组件部署就已经完成了。

## 2.3.4 添加 Registry

接下来部署集群内部的 Docker Registry，即内部的 Docker 镜像仓库。从功能上说，OpenShift 内部的镜像仓库和外部的企业镜像仓库或者 DockerHub 没有本质的区别。只是这个内部的镜像仓库会用来存放一些“特殊的”镜像，这些镜像是由一个叫 Source to Image (S2I) 的



流程产生的。简单地说，S2I 的工作是辅助将应用的源代码转换成可以部署的 Docker 镜像。关于 S2I，后续再详细介绍。

1) 切换到 default 项目。

```
[root@master ~]# oc project default
```

2) 执行如下命令部署 Registry。

```
[root@master ~]# oadm registry --config=/opt/openshift/openshift.local.config/master/
admin.kubeconfig --service-account=registry
--> Creating registry registry ...
serviceaccount "registry" created
clusterrolebinding "registry-registry-role" created
deploymentconfig "docker-registry" created
service "docker-registry" created
--> Success
```

3) 稍候片刻，执行 `oc get pod` 便可见 Registry 容器处于运行状态了。

```
[root@master ~]# oc get pod
```

NAME	READY	STATUS	RESTARTS	AGE
docker-registry-1-xm3un	1/1	Running	0	1m
router-1-e95qa	1/1	Running	0	9m

在本例中，因为我们部署的 Registry 没有启用 HTTPS，所以需要修改 Docker 的配置让 Docker 以非 HTTPS 的方式连接到 Registry。修改 `/etc/sysconfig/docker` 文件，为 `OPTIONS` 变量值追加 `--insecure-registry=https://172.30.0.0/16`。修改后的变量值如下：

```
OPTIONS='--selinux-enabled --log-driver=journald --registry-mirror=https://docker.
mirrors.ustc.edu.cn --insecure-registry=172.30.0.0/16'
```

4) 重启 Docker 服务，使修改的配置生效。

```
[root@master opt]# systemctl restart docker
```

至此，Registry 组件部署完成。

## 2.3.5 添加 Image Stream

Image Stream 是一组镜像的集合。可以在一个 Image Stream 中定义一些名称及标签 (tag)，并定义这些名字及标签指向的具体镜像。值得指出的是，在 OpenShift 上部署容器应用，并不一定要用到 Image Stream，直接指定镜像的地址也可以完成部署。使用 Image Stream 为的是方便地将一组相关联的镜像进行整合管理和使用。OpenShift Origin 默认为用户定义了一系列开箱即用的 Image Stream。

1) 切换到 openshift 项目。

```
[root@master ~]# oc project openshift
Now using project "openshift" on server "https://192.168.172.167:8443".
```

## 2) 通过以下命令可以导入 Image Stream。

```
[root@master ~]# curl https://raw.githubusercontent.com/openshift/origin/v1.3.0/examples/
image-streams/image-streams-centos7.json|oc create -f - -n openshift
      % Total      % Received % Xferd  Average Speed   Time    Time     Current
Dload  Upload  Total    Spent    Left  Speed
100 18953 100 18953    0     0  9354      0  0:00:02  0:00:02 --:--:--  9359
imagestream "ruby" created
imagestream "nodejs" created
imagestream "perl" created
imagestream "php" created
imagestream "python" created
imagestream "wildfly" created
imagestream "mysql" created
imagestream "mariadb" created
imagestream "postgresql" created
imagestream "mongodb" created
imagestream "jenkins" created
```

## 3) 通过 `oc get is -n openshift` 命令，可以列出刚才导入的 Image Stream 对象。

```
[root@master ~]# oc get is -n openshift
```

NAME	DOCKER REPO	TAGS	UPDATED
jenkins	172.30.73.49:5000/openshift/jenkins	latest,1	44 seconds ago
mariadb	172.30.73.49:5000/openshift/mariadb	latest,10.1	About a minute ago
mongodb	172.30.73.49:5000/openshift/mongodb	latest,3.2,2.6 + 1 more...	53 seconds ago
mysql	172.30.73.49:5000/openshift/mysql	latest,5.6,5.5	About a minute ago
nodejs	172.30.73.49:5000/openshift/nodejs	0.10,latest,4	2 minutes ago
perl	172.30.73.49:5000/openshift/perl	latest,5.20,5.16	2 minutes ago
php	172.30.73.49:5000/openshift/php	5.5,latest,5.6	2 minutes ago
postgresql	172.30.73.49:5000/openshift/postgresql	9.4,9.2,latest + 1 more...	About a minute ago
python	172.30.73.49:5000/openshift/python	latest,3.5,3.4 + 2 more...	About a minute ago
ruby	172.30.73.49:5000/openshift/ruby	latest,2.3,2.2 + 1 more...	2 minutes ago
wildfly	172.30.73.49:5000/openshift/wildfly	10.0,9.0,8.1 + 1 more...	About a minute ago

此时，如果访问 OpenShift 的 Web 控制台，进入 Hello World 项目，单击项目 Overview 页面顶部的 `Add to project` 按钮，则会看见一系列可用的镜像被罗列在页面上，如图 2-11 所示。

### 2.3.6 添加 Template

部署容器应用，可以很简单：直接通过 `docker run` 或 `oc new-app` 命令即可完成。但是有时候它也可以是一项很复杂的任务。在现实中，企业的应用往往不是孤立存在的，应用往往有多个模块；部署需要满足外部的依赖；用户需要根据实际的需求，结合环境的配置给部署传递不同的参数。为了满足用户对复杂应用部署的需求，提高应用部署的效率，

OpenShift 引入了应用部署模板（Template）的概念。通过 Template，用户可以定义一个或多个需要部署的镜像，定义部署依赖的对象，定义可供用户输入配置的参数项。OpenShift 默认提供了一些示例的 Template 供用户使用。后续用户可以根据实际的需求，定义满足企业需求的应用部署模板，构建企业内部的软件市场。

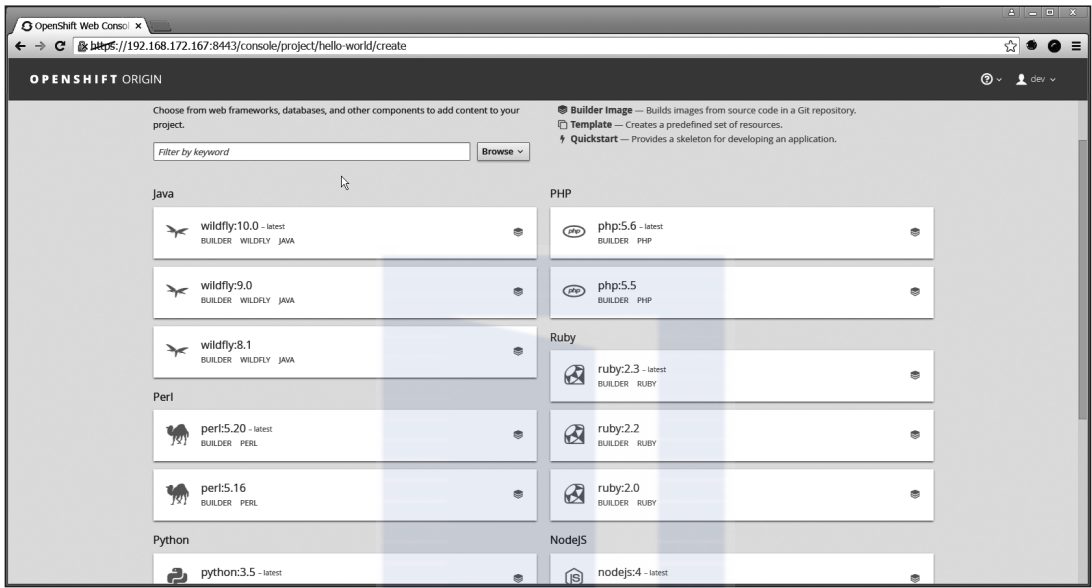


图 2-11 导入 Image Stream 后 Web 界面展示可用的镜像列表

- 1) 切换到 openshift 项目。
- ```
[root@master ~]# oc project openshift
Now using project "openshift" on server "https://192.168.172.167:8443".
```
- 2) 下载并创建一个 CakePHP 示例应用的 Template。通过这个 Template，用户可以在服务目录单击相关的条目一键部署一个 CakePHP 应用和一个 MySQL 数据库。
- ```
[root@master ~]# oc create -f https://raw.githubusercontent.com/openshift/origin/v1.3.0/examples/quickstarts/cakephp-mysql.json -n openshift
template "cakephp-mysql-example" created
```
- 3) 创建完毕后，可以通过 `oc get template -n openshift` 命令查看导入的模板信息。
- ```
[root@master ~]# oc get template -n openshift
```
- | NAME                  | DESCRIPTION                                          | PARAMETERS   | OBJECTS |
|-----------------------|------------------------------------------------------|--------------|---------|
| cakephp-mysql-example | An example CakePHP application with a MySQL database | 19 (4 blank) | 7       |
- 如果要查看模板的详细内容，可以通过 `oc get template cakephp-mysql-example -o json -n openshift` 命令查看。`-o` 参数指定了命令以 json 格式输出返回值。

```
oc get template cakephp-mysql-example -o json -n openshift
```

刷新 OpenShift Web 控制台的服务目录界面，在过滤器中输入 cake，即可看到刚导入的应用模板，如图 2-12 所示。

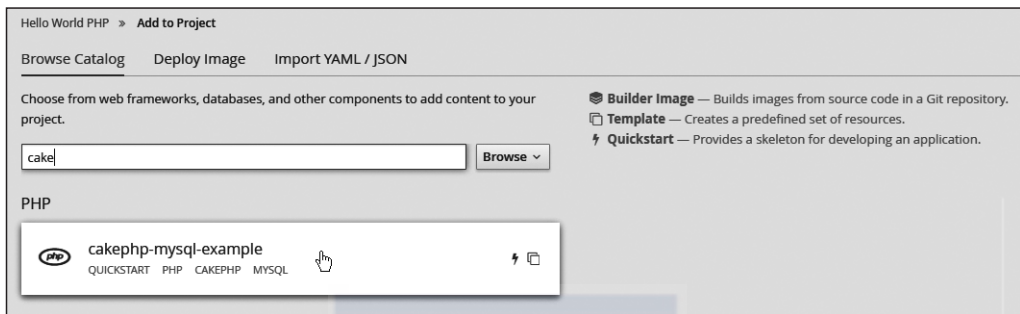


图 2-12 导入的 CakePHP 应用模板显示在 Web 控制台

在 OpenShift Origin 的 GitHub 仓库中还有许多预定义好的 Template 示例。你可以按需下载，并通过 `oc create -f` 命令导入系统中。



**提示** OpenShift Origin 示例：<https://github.com/openshift/origin/tree/v1.3.0/examples>。

请执行下面的命令导入 `wildfly-basic-s2i` 模板，这在后面的章节会使用到。

```
oc create -f https://raw.githubusercontent.com/nichochen/openshift-book-source/master/template/wildfly-basic-s2i.template.json -n openshift
```



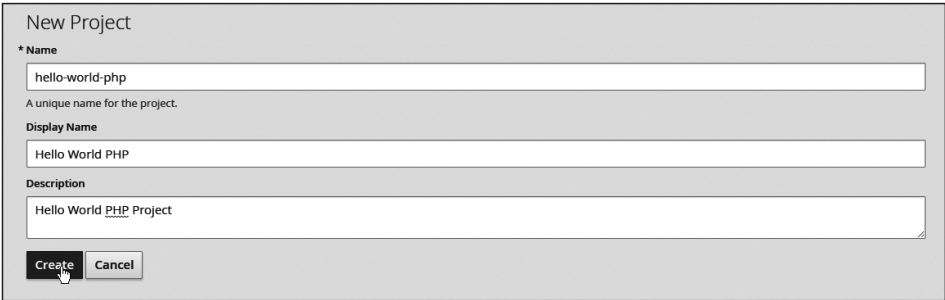
**提示** 细心的读者也许会发现之前创建 **Router** 和 **Registry** 是在 `default` 项目中，而创建 **Image Stream** 是在 `openshift` 项目中。`openshift` 项目是一个特殊的项目，在这个项目下创建的所有 **Image Stream** 及 **Template** 对集群内所有的用户和项目可见。如果 **Image Stream** 及 **Template** 在其他项目创建，则只能在创建这些对象的项目内可见。

## 2.4 部署应用

在前几节中，我们完成了众多关键组件的部署。现在是时候尝试部署一些应用了。本节我们将部署一个 CakePHP 应用及 MySQL 数据库。

1) 登录 OpenShift Web 控制台。单击 **New project** 按钮。创建一个名为 `hello-world-php` 的项目。输入项目名称 `hello-world-php` 及项目显示名 `Hello World PHP`。单击 **Create** 按钮创建项目，如图 2-13 所示。

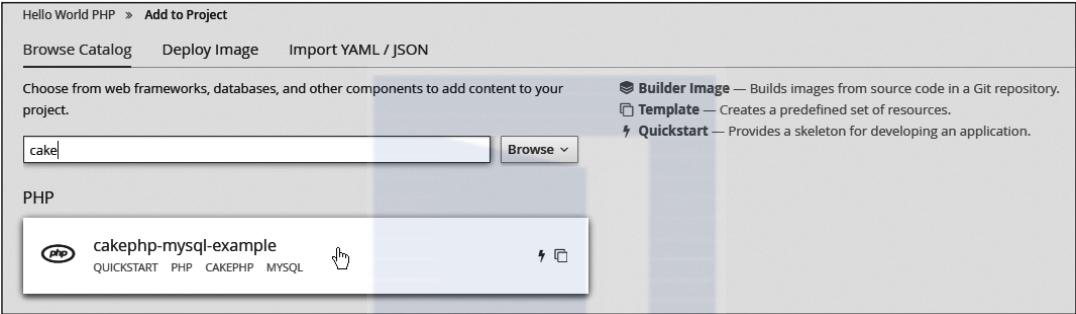
2) 在服务目录的过滤器中输入 `cake`，找到 `cakephp-mysql-example` 模板，如图 2-14 所示。



The 'New Project' form contains the following fields and controls:

- \* Name:** A text input field containing 'hello-world-php'. Below it is a note: 'A unique name for the project.'
- Display Name:** A text input field containing 'Hello World PHP'.
- Description:** A text input field containing 'Hello World PHP Project'.
- Buttons:** 'Create' and 'Cancel' buttons at the bottom left.

图 2-13 创建 helloworld-ng 项目

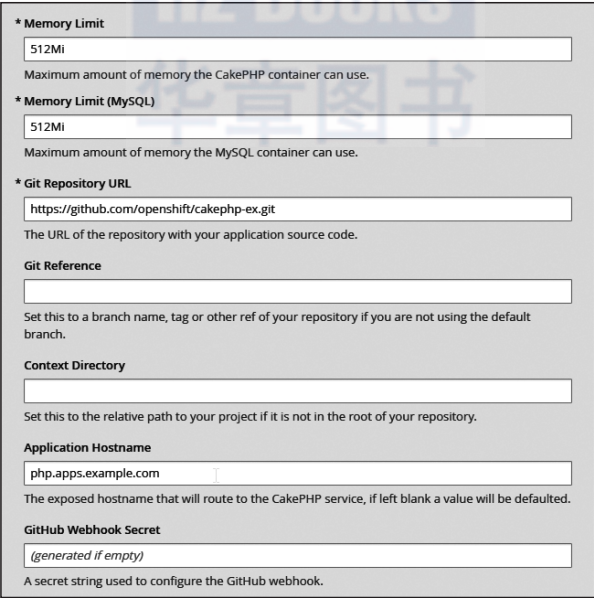


The 'Add to Project' dialog shows the following components:

- Navigation:** 'Browse Catalog' (active), 'Deploy Image', and 'Import YAML / JSON' tabs.
- Search:** A search bar with 'cake' entered and a 'Browse' button.
- Results:** A list of items under the 'PHP' category. The first item is 'cakephp-mysql-example', which includes sub-items 'QUICKSTART', 'PHP', 'CAKEPHP', and 'MYSQL'. It has a PHP icon, a hand cursor, and a lightning bolt icon.
- Help:** On the right, there are links for 'Builder Image' (Builds images from source code in a Git repository), 'Template' (Creates a predefined set of resources), and 'Quickstart' (Provides a skeleton for developing an application).

图 2-14 选取 cakephp-mysql-example 模板

3) 选取 Template 后将跳转至 Template 的参数输入页面。在参数输入页面为 Application Hostname 属性赋值 php.apps.exmaple.com，如图 2-15 所示。



The parameter input form includes the following sections:

- \* Memory Limit:** A text input field with '512Mi'. Below it: 'Maximum amount of memory the CakePHP container can use.'
- \* Memory Limit (MySQL):** A text input field with '512Mi'. Below it: 'Maximum amount of memory the MySQL container can use.'
- \* Git Repository URL:** A text input field with 'https://github.com/openshift/cakephp-ex.git'. Below it: 'The URL of the repository with your application source code.'
- Git Reference:** An empty text input field. Below it: 'Set this to a branch name, tag or other ref of your repository if you are not using the default branch.'
- Context Directory:** An empty text input field. Below it: 'Set this to the relative path to your project if it is not in the root of your repository.'
- Application Hostname:** A text input field with 'php.apps.example.com'. Below it: 'The exposed hostname that will route to the CakePHP service, if left blank a value will be defaulted.'
- GitHub Webhook Secret:** A text input field with '(generated if empty)'. Below it: 'A secret string used to configure the GitHub webhook.'

图 2-15 模板参数输入界面



4) 单击模板参数输入页面底部的 **Create** 按钮, 执行部署, 如图 2-16 所示。

Labels

The following labels are being added automatically. If you want to override them, you can do so below.

|          |                       |
|----------|-----------------------|
| template | cakephp-mysql-example |
| app      | cakephp-mysql-example |

Each label is applied to each created resource.

| Name | Value |
|------|-------|
|      |       |

Add label

**Create** **Cancel**

图 2-16 执行实例化 Template

5) 执行部署后, 浏览器将跳转至部署完成页面, 如图 2-17 所示。

Hello World PHP » Add to Project » cakephp-mysql-example » **Next Steps**

Application created. [Continue to overview.](#)

**Manage your app**

The web console is convenient, but if you need deeper control you may want to try our command line tools.

**Command line tools**

Download and install the `oc` command line tool. After that, you can start by logging in, switching to this particular project, and displaying an overview of it, by doing:

```
oc login https://192.168.172.167:8443
oc project hello-world-php
oc status
```

For more information about the command line tools, check the [CLI Reference](#) and [Basic CLI Operations](#).

**Making code changes**

A GitHub webhook trigger has been created for the **cakephp-mysql-example** build config.

You can now set up the webhook in the GitHub repository settings if you own it, in <https://github.com/openshift/cakephp-ex/settings/hooks>, using the following payload URL:

`https://192.168.172.167:8443/oapi/v1/nam`

图 2-17 部署完成页面

6) 单击确认页面的 **Continue overview** 链接, 跳转到项目的概览页面。此时 OpenShift 会在后台创建相应的对象, 并下载相关的容器镜像。MySQL 容器一般会较快完成启动, 因为 CakePHP 应用涉及一个镜像构建的过程, 即 **Source to Image**。关于这个镜像构建的细节, 本书后续再表。单击界面上的 **view log** 链接可以查看相关的日志, 如图 2-18 所示。

**Created cakephp-mysql-example in project Hello World PHP.**

[Show details](#) | [Dismiss](#)

▼ CAKEPHP MYSQL EXAMPLE

Build cakephp-mysql-example, #1 **Running**. A new deployment will be created automatically once the build completes. in 10 hours [View Log](#)

cakephp-mysql-example

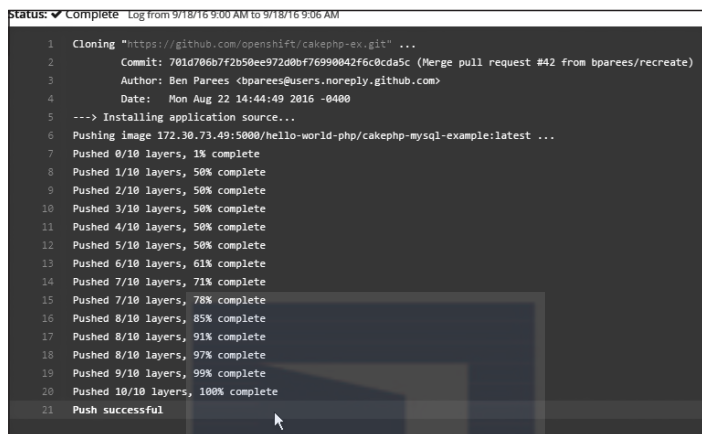
No deployments.

There are no deployments or pods for service cakephp-mysql-example.

图 2-18 hello-world-php 项目主页

7) 稍等片刻后, 在 CakePHP 的构建日志界面, 可以看到镜像构建的实时日志输出, 如

图 2-19 所示。从日志中可以看到，OpenShift 会从 GitHub 仓库中下载指定的 PHP 源代码，然后将代码注入一个含 PHP 运行环境的镜像，最终生成一个包含 PHP 应用及 PHP 运行环境的新镜像，并将新的镜像推送到前文部署的内部镜像仓库。



```

status: Complete Log from 9/18/16 9:00 AM to 9/18/16 9:06 AM
1 Cloning "https://github.com/openshift/cakephp-ex.git" ...
2 Commit: 701d706b7f2b50ee972d0bf76990042f6c0cda5c (Merge pull request #42 from bparees/recreate)
3 Author: Ben Parees <bparees@users.noreply.github.com>
4 Date: Mon Aug 22 14:44:49 2016 -0400
5 ---> Installing application source...
6 Pushing image 172.30.73.49:5000/hello-world-php/cakephp-mysql-example:latest ...
7 Pushed 0/10 layers, 1% complete
8 Pushed 1/10 layers, 50% complete
9 Pushed 2/10 layers, 50% complete
10 Pushed 3/10 layers, 50% complete
11 Pushed 4/10 layers, 50% complete
12 Pushed 5/10 layers, 50% complete
13 Pushed 6/10 layers, 61% complete
14 Pushed 7/10 layers, 71% complete
15 Pushed 7/10 layers, 78% complete
16 Pushed 8/10 layers, 85% complete
17 Pushed 8/10 layers, 91% complete
18 Pushed 8/10 layers, 97% complete
19 Pushed 9/10 layers, 99% complete
20 Pushed 10/10 layers, 100% complete
21 Push successful
  
```

图 2-19 CakePHP 应用的 S2I 构建日志

```

Cloning "https://github.com/openshift/cakephp-ex.git" ...
Commit: 701d706b7f2b50ee972d0bf76990042f6c0cda5c (Merge pull request #42 from bparees/recreate)
Author: Ben Parees<bparees@users.noreply.github.com>
Date: Mon Aug 22 14:44:49 2016 -0400
---> Installing application source...
Pushing image 172.30.73.49:5000/hello-world-php/cakephp-mysql-example:latest ...
Pushed 0/10 layers, 1% complete
Pushed 1/10 layers, 50% complete
Pushed 2/10 layers, 50% complete
Pushed 3/10 layers, 50% complete
Pushed 4/10 layers, 50% complete
Pushed 5/10 layers, 50% complete
Pushed 6/10 layers, 61% complete
Pushed 7/10 layers, 71% complete
Pushed 7/10 layers, 78% complete
Pushed 8/10 layers, 85% complete
Pushed 8/10 layers, 91% complete
Pushed 8/10 layers, 97% complete
Pushed 9/10 layers, 99% complete
Pushed 10/10 layers, 100% complete
Push successful
  
```



如果构建过程中出现了 `docker push` 镜像到内部镜像仓库相关的错误，请检查内部镜像仓库是否正确部署与配置。第一次推送镜像的时间会比较长，因为此时镜像仓库中还没有相应的镜像层（Layer）。后续构建的镜像推送时间将会大大加快，因为大量可以重用的镜像层已经存在于内部的镜像仓库中了。

8) 构建完成后, 单击左侧菜单栏的 **Overview** 按钮, 回到项目主页, 如图 2-20 所示。此时可见 CakePHP 应用已经启动完毕。



图 2-20 项目主页

OpenShift 将我们指定的域名 `php.apps.example.com` 与 CakePHP 容器应用进行了关联。单击 CakePHP 应用右上角的 `php.apps.example.com` 域名链接即可打开容器应用, 如图 2-21 所示。

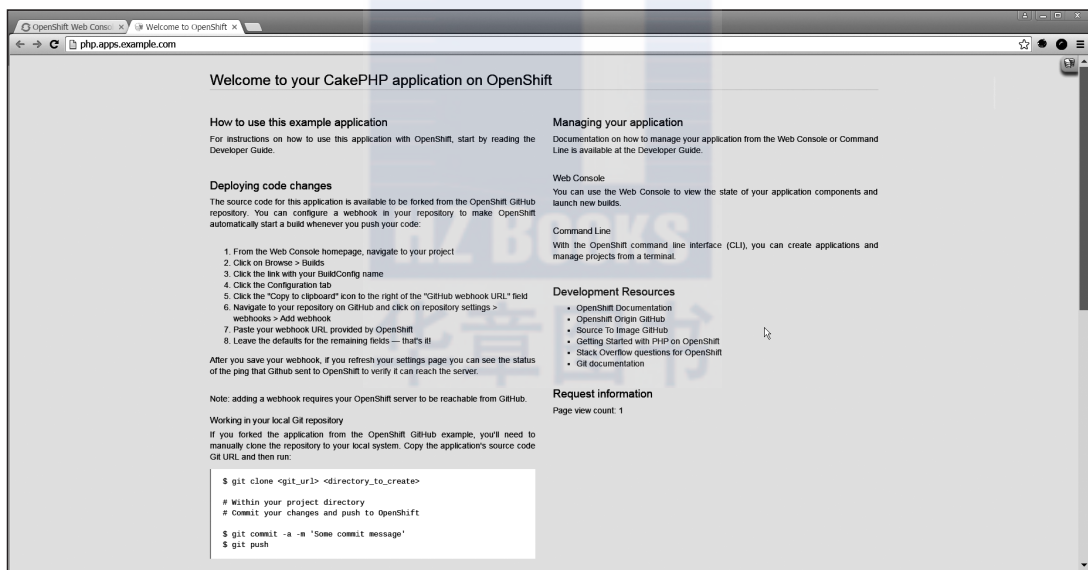


图 2-21 CakePHP 应用页面



**注意**

`php.apps.example.com` 域名只是我们测试用的域名, 并不能被互联网的域名解析服务器解析。需要修改浏览器所在机器的 `hosts` 文件, 手工添加解析将 `php.apps.example.com` 指向实验机器的 IP 地址。

❑ Windows 系统, 请修改 `c:\windows\system32\drivers\etc\hosts` 文件。

❑ Linux 系统, 请修改 `/etc/hosts` 文件。

在这个应用部署的例子中，我们通过选择一个预定义的应用部署模板，快速部署了一个 CakePHP 应用及一个 MySQL 数据库。整个部署的过程，不外乎几次鼠标单击。在实际的使用中，企业可以在服务目录中加入各种不同的应用服务模板，构建出企业内部软件市场式的服务目录。用户或管理员可以通过服务目录选取需要部署的软件应用模板，输入相应的参数，然后执行部署，相关的应用服务便会以容器的方式运行在指定的服务器集群上。这些应用服务可以是一个单体的应用，也可以包含多个不同的组件，如前文部署的示例，包含了一个前端 PHP 应用及一个后端 MySQL 数据库。通过软件市场式的服务目录，即使对 OpenShift 没有太多了解的用户，也能快速部署复杂的应用。作为一个容器云平台，OpenShift 极大地提升了应用部署的效率，使得应用部署实现自动化及标准化。

应用部署出错了？别担心，可以通过项目左边的侧栏菜单打开 Monitor 界面查看项目后台的事件，排查相应的错误，如图 2-22 所示。

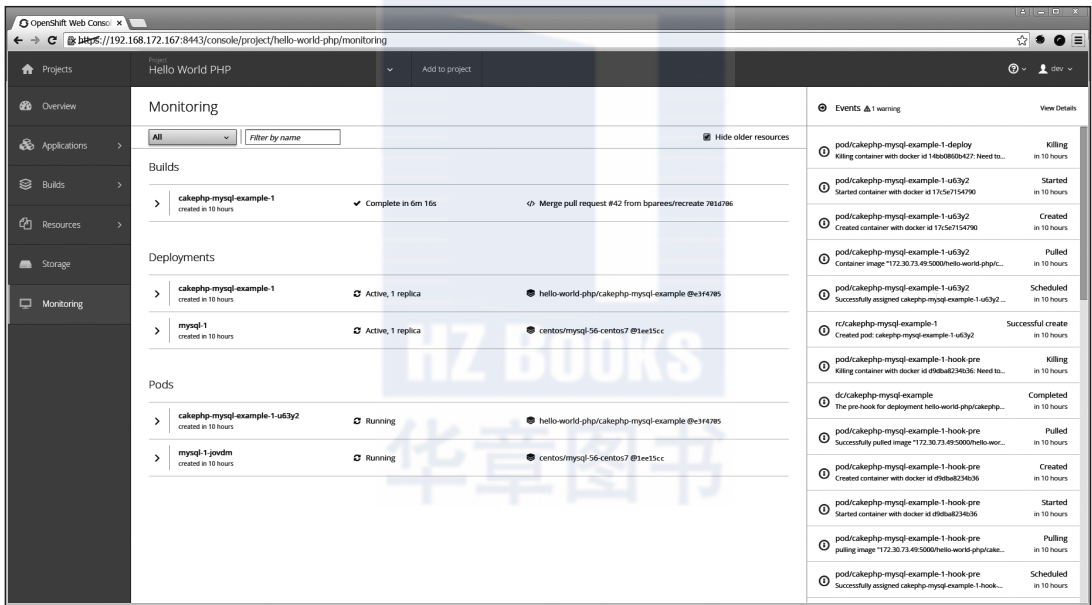
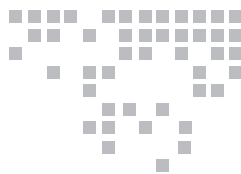


图 2-22 项目事件监控界面

## 2.5 本章小结

通过本章，我们成功安装及运行了 OpenShift 集群，并完成了对 OpenShift 集群的完善和升级，增加了 Router 及 Registry 组件；丰富了 OpenShift 平台上可供用户选择的应用和服务；通过服务目录，快速部署了一个带有前端 PHP 应用及后端数据库的应用。通过本章的探索，相信你对 OpenShift 的了解更上了一个层次。



## OpenShift 架构探秘

在上一章中，我们通过模板部署了一个前端 PHP 应用及一个后端 MySQL 数据库。对用户而言，部署的过程十分简单，通过几次鼠标单击即可完成应用的部署。但在用户便利的幕后，其实 OpenShift 平台为用户完成了大量操作。在这一章，我们将会深入了解应用部署背后的故事，深入了解 OpenShift 容器云的架构。

### 3.1 架构概览

从技术堆栈的角度分析，作为一个容器云，OpenShift 自底而上包含了以下几个层次：基础架构层、容器引擎层、容器编排层、PaaS 服务层、界面及工具层，如图 3-1 所示。

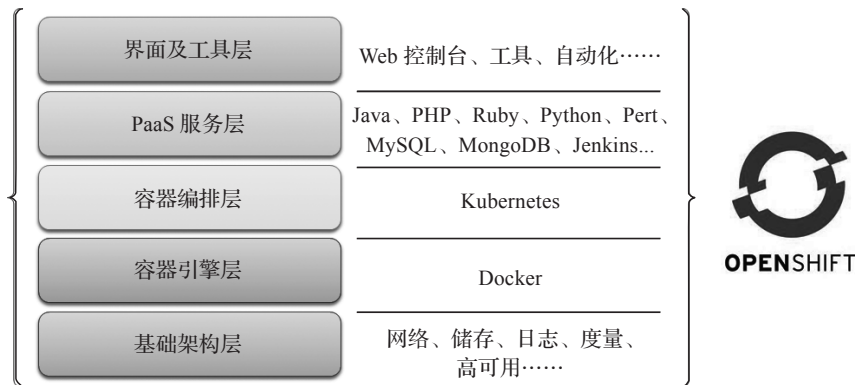


图 3-1 OpenShift 技术堆栈

### 3.1.1 基础架构层

基础架构层为 OpenShift 平台的运行提供了基础的运行环境。OpenShift 支持运行在物理机、虚拟机、基础架构云（如 OpenStack、Amazon Web Service、Microsoft Azure 等）或混合云上。在操作系统层面，OpenShift 支持多种不同的 Linux 操作系统，如企业级的 Red Hat Enterprise Linux、社区的 CentOS。值得一提的是，2015 年 Red Hat 针对容器平台启动了 Atomic Project，并推出了专门针对容器化运行环境的操作系统 Atomic Host。从技术上来看，Atomic Host 也是一个 Linux 操作系统，是基于 Red Hat 的企业版 Linux 的基础上优化和定制而来。通过根分区只读、双根分区、RPM OSTree 等特性，Atomic Host 可以为容器应用的运行提供一个高度一致的环境，保证在大规模容器集群环境中容器应用的稳定与安全。

在谈到容器时，大家经常会提及容器的一个优点，那就是可以保证应用的一致性。同样的容器镜像，在开发、测试和生产环境中运行的结果应该是一致的。但是容器的一致性和可移植性是有前提条件的，那就是底层操作系统的内核及相关的配置要一致。容器为应用提供了一个隔离的运行环境，这个隔离的实现依赖于底层 Linux 内核的系统调用。如果大量服务器的 Linux 内核及操作系统的配置不能保证一致，那么容器运行的最终结果的一致性也不可能有保障。



提示 要了解更多关于 Atomic 容器操作系统的信息可以访问 Atomic Project 项目主页：<http://www.atomic-project.org>。

### 3.1.2 容器引擎层

OpenShift 目前以 Docker 作为平台的容器引擎。Docker 是当前主流的容器引擎，已经在社区及许多企业的环境中进行了检验。事实证明 Docker 有能力为应用提供安全、稳定及高性能的运行环境。OpenShift 运行的所有容器应用最终落到最底层的实现，其实就是一个 Docker 容器实例。OpenShift 对 Docker 整合是开放式的。OpenShift 并没有修改 Docker 的任何代码，完全基于原生的 Docker。熟悉 Docker 的用户对 OpenShift 能快速上手。同时，Docker 现有的庞大的镜像资源都可以无缝地接入 OpenShift 平台。

### 3.1.3 容器编排层

目前大家对容器编排的讨论已经成为容器相关话题中的一个热点。Kubernetes 是 Google 在内部多年容器使用经验基础上的一次总结。Kubernetes 设计的目的是满足在大规模集群环境下对容器的调度和部署的需求。Kubernetes 是 OpenShift 的重要组件，OpenShift 平台上的许多对象和概念都是衍生自 Kubernetes，如 Pod、Namespace、Replication Controller 等。与对 Docker 的集成一样，OpenShift 并没有尝试从代码上定制 Kubernetes，OpenShift 对 Kubernetes



的整合是叠加式的，在 OpenShift 集群上仍然可以通过 Kubernetes 的原生命令来操作 Kubernetes 的原生对象。

### 3.1.4 PaaS 服务层

Docker 和 Kubernetes 为 OpenShift 提供了一个良好的基础，但是只有容器引擎和容器编排工具并不能大幅度提高生产效率，形成真正的生产力。正如 Kubernetes 在其主页上自我介绍所描述的那样，Kubernetes 关注的核心是容器应用的编排和部署，它并不是一个完整的 PaaS 解决方案。容器平台最终的目的是向上层应用服务提供支持，加速应用开发、部署和运维的速度和效率。OpenShift 在 PaaS 服务层默认提供了丰富的开发语言、开发框架、数据库及中间件的支持。用户可以在 OpenShift 这个平台上快速部署和获取一个数据库、分布式缓存或者业务规则引擎的服务。除了 Docker Hub 上的社区镜像外，OpenShift 还有一个重要的服务提供方：Red Hat。Red Hat 旗下的 JBoss 中间件系列几乎全线的产品都已经容器化。JBoss 中间件包含了开发框架、开发工具、应用服务器、消息中间件、SOA 套件、业务流程平台（BPM）、单点登录、应用监控、应用性能管理（APM）、分布式缓存及数据虚拟化等产品。这些中间件可以直接通过 OpenShift 容器云对用户提供服务。通过 OpenShift，可以快速搭建一个 Database as a Service，即 DBaaS，一个 BPMaaS，或者 Redis-aaS 等。

### 3.1.5 界面及工具层

云平台一个很重要的特点是强调用户的自助服务，从而降低运维成本，提高服务效率。界面和工具是容器云平台上的最后一公里接入，好的界面和工具集合能帮助用户更高效地完成相关的任务。OpenShift 提供了自动化流程 Source to Image，即 S2I，帮助用户容器化用各种编程语言开发的应用源代码。用户可以直接使用 S2I 或者把现有的流程与 S2I 整合，从而实现开发流程的持续集成和持续交付。提升开发、测试和部署的自动化程度，最终提高开发、测试及部署的效率，缩短上市时间。OpenShift 提供了多种用户的接入渠道：Web 控制台、命令行、IDE 集成及 RESTful 编程接口。这些都是一个完善的企业级平台必不可少的组件。

针对容器应用的运维及集群的运维，OpenShift 提供了性能度量采集、日志聚合模块及运维管理套件，帮助运维用户完成日常的应用及集群运维任务。

## 3.2 核心组件详解

OpenShift 的核心组件及其之间的关联关系如图 3-2 所示。OpenShift 在容器编排层使用了 Kubernetes，所以 OpenShift 在架构上和 Kubernetes 十分接近。其内部的许多组件和概念是从 Kubernetes 衍生而来，但是也存在一些在容器编排层之上，OpenShift 特有的组件和概念。下面将详细介绍 OpenShift 内部的核心组件和概念。

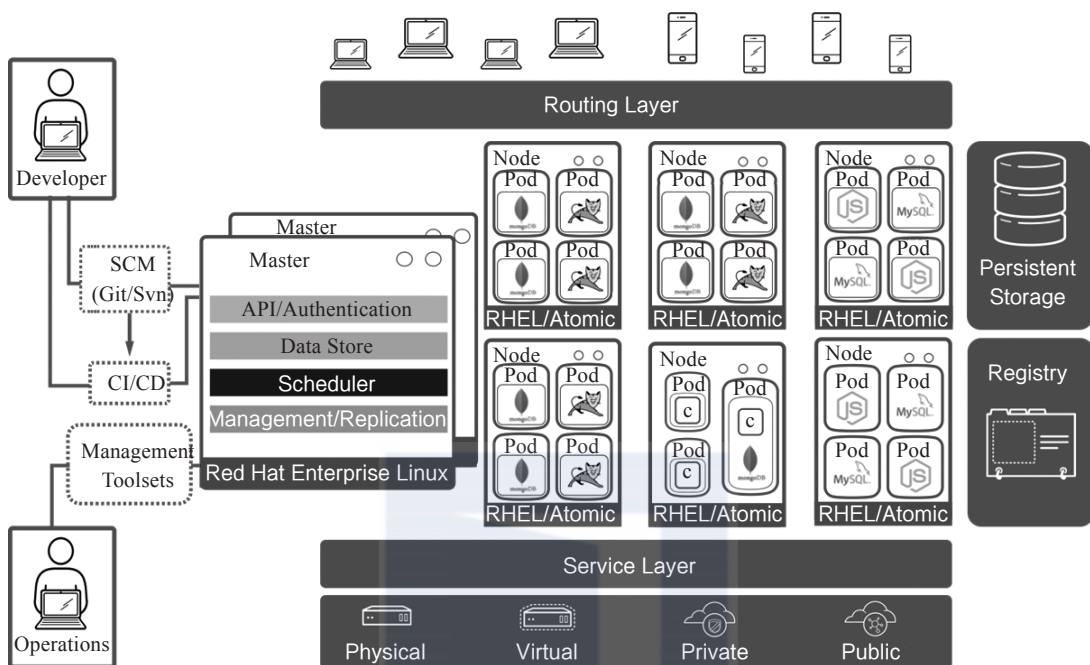


图 3-2 OpenShift 核心组件

图片来源：Red Hat Inc.

### 3.2.1 Master 节点

在介绍 Master 节点前，我们先补充一些内容。OpenShift 集群可以由一台或多台主机组成。这些主机可以是物理机或虚拟机，同时可以运行在私有云、公有云，或混合云上。在 OpenShift 的集群成员有两种角色。

❑ Master 节点：即主控节点。集群内的管理组件均运行于 Master 节点之上。Master 节点负责管理和维护 OpenShift 集群的状态。

❑ Node 节点：即计算节点。集群内的容器实例均运行于 Node 节点之上。

如图 3-2 所示，在 Master 节点上运行了众多集群的服务组件：

❑ API Server。负责提供集群的 Web Console 以及 RESTful API 服务。集群内的所有 Node 节点都会访问 API Server 更新各节点的状态及其上运行的容器的状态。

❑ 数据源（Data Store）。集群所有动态的状态信息都会存储在后端的一个 etcd 分布式数据库中。默认的 etcd 实例安装在 Master 节点上。如有需要，也可以将 etcd 节点部署在集群之外。

❑ 调度控制器（Scheduler）。调度控制器在容器部署时负责按照用户输入的要求寻找合适的计算节点。例如，在前面章节我们部署的 Router 组件需要占用计算节点主机的 80、443 及 1936 端口。部署 Router 时，调度控制器会寻找端口没有被占用的计算节点并分配给 Router 进行部署。除了端口外，用户还能指定如 CPU、内存及标签匹配

等多种调度条件。

- ❑ 复制控制器 (Replication Controller)。对容器云而言, 一个很重要的特性是异常自恢复。复制控制器负责监控当前容器实例的数量和用户部署指定的数量是否匹配。如果容器异常退出, 复制控制器将会发现实际的容器实例数少于部署定义的数量, 从而触发部署新的容器实例, 以恢复原有的状态。

### 3.2.2 Node 节点

在 Node 节点上没有这么多系统组件, 其主要职责就是接收 Master 节点的指令, 运行和维护 Docker 容器。

这里要指出的是, Master 节点本身也是一个 Node 节点, 只是在一般环境中会将其运行容器的功能关闭。但是在我们这个实验集群中, 由于只有一个节点, 所以这个 Master 节点也必须承担运行容器的责任。

通过执行 `oc get node` 命令可以查看系统中的所有节点。

```
[root@master ~]# oc get nodes
NAME                STATUS    AGE
master.example.com   Ready     1d
```



**提示** 查看集群信息需要集群管理员的权限, 请先登录为 `system:admin`。具体方法请查看之前的章节介绍。

可以看到, 目前集群中只有一个节点, 状态是 Ready。通过 `oc describe node master.example.com` 命令查看节点的详细信息。

```
[root@master ~]# oc describe node master.example.com
Name:                master.example.com
Labels:              beta.kubernetes.io/arch=amd64
                    beta.kubernetes.io/os=linux
                    kubernetes.io/hostname=master.example.com
Taints:              <none>
CreationTimestamp:    Sat, 17 Sep 2016 18:24:40 -0400
Phase:
Conditions:
  Type                Status  LastHeartbeatTime   LastTransitionTime   Reason           Message
  ----                -
  OutOfDisk            False   Sat, 17 Sep 2016... Sat, 17 Sep 2016... KubeletHasSufficient...
  MemoryPressure       False   Sat, 17 Sep 2016... Sat, 17 Sep 2016... KubeletHasSufficient...
  Ready               True    Sat, 17 Sep 2016... Sat, 17 Sep 2016... KubeletReadykubel...
Addresses:            192.168.172.167,192.168.172.167
Capacity:
  alpha.kubernetes.io/nvidia-gpu:    0
  cpu:                                1
  memory:                            1868664Ki
  pods:                              110
Allocatable:
```

```

alpha.kubernetes.io/nvidia-gpu:    0
cpu:                                1
memory:                             1868664Ki
pods:                               110
System Info:
Machine ID:                         68b21a5dd46c4a34a8b7f66cd66b3b73
System UUID:                       564D2779-5B04-6978-470E-5796F8DF3ECB
Boot ID:                           0b3c164b-6d0b-4082-b5da-bac31b8f61a2
Kernel Version:                    3.10.0-327.el7.x86_64
OS Image:                          CentOS Linux 7 (Core)
Operating System:                  linux
Architecture:                     amd64
Container Runtime Version: docker://1.10.3
Kubelet Version:                   v1.3.0+52492b4
Kube-Proxy Version:                v1.3.0+52492b4
ExternalID:                        master.example.com
.....

```

从上面的输出可以看到该节点详细的系统信息、节点上运行的容器资源使用情况、网络地址等。

### 3.2.3 Project 与 Namespace

在 Kubernetes 中使用命名空间的概念来分隔资源。在同一个命名空间中，某一个对象的名称在其分类中必须是唯一的，但是分布在不同命名空间中的对象则可以同名。OpenShift 中继承了 Kubernetes 命名空间的概念，而且在其之上定义了 Project 对象的概念。每一个 Project 会和一个 Namespace 相关联，甚至可以简单地认为，Project 就是 Namespace。所以在 OpenShift 中进行操作时，首先要确认当前执行的上下文是哪一个 Project。

通过 `oc project` 命令可以查看用户当前所在的 Project。

```

[root@master ~]# oc project
Using project "default" on server "https://192.168.172.167:8443".

```

通过 `oc project <PROJECT-NAME>` 可以切换到指定的项目。现在请切换到上一章创建的 `helloworld-php` 项目，接下来我们会以这个项目为基础进行讲解。

```

[root@master ~]# oc project hello-world-php
Now using project "hello-world-php" on server "https://192.168.172.167:8443".

```

### 3.2.4 Pod

在 OpenShift 上运行的容器会被一种叫 Pod 的对象所“包裹”，用户不会直接看到 Docker 容器本身。从技术上来说，Pod 其实也是一种特殊的容器。执行 `oc get pods` 命令可以看到当前所在项目的 Pod。

```

[root@master ~]# oc get pod

```

| NAME                          | READY | STATUS    | RESTARTS | AGE |
|-------------------------------|-------|-----------|----------|-----|
| cakephp-mysql-example-1-build | 0/1   | Completed | 0        | 1h  |
| cakephp-mysql-example-1-u63y2 | 1/1   | Running   | 0        | 1h  |

```
mysql-1-jovdm          1/1          Running          0          1h
```

执行 `oc describe pod` 命令可以查看容器的详细信息，如 Pod 部署的 Node 节点名、所处的 Project、IP 地址等。

```
[root@master ~]# oc describe pod mysql-1-jovdm
Name:          mysql-1-jovdm
Namespace:     hello-world-php
Security Policy:  restricted
Node:          master.example.com/192.168.172.167
Start Time:    Sat, 17 Sep 2016 21:00:28 -0400
Labels:        deployment=mysql-1
deploymentconfig=mysql
name=mysql
Status:        Running
IP:            172.17.0.6
Controllers:   ReplicationController/mysql-1
.....
```

用户可以近似认为实际部署的容器会运行在 Pod 内部。一个 Pod 内部可以运行一个或多个容器，运行在一个 Pod 内的多个容器共享这个 Pod 的网络及存储资源。从这个层面上，可以将 Pod 理解为一个虚拟主机，在这个虚拟主机中，用户可以运行一个或多个容器。虽然一个 Pod 内可以有多个容器，但是在绝大多数情况下，一个 Pod 内部只运行一个容器实例。Pod 其实也是一个 Docker 容器，通过 `dockerps` 命令可以查看 Pod 的实例信息。



**提示** 因为大多数情况下都是一个容器运行在一个 Pod 内，很多时候可以将 Pod 等同于我们所要运行的容器本身。

```
[root@master ~]# dockerps |grep php
17c5e7154790          172.30.73.49:5000/hello-world-php/cakephp-mysql-
example@sha256:e3f4705aac3e718c22e4d8d1bf12ab3041d0f417752289ea132e503c
f5adc91d  "container-entrypoint"  About an hour ago  Up About an hour
k8s_cakephp-mysql-example.9fcdalc6_cakephp-mysql-example-1-u63y2_hello-world-
php_382e1e18-7d3c-11e6-a285-000c29df3ecb_ced53322
3eb08d061fa9          openshift/origin-pod:v1.3.0  "/pod"
About an hour ago  Up About an hour
.....
```

上述代码是查找 PHP 的容器实例。一共有两个输出，一个是实际的 PHP 应用的容器实例，另一个是镜像类型为 `openshift/origin-pod:v1.3.0` 的容器，即 Pod 容器实例。

容器像盒子一样为应用提供一个独立的运行环境，但它并不是一个黑盒子。用户可以实时地查看容器的输出，也可以进入容器内部执行操作。

执行 `oc logs <POD NAME>` 命令，可以查看 Pod 的输出。

```
[root@master ~]# oc logs mysql-1-jovdm
```

```

---> 16:22:33      Processing MySQL configuration files ...
---> 16:22:33      Initializing database ...
.....

```

执行 `ocrsh<POD NAME>` 命令，可以进入容器内部执行命令进行调试。

```

[root@master ~]# ocrsh mysql-1-jovdm
sh-4.2$ hostname
mysql-1-jovdm
sh-4.2$ ps ax

```

|       | PID  | TTY  | STAT | TIME | COMMAND                                              |
|-------|------|------|------|------|------------------------------------------------------|
| 1     | ?Ssl | 0:51 |      |      | /opt/rh/rh-mysql56/root/usr/libexec/mysqld --default |
| 43096 | ?Ss  | 0:00 |      |      | /bin/sh                                              |
| 43104 | ?    |      | R+   | 0:00 | ps ax                                                |

`oc logs` 及 `ocrsh` 是两个非常有用的命令，是排查容器相关问题的重要手段。

### 3.2.5 Service

容器是一个一个非持久化的对象。所有对容器的更改在容器销毁后默认都会丢失。同一个 Docker 镜像实例化形成容器后，会恢复到这个镜像定义的状态，并且获取一个新的 IP 地址。容器的这种特性在某些场景下非常难能可贵，但是每个新容器的 IP 地址都在不断变化，这对应用来说不是一件好事。拿前文部署的 PHP 和 MySQL 应用来说，后端 MySQL 容器在重启后 IP 地址改变了，就意味着必须更新 PHP 应用的数据库地址指向。如果不修改应用地址指向，就需要有一种机制使得前端的 PHP 应用总是能连接到后端的 MySQL 数据库。

为了克服容器变化引发的连接信息的变化，Kubernetes 提供了一种叫 Service（服务）的组件。当部署某个应用时，我们会为该应用创建一个 Service 对象。Service 对象会与该应用的一个或多个 Pod 关联。同时每个 Service 会被分配到一个 IP 地址，这个 IP 地址是相对恒定的。通过访问这个 IP 地址及相应的端口，请求就会被转发到对应 Pod 的相应端口。这意味着，无论后端的 Pod 实例的数量或地址如何变化，前端的应用只需要访问 Service 的 IP 地址，就能连接到正确的后端容器实例。Service 起到了代理的作用，在相互依赖的容器应用之间实现了解耦。

通过 `oc get svc` 命令，可以获取当前项目下所有 Service 对象的列表。

```

[root@master ~]# oc get svc

```

| NAME                  | CLUSTER-IP    | EXTERNAL-IP | PORT(S)  | AGE |
|-----------------------|---------------|-------------|----------|-----|
| cakephp-mysql-example | 172.30.1.84   | <none>      | 8080/TCP | 17h |
| mysql                 | 172.30.166.12 | <none>      | 3306/TCP | 17h |

通过 CakePHP 的 Service 的 IP 地址加端口 `172.30.1.84:8080`，可以访问到 CakePHP 的服务。

```

[root@master ~]# curl -q 172.30.1.84:8080

```





**提示** 如果尝试 ping 一下 Service 的 IP 地址，结果是不会成功的。因为 Service 的 IP 地址是虚拟的 IP 地址，而且这个地址只有集群内的节点和容器可以识别。

除了通过 IP 地址访问 Service 所指向的服务外，还可以通过域名访问某一个 Service。监听在 Master 上的集群内置 DNS 服务器会负责解析这个 DNS 请求。Service 域名的格式是 <SERVICE NAME>.<PROJECT NAME>.svc.cluster.local。比如上面例子中的 PHP 应用的 Service 域名将会是 cakephp-mysql-example.helloworld-ng.svc.cluster.local:8080。可以在 Master 节点上用 ping 检查域名解析。

```
[root@master ~]# pingcakephp-mysql-example.helloworld-ng.svc.cluster.local
PING cakephp-mysql-example.helloworld-ng.svc.cluster.local (172.30.1.84) 56(84)
bytes of data.
```

如果发现内部域名解析失败，可以在 /etc/resolv.conf 中添加一条指向本机的域名服务器的记录。

```
nameserver 127.0.0.1
```

### 3.2.6 Router 与 Route

Service 提供了一个通往后端 Pod 集群的稳定的入口，但是 Service 的 IP 地址只是集群内部的节点及容器可见。对于外部的应用或者用户来说，这个地址是不可达的。那么外面的用户想要访问 Service 指向的服务应该怎么办呢？OpenShift 提供了 Router（路由器）来解决这个问题。上一章中介绍了 Router 组件的部署。其实 Router 组件就是一个运行在容器内的 Haproxy，是一个特殊定制的 Haproxy。用户可以创建一种叫 Route 的对象，笔者称为路由规则。一个 Route 会与一个 Service 相关联，并且绑定一个域名。Route 规则会被 Router 加载。当用户通过指定域名访问应用时，域名会被解析并指向 Router 所在的计算节点上。Router 获取这个请求，然后根据 Route 规则定义转发给与这个域名对应的 Service 后端所关联的 Pod 容器实例。在上一章部署 CakePHP 应用时，我们将 Route 域名修改为 php.apps.example.com。当访问域 php.apps.example.com 时，请求到达 Router，并由其向后端分发。当 Pod 的数量或者状态变化时，OpenShift 负责更新 Router 内的配置，确保请求总是能被正确路由到对应的 Pod。

通过命令 `oc get routes`，可以查看项目中的所有 Route。

```
[root@master ~]# oc get route -n hello-world-php
NAME                                HOST/PORT                                PATH    SERVICES    PORT    TERMINATION
cakephp-mysql-example php.apps.example.com    cakephp-mysql-example    <all>
```



**提示** 经常会有用户混淆了 Router 和 Service 的作用。Router 负责将集群外的请求转发到集群的容器。Service 则负责把来自集群内部的请求转发到指定的容器中。一个是对外，

一个是对内。

### 3.2.7 Persistent Storage

容器默认是非持久化的，所有的修改在容器销毁时都会丢失。但现实是传统的应用大多都是有状态的，因此要求某些容器内的数据必须持久化，容器云平台必须为容器提供持久化储存（persistent storage）。Docker 本身提供了持久化卷挂载的能力。相对于单机容器的场景，在容器云集群的场景中，持久化的实现有更多细节需要考虑。OpenShift 除了支持 Docker 持久化卷的挂载方式外，还提供了一种持久化供给模型，即 Persistent Volume（持久化卷，PV）及 Persistent Volume Claim（持久化卷请求，PVC）模型。在 PV 和 PVC 模型中，集群管理员会创建大量不同大小和不同特性的 PV。用户在部署应用时，显式声明对持久化的需求，创建 PVC。用户在 PVC 中定义所需存储的大小、访问方式（只读或可读可写；独占或共享）。OpenShift 集群会自动寻找符合要求的 PV 与 PVC 自动对接。通过 PV 和 PVC 模型，OpenShift 为用户提供了一种灵活的方式来消费存储资源。

OpenShift 对持久化后端的支持比较广泛，除了 NFS 及 iSCSI 外，还支持如 Ceph、GlusterFS 等的分布式储存，以及 Amazon WebService 和 Google Compute Engine 的云硬盘。关于存储相关的话题，在后续章节会有更详细的探讨。

### 3.2.8 Registry

OpenShift 提供了一个内部的 Docker 镜像仓库（Registry），该镜像仓库用于存放用户通过内置的 Source to Image 镜像构建流程所产生的镜像。Registry 组件默认以容器的方式提供，在上一章中，我们手工部署了 Registry 组件。

通过 `oc get pod -n default` 命令可以查看 Registry 容器的状态。

```
[root@master ~]# oc get pod -n default
```

| NAME                    | READY | STATUS  | RESTARTS | AGE |
|-------------------------|-------|---------|----------|-----|
| docker-registry-1-xm3un | 1/1   | Running | 1        | 7h  |
| router-1-e95qa          | 1/1   | Running | 1        | 7h  |

通过 `oc get svc -n default` 命令可以查看 Registry 容器对应的 Service 信息。

```
[root@master ~]# oc get svc -n default
```

| NAME            | CLUSTER-IP   | EXTERNAL-IP | PORT(S)                 | AGE |
|-----------------|--------------|-------------|-------------------------|-----|
| docker-registry | 172.30.73.49 | <none>      | 5000/TCP                | 7h  |
| kubernetes      | 172.30.0.1   | <none>      | 443/TCP,53/UDP,53/TCP   | 9h  |
| router          | 172.30.58.19 | <none>      | 80/TCP,443/TCP,1936/TCP | 7h  |

每当 S2I 完成镜像构建，就会向内部的镜像仓库推送构建完成的镜像。在上面的输出示例中，镜像仓库的访问点为 172.30.73.49:5000。如果查看上一章 CakePHP 的 S2I 构建日志，就会看到最后有成功推送镜像的日志输出：Push successful。

```
[root@master ~]# oc logs cakephp-mysql-example-1-build -n hello-world-php
Cloning "https://github.com/openshift/cakephp-ex.git" ...
Commit: 701d706b7f2b50ee972d0bf76990042f6c0cda5c (Merge pull request #42 from
bparees/recreate)
Author: Ben Parees<bparees@users.noreply.github.com>
Date: Mon Aug 22 14:44:49 2016 -0400
---> Installing application source...
Pushing image 172.30.73.49:5000/hello-world-php/cakephp-mysql-example:latest ...
Pushed 0/10 layers, 1% complete
Pushed 1/10 layers, 50% complete
Pushed 2/10 layers, 50% complete
Pushed 3/10 layers, 50% complete
Pushed 4/10 layers, 50% complete
Pushed 5/10 layers, 50% complete
Pushed 6/10 layers, 61% complete
Pushed 7/10 layers, 71% complete
Pushed 7/10 layers, 78% complete
Pushed 8/10 layers, 85% complete
Pushed 8/10 layers, 91% complete
Pushed 8/10 layers, 97% complete
Pushed 9/10 layers, 99% complete
Pushed 10/10 layers, 100% complete
Push successful
```



提示 一个常见的疑问是“是不是 OpenShift 用到的镜像都要存放到内置的仓库?”答案是肯定的。内部的镜像仓库存放的只是 S2I 产生的镜像。其他镜像可以存放在集群外部的镜像仓库,如企业的镜像仓库或社区的镜像仓库。只要保证 OpenShift 的节点可以访问到这些镜像所在的镜像仓库即可。

### 3.2.9 Source to Image

前文多次提及 Source to Image (S2I), 因为 S2I 的确是 OpenShift 的一个重要功能。容器镜像是容器云的应用交付格式。容器镜像中包含了应用及其所依赖的运行环境。可以从社区或者第三方厂商获取基础的操作系统或者中间件的镜像。但是这些外部获取的操作系统或中间件的镜像并不包含企业内部开发和定制的应用。企业内部的开发人员必须自行基于外部的基础镜像构建包含企业自身开发的应用。这个镜像的构建过程是必须的, 要么由企业的 IT 人员手工完成, 要么使用某种工具实现自动化。

作为一个面向应用的平台, OpenShift 提供了 S2I 的流程, 使得企业内容器的构建变得标准化和自动化, 从而提高了软件从开发到上线的效率。一个典型的 S2I 流程包含了以下几个步骤。

- 1) 用户输入源代码仓库的地址。
- 2) 用户选择 S2I 构建的基础镜像(又称为 Builder 镜像)。Builder 镜像中包含了操作

系统、编程语言、框架等应用所需的软件及配置。OpenShift 默认提供了多种编程语言的 Builder 镜像，如 Java、PHP、Ruby、Python、Perl 等。用户也可以根据自身需求定制自己的 Builder 镜像，并发布到服务目录中供用户选用。

- 3) 用户或系统触发 S2I 构建。OpenShift 将实例化 S2I 构建执行器。
- 4) S2I 构建执行器将从用户指定的代码仓库下载源代码。
- 5) S2I 构建执行器实例化 Builder 镜像。代码将会被注入 Builder 镜像中。
- 6) Builder 镜像将根据预定义的逻辑执行源代码的编译、构建并完成部署。
- 7) S2I 构建执行器将完成操作的 Builder 镜像并生成新的 Docker 镜像。
- 8) S2I 构建执行器将新的镜像推送到 OpenShift 内部的镜像仓库。
- 9) S2I 构建执行器更新该次构建相关的 Image Stream 信息。

S2I 构建完成后，根据用户定义的部署逻辑，OpenShift 将把镜像实例化部署到集群中。

除了接受源代码仓库地址作为输入外，S2I 还接受 Dockerfile 以及二进制文件作为构建的输入。用户甚至可以完全自定义构建逻辑来满足特殊的需求。

### 3.2.10 开发及管理工具集

OpenShift 提供了不同的工具集为开发和运维的用户提供良好的体验，也为持续集成和打通 DevOps 流程提供便利。例如，OpenShift 提供了 Eclipse 插件，开发工程师可以在 Eclipse 中完成应用及服务的创建和部署、远程调试、实时日志查询等功能。

## 3.3 核心流程详解

OpenShift 容器云提供了众多基础设施和工具，承载了众多功能和特性，帮助用户通过这个平台提升企业 IT 的效率和敏捷度。纵观 OpenShift 容器云项目，其中最重要的核心流程是将应用从静态的源代码变成动态的应用服务的过程。前文介绍的 OpenShift 及 Kubernetes 的核心组件和概念都是为了支持和实现这个过程而引入的。

应用部署到应用上线响应用户请求的全流程如图 3-3 所示。这个流程涉及了多种不同类型的 OpenShift 对象。所有对象的信息最终都记录在 etcd 集群数据库中。

### 3.3.1 应用构建

- ❑ 第 1 步，部署应用。流程的开始是用户通过 OpenShift 的 Web 控制台或命令行 `oc new-app` 创建应用。根据用户提供的源代码仓库地址及 Builder 镜像，平台将生成构建配置 (Build Config)、部署配置 (Deployment Config)、Service 及 Route 等对象。
- ❑ 第 2 步，触发构建。应用相关的对象创建完毕后，平台将触发一次 S2I 构建。
- ❑ 第 3 步，实例化构建。平台依据应用的 Build Config 实例化一次构建，生成一个 Build 对象。Build 对象生成后，平台将执行具体的构建操作，包括下载源代码、实例

化 Builder 镜像、执行编译和构建脚本等。

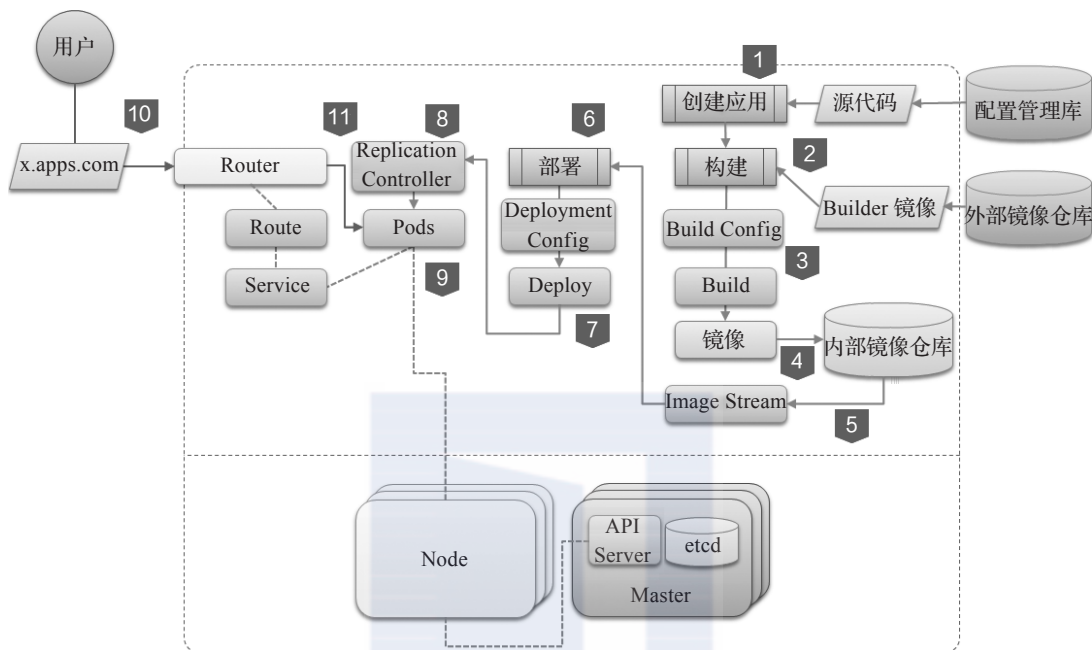


图 3-3 OpenShift 核心组件及流程

- ❑ 第 4 步，生成镜像。构建成功后将生成一个可供部署的应用容器镜像。平台将把此镜像推送到内部的镜像仓库组件 Registry 中。
- ❑ 第 5 步，更新 Image Stream。镜像推送至内部的仓库后，平台将创建或更新应用的 Image Stream 的镜像信息，使之指向最新的镜像。

### 3.3.2 应用部署

- ❑ 第 6 步，触发镜像部署。当 Image Stream 的镜像信息更新后，将触发平台部署 S2I 构建生成的镜像。
- ❑ 第 7 步，实例化镜像部署。Deployment Config 对象记录了部署的定义，平台将依据此配置实例化一次部署，生成一个 Deploy 对象跟踪当次部署的状态。
- ❑ 第 8 步，生成 Replication Controller。平台部署将实例化一个 Replication Controller，用以调度应用容器的部署。
- ❑ 第 9 步，部署容器。通过 Replication Controller，OpenShift 将 Pod 及应用容器部署到集群的计算节点中。

### 3.3.3 请求处理

- ❑ 第 10 步，用户访问。用户通过浏览器访问 Route 对象中定义的应用域名。

- 第 11 步，请求处理并返回。请求到 Router 组件后，Router 根据 Route 定义的规则，找到请求所含域名相关联的 Service 的容器，并将请求转发给容器实例。容器实例除了请求后返回数据，还会通过 Router 将数据返回给调用的客户端。

### 3.3.4 应用更新

在应用更新时，平台将重复上述流程的第 1 步至第 9 步。平台将用下载更新后的代码构建应用，生成新的镜像，并将镜像部署至集群中。值得注意的是，OpenShift 支持滚动更新。在第 9 步时，平台将通过滚动更新的方式，保证应用在新老实例交替时服务不间断。关于滚动更新的细节，在后面的章节将会有更详细的讨论。

## 3.4 本章小结

本章探讨了 OpenShift 的技术架构堆栈，了解了技术堆栈中各层的内容以及作用。结合上一章的内容，我们讲解了 OpenShift 集群中重要的组件及核心概念。了解这些知识后，在后面的章节里，我们将一起了解如何利用 OpenShift 的这些组件来解决实际构建企业容器云平台开发和运维中遇到的问题。

