Openshift简介与安装

OpenShift是基于Docker和Kubernetes构建的开源的容器云，是帮助企业、组织搭建及管理基于容器的应用平台而产生的解决方案。通过OpenShift，企业可以快速搭建稳定、安全、高效的应用平台，在这个平台上：

* 可以构建企业内部的容器应用市场，为开发人员快速提供应用开发所依赖的中间件、数据库等服务
* 通过自动化的流程，开发人员可以快速进行应用的构建、容器化和部署
* 通过OpenShift，用户可以贯通从应用开发到测试，再到上线的全流程，开发、测试和运维等不同的角色可以在一个平台上进行协作
* OpenShift可以提高应用从研发到上线的效率和速度，缩短产品上市的时间，可以有效地帮助企业推进DevOps，提升生产效率

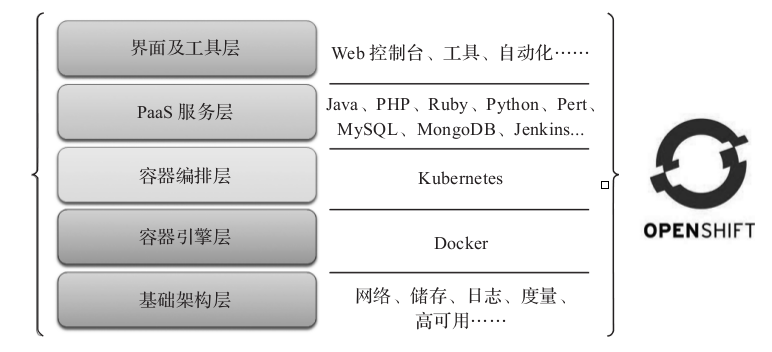
OpenShift底层以Docker作为容器引擎驱动，以Kubernetes作为容器编排引擎组件，为业务提供的功能包括：

* 容器调度，按业务的要求快速部署容器到指定的目标
* 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

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，

# OpenShift系统架构

从技术堆栈的角度分析，作为容器云，OpenShift自定而上包含了以下几个层次：



**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，支持的运行环境包括：物理机、虚拟机、基础架构云（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。

**2）容器引擎层**

OpenShift目前以Docker作为平台的容器引擎，完全基于原生的Docker。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容器的实例。

**3）容器编排层**

OpenShift使用Kubernetes为容器编排引擎，满足大规模集群环境下对容器的调度和部署的需求。OpenShift平台上的许多对象和概念都是衍生子Kubernetes，如Pod、Namespace、Replication Controller等。在OpenShift集群上可以通过Kubernetes的原生命令来操作原生对象。

**4）PaaS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，不能作为一个完整的PaaS解决方案。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、分布式换成或者业务规则引擎的服务。

**5）界面及工具层**

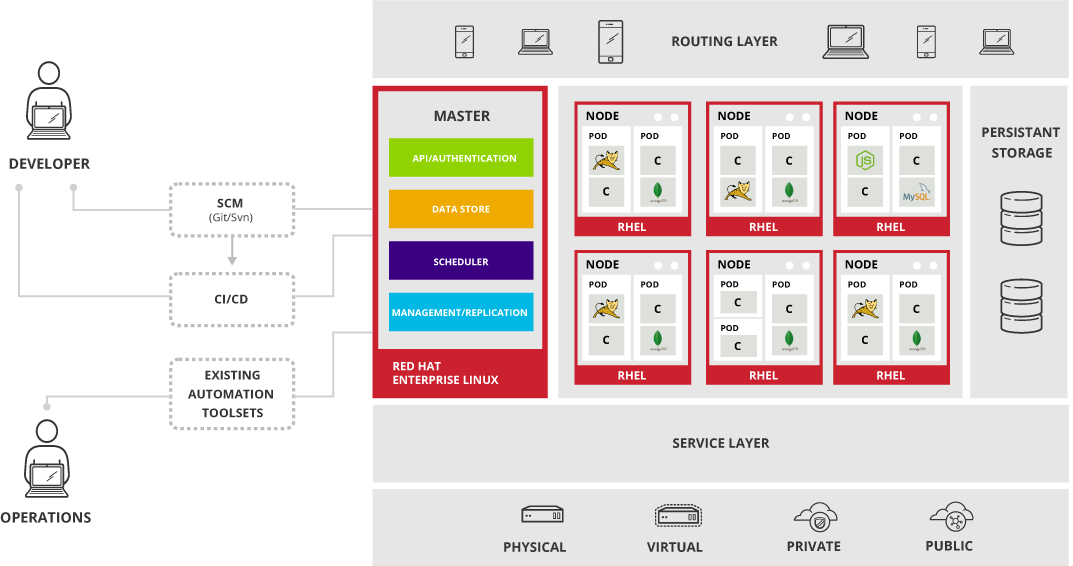
界面和工具可以帮助云平台更高效的帮助用户完成相关的任务，OpenShift提供了自动化流程Source to Image(S2I)，帮助用户容器化用各种编程语言开发的应用源代码。用户可以直接使用S2I或者把所有流程整合到S2I，从而实现开发流程的持续集成和持续交付。提高开发、测试和部署的自动化程度，最终提高开发、测试和部署的效率。OpenShift还提供了多种用户的接入渠道：Web控制台、命令行、IDE集成及RESTFul编程接口。

针对容器应用的运维及集群的运维，OpenShift还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。

<https://yq.aliyun.com/articles/110419>

# 核心组件详解

OpenShift的核心组件与其之间的关联关系如下图所示：



OpenShift容器平台是一个基于微服务的架构，它是一个小型的、解耦的单元，可以协调工作。其运行在Kuberntes集群顶部，相关的对象数据存储在Key-Value数据库etcd中。OpenShift集群有一台或者多台主机组成，这些主机可以是物理机、虚拟机及云主机上。

在OpenShift上的集群有两种角色Master和Node：

1. **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，负责管理和维护OpenShift集群状态。在Master节点上运行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状态。
* 数据源（Data Source），集群内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，如果有需要也可以部署在集群外。
* 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。调度的资源包括端口、CPU、内存及标签等。
* 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，以恢复原有状态。

1. **Node节点**，即计算节点，其职责是接收Master节点的指令，运行和维护Docker容器群内的容器实例都运行于Node节点之上。Master本身也是一个Node节点，只是在一般情况下回将其运行容器的功能关闭。

https://access.redhat.com/documentation/en-us/openshift\_enterprise/3.0/html-single/architecture/index

## 2.1 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继承了Kubernetes命名空间的概念，而且在其上定义了Project对象的概念，每一个Project会和一个NameSpace相关联，甚至可以简单的认为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。

## 2.2 Pod

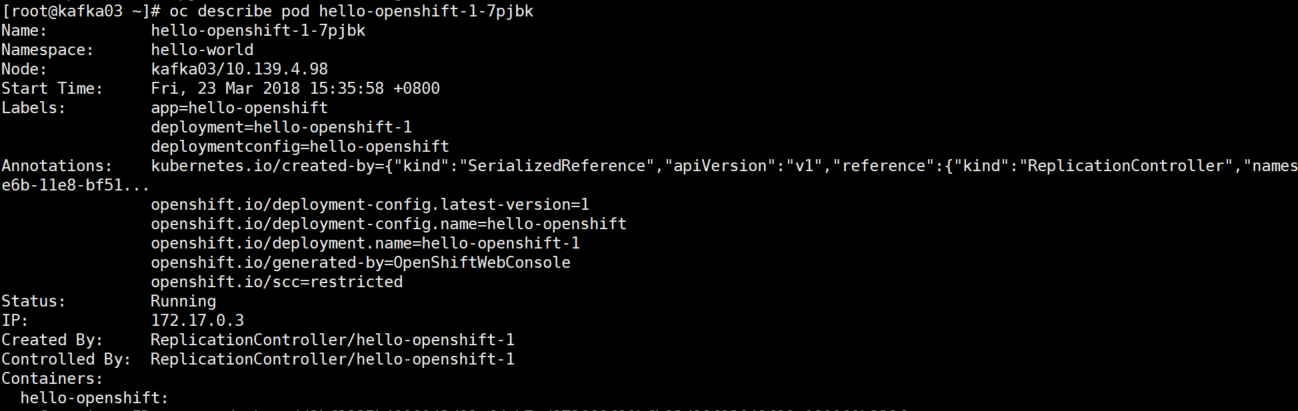
在OpenShift上运行的容器会被叫Pod对象所”包裹”，用户不会直接看到Docker容器本身，从技术上看，Pod其实是一种特殊的容器，执行oc get pod命令可以查看当前项目的容器列表，和Kubernetes一样：

*# oc get pod*

*NAME READY STATUS RESTARTS AGE*

*hello-openshift-1-7pjbk 1/1 Running 0 1h*

可以执行oc describe pod命令查看pod的详细信息，包含容器的名称、状态和所处的命名空间等



用户可以近似的认为实际部署的容器会运行的Pod内部，一个Pod内部可以运行一个或者多个容器，运行在一个pod内的多个容器共享这个pod的网络和存储资源。可以将Pod理解为一个虚拟机主机，在这个虚拟主机上可以运行一个或者多个容器。

## 2.3 Service

容器是一个非持久化的对象，所有对其容器的更改在容器销毁后默认后会丢失。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，并且获得一个新的IP地址。容器的这种特性在某些场景下比较有用，但是每个新容器的IP地址都在不断变化，这对应用来说不好维护。

为了克服容器变化引起的连接信息的变化，Kubernetes提供了Service的组件，当部署某个应用时，为该应用创建一个Service对象。Service对象会与该应用的一个或者多个Pod关联，同时为Service分配一个IP地址，这个IP地址是相对稳定的。通过访问这个IP地址和相应的端口，请求就会被转发到对应的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，可以获取当前项目的所有Service对象的列表：

*# oc get svc*

*NAME CLUSTER-IP EXTERNAL-IP PORT(S) AGE*

*hello-openshift 172.30.110.36 <none> 8080/TCP,8888/TCP 2h*

除了通过IP地址访问Service所指向的服务外，还可以通过域名来访问某个Service。监听在Master上的集群内置DNS服务会负责解析这个DNS请求。

## 2.4 Router与Route

Service提供了一个通向后端Pod集群稳定的入口，但是Service的IP地址只是集群内部的节点和容器可见。对于外部的应用或者应用来说，这个地址是不可达的。OpenShift提供了Router(路由器)来解决这个问题，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Route对象，可以看做是路由规则。

一个Router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规则定义转发给这个域名对应的Server后端所关联的Pod容器实例。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。

## 2.5 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，但是现实是传统的应用大多是由状态的，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存储（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的细节需要考虑。

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要，创建PVC。用户在PVC中定义所需存储的大小、访问方式（只读或者可读可写；独占或共享）。OpenShift集群会自动寻找符合要求的PV和PVC自动对接，通过PV和PVC模型，OpenShift为用户提供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和iSCSI外，还支持如Ceph、GlusterFS等的分布式存储，以及Amazon WebServic和Google Compute Engine的云硬盘。

## 2.6 Registry及S2I

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。

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，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。

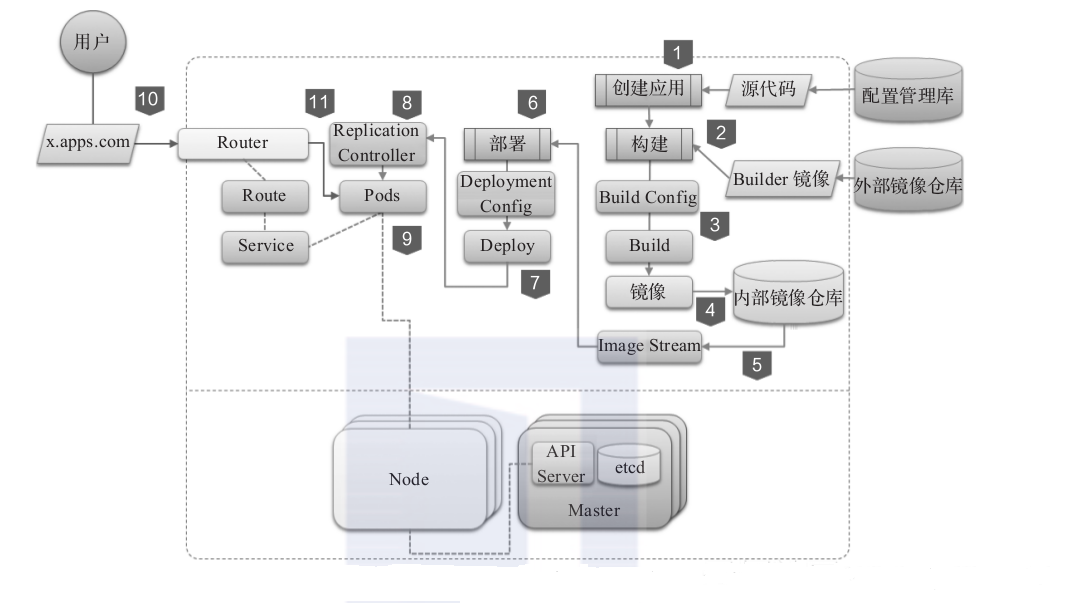
容器镜像是容器云的应用交付格式，包含了应用及其所依赖的运行环境，可以从社区或者第三方厂商获取基础的操作系统或中间件镜像。但是企业内部的开发人员必须自行基于外部的基础镜像构建包含企业自身开发的应用，一个典型的S2I流程包含了以下几个步骤：

1. 用户输入源代码仓库的地址
2. 用户选择S2I构建基础镜像（Builder镜像），包含了操作系统、编程语言、框架都应用所需要的软件及配置。OpenShift默认提供额多种编程语言的Builder镜像，如Java、PHP、Ruby、Python等，用户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。

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出，用户甚至可以完全自定义构建逻辑来满足特殊的需求。

# 核心流程详解

OpenShift容器云项目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，应用部署到应用上线响应用户请求的全流程如下图所示：



这个流程涉及了多种不同类型的OpenShift对象，所有的对象的信息最终都记录在etcd集群数据库中。

## 3.1 应用构建

1. 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）、Server及Route等对象。
2. 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
3. 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，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、执行编译及构建脚本等。
4. 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
5. 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。

## 3.2 应用部署

1. 触发镜像部署，当Image Stream的镜像信息更新后，将镜像平台部署S2I构建生成的镜像。
2. 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
3. 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
4. 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 请求处理