目录

**[一、 云原生发展简史](#_Toc1123426680_WPSOffice_Level1)** **[4](#_Toc1123426680_WPSOffice_Level1)**

[1． 云原生是什么？](#_Toc549945810_WPSOffice_Level2) [4](#_Toc549945810_WPSOffice_Level2)

[2． 云原生价值和愿景？](#_Toc169611982_WPSOffice_Level2) [4](#_Toc169611982_WPSOffice_Level2)

[3． 云原生思想的两个理论](#_Toc957781905_WPSOffice_Level2) [4](#_Toc957781905_WPSOffice_Level2)

[4． 云原生的技术范畴](#_Toc2050543070_WPSOffice_Level2) [4](#_Toc2050543070_WPSOffice_Level2)

**[二、 K8s 容器基本概念](#_Toc549945810_WPSOffice_Level1)** **[5](#_Toc549945810_WPSOffice_Level1)**

[1. 进程：](#_Toc659810434_WPSOffice_Level2) [5](#_Toc659810434_WPSOffice_Level2)

[2. 独立运行环境](#_Toc1975894777_WPSOffice_Level2) [5](#_Toc1975894777_WPSOffice_Level2)

[3. 什么是容器](#_Toc176399831_WPSOffice_Level2) [5](#_Toc176399831_WPSOffice_Level2)

[4. 什么是镜像](#_Toc1224526757_WPSOffice_Level2) [5](#_Toc1224526757_WPSOffice_Level2)

[5. changeset 的分层以及复用特点](#_Toc1285415698_WPSOffice_Level2) [5](#_Toc1285415698_WPSOffice_Level2)

[6. 如何构建镜像](#_Toc296147466_WPSOffice_Level2) [5](#_Toc296147466_WPSOffice_Level2)

[7. 如何运行容器](#_Toc1630850963_WPSOffice_Level2) [5](#_Toc1630850963_WPSOffice_Level2)

[8. 数据卷管理](#_Toc1378348480_WPSOffice_Level2) [5](#_Toc1378348480_WPSOffice_Level2)

**[三、K8S](#_Toc169611982_WPSOffice_Level1)** **[7](#_Toc169611982_WPSOffice_Level1)**

[1． 什么是k8s?](#_Toc996803171_WPSOffice_Level2) [7](#_Toc996803171_WPSOffice_Level2)

[2． K8s核心功能](#_Toc750964750_WPSOffice_Level2) [7](#_Toc750964750_WPSOffice_Level2)

[3． K8s核心架构](#_Toc703159831_WPSOffice_Level2) [7](#_Toc703159831_WPSOffice_Level2)

[4． Node](#_Toc404770176_WPSOffice_Level2) [7](#_Toc404770176_WPSOffice_Level2)

[5． Kubernetes 的核心概念](#_Toc1891637983_WPSOffice_Level2) [7](#_Toc1891637983_WPSOffice_Level2)

[6． K8s API](#_Toc1411670093_WPSOffice_Level2) [8](#_Toc1411670093_WPSOffice_Level2)

[7． Pod是什么](#_Toc539920995_WPSOffice_Level2) [8](#_Toc539920995_WPSOffice_Level2)

[8． Pod要解决的问题以及怎么解决？](#_Toc1333754390_WPSOffice_Level2) [8](#_Toc1333754390_WPSOffice_Level2)

[9． 为什么Pod必须是原子调度单位？](#_Toc975725344_WPSOffice_Level2) [8](#_Toc975725344_WPSOffice_Level2)

[10． 超亲密关系？](#_Toc830728116_WPSOffice_Level2) [8](#_Toc830728116_WPSOffice_Level2)

[11． 容器设计模式](#_Toc1256256465_WPSOffice_Level2) [8](#_Toc1256256465_WPSOffice_Level2)

[12． 总结](#_Toc1990673598_WPSOffice_Level2) [9](#_Toc1990673598_WPSOffice_Level2)

**[四、 K8s应用编排与管理](#_Toc957781905_WPSOffice_Level1)** **[10](#_Toc957781905_WPSOffice_Level1)**

[1、 资源元信息](#_Toc1603424973_WPSOffice_Level2) [10](#_Toc1603424973_WPSOffice_Level2)

[2、 操作命令](#_Toc2138718655_WPSOffice_Level2) [10](#_Toc2138718655_WPSOffice_Level2)

[3、 控制器模式](#_Toc863151099_WPSOffice_Level2) [10](#_Toc863151099_WPSOffice_Level2)

[4、 控制器模式总结](#_Toc728485408_WPSOffice_Level2) [11](#_Toc728485408_WPSOffice_Level2)

[5、 Deployment是什么？](#_Toc849980709_WPSOffice_Level2) [12](#_Toc849980709_WPSOffice_Level2)

[6、 Deployment要解决的问题？](#_Toc564556319_WPSOffice_Level2) [12](#_Toc564556319_WPSOffice_Level2)

[7、 Spec字段解析](#_Toc915300987_WPSOffice_Level2) [12](#_Toc915300987_WPSOffice_Level2)

[8、 升级策略字段解析](#_Toc1038325048_WPSOffice_Level2) [13](#_Toc1038325048_WPSOffice_Level2)

[9、 Deployment总结](#_Toc676966214_WPSOffice_Level2) [13](#_Toc676966214_WPSOffice_Level2)

[10、 Job](#_Toc402796892_WPSOffice_Level2) [13](#_Toc402796892_WPSOffice_Level2)

[11、 CronJob](#_Toc938908500_WPSOffice_Level2) [14](#_Toc938908500_WPSOffice_Level2)

[12、 DaemonSet](#_Toc525321344_WPSOffice_Level2) [15](#_Toc525321344_WPSOffice_Level2)

**[五、 配置管理](#_Toc2050543070_WPSOffice_Level1)** **[16](#_Toc2050543070_WPSOffice_Level1)**

[1、 背景](#_Toc770555791_WPSOffice_Level2) [16](#_Toc770555791_WPSOffice_Level2)

[2、 实现方式](#_Toc1404787927_WPSOffice_Level2) [16](#_Toc1404787927_WPSOffice_Level2)

[3、 ConfigMap](#_Toc835473971_WPSOffice_Level2) [16](#_Toc835473971_WPSOffice_Level2)

[4、 Secret](#_Toc1562946511_WPSOffice_Level2) [17](#_Toc1562946511_WPSOffice_Level2)

[5、 ServiceAccount](#_Toc422040273_WPSOffice_Level2) [17](#_Toc422040273_WPSOffice_Level2)

[6、 Resources](#_Toc92382270_WPSOffice_Level2) [17](#_Toc92382270_WPSOffice_Level2)

[7、 SecurityContext](#_Toc38135109_WPSOffice_Level2) [18](#_Toc38135109_WPSOffice_Level2)

[8、 InitContainer](#_Toc986650157_WPSOffice_Level2) [18](#_Toc986650157_WPSOffice_Level2)

**[六、 存储与持久化](#_Toc659810434_WPSOffice_Level1)** **[19](#_Toc659810434_WPSOffice_Level1)**

[1、 Pod Volumes](#_Toc1907950212_WPSOffice_Level2) [19](#_Toc1907950212_WPSOffice_Level2)

[2、 PV（Persistent Volumes）](#_Toc693396080_WPSOffice_Level2) [19](#_Toc693396080_WPSOffice_Level2)

[3、 PVC（Persistent Volume Claim）](#_Toc1661647938_WPSOffice_Level2) [19](#_Toc1661647938_WPSOffice_Level2)

[4、 Static PV provisioning](#_Toc1439548378_WPSOffice_Level2) [20](#_Toc1439548378_WPSOffice_Level2)

[5、 Dynamic PV provisioning](#_Toc938821944_WPSOffice_Level2) [20](#_Toc938821944_WPSOffice_Level2)

[6、 Spec重要字段解析](#_Toc1218058299_WPSOffice_Level2) [20](#_Toc1218058299_WPSOffice_Level2)

[7、 PV状态流转](#_Toc2091708089_WPSOffice_Level2) [20](#_Toc2091708089_WPSOffice_Level2)

[8、 PV和PVC架构设计](#_Toc1030550433_WPSOffice_Level2) [21](#_Toc1030550433_WPSOffice_Level2)

**[七、 监控与日志](#_Toc1975894777_WPSOffice_Level1)** **[22](#_Toc1975894777_WPSOffice_Level1)**

**[八、 网络](#_Toc176399831_WPSOffice_Level1)** **[22](#_Toc176399831_WPSOffice_Level1)**

**[九、 服务发现与负载均衡](#_Toc1224526757_WPSOffice_Level1)** **[22](#_Toc1224526757_WPSOffice_Level1)**

**[十、 Etcd](#_Toc1285415698_WPSOffice_Level1)** **[22](#_Toc1285415698_WPSOffice_Level1)**

**[十一、 调度与资源管理](#_Toc296147466_WPSOffice_Level1)** **[22](#_Toc296147466_WPSOffice_Level1)**

# 云原生发展简史

1. 云原生是什么？

为用户指定了一条低心智负担的、敏捷的、能够以可扩展、可复制的方式最大化地利用云的能力、发挥云的价值的最佳路径。

1. 云原生价值和愿景？

就是认为未来的软件，会从诞生起就生长在云上，并且遵循一种新的软件开发、发布和运维模式，从而使得软件能够最大化地发挥云的能力。

1. 云原生思想的两个理论

1）第一个理论基础是：不可变基础设施。2）第二个理论基础就是：云应用编排理论。

1. 云原生的技术范畴

1）第一部分是云应用定义与开发流程。2）第二部分是云应用的编排与管理流程。3）第三部分是监控与可观测性。4）第四部分就是云原生的底层技术，5）第五部分是云原生工具集6）最后则是 Serverless。

# 二、 K8s 容器基本概念

1. 进程：

第一，这些进程可以相互看到、相互通信；【不安全】

第二，它们使用的是同一个文件系统，可以对同一个文件进行读写操作；【共享文件系统】

第三，这些进程会使用相同的系统资源。【资源抢占】

1. 独立运行环境

Chroot：子目录变成根目录，实现视图级别隔离，进程具有独立文件系统

Namespace： 进程在资源的视图上进行隔离

Cgroup： 来限制其资源使用率

1. 什么是容器

容器就是一个视图隔离、资源可限制、独立文件系统的进程集合。

1. 什么是镜像

容器运行时所需要的所有的文件集合称之为容器镜像。

1. changeset 的分层以及复用特点

能够提高分发效率，并行下载；

下载缺失的即可，本地没有的；

节约磁盘空间；

1. 如何构建镜像
2. FROM 行表示以下的构建步骤基于什么镜像进行构建，正如前面所提到的，镜像是可以复用的；
3. WORKDIR 行表示会把接下来的构建步骤都在哪一个相应的具体目录下进行，其起到的作用类似于 Shell 里面的 cd；
4. COPY 行表示的是可以将宿主机上的文件拷贝到容器镜像内；
5. RUN 行表示在具体的文件系统内执行相应的动作。当我们运行完毕之后就可以得到一个应用了；
6. CMD 行表示使用镜像时的默认程序名字。
7. 如何运行容器

第一步：从镜像仓库中将相应的镜像下载下来；

第二步：当镜像下载完成之后就可以通过 docker images 来查看本地镜像，这里会给出一个完整的列表，我们可以在列表中选中想要的镜像；

第三步：当选中镜像之后，就可以通过 docker run 来运行这个镜像得到想要的容器，当然可以通过多次运行得到多个容器。一个镜像就相当于是一个模板，一个容器就像是一个具体的运行实例，因此镜像就具有了一次构建、到处运行的特点。

1. 数据卷管理

第一种是通过 bind 的方式，直接将宿主机的目录直接挂载到容器内；这种方式比较简单，但是会带来运维成本，因为其依赖于宿主机的目录，需要对于所有的宿主机进行统一管理。

第二种是将目录管理交给运行引擎。

# 三、K8S

1. 什么是k8s?

Kubernetes 是一个自动化的容器编排平台，它负责应用的部署、应用的弹性以及应用的管理，这些都是基于容器的。

1. K8s核心功能

服务的发现与负载的均衡；

调度

自动化的容器的恢复

自动发布与回滚，配置密文管理

job批量执行

水平伸缩

1. K8s核心架构

API Server：顾名思义是用来处理 API 操作的，Kubernetes 中所有的组件都会和 API Server 进行连接，组件与组件之间一般不进行独立的连接，都依赖于 API Server 进行消息的传送；

Controller：是控制器，它用来完成对集群状态的一些管理。比如刚刚我们提到的两个例子之中，第一个自动对容器进行修复、第二个自动进行水平扩张，都是由 Kubernetes 中的 Controller 来进行完成的；

Scheduler：是调度器，“调度器”顾名思义就是完成调度的操作，就是我们刚才介绍的第一个例子中，把一个用户提交的 Container，依据它对 CPU、对 memory 请求大小，找一台合适的节点，进行放置；

etcd：是一个分布式的一个存储系统，API Server 中所需要的这些原信息都被放置在 etcd 中，etcd 本身是一个高可用系统，通过 etcd 保证整个 Kubernetes 的 Master 组件的高可用性。

1. Node



1. Kubernetes 的核心概念

Pod：Pod 是 Kubernetes 的一个最小调度以及资源单元。一个 Pod 简单来说是对一组容器的抽象，它里面会包含一个或多个容器。

Volume：声明在 Pod 中容器可以访问文件目录

Deployment：定义一组 Pod 的副本数目、以及这个 Pod 的版本

Service：提供了一个或者多个 Pod 实例的稳定访问地址

Namespace：一个集群内部的逻辑隔离机制（鉴权、资源额度等）

1. K8s API

从 high-level 上看，Kubernetes API 是由 HTTP+JSON 组成的：用户访问的方式是 HTTP，访问的 API 中 content 的内容是 JSON 格式的

Label：这个 label 可以是一组 KeyValuePair。

1. Pod是什么

逻辑概念：“进程组”

1. Pod要解决的问题以及怎么解决？

问题：

我们怎么去描述超亲密关系；

我们怎么去对超亲密关系的容器或者说是业务去做统一调度，这是 Pod 最主要的一个诉求。

解决方法：

共享网络：Infra container，Join Namespace加入到infra container的Network namepsace中

共享存储：volume，Pod Level

1. 为什么Pod必须是原子调度单位？

Task co-scheduling问题：Mesos解法是资源囤积，Google解法是乐观调度，冲突会回滚，但是对于Pod为单位调度，这个问题根本不存在！

1. 超亲密关系？

比如说两个进程之间会发生文件交换，前面提到的例子就是这样，一个写日志，一个读日志；

两个进程之间需要通过 localhost 或者说是本地的 Socket 去进行通信，这种本地通信也是超亲密关系；

这两个容器或者是微服务之间，需要发生非常频繁的 RPC 调用，出于性能的考虑，也希望它们是超亲密关系；

两个容器或者是应用，它们需要共享某些 Linux Namespace。最简单常见的一个例子，就是我有一个容器需要加入另一个容器的 Network Namespace。这样我就能看到另一个容器的网络设备，和它的网络信息。

像以上几种关系都属于超亲密关系，它们都是在 Kubernetes 中会通过 Pod 的概念去解决的

1. 容器设计模式

Sidecar：在 Pod 里面，可以定义一些专门的容器，来执行主业务容器所需要的一些辅助工作

用途：1）应用日志收集；2）代理容器；3）容器适配器

1. 总结

Pod 是 Kubernetes 项目里实现“容器设计模式”的核心机制；

“容器设计模式”是 Google Borg 的大规模容器集群管理最佳实践之一，也是 Kubernetes 进行复杂应用编排的基础依赖之一；

所有“设计模式”的本质都是：解耦和重用。

# K8s应用编排与管理

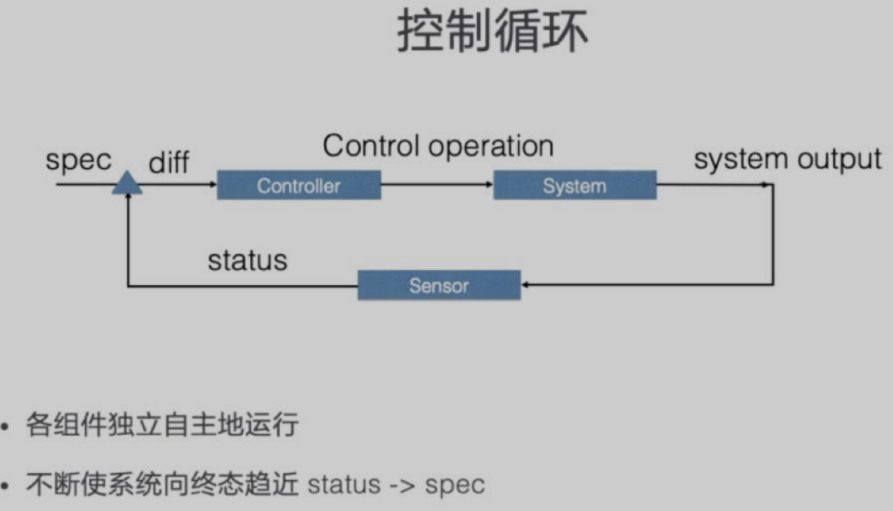
1. 资源元信息
2. Kubernates资源对象
3. Labels
4. Selector
5. Annotations
6. Ownereference
7. 操作命令

Kubectl apply -f \*.yaml

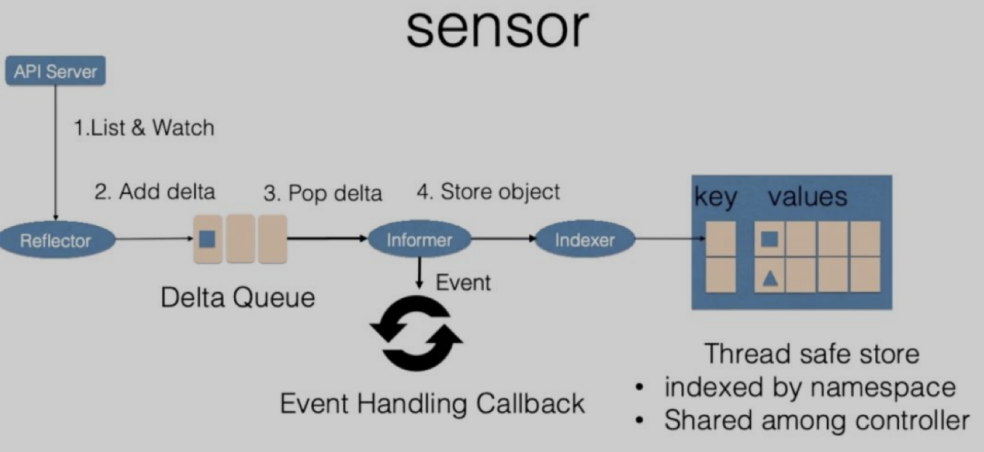
Kubectl [get|label|annotation] [pods|replicasets|deployment] [resname]

1. 控制器模式

控制型模式最核心的就是控制循环的概念。在控制循环中包括了控制器、被控制的系统，以及能够观测系统的传感器，三个逻辑组件。

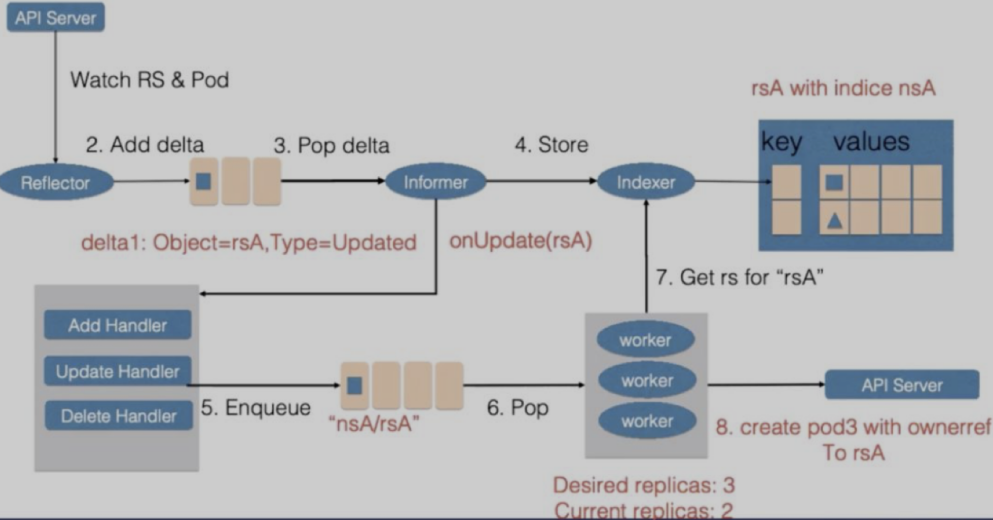


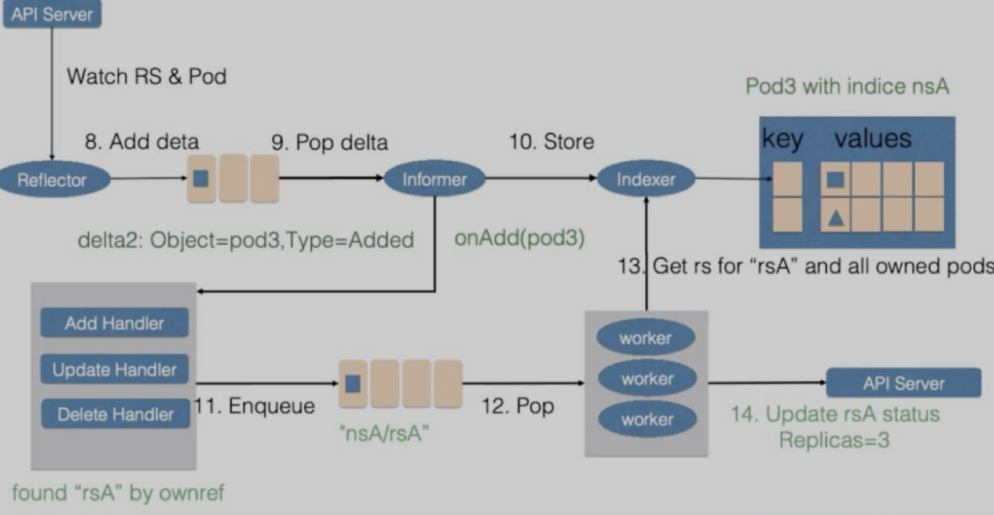
控制循环中逻辑的传感器主要由 Reflector、Informer、Indexer 三个组件构成。



ReplicaSet 是一个用来描述无状态应用的扩缩容行为的资源







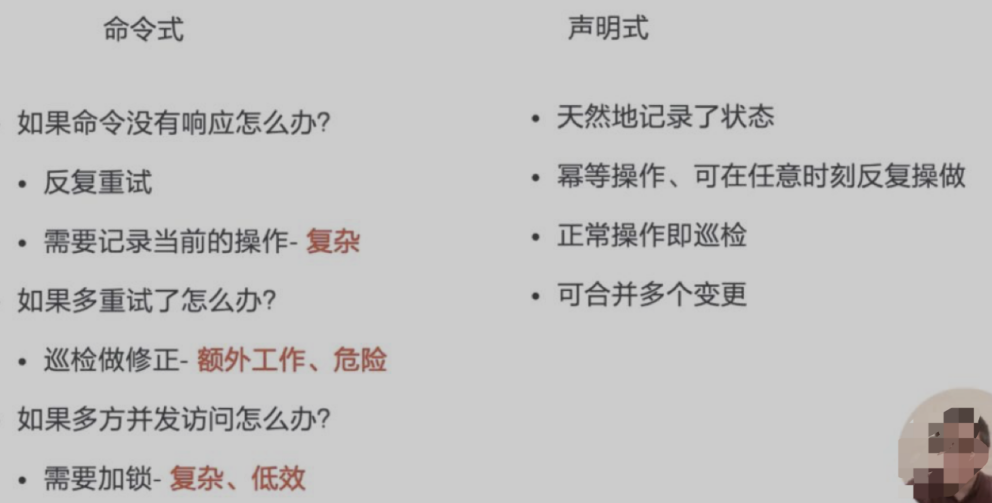
1. 控制器模式总结

1）Kubernetes 所采用的控制器模式，是由声明式 API 驱动的。确切来说，是基于对 Kubernetes 资源对象的修改来驱动的；

2）Kubernetes 资源之后，是关注该资源的控制器。这些控制器将异步的控制系统向设置的终态驱近；

3）这些控制器是自主运行的，使得系统的自动化和无人值守成为可能；

因为 Kubernetes 的控制器和资源都是可以自定义的，因此可以方便的扩展控制器模式。 4）特别是对于有状态应用，我们往往通过自定义资源和控制器的方式，来自动化运维操作。Operator





1. Deployment是什么？

管理部署发布的控制器

1. Deployment要解决的问题？

1）首先，如何保证集群内可用 Pod 的数量？也就是说我们应用 A 四个 Pod 如果出 现了一些宿主机故障，或者一些网络问题，如何能保证它可用的数量？

2）如何为所有 Pod 更新镜像版本？我们是否要某一个 Pod 去重建新版本的 Pod？

3）然后在更新过程中，如何保证服务的可用性？

4）以及更新过程中，如果发现了问题，如何快速回滚到上一个版本？



1. Spec字段解析

MinReadySeconds：Deployment 会根据 Pod ready 来看 Pod 是否可用，但是如果我们设置了 MinReadySeconds 之后，比如设置为 30 秒，那 Deployment 就一定会等到 Pod ready 超过 30 秒之后才认为 Pod 是 available 的。Pod available 的前提条件是 Pod ready，但是 ready 的 Pod 不一定是 available 的，它一定要超过 MinReadySeconds 之后，才会判断为 available；

revisionHistoryLimit：保留历史 revision，即保留历史 ReplicaSet 的数量，默认值为 10 个。这里可以设置为一个或两个，如果回滚可能性比较大的话，可以设置数量超过 10；

paused：paused 是标识，Deployment 只做数量维持，不做新的发布，这里在 Debug 场景可能会用到；

progressDeadlineSeconds：前面提到当 Deployment 处于扩容或者发布状态时，它的 condition 会处于一个 processing 的状态，processing 可以设置一个超时时间。如果超过超时时间还处于 processing，那么 controller 将认为这个 Pod 会进入 failed 的状态。

1. 升级策略字段解析

Deployment 在 RollingUpdate 中主要提供了两个策略，一个是 MaxUnavailable，另一个是 MaxSurge。这两个字段解析的意思，可以看下图中详细的 comment，或者简单解释一下：

MaxUnavailable：滚动过程中最多有多少个 Pod 不可用；

MaxSurge：滚动过程中最多存在多少个 Pod 超过预期 replicas 数量。

1. Deployment总结

1）Deployment 是 Kubernetes 中常见的一种 Workload，支持部署管理多版本的 Pod；

2）Deployment 管理多版本的方式，是针对每个版本的 template 创建一个 ReplicaSet，由 ReplicaSet 维护一定数量的 Pod 副本，而 Deployment 只需要关心不同版本的 ReplicaSet 里要指定多少数量的 Pod；

3）因此，Deployment 发布部署的根本原理，就是 Deployment 调整不同版本 ReplicaSet 里的终态副本数，以此来达到多版本 Pod 的升级和回滚。

1. Job

1）背景

我们如何保证 Pod 内进程正确的结束？

如何保证进程运行失败后重试？

如何管理多个任务，且任务之间有依赖关系？

如何并行地运行任务，并管理任务的队列大小？

2）功能

首先 Kubernetes 的 Job 是一个管理任务的控制器，它可以创建一个或多个 Pod 来指定 Pod 的数量，并可以监控它是否成功地运行或终止；

我们可以根据 Pod 的状态来给 Job 设置重置的方式及重试的次数；

我们还可以根据依赖关系，保证上一个任务运行完成之后再运行下一个任务；

同时还可以控制任务的并行度，根据并行度来确保 Pod 运行过程中的并行次数和总体完成大小

3）关键参数

第一个是 restartPolicy，在 Job 里面我们可以设置 Never、OnFailure、Always 这三种重试策略。在希望 Job 需要重新运行的时候，我们可以用 Never；希望在失败的时候再运行，再重试可以用 OnFailure；或者不论什么情况下都重新运行时 Alway；

另外，Job 在运行的时候不可能去无限地重试，所以我们需要一个参数来控制重试的次数。这个 backoffLimit 就是来保证一个 Job 到底能重试多少次。

1. Job状态

**AGE**的含义是指这个 Pod 从当前时间算起，减去它当时创建的时间。这个时长主要用来告诉你 Pod 的历史、Pod 距今创建了多长时间；

**DURATION** 主要来看我们 Job 里面的实际业务到底运行了多长时间，当我们的性能调优的时候，这个参数会非常的有用；

**COMPLETIONS**主要来看我们任务里面这个 Pod 一共有几个，然后它其中完成了多少个状态，会在这个字段里面做显示。

5）举个例子



1. CronJob

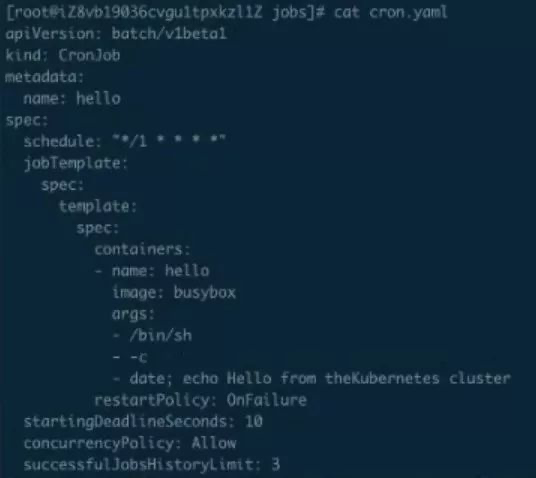
定时任务和 Job 相比会多几个不同的字段：

**schedule**：schedule 这个字段主要是设置时间格式，它的时间格式和 Linux 的 crontime 是一样的，所以直接根据 Linux 的 crontime 书写格式来书写就可以了。举个例子：\*/1 指每分钟去执行一下 Job，这个 Job 需要做的事情就是打印出大约时间，然后打印出“Hello from the kubernetes cluster” 这一句话；

**startingDeadlineSeconds**：即每次运行 Job 的时候，它最长可以等多长时间，有时这个 Job 可能运行很长时间也不会启动。所以这时，如果超过较长时间的话，CronJob 就会停止这个 Job；

**concurrencyPolicy**：就是说是否允许并行运行。所谓的并行运行就是，比如说我每分钟执行一次，但是这个 Job 可能运行的时间特别长，假如两分钟才能运行成功，也就是第二个 Job 要到时间需要去运行的时候，上一个 Job 还没完成。如果这个 policy 设置为 true 的话，那么不管你前面的 Job 是否运行完成，每分钟都会去执行；如果是 false，它就会等上一个 Job 运行完成之后才会运行下一个；

**JobsHistoryLimit**：这个就是每一次 CronJob 运行完之后，它都会遗留上一个 Job 的运行历史、查看时间。当然这个额不能是无限的，所以需要设置一下历史存留数，一般可以设置默认 10 个或 100 个都可以，这主要取决于每个人集群不同，然后根据每个人的集群数来确定这个时间。



1. DaemonSet
2. 背景

首先如果希望每个节点都运行同样一个 pod 怎么办？

如果新节点加入集群的时候，想要立刻感知到它，然后去部署一个 pod，帮助我们初始化一些东西，这个需求如何做？

如果有节点退出的时候，希望对应的 pod 会被删除掉，应该怎么操作？

如果 pod 状态异常的时候，我们需要及时地监控这个节点异常，然后做一些监控或者汇报的一些动作，那么这些东西运用什么控制器来做？

1. 功能

首先能保证集群内的每一个节点都运行一组相同的 pod；

同时还能根据节点的状态保证新加入的节点自动创建对应的 pod；

在移除节点的时候，能删除对应的 pod；

而且它会跟踪每个 pod 的状态，当这个 pod 出现异常、Crash 掉了，会及时地去 recovery 这个状态

1. 关键参数

DaemonSet 和 deployment 特别像，它也有两种更新策略：一个是 RollingUpdate，另一个是 OnDelete。

RollingUpdate 其实比较好理解，就是会一个一个的更新。先更新第一个 pod，然后老的 pod 被移除，通过健康检查之后再去见第二个 pod，这样对于业务上来说会比较平滑地升级，不会中断；

OnDelete 其实也是一个很好的更新策略，就是模板更新之后，pod 不会有任何变化，需要我们手动控制。我们去删除某一个节点对应的 pod，它就会重建，不删除的话它就不会重建，这样的话对于一些我们需要手动控制的特殊需求也会有特别好的作用。

# 配置管理

1. 背景

第一，比如说一些可变的配置。因为我们不可能把一些可变的配置写到镜像里面，当这个配置需要变化的时候，可能需要我们重新编译一次镜像，这个肯定是不能接受的；

第二就是一些敏感信息的存储和使用。比如说应用需要使用一些密码，或者用一些 token；

第三就是我们容器要访问集群自身。比如我要访问 kube-apiserver，那么本身就有一个身份认证的问题；

第四就是容器在节点上运行之后，它的资源需求；

第五个就是容器在节点上，它们是共享内核的，那么它的一个安全管控怎么办？

最后一点我们说一下容器启动之前的一个前置条件检验。比如说，一个容器启动之前，我可能要确认一下 DNS 服务是不是好用？又或者确认一下网络是不是联通的？那么这些其实就是一些前置的校验。

1. 实现方式

可变配置就用 ConfigMap；

敏感信息是用 Secret；

身份认证是用 ServiceAccount 这几个独立的资源来实现的；

资源配置是用 Resources；

安全管控是用 SecurityContext；

前置校验是用 InitContainers 这几个在 spec 里面加的字段，来实现的这些配置管理。

1. ConfigMap
2. 创建

Kubectl create configMap [NAME] | [DATA]

1. 使用

第一种是环境变量。环境变量的话通过 valueFrom，然后 ConfigMapKeyRef 这个字段，下面的 name 是指定 ConfigMap 名，key 是 ConfigMap.data 里面的 key。

第二个是命令行参数。

最后一个是通过 volume 挂载的方式直接挂到容器的某一个目录下面去。

1. 注意事项

ConfigMap 文件的大小。虽然说 ConfigMap 文件没有大小限制，但是在 ETCD 里面，数据的写入是有大小限制的，现在是限制在 1MB 以内；

第二个注意点是 pod 引入 ConfigMap 的时候，必须是相同的 Namespace 中的 ConfigMap

创建 pod 前，必须先把要引用的 ConfigMap 创建好

把 ConfigMap 里面所有的信息导入成环境变量时，如果 ConfigMap 里有些 key 是无效的，比如 key 的名字里面带有数字，那么这个环境变量其实是不会注入容器的，它会被忽略。

只有通过 K8s api 创建的 pod 才能使用 ConfigMap，比如说通过用命令行 kubectl 来创建的 pod

1. Secret
2. 类型

常用的四种类型：

第一种是 Opaque，它是普通的 Secret 文件；

第二种是 service-account-token，是用于 service-account 身份认证用的 Secret；

第三种是 dockerconfigjson，这是拉取私有仓库镜像的用的一种 Secret；

第四种是 bootstrap.token，是用于节点接入集群校验用的 Secret。

1. 创建

Kubectl create secret generic [NAME] [DATA] [TYPE]

举个例子

kubectl create secret generic myregistrykey --from-file=/root/.docker/config.json --type=kubernates.io/dockerconfigjson

1. 使用

它主要是被 pod 来使用，一般是通过 volume 形式挂载到容器里指定的目录，然后容器里的业务进程再到目录下读取 Secret 来进行使用。另外在需要访问私有镜像仓库时，也是通过引用 Secret 来实现。

1. 注意事项
2. ServiceAccount

ServiceAccount 首先是用于解决 pod 在集群里面的身份认证问题，身份认证信息是存在于 Secret 里面



1. Resources

容器的一个资源配置管理。目前内部支持类型有三种：CPU、内存，以及临时存储。



**Pod 服务质量 (QoS) 配置**

根据 CPU 对容器内存资源的需求，我们对 pod 的服务质量进行一个分类，分别是 Guaranteed、Burstable 和 BestEffort。

Guaranteed ：pod 里面每个容器都必须有内存和 CPU 的 request 以及 limit 的一个声明，且 request 和 limit 必须是一样的，这就是 Guaranteed；

Burstable：Burstable 至少有一个容器存在内存和 CPU 的一个 request；

BestEffort：只要不是 Guaranteed 和 Burstable，那就是 BestEffort。

那么这个服务质量是什么样的呢？资源配置好后，当这个节点上 pod 容器运行，比如说节点上 memory 配额资源不足，kubelet会把一些低优先级的，或者说服务质量要求不高的（如：BestEffort、Burstable）pod 驱逐掉。它们是按照先去除 BestEffort，再去除 Burstable 的一个顺序来驱逐 pod 的。

1. SecurityContext

SecurityContext 主要是用于限制容器的一个行为，它能保证系统和其他容器的安全。这一块的能力不是 Kubernetes 或者容器 runtime 本身的能力，而是 Kubernetes 和 runtime 通过用户的配置，最后下传到内核里，再通过内核的机制让 SecurityContext 来生效。所以这里介绍的内容，会比较简单或者说比较抽象一点。

SecurityContext 主要分为三个级别：

1）第一个是容器级别，仅对容器生效；

2）第二个是 pod 级别，对 pod 里所有容器生效；

3）第三个是集群级别，就是 PSP，对集群内所有 pod 生效。

1. InitContainer

 InitContainer 和普通 container 的区别：

1）InitContainer 首先会比普通 container 先启动，并且直到所有的 InitContainer 执行成功后，普通 container 才会被启动；

2）InitContainer 之间是按定义的次序去启动执行的，执行成功一个之后再执行第二个，而普通的 container 是并发启动的；

3）InitContainer 执行成功后就结束退出，而普通容器可能会一直在执行。它可能是一个 longtime 的，或者说失败了会重启，这个也是 InitContainer 和普通 container 不同的地方。

InitContainer 的一个用途：为普通 container 服务，比如说它可以为普通 container 启动之前做一个初始化（配置文件，前置校验等）

# 存储与持久化

1. Pod Volumes

使用场景：

场景一：如果 pod 中的某一个容器在运行时异常退出，被 kubelet 重新拉起之后，如何保证之前容器产生的重要数据没有丢失？

场景二：如果同一个 pod 中的多个容器想要共享数据，应该如何去做？

以上两个场景，其实都可以借助 Volumes 来很好地解决，Pod Volumes 的常见类型：

1）本地存储，常用的有 emptydir/hostpath；

2）网络存储：网络存储当前的实现方式有两种，一种是 in-tree，它的实现代码是放在 K8s 代码仓库中的，随着 K8s 对存储类型支持的增多，这种方式会给 K8s 本身的维护和发展带来很大的负担；而第二种实现方式是 out-of-tree，它的实现其实是给 K8s 本身解耦的，通过抽象接口将不同存储的 driver 实现从 K8s 代码仓库中剥离，因此 out-of-tree 是后面社区主推的一种实现网络存储插件的方式；

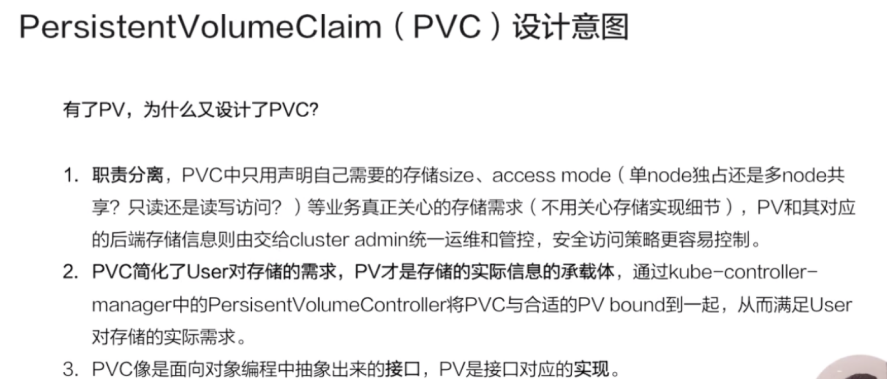
3）Projected Volumes：它其实是将一些配置信息，如 secret/configmap 用卷的形式挂载在容器中，让容器中的程序可以通过 POSIX 接口来访问配置数据；

4）PV 与 PVC

1. PV（Persistent Volumes）



1. PVC（Persistent Volume Claim）



1. Static PV provisioning

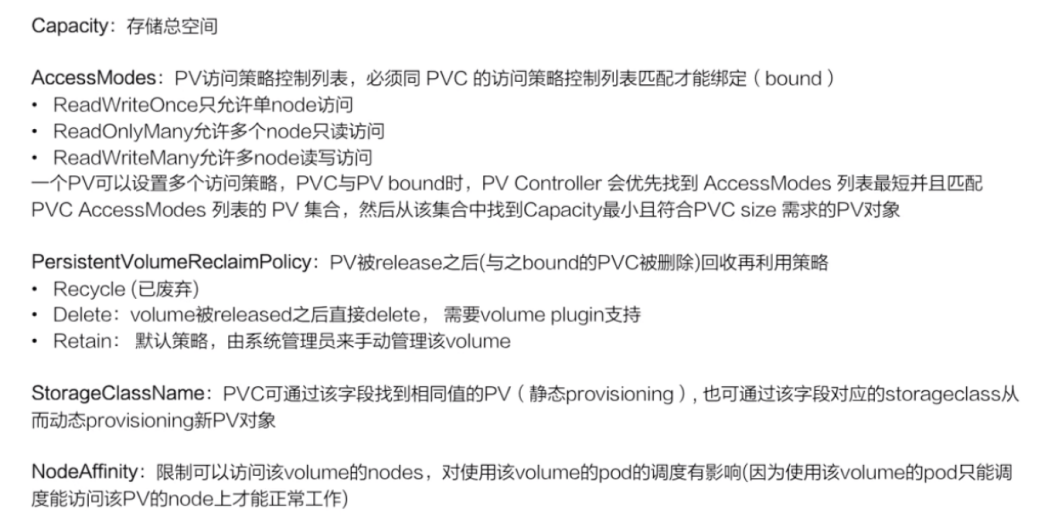
由集群管理员事先去规划这个集群中的用户会怎样使用存储，它会先预分配一些存储，也就是预先创建一些 PV；然后用户在提交自己的存储需求（也就是 PVC）的时候，K8s 内部相关组件会帮助它把 PVC 和 PV 做绑定；之后用户再通过 pod 去使用存储的时候，就可以通过 PVC 找到相应的 PV，它就可以使用了。

1. Dynamic PV provisioning

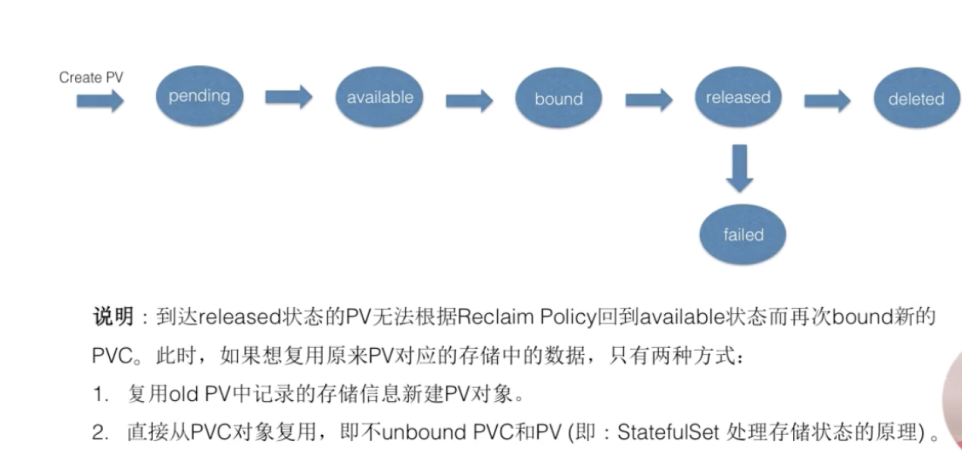
集群管理员不预分配 PV，他写了一个模板文件，这个模板文件是用来表示创建某一类型存储（块存储，文件存储等）所需的一些参数，这些参数是用户不关心的，给存储本身实现有关的参数。用户只需要提交自身的存储需求，也就是 PVC 文件，并在 PVC 中指定使用的存储模板（StorageClass）。

K8s 集群中的管控组件，会结合 PVC 和 StorageClass 的信息动态，生成用户所需要的存储（PV），将 PVC 和 PV 进行绑定后，pod 就可以使用 PV 了。通过 StorageClass 配置生成存储所需要的存储模板，再结合用户的需求动态创建 PV 对象，做到按需分配，在没有增加用户使用难度的同时也解放了集群管理员的运维工作

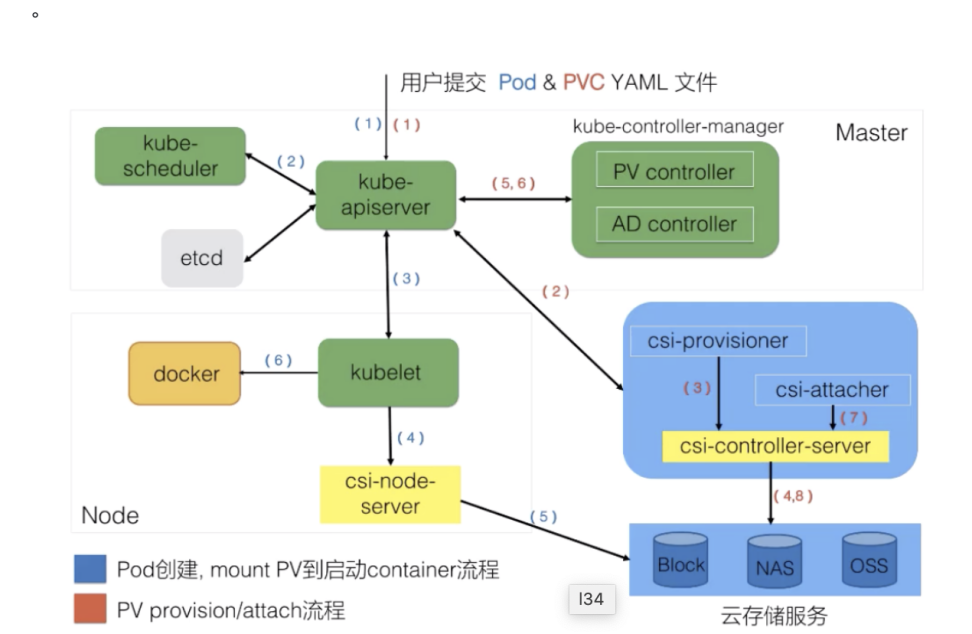
1. Spec重要字段解析

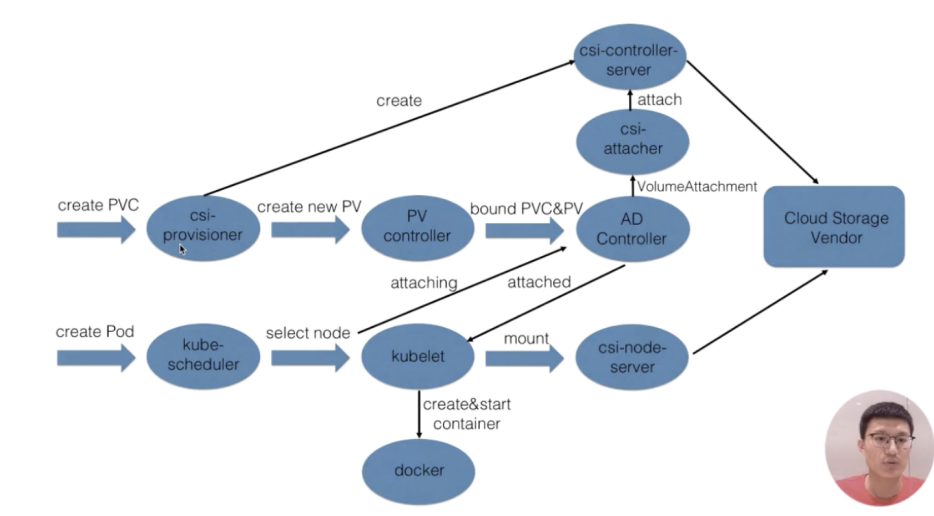


1. PV状态流转



1. PV和PVC架构设计





# 监控与日志

1. 背景：如何保证应用的健康与稳定
2. 提高应用的可观测性
   * 1. 首先是应用的健康状态上面，可以实时地进行观测；
     2. 第二个是可以获取应用的资源使用情况；
     3. 第三个是可以拿到应用的实时日志，进行问题的诊断与分析。
3. 提高应用的可恢复能力
4. Readiness probe就绪探针
5. Liveness probe存活探针
6. 探测方式

Liveness 指针和 Readiness 指针支持三种不同的探测方式：

第一种是 httpGet。它是通过发送 http Get 请求来进行判断的，当返回码是 200-399 之间的状态码时，标识这个应用是健康的；

第二种探测方式是 Exec。它是通过执行容器中的一个命令来判断当前的服务是否是正常的，当命令行的返回结果是 0，则标识容器是健康的；

第三种探测方式是 tcpSocket 。它是通过探测容器的 IP 和 Port 进行 TCP 健康检查，如果这个 TCP 的链接能够正常被建立，那么标识当前这个容器是健康的。

1. 探测结果

第一种是 success，当状态是 success 的时候，表示 container 通过了健康检查，也就是 Liveness probe 或 Readiness probe 是正常的一个状态；

第二种是 Failure，Failure 表示的是这个 container 没有通过健康检查，如果没有通过健康检查的话，那么此时就会进行相应的一个处理，那在 Readiness 处理的一个方式就是通过 service。service 层将没有通过 Readiness 的 pod 进行摘除，而 Liveness 就是将这个 pod 进行重新拉起，或者是删除；

第三种状态是 Unknown，Unknown 是表示说当前的执行的机制没有进行完整的一个执行，可能是因为类似像超时或者像一些脚本没有及时返回，那么此时 Readiness-probe 或 Liveness-probe 会不做任何的一个操作，会等待下一次的机制来进行检验。

1. 总结



1. 应用故障排查
   1. Pod停留在Pending

pending 表示调度器没有进行介入。此时可以通过 kubectl describe pod 来查看相应的事件，可能原因：端口占用、node selector限制

* 1. Pod停留在waiting

镜像没有正常拉取，可能原因：私有镜像没有配置Pod Secret或者镜像地址不存在

* 1. Pod不断被拉起并且可以看到crashing

可以看到类似像 backoff 。这个通常表示说 pod 已经被调度完成了，但是启动失败，那这个时候通常要关注的应该是这个应用自身的一个状态，并不是说配置是否正确、权限是否正确，此时需要查看的应该是 pod 的具体日志

* 1. Pod处于running但是没有正常工作

比较常见的一个点就可能是由于一些非常细碎的配置，类似像有一些字段可能拼写错误，造成了 yaml 下发下去了，但是有一段没有正常生效，从而使得这个 pod 处在 running 的状态没有对外服务。  
此时可以通过 apply-validate-f pod.yaml 的方式来进行判断当前 yaml 是否是正常的，如果 yaml 没有问题，那么接下来可能要诊断配置的端口是否是正常的，以及 Liveness 或 Readiness 是否已经配置正确。

* 1. Service无法正常工作

因为 service 和底层的 pod 之间的关联关系是通过 selector 的方式来匹配的，也就是说 pod 上面配置了一些 label，然后 service 通过 match label 的方式和这个 pod 进行相互关联。如果这个 label 配置的有问题，可能会造成这个 service 无法找到后面的 endpoint，从而造成相应的 service 没有办法对外提供服务，那如果 service 出现异常的时候，第一个要看的是这个 service 后面是不是有一个真正的 endpoint，其次来看这个 endpoint 是否可以对外提供正常的服务。

1. 应用远程调试
   1. Pod远程调试

Kubectl exec -it pod-name -c container-name /bin/bash

* 1. Service远程调试

Telepresence 端口映射

* 1. Kubectl-debug

# 网络

# 服务发现与负载均衡

# Etcd

# 调度与资源管理