K8S 优雅升级系列（上） | 如何让应用和容器优雅下线

> 在生产环境中，如何保证在服务升级的时候，不影响用户的体验，这个是一个非常重要的问题。如果在我们升级服务的时候，会造成一段时间内的服务不可用，这就是不够优雅的。
>
> 那什么是优雅的呢？主要就是指在服务升级的时候，**不中断整个服务，让用户无感知，进而不会影响用户的体验**，这就是优雅的。

> ***微服务发布方式***
>
> 说到优雅升级，就不得不了解一下常见的微服务的发布方式，选择不同的发布方式，就会达到不同的效果，在产线实践中最为常用的发布方式有：
>
> - 蓝绿发布
> - 灰度发布
> - 滚动发布

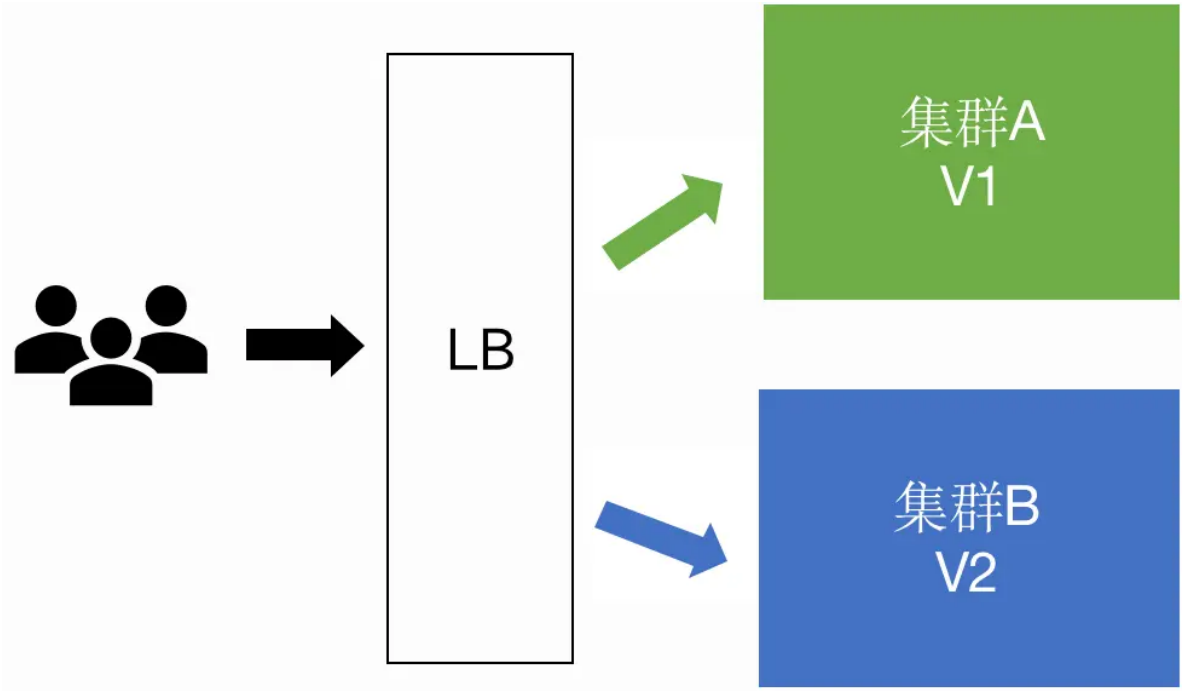
- **概念**

蓝绿部署（Blue Green Deployment）是一种可以保证系统在不间断提供服务的情况下上线的部署方式，主要的流程为：**不停老版本，部署新版本然后进行测试。**确认 OK 后将流量切到新版本，然后老版本同时也升级到新版本。

- **步骤**

其大致步骤为：

1. 首先把 B 组从负载均衡中摘除，进行新版本的部署。A 组仍然继续提供服务；
2. 当 B 组升级完毕，负载均衡重新接入 B 组，再把 A 组从负载列表中摘除，进行新版本的部署，B 组重新提供服务；
3. A 组也升级完成，负载均衡重新接入 A 组，此时，AB 组版本都已经升级完成，并且都对外提供服务。



- **特点**

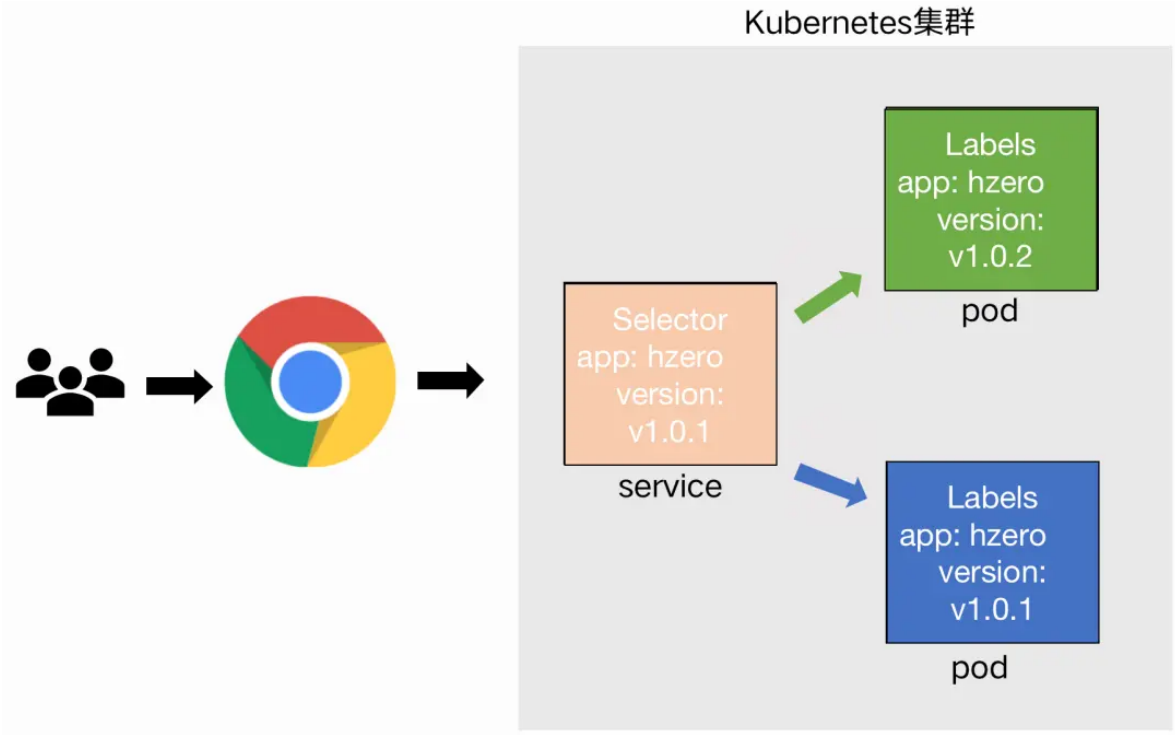
1. 通过 LB 来实现流量控制，比如 Nginx;
2. 如果出问题，影响范围较大；
3. 发布策略简单；
4. 用户无感知，平滑过渡；
5. 升级 / 回滚速度快。

- **缺点**

1. 需要准备正常业务使用资源的两倍以上服务器，防止升级期间单组无法承载业务突发；
2. 短时间内浪费一定资源成本。

- **Kubernetes 中的蓝绿发布**

在 Kubernetes 中，可以使用 Service 中的 Selector 标签结合 Pod 中的 Labels 标签进行流量的切换处理，从而达到蓝绿部署的效果。



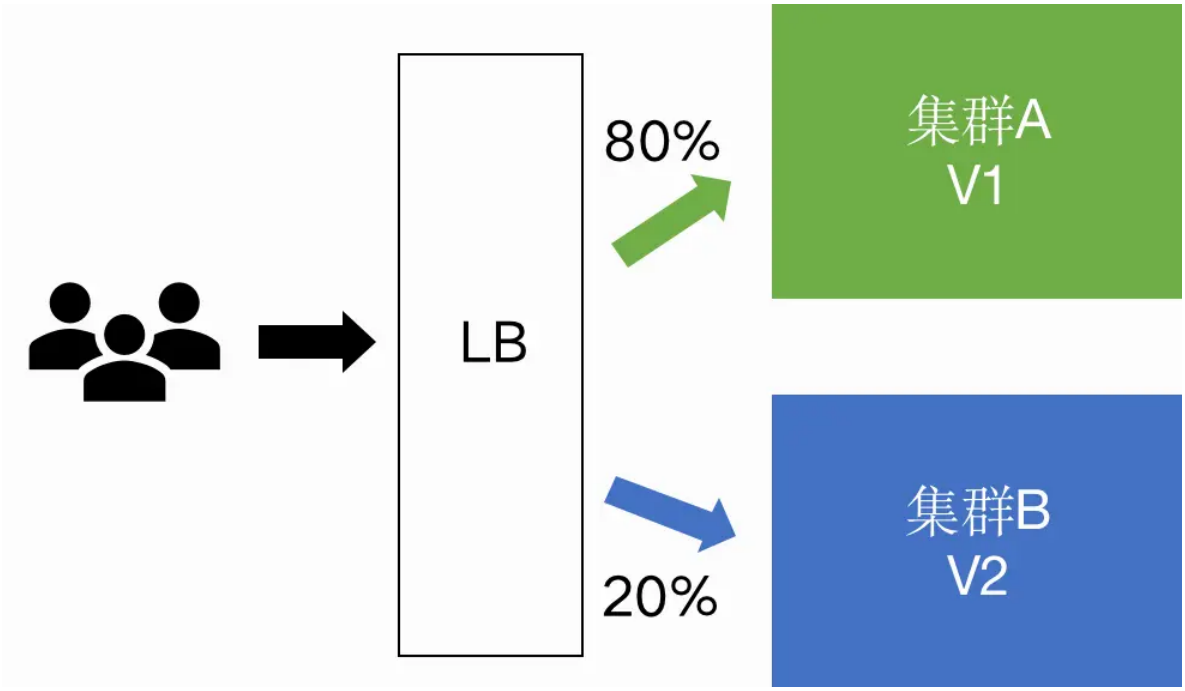
- **概念**

灰度发布（Canary Deployment）又叫金丝雀发布，以前，旷工在下矿洞是面临的一个重要危险是矿井中的毒气，他们想到一个办法来辨别矿井中是否有毒气，矿工们随身携带一只金丝雀下矿井，金丝雀对毒气的抵抗能力比人类要弱，在毒气环境下会先挂掉起到预警的作用。

- **步骤**

其大致步骤为：

1. 准备好部署各个阶段的工件，包括：构建工件，测试脚本，配置文件和部署清单文件；
2. 从 LB 摘掉灰度服务器；
3. 升级 “金丝雀” 应用（切断原有流量并进行部署）；
4. 对灰度服务进行自动化测试或者少量用户流量到新版本进行测试；
5. 如果灰度服务器测试成功，升级剩余服务器（否则就回滚）。



- **特点**

1. 保证整体系统稳定性，在初始灰度的时候就可以发现、调整问题，影响范围可控；
2. 新功能逐步评估性能，稳定性和健康状况，如果出问题影响范围很小，相对用户体验也少；
3. 用户无感知，平滑过渡。

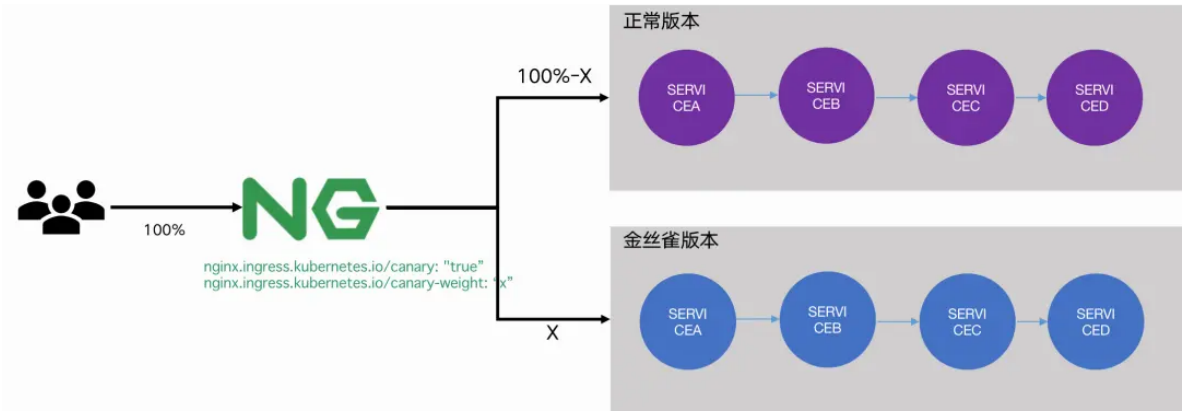
- **缺点**

1. 自动化要求高，如果发布自动化程度不够，发布期间可引发服务中断。

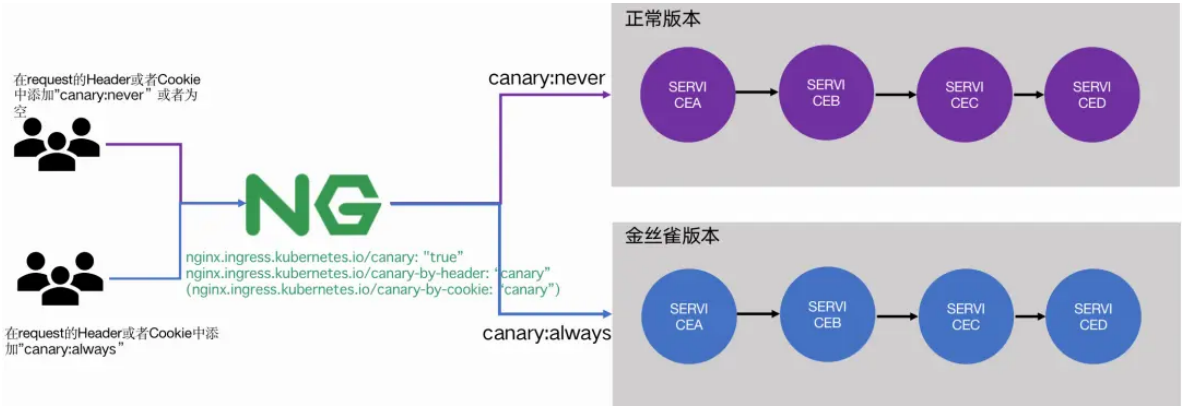
- **Kubernetes 中的灰度发布**

在 Kubernetes 中，有很多的方案可以实现灰度发布，比如 Nginx Ingress Controller、Kong Ingress Controller、Istio 等，这里案例展示的是最常用的 Nginx Ingress Controller 的 Canary 实现。

基于权重的金丝雀发布



基于请求的金丝雀发布（A/B TEST）

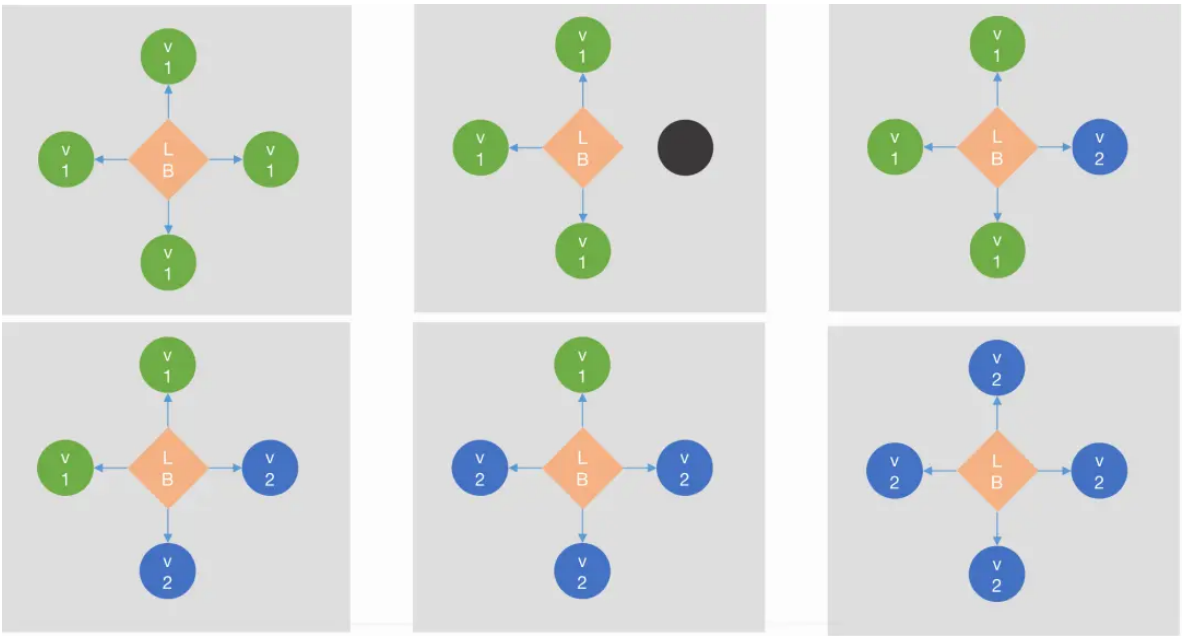


- **概念**

滚动发布（Rolling Update），一种高级的发布策略，发布过程中，**应用不中断，用户体验平滑，也是 Kubernetes 应用更新默认的发布方式。**

- **步骤**

1. 先升级 1 个副本，主要做部署验证；
2. 每次升级副本，自动从 LB 上摘掉，升级成功后自动加入集群；
3. 事先需要有自动更新策略，分为若干次，每次数量 / 百分比可配置；
4. 回滚是发布的逆过程，先从 LB 摘掉新版本，再升级老版本，这个过程一般时间比较长；
5. 自动化要求高。



- **特点**

1. 应用不中断，用户体验平滑；
2. 节约资源。

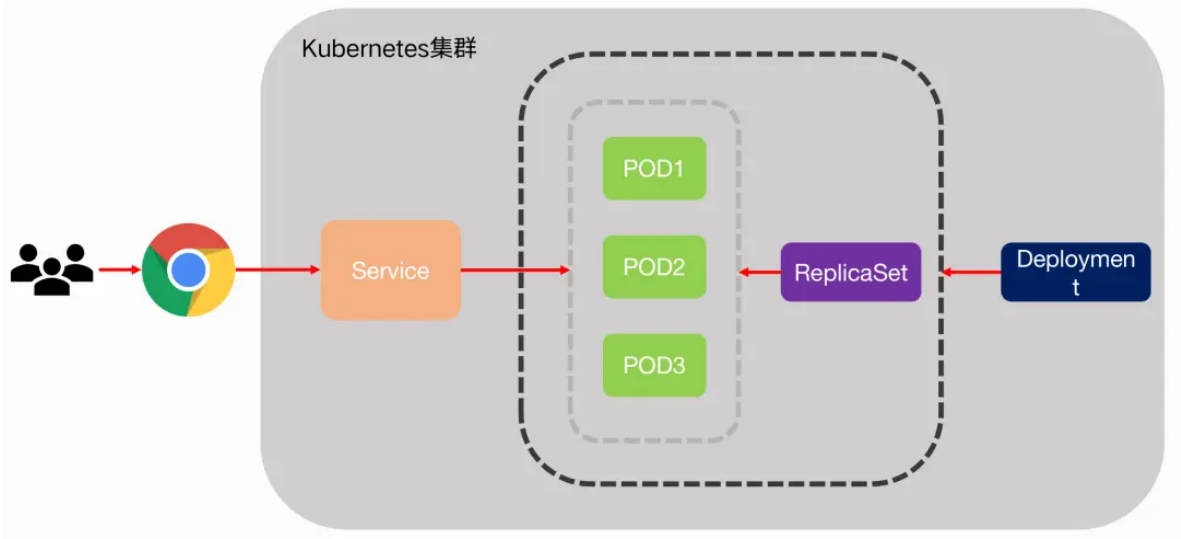
- **缺点**

1. 没有一个确定 OK 的环境。使用蓝绿部署，我们能够清晰地知道老版本是 OK 的，而使用滚动发布，我们无法确定；
2. 修改了现有的环境；
3. 部署时间慢，取决于每阶段更新时间；
4. 发布策略较复杂；

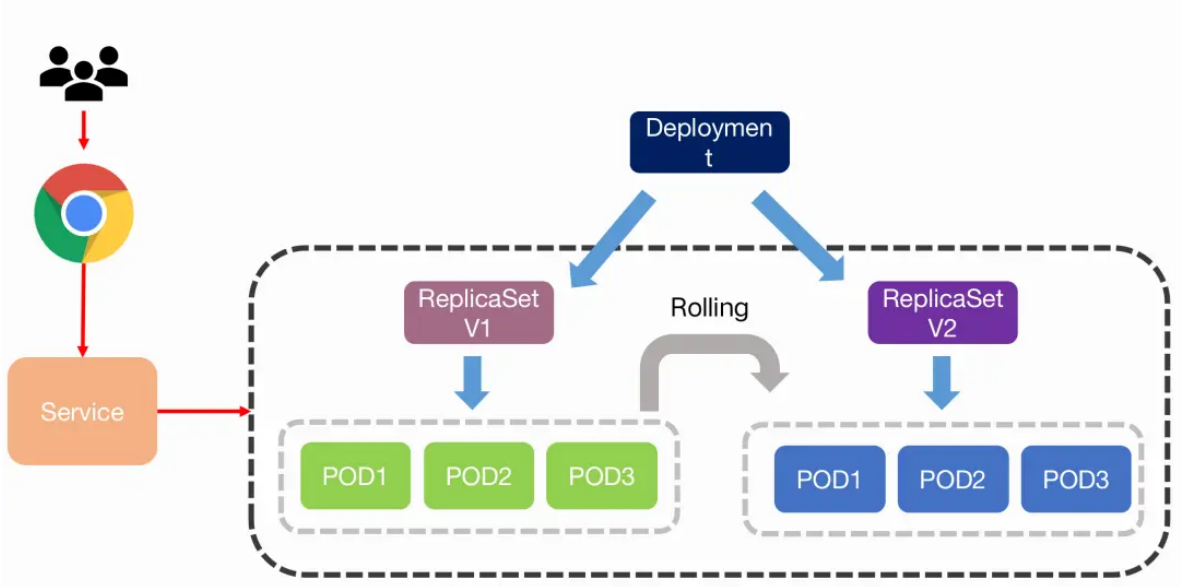
5. 如果需要回滚，很困难。举个例子，在某一次发布中，我们需要更新 100 个实例，每次更新 10 个实例，每次部署需要 5 分钟。当滚动发布到第 80 个实例时，发现了问题，需要回滚，这个时候就会很难回滚；
6. 有的时候，我们还可能对系统进行动态伸缩，如果部署期间，系统自动扩容 / 缩容了，我们还需判断到底哪个节点使用的是哪个代码。尽管有一些自动化的运维工具，但是依然令人心惊胆战。

- **Kubernetes 中的滚动发布**

滚动发布虽然发布策略复杂，回滚难度高，但是却是 K8S 默认的服务更新方式，K8S 中，Deployment 资源本质上是使用 RS+Rollout Update 组成的。



这样完全将服务版本与滚动更新机制结合，自动完成滚动更新的一系列操作，增强用户体验、节约资源、易于回滚。Kubernetes 中的滚动更新机制如下：



> ***如何优雅下线应用***
>
> 在生产环境中，要保证服务不中断，服务的上下线是不可避免的，我们希望能够优雅地下线微服务，在此总结一下 Spring Boot 中的应用的几种下线方式。

- **Kill java 进程**

该方式借助的是 Spring Boot 应用的 Shutdown hook，应用本身的下线也是优雅的，但如果你的服务发现组件使用的是 Eureka，那么默认最长会有 90 秒的延迟，其他应用才会感知到该服务下线，这意味着：该实例下线后的 90 秒内，其他服务仍然可能调用到这个已下线的实例。

因此，**该方式是不够优雅的。**

- **/shutdown 接口**

Spring Boot 提供了 /shutdown 端点，可以借助它实现优雅停机。

使用方式：在想下线应用的 Application.yml 中添加如下配置，从而启用并暴露 /shutdown 端点：

```
management:  endpoint:    shutdown:      enabled: true  endpoints:    web:
exposure:        include: shutdown
```

发送 POST 请求到 /shutdown 端点 Curl -X Http://ip:port/actuator/shutdown

该方式本质和方式一样也是借助 Spring Boot 应用的 Shutdown Hook 去实现的，**所以不建议使用。**

- **/pause 接口**

Spring Boot 应用提供了 /pause 端点，利用该端点可实现优雅下线。

使用方式：在想下线应用的 Application.yml 中添加如下配置，从而启用并暴露 /pause 端点：

```
management:  endpoint:    # 启用pause端点    pause:      enabled: true    # 启用
restart端点    restart:      enabled: true  endpoints:    web:      exposure:
   include: pause,restart
```

发送 POST 请求到 /actuator/pause 端点 Curl -X http://ip:port/actuator/pause

该应用在 Eureka Server 上的状已被标记为 DOWN，但是**应用本身其实依然是可以正常对外服务的。**

- **/service-registry 接口**

在想下线应用的 Application.yml 中添加配置，从而暴露 /service-registry 端点。

```
management:  endpoint    web:      exposure:        include: service-registry
```

发送 POST 请求到 /actuator/service-registry 端点：

```
curl -X "POST" "http://localhost:8000/actuator/service-registry?status=DOWN" \
-H "Content-Type: application/vnd.spring-boot.actuator.v2+json;charset=UTF-8"
```

这个接口会将服务的 Eureka 的状态标记为 Down，等服务流量变为 0，可以进行删除或者升级操作，算是**比较优雅**的一种方式。

- **EurekaAutoServiceRegistration**

除了上述的下线方式之外，还有一种利用 EurekaAutoServiceRegistration 对象达到**优雅下线**的目标。

执行 EurekaAutoServiceRegistration.start() 方法时，当前服务向 Eureka 注册中心注册服务；

执行 EurekaAutoServiceRegistration.stop () 方法时，当前服务会向 Eureka 注册中心进行反注册，注册中心收到请求后，会将此服务从注册列表中删除。

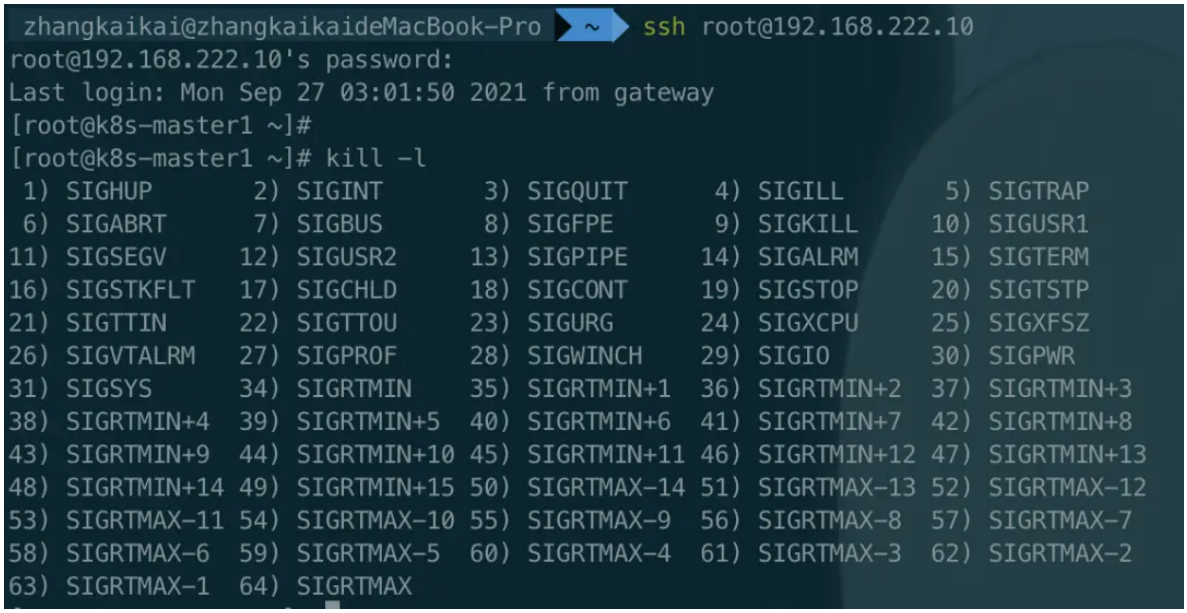
```
@RestController@RequestMapping(value = "/graceful/registry-service")public class GracefulOffline {    @Autowired    private EurekaAutoServiceRegistration eurekaAutoServiceRegistration;    @RequestMapping("/online")    public String online() {        this.eurekaAutoServiceRegistration.start();        return "execute online method, online success.";    }    @RequestMapping("/offline") public String offline() {        this.eurekaAutoServiceRegistration.stop();    return "execute offline method, offline success.";    }}
```

可以根据此 API 去自定义接口去进行服务的下线

> ### *如何优雅关闭容器*
>
> 服务在容器里正常跑着，前面讲了服务的优雅下线方式，下来就该容器的优雅关闭内容了，要不然，光服务进行优雅关闭了，容器一整个直接被 Kill 了，也是没有用的。

- **信号**

提到容器关闭，就不得不了解一个基础的东西，就是信号，信号是事件发生时对进程的通知机制，有时也称之为软件中断，信号有不同的类型，可以通过 Kill -l 获取信号名称：



- **常用的信号**

1. 9) SIGKILL 此信号为 “必杀（Sure Kill）” 信号，处理器程序无法将其阻塞、忽略或者捕获，故而 “一击必杀”，总能终止程序；
2. 15) SIGTERM 这是用来终止进程的标准信号，也是 Kill 、 Killall 、 Pkill 命令所发送的默认信号。精心设计的应用程序应当为 SIGTERM 信号设置处理器程序，以便其能够预先清除临时文件和释放其它资源，从而全身而退。因此，总是应该先尝试使用 SIGTERM 信号来终止进程，而把 SIGKILL 作为最后手段，去对付那些不响应 SIGTERM 信号的失控进程。

- **Dockerfile 中的 ENTRYPOINT 和 CMD 指令**

接着说 Dockerfile 中的 ENTRYPOINT 和 CMD 指令，它们的主要功能是指定容器启动时执行的程序

CMD 有三种格式：

- CMD ["Executable","Param1","Param2"] （Exec 格式，推荐使用这种格式）

- CMD ["Param1","Param2"] （作为 ENTRYPOINT 指令参数）
- CMD Command Param1 Param2 （Shell 格式，默认 /bin/sh -c ）

ENTRYPOINT 有两种格式：

- ENTRYPOINT ["Executable", "Param1", "Param2"] （Exec 格式，推荐优先使用这种格式）
- ENTRYPOINT Command Param1 Param2 （Shell 格式）

Docker Stop 停掉容器的时候，默认会发送一个 SIGTERM 的信号，默认 10s 后容器没有停止的话，就 SIGKILL 强制停止容器。通过 -t 选项可以设置等待时间。

在停止容器的时候系统底层默认会向主进程发送 SIGTERM 信号，而对剩余子进程发送 SIGKILL 信号。系统这样做的大概原因是因为大家在设计主进程脚本的时候都不会进行信号的捕获和传递，这会导致容器关闭时，多个子进程无法被正常终止，所以系统使用 SIGKILL 这个不可屏蔽信号，而是为了能够在没有任何前提条件的情况下，能够把容器中所有的进程关掉。

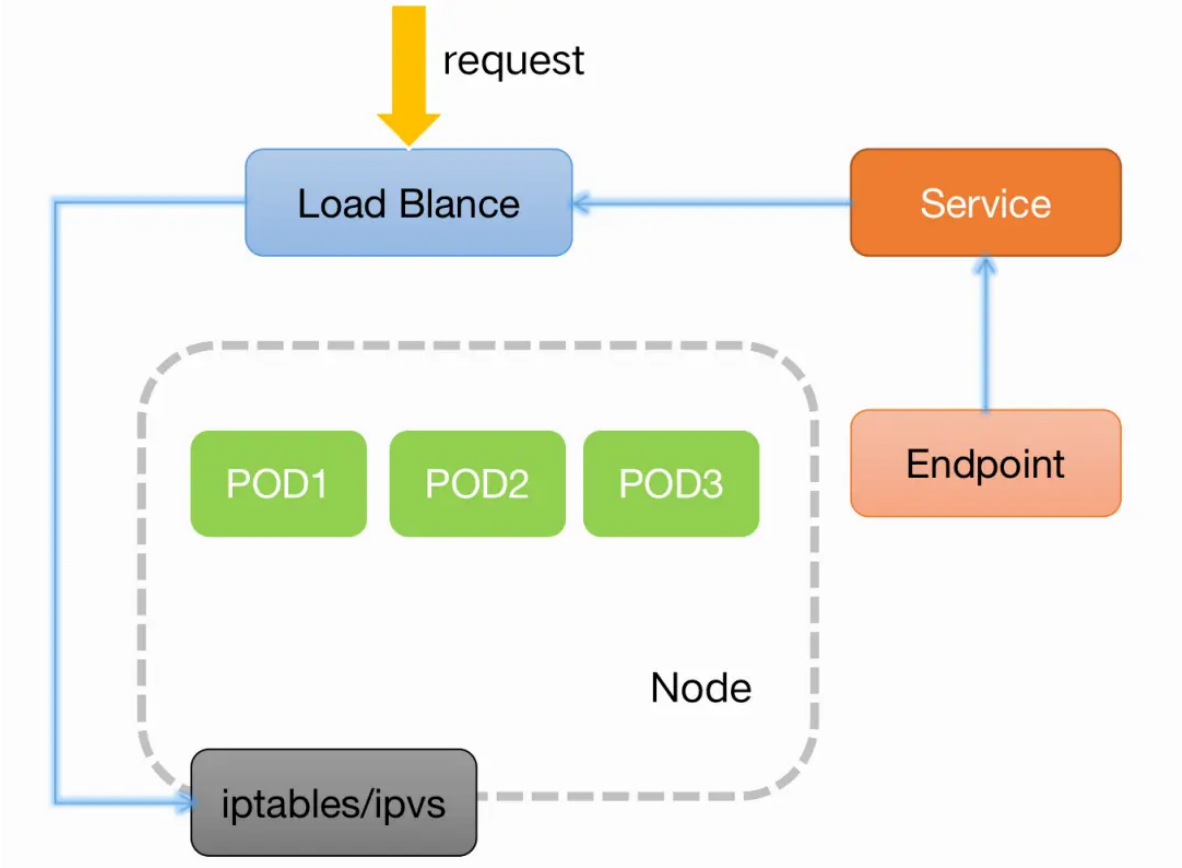
不管你 Dockerfile 用其中哪个指令，两个指令都推荐使用 Exec 格式，而不是 Shell 格式。原因就是因为使用 Shell 格式之后，程序会以 /bin/sh -c 的子命令启动，Shell 会作为主进程，并且 Shell 格式下不会传递任何信号给程序。这也就导致，在 Docker Stop 容器的时候，以这种格式运行的程序捕捉不到发送的信号，也就谈不上优雅的关闭了。

K8S 优雅升级系列（中） | 如何 “优雅” 滚动发布？看这篇就够了

> 什么是优雅升级？
>
> 首先，我们认为发布过程中如果遇到以下的问题都是 “不优雅” 的：
>
> - 发布过程中，出现正在执行的请求被中断；
> - 下游服务节点已经下线，上游依然继续调用已经下线的节点导致请求报错，进而导致业务异常；
> - 发布过程造成数据不一致，需要对脏数据进行修复。
>
> 那么，反之，优雅就是一种避免上述情况发生的手段，就是在服务升级的时候，**不中断整个服务，让用户无感知，不影响用户体验。**

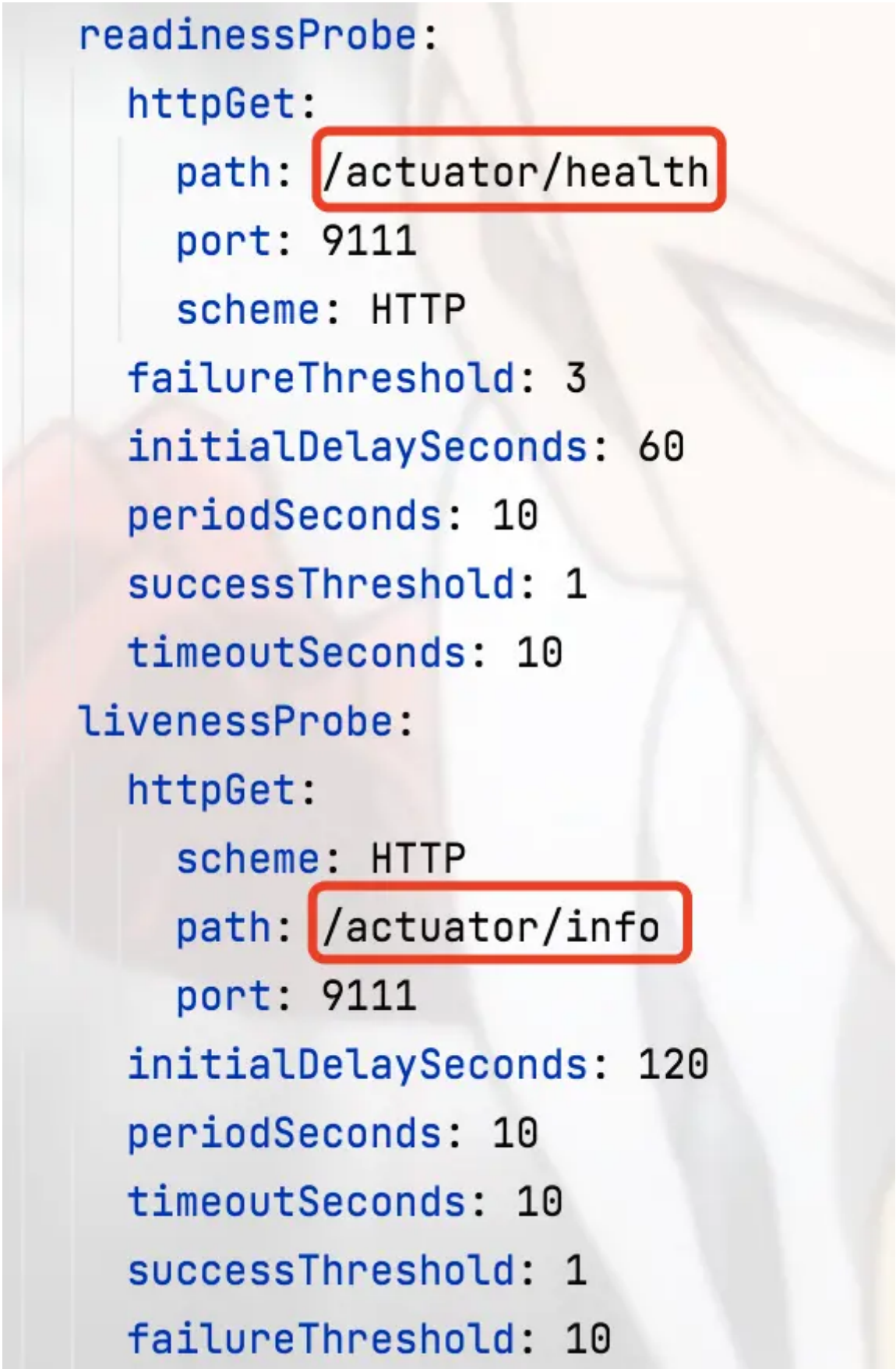
前面一篇讲了微服务应用发布方式、应用优雅下线以及容器优雅关闭的相关内容，接下来就分析下 K8S 滚动更新中，在什么样的情况下服务会中断以及相关的解决方案。

> ## 分析
>
> 前面在应用发布章节描述了 K8S 滚动发布的原理，Deployment 滚动更新时会先创建新 Pod，等待新 Pod Running 后再删除旧 Pod。
>
> 在 K8S 的网络中，Service 是借助于 Endpoint 资源来跟踪与其相关联的后端服务，Service 会根据 Selector 直接创建同名的 Endpoint 对象，Endpoint 对象会根据就绪状态把同名 Service 对象标签选择器筛选出的后端端点的 IP 地址分别保存在 Subsets.addresses 字段和 Subsets.notReadyAddresses 字段中，它通过 API Server 持续、动态跟踪每个端点的状态变动，并即时反映到端点 IP 所属的字段。

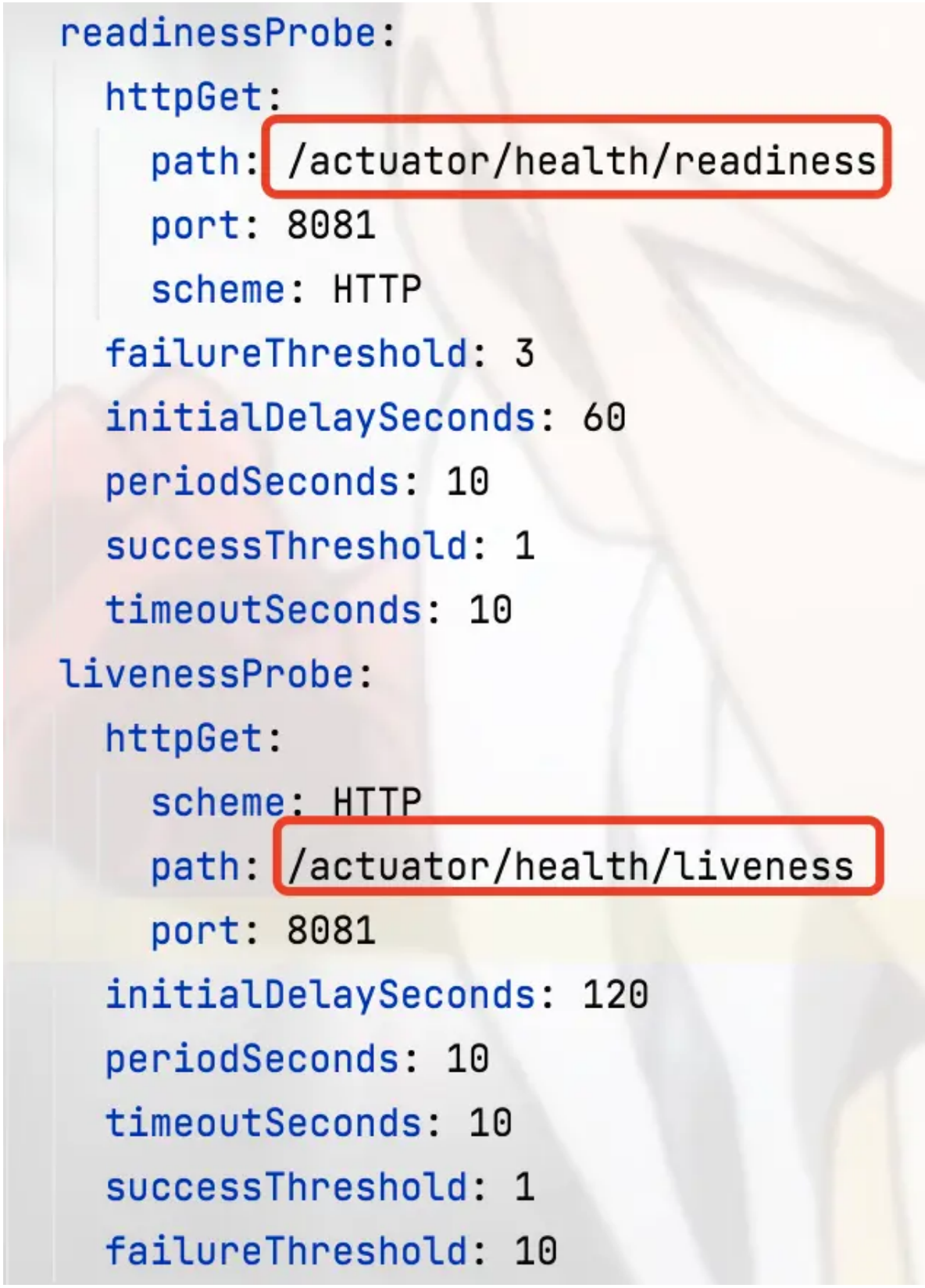


在 Deployment 对象对 POD 进行滚动更新的时候，根据 Endpoint 的机制，在新建 POD 以及删除 POD 的时候，会发生 Endpoint 的实时更新，大多数的服务的中断，就在这两部分 Endpoint 列表变更的时候发生。

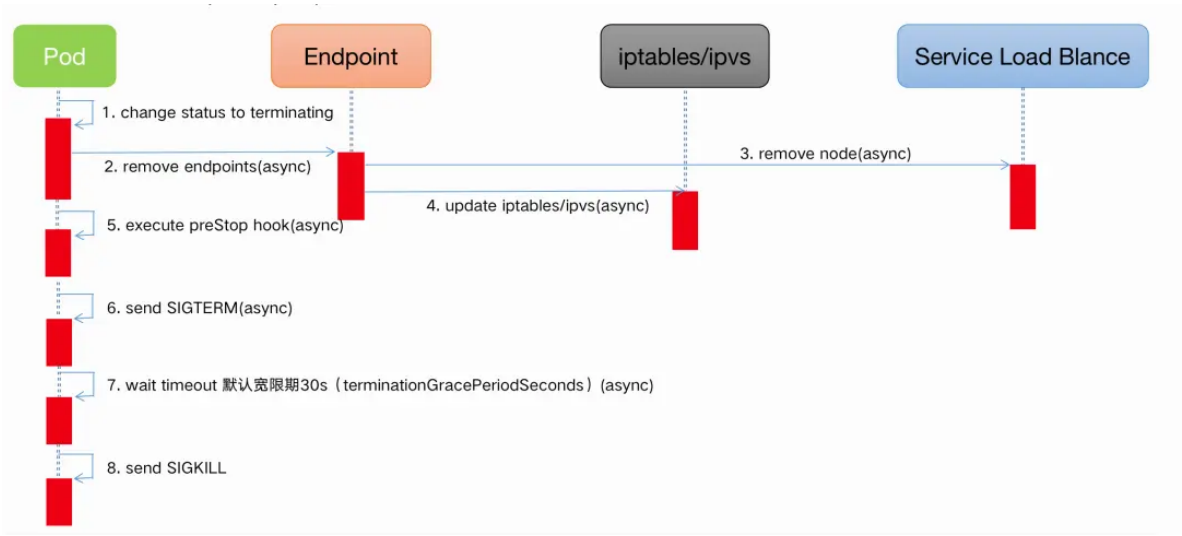
> ### POD 新建导致服务中断
>
> **具体原因**
>
> Pod Running 后被加入到 Endpoint 后端，容器服务监控到 Endpoint 变更后将 Pod Ip 加入到 SLB 后端。此时请求从 SLB 转发到 Pod 中，但是 Pod 业务代码还未初始化完毕，无法处理请求，导致服务中断。
>
> **解决方法**
>
> 为 Pod 配置就绪检测，等待业务代码初始化完毕后后再将 Node 加入到 SLB 后端。
>
> Hzero1.7 之前，整个系统依赖的 Springboot2.06，默认只有 /actuator/health 这一个健康检查端口，产线环境 Liveness 建议使用 /actuator/info，可以提高服务稳定性。



Hzero 1.7 版本开始，整个系统依赖 Springboot2.4.6，Springboot2.3.0 之后，Spring 在提供了 /actuator/health/readiness 和 /actuator/health/liveness 两个接口分别适配 K8S 的两个探针，建议健康。



> **POD 删除导致服务中断**
>
> 在 Deployment 做滚动更新时，一旦有新版本的 Pod 启动，就会删除旧的 Pod，一旦 Kubernetes 决定终止您的 Pod，就会发生一系列事件，需要对多个对象（如 Endpoint、Ipvs/iptables、SLB）进行状态同步，并且这些同步操作是异步执行的。
>
> Pod 在删除的时候，大致的生命周期如图所示：



1. Pod 状态变更

将 Pod 设置为 Terminating 状态，并从所有 Service 的 Endpoints 列表中删除。此时，Pod 停止获得新的流量，但在 Pod 中运行的容器不会受到影响，将会继续处理之前的请求；

2. 执行 PreStop Hook

Pod 删除时会触发 PreStop Hook，PreStop Hook 支持 Bash 脚本、TCP 或 HTTP 请求；

3. 发送 SIGTERM 信号：

向 Pod 中的容器发送 SIGTERM 信号；

4. 等待指定的时间：

TerminationGracePeriodSeconds 字段用于控制等待时间，默认值为 30 秒。该步骤与 PreStop Hook 同时执行，因此 TerminationGracePeriodSeconds 需要大于 PreStop 的时间，否则会出现 PreStop 未执行完毕，Pod 就被 Kill 的情况；

5. 发送 SIGKILL 信号：

等待指定时间后，如果容器在优雅终止宽限期后仍在运行，则会发送 SIGKILL 信号并强制删除。与此同时，所有的 Kubernetes 对象也会被清除。

具体原因

上述 1、2、3、4 步骤同时进行，因此有可能存在 Pod 收到 SIGTERM 信号并且停止工作后，还未从 Endpoints 中移除的情况。此时，请求从 Slb 转发到 Pod 中，而 Pod 已经停止工作，因此会出现服务中断。

解决方法

为 Pod 配置 PreStop Hook，使 Pod 收到 SIGTERM 时 Sleep 一段时间而不是立刻停止工作，从而确保从 SLB 转发的流量还可以继续被 Pod 处理，同时需要配合修改最大宽限时间（TerminationGracePeriodSeconds）

最大宽限时间修改示例：

-

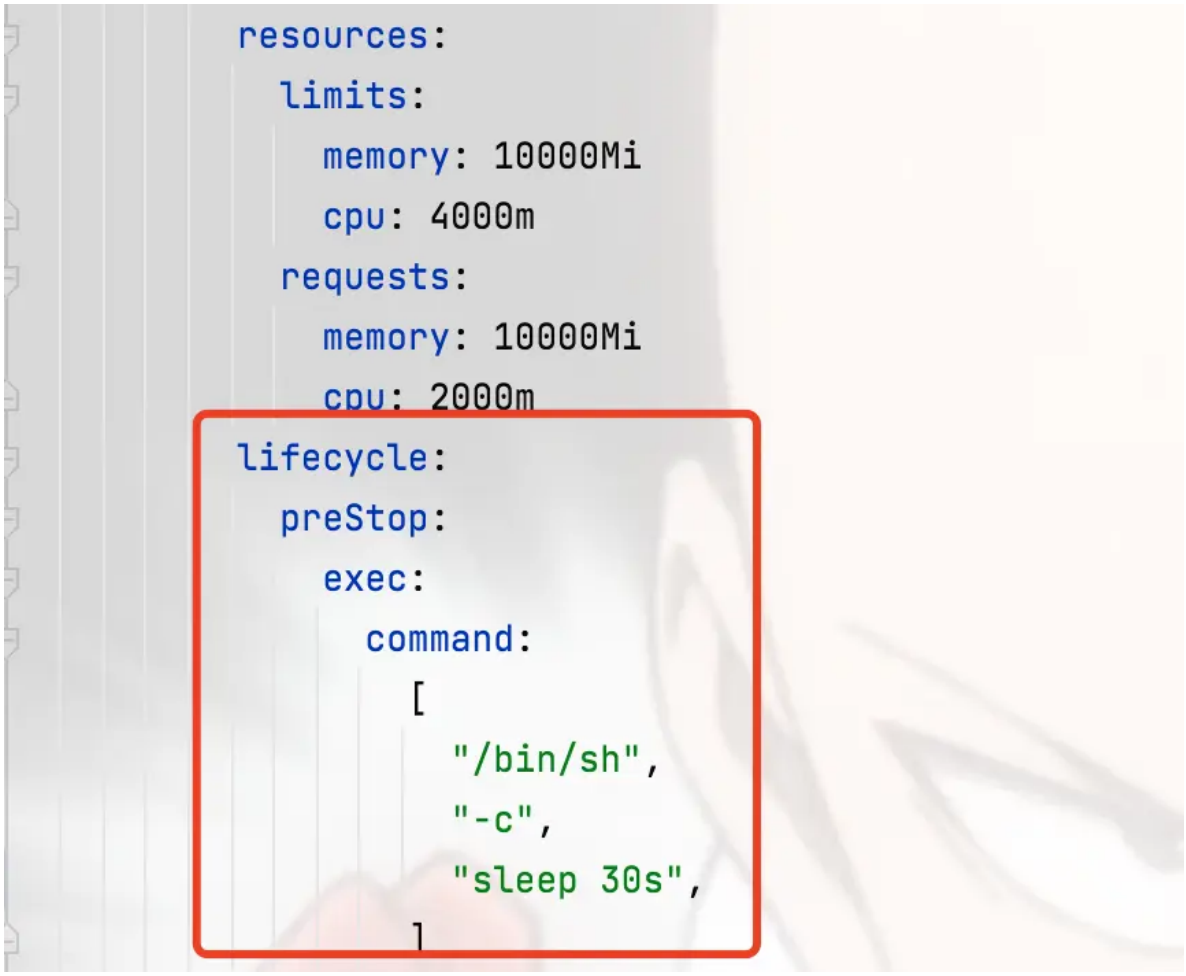
```
最大宽限时间修改示例：apiVersion: v1kind: Podmetadata:  name: nginx  namespace:
defaultspec:  containers:  - name: nginx    image: nginx
terminationGracePeriodSeconds: 50
```

K8S 优雅关闭 POD - 解决方案

> 现在我们的目标就是如何增强我们的应用程序能力，让它以真正的零宕机更新版本，首先，实现这个目标的前提条件是我们的容器要能正确处理终止信号，即进程会在 SIGTERM 上优雅地关闭。

解决方案 1

第一种思路，在 K8S Deployment 资源文件种配置响应 PreStop Hook，并且将 TerminationGracePeriodSeconds 调整为 30 以上（比 Prestop 大即可）



这里等待 30s 主要是用于等待处理残余流量，当然，Prestop 如果是定位为 30s，那么相应的 TerminationGracePeriodSeconds 得修改肯定要比 30s 大，比如 40s。

当然，这个时候也需要配合应用和容器本身的优雅关闭，这样才能从应用、容器、K8S 滚动三个层面同时实现优雅，但是 SpringCloud 本身的流量控制也比较多环节，Sleep 30s 也并不能做到 100% 的零宕机，只能说是能减少很大一部分的滚动更新造成的宕机问题。

Hzero1.7 版本之后，Hzero 底层使用的是 SpringBoot 2.46，在 Spring Boot 2.3.0 中，优雅停机非常容易实现，并且可以通过在应用程序配置文件中设置两个属性来进行管理

- Server.shutdown：此属性可以支持的值有
  - Immediate：这是默认值，将导致服务器立即关闭。
  - Graceful：启用优雅停机，并遵守 Spring.lifecycle.timeout-per-shutdown-phase 属性中给出的超时。
- Spring.lifecycle.timeout-per-shutdown-phase：采用 java.time.Duration 格式的值。

配置示例：

可以在 JVM 参数中添加 JAVA_OPTS= -Dserver.shutdown=graceful -Dspring.lifecycle.timeout-per-shutdown-phase=30s

再配合将 Prestop Sleep 等待时间设置为 40S，将 TerminationGracePeriodSeconds 设置为 50s。

配置效果：

应用中没有正在进行的要求。在这种情况下，应用程序将会直接关闭，而无需等待宽限期结束后才关闭。

如果应用中有正在处理的请求，则应用程序将等待宽限期结束后才能关闭。如果应用在宽限期之后仍然有待处理的请求，应用程序将抛出异常并继续强制关闭，但是这个配置只是该 SpringBoot 服务本身的服务优雅关停，还没涉及到 SpringCloud 流量控制等问题，相比较之前的关闭方式，多了应用本身的等待处理，在此期间 Hzero-register 以及 Hzero-gateway 有充足的时间去刷新服务，降低报错率，理论上无法完全规避滚动更新中 Hzero 架构服务中断的风险，但是优点在于方便快捷，不用代码侵入，能解决绝大部分问题。

想要做到完美滚动，那就还需要对应的流量剔除的动作配合起来使用。

解决方案 2

结合 K8S 的健康检查，和 Springboot（2.3 版本以上）中的自定义事件监听器，来构建自定义的健康检查结构，在 Prestop 的时候主动调用接口改变应用监听状态为拒绝接收流量，然后给参数约 40 秒事件去处理剩余流量，Readiness 探针通过健康检查会提前主动将 Endpoint 剔除，然后 POD 再进行关闭，这样就避免了滚动更新时候的流量损耗，这里需要自定义代码开发，开发部分有两个最重要的逻辑。

Eureka 流量剔除实现

就是以上说的 EurekaAutoServiceRegistration 可以实现

```
@RestController@RequestMapping(value = "/graceful/registry-service")public class GracefulOffline {    @Autowired    private EurekaAutoServiceRegistration eurekaAutoServiceRegistration;    @RequestMapping("/online")    public String online() {        this.eurekaAutoServiceRegistration.start();        return "execute online method, online success.";    }    @RequestMapping("/offline") public String offline() {        this.eurekaAutoServiceRegistration.stop();    return "execute offline method, offline success.";    }}
```

K8S 流量剔除实现

-

```
package com.adidas.token.api.v1;
import org.slf4j.Logger;import org.slf4j.LoggerFactory;import
org.springframework.beans.factory.annotation.Autowired;import
org.springframework.boot.availability.AvailabilityChangeEvent;import
org.springframework.boot.availability.ReadinessState;import
org.springframework.context.ApplicationEventPublisher;import
org.springframework.web.bind.annotation.GetMapping;import
org.springframework.web.bind.annotation.PathVariable;import
org.springframework.web.bind.annotation.RequestMapping;import
org.springframework.web.bind.annotation.RestController;
/** * control kubernetes readiness so we can stop traffic before kill pod * <p>
* url could configure in k8s "preStop"
*/@RestController@RequestMapping("/readiness")public class
ReadinessProbeController {
    private static final Logger logger =
LoggerFactory.getLogger(ReadinessProbeController.class);
    @Autowired    private ApplicationEventPublisher applicationEventPublisher;
    @GetMapping("/out-of-rotation/{sleepTime}")    public String
takeOOR(@PathVariable("sleepTime") Long sleepTime) {        logger.info("start
to take service out of rotation");
AvailabilityChangeEvent.publish(applicationEventPublisher,
"ReadinessProbeController", ReadinessState.REFUSING_TRAFFIC);        try {
     Thread.sleep(sleepTime * 1000);        } catch (InterruptedException e) {
         logger.warn("exception when take service out of rotation", e);        }
       logger.info("finish to take service out of rotation");        return
"REFUSING_TRAFFIC";    }
    @GetMapping("/take-into-rotation")    public String takeIntoRotation() {
   logger.info("start to take service into rotation");
AvailabilityChangeEvent.publish(applicationEventPublisher,
"ReadinessProbeController", ReadinessState.ACCEPTING_TRAFFIC);
logger.info("finish to take service into rotation");        return
"ACCEPTING_TRAFFIC";    }}
```

目前对于 Springcloud 应用来说最保险的方式，就是需要走这样的流量剔除的一套流程，上面两部分代码逻辑可以结合起来使用，封装成两个方法进行调用，需要自开发相关代码去控制：

\1. Eureka 节点剔除 --- 把服务从 Eureka 的服务清单里里面标记成下线状态；

\2. K8s 流量剔除 --- 把服务的 Readiness 状态改成下线状态（Springboot2.3 以上可以用 AvailabilityChangeEvent 实现）；

\3. 等待服务处理剩余流量；

\4. 等待时间到，服务停掉；

以上，就是总结的在 K8S 滚动更新中，零宕机目标的几种解决方案，大家在进行项目交付或者实施中可以根据实际情况来选择不同的方案

K8S 优雅升级系列（下） | 项目实战配置

> 前面我们已经讲解了如何让应用和容器优雅下线以及如何优雅滚动发布，相信大家对 “优雅” 已经有了一定的认识，这里提供某大型 ToC 项目的配置实例：

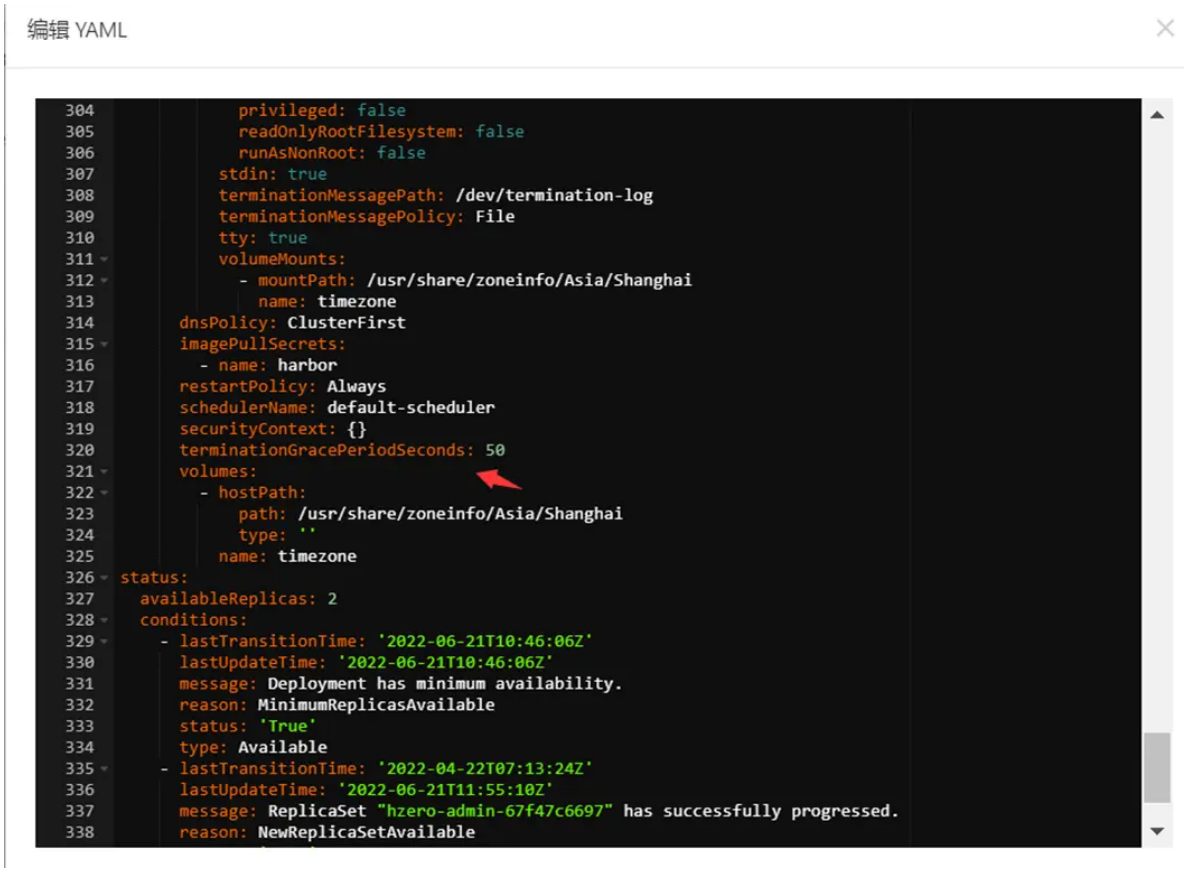
> **问题现象：**在生产环境进行发版、服务滚动更新的这段时间内，如果刚好有用户访问系统会出现报错。针对流量比较大且始终有顾客在使用的商城系统，短暂的出现访问出错也是不可接受的。
>
> **问题原因：**新的 POD 拉起，旧的 POD 销毁，在这个过程中资源会有一定的延迟更新，导致流量可能进入还没准备好的新 POD 或者进入已经销毁的旧 POD，从而出现访问出错。

旧的 POD 销毁这段时间，只要我们控制流量不进入销毁中的 POD，就能达到优雅下线的目的：

高版本 Spring Boot (Hzero1.7 及 1.7 之后)，已经提供了优雅下线的参数，配合 K8S 的钩子和优雅下线宽限期，可以达到真正的优雅下线，三个关键参数分别为 50 秒 + 40 秒 + 30 秒。

设置要求：TerminationGracePeriodSeconds>Prestop 中 Sleep 的时间 >Spring.lifecycle.timeout-per-shutdown-phas 时间

> **步骤一：****设置 50 秒 TerminationGracePeriodSeconds**



> **步骤二：****休眠 40 秒 Prestop**

利用 K8S 的钩子，在程序销毁前先睡眠 40 秒（建议值，具体时间视请求的处理时间而定）；在这 40 秒的时间内 Eureka 发现检测失败会把这个 Pod 剔除，不再转发流量到 Pod 上，同时也有 40 秒的时间能处理剩余的流量正常返回，从而避免出错的情况。

项目上通过步骤二单一配置，报错率大概降低了 98%，针对流量不大的 ToB 系统基本可以满足了，针对大流量的系统或者 ToC 的场景，建议把步骤一到步骤三做完（或者在休眠前调用自定义脚本，主动剔除 Eureka 的注册 POD，也能达到目的）

> **步骤三：****设置 30 秒 timeout-per-shutdown-phas**

停止设置为优雅停止，时间设置为 30 秒，可以在 Yml 文件中配置也可以设置在 VM 参数中

JAVA_OPTS=-Dserver.shutdown=graceful-Dspring.lifecycle.timeout-per-shutdown-phase=30s

新的 POD 拉起这段时间，只要我们控制流量不进入未完全准备好的 POD，就能达到优雅上线的目的，两个配置步骤解决：

> **步骤一：****设置探针**

K8S 提供了两种检查探针，分别为**存活检查**和**就绪检查**，从名字上也能清晰地知道两种探针的作用。

存活检查：让 K8S 知道应用是否还正常提供服务，如果应用已经挂了，K8S 会自动移除该 POD 并启动一个新的 POD 替换它；

就绪检查：让 K8S 知道应用是否已经准备好接收流量，只有就绪检查通过才会把流量转发到 POD，否则会停止向 POD 发送流量，直至检查通过；

探针配置如下：

协议	**HTTP**
路径	Hzero 的服务默认都集成了 Actuator 健康监控，直接配置即可。**Hzero1.7 版本及以上**，两种探针分别配置：/actuator/health/liveness/actuator/health/readiness **Hzero1.7 版本以下的**，两种探针都配置：/actuator/health
端口	Yml 配置文件的管理端口，Management.server.port
延迟探测时间	120 秒如果服务启动时间长可适当延长
执行探测频率	10 秒

超时的协议的间	HTTP 10 秒
不健康阈值	3

> **步骤二：配置服务注册**

之所以需要配置服务注册参数，是为了避免这种**情况**：

当探针延迟 120 秒 POD 状态还没变为 Running 时，Eureka 注册中心上的节点已经是 UP 的状态，如果是以 POD IP 的形式注册到 Eureka 上，此时内部 Fegin 调用的流量有可能会分发到非探测通过的 POD 上，导致程序出错。

我们的**解决整体思路**是配置改为 Hostname 注册值 Eureka，Hostname 和 SVC 名称一致，这样就会去请求 SVC 再负载到 POD 上，当 POD 未就绪的时候，就不会参与到 SVC 的负载上。

- **配置 PreferIpAddress**

修改 Bootstrap.yml，PreferIpAddress: ${EUREKA_INSTANCE_PREFER_IP_ADDRESS:true} 修改为 PreferIpAddress: ${EUREKA_INSTANCE_PREFER_IP_ADDRESS:false}；

或者在环境启动参数 EUREKA_INSTANCE_PREFER_IP_ADDRESS 设置为 False，效果是一样的。

- **配置 Hostname**

根据服务名称命名，且需要跟后续的 SVC 名称保持一致，修改 Bootstrap.yml，加入以下配置：

```
eureka:  instance:    hostname: hzero-gateway
```

或者服务启动指定参数 Eureka.instance.hostname=hzero-gateway，效果是一样的。

- **创建 SVC**

Name: Hzero-gateway，与 Bootstrap.yml 文件的 Hostname 需要保持一致，否则会报 500

```
apiVersion: v1kind: Servicemetadata:  labels:    choerodon.io/release: hzero-
gateway  name: hzero-gateway  namespace: xxxspec:  ports:    - port: 8080
targetPort: 8080      protocol: TCP      name: service-0    - port: 8081
targetPort: 8081      protocol: TCP      name: service-1  selector:
choerodon.io/release: hzero-gateway  type: ClusterIP
```

> 通过上述的两大配置五大步骤，能够达到系统 24 小时可用，代码发布不影响顾客前端操作体验的效果。
>
> 所谓优雅就是平滑切换不报错，核心思路是在新的 POD 拉起、旧的 POD 销毁这个过程中，也就是 POD 还没准备好的时间窗口，不要把流量分发过去。理解了流量会经过哪些组件、组件的缓存策略，要达到优雅这个目的，我们的配置方式可以是多种多样，但万变不离其宗，大家也可以自己尝试新的配置方式。