

## 04 | 如何借助GitOps实现应用秒级自动发布和回滚？

2022-12-12 王炜 来自北京



《云原生架构与GitOps实战》

课程介绍 >



讲述：王炜

时长 10:46 大小 9.83M



你好，我是王炜。

在上一节课，我为你介绍了 K8s 在自愈和自动扩容方面的强大能力，它们对提升业务稳定性有非常大的帮助。

其实，除了保障业务稳定以外，在日常软件研发的生命周期中，还有非常重要的一环：发布和回滚。发布周期是体现研发效率的一项重要指标，更早更快的发布有利于我们及时发现问题。

在我们有了关于容器化、K8s 工作负载的基础之后，这节课，我们来看看 K8s 应用发布的一般做法，此外，我还会带你从零开始构建 GitOps 工作流，体验 GitOps 在应用发布上为我们带来的全新体验。

在正式开始之前，你需要做好以下准备：

- 准备一台电脑（首选 Linux 或 macOS，Windows 系统注意操作差异）；
- [🔗 安装 Docker](#)；
- [🔗 安装 Kubectl](#)；
- 按照上一节课的内容在本地 Kind 集群安装 Ingress-Nginx。



## 传统 K8s 应用发布流程

还记得在上节课学习的如何创建 Deployment 工作负载吗？下面这段 Deployment Manifest 可以帮助你复习一下：

复制代码

```
1 apiVersion: apps/v1
2 kind: Deployment
3 metadata:
4   creationTimestamp: null
5   labels:
6     app: hello-world-flask
7   name: hello-world-flask
8 spec:
9   replicas: 2
10  selector:
11    matchLabels:
12      app: hello-world-flask
13  strategy: {}
14  template:
15    metadata:
16      creationTimestamp: null
17      labels:
18        app: hello-world-flask
19    spec:
20      containers:
21        - image: lyzhang1999/hello-world-flask:latest
22          name: hello-world-flask
23          resources: {}
24  status: {}
```

当我们在部署 Deployment 工作负载的时候，Image 字段同时指定了镜像名称和版本号。在发布应用的过程中，一般会先修改 Manifest 镜像版本，再使用 `kubectl apply` 重新将 Manifest 应用到集群来更新应用。

你可能会问，那在升级应用的过程中，新老版本的切换会导致服务中断吗？答案当然是不会的，并且 K8s 将会自动处理，无需人工干预。



接下来，我们进入实战环节。我们先尝试通过手动的方式来更新应用，这也是传统 K8s 发布应用的过程。

通常，更新应用可以使用下面三种方式：

- 使用 `kubectl set image` 命令；
- 修改本地的 Manifest ；
- 修改集群内 Manifest 。

## 通过 `kubectl set image` 命令更新应用

要想更新应用，最简单的方式是通过 `kubectl set image` 来更新集群内已经存在工作负载的镜像版本，例如更新 `hello-world-flask` Deployment 工作负载：

复制代码

```
1 $ kubectl set image deployment/hello-world-flask hello-world-flask=lyzhang1999/  
2 deployment.apps/hello-world-flask image updated
```

为了方便你动手实践，我已经给你制作了 `hello-world-flask:v1` 版本的镜像，新镜像版本修改了 Python 的返回内容，你可以直接使用。

当 K8s 接收到镜像更新的指令时，K8s 会用新的镜像版本重新创建 Pod。你可以使用 `kubectl get pods` 来查看 Pod 的更新情况：

复制代码

```
1 $ kubectl get pods  
2 NAME                                READY   STATUS    RESTARTS   AGE  
3 hello-world-flask-8f94845dc-qsm8b   1/1     Running   0           3m38s  
4 hello-world-flask-8f94845dc-spd6j   1/1     Running   0           3m21s  
5 hello-world-flask-64dd645c57-rfhw5   0/1     ContainerCreating   0           1s  
6 hello-world-flask-64dd645c57-ml74f   0/1     ContainerCreating   0           0s
```

在更新 Pod 的过程中，K8s 会确保先创建新的 Pod，然后再终止旧镜像版本的 Pod。Pod 的副本数始终保持在我们在 Deployment Manifest 中配置的 2。



现在，你可以打开浏览器访问 127.0.0.1，查看返回内容：

复制代码

```
1 Hello, my v1 version docker images! hello-world-flask-8f94845dc-bpgnp
```

通过返回内容我们可以发现，新镜像对应的 Pod 已经替换了老的 Pod，这也意味着我们的应用更新成功了。

**从本质上来看，`kubectl set image` 是修改了集群内已部署的 Deployment 工作负载的 Image 字段，继而触发了 K8s 对 Pod 的变更。**

有时候，我们在一次发布中希望变更的内容不仅仅是镜像版本，可能还有副本数、端口号等等。这时候，我们可以对新的 Manifest 文件再执行一次 `kubectl apply`，K8s 会比对它们之间的差异，然后做出变更。

## 通过修改本地的 Manifest 更新应用

除了使用 `kubectl set image` 来更新应用，我们还可以修改本地的 Manifest 文件并将其重新应用到集群来更新。

以 `hello-world-flask Deployment` 为例，我们重新把镜像版本修改为 `latest`，将下面的内容保存为 `new-hello-world-flask.yaml`：

复制代码

```
1 apiVersion: apps/v1
2 kind: Deployment
3 metadata:
4   labels:
5     app: hello-world-flask
6   name: hello-world-flask
7 spec:
8   replicas: 2
9   selector:
10    matchLabels:
11      app: hello-world-flask
12   template:
```

```
13     metadata:
14         labels:
15             app: hello-world-flask
16     spec:
17         containers:
18             - image: lyzhang1999/hello-world-flask:latest
19               name: hello-world-flask
```



接下来，执行 `kubectl apply -f new-hello-world-flask.yaml` 来更新应用：

```
1 $ kubectl apply -f new-hello-world-flask.yaml
2 deployment.apps/hello-world-flask configured
```

复制代码

也就是说，`kubectl apply` 命令会自动处理两种情况：

1. 如果该资源不存在，那就创建资源；
2. 如果资源存在，那就更新资源。

到这里，我相信有一些同学可能会有疑问，如果我本地的 **Manifest** 找不到了，我可以直接更新集群内已经存在的 **Manifest** 吗？答案是肯定的。也就是说，我们还可以直接编辑集群内的 **Manifest** 来更新应用，这就是更新应用的第三种方式。

## 通过修改集群内 **Manifest** 更新应用

以 `hello-world-flask Deployment` 为例，要直接修改集群内已部署的 **Manifest**，你可以使用 `kubectl edit` 命令：

```
1 $ kubectl edit deployment hello-world-flask
```

复制代码

执行命令后，`kubectl` 会自动为我们下载集群内的 **Manifest** 到本地，并且用 **VI** 编辑器自动打开。你可以进入 **VI** 的编辑模式修改任何字段，**保存退出后修改即时生效**。



总结来说，要更新 K8s 的工作负载，我们可以修改本地的 Manifest，再使用 `kubectl apply` 将它重新应用到集群内，或者通过 `kubectl edit` 命令直接修改集群内已存在的工作负载。



在实际项目的实践中，负责更新应用的同学早期可能会在自己的电脑上操作，然后把这部分操作挪到 CI 过程，例如使用 Jenkins 来执行。

但是，随着项目的发展，我们会需要发布流程更加自动化、安全、可追溯。这时候，我们应该考虑用 GitOps 的方式来发布应用。

## 从零搭建 GitOps 发布 workflow

在正式搭建 GitOps 之前，我想先让你对 GitOps 有个简单的理解。通俗来说，GitOps 就是以 Git 版本控制为理念的 DevOps 实践。

对于这节课要设计的 GitOps 发布 workflow 来说，我们会将 Manifest 存储在 Git 仓库中作为期望状态，一旦修改并提交了 Manifest，那么 GitOps 工作流就会自动比对 Git 仓库和集群内工作负载的实际差异，并进行部署。

## 安装 FluxCD 并创建工作流

要实现 GitOps 工作流，首先我们需要一个能够帮助我们监听 Git 仓库变化，自动部署的工具。这节课，我以 FluxCD 为例，带你一步步构建出一个 GitOps 工作流。

接下来，我们进入实战环节。

首先，我们需要在集群内安装 FluxCD：

复制代码

```
1 $ kubectl apply -f https://ghproxy.com/https://raw.githubusercontent.com/lyzhan
```

由于安装 FluxCD 的工作负载比较多，你可以使用 `kubectl wait` 来等待安装完成：

复制代码

```
1 $ kubectl wait --for=condition=available --timeout=300s --all deployments -n fl
2 deployment.apps/helm-controller condition met
3 deployment.apps/image-automation-controller condition met
```

```
4 deployment.apps/image-reflector-controller condition met
5 deployment.apps/kustomize-controller condition met
6 deployment.apps/notification-controller condition met
7 deployment.apps/source-controller condition met
```



接下来，在本地创建 **fluxcd-demo** 目录：

复制代码

```
1 $ mkdir fluxcd-demo && cd fluxcd-demo
```

然后，我们在 **fluxcd-demo** 目录下创建 **deployment.yaml** 文件，并将下面的内容保存到这个文件里：

复制代码

```
1 apiVersion: apps/v1
2 kind: Deployment
3 metadata:
4   labels:
5     app: hello-world-flask
6   name: hello-world-flask
7 spec:
8   replicas: 2
9   selector:
10    matchLabels:
11      app: hello-world-flask
12   template:
13     metadata:
14       labels:
15         app: hello-world-flask
16     spec:
17       containers:
18       - image: lyzhang1999/hello-world-flask:latest
19         name: hello-world-flask
```

最后，在 **Github** 或 **Gitlab** 中创建 **fluxcd-demo** 仓库。为了方便测试，我们需要将仓库设置为公开权限，主分支为 **Main**，并将我们创建的 **Manifest** 推送至远端仓库：

复制代码

```
1 $ ls
2 deployment.yaml
3 $ git init
4 .....
```

```
5 Initialized empty Git repository in /Users/wangwei/Downloads/fluxcd-demo/.git/
6 $ git add -A && git commit -m "Add deployment"
7 [master (root-commit) 538f858] Add deployment
8 1 file changed, 19 insertions(+)
9 create mode 100644 deployment.yaml
10 $ git branch -M main
11 $ git remote add origin https://github.com/lyzhang1999/fluxcd-demo.git
12 $ git push -u origin main
```



这是我的 [仓库地址](#)，你可以参考一下。

下一步，我们为 FluxCD 创建仓库连接信息，将下面的内容保存为 fluxcd-repo.yaml:

复制代码

```
1 apiVersion: source.toolkit.fluxcd.io/v1beta2
2 kind: GitRepository
3 metadata:
4   name: hello-world-flask
5 spec:
6   interval: 5s
7   ref:
8     branch: main
9   url: https://github.com/lyzhang1999/fluxcd-demo
```

注意，要将 URL 字段修改为你实际仓库的地址并使用 HTTPS 协议，branch 字段设置 main 分支。这里的 interval 代表每 5 秒钟主动拉取一次仓库并把它作为制品存储。

接着，使用 kubectl apply 将其 GitRepository 对象部署到集群内:

复制代码

```
1 $ kubectl apply -f fluxcd-repo.yaml
2 gitrepository.source.toolkit.fluxcd.io/hello-world-flask created
```

你可以使用 kubectl get gitrepository 来检查配置是否生效:

复制代码

```
1 $ kubectl get gitrepository
2 NAME                                URL                                AGE   READY
3 hello-world-flask                   https://github.com/lyzhang1999/fluxcd-demo 5s    True
```



接下来，我们还需要为 FluxCD 创建部署策略。将下面的内容保存为 `fluxcd-kustomize.yaml`：



```
1 apiVersion: kustomize.toolkit.fluxcd.io/v1beta2
2 kind: Kustomization
3 metadata:
4   name: hello-world-flask
5 spec:
6   interval: 5s
7   path: ./
8   prune: true
9   sourceRef:
10    kind: GitRepository
11    name: hello-world-flask
12    targetNamespace: default
```

在上面的配置中，`interval` 参数表示 FluxCD 会每 5 秒钟运行一次工作负载差异对比，`path` 参数表示我们的 `deployment.yaml` 位于仓库的根目录中。FluxCD 在对比期望状态和集群实际状态的时候，如果发现差异就会触发重新部署。

我们再次使用 `kubectl apply` 将 `Kustomization` 对象部署到集群内：

复制代码

```
1 $ kubectl apply -f fluxcd-kustomize.yaml
2 kustomization.kustomize.toolkit.fluxcd.io/hello-world-flask created
```

同样地，你可以使用 `kubectl get kustomization` 来检查配置是否生效：

复制代码

```
1 $ kubectl get kustomization
2 NAME                AGE      READY   STATUS
3 hello-world-flask    8m21s   True    Applied revision: main/8260f5a0ac1e4ccdba64
```

配置完成后，接下来，我们就可以正式体验 **GitOps** 的秒级自动发布和回滚了。

## 自动发布

现在，我们修改 fluxcd-demo 仓库的 deployment.yaml 文件，将 image 字段的镜像版本从 latest 修改为 v1:



复制代码

```
1 .....
2     spec:
3         containers:
4             - image: lyzhang1999/hello-world-flask:v1 # 修改此处
5               name: hello-world-flask
6     .....
```

然后，我们将修改推送到远端仓库:

复制代码

```
1 $ git add -A && git commit -m "Update image tag to v1"
2 $ git push origin main
```

你可以使用 `kubectl describe kustomization hello-world-flask` 查看触发重新部署的事件:

复制代码

```
1 $ kubectl describe kustomization hello-world-flask
2 .....
3 Status:
4   Conditions:
5     Last Transition Time: 2022-09-10T03:46:37Z
6     Message:             Applied revision: main/8260f5a0ac1e4ccdba64e074d1ee2
7     Reason:              ReconciliationSucceeded
8     Status:              True
9     Type:                Ready
10  Inventory:
11    Entries:
12      Id:                 default_hello-world-flask_apps_Deployment
13      V:                  v1
14      Last Applied Revision: main/8260f5a0ac1e4ccdba64e074d1ee2c154956f12d
15      Last Attempted Revision: main/8260f5a0ac1e4ccdba64e074d1ee2c154956f12d
16      Observed Generation: 1
17    .....
```

从返回的结果可以看出，我们将镜像版本修改为了 v1，并且，FluxCD 最后一次部署仓库的 Commit ID 是 8260f5a0ac1e4ccdba64e074d1ee2c154956f12d，这对应了我们最后一次的提

交记录，说明变更已经生效了。

现在，我们打开浏览器访问 **127.0.0.1**，可以看到 **v1** 镜像版本的输出内容：



复制代码

```
1 Hello, my v1 version docker images! hello-world-flask-6d7b779cd4-spf4q
```

通过上面的配置，我们让 **FluxCD** 自动完成了监听修改、比较和重新部署三个过程。怎么样，**GitOps** 的发布流程是不是比手动发布方便多了呢？

接下来我们再感受一下 **GitOps** 的快速回滚能力。

## 发布回滚

既然 **GitOps** 工作流中，**Git** 仓库是描述期望状态的唯一可信源，那么我们是不是只要对 **Git** 仓库执行回滚，就可以实现发布回滚呢？

我们通过实战来验证一下这个猜想。

要回滚 **fluxcd-demo** 仓库，首先需要找到上一次的提交记录。我们可以使用 **git log** 来查看它：

复制代码

```
1 $ git log
2 commit 900357f4cfec28e3f80fde239906c1af4b807be6 (HEAD -> main, origin/main)
3 Author: wangwei <434533508@qq.com>
4 Date: Sat Sep 10 11:24:22 2022 +0800
5
6 Update image tag to v1
7
8 commit 75f39dc58101b2406d4aaacf276e4d7b2d429fc9
9 Author: wangwei <434533508@qq.com>
10 Date: Sat Sep 10 10:35:41 2022 +0800
11
12 first commit
```

可以看到，上一次的 **commit id** 为 **75f39dc58101b2406d4aaacf276e4d7b2d429fc9**，接下来使用 **git reset** 来回滚到上一次提交，并强制推送到 **Git** 仓库：

```

1 $ git reset --hard 75f39dc58101b2406d4aaacf276e4d7b2d429fc9
2 HEAD is now at 538f858 Add deployment
3
4 $ git push origin main -f
5 Total 0 (delta 0), reused 0 (delta 0), pack-reused 0
6 To https://github.com/lyzhang1999/fluxcd-demo.git
7 + 8260f5a...538f858 main -> main (forced update)

```



再次使用 `kubectl describe kustomization hello-world-flask` 查看触发重新部署的事件：

```

1 .....
2 Status:
3   Conditions:
4     Last Transition Time: 2022-09-10T03:51:28Z
5     Message:             Applied revision: main/538f858909663f4be3a62760cb571
6     Reason:              ReconciliationSucceeded
7     Status:              True
8     Type:                Ready
9   Inventory:
10    Entries:
11      Id:                 default_hello-world-flask_apps_Deployment
12      V:                 latest
13    Last Applied Revision: main/538f858909663f4be3a62760cb571529eb50a831
14    Last Attempted Revision: main/538f858909663f4be3a62760cb571529eb50a831
15    Observed Generation:  1
16 .....

```

从返回结果的 `Last Applied Revision` 可以看出，FluxCD 已经检查到了变更，并已经进行了同步。

再次打开浏览器访问 `127.0.0.1`，可以看到返回结果已回滚到了 `latest` 镜像对应的内容：

```

1 Hello, my first docker images! hello-world-flask-56fbff68c8-c8dc4

```

到这里，我们就成功实现了 GitOps 的发布和回滚。

## 总结

这节课，我为你归纳了 K8s 更新应用镜像的 3 种基本操作，他们分别是：



1. `kubectl set image`;
2. 修改本地 Manifest 并重新执行 `kubectl apply -f`;
3. 通过 `kubectl edit` 直接修改集群内的工作负载。

这种手动更新应用的方法效率非常低，最重要的是很难回溯，会让应用回滚变得困难。所以，我们引入了一种全新 **GitOps** 工作流的发布方式来解决这些问题。

在这节课的实战当中，我们只实现了 **GitOps** 环节中的一小部分，我希望通过这个小小的试炼，让你认识到 **GitOps** 的价值。

在实际项目中，构建端到端的 **GitOps** 工作流其实还有非常多的细节，例如如何在修改代码后自动构建并推送镜像，如何自动更新 Manifest 仓库等，这些进阶的内容我都会在后续的课程中详细介绍。

另外，能实现 **GitOps** 的工具其实并不止 **FluxCD**，在你为实际项目构建生产级的 **GitOps** 工作流时，我推荐你使用 **ArgoCD**，这也是我们这个专栏接下来会重点介绍的内容。

最后，在前面几节课里，我们引出了非常多 K8s 相关的概念，例如工作负载、**Service**、**Ingress**、**HPA** 等等，为了快速实战并让你感受 K8s 和 **GitOps** 的价值，之前我并没有详细解释这些概念。但当你要将真实的项目迁移到 K8s 的时候，这些内容是我们必须要熟练掌握的。

所以，为了让你在工作过程中对 K8s 更加得心应手，我会在接下来第二模块为你提供零基础的 K8s 极简入门教程。我会详细介绍之前出现过的 K8s 常用对象，让你真正掌握他们，扫除将公司项目迁移到 K8s 的技术障碍，**迈出 GitOps 的第一步**。

## 思考题

最后，给你留一道思考题吧。

请你分享一下你们现在使用的发布方案是什么？相比 **GitOps** 的发布方式，你认为它有哪些优缺点呢？

欢迎你给我留言交流讨论，你也可以把这节课分享给更多的朋友一起阅读。我们下节课见。



分享给需要的人，Ta购买本课程，你将得 18 元

生成海报并分享

赞 4 提建议

© 版权归极客邦科技所有，未经许可不得传播售卖。页面已增加防盗追踪，如有侵权极客邦将依法追究其法律责任。

上一篇 03 | 业务永不宕机，K8s如何实现自动扩容和自愈？

## 更多课程推荐

# 云原生架构与 GitOps 实战

## 即学即用，攻破云原生核心技术

王炜  
前腾讯云 CODING 架构师



新版升级：点击「👤请朋友读」，20位好友免费读，邀请订阅更有**现金**奖励。

## 精选留言 (7)

写留言




includestdio.h

2022-12-13 来自广东

我们目前是通过gitlab-ci和自己搭建的CD平台实现流水线，代码变更后ci发起镜像构建，并推



送到镜像仓库，CD平台通过hook监听到CI流水线完成，重建Pod并拉取最新的镜像。缺点是#我们的镜像版本号只区分了 dev mirror prod，回滚不是很优雅，只能是通过gitlab回滚代码重新构建镜像，重新触发CD，效率低，对开发同学也不太友好。最近正在着手看怎么调整，实现通过镜像版本号回滚。 <https://shikey.com/>

作者回复: 欢迎你关注后续的课程，你的问题在第 17 讲中有提到。

我建议你先使用 git commit id 作为镜像版本号，这样可以把代码版本和镜像对应起来，回滚的话只要找到 commit id 就可以了。对于生产镜像，则可以采用额外的策略，例如 prod-commit\_id 把他和常规开发镜像区分开。

共 4 条评论 >

👍 1



Y

2022-12-15 来自广东

我本地windows实验成功了，感谢大佬

作者回复: 加油，后续课程还有更好玩的实验。



Fchen

2022-12-14 来自广东

这个gitops阶段看起来并不完整，最直接的是缺少了镜像构建阶段的流程。对于企业生产使用，还是简单了点，我们是自研的，可以根据不同角色，不同业务场景，不同发布场景等做一些适配

作者回复: 是的，本章介绍的是一个最简单的例子。企业级的 GitOps 工作流在 22 讲中有介绍，期待再次看到你的回复~



Promise

2022-12-14 来自广东

我们目前使用的方案是gitlab+Tekton+Argocd的方案但是我只搞了80%还没有完成在打包完镜像以后修改Helm部分。Tekton负责CI的部分，通过git提交触发gitlab的的push事件，Tekton监听push事件，使用doud的方式打包镜像。Argocd负责CD的部分监听gitlab上Helm项目的变化，然后自动部署到k8s上。老师会讲kaniko打包镜像吗？还有CICD项目很多时使用Tekton如何管理，是一个项目创建一个pipeline吗？

作者回复: 这部分内容在第 18 讲中有介绍，期待你的留言。



JianXu

2022-12-13 来自广东



天下无鱼

<https://shikey.com/>

老师，你能介绍一下为什么flux 能成功吗？他比其他gitops 方案到底强在哪里

作者回复: FluxCD 在 11 月底刚通过 CNCF 的评审，进入了毕业阶段。

这意味着社区已经在生产环境下大规模采用了 FluxCD，它的稳定性得到了充分的验证。

GitOps 领域目前两大工具中，FluxCD 的强项在于做集成，而 ArgoCD 更适合工程实践。这两款工具都非常优秀，所以在专栏里我都有进行介绍。

关于 ArgoCD，在后续第 22 讲中会深入介绍，期待我们再见面。

共 2 条评论 >



陈斯佳

2022-12-13 来自广东

我们现在使用的是在Jenkins上通过Terraform部署Helm chart，只要修改Terraform里的镜像版本号就能部署或回滚应用。类似这个lab: <https://github.com/chance2021/devopsdaydayup/blob/main/004-TerraformDockerDeployment/README.md>

作者回复: 也是一种方案。

不过在工程实践中不建议在 CI（Jenkins）里干持续部署的活，本质上持续部署需要更多的能力支持。比如在界面上应该能够很方便看出应用版本、健康状态、应用资源拓扑等，在部署能力支持上，可能还需要蓝绿部署、灰度和金丝雀发布，甚至是结合人工审核定制发布工作流。

这部分的内容可以在第 22、24、25 和 26 将深入了解。

共 2 条评论 >



helloworld

2022-12-12 来自北京

good

