

21 | 应用定义：如何使用 Helm 定义应用？

2023-01-25 王伟 来自北京



天下无鱼

<https://shikey.com/>

《云原生架构与GitOps实战》

课程介绍 >



讲述：王伟

时长 16:12 大小 14.81M



你好，我是王伟。

在上一节课，我们学习了如何使用 Kustomize 定义应用。在将示例应用改造成 Kustomize 应用的过程中，我介绍了 Kustomize base 和 overlay 的概念，并通过 3 个不同环境的配置差异来说明如何对 base 通用资源的字段值进行覆盖。

在使用 Kustomize 对某个对象进行覆写时，你可能注意到了一个细节，那就是我们需要了解 base 目录下通用 Kubernetes 对象的具体细节，例如工作负载的名称和类型，以及字段的结构层级。当业务应用比较简单的时候，由于 Kubernetes 对象并不多，所以提前了解这些细节并没有太大的问题。

但是，当业务应用变得复杂，例如有数十个微服务场景时，那么 Kubernetes 对象可能会有上百个之多，这时候 Kustomize 的应用定义方式可能会变得难以维护，尤其是当我们在 kustomization.yaml 文件定义大量覆写操作时，这种隐式的定义方式会让人产生迷惑。

其次，如果我们站在应用的发行角度来说，你会发现 Kustomize 对最终用户暴露所有 Kubernetes 对象概念的方式太过于底层，我们可能需要一种更上层的应用定义方式。



所以，社区诞生了另一种应用定义方式：**Helm**。

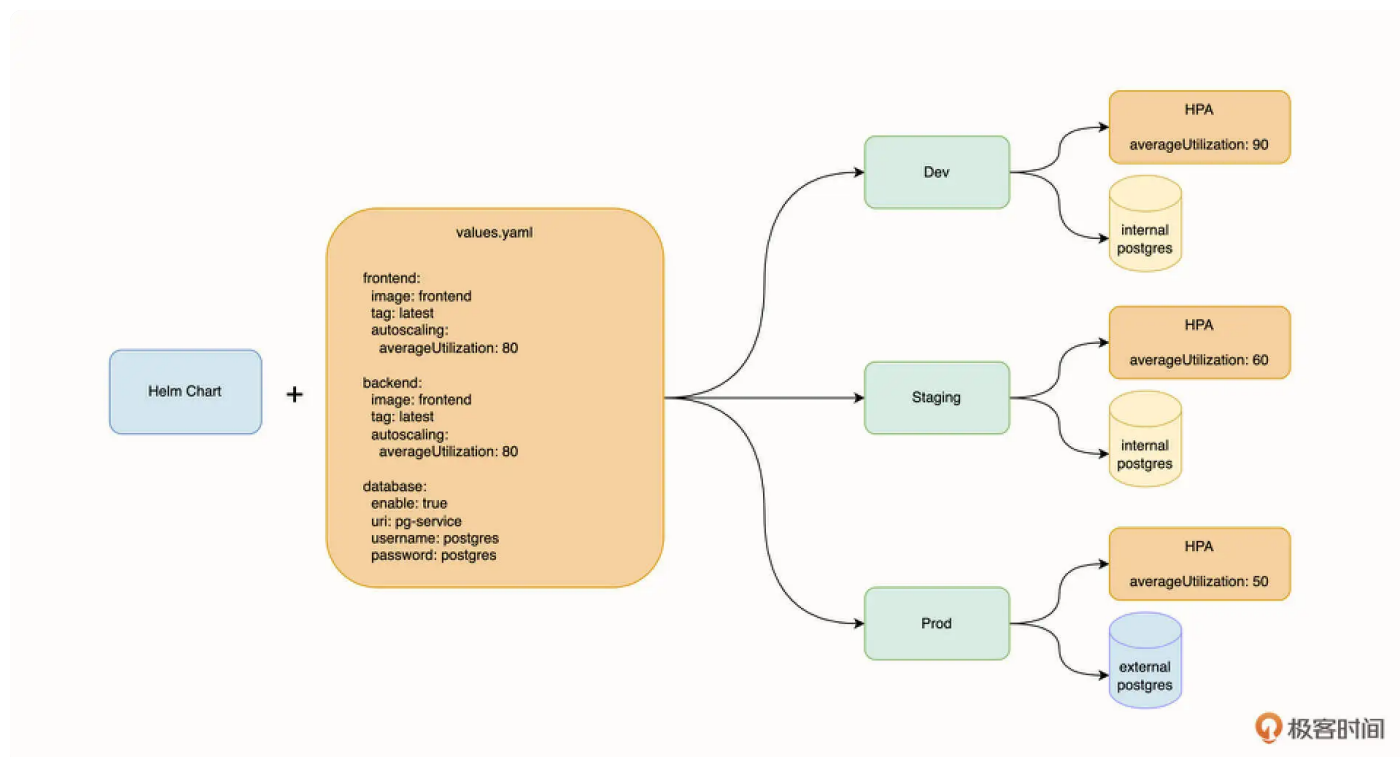
Helm 是一种真正意义上的 Kubernetes 应用的包管理工具，它对最终用户屏蔽了 Kubernetes 对象概念，将复杂度左移到了应用开发者侧，终端用户只需要提供安装参数，就可以将应用安装到 Kubernetes 集群内。

在这节课，我仍然以示例应用为例子，把它从原始的 Kubernetes Manifest 改造成 Helm 应用，在实践的过程中带你进一步了解 Helm 相关的概念以及具体用法。当然，要想在一节课内完整介绍 Helm 是有困难的，所以我还是从实战入手，尽可能精简内容，以便让你快速掌握 Helm 的基本使用方法。

在进入实战之前，你需要在本地安装 Helm，具体你可以参考 [🔗 第 19 讲](#) 的内容，同时将 [🔗 示例应用](#) 代码仓库克隆到本地，仓库中也包含了这节课的代码，供你参考。

实战简介和基本概念

我们先来看将示例应用改造成 Helm 之后，我们期望得到的效果，如下图所示。



我们希望能实现和上一讲 **Kustomize** 一样的效果，将同一个 **Helm Chart** 部署到 **Dev**（开发）、**Staging**（预发布）、**Prod**（生产）三个环境时，控制不同环境的 **HPA**、数据库以及镜像版本配置。



环境差异方面和上一讲提到的一致，除了 **Prod** 环境以外，其他两个环境都使用在集群内部署的 **postgres** 数据库，另外，三个环境的 **HPA CPU** 配置也不同。

上面的图中出现两个新的概念：**Helm Chart** 和 **values.yaml**。接下来我们简单介绍一下它们。

Helm Chart 和 values.yaml

Chart 是 **Helm** 的一种应用封装格式，它由一些特定文件和目录组成。为了方便 **Helm Chart** 存储、分发和下载，它采用 **tgz** 的格式对文件和目录进行打包。

在第 19 讲中，我们通过 **Helm** 命令安装了 **Cert-manager**，实际上 **Helm** 会先从指定的仓库下载 **tgz** 格式的 **Helm Chart**，然后再进行安装。

一个标准的 **Helm Chart** 目录结构是下面这样的。

复制代码

```
1 $ ls
2 Chart.yaml  templates  values.yaml
```

其中，**Chart.yaml** 文件是 **Helm Chart** 的描述文件，例如名称、描述和版本等。

templates 目录用来存放模板文件，你可以把它视作 **Kubernetes Manifest**，但它和 **Manifest** 最大的区别是，模板文件可以包含变量，变量的值则来自于 **values.yaml** 文件定义的内容。

values.yaml 文件是安装参数定义文件，它不是必需的。在 **Helm Chart** 被打包成 **tgz** 包时，如果 **templates** 目录下的 **Kubernetes Manifest** 包含变量，那么你需要通过它来提供默认的安装参数。作为最终用户，当安装某一个 **Helm Chart** 的时候，也可以提供额外的 **YAML** 文件来覆盖默认值。比如在上面的期望效果图中，我们为同一个 **Helm Chart** 提供不同的安装参数，就可以得到具有配置差异的多套环境。

Helm Release

Helm Release 实际上是一个“安装”阶段的概念，它指的是本次安装的唯一标识（名称）。

我们知道 Helm Chart 实际上是一个应用安装包，只有在安装（实例化）它时才会生效。它可以在同一个集群中甚至是同一个命名空间下安装多次，所以我们就需要为每次安装都起一个唯一的名字，这就是 Helm Release Name。

改造示例应用

接下来，我们开始改造示例应用，下面所有的命令都默认在示例应用根目录下执行。

创建 Helm Chart 目录结构

首先，进入示例应用目录并创建 helm 目录。

```
1 $ cd kubernetes-example && mkdir helm
```

复制代码

然后，我们根据 Helm Chart 的目录结构进一步创建 templates 目录、Chart.yaml 以及 values.yaml。

```
1 $ mkdir helm/templates && touch helm/Chart.yaml && touch helm/values.yaml
```

复制代码

现在，helm 目录结构是下面这样。

```
1 $ ls helm
2 Chart.yaml  templates  values.yaml
```

复制代码

到这里，Helm Chart 目录结构就创建完成了。

配置 Chart.yaml 内容

接下来，我们需要将 Helm Chart 的基础信息写入 Chart.yaml 文件中，将下面的内容复制到 Chart.yaml 文件内。

```
1 apiVersion: v2
2 name: kubernetes-example
3 description: A Helm chart for Kubernetes
4 type: application
5 version: 0.1.0
6 appVersion: "0.1.0"
```



其中，`apiVersion` 字段设置为 `v2`，代表使用 Helm 3 来安装应用。

`name` 表示 Helm Chart 的名称，当使用 `helm install` 命令安装 Helm Chart 时，指定的名称也就是这里配置的名称。

`description` 表示 Helm Chart 的描述信息，你可以根据需要填写。

`type` 表示类型，这里我们将其固定为 `application`，代表 Kubernetes 应用。

`version` 表示我们打包的 Helm Chart 的版本，当使用 `helm install` 时，可以指定这里定义版本号。Helm Chart 的版本就是通过这个字段来管理的，当我们发布新的 Chart 时，需要更新这里的版本号。

`appVersion` 和 Helm Chart 无关，它用于定义应用的版本号，建立当前 Helm Chart 和应用版本的关系。

最简单的 Helm Chart

之前我们提到过，`helm/templates` 目录是用来存放模板文件的，这个模板文件也可以是 Kubernetes Manifest。所以，我们现在尝试不使用 Helm Chart 的模板功能，而是直接将 `deploy` 目录下的 Kubernetes Manifest 复制到 `helm/templates` 目录下。

```
1 $ cp -r deploy/ helm/templates/
```

现在，`helm` 目录的结构如下。


```
1 helm
2 |— Chart.yaml
3 |— templates
4 |   |— deploy
5 |       |— backend.yaml
6 |       |— database.yaml
7 |       |— frontend.yaml
8 |       |— hpa.yaml
9 |       |— ingress.yaml
10 |— values.yaml
```



其中，`values.yaml` 的文件内容为空。

到这里，一个最简单的 **Helm Chart** 就编写完成了。在这个 Helm Chart 中，`templates` 目录下的 Manifest 的内容是确定的，安装这个 Helm Chart 等同于使用 `kubectl apply` 命令，接下来我们尝试使用 `helm install` 命令来安装这个 Helm Chart。

```
1 $ helm install my-kubernetes-example ./helm --namespace example --create-namesp
2 NAME: my-kubernetes-example
3 LAST DEPLOYED: Thu Oct 20 15:55:31 2022
4 NAMESPACE: example
5 STATUS: deployed
6 REVISION: 1
7 TEST SUITE: None
```

在上面这条命令中，我们指定了应用的名称为 `my-kubernetes-example`，Helm Chart 目录为 `./helm` 目录，并且为应用指定了命名空间为 `example`。要注意的是，`example` 命名空间并不存在，所以我同时使用 `--create-namespace` 来让 Helm 自动创建这个命名空间。

此外，这里还有一个非常重要的概念：**Release Name**。在安装时，我们需要指定 Release Name 也就是 `my-kubernetes-example`，它和 Helm Chart Name 有本质的区别。Release Name 是在安装时指定的，Helm Chart Name 在定义阶段就已经固定了。

注意，命令运行完成后，只能代表 Helm 已经将 Manifest 应用到了集群内，并不能表示应用已经就绪了。接下来 Kubernetes 集群会完成拉取镜像和 Pod 调度的操作，这些都是异步的。

使用模板变量

不过，刚才改造的最简单的 Helm Chart 并不能满足我们的目标。到目前为止，它只是一个纯静态的应用，无法应对多环境对配置差异的需求。



要将这个静态的 Helm Chart 改造成参数动态可控制的，**我们需要用到模板变量和 values.yaml。**

还记得我之前提到的 values.yaml 的概念吗？模板变量的值都会引自这个文件。在这个例子中，根据我们对不同环境配置差异化的要求，我抽象了这几个可配置项：

- 镜像版本
- HPA CPU 平均使用率
- 是否启用集群内数据库
- 数据库连接地址、账号和密码

这些可配置项都需要从 values.yaml 文件中读取，所以，你需要将下面的内容复制到 helm/values.yaml 文件内。

 复制代码

```
1 frontend:
2   image: lyzhang1999/frontend
3   tag: latest
4   autoscaling:
5     averageUtilization: 90
6
7 backend:
8   image: lyzhang1999/backend
9   tag: latest
10  autoscaling:
11    averageUtilization: 90
12
13 database:
14   enabled: true
15   uri: pg-service
16   username: postgres
17   password: postgres
18
```

除了 values.yaml，我们还需要让 helm/templates 目录下的文件能够读取到 values.yaml 的内容，这就需要模板变量了。

举一个最简单的例子，要读取 `values.yaml` 中的 `frontend.image` 字段，可以通过 `"{{ .Values.frontend.image }}"` 模板变量来获取值。



所以，要将这个“静态”的 Helm Chart 改造成“动态”的，我们只需要用模板变量来替换 **templates** 目录下需要实现“动态”的字段。

了解原理后，我们来修改 `helm/templates/backend.yaml` 文件，用模板替换需要从 `values.yaml` 读取的字段。

复制代码

```
1 apiVersion: apps/v1
2 kind: Deployment
3 metadata:
4   name: backend
5   .....
6 spec:
7   .....
8   spec:
9     containers:
10    - name: flask-backend
11      image: "{{ .Values.backend.image }}:{{ .Values.backend.tag }}"
12      env:
13        - name: DATABASE_URI
14          value: "{{ .Values.database.uri }}"
15        - name: DATABASE_USERNAME
16          value: "{{ .Values.database.username }}"
17        - name: DATABASE_PASSWORD
18          value: "{{ .Values.database.password }}"
```

同理，修改 `helm/templates/frontend.yaml` 文件的 `image` 字段。

复制代码

```
1 apiVersion: apps/v1
2 kind: Deployment
3 metadata:
4   name: frontend
5   .....
6 spec:
7   .....
8   spec:
9     containers:
10    - name: react-frontend
11      image: "{{ .Values.frontend.image }}:{{ .Values.frontend.tag }}"
```


此外，还需要修改 `helm/templates/hpa.yaml` 文件的 `averageUtilization` 字段。



```
1 .....
2 metadata:
3   name: frontend
4 spec:
5   .....
6   metrics:
7   - type: Resource
8     resource:
9       name: cpu
10      target:
11        type: Utilization
12        averageUtilization: {{ .Values.frontend.autoscaling.averageUtilization
13 ---
14 .....
15 metadata:
16   name: backend
17 spec:
18   .....
19   metrics:
20   - type: Resource
21     resource:
22       name: cpu
23       target:
24         type: Utilization
25         averageUtilization: {{ .Values.backend.autoscaling.averageUtilization }}
```

注意，相比较其他的模板变量，在这里我们没有在模板变量的外部使用双引号，这是因为 `averageUtilization` 字段是一个 `integer` 类型，而双引号加上模板变量的意思是 `string` 类型。

最后，我们希望使用 `values.yaml` 中的 `database.enable` 字段来控制是否向集群提交 `helm/templates/database.yaml` 文件。所以我们可以文件首行和最后一行增加下面的内容。

 复制代码

```
1 {{- if .Values.database.enabled -}}
2 .....
3 {{- end }}
```

到这里，我们就成功地将“静态”的 Helm Chart 改造为了“动态”的 Helm Chart。

部署

在将示例应用改造成 Helm Chart 之后，我们就可以使用 `helm install` 进行安装了。这里我会将 Helm Chart 分别安装到 `helm-staging` 和 `helm-prod` 命名空间，它们对应预发布环境和生产环境，接着我会介绍如何为不同的环境传递不同的参数。

部署预发布环境

我们为 Helm Chart 创建的 `values.yaml` 实际上是默认值，在预发布环境，我们希望将前后端的 HPA CPU `averageUtilization` 从默认的 90 调整为 60，你可以在安装时使用 `--set` 来调整特定的字段，不需要修改 `values.yaml` 文件。

 复制代码

```
1 $ helm install my-kubernetes-example ./helm --namespace helm-staging --create-namespace
2 NAME: my-kubernetes-example
3 LAST DEPLOYED: Thu Oct 20 18:13:34 2022
4 NAMESPACE: helm-staging
5 STATUS: deployed
6 REVISION: 1
7 TEST SUITE: None
```

在这个安装例子中，我们使用 `--set` 参数来调整 `frontend.autoscaling.averageUtilization` 字段值，其它的字段值仍然采用 `values.yaml` 提供的默认值。

部署完成后，你可以查看我们为预发布环境配置的后端服务 HPA `averageUtilization` 字段值。

 复制代码

```
1 $ kubectl get hpa backend -n helm-staging --output jsonpath='{.spec.metrics[0].averageUtilization}'
2 60
```

返回值为 60，和我们配置的安装参数一致。

同时，你也可以查看是否部署了数据库，也就是 `postgres` 工作负载。

 复制代码

```
1 $ kubectl get deployment postgres -n helm-staging
2 NAME          READY   UP-TO-DATE   AVAILABLE   AGE
3 postgres      1/1     1             1           46s
```

postgres 工作负载存在，符合预期。



最后，你可以查看 backend Deployment 的 Env 环境变量，以便检查是否使用集群内的数据库实例。

 复制代码

```
1 $ kubectl get deployment backend -n helm-staging --output jsonpath='{.spec.template.spec.containers[0].env[0].value}'
2
3 {"name":"DATABASE_URI","value":"pg-service"} {"name":"DATABASE_USERNAME","value":"postgres"}
```

返回结果同样符合预期。

部署生产环境

部署到生产环境的例子相对来说配置项会更多，除了需要修改 `database.enable` 字段，禁用集群内数据库实例以外，还需要修改数据库连接的三个环境变量。所以，我们使用另一种安装参数传递方式：**使用文件传递**。

要使用文件来传递安装参数，首先需要准备这个文件。你需要将下面的内容保存为 `helm/values-prod.yaml` 文件。

 复制代码

```
1 frontend:
2   autoscaling:
3     averageUtilization: 50
4
5 backend:
6   autoscaling:
7     averageUtilization: 50
8
9 database:
10  enabled: false
11  uri: 10.10.10.10
12  username: external_postgres
13  password: external_postgres
```

在 `values-prod.yaml` 文件内，我们只需要提供覆写的 Key 而不需要原样复制默认的 `values.yaml` 文件内容，这个操作和 Kustomize 的 Patch 操作有一点类似。



接下来，我们使用 `helm install` 命令来安装它，同时指定新的 `values-prod.yaml` 文件作为安装参数。

复制代码

```
1 $ helm install my-kubernetes-example ./helm -f ./helm/values-prod.yaml --namesp
2 NAME: my-kubernetes-example
3 LAST DEPLOYED: Thu Oct 20 20:21:07 2022
4 NAMESPACE: helm-prod
5 STATUS: deployed
6 REVISION: 1
7 TEST SUITE: None
```

部署完成后，你可以查看我们为生产环境配置的后端服务 HPA `averageUtilization` 字段值。

复制代码

```
1 $ kubectl get hpa backend -n helm-staging --output jsonpath='{.spec.metrics[0].
2 50
```

返回值为 50，和我们在 `values-prod.yaml` 文件中定义的安装参数一致。

同时，你也可以查看是否部署了数据库，也就是 `postgres` 工作负载。

复制代码

```
1 $ kubectl get deployment postgres -n helm-prod
2 Error from server (NotFound): deployments.apps "postgres" not found
```

可以发现，`postgres` 工作负载并不存在，符合预期。

最后，你可以查看 `backend Deployment` 的 `Env` 环境变量，检查是否使用了外部数据库。

复制代码

```
1 $ kubectl get deployment backend -n helm-prod --output jsonpath='{.spec.templat
2
3 {"name":"DATABASE_URI","value":"10.10.10.10"} {"name":"DATABASE_USERNAME","valu
```

返回结果同样符合预期。

到这里，将实例应用改造成 Helm Chart 的工作已经全部完成了。

发布 Helm Chart

在 Helm Chart 编写完成之后，我们便能够在本地安装它了。不过，我们通常还会有和其他人分享 Helm Chart 的需求。

还记得我们在 [第 19 讲](#) 提到的安装 Cert-manager 的例子吗？我们首先执行 `helm repo add` 命令添加了一个 Helm 仓库，然后使用 `helm install` 直接安装了一个远端仓库的 Helm Chart，这种方式和我们上面介绍的指定 Helm Chart 目录的安装方式并不相同。

那么，我们如何实现和 Cert-manager 相同的安装方式呢？

很简单，只需要将我们在上面创建的 Helm Chart 打包并且上传到 Helm 仓库中即可。

这里我以 GitHub Package 为例，介绍如何将 Helm Chart 上传到镜像仓库。

创建 GitHub Token


要将 Helm Chart 推送到 GitHub Package，首先我们需要创建一个具备推送权限的 Token，你可以在 [这个链接](#) 创建，并勾选 `write:packages` 权限。

Select scopes

Scopes define the access for personal tokens. [Read more about OAuth scopes.](#)

<input checked="" type="checkbox"/> repo	Full control of private repositories
<input checked="" type="checkbox"/> repo:status	Access commit status
<input checked="" type="checkbox"/> repo_deployment	Access deployment status
<input checked="" type="checkbox"/> public_repo	Access public repositories
<input checked="" type="checkbox"/> repo:invite	Access repository invitations
<input checked="" type="checkbox"/> security_events	Read and write security events
<input type="checkbox"/> workflow	Update GitHub Action workflows
<input checked="" type="checkbox"/> write:packages	Upload packages to GitHub Package Registry
<input checked="" type="checkbox"/> read:packages	Download packages from GitHub Package Registry

点击“Generate token”按钮生成 Token 并复制。


<https://shikey.com/>

Personal access tokens (classic)

Generate new token ▼ Revoke all

Tokens you have generated that can be used to access the [GitHub API](#).

Make sure to copy your personal access token now. You won't be able to see it again!

✓ ghp_lMUjgEpFWF3hUaUjKUMc [redacted]  Delete

推送 Helm Chart

在推送之前，还需要使用 GitHub ID 和刚才创建的 Token 登录到 GitHub Package。

复制代码

```
1 $ helm registry login -u lyzhang1999 https://ghcr.io
2 Password: token here
3 Login Succeeded
```

请注意，由于 GitHub Package 使用的是 OCI 标准的存储格式，如果你使用的 helm 版本小于 3.8.0，则需要在运行这条命令之前增加 HELM_EXPERIMENTAL_OCI=1 的环境变量启用实验性功能。

然后，返回到示例应用的根目录下，执行 helm package 命令来打包 Helm Chart。

复制代码

```
1 $ helm package ./helm
2 Successfully packaged chart and saved it to: /Users/weiwang/Downloads/kubernetes-
```

这条命令会将 helm 目录打包，并生成 kubernetes-example-0.1.0.tgz 文件。

接下来，就可以使用 helm push 命令推送到 GitHub Package 了。

复制代码

```
1 $ helm push kubernetes-example-0.1.0.tgz oci://ghcr.io/lyzhang1999/helm
```



```
2 Pushed: ghcr.io/lyzhang1999/helm/kubernetes-example:0.1.0
3 Digest: sha256:8a0cc4a2ac00f5b1f7a50d6746d54a2ecc96df6fd419a70614fe2b9b975c4f42
4
```



命令运行结束后将展示 Digest 字段，就说明 Helm Chart 推送成功了。

安装远端仓库的 Helm Chart

当我们成功把 Helm Chart 推送到 GitHub Package 之后，就可以直接使用远端仓库来安装 Helm Chart 了。和一般的安装步骤不同的是，由于 GitHub Package 仓库使用的是 OCI 标准的存储方式，所以无需执行 `helm repo add` 命令添加仓库，可以直接使用 `helm install` 命令来安装。

 复制代码

```
1 $ helm install my-kubernetes-example oci://ghcr.io/lyzhang1999/helm/kubernetes-
2
3 Pulled: ghcr.io/lyzhang1999/helm/kubernetes-example:0.1.0
4 Digest: sha256:8a0cc4a2ac00f5b1f7a50d6746d54a2ecc96df6fd419a70614fe2b9b975c4f42
5
6 NAME: my-kubernetes-example
7 LAST DEPLOYED: Thu Oct 20 21:38:41 2022
8 NAMESPACE: remote-helm-staging
9 STATUS: deployed
10 REVISION: 1
11 TEST SUITE: None
```

在上面的安装命令中，`oci://ghcr.io/lyzhang1999/helm/kubernetes-example` 是 Helm Chart 的完整的地址，并标识了 OCI 关键字。

另外，`version` 字段指定的是 Helm Chart 的版本号。在安装时，同样可以使用 `--set` 或者指定 `-f` 参数来覆写 `values.yaml` 的字段。

Helm 应用管理

通过上面内容的学习，我相信你已经掌握了如何使用 Helm 来定义应用，在实际的工作中，这些知识也基本上够用了。当然，如果你希望深度使用 Helm，那么你需要继续了解 Helm 应用管理功能和相关命令。

总结来说，Helm Chart 和 Manifest 之间一个最大的区别是，Helm 从应用的角度出发，提供了应用的管理功能，通常我们在实际使用 Helm 过程中会经常遇到下面几种场景。



- 调试 Helm Chart。
- 查看已安装的 Helm Release。
- 更新 Helm Release。
- 查看 Helm Release 历史版本。
- 回滚 Helm Release。
- 卸载 Helm Release。

接下来我们来看如何使用 Helm 命令行工具来实现这些操作。

调试 Helm Chart

在编写 Helm Chart 的过程中，为了方便验证，我们会经常渲染完整的 Helm 模板而又不安装它，这时候你就可以使用 `helm template` 命令来调试 Helm Chart。

复制代码

```
1 $ helm template ./helm -f ./helm/values-prod.yaml
2 ---
3 # Source: kubernetes-example/templates/backend.yaml
4 apiVersion: v1
5 kind: Service
6 .....
7 ---
8 # Source: kubernetes-example/templates/frontend.yaml
9 apiVersion: v1
10 kind: Service
11 .....
```

此外，你还可以在运行 `helm install` 命令时增加 `--dry-run` 参数来实现同样的效果。

复制代码

```
1 $ helm install my-kubernetes-example oci://ghcr.io/lyzhang1999/helm/kubernetes-
2
3 Pulled: ghcr.io/lyzhang1999/helm/kubernetes-example:0.1.0
4 Digest: sha256:8a0cc4a2ac00f5b1f7a50d6746d54a2ecc96df6fd419a70614fe2b9b975c4f42
5 NAME: my-kubernetes-example
```

```
6 .....
7 ---
8 # Source: kubernetes-example/templates/database.yaml
9 .....
```



查看已安装的 Helm Release

要查看已安装的 Helm Release，可以使用 `helm list` 命令。

 复制代码

```
1 $ helm list -A
2 NAME                                NAMESPACE          REVISION    UPDATED
3 my-kubernetes-example             helm-prod           1           2022-10-20 20:2
4 my-kubernetes-example             helm-staging        1           2022-10-20 20:2
5 my-kubernetes-example             remote-helm-staging 1           2022-10-20 21:3
6
```

返回结果中展示了我们刚才安装的所有 Helm Release 以及它们所在的命名空间。

更新 Helm Release

要更新 Helm Release，可以使用 `helm upgrade` 命令，Helm 会自动对比新老版本之间的 Manifest 差异，并执行升级。

 复制代码

```
1 $ helm upgrade my-kubernetes-example ./helm -n example
2 Release "my-kubernetes-example" has been upgraded. Happy Helming!
3 NAME: my-kubernetes-example
4 LAST DEPLOYED: Thu Oct 20 16:31:25 2022
5 NAMESPACE: example
6 STATUS: deployed
7 REVISION: 2
8 TEST SUITE: None
```

查看 Helm Release 历史版本

要查看 Helm Release 的历史版本，你可以使用 `helm history` 命令。

 复制代码

```
1 $ helm history my-kuebrnetes-example -n example
```

2	REVISION	UPDATED	STATUS	CHART
3	1	Thu Oct 20 16:09:22 2022	superseded	kubernetes-exar
4	2	Thu Oct 20 16:31:25 2022	deployed	kubernetes-exar



天下无鱼

<https://shikey.com/>

从返回结果来看，my-kuebrnetes-example Release 有两个版本（REVISION），分别是 1 和 2，我们还能看到每一个版本的状态。

回滚 Helm Release

当 Helm Release 有多个版本时，你可以通过 `helm rollback` 命令回滚到指定的版本。

 复制代码

```
1 $ helm rollback my-kubernetes-example 1 -n example
2 Rollback was a success! Happy Helming!
```

卸载 Helm Release

最后，要卸载 Helm Release，你可以使用 `helm uninstall`。

 复制代码

```
1 $ helm uninstall my-kubernetes-example -n example
2 release "my-kubernetes-example" uninstalled
```

总结

这节课，我们以示例应用为例子，介绍了如何使用 Helm Chart 来定义应用。

Helm Chart 实际上是由特定的文件和目录组成的，一个最简单的 Helm Chart 包含 Chart.yaml、values.yaml 和 templates 目录，当我们把这个特定的目录打包为 `tgz` 压缩文件时，实际上它也就是标准的 Helm Chart 格式。

相比较 Kustomize 和原生 Manifest，Helm Chart 更多是从“应用”的视角出发的，它为 Kubernetes 应用提供了打包、存储、发行和启动的能力，实际上它就是一个 Kubernetes 的应用包管理工具。此外，Helm 通过模板语言为我们提供了暴露应用关键参数的能力，使用者只需要关注安装参数而不需要去理解内部细节。

而 Kustomize 和 Manifest 则使用原生的 YAML 和 Kubernetes API 进行交互，不具备包管理的概念，所以在这方面它们之间有着本质的区别。



那么，如果把 Kubernetes 比作是操作系统，Helm Chart 其实就可以类比为 Windows 的应用安装包，它们都是应用的一种安装方式。

在 Helm 的具体使用方面，我还介绍了如何通过 `helm install` 命令来安装两种类型的 Helm Chart，它们分别是本地目录和远端仓库。在安装时，它们都可以使用 `--set` 参数来对默认值覆写，也可以使用 `-f` 参数来指定新的 `values.yaml` 文件。此外，我还以 GitHub Package 为例子，介绍了如何打包 Helm Chart 并上传到 GitHub Package 仓库中。

最后，在 Helm 应用管理方面，我希望你能够熟记几条简单的命令，例如 `helm list`、`helm upgrade` 和 `helm rollback` 等，这些命令在工作中都是很常用的。

到这里，我们对应用定义的讲解就全部结束了，希望你能有所收获。

思考题

最后，给你留两道思考题吧。

1. 请你结合 [第 16 讲](#) 的内容，为示例应用配置 GitHub Action，要求是每次提交代码后都自动打包 Helm Chart，并将它上传到 GitHub Package 中。你可以将 GitHub Action Workflow 的 YAML 内容放到留言中。
2. 如何实现在同一个命名空间下对同一个 Helm Chart 安装多个 Helm Release？以示例应用为例，请你分享核心的思路。

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

上一篇 20 | 应用定义：如何使用 Kustomize 定义应用？

下一篇 22 | 如何使用 ArgoCD 快速打造生产可用的 GitOps 工作流？

精选留言 (8)

 写留言



ghostwritten

2023-02-02 来自北京

我的 centos 7.9.2009 & helm v3.11.0 没有遇到拒绝问题。

```
$ helm version
```

```
version.BuildInfo{Version:"v3.11.0", GitCommit:"472c5736ab01133de504a826bd9ee12cbe4e7904", GitTreeState:"clean", GoVersion:"go1.18.10"}
```

```
$ helm registry login -u ghostwritten https://ghcr.io
```

```
Password:
```

```
Login Succeeded
```

```
$ helm package ./helm
```

```
Successfully packaged chart and saved it to: /root/github/kubernetes-example/kubernetes-example-0.1.0.tgz
```

```
$ helm push kubernetes-example-0.1.0.tgz oci://ghcr.io/ghostwritten/helm
```

```
Pushed: ghcr.io/ghostwritten/helm/kubernetes-example:0.1.0
```

```
Digest: sha256:46bef623e43f4525ebfd25c368dfea69e70efbe7590f1e3eccc321fbb6b16882
```

```
$ helm install my-kubernetes-example oci://ghcr.io/ghostwritten/helm/kubernetes-example --version 0.1.0 --namespace remote-helm-staging --create-namespace --set frontend.autoscaling.averageUtilization=60 --set backend.autoscaling.averageUtilization=60
```

```
Pulled: ghcr.io/ghostwritten/helm/kubernetes-example:0.1.0
```

```
Digest: sha256:46bef623e43f4525ebfd25c368dfea69e70efbe7590f1e3eccc321fbb6b16882
```

```
W0202 16:15:56.276585 3957 warnings.go:70] autoscaling/v2beta2 HorizontalPodAutoscaler is deprecated in v1.23+, unavailable in v1.26+; use autoscaling/v2 HorizontalPodAutoscaler
```

```
AME: my-kubernetes-example
```

```
LAST DEPLOYED: Thu Feb 2 16:15:55 2023
```

```
NAMESPACE: remote-helm-staging
```

```
STATUS: deployed
```

```
REVISION: 1
```


TEST SUITE: None

```
$ k get pods -n remote-helm-staging
```

NAME	READY	STATUS	RESTARTS	AGE
backend-bcb7687c6-s7lxh	1/1	Running	0	29m
backend-bcb7687c6-v4cx6	1/1	Running	0	29m
frontend-7c59d655fb-p6lpm	1/1	Running	1 (21m ago)	29m
frontend-7c59d655fb-xnl8x	1/1	Running	0	29m
postgres-7745b57d5d-2nndw	1/1	Running	0	29m



天下无鱼

<https://shikey.com/>



天地有雪

2023-01-29 来自广东

可以了，原来ubuntu系统，helm 3.11版本，换成centos7 helm 3.8.0 没有问题



天地有雪

2023-01-29 来自广东

helm version

```
version.BuildInfo{Version:"v3.11.0", GitCommit:"472c5736ab01133de504a826bd9ee12cbe4e7904", GitTreeState:"clean", GoVersion:"go1.18.10"}
```

1

```
helm registry login -u mlkkkfriend https://ghcr.io
```

Password:

Error: Get "https://ghcr.io/v2/": denied: denied

输入账号，密码无法登录

2 文章中提到 在推送之前，还需要使用 GitHub ID 和刚才创建的 Token 登录到 GitHub Package。需要什么操作

```
3 echo $CR_PAT | docker login ghcr.io -u 用户 --password-stdin
```

可以登录

docker push 可以上传

```
4 echo $CR_PAT | helm registry login -u mlkkkfriend https://ghcr.io --password-stdin
```

Login Succeeded

可以登录

```
helm push kubernetes-example-0.1.0.tgz oci://ghcr.io/mlkkkfriend
```

Error: failed commit on ref "manifest-sha256:3790edf4411c5d6fbf3e40548ebdf78979ab99f5"

d0206031d436805186f0ae20": unexpected status: 403 Forbidden

上传不了



天下无鱼

<https://shikey.com/>



天地有雪

2023-01-29 来自广东

helm version

```
version.BuildInfo{Version:"v3.11.0", GitCommit:"472c5736ab01133de504a826bd9ee12cbe4e7904", GitTreeState:"clean", GoVersion:"go1.18.10"}
```

1

```
helm registry login -u mlkkkfriend https://ghcr.io
```

Password:

```
Error: Get "https://ghcr.io/v2/": denied: denied
```

输入账号，密码无法登录

2 文章中提到 在推送之前，还需要使用 GitHub ID 和刚才创建的 Token 登录到 GitHub Package。需要什么操作

```
3 echo $CR_PAT | docker login ghcr.io -u 用户 --password-stdin
```

可以登录

docker push 可以上传



天地有雪

2023-01-29 来自广东

老师：请教一下



天地有雪

2023-01-29 来自广东

老师：

```
1 helm registry login -u mlkkkfriend https://ghcr.io
```

Password:

```
Error: Get "https://ghcr.io/v2/": denied: denied
```



天地有雪

2023-01-29 来自广东

你好：

1 helm registry login -u mlkkkfriend https://ghcr.io

Password:

Error: Get "https://ghcr.io/v2/": denied: denied

文章中 在推送之前，还需要使用 GitHub ID 和刚才创建的 Token 登录到 GitHub Package。
需要什么操作

共 1 条评论 >



Amos

2023-01-25 来自江苏

2、执行安装时使用不同的 helm release name，并且通过命令行参数或 values.yaml 的方式修改 deploy、service 等对象的名字

