Helm 简介和使用

AUTHOR: 彭玲 TIME: 2021/10/28

Helm 简介

Helm 是 Kubernetes 的包管理工具，用来简化 K8S 应用的部署和管理。

在 Kubernetes 中部署容器云应用（容器或微服务编排）是一项有挑战性的工作，Helm 就是为了简化在 Kubernetes 中安装部署容器云应用的一个客户端工具。通过 Helm 能够帮助开发者定义、安装和升级 Kubernetes 中的容器云应用，以及容器云应用的分享。

Helm 解决了什么痛点?

在 Kubernetes 中部署一个可以使用的应用，需要涉及到很多的 Kubernetes 资源的共同协作。

比如，安装一个 WordPress 博客，用到了一些 Kubernetes 的一些资源对象，包括 Deployment 用于`部署应用`、Service 提供`服务发现`、Secret 配置 WordPress 的用户名和密码，可能还需要 PV 和 PVC 来提供持久化服务。并且 WordPress 数据是存储在 mariadb 里面的，所以需要 mariadb 启动就绪后才能启动 WordPress。这些 k8s 资源过于分散，不方便进行管理，直接通过 kubectl 来管理一个应用，会发

现这十分繁琐。

通过 Helm 在 k8s 中部署一个应用，可以解决以下几个问题：

- 统一管理、配置和更新这些分散的 k8s 的应用资源文件。
- 分发和复用一套应用模板。
- 将应用的一系列资源当做一个软件包管理。

Helm 架构

使用 Helm 能够从 Chart repository（Helm 应用仓库）快速查找、下载安装软件包，并通过与 K8s API Server 交互构建应用。

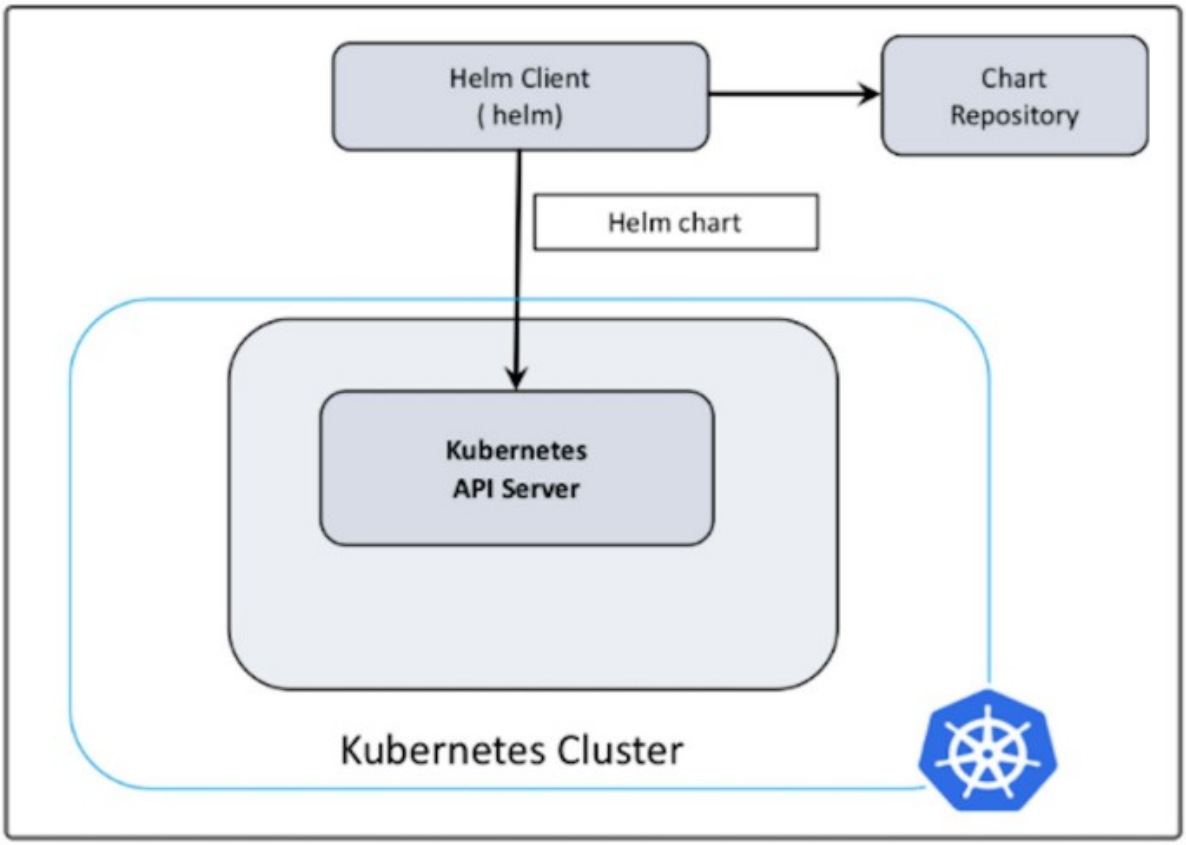

图1 Helm3 Workflow

Helm 重要概念

在 Helm 中，有三个需要了解的重要概念：Chart、Config、Release。

- Charts：Helm 使用的打包格式，一个 Chart 包含了一组 K8s 资源集合的描述文件。
- Config：创建发布对象的 Chart 的配置信息。
- Release：Chart 的运行实例，包含特定的 Config。

Helm 安装

Helm 可以用 `源码` 或构建的 `二进制版本` 安装。这里使用官方提供的 `脚本` 一键安装：

```
# 从 https://raw.githubusercontent.com/kubernetes/helm/master/scripts/get 获取脚本，并保存到 get_helm.sh
$ curl https://raw.githubusercontent.com/kubernetes/helm/master/scripts/get > get_helm.sh

$ chmod 700 get_helm.sh

$ ./get_helm.sh
```

Chart 文件结构

Helm 的打包格式叫做`chart`，所谓`chart`就是一系列文件，它描述了一组相关的 k8s 集群资源。

Chart 基本结构

使用`helm create my-nginx`创建一个 Chart。该 Chart 目录如下所示：

```
anxin@node38:~/pengling/k8s/helm-demo$ tree
.
└── my-nginx
    ├── charts # 存放依赖的 chart
    ├── Chart.yaml # 该 Chart 的描述（包含 Chart 的基本信息，包括 chart 版本，名称等）。
    ├── templates # 放置模板文件（存放应用一系列 k8s 资源的 yaml 模板），想要渲染什么文件出来，就在这个目录放置对应模板。
    |   ├── deployment.yaml
    |   ├── _helpers.tpl # 定义一些可重用的模板片断，此文件中的定义在任何资源定义模板中可用。
    |   ├── hpa.yaml
    |   ├── ingress.yaml
    |   ├── NOTES.txt # 介绍 chart 部署后的帮助信息，如何使用 chart 等。
    |   ├── serviceaccount.yaml
    |   ├── service.yaml
    |   └── tests
    |       └── test-connection.yaml
    └── values.yaml # 包含了必要的值定义（变量默认值），用于存储 templates 目录中模板文件中用到变量的值。

4 directories, 10 files
```

Subchart 依赖文件

使用`helm fetch bitnami/wordpress -untar`下载并解压后，查看`wordpress`chart 包的`Chart.yaml`。

其中`dependencies`属性下列出了依赖（Subchart），如`mariadb`等。

```
annotations:
  category: CMS
apiVersion: v2
appVersion: 5.8.1
dependencies: # 依赖 ( 子 Chart )
- condition: mariadb.enabled
```

```
  name: mariadb
  repository: https://charts.bitnami.com/bitnami
  version: 9.x.x
- condition: memcached.enabled
  name: memcached
  repository: https://charts.bitnami.com/bitnami
  version: 5.x.x
- name: common
  repository: https://charts.bitnami.com/bitnami
  tags:
  - bitnami-common
  version: 1.x.x
description: Web publishing platform for building blogs and websites.
home: https://github.com/bitnami/charts/tree/master/bitnami/wordpress
icon: https://bitnami.com/assets/stacks/wordpress/img/wordpress-stack-220x234.png
keywords:
- application
- blog
- cms
- http
- php
- web
- wordpress
maintainers:
- email: containers@bitnami.com
  name: Bitnami
name: wordpress
sources:
- https://github.com/bitnami/bitnami-docker-wordpress
- https://wordpress.org/
version: 12.1.21
```

Chart 模板

模板渲染是 Helm 的核心内容，主要依赖其模板语言（`template/`）和内置对象（`Chart`，`Template`，`Values` 等）。

创建一个 Nginx 的 Chart

使用 Helm 的 `Template` 功能，需要先创建一个 Chart，这是 Helm 的基本文件组成架构。

```
anxin@node38:~/pengling/k8s/helm-demo$ helm create my-nginx
Creating my-nginx
```

命令执行完成后，就会自动创建 Chart 的相关文件：

```
anxin@node38:~/pengling/k8s/helm-demo$ tree
.
└── my-nginx
    ├── charts # 存放依赖的 chart
    ├── Chart.yaml # 该 Chart 的描述（包含 Chart 的基本信息，包括 chart 版本，名称等）。
```

```
    ├── templates # 放置模板文件（存放应用一系列 k8s 资源的 yaml 模板），想要渲染什么文件出来，就在这个目录放置对应模板。
    |   ├── deployment.yaml
    |   ├── _helpers.tpl # 定义一些可重用的模板片断，此文件中的定义在任何资源定义模板中可用。
    |   ├── hpa.yaml
    |   ├── ingress.yaml
    |   ├── NOTES.txt # 介绍 chart 部署后的帮助信息，如何使用 chart 等。
    |   ├── serviceaccount.yaml
    |   ├── service.yaml
    |   └── tests
    |       └── test-connection.yaml
    └── values.yaml # 包含了必要的值定义（变量默认值），用于存储 templates 目录中模板文件中用到变量的值。

4 directories, 10 files
```

模板渲染

不修改模板、不添加变量，使用`helm template my-nginx/ --output-dir ./result`直接渲染出结果文件。

```
anxin@node38:~/pengling/k8s/helm-demo$ helm template my-nginx/ --output-dir ./result
wrote ./result/my-nginx/templates/serviceaccount.yaml
wrote ./result/my-nginx/templates/service.yaml
wrote ./result/my-nginx/templates/deployment.yaml

---
# Source: my-nginx/templates/tests/test-connection.yaml
apiVersion: v1
kind: Pod
metadata:
  name: "RELEASE-NAME-my-nginx-test-connection"
  labels:
    helm.sh/chart: my-nginx-0.1.0
    app.kubernetes.io/name: my-nginx
    app.kubernetes.io/instance: RELEASE-NAME
    app.kubernetes.io/version: "1.16.0"
    app.kubernetes.io/managed-by: Helm
  annotations:
    "helm.sh/hook": test-success
spec:
  containers:
    - name: wget
      image: busybox
      command: ['wget']
      args: ['RELEASE-NAME-my-nginx:80']
  restartPolicy: Never
```

渲染结果（`result/`）

根据一些变量和判断，helm 直接帮我们生成了三种资源（`ServiceAccount`,`Service`,`Deployment`）的文件。

```
anxin@node38:~/pengling/k8s/helm-demo$ tree result
result
└── my-nginx
    └── templates
        ├── deployment.yaml
        ├── serviceaccount.yaml
        └── service.yaml

2 directories, 3 files
```

deployment.yaml

```
anxin@node38:~/pengling/k8s/helm-demo$ cat result/my-nginx/templates/deployment.yaml
---
# Source: my-nginx/templates/deployment.yaml
apiVersion: apps/v1
kind: Deployment
metadata:
  name: RELEASE-NAME-my-nginx
  labels:
    helm.sh/chart: my-nginx-0.1.0
    app.kubernetes.io/name: my-nginx
    app.kubernetes.io/instance: RELEASE-NAME
    app.kubernetes.io/version: "1.16.0"
    app.kubernetes.io/managed-by: Helm
spec:
  replicas: 1
  selector:
    matchLabels:
      app.kubernetes.io/name: my-nginx
      app.kubernetes.io/instance: RELEASE-NAME
  template:
    metadata:
      labels:
        app.kubernetes.io/name: my-nginx
        app.kubernetes.io/instance: RELEASE-NAME
    spec:
      serviceAccountName: RELEASE-NAME-my-nginx
      securityContext:
        {}
      containers:
        - name: my-nginx
          securityContext:
            {}
          image: "nginx:1.16.0"
          imagePullPolicy: IfNotPresent
          ports:
            - name: http
              containerPort: 80
              protocol: TCP
```

```
          livenessProbe:
            httpGet:
              path: /
              port: http
          readinessProbe:
            httpGet:
              path: /
              port: http
          resources:
            {}
```

service.yaml

查看其中一个文件`service.yaml`，还是非常完整的，基本可以满足需要了，再根据自己的需求改改就好了。

```
anxin@node38:~/pengling/k8s/helm-demo$ cat result/my-nginx/templates/service.yaml
---
# Source: my-nginx/templates/service.yaml
apiVersion: v1
kind: Service
metadata:
  name: RELEASE-NAME-my-nginx
  labels:
    helm.sh/chart: my-nginx-0.1.0
    app.kubernetes.io/name: my-nginx
    app.kubernetes.io/instance: RELEASE-NAME
    app.kubernetes.io/version: "1.16.0"
    app.kubernetes.io/managed-by: Helm
spec:
  type: ClusterIP
  ports:
    - port: 80
      targetPort: http
      protocol: TCP
      name: http
  selector:
    app.kubernetes.io/name: my-nginx
    app.kubernetes.io/instance: RELEASE-NAME
```

serviceaccount.yaml

```
anxin@node38:~/pengling/k8s/helm-demo$ cat result/my-
nginx/templates/serviceaccount.yaml
---
# Source: my-nginx/templates/serviceaccount.yaml
apiVersion: v1
kind: ServiceAccount
metadata:
  name: RELEASE-NAME-my-nginx
  labels:
    helm.sh/chart: my-nginx-0.1.0
    app.kubernetes.io/name: my-nginx
    app.kubernetes.io/instance: RELEASE-NAME
    app.kubernetes.io/version: "1.16.0"
    app.kubernetes.io/managed-by: Helm
```

添加模板文件

ConfigMap 模板

试着添加一个模板文件`configmap.yaml`到`templates`目录（`my-nginx/templates/`），内容如下：

```
anxin@node38:~/pengling/k8s/helm-demo/my-nginx/templates$ vi configmap.yaml

apiVersion: v1
kind: ConfigMap
metadata:
  name: julin-file
  namespace: default
data:
  application.yaml: |-
    server:
      port: 8080
    julin:
      name: Julin
      age: 18
      webSite: www.julin.com
```

渲染结果

执行`helm template my-nginx/ --output-dir ./result`命令后渲染的结果如下：

```
anxin@node38:~/pengling/k8s/helm-demo$ tree result
result
└── my-nginx
    └── templates
        ├── configmap.yaml
        ├── deployment.yaml
        ├── serviceaccount.yaml
        └── service.yaml

2 directories, 4 files
```

结果中`configmap.yaml`与模板并没有什么不同，因为我们没有在`模板文件`里使用`变量`和`判断语句`等。

```
# ConfigMap 模板渲染结果
```

```
anxin@node38:~/pengling/k8s/helm-demo$ cat result/my-
nginx/templates/configmap.yaml
---
# Source: my-nginx/templates/configmap.yaml
apiVersion: v1
kind: ConfigMap
metadata:
  name: julin-file
  namespace: default
data:
  application.yaml: |-
    server:
      port: 8080
    julin:
      name: Julin
      age: 18
      webSite: www.julin.com
```

模板中使用变量

修改 ConfigMap 模板如下：

```
anxin@node38:~/pengling/k8s/helm-demo$ vi my-nginx/templates/configmap.yaml

apiVersion: v1
kind: ConfigMap
metadata:
  name: julin-config-{{ .Values.environment }}
  namespace: default
data:
  application.yaml: |-
    server:
      port: {{ .Values.server.port }}
    julin:
      name: {{ .Values.julin.name }}
      age: {{ .Values.julin.age }}
    {{- if .Values.julin.webSite }}
      webSite: {{ .Values.julin.webSite }}
    {{- end }}
```

可以看见在模板中使用了许多双大括号的变量`{{ .Values.xxx }}`，同时需要在`values.yaml`文件中定义这些变量。

定义变量

在`values.yaml`文件中定义变量：

```
anxin@node38:~/pengling/k8s/helm-demo$ vi my-nginx/values.yaml

... # 原 values.yaml 内容

# 自定义变量
environment: dev
server:
  port: 80
julin:
  name: Julin Peng
  age: 28
  webSite: https://www.julin.com
```

重新渲染

重新执行命令`helm template my-nginx/ -output-dir ./result`，渲染的结果如下：

```
anxin@node38:~/pengling/k8s/helm-demo$ cat result/my-
nginx/templates/configmap.yaml
---
# Source: my-nginx/templates/configmap.yaml
apiVersion: v1
kind: ConfigMap
metadata:
  name: julin-config-dev
  namespace: default
data:
  application.yaml: |-
    server:
      port: 80
    julin:
      name: Julin Peng
      age: 28
      webSite: https://www.julin.com
```

不同环境变量设置

多环境管理在 Helm Template 这也是非常简单的，我们创建一个`values-dev.yaml`的变量文件，内容如下：

```
anxin@node38:~/pengling/k8s/helm-demo/my-nginx$ vi values-dev.yaml

environment: dev
server:
  port: 8080
julin:
  name: Julin Peng
  age: 1
```

通过以下命令来指定`dev`环境的变量文件。这样渲染出来的结果就是`dev`的相关配置了。其它环境同理。

```
# 指定 dev 环境的变量文件(file)：-f my-nginx/values-dev.yaml
```

```
anxin@node38:~/pengling/k8s/helm-demo$ helm template my-nginx/ --output-dir ./result -f my-nginx/values-dev.yaml
wrote ./result/my-nginx/templates/serviceaccount.yaml
wrote ./result/my-nginx/templates/configmap.yaml
wrote ./result/my-nginx/templates/service.yaml
wrote ./result/my-nginx/templates/deployment.yaml

---
# Source: my-nginx/templates/tests/test-connection.yaml
apiVersion: v1
kind: Pod
metadata:
  name: "RELEASE-NAME-my-nginx-test-connection"
  labels:
    helm.sh/chart: my-nginx-0.1.0
    app.kubernetes.io/name: my-nginx
    app.kubernetes.io/instance: RELEASE-NAME
    app.kubernetes.io/version: "1.16.0"
    app.kubernetes.io/managed-by: Helm
  annotations:
    "helm.sh/hook": test-success
spec:
  containers:
    - name: wget
      image: busybox
      command: ['wget']
      args: ['RELEASE-NAME-my-nginx:80']
  restartPolicy: Never
```

通过命令行设置变量

使用`--set`或`--set-string`，使用如下：

```
# 设置 julin.webSite=https://www.pengling.com (注意：webSite 大小写敏感。)
anxin@node38:~/pengling/k8s/helm-demo$ helm template my-nginx/ --output-dir ./result -f my-nginx/values-dev.yaml --set julin.webSite=https://www.pengling.com
wrote ./result/my-nginx/templates/serviceaccount.yaml
wrote ./result/my-nginx/templates/configmap.yaml
wrote ./result/my-nginx/templates/service.yaml
wrote ./result/my-nginx/templates/deployment.yaml
...
```

查看输出结果：`webSite: https://www.pengling.com`。

```
anxin@node38:~/pengling/k8s/helm-demo$ cat result/my-nginx/templates/configmap.yaml
---
# Source: my-nginx/templates/configmap.yaml
apiVersion: v1
kind: ConfigMap
metadata:
  name: julin-config-dev
  namespace: default
data:
  application.yaml: |-
```

```
  server:
    port: 8080
  julin:
    name: Julin Peng
    age: 1
    webSite: https://www.pengling.com
```

Helm vs. kubebuilder 总结

通过以上 Helm 的介绍和实践，结合我司业务场景，可知：Helm 可用于具有依赖关系的应用管理（如 web 需要 webapi、萤石代理等服务的支持），但存在一个比较明显的问题：values 配置项较为繁琐，尤其是项目依赖较多时尤为突出，而且发行 Chart 后，如果需要修改配置，需要本地下载并修改配置，然后本地进行测试，应用管理极为不便。

Helm 更适用于对项目的快速安装体验，而非应用。

相比而言，kubebuilder 对于项目依赖及参数配置管理更趋向于应用的定制化 (模板化)，通过 CRD 的 yaml 文件配置项目应用即可。