helm

helm 介绍

Helm 是什么

Helm 是 Kubernetes 的包管理器。包管理器类似于我们在 Ubuntu 中使用的apt、Centos中使用的yum 或者 Python中的 pip 一样，能快速查找、下载和安装软件包。Helm 由客户端组件 helm 和服务端组件 Tiller 组成, 能够将一组K8S资源打包统一管理, 是查找、共享和使用为Kubernetes构建的软件的最佳方式。

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。比如你安装一个 WordPress 博客，用到了一些 Kubernetes (下面全部简称k8s)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，你会发现这十分蛋疼。
所以总结以上，我们在 k8s 中部署一个应用，通常面临以下几个问题：

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

Helm 相关组件及概念

Helm 包含两个组件，分别是 helm 客户端 和 Tiller 服务器：

- **helm** 是一个命令行工具，用于本地开发及管理chart，chart仓库管理等
- **Tiller** 是 Helm 的服务端。Tiller 负责接收 Helm 的请求，与 k8s 的 apiserver 交互，根据chart 来生成一个 release 并管理 release
- **chart** 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
- **release** 使用 helm install 命令在 Kubernetes 集群中部署的 Chart 称为 Release
- Repoistory Helm chart 的仓库，Helm 客户端通过 HTTP 协议来访问存储库中 chart 的索引文件和压缩包

Helm 原理

下面两张图描述了 Helm 的几个关键组件 Helm（客户端）、Tiller（服务器）、Repository（Chart 软件仓库）、Chart（软件包）之间的关系以及它们之间如何通信

helm 组件通信

helm 架构

创建release

- helm 客户端从指定的目录或本地tar文件或远程repo仓库解析出chart的结构信息
- helm 客户端指定的 chart 结构和 values 信息通过 gRPC 传递给 Tiller
- Tiller 服务端根据 chart 和 values 生成一个 release
- Tiller 将install release请求直接传递给 kube-apiserver

删除release

- helm 客户端从指定的目录或本地tar文件或远程repo仓库解析出chart的结构信息
- helm 客户端指定的 chart 结构和 values 信息通过 gRPC 传递给 Tiller
- Tiller 服务端根据 chart 和 values 生成一个 release
- Tiller 将delete release请求直接传递给 kube-apiserver

更新release

- helm 客户端将需要更新的 chart 的 release 名称 chart 结构和 value 信息传给 Tiller
- Tiller 将收到的信息生成新的 release，并同时更新这个 release 的 history
- Tiller 将新的 release 传递给 kube-apiserver 进行更新

helm 安装

文档：https://helm.sh/docs/intro/install/

master01

```
wget https://get.helm.sh/helm-v3.1.2-linux-amd64.tar.gz
tar -zxvf helm-v3.1.2-linux-amd64.tar.gz

cp -rp helm /usr/local/bin/
helm version

# 命令
helm -help
```

添加一个仓库到helm

https://github.com/bitnami/charts

```
# add repo into helm
helm repo add bitnami https://charts.bitnami.com/bitnami

helm repo add ali-stable    https://kubernetes.oss-cn-hangzhou.aliyuncs.com/charts

# 查看repo 列表
helm repo list

# search
helm search repo harbor
```

```
# pull
helm pull bitnami/harbor
cd harbor && ls
```

helm 目录层级

```
mkdir -p /root/install-some-apps

# 创建chart
helm create helm=test
cd helm-test && ls
tree
```

```
# 目录说明
helm=test
├── charts # 依赖文件
├── Chart.yaml # 这个chart的版本信息
├── README.md
├── requirements.lock
├── requirements.yaml
├── templates # 模板
│   ├── deployment.yaml
│   ├── externaldb-secrets.yaml
│   ├── _helpers.tpl  # 自定义的模板
│   ├── ingress.yaml
│   ├── NOTES.txt # 这个 chart的信息
│   ├── pvc.yaml
│   ├── secrets.yaml
│   ├── svc.yaml
│   └── tls-secrets.yaml
└── values.yaml #配置全局变量或者一些参数
```

helm 语法

Helm templates 中的语法

官方文档：https://helm.sh/docs/chart_template_guide/function_list/

常用函数：http://masterminds.github.io/sprig/strings.html

_helpers.tpl

在chart中以 “下划线” 开头的文件，称为”子模版”。
例如在 _helper.tpl 中定义子模块，格式：{{- define "模版名字" -}} 模版内容 {{- end -}}

```
{{- define "nginx.name" -}}
{{- default .Chart.Name .Values.nameOverride | trunc 63 | trimSuffix "-" -}}
{{- end -}}

# 若 .Values.nameOverride 为空，则默认值为 .Chart.Name
```

引用模板，格式：{{ include "模版名字" 作用域}}

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: {{ include "nginx.fullname" . }}
```

内置对象

Build-in Objects: https://helm.sh/docs/chart_template_guide/builtin_objects/

Chart 预定义对象可直接在各模板中使用。

```
Release:       代表Release对象，属性包含：Release.Name、Release.Namespace、Release.Revision
等
Values:        表示 values.yaml 文件数据
Chart:         表示 Chart.yaml 数据
Files:         用于访问 chart 中非标准文件

Capabilities:  用于获取 k8s 集群的一些信息
   - Capabilities.KubeVersion.Major：K8s的主版本

Template:      表示当前被执行的模板
   - Name：表示模板名，如：mychart/templates/mytemplate.yaml
   - BasePath：表示路径，如：mychart/templates
```

变量

默认情况点(.), 代表全局作用域，用于引用全局对象。
helm 全局作用域中有两个重要的全局对象：Values 和 Release

```
# Values
# 这里引用了全局作用域下的Values对象中的key属性。
{{ .Values.key }}
Values代表的就是values.yaml定义的参数，通过.Values可以引用任意参数。
例子：
```

```
{{ .Values.replicaCount }}

# 引用嵌套对象例子，跟引用json嵌套对象类似
{{ .Values.image.repository }}

# Release
其代表一次应用发布，下面是Release对象包含的属性字段：
Release.Name       - release的名字，一般通过Chart.yaml定义，或者通过helm命令在安装应用的时候指定。
Release.Time       - release安装时间
Release.Namespace  - k8s名字空间
Release.Revision   - release版本号，是一个递增值，每次更新都会加一
Release.IsUpgrade  - true代表，当前release是一次更新.
Release.IsInstall  - true代表，当前release是一次安装
Release.Service:   - The service that is rendering the present template. On Helm, this is always Helm.
```

自定义模版变量。

```
# 变量名以$开始命名， 赋值运算符是 := (冒号+等号)
{{- $relname := .Release.Name -}}

引用自定义变量:
#不需要 . 引用
{{ $relname }}
```

include

```
include 是一个函数，所以他的输出结果是可以传给其他函数的

# 例子1:
env:
  {{- include "xiaomage" . }}

# 结果:
          env:
- name: name
  value: xiaomage
- name: age
  value: secret
- name: favourite
  value: "Cloud Native DevSecOps"
- name: wechat
  value: majinghe11

# 例子2:
env:
  {{- include "xiaomage" . | indent 8}}
```

```
# 结果:
          env:
            - name: name
              value: xiaomage
            - name: age
              value: secret
            - name: favourite
              value: "Cloud Native DevSecOps"
            - name: wechat
              value: majinghe11
```

with

with 关键字可以控制变量的作用域,主要就是用来修改 . 作用域的，默认 . 代表全局作用域，with 语句可以修改 . 的含义

```
# 例子:
# .Values.favorite 是一个 object 类型
{{- with .Values.favorite }}
drink: {{ .drink | default "tea" | quote }}   # 相当于.Values.favorite.drink
food:  {{ .food  | upper | quote }}
{{- end }}
```

toYaml 转 yaml

将数据转为yaml格式

```
spec:
  strategy:
{{ toYaml .Values.strategy | indent 4 }}

----------------------------------------------------------------
values.yaml数据:
strategy:
  type: RollingUpdate
  rollingUpdate:
    maxUnavailable: 0

----------------------------------------------------------------
渲染效果:
spec:
  strategy:
    type: RollingUpdate
    rollingUpdate:
      maxUnavailable: 0
```

Values 对象

values 对象的值有四个来源

1. chart 包中的 values.yaml 文件
2. 父 chart 包的 values.yaml 文件
3. 使用 helm install 或者 helm upgrade 的 -f 或者 --values 参数传入的自定义的 yaml 文件
4. 通过 --set 参数传入的值

```
cat global.yaml
course: k8s

cat mychart/templates/configmap.yaml
apiVersion: v1
kind: ConfigMap
metadata:
  name: {{ .Release.Name }}-configmap
data:
  myvalue: "Hello World"
  course:  {{ .Values.course }}

helm install --name mychart --dry-run --debug -f global.yaml ./mychart/
helm install --name mychart --dry-run --debug --set course="k8s" ./mychart/

# 运行部分结果:
# Source: mychart/templates/configmap.yaml
apiVersion: v1
kind: ConfigMap
metadata:
  name: mychart-configmap
data:
  myvalue: "Hello World"
  course:  k8s
# 编辑 mychart/values.yaml，在最后加入
course:
  k8s: klvchen
  python: lily

cat mychart/templates/configmap.yaml
apiVersion: v1
kind: ConfigMap
metadata:
  name: {{ .Release.Name }}-configmap
data:
  myvalue: "Hello World"
  k8s:  {{ quote .Values.course.k8s }}      # quote 叫双引号
  python:  {{ .Values.course.python }}

helm install --name mychart --dry-run --debug ./mychart/
```

```
# 运行结果：
# Source: mychart/templates/configmap.yaml
apiVersion: v1
kind: ConfigMap
metadata:
  name: mychart-configmap
data:
  myvalue: "Hello World"
  k8s:  "klvchen"
  python:  lily
```

管道

```
k8s:  {{ quote .Values.course.k8s }} # 加双引号
k8s:  {{ .Values.course.k8s | upper | quote }} # 大写字符串加双引号
k8s:  {{ .Values.course.k8s | repeat 3 | quote }} # 加双引号和重复3次字符串
```

if/else 条件

```
if/else 块是用于在模板中有条件地包含文本块的方法，条件块的基本结构

{{ if PIPELINE }}
   # Do something
{{ else if OTHER PIPELINE }}
   # Do something else
{{ else }}
   # Default case
{{ end }}

# 判断条件，如果值为以下几种情况，管道的结果为 false：
1. 一个布尔类型的假
2. 一个数字零
3. 一个空的字符串
4. 一个 nil(空或null)
5. 一个空的集合(map, slice, tuple, dict, array)
除了上面的这些情况外，其他所有的条件都为真。

# 例子
cat mychart/templates/configmap.yaml
apiVersion: v1
kind: ConfigMap
metadata:
  name: {{ .Release.Name }}-configmap
data:
  myvalue: {{ .Values.hello | default "Hello World" | quote }}
  k8s:  {{ .Values.course.k8s | upper | quote | repeat 3 }}
  python:  {{ .Values.course.python | repeat 3 | quote }}
```

```
  {{ if eq .Values.course.python "django" }}web: true{{ end }}

helm install --name mychart --dry-run --debug ./mychart/

运行部分结果:
# Source: mychart/templates/configmap.yaml
apiVersion: v1
kind: ConfigMap
metadata:
  name: mychart-configmap
data:
  myvalue: "Hello World"
  k8s:  "KLVCHEN""KLVCHEN""KLVCHEN"
  python:  "djangodjangodjango"
  web: true

# 空格控制
{{- if eq .Values.course.python "django" }}
web: true
{{- end }}
```

With 关键字

```
with 关键字可以控制变量的作用域

{{ .Release.xxx }} 其中的.就是表示对当前范围的引用，.Values就是告诉模板在当前范围中查找Values对象的值。

with 语句可以允许将当前范围 . 设置为特定的对象，比如我们前面一直使用的 .Values.course,我们可以使用 with 来将范围指向 .Values.course:(templates/configmap.yaml)

with主要就是用来修改 . 作用域的，默认 . 代表全局作用域，with语句可以修改.的含义.

语法:
{{ with 引用的对象 }}
这里可以使用 . (点)， 直接引用with指定的对象
{{ end }}

例子:
#.Values.favorite是一个object类型
{{- with .Values.favorite }}
drink: {{ .drink | default "tea" | quote }}   #相当于.Values.favorite.drink
food: {{ .food | upper | quote }}
{{- end }}
```

range 关键字

```
range主要用于循环遍历数组类型。

语法1:
# 遍历map类型，用于遍历键值对象
# 变量key代表对象的属性名，val代表属性值
{{- range key,val := 键值对象 }}
{{ $key }}: {{ $val | quote }}
{{- end}}

语法2:
{{- range 数组 }}
{{ . | title | quote }} # . (点)，引用数组元素值。
{{- end }}

例子:
# values.yaml定义
# map类型
favorite:
  drink: coffee
  food: pizza

# 数组类型
pizzaToppings:
  - mushrooms
  - cheese
  - peppers
  - onions

# map类型遍历例子:
{{- range $key, $val := .Values.favorite }}
{{ $key }}: {{ $val | quote }}
{{- end}}

# 数组类型遍历例子:
{{- range .Values.pizzaToppings}}
{{ . | quote }}
{{- end}}
```

技巧

```
# 判断语法是否正确，  只打印不部署
helm install test --dry-run .
```