【云原生】Helm 架构和基础语法详解

大数据老司机 于 2022-09-05 07:30:00 发布 2363

分类专栏： 云原生 Kubernetes docker 文章标签： 云原生 kubernetes docker

文章目录

一、概述
二、Helm 架构
三、Helm 安装
四、Helm 组件及相关术语
五、Helm Chart 详解
1）Chart 目录结构
2）Chart.yaml 文件
3）Chart 依赖管理（dependencies）
4）通过依赖导入子Value
六、Templates and Values
1）Templates and Values 简介
2）预定义的 Values
3）范围，依赖和值
4）全局Values
七、Helm 资源安装顺序
八、Helm 安装 Chart 包的三种方式
1）values 传参
2）【第一种方式】直接在线 安装不需要先下载包到本地
3）【第二种方式】离线安装 直接通过安装包安装
4）【第三种方式】离线安装 解压包再安装
九、Helm 基础语法
1）变量
2）内置对象

3）常用的内置函数
    1、quote and squote
    2、default
    3、print
    4、println
    5、printf
    6、trim
    7、trimAll
    8、lower
    9、upper
    10、title
    11、substr
    12、abbrev
    13、contains
    14、cat
    15、indent
    16、nindent
    17、replace
    18、date
4）类型转换函数
5）正则表达式（Regular Expressions）
6）编码和解码函数
7）Dictionaries and Dict Functions
    1、创建字典（dict）
    2、获取值（get）
    3、添加键值对（set）
    4、删除（unset）
    5、判断key（hasKey）
    6、pluck
    7、合并 dict（merge, mustMerge）
    8、获取所有 keys

9、获取所有 values

8）Lists and List Functions

1、创建列表

2、获取列表第一项（first, mustFirst）

3、获取列表的尾部内容（rest, mustRest）

4、获取列表的最后一项（last, mustLast）

5、获取列表所有内容（initial, mustInitial）

6、末尾添加元素（append, mustAppend）

7、前面添加元素（prepend, mustPrepend）

8、多列表连接（concat）

9、反转（reverse, mustReverse）

10、去重（uniq, mustUniq）

11、过滤（without, mustWithout）

12、判断元素是否存在（has, mustHas）

13、删除空项（compact, mustCompact）

14、index

15、获取部分元素（slice, mustSlice）

16、构建一个整数列表（until）

17、seq

9）数学函数（Math Functions）

1、求和（add）

2、自加1（add1）

3、相减（sub）

4、除（div）

5、取模（mod）

6、相乘（mul）

7、获取最大值（max）

8、获取最小值（min）

9、获取长度（len）

10）Network Functions

10）条件语句

11）变更作用域 with
12）rang循环语句
13）命名模板
    1、用define和template声明和使用模板
    2、设置模板范围
    3、include 方法
14）NOTES.txt文件
15）模板调试

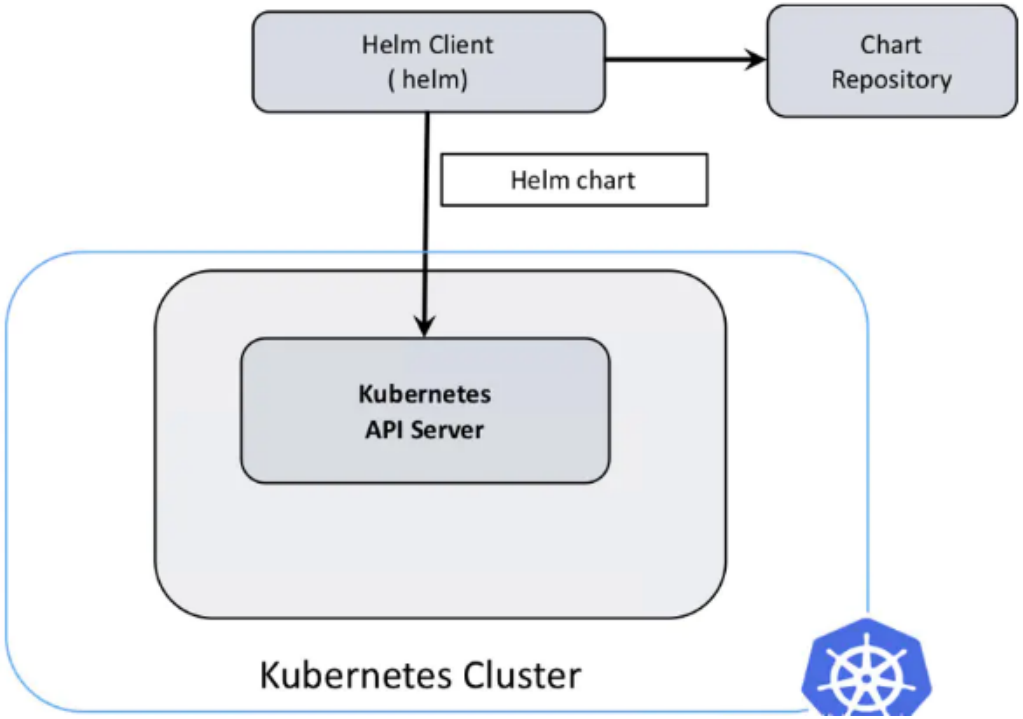
一、概述

> 我们可以将 Helm　看作Kubernetes下的apt-get/yum。Helm是kubernetes的包管理器，helm仓库里面只有配置清单文件,而没有镜像,镜像还是由镜像仓库来提供,比如hub.docker.com、私有仓库。

官方文档：https://v3.helm.sh/zh/docs/

其实之前也写过关于helm的一篇文章【Kubernetes（k8s）包管理器Helm（Helm3）介绍&Helm3安装Harbor】，可能讲的不够细致，这里会更加细致的讲解helm。

二、Helm 架构



三、Helm 安装

下载地址：https://github.com/helm/helm/releases

```
# 下载包
$  wget https://get.helm.sh/helm-v3.9.4-linux-amd64.tar.gz
# 解压压缩包
$ tar -xf helm-v3.9.4-linux-amd64.tar.gz
# 制作软连接
$ ln -s /opt/helm/linux-amd64/helm /usr/local/bin/helm
# 验证
$ helm version
$ helm help
```

四、Helm 组件及相关术语

- `Helm` ——Helm 是一个命令行下的**客户端工具**。主要用于 Kubernetes 应用程序 Chart 的创建、打包、发布以及创建和管理本地和远程的 Chart 仓库。
- `Chart` ——Chart 代表着 **Helm 包**。它包含在 Kubernetes 集群内部运行应用程序，工具或服务所需的所有资源定义。你可以把它看作是 Homebrew formula，Apt dpkg，或 Yum RPM 在Kubernetes 中的等价物。
- `Release` ——Release 是运行在 Kubernetes 集群中的 **chart 的实例**。一个 chart 通常可以在同一个集群中安装多次。每一次安装都会创建一个新的 release。
- `Repoistory` ——Repository（**仓库**） 是用来存放和共享 charts 的地方。它就像 Perl 的 CPAN 档案库网络 或是 Fedora 的 软件包仓库，只不过它是供 Kubernetes 包所使用的。

五、Helm Chart 详解

1）Chart 目录结构

```
# 通过helm create命令创建一个新的chart包
helm create nginx
tree nginx
```

```
[root@local-168-182-110 helm]# helm create nginx
Creating nginx
[root@local-168-182-110 helm]# tree nginx/
nginx/
├── charts
├── Chart.yaml
├── templates
│   ├── deployment.yaml
│   ├── _helpers.tpl
│   ├── hpa.yaml
│   ├── ingress.yaml
│   ├── NOTES.txt
│   ├── serviceaccount.yaml
│   ├── service.yaml
│   └── tests
│       └── test-connection.yaml
└── values.yaml

3 directories, 10 files
[root@local-168-182-110 helm]#
```

```
nginx/
├── charts  #依赖其他包的charts文件
├── Chart.yaml # 该chart的描述文件,包括ico地址,版本信息等
├── templates  # #存放k8s模板文件目录
│   ├── deployment.yaml # 创建k8s资源的yaml 模板
│   ├── _helpers.tpl # 下划线开头的文件,可以被其他模板引用
│   ├── hpa.yaml # 弹性扩缩容，配置服务资源CPU 内存
│   ├── ingress.yaml # ingress 配合service域名访问的配置
│   ├── NOTES.txt # 说明文件,helm install之后展示给用户看的内容
│   ├── serviceaccount.yaml # 服务账号配置
│   ├── service.yaml # kubernetes Serivce yaml 模板
│   └── tests # 测试模块
│       └── test-connection.yaml
└── values.yaml # 给模板文件使用的变量
```

可能有写包还会有以下几个目录：

```
wordpress/
...
  LICENSE             # 可选: 包含chart许可证的纯文本文件
  README.md           # 可选: 可读的README文件
  values.schema.json  # 可选: 一个使用JSON结构的values.yaml文件
  charts/             # 包含chart依赖的其他chart
  crds/               # 自定义资源的定义
...
```

2）Chart.yaml 文件

```
apiVersion: chart API 版本 （必需）
name: chart名称 （必需）
version: chart 版本，语义化2 版本（必需）
kubeVersion: 兼容Kubernetes版本的语义化版本（可选）
description: 一句话对这个项目的描述（可选）
type: chart类型 （可选）
keywords:
  - 关于项目的一组关键字（可选）
```

```
home: 项目home页面的URL （可选）
sources:
  - 项目源码的URL列表（可选）
dependencies: # chart 必要条件列表 （可选）
  - name: chart名称 (nginx)
    version: chart版本 ("1.2.3")
    repository: （可选）仓库URL ("https://example.com/charts") 或别名 ("@repo-name")
    condition: （可选） 解析为布尔值的yaml路径，用于启用/禁用chart (e.g. subchart1.enabled )
    tags: # （可选）
      - 用于一次启用/禁用 一组chart的tag
    import-values: # （可选）
      - ImportValue 保存源值到导入父键的映射。每项可以是字符串或者一对子/父列表项
    alias: （可选） chart中使用的别名。当你要多次添加相同的chart时会很有用
maintainers: # （可选）
  - name: 维护者名字 （每个维护者都需要）
    email: 维护者邮箱 （每个维护者可选）
    url: 维护者URL （每个维护者可选）
icon: 用做icon的SVG或PNG图片URL （可选）
appVersion: 包含的应用版本（可选）。不需要是语义化，建议使用引号
deprecated: 不被推荐的chart （可选，布尔值）
annotations:
  example: 按名称输入的批注列表 （可选）.
```

- 从 v3.3.2，不再允许额外的字段。推荐的方法是在 `annotations` 中添加自定义元数据。
- 每个chart都必须有个版本号（ `version` ）。版本必须遵循 语义化版本 2 标准。 不像经典Helm， Helm v2以及后续版本会使用版本号作为发布标记。仓库中的包通过名称加版本号标识。

比如 nginx chart的版本字段version: 1.2.3按照名称被设置为：

```
nginx-1.2.3.tgz
```

> 【温馨提示】 `appVersion` 字段与 `version` 字段并不相关。这是指定应用版本的一种方式。比如，这个drupal chart可能有一个 appVersion: “8.2.1”，表示包含在chart（默认）的Drupal的版本是8.2.1。

3）Chart 依赖管理（dependencies）

当前chart依赖的其他chart会在dependencies字段定义为一个列表。

```
dependencies:
  - name: apache
    version: 1.2.3
    repository: https://example.com/charts
  - name: mysql
    version: 3.2.1
    repository: https://another.example.com/charts
```

- name字段是你需要的chart的名称
- version字段是你需要的chart的版本
- repository字段是chart仓库的完整URL。注意你必须使用helm repo add在本地添加仓库
- 你可以使用仓库的名称代替URL

示例演示：

```
helm repo add bitnami https://charts.bitnami.com/bitnami
helm pull bitnami/wordpress
tar -xf wordpress
cat wordpress/Chart.yaml
```

```
[root@local-168-182-110 helm]# cat wordpress/Chart.yaml
annotations:
  category: CMS
apiVersion: v2
appVersion: 6.0.2
dependencies:
- condition: memcached.enabled
  name: memcached
  repository: https://charts.bitnami.com/bitnami
  version: 6.x.x
- condition: mariadb.enabled
  name: mariadb
  repository: https://charts.bitnami.com/bitnami
  version: 11.x.x
- name: common
  repository: https://charts.bitnami.com/bitnami
  tags:
  - bitnami-common
  version: 2.x.x
description: WordPress is the world's most popular blogging and content management
  platform. Powerful yet simple, everyone from students to global corporations use
  it to build beautiful, functional 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:
- name: Bitnami
  url: https://github.com/bitnami/charts
name: wordpress
```

一旦你定义好了依赖，运行 `helm dependency update` 就会使用你的依赖文件下载所有你指定的chart到你的charts/目录。

```
1 helm dependency update ./wordpress
```

当 helm dependency update 拉取chart时，会在charts/目录中形成一个chart包。因此对于上面的示例，会在chart目录中期望看到以下文件：

```
wordpress/charts/
├── common
├── common-2.0.1.tgz
├── mariadb
├── mariadb-11.2.2.tgz
├── memcached
└── memcached-6.2.3.tgz
```

```
[root@local-168-182-110 helm]# tree wordpress/charts/ -L 1
wordpress/charts/
├── common
├── common-2.0.1.tgz
├── mariadb
├── mariadb-11.2.2.tgz
├── memcached
└── memcached-6.2.3.tgz

3 directories, 3 files
[root@local-168-182-110 helm]#
```

依赖中的tag和条件字段

> 除了上面的其他字段外，每个需求项可以包含可选字段 `tags` 和 `condition`。**所有的chart会默认加载。**如果存在 tags 或者 condition 字段，它们将被评估并用于控制它们应用的chart的加载。

- `Condition` ——**条件字段**field 包含一个或多个YAML路径（用逗号分隔）。 如果这个路径在上层values中已存在并解析为布尔值，chart会基于布尔值启用或禁用chart。 只会使用列表中找到的第一个有效路径，如果路径为未找到则条件无效。
- `Tags` —— `tag` 字段是与chart关联的YAML格式的标签列表。在顶层value中，通过指定tag和布尔值，可以启用或禁用所有的带tag的chart。

```
# parentchart/Chart.yaml

```

```yaml
dependencies:
  - name: subchart1
    repository: http://localhost:10191
    version: 0.1.0
    condition: subchart1.enabled, global.subchart1.enabled
    tags:
      - front-end
      - subchart1
  - name: subchart2
    repository: http://localhost:10191
    version: 0.1.0
    condition: subchart2.enabled,global.subchart2.enabled
    tags:
      - back-end
      - subchart2
```

```yaml
# parentchart/values.yaml

subchart1:
  enabled: true
tags:
  front-end: false
  back-end: true
```

- 在上面的例子中，所有带 front-end tag的chart都会被禁用，但只要上层的value中 `subchart1.enabled` 路径被设置为 `'true'`，**该条件会覆盖 front-end标签且 subchart1 会被启用。**
- 一旦 subchart2使用了back-end标签并被设置为了 true，subchart2就会被启用。 也要注意尽管subchart2 指定了一个条件字段， 但是上层value没有相应的路径和value，因此这个条件不会生效。

`--set` 参数可以用来设置标签和条件值。

```shell
helm install --set tags.front-end=true --set subchart2.enabled=false
```

标签和条件的解析：

- 条件 （当设置在value中时）总是会覆盖标签 第一个chart条件路径存在时会忽略后面的路径。
- 标签被定义为 ‘如果任意的chart标签是true，chart就可以启用’。
- 标签和条件值必须被设置在顶层value中。
- value中的tags:键必须是顶层键。

4）通过依赖导入子Value

- 在某些情况下，**允许子chart的值作为公共默认传递到父chart中**是值得的。使用 `exports` 格式的额外好处是它可是将来的工具可以自检用户可设置的值。
- 被导入的包含值的key可以在父chart的 dependencies 中的 `import-values` 字段以YAML列表形式指定。 列表中的每一项是从子chart中exports字段导入的key。
- 导入exports key中未包含的值，使用 **子-父格式**。两种格式的示例如下所述。

使用导出格式：
如果子chart的values.yaml文件中在根节点包含了exports字段，它的内容可以通过指定的可以被直接导入到父chart的value中， 如下所示：

```
# parent's Chart.yaml file

dependencies:
  - name: subchart
    repository: http://localhost:10191
    version: 0.1.0
    import-values:
      - data
```

```
# child's values.yaml file

exports:
  data:
    myint: 99
```

> 只要我们再导入列表中指定了键data，Helm就会在子chart的exports字段查找data键并导入它的内容。

最终的父级value会包含我们的导出字段：

```
# parent's values

myint: 99
```

> 【注意】父级键 data 没有包含在父级最终的value中，如果想指定这个父级键，要使用 `'子-父' 格式` 。

下面示例中的 `import-values` 指示Helm去拿到能再child:路径中找到的任何值，并拷贝到parent:的指定路径。

```
# parent's Chart.yaml file

dependencies:
  - name: subchart1
    repository: http://localhost:10191
    version: 0.1.0
    ...
    import-values:
      - child: default.data
        parent: myimports
```

上面的例子中，在subchart1里面找到的default.data的值会被导入到父chart的myimports键中，细节如下：

```
# parent's values.yaml file

myimports:
  myint: 0
  mybool: false
  mystring: "helm rocks!"
```

```yaml
# subchart1's values.yaml file

default:
  data:
    myint: 999
    mybool: true
```

父chart的结果值将会是这样：

```yaml
# parent's final values

myimports:
  myint: 999
  mybool: true
  mystring: "helm rocks!"
```

六、Templates and Values

1）Templates and Values 简介

- Helm Chart 模板是按照 Go模板语言书写， 增加了50个左右的附加模板函数 来自 Sprig库 和一些其他 指定的函数。
- 所有模板文件存储在chart的 `templates/` 文件夹。 当Helm渲染chart时，它会通过模板引擎遍历目录中的每个文件。

模板的Value通过两种方式提供：

- Chart开发者可以在chart中提供一个命名为 `values.yaml` 的文件。这个文件包含了默认值。
- Chart用户可以提供一个包含了value的YAML文件。可以在命令行使用 helm install命令时通过 `-f` 指定value文件。

模板示例

```yaml
apiVersion: v1
kind: ReplicationController
metadata:
```

```
  name: deis-database
  namespace: deis
  labels:
    app.kubernetes.io/managed-by: deis
spec:
  replicas: 1
  selector:
    app.kubernetes.io/name: deis-database
  template:
    metadata:
      labels:
        app.kubernetes.io/name: deis-database
    spec:
      serviceAccount: deis-database
      containers:
        - name: deis-database
          image: {{ .Values.imageRegistry }}/postgres:{{ .Values.dockerTag }}
          imagePullPolicy: {{ .Values.pullPolicy }}
          ports:
            - containerPort: 5432
          env:
            - name: DATABASE_STORAGE
              value: {{ default "minio" .Values.storage }}
```

上面的例子，松散地基于 https://github.com/deis/charts， 是一个Kubernetes副本控制器的模板。可以使用下面四种模板值（一般被定义在values.yaml文件）：

- imageRegistry: Docker镜像的源注册表
- dockerTag: Docker镜像的tag
- pullPolicy: Kubernetes的拉取策略
- storage: 后台存储，默认设置为"minio"

2）预定义的 Values

> Values通过模板中.Values对象可访问的values.yaml文件（或者通过 --set 参数)提供， 但可以模板中访问其他预定义的数据片段。

以下值是预定义的，对每个模板都有效，并且可以被覆盖。和所有值一样，名称 区分大小写。

- `Release.Name` : 版本名称(非chart的)
- `Release.Namespace` : 发布的chart版本的命名空间
- `Release.Service` : 组织版本的服务
- `Release.IsUpgrade` : 如果当前操作是升级或回滚，设置为true
- `Release.IsInstall` : 如果当前操作是安装，设置为true
- `Chart` : `Chart.yaml` 的内容。因此，chart的版本可以从 Chart.Version 获得， 并且维护者在Chart.Maintainers里。
- `Files` : chart中的包含了非特殊文件的类图对象。这将不允许您访问模板， 但是可以访问现有的其他文件（除非被.helmignore排除在外）。 使用{{ index .Files “file.name” }}可以访问文件或者使用{{.Files.Get name }}功能。 您也可以使用{{ .Files.GetBytes }}作为[]byte访问文件内容。
- `Capabilities` : 包含了Kubernetes版本信息的类图对象。({{ .Capabilities.KubeVersion }}) 和支持的Kubernetes API 版本({{ .Capabilities.APIVersions.Has “batch/v1” }})

考虑到前面部分的模板， `values.yaml` 文件提供的必要值如下：

```
imageRegistry: "quay.io/deis"
dockerTag: "latest"
pullPolicy: "Always"
storage: "s3"
```

values文件被定义为YAML格式。chart会包含一个默认的values.yaml文件。 Helm安装命令允许用户使用附加的YAML values覆盖这个values：

```
helm install --generate-name --values=myvals.yaml wordpress
```

3）范围，依赖和值

Values文件可以声明顶级chart的值，以及 `charts/` 目录中包含的其他任意chart。 或者换个说法，**values文件可以为chart及其任何依赖项提供值。**比如，上面示范的WordPress chart同时有 mysql 和 apache 作为依赖。values文件可以为以下所有这些组件提供依赖：

```yaml
title: "My WordPress Site" # Sent to the WordPress template

mysql:
  max_connections: 100 # Sent to MySQL
  password: "secret"

apache:
  port: 8080 # Passed to Apache
```

更**高阶的chart可以访问下面定义的所有变量。**因此**WordPress chart可以用.Values.mysql.password访问MySQL密码。** 但是**低阶的chart不能访问父级chart**，所以MySQL无法访问title属性。同样也无法访问apache.port。

4）全局Values

从2.0.0-Alpha.2开始，Helm 支持特殊的"global"值。设想一下前面的示例中的修改版本：

```yaml
title: "My WordPress Site" # Sent to the WordPress template

global:
  app: MyWordPress

mysql:
  max_connections: 100 # Sent to MySQL
  password: "secret"

apache:
  port: 8080 # Passed to Apache
```

> 面添加了global部分和一个值app: MyWordPress。这个值以 `.Values.global.app` 在 **所有 chart中有效。**

比如，mysql模板可以以 `{{.Values.global.app}}` 访问app，同样apache chart也可以访问。 实际上，上面的values文件会重新生成为这样：

```yaml
title: "My WordPress Site" # Sent to the WordPress template

```

```yaml
global:
  app: MyWordPress

mysql:
  global:
    app: MyWordPress
  max_connections: 100 # Sent to MySQL
  password: "secret"

apache:
  global:
    app: MyWordPress
  port: 8080 # Passed to Apache
```

七、Helm 资源安装顺序

- Namespace
- NetworkPolicy
- ResourceQuota
- LimitRange
- PodSecurityPolicy
- PodDisruptionBudget
- ServiceAccount
- Secret
- SecretList
- ConfigMap
- StorageClass
- PersistentVolume
- PersistentVolumeClaim

- CustomResourceDefinition
- ClusterRole
- ClusterRoleList
- ClusterRoleBinding
- ClusterRoleBindingList
- Role
- RoleList
- RoleBinding
- RoleBindingList
- Service
- DaemonSet
- Pod
- ReplicationController
- ReplicaSet
- Deployment
- HorizontalPodAutoscaler
- StatefulSet
- Job
- CronJob
- Ingress
- APIService

八、Helm 安装 Chart 包的三种方式

Helm 自带一个强大的搜索命令，可以用来从两种来源中进行搜索：

- `helm search hub` 从 Artifact Hub 中查找并列出 helm charts。 Artifact Hub中存放了大量不同的仓库。
- `helm search repo` 从你添加（使用 `helm repo add`）到本地 helm 客户端中的仓库中进行查找。该命令基于本地数据进行搜索，无需连接互联网。

```
# 添加bitnami仓库源
helm repo add bitnami https://charts.bitnami.com/bitnami
# 从bitnami源查找所有chart包，不指定具体源的话，会查找本地添加的所有源地址的所有chart包
helm search repo bitnami
```

1）values 传参

安装过程中有两种方式传递配置数据：

- `--values（或 -f）`：使用 YAML 文件覆盖配置。可以指定多次，优先使用最右边的文件。
- `--set`：通过命令行的方式对指定项进行覆盖。

如果同时使用两种方式，则 --set 中的值会被合并到 --values 中，但是 `--set` 中的值**优先级更高**。在–set 中覆盖的内容会被被保存在 ConfigMap 中。可以通过 `helm get values <release-name>` 来**查看指定 release 中 --set 设置的值**。也可以通过运行 helm upgrade 并指定 --reset-values 字段来清除 --set 中设置的值。示例如下：

```
echo '{mariadb.auth.database: user0db, mariadb.auth.username: user0}' > values.yaml
helm install -f values.yaml bitnami/wordpress --generate-name
```

2）【第一种方式】直接在线 安装不需要先下载包到本地

```
helm install mysql bitnami/mysql
helm list
```

3）【第二种方式】离线安装 直接通过安装包安装

```
# 先删除
helm uninstall mysql
# 拉包到本地
helm pull bitnami/mysql
```

```
# 不解压直接安装
helm install mysql ./mysql-9.3.1.tgz
helm list
```

4）【第三种方式】离线安装 解压包再安装

```
# 拉包到本地
helm pull bitnami/mysql
# 解压安装
tar -xf mysql-9.3.1.tgz

# 开始安装
helm install mysql ./mysql \
--namespace=mysql \
--create-namespace \
--set image.registry=myharbor.com \
--set image.repository=bigdata/mysql \
--set image.tag=8.0.30 \
--set primary.service.type=NodePort \
--set service.nodePorts.mysql=30306

# 查看在运行的Release
helm list

# 卸载
helm uninstall mysql -n mysql
```

九、Helm 基础语法

1）变量

模板（`templates/`）中的**变量**都放在 `{{}}` 中，比如： `{{ .Values.images }}` 表示 `Values` 对象 下的images 字段。Values来源于 `values.yaml` 文件或者 `-f` 指定的yaml文件，或者 `--set` 设置的变量。

> 【温馨提示】使用 - 删除空格和换行符，要想删除那行其他的空格和换行符可以用 {{- 或者 -}} ，**一个是删除左边的** 空格 **和** 换行符 ，**一个是删除右边的** 空格 **和** 换行符 。

2）内置对象

- Release： Release对象描述了版本发布本身。包含了以下对象：
  - Release.Name： release名称；
  - Release.Namespace： 版本中包含的命名空间(如果manifest没有覆盖的话)；
  - Release.IsUpgrade： 如果当前操作是升级或回滚的话，该值将被设置为true
  - Release.IsInstall： 如果当前操作是安装的话，该值将被设置为true
  - Release.Revision： 此次修订的版本号。安装时是1，每次升级或回滚都会自增；
  - Release.Service： 该service用来渲染当前模板。Helm里始终Helm。
- Values： Values对象是从 values.yaml 文件和用户提供的文件传进模板的。默认为空
- Chart： Chart.yaml 文件内容。 Chart.yaml里的所有数据在这里都可以可访问的。比如 {{ .Chart.Name }}-{{ .Chart.Version }} 会打印出 mychart-0.1.0。
- Template： 包含当前被执行的当前模板信息
  - Template.Name : 当前模板的命名空间文件路径 (e.g. mychart/templates/mytemplate.yaml)；
  - Template.BasePath : 当前chart模板目录的路径 (e.g. mychart/templates)。

3）常用的 内置函数

1、quote and squote

该函数将值转**换成字符串**用**双引号(quote)** 或者**单引号(squote)**括起来。示例如下：

```
apiVersion: v1
kind: ConfigMap
metadata:
  name: {{ .Release.Name }}-configmap
data:
  myvalue: "Hello World"
```

```
  drink: {{ .Values.favorite.drink | quote }}
  food: {{ .Values.favorite.food | upper | quote }}
```

> 倒置命令是模板中的常见做法。可以经常看到 `.val | quote` 而不是 `quote .val`。实际上两种操作都是可以的。

2、default

这个函数允许你在模板中指定一个默认值，以防这个值被忽略。

```
# 如果.Values.favorite.drink是非空值，则使用它，否则会返回tea。
drink: {{ .Values.favorite.drink | default "tea" | quote }}

# 还可以这样写，如果.Bar是非空值，则使用它，否则会返回foo。
default "foo" .Bar
```

"空"定义取决于以下类型：

```
整型: 0
字符串: ""
列表: []
字典: {}
布尔: false
以及所有的nil (或 null)
```

3、print

返回各部分组合的字符串，非字符串类型会被转换成字符串。

```
print "Matt has " .Dogs " dogs"
```

> 【温馨提示】当相邻两个参数不是字符串时会在它们之间添加一个空格。

4、println

和 print效果一样，但会在末尾新添加一行。

5、printf

返回参数按顺序传递的格式化字符串。

```
printf "%s has %d dogs." .Name .NumberDogs
{{- printf "%d" (.Values.externalCache.port | int ) -}}
{{- printf "%s" .Values.existingSecret -}}

{{- printf "%v" .context.Values.redis.enabled -}}

# %s 字符串占位符，未解析的二进制字符串或切片
# %d 数字占位符，十进制
# %v 默认格式的值，当打印字典时，加号参数(%+v)可以添加字段名称
```

更多占位符的使用，可以参考官方文档：https://helm.sh/zh/docs/chart_template_guide/function_list/

6、trim

trim行数移除字符串两边的空格：

```
trim "   hello    "
```

7、trimAll

从字符串中移除给定的字符：

```
trimAll "$" "$5.00"
```

上述结果为：5.00 (作为一个字符串)。

8、lower

将整个字符串转换成小写：

```
lower "HELLO"
```

上述结果为： hello

9、upper

将整个字符串转换成大写：

```
upper "hello"
```

上述结果为： HELLO

10、title

首字母转换成大写：

```
title "hello world"
```

上述结果为： Hello World

11、substr

获取字符串的子串，有三个参数：

- start (int)
- end (int)
- string (string)

```
substr 0 5 "hello world"
```

上述结果为： hello

12、abbrev

用省略号截断字符串 (…)

```
abbrev 5 "hello world"
# 第一个参数：最大长度
# 第二个参数：字符串
```

上述结果为： he…， 因为将省略号算进了长度中。

13、contains

测试字符串是否包含在另一个字符串中：

```
contains "cat" "catch"
```

14、cat

cat 函数将多个字符串合并成一个，用空格分隔：

```
cat "hello" "beautiful" "world"
```

上述结果为： hello beautiful world

15、indent

indent 以**指定长度缩进**给定字符串所在行，在对齐多行字符串时很有用：

```
indent 4 $lots_of_text
```

上述结果会将每行缩进4个空格。

16、nindent

nindent 函数和indent函数一样，但可以在字符串开头添加新行。

```
nindent 4 $lots_of_text
```

上述结果会在字符串所在行缩进4个字符，并且在开头新添加一行。

17、replace

执行简单的字符串替换。

```
# 下面两行等价
replace " " "-" "I Am Henry VIII"
"I Am Henry VIII" | replace " " "-"

# 参数1：待替换字符串
# 参数2：要替换字符串
# 参数3：源字符串
```

上述结果为： I-Am-Henry-VIII

18、date

date函数格式化日期，日期格式化为YEAR-MONTH-DAY：

```
now | date "2006-01-02"
```

想了解更多内置函数，可以参考官方文档：https://helm.sh/zh/docs/chart_template_guide/function_list/

4）类型转换函数

Helm提供了以下类型转换函数：

- `atoi`：字符串转换成整型。
- `float64`：转换成 float64。
- `int`：按系统整型宽度转换成int。

- `int64` : 转换成 int64。
- `toDecimal` : 将unix八进制转换成int64。
- `toString` : 转换成字符串。
- `toStrings` : 将列表、切片或数组转换成字符串列表。
- `toJson (mustToJson)` : 将列表、切片、数组、字典或对象转换成JSON。
- `toPrettyJson (mustToPrettyJson)` : 将列表、切片、数组、字典或对象转换成格式化JSON。
- `toRawJson (mustToRawJson)` : 将列表、切片、数组、字典或对象转换成HTML字符未转义的JSON。

5）正则表达式（Regular Expressions）

Helm 包含以下正则表达式函数

- regexFind(mustRegexFind)
- regexFindAll(mustRegexFindAll)
- regexMatch (mustRegexMatch)
- regexReplaceAll (mustRegexReplaceAll)
- regexReplaceAllLiteral(mustRegexReplaceAllLiteral)
- regexSplit (mustRegexSplit)

6）编码和解码函数

Helm有以下编码和解码函数：

- b64enc/b64dec: 编码或解码 Base64
- b32enc/b32dec: 编码或解码 Base32

7）Dictionaries and Dict Functions

>

> Helm 提供了一个key/value存储类型称为dict（"dictionary"的简称，Python中也有）。dict是无序类型。**字典的key 必须是字符串。但值可以是任意类型，甚至是另一个dict 或 list。**

1、创建字典（dict）

下面是创建三个键值对的字典：

```
$myDict := dict "name1" "value1" "name2" "value2" "name3" "value 3"
```

2、获取值（get）

给定一个映射和一个键，从映射中获取值。

```
get $myDict "name1"
```

上述结果为：“value1”

> 注意如果没有找到，会简单返回""。不会生成error。

3、添加键值对（set）

使用set给字典添加一个键值对。

```
$_ := set $myDict "name4" "value4"
```

> 注意set 返回字典 (Go模板函数的一个要求)，因此你可能需要像上面那样使用使用 $_ 赋值来获取值。

4、删除（unset）

给定一个映射和key，从映射中删除这个key。

```
$_ := unset $myDict "name4"
```

和set一样，需要返回字典。

5、判断key（hasKey）

hasKey函数会在给定字典中包含了给定key时返回true。

```
hasKey $myDict "name1"
```

如果key没找到，会返回false。

6、pluck

pluck 函数给定一个键和多个映射，并获得所有匹配项的列表：

```
pluck "name1" $myDict $myOtherDict
```

上述会返回的list包含了每个找到的值([value1 otherValue1])。

7、合并 dict（merge, mustMerge）

将两个或多个字典合并为一个， 目标字典优先：

```
$newdict := merge $dest $source1 $source2
```

8、获取所有 keys

keys函数会返回一个或多个dict类型中所有的key的list。由于字典是 无序的，key不会有可预料的顺序。 可以使用sortAlpha存储。

```
keys $myDict | sortAlpha
```

> 当提供了多个词典时，key会被串联起来。使用 `uniq` 函数和 `sortAlpha` 获取一个唯一有序的键列表。

```
keys $myDict $myOtherDict | uniq | sortAlpha
```

9、获取所有 values

values函数类似于keys，返回一个新的list包含源字典中所有的value(只支持一个字典)。

```
$vals := values $myDict
```

上述结果为： list[“value1”, “value2”, “value 3”]。

> 注意 values不能保证结果的顺序；如果你需要顺序， 请使用 `sortAlpha` 。

8）Lists and List Functions

> Helm 提供了一个简单的list类型，包含任意顺序的列表。类似于数组或切片，但列表是被设计用于不可变数据类型。

1、创建列表

```
$myList := list 1 2 3 4 5
```

上述会生成一个列表 [1 2 3 4 5]。

2、获取列表第一项（first, mustFirst）

获取列表中的第一项，使用 first。

```
first $myList
# 返回 1
```

> first 有问题时会出错，mustFirst 有问题时会向模板引擎返回错误。

3、获取列表的尾部内容（rest, mustRest）

获取列表的尾部内容(除了第一项外的所有内容)，使用rest。

```
rest $myList
# 返回 [2 3 4 5]
```

> rest有问题时会出错，mustRest 有问题时会向模板引擎返回错误。

4、获取列表的最后一项（last, mustLast）

使用last获取列表的最后一项：

```
last $myList
# 返回 5。这大致类似于反转列表然后调用first。
```

5、获取列表所有内容（initial, mustInitial）

通过返回所有元素 但 除了最后一个元素。

```
 initial $myList
 # 返回 [1 2 3 4]。
```

> initial有问题时会出错，但是 mustInitial 有问题时会向模板引擎返回错误。

6、末尾添加元素（append, mustAppend）

在已有列表中追加一项，创建一个新的列表。

```
$new = append $myList 6
```

上述语句会设置 $new 为 [1 2 3 4 5 6]。 $myList会保持不变。

> append 有问题时会出错，但 mustAppend 有问题时会向模板引擎返回错误。

7、前面添加元素（prepend, mustPrepend）

将元素添加到列表的前面，生成一个新的列表。

```
prepend $myList 0
```

上述语句会生成 [0 1 2 3 4 5]。 $myList会保持不变。

> prepend 有问题时会出错，但 mustPrepend 有问题时会向模板引擎返回错误。

8、多列表连接（concat）

将任意数量的列表串联成一个。

```
concat $myList ( list 6 7 ) ( list 8 )
```

上述语句会生成 [1 2 3 4 5 6 7 8]。 $myList 会保持不变。

9、反转（reverse, mustReverse）

反转给定的列表生成一个新列表。

```
reverse $myList
```

上述语句会生成一个列表： [5 4 3 2 1]。

> reverse 有问题时会出错，但 mustReverse 有问题时会向模板引擎返回错误。

10、去重（uniq, mustUniq）

生成一个移除重复项的列表。

```
list 1 1 1 2 | uniq
```

上述语句会生成 [1 2]

> uniq 有问题时会出错，但 mustUniq 有问题时会向模板引擎返回错误。

11、过滤（without, mustWithout）

without 函数从列表中过滤内容。

```
without $myList 3
# 上述语句会生成 [1 2 4 5]
```

一个过滤器可以过滤多个元素：

```
without $myList 1 3 5
# 这样会得到： [2 4]
```

> without 有问题时会出错，但 mustWithout 有问题时会向模板引擎返回错误。

12、判断元素是否存在（has, mustHas）

验证列表是否有特定元素。

```
has 4 $myList
```

上述语句会返回 true, 但 has “hello” $myList 就会返回false。

> has 有问题时会出错，但 mustHas 有问题时会向模板引擎返回错误。

13、删除空项（compact, mustCompact）

接收一个列表并删除空值项。

```
$list := list 1 "a" "foo" ""
$copy := compact $list
```

compact 会返回一个移除了空值(比如， “”)的新列表。

> compact 有问题时会出错，但 mustCompact 有问题时会向模板引擎返回错误。

14、index

使用index list [n]获取列表的第n个元素。使用index list [n] [m] …获取多位列表元素。

- index $myList 0 返回 1，同 myList[0]
- index $myList 0 1 同 myList[0][1]

15、获取部分元素（slice, mustSlice）

从列表中获取部分元素，使用 slice list [n] [m]。等同于 list[n:m].

- slice $myList 返回 [1 2 3 4 5]。 等同于 myList[:]。
- slice $myList 3 返回 [4 5]等同于 myList[3:]。
- slice $myList 1 3 返回 [2 3]等同于 myList[1:3]。
- slice $myList 0 3 返回 [1 2 3]等同于 myList[:3]。

> slice 有问题时会出错，但 mustSlice 有问题时会向模板引擎返回错误。

16、构建一个整数列表（until）

until 函数构建一个整数范围。

```
until 5
```

上述语句会生成一个列表： [0, 1, 2, 3, 4]。

对循环语句很有用： range $i, $e := until 5。

17、seq

```
seq 5       => 1 2 3 4 5
seq -3      => 1 0 -1 -2 -3
seq 0 2     => 0 1 2
seq 2 -2    => 2 1 0 -1 -2
seq 0 2 10  => 0 2 4 6 8 10
seq 0 -2 -5 => 0 -2 -4
```

9）数学函数（Math Functions）

1、求和（add）

使用add求和。接受两个或多个输入。

```
add 1 2 3
```

2、自加1（add1）

自增加1，使用 add1。

3、相减（sub）

相减使用 sub。

4、除（div）

整除使用 div。

5、取模（mod）

取模使用mod。

6、相乘（mul）

相乘使用mul。接受两个或多个输入。

```
mul 1 2 3
```

7、获取最大值（max）

返回一组整数中最大的整数。

```
max 1 2 3
# 返回 3
```

8、获取最小值（min）

返回一组数中最小的数。

```
min 1 2 3
# 会返回 1。
```

9、获取长度（len）

以整数返回参数的长度。

```
len .Arg
```

10）Network Functions

Helm提供了几个网络函数：

- `getHostByName` 接收一个域名返回IP地址。
- `getHostByName` "www.google.com"会返回对应的www.google.com的地址。

10）条件语句

运算符：

```
eq: 等于（equal to）
ne: 不等于（not equal to）
lt: 小于（less than）
le: 小于等于（less than or equal to）
gt: 大于（greater than）
ge: 大于等于（greater than or equal to）
```

if/else 用法：

```
{{if 命令}}
…
{{else if 命令}}
…
{{else}}
…
{{end}}
```

如果是以下值时，管道会被设置为 false：

```
布尔false
数字0
空字符串
nil (空或null)
空集合(map, slice, tuple, dict, array)
```

【示例】

```
apiVersion: v1
kind: ConfigMap
metadata:
  name: {{ .Release.Name }}-configmap
data:
  myvalue: "Hello World"
```

```
  drink: {{ .Values.favorite.drink | default "tea" | quote }}
  food: {{ .Values.favorite.food | upper | quote }}
  {{ if eq .Values.favorite.drink "coffee" }}mug: "true"{{ end }}
```

11）变更作用域 with

> 下一个控制结构是 `with` 操作。这个用来**控制变量范围**。回想一下， `.` 是对 当前作用域 的引用。因此 `.Values` 就是告诉模板在当前作用域查找Values对象。

with的语法与if语句类似：

```
{{ with PIPELINE }}
  # restricted scope
{{ end }}
```

作用域可以被改变。with允许你为特定对象设定当前作用域(`.`)。比如，我们已经在使用.Values.favorite。 修改配置映射中的.的作用域指向.Values.favorite：

```
apiVersion: v1
kind: ConfigMap
metadata:
  name: {{ .Release.Name }}-configmap
data:
  myvalue: "Hello World"
  {{- with .Values.favorite }}
  drink: {{ .drink | default "tea" | quote }}
  food: {{ .food | upper | quote }}
  {{- end }}
```

但是这里有个注意事项，在限定的作用域内，**无法使用.访问父作用域**的对象。错误示例如下：

```
{{- with .Values.favorite }}
drink: {{ .drink | default "tea" | quote }}
food: {{ .food | upper | quote }}
release: {{ .Release.Name }}
{{- end }}
```

这样会报错因为 `Release.Name` **不在 `.` 限定的作用域内。**但是如果对调最后两行就是正常的， 因为在{{ end }}之后作用域被重置了。

```
{{- with .Values.favorite }}
drink: {{ .drink | default "tea" | quote }}
food: {{ .food | upper | quote }}
{{- end }}
release: {{ .Release.Name }}
```

或者，我们可以使用 `$` **从父作用域中访问Release.Name对象。**当模板开始执行后 `$` 会被映射到**根作用域**，且执行过程中不会更改。 下面这种方式也可以正常工作：

```
{{- with .Values.favorite }}
drink: {{ .drink | default "tea" | quote }}
food: {{ .food | upper | quote }}
release: {{ $.Release.Name }}
{{- end }}
```

也可以在外边定义变量，遵循 `$name变量` 的格式且指定了一个特殊的赋值运算符： `:=` 。 我们可以使用针对Release.Name的变量重写上述内容。

```
apiVersion: v1
kind: ConfigMap
metadata:
  name: {{ .Release.Name }}-configmap
data:
  myvalue: "Hello World"
  {{- $relname := .Release.Name -}}
  {{- with .Values.favorite }}
  drink: {{ .drink | default "tea" | quote }}
  food: {{ .food | upper | quote }}
  release: {{ $relname }}
  {{- end }}
```

> 注意在with块开始之前，赋值 `$relname := .Release.Name` 。 现在在with块中，$relname变量仍会执行版本名称。

12）rang循环语句

很多编程语言支持使用for循环，foreach循环，或者类似的方法机制。 在Helm的模板语言中，在一个集合中迭代的方式是使用 `range` 操作符。

定义values

```
favorite:
  drink: coffee
  food: pizza
pizzaToppings:
  - mushrooms
  - cheese
  - peppers
  - onions
```

现在我们有了一个pizzaToppings列表（模板中称为切片）。修改模板把这个列表打印到配置映射中：

```
apiVersion: v1
kind: ConfigMap
metadata:
  name: {{ .Release.Name }}-configmap
data:
  myvalue: "Hello World"
  {{- with .Values.favorite }}
  drink: {{ .drink | default "tea" | quote }}
  food: {{ .food | upper | quote }}
  {{- end }}
  toppings: |-
    {{- range .Values.pizzaToppings }}
    - {{ . | title | quote }}
    {{- end }}
```

有时能在模板中快速创建列表然后迭代很有用，Helm模板的tuple可以很容易实现该功能。在计算机科学中， 元组表示一个有固定大小的类似列表的集合，但可以是任意数据类型。这大致表达了 `tuple` 的用法。

```
  sizes: |-
    {{- range tuple "small" "medium" "large" }}
    - {{ . }}
    {{- end }}
```

上述模板会生成以下内容：

```
  sizes: |-
    - small
    - medium
    - large
```

13）命名模板

> 此时需要越过模板，开始创建其他内容了。该部分我们会看到如何在一个文件中定义 命名模板，并在其他地方使用。**命名模板** (有时称作一个 部分 或一个 子模板)仅仅是在文件内部定义的模板，并使用了一个名字。有两种创建方式和几种不同的使用方法。

- 三种声明和管理模板的方法： `define` ， `template` ，和 `block` ，在这部分，我们将使用这三种操作并介绍一种特殊用途的 `include` 方法，类似于template操作。
- 命名模板时要记住一个重要细节：**模板名称是全局的。**如果您想声明两个相同名称的模板，**哪个最后加载就使用哪个。** 因为在子chart中的模板和顶层模板一起编译，命名时要注意 chart特定名称。
- 一个常见的命名惯例是用chart名称作为模板前缀： `{{ define "mychart.labels" }}` 。使用**特定chart名称作为前缀**可以**避免可能因为 两个不同chart使用了相同名称的模板**而引起的冲突。

在编写模板细节之前，文件的命名惯例需要注意：

- templates/中的大多数文件被视为包含Kubernetes清单
- NOTES.txt是个例外
- 命名以下划线(_)开始的文件则假定 没有 包含清单内容。这些文件不会渲染为Kubernetes对象定义，但在其他chart模板中都可用。

> 这些文件用来存储局部和辅助对象，实际上当我们第一次创建mychart时，会看到一个名为 `_helpers.tpl` 的文件，这个文件是**模板局部的默认位置。**

1、用define和template声明和使用模板

define操作允许我们在模板文件中创建一个命名模板，**语法**如下：

```
{{- define "MY.NAME" }}
  # body of template here
{{- end }}
```

比如我们可以定义一个模板封装Kubernetes的标签：

```
{{- define "mychart.labels" }}
  labels:
    generator: helm
    date: {{ now | htmlDate }}
{{- end }}
```

现在我们将模板嵌入到了已有的配置映射中，然后使用 `template` 包含进来：

```
{{- define "mychart.labels" }}
  labels:
    generator: helm
    date: {{ now | htmlDate }}
{{- end }}
apiVersion: v1
kind: ConfigMap
metadata:
  name: {{ .Release.Name }}-configmap
  {{- template "mychart.labels" }}
data:
  myvalue: "Hello World"
  {{- range $key, $val := .Values.favorite }}
  {{ $key }}: {{ $val | quote }}
  {{- end }}
```

当模板引擎读取该文件时，它会存储mychart.labels的引用直到template "mychart.labels"被调用。 然后会按行渲染模板，因此结果类似这样：

```
# Source: mychart/templates/configmap.yaml
apiVersion: v1
kind: ConfigMap
metadata:
  name: running-panda-configmap
  labels:
    generator: helm
    date: 2022-09-04
data:
  myvalue: "Hello World"
  drink: "coffee"
  food: "pizza"
```

> 注意：define不会有输出，除非像本示例一样用模板调用它。

按照惯例，Helm chart将这些模板放置在局部文件中，一般是 `_helpers.tpl` 。把这个方法移到那里：

```
{{/* Generate basic labels */}}
{{- define "mychart.labels" }}
  labels:
    generator: helm
    date: {{ now | htmlDate }}
{{- end }}
```

2、设置模板范围

在上面定义的模板中，我们没有使用任何对象，仅仅使用了方法。修改定义好的模板让其包含chart名称和版本号：

```
{{/* Generate basic labels */}}
{{- define "mychart.labels" }}
  labels:
    generator: helm

```

```
    date: {{ now | htmlDate }}
    chart: {{ .Chart.Name }}
    version: {{ .Chart.Version }}
{{- end }}
```

3、include 方法

假设定义了一个简单模板如下：

```
{{- define "mychart.app" -}}
app_name: {{ .Chart.Name }}
app_version: "{{ .Chart.Version }}"
{{- end -}}
```

现在假设我想把这个插入到模板的labels:部分和data:部分：

```
apiVersion: v1
kind: ConfigMap
metadata:
  name: {{ .Release.Name }}-configmap
  labels:
    {{ template "mychart.app" . }}
data:
  myvalue: "Hello World"
  {{- range $key, $val := .Values.favorite }}
  {{ $key }}: {{ $val | quote }}
  {{- end }}
{{ template "mychart.app" . }}
```

如果渲染这个，会得到以下错误：

```
$ helm install --dry-run measly-whippet ./mychart
Error: unable to build kubernetes objects from release manifest: error validating "": error validating data: [ValidationError(Co
```

要查看渲染了什么，可以用 `--disable-openapi-validation` 参数重新执行： `helm install --dry-run --disable-openapi-validation measly-whippet ./mychart` 。 输入不是我们想要的：

```
# Source: mychart/templates/configmap.yaml
apiVersion: v1
kind: ConfigMap
metadata:
  name: measly-whippet-configmap
  labels:
    app_name: mychart
app_version: "0.1.0"
data:
  myvalue: "Hello World"
  drink: "coffee"
  food: "pizza"
app_name: mychart
app_version: "0.1.0"
```

> 注意两处的**app_version缩进都不对**，为啥？因为被替换的模板中文本是左对齐的。由于 `template` 是一个行为，**不是方法，无法将 `template` 调用的输出传给其他方法，数据只是简单地按行插入。**

为了处理这个问题，Helm提供了一个 `include` ，可以将模板内容导入当前管道，然后传递给管道中的其他方法。下面这个示例，使用 `indent` 正确地缩进了mychart.app模板：

```
apiVersion: v1
kind: ConfigMap
metadata:
  name: {{ .Release.Name }}-configmap
  labels:
{{ include "mychart.app" . | indent 4 }}
data:
  myvalue: "Hello World"
  {{- range $key, $val := .Values.favorite }}
  {{ $key }}: {{ $val | quote }}
```

```
  {{- end }}
{{ include "mychart.app" . | indent 2 }}
```

现在生成的YAML每一部分都可以正确缩进了：

```yaml
# Source: mychart/templates/configmap.yaml
apiVersion: v1
kind: ConfigMap
metadata:
  name: edgy-mole-configmap
  labels:
    app_name: mychart
    app_version: "0.1.0"
data:
  myvalue: "Hello World"
  drink: "coffee"
  food: "pizza"
  app_name: mychart
  app_version: "0.1.0"
```

> `include` 相较于使用template，在helm中**使用include被认为是更好的方式** 只是为了更好地处理YAML文档的输出格式。

14）NOTES.txt文件

> 该部分会**介绍为chart用户提供说明的Helm工具。**在helm install 或 helm upgrade命令的最后，**Helm会打印出对用户有用的信息。** 使用模板可以高度自定义这部分信息。

要在chart添加安装说明，只需创建 `templates/NOTES.txt` 文件即可。**该文件是纯文本，但会像模板一样处理**， 所有正常的模板函数和对象都是可用的。让我们创建一个简单的NOTES.txt文件：

```
Thank you for installing {{ .Chart.Name }}.

Your release is named {{ .Release.Name }}.
```

```
To learn more about the release, try:

  $ helm status {{ .Release.Name }}
  $ helm get all {{ .Release.Name }}
```

现在如果我们执行helm install rude-cardinal ./mychart 会在底部看到：

```
RESOURCES:
==> v1/Secret
NAME                   TYPE      DATA      AGE
rude-cardinal-secret   Opaque    1         0s

==> v1/ConfigMap
NAME                       DATA      AGE
rude-cardinal-configmap    3         0s


NOTES:
Thank you for installing mychart.

Your release is named rude-cardinal.

To learn more about the release, try:

  $ helm status rude-cardinal
  $ helm get all rude-cardinal
```

> 使用NOTES.txt这种方式是给用户提供关于如何使用新安装的chart细节信息的好方法。尽管并不是必需的，**强烈建议创建一个 `NOTES.txt` 文件。**

15）模板调试

调试模板可能很棘手，因为渲染后的模板发送给了Kubernetes API server，可能会以格式化以外的原因拒绝YAML文件。以下命令有助于调试：

- `helm lint` 是验证chart是否遵循最佳实践的首选工具
- `helm install --dry-run --debug` 或 `helm template --debug`： 我们已经看过这个技巧了， 这是让服务器渲染模板的好方法，然后返回生成的清单文件。
- `helm get manifest`：这是查看安装在服务器上的模板的好方法。

当你的YAML文件解析失败，但你想知道生成了什么，检索YAML一个简单的方式是**注释掉模板中有问题的部分**， 然后重新运行 `helm install --dry-run --debug`：

```
apiVersion: v2
# some: problem section
# {{ .Values.foo | quote }}
```

以上内容会被渲染同时返回完整的注释：

```
apiVersion: v2
# some: problem section
#  "bar"
```

自定义资源定义 (CRD)

Kubernetes 提供了一种机制来声明新类型的 `Kubernetes` 对象。使用 `CustomResourceDefinitions (CRD)`，Kubernetes 开发人员可以声明自定义资源类型。

在 `Helm 3` 中， `CRD` 被视为一种特殊的对象。它们在图表的其余部分之前安装，并且受到一些限制。

`CRD YAML` 文件应放在crds/图表内的目录中。多个 CRD（由 YAML 开始和结束标记分隔）可以放在同一个文件中。Helm 将尝试将CRD 目录中的所有文件加载到 Kubernetes 中。

`CRD` 文件不能被模板化。它们必须是纯 `YAML` 文档。

当 Helm 安装一个新图表时，它会上传 CRD，暂停直到 API 服务器使 CRD 可用，然后启动模板引擎，渲染图表的其余部分，并将其上传到 Kubernetes。由于这种排序，CRD 信息 `.Capabilities` 在 Helm 模板中的对象中可用，并且 Helm 模板可以创建在 CRD 中声明的对象的新实例。

例如，如果您的图表在目录中有一个 CRD ， 您可以 `CronTab` 在目录中创建该类型的crds/实例： `CronTabtemplates/`

```
crontabs/
  Chart.yaml
  crds/
    crontab.yaml
  templates/
    mycrontab.yaml
```

该 `crontab.yaml` 文件必须包含没有模板指令的 `CRD`：

```
kind: CustomResourceDefinition
metadata:
  name: crontabs.stable.example.com
spec:
  group: stable.example.com
  versions:
    - name: v1
      served: true
      storage: true
  scope: Namespaced
  names:
    plural: crontabs
    singular: crontab
    kind: CronTab
```

然后模板 `mycrontab.yaml` 可能会创建一个新的CronTab（像往常一样使用模板）：

```
apiVersion: stable.example.com
kind: CronTab
metadata:
  name: {{ .Values.name }}
spec:
   # ...
```

`Helm` 将确保该 `CronTab` 类型已安装并且在 `Kubernetes API` 服务器中可用，然后再继续在 `templates/.`

13.1 CRD 的限制

与 Kubernetes 中的大多数对象不同，CRD 是全局安装的。出于这个原因，Helm 在管理 CRD 时采取了非常谨慎的方法。CRD 受到以下限制：

- 永远不会重新安装 CRD。如果 Helm 确定 `crds/` 目录中的 CRD 已经存在（无论版本如何），Helm 将不会尝试安装或升级。
- CRD 永远不会在升级或回滚时安装。Helm 只会在安装操作时创建 CRD。
- CRD 永远不会被删除。删除 CRD 会自动删除集群中所有命名空间中的所有 CRD 内容。因此，Helm 不会删除 CRD。

鼓励想要升级或删除 CRD 的操作员手动执行此操作并非常小心。

其实这里主要是正对官方文档进行整理，列出了常见的使用语法，想了解更多，可以参考官方文档，官方文档讲解的很细致，有疑问的小伙伴欢迎给我留言哦，后续会持续分享【云原生和大数据】相关的文章，请小伙伴耐心等待哦~