



Helm User and Developer's Guide

目录

前言

序言	1.1
----	-----

用户指南

快速入门	2.1
安装	2.2
Kubernetes 各发行版 Helm	2.3
安装 FAQ	2.4
使用	2.5
插件	2.6
RBAC	2.7
使用 SSL	2.8
安全安装	2.9

Helm 命令参考

Charts

Charts	4.1
Hooks	4.2
提示和技巧	4.3
存储库 repository	4.4
同步 repository	4.5
验证出处和完整性	4.6
测试	4.7
Repository FAQ	4.8

开发 chart 模板

介绍	5.1
入门	5.2
内置对象	5.3
Values 文件	5.4
函数和管道	5.5
控制结构	5.6

变量

命名模板	5.8	5.7
访问文件		5.9
notes 文件		5.10
子 chart 和全局值		5.11
调试模板		5.12
总结		5.13
附录 1：Yaml 技巧		5.14
附录 2：数据类型		5.15

chart 最佳实践

介绍		6.1
通用约定		6.2
Values		6.3
Templates		6.4
Requirements		6.5
Labels 和 Annotations		6.6
Pods and PodTemplates		6.7
资源定义定制		6.8
RBAC		6.9

相关项目和文档

相关项目和文档		7.1
---------	--	-----

Kubernetes Helm 架构

Kubernetes Helm 架构		8.1
--------------------	--	-----

开发者指南

开发者指南		9.1
-------	--	-----

项目历史

项目历史		10.1
------	--	------

术语表

术语表		11.1
-----	--	------

何处寻找Charts

Helm User Guide - Helm 用户指南

本指南是官方 Kubernetes 的 github 库下，helm 子目录下的文档的翻译，依照 <https://docs.helm.sh/> 的文档架构和组织，当前非常粗糙的翻译了一版（剩余命令参考和术语表两部分还没有翻译），后续会陆续更新优化，用于给刚接触 Helm 这个工具的朋友一个参考。

备注：（如果有朋友知道如何更好的在 gitbook 里面处理花括弧 - 模板里的值引用，欢迎告知我）！！

电子书下载链接 <https://github.com/whmzsu/helm-doc-zh-cn/releases>

目录

前言

- [序言](#)

用户指南

- [快速入门](#)
- [安装](#)
- [Kubernetes 各发行版 Helm](#)
- [安装 FAQ](#)
- [使用](#)
- [插件](#)
- [RBAC](#)
- [使用 SSL](#)
- [安全安装](#)

Helm 命令参考

Charts

- [Charts](#)
- [Hooks](#)
- [提示和技巧](#)
- [存储库 repository](#)
- [同步 repository](#)
- [验证出处和完整性](#)
- [测试](#)
- [Repository FAQ](#)

开发模板

- [介绍](#)
- [快速入门](#)
- [内置对象](#)
- [Values 文件](#)
- [函数和管道](#)
- [控制结构](#)

- [变量](#)
- [命名模板](#)
- [访问文件](#)
- [notes 文件](#)
- [子 chart 和全局值](#)
- [调试模板](#)
- [总结](#)
- [附录 1：Yaml 技巧](#)
- [附录 2：数据类型](#)

最佳实践

- [介绍](#)
- [通用约定](#)
- [Values](#)
- [Templates](#)
- [Requirements](#)
- [Labels 和 Annotations](#)
- [Pods 和 PodTemplates](#)
- [资源定义定制](#)
- [RBAC](#)

相关项目和文档

- [相关项目和文档](#)

Kubernetes Helm 架构

- [Kubernetes Helm 架构](#)

开发者指南

- [开发者指南](#)

项目历史

- [项目历史](#)

术语表

- [术语表](#)

何处寻找 Charts

Copyright © Mingo(whmzsu@gmail.com) 2017-2018 all right reserved , powered by GitbookUpdated at 2018-06-29 17:18:20

快速入门

本指南介绍如何快速开始使用 Helm。

前提条件

需要准备以下前提条件才能成功且安全地使用 Helm。

1. 一个 Kubernetes 集群
2. 确定使用哪种安装安全配置（如果有的话）
3. 安装和配置 Helm 和集群端服务 Tiller。

安装 Kubernetes 或有权访问集群

- 必须已安装 Kubernetes。对于 Helm 的最新版本，我们推荐最新的 Kubernetes 稳定版本，在大多数情况下它是次新版本。
- 应该有一个本地配置好的 `kubectl`。

注意：1.6 之前的 Kubernetes 版本对于基于角色的访问控制（RBAC），要么有限制，或者不支持。

Helm 将通过 Kubernetes 配置文件（通常是 `$HOME/.kube/config`）来确定在哪里安装 Tiller。这个配置文件也是 `kubectl` 使用的文件。

要找出 Tiller 将安装到哪个集群，可以运行 `kubectl config current-context` 或 `kubectl cluster-info`。

```
$ kubectl config current-context
my-cluster
```

了解集群配置的安全上下文

与所有强大的工具一样，需要确保为你的场景正确安装它。

如果你在完全控制的集群上使用 Helm，如 minikube 或专用网络中的不考虑共享的集群，则默认安装（不采用安全配置）很合适，并且是最容易的。要在无需额外安全措施的场景下安装 Helm，请参考 [安装 Helm](#)，然后 [初始化 Helm](#)。

但是，如果集群暴露于更大的网络中，或者集群与他人共享 - 生产集群属于此类别 - 则必须采取额外步骤来确保安装安全，以防止不小心或恶意的操作者损坏集群或其集群数据。在生产环境和其他多租户方案中，要使用安全配置安装 Helm，请参阅 [Helm 安全安装](#)。

如果集群启用了基于角色的访问控制（RBAC），在继续之前配置 [服务帐户 \(service account\)](#) 和 [规则](#)。

安装 Helm

下载 Helm 客户端的二进制版本。可以使用类似工具如 `homebrew`，或查看 [官方发布页面](#)。

有关更多详细信息或其他选项，请参阅 [安装指南](#)。

初始化 Helm 并安装 Tiller

有了 Helm 安装文件，就可以初始化本地 CLI，并将 Tiller 安装到 Kubernetes 集群中：

```
$ helm init
```

这会将 Tiller 安装到对应的 Kubernetes 群集中, 集群同 `kubect1 config current-context`。

提示：想要安装到不同的群集中？使用 `--kube-context` 参数。

提示：如果要升级 Tiller，请运行 `helm init --upgrade`。

默认情况下，安装 Tiller 时，没有启用身份验证。要了解有关为 Tiller 配置增强 TLS 身份验证的更多信息，请参阅 [Tiller TLS 指南](#)。

安装示例 Chart

要安装一个 chart，可以运行 `helm install` 命令。Helm 有几种方法来查找和安装 chart，但最简单的方法是使用其中一个官方 `stable` 稳定版本的 chart。

```
$ helm repo update          # 确保我们获得最新的 chart 清单
$ helm install stable/mysql
Released smile-penguin
NAME:      wintering-rodent
LAST DEPLOYED: Thu Oct 18 14:21:18 2018
NAMESPACE: default
STATUS:    DEPLOYED

RESOURCES:
==> v1/Secret
NAME                                AGE
wintering-rodent-mysql              0s

==> v1/ConfigMap
wintering-rodent-mysql-test         0s

==> v1/PersistentVolumeClaim
wintering-rodent-mysql              0s

==> v1/Service
wintering-rodent-mysql              0s

==> v1beta1/Deployment
wintering-rodent-mysql              0s

==> v1/Pod(related)

NAME                                READY  STATUS   RESTARTS  AGE
wintering-rodent-mysql-6986fd6fb-988x7  0/1    Pending  0          0s

NOTES:
MySQL can be accessed via port 3306 on the following DNS name from within your cluster:
wintering-rodent-mysql.default.svc.cluster.local

To get your root password run:

    MYSQL_ROOT_PASSWORD=$(kubect1 get secret --namespace default wintering-rodent-mysql -o jsonpath="{.data.mysql-root-password}" | base64 --decode; echo)

To connect to your database:

1. Run an Ubuntu pod that you can use as a client:

    kubect1 run -i --tty ubuntu --image=ubuntu:16.04 --restart=Never -- bash -il

2. Install the mysql client:
```



```
$ apt-get update && apt-get install mysql-client -y
```

3. Connect using the mysql cli, then provide your password:
\$ mysql -h wintering-rodent-mysql -p

To connect to your database directly from outside the K8s cluster:
MYSQL_HOST=127.0.0.1
MYSQL_PORT=3306

```
# Execute the following command to route the connection:
kubectl port-forward svc/wintering-rodent-mysql 3306
```

```
mysql -h ${MYSQL_HOST} -P${MYSQL_PORT} -u root -p${MYSQL_ROOT_PASSWORD}
```

在上面的例子中，stable/mysql 已经安装，安装版本的 release 的名字是 `wintering-rodent`。通过运行 `helm inspect stable/mysql` 可以简单了解这个 MySQL chart 的功能。

无论何时安装 chart，都会创建一个新 release 版本。所以一个 chart 可以多次安装到同一个群集中。而且每个都可以独立管理和升级。

`helm install` 命令功能非常丰富，具有很多强大功能。要了解更多信息，请查看 [使用 Helm 指南](#)

了解安装的 release

很容易通过如下命令查看已使用 Helm 安装的 release：

```
$ helm ls
```

+NAME	REVISION	UPDATED	STATUS	CHART	APP VERSION	NAMESPA
CE						
+wintering-rodent	1	Thu Oct 18 15:06:58 2018	DEPLOYED	mysql-0.10.1	5.7.14	default

卸载安装的 release

要卸载安装的 release，请使用以下 `helm delete` 命令：

```
$ helm delete wintering-rodent
release "wintering-rodent" deleted
```

`wintering-rodent` release 将从 Kubernetes 卸载，但仍然可以查询有关该 release 的信息：

```
$ helm status wintering-rodent
LAST DEPLOYED: Thu Oct 18 14:21:18 2018
NAMESPACE: default
STATUS: DELETED
```

NOTES:

MySQL can be accessed via port 3306 on the following DNS name from within your cluster:
wintering-rodent-mysql.default.svc.cluster.local

To get your root password run:

```
MYSQL_ROOT_PASSWORD=$(kubectl get secret --namespace default wintering-rodent-mysql -o jsonpath="{.data.mysql-root-password}" | base64 --decode; echo)
```

To connect to your database:

1. Run an Ubuntu pod that you can use as a client:

```
kubect1 run -i --tty ubuntu --image=ubuntu:16.04 --restart=Never -- bash -il
```

2. Install the mysql client:

```
$ apt-get update && apt-get install mysql-client -y
```

3. Connect using the mysql cli, **then** provide your password:

```
$ mysql -h wintering-rodent-mysql -p
```

To connect to your database directly from outside the K8s cluster:

```
MYSQL_HOST=127.0.0.1
```

```
MYSQL_PORT=3306
```

```
# Execute the following command to route the connection:
```

```
kubect1 port-forward svc/wintering-rodent-mysql 3306
```

```
mysql -h ${MYSQL_HOST} -P${MYSQL_PORT} -u root -p${MYSQL_ROOT_PASSWORD}
```

由于 Helm 在删除它们之后也会跟踪该 release，因此可以审核群集的历史记录，甚至可以取消删除动作（使用 `helm rollback`）。

阅读帮助文本

要了解有关 Helm 命令的更多信息，请使用 `helm help` 或键入一个后跟 `-h` 标志的命令：

```
$ helm get -h
```

Copyright © Mingo(whmzsu@gmail.com) 2017-2018 all right reserved，powered by GitbookUpdated at 2018-11-23 22:31:49

安装

Helm 有两个部分：Helm 客户端（helm）和 Helm 服务端（Tiller）。本指南介绍如何安装客户端，然后继续演示两种安装服务端的方法。

重要提示：如果你负责的群集是在受控的环境，尤其是在共享资源时，强烈建议使用安全配置安装 Tiller。有关指导，请参阅 [安全 Helm 安装](#)。

安装 Helm 客户端

Helm 客户端可以从源代码安装，也可以从预构建的二进制版本安装。

从二进制版本

每一个版本 [release](#)Helm 提供多种操作系统的二进制版本。这些二进制版本可以手动下载和安装。

1. 下载你 [想要的版本](#)
2. 解压缩（`tar -zxvf helm-v2.0.0-linux-amd64.tgz`）
3. `helm` 在解压后的目录中找到二进制文件，并将其移动到所需的位置（`mv linux-amd64/helm /usr/local/bin/helm`）

到这里，你应该可以运行客户端了：`helm help`。

通过 Snap (Linux)

Snap package 维护站点 [Snapcrafters](#).

```
$ sudo snap install helm --classic
```

通过 homebrew (macOS)

Kubernetes 社区的成员为 Homebrew 贡献了 Helm。这个通常是最新的。

```
brew install kubernetes-helm
```

（注意：emacs-helm 也是一个软件，这是一个不同的项目。）

从 Chocolatey (Windows)

Kubernetes 社区的成员为 Chocolatey 贡献了 Helm 包。这个软件包通常是最新的。

```
choco install kubernetes-helm
```

从脚本

Helm 现在有一个安装 shell 脚本，将自动获取最新版本的 Helm 客户端并在本地安装。

可以获取该脚本，然后在本地执行它。这种方法也有文档指导，以便可以在运行之前仔细阅读并理解它在做什么。

```
$ curl https://raw.githubusercontent.com/helm/master/scripts/get > get_helm.sh
$ chmod 700 get_helm.sh
$ ./get_helm.sh
```

```
curl https://raw.githubusercontent.com/helm/master/scripts/get | bash
```

也可以做到这一点。

从金丝雀 (Canary) 构建

“Canary” 版本是从最新的主分支构建的 Helm 软件的版本。它们不是正式版本，可能不稳定。但是，它们提供了测试最新功能的机会。

“Canary” 版本 Helm 二进制文件存储在 Kubernetes Helm GCS 存储中。以下是常见构建的链接：

- [Linux AMD64](#)
- [macOS AMD64](#)
- [Experimental Windows AMD64](#)

源代码方式（Linux，macOS）

从源代码构建 Helm 的工作稍微多一些，但如果你想测试最新的（预发布）Helm 版本，那么这是最好的方法。

你必须有一个安装 Go 工作环境。

```
$ cd $GOPATH
$ mkdir -p src/k8s.io
$ cd src/k8s.io
$ git clone https://github.com/helm.git
$ cd helm
$ make bootstrap build
```

该 `bootstrap` 目标将尝试安装依赖，重建 `vendor/` 树，并验证配置。

该 `build` 目标编译 `helm` 并将其放置在 `bin/helm` 目录。Tiller 也会编译，并且被放置在 `bin/tiller` 目录。

安装 Tiller

Helm 的服务器端部分 Tiller 通常运行在 Kubernetes 集群内部。但是对于开发，它也可以在本地运行，并配置为与远程 Kubernetes 集群通信。

Special Note for RBAC Users

大多数云提供商都支持名为基于角色的访问控制（简称 RBAC）的特性。如果您的云提供商启用了该特性，您将需要为 Tiller 创建一个具有访问资源的正确角色和权限的服务帐户 (service account)。查看 [Kubernetes Distribution Guide](#) 在云提供商中使用 Helm 是否还有其他兴趣点。也可以查看 [Tiller and Role-Based Access Control](#) 来获取关于如何在 RBAC 的 K8S 集群中使用 Tiller 的更多信息。

快捷集群内安装

安装 `tiller` 到集群中最简单的方法就是运行 `helm init`。这将验证 `helm` 本地环境设置是否正确（并在必要时进行设置）。然后它会连接到 `kubectl` 默认连接的任何集群（`kubectl config view`）。一旦连接，它将安装 `tiller` 到 `kube-system` 命名空间中。

`helm init` 以后，可以运行 `kubectl get pods --namespace kube-system` 并看到 Tiller 正在运行。

你可以通过参数运行 `helm init`：

- `--canary-image` 参数安装金丝雀版本
- `--tiller-image` 安装特定的镜像（版本）
- `--kube-context` 使用安装到特定群集
- `--tiller-namespace` 用一个特定的命名空间 (namespace) 安装
- `--service-account` 使用 Service Account 安装 [RBAC enabled clusters](#))
- `--automount-service-account false` 不适用 service account 安装

一旦安装了 Tiller，运行 `helm version` 会显示客户端和服务端版本。（如果它仅显示客户端版本，`helm` 则无法连接到服务器，使用 `kubectl` 查看是否有任何 tiller Pod 正在运行。）

除非设置 `--tiller-namespace` 或 `TILLER_NAMESPACE` 参数，否则 Helm 将在命名空间 `kube-system` 中查找 Tiller。

安装 Tiller 金丝雀版本

Canary 镜像是从 master 分支建立的。他们可能不稳定，但他们提供测试最新功能的机会。

安装 Canary 镜像最简单的方法是 `helm init` 与 `--canary-image` 参数一起使用：

```
$ helm init --canary-image
```

这将使用最近构建的容器镜像。可以随时使用 `kubectl delete kube-system` 名称空间中的 Tiller deployment 来卸载 Tiller。

本地运行 Tiller

对于开发而言，有时在本地运行 Tiller 更容易，将其配置为连接到远程 Kubernetes 群集。

上面介绍了构建部署 Tiller 的过程。

一旦 tiller 构建部署完成，只需启动它：

```
$ bin/tiller
Tiller running on :44134
```

当 Tiller 在本地运行时，它将尝试连接到由 `kubectl` 配置的 Kubernetes 群集。（运行 `kubectl config view` 以查看是哪个群集。）

必须告知 `helm` 连接到这个新的本地 Tiller 主机，而不是连接到群集中的一个。有两种方法可以做到这一点。第一种是在命令行上指定 `--host` 选项。第二个是设置 `$HELM_HOST` 环境变量。

```
$ export HELM_HOST=localhost:44134
$ helm version # Should connect to localhost.
Client: &version.Version{SemVer:"v2.0.0-alpha.4", GitCommit:"db...", GitTreeState:"dirty"}
Server: &version.Version{SemVer:"v2.0.0-alpha.4", GitCommit:"a5...", GitTreeState:"dirty"}
```

注意，即使在本地运行，Tiller 也会将安装的 release 配置存储在 Kubernetes 内的 ConfigMaps 中。

升级 Tiller

从 Helm 2.2.0 开始，Tiller 可以升级使用 `helm init --upgrade`。

对于旧版本的 Helm 或手动升级，可以使用 `kubectl` 修改 Tiller 容器镜像：

```
$ export TILLER_TAG=v2.0.0-beta.1          # Or whatever version you want
$ kubectl --namespace=kube-system set image deployments/tiller-deploy tiller=gcr.io/kubernetes-helm/tiller:$TILLER_TAG
deployment "tiller-deploy" image updated
```

设置 `TILLER_TAG=canary` 将获得 master 版本的最新快照。

删除或重新安装 Tiller

由于 Tiller 将其数据存储在 Kubernetes ConfigMaps 中，因此可以安全地删除并重新安装 Tiller，而无需担心丢失任何数据。推荐删除 Tiller 的方法是使用 `kubectl delete deployment tiller-deploy --namespace kube-system` 或更简洁使用 `helm reset`。

然后可以从客户端重新安装 Tiller：

```
$ helm init
```

高级用法

`helm init` 提供了额外的参数，用于在安装之前修改 Tiller 的 deployment manifest。

使用 `--node-selectors`

`--node-selectors` 参数允许我们指定调度 Tiller Pod 所需的节点标签。

下面的例子将在 `nodeSelector` 属性下创建指定的标签。

```
helm init --node-selectors "beta.kubernetes.io/os"="linux"
```

已安装的 deployment manifest 将包含我们的节点选择器标签。

```
...
spec:
  template:
    spec:
      nodeSelector:
        beta.kubernetes.io/os: linux
...
```

使用 `--override`

`--override` 允许指定 Tiller 的 deployment manifest 的属性。与在 Helm 其他地方 `--set` 使用的命令不同，`helm init --override` 修改最终 manifest 的指定属性（没有 "values" 文件）。因此，可以为 deployment manifest 中的任何有效属性指定任何有效值。

覆盖注释

在下面的示例中，我们使用 `--override` 添加修订版本属性并将其值设置为 1。

```
helm init --override metadata.annotations."deployment\.kubernetes\.io/revision"="1"
```

输出：

```
apiVersion: extensions/v1beta1
kind: Deployment
metadata:
  annotations:
    deployment.kubernetes.io/revision: "1"
...
```

覆盖亲和性

在下面的例子中，我们为节点设置了亲和性属性。 `--override` 可以组合来修改同一列表项的不同属性。

```
helm init --override "spec.template.spec.affinity.nodeAffinity.preferredDuringSchedulingIgnoredDuringExecution[0].weight="1" --override "spec.template.spec.affinity.nodeAffinity.preferredDuringSchedulingIgnoredDuringExecution[0].preference.matchExpressions[0].key"="e2e-az-name"
```

指定的属性组合到“preferredDuringSchedulingIgnoredDuringExecution”属性的第一个列表项中。

```
...
spec:
  strategy: {}
  template:
    ...
    spec:
      affinity:
        nodeAffinity:
          preferredDuringSchedulingIgnoredDuringExecution:
            - preference:
                matchExpressions:
                  - key: e2e-az-name
                    operator: ""
                weight: 1
...

```

使用 --output

`--output` 参数允许我们跳过安装 Tiller 的 deployment manifest，并以 JSON 或 YAML 格式简单地将 deployment manifest 输出到标准输出 stdout。然后可以使用 `jq` 类似工具修改输出，并使用 `kubect1` 手动安装。

在下面的例子中，我们 `helm init` 用 `--output json` 参数执行。

```
helm init --output json
```

Tiller 安装被跳过，manifest 以 JSON 格式输出到 stdout。

```
"apiVersion": "extensions/v1beta1",
"kind": "Deployment",
"metadata": {
  "creationTimestamp": null,
  "labels": {
    "app": "helm",
    "name": "tiller"
  },
  "name": "tiller-deploy",
  "namespace": "kube-system"
},
...
```

存储后端

默认情况下，tiller 将安装 release 信息存储在其运行的名称空间中的 ConfigMaps 中。从 Helm 2.7.0 开始，现在有一个 Secrets 用于存储安装 release 信息的 beta 存储后端。添加了这个功能是为和 Kubernetes 的加密 Secret 一起，保护 chart 的安全性。

要启用 secrets 后端，需要使用以下选项启动 Tiller：

```
helm init --override 'spec.template.spec.containers[0].command={['/tiller,--storage=secret}]'
```

目前，如果想从默认后端切换到 secrets 后端，必须自行为此进行迁移配置信息。当这个后端从 beta 版本毕业时，将会有更正式的移徙方法。

总结

在大多数情况下，安装和获取预先构建的 helm 二进制代码和 `helm init` 一样简单。这个文档提供而了一些用例给那些想要用 Helm 做更复杂的事情的人。

一旦成功安装了Helm Client和Tiller，可以继续下一步使用Helm来管理charts。

Copyright © Mingo(whmzsu@gmail.com) 2017-2018 all right reserved，powered by GitbookUpdated at 2018-11-24 09:45:49

Kubernetes 各发行版本指南

本文档描述有关在各 Kubernetes 发行版本环境中使用 Helm 的信息。

我们尝试为此文档添加更多详细信息。如果可以，请通过 Pull Requests 提供。

MiniKube

Helm 已经过测试并且已知可以与 minikube 一起使用。它不需要额外的配置。

scripts/local-cluster 和 Hyperkube

通过配置 Hyperkube scripts/local-cluster.sh 已知可以工作。对于原始的 Hyperkube，可能需要进行一些手动配置。

GKE

已知 Google 的 GKE 托管 Kubernetes 平台 默认启用 RBAC。因此需要为 tiller 创建一个服务帐户（service account），并在初始化 helm 服务端时使用 --service-account 参数。查看 [Tiller](#) 和 [RBAC](#) 获取更详细的信息。

Ubuntu 与 'kubeadm'

kubeadm 构建的 Kubernetes 已知可用于以下 Linux 发行版：

- Ubuntu 16.04
- Fedora 发布 25

某些版本的 Helm（v2.0.0-beta2）要求 `export KUBECONFIG=/etc/kubernetes/admin.conf` 或创建一个 `~/.kube/config` 文件。

CoreOS 提供的 Container Linux

Helm 要求 kubelet 可以访问 socat 程序的副本，以代理与 Tiller API 的连接。在 Container Linux 上，Kubelet 在具有 socat 的 [hyperkube](#) 容器映像中运行。因此，尽管 Container Linux 没有 socat，运行 kubelet 的容器文件系统具有 socat。要了解更多信息，请阅读 [Kubelet Wrapper](#) 文档。

OpenShift

Helm 可在 OpenShift Online，OpenShift Dedicated，OpenShift Container Platform（版本 >= 3.6）或 OpenShift Origin（版本 >= 3.6）中直接使用。要了解更多信息，请阅读此 [博客文章](#)。

Platform9

Helm Client 和 Helm Server（Tiller）预装在 [Platform9 Managed Kubernetes](#)。Platform9 通过 App 目录 UI 和本地 Kubernetes CLI 提供对所有官方 Helm charts 的访问。其他 repo 存储库可以手动添加。有关更多详细信息，请参阅 [Platform9 App Catalog](#) 文章。

DC / OS

Helm（客户端和服务端）已经过测试，在Mesosphere DC / OS 1.11 Kubernetes平台工作正常，无需其他配置。

Copyright © Mingo(whmzs@gmail.com) 2017-2018 all right reserved , powered by GitbookUpdated at 2018-11-23 22:31:49

安装 FAQ

本节跟踪安装或开始使用 Helm 时遇到的一些经常遇到的问题。

欢迎你的帮助 来更好的提供此文档。要添加，更正或删除信息，提出问题 [issue](#) 或向我们发送 PR 请求。

下载

我想知道更多关于我的下载选项。

问：我无法获得最新 Helm 的 GitHub 发布。他们在哪？

答：我们不再使用 GitHub 发布版本。二进制文件现在存储在 GCS 公共存储区中 [GCS public bucket](#)。

问：为什么没有 Debian/Fedora/... Helm 的原生的软件包？

我们很乐意提供这些信息，或者指向可靠的提供商。如果你对帮助感兴趣，我们很乐意。这就是 Homebrew 式的开始。

问：你为什么要提供一个 `curl ...|bash` 脚本？

答：我们的 repo 库（ `scripts/get` ）中有一个脚本可以作为 `curl ...|bash` 脚本执行。这些传输全部受 HTTPS 保护，并且脚本会对其获取的包进行一些审计。但是，脚本具有任何 shell 脚本的所有常见危险。

我们提供它是因为它很有用，但我们建议用户先仔细阅读脚本。并且，我们真正喜欢的是 Helm 的的打包版本。

安装

我正在尝试安装 Helm/Tiller，但有些地方出了问题。

问：我如何将 Helm 客户端文件放在 `~/helm` 以外的地方？

设置 `$HELM_HOME` 环境变量，然后运行 `helm init`：

```
export HELM_HOME=/some/path
helm init --client-only
```

注意，如果你有现有的 repo 存储库，则需要通过 `helm repo add...` 重新添加它们。

问：我如何配置 Helm，但不安装 Tiller？

答：默认情况下，`helm init` 将确认本地 `$HELM_HOME` 配置，然后在集群上安装 Tiller。要本地配置，但不安装 Tiller，请使用 `helm init --client-only`。

问：如何在集群上手动安装 Tiller？

答：Tiller 是作为 Kubernetes deployment 安装的。可以通过运行 `helm init --dry-run --debug` 获取 manifest，然后通过 `kubectl` 手动安装。建议不要删除或更改该 deployment 中的标签 labels，因为它们有时支持脚本和工具需要用到。

问：为什么安装 Tiller 期间报错误 `Error response from daemon: target is unknown`？

答：有用户报告无法在使用 Docker 1.13.0 的 Kubernetes 实例上安装 Tiller。造成这种情况的根本原因是 Docker 中的一个错误，它使得一个版本与早期版本的 Docker 推送到 Docker 注册表的镜像不兼容。

该问题在发布后不久就已修复，并在 Docker 1.13.1-RC1 和更高版本中提供。

入门

我成功安装了 Helm/Tiller，但我使用时碰到问题。

问：使用 **Helm** 时，收到错误“客户端传输中断”

```
E1014 02:26:32.885226 16143 portforward.go:329] an error occurred forwarding 37008 -> 44134: error forwarding
port 44134 to pod tiller-deploy-2117266891-e4lev_kube-system, uid : unable to do port forwarding: socat not fo
und.
2016/10/14 02:26:32 transport: http2Client.notifyError got notified that the client transport was broken EOF.
Error: transport is closing
```

答：这通常表明 Kubernetes 未设置为允许端口转发。

通常情况下，缺少的部分是 socat。如果正在运行 CoreOS，我们被告知它可能在安装时配置错误。CoreOS 团队建议阅读以下内容：

<https://coreos.com/kubernetes/docs/latest/kubelet-wrapper.html>

以下是一些解决的问题案例，可以帮助开始使用：

- <https://github.com/kubernetes/helm/issues/1371>
- <https://github.com/kubernetes/helm/issues/966>

Q：使用 **Helm** 时，报错误 "lookup XXXXX on 8.8.8.8:53: no such host"

```
Error: Error forwarding ports: error upgrading connection: dial tcp: lookup kube-4gb-1on1-02 on 8.8.8.8:53: no
such host
```

答：我们在 Ubuntu 和 Kubeadm 多节点群集中有这个问题。问题原因是节点期望某些 DNS 记录可以通过全局 DNS 获得。在上游解决此问题之前，可以按照以下方式解决该问题。在每个控制平面节点上：

1. 添加条目到 `/etc/hosts`，将主机名映射到其 public IP
2. 安装 `dnsmasq`（例如 `apt install -y dnsmasq`）
3. 删除 k8s api 服务容器（kubelet 会重新创建它）
4. 然后 `systemctl restart docker`（或重新启动节点）请 `/etc/resolv.conf` 更改 请参阅此问题以获取更多信息：<https://github.com/kubernetes/helm/issues/1455>

问：在 GKE（Google Container Engine）上，报错 "No SSH tunnels currently open"

```
Error: Error forwarding ports: error upgrading connection: No SSH tunnels currently open. Were the targets able
to accept an ssh-key for user "gke-[redacted]"?
```

错误消息的另一个形式是：

```
Unable to connect to the server: x509: certificate signed by unknown authority
```

答：这个问题是你的本地 Kubernetes 配置文件必须具有正确的凭据。

在 GKE 上创建集群时，它将提供凭证，包括 SSL 证书和证书颁发机构信息。这些需要存储在一个 Kubernetes 配置文件中（默认：`~/.kube/config`，这样 `kubectl` 和 `helm` 可以访问它们）。

问：当我运行 **Helm** 命令时，出现有关隧道 **tunnel** 或代理 **proxy** 的错误

答：Helm 使用 Kubernetes 代理服务连接到 Tiller 服务器。如果命令 `kubectl proxy` 不适用，Helm 也不行。通常，错误与缺失的 `socat` 服务有关。

问：**Tiller** 崩溃

当我在 Helm 上运行命令时，Tiller 崩溃时会出现如下错误：

```
Tiller is listening on :44134
Probes server is listening on :44135
Storage driver is ConfigMap
Cannot initialize Kubernetes connection: the server has asked for the client to provide credentials 2016-12-20
15:18:40.545739 I | storage.go:37: Getting release "bailing-chinchilla" (v1) from storage
panic: runtime error: invalid memory address or nil pointer dereference
[signal SIGSEGV: segmentation violation code=0x1 addr=0x0 pc=0x8053d5]

goroutine 77 [running]:
panic(0x1abbfc0, 0xc42000a040)
    /usr/local/go/src/runtime/panic.go:500 +0x1a1
k8s.io/helm/vendor/k8s.io/kubernetes/pkg/client/unversioned.(*ConfigMaps).Get(0xc4200c6200, 0xc420536100, 0x15,
    0x1ca7431, 0x6, 0xc42016b6a0)
    /home/ubuntu/.go_workspace/src/k8s.io/helm/vendor/k8s.io/kubernetes/pkg/client/unversioned/configmap.go
:58 +0x75
k8s.io/helm/pkg/storage/driver.(*ConfigMaps).Get(0xc4201d6190, 0xc420536100, 0x15, 0xc420536100, 0x15, 0xc42053
60c0)
    /home/ubuntu/.go_workspace/src/k8s.io/helm/pkg/storage/driver/cfgmaps.go:69 +0x62
k8s.io/helm/pkg/storage.(*Storage).Get(0xc4201d61a0, 0xc4205360c0, 0x12, 0xc400000001, 0x12, 0x0, 0xc420200070)
    /home/ubuntu/.go_workspace/src/k8s.io/helm/pkg/storage/storage.go:38 +0x160
k8s.io/helm/pkg/tiller.(*ReleaseServer).uniqName(0xc42002a000, 0x0, 0x0, 0xc42016b800, 0xd66a13, 0xc42055a040,
    0xc420558050, 0xc420122001)
    /home/ubuntu/.go_workspace/src/k8s.io/helm/pkg/tiller/release_server.go:577 +0xd7
k8s.io/helm/pkg/tiller.(*ReleaseServer).prepareRelease(0xc42002a000, 0xc42027c1e0, 0xc42002a001, 0xc42016bad0,
    0xc42016ba08)
    /home/ubuntu/.go_workspace/src/k8s.io/helm/pkg/tiller/release_server.go:630 +0x71
k8s.io/helm/pkg/tiller.(*ReleaseServer).InstallRelease(0xc42002a000, 0x7f284c434068, 0xc420250c00, 0xc42027c1e0
, 0x0, 0x31a9, 0x31a9)
    /home/ubuntu/.go_workspace/src/k8s.io/helm/pkg/tiller/release_server.go:604 +0x78
k8s.io/helm/pkg/proto/hapi/services._ReleaseService_InstallRelease_Handler(0x1c51f80, 0xc42002a000, 0x7f284c434
068, 0xc420250c00, 0xc42027c190, 0x0, 0x0, 0x0, 0x0, 0x0)
    /home/ubuntu/.go_workspace/src/k8s.io/helm/pkg/proto/hapi/services/tiller.pb.go:747 +0x27d
k8s.io/helm/vendor/google.golang.org/grpc.(*Server).processUnaryRPC(0xc4202f3ea0, 0x28610a0, 0xc420078000, 0xc4
20264690, 0xc420166150, 0x288cbe8, 0xc420250bd0, 0x0, 0x0)
    /home/ubuntu/.go_workspace/src/k8s.io/helm/vendor/google.golang.org/grpc/server.go:608 +0xc50
k8s.io/helm/vendor/google.golang.org/grpc.(*Server).handleStream(0xc4202f3ea0, 0x28610a0, 0xc420078000, 0xc4202
64690, 0xc420250bd0)
    /home/ubuntu/.go_workspace/src/k8s.io/helm/vendor/google.golang.org/grpc/server.go:766 +0x6b0
k8s.io/helm/vendor/google.golang.org/grpc.(*Server).serveStreams.func1.1(0xc420124710, 0xc4202f3ea0, 0x28610a0,
    0xc420078000, 0xc420264690)
    /home/ubuntu/.go_workspace/src/k8s.io/helm/vendor/google.golang.org/grpc/server.go:419 +0xab
created by k8s.io/helm/vendor/google.golang.org/grpc.(*Server).serveStreams.func1
    /home/ubuntu/.go_workspace/src/k8s.io/helm/vendor/google.golang.org/grpc/server.go:420 +0xa3
```

答：请检查 Kubernetes 的安全设置。

Tiller 中的崩溃几乎总是由于未能与 Kubernetes API 服务器进行协商而导致的结果（此时，Tiller 功能不正常，因此崩溃并退出）。

通常，这是认证失败的结果，因为运行 Tiller 的 Pod 没有正确的令牌 token。

要解决这个问题，你需要修改 Kubernetes 配置。确保 `--service-account-private-key-file` 从 controller-manager 和 `--service-account-key-file` 从 API 服务器指向同一个 X509 RSA 密钥。

升级

我的 Helm 原来工作正常，然后我升级了。现在它工作不正常。

问：升级后，我收到错误 **“Client version is incompatible”**。怎么回事？

Tiller 和 Helm 必须协商一个通用版本，以确保他们可以安全地进行通信而不会违反 API 假设。该错误意味着版本差异太大而无法安全地继续。通常，需要为此手动升级 Tiller。

该安装指南 [Installation Guide](#) 有关于安全 Helm 升级和 Tiller 的详细信息。

版本号的规则如下：

- 预发布版本与其他一切不兼容。Alpha.1 与... 不相容 Alpha.2。
- 修补程序版本兼容：1.2.3 与 1.2.4 兼容
- 少量修订不兼容：1.2.0 与 1.3.0 不兼容，但我们可能在未来放宽这一限制。
- 主要版本不兼容：1.0.0 与 2.0.0 不兼容。

卸载

我正在尝试删除某些东西。

问：当我删除 **Tiller deployment** 时，为何所有安装的 **release** 信息还在集群里？

安装 release 信息存储在 kube-system 名称空间内的 ConfigMaps 中。需要手动删除它们以删除记录或使用 `helm delete --purge`。

问：我想删除我的本地 Helm。它的所有文件在哪里？

包括helm二进制文件，Helm存储了一些文件在\$HELM_HOME，默认位于~/.helm。

Copyright © Mingo(whmzs@gmail.com) 2017-2018 all right reserved , powered by GitbookUpdated at 2018-11-23 22:31:49

使用

本指南讲述使用 Helm（和 Tiller）来管理 Kubernetes 集群上的软件包的基础知识。前提是假定你已经安装了 Helm 客户端和 Tiller 服务端（通常通过 `helm init`）。

如果只是想运行一些简单命令，可以从 [快速入门指南](#) 开始。本章将介绍 Helm 命令的具体内容，并解释如何使用 Helm。

三大概念

一个 *Chart* 是一个 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。把它想像为一个自制软件，一个 Apt `dpkg` 或一个 Yum RPM 文件的 Kubernetes 环境里面的等价物。

一个 *Repository* 是 Charts 收集和共享的地方。它就像 Perl 的 [CPAN archive](#) 或 Fedora 软件包 repo [Fedora Package Database](#)。

一个 *Release* 是处于 Kubernetes 集群中运行的 Chart 的一个实例。一个 chart 通常可以多次安装到同一个群集中。每次安装时，都会创建一个新 *release*。比如像一个 MySQL chart。如果希望在群集中运行两个数据库，则可以安装该 chart 两次。每个都有自己的 *release*，每个 *release* 都有自己的 *release name*。

有了这些概念，我们现在可以这样解释 Helm：

Helm 将 *charts* 安装到 Kubernetes 中，每个安装创建一个新 *release*。要找到新的 chart，可以搜索 Helm charts 存储库 *repositories*。

'helm search': 查找 Charts

首次安装 Helm 时，它已预配置为使用官方 Kubernetes chart 存储库 repo。该 repo 包含许多精心设计和维护的 charts。此 charts repo 默认以 *stable* 命名。

可以通过运行 `helm search` 查看有哪些 charts 可用：

```
$ helm search
NAME                VERSION      DESCRIPTION
stable/drupal       0.3.2        One of the most versatile open source content m...
stable/jenkins      0.1.0        A Jenkins Helm chart for Kubernetes.
stable/mariadb      0.5.1        Chart for MariaDB
stable/mysql        0.1.0        Chart for MySQL
...
```

如果没有使用过滤条件，`helm search` 显示所有可用的 charts。可以通过使用过滤条件进行搜索来缩小搜索的结果范围：

```
$ helm search mysql
NAME                VERSION      DESCRIPTION
stable/mysql        0.1.0        Chart for MySQL
stable/mariadb      0.5.1        Chart for MariaDB
```

现在只会看到与过滤条件匹配的结果。

为什么 `mariadb` 在列表中？因为它的包描述与 MySQL 相关。我们可以使用 `helm inspect chart` 到这个：

```
$ helm inspect stable/mariadb
Fetched stable/mariadb to mariadb-0.5.1.tgz
description: Chart for MariaDB
```

```
engine: gotpl
home: https://mariadb.org
keywords:
- mariadb
- mysql
- database
- sql
...
```

搜索是找到可用软件包的好方法。一旦找到想要安装的软件包，可以使用 `helm install` 它来安装它。

'helm install'：安装一个软件包

要安装新的软件包，请使用该 `helm install` 命令。最简单的方法，它只需要一个参数：chart 的名称。

```
$ helm install stable/mariadb
Fetched stable/mariadb-0.3.0 to /Users/mattbutcher/Code/Go/src/k8s.io/helm/mariadb-0.3.0.tgz
NAME: happy-panda
LAST DEPLOYED: Wed Sep 28 12:32:28 2016
NAMESPACE: default
STATUS: DEPLOYED

Resources:
==> extensions/Deployment
NAME                DESIRED    CURRENT    UP-TO-DATE    AVAILABLE    AGE
happy-panda-mariadb 1            0            0              0            1s

==> v1/Secret
NAME                TYPE      DATA      AGE
happy-panda-mariadb Opaque    2           1s

==> v1/Service
NAME                CLUSTER-IP    EXTERNAL-IP    PORT(S)    AGE
happy-panda-mariadb 10.0.0.70     <none>         3306/TCP    1s

Notes:
MariaDB can be accessed via port 3306 on the following DNS name from within your cluster:
happy-panda-mariadb.default.svc.cluster.local

To connect to your database run the following command:

    kubectl run happy-panda-mariadb-client --rm --tty -i --image bitnami/mariadb --command -- mysql -h happy-panda-mariadb
```

现在 mariadb chart 已安装，请注意，安装 chart 会创建一个新 *release* 对象。上面的 release 被命名为 `happy-panda`。（如果你想使用你自己的 release 名称，只需使用 `--name` 参数 配合 `helm install`。）

在安装过程中，`helm` 客户端将打印有关创建哪些资源的有用信息，release 的状态以及是否可以或应该采取其他的配置步骤。

Helm 不会一直等到所有资源都运行才退出。许多 charts 需要大小超过 600M 的 Docker 镜像，因此可能需要很长时间才能安装到群集中。

要跟踪 release 状态或重新读取配置信息，可以使用 `helm status`：

```
$ helm status happy-panda
Last Deployed: Wed Sep 28 12:32:28 2016
Namespace: default
Status: DEPLOYED

Resources:
```



```

==> v1/Service
NAME                CLUSTER-IP    EXTERNAL-IP    PORT(S)    AGE
happy-panda-mariadb  10.0.0.70     <none>         3306/TCP    4m

==> extensions/Deployment
NAME                DESIRED    CURRENT    UP-TO-DATE    AVAILABLE    AGE
happy-panda-mariadb  1          1          1              1             4m

==> v1/Secret
NAME                TYPE        DATA    AGE
happy-panda-mariadb  Opaque      2        4m

Notes:
MariaDB can be accessed via port 3306 on the following DNS name from within your cluster:
happy-panda-mariadb.default.svc.cluster.local

To connect to your database run the following command:

    kubectl run happy-panda-mariadb-client --rm --tty -i --image bitnami/mariadb --command -- mysql -h happy-panda-mariadb

```

以上显示了集群内 release 的当前状态。

在安装前自定义 chart

上面的安装方式使用 chart 的默认配置选项。很多时候，我们需要自定义 chart 以使用自定义配置。

要查看 chart 上可配置的选项，请使用 `helm inspect values`：

```

helm inspect values stable/mariadb
Fetched stable/mariadb-0.3.0.tgz to /Users/mattbutcher/Code/Go/src/k8s.io/helm/mariadb-0.3.0.tgz
## Bitnami MariaDB image version
## ref: https://hub.docker.com/r/bitnami/mariadb/tags/
##
## Default: none
imageTag: 10.1.14-r3

## Specify a imagePullPolicy
## Default to 'Always' if imageTag is 'latest', else set to 'IfNotPresent'
## ref: http://kubernetes.io/docs/user-guide/images/#pre-pulling-images
##
# imagePullPolicy:

## Specify password for root user
## ref: https://github.com/bitnami/bitnami-docker-mariadb/blob/master/README.md#setting-the-root-password-on-first-run
##
# mariadbRootPassword:

## Create a database user
## ref: https://github.com/bitnami/bitnami-docker-mariadb/blob/master/README.md#creating-a-database-user-on-first-run
##
# mariadbUser:
# mariadbPassword:

## Create a database
## ref: https://github.com/bitnami/bitnami-docker-mariadb/blob/master/README.md#creating-a-database-on-first-run
##
# mariadbDatabase:

```

然后，可以在 YAML 格式的文件中覆盖任何这些设置，然后在安装过程中使用该文件。

```
$ echo '{mariadbUser: user0, mariadbDatabase: user0db}' > config.yaml
$ helm install -f config.yaml stable/mariadb
```

以上将创建一个名称为 MariaDB 的默认用户 `user0`，并授予此用户对新创建 `user0db` 数据库的访问权限，其他使用这个 chart 的默认值。

在安装过程中有两种方式传递自定义配置数据：

- `--values` (或 `-f`)：指定一个 overrides 的 YAML 文件。可以指定多次，最右边的文件将优先使用
- `--set` (也包括 `--set-string` 和 `--set-file`)：在命令行上指定 overrides。

如果两者都使用，则将 `--set` 值合并到 `--values` 更高的优先级中。指定的 override `--set` 将保存在 configmap 中。`--set` 可以通过使用特定的版本查看已经存在的值 `helm get values <release-name>`，`--set` 设置的值可以通过运行 `helm upgrade` 带有 `--reset-values` 参数重置。

`--set` 格式和限制

`--set` 选项使用零个或多个 name/value 对。最简单的用法：`--set name=value`。YAML 的表示是：

```
name: value
```

多个值由, 字符分隔。因此 `--set a=b,c=d` 变成：

```
a: b
c: d
```

支持更复杂的表达式。例如，`--set outer.inner=value` 变成这样：

```
outer:
  inner: value
```

列表可以通过在 {和} 中包含值来表示。例如，`--set name={a, b, c}` 转化为：

```
name:
- a
- b
- c
```

从 Helm 2.5.0 开始，可以使用数组索引语法访问列表项。例如，`--set servers[0].port=80` 变成：

```
servers:
- port: 80
```

可以通过这种方式设置多个值。该行 `--set servers[0].port=80,servers[0].host=example` 变成：

```
servers:
- port: 80
  host: example
```

有时候你需要在 `--set` 行中使用特殊字符。可以使用反斜杠来转义字符；`--set name="value1\,value2"` 会变成：

```
name: "value1,value2"
```

同样，也可以转义点序列，这可能在 chart 中使用 `toYaml` 函数解析注释，标签和节点选择器时派上用场。`--set nodeSelector."kubernetes.io/role"=master` 的语法变为：

```
nodeSelector:
  kubernetes.io/role: master
```

使用深层嵌套的数据结构可能很难用 `--set` 表达。鼓励 chart 设计师在设计 `values.yaml` 文件格式时考虑 `--set` 使用情况。

Helm 会使用 `--set` 将指定的某些值转换为整数。例如，`--set foo = true` Helm 会将 `true` 强制转换为 `int64` 值。如果你想要一个字符串，请使用 `--set` 的变体名为 `--set-string`。`--set-string foo = true` 会设置字符串值为 `"true"`。

`--set-file key = filepath` 是 `--set` 的另一种变体。它读取文件并将其内容用作值。它的一个示例用例是将多行文本注入值而不处理 YAML 中的缩进。假设您要创建一个 `brigade` 项目，其中包含包含 5 行 JavaScript 代码的特定值，您可以编写一个 `values.yaml`，如：

```
defaultScript: |
  const {events, Job} = require("brigadier")
  function run(e, project) {
    console.log("hello default script")
  }
  events.on("run", run)
```

嵌入在 YAML 中，这使你更难以使用支持编写代码的 IDE 功能和测试框架等。因此，你可以使用 `-set-file defaultScript = brigade.js` 替代，`brigade.js` 包含：

```
const {events, Job} = require("brigadier")
function run(e, project) {
  console.log("hello default script")
}
events.on("run", run)
```

更多的安装方法

`helm install` 命令可以从多个来源安装：

- 一个 chart repository (像上面看到的)
- 一个本地 chart 压缩包 (`helm install foo-0.1.1.tgz`)
- 一个解压后的 chart 目录 (`helm install path/to/foo`)
- 一个完整 URL (`helm install https://example.com/charts/foo-1.2.3.tgz`)

'helm upgrade' and 'helm rollback'：升级版本和失败时恢复

当新版本的 chart 发布时，或者当你想要更改 release 配置时，可以使用 `helm upgrade` 命令。

升级需要已有的 release 并根据提供的信息进行升级。由于 Kubernetes chart 可能很大而且很复杂，因此 Helm 会尝试执行最小侵入式升级。它只会更新自上次发布以来发生更改的内容。

```
$ helm upgrade -f panda.yaml happy-panda stable/mariadb
Fetched stable/mariadb-0.3.0.tgz to /Users/mattbutcher/Code/Go/src/k8s.io/helm/mariadb-0.3.0.tgz
happy-panda has been upgraded. Happy Helming!
Last Deployed: Wed Sep 28 12:47:54 2016
Namespace: default
Status: DEPLOYED
...
```

在上面的例子中，happy-panda release 使用相同的 chart 进行升级，但使用新的 YAML 文件：

```
mariadbUser: user1
```

我们可以使用 `helm get values` 看看这个新设置是否生效。

```
$ helm get values happy-panda
mariadbUser: user1
```

该 `helm get` 命令是查看集群中的 release 的有用工具。正如我们上面所看到的，它表明我们的新值 `panda.yaml` 已被部署到群集中。

现在，如果在发布过程中某些事情没有按计划进行，那么使用回滚到以前的版本很容易 `helm rollback [RELEASE] [REVISION]`。

```
$ helm rollback happy-panda 1
```

上述回滚我们的“happy-panda”到它的第一个 release 版本。release 版本是增量修订。每次安装，升级或回滚时，修订版本号都会增加 1。第一个修订版本号始终为 1。我们可以使用 `helm history [RELEASE]` 查看特定版本的修订版号。

安装 / 升级 / 回滚的有用选项

在安装 / 升级 / 回滚期间，可以指定几个其他有用的选项来定制 Helm 的行为。请注意，这不是 cli 参数的完整列表。要查看所有参数的说明，请运行 `helm --help`。

- `--timeout`：等待 Kubernetes 命令完成的超时时间值（秒），默认值为 300（5 分钟）
- `--wait`：等待所有 Pod 都处于就绪状态，PVC 绑定完，将 release 标记为成功之前，Deployments 有最小（Desired-maxUnavailable）Pod 处于就绪状态，并且服务具有 IP 地址（如果是 `LoadBalancer`，则为 `Ingress`）。它会等待 `--timeout` 的值。如果达到超时，release 将被标记为 FAILED。注意：在部署 replicas 设置为 1 maxUnavailable 且未设置为 0，作为滚动更新策略的一部分的情况下，`--wait` 它将返回就绪状态，因为它已满足就绪状态下的最小 Pod。
- `--no-hooks`：这会跳过命令的运行钩子
- `--recreate-pods`（仅适用于 upgrade 和 rollback）：此参数将导致重新创建所有 pod（属于 deployment 的 pod 除外）

'helm delete'：删除 Release

在需要从群集中卸载或删除 release 时，请使用以下 `helm delete` 命令：

```
$ helm delete happy-panda
```

这将从集群中删除该 release。可以使用以下 `helm list` 命令查看当前部署的所有 release：

```
$ helm list
NAME          VERSION   UPDATED              STATUS      CHART
inky-cat      1         Wed Sep 28 12:59:46 2016  DEPLOYED   alpine-0.1.0
```

从上面的输出中，我们可以看到该 happy-panda release 已被删除。

尽快如此，Helm 总是保留记录发生了什么。需要查看已删除的版本？`helm list --deleted` 可显示这些内容，并 `helm list --all` 显示了所有 release（已删除和当前部署的，以及失败的版本）：

```
=> helm list --all
NAME          VERSION   UPDATED              STATUS      CHART
happy-panda   2         Wed Sep 28 12:47:54 2016  DELETED    mariadb-0.3.0
inky-cat      1         Wed Sep 28 12:59:46 2016  DEPLOYED   alpine-0.1.0
kindred-angelf 2         Tue Sep 27 16:16:10 2016  DELETED    alpine-0.1.0
```

由于 Helm 保留已删除 release 的记录，因此不能重新使用 release 名称。（如果确实需要重新使用此 release 名称，则可以使用此 `--replace` 参数，但它只会重用现有 release 并替换其资源。）

请注意，因为 release 以这种方式保存，所以可以回滚已删除的资源并重新激活它。

'helm repo'：使用存储库

到目前为止，我们一直只从 stable 存储库 repo 安装 chart。但是可以配置 helm 使用其他 repo。Helm 在该 `helm repo` 命令下提供了多个 repo 工具。

可以使用 `helm repo list` 以下命令查看配置了哪些 repo：

```
$ helm repo list
NAME          URL
stable        https://kubernetes-charts.storage.googleapis.com
local         http://localhost:8879/charts
mumoshu       https://mumoshu.github.io/charts
```

新的 repo 可以通过 `helm repo add` 添加：

```
$ helm repo add dev https://example.com/dev-charts
```

由于 chart repo 经常更改，因此可以随时通过运行 `helm repo update` 确保 Helm 客户端处于最新状态。

创建你自己的 charts

该 chart 开发指南 [Chart Development Guide](#) 介绍了如何开发自己的 charts。也可以通过使用以下 `helm create` 命令快速入门：

```
$ helm create deis-workflow
Creating deis-workflow
```

现在有一个 chart `./deis-workflow`。可以编辑它并创建自己的模板。

在编辑 chart 时，可以通过 `helm lint` 验证它是否格式正确。

当将 chart 打包分发时，可以运行以下 `helm package` 命令：

```
$ helm package deis-workflow
deis-workflow-0.1.0.tgz
```

现在可以通过 `helm install` 以下方式轻松安装该 chart：

```
$ helm install ./deis-workflow-0.1.0.tgz
...
```

可以将已归档的 chart 加载到 chart repo 中。请参阅 chart repo 服务器的文档以了解如何上传。

注意：stable repo 在 Helm Charts GitHub 存储库 [Helm Charts GitHub repository](#) 上进行管理。该项目接受 chart 源代码，并且（在审计后）自动打包。

Tiller , Namespaces 和 RBAC

在某些情况下，可能希望将 Tiller 的范围或将多个 Tillers 部署到单个群集。以下是在这些情况下操作的一些最佳做法。

1. Tiller 可以安装到任何 namespace。默认情况下，它安装在 kube-system 中。可以运行多个 Tillers，只要它们各自在自己的 namespace 中运行。
2. 限制 Tiller 只能安装到特定的 namespace 和 / 或资源类型由 Kubernetes RBAC 角色和角色绑定控制。可以通过在配置 Helm 时通过 `helm init --service-account <NAME>` 向 Tiller 添加服务帐户。你可以在这里 [here](#) 找到更多的信息。
3. Release 名称在每个 Tiller 实例中是唯一的。
4. chart 应该只包含存在于单个命名空间中的资源。
5. 不建议将多个 Tillers 配置为在相同的命名空间中管理资源。

总结

本章介绍了 helm 客户端的基本使用模式，包括搜索，安装，升级和删除。它也涵盖了有用的工具命令类似如 `helm status`，`helm get` 和 `helm repo`。

有关这些命令的更多信息，请查看 Helm 的内置帮助：`helm help`。

在下一章中，我们将看看开发chart的过程。

Copyright © Mingo(whmzsu@gmail.com) 2017-2018 all right reserved , powered by GitbookUpdated at 2018-11-24 09:56:09

插件指南

Helm 2.1.0 引入了客户端 Helm 插件 *plugin* 的概念。插件是一种可以通过 helm CLI 访问的工具，但它不是内置 Helm 代码库的一部分。

现有的插件可以在相关部分 [related](#) 找到或者通过搜索 [Github](#)。

本指南介绍了如何使用和创建插件。

概述

Helm 插件是与 Helm 无缝集成的附加工具。它们提供了扩展 Helm 核心功能集的方法，但不需要将每个新功能都通过 Go 语言写入并添加到核心工具中。

Helm 插件具有以下功能：

- 可以在 Helm 安装中添加和删除它们，而不会影响核心 Helm 工具。
- 它们可以用任何编程语言编写。
- 它们与 Helm 集成，并出现在 helm help 和其他地方。

Helm 插件放置在 `$(helm home)/plugins`。

Helm 插件模型部分建模在 Git 的插件模型上。为此，有时可能会听到 helm 称为瓷层 *porcelain*，插件是管道 *plumbing*。这是揭示 Helm 提供用户体验和顶级处理逻辑，而插件则是执行所需操作的“细节工作”的简略说法。

安装插件

使用 `$ helm plugin install <path|url>` 命令安装插件。可以将路径设置为本地文件系统上的插件或远程 VCS repo 的 URL。 `helm plugin install` 命令克隆或复制该插件的路径 / URL 到给定的 `$(helm home)/plugins`

```
$ helm plugin install https://github.com/technosophos/helm-template
```

如果你有一个插件 tar 分发版，只需将插件解压到 `$(helm home)/plugins` 目录中即可。

也可以通过直接从 URL 安装 tarball 插件 `helm plugin install http://domain/path/to/plugin.tar.gz`

构建插件

在很多方面，插件类似于 chart。每个插件都有一个顶级目录，然后是一个 `plugin.yaml` 文件。

```
$(helm home)/plugins/  
|- keybase/  
|  
| |- plugin.yaml  
| |- keybase.sh
```

在上面的例子中，keybase 插件包含在名为 keybase 的目录中。它有两个文件：（ `plugin.yaml` 必需）和一个可执行脚本 `keybase.sh`（可选）。

插件的核心是一个简单的 YAML 文件 `plugin.yaml`。这是一个插件的一个插件 YAML，它增加了对 Keybase 操作的支持：

```
name: "keybase"
version: "0.1.0"
usage: "Integrate Keybase.io tools with Helm"
description: |-
  This plugin provides Keybase services to Helm.
ignoreFlags: false
useTunnel: false
command: "$HELM_PLUGIN_DIR/keybase.sh"
```

`name` 是插件的名称。当 Helm 执行插件时，这是它将使用的名称（例如，`helm NAME` 将调用此插件）。

`name` 应该匹配目录名称。在我们上面的例子中，这意味着插件 `name: keybase` 应该在一个名为 `keybase` 的目录中。

`name` 的限制：

- `name` 不能一个现有的 helm 顶级命令重复。
- `name` 必须限制为 ASCII `az`，`AZ`，`0-9` `_` 和 ```。

`version` 是插件的 SemVer 2 版本。`usage` 和 `description` 都用于生成命令的帮助文本。

`ignoreFlags` 告诉 Helm 不会将参数传递给插件。所以，如果一个插件被 `helm myplugin --foo` 调用，并且 `ignoreFlags: true`，那么 `--foo` 将被忽略。

`useTunnel` 指示插件需要一个隧道去连接 Tiller。这在任何时候插件与 Tiller 对接都应该设置为 `true`。它会使 Helm 打开一个隧道，然后 `$TILLER_HOST` 为该隧道设置正确的本地地址。不用担心：如果 Helm 由于 Tiller 在本地运行而检测到隧道是不必啊哟的，它就不会创建隧道。

最后，也是最重要的是，`command`，是这个插件在调用时会执行的命令。在执行插件之前会插入环境变量。上面的模式说明了指出插件程序所在位置的首选方式。

有一些使用插件命令的策略：

- 如果插件包含可执行文件 `command:`，则应将可执行文件打包到插件目录中。
- `command:` 将在执行前展开任何环境变量。`$HELM_PLUGIN_DIR` 将指向插件目录。
- 该命令本身不在 shell 中执行。所以你不能在一个 shell 脚本上运行。
- Helm 将大量配置注入到环境变量中。查看环境以查看可用信息。
- Helm 对插件的语言没有任何设限。你可以用你喜欢的任何方式来写。
- 命令负责执行具体的帮助文本 `-h` 和 `--help`。helm 将使用 `usage` 和 `description` 对 `helm help` 和 `helm help myplugin` 进行处理，但不会处理 `helm myplugin --help`。

下载器插件

默认情况下，Helm 可以使用 HTTP/S 获取图表。从 Helm 2.4.0 开始，插件可以从任意源下载 chart。

插件应在 `plugin.yaml` 文件（顶层）中声明这个特殊功能：

```
downloaders:
- command: "bin/mydownloader"
  protocols:
  - "myprotocol"
  - "myprotocols"
```

如果安装了这样的插件，Helm 可以通过调用 `command` 指定的协议方案与存储库 `repo` 进行交互。特殊存储库应与常规存储库类似添加：特殊存储库 `helm repo add favorite myprotocol://example.com/` 的规则与常规存储库的规则相同：Helm 必须能够下载 `index.yaml` 文件以发现并缓存可用 charts 列表。

定义的命令将使用以下方案调用：`command certFile keyFile caFile full-URL`。SSL 凭证来自存储在 `$HELM_HOME/repository/repositories.yaml` 其中的 `repo` 定义。下载器插件将原始内容转储到 `stdout` 并在 `stderr` 上报告错误。

环境变量

当 Helm 执行插件时，它将外部环境传递给插件，并且还会注入一些其他环境变量。

类似 `KUBECONFIG` 的变量将为插件设置，如果他们设置在外环境变量中。

保证以下变量设置：

- `HELM_PLUGIN`：插件目录的路径
- `HELM_PLUGIN_NAME`：插件的名称，正如 `helm` 所调用的。所以 `helm myplug` 会有简称 `myplug`。
- `HELM_PLUGIN_DIR`：包含该插件的目录。
- `HELM_BIN`：`helm` 命令的路径（由用户执行）。
- `HELM_HOME`：Helm 的 home 的路径。
- `HELM_PATH*`：重要 Helm 文件和目录的路径存储在前缀为 `HELM_PATH` 的环境变量中。
- `TILLER_HOST`：Tiller 的 `domain:port`。如果创建隧道，则会指向隧道的本地端点。否则，它会指向 ``$HELM_HOST``，`--host` 或默认主机（按照优先级的规则）。

虽然 `HELM_HOST` 可以设置，但不能保证它会指向正确的 Tiller 实例。这是为了允许插件开发人员在插件本身需要手动配置连接时以其原始状态进行访问 `HELM_HOST`。

关于 `useTunnel`

如果插件指定 `useTunnel: true`，Helm 将执行以下操作（按顺序）：

1. 解析全局标志和环境
2. 创建隧道
3. 设置 `TILLER_HOST`
4. 执行插件
5. 关闭隧道

命令退出后，隧道即被删除。因此，一个进程要使用该隧道，它不能是后台进程。

关于参数标记解析

在执行插件时，Helm 会解析全局标志以供自己使用。其中一些参数标志不会传递给插件。

- `--debug`：如果已指定，`$HELM_DEBUG` 则设为 `1`
- `--home`：这被转换为 `$HELM_HOME`
- `--host`：这被转换为 `$HELM_HOST`
- `--kube-context`：将丢弃。如果你的插件使用 `useTunnel`，这是用来为你设置隧道的。

`-h` 和 `--help`，插件应该显示帮助文本，然后退出。在所有其他情况下，插件可以根据需要使用参数标志。

Copyright © Mingo(whmzsu@gmail.com) 2017-2018 all right reserved, powered by GitbookUpdated at 2018-06-29 10:47:01

RBAC - 基于角色的访问控制

在 Kubernetes 中，确保应用程序在指定的范围内运行, 最佳的做法是，为特定的应用程序的服务帐户授予角色。要详细了解服务帐户权限请阅读 [官方 Kubernetes 文档](#)。

Bitnami 写了一个在集群中配置 RBAC 的 [指导](#)，可让你了解 RBAC 基础知识。

本指南面向希望对 Helm 限制如下权限的用户：

1. Tiller 将资源安装到特定 namespace 能力
2. 授权 Helm 客户端对 Tiller 实例的访问

Tiller 和基于角色的访问控制

可以在配置 Helm 时使用 `--service-account <NAME>` 参数将服务帐户添加到 Tiller。前提条件是必须创建一个角色绑定，来指定预先设置的角色 `role` 和服务帐户 `service account` 名称。

在前提条件下，并且有了一个具有正确权限的服务帐户，就可以像这样运行一个命令来初始化 Tiller：`helm init --service-account <NAME>`

Example: 服务账户带有 cluster-admin 角色权限

```
$ kubectl create serviceaccount tiller --namespace kube-system
serviceaccount "tiller" created
```

文件 `rbac-config.yaml`：

```
apiVersion: v1
kind: ServiceAccount
metadata:
  name: tiller
  namespace: kube-system
---
apiVersion: rbac.authorization.k8s.io/v1
kind: ClusterRoleBinding
metadata:
  name: tiller
roleRef:
  apiGroup: rbac.authorization.k8s.io
  kind: ClusterRole
  name: cluster-admin
subjects:
- kind: ServiceAccount
  name: tiller
  namespace: kube-system
```

Note: cluster-admin 角色是在 Kubernetes 集群中默认创建的，因此不必再显式地定义它。

```
$ kubectl create -f rbac-config.yaml
serviceaccount "tiller" created
clusterrolebinding "tiller" created
$ helm init --service-account tiller
```

在特定 namespace 中部署 Tiller，并仅限于在该 namespace 中部署资源

在上面的例子中，我们让 Tiller 管理访问整个集群。当然，Tiller 正常工作并不一定要为它设置集群管理员访问权限。我们可以指定 Role 和 RoleBinding 来将 Tiller 的范围限制为特定的 namespace，而不是指定 ClusterRole 或 ClusterRoleBinding。

```
$ kubectl create namespace tiller-world
namespace "tiller-world" created
$ kubectl create serviceaccount tiller --namespace tiller-world
serviceaccount "tiller" created
```

定义允许 Tiller 管理 namespace `tiller-world` 中所有资源的角色，文件 `role-tiller.yaml`：

```
kind: Role
apiVersion: rbac.authorization.k8s.io/v1
metadata:
  name: tiller-manager
  namespace: tiller-world
rules:
- apiGroups: ["", "extensions", "apps"]
  resources: ["*"]
  verbs: ["*"]
```

```
$ kubectl create -f role-tiller.yaml
role "tiller-manager" created
```

文件 `rolebinding-tiller.yaml`，

```
kind: RoleBinding
apiVersion: rbac.authorization.k8s.io/v1
metadata:
  name: tiller-binding
  namespace: tiller-world
subjects:
- kind: ServiceAccount
  name: tiller
  namespace: tiller-world
roleRef:
  kind: Role
  name: tiller-manager
  apiGroup: rbac.authorization.k8s.io
```

```
$ kubectl create -f rolebinding-tiller.yaml
rolebinding "tiller-binding" created
```

之后，运行 `helm init` 来在 `tiller-world` namespace 中安装 Tiller。

```
$ helm init --service-account tiller --tiller-namespace tiller-world
$HELM_HOME has been configured at /Users/awesome-user/.helm.
```

```
Tiller (the Helm server side component) has been installed into your Kubernetes Cluster.
Happy Helming!
```

```
$ helm install nginx --tiller-namespace tiller-world --namespace tiller-world
NAME:      wayfaring-yak
LAST DEPLOYED: Mon Aug  7 16:00:16 2017
NAMESPACE: tiller-world
STATUS:    DEPLOYED
```

```
RESOURCES:
==> v1/Pod
NAME                                READY  STATUS   RESTARTS  AGE
```

```
wayfaring-yak-alpine 0/1    ContainerCreating 0      0s
```

Example: 在一个 namespace 中部署 Tiller，并限制它在另一个 namespace 部署资源

在上面的例子中，我们让 Tiller 管理它部署所在的 namespace。现在，让我们限制 Tiller 的范围，将资源部署在不同的 namespace 中！

下面例子中，让我们在 `myorg-system` namespace 中安装 Tiller，并允许 Tiller 在 `myorg-users` namespace 中部署资源。

```
$ kubectl create namespace myorg-system
namespace "myorg-system" created
$ kubectl create serviceaccount tiller --namespace myorg-system
serviceaccount "tiller" created
```

在 `role-tiller.yaml` 中，定义了一个允许 Tiller 管理所有 `myorg-users` 资源的角色：

```
kind: Role
apiVersion: rbac.authorization.k8s.io/v1
metadata:
  name: tiller-manager
  namespace: myorg-users
rules:
- apiGroups: [ "", "extensions", "apps" ]
  resources: [ "*" ]
  verbs: [ "*" ]
```

```
$ kubectl create -f role-tiller.yaml
role "tiller-manager" created
```

将 service account 与那个 role 绑定。 `rolebinding-tiller.yaml`，

```
kind: RoleBinding
apiVersion: rbac.authorization.k8s.io/v1
metadata:
  name: tiller-binding
  namespace: myorg-users
subjects:
- kind: ServiceAccount
  name: tiller
  namespace: myorg-system
roleRef:
  kind: Role
  name: tiller-manager
  apiGroup: rbac.authorization.k8s.io
```

```
$ kubectl create -f rolebinding-tiller.yaml
rolebinding "tiller-binding" created
```

我们还需要授予 Tiller 访问权限来读取 `myorg-system` 中的 configmaps，以便它可以存储 release 信息。如 `role-tiller-myorg-system.yaml`：

```
kind: Role
apiVersion: rbac.authorization.k8s.io/v1
metadata:
  namespace: myorg-system
  name: tiller-manager
rules:
- apiGroups: [ "", "extensions", "apps" ]
```

```
resources: ["configmaps"]
verbs: ["*"]
```

```
$ kubectl create -f role-tiller-myorg-system.yaml
role "tiller-manager" created
```

相应的 role 绑定, 如 rolebinding-tiller-myorg-system.yaml :

```
kind: RoleBinding
apiVersion: rbac.authorization.k8s.io/v1
metadata:
  name: tiller-binding
  namespace: myorg-system
subjects:
- kind: ServiceAccount
  name: tiller
  namespace: myorg-system
roleRef:
  kind: Role
  name: tiller-manager
  apiGroup: rbac.authorization.k8s.io
```

```
$ kubectl create -f rolebinding-tiller-myorg-system.yaml
rolebinding "tiller-binding" created
```

Helm 和基于角色的访问控制

在 pod 中运行 Helm 客户端时, 为了让 Helm 客户端与 Tiller 实例进行通信, 需要授予某些特权。具体来说, Helm 客户端需要能够创建 pods, 转发端口并能够在 Tiller 运行的 namespace 中列出 pod (这样它才可以找到 Tiller)。

Example: 在一个 namespace 中部署 Helm, 与在另一个 namespace 中与 Tiller 交互

在这个例子中, 我们将假设 Tiller 在名为 `tiller-world` 的 namespace 中运行, 并且 Helm 客户端在 `helm-world` 的 namespace 中运行。默认情况下, Tiller 在 `kube-system` namespace 中运行。

如 helm-user.yaml :

```
apiVersion: v1
kind: ServiceAccount
metadata:
  name: helm
  namespace: helm-world
---
apiVersion: rbac.authorization.k8s.io/v1
kind: Role
metadata:
  name: tiller-user
  namespace: tiller-world
rules:
- apiGroups:
  - ""
  resources:
  - pods/portforward
  verbs:
  - create
- apiGroups:
  - ""
  resources:
  - pods
```

```
  verbs:
  - list
---
apiVersion: rbac.authorization.k8s.io/v1
kind: RoleBinding
metadata:
  name: tiller-user-binding
  namespace: tiller-world
roleRef:
  apiGroup: rbac.authorization.k8s.io
  kind: Role
  name: tiller-user
subjects:
- kind: ServiceAccount
  name: helm
  namespace: helm-world
```

```
$ kubectl create -f helm-user.yaml
serviceaccount "helm" created
role "tiller-user" created
rolebinding "tiller-user-binding" created
```

Copyright © Mingo(whmzsu@gmail.com) 2017-2018 all right reserved , powered by GitbookUpdated at 2018-11-23 22:31:49

在 Helm 和 Tiller 之间使用 SSL

本文讲述了如何在 Helm 和 Tiller 之间创建强 SSL/TLS 连接。这里强调的是创建一个内部 CA，并使用 SSL 的加密和身份识别功能。

在 Helm 2.3.0 中引入了对基于 TLS 的身份验证的支持

配置 SSL 是一个高级主题，需要你已了解 Helm 和 Tiller。

概述

Tiller 认证模型使用客户端 SSL 证书。Tiller 自己使用证书认证授权验证这些证书。同样，客户端还通过证书授权验证 Tiller 的身份。

有许多可能的设置证书和权限的配置，但我们在这里覆盖的方法适用于大多数情况。

从 Helm 2.7.2 开始，Tiller 要求客户端证书由其 CA 验证。在之前的版本中，Tiller 使用了允许自签名证书的较弱验证策略。

在本指南中，我们将展示如何：

- 创建用于为 Tiller 客户端和服务端颁发证书的私有 CA。
- 为 Tiller 创建证书
- 为 Helm 客户端创建一个证书
- 创建一个使用该证书的 Tiller 实例
- 配置 Helm 客户端以使用 CA 和客户端证书

在本指南结束时，你应该有一个正在运行的 Tiller 实例，它只接受来自可以通过 SSL 证书进行身份验证的客户端的连接。

生成证书认证授权和证书

生成 SSL CA 的一种方法是通过 `openssl` 命令行工具。线上提供了许多指南和最佳实践文档。此文档着重于在少量时间内准备好配置。对于生产配置，我们建议读者阅读官方文档 [the official documentation](#) 并咨询其他资源。

生成证书授权

生成证书授权的最简单方法是运行两个命令：

```
$ openssl genrsa -out ./ca.key.pem 4096
$ openssl req -key ca.key.pem -new -x509 -days 7300 -sha256 -out ca.cert.pem -extensions v3_ca
Enter pass phrase for ca.key.pem:
You are about to be asked to enter information that will be incorporated
into your certificate request.
What you are about to enter is what is called a Distinguished Name or a DN.
There are quite a few fields but you can leave some blank
For some fields there will be a default value,
If you enter '.', the field will be left blank.
-----
Country Name (2 letter code) [AU]:US
State or Province Name (full name) [Some-State]:CO
Locality Name (eg, city) []:Boulder
Organization Name (eg, company) [Internet Widgits Pty Ltd]:tiller
Organizational Unit Name (eg, section) []:
Common Name (e.g. server FQDN or YOUR name) []:tiller
```

```
Email Address []:tiller@example.com
```

请注意，上面输入的数据是样例数据。你应该根据自己的规格进行定制。

以上将生成一个密钥和一个 CA。请注意，这两个文件非常重要。尤其是 key 文件要特别注意处理。

通常，你需要生成中间签名密钥。为了简洁起见，我们将使用我们的根 CA 签署密钥。

生成证书

我们将生成两个证书，每个证书代表一种证书类型：

- 一个证书是用于 Tiller 的。每个 tiller 主机需要一个。
- 一个证书是给用户的。每个 helm 用户需要一个。

由于生成这些命令的命令是相同的，我们将同时创建。名字将表明他们的目标用处。

首先，Tiller 密钥：

```
$ openssl genrsa -out ./tiller.key.pem 4096
Generating RSA private key, 4096 bit long modulus
.....
.....++
e is 65537 (0x10001)
Enter pass phrase for ./tiller.key.pem:
Verifying - Enter pass phrase for ./tiller.key.pem:
```

接下来，生成 Helm 客户端的密钥：

```
$ openssl genrsa -out ./helm.key.pem 4096
Generating RSA private key, 4096 bit long modulus
.....++
.....++
e is 65537 (0x10001)
Enter pass phrase for ./helm.key.pem:
Verifying - Enter pass phrase for ./helm.key.pem:
```

同样，对于生产用途，将为每个用户生成一个客户端证书。

接下来，我们需要从这些密钥创建证书。对于每个证书，这有两个步骤，创建 CSR，然后创建证书。

```
$ openssl req -key tiller.key.pem -new -sha256 -out tiller.csr.pem
Enter pass phrase for tiller.key.pem:
You are about to be asked to enter information that will be incorporated
into your certificate request.
What you are about to enter is what is called a Distinguished Name or a DN.
There are quite a few fields but you can leave some blank
For some fields there will be a default value,
If you enter '.', the field will be left blank.
-----
Country Name (2 letter code) [AU]:US
State or Province Name (full name) [Some-State]:CO
Locality Name (eg, city) []:Boulder
Organization Name (eg, company) [Internet Widgits Pty Ltd]:Tiller Server
Organizational Unit Name (eg, section) []:
Common Name (e.g. server FQDN or YOUR name) []:tiller-server
Email Address []:

Please enter the following 'extra' attributes
```



```
to be sent with your certificate request
A challenge password []:
An optional company name []:
```

我们为 Helm 客户端证书重复这一步骤：

```
$ openssl req -key helm.key.pem -new -sha256 -out helm.csr.pem
# Answer the questions with your client user's info
```

（在极少数情况下，我们必须在生成请求时添加标志 `-nodes`。）

现在我们使用我们创建的 CA 证书对每个 CSR 进行签名（调整 `days` 参数以满足你的要求）：

```
$ openssl x509 -req -CA ca.cert.pem -CAkey ca.key.pem -CAcreateserial -in tiller.csr.pem -out tiller.cert.pem -
days 365
Signature ok
subject=/C=US/ST=CO/L=Boulder/O=Tiller Server/CN=tiller-server
Getting CA Private Key
Enter pass phrase for ca.key.pem:
```

再次为客户证书：

```
$ openssl x509 -req -CA ca.cert.pem -CAkey ca.key.pem -CAcreateserial -in helm.csr.pem -out helm.cert.pem -day
s 365
```

到此，对我们来说重要的文件是这些：

```
# The CA. Make sure the key is kept secret.
ca.cert.pem
ca.key.pem
# The Helm client files
helm.cert.pem
helm.key.pem
# The Tiller server files.
tiller.cert.pem
tiller.key.pem
```

现在我们准备好继续下一步。

创建自定义 Tiller 安装

Helm 全面支持创建 SSL 配置的部署。通过指定几个标志，`helm init` 命令可以创建一个新的 Tiller 安装，并完成所有 SSL 配置。

要看看这将产生什么，运行这个命令：

```
$ helm init --dry-run --debug --tiller-tls --tiller-tls-cert ./tiller.cert.pem --tiller-tls-key ./tiller.key.pe
m --tiller-tls-verify --tls-ca-cert ca.cert.pem
```

输出将显示一个 Deployment，一个 Secret 和一个 Service。SSL 信息将预先加载到 Secret 中，Deployment 将在启动时挂载到 pod。

如果要定制 manifest，可以将该输出保存到文件中，然后用 `kubectl create` 将其加载到群集中。

我们强烈建议在集群上启用 RBAC 并使用 RBAC 添加服务帐户 [service accounts](#)。

另外，可以删除 `--dry-run` 和 `--debug` 标志。我们还建议将 Tiller 放入非系统 namespace (`--tiller-namespace=something`) 并启用服务帐户 (`--service-account=somename`)。但是对于这个例子，我们将继续使用基础配置：

```
$ helm init --tiller-tls --tiller-tls-cert ./tiller.cert.pem --tiller-tls-key ./tiller.key.pem --tiller-tls-verify --tls-ca-cert ca.cert.pem
```

在一两分钟内它就应该准备好了。我们可以像这样检查 Tiller：

```
$ kubectl -n kube-system get deployment
NAME                DESIRED    CURRENT    UP-TO-DATE    AVAILABLE    AGE
... other stuff
tiller-deploy       1          1          1             1            2m
```

如果出现问题，可能需要使用 `kubectl get pods -n kube-system` 以找出问题所在。通过 SSL/TLS 支持，最常见的问题都与不正确生成的 TLS 证书有关，或意外更换证书和密钥。

此时，运行基本的 Helm 命令时应该会报错：

```
$ helm ls
Error: transport is closing
```

这是因为您的 Helm 客户端没有正确的证书来向 Tiller 进行身份验证。

配置 Helm 客户端

Tiller 服务现在运行通过 TLS 保护。现在需要配置 Helm 客户端来执行 TLS 操作。

对于快速测试，我们可以手动指定我们的配置。我们将运行一个普通的 Helm 命令 (`helm ls`)，但启用 SSL/TLS。

```
helm ls --tls --tls-ca-cert ca.cert.pem --tls-cert helm.cert.pem --tls-key helm.key.pem
```

此配置将发送我们的客户端证书以确认身份，使用客户端密钥进行加密，并使用 CA 证书验证远程 Tiller 的身份。

尽管如此，键入长命令很麻烦。快捷方法是将密钥，证书和 CA 移入 `$HELM_HOME`：

```
$ cp ca.cert.pem $(helm home)/ca.pem
$ cp helm.cert.pem $(helm home)/cert.pem
$ cp helm.key.pem $(helm home)/key.pem
```

有了这个，你可以简单地运行 `helm ls --tls` 以启用 TLS。

故障排除

- 运行命令，报错 `Error: transport is closing *`

这几乎总是由于配置错误导致客户端缺少证书 (`--tls-cert`) 或证书不正确。

- 我使用证书，但得到 `Error: remote error: tls: bad certificate *`

这意味着 Tiller 的 CA 无法验证你的证书。在上面的例子中，我们使用一个 CA 来生成客户端和服务端证书。在这些例子中，CA 已经签署了客户的证书。然后，我们将该 CA 加载到 Tiller。因此，当客户端证书发送到服务器时，Tiller 会根据 CA 检查客户端证书。

- 如果我使用 `--tls-verify` 客户端，报错 `Error: x509: certificate is valid for tiller-server, not localhost *`

如果打算 `--tls-verify` 在客户端上使用，则需要确保 Helm 连接的主机名与证书上的主机名匹配。在某些情况下，这很尴尬，因为 Helm 将通过本地主机 `localhost` 连接，或者 FQDN 不可用于公共解析。

- 如果我在客户端使用 `--tls-verify`，返回报错信息 `Error: x509: cannot validate certificate for 127.0.0.1 because it doesn't contain any IP SANs *`

默认情况下，Helm 客户端通过隧道（即 kube 代理）127.0.0.1 连接到 Tiller。在 TLS 握手期间，通常提供主机名（例如 `example.com`），对证书进行检查，包括附带的信息。但是，由于通过隧道，目标是 IP 地址。因此，要验证证书，必须在 Tiller 证书中将 IP 地址 127.0.0.1 列为 IP 附带备用名称（IP SAN：IP subject alternative name）。

例如，要在生成 Tiller 证书时将 127.0.0.1 列为 IP SAN：

```
$ echo subjectAltName=IP:127.0.0.1 > extfile.cnf
$ openssl x509 -req -CA ca.cert.pem -CAkey ca.key.pem -CAcreateserial -in tiller.csr.pem -out tiller.cert.pem -
days 365 -extfile extfile.cnf
```

- 如果我在客户端使用 `--tls-verify`，报错 `Error: x509: certificate has expired or is not yet valid *`

你的 Helm 证书已过期，需要使用你的私钥和 CA 签署新证书（并考虑增加天数）

如果你的 Tiller 证书已经过期，你需要签署一个新的证书，使用 base64 对它进行编码并更新 Tiller Secret：`kubect1`
`edit secret tiller-secret`

参考

<https://github.com/denji/golang-tls>

<https://www.openssl.org/docs/>

<https://jamielinux.com/docs/openssl-certificate-authority/sign-server-and-client-certificates.html>

Copyright © Mingo(whmzsu@gmail.com) 2017-2018 all right reserved，powered by GitbookUpdated at 2018-11-23 22:31:49

安全安装

Helm 是一款强大而灵活的 Kubernetes 软件包管理和运维工具。使用默认安装命令 `helm init` 可以快速轻松地安装它和 Tiller，与 Helm 相对应的服务端组件。

但是，默认安装没有启用任何安全配置。使用这种类型的安装下面的场景下是完全合适的，在没有安全问题或几乎没有安全问题的群集时可以使用这种安装方式，例如使用 Minikube 进行本地开发，或者使用在专用网络中，安全性良好且无数据共享或无其他用户或团队。如果是这种情况，那么默认安装很合适，但请记住：权力越大，责任越大。决定使用默认安装时始终要注意相应的安全问题。

谁需要安全配置？

对于以下类型的集群，我们强烈建议使用正确的安全配置应用于 Helm 和 Tiller，以确保集群，集群中的数据以及它所连接的网络安全。

- 暴露于不受控制的网络环境的集群：不受信任的网络参与者可以访问集群，也可以访问网络环境的不受信任的应用程序。
- 许多人使用的集群 - 多租户集群 - 作为共享环境
- 有权访问或使用高价值数据或任何类型网络的集群

通常，像这样的环境被称为 生产等级 或 生产质量 的环境，因为任何因滥用集群而对任何公司造成的损害对于客户，对公司本身或者两者都是深远的。一旦损害风险变得足够高，无论实际风险如何，都需要确保集群的安全完整性。

要为环境正确配置安装，必须：

- 了解群集的安全上下文
- 选择合适的 helm 安装的最佳实践

以下假定有一个 Kubernetes 配置文件（一个 kubeconfig 文件），或者有一个用于访问群集的文件。

了解群集的安全上下文

helm init 将 Tiller 安装到 kube-system 名称空间中的集群中，而不应应用任何 RBAC 规则。这适用于本地开发和其他私人场景，因为它可以让立即开始工作。它还使你能够继续使用没有基于角色的访问控制（RBAC）支持的 Kubernetes 群集来运行 Helm，直到可以将工作负载移动到更新的 Kubernetes 版本。

在 Tiller 安全安装时，需要考虑四个主要方面：

1. 基于角色的访问控制或 RBAC
2. Tiller 的 gRPC 端点及 Helm 的使用情况
3. Tiller 的 release 信息
4. Helm harts

RBAC

Kubernetes 的最新版本采用基于角色的访问控制（[RBAC] (https://en.wikipedia.org/wiki/Role-based_access_control)）系统（与现代操作系统一样），以帮助缓解证书被滥用或存在错误时可能造成的损害。即使在身份被劫持的情况下，这个身份在受控空间也只有这么多的权限。这有效地增加了一层安全性，以限制使用该身份进行攻击的范围。

Helm 和 Tiller 在安装，删除和修改逻辑应用程序时，可以包含许多服务交互。因此，它的使用通常涉及整个集群的操作，在多租户集群中意味着 Tiller 安装必须非常小心才能访问整个集群，以防止不正确的安全活动。

特定用户和团队 - 开发人员，运维人员，系统和网络管理员 - 需要他们自己的群集分区，以便他们可以使用 Helm 和 Tiller，而不会冒着集群其他分区的风险。这需要启用 RBAC 的 Kubernetes 集群，并配置 Tiller 的 RBAC 权限。有关在 Kubernetes 中使用 RBAC 的更多信息，请参阅使用 RBAC 授权 [Using RBAC Authorization](#)。

Tiller 和用户权限

当前情况下的 Tiller 不提供将用户凭据映射到 Kubernetes 内的特定权限的方法。当 Tiller 在集群内部运行时，它将使用其服务帐户的权限运行。如果没有服务帐户名称提供给 Tiller，它将使用该名称空间的默认服务帐户运行。这意味着该服务器上的所有 Tiller 操作均使用 Tiller pod 的凭据和权限执行。

为了合适的限制 Tiller 本身的功能，标准 Kubernetes RBAC 机制必须配置到 Tiller 上，包括角色和角色绑定，这些角色明确的限制了 Tiller 实例可以安装什么以及在哪里安装。

这种情况在未来可能会改变。社区有几种方法可以解决这个问题，采用客户端权限而不是 Tiller 权限的情况下，活动的权限取决于 Pod 身份工作组，已经解决了的安全的一般性问题。

Tiller gRPC 端点和 TLS

在默认安装中，Tiller 提供的 gRPC 端点在集群内部（不在集群外部）可用，不需要应用认证配置。如果不应身份验证，集群中的任何进程都可以使用 gRPC 端点在集群内执行操作。在本地或安全的专用群集中，这可以实现快速使用并且是合适的。（当在集群外部运行时，Helm 通过 Kubernetes API 服务器进行身份验证，以达到 Tiller，利用现有的 Kubernetes 身份验证支持。）

The following two sub-sections describe options of how to setup Tiller so there isn't an unauthenticated endpoint (i.e. gRPC) in your cluster.

Enabling TLS

(Note that out of the two options, this is the recommended one for Helm 2.) 共享和生产群集 - 大多数情况下 - 应至少使用 Helm 2.7.2，并为每个 Tiller gRPC 端点配置 TLS，以确保群集内 gRPC 端点的使用仅适用于该端点的正确身份验证标识。这样做可以在任意数量的 namespace 中部署任意数量的 Tiller 实例，任何 gRPC 端点未经授权不可使用。使用 Helm `init` 和 `--tiller-tls-verify` 选择安装启用 TLS 的 Tiller，并验证远程证书，所有其他 Helm 命令都应该使用该 `--tls` 选项。

有关正确配置并使用 TLS 的 Tiller 和 Helm 的正确步骤的更多信息，请参阅下面的章节 [Best Practices](#) 以及 Helm 和 Tiller 使用 SSL [在 Helm 和 Tiller 之间使用 SSL](#)。

当 Helm 客户端从群集外部连接时，Helm 客户端和 API 服务器之间的安全性由 Kubernetes 本身管理。你可能需要确保这个链接是安全的。请注意，如果使用上面建议的 TLS 配置，则 Kubernetes API 服务器也无法访问客户端和 Tiller 之间的加密消息。

Running Tiller Locally

与上面的章节 [Enabling TLS](#) 相反，本节不涉及在集群中运行分蘖服务器 pod（就其价值而言，它符合当前情况 [helm v3 proposal](#)），因此没有 gRPC 端点（因此不需要创建和管理 TLS 证书来保护每个 gRPC 端点）。

步骤:

- 获取最新安装包 [GitHub release page](#)，解压缩，并将 `helm` and `tiller` 放到你的路径 `$PATH`。
- "服务端": 运行 `tiller --storage=secret`。（`tiller` 默认监听 ":44134" 通过 `--listen` 参数。）
- "客户端": 在另一个终端 (同一台运行 `tiller` 的机器上): 运行 `export HELM_HOST=:44134`，然后运行 `helm`。

Tiller Release 信息

由于历史原因，Tiller 将其 release 信息存储在 ConfigMaps 中。我们建议将默认设置更改为 Secrets。

Secrets 是 Kubernetes 用于保存被认为是敏感的配置数据的可接受的方法。尽管 secrets 本身并不提供很多保护，但 Kubernetes 集群管理软件经常将它们与其他对象区别开来。因此，我们建议使用 secrets 来存储 release 信息。

启用此功能目前需要在 Tiller 部署时设置参数 `--storage=secret`。这需要直接修改 deployment 或使用 `helm init --override 'spec.template.spec.containers[0].command={['/tiller,--storage=secret}]'`，因为当前没有 `helm init` 参数可供执行此操作。

关于 chart

由于 Helm 的相对生命周期，Helm chart 生态系统的发展并没有考虑到整个集群的控制，这在开发人员来说，是完全合理的。但是，chart 是一种不仅可以安装可能已经验证或可能未验证的容器的包，它也可以安装到多个 namespace 中。

与所有共享的软件一样，在受控或共享的环境中，必须在安装之前验证自己安装的所有软件。如果已经通过 TLS 配置安装了 Tiller，并且只有一个或部分 namespace 的权限，某些 chart 可能无法安装 - 在这些环境中，这正是你想要的。如果需要使用 chart，可能必须与创建者一起工作或自行修改它，以便在应用了适当的 RBAC 规则的多租户群集中安全地使用它。`helm template` 命令在本地呈现 chart 并显示输出。

一旦通过检查，可以使用 Helm 的工具来确保使用的 chart 的出处和完整性 [ensure the provenance and integrity of charts](#)。

gRPC 工具和安全 Tiller 配置

许多非常有用的工具直接使用 gRPC 接口，并且已经针对默认安装构建 - 它们提供了集群范围的访问 - 一旦应用了安全配置后就可能工作不正常。RBAC 策略由你或集群运维人员控制，并且可以针对该工具进行调整，或者可以将该工具配置为，在应用于 Tiller 的特定 RBAC 策略的约束范围内来正常工作。如果 gRPC 端点受到保护，则可能需要执行相同的操作：为了使用特定的 Tiller 实例，这些工具需要自己的安全 TLS 配置。RBAC 策略和 gRPC 工具一起配置的安全 gRPC 端点的组合，使你能够按照自己的需要控制群集环境。

Helm 和 Tiller 安全最佳实践

以下指导原则重申了 Helm 和 Tiller 安全并正确使用它们的最佳方法。

1. 创建一个启用了 RBAC 的集群
2. 配置每个 Tiller gRPC 端点以使用单独的 TLS 证书
3. Release 信息应该使用 Kubernetes Secret
4. 为每个用户，团队或其他具有 `--service-account` 参数，role 和 RoleBindings 的组织安装一个 Tiller
5. `helm init` 使用 `--tiller-tls-verify`，其他 Helm 命令 `--tls` 来强制验证

如果遵循这些步骤，则 `helm init` 命令可能如下所示：

```
$ helm init \
--override 'spec.template.spec.containers[0].command={['/tiller,--storage=secret}]' \
--tiller-tls \
--tiller-tls-verify \
--tiller-tls-cert=cert.pem \
--tiller-tls-key=key.pem \
--tls-ca-cert=ca.pem \
--service-account=accountname
```

此命令将通过gRPC进行强身份验证，release信息存储在Kubernetes Secret，并使用RBAC策略的服务帐户安装启动Tiller。

Copyright © Mingo(whmzsu@gmail.com) 2017-2018 all right reserved, powered by GitbookUpdated at 2018-11-24 10:14:49

Charts

Helm 使用称为 chart 的包装格式。chart 是描述相关的一组 Kubernetes 资源的文件集合。单个 chart 可能用于部署简单的东西，比如 memcached pod，或者一些复杂的东西，比如完整的具有 HTTP 服务，数据库，缓存等的 Web 应用程序堆栈。

chart 通过创建为特定目录树的文件，将它们打包到版本化的压缩包，然后进行部署。

本文档解释了 chart 格式，提供使用 Helm 构建 chart 的基本指导。

Chart 文件结构

chart 被组织为一个目录内的文件集合。目录名称是 chart 的名称（没有版本信息）。例如，描述 WordPress 的 chart 将被存储在 wordpress / 目录中。

在这个目录里面，Helm 期望如下这样一个结构的目录树：

```
wordpress/
  Chart.yaml           # A YAML file containing information about the chart
  LICENSE              # OPTIONAL: A plain text file containing the license for the chart
  README.md            # OPTIONAL: A human-readable README file
  requirements.yaml    # OPTIONAL: A YAML file listing dependencies for the chart
  values.yaml          # The default configuration values for this chart
  charts/              # A directory containing any charts upon which this chart depends.
  templates/           # A directory of templates that, when combined with values,
                        # will generate valid Kubernetes manifest files.
  templates/NOTES.txt  # OPTIONAL: A plain text file containing short usage notes
```

Helm 保留使用 charts / 和 templates / 目录以及上面列出的文件名称。其他文件将被忽略。

Chart.yaml 文件

Chart.yaml 文件是 chart 所必需的。它包含以下字段：

```
apiVersion: The chart API version, always "v1" (required)
name: The name of the chart (required)
version: A SemVer 2 version (required)
kubeVersion: A SemVer range of compatible Kubernetes versions (optional)
description: A single-sentence description of this project (optional)
keywords:
  - A list of keywords about this project (optional)
home: The URL of this project's home page (optional)
sources:
  - A list of URLs to source code for this project (optional)
maintainers: # (optional)
  - name: The maintainer's name (required for each maintainer)
    email: The maintainer's email (optional for each maintainer)
    url: A URL for the maintainer (optional for each maintainer)
engine: gotpl # The name of the template engine (optional, defaults to gotpl)
icon: A URL to an SVG or PNG image to be used as an icon (optional).
appVersion: The version of the app that this contains (optional). This needn't be SemVer.
deprecated: Whether this chart is deprecated (optional, boolean)
tillerVersion: The version of Tiller that this chart requires. This should be expressed as a SemVer range: ">2.0.0" (optional)
```


如果熟悉 `Chart.yaml` Helm Classic 的文件格式，注意到指定依赖性的字段已被删除。这是因为新的 chart 使用 `charts/` 目录表示依赖关系。

其他字段将被忽略。

Charts 和版本控制

每个 chart 都必须有一个版本号。版本必须遵循 [SemVer 2](#) 标准。与 Helm Class 格式不同，Kubernetes Helm 使用版本号作为发布标记。存储库中的软件包由名称加版本识别。

例如，`nginx` `version` 字段设置为 1.2.3 将被命名为：

```
nginx-1.2.3.tgz
```

更复杂的 SemVer 2 命名也是支持的，例如 `version: 1.2.3-alpha.1+ef365`。但非 SemVer 命名是明确禁止的。

注意：虽然 Helm Classic 和 Deployment Manager 在 chart 方面都非常适合 GitHub，但 Kubernetes Helm 并不依赖或需要 GitHub 甚至 Git。因此，它不使用 Git SHA 进行版本控制。

许多 Helm 工具都使用 `Chart.yaml` 的 `version` 字段，其中包括 CLI 和 Tiller 服务。在生成包时，`helm package` 命令将使用它在 `Chart.yaml` 中的版本名作为包名。系统假定 chart 包名称中的版本号与 `Chart.yaml` 中的版本号相匹配。不符合这个情况会导致错误。

appVersion 字段

请注意，`appVersion` 字段与 `version` 字段无关。这是一种指定应用程序版本的方法。例如，`drupal` chart 可能有一个 `appVersion: 8.2.1`，表示 chart 中包含的 Drupal 版本（默认情况下）是 8.2.1。该字段是信息标识，对 chart 版本没有影响。

弃用 charts

在管理 chart repo 库中的 chart 时，有时需要弃用 chart。`Chart.yaml` 的 `deprecated` 字段可用于将 chart 标记为已弃用。如果存储库中最新版本的 chart 标记为已弃用，则整个 chart 被视为已弃用。chart 名称稍后可以通过发布未标记为已弃用的较新版本来重新使用。废弃 chart 的工作流程根据 [helm/charts](#) 项目的工作流程如下：

- 更新 chart 的 `Chart.yaml` 以将 chart 标记为启用，并且更新版本
- 在 chart Repository 中发布新的 chart 版本
- 从源代码库中删除 chart（例如 git）

Chart 许可证文件，自述文件和说明文件

chart 还可以包含描述 chart 的安装，配置，使用和许可证的文件。chart 的自述文件应由 Markdown（`README.md`）语法格式化，并且通常应包含：

- chart 提供的应用程序或服务的描述
- 运行 chart 的任何前提条件或要求
- 选项 `values.yaml` 和默认值的说明
- 任何其他可能与安装或配置 chart 相关的信息

chart 还可以包含一个简短的纯文本 `templates/NOTES.txt` 文件，在安装后以及查看版本状态时将打印出来。此文件将作为模板 `template` 进行评估，并可用于显示使用说明，后续步骤或任何其他与发布 chart 相关的信息。例如，可以提供用于连接到数据库或访问 Web UI 的指令。由于运行时，该文件被打印到标准输出 `helm install` 或 `helm status`，建议保持内容简短并把更多细节指向自述文件。

Chart 依赖关系

在 Helm 中，一个 chart 可能依赖于任何数量的其他 chart。这些依赖关系可以通过 requirements.yaml 文件动态链接或引入 charts/ 目录并手动管理。

虽然有一些团队需要手动管理依赖关系的优势，但声明依赖关系的首选方法是使用 chart 内部的 requirements.yaml 文件。

注意：传统 Helm 的 Chart.yaml dependencies: 部分字段已被完全删除弃用。

用 requirements.yaml 来管理依赖关系

requirements.yaml 文件是列出 chart 的依赖关系的简单文件。

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

- 该 name 字段是 chart 的名称。
- version 字段是 chart 的版本。
- repository 字段是 chart repo 的完整 URL。请注意，还必须使用 helm repo add 添加该 repo 到本地才能使用。

有了依赖关系文件，你可以通过运行 helm dependency update，它会使用你的依赖关系文件将所有指定的 chart 下载到你的 charts/ 目录中。

```
$ helm dep up foochart
Hang tight while we grab the latest from your chart repositories...
...Successfully got an update from the "local" chart repository
...Successfully got an update from the "stable" chart repository
...Successfully got an update from the "example" chart repository
...Successfully got an update from the "another" chart repository
Update Complete. Happy Helming!
Saving 2 charts
Downloading apache from repo http://example.com/charts
Downloading mysql from repo http://another.example.com/charts
```

当 helm dependency update 检索 chart 时，它会将它们作为 chart 存档存储在 charts/ 目录中。因此，对于上面的示例，可以在 chart 目录中看到以下文件：

```
charts/
  apache-1.2.3.tgz
  mysql-3.2.1.tgz
```

通过 requirements.yaml 管理 chart 是一种轻松更新 chart 的好方法，还可以在整个团队中共享 requirements 信息。

requirements.yaml 中的 alias 字段

除上述其他字段外，每个 requirement 条目可能包含可选字段 alias。

为依赖的 chart 添加别名会将 chart 放入依赖关系中，并使用别名作为新依赖关系的名称。

如果需要使用其他名称访问 chart，可以使用 alias。

```
# parentchart/requirements.yaml
dependencies:
- name: subchart
  repository: http://localhost:10191
  version: 0.1.0
  alias: new-subchart-1
- name: subchart
  repository: http://localhost:10191
  version: 0.1.0
  alias: new-subchart-2
- name: subchart
  repository: http://localhost:10191
  version: 0.1.0
```

在上面的例子中，我们将得到 parentchart 的 3 个依赖关系

```
subchart
new-subchart-1
new-subchart-2
```

实现这一目的的手动方法是 charts/ 中用不同名称多次复制 / 粘贴目录中的同一 chart。

requirements.yaml 中的 tags 和 condition 字段

除上述其他字段外，每个需求条目可能包含可选字段 tags 和 condition。

所有 charts 都会默认加载。如果存在 tags 或 condition 字段，将对它们进行评估并用于控制应用的 chart 的加载。

Condition - condition 字段包含一个或多个 YAML 路径（用逗号分隔）。如果此路径存在于顶级父级的值中并且解析为布尔值，则将根据该布尔值启用或禁用 chart。只有在列表中的第一个有效路径才被评估，如果没有路径存在，那么该条件不起作用。

Tags - 标签字段是与此 chart 关联的 YAML 标签列表。在顶级父级的值中，可以通过指定标签和布尔值来启用或禁用所有带有标签的 chart。

```
# parentchart/requirements.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
```

```
# parentchart/values.yaml
```

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

在上面的示例中，所有带有标签 `front-end` 的 charts 都将被禁用，但由于 `subchart1.enabled` 的值在父项值中为“真”，因此条件将覆盖该 `front-end` 标签，`subchart1` 会启用。

由于 `subchart2` 被标记 `back-end` 和标签的计算结果为 `true`，`subchart2` 将被启用。还要注意的，虽然 `subchart2` 有一个在 `requirements.yaml` 中指定的条件，但父项的值中没有对应的路径和值，因此条件无效。

使用命令行时带有 tag 和 conditions

`--set` 参数可使用来更改 tag 和 conditions 值。

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

tags 和 conditions 解析

- **Conditions** (设置 values) 会覆盖 tags 配置。第一个存在的 condition 路径生效，后续该 chart 的 condition 路径将被忽略。
- 如果 chart 的某 tag 的任一 tag 的值为 true，那么该 tag 的值为 true，并启用这个 chart。
- Tags 和 conditions 值必须在顶级父级的值中进行设置。
- `tags:` 值中的关键字必须是顶级关键字。目前不支持全局和嵌套 `tags:` 表格。

通过 requirements.yaml 导入子值

在某些情况下，希望允许子 chart 的值传到父 chart 并作为通用默认值共享。使用这种 exports 格式的另一个好处是它可以使未来的工具能够考虑用户可设置的值。

要导入的值的键可以在父 chart 文件中 `requirements.yaml` 使用 YAML list 指定。list 中的每个项目都是从子 chart `exports` 字段导入的 key。

要导入不包含在 exports key 中的值，请使用子父级 `child-parent` 格式。下面描述了两种格式的例子。

使用 exports 格式

如果子 chart 的 values.yaml 文件 exports 在根目录中包含一个字段，则可以通过指定要导入的关键字将其内容直接导入到父项的值中，如下例所示：

```
# parent's requirements.yaml file
...
import-values:
  - data

# child's values.yaml file
...
exports:
  data:
    myint: 99
```

由于我们在导入列表中指定了 `data` 键，因此 Helm 会在 exports 子图的字段中查找 data 键并导入其内容。

最终的父值将包含我们的导出字段：

```
# parent's values file
...
myint: 99
```

请注意，父键 data 不包含在父 chart 的最终值中。如果需要指定父键，请使用 `'child-parent'` 格式。

使用 child-parent 格式

要访问未包含在子 chart 键值 `exports` 中的值，需要指定要导入的值的源键（`child`）和父 chart 值（`parent`）中的目标路径。

下面的例子中的 `import-values` 告诉 Helm 去拿在 `child:` 路径发现的任何值，并将其复制到父值 `parent:` 指定的路径

```
# parent's requirements.yaml file
dependencies:
- name: subchart1
  repository: http://localhost:10191
  version: 0.1.0
...
import-values:
- child: default.data
  parent: myimports
```

在上面的例子中，在 `subchart1 default.data` 的值中找到的值将被导入到父 chart 值中 `myimports` 的键值，详细如下：

```
# parent's values.yaml file

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

# subchart1's values.yaml file

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

父 chart 的结果值为：

```
# parent's final values

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

父 chart 的最终值现在包含从 `subchart1` 导入的 `myint` 和 `mybool` 字段。

通过 `charts/` 目录手动管理依赖性

如果需要更多的控制依赖关系，可以通过将依赖的 charts 复制到 `charts/` 目录中来明确表达这些依赖关系。

依赖关系可以是 chart 归档（`foo-1.2.3.tgz`）或解压缩的 chart 目录。但它的名字不能从 `_` 或 `.` 开始。这些文件被 chart 加载器忽略。

例如，如果 WordPress chart 依赖于 Apache chart，则在 WordPress chart 的 `charts/` 目录中提供（正确版本的）Apache chart：

```
wordpress:
  Chart.yaml
  requirements.yaml
# ...
```

```
charts/  
  apache/  
    Chart.yaml  
    # ...  
  mysql/  
    Chart.yaml  
    # ...
```

上面的示例显示了 WordPress chart 如何通过在其 `charts/` 目录中包含这些 charts 来表示它对 Apache 和 MySQL 的依赖关系。

提示：将依赖项放入 `charts/` 目录，请使用 `helm fetch` 命令

使用依赖关系的操作方面影响

上面的部分解释了如何指定 chart 依赖关系，但是这会如何影响使用 `helm install` 和 `helm upgrade` 的 chart 安装？

假设为“A”的 chart 创建以下 Kubernetes 对象

- namespace "A-Namespace"
- statefulset "A-StatefulSet"
- service "A-Service"

此外，A 依赖于创建对象的 chart B.

- namespace "B-Namespace"
- replicaset "B-ReplicaSet"
- service "B-Service"

安装 / 升级 chart A 后，会创建 / 修改单个 Helm 版本。该版本将按以下顺序创建 / 更新所有上述 Kubernetes 对象：

- A-Namespace
- B-Namespace
- A-StatefulSet
- B-ReplicaSet
- A-Service
- B-Service

这是因为当 Helm 安装 / 升级 charts 时，charts 中的 Kubernetes 对象及其所有依赖项都是如下

- 聚合成一个单一的集合; 然后
- 按类型排序，然后按名称排序; 接着
- 按该顺序创建 / 更新。

因此，单个 release 是使用 charts 及其依赖关系创建的所有对象。

Kubernetes 类型的安装顺序由 `kind_sorter.go` 中的枚举 `InstallOrder` 给出 ([the Helm source file](#))。

模板 Templates 和值 Values

Helm chart 模板是用 Go 模板语言 [Go template language](#) 编写的，其中添加了来自 Sprig 库 [from the Sprig library](#) 的 50 个左右的附加模板函数以及一些其他专用函数 [specialized functions](#)。

所有模板文件都存储在 chart 的 `templates/` 文件夹中。当 Helm 渲染 charts 时，它将通过模板引擎传递该目录中的每个文件。

模板的值有两种提供方法：

- chart 开发人员可能会在 chart 内部提供一个 `values.yaml` 文件。该文件可以包含默认值。

- chart 用户可能会提供一个包含值的 YAML 文件。这可以通过命令行提供 `helm install -f`。

当用户提供自定义值时，这些值将覆盖 chart 中 `values.yaml` 文件中的值。

模板文件

模板文件遵循用于编写 Go 模板的标准约定（请参阅文 [the text/template Go package documentation](#) 以了解详细信息）。示例模板文件可能如下所示：

```
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}}
```

上面的示例基于 [此网址](#)，是 Kubernetes replication controller 的模板。它可以使用以下四个模板值（通常在 `values.yaml` 文件中定义）：

- `imageRegistry`：Docker 镜像的源。
- `dockerTag`：docker 镜像的标签。
- `pullPolicy`：Kubernetes 镜像拉取策略。
- `storage`：存储后端，其默认设置为 `"minio"`

所有这些值都由模板作者定义。Helm 不需要或指定参数。

要查更多 charts，请查看 Kubernetes charts 项目

预定义值

通过 `values.yaml` 文件（或通过 `--set` 标志）提供的值可以从 `.Values` 模板中的对象访问。可以在模板中访问其他预定义的数据片段。

以下值是预定义的，可用于每个模板，并且不能被覆盖。与所有值一样，名称区分大小写。

- `Release.Name`：release 的名称（不是 chart 的）
- `Release.Time`：chart 版本上次更新的时间。这将匹配 `Last Released` 发布对象上的时间。
- `Release.Namespace`：chart release 发布的 namespace。
- `Release.Service`：处理 release 的服务。通常是 Tiller。
- `Release.IsUpgrade`：如果当前操作是升级或回滚，则设置为 true。

- `Release.IsInstall` : 如果当前操作是安装, 则设置为 `true`。
- `Release.Revision` : 版本号。它从 1 开始, 并随着每个 `helm upgrade` 增加。
- `Chart` : `Chart.yaml` 的内容。chart 版本可以从 `Chart.Version` 和维护人员 `Chart.Maintainers` 一起获得。
- `Files` : 包含 chart 中所有非特殊文件的 map-like 对象。不会允许你访问模板, 但会让你访问存在的其他文件 (除非它们被排除使用 `.helmignore`)。可以使用 `index.Files "file.name"` 或使用 `Files.Get name` 或 `Files.GetString name` 功能来访问文件。也可以使用 `Files.GetBytes` 访问该文件的内容 `[byte]`
- `Capabilities` : 包含有关 Kubernetes 版本信息的 map-like 对象 (`.Capabilities.KubeVersion`), Tiller (`.Capabilities.TillerVersion`) 和支持的 Kubernetes API 版本 (`.Capabilities.APIVersions.Has "batch/v1"`)

注意: 任何未知的 `Chart.yaml` 字段将被删除。它们不会在 chart 对象内部被访问。因此, `Chart.yaml` 不能用于将任意结构化的数据传递到模板中。`values` 文件可以用于传递。

值 values 文件

考虑到上一节中的模板 `values.yaml`, 提供了如下必要值的信息:

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

`values` 文件是 YAML 格式的。chart 可能包含一个默认 `values.yaml` 文件。Helm `install` 命令允许用户通过提供额外的 YAML 值来覆盖值:

```
$ helm install --values=myvals.yaml wordpress
```

当以这种方式传递值时, 它们将被合并到默认 `values` 文件中。例如, 考虑一个如下所示的 `myvals.yaml` 文件:

```
storage: "gcs"
```

当它与 chart 中 `values.yaml` 的内容合并时, 生成的内容将为:

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

注意只有最后一个字段被覆盖了, 其他的不变。

注: 包含在 chart 内的默认 `values` 文件必须命名 `values.yaml`。但是在命令行上指定的文件可以被命名为任何名称。

注: 如果在 `helm install` 或 `helm upgrade` 使用 `--set`, 则这些值仅在客户端转换为 YAML。

注意: 如果 `values` 文件中存在任何必需的条目, 则可以使用 'required' 功能 ['required' function](#) 在 chart 模板中声明它们

然后可以在模板内部访问任何这些 `.Values` 对象值:

```
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}}

```

范围 Scope，依赖 Dependencies 和值 Values

values 文件可以声明顶级 chart 的值，也可以为 chart 的 charts / 目录中包含的任何 chart 声明值。或者，用不同的方式来描述它，values 文件可以为 chart 及其任何依赖项提供值。例如，上面的演示 WordPresschart 具有 mysql 和 apache 依赖性。values 文件可以为所有这些组件提供值：

```

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

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

apache:
  port: 8080 # Passed to Apache

```

更高级别的 chart 可以访问下面定义的所有变量。所以 WordPresschart 可以访问 MySQL 密码 `.Values.mysql.password`。但较低级别的 chart 无法访问父 chart 中的内容，因此 MySQL 将无法访问该 `title` 属性。同样的，也不能访问 `apache.port`。

值是命名空间限制的，但命名空间已被修剪。因此对于 WordPresschart 来说，它可以访问 MySQL 密码字段 `.Values.mysql.password`。但是对于 MySQL chart 来说，这些值的范围已经减小了，并且删除了名 namespace 前缀，所以它会将密码字段简单地视为 `.Values.password`。

全局值

从 2.0.0-Alpha.2 开始，Helm 支持特殊的“全局”值。考虑前面例子的这个修改版本：

```

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

```

上面添加了一个 global 区块，值 `app: MyWordPress`。此值可供所有 chart 使用 `.Values.global.app`。

比如，该 mysql 模板可以访问 `app` 如 `.Values.global.app`，apache chart 也同样的。上面的 values 文件是这样高效重新生成的：


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

global:
  app: MyWordPress

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

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

这提供了一种与所有子 chart 共享一个顶级变量的方法，这对设置 metadata 中像标签这样的属性很有用。

如果子 chart 声明了一个全局变量，则该全局将向下传递（到子 chart 的子 chart），但不向上传递到父 chart。子 chart 无法影响父 chart 的值。

此外，父 chart 的全局变量优先于子 chart 中的全局变量。

参考

当涉及到编写模板和 values 文件时，有几个标准参考可以帮助你。

- [Go templates](#)
- [Extra template functions](#)
- [The YAML format](#)

使用 Helm 管理 chart

该 helm 工具有几个用于处理 chart 的命令。

它可以为你创建一个新的 chart：

```
$ helm create mychart
Created mychart/
```

编辑完 chart 后，helm 可以将其打包到 chart 压缩包中：

```
$ helm package mychart
Archived mychart-0.1.0.tgz
```

可以用 helm 来帮助查找 chart 格式或信息的问题：

```
$ helm lint mychart
No issues found
```

Chart repo 库

chart repo 库是容纳一个或多个封装的 chart 的 HTTP 服务器。虽然 helm 可用于管理本地 chart 目录，但在共享 chart 时，首选机制是 chart repo 库。

任何可以提供 YAML 文件和 tar 文件并可以回答 GET 请求的 HTTP 服务器都可以用作 repo 库服务器。

Helm 附带有用于开发人员测试的内置服务器 (`helm serve`)。Helm 团队测试了其他服务器，包括启用了网站模式的 Google Cloud Storage 以及启用了网站模式的 S3。

repo 库的主要特征是存在一个名为的特殊文件 `index.yaml`，它具有 repo 库提供的所有软件包的列表以及允许检索和验证这些软件包的元数据。

在客户端，repo 库使用 `helm repo` 命令进行管理。但是，Helm 不提供将 chart 上传到远程存储服务器的工具。这是因为这样做会增加部署服务器的需求，从而增加配置 repo 库的难度。

Chart 起始包

`helm create` 命令采用可选 `--starter` 选项，可以指定“起始 chart”。

起始 chart 只是普通的 chart，位于 `$HELM_HOME/starters`。作为 chart 开发人员，可以创作专门设计用作起始的 chart。记住这些 chart 时应考虑以下因素：

- `Chart.yaml` 将被生成器覆盖。
- 用户将期望修改这样的 chart 内容，因此文档应该指出用户如何做到这一点。
- 所有 `templates` 目录下的匹配项 `<CHARTNAME>` 将被替换为指定的 chart 名称，以便起始 chart 可用作模板。另外，`values.yaml` 的 `<CHARTNAME>` 也会被替换。
- 目前添加 chart 的唯一方法是手动将其复制到 `$HELM_HOME/starters`。在 chart 的文档中，你需要解释该过程。

Copyright © Mingo(whmzsu@gmail.com) 2017-2018 all right reserved，powered by GitbookUpdated at 2018-11-24 09:58:32

Hooks

Helm 提供了一个 hook 机制，允许 chart 开发人员在 release 的生命周期中的某些点进行干预。例如，可以使用 hooks 来：

- 在加载任何其他 chart 之前，在安装过程中加载 ConfigMap 或 Secret。
- 在安装新 chart 之前执行作业以备份数据库，然后在升级后执行第二个作业以恢复数据。
- 在删除 release 之前运行作业，以便在删除 release 之前优雅地停止服务。

Hooks 像常规模板一样工作，但它们具有特殊的注释，可以使 Helm 以不同的方式使用它们。在本节中，我们介绍 Hooks 的基本使用模式。

可用的 Hooks

定义了以下 hooks：

- 预安装 pre-install：在模板渲染后执行，但在 Kubernetes 中创建任何资源之前执行。
- 安装后 post-install：在所有资源加载到 Kubernetes 后执行
- 预删除 pre-delete：在从 Kubernetes 删除任何资源之前执行删除请求。
- 删除后 post-delete：删除所有 release 的资源后执行删除请求。
- 升级前 pre-upgrade：在模板渲染后，但在任何资源加载到 Kubernetes 之前执行升级请求（例如，在 Kubernetes 应用操作之前）。
- 升级后 post-upgrade：在所有资源升级后执行升级。
- 预回滚 pre-rollback：在渲染模板之后，但在任何资源已回滚之前，在回滚请求上执行。
- 回滚后 post-rollback：在修改所有资源后执行回滚请求。

Hooks 和 release 的生命周期

Hooks 让 chart 开发人员有机会在 release 的生命周期中的关键点执行操作。例如，考虑 a 的生命周期。默认情况下，生命周期如下所示：

- 用户运行 `helm install foo`
- chart 被加载到 Tiller 中
- 经过一些验证后，Tiller 渲染 foo 模板
- Tiller 将产生的资源加载到 Kubernetes 中
- Tiller 将 release 名称（和其他数据）返回给客户端
- 客户端退出

Helm 为 install 生命周期定义了两个 hook：`pre-install` 和 `post-install`。如果 `foo` chart 的开发者实现了两个 hook，那么生命周期就像这样改变：

- 用户运行 `helm install foo`
- chart 被加载到 Tiller 中
- 经过一些验证后，Tiller 渲染 `foo` 模板
- Tiller 准备执行 `pre-install` hook（将 hook 资源加载到 Kubernetes 中）
- Tiller 会根据权重对 hook 进行排序（默认分配权重 0），并按相同权重的 hook 按升序排序。
- Tiller 然后装载最低权重的 hook（从负到正）
- Tiller 等待，直到 hook“准备就绪”
- Tiller 将产生的资源加载到 Kubernetes 中。请注意，如果设置 `--wait` 标志，Tiller 将等待，直到所有资源都处于就绪状态，并且在准备就绪之前不会运行 `post-install` hook。

- Tiller 执行 `post-install` hook (加载 hook 资源)
- Tiller 等待, 直到 hook“准备就绪”
- Tiller 将 release 名称 (和其他数据) 返回给客户端
- 客户端退出

等到 hook 准备就绪是什么意思？这取决于在 hook 中声明的资源。如果资源是 Job 者一种资源, Tiller 将等到作业成功完成。如果作业失败, 则发布失败。这是一个阻塞操作, 所以 Helm 客户端会在 Job 运行时暂停。

对于所有其他类型, 只要 Kubernetes 将资源标记为加载 (添加或更新), 资源就被视为“就绪”。当一个 hook 声明了很多资源时, 这些资源将被串行执行。如果他们有 hook 权重 (见下文), 他们按照加权顺序执行。否则, 订购过程不能保证。(在 Helm 2.3.0 及之后的版本中, 它们按字母顺序排列, 但这种行为并未被视为具有约束力, 将来可能会发生变化)。添加挂钩权重被认为是很好的做法, 并将其设置为 0 如果权重不是重要。

Hook 资源不与相应的 release 一起进行管理

Hook 创建的资源不作为 release 的一部分进行跟踪或管理。一旦 Tiller 验证 hook 已经达到其就绪状态, 它将 hook 资源放在一边。

实际上, 这意味着如果在 hook 中创建资源, 则不能依赖于 `helm delete` 删除资源。要销毁这些资源, 需要编写代码在 `pre-delete` 或 `post-delete` hook 中执行此操作, 或者将 `"helm.sh/hook-delete-policy"` 注释添加到 hook 模板文件。

写一个 hook

Hook 只是 Kubernetes manifest 文件, 在 metadata 部分有特殊的注释。因为他们是模板文件, 可以使用所有的 Normal 模板的功能, 包括读取 `.Values`, `.Release` 和 `.Template`。

例如, 在此模板中, 存储在 `templates/post-install-job.yaml` 的声明要在 `post-install` 阶段运行作业:

```
apiVersion: batch/v1
kind: Job
metadata:
  name: "{{.Release.Name}}"
  labels:
    app.kubernetes.io/managed-by: {{.Release.Service | quote}}
    app.kubernetes.io/instance: {{.Release.Name | quote}}
    helm.sh/chart: "{{.Chart.Name}}-{{.Chart.Version}}"
  annotations:
    # This is what defines this resource as a hook. Without this line, the
    # job is considered part of the release.
    "helm.sh/hook": post-install
    "helm.sh/hook-weight": "-5"
    "helm.sh/hook-delete-policy": hook-succeeded
spec:
  template:
    metadata:
      name: "{{.Release.Name}}"
      labels:
        app.kubernetes.io/managed-by: {{.Release.Service | quote}}
        app.kubernetes.io/instance: {{.Release.Name | quote}}
        helm.sh/chart: "{{.Chart.Name}}-{{.Chart.Version}}"
    spec:
      restartPolicy: Never
      containers:
        - name: post-install-job
          image: "alpine:3.3"
          command: ["/bin/sleep", "{{default \"10\" .Values.sleepyTime}}"]
```

注释使这个模板成为 hook:

```

annotations:
  "helm.sh/hook": post-install

```

一个资源可以部署多个 hook :

```

annotations:
  "helm.sh/hook": post-install,post-upgrade

```

同样,实现一个给定的 hook 的不同种类资源数量没有限制。例如,我们可以将 secret 和 config map 声明为预安装 hook。

子 chart 声明 hook 时,也会评估这些 hook。顶级 chart 无法禁用子 chart 所声明的 hook。

可以为一个 hook 定义一个权重,这将有助于建立一个确定性的执行顺序。权重使用以下注释来定义:

```

annotations:
  "helm.sh/hook-weight": "5"

```

hook 权重可以是正数或负数,但必须表示为字符串。当 Tiller 开始执行一个特定类型的 hook (例: `pre-install` hooks `post-install` hooks, 等等) 执行周期时,它会按升序对这些 hook 进行排序。

还可以定义确定何时删除相应的 hook 资源的策略。hook 删除策略使用以下注释来定义:

```

annotations:
  "helm.sh/hook-delete-policy": hook-succeeded

```

可以选择一个或多个定义的注释值:

- "hook-succeeded" 指定 Tiller 应该在 hook 成功执行后删除 hook。
- "hook-failed" 指定如果 hook 在执行期间失败, Tiller 应该删除 hook。
- "before-hook-creation" 指定 Tiller 应在删除新 hook 之前删除以前的 hook。

自动删除以前版本的 hook

当 helm 的 release 更新时,有可能 hook 资源已经存在于群集中。默认情况下, helm 会尝试创建资源,并抛出错误 "...already exists"。

我们可以选择 `"helm.sh/hook-delete-policy": "before-hook-creation"`, 取代 `"helm.sh/hook-delete-policy": "hook-succeeded, hook-failed"` 因为:

- 例如为了手动调试,将错误的 hook 作业资源保存在 kubernetes 中是很方便的。
- 出于某种原因,可能有必要将成功的 hook 资源保留在 kubernetes 中。
- 同时,在 helm release 升级之前进行手动资源删除是不可取的。

`"helm.sh/hook-delete-policy": "before-hook-creation"` 在 hook 中的注释,如果新的 hook 启动前有一个 hook 的话,会使 Tiller 将以前的 release 中的 hook 删除,而这个 hook 同时它可能正在被其他一个策略使用。

Copyright © Mingo(whmzsu@gmail.com) 2017-2018 all right reserved, powered by Gitbook Updated at 2018-11-23 22:31:49

Chart 开发 Tips 和 Tricks

本指南涵盖了 Helm chart 开发人员在构建生产级质量的 chart 时学到的一些提示和技巧。

了解模板函数

Helm 使用 Go 模板 [Go templates](#) 来模板化你的资源文件。虽然 Go 提供了几个内置函数，但我们添加了许多其他函数。

首先，我们在 Sprig 库 [Sprig library](#) 中添加了几几乎所有的函数。出于安全原因，我们删除了两个：`env` 和 `expandenv`（这会让 chart 作者访问 Tiller 的环境）。

我们还添加了两个特殊的模板函数：`include` 和 `required`。`include` 函数允许引入另一个模板，然后将结果传递给其他模板函数。

例如，该模板片段包含一个调用的模板 `mytpl`，然后将结果小写，然后用双引号将其包起来。

```
value: {{include "mytpl" . | lower | quote}}
```

`required` 函数允许根据模板渲染的要求声明特定的值条目。如果该值为空，则模板渲染将失败并显示用户提交的错误消息。

下面的 `required` 函数示例声明了 `.Values.who` 的条目是必需的，并且在缺少该条目时将显示错误消息：

```
value: {{required "A valid .Values.who entry required!" .Values.who}}
```

引用字符串，不要引用整数

当使用字符串数据时，引用字符串比把它们留为空白字符更安全：

```
name: {{.Values.MyName | quote}}
```

但是，使用整数时不要引用值。在很多情况下，这可能会导致 Kubernetes 内部的解析错误。

```
port: {{.Values.Port}}
```

这种做法不适用于预期为字符串的 `env` 变量值，即使它们表示为整数：

```
env:
  -name: HOST
    value: "http://host"
  -name: PORT
    value: "1234"
```

使用 'include' 函数

Go 提供了一种使用内置 `template` 指令将一个模板包含在另一个模板中的方法。但是，Go 模板管道中不能使用内置函数。

为了能够包含模板，然后对该模板的输出执行操作，Helm 有一个特殊的 `include` 函数：

```
{{- include "toYaml" $value | nindent 2}}
```

上面包含一个名为的模板 `toYaml`，传递它 `$value` 的值，然后将该模板的输出传递给该 `indent` 函数。

因为 YAML 的缩进级别和空白的很重要，所以这是包含代码片段的好方法，并在相关的上下文中处理缩进。

使用'required'函数

Go 提供了一种设置模板选项以控制 `map` 使用 `map` 中不存在的键编制索引时的行为的方法。这通常使用 `template.Options`（“missingkey = option”）设置，其中 `option` 可以是默认值，零或错误。将此选项设置为错误将停止执行并出现错误，这应用于 `map` 中每个缺失的键。这适用于 `chart` 开发人员想要强制为 `values.yml` 文件选择值来实施此行为的情况。

该 `required` 函数使开发人员能够根据模板渲染的要求声明值条目。如果 `values.yml` 中的条目为空，模板将不会渲染，并会返回开发人员提供的错误消息。

例如：

```
{{required "A valid foo is required!" .Values.foo}}
```

上面将在定义 `Values.foo` 时渲染模板，但在未定义 `Values.foo` 时无法渲染并报错退出。

使用'tpl' 函数

`tpl` 函数允许开发人员将字符串计算为模板内的模板。这对于将模板字符串作为值传递给 `chart` 或渲染外部配置文件很有用。

语法: `{{tpl TEMPLATE_STRING VALUES}}`

样例:

```
# values
template: "{{.Values.name}}"
name: "Tom"

# template
{{tpl .Values.template .}}

# output
Tom
```

渲染一个外部配置文件:

```
# external configuration file conf/app.conf
firstName={{.Values.firstName}}
lastName={{.Values.lastName}}

# values
firstName: Peter
lastName: Parker

# template
{{tpl (.Files.Get "conf/app.conf") . }}

# output
firstName=Peter
lastName=Parker
```

创建镜像拉取的 Secrets

镜像拉的 secrets 实质上是注册，用户名和密码的组合。在正在部署的应用程序中可能需要它们，但要创建它们需要多次运行 base64。我们可以编写一个帮助程序模板来组合 Docker 配置文件，以用作 Secret 的有效载体。这里是一个例子：

首先，假设凭证在 `values.yaml` 文件中定义如下：

```
imageCredentials:
  registry: quay.io
  username: someone
  password: sillyness
```

然后我们定义我们的帮助模板如下：

```
{{- define "imagePullSecret"}}
{{- printf "\auths\": {\\"%s\": {\\"auth\": \\"%s\\"}}}" .Values.imageCredentials.registry (printf "%s:%s" .Values.imageCredentials.username .Values.imageCredentials.password | b64enc) | b64enc }}
{{- end}}
```

最后，我们在更大的模板中使用助手模板来创建 Secret manifest：

```
apiVersion: v1
kind: Secret
metadata:
  name: myregistrykey
type: kubernetes.io/dockerconfigjson
data:
  .dockerconfigjson: {{template "imagePullSecret" .}}
```

ConfigMaps 或 Secrets 更改时自动 Roll Deployments

通常情况下，configmaps 或 secrets 被作为配置文件注入容器中。根据应用程序的不同，可能需要重新启动才能使用后续更新 `helm upgrade`，但如果 deployment spec 本身未更改，则应用程序会一直以旧配置运行，导致 deployment 不一致。

该 `sha256sum` 函数可用于确保在一个文件发生更改时更新 deployment 的注释部分：

```
kind: Deployment
spec:
  template:
    metadata:
      annotations:
        checksum/config: {{include (print $.Template.BasePath "/configmap.yaml") . | sha256sum }}
[...]
```

另请参阅 `helm upgrade --recreate-pods` 标志以解决此问题的稍微不同的方式。

告诉 Tiller 不要删除资源

有时候有一些资源在 Helm 运行 `helm delete` 时不应该被删除。chart 开发人员可以将注释添加到资源以防止被删除。

```
kind: Secret
metadata:
```



```
annotations:
  "helm.sh/resource-policy": keep
[...]
```

（需要双引号）

注释 `"helm.sh/resource-policy": keep` 指示 Tiller 在 `helm delete` 操作过程中跳过此资源。但是，此资源变成孤儿资源。Helm 将不再以任何方式管理它。如果 `helm install --replace` 在已被删除的 release 上使用，但保留了资源，则这可能会引发问题。

使用“Partials”和 includes 模板

有时候你想在 chart 中创建一些可重用的部分，无论它们是块还是模板部分。通常，将这些文件保存在自己的文件中会更整洁。

在 `templates/` 目录中，任何以下划线“-”开头的文件都不会输出 Kubernetesmanifest 文件。另按照惯例，辅助模板和 partials 被放置在一个 `_helpers.tpl` 文件中。

具有许多依赖关系的复杂 chart

官方 chart repo 存储库 [official charts repository](#) 中的许多 chart 是用于创建更高级应用程序的“构建块”。chart 可能用于创建大规模应用程序的实例。在这种情况下，一张伞形 chart 可能有多个子 chart，每个子 chart 都是整体的一部分。

当前最佳做法是：从各个子 chart 组成复杂应用程序，创建公开全局配置的顶层伞形 chart，然后使用 `charts/` 子目录嵌入每个组件 chart。

下面的项目说明了两种强大的设计模式：

SAP's OpenStack chart: 该 chart 在 Kubernetes 上安装完整的 OpenStack IaaS。所有 chart 都收集在一个 GitHub 存储库中。

Deis's Workflow: 该 chart 显示了整个 Deis PaaS 系统的一个 chart。但与 SAP chart 不同的是，该伞形 chart 是从每个组件构建而来的，每个组件都在不同的 Git 存储库中进行跟踪。查看 `requirements.yaml` 文件以查看此 chart 是如何由其 CI/CD 流水线组成的。

这两个 chart 都说明了使用 Helm 建立复杂环境的成熟技术。

YAML 是 JSON 的超级 Superset

根据 YAML 规范，YAML 是 JSON 的超集。这意味着任何有效的 JSON 结构都应该在 YAML 中有效。

这有一个优点：有时模板开发人员可能会发现使用类似 JSON 的语法来表示数据结构更容易，而不是处理 YAML 的空白敏感度。

作为最佳实践，模板应遵循类似 YAML 的语法，除非 JSON 语法大幅降低了格式问题的风险。

小心随机值生成

Helm 中有一些函数允许生成随机数据，加密密钥等。这些都很好用。但请注意，在升级过程中，模板会被重新执行。当模板运行产生与上次运行不同的数据时，将触发该资源的更新。

升级一个 release 的 idempotently

为了在安装和升级发行版时使用相同的命令，请使用以下命令：

```
helm upgrade --install <release name> --values <values file> <chart directory>
```

Copyright © Mingo(whmzs@gmail.com) 2017-2018 all right reserved , powered by GitbookUpdated at 2018-11-24 10:20:56

Chart Repository 存储库指南

本节介绍如何创建和使用 Helm chart repo。在高层次上，chart 库是可以存储和共享打包 chart 的位置。

官方 chart 库由 [Helm Charts](#) 维护，我们欢迎参与贡献。Helm 还可以轻松创建和运行自己的 chart 库。本指南讲解了如何做到这一点。

前提条件

- 阅读快速入门指南 [Quickstart](#)
- 通读 chart 文件 [Charts](#)

创建 chart 库

chart 库是带有一个 index.yaml 文件和任意个打包 chart 的 HTTP 服务器。当准备好分享 chart 时，首选方法是将其上传到 chart 库。

注意：对于 Helm 2.0.0，chart 库没有任何内部认证。在 GitHub 中有一个跟踪进度的问题 [issue tracking progress](#)。

由于 chart 库可以是任何可以提供 YAML 和 tar 文件并可以回答 GET 请求的 HTTP 服务器，因此当托管自己的 chart 库时，很多选择。例如，可以使用 Google 云端存储（GCS）存储桶，Amazon S3 存储桶，Github Pages，甚至可以创建自己的 Web 服务器。

chart 库结构

chart 库由打包的 chart 和一个名为的特殊文件组成，index.yaml 其中包含 chart 库中所有 chart 的索引。通常，index.yaml 描述的 chart 也是托管在同一台服务器上，源代码文件也是如此。

例如，chart 库的布局 <https://example.com/charts> 可能如下所示：

```
charts/
|
|- index.yaml
|
|- alpine-0.1.2.tgz
|
|- alpine-0.1.2.tgz.prov
```

这种情况下，索引文件包含有关一个 chart（Alpine chart）的信息，并提供该 chart 的下载 URL

`https://example.com/charts/alpine-0.1.2.tgz`。

不要求 chart 包与 index.yaml 文件位于同一台服务器上。但是，发在一起这样做通常是最简单的。

索引文件

索引文件是一个叫做 yaml 文件 index.yaml。它包含一些关于包的元数据，包括 chart 的 Chart.yaml 文件的内容。一个有效的 chart 库必须有一个索引文件。索引文件包含有关 chart 库中每个 chart 的信息。`helm repo index` 命令将根据包含打包的 chart 的给定本地目录生成索引文件。

下面一个索引文件的例子：

```
apiVersion: v1
entries:
```

```

alpine:
- created: 2016-10-06T16:23:20.499814565-06:00
  description: Deploy a basic Alpine Linux pod
  digest: 99c76e403d752c84ead610644d4b1c2f2b453a74b921f422b9dcb8a7c8b559cd
  home: https://k8s.io/helm
  name: alpine
  sources:
  - https://github.com/helm
  urls:
  - https://technosophos.github.io/tscharts/alpine-0.2.0.tgz
  version: 0.2.0
- created: 2016-10-06T16:23:20.499543808-06:00
  description: Deploy a basic Alpine Linux pod
  digest: 515c58e5f79d8b2913a10cb400ebb6fa9c77fe813287afbacf1a0b897cd78727
  home: https://k8s.io/helm
  name: alpine
  sources:
  - https://github.com/helm
  urls:
  - https://technosophos.github.io/tscharts/alpine-0.1.0.tgz
  version: 0.1.0
nginx:
- created: 2016-10-06T16:23:20.499543808-06:00
  description: Create a basic nginx HTTP server
  digest: aaff4545f79d8b2913a10cb400ebb6fa9c77fe813287afbacf1a0b897cdffffff
  home: https://k8s.io/helm
  name: nginx
  sources:
  - https://github.com/helm/charts
  urls:
  - https://technosophos.github.io/tscharts/nginx-1.1.0.tgz
  version: 1.1.0
generated: 2016-10-06T16:23:20.499029981-06:00

```

生成的索引和包可以从基本的网络服务器提供。可以使用 `helm serve` 启动本地服务器，在本地测试所有内容。

```

$ helm serve --repo-path ./charts
Regenerating index. This may take a moment.
Now serving you on 127.0.0.1:8879

```

上面启动了一个本地 web 服务器，为它在 `./charts` 目录找到的 chart 提供服务。serve 命令将在启动过程中自动生成一个 `index.yaml` 文件。

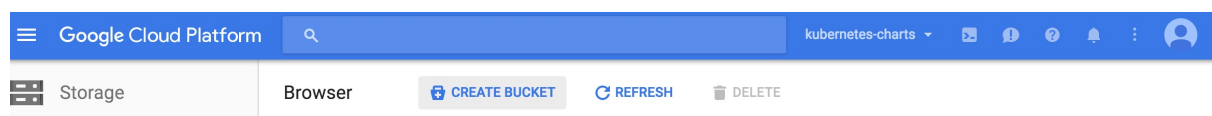
托管 chart 库

本部分介绍了提供 chart 库的几种方法。

Google 云端存储

第一步是创建 GCS 存储桶。我们会给我们称之为 `fantastic-charts`。

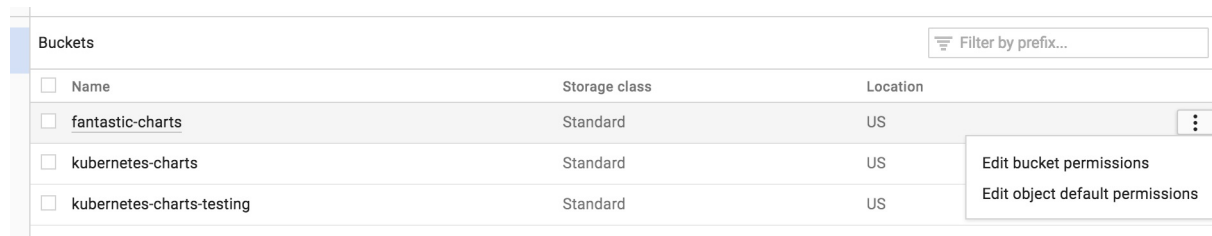
创建一个 GCS 桶



图片 - Create a GCS Bucket

接下来，通过 编辑存储桶权限 使存储桶公开。

编辑权限



<input type="checkbox"/>	Name	Storage class	Location	
<input type="checkbox"/>	fantastic-charts	Standard	US	⋮
<input type="checkbox"/>	kubernetes-charts	Standard	US	
<input type="checkbox"/>	kubernetes-charts-testing	Standard	US	

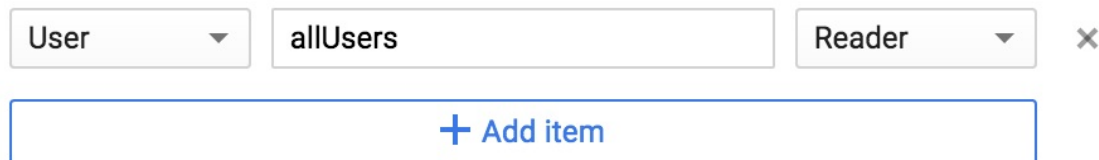
Edit bucket permissions

Edit object default permissions

图片 - Edit Permissions

插入此行 item 来 公开存储 bucket：

开放 Bucket



User Reader

+ Add item

图片 - Make Bucket Public

恭喜，现在你有一个空的 GCS bucket 准备好给 chart 提供服务！

可以使用 Google Cloud Storage 命令行工具或使用 GCS Web UI 上传 chart 库。这是官方 Kubernetes Charts 存储库托管其 chart 的技术，因此如果遇到困 难，可能需要查看该项目 [peek at that project](#)。

注意：可以通过此处的 HTTPS 地址方便的访问公开的 GCS 存储桶 `https://bucket-name.storage.googleapis.com/`。

JFrog Artifactory

还可以使用 JFrog Artifactory 来设置 chart 库。在 [此处](#) 阅读更多关于 JFrog Artifactory 和 chart 库的信息

Github Pages 示例

以类似的方式，可以使用 GitHub Pages 创建 chart 库。

GitHub 允许两种不同的方式提供静态网页：

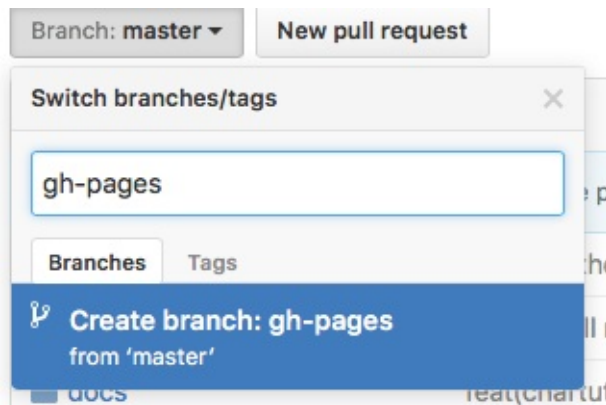
- 通过配置一个项目来提供其 ``docs/`` 目录的内容
- 通过配置一个项目来为特定分支提供服务

我们会采取第二种方法，尽管第一种方法很简单。

第一步是创建你的 gh-pages 分支。可以在本地做到这一点。

```
$ git checkout -b gh-pages
```

或者通过使用网络浏览器在 Github 存储库上的分支按钮：



图片 - Create Github Pages branch

接下来，需要确保 gh-pages 分支设置为 Github Pages，点击 repo Settings 并向下滚动到 Github Pages 部分并按照以下设置：

GitHub Pages

Your site is ready to be published at <https://rimusz-lab.github.io/charts/>.

GitHub Pages is designed to host your personal, organization, or project pages from a GitHub repository.

Source

Your GitHub Pages site is currently being built from the `gh-pages` branch. [Learn more.](#)

gh-pages branch ▾

Save

Custom domain

Custom domains allow you to serve your site from a domain other than `rimusz-lab.github.io`. [Learn more.](#)

Save

Update your site

To update your site, push your HTML or [Jekyll](#) updates to the `gh-pages` branch. Read the [Pages help article](#) for more information.

Overwrite site

Replace your existing site by using our automatic page generator. Author your content in our Markdown editor, select a theme, then publish.

Launch automatic page generator

☒ **Enforce HTTPS** — Required for your site because you are using the default domain (`rimusz-lab.github.io`)
HTTPS provides a layer of encryption that prevents others from snooping on or tampering with traffic to your site. When HTTPS is enforced, your site will only be served over HTTPS. [Learn more.](#)

图片 - Create Github Pages branch

默认情况下，源通常设置为 gh-pages 分支。如果这不是默认设置，那么请选择 gh-pages。

也可以在那里使用自定义域名。

并检查是否勾选了强制使用 HTTPS，以便在提供 chart 时使用 HTTPS。

这样的设置中，可以使用 master 分支来存储 chart 代码，并将 gh-pages 分支作为 chart 库，例如：

`https://USERNAME.github.io/REPONAME`。演示 [TS Charts](https://technosophos.github.io/tscharts/) 库可以通过 `https://technosophos.github.io/tscharts/` 访问。

普通的 web 服务器

要配置普通 Web 服务器来服务 Helm chart，只需执行以下操作：

- 将索引和 chart 置于服务器目录中
- 确保 `index.yaml` 可以在没有认证要求的情况下访问
- 确保 `yaml` 文件的正确内容类型（`text/yaml` 或 `text/x-yaml`）

例如，如果想在 `$WEBROOT/charts` 以外的目录为 chart 提供服务，请确保 Web 根目录中有一个 `charts/` 目录，并将索引文件和 chart 放入该文件夹内。

管理 chart 库

现在已有一个 chart 存储库，本指南的最后一部分将介绍如何维护该库中的 chart。

将 chart 存储在 chart 库中

现在已有一个 chart 存储库，让我们上传一个 chart 和一个索引文件到存储库。chart 库中的 chart 必须正确打包（`helm package chart-name/`）和版本（遵循 [SemVer 2](#) 标准）。

接下来的这些步骤是一个示例工作流程，也可以用你喜欢的任何工作流程来存储和更新 chart 库中的 chart。

准备好打包 chart 后，创建一个新目录，并将打包 chart 移动到该目录。

```
$ helm package docs/examples/alpine/
$ mkdir fantastic-charts
$ mv alpine-0.1.0.tgz fantastic-charts/
$ helm repo index fantastic-charts --url https://fantastic-charts.storage.googleapis.com
```

最后一条命令采用刚创建的本地目录的路径和远程 chart 库的 URL，并在给定的目录路径中生成 `index.yaml`。

现在可以使用同步工具或手动将 chart 和索引文件上传到 chart 库。如果使用 Google 云端存储，请使用 gsutil 客户端查看此示例工作流程。对于 GitHub，可以简单地将 chart 放入适当的目标分支中。

新添加 chart 添加到现有存储库

每次将新 chart 添加到存储库时，都必须重新生成索引。`helm repo index` 命令将 `index.yaml` 从头开始完全重建该文件，但仅包括它在本地找到的 chart。

可以使用 `--merge` 标志向现有 `index.yaml` 文件增量添加新 chart（在使用远程存储库（如 GCS）时，这是一个很好的选择）。运行 `helm repo index --help` 以了解更多信息，

确保上传修改后的 `index.yaml` 文件和 chart。如果生成了出处 `provenance` 文件，也要上传。

与他人分享 chart

准备好分享 chart 时，只需让别人知道存储库的 URL 是什么就可以了。

他们将通过 `helm repo add [NAME] [URL]` 命令将仓库添加到他们的 helm 客户端，并可以起一个带有任何想用来引用仓库的名字。

```
$ helm repo add fantastic-charts https://fantastic-charts.storage.googleapis.com
$ helm repo list
fantastic-charts    https://fantastic-charts.storage.googleapis.com
```

如果 chart 由 HTTP 基本认证支持，也可以在此处提供用户名和密码：

```
$ helm repo add fantastic-charts https://fantastic-charts.storage.googleapis.com --username my-username --password my-password
$ helm repo list
fantastic-charts    https://fantastic-charts.storage.googleapis.com
```

注意：如果存储库不包含有效信息库 `index.yaml` 文件，则添加不会成功。

之后，用户将能够搜索 chart。更新存储库后，他们可以使用该 `helm repo update` 命令获取最新的 chart 信息。

原理是 `helm repo add` 和 `helm repo update` 命令获取 `index.yaml` 文件并将它们存储在 `$HELM_HOME/repository/cache/` 目录中。这是 `helm search` 找到有关 chart 的信息的地方。

Copyright © Mingo(whmzsu@gmail.com) 2017-2018 all right reserved , powered by Gitbook Updated at 2018-11-24 10:01:24

同步 chart 库

注意：这个样例适用于提供 chart 库的 Google Cloud Storage (GCS) 存储 bucket。

前提条件

- 安装 [gsutil](#) 工具。这个样例依赖于 `gsutil rsync` 功能。
- 确保有权访问 Helm 客户端文件
- 可选：我们建议在 GCS 存储桶上设置对象版本控制，以防意外删除某些内容。

设置本地 chart 库目录

像我们在 [the chart repository guide](#) 中一样创建一个本地目录，并将打包的 chart 放入该目录中。

例如：

```
$ mkdir fantastic-charts
$ mv alpine-0.1.0.tgz fantastic-charts/
```

生成更新的 index.yaml

使用 Helm 通过将远程存储库的目录路径和 url 传递到 `helm repo index` 命令来生成更新的 index.yaml 文件，如下所示：

```
$ helm repo index fantastic-charts/ --url https://fantastic-charts.storage.googleapis.com
```

这将生成一个更新的 index.yaml 文件并放置在 `fantastic-charts/` 目录中。

同步本地和远程 chart 库

通过运行 `scripts/sync-repo.sh` 并传入本地目录名称和 GCS 存储桶名称，将目录的内容上传到您的 GCS 存储桶。

例如：

```
$ pwd
/Users/funuser/go/src/github.com/kubernetes/helm
$ scripts/sync-repo.sh fantastic-charts/ fantastic-charts
Getting ready to sync your local directory (fantastic-charts/) to a remote repository at gs://fantastic-charts
Verifying Prerequisites....
Thumbs up! Looks like you have gsutil. Let's continue.
Building synchronization state...
Starting synchronization
Would copy file://fantastic-charts/alpine-0.1.0.tgz to gs://fantastic-charts/alpine-0.1.0.tgz
Would copy file://fantastic-charts/index.yaml to gs://fantastic-charts/index.yaml
Are you sure you would like to continue with these changes?? [y/N] y
Building synchronization state...
Starting synchronization
Copying file://fantastic-charts/alpine-0.1.0.tgz [Content-Type=application/x-tar]...
Uploading gs://fantastic-charts/alpine-0.1.0.tgz: 740 B/740 B
Copying file://fantastic-charts/index.yaml [Content-Type=application/octet-stream]...
```

```
Uploading gs://fantastic-charts/index.yaml: 347 B/347 B
Congratulations your remote chart repository now matches the contents of fantastic-charts/
```

更新 chart 库

你可能需要保留 chart 库内容的本地副本，或者运行 `gsutil rsync` 将远程 chart 存储库的内容复制到本地目录。

例如：

```
$ gsutil rsync -d -n gs://bucket-name local-dir/ # the -n flag does a dry run
Building synchronization state...
Starting synchronization
Would copy gs://bucket-name/alpine-0.1.0.tgz to file://local-dir/alpine-0.1.0.tgz
Would copy gs://bucket-name/index.yaml to file://local-dir/index.yaml

$ gsutil rsync -d gs://bucket-name local-dir/ # performs the copy actions
Building synchronization state...
Starting synchronization
Copying gs://bucket-name/alpine-0.1.0.tgz...
Downloading file://local-dir/alpine-0.1.0.tgz: 740 B/740 B
Copying gs://bucket-name/index.yaml...
Downloading file://local-dir/index.yaml: 346 B/346 B
```

有用的网址：

- 文档 [gsutil rsync](#)
- [Chart 库指南](#)
- 文档[object versioning and concurrency control](#)

Copyright © Mingo(whmzs@gmail.com) 2017-2018 all right reserved , powered by GitbookUpdated at 2018-06-29 10:02:01

Helm 的出处与完整性验证

Helm 拥有可帮助 chart 用户验证 chart 包完整性和出处的工具。使用基于 PKI，GnuPG 和备受尊敬的软件包管理器的行业标准工具，Helm 可以生成并验证签名文件。

概述

完整性通过将 chart 与其出处记录进行比较来确定。出处记录存储在出处文件 `provenance` 中，并与打包的 chart 一起存储。例如，如果 chart 包被命名 `myapp-1.2.3.tgz`，其出处文件将是 `myapp-1.2.3.tgz.prov`。

出处文件在打包时生成（`helm package --sign ...`），并且可以通过多个命令检查，比如 `helm install --verify`。

工作流程

本节描述了有效使用出处数据的可用的工作流程。

前提条件：

- 二进制（非 ASCII）格式的有效 PGP 密钥对
- helm 命令行工具
- GnuPG >=2.1 命令行工具（可选）
- Keybase 命令行工具（可选）- 注意：如果 PGP 私钥有密码，支持 `--sign` 选项的任何命令系统将提示输入密码。可以设置 `HELM_KEY_PASSPHRASE` 环境变量避免每次输入。

注意：GnuPG 的密钥文件格式在 2.1 版中已更改。在该版本之前，没有必要从 GnuPG 中导出密钥，可以将 Helm 指向你的 `*.gpg` 文件。使用 2.1 时，引入了新的 `.kbx` 格式，Helm 不支持这种格式。

创建 chart：

```
$ helm create mychart
Creating mychart
```

准备打包后，将 `--sign` 参数加到 `helm package`。另外，指定已知签名密钥和包含相应私钥的密钥环 `keyring`：

```
$ helm package --sign --key 'helm signing key' --keyring path/to/keyring.secret mychart
```

提示：对于 GnuPG 用户，密钥环已存在 `~/.gnupg/secring.kbx`。可以使用 `gpg --list-secret-keys` 列出拥有的密钥。

警告：GnuPG v2.1 在默认位置在 `~/.gnupg/pubring.kbx`，使用新格式 `'kbx'` 存储密钥 `keyring`。请使用以下命令将钥匙 `keyring` 转换为传统的 `gpg` 格式：

```
$ gpg --export-secret-keys >~/.gnupg/secring.gpg
```

到这里，应该可用看到 `mychart-0.1.0.tgz` 和 `mychart-0.1.0.tgz.prov`。这两个文件最终应该上传到想要的 chart 库。

可以使用 `helm verify` 以下方式验证 chart：

```
$ helm verify mychart-0.1.0.tgz
```

验证失败如下所示样例：

```
$ helm verify topchart-0.1.0.tgz
Error: sha256 sum does not match for topchart-0.1.0.tgz: "sha256:1939fbf7c1023d2f6b865d137bbb600e0c42061c3235528b1e8c82f4450c12a7" != "sha256:5a391a90de56778dd3274e47d789a2c84e0e106e1a37ef8cfa51fd60ac9e623a"
```

要在安装过程中进行验证，请使用该 `--verify` 标志。

```
$ helm install --verify mychart-0.1.0.tgz
```

如果密钥 keyring（包含与签名 chart 关联的公钥）不在默认位置，则可能需要 `--keyring PATH` 像 `helm package` 示例中那样指向密钥 keyring。

如果验证失败，在 chart 被推到 Tiller 之前中止安装终止。

使用 Keybase.io 凭据

该 Keybase.io 服务可以很容易建立信任链的密码身份。密钥库凭证可用于对 chart 进行签名。

前提条件：

- 已配置的 Keybase.io 帐户
- GnuPG 在本地安装
- keybaseCLI 本地安装

签署软件包

第一步是将 keybase 密钥导入到本地 GnuPG 密钥 keyring 中：

```
$ keybase pgp export -s > secring.gpg
```

这会将 Keybase 密钥转换为 OpenPGP 格式，然后将其本地导入到 `secring.gpg` 文件中。

可以通过运行 `gpg --list-secret-keys` 进行仔细检查。

```
$ gpg --list-secret-keys /Users/mattbutcher/.gnupg/secring.gpg
-----
sec   2048R/1FC18762 2016-07-25
uid           technosophos (keybase.io/technosophos) <technosophos@keybase.io>
ssb   2048R/D125E546 2016-07-25
```

提示: 如果你想添加一个 Keybase key 到已存在的 keyring, 你需要执行 `keybase pgp export -s | gpg --import && gpg --export-secret-keys --outfile secring.gpg`

你的密钥有一个标识符字符串：

```
technosophos (keybase.io/technosophos) <technosophos@keybase.io>
```

这是钥匙的全名。

接下来，可以使用 `helm package` 打包和签名 chart。 `--key` 确保至少使用该名称字符串的一部分。

```
$ helm package --sign --key technosophos --keyring ~/.gnupg/secring.gpg mychart
```

结果，`package` 命令应该生成一个 `.tgz` 文件和一个 `.tgz.prov` 文件。

验证软件包

还可以使用类似的技术来验证由其他人的 Keybase 密钥签名的 chart。假设想验证签名的软件包 `keybase.io/technosophos`。请使用该 `keybase` 工具：

```
$ keybase follow technosophos
$ keybase pgp pull
```

上面的第一条命令跟踪用户 `technosophos`。接下来 `keybase pgp pull`，将关注的所有帐户的 OpenPGP 密钥下载到 GnuPG 密钥环（`~/.gnupg/pubring.gpg1`）中

到此，可以使用 `helm verify` 或带有 `--verify` 参数的任何命令：

```
$ helm verify somechart-1.2.3.tgz
```

可能无法验证的原因

下面是失败的常见原因。

- `prov` 文件丢失或损坏。这表明某些内容配置错误或原始维护人员未创建出处文件。
- 用于签署文件的密钥不在钥匙 `keyring` 中。这表明签名 chart 的组织不是已经信任的人员。
- `prov` 文件的验证失败。这表明 chart 或出处数据有问题。
- `prov` 文件中的文件哈希与压缩包文件的哈希不匹配。这表明压缩包已被篡改。
- 如果验证失败，则有理由怀疑该软件包有问题。

Provenance 文件

Provenance 文件包含 chart 的 YAML 文件以及几条验证信息。Provenance 文件被设计自动生成。

添加了以下几个出处数据：

- 包含 chart 文件（`Chart.yaml`）可以让人员和工具轻松查看 chart 内容。
- 包括 chart 包（`.tgz` 文件）的签名（SHA256，就像 Docker）一样，可用于验证 chart 包的完整性。
- 整个文件使用 PGP 使用的算法进行签名（参见 <http://keybase.io>，这是一种使加密签名和验证变得容易的新方法）。

这样的组合给了用户以下保证：

- 包本身没有被篡改（校验和包 `tgz`）。
- 已知发布此包的组织（通过 GnuPG / PGP 签名）。

该文件的格式如下所示：

```
-----BEGIN PGP SIGNED MESSAGE-----
name: nginx
description: The nginx web server as a replication controller and service pair.
version: 0.5.1
keywords:
- https
- http
- web server
- proxy
source:
- https://github.com/foo/bar
home: http://nginx.com

...
files:
  nginx-0.5.1.tgz: "sha256:9f5270f50fc842cfc717f817e95178f"
-----BEGIN PGP SIGNATURE-----
```

```
Version: GnuPG v1.4.9 (GNU/Linux)

iEYEAReCAAYFAKjilUEACgQkB01zfu119ZnHuQCdGCGcg2YxF3XFscJLS41zHlvte
WkQAmQGhuuoLEJuKhRNo+Wy7mhE7u1YG
=EIFq
-----END PGP SIGNATURE-----
```

注意，YAML 部分包含两个文档（由分隔 `... \n`）。首先是 Chart.yaml。第二个是校验和，一个文件名到 SHA-256 摘要的映射（上显示的值是假的 / 被截断的）

签名块是一个标准的 PGP 签名，它提供了 [防篡改](#) 功能。

Chart 库

Chart 库用作 Helm chart 的集中场所。

Chart 存储库必须能够通过特定的请求通过 HTTP 为 provenance 文件提供服务，并且必须使它们在与 chart 相同的 URI 路径下可用。

例如，如果软件包的基本 URL 是 `https://example.com/charts/mychart-1.2.3.tgz`，provenance 文件（如果存在）必须可以通过 `https://example.com/charts/mychart-1.2.3.tgz.prov` 访问。

从最终用户的角度来看，`helm install --verify myrepo/mychart-1.2.3` 应该无需额外的配置或操作即可下载 chart 和 provenance 文件。

确认认证和鉴权

在处理信任链系统时，能够确认签名者的认证很重要。或者，简单地说，上述系统取决于相信签名人员的事实。这反过来又意味着你需要信任签名者的公钥。

Kubernetes Helm 的设计决策之一是 Helm 项目不会将自己插入信任链中作为必要的组分。我们不希望成为所有 chart 签名者的“证书颁发机构”。相反，我们强烈支持分散模式，这是我们选择 OpenPGP 作为基础技术的原因之一。所以说到确认认证时，我们在 Helm 2.0.0 中对这个步骤或多或少未明确定义。

但是，对于那些有兴趣使用 provenance 系统的人，我们有一些建议：

- [Keybase](#) 平台提供了可靠信息的公开集中存放。
- 可以使用 Keybase 存储密钥或获取其他公钥。
- Keybase 也有很多可用的文档
- 虽然我们还没有对它进行测试，但 Keybase 的“安全网站”功能可用于服务 Helm chart。
- Kubernetes chart 项目正在设法解决这个官方 chart 库 [official Kubernetes Charts project](#) 问题。
- 这里有一个 [很长的 issue](#)，详细介绍了当前的想法。
- 基本思想原则是官方的“chart reviewer”用她或他的钥匙签名 chart，然后将得到的 Provenance 文件上传到 chart 存储库。
- 关于 `有效签名密钥列表可包含在 index.yaml 存储库文件中` 的想法已经有了一些工作进展。

最后，信任链是 Helm 的一个发展特征，一些社区成员已经提出了将 OSI 模型的一部分用于签名。这是 Helm 团队的一个开放性调查。如果你有兴趣，请参与其中。

Copyright © Mingo(whmzsu@gmail.com) 2017-2018 all right reserved, powered by Gitbook Updated at 2018-11-23 22:31:49

Chart 测试

一个 chart 包含许多一起工作的 Kubernetes 资源和组件。作为 chart 作者，可能需要编写一些测试来验证 chart 在安装时是否按预期工作。这些测试还有助于 chart 消费者了解 chart 应该做什么。

测试在 Helm chart 中的 `templates/` 目录，是一个 pod 定义，指定一个给定的命令来运行容器。容器应该成功退出（`exit 0`），测试被认为是成功的。该 pod 定义必须包含 helm 测试 hook 注释之一：`helm.sh/hook: test-success` 或 `helm.sh/hook: test-failure`。

示例测试：

- 验证来自 `values.yaml` 文件的配置是否正确注入。
- 确保用户名和密码正常工作
- 确保不正确的用户名和密码不起作用
- 断言服务已启动并正确进行负载均衡
- 等等

可以使用该 `helm test` 命令在 `release` 中运行 Helm 中的预定义测试。对于 chart 使用者来说，这是一种很好的方式来检查他们发布的 chart（或应用程序）是否按预期工作。

Helm 测试 hook 的分解

在 Helm 中，有两个测试 hook：`test-success` 和 `test-failure`。

`test-success` 表示测试 pod 应该成功完成。换句话说，容器中的容器应该 `exit 0`。`test-failure` 是一种断言测试容器不能成功完成的方式。如果 pod 中的容器未 `exit 0`，则表示成功。

示例测试

下面是一个示例 WordPress chart 中 helm 测试 pod 定义的示例。这个测试验证了 mariadb 的连接和登录：

```
wordpress/  
  Chart.yaml  
  README.md  
  values.yaml  
  charts/  
  templates/  
    templates/tests/test-mariadb-connection.yaml
```

在 `wordpress/templates/tests/test-mariadb-connection.yaml` 中：

```
apiVersion: v1  
kind: Pod  
metadata:  
  name: "{{.Release.Name}}-credentials-test"  
  annotations:  
    "helm.sh/hook": test-success  
spec:  
  containers:  
  - name: "{{.Release.Name}}-credentials-test"  
    image: "{{.Values.image}}"   
    env:  
    - name: MARIADB_HOST  
      value: "{{template \"mariadb.fullname\" .}}"   
    - name: MARIADB_PORT
```

```
    value: "3306"
  - name: WORDPRESS_DATABASE_NAME
    value: {{default "" .Values.mariadb.mariadbDatabase | quote}}
  - name: WORDPRESS_DATABASE_USER
    value: {{default "" .Values.mariadb.mariadbUser | quote}}
  - name: WORDPRESS_DATABASE_PASSWORD
    valueFrom:
      secretKeyRef:
        name: {{template "mariadb.fullname" .}}
        key: mariadb-password
    command: ["sh", "-c", "mysql --host=$MARIADB_HOST --port=$MARIADB_PORT --user=$WORDPRESS_DATABASE_USER --password=$WORDPRESS_DATABASE_PASSWORD"]
    restartPolicy: Never
```

在 release 上运行测试套件的步骤

1. `$ helm install wordpress`

```
NAME:    quirky-walrus
LAST DEPLOYED: Mon Feb 13 13:50:43 2017
NAMESPACE: default
STATUS: DEPLOYED
```

2. `$ helm test quirky-walrus`

```
RUNNING: quirky-walrus-credentials-test
SUCCESS: quirky-walrus-credentials-test
```

注意

- 可以在单个 yaml 文件中定义尽可能多的测试，也可以在 templates / 目录中的多个 yaml 文件中进行分布测试
- 提倡将测试套件嵌入到一个 tests/ 目录下，比如 `<chart-name>/templates/tests/` 以便实现更多隔离

Copyright © Mingo(whmzsu@gmail.com) 2017-2018 all right reserved , powered by GitbookUpdated at 2018-11-23 22:31:49

Chart 库：常见问题

本节跟踪使用 chart 库时的一些常遇到的问题。

我们很欢迎你的帮助来改进这个文档。要添加，更正或删除信息，提出问题[file an issue](#)或向我们发送 PR。

Fetching

问：当我试图从我的自定义 **repo** 中获取 **chart** 时，为什么会出现错误 `unsupported protocol scheme` ？

答:(helm 版本小于 2.5.0) 这很可能是由于创建 chart 索引而未指定 `--url` 参数。尝试使用类似命令 `helm repo index --url http://my-repo/charts 重建 index.yaml`，然后将其重新上传到自定义 chart repo 库。

这个问题在Helm 2.5.0中进行了更改。

Copyright © Mingo(whmzsu@gmail.com) 2017-2018 all right reserved , powered by GitbookUpdated at 2018-11-23 22:31:49

Chart 模板开发人员指南

本指南介绍了 Helm 的 chart 模板，重点介绍模板语言。

模板生成 manifest 文件，它们是 Kubernetes 可以识别的 YAML 格式的资源描述。我们将了解模板的结构，如何使用，如何编写 Go 模板以及如何调试。

本指南着重介绍以下概念：

- Helm 模板语言
- 使用 values
- 使用模板的技巧

本指南面向学习Helm模板语言的细节的人。其他指南提供介绍性材料，示例和最佳实践。

Copyright © Mingo(whmzsu@gmail.com) 2017-2018 all right reserved , powered by GitbookUpdated at 2018-06-29 10:04:25

开始使用 chart 模板

在本指南的这部分，我们将创建一个 chart，然后添加第一个模板。我们在这里创建的 chart 将在指南的其他部分使用。

开始，我们来看一下 Helm chart。

Charts

如 chart 指南中所述，Helm chart 的结构如下所示：

```
mychart/  
  Chart.yaml  
  values.yaml  
  charts/  
  templates/  
  ...
```

`templates/` 目录用于放置模板文件。当 Tiller 评估 chart 时，它将 `templates/` 通过模板渲染引擎发送目录中的所有文件。然后，Tiller 收集这些模板的结果并将它们发送给 Kubernetes。

`values.yaml` 文件对模板也很重要。该文件包含 chart 默认值。这些值可能在用户在 `helm install` 或 `helm upgrade` 期间被覆盖。

`Chart.yaml` 文件包含 chart 的说明。可以从模板中查看访问它。该 `charts/` 目录可能包含其他 chart（我们称之为子 chart）。在本指南的后面，我们将看到它们在模板渲染方面如何起作用。

初始 chart

对于本指南，我们将创建一个名为 mychart 的简单 chart，然后我们将在 chart 内部创建一些模板。

```
$ helm create mychart  
Creating mychart
```

从这里开始，我们将在 `mychart` 目录中工作。

快速看一下目录 `mychart/templates/`

看一下 `mychart/templates/` 目录，发现如下几个文件已经存在。

- `NOTES.txt`：chart 的“帮助文本”。这会在用户运行 `helm install` 时显示给用户。
- `deployment.yaml`：创建 Kubernetes [deployment](#) 的基本 manifest
- `service.yaml`：为 deployment 创建 service 端点的基本 manifest
- `_helpers.tpl`：放置模板助手的地方，可以在整个 chart 中重复使用

而我们要做的就是..... 全部删除它们！这样我们就可以从头开始学习我们的教程。实际上，我们将创建自己的 `NOTES.txt` 和 `_helpers.tpl`。

```
$ rm -rf mychart/templates/*.*
```

在编写生产级 chart 时，使用这些 chart 的基本版本可能非常有用。所以在你的日常 chart 制作中，可以不删除它们。

第一个模板

我们要创建的第一个模板将是一个 ConfigMap。在 Kubernetes 中，ConfigMap 只是存储配置数据的地方。其他的東西，比如 Pod，可以访问 ConfigMap 中的数据。

由于 ConfigMaps 是基础资源，它们为我们提供了一个很好的起点。

我们首先创建一个名为 mychart/templates/configmap.yaml：

```
apiVersion: v1
kind: ConfigMap
metadata:
  name: mychart-configmap
data:
  myvalue: "Hello World"
```

提示：模板名称不遵循严格的命名模式。但是，我们建议 .yaml 为 YAML 文件后缀，.tpl 为模板助手后缀。

上面的 YAML 文件是一个简单的 ConfigMap，具有最少的必要字段。由于该文件位于 templates/ 目录中，因此将通过模板引擎发送。

在 templates/ 目录中放置一个像这样的纯 YAML 文件。当 Tiller 读取这个模板时，它会直接发送给 Kubernetes。

有了这个简单的模板，我们现在有一个可安装的 chart。我们可以像这样安装它：

```
$ helm install ./mychart
NAME: full-coral
LAST DEPLOYED: Tue Nov 1 17:36:01 2016
NAMESPACE: default
STATUS: DEPLOYED

RESOURCES:
==> v1/ConfigMap
NAME          DATA      AGE
mychart-configmap 1          1m
```

在上面的输出中，我们可以看到我们的 ConfigMap 已经创建。使用 Helm，我们可以检索版本并查看加载的实际模板。

```
$ helm get manifest full-coral

---
# Source: mychart/templates/configmap.yaml
apiVersion: v1
kind: ConfigMap
metadata:
  name: mychart-configmap
data:
  myvalue: "Hello World"
```

该 helm get manifest 命令获取 release 名称（full-coral）并打印出上传到服务器的所有 Kubernetes 资源。每个文件都以 --- 开始作为 YAML 文档的开始，然后是一个自动生成的注释行，告诉我们该模板文件生成的这个 YAML 文档。

从那里开始，我们可以看到 YAML 数据正是我们在我们的 configmap.yaml 文件中所设计的。

现在我们可以删除我们的 release：helm delete full-coral。

添加一个简单的模板调用

硬编码 name：成资源通常被认为是不好的做法。名称应该是唯一的一个版本。所以我们可能希望通过插入 release 名称来生成一个名称字段。

提示：name: 由于 DNS 系统的限制，该字段限制为 63 个字符。因此，release 名称限制为 53 个字符。Kubernetes 1.3 及更早版本仅限于 24 个字符（即 14 个字符名称）。

让我们改一下 configmap.yaml。

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

name: 现在这个值发生了变化成了 {{.Release.Name}}-configmap。

模板指令包含在 {{ 和 }} 块中。

模板指令 {{.Release.Name}} 将 release 名称注入模板。传递给模板的值可以认为是 namespace 对象，其中 dot (.) 分隔每个 namespace 元素。

Release 前面的前一个小圆点表示我们从这个范围的最上面的 namespace 开始（我们将稍微谈一下 scope）。所以我们可以这样理解 .Release.Name："从顶层命名空间开始，找到 Release 对象，然后在里面查找名为 Name 的对象"。

该 Release 对象是 Helm 的内置对象之一，稍后我们将更深入地介绍它。但就目前而言，这足以说明这会显示 Tiller 分配给我们发布的 release 名称。

现在，当我们安装我们的资源时，我们会立即看到使用这个模板指令的结果：

```
$ helm install ./mychart
NAME: clunky-serval
LAST DEPLOYED: Tue Nov 1 17:45:37 2016
NAMESPACE: default
STATUS: DEPLOYED

RESOURCES:
==> v1/ConfigMap
NAME                DATA      AGE
clunky-serval-configmap 1          1m
```

注意，在该 RESOURCES 部分中，我们看到的名称 clunky-serval-configmap 不是 mychart-configmap。

可以运行 helm get manifest clunky-serval 以查看整个生成的 YAML。

现在，我们看过了基础的模板：YAML 文件嵌入了模板指令，通过。在下一部分中，我们将深入研究模板。但在继续之前，有一个快速技巧可以使构建模板更快：当您想测试模板渲染，但实际上没有安装任何东西时，可以使用 helm install --debug --dry-run ./mychart。这会将 chart 发送到 Tiller 服务器，它将渲染模板。但不是安装 chart，它会将渲染模板返回，以便可以看到输出：

```
$ helm install --debug --dry-run ./mychart
SERVER: "localhost:44134"
CHART PATH: /Users/mattbutcher/Code/Go/src/k8s.io/helm/_scratch/mychart
NAME: goodly-guppy
TARGET NAMESPACE: default
CHART: mychart 0.1.0
MANIFEST:
---
# Source: mychart/templates/configmap.yaml
apiVersion: v1
kind: ConfigMap
metadata:
  name: goodly-guppy-configmap
data:
  myvalue: "Hello World"
```

使用 `--dry-run` 可以更容易地测试代码，但不能确保 Kubernetes 本身会接受生成的模板。最好不要假定你的 chart 只要 `--dry-run` 成功而被安装。

在接下来的几节中，我们将采用我们在这里定义的基本 chart，并详细探索 Helm 模板语言。我们将开始使用内置对象。

Copyright © Mingo(whmzsus@gmail.com) 2017-2018 all right reserved , powered by Gitbook Updated at 2018-06-29 10:17:10

内置对象

对象从模板引擎传递到模板中。你的代码可以传递对象（我们将在说明 `with` 和 `range` 语句时看到示例）。甚至有几种方法在模板中创建新对象，就像我们稍后会看的 `tuple` 函数一样。

对象可以很简单，只有一个值。或者他们可以包含其他对象或函数。例如，`Release` 对象包含多个对象（如 `Release.Name`）并且 `Files` 对象具有一些函数。

在上一节中，我们使用 `{{.Release.Name}}` 将 `release` 的名称插入到模板中。`Release` 是可以在模板中访问的顶级对象之一。

- `Release`：这个对象描述了 `release` 本身。它里面有几个对象：
- `Release.Name`：release 名称
- `Release.Time`：release 的时间
- `Release.Namespace`：release 的 namespace（如果清单未覆盖）
- `Release.Service`：release 服务的名称（始终是 `Tiller`）。
- `Release.Revision`：此 release 的修订版本号。它从 1 开始，每 `helm upgrade` 一次增加一个。
- `Release.IsUpgrade`：如果当前操作是升级或回滚，则将其设置为 `true`。
- `Release.IsInstall`：如果当前操作是安装，则设置为 `true`。
- `Values`：从 `values.yaml` 文件和用户提供的文件传入模板的值。默认情况下，`Values` 是空的。
- `Chart`：`Chart.yaml` 文件的内容。任何数据 `Chart.yaml` 将在这里访问。例如 `{{.Chart.Name}}`-`{{.Chart.Version}}` 将打印出来 `mychart-0.1.0`。chart 指南中 [Charts Guide](#) 列出了可用字段
- `Files`：这提供对 chart 中所有非特殊文件的访问。虽然无法使用它来访问模板，但可以使用它来访问 chart 中的其他文件。请参阅“访问文件”部分。
- `Files.Get` 是一个按名称获取文件的函数（`.Files.Get config.ini`）
- `Files.GetBytes` 是将文件内容作为字节数组而不是字符串获取的函数。这对于像图片这样的东西很有用。
- `Capabilities`：这提供了关于 Kubernetes 集群支持的功能的信息。
- `Capabilities.APIVersions` 是一组版本信息。
- `Capabilities.APIVersions.Has $version` 指示是否在群集上启用版本（`batch/v1`）。
- `Capabilities.KubeVersion` 提供了查找 Kubernetes 版本的方法。它具有以下值：`Major`，`Minor`，`GitVersion`，`GitCommit`，`GitTreeState`，`BuildDate`，`GoVersion`，`Compiler`，和 `Platform`。
- `Capabilities.TillerVersion` 提供了查找 Tiller 版本的方法。它具有以下值：`SemVer`，`GitCommit`，和 `GitTreeState`。
- `Template`：包含有关正在执行的当前模板的信息
- `Name`：到当前模板的 namespace 文件路径（例如 `mychart/templates/mytemplate.yaml`）
- `BasePath`：当前 chart 模板目录的 namespace 路径（例如 `mychart/templates`）。

这些值可用于任何顶级模板。我们稍后会看到，这并不意味着它们将在任何地方都要有。

内置值始终以大写字母开头。这符合 Go 的命名约定。当你创建自己的名字时，你可以自由地使用适合你的团队的惯例。一些团队，如 Kubernetes chart 团队，选择仅使用首字母小写字母来区分本地名称与内置名称。在本指南中，我们遵循该约定。

Copyright © Mingo(whmzsu@gmail.com) 2017-2018 all right reserved, powered by Gitbook Updated at 2018-06-29 10:04:16

values 文件

在上一节中，我们看了 Helm 模板提供的内置对象。四个内置对象之一是 Values。该对象提供对传入 chart 的值的访问。其内容来自四个来源：

- chart 中的 `values.yaml` 文件
- 如果这是一个子 chart，来自父 chart 的 `values.yaml` 文件
- value 文件通过 `helm install` 或 `helm upgrade` 的 `-f` 标志传入文件（`helm install -f myvals.yaml ./mychart`）
- 通过 `--set`（例如 `helm install --set foo=bar ./mychart`）

上面的列表按照特定的顺序排列：values.yaml 在默认情况下，父级 chart 的可以覆盖该默认级别，而该 chart values.yaml 又可以被用户提供的 values 文件覆盖，而该文件又可以被 `--set` 参数覆盖。

值文件是纯 YAML 文件。我们编辑 `mychart/values.yaml`，然后来编辑我们的 `ConfigMap` 模板。

删除默认带的 values.yaml，我们只设置一个参数：

```
favoriteDrink: coffee
```

现在我们可以使用这个：

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

注意我们在最后一行 `{{ .Values.favoriteDrink }}` 获取 `favoriteDrink` 的值。

让我们看看这是如何渲染的。

```
$ helm install --dry-run --debug ./mychart
SERVER: "localhost:44134"
CHART PATH: /Users/mattbutcher/Code/Go/src/k8s.io/helm/_scratch/mychart
NAME: geared-marsupi
TARGET NAMESPACE: default
CHART: mychart 0.1.0
MANIFEST:
---
# Source: mychart/templates/configmap.yaml
apiVersion: v1
kind: ConfigMap
metadata:
  name: geared-marsupi-configmap
data:
  myvalue: "Hello World"
  drink: coffee
```

由于 `favoriteDrink` 在默认 `values.yaml` 文件中设置为 `coffee`，这就是模板中显示的值。我们可以轻松地在我们的 `helm install` 命令中通过加一个 `--set` 添标志来覆盖：

```
helm install --dry-run --debug --set favoriteDrink=slurm ./mychart
SERVER: "localhost:44134"
CHART PATH: /Users/mattbutcher/Code/Go/src/k8s.io/helm/_scratch/mychart
NAME: solid-vulture
TARGET NAMESPACE: default
```



```
CHART: mychart 0.1.0
MANIFEST:
---
# Source: mychart/templates/configmap.yaml
apiVersion: v1
kind: ConfigMap
metadata:
  name: solid-vulture-configmap
data:
  myvalue: "Hello World"
  drink: slurm
```

由于 `--set` 比默认 `values.yaml` 文件具有更高的优先级，我们的模板生成 `drink: slurm`。

`values` 文件也可以包含更多结构化内容。例如，我们在 `values.yaml` 文件中可以创建 `favorite` 部分，然后在其中添加几个键：

```
favorite:
  drink: coffee
  food: pizza
```

现在我们稍微修改模板：

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

虽然以这种方式构建数据是可以的，但建议保持 `value` 树浅一些，平一些。当我们看看为子 `chart` 分配值时，我们将看到如何使用树结构来命名值。

删除默认 key

如果您需要从默认值中删除一个键，可以覆盖该键的值为 `null`，在这种情况下，Helm 将从覆盖值合并中删除该键。

例如，stable 版本的 Drupal chart 允许配置 liveness 探测器，如果你配置自定义的 `image`。以下是默认值：

```
livenessProbe:
  httpGet:
    path: /user/login
    port: http
  initialDelaySeconds: 120
```

如果尝试覆盖 liveness Probe 处理程序 `exec` 而不是 `httpGet`，使用 `--set livenessProbe.exec.command=[cat,docroot/CHANGELOG.txt]`，Helm 会将默认和重写的键合并在一起，从而产生以下 YAML：

```
livenessProbe:
  httpGet:
    path: /user/login
    port: http
  exec:
    command:
      - cat
      - docroot/CHANGELOG.txt
  initialDelaySeconds: 120
```

但是，Kubernetes 会报错，因为无法声明多个 liveness Probe 处理程序。为了克服这个问题，你可以指示 Helm 过将 livenessProbe.httpGet 通设置为空来删除它：

```
helm install stable/drupal --set image=my-registry/drupal:0.1.0 --set livenessProbe.exec.command=[cat,docroot/C  
HANGELOG.txt] --set livenessProbe.httpGet=null
```

到这里，我们已经看到了几个内置对象，并用它们将信息注入到模板中。现在我们来看看模板引擎的另外内容：函数和管道。

Copyright © Mingo(whmzsu@gmail.com) 2017-2018 all right reserved , powered by GitbookUpdated at 2018-06-29 10:04:32

模板函数和管道

目前为止，我们已经知道如何将信息放入模板中。但是这些信息未经修改就被放入模板中。有时我们想要转换这些数据，使得他们对我们来说更有用。

让我们从一个最佳实践开始：当从 `Values` 对象注入字符串到模板中时，我们引用这些字符串。我们可以通过调用 `quote` 模板指令中的函数来实现：

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

模板函数遵循语法 `functionName arg1 arg2...`。在上面的代码片段中，`quote .Values.favorite.drink` 调用 `quote` 函数并将一个参数传递给它。

Helm 拥有超过 60 种可用函数。其中一些是由 Go 模板语言 [Go template language](#) 本身定义的。其他大多数都是 Sprig 模板库 [Sprig template library](#) 的一部分。在我们讲解例子进行的过程中，我们会看到很多。

虽然我们将 Helm 模板语言视为 Helm 特有的，但它实际上是 Go 模板语言，一些额外函数和各种包装器的组合，以将某些对象暴露给模板。Go 模板上的许多资源在了解模板时可能会有所帮助。

管道

模板语言的强大功能之一是其管道概念。利用 UNIX 的一个概念，管道是一个链接在一起的一系列模板命令的工具，以紧凑地表达一系列转换。换句话说，管道是按顺序完成几件事情的有效方式。我们用管道重写上面的例子。

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

在这个例子中，没有调用 `quote ARGUMENT`，我们调换了顺序。我们使用管道 (`|`) 将“参数”发送给函数：`.Values.favorite.drink | quote`。使用管道，我们可以将几个功能链接在一起：

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

反转顺序是模板中的常见做法。你会看到 `.val | quote` 比 `quote .val` 更常见。练习也是。

当评估时，该模板将产生如下结果：

```
# Source: mychart/templates/configmap.yaml
apiVersion: v1
kind: ConfigMap
metadata:
  name: trendsetting-p-configmap
data:
  myvalue: "Hello World"
  drink: "coffee"
  food: "PIZZA"
```

请注意，我们的原来 `pizza` 现在已经转换为 `"PIZZA"`。

当有像这样管道参数时，第一个评估（`.Values.favorite.drink`）的结果将作为函数的最后一个参数发送。我们可以修改上面的饮料示例来说明一个带有两个参数的函数 `repeat COUNT STRING`：

```
apiVersion: v1
kind: ConfigMap
metadata:
  name: {{ .Release.Name }}-configmap
data:
  myvalue: "Hello World"
  drink: {{ .Values.favorite.drink | repeat 5 | quote }}
  food: {{ .Values.favorite.food | upper | quote }}
```

该 `repeat` 函数将回送给定的字符串和给定的次数，所以我们将得到这个输出：

```
# Source: mychart/templates/configmap.yaml
apiVersion: v1
kind: ConfigMap
metadata:
  name: melting-porcup-configmap
data:
  myvalue: "Hello World"
  drink: "coffeecoffeecoffeecoffeecoffee"
  food: "PIZZA"
```

使用 default 函数

经常使用的一个函数是 `default`：`default DEFAULT_VALUE GIVEN_VALUE`。该功能允许在模板内部指定默认值，以防该值被省略。让我们用它来修改上面的饮料示例：

```
drink: {{ .Values.favorite.drink | default "tea" | quote }}
```

如果我们像往常一样运行，我们会得到我们的 `coffee`：

```
# Source: mychart/templates/configmap.yaml
apiVersion: v1
kind: ConfigMap
metadata:
  name: virtuous-mink-configmap
data:
  myvalue: "Hello World"
  drink: "coffee"
  food: "PIZZA"
```

现在，我们将从以下位置删除喜欢的饮料设置 `values.yaml`：

```
favorite:
```

```
#drink: coffee
food: pizza
```

现在重新运行 `helm install --dry-run --debug ./mychart` 会产生这个 YAML：

```
# Source: mychart/templates/configmap.yaml
apiVersion: v1
kind: ConfigMap
metadata:
  name: fair-worm-configmap
data:
  myvalue: "Hello World"
  drink: "tea"
  food: "PIZZA"
```

在实际的 chart 中，所有静态默认值应该存在于 values.yaml 中，不应该使用该 default 命令重复（否则它们将是重复多余的）。但是，default 命令对于计算的值是合适的，因为计算值不能在 values.yaml 中声明。例如：

```
drink: {{ .Values.favorite.drink | default (printf "%s-tea" (include "fullname" .)) }}
```

在一些地方，一个 `if` 条件可能比这 `default` 更适合。我们将在下一节中看到这些。

模板函数和管道是转换信息并将其插入到 YAML 中的强大方法。但有时候需要添加一些比插入字符串更复杂一些的模板逻辑。在下一节中，我们将看看模板语言提供的控制结构。

运算符函数

对于模板，运算符（eq，ne，lt，gt，and，or 等等）都是已实现的功能。在管道中，运算符可以用圆括号（`()` 和 `{ }`）分组。

现在我们可以从函数和管道转向流控制,条件，循环和范围修饰符。

Copyright © Mingo(whmzsu@gmail.com) 2017-2018 all right reserved，powered by GitbookUpdated at 2018-06-29 10:04:22

流程控制

控制结构（模板说法中称为“动作”）为模板作者提供了控制模板生成流程的能力。Helm 的模板语言提供了以下控制结构：

- `if/else` 用于创建条件块
- `with` 指定范围
- `range`，它提供了一个“for each”风格的循环

除此之外，它还提供了一些声明和使用命名模板区的操作：

- `define` 在模板中声明一个新的命名模板
- `template` 导入一个命名模板
- `block` 声明了一种特殊的可填写模板区域

在本节中，我们将谈论 `if`，`with` 和 `range`。其他内容在本指南后面的“命名模板”一节中介绍。

if/else

我们要看的第一个控制结构是用于在模板中有条件地包含文本块。这就是 `if/else` 块。

条件的基本结构如下所示：

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

注意，我们现在讨论的是管道而不是值。其原因是要明确控制结构可以执行整个管道，而不仅仅是评估一个值。

如果值为如下情况，则管道评估为 `false`。

- 一个布尔型的假
- 一个数字零
- 一个空的字符串
- 一个 `nil`（空或 `null`）
- 一个空的集合（`map`，`slice`，`tuple`，`dict`，`array`）

在其他情况下，条件值为 `true` 此管道被执行。

我们为 `ConfigMap` 添加一个简单的条件。如果饮料被设置为咖啡，我们将添加另一个设置：

```
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 and (.Values.favorite.drink) (eq .Values.favorite.drink "coffee") }}mug: true{{ end }}
```

注意 `.Values.favorite.drink` 必须已定义，否则在将它与“coffee”进行比较时会抛出错误。由于我们在上一个例子中注释掉了 `drink: coffee`，因此输出不应该包含 `mug: true` 标志。但是如果我们将该行添加回 `values.yaml` 文件中，输出应该如下所示：

```
# Source: mychart/templates/configmap.yaml
apiVersion: v1
kind: ConfigMap
metadata:
  name: eyewitness-elk-configmap
data:
  myvalue: "Hello World"
  drink: "coffee"
  food: "PIZZA"
  mug: true
```

控制空格

在查看条件时，我们应该快速查看模板中的空格控制方式。让我们看一下前面的例子，并将其格式化为更容易阅读的格式：

```
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}}
```

最初，这看起来不错。但是如果我们通过模板引擎运行它，我们会得到一个错误的结果：

```
$ helm install --dry-run --debug ./mychart
SERVER: "localhost:44134"
CHART PATH: /Users/mattbutcher/Code/Go/src/k8s.io/helm/_scratch/mychart
Error: YAML parse error on mychart/templates/configmap.yaml: error converting YAML to JSON: yaml: line 9: did not find expected key
```

发生了什么？由于上面的空格，我们生成了不正确的 YAML。

```
# Source: mychart/templates/configmap.yaml
apiVersion: v1
kind: ConfigMap
metadata:
  name: eyewitness-elk-configmap
data:
  myvalue: "Hello World"
  drink: "coffee"
  food: "PIZZA"
  mug: true
```

`mug` 不正确地缩进。让我们简单地缩进那行，然后重新运行：

```
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}}
```

当我们发送该信息时，我们会得到有效的 YAML，但仍然看起来有点意思：

```
# Source: mychart/templates/configmap.yaml
apiVersion: v1
kind: ConfigMap
metadata:
  name: telling-chimp-configmap
data:
  myvalue: "Hello World"
  drink: "coffee"
  food: "PIZZA"

  mug: true
```

请注意，我们在 YAML 中收到了一些空行。为什么？当模板引擎运行时，它将删除，但它完全按原样保留剩余的空白。

YAML 中的缩进空格是严格的，因此管理空格变得非常重要。幸运的是，Helm 模板有几个工具可以帮助我们。

首先，可以使用特殊字符修改模板声明的大括号语法，以告诉模板引擎填充空白。{{- （添加了破折号和空格）表示应该将空白左移，而 -}} 意味着应该删除右空格。注意！换行符也是空格！

确保 - 和其他指令之间有空格。-3 意思是“删除左空格并打印 3”，而 -3 意思是“打印 - 3”。

使用这个语法，我们可以修改我们的模板来摆脱这些新行：

```
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}}
```

为了清楚说明这一点，让我们调整上面的内容，将空格替换为 *，按照此规则将每个空格将被删除。一个在该行的末尾的 * 指示换行符将被移除

```
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}}
```

牢记这一点，我们可以通过 Helm 运行我们的模板并查看结果：


```
# Source: mychart/templates/configmap.yaml
apiVersion: v1
kind: ConfigMap
metadata:
  name: clunky-cat-configmap
data:
  myvalue: "Hello World"
  drink: "coffee"
  food: "PIZZA"
  mug: true
```

小心使用 `chomping` 修饰符。这样很容易引起以外：

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

这将会产生 `food: "PIZZA"mug:true`，因为它删除了双方的换行符。

有关模板中空格控制的详细信息，请参阅官方 Go 模板文档 [Official Go template documentation](#)

最后，有时候告诉模板系统如何缩进更容易，而不是试图掌握模板指令的间距。因此，有时可能会发现使用 `indent` 函数（`{{indent 2 "mug:true"}}`）会很有用。

使用 `with` 修改范围

下一个要看的控制结构是 `with`。它控制着变量作用域。回想一下，`.` 是对当前范围的引用。因此，`.Values` 告诉模板在当前范围中查找 `Values` 对象。

其语法 `with` 类似于一个简单的 `if` 语句：

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

范围可以改变。`with` 可以允许将当前范围（`.`）设置为特定的对象。例如，我们一直在使用的 `.Values.favorites`。让我们重写我们的 `ConfigMap` 来改变 `.` 范围来指向 `.Values.favorites`：

```
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}}
```

注意，现在我们可以引用 `.drink` 和 `.food` 无需对其进行限定。这是因为该 `with` 声明设置 `.` 为指向 `.Values.favorite`。在 `{{end}}` 后 `.` 复位其先前的范围。

但是请注意！在受限范围内，此时将无法从父范围访问其他对象。例如，下面会报错：

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

```
release: {{.Release.Name}}
{{- end}}
```

它会产生一个错误，因为 `Release.Name` 它不在 `.` 限制范围内。但是，如果我们交换最后两行，所有将按预期工作，因为范围在之后被重置。

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

看下 `range`，我们看看模板变量，它提供了一个解决上述范围问题的方法。

循环 `range` 动作

许多编程语言都支持使用 `for` 循环，`foreach` 循环或类似的功能机制进行循环。在 Helm 的模板语言中，遍历集合的方式是使用 `range` 操作子。

首先，让我们在我们的 `values.yaml` 文件中添加一份披萨配料列表：

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

现在我们有一个列表（模板中称为 `slice`）`pizzaToppings`。我们可以修改我们的模板，将这个列表打印到我们的 `ConfigMap` 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}}
```

让我们仔细看看 `toppings :list`。该 `range` 函数将遍历 `pizzaToppings` 列表。但现在发生了一些有趣的事。就像 `with sets` 的范围 `.`，`range` 操作子也是一样。每次通过循环时，`.` 都设置为当前比萨饼顶部。也就是第一次 `.` 设定 `mushrooms`。第二个迭代它设置为 `cheese`，依此类推。

我们可以直接向管道发送 `.` 的值，所以当我们这样做时 `{{. | title | quote}}`，它会发送 `.` 到 `title`（标题 `case` 函数），然后发送到 `quote`。如果我们运行这个模板，输出将是：

```
# Source: mychart/templates/configmap.yaml
apiVersion: v1
kind: ConfigMap
```

```
metadata:
  name: edgy-dragonfly-configmap
data:
  myvalue: "Hello World"
  drink: "coffee"
  food: "PIZZA"
  toppings: |-
    - "Mushrooms"
    - "Cheese"
    - "Peppers"
    - "Onions"
```

现在，在这个例子中，我们碰到了一些棘手的事情。该 `toppings: |-` 行声明了一个多行字符串。所以我们的 `toppings` list 实际上不是 YAML 清单。这是一个很大的字符串。我们为什么要这样做？因为 ConfigMaps 中的数据 `data` 由键 / 值对组成，其中键和值都是简单的字符串。要理解这种情况，请查看 Kubernetes ConfigMap 文档。但对我们来说，这个细节并不重要。

YAML 中的 `|-` 标记表示一个多行字符串。这可以是一种有用的技术，用于在清单中嵌入大块数据，如此处所示。

有时能快速在模板中创建一个列表，然后遍历该列表是很有用的。Helm 模板有一个功能可以使这个变得简单：`tuple`。在计算机科学中，元组是类固定大小的列表类集合，但是具有任意数据类型。这粗略地表达了 `tuple` 的使用方式。

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

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

除了list和tuple之外，`range` 还可以用于遍历具有键和值的集合（如 `map` 或 `dict`）。当在下一节我们介绍模板变量时，将看到如何做到这一点。

Copyright © Mingo(whmzsu@gmail.com) 2017-2018 all right reserved , powered by GitbookUpdated at 2018-11-24 10:25:18

变量

我们已经了解了函数，管道，对象和控制结构，我们可以在许多编程语言中找到更基本的用法之一：变量。在模板中，它们使用的频率较低。我们将看到如何使用它们来简化代码，并更好地使用 `with` 和 `range`。

在前面的例子中，我们看到这段代码会失败：

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

`Release.Name` 不在该 `with` 块中限制的范围。解决范围问题的一种方法是将对象分配给可以在不考虑当前范围的情况下访问的变量。

在 Helm 模板中，变量是对另一个对象的命名引用。它遵循这个形式 `$name`。变量被赋予一个特殊的赋值操作符 `:=`。我们可以使用变量重写上面的 `Release.Name`。

```
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，在我们开始 `with` 块之前，我们赋值 `$relname := .Release.Name`。现在在 `with` 块内部，`$relname` 变量仍然指向发布名称。

会产生这样的结果：

```
# Source: mychart/templates/configmap.yaml
apiVersion: v1
kind: ConfigMap
metadata:
  name: viable-badger-configmap
data:
  myvalue: "Hello World"
  drink: "coffee"
  food: "PIZZA"
  release: viable-badger
```

变量在 `range` 循环中特别有用。它们可以用于类似列表的对象以同时捕获索引和值：

```
toppings: |-
  {{- range $index, $topping := .Values.pizzaToppings}}
    {{$index}}: {{ $topping }}
  {{- end}}
```

注意，`range` 首先是变量，然后是赋值运算符，然后是列表。这将分配整数索引（从零开始）给 `$index`，值给 `$topping`。运行它将产生：

```
toppings: |-
  0: mushrooms
  1: cheese
  2: peppers
  3: onions
```

对于同时具有键和值的数据结构，我们可以使用 `range` 来获得两者。例如，我们可以对 `.Values.favorite` 像这样循环：

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

现在在第一次迭代中，`$key` 是 `drink`，`$val` 是 `coffee`，第二次，`$key` 是 `food`，`$val` 是 `pizza`。运行上面的代码会生成下面这个：

```
# Source: mychart/templates/configmap.yaml
apiVersion: v1
kind: ConfigMap
metadata:
  name: eager-rabbit-configmap
data:
  myvalue: "Hello World"
  drink: "coffee"
  food: "pizza"
```

变量通常不是“全局”的。它们的范围是它们所在的块。之前，我们在模板的顶层赋值 `$relname`。该变量将在整个模板的范围内起作用。但在我们的最后一个例子中，`$key` 和 `$val` 只会在该 `{{range...}}{{end}}` 块的范围内起作用。

然而，总有一个变量是全局 `$` 变量 - 这个变量总是指向根上下文。当你在需要知道 chart 发行名称的范围内循环时，这非常有用。

举例说明：

```
{{- range .Values.tlsSecrets}}
apiVersion: v1
kind: Secret
metadata:
  name: {{.name}}
labels:
  # Many helm templates would use `.` below, but that will not work,
  # however `$` will work here
  app.kubernetes.io/name: {{template "fullname" $}}
  # I cannot reference .Chart.Name, but I can do $.Chart.Name
  helm.sh/chart: "{{$.Chart.Name}}-{{ $.Chart.Version }}"
  app.kubernetes.io/instance: "{{$.Release.Name}}"
  app.kubernetes.io/managed-by: "{{$.Release.Service}}"
type: kubernetes.io/tls
data:
  tls.crt: {{.certificate}}
  tls.key: {{.key}}
---
{{- end}}
```

到目前为止，我们只查看了一个文件中声明的一个模板。但是Helm模板语言的强大功能之一是它能够声明多个模板并将它们一起使用。我们将在下一节中讨论。

Copyright © Mingo(whmzs@gmail.com) 2017-2018 all right reserved , powered by GitbookUpdated at 2018-11-23 22:31:49

命名模板

现在是开始创建超过一个模板的时候了。在本节中，我们将看到如何在一个文件中定义命名模板，然后在别处使用它们。命名模板（有时称为部分或子模板）是限定在一个文件内部的模板，并起一个名称。我们有两种创建方法，以及几种不同的使用方法。

在“流量控制”部分中，我们介绍了声明和管理模板三个动作：`define`，`template`，和 `block`。在本节中，我们将介绍这三个动作，并介绍一个 `include` 函数，与 `template` 类似功能。

在命名模板时要注意一个重要的细节：模板名称是全局的。如果声明两个具有相同名称的模板，则最后加载一个模板是起作用的模板。由于子 chart 中的模板与顶级模板一起编译，因此注意小心地使用特定 chart 的名称来命名模板。

通用的命名约定是为每个定义的模板添加 chart 名称：`{{define "mychart.labels"}}`。通过使用特定 chart 名称作为前缀，我们可以避免由于同名模板的两个不同 chart 而可能出现的任何冲突。

partials 和 _ 文件

到目前为止，我们已经使用了一个文件，一个文件包含一个模板。但 Helm 的模板语言允许创建指定的嵌入模板，可以通过名称访问。

在我们开始编写这些模板之前，有一些文件命名约定值得一提：

- 大多数文件 `templates/` 被视为包含 Kubernetes manifests
- `NOTES.txt` 是一个例外
- 名称以下划线（`_`）开头的文件被假定为没有内部 manifest。这些文件不会渲染 Kubernetes 对象定义，而是在其他 chart 模板中随处可用以供调用。

这些文件用于存储 partials 和辅助程序。事实上，当我们第一次创建时 mychart，我们看到一个叫做文件 `_helpers.tpl`。该文件是模板 partials 的默认位置。

用 `define` 和 `template` 声明和使用模板

该 `define` 操作允许我们在模板文件内创建一个命名模板。它的语法如下所示：

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

例如，我们可以定义一个模板来封装一个 Kubernetes 标签块：

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

现在我们可以将此模板嵌入到现有的 ConfigMap 中，然后将其包含在 `template` 操作中：

```
{{- 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，它将存储引用 mychart.labels 直到 template "mychart.labels" 被调用。然后它将在文件内渲染该模板。所以结果如下所示：

```

# Source: mychart/templates/configmap.yaml
apiVersion: v1
kind: ConfigMap
metadata:
  name: running-panda-configmap
  labels:
    generator: helm
    date: 2016-11-02
data:
  myvalue: "Hello World"
  drink: "coffee"
  food: "pizza"

```

通常，Helm chart 通常将这些模板放入 partials 文件中，通常是 `_helpers.tpl`。让我们在这里移动这个功能：

```

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

```

按照惯例，define 函数应该有一个简单的文档块（`{{/* ... */}}`）来描述他们所做的事情。

即使这个定义在 `_helpers.tpl`，它仍然可以在 configmap.yaml 以下位置访问：

```

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 }}

```

如上所述，模板名称是全局的。因此，如果两个模板被命名为相同的名称，则最后一次使用的模板将被使用。由于子 chart 中的模板与顶级模板一起编译，因此最好使用 chart 专用名称命名模板。一个流行的命名约定是为每个定义的模板添加 chart 名称：`{{define "mychart.labels"}}`。

设置模板的范围

在我们上面定义的模板中，我们没有使用任何对象。我们只是使用函数。让我们修改我们定义的模板以包含 chart 名称和 chart 版本：


```

{{/* Generate basic labels */}}
{{- define "mychart.labels" }}
  labels:
    generator: helm
    date: {{ now | htmlDate }}
    chart: {{ .Chart.Name }}
    version: {{ .Chart.Version }}
{{- end }}

```

如果我们这样做，将不会得到我们所期望的结果：

```

# Source: mychart/templates/configmap.yaml
apiVersion: v1
kind: ConfigMap
metadata:
  name: moldy-jaguar-configmap
  labels:
    generator: helm
    date: 2016-11-02
    chart:
    version:

```

名称和版本发生了什么变化？他们不在我们定义的模板的范围内。当一个已命名的模板（用于创建 define）被渲染时，它将接收由该 template 调用传入的作用域。在我们的例子中，我们包含了这样的模板：

```

{{- template "mychart.labels" }}

```

没有范围被传入，因此在模板中我们无法访问任何内容。虽然这很容易解决。我们只需将范围传递给模板：

```

apiVersion: v1
kind: ConfigMap
metadata:
  name: {{ .Release.Name }}-configmap
  {{- template "mychart.labels" . }}

```

请注意，我们在条用 template 时末尾传递了 `.`。我们可以很容易地通过 `.Values` 或者 `.Values.favorite` 或者我们想要的任何范围。但是我们想要的是顶级范围。

现在，当我们用 `helm install --dry-run --debug ./mychart` 执行这个模板，我们得到这个：

```

# Source: mychart/templates/configmap.yaml
apiVersion: v1
kind: ConfigMap
metadata:
  name: plinking-anaco-configmap
  labels:
    generator: helm
    date: 2016-11-02
    chart: mychart
    version: 0.1.0

```

现在 `{{.Chart.Name}}` 解析为 `mychart`，`{{.Chart.Version}}` 解析为 `0.1.0`。

include 函数

假设我们已经定义了一个如下所示的简单模板：

```

{{- define "mychart.app" -}}

```

```

app_name: {{ .Chart.Name }}
app_version: "{{ .Chart.Version }}" + {{ .Release.Time.Seconds }}"
{{- end -}}

```

现在我想插入到我的模板的 `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" . }}

```

输出不是我们所期望的：

```

# Source: mychart/templates/configmap.yaml
apiVersion: v1
kind: ConfigMap
metadata:
  name: measly-whippet-configmap
  labels:
    app_name: mychart
app_version: "0.1.0+1478129847"
data:
  myvalue: "Hello World"
  drink: "coffee"
  food: "pizza"
  app_name: mychart
app_version: "0.1.0+1478129847"

```

注意，`app_version` 缩进在两个地方都是错误的。为什么？因为被替换的模板具有与右侧对齐的文本。因为 `template` 是一个动作，而不是一个函数，所以没有办法将 `template` 调用的输出传递给其他函数；数据只是内嵌插入。

为了解决这个问题，Helm 提供了一个替代 `template` 方案，将模板的内容导入到当前管道中，并将其传递到管道中的其函数。

这里是上面的例子，用 `indent` 纠正正确缩进 `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 每个部分都正确缩进：

```

# Source: mychart/templates/configmap.yaml
apiVersion: v1
kind: ConfigMap

```

```
metadata:
  name: edgy-mole-configmap
  labels:
    app_name: mychart
    app_version: "0.1.0+1478129987"
data:
  myvalue: "Hello World"
  drink: "coffee"
  food: "pizza"
  app_name: mychart
  app_version: "0.1.0+1478129987"
```

在 Helm 模板中使用 `include` 比 `template` 会更好，可以更好地为 YAML 处理输出格式。

有时我们想要导入内容，但不是作为模板。也就是说，我们要逐字输入文件。我们下一节中描述可以通过访问 `.Files` 的对象来读取文件。

Copyright © Mingo(whmzsu@gmail.com) 2017-2018 all right reserved , powered by GitbookUpdated at 2018-06-29 10:04:26

模板内访问文件

在上一节中，我们介绍了几种创建和访问命名模板的方法。这可以很容易地从另一个模板中导入一个模板。但有时需要导入不是模板的文件，并注入其内容而不通过模板渲染器发送内容。

Helm 通过 `.Files` 对象提供对文件的访问。在我们开始使用模板示例之前，需要注意一些关于它如何工作的内容：

- 向 Helm chart 添加额外的文件是可以的。这些文件将被捆绑并发送给 Tiller。不过要注意，由于 Kubernetes 对象的存储限制，chart 必须小于 1M。
- 通常出于安全原因，某些文件不能通过 `.Files` 对象访问。
 - `templates/` 无法访问文件。
 - 使用 `.helmignore` 排除的文件不能被访问。
- chart 不保留 UNIX 模式信息，因此文件级权限在涉及 `.Files` 对象时不会影响文件的可用性。

- [基本示例](#)
- [路径助手](#)
- [Glob 模式](#)
- [ConfigMap 和 Secrets 工具函数](#)
- [编码](#)
- [行](#)

基本示例

留意这些注意事项，我们编写一个模板，从三个文件读入我们的 ConfigMap。首先，我们将三个文件添加到 chart 中，将所有三个文件直接放在 `mychart/` 目录中。

`config1.toml`：

```
message = Hello from config 1
```

`config2.toml`：

```
message = This is config 2
```

`config3.toml`：

```
message = Goodbye from config 3
```

这些都是一个简单的 TOML 文件（想想老派的 Windows INI 文件）。我们知道这些文件的名称，所以我们可以使用一个 `range` 函数来遍历它们并将它们的内容注入到我们的 ConfigMap 中。

```
apiVersion: v1
kind: ConfigMap
metadata:
  name: {{.Release.Name}}-configmap
data:
  {{- $files := .Files }}
  {{- range tuple "config1.toml" "config2.toml" "config3.toml"}}
  {{.}}: |-
    {{$files.Get .}}
  {{- end}}
```

这个配置映射使用了前几节讨论的几种技术。例如，我们创建一个 `$files` 变量来保存 `.Files` 对象的引用。我们还使用该 `tuple` 函数来创建我们循环访问的文件列表。然后我们打印每个文件名（`{{.}}: |-`），然后打印文件的内容 `{{ $files.Get . }}`。

运行这个模板将产生一个包含所有三个文件内容的 ConfigMap：

```
# Source: mychart/templates/configmap.yaml
apiVersion: v1
kind: ConfigMap
metadata:
  name: quieting-giraf-configmap
data:
  config1.toml: |-
    message = Hello from config 1

  config2.toml: |-
    message = This is config 2

  config3.toml: |-
    message = Goodbye from config 3
```

路径助手

在处理文件时，对文件路径本身执行一些标准操作会非常有用。为了协助这个能力，Helm 从 Go 的 `path` 包中导入了许多函数供使用。它们都可以使用 Go 包中的相同名称访问，但使用时小写第一个字母，例如，`Base` 变成 `base`，等等

导入的功能是：

- `Base`
- `Dir`
- `Ext`
- `IsAbs`
- `Clean`

Glob 模式

随着 chart 的增长，可能会发现需要组织更多地文件，因此我们提供了一种 `Files.Glob(pattern string)` 方法通过具有灵活性的模式 `glob patterns` 协助提取文件。

`.Glob` 返回一个 `Files` 类型，所以可以调用 `Files` 返回对象的任何方法。

例如，想象一下目录结构：

```
foo/:
  foo.txt foo.yaml

bar/:
  bar.go bar.conf baz.yaml
```

Globs 有多个方法可选择：

```
{{ $root := . }}
{{ range $path, $bytes := .Files.Glob "**.yaml" }}
{{ $path }}: |-
{{ $root.Files.Get $path }}
{{ end }}
```

或

```
{{range $path, $bytes := .Files.Glob "foo/*"}}
  {{$path.base}}: '{{ $root.Files.Get $path | b64enc }}'
{{end}}
```

ConfigMap 和 Secrets 工具函数

(不存在于 2.0.2 或更早的版本中)

想要将文件内容放置到 configmap 和 secret 中非常常见，以便在运行时安装到 pod 中。为了解决这个问题，我们在这个 `Files` 类型上提供了一些实用的方法。

为了进一步组织文件，将这些方法与 `Glob` 方法结合使用尤其有用。

根据上面的 `Glob` 示例中的目录结构：

```
apiVersion: v1
kind: ConfigMap
metadata:
  name: conf
data:
  '{{- (.Files.Glob "foo/*").AsConfig | nindent 2 }}'
---
apiVersion: v1
kind: Secret
metadata:
  name: very-secret
type: Opaque
data:
  '{{(.Files.Glob "bar/*").AsSecrets | nindent 2 }}'
```

编码

我们可以导入一个文件，并使用 base64 对模板进行编码以确保成功传输：

```
apiVersion: v1
kind: Secret
metadata:
  name: '{{.Release.Name}}-secret'
type: Opaque
data:
  token: |-
    '{{.Files.Get "config1.toml" | b64enc}}'
```

以上例子将采用 `config1.toml` 文件，我们之前使用的相同文件并对其进行编码：

```
# Source: mychart/templates/secret.yaml
apiVersion: v1
kind: Secret
metadata:
  name: lucky-turkey-secret
type: Opaque
data:
  token: |-
    bWVzc2FnZSA9IEh1bGxvIGZyb20gY29uZm1nIDEK
```

行

有时需要访问模板中文件的每一行。`Lines` 为此提供了一种方便的方法。

```
data:
  some-file.txt: {{range .Files.Lines "foo/bar.txt"}}
    {{.}}{{ end }}
```

目前，无法将 `helm install` 期间将外部文件传递给 chart。因此，如果要求用户提供数据，则必须使用 `helm install -f` 或进行加载 `helm install --set`。

这个讨论将我们深入到写作Helm模板的工具和技术中。在下一节中，我们将看到如何使用一个特殊文件 `templates/NOTES.txt`，向chart的用户发送安装后指导。

Copyright © Mingo(whmzsu@gmail.com) 2017-2018 all right reserved , powered by GitbookUpdated at 2018-11-24 10:23:53

创建一个 NOTES.txt 文件

在本节中，我们将看看 Helm 工具如何向你的 chart 用户提供说明。在 `chart install` 或 `chart upgrade` 结束时，Helm 可以为用户打印出一大堆有用的信息。这些信息是使用模板高度定制的。

要将安装说明添加到 chart，只需创建一个 `templates/NOTES.txt` 文件即可。这个文件是纯文本的，但是它像一个模板一样处理，并且具有所有可用的普通模板函数和对象。

我们来创建一个简单的 `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 {{ .Release.Name }}
```

现在，如果我们运行 `helm install ./mychart` 我们会在底部看到这条消息：

```
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 rude-cardinal
```

使用 `NOTES.txt` 这种方式是一种很好的方式，可以为用户提供有关如何使用新安装chart的详细信息。强烈建议创建一个文件 `NOTES.txt`，尽管这不是必需的。

Copyright © Mingo(whmzsu@gmail.com) 2017-2018 all right reserved , powered by GitbookUpdated at 2018-06-29 10:04:28

子 chart 和全局值

到目前为止，我们只用一个 chart。但是 chart 可以有称为子 chart 的依赖关系，它们也有自己的值和模板。在本节中，我们将创建一个子 chart，并查看我们可以从模板中访问值的不同方式。

在我们深入了解代码之前，需要了解一些有关子 chart 的重要细节。

- 子 chart 被认为是“独立的”，这意味着子 chart 不能明确依赖于其父 chart。
- 因此，子 chart 无法访问其父项的值。
- 父 chart 可以覆盖子 chart 的值。
- Helm 有全局值的概念，可以被所有 chart 访问。

当我们在本节中通过示例时，其中许多概念将变得更加清晰。

创建一个子 chart

对于这些练习，我们将从本指南开始时创建的 chart `mychart/` 开始，并在其中添加一个新 chart。

```
$ cd mychart/charts
$ helm create mysubchart
Creating mysubchart
$ rm -rf mysubchart/templates/*.*
```

注意，和以前一样，我们删除了所有的基本模板，以便我们可以从头开始。在本指南中，我们专注于模板如何工作，而不是管理依赖关系。但 chart 指南有更多关于子 chart 工作的信息。

将值和模板添加到子 chart

接下来，我们为 `mysubchart` chart 创建一个简单的模板和 values 文件。应该已经有一个 `values.yaml` 在文件夹 `mychart/charts/mysubchart` 中了。我们将这样设置：

```
dessert: cake
```

接下来，我们将在下面创建一个新的 ConfigMap 模板 `mychart/charts/mysubchart/templates/configmap.yaml`：

```
apiVersion: v1
kind: ConfigMap
metadata:
  name: {{.Release.Name}}-cfgmap2
data:
  dessert: {{.Values.dessert}}
```

由于每个子 chart 都是独立的 chart，因此我们可以给 `mysubchart` 自行测试：

```
$ helm install --dry-run --debug mychart/charts/mysubchart
SERVER: "localhost:44134"
CHART PATH: /Users/mattbutcher/Code/Go/src/k8s.io/helm/_scratch/mychart/charts/mysubchart
NAME:      newbie-elk
TARGET_NAMESPACE: default
CHART:     mysubchart 0.1.0
MANIFEST:
---
# Source: mysubchart/templates/configmap.yaml
```

```
apiVersion: v1
kind: ConfigMap
metadata:
  name: newbie-elk-cfgmap2
data:
  dessert: cake
```

覆盖父 chart 中的值

我们原来的 chart，`mychart` 现在在其 `mysubchart` 的父 chart。这种关系完全是因为 `mysubchart` 内在 `mychart/charts` 目录中。

由于 `mychart` 是父级，我们可以指定配置 `mychart` 并将配置推入 `mysubchart`。例如，我们可以 `mychart/values.yaml` 像这样修改：

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

mysubchart:
  dessert: ice cream
```

请注意最后两行。该 `mysubchart` 部分内的任何指令都将发送到 `mysubchart` chart。所以如果我们运行 `helm install --dry-run --debug mychart`，我们将看到的一个是 `mysubchartConfigMap`：

```
# Source: mychart/charts/mysubchart/templates/configmap.yaml
apiVersion: v1
kind: ConfigMap
metadata:
  name: unhinged-bee-cfgmap2
data:
  dessert: ice cream
```

顶层的值现在已经覆盖了子 chart 的值。

这里有一个重要的细节需要注意。我们没有改变 `mychart/charts/mysubchart/templates/configmap.yaml` 模板指向 `.Values.mysubchart.dessert`。从该模板的角度来看，该值仍位于 `.Values.dessert`。随着模板引擎一起传递值，它会设置范围。所以对于 `mysubchart` 模板，只有指定给 `mysubchart` 的值才会在 `.Values` 里。

但有时候，确实希望某些值可用于所有模板。这是使用全局 chart 值完成的。

全局 chart 值

全局值是可以从任何 chart 或子 chart 用完全相同的名称访问的值。全局值需要明确声明。不能像使用现有的非全局值一样来使用全局值。

`values` 数据类型有一个保留部分，称为 `values.global`，可以设置全局值。让我们在我们的 `mychart/values.yaml` 文件中设置一个。

```
favorite:
  drink: coffee
  food: pizza
```

```
pizzaToppings:
  - mushrooms
  - cheese
  - peppers
  - onions

mysubchart:
  dessert: ice cream

global:
  salad: caesar
```

因为这样全局值的使用方法，`mychart/templates/configmap.yaml` 和 `mychart/charts/mysubchart/templates/configmap.yaml` 都能够访问该值 `{{.Values.global.salad}}`。

`mychart/templates/configmap.yaml`：

```
apiVersion: v1
kind: ConfigMap
metadata:
  name: {{.Release.Name}}-configmap
data:
  salad: {{.Values.global.salad}}
```

`mysubchart/templates/configmap.yaml`：

```
apiVersion: v1
kind: ConfigMap
metadata:
  name: {{.Release.Name}}-cfgmap2
data:
  dessert: {{.Values.dessert}}
  salad: {{.Values.global.salad}}
```

现在，如果我们运行 dry run，我们会在两个输出中看到相同的值：

```
# Source: mychart/templates/configmap.yaml
apiVersion: v1
kind: ConfigMap
metadata:
  name: silly-snake-configmap
data:
  salad: caesar

---
# Source: mychart/charts/mysubchart/templates/configmap.yaml
apiVersion: v1
kind: ConfigMap
metadata:
  name: silly-snake-cfgmap2
data:
  dessert: ice cream
  salad: caesar
```

全局变量对于传递这样的信息非常有用，但它确实需要一些计划来确保将正确的模板配置为使用全局变量。

与子 chart 共享模板

父 chart 和子 chart 可以共享模板。任何 chart 中的任何定义块都可用于其他 chart。

例如，我们可以像这样定义一个简单的模板：

```
{{- define "labels"}}from: mychart{{ end }}
```

回想一下模板上的标签是如何全局共享的。因此，`labels` chart 可以包含在其他 chart 中。

尽管 chart 开发人员可以选择 `include` 和 `template`，使用 `include` 的一个优点是，`include` 可以动态地引用模板：

```
{{include $mytemplate}}
```

以上例子不会引用 `$mytemplate`。`template` 相反，将只接受一个字符串。

避免使用块

Go 模板语言提供了一个 `block` 关键字，允许开发人员提供一个默认的实现，后续将被覆盖。在 Helm chart 中，块不是重写的最佳工具，因为如果提供了同一个块的多个实现，那么所选哪个是不可预知的。

建议是改为使用 `include`。

Copyright © Mingo(whmzsus@gmail.com) 2017-2018 all right reserved, powered by GitbookUpdated at 2018-11-24 09:45:42

调试模板

调试模板可能会很棘手，因为模板在 Tiller 服务器而不是 Helm 客户端上渲染。然后渲染的模板被发送到 Kubernetes API 服务器，可能由于格式以外的原因，服务器可能会拒绝接收这些 YAML 文件。

有几个命令可以帮助您进行调试。

- `helm lint` 是验证 chart 是否遵循最佳实践的首选工具
- `helm install --dry-run --debug`：我们已经知道了这个窍门。这是让服务器渲染你的模板，然后返回结果清单文件的好方法。
- `helm get manifest`：这是查看服务器上安装的模板的好方法。

当你的 YAML 没有解析，但想看看生成了什么时，检索 YAML 的一个简单方法是注释模板中的问题部分，然后重新运行 `helm install --dry-run --debug`：

```
apiVersion: v1
# some: problem section
# {{ .Values.foo | quote }}
```

以上内容将被完整渲染并返回。

```
apiVersion: v1
# some: problem section
# "bar"
```

这提供了一种快速查看生成的容的方式，而不会由于YAML分析错误而被阻止。

Copyright © Mingo(whmzsu@gmail.com) 2017-2018 all right reserved , powered by GitbookUpdated at 2018-06-29 10:04:20

总结

本指南旨在为你提供 chart 开发人员对如何使用 Helm 模板语言的深入了解。本指南着重介绍模板开发的技术方面。

但是当谈到 chart 的实际日常开发时，本指南还没有涉及很多事情。以下是一些有用的指向其他文档的指南，这些指南将帮助您创建新 chart：

- Kubernetes chart 项目 [Helm Charts project](#) 是 chart 不可缺少的来源。该项目也是 chart 开发中最佳实践的标准。
- Kubernetes 用户指南 [User's Guide](#) 提供了可以使用的各种资源类型的详细示例，从 ConfigMaps 和 Secrets 到 DaemonSet Kubernetes 和 Deployments。
- Helm chart 指南 [Charts Guide](#) 介绍了使用 chart 的工作流程。
- Helm Chart Hooks 指南 [Chart Hooks Guide](#) 解释了如何创建生命周期 hook。
- Helm chart 技巧和窍门文章 [Chart Hooks Guide](#) 提供了一些写 chart 的有用技巧。
- [Sprig documentation](#) 文档介绍了提供了六十余的模板功能。
- 在 Go 模板 [Go template docs](#) 文档详细解释模板语法。
- Schelm 工具 [Schelm tool](#) 是用于调试 chart 一个很好的帮手工具。

有时候，问几个问题，并从经验丰富的开发人员那里获得答案会更容易。最好的地方是在 [Kubernetes Slack Helm](#) 频道：

- [#helm-users](#)
- [#helm-dev](#)
- [#charts](#)

最后，如果在本文中发现错误或遗漏，想要推荐一些新内容或希望参与，请访问Helm项目[The Helm Project](#)。

Copyright © Mingo(whmzsu@gmail.com) 2017-2018 all right reserved , powered by GitbookUpdated at 2018-11-24 09:59:55

附录1：YAML技巧

本指南的大部分内容都集中在编写模板语言。在这里，我们将看看YAML格式。YAML具有一些有用的特性，可以让我们作为模板的作者，可以使我们的模板更少出错并更易于阅读。

标量和集合

根据YAML规范[YAML spec](#)，有两种类型的集合，以及许多标量类型。

这两种类型的集合是maps和sequence：

```
map:
  one: 1
  two: 2
  three: 3

sequence:
- one
- two
- three
```

标量值是单个值（与集合相对）

YAML中的标量类型

在Helm的YAML语言中，值的标量数据类型由一组复杂的规则确定，包括用于资源定义的Kubernetes schema。但是，在推断类型时，以下规则成立。

如果一个整数或浮点数是不加引号的单词，它通常被视为一个数字类型：

```
count: 1
size: 2.34
```

但如果他们被引号括起来，他们被视为字符串：

```
count: "1" # <-- string, not int
size: '2.34' # <-- string, not float
```

布尔值也是如此：

```
isGood: true # bool
answer: "true" # string
```

空值的词是null（not nil）。

请注意，这 `port: "80"` 是有效的YAML，并且将通过模板引擎和YAML分析器，但如果Kubernetes预期port为整数，则会失败。

在某些情况下，您可以使用YAML节点标签强制进行特定的类型推断：

```
coffee: "yes, please"
age: !!str 21
port: !!int "80"
```

在上面，`!!str` 告诉解析器 `age` 是一个字符串，即使它看起来像一个 `int`。而 `port` 被视为一个 `int`，即使它被引号括起来。

YAML中的字符串

我们放在YAML文档中的大部分数据都是字符串。YAML有多种表示字符串的方式。本节将介绍这些方法并演示如何使用其中的一些方法。

有三种内置方式来声明一个字符串：

```
way1: bare words
way2: "double-quoted strings"
way3: 'single-quoted strings'
```

所有内置样式必须位于同一行上。

- 单词没有被引用，并且没有escape。出于这个原因，你必须小心你使用什么字符。
- 双引号的字符串可以使用特定的字符 `\` 进行转义。例如 `"\"Hello\", she said"`。你也可以用换行符换行 `\n`。
- 单引号字符串是“文字”字符串，并且不使用 `\` 转义字符。唯一的转义序列是 `''`，它被解码为一个单独的 `'`。

除了单行字符串外，还可以声明多行字符串：

```
coffee: |
  Latte
  Cappuccino
  Espresso
```

以上将把 `coffee` 的值视为等价于单独字符串 `Latte\nCappuccino\nEspresso\n`。

请注意，`|` 必须后的第一行正确缩进。所以我们可以通过这样做来破坏上面的例子：

```
coffee: |
  Latte
  Cappuccino
  Espresso
```

由于Latte不正确缩进，我们会得到如下错误：

```
Error parsing file: error converting YAML to JSON: yaml: line 7: did not find expected key
```

在模板中，为了防止出现上述错误，在多行文档中放置假“第一行”内容有时更安全：

```
coffee: |
  # Commented first line
  Latte
  Cappuccino
  Espresso
```

请注意，无论第一行是什么，它都将保留在字符串的输出中。因此，例如，如果使用这种技术将文件内容注入到 `ConfigMap` 中，那么该注释应该是任何正在读取该条目的预期类型。

控制多行字符串中的空格

在上面的例子中，我们用来 `|` 表示一个多行字符串。但请注意，我们的字符串的内容后跟着 `\n`。如果我们希望YAML处理器去掉尾随的换行符，我们可以在 `|` 后添加 `-`：


```
coffee: |-
  Latte
  Cappuccino
  Espresso
```

现在的 `coffee` 值将是: `Latte\nCappuccino\nEspresso` (没有尾随 `\n`)。

其他时候, 我们可能希望保留所有尾随空格。我们可以用 `|+` 符号来做到这一点:

```
coffee: |+
  Latte
  Cappuccino
  Espresso
```

```
another: value
```

现在的值 `coffee` 将会是 `Latte\nCappuccino\nEspresso\n\n\n`。

文本块内部的缩进被保留, 并保留换行符:

```
coffee: |-
  Latte
    12 oz
    16 oz
  Cappuccino
  Espresso
```

在上述情况下, `coffee` 将是 `Latte\n 12 oz\n 16 oz\nCappuccino\nEspresso`。

缩进和模板

在编写模板时, 可能会发现自己希望将文件内容注入模板。正如我们在前几章中看到的, 有两种方法可以做到这一点:

- 使用 `{{ .Files.Get "FILENAME" }}` 得到chart中的文件的内容。
- 使用 `{{ include "TEMPLATE" . }}` 渲染模板, 然后其内容放入chart。

将文件插入YAML时, 最好理解上面的多行规则。通常情况下, 插入静态文件的最简单方法是做这样的事情:

```
myfile: |
  {{ .Files.Get "myfile.txt" | indent 2 }}
```

请注意我们如何执行上面的缩进: `indent 2` 告诉模板引擎使用两个空格缩进“myfile.txt”中的每一行。请注意, 我们不缩进该模板行。那是因为如果我们做了, 第一行的文件内容会缩进两次。

折叠多行字符串

有时候你想在你的YAML中用多行代表一个字符串, 但是当它被解释时, 要把它当作一个长行。这被称为“折叠”。要声明一个折叠块, 使用 `>` 代替 `|`:

```
coffee: >
  Latte
  Cappuccino
  Espresso
```

`coffee` 的值将会是 `Latte Cappuccino Espresso\n`。请注意, 除最后一个换行符之外的所有内容都将转换为空格。您可以将空格控件与折叠文本标记组合起来, 因此将替换或去掉所有换行符。

请注意，在折叠语法中，缩进文本将导致行被保留。

```
coffee: >-
  Latte
  12 oz
  16 oz
  Cappuccino
  Espresso
```

以上将产生 `Latte\n 12 oz\n 16 oz\nCappuccino Espresso`。请注意，空格和换行符都还在那里。

将多个文档嵌入到一个文件中

可以将多个YAML文档放入单个文件中。这是通过在一个新文档前加 `---`，在文档结束加 `...` 来完成的

```
---
document: 1
...
---
document: 2
...
```

在许多情况下，无论是 `---` 或 `...` 可被省略。

Helm中的某些文件不能包含多个文档。例如，如果文件内部提供了多个 `values.yaml` 文档，则只会使用第一个文档。

但是，模板文件可能有多个文档。发生这种情况时，文件（及其所有文档）在模板渲染期间被视为一个对象。但是，最终的YAML在被送到Kubernetes之前被分成多个文件。

我们建议每个文件在绝对必要时才使用多个文档。在一个文件中有多个文件可能很难调试。

YAML是JSON的Superset

因为YAML是JSON的超集，所以任何有效的JSON文档都应该是有效的YAML。

```
{
  "coffee": "yes, please",
  "coffees": [
    "Latte", "Cappuccino", "Espresso"
  ]
}
```

以上是下面另一种表达方式：

```
coffee: yes, please
coffees:
- Latte
- Cappuccino
- Espresso
```

这两者可以混合使用（小心使用）：

```
coffee: "yes, please"
coffees: [ "Latte", "Cappuccino", "Espresso"]
```

所有这三个都应该解析为相同的内部表示。

虽然这意味着诸如 `values.yaml` 可能包含JSON数据的文件，但Helm不会将文件扩展名 `.json` 视为有效的后缀。

YAML锚

YAML规范提供了一种方法来存储对某个值的引用，并稍后通过引用来引用该值。YAML将此称为“锚定”：

```
coffee: "yes, please"
favorite: &favoriteCoffee "Cappuccino"
coffees:
  - Latte
  - *favoriteCoffee
  - Espresso
```

在上面，`&favoriteCoffee` 设置一个引用到 `Cappuccino`。之后，该引用被用作 `*favoriteCoffee`。所以`coffees`变成了 `Latte, Cappuccino, Espresso`。

虽然在少数情况下锚点是有用的，但它们的一个方面可能导致细微的错误：第一次使用YAML时，引用被扩展，然后被丢弃。

所以如果我们要解码然后重新编码上面的例子，那么产生的YAML将是：

```
coffee: yes, please
favorite: Cappuccino
coffees:
  - Latte
  - Cappuccino
  - Espresso
```

因为Helm和Kubernetes经常读取，修改并重写YAML文件，锚将会丢失。

Copyright © Mingo(whmzsu@gmail.com) 2017-2018 all right reserved, powered by GitbookUpdated at 2018-06-19 22:04:14

附录 2：Go 数据类型和模板

Helm 模板语言是用强类型的 Go 编程语言实现的。出于这个原因，模板中的变量是强类型的。大多数情况下，变量为以下类型之一：

- 字符串：一串文本
- 布尔：`true` 或 `false`
- 整数：整数值（也有 8,16,32 和 64 位有符号和无符号变体）
- `float64`：一个 64 位浮点值（也有 8,16 和 32 位的变体）
- 一个字节 slice（`[]byte`），通常用于保存（可能）二进制数据
- 结构体：具有属性和方法的对象
- 一个上面类型的片段（索引列表）
- 一个字符串键映射（`map[string]interface{}`），其中的值是上面的类型之一

Go 中还有很多其他类型，有时需要在模板中进行转换。调试对象类型的最简单方法是用 `printf "%t"` 在模板中传递它，该模板将打印类型。另请参阅 `typeof` 和 `kindOf` 函数。

Copyright © Mingo(whmzsu@gmail.com) 2017-2018 all right reserved , powered by Gitbook Updated at 2018-06-29 10:04:19

Chart 最佳实践指南

本指南涵盖了 Helm Team 创建 chart 的最佳实践。它着重于如何构建 chart。

我们主要关注有可能公开部署的 chart 的最佳实践。我们知道许多 chart 仅供内部使用，这些 chart 的作者可能会由于为了其内部利益，可能不拘泥于我们的建议。

目录

- [一般约定](#)：了解 chart 一般约定。
- [values 文件](#)：查看结构化 `values.yaml` 的最佳实践。
- [Template](#)：学习一些编写模板的最佳技巧。
- [Requirement](#)：遵循 `requirements.yaml` 文件的最佳做法。
- [标签和注释](#)：helm 具有标签和注释的传统。
- Kubernetes 资源：
 - [Pod 及其规格](#)：查看使用 pod 规格的最佳做法。
 - [基于角色的访问控制](#)：有关创建和使用服务帐户，角色和角色绑定的指导。
 - [自定义资源](#)：自定义资源（CRDs）有其自己的相关最佳实践。

Copyright © Mingo(whmzsu@gmail.com) 2017-2018 all right reserved , powered by GitbookUpdated at 2018-11-23 22:31:49

一般约定

最佳实践指南的这一部分介绍了一般约定。

Chart 名称

Chart 名称应该是小写字母和数字组成，字母开头：

举例：可以使用破折号 `-`，但在 Helm templates 中使用需要一些小技巧 (查看 [issue #2192](#) 获取更多信息)。+ 这里有一些好例子 [Helm Community Charts](#):

```
drupal
cert-manager
oauth2-proxy
```

Chart 名称中不能使用大写字母和下划线。Chart 名称不应使用点。

包含 chart 的目录必须与 chart 具有相同的名称。因此，chart `cert-manager` 必须在名为 `cert-manager/` 的目录中创建。这不仅仅是一种风格的细节，而是 Helm Chart 格式的要求。

版本号

只要有可能，Helm 使用 [SemVer 2](#) 来表示版本号。（请注意，Docker 镜像 tag 不一定遵循 SemVer，因此被视为该规则的一个例外。）

当 SemVer 版本存储在 Kubernetes 标签中时，我们通常会将其 `+` 字符更改为一个 `_` 字符，因为标签不允许 `+` 标志作为值。

格式化 YAML

YAML 文件应该使用两个空格缩进（而不是制表符）。

单词 Helm，Tiller 和 Chart 的用法

使用 Helm，helm，Tiller 和 tiller 这两个词有一些小的惯例。

- Helm 是指该项目，通常用作总括术语
- `helm` 指的是客户端命令
- Tiller 是后端的专有名称
- `tiller` 是后端二进制运行的名称
- 术语“chart”不需要大写，因为它不是专有名词。

如有疑问，请使用 Helm（大写'H'）。

通过版本限制 Tiller

一个 `Chart.yaml` 文件可以指定一个 `tillerVersion` SemVer 约束：

```
name: mychart
version: 0.2.0
tillerVersion: ">=2.4.0"
```

当模板使用 Helm 旧版本不支持的新功能时，应该设置此限制。虽然此参数将接受复杂的 SemVer 规则，但最佳做法是默认为格式 `>=2.4.0`，其中 2.4.0 引入了 chart 中使用的新功能的版本。

此功能是在 Helm 2.4.0 中引入的，因此任何 2.4.0 版本以下的 Tiller 都会忽略此字段。

Copyright © Mingo(whmzsu@gmail.com) 2017-2018 all right reserved , powered by Gitbook Updated at 2018-11-23 22:31:49

Values

这部分最佳实践指南涵盖了 values 的使用。在指南的这一部分，我们提供关于如何构建和使用 values 的建议，重点在于设计 chart 的 `values.yaml` 文件。

命名约定

变量名称应该以小写字母开头，单词应该用 camelcase 分隔：

正确写法：

```
chicken: true
chickenNoodleSoup: true
```

不正确写法：

```
Chicken: true # initial caps may conflict with built-ins
chicken-noodle-soup: true # do not use hyphens in the name
```

请注意，Helm 的所有内置变量都以大写字母开头，以便将它们与用户定义的 value 区分开来，如：`.Release.Name`，`.Capabilities.KubeVersion`。

展平或嵌套值

YAML 是一种灵活的格式，并且值可以嵌套或扁平化。

嵌套：

```
server:
  name: nginx
  port: 80
```

展平：

```
serverName: nginx
serverPort: 80
```

在大多数情况下，展平应该比嵌套更受青睐。原因是对模板开发人员 and 用户来说更简单。

为了获得最佳安全性，必须在每个级别检查嵌套值：

```
{{if .Values.server}}
  {{default "none" .Values.server.name}}
{{end}}
```

对于每一层嵌套，都必须进行存在检查。但对于展平配置，可以跳过这些检查，使模板更易于阅读和使用。

```
{{default "none" .Values.serverName}}
```

当有大量相关变量时，且至少有一个是非可选的，可以使用嵌套值来提高可读性。

使类型清晰

YAML 的类型强制规则有时是违反直觉的。例如，`foo: false` 与 `foo: "false"` 不一样。`foo: 12345678` 在某些情况下，大整数将被转换为科学记数法。

避免类型转换错误的最简单方法是明确地表示字符串，并隐含其他所有内容。或者，简而言之，引用所有字符串。

通常，为了避免整型转换问题，最好将整型存储为字符串，并在模板中使用 `{{int $value}}` 将字符串转换为整数。

在大多数情况下，显式类型标签受到重视，所以 `foo: !!string 1234` 应该将 `1234` 视为一个字符串。但是，YAML 解析器消费标签，因此类型数据在解析后会丢失。

考虑用户如何使用你的 values

有几种潜在的 values 来源：

- chart 的 `values.yaml` 文件
- 由 `helm install -f` 或 `helm upgrade -f` 提供的 value 文件
- 传递给 `--set` 或的 `--set-string` 标志 `helm install` 或 `helm upgrade` 命令
- 通过 `--set-file` 将文件内容传递给 `helm install` 或 `helm upgrade`

在设计 value 的结构时，请记住 chart 的用户可能希望通过 `-f` 标志或 `--set` 选项覆盖它们。

由于 `--set` 在表现力方面比较有限，编写 `values.yaml` 文件的第一个指导原则可以轻松使用 `--set` 覆盖。

出于这个原因，使用 map 来构建 value 文件通常会更好。

难以配合 `--set` 使用：

```
servers:
- name: foo
  port: 80
- name: bar
  port: 81
```

Helm `<=2.4` 时，以上不能用 `--set` 来表示。在 Helm 2.5 中，访问 `foo` 上的端口是 `--set servers[0].port=80`。用户不仅难以弄清楚，而且如果稍后 `servers` 改变顺序，则容易出错。

使用方便：

```
servers:
  foo:
    port: 80
  bar:
    port: 81
```

访问 `foo` 的端口更为方便：`--set servers.foo.port=80`。

文档'values.yaml'

应该记录'values.yaml'中的每个定义的属性。文档字符串应该以它描述的属性的名称开始，然后至少给出一个单句描述。

不正确：

```
# the host name for the webserver
serverHost = example
serverPort = 9191
```

正确：

```
# serverHost is the host name for the webserver
serverHost = example
# serverPort is the HTTP listener port for the webserver
serverPort = 9191
```

使用参数名称开始每个注释，它使文档易于grep，并使文档工具能够可靠地将文档字符串与其描述的参数关联起来。

Copyright © Mingo(whmzsu@gmail.com) 2017-2018 all right reserved , powered by GitbookUpdated at 2018-11-23 22:31:49

模板

最佳实践指南的这一部分重点介绍模板。

templates 目录结构

templates 目录的结构应如下所示：

- 如果他们产生 YAML 输出，模板文件应该有扩展名 `.yaml`。扩展名 `.tpl` 可用于产生不需要格式化内容的模板文件。
- 模板文件名应该使用横线符号（ `my-example-configmap.yaml` ），而不是 camelcase。
- 每个资源定义应该在它自己的模板文件中。
- 模板文件名应该反映名称中的资源种类。例如 `foo-pod.yaml` ， `bar-svc.yaml`

定义模板的名称

定义的模板（在 `{{define}}` 指令内创建的模板）可以全局访问。这意味着 chart 及其所有子 chart 都可以访问所有使用 `{{ define }}` 创建的模板。

出于这个原因，所有定义的模板名称应该是带有某个 namespace。

正确：

```
{{- define "nginx.fullname"}}  
{{/* ... */}}  
{{end -}}
```

不正确：

```
{{- define "fullname" -}}  
{{/* ... */}}  
{{end -}}
```

强烈建议通过 `helm create` 命令创建新 chart，因为根据此最佳做法自动定义模板名称。

格式化模板

模板应该使用两个空格缩进（不是制表符）。

模板指令在大括号之后和大括号之前应该有空格：

正确：

```
{{.foo}}  
{{print "foo"}}  
{{- print "bar" -}}
```

不正确：

```
{{.foo}}  
{{print "foo"}}
```

```
{{-print "bar"-}}
```

模板应尽可能地填充空格：

```
foo:
  {{- range .Values.items}}
  {{.}}
  {{end -}}
```

块（如控制结构）可以缩进以指示模板代码的流向。

```
{{if $foo -}}
  {{- with .Bar}}Hello{{ end -}}
{{- end -}}
```

但是，由于 YAML 是一种面向空格的语言，因此代码缩进有时经常不能遵循该约定。

生成模板中的空格

最好将生成的模板中的空格保持最小。特别是，许多空行不应该彼此相邻。但偶尔空行（特别是逻辑段之间）很好。

这是最好的：

```
apiVersion: batch/v1
kind: Job
metadata:
  name: example
  labels:
    first: first
    second: second
```

这没关系：

```
apiVersion: batch/v1
kind: Job

metadata:
  name: example

  labels:
    first: first
    second: second
```

但这应该避免：

```
apiVersion: batch/v1
kind: Job

metadata:
  name: example


labels:
  first: first

  second: second
```

Resource Naming in Templates

将 `name` 硬编码到资源中通常被认为是不好的做法。名称对于 `release` 应该是唯一的。因此，我们可能希望通过插入 `release` 名称来生成 `name` 字段，例如：

```
apiVersion: v1
kind: Service
metadata:
  name: {{.Release.Name}}-myservice
```

Or if there is only one resource of this kind then we could use `.Release.Name` or the template `fullname` function defined in `_helpers.tpl` (which uses `release name`):

或者，如果只有一个此类资源，那么可以使用 `.Release.Name` 或 `_helpers.tpl`（使用 `release 名称`）中定义的模板 `fullname` 函数：

```
apiVersion: v1
kind: Service
metadata:
  name: {{template "fullname" .}}
```

尽快如此，可能还存在不会来自固定名称的命名冲突的情况。在这些情况下，固定名称可能使应用程序更容易找到诸如服务之类的资源。如果需要固定名称，那么一种可能的管理方法是通过使用 `values.yaml` 中的 `service.name` 值来显式设置名称（如果提供的话）：

```
apiVersion: v1
kind: Service
metadata:
  {{- if .Values.service.name}}
    name: {{.Values.service.name}}
  {{- else}}
    name: {{template "fullname" .}}
  {{- end}}
```

注释（YAML 注释与模板注释）

YAML 和头盔模板都有注释标记。

YAML 注释：

```
# This is a comment
type: sprocket
```

模板注释：

```
{{- /*
This is a comment.
*/ -}}
```

```
type: frobnitz
```

记录模板功能时应使用模板注释，如解释定义的模板：

```
{{- /*
```

```
mychart.shortname provides a 6 char truncated version of the release name.
*/ -}}
{{define "mychart.shortname" -}}
{{.Release.Name | trunc 6}}
{{- end -}}
```

在模板内部，当 Helm 用户可能（有可能）在调试过程中看到注释时，可以使用 YAML 注释。

```
# This may cause problems if the value is more than 100Gi
memory: {{.Values.maxMem | quote}}
```

上面的注释在用户运行 `helm install --debug` 时可见，而在 `{{- /* */ -}}` 部分中指定的注释不是。

在模板和模板输出中使用 JSON

YAML 是 JSON 的超集。在某些情况下，使用 JSON 语法可以比其他 YAML 表示更具可读性。

例如，这个 YAML 更接近表达列表的正常 YAML 方法：

```
arguments:
- "--dirname"
- "/foo"
```

但是，当折叠为 JSON 列表样式时，它更容易阅读：

```
arguments: ["--dirname", "/foo"]
```

使用 JSON 增加易读性是很好的。但是，不应该使用 JSON 语法来表示更复杂的构造。

在处理嵌入到YAML中的纯JSON时（例如init容器配置），使用JSON格式当然是合适的。

Copyright © Mingo(whmzsu@gmail.com) 2017-2018 all right reserved , powered by GitbookUpdated at 2018-11-24 09:50:30

Requirements 文件

本指南的这一部分介绍了 `requirements.yaml` 文件的最佳实践。

版本

在可能的情况下，使用版本范围，而不是固定到确切版本。建议的默认值是使用补丁级别的版本匹配：

```
version: ~1.2.3
```

这将匹配版本 `1.2.3` 和该版本的任何补丁。换句话说，`~1.2.3` 相当于 `>= 1.2.3, < 1.3.0`

有关完整的版本匹配语法，请参阅 [semver documentation](#)

存储库 URL

如有可能，请使用 `https://` 存储库 URL，然后使用 `http://URL`。

如果存储库已添加到存储库索引文件，则存储库名称可用作 URL 的别名。使用 `alias:` 或 `@` 跟随存储库名称。

文件 URL (`file://...`) 被视为对于由固定部署管道组装的 chart“特殊情况”。正式 Helm 库中是不允许在一个 `requirements.yaml` 使用 `file://` 的。

条件和标签

条件或标签应添加到任何可选的依赖项中。

条件的优选形式是：

```
condition: somechart.enabled
```

`somechart` 是依赖的 chart 名称

当多个子 chart（依赖关系）一起提供可选或可交换功能时，这些图应共享相同的标签。

例如，如果 `nginx` 和 `memcached` 在一起，共同提供性能优化，给 chart 中的主应用程序，并要求已启用该功能时两者都存在，那么他们可能有这样的标记：

```
tags:
- webaccelerator
```

这允许用户使用一个标签打开和关闭该功能。

Copyright © Mingo(whmzsu@gmail.com) 2017-2018 all right reserved , powered by GitbookUpdated at 2018-06-29 10:04:00

标签和注释

最佳实践指南的这一部分讨论了在 chart 中使用标签和注释的最佳做法。

它是一个标签还是一个注释？

在下列条件下，元数据项应该是标签：

- Kubernetes 使用它来识别此资源
- 为了查询系统目的，向操作员暴露是非常有用的。

例如，我们建议使用 `helm.sh/chart: NAME-VERSION` 标签作为标签，以便操作员可以方便地查找要使用的特定 chart 的所有实例。

如果元数据项不用于查询，则应将其设置为注释。

Helm hook 总是注释。

标准标签

下表定义了 Helm chart 使用的通用标签。Helm 本身从不要求特定的标签。标记为 REC 的标签是表示推荐的，应放置在 chart 上以保持全局一致性。那些标记 OPT 是表示可选的。这些都是惯用的或通常使用的，但不是经常用于运维目的。

名称	状态	描述	
<code>app.kubernetes.io/name</code>	REC	This should be the app name, reflecting the entire app. Usually <code>{{template "name" .}}</code> is used for this. This is used by many Kubernetes manifests, and is not Helm-specific.	
<code>helm.sh/chart</code>	REC	This should be the chart name and version: <code>{{.Chart.Name}}-{{.Chart.Version}}</code>	replace "+" with "-"
<code>app.kubernetes.io/managed-by</code>	REC	This should always be set to <code>{{.Release.Service}}</code> . It is for finding all things managed by Tiller.	
<code>app.kubernetes.io/instance</code>	REC	This should be the <code>{{.Release.Name}}</code> . It aid in differentiating between different instances of the same application.	
<code>app.kubernetes.io/version</code>	OPT	The version of the app and can be set to <code>{{.Chart.AppVersion}}</code> .	
<code>app.kubernetes.io/component</code>	OPT	This is a common label for marking the different roles that pieces may play in an application. For example, <code>app.kubernetes.io/component: frontend</code> .	
<code>app.kubernetes.io/part-of</code>	OPT	When multiple charts or pieces of software are used together to make one application. For example, application software and a database to produce a website. This can be set to the top level application being supported.	

获取更多关于 `app.kubernetes.io` 前缀的 Kubernetes labels 的信息 [Kubernetes documentation](#)

Copyright © Mingo(whmzsu@gmail.com) 2017-2018 all right reserved , powered by GitbookUpdated at 2018-11-24 09:45:45

Pod 和 Pod 模板

最佳实践指南的这一部分讨论了如何格式化 chart 清单中的 Pod 和 PodTemplate 部分。

以下（非详尽）资源列表使用 PodTemplates：

- Deployment
- ReplicationController
- ReplicaSet
- DaemonSet
- StatefulSet

镜像

容器镜像应该使用固定标签或镜像的 SHA。它不应该使用的标签 `latest`，`head`，`canary`，或其他设计为“浮动”的标签。

镜像可以在 `values.yaml` 文件中定义，可以很容易地换为镜像地址。

```
image: {{.Values.redisImage | quote}}
```

镜像和标签可以在 `values.yaml` 中定义为两个单独的字段：

```
image: "{{.Values.redisImage}}:{{.Values.redisTag }}"
```

ImagePullPolicy

`helm create` 设置 `imagePullPolicy` 为 `IfNotPresent`，在 `deployment.yaml` 中：

```
imagePullPolicy: {{.Values.image.pullPolicy}}
```

和 `values.yaml` 中：

```
pullPolicy: IfNotPresent
```

同样，Kubernetes 默认 `imagePullPolicy` 为 `IfNotPresent`，如果它根本没有被定义。如果想要的值不是 `IfNotPresent`，只需将 `values.yaml` 中的值更新为所需的值即可。

PodTemplates 应声明选择器

所有的 PodTemplate 部分都应该指定一个选择器。例如：

```
selector:
  matchLabels:
    app.kubernetes.io/name: MyName
template:
  metadata:
    labels:
      app.kubernetes.io/name: MyName
```

这是一个很好的做法，因为它可以使 set 和 pod 之间保持关系。

但对于像Deployment这样的集合来说，这更为重要。如果没有这一点，整套标签将用于选择匹配的pod，如果使用的标签（如版本或发布日期）变化了，则将会导致app中断。

Copyright © Mingo(whmzsu@gmail.com) 2017-2018 all right reserved , powered by GitbookUpdated at 2018-11-23 22:31:49

自定义资源定义

最佳实践指南的这一部分涉及创建和使用自定义资源定义对象。

使用自定义资源定义（CRD）时，区分两个不同的部分很重要：

- 有一个 CRD 的声明。这是一个 YAML 文件，kind 类型为 `CustomResourceDefinition`
- 然后有资源使用 CRD。CRD 定义 `foo.example.com/v1`。任何拥有 `apiVersion: example.com/v1` 和种类 `Foo` 的资源都是使用 CRD 的资源。

在使用资源之前安装 CRD 声明

Helm 优化为尽可能快地将尽可能多的资源加载到 Kubernetes 中。通过设计，Kubernetes 可以采取一整套 manifests，并将它们全部启动在线（这称为 reconciliation 循环）。

但是与 CRD 有所不同。

对于 CRD，声明必须在该 CRDs 种类的任何资源可以使用之前进行注册。注册过程有时需要几秒钟。

方法 1：独立的 chart

一种方法是将 CRD 定义放在一个 chart 中，然后将所有使用该 CRD 的资源放入另一个 chart 中。

在这种方法中，每个 chart 必须单独安装。

方法 2：预安装 hook

要将这两者打包在一起，在 CRD 定义中添加一个 `pre-install` 钩子，以便在执行 chart 的其余部分之前完全安装它。

请注意，如果使用 `pre-install` hook 创建 CRD，则该 CRD 定义在 `helm delete` 运行时不会被删除。

Copyright © Mingo(whmzsu@gmail.com) 2017-2018 all right reserved，powered by Gitbook Updated at 2018-06-29 10:03:48

基于角色的访问控制

最佳实践指南的这一部分讨论了 chart 清单中 RBAC 资源的创建和格式化。

RBAC 资源是：

- ServiceAccount (namespaced)
- Role (namespaced)
- ClusterRole
- RoleBinding (namespaced)
- ClusterRoleBinding

YAML 配置

RBAC 和 ServiceAccount 配置应该在单独的密钥下进行。他们是不同的东西。将 YAML 中的这两个概念拆分出来可以消除混淆并使其更清晰。

```
rbac:
  # Specifies whether RBAC resources should be created
  create: true

serviceAccount:
  # Specifies whether a ServiceAccount should be created
  create: true
  # The name of the ServiceAccount to use.
  # If not set and create is true, a name is generated using the fullname template
  name
```

此结构可以扩展到需要多个 ServiceAccounts 的更复杂的 chart。

```
serviceAccounts:
  client:
    create: true
    name:
  server:
    create: true
    name:
```

RBAC 资源应该默认创建

`rbac.create` 应该是一个布尔值，控制是否创建 RBAC 资源。默认应该是 `true`。想要管理 RBAC 访问控制的用户可以将此值设置为 `false`（在这种情况下请参阅下文）。

使用 RBAC 资源

`serviceAccount.name` 应设置为由 chart 创建的访问控制资源使用的 `ServiceAccount` 的名称。如果 `serviceAccount.create` 为 `true`，则应该创建一个带有该名称的 `ServiceAccount`。如果名称未设置，则使用该 `fullname` 模板生成名称，如果 `serviceAccount.create` 为 `false`，则不应创建该名称，但它仍应与相同的资源相关联，以便稍后通过手动创建的 RBAC 资源将引用它从而功能正常。如果 `serviceAccount.create` 为 `false` 且名称未指定，则使用默认的 `ServiceAccount`。

为 `ServiceAccount` 使用以下 helper 模板。

```
{{/*
Create the name of the service account to use
*/}}
{{- define "mychart.serviceAccountName" -}}
{{- if .Values.serviceAccount.create -}}
    {{ default (include "mychart.fullname" .) .Values.serviceAccount.name }}
{{- else -}}
    {{ default "default" .Values.serviceAccount.name }}
{{- end -}}
{{- end -}}
```

Copyright © Mingo(whmzsu@gmail.com) 2017-2018 all right reserved , powered by GitbookUpdated at 2018-06-29 10:03:55

相关项目和文档

Helm 社区已经制作了许多关于 Helm 的额外工具，插件和文档。我们喜欢听到这些项目。如果有任何想要添加到此列表中的内容，请 [issue](#) 或 [pull request](#)。

文章，博客，操作方法和额外文档

- [Awesome Helm](#) - List of awesome Helm resources
- [CI/CD with Kubernetes, Helm & Wercker](#)
- [Creating a Helm Plugin in 3 Steps](#)
- [Awesome Helm](#) - List of awesome Helm resources
- [Deploying Kubernetes Applications with Helm](#)
- [GitLab, Consumer Driven Contracts, Helm and Kubernetes](#)
- [Honestbee's Helm Chart Conventions](#)
- [Releasing backward-incompatible changes: Kubernetes, Jenkins, Prometheus Operator, Helm and Traefik](#)
- [The Missing CI/CD Kubernetes Component: Helm package manager](#)
- [Using Helm to Deploy to Kubernetes](#)
- [Writing a Helm Chart](#)
- [A basic walk through Kubernetes Helm](#)
- [Tillerless Helm v2](#)

视频, 音频, and Podcast

- [CI/CD with Jenkins, Kubernetes, and Helm](#): AKA "The Infamous Croc Hunter Video".
- [Helm with Michelle Noorali and Matthew Butcher](#): The official Google CloudPlatform Podcast interviews Michelle and Matt about Helm.
- [KubeCon2016: Delivering Kubernetes-Native Applications by Michelle Noorali](#)

Helm 插件

- [App Registry](#) - Plugin to manage charts via the [App Registry specification](#)
- [helm-backup](#) - Plugin which performs backup/restore of releases in a namespace to/from a file
- [Helm Diff](#) - Preview `helm upgrade` as a coloured diff
- [Helm Value Store](#) - Plugin for working with Helm deployment values
- [Technosophos's Helm Plugins](#) - Plugins for GitHub, Keybase, and GPG
- [helm-convert](#) - Plugin to convert charts into Kustomize compatible packages
- [helm-cos](#) - Plugin to manage repositories on Tencent Cloud Object Storage
- [helm-edit](#) - Plugin for editing release's values
- [helm-env](#) - Plugin to show current environment
- [helm-gcs](#) - Plugin to manage repositories on Google Cloud Storage
- [helm-github](#) - Plugin to install Helm Charts from Github repositories
- [helm-hashtag](#) - Plugin for tracking docker tag hash digests as values
- [helm-inject](#) - Plugin for injecting additional configurations during release upgrade
- [helm-k8comp](#) - Plugin to create Helm Charts from hiera using k8comp
- [helm-last](#) - Plugin to show the latest release
- [helm-local](#) - Plugin to run Tiller as a local daemon
- [helm-logs](#) - Plugin to view changed releases over time

- [helm-monitor](#) - Plugin to monitor a release and rollback based on Prometheus/ElasticSearch query
- [helm-nuke](#) - Plugin to destroy all releases
- [helm-plugin-utils](#) - Utility functions to be used within Helm plugins
- [helm-restore](#) - Plugin to restore a deployed release to its original state
- [helm-secrets](#) - Plugin to manage and store secrets safely
- [helm-stop](#) - Plugin for stopping a release pods
- [helm-template](#) - Debug/render templates client-side
- [helm-tiller](#) - Additional commands to work with Tiller
- [helm-unittest](#) - Plugin for unit testing chart locally with YAML
- [Tillerless Helm v2](#) - Helm plugin for using Tiller locally and in CI/CD pipelines We also encourage GitHub authors to use the [helm-plugin](#) tag on their plugin repositories.

其他工具

分布在 Helm 或 Tiller 之上的工具。

- [AppsCode Swift](#) - Ajax friendly Helm Tiller Proxy using [grpc-gateway](#)
- [Armada](#) - Manage prefixed releases throughout various Kubernetes namespaces, and removes completed jobs for complex deployments. Used by the [Openstack-Helm](#) team.
- [Autohelm](#) - Autohelm is *another* simple declarative spec for deploying helm charts. Written in python and supports git urls as a source for helm charts.
- [ChartMuseum](#) - Helm Chart Repository with support for Amazon S3 and Google Cloud Storage
- [Chartify](#) - Generate Helm charts from existing Kubernetes resources.
- [Codefresh](#) - Kubernetes native CI/CD and management platform with UI dashboards for managing Helm charts and releases
- [Cog](#) - Helm chart to deploy Cog on Kubernetes
- [Drone.io Helm Plugin](#) - Run Helm inside of the Drone CI/CD system
- [Helm Chart Publisher](#) - HTTP API for publishing Helm Charts in an easy way
- [Helm.NET](#) - A .NET client for Tiller's API
- [Helmfile](#) - Helmfile is a declarative spec for deploying helm charts
- [Helmsman](#) - Helmsman is a helm-charts-as-code tool which enables installing/upgrading/protecting/moving/deleting releases from version controlled desired state files (described in a simple TOML format).
- [Landscape](#) - "Landscape takes a set of Helm Chart references with values (a desired state), and realizes this in a Kubernetes cluster."
- [Monocular](#) - Web UI for Helm Chart repositories
- [Orca](#) - Advanced CI/CD tool for Kubernetes and Helm made simple.
- [Quay App Registry](#) - Open Kubernetes application registry, including a Helm access client
- [Rudder](#) - RESTful (JSON) proxy for Tiller's API
- [Schelm](#) - Render a Helm manifest to a directory
- [VIM-Kubernetes](#) - VIM plugin for Kubernetes and Helm

集成 Helm

包含 Helm 支持的平台，发行版和服务。

- [Cabin](#) - Mobile App for Managing Kubernetes
- [Fabric8](#) - Integrated development platform for Kubernetes
- [Jenkins X](#) - open source automated CI/CD for Kubernetes which uses Helm for [promoting](#) applications through [environments via GitOps](#)
- [Kubernetic](#) - Kubernetes Desktop Client
- [Qstack](#)

杂项

为 chart 作者和 Helm 用户抓取有用的东西

- [Await](#) - Docker image to "await" different conditions--especially useful for init containers. [More Info](#)

Copyright © Mingo(whmzs@gmail.com) 2017-2018 all right reserved , powered by GitbookUpdated at 2018-11-23 22:31:49

Kubernetes Helm 架构

本文档介绍了 Helm 高层次体系结构。

Helm 的目的

Helm 是管理称为 chart 的 Kubernetes 包的工具。Helm 可以做到以下几点：

- 从头开始创建新 chart
- 将 chart 打包成 chart 归档（tgz）文件
- 与存储 chart 的 chart 存储库交互
- 安装或卸载 chart 到现有的 Kubernetes 集群中
- 管理用 Helm 安装的 chart 的 release 周期

对于 Helm，有三个重要的概念：

- chart 是创建 Kubernetes 应用程序实例所需的一系列信息。
- 配置包含可以合并到打包 chart 以创建可发布对象的配置信息。
- Release 是一个的运行实例的 chart，具有特定的组合配置。

组件

Helm 有两个主要部分：

Helm Client 是最终用户的命令行客户端。客户端负责以下部分：

- 本地 chart 开发
- 管理存储库
- 与 Tiller 服务交互
- 发送要安装的 chart
- 查询有关发布的信息
- 请求升级或卸载现有 release

Tiller Server 是一个集群内服务，与 Helm 客户端进行交互，并与 Kubernetes API 服务进行交互。服务负责以下内容：

- 监听来自 Helm 客户端的传入请求
- 结合 chart 和配置来构建发布
- 将 chart 安装到 Kubernetes 中，然后跟踪后续 release
- 通过与 Kubernetes 交互来升级和卸载 chart

简而言之，客户端负责管理 chart，而服务端负责管理 release。

部署

Helm 客户端使用 Go 编程语言编写，并使用 gRPC 协议套件与 Tiller 服务进行交互。

Tiller 服务也用 Go 编写。它提供了一个与客户端连接的 gRPC 服务，它使用 Kubernetes 客户端库与 Kubernetes 进行通信。目前，该库使用 REST + JSON。

Tiller 服务将信息存储在位于 Kubernetes 内的 ConfigMaps 中。它不需要自己的数据库。

如有可能，配置文件用YAML编写。

Copyright © Mingo(whmzs@gmail.com) 2017-2018 all right reserved , powered by GitbookUpdated at 2018-06-29 10:00:33

开发者指南

本指南解释了如何设置环境来开发 Helm 和 Tiller。

前提条件

- Go 的最新版本
- 最新版本的 Glide
- Kubernetes 集群和 kubectl (可选)
- gRPC 工具链
- Git

构建 Helm/tiller

我们使用 Make 来构建我们的程序。最简单的入门方法是：

```
$ make bootstrap build
```

注意：如果不从路径 `$GOPATH/src/k8s.io/helm` 运行，命令将会失败。目录 `k8s.io` 不应该是符号链接，否则 `build` 找不到相关的包。

这将构建 Helm 和 Tiller。`make bootstrap` 将尝试安装某些工具，如果它们不存在的话。

要运行所有测试（无需运行测试 `vendor/`），请运行 `make test`。在容器环境运行所有测试，请运行 `make docker-test`

要在本地运行 Helm 和 Tiller，可以运行 `bin/helm` 或 `bin/tiller`。

- 已知 Helm 和 Tiller 可在 macOS 和大多数 Linux，包括 Alpine 上运行。
- Tiller 必须能够访问 Kubernetes 群集。它通过检查使用 kubectl 的 Kube 配置文件来了解群集信息。

手册页

手册页和 Markdown 文档已经在 `docs/` 预先建好。可以使用重新生成文档 `make docs`。

要将 Helm 手册页公开给 `man` 客户端，您可以将这些文件放入 `$MANPATH`：

```
$ export MANPATH=$GOPATH/src/k8s.io/helm/docs/man:$MANPATH
$ man helm
```

gRPC 和 Protobuf

Helm 和 Tiller 使用 gRPC 进行通信。要开始使用 gRPC，需要如下准备

- 安装 protoc 编译 protobuf 文件。发布在 [这里](#)
- 运行 Helm 的 `make bootstrap` 来生成 `p_rotoc-gen-go` 插件并将其放入 `bin/`。

请注意，需要使用 protobuf 3.2.0 (`protoc --version`) 的版本。protoc-gen-go 版本与 Kubernetes 使用 GRPC 的版本绑定。所以这个插件是在本地维护的。

虽然 gRPC 和 ProtoBuf 规范缩进时没有规定，但我们要求缩进样式与 Go 格式规范相匹配。也就是说，协议缓冲区应该使用基于标签的缩进，并且 rpc 声明应该遵循 Go 函数声明的风格。

Helm API (HAPI)

我们使用 gRPC 作为 API 层。请参阅 `pkg/proto/hapi` 生成的 Go 代码以及 `_proto` 协议缓冲区定义。

要从 protobuf 源重新生成 Go 文件，使用 `make protoc`。

Docker 镜像

要构建 Docker 镜像，请使用 `make docker-build`。

预构建的镜像已经在官方 Kubernetes Helm GCR registry 中提供。

运行本地群集

对于开发，我们强烈建议使用 [Kubernetes Minikube](#) 这个面向开发人员的发行版。安装完成后，可以使用 `helm init` 安装到群集中。请注意，用于开发的 Tiller 版本可能无法在 Google Cloud Container Registry 中使用。如果遇到镜像 Pull 错误，可以覆盖 Tiller 的版本。例：

```
helm init --tiller-image=gcr.io/kubernetes-helm/tiller:2.7.2
```

或使用最新版本：

```
helm init --canary-image
```

为了在 Tiller 上进行开发，在本地运行 Tiller 有时更方便，而不是将其打包到镜像中并在群集中运行。你可以通过告诉 Helm 客户端使用一个本地实例来做到这一点。

```
$ make build
$ bin/tiller
```

要配置 Helm 客户端，请使用 `--host` 标志或导出 `HELM_HOST` 环境变量：

```
$ export HELM_HOST=localhost:44134
$ helm install foo
```

（ 请注意，直接运行 Tiller 时不需要使用 `helm init` ）

Tiller 应该在 `>= 1.3` Kubernetes 群集上运行。

贡献指南

我们欢迎捐款。这个项目已经制定了一些指导方针，以确保（a）代码质量仍然很高，（b）项目保持一致，（c）贡献遵循开源法律要求。我们的目的不是为了给贡献者带来负担，而是为了构建优雅和高质量的开源代码，让我们的用户受益。

确保你已经阅读并理解了贡献指南：

<https://github.com/helm/blob/master/CONTRIBUTING.md>

代码的结构

Helm 项目的代码组织如下：

- 单独的程序位于 `cmd/`。`cmd/` 里面的代码 不是为库的重复使用而设计的。
- 共享库存储在 `pkg/`。
- 原始 ProtoBuf 文件存储在 `_proto/hapi`（`hapi` 表示 Helm 应用程序编程接口）。
- 从 proto 定义生成的 Go 文件存储在 `pkg/proto`。
- `scripts/` 目录包含许多实用程序脚本。其中大部分由 CI/CD 管道使用。
- `rootfs/` 文件夹用于 Docker 特定的文件。
- `docs/` 文件夹用于文档和示例。

Go 依赖关系由 Glide 管理 并存储在 `vendor/` 目录中。

Git 约定

我们将 Git 用于版本控制系统。Master 分支是当前发展候选目录。发布版本有标记。

我们通过 GitHub Pull Requests（PR）接受对代码的更改。执行此操作的一个工作流程如下所示：

1. 转到 `$GOPATH/src/k8s.io` 目录，`git clone github.com/helm` 存储库。
2. 将该存储库 fork 到你自己的 GitHub 帐户
3. 将你的存储库添加为远程服务器 `$GOPATH/src/k8s.io/helm`
4. 创建一个新的工作分支（`git checkout -b feat/my-feature`）并在该分支上完成工作。
5. 当准备好让我们 review 时，将你的分支推送到 GitHub，然后给我们提交一个新的 PR 请求。

对于 Git 提交消息，我们遵循语义提交消息 [Semantic Commit Messages](#)：

```
fix(helm): add --foo flag to 'helm install'
```

When 'helm install --foo bar' is run, this will print "foo" in the output regardless of the outcome of the installation.

Closes #1234

常见提交类型：

- fix: 修复一个 bug 或错误
- feat: 添加一项新功能
- docs: 更改文档
- test: 改进测试
- ref: 重构现有的代码

常见范围：

- helm：Helm CLI
- tiller：Tiller 服务器
- proto：Protobuf 定义
- pkg/lint：lint 包。对于任何软件包遵循类似的约定
- *：两个或更多范围

更多内容：

- DEIS 准则 [Deis Guidelines](#) 是这一部分的灵感启发。
- Karma Runner [定义](#) 了语义提交消息的主意。

Go 语言约定

我们非常密切地遵循 Go 编码风格标准。通常，运行 `go fmt` 会让你的代码更加漂亮。

我们通常也遵循由 `go lint` 和 `gometalinter` 推荐的约定。运行 `make test-style` 以测试样式一致性。

如果你不想将 `gometalinter` 中的所有 linters 安装到你的全局 Go 环境中，你可以运行 `make docker-test-style`，它将运行相同的测试，但是在 docker 容器中是隔离的。

更多阅读：

- Effective Go [introduces formatting](#)。
- Go Wiki 有一个关于格式化的非常好的文章 [formatting](#)。

Protobuf 约定

由于项目主要是 Go 代码，因此我们尽可能将 Protobuf 文件格式化为 Go。目前 Protobuf 没有真正的格式规则或准则，但随着它们的出现，我们可能会选择遵循这些规则或准则。

标准：

- Tab 缩进，而不是空格。
- 间距规则遵循 Go 约定（线条末端的花括号，运算符周围的空格）。

约定：

- 文件应该用 `option go_package = "...";` 来指定它们的包
- 注释应该转化为良好的 Go 代码注释（因为 `protoc` 会拷贝注释到目标源代码文件中）。
- RPC 功能在它们的请求 / 响应消息的同一文件中定义。
- 不推荐使用的RPC，消息和字段在注释中不推荐使用（`// UpdateFoo DEPRECATED updates a foo.`）。

Copyright © Mingo(whmzsus@gmail.com) 2017-2018 all right reserved，powered by GitbookUpdated at 2018-11-23 22:31:49

项目的历史

Kubernetes Helm 是 [Helm Classic](#) 和 GCS Deployment Manager 的 Kubernetes 移植的合并结果。该项目由 Google 和 Deis 联合创建，尽管它现在是 CNCF 的一部分。许多公司现在定期为 Helm 做出贡献。

与 Helm Classic 的区别：

- Helm 现在有一个客户端 (`helm`) 和一个服务端 (`tiller`)。服务端在 Kubernetes 内部运行，并管理资源。
- Helm 的 chart 格式更好：
 - 依赖关系是不可变的，并存储在 chart 的 `charts/` 目录中。
 - chart 使用 [SemVer 2](#) 版本标准化
 - chart 可以从目录或 chart 归档文件中加载
 - Helm 支持 Go 模板，无需运行 `generate` 或 `template` 命令。
 - Helm 可以轻松配置 release - 并与团队的其他成员共享配置。
- Helm chart 存储库现在使用普通的 HTTP (S) 而不是 Git / GitHub。不再有任何 GitHub 依赖。
 - chart 存储库服务是一个简单的 HTTP 服务器
 - chart 由版本引用
 - `helm serve` 命令将运行本地 chart 服务器，也可以轻松使用对象存储 (S3, GCS) 或常规 Web 服务器。
 - 而且仍然可以从本地目录加载 chart。
- Helm 工作空间不存在了。现在可以在想要工作的文件系统上的任何地方工作。

Copyright © Mingo(whmzsu@gmail.com) 2017-2018 all right reserved , powered by GitbookUpdated at 2018-06-29 10:00:56

Helm 词汇表

Helm 使用一些特殊术语来描述体系结构的组件。

Chart

包含足以将一组 Kubernetes 资源安装到 Kubernetes 集群中的信息的 Helm 软件包。

Chart 包含 Chart.yaml 文件以及模板，默认值（values.yaml）和依赖关系。

Chart 是在定义良好的目录结构中开发的，然后打包成一个称为 chart 压缩包的压缩格式。

Chart 压缩包

一个 chart 压缩包是一个 tar 打包和 gzip 压缩（签名可选）的 chart。

Chart 依赖（Subcharts）

Chart 可能依赖于其他 chart。有两种方式可能会出现依赖性：

- 软依赖性：如果没有在集群中安装另一个 chart，chart 可能无法正常工作。Helm 不为这种情况提供工具。在这种情况下，依赖关系可以单独管理。
- 硬性依赖性：chart 可能包含（在其 charts/ 目录内）其所依赖的另一个 chart。在这种情况下，安装 chart 将安装它的所有依赖关系。在这种情况下，chart 及其依赖关系作为集合进行管理。

当一个 chart 打包（通过 `helm package`）时，它的所有硬依赖关系都与它捆绑在一起。

Chart 版本

根据 [SemVer 2 spec](#) 规范对 chart 进行版本控制。每个 chart 上都需要一个版本号。

Chart.yaml

有关 chart 的信息存储在名为 Chart.yaml 的特殊文件中。每个 chart 都必须有这个文件。

Helm（和 helm）

Helm 是 Kubernetes 的软件包管理员。由于操作系统软件包管理器可以轻松在 OS 上安装工具，因此 Helm 可以轻松将应用程序和资源安装到 Kubernetes 群集中。

虽然 Helm 是该项目的名称，命令行客户端也被命名 helm。按照惯例，当谈到这个项目时，Helm 被大写。在谈到客户时，掌舵是小写的。

Helm Home (HELM_HOME)

Helm 客户端将信息存储在称为 helm home 的本地目录中。默认情况下，这是在 `$HOME/.helm` 目录中。

该目录包含配置和缓存数据，并由 `helm init` 创建。

Kube Config (KUBECONFIG)

Helm 客户端通过使用 Kube 配置文件格式的文件来了解 Kubernetes 集群。默认情况下，Helm 尝试在 `kubect1` 创建的地方找到这个文件 (`$HOME/.kube/config`)。

Lint (Linting)

Lint chart 是验证它遵循约定和 Helm chart 标准的要求。Helm 提供了执行此操作的工具，特别是 `helm lint` 命令。

出处 (出处文件)

Helm chart 可能伴随着一个出处文件，该文件提供关于 chart 来自哪里以及它包含什么的信息。

出处文件是 Helm 安全的一部分。出处包含 chart 压缩文件，Chart.yaml 数据和签名块 (OpenPGP“clearsign”块) 的加密哈希。当与钥匙串结合使用时，这为 chart 用户提供了以下功能：

- 验证 chart 是否由可信方签署
- 验证 chart 文件没有被篡改
- 验证 chart 元数据 (Chart.yaml) 的内容
- 快速将 chart 与其出处数据进行匹配

Provenance 文件具有 .prov 扩展名，可以从 chart 存储库服务器或任何其他 HTTP 服务器提供。

Release

当安装 chart 时，Tiller (Helm 服务器) 创建一个 Release 来跟踪该安装。

单个 chart 可以多次安装到同一个群集中，并创建许多不同的 release。例如，可以通过 `helm install` 以不同的 release 名称运行三次来安装三个 PostgreSQL 数据库。

(在 2.0.0-Alpha.1 之前，release 被称为 deployment，但这造成了与 Kubernetes Deployment 类型的混淆。)

Release 版本号

单个版本可以多次更新。顺序计数器用于在 release 更改时跟踪 release。通过 `helm install` 第一次安装后，release 版本的版本号为 1。每次发布 release 升级或回滚时，版本号都会增加。

回滚

Release 可以升级到更新 chart 或配置。但是，由于发布历史已存储，release 版本也可以回滚到以前的版本号。这是通过 `helm rollback` 命令完成的。

重要的是，回滚版本将获得新版本号。

Operation	Release Number
install	release 1
upgrade	release 2

upgrade	release 3
rollback 1	release 4 (但运行与 release 1 相同的配置)

上表说明了如何在安装，升级和回滚都会增加版本号。

Tiller

Tiller 是 Helm 的集群组件。它直接与 Kubernetes API 服务器交互以安装，升级，查询和删除 Kubernetes 资源。它还存储代表 release 的对象。

Repository (Repo , Chart Repository)

Helm chart 可以存储在专用的 HTTP 服务器上，称为 chart 存储库（存储库或库）。

chart 存储库服务器是一个简单的 HTTP 服务器，可以提供 index.yaml 描述一批 chart 的文件，并提供有关每个 chart 可从哪里下载的信息。（许多 chart 存储库同时保存 chart 以及 index.yaml 文件。）

Helm 客户端可以指向零个或多个 chart 存储库。默认情况下，Helm 客户端指向 stable 官方 Kubernetes chart 存储库。

Values (值文件 , values.yaml)

Values 提供了一种用自己的信息覆盖模板默认值的方法。

Helm chart 是“参数化”的，这意味着 chart 开发人员可能会公开可在安装时被覆盖的配置。例如，chart 可能会公开 username 允许为服务设置用户名的字段。

这些暴露的变量在 Helm 说法中被称为值 values。

values 可以在 `helm install` 和 `helm upgrade` 操作时设置，或通过直接，或通过上传 values.yaml 文件。

Copyright © Mingo(whmzsu@gmail.com) 2017-2018 all right reserved , powered by Gitbook Updated at 2018-06-29 10:00:52