16Helm 专题（四） Chart 是个啥

Helm 专题（四） Chart 是个啥

本文主要解释说明 chart 格式，以及使用 Helm 构建 chart 的基本指导。

Helm 使用的包格式称为 `chart。` chart就是一个描述Kubernetes相关资源的文件集合。单个chart可以用来部署一些简单的，类似于memcache pod，或者某些复杂的HTTP服务器以及web全栈应用、数据库、缓存等等。

Chart是作为特定目录布局的文件被创建的。它们可以打包到要部署的版本存档中。

如果你想下载和查看一个发布的chart，但不安装它，你可以用这个命令： `helm pull chartrepo/chartname` 。

Chart 文件结构

chart 是一个组织在文件目录中的集合。 `目录名称就是chart名称` （没有版本信息）。因而描述WordPress的chart可以存储在 `wordpress/` 目录中。

在这个目录中，Helm 期望可以匹配以下结构：

```
wordpress/
  Chart.yaml          # 包含了chart信息的YAML文件
  LICENSE             # 可选：包含chart许可证的纯文本文件
  README.md           # 可选：可读的README文件
  values.yaml         # chart 默认的配置值
  values.schema.json  # 可选：一个使用JSON结构的values.yaml文件
  charts/             # 包含chart依赖的其他chart
  crds/               # 自定义资源的定义
  templates/          # 模板目录， 当和values 结合时，可生成有效的Kubernetes manifest文件
  templates/NOTES.txt # 可选：包含简要使用说明的纯文本文件
```

Chart.yaml 文件

Chart.yaml 文件是 chart 必须的。包含了以下字段：

```yaml
apiVersion: chart API 版本 （必需）
name: chart名称 （必需）
version: 语义化2 版本（必需）
kubeVersion: 兼容Kubernetes版本的语义化版本（可选）
description: 一句话对这个项目的描述（可选）
type: chart类型 （可选）
keywords:
  - 关于项目的一组关键字（可选）
home: 项目home页面的URL （可选）
sources:
  - 项目源码的URL列表（可选）
dependencies: # chart 必要条件列表 （可选）
  - name: chart名称 (nginx)
    version: chart版本 ("1.2.3")
    repository: （可选）仓库URL ("https://example.com/charts") 或别名 ("@repo-name")
    condition: （可选） 解析为布尔值的yaml路径，用于启用/禁用chart (e.g. subchart1.enabled )
    tags: # （可选）
      - 用于一次启用/禁用 一组chart的tag
    import-values: # （可选）
      - ImportValue 保存源值到导入父键的映射。每项可以是字符串或者一对子/父列表项
    alias: （可选） chart中使用的别名。当你要多次添加相同的chart时会很有用
maintainers: # （可选）
  - name: 维护者名字 （每个维护者都需要）
    email: 维护者邮箱 （每个维护者可选）
    url: 维护者URL （每个维护者可选）
icon: 用做icon的SVG或PNG图片URL （可选）
appVersion: 包含的应用版本（可选）。不需要是语义化，建议使用引号
deprecated: 不被推荐的chart （可选，布尔值）
annotations:
  example: 按名称输入的批注列表 （可选）.
```

从 v3.3.2 ，不再允许额外的字段。推荐的方法是在 `annotations` 中添加自定义元数据。

Chart 和版本控制

每个chart都必须有个版本号。版本必须遵循 语义化版本 2(`https://semver.org/spec/v2.0.0.html`) 标准。不像经典 Helm， Helm v2以及后续版本会使用版本号作为发布标记。仓库中的包通过名称加版本号标识。

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

`nginx-1.2.3.tgz`

更多复杂的语义化版本2 都是支持的，比如 `version: 1.2.3-alpha.1+ef365` 。但系统明确禁止非语义化版本名称。

注意： 鉴于经典Helm和部署管理器在使用chart时都非常倾向于GitHub，Helm v2 和后续版本不再依赖或需要GitHub甚至是Git。因此，它完全不使用Git SHA进行版本控制。

`Chart.yaml` 文件中的 `version` 字段被很多Helm工具使用，包括CLI。当生成一个包时， `helm package` 命令可以用 `Chart.yaml` 文件中找到的版本号作为包名中的token。系统假设chart包名中的版本号可以与 `Chart.yaml` 文件中的版本号匹配。如果不满足这一假设会导致错误。

`apiVersion` 字段

对于至少需要Helm 3的chart， `apiVersion` 字段应该是 `v2` 。Chart支持之前 `apiVersion` 设置为 `v1` 的Helm 版本，并且在Helm 3中仍然可安装。

`v1` 到 `v2` 的改变：

- `dependencies` 字段定义了chart的依赖，针对于 `v1` 版本的chart被放置在分隔开的 `requirements.yaml` 文件中 （查看 Chart 依赖(`https://helm.sh/zh/docs/topics/charts/#chart-dependency`)).
- `type` 字段，用于识别应用和库类型的chart （查看 Chart 类型(`https://helm.sh/zh/docs/topics/charts/#chart-types`)).

`kubeVersion` 字段

可选的 `kubeVersion` 字段可以在支持的Kubernetes版本上定义语义化版本约束，Helm 在安装chart时会验证这个版本约束，并在集群运行不支持的Kubernetes版本时显示失败。

版本约束可以包括空格分隔和比较运算符，比如：

```
>= 1.13.0 < 1.15.0
```

或者它们可以用或操作符 `||` 连接，比如：

```
>= 1.13.0 < 1.14.0 || >= 1.14.1 < 1.15.0
```

这个例子中排除了 `1.14.0` 版本，如果确定某些版本中的错误导致chart无法正常运行，这一点就很有意义。

除了版本约束外，使用运算符 `=` `!=` `>` `<` `>=` `<=` 支持一下速记符号：

- 闭合间隔的连字符范围， `1.1 - 2.3.4` 等价于 `>= 1.1 <= 2.3.4`
- 通配符 `x` ， `X` 和``， `1.2.x` 等价于 `>= 1.2.0 < 1.3.0`
- 波浪符号 范围（允许改变补丁版本），`1.2.3 `等价于` >= 1.2.3 < 1.3.0`
- 插入符号^范围（允许改变次版本）， `^1.2.3` 等价于 `>= 1.2.3 < 2.0.0`

支持的语义化版本约束的细节说明请查看 Masterminds/semver(`https://github.com/Masterminds/semver`)

已弃用的Chart

在Chart仓库管理chart时，有时需要废弃一个chart。 `Chart.yaml` 中可选的 `deprecated` 字段可以用来标记已弃用的chart。如果 **latest** 版本被标记为已弃用，则所有的chart都会被认为是已弃用的。以后可以通过发布未标记为已弃用的新版本来重新使用chart名称。弃用chart的工作流是：

1. 升级chart的 `Chart.yaml` 文件，将这个chart标记为已弃用， 并更改版本
2. 在chart仓库中发布新版的chart
3. 从源仓库中移除这个chart （比如用 git）

Chart Types

`type` 字段定义了chart的类型。有两种类型： `application` 和 `library` 。应用是默认类型，是可以完全操作的标准chart。 库类型 chart(`http://helm.sh/zh/docs/topics/library_charts`) 提供针对chart构建的实用程序和功能。库类型chart与应用类型chart不同，因为它不能安装，通常不包含任何资源对象。

注意： 应用类型chart 可以作为库类型chart使用。可以通过将类型设置为 `library` 来实现。然后这个库就被渲染成了一个库类型chart，所有的实用程序和功能都可以使用。所有的资源对象不会被渲染。

Chart 许可证，自述和注释

Chart也可以包含描述安装，配置和使用文件，以及chart许可证。

许可证（LICENSE）是一个包含了chart license 的纯文本文件。chart可以包含一个许可证，因为在模板里不只是配置，还可能有编码逻辑。如果需要，还可以为chart安装的应用程序提供单独的许可证。

chart的自述文件README文件应该使用Markdown格式(README.md)，一般应包含：

- chart提供的应用或服务的描述
- 运行chart的先决条件或要求
- `values.yaml` 的可选项和默认值的描述
- 与chart的安装或配置相关的其他任何信息

`README.md` 文件会包含hub和用户接口显示的chart的详细信息。

chart也会包含一个简短的纯文本 `templates/NOTES.txt` 文件，这会在安装后及查看版本状态时打印出来。这个文件会作为一个 模板 来评估，并用来显示使用说明，后续步骤，或者其他chart版本的相关信息。比如，可以提供连接数据库的说明，web UI 的访问。由于此文件是在运行 `helm install` 或 `helm status` 时打印到STDOUT的， 因此建议保持内容简短，并指向自述文件以获取更多详细信息。

Chart dependency

Helm 中，chart可能会依赖其他任意个chart。这些依赖可以使用 `Chart.yaml` 文件中的 `dependencies` 字段动态链接，或者被带入到 `charts/` 目录并手动配置。

使用 `dependencies` 字段管理依赖

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

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

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

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

```
dependencies:
  - name: awesomeness
    version: 1.0.0
    repository: "@fantastic-charts"
```

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

```
$ 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 https://example.com/charts
Downloading mysql from repo https://another.example.com/charts
```

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

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

依赖的别名字段

除了上面的其他字段之外，每个需求项可以包含一个可选的字段 `alias` 。为依赖chart添加一个别名，会使用别名作为新依赖chart的名称。需要使用其他名称访问chart时可以使用 `alias` 。

```
# parentchart/Chart.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
```

上述例子中，我们会获得 `parentchart` 所有的3个依赖项：

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

手动完成的方式是将同一个chart用不同的名称复制/粘贴多次到 `charts/` 目录中。

依赖中的tag和条件字段

除了上面的其他字段外，每个需求项可以包含可选字段 `tags` 和 `condition` 。

所有的chart会默认加载。如果存在 `tags` 或者 `condition` 字段，它们将被评估并用于控制它们应用的chart的加载。

Condition - 条件字段field 包含一个或多个YAML路径（用逗号分隔）。如果这个路径在上层values中已存在并解析为布尔值，chart会基于布尔值启用或禁用chart。只会使用列表中找到的第一个有效路径，如果路径为未找到则条件无效。

Tags - tag字段是与chart关联的YAML格式的标签列表。在顶层value中，通过指定tag和布尔值，可以启用或禁用所有的带tag的chart。

```
# parentchart/Chart.yaml

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
```

在上面的例子中，所有带 `front-end` tag的chart都会被禁用，但只要上层的value中 `subchart1.enabled` 路径被设置为'true'，该条件会覆盖 `front-end` 标签且 `subchart1` 会被启用。

一旦 `subchart2` 使用了 `back-end` 标签并被设置为了 `true` ， `subchart2` 就会被启用。也要注意尽管 `subchart2` 指定了一个条件字段， 但是上层value没有相应的路径和value，因此这个条件不会生效。

使用带有标签和条件的CLI

`--set` 参数一如既往可以用来设置标签和条件值。

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

标签和条件的解析

- **条件（当设置在value中时）总是会覆盖标签** 第一个chart条件路径存在时会忽略后面的路径。
- 标签被定义为 '如果任意的chart标签是true，chart就可以启用'。
- 标签和条件值必须被设置在顶层value中。
- value中的 `tags:` 键必须是顶层键。全局和嵌套的 `tags:` 表现在不支持了。

通过依赖导入子Value

在某些情况下，允许子chart的值作为公共默认传递到父chart中是值得的。使用 `exports` 格式的额外好处是它可是将来的工具可以自检用户可设置的值。

被导入的包含值的key可以在父chart的 `dependencies` 中的 `import-values` 字段以YAML列表形式指定。列表中的每一项是从子chart中 `exports` 字段导入的key。

导入 `exports` key中未包含的值，使用 子-父 格式。两种格式的示例如下所述。

使用导出格式

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

```
# parent's Chart.yaml file

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


# child's values.yaml file

exports:
  data:
    myint: 99
```

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

最终的父级value会包含我们的导出字段:

```
# parent's values

myint: 99
```

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

Using the child-parent format

要访问子chart中未包含的 `exports` 键的值，你需要指定要导入的值的源键(`child`)和父chart(`parent`)中值的目标路径。

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

```
# parent's Chart.yaml file

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

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

```
# parent's values.yaml file

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


# subchart1's values.yaml file

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

父chart的结果值将会是这样：

```
# parent's final values

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

父chart中的最终值包含了从 subchart1中导入的 `myint` 和 `mybool` 字段。

通过 `charts/` 目录手动管理依赖

如果对依赖进行更多控制，通过将有依赖关系的chart复制到 `charts/` 目录中来显式表达这些依赖关系。

依赖应该是一个解压的chart目录。但是名字不能以 `_` 或 `.` 开头，否则会被chart加载器忽略。

比如，如果WordPress chart依赖于Apache chart，那么（正确版本的）Apache chart需要放在WordPress chart 的 `charts/` 目录中：

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

上面的例子展示了WordPress chart 如何通过将这些chart包含在 `charts/` 目录中来表达它对Apache 和 MySQL的依赖。

提示： *要将依赖放入 `charts/` 目录，使用 `helm pull` 命令*

使用依赖的操作部分

上面的部分说明如何指定chart的依赖，但是对使用 `helm install` 和 `helm upgrade` 安装chart有什么影响？

假设有个chart "A" 创建了下面的Kubernetes对象：

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

另外，A是依赖于chart B创建的对象：

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

安装/升级chart A后，会创建/修改一个单独的Helm版本。这个版本会按顺序创建/升级以下所有的Kubernetes对象：

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

这是因为当Helm安装/升级chart时，chart中所有的Kubernetes对象以及依赖会

- 聚合成一个单一的集合；然后

• 按照类型和名称排序；然后

• 按这个顺序创建/升级。

至此会为chart及其依赖创建一个包含所有对象的release版本。

Kubernetes类型的安装顺序会按照kind_sorter.go(查看 Helm源文件(`https://github.com/helm/helm/blob/484d43913f97292648c867b56768775a55e4bba6/pkg/releaseutil/kind_sorter.go`))中给出的枚举顺序进行。

欢迎关注我的公众号“ **云原生拓展** ”，原创技术文章第一时间推送。