第 4 章

Kubernetes 开发指南

本章将引入 REST 的概念，详细说明 Kubernetes API，并举例说明如何基于 Jersey 和 Fabric8 框架访问 Kubernetes API，深入分析基于这两个框架访问 Kubernetes API 的优缺点。下面从 REST 开始说起。

4.1 REST 简述

REST（Representational State Transfer）是由 Roy Thomas Fielding 博士在他的论文 *Architectural Styles and the Design of Network-based Software Architectures* 中提出的一个术语。REST 本身只是为分布式超媒体系统设计的一种架构风格，而不是标准。

基于 Web 的架构实际上就是各种规范的集合，这些规范共同组成了 Web 架构，比如 HTTP、客户端服务器模式都是规范。每当我们在原有规范的基础上增加新的规范时，就会形成新的架构。而 REST 正是这样一种架构，它结合了一系列规范，形成了一种新的基于 Web 的架构风格。

传统的 Web 应用大多是 B/S 架构，涉及如下规范。

（1）客户-服务器：这种规范的提出，改善了用户接口跨多个平台的可移植性，并且通过简化服务器组件，改善了系统的可伸缩性。最为关键的是通过分离用户接口和数据存储，使得不同的用户终端共享相同的数据成为可能。

（2）无状态性：无状态性是在客户-服务器约束的基础上添加的又一层规范，它要求通信必须在本质上是无状态的，即从客户端到服务器的每个 request 都必须包含理解该 request 所必需

的所有信息。这个规范改善了系统的可见性（无状态性使得客户端和服务器端不必保存对方的详细信息，服务器只需要处理当前的 request，而不必了解所有 request 的历史）、可靠性（无状态性减少了服务器从局部错误中恢复的任务量）、可伸缩性（无状态性使得服务器端可以很容易地释放资源，因为服务器端不必在多个 request 中保存状态）。同时，这种规范的缺点也是显而易见的，由于不能将状态数据保存在服务器上，因此增加了在一系列 request 中发送重复数据的开销，严重降低了效率。

（3）缓存：为了改善无状态性带来的网络的低效性，我们添加了缓存约束。缓存约束允许隐式或显式地标记一个 response 中的数据，赋予了客户端缓存 response 数据的功能，这样就可以为以后的 request 共用缓存的数据，部分或全部地消除一部分交互，提高了网络效率。但是由于客户端缓存了信息，所以增加了客户端与服务器数据不一致的可能性，从而降低了可靠性。

B/S 架构的优点是部署非常方便，在用户体验方面却不很理想。为了改善这种情况，我们引入了 REST。REST 在原有架构上增加了三个新规范：统一接口、分层系统和按需代码。

（1）统一接口：REST 架构风格的核心特征就是强调组件之间有一个统一的接口，表现为在 REST 世界里，网络上的所有事物都被抽象为资源，REST 通过通用的链接器接口对资源进行操作。这样设计的好处是保证系统提供的服务都是解耦的，极大地简化了系统，从而改善了系统的交互性和可重用性。

（2）分层系统：分层系统规则的加入提高了各种层次之间的独立性，为整个系统的复杂性设置了边界，通过封装遗留的服务，使新的服务器免受遗留客户端的影响，也提高了系统的可伸缩性。

（3）按需代码：REST 允许对客户端功能进行扩展。比如，通过下载并执行 applet 或脚本形式的代码来扩展客户端的功能。但这在改善系统可扩展性的同时降低了可见性，所以它只是 REST 的一个可选约束。

REST 架构是针对 Web 应用而设计的，其目的是为了降低开发的复杂性，提高系统的可伸缩性。REST 提出了如下设计准则。

（1）网络上的所有事物都被抽象为资源（Resource）。

（2）每个资源对应一个唯一的资源标识符（Resource Identifier）。

（3）通过通用的连接器接口（Generic Connector Interface）对资源进行操作。

（4）对资源的各种操作不会改变资源标识符。

（5）所有的操作都是无状态的（Stateless）。

REST 中的资源所指的不是数据，而是数据和表现形式的组合，比如“最新访问的 10 位会员”和“最活跃的 10 位会员”在数据上可能有重叠或者完全相同，而由于它们的表现形式不同，所以被归为不同的资源，这也就是为什么 REST 的全名是 Representational State Transfer。资源标识符就是 URI（Uniform Resource Identifier），不管是图片、Word 还是视频文件，甚至只是一种虚拟的服务，也不管是 xml、txt 还是其他文件格式，全部通过 URI 对资源进行唯一标识。

REST 是基于 HTTP 的，任何对资源的操作行为都通过 HTTP 来实现。以往的 Web 开发大多数用的是 HTTP 中的 GET 和 POST 方法，很少使用其他方法，这实际上是因为对 HTTP 的片面理解造成的。HTTP 不仅仅是一个简单的运载数据的协议，而且是一个具有丰富内涵的网络软件的协议，它不仅能对互联网资源进行唯一定位，还能告诉我们如何对该资源进行操作。HTTP 把对一个资源的操作限制在 4 种方法内：GET、POST、PUT 和 DELETE，这正是对资源 CRUD 操作的实现。由于资源和 URI 是一一对应的，在执行这些操作时 URI 没有变化，和以往的 Web 开发有很大的区别，所以极大地简化了 Web 开发，也使得 URI 可以被设计成更为直观地反映资源的结构。这种 URI 的设计被称作 RESTful 的 URI，为开发人员引入了一种新的思维方式：通过 URL 来设计系统结构。当然了，这种设计方式对于一些特定情况也是不适用的，也就是说不是所有 URI 都适用于 RESTful。

REST 之所以可以提高系统的可伸缩性，就是因为它要求所有操作都是无状态的。由于没有了上下文（Context）的约束，做分布式和集群时就更为简单，也可以让系统更为有效地利用缓冲池（Pool），并且由于服务器端不需要记录客户端的一系列访问，也就减少了服务器端的性能损耗。

Kubernetes API 也符合 RESTful 规范，下面对其进行介绍。

4.2　Kubernetes API 详解

4.2.1　Kubernetes API 概述

Kubernetes API 是集群系统中的重要组成部分，Kubernetes 中各种资源（对象）的数据通过该 API 接口被提交到后端的持久化存储（etcd）中，Kubernetes 集群中的各部件之间通过该 API 接口实现解耦合，同时 Kubernetes 集群中一个重要且便捷的管理工具 kubectl 也是通过访问该 API 接口实现其强大的管理功能的。Kubernetes API 中的资源对象都拥有通用的元数据，资源对象也可能存在嵌套现象，比如在一个 Pod 里面嵌套多个 Container。创建一个 API 对象是指通过 API 调用创建一条有意义的记录，该记录一旦被创建，Kubernetes 将确保对应的资源对象会被

自动创建并托管维护。

在 Kubernetes 系统中，大多数情况下，API 定义和实现都符合标准的 HTTP REST 格式，比如通过标准的 HTTP 动词（POST、PUT、GET、DELETE）来完成对相关资源对象的查询、创建、修改、删除等操作。但同时 Kubernetes 也为某些非标准的 REST 行为实现了附加的 API 接口，例如 Watch 某个资源的变化、进入容器执行某个操作等。另外，某些 API 接口可能违背严格的 REST 模式，因为接口不是返回单一的 JSON 对象，而是返回其他类型的数据，比如 JSON 对象流（Stream）或非结构化的文本日志数据等。

Kubernetes 开发人员认为，任何成功的系统都会经历一个不断成长和不断适应各种变更的过程。因此，他们期望 Kubernetes API 是不断变更和增长的。同时，他们在设计和开发时，有意识地兼容了已存在的客户需求。通常，新的 API 资源（Resource）和新的资源域不希望被频繁地加入系统。资源或域的删除需要一个严格的审核流程。

为了方便查阅 API 接口的详细定义，Kubernetes 使用了 swagger-ui 提供 API 在线查询功能，其官网为 http://kubernetes.io/third_party/swagger-ui/，Kubernetes 开发团队会定期更新、生成 UI 及文档。Swagger UI 是一款 REST API 文档在线自动生成和功能测试软件，关于 Swagger 的内容请访问官网 http://swagger.io。

运行在 Master 节点上的 API Server 进程同时提供了 swagger-ui 的访问地址：http://<master-ip>:<master-port>/swagger-ui/。假设我们的 API Server 安装在 192.168.1.128 服务器上，绑定了 8080 端口，则可以通过访问 http://192.168.1.128:8080/swagger-ui/来查看 API 信息，如图 4.1 所示。

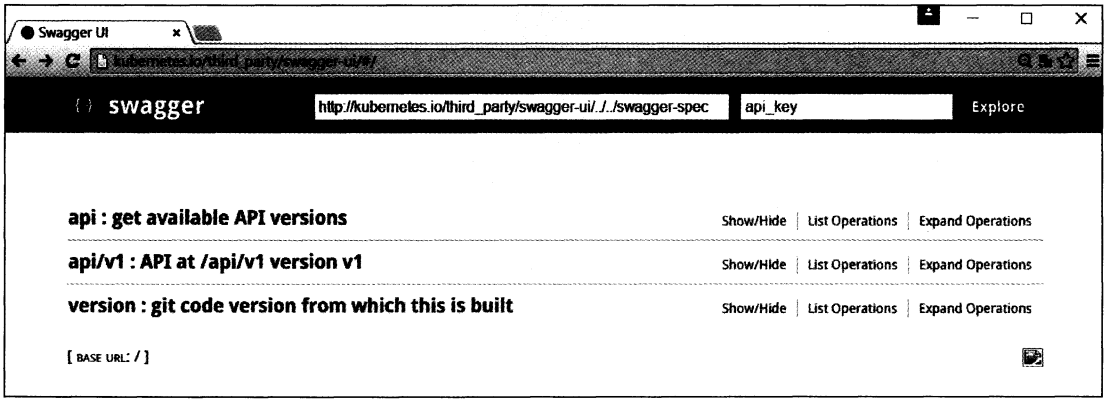

图 4.1　swagger-ui

单击 api/v1 可以查看所有 API 的列表，如图 4.2 所示。

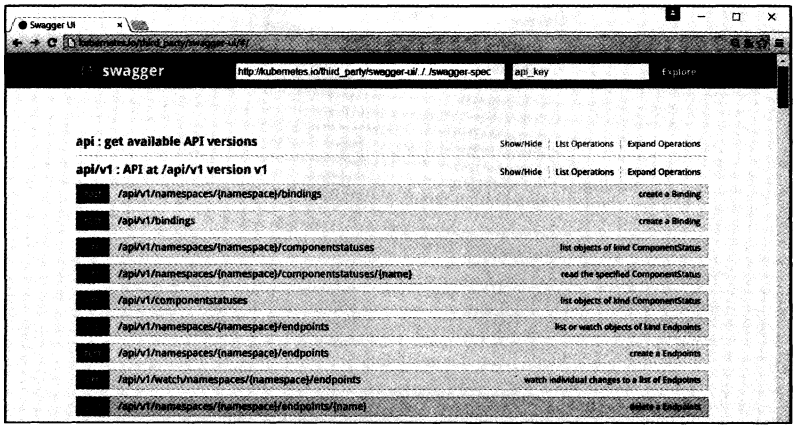

图 4.2　查看 API 列表

以 create a Pod 为例，找到 Rest API 的访问路径为：/api/v1/namespaces/{namespace}/pods，如图 4.3 所示。

图 4.3　Create a Pod API

单击链接展开，即可查看详细的 API 接口说明，如图 4.4 所示。

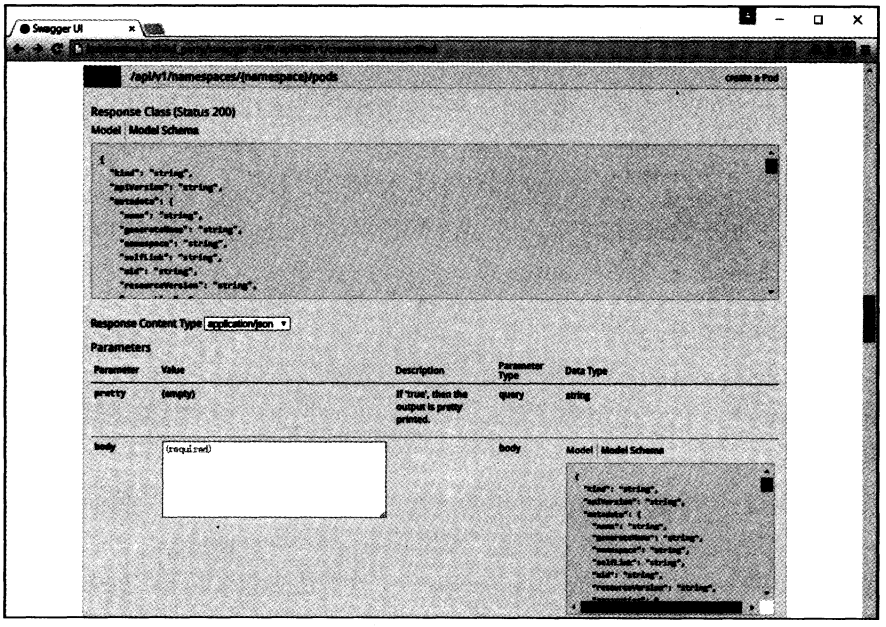

图 4.4　Create a Pod API 详细说明

单击 Model 链接，则可以查看文本格式显示的 API 接口描述，如图 4.5 所示。

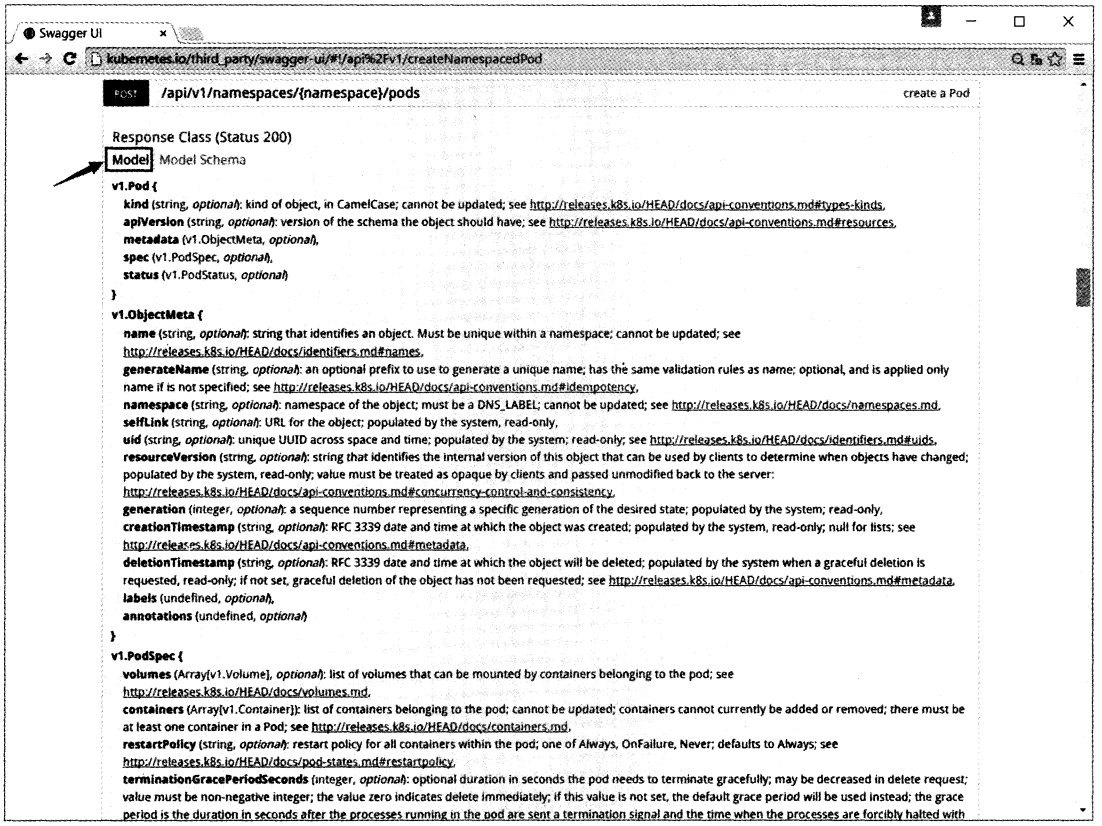

图 4.5　Create a Pod API 文本格式详细说明

我们看到，在 Kubernetes API 中，一个 API 的顶层（Top Level）元素由 kind、apiVersion、metadata、spec 和 status 等几个部分组成，接下来，我们分别对这几个部分进行说明。

kind 表明对象有以下三大类别。

（1）对象（objects）：代表在系统中的一个永久资源（实体），例如 Pod、RC、Service、Namespace 及 Node 等。通过操作这些资源的属性，客户端可以对该对象进行创建、修改、删除和获取操作。

（2）列表（list）：一个或多个资源类别的集合。列表有一个通用元数据的有限集合。所有列表（lists）通过“items”域获得对象数组，例如 PodLists、ServiceLists、NodeLists。大部分定义在系统中的对象都有一个返回所有资源（resource）集合的端点，以及零到多个返回所有资源集合的子集的端点。某些对象有可能是单例对象（singletons），例如当前用户、系统默认用户等，

这些对象没有列表。

（3）简单类别（simple）：该类别包含作用在对象上的特殊行为和非持久实体。该类别限制了使用范围，它有一个通用元数据的有限集合，例如 Binding、Status。

apiVersion 表明 API 的版本号，当前版本默认只支持 v1。

Metadata 是资源对象的元数据定义，是集合类的元素类型，包含一组由不同名称定义的属性。在 Kubernetes 中每个资源对象都必须包含以下 3 种 Metadata。

（1）namespace：对象所属的命名空间，如果不指定，系统则会将对象置于名为“default”的系统命名空间中。

（2）name：对象的名字，在一个命名空间中名字应具备唯一性。

（3）uid：系统为每个对象生成的唯一 ID，符合 RFC 4122 规范的定义。

此外，每种对象还应该包含以下几个重要元数据。

（1）labels：用户可定义的“标签”，键和值都为字符串的 map，是对象进行组织和分类的一种手段，通常用于标签选择器（Label Selector），用来匹配目标对象。

（2）annotations：用户可定义的“注解”，键和值都为字符串的 map，被 Kubernetes 内部进程或者某些外部工具使用，用于存储和获取关于该对象的特定元数据。

（3）resourceVersion：用于识别该资源内部版本号的字符串，在用于 Watch 操作时，可以避免在 GET 操作和下一次 Watch 操作之间造成的信息不一致，客户端可以用它来判断资源是否改变。该值应该被客户端看作不透明，且不做任何修改就返回给服务端。客户端不应该假定版本信息具有跨命名空间、跨不同资源类别、跨不同服务器的含义。

（4）creationTimestamp：系统记录创建对象时的时间戳，符合 RFC 3339 规范。

（5）deletionTimestamp：系统记录删除对象时的时间戳，符合 RFC 3339 规范。

（6）selfLink：通过 API 访问资源自身的 URL，例如一个 Pod 的 link 可能是/api/v1/namespaces/default/pods/frontend-o8bg4。

spec 是集合类的元素类型，用户对需要管理的对象进行详细描述的主体部分都在 spec 里给出，它会被 Kubernetes 持久化到 etcd 中保存，系统通过 spec 的描述来创建或更新对象，以达到用户期望的对象运行状态。spec 的内容既包括用户提供的配置设置、默认值、属性的初始化值，也包括在对象创建过程中由其他相关组件（例如 schedulers、auto-scalers）创建或修改的对象属性，比如 Pod 的 Service IP 地址。如果 spec 被删除，那么该对象将会从系统中被删除。

Status 用于记录对象在系统中的当前状态信息，它也是集合类元素类型，status 在一个自动处理的进程中被持久化，可以在流转的过程中生成。如果观察到一个资源丢失了它的状态

（Status），则该丢失的状态可能被重新构造。以 Pod 为例，Pod 的 status 信息主要包括 conditions、containerStatuses、hostIP、phase、podIP、startTime 等。其中比较重要的两个状态属性如下。

（1）phase：描述对象所处的生命周期阶段，phase 的典型值是“Pending（创建中）”“Running”“Active（正在运行中）”或“Terminated（已终结）”，这几种状态对于不同的对象可能有轻微的差别，此外，关于当前 phase 附加的详细说明可能包含在其他域中。

（2）condition：表示条件，由条件类型和状态值组成，目前仅有一种条件类型 Ready，对应的状态值可以为 True、False 或 Unknown。一个对象可以具备多种 condition，而 condition 的状态值也可能不断发生变化，condition 可能附带一些信息，例如最后的探测时间或最后的转变时间。

4.2.2 API 版本

为了在兼容旧版本的同时不断升级新的 API，Kubernetes 提供了多版本 API 的支持能力，每个版本的 API 通过一个版本号路径前缀进行区分，例如/api/v1beta3。通常情况下，新旧几个不同的 API 版本都能涵盖所有的 Kubernetes 资源对象，在不同的版本之间这些 API 接口存在一些细微差别。Kubernetes 开发团队基于 API 级别选择版本而不是基于资源和域级别，是为了确保 API 能够描述一个清晰的连续的系统资源和行为的视图，能够控制访问的整个过程和控制实验性 API 的访问。

API 及版本发布建议描述了版本升级的当前思路。版本 v1beta1、v1beta2 和 v1beta3 为不建议使用（Deprecated）的版本，请尽快转到 v1 版本。在 2015 年 6 月 4 日，Kubernetes v1 版本 API 正式发布。版本 v1beta1 和 v1beta2 API 在 2015 年 6 月 1 日被删除，版本 v1beta3 API 在 2015 年 7 月 6 日被删除。

4.2.3 API 详细说明

API 资源使用 REST 模式，具体说明如下。

（1）GET /<资源名的复数格式>：获得某一类型的资源列表，例如 GET /pods 返回一个 Pod 资源列表。

（2）POST /<资源名的复数格式>：创建一个资源，该资源来自用户提供的 JSON 对象。

（3）GET /<资源名复数格式>/<名字>：通过给出的名称（Name）获得单个资源，例如 GET /pods/first 返回一个名称为“first”的 Pod。

（4）DELETE /<资源名复数格式>/<名字>：通过给出的名字删除单个资源，在删除选项（DeleteOptions）中可以指定优雅删除（Grace Deletion）的时间（GracePeriodSeconds），该可选项表明了从服务端接收到删除请求到资源被删除的时间间隔（单位为秒）。不同的类别（Kind）

可能为优雅删除时间（Grace Period）申明默认值。用户提交的优雅删除时间将覆盖该默认值，包括值为 0 的优雅删除时间。

（5）PUT /<资源名复数格式>/<名字>：通过给出的资源名和客户端提供的 JSON 对象来更新或创建资源。

（6）PATCH /<资源名复数格式>/<名字>：选择修改资源详细指定的域。

对于 PATCH 操作，目前 Kubernetes API 通过相应的 HTTP 首部“Content-Type”对其进行识别。

目前支持以下三种类型的 PATCH 操作。

（1）JSON Patch, Content-Type: application/json-patch+json。在 RFC6902 的定义中，JSON Patch 是执行在资源对象上的一系列操作，例如 {"op": "add", "path": "/a/b/c", "value": ["foo", "bar"]}。详情请查看 RFC6902 说明，网址为 HTTPs://tools.ietf.org/html/rfc6902。

（2）Merge Patch, Content-Type: application/merge-json-patch+json。在 RFC7386 的定义中，Merge Patch 必须包含对一个资源对象的部分描述，这个资源对象的部分描述就是一个 JSON 对象。该 JSON 对象被提交到服务端，并和服务端的当前对象合并，从而创建一个新的对象。详情请查看 RFC73862 说明，网址为 HTTPs://tools.ietf.org/html/rfc7386。

（3）Strategic Merge Patch, Content-Type: application/strategic-merge-patch+json。Strategic Merge Patch 是一个定制化的 Merge Patch 实现。接下来将详细讲解 Strategic Merge Patch。

在标准的 JSON Merge Patch 中，JSON 对象总是被合并（merge）的，但是资源对象中的列表域总是被替换的。通常这不是用户所希望的。例如，我们通过下列定义创建一个 Pod 资源对象：

```
spec:
  containers:
    - name: nginx
      image: nginx-1.0
```

接着我们希望添加一个容器到这个 Pod 中，代码和上传的 JSON 对象如下所示：

```
PATCH /api/v1/namespaces/default/pods/pod-name
spec:
  containers:
    - name: log-tailer
      image: log-tailer-1.0
```

如果我们使用标准的 Merge Patch，则其中的整个容器列表将被单个的“log-tailer”容器所替换。然而我们的目的是两个容器列表能够合并。

为了解决这个问题，Strategic Merge Patch 通过添加元数据到 API 对象中，并通过这些新元数据来决定哪个列表被合并，哪个列表不被合并。当前这些元数据作为结构标签，对于 API 对象自身来说是合法的。对于客户端来说，这些元数据作为 Swagger annotations 也是合法的。在

上述例子中，向“containers”中添加“patchStrategy”域，且它的值为“merge”，通过添加“patchMergeKey”，它的值为“name”。也就是说，“containers”中的列表将会被合并而不是替换，合并的依据为“name”域的值。

此外，Kubernetes API 添加了资源变动的“观察者”模式的 API 接口。

- GET /watch/<资源名复数格式>：随时间变化，不断接收一连串的 JSON 对象，这些 JSON 对象记录了给定资源类别内所有资源对象的变化情况。
- GET /watch/<资源名复数格式>/<name>：随时间变化，不断接收一连串的 JSON 对象，这些 JSON 对象记录了某个给定资源对象的变化情况。

上述接口改变了返回数据的基本类别，watch 动词返回的是一连串的 JSON 对象，而不是单个的 JSON 对象。并不是所有的对象类别都支持“观察者”模式的 API 接口，在后续的章节中将会说明哪些资源对象支持这种接口。

另外，Kubernetes 还增加了 HTTP Redirect 与 HTTP Proxy 这两种特殊的 API 接口，前者实现资源重定向访问，后者则实现 HTTP 请求的代理。

4.2.4 API 响应说明

API Server 响应用户请求时附带一个状态码，该状态码符合 HTTP 规范。表 4.1 列出了 API Server 可能返回的状态码。

表 4.1 API Server 可能返回的状态码

状 态 码	编 码	描 述
200	OK	表明请求完全成功
201	Created	表明创建类的请求完全成功
204	NoContent	表明请求完全成功，同时 HTTP 响应不包含响应体。 在响应 OPTIONS 方法的 HTTP 请求时返回
307	TemporaryRedirect	表明请求资源的地址被改变，建议客户端使用 Location 首部给出的临时 URL 来定位资源
400	BadRequest	表明请求是非法的，建议客户不要重试，修改该请求
401	Unauthorized	表明请求能够到达服务端，且服务端能够理解用户请求，但是拒绝做更多的事情，因为客户端必须提供认证信息。如果客户端提供了认证信息，则返回该状态码，表明服务端指出所提供的认证信息不合适或非法
403	Forbidden	表明请求能够到达服务端，且服务端能够理解用户请求，但是拒绝做更多的事情，因为该请求被设置成拒绝访问。建议客户不要重试，修改该请求
404	NotFound	表明所请求的资源不存在。建议客户不要重试，修改该请求
405	MethodNotAllowed	表明请求中带有该资源不支持的方法。建议客户不要重试，修改该请求

续表

状 态 码	编　码	描　述
409	Conflict	表明客户端尝试创建的资源已经存在，或者由于冲突请求的更新操作不能被完成
422	UnprocessableEntity	表明由于所提供的作为请求部分的数据非法，创建或修改操作不能被完成
429	TooManyRequests	表明超出了客户端访问频率的限制或者服务端接收到多于它能处理的请求。建议客户端读取相应的 Retry-After 首部，然后等待该首部指出的时间后再重试
500	InternalServerError	表明服务端能被请求访问到，但是不能理解用户的请求；或者服务端内产生非预期中的一个错误，而且该错误无法被认知；或者服务端不能在一个合理的时间内完成处理（这可能由于服务器临时负载过重造成或者由于和其他服务器通信时的一个临时通信故障造成）
503	ServiceUnavailable	表明被请求的服务无效。建议客户不要重试修改该请求
504	ServerTimeout	表明请求在给定的时间内无法完成。客户端仅在为请求指定超时（Timeout）参数时会得到该响应

在调用 API 接口发生错误时，Kubernetes 将会返回一个状态类别（Status Kind）。下面是两种常见的错误场景：

（1）当一个操作不成功时（例如，当服务端返回一个非 2xx HTTP 状态码时）；

（2）当一个 HTTP DELETE 方法调用失败时。

状态对象被编码成 JSON 格式，同时该 JSON 对象被作为请求的响应体。该状态对象包含人和机器使用的域，这些域中包含来自 API 的关于失败原因的详细信息。状态对象中的信息补充了对 HTTP 状态码的说明。例如：

```
    $ curl -v -k -H "Authorization: Bearer WhCDvq4VPpYhrcfmF6ei7V9qlbqTubUc"
HTTPs://10.240.122.184:443/api/v1/namespaces/default/pods/grafana
    > GET /api/v1/namespaces/default/pods/grafana HTTP/1.1
    > User-Agent: curl/7.26.0
    > Host: 10.240.122.184
    > Accept: */*
    > Authorization: Bearer WhCDvq4VPpYhrcfmF6ei7V9qlbqTubUc
    >

    < HTTP/1.1 404 Not Found
    < Content-Type: application/json
    < Date: Wed, 20 May 2015 18:10:42 GMT
    < Content-Length: 232
    <
    {
      "kind": "Status",
      "apiVersion": "v1",
      "metadata": {},
      "status": "Failure",
      "message": "pods \"grafana \"not found",
```

```
  "reason": "NotFound",
  "details": {
    "name": "grafana",
    "kind": "pods"
  },
  "code": 404
}
```

- “status”域包含两个可能的值：Success 和 Failure。
- “message”域包含对错误的可读描述。
- “reason”域包含说明该操作失败原因的可读描述。如果该域的值为空，则表示该域内没有任何说明信息。“reason”域澄清 HTTP 状态码，但没有覆盖该状态码。
- “details”可能包含和“reason”域相关的扩展数据。每个“reason”域可以定义它的扩展的“details”域。该域是可选的，返回数据的格式是不确定的，不同的 reason 类型返回的“details”域的内容不一样。

4.3 使用 Java 程序访问 Kubernetes API

本节介绍如何使用 Java 程序访问 Kubernetes API。在 Kubernetes 的官网上列出了多个访问 Kubernetes API 的开源项目，其中有两个是用 Java 语言开发工具的开源项目，一个是 OSGI，另一个是 Fabric8。在本节所列的两个 Java 开发例子中，一个是基于 Jersey 的，另一个是基于 Fabric8 的。

4.3.1 Jersey

Jersey 是一个 RESTful 请求服务 JAVA 框架。与 Struts 类似，它可以和 Hibernate、Spring 框架整合。通过它不仅方便开发 RESTful Web Service，而且可以将它作为客户端方便地访问 RESTful Web Service 服务端。

如果没有一个好的工具包，则开发一个能够用不同的媒介（Media）类型无缝地暴露你的数据，以及很好地抽象客户、服务端通信的底层通信的 RESTful Web Services，会很不容易。为了能够简化用 Java 开发 RESTful Web Service 及其客户端的流程，业界设计了 JAX-RS API。Jersey RESTful Web Services 框架是一个开源的高质量的框架，它为用 JAVA 语言开发 RESTful Web Service 及其客户端而生，支持 JAX-RS APIs。Jersey 不仅支持 JAX-RS APIs，而且在此基础上扩展了 API 接口，这些扩展更加方便和简化了 RESTful Web Services 及其客户端的开发。

由于 Kuberetes API Server 是 RESTful Web Service，因此此处选用 Jersey 框架开发 RESTful

Web Service 客户端，用来访问 Kubernetes API。在本例中选用的 Jersey 框架的版本为 1.19，所涉及的 Jar 包如图 4.6 所示。

commons-codec-1.2.jar	2015/9/13 11:10	Executable Jar File	30 KB
commons-httpclient-3.1.jar	2015/9/13 11:09	Executable Jar File	298 KB
commons-logging-1.0.4.jar	2015/9/13 11:10	Executable Jar File	38 KB
jackson-core-asl-1.9.2.jar	2015/2/11 5:41	Executable Jar File	223 KB
jackson-jaxrs-1.9.2.jar	2015/2/11 5:41	Executable Jar File	18 KB
jackson-mapper-asl-1.9.2.jar	2015/2/11 5:41	Executable Jar File	748 KB
jackson-xc-1.9.2.jar	2015/2/11 5:41	Executable Jar File	27 KB
jersey-apache-client-1.19.jar	2015/2/11 5:41	Executable Jar File	22 KB
jersey-atom-abdera-1.19.jar	2015/2/11 5:41	Executable Jar File	20 KB
jersey-client-1.19.jar	2015/2/11 5:41	Executable Jar File	131 KB
jersey-core-1.19.jar	2015/2/11 5:41	Executable Jar File	427 KB
jersey-guice-1.19.jar	2015/2/11 5:41	Executable Jar File	16 KB
jersey-json-1.19.jar	2015/2/11 5:41	Executable Jar File	162 KB
jersey-multipart-1.19.jar	2015/2/11 5:41	Executable Jar File	53 KB
jersey-server-1.19.jar	2015/2/11 5:41	Executable Jar File	687 KB
jersey-servlet-1.19.jar	2015/2/11 5:41	Executable Jar File	126 KB
jersey-simple-server-1.19.jar	2015/2/11 5:41	Executable Jar File	12 KB
jersey-spring-1.19.jar	2015/2/11 5:41	Executable Jar File	18 KB
jettison-1.1.jar	2015/2/11 5:41	Executable Jar File	67 KB
jsr311-api-1.1.1.jar	2015/2/11 5:41	Executable Jar File	46 KB
oauth-client-1.19.jar	2015/2/11 5:41	Executable Jar File	15 KB
oauth-server-1.19.jar	2015/2/11 5:41	Executable Jar File	30 KB
oauth-signature-1.19.jar	2015/2/11 5:41	Executable Jar File	24 KB

图 4.6　本例所涉及的 Jar 包

对 Kubernetes API 的访问包含如下三个方面。

（1）指明访问资源的类型。

（2）访问时的一些选项（参数），比如命名空间、对象的名称、过滤方式（标签和域）、子目录、访问的目标是否是代理和是否用 watch 方式访问等。

（3）访问的方法，比如增、删、改、查。

在使用 Jersey 框架访问 Kubernetes API 之前，为这三个方面定义了三个对象。第 1 个定义的对象为 ResourceType，它定义了访问资源的类型；第 2 个定义的对象是 Params，它定义了访问 API 时的一些选项，以及通过这些选项如何生成完整的 URI；第 3 个定义的对象是 RestfulClient，它是一个接口，该接口定义了访问 API 的方法（Method）。

ResourceType 是一个 ENUM 类型的对象，定义了 16 种资源，代码如下：

```
package com.hp.k8s.apiclient.imp;
```

```
    public enum ResourceType {
        NODES("nodes"),
        NAMESPACES("namespaces"),
        SERVICES("services"),
        REPLICATIONCONTROLLERS("replicationcontrollers"),
        PODS("pods"),
        BINDINGS("bindings"),
        ENDPOINTS("endpoints"),
        SERVICEACCOUNTS("serviceaccounts"),
        SECRETS("secrets"),
        EVENTS("events"),
        COMPOMENTSTATUSES("componentstatuses"),
        LIMITRANGES("limitranges"),
        RESOURCEQUOTAS("resourcequotas"),
        PODTEMPLATES("podtemplates"),
        PERSISTENTVOLUMECLAIMS("persistentvolumeclaims"); PERSISTENTVOLUMES
("persistentvolumes");
        private String type;

        private ResourceType(String type) {
            this.type = type;
        }

        public String getType() {
            return type;
        }
    }
```

Params 对象的代码如下：

```
    package com.hp.k8s.apiclient.imp;

    import java.io.UnsupportedEncodingException;
    import java.net.URLEncoder;
    import java.util.List;
    import java.util.Map;

    import org.apache.logging.log4j.LogManager;
    import org.apache.logging.log4j.Logger;

    public class Params {
        private static final Logger LOG = LogManager.getLogger(Params.class.getName());
        private String namespace = null;
        private String name = null;
        private Map<String, String> fields = null;
        private Map<String, String> labels = null;
        private Map<String, String> notLabels = null;
        private Map<String, List<String>> inLabels = null;
```

```
        private Map<String, List<String>> notInLabels = null;
        private String json = null;
        private ResourceType resourceType = null;
        private String subPath = null;
        private boolean isVisitProxy = false;
        private boolean isSetWatcher = false;

        public String buildPath() {
            StringBuilder result = (isVisitProxy ? new StringBuilder("/proxy")
                    : (isSetWatcher ? new StringBuilder("/watch") : new
StringBuilder("")));
            if (null != namespace)
                result.append("/namespaces/").append(namespace);

            result.append("/").append(resourceType.getType());
            if (null != name)
                result.append("/").append(name);
            if(null!=subPath)
                result.append("/").append(subPath);

            if (null != labels && !labels.isEmpty() || null != notLabels && !notLabels.
isEmpty()
                    || null != inLabels && inLabels.size() > 0 || null != notInLabels
&& notInLabels.size() > 0
                    || null != fields && fields.size() > 0) {
                StringBuilder labelSelectorStr = null;
                StringBuilder fieldSelectorStr = null;
                try {
                    labelSelectorStr = builderLabelSelector();
                    fieldSelectorStr = builderFiledSelector();
                } catch (UnsupportedEncodingException e1) {
                    LOG.error(e1);
                }

                if (labelSelectorStr.length() + fieldSelectorStr.length() > 0)
                    result.append("?");
                if (labelSelectorStr.length() > 0) {
                    result.append("labelSelector=") . append(labelSelectorStr.
toString());

                    if (fieldSelectorStr.length() > 0) {
                        result.append(",");
                    }
                }
                if (fieldSelectorStr.length() > 0) {
                    result.append("fieldSelector=").append(fieldSelectorStr.
toString());
```

```
            }

        }

        return result.toString();
    }

    private StringBuilder builderLabelSelector() throws UnsupportedEncoding
Exception {
        StringBuilder result = new StringBuilder();
        if (null != labels) {
            for (String key : labels.keySet()) {
                if (result.length() > 0) {
                    result.append(",");
                }

                result.append(URLEncoder.encode(key + "=" + labels.get(key),
"GBK"));
            }
        }

        if (null != notLabels) {
            for (String key : labels.keySet()) {
                if (result.length() > 0) {
                    result.append(",");
                }

                result.append(URLEncoder.encode(key + "!=" + labels.get(key),
"GBK"));
            }
        }

        if (null != inLabels) {
            for (String key : inLabels.keySet()) {
                if (result.length() > 0) {
                    result.append(URLEncoder.encode(",","GBK"));
                }
                result.append(URLEncoder.encode(key + " in (" + listToString
(inLabels.get(key), ",") + ")", "GBK"));
            }
        }

        if (null != notInLabels) {
            for (String key : inLabels.keySet()) {
                if (result.length() > 0) {
                    result.append(URLEncoder.encode(",","GBK"));
                }
```

```
                result.append(URLEncoder.encode(key + " notin (" + listToString
(inLabels.get(key), ",") + ")", "GBK"));
            }
        }

        LOG.info("label result: " + result);
        return result;
    }

    private StringBuilder builderFiledSelector() throws UnsupportedEncoding
Exception {
        StringBuilder result = new StringBuilder();
        if (null != fields) {
            for (String key : fields.keySet()) {
                if (result.length() > 0) {
                    result.append(",");
                }

                result.append(URLEncoder.encode(key + "=" + fields.get(key),
"GBK"));
            }
        }

        return result;
    }

    private String listToString(List<String> list, String delim) {
        boolean isFirst = true;
        StringBuilder result = new StringBuilder();
        for (String str : list) {
            if (isFirst) {
                result.append(str);
                isFirst = false;
            } else {
                result.append(delim).append(str);
            }
        }

        return result.toString();
    }

    public String getNamespace() {
        return namespace;
    }

    public void setNamespace(String namespace) {
        this.namespace = namespace;
```

```
    }

    public String getName() {
        return name;
    }

    public void setName(String name) {
        this.name = name;
    }

    public Map<String, String> getFields() {
        return fields;
    }

    public void setFields(Map<String, String> fields) {
        this.fields = fields;
    }

    public Map<String, String> getLabels() {
        return labels;
    }

    public void setLabels(Map<String, String> labels) {
        this.labels = labels;
    }

    public String getJson() {
        return json;
    }

    public void setJson(String json) {
        this.json = json;
    }

    public ResourceType getResourceType() {
        return resourceType;
    }

    public void setResourceType(ResourceType resourceType) {
        this.resourceType = resourceType;
    }

    public String getSubPath() {
        return subPath;
    }

    public void setSubPath(String subPath) {
```

```
        this.subPath = subPath;
    }

    public boolean isVisitProxy() {
        return isVisitProxy;
    }

    public void setVisitProxy(boolean isVisitProxy) {
        this.isVisitProxy = isVisitProxy;
    }

    public boolean isSetWatcher() {
        return isSetWatcher;
    }

    public void setSetWatcher(boolean isSetWatcher) {
        this.isSetWatcher = isSetWatcher;
    }

    public Map<String, String> getNotLabels() {
        return notLabels;
    }

    public void setNotLabels(Map<String, String> notLabels) {
        this.notLabels = notLabels;
    }

    public Map<String, List<String>> getInLabels() {
        return inLabels;
    }

    public void setInLabels(Map<String, List<String>> inLabels) {
        this.inLabels = inLabels;
    }

    public Map<String, List<String>> getNotInLabels() {
        return notInLabels;
    }

    public void setNotInLabels(Map<String, List<String>> notInLabels) {
        this.notInLabels = notInLabels;
    }
}
```

Params 对象包含的属性说明如表 4.2 所示。

表 4.2 Params 对象包含的属性列表

属　性	说　明
namespace	String 类型属性，指明资源所在的命名空间，如果没有指定该值，则表明访问所有命名空间下的资源对象
name	String 类型属性，在访问单个资源对象时使用，如果没有指定该值，则表明访问该类资源列表
fields	Map<String, String>类型属性，通过资源对象的域值过滤访问结果
labels	Map<String, String>类型属性，通过指定的标签选择器列表来选择资源对象。选择出的资源对象包含标签列表中所列的标签（即 Map 的 key），且所选资源的标签的 value 和标签列表中的 value 值（即 Map 的 value）相等
notLabels	Map<String, String>类型属性，通过指定的标签选择器列表来选择资源对象。选择出的资源对象包含标签列表中所列的标签（即 Map 的 key），且所选资源的标签的 value 和标签列表中的 value 值（即 Map 的 value）不相等
inLabels	Map<String, List<String>>类型属性，通过指定的标签选择器列表来选择资源对象。Map 对象的 key 值为标签名称，Map 对象的 value 值为该标签可能包含的值
notInLabels	Map<String, List<String>>类型属性，通过指定的标签选择器列表来选择资源对象。Map 对象的 key 值为标签名称，Map 对象的 value 值为列表，表明资源对象包含和 key 值同名的标签，且这些标签的值不在该列表中
json	String 类型属性，在创建或修改资源对象时使用，用于向 API Server 提供资源对象的定义
resourceType	ResourceType 类型属性，用于指明访问资源对象的类型
subPath	String 类型属性，用于指明访问资源的子目录
isVisitProxy	Boolean 类型属性，用于指明是否通过 Proxy 的方式访问资源对象
isSetWatcher	Boolean 类型属性，表明是否通过 Watcher 方式访问资源对象

Params 的 buildPath 方法用于构建访问 URL 的完整路径。

接口对象 RestfulClient 定义了访问 API 接口的所有方法（Method），其代码列表如下：

```
package com.hp.k8s.apiclient;

import com.hp.k8s.apiclient.imp.Params;

public interface RestfulClient {
    public String get(Params params); //获得单个资源对象
    public String list(Params params);  //获得资源对象列表
    public String create(Params params);  //创建资源对象
    public String delete(Params params);  //删除某个资源对象
    public String update(Params params);  //部分更新某个资源对象
    public String updateWithMediaType(Params params,String mediaType);  //通过
mediaType，实现 Merge
    public String replace(Params params);  //替换某个资源对象
    public String options(Params params);
    public String head(Params params);
}
}
```

其中 get 和 list 方法对应 Kubernetes API 的 GET 方法；create 方法对应 API 中的 POST 方法；delete 方法对应 API 中的 DELETE 方法；update 方法对应 API 中的 PATCH 方法；replace 方法对应 API 中的 PUT 方法；options 方法对应 API 中的 OPTIONS 方法；head 方法对应 API 中的