第 6 章
Kubernetes 源码导读

6.1 Kubernetes 源码结构和编译步骤

Kubernetes 的源码现在托管在 GitHub 上，地址为 https://github.com/googlecloudplatform/kubernetes。

编译脚本存放在 build 子目录下，在 Linux 环境（可以是虚拟机）中执行如下命令即可完成代码的编译过程：

```
git clone https://github.com/GoogleCloudPlatform/kubernetes.git
cd kubernetes/build
./release.sh
```

制作 release 的过程其实有不少有意思的事情发生，包括启动 Docker 容器来安装 Go 语言环境、etcd 等，读者若有兴趣则可以查看 release.sh 脚本。另外，如果编译环境是通过 HTTP 代理上网的，则需要设置好 Git 与 Docker 相关的 HTTP 代理参数，同时在文件 kubernetes/build/build-image/Dockerfile 中增加如下 HTTP 代理参数。

- ENV http_proxy=http://username:password@proxyaddr:proxyport。
- ENV https_proxy=http://username:password@proxyaddr:proxyport。

在编译过程中产生的与 Docker 相关的 docker image、dockerfile 及编译好的二进制文件包，则存放在 kubernetes/_output 目录下，这个目录总共有 4 个子目录：dockerized、images、release-stage、release-tars，我们关心后两个目录，其中 release-stage 目录下存放的是支持 linux-amd64 架构的 Server 端的二进制可执行文件（放在 server 子目录下），以及支持不同平台的 Client 端的二进制可执行文件（放在 client 子目录下），release-tars 则存放的是 release-stage 目录下各级子目录的压缩包，与从官方网站下载的完全一样。

考虑到学习和调试 Kubernetes 代码的便利性，我们接下来介绍如何在 Windows 的 LiteIDE 开发环境中完成 Kubernetes 代码的编译和调试。本文假设 Windows 上的 GO 运行时框架和 LiteIDE 开发环境已经建立好，并通过 git clone 命令已经将 https://github.com/GoogleCloudPlatform/kubernetes.git 下载到本地 C:\kubernetes 目录中。通过分析 Kubernetes 的目录结构，我们发现 Kubernetes 的源码都在 pkg 子目录下。接下来建立 k8s 工程目录，目录位置为 C:\project\go\k8s，并在里面建立 src、pkg 两个子目录，然后把 C:\kubernetes\Godeps_workspace\src 全部转移到 C:\project\go\k8s\src 目录下，因为这里是 Kubernetes 源码的所有依赖包，所以如果手动一个一个地下载，则恐怕以国内的网速一天也搞不定。转移完成后，C:\project\go\k8s\src 的目录结构包括如下内容：

```
C:\project\go\k8s\src>dir
2015-07-14  11:56    <DIR>          bitbucket.org
2015-07-14  11:56    <DIR>          code.google.com
2015-07-17  12:30    <DIR>          github.com
2015-07-14  11:56    <DIR>          golang.org
2015-07-14  11:56    <DIR>          google.golang.org
2015-07-14  11:56    <DIR>          gopkg.in
2015-07-14  11:56    <DIR>          speter.net
```

接下来把 C:\kubernetes 的整个目录移动到 C:\project\go\k8s\src\github.com\GoogleCloudPlatform\下，因为 Kubernetes 的源码包的完整名字为“github.com/GoogleCloudPlatform/kubernetes/pkg”。上述工作完成以后，所有的源码都在 C:\project\go\k8s\src 目录下了，我们用 LiteIDE 打开 C:\project\go\k8s，单击菜单“查看”→“管理 Gopath”→添加目录“C:\project\go\k8s”，然后可以进入目录 github.com/ GoogleCloudPlatform/kubernetes/pkg 下，逐一编译每个 package 目录了，如图 6.1 所示。

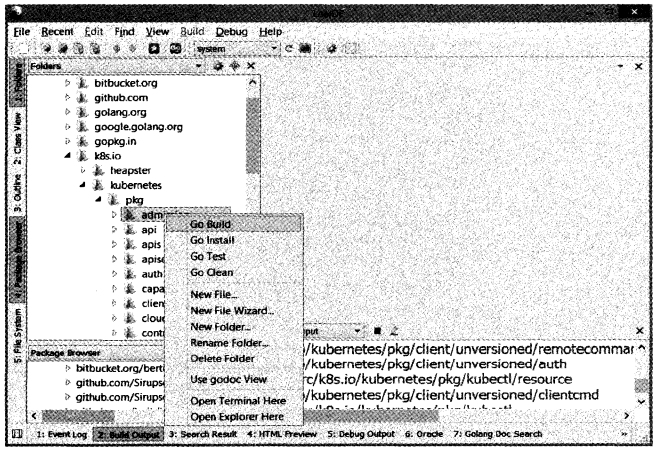

图 6.1　LiteIDE 编译 Kubernetes 的 package

在每个 package 都编译完成以后，我们可以尝试启动 kube-scheduler 进程：在 LiteIDE 里打开 github.com/GoogleCloudPlatform/kubernetes/pkg/plugin/cmd/kube-scheduler/scheduler.go，并且按快捷键 Ctrl+R，你会惊奇地发现这个 Kubernetes 服务器端进程竟然也能在 Windows 下运行起来。以下是 LiteIDE 输出的控制台日志：

```
c:/go/bin/go.exe build -i [C:/project/go/k8s/src/github.com/GoogleCloudPlatform/
kubernetes/plugin/cmd/kube-scheduler]
成功：进程退出代码 0。
C:/project/go/k8s/src/github.com/GoogleCloudPlatform/kubernetes/plugin/cmd/
kube-scheduler/kube-scheduler.exe [C:/project/go/k8s/src/github.com/GoogleCloud
Platform/kubernetes/plugin/cmd/kube-scheduler]
W0717 16:05:26.742413 11344 server.go:83] Neither --kubeconfig nor --master was
specified. Using default API client. This might not work.
E0717 16:05:27.747413 11344 reflector.go:136] Failed to list *api.Node: Get
http://localhost:8080/api/v1/nodes?fieldSelector=spec.unschedulable%3Dfalse: dial
tcp 127.0.0.1:8080: ConnectEx tcp: No connection could be made because the target
machine actively refused it.
E0717 16:05:27.748413 11344 reflector.go:136] Failed to list *api.Pod: Get
http://localhost:8080/api/v1/pods?fieldSelector=spec.nodeName%21%3D: dial tcp
127.0.0.1:8080: ConnectEx tcp: No connection could be made because the target machine
actively refused it.
```

在 Kubernetes 的源码里包括不少单元测试，你可以在 LiteIDE 里运行通过，但有部分测试代码目前在 Windows 上无法通过，毕竟 Kubernetes 是为 Linux 打造的。接下来我们分析下 Kubernetes 源码的整体结构，Kubernetes 的源码总体分为 pkg、cmd、plugin、test 等顶级 package，其中 pkg 为 Kubernetes 的主体代码，cmd 为 Kubernetes 所有后台进程的代码（如 kube-apiserver 进程、kube-controller-manager 进程、kube-proxy 进程、kubelet 进程等），plugin 则包括一些插件及 kuber-scheduler 的代码，test 包是 Kubernetes 的一些测试代码。

从总体来看，Kubernetes 1.0 的当前包结构还是有点乱，开源团队还在继续优化中，可以从源码的 TODO 注释中看出这一点。表 6.1 给出了 Kubernetes 当前主要 package 的源码分析结果。

表 6.1　Kubernetes 主要 package 的源码分析结果

package	模 块 用 途	类 数 量
admission	权限控制框架，采用了责任链模式、插件机制	少
api	Kubernetes 所提供的 Rest API 接口的相关类，例如接口数据结构相关的 MetaData 结构、Volume 结构、Pod 结构、Service 结构等，以及数据格式验证转换工具类等，由于 API 是分版本的，所以这里是每个版本一个子 Package，例如 v1beta、v1 及 latest	中
apiserver	实现了 HTTP Rest 服务的一个基础性框架，用于 Kubernetes 的各种 Rest API 的实现，在 apiserver 包里也实现了 HTTP Proxy，用于转发请求（到其他组件，比如 Minion 节点上）	中
auth	3A 认证模块，包括用户认证、鉴权的相关组件	少

续表

package	模 块 用 途	类 数 量
client	是 Kubernetes 中公用的客户端部分的相关代码，实现协议为 HTTP Rest，用于提供一个具体的操作，例如对 Pod、Service 等的增删改查，这个模块也定义了 kubeletClient，同时为了高效地进行对象查询，此模块也实现了一个带缓存功能的存储接口 Store	多
cloudprovider	定义了云服提供商运行 Kubernetes 所需的接口，包括 TCPLoadBalancer 的获取和创建；获取当前环境中的节点列表（节点是一个云主机）和节点的具体信息；获取 Zone 信息；获取和管理路由的接口等，默认实现了 AWS、GCE、Mesos、OpenStack、RackSpace 等云服务供应商的接口	中
controller	这部分提供了资源控制器的简单框架，用于处理资源的添加、变更、删除等事件的派发和执行，同时实现了 Kubernetes 的 ReplicationController 的具体逻辑	少
kubectl	Kubernetes 的命令行工具 kubectl 的代码模块，包括创建 Pod、服务、Pod 扩容、Pod 滚动升级等各种命令的具体实现代码	多
kubelet	Kubernetes 的 kubelet 的代码模块，是 Kubernetes 的核心模块之一，定义了 Pod 容器的接口，提供了 Docker 与 Rkt 两种容器实现类，完成了容器及 Pod 的创建，以及容器状态的监控、销毁、垃圾回收等功能	多
master	Kubernetes 的 Master 节点代码模块，创建 NodeRegistry、PodRegistry、ServiceRegistry、EndpointRegistry 等组件，并且启动 Kubernetes 自身的相关服务，服务的 ClusterIP 地址分配及服务的 NodePort 端口分配，也是在这里完成的	少
proxy	Kubernetes 的服务代理和负载均衡相关功能的模块代码，目前实现了 round-robin 的负载均衡算法	少
registry	Kubernetes 的 NodeRegistry、PodeRegistry、ReplicationControllerRegistry、ServiceRegistry、EndpointRegistry、PersistentVolumeRegisty 等注册表服务的接口及对应 Rest 服务的相关代码	多
runtime	为了让多个 API 版本共存，需要采用一些设计来完成不同 API 版本的数据结构的转换，API 中数据对象的 Encode/Decode 逻辑也最好集中化，Runtime 包就是为了这个目的而设计的	少
volume	实现了 Kubernetes 的各种 Volume 类型，分别对应亚马逊 ESB 存储、谷歌 GCE 的存储、Linux Host 目录存储、GlusterFS 存储、iSCSI 存储、NFS 存储、RBD 存储等，volume 包同时实现了 Kubernetes 容器的 Volume 卷的挂载、卸载功能	多
cmd	包括了 Kubernetes 所有后台进程的代码（如 kube-apiserver 进程、kube-controller-manager 进程、kube-proxy 进程、kubelet 进程等），而这些进程具体的业务逻辑代码则都在 pkg 中实现了	
plugin	子包 cmd/kuber-scheduler 实现了 Schedule Server 的框架，用于执行具体的 Scheduler 的调度，pkg/admission 子包则实现了 Admission 权限框架的一些默认实现类，例如 alwaysAdmit、alwaysDeny 等；pkg/auth 子包实现了权限认证框架（auth 包的）里定义的认证接口类，例如 HTTP BasicAuth、X509 证书认证；pkg/scheduler 子包则定义了一些具体的 Pod 调度器（Scheduler）	中

6.2 kube-apiserver 进程源码分析

Kubernetes API Server 是由 kube-apiserver 进程实现的，它运行在 Kubernetes 的管理节点——Master 上并对外提供 Kubernetes Restful API 服务，它提供的主要是与集群管理相关的 API 服务，例如校验 pod、service、replication controller 的配置并存储到后端的 etcd Server 上。下面我们分别对其启动过程、关键代码分析及设计总结等进行深入讲解。

6.2.1 进程启动过程

kube-apiserver 进程的入口类源码位置如下：

```
github/com/GoogleCloudPlatform/kubernetes/cmd/kube-apiserver/apiserver.go
```

入口 main()函数的逻辑如下：

```
func main() {
    runtime.GOMAXPROCS(runtime.NumCPU())
    rand.Seed(time.Now().UTC().UnixNano())

    s := app.NewAPIServer()
    s.AddFlags(pflag.CommandLine)

    util.InitFlags()
    util.InitLogs()
    defer util.FlushLogs()

    verflag.PrintAndExitIfRequested()

    if err := s.Run(pflag.CommandLine.Args()); err != nil {
        fmt.Fprintf(os.Stderr, "%v\n", err)
        os.Exit(1)
    }
}
```

上述代码的核心为下面三行，创建一个 APIServer 结构体并将命令行启动参数传入，最后启动监听：

```
s := app.NewAPIServer()
s.AddFlags(pflag.CommandLine)
s.Run(pflag.CommandLine.Args())
```

我们先来看看都有哪些常用的命令行参数被传递给了 APIServer 对象，下面是运行在 Master

节点的 kube-apiserver 进程的命令行信息：

```
/usr/bin/kube-apiserver --logtostderr=true --etcd_servers=http://127.0.0.1:
4001 --address=0.0.0.0 --port=8080 --kubelet_port=10250 --allow_privileged=false
--service-cluster-ip-range=10.254.0.0/16
```

可以看到关键的几个参数有 etcd_servers 的地址、APIServer 绑定和监听的本地地址、kubelet 的运行端口及 Kubernetes 服务的 clusterIP 地址。

下面是 app.NewAPIServer()的代码，我们看到这里的控制还是很全面的，包括安全控制（CertDirectory、HTTPS 默认启动）、权限控制（AuthorizationMode、AdmissionControl）、服务限流控制（APIRate、APIBurst）等，这些逻辑说明了 APIServer 是按照企业级平台的标准所设计和实现的。

```
func NewAPIServer() *APIServer {
    s := APIServer{
        InsecurePort:          8080,
        InsecureBindAddress:    util.IP(net.ParseIP("127.0.0.1")),
        BindAddress:           util.IP(net.ParseIP("0.0.0.0")),
        SecurePort:            6443,
        APIRate:               10.0,
        APIBurst:              200,
        APIPrefix:             "/api",
        EventTTL:              1 * time.Hour,
        AuthorizationMode:      "AlwaysAllow",
        AdmissionControl:       "AlwaysAdmit",
        EtcdPathPrefix:         master.DefaultEtcdPathPrefix,
        EnableLogsSupport:      true,
        MasterServiceNamespace: api.NamespaceDefault,
        ClusterName:           "kubernetes",
        CertDirectory:         "/var/run/kubernetes",

        RuntimeConfig: make(util.ConfigurationMap),
        KubeletConfig: client.KubeletConfig{
            Port:       ports.KubeletPort,
            EnableHttps: true,
            HTTPTimeout: time.Duration(5) * time.Second,
        },
    }

    return &s
}
```

创建了 APIServer 结构体实例后，apiserver.go 将此实例传入子包 app/server.go 的 func(s *APIServer) Run(_ []string)方法里，最终绑定本地端口并创建一个 HTTP Server 与一个 HTTPS Server，从而完成整个进程的启动过程。

Run 方法的代码有很多，这里就不再列出源码，对该方法的源码解读如下。

（1）调用 verifyClusterIPFlags 方法，验证 ClusterIP 参数是否已设置及是否有效。

（2）验证 etcd-servers 的参数是否已设置。

（3）如果初始化 CloudProvider，且没有 CloudProvider 的参数，则日志告警并继续。

（4）根据 KubeletConfig 的配置参数，调用 pkg/Client/kubeclient.go 中的方法 NewKubeletClient() 创建一个 kubelet Client 对象，这其实是一个 HTTPKubeletClient 实例，目前只用于 kubelet 的健康检查（KubeletHealthChecker）。

（5）判断哪些 API Version 需要关闭，目前在 1.0 代码中默认关闭了 v1beta3 的 API 版本。

（6）创建一个 Kubernetes 的 RestClient 对象，具体的代码在 pkg/client/helper.go 的 TransportFor() 方法里完成，通过它完成 Pod、Replication Controller 及 Kubernetes Service 等对象的 CRUD 操作。

（7）创建用于访问 etcd Server 的客户端，具体代码在 newEtcd()方法里实现，从代码调用中可以看出，Kubernetes 采用的是 github.com/coreos/go-etcd/client.go 这个客户端实现。

（8）建立鉴权（Authenticator）、授权（Authorizer）、服务许可框架和插件（AdmissionControl）的相关代码逻辑。

（9）获取和设置 APIServer 的 ExternalHost 的名称，如果没有提供 ExternalHost 参数，且 Kubernetes 运行在谷歌的 GCE 云平台上，则尝试通过 CloudProvider 接口获取本机节点的外部 IP 地址。

（10）如果运行在云平台中，则安装本机的 SSH Key 到 Kubernetes 集群中的所有虚拟机上。

（11）用 APIServer 的数据及上述过程中创建的一些对象（KubeletClient、etcdClient、authenticator、admissionController 等）作为参数，构造 Kubernetes Master 的 Config 结构（pkg/master/master.go），以此生成一个 Master 实例，具体代码在 master.go 中的 New（c *Config）方法里。

（12）用上述创建的 Master 实例，分别创建 HTTP Server 及安全的 HTTPS Server 来开始监听客户端的请求，至此整个进程启动完毕。

6.2.2 关键代码分析

在 6.2.1 节里对 kube-apiserver 进程的启动过程进行了详细分析，我们发现 Kubernetes API Service 的关键代码就隐藏在 pkg/master/master.go 里，APIServer 这个结构体只不过是一个参数传递通道而已，它的数据最终传给了 pkg/master/master.go 里的 Master 结构体，下面是它的完整定义：

```
// Master contains state for a Kubernetes cluster master/api server.
```

```
    type Master struct {
        // "Inputs", Copied from Config
        serviceClusterIPRange *net.IPNet
        serviceNodePortRange  util.PortRange
        cacheTimeout          time.Duration
        minRequestTimeout     time.Duration

        mux                 apiserver.Mux
        muxHelper           *apiserver.MuxHelper
        handlerContainer    *restful.Container
        rootWebService      *restful.WebService
        enableCoreControllers bool
        enableLogsSupport    bool
        enableUISupport      bool
        enableSwaggerSupport  bool
        enableProfiling      bool
        apiPrefix           string
        corsAllowedOriginList util.StringList
        authenticator        authenticator.Request
        authorizer           authorizer.Authorizer
        admissionControl     admission.Interface
        masterCount          int
        v1beta3              bool
        v1                  bool
        requestContextMapper  api.RequestContextMapper

        // External host is the name that should be used in external (public internet)
URLs for this master
        externalHost string
        // clusterIP is the IP address of the master within the cluster.
        clusterIP          net.IP
        publicReadWritePort  int
        serviceReadWriteIP   net.IP
        serviceReadWritePort int
        masterServices      *util.Runner

        // storage contains the RESTful endpoints exposed by this master
        storage map[string]rest.Storage
        // registries are internal client APIs for accessing the storage layer
        // TODO: define the internal typed interface in a way that clients can
        // also be replaced
        nodeRegistry           minion.Registry
        namespaceRegistry       namespace.Registry
        serviceRegistry         service.Registry
        endpointRegistry        endpoint.Registry
```

```
    serviceClusterIPAllocator service.RangeRegistry
serviceNodePortAllocator  service.RangeRegistry
// "Outputs"
    Handler        http.Handler
    InsecureHandler http.Handler

    // Used for secure proxy
    dialer        apiserver.ProxyDialerFunc
    tunnels       *util.SSHTunnelList
    tunnelsLock   sync.Mutex
installSSHKey  InstallSSHKey
   lastSync      int64 // Seconds since Epoch
    lastSyncMetric prometheus.GaugeFunc
    clock         util.Clock
}
```

在这段代码里，除了之前我们熟悉的那些变量，又多了几个陌生的重要变量，接下来我们逐一对其进行分析讲解。

首先是类型为 apiserver.Mux（来自文件 pkg/apiserver/apiserver.go）的 mux 变量，下面是对它的定义：

```
// mux is an object that can register http handlers.
type Mux interface {
  Handle(pattern string, handler http.Handler)
  HandleFunc(pattern string, handler func(http.ResponseWriter, *http.Request))
}
```

如果你熟悉 Socket 编程，特别使用过或者研究过 HTTP Rest 的一些框架，那么对于这个 Mux 接口就再熟悉不过了，它是一个 HTTP 的多分器（Multiplexer），其实它也是 Golang HTTP 基础包里的 http.ServeMux 的一个接口子集，用于派发（Dispatch）某个 Request 路径（这里用 pattern 变量表示）到对应的 http.Handler 进行处理。实际上在 master.go 代码中是生成一个 http.ServeMux 对象并赋值给 apiserver.Mux 变量，在代码中还有强制类型转换的语句。从上述分析来看，apiserver.Mux 的引入是设计的一个败笔，并没有增加什么价值，反而增加了理解代码的难度。此外，为了更好地实现 Rest 服务，Kubernetes 在这里引入了一个第三方的 REST 框架：github.com/emicklei/go-restful。

go-restful 在 GitHub 上有 36 个贡献者，采用了“路由”映射的设计思想，并且在 API 设计中使用了流行的 Fluent Style 风格，使用起来酣畅淋漓，也难怪 Kubernetes 选择了它。下面是 go-restful 的优良特性。

- Ruby on Rails 风格的 Rest 路由映射，例如/people/{person_id}/groups/{group_id}。
- 大大简化了 Rest API 的开发工作。

- 底层实现采用 Golang 的 HTTP 协议栈，几乎没有限制。
- 拥有完整的单元包代码，很容易开发一个可测试的 Rest API。
- Google AppEngine ready。

go-restful 框架中的核心对象如下。

- restful.Container：代表一个 HTTP Rest 服务器，包括一组 restful.WebService 对象和一个 http.ServeMux 对象，使用 RouteSelector 进行请求派发。
- restful.WebService：表示一个 Rest 服务，由多个 Rest 路由（restful.Route）组成，这一组 Rest 路由共享同一个 Root Path。
- restful.Route：表示一个 Rest 路由，Rest 路由主要由 Rest Path、HTTP Method、输入输出类型（HTML/JSON）及对应的回调函数 restful. RouteFunction 组成。
- restful.RouteFunction：一个用于处理具体的 REST 调用的函数接口定义，具体定义为 type RouteFunction func(*Request, *Response)。

Master 结构体里包含了对 restful.Container 与 restful.WebService 这两个 go-restful 核心对象的引用，在接下来的 Master 对象的构造方法中（对应代码为 master.go 的 func New(c *Config) *Master）被初始化。那么，问题又来了，Kubernetes 的这么一堆 Rest API 又是在哪里定义的，是如何被绑定到 restful.Route 里的呢？

要理解这个问题，我们要首先弄清楚 Master 结构体中的变量：

```
storage map[string]rest.Storage
```

storage 变量是一个 Map，Key 为 Rest API 的 path，Value 为 rest.Storage 接口，此接口是一个通用的符合 Restful 要求的资源存储服务接口，每个服务接口负责处理一类（Kind）Kubernetes Rest API 中的数据对象——资源数据 ，只有一个接口方法：New()，New()方法返回该 Storage 服务所能识别和管理的某种具体的资源数据的一个空实例。

```
type Storage interface {
    New() runtime.Object
}
```

在运行期间，Kubernetes API Runtime 运行时框架会把 New()方法返回的空对象的指针传入 Codec.DecodeInto([]byte, runtime.Object)方法中，从而完成 HTTP Rest 请求中的 Byte 数组反序列化逻辑。Kubernetes API Server 中所有对外提供服务的 Restful 资源都实现了此接口，这些资源包括 pods、bindings、podTemplates、replicationControllers、services 等，完整的列表就在 master.go 的 func (m *Master) init(c *Config)中，下面是相关代码片段（截取部分代码）。

```
m.storage = map[string]rest.Storage{
        "pods":            podStorage.Pod,
```

```
            "pods/status":      podStorage.Status,
            "pods/log":         podStorage.Log,
            "pods/exec":        podStorage.Exec,
            "pods/portforward": podStorage.PortForward,
            "pods/proxy":       podStorage.Proxy,
            "pods/binding":     podStorage.Binding,
            "bindings":         podStorage.Binding,

            "podTemplates": podTemplateStorage,

            "replicationControllers": controllerStorage,
            "services":               service.NewStorage(m.serviceRegistry,
m.nodeRegistry, m.endpointRegistry, serviceClusterIPAllocator, serviceNodePort
Allocator, c.ClusterName),
            "endpoints":              endpointsStorage,
            "minions":                nodeStorage,
```

看到上面这段代码，你在潜意识里已经明白，这其实就是似曾相识的 Kubernetes Rest API 列表，storage 这个 Map 的 Key 就是 Rest API 的访问路径，Value 却不是之前说好的 restful.Route。聪明的你一定想到了答案：必然存在一个“转换适配”的方法来实现上述转换！这段不难理解但源码超长的方法就在 pkg/apiserver/api_installer.go 的下述方法里：

```
    func (a *APIInstaller) registerResourceHandlers(path string, storage rest.
Storage, ws *restful.WebService, proxyHandler http.Handler)
```

上述方法把一个 path 对应的 rest.Storage 转换成一系列的 restful.Route 并添加到指针 restful.WebService 中。这个函数的代码之所以很长，是因为有各种情况要考虑，比如 pods/portforward 这种路径要处理 child，还要判断每种的 Storage 资源类型所支持的操作类型；比如是否支持 create、delete、update 及是否支持 list、watch、patcher 操作等，对各种情况都考虑以后，这个函数的代码量已接近 500 行！估计 Kubernetes 这段代码的作者也不大好意思，于是外面封装了简单函数：func(a *APIInstaller)Install，内部循环调用 registerResourceHandlers，返回最终的 restful.WebService 对象，此方法的主要代码如下：

```
    // Installs handlers for API resources.
    func (a *APIInstaller) Install() (ws *restful.WebService, errors []error) {
        // Register the paths in a deterministic (sorted) order to get a deterministic
swagger spec.
        paths := make([]string, len(a.group.Storage))
        var i int = 0
        for path := range a.group.Storage {
            paths[i] = path
            i++
        }
        sort.Strings(paths)
        for _, path := range paths {
```

```
            if err := a.registerResourceHandlers(path, a.group.Storage[path], ws,
proxyHandler); err != nil {
                errors = append(errors, err)
            }
        }
        return ws, errors
    }
```

为了区分 API 的版本，在 apiserver.go 里定义了一个结构体：APIGroupVersion。以下是其代码：

```
    type APIGroupVersion struct {
        Storage map[string]rest.Storage
        Root    string
        Version string
        // ServerVersion controls the Kubernetes APIVersion used for common objects
in the apiserver
        // schema like api.Status, api.DeleteOptions, and api.ListOptions. Other
implementors may
        // define a version "v1beta1" but want to use the Kubernetes "v1beta3" internal
objects. If
        // empty, defaults to Version.
        ServerVersion string

        Mapper meta.RESTMapper

        Codec     runtime.Codec
        Typer     runtime.ObjectTyper
        Creater   runtime.ObjectCreater
        Convertor runtime.ObjectConvertor
        Linker    runtime.SelfLinker

        Admit   admission.Interface
        Context api.RequestContextMapper

        ProxyDialerFn     ProxyDialerFunc
        MinRequestTimeout time.Duration
    }
```

我们注意到 APIGroupVersion 是与 rest.Storage Map 捆绑的，并且绑定了相应版本的 Codec、Convertor 用于版本转换，这样就很容易理解 Kubernetes 是怎样区分多版本 API 的 Rest 服务的。以下是过程详解。

首先，在 APIGroupVersion 的 InstallREST(container *restful.Container)方法里，用 Version 变量来构造一个 Rest API Path 前缀并赋值给 APIINstaller 的 prefix 变量，并调用它的 Install()方法完成 Rest API 的转换，代码如下：

```
    func (g *APIGroupVersion) InstallREST(container *restful.Container) error {
```

```
    info := &APIRequestInfoResolver{util.NewStringSet(strings.TrimPrefix(g.Root,
"/")), g.Mapper}
    prefix := path.Join(g.Root, g.Version)
        installer := &APIInstaller{
            group:           g,
            info:            info,
            prefix:          prefix,
            minRequestTimeout: g.MinRequestTimeout,
            proxyDialerFn:     g.ProxyDialerFn,
        }
        ws, registrationErrors := installer.Install()
        container.Add(ws)
```

接着，在 APIInstaller 的 Install()方法里用 prefix（API 版本）前缀生成 WebService 的相对根路径：

```
    func (a *APIInstaller) newWebService() *restful.WebService {
    ws := new(restful.WebService)
    ws.Path(a.prefix)
    ws.Doc("API at"+ a.prefix +"version"+ a.group.Version)
        // TODO: change to restful.MIME_JSON when we set content type in client
        ws.Consumes("*/*")
        ws.Produces(restful.MIME_JSON)
        ws.ApiVersion(a.group.Version)

        return ws
    }
```

最后，在 Kubernetes 的 Master 初始化方法 func (m *Master) init (c *Config)里生成不同的 APIGroupVersion 对象，并调用 InstallRest()方法，完成最终的多版本 API 的 Rest 服务装配流程：

```
        if m.v1beta3 {
            if err := m.api_v1beta3().InstallREST(m.handlerContainer); err != nil {
                glog.Fatalf("Unable to setup API v1beta3: %v", err)
            }
            apiVersions = append(apiVersions, "v1beta3")
        }
        if m.v1 {
            if err := m.api_v1().InstallREST(m.handlerContainer); err != nil {
                glog.Fatalf("Unable to setup API v1: %v", err)
            }
            apiVersions = append(apiVersions, "v1")
    }
```

至此，Rest API 的多版本问题还有最后一个需要澄清，即在不同的版本中接口的输入输出

参数的格式是有差别的，Kubernetes 是怎么处理这个问题的？

要弄明白这一点，我们首先要研究 Kubernetes API 里的数据对象的序列化、反序列化的实现机制。为了同时解决数据对象的序列化、反序列化与多版本数据对象的兼容和转换问题，Kubernetes 设计了一套复杂的机制，首先，它设计了 conversion.Scheme 这个结构体（pkg/conversion/schema.go 里），以下是对它的定义：

```
    // Scheme defines an entire encoding and decoding scheme.
    type Scheme struct {
        // versionMap allows one to figure out the go type of an object        //with
the given version and name.
        versionMap map[string]map[string]reflect.Type
        // typeToVersion allows one to figure out the version for a given //go object
The reflect.Type we index by should *not* be a pointer. If the same type
        // is registered for multiple versions, the last one wins.
        typeToVersion map[reflect.Type]string
        // typeToKind allows one to figure out the desired "kind" field //for a given
go object. Requirements and caveats are the same as typeToVersion.
        typeToKind map[reflect.Type][]string
        // converter stores all registered conversion functions. It also //has default
coverting behavior.
        converter *Converter
        // cloner stores all registered copy functions. It also has default
        // deep copy behavior.
        cloner *Cloner
        // Indent will cause the JSON output from Encode to be indented, iff it is true.
        Indent bool
        // InternalVersion is the default internal version. It is recommended that
        // you use "" for the internal version.
        InternalVersion string
        // MetaInsertionFactory is used to create an object to store and retrieve
        // the version and kind information for all objects. The default // uses
the   keys "apiVersion" and "kind" respectively.
        MetaFactory MetaFactory
    }
```

在上述代码中可以看到，typeToVersion 与 versionMap 属性是为了解决数据对象的序列化与反序列化问题，converter 属性则负责不同版本的数据对象转换问题，Kubernetes 这个设计思路简单方便地解决了多版本的序列化和数据转换问题，不得不赞！下面是 conversion.Scheme 里序列化、反序列化的核心方法 NewObject()的代码：通过查找 versionMap 里匹配的注册类型，以反射方式生成一个空的数据对象：

```
    func (s *Scheme) NewObject(versionName, kind string) (interface{}, error) {
        if types, ok := s.versionMap[versionName]; ok {
            if t, ok := types[kind]; ok {
                return reflect.New(t).Interface(), nil
```

```
            }
            return nil, &notRegisteredErr{kind: kind, version: versionName}
        }
        return nil, &notRegisteredErr{kind: kind, version: versionName}
    }
```

而 pkg/conversion/encode.go 与 decode.go 则在 conversion.Scheme 提供的基础功能之上，完成了最终的序列化、反序列化功能。下面是 encode.go 里的主方法 EncodeToVersion(..)的关键代码片段：

```
//确定要转换的源对象的版本号和类别
objVersion, objKind, err := s.ObjectVersionAndKind(obj) 象
//生成目标版本的空对象
objOut, err := s.NewObject(destVersion, objKind)
//生成转换过程中所需的 Metadata 信息
flags, meta := s.generateConvertMeta(objVersion, destVersion, obj)
//调用 converter 的方法将源对象的数据填充到目标对象 objOut
err = s.converter.Convert(obj, objOut, flags, meta)
//用 JSON 将目标对象转换成 byte[]数组，完成序列化过程
data, err = json.Marshal(obj)
```

再进一步，Kubernetes 在 conversion.Scheme 的基础上又做了一个封装工具类 runtime.Scheme，可以看作前者的代理类，主要增加了 fieldLabelConversionFuncs 这个 Map 属性，用于解决数据对象的属性名称的兼容性转换和校验，比如将需要兼容 Pod 的 spec.host 属性改为 spec.nodeName 的情况。

注意到 conversion.Scheme 只是实现了一个序列化与类型转换的框架 API，提供了注册资源数据类型与转换函数的功能，那么具体的资源数据对象类型、转换函数又是在哪个包里实现的呢？答案是 pkg/api。Kubernetes 为不同的 API 版本提供了独立的数据类型和相关的转换函数并按照版本号命名 Package，如 pkg/api/v1、pkg/api/v1beta3 等，而当前默认版本（内部版本）则存在于 pkg/api 目录下。

以 pkg/api/v1 为例，在每个目录里都包括如下关键源码：

- types.go 定义了 Rest API 接口里所涉及的所有数据类型，v1 版本有 2000 行代码；
- 在 conversion.go 与 conversion_generated.go 里定义了 conversion.Scheme 所需的从内部版本到 v1 版本的类型转换函数，其中 conversion_generated.go 中的代码有 5000 行之多，当然这是通过工具自动生成的代码；
- register.go 负责将 types.go 里定义的数据类型与 conversion.go 里定义的数据转换函数注册到 runtime.Schema 里。

pkg/api 里的 register.go 初始化生成并持有一个全局的 runtime.Scheme 对象，并将当前默认版本的数据类型（pkg/api/types.go）注册进去，相关代码如下：

```
var Scheme = runtime.NewScheme()
func init() {
    Scheme.AddKnownTypes("",
        &Pod{},
        &PodList{},
        &PodStatusResult{},
        &PodTemplate{},
        &PodTemplateList{},
        &ReplicationControllerList{},
//此次省略 30 多个数据类型
        &ServiceList{},
        &Service{},
        &NodeList{},
        &Node{},
//省略
```

而 pkg/api/v1/register.go 与 v1beta3 下的 register.go 在初始化过程中分别把与版本相关的数据类型和转换函数注册到全局的 runtime.Scheme 中：

```
func init() {
    // Check if v1 is in the list of supported API versions.
    if !registered.IsRegisteredAPIVersion("v1") {
        return
    }

    // Register the API.
    addKnownTypes()
    addConversionFuncs()
    addDefaultingFuncs()
}
```

这样一来，其他地方都可以通过 runtime.Scheme 这个全局变量来完成 Kubernetes API 中的数据对象的序列化和反序列化逻辑了，比如 Kubernetes API Client 包就大量使用了它，下面是 pkg/client/pods.go 里 Pod 删除的 Delete()方法的代码：

```
// Delete takes the name of the pod, and returns an error if one occurs
func (c *pods) Delete(name string, options *api.DeleteOptions) error {
    // TODO: to make this reusable in other client libraries
    if options == nil {
        return c.r.Delete().Namespace(c.ns).Resource("pods").Name(name).
Do().Error()
    }
    body, err := api.Scheme.EncodeToVersion(options, c.r.APIVersion())
    if err != nil {
        return err
    }
    return c.r.Delete().Namespace(c.ns).Resource("pods").Name(name).
Body(body).Do().Error()
```

```
}
```

清楚了 Kubernetes Rest API 中的数据对象的序列化机制及多版本的实现原理之后，我们接着分析下面这个重要流程的实现细节。

Kubernetes 中实现了 rest.Storage 接口的服务在转换成 restful.RouteFunction 以后，是怎样处理一个 Rest 请求并最终完成基于后端存储服务 etcd 上的具体操作过程的？

首先，Kubernetes 设计了一个名为“注册表”的 Package（pkg/registry），这个 Package 按照 rest.Storage 服务所管理的资源数据的类型而划分为不同的子包，每个子包都由相同命名的一组 Golang 代码来完成具体的 Rest 接口的实现逻辑。

下面我们以 Pod 的 Rest 服务实现为例，其与“注册表”相关的代码位于 pkg/registry/pod 中，在 registry.go 里定义了 Pod 注册表服务的接口：

```
type Registry interface {
    // ListPods obtains a list of pods having labels which match selector.
    ListPods(ctx api.Context, label labels.Selector) (*api.PodList, error)
    // Watch for new/changed/deleted pods
    WatchPods(ctx api.Context, label labels.Selector, field fields.Selector,
resourceVersion string) (watch.Interface, error)
    // Get a specific pod
    GetPod(ctx api.Context, podID string) (*api.Pod, error)
    // Create a pod based on a specification.
    CreatePod(ctx api.Context, pod *api.Pod) error
    // Update an existing pod
    UpdatePod(ctx api.Context, pod *api.Pod) error
    // Delete an existing pod
DeletePod(ctx api.Context, podID string) error
}
```

我们看到这个 Pod 注册表服务是针对 Pod 的 CRUD 的操作接口的一个定义，在入口参数中除了调用的上下文环境 api.Context，就是我们之前分析过的 pkg/api 包中的 Pod 这个资源数据对象。为了实现强类型的方法调用，在 registry.go 里定义了一个名为 storage 的结构体，storage 实现 Registry 接口，可以看作一种代理设计模式，因为具体的操作都是通过内部 rest.StandardStorage 来实现的。下面是截取的 registry.go 中的 create、update、delete 的源码：

```
func (s *storage) CreatePod(ctx api.Context, pod *api.Pod) error {
    _, err := s.Create(ctx, pod)
    return err
}

func (s *storage) UpdatePod(ctx api.Context, pod *api.Pod) error {
    _, _, err := s.Update(ctx, pod)
    return err
}
```

```
func (s *storage) DeletePod(ctx api.Context, podID string) error {
    _, err := s.Delete(ctx, podID, nil)
    return err
}
```

那么，这个实现了 rest.StandardStorage 通用接口的真正 Storage 又是什么？从 Master 对象的初始化函数中，我们发现了下面的相关代码：

```
func (m *Master) init(c *Config) {
   healthzChecks := []healthz.HealthzChecker{}
   m.clock = util.RealClock{}
   podStorage := podetcd.NewStorage(c.EtcdHelper, c.KubeletClient)
   podRegistry := pod.NewRegistry(podStorage.Pod)
```

Master 对象创建了一个私有变量 podStorage，其类型为 PodStorage（pkg/registry/pod/etcd/etcd.go），Pod 注册表服务实例（podRegistry）里真正的 Storage 是 podStorage.Pod。下面是 podetcd 的函数 NewStorage 中的关键代码：

```
func NewStorage(h tools.EtcdHelper, k client.ConnectionInfoGetter) PodStorage {
store := &etcdgeneric.Etcd{
        NewFunc:     func() runtime.Object { return &api.Pod{} },
        NewListFunc: func() runtime.Object { return &api.PodList{} },
.....
return PodStorage{
        Pod:         &REST{*store},
        Binding:     &BindingREST{store: store},
        Status:      &StatusREST{store: &statusStore},
        Log:         &LogREST{store: store, kubeletConn: k},
        Proxy:       &ProxyREST{store: store},
        Exec:        &ExecREST{store: store, kubeletConn: k},
        PortForward: &PortForwardREST{store: store, kubeletConn: k},
    }
```

在上述代码中我们看到：位于 pkg/registry/generic/etcd/etcd.go 里的 etcd 才是真正的 Storage 实现。而具体操作 etcd 的代码是靠 tools.EtcdHelper 这个类完成的，通过分析 etcd.go 里的 func (e *Etcd)Create(ctx api.Context, obj runtime.Object)方法，我们知道创建一个 etcd 里的键值对的关键逻辑如下。

- 获取对象的名字：name, err := e.ObjectNameFunc(obj)。
- 获取 Key：key, err := e.KeyFunc(ctx, name)。
- 生成一个空的 Object 对象：out := e.NewFunc()。
- 将键值对写入 etcd：在 e.Helper.CreateObj(key, obj, out, ttl)方法中通过调用 runtime.Codec 完成从对象到字符串的转换，最终保存到 etcd 中。

◎ 回调创建完成后的处理逻辑：e.AfterCreate(out)。

注意到之前 PodStorage 创建 store 时重载了 ObjectNameFunc()、KeyFunc()、NewFunc()等函数，于是完成了针对 Pod 的创建过程，Kubernetes API 服务中的其他数据对象也都遵循同样的设计模式。

进一步研究代码，我们发现 PodStorage 中的 Pod、Binding、Status 等属性是 pkg/api/rest/rest.go 中几个不同的 Rest 接口的实现，并且通过 etcdgeneric.Etcd 这个实例来完成 Pod 的一些具体操作，比如这里的 StatusREST。下面是其相关代码片段：

```
// StatusREST implements the REST endpoint for changing the status of a pod.
type StatusREST struct {
    store *etcdgeneric.Etcd
}
// New creates a new pod resource
func (r *StatusREST) New() runtime.Object {
    return &api.Pod{}
}
// Update alters the status subset of an object.
func (r *StatusREST) Update(ctx api.Context, obj runtime.Object) (runtime.Object,
bool, error) {
    return r.store.Update(ctx, obj)
}
```

表 6.2 展现了 PodStorage 中的各个 XXXREST 接口与 pkg/api/rest/rest.go 里的相关 Rest 接口的一一对应关系。

表 6.2 PodStorage 中的各个 XXXREST 接口与 pkg/api/rest/rest.go 里的相关 Rest 接口的一一对应关系

PodStorage Rest 接口	对应 API Rest 框架的接口	接 口 功 能
REST	rest.Redirector	重定向资源的路径
	rest.CreaterUpdater	资源创建、更新接口
	rest.Lister	资源列表查询接口
	rest.Watcher	Watcher 资源变化接口
	rest. GracefulDeleter	支持延迟的资源删除接口
	rest.Getter	获取具体资源的信息接口
BindingREST	rest.Creater	创建资源的接口
StatusREST	Rest.Updater	更新资源的接口
LogREST	rest.GetterWithOptions	获取资源的接口
ExecREST\ProxyREST\ PortForwardREST	rest.Connecter	连接资源的接口

其中 PodStorage.REST 接口究竟实现了哪些 API Rest 接口，这个比较隐晦，笔者也花费了一些时间来研究这个问题，这涉及 Go 语言的一个特殊特性：结构体内嵌一个其他类型的结构体指针，就可以使用内嵌结构体的方法，相当于面向对象语言中的“继承”。而 PodStorage.REST

恰恰嵌套了 etcdgeneric.Etcd 类型的匿名指针：&REST{*store}，而 etcdgeneric.Etcd 则实现了 rest.Creater、rest.Lister、rest.Watcher 等资源管理接口的所有方法，PodStorage.REST 也“继承”了这些接口。

我们回头看看下面这段来自 api_installer.go 的 registerResourceHandlers 函数中的片段：

```
creater, isCreater := storage.(rest.Creater)
namedCreater, isNamedCreater := storage.(rest.NamedCreater)
lister, isLister := storage.(rest.Lister)
getter, isGetter := storage.(rest.Getter)
getterWithOptions, isGetterWithOptions := storage.(rest.GetterWithOptions)
deleter, isDeleter := storage.(rest.Deleter)
gracefulDeleter, isGracefulDeleter := storage.(rest.GracefulDeleter)
updater, isUpdater := storage.(rest.Updater)
patcher, isPatcher := storage.(rest.Patcher)
watcher, isWatcher := storage.(rest.Watcher)
_, isRedirector := storage.(rest.Redirector)
connecter, isConnecter := storage.(rest.Connecter)
storageMeta, isMetadata := storage.(rest.StorageMetadata)
```

上述代码对 storage 对象进行判断，以确定并标记它所满足的 API Rest 接口类型，而接下来的这段代码在此基础上确定此接口所包含的 actions，后者则对应到某种 HTTP 请求方法（GET/POST/PUT/DELETE）或者 HTTP PROXY、WATCH、CONNECT 等动作：

```
    ctions = appendIf(actions, action{"GET", itemPath, nameParams, namer}, isGetter)
    actions = appendIf(actions, action{"PATCH", itemPath, nameParams, namer},
isPatcher)
    actions = appendIf(actions, action{"DELETE", itemPath, nameParams, namer},
isDeleter)
    actions = appendIf(actions, action{"WATCH", "watch/" + itemPath, nameParams,
namer}, isWatcher)
    actions = appendIf(actions, action{"PROXY", "proxy/" + itemPath + "/{path:*} ",
proxyParams, namer}, isRedirector)
    actions = appendIf(actions, action{"CONNECT", itemPath, nameParams, namer},
isConnecter)
```

我们注意到 rest.Redirector 类型的 storage 被当作 PROXY 进行处理，由 apiserver.ProxyHandler 进行拦截，并调用 rest.Redirector 的 ResourceLocation 方法获取到资源的处理路径（可能包括一个非空的 http.RoundTripper，用于处理执行 Redirector 返回的 URL 请求）。Kubernetes API Server 中 PROXY 请求存在的意义在于透明地访问其他某个节点（比如某个 Minion）上的 API。

最后，我们来分析下 registerResourceHandlers 中完成从 rest.Storage 到 restful.Route 映射的最后一段关键代码。下面是 rest.Getter 接口的 Storage 的映射代码：

```
case "GET": // Get a resource.
```

```
    var handler restful.RouteFunction
    handler = GetResource(getter, reqScope)
    doc := "read the specified " + kind
    route := ws.GET(action.Path).To(handler).Filter(m).Doc(doc).
    Param(ws.QueryParameter("pretty", "If 'true', then the output is pretty printed.
")).
    Operation("read"+namespaced+kind+strings.Title(subresource)).
    Produces(append(storageMeta.ProducesMIMETypes(action.Verb), "application/
json")...).
    Returns(http.StatusOK, "OK", versionedObject).Writes(versionedObject)

    addParams(route, action.Params)
    ws.Route(route)
```

上述代码首先通过函数 GetResource()创建了一个 restful.RouteFunction，然后生成一个 restful.route 对象，最后注册到 restful.WebService 中，从而完成了 rest.Storage 到 Rest 服务的“最后一公里”通车。GetResource()函数存在于 pkg/apiserver/resthandler.go 里，resthandler.go 提供了各种具体的 restful.RouteFunction 的实现函数，是真正触发 rest.Storage 调用的地方。下面是 GetResource()方法的主要代码，可以看出这里是调用 rest.Getter 接口的 Get()方法以返回某个资源对象：

```
    func GetResource(r rest.Getter, scope RequestScope) restful.RouteFunction {
        return getResourceHandler(scope,
            func(ctx api.Context, name string, req *restful.Request) (runtime.Object,
error) {
                return r.Get(ctx, name)
            })
    }
```

看了上面的代码，你可能会有一个疑问：“说好的权限控制呢？”别急，看看下面的资源创建的 createHandler()代码：

```
            if admit.Handles(admission.Create) {
                userInfo, _ := api.UserFrom(ctx)
                err = admit.Admit(admission.NewAttributesRecord(obj, scope.Kind,
namespace, name, scope.Resource, scope.Subresource, admission.Create, userInfo))
                if err != nil {
                    errorJSON(err, scope.Codec, w)
                    return
                }
            }
```

资源的 Update、Delete、Connect、Patch 等操作都有类似的权限控制，从 Admit 的参数 admission.Attributes 的属性来看，第三方系统可以开发细粒度的权限控制插件，针对任意资源的任意属性进行细粒度的权限控制，因为资源对象本身都传递到参数中了。