第 5 章

Generic Server

API Server 内部由四类子 Server 构成，分别是主 Server、扩展 Server、聚合器和聚合 Server，从实现角度看，子 Server 的构造高度同质化，包括但不限于：

(1) 都需要 Web Server 的基本功能，如端口监听、安全连接机制等。

(2) 将所负责的 API 以 RESTful 形式对外暴露。

(3) 提供装载和暴露 Kubernetes API 的统一接口。

(4) 请求的过滤、权限检查和准入控制。

为了避免重复造轮子，Kubernetes 项目为所有子 Server 开发了统一基座——Generic Server 来集中提供通用能力。本章将抽丝剥茧，讲解 Generic Server 的设计与实现。

5.1 Go 语言实现 Web Server

Generic Server 首先是一个 Web Server，在 Go 语言中构造 Web Server 用到的全部依赖均在包 net.http 中。一个基础 Web Server 的代码如下：

```
func main(){
    company := employees{"Jacky": 40, "Tom": 35}
    mux := http.NewServeMux()
    mux.Handle("/employees", http.HandlerFunc(company.allEmployees))
    mux.Handle("/age", http.HandlerFunc(company.age)) //要点①
    Log.Fatal(http.ListenAndServe("localhost:8000", mux))
}

type employees map[string]int

func (company employees) allEmployees(w http.ResponseWriter,
                                req * http.Request) {
    for name, age := range company {
        fmt.Fprintf(w, "%s: %s\n", name, age)
    }
}
```

```
func (company employees) age(w http.ResponseWriter, req * http.Request) {
    item := req.url.Query().Get("name")
    age, ok := company[item]
    if !ok {
        w.WriteHeader(http.StatusNotFound)
        fmt.Fprintf(w, "no such people in company: %q\n", item)
        return
    }
    fmt.Fprintf(w, "%s\n", age)
}
```

这个简单的 Server 揭示了编写一个 Web Server 所需要的主要构件。

(1) http 包：这个包提供了端口监听等底层服务，有它的辅助，制作一个 Web Server 只需调用名为 ListenAndServe()的方法。

(2) Handler：一个名为 Handler 的对象会被 http 包中的代码用来接收所有请求并调用目标函数，去响应请求。之所以被称为 Handler，是因为它实现了 http.Handler 接口，该接口定义了 ServeHTTP()方法，在 Generic Server 源码中将常常看到它。在以上示例代码中，变量 mux 就是这个 Handler。在调用 http.ListenAndServe 方法时，mux 被作为实参交给启动的 Server。后续到来的请求都被交给它，http 包通过调用 mux 的 ServeHTTP()方法完成请求转交。

(3) 请求处理函数：这个是用户实现的函数，名字任取，但参数类型需要固定，示例中的 allEmployees()方法和 age()方法都是请求处理函数。请求处理函数会被上述 Handler 调用，正是在这些函数中，一个请求最终得到响应。

一个简单的 Web Server 通过以上 3 个构件便可构造出来。当 Server 变得复杂时一个潜在问题便会突显出来：http.ListenAndServe()方法只接收了一个 Handler 对象，所有请求都将转交于它，当 Server 对外提供众多服务时，单一 Handler 的内部逻辑有迅速膨胀失控的风险。为了管控这一风险，ServerMux(也就是示例中的变量 mux 对象具有的类型)被应用上来，ServerMux 是 http 包提供的请求分发器。

ServerMux 是一个 Handler，同样实现 http.Handler 接口，所以它就能被作为实参调用 http.ListenAndServe()方法，但 ServerMux 这个 Handler 比较特殊，它只做请求的分发而不含响应逻辑。ServerMux 握有 URL 模式与其 Handler 的映射关系，分发的过程是将符合一个 URL 模式的请求路由到相应 Handler 的过程。示例代码中在要点①处调用 mux.Handle()方法就是在 mux 内构建 URL 到下层 Handler 的映射关系。

这里读者可自行解答一个问题：上述代码在要点①处调用了方法 http.HandlerFunc()将另一种方法 company.age()包装成接口 http.Handler 的实例，使它有资格成为方法 http.ListenAndServe()的实参，http.HandlerFunc()方法是怎么做到这一点的呢？答案在其源码中。

以上 Web Server 是极度简化后的结果，没有考虑支持 TLS、没有控制 Http Header、没有超时设置等，这样的 Server 直接用于生产显然不行。事实也的确如此，Go 语言提供了功

能更加完善的选择：http.Server 结构体。需要强调的是，http.Server 最终仍是依靠与 http.ListenAndServe()方法同根同源的其他方法来构建出 Web Server，它最大的价值在于提供了更简便的 Server 配置方式。同时，Handler 和 ServerMux 在 http.Server 中的作用照旧。http.Server 提供了丰富的配置选项，该结构体的关键字段如下：

```
//代码 5-1 net/http/server.go
type Server struct {
    Addr string
    Handler Handler
    DisableGeneralOptionsHandler bool
    TLSConfig * tls.Config
    ReadTimeout time.Duration
    ReadHeaderTimeout time.Duration
    WriteTimeout time.Duration
    IdleTimeout time.Duration
    MaxHeaderBytes int
    TLSNextProto map[string]func( * Server, * tls.Conn, Handler)
    ConnState func(net.Conn, ConnState)
    ErrorLog * log.Logger
    BaseContext func(net.Listener) context.Context
    ConnContext func(ctx context.Context, c net.Conn) context.Context
    inShutdown atomic.Bool
    disableKeepAlives atomic.Bool
    nextProtoOnce     sync.Once
    nextProtoErr      error

    mu        sync.Mutex
    listeners  map[ * net.Listener]struct{}
    activeConn map[ * conn]struct{}
    onShutdown []func()

    listenerGroup sync.WaitGroup
}
```

http.Server 结构体的 Handler 字段在实践中会接收一个 ServerMux 转发器，这和上述简单的 Web Server 是一致的。当使用 http.Server 时，首先需要声明该结构体变量并给该变量的各个字段赋值，然后只需调用变量的 Serve()方法便可完成启动。Kubernetes 的 Generic Server 正是如此构建起来的，变量声明部分的代码如下：

```
//代码 5-2 staging/k8s.io/apiserver/pkg/server/secure_serving.go
secureServer := &http.Server{
    Addr:            s.Listener.Addr().String(),
    Handler:        handler,
    MaxHeaderBytes: 1 << 20,
    TLSConfig:      tlsConfig,

    IdleTimeout:       90 * time.Second,
    ReadHeaderTimeout: 32 * time.Second,
}
```

变量 secureServer 被作为参数调用另一种方法[①]，在该方法内，secureServer 实例的 Serve()方法被调用，从而启动 Server，代码如下：

```
//代码 5-3 staging/k8s.io/apiserver/pkg/server/secure_serving.go
go func() {
    defer utilruntime.HandleCrash()
    defer close(listenerStoppedCh)

    var listener net.Listener
    listener = tcpKeepAliveListener{ln}
    if server.TLSConfig != nil {
            listener = tls.NewListener(listener, server.TLSConfig)
    }

    err := server.Serve(listener) //启动 Server

    msg := fmt.Sprintf("Stopped listening on %s", ln.Addr().String())
    select {
    case <-stopCh:
            klog.Info(msg)
    default:
            panic(fmt.Sprintf("%s due to error: %v", msg, err))
    }
}()
```

为了简化以便于理解，以上两个代码片段没有包含 TLS，以及 HTTP2.0 等相关设置环节，感兴趣的读者可以将 secure_serving.go 作为入口进行查阅。

5.2 go-restful

5.2.1 go-restful 简介

API Server 可以采用 RESTful 的形式对外提供服务，它用 JSON 为信息表述格式，以 HTTP 的 POST、PUT、DELETE、GET 和 PATCH 等方法来分别表示创建、全量修改、删除、读取和部分修改等操作。为了支持 RESTful，在 Web Server 的基础能力之上，Generic Server 借助额外框架来构建相关的能力。在 Java 世界里，VMware 的 Spring Web 框架包含了这部分能力，而在 Go 的世界里也有框架做这件事情，Kubernetes 选择了 go-restful。为了全面地理解 API Server 的 RESTful 能力，读者需要认识一下 go-restful。

go-restful 库为开发者构建支持 RESTful 的 Web 应用提供了开箱即用的能力，包括但不限于以下几种：

(1) 对服务请求进行路由能力。粒度细到可以根据路径参数映射到响应函数。

① 方法名为 RunServer，secureServer 对应的形式参数名称为 server。

(2) 路由器可配置,路由算法更出众。

(3) 强大的请求对象(Request)API,可以从 JSON/XML 解析为结构体实例并获取其中的路径参数、一般参数和请求头等信息。

(4) 方便的响应对象(Response) API,序列化结果到 JSON/XML 并支持客制化格式、设置响应头等。

(5) 请求处理的拦截能力。可在处理流程中以加 Filter 的形式截留处理请求。

(6) 提供 Container,以此来汇集多个 Web Service,它扮演了 ServerMux 的角色。

(7) 跨源资源共享(CORS)的能力。

(8) 支持 OpenAPI。

(9) 自动拦截恐慌(Panic),形成 HTTP 500 返回。

(10) 将错误映射到 HTTP 404/405/406/415。

(11) 可配置的 log/trace。

5.2.2 go-restful 中的核心概念

go-restful 是简单易用的框架,如果读者有网络应用编程经验,则很快便可参悟其工作原理。它有很少的几个核心概念:Container、Web Service 和 Route,围绕着它们,go-restful 构建了主体功能。go-restful 核心概念及其相互关系如图 5-1 所示。

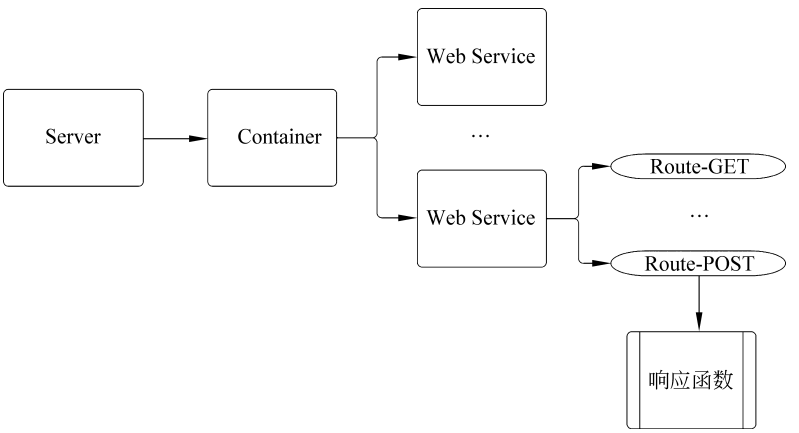

图 5-1 go-restful 核心概念及其相互关系

1. Web Service

一个 Web 服务是服务器暴露给外部的一个访问入口,在 RESTful 背景下,一个 Web 服务代表一个资源。它具有一个端点,即一个 URL,对该类资源的增、删、改、查均通过该端点进行。例如以 RESTful 形式暴露一个 Web 服务,用于操作“用户”这一资源,这个 Web 服务的端点便可以是:

```
employeedb/user
```

如果该服务暴露于域名 https://www.<我的域名>.com/下，则通过类似如下 HTTP 请求可以创建一个叫作 jacky 的用户：

```
$ curl -X POST -H "Content-Type: application/json" -d '{"name": "jacky",
"department": "HR"}' https://www.<我的域名>.com/employeedb/user
```

在 go-restful 中，代表一个 Web Service 的变量具有类型 WebService 结构体。

2. Container

go-restful 中的 Container 位于 Web 服务的上面一层，是 Web 服务的容器。一个 Web Server 当然不只提供一种资源，所以需要一个机制把 Web 服务组织起来并暴露出去，这就是 Container 的责任。从另一个角度讲更容易理解其作用，读者是否还记得 ServerMux?

Container 也实现了 http.Handler 接口，它将扮演请求转发器的角色，如代码 5-4 所示。

```
//代码 5-4 Container 实现了 http.Handler 接口
func (c *Container) ServeHTTP(httpWriter http.ResponseWriter, httpRequest *
http.Request) {
    //如果对内容的 encoding 被禁止，则跳过这一步
    if !c.contentEncodingEnabled {
        c.ServeMux.ServeHTTP(httpWriter, httpRequest)
        return
    }
    //内容 encoding 被启用了

    //如果 httpWriter 已经是 CompressingResponseWriter 类型，则跳过这一步
    if _, ok := httpWriter.(*CompressingResponseWriter); ok {
        c.ServeMux.ServeHTTP(httpWriter, httpRequest)
        return
    }

    writer := httpWriter
    //CompressingResponseWriter 在所有操作完成后需要被关闭
    defer func() {
        if compressWriter, ok := writer.(*CompressingResponseWriter); ok {
            compressWriter.Close()
      }
    }()

    doCompress, encoding := wantsCompressedResponse(httpRequest, httpWriter)
    if doCompress {
        var err error
        writer, err = NewCompressingResponseWriter(httpWriter, encoding)
        if err != nil {
            log.Print("unable to install compressor: ", err)
            httpWriter.WriteHeader(http.StatusInternalServerError)
            return
        }
    }

    c.ServeMux.ServeHTTP(writer, httpRequest)
}
```

Container 具有 Add 方法,通过它可以逐个把 Web 服务实例纳入 Container 的转发服务范围,后续这个 Container 实例将被作为 Hanlder 交给 http.Server 实例,这样所有请求都会交由该 Container 去分发。

3. Route

路径(Route)位于 Web 服务的下面一层,从程序上看每个 WebService 结构体变量都会包含一个或多个 Route,每个 Route 负责将一类针对资源的操作请求路由到正确的响应函数。Route 上会有几个关键信息:

(1) 做什么操作(HTTP 方法所代表)。

(2) 有没有参数? 如果有,则其值是什么。

(3) 由哪个函数来响应针对该资源的该类操作。

至于对哪种资源进行操作,这个已经由 Route 所属的 Web Service 决定了。

5.2.3 使用 go-restful

有了上述概念就可以构建支持 RESTful 的 Web Server 了。下面这段示例代码创建了一个 RESTful 服务,对应上述"用户"端点的例子,代码如下:

```
func main() {
    ws := new(restful.WebService)
    ws.
        Path("/user").
        Consumes(restful.MIME_XML, restful.MIME_JSON).
        Produces(restful.MIME_JSON, restful.MIME_XML)
    ws.Route(ws.GET("{user-id}").To(findUser))
    container := restful.NewContainer()
    container.Add(ws)
    server := &http.Server{Addr: ":8080", Handler: container}
    log.Fatal(server.ListenAndServe())
}

func findUser(req * restful.Request, resp * restful.Response) {
    io.WriteString(resp, "hello, there is only one user - Jacky")
}
```

上述代码先创建一个 Web 服务实例,指明端点为/user,然后为它添加一个 Route,把针对某个用户 id 的 GET 请求路由到函数 findUser();接下来创建 Container,容纳以上 Web 服务;最后以该 Container 为 Handler 构造 http.Server 并启动它。这样一个支持 RESTful 服务的 Web Server 就构建完成了。

5.3 OpenAPI

5.3.1 什么是 OpenAPI

OpenAPI 是一个 HTTP 程序编程接口描述规范,要解决的问题是兼顾人类可读与机

器可读可理解，以统一格式准确地描述基于 HTTP 的服务接口。服务的生产者为了让使用者正确地进行消费，一般会附加文档去描述如何对服务进行调用，描述一般涵盖 API 的名称、参数列表及其类型、返回值类型和意义等。传统式描述由人撰写文档供开发者阅读，这种形式的描述既不精确也容易过时，不利于使用。

机器可读可理解是围绕 API 进行各种自动化的前提。以自动化测试领域为例，针对一个即将发布的 API 需要完整地进行测试，测试用例越多越有利，仅依靠手工构建测试用例工作量可想而知。对于一个 API 来讲，如果给定输入，则其输出是一定的，并且每个输入的边界值深受数据类型影响，可以根据类型推算出来；如此一来，如果工具能够读懂一个 API 的输入和输出类型，则让工具生成边界值测试用例和主场景测试用例就是可行的。

机器可读可理解的另一个典型应用场景是生成供人类阅读的 API 文档。文档质量的高低取决于更新的频度，如果不能及时地将最新情况更新上去，则再优美的文档也是价值寥寥的。相比于文档撰写，开发人员更注重程序的编写。借助机器可读的 API 描述，可以根据预制格式模板生成供人阅读的文档，这种生成不需要耗费人力，只需作为工程打包的一部分，通过脚本自动生成内容。

OpenAPI 不是第 1 个 RESTful API 描述解决方案，但是最成功的一个，这归功于它的简单轻便。OpenAPI 主张，针对一个 Web 应用提供的服务接口，应以 YAML/JSON 格式撰写一份规格说明书，并且该说明书是一份自包含的文档，即不需要借助外部信息就能理解内部内容。采用 YAML/JSON 格式使机器和人都可以阅读，并且文档相对于 XML 等其他格式稍小。OpenAPI v3 文档的重要元素见表 5-1。

表 5-1 OpenAPI v3 文档的重要元素

#	名 称	作 用
1	(空)	最顶层节点，在 JSON 格式下代表定义于最外层的 Object，符号上就是一对大括号，其内包含直接属性，称为子节点
2	openapi	顶层节点的直接子节点，指明使用的 OpenAPI 版本号。目前大版本有两个，即 v2 和 v3，小版本众多
3	info	顶层节点的直接子节点，给出文档的一些元属性，例如标题、作者等
4	paths	顶层节点的直接子节点，给出了各个 RESTful 服务的端点、操作及输入输出参数等，非常重要
5	components	顶层节点的直接子节点，用于定义可复用的信息。例如在 paths 内定义的各个端点会用到相同的参数，可以放在这里定义一次，多处复用

其中信息量最大且最重要的节点是 paths，每个 RESTful 服务都会在 paths 中有一条描述信息，除了服务路径信息，同时给出了服务支持的 HTTP 方法、输入输出参数及其类型、输出 HTTP 状态值等。

5.3.2 Kubernetes 使用 OpenAPI 规格说明

早在 v1.5 中 Kubernetes 就开始使用 OpenAPI v2 严谨描述 Kubernetes API Server 中

所有内置 API 的端点了。从 v1.23 开始,在支持 v2 的同时引入了 OpenAPI v3,在 v1.24 版中对 v3 的支持到达 beta 状态[①],并最终在 v1.27 中 GA 正式发布。本书所基于的 Kubernetes 版本恰是 v1.27,打开其源代码,导航到 api/openapi-spec/v3 目录,可以看到其内有一系列 JSON 文件,它们是 API 端点的 OpenAPI 规格说明文件。Kubernetes 将全部 API 端点按照 API Group 和 Version 分组提供规范描述,每个 API Group 和 Version 有一份规格说明,部分文件如图 5-2 所示。

图 5-2 API 端点规格说明文件

注意:用 OpenAPI 描述的并非 Kubernetes API,而是它们的端点。基于 OpenAPI 规格说明文件,客户端可以向它们发起 GET、POST 等 HTTP 请求。

用户可以通过访问如下端点来向 API Server 索取集群提供的 OpenAPI 规格说明文档。

(1) /openapi/v2:获取 v2 版本的文档。

(2) /openapi/v3:获取 v3 版本的文档。由于在 v3 中 Kubernetes 以多个文件的形式对 API 端点进行描述,所以会有多个文件。这一端点将返回所有文件的获取地址。

没有无缘无故的爱和恨,Kubernetes 为什么拥抱 OpenAPI? 利用 OpenAPI,Kubernetes 达到了如下目的:

(1) 为开发人员自动生成 API Server 端点文档。虽然基于机器可读可理解的 OpenAPI 描述文档,程序已经可以自主做很多事情了,但人始终是决策者,为开发人员提供文档,从而帮助其理解 API 的能力和使用方式是必要的,问题是如何高效准确地撰写。有 Java 经验的读者一定知道 Javadoc 这一机制:通过在类、方法上加特定格式的注释就可以生成 HTML 格式的程序文档。辅以一些格式定义,基于 OpenAPI 描述文档也可以生成针对 API Server 端点的 HTML 文档,供人阅读。技术上这完全行得通,毕竟 OpenAPI 文档中已经具有了完备的 API 端点信息。Kubernetes 这么做了,为 v1.27 版生成的端点描述文档发布在 https://kubernetes.io/docs/reference/generated/kubernetes-api/v1.27/。同时,Kubernetes 开放了工具,以便帮助用户在本地自助构建这些 HTML 文档[②]。

(2) 支持数据一致性校验。Kubernetes 官网上描述了一个很有说服力的例子,在 v1.8 之前,如果用户在定义一个 Deployment 时错误地将 replicas 拼写为 replica,则可能产生灾难性后果。因为 ReplicaSet 控制器将会使用默认的 replicas 值(默认为 1)去调整 Pod 的数

① 默认可用,不必手动打开 feature gate。

② 步骤参见 https://kubernetes.io/zh-cn/docs/contribute/generate-ref-docs/kubernetes-api/。

量。可想而知,目标服务极有可能因此下线。这个问题的解决是改进客户端程序,让它下载 API Server 所有端点的 OpenAPI 规格说明,在将每个请求发送至 API Server 前,均须用规格说明校验用户提交的请求,如果校验失败,则停止发送。这很有效,在 OpenAPI 描述文档内,针对请求的参数有准确的描述。以 apps/v1 为例,其下 API 端点规格说明中关于 Deployment 的一个端点的描述如图 5-3 所示。

图 5-3 apps/v1 下端点描述文档节选

注意它的 requestBody 内的 Schema 信息 #/components/schemas/io.k8s.api.apps.v1.Deployment,该 Schema 定义可以在规格说明的 components 节点下找到,它完整地描述了 Deployment 的 v1 外部版本具有的属性,如图 5-4 所示。

有如此精确的信息在手,客户端便可胜任众多校验。获取 OpenAPI 描述文件的工作也被简化了,客户端可以借助 client-go 工具库提供的 client 完成。

5.3.3 生成 API OpenAPI 规格说明

用 OpenAPI 规范精确地描述 REST API 也是有代价的:工作量大,并且不容有错。API Server 提供那么多 API 资源,有如此之多的端点,手工为它们撰写 OpenAPI 服务规格说明不可接受。解决办法是通过代码生成来自动生成。生成与消费的过程如图 5-5 所示。