启动一个最简单的 RESTful API 服务器

本节核心内容

- 启动一个最简单的 RESTful API 服务器
- 设置 HTTP Header
- API 服务器健康检查和状态查询
- 编译并测试 API

> 本小节源码下载路径：demo01 (https://github.com/lexkong/apiserver_demos/tree/mas
>
> 可先下载源码到本地，结合源码理解后续内容，边学边练。

如无特别说明，本小册的操作和编译目录均是 API 源码的根目录，并且本 API 服务器名字（也是二进制命令的名字）小册中统一叫作 apiserver。

REST Web 框架选择

要编写一个 RESTful 风格的 API 服务器，首先需要一个 RESTful Web 框架，笔者经过调研选择了 GitHub star 数最多的 Gin (https://github.com/gin-gonic/gin)。采用轻量级的 Gin 框架，具有如下优点：高性能、扩展性强、稳定性强、相对而言比较简洁（查看 性能对比 (https://github.com/gin-

gonic/gin/blob/master/BENCHMARKS.md)）。关于 Gin 的更多介绍可以参考 Golang 微框架 Gin 简介 (https://www.jianshu.com/p/a31e4ee25305)。

加载路由，并启动 HTTP 服务

main.go 中的 main() 函数是 Go 程序的入口函数，在 main() 函数中主要做一些配置文件解析、程序初始化和路由加载之类的事情，最终调用 http.ListenAndServe() 在指定端口启动一个 HTTP 服务器。本小节是一个简单的 HTTP 服务器，仅初始化一个 Gin 实例，加载路由并启动 HTTP 服务器。

编写入口函数

编写 main() 函数，main.go 代码：

```
package main

import (
    "log"
    "net/http"

    "apiserver/router"

    "github.com/gin-gonic/gin"
)

func main() {
    // Create the Gin engine.
    g := gin.New()

    // gin middlewares
    middlewares := []gin.HandlerFunc{}

    // Routes.
    router.Load(
        // Cores.
        g,

        // Middlewares.
        middlewares...,
    )

    log.Printf("Start to listening the incoming
requests on http address: %s", ":8080")
    log.Printf(http.ListenAndServe(":8080",
g).Error())
}
```

加载路由

main() 函数通过调用 router.Load 函数来加载路由（函数路径为 router/router.go，具体函数实现参照 demo01/router/router.go (https://github.com/lexkong/apiserver_demos/blob/master/

```
    "apiserver/handler/sd"

    ....

    // The health check handlers
    svcd := g.Group("/sd")
    {
        svcd.GET("/health", sd.HealthCheck)
        svcd.GET("/disk", sd.DiskCheck)
        svcd.GET("/cpu", sd.CPUCheck)
        svcd.GET("/ram", sd.RAMCheck)
    }
```

该代码块定义了一个叫 sd 的分组，在该分组下注册了 /health、/disk、/cpu、/ram HTTP 路径，分别路由到 sd.HealthCheck、sd.DiskCheck、sd.CPUCheck、sd.RAMCh 函数。sd 分组主要用来检查 API Server 的状态：健康状况、服务器硬盘、CPU 和内存使用量。具体函数实现参照 demo01/handler/sd/check.go (https://github.com/lexkong/apiserver_demos/blob/master/

设置 HTTP Header

router.Load 函数通过 g.Use() 来为每一个请求设置 Header，在 router/router.go 文件中设置 Header：

```
    g.Use(gin.Recovery())
    g.Use(middleware.NoCache)
    g.Use(middleware.Options)
    g.Use(middleware.Secure)
```

- gin.Recovery()：在处理某些请求时可能因为程序 bug 或者其他异常情况导致程序 panic，这时候为了不影响下一次请求的调用，需要通过 gin.Recovery()来恢复 API 服务器
- middleware.NoCache：强制浏览器不使用缓存
- middleware.Options：浏览器跨域 OPTIONS 请求设置
- middleware.Secure：一些安全设置

> middleware包的实现见
> [demo01/router/middleware](https://github.com/lexkong/apiserver_demos/tree/mas
> (https://github.com/lexkong/apiserver_demos/tree/mas

API 服务器健康状态自检

有时候 API 进程起来不代表 API 服务器正常，笔者曾经就遇到过这种问题：API 进程存在，但是服务器却不能对外提供服务。因此在启动 API 服务器时，如果能够最后做一个自检会更好些。笔者在 apiserver 中也添加了自检程序，在启动 HTTP 端口前 go 一个 pingServer 协程，启动 HTTP 端口后，该协程不断地 ping /sd/health 路径，如果失败次数超过一定次数，则终止 HTTP 服务器进程。通过自检可以最大程度地保证启动后的 API 服务器处于健康状态。自检部分代码位于 main.go 中：

```
func main() {
    ....

    // Ping the server to make sure the router is
```

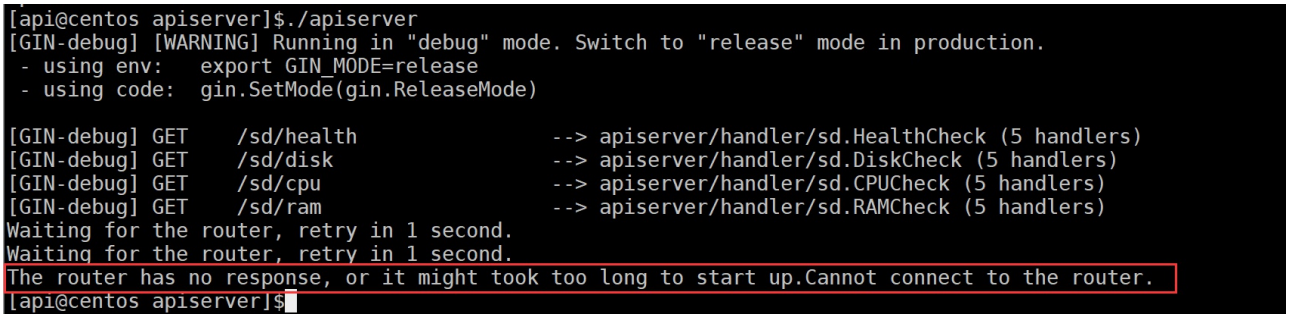
```
working.
    go func() {
        if err := pingServer(); err != nil {
            log.Fatal("The router has no
response, or it might took too long to start
up.", err)
        }
        log.Print("The router has been deployed
successfully.")
    }()
    ....
}

// pingServer pings the http server to make sure
the router is working.
func pingServer() error {
    for i := 0; i < 10; i++ {
        // Ping the server by sending a GET
request to `/health`.
        resp, err :=
http.Get("http://127.0.0.1:8080" + "/sd/health")
        if err == nil && resp.StatusCode == 200 {
            return nil
        }

        // Sleep for a second to continue the
next ping.
        log.Print("Waiting for the router, retry
in 1 second.")
        time.Sleep(time.Second)
    }
    return errors.New("Cannot connect to the
router.")
```

```
}
```

在 pingServer() 函数中，http.Get 向 http://127.0.0.1:8080/sd/health 发送 HTTP GET 请求，如果函数正确执行并且返回的 HTTP StatusCode 为 200，则说明 API 服务器可用，pingServer 函数输出部署成功提示；如果超过指定次数，pingServer 直接终止 API Server 进程，如下图所示。



> /sd/health 路径会匹配到 handler/sd/check.go 中的 HealthCheck 函数，该函数只返回一个字符串：OK。

编译源码

1. 下载 apiserver_demos 源码包

```
$ git clone
https://github.com/lexkong/apiserver_demos
```

2. 将 apiserver_demos/demo01 复制为 $GOPATH/src/apiserver

```
$ cp -a apiserver_demos/demo01/
$GOPATH/src/apiserver
```

3. 首次编译需要下载 vendor 包

因为 apiserver 功能比较丰富，需要用到很多 Go package，统计了下需要用到 60 个非标准 Go 包。为了让读者更容易地上手编写代码，这里将这些依赖用 go vendor 进行管理，并放在 GitHub 上供读者下载安装，安装方法为：

```
$ cd $GOPATH/src
$ git clone https://github.com/lexkong/vendor
```

4. 进入 apiserver 目录编译源代码

```
$ cd $GOPATH/src/apiserver
$ gofmt -w .
$ go tool vet .
$ go build -v .
```

编译后的二进制文件存放在当前目录，名字跟目录名相同：apiserver。

> 笔者建议每次编译前对 Go 源码进行格式化和代码静态检查，以发现潜在的 Bug 或可疑的构造。

cURL 工具测试 API

cURL 工具简介

本小册采用 cURL 工具来测试 RESTful API，标准的 Linux 发行版都安装了 cURL 工具。cURL 可以很方便地完成对 REST API 的调用场景，比如：设置 Header，指定 HTTP 请求方法，指定 HTTP 消息体，指定权限认证信息等。通过 -v 选项也能输出 REST 请求的所有返回信息。cURL 功能很强大，有很多参数，这里列出 REST 测试常用的参数：

-X/--request [GET|POST|PUT|DELETE|…]	指定请求的 HTTP 方法
-H/--header	指定请求的 HTTP Header
-d/--data	指定请求的 HTTP 消息体（Body）
-v/--verbose	输出详细的返回信息
-u/--user	指定账号、密码
-b/--cookie	读取 cookie

典型的测试命令为：

```
$ curl -v -XPOST -H "Content-Type: application/json" http://127.0.0.1:8080/user -d'{"username":"admin","password":"admin1234"}'
```

启动 API Server

```
$ ./apiserver
[GIN-debug] [WARNING] Running in "debug" mode.
Switch to "release" mode in production.
 - using env:   export GIN_MODE=release
 - using code:  gin.SetMode(gin.ReleaseMode)

[GIN-debug] GET    /sd/health                -->
apiserver/handler/sd.HealthCheck (5 handlers)
[GIN-debug] GET    /sd/disk                  -->
apiserver/handler/sd.DiskCheck (5 handlers)
[GIN-debug] GET    /sd/cpu                   -->
apiserver/handler/sd.CPUCheck (5 handlers)
[GIN-debug] GET    /sd/ram                   -->
apiserver/handler/sd.RAMCheck (5 handlers)
Start to listening the incoming requests on http
address: :8080
The router has been deployed successfully.
```

发送 HTTP GET 请求

```
$ curl -XGET http://127.0.0.1:8080/sd/health
OK

$ curl -XGET http://127.0.0.1:8080/sd/disk
OK - Free space: 16321MB (15GB) / 51200MB (50GB)
| Used: 31%

$ curl -XGET http://127.0.0.1:8080/sd/cpu
CRITICAL - Load average: 2.39, 2.13, 1.97 |
Cores: 2

$ curl -XGET http://127.0.0.1:8080/sd/ram
OK - Free space: 455MB (0GB) / 8192MB (8GB) |
Used: 5%
```

可以看到 HTTP 服务器均能正确响应请求。

小结

本小节通过具体的例子教读者快速启动一个 API 服务器，这只是一个稍微复杂点的 "Hello World"。读者可以先通过该 Hello World 熟悉 Go API 开发流程，后续小节会基于这个简单的 API 服务器，一步步构建一个企业级的 API 服务器。