HTTP 调用添加自定义处理逻辑

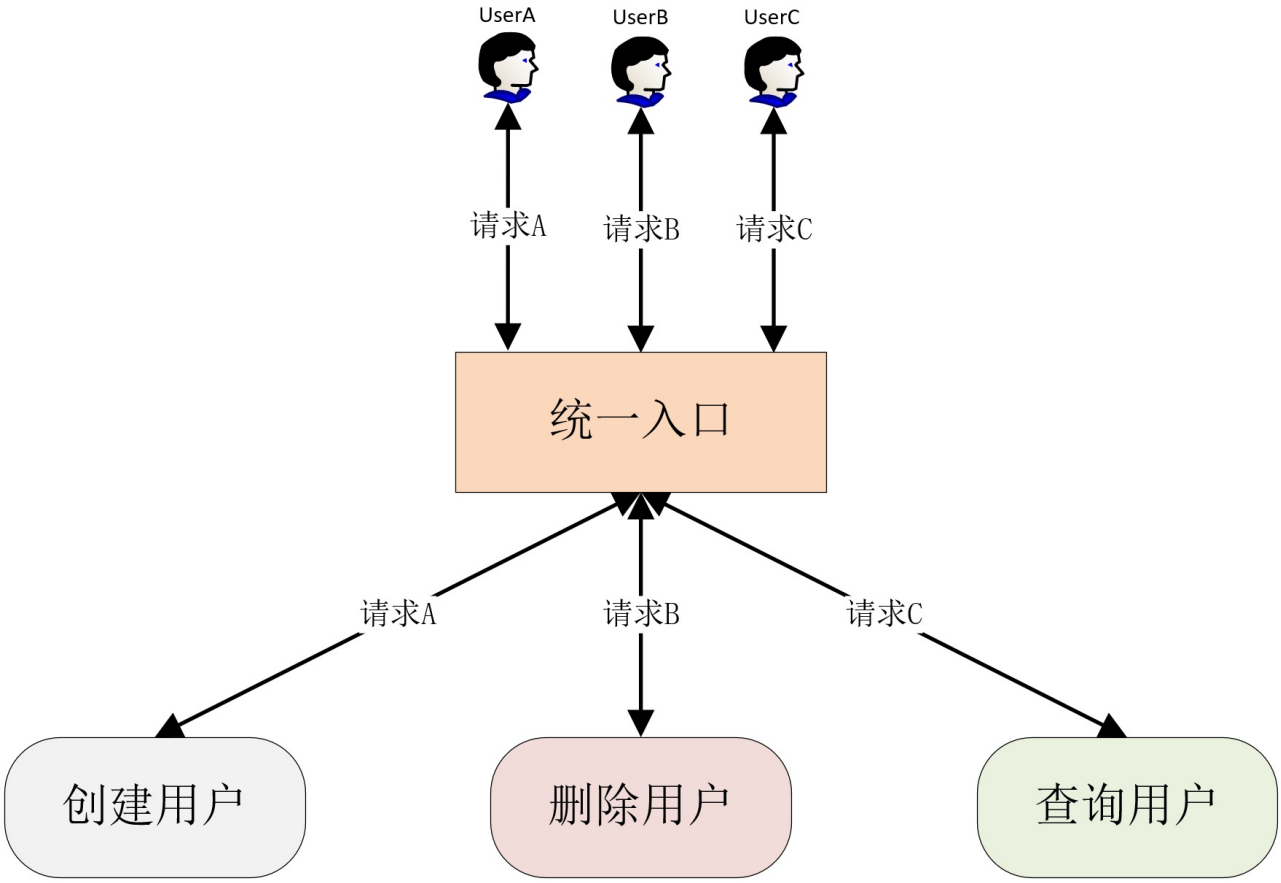
本节核心内容

- 介绍 gin middleware 基本用法
- 介绍如何用 gin middleware 特性给 API 添加唯一请求 ID 和记录请求信息

> 本小节源码下载路径：[demo08 (https://github.com/lexkong/apiserver_demos/tree/mas](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。
>
> 本小节的代码是基于 [demo07 (https://github.com/lexkong/apiserver_demos/tree/mas](https://github.com/lexkong/apiserver_demos/tree/mas) 来开发的。

需求背景

在实际开发中，我们可能需要对每个请求/返回做一些特定的操作，比如记录请求的 log 信息，在返回中插入一个 Header，对部分接口进行鉴权，这些都需要一个统一的入口，逻辑如下：



这个功能可以通过引入 middleware 中间件来解决。Go 的 net/http 设计的一大特点是特别容易构建中间件。apiserver 所使用的 gin 框架也提供了类似的中间件。

gin middleware 中间件

在 gin 中，可以通过如下方法使用 middleware：

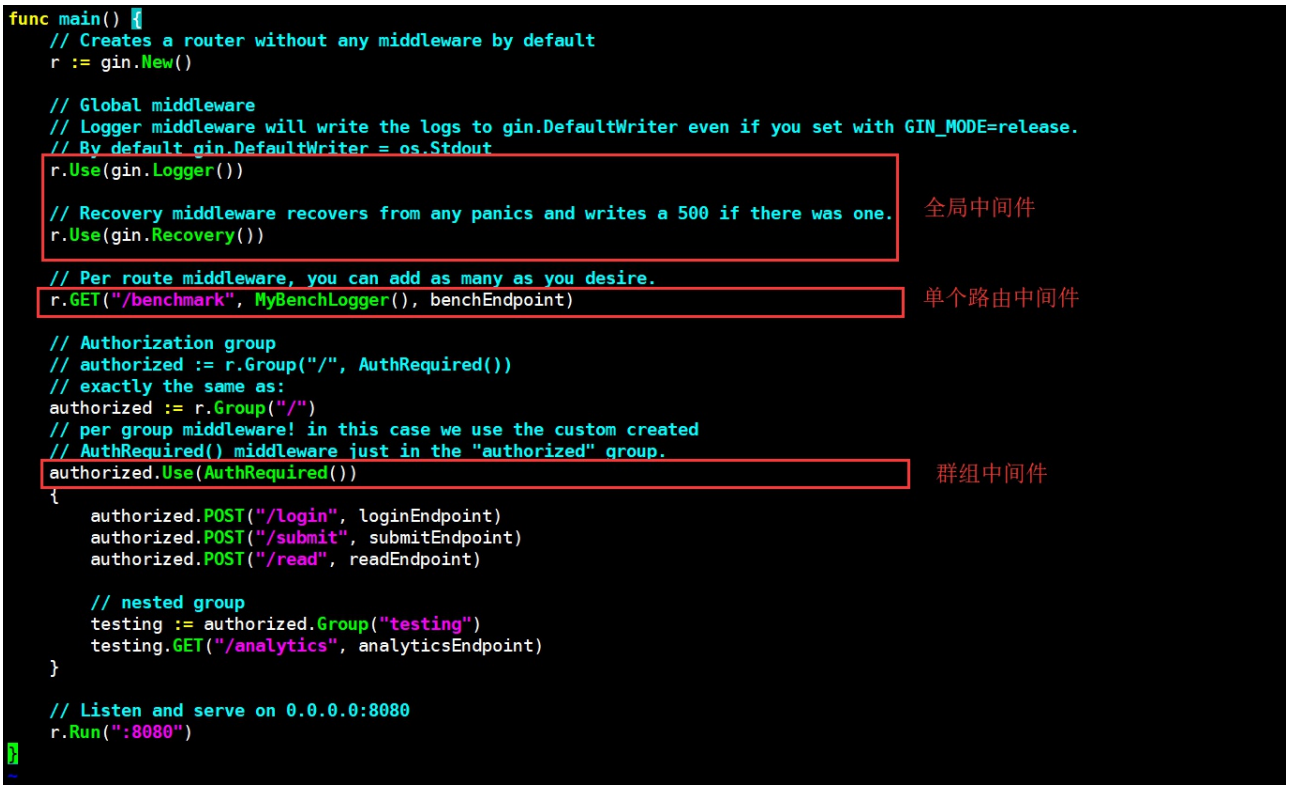
```
g := gin.New()
g.Use(middleware.AuthMiddleware())
```

其中 middleware.AuthMiddleware() 是 func(*gin.Context) 类型的函数。中间件只对注册过的路由函数起作用。

在 gin 中可以设置 3 种类型的 middleware：

- 全局中间件
- 单个路由中间件
- 群组中间件

这里通过一个例子来说明这 3 种中间件。



- 全局中间件：注册中间件的过程之前设置的路由，将不会受注册的中间件所影响。只有注册了中间件之后代码的路由函数规则，才会被中间件装饰。
- 单个路由中间件：需要在注册路由时注册中间件 `r.GET("/benchmark", MyBenchLogger(), benchEndpoint)`
- 群组中间件：只要在群组路由上注册中间件函数即可。

中间件实践

为了演示中间件的功能，这里给 apiserver 新增两个功能：

1. 在请求和返回的 Header 中插入 X-Request-Id（X-

Request-Id 值为 32 位的 UUID，用于唯一标识一次 HTTP 请求）
2. 日志记录每一个收到的请求

插入 X-Request-Id

首先需要实现 middleware.RequestId() 中间件，在 router/middleware 目录下新建一个 Go 源文件 requestid.go，内容为（详见 [demo08/router/middleware/requestid.go](https://github.com/lexkong/apiserver_demos/blob/master/

```
package middleware

import (
    "github.com/gin-gonic/gin"
    "github.com/satori/go.uuid"
)

func RequestId() gin.HandlerFunc {
    return func(c *gin.Context) {
        // Check for incoming header, use it if
exists
        requestId := c.Request.Header.Get("X-
Request-Id")

        // Create request id with UUID4
        if requestId == "" {
            u4, _ := uuid.NewV4()
            requestId = u4.String()
        }

        // Expose it for use in the application
        c.Set("X-Request-Id", requestId)

        // Set X-Request-Id header
        c.Writer.Header().Set("X-Request-Id",
requestId)
        c.Next()
    }
}
```

该中间件调用 github.com/satori/go.uuid 包生成一个 32 位的 UUID，并通过 c.Writer.Header().Set("X-Request-Id", requestId) 设置在返回包的 Header 中。

该中间件是个全局中间件，需要在 main 函数中通过 g.Use() 函数加载：

```
func main() {
    ...
    // Routes.
    router.Load(
        // Cores.
        g,

        // Middlwares.
        middleware.RequestId(),
    )
    ...
}
```

main 函数调用 router.Load()，函数 router.Load() 最终调用 g.Use() 加载该中间件。

日志记录请求

同样，需要先实现日志请求中间件 middleware.Logging()，然后在 main 函数中通过 g.Use() 加载该中间件：

```
func main() {
    ...
    // Routes.
    router.Load(
        // Cores.
        g,

        // Middlwares.
        middleware.Logging(),
    )
    ...
}
```

middleware.Logging() 实现稍微复杂点，读者可以直接参考源码实现：[demo08/router/middleware/logging.go (https://github.com/lexkong/apiserver_demos/blob/master/](https://github.com/lexkong/apiserver_demos/blob/master/)

这里有几点需要说明：

1. 该中间件需要截获 HTTP 的请求信息，然后打印请求信息，因为 HTTP 的请求 Body，在读取过后会被置空，所以这里读取完后会重新赋值：

```
var bodyBytes []byte
if c.Request.Body != nil {
    bodyBytes, _ = ioutil.ReadAll(c.Request.Body)
}

// Restore the io.ReadCloser to its original
state
c.Request.Body =
ioutil.NopCloser(bytes.NewBuffer(bodyBytes))
```

2. 截获 HTTP 的 Response 更麻烦些，原理是重定向 HTTP 的

Response 到指定的 IO 流，详见源码文件。
3. 截获 HTTP 的 Request 和 Response 后，就可以获取需要的信息，最终程序通过 log.Infof() 记录 HTTP 的请求信息。
4. 该中间件只记录业务请求，比如 /v1/user 和 /login 路径。

编译并测试

1. 下载 apiserver_demos 源码包（如前面已经下载过，请忽略此步骤）

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

2. 将 apiserver_demos/demo08 复制为 $GOPATH/src/apiserver

```
$ cp -a apiserver_demos/demo08
$GOPATH/src/apiserver
```

3. 在 apiserver 目录下编译源码

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

测试 middleware.RequestId() 中间件

发送 HTTP 请求 —— 查询用户列表：

```
[api@centos apiserver]$ curl -v -XGET -H "Content-Type: application/json" http://127.0.0.1:8080/v1/user
* About to connect() to 127.0.0.1 port 8080 (#0)
*   Trying 127.0.0.1... connected
* Connected to 127.0.0.1 (127.0.0.1) port 8080 (#0)
> GET /v1/user HTTP/1.1
> User-Agent: curl/7.19.7 (x86_64-redhat-linux-gnu) libcurl/7.19.7 NSS/3.13.1.0 zlib/1.2.3 libidn/1.18 libssh2/1.2.2
> Host: 127.0.0.1:8080
> Accept: */*
> Content-Type: application/json
>
< HTTP/1.1 200 OK
< Access-Control-Allow-Origin: *
< Cache-Control: no-cache, no-store, max-age=0, must-revalidate, value
< Content-Type: application/json; charset=utf-8
< Expires: Thu, 01 Jan 1970 00:00:00 GMT
< Last-Modified: Wed, 06 Jun 2018 11:02:26 GMT
< X-Content-Type-Options: nosniff
< X-Frame-Options: DENY
< X-Request-Id: 1f8b1ae2-8009-4921-b354-86f25022dfa0
< X-Xss-Protection: 1; mode=block
< Date: Wed, 06 Jun 2018 11:02:26 GMT
< Content-Length: 461
<
* Connection #0 to host 127.0.0.1 left intact
* Closing connection #0
{"code":0,"message":"OK","data":{"totalCount":2,"userList":[{"id":6,"username":"user3","sayHello":"Hello ZmA2GtSmR","password":"$2a$10$10cGD76PLgrYcrdYshyKJO2GIb/79sMRfixfc
kH2N.pPqtHB7ya0C","createdAt":"2018-06-05 21:25:53","updatedAt":"2018-06-05 21:25:53"},{"id":0,"username":"admin","sayHello":"Hello WiA2MtSigz","password":"$2a$10$veGcArz47
VGj7l9xN7g2iuT9TF21jLI1YGXarGzvARNdnt4inC9PG","createdAt":"2018-05-28 00:25:33","updatedAt":"2018-05-28 00:25:33"}]}}[api@centos apiserver]$
```

可以看到，HTTP 返回的 Header 有 32 位的 UUID：X-Request-Id：1f8b1ae2-8009-4921-b354-86f25022dfa0。

测试 middleware.Logging() 中间件

在 API 日志中，可以看到有 HTTP 请求记录：

```
[api@centos apiserver]$./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] POST   /login                  --> apiserver/handler/user.Login (7 handlers)
[GIN-debug] POST   /v1/user                --> apiserver/handler/user.Create (7 handlers)
[GIN-debug] DELETE /v1/user/:id            --> apiserver/handler/user.Delete (7 handlers)
[GIN-debug] PUT    /v1/user/:id            --> apiserver/handler/user.Update (7 handlers)
[GIN-debug] GET    /v1/user                --> apiserver/handler/user.List (7 handlers)
[GIN-debug] GET    /v1/user/:username      --> apiserver/handler/user.Get (7 handlers)
[GIN-debug] GET    /sd/health              --> apiserver/handler/sd.HealthCheck (7 handlers)
[GIN-debug] GET    /sd/disk                --> apiserver/handler/sd.DiskCheck (7 handlers)
[GIN-debug] GET    /sd/cpu                 --> apiserver/handler/sd.CPUCheck (7 handlers)
[GIN-debug] GET    /sd/ram                 --> apiserver/handler/sd.RAMCheck (7 handlers)
{"level":"INFO","timestamp":"2018-06-06 19:00:45.147","file":"apiserver/main.go:59","msg":"Start to listening the inco
ming requests on http address: :8080"}
{"level":"INFO","timestamp":"2018-06-06 19:00:45.148","file":"apiserver/main.go:56","msg":"The router has been deploye
d successfully."}
{"level":"INFO","timestamp":"2018-06-06 19:01:26.625","file":"middleware/logging.go:78","msg":"1.699376ms    | 127.0.0
.1    | GET /v1/user | {code: 0, message: OK}"}
{"level":"INFO","timestamp":"2018-06-06 19:01:28.417","file":"middleware/logging.go:78","msg":"1.409232ms    | 127.0.0
.1    | GET /v1/user | {code: 0, message: OK}"}
{"level":"INFO","timestamp":"2018-06-06 19:01:44.945","file":"middleware/logging.go:78","msg":"1.312475ms    | 127.0.0
.1    | GET /v1/user | {code: 0, message: OK}"}
{"level":"INFO","timestamp":"2018-06-06 19:02:26.627","file":"middleware/logging.go:78","msg":"2.932538ms    | 127.0.0
.1    | GET /v1/user | {code: 0, message: OK}"}
```

日志记录了 HTTP 请求的如下信息，依次为：

1. 耗时
2. 请求 IP
3. HTTP 方法 HTTP 路径
4. 返回的 Code 和 Message

小结

本小节通过具体实例展示，如何通过 gin 的 middleware 特性来对 HTTP 请求进行必要的逻辑处理。下一小节即是基于 gin 中间件实现的。