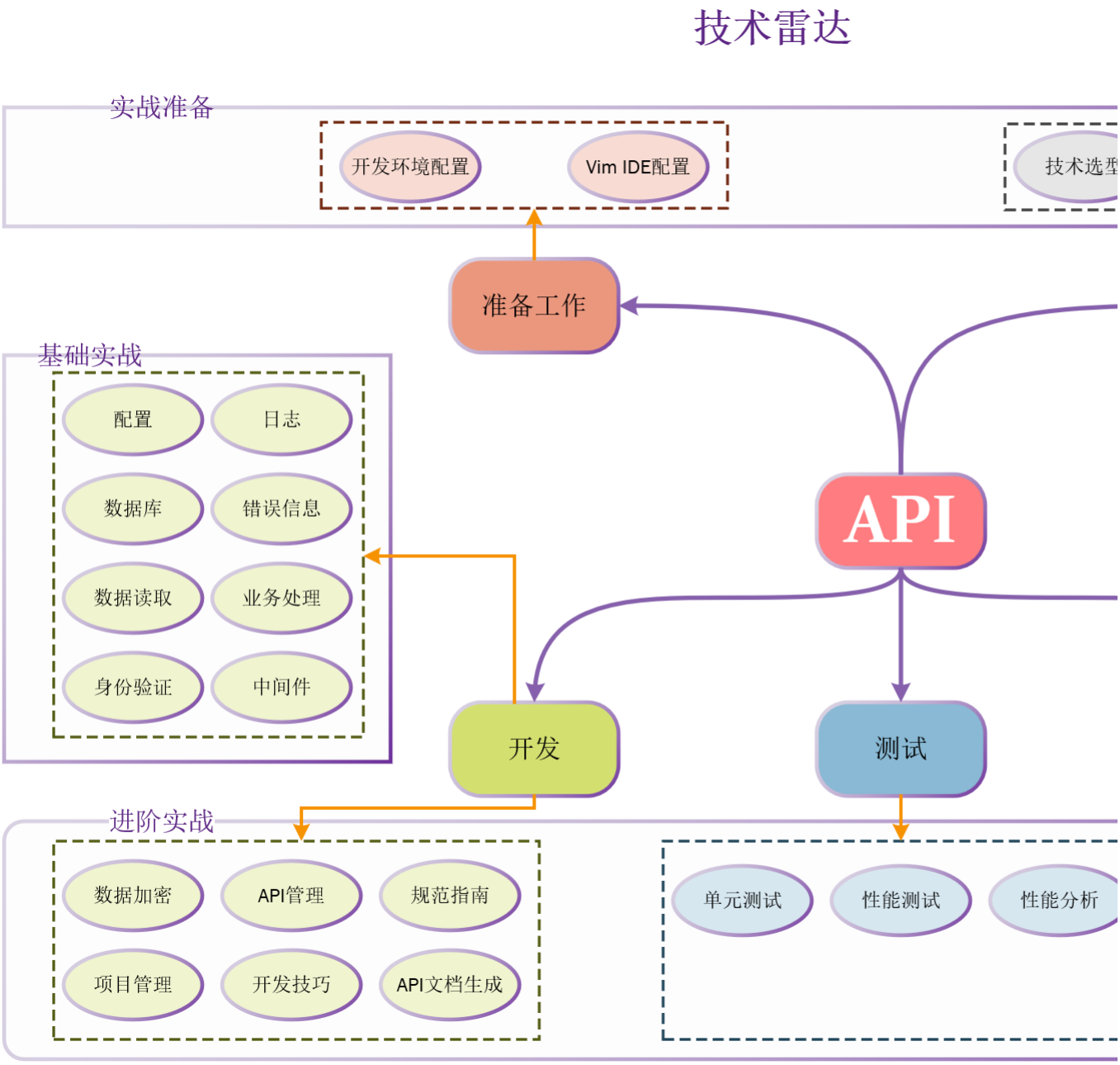
本小册所实现的 API 功能

本小册所实现的功能

本小册通过实现一个账号系统，来演示如何构建一个真实的 API 服务器，构建方法和技术是笔者根据多年的大型 API 服务器开发经验不断优化沉淀而成。通过实战展示了 API 构建过程中各个流程（准备 -> 设计 -> 开发 -> 测试 -> 部署）的实现方法，小册涵盖的内容如下（包括但不限于）：



详细为：

- 准备阶段
  - 如何安装和配置 Go 开发环境
  - 如何安装和配置 Vim IDE
- 设计阶段
  - API 构建技术选型
  - API 基本原理
  - API 规范设计
- 开发阶段
  - 如何读取配置文件
  - 如何管理和记录日志
  - 如何做数据库的 CURD 操作
  - 如何自定义错误 Code
  - 如何读取和返回 HTTP 请求
  - 如何进行业务逻辑开发
  - 如何对请求插入自己的处理逻辑
  - 如何进行 API 身份验证
  - 如何进行 HTTPS 加密
  - 如何用 Makefile 管理 API 源码
  - 如何给 API 命令添加版本功能
  - 如何管理 API 命令
  - 如何生成 Swagger 在线文档
- 测试阶段
  - 如何进行单元测试
  - 如何进行性能测试（函数性能）
  - 如何做性能分析
  - API 性能测试和调优
- 部署阶段
  - 如何用 Nginx 部署 API 服务
  - 如何做 API 高可用

通过以上各功能的介绍，读者可以完整、系统地学习 API 构建方法和技巧，笔者也会在文章中融入自己的开发经验以供读者参考。

账号系统业务功能

本小册为了演示，构建了一个账号系统（后面统称为`apiserver`），功能如下：

- API 服务器状态检查
- 登录用户
- 新增用户
- 删除用户
- 更新用户
- 获取指定用户的详细信息
- 获取用户列表

本小册执行环境

本小册所有的软件安装，运行均是在 CentOS 7.1 系统上执行的。

理论上本小册所构建的 API 可以在所有的 Unix/Linux 系统上编译和运行，小册中的软件安装用的是 `yum` 工具，小册中所列举的 yum 软件理论上可以在 CentOS 6 和 CentOS 7 上直接执行 `yum` 命令安装。

小结

本小节介绍了小册所要实现的 API 功能，以及 API 系统的业务功能，让读者在实战前对小册所要构建的系统有个整体了解，以便于接下来的学习。小册每一节都会提供源码，供读者学习参考。

RESTful API 介绍

什么是 API

API（Application Programming Interface，应用程序编程接口）是一些预先定义的函数或者接口，目的是提供应用程序与开发人员基于某软件或硬件得以访问一组例程的能力，而又无须访问源码，或理解内部工作机制的细节。

要实现一个 API 服务器，首先要考虑两个方面：API 风格和媒体类型。Go 语言中常用的 API 风格是 RPC 和 REST，常用的媒体类型是 JSON、XML 和 Protobuf。在 Go API 开发中常用的组合是 gRPC + Protobuf 和 REST + JSON。

REST 简介

REST 代表表现层状态转移（REpresentational State Transfer），由 Roy Fielding 在他的 论文 中提出。REST 是一种软件架构风格，不是技术框架，REST 有一系列规范，满足这些规范的 API 均可称为 RESTful API。REST 规范中有如下几个核心：

1. REST 中一切实体都被抽象成资源，每个资源有一个唯一的标识 —— URI，所有的行为都应该是在资源上的 CRUD 操作
2. 使用标准的方法来更改资源的状态，常见的操作有：资源的增删改查操作
3. 无状态：这里的无状态是指每个 RESTful API 请求都包含了所有足够完成本次操作的信息，服务器端无须保持 Session

   无状态对于服务端的弹性扩容是很重要的。

REST 风格虽然适用于很多传输协议，但在实际开发中，REST 由于天生和 HTTP 协议相辅相成，因此 HTTP 协议已经成了实现 RESTful API 事实上的标准。在 HTTP 协议中通过 POST、DELETE、PUT、GET 方法来对应 REST 资源的增、删、改、查操作，具体的对应关系如下：

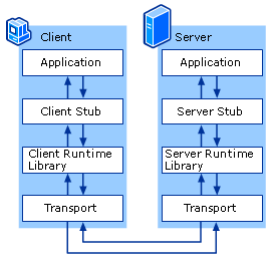
HTTP 方法	行为	URI	示例说明
GET	获取资源列表	/users	获取用户列表
GET	获取一个具体的资源	/users/admin	获取 admin 用户的详细信息
POST	创建一个新的资源	/users	创建一个新用户
PUT	以整体的方式更新一个资源	/users/1	更新 id 为 1 的用户
DELETE	删除服务器上的一个资源	/users/1	删除 id 为 1 的用户

RPC 简介

根据维基百科的定义：远程过程调用（Remote Procedure Call，RPC）是一个计算机通信协议。该协议允许运行于一台计算机的程序调用另一台计算机的子程序，而程序员无须额外地为这个交互作用编程。

通俗来讲，就是服务端实现了一个函数，客户端使用 RPC 框架提供的接口，调用这个函数的实现，并获取返回值。RPC 屏蔽了底层的网络通信细节，使得开发人员无须关注网络编程的细节，而将更多的时间和精力放在业务逻辑本身的实现上，从而提高开发效率。

RPC 的调用过程如下（图片来自 How RPC Works）：



1. Client 通过本地调用，调用 Client Stub
2. Client Stub 将参数打包（也叫 Marshalling）成一个消息，然后发送这个消息
3. Client 所在的 OS 将消息发送给 Server
4. Server 端接收到消息后，将消息传递给 Server Stub
5. Server Stub 将消息解包（也叫 Unmarshalling）得到参数
6. Server Stub 调用服务端的子程序（函数），处理完后，将最终结果按照相反的步骤返回给 Client

   Stub 负责调用参数和返回值的流化（serialization）、参数的打包解包，以及负责网络层的通信。Client 端一般叫 Stub，Server 端一般叫 Skeleton。

REST vs RPC

在做 API 服务器开发时，很多人都会遇到这个问题 —— 选择 REST 还是 RPC。RPC 相比 REST 的优点主要有 3 点：

1. RPC+Protobuf 采用的是 TCP 做传输协议，REST 直接使用 HTTP 做应用层协议，这种区别导致 REST 在调用性能上会比 RPC+Protobuf 低
2. RPC 不像 REST 那样，每一个操作都要抽象成对资源的增删改查，在实际开发中，有很多操作很难抽象成资源，比如登录操作。所以在实际开发中并不能严格按照 REST 规范来写 API，RPC 就不存在这个问题
3. RPC 屏蔽网络细节、易用，和本地调用类似

   这里的易用指的是调用方式上的易用性。在做 RPC 开发时，开发过程很烦琐，需要先写一个 DSL 描述文件，然后用代码生成器生成各种语言代码，当描述文件有更改时，必须重新定义和编译，维护性差。

但是 REST 相较 RPC 也有很多优势：

1. 轻量级，简单易用，维护性和扩展性都比较好
2. REST 相对更规范，更标准，更通用，无论哪种语言都支持 HTTP 协议，可以对接外部很多系统，只要满足 HTTP 调用即可，更适合对外，RPC 会有语言限制，不同语言的 RPC 调用起来很麻烦
3. JSON 格式可读性更强，开发调试都很方便
4. 在开发过程中，如果严格按照 REST 规范来写 API，API 看起来更清晰，更容易被大家理解

   在实际开发中，严格按照 REST 规范来写很难，只能尽可能 RESTful 化。

其实业界普遍采用的做法是，内部系统之间调用用 RPC，对外用 REST，因为内部系统之间可能调用很频繁，需要 RPC 的高性能支撑。对外用 REST 更易理解，更通用些。当然以现有的服务器性能，如果两个系统间调用不是特别频繁，对性能要求不是非常高，以笔者的开发经验来看，REST 的性能完全可以满足。本小册不是讨论微服务，所以不存在微服务之间的高频调用场景，此外 REST 在实际开发

中，能够满足绝大部分的需求场景，所以 RPC 的性能优势可以忽略，相反基于 REST 的其他优势，笔者更倾向于用 REST 来构建 API 服务器，本小册正是用 REST 风格来构建 API 的。

媒体类型选择

媒体类型是独立于平台的类型，设计用于分布式系统间的通信，媒体类型用于传递信息，一个正式的规范定义了这些信息应该如何表示。HTTP 的 REST 能够提供多种不同的响应形式，常见的是 XML 和 JSON。JSON 无论从形式上还是使用方法上都更简单。相比 XML，JSON 的内容更加紧凑，数据展现形式直观易懂，开发测试都非常方便，所以在媒体类型选择上，选择了 JSON 格式，这也是很多大公司所采用的格式。

小结

本小节介绍了软件架构中 API 的实现方式，并简单介绍了相应的技术，通过对比，得出本小册所采用的实现方式：API 风格采用 REST，媒体类型选择 JSON。通过本小节的学习，读者可以了解小册所构建 API 服务器核心技术的选型和原因。

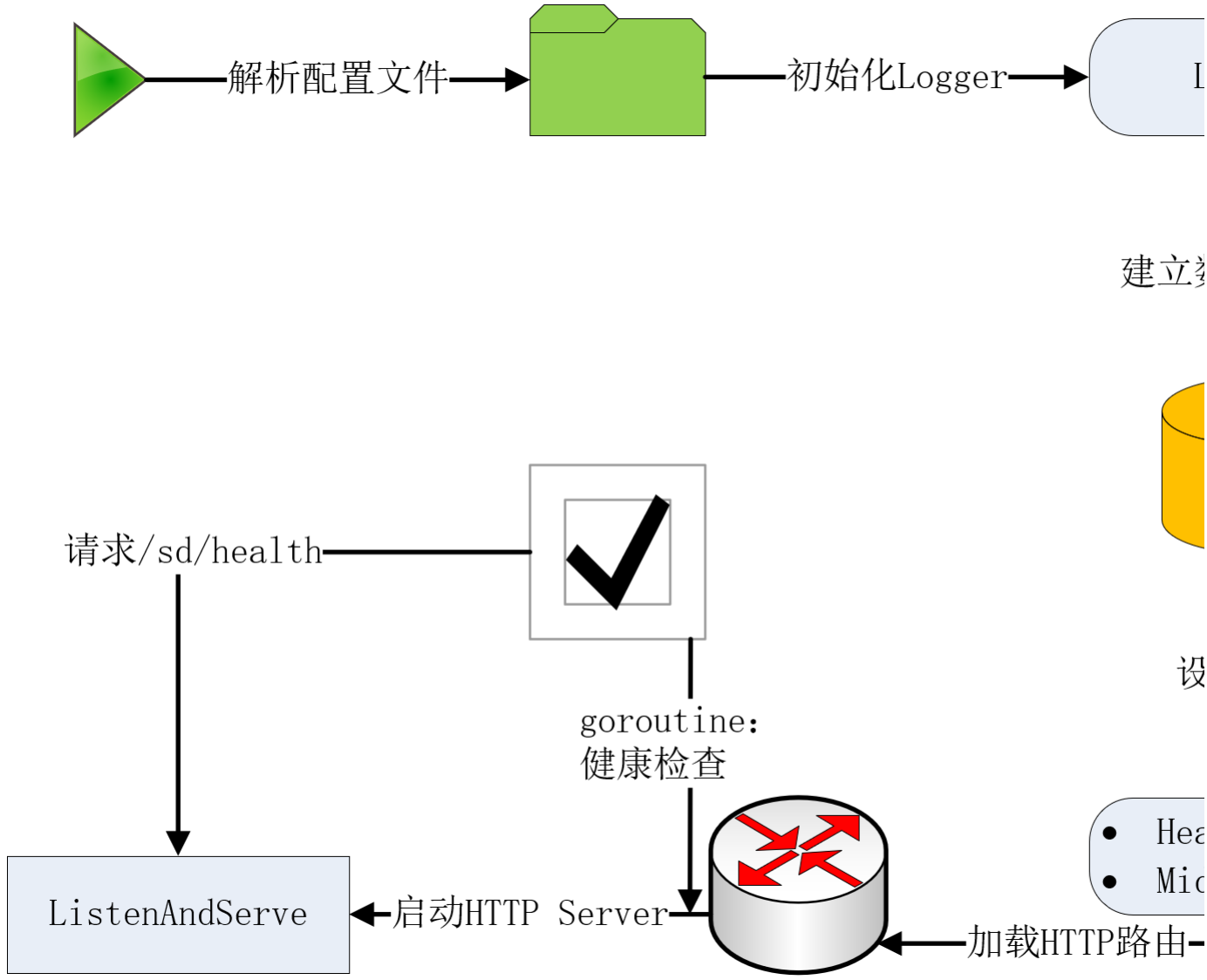
API 流程和代码结构

为了使读者在开始实战之前对 API 开发有个整体的了解，这里选择了两个流程来介绍：

- HTTP API 服务器启动流程
- HTTP 请求处理流程

本小节也提前给出了程序代码结构图，让读者从宏观上了解将要构建的 API 服务器的功能。

HTTP API 服务器启动流程



如上图，在启动一个 API 命令后，API 命令会首先加载配置文件，根据配置做后面的处理工作。通常会将日志相关的配置记录在配置文件中，在解析完配置文件后，就可以加载日志包初始化函数，来初始化日志实例，供后面的程序调用。接下来会初始化数据库实例，建立数据库连接，供后面对数据库的 CRUD 操作使用。在建立完数据库连接后，需要设置 HTTP，通常包括 3 方面的设置：

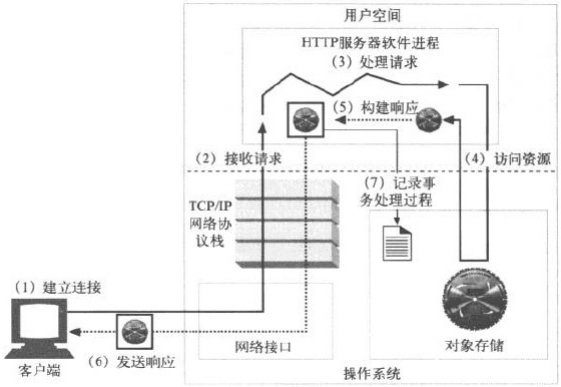
1. 设置 Header
2. 注册路由
3. 注册中间件

之后会调用 `net/http` 包的 `ListenAndServe()` 方法启动 HTTP 服务器。

在启动 HTTP 端口之前，程序会 go 一个协程，来ping HTTP 服务器的 `/sd/health` 接口，如果程序成功启动，ping 协程在 timeout 之前会成功返回，如果程序启动失败，则 ping 协程最终会 timeout，并终止整个程序。

> 解析配置文件、初始化 Log 、初始化数据库的顺序根据自己的喜好和需求来排即可。

HTTP 请求处理流程



一次完整的 HTTP 请求处理流程如上图所示。（图片出自《HTTP 权威指南》，推荐想全面理解 HTTP 的读者阅读此书。）

1. 建立连接

客户端发送 HTTP 请求后，服务器会根据域名进行域名解析，就是将网站名称转变成 IP 地址：localhost -> 127.0.0.1，Linux hosts文件、DNS 域名解析等可以实现这种功能。之后通过发起 TCP 的三次握手建立连接。TCP 三次连接请参考 TCP 三次握手详解及释放连接过程，建立连接之后就可以发送 HTTP 请求了。

2. 接收请求

HTTP 服务器软件进程，这里指的是 API 服务器，在接收到请求之后，首先根据 HTTP 请求行的信息来解析到 HTTP 方法和路径，在上图所示的报文中，方法是 `GET`，路径是 `/index.html`，之后根据 API 服务器注册的路由信息（大概可以理解为：HTTP 方法 + 路径和具体处理函数的映射）找到具体的处理函数。

3. 处理请求

在接收到请求之后，API 通常会解析 HTTP 请求报文获取请求头和消息体，然后根据这些信息进行相应的业务处理，HTTP 框架一般都有自带的解析函数，只需要输入 HTTP 请求报文，就可以解析到需要的请求头和消息体。通常情况下，业务逻辑处理可以分为两种：包含对数据库的操作和不包含对数据的操作。大型系统中通常两种都会有：

1. 包含对数据库的操作：需要访问数据库（增删改查），然后获取指定的数据，对数据处理后构建指定的响应结构体，返回响应包。数据库通常用的是 MySQL，因为免费，功能和性能也都能满足企业级应用的要求。
2. 不包含对数据库的操作：进行业务逻辑处理后，构建指定的响应结构体，返回响应包。

4. 记录事务处理过程

在业务逻辑处理过程中，需要记录一些关键信息，方便后期 Debug 用。在 Go 中有各种各样的日志包可以用来记录这些信息。

HTTP 请求和响应格式介绍

一个 HTTP 请求报文由请求行（request line）、请求头部（header）、空行和请求数据四部分组成，下图是请求报文的一般格式。

```
POST https://admin.cloud.tyk.io/api/users HTTP/1.1
Host: admin.cloud.tyk.io
Connection: keep-alive
Content-Length: 151
Origin: https://admin.cloud.tyk.io
User-Agent: Mozilla/5.0 (Windows NT 10.0; WOW64) AppleWeb
Content-Type: application/json; charset=utf-8
Accept: */*
Accept-Encoding: gzip, deflate, br
Accept-Language: en-US,en;q=0.9,es;q=0.8,zh-TW;q=0.7,zh;q

{"first_name":"Lex","last_name":"Lex","email_address":"le
ssions":{"IsAdmin":"admin"},"password":"123456"}
```

1

请求行

请求行又分为 3 部分：
1. 请求的方式 POST
2. 请求的资源路径
3. 请求的协议和版本号

2

请求头部

Host: 请求的主机名
Connection: 连接状态保存时间
Content-Length: 请求体长度
User-Agent: 用户代理（浏览器）信息
Content-Type: 提交的内容格式
Accept: 告诉服务器，客户端可以接收
Accept-Encoding: 告诉服务器客户端
Accept-Language: 告诉服务器客户端

- 第一行必须是一个请求行（request line），用来说明请求类型、要访问的资源以及所使用的 HTTP 版本
- 紧接着是一个头部（header）小节，用来说明服务器要使用的附加信息
- 之后是一个空行
- 再后面可以添加任意的其他数据（称之为主体：body）

HTTP 响应格式跟请求格式类似，也是由 4 个部分组成：状态行、消息报头、空行和响应数据。

目录结构

```
├── admin.sh                # 进程的start|stop|status|restart控制文件
├── conf                    # 配置文件统一存放目录
│   ├── config.yaml         # 配置文件
│   ├── server.crt          # TLS配置文件
│   └── server.key
├── config                  # 专门用来处理配置和配置文件的Go package
│   └── config.go
├── db.sql                  # 在部署新环境时，可以登录MySQL客户端，执行source db.sql创建数据库和表
├── docs                    # swagger文档，执行 swag init 生成的
│   ├── docs.go
│   └── swagger
│       ├── swagger.json
│       └── swagger.yaml
├── handler                 # 类似MVC架构中的C，用来读取输入，并将处理流程转发给实际的处理函数，最后返回结果
│   ├── handler.go
│   ├── sd                  # 健康检查handler
│   │   └── check.go
│   └── user                # 核心：用户业务逻辑handler
│       ├── create.go       # 新增用户
│       ├── delete.go       # 删除用户
│       ├── get.go          # 获取指定的用户信息
│       ├── list.go         # 查询用户列表
│       ├── login.go        # 用户登录
│       ├── update.go       # 更新用户
│       └── user.go         # 存放用户handler公用的函数、结构体等
├── main.go                 # Go程序唯一入口
├── Makefile                # Makefile文件，一般大型软件系统都是采用make来作为编译工具
├── model                   # 数据库相关的操作统一放在这里，包括数据库初始化和对表的增删改查
│   ├── init.go             # 初始化和连接数据库
│   ├── model.go            # 存放一些公用的go struct
│   └── user.go             # 用户相关的数据库CURD操作
├── pkg                     # 引用的包
│   ├── auth                # 认证包
│   │   └── auth.go
│   ├── constvar            # 常量统一存放位置
│   │   └── constvar.go
│   ├── errno               # 错误码存放位置
│   │   ├── code.go
│   │   └── errno.go
│   ├── token
```

```
|   |   └── token.go
|   └── version                 # 版本包
|       ├── base.go
|       ├── doc.go
|       └── version.go
├── README.md                   # API目录README
├── router                      # 路由相关处理
|   ├── middleware              # API服务器用的是Gin Web框架，Gin中间件存放位置
|   |   ├── auth.go
|   |   ├── header.go
|   |   ├── logging.go
|   |   └── requestid.go
|   └── router.go
├── service                     # 实际业务处理函数存放位置
|   └── service.go
├── util                        # 工具类函数存放目录
|   ├── util.go
|   └── util_test.go
└── vendor                        # vendor目录用来管理依赖包
    ├── github.com
    ├── golang.org
    ├── gopkg.in
    └── vendor.json
```

Go API 项目中，一般都会包括这些功能项：Makefile 文件、配置文件目录、RESTful API 服务器的 handler 目录、model 目录、工具类目录、vendor 目录，以及实际处理业务逻辑函数所存放的 service 目录。这些都在上述的代码结构中有列出，新加功能时将代码放入对应功能的目录/文件中，可以使整个项目代码结构更加清晰，非常有利于后期的查找和维护。

小结

本小节通过介绍 API 服务器启动流程和 HTTP 请求处理流程，来让读者对 API 服务器中的关键流程有个宏观的了解，更好地理解 API 服务器是如何工作的。API 服务器源码结构也非常重要，一个好的源码结构通常能让逻辑更加清晰，编写更加顺畅，后期维护更加容易，本小册介绍了笔者倾向的源码组织结构，供读者参考。

Go API 开发环境配置

Go 命令安装

Go 有多种安装方式，比如 Go 源码安装、Go 标准包安装、第三方工具（yum、apt-get 等）安装。本小册 API 运行在 Linux 服务器上，选择通过标准包来安装 Go 编译环境。Go 提供了每个平台打好包的一键安装，这些包默认会安装到如下目录：`/usr/local/go`。当然你可以改变它们的安装位置，但是改变之后你必须在你的环境变量中设置如下两个环境变量：

- GOROOT：GOROOT 就是 Go 的安装路径
- GOPATH：GOPATH 是作为编译后二进制的存放目的地和 import 包时的搜索路径

假定你想要安装 Go 的目录为 `$GO_INSTALL_DIR`，后面替换为相应的目录路径，安装步骤如下。

1. 下载安装包

安装包下载地址为 golang.org，如果打不开可以使用这个地址：golang.google.cn。

Linux 版本选择 goxxxxx.linux-amd64.tar.gz 格式的安装包，这里在 Linux 服务器上直接用 `wget` 命令下载：

```
$ wget https://dl.google.com/go/go1.10.2.linux-amd64.tar.gz
```

2. 设置安装目录

```
$ export GO_INSTALL_DIR=$HOME
```

这里我们安装到用户主目录下。

3. 解压 Go 安装包

```
$ tar -xvzf go1.10.2.linux-amd64.tar.gz -C $GO_INSTALL_DIR
```

4. 设置环境变量

```
$ export GO_INSTALL_DIR=$HOME
$ export GOROOT=$GO_INSTALL_DIR/go
$ export GOPATH=$HOME/mygo
$ export PATH=$GOPATH/bin:$PATH:$GO_INSTALL_DIR/go/bin
```

如果不想每次登录系统都设置一次环境变量，可以将上面 4 行追加到 `$HOME/.bashrc` 文件中。

5. 执行 `go version` 检查 Go 是否成功安装

```
$ go version
go version go1.10.2 linux/amd64
```

看到 `go version` 命令输出 go 版本号 `go1.10.2 linux/amd64`，说明 go 命令安装成功。

6. 创建 `$GOPATH/src` 目录

`$GOPATH/src` 是 Go 源码存放的目录，所以在正式开始编码前要先确保 `$GOPATH/src` 目录存在，执行命令：

```
$ mkdir -p $GOPATH/src
```

Vim 配置

因为 Vim 是 Linux 下开发的最基本工具，为了通用这里基于 Vim 来配置开发环境。如果要配置一个 Vim IDE 有很多步骤需要一步一步去做，笔者调研了很多 Go vim ide 的配置方法，编写了一个安装工具，这里直接用该工具来配置，具体配置步骤如下。

1. 下载 Vim 配置工具

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

2. 进入 lexVim 目录，下载 go ide 需要的二进制文件：

```
$ cd lexVim
$ git clone https://github.com/lexkong/vim-go-ide-bin
```

都是二进制文件，大概有 141MB，请耐心等待 :-)

3. 启动安装脚本：

```
$ ./start_vim.sh
```

启动后，会进入一个交互环境，依次输入：`1 -> yourname -> youremail@qq.com`，脚本最后输出 `this vim config is success !`说明安装成功。很简单，只需 3 个选择即可安装成功，配置 IDE so easy。

Vim IDE 常用功能

在 Go 项目开发中最常用的功能是：

- `gd` 或者`ctrl + ]` 跳转到对应的函数定义处
- `ctrl + t` 标签退栈
- `ctrl + o` 跳转到前一个位置
- `<F4>` 最近文件列表
- `<F5>` 在 Vim 的上面打开文件查找窗口
- `<F9>` 生成供函数跳转的 tag
- `<F2>` 打开目录窗口，再按会关闭目录窗口
- `<F6>` 添加函数注释

在代码间跳来跳去，将光标放在某个函数调用上，按 `ctl + ]` 就会跳到函数的定义处，按 `ctrl + o` 就会跳回来。

更多 Go vim ide 功能请参考 Vim IDE 功能。

小结

“工欲善其事，必先利其器。”在开始 Go 开发之前，需要安装基本的 Go 编译工具，设置基本的环境变量。如果有一个顺手的开发工具就更好了。该小节向读者介绍了：

1. 如何安装 Go 编译环境
2. 如何配置 Vim IDE

开头的这 4 小节介绍了 API 开发的一些基本的知识，并做了开发前的准备工作，接下来开始 API 开发实战，一步一步教你构建一个账号管理的 API 服务，满满的干货等你来 Get。

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

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。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）。关于 Gin 的更多介绍可以参考 Golang 微框架 Gin 简介。

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）：

```
    "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.RAMCheck` 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。

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。

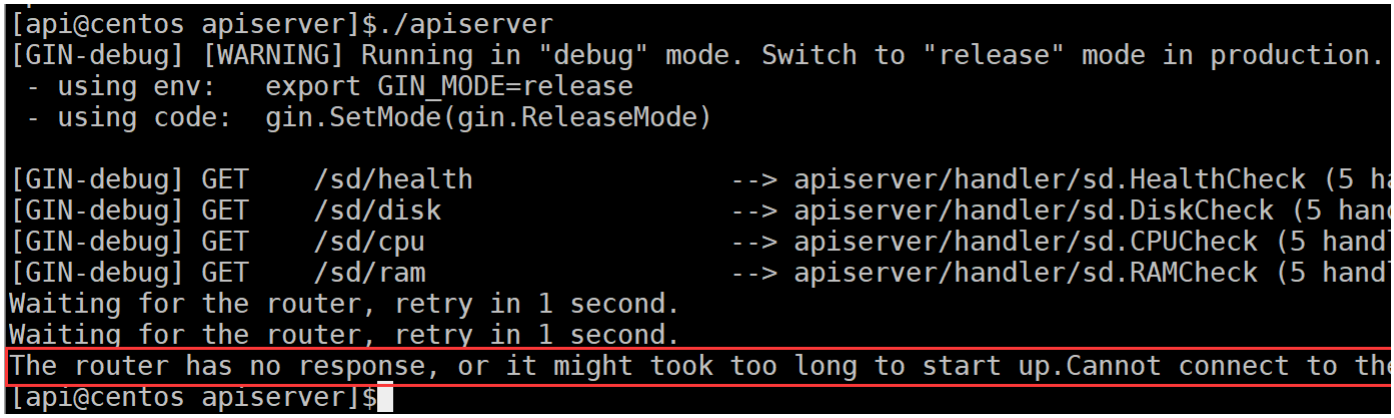
API 服务器健康状态自检

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。

配置文件读取

本节核心内容

- 介绍 apiserver 所采用的配置解决方案
- 介绍如何配置 apiserver 并读取其配置，以及配置的高级用法

> 本小节源码下载路径：demo02
>
> 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 demo01 来开发的。

Viper 简介

Viper 是国外大神 **spf13** 编写的开源配置解决方案，具有如下特性:

- 设置默认值
- 可以读取如下格式的配置文件：JSON、TOML、YAML、HCL
- 监控配置文件改动，并热加载配置文件
- 从环境变量读取配置
- 从远程配置中心读取配置（etcd/consul），并监控变动
- 从命令行 flag 读取配置
- 从缓存中读取配置
- 支持直接设置配置项的值

Viper 配置读取顺序:

- `viper.Set()` 所设置的值
- 命令行 flag
- 环境变量
- 配置文件
- 配置中心：etcd/consul
- 默认值

从上面这些特性来看，Viper 毫无疑问是非常强大的，而且 Viper 用起来也很方便，在初始化配置文件后，读取配置只需要调用 `viper.GetString()`、`viper.GetInt()` 和 `viper.GetBool()` 等函数即可。

Viper 也可以非常方便地读取多个层级的配置，比如这样一个 YAML 格式的配置:

```
common:
  database:
    name: test
    host: 127.0.0.1
```

如果要读取 host 配置，执行 `viper.GetString("common.database.host")` 即可。

apiserver 采用 YAML 格式的配置文件，采用 YAML 格式，是因为 YAML 表达的格式更丰富，可读性更强。

初始化配置

主函数中增加配置初始化入口

```
package main

import (
 "errors"
 "log"
 "net/http"
 "time"

 "apiserver/config"

  ...

 "github.com/spf13/pflag"
)

var (
 cfg = pflag.StringP("config", "c", "", "apiserver config file path.")
)

func main() {
 pflag.Parse()

 // init config
 if err := config.Init(*cfg); err != nil {
  panic(err)
 }

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

 ...
}
```

在 `main` 函数中增加了 `config.Init(*cfg)` 调用，用来初始化配置，`cfg` 变量值从命令行 flag 传入，可以传值，比如 `./apiserver -c config.yaml`，也可以为空，如果为空会默认读取 `conf/config.yaml`。

解析配置

`main` 函数通过 `config.Init` 函数来解析并 watch 配置文件（函数路径：`config/config.go`），config.go 源码为:

```
package config

import (
 "log"
 "strings"

 "github.com/fsnotify/fsnotify"
 "github.com/spf13/viper"
)

type Config struct {
 Name string
}

func Init(cfg string) error {
 c := Config {
  Name: cfg,
 }

 // 初始化配置文件
 if err := c.initConfig(); err != nil {
  return err
 }

 // 监控配置文件变化并热加载程序
 c.watchConfig()

 return nil
}

func (c *Config) initConfig() error {
 if c.Name != "" {
  viper.SetConfigFile(c.Name) // 如果指定了配置文件，则解析指定的配置文件
 } else {
  viper.AddConfigPath("conf") // 如果没有指定配置文件，则解析默认的配置文件
  viper.SetConfigName("config")
 }
 viper.SetConfigType("yaml") // 设置配置文件格式为YAML
 viper.AutomaticEnv() // 读取匹配的环境变量
 viper.SetEnvPrefix("APISERVER") // 读取环境变量的前缀为APISERVER
 replacer := strings.NewReplacer(".", "_")
 viper.SetEnvKeyReplacer(replacer)
 if err := viper.ReadInConfig(); err != nil { // viper解析配置文件
  return err
 }

 return nil
}

// 监控配置文件变化并热加载程序
func (c *Config) watchConfig() {
 viper.WatchConfig()
 viper.OnConfigChange(func(e fsnotify.Event) {
  log.Printf("Config file changed: %s", e.Name)
 })
}
```

`config.Init()` 通过 `initConfig()` 函数来解析配置文件，通过 `watchConfig()` 函数来 watch 配置文件，两个函数解析如下:

1. `func (c *Config) initConfig() error`

设置并解析配置文件。如果指定了配置文件 `*cfg` 不为空，则解析指定的配置文件，否则解析默认的配置文件 `conf/config.yaml`。通过指定配置文件可以很方便地连接不同的环境（开发环境、测试环境）并加载不同的配置，方便开发和测试。

通过如下设置

```
viper.AutomaticEnv()
viper.SetEnvPrefix("APISERVER")
replacer := strings.NewReplacer(".", "_")
```

可以使程序读取环境变量，具体效果稍后会演示。

`config.Init` 函数中的 `viper.ReadInConfig()` 函数最终会调用 Viper 解析配置文件。

2. `func (c *Config) watchConfig()`

   通过该函数的 viper 设置，可以使 viper 监控配置文件变更，如有变更则热更新程序。所谓热更新是指：可以不重启 API 进程，使 API 加载最新配置项的值。

配置并读取配置

API 服务器端口号可能经常需要变更，API 服务器启动时间可能会变长，自检程序超时时间需要是可配的（通过设置次数），另外 API 需要根据不同的开发模式（开发、生产、测试）来匹配不同的行为。开发模式也需要是可配置的，这些都可以在配置文件中配置，新建配置文件 `conf/config.yaml`（默认配置文件名字固定为 `config.yaml`），`config.yaml` 的内容为：

```
runmode: debug               # 开发模式, debug, release, test
addr: :6663                  # HTTP绑定端口
name: apiserver              # API Server的名字
url: http://127.0.0.1:6663   # pingServer函数请求的API服务器的ip:port
max_ping_count: 10           # pingServer函数尝试的次数
```

在 `main` 函数中将相应的配置改成从配置文件读取，需要替换的配置见下图中红框部分。

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

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

// pingServer pings the http server to make sure the router is
func pingServer() error {
    for i := 0; i < 2; i++ {
        // Ping the server by sending a GET request to `/health
        resp, err := http.Get("http://127.0.0.1:6662" + "/sd/he
        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.")
}
```

main.go

替换后，代码为：

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

    log.Printf("Start to listening the incoming requests on htt
    log.Printf(http.ListenAndServe(viper.GetString("addr"), g).
}

// pingServer pings the http server to make sure the router is
func pingServer() error {
    for i := 0; i < viper.GetInt("max_ping_count"); i++ {
        // Ping the server by sending a GET request to `/health
        resp, err := http.Get(viper.GetString("url") + "/sd/hea
        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.")
}
```

main.go

另外根据配置文件的 `runmode` 调用 `gin.SetMode` 来设置 gin 的运行模式：

```
func main() {
    pflag.Parse()

    // init config
    if err := config.Init(*cfg); err != nil {
        panic(err)
    }

    // Set gin mode.
    gin.SetMode(viper.GetString("runmode"))

    ....
}
```

gin 有 3 种运行模式：debug、release 和 test，其中 debug 模式会打印很多 debug 信息。

编译并运行

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/demo02` 复制为 `$GOPATH/src/apiserver`

```
$ cp -a apiserver_demos/demo02/ $GOPATH/src/apiserver
```

3. 在 apiserver 目录下编译源码

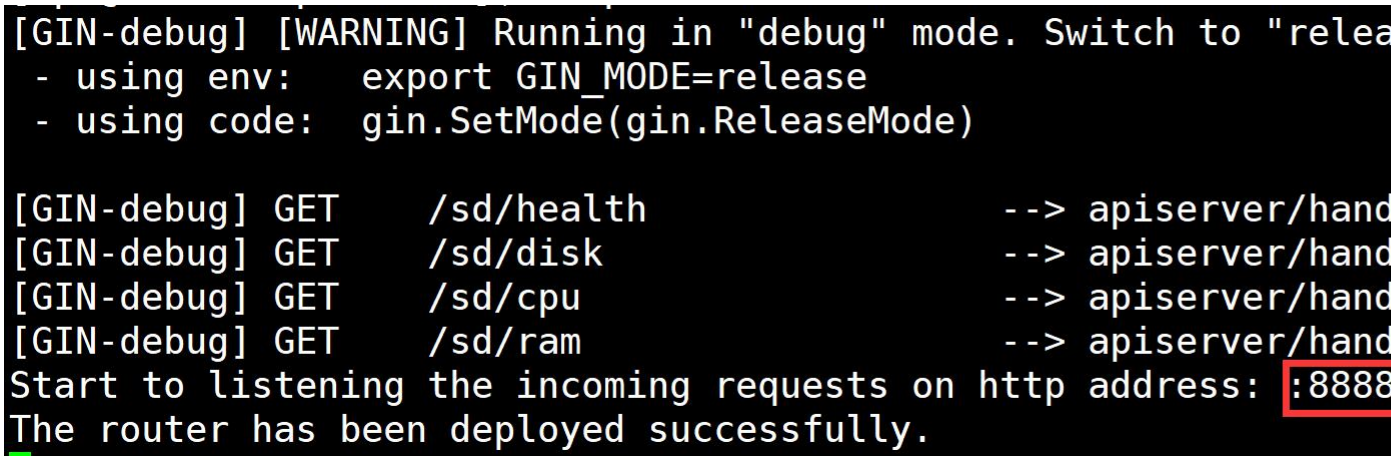
```
$ cd $GOPATH/src/apiserver
$ gofmt -w .
$ go tool vet .
$ go build -v .
```

4. 修改 `conf/config.yaml` 将端口修改为 `8888`，并启动 apiserver

修改后配置文件为：

```
runmode: debug               # 开发模式, debug, release, test
addr: :8888                  # HTTP绑定端口
name: apiserver              # API Server的名字
url: http://127.0.0.1:8888   # pingServer函数请求的API服务器的ip:port
max_ping_count: 10           # pingServer函数try的次数
```

修改后启动 apiserver：



可以看到，启动 apiserver 后端口为配置文件中指定的端口。

Viper 高级用法

从环境变量读取配置

在本节第一部分介绍过，Viper 可以从环境变量读取配置，这是个非常有用的功能。现在越来越多的程序是运行在 Kubernetes 容器集群中的，在 API 服务器迁移到容器集群时，可以直接通过 Kubernetes 来设置环境变量，然后程序读取设置的环境变量来配置 API 服务器。读者不需要了解如何通过 Kubernetes 设置环境变量，只需要知道 Viper 可以直接读取环境变量即可。

例如，通过环境变量来设置 API Server 端口：

```
$ export APISERVER_ADDR=:7777
$ export APISERVER_URL=http://127.0.0.1:7777
$ ./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: :7777
The router has been deployed successfully.
```

从输出可以看到，设置 `APISERVER_ADDR=:7777` 和 `APISERVER_URL=http://127.0.0.1:7777` 后，启动 apiserver，API 服务器的端口变为 `7777`。

环境变量名格式为 `config/config.go` 文件中 `viper.SetEnvPrefix("APISERVER")` 所设置的前缀和配置名称大写，二者用 _ 连接，比如 `APISERVER_RUNMODE`。如果配置项是嵌套的，情况可类推，比如

```
....
max_ping_count: 10          # pingServer函数try的次数
db:
  name: db_apiserver
```

对应的环境变量名为 `APISERVER_DB_NAME`。

热更新

在 `main` 函数中添加如下测试代码（`for {}` 部分，循环打印 `runmode` 的值）：

```
import (
    "fmt"
    ....
)

var (
    cfg = pflag.StringP("config", "c", "", "apiserver config file path.")
)

func main() {
    pflag.Parse()

    // init config
     if err := config.Init(*cfg); err != nil {
     panic(err)
    }

    for {
        fmt.Println(viper.GetString("runmode"))
        time.Sleep(4*time.Second)
    }
    ....
}
```

编译并启动 `apiserver` 后，修改配置文件中 `runmode` 为 `test`，可以看到 `runmode` 的值从 `debug` 变为 `test`：

小结

本小节展示了如何用强大的配置管理工具 Viper 来解析配置文件并读取配置，还演示了 Viper 的高级用法。

记录和管理 API 日志

本节核心内容

- Go 日志包数量众多，功能不同、性能不同，本小册介绍一个笔者认为比较好的日志库，并给出原因
- 介绍如何初始化日志包
- 介绍如何调用日志包
- 介绍如何转存（rotate）日志文件

  本小节源码下载路径：demo03

  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 demo02 来开发的。

日志包介绍

apiserver 所采用的日志包 lexkong/log 是笔者根据开发经验，并调研 GitHub 上的 开源log 包后封装的一个日志包，也是笔者所在项目使用的日志包。它参考华为 paas-lager，做了一些便捷性的改动，功能完全一样，只不过更为便捷。相较于 Go 的其他日志包，该日志包有如下特点：

- 支持日志输出流配置，可以输出到 stdout 或 file，也可以同时输出到 stdout 和 file
- 支持输出为 JSON 或 plaintext 格式
- 支持彩色输出
- 支持 log rotate 功能
- 高性能

初始化日志包

在 **conf/config.yaml** 中添加 **log** 配置

```
runmode: test                 # 开发模式, debug, release, test
addr: :8080                   # HTTP绑定端口
name: apiserver               # API Server的名字
url: http://127.0.0.1:8080    # pingServer函数请求的API服务器的ip:port
max_ping_count: 10            # pingServer函数try的次数
log:
  writers: file,stdout
  logger_level: DEBUG
  logger_file: log/apiserver.log
  log_format_text: false
  rollingPolicy: size
  log_rotate_date: 1
  log_rotate_size: 1024
  log_backup_count: 7
```

在 **config/config.go** 中添加日志初始化代码

```
package config

import (
    ....
    "github.com/lexkong/log"
    ....
)
....
func Init(cfg string) error {
    ....
    // 初始化配置文件
    if err := c.initConfig(); err != nil {
        return err
    }

    // 初始化日志包
    c.initLog()
    ....
}

func (c *Config) initConfig() error {
    ....
}

func (c *Config) initLog() {
    passLagerCfg := log.PassLagerCfg {
        Writers:        viper.GetString("log.writers"),
        LoggerLevel:    viper.GetString("log.logger_level"),
        LoggerFile:     viper.GetString("log.logger_file"),
        LogFormatText:  viper.GetBool("log.log_format_text"),
        RollingPolicy:  viper.GetString("log.rollingPolicy"),
        LogRotateDate:  viper.GetInt("log.log_rotate_date"),
        LogRotateSize:  viper.GetInt("log.log_rotate_size"),
        LogBackupCount: viper.GetInt("log.log_backup_count"),
    }

    log.InitWithConfig(&passLagerCfg)
}

// 监控配置文件变化并热加载程序
func (c *Config) watchConfig() {
    ....
}
```

这里要注意，日志初始化函数 `c.initLog()` 要放在配置初始化函数 `c.initConfig()` 之后，因为日志初始化函数要读取日志相关的配置。`func (c *Config) initLog()` 是日志初始化函数，会设置日志包的各项参数，参数为：

- `writers`：输出位置，有两个可选项 —— file 和 stdout。选择 file 会将日志记录到 `logger_file` 指定的日志文件中，选择 stdout 会将日志输出到标准输出，当然也可以两者同时选择
- `logger_level`：日志级别，DEBUG、INFO、WARN、ERROR、FATAL
- `logger_file`：日志文件
- `log_format_text`：日志的输出格式，JSON 或者 plaintext，`true` 会输出成非 JSON 格式，`false` 会输出成 JSON 格式
- `rollingPolicy`：rotate 依据，可选的有 daily 和 size。如果选 daily 则根据天进行转存，如果是 size 则根据大小进行转存
- `log_rotate_date`：rotate 转存时间，配 合`rollingPolicy: daily` 使用
- `log_rotate_size`：rotate 转存大小，配合 `rollingPolicy: size` 使用
- `log_backup_count`：当日志文件达到转存标准时，log 系统会将该日志文件进行压缩备份，这里指定了备份文件的最大个数

调用日志包

日志初始化好了，将 demo02 中的 log 用 lexkong/log 包来替换。替换前（这里 grep 出了需要替换的行，读者可自行确认替换后的效果）：

```
$ grep log * -R
config/config.go: "log"
config/config.go:  log.Printf("Config file changed: %s", e.Name)
main.go: "log"
main.go:   log.Fatal("The router has no response, or it might took too long to start up.", err)
main.go:  log.Print("The router has been deployed successfully.")
main.go: log.Printf("Start to listening the incoming requests on http address: %s", viper.GetString("addr"))
main.go: log.Printf(http.ListenAndServe(viper.GetString("addr"), g).Error())
main.go:  log.Print("Waiting for the router, retry in 1 second.")
```

替换后的源码文件见 demo03。

编译并运行

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/demo03`复制为 `$GOPATH/src/apiserver`

```
$ cp -a apiserver_demos/demo03/ $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 .
```

4. 启动 apiserver

```
$ ./apiserver
```

启动后，可以看到 apiserver 有 JSON 格式的日志输出：

```
[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] 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)
{"level":"INFO","timestamp":"2018-06-07 16:05:41.541","file":"apiserver/main.go:54","msg":"Start to listening the incoming rec
{"level":"INFO","timestamp":"2018-06-07 16:05:41.542","file":"apiserver/main.go:51","msg":"The router has been deployed succes
```

管理日志文件

这里将日志转存策略设置为 `size`，转存大小设置为 1 MB

```
  rollingPolicy: size
  log_rotate_size: 1
```

并在 `main` 函数中加入测试代码：

```
    // Set gin mode.
    gin.SetMode(viper.GetString("runmode"))

    for {
        log.Info("1111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111")
        time.Sleep(100 * time.Millisecond)
    }

    // Create the Gin engine.
    g := gin.New()
```

启动 apiserver 后发现，在当前目录下创建了 log/apiserver.log 日志文件：

```
$ ls log/
apiserver.log
```

程序运行一段时间后，发现又创建了 zip 文件：

```
$ ls log/
apiserver.log  apiserver.log.20180531134509631.zip
```

该 zip 文件就是当 apiserver.log 大小超过 1MB 后，日志系统将之前的日志压缩成 zip 文件后的文件。

小结

本小节通过具体实例讲解了如何配置、使用和管理日志。

安装 MySQL 并初始化表

本节核心内容

- 如何安装 MySQL 数据库
- 如何创建示例需要的数据库和表

本节主要是为第 12 节「用户业务逻辑处理」做准备工作。

准备工作

安装 MySQL

apiserver 用的是 MySQL，所以首先要确保服务器上安装有 MySQL，执行如下命令来检查是否安装了 MySQL（CentOS 7 上是 mariadb-server，CentOS 6 上是 mysql-server，这里以 CentOS 7 为例）：

```
$ rpm -q mariadb-server
```

如果提示 `package mariadb-server is not installed` 则说明没有安装 MySQL，需要手动安装。如果出现 `mariadb-server-xxx.xxx.xx.el7.x86_64` 则说明已经安装。

安装 MySQL 的步骤为：

1. 安装 MySQL 和 MySQL 客户端

```
$ sudo yum -y install mariadb  mariadb-server
```

2. 启动 MySQL

```
$ sudo systemctl start mariadb
```

3. 设置开机启动

```
$ sudo systemctl enable mariadb
```

4. 设置初始密码

```
$ sudo mysqladmin -u root password root
```

> 因为版权原因，在 CentOS 7 中用的是基于 MySQL fork 的一个开源分支 MariaDB，它的功能和用法跟 MySQL 完全一样。
>
> 如果你的系统之前已经安装过 MySQL，需要在 conf/config.yaml 配置文件中更新配置。

创建示例需要的数据库和表

1. 创建 `db.sql`，内容为：

```
/*!40101 SET @OLD_CHARACTER_SET_CLIENT=@@CHARACTER_SET_CLIENT */;
/*!40101 SET @OLD_CHARACTER_SET_RESULTS=@@CHARACTER_SET_RESULTS */;
/*!40101 SET @OLD_COLLATION_CONNECTION=@@COLLATION_CONNECTION */;
/*!40101 SET NAMES utf8 */;
/*!40103 SET @OLD_TIME_ZONE=@@TIME_ZONE */;
/*!40103 SET TIME_ZONE='+00:00' */;
/*!40014 SET @OLD_UNIQUE_CHECKS=@@UNIQUE_CHECKS, UNIQUE_CHECKS=0 */;
/*!40014 SET @OLD_FOREIGN_KEY_CHECKS=@@FOREIGN_KEY_CHECKS, FOREIGN_KEY_CHECKS=0 */;
/*!40101 SET @OLD_SQL_MODE=@@SQL_MODE, SQL_MODE='NO_AUTO_VALUE_ON_ZERO' */;
/*!40111 SET @OLD_SQL_NOTES=@@SQL_NOTES, SQL_NOTES=0 */;

CREATE DATABASE /*!32312 IF NOT EXISTS*/ `db_apiserver` /*!40100 DEFAULT CHARACTER SET utf8 */;

USE `db_apiserver`;

--
-- Table structure for table `tb_users`
--

DROP TABLE IF EXISTS `tb_users`;
/*!40101 SET @saved_cs_client     = @@character_set_client */;
/*!40101 SET character_set_client = utf8 */;
```

```
CREATE TABLE `tb_users` (
  `id` bigint(20) unsigned NOT NULL AUTO_INCREMENT,
  `username` varchar(255) NOT NULL,
  `password` varchar(255) NOT NULL,
  `createdAt` timestamp NULL DEFAULT NULL,
  `updatedAt` timestamp NULL DEFAULT NULL,
  `deletedAt` timestamp NULL DEFAULT NULL,
  PRIMARY KEY (`id`),
  UNIQUE KEY `username` (`username`),
  KEY `idx_tb_users_deletedAt` (`deletedAt`)
) ENGINE=MyISAM AUTO_INCREMENT=2 DEFAULT CHARSET=utf8;
/*!40101 SET character_set_client = @saved_cs_client */;

--
-- Dumping data for table `tb_users`
--

LOCK TABLES `tb_users` WRITE;
/*!40000 ALTER TABLE `tb_users` DISABLE KEYS */;
INSERT INTO `tb_users` VALUES (0,'admin','$2a$10$veGcArz47VGj719xN7g2iuT9TF21jLI1YGXarGzvARNdnt4inC9PG','2018-05-27 16:25:33','2018-05-27 16:25:33',NULL);
/*!40000 ALTER TABLE `tb_users` ENABLE KEYS */;
UNLOCK TABLES;
/*!40103 SET TIME_ZONE=@OLD_TIME_ZONE */;

/*!40101 SET SQL_MODE=@OLD_SQL_MODE */;
/*!40014 SET FOREIGN_KEY_CHECKS=@OLD_FOREIGN_KEY_CHECKS */;
/*!40014 SET UNIQUE_CHECKS=@OLD_UNIQUE_CHECKS */;
/*!40101 SET CHARACTER_SET_CLIENT=@OLD_CHARACTER_SET_CLIENT */;
/*!40101 SET CHARACTER_SET_RESULTS=@OLD_CHARACTER_SET_RESULTS */;
/*!40101 SET COLLATION_CONNECTION=@OLD_COLLATION_CONNECTION */;
/*!40111 SET SQL_NOTES=@OLD_SQL_NOTES */;

-- Dump completed on 2018-05-28  0:25:41
```

2. 登录 MySQL:

```
$ mysql -uroot -p
```

3. source db.sql

```
mysql> source db.sql
```

可以看到，db.sql 创建了 db_apiserver 数据库和 tb_users 表，并默认添加了一条记录（用户名：admin，密码：admin）：

```
mysql> select * from tb_users \G;
*************************** 1. row ***************************
        id: 0
  username: admin
  password: $2a$10$veGcArz47VGj719xN7g2iuT9TF21jLI1YGXarGzvARNdnt4inC9PG
createdAt: 2018-05-28 00:25:33
updatedAt: 2018-05-28 00:25:33
deletedAt: NULL
1 row in set (0.00 sec)
```

在配置文件中添加数据库配置

API 启动需要连接数据库，所以需要在配置文件 conf/config.yaml 中配置数据库的 IP、端口、用户名、密码和数据库名信息。

```
runmode: debug                 # 开发模式, debug, release, test
addr: :8080                    # HTTP绑定端口
name: apiserver                # API Server的名字
url: http://127.0.0.1:8080     # pingServer函数请求的API服务器的i
max_ping_count: 10             # pingServer函数try的次数
log:
  writers: file,stdout
  logger_level: DEBUG
  logger_file: log/apiserver.log
  log_format_text: false
  rollingPolicy: size
  log_rotate_date: 1
  log_rotate_size: 1
  log_backup_count: 7
db:
  name: db_apiserver
  addr: 127.0.0.1:3306
  username: root
  password: root
docker_db:
  name: db_apiserver
  addr: 127.0.0.1:3306
  username: root
  password: root
```

小结

本小节通过一步步操作，介绍了如何安装 MySQL 数据库，以及如何创建小册示例需要的数据库和表，为后面的小节作好环境准备。

初始化 MySQL 数据库并建立连接

本节核心内容

- Go ORM 数量众多，本小册介绍一个笔者认为比较好的 ORM 包，并给出原因
- 介绍如何初始化数据库
- 介绍如何连接数据库

本小节源码下载路径：demo04

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 demo03 来开发的。

apiserver 用的 ORM 是 GitHub 上 star 数最多的 gorm，相较于其他 ORM，它用起来更方便，更稳定，社区也更活跃。

初始化数据库

在 `model/init.go` 中添加数据初始化代码

因为一个 API 服务器可能需要同时访问多个数据库，为了对多个数据库进行初始化和连接管理，这里定义了一个叫 `Database` 的 struct：

```
type Database struct {
    Self   *gorm.DB
    Docker *gorm.DB
}
```

`Database` 结构体有个 `Init()` 方法用来初始化连接：

```
func (db *Database) Init() {
    DB = &Database {
        Self:   GetSelfDB(),
        Docker: GetDockerDB(),
    }
}
```

`Init()` 函数会调用 `GetSelfDB()` 和 `GetDockerDB()` 方法来同时创建两个 `Database` 的数据库对象。这两个 Get 方法最终都会调用 `func openDB(username, password, addr, name string) *gorm.DB` 方法来建立数据库连接，不同数据库实例传入不同的 username、password、addr 和名字信息，从而建立不同的数据库连接。`openDB` 函数为：

```
func openDB(username, password, addr, name string) *gorm.DB {
    config := fmt.Sprintf("%s:%s@tcp(%s)/%s?charset=utf8&parseTime=%t&loc=%s",
        username,
        password,
        addr,
        name,
        true,
        //"Asia/Shanghai"),
        "Local")

    db, err := gorm.Open("mysql", config)
    if err != nil {
        log.Errorf(err, "Database connection failed. Database name: %s", name)
    }

    // set for db connection
    setupDB(db)

    return db
}
```

可以看到，`openDB()` 最终调用 `gorm.Open()` 来建立一个数据库连接。

完整的 model/init.go 源码文件请参考 demo04/model/init.go。

主函数中增加数据库初始化入口

```
package main

import (
    ...
    "apiserver/model"

    ...
)

...

func main() {
    ...

    // init db
    model.DB.Init()
    defer model.DB.Close()

    ...
}
```

通过 `model.DB.Init()` 来建立数据库连接，通过 `defer model.DB.Close()` 来关闭数据库连接。

编译并运行

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/demo04`复制为`$GOPATH/src/apiserver`

```
$ cp -a apiserver_demos/demo04/ $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 .
```

小结

本小结介绍了如何用 `gorm` 建立数据库连接，为之后的业务逻辑处理作准备。至于具体怎么使用 `gorm` 来进行增删改查等操作，请参考 GORM 指南。

本小节只是介绍了如何初始化数据库，至于怎么对数据库做 CURD 操作，请参考第 12 节：用户业务逻辑处理。

自定义业务错误信息

本节核心内容

- 如何自定义业务自己的错误信息
- 实际开发中是如何处理错误的
- 实际开发中常见的错误类型
- 通过引入新包 `errno` 来实现此功能，会展示该包的如下用法：
  - 如何新建 Err 类型的错误
  - 如何从 Err 类型的错误中获取 code 和 message

本小节源码下载路径：demo05

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 demo04 来开发的。

为什么要定制业务自己的错误码

在实际开发中引入错误码有如下好处：

- 可以非常方便地定位问题和定位代码行（看到错误码知道什么意思，grep 错误码可以定位到错误码所在行）
- 如果 API 对外开放，有个错误码会更专业些
- 错误码包含一定的信息，通过错误码可以判断出错误级别、错误模块和具体错误信息
- 在实际业务开发中，一个条错误信息需要包含两部分内容：直接展示给用户的 `message` 和用于开发人员 debug 的 `error` 。`message` 可能会直接展示给用户，`error` 是用于 debug 的错误信息，可能包含敏感/内部信息，不宜对外展示
- 业务开发过程中，可能需要判断错误是哪种类型以便做相应的逻辑处理，通过定制的错误码很容易做到这点，例如：

```
if err == errno.ErrBind {
    ...
}
```

- Go 中的 HTTP 服务器开发都是引用 `net/http` 包，该包中只有 60 个错误码，基本都是跟 HTTP 请求相关的。在大型系统中，这些错误码完全不够用，而且跟业务没有任何关联，满足不了业务需求。

在 apiserver 中引入错误码

我们通过一个新包 `errno` 来做错误码的定制，详见 demo05/pkg/errno。

```
$ ls pkg/errno/
code.go  errno.go
```

`errno` 包由两个 Go 文件组成：`code.go` 和 `errno.go`。`code.go` 用来统一存自定义的错误码，`code.go` 的代码为：

```
package errno

var (
 // Common errors
 OK                  = &Errno{Code: 0, Message: "OK"}
 InternalServerError = &Errno{Code: 10001, Message: "Internal server error"}
 ErrBind             = &Errno{Code: 10002, Message: "Error occurred while binding the request body to the struct."}

 // user errors
 ErrUserNotFound      = &Errno{Code: 20102, Message: "The user was not found."}
)
```

代码解析

在实际开发中，一个错误类型通常包含两部分：Code 部分，用来唯一标识一个错误；Message 部分，用来展示错误信息，这部分错误信息通常供前端直接展示。这两部分映射在 `errno` 包中即为 `&Errno{Code: 0, Message: "OK"}`。

错误码设计

目前错误码没有一个统一的设计标准，笔者研究了 BAT 和新浪开放平台对外公布的错误码设计，参考新浪开放平台 Error code 的设计，如下是设计说明：

错误返回值格式：

```
{
  "code": 10002,
  "message": "Error occurred while binding the request body to the struct."
}
```

错误代码说明：

1	00	02
服务级错误（1 为系统级错误）	服务模块代码	具体错误代码

- 服务级别错误：1 为系统级错误；2 为普通错误，通常是由用户非法操作引起的
- 服务模块为两位数：一个大型系统的服务模块通常不超过两位数，如果超过，说明这个系统该拆分了
- 错误码为两位数：防止一个模块定制过多的错误码，后期不好维护
- code = 0 说明是正确返回，code > 0 说明是错误返回
- 错误通常包括系统级错误码和服务级错误码
- 建议代码中按服务模块将错误分类
- 错误码均为 >= 0 的数
- 在 apiserver 中 HTTP Code 固定为 `http.StatusOK`，错误码通过 code 来表示。

错误信息处理

通过 `errno.go` 来对自定义的错误进行处理，`errno.go` 的代码为：

```
package errno

import "fmt"

type Errno struct {
 Code    int
 Message string
}

func (err Errno) Error() string {
 return err.Message
}

// Err represents an error
type Err struct {
 Code    int
 Message string
 Err     error
}

func New(errno *Errno, err error) *Err {
 return &Err{Code: errno.Code, Message: errno.Message, Err: err}
}

func (err *Err) Add(message string) error {
 err.Message += " " + message
 return err
}

func (err *Err) Addf(format string, args ...interface{}) error {
 err.Message += " " + fmt.Sprintf(format, args...)
 return err
}

func (err *Err) Error() string {
 return fmt.Sprintf("Err - code: %d, message: %s, error: %s", err.Code, err.Message, err.Err)
}

func IsErrUserNotFound(err error) bool {
 code, _ := DecodeErr(err)
 return code == ErrUserNotFound.Code
}

func DecodeErr(err error) (int, string) {
 if err == nil {
  return OK.Code, OK.Message
 }

 switch typed := err.(type) {
 case *Err:
  return typed.Code, typed.Message
 case *Errno:
  return typed.Code, typed.Message
 default:
 }

 return InternalServerError.Code, err.Error()
}
```

代码解析

`errno.go` 源码文件中有两个核心函数 `New()` 和 `DecodeErr()`，一个用来新建定制的错误，一个用来解析定制的错误，稍后会介绍如何使用。

errno.go 同时也提供了 Add() 和 Addf() 函数，如果想对外展示更多的信息可以调用此函数，使用方法下面有介绍。

错误码实战

上面介绍了错误码的一些知识，这一部分讲开发中是如何使用 errno 包来处理错误信息的。为了演示，我们新增一个创建用户的 API：

1. router/router.go 中添加路由，详见 demo05/router/router.go：

```
func Load(g *gin.Engine, mw ...gin.HandlerFunc) *gin.Engine {
    // Middlewares.
    g.Use(gin.Recovery())
    g.Use(middleware.NoCache)
    g.Use(middleware.Options)
    g.Use(middleware.Secure)
    g.Use(mw...)
    // 404 Handler.
    g.NoRoute(func(c *gin.Context) {
        c.String(http.StatusNotFound, "The incorrect API route.")
    })

    u := g.Group("/v1/user")
    {
        u.POST("", user.Create)
    }

    // 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)
    }

    return g
}
```

router.go

2. handler 目录下增加业务处理函数 handler/user/create.go，详见 demo05/handler/user/create.go。

编译并运行

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/demo05 复制为 $GOPATH/src/apiserver

```
$ cp -a apiserver_demos/demo05/ $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 .
```

测试验证

启动 apiserver：./apiserver

```
$ curl -XPOST -H "Content-Type: application/json" http://127.0.0.1:8080/v1/user

{
  "code": 10002,
  "message": "Error occurred while binding the request body to the struct."
}
```

因为没有传入任何参数，所以返回 errno.ErrBind 错误。

```
$ curl -XPOST -H "Content-Type: application/json" http://127.0.0.1:8080/v1/user -d'{"username":"admin"}'

{
  "code": 10001,
  "message": "password is empty"
}
```

因为没有传入 password，所以返回 fmt.Errorf("password is empty") 错误，该错误信息不是定制的错误类型，errno.DecodeErr(err) 解析时会解析为默认的 errno.InternalServerError 错误，所以返回结果中 code 为 10001，message 为 err.Error()。

```
$ curl -XPOST -H "Content-Type: application/json" http://127.0.0.1:8080/v1/user -d'{"password":"admin"}'

{
  "code": 20102,
  "message": "The user was not found. This is add message."
}
```

因为没有传入 username，所以返回 errno.ErrUserNotFound 错误信息，并通过 Add() 函数在 message 信息后追加了 This is add message. 信息。

通过

```
    if errno.IsErrUserNotFound(err) {
        log.Debug("err type is ErrUserNotFound")
    }
```

演示了如何通过定制错误方便地对比是不是某个错误，在该请求中，apiserver 会输出如下错误：

```
[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   /v1/user                  --> apiserver/handler/user.Create (5 handl
[GIN-debug] GET    /sd/health                --> apiserver/handler/sd.HealthCheck (5 ha
[GIN-debug] GET    /sd/disk                  --> apiserver/handler/sd.DiskCheck (5 hand
[GIN-debug] GET    /sd/cpu                   --> apiserver/handler/sd.CPUCheck (5 handl
[GIN-debug] GET    /sd/ram                   --> apiserver/handler/sd.RAMCheck (5 handl
{"level":"INFO","timestamp":"2018-06-01 13:08:43.515","file":"apiserver/main.go:59","ms
ming requests on http address: :8080"}
{"level":"INFO","timestamp":"2018-06-01 13:08:43.516","file":"apiserver/main.go:56","ms
d successfully."}
{"level":"DEBUG","timestamp":"2018-06-01 13:08:48.719","file":"user/create.go:26","msg
[admin]"}
{"level":"ERROR","timestamp":"2018-06-01 13:08:48.720","file":"user/create.go:29","msg
:"Err - code: 20102, message: The user was not found. This is add message., error: use
.xx.xx"}}
{"level":"DEBUG","timestamp":"2018-06-01 13:08:48.720","file":"user/create.go:33","msg
```

可以看到在后台日志中会输出敏感信息 `username can not found in db: xx.xx.xx.xx`，但是返回给用户的 message （`{"code":20102,"message":"The user was not found. This is add message."}`）不包含这些敏感信息，可以供前端直接对外展示。

```
$ curl -XPOST -H "Content-Type: application/json" http://127.0.0.1:8080/v1/user -d'{"username":"admin","password":"admin"}'

{
  "code": 0,
  "message": "OK"
}
```

如果 `err = nil`，则 `errno.DecodeErr(err)` 会返回成功的 `code: 0` 和 `message: OK`。

> 如果 API 是对外的，错误信息数量有限，则制定错误码非常容易，强烈建议使用错误码。如果是内部系统，特别是庞大的系统，内部错误会非常多，这时候没必要为每一个错误制定错误码，而只需为常见的错误制定错误码，对于普通的错误，系统在处理时会统一作为 `InternalServerError` 处理。

小结

本小节详细介绍了实际开发中是如何处理业务错误信息的，并给出了笔者倾向的错误码规范供读者参考，最后通过大量的实例来展示如何通过 `errno` 包来处理不同场景的错误。

读取和返回 HTTP 请求

本节核心内容

- 如何读取 HTTP 请求数据
- 如何返回数据
- 如何定制业务的返回格式

> 本小节源码下载路径：demo06
>
> 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 demo05 来开发的。

读取和返回参数

在业务开发过程中，需要读取请求参数（消息体和 HTTP Header），经过业务处理后返回指定格式的消息。apiserver 也展示了如何进行参数的读取和返回，下面展示了如何读取和返回参数：

读取 **HTTP** 信息：在 API 开发中需要读取的参数通常为：HTTP Header、路径参数、URL参数、消息体，读取这些参数可以直接使用 `gin` 框架自带的函数：

- `Param()`：返回 URL 的参数值，例如

```
router.GET("/user/:id", func(c *gin.Context) {
    // a GET request to /user/john
    id := c.Param("id") // id == "john"
})
```

- `Query()`：读取 URL 中的地址参数，例如

```
// GET /path?id=1234&name=Manu&value=
c.Query("id") == "1234"
c.Query("name") == "Manu"
c.Query("value") == ""
c.Query("wtf") == ""
```

- `DefaultQuery()`：类似 `Query()`，但是如果 key 不存在，会返回默认值，例如

```
//GET /?name=Manu&lastname=
c.DefaultQuery("name", "unknown") == "Manu"
c.DefaultQuery("id", "none") == "none"
c.DefaultQuery("lastname", "none") == ""
```

- `Bind()`：检查 `Content-Type` 类型，将消息体作为指定的格式解析到 Go struct 变量中。apiserver 采用的媒体类型是 JSON，所以 `Bind()` 是按 JSON 格式解析的。
- `GetHeader()`：获取 HTTP 头。

返回**HTTP**消息：因为要返回指定的格式，apiserver 封装了自己的返回函数，通过统一的返回函数 `SendResponse` 来格式化返回，小节后续部分有详细介绍。

增加返回函数

API 返回入口函数，供所有的服务模块返回时调用，所以这里将入口函数添加在 `handler` 目录下，`handler/handler.go` 的源码为：

```
package handler

import (
 "net/http"

 "apiserver/pkg/errno"

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

type Response struct {
 Code    int         `json:"code"`
 Message string      `json:"message"`
 Data    interface{} `json:"data"`
}
```

```
func SendResponse(c *gin.Context, err error, data interface{}) {
 code, message := errno.DecodeErr(err)

 // always return http.StatusOK
 c.JSON(http.StatusOK, Response{
  Code:    code,
  Message: message,
  Data:    data,
 })
}
```

可以看到返回格式固定为：

```
type Response struct {
 Code    int         `json:"code"`
 Message string      `json:"message"`
 Data    interface{} `json:"data"`
}
```

在返回结构体中，固定有 `Code` 和 `Message` 参数，这两个参数通过函数 `DecodeErr()` 解析 error 类型的变量而来（`DecodeErr()` 在上一节介绍过）。`Data` 域为 `interface{}` 类型，可以根据业务自己的需求来返回，可以是 map、int、string、struct、array 等 Go 语言变量类型。`SendResponse()` 函数通过 `errno.DecodeErr(err)` 来解析出 code 和 message，并填充在 `Response` 结构体中。

在业务处理函数中读取和返回数据

通过改写上一节 `handler/user/create.go` 源文件中的 `Create()` 函数，来演示如何读取和返回数据，改写后的源码为：

```
package user

import (
 "fmt"

 . "apiserver/handler"
 "apiserver/pkg/errno"

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

// Create creates a new user account.
func Create(c *gin.Context) {
 var r CreateRequest
 if err := c.Bind(&r); err != nil {
  SendResponse(c, errno.ErrBind, nil)
  return
 }

 admin2 := c.Param("username")
 log.Infof("URL username: %s", admin2)

 desc := c.Query("desc")
 log.Infof("URL key param desc: %s", desc)

 contentType := c.GetHeader("Content-Type")
 log.Infof("Header Content-Type: %s", contentType)

 log.Debugf("username is: [%s], password is [%s]", r.Username, r.Password)
 if r.Username == "" {
  SendResponse(c, errno.New(errno.ErrUserNotFound, fmt.Errorf("username can not found in db: xx.xx.xx.xx")), nil)
  return
 }

 if r.Password == "" {
  SendResponse(c, fmt.Errorf("password is empty"), nil)
 }

 rsp := CreateResponse{
  Username: r.Username,
 }

 // Show the user information.
 SendResponse(c, nil, rsp)
}
```

这里也需要更新下路由，`router/router.go`（详见 demo06/router/router.go）：

```
u := g.Group("/v1/user")
{
    u.POST("/:username", user.Create)
}
```

增加了/:username

上例展示了如何通过 `Bind()`、`Param()`、`Query()` 和 `GetHeader()` 来获取相应的参数。

根据笔者的研发经验，建议：如果消息体有 JSON 参数需要传递，针对每一个 API 接口定义独立的 go struct 来接收，比如 `CreateRequest` 和 `CreateResponse`，并将这些结构体统一放在一个 Go 文件中，以方便后期维护和修改。这样做可以使代码结构更加规整和清晰，本例统一放在 `handler/user/user.go` 中，源码为：

```
package user

type CreateRequest struct {
 Username string `json:"username"`
 Password string `json:"password"`
}

type CreateResponse struct {
 Username string `json:"username"`
}
```

编译并运行

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/demo06` 复制为 `$GOPATH/src/apiserver`

```
$ cp -a apiserver_demos/demo06/ $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

启动apiserver：`./apiserver`，发送 HTTP 请求：

```
$ curl -XPOST -H "Content-Type: application/json" http://127.0.0.1:8080/v1/user/admin2?desc=test -d'{"username":"admin","password":"admin"}'

{
  "code": 0,
  "message": "OK",
  "data": {
    "username": "admin"
  }
}
```

查看 apiserver 日志：

```
[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   /v1/user/:username        --> apiserver/handler/user.Create (5 hand
[GIN-debug] GET    /sd/health               --> apiserver/handler/sd.HealthCheck (5 h
[GIN-debug] GET    /sd/disk                 --> apiserver/handler/sd.DiskCheck (5 han
[GIN-debug] GET    /sd/cpu                  --> apiserver/handler/sd.CPUCheck (5 hand
[GIN-debug] GET    /sd/ram                  --> apiserver/handler/sd.RAMCheck (5 hand
{"level":"INFO","timestamp":"2018-06-01 15:31:37.316","file":"apiserver/main.go:59","m
ming requests on http address: :8080"}
{"level":"INFO","timestamp":"2018-06-01 15:31:37.316","file":"apiserver/main.go:56","m
d successfully."}
{"level":"INFO","timestamp":"2018-06-01 15:31:41.307","file":"user/create.go:22","msg"
{"level":"INFO","timestamp":"2018-06-01 15:31:41.307","file":"user/create.go:25","msg"
{"level":"INFO","timestamp":"2018-06-01 15:31:41.307","file":"user/create.go:28","msg"
on/json"}
{"level":"DEBUG","timestamp":"2018-06-01 15:31:41.307","file":"user/create.go:30","msg
d is [admin]"}
```

可以看到成功读取了请求中的各类参数。并且 `curl` 命令返回的结果格式为指定的格式：

```
{
  "code": 0,
  "message": "OK",
  "data": {
    "username": "admin"
  }
}
```

小结

本小节介绍了如何进行 HTTP 请求的读取和返回，读取主要用 `gin` 框架自带的函数，返回要统一用函数 `SendResponse`。

用户业务逻辑处理

本节核心内容

这一节是核心小节，讲解如何处理用户业务，这也是 API 的核心功能。本小节会讲解实际开发中需要的一些重要功能点，并根据笔者的开发经验，给出一些建议。功能点包括：

- 各种场景的业务逻辑处理
  - 创建用户
  - 删除用户
  - 更新用户
  - 查询用户列表
  - 查询指定用户的信息
- 数据库的 CURD 操作

本小节源码下载路径：demo07

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 demo06 来开发的。

配置路由信息

需要先在 `router/router.go` 文件中，配置路由信息：

```
func Load(g *gin.Engine, mw ...gin.HandlerFunc) *gin.Engine {
    ...
 // 用户路由设置
 u := g.Group("/v1/user")
 {
  u.POST("", user.Create)         // 创建用户
  u.DELETE("/:id", user.Delete)   // 删除用户
  u.PUT("/:id", user.Update)      // 更新用户
  u.GET("", user.List)            // 用户列表
  u.GET("/:username", user.Get)   // 获取指定用户的详细信息
 }
    ...
 return g
}
```

在 RESTful API 开发中，API 经常会变动，为了兼容老的 API，引入了版本的概念，比如上例中的 `/v1/user`，说明该 API 版本是 `v1`。

很多 RESTful API 最佳实践文章中均建议使用版本控制，笔者这里也建议对 API 使用版本控制。

注册新的错误码

在 `pkg/errno/code.go` 文件中（详见 demo07/pkg/errno/code.go），新增如下错误码：

```
var (
 // Common errors
        ...

 ErrValidation       = &Errno{Code: 20001, Message: "Validation failed."}
 ErrDatabase         = &Errno{Code: 20002, Message: "Database error."}
 ErrToken            = &Errno{Code: 20003, Message: "Error occurred while signing the JSON web token."}

 // user errors
 ErrEncrypt          = &Errno{Code: 20101, Message: "Error occurred while encrypting the user password."}
 ErrTokenInvalid     = &Errno{Code: 20103, Message: "The token was invalid."}
 ErrPasswordIncorrect = &Errno{Code: 20104, Message: "The password was incorrect."}
)
```

新增用户

更新 `handler/user/create.go` 中 `Create()` 的逻辑，更新后的内容见 demo07/handler/user/create.go。

创建用户逻辑：

1. 从 HTTP 消息体获取参数（用户名和密码）
2. 参数校验
3. 加密密码
4. 在数据库中添加数据记录
5. 返回结果（这里是用户名）

从 HTTP 消息体解析参数，前面小节已经介绍了。

参数校验这里用的是 gopkg.in/go-playground/validator.v9 包（详见 go-playground/validator），实际开发过程中，该包可能不能满足校验需求，这时候可在程序中加入自己的校验逻辑，比如在 handler/user/creater.go 中添加校验函数 checkParam：

```
package user

import (
    ...
)

// Create creates a new user account.
func Create(c *gin.Context) {
 log.Info("User Create function called.", lager.Data{"X-Request-Id": util.GetReqID(c)})
 var r CreateRequest
 if err := c.Bind(&r); err != nil {
  SendResponse(c, errno.ErrBind, nil)
  return
 }

 if err := r.checkParam(); err != nil {
  SendResponse(c, err, nil)
  return
 }
        ...
}

func (r *CreateRequest) checkParam() error {
 if r.Username == "" {
  return errno.New(errno.ErrValidation, nil).Add("username is empty.")
 }

 if r.Password == "" {
  return errno.New(errno.ErrValidation, nil).Add("password is empty.")
 }

 return nil
}
```

例子通过 Encrypt() 对密码进行加密：

```
// Encrypt the user password.
func (u *UserModel) Encrypt() (err error) {
    u.Password, err = auth.Encrypt(u.Password)
    return
}
```

Encrypt() 函数引用 auth.Encrypt() 来进行密码加密，具体实现见 demo07/pkg/auth/auth.go。

最后例子通过 u.Create() 函数来向数据库中添加记录，ORM 用的是 gorm，gorm 详细用法请参考 GORM 指南。在 Create() 函数中引用的数据库实例是 DB.Self，该实例在 API 启动之前已经完成初始化。DB 是个全局变量，可以直接引用。

> 在实际开发中，为了安全，数据库中是禁止保存密码的明文信息的，密码需要加密保存。
>
> 笔者将接收和处理相关的 Go 结构体统一放在 handler/user/user.go 文件中，这样可以使程序结构更清晰，功能更聚焦。当然每个人习惯不一样，读者根据自己的习惯放置即可。handler/user/user.go 对 UserInfo 结构体的处理，也出于相同的目的。

删除用户

删除用户代码详见 demo07/handler/user/delete.go。

删除时，首先根据 URL 路径 DELETE http://127.0.0.1/v1/user/1 解析出 id 的值 1，该 id 实际上就是数据库中的 id 索引，调用 model.DeleteUser() 函数删除，函数详见 demo07/model/user.go。

更新用户

更新用户代码详见 demo07/handler/user/update.go。

更新用户逻辑跟创建用户差不多，在更新完数据库字段后，需要指定 gorm model 中的 id 字段的值，因为 gorm 在更新时默认是按照 id 来匹配记录的。通过解析 PUT http://127.0.0.1/v1/user/1 来获取 id。

查询用户列表

查询用户列表代码详见 demo07/handler/user/list.go。

一般在 handler 中主要做解析参数、返回数据操作，简单的逻辑也可以在 handler 中做，像新增用户、删除用户、更新用户，代码量不大，所以也可以放在 handler 中。有些代码量很大的逻辑就不适合放在 handler 中，因为这样会导致 handler 逻辑不是很清晰，这时候实际处理的部分通常放在 service 包中。比如本例的 LisUser() 函数：

```
package user

import (
    "apiserver/service"
    ...
)

// List list the users in the database.
func List(c *gin.Context) {
    ...
    infos, count, err := service.ListUser(r.Username, r.Offset, r.Limit)
    if err != nil {
        SendResponse(c, err, nil)
        return
    }
    ...
}
```

查询一个 REST 资源列表，通常需要做分页，如果不做分页返回的列表过多，会导致 API 响应很慢，前端体验也不好。本例中的查询函数做了分页，收到的请求中传入的 offset 和 limit 参数，分别对应于 MySQL 的 offset 和 limit。

service.ListUser() 函数用来做具体的查询处理，代码详见 demo07/service/service.go。

在 ListUser() 函数中用了 sync 包来做并行查询，以使响应延时更小。在实际开发中，查询数据后，通常需要对数据做一些处理，比如 ListUser() 函数中会对每个用户记录返回一个 sayHello 字段。sayHello 只是简单输出了一个 Hello shortId 字符串，其中 shortId 是通过 util.GenShortId() 来生成的（GenShortId 实现详见 demo07/util/util.go）。像这类操作通常会增加 API 的响应延时，如果列表条目过多，列表中的每个记录都要做一些类似的逻辑处理，这会使得整个 API 延时很高，所以笔者在实际开发中通常会做并行处理。根据笔者经验，效果提升十分明显。

读者应该已经注意到了，在 ListUser() 实现中，有 sync.Mutex 和 IdMap 等部分代码，使用 sync.Mutex 是因为在并发处理中，更新同一个变量为了保证数据一致性，通常需要做锁处理。

使用 IdMap 是因为查询的列表通常需要按时间顺序进行排序，一般数据库查询后的列表已经排过序了，但是为了减少延时，程序中用了并发，这时候会打乱排序，所以通过 IdMap 来记录并发处理前的顺序，处理后再重新复位。

获取指定用户的详细信息

代码详见 demo07/handler/user/get.go。

获取指定用户信息时，首先根据 URL 路径 GET http://127.0.0.1/v1/user/admin 解析出 username 的值 admin，然后调用 model.GetUser() 函数查询该用户的数据库记录并返回，函数详见 demo07/model/user.go。

编译并运行

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/demo07 复制为 $GOPATH/src/apiserver

```
$ cp -a apiserver_demos/demo07/ $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 .
```

创建用户

```
$ curl -XPOST -H "Content-Type: application/json" http://127.0.0.1:8080/v1/user -d'{"username":"kong","password":"kong123"}'

{
  "code": 0,
  "message": "OK",
  "data": {
    "username": "kong"
  }
}
```

查询用户列表

```
$ curl -XGET -H "Content-Type: application/json" http://127.0.0.1:8080/v1/user -d'{"offset": 0, "limit": 20}'

{
  "code": 0,
  "message": "OK",
  "data": {
    "totalCount": 2,
    "userList": [
      {
        "id": 2,
        "username": "kong",
        "sayHello": "Hello qhXO5iIig",
        "password": "$2a$10$vE9jG7loyzstWVwB/QfU3u00Pxb.ye8hFIDvnyw60nHBv/xsJZoUO",
        "createdAt": "2018-06-02 14:47:54",
        "updatedAt": "2018-06-02 14:47:54"
      },
      {
        "id": 0,
        "username": "admin",
        "sayHello": "Hello qhXO5iSmgz",
        "password": "$2a$10$veGcArz47VGj7l9xN7g2iuT9TF21jLI1YGXarGzvARNdnt4inC9PG",
        "createdAt": "2018-05-28 00:25:33",
        "updatedAt": "2018-05-28 00:25:33"
      }
    ]
  }
}
```

可以看到，新增了一个用户 `kong`，数据库 id 索引为 `2`。`admin` 用户是上一节中初始化数据库时初始化的。

> 笔者建议在 API 设计时，对资源列表进行分页。

获取用户详细信息

```
$ curl -XGET -H "Content-Type: application/json" http://127.0.0.1:8080/v1/user/kong

{
  "code": 0,
  "message": "OK",
  "data": {
    "username": "kong",
    "password": "$2a$10$vE9jG7loyzstWVwB/QfU3u00Pxb.ye8hFIDvnyw60nHBv/xsJZoUO"
  }
}
```

更新用户

在 **查询用户列表** 部分，会返回用户的数据库索引。例如，用户 `kong` 的数据库 id 索引是 `2`，所以这里调用如下 URL 更新 `kong` 用户：

```
$ curl -XPUT -H "Content-Type: application/json" http://127.0.0.1:8080/v1/user/2 -d'{"username":"kong","password":"kongmodify"}'

{
  "code": 0,
  "message": "OK",
  "data": null
}
```

获取 `kong` 用户信息：

```
$ curl -XGET -H "Content-Type: application/json" http://127.0.0.1:8080/v1/user/kong

{
  "code": 0,
  "message": "OK",
  "data": {
    "username": "kong",
    "password": "$2a$10$E0kwtmtLZbwW/bDQ8qI8e.eHPqhQOW9tvjwpyo/p05f/f4Qvr3OmS"
  }
}
```

可以看到密码已经改变（旧密码为 `$2a$10$vE9jG7loyzstWVwB/QfU3u00Pxb.ye8hFIDvnyw60nHBv/xsJZoUO`）。

删除用户

在 **查询用户列表** 部分，会返回用户的数据库索引。例如，用户 `kong` 的数据库 id 索引是 `2`，所以这里调用如下 URL 删除 `kong` 用户：

```
$ curl -XDELETE -H "Content-Type: application/json" http://127.0.0.1:8080/v1/user/2

{
  "code": 0,
  "message": "OK",
  "data": null
}
```

获取用户列表：

```
$ curl -XGET -H "Content-Type: application/json" http://127.0.0.1:8080/v1/user -d'{"offset": 0, "limit": 20}'

{
  "code": 0,
  "message": "OK",
  "data": {
    "totalCount": 1,
    "userList": [
      {
        "id": 0,
        "username": "admin",
        "sayHello": "Hello EnqntiSig",
        "password": "$2a$10$veGcArz47VGj7l9xN7g2iuT9TF21jLI1YGXarGzvARNdnt4inC9PG",
        "createdAt": "2018-05-28 00:25:33",
        "updatedAt": "2018-05-28 00:25:33"
      }
    ]
  }
}
```

可以看到用户 `kong` 未出现在用户列表中，说明他已被成功删除。

小结

本小节通过对用户增删改查和查询列表的操作，介绍了实际开发中如何对 REST 资源进行操作，并结合笔者的实际开发经验给出了一些开发习惯和建议。

HTTP 调用添加自定义处理逻辑

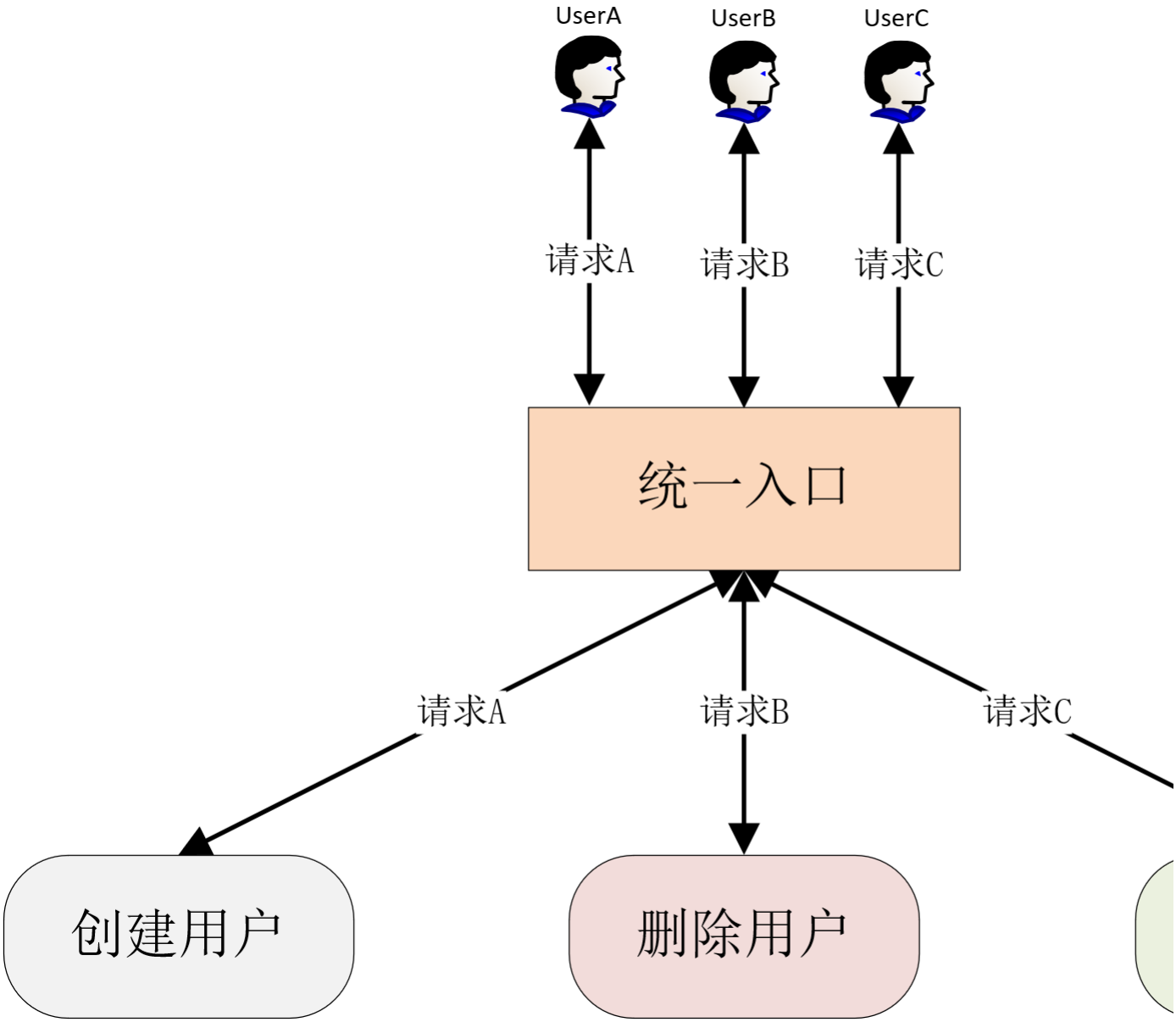
本节核心内容

- 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
>
> 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 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。

```go
func main() {
    // Creates a router without any middleware by default
    r := gin.New()

    // Global middleware
    // Logger middleware will write the logs to gin.DefaultWriter even if you set with GIN_MODE=release.
    // By default gin.DefaultWriter = os.Stdout
    r.Use(gin.Logger())

    // Recovery middleware recovers from any panics and writes a 500 if there was one.
    r.Use(gin.Recovery())

    // Per route middleware, you can add as many as you desire.
    r.GET("/benchmark", MyBenchLogger(), benchEndpoint)

    // Authorization group
    // authorized := r.Group("/", AuthRequired())
    // exactly the same as:
    authorized := r.Group("/")
    // per group middleware! in this case we use the custom created
    // AuthRequired() middleware just in the "authorized" group.
    authorized.Use(AuthRequired())
    {
        authorized.POST("/login", loginEndpoint)
        authorized.POST("/submit", submitEndpoint)
        authorized.POST("/read", readEndpoint)

        // nested group
        testing := authorized.Group("testing")
        testing.GET("/analytics", analyticsEndpoint)
    }

    // Listen and serve on 0.0.0.0:8080
    r.Run(":8080")
}
```

全局中间件

单个路由中间件

群组中间件

- 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）：

```go
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：

```go
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：

```go
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。

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"
kH2N.pPqtHB7ya0C","createdAt":"2018-06-05 21:25:53","updatedAt":"2018-06-05 21:25:53"},{"id":0,"username":"admin","sayHello":"
VGj7l9xN7g2iuT9TF21jLI1YGXarGzvARNdnt4inC9PG","createdAt":"2018-05-28 00:25:33","updatedAt":"2018-05-28 00:25:33"}]}}[api@cent
```

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 handl
[GIN-debug] POST   /v1/user                  --> apiserver/handler/user.Create (7 hand
[GIN-debug] DELETE /v1/user/:id              --> apiserver/handler/user.Delete (7 hand
[GIN-debug] PUT    /v1/user/:id              --> apiserver/handler/user.Update (7 hand
[GIN-debug] GET    /v1/user                  --> apiserver/handler/user.List (7 handle
[GIN-debug] GET    /v1/user/:username        --> apiserver/handler/user.Get (7 handler
[GIN-debug] GET    /sd/health                --> apiserver/handler/sd.HealthCheck (7 h
[GIN-debug] GET    /sd/disk                  --> apiserver/handler/sd.DiskCheck (7 han
[GIN-debug] GET    /sd/cpu                   --> apiserver/handler/sd.CPUCheck (7 hand
[GIN-debug] GET    /sd/ram                   --> apiserver/handler/sd.RAMCheck (7 hand
{"level":"INFO","timestamp":"2018-06-06 19:00:45.147","file":"apiserver/main.go:59","m
ming requests on http address: :8080"}
{"level":"INFO","timestamp":"2018-06-06 19:00:45.148","file":"apiserver/main.go:56","m
d successfully."}
{"level":"INFO","timestamp":"2018-06-06 19:01:26.625","file":"middleware/logging.go:78
.1    | GET /v1/user | {code: 0, message: OK}"}
{"level":"INFO","timestamp":"2018-06-06 19:01:28.417","file":"middleware/logging.go:78
.1    | GET /v1/user | {code: 0, message: OK}"}
{"level":"INFO","timestamp":"2018-06-06 19:01:44.945","file":"middleware/logging.go:78
.1    | GET /v1/user | {code: 0, message: OK}"}
{"level":"INFO","timestamp":"2018-06-06 19:02:26.627","file":"middleware/logging.go:78
.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。

API 身份验证

本节核心内容

- 介绍 API 身份验证的常用机制
- 介绍如何进行 API 身份验证

本小节源码下载路径：demo09

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 demo08 来开发的。

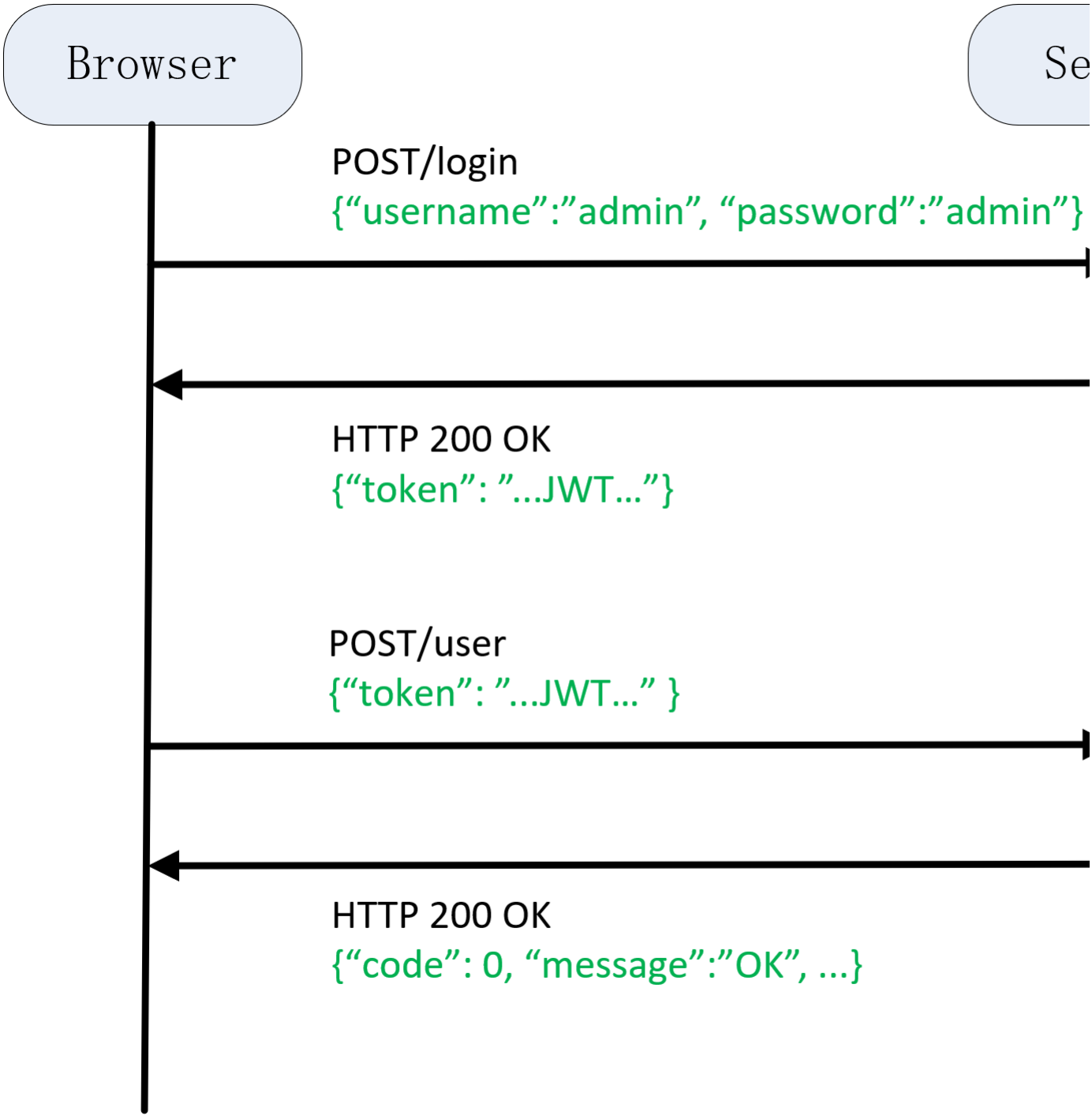
API 身份验证

在典型业务场景中，为了区分用户和安全保密，必须对 API 请求进行鉴权，但是不能要求每一个请求都进行登录操作。合理做法是，在第一次登录之后产生一个有一定有效期的 token，并将其存储于浏览器的 Cookie 或 LocalStorage 之中，之后的请求都携带该 token，请求到达服务器端后，服务器端用该 token 对请求进行鉴权。在第一次登录之后，服务器会将这个 token 用文件、数据库或缓存服务器等方法存下来，用于之后请求中的比对。或者，更简单的方法是，直接用密钥对用户信息和时间戳进行签名对称加密，这样就可以省下额外的存储，也可以减少每一次请求时对数据库的查询压力。这种方式，在业界已经有一种标准的实现方式，该方式被称为 JSON Web Token（JWT，音同 jot，详见 JWT RFC 7519）。

> token 的意思是“令牌”，里面包含了用于认证的信息。这里的 token 是指 JSON Web Token（JWT）。

JWT 简介

JWT 认证流程



1. 客户端使用用户名和密码请求登录
2. 服务端收到请求后会去验证用户名和密码，如果用户名和密码跟数据库记录不一致则验证失败，如果一致则验证通过，服务端会签发一个 Token 返回给客户端
3. 客户端收到请求后会将 Token 缓存起来，比如放在浏览器 Cookie 中或者本地存储中，之后每次请求都会携带该 Token
4. 服务端收到请求后会验证请求中携带的 Token，验证通过则进行业务逻辑处理并成功返回数据

在 JWT 中，Token 有三部分组成，中间用 . 隔开，并使用 Base64 编码：

- header
- payload
- signature

如下是 JWT 中的一个 Token 示例：

```
eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.eyJpYXQiOjE1MjgwMTY5MjIsImlkIjowLCJuYmYiOjE1MjgwMTY5MjIsInVzZXJuYWllIjoiYWRtaW4ifQ.LjxrK9DuAwAzUD8-9v43NzWBN7HXsSLfebw92DKdlJQ
```

header 介绍

JWT Token 的 header 中，包含两部分信息：

1. Token 的类型
2. Token 所使用的加密算法

例如：

```
{
  "typ": "JWT",
  "alg": "HS256"
}
```

该例说明 Token 类型是 JWT，加密算法是 HS256（alg 算法可以有多种）。

Payload 载荷介绍

Payload 中携带 Token 的具体内容，里面有一些标准的字段，当然你也可以添加额外的字段，来表达更丰富的信息，可以用这些信息来做更丰富的处理，比如记录请求用户名，标准字段有：

- iss：JWT Token 的签发者
- sub：主题
- exp：JWT Token 过期时间

- aud：接收 JWT Token 的一方
- iat：JWT Token 签发时间
- nbf：JWT Token 生效时间
- jti：JWT Token ID

本例中的 payload 内容为：

```
{
 "id": 2,
 "username": "kong",
 "nbf": 1527931805,
 "iat": 1527931805
}
```

Signature 签名介绍

`Signature` 是 Token 的签名部分，通过如下方式生成：

1. 用 Base64 对 header.payload 进行编码
2. 用 Secret 对编码后的内容进行加密，加密后的内容即为 `Signature`

Secret 相当于一个密码，存储在服务端，一般通过配置文件来配置 Secret 的值，本例中是配置在 `conf/config.yaml` 配置文件中：

```
runmode: debug                 # 开发模式, debug, release, test
addr: :8080                    # HTTP绑定端口
name: apiserver                # API Server的名字
url: http://127.0.0.1:8080     # pingServer函数请求的API服务器的i
max_ping_count: 10             # pingServer函数try的次数
jwt_secret: Rtg8BPKNEf2mB4mgvKONGPZZQSaJWNLijxR42qRgq0iBb5
log:
  writers: file,stdout
  logger_level: DEBUG
  logger_file: log/apiserver.log
  log_format_text: false
  rollingPolicy: size
  log_rotate_date: 1
  log_rotate_size: 1
  log_backup_count: 7
db:
```

最后生成的 Token 像这样：

```
eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.eyJpYXQiOjE1MjgwMTY5MjIsImlkIjowLCJuYmYiOjE1MjgwMTY5MjIsInVzZXJuYW1lIjoiYWRtaW4ifQ.LjxrK9DuAwAzUD8-9v43NzWBN7HXsSLfebw92DKdlJQ
```

签名后服务端会返回生成的 Token，客户端下次请求会携带该 Token，服务端收到 Token 后会解析出 header.payload，然后用相同的加密算法和密码对 header.payload 再进行一次加密，并对比加密后的 Token 和收到的 Token 是否相同，如果相同则验证通过，不相同则返回 `HTTP 401 Unauthorized` 的错误。

> 详细的 JWT 介绍参考 JWT – 基于 Token 的身份验证。

如何进行 API 身份验证

API 身份认证包括两步：

1. 签发 token
2. API 添加认证 middleware

签发 token

首先要实现登录接口。在登录接口中采用明文校验用户名密码的方式，登录成功之后再产生 token。在 `router/router.go` 文件中添加登录入口：

```
// api for authentication functionalities
g.POST("/login", user.Login)
```

在 `handler/user/login.go`（详见 demo09/handler/user/login.go）中添加 login 的具体实现：

1. 解析用户名和密码
2. 通过 `auth.Compare()` 对比密码是否是数据库保存的密码，如果不是，返回 `errno.ErrPasswordIncorrect` 错误
3. 如果相同，授权通过，通过 `token.Sign()` 签发 token 并返回

> `auth.Compare()` 的实现详见 demo09/pkg/auth/auth.go。
>
> `token.Sign()` 的实现详见 demo09/pkg/token/token.go。

API 添加认证 middleware

在 `router/router.go` 中对 user handler 添加授权 middleware：

```go
// api for authentication functionalities
g.POST("/login", user.Login)

// The user handlers, requiring authentication
u := g.Group("/v1/user")
u.Use(middleware.AuthMiddleware())
{
    u.POST("", user.Create)
    u.DELETE("/:id", user.Delete)
    u.PUT("/:id", user.Update)
    u.GET("", user.List)
    u.GET("/:username", user.Get)
}
```

通过该 middleware，所有对 `/v1/user` 路径的请求，都会经过 `middleware.AuthMiddleware()` 中间件的处理：token 校验。`middleware.AuthMiddleware()` 函数是通过调用 `token.ParseRequest()` 来进行 token 校验的。

`middleware.AuthMiddleware()` 实现详见 demo09/router/middleware/auth.go。

`token.ParseRequest()` 实现详见 demo09/pkg/token/token.go。

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/demo09` 复制为`$GOPATH/src/apiserver`

```
$ cp -a apiserver_demos/demo09/ $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 .
```

上文已经介绍过，API 身份验证首先需要登录，登录成功后会签发 token，之后请求时在 HTTP Header 中带上 token 即可。

1. 用户登录

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

{
  "code": 0,
  "message": "OK",
  "data": {
    "token": "eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.eyJpYXQiOjE1MjgwMTY5MjIsImlkIjowLCJuYmYiOjE1MjgwMTY5MjIsInVzZXJuYW1lIjoiYWRtaW4ifQ.LjxrK9DuAwAzUD8-9v43NzWBN7HXsSLfebw92DKdlJQ"
  }
}
```

返回的 token 为 `eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.eyJpYXQiOjE1MjgwMTY5MjIsImlkIjowLCJuYmYiOjE1MjgwMTY5MjIsInVzZXJuYW1lIjoiYWRtaW4ifQ.LjxrK9DuAwAzUD8-9v43NzWBN7HXsSLfebw92DKdlJQ`。

2. 请求时如果不携带签发的 token，会禁止请求

```
$ curl -XPOST -H "Content-Type: application/json" http://127.0.0.1:8080/v1/user -d'{"username":"user1","password":"user1234"}'

{
  "code": 20103,
  "message": "The token was invalid.",
  "data": null
}
```

3. 请求时携带 token

```
$ curl -XPOST -H "Authorization: Bearer eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.eyJpYXQiOjE1MjgwMTY5MjIsImlkIjowLCJuYmYiOjE1MjgwMTY5MjIsInVzZXJuYW1lIjoiYWRtaW4ifQ.LjxrK9DuAwAzUD8-9v43NzW

{
  "code": 0,
  "message": "OK",
  "data": {
    "username": "user1"
  }
}
```

可以看到携带 token 后验证通过，成功创建用户。通过 HTTP Header `Authorization: Bearer $token` 来携带 token。携带 token 后不需要再次查询数据库核对密码，即可完成授权。

小结

本小节介绍了 API 身份验证的相关知识。apiserver 采用的认证方式为 JWT，小节简单介绍了 JWT 的认证流程，并通过实例展示了具体如何进行 JWT 认证。

通过以上小节的学习，读者已经可以进行基本的 API 开发了，下一节开始介绍 API 开发的进阶内容。

用 HTTPS 加密 API 请求

本节核心内容

- 介绍 HTTPS 基本原理
- 介绍如何用 HTTPS 加密 API 请求，并测试加密效果

本小节源码下载路径：demo10

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 demo09 来开发的。

什么是 HTTPS

在前面的小节中，客户端与 API 服务器请求响应用的是 HTTP，不过 HTTP 是明文的，在网络上进行传输可能会被窃听、篡改甚至冒充，因此对于一个企业级的 API 服务器来说，通常需要采用更安全的 HTTPS 协议。

HTTPS（全称 Hyper Text Transfer Protocol over Secure Socket Layer），是以安全为目标的 HTTP 通道，简单讲是 HTTP 的安全版。即 HTTP 下加入 SSL 层，HTTPS 的安全基础是 SSL，因此加密的详细内

容就需要 SSL。

SSL：安全套接层，是 Netscape 公司设计的主要用于 Web 的安全传输协议。这种协议在 Web 上获得了广泛的应用。通过证书认证来确保客户端和网站服务器之间的通信数据是加密安全的。

TLS 是 SSL 的升级版，使用层面，读者可以理解为二者无区别。

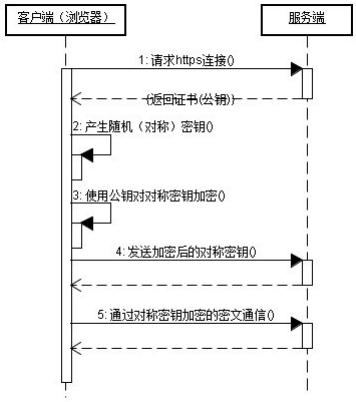
HTTPS 的实现原理

HTTPS 在传输数据之前需要客户端（浏览器）与服务端（网站）之间进行一次握手，在握手过程中将确立双方加密传输数据的密码信息。

有两种基本的加解密算法类型：

1. 对称加密：密钥只有一个，加密解密为同一个密码，且加解密速度快，典型的对称加密算法有 DES、AES 等。
2. 非对称加密：密钥成对出现（且根据公钥无法推知私钥，根据私钥也无法推知公钥），加密解密使用不同密钥（公钥加密需要私钥解密，私钥加密需要公钥解密），相对对称加密速度较慢，典型的非对称加密算法有 RSA、DSA 等。

下图是 HTTPS 的通信过程：



流程大概是这样的：

1. SSL 客户端通过 TCP 和服务器建立连接之后（443 端口），并且在一般的 TCP 连接协商（握手）过程中请求证书。

   即客户端发出一个消息给服务器，这个消息里面包含了自己可实现的算法列表和其它一些需要的消息，SSL 的服务器端会回应一个数据包，这里面确定了这次通信所需要的算法，然后服务器向客户端返回证书。（证书里面包含了服务器信息 —— 域名、申请证书的公司、公共秘钥。）

2. 客户端在收到服务器返回的证书后，判断签发这个证书的公共签发机构，并使用这个机构的公共秘钥确认签名是否有效，客户端还会确保证书中列出的域名就是它正在连接的域名。
3. 如果确认证书有效，那么生成对称秘钥并使用服务器的公共秘钥进行加密。然后把它发送给服务器，服务器使用它的私钥解密出会话密钥，然后把内容通过会话密钥对称加密，这样两台计算机可以开始进行对称加密进行通信。

HTTPS 通信的优点：

1. 客户端产生的密钥只有客户端和服务器端能得到；
2. 加密的数据只有客户端和服务器端才能得到明文；
3. 客户端到服务端的通信是安全的。

Go 语言 HTTPS 支持

Go 语言的 `net/http` 包中的 `ListenAndServeTLS()` 函数提供了对 HTTPS 的支持。`ListenAndServeTLS()` 函数的原型为：

```
func ListenAndServeTLS(addr string, certFile string, keyFile string, handler Handler) error
```

可以看出，这个函数原型其实和 HTTP 方式的差别就在于，需要提供数字证书 certFile 和私钥文件 keyFile。在测试环境，我们没有必要花钱去购买什么证书，利用 OpenSSL 工具，我们可以自己生成私钥文件和自签发的数字证书。

API Server 添加 HTTPS 支持

在 apiserver 中添加 HTTPS 功能，步骤很简单，大概分为以下三步。

1. 生成私钥文件（server.key）和自签发的数字证书（server.crt）：

```
$ openssl req -new -nodes -x509 -out conf/server.crt -keyout conf/server.key -days 3650 -subj "/C=DE/ST=NRW/L=Earth/O=Random Company/OU=IT/CN=127.0.0.1/emailAddress=xxxxx@qq.com"
```

```
[api@centos apiserver]$ ls conf
config.yaml  server.crt  server.key
[api@centos apiserver]$
```

2. 在配置文件中配置私钥文件、数字证书文件的路径和 HTTPS 端口，供 `ListenAndServeTLS()` 函数调用：

```
runmode: debug                 # 开发模式, debug, release, test
addr: :8080                    # HTTP绑定端口
name: apiserver                # API Server的名字
url: http://127.0.0.1:8080     # pingServer函数请求的API服务器的ip:port
max_ping_count: 10             # pingServer函数try的次数
jwt_secret: Rtg8BPKNEf2mB4mgvKONGPZZQSaJWNLijxR42qRgq0iBb5
tls:
  addr: :8081
  cert: conf/server.crt
  key: conf/server.key
log:
```

3. 在 `main` 函数中增加 `ListenAndServeTLS()` 调用，启动 HTTPS 端口：

```
    // 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.Info("The router has been deployed successfully.")
    }()

    // Start to listening the incoming requests.
    cert := viper.GetString("tls.cert")
    key := viper.GetString("tls.key")
    if cert != "" && key != "" {
        go func() {
            log.Infof("Start to listening the incoming requests on https address: %s", viper.GetString("tls.addr"))
            log.Info(http.ListenAndServeTLS(viper.GetString("tls.addr"), cert, key, g).Error())
        }()
    }

    log.Infof("Start to listening the incoming requests on http address: %s", viper.GetString("addr"))
    log.Info(http.ListenAndServe(viper.GetString("addr"), g).Error())
}
```

`main` 函数的逻辑是：如果提供了 TLS 证书和私钥则启动 HTTPS 端口。

创建证书和密钥需要 Linux 安装 `openssl` 工具，大部分 Linux 发行版已经默认安装，如果没有安装请安装。

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/demo10` 复制为 `$GOPATH/src/apiserver`

```
$ cp -a apiserver_demos/demo10/ $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 .
```

请求时不携带 CA 证书和私钥

```
$ curl -XGET -H "Authorization: Bearer eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.eyJpYXQiOjE1MjgwMTY5MjIsImlkIjowLCJuYmYiOjE1MjgwMTY5MjIsInVzZXJuYW1lIjoiYWRtaW4ifQ.LjxrK9DuAwAzUD8-9v43NzWB

curl: (60) Peer certificate cannot be authenticated with known CA certificates
More details here: http://curl.haxx.se/docs/sslcerts.html

curl performs SSL certificate verification by default, using a "bundle"
 of Certificate Authority (CA) public keys (CA certs). If the default
 bundle file isn't adequate, you can specify an alternate file
 using the --cacert option.
If this HTTPS server uses a certificate signed by a CA represented in
 the bundle, the certificate verification probably failed due to a
 problem with the certificate (it might be expired, or the name might
 not match the domain name in the URL).
If you'd like to turn off curl's verification of the certificate, use
 the -k (or --insecure) option.
```

可以看到请求认证失败。

请求协议需要是 HTTPS，请求的端口需要是 HTTPS 的 8081 端口。

请求时携带 CA 证书和私钥

```
$ curl -XGET -H "Authorization: Bearer eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.eyJpYXQiOjE1MjgwMTY5MjIsImlkIjowLCJuYmYiOjE1MjgwMTY5MjIsInVzZXJuYW1lIjoiYWRtaW4ifQ.LjxrK9DuAwAzUD8-9v43NzWB

{
  "code": 0,
  "message": "OK",
  "data": {
    "totalCount": 1,
    "userList": [
      {
        "id": 0,
        "username": "admin",
        "sayHello": "Hello qybyTdSmq",
        "password": "$2a$10$veGcArz47VGj7l9xN7g2iuT9TF21jLI1YGXarGzvARNdnt4inC9PG",
        "createdAt": "2018-05-28 00:25:33",
        "updatedAt": "2018-05-28 00:25:33"
      }
    ]
  }
}
```

成功请求。

小结

本小节是 API 开发的进阶内容，讲解了如何给 HTTP 请求进行数据加密。在企业级的 API 服务器中，通常会进行 HTTPS 加密。

用 Makefile 管理 API 项目

本节核心内容

- 介绍 Makefile
- 介绍如何使用 Makefile

本小节源码下载路径：demo11

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 demo10 来开发的。

为什么需要 Makefile

Go 语言的 `go` 命令自带源码管理功能，比如通过 `go build` 可以实现对源码的编译，但是 Go 自带的源码管理功能在实际项目中还是满足不了需求，有时候执行 `go build` 时，会附带很多编译参数，直接执行 `go build` 命令也会很麻烦。这时候一般是通过更专业的 Makefile 来管理源码，通过 Makefile 可以实现诸如：编译、安装、清理等功能，其实需要的管理功能都可以通过 Makefile 来添加，Makefile 生来就是做这些的。

Makefile 简介

一个工程中的源文件不计其数，其按类型、功能、模块分别放在若干个目录中，Makefile 定义了一系列的规则来指定，哪些文件需要先编译，哪些文件需要后编译，哪些文件需要重新编译，甚至于进行更复杂的功能操作，因为 Makefile 就像一个 Shell 脚本一样，其中也可以执行操作系统的命令（摘自百度百科）。

makefile 的规则

Makefile 基本格式如下：

```
target ... : prerequisites ...
    command
    ...
```

其中：

- target - 编译文件要生成的目标
- prerequisites - 编译文件需要的依赖
- command - 依赖生成目标所需要执行的命令（任意的 shell 命令），Makefile 中的命令必须以 [tab] 开头

比如我们平时使用的 `gcc a.c b.c -o test` 这里的 test 就是我们要生成的目标， a.c、b.c就是我们生成目标需要的依赖，而 `gcc a.c b.c -o test` 则是命令。将这行命令用 Makefile 的方式来写就是：

```
test: a.c b.c
    gcc a.c b.c -o test
```

API Server 添加 Makefile

在 apiserver 根目录下新建文件 `Makefile`，内容为：

```
all: gotool
 @go build -v .
clean:
 rm -f apiserver
 find . -name "[._]*.s[a-w][a-z]" | xargs -i rm -f {}
gotool:
 gofmt -w .
 go tool vet . |& grep -v vendor;true
ca:
 openssl req -new -nodes -x509 -out conf/server.crt -keyout conf/server.key -days 3650 -subj "/C=DE/ST=NRW/L=Earth/O=Random Company/OU=IT/CN=127.0.0.1/emailAddress=xxxxx@qq.com"

help:
 @echo "make - compile the source code"
 @echo "make clean - remove binary file and vim swp files"
 @echo "make gotool - run go tool 'fmt' and 'vet'"
 @echo "make ca - generate ca files"

.PHONY: clean gotool ca help
```

上面的 Makefile 文件中，`.PHONY` 是个伪目标，形式上是一个目标，但是不需要依赖，伪目标一般只是为了执行目标下面的命令（比如 clean 就是伪目标）。`@` 放在行首，表示不打印此行。默认在编译的过程中，会把此行的展开效果字符串打印出来。

上面的 Makefile 实现了如下功能：

- `make`：执行 `go build -v .` 生成 Go 二进制文件
- `make gotool`：执行 `gofmt -w .` 和 `go tool vet .`（格式化代码和源码静态检查）
- `make clean`：做一些清理工作：删除二进制文件、删除 vim swp 文件
- `make ca`：生成证书
- `make help`：打印 help 信息

编译

在前面各小节中编译二进制均是通过 `go build -v .` 的方式，添加 Makefile 后可以通过如下方式来编译：

```
$ make
```

小结

本小节简单介绍了 Makefile，并介绍了 apiserver 所使用的 Makefile 文件，通过该小节，展示了如何通过 Makefile 来管理和编译 API 源码。

本小册不是专门介绍 Makefile 的，想要了解更多 Makefile 知识，请参考 Makefile使用总结。

给 API 命令增加版本功能

本节核心内容

- 如何给 apiserver 增加版本功能

  本小节源码下载路径：demo12

  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 demo11 来开发的。

为什么需要版本

在实际开发中，当开发完一个 apiserver 特性后，会编译 apiserver 二进制文件并发布到生产环境，很多时候为了定位问题和出于安全目的（不能发错版本），我们需要知道当前 apiserver 的版本，以及一些编译时候的信息，如编译时 Go 的版本、Git 目录是否 clean，以及基于哪个 git commmit 来编译的。在一个编译好的可执行程序中，我们通常可以用类似 `./app_name -v` 的方式来获取版本信息。

我们可以将这些信息写在配置文件中，程序运行时从配置文件中取得这些信息进行显示。但是在部署程序时，除了二进制文件还需要额外的配置文件，不是很方便。或者将这些信息写入代码中，这样不需要额外的配置，但要在每次编译时修改代码文件，也比较麻烦。Go 官方提供了一种更好的方式：通过 `-ldflags -X importpath.name=value`（详见 -ldflags -X importpath.name=value）来给程序自动添加版本信息。

> 在实际开发中，绝大部分都是用 Git 来做源码版本管理的，所以 apiserver 的版本功能也基于 Git。

给 `apiserver` 添加版本功能

假设我们程序发布的流程是这样：

1. 编码完成，提交测试工程师测试
2. 测试工程师测试代码，提交 bug，更改 bug 并重新测试后验证通过
3. 开发人员把验证通过的代码合并到 master 分支，并打上版本号：`git tag -a v1.0.0`
4. 开发人员将 `v1.0.0` 版本发布到生产环境

最终发布后，我们希望通过 `./apiserver -v` 参数提供如下版本信息：

- 版本号
- git commit
- git tree 在编译时的状态
- 构建时间
- go 版本
- go 编译器
- 运行平台

为了实现这些功能，我们首先要在 `main` 函数中添加用于接收 `-v` 参数的入口（详见 demo12/main.go）：

```
package main

import (
 "encoding/json"
 "fmt"
 "os"
 ...
 v "apiserver/pkg/version"
 ...
)

var (
 version = pflag.BoolP("version", "v", false, "show version info.")
)
```

```
func main() {
 pflag.Parse()
 if *version {
  v := v.Get()
  marshalled, err := json.MarshalIndent(&v, "", "  ")
  if err != nil {
   fmt.Printf("%v\n", err)
   os.Exit(1)
  }

  fmt.Println(string(marshalled))
  return
 }
 ...
}
```

通过 `pflag` 来解析命令行上传入的 `-v` 参数。

通过 `pkg/version` 的 `Get()` 函数来获取 apiserver 的版本信息。

通过 `json.MarshalIndent` 来格式化打印版本信息。

`pkg/version` 的 `Get()` 函数实现为（详见 demo12/pkg/version）：

```
func Get() Info {
 return Info{
  GitTag:       gitTag,
  GitCommit:    gitCommit,
  GitTreeState: gitTreeState,
  BuildDate:    buildDate,
  GoVersion:    runtime.Version(),
  Compiler:     runtime.Compiler,
  Platform:     fmt.Sprintf("%s/%s", runtime.GOOS, runtime.GOARCH),
 }
}
```

其中 `gitTag`、`gitCommit`、`gitTreeState` 等变量的值是通过 `-ldflags -X importpath.name=value` 在编译时传到程序中的。为此我们需要在编译时传入这些信息，在 Makefile 中添加如下信息（详见 demo12/Makefile）：

```
SHELL := /bin/bash
BASEDIR = $(shell pwd)

# build with verison infos
versionDir = "apiserver/pkg/version"
gitTag = $(shell if [ "`git describe --tags --abbrev=0 2>/dev/null`" != "" ];then git describe --tags --abbrev=0; else git log --pretty=format:'%h' -n 1; fi)
buildDate = $(shell TZ=Asia/Shanghai date +%FT%T%z)
gitCommit = $(shell git log --pretty=format:'%H' -n 1)
gitTreeState = $(shell if git status|grep -q 'clean';then echo clean; else echo dirty; fi)

ldflags="-w -X ${versionDir}.gitTag=${gitTag} -X ${versionDir}.buildDate=${buildDate} -X ${versionDir}.gitCommit=${gitCommit} -X ${versionDir}.gitTreeState=${gitTreeState}"
```

并在 `go build` 中添加这些 flag：

```
go build -v -ldflags ${ldflags} .
```

> `-w` 为去掉调试信息（无法使用 gdb 调试），这样可以使编译后的二进制文件更小。

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/demo12` 复制为 `$GOPATH/src/apiserver`

```
$ cp -a apiserver_demos/demo12/ $GOPATH/src/apiserver
```

3. 在 apiserver 目录下编译源码

```
$ cd $GOPATH/src/apiserver
$ make
```

查看 apiserver 版本

```
$ ./apiserver -v
```

```
{
  "gitTag": "7322949",
  "gitCommit": "732294928b3c4dff5b898fde0bb5313752e1173e",
  "gitTreeState": "dirty",
  "buildDate": "2018-06-05T07:43:26+0800",
  "goVersion": "go1.10.2",
  "compiler": "gc",
  "platform": "linux/amd64"
}
```

可以看到 `./apiserver -v` 输出了我们需要的版本信息。

> 在上一小节中我们已经给 `apiserver` 添加过 Makefile 文件。

小结

本小节主要介绍如何用 Makefile 以及 Go 本身所支持的编译特性，实现编译时自动生成版本号的功能。后续小节编译 API 源码均会通过 `make` 来编译。

给 API 增加启动脚本

本节核心内容

- 如何管理 apiserver 启动命令，包括启动、重启、停止和查看运行状态

  本小节源码下载路径：demo13

  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 demo12 来开发的。

为什么要添加启动脚本

通过添加服务器启动脚本可以很方便地启动、重启、停止和查看服务的状态。一些监控系统、发布系统需要有方法告诉它怎么启停和查看服务状态，这时候如果有个服务控制脚本就可以很方便地添加，要不然输入一堆启动参数不仅烦琐还容易出错。

添加启动脚本

apiserver 是通过 `admin.sh` 脚本来实现服务启动、重启、停止和查看服务状态操作的（详见 demo13/admin.sh），源码为：

```
#!/bin/bash

SERVER="apiserver"
BASE_DIR=$PWD
INTERVAL=2

# 命令行参数，需要手动指定
ARGS=""

function start()
{
 if [ "`pgrep $SERVER -u $UID`" != "" ];then
  echo "$SERVER already running"
  exit 1
 fi

 nohup $BASE_DIR/$SERVER $ARGS  server &>/dev/null &

 echo "sleeping..." &&  sleep $INTERVAL
```

```
 # check status
 if [ "`pgrep $SERVER -u $UID`" == "" ];then
  echo "$SERVER start failed"
  exit 1
 fi
}

function status()
{
 if [ "`pgrep $SERVER -u $UID`" != "" ];then
  echo $SERVER is running
 else
  echo $SERVER is not running
 fi
}

function stop()
{
 if [ "`pgrep $SERVER -u $UID`" != "" ];then
  kill -9 `pgrep $SERVER -u $UID`
 fi

 echo "sleeping..." &&  sleep $INTERVAL

 if [ "`pgrep $SERVER -u $UID`" != "" ];then
  echo "$SERVER stop failed"
  exit 1
 fi
}

case "$1" in
 'start')
 start
 ;;
 'stop')
 stop
 ;;
 'status')
 status
 ;;
 'restart')
 stop && start
 ;;
 *)
 echo "usage: $0 {start|stop|restart|status}"
 exit 1
 ;;
esac
```

看 shell 源码发现在 start 和 stop 时会 sleep 几秒，这是因为 API 服务器的启动需要时间去做准备工作，停止也需要时间去做清理工作。

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/demo13` 复制为 `$GOPATH/src/apiserver`

```
$ cp -a apiserver_demos/demo13/ $GOPATH/src/apiserver
```

3. 在 apiserver 目录下编译源码

```
$ cd $GOPATH/src/apiserver
$ make
```

查看 `admin.sh` 用法

```
$ ./admin.sh -h
usage: ./admin.sh {start|stop|restart|status}
```

查看 `apiserver` 运行状态

```
$ ./admin.sh status
apiserver is not running
```

启动 `apiserver`

```
$ ./admin.sh start
sleeping...
```

查看 `apiserver` 状态

```
$ ./admin.sh status
apiserver is running
```

重启 `apiserver`

```
$ ./admin.sh restart
sleeping...
sleeping...
```

停止 `apiserver`

```
$ ./admin.sh stop
sleeping...
```

小结

本小结展示了如何用 `admin.sh` 去管理 API 服务器：启动、重启、停止和查看状态。该 `admin.sh` 命令在进行 start、stop、restart 和 status 操作时做了很多检查工作，以确保运行结果是正确的。

基于 Nginx 的 API 部署方案

本节核心内容

- 介绍 Nginx
- 介绍如何安装 Nginx
- 介绍如何配置 Nginx

本小节源码下载路径：demo14

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 demo13 来开发的。

Nginx 介绍

Nginx 是一个自由、开源、高性能及轻量级的 HTTP 服务器和反向代理服务器，它有很多功能，主要功能为：

1. 正向代理
2. 反向代理
3. 负载均衡
4. HTTP 服务器（包含动静分离）

本小册使用了 Nginx 反向代理和负载均衡的功能。

Nginx 的更详细介绍可以参考 nginx简易教程。

Nginx 反向代理功能

Nginx 最常用的功能之一是作为一个反向代理服务器。反向代理（Reverse Proxy）是指以代理服务器来接收 Internet 上的连接请求，然后将请求转发给内部网络上的服务器，并将从服务器上得到的结果返

回给 Internet 上请求连接的客户端，此时代理服务器对外就表现为一个反向代理服务器（摘自百度百科）。

为什么需要反向代理呢？在实际的生产环境中，服务部署的网络（内网）跟外部网络（外网）通常是不通的，需要通过一台既能够访问内网又能够访问外网的服务器来做中转，这种服务器就是反向代理服务器。Nginx 作为反向代理服务器，简单的配置如下：

```
    server {
        listen      80;
        server_name  apiserver.com;
        client_max_body_size 1024M;

        location / {
            proxy_set_header Host $http_host;
            proxy_set_header X-Forwarded-Host $http_host;
            proxy_set_header X-Real-IP $remote_addr;
            proxy_set_header X-Forwarded-For $proxy_add_x_forwarded_for;
            proxy_pass  http://127.0.0.1:8080/;
            client_max_body_size 100m;
        }
    }
```

Nginx 在做反向代理服务器时，能够根据不同的配置规则转发到后端不同的服务器上。

Nginx 负载均衡功能

Nginx 另一个常用的功能是负载均衡，所谓的负载均衡就是指当 Nginx 收到一个 HTTP 请求后，会根据负载策略将请求转发到不同的后端服务器上。比如，apiserver 部署在两台服务器 A 和 B 上，当请求到达 Nginx 后，Nginx 会根据 A 和 B 服务器上的负载情况，将请求转发到负载较小的那台服务器上。这里要求 apiserver 是无状态的服务。

安装和启动 Nginx（需要切换到 root 用户）

1. 安装 Nginx（CentOS 7.x 安装流程）

```
$ yum -y install nginx
```

2. 确认 Nginx 安装成功

```
$ nginx -v
```

3. 启动 Nginx

```
$ systemctl start nginx
```

4. 设置开机启动

```
$ systemctl enable nginx
```

5. 查看 Nginx 启动状态

```
$ systemctl status nginx
```

Nginx 常用命令

Nginx 常用命令如下（执行 `which nginx` 可以找到 Nginx 命令所在的路径）：

```
nginx -s stop       快速关闭 Nginx，可能不保存相关信息，并迅速终止 Web 服务
nginx -s quit       平稳关闭 Nginx，保存相关信息，有安排的结束 Web 服务
nginx -s reload     因改变了 Nginx 相关配置，需要重新加载配置而重载
nginx -s reopen     重新打开日志文件
nginx -c filename   为 Nginx 指定一个配置文件，来代替默认的
nginx -t            不运行，而仅仅测试配置文件。Nginx 将检查配置文件的语法的正确性，并尝试打开配置文件中所引用到的文件
nginx -v            显示 Nginx 的版本
nginx -V            显示 Nginx 的版本、编译器版本和配置参数
```

Nginx 默认监听 80 端口，启动 Nginx 前要确保 80 端口没有被占用。当然你也可以通过修改 Nginx 配置文件 `/etc/nginx/nginx.conf` 改 Nginx 监听端口。

配置 Nginx 作为反向代理

假定要访问的 API 服务器域名为 `apiserver.com`，在 `/etc/nginx/nginx.conf` 配置 API 服务器的 `server` 入口：

```
http {
    include       /etc/nginx/mime.types;
    default_type  application/octet-stream;

    log_format  main  '$remote_addr - $remote_user [$time_local] "$request" '
                      '$status $body_bytes_sent "$http_referer" '
                      '"$http_user_agent" "$http_x_forwarded_for"';

    access_log  /var/log/nginx/access.log  main;

    sendfile        on;
    #tcp_nopush     on;

    keepalive_timeout  65;

    #gzip  on;

    include /etc/nginx/conf.d/*.conf;

    server {
        listen      80;
        server_name  apiserver.com;
        client_max_body_size 1024M;

        location / {
            proxy_set_header Host $http_host;
            proxy_set_header X-Forwarded-Host $http_host;
            proxy_set_header X-Real-IP $remote_addr;
            proxy_set_header X-Forwarded-For $proxy_add_x_forwarded_for;
            proxy_pass  http://127.0.0.1:8080/;
            client_max_body_size 100m;
        }
    }
}
```

完成 nginx.conf 内容如下：

```
user  nginx;
worker_processes  1;

error_log  /var/log/nginx/error.log warn;
pid        /var/run/nginx.pid;


events {
    worker_connections  1024;
}

http {
```

```
    include       /etc/nginx/mime.types;
    default_type  application/octet-stream;

    log_format  main  '$remote_addr - $remote_user [$time_local] "$request" '
                      '$status $body_bytes_sent "$http_referer" '
                      '"$http_user_agent" "$http_x_forwarded_for"';

    access_log  /var/log/nginx/access.log  main;

    sendfile        on;
    #tcp_nopush     on;

    keepalive_timeout  65;

    #gzip  on;

    include /etc/nginx/conf.d/*.conf;

    server {
        listen      80;
        server_name  apiserver.com;
        client_max_body_size 1024M;

        location / {
            proxy_set_header Host $http_host;
            proxy_set_header X-Forwarded-Host $http_host;
            proxy_set_header X-Real-IP $remote_addr;
            proxy_set_header X-Forwarded-For $proxy_add_x_forwarded_for;
            proxy_pass  http://127.0.0.1:8080/;
            client_max_body_size 5m;
        }
    }
}
```

配置说明

- 由于 Nginx 默认允许客户端请求的最大单文件字节数为 1MB，实际生产环境中可能太小，所以这里将此限制改为 5MB（`client_max_body_size 5m`）
- `server_name`：说明使用哪个域名来访问
- `proxy_pass`：反向代理的路径（这里是本机的 API 服务，所以IP为 127.0.0.1。端口要和 API 服务端口一致：8080）

  如果需要上传图片之类的，可能需要设置成更大的值，比如 50m。

  因为 Nginx 配置选项比较多，跟实际需求和环境有关，所以这里的配置是基础的、未经优化的配置，在实际生产环境中，需要读者再做调节。

测试

1. 配置完 Nginx 后重启 Nginx

```
$ systemctl restart nginx
```

2. 在编译完 apiserver 后，启动 API 服务器

```
$ ./apiserver
```

3. 在 `/etc/hosts` 中添加一行：`127.0.0.1 apiserver.com`

4. 发送 HTTP 请求

```
$ curl -XGET -H "Content-Type: application/json" -H "Authorization: Bearer eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.eyJpYXQiOjE1MjgwMTY5MjIsImlkIjowLCJuYmYiOjE1MjgwMTY5MjIsInVzZXJuYW1lIjo
```

```
{
  "code": 0,
  "message": "OK",
  "data": {
    "totalCount": 1,
    "userList": [
      {
        "id": 0,
        "username": "admin",
        "sayHello": "Hello Jypl3DSig",
        "password": "$2a$10$veGcArz47VGj7l9xN7g2iuT9TF21jLI1YGXarGzvARNdnt4inC9PG",
        "createdAt": "2018-05-28 00:25:33",
        "updatedAt": "2018-05-28 00:25:33"
      }
    ]
  }
}
```

可以看到成功通过代理访问后端的 API 服务。

请求流程说明

在用 `curl` 请求 `http://apiserver.com/v1/user` 后，后端的请求流程实际上是这样的：

1. 因为在 `/etc/hosts` 中配置了 `127.0.0.1 apiserver.com`，所以请求 `http://apiserver.com/v1/use` 实际上是请求本机的 Nginx 端口（`127.0.0.1:80`）
2. Nginx 在收到请求后，解析到请求域名为 `apiserver.com`，根据请求域名去匹配 Nginx 的 server 配置，匹配到 `server_name apiserver.com` 配置
3. 匹配到 server 后，把请求转发到该 server 的 `proxy_pass` 路径
4. 等待 API 服务器返回结果，并返回客户端

配置 Nginx 作为负载均衡

负载均衡的演示需要多个后端服务，为此我们在同一个服务器上启动多个 apiserver，配置不同的端口（8080、8082），并采用 Nginx 默认的轮询转发策略（轮询：每个请求按时间顺序逐一分配到不同的后端服务器）。

在 `/etc/nginx/nginx.conf` 中添加 `upstream` 配置：

```
http {
    include       /etc/nginx/mime.types;
    default_type  application/octet-stream;

    log_format  main  '$remote_addr - $remote_user [$time_local] "$request" '
                      '$status $body_bytes_sent "$http_referer" '
                      '"$http_user_agent" "$http_x_forwarded_for"';

    access_log  /var/log/nginx/access.log  main;

    sendfile        on;
    #tcp_nopush     on;

    keepalive_timeout  65;

    #gzip  on;

    include /etc/nginx/conf.d/*.conf;
    upstream apiserver.com {
        server 127.0.0.1:8080;
        server 127.0.0.1:8082;
    }

    server {
        listen      80;
        server_name  apiserver.com;
        client_max_body_size 1024M;

        location / {
            proxy_set_header Host $http_host;
            proxy_set_header X-Forwarded-Host $http_host;
            proxy_set_header X-Real-IP $remote_addr;
            proxy_set_header X-Forwarded-For $proxy_add_x_forwarded_for;
            proxy_pass  http://apiserver.com/;
            client_max_body_size 100m;
        }
    }
}
```

配置说明

- 因为有多个后端，所以需要将之前固定的后端 `proxy_pass http://127.0.0.1:8080/` 换成具有多个后端的 `apiserver.com`（通过 `upstream`）
- `upstream` 配置中配置多个后端（ip:port）

```
upstream apiserver.com {
    server 127.0.0.1:8080;
    server 127.0.0.1:8082;
}
```

测试

1. 配置完 Nginx 后重启 Nginx

```
$ systemctl restart nginx
```

2. 这里需要构建并发请求，编写测试脚本 `test.sh`，内容为：

```
#!/bin/bash


for n in $(seq 1 1 10)
do
    nohup curl -XGET -H "Content-Type: application/json" -H "Authorization: Bearer eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.eyJpYXQiOjE1MjgwMTY5MjIsImlkIjowLCJuYmYiOjE1MjgwMTY5MjIsInVzZXJ
done
```

3. 为了展示哪个 API 被调用，需要在查询用户列表的入口函数（`handler/user/list.go`文件中的 `List()` 函数）中添加日志打印信息：

```
package user

import (
    . "apiserver/handler"
    "apiserver/pkg/errno"
    "apiserver/service"

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

// List list the users in the database.
func List(c *gin.Context) {
    log.Info("List function called.")
    var r ListRequest
    if err := c.Bind(&r); err != nil {
        SendResponse(c, errno.ErrBind, nil)
        return
    }

    infos, count, err := service.ListUser(r.Username, r.Offset, r.Limit)
    if err != nil {
        SendResponse(c, err, nil)
        return
    }

    SendResponse(c, nil, ListResponse{
        TotalCount: count,
        UserList:   infos,
    })
}
```

4. 在相同服务器上启动两个不同的 HTTP 端口：8080 和 8082

5. 执行 test.sh 脚本

```
$ ./test.sh
```

观察 API 日志，可以看到请求被均衡地转发到后端的两个服务：

apiserver1（8080 端口）：

```
[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 (5 handlers)
[GIN-debug] POST   /v1/user                --> apiserver/handler/user.Create (6 handlers)
[GIN-debug] DELETE /v1/user/:id            --> apiserver/handler/user.Delete (6 handlers)
[GIN-debug] PUT    /v1/user/:id            --> apiserver/handler/user.Update (6 handlers)
[GIN-debug] GET    /v1/user                --> apiserver/handler/user.List (6 handlers)
[GIN-debug] GET    /v1/user/:username      --> apiserver/handler/user.Get (6 handlers)
[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)
2018-06-05 14:03:01.395 INFO apiserver/main.go:85 Start to listening the incoming requests on http address: :8080
2018-06-05 14:03:01.396 INFO apiserver/main.go:80 Start to listening the incoming requests on https address: :8081
2018-06-05 14:03:01.396 INFO apiserver/main.go:81 listen tcp :8081: bind: address already in use
2018-06-05 14:03:01.396 INFO apiserver/main.go:72 The router has been deployed successfully.
2018-06-05 14:04:49.835 INFO user/list.go:14 List function called.
2018-06-05 14:04:49.847 INFO user/list.go:14 List function called.
2018-06-05 14:04:49.858 INFO user/list.go:14 List function called.
2018-06-05 14:04:49.868 INFO user/list.go:14 List function called.
2018-06-05 14:04:49.881 INFO user/list.go:14 List function called.
```

收到5个请求

apiserver2（8082 端口）：

```
[api@centos apiserver]$ ./apiserver -c config.yaml
[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 (5 handlers)
[GIN-debug] POST   /v1/user                --> apiserver/handler/user.Create (6 handlers)
[GIN-debug] DELETE /v1/user/:id            --> apiserver/handler/user.Delete (6 handlers)
[GIN-debug] PUT    /v1/user/:id            --> apiserver/handler/user.Update (6 handlers)
[GIN-debug] GET    /v1/user                --> apiserver/handler/user.List (6 handlers)
[GIN-debug] GET    /v1/user/:username      --> apiserver/handler/user.Get (6 handlers)
[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)
2018-06-05 14:03:04.018 INFO apiserver/main.go:85 Start to listening the incoming requests on http address: :8082
2018-06-05 14:03:04.018 INFO apiserver/main.go:80 Start to listening the incoming requests on https address: :8081
2018-06-05 14:03:04.020 INFO apiserver/main.go:72 The router has been deployed successfully.
2018-06-05 14:04:49.841 INFO user/list.go:14 List function called.
2018-06-05 14:04:49.852 INFO user/list.go:14 List function called.
2018-06-05 14:04:49.863 INFO user/list.go:14 List function called.
2018-06-05 14:04:49.875 INFO user/list.go:14 List function called.
2018-06-05 14:04:49.886 INFO user/list.go:14 List function called.
```

收到5个请求

小结

在生产环境中，API 服务器所在的网络通常不能直接通过外网访问，需要通过可从外网访问的 Nginx 服务器，将请求转发到内网的 API 服务器。并且随着业务规模越来越大，请求量也会越来越大，这时候需要将 API 横向扩容，也需要 Nginx。所以在实际的 API 服务部署中 Nginx 经常能派上用场。通过本小节的学习，读者可以了解到如何在实际生产环境中部署 API 服务。

API 高可用方案

本小节为可选小节。因为该方案需要有至少两台服务器，没有多台服务器的读者只需要了解即可。

方案介绍

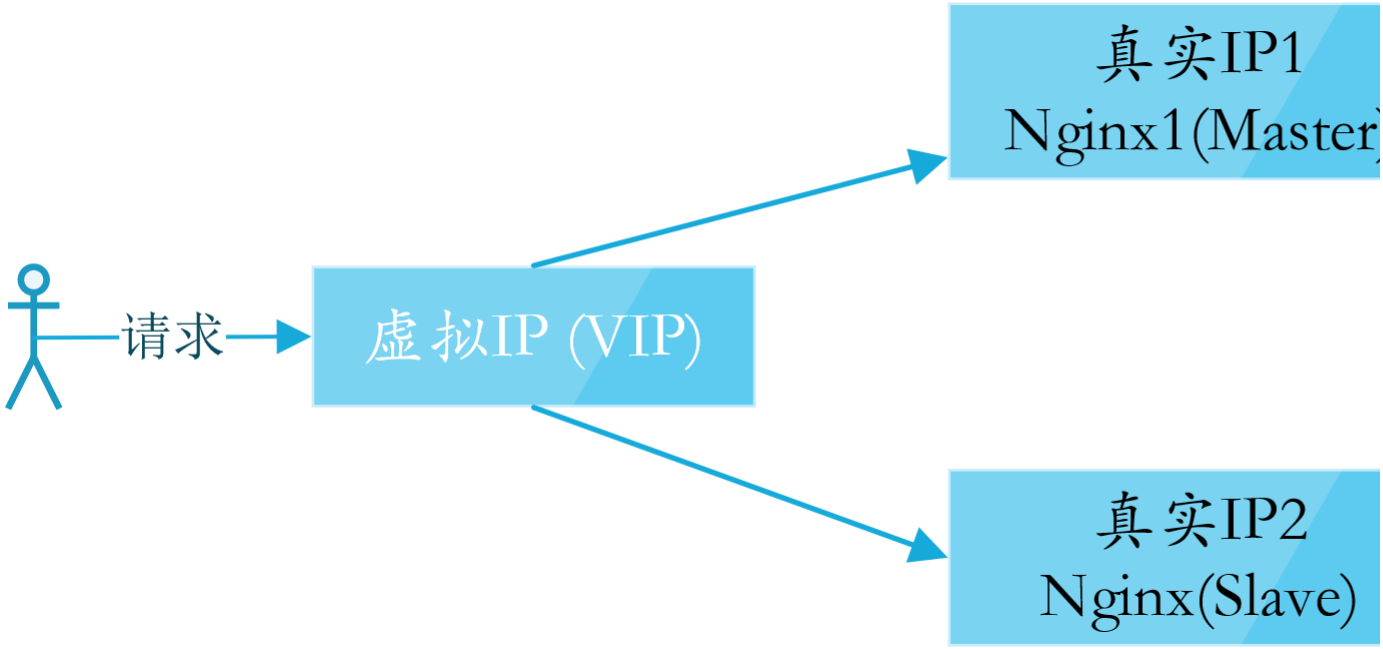
Nginx 自带负载均衡功能，并且当 Nginx 后端某个服务器挂掉后，Nginx 会自动剔除该服务器，将请求转发到可用的服务器，通过这种方式实现了后端 API 服务的高可用（HA）。但是 Nginx 是单点的，如果 Nginx 挂了，后端的所有服务器就都不能访问，所以在实际生产环境中，也需要对 Nginx 做高可用。

Keepalived 是一个高性能的服务器高可用或热备解决方案，Keepalived 主要来防止服务器单点故障的发生问题，可以通过 Keepalived 对前端 Nginx 实现高可用。Keepalived + Nginx 的高可用方案具有如下特点：

1. 服务功能强大
2. 维护简单

Keepalived 简介

Keepalived 以 VRRP 协议为基础来实现高可用性。VRRP（Virtual Router Redundancy Protocol，虚拟路由冗余协议）是用于实现路由器冗余的协议，它将两台或多台路由器设备虚拟成一个设备，对外提供虚拟路由器 IP（一个或多个），如下图所示。



在上图中，将 Nginx + Keepalived 部署在两台服务器上，拥有两个真实的 IP（IP1 和 IP2），通过一定的技术（如 LVS）虚拟出一个虚拟 IP（VIP），外界请求通过访问 VIP 来访问服务。在两台 Nginx + Keepalived 的服务器上，同一时间只有一台会接管 VIP（叫做 Master）提供服务，另一台（叫做 Slave）会检测 Master 的心跳，当发现 Master 停止心跳后，Slave 会自动接管 VIP 以提供服务（此时，Slave 变成 Master）。通过这种方式来实现 Nginx 的高可用，通过第 19 节可以知道，Nginx 可以对后台 API 服务器做高可用，这样通过 Nginx + Keepalived 的组合方案就实现了整个 API 集群的高可用。

部署

Keepalived + Nginx 的部署方案网上有很多详细的教程。因为小册篇幅限制，这里不再详细说明，读者如需了解，请参考 Keepalived+Nginx实现高可用（HA）。

go test 测试你的代码

在实际开发中，不仅要开发功能，更重要的是确保这些功能稳定可靠，并且拥有一个不错的性能，要确保这些就要对代码进行测试，开发人员通常会进行单元测试和性能测试。不同的语言通常都有自己的测试包/模块，Go 语言也一样，在 Go 中可以通过 testing 包对代码进行单元和性能测试，下面就来详细介绍。

本节核心内容

- 如何进行单元测试
- 如何进行压力/性能测试
- 如何进行性能分析

本小节源码下载路径：demo15

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 demo14 来开发的。

Go 语言测试支持

Go 语言有自带的测试框架 testing，可以用来实现单元测试和性能测试，通过 `go test` 命令来执行单元测试和性能测试。

`go test` 执行测试用例时，是以 go 包为单位进行测试的。执行时需要指定包名，比如：`go test 包名`，如果没有指定包名，默认会选择执行命令时所在的包。`go test` 在执行时会遍历以 `_test.go` 结尾的源码文件，执行其中以 `Test`、`Benchmark`、`Example` 开头的测试函数。其中源码文件需要满足以下规范：

- 文件名必须是 `_test.go` 结尾，跟源文件在同一个包。
- 测试用例函数必须以 Test、Benchmark、Example 开头
- 执行测试用例时的顺序，会按照源码中的顺序依次执行
- 单元测试函数 TestXxx() 的参数是 testing.T，可以使用该类型来记录错误或测试状态
- 性能测试函数 BenchmarkXxx() 的参数是 testing.B，函数内以 b.N 作为循环次数，其中 N 会动态变化
- 示例函数 ExampleXxx() 没有参数，执行完会将输出与注释 `// Output:` 进行对比
- 测试函数原型：func TestXxx(t *testing.T)，Xxx 部分为任意字母数字组合，首字母大写，例如：`TestgenShortId` 是错误的函数名，`TestGenShortId` 是正确的函数名
- 通过调用 testing.T 的 Error、Errorf、FailNow、Fatal、FatalIf 方法来说明测试不通过，通过调用 Log、Logf 方法来记录测试信息：

```
t.Log t.Logf     # 正常信息
t.Error t.Errorf # 测试失败信息
t.Fatal t.Fatalf # 致命错误，测试程序退出的信息
t.Fail     # 当前测试标记为失败
t.Failed   # 查看失败标记
t.FailNow  # 标记失败，并终止当前测试函数的执行，需要注意的是，我们只能在运行测试函数的 Goroutine 中调用 t.FailNow 方法，而不能在我们在测试代码创建出的 Goroutine 中调用它
t.Skip     # 调用 t.Skip 方法相当于先后对 t.Log 和 t.SkipNow 方法进行调用，而调用 t.Skipf 方法则相当于先后对 t.Logf 和 t.SkipNow 方法进行调用。方法 t.Skipped 的结果值会告知我们当前的测试是否已被忽略
t.Parallel # 标记为可并行运算
```

编写测试用例（对 `GenShortId` 函数进行单元测试）

1. 在 `util` 目录下创建文件 `util_test.go`，内容为：

```
package util

import (
 "testing"
)

func TestGenShortId(t *testing.T) {
 shortId, err := GenShortId()
 if shortId == "" || err != nil {
  t.Error("GenShortId failed!")
 }

 t.Log("GenShortId test pass")
}
```

从用例可以看出，如果 `GenShortId()` 返回的 `shortId` 为空或者 `err` 不为空，则调用 `t.Error()` 函数标明该用例测试不通过。

执行用例

在 `util` 目录下执行命令 `go test`：

```
$ cd util/
$ go test
PASS
ok   apiserver/util 0.006s
```

要查看更详细的执行信息可以执行 `go test -v`：

```
$ go test -v
```

```
=== RUN   TestGenShortId
--- PASS: TestGenShortId (0.00s)
 util_test.go:13: GenShortId test pass
PASS
ok   apiserver/util 0.006s
```

根据 `go test` 的输出可以知道 `TestGenShortId` 用例测试通过。

如果要执行测试 N 次可以使用 `-count N`：

```
$ go test -v -count 2
=== RUN   TestGenShortId
--- PASS: TestGenShortId (0.00s)
 util_test.go:13: GenShortId test pass
=== RUN   TestGenShortId
--- PASS: TestGenShortId (0.00s)
 util_test.go:13: GenShortId test pass
PASS
ok   apiserver/util 0.006s
```

编写性能测试用例

在 `util/util_test.go` 测试文件中，新增两个性能测试函数：`BenchmarkGenShortId()` 和 `BenchmarkGenShortIdTimeConsuming()`（详见 demo15/util/util_test.go）：

```
func BenchmarkGenShortId(b *testing.B) {
 for i := 0; i < b.N; i++ {
  GenShortId()
 }
}

func BenchmarkGenShortIdTimeConsuming(b *testing.B) {
 b.StopTimer() // 调用该函数停止压力测试的时间计数

 shortId, err := GenShortId()
 if shortId == "" || err != nil {
  b.Error(err)
 }

 b.StartTimer() // 重新开始时间

 for i := 0; i < b.N; i++ {
  GenShortId()
 }
}
```

说明

- 性能测试函数名必须以 Benchmark 开头，如 BenchmarkXxx 或 Benchmark_xxx
- `go test` 默认不会执行压力测试函数，需要通过指定参数 `-test.bench` 来运行压力测试函数，`-test.bench` 后跟正则表达式，如 `go test -test.bench=".*"` 表示执行所有的压力测试函数
- 在压力测试中，需要在循环体中指定 `testing.B.N` 来循环执行压力测试代码

执行压力测试

在 `util` 目录下执行命令 `go test -test.bench=".*"`：

```
$ go test -test.bench=".*"
goos: linux
goarch: amd64
pkg: apiserver/util
BenchmarkGenShortId-2                   500000       2291 ns/op
BenchmarkGenShortIdTimeConsuming-2      500000       2333 ns/op
PASS
ok   apiserver/util 2.373s
```

- 上面的结果显示，我们没有执行任何 TestXXX 的单元测试函数，只执行了压力测试函数
- 第一条显示了 `BenchmarkGenShortId` 执行了 500000 次，每次的执行平均时间是 2291 纳秒
- 第二条显示了 `BenchmarkGenShortIdTimeConsuming` 执行了 500000，每次的平均执行时间是 2333 纳秒
- 最后一条显示总执行时间

  `BenchmarkGenShortIdTimeConsuming` 比 `BenchmarkGenShortId` 多了两个调用 `b.StopTimer()` 和 `b.StartTimer()`。

  - `b.StopTimer()`：调用该函数停止压力测试的时间计数
  - `b.StartTimer()`：重新开始时间

  在 `b.StopTimer()` 和 `b.StartTimer()` 之间可以做一些准备工作，这样这些时间不影响我们测试函数本身的性能。

查看性能并生成函数调用图

1. 执行命令：

```
$ go test -bench=".*" -cpuprofile=cpu.profile ./util
```

上述命令会在当前目录下生成 `cpu.profile` 和 `util.test` 文件。

2. 执行 `go tool pprof util.test cpu.profile` 查看性能（进入交互界面后执行 `top` 指令）：

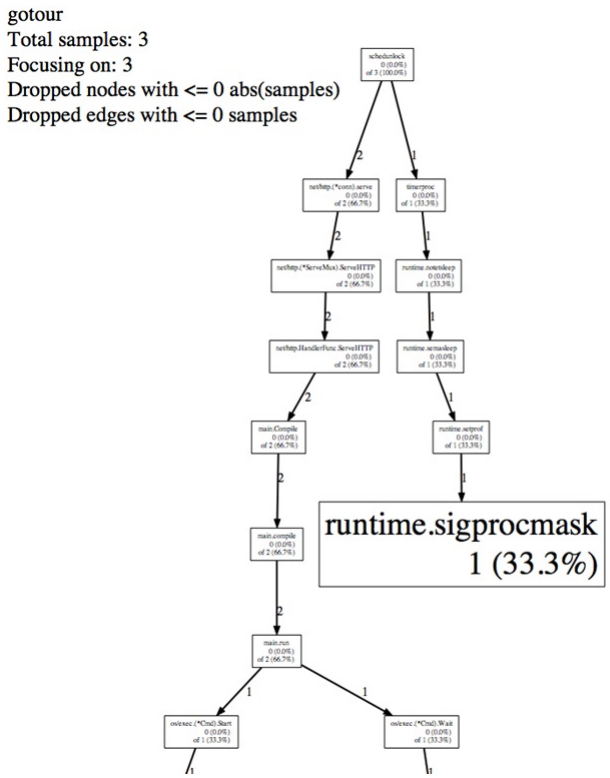
```
$ go tool pprof util.test cpu.profile
```

```
File: util.test
Type: cpu
Time: Jun 5, 2018 at 7:28pm (CST)
Duration: 4.93s, Total samples = 4.97s (100.78%)
Entering interactive mode (type "help" for commands, "o" for options)
(pprof) top
Showing nodes accounting for 3480ms, 70.02% of 4970ms total
Dropped 34 nodes (cum <= 24.85ms)
Showing top 10 nodes out of 75
      flat  flat%   sum%        cum   cum%
    1890ms 38.03% 38.03%     1900ms 38.23%  syscall.Syscall
     500ms 10.06% 48.09%      620ms 12.47%  runtime.mallocgc
     240ms  4.83% 52.92%     3700ms 74.45%  vendor/github.com/teris-io/shortid.(*Abc).Encode
     150ms  3.02% 55.94%      200ms  4.02%  runtime.scanobject
     140ms  2.82% 58.75%      640ms 12.88%  runtime.makeslice
     140ms  2.82% 61.57%      280ms  5.63%  runtime.slicerunetostring
     120ms  2.41% 63.98%      120ms  2.41%  math.Log
     110ms  2.21% 66.20%     2430ms 48.89%  io.ReadAtLeast
     110ms  2.21% 68.41%      110ms  2.21%  runtime._ExternalCode
      80ms  1.61% 70.02%      140ms  2.82%  runtime.deferreturn
(pprof)
```

pprof 程序中最重要的命令就是 `topN`，此命令用于显示 profile 文件中的最靠前的 N 个样本（sample），它的输出格式各字段的含义依次是：

1. 采样点落在该函数中的总时间
2. 采样点落在该函数中的百分比
3. 上一项的累积百分比
4. 采样点落在该函数，以及被它调用的函数中的总时间
5. 采样点落在该函数，以及被它调用的函数中的总次数百分比
6. 函数名

此外，在 pprof 程序中还可以使用 `svg` 来生成函数调用关系图（需要安装 graphviz），例如：



该调用图生成方法如下：

1. 安装 `graphviz` 命令

```
# yum -y install graphviz.x86_64
```

2. 执行 `go tool pprof` 生成 svg 图：

```
$ go tool pprof util.test cpu.profile
File: util.test
Type: cpu
Time: Jun 5, 2018 at 7:28pm (CST)
Duration: 4.93s, Total samples = 4.97s (100.78%)
Entering interactive mode (type "help" for commands, "o" for options)
(pprof) svg
Generating report in profile001.svg
```

`svg` 子命令会提示在 `$GOPATH/src` 目录下生成了一个 svg 文件 `profile001.svg`。

> 关于如何看懂 pprof 信息，请参考官方文档 Profiling Go Programs。
>
> 关于如何做性能分析，请参考郝林大神的文章 go tool pprof。

测试覆盖率

我们写单元测试的时候应该想得很全面，能够覆盖到所有的测试用例，但有时也会漏过一些 case，go 提供了 cover 工具来统计测试覆盖率。

`go test -coverprofile=cover.out`：在测试文件目录下运行测试并统计测试覆盖率

`go tool cover -func=cover.out`：分析覆盖率文件，可以看出哪些函数没有测试，哪些函数内部的分支没有测试完全，cover 工具会通过执行代码的行数与总行数的比例表示出覆盖率

测试覆盖率

```
$ go test -coverprofile=cover.out
PASS
coverage: 14.3% of statements
ok   apiserver/util 0.006s
[api@centos util]$ go tool cover -func=cover.out
apiserver/util/util.go:8: GenShortId 100.0%
apiserver/util/util.go:12: GetReqID 0.0%
total:    (statements) 14.3%
```

可以看到 `GenShortId()` 函数测试覆盖率为 100%，`GetReqID()` 测试覆盖率为 0%。

小结

本小节简单介绍了如何用 testing 包做单元和性能测试。在实际的开发中，要养成编写单元测试代码的好习惯，在项目上线前，最好对一些业务逻辑比较复杂的函数做一些性能测试，提前发现性能问题。

至于怎么去分析性能，比如查找耗时最久的函数等，笔者链接了郝林大神专业的分析方法（go tool pprof），更深的分析技巧需要读者在实际开发中自己去探索。

API 性能分析

作为开发，我们一般都局限在功能上的单元测试，对一些性能上的细节我们往往没有去关注，如果我们在上线的时候对项目整体性能没有一个全面的了解的话，当流量越来越大时，可能会出现各种各样的问题，比如 CPU 占用高、内存使用率高等。为了避免这些性能瓶颈，我们在开发的过程中需要通过一定的手段来对程序进行性能分析。

Go 语言已经为开发者内置配套了很多性能调优监控的好工具和方法，这大大提升了我们 profile 分析的效率，借助这些工具我们可以很方便地来对 Go 程序进行性能分析。在 Go 语言开发中，通常借助于内置的 pprof 工具包来进行性能分析。

本节核心内容

- 如何用 pprof 工具对 API 程序进行性能分析

  本小节源码下载路径：demo16

  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 demo15 来开发的。

pprof 是什么

PProf 是一个 Go 程序性能分析工具，可以分析 CPU、内存等性能。Go 在语言层面上集成了 profile 采样工具，只需在代码中简单地引入 `runtime/ppro` 或者 `net/http/pprof` 包即可获取程序的 profile 文件，并通过该文件来进行性能分析。

> runtime/pprof 还可以为控制台程序或者测试程序产生 pprof 数据。

其实 net/http/pprof 中只是使用 runtime/pprof 包来进行封装了一下，并在 HTTP 端口上暴露出来。

使用 pprof

在 gin 中使用 pprof 功能，需要用到 github.com/gin-contrib/pprof middleware，使用时只需要调用 pprof.Register() 函数即可。本例中，通过在router/router.go中添加如下代码来实现（详见 demo16/router/router.go）：

```
package router

import (
 "github.com/gin-contrib/pprof"
 ....
)

// Load loads the middlewares, routes, handlers.
func Load(g *gin.Engine, mw ...gin.HandlerFunc) *gin.Engine {
 // pprof router
 pprof.Register(g)
 ....
}
```

编译

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/demo16 复制为 $GOPATH/src/apiserver

```
$ cp -a apiserver_demos/demo16/ $GOPATH/src/apiserver
```

3. 在 apiserver 目录下编译源码

```
$ cd $GOPATH/src/apiserver
$ make
```

获取 profile 采集信息

通过 go tool pprof http://127.0.0.1:8080/debug/pprof/profile，可以获取 profile 采集信息并分析。

也可以直接在浏览器访问 http://localhost:8080/debug/pprof 来查看当前 API 服务的状态，包括 CPU 占用情况和内存使用情况等。

执行命令后，需要等待 30s，pprof 会进行采样。

性能分析

在上一小节我们介绍函数性能测试时已经介绍过性能分析的一部分知识，为了使内容完整，我们这里再次介绍下相关知识。

通过上一部分我们已经获取到了程序的 profile 信息，并且进入到了 pprof 的交互界面，在交互界面执行 topN 可以获取采样信息。

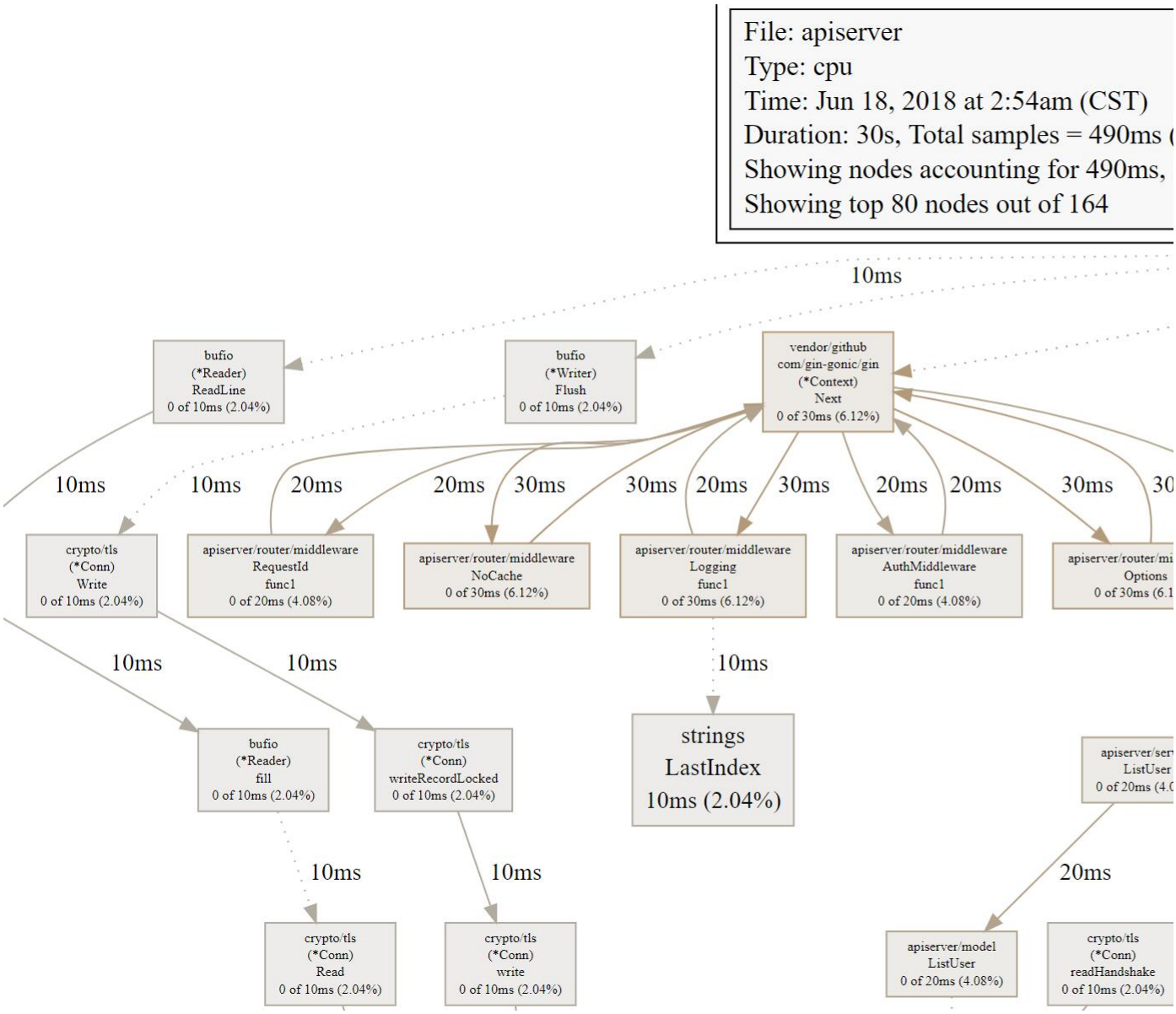
```
[api@centos src]$ go tool pprof http://127.0.0.1:8080/debug/ppr
Fetching profile over HTTP from http://127.0.0.1:8080/debug/ppr
Saved profile in /home/api/pprof/pprof.apiserver.samples.cpu.01
File: apiserver
Type: cpu
Time: Jun 18, 2018 at 2:54am (CST)
Duration: 30s, Total samples = 490ms ( 1.63%)
Entering interactive mode (type "help" for commands, "o" for op
(pprof) top10
Showing nodes accounting for 350ms, 71.43% of 490ms total
Showing top 10 nodes out of 164
      flat  flat%   sum%        cum   cum%
      90ms 18.37% 18.37%      220ms 44.90%  math/big.nat.montgo
      80ms 16.33% 34.69%       80ms 16.33%  math/big.addMulVVW
      50ms 10.20% 44.90%       50ms 10.20%  runtime.memmove
      40ms  8.16% 53.06%      110ms 22.45%  math/big.nat.divLar
      20ms  4.08% 57.14%       30ms  6.12%  math/big.getNat
      20ms  4.08% 61.22%       20ms  4.08%  math/big.mulAddVWW
      20ms  4.08% 65.31%       20ms  4.08%  runtime.futex
      10ms  2.04% 67.35%       10ms  2.04%  crypto/sha256.block
      10ms  2.04% 69.39%       10ms  2.04%  internal/poll.conve
      10ms  2.04% 71.43%      120ms 24.49%  math/big.nat.div
```

通过 topN 的输出可以分析出哪些函数占用 CPU 时间片最多，这些函数可能存在性能问题。性能分析详细防范请参考：

如果觉得不直观，可以直接生成函数调用图，通过调用图来判断哪些函数耗时最久，在 pprof 交互界面，执行 svg 生成 svg 文件。

```
(pprof) svg
Generating report in profile001.svg
```

用浏览器打开 profile001.svg：



框框最大的函数耗时比较久，说明函数可能存在性能问题。

> 确保系统已经安装 `graphviz` 命令。

小结

本节展示了如何对 API 服务进行性能分析，这里只是介绍了如何添加性能分析入口和基本的流程。

生成 Swagger 在线文档

本节核心内容

- 如何给 API 添加 Swagger 文档功能
- 如何编写 API 文档

> 本小节源码下载路径：demo17
>
> 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 demo16 来开发的。

背景

开发 API 服务，API 文档必不可少，很多人选择手写 API 文档，手写 API 文档有很多问题，比如工作量大、手写容易出错、更新麻烦、格式不固定、维护困难等。所以在实际的开发中，建议自动生成 API 文档。

本小册所用的 API 服务器 RESTful 框架采用的是 `gin`，`gin` 在 GitHub 上有很多 middleware 可用，其中就有可以自动生成 Swagger 文档的 `gin-swagger` middleware。

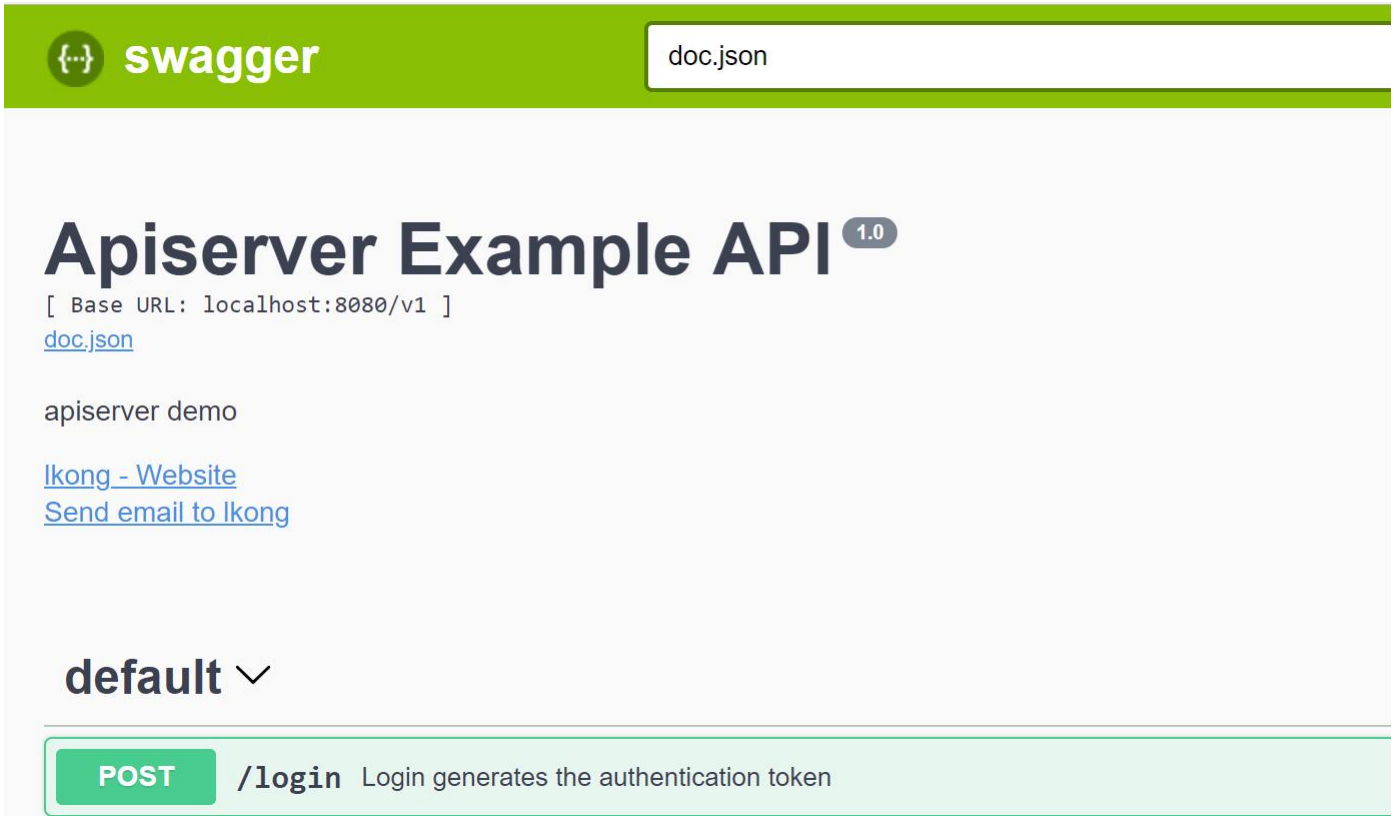
Swagger 简介

Swagger 是一个强大的 API 文档构建工具，可以自动为 RESTful API 生成 Swagger 格式的文档，可以在浏览器中查看 API 文档，也可以通过调用接口来返回 API 文档（JSON 格式）。Swagger 通常会展示如下信息：

1. HTTP 方法（GET、POST、PUT、DELETE 等）
2. URL 路径
3. HTTP 消息体（消息体中的参数名和类型）
4. 参数位置
5. 参数是否必选
6. 返回的参数（参数名和类型）
7. 请求和返回的媒体类型

Swagger 还有一个强大的功能：可以通过 API 文档描述的参数来构建请求，测试 API。

浏览器访问截图：



JSON 返回截图：

```
[api@centos user]$ curl http://127.0.0.1:8080/swagger/doc.json
{
    "swagger": "2.0",
    "info": {
        "description": "apiserver demo",
        "title": "Apiserver Example API",
        "contact": {
            "name": "lkong",
            "url": "http://www.swagger.io/support",
            "email": "466701708@qq.com"
        },
        "license": {},
        "version": "1.0"
    },
    "host": "localhost:8080",
    "basePath": "/v1",
    "paths": {
        "/login": {
            "post": {
                "produces": [
                    "application/json"
```

Swagger 配置步骤

我们用 gin-swagger gin middleware来生成 Swagger API 文档。步骤如下：

1. 安装 swag 命令

```
$ mkdir -p $GOPATH/src/github.com/swaggo
$ cd $GOPATH/src/github.com/swaggo
$ git clone https://github.com/swaggo/swag
$ cd swag/cmd/swag/
$ go install -v
```

因为该包引用 golang.org 中的包，而网络环境原因，一般很难连上 golang.org，所以这里不采用 go get 方式安装。

swag 的依赖包已经包含在第 4 节的 vendor 包下。

2. 进入 apiserver 根目录执行 `swag init`

```
$ cd $GOPATH/src/apiserver
$ swag init
```

3. 下载 `gin-swagger`

```
$ cd $GOPATH/src/github.com/swaggo
$ git clone https://github.com/swaggo/gin-swagger
```

4. 在 `router/router.go` 中添加 swagger 路由（详见 demo17/router/router.go）

```go
package router

import (
    "net/http"

    _ "apiserver/docs" // docs is generated by Swag CLI, you have to im
    "apiserver/handler/sd"
    "apiserver/handler/user"
    "apiserver/router/middleware"

    "github.com/gin-contrib/pprof"
    "github.com/gin-gonic/gin"
    "github.com/swaggo/gin-swagger"
    "github.com/swaggo/gin-swagger/swaggerFiles"
)

// Load loads the middlewares, routes, handlers.
func Load(g *gin.Engine, mw ...gin.HandlerFunc) *gin.Engine {
    // Middlewares.
    g.Use(gin.Recovery())
    g.Use(middleware.NoCache)
    g.Use(middleware.Options)
    g.Use(middleware.Secure)
    g.Use(mw...)
    // 404 Handler.
    g.NoRoute(func(c *gin.Context) {
        c.String(http.StatusNotFound, "The incorrect API route.")
    })

    // swagger api docs
    g.GET("/swagger/*any", ginSwagger.WrapHandler(swaggerFiles.Handler)
```

router.go

5. 编写 API 注释，Swagger 中需要将相应的注释或注解编写到方法上，再利用生成器自动生成说明文件

这里用创建用户 API 来举例，其它 API 请参考 demo17/handler/user 下的 API 文件。

```
package user

import (
    ...
)


// @Summary Add new user to the database
// @Description Add a new user
// @Tags user
// @Accept  json
// @Produce  json
// @Param user body user.CreateRequest true "Create a new user"
// @Success 200 {object} user.CreateResponse "{"code":0,"message":"OK","data":{"username":"kong"}}"
// @Router /user [post]
func Create(c *gin.Context) {
    ...
}
```

6. 执行 `swag init`，在 apiserver 根目录下生成 `docs` 目录

```
$ swag init
```

文档语法说明

- Summary：简单阐述 API 的功能
- Description：API 详细描述
- Tags：API 所属分类
- Accept：API 接收参数的格式
- Produce：输出的数据格式，这里是 JSON 格式
- Param：参数，分为 6 个字段，其中第 6 个字段是可选的，各字段含义为：
    1. 参数名称
    2. 参数在 HTTP 请求中的位置（body、path、query）
    3. 参数类型（string、int、bool 等）
    4. 是否必须（true、false）
    5. 参数描述

6. 选项，这里用的是 `default()` 用来指定默认值

- Success：成功返回数据格式，分为 4 个字段
  1. HTTP 返回 Code
  2. 返回数据类型
  3. 返回数据模型
  4. 说明
- 路由格式，分为 2 个字段：
  1. API 路径
  2. HTTP 方法

API 文档编写规则请参考 See Declarative Comments Format。

API 文档有更新时，要重新执行 `swag init` 并重新编译 apiserver。

编译并运行

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/demo17` 复制为 `$GOPATH/src/apiserver`

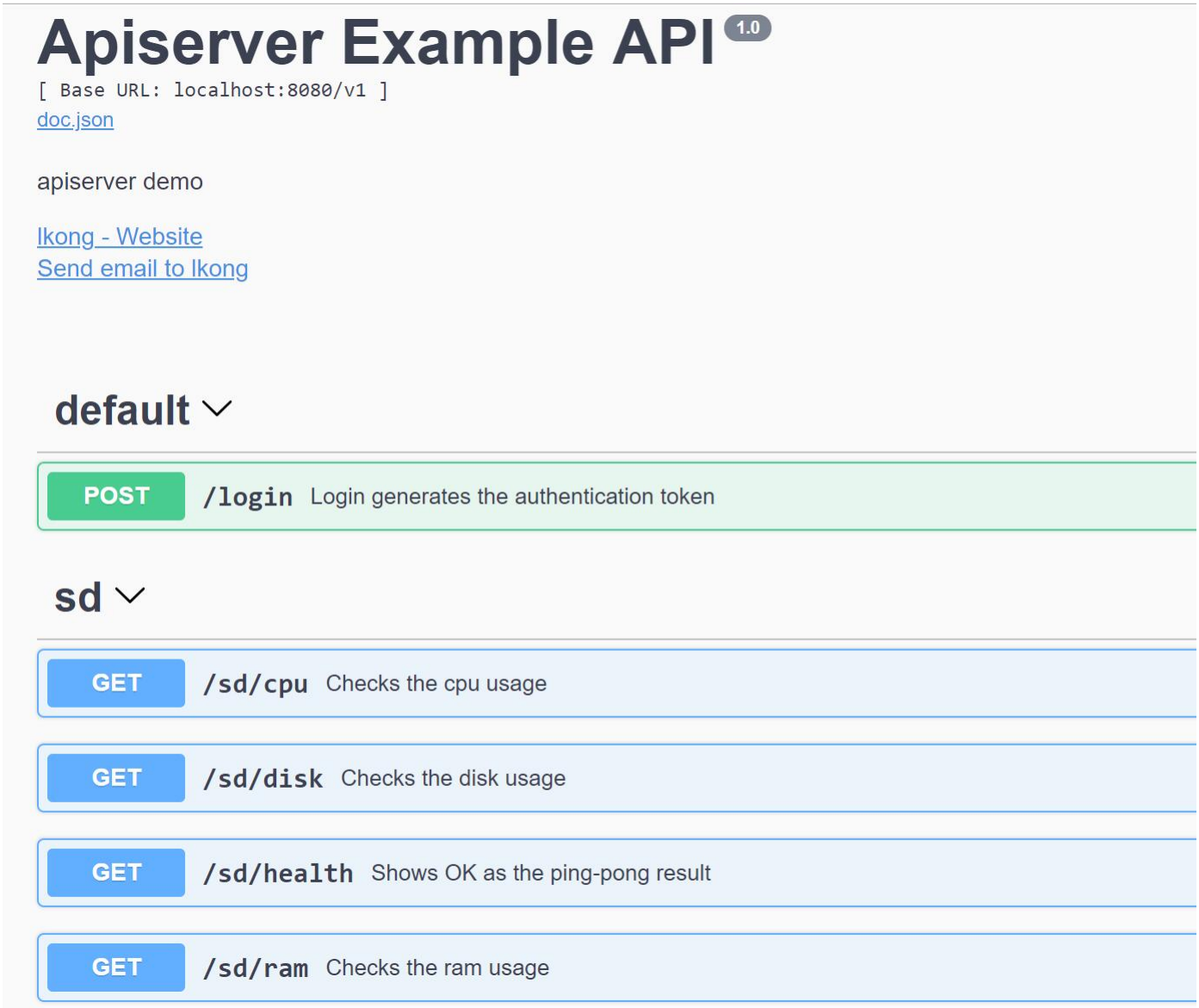
```
$ cp -a apiserver_demos/demo17/ $GOPATH/src/apiserver
```

3. 在 apiserver 目录下编译源码

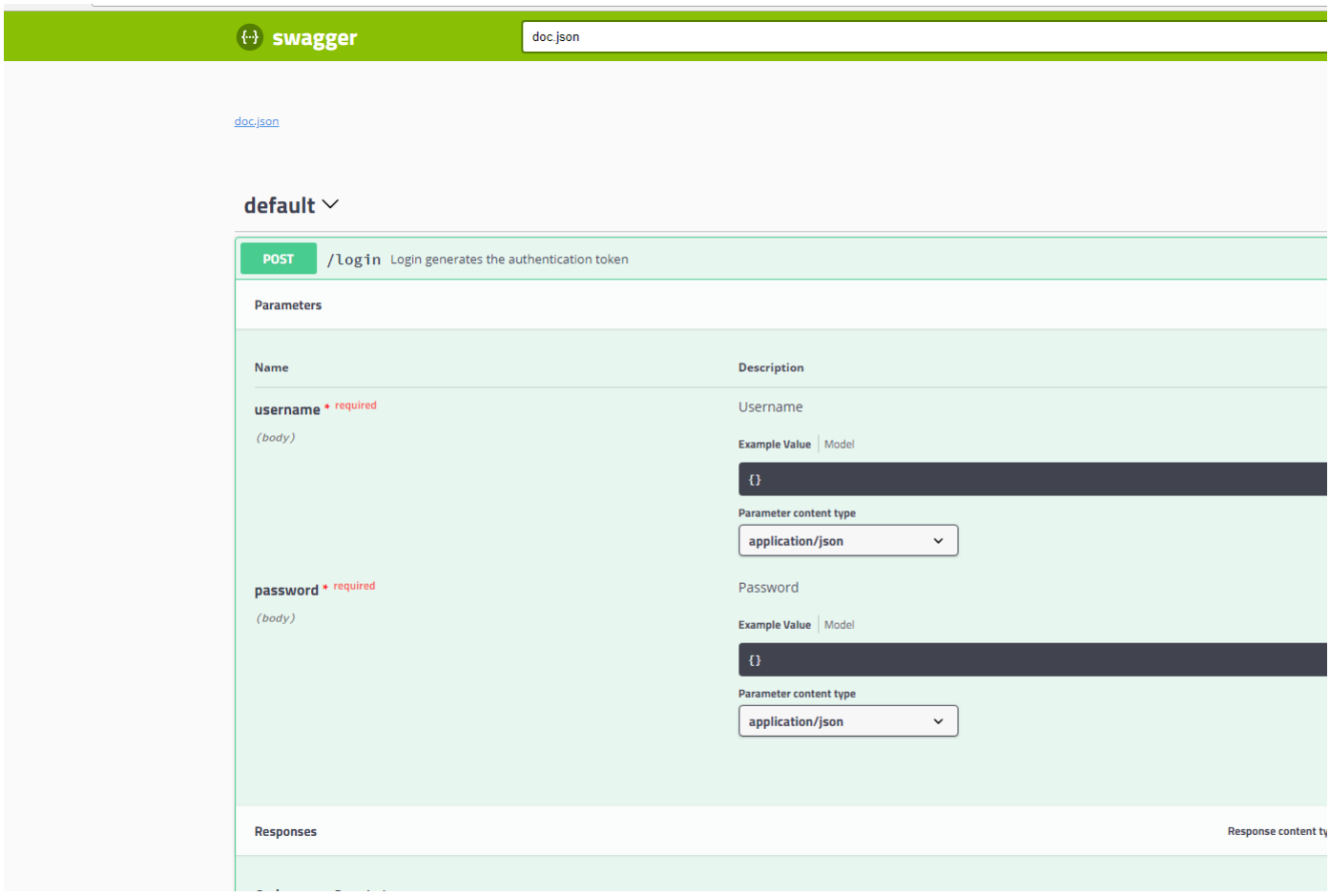
```
$ cd $GOPATH/src/apiserver
$ make
```

执行 `./apiserver` 启动 apiserver 后，在浏览器中打开： `http://localhost:8080/swagger/index.html` 访问 Swagger 2.0 API文档。

API 总览：



点击 `/login`，查看 login API 详情：



小结

本小节介绍了如何生成 Swagger 文档，并演示了具体的效果。本小节也是动手操作的最后一个小节，至此恭喜你成功构建了一个企业级的 API 服务器，demo17 即为此 API 服务器的最终源码。

API 性能测试和调优

本节核心内容

- 简单介绍 API 性能测试知识
- 介绍如何进行 API 性能测试
- 简单介绍如何进行 API 性能分析
- 给出 apiserver 的性能数据

本节最后会给出性能测试脚本：`wrktest.sh`，脚本见 `demo17/wrktest.sh`

性能测试

在 API 上线之前，我们需要知道 API 的性能，以便知道 API 服务器所能承载的最大请求量、性能瓶颈，再根据业务的需求量来对 API 进行性能调优或者扩缩容。通过这些可以使 API 稳定地对外提供服务，并且请求在合理的时间内返回。

API 性能测试指标

API 性能测试，大的方面包括 API 框架的性能和指定 API 的性能，因为指定 API 的性能跟该 API 具体的实现有关，比如有无数据库连接，有无复杂的逻辑处理等，脱离了具体实现来探讨单个 API 的性能是毫无意义的，所以本小节只探讨 API 框架的性能。

衡量 API 性能的指标主要有 3 个：

1. 并发数（Concurrent）

   并发数是指某个时间范围内，同时正在使用系统的用户个数。

   广义上的并发数是指同时使用系统的用户个数，这些用户可能调用不同的 API。严格意义上的并发数是指同时请求同一个 API 的用户个数。本小节所讨论的并发数是严格意义上的并发数。

2. 每秒查询数（QPS）

   每秒查询数 QPS 是对一个特定的查询服务器在规定时间内所处理流量多少的衡量标准。

   QPS = 并发数 / 平均请求响应时间。

3. 请求响应时间（TTLB）

   请求响应时间指的是从客户端发出请求到得到响应的整个时间。这个过程从客户端发起的一个请求开始，到客户端收到服务器端的响应结束。在一些工具中，请求响应时间通常会被称为 TTLB（Time to last byte，意思是从发送一个请求开始，到客户端收到最后一个字节的响应为止所消费的时间）。请求响应时间的单位一般为"秒"或"毫秒"。

衡量 API 性能的最主要指标是 QPS，但是在说明 QPS 时，需要指明是多少并发数下的 QPS，否则毫无意义，因为不同并发数下的 QPS 是不同的。比如单用户 100 QPS 和 100 用户 100 QPS 是两个不同的概念，前者说明 API 可以在一秒内串行执行 100 个请求，而后者说明在并发数为 100 的情况下，API 可以在一秒内处理 100 个请求。当 QPS 相同时，并发数越大，说明 API 性能越好，并发处理能力越强。

在并发数设置过大时，API 同时要处理很多请求，会频繁切换进程，而真正用于处理请求的时间变少，使得 QPS 反而会降低。并发数设置过大时，请求响应时间也会变大。API 会有一个合适的并发数，在该并发数下，API 的 QPS 可以达到最大，但该并发数不一定是最佳并发数，还要参考该并发数下的平均请求响应时间。

API 性能测试方法

Linux 下有很多 Web 性能测试工具，常用的有 Jmeter、AB、Webbench 和 Wrk。每个工具都有自己的特点，本小节用 Wrk 来对 API 进行性能测试。Wrk 非常简单，安装方便，测试结果也相对专业些，并且可以支持 Lua 脚本来创建更复杂的测试场景。

Wrk 安装

安装步骤如下（需要切换到 root 用户）：

1. Clone wrk repo

```
git clone https://github.com/wg/wrk
```

2. 执行 `make`和 `make install` 安装

```
make
cp ./wrk /usr/bin
```

Wrk 使用简介

Wrk 使用方法

Wrk 使用起来不复杂，执行 `wrk --help` 可以看到 wrk 的所有运行参数：

```
$ wrk --help
Usage: wrk <options> <url>
  Options:
    -c, --connections <N>  Connections to keep open
    -d, --duration    <T>  Duration of test
    -t, --threads     <N>  Number of threads to use

    -s, --script      <S>  Load Lua script file
    -H, --header      <H>  Add header to request
        --latency          Print latency statistics
        --timeout     <T>  Socket/request timeout
    -v, --version          Print version details

  Numeric arguments may include a SI unit (1k, 1M, 1G)
  Time arguments may include a time unit (2s, 2m, 2h)
```

常用的参数为：

- `-t`: 线程数（线程数不要太多，是核数的 2 到 4 倍即可，多了反而会因为线程切换过多造成效率降低）
- `-c`: 并发数
- `-d`: 测试的持续时间，默认为 10s
- `-T`: 请求超时时间
- `-H`: 指定请求的 HTTP Header，有些 API 需要传入一些 Header，可通过 Wrk 的 `-H` 参数来传入
- `--latency`: 打印响应时间分布
- `-s`: 指定 Lua 脚本，Lua 脚本可以实现更复杂的请求

Wrk 结果解析

一个简单的测试如下：

```
$ wrk -t144 -c3000 -d30s -T30s --latency http://127.0.0.1:8080/sd/health
Running 30s test @ http://127.0.0.1:8088/sd/health
  144 threads and 3000 connections
  Thread Stats   Avg      Stdev     Max   +/- Stdev
    Latency    32.01ms   39.32ms 488.62ms   87.93%
    Req/Sec     1.00k   251.79     3.35k    69.00%
  Latency Distribution
     50%   25.05ms
     75%   55.36ms
     90%   78.45ms
     99%  166.76ms
  4329733 requests in 30.10s, 1.81GB read
  Socket errors: connect 0, read 5, write 0, timeout 64
Requests/sec: 143850.26
Transfer/sec:     61.46MB
```

- `144 threads and 3000 connections`: 用 144 个线程模拟 3000 个连接，分别对应 `-t` 和 `-c` 参数
- `Thread Stats`： 线程统计
  - `Latency`: 响应时间，有平均值、标准偏差、最大值、正负一个标准差占比
  - `Req/Sec`: 每个线程每秒完成的请求数, 同样有平均值、标准偏差、最大值、正负一个标准差占比
- `Latency Distribution`: 响应时间分布
  - `50%`: 50% 的响应时间为： 4.74ms
  - `75%`: 75% 的响应时间为： 23.42ms
  - `90%`: 90% 的响应时间为： 82.88ms
  - `99%`: 99% 的响应时间为： 236.39ms
- `19373531 requests in 30.10s, 1.35GB read`: 30s 完成的总请求数（19373531）和数据读取量（1.35GB）
- `Socket errors: connect 0, read 5, write 0, timeout 64`：错误统计
- `Requests/sec`: QPS
- `Transfer/sec`: TPS

apiserver 第一次性能测试

测试服务器配置：6 核 12G

在 apiserver 中，Gin middleware： `Logging` 会记录请求参数和返回参数，该 middleare 很消耗性能，为了测试框架的性能，这里暂时将该 middleware 禁掉，在 main.go 函数中将 `middleware.Logging()` 一行注释掉，如图：

```
    // Set gin mode.
    gin.SetMode(viper.GetString("runmode"))

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

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

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

编译并运行 `apiserver`

```
$ make
$ ./apiserver
```

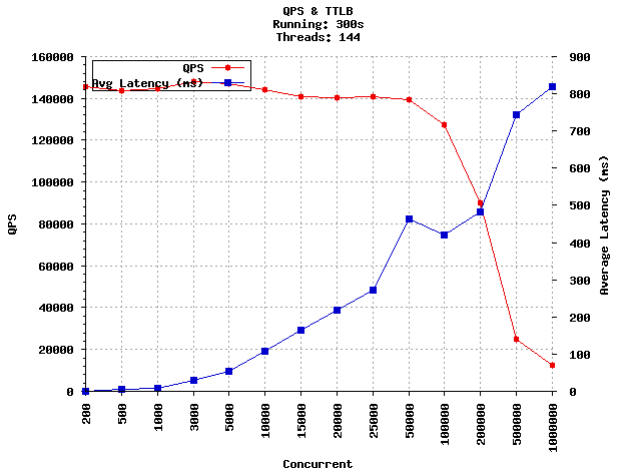
执行 wrk 命令测试 API 性能（分别测试多个并发数：`200 500 1000 3000 5000 10000 15000 20000 25000 50000 100000 200000 500000 1000000`）

```
$ wrk --latency -t144 -d60s -T300s http://127.0.0.1:8080/sd/health -c 200
```

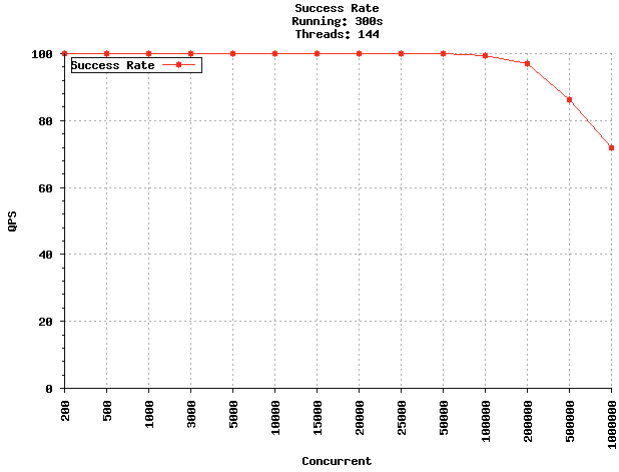
调用 `apiserver` 的健康检查接口：`/sd/health`

根据测试数据绘制出 QPS & TTLB 图和成功率图：

QPS & 平均响应时间:



成功率



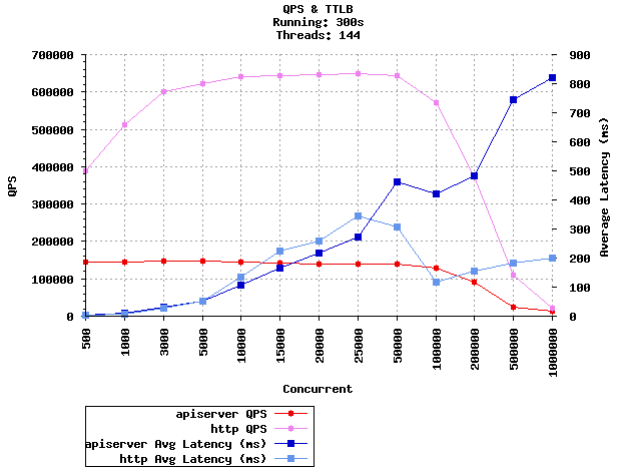
通过上面二图可以看到，apiserver 在并发数为 5000 时，QPS 最大，为 146953，平均响应时间为 52.75ms，在并发数达到 50000 时，成功率开始下降。

那么该 apiserver 的 QPS 处于什么水平呢？一方面可以根据自己的业务需要来对比，另一方面可以对比性能更好的 Web 框架。这里用 `net/http` 构建最简单的 HTTP 服务器，测试性能并作对比（相同的测试工具和测试服务器），HTTP 服务源码为：

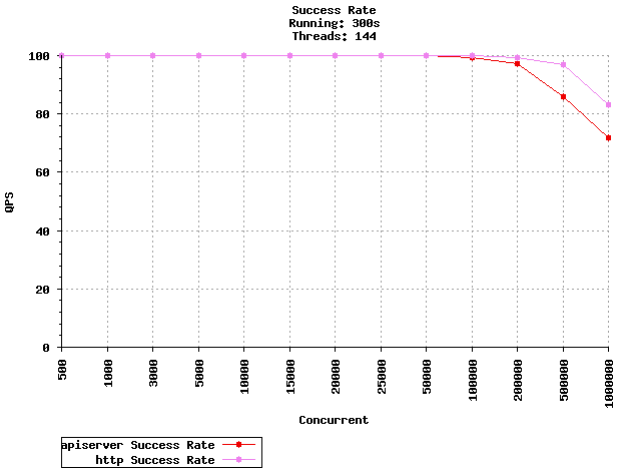
```
package main

import (
 "fmt"
 "log"
 "net/http"
)

func main() {
 http.HandleFunc("/", func(w http.ResponseWriter, r *http.Request) {
  message := "OK"
  fmt.Fprintln(w, "\n"+message)
 })

 log.Fatal(http.ListenAndServe(":6667", nil))
}
```

可以看到该 HTTP 服务器很简单，只是利用 `net/http` 包最原生的功能，在 Go 中几乎所有的 Web 框架都是基于 `net/http` 包封装的，既然是封装，相对于原生的性能肯定有所不及，所以这里拿 `net/http` 直接启动的 HTTP 服务器来做对比，对比结果如下：

QPS & 平均响应时间对比



成功率对比



通过上面两个对比图可以看出，apiserver 在 QPS、响应时间和成功率上都不如原生的 HTTP Server，特别是 QPS，最大 QPS 只有原生 HTTP Server 最大 QPS 的 22%，性能需要调优。

看到需要绘图，是不是觉得有点麻烦，不慌，笔者最后会奉上自动化测试脚本，该脚本会自动解析 wrk 结果并生成需要的图表。

apiserver 性能分析

API 性能分析涉及的范围比较广，本小节不是专门教读者如何进行详细的性能分析的教程，这里仅仅展示性能分析的步骤和基本思路。

在执行性能测试的过程中，运行 `go tool pprof http://127.0.0.1:8080/debug/pprof/profile`，采集 30s 的性能数据并查看耗时比较久的 20 个函数：

```
(pprof) top20
Showing nodes accounting for 1.57mins, 52.96% of 2.97mins total
Dropped 510 nodes (cum <= 0.01mins)
Showing top 20 nodes out of 199
      flat  flat%   sum%        cum   cum%
  0.37mins 12.35% 12.35%   0.37mins 12.35%  runtime.futex
  0.28mins  9.49% 21.84%   0.30mins 10.02%  syscall.Syscall
  0.12mins  4.10% 25.95%   0.30mins 10.27%  runtime.lock
  0.12mins  4.09% 30.04%   0.30mins 10.05%  runtime.mallocgc
  0.12mins  3.98% 34.02%   0.12mins  3.98%  runtime.epollwait
  0.09mins  2.94% 36.96%   0.09mins  2.94%  runtime.usleep
  0.08mins  2.74% 39.70%   0.96mins 32.43%  runtime.findrunnable
  0.06mins  2.01% 41.71%   0.15mins  4.94%  runtime.runqgrab
  0.04mins  1.31% 43.03%   0.04mins  1.31%  runtime.memmove
  0.03mins  1.14% 44.17%   0.03mins  1.14%  runtime.heapBitsSetType
  0.03mins  1.12% 45.29%   0.07mins  2.20%  runtime.scanobject
  0.03mins  1.07% 46.36%   0.03mins  1.07%  runtime.procyield
  0.03mins  1.03% 47.39%   0.03mins  1.03%  crypto/sha256.block
  0.03mins  0.98% 48.38%   0.26mins  8.66%  runtime.unlock
  0.03mins  0.88% 49.26%   0.03mins  0.88%  runtime.greyobject
  0.02mins  0.82% 50.08%   0.02mins  0.82%  runtime.osyield
  0.02mins  0.76% 50.84%   0.02mins  0.76%  runtime.runqempty
  0.02mins  0.72% 51.56%   0.03mins  0.92%  runtime.step
  0.02mins  0.71% 52.27%   0.05mins  1.54%  runtime.mapassign_faststr
  0.02mins  0.69% 52.96%   0.14mins  4.73%  runtime.netpoll
```

在 go tool pprof 文章中，我们知道，在默认情况下，top 命令输出的列表中只包含本地取样计数最大的前十个函数，统计的是这些函数本身运行的执行时间，实际上我们还需要知道，函数中有没有调用耗时的函数以及执行其它函数所耗费的时间，这种情况下我们需要按累积取样计数来排序，这在 pprof 中需要加上 `-cum` 参数：

```
(pprof) top -cum
Showing nodes accounting for 6.38s, 3.58% of 178.08s total
Dropped 510 nodes (cum <= 0.89s)
Showing top 10 nodes out of 199
      flat  flat%   sum%        cum   cum%
     0.31s  0.17%  0.17%    104.25s 58.54%  net/http.(*conn).serve
     0.08s 0.045%  0.22%     63.81s 35.83%  net/http.serverHandler.ServeHTTP
     0.09s 0.051%  0.27%     63.73s 35.79%  vendor/github.com/gin-gonic/gin.(*Engine).ServeHTTP
     0.15s 0.084%  0.35%     63.08s 35.42%  vendor/github.com/gin-gonic/gin.(*Engine).handleHTTPRequest
     0.09s 0.051%   0.4%     60.25s 33.83%  runtime.mcall
     0.22s  0.12%  0.53%     59.60s 33.47%  vendor/github.com/gin-gonic/gin.(*Context).Next
     0.04s 0.022%  0.55%     59.58s 33.46%  vendor/github.com/gin-gonic/gin.RecoveryWithWriter.func1
     0.50s  0.28%  0.83%     59.20s 33.24%  runtime.schedule
     0.02s 0.011%  0.84%     59.16s 33.22%  apiserver/router/middleware.NoCache
     4.88s  2.74%  3.58%     57.76s 32.43%  runtime.findrunnable
    ....
```

为了能够查看到所有函数的耗时排名，你需要列出更多的函数（本小节列出了 top100）。

如果你对代码很熟悉，通过最后一列的函数名，你应该可以定位到程序中所调用函数的位置，并进行优化。因为 top100 内容过多，这里筛选程序中所调用的函数（顺序不变）。

```
Showing nodes accounting for 67.31s, 37.80% of 178.08s total
Dropped 510 nodes (cum <= 0.89s)
Showing top 50 nodes out of 199
      flat  flat%   sum%        cum   cum%
     0.31s  0.17%  0.17%    104.25s 58.54%  net/http.(*conn).serve
     0.08s 0.045%  0.22%     63.81s 35.83%  net/http.serverHandler.ServeHTTP
     0.09s 0.051%  0.27%     63.73s 35.79%  vendor/github.com/gin-gonic/gin.(*Engine).ServeHTTP
     0.15s 0.084%  0.35%     63.08s 35.42%  vendor/github.com/gin-gonic/gin.(*Engine).handleHTTPRequest
     0.09s 0.051%   0.4%     60.25s 33.83%  runtime.mcall
     0.22s  0.12%  0.53%     59.60s 33.47%  vendor/github.com/gin-gonic/gin.(*Context).Next
     0.04s 0.022%  0.55%     59.58s 33.46%  vendor/github.com/gin-gonic/gin.RecoveryWithWriter.func1
     0.50s  0.28%  0.83%     59.20s 33.24%  runtime.schedule
     0.02s 0.011%  0.84%     59.16s 33.22%  apiserver/router/middleware.NoCache
     4.88s  2.74%  3.58%     57.76s 32.43%  runtime.findrunnable
     0.05s 0.028%  3.61%     56.76s 31.87%  runtime.park_m
     0.05s 0.028%  3.64%     56.72s 31.85%  apiserver/router/middleware.Options
     0.06s 0.034%  3.67%     55.32s 31.06%  apiserver/router/middleware.RequestId.func1
     0.07s 0.039%  3.71%     51.34s 28.83%  apiserver/router/middleware.AuthMiddleware.func1
     0.11s 0.062%  3.77%     51.17s 28.73%  apiserver/pkg/token.ParseRequest
     0.05s 0.028%  3.80%     26.59s 14.93%  apiserver/pkg/token.Parse
     0.07s 0.039%  3.84%     26.12s 14.67%  vendor/github.com/dgrijalva/jwt-go.Parse
     0.10s 0.056%  3.90%     26.05s 14.63%  vendor/github.com/dgrijalva/jwt-go.(*Parser).Parse
     0.29s  0.16%  4.06%     25.84s 14.51%  vendor/github.com/dgrijalva/jwt-go.(*Parser).ParseWithClaims
     ...
     0.05s 0.028% 36.28%     11.08s  6.22%  vendor/github.com/spf13/viper.GetString
     0.02s 0.011% 36.29%     11.03s  6.19%  vendor/github.com/spf13/viper.(*Viper).GetString
     0.12s 0.067% 36.36%     10.92s  6.13%  vendor/github.com/spf13/viper.(*Viper).Get
```

从上面，我们可以知道 apiserver 中函数耗时排名如下：

1. `apiserver/router/middleware.NoCache`: Gin middleware，强制浏览器不使用缓存
2. `apiserver/router/middleware.Options`: Gin middleware，跨域设置
3. `apiserver/router/middleware.RequestId.func1`: Gin middleware，记录 RequestId
4. `apiserver/router/middleware.AuthMiddleware.func1`: Gin middleware，JWT 认证
5. `apiserver/pkg/token.ParseRequest`: Token 功能，JWT 认证相关
6. `apiserver/pkg/token.Parse`: Token 功能，JWT 认证相关
7. `vendor/github.com/dgrijalva/jwt-go.Parse`: Token 功能，JW 认证相关
8. `vendor/github.com/dgrijalva/jwt-go.(*Parser).Parse`: Token 功能，JWT 认证相关
9. `vendor/github.com/dgrijalva/jwt-go.(*Parser).ParseWithClaims`: Token 功能，JWT 认证相关
10. `vendor/github.com/spf13/viper.GetString`: `pkg/token/token.go` 中获取 jwt_secret 的值
11. `vendor/github.com/spf13/viper.(*Viper).GetString`: `pkg/token/token.go` 中获取 jwt_secret 的值
12. `vendor/github.com/spf13/viper.(*Viper).Get`: `pkg/token/token.go`中获取 jwt_secret 的值

上面的列表中可以看到有 ServeHTTP 字样的函数，这些函数是 gin/http 自带的函数，需要的函数，无法进行优化，所以上述列表没有列出。可以看到主要是 Gin middleware 耗时较久，这里处理方法是删除不需要的 Gin middleware，删除 Middleware 如下：

1. middleware.RequestId（main.go 文件中）

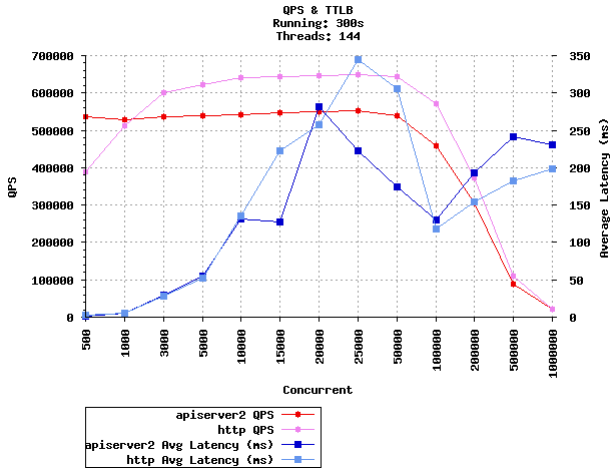
```
    // Routes.
    router.Load(
        // Cores.
        g,

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

2. middleware.NoCache 和 middleware.Options（router/router.go 文件中）

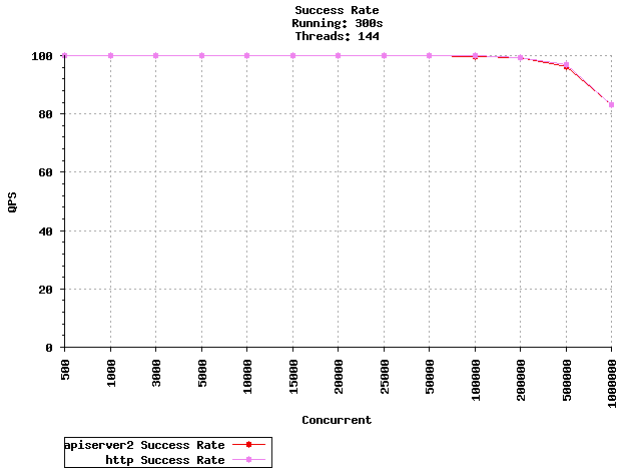
```
// Load loads the middlewares, routes, handlers.
func Load(g *gin.Engine, mw ...gin.HandlerFunc) *gin.Engine {
    // Middlewares.
    g.Use(gin.Recovery())
    //g.Use(middleware.NoCache)
    //g.Use(middleware.Options)
    g.Use(middleware.Secure)
    g.Use(mw...)
    // 404 Handler.
    g.NoRoute(func(c *gin.Context) {
        c.String(http.StatusNotFound, "The incorrect API route.")
    })
```

删除无用的 Gin middleware 重新编译 apiserver，启动 apiserver，测试性能后，再跟原生的 HTTP Server 对比，结果如下。

QPS & 平均响应时间对比



成功率对比



可以看到删除无用 Gin middleware 后，apiserver 的性能有了很大的提升，并发数为 25000 时，QPS 最大，为 553335（实际上并发数为 50000 时依然能达到很高的 QPS: 538144），响应时间为 222.91ms，QPS 很高，是原生 HTTP Server 的 85.34%。成功率基本跟原生的 HTTP Server 一致。优化后的 API 服务器可以支持很高的并发，在 20w+ 的并发下，API 服务器请求成功率可以达到 99.16%。

这些性能远远好于企业级 API 服务器的要求。

性能测试自动化

本小节的性能测试脚本请参考最终源码目录下的 `wrktest.sh` 脚本。脚本大致流程是：先执行 wrk 测试，收集测试数据，格式化测试数据，最后调用 gnuplot 生成图表。

确保系统安装了 gnuplot，如果没有安装，CentOS 系统中可通过如下命令安装：

```
yum -y install gnuplot
```

附件：API 性能测试数据

原生的 HTTP Server 性能数据

```
并发数 QPS 平均响应时间(毫秒) 成功率

200 107975.55 2.18 100.00
500 387894.92 2.52 100.00
1000 512223.67 5.89 100.00
3000 599781.96 27.75 100.00
5000 623458.30 52.72 100.00
10000 640701.55 134.92 100.00
15000 644269.17 222.44 100.00
20000 646675.63 257.28 100.00
25000 648380.78 344.17 100.00
50000 642420.16 305.24 99.99
100000 572197.78 118.41 99.84
200000 372247.81 154.68 99.31
500000 110261.20 181.90 96.90
1000000 20954.71 198.87 83.15
```

优化前 apiserver 性能数据

```
并发数 QPS 平均响应时间(毫秒) 成功率

200 145651.44 1.28 100.00
500 143562.75 4.30 100.00
1000 144860.93 8.79 100.00
3000 147833.70 30.91 100.00
5000 146953.40 52.75 100.00
10000 144015.46 108.19 100.00
15000 140960.03 164.76 100.00
20000 140586.00 218.57 100.00
25000 140783.38 272.12 100.00
50000 139312.92 462.26 99.93
100000 127629.61 419.98 99.28
200000 90035.14 483.31 97.14
500000 25118.75 743.60 86.08
1000000 12304.60 819.44 71.78
```

优化后 apiserver 性能数据

```
并发数 QPS 平均响应时间(毫秒) 成功率

200 540539.65 0.44213 100.00
500 536362.78 1.89 100.00
1000 529081.92 5.61 100.00
3000 535506.99 30.00 100.00
5000 539251.92 55.10 100.00
10000 541375.64 131.69 100.00
15000 547164.96 127.14 100.00
20000 550434.18 282.16 100.00
25000 553335.38 222.91 100.00
50000 538144.69 174.27 99.98
100000 457695.17 130.28 99.80
200000 305915.69 193.37 99.16
500000 87672.39 241.65 96.14
1000000 20351.67 231.12 82.99
```

总结

本小节介绍了如何进行 API 的性能测试，并给出了本小册 apiserver 的性能数据，最后笔者附上了自己测试用的自动化测试脚本。

本小节介绍的是框架的性能，具体到某个接口的性能，因为影响因素比较多，需要读者自己去优化，这里给出 HTTP 接口性能要求，供读者在优化时参考。

指标名称	要求	优先级
响应时间	500 ms	1
请求成功率	99%	2
QPS	在满足预期要求的情况下服务器状态稳定，单台服务器 QPS 要求在 1000+	3

Go 开发技巧

说明

本小节是拓展内容，笔者会不定期更新 Go 开发技巧，使该技巧的内容尽可能全，并保证技巧的实用性。

Go 开发技巧

1. package 的名字和目录名一样，main 除外
2. string 表示的是不可变的字符串变量，对 string 的修改是比较重的操作，基本上都需要重新申请内存，如果没有特殊需要，需要修改时多使用 []byte
3. 尽量使用 strings 库操作 string，这样做可以提高性能
4. append 要小心自动分配内存，append 返回的可能是新分配的地址
5. 如果要直接修改 map 的 value 值，则 value 只能是指针，否则要覆盖原来的值
6. map 在并发中需要加锁
7. 编译过程无法检查 interface{} 的转换，只有运行时检查，小心引起 panic
8. 使用 defer，保证退出函数时释放资源
9. 尽量少用全局变量，通过参数传递，使每个函数都是“无状态”的，这样减少耦合，也方便分工和单元测试
10. 参数如果比较多，将相关参数定义成结构体传递

Go 规范指南

说明

本小节是拓展内容，笔者会不定期更新 Go 规范指南，使该指南的内容尽可能全，并保证规范的实用性。

> 说明：本指南参考了网络上各种 REST 最佳实践，结合笔者的实际经验汇总而来。

Go 规范指南

1. 写完代码都必须格式化，保证代码优雅：gofmt goimports
2. 编译前先执行代码静态分析：go vet pathxxx/
3. package 名字：包名与目录保持一致，尽量有意义，简短，不和标准库冲突， 全小写，不要有下划线
4. 竞态检测：go build –race (测试环境编译时加上 -race 选项，生产环境必须去掉，因为 race 限制最多 goroutine 数量为 8192 个)
5. 每行长度约定：一行不要太长，超过请使用换行展示，尽量保持格式优雅；单个文件也不要太大，最好不要超过 500 行
6. 多返回值最多返回三个，超过三个请使用 struct
7. 变量名采用驼峰法，不要有下划线，不要全部大写
8. 在逻辑处理中禁用 panic，除非你知道你在做什么
9. 错误处理的原则就是不能丢弃任何有返回 err 的调用，不要采用_丢弃，必须全部处理。接收到错误，要么返回 err，要么实在不行就 panic，或者使用 log 记录下来。 不要这样写:

```
if err != nil {
    // error handling
} else {
```

```
    // normal code

}
```

而应该是:

```
if err != nil {

    // error handling

    return // or continue, etc.
}

// normal code
```

10. 常用的首字母缩写名词，使用全小写或者全大写，如 UIN URL HTTP ID IP OK
11. Receiver:：用一两个字符，能够表示出类型，不要使用 me self this
12. 参数传递:

- 对于少量数据，不要传递指针
- 对于大量数据的 struct 可以考虑使用指针
- 传入参数是 map，slice，chan，interface，string 不要传递指针

13. 每个基础库都必须有实际可运行的例子, 基础库的接口都要有单元测试用例
14. 不要在 for 循环里面使用 defer，defer只有在函数退出时才会执行
15. panic 捕获只能到goroutine最顶层，每个自己启动的 goroutine，必须在入口处就捕获panic，并打印出详细的堆栈信息
16. Go 的内置类型slice、map、chan都是引用，初次使用前，都必须先用 make 分配好对象，不然会有空指针异常
17. 使用 map 时需要注意：map 初次使用，必须用 make 初始化；map 是引用，不用担心赋值内存拷贝；并发操作时，需要加锁；range 遍历时顺序不确定，不可依赖；不能使用 slice、map 和 func 作为 key
18. import 在多行的情况下，goimports 会自动帮你格式化，但是我们这里还是规范一下 import 的一些规范，如果你在一个文件里面引入了一个 package，还是建议采用如下格式:

```
import (
    "fmt"
)
```

如果你的包引入了三种类型的包，标准库包，程序内部包，第三方包，建议采用如下方式进行组织你的包:

```
import (
    "encoding/json"
    "strings"

    "myproject/models"
    "myproject/controller"
    "myproject/utils"

    "github.com/astaxie/beego"
    "github.com/go-sql-driver/mysql"
)
```

有顺序的引入包，不同的类型采用空格分离，第一种实标准库，第二是项目包，第三是第三方包。

19. 如果你的函数很短小，少于 10 行代码，那么可以使用，不然请直接使用类型，因为如果使用命名变量很容易引起隐藏的 bug。 当然如果是有多个相同类型的参数返回，那么命名参数可能更清晰:

```
func (f *Foo) Location() (float64, float64, error)
```

20. 长句子打印或者调用，使用参数进行格式化分行 我们在调用 `fmt.Sprint` 或者 `log.Sprint` 之类的函数时，有时候会遇到很长的句子，我们需要在参数调用处进行多行分割:

下面是错误的方式:

```
log.Printf("A long format string: %s %d %d %s", myStringParameter, len(a),
    expected.Size, defrobnicate("Anotherlongstringparameter",
        expected.Growth.Nanoseconds() /1e6))
```

应该是如下的方式:

```
log.Printf(
    "A long format string: %s %d %d %s",
    myStringParameter,
    len(a),
    expected.Size,
    defrobnicate(
        "Anotherlongstringparameter",
        expected.Growth.Nanoseconds()/1e6,
    ),
)
```

21. 注意闭包的调用 在循环中调用函数或者 goroutine 方法，一定要采用显示的变量调用，不要在闭包函数里调用循环的参数

```
fori:=0;i<limit;i++{
    go func(){ DoSomething(i) }() //错误的做法
    go func(i int){ DoSomething(i) }(i)//正确的做法
}
```

22. recieved 是值类型还是指针类型 到底是采用值类型还是指针类型主要参考如下原则:

```
func(w Win) Tally(playerPlayer)int    //w不会有任何改变
func(w *Win) Tally(playerPlayer)int    //w会改变数据
```

23. struct 声明和初始化格式采用多行： 定义如下:

```
type User struct{
    Username  string
    Email     string
}
```

初始化如下:

```
u := User{
    Username: "astaxie",
    Email:    "astaxie@gmail.com",
}
```

24. 变量命名

- 和结构体类似，变量名称一般遵循驼峰法，首字母根据访问控制原则大写或者小写，但遇到特有名词时，需要遵循以下规则:
    - 如果变量为私有，且特有名词为首个单词，则使用小写，如 apiClient
    - 其它情况都应当使用该名词原有的写法，如 APIClient、repoID、UserID
    - 错误示例：UrlArray，应该写成 urlArray 或者 URLArray
- 若变量类型为 bool 类型，则名称应以 Has、Is、Can 或 Allow 开头

```
var isExist bool
var hasConflict bool
var canManage bool
var allowGitHook bool
```

25. 常量命名 常量均需使用全部大写字母组成，并使用下划线分词

```
const APP_VER = "1.0"
```

总结

至此，我们已完成了本小册的学习。通过前面各小节的学习，相信你已经对 API 开发的整个流程有了一定的了解，也知道如何去构建流程中的每个功能。本小册是一个入门教程，更复杂的场景，需要在具体的业务开发中，由读者去探索。这是一个大型的实战类小册，笔者耗费了不少精力来构建这本小册，也希望这本小册确实能够帮助到大家。Go 语言越来越火，希望通过本小册的学习，能使你及早搭上 Go 的顺风车。

最后感谢大家的关注，小册有些不完善的地方还请大家多多包容，祝大家工作顺利，事事顺意。