使用 go-zero 快速开发微服务

读者画像及收益

本文预设的读者画像：

- 【必须】对微服务有基本概念，比如知道微服务的使用场景，了解过微服务框架，了解服务发现，gRPC 等
- 【必须】熟练掌握单体 web 服务开发，熟悉用户、登录等常见模块的开发，了解 Restful-api
- 【必须】熟悉 Golang 语法，可能在 Golang 生态中一直没找到趁手的工具快速进行微服务开发
- 【可选】了解 Vue，有一定的前端基础

读者收益：

阅读本文后，初步掌握使用 go-zero 进行 CRDU 级别需求的开发。

框架对比

简单提一下我在选型时，了解过的其他框架：

- Go-micro, v3 版本与该公司的云平台绑定了要注册账号才能使用；v2 版本已经不维护了，连 go1.6 也不支持；
- Go-kit, 是一系列工具包，算不上框架；

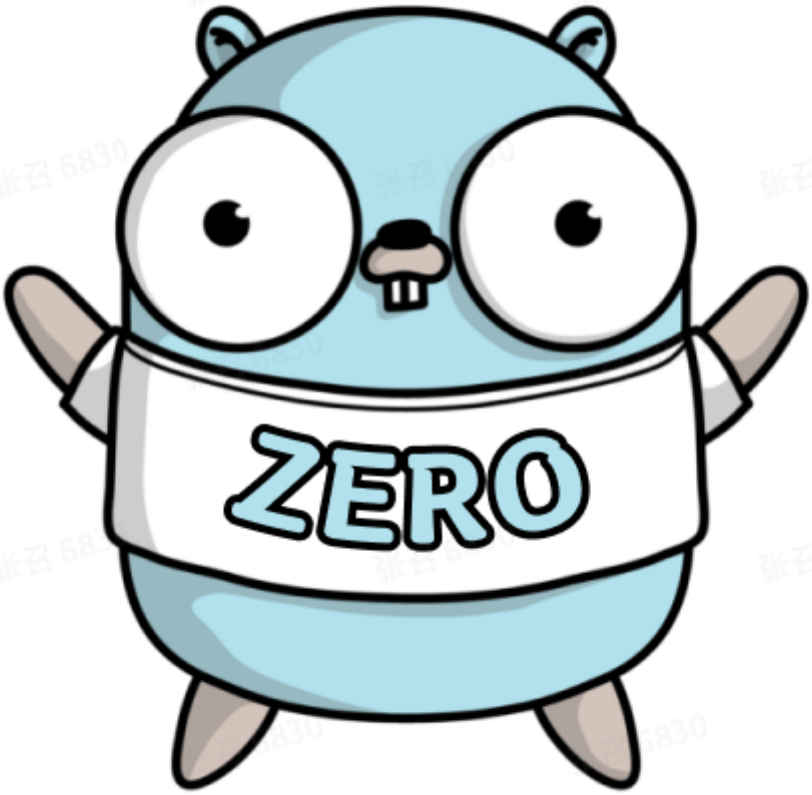
环境配置

在开始具体演示前，先介绍一下当前的环境配置。

- 系统开发环境：Macos
- Golang 版本：go version go1.16.2 darwin/amd64
- 编辑器：Goland 或 Vscode，安装 “go-zero” 插件提高编辑体验。
- goctl version: 1.1.6 darwin/amd64，必选，go-zero 脚手架工具。安装文档
- Mysql: 5.7，必选，blog 项目数据库。
- Redis: 4.0.10，必选，blog 项目缓存层。
- etcd Version: 3.4.1，必选，blog 项目服务发现。

本文涉及到的代码及相关资料都上传到该仓库 go-zero-demo

1. go-zero 速览

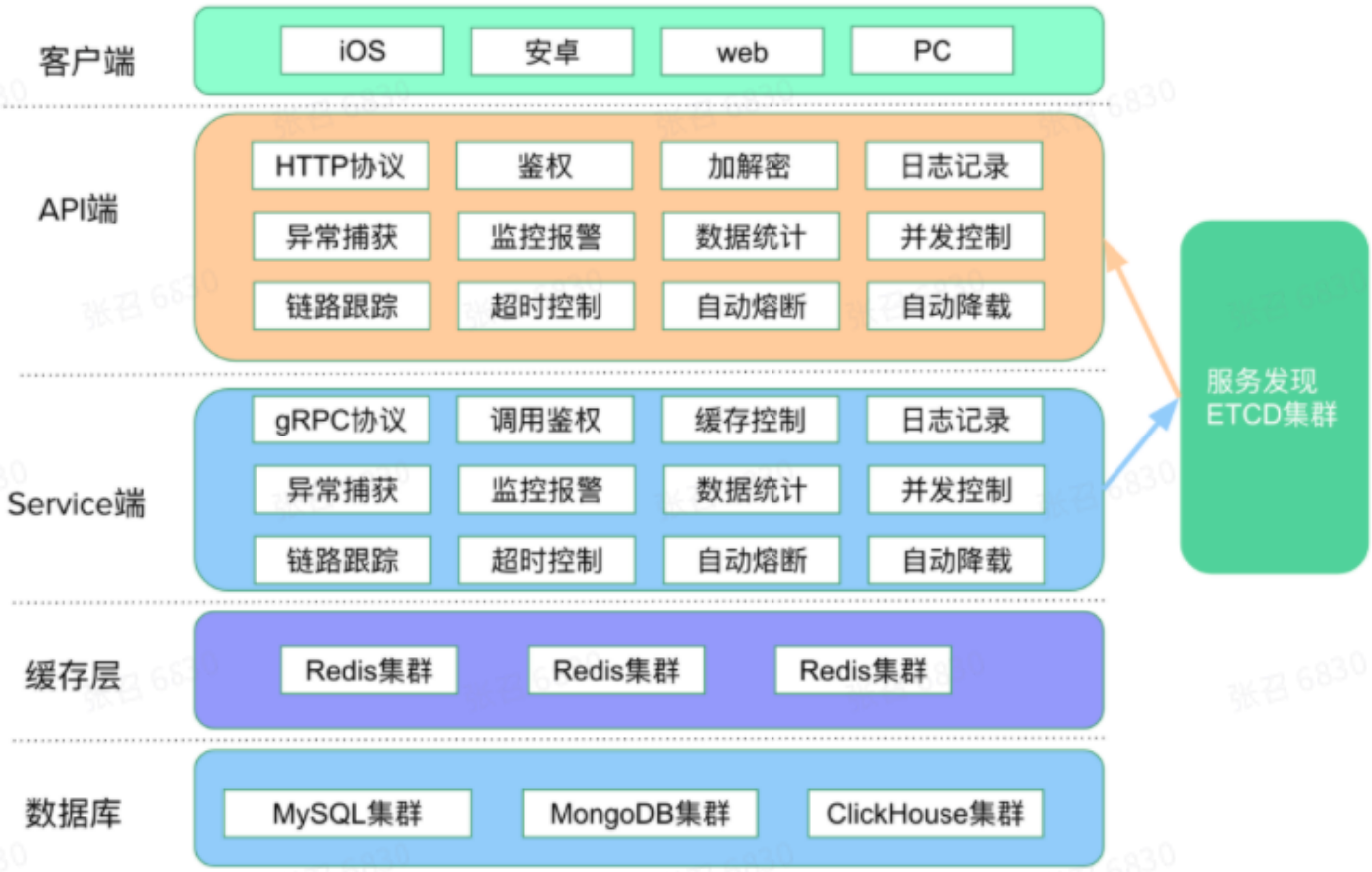


1.1 go-zero 中有哪些特性适合微服务开?

- 强大的项目脚手架工具 goctl。 goctl 和前端中的 Vue-cli、React-cli 一样方便。 goctl 通过配置文件可以生成 api、rpc 和 model 等相关代码。
- 较完备的项目框架。脚手架生成的项目框架足以应对常见的需求。CRDU 等需求只需要做做 “填空题”，在已生成的代码上填充必要的业务逻辑。 其他缓存鉴权等需求，框架中也早已内置。
- “渐进式”框架。 “渐进式”是前端 Vue 框架的一大特性，大意是“易于上手，还便于与第三方库或既有项目整合”。 本文借用这个概念是想表明 go-zero 对项目的入侵性较少，go-zero 生成的代码可以拆开使用，逐步对老项目进行改造。
- 低耦合的模块设计，丰富的中间件，插件和工具:
  - go-zero 中各模块耦合程度低，我们可以通过文档中的组件中心寻找合适的中间件或自研中间件。
  - 如果觉得 goctl 不能满足需求，goctl 还支持 plugin 命令对 goctl 进行扩展。

- go-zero 的很多配置文件是自定义语法。 go-zero 还提供了 intellij 和 vscode 插件，提供了语法高亮错误检查等编辑增强功能。

1.2 什么情况不适宜使用 go-zero 做微服务开发?



A. 当前需求与 goctl 的理念相冲突

go-zero 的一大卖点是脚手架工具 goctl，如果定制需求过多可能与 goctl 生成的代码相冲突。但是如果放弃 goctl 手动编写代码的话，开发效率会大大降低。

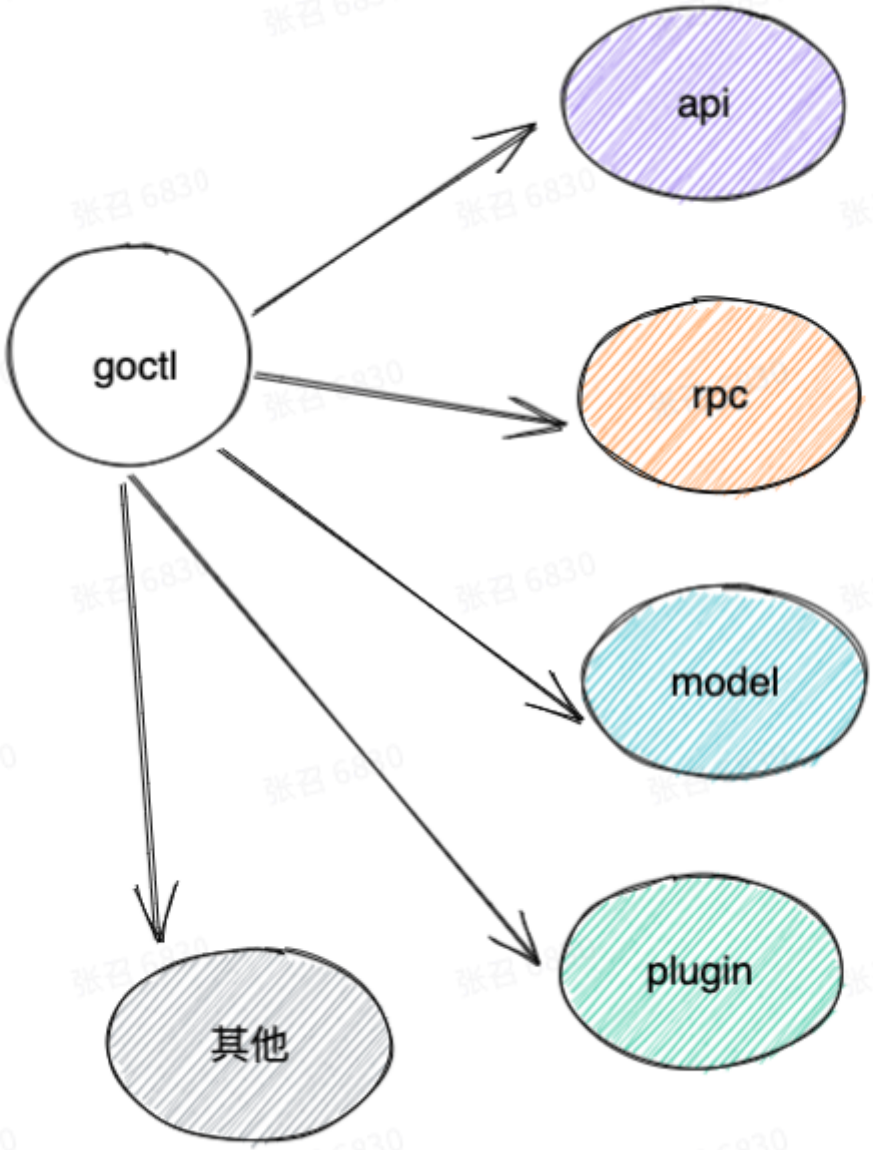
举个例子，如上图所示， go-zero 在 RPC 层目前只支持 gRPC，在数据库层只支持 Mysql、MongoDB 和 ClickHouse，服务发现只支持 ETCD。在这种情况下如果想实现 PostgreSQL 替换 Mysql、Consul 替换 ETCD 等定制操作，goctl 生成的代码执行时很可能会出现异常。

B. 希望框架提供的功能非常完善

go-zero 大部分组件是自研，比如 sqlx，httpx 等。这些自研组件满足 CRDU 的操作绰绰有余，但是与 gorm、gin 等专攻某一方向的开源项目相比还是有非常大的差距的。

所以随着公司业务发展需求越来越五花八门，当前的主要矛盾从“快速开发”变成“精细化开发”时，会发现该框架有这样或那样的不足。这种情况下就需要提 RP 或自己 fork 一份魔改了。个人觉得这种情况比 Spring 或 Django 那样一个“全家桶” 改动起来要省力省心。

2. goctl 介绍



goctl 的命令可归纳为如下几类：

- api命令，快速生成一个api服务
- rpc命令，支持proto模板生成和rpc服务代码生成
- model命令，目前支持识别mysql ddl进行model层代码生成
- plugin命令，支持针对api自定义插件
- 其他命令，目前是发布相关

goctl 的命令众多，本次涉及到的只是其中 api、rpc 和 model 相关的基础命令。

3. *go-zero 实战项目：blog*

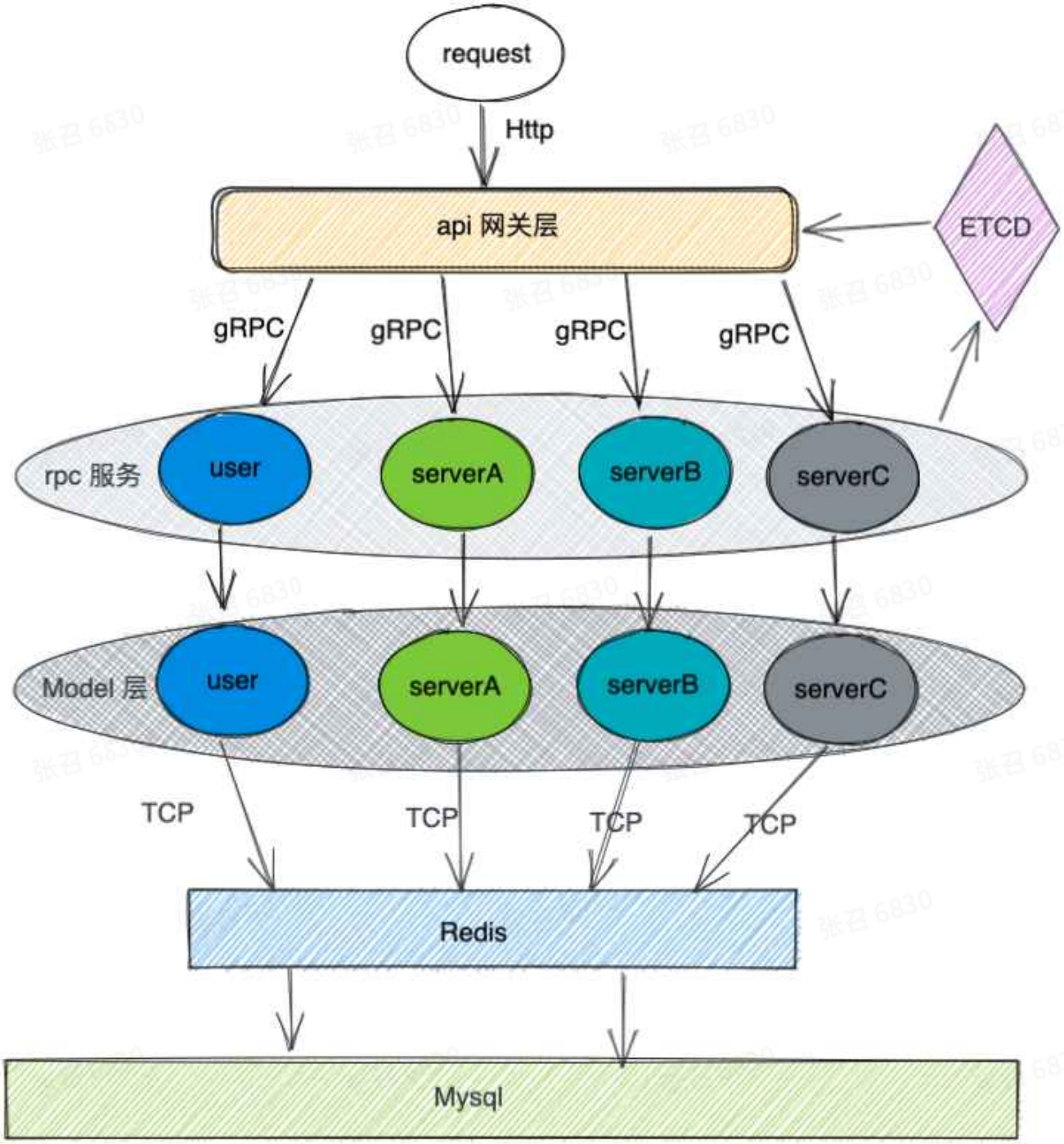
用户模块是后台管理系统常见的模块，它的功能大家也非常熟悉。 管理用户涉及到前端操作，用户信息持久化又离不开数据库。 所以用户模块可谓是 "麻雀虽小五脏俱全"。 下面演示如何使用 go-zero 开发一个 *blog 系统的*用户模块。

3.1 *用户模块功能及 Api*

用户模块包含的功能及大致如下 （鉴于文章篇幅考虑完整的 Api 文档请参考 gitee 上的仓库）：

- 用户登录
- 添加用户
- 删除用户
- 修改用户
- 查询用户

3.2 整体架构



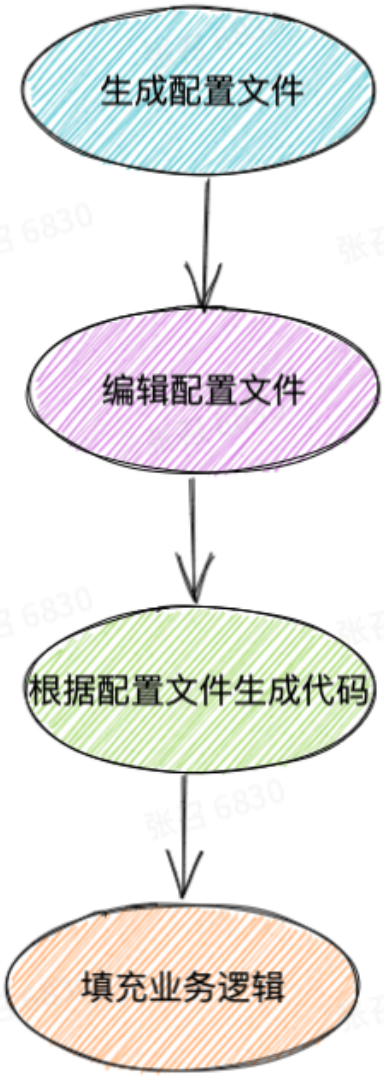
blog 系统整体架构如上图所示。

- api 网关层:
  - go-zero 需要 api 网关层来代理请求，把 request 通过 gRPC 转发给对应的 rpc 服务去处理。
  - 这块把具体请求转发到对应的 rpc 服务的业务逻辑，需要手写。
- rpc 服务:
  - 上图 rpc 服务中的 user 就是接下来向大家演示的模块。

- 每个 rpc 服务可以单独部署。服务启动后会把相关信息注册到 ETCD，这样 api 网关层就可以通过 ECTD 发现具体服务的地址。
- rpc 服务处理具体请求的业务逻辑，需要手写。

- Model 层:
  - model 层封装的是数据库操作的相关逻辑。
  - 如果是查询类的相关操作，会先查询 redis 中是否有对应的缓存。非查询类操作，则会直接操作 MySQL。
  - goctl 能通过 sql 文件生成普通的 CRDU 代码。上文也有提到，目前 goctl 这部分功能只支持 MySQL。

4. 使用 goctl 的基本流程



使用 goctl 生成代码的流程大致可以分为 4 步：

1. 使用命令 a 生成默认的配置文件；
2. 按照业务需求编辑该配置文件；
3. 使用命令 b 按照配置文件生成默认的代码文件；
4. 按照业务逻辑填充对应的代码文件。

5. api 服务

5.1 编写 blog.api 文件

A. 生成 blog.api 文件

执行命令 `goctl api -o blog.api`，创建 blog.api 文件。

B. api 文件的作用

api 文件的详细语法请参阅文档，本文按照个人理解谈一谈 api 文件的作用和基础语法。

api 文件是用来生成 api 网关层的相关代码的。

C. api 文件的语法

api 文件的语法和 Golang 语言非常类似，`type` 关键字用来定义结构体，`service` 部分用来定义 api 服务。

`type` 定义的结构体，主要是用来声明请求的入参和返回值的，即 request 和 response.

`service` 定义的 api 服务，则声明了路由，handler，request 和 response.

具体内容请结合下面的默认的生成的 api 文件进行理解。

Plain Text

```
// 声明版本，可忽略
syntax = "v1"

// 声明一些项目信息，可忽略
info(
    title: // TODO: add title
    desc: // TODO: add description
    author: "zhao.zhang"
    email: "zhao.zhang@upai.com"
)

// 重要配置
// request 是结构体的名称，可以使用 type 关键词定义新的结构体
type request {
    // TODO: add members here and delete this comment
    // 与 golang 语言一致，这里声明结构体的成员
}

// 语法同上，只是业务含义不同。response 一般用来声明返回值。
type response {
    // TODO: add members here and delete this comment
}

// 重要配置
// blog-api 是 service 的名称.
service blog-api {
    // GetUser 是处理请求的视图函数
    @handler GetUser // TODO: set handler name and delete this comment
    // get 声明了该请求使用 GET 方法
    // /users/id/:userId 是 url，:userId 表明是一个变量
    // request 就是上面 type 定义的那个 request，是该请求的入参
    // response 就是上面 type 定义的那个 response，是该请求的返回值。
    get /users/id/:userId(request) returns(response)

    @handler CreateUser // TODO: set handler name and delete this comment
    post /users/create(request)
}
```

D. 编写 blog.api 文件

鉴于文章篇幅考虑完整的 blog.api 文件请参考 gitee 上的仓库。下面生成的代码是按照仓库上的 blog.api 文件生成的。

5.2 api 相关代码

A. 生成相关的代码

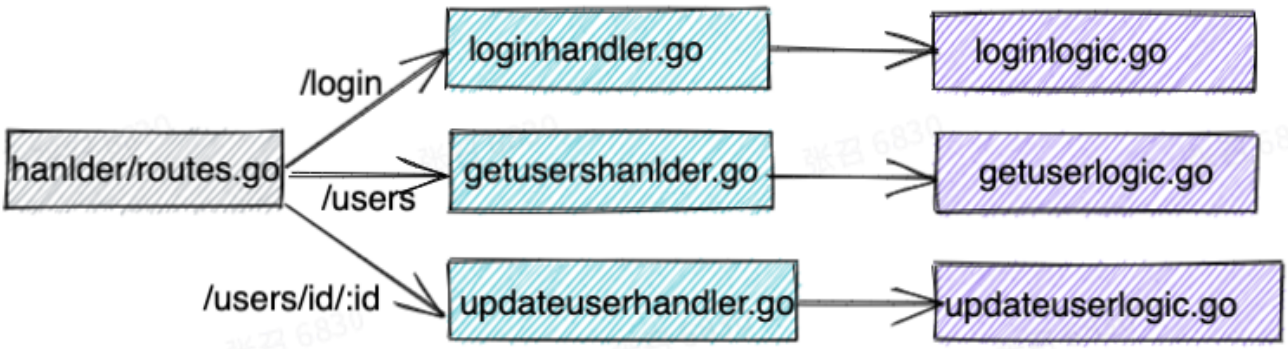
执行命令 `goctl api go -api blog.api -dir .` ，生成 api 相关代码。

B. 目录介绍

CSS

```
├──── blog.api # api 文件
├──── blog.go # 程序入口文件
├──── etc
│    └──── blog-api.yaml # api 网关层配置文件
├──── go.mod
├──── go.sum
└──── internal
     ├──── config
     │    └──── config.go # 配置文件
     ├──── handler # 视图函数层，handler 文件与下面的 logic 文件一一对应
     │    ├──── adduserhandler.go
     │    ├──── deleteuserhandler.go
     │    ├──── getusershandler.go
     │    ├──── loginhandler.go
     │    ├──── routes.go
     │    └──── updateuserhandler.go
     ├──── logic # 需要手动填充代码的地方
     │    ├──── adduserlogic.go
     │    ├──── deleteuserlogic.go
     │    ├──── getuserslogic.go
     │    ├──── loginlogic.go
     │    └──── updateuserlogic.go
     ├──── svc # 封装 rpc 对象的地方，后面会将
     │    └──── servicecontext.go
     └──── types # 把 blog.api 中定义的结构体映射为真正的 golang 结构体
          └──── types.go
```

C. 文件间的调用关系



因为到此时还没涉及到 rpc 服务，所以 api 内各模块的调用关系就是非常简单的单体应用间的调用关系。routers.go 是路由，根据 request Method 和 url 把请求分发到对应到的 handler 上，handler 内部会去调用对应的 logic. logic 文件内是我们注入代码逻辑的地方。

D. 为什么 logic 文件前要封装一层 handler?

- Handler 部分封装是请求方式和请求入参；
- Logic 部分是纯粹的业务逻辑。

这样分离的好处是，当接口的请求方式发生变化时，可以修改 api 文件删除对应的 handler 文件，然后重新生成 handler 文件，对应的 logic 文件不会受到影响。

6. rpc 服务

6.1 编写 proto 文件

A. 生成 user.proto 文件

使用命令 `goctl rpc template -o user.proto`，生成 user.proto 文件

B. user.proto 文件的作用

user.proto 的作用是用来生成 rpc 服务的相关代码。

protobuf 的语法已经超出了 go-zero 的范畴了，这里就不详细展开了。

C. 编写 user.proto 文件

鉴于文章篇幅考虑完整的 user.proto 文件请参考 gitee 上的仓库。

6.2 生成 rpc 相关代码

A. 生成 user rpc 服务相关代码

使用命令 `goctl rpc proto -src user.proto -dir .` 生成 user rpc 服务的代码。

7. api 服务调用 rpc 服务

A. 为什么本节要安排在 rpc 服务的后面?

因为 logic 部分的内容主体就是调用对应的 user rpc 服务，所以我们必须要在 user rpc 的代码已经生成后才能开始这部分的内容。

B. api 网关层调用 rpc 服务的步骤

对这部分目录结构不清楚的，可以参考 “5.2.B 目录介绍”。

a. 编辑配置文件 `etc/blog-api.yaml`，配置 rpc 服务的相关信息。

YAML

```yaml
Name: blog-api
Host: 0.0.0.0
Port: 8888
# 新增 user rpc 服务.
User:
  Etcd:
#  Hosts 是 user.rpc 服务在 etcd 中的 value 值
    Hosts:
      - localhost:2379
# Key 是 user.rpc 服务在 etcd 中的 key 值
    Key: user.rpc
```

b. 编辑文件 `config/config.go`

Go

```go
type Config struct {
   rest.RestConf
   // 手动添加
   // RpcClientConf 是 rpc 客户端的配置，用来解析在 blog-api.yaml 中的配置
   User zrpc.RpcClientConf
}
```

c. 编辑文件 `internal/svc/servicecontext.go`

Go

```go
type ServiceContext struct {
    Config config.Config
    // 手动添加
    // users.Users 是 user rpc 服务对外暴露的接口
    User   users.Users
}

func NewServiceContext(c config.Config) *ServiceContext {
    return &ServiceContext{
        Config: c,
        // 手动添加
        //  zrpc.MustNewClient(c.User) 创建了一个 grpc 客户端
        User:   users.NewUsers(zrpc.MustNewClient(c.User)),
    }
}
```

d. 编辑各个 logic 文件，这里以 `internal/logic/loginlogic.go` 为例

Go

```go
func (l *LoginLogic) Login(req types.ReqUser) (*types.RespLogin, error) {
    // 调用 user rpc 的 login 方法
    resp, err := l.svcCtx.User.Login(l.ctx, &users.ReqUser{Username:
req.Username, Password: req.Password})
    if err != nil {
        return nil, err
    }
    return &types.RespLogin{Token: resp.Token}, nil
}
```

8. model 层

8.1 编写 sql 文件

编写创建表的 SQL 文件 user.sql, 并在数据库中执行。

SQL

```sql
CREATE TABLE `user`
(
  `id` int NOT NULL AUTO_INCREMENT COMMENT 'id',
  `username` varchar(255) NOT NULL UNIQUE COMMENT 'username',
  `password` varchar(255) NOT NULL COMMENT 'password',
  PRIMARY KEY(`id`)
) ENGINE=InnoDB DEFAULT CHARSET=utf8mb4
```

8.2 生成 model 相关代码

运行命令 `goctl model mysql ddl -c -src user.sql -dir .`, 会生成操作数据库的 CRDU 的代码。

此时的 model 目录：

CSS

```
├─── user.sql # 手写
├─── usermodel.go # 自动生成
└─── vars.go # 自动生成
```

8.3 model 生成的代码注意点

- Model 这块代码使用的是拼接 SQL 语句，可能会存在 SQL 注入的风险。
- 生成 CRUD 的代码比较初级，需要我们手动编辑 `usermodel.go` 文件，自己拼接业务需要的 SQL。参见 usermdel.go 中的 `FindByName` 方法。

9. rpc 调用 model 层的代码

9.1 rpc 目录结构

Rpc 服务我们只需要关注下面加注释的文件或目录即可。

CSS

```
├─── etc
│    └─── user.yaml # 配置文件，数据库的配置写在这
├─── internal
│    ├─── config
│    │    └─── config.go # config.go 是 yaml 对应的结构体
│    ├─── logic # 填充业务逻辑的地方
│    │    ├─── createlogic.go
│    │    ├─── deletelogic.go
│    │    ├─── getalllogic.go
│    │    ├─── getlogic.go
│    │    ├─── loginlogic.go
│    │    └─── updatelogic.go
│    ├─── server
│    │    └─── usersserver.go
│    └─── svc
│         └─── servicecontext.go # 封装各种依赖
├─── user
│    └─── user.pb.go
├─── user.go
├─── user.proto
└─── users
     └─── users.go
```

9.2 rpc 调用 model 层代码的步骤

A. 编辑 `etc/user.yaml` 文件

YAML

```yaml
Name: user.rpc
ListenOn: 127.0.0.1:8080
Etcd:
  Hosts:
  - 127.0.0.1:2379
  Key: user.rpc
# 以下为手动添加的配置
# mysql 配置
DataSource: root:1234@tcp(localhost:3306)/gozero
# 对应的表
Table: user
# redis 作为换存储
Cache:
  - Host: localhost:6379
```

B. 编辑 `internal/config/config.go` 文件

Go

```go
type Config struct {
// zrpc.RpcServerConf 表明继承了 rpc 服务端的配置
   zrpc.RpcServerConf
   DataSource string          // 手动代码
   Cache      cache.CacheConf // 手动代码
}
```

C. 编辑 `internal/svc/servicecontext.go`, 把 model 等依赖封装起来。

Go

```go
type ServiceContext struct {
   Config config.Config
   Model  model.UserModel // 手动代码
}

func NewServiceContext(c config.Config) *ServiceContext {
   return &ServiceContext{
      Config: c,
      Model:  model.NewUserModel(sqlx.NewMysql(c.DataSource), c.Cache), // 手动代码
   }
}
```

D. 编辑对应的 logic 文件，这里以 `internal/logic/loginlogic.go` 为例：

Go

```go
func (l *LoginLogic) Login(in *user.ReqUser) (*user.RespLogin, error) {
   // todo: add your logic here and delete this line
   one, err := l.svcCtx.Model.FindByName(in.Username)
   if err != nil {
      return nil, errors.Wrapf(err, "FindUser %s", in.Username)
   }

   if one.Password != in.Password {
      return nil, fmt.Errorf("user or password is invalid")
   }

   token := GenTokenByHmac(one.Username, secretKey)
   return &user.RespLogin{Token: token}, nil
}
```

10. 小结

goctl 命令小结

Api 层相关命令：

- 执行命令 `goctl api -o blog.api`，创建 blog.api 文件。
- 执行命令 `goctl api go -api blog.api -dir .` ，生成 api 相关代码。

- 加参数 goctl 也可以生成其他语言的 api 层的文件，比如 java、ts 等，尝试之后发现很难用，所以不展开了。

rpc 服务相关：

- 使用命令 `goctl rpc template -o user.proto`，生成 user.proto 文件
- 使用命令 `goctl rpc proto -src user.proto -dir .` 生成 user rpc 服务的代码。

Model 相关：

- 运行命令 `goctl model mysql ddl -c -src user.sql -dir .`，会生成操作数据库的 CRDU 的代码。

目录结构及常用文件

- 配置文件：`etc/xx.yaml`
- 配置文件对应的结构体：`interval/config/config.go`
- 封装外部依赖的文件：`interval/svc/xxcontext.go`
- 填充业务逻辑的文件：`interval/logic`

微服务演示运行

我们是在单机环境下运行整个微服务，需要启动以下服务：

- Redis
- Mysql
- Etcd
- `go run blog.go -f etc/blog-api.yaml`
- `go run user.go -f etc/user.yaml`

在上述服务中，rpc 服务要先启动，然后网关层再启动。

在仓库中我封装了 start.sh 和 stop.sh 脚本来分别在单机环境下运行和停止微服务。