Golang-backend-skeleton

Golang-backend-skeleton

- 项目基本环境
- 项目意义
  - 简化开发流程
  - 定义代码规范
- 代码骨架
- 结构解析
  - 项目入口： `main.go`
  - 项目路由： `router.go`
  - 配置文件夹： `conf`
  - MySQL表结构文件夹： `ddl`
  - 业务逻辑文件夹： `biz`
  - 异常检测模块

项目基本环境

```
go 1.16.2
gin v1.7.2
gorm.io/gorm v1.21.10
```

请确保已经安装好Go环境，并 `pull` 项目最新版本。

在项目根目录下输入： `go get` 下载依赖

同样在根目录下输入： `go build` 进行编译

编译完成后，会在项目目录下生成一个基于操作系统的可执行文件，直接打开即可启动服务

如果你已经配置好环境并正常启动了服务，请调用测试API： `127.0.0.1:8080/ping` ，如果成功返回如下数据，则环境配置完成

```
{"data":"pong","message":"success","status":0}
```

项目意义

简化开发流程

这是一个基于Golang语言，GIN Web开发框架的灵活的后端骨架，这不是一个成型的后端服务！

我们希望这套代码骨架可以最大限度地帮助程序员减少环境配置成本，以最快的速度初始化一个后端项目，把更多精力直接投入到业务逻辑当中

定义代码规范

在实际开发过程当中，同样的功能可以有无数种实现方式，每个程序员也会有自己独特的开发习惯。但结构良好的实现对后续的进一步开发以及维护有着无形的帮助并产生巨大收益，这种收益会随着项目的不断扩张而成指数型增长。我们希望可以利用本套后端骨架的结构设计，在严格约束每个模块允许实现的功能的条件下，创造出结构清晰、逻辑严谨、代码复用性高的优秀后端项目。

代码骨架

```
.
├── biz
|   ├── config
|   ├── consts
|   |   └── consts.go
|   ├── dal
|   |   ├── db_dal
|   |   |   └── user_info_dal.go
|   |   └── rds_dal
|   ├── handler
|   |   ├── common_handler.go
|   |   ├── error_handler.go
|   |   └── personal_info_handler.go
|   ├── middleware
|   |   └── cors_config.go
|   ├── model
|   |   ├── dao
|   |   |   ├── db_init.go
|   |   |   └── userinfo_dao.go
|   |   └── dto
|   |       └── userinfo_dto.go
|   ├── service
|   |   └── personal_info_service.go
|   └── utils
|       └── config_yaml.go
├── conf
|   └── config.yaml
├── ddl
|   └── user_info.sql
├── go.mod
├── go.sum
├── main.go
└── router.go
```

结构解析

项目入口：`main.go`

- Web服务配置选用默认配置
- 建立与MySQL的连接
- 定义路由（API）
- 启动Web服务

```
func main() {
    r := gin.Default()
    r.Use(middleware.Cors())
    dao.ConnectDB()
    customizeRegister(r)
    if err := r.Run(); err != nil {
        log.Fatal("Initialize server failed, error: ", err)
    }
}
```

项目路由：`router.go`

```
func customizeRegister(r *gin.Engine) {
   r.GET("/ping", handler.Ping)
   root := r.Group("/api/v1")
   {
      root.GET("/get-user-info", handler.SearchUserInfoHandler)
      root.GET("/get-navigation")
   }
}
```

配置文件夹：`conf`

项目往往有一些全局的配置文件，在这个代码骨架中，我们就将MySQL的配置信息保存在了配置文件夹中（如果你需要在你的环境建立与MySQL的连接，请务必修改此处配置）。我们会在项目的某些地方调用这些信息（比如我们会在启动Web服务前连接MySQL数据库，就需要调用这里的信息）

MySQL表结构文件夹：`ddl`

```
/*
user_info.sql
表结构为具体的逻辑设计提供参照
*/
CREATE TABLE `user_info` (
  `id` int unsigned NOT NULL AUTO_INCREMENT COMMENT '自增id',
  `name` varchar(255) COLLATE utf8mb4_general_ci DEFAULT NULL COMMENT '姓名',
  `age` int DEFAULT NULL COMMENT '年龄',
```

```
  `gender` tinyint DEFAULT NULL COMMENT '性别',
  `tel-number` varchar(255) COLLATE utf8mb4_general_ci DEFAULT NULL COMMENT '联系电话',
  `email` varchar(255) COLLATE utf8mb4_general_ci DEFAULT NULL COMMENT '电子邮箱',
  `company` varchar(255) COLLATE utf8mb4_general_ci DEFAULT NULL COMMENT '所在单位',
  `university` varchar(255) COLLATE utf8mb4_general_ci DEFAULT NULL COMMENT '毕业院校',
  `create_time` datetime DEFAULT NULL COMMENT '创建时间',
  `update_time` datetime DEFAULT NULL ON UPDATE CURRENT_TIMESTAMP COMMENT '更新时间',
  PRIMARY KEY (`id`)
) ENGINE=InnoDB DEFAULT CHARSET=utf8mb4 COLLATE=utf8mb4_general_ci;
```

业务逻辑文件夹：`biz`

`biz`文件夹下的文件是项目主题逻辑部分，主要分为以下几个部分

- `handler`文件夹：处理器层，一般来说，每一个API对应一个`handler`，用于接收与解析从前端传来的请求，调用服务层的函数，并返回响应

```
func SearchUserInfoHandler(c *gin.Context) {
   userId, err := strconv.ParseInt(c.Query("id"), 10, 64)
   if err != nil {
      log.Println("Cannot parse user_id to integer type correctly. Error: ", err)
      GenerateEmptyDataResponse(c, err)
   }
   user, err := service.SearchUserInfoById(userId)
   GenerateResponse(c, user, err)
}
```

- `service`文件夹：服务层，由`handler`层的函数直接调用，服务层会调用`dal`层的函数从数据源头（如：MySQL）获取数据，实现数据解耦，封装成响应体回传`handler`层

```
func SearchUserInfoById(userId int64) (dto.UserInfoResponse, error) {
   var response dto.UserInfoResponse
   user, err := db_dal.SearchUserInfo(userId)
   if err != nil {
      return response, err
   }
   response = dto.UserInfoResponse{
      Id:         user.Id,
      Name:       user.Name,
      TelNumber:  user.TelNumber,
      Email:      user.Email,
      Company:    user.Company,
      University: user.University,
   }
   return response, err
}
```

- `dal` 文件夹：数据访问层（data access layer），由 `service` 层的函数调用，数据访问层会直接建立与（泛）数据库的联系，并将查询结果回传给 `service` 层的函数

```
func SearchUserInfo(userId int64) (dao.UserInfo, error) {
   var user dao.UserInfo
   err := dao.DB.Where("id = ?", userId).First(&user).Error
   return user, err
}
```

- `middleware` 文件夹：中间件，在这个代码骨架当中，我们使用了一个用于跨域资源共享（CORS）的中间件帮助我们实现前后端分离
- `consts` 文件夹：用于存放项目中需要使用到的常量（在项目代码当中，切忌直接使用形如： `if number == 0` 的代码，这里的 `0` 被称作 `magic number` ，只有你自己知道代码中的 `0` 表示什么，甚至在你开发一个月以后你自己都忘了它表示什么……，如果需要进行类似这种判断，请声明一个枚举对象或常量来存储此类标签）

```
// 这里我定义了一个新的类型用于表示性别，1表示男性，2表示女性
type GenderType int8
const (
   Male GenderType = iota + 1
   Female
   Trans
)
```

- `model` 文件夹：数据模型文件夹，主要存放数据访问对象DAO（data access object）和数据传输对象DTO（data transfer object）结构体，数据访问对象与底层数据表密切关联，数据传输对象与前端的请求与响应密切关联。在我们的实际开发过程中，大多数情况下我们是从底层数据拿到一个DAO结构，并在 `service` 层将其解析为DTO结构（解耦）回传前端

```
// DAO
type UserInfo struct {
   Id         int64             `gorm:"column:id" json:"id"`
   Name       string            `gorm:"column:name" json:"name"`
   Gender     consts.GenderType `gorm:"column:gender" json:"gender"`
   Age        int8              `gorm:"column:age" json:"age"`
   TelNumber  string            `gorm:"column:tel-number" json:"tel-number"`
   Email      string            `gorm:"column:email" json:"email"`
   Company    string            `gorm:"column:company" json:"company"`
   University string            `gorm:"column:university" json:"university"`
   CreateTime time.Time         `gorm:"column:create_time" json:"create_time"`
   UpdateTime time.Time         `gorm:"column:update_time" json:"update_time"`
}
```

```
// DTO
type UserInfoResponse struct {
    Id         int64  `gorm:"column:id" json:"id"`
    Name       string `gorm:"column:name" json:"name"`
    TelNumber  string `gorm:"column:tel-number" json:"tel-number"`
    Email      string `gorm:"column:email" json:"email"`
    Company    string `gorm:"column:company" json:"company"`
    University string `gorm:"column:university" json:"university"`
}
```

- `utils` 文件夹：工具类文件夹，主要存放项目中需要使用到的，又与项目数据结构不直接关联的函数，此类函数往往是对一些数据进行处理，如：获取当前时间与某条数据创建时间之差。在这个代码骨架中，我们写了一个解析 `yaml` 类型配置文件的工具类，生成用于连接数据库的 `dsn` 字段。工具类中的函数不应与DAO层或是DTO层建立直接联系（传入与传出均不应是定义在 `model` 中的结构体）

异常检测模块

在实际项目中，我们需要追踪所有可能产生异常的位置，当调用某个接口产生异常时，我们必须能够立即感知并迅速定位至异常产生的位置。在后端结构中，一切可能抛出异常的地方都必须被检测和标记。同时，我们在代码的 `handler` 层提供了异常处理器用于向前端回传异常

```
func GenerateResponse(c *gin.Context, data interface{}, err error) {
    if err != nil {
        c.JSON(http.StatusBadRequest, gin.H{
            "status":  -1,
            "message": err.Error(),
        })
    } else {
        c.JSON(http.StatusOK, gin.H{
            "status":  0,
            "message": "success",
            "data":    data,
        })
    }
}
```

例如：当我调用这个API：`http://127.0.0.1:8080/api/v1/get-user-info?id=1`，我可以接收到正确响应

```
{"data":{"id":1,"name":"zhenghui","tel-number":"18055627612","email":"wangzhenghui@bupt.edu.cn","company":"Bytedance.Inc","university":"BUPT"},"message":"success","status":0}
```

但如果你调用这个API：`http://127.0.0.1:8080/api/v1/get-user-info?id=2`，你接收的返回应该是下面的文本，因为我的数据表中不存在 `id=2` 的行（请注意：如果你运行在自己的环境下，你的返回和我的不会一样，因为你数据库的配置跟我不同）

```
{"message":"record not found","status":-1}
```

同时，为了迅速定位产生异常的位置，我们需要在所有抛出异常的地方打印日志

```
user, err := db_dal.SearchUserInfo(userId)
if err != nil {
   log.Println("db_dal.SearchUserInfo failed to execute, error: ", err)
   return response, err
}
```

当此处抛出异常时，我们会在日志中看到如下的记录，这让我们拥有迅速定位问题的能力：

```
2021/06/06 18:32:51 /Users/bytedance/go/src/bootstrap/biz/dal/db_dal/user_info_dal.go:9
record not found
[13.064ms] [rows:0] SELECT * FROM `user_info` WHERE id = 2 ORDER BY `user_info`.`id` LIMIT
1
2021/06/06 18:32:51 db_dal.SearchUserInfo failed to execute, error:  record not found
```