极客时间 训练营

# 第二周 用户基本功能与 GORM 入门

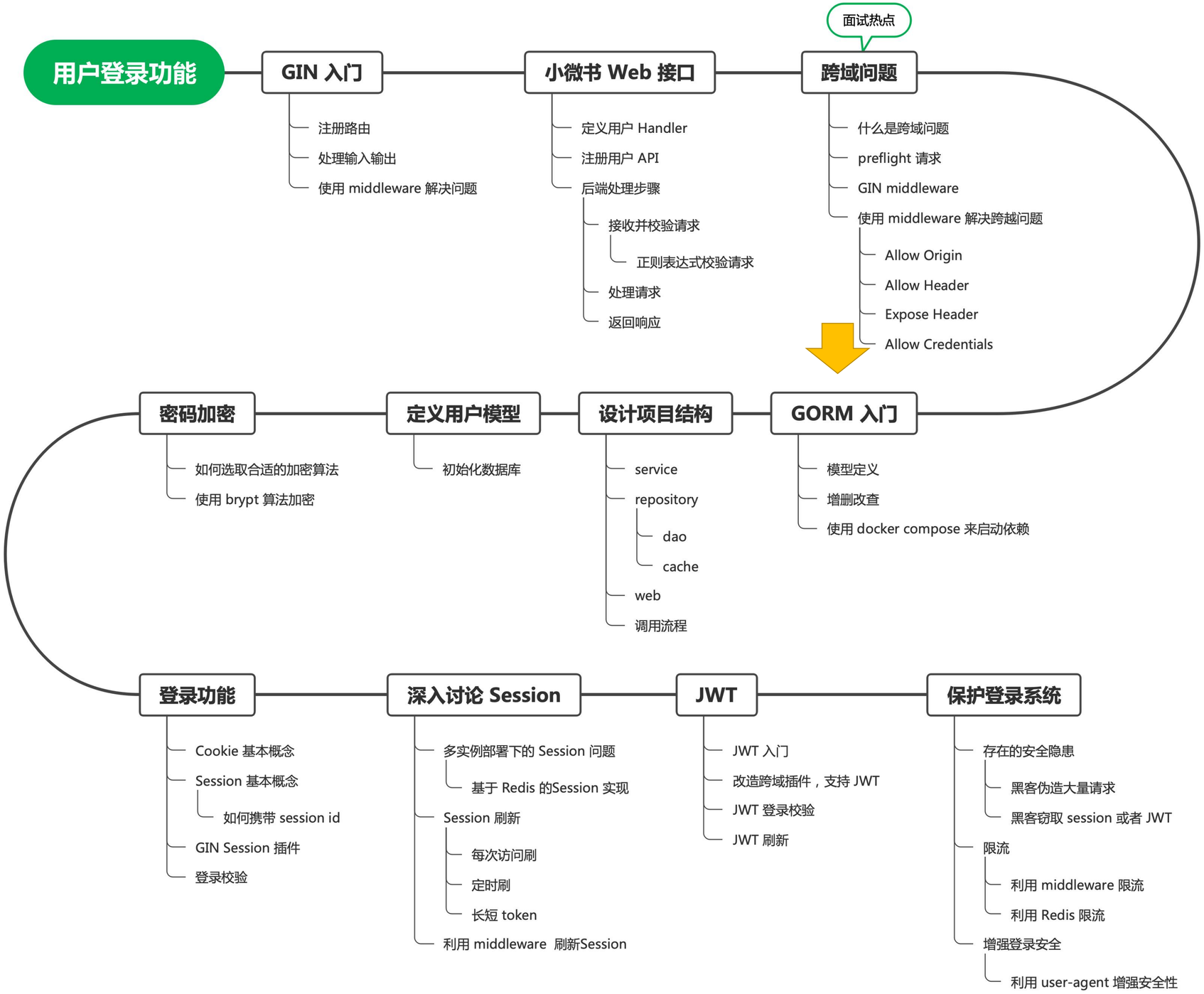
大明

# 目录

- GORM 入门
- 密码加密
- 登录与登录校验

极客时间 训练营

# GORM 入门


用户登录功能
GIN 入门
注册路由
处理输入输出
使用 middleware 解决问题
小微书 Web 接口
定义用户 Handler
注册用户 API
后端处理步骤
接收并校验请求
正则表达式校验请求
处理请求
返回响应
面试热点
跨域问题
什么是跨域问题
preflight 请求
GIN middleware
使用 middleware 解决跨越问题
Allow Origin
Allow Header
Expose Header
Allow Credentials
GORM 入门
模型定义
增删改查
使用 docker compose 来启动依赖
设计项目结构
service
repository
dao
cache
web
调用流程
定义用户模型
初始化数据库
密码加密
如何选取合适的加密算法
使用 brypt 算法加密
登录功能
Cookie 基本概念
Session 基本概念
如何携带 session id
GIN Session 插件
登录校验
深入讨论 Session
多实例部署下的 Session 问题
基于 Redis 的Session 实现
Session 刷新
每次访问刷
定时刷
长短 token
利用 middleware 刷新Session
JWT
JWT 入门
改造跨域插件，支持 JWT
JWT 登录校验
JWT 刷新
保护登录系统
存在的安全隐患
黑客伪造大量请求
黑客窃取 session 或者 JWT
限流
利用 middleware 限流
利用 Redis 限流
增强登录安全
利用 user-agent 增强安全性

# GORM 简介

- GORM 是什么？GORM 是一个 Go 语言的 ORM 框架，性能优秀，简单易用。
- GORM 功能丰富：
  - 支持多种数据库，包括 MySQL、PostgreSQL。
  - 支持简单查询，支持事务，也支持关联关系。
  - 支持钩子。
  - 支持自动迁移工具。

遇事不决选 GORM，也可以期待我将来发布的泛型 ORM 框架。

# GORM 入门：增删改查

安装 GORM 依赖：

- 安装本体：go get -u gorm.io/gorm
- 安装对应数据库的驱动，注意 GORM 做了二次封装：go get -u gorm.io/driver/mysql

基本使用步骤：

- 初始化 DB 实例
- （可选）初始化表结构
- 发起查询

```
unc main() {  new *
    db, err := gorm.Open(mysql.Open( dsn: "root:root@tcp(localhost:13306)/xiaohongshu"))
    if err != nil {
        panic(err)
    }
    // 迁移 schema，这里其实就是会同步你的表结构
    db.AutoMigrate(&Product{})

    // Create
    db.Create(&Product{Code: "D42", Price: 100})

    // Read
    var product Product
    db.First(&product,  conds...: 1)                 // 根据整型主键查找
    db.First(&product,  conds...: "code = ?", "D42") // 查找 code 字段值为 D42 的记录

    // Update - 将 product 的 price 更新为 200
    db.Model(&product).Update( column: "Price",  value: 200)
    // Update - 更新多个字段
    db.Model(&product).Updates(Product{Price: 200, Code: "F42"}) // 仅更新非零值字段
    db.Model(&product).Updates(map[string]interface{}{"Price": 200, "Code": "F42"})

    // Delete - 删除 product
    db.Delete(&product,  conds...: 1)
```

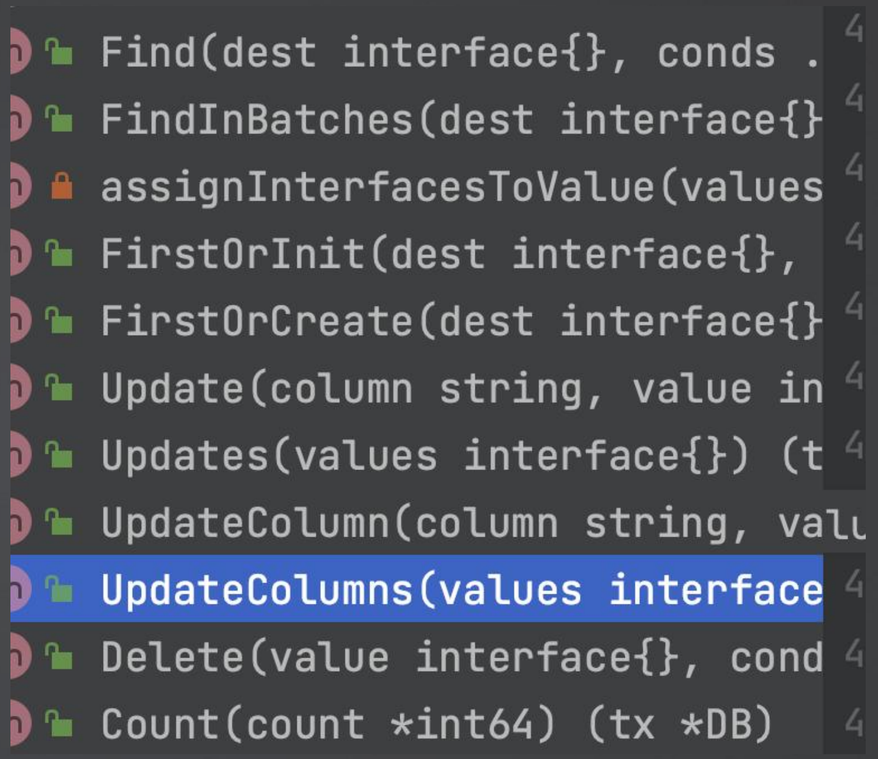
# GORM 学习难点

GORM 的很多接口都是接收 interface 的，根据你传入不同类型的参数，会执行不同的行为。

典型的就是 Updates 方法。

所以，记得看文档、看源码。要是拿捏不准，就写一个例子来测试。

在课程后面的内容中，我们差不多都会接触到。




虽然 GORM 我用过很多次，但是我还是觉得这些方法语义过于模糊，经常写错。

# Product 定义

这个是直接源自 GORM 的例子。

Product 组合了一个 gorm.Model。

gorm.Model 里面已经提前定义好了四个公共字段。

其中，DeleteAt 代表这是一个希望软删除的模型。

在实践中，每个公司用的公共字段可能都不同。

```go
type Product struct {  4 usages
    gorm.Model
    Code  string
    Price uint
}
```

```go
type Model struct {
    ID        uint `gorm:"primarykey"`
    CreatedAt time.Time
    UpdatedAt time.Time
    DeletedAt DeletedAt `gorm:"index"`
}
```

# 模型定义

模型定义，我建议大家不要死记硬背，每次要用的时候就打开官方文档来看。

https://gorm.io/zh_CN/docs/models.html

比较常用的可以记一下。这个东西连面试也不会问，所以记不记都无所谓。

面试都不会问你这些标签的。

| | |
|---|---|
| | AUTO_INCREMENT |
| serializer | 指定将数据序列化或反序列化到数据库中的序列化器, 例如: `serializer:json/gob/unixtime` |
| size | 定义列数据类型的大小或长度，例如 `size: 256` |
| primaryKey | 将列定义为主键 |
| unique | 将列定义为唯一键 |
| default | 定义列的默认值 |
| precision | 指定列的精度 |
| scale | 指定列大小 |
| not null | 指定列为 NOT NULL |
| autoIncrement | 指定列为自动增长 |
| autoIncrementIncrement | 自动步长，控制连续记录之间的间隔 |
| embedded | 嵌套字段 |
| embeddedPrefix | 嵌入字段的列名前缀 |
| autoCreateTime | 创建时追踪当前时间，对于 `int` 字段，它会追踪时间戳秒数，您可以使用 `nano` / `milli` 来追踪纳秒、毫秒时间戳，例如: `autoCreateTime:nano` |

# 用户注册：存储用户基本信息

前面，我们已经把前端内容都搞好了，现在我们要把接收到的数据存储到数据库中。

为此我们需要准备一个数据库。

我们使用 docker-compose 来搭建开发环境所需的依赖。

# Docker Compose 基本语法

docker compose 的基本语法很简单。

- services 是顶级节点，也就是你要启动的服务全部放在这里。MySQL 就是我们预期中的一个服务。
- mysql8：指的是我们这个服务叫 mysql8。
- image：我们这个服务里运行的是什么镜像，或者说跑的是什么。这里指定了使用 mysql:8.0.29 这个版本。
- command：启动命令，这里相当于加上了这个命令行参数。
- volumes：挂载文件。这里我挂载了一个文件用来初始化数据库。
- ports：指定端口映射关系。

```yaml
services:
  mysql8:
    image: mysql:8.0.29
    restart: always
    command: --default-authentication-plugin=mysql_native_password
    environment:
      MYSQL_ROOT_PASSWORD: root
    volumes:
      #       设置初始化脚本
      - ./script/mysql/:/docker-entrypoint-initdb.d/
    ports:
      #       注意这里我映射为了 13316 端口
      - "13316:3306"
```

docker compose 本身还有很多选项，你可以去看看文档。

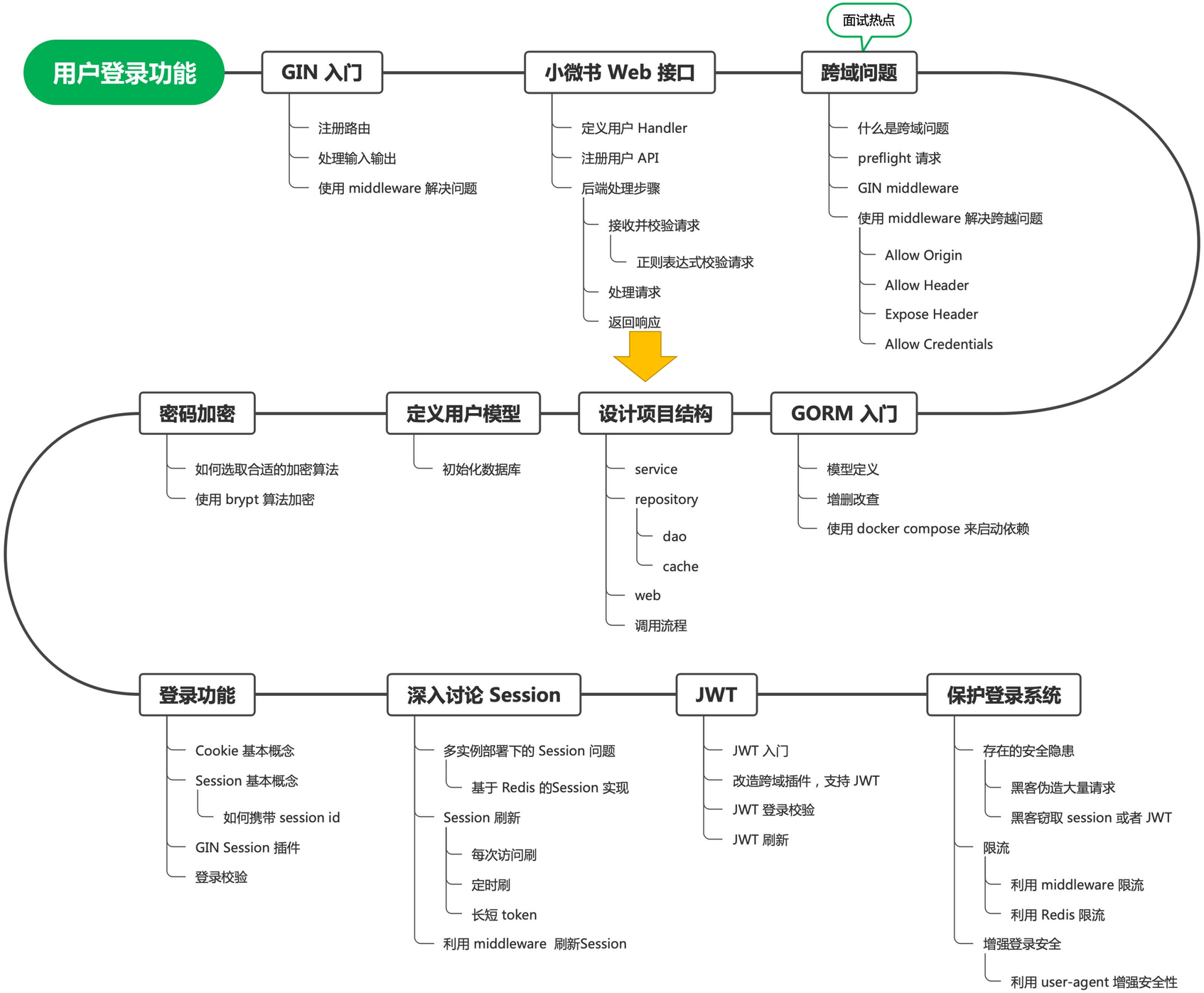
# Docker Compose 基本命令

docker compose 有关的命令，你只需要掌握两个：

- docker compose up：初始化 docker-compose 并启动。
- docker compose down：删除 docker compose 里面创建的各种容器。

```
webook-mysql8-1  | 2023-07-15T09:16:09.629286Z 0 [System] [MY-013602] [Server] Channel mysql_m
ain configured to support TLS. Encrypted connections are now supported for this channel.
webook-mysql8-1  | 2023-07-15T09:16:09.629974Z 0 [Warning] [MY-011810] [Server] Insecure confi
guration for --pid-file: Location '/var/run/mysqld' in the path is accessible to all OS users.
 Consider choosing a different directory.
webook-mysql8-1  | 2023-07-15T09:16:09.636212Z 0 [System] [MY-011323] [Server] X Plugin ready
for connections. Bind-address: '::' port: 33060, socket: /var/run/mysqld/mysqlx.sock
webook-mysql8-1  | 2023-07-15T09:16:09.636227Z 0 [System] [MY-010931] [Server] /usr/sbin/mysql
d: ready for connections. Version: '8.0.29'  socket: '/var/run/mysqld/mysqld.sock'  port: 3306
  MySQL Community Server - GPL.
```

看到这里的 ready for connections 就说明启动成功了。如果机器性能比较差的话，可能要等几分钟。


用户登录功能
GIN 入门
注册路由
处理输入输出
使用 middleware 解决问题
小微书 Web 接口
定义用户 Handler
注册用户 API
后端处理步骤
接收并校验请求
正则表达式校验请求
处理请求
返回响应
面试热点
跨域问题
什么是跨域问题
preflight 请求
GIN middleware
使用 middleware 解决跨越问题
Allow Origin
Allow Header
Expose Header
Allow Credentials
GORM 入门
模型定义
增删改查
使用 docker compose 来启动依赖
设计项目结构
service
repository
dao
cache
web
调用流程
定义用户模型
初始化数据库
密码加密
如何选取合适的加密算法
使用 brypt 算法加密
登录功能
Cookie 基本概念
Session 基本概念
如何携带 session id
GIN Session 插件
登录校验
深入讨论 Session
多实例部署下的 Session 问题
基于 Redis 的Session 实现
Session 刷新
每次访问刷
定时刷
长短 token
利用 middleware 刷新Session
JWT
JWT 入门
改造跨域插件，支持 JWT
JWT 登录校验
JWT 刷新
保护登录系统
存在的安全隐患
黑客伪造大量请求
黑客窃取 session 或者 JWT
限流
利用 middleware 限流
利用 Redis 限流
增强登录安全
利用 user-agent 增强安全性

# 数据库相关代码放哪里？

数据库准备好了，现在就要考虑，数据库相关的增删改查代码放在哪里比较好？

能不能直接在 UserHandler 里面操作数据库？

```
    if !isPassword {
        ctx.String(http.StatusOK,
            format: "密码必须包含数字、特殊字符，并且长度不能小于 8 位")
        return
    }
    // 能不能考虑在这里直接使用 GORM?

    ctx.String(http.StatusOK, format: "hello, 你在注册")
}
```

不能。因为 Handler 只是负责和 HTTP 有关的东西。我们需要一个代表数据库抽象的东西。

# 引入 Service - Repostiory - DAO 三层结构

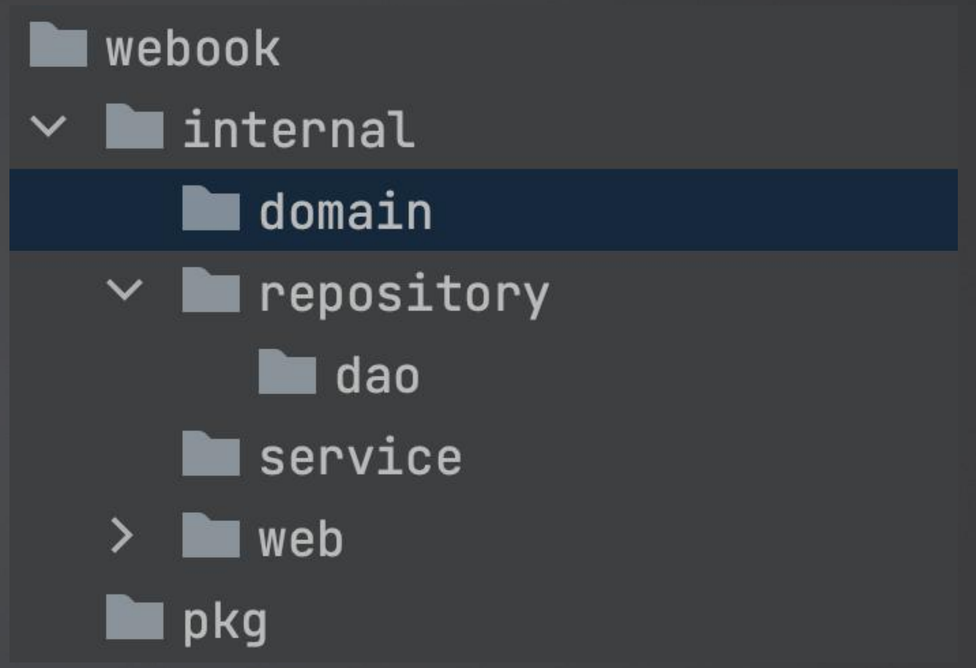
这里我们直接引入 Service - Repository - DAO 三层结构。其中 service、repostory 参考的是 DDD 设计。

service: 代表的是领域服务（domain service），代表一个业务的完整的处理过程。

repository：按照 DDD 的说法，是代表领域对象的存储，这里你直观理解为存储数据的抽象。

dao: 代表的是数据库操作。

同时，我们还需要一个 domain，代表领域对象。



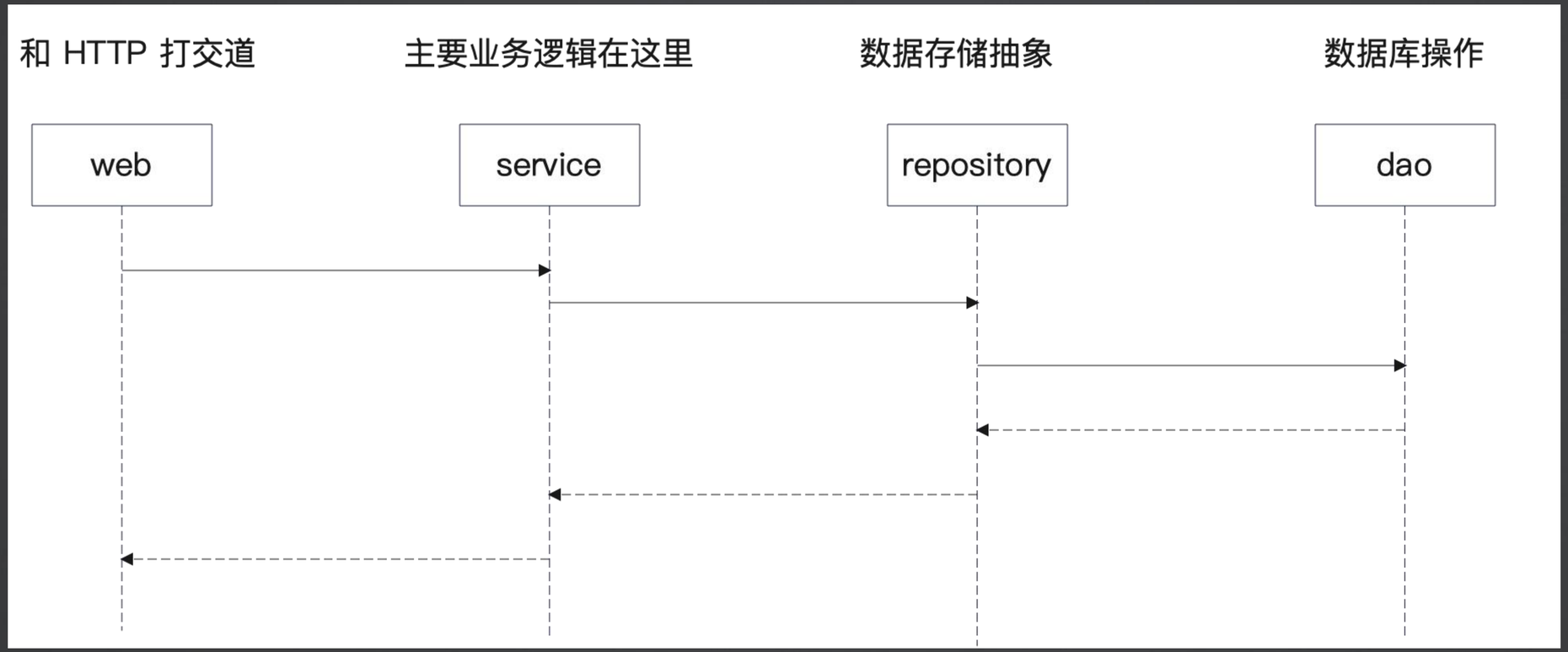

这时候你先不要去探究什么是 DDD，DDD 是那种看起来很简单，但实际上很难理解的东西。

# 如何理解这些东西？

- 为什么有 repository 之后，还要有 dao？repository 是一个整体抽象，它里面既可以考虑用 ElasticSearch，也可以考虑使用 MySQL，还可以考虑用 MongoDB。所以它只代表数据存储，但是不代表数据库。
- service 是拿来干嘛的？简单来说，就是组合各种 repository、domain，偶尔也会组合别的 service，来共同完成一个业务功能。
- domain 又是什么？它被认为是业务在系统中的直接反应，或者你直接理解为一个业务对象，又或者就是一个现实对象在程序中的反应。

# 调用流程

总结起来，预期中的调用流程如下图。



# 改造代码

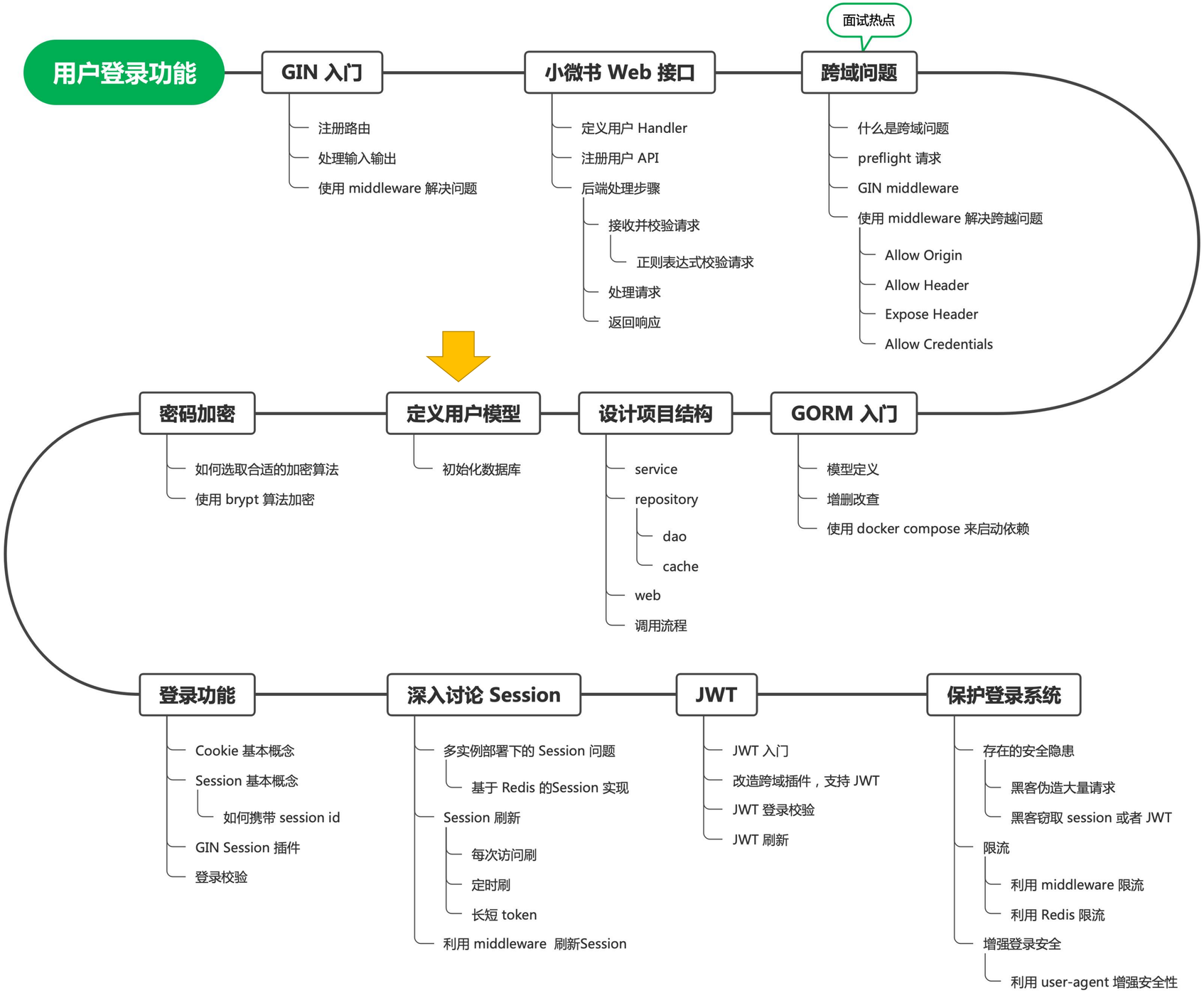
先创建一个 UserService，但是它基本上不做什么事情。

repository 也没做啥，最关键的都在 dao 里面。

```go
type UserService struct {  2 usages   new *
    repo *repository.UserRepository
}

func (svc *UserService) Signup(ctx context.Co
    return svc.repo.Create(ctx, u)
}
```

```go
type UserRepository struct {  2 usages   n
    dao *dao.UserDAO
}

func (ur *UserRepository) Create(ctx cont
    return ur.dao.Insert(ctx, dao.User{
        Email:    u.Email,
        Password: u.Password,
    })
}
```

```go
type UserDAO struct {  2 usages   new *
    db *gorm.DB
}

func (ud *UserDAO) Insert(ctx context.Context, u User) error
    return ud.db.WithContext(ctx).Create(&u).Error
}
```


用户登录功能
GIN 入门
注册路由
处理输入输出
使用 middleware 解决问题
小微书 Web 接口
定义用户 Handler
注册用户 API
后端处理步骤
接收并校验请求
正则表达式校验请求
处理请求
返回响应
面试热点
跨域问题
什么是跨域问题
preflight 请求
GIN middleware
使用 middleware 解决跨越问题
Allow Origin
Allow Header
Expose Header
Allow Credentials
GORM 入门
模型定义
增删改查
使用 docker compose 来启动依赖
设计项目结构
service
repository
dao
cache
web
调用流程
定义用户模型
初始化数据库
密码加密
如何选取合适的加密算法
使用 brypt 算法加密
保护登录系统
存在的安全隐患
黑客伪造大量请求
黑客窃取 session 或者 JWT
限流
利用 middleware 限流
利用 Redis 限流
增强登录安全
利用 user-agent 增强安全性
JWT
JWT 入门
改造跨域插件，支持 JWT
JWT 登录校验
JWT 刷新
深入讨论 Session
多实例部署下的 Session 问题
基于 Redis 的Session 实现
Session 刷新
每次访问刷
定时刷
长短 token
利用 middleware 刷新Session
登录功能
Cookie 基本概念
Session 基本概念
如何携带 session id
GIN Session 插件
登录校验

极客时间 训练营

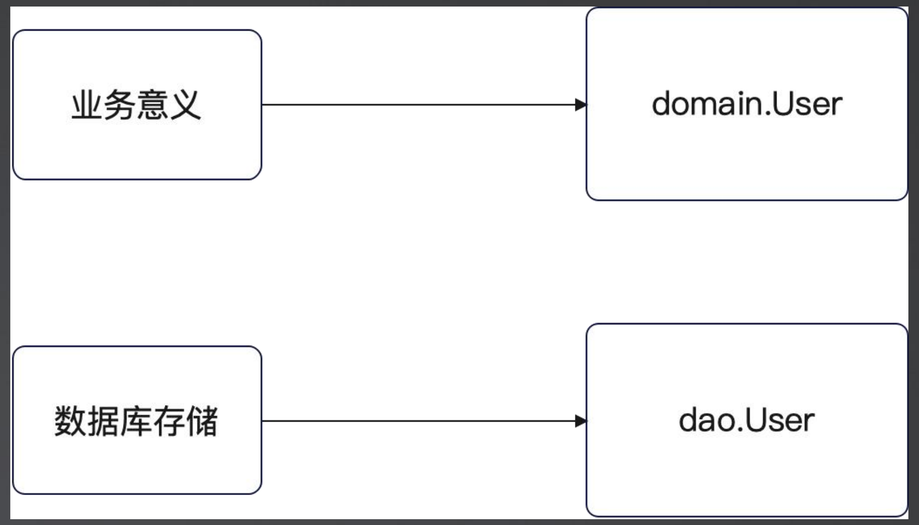
# dao 中的 User 模型

你应该注意到，我们 dao 里面操作的并不是 domain.User，而是定义了一个新的类型。

这是因为：domain.User 是业务概念，它不一定和数据库中表或者列完全对应得上。而 dao.User 则是直接映射到表里面的。

比如说，有些字段在数据库是 JSON 格式存储的，那么在 domain 里面就会被转为结构体。

```
type User struct {  3 usages   new *
    Email    string
    Password string
    Ctime    time.Time
}
```




```
type User struct {  2 usages   new *
    Id int64 `gorm:"primaryKey,autoIncrement"`
    // 设置为唯一索引
    Email    string `gorm:"unique"`
    Password string

    // 创建时间
    Ctime int64
    // 更新时间
    Utime int64
}
```

# User 模型详解

暂时在我们的系统里面，只需要这么一点字段。

后面当你需要其它信息的时候，可以考虑加上诸如生日、学历之类的东西。

这里比较关键的两个点：

- 我们使用了自增主键，也就是数据库会帮我们生成主键。
- Email 上被定义成唯一索引，也就是每个用户的邮箱不能冲突。

```
type User struct {  2 usages   new *
    Id int64 `gorm:"primaryKey,autoIncrement"`
    // 设置为唯一索引
    Email    string `gorm:"unique"`
    Password string

    // 创建时间
    Ctime int64
    // 更新时间
    Utime int64
}
```

# 怎么建表呢？

一般在规范比较严格的公司里面，表结构变更都是要走审批流程的。相当于你提供 SQL 给你 leader 看，然后给 DBA 看，最后再由 DBA 在目标库上执行。

评审最重要的就是看你的索引对不对。

不过在中小企业，可以考虑使用 GORM 自带的建表功能，也就是前面你看到的。

```go
import "gorm.io/gorm"

func InitTables(db *gorm.DB) error {
    return db.AutoMigrate(&User{})
}
```

后续有新的数据要存储，都要来这里初始化一下表。

# 怎么建表呢？

后续有新的数据要存储，都要来这里初始化一下表。

```go
type User struct {  2 usages   new *
    Id int64 `gorm:"primaryKey,autoIncrement"`
    // 设置为唯一索引
    Email    string `gorm:"unique"`
    Password string

    // 创建时间
    Ctime int64
    // 更新时间
    Utime int64
}
```

```sql
create table users
(
    id       bigint auto_increment
        primary key,
    email    varchar(191) null,
    password longtext     null,
    ctime    bigint       null,
    utime    bigint       null,
    constraint email
        unique (email)
)
    collate = utf8mb4_0900_ai_ci;
```

# 初始化结构体

因为从 UserHandler 到 UserDAO 都修改了，里面都有一些字段，所以我们需要考虑初始化这些东西。

这里，我直接按照将来准备使用的依赖注入来准备初始化过程。

现在为每一个类都加上一个对应的初始化方法 NewXXX。

```go
func NewUserDAO(db *gorm.DB) *UserDAO {
    return &UserDAO{
        db: db,
    }
}
```

```go
func NewUserHandler(svc *service.UserService) *
    return &UserHandler{
        svc:              svc,
        emailRegexExp:    regexp.MustCompile(em
        passwordRegexExp: regexp.MustCompile(pa
    }
}
```

```go
func NewUserService(repo *repository.UserRepository)
    return &UserService{
        repo: repo,
    }
}
```

```go
func NewUserRepository(d *dao.UserDAO) *UserRepository
    return &UserRepository{
        dao: d,
    }
}
```

# main 函数

在 main 函数里，组装好全部东西，而后抽取到不同的方法里面。

```go
func main() {  Deng Ming *
    db := initDB()
    server := initWebServer()
    initUser(server, db)
    server.Run( addr...: ":8080")
}
```

```go
func initWebServer() *gin.Engine {  1
    server := gin.Default()
    server.Use(cors.New(cors.Config{
        AllowCredentials: true,
        AllowHeaders:     []string{"C
        AllowOriginFunc: func(origin
            if strings.HasPrefix(orig
            return strings.Contains(o
        },
        MaxAge: 12 * time.Hour,
    }))
    return server
}
```

```go
func initUser(server *gin.Engine, db *gorm.DB)
    ud := dao.NewUserDAO(db)
    ur := repository.NewUserRepository(ud)
    us := service.NewUserService(ur)
    c := web.NewUserHandler(us)
    c.RegisterRoutes(server)
}
```

```go
func initDB() *gorm.DB {  1 usage   n
    db, err := gorm.Open(mysql.Open(
    if err != nil {
        panic(err)
    }
    err = dao.InitTables(db)
    if err != nil {
        panic(err)
    }
    return db
}
```

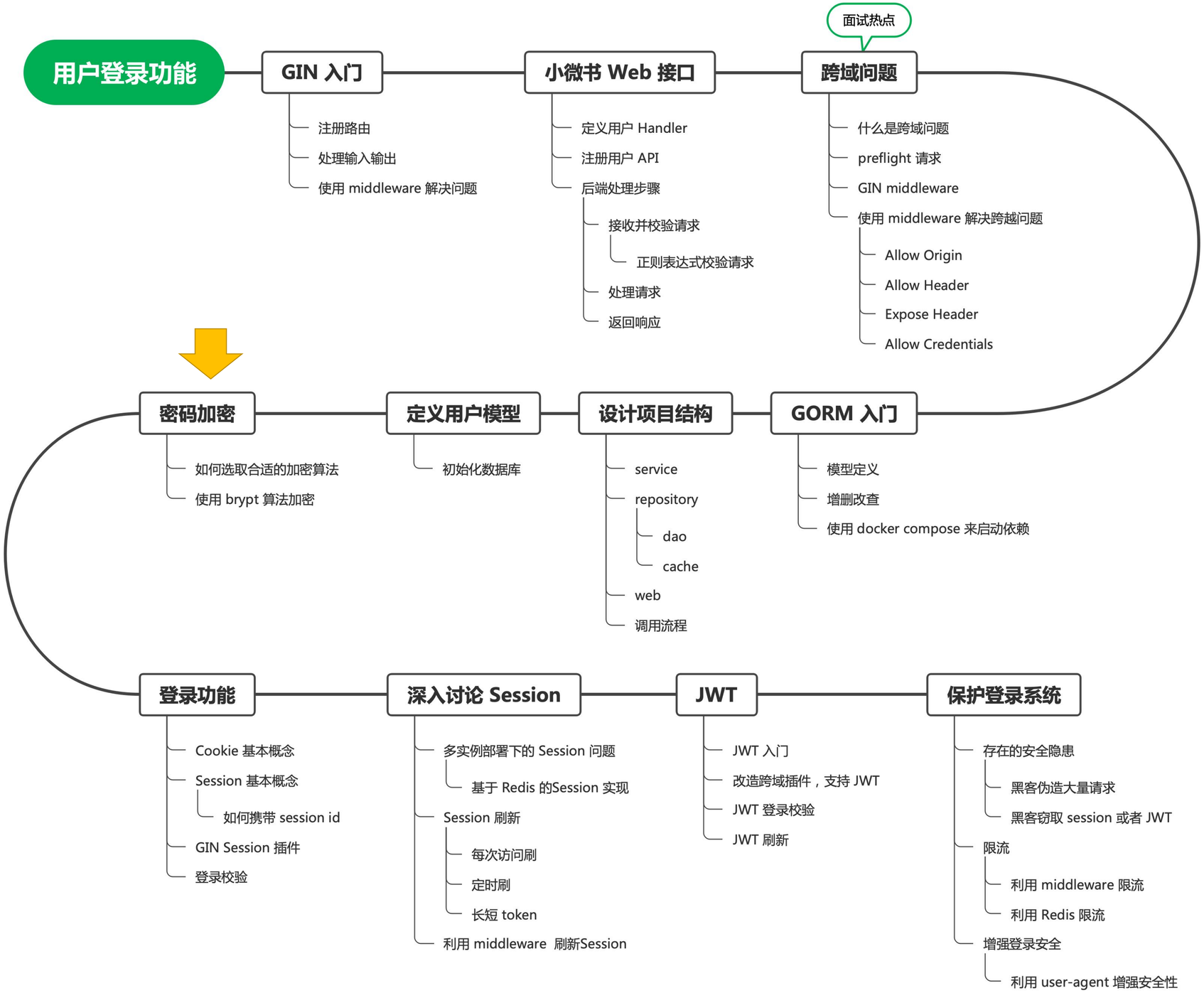
# 测试效果

现在启动 main 函数，然后我们界面操作试试，看看能不能成功。我注册了两个不同的数据。


6 rows　Tx: Auto　DDL

| | id | email | password | ctim |
|---|---|---|---|---|
| 1 | 1 | 123@qq.com | $2a$10$e5kyvQPQ6a3dA3LqxRTy… | |
| 2 | 2 | 1234@qq.com | hello#world123 | |

# 密码加密


用户登录功能
GIN 入门
注册路由
处理输入输出
使用 middleware 解决问题
小微书 Web 接口
定义用户 Handler
注册用户 API
后端处理步骤
接收并校验请求
正则表达式校验请求
处理请求
返回响应
面试热点
跨域问题
什么是跨域问题
preflight 请求
GIN middleware
使用 middleware 解决跨越问题
Allow Origin
Allow Header
Expose Header
Allow Credentials
GORM 入门
模型定义
增删改查
使用 docker compose 来启动依赖
设计项目结构
service
repository
dao
cache
web
调用流程
定义用户模型
初始化数据库
密码加密
如何选取合适的加密算法
使用 brypt 算法加密
保护登录系统
存在的安全隐患
黑客伪造大量请求
黑客窃取 session 或者 JWT
限流
利用 middleware 限流
利用 Redis 限流
增强登录安全
利用 user-agent 增强安全性
JWT
JWT 入门
改造跨域插件，支持 JWT
JWT 登录校验
JWT 刷新
深入讨论 Session
多实例部署下的 Session 问题
基于 Redis 的Session 实现
Session 刷新
每次访问刷
定时刷
长短 token
利用 middleware 刷新Session
登录功能
Cookie 基本概念
Session 基本概念
如何携带 session id
GIN Session 插件
登录校验

极客时间 训练营

# 密码怎么加密？

代码看上去没有问题，但是好像忘了一件事情：密码是敏感信息，需要加密存储。

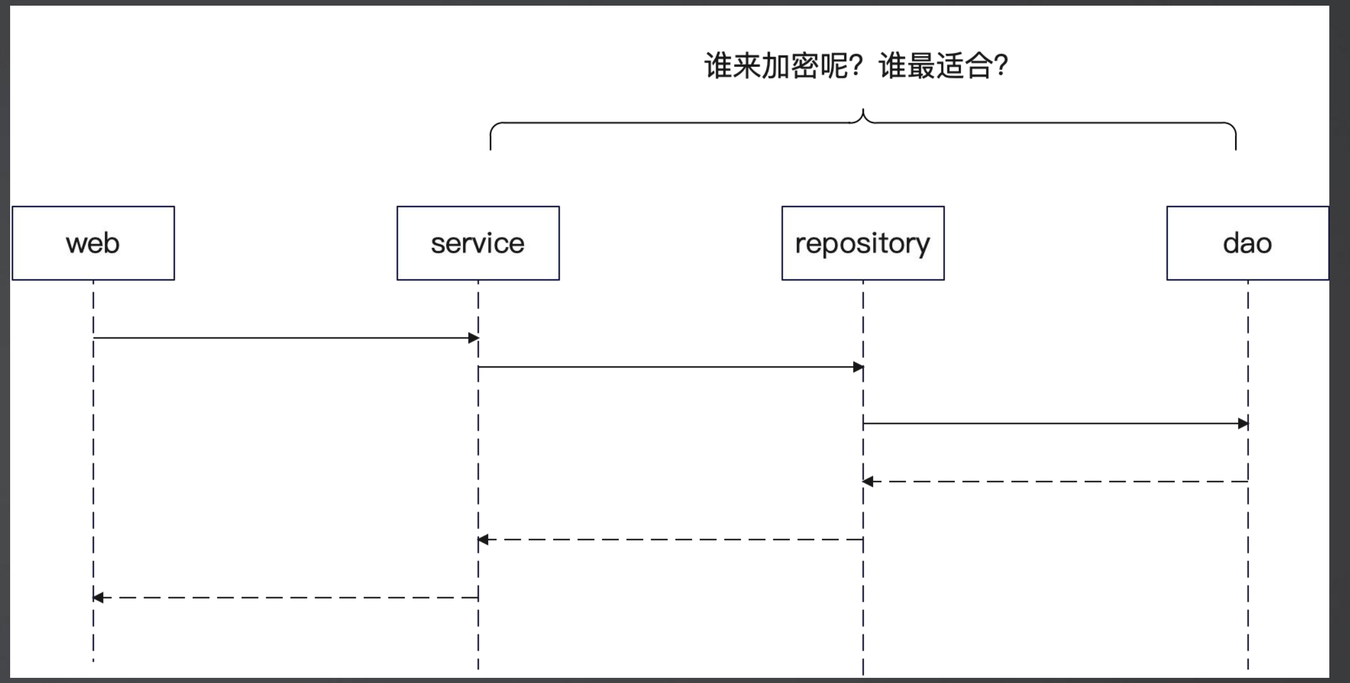
问题来了：

- 谁来加密？service 还是 repository 还是 dao？
- 怎么加密？怎么选择一个安全的加密算法？

敏感信息你要防两类人：

- 研发，包括你和你的同事。
- 攻击者。

PS：敏感信息应该是连日志都不能打。



# 加密的位置

实际上，你选择 service、repository、dao, 包括 domain 都有说得过去的理由。

- service 加密：加密是一个业务概念，不是一个存储概念。
- repository 加密：加密是一个存储概念，毕竟我们说的是“加密存储”。
- dao 加密：加密是一个数据库概念，因为我完全可以选择利用数据库本身的加密功能来实现。
- domain 加密：加密是一个业务概念，但是应该是“用户（User）”自己才知道怎么加密。

这种就是编程里面比较无聊的、没有正确答案的实践问题。这里我选择 service 加密，也就是认为加密是业务逻辑的一部分，但是它不是 domain 应该管的。

如果你选择不同的加密位置，那么它会影响到你别的接口的实现细节，比如说登录。

# 如何加密？

加密算法的选择会直接影响你整个系统的安全性，因为攻击者一旦拿到了密码，差不多就可以为所欲为了。

选择加密算法的标准就一个，难破解。你要考虑以下问题：

- 相同的密码，加密后的结果应该不同。你可以预期，很多用户习惯用 123456 这种密码，但是我们希望数据库存储的值还是不一样。
- 难以通过碰撞、彩虹表来破解。

常见的加密算法无非就是下面这些，安全性逐步提高：

1. md5 之类的哈希算法。
2. 在 1 的基础上，引入了盐值(salt），或者进行多次哈希等。
3. PBKDF2、BCrypt 这一类随机盐值的加密算法，同样的文本加密后的结果都不同。

# 使用 BCrypt 加密

bcrypt 是一个号称最安全的加密算法。

优点：

- 不需要你自己去生成盐值。
- 不需要额外存储盐值。
- 可以通过控制 cost 来控制加密性能。
- 同样的文本，加密后的结果不同。

要是用它，需要使用 golang.org/x/crypto。

因为 bcrypt 限制密码长度不能超过 72 字节，所以在校验时要校验这个长度。只需要修改一下正则表达式就可以。

```go
func TestPasswordEncrypt(t *testing.T) {  new *
    pwd := []byte("123456#123456")
    // 加密
    encrypted, err := bcrypt.GenerateFromPassword(pwd, bcrypt.DefaultCost)
    // 比较
    err = bcrypt.CompareHashAndPassword(encrypted, pwd)
    require.NoError(t, err)
}
```

bcrypt 加密之后是没办法解密的，所以只能同时比较加密之后的值来确定两者是否相等。

# 效果

```go
func (svc *UserService) Signup(ctx context.Context, u
    hash, err := bcrypt.GenerateFromPassword([]byte(u
    if err != nil {
        return err
    }
    u.Password = string(hash)
    return svc.repo.Create(ctx, u)
}
```

| | | |
|---|---|---|
| 6 | 12347@qq.com | $2a$10$bdT2iHbkfvCtUlIhaYHi… |
| 7 | 12348@qq.com | $2a$10$b8WItpl6kbFJt/KRwIV2… |

# 怎么获得邮件冲突的错误？

我们尽可能给前端返回了准确的错误信息，但是有一个信息没有返回，即怎么知道用户的邮件冲突了呢？

答案就是，我们需要拿到数据库的唯一索引冲突错误。

在这里，我们需要用 MySQL GO 驱动的 error 定义，找到准确的错误。

在 dao 这一层，我们转为了 ErrUserDuplicateEmail 错误，并且将这个错误一路往上返回。

```go
func (ud *UserDAO) Insert(ctx context.Context, u User)
    now := time.Now().UnixMilli()
    u.Ctime = now
    u.Utime = now
    err := ud.db.WithContext(ctx).Create(&u).Error
    if me, ok := err.(*mysql.MySQLError); ok {
        const uniqueIndexErrNo uint16 = 1062
        if me.Number == uniqueIndexErrNo {
            return ErrUserDuplicateEmail
        }
    }
    return err
}
```

# 传导错误与检测

在 repository 和 service 层，我们都使用别名机制，继续向上返回错误。

在最顶层的 Handler 里面，我们进行检测。

在确定是 ErrUserDuplicateEmail 的情况下，提示邮箱冲突了。

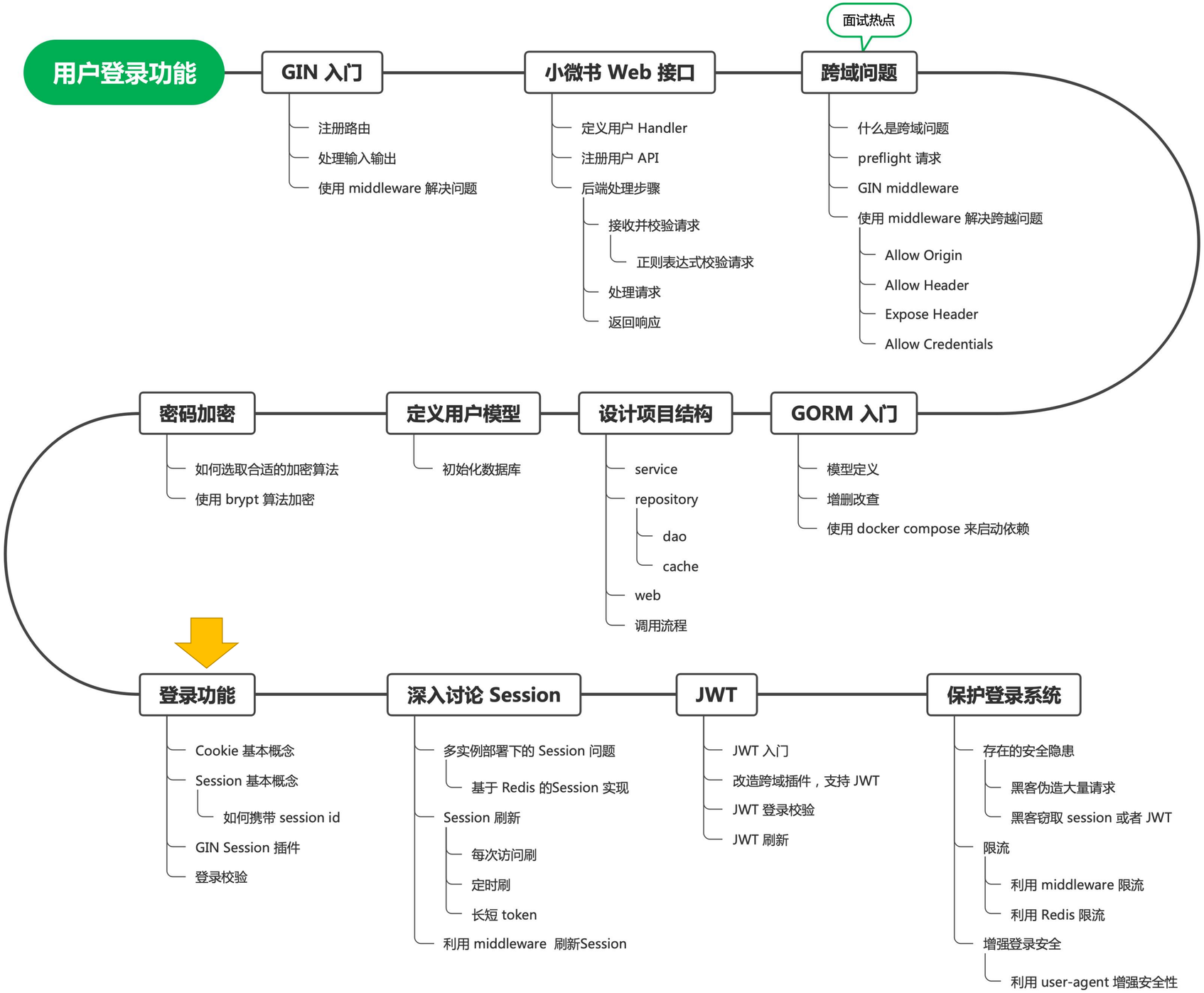
使用别名的机制，层层传导，让我们在 Handler 里面依旧保持只依赖 service，避免了跨层依赖的问题。

```
var ErrUserDuplicateEmail = dao.ErrUserDuplicateEmail
```

```
var ErrUserDuplicateEmail = repository.ErrUserDuplicateEmail
```

```
err = c.svc.Signup(ctx.Request.Context(),
    domain.User{Email: req.Email, Password: req.ConfirmPassword})

if err == service.ErrUserDuplicateEmail {
    ctx.String(http.StatusOK, format: "重复邮箱，请换一个邮箱")
    return
}
if err != nil {
    ctx.String(http.StatusOK, format: "服务器异常，注册失败")
    return
}
```

# 登录功能


用户登录功能
GIN 入门
注册路由
处理输入输出
使用 middleware 解决问题
小微书 Web 接口
定义用户 Handler
注册用户 API
后端处理步骤
接收并校验请求
正则表达式校验请求
处理请求
返回响应
面试热点
跨域问题
什么是跨域问题
preflight 请求
GIN middleware
使用 middleware 解决跨越问题
Allow Origin
Allow Header
Expose Header
Allow Credentials
GORM 入门
模型定义
增删改查
使用 docker compose 来启动依赖
设计项目结构
service
repository
dao
cache
web
调用流程
定义用户模型
初始化数据库
密码加密
如何选取合适的加密算法
使用 brypt 算法加密
登录功能
Cookie 基本概念
Session 基本概念
如何携带 session id
GIN Session 插件
登录校验
深入讨论 Session
多实例部署下的 Session 问题
基于 Redis 的Session 实现
Session 刷新
每次访问刷
定时刷
长短 token
利用 middleware 刷新Session
JWT
JWT 入门
改造跨域插件，支持 JWT
JWT 登录校验
JWT 刷新
保护登录系统
存在的安全隐患
黑客伪造大量请求
黑客窃取 session 或者 JWT
限流
利用 middleware 限流
利用 Redis 限流
增强登录安全
利用 user-agent 增强安全性

# 登录功能

大多数网站的资源，都是要求你必须登录才能访问的。

比如，我们现在希望编辑和查看用户信息都必须登录之后才能访问。

所以登录本身分成两件事：

- 实现登录功能
- 登录态校验

我们先来看登录功能，登录请求被发到 /users/login 上。

```
// 分组注册
ug := server.Group( relativePath: "/users")
ug.POST( relativePath: "/signup", c.SignUp)
ug.POST( relativePath: "/login", c.Login)
ug.POST( relativePath: "/edit", c.Edit)
ug.GET( relativePath: "/profile", c.Profile)
```

必须登录

# 登录接口实现

在这里，你就能看出 service 和 repository 之间的分界线了。

service 会调用 repository 查找邮箱所对应的用户。

而后 service 会匹配输入的密码和数据库中保存的是否一致。

不管是用户没找到，还是密码错误，我们都返回同一个 error。

```
func (svc *UserService) Login(ctx context.Context,  1 usage
    email, password string) (domain.User, error) {
    u, err := svc.repo.FindByEmail(ctx, email)
    if err == repository.ErrUserNotFound {
        return domain.User{}, ErrInvalidUserOrPassword
    }
    err = bcrypt.CompareHashAndPassword([]byte(u.Password),
    if err != nil {
        return domain.User{}, ErrInvalidUserOrPassword
    }
    return u, err
}
```

# 登录校验

登录成功之后，我要去 /users/profile 的时候，我怎么知道用户登录没登录?

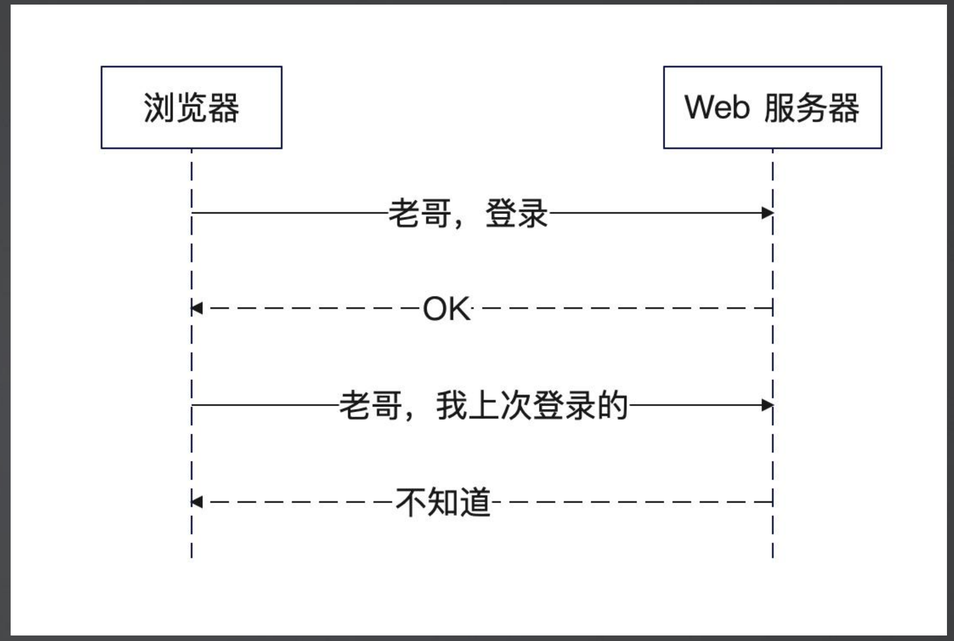
# 无状态的 HTTP 协议

什么叫做 HTTP 是无状态的？

是指，你连续发两次请求，HTTP 并不知道这两个都是你发的。也就是，它没办法将上一次请求和这一次请求关联起来。

所以我们需要有一种机制，记录一下这个状态。

于是就有两个东西：Cookie 和 Session。

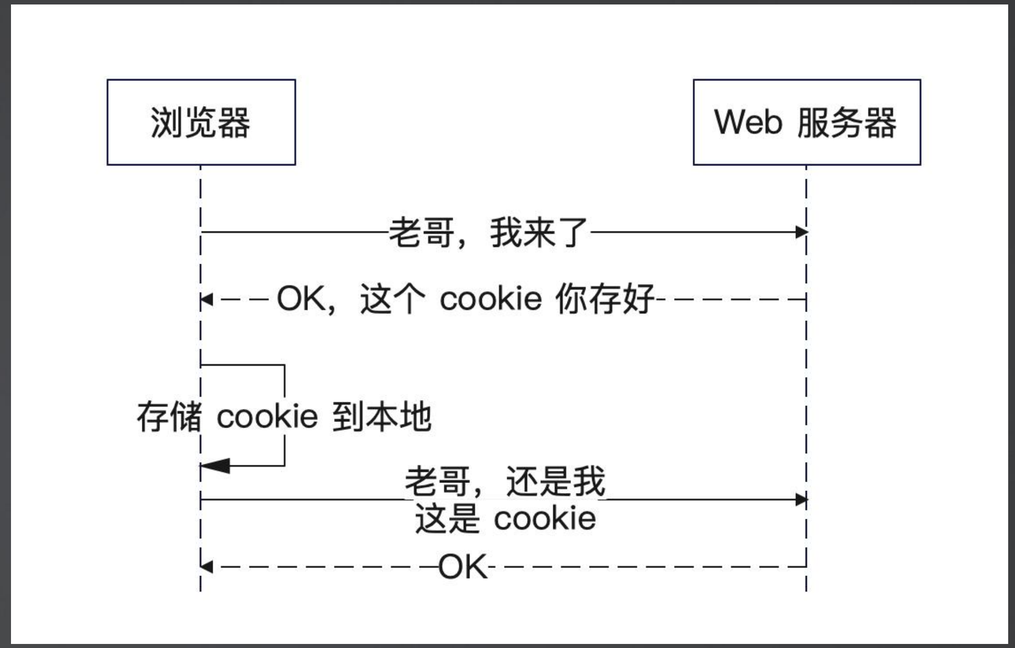


# Cookie

浏览器存储一些数据到本地，这些数据就是 Cookie。

简单理解，就是存储在你电脑上的键值对。

也正因为 Cookie 是放在浏览器本地的，所以很不安全。



# Cookie 关键配置

你在使用 Cookie 的时候，要注意“安全使用”。

Domain: 也就是 Cookie 可以用在什么域名下，按照最小化原则来设定。
Path：Cookie 可以用在什么路径下，同样按照最小化原则来设定。
Max-Age 和 Expires：过期时间，只保留必要时间。
Http-Only：设置为 true 的话，那么浏览器上的 JS 代码将无法使用这个 Cookie。永远设置为 true。
Secure：只能用于 HTTPS 协议，生产环境永远设置为 true。
SameSite：是否允许跨站发送 Cookie，尽量避免。

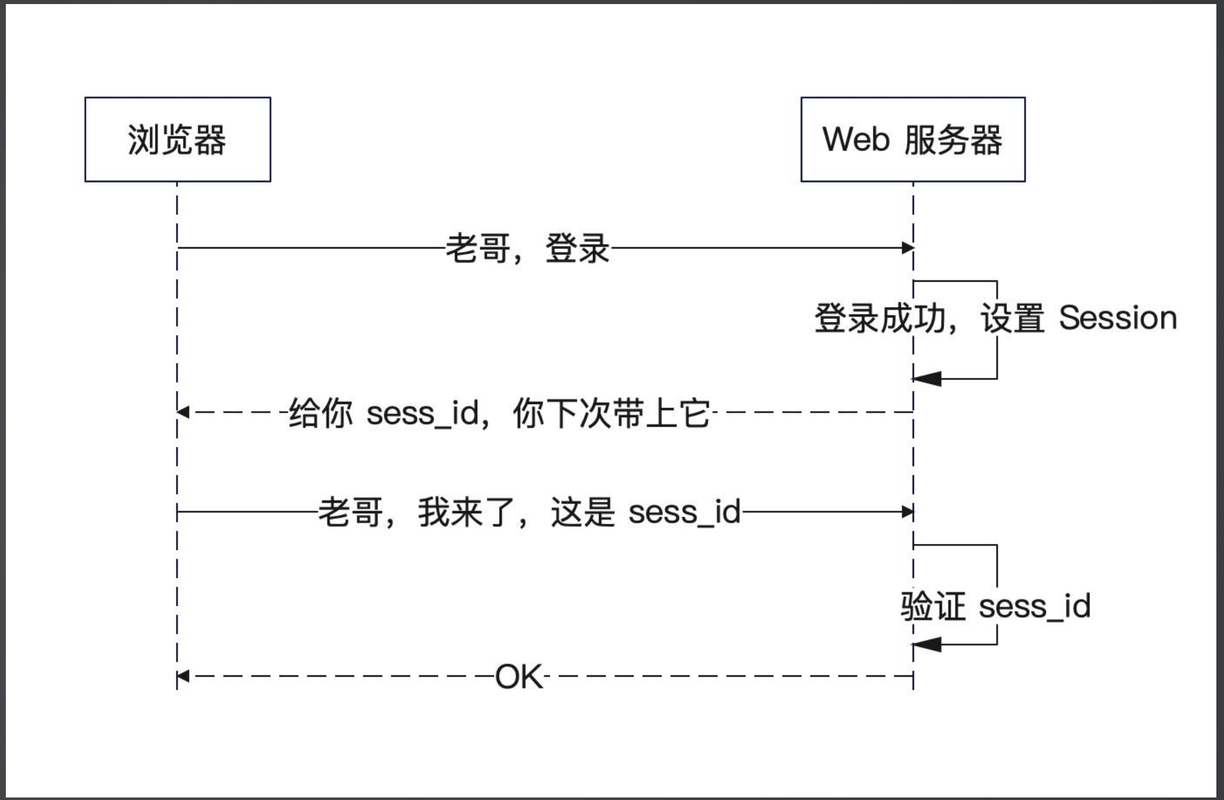
出去面试的时候要详细解释这些参数的含义。在面试初级工程师岗位的时候，会让你赢得微小的竞争优势。

# Session

因为 Cookie 本身不安全的特性，所以大部分时候，我们都只在 Cookie 里面放一些不太关键的数据。

关键数据我们希望放在后端，这个存储的东西就叫做 Session。

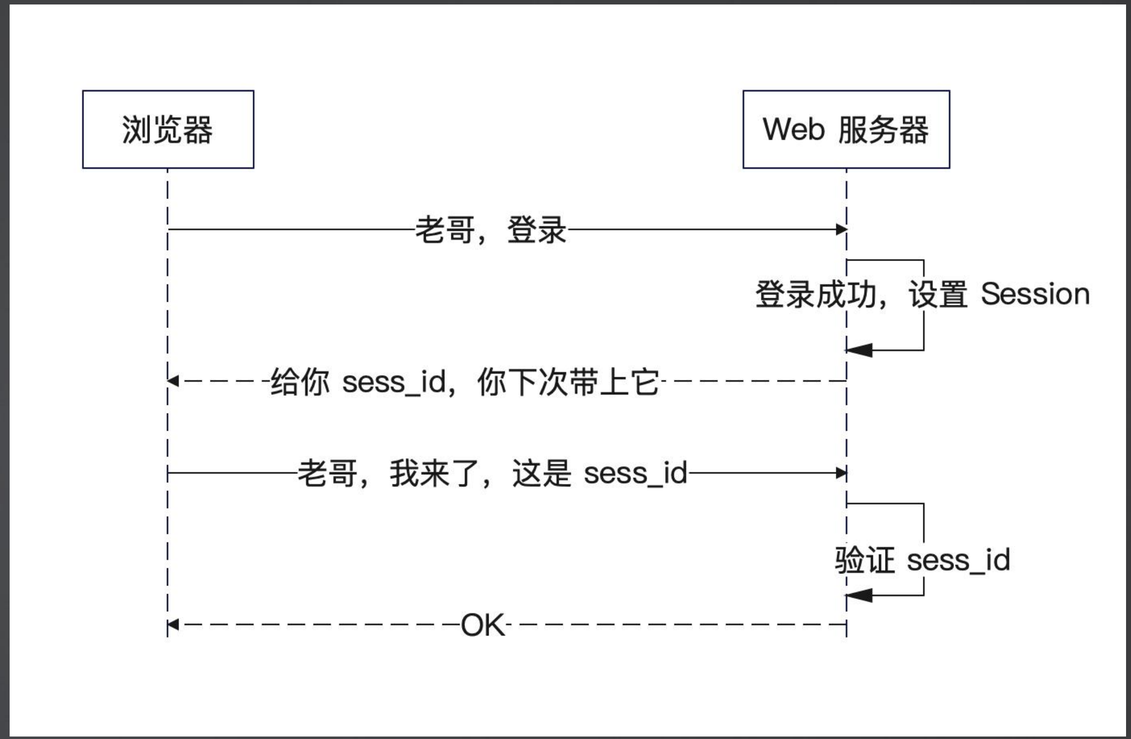
因此在登录里面，我们就可以通过 Session 来记录登录状态。



# Session 用于登录

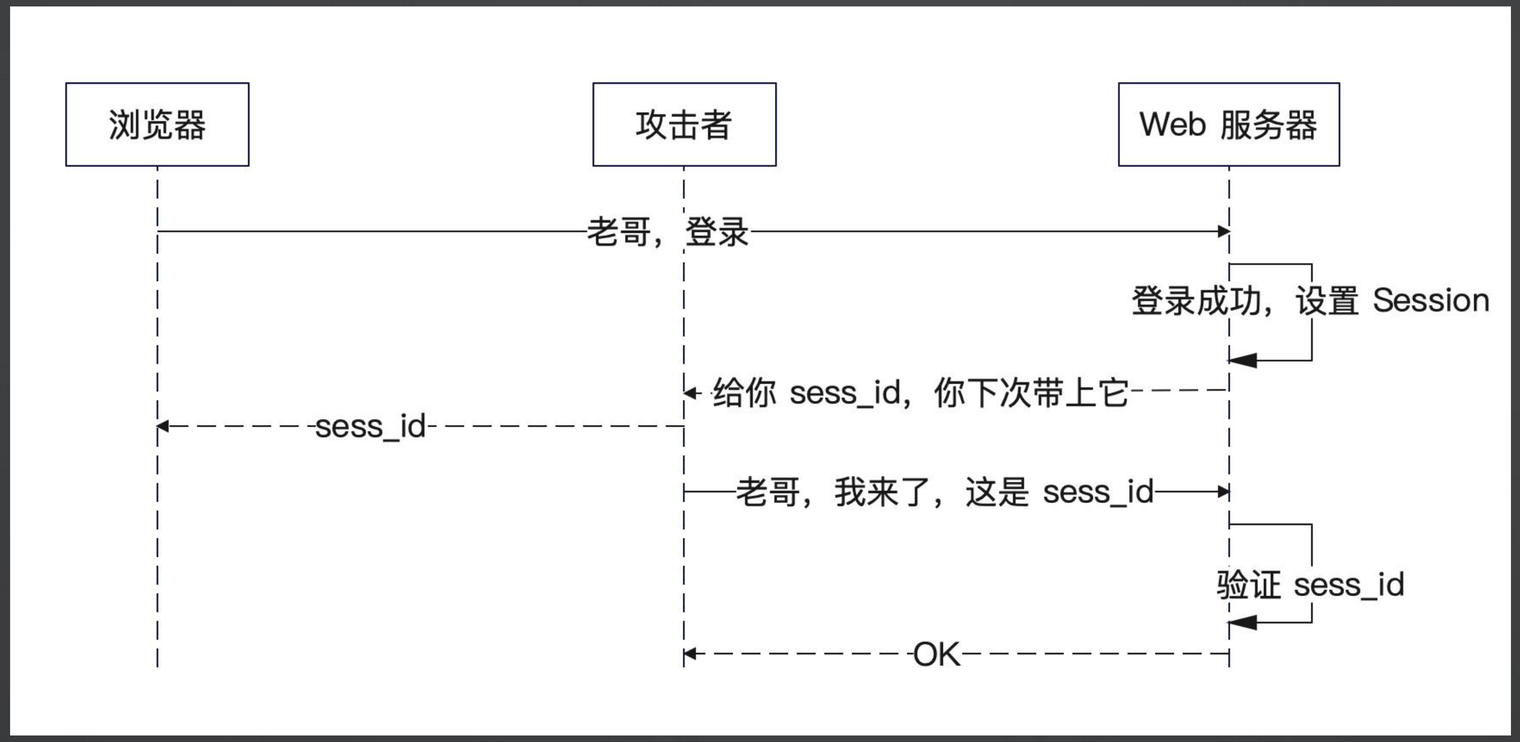
从右边这张图里面也能看出来的，关键在于服务器要给浏览器一个 sess_id，也就是 Session 的 ID。

后续每一次请求都带上这个 Session ID，服务端就知道你是谁了。



# Session 认 ID 不认人

后端服务器是认 ID 不认人的。也就是说，如果攻击者拿到了你的 ID，那么服务器就会把攻击者当成你。在下图中，攻击者窃取到了 sess_id，就冒充是你了。

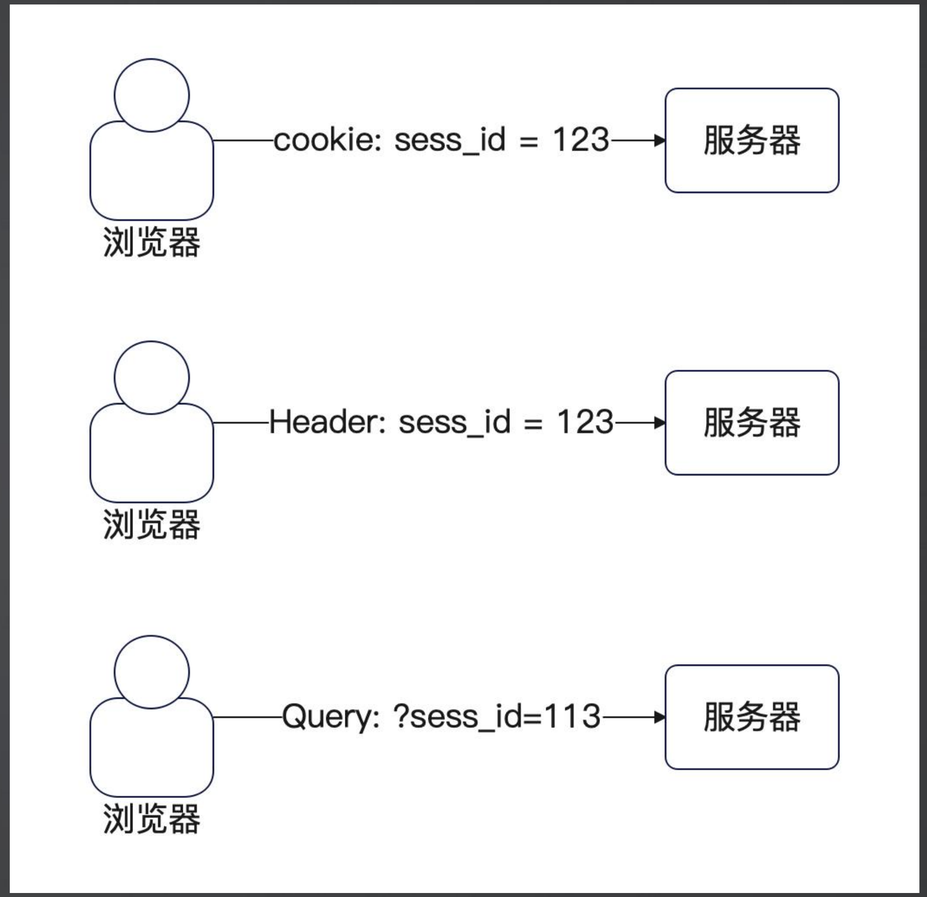


# 如何让客户端携带 sess_id

因为 sess_id 是标识你身份的东西，所以你需要在每一次访问系统的时候都带上。

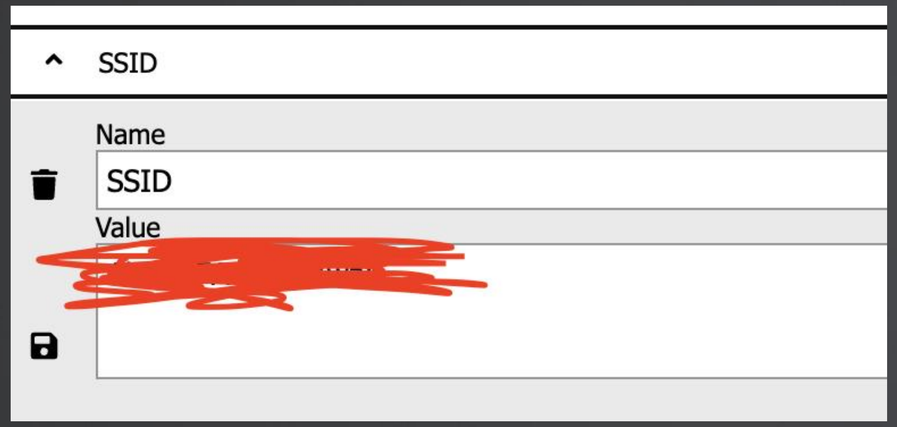
- 最佳方式就是用 Cookie，也就是 sess_id 放到 Cookie 里面。sess_id 自身没有任何敏感信息，所以放 Cookie 也可以。
- 也可以考虑放 Header，比如说在 Header 里面带一个 sess_id。这就需要前端的研发记得在 Header 里面带上。
- 还可以考虑放查询参数，也就是 ?sess_id = xxx。

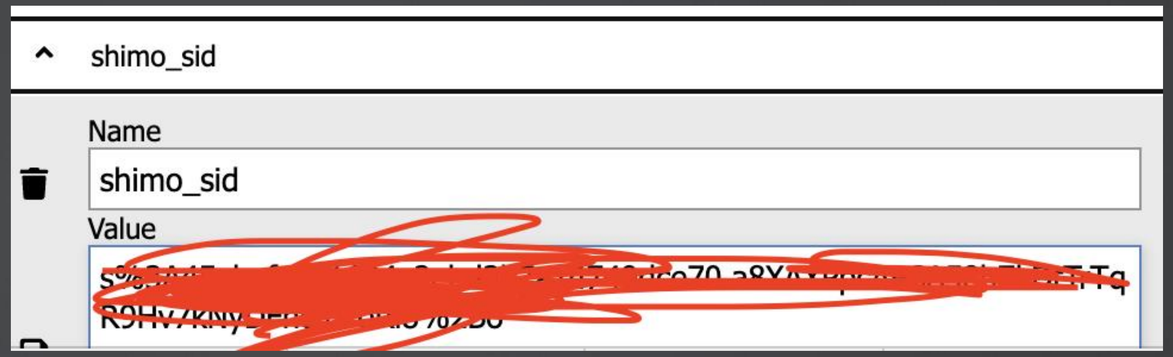
理论上来说还可以放 body，但是基本没人这么干。在一些禁用了 Cookie 功能的浏览器上，只能考虑后两者。



# 几个网站的 Cookie

在浏览器上，你可以通过插件 cookieEditor 来查看某个网站的 Cookie 信息。下面依次是谷歌、石墨文档存放在 Cookie 中的 ssid。



# 使用 Gin 的 Session 插件来实现登录功能

遇事不决找插件，基本上热门的功能 Gin 都是有插件的。这里我们使用 Gin 的 Session 插件来实现登录功能。

https://github.com/gin-contrib/sessions

Gin 的 Session 插件用起来分成两部分：

- 一个是在 middleware 里面接入，它会帮你从 Cookie 里面找到 sess_id，再根据 sess_id 找到对应的 Session。
- 另外一部分就是你拿到这个 Session 之后，就可以为所欲为了，例如这里用来校验是否登录。

# 登录校验实现

```go
func (*LoginMiddlewareBuilder) CheckLogin() gin.HandlerFunc {
    return func(ctx *gin.Context) {
        // 不需要校验
        if ctx.Request.URL.Path == "/users/signup" ||
            ctx.Request.URL.Path == "/users/login" {
            return
        }
        sess := sessions.Default(ctx)
        // 验证一下就可以
        if sess.Get(key: "userId") == nil {
            ctx.AbortWithStatus(http.StatusUnauthorized)
            return
        }
    }
}
```

```go
store := cookie.NewStore([]byte("secret"))
// cookie 的名字叫做ssid
server.Use(sessions.Sessions(name: "ssid", store))
// 登录校验
login := &middleware.LoginMiddlewareBuilder{}
server.Use(login.CheckLogin())
```

# 登录校验实现

在这里我们用了两个 middleware。

- 第一个 Sessions 是 Gin 帮我们提取 Session 的。
- 第二个是我们执行登录校验的。

从右边可以看到，我们这里用了基于 Cookie 的实现来存储数据。

```
store := cookie.NewStore([]byte("secret"))
// cookie 的名字叫做ssid
server.Use(sessions.Sessions( name: "ssid", store))
// 登录校验
login := &middleware.LoginMiddlewareBuilder{}
server.Use(login.CheckLogin())
```

极客时间 训练营

# 升职加薪指南

# 增强扩展 GORM 功能

GORM 本身很强，但是你可以做到更强，后续课程你们会陆续接触到这些插件。

- 为 GORM 提供可观测性的插件实现。
- 为 GORM 提供读写分离插件。
- 为 GORM 提供 BeforeFind 功能。
- 为 GORM 提供辅助方法。

# 为公司引入 SQL 规范和 review 流程

如果你们公司现在还没有任何的 MySQL 使用规范，和 SQL 的 review 流程，那么你可以尝试在公司内部提出建议。

当然，前提是你有技术影响力。

其中 MySQL 规范可以参考 https://developer.aliyun.com/special/tech-java 中的 MySQL 章节。

小公司的 review SQL 可以利用合并请求来达成，尤其是 DDL，最好不要依赖于 GORM 的 AutoMigrate，而是自己手动修改，更加保险一点。

# 面试要点

# 登录流程面试题

- 什么是 Cookie，什么是 Session?
- Cookie 和 Session 比起来有什么缺点?
- Session ID 可以放在哪里？这个问题，你要记得提起 Cookie 禁用的问题。
- 用户密码加密算法选取有什么注意事项？你用的是什么?
- 怎么做登录校验？核心是利用 Gin 的 middleware。

# 作业

# 实现编辑功能

你需要完善 /users/edit 对应的接口。要求：

- 允许用户补充基本个人信息，包括：
  - 昵称：字符串，你需要考虑允许的长度。
  - 生日：前端输入为 1992-01-01 这种字符串。
  - 个人简介：一段文本，你需要考虑允许的长度。
- 尝试校验这些输入，并且返回准确的信息。
- 修改 /users/profile 接口，确保这些信息也能输出到前端。

不要求你开发前端页面。提交作业的时候，顺便提交 postman 响应截图。加一个 README 文件，里面贴个图。

就是补充 live 分支上的 Edit 和 Profile 接口

PS：暂时不要求上传头像，后面我们讲到 OSS 之后直接用 OSS。

极客时间 训练营

# THANKS