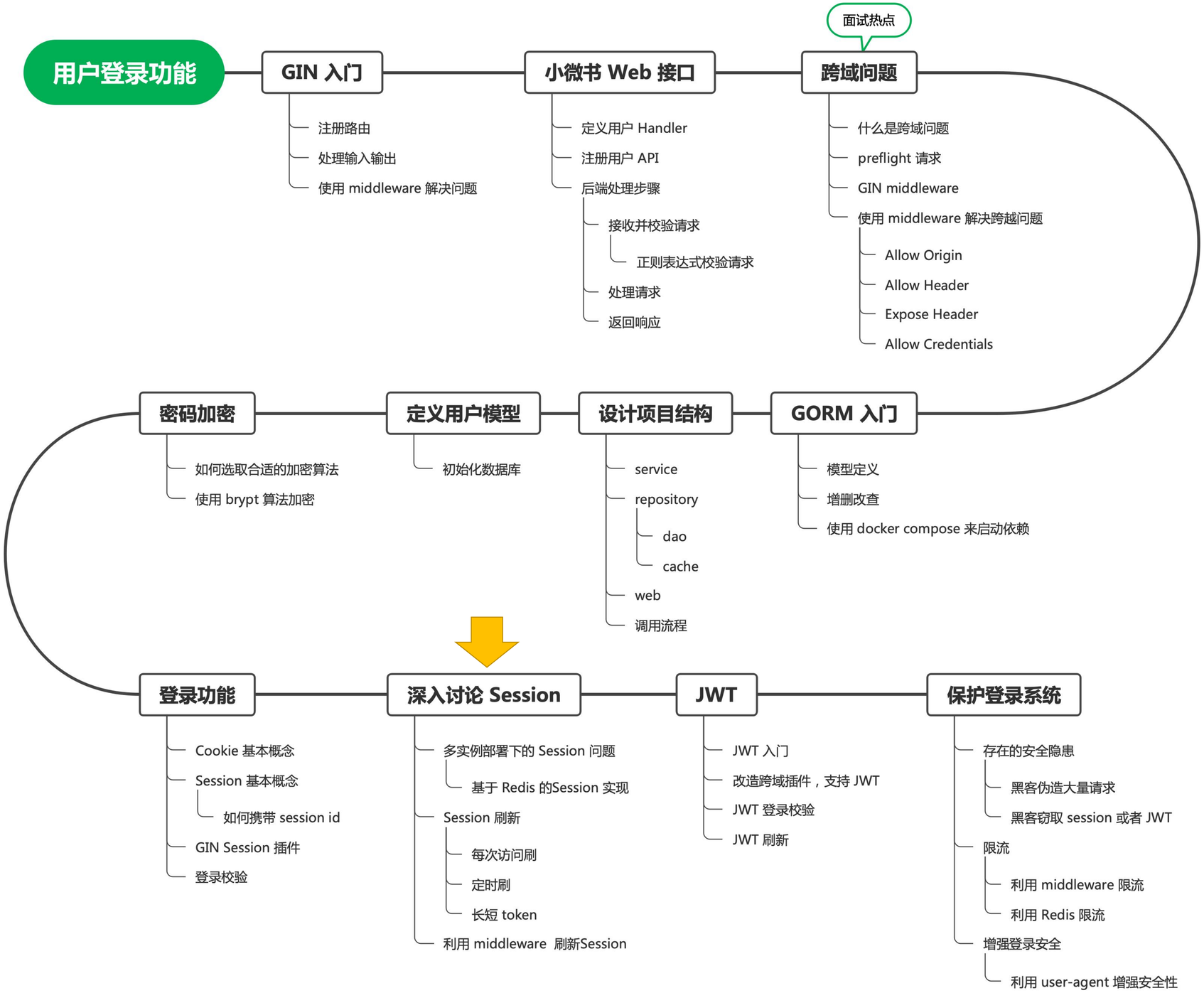
Session 和 JWT

大明

主要内容

- 多实例部署的 Session 问题
- 刷新 Session 的过期时间
- JWT
- 初步保护系统

多实例部署的 Session 问题


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

已有的实现

上节课，我们讲完了最简单的登录和登录校验功能实现。

使用的是基于 Cookie 的实现来保存 Session 数据。

但 Cookie 本身是不安全的，那么还可以考虑怎么办呢？

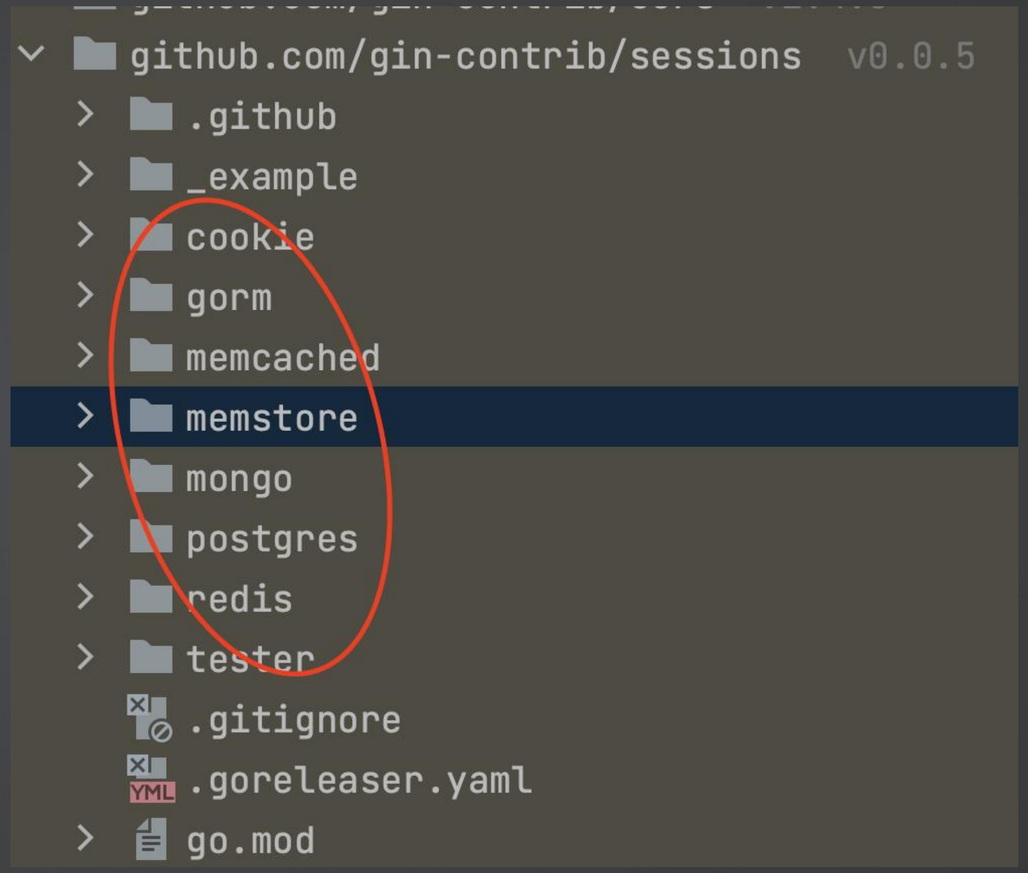
```
store := cookie.NewStore([]byte("secret"))
// cookie 的名字叫做ssid
server.Use(sessions.Sessions( name: "ssid", store))
// 登录校验
login := &middleware.LoginMiddlewareBuilder{}
server.Use(login.CheckLogin())
```

Gin Session 存储的实现

答案是 Gin 本身提供了很多的实现，包括：

- cookie：基于内存的实现
- gorm：基于 GORM 的实现
- memcached：基于 Memcached 的实现
- memstore：基于内存的实现
- mongo：基于 MongoDB 的实现
- postgres：基于 PostgreSQL 的实现
- redis：基于 Redis 的实现
- tester：用于测试的实现

注意不要搞错的一点，就是 sess_id 肯定是放在 cookie 里面的，但是 Session 里面的数据，比如说我们代码里面的 user_id 才是 store 存储的。



Gin Session 存储的实现

也就是说，你可以根据自己的需要来选择。

一般来说，单机单实例部署，你可以考虑 memstore 实现。

多实例部署，你应该选择 redis 实现。

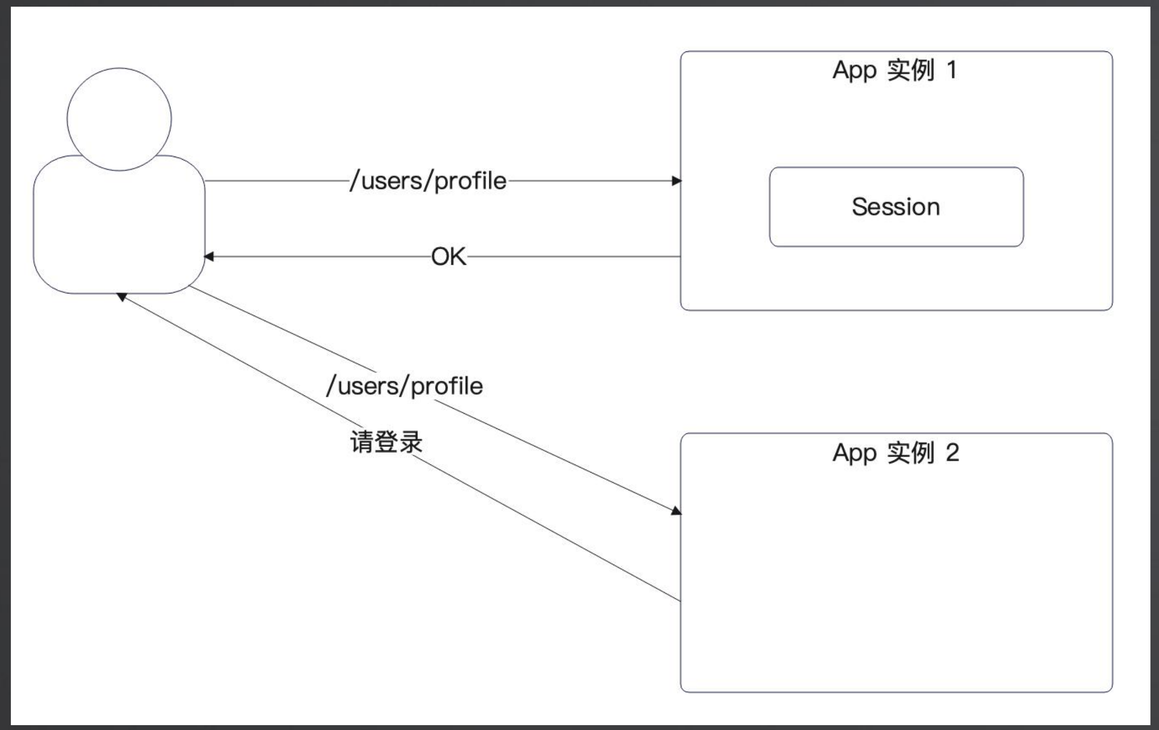
其它实现都比较少使用。

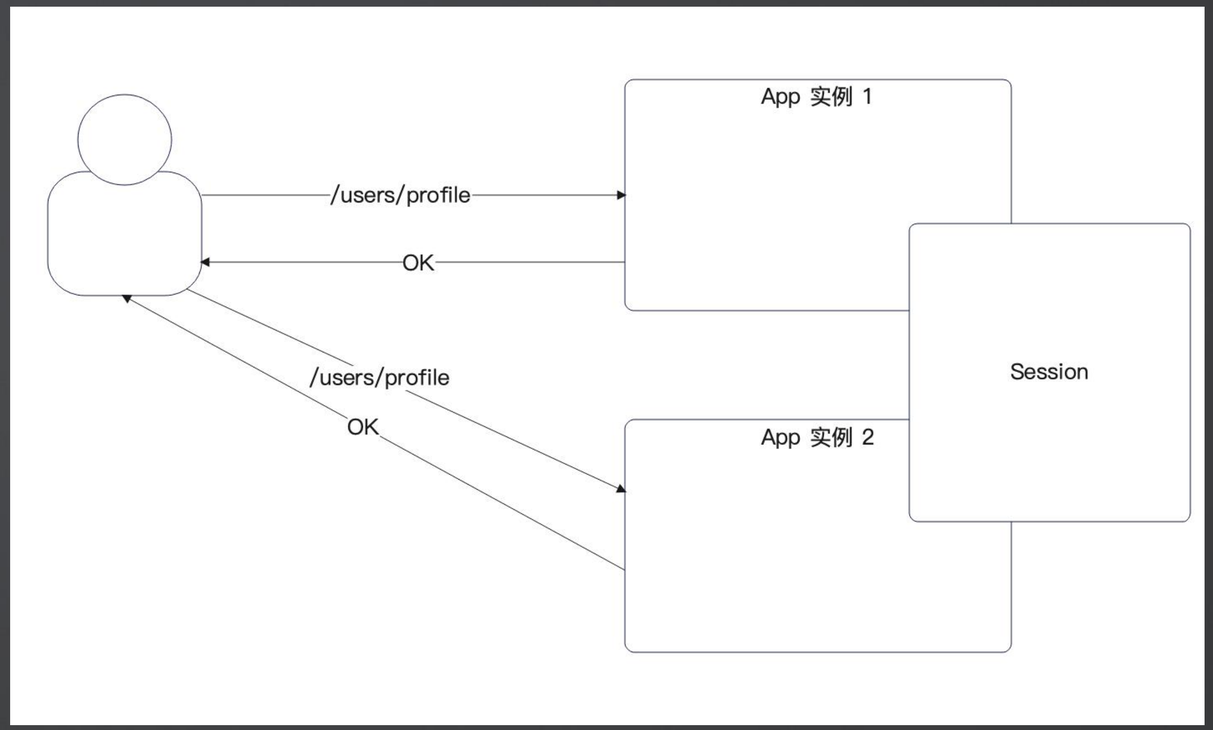
平时无脑选 redis 的实现。

```
// 这是基于内存的实现，第一个参数是 authentication key，最好是 32 或者 64 位
// 第二个参数是 encryption key
store := memstore.NewStore([]byte("moyn8y9abnd7q4zkq2m73yw8tu9j5ixm"),
    []byte("o6jdlo2cb9f9pb6h46fjmllw481ldebj"))
// 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())
return server
```

Gin Session 存储的实现

在分布式环境下（包括单例应用多实例部署）都需要确保， Session 在每一个实例上都可以访问到。





启动 Redis

类似于用 MySQL，我们需要通过 docker compose 来启动一个测试的 Redis。

在 docker-compose 中加入右图中的这一段。各个字段你在 MySQL 中就已经学过了。

这里新出现的选项 ALLOW_EMPTY_PASSWORD 意思就是不用密码。

```
redis:
  image: 'bitnami/redis:latest'
  environment:
    - ALLOW_EMPTY_PASSWORD=yes
  ports:
    - '6379:6379'
```

使用基于 Redis 的实现

注意我在图里标注的各个字段的含义。

这里你应该注意到，在 cookie、memstore 和 redis 三个实现中，都需要传入这两个 key。

Authentication：是指身份认证。
Encryption：是指数据加密。

这两者再加上授权（权限控制），就是信息安全的三个核心概念。

```
// 第一个参数是最大空闲连接数量
// 第二个就是 tcp，你不太可能用 udp
// 第三、四个 就是连接信息和密码
// 第五第六就是两个 key
store, err := redis.NewStore( size: 16, network: "tcp",
    address: "localhost:6379", password: "",
    // authentication key, encryption key
    []byte("movn8y9abnd7q4zkq2m73yw8tu9j5ixm"),
    []byte("o6jdlo2cb9f9pb6h46fjmllw481ldebj"))
if err != nil {
    panic(err)
}
```

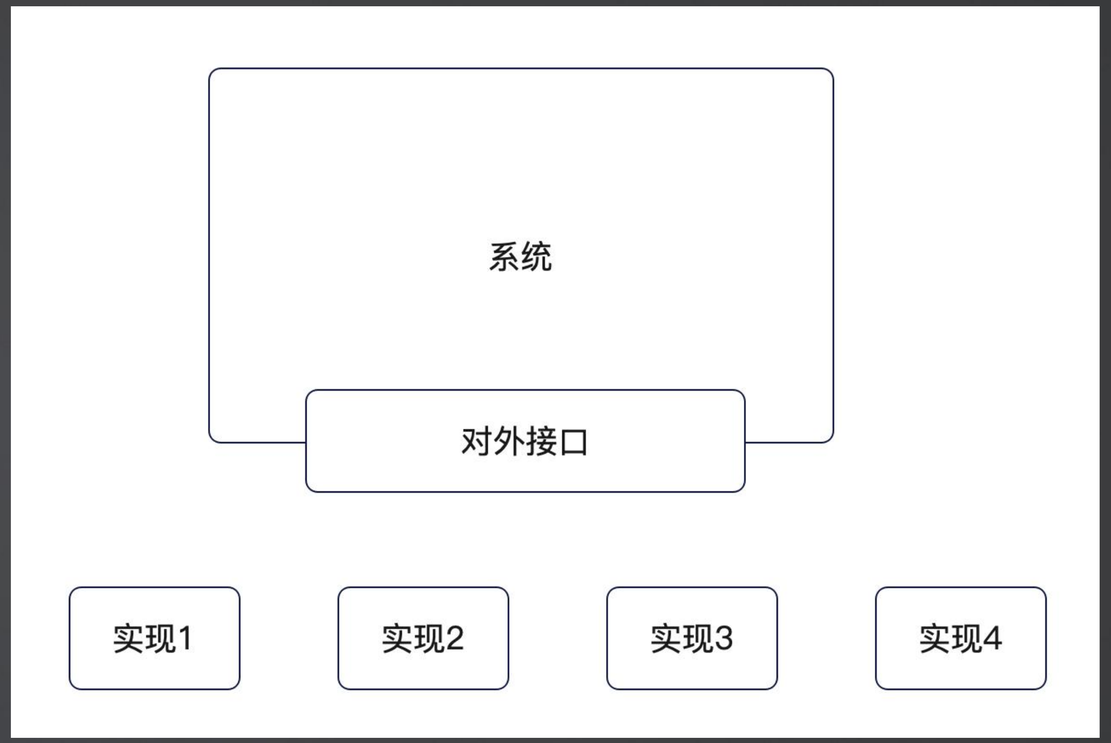
自由切换的好处

你这是第一次感受到面向接口编程的好处。

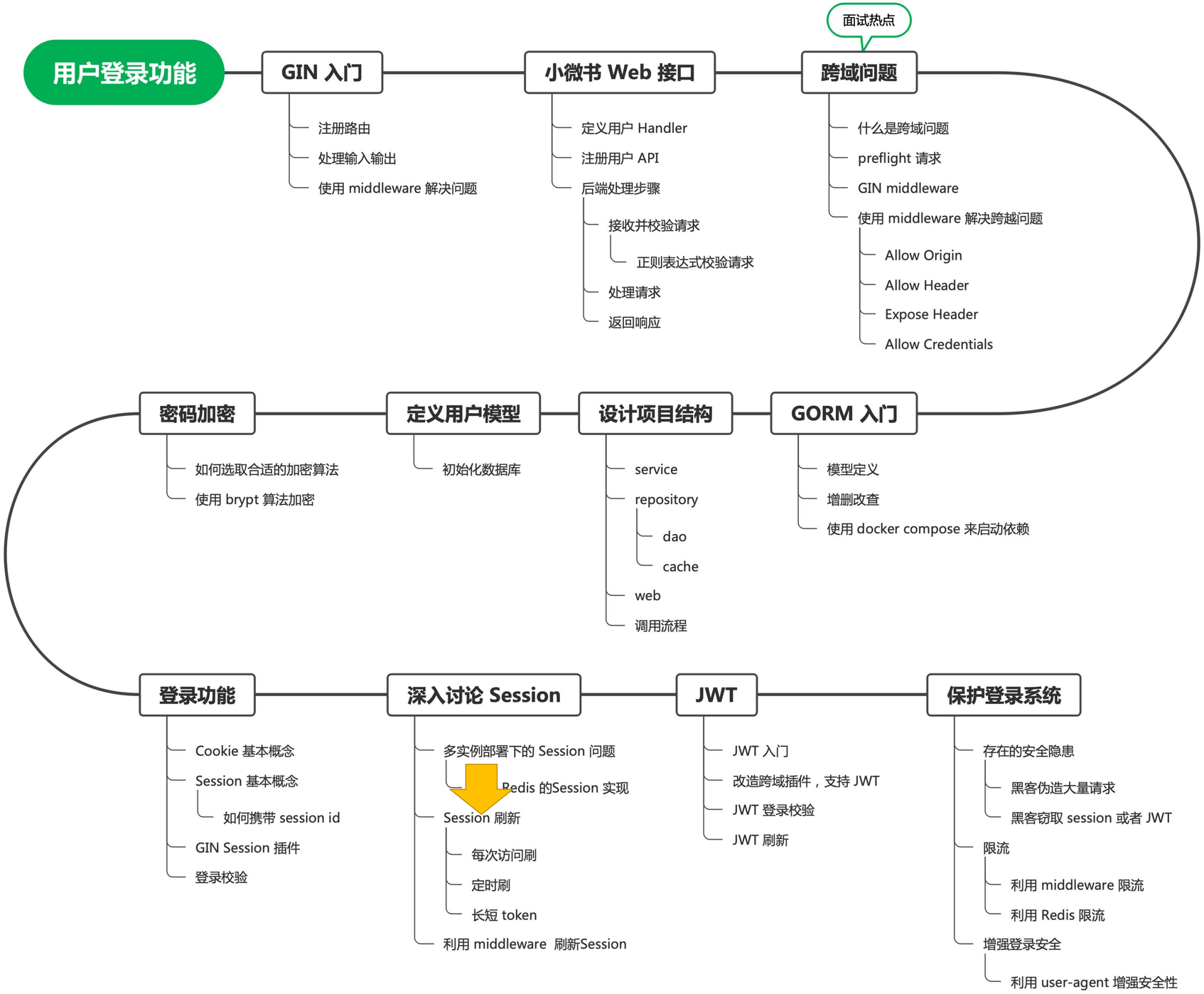
在 Gin 的 Session 设计里面，怎么存储你的 Session 数据，被抽象成了一个接口。

而后，Gin 提供了不同的实现，于是你可以自由切换了。

当你在设计核心系统的时候，或者你打算提供什么功能给用户的时候，一定要问问自己，将来有没有可能需要不同的实现。



Session 参数与刷新


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
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

Gin Session 参数

Gin 的 Session 有很多参数，你可以通过 Options 方法来传入 Option。

```
sess := sessions.Default(ctx)
sess.Set(userIdKey, u.Id)
sess.Options(sessions.Options{
    // 60 秒过期
    MaxAge: 60,
})
err = sess.Save()
```

Gin Session 参数

右边的参数中，除了 MaxAge 有多层含义，其它参数就是你在 Cookie 中学到的那个含义。

或者你可以理解为，Gin 的 Session 用这些选项来初始化 Cookie。

MaxAge 则不同，它一方面用来控制 Cookie，而有一些实现，也用它来控制 Session 中的 key、value 的过期时间。

比如 Redis，它会用这个来控制你的数据的过期时间。

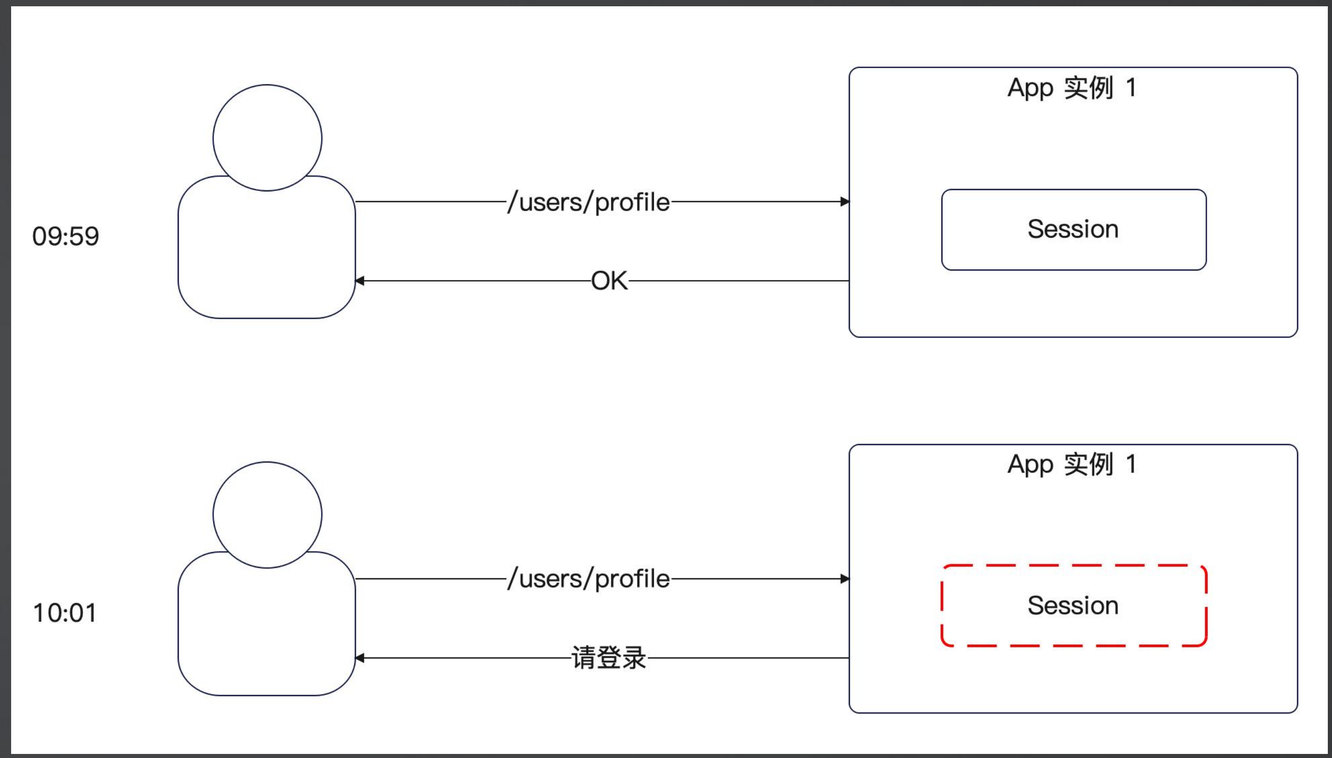
```
// Options stores configuration for a sessi
// Fields are a subset of http.Cookie field
type Options struct {
    Path   string
    Domain string
    // MaxAge=0 means no 'Max-Age' attribut
    // MaxAge<0 means delete cookie now, eq
    // MaxAge>0 means Max-Age attribute pre
    MaxAge   int
    Secure   bool
    HttpOnly bool
    // rfc-draft to preventing CSRF: https:
    //   refer: https://godoc.org/net/http
    //          https://www.sjoerdlangkempe
    SameSite http.SameSite
}
```

刷新登录状态

这里有一个登录状态很经常遇到的问题，就是我们这 Session id 所在的 Cookie，过期时间是固定的。

举个例子：假如你设置为 10 分钟，那么用户登录了 9:59 秒之后，还能访问网站，结果过了两秒，他就被要求重新登录。

也就是你需要在用户持续使用网站的时候，刷新过期时间。



如何刷新？

- 用户每次访问，我都刷新。
  - 性能差，对 Redis 之类的影响很大。

- 快要过期了我再刷新，比如说 10 分钟过期。当用户第 9 分钟访问过来的时候，我就刷新。
  - 万一我在第 9 分钟以后都没再访问过呢？

- 固定间隔时间刷新，比如说每分钟内第一次访问我都刷新。

- 使用长短 token。这个我们在后面接入微信登录的时候再深入讨论。

在 Middleware 中刷新

一个简单的道理，就是你肯定不想在所有的 HTTP 接口里面都手动刷新过期时间。

而刷新过期时间显然也是一个大部分业务都要完成的，因此最适合的地方肯定是在 middleware 里面。

于是你想到，我们有一个登录校验的 middleware，显然可以在登录校验之后顺手刷新一下。

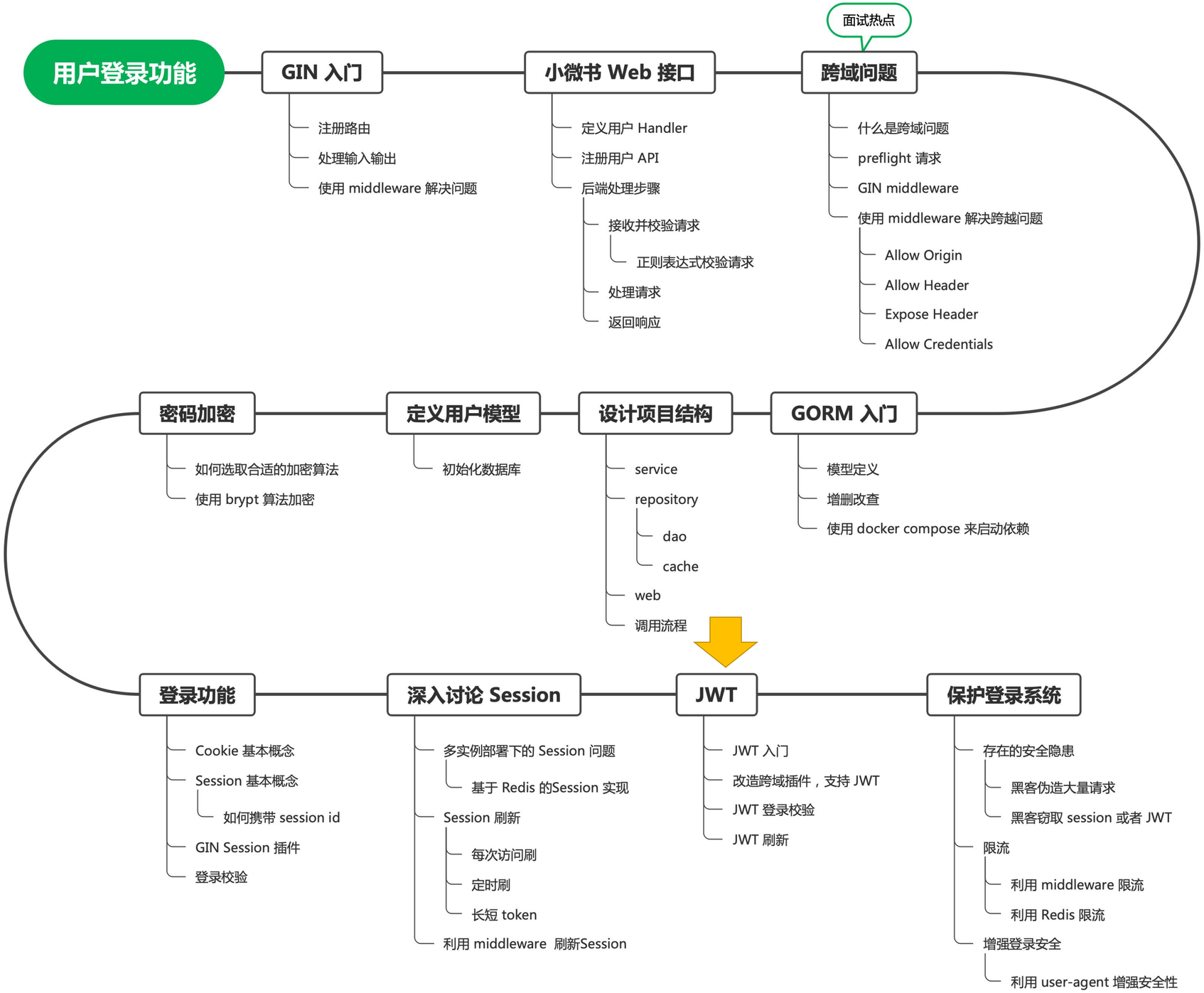
登录状态保持多久比较好？

登录状态保持多久比较好？也就是，一次登录之后，要隔多久才需要继续登录？

答案是取决于你的产品经理，也取决于你系统其它方面的安全措施。

简单来说，就是如果你有别的验证用户身份的机制，那么你就可以让用户长时间不需要登录。

JWT


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
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
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

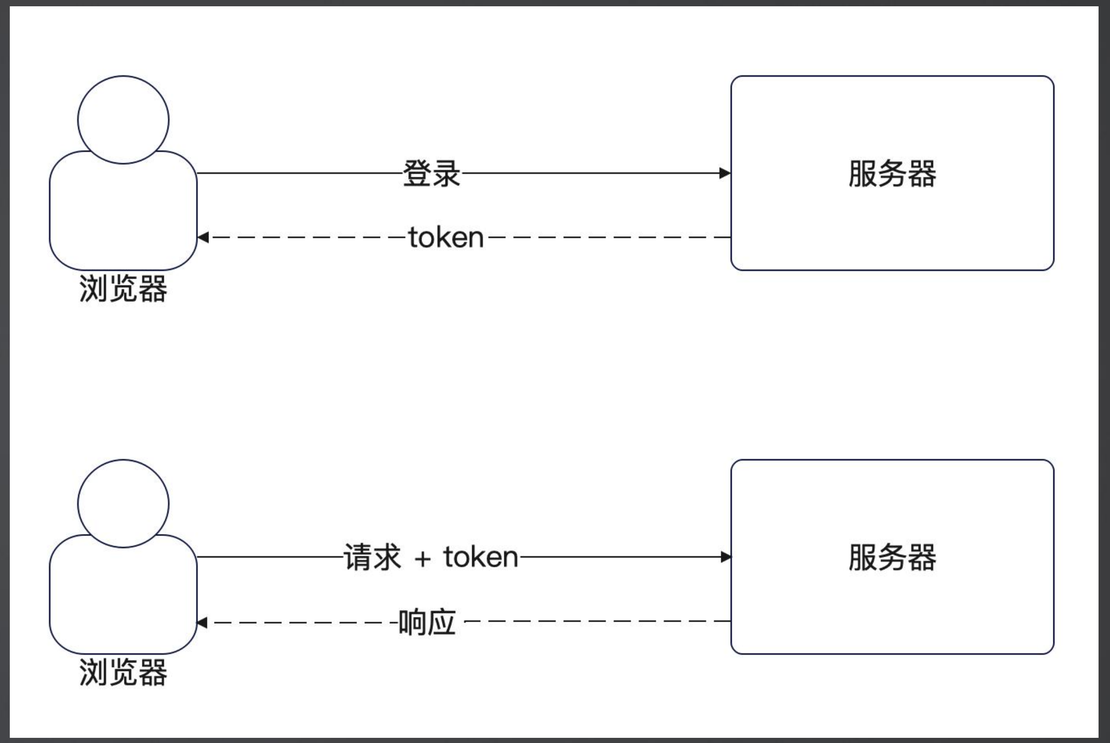
JWT 简介

前面我们实现的登录功能，是直接依赖于 Gin 的 Session 插件达成的。

我们还可以考虑使用 JWT 来实现登录功能。

JWT（JSON Web Token）是很常用的一种机制，主要用于身份认证，也就是登录。

它的基本原理就是通过加密生成一个 token，而后客户端每次访问的时候都带上这个 token。



JWT 简介

它由三部分组成：

- Header：头部，JWT 的元数据，也就是描述这个 token 本身的数据，一个 JSON 对象。
- Payload：负载，数据内容，一个 JSON 对象。
- Signature：签名，根据 header 和 token 生成。

```
eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.ey
JzdWIiOiIxMjM0NTY3ODkwIiwibmFtZSI6Ikpva
G4gRG9lIiwiaWF0IjoxNTE2MjM5MDIyfQ.SflKx
wRJSMeKKF2QT4fwpMeJf36POk6yJV_adQssw5c
```

JWT 使用

虽然有人为 Gin 提供了 JWT 插件，但是那个插件过于复杂并且不好用。而且它屏蔽了和 JWT 有关的操作，所以这次我们直接使用 JWT 原始 API。

```
go get github.com/golang-jwt/jwt/v5
```

在登录过程中，使用 JWT 也是两步：

- JWT 加密和解密数据。
- 登录校验。

JWT 改造 Login 接口

在登录成功后，我们使用 JWT 来写入数据。

注意，这里我们通过一个 x-jwt-token 的头部，返回了给前端。

其中 UserClaims 是我们在 JWT 里面放的数据。

我们希望前端在请求的 Authorization 头部带上 Bearer token。

```
}
token := jwt.NewWithClaims(jwt.SigningMethodHS256, UserClaims{
    Id: u.Id,
    RegisteredClaims: jwt.RegisteredClaims{
        // 演示目的设置为一分钟过期
        ExpiresAt: jwt.NewNumericDate(time.Now().Add(time.Minute)),
    },
})
tokenStr, err := token.SignedString(JWTKey)
if err != nil {
    ctx.String(http.StatusOK, format: "系统异常")
    return
}
ctx.Header( key: "x-jwt-token", tokenStr)
ctx.String(http.StatusOK, format: "登录成功")
```

JWT 改造跨域设置

我们的约定是，后端在 x-jwt-token 里面返回 token，前端在 Authorization 里面带上 token。

所以需要改造 AllowHeaders 和 ExposeHeaders。

```
server.Use(cors.New(cors.Config{
    AllowCredentials: true,
    // 在使用 JWT 的时候，因为我们使用了 Authorization 的头部，所以要加上
    AllowHeaders: []string{"Content-Type", "Authorization"},
    // 为了 JWT
    ExposeHeaders: []string{"X-Jwt-Token"},
    AllowOriginFunc: func(origin string) bool {
        if strings.HasPrefix(origin, prefix: "http://localhost") : true
        return strings.Contains(origin, substr: "your_company.com")
    },
    MaxAge: 12 * time.Hour,
}))
```

JWT 登录校验

对应的 JWT 登录校验也要刷新 token 的过期时间。

```
// Authorization 头部
// 得到的格式 Bearer token
authCode := ctx.GetHeader( key: "Authorization")
if authCode == "" {...}
// SplitN 的意思是切割字符串，但是最多 N 段
// 如果要是 N 为 0 或者负数，则是另外的含义，可以看它的文档
authSegments := strings.SplitN(authCode, sep: " ", n: 2)
if len(authSegments) != 2 {...}

                                    这一步，拿到 token
tokenStr := authSegments[1]
uc := web.UserClaims{}
token, err := jwt.ParseWithClaims(tokenStr, &uc, func(toke
    return web.JWTKey, nil
})
```

```
expireTime, err := uc.GetExpirationTime()          校验
if err != nil {...}
if expireTime.Before(time.Now()) {...}
// 每 10 秒刷新一次
if expireTime.Sub(time.Now()) < time.Second*50 {
    uc.ExpiresAt = jwt.NewNumericDate(time.Now().Add(time.Minute))
    newToken, err := token.SignedString(web.JWTKey)   刷新
    if err != nil {
        // 因为刷新这个事情，并不是一定要做的，所以这里可以考虑打印日志
        // 暂时这样打印
        log.Println(err)
    } else {
        ctx.Header( key: "x-jwt-token", newToken)
    }
```

前端携带 JWT

对应的前端每次访问都带上 JWT 的 token。

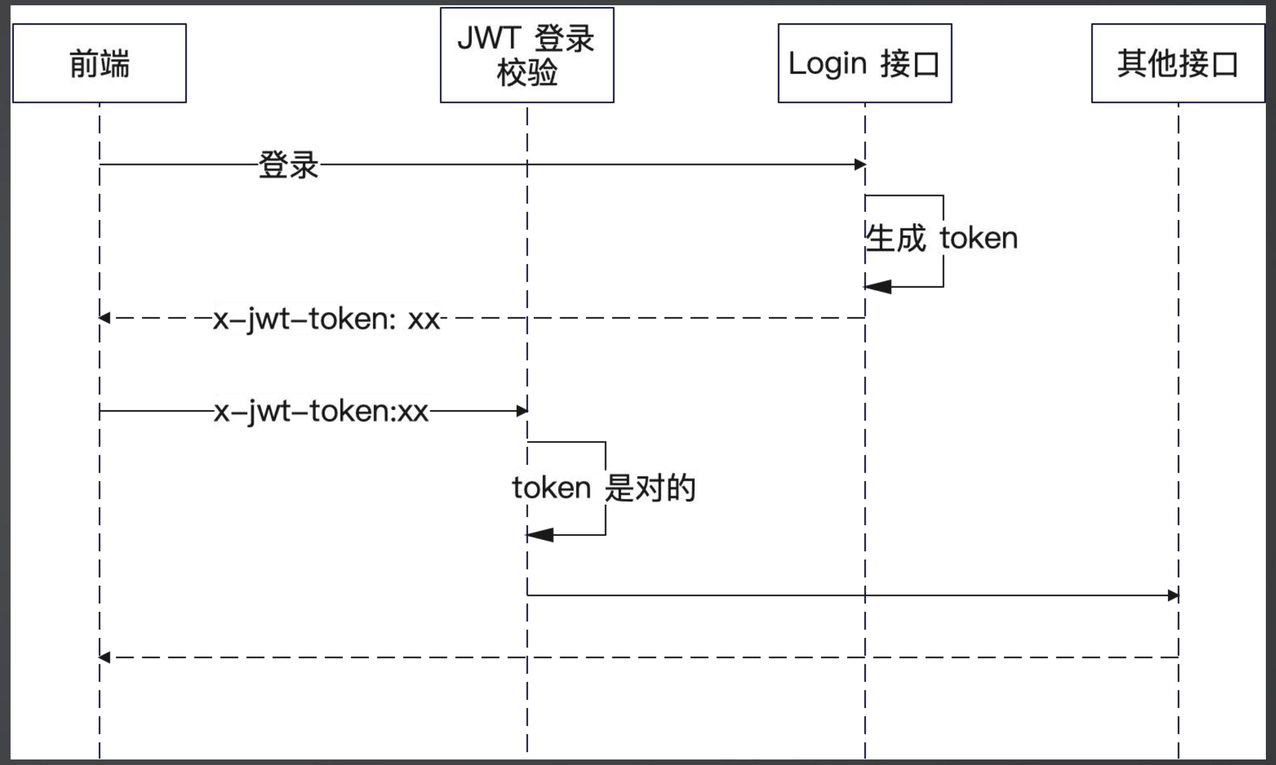
这部分你看一下就可以，不需要理解。

```
instance.interceptors.response.use( onFulfilled: function
    const newToken = resp.headers["x-jwt-token"]
    console.log("resp headers", resp.headers)
    console.log("token" + newToken)
    if (newToken) {
        localStorage.setItem("token", newToken)
    }
    if (resp.status == 401) {
        window.location.href="/users/login"
    }
    return resp
```

```
// 在这里让每一个请求都加上 authorization 的头部
instance.interceptors.request.use( onFulfilled: (req : InternalAxiosRequest
    const token : string | null  = localStorage.getItem( key: "token")
    req.headers.setAuthorization( value: "Bearer " + token,  rewrite: true)
    return req
}, onRejected: (err) : void  => {
    console.log(err)
})
```

接入 JWT 的步骤总结

- 要在 Login 接口中，登录成功后生成 JWT token。
  - 在 JWT token 中写入数据。
  - 把 JWT token 通过 HTTP Response Header x-jwt-token 返回。
- 改造跨域中间，允许前端访问 x-jwt-token 这个响应头。
- 要接入 JWT 登录校验的 Gin middleware。
  - 读取 JWT token。
  - 验证 JWT token 是否合法。
- 前端要携带 JWT token。



使用 JWT 的优缺点

和 Session 比起来，优点：

- 不依赖于第三方存储。
- 适合在分布式环境下使用。
- 提高性能（因为没有 Redis 访问之类的）。

缺点：

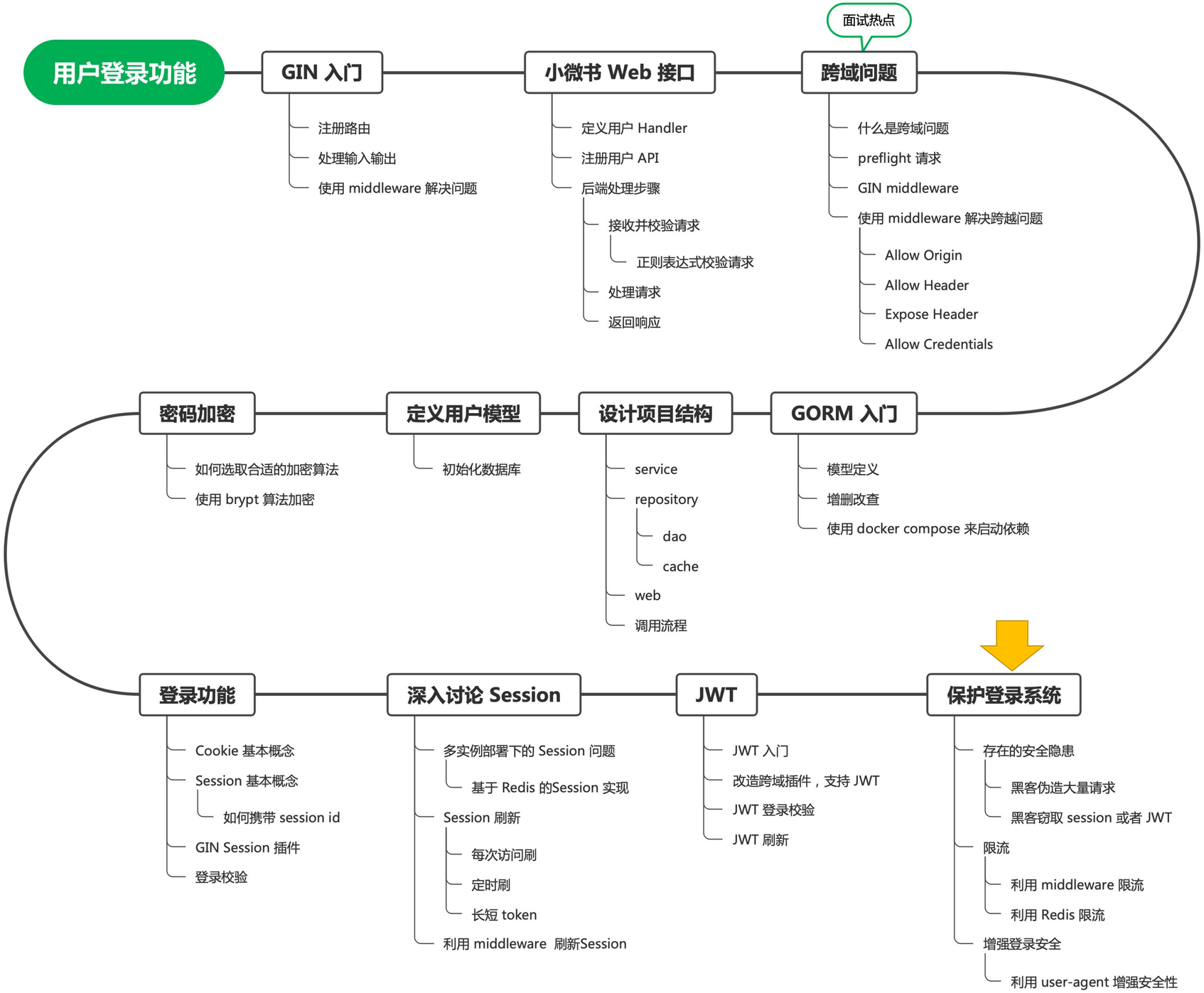
- 对加密依赖非常大，比 Session 容易泄密。
- 最好不要在 JWT 里面放置敏感信息。

混用 JWT 和 Session 机制

前面 JWT 限制了我们不能使用敏感数据，那么你真有类似需求的时候，就可以考虑将数据放在“Session”里面。

基本的思路就是：你在 JWT 里面存储你的 userId，然后用 userId 来组成 key，比如说 user.info:123 这种 key，然后用这个 key 去 Redis 里面取数据，也可以考虑使用本地缓存数据。

保护系统


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

保护系统

在功能完成之后，现在要进一步考虑保护我们的系统。

一般来说，你要考虑两方面的事情：

- 正常用户会不会搞崩你的系统？
- 如果有人攻击你的系统，你能撑住吗？

对于中小型公司来说，第一条不会是问题。

对于大公司来说，就要两条都考虑。

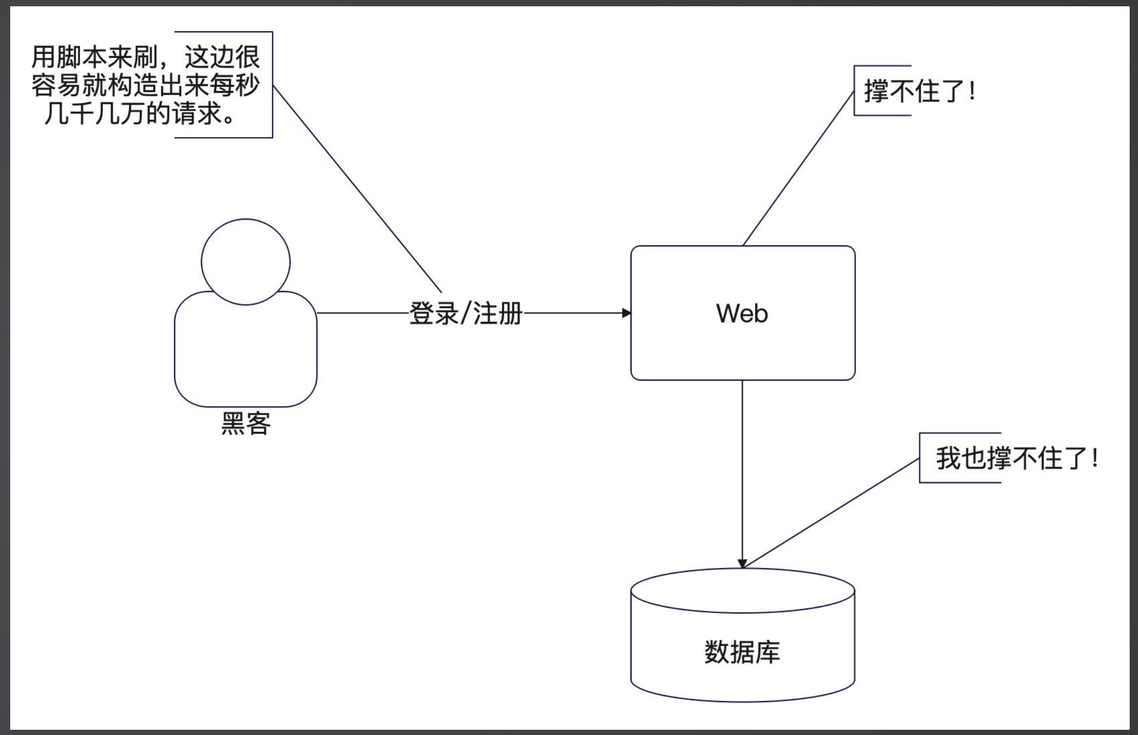
系统的漏洞

现在我们的系统最明显的漏洞就是：

- 任何人都可以注册。
- 任何人都可以登录。

也就是说，万一有一个人，用 shell 脚本拼命给你发注册请求、登录请求，系统负载就会很高。

而且这两个请求都会查询数据库，也就是说数据库负载也很高。

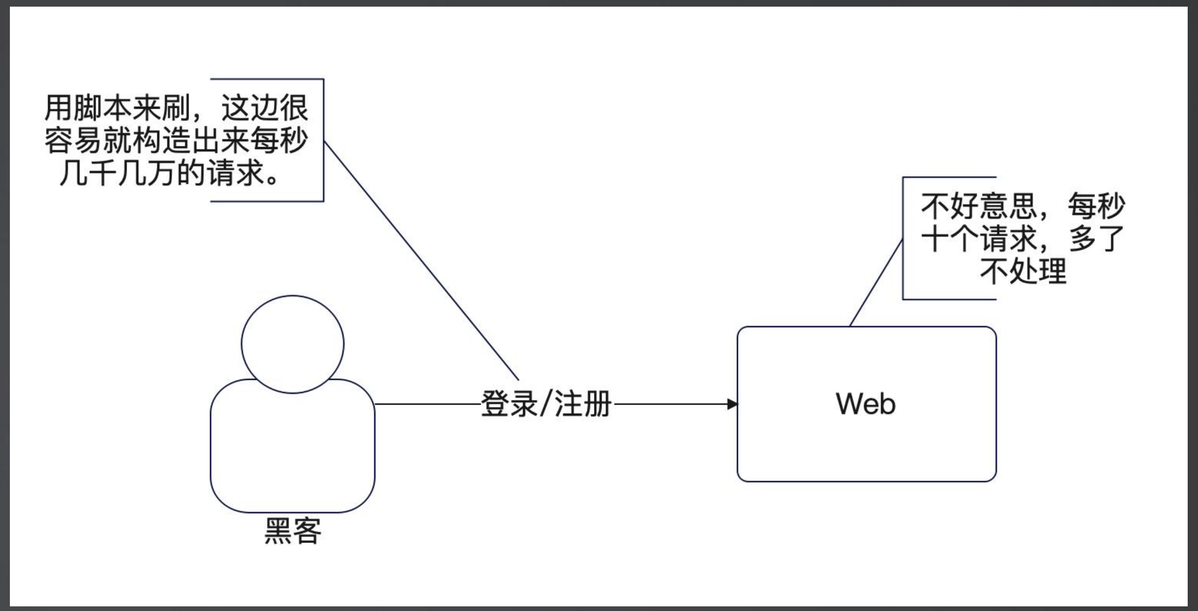


怎么办？

这个时候，我们可以考虑，能不能限制住每个人发的请求数量？

又或者，限制住系统处理的请求数量？

这就是限流。



限流

限流是最常见的保护系统的办法。限流有很多算法，但是都大同小异，后面在微服务架构部分会进一步讲解。

这里我们使用限流，限制每一个用户，每秒最多发送固定数量的请求。

所以问题来了：

- 我怎么认定谁是谁？尤其是在登录和注册这个接口里，都还没登录成功，我都不知道他是谁。
- 我怎么确定我限流的这个阈值应该是多少？每秒 100 还是每秒 200？

限流对象

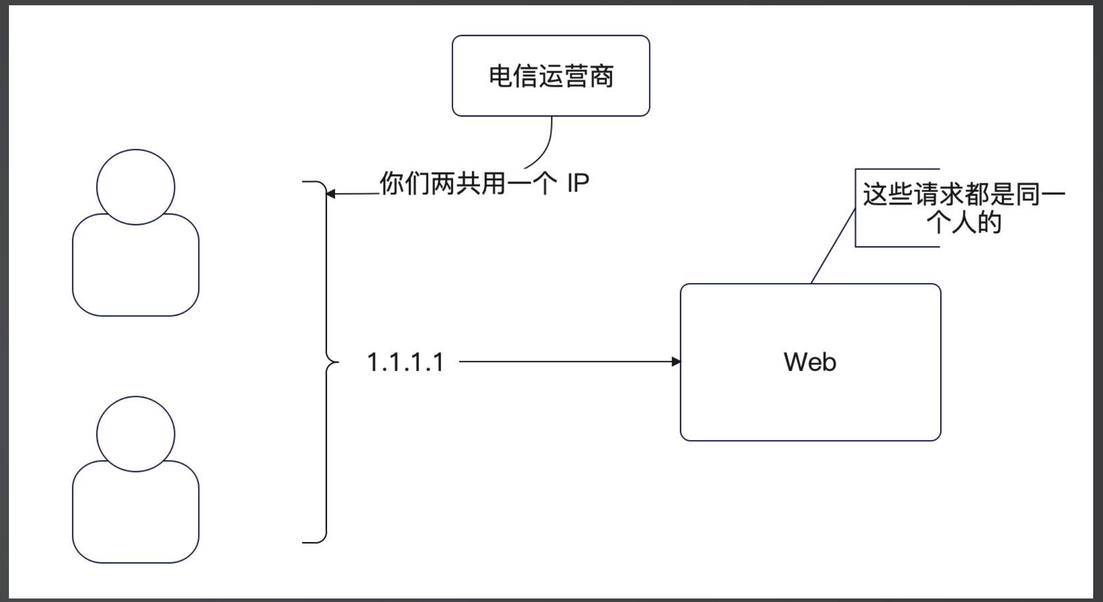
第一个问题的答案是：用 IP。也就是说我们的限流针对的是 IP。

IP 虽然并不能实际意义上代表一个人，但这已经是我们比较好的选择了。

更好的选择是用 MAC 地址或者设备标识符之类的，比如说 CPU 序列号，但是在 Web 端很少用。

APP 端就可以考虑用设备序列号。

当然，在使用 IP 的情况下，我们可能会误把不同的人看成是同一个人。但是只要我们限制的阈值不是很小，就不会有问题。

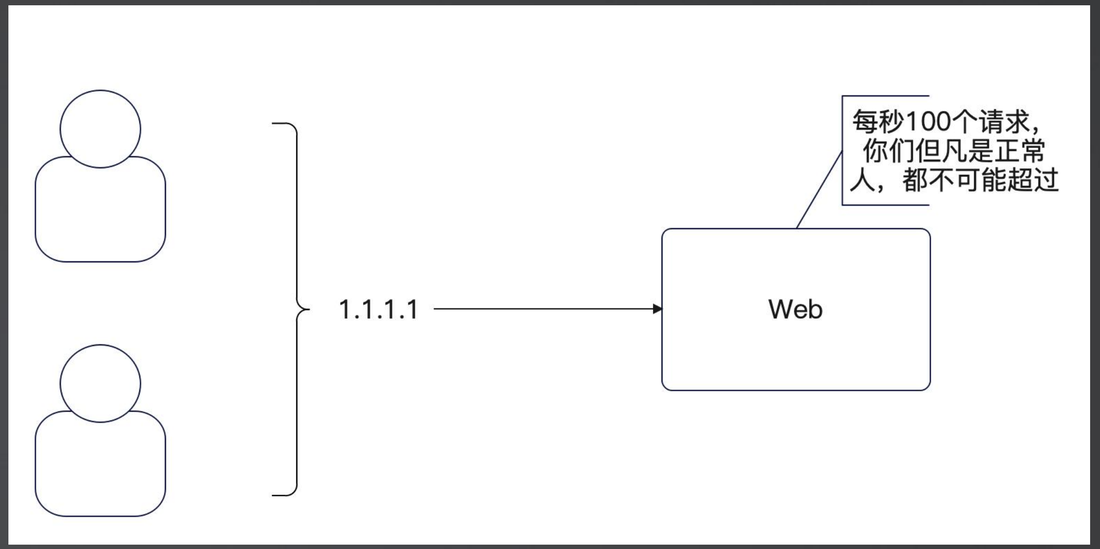


限流阈值

限流阈值应该是多少？

理论上来说，这应该是通过压测来得到的（面试回答这个）。比如说你压测整个系统，发现最多只能撑住每秒 1000 个请求，那么阈值就是 1000。

而我们是针对个人，搞不了压测。所以可以凭借经验来设置，比如说我们正常人手速，一秒钟撑死一个请求，那么就算我们考虑到共享 IP 之类的问题，给个每秒 100 也已经足够了。

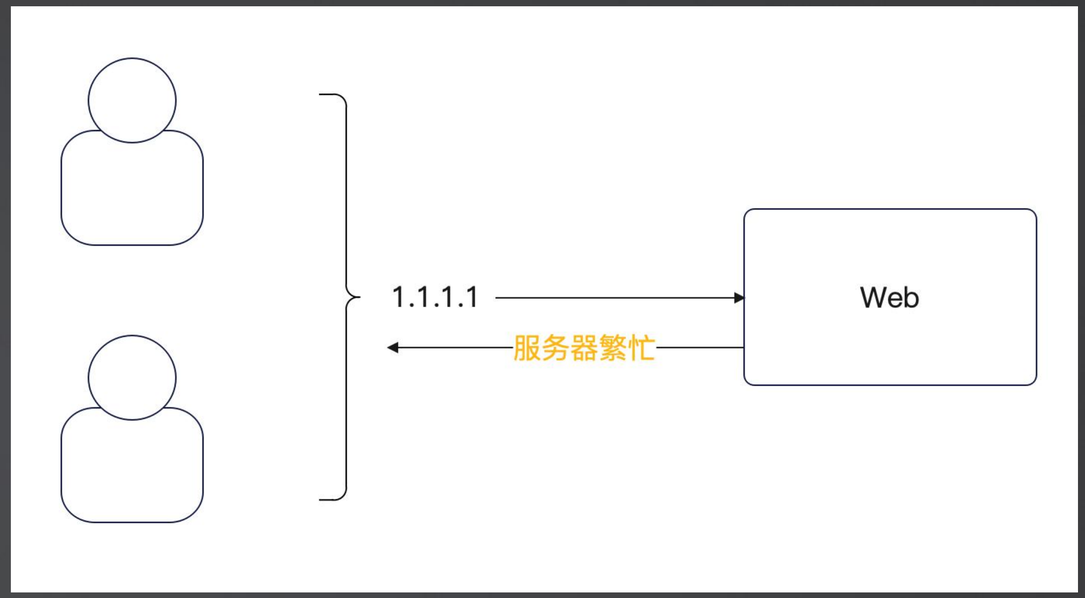


被限流的请求怎么办？

如果我每秒处理 100 个请求，那第 101 个请求过来怎么办？

显然，只能拒绝了，也就是返回错误。

这个错误，不同公司有不同的规范。如果你自己决策的话，可以返回什么服务器繁忙之类的信息。



使用 Gin 的限流插件

Gin 里面有很多限流插件，从功能和非功能特性上，它们都没有什么区别。

但是你们要小心并发问题。

这里我用我纯手写的一个。

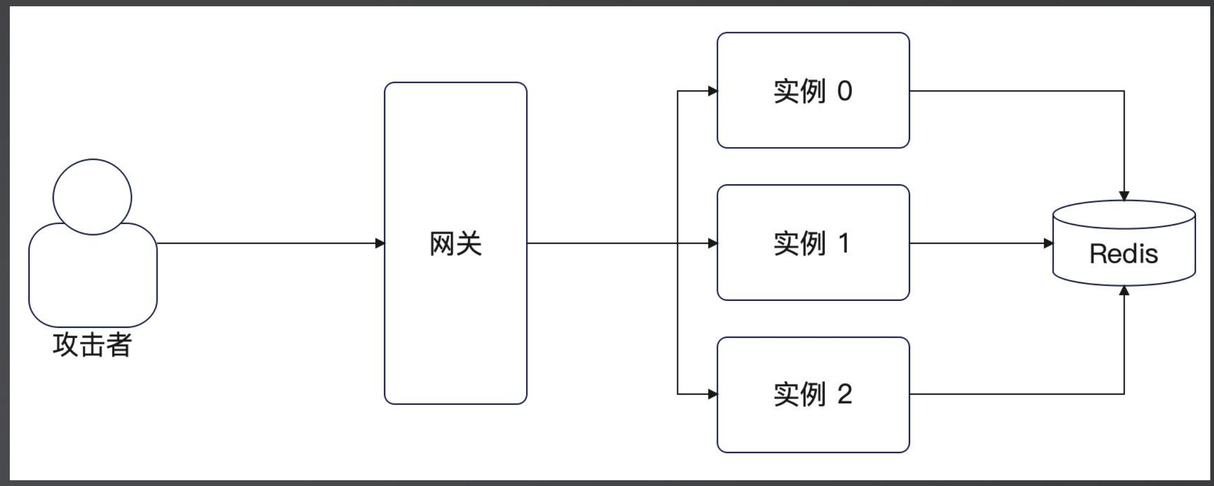
```
cmd := redis.NewClient(&redis.Options{
    Addr:     "localhost:6379",
    Password: "",
    DB:       1,
})
// 一分钟 100 次。
server.Use(ratelimit.NewBuilder(cmd, time.Minute, rate: 100).Build())
```

为啥用 Redis 限流？

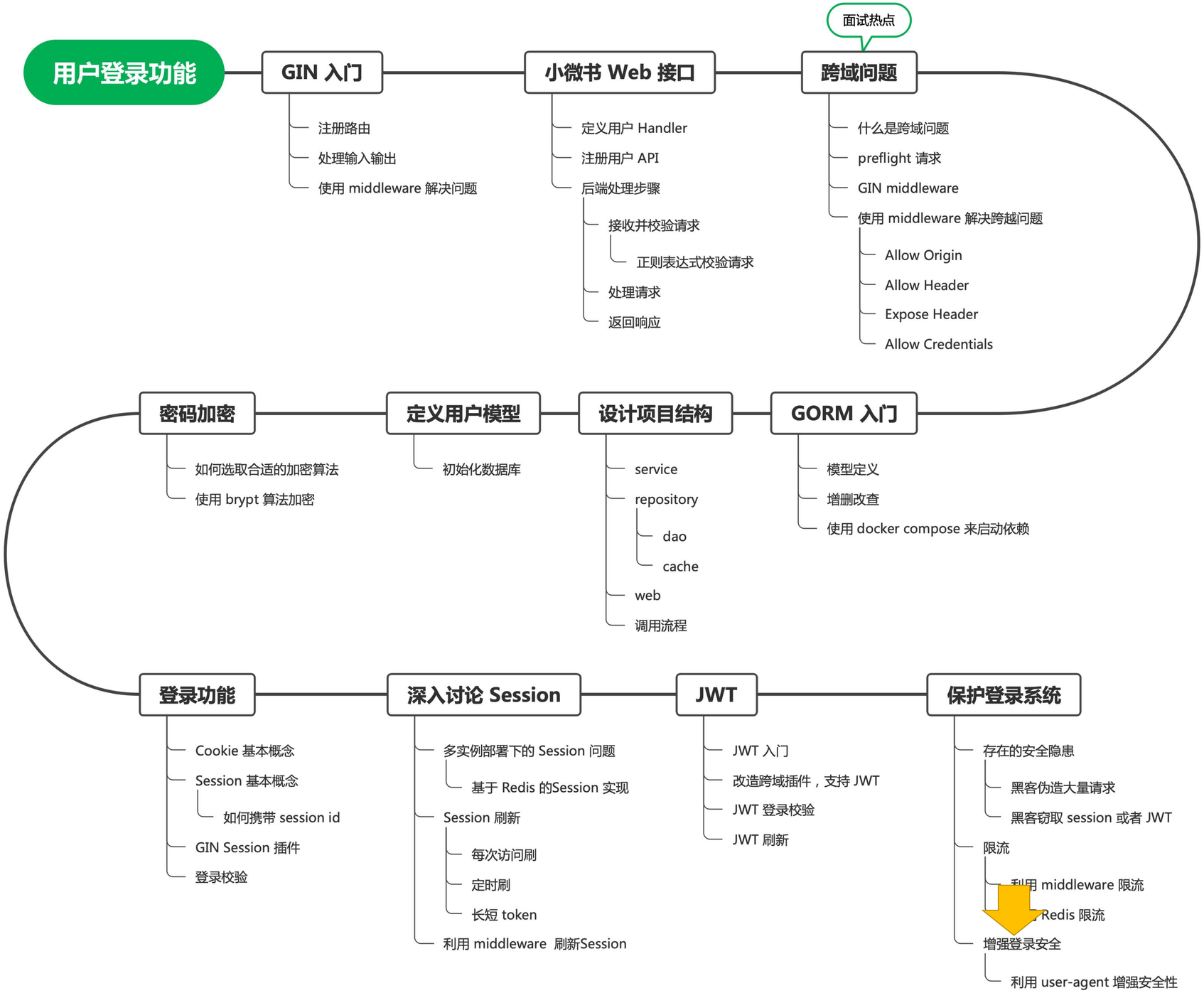
为什么要基于 Redis 来实现呢？

因为你要考虑到整个单体应用部署多个实例，用户的请求经过负载均衡之类的东西之后，就不一定落到同一个机器上了。

因此需要 Redis 来计数。



增强登录安全

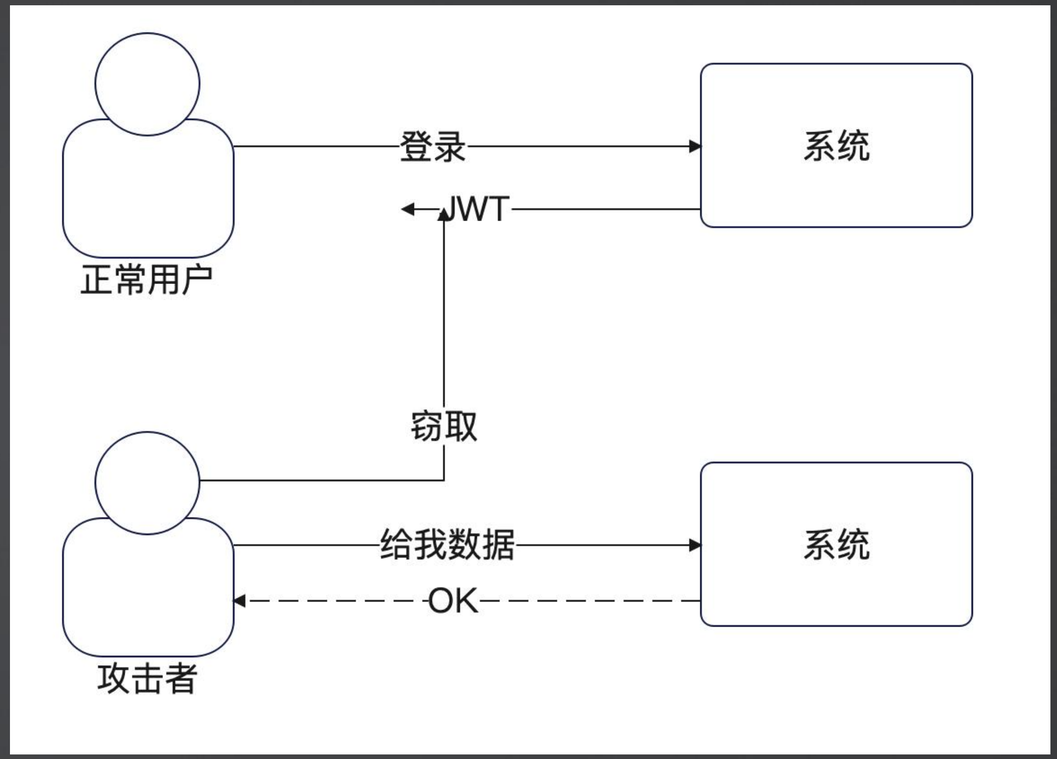

用户登录功能
GIN 入门
注册路由
处理输入输出
使用 middleware 解决问题
小微书 Web 接口
定义用户 Handler
注册用户 API
后端处理步骤
接收并校验请求
正则表达式校验请求
处理请求
返回响应
面试热点
跨域问题
什么是跨域问题
preflight 请求
GIN middleware
使用 middleware 解决跨越问题
Allow Origin
Allow Header
Expose Header
Allow Credentials
GORM 入门
模型定义
增删改查
使用 docker compose 来启动依赖
设计项目结构
service
repository
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
Redis 限流
增强登录安全
利用 user-agent 增强安全性

安全问题

现在的问题是，不管是用 JWT 还是 Session，一旦被攻击者拿到关键的 JWT 或者 ssid，攻击者就能假冒你。

HTTPS 可以有效阻止攻击者拿到你的 JWT 或者 ssid。

但是如果你电脑中了病毒，那 HTTPS 就无能为力。

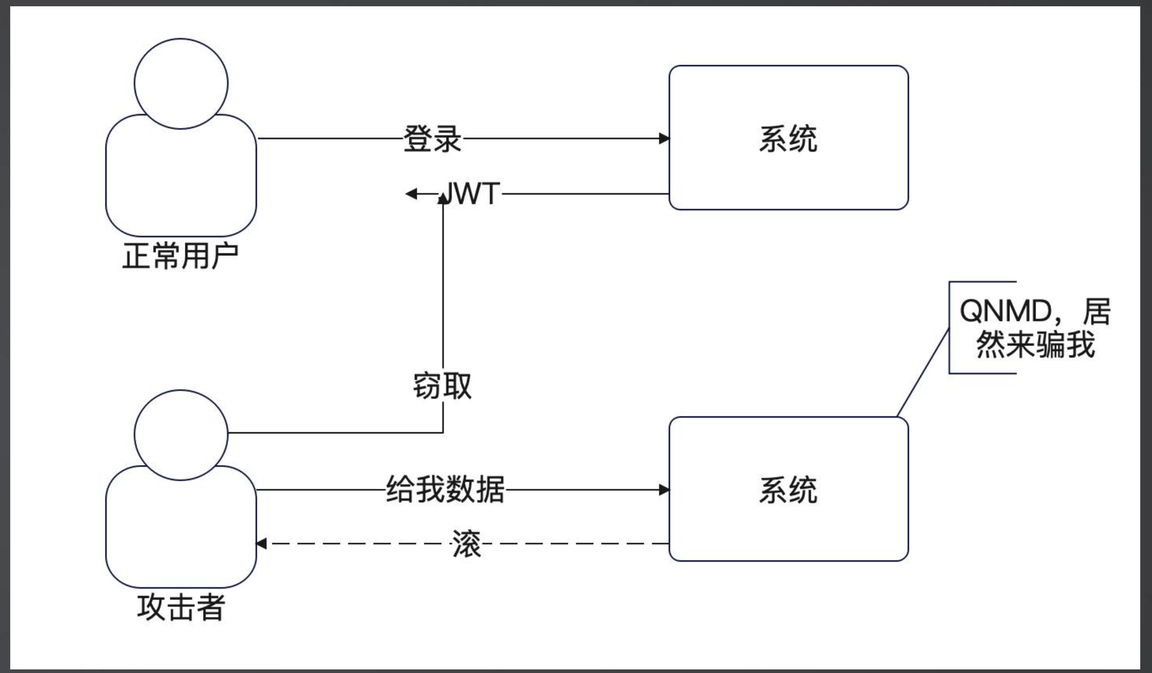


怎么解决呢？

所以我们就要想，在用户登录校验过程中，我得进一步判断，用这个 JWT/ssid 的人是不是原本登录的那个人。

目前做得好的都是使用二次验证，也就是给你发邮件、发短信等。

但是也有一些比较初步但也好用的手段，那就是用登录的辅助信息来判断。



登录的其他信息

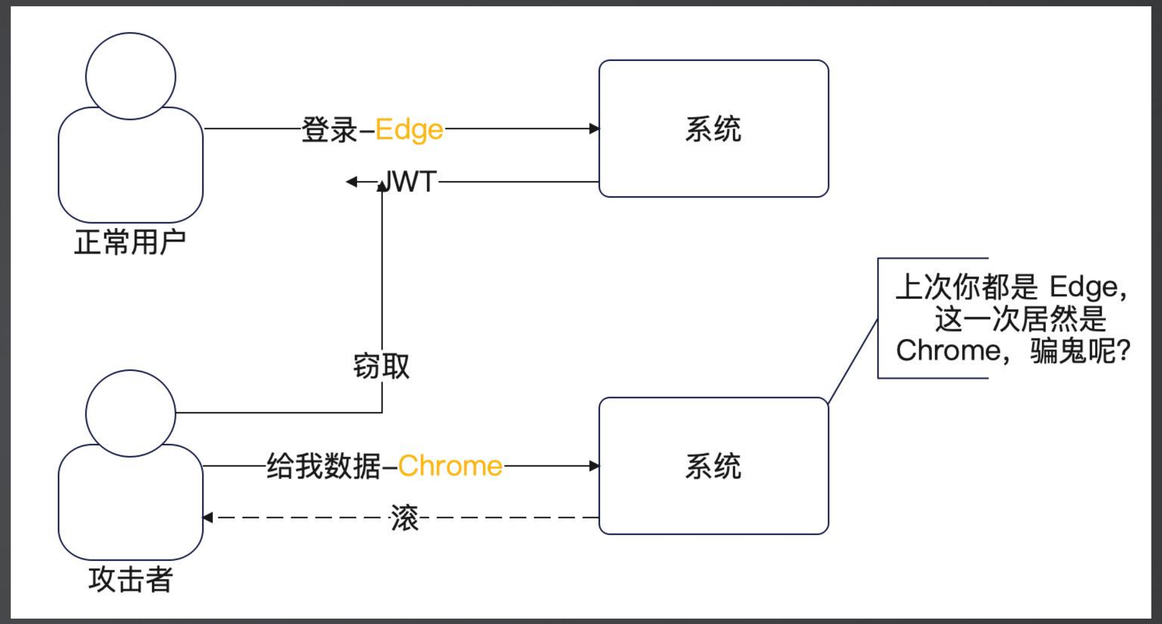
在你登录的时候，记录一下你当时登录的一些额外信息。

比如说：

- 你使用的浏览器，对应到 HTTP 的 User-Agent 头部。
- 你的硬件信息——手机 APP 比较多见。

问题：能不能用 IP？

在登录校验的时候，比较一下你当次请求的这些辅助信息和上一次的信息，不一样就认为有风险。



利用 User-Agent 增强安全性

右图则是在登录校验的时候比较了 User-Agent 这个参数。

为此你需要改造两个地方：

- Login 接口，在 JWT token 里面带上 User-Agent 信息。
- JWT 登录校验中间件，在里面比较 User-Agent。

```
token := jwt.NewWithClaims(jwt.SigningMethodHS256, UserClaims{
    Id:        u.Id,
    UserAgent: ctx.GetHeader( key: "User-Agent"),
    RegisteredClaims: jwt.RegisteredClaims{
        // 演示目的设置为一分钟过期
        ExpiresAt: jwt.NewNumericDate(time.Now().Add(time.Minute)),
    },
})
```

```
if ctx.GetHeader( key: "User-Agent") != uc.UserAgent {
    // 换了一个 User-Agent，可能是攻击者
    ctx.AbortWithStatus(http.StatusUnauthorized)
    return
}
```

升职加薪

保护公司前端接口

检查公司的前端接口，而后加入限流功能。

- 可以考虑整个集群限流
- 针对核心业务的接口限流
- 针对不需要登录就可以访问的接口限流
- 为限流添加对应的监控和告警

Gin 插件库添加限流插件

为 Gin 插件库添加限流插件，包含：

- 单机限流
  - 令牌桶算法
  - 漏桶算法
  - 滑动窗口算法
  - 固定窗口算法
- 基于 Redis 限流
- 基于 Redis 的 IP 限流

面试要点

面试要点

- 刷新 Session 过期时间的几种可行的办法，你要能够深入分析不同做法的优缺点。注意，面试不是实践，不能说我记住最佳的就行，而是你要尽可能”水“时间，展示自己博闻广识。
- 增强登录的安全性：
  - 怎么保护 Session id？主要还是要启用 HTTPS 协议，把 Cookie 的 Secure 和 HttpOnly 都设置为 true。
  - 怎么做到在 Session id 或者 JWT token 泄露之后保护住用户？要在登录的时候记录一下登录的附加信息，例如 User-Agent，在登录校验的时候一并校验。
- 如何保护 Web 服务？ 针对 IP 限流、整个集群限流。后续你还会接触到更加多的保护措施。

注意：这一部分的内容在你们面试初级工程师的时候，是属于比较有技术含量的点了，你在面试前一定要试着按照自己的说话习惯，整理好对应的答案，写出来多读几遍。

THANKS