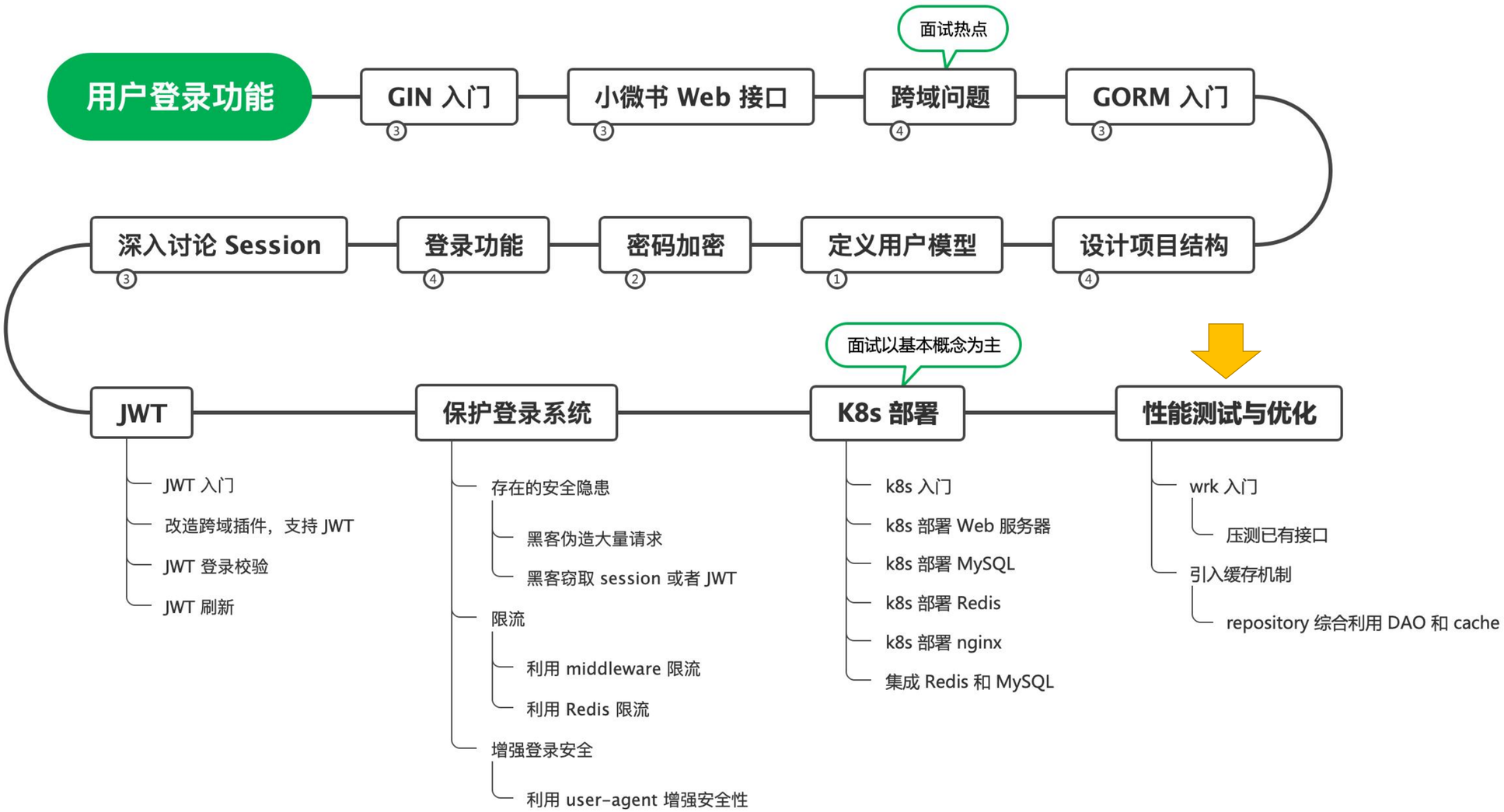
极客时间 训练营

接口抽象技巧与短信服务实现
之
优化登录性能

大明


用户登录功能
GIN 入门 3
小微书 Web 接口 3
面试热点
跨域问题 4
GORM 入门 3
设计项目结构 4
定义用户模型 1
密码加密 2
登录功能 4
深入讨论 Session 3
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
面试以基本概念为主
K8s 部署
k8s 入门
k8s 部署 Web 服务器
k8s 部署 MySQL
k8s 部署 Redis
k8s 部署 nginx
集成 Redis 和 MySQL
性能测试与优化
wrk 入门
压测已有接口
引入缓存机制
repository 综合利用 DAO 和 cache

利用 wrk 来压测接口

我们可以考虑使用 wrk 先来压测接口。

我们压测三个接口：

- 注册：写为主的接口
- 登录：读为主的接口
- Profile：读为主的接口

wrk 这个工具前期不要求你掌握，只需要把我这里运行的命令贴到命令行跑起来就可以。

wrk 安装

- 可以用 apt install wrk 或者 Mac 上 brew install wrk。
- 源码安装：直接源码下载 git clone https://github.com/wg/wrk.git

而后进去这个 wrk 目录下，执行 make 命令编译。

编译之后你会得到一个 wrk 可执行文件，将它加入你的环境变量。

压测前准备

压测注册接口，关键的是要发起 post 请求，然后传入不同的邮箱来模拟注册过程。

我已经提前准备好了脚本，你按照步骤运行就可以。

首先：

- 启用 JWT 来测试——因为比较好测试。如果你对代码理解比较深刻，也可以使用 Session。
- 修改 /users/login 对应的登录态保持时间，修改为 30 分钟，本质上是确保在你测试 profile 接口的时候，你拿到的 JWT token 没有过期。
- 去除 ratelimit 限制。

```
    //rCfg := config.Config.Redis
    //cmd := redis.NewClient(&redis.Options{
    //  Addr:     rCfg.Addr,
    //  Password: rCfg.Password,
    //  DB:       rCfg.DB,
    //})
    // 一分钟 100 次。
    //server.Use(ratelimit.NewBuilder(cmd, time.Minute, 100).Build())
```

```
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)),
        // 在压测的时候，要将过期时间设置更长一些
        ExpiresAt: jwt.NewNumericDate(time.Now().Add(time.Minute * 30)),
    },
})
```

压测注册接口

在项目根目录下执行：

wrk -t1 -d1s -c2 -s ./scripts/wrk/signup.lua
http://localhost:8080/users/signup

注意，你可以不断调整这些参数：

- -t：后面跟着的是线程数量。
- -d：后面跟着的是持续时间，比如说 1s 是一秒，也可以是 1m，是一分钟。
- -c：后面跟着的是并发数。
- -s：后面跟着的是测试的脚本。

最终能跑多少，和你机器有关。

```
Running 1s test @ http://localhost:8080/users/signup
  1 threads and 2 connections
  Thread Stats   Avg      Stdev     Max   +/- Stdev
    Latency    76.72ms    9.75ms 108.95ms   92.31%
    Req/Sec    26.33     10.26    40.00     66.67%
  26 requests in 1.01s, 3.45KB read
Requests/sec:     25.83
Transfer/sec:      3.43KB
```

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

换用不同的加密算法，试试性能。

压测登录接口

在项目根目录下执行：

wrk -t1 -d1s -c2 -s ./scripts/wrk/login.lua http://localhost:8080/users/login

因为登录接口也需要比较密码，所以你同样可以考虑换加密算法。

```
Running 1s test @ http://localhost:8080/users/login
  1 threads and 2 connections
  Thread Stats   Avg      Stdev     Max   +/- Stdev
    Latency    70.22ms    9.73ms 103.90ms   92.86%
    Req/Sec    28.67     10.52    40.00     66.67%
  28 requests in 1.01s, 7.63KB read
Requests/sec:     27.81
Transfer/sec:      7.58KB
```

```
wrk.method="POST"
wrk.headers["Content-Type"] = "application/json"
-- 这个要改为你的注册的数据
wrk.body='{"email":"12347@qq.com", "password": "hello#world123"}'
```

记得修改这里为你用的账号。

压测 Profile 接口

在项目根目录下执行：

wrk -t1 -d1s -c2 -s ./scripts/wrk/profile.lua http://localhost:8080/users/profile

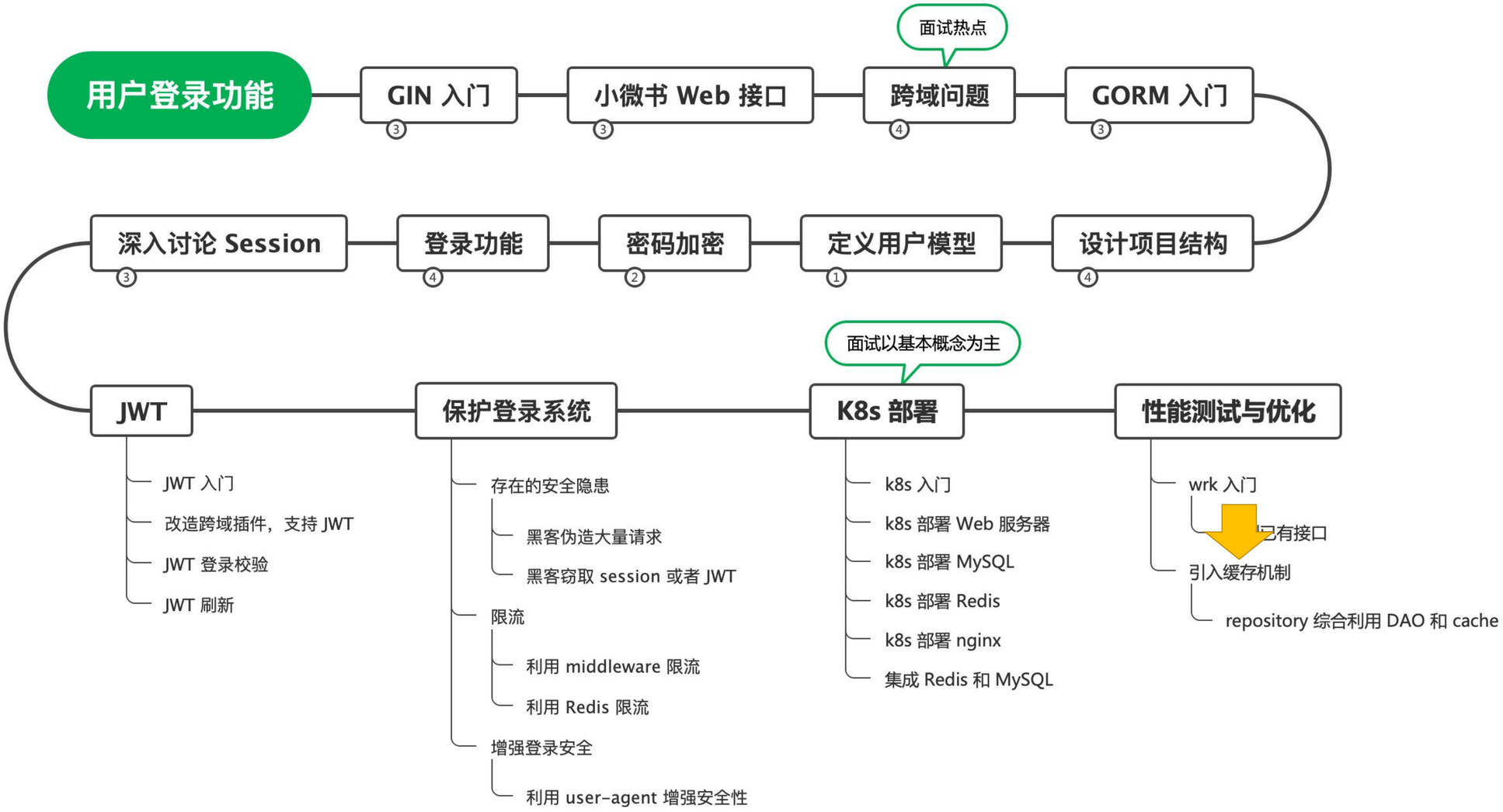
你要修改 User-Agent 和 对应的 Authorization。

扩展练习（可选）

- 切换不同的加密算法，测试注册和登录接口。
- 在我准备的这些脚本的基础上，你在数据库中插入 100W 条用户数据，然后再测试登录接口。
- 在数据库中插入 1000W 用户数据，然后再测试登录接口。

如果你做了这个测试，可以在群里分享一下你准备数据的脚本。

性能优化


用户登录功能
GIN 入门 3
小微书 Web 接口 3
面试热点
跨域问题 4
GORM 入门 3
设计项目结构 4
定义用户模型 1
密码加密 2
登录功能 4
深入讨论 Session 3
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
面试以基本概念为主
K8s 部署
k8s 入门
k8s 部署 Web 服务器
k8s 部署 MySQL
k8s 部署 Redis
k8s 部署 nginx
集成 Redis 和 MySQL
性能测试与优化
wrk 入门
已有接口
引入缓存机制
repository 综合利用 DAO 和 cache

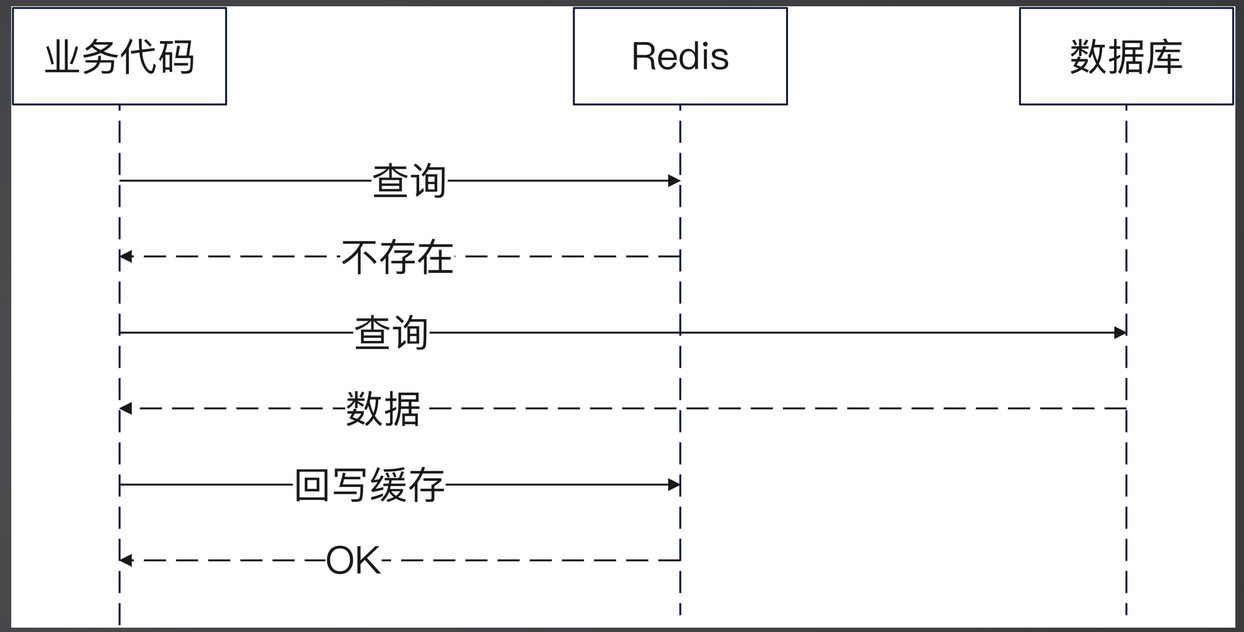
性能瓶颈

在前面的代码里面，基本上性能瓶颈是出在两个地方：

- 加密算法，耗费 CPU，会令 CPU 成为瓶颈。
- 数据库查询。

所以我们可以考虑引入 Redis 来优化性能。

用户会先从 Redis 里面查询，而后在缓存未命中的情况下，就会直接从数据库中查询。



引入缓存

但是，我们并不会直接 Redis，而是要引入一个缓存，来避免上层业务直接操作 Redis。

同时我们也不是引入一个通用的 Cache，而是为业务编写专门的 Cache。

也就是 UserCache。

```go
type UserCache struct {  7 usages   new *
    cmd redis.Cmdable
    // 过期时间
    expiration time.Duration
}

func NewUserCache(cmd redis.Cmdable) *UserCache {  1 u
    return &UserCache{
        cmd:        cmd,
        expiration: time.Minute * 15,
    }
}
```

业务专属缓存抽象的作用

引入一个专门的 UserCache 是为了解决：

- 屏蔽过期时间设置问题。也就是说，使用这个 UserCache 的人不再关心过期时间的问题。
- 屏蔽 key 的结构。也就是调用者不用知道在缓存里面的这个 key 是怎么组成的。
- 屏蔽序列化与反序列协议。当结构体写入到 Redis 的时候，要决定如何序列化和反序列化。

```go
func NewUserCache(cmd redis.Cmdable) *UserCache {
    return &UserCache{
        cmd:        cmd,
        expiration: time.Minute * 15,
    }
}
```

```go
func (cache *UserCache) key(id int64) string {  2 usages
    return fmt.Sprintf( format: "user:info:%d", id)
}
```

```go
func (cache *UserCache) Set(ctx context.Context, u domain.User) error
    data, err := json.Marshal(u)
    if err != nil {
        return err
    }
    key := cache.key(u.Id)
    return cache.cmd.Set(ctx, key, data, cache.expiration).Err()
}
```

序列化与反序列化

序列化与反序列化是一对相反的操作。

- 序列化：将结构体转化为 []byte。在 Go 中，对应的方法名字一般叫做 Marshal。
- 反序列化：将 []byte 转化为结构体。在 Go 中，对应的方法名字一般叫做 Unmarshal。

例如右图中就是对应的 JSON 的序列化和反序列化操作。

```go
func (cache *UserCache) Get(ctx context.Context, id int64) (domain.
    key := cache.key(id)
    data, err := cache.cmd.Get(ctx, key).Result()
    if err != nil {
        return domain.User{}, err
    }
    // 反序列化回来
    var u domain.User
    err = json.Unmarshal([]byte(data), &u)
    return u, err
}

func (cache *UserCache) Set(ctx context.Context, u domain.User) err
    data, err := json.Marshal(u)
    if err != nil {
        return err
    }
    key := cache.key(u.Id)
    return cache.cmd.Set(ctx, key, data, cache.expiration).Err()
}
```

反序列化

序列化

集成 UserCache

缓存属于“如何存储数据”的范畴，所以要在 Repository 这一层集成进去，Service 对这个应该是没有感知的。

同样保持依赖注入的风格，将 DAO 和 Cache 实例都注入进去。

```
type UserRepository struct {  7 usages   Deng Ming *
    dao   *dao.UserDAO
    cache *cache.UserCache
}

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

代码详解

右边的代码有很显著的特点：

- 只要缓存返回了 error，就直接去数据库查询。
- 回写缓存的时候，忽略掉了错误。

这种写法也是有隐患的。

那就是万一 Redis 本身崩溃了，那么查询都会落到数据库上。

```
func (ur *UserRepository) FindById(ctx context.Context,
    id int64) (domain.User, error) {
    u, err := ur.cache.Get(ctx, id)
    // 注意这里的处理方式
    if err == nil {
        return u, err
    }
    ue, err := ur.dao.FindById(ctx, id)
    if err != nil {
        return domain.User{}, err
    }
    u = domain.User{
        Id:       ue.Id,
        Email:    ue.Email,
        Password: ue.Password,
    }
    // 忽略掉这里的错误
    _ = ur.cache.Set(ctx, u)
    return u, nil
}
```

检测数据不存在的写法

替换性写法是在 Redis 里面没有查询到数据的时候，才往数据库查询。

如果 Redis 本身出错了，不继续查询数据库。

代价就是 Redis 崩溃之后，业务也不可用，但是数据库保住了。

而别的使用数据库的业务，就不会受到影响。

个人推荐，优先用上一种写法。

```go
func (ur *UserRepository) FindByIdV1(ctx context.Context,
    id int64) (domain.User, error) {
    u, err := ur.cache.Get(ctx, id)
    switch err {
    case nil:
        return u, err
    case cache.ErrKeyNotExist:
        ue, err := ur.dao.FindById(ctx, id)
        if err != nil {
            return domain.User{}, err
        }
        u = domain.User{...}
        // 忽略掉这里的错误
        _ = ur.cache.Set(ctx, u)
        return u, nil
    default:
        return domain.User{}, err
    }
}
```

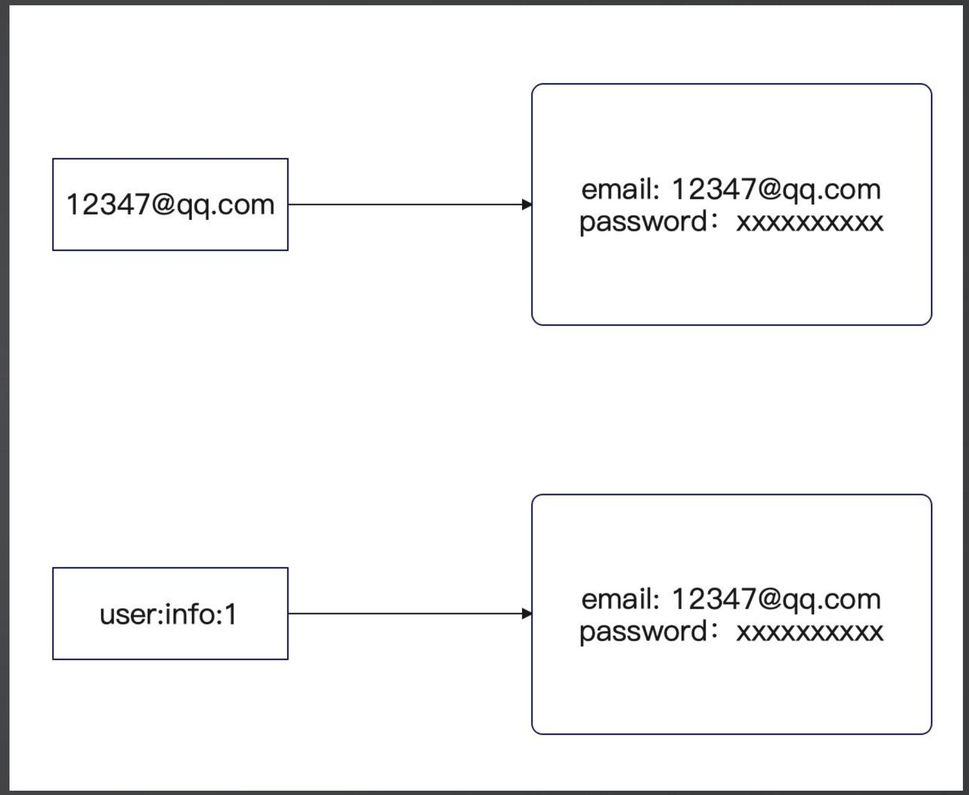
登录要不要利用 Redis 来优化性能？

在这里我们只是使用缓存优化了 /users/profile 的性能，
那么登录需不需要呢？

要不要再按照 email 映射到用户基本信息缓存一下？

答案是：可以，但是收益不大。

因为登录是一个非常低频的事情，正常的互联网网站都是
好几天才会让你登录一次，你缓存了也没用。



Redis 数据结构

在这里先简单过一下 Redis 支持的数据结构（非底层实现）。后续课程里遇到了再继续深入讨论。

Redis 数据结构主要有：

- string：你存储的 key 对应的值，是一个字符串。
- list：你存储的 key 对应的值，就是一个链表。
- set：你存储的 key 对应的值，是一个集合。
- sorted set：你存储的 key 对应的值，是一个有序集合。
- hash：你存储的 key 对应的值，是一个 hash 结构，也叫做字典结构、map 结构。

还有不常用的：bitmaps、JSON、streams、bitfields、time series。

支持一些什么操作，你可以通过 https://redis.io/commands/?group=set 来查看。

极客时间 训练营

升职加薪指南

测试并分析公司核心接口的性能

一般来说，如果你所在的公司没有任何的性能测试平台、工具，那么你要从 0 - 1 搭建起来，是一件很难的事情。

但是换一句话来说，也是一个很有挑战，很有技术含量的事情。

在最开始的时候，你只需要能够做到用 wrk 来测试接口进行。而后分析可能的性能瓶颈，一般出现在这些地方：

- 数据库查询。
- 调用别人的服务。

可以先输出一个性能分析报告，在公司内部做一个分享。后续你掌握了更多优化性能的技巧之后，再去输出性能优化方案。

面试要点

如何在测试中维护登录态

你有两条思路：

- 在测试初始化的时候去模拟登录，拿到对应的登录态的数据，例如 JWT token 或者 session id。
- 直接手动登录，然后复制对应的 token 或者 cookie 的值到脚本中，运行测试。

如果你的测试里面有权限控制，那么需要使用一个特殊的用户，这个用户有所有的权限。

Redis 面试题目

- 你用 Redis 解决过什么问题？
- 你知道 Redis 支持哪些数据结构吗？你用过哪些？用来解决什么问题？
- Redis 各个数据结构的底层实现？
- 当你更新数据的时候，你先更新数据库，还是先更新缓存？有没有一致性问题？
- 如何解决一致性问题？

THANKS