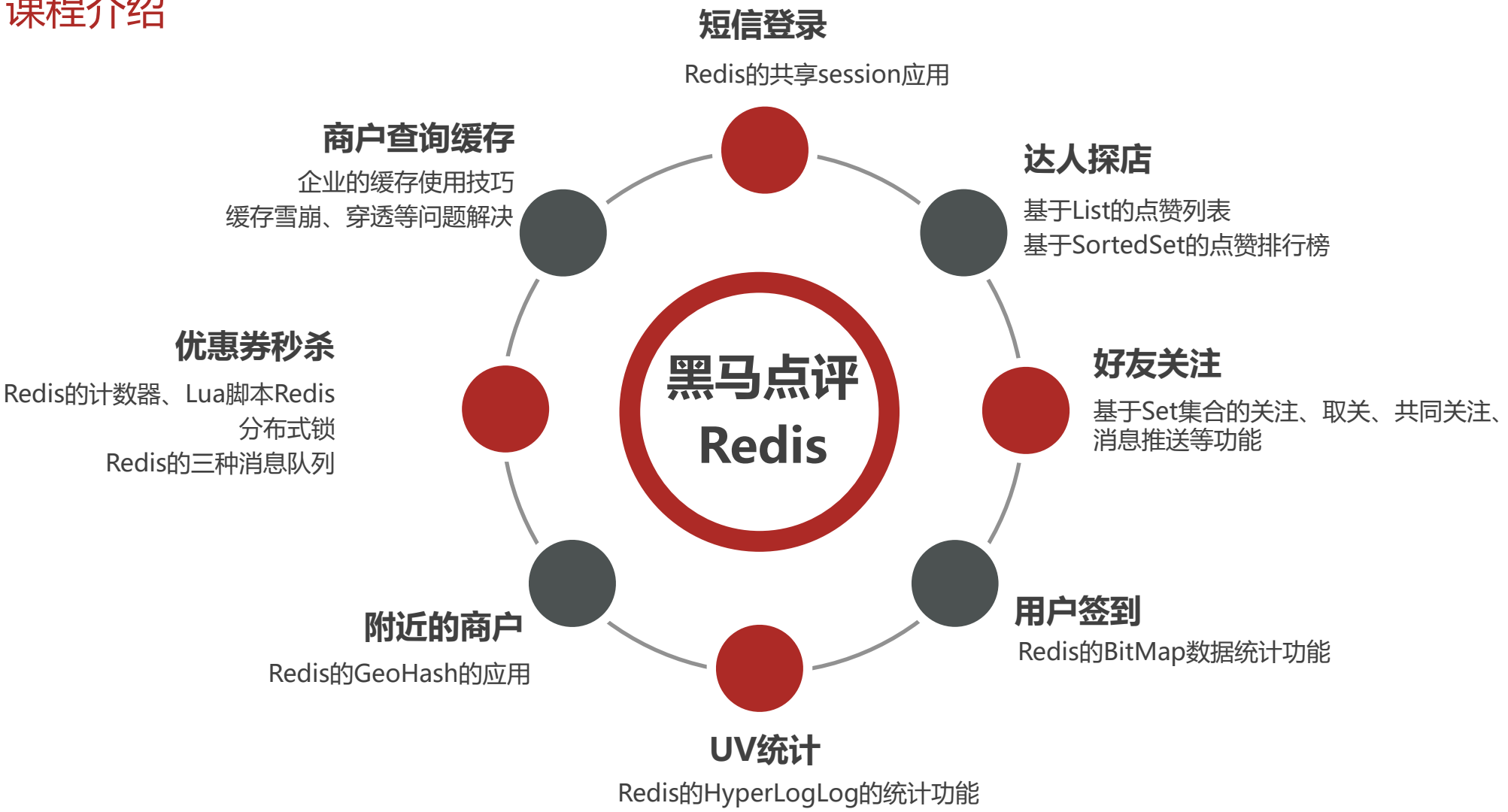
Redis企业实战

Redis的企业应用案例

黑马程序员
www.itheima.com

传智教育旗下
高端IT教育品牌

今日课程介绍


短信登录
Redis的共享session应用
商户查询缓存
企业的缓存使用技巧
缓存雪崩、穿透等问题解决
达人探店
基于List的点赞列表
基于SortedSet的点赞排行榜
优惠券秒杀
Redis的计数器、Lua脚本Redis
分布式锁
Redis的三种消息队列
黑马点评
Redis
好友关注
基于Set集合的关注、取关、共同关注、
消息推送等功能
附近的商户
Redis的GeoHash的应用
用户签到
Redis的BitMap数据统计功能
UV统计
Redis的HyperLogLog的统计功能

目录

Contents

01 短信登录

目录

Contents

导入黑马点评项目

首先，导入课前资料提供的SQL文件：

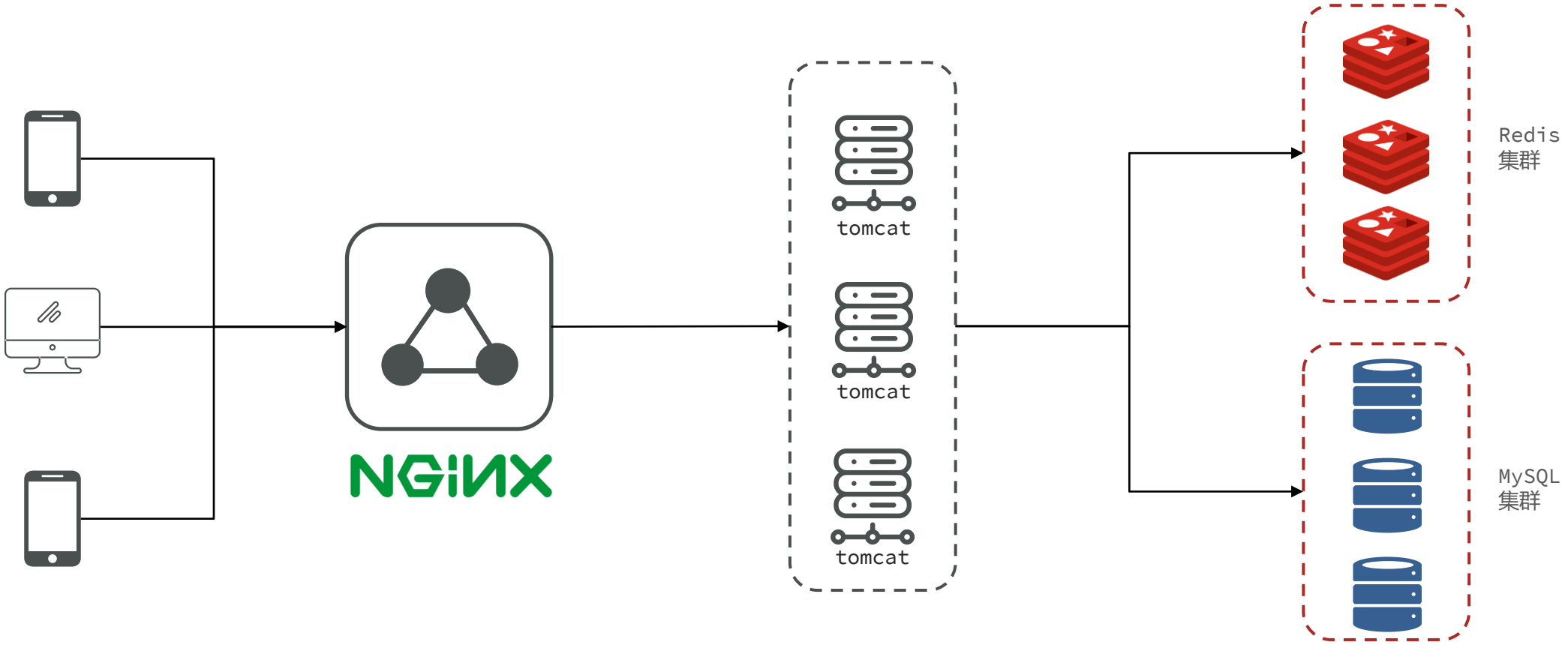
其中的表有：

- tb_user：用户表
- tb_user_info：用户详情表
- tb_shop：商户信息表
- tb_shop_type：商户类型表
- tb_blog：用户日记表（达人探店日记）
- tb_follow：用户关注表
- tb_voucher：优惠券表
- tb_voucher_order：优惠券的订单表

注意

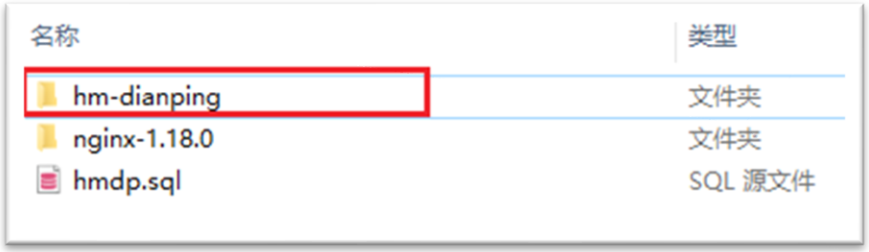
Mysql的版本采用5.7及以上版本

导入黑马点评项目



导入后端项目

在资料中提供了一个项目源码：



将其复制到你的idea工作空间，然后利用idea打开即可：

启动项目后，在浏览器访问：http://localhost:8081/shop-type/list ，如果可以看到数据则证明运行没有问题

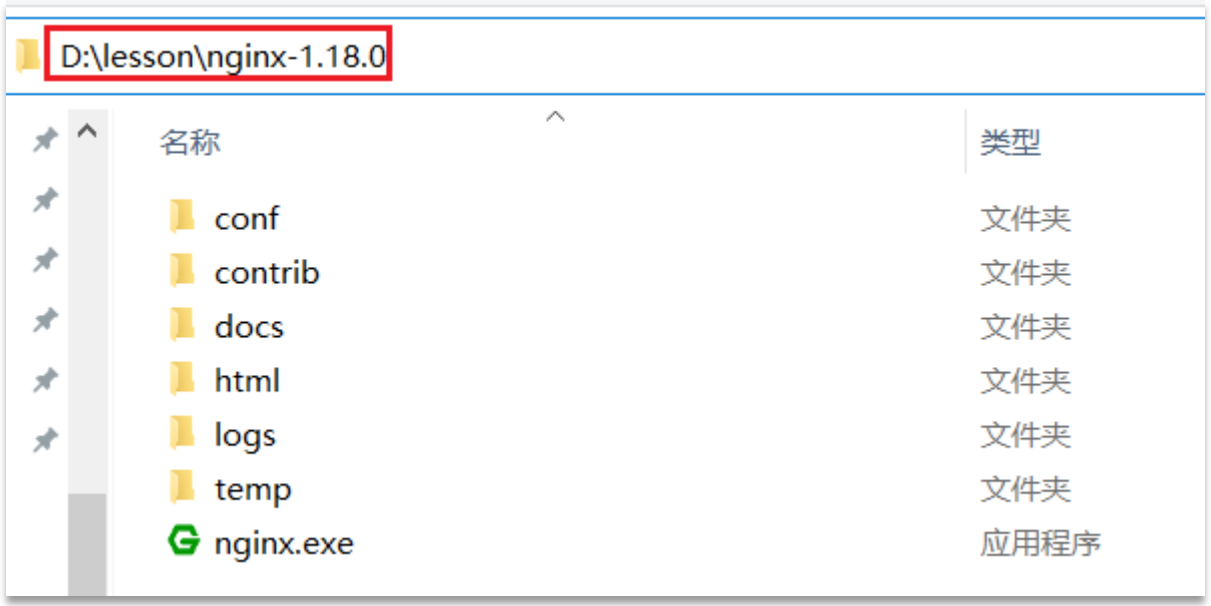
注意

不要忘了修改application.yaml文件中的mysql、redis地址信息

导入前端项目

在资料中提供了一个nginx文件夹：

将其复制到任意目录，要确保该目录不包含中文、特殊字符和空格，例如：

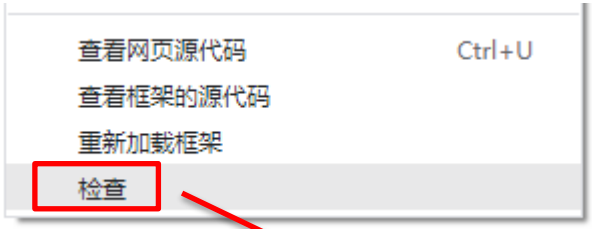


运行前端项目

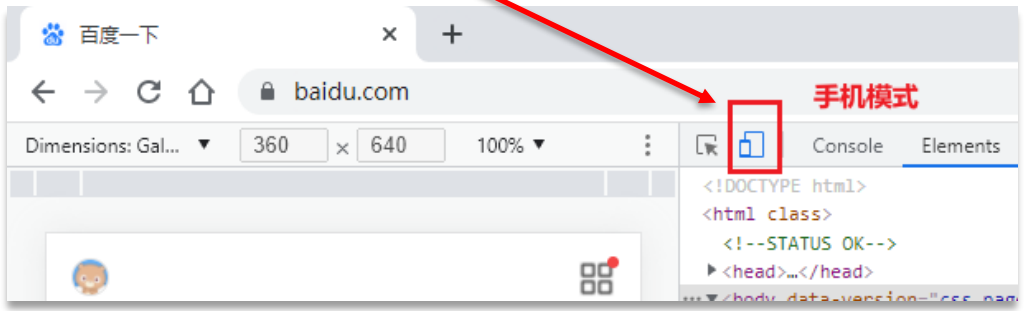
在nginx所在目录下打开一个CMD窗口，输入命令：

```
start nginx.exe
```

打开chrome浏览器，在空白页面点击鼠标右键，选择检查，即可打开开发者工具：



然后打开手机模式：

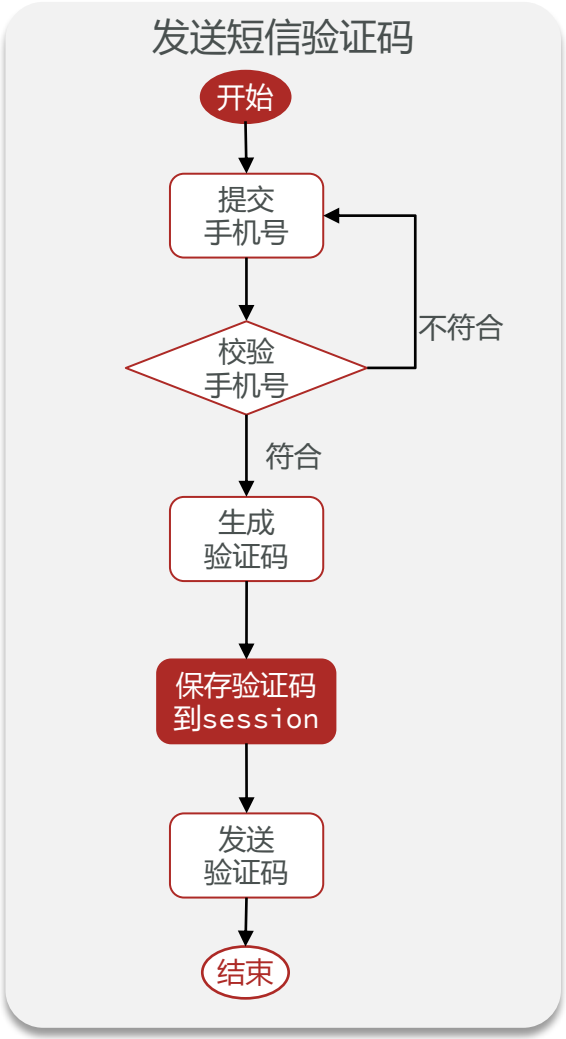


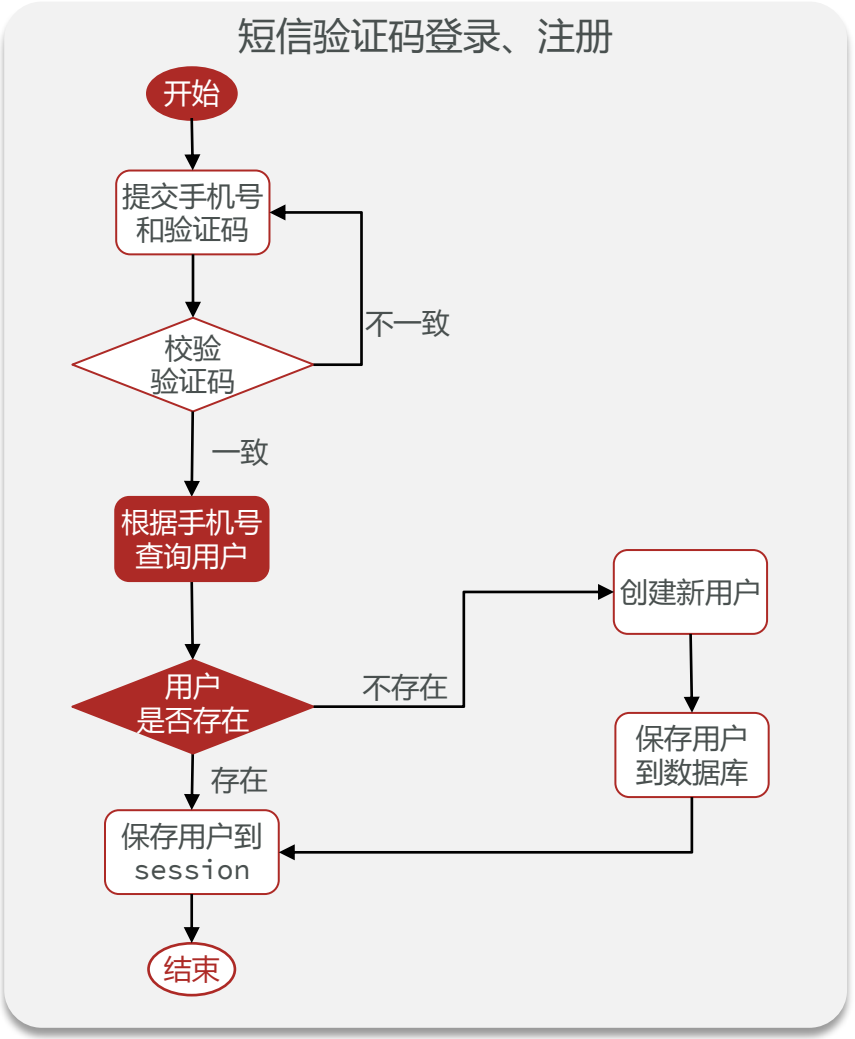
然后访问: http://127.0.0.1:8080 ，即可看到页面：

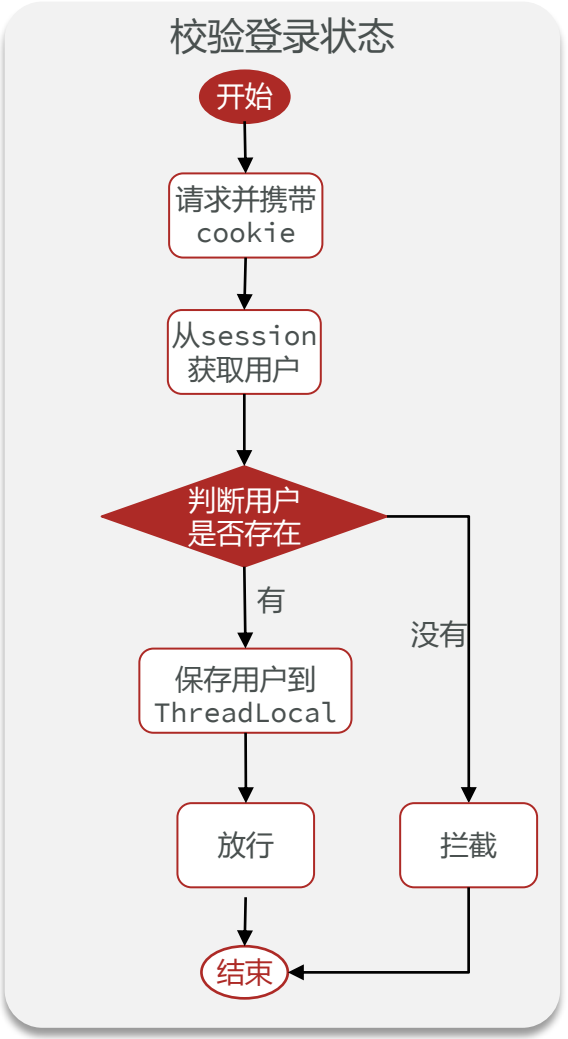
目录

Contents

基于Session实现登录



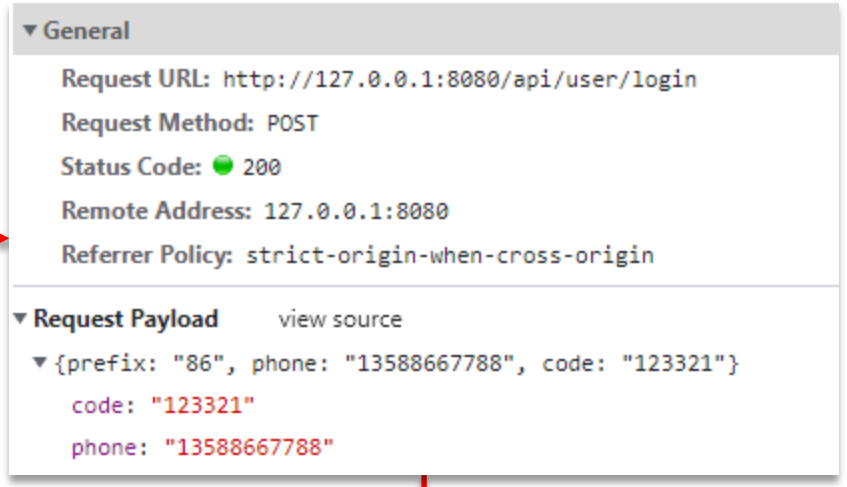




发送短信验证码

	说明
请求方式	POST
请求路径	/user/code
请求参数	phone,电话号码
返回值	无

短信验证码登录



	说明
请求方式	POST
请求路径	/user/login
请求参数	phone:电话号码；code:验证码
返回值	无

登录验证功能

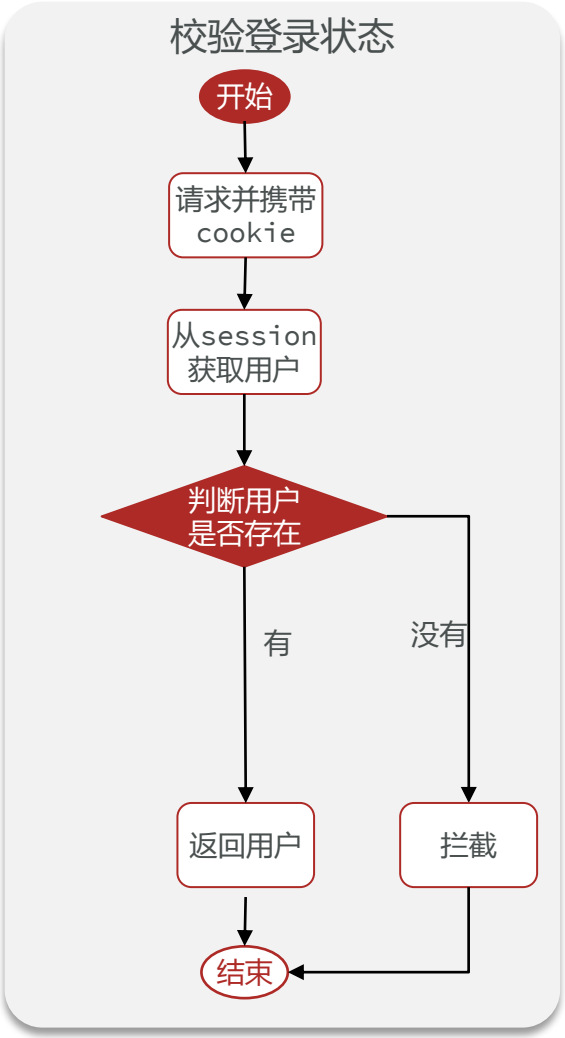
General

Request URL: http://localhost:8080/api/user/me
Request Method: GET
Status Code: 200
Remote Address: 127.0.0.1:8080
Referrer Policy: strict-origin-when-cross-origin

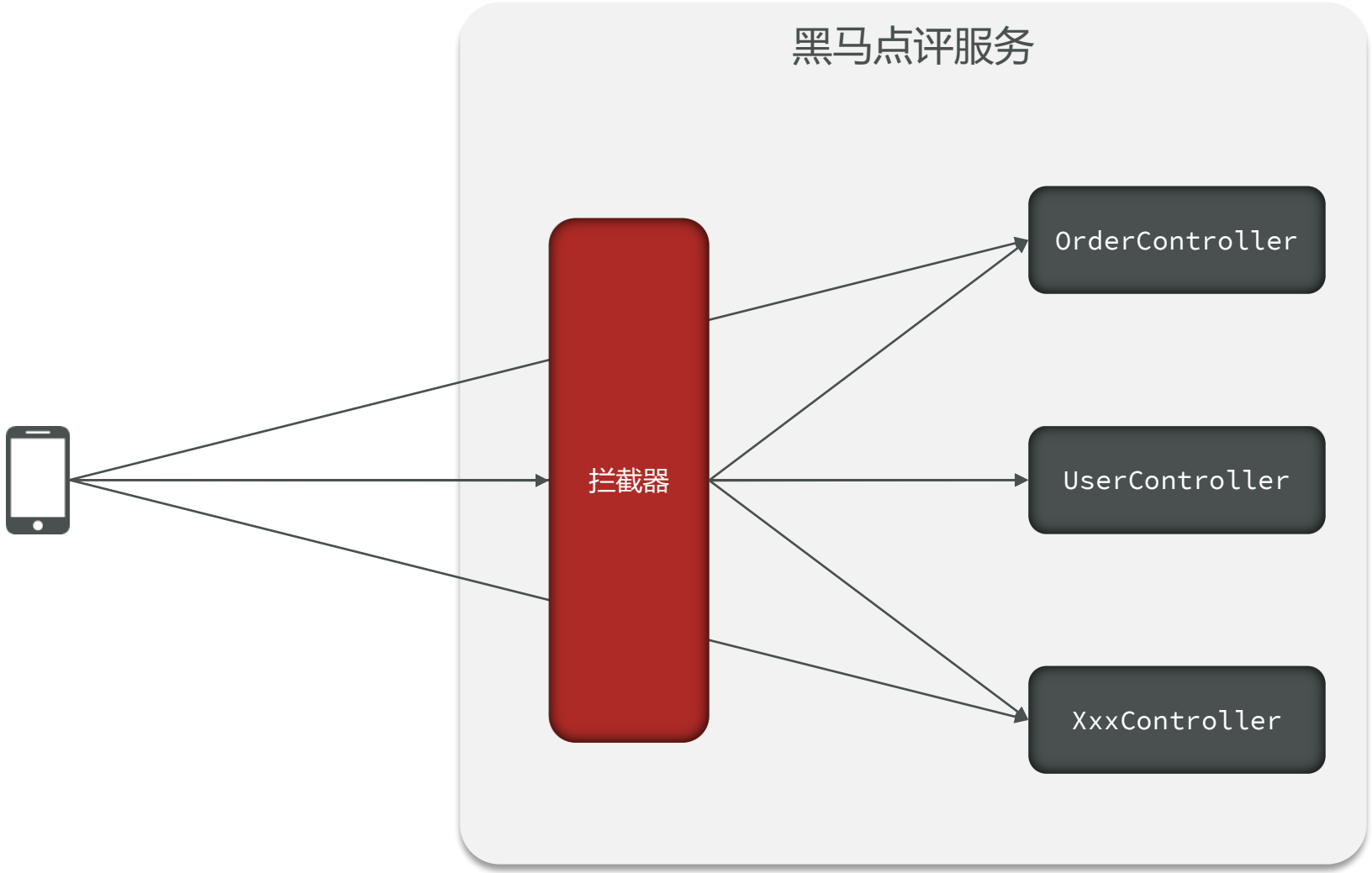
Response Headers (5)

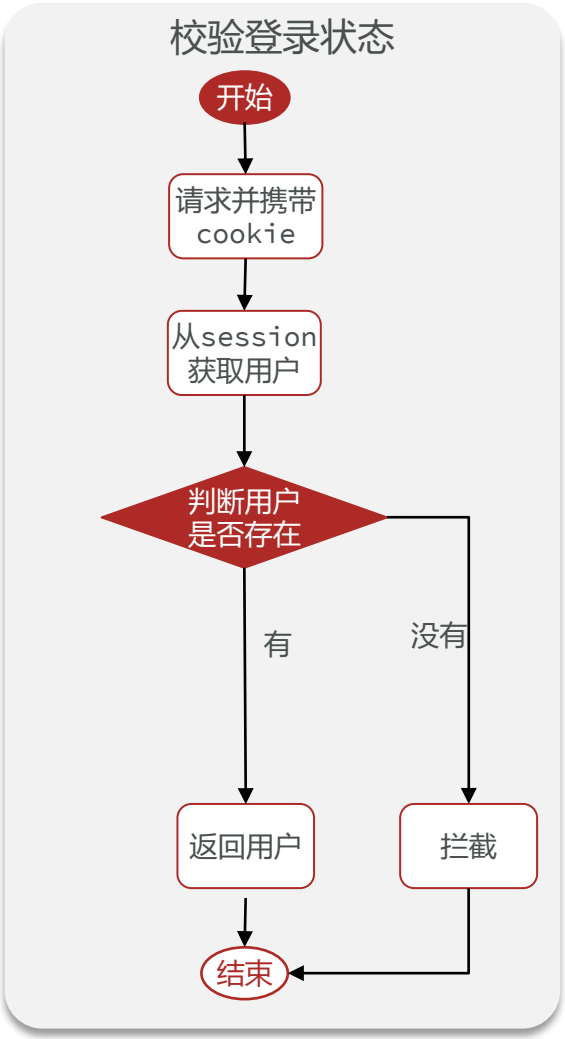
Request Headers View source

Accept: application/json, text/plain, */*
Accept-Encoding: gzip, deflate, br
Accept-Language: zh-CN,zh;q=0.9,en-US;q=0.8,en;q=0.7
Connection: keep-alive
Cookie: Idea-2e2d9f91=4ee107b6-b215-4cec-9d9e-b6804781dffd; JSESSIONID=40DF9A8FA7A0BDA229C5B41B2054092D
Host: localhost:8080
Referer: http://localhost:8080/info.html

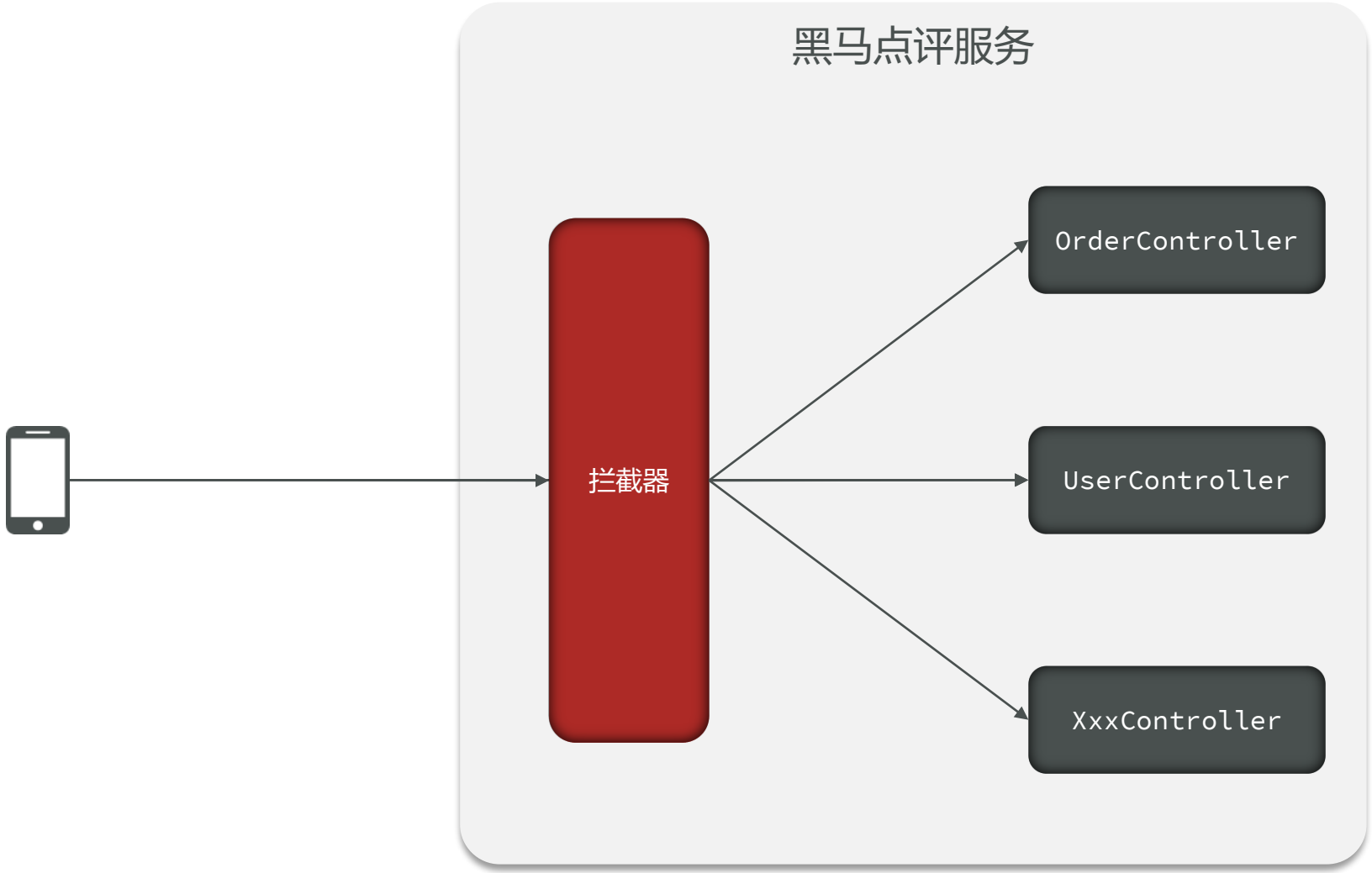


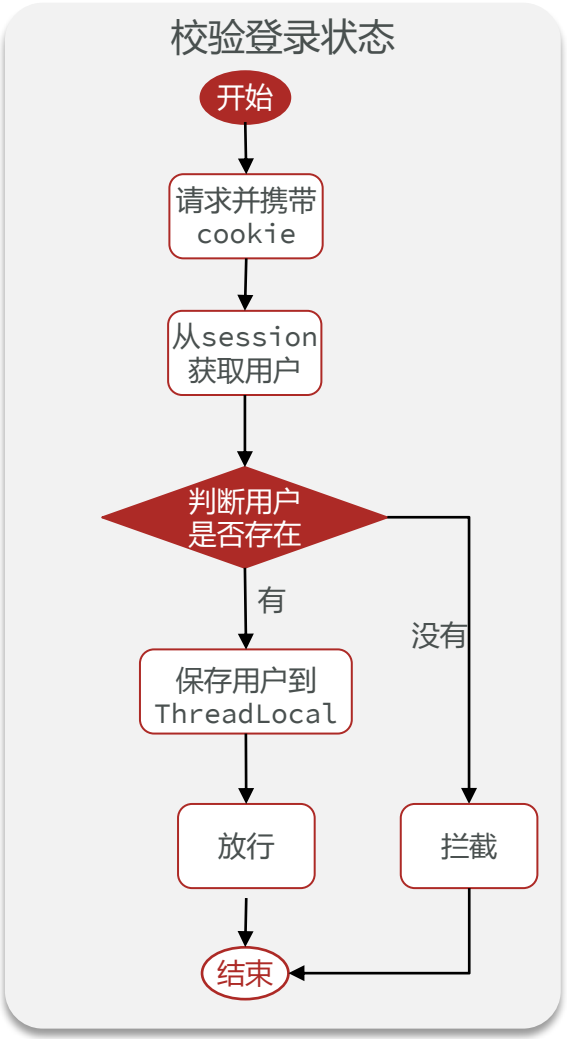
登录验证功能


黑马点评服务
拦截器
OrderController
UserController
XxxController


校验登录状态
开始
请求并携带cookie
从session获取用户
判断用户是否存在
有
没有
返回用户
拦截
结束

登录验证功能


黑马点评服务
拦截器
OrderController
UserController
XxxController


校验登录状态
开始
请求并携带cookie
从session获取用户
判断用户是否存在
有
没有
保存用户到ThreadLocal
放行
拦截
结束

目录

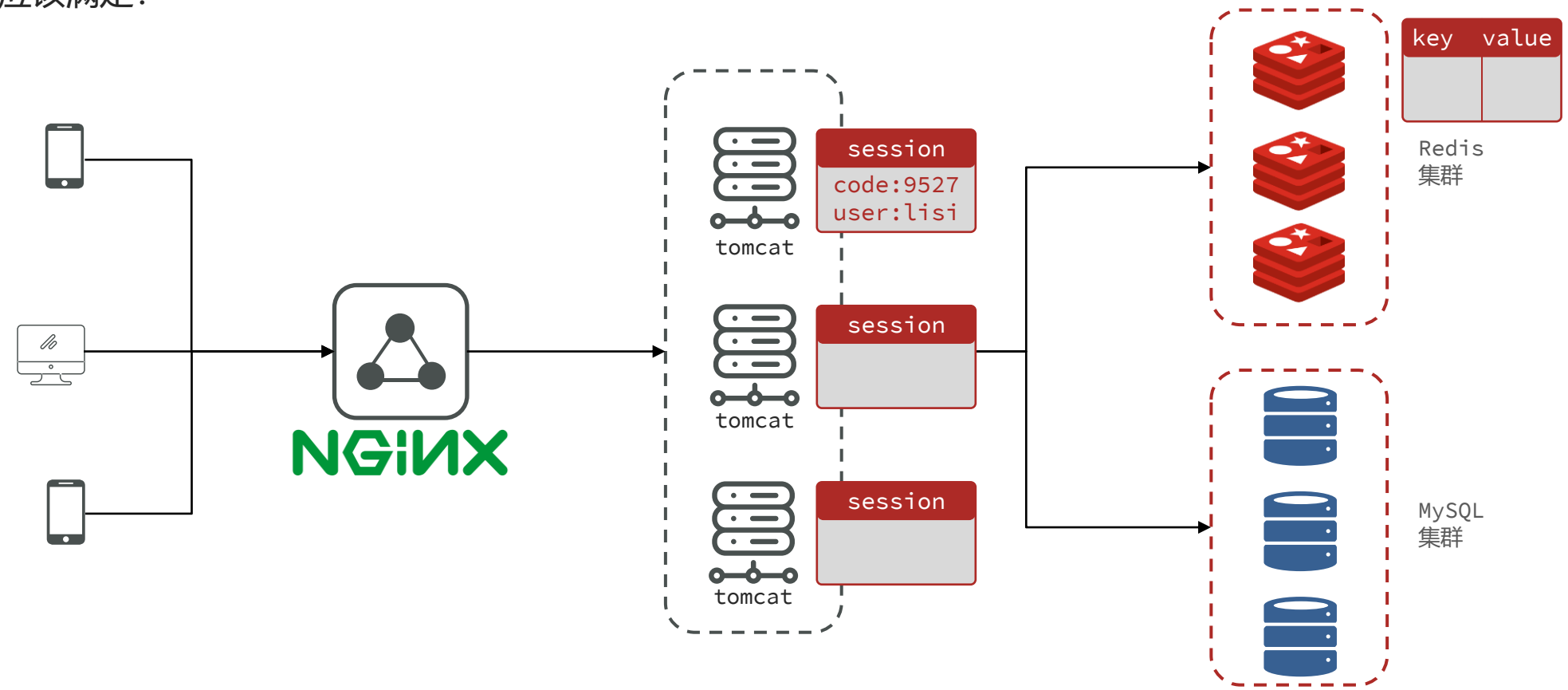
Contents

集群的session共享问题

session共享问题：多台Tomcat并不共享session存储空间，当请求切换到不同tomcat服务时导致数据丢失的问题。

session的替代方案应该满足：

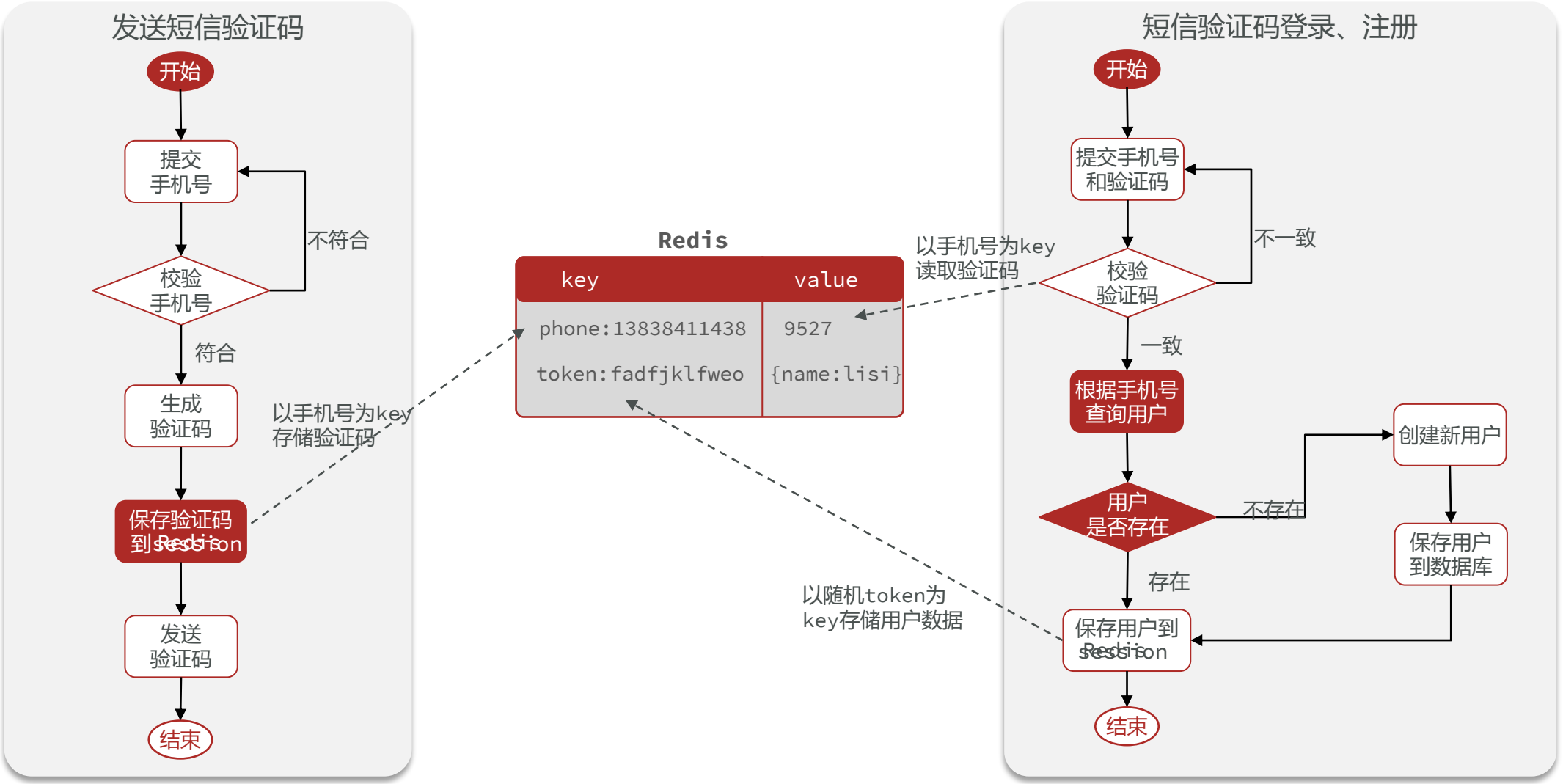
- 数据共享
- 内存存储
- key、value结构



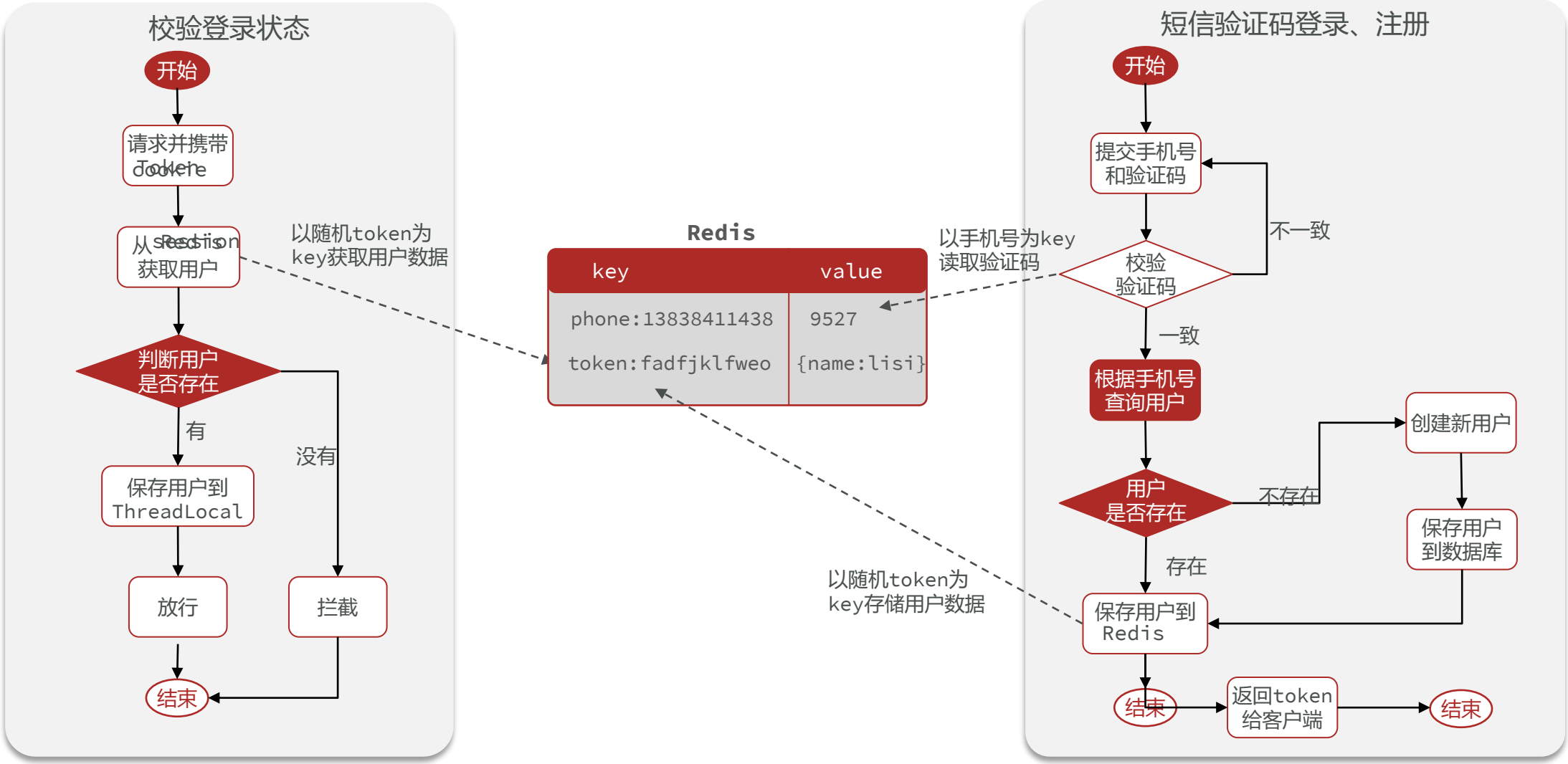
目录

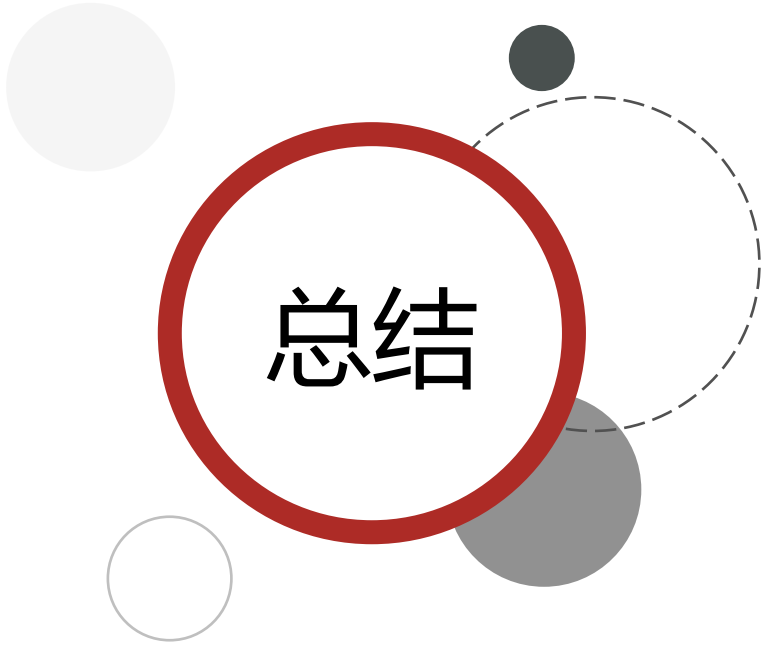
Contents

基于Redis实现共享session登录



基于Redis实现共享session登录

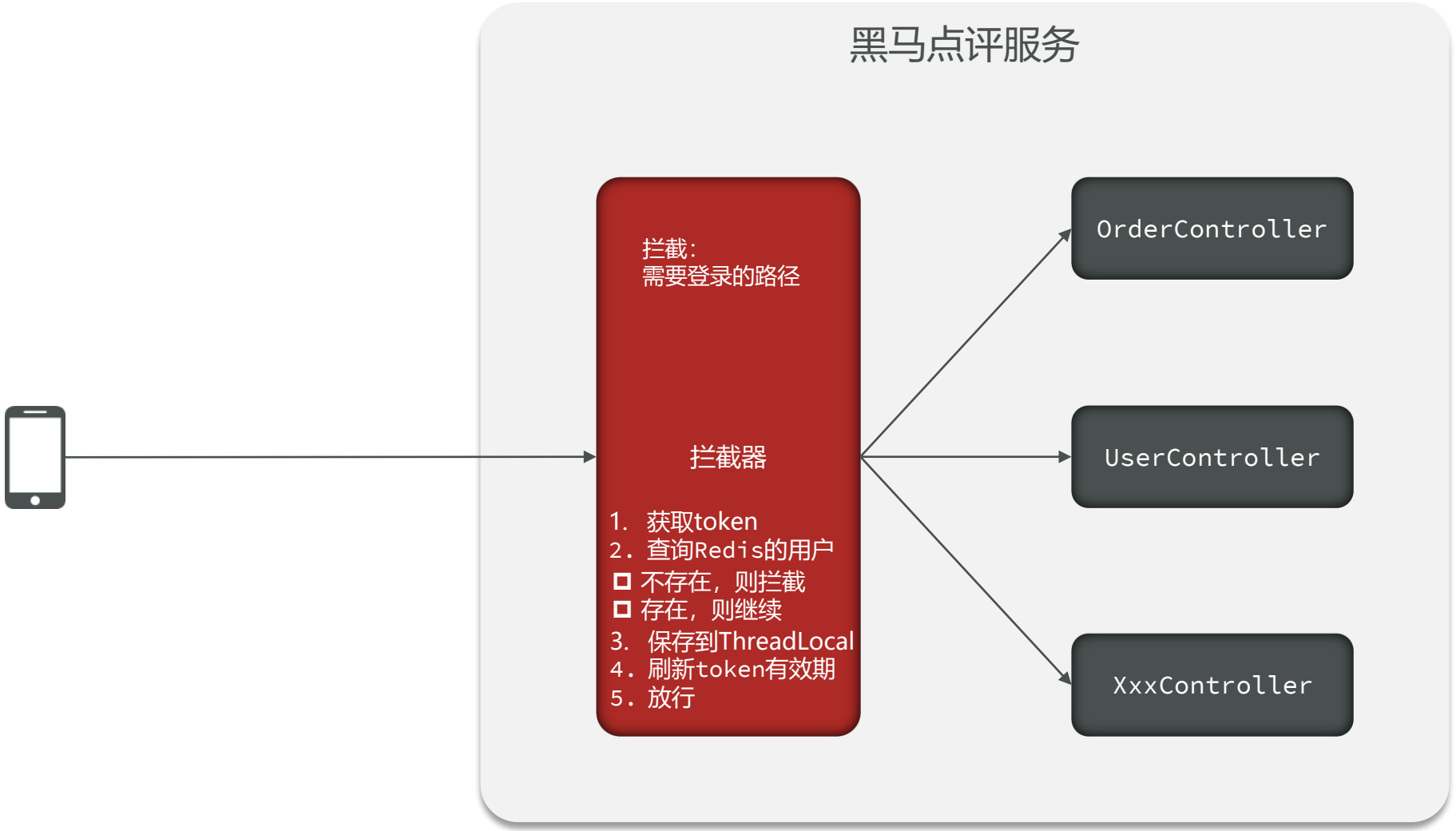




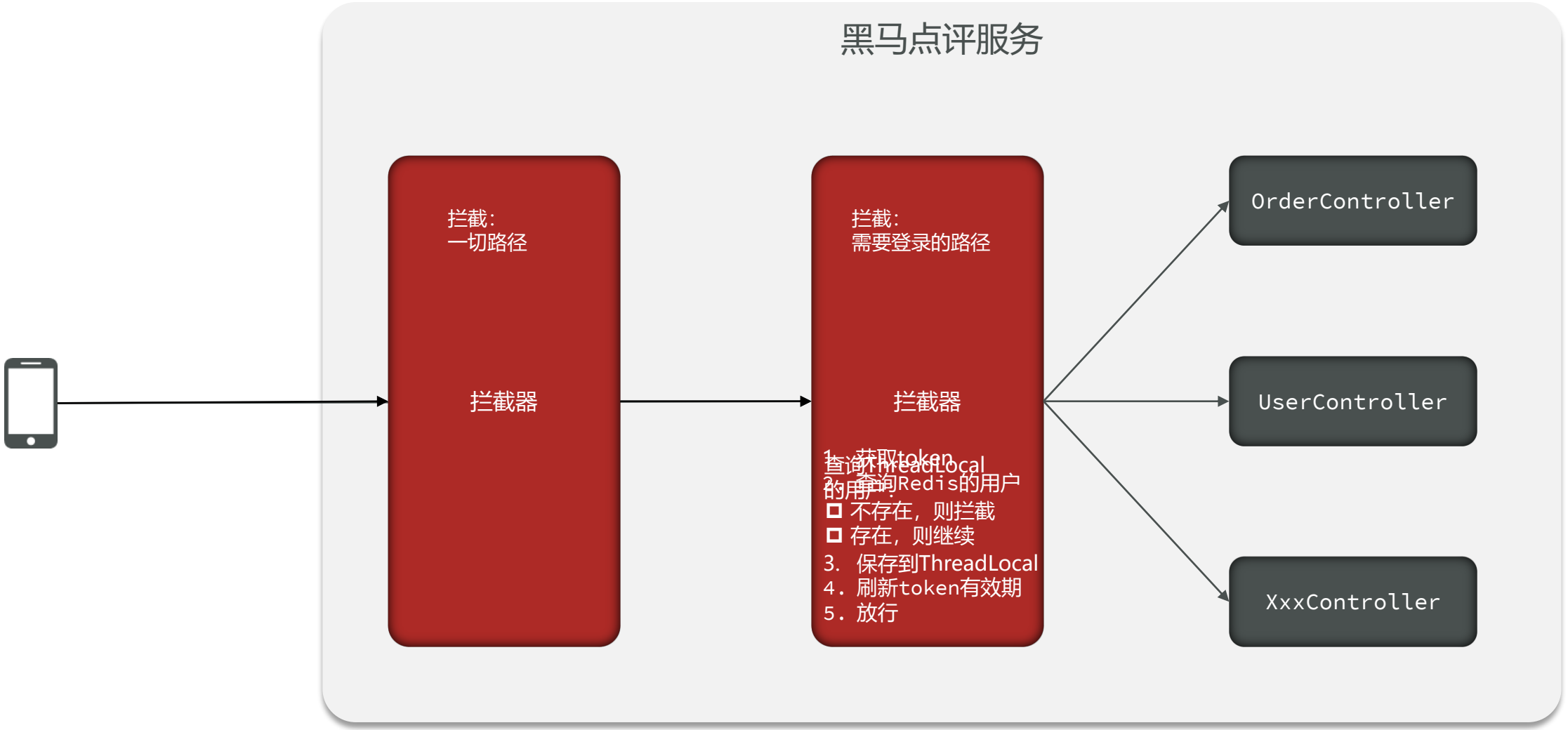
Redis代替session需要考虑的问题：

- 选择合适的数据结构
- 选择合适的key
- 选择合适的存储粒度

登录拦截器的优化


黑马点评服务
拦截：
需要登录的路径
拦截器
1. 获取token
2. 查询Redis的用户
不存在，则拦截
存在，则继续
3. 保存到ThreadLocal
4. 刷新token有效期
5. 放行
OrderController
UserController
XxxController

登录拦截器的优化


黑马点评服务
拦截：
一切路径
拦截器
拦截：
需要登录的路径
拦截器
1. 获取token
查询ThreadLocal
2. 查询Redis的用户
的用户：
不存在，则拦截
存在，则继续
3. 保存到ThreadLocal
4. 刷新token有效期
5. 放行
OrderController
UserController
XxxController

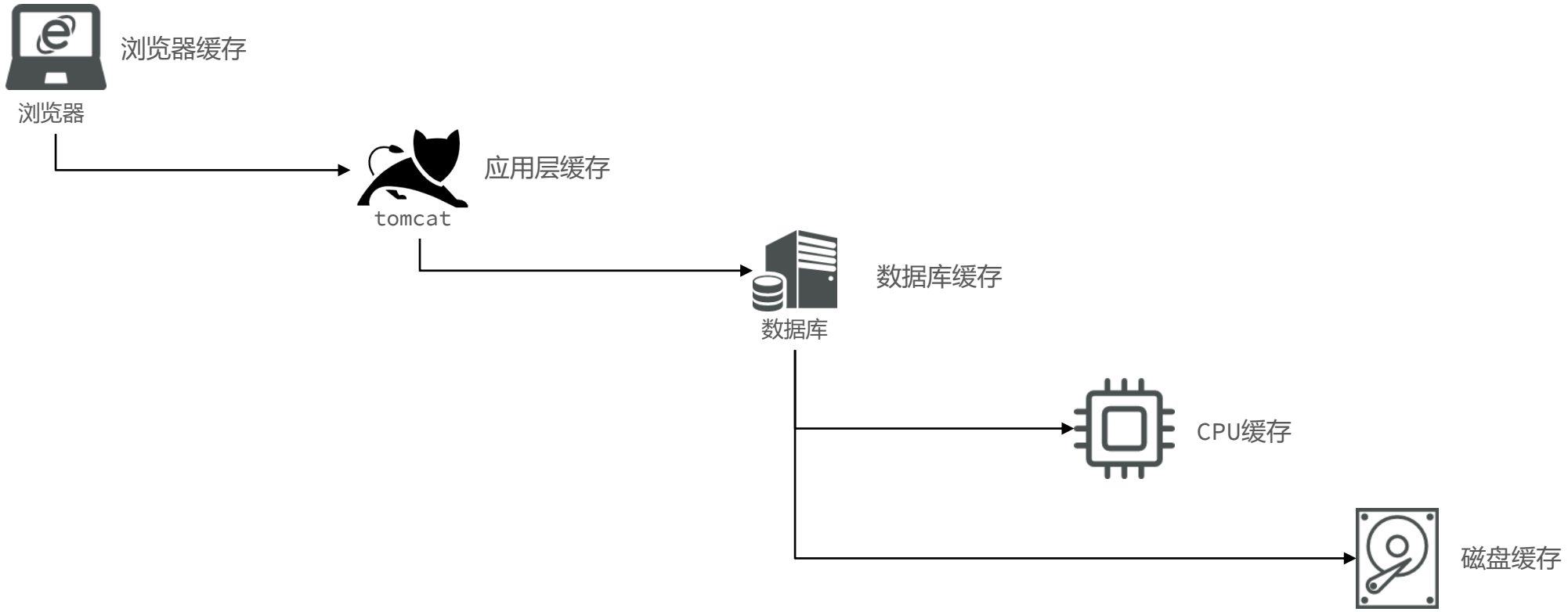
02 商户查询缓存

目录

Contents

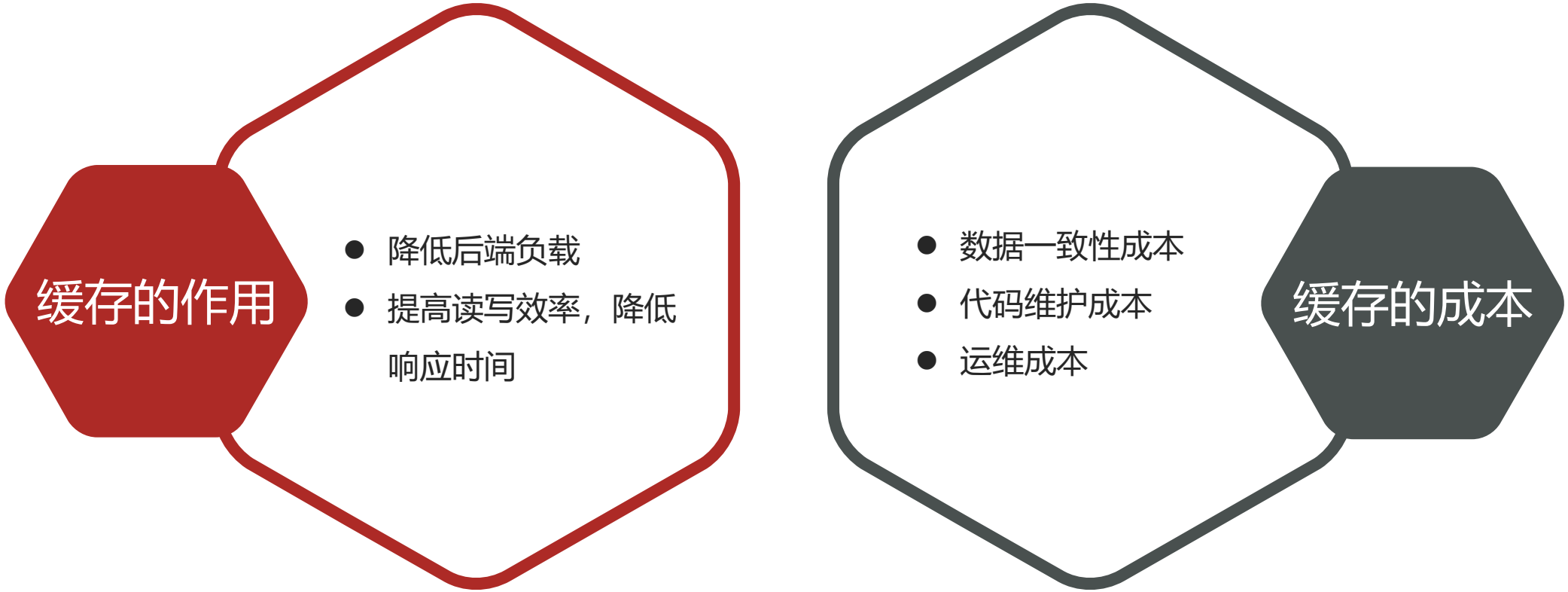
什么是缓存

缓存就是数据交换的缓冲区（称作Cache [kæʃ]），是存贮数据的临时地方，一般读写性能较高。



什么是缓存

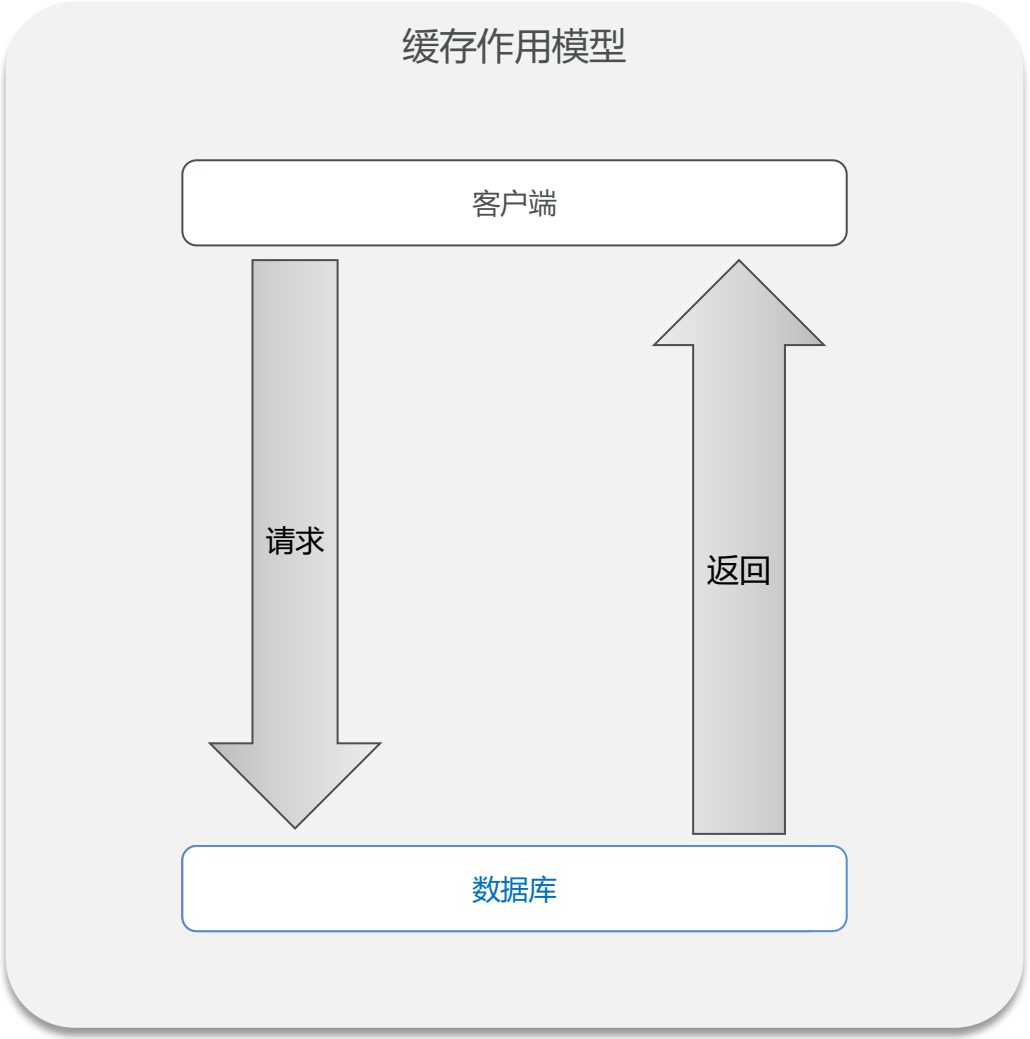
缓存就是数据交换的缓冲区（称作Cache [kæʃ]），是存贮数据的临时地方，一般读写性能较高。



目录

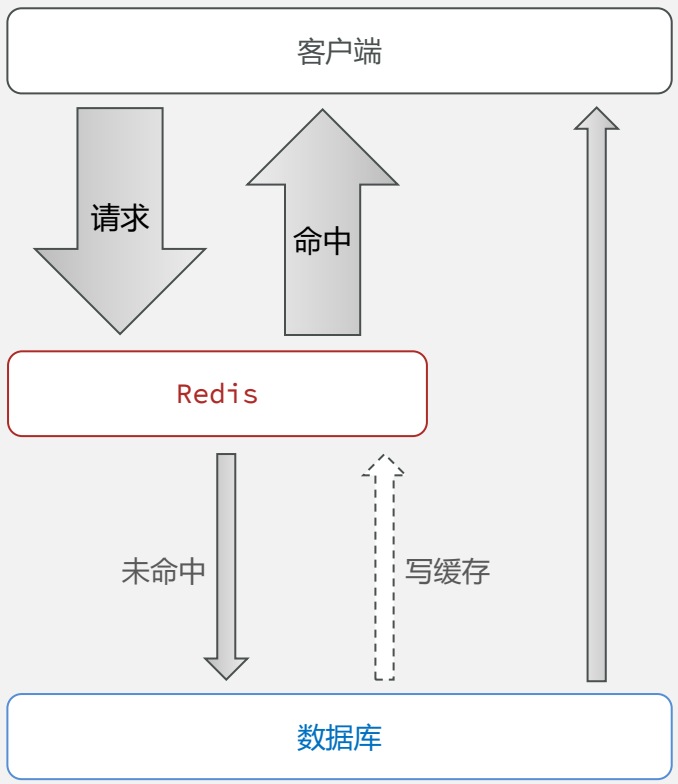
Contents

添加Redis缓存

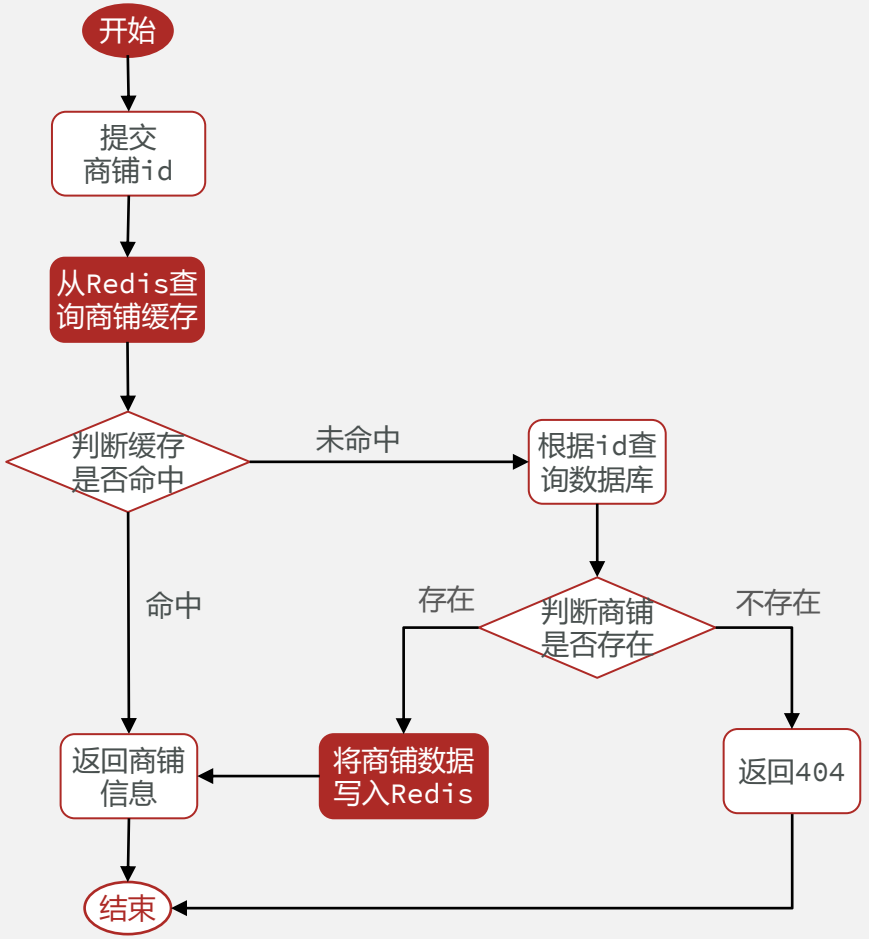


添加Redis缓存

缓存作用模型



根据id查询商铺缓存的流程



练习 给店铺类型查询业务添加缓存

店铺类型在首页和其它多个页面都会用到，如图：

需求：修改ShopTypeController中的queryTypeList方法，添加查询缓存

```
public class ShopTypeController {
    @Resource
    private IShopTypeService typeService;

    @GetMapping("list")
    public Result queryTypeList() {
        List<ShopType> typeList = typeService
                .query().orderByAsc( column: "sort").list();
        return Result.ok(typeList);
    }
}
```

目录

Contents

缓存更新策略

	内存淘汰	超时剔除	主动更新
说明			
一致性			
维护成本			

业务场景：

- 低一致性需求：使用内存淘汰机制。例如店铺类型的查询缓存
- 高一致性需求：主动更新，并以超时剔除作为兜底方案。例如店铺详情查询的缓存

主动更新策略

01

Cache Aside Pattern

由缓存的调用者，在更新数据库的同时更新缓存

02

Read/Write Through Pattern

缓存与数据库整合为一个服务，由服务来维护一致性。调用者调用该服务，无需关心缓存一致性问题。

03

Write Behind Caching Pattern

调用者只操作缓存，由其它线程异步的将缓存数据持久化到数据库，保证最终一致。

主动更新策略

01

Cache Aside Pattern

由缓存的调用者，在更新数据库的同时更新缓存

操作缓存和数据库时有三个问题需要考虑：

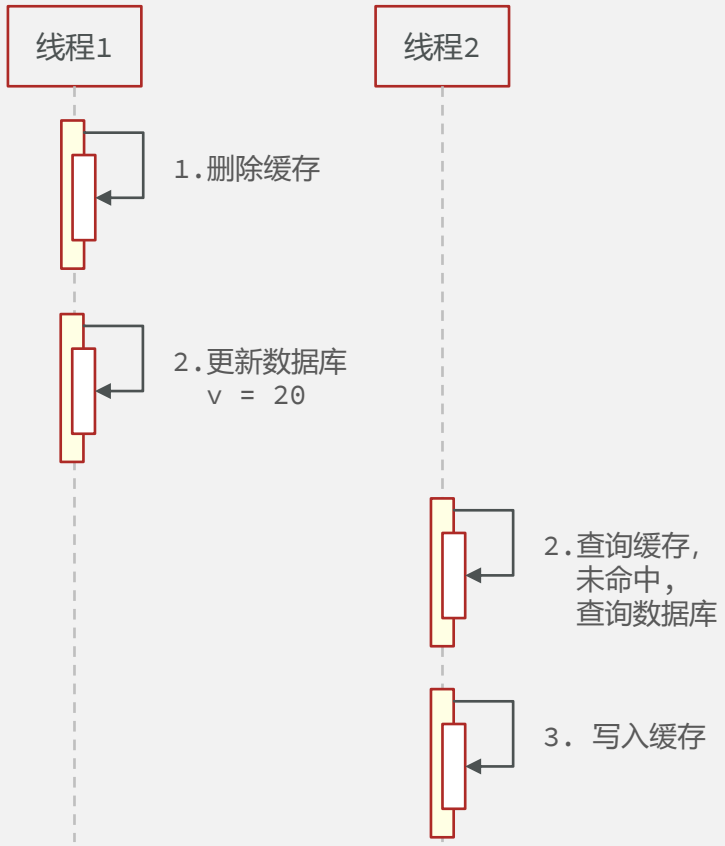
1. 删除缓存还是更新缓存？
   - 更新缓存：每次更新数据库都更新缓存，无效写操作较多 ✗
   - 删除缓存：更新数据库时让缓存失效，查询时再更新缓存 ✓
2. 如何保证缓存与数据库的操作的同时成功或失败？
   - 单体系统，将缓存与数据库操作放在一个事务
   - 分布式系统，利用TCC等分布式事务方案
3. 先操作缓存还是先操作数据库？
   - 先删除缓存，再操作数据库
   - 先操作数据库，再删除缓存

Cache Aside Pattern

缓存 20

数据库 20

先删除缓存，再操作数据库

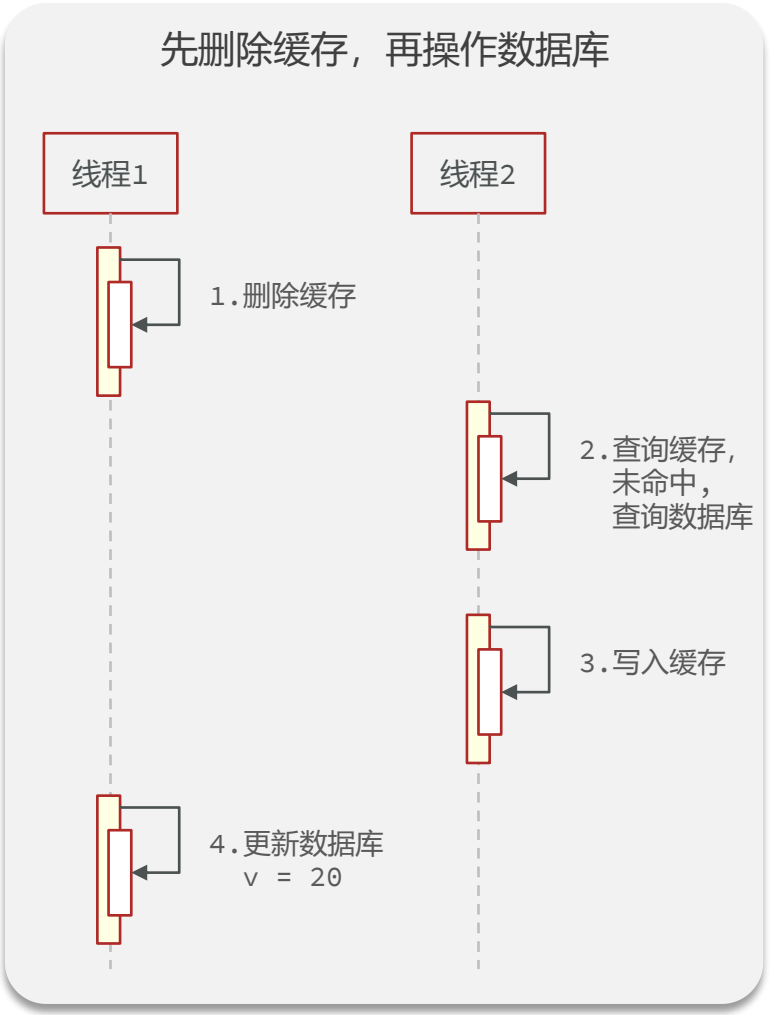


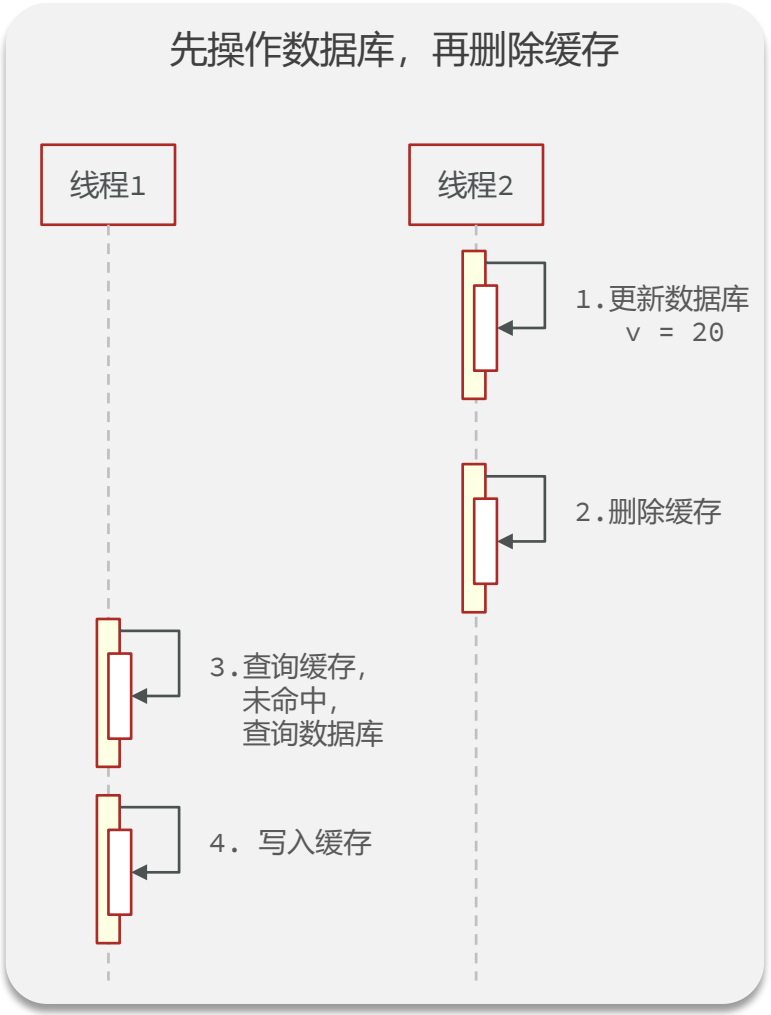
先操作数据库，再删除缓存

Cache Aside Pattern

缓存 10

数据库 10

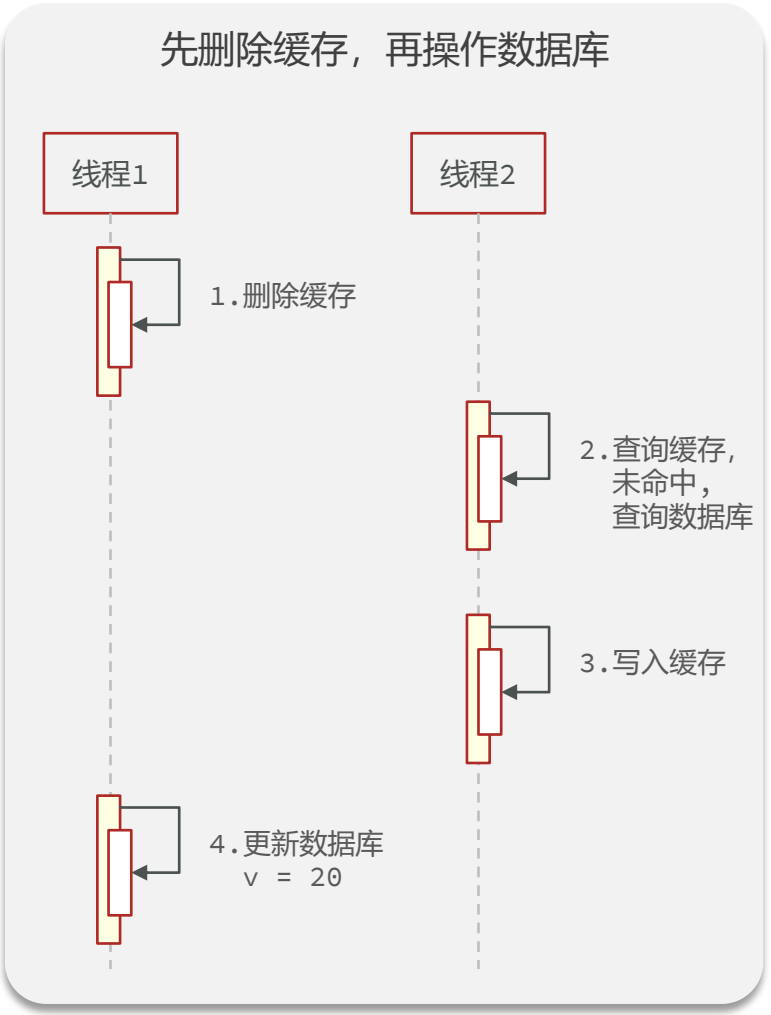


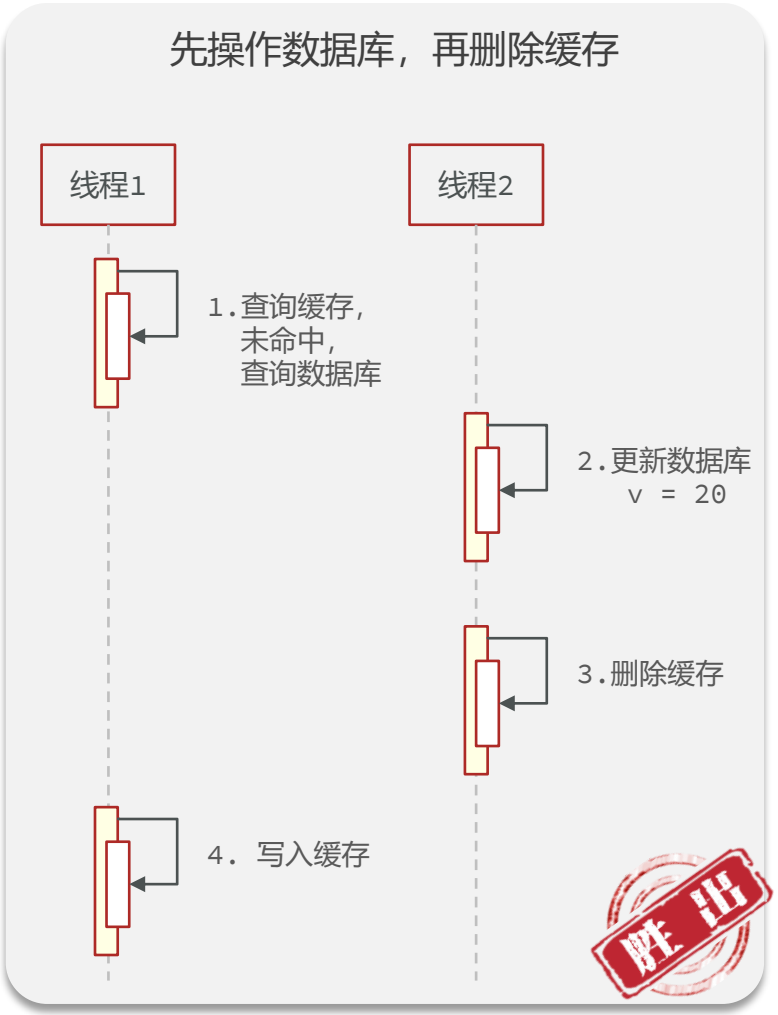


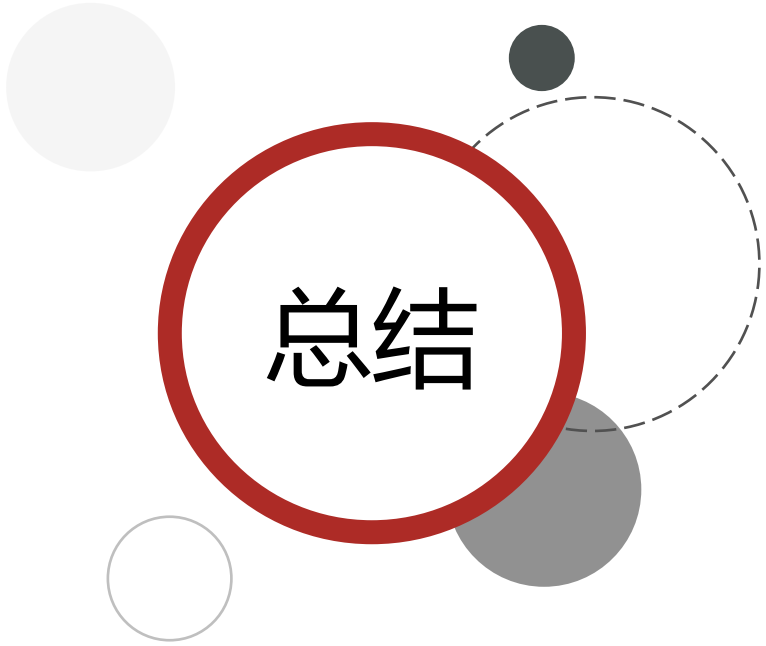
Cache Aside Pattern

缓存 10

数据库 20







缓存更新策略的最佳实践方案：

1. 低一致性需求：使用Redis自带的内存淘汰机制
2. 高一致性需求：主动更新，并以超时剔除作为兜底方案
   - ◆ 读操作：
     - 缓存命中则直接返回
     - 缓存未命中则查询数据库，并写入缓存，设定超时时间
   - ◆ 写操作：
     - 先写数据库，然后再删除缓存
     - 要确保数据库与缓存操作的原子性

案例 给查询商铺的缓存添加超时剔除和主动更新的策略

修改ShopController中的业务逻辑，满足下面的需求：

① 根据id查询店铺时，如果缓存未命中，则查询数据库，将数据库结果写入缓存，并设置超时时间

② 根据id修改店铺时，先修改数据库，再删除缓存

目录

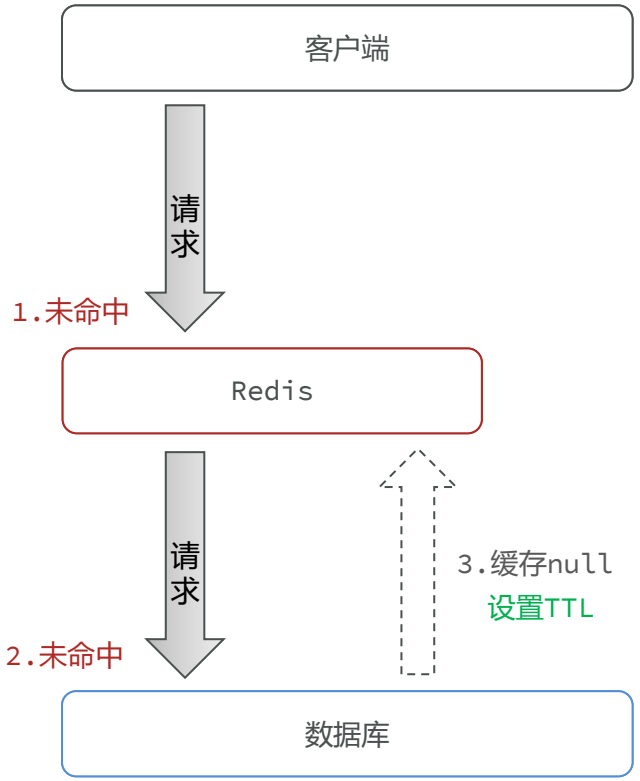
Contents

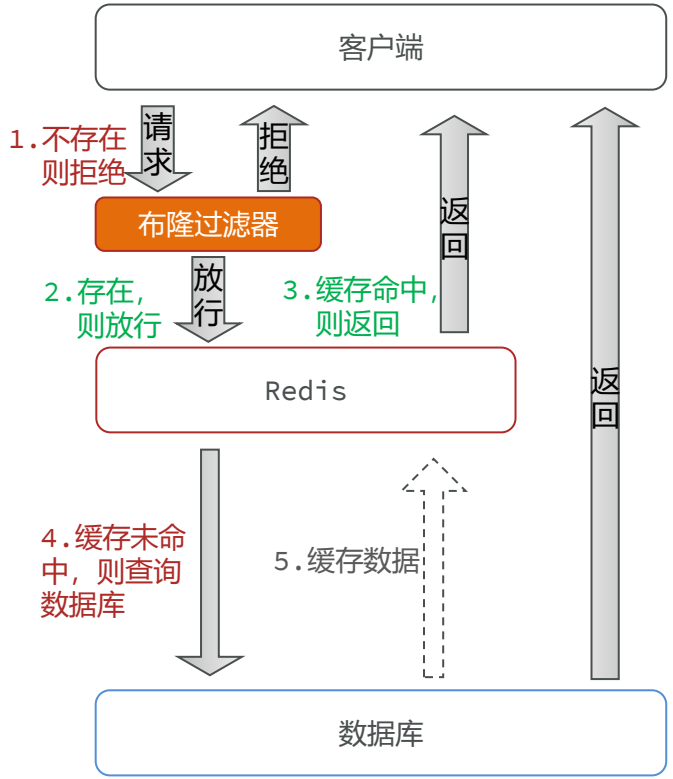
缓存穿透

缓存穿透是指客户端请求的数据在缓存中和数据库中都不存在，这样缓存永远不会生效，这些请求都会打到数据库。

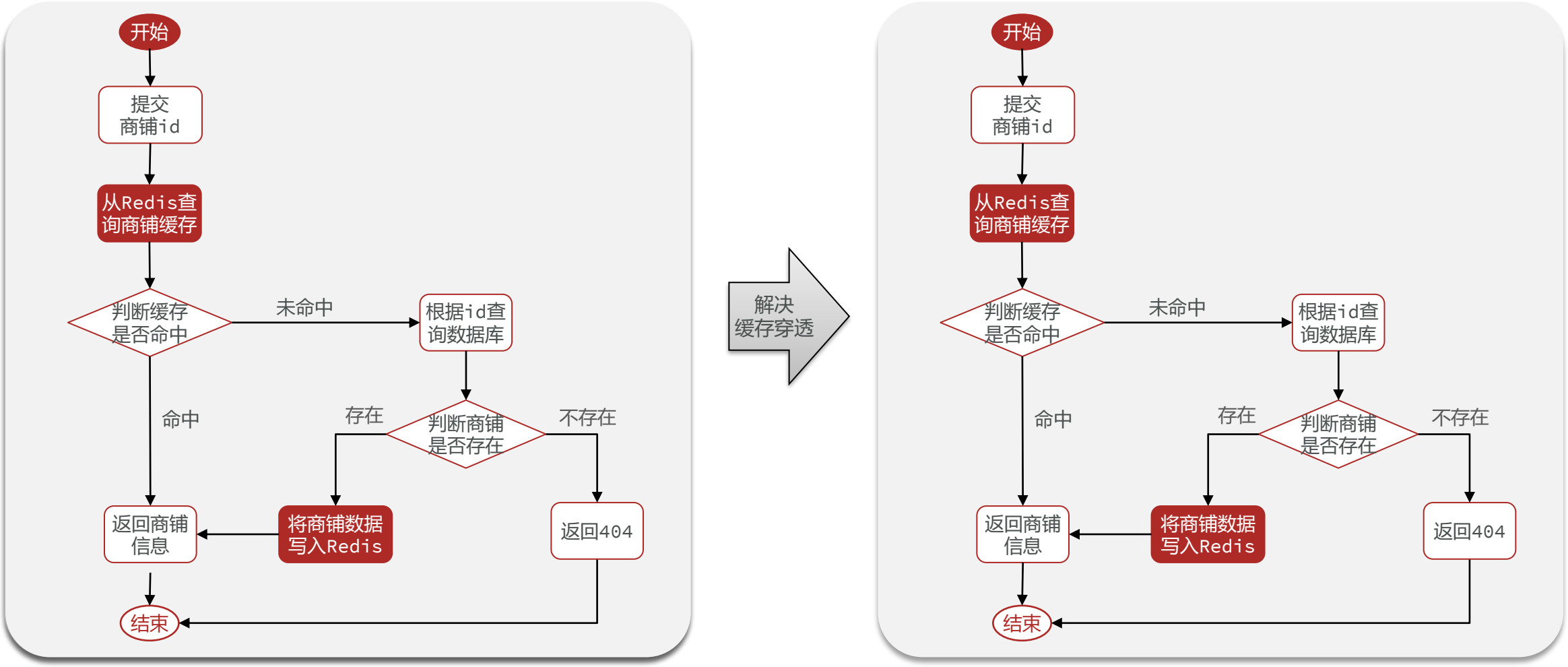
常见的解决方案有两种：

- 缓存空对象
  - 优点：实现简单，维护方便
  - 缺点：
    - 额外的内存消耗
    - 可能造成短期的不一致
- 布隆过滤
  - 优点：内存占用较少，没有多余key
  - 缺点：
    - 实现复杂
    - 存在误判可能

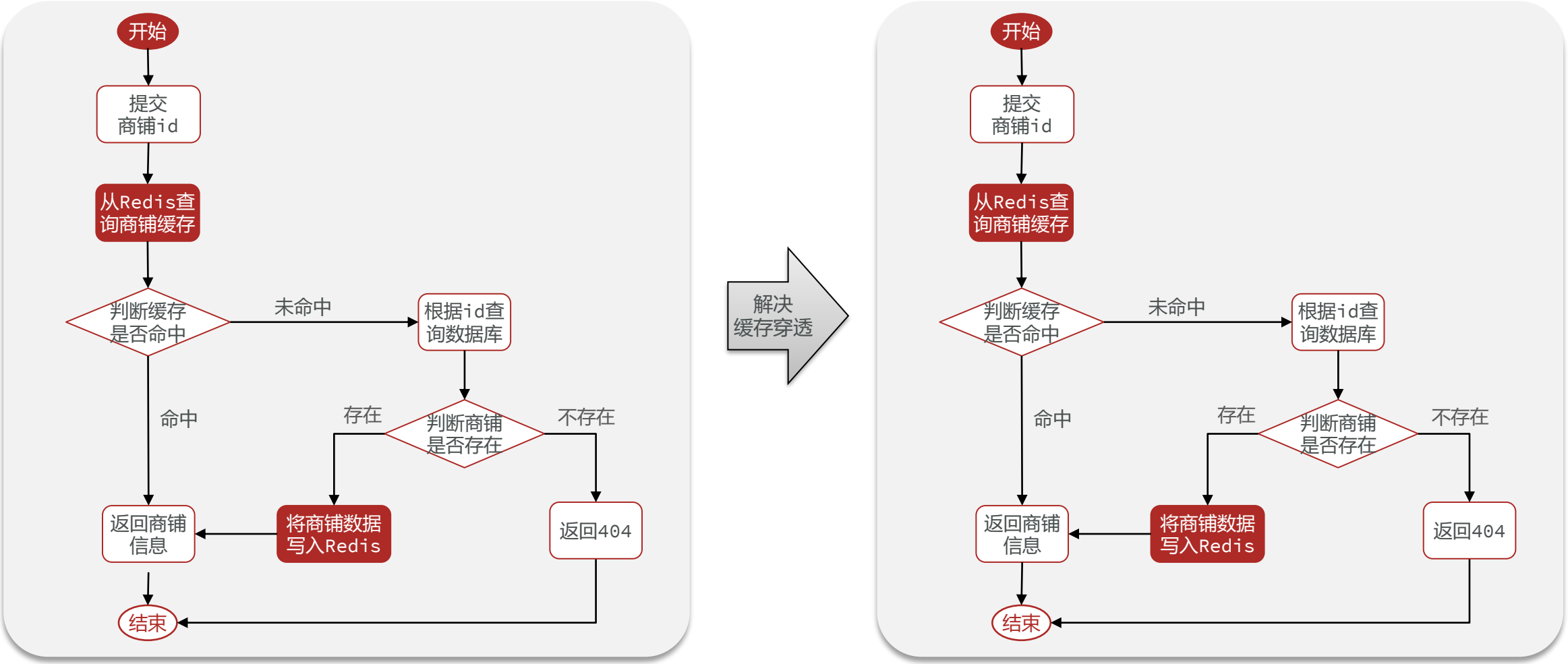




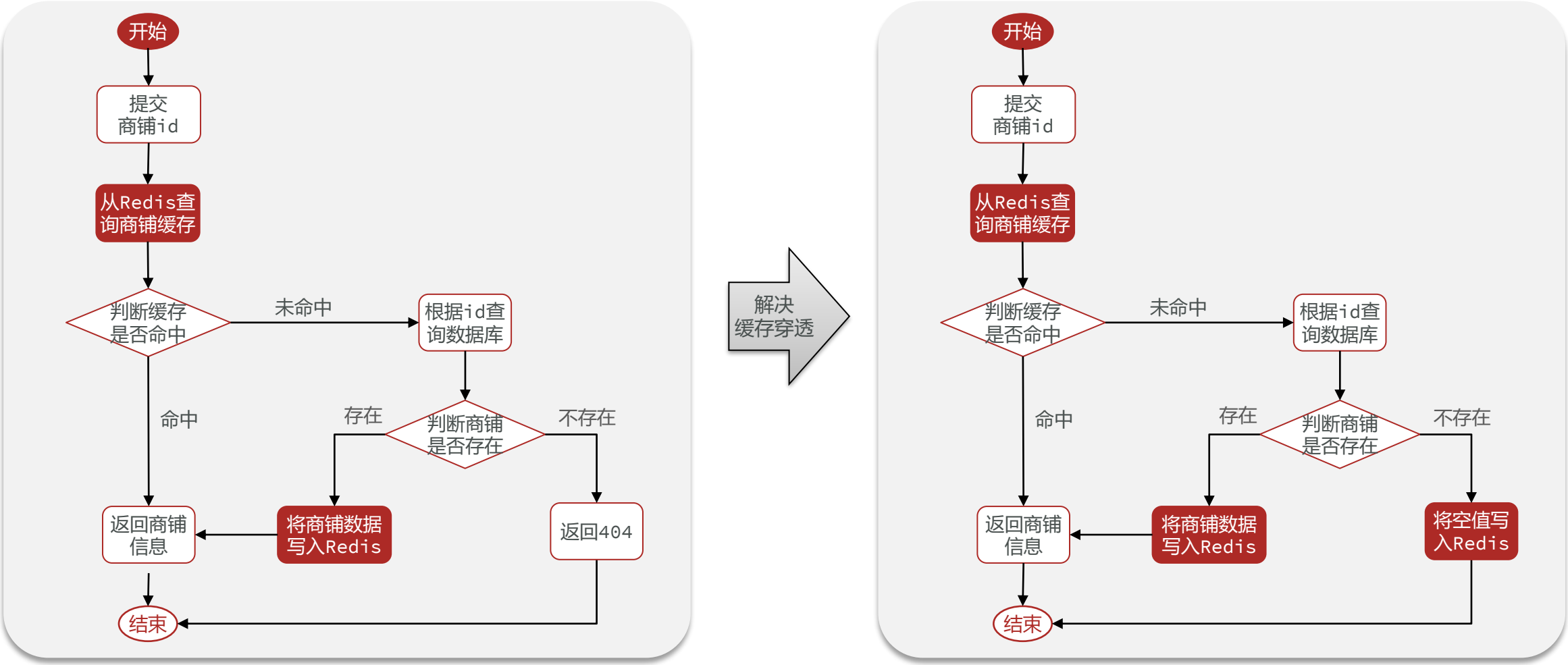
缓存穿透


开始
提交
商铺id
从Redis查
询商铺缓存
判断缓存
是否命中
未命中
根据id查
询数据库
命中
存在
判断商铺
是否存在
不存在
返回商铺
信息
将商铺数据
写入Redis
返回404
结束
解决
缓存穿透
开始
提交
商铺id
从Redis查
询商铺缓存
判断缓存
是否命中
未命中
根据id查
询数据库
命中
存在
判断商铺
是否存在
不存在
返回商铺
信息
将商铺数据
写入Redis
返回404
结束

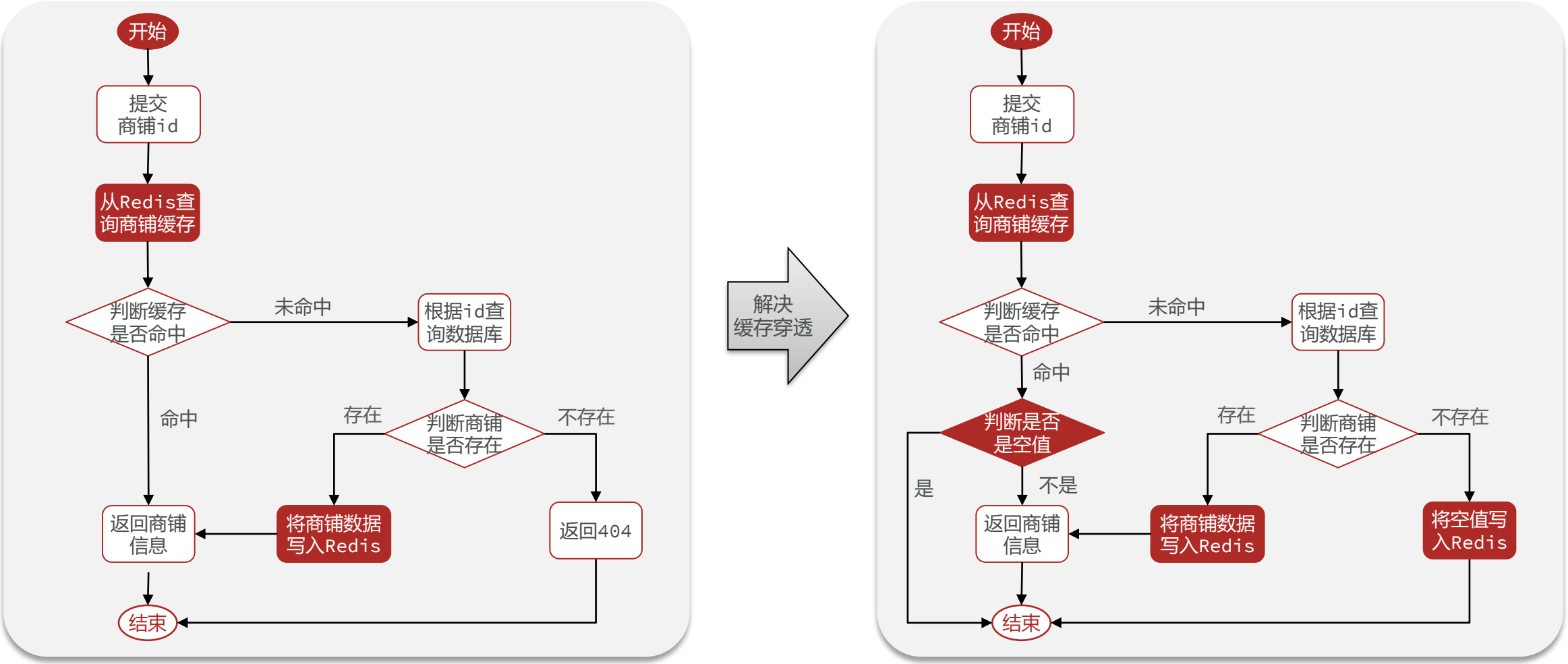
缓存穿透

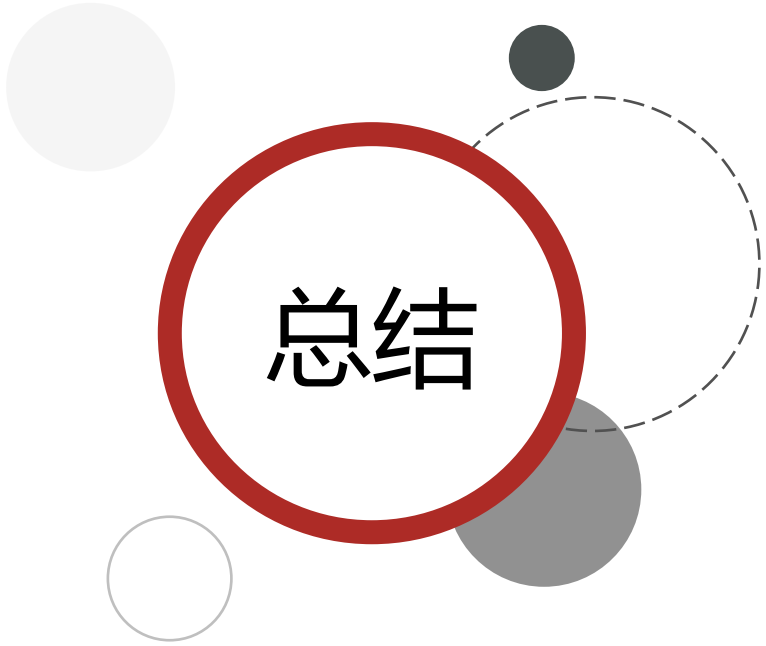

开始
提交
商铺id
从Redis查
询商铺缓存
判断缓存
是否命中
未命中
根据id查
询数据库
命中
存在
判断商铺
是否存在
不存在
返回商铺
信息
将商铺数据
写入Redis
返回404
结束
解决
缓存穿透
开始
提交
商铺id
从Redis查
询商铺缓存
判断缓存
是否命中
未命中
根据id查
询数据库
命中
存在
判断商铺
是否存在
不存在
返回商铺
信息
将商铺数据
写入Redis
返回404
结束

缓存穿透


开始
提交商铺id
从Redis查询商铺缓存
判断缓存是否命中
未命中
根据id查询数据库
命中
存在
判断商铺是否存在
不存在
返回商铺信息
将商铺数据写入Redis
返回404
结束
解决缓存穿透
开始
提交商铺id
从Redis查询商铺缓存
判断缓存是否命中
未命中
根据id查询数据库
命中
存在
判断商铺是否存在
不存在
返回商铺信息
将商铺数据写入Redis
将空值写入Redis
结束

缓存穿透


开始
提交商铺id
从Redis查询商铺缓存
判断缓存是否命中
未命中
根据id查询数据库
命中
存在
判断商铺是否存在
不存在
返回商铺信息
将商铺数据写入Redis
返回404
结束
解决缓存穿透
开始
提交商铺id
从Redis查询商铺缓存
判断缓存是否命中
未命中
根据id查询数据库
命中
判断是否是空值
存在
判断商铺是否存在
不存在
是
不是
返回商铺信息
将商铺数据写入Redis
将空值写入Redis
结束



缓存穿透产生的原因是什么？

- 用户请求的数据在缓存中和数据库中都不存在，不断发起这样的请求，给数据库带来巨大压力

缓存穿透的解决方案有哪些？

- 缓存null值
- 布隆过滤
- 增强id的复杂度，避免被猜测id规律
- 做好数据的基础格式校验
- 加强用户权限校验
- 做好热点参数的限流

目录

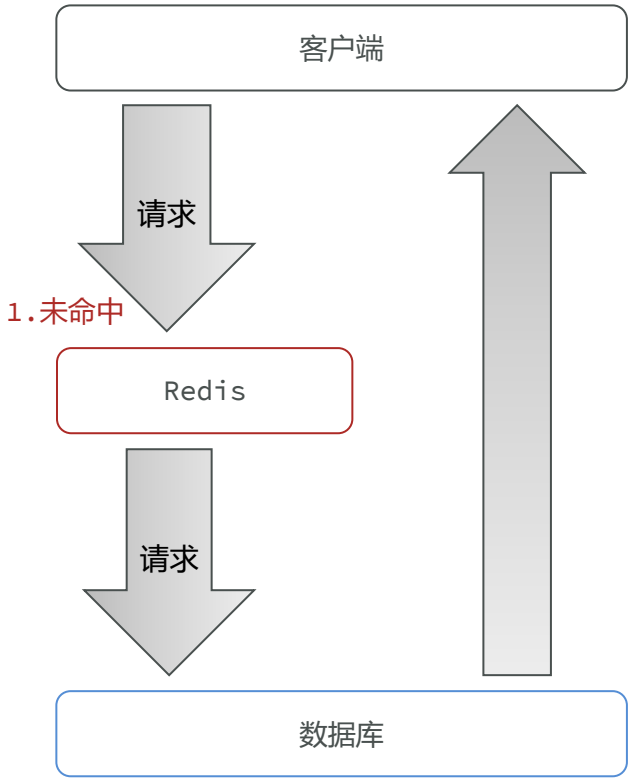
Contents

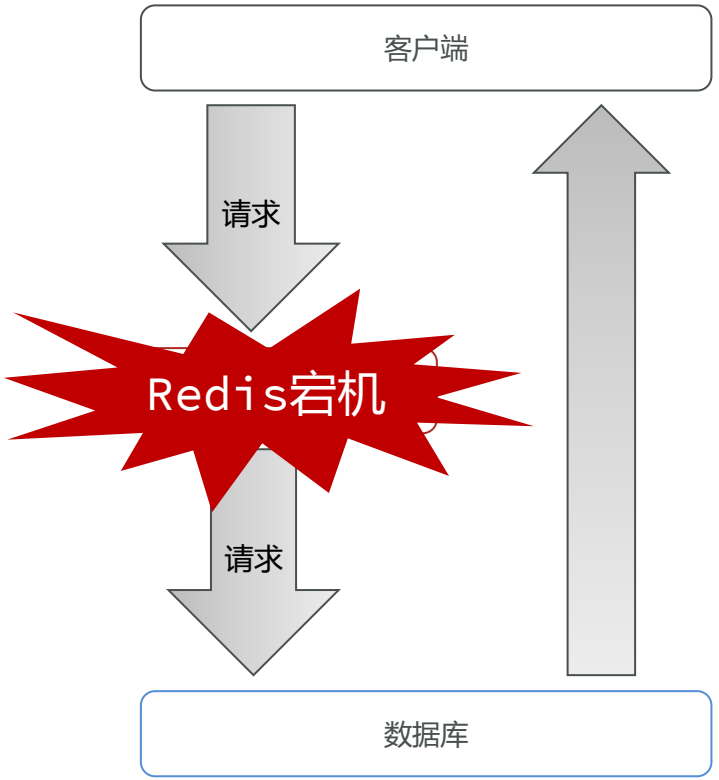
缓存雪崩

缓存雪崩是指在同一时段大量的缓存key同时失效或者Redis服务宕机，导致大量请求到达数据库，带来巨大压力。

解决方案：

- 给不同的Key的TTL添加随机值
- 利用Redis集群提高服务的可用性
- 给缓存业务添加降级限流策略
- 给业务添加多级缓存





目录

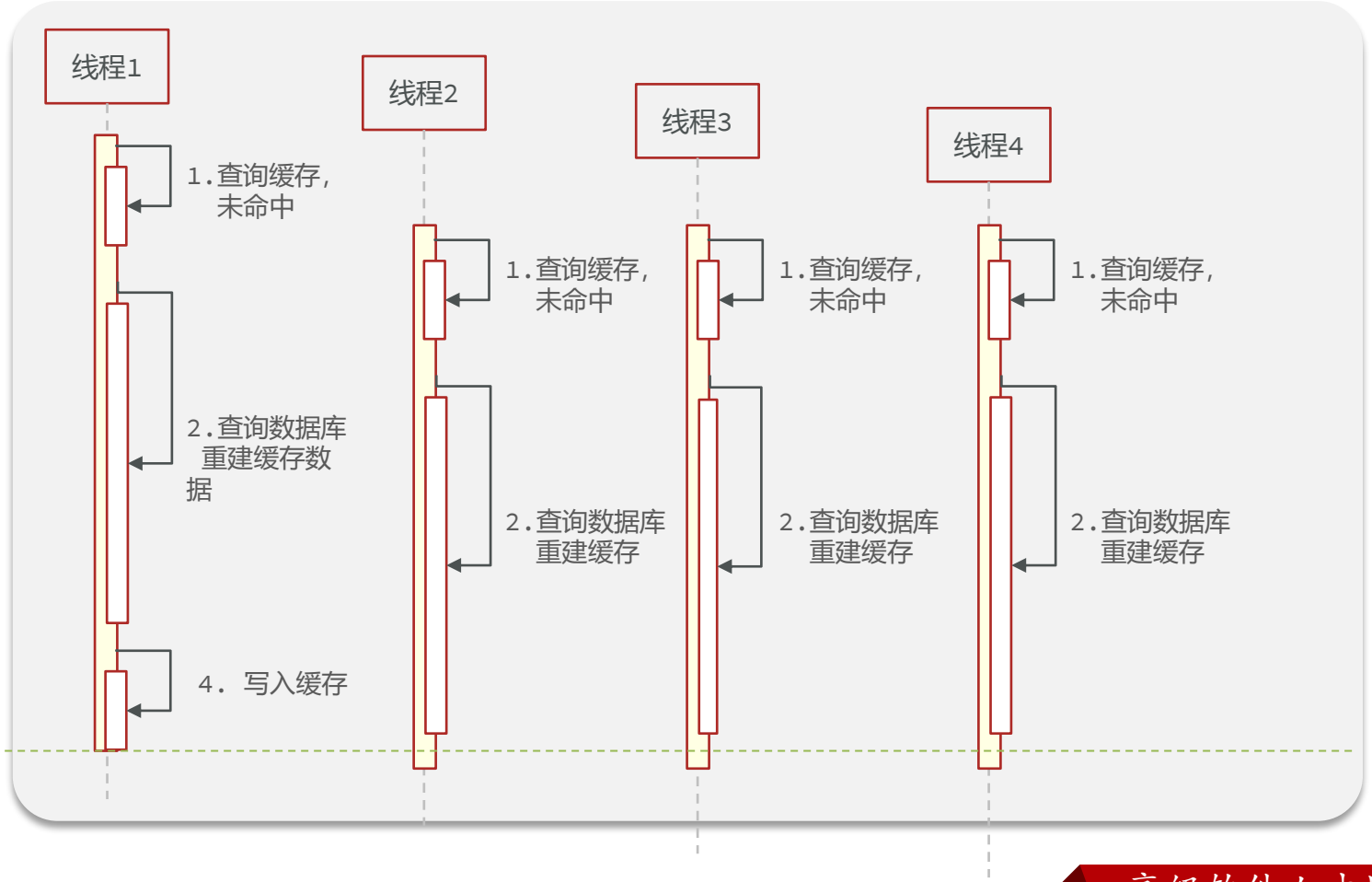
Contents

缓存击穿

缓存击穿问题也叫热点Key问题，就是一个被**高并发访问**并且**缓存重建业务较复杂**的key突然失效了，无数的请求访问会在瞬间给数据库带来巨大的冲击。

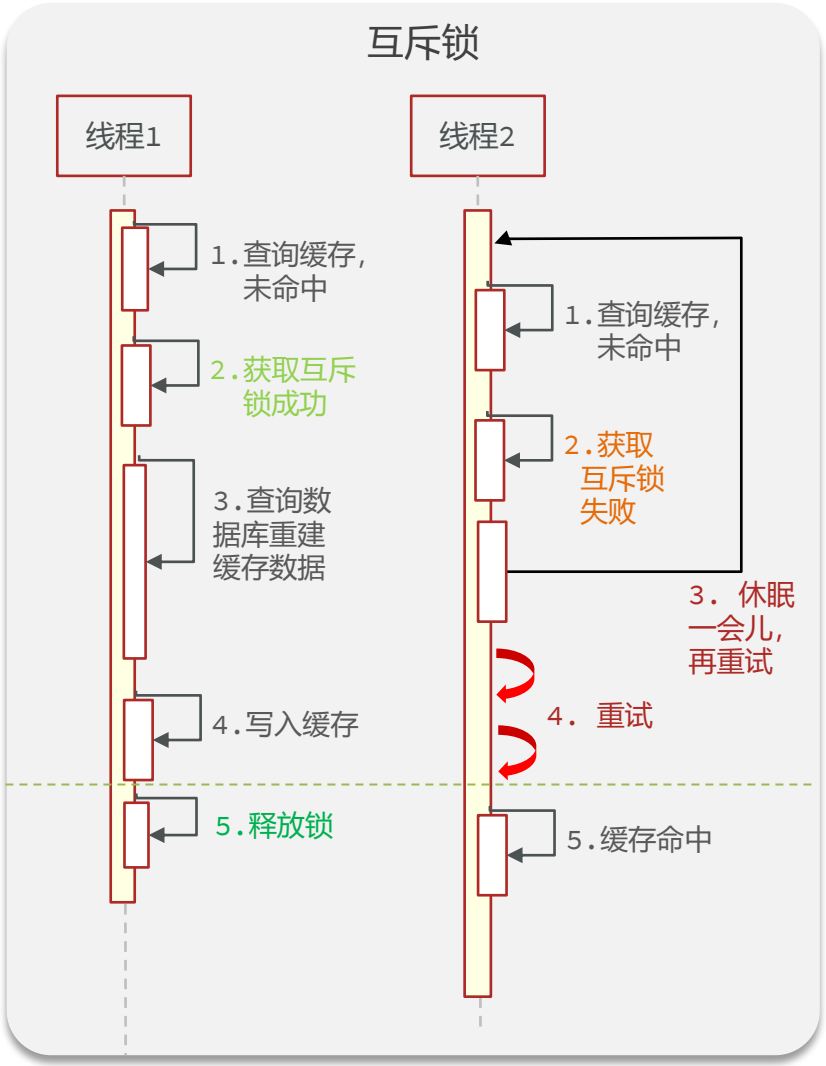
常见的解决方案有两种：

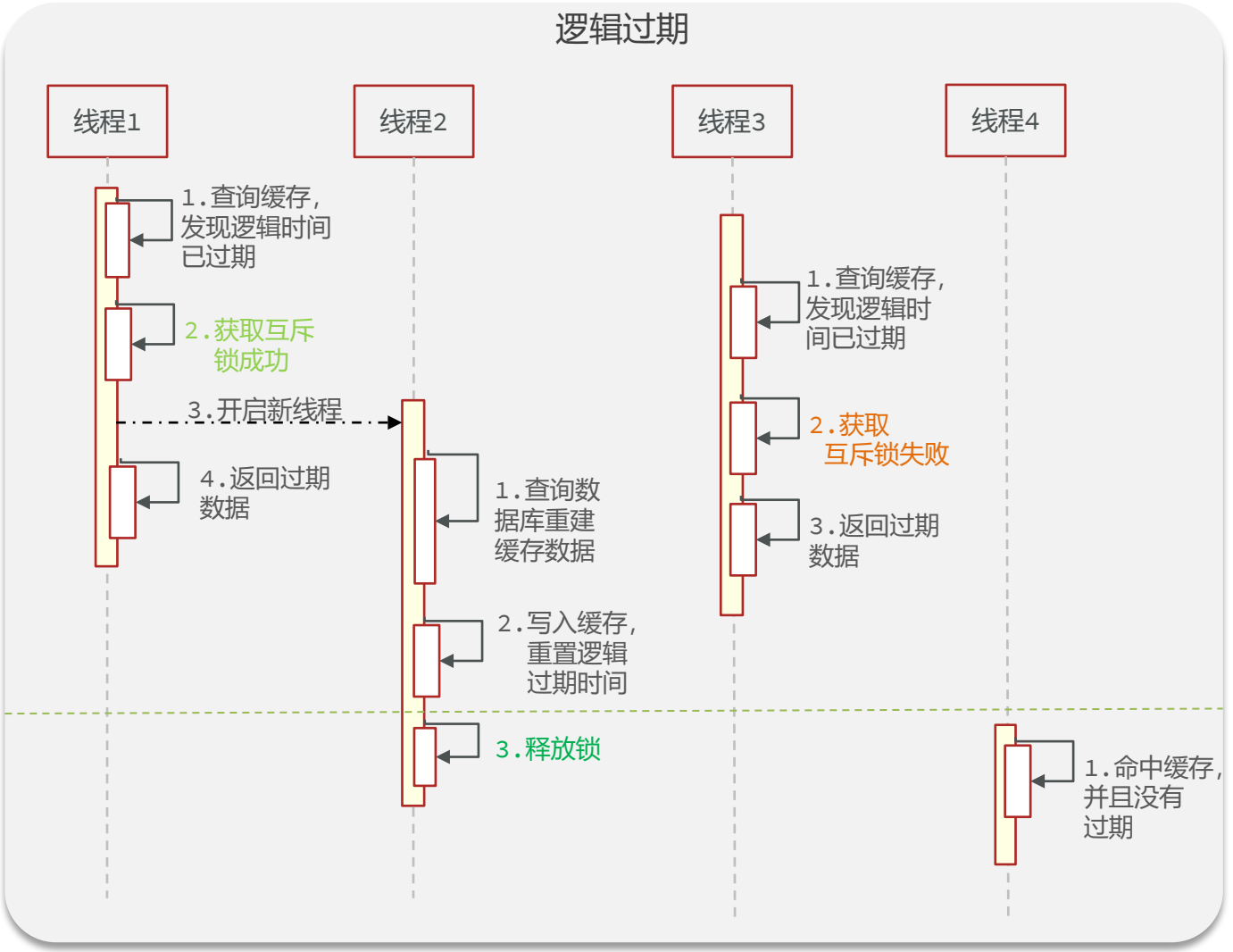
- 互斥锁
- 逻辑过期



缓存击穿

KEY	VALUE
heima:user:1	{name:"Jack", age:21, expire:152141223}



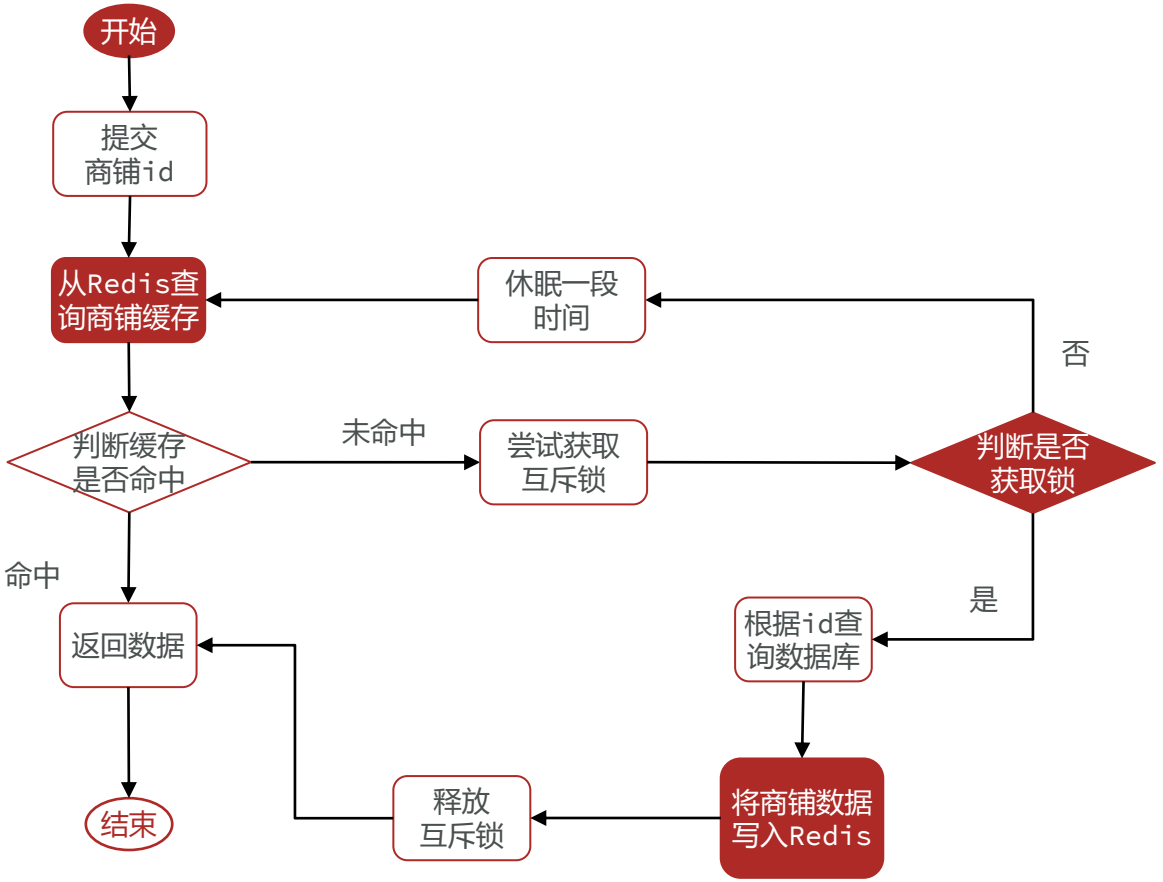


缓存击穿

解决方案	优点	缺点
互斥锁	• 没有额外的内存消耗 • 保证一致性 • 实现简单	• 线程需要等待，性能受影响 • 可能有死锁风险
逻辑过期	• 线程无需等待，性能较好	• 不保证一致性 • 有额外内存消耗 • 实现复杂

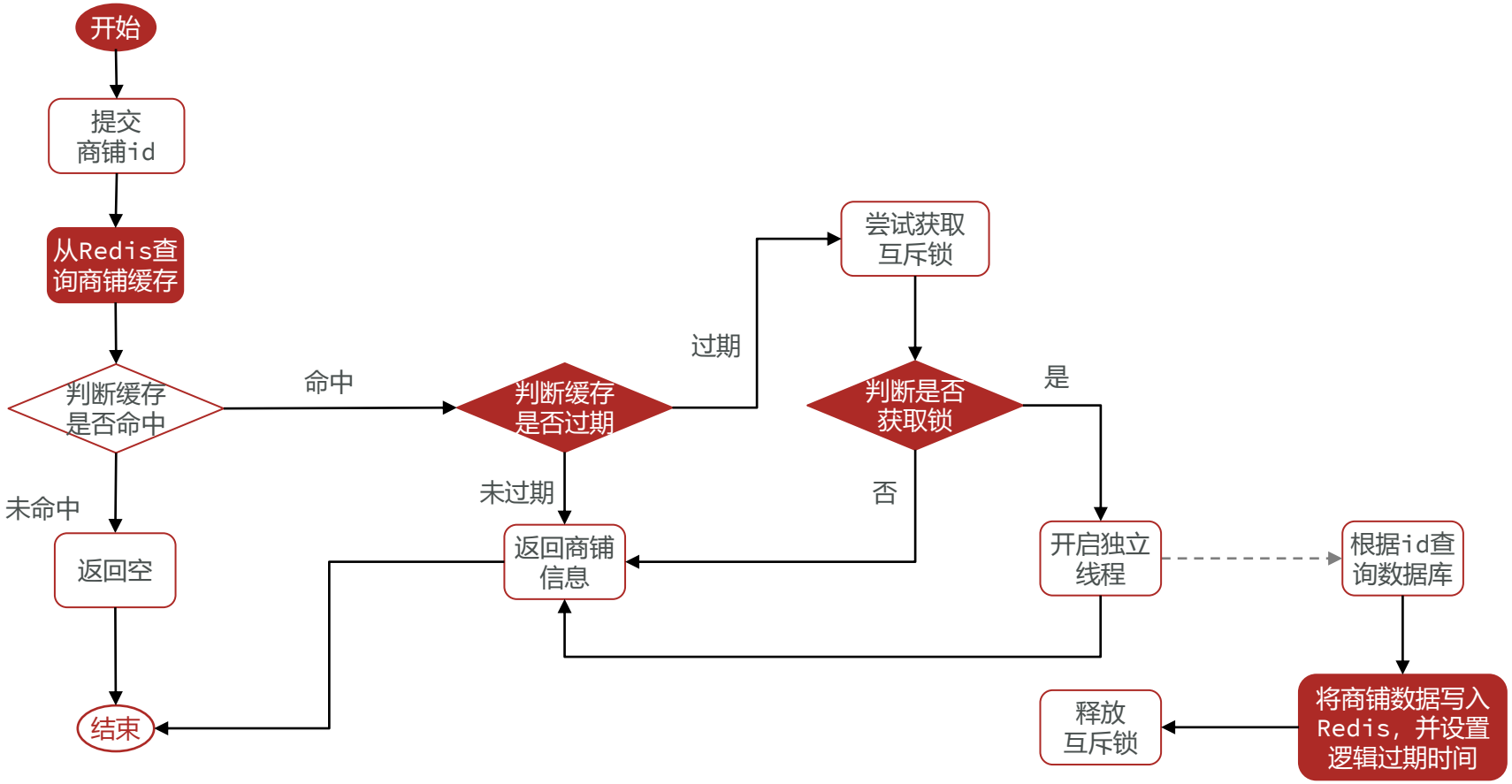
案例 基于互斥锁方式解决缓存击穿问题

需求：修改根据id查询商铺的业务，基于互斥锁方式来解决缓存击穿问题



案例 基于逻辑过期方式解决缓存击穿问题

需求：修改根据id查询商铺的业务，基于逻辑过期方式来解决缓存击穿问题



目录

Contents

案例 缓存工具封装

基于StringRedisTemplate封装一个缓存工具类，满足下列需求：

- ✓ 方法1：将任意Java对象序列化为json并存储在string类型的key中，并且可以设置TTL过期时间
- ✓ 方法2：将任意Java对象序列化为json并存储在string类型的key中，并且可以设置逻辑过期时间，用于处理缓存击穿问题
- ✓ 方法3：根据指定的key查询缓存，并反序列化为指定类型，利用缓存空值的方式解决缓存穿透问题
- ✓ 方法4：根据指定的key查询缓存，并反序列化为指定类型，需要利用逻辑过期解决缓存击穿问题

03 优惠券秒杀

目录

Contents

全局唯一ID

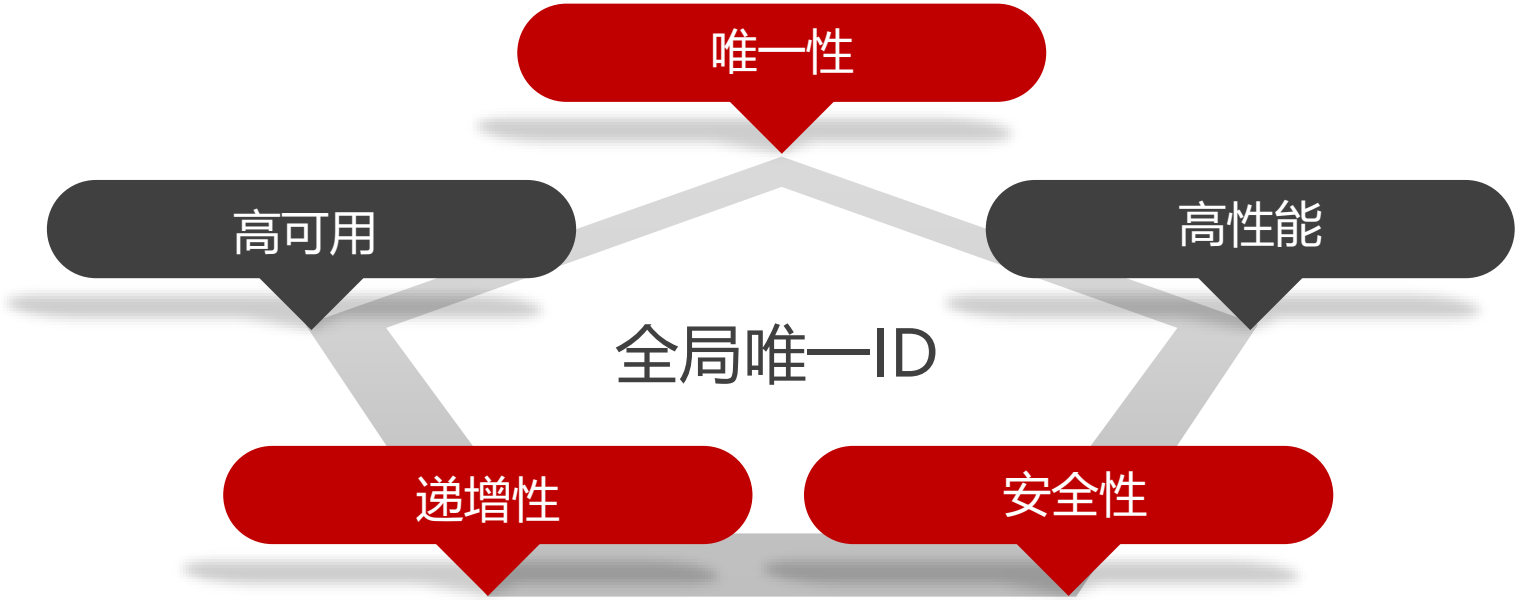
每个店铺都可以发布优惠券：

当用户抢购时，就会生成订单并保存到tb_voucher_order这张表中，而订单表如果使用数据库自增ID就存在一些问题：

- id的规律性太明显
- 受单表数据量的限制

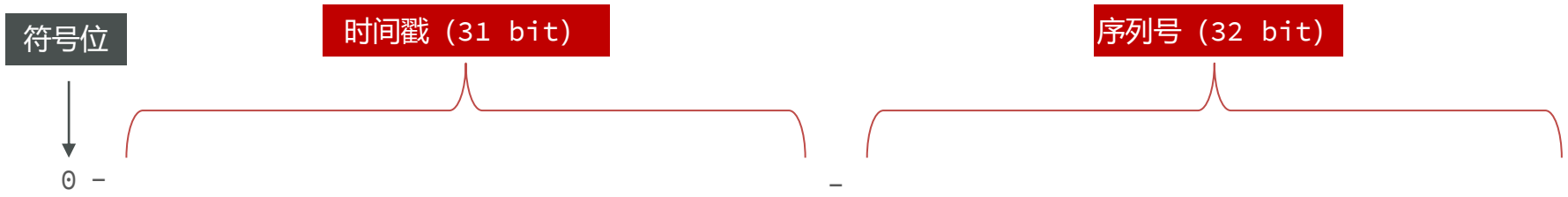
全局唯一ID

全局ID生成器，是一种在分布式系统下用来生成全局唯一ID的工具，一般要满足下列特性：



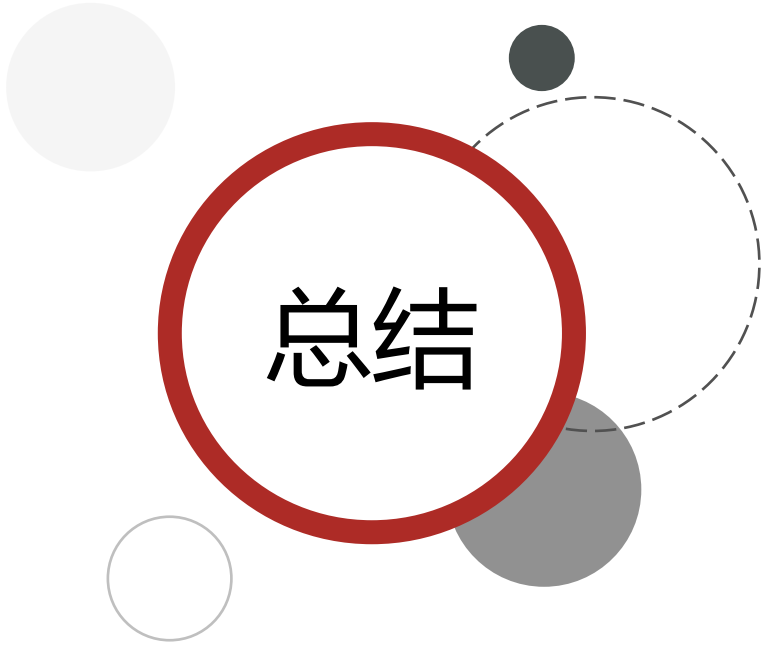
全局ID生成器

为了增加ID的安全性，我们可以不直接使用Redis自增的数值，而是拼接一些其它信息：



ID的组成部分：

- 符号位：1bit，永远为0
- 时间戳：31bit，以秒为单位，可以使用69年
- 序列号：32bit，秒内的计数器，支持每秒产生2^32个不同ID



全局唯一ID生成策略：

- UUID
- Redis自增
- snowflake算法
- 数据库自增

Redis自增ID策略：

- 每天一个key，方便统计订单量
- ID构造是 时间戳 + 计数器

目录

Contents

实现优惠券秒杀下单

每个店铺都可以发布优惠券，分为平价券和特价券。平价券可以任意购买，而特价券需要秒杀抢购：

表关系如下：

- tb_voucher：优惠券的基本信息，优惠金额、使用规则等
- tb_seckill_voucher：优惠券的库存、开始抢购时间，结束抢购时间。特价优惠券才需要填写这些信息

实现优惠券秒杀下单

在VoucherController中提供了一个接口，可以添加秒杀优惠券：

```
@RestController
@RequestMapping("/voucher")
public class VoucherController {

    @Resource
    private IVoucherService voucherService;

    /**
     * 新增秒杀券
     * @param voucher 优惠券信息，包含秒杀信息
     * @return 优惠券id
     */
    @PostMapping("seckill")
    public Result addSeckillVoucher(@RequestBody Voucher voucher) {
        voucherService.addSeckillVoucher(voucher);
        return Result.ok(voucher.getId());
    }
```

实现优惠券秒杀下单

用户可以在店铺页面中抢购这些优惠券：

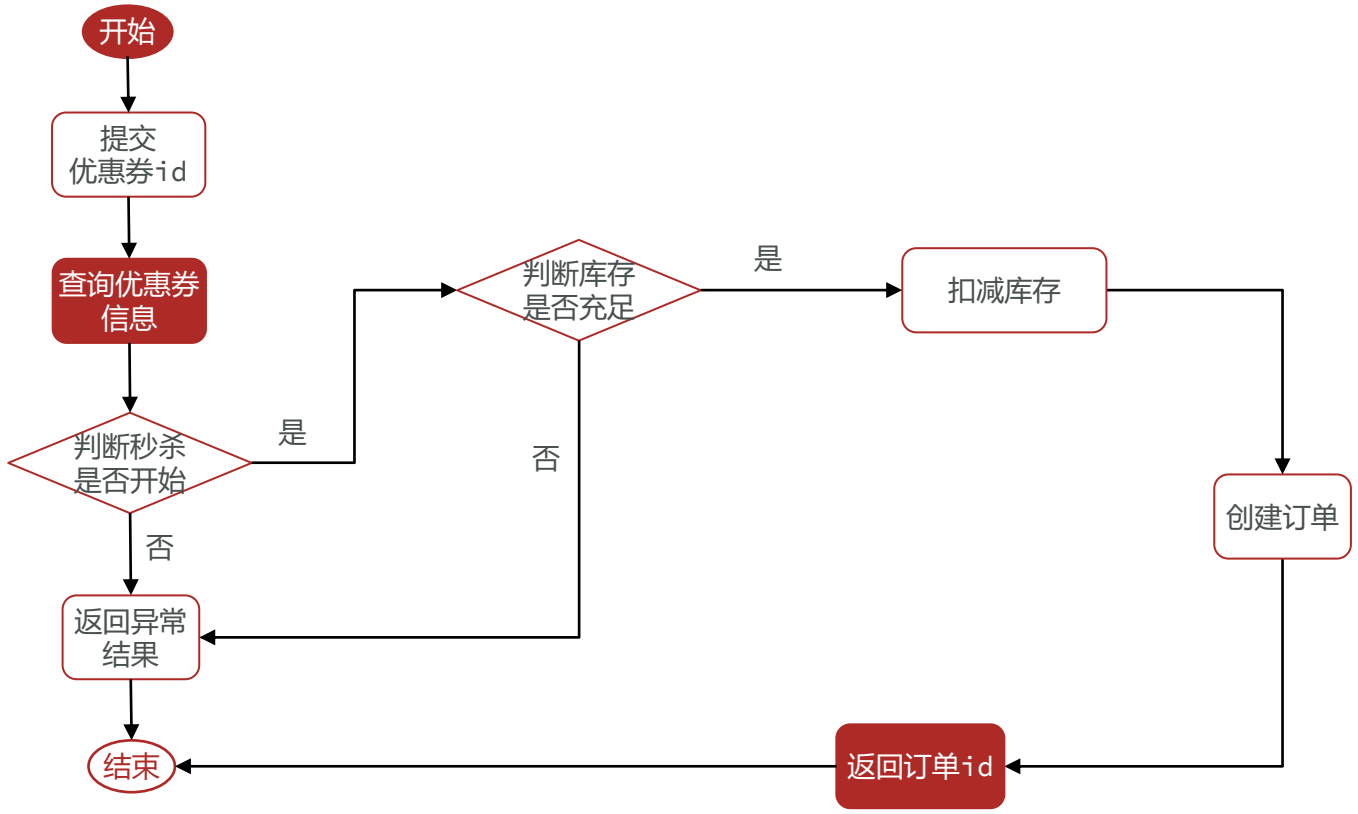
	说明
请求方式	POST
请求路径	/voucher-order/seckill/{id}
请求参数	id,优惠券id
返回值	订单id

案例

实现优惠券秒杀的下单功能

下单时需要判断两点：

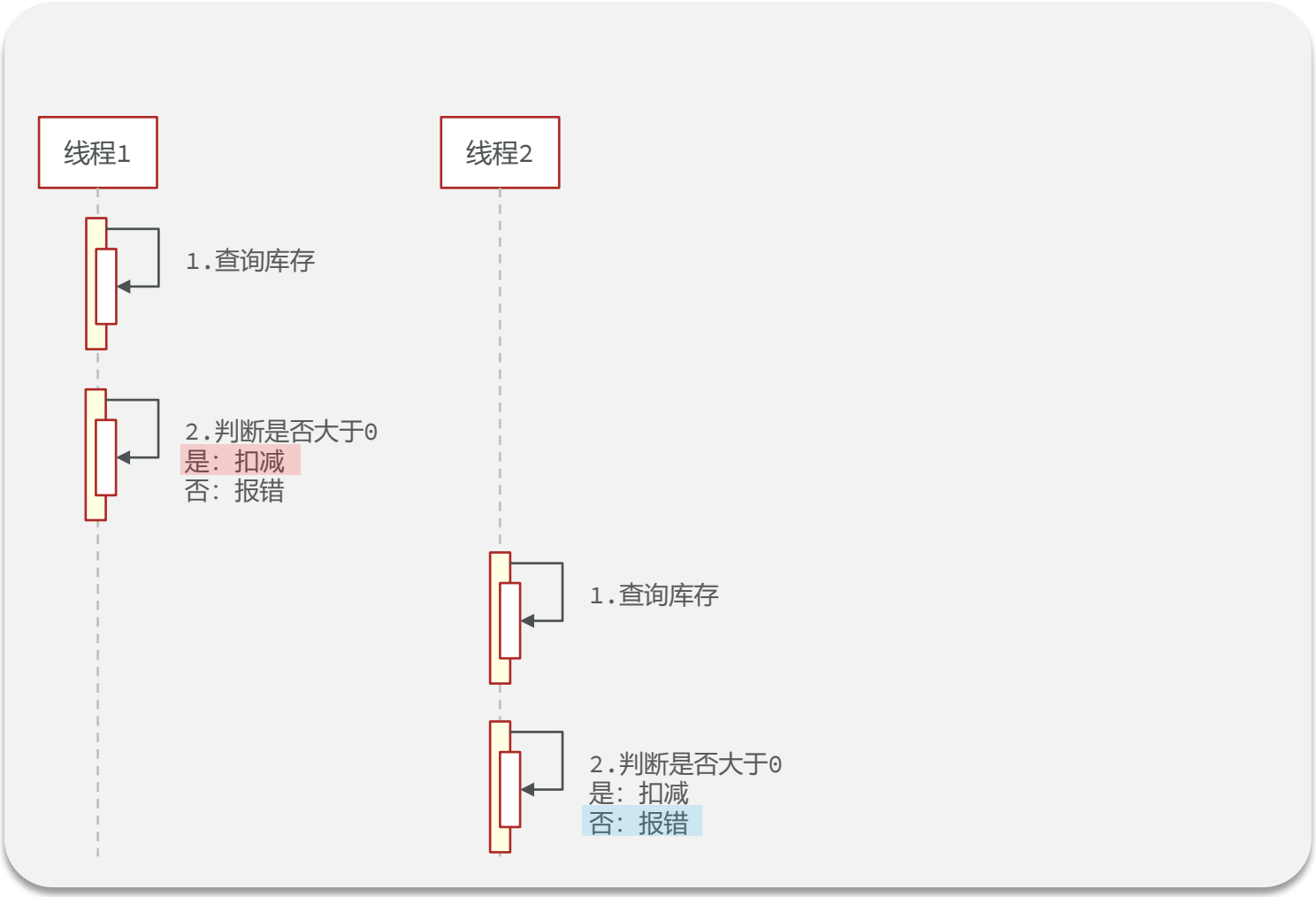
- 秒杀是否开始或结束，如果尚未开始或已经结束则无法下单
- 库存是否充足，不足则无法下单



目录

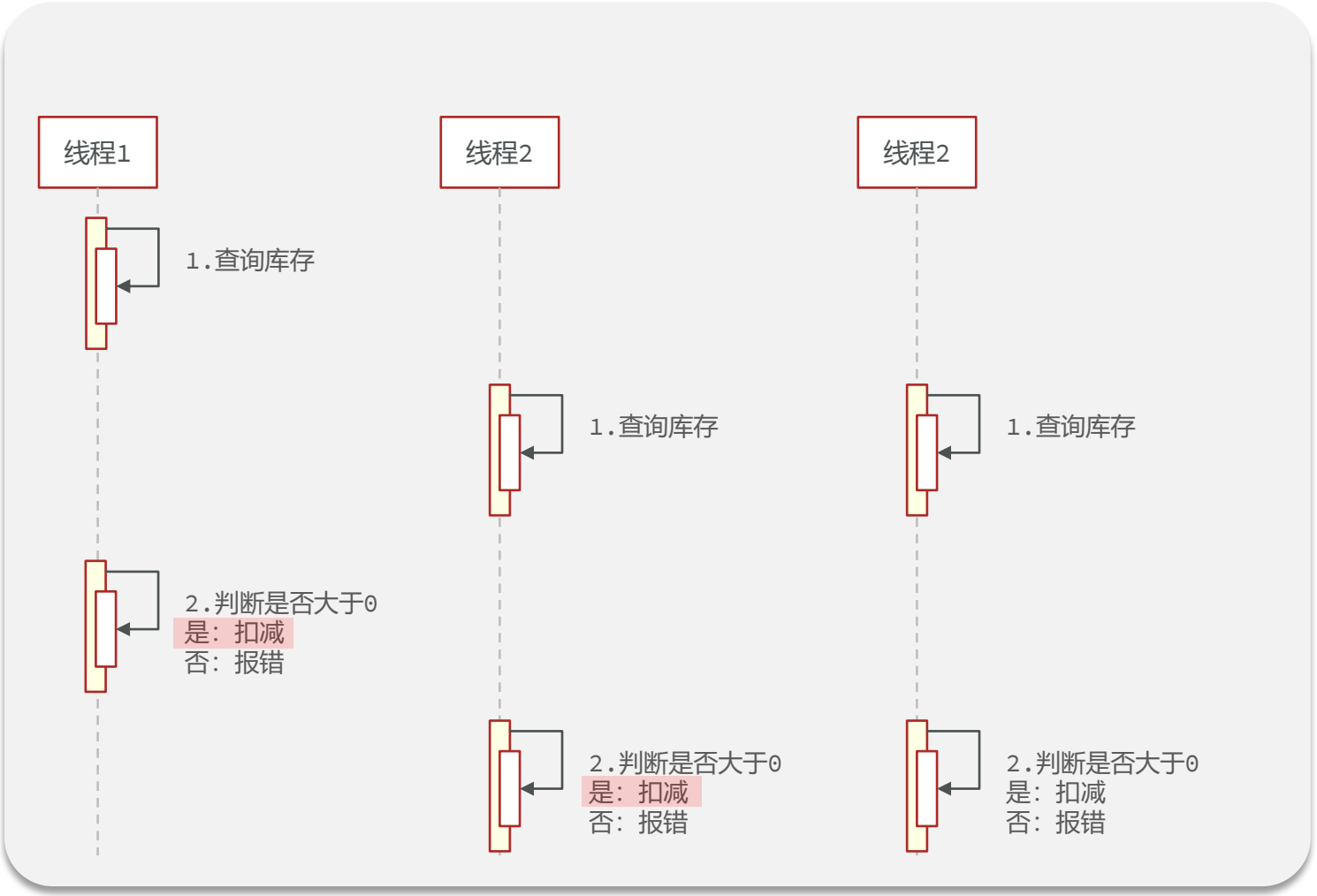
Contents

超卖问题



超卖问题


库存
-1


线程1
线程2
线程2
1.查询库存
1.查询库存
1.查询库存
2.判断是否大于0
是：扣减
否：报错
2.判断是否大于0
是：扣减
否：报错
2.判断是否大于0
是：扣减
否：报错

超卖问题

超卖问题是典型的多线程安全问题，针对这一问题的常见解决方案就是加锁：

悲观锁

认为线程安全问题一定会发生，因此在操作数据之前先获取锁，确保线程串行执行。

- 例如Synchronized、Lock都属于悲观锁

锁

乐观锁

认为线程安全问题不一定会发生，因此不加锁，只是在更新数据时去判断有没有其它线程对数据做了修改。

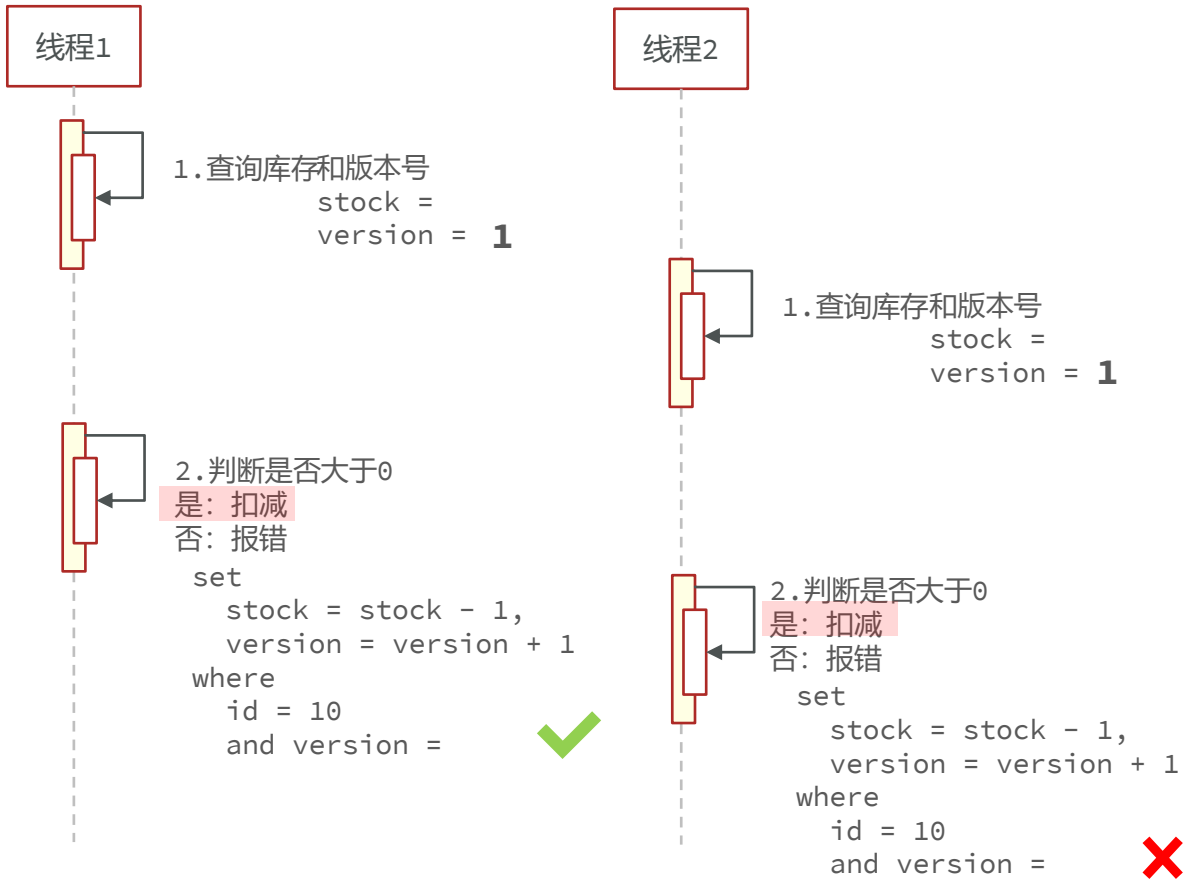
- 如果没有修改则认为是安全的，自己才更新数据。
- 如果已经被其它线程修改说明发生了安全问题，此时可以重试或异常。

乐观锁

乐观锁的关键是判断之前查询得到的数据是否有被修改过，常见的方式有两种：

- **版本号法**

id	stock	version
10	1	1

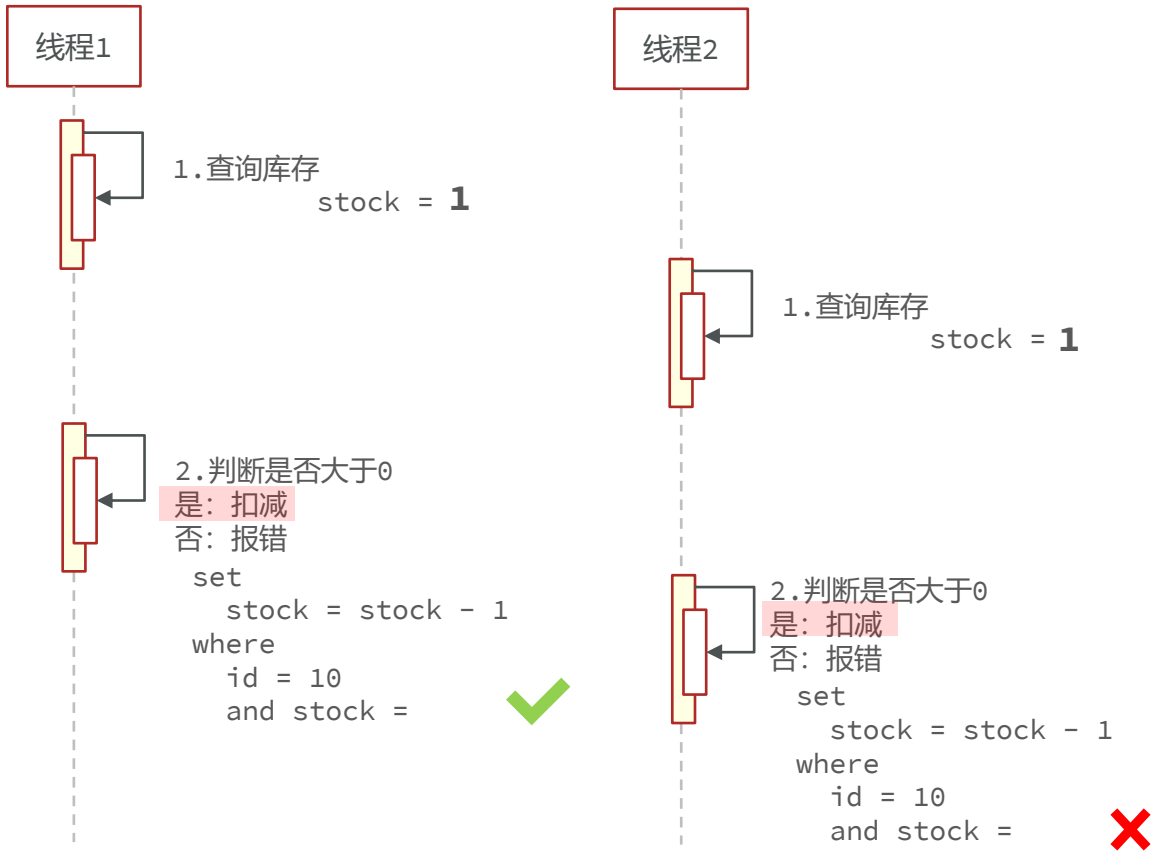


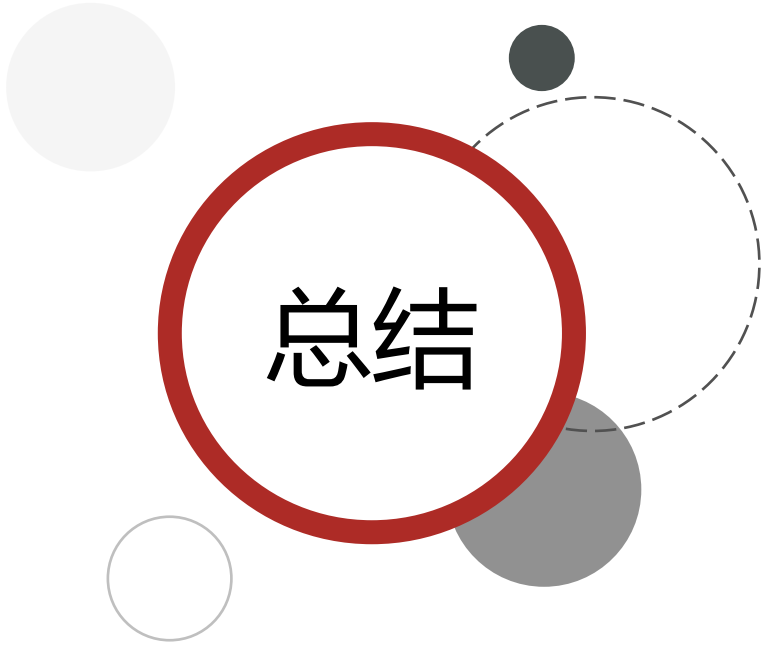
乐观锁

乐观锁的关键是判断之前查询得到的数据是否有被修改过，常见的方式有两种：

- **CAS法**

id	stock
10	1





超卖这样的线程安全问题，解决方案有哪些？

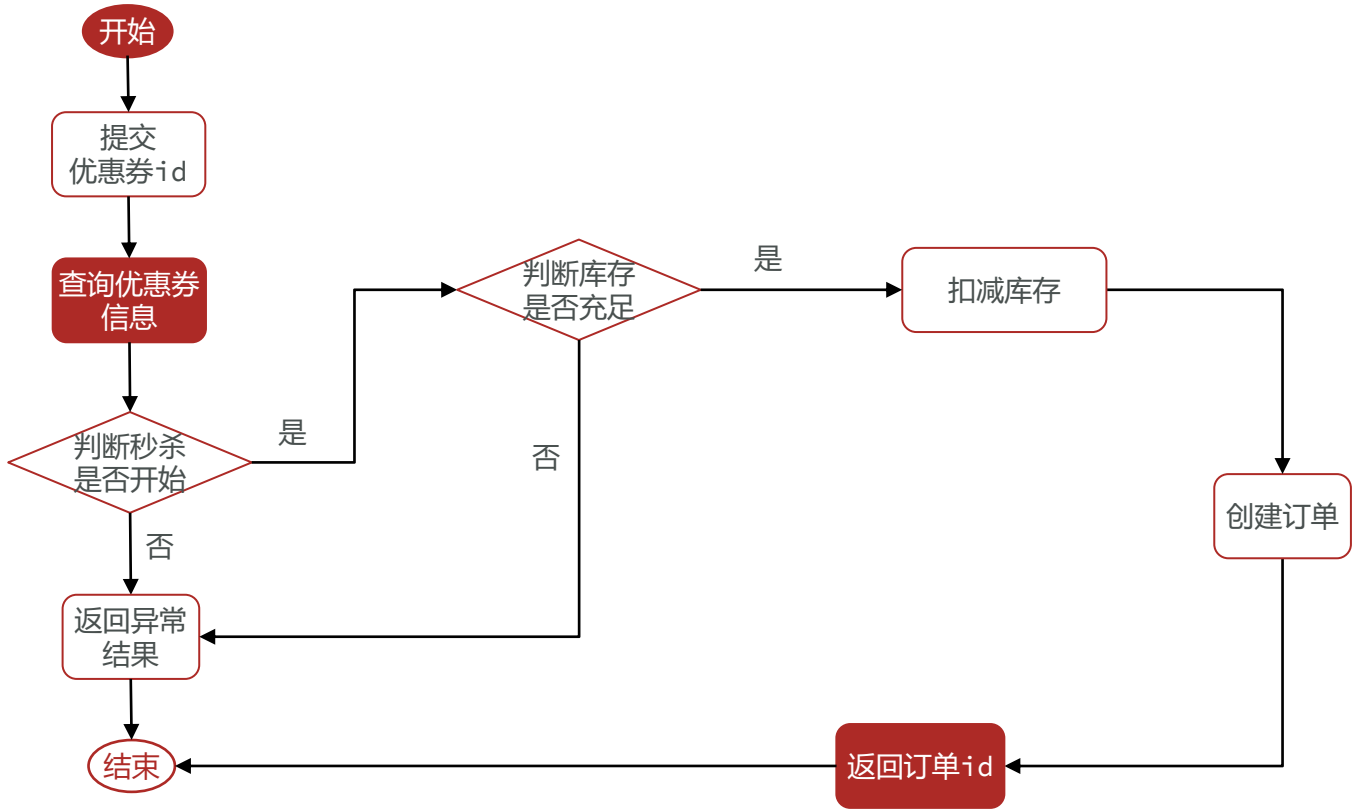
1. 悲观锁：添加同步锁，让线程串行执行
   - 优点：简单粗暴
   - 缺点：性能一般
2. 乐观锁：不加锁，在更新时判断是否有其它线程在修改
   - 优点：性能好
   - 缺点：存在成功率低的问题

目录

Contents

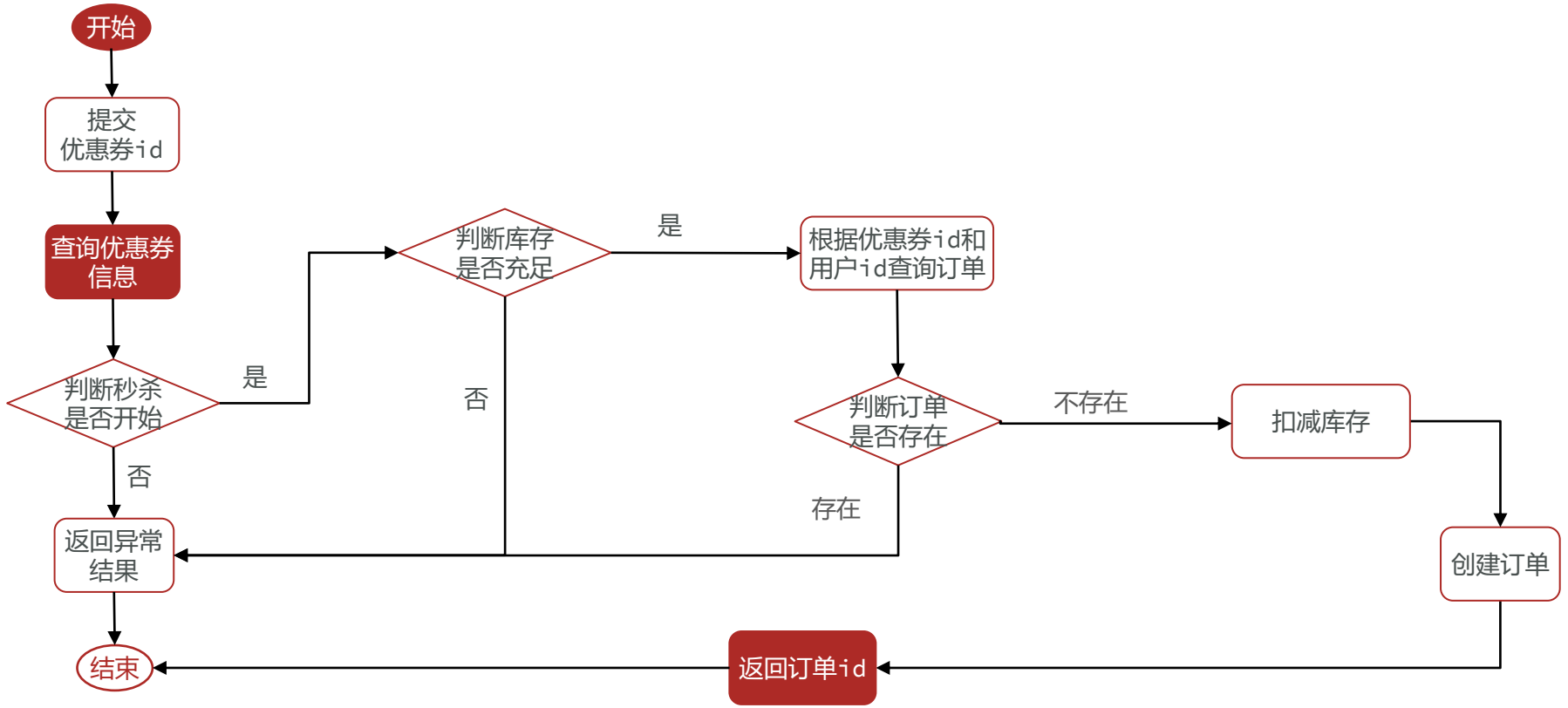
案例 一人一单

需求：修改秒杀业务，要求同一个优惠券，一个用户只能下一单



案例 一人一单

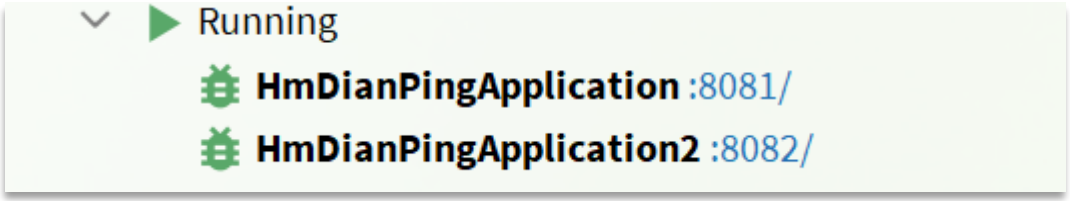
需求：修改秒杀业务，要求同一个优惠券，一个用户只能下一单



一人一单的并发安全问题

通过加锁可以解决在单机情况下的一人一单安全问题，但是在集群模式下就不行了。

1. 我们将服务启动两份，端口分别为8081和8082：

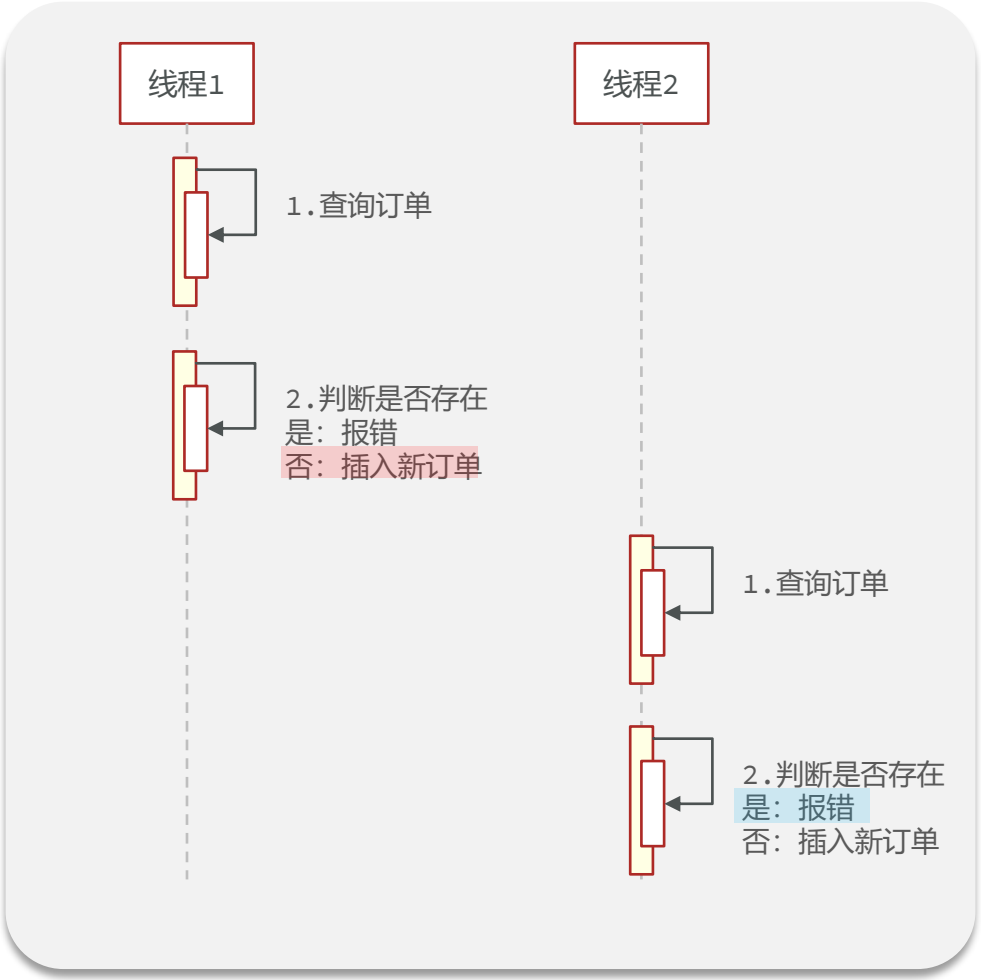


2. 然后修改nginx的conf目录下的nginx.conf文件，配置反向代理和负载均衡：

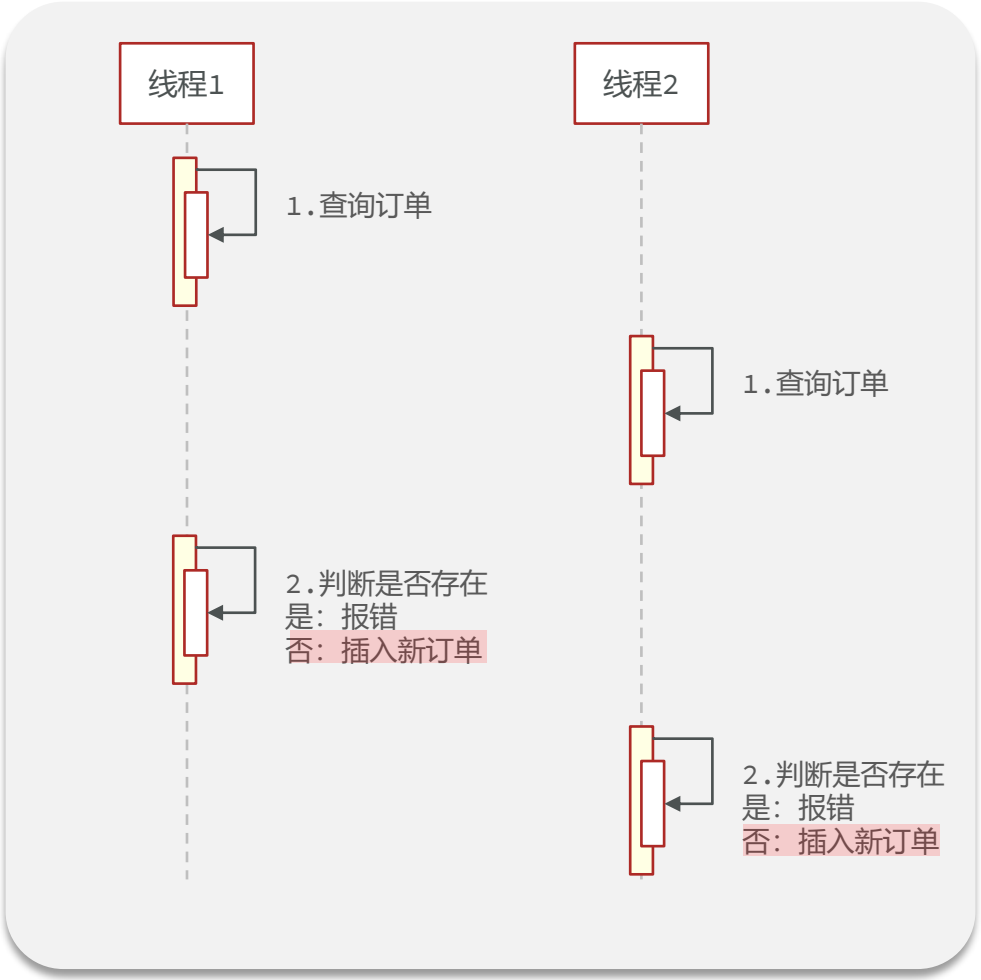
```
upstream backend {
    server 127.0.0.1:8081 max_fails=5 fail_timeout=10s weight=1;
    server 127.0.0.1:8082 max_fails=5 fail_timeout=10s weight=1;
}
```

现在，用户请求会在这两个节点上负载均衡，再次测试下是否存在线程安全问题。

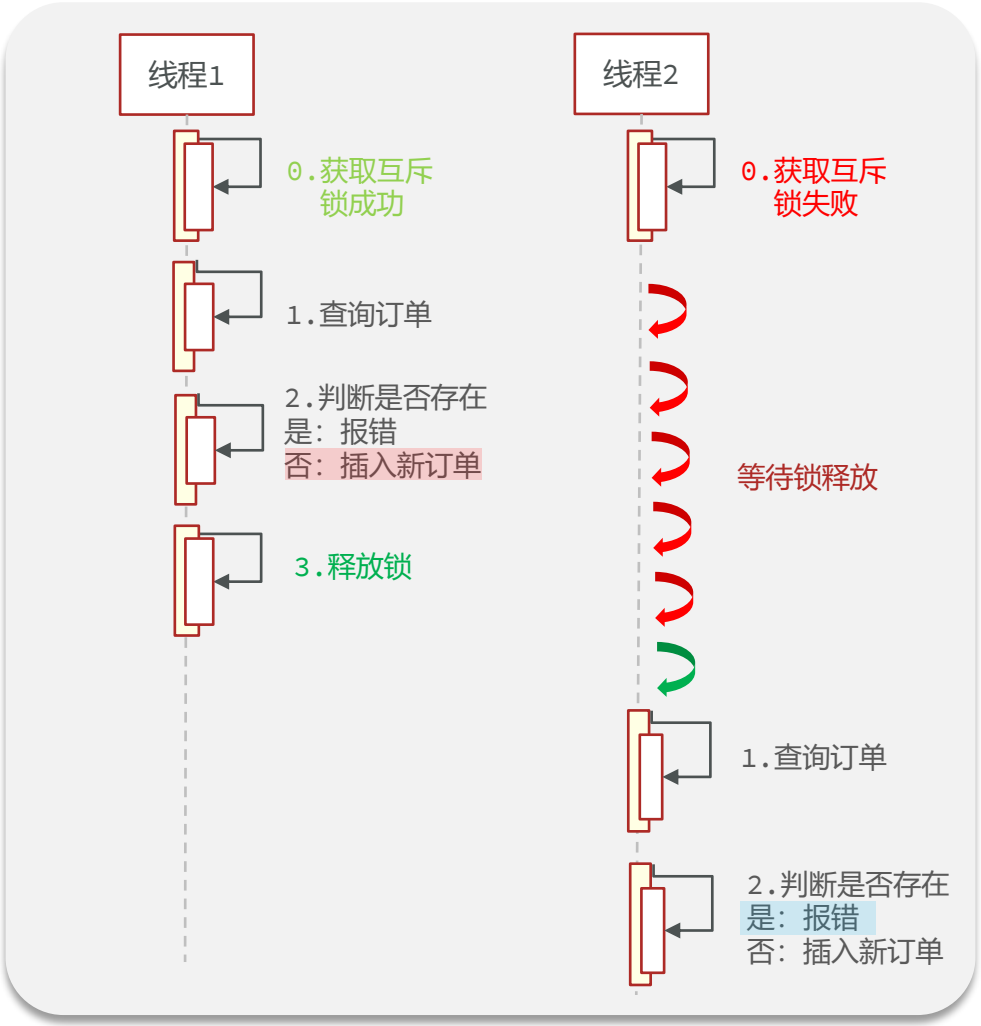
一人一单的并发安全问题



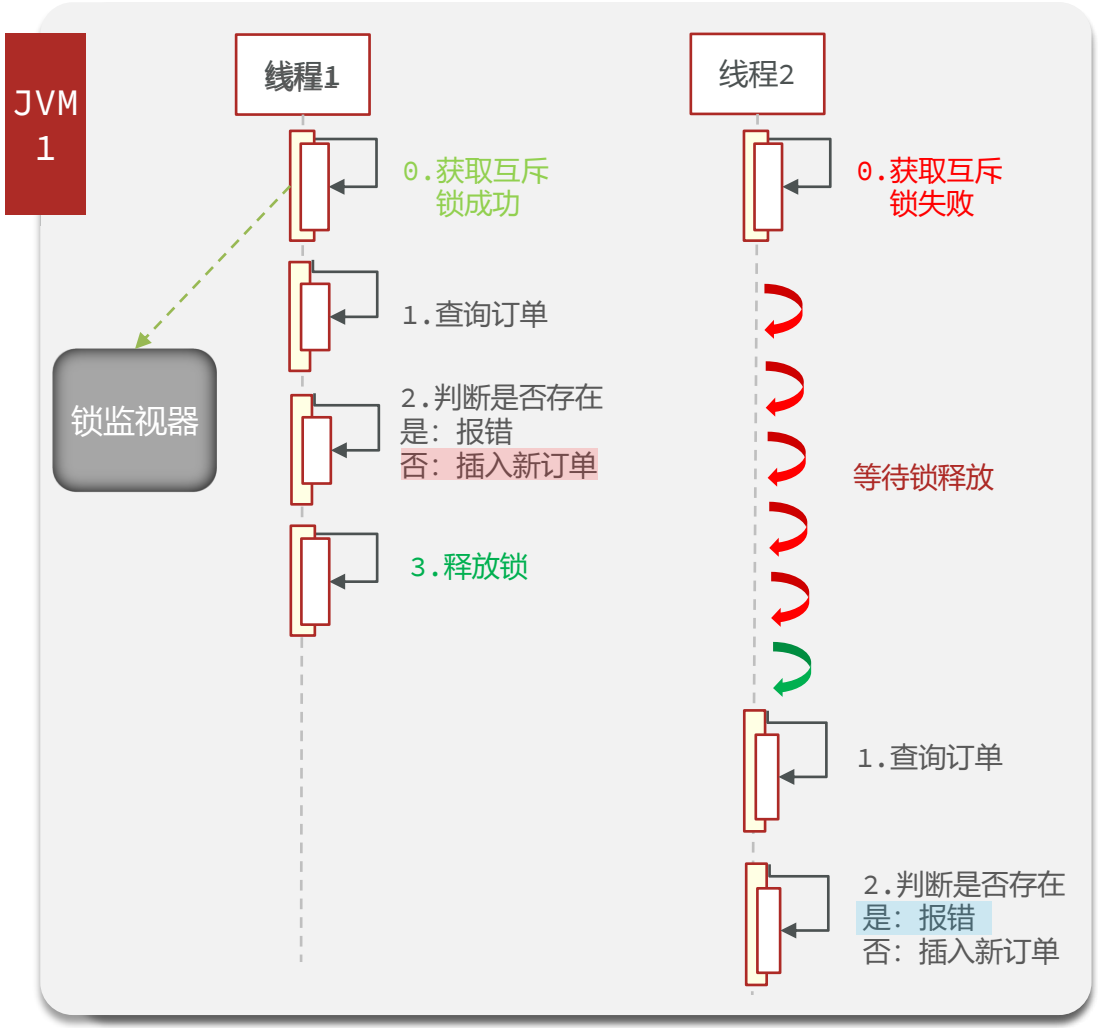
一人一单的并发安全问题



一人一单的并发安全问题



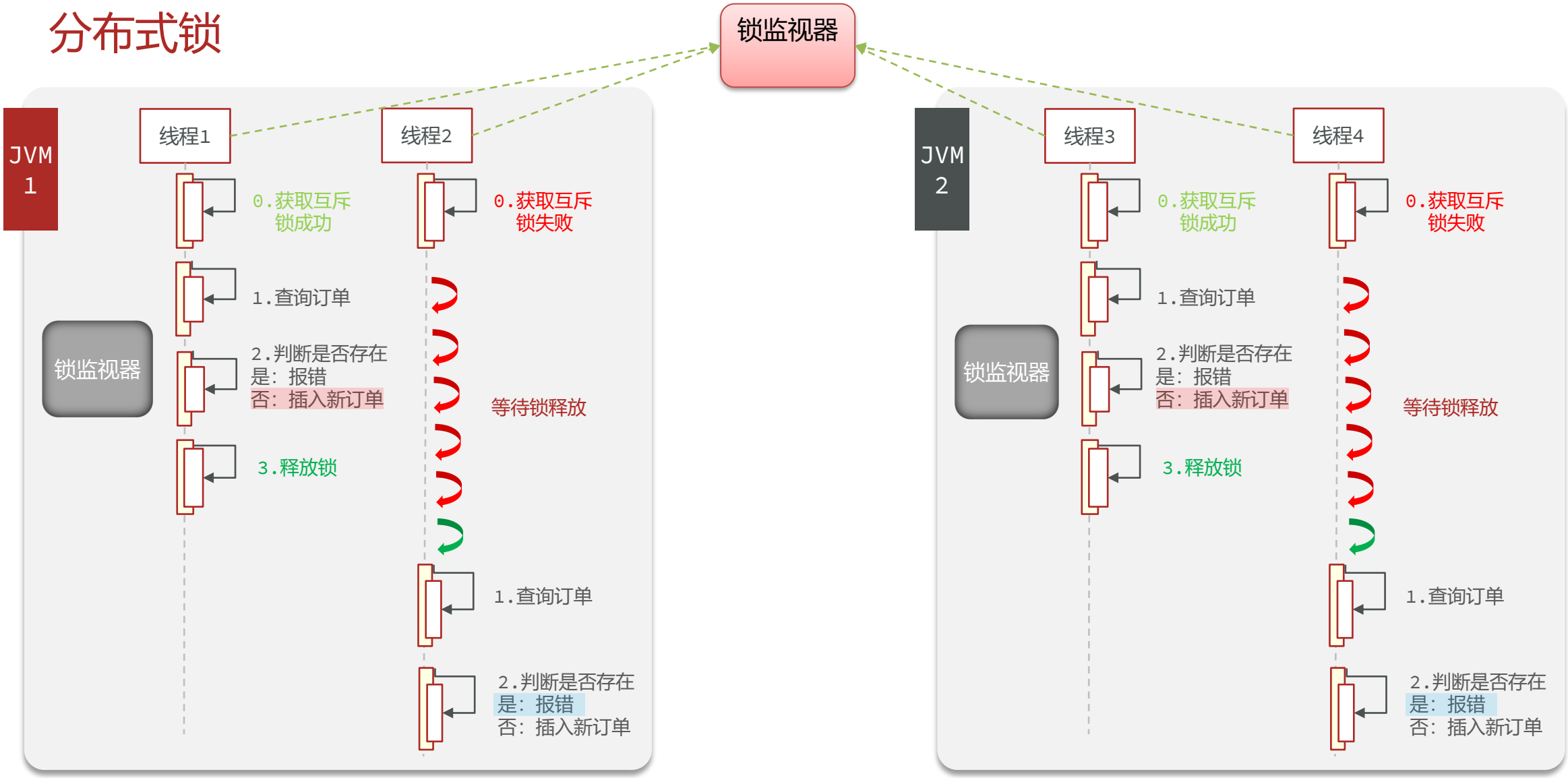
一人一单的并发安全问题



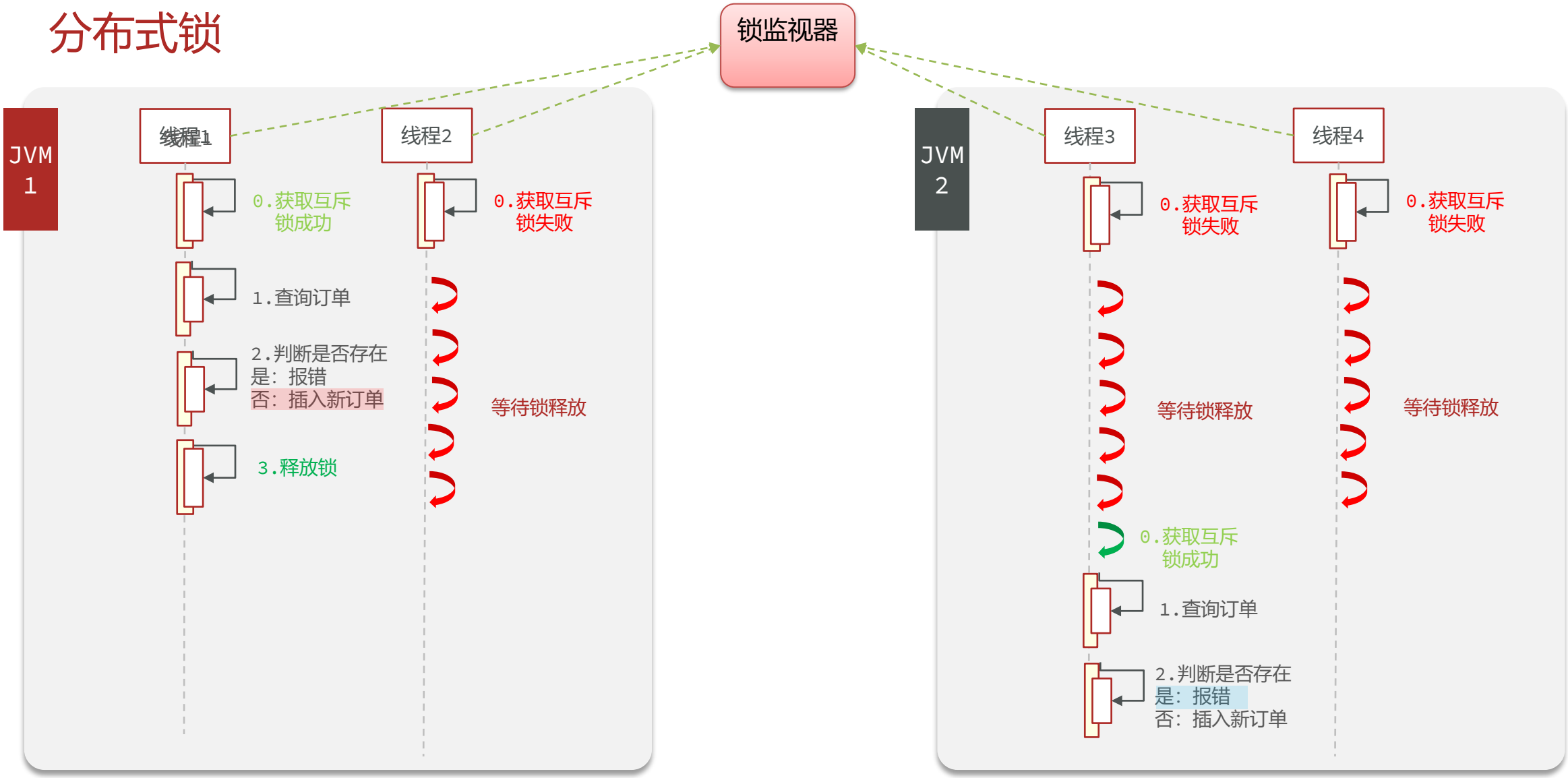
目录

Contents

分布式锁

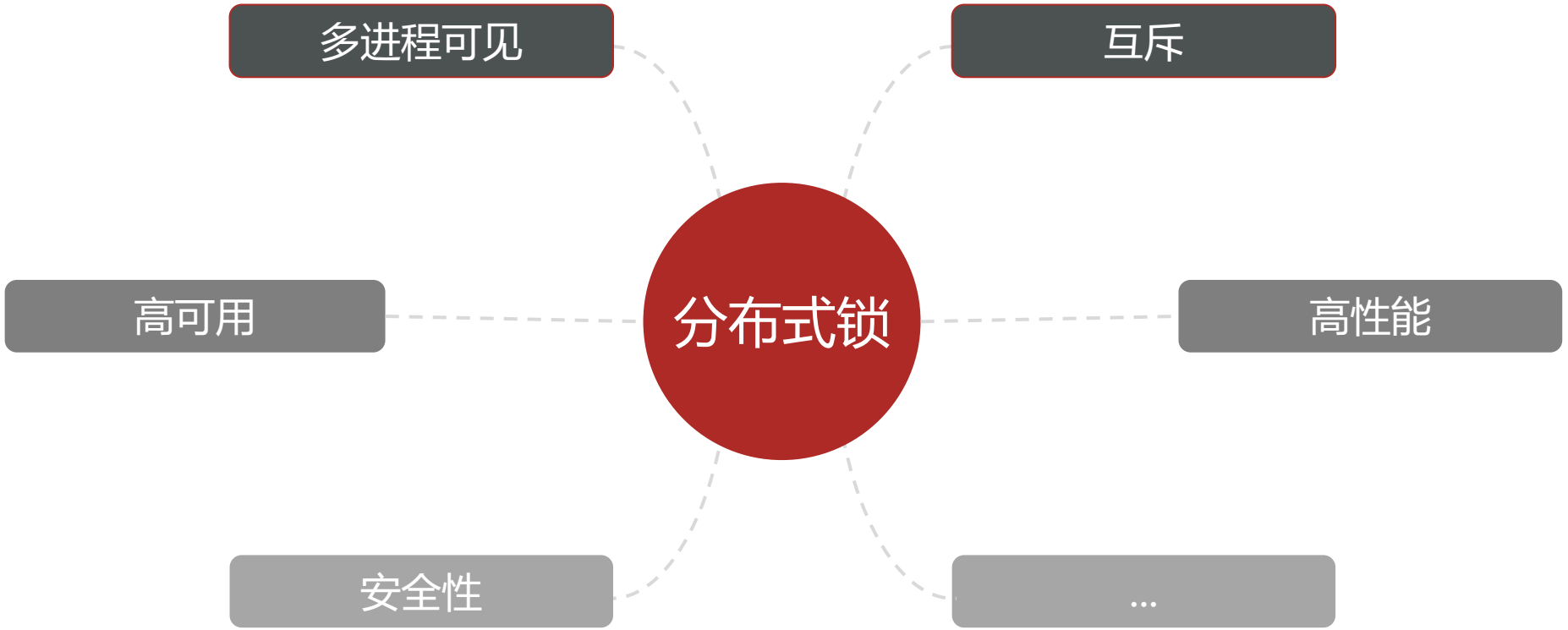

锁监视器
JVM 1
线程1
线程2
0.获取互斥锁成功
0.获取互斥锁失败
1.查询订单
2.判断是否存在
是：报错
否：插入新订单
锁监视器
等待锁释放
3.释放锁
1.查询订单
2.判断是否存在
是：报错
否：插入新订单
JVM 2
线程3
线程4
0.获取互斥锁成功
0.获取互斥锁失败
1.查询订单
2.判断是否存在
是：报错
否：插入新订单
锁监视器
等待锁释放
3.释放锁
1.查询订单
2.判断是否存在
是：报错
否：插入新订单

分布式锁


锁监视器
JVM 1
线程1
线程2
0.获取互斥锁成功
0.获取互斥锁失败
1.查询订单
2.判断是否存在
是：报错
否：插入新订单
3.释放锁
等待锁释放
JVM 2
线程3
线程4
0.获取互斥锁失败
0.获取互斥锁失败
等待锁释放
等待锁释放
0.获取互斥锁成功
1.查询订单
2.判断是否存在
是：报错
否：插入新订单

什么是分布式锁

分布式锁：满足分布式系统或集群模式下多进程可见并且互斥的锁。



分布式锁的实现

分布式锁的核心是实现多进程之间互斥，而满足这一点的方式有很多，常见的有三种：

	MySQL	Redis	Zookeeper
互斥	利用mysql本身的互斥锁机制	利用setnx这样的互斥命令	利用节点的唯一性和有序性实现互斥
高可用	好	好	好
高性能	一般	好	一般
安全性	断开连接，自动释放锁	利用锁超时时间，到期释放	临时节点，断开连接自动释放

基于Redis的分布式锁

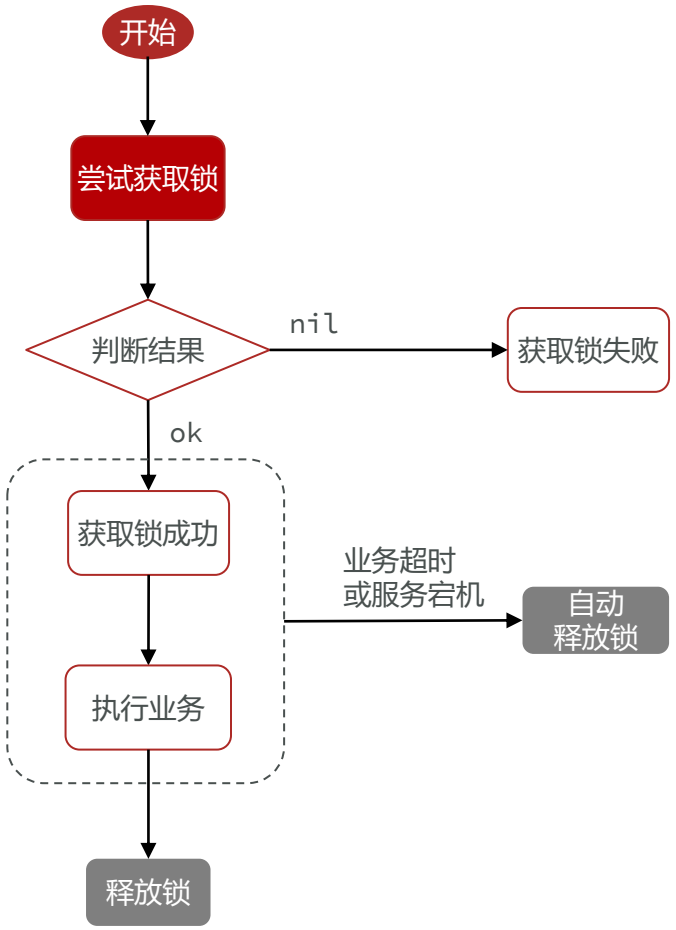
实现分布式锁时需要实现的两个基本方法：

- 获取锁：
  - 互斥：确保只能有一个线程获取锁
  -

```
# 添加锁，利用setnx的互斥特性
SETNX lock thread1

# 添加锁过期时间，避免服务宕机引起的死锁
EXPIRE lock 10
```

- 释放锁：
  - 手动释放
  - 超时释放：获取锁时添加一个超时时间

```
# 释放锁，删除即可
DEL key
```

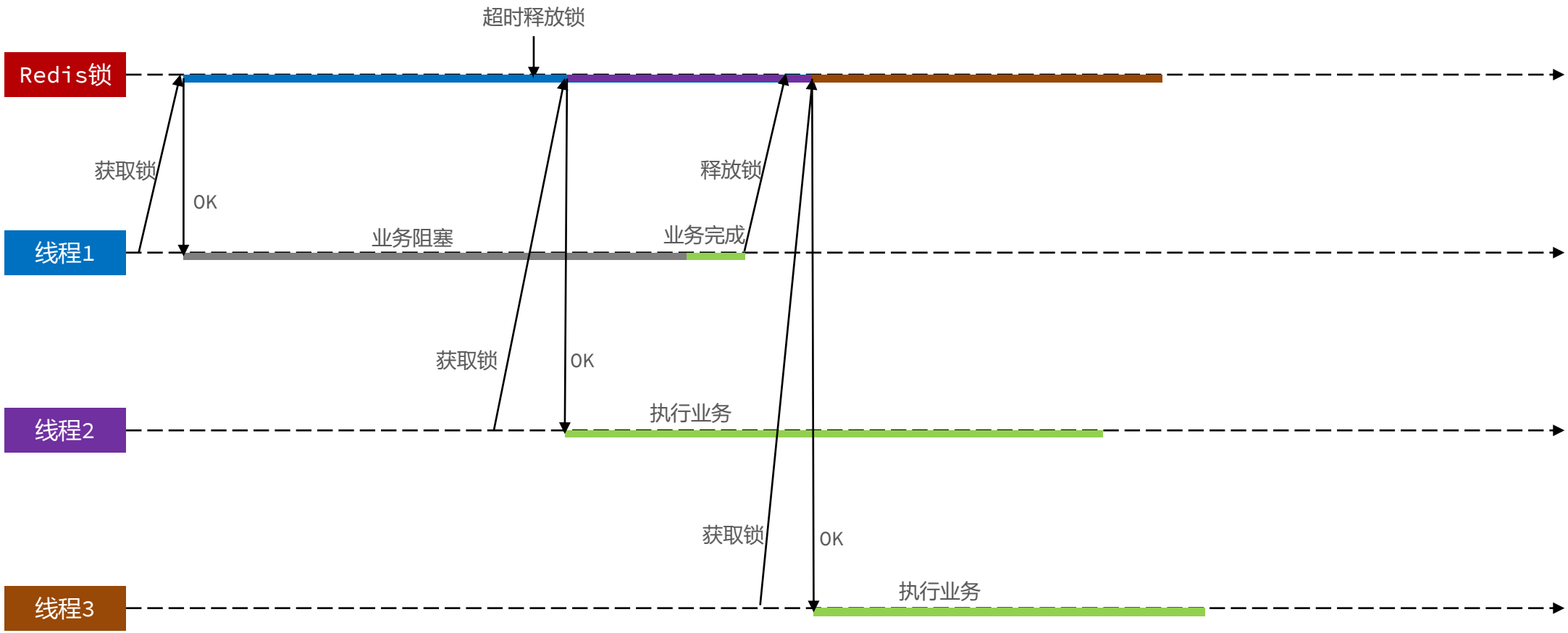


案例 基于Redis实现分布式锁初级版本

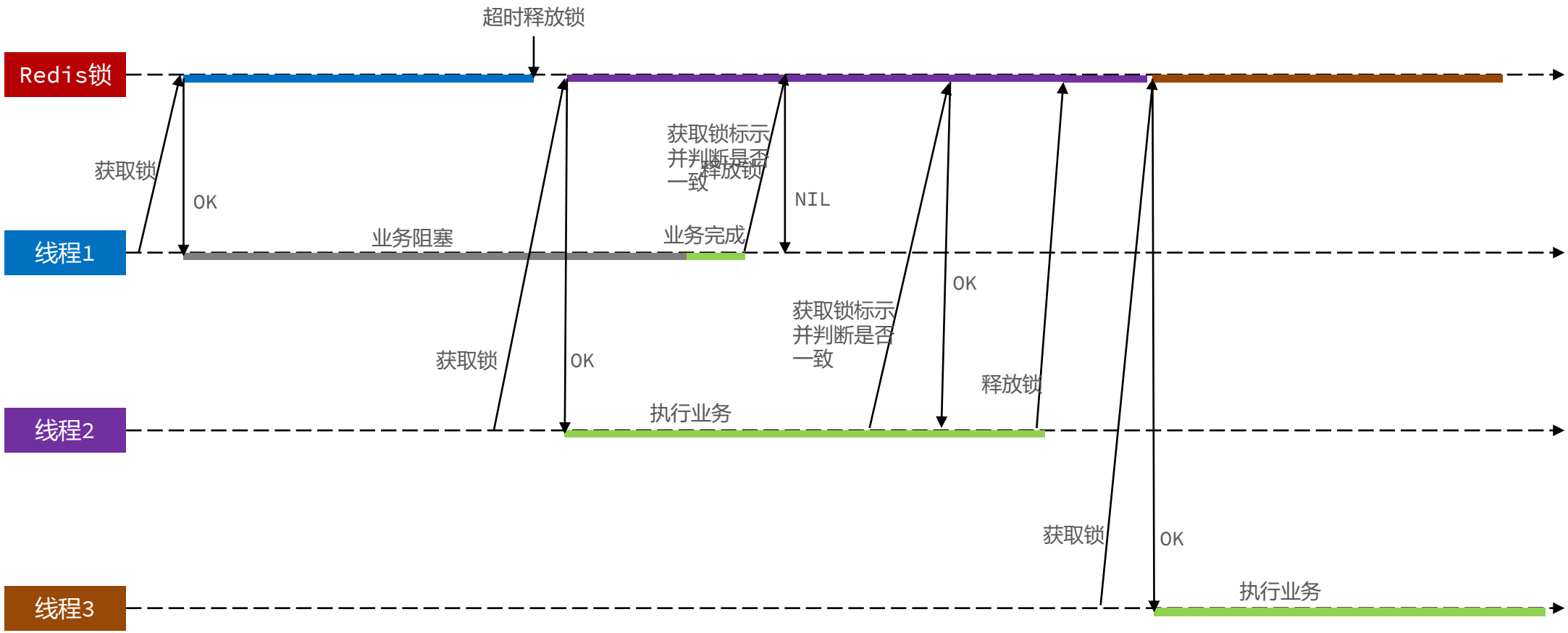
需求：定义一个类，实现下面接口，利用Redis实现分布式锁功能。

```
public interface ILock {

    /**
     * 尝试获取锁
     * @param timeoutSec 锁持有的超时时间，过期后自动释放
     * @return true代表获取锁成功; false代表获取锁失败
     */
    boolean tryLock(long timeoutSec);

    /**
     * 释放锁
     */
    void unlock();
}
```

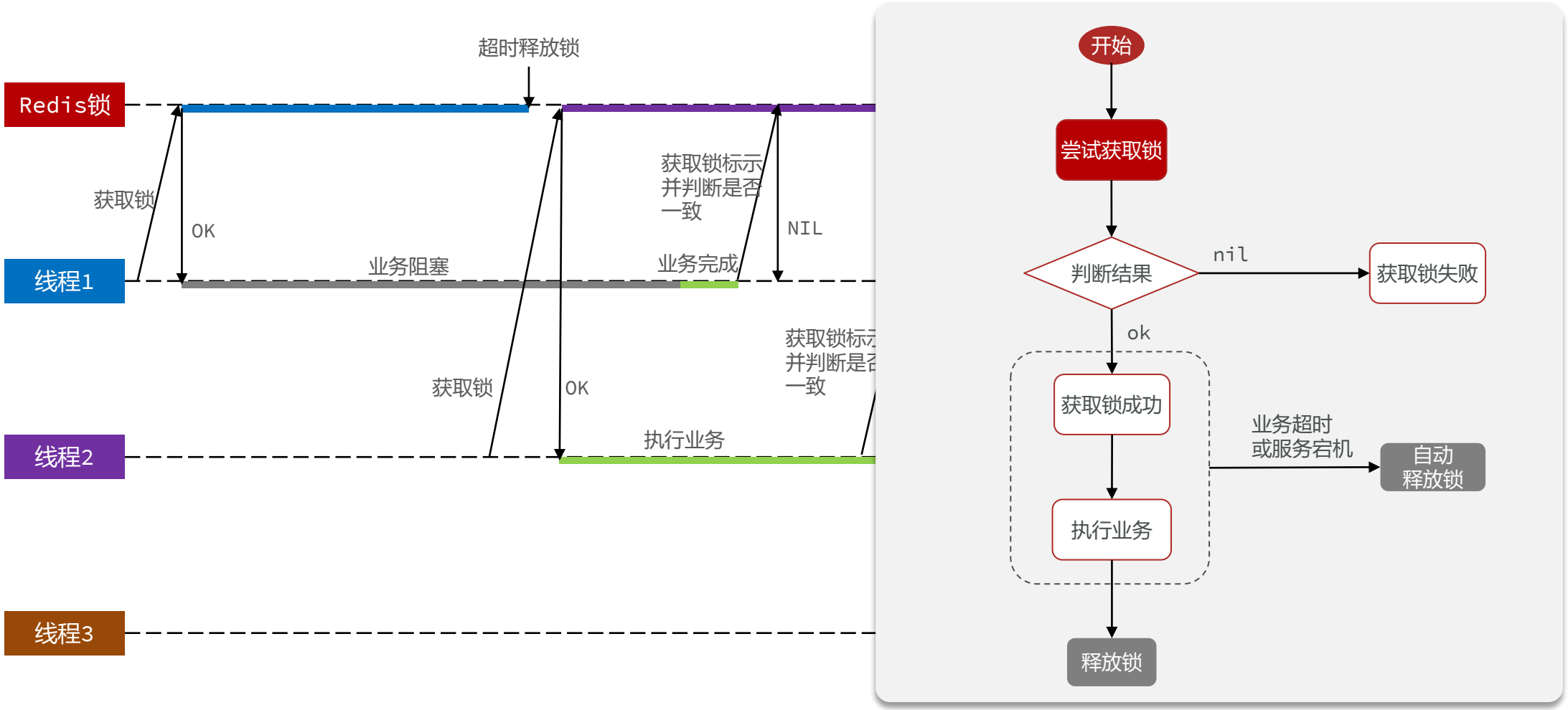
基于Redis的分布式锁



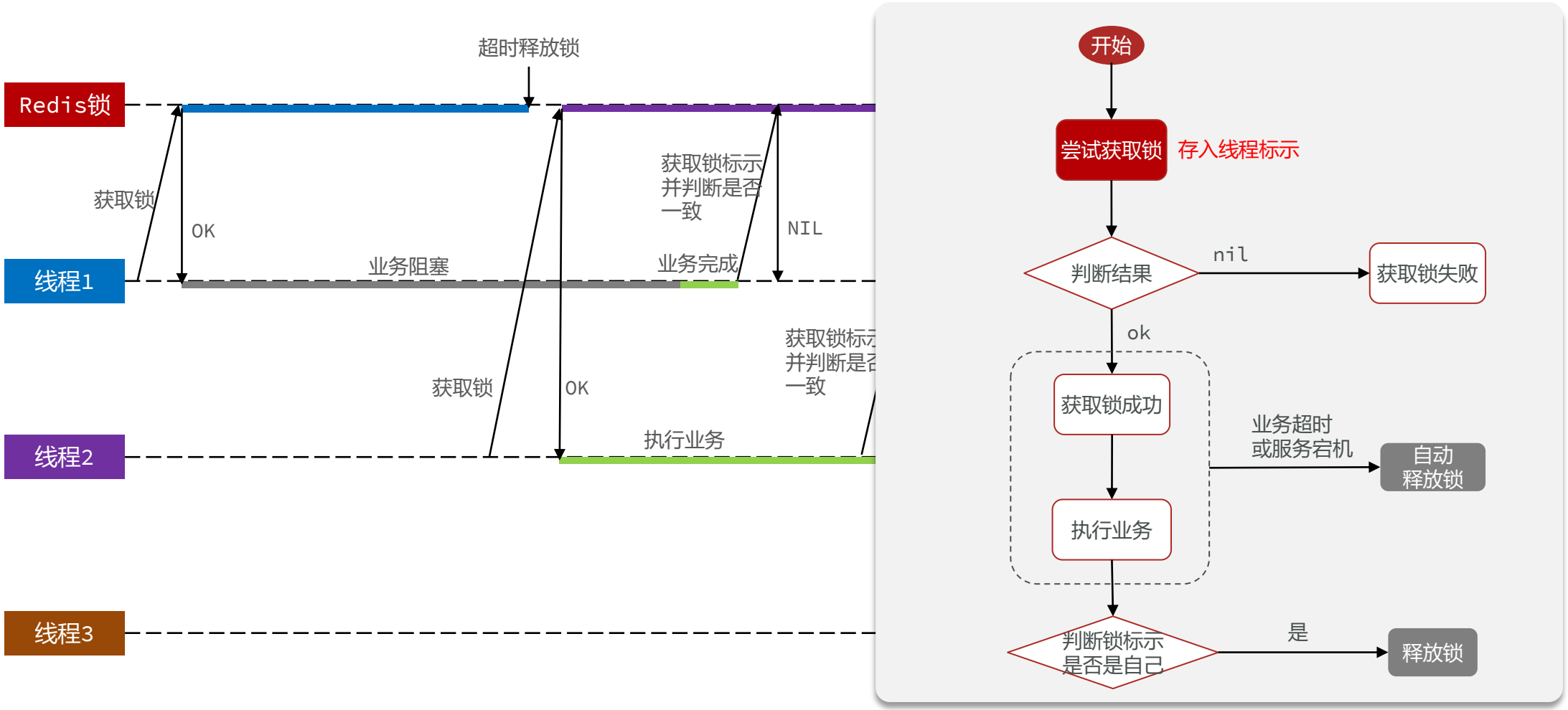
基于Redis的分布式锁



基于Redis的分布式锁



基于Redis的分布式锁



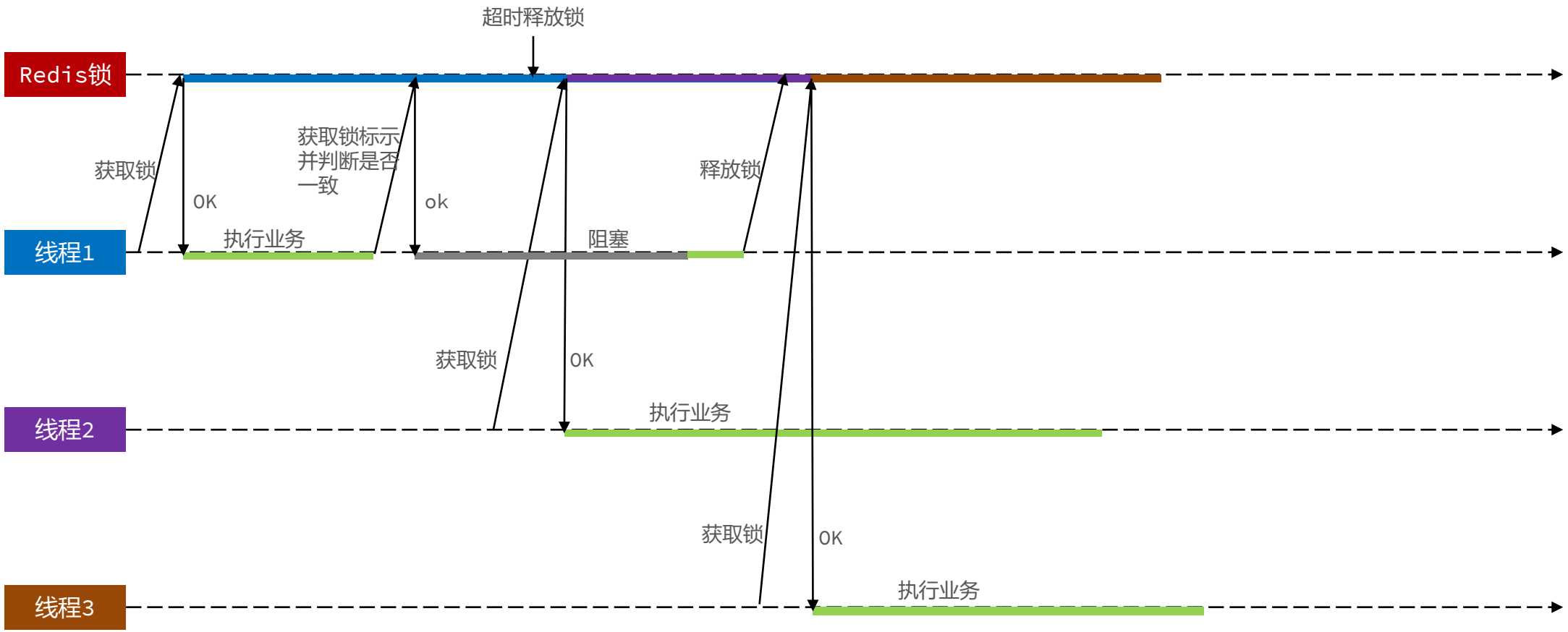
案例

改进Redis的分布式锁

需求：修改之前的分布式锁实现，满足：

1. 在获取锁时存入线程标示（可以用UUID表示）
2. 在释放锁时先获取锁中的线程标示，判断是否与当前线程标示一致
   - 如果一致则释放锁
   - 如果不一致则不释放锁

基于Redis的分布式锁



Redis的Lua脚本

Redis提供了Lua脚本功能，在一个脚本中编写多条Redis命令，确保多条命令执行时的原子性。Lua是一种编程语言，它的基本语法大家可以参考网站：https://www.runoob.com/lua/lua-tutorial.html

这里重点介绍Redis提供的调用函数，语法如下：

```
# 执行redis命令
redis.call('命令名称', 'key', '其它参数', ...)
```

例如，我们要执行set name jack，则脚本是这样：

```
# 执行 set name jack
redis.call('set', 'name', 'jack')
```

例如，我们要先执行set name Rose，再执行get name，则脚本如下：

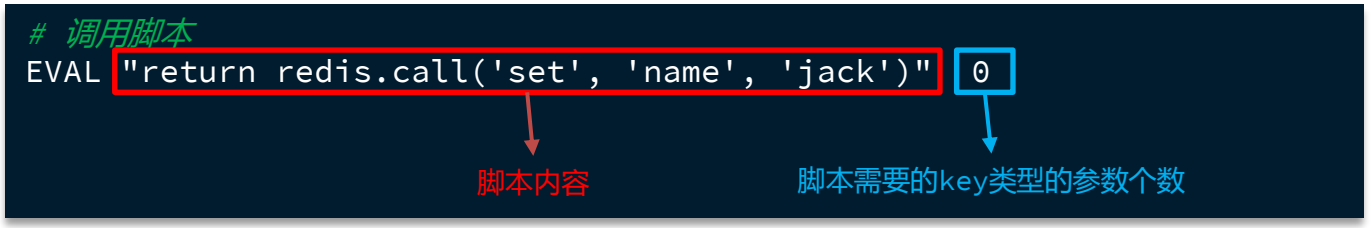
```
# 先执行 set name jack
redis.call('set', 'name', 'jack')
# 再执行 get name
local name = redis.call('get', 'name')
# 返回
return name
```

Redis的Lua脚本

写好脚本以后，需要用Redis命令来调用脚本，调用脚本的常见命令如下：

```
127.0.0.1:6379> help @scripting

  EVAL script numkeys key [key ...] arg [arg ...]
  summary: Execute a Lua script server side
  since: 2.6.0
```

例如，我们要执行 redis.call('set', 'name', 'jack') 这个脚本，语法如下：



如果脚本中的key、value不想写死，可以作为参数传递。key类型参数会放入KEYS数组，其它参数会放入ARGV数组，在脚本中可以从KEYS和ARGV数组获取这些参数：

基于Redis的分布式锁

释放锁的业务流程是这样的：

1. 获取锁中的线程标示
2. 判断是否与指定的标示（当前线程标示）一致
3. 如果一致则释放锁（删除）
4. 如果不一致则什么都不做

如果用Lua脚本来表示则是这样的：

```
-- 这里的 KEYS[1] 就是锁的key，这里的ARGV[1] 就是当前线程标示
-- 获取锁中的标示，判断是否与当前线程标示一致
if (redis.call('GET', KEYS[1]) == ARGV[1]) then
  -- 一致，则删除锁
  return redis.call('DEL', KEYS[1])
end
-- 不一致，则直接返回
return 0
```

案例 再次改进Redis的分布式锁

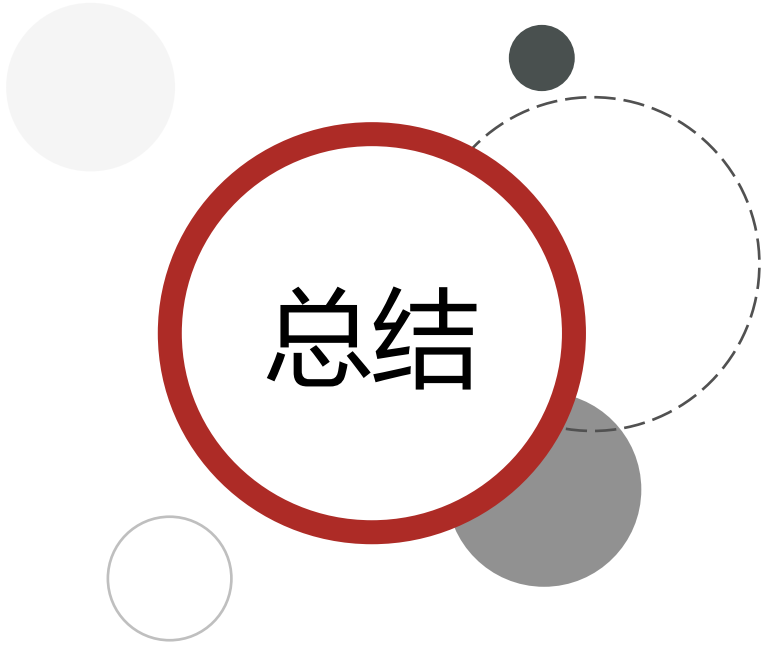
需求：基于Lua脚本实现分布式锁的释放锁逻辑

提示：RedisTemplate调用Lua脚本的API如下：

RedisTemplate.java

```
/*
 * (non-Javadoc)
 * @see org.springframework.data.redis.core.RedisOperations#execute(org.spring
 */
@Override
public <T> T execute(RedisScript<T> script, List<K> keys, Object... args) {
    return scriptExecutor.execute(script, keys, args);
}
```

```
127.0.0.1:6379> help @scripting

  EVAL script numkeys key [key ...] arg [arg ...]
  summary: Execute a Lua script server side
  since: 2.6.0
```



基于Redis的分布式锁实现思路：

- 利用set nx ex获取锁，并设置过期时间，保存线程标示
- 释放锁时先判断线程标示是否与自己一致，一致则删除锁

特性：

- 利用set nx满足互斥性
- 利用set ex保证故障时锁依然能释放，避免死锁，提高安全性
- 利用Redis集群保证高可用和高并发特性

基于Redis的分布式锁优化

基于setnx实现的分布式锁存在下面的问题：

01

不可重入

同一个线程无法多次获取同一把锁

02

不可重试

获取锁只尝试一次就返回false，没有重试机制

03

超时释放

锁超时释放虽然可以避免死锁，但如果是业务执行耗时较长，也会导致锁释放，存在安全隐患

04

主从一致性

如果Redis提供了主从集群，主从同步存在延迟，当主宕机时，如果从并同步主中的锁数据，则会出现锁实现

Redisson

Redisson是一个在Redis的基础上实现的Java驻内存数据网格（In-Memory Data Grid）。它不仅提供了一系列的分布式的Java常用对象，还提供了许多分布式服务，其中就包含了各种分布式锁的实现。

官网地址： https://redisson.org

GitHub地址： https://github.com/redisson/redisson

Redisson入门

1. 引入依赖：

```xml
<dependency>
    <groupId>org.redisson</groupId>
    <artifactId>redisson</artifactId>
    <version>3.13.6</version>
</dependency>
```

2. 配置Redisson客户端：

```java
@Configuration
public class RedisConfig {
    @Bean
    public RedissonClient redissonClient() {
        // 配置类
        Config config = new Config();
        // 添加redis地址，这里添加了单点的地址，也可以使用config.useClusterServers()添加集群地址
        config.useSingleServer().setAddress("redis://192.168.150.101:6379").setPassowrd("123321");
        // 创建客户端
        return Redisson.create(config);
    }
}
```

Redisson入门

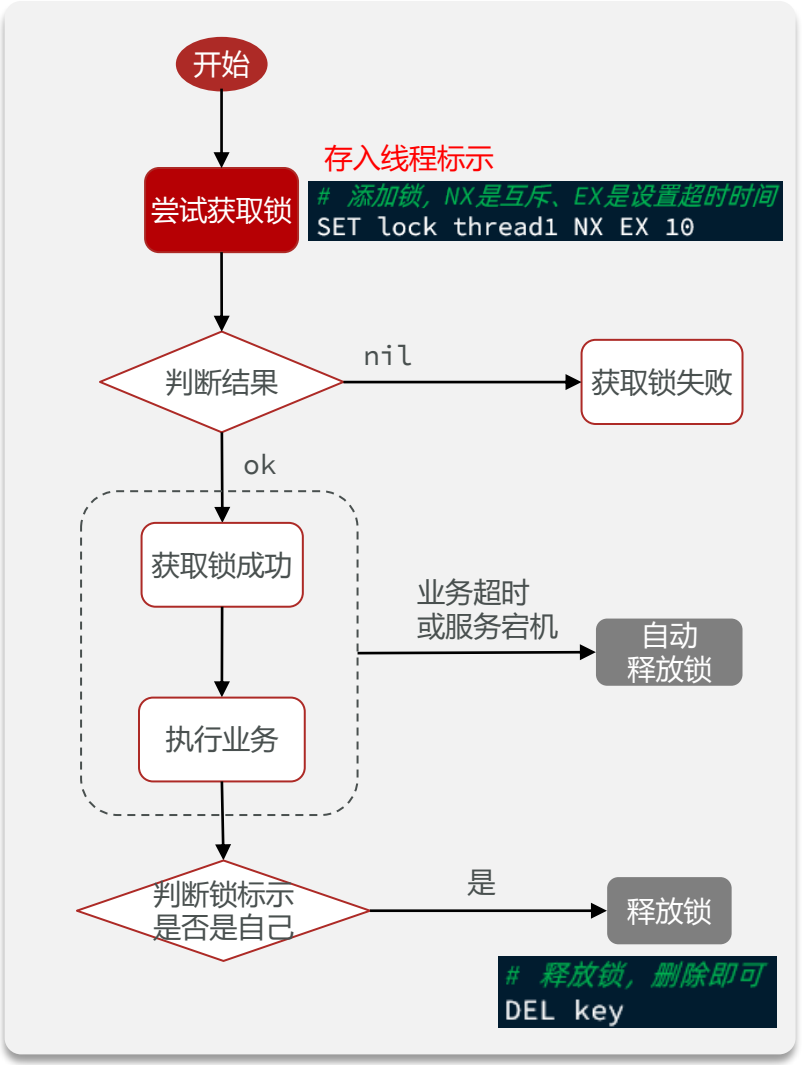
3. 使用Redisson的分布式锁

```
@Resource
private RedissonClient redissonClient;

@Test
void testRedisson() throws InterruptedException {
    // 获取锁（可重入），指定锁的名称
    RLock lock = redissonClient.getLock("anyLock");
    // 尝试获取锁，参数分别是：获取锁的最大等待时间（期间会重试），锁自动释放时间，时间单位
    boolean isLock = lock.tryLock(1, 10, TimeUnit.SECONDS);
    // 判断释放获取成功
    if(isLock){
        try {
            System.out.println("执行业务");
        }finally {
            // 释放锁
            lock.unlock();
        }
    }
}
```

Redisson可重入锁原理

```java
// 创建锁对象
RLock lock = redissonClient.getLock("lock");

@Test
void method1() {
    boolean isLock = lock.tryLock();
    if(!isLock){
        log.error("获取锁失败，1");
        return;
    }
    try {
        log.info("获取锁成功，1");
        method2();
    } finally {
        log.info("释放锁，1");
        lock.unlock();
    }
}
void method2(){
    boolean isLock = lock.tryLock();
    if(!isLock){
        log.error("获取锁失败，2");
        return;
    }
    try {
        log.info("获取锁成功，2");
    } finally {
        log.info("释放锁，2");
        lock.unlock();
    }
}
```

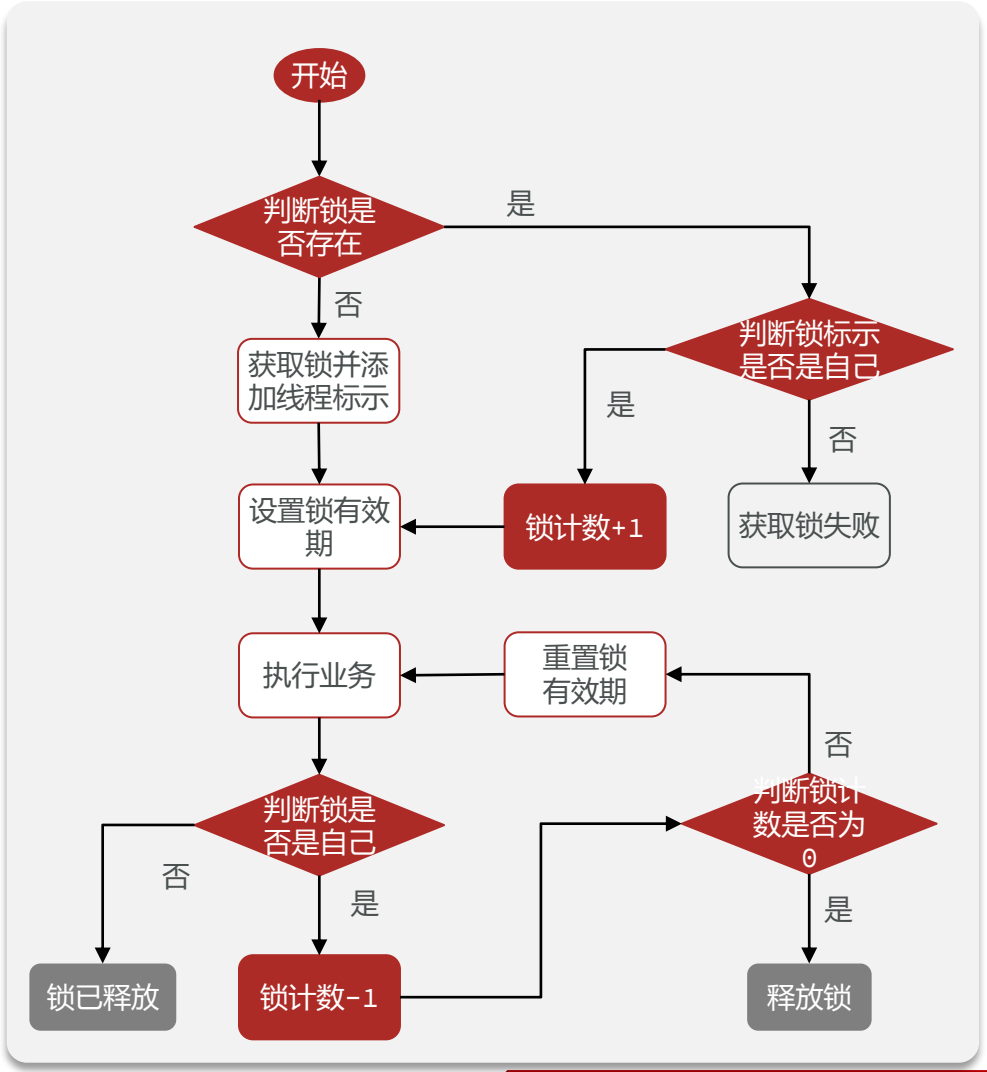
KEY	VALUE
lock	thread1



Redisson可重入锁原理

```
// 创建锁对象
RLock lock = redissonClient.getLock("lock");

@Test
void method1() {
    boolean isLock = lock.tryLock();
    if(!isLock){
        log.error("获取锁失败, 1");
        return;
    }
    try {
        log.info("获取锁成功, 1");
        method2();
    } finally {
        log.info("释放锁, 1");
        lock.unlock();
    }
}
void method2(){
    boolean isLock = lock.tryLock();
    if(!isLock){
        log.error("获取锁失败, 2");
        return;
    }
    try {
        log.info("获取锁成功, 2");
    } finally {
        log.info("释放锁, 2");
        lock.unlock();
    }
}
```

KEY	VALUE	
	field	value
lock	thread1	2

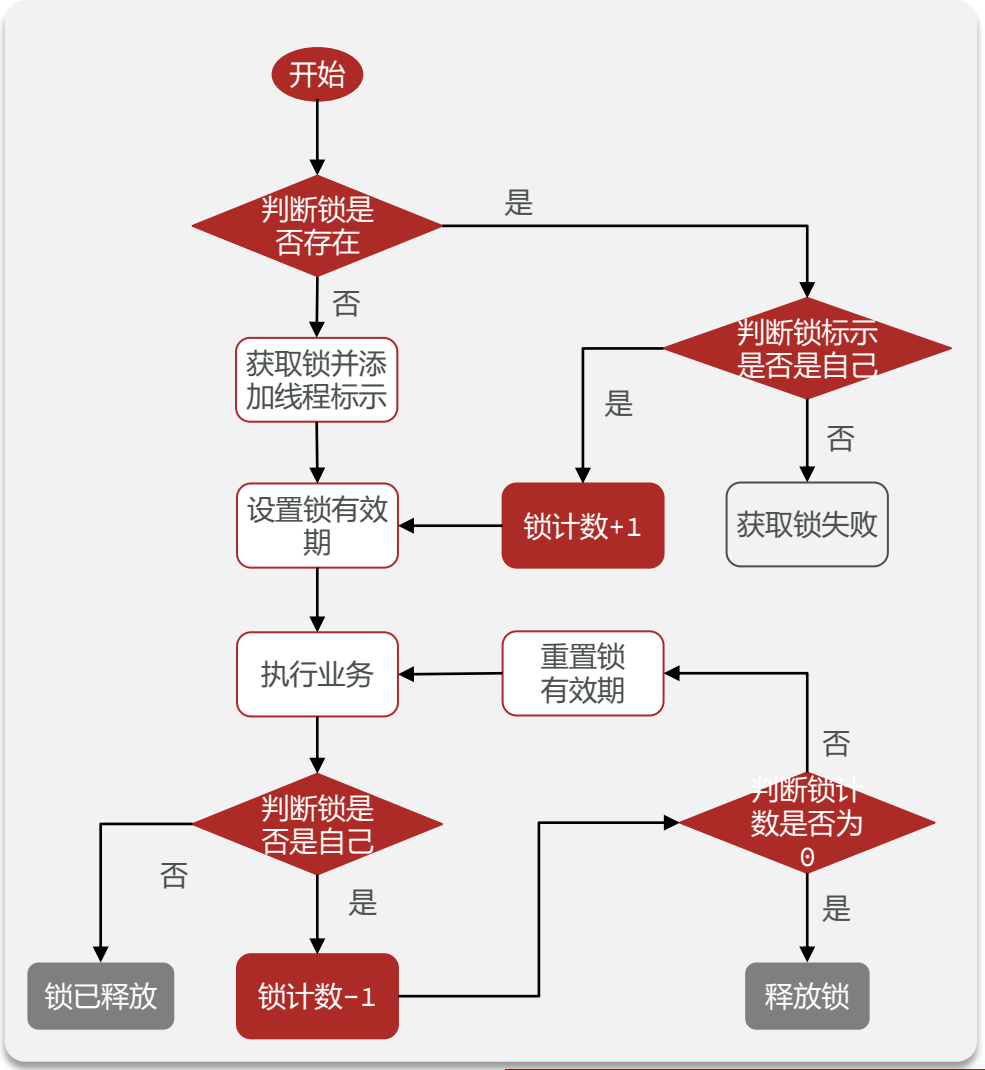


Redisson可重入锁原理

KEY	VALUE	
	field	value
lock	thread1	0

获取锁的Lua脚本：

```lua
local key = KEYS[1]; -- 锁的key
local threadId = ARGV[1]; -- 线程唯一标识
local releaseTime = ARGV[2]; -- 锁的自动释放时间
-- 判断是否存在
if(redis.call('exists', key) == 0) then
    -- 不存在，获取锁
    redis.call('hset', key, threadId, '1');
    -- 设置有效期
    redis.call('expire', key, releaseTime);
    return 1; -- 返回结果
end;
-- 锁已经存在，判断threadId是否是自己
if(redis.call('hexists', key, threadId) == 1) then
    -- 不存在，获取锁，重入次数+1
    redis.call('hincrby', key, threadId, '1');
    -- 设置有效期
    redis.call('expire', key, releaseTime);
    return 1; -- 返回结果
end;
return 0; -- 代码走到这里,说明获取锁的不是自己，获取锁失败
```

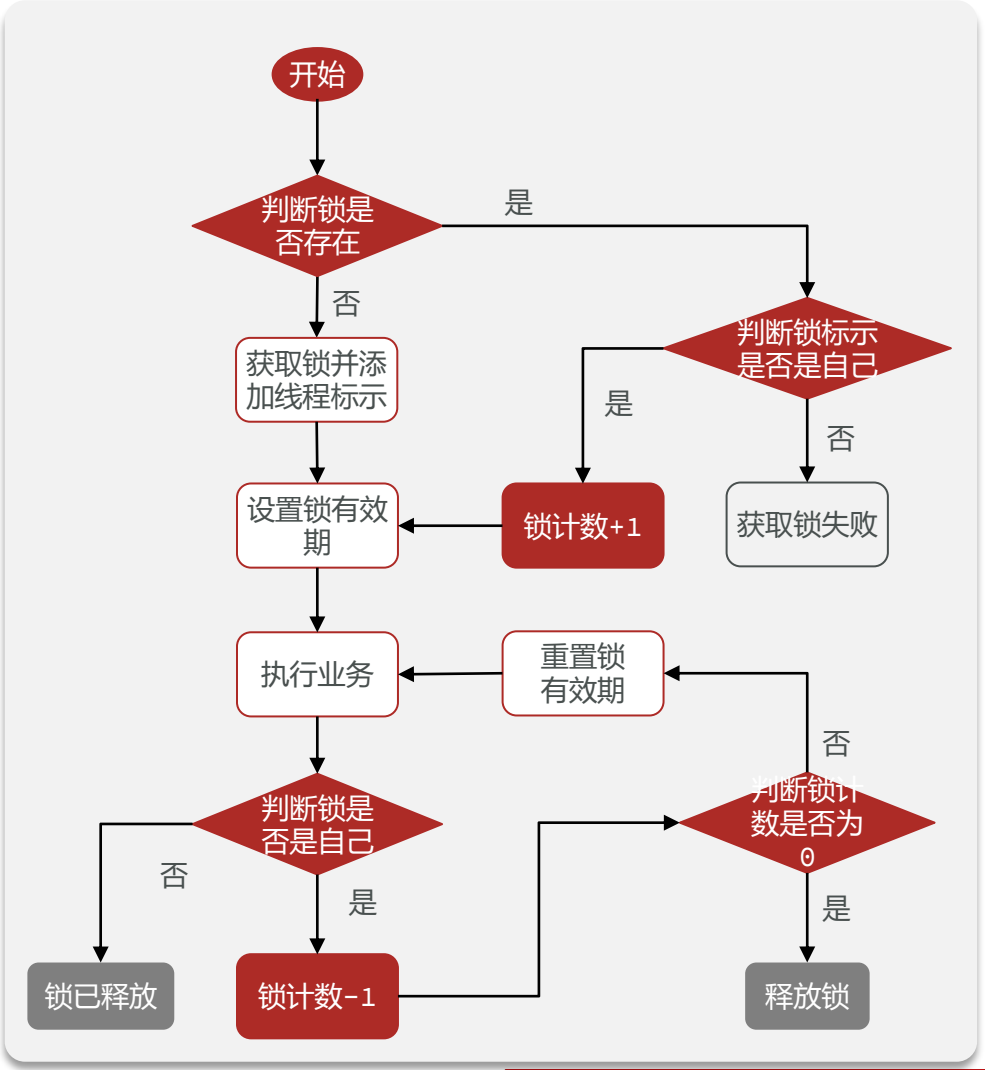


Redisson可重入锁原理

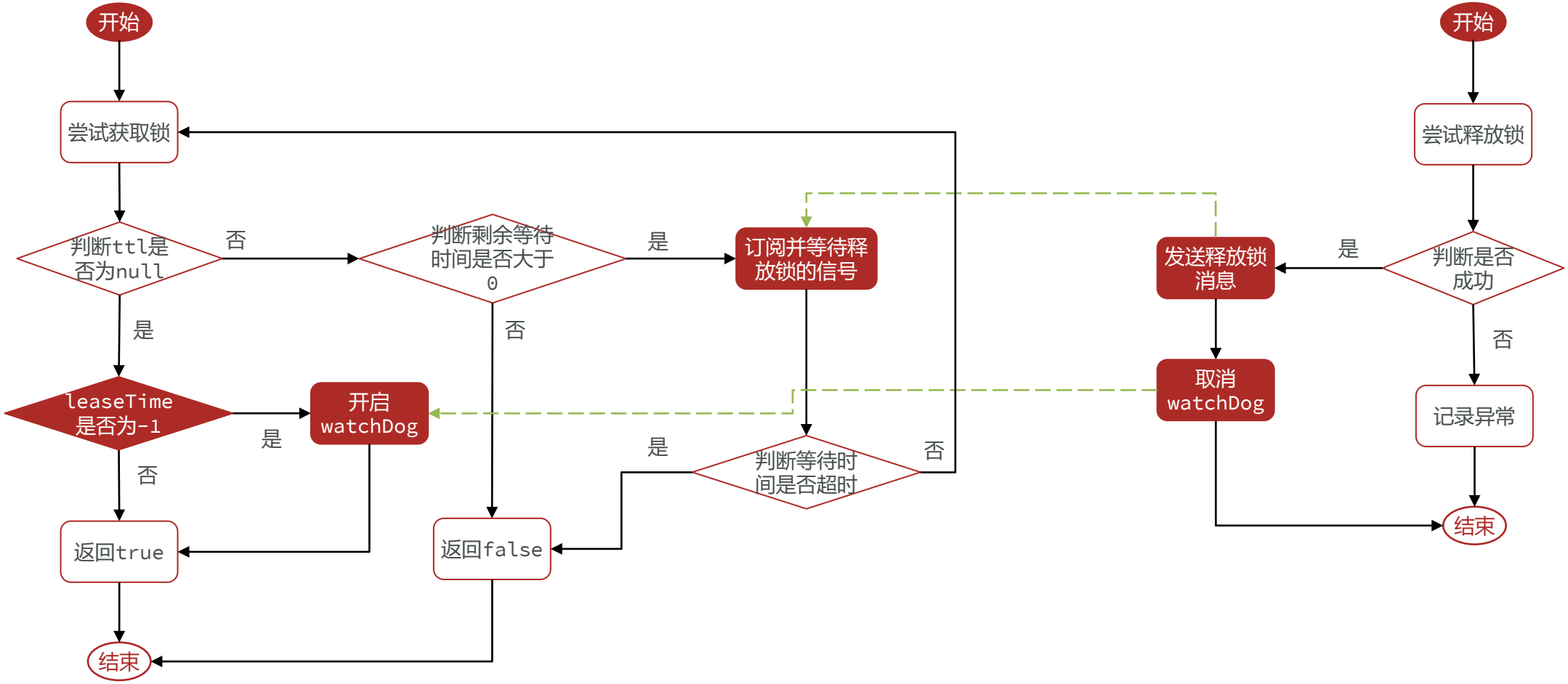
KEY	VALUE	
	field	value
lock:order	thread1	0

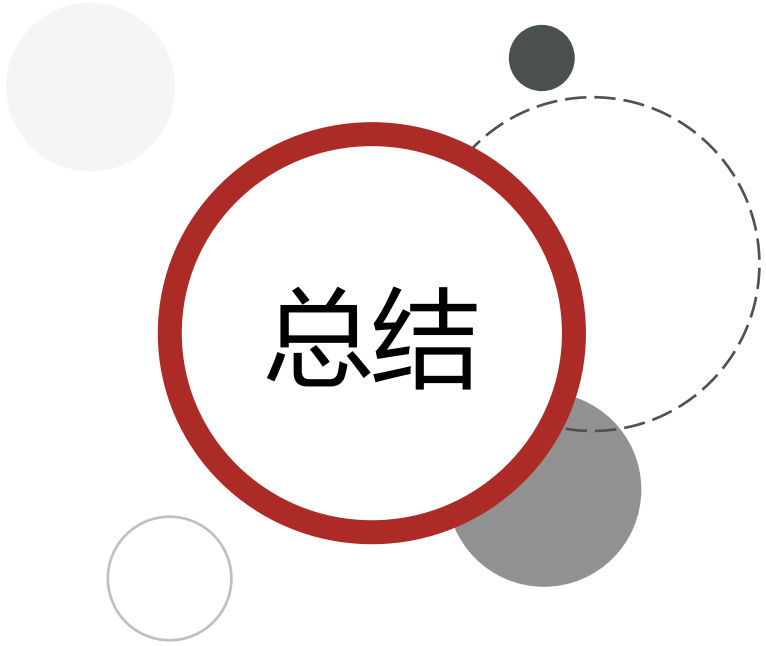
释放锁的Lua脚本：

```lua
local key = KEYS[1]; -- 锁的key
local threadId = ARGV[1]; -- 线程唯一标识
local releaseTime = ARGV[2]; -- 锁的自动释放时间
-- 判断当前锁是否还是被自己持有
if (redis.call('HEXISTS', key, threadId) == 0) then
    return nil; -- 如果已经不是自己，则直接返回
end;
-- 是自己的锁，则重入次数-1
local count = redis.call('HINCRBY', key, threadId, -1);
-- 判断是否重入次数是否已经为0
if (count > 0) then
    -- 大于0说明不能释放锁，重置有效期然后返回
    redis.call('EXPIRE', key, releaseTime);
    return nil;
else  -- 等于0说明可以释放锁，直接删除
    redis.call('DEL', key);
    return nil;
end;
```



Redisson分布式锁原理

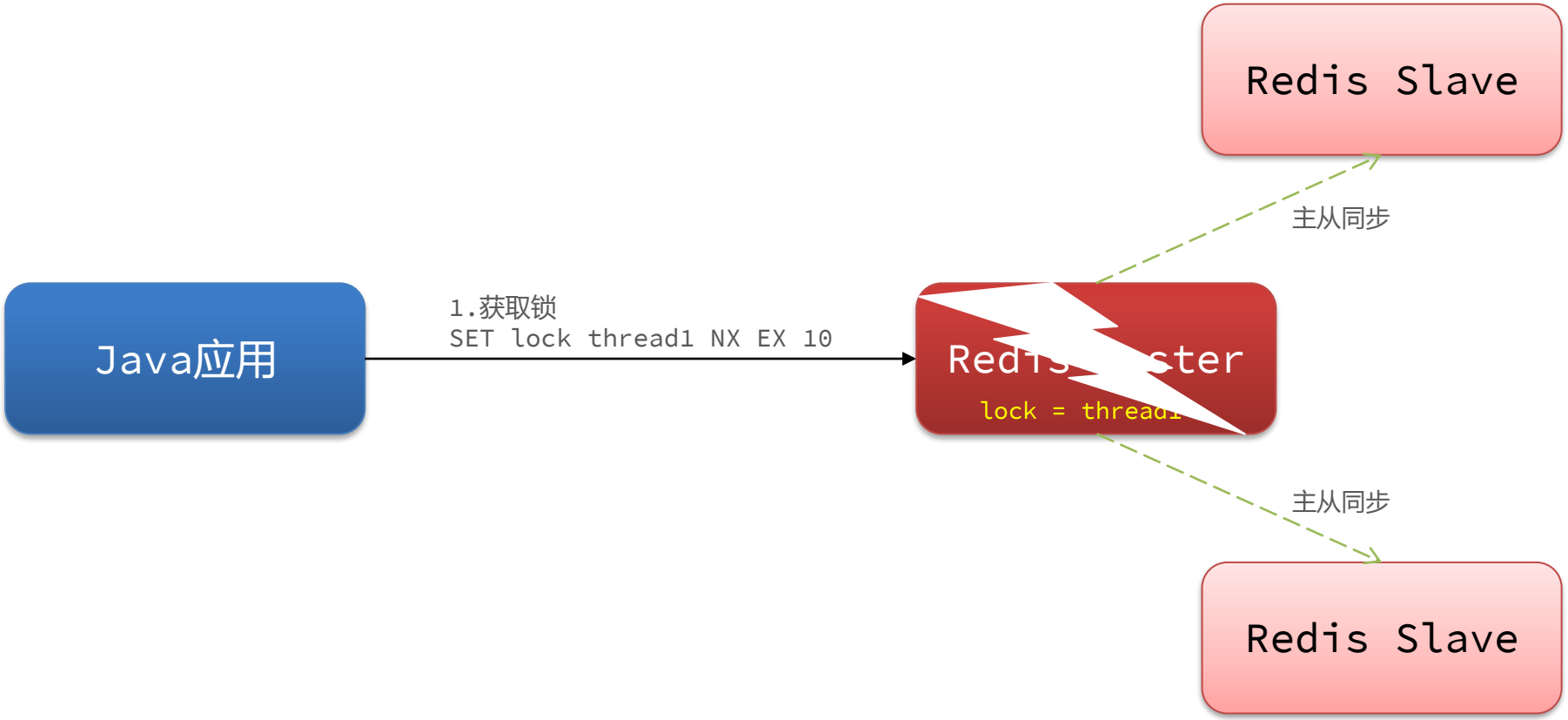




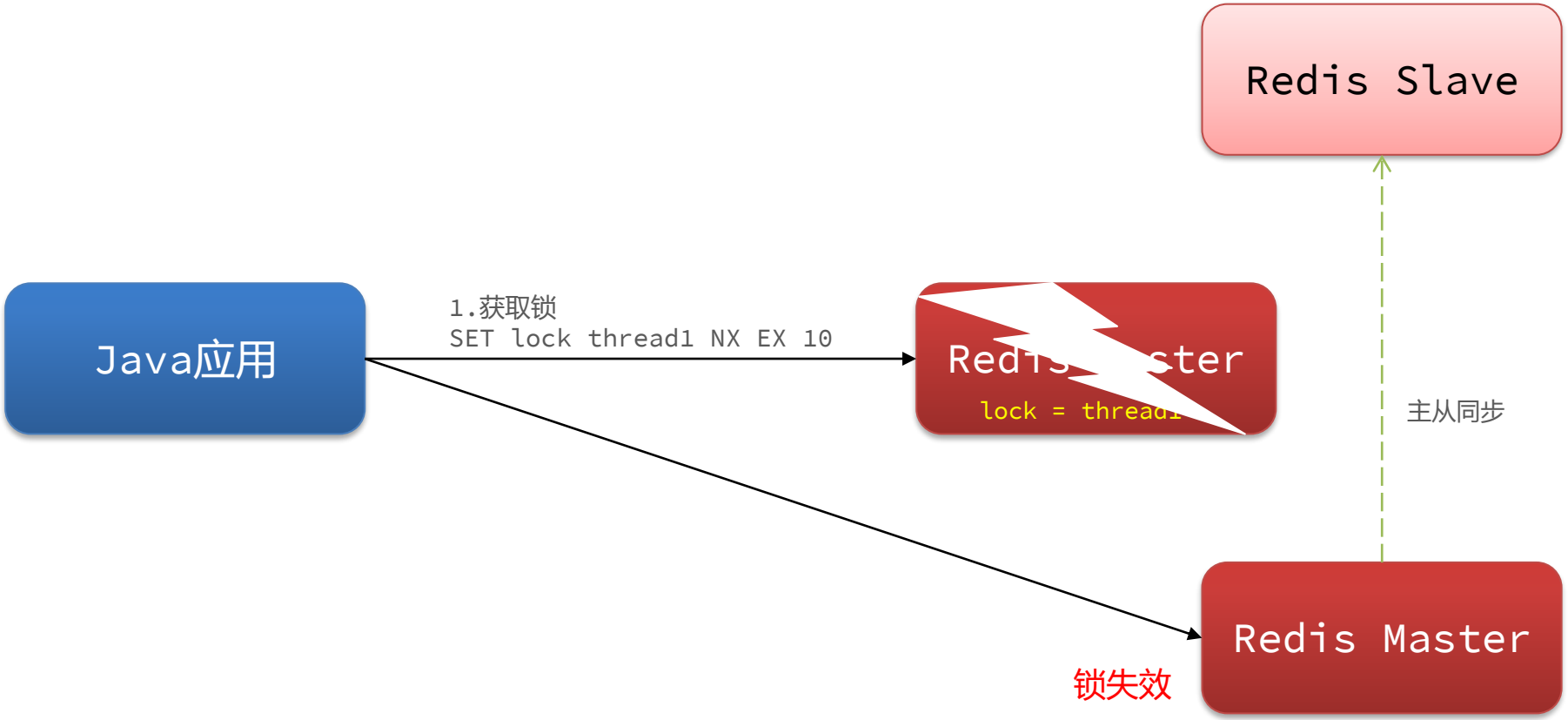
Redisson分布式锁原理：

- **可重入**：利用hash结构记录线程id和重入次数
- **可重试**：利用信号量和PubSub功能实现等待、唤醒，获取锁失败的重试机制
- **超时续约**：利用watchDog，每隔一段时间（releaseTime / 3），重置超时时间

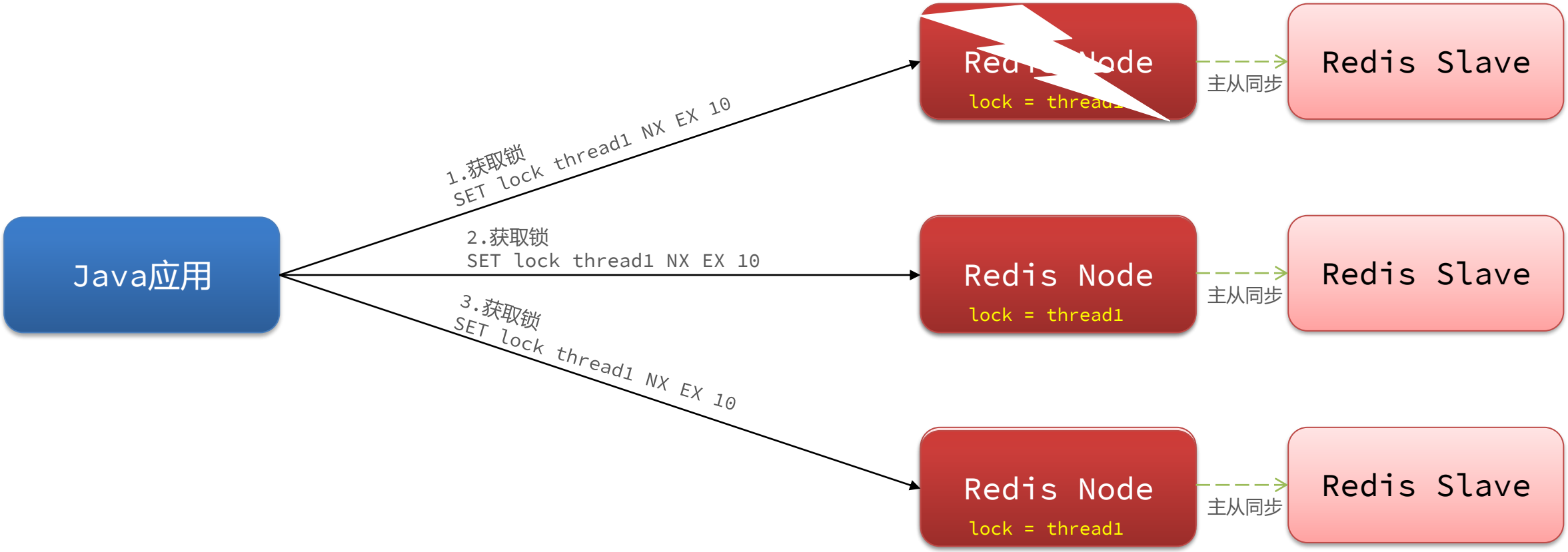
Redisson分布式锁主从一致性问题



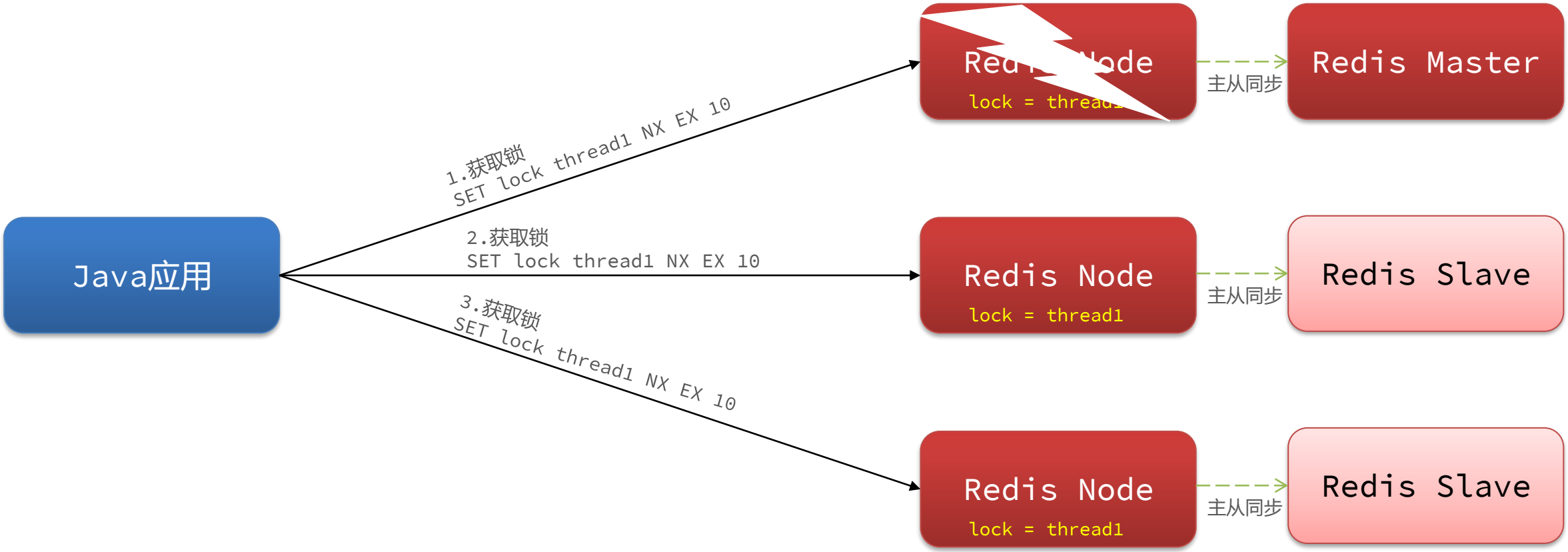
Redisson分布式锁主从一致性问题

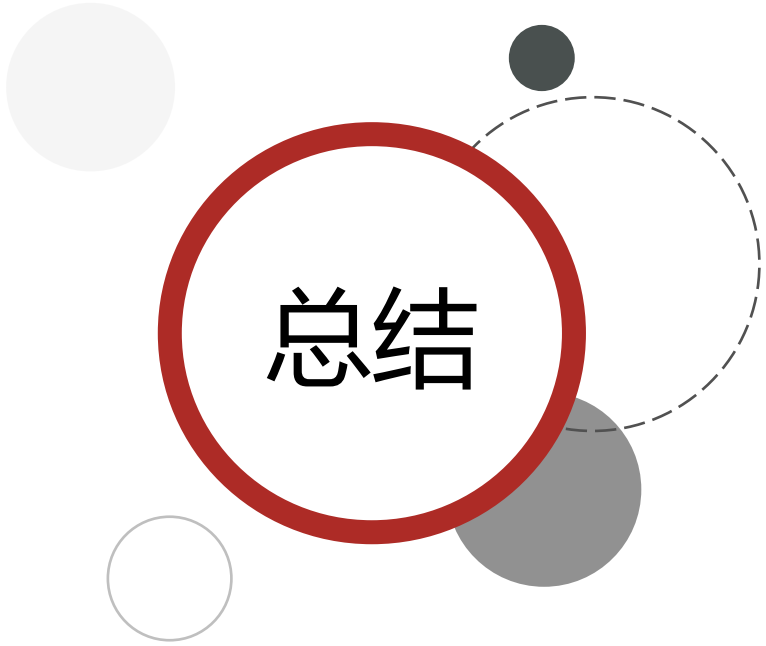


Redisson分布式锁主从一致性问题



Redisson分布式锁主从一致性问题





1）不可重入Redis分布式锁：

- 原理：利用setnx的互斥性；利用ex避免死锁；释放锁时判断线程标示
- 缺陷：不可重入、无法重试、锁超时失效

2）可重入的Redis分布式锁：

- 原理：利用hash结构，记录线程标示和重入次数；利用watchDog延续锁时间；利用信号量控制锁重试等待
- 缺陷：redis宕机引起锁失效问题

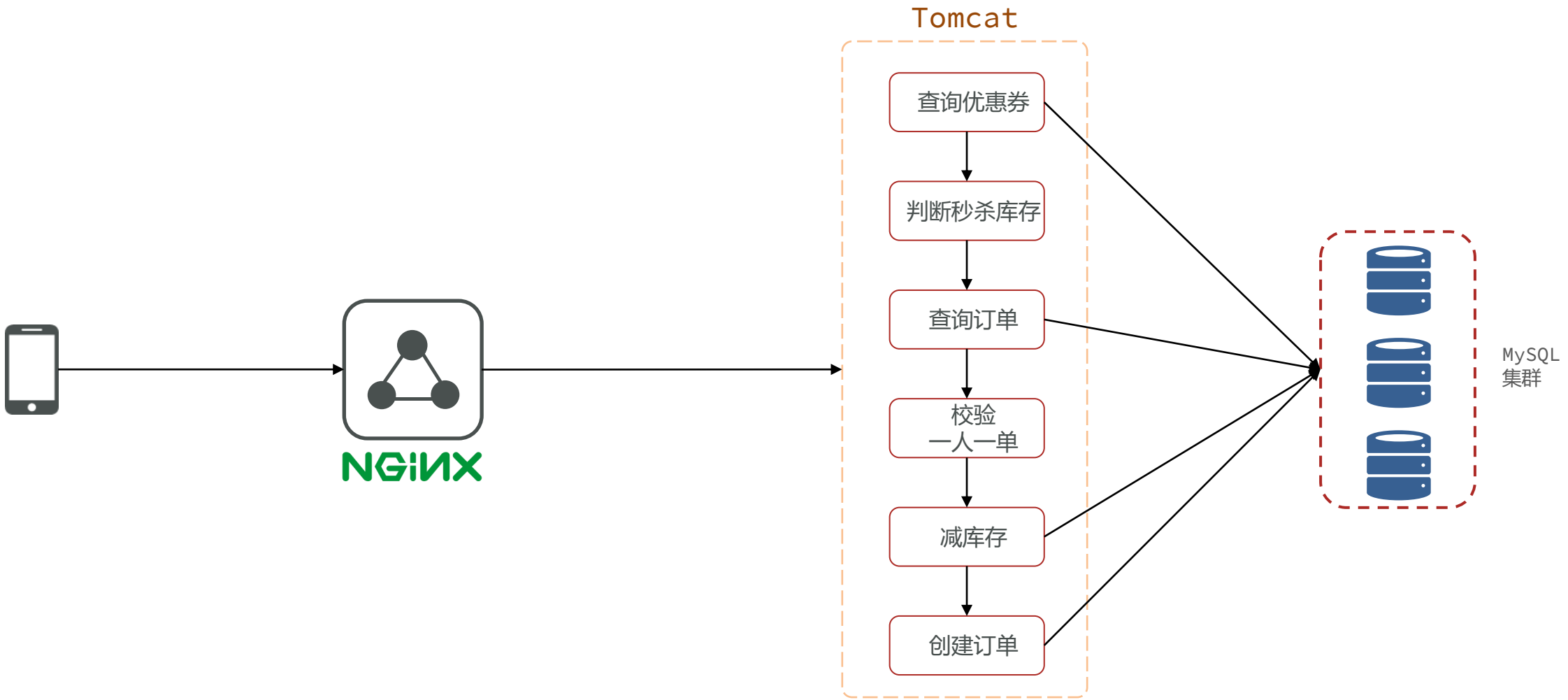
3）Redisson的multiLock：

- 原理：多个独立的Redis节点，必须在所有节点都获取重入锁，才算获取锁成功
- 缺陷：运维成本高、实现复杂

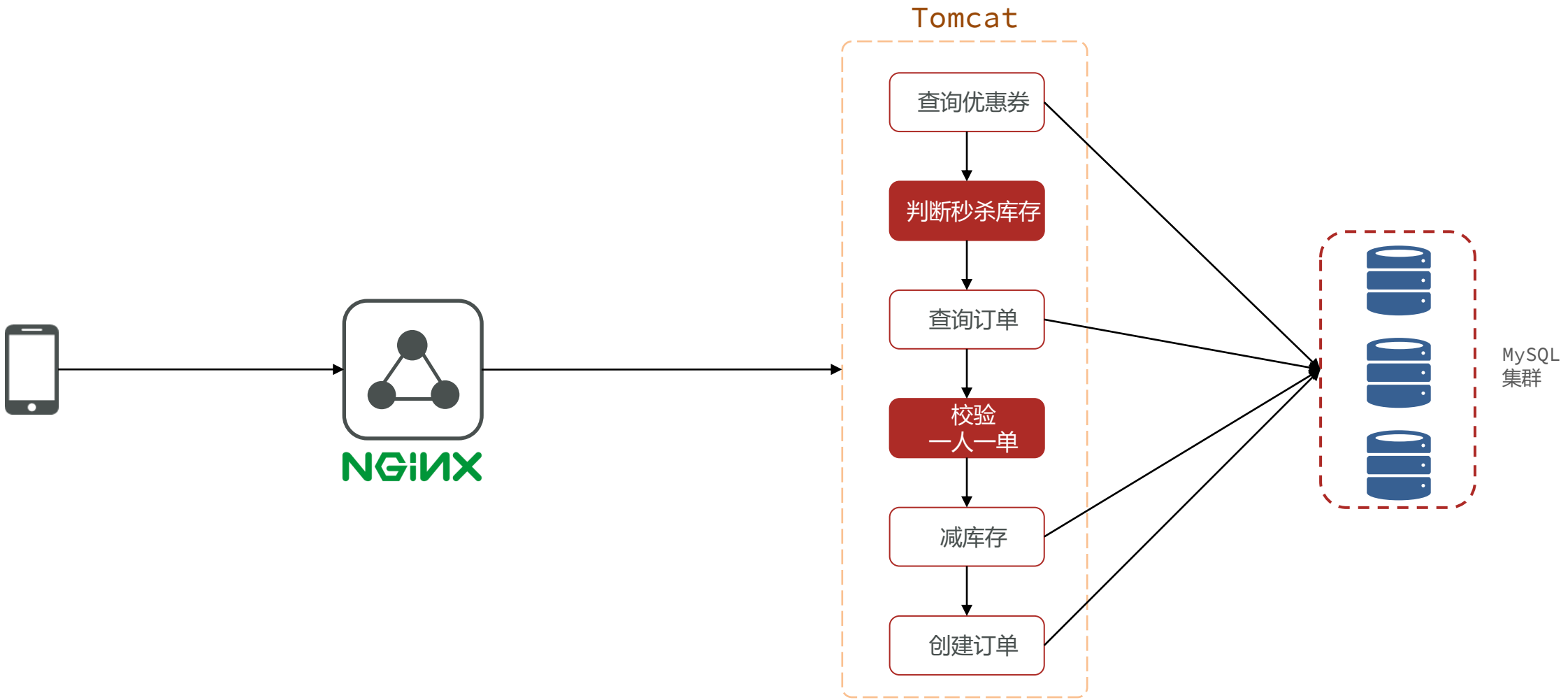
目录

Contents

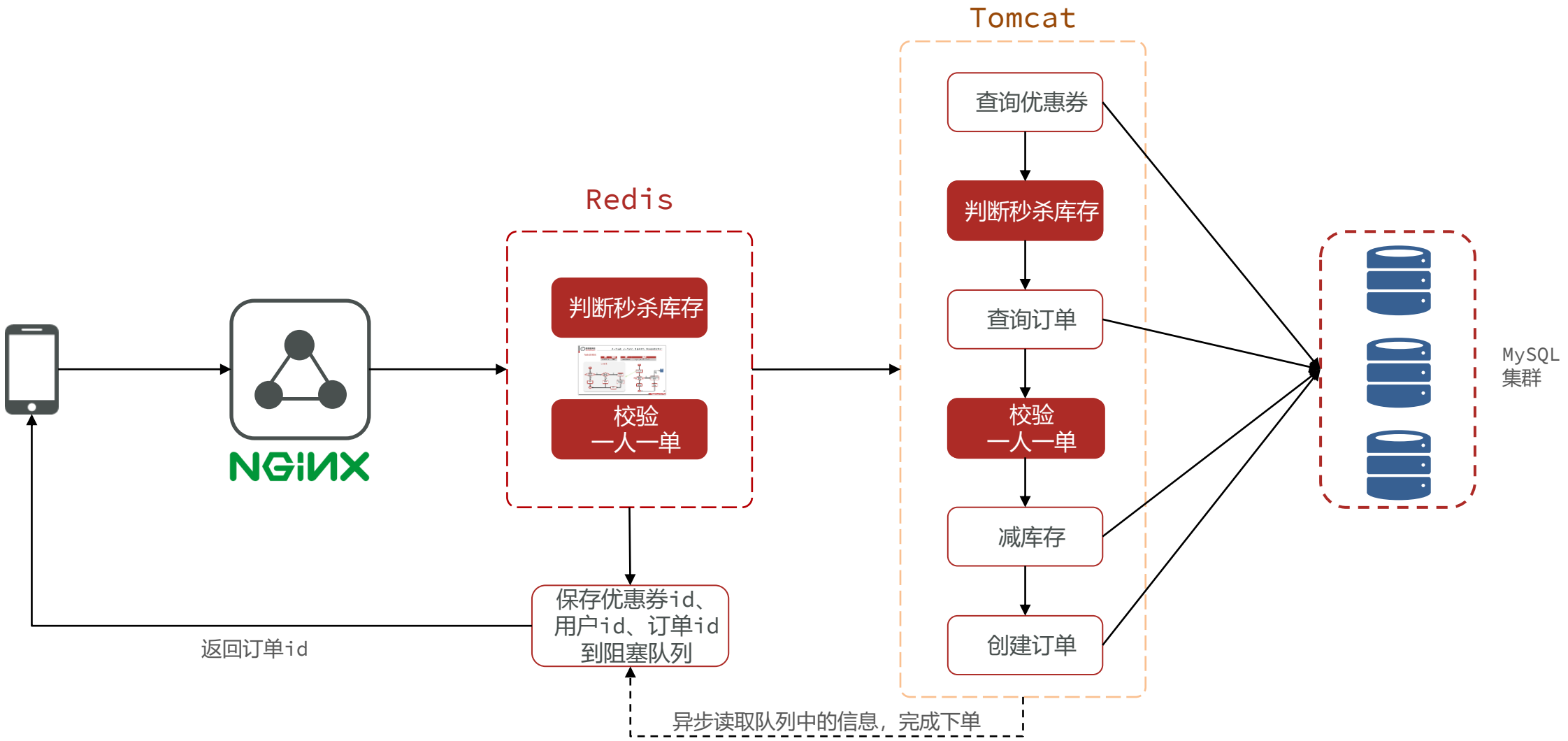
Redis优化秒杀



Redis优化秒杀



Redis优化秒杀

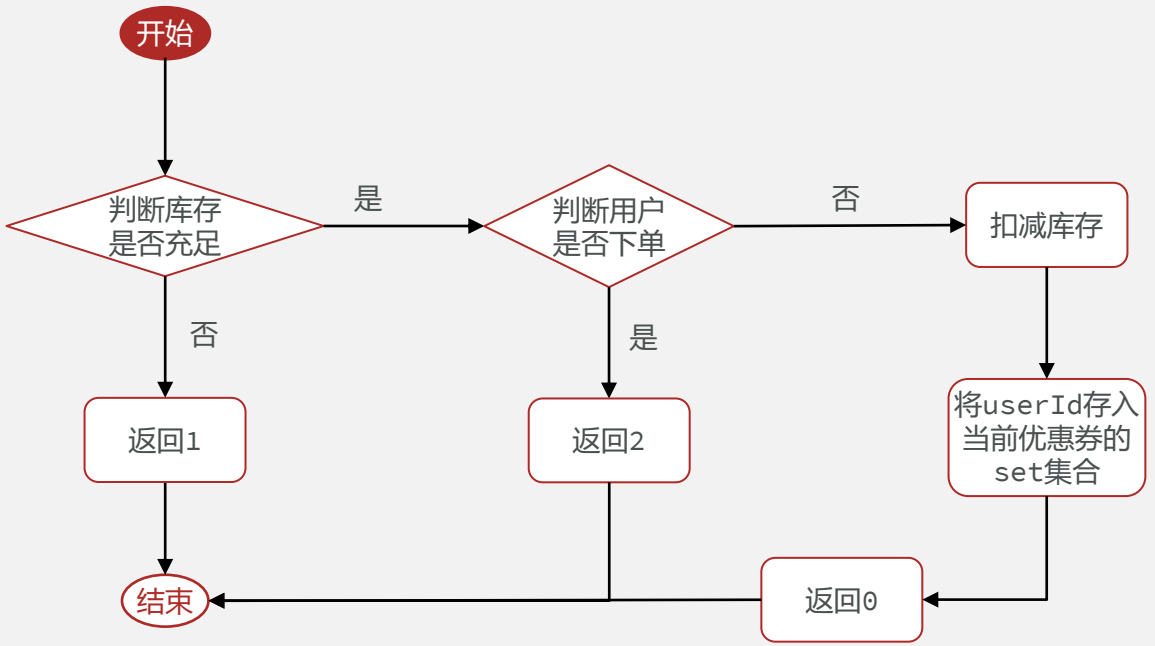

Tomcat
查询优惠券
判断秒杀库存
查询订单
校验
一人一单
减库存
创建订单
Redis
判断秒杀库存
校验
一人一单
NGINX
MySQL
集群
保存优惠券id、
用户id、订单id
到阻塞队列
返回订单id
异步读取队列中的信息，完成下单

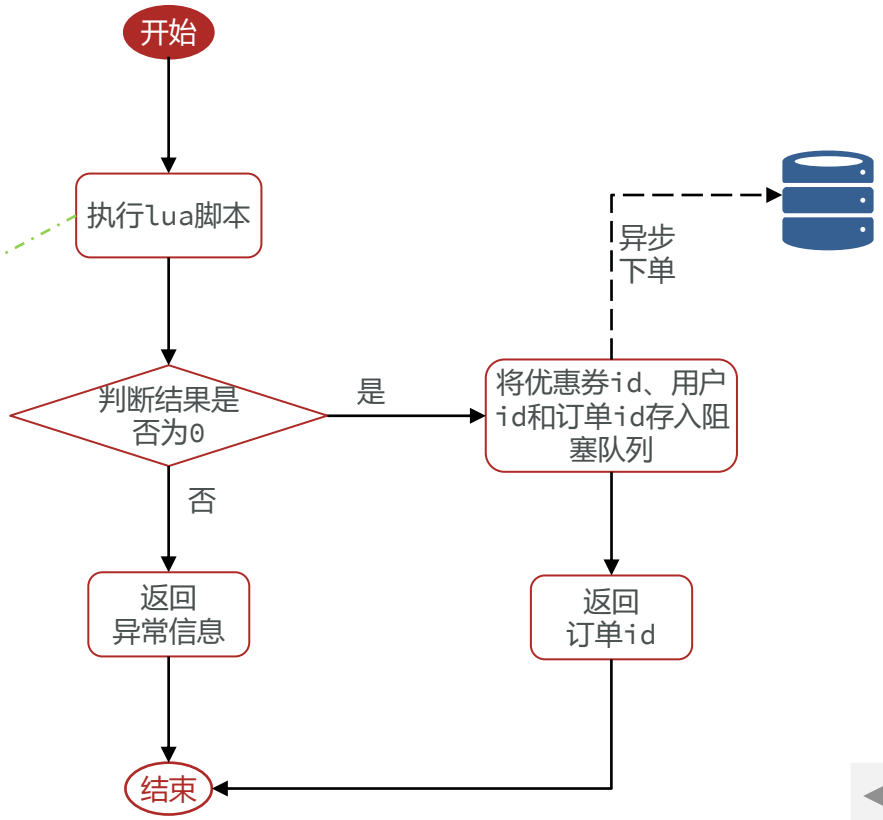
Redis优化秒杀

KEY	VALUE
stock:vid:7	100

KEY	VALUE
order:vid:7	1 , 2 , 3 , 5 , 7 , 8

Lua脚本





案例 改进秒杀业务，提高并发性能

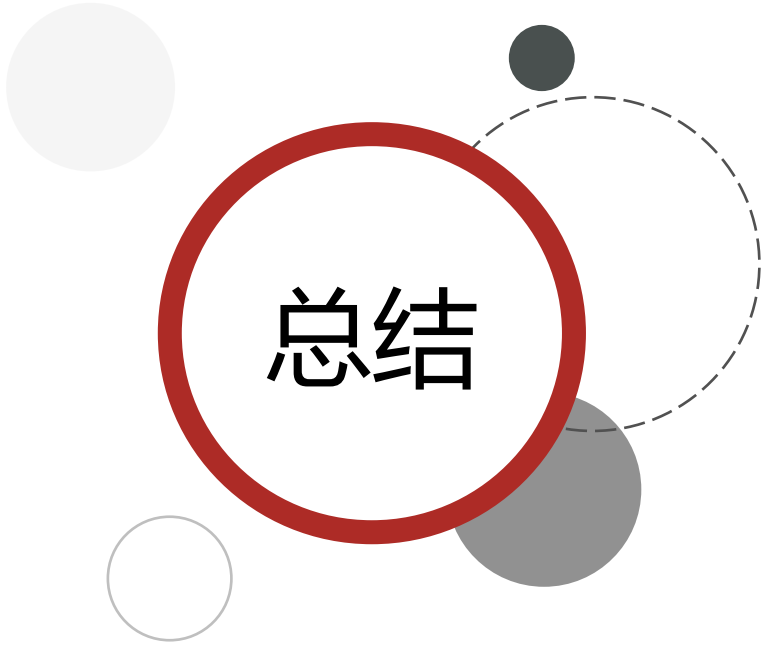
需求：

① 新增秒杀优惠券的同时，将优惠券信息保存到Redis中

② 基于Lua脚本，判断秒杀库存、一人一单，决定用户是否抢购成功

③ 如果抢购成功，将优惠券id和用户id封装后存入阻塞队列

④ 开启线程任务，不断从阻塞队列中获取信息，实现异步下单功能



秒杀业务的优化思路是什么？

① 先利用Redis完成库存余量、一人一单判断，完成抢单业务

② 再将下单业务放入阻塞队列，利用独立线程异步下单

基于阻塞队列的异步秒杀存在哪些问题？

- 内存限制问题
- 数据安全问题

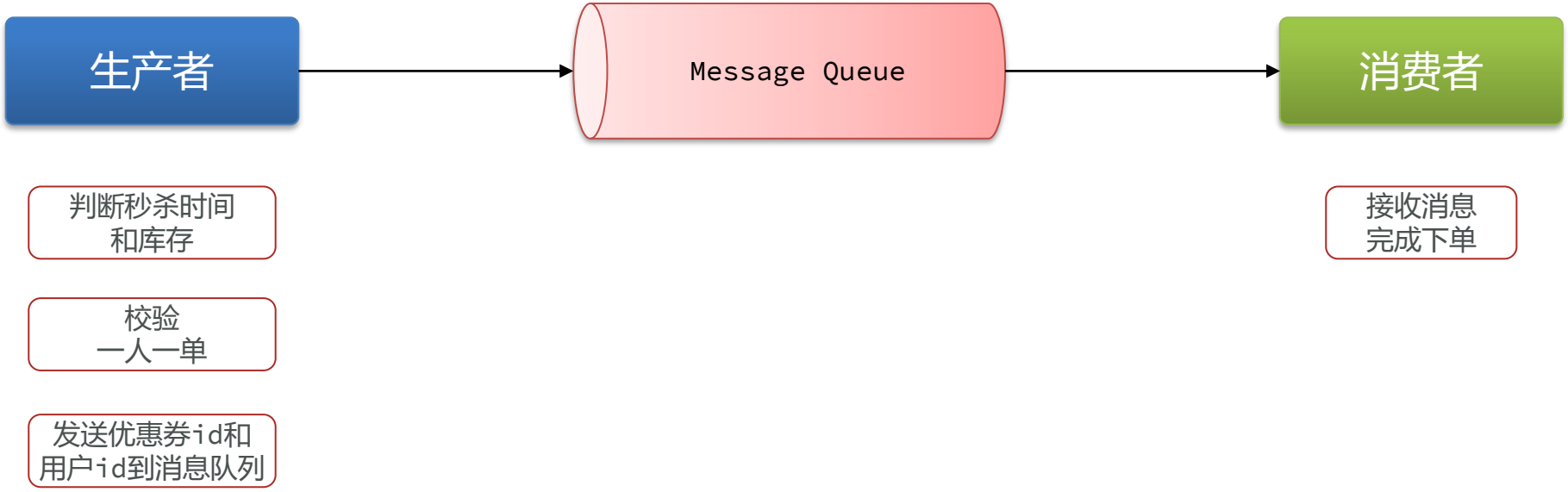
目录

Contents

Redis消息队列实现异步秒杀

消息队列（Message Queue），字面意思就是存放消息的队列。最简单的消息队列模型包括3个角色：

- 消息队列：存储和管理消息，也被称为消息代理（Message Broker）
- 生产者：发送消息到消息队列
- 消费者：从消息队列获取消息并处理消息



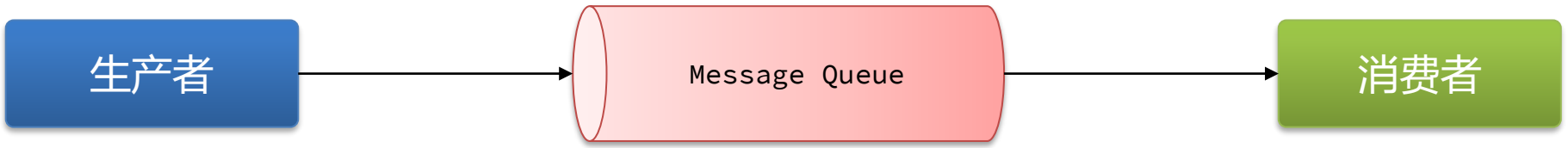
Redis消息队列实现异步秒杀

消息队列（Message Queue），字面意思就是存放消息的队列。最简单的消息队列模型包括3个角色：

- 消息队列：存储和管理消息，也被称为消息代理（Message Broker）
- 生产者：发送消息到消息队列
- 消费者：从消息队列获取消息并处理消息

Redis提供了三种不同的方式来实现消息队列：

- list结构：基于List结构模拟消息队列
- PubSub：基本的点对点消息模型
- Stream：比较完善的消息队列模型

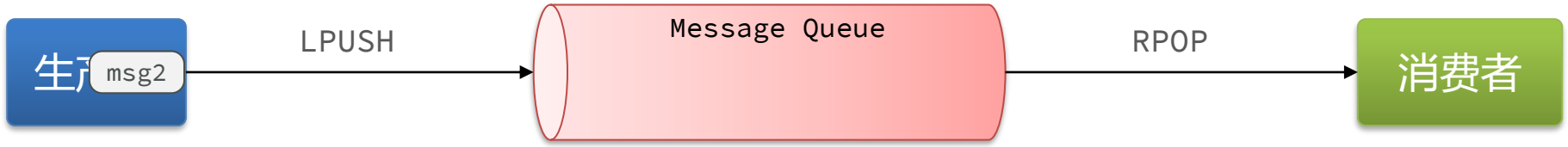


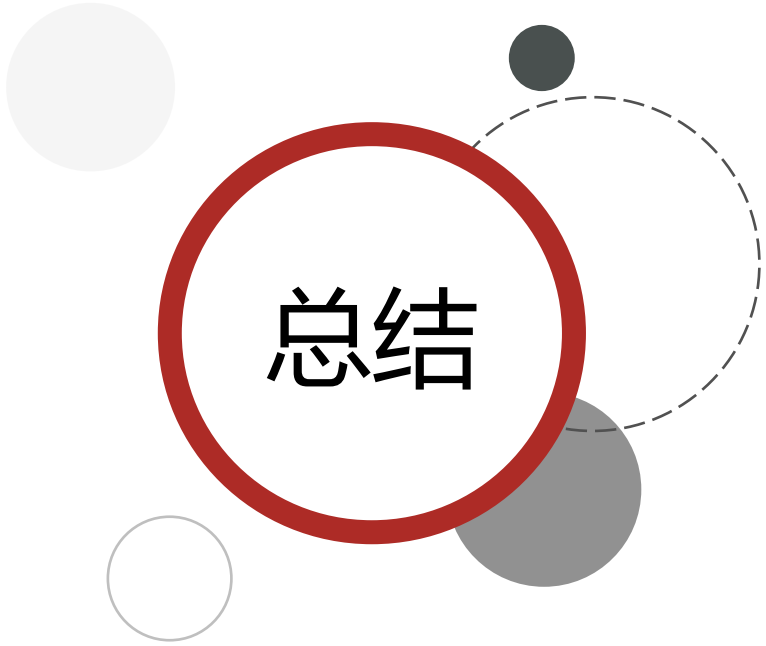
基于List结构模拟消息队列

消息队列（Message Queue），字面意思就是存放消息的队列。而Redis的list数据结构是一个双向链表，很容易模拟出队列效果。

队列是入口和出口不在一边，因此我们可以利用：LPUSH 结合 RPOP、或者 RPUSH 结合 LPOP来实现。

不过要注意的是，当队列中没有消息时RPOP或LPOP操作会返回null，并不像JVM的阻塞队列那样会阻塞并等待消息。因此这里应该使用**BRPOP**或者**BLPOP**来实现阻塞效果。





基于List的消息队列有哪些优缺点?

优点:

- 利用Redis存储，不受限于JVM内存上限
- 基于Redis的持久化机制，数据安全性有保证
- 可以满足消息有序性

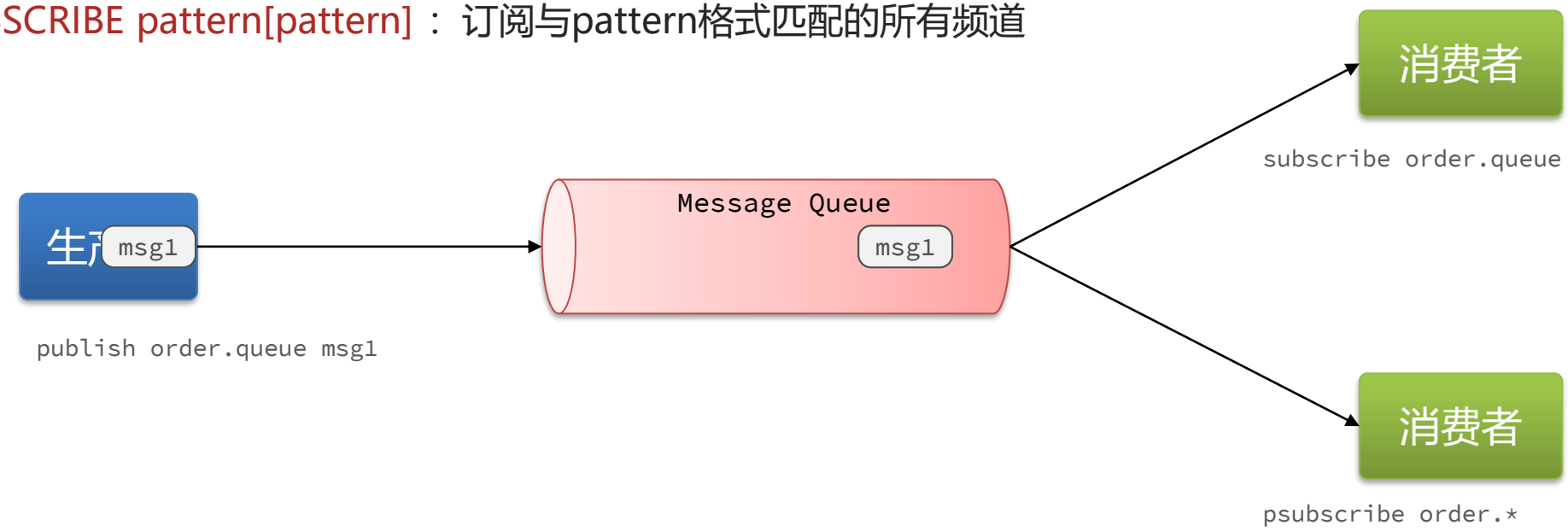
缺点:

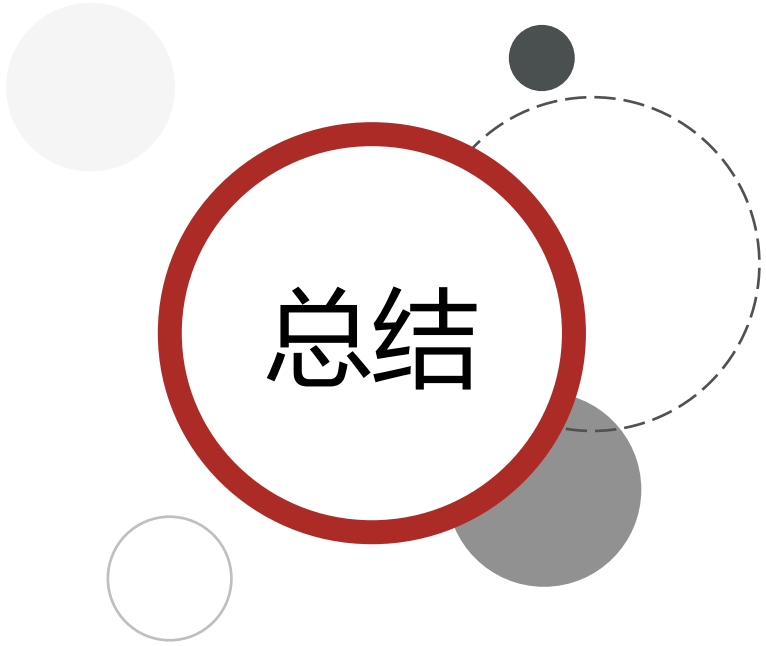
- 无法避免消息丢失
- 只支持单消费者

基于PubSub的消息队列

PubSub（发布订阅）是Redis2.0版本引入的消息传递模型。顾名思义，消费者可以订阅一个或多个channel，生产者向对应channel发送消息后，所有订阅者都能收到相关消息。

- SUBSCRIBE channel [channel]：订阅一个或多个频道
- PUBLISH channel msg：向一个频道发送消息
- PSUBSCRIBE pattern[pattern]：订阅与pattern格式匹配的所有频道





基于PubSub的消息队列有哪些优缺点?

优点:

- 采用发布订阅模型，支持多生产、多消费

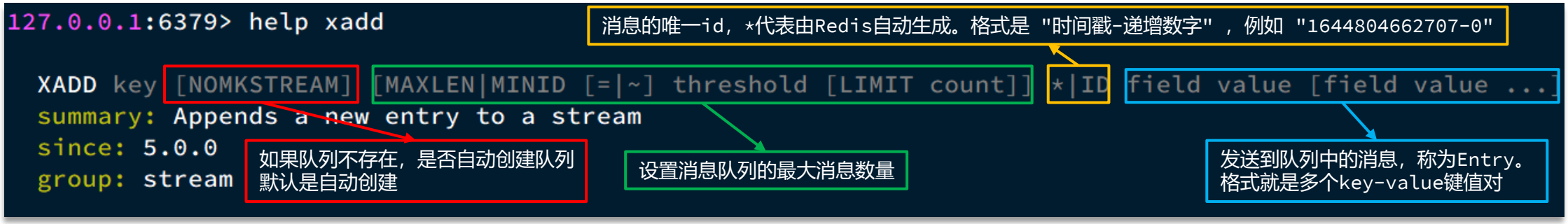
缺点:

- 不支持数据持久化
- 无法避免消息丢失
- 消息堆积有上限，超出时数据丢失

基于Stream的消息队列

Stream 是 Redis 5.0 引入的一种新数据类型，可以实现一个功能非常完善的消息队列。

发送消息的命令：



例如：

```
## 创建名为 users 的队列，并向其中发送一个消息，内容是：{name=jack,age=21}，并且使用Redis自动生成ID
127.0.0.1:6379> XADD users * name jack age 21
"1644805700523-0"
```

基于Stream的消息队列-XREAD

读取消息的方式之一：XREAD

```
127.0.0.1:6379> help XREAD

  XREAD [COUNT count] [BLOCK milliseconds] STREAMS key [key ...] ID [ID ...]
  summary: Return never seen elements in multiple streams, with IDs greater than
 the ones reported by the caller for each stream. Can block.
  since: 5.0.0
  group: stream
```

每次读取消息的最大数量

当没有消息时，是否阻塞、阻塞时长

要从哪个队列读取消息，key就是队列名

起始id，只返回大于该ID的消息
0：代表从第一个消息开始
$:代表从最新的消息开始

例如，使用XREAD读取第一个消息：

```
127.0.0.1:6379> XREAD COUNT 1 STREAMS users 0
1) 1) "users"
   2) 1) 1) "1644805700523-0"
         2) 1) "name"
            2) "jack"
            3) "age"
            4) "21"
```

基于Stream的消息队列-XREAD

XREAD阻塞方式，读取最新的消息：

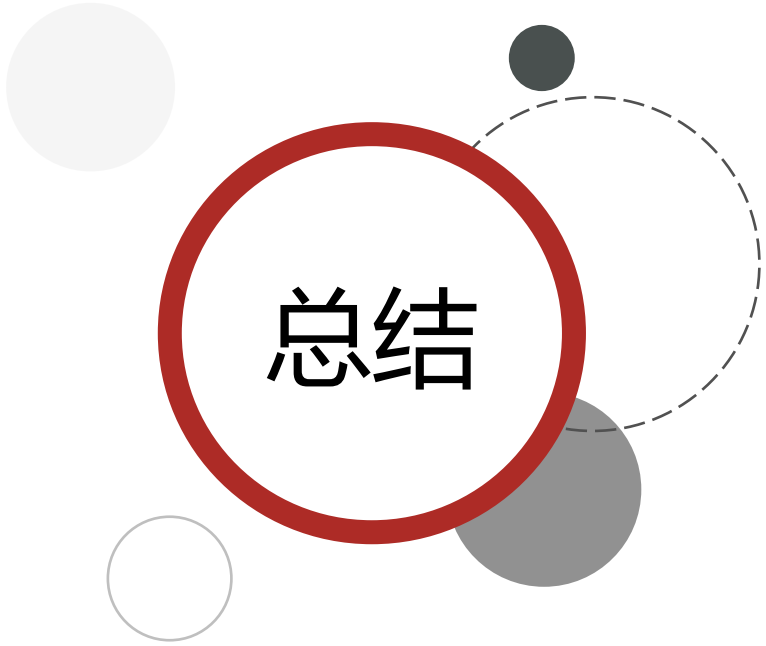
```
127.0.0.1:6379> XREAD COUNT 1 BLOCK 1000 STREAMS users $
(nil)
(1.07s)
```

在业务开发中，我们可以循环的调用XREAD阻塞方式来查询最新消息，从而实现持续监听队列的效果，伪代码如下：

```
while(true){
    // 尝试读取队列中的消息，最多阻塞2秒
    Object msg = redis.execute("XREAD COUNT 1 BLOCK 2000 STREAMS users $");
    if(msg == null){
        continue;
    }
    // 处理消息
    handleMessage(msg);
}
```

注意

当我们指定起始ID为$时，代表读取最新的消息，如果我们处理一条消息的过程中，又有超过1条以上的消息到达队列，则下次获取时也只能获取到最新的一条，会出现漏读消息的问题。



STREAM类型消息队列的XREAD命令特点：

- 消息可回溯
- 一个消息可以被多个消费者读取
- 可以阻塞读取
- 有消息漏读的风险

基于Stream的消息队列-消费者组

消费者组（Consumer Group）：将多个消费者划分到一个组中，监听同一个队列。具备下列特点：

01

消息分流

队列中的消息会分流给组内的不同消费者，而不是重复消费，从而加快消息处理的速度

02

消息标示

消费者组会维护一个标示，记录最后一个被处理的消息，哪怕消费者宕机重启，还会从标示之后读取消息。确保每一个消息都会被消费

03

消息确认

消费者获取消息后，消息处于pending状态，并存入一个pending-list。当处理完成后需要通过XACK来确认消息，标记消息为已处理，才会从pending-list移除。

基于Stream的消息队列-消费者组

创建消费者组：

```
XGROUP CREATE  key groupName ID [MKSTREAM]
```

- key：队列名称
- groupName：消费者组名称
- ID：起始ID标示，$代表队列中最后一个消息，0则代表队列中第一个消息
- MKSTREAM：队列不存在时自动创建队列

其它常见命令：

```
# 删除指定的消费者组
XGROUP DESTORY key groupName

# 给指定的消费者组添加消费者
XGROUP CREATECONSUMER key groupname consumername

# 删除消费者组中的指定消费者
XGROUP DELCONSUMER key groupname consumername
```

基于Stream的消息队列-消费者组

从消费者组读取消息：

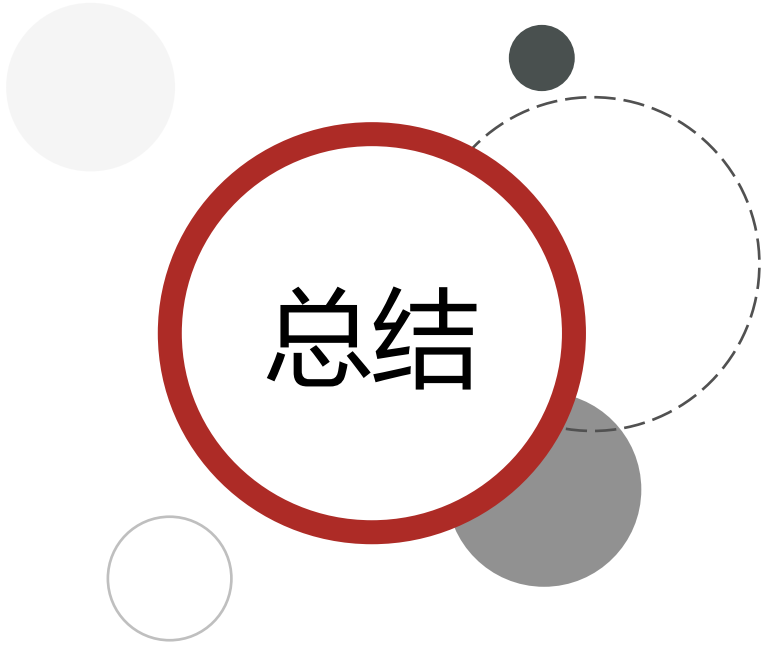
```
XREADGROUP GROUP group consumer [COUNT count] [BLOCK milliseconds] [NOACK] STREAMS
key [key ...] ID [ID ...]
```

- group：消费组名称
- consumer：消费者名称，如果消费者不存在，会自动创建一个消费者
- count：本次查询的最大数量
- BLOCK milliseconds：当没有消息时最长等待时间
- NOACK：无需手动ACK，获取到消息后自动确认
- STREAMS key：指定队列名称
- ID：获取消息的起始ID：
  - ">"：从下一个未消费的消息开始
  - 其它：根据指定id从pending-list中获取已消费但未确认的消息，例如0，是从pending-list中的第一个消息开始

基于Stream的消息队列-消费者组

消费者监听消息的基本思路：

```
while(true){
    // 尝试监听队列，使用阻塞模式，最长等待 2000 毫秒
    Object msg = redis.call("XREADGROUP GROUP g1 c1 COUNT 1 BLOCK 2000 STREAMS s1 >");
    if(msg == null){ // null说明没有消息，继续下一次
        continue;
    }
    try {
        // 处理消息，完成后一定要ACK
        handleMessage(msg);
    } catch(Exception e){
        while(true){
            Object msg = redis.call("XREADGROUP GROUP g1 c1 COUNT 1 STREAMS s1 0");
            if(msg == null){ // null说明没有异常消息，所有消息都已确认，结束循环
                break;
            }
            try {
                // 说明有异常消息，再次处理
                handleMessage(msg);
            } catch(Exception e){
                // 再次出现异常，记录日志，继续循环
                continue;
            }
        }
    }
}
```



STREAM类型消息队列的XREADGROUP命令特点：

- 消息可回溯
- 可以多消费者争抢消息，加快消费速度
- 可以阻塞读取
- 没有消息漏读的风险
- 有消息确认机制，保证消息至少被消费一次

Redis消息队列

	List	PubSub	Stream
消息持久化	支持	不支持	支持
阻塞读取	支持	支持	支持
消息堆积处理	受限于内存空间，可以利用多消费者加快处理	受限于消费者缓冲区	受限于队列长度，可以利用消费者组提高消费速度，减少堆积
消息确认机制	不支持	不支持	支持
消息回溯	不支持	不支持	支持

案例 基于Redis的Stream结构作为消息队列，实现异步秒杀下单

需求：

① 创建一个Stream类型的消息队列，名为stream.orders

② 修改之前的秒杀下单Lua脚本，在认定有抢购资格后，直接向stream.orders中添加消息，内容包含voucherId、userId、orderId

③ 项目启动时，开启一个线程任务，尝试获取stream.orders中的消息，完成下单

04 达人探店

目录

Contents

发布探店笔记

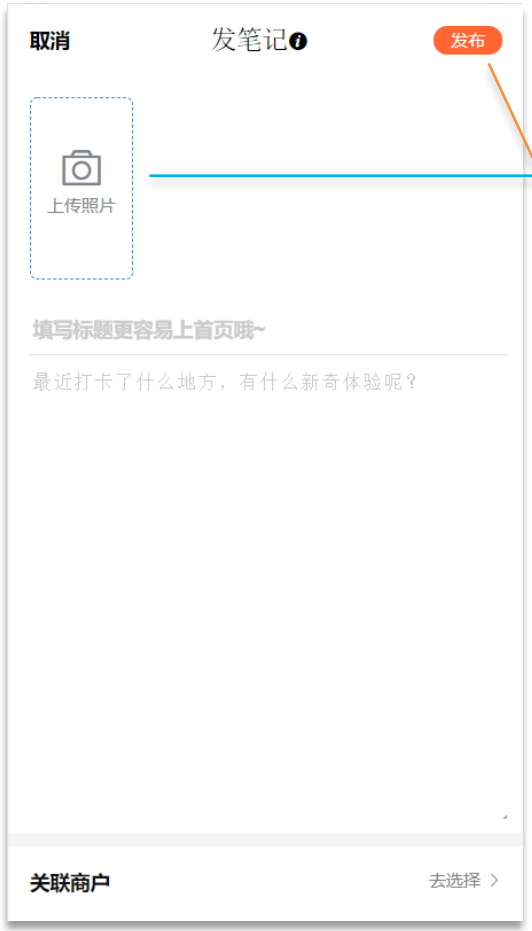
探店笔记类似点评网站的评价，往往是图文结合。对应的表有两个：

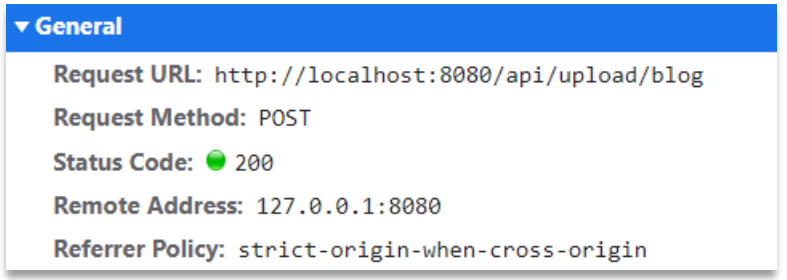
- tb_blog：探店笔记表，包含笔记中的标题、文字、图片等
- tb_blog_comments：其他用户对探店笔记的评价

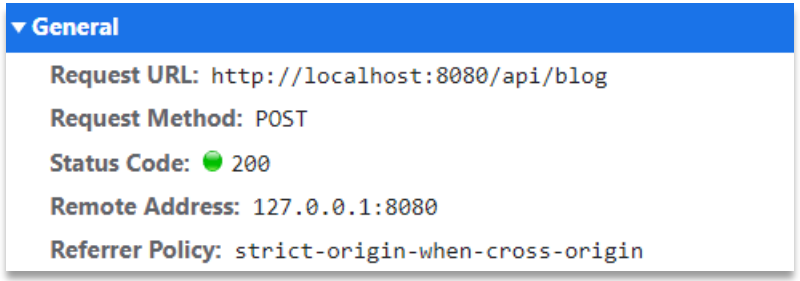

```
CREATE TABLE `tb_blog` (
  `id` bigint(20) unsigned NOT NULL AUTO_INCREMENT COMMENT '主键',
  `shop_id` bigint(20) NOT NULL COMMENT '商户id',
  `user_id` bigint(20) unsigned NOT NULL COMMENT '用户id',
  `title` varchar(255) CHARACTER SET utf8mb4 COLLATE utf8mb4_unicode_ci NOT NULL
COMMENT '标题',
  `images` varchar(2048) NOT NULL COMMENT '探店的照片，最多9张，多张以","隔开',
  `content` varchar(2048) CHARACTER SET utf8mb4 COLLATE utf8mb4_unicode_ci NOT NULL
COMMENT '探店的文字描述',
  `liked` int(8) unsigned zerofill DEFAULT '00000000' COMMENT '点赞数量',
  `comments` int(8) unsigned zerofill DEFAULT NULL COMMENT '评论数量',
  `create_time` timestamp NOT NULL DEFAULT CURRENT_TIMESTAMP COMMENT '创建时间',
  `update_time` timestamp NOT NULL DEFAULT CURRENT_TIMESTAMP ON UPDATE
CURRENT_TIMESTAMP COMMENT '更新时间',
  PRIMARY KEY (`id`)
) ENGINE=InnoDB AUTO_INCREMENT=9 DEFAULT CHARSET=utf8mb4;
```

发布探店笔记

点击首页最下方菜单栏中的+按钮，即可发布探店图文：

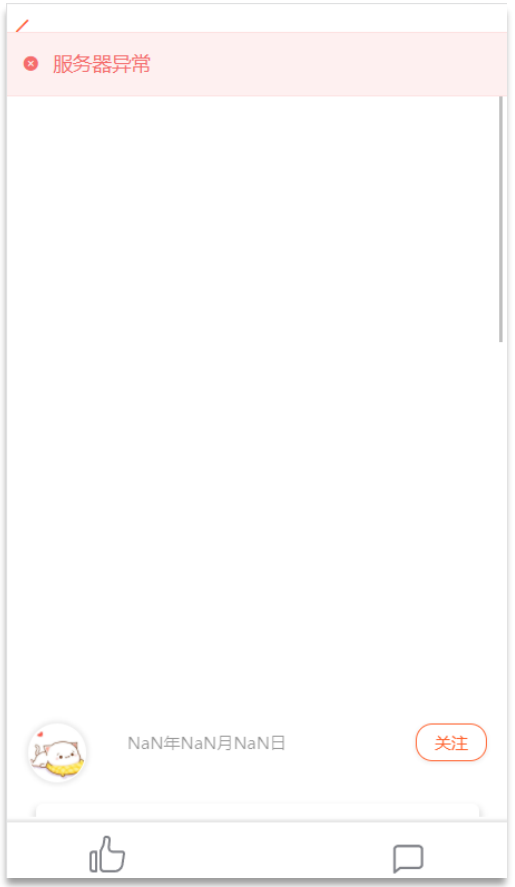






案例 实现查看发布探店笔记的接口

需求：点击首页的探店笔记，会进入详情页面，实现该页面的查询接口：



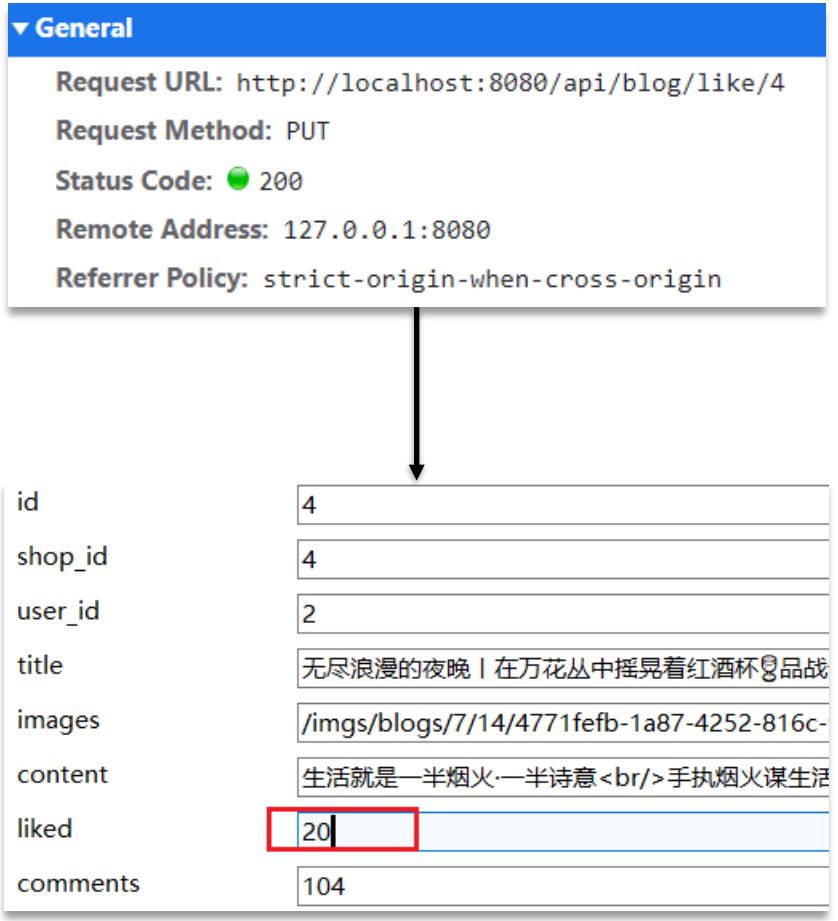
	说明
请求方式	GET
请求路径	/blog/{id}
请求参数	id：blog的id
返回值	Blog：笔记信息，包含用户信息

目录

Contents

点赞

在首页的探店笔记排行榜和探店图文详情页面都有点赞的功能：



案例 完善点赞功能

需求：

- 同一个用户只能点赞一次，再次点击则取消点赞
- 如果当前用户已经点赞，则点赞按钮高亮显示（前端已实现，判断字段Blog类的isLike属性）

实现步骤：

① 给Blog类中添加一个isLike字段，标示是否被当前用户点赞

② 修改点赞功能，利用Redis的set集合判断是否点赞过，未点赞过则点赞数+1，已点赞过则点赞数-1

③ 修改根据id查询Blog的业务，判断当前登录用户是否点赞过，赋值给isLike字段

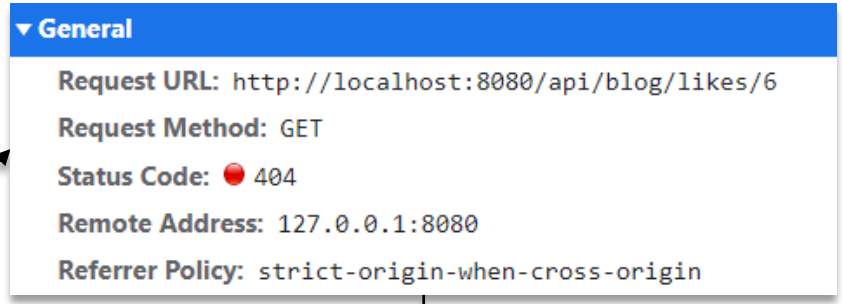
④ 修改分页查询Blog业务，判断当前登录用户是否点赞过，赋值给isLike字段

目录

Contents

点赞排行榜

在探店笔记的详情页面，应该把给该笔记点赞的人显示出来，比如最早点赞的TOP5，形成点赞排行榜：



	说明
请求方式	GET
请求路径	/blog/likes/{id}
请求参数	id: blog的id
返回值	List<UserDTO> 给这个笔记点赞的TopN用户集合

案例 实现查询点赞排行榜的接口

需求：按照点赞时间先后排序，返回Top5的用户

	List	Set	SortedSet
排序方式	按添加顺序排序	无法排序	根据score值排序
唯一性	不唯一	唯一	唯一
查找方式	按索引查找 或首尾查找	根据元素查找	根据元素查找

05 好友关注

目录

Contents

关注和取关

在探店图文的详情页面中，可以关注发布笔记的作者：

1.尝试关注用户

General

Request URL: http://localhost:8080/api/follow/2/true
Request Method: PUT
Status Code: 404
Remote Address: 127.0.0.1:8080
Referrer Policy: strict-origin-when-cross-origin

2.是否关注用户?

General

Request URL: http://localhost:8080/api/follow/or/not/2
Request Method: GET
Status Code: 404
Remote Address: 127.0.0.1:8080
Referrer Policy: strict-origin-when-cross-origin

案例 实现关注和取关功能

需求：基于该表数据结构，实现两个接口：

① 关注和取关接口

② 判断是否关注的接口

关注是User之间的关系，是博主与粉丝的关系，数据库中有一张tb_follow表来标示：

```
CREATE TABLE `tb_follow` (
  `id` bigint(20) NOT NULL COMMENT '主键',
  `user_id` bigint(20) unsigned NOT NULL COMMENT '用户id',
  `follow_user_id` bigint(20) unsigned NOT NULL COMMENT '关联的用户id',
  `create_time` timestamp NOT NULL DEFAULT CURRENT_TIMESTAMP COMMENT '创建时间',
  PRIMARY KEY (`id`)
) ENGINE=InnoDB DEFAULT CHARSET=utf8mb4;
```

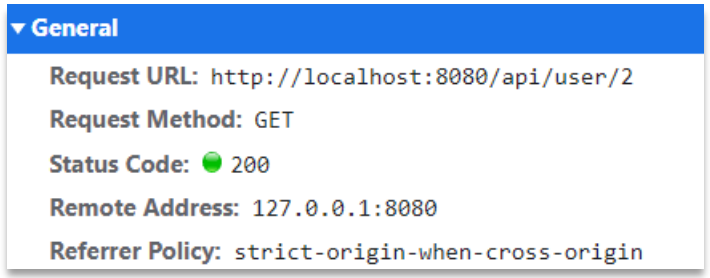
注意: 这里需要把主键修改为自增长，简化开发。

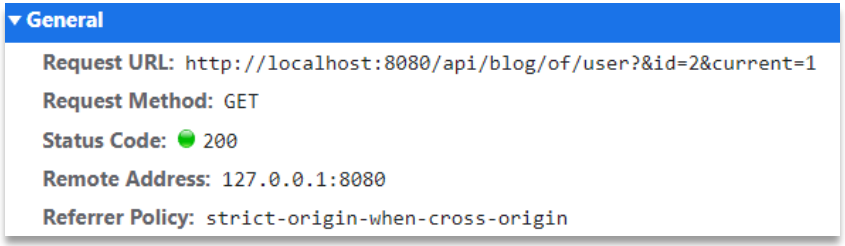
目录

Contents

共同关注

点击博主头像，可以进入博主首页：





博主个人主页

博主个人首页依赖两个接口：

① 根据id查询user信息：

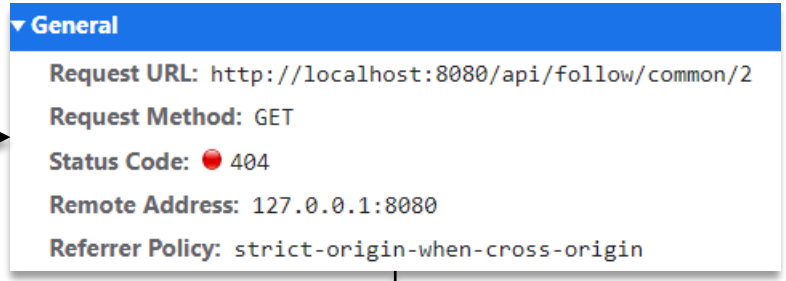
```
@GetMapping("/{id}")
public Result queryUserById(@PathVariable("id") Long userId){
    User user = userService.getById(userId);
    if (user == null) {
        return Result.ok();
    }
    UserDTO userDTO = BeanUtil.copyProperties(user, UserDTO.class);
    return Result.ok(userDTO);
}
```

② 根据id查询博主的探店笔记：

```
@GetMapping("/of/user")
public Result queryBlogByUserId(@RequestParam(value = "current", defaultValue = "1") Integer current,
                                @RequestParam("id") Long id) {
    Page<Blog> page = blogService.query()
            .eq("user_id", id).page(new Page<>(current, SystemConstants.MAX_PAGE_SIZE));
    return Result.ok(page.getRecords());
}
```

案例 实现共同关注功能

需求：利用Redis中恰当的数据结构，实现共同关注功能。在博主个人页面展示出当前用户与博主的共同好友。



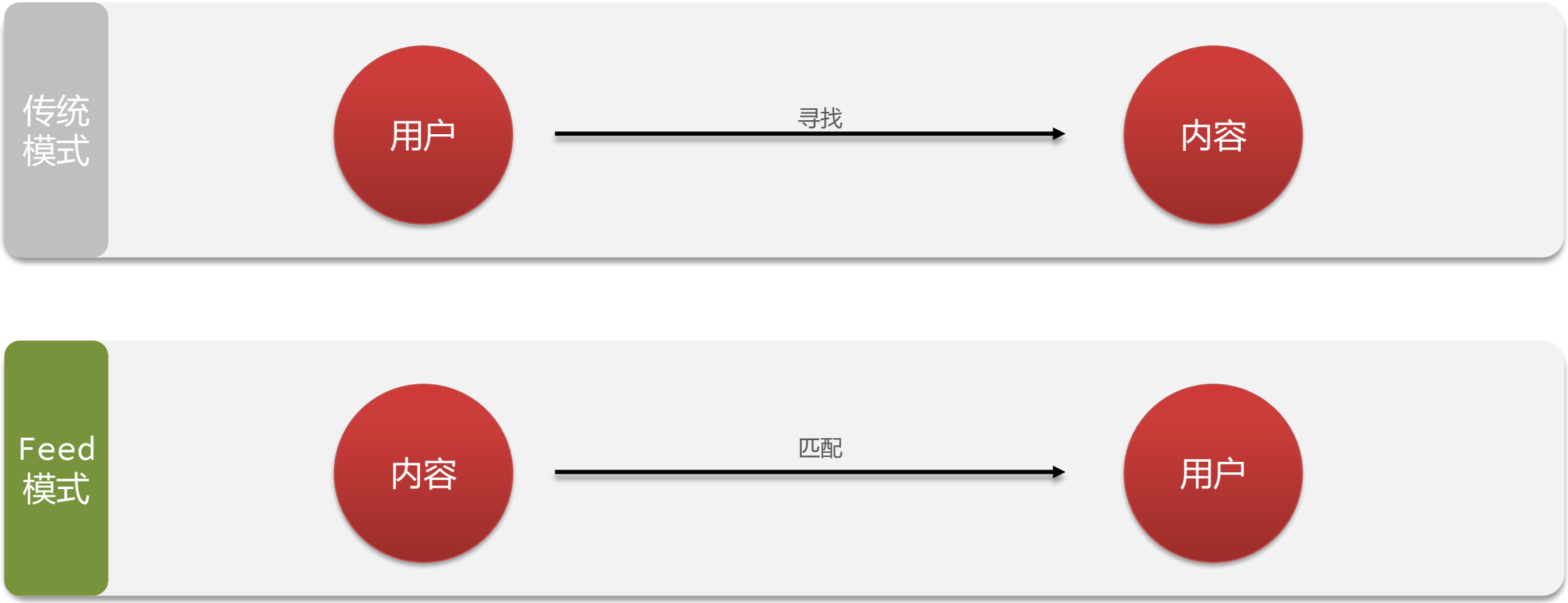
	说明
请求方式	GET
请求路径	/follow/common/{id}
请求参数	id:目标用户id
返回值	List<UserDTO>：两人共同关注的人

目录

Contents

关注推送

关注推送也叫做Feed流，直译为投喂。为用户持续的提供“沉浸式”的体验，通过无限下拉刷新获取新的信息。



Feed流的模式

Feed流产品有两种常见模式：

- **Timeline**：不做内容筛选，简单的按照内容发布时间排序，常用于好友或关注。例如朋友圈
  - 优点：信息全面，不会有缺失。并且实现也相对简单
  - 缺点：信息噪音较多，用户不一定感兴趣，内容获取效率低
- **智能排序**：利用智能算法屏蔽掉违规的、用户不感兴趣的内容。推送用户感兴趣信息来吸引用
  - 优点：投喂用户感兴趣信息，用户粘度很高，容易沉迷
  - 缺点：如果算法不精准，可能起到反作用

本例中的个人页面，是基于关注的好友来做Feed流，因此采用Timeline的模式。该模式的实现方

① 拉模式

② 推模式

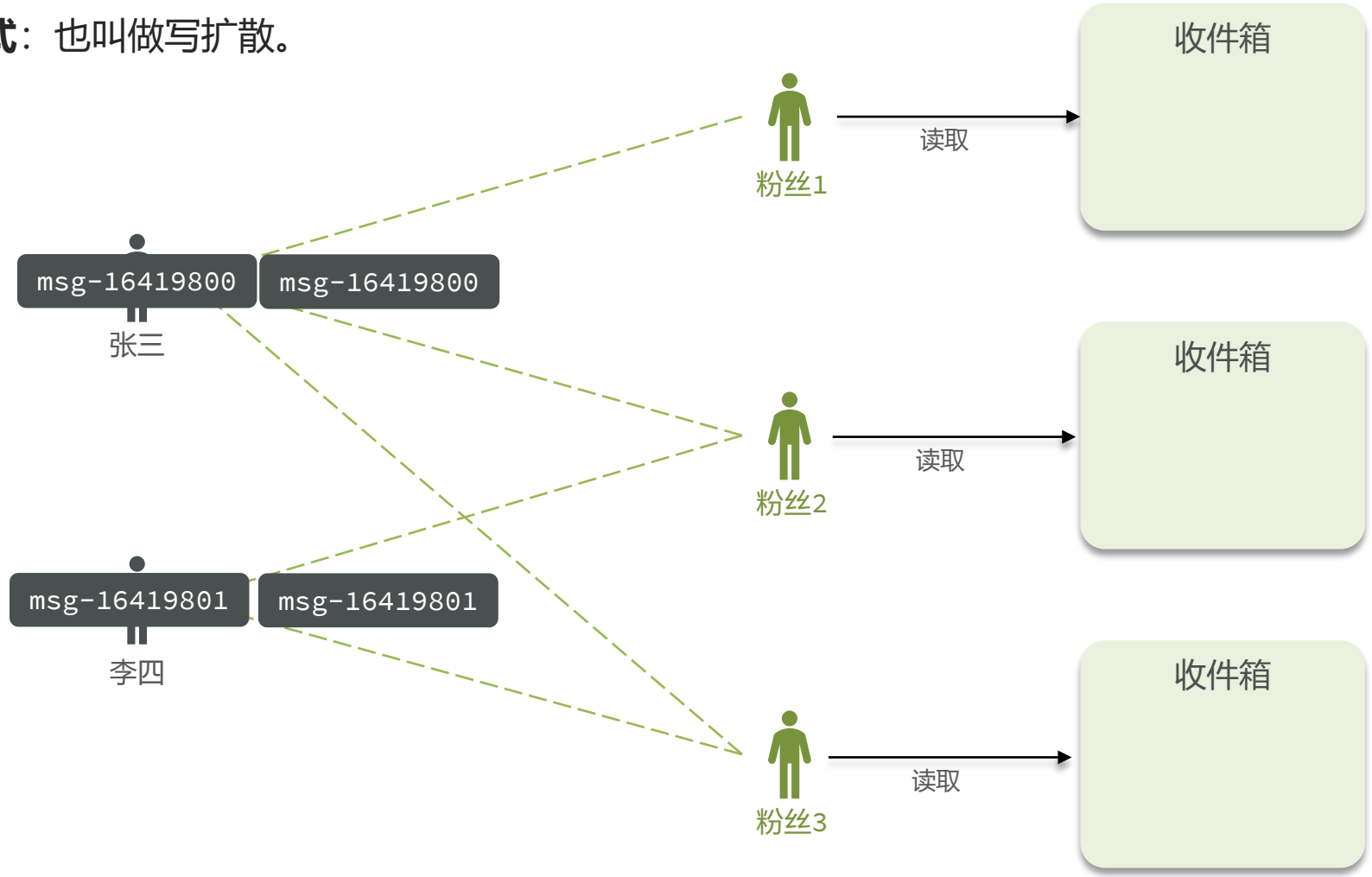
③ 推拉结合

Feed流的实现方案1

拉模式：也叫做读扩散。

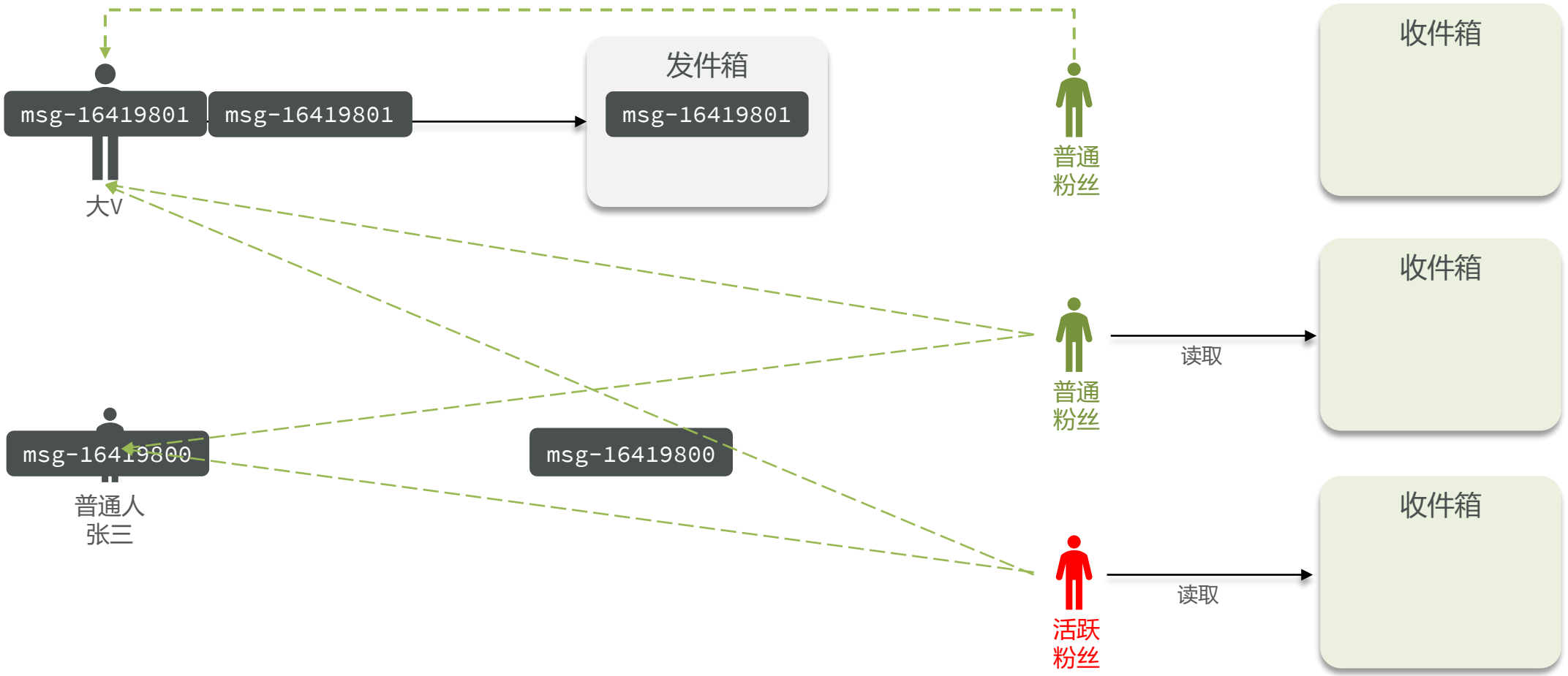
Feed流的实现方案2

推模式：也叫做写扩散。



Feed流的实现方案3

推拉结合模式：也叫做读写混合，兼具推和拉两种模式的优点。



Feed流的实现方案

	拉模式	推模式	推拉结合
写比例	低	高	中
读比例	高	低	中
用户读取延迟	高	低	低
实现难度	复杂	简单	很复杂
使用场景	很少使用	用户量少、没有大V	过千万的用户量，有大V

案例 基于推模式实现关注推送功能

需求：

① 修改新增探店笔记的业务，在保存blog到数据库的同时，推送到粉丝的收件箱

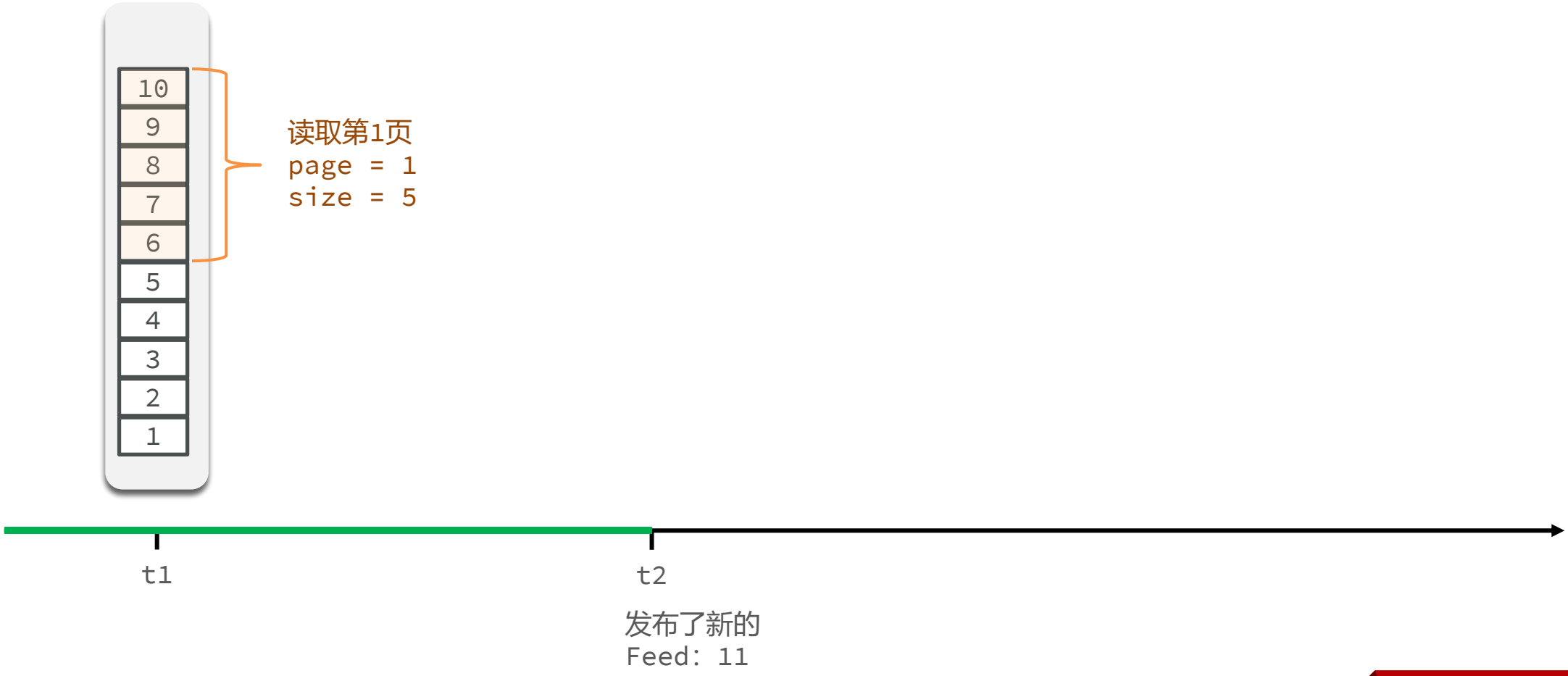
② 收件箱满足可以根据时间戳排序，必须用Redis的数据结构实现

③ 查询收件箱数据时，可以实现分页查询

```
@PostMapping
public Result saveBlog(@RequestBody Blog blog) {
    // 获取登录用户
    UserDTO user = UserHolder.getUser();
    blog.setUserId(user.getId());
    // 保存探店笔记
    blogService.save(blog);
    return Result.ok();
}
```

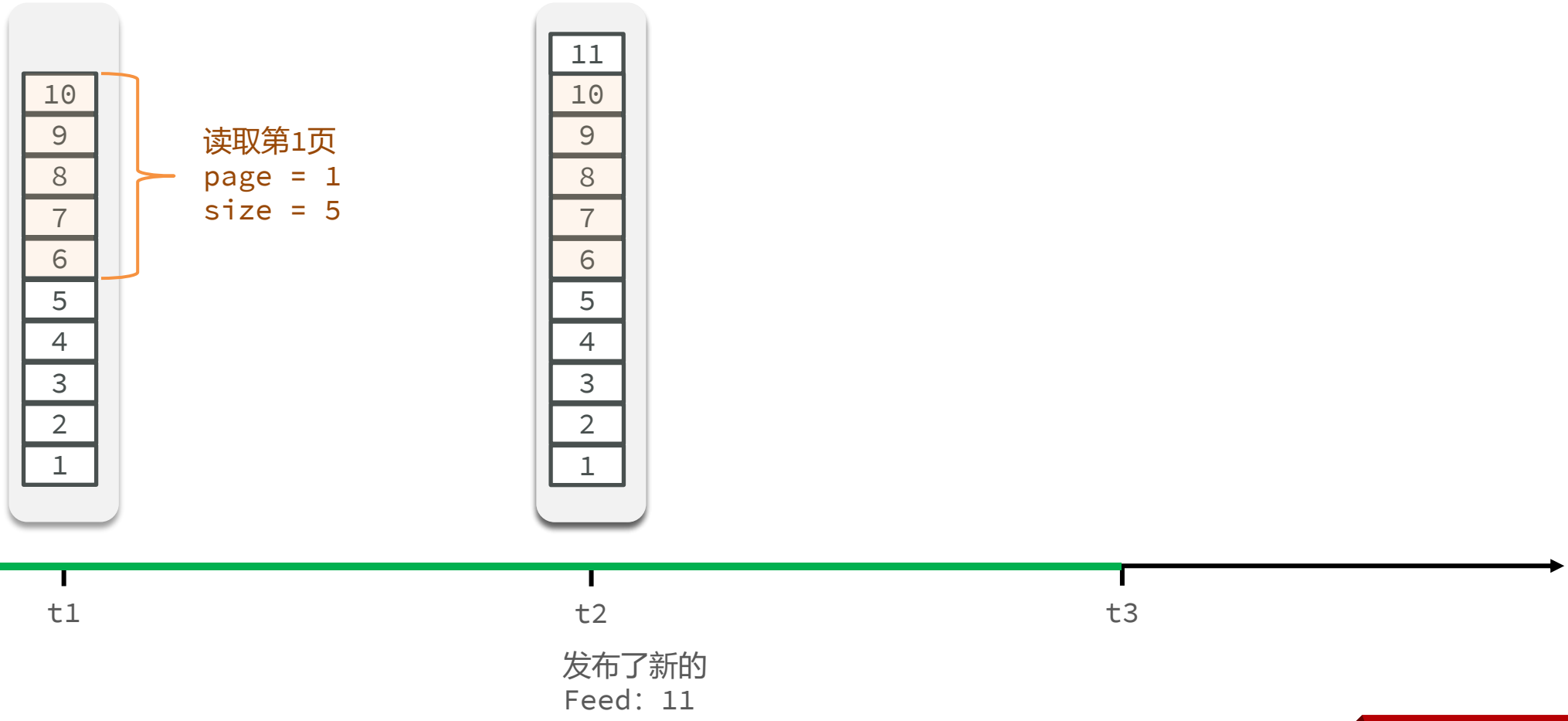
Feed流的分页问题

Feed流中的数据会不断更新，所以数据的角标也在变化，因此不能采用传统的分页模式。



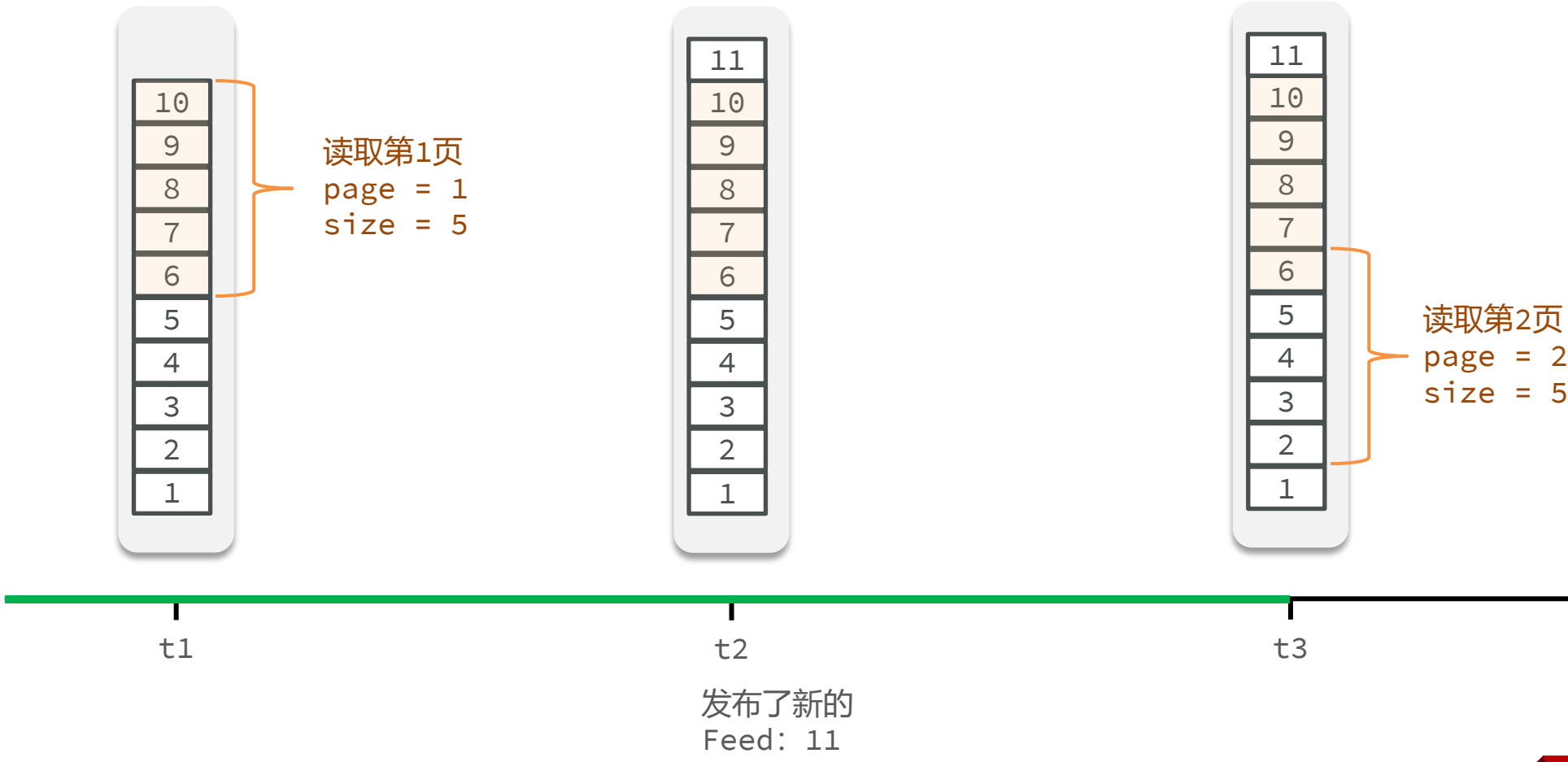
Feed流的分页问题

Feed流中的数据会不断更新，所以数据的角标也在变化，因此不能采用传统的分页模式。



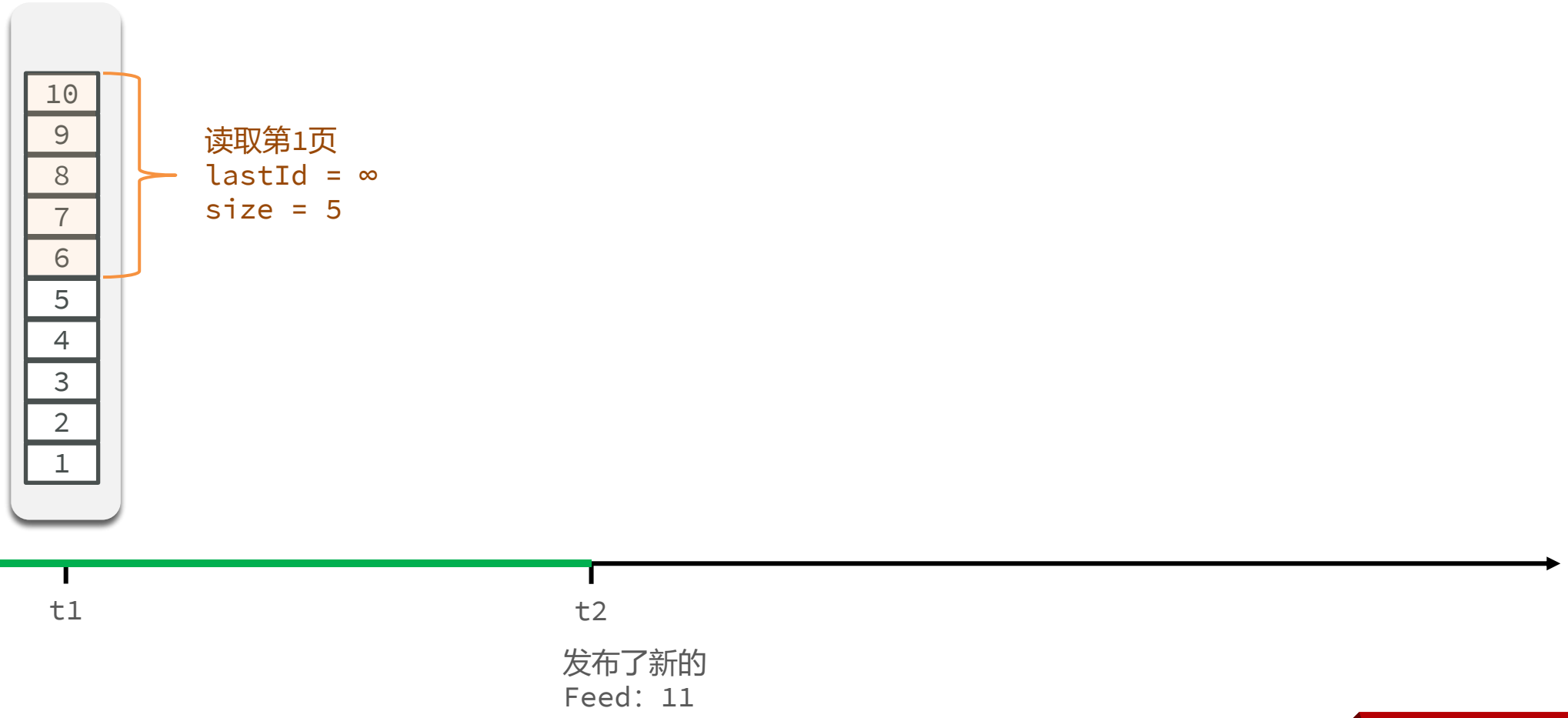
Feed流的分页问题

Feed流中的数据会不断更新，所以数据的角标也在变化，因此不能采用传统的分页模式。



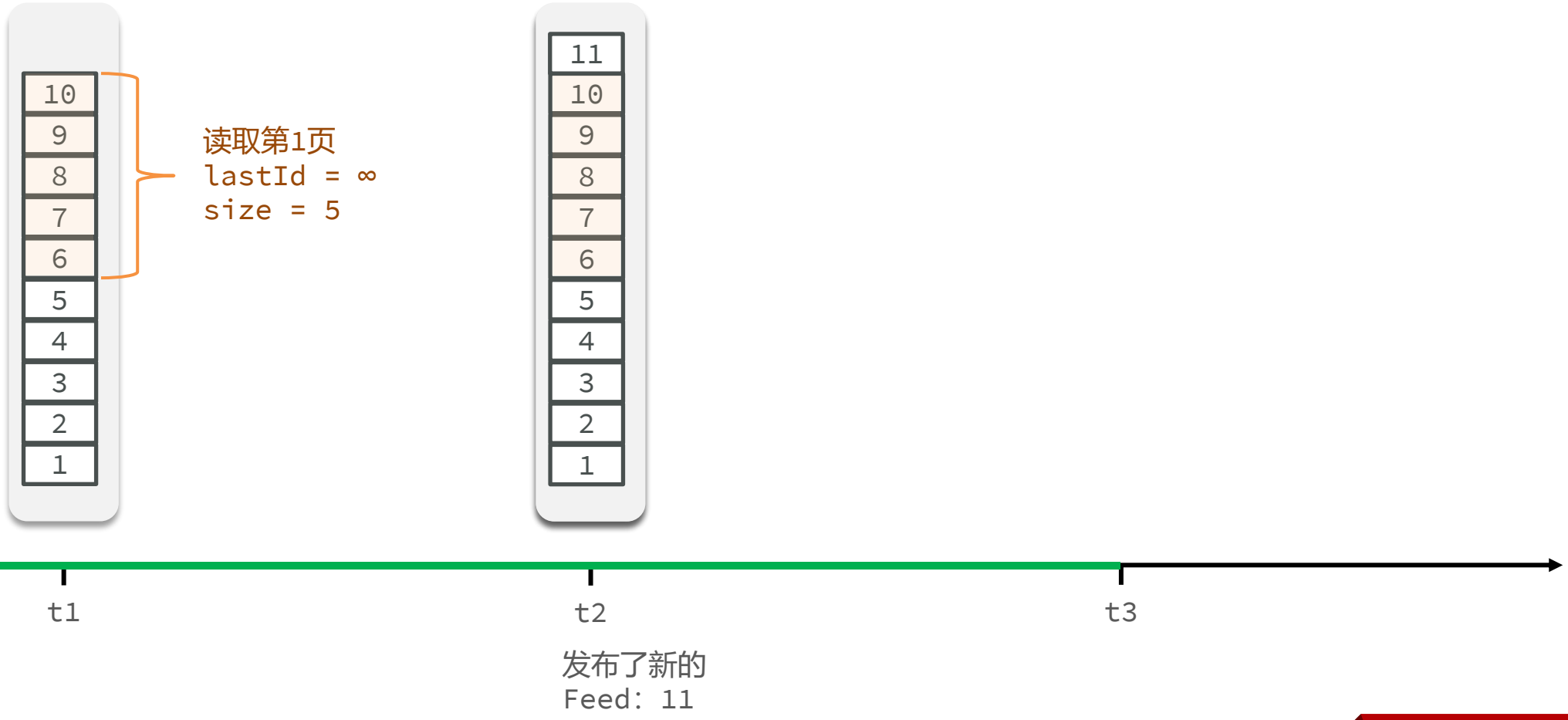
Feed流的滚动分页

Feed流中的数据会不断更新，所以数据的角标也在变化，因此不能采用传统的分页模式。



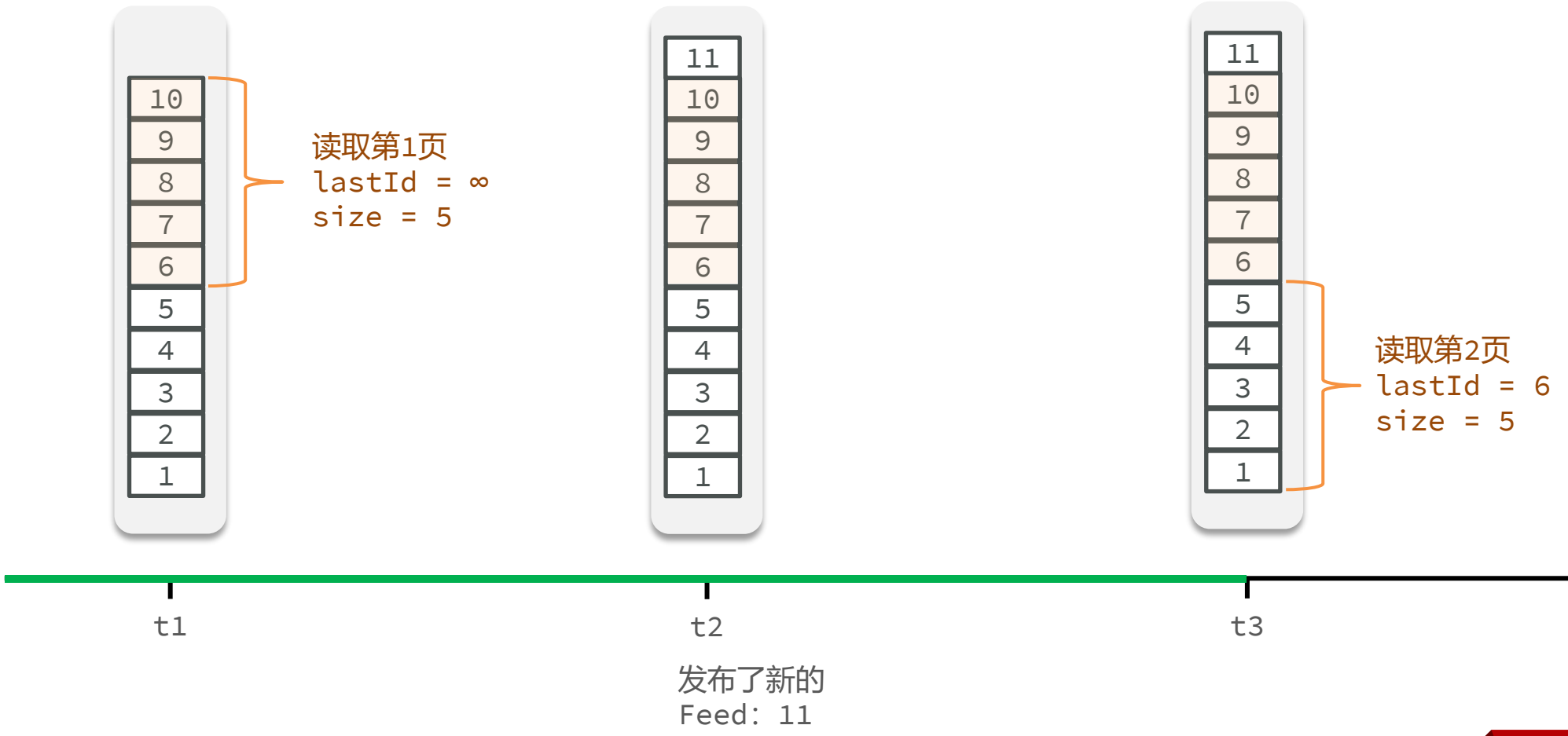
Feed流的滚动分页

Feed流中的数据会不断更新，所以数据的角标也在变化，因此不能采用传统的分页模式。



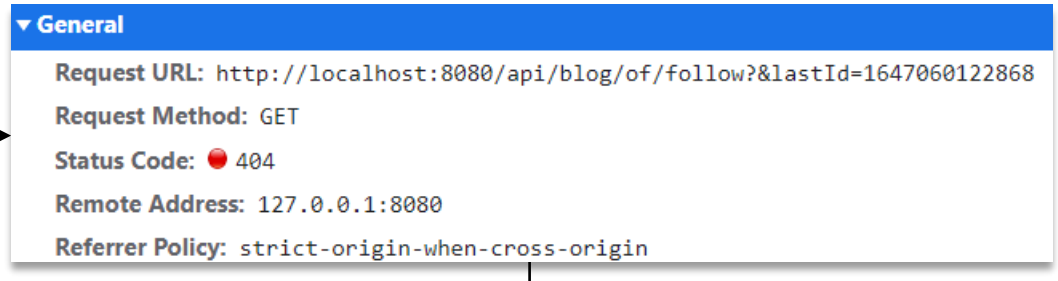
Feed流的滚动分页

Feed流中的数据会不断更新，所以数据的角标也在变化，因此不能采用传统的分页模式。



案例 实现关注推送页面的分页查询

需求：在个人主页的“关注”卡片中，查询并展示推送的Blog信息：



	说明
请求方式	GET
请求路径	/blog/of/follow
请求参数	lastId：上一次查询的最小时间戳 offset：偏移量
返回值	List<Blog>:小于指定时间戳的笔记集合 minTime：本次查询的推送的最小时间戳 offset：偏移量

06 附近商户

目录

Contents

GEO数据结构

GEO就是Geolocation的简写形式，代表地理坐标。Redis在3.2版本中加入了对GEO的支持，允许存储地理坐标信息，帮助我们根据经纬度来检索数据。常见的命令有：

GEOADD：添加一个地理空间信息，包含：经度（longitude）、纬度（latitude）、值（member）

GEODIST：计算指定的两个点之间的距离并返回

GEOHASH：将指定member的坐标转为hash字符串形式并返回

GEOPOS：返回指定member的坐标

GEORADIUS：指定圆心、半径，找到该圆内包含的所有member，并按照与圆心之间的距离排序后返回。6.2以后已废弃

GEOSEARCH：在指定范围内搜索member，并按照与指定点之间的距离排序后返回。范围可以是圆形或矩形。6.2.新功能

GEOSEARCHSTORE：与GEOSEARCH功能一致，不过可以把结果存储到一个指定的key。 6.2.新功能

案例 练习Redis的GEO功能

需求：

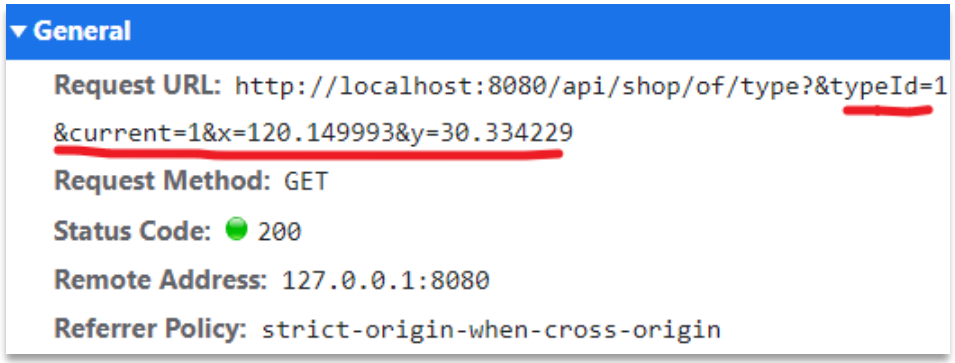
1. 添加下面几条数据：
   - 北京南站（116.378248 39.865275）
   - 北京站（116.42803 39.903738）
   - 北京西站（116.322287 39.893729）
2. 计算北京西站到北京站的距离
3. 搜索天安门（116.397904 39.909005）附近10km内的所有火车站，并按照距离升序排序

目录

Contents

附近商户搜索

在首页中点击某个频道，即可看到频道下的商户：



	说明
请求方式	GET
请求路径	/shop/of/type
请求参数	typeId：商户类型 current：页码，滚动查询 x：经度 y：纬度
返回值	List<Shop>:符合要求的商户信息

附近商户搜索

按照商户类型做分组，类型相同的商户作为同一组，以typeId为key存入同一个GEO集合中即可

Key	Value	Score
shop:geo:美食		4069152240174578
		4069879450313142
shop:geo:KTV		4069885469876391
		4069885424176331

附近商户搜索

SpringDataRedis的2.3.9版本并不支持Redis 6.2提供的GEOSEARCH命令，因此我们需要提示其版本，修改自己的POM文件，内容如下：

```
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-data-redis</artifactId>
    <exclusions>
        <exclusion>
            <groupId>org.springframework.data</groupId>
            <artifactId>spring-data-redis</artifactId>
        </exclusion>
        <exclusion>
            <artifactId>lettuce-core</artifactId>
            <groupId>io.lettuce</groupId>
        </exclusion>
    </exclusions>
</dependency>
<dependency>
    <groupId>org.springframework.data</groupId>
    <artifactId>spring-data-redis</artifactId>
    <version>2.6.2</version>
</dependency>
<dependency>
    <artifactId>lettuce-core</artifactId>
    <groupId>io.lettuce</groupId>
    <version>6.1.6.RELEASE</version>
</dependency>
```

07 用户签到

目录

Contents

BitMap用法

假如我们用一张表来存储用户签到信息，其结构应该如下：

```
CREATE TABLE `tb_sign` (
  `id` bigint(20) unsigned NOT NULL AUTO_INCREMENT COMMENT '主键',
  `user_id` bigint(20) unsigned NOT NULL COMMENT '用户id',
  `year` year(4) NOT NULL COMMENT '签到的年',
  `month` tinyint(2) NOT NULL COMMENT '签到的月',
  `date` date NOT NULL COMMENT '签到的日期',
  `is_backup` tinyint(1) unsigned DEFAULT NULL COMMENT '是否补签',
  PRIMARY KEY (`id`)
) ENGINE=InnoDB DEFAULT CHARSET=utf8mb4;
```

假如有1000万用户，平均每人每年签到次数为10次，则这张表一年的数据量为 1亿条

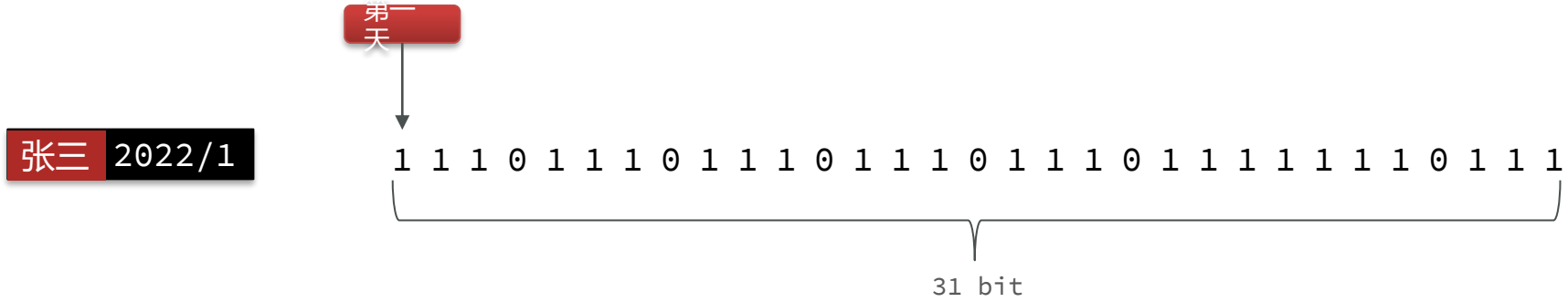
每签到一次需要使用（8 + 8 + 1 + 1 + 3 + 1）共22 字节的内存，一个月则最多需要600多字节

BitMap用法

我们按月来统计用户签到信息，签到记录为1，未签到则记录为0.

BitMap用法

我们按月来统计用户签到信息，签到记录为1，未签到则记录为0.



课程	签到	课程	签到	课程	签到
1	1	11	1	21	1
2	1	12	0	22	1
3	1	13	1	23	1
4	0	14	1	24	1
5	1	15	1	25	1
6	1	16	0	26	1
7	1	17	1	27	1
8	0	18	1	28	0
9	1	19	1	29	1
10	1	20	0	30	1

把每一个bit位对应当月的每一天，形成了映射关系。用0和1标示业务状态，这种思路就称为**位图**

Redis中是利用string类型数据结构实现**BitMap，**因此最大上限是512M，转换为bit则是 2^32个bit位。

BitMap用法

Redis中是利用string类型数据结构实现**BitMap，**因此最大上限是512M，转换为bit则是 2^32个bit位。

BitMap的操作命令有：

SETBIT：向指定位置（offset）存入一个0或1

GETBIT ： 获取指定位置（offset）的bit值

BITCOUNT ： 统计BitMap中值为1的bit位的数量

BITFIELD ： 操作（查询、修改、自增）BitMap中bit数组中的指定位置（offset）的值

BITFIELD_RO ： 获取BitMap中bit数组，并以十进制形式返回

BITOP ： 将多个BitMap的结果做位运算（与 、或、异或）

BITPOS ： 查找bit数组中指定范围内第一个0或1出现的位置

目录

Contents

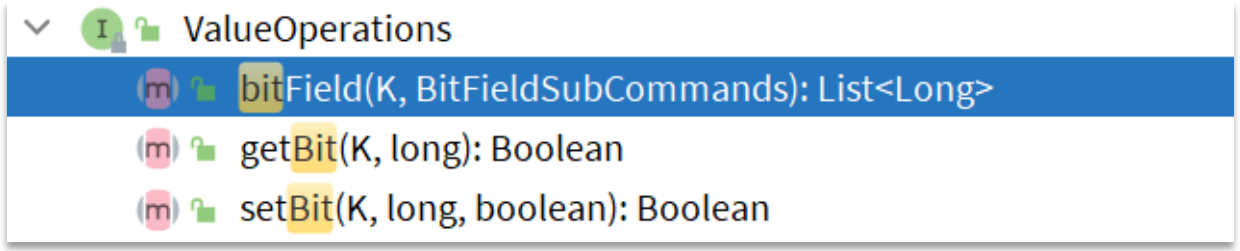
案例

签到功能

需求：实现签到接口，将当前用户当天签到信息保存到Redis中

	说明
请求方式	Post
请求路径	/user/sign
请求参数	无
返回值	无

提示：因为BitMap底层是基于String数据结构，因此其操作也都封装在字符串相关操作中了。



目录

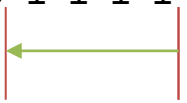
Contents

签到统计

问题1：什么叫做连续签到天数?

从最后一次签到开始向前统计，直到遇到第一次未签到为止，计算总的签到次数，就是连续签到天数。

1 1 1 0 0 0 1 1 0 1 1 0 0 0 1 0 1 1 1 0 1 1 1 1 1 0 1 1 1 1



问题2：如何得到本月到今天为止的所有签到数据?

BITFIELD key GET u[dayOfMonth] 0

```
1 0 1 1 1 1
        1
-----------
        1
```

问题3：如何从后向前遍历每个bit位?

与 1 做与运算，就能得到最后一个bit位。

随后右移1位，下一个bit位就成为了最后一个bit位。

案例 实现签到统计功能

需求：实现下面接口，统计当前用户截止当前时间在本月的连续签到天数

	说明
请求方式	GET
请求路径	/user/sign/count
请求参数	无
返回值	连续签到天数

08 UV统计

目录

Contents

HyperLogLog用法

首先我们搞懂两个概念：

- **UV**：全称**U**nique **V**isitor，也叫独立访客量，是指通过互联网访问、浏览这个网页的自然人。1天内同一个用户多次访问该网站，只记录1次。
- **PV**：全称**P**age **V**iew，也叫页面访问量或点击量，用户每访问网站的一个页面，记录1次PV，用户多次打开页面，则记录多次PV。往往用来衡量网站的流量。

UV统计在服务端做会比较麻烦，因为要判断该用户是否已经统计过了，需要将统计过的用户信息保存。但是如果每个访问的用户都保存到Redis中，数据量会非常恐怖。

HyperLogLog用法

Hyperloglog(HLL)是从Loglog算法派生的概率算法，用于确定非常大的集合的基数，而不需要存储其所有值。相关算法原理大家可以参考：https://juejin.cn/post/6844903785744056333#heading-0

Redis中的HLL是基于string结构实现的，单个HLL的内存永远小于16kb，内存占用低的令人发指！作为代价，其测量结果是概率性的，有小于0.81%的误差。不过对于UV统计来说，这完全可以忽略。

```
PFADD key element [element ...]
summary: Adds the specified elements to the specified HyperLogLog.
since: 2.8.9

PFCOUNT key [key ...]
summary: Return the approximated cardinality of the set(s) observed by
since: 2.8.9

PFMERGE destkey sourcekey [sourcekey ...]
summary: Merge N different HyperLogLogs into a single one.
since: 2.8.9
```

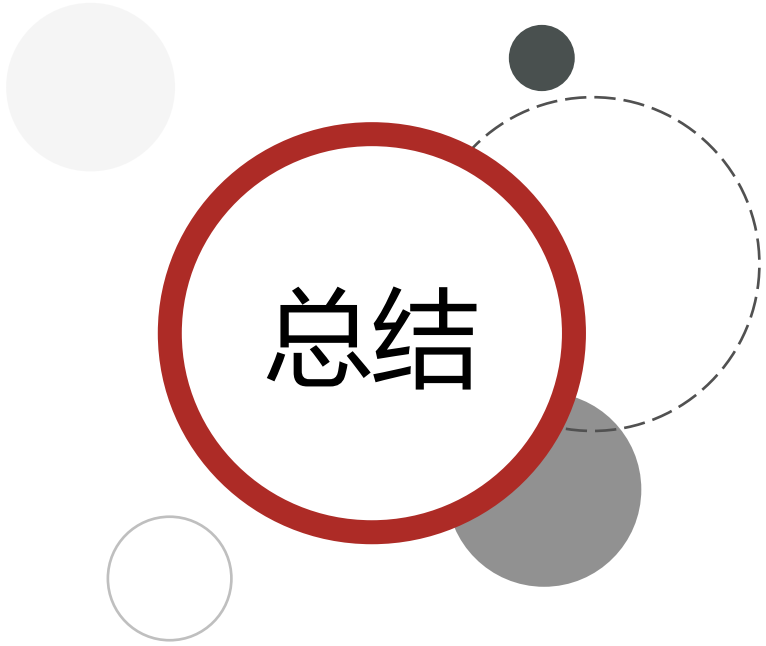
目录

Contents

实现UV统计

我们直接利用单元测试，向HyperLogLog中添加100万条数据，看看内存占用和统计效果如何：

```
@Test
void testHyperLogLog() {
    // 准备数组，装用户数据
    String[] users = new String[1000];
    // 数组角标
    int index = 0;
    for (int i = 1; i <= 1000000; i++) {
        // 赋值
        users[index++] = "user_" + i;
        // 每1000条发送一次
        if (i % 1000 == 0) {
            index = 0;
            stringRedisTemplate.opsForHyperLogLog().add("hll1", users);
        }
    }
    // 统计数量
    Long size = stringRedisTemplate.opsForHyperLogLog().size("hll1");
    System.out.println("size = " + size);
}
```



HyperLogLog的作用：

- 做海量数据的统计工作

HyperLogLog的优点：

- 内存占用极低
- 性能非常好

HyperLogLog的缺点：

- 有一定的误差

传智教育旗下高端IT教育品牌