

# Redis

xbZhong

2025-09-25

## Contents

Redis . . . . .	1
实战 . . . . .	8
高级 . . . . .	28
原理 . . . . .	40
最佳实践 . . . . .	67

本页 PDF

## Redis

Redis 是键值数据库，属于 NoSQL 数据库

特征：

- 键值型
- 单线程，核心命令原子性
- 低延迟，速度快（基于内存）
- 支持数据持久化，会定期把数据从内存存储进磁盘
- 支持多语言客户端

## Redis 配置文件

```
## 允许访问的地址，默认为 127.0.0.1，导致只能在本地访问，修改为 0.0.0.0 则可以在任意 IP 访问
bind 0.0.0.0
## 守护进程，修改为 yes 后可在后台运行
daemonize yes
## 密码，设置后访问 Redis 需输入密码
requirepass 123456
## 监听的端口
port 6379
## 工作目录
dir .
## 数据库数量，设置为 1 代表只使用 1 个库，默认有 16 个库，编号为 0-15
databases 1
## 设置 Redis 能够使用的最大内存
maxmemory 512mb
## 日志文件，默认为空，不记录日志
logfile "redis.log"
```

## 命令行客户端

```
redis-cli [options] [commands]
```

常见的 options 有：

- -h：指定要连接的 redis 节点的 IP 地址
- -p：指定要连接的 redis 节点的端口
- -a：指定 redis 的访问密码

commands 是 redis 的操作命令

- ping：与 redis 服务端做心跳测试，服务端正常会返回 pong
- help：可以使用 help commands 获取操作指引

## 常见命令

- KEYS：查看符合模板的所有 key
- DEL：删除一个指定的 key，返回值是被删除 key 的数量
- EXISTS：判断 key 是否存在
- EXPIRE：给 key 设置有效期，有效期到期时 key 会被自动删除，**单位为秒**
- TTL：查看 key 剩余的有效期
  - 返回值为-1 代表永久有效
  - 返回值为-2 代表 key 已经过期

## 模糊查询

- \*：匹配任意数量字符
- ?: 匹配任意单个字符
  - 在生产环境中不推荐使用，因为这种模糊查询**在数据量庞大的情况下会阻塞主线程**

## 常见数据结构

redis 是一个 key-value 数据库，key 一般是 String 类型，不过 value 的**类型多样**：

- String
- Hash
- List
- Set
- SortedSet
- GEO
- BitMap
- HyperLog

**Redis 的 key** Redis 的 key 允许有多个单词形成层级结构，多个单词之间用 ':' 隔开，如下项目名：业务名：类型:id

### String 最大空间不能超过 512MB

根据字符串格式不同，可以分为 3 类，但它们底层都是**字节数组形式存储**

- string：普通字符串

- int: 整数类型
- float: 浮点数类型

## 常见命令

- SET: 添加或者修改已经存在的一个 String 类型的键值对
- GET: 根据 key 获取 String 类型的 value
- MSET: 批量添加多个 String 类型的键值对
- MGET: 根据多个 key 获取多个 String 类型 value
- INCR: 让一个整型的 key 自增 1
- INCRBY: 让一个整型的 key 自增并指定步长, 如 INCRBY num 2
- INCRBYFLOAT: 让一个浮点型的 key 自增并指定步长
- SETNX: 添加一个 String 类型的键值对, 前提是这个 key 不存在, 否则不执行
- SETEX: 添加一个 String 类型的键值对, 并指定有效期

**Hash** 也叫做散列, 其 value 是一个无序字典, 类似 Java 中的 HashMap, 如图所示

KEY	VALUE	
	field	value
heima:user:1	name	Jack
	age	21
heima:user:2	name	Rose
	age	18

Figure 1: image-20250907134423516

## 常见命令

- HSET key field value: 添加或修改 hash 类型 key 的 field 的值
- HGET key field: 获取一个 hash 类型的 key 的 field 的值
- HMSET: 批量添加多个 hash 类型的 key 的 field 的值
- HMGET: 批量获取多个 hash 类型的 key 的 field 的值
- HGETALL: 获取一个 hash 类型的 key 中的所有 field 和 value
- HKEYS: 获取一个 hash 类型的 key 中的所有 key
- HVALS: 获取一个 hash 类型的 key 中的所有 value
- HINCRBY: 让一个 hash 类型的 key 的字段值自增并指定步长
- HSETNX: 添加一个 hash 类型的 key 的 field 值, 前提是这个 field 不存在

**List** 和 Java 中的 `LinkedList` 类似, 可以看作是一个双向链表结构

特征:

- 有序

- 元素可重复
- 插入和删除快
- 查询速度一般

## 常见命令

- LPUSH key element: 向列表左侧插入一个或多个元素
- LPOP key: 移除并返回列表左侧的第一个元素，没有则返回 null
- RPUSH key element: 向列表右侧插入一个或多个元素
- RPOP key: 移除并返回列表右侧的第一个元素，没有则返回 null
- LRANGE key start end: 返回一段角标范围内的所有元素，闭区间，start 为起始角标，end 为末尾角标
- BLPOP 和 BRPOP: 与 LPOP 和 RPOP 类似，在没有元素时等待指定时间，而不是直接返回 NULL
  - 后面跟数字，单位是秒，为等待的时间

**Set** 与 java 中的 HashSet 类似，可以看作是一个 value 为 null 的 HashMap

## 特征：

- 无序
- 元素不可重复
- 查找快
- 支持交集、并集、差集等功能

## 常见命令：

- SADD key member: 向 set 中添加一个或多个元素
- SREM key member: 移除 set 中的指定元素
- SCARD key: 返回 set 中元素的个数
- SISMEMBER: 判断一个元素是否存在与 set 中
- SMEMBERS: 返回 set 集合中的所有元素
- SINTER key1 key2: 求 key1 和 key2 的交集
- SDIFF key1 key2: 求 key1 和 key2 的差集
- SUNION key1 key2: 求 key1 和 key2 的并集

**SortedSet** 每一个元素都带有一个 score 属性，可以基于 score 属性对元素进行排序，底层的实现是一个跳表 + 哈希表

## 特性：

- 可排序
- 元素不重复
- 查询速度快

## 常见命令：

- ZADD key score member: 添加一个或多个元素到 sortedset (需要添加 score 值)，如果已经存在则更新其 score 值
- ZREM key member: 删除 sortedset 中的一个指定元素
- ZSCORE key member: 获取 sortedset 中指定元素的 score 值
- ZRANK key member: 获取 sortedset 中指定元素的排名
- ZCARD key: 获取 sortedset 中的元素个数
- ZCOUNT key min max: 统计 score 值在给定范围内的所有元素的个数

- ZINCRBY key increment member: 让 sortedset 中的指定元素的 score 自增, 步长为指定的 **increment** 值
- ZRANGE key min max: 按照 score 排序后, 获取**指定排名范围内的元素**, min 和 max 是起始排名和末尾排名
- ZRANGEBYSCORE key min max: 按照 score 排序后, 获取**指定 score 范围内的元素**, min 和 max 是起始分数和末尾分数
- ZDIFF、ZINTER、ZUNION: 求差集、交集、并集

注意: 所有的排名都是默认升序, 如果要降序则在命令的 Z 后面添加 REV 即可

## Jedis

一个 Java 形式的 Redis 客户端

使用步骤:

1. 引入依赖

```
<dependency>
    <groupId>redis.clients</groupId>
    <artifactId>jedis</artifactId>
    <version>3.7.0</version>
</dependency>
```

2. 建立连接

```
private Jedis jedis;

void setup(){
    // 建立连接
    jedis = new Jedis("ip", 端口号);
    // 设置密码
    jedis.auth(密码);
    // 选择数据库
    jedis.select(0);
}
```

3. 进行使用

4. 释放资源

```
void tearDown(){
    if(jedis != null){
        // 关闭连接
        jedis.close();
    }
}
```

**Jedis 连接池** Jedis 本身是线程不安全的, 并且频繁创建和销毁线程会有性能损耗, 因此应该使用 Jedis 连接池代替 Jedis 的直连方式

```
public class JedisConnectionFactory{
    private static final JedisPool jedisPool;

    static{
```

```

// 配置连接池
JedisPoolConfig poolConfig = new JedisPoolConfig();
// 最大连接数
poolConfig.setMaxTotal(8);
// 最大空闲连接
poolConfig.setMaxIdle(8);
// 最小空闲连接
poolConfig.setMinIdle(0);
// 最大等待时间
poolConfig.setMaxWaitMillis(1000);
// 创建连接对象
jedisPool = new JedisPool(poolConfig,主机名,端口,超时时间,密码);
}

public static Jedis getJedis(){
    return jedisPool.getResource();
}
}

```

## Spring Data Redis

是 Spring 中数据操作的模块，包含对各种数据库的集成

- 底层是基于 Letture 和 Jedis 的
- 支持 Redis 哨兵和集群以及发布订阅模型
- 支持基于 Letture 的响应式编程

### 常用 API

- redisTemplate.opsForValue(): 操作 String 类型数据
- redisTemplate.opsForHash(): 操作 Hash 类型数据
- redisTemplate.opsForList(): 操作 List 类型数据
- redisTemplate.opsForSet(): 操作 Set 类型数据
- redisTemplate.opsForZSet(): 操作 SortedSet 类型数据
- redisTemplate: 通用命令
  - 可以在存入数据的时候设置有效期，需要通过 **TimeUnit** 指定单位
  - 也可以通过.expire 设置有效期，里面要填入键名，并需要通过 TimeUnit 指定单位

### 使用步骤

1. 引入依赖：SpringBoot 默认装配了 SpringDataRedis，默认使用的连接池是 Letture

```

<!--连接池依赖-->
<dependency>
    <groupId>org.apache.commons</groupId>
    <artifactId>commons-pool2</artifactId>
</dependency>

```

2. 引入配置

```
spring:  
  redis:  
    host:  
    port:  
    password:  
    lettuce:  
      pool:  
        max-active:  
        max-idle:  
        min-idle:  
        max-wait:
```

### 3. 注入 RedisTemplate

```
@Autowired  
private RedisTemplate redisTemplate;
```

### 4. 进行使用

**序列化** RedisTemplate 可以接收任意 Object 作为值写入 Redis，写入前会把 Object 序列化成字节形式（默认采用 JDK 序列化），得到的结果如下

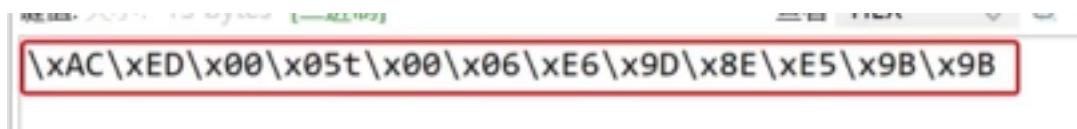


Figure 2: image-20250908160754439

### 缺点：

- 可读性差
- 内存占用较大

### 解决方法：改变 RedisTemplate 的序列化方式

- key 使用 StringRedisSerializer 序列化器进行 String 序列化
- value 使用 GenericJackson2JsonRedisSerializer 序列化器进行 Json 反序列化
  - 使用这种方式在存储的时候需要在 Redis 里面额外存储类的路径，进行反序列化的时候才能根据这个路径找到类的字节码文件从而进行反序列化
  - 为了节省内存空间，并不会使用 Json 序列化器来处理 value，而是统一使用 String 序列化器，要求只存储 String 类型的 key 和 value，需要存储 Java 对象的时候手动进行对象的序列化和反序列化

```
// 依赖注入  
@Autowired  
private StringRedisTemplate stringRedisTemplate;  
// Json 工具  
private String final ObjectMapper mapper = new ObjectMapper();  
void testStringTemplate(){  
    // 准备对象  
    User user = new User(" 虎哥", 18);
```

```

// 手动序列化
String json = mapper.writeValueAsString(user);
// 导入一条数据到 redis
stringRedisTemplate.opsForValue().set("user:200",json);
// 读取数据
String val = stringRedisTemplate.opsForValue().get("user:200");
// 反序列化
User user1 = mapper.readValue(val,User.class);
}

```

- Spring 默认提供了一个 StringRedisTemplate 类，它的 key 和 value 的序列化方式默认就是 String 方式，不用我们自定义序列化方式

## 操作 Hash 类型

- .put(): 插入数据
- .entries(): 获取 value 的全部键值对

## 实战

缓存更新策略一般采用**主动更新策略**，即程序员自己编写业务逻辑，**修改数据库的同时更新缓存**

**主动更新策略有三种实现方式：**

- 缓存的调用者**更新数据库的同时更新缓存**（用的最多的方案!!!）
  - 更新数据库的信息的**同时把缓存删除**，查询时再更新缓存
  - 将缓存和数据库操作**放在一个事务**
  - 先操作数据库再删除缓存!!!**
  - 下图是两种方案的对比，第二种方案发生的可能性较小，因此选择第二种
- 缓存和数据库整合为一个服务，**由这个服务来维护一致性**
- 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**
  - 也就是 CRUD 都在缓存进行，由其它线程去进行数据库的更新

## 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**

### 解决方案

- 缓存空对象**
  - 在 Redis 中存储 Null 并设置 TTL，客户端尝试获取不存在的数据的时候会从 Redis 中拿到 Null
- 布隆过滤**
  - 在客户端和 Redis 中间加一个布隆过滤器
  - 有一定的**穿透风险**，因为它的判断是基于概率的统计，**数据不存在则一定不存在，数据存在则不一定存在**

## 缓存雪崩

是指在同一时段**大量缓存 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：

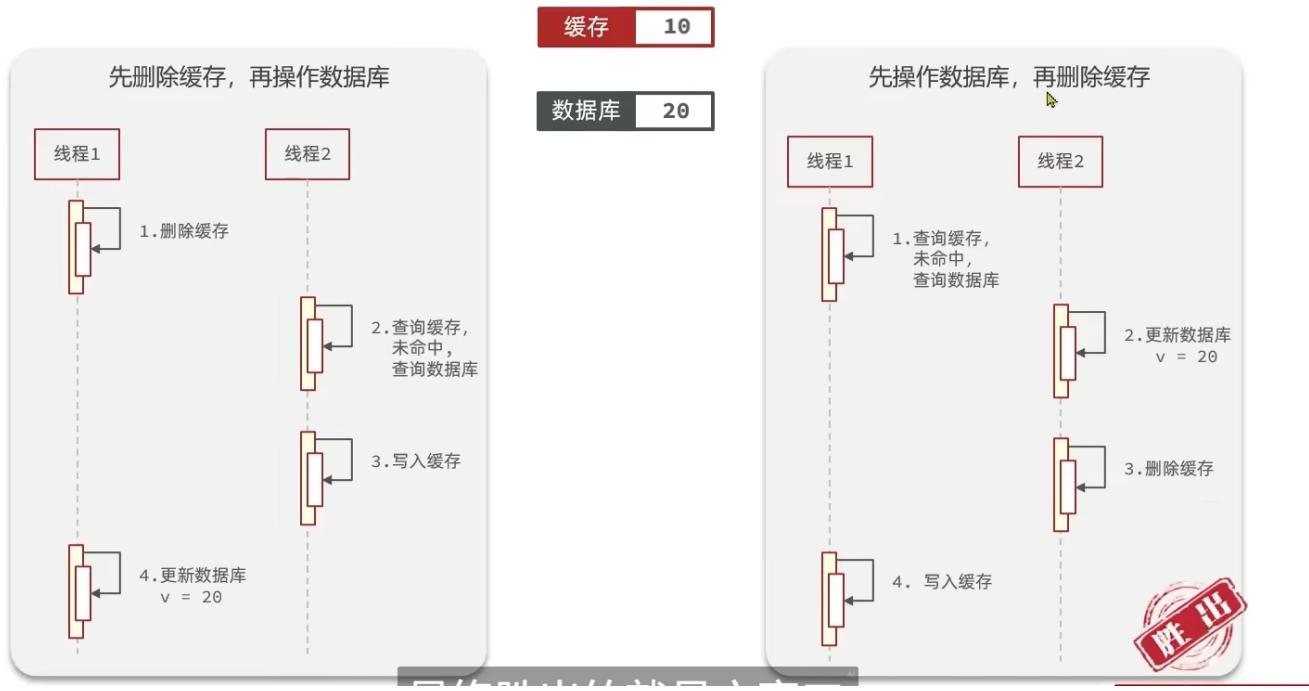


Figure 3: image-20250919122313670

- 给不同的 key 的 TTL 增加随机值
- 给业务添加**多级缓存**
- 给缓存业务添加**降级限流策略**
- 利用 **Redis 集群** 提高服务可用性 (Redis 哨兵模式)

### 缓存击穿

也被称为**热点 key** 问题，就是一个被高并发访问并且缓存重建业务比较复杂的 key 突然失效，无效的请求访问会给数据库带来巨大冲击

- 缓存重建业务就是**缓存未命中**，线程要去请求数据库重新构建缓存这一过程
- 对于热点 key，我们会**提前加入缓存**，如果用户在 redis 找不到数据直接返回 null

### 解决方案

- 互斥锁**
  - 请求数据库时加锁
  - 线程如果尝试获取锁失败，会从头开始，尝试获取缓存中的数据
- 逻辑过期**
  - 设置字段值的时候加上一个**过期时间（不是 TTL）**，**逻辑上进行维护**
  - 线程一进行查询，缓存命中但是已经过期，此时线程一会获取锁并且**开启一个新的线程帮忙去做缓存重建，并且把查询到的旧数据进行返回**
  - 当缓存重建业务还未完成，此时又有新的线程（线程二）来进行数据获取，**线程二尝试获取锁，失败，将查询到的旧数据返回**

**代码** 在写 Redis 的通用工具类的时候，可以使用泛型来实现工具类

- 不知道传入的对象是什么类，使用泛型接收

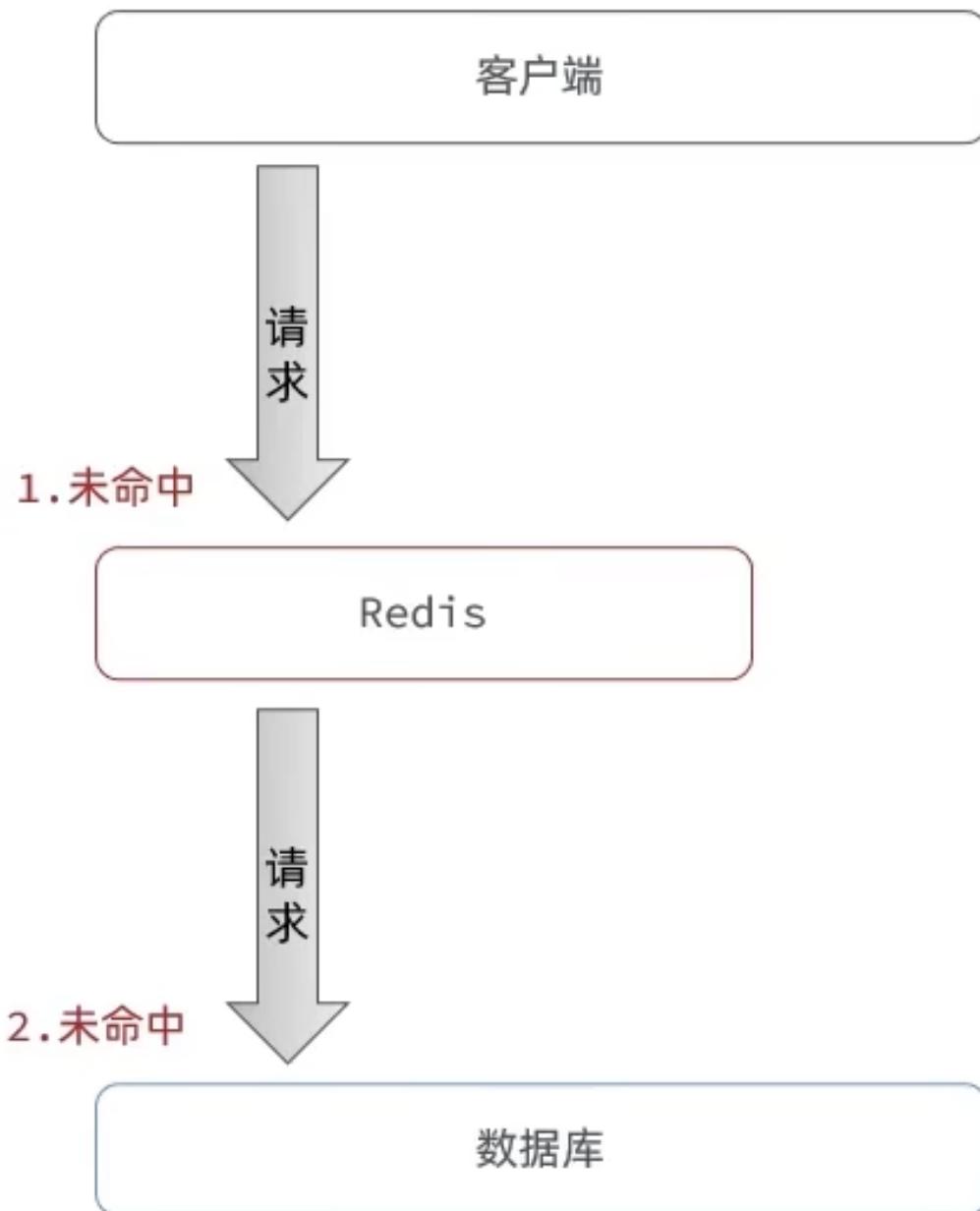


Figure 4: image-20250919125659906

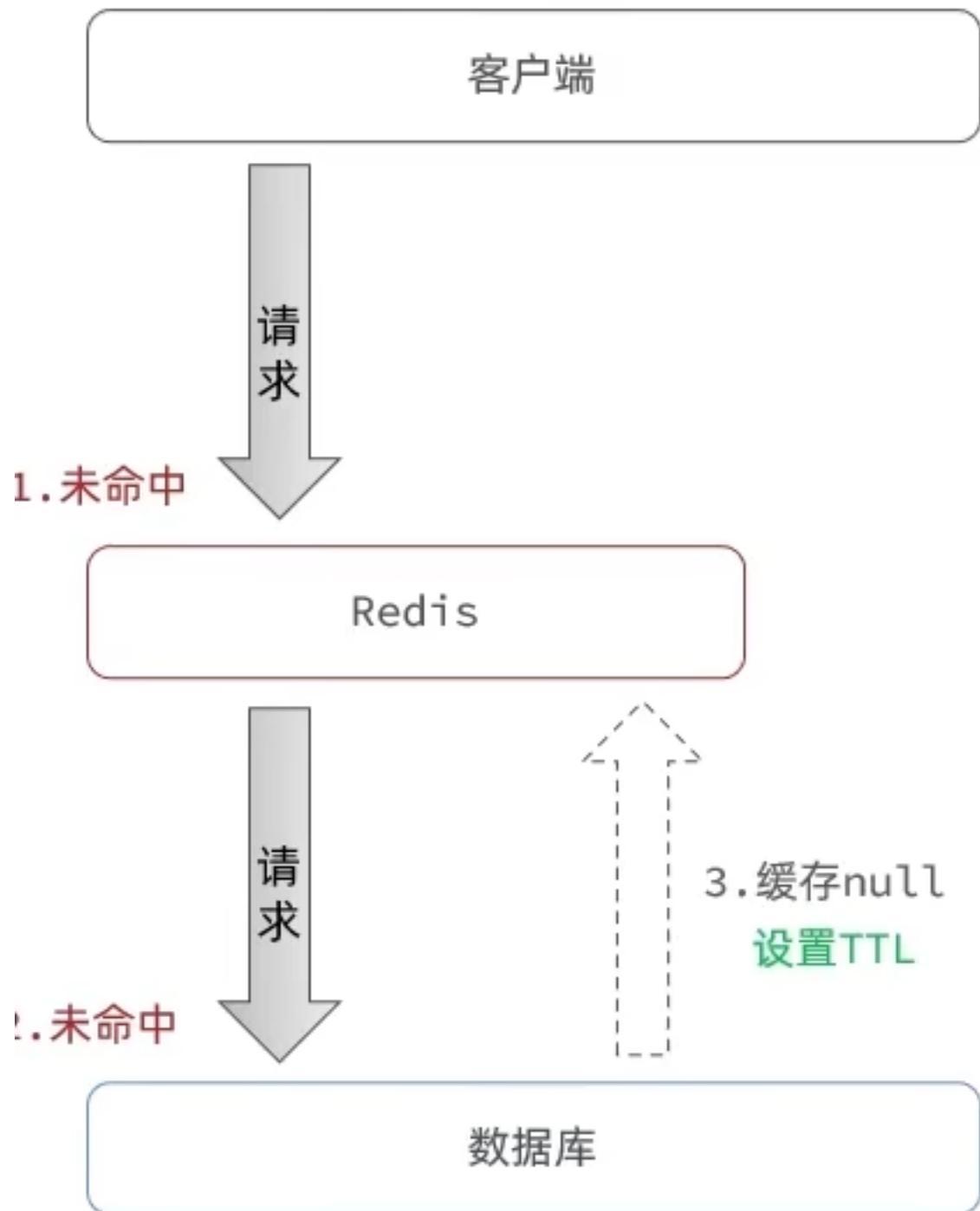


Figure 5: image-20250919125921921

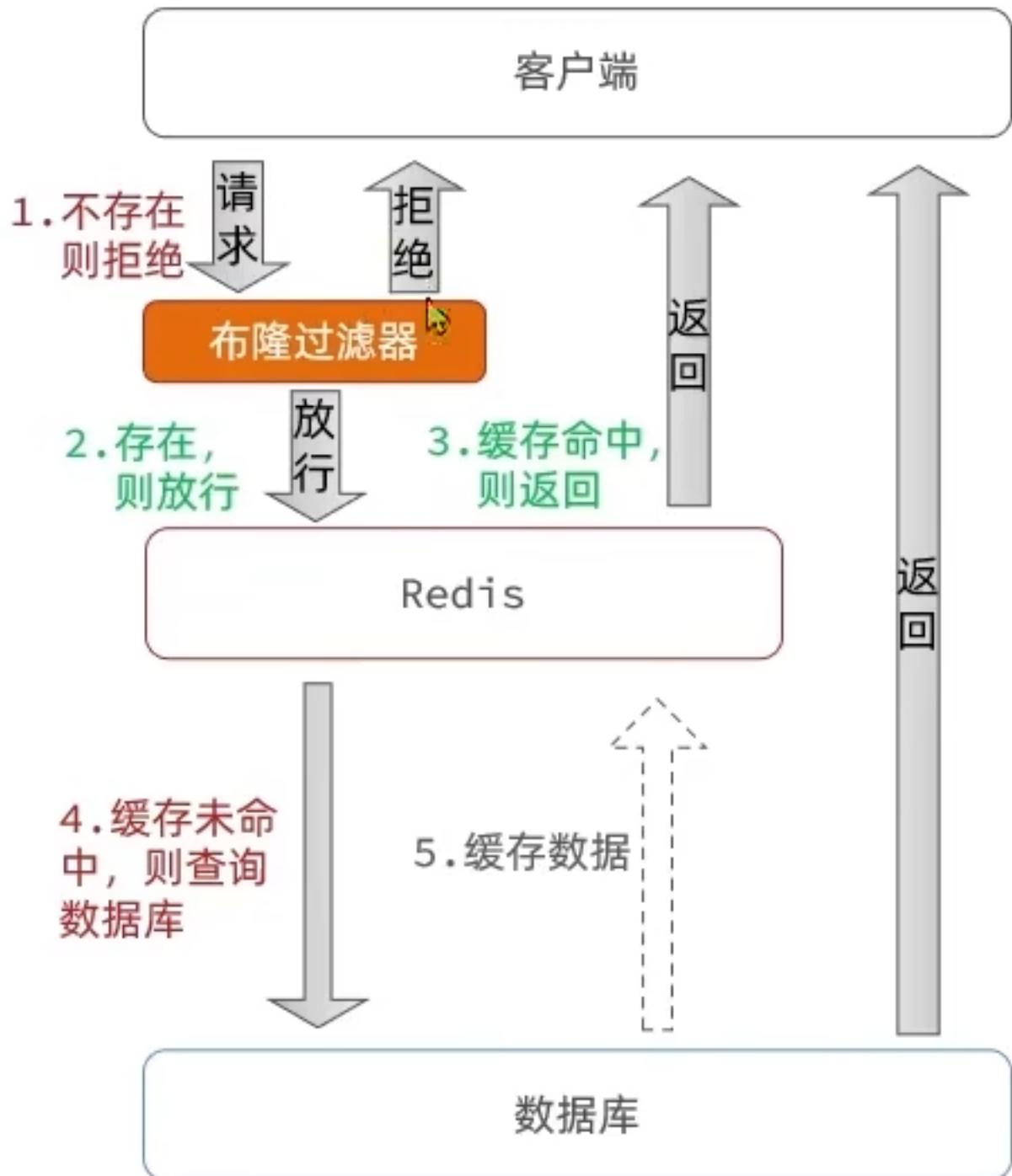


Figure 6: image-20250919130012267

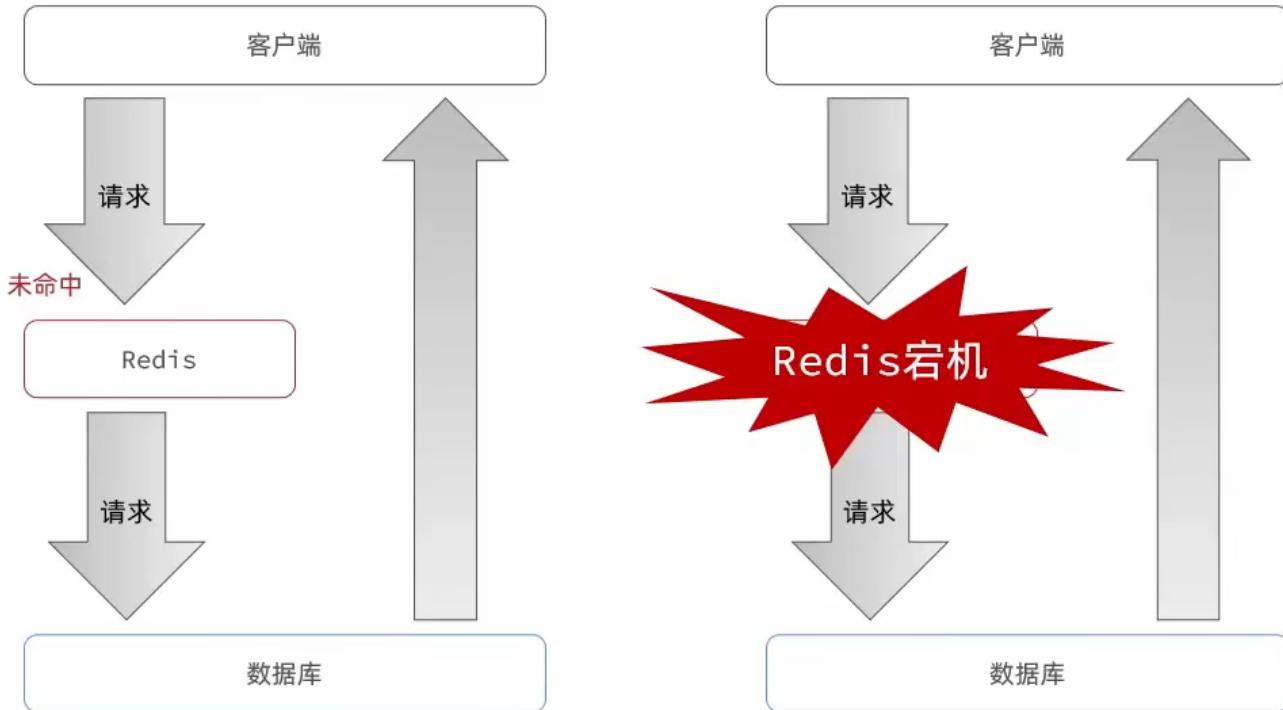


Figure 7: image-20250920100653966

- 不知道传入的 id 是什么类，用泛型接收
- 不知道要对哪个数据库进行操作，用 Function 接收
  - Function<ID,R> 表示传入类型为 ID 的参数，返回类型为 R 的值
  - 在工具类里面用 .apply(参数) 进行方法执行
  - 在调用工具类的地方使用匿名内部类对 Function 进行方法重写（也可以用 lambda 表达式）
- 不知道要反序列化成什么类型，用 Class<R> 泛型接收

```
public <R, ID> R queryWithPassThrough(String keyPrefix, ID id, Class<R> type, Function<ID, R>
    dbFallback, Long time, TimeUnit timeUnit) {
    String key = keyPrefix + id;

    // 尝试从 redis 查询商户缓存
    String json = stringRedisTemplate.opsForValue().get(RedisConstants.CACHE_SHOP_KEY + id);
    // 判断是否存在，不满足这个条件的话要不就是 Null，要不就是空字符串
    if (StrUtil.isNotBlank(json)) {
        // 存在，直接返回
        return JSONUtil.toBean(json, type);
    }
    // 判断是否是空值（空字符串）
    if (json != null) {
        return null;
    }
    // 不存在，去数据库查询
    R r = dbFallback.apply(id);
```

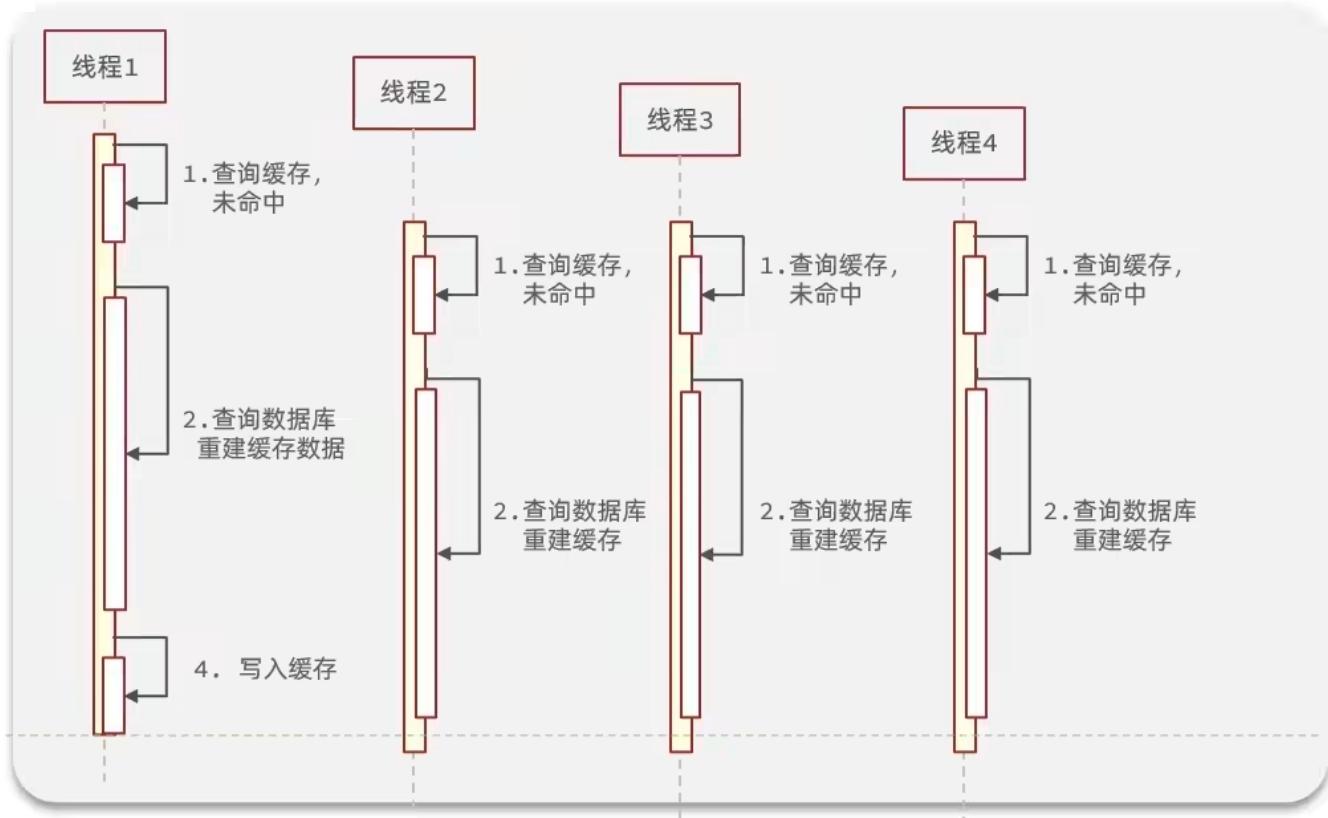


Figure 8: image-20250920102746920

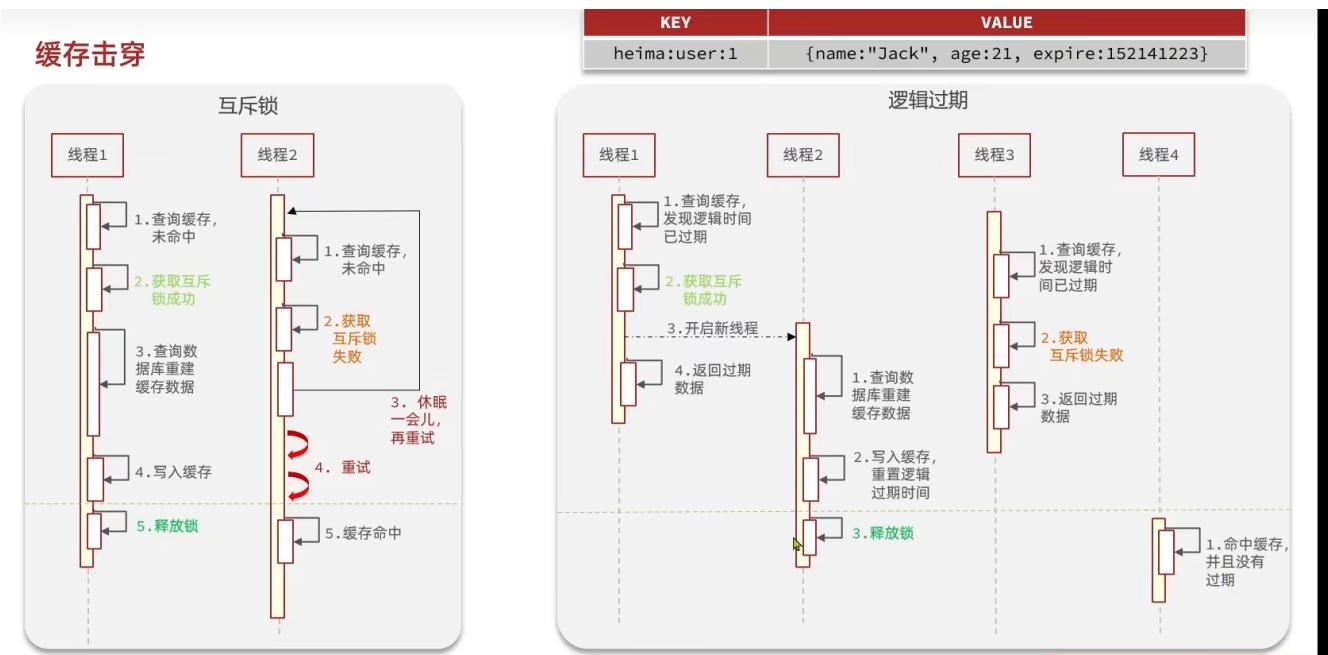


Figure 9: image-20250920102730424

```

// 不存在，数据库查询不到，缓存空字符串并返回错误
if(r==null){
    stringRedisTemplate.opsForValue().set(RedisConstants.CACHE_SHOP_KEY +
→ id,"",RedisConstants.CACHE_NULL_TTL,TimeUnit.MINUTES);
    return null;
}
// 写入 redis
this.set(key, r, time, timeUnit);
return r;
}

```

## 全局唯一 ID 生成器

是用来生成全局唯一 ID 的工具，要满足以下特性：

- 唯一性
- 高可用
- 高性能
- 递增性
- 安全性

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

- 符号位：1bit
- 时间戳：31bit，以秒为单位
- 序列号：32bit，秒内的计数器，支持每秒产生  $2^{32}$  个不同的 ID

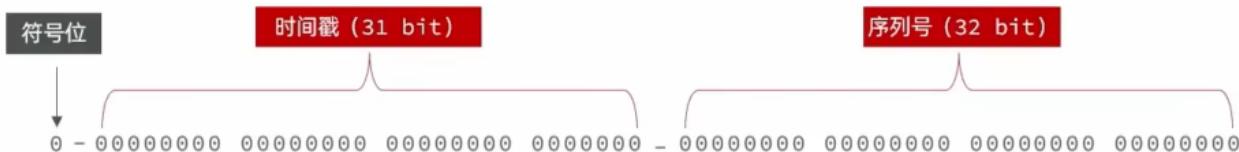


Figure 10: image-20250920123640942

## 分布式锁

集群模式下，**synchronized** 的锁会失效，因为其只能保证一个 JVM 内部的多个线程的互斥

此时要使用**分布式锁**，其满足特性：**分布式系统或集群模式下多进程可见并且互斥的锁**

### 分布式锁的实现方式

**基于 Redis 的分布式锁** 实现分布式锁需要实现的两个基本方法：

- 获取锁
  - 互斥：确保只有一个线程能获取锁
    - SETNX lock thread1
  - 设置超时时间，避免服务宕机导致死锁
    - EXPIRE lock 10

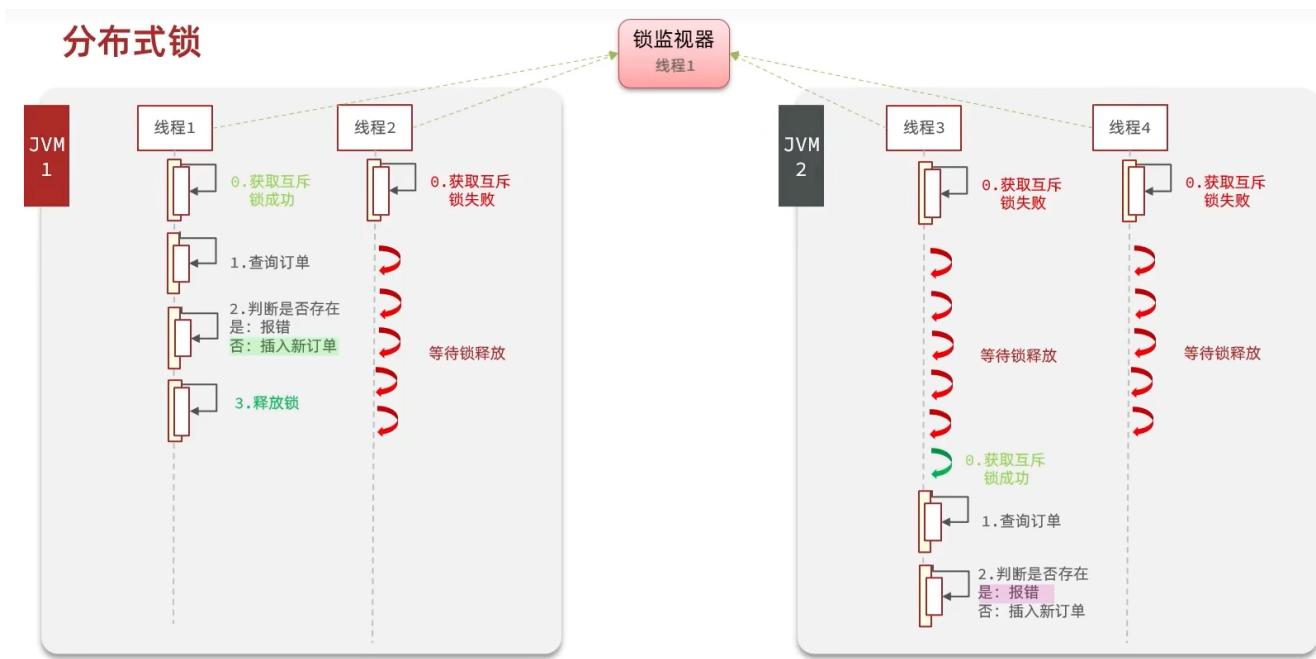


Figure 11: image-20250920155346096

	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

Figure 12: image-20250920160104737

- 同时设置锁和超时时间，实现原子操作
  - SET lock thread1 EX 10 NX
- 释放锁
  - 手动释放
    - DEL key
  - 超时释放

### 误删问题

- 线程 1 持有锁，但是业务阻塞，阻塞时间大于锁的超时时间，导致锁释放
- 线程 2 进入，由于锁超时，此时锁空闲，线程 2 获取锁
- 接着线程 1 业务完成，执行释放锁的逻辑，**把线程 2 持有的锁给释放掉了!!!!**

**线程 1 由于业务阻塞释放了不属于它的锁!!!!**

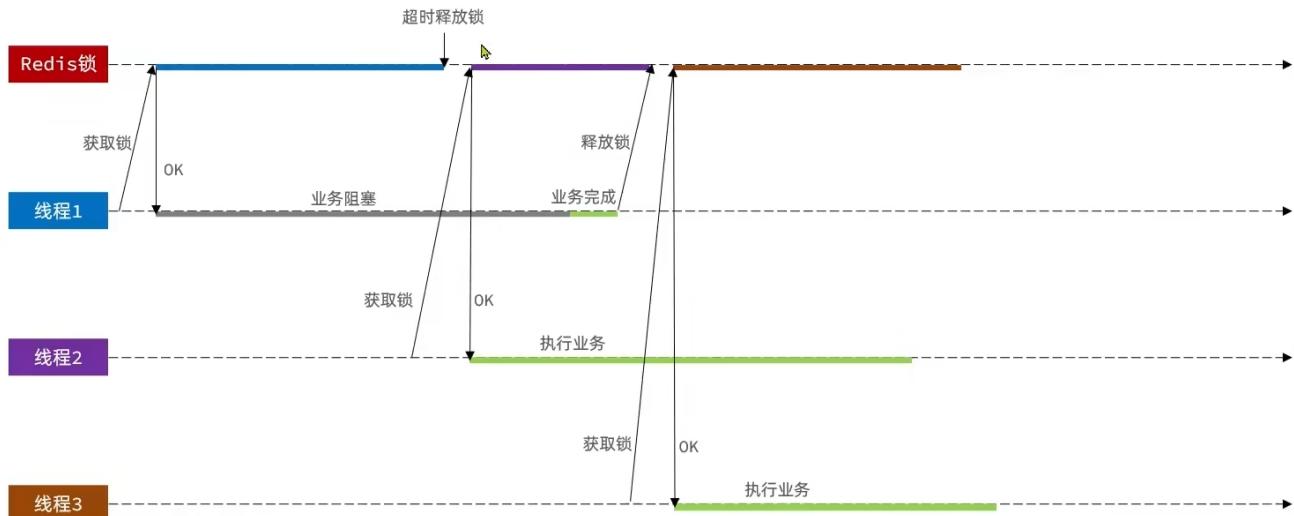


Figure 13: image-20250920164654574

### 改进

- 获取锁的同时加入线程标识
  - 线程标识由 **UUID 加上线程 ID 组成**
  - **不同的 JVM 有不同的 UUID，不同的线程有不同的线程 ID**
- 删锁的时候根据线程标识判断当前线程是否持有该锁
  - 不持有则不能删除锁

### 原子性问题

- 线程 1 正常获取锁，执行完业务，进行锁识别，接着准备进行锁的释放
- **此时由于 JVM 的 GC 发生阻塞!!!，并且阻塞时间大于锁的超时时间导致锁被释放**
- 在这个阻塞过程中线程 2 进入正常获取锁
- 并且此时线程 1 阻塞完成，执行释放锁的逻辑，**把线程 2 的锁给释放掉了!!!!**

**原子性问题主要是判断锁标识和释放锁之间产生了阻塞**

因此必须确保判断锁标识和释放锁必须是一个原子性操作

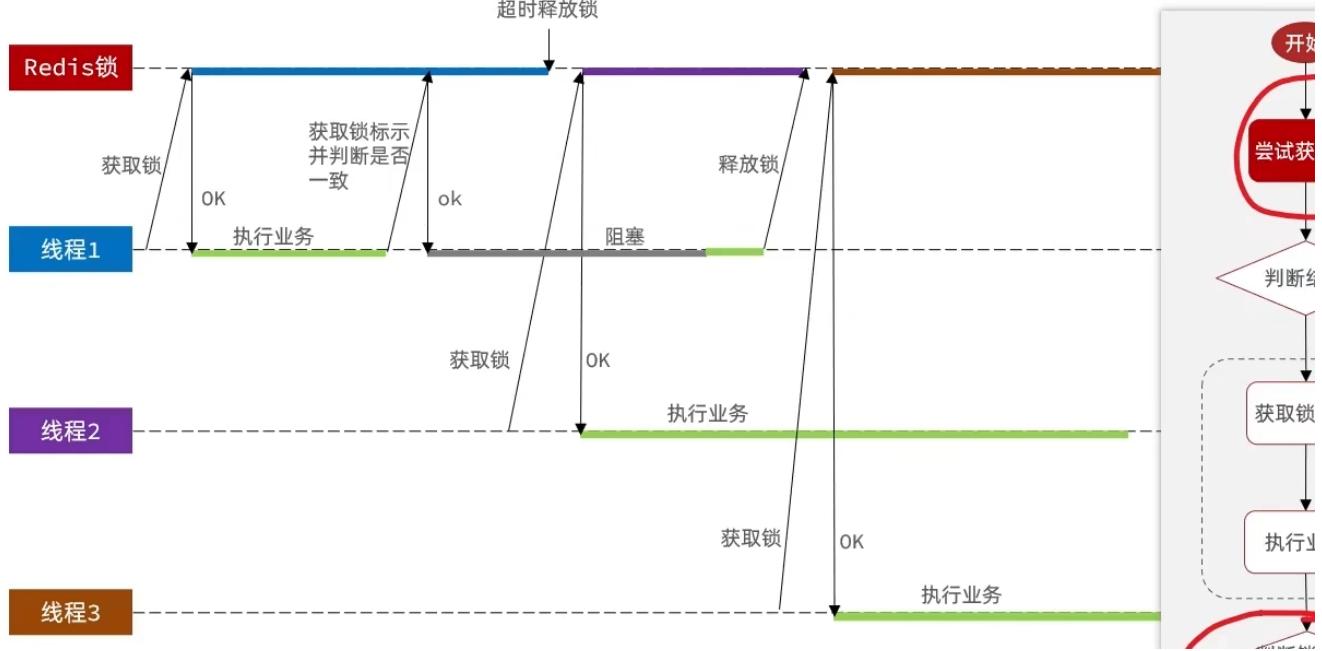


Figure 14: image-20250920165148633

**Redis 的 Lua 脚本** Redis 提供了 Lua 脚本，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  
可以自己看看 Lua 脚本的语法

### Redis 命令调用 Lua 脚本

- EVAL script numkeys key[key...] arg [arg ...]
  - 例子：EVAL "return redis.call('set','name','jack')" 0
- 脚本中的 key、value 可以作为参数传递，key 类型的参数会放进 **KEYS** 数组，其他参数会放入 **ARGV** 数组，**数字表示 KEY 类型参数的数量**
  - 例子：EVAL "return redis.call('set',KEYS[1],ARGV[1])" 1 name Rose

### 在 IDEA 中调用 Lua 脚本

- 使用 DefaultRedisScript 类初始化脚本
- 写静态代码块指定脚本路径以及返回的数据类型
- 使用 stringRedisTemplate 的 execute 方法执行脚本

### Redisson

基于 setnx 实现的**分布式锁**存在下面问题：

- **不可重入**：同一个线程无法多次获取同一把锁
- **不可重试**：获取锁只尝试一次就返回，没有重试机制
- **主从一致性**：主从同步存在延迟

Redisson 是一个在 Redis 基础上实现的 Java 驻内存数据网络（数据存储在内存中的分布式存储系统），提供了一系列的分布式的 Java 常用对象与许多分布式服务

### 使用步骤

1. 导入 Maven 依赖

```

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

```

## 2. 配置 Redisson 客户端

- 使用 @Configuration 注解声明配置类
- 使用 @Bean 注解进行第三方 Bean 注册

```

@Configuration
public class RedissonConfig {
    @Bean
    public RedissonClient redissonClient() {
        // 配置 config
        Config config = new Config();
        config.useSingleServer().setAddress("redis://127.0.0.1:6379").setPassword("123456");
        // 创建 RedissonClient 对象
        return Redisson.create(config);
    }
}

```

## 3. 方法说明

- redissonClient.getLock("Lock")：创建锁
  - "Lock" 为锁的名字
- redissonClient.tryLock(1,10,TimeUnit.SECONDS)：获取锁
  - 1 为获取锁的最大等待时间，**不指定则不进行重试**
  - 10 为锁的自动释放时间
  - 最后一个参数为时间单位
- redissonClient.unLock()：释放锁

**Redisson 可重入锁原理** 可重入锁就是同一个线程可以反复获得它拥有的锁

Redisson 使用 Hash 进行锁重入次数的存储

KEY	VALUE	
field	value	
lock	thread1	2

Figure 15: image-20250921102428234

- 释放锁的时候需要判断重入次数是不是为 1
  - 不是 1 的话不能释放锁，需要把重入次数减 1
  - 是 1 的话可以释放锁
- redisson 还会记录锁的名称和线程唯一标识的映射
  - 当有线程想要获取锁或者释放锁的时候，需要判断这个线程的唯一标识和这个锁对应的线程唯一标识是否一致，不一致则无法操作

## Redisson 的锁重试和 WatchDog 机制 全流程

### 锁重试

- 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

### 看门狗机制

- 获取锁之后开启一个定时任务，这个定时任务每隔一段时间就会去刷新锁的有效时间，避免业务阻塞导致锁超时
- 不设置锁的释放时间才会触发

## Redis 的 MultiLock 原理 Redisson 保留传统的主从分布式方案，但将每个 Redis 节点都作为独立的节点

- 用户需要从每一个 Redis 节点都拿到锁才能代表成功获取锁
- 当有一个节点宕机，此时新的线程想趁虚而入获取锁，虽然在第一个节点成功获取锁，但在其它节点无法获取锁，最终还是无法获得锁的

### 常见方法

- @PostConstruct：这个注解的意思是当前类被初始化就会执行方法里面的逻辑

### 消息队列

存放消息的队列，消息队列模型包括 3 个角色：

- 消息队列：存储和管理消息
- 生产者：发送消息到消息队列
- 消费者：从消息队列获取信息并处理消息

Redis 提供了三种不同的方式来实现消息队列模型

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

### 消息队列的优点

- 解耦

## 基于 List 结构的消息队列 要实现阻塞队列的话，需要使用 BRPOP 或者 BLPOP 命令，可以实现阻塞等待

### 缺点：

- 只支持单消费之
- 有数据安全的问题

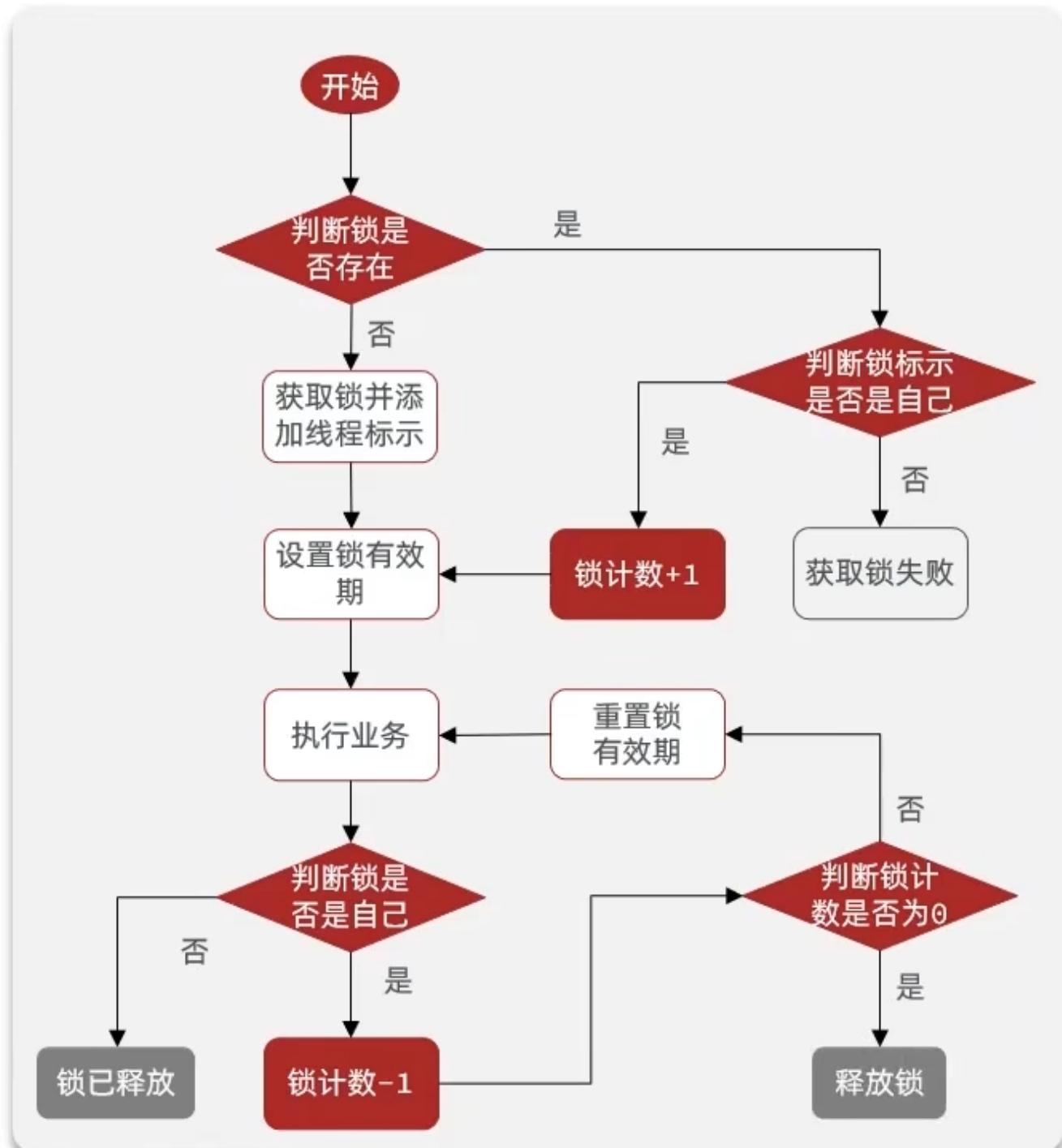


Figure 16: image-20250921103216710

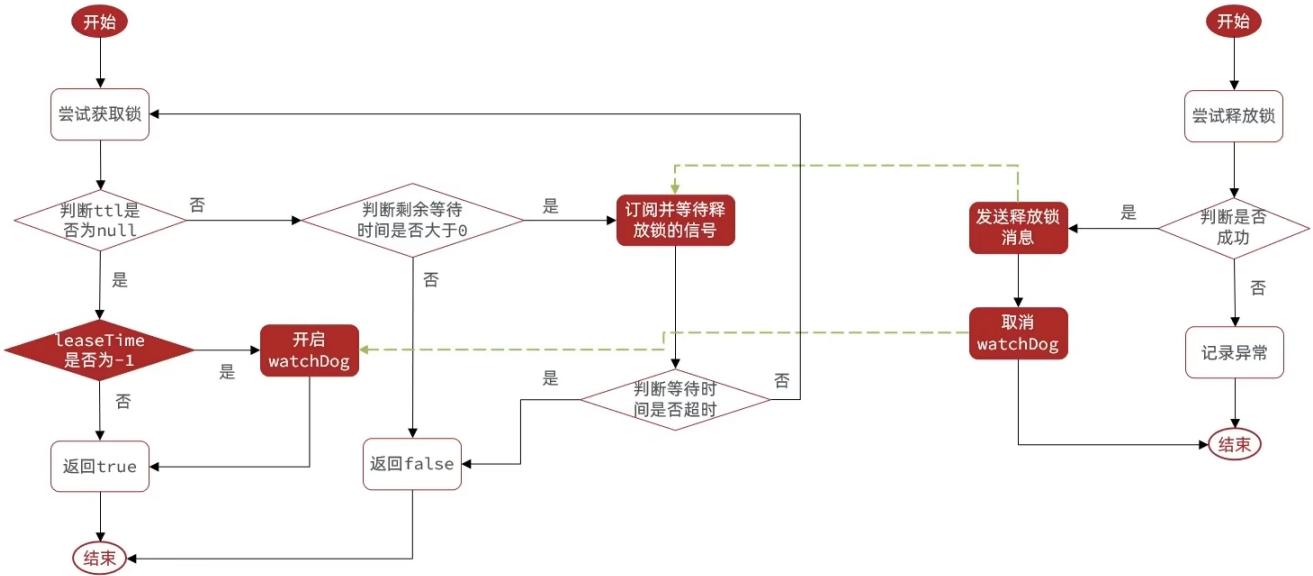


Figure 17: image-20250921105223591

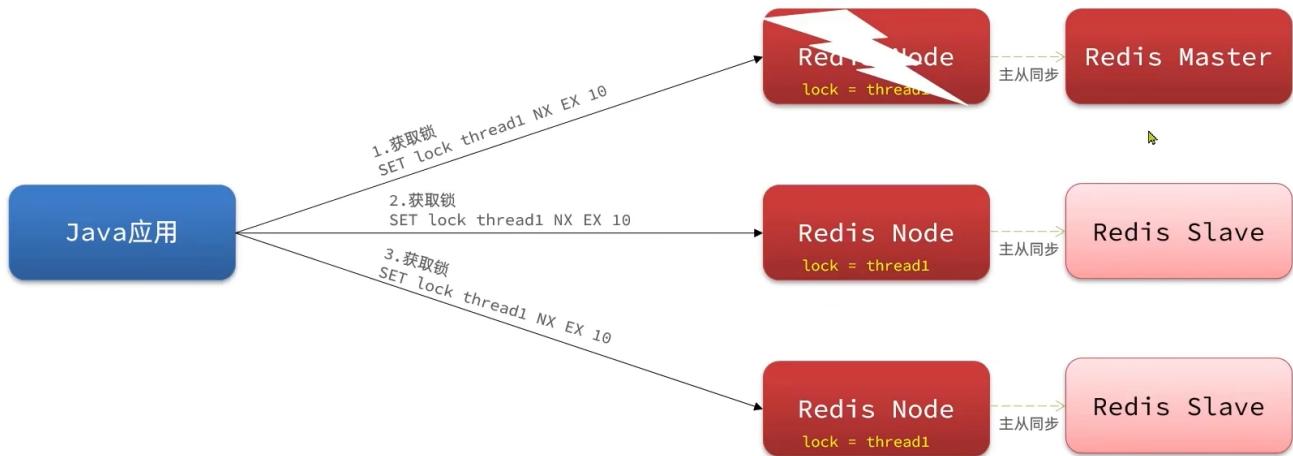


Figure 18: image-20250921105936534

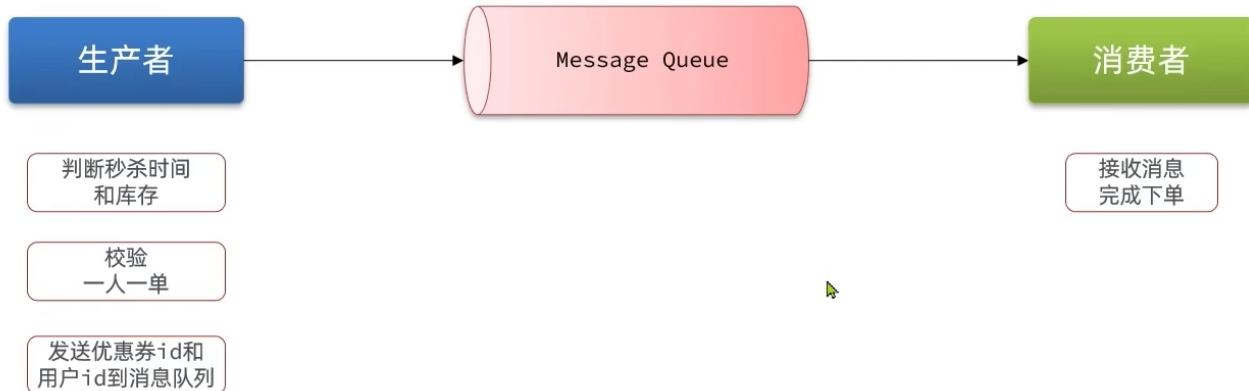


Figure 19: image-20250921132004243

**基于 PubSub 的消息队列** 消费者可以订阅一个或多个 channel，生产者向对应的 channel 发送消息后，所有订阅者都能收到相关信息

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

**SUBSCRIBE pattern[pattern]** : 订阅与pattern格式匹配的所有频道

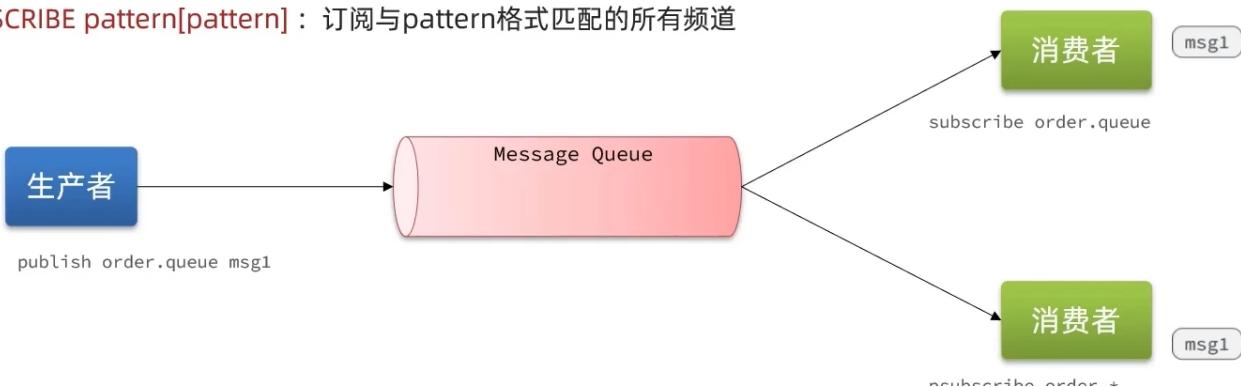


Figure 20: image-20250921133111828

**缺点：**

- 不支持数据持久化
- 无法避免信息丢失
- 数据存储有上限

**基于 stream 的消息队列 完美!!! 是一种数据类型!!!**

**常用方法** ([] 标起来的是可选参数)

- XADD: 向消息队列添加信息
  - key: 消息队列名称
  - [NOMKSTREAM]: 如果队列不存在, 是否创建, 默认为自动创建
  - [MAXLEN]: 消息队列的最大数量

- [\*|ID]: 消息队列的 id, 格式是'时间戳-递增数字'
  - 可以自己指定, 也可以由 redis 自动指定
- field value: 发送到消息队列的消息, 称为 Entry, 格式是多个 key-value 的键值对
- 例子: XADD users \* name jack age 21
- XREAD: 从消息队列读取消息
  - [COUNT]: 每次读取消息的最大数量
  - [BLOCK milliseconds]: 没有消息时, 是否阻塞, 阻塞时长
  - STREAMS key [key ...]: 从哪个队列读取消息, key 就是队列名
  - ID [ID ...]: 起始 id, 只读取大于该 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
- XLEN: 获取消息队列的消息数量

### 消费者组 一个消费者组只能监听一个队列!!!

把多个消费者划分到一个组, 监听同一个队列, 具有下列特点

- 消息分流
- 消息标识
- 消息确认

## 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。

Figure 21: image-20250921135538561

### 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 标识
  - 0: 代表从第一个消息开始读取
  - \$: 代表从最新的消息开始读取
- MKSTREAM: 队列不在时自动创建

### 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
```

### 从消费者组中读取消息

```
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
- count: 本次查询的最大数量
- BLOCK milliseconds: 没有消息时的最长等待时间
- NOACK: 无需手动确认, 收到消息后自动确认
  - 消息不会进入 pending-list (每个消费者组有自己的 pending-list)
- 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 (待处理列表) 中获取已消费但未确认的消息, 0 是从 pending-list 的第一个消息开始

### 确认消息已处理

```
XACK key group ID[ID ...]
```

- key: 队列名称
- group: 消费组名称
- ID: 要确认的消息 ID

### 获取 pending-list

```
XPENDING key group [[IDLE min-dile-time]] start end count [[consumer]]
```

- key: 队列名称
- group: 消费组名称
- IDLE min-dile-time: 消息的空闲时间
- start end: 最小 ID 和最大 ID
  - -: 最小的 ID
  - +: 最大的 ID
- count: 获取的消息数量
- consumer: 要获取哪个消费者的 pending-list

## Feed 流的模式

### 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。
- 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。推送用户感兴趣的信息

### Timeline 的三种实现方式:

- **拉模式**: 也叫做读扩散
- **推模式**: 也叫做写扩散
- **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
  - 将博主分为热门博主和普通博主



Figure 22: image-20250923094109650

**推模式**: 也叫做写扩散。

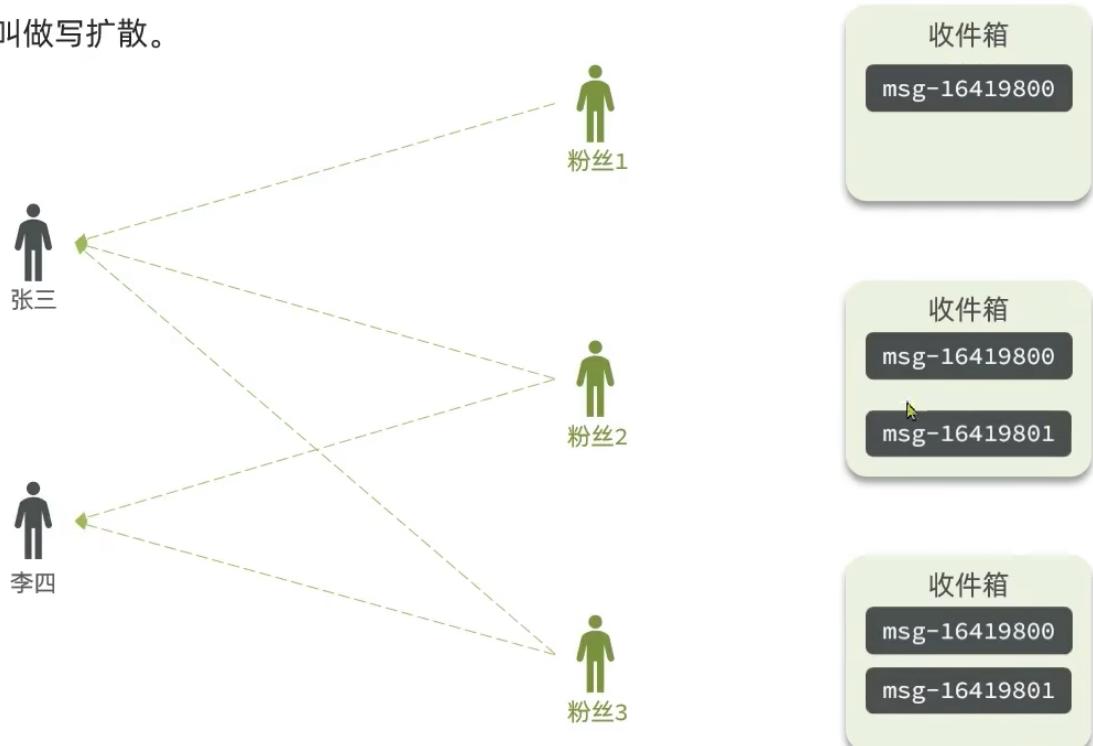


Figure 23: image-20250923094049259

- 普通博主粉丝数较少，直接用推模式
- 热门博主采用推拉结合模式
- 将粉丝分为活跃粉丝和普通粉丝
- 活跃粉丝采用推模式，普通粉丝采用拉模式

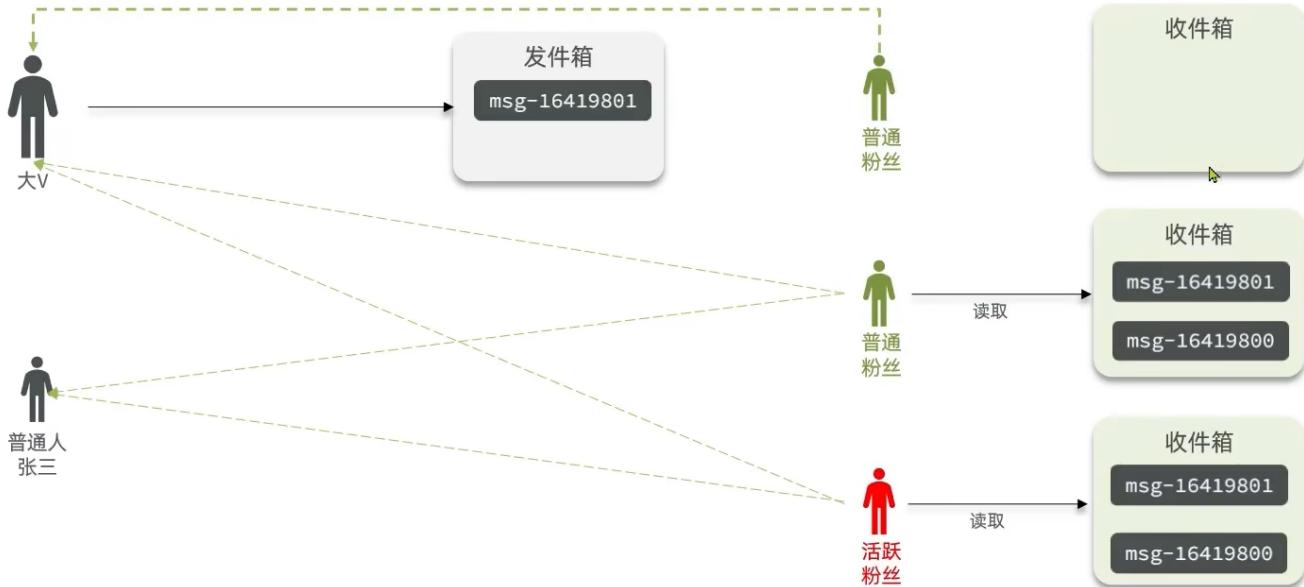


Figure 24: image-20250923094355770

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，需要实现滚动查询（使用 Redis 的 ZSet）

- 记住上一次查询的最小值即可
- 第一次查询，偏移量是 0，其它情况取决于与上一次查询的最小值相同的元素的个数

## GEO 数据结构

GEO 是 Geolocation 的简写形式，代表地理坐标，Redis 中支持这种数据结构，常见命令有：

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

GEO 数据结构存储的时候是用 ZSet 进行存储：

- 存入的经度和纬度会被转换成一个 score
- value 为地名

## BitMap

Redis 中是利用 String 类型数据结构实现 BitMap

#	value	score
1	bjn	4069152240174578
2	bjx	4069879450313142
3	bjz	4069885469876391

Figure 25: image-20250923105504319

- 按月来统计用户签到信息，签到记录为 1，未签到记录为 0
- 把每一个 bit 位对应当月的每一天，形成映射关系，用 0 和 1 标识业务状态，这种思路称为位图

#### 常见操作命令：

- 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
  - type 为要读几个 bit 位
  - offset 为第几个 bit 位开始读
  - u 为无符号，i 为有符号，最高位当符号位
- 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

#### HyperLogLog

- UV:** 独立访客量，1 天内同一个用户多次访问该网站只记录 1 次
- PV:** 页面访问量或者点击量，用户多次打开页面，则记录多次 PV

Redis 中也支持 HyperLogLog 这种数据结构，底层是基于 String 实现的，单个 HLL 的内存永远小于 16kb！

#### 常见命令：

- PFADD:** 向 HLL 添加元素，自动去重
- PFCOUNT:** 统计 HLL 的近似基数
- PFMERGE:** 合并多个 HLL

#### 高级

- 数据丢失问题：**实现 Redis 数据持久化
- 并发能力问题：**搭建主从集群，实现读写分离
- 故障恢复问题：**利用 Redis 哨兵，实现健康检测和自动恢复
- 存储能力问题：**搭建分片集群，利用插槽机制实现动态扩容

#### Redis 持久化

**RDB 持久化** RDB 为 Redis 数据快照，就是把内存中的所有数据记录到磁盘中

- 使用 save 命令进行 RDB 持久化
  - 由 Redis 主进程来执行 RDB，会阻塞所有命令
- 使用 bgsave 也可以执行 RDB 持久化

- 开启子进程执行 RDB，避免主进程受到影响
- Redis 停机时会执行一次 RDB
- 配置文件规范

```

    • # 900 秒内如果至少有一个 key 被修改，则执行 bgsave
      save 900 1

      # 禁用 RDB
      save ""

      # 是否压缩
      rdbcompression yes

      # RDB 文件名称
      dbfilename dump.rdb

      # 文件保存的路径目录
      dir ./
```

### bgsave

- bgsave 开始时会 **fork** 主进程得到子进程，**子进程共享主进程的内存数据**，完成 fork 后读取内存数据并写入 RDB 文件
  - 主进程无法直接操作物理内存，操作系统会给主进程分配虚拟内存，操作系统会维护物理内存和虚拟内存的映射表（页表）
  - 主进程操作虚拟内存，虚拟内存基于页表的映射关系到物理内存
  - 执行 fork 的时候是把页表进行拷贝
- fork 采用的是 copy-on-write 技术
  - 主进程进行读操作，**访问共享内存**
  - 主进程执行写操作，则**会拷贝一份数据执行写操作**

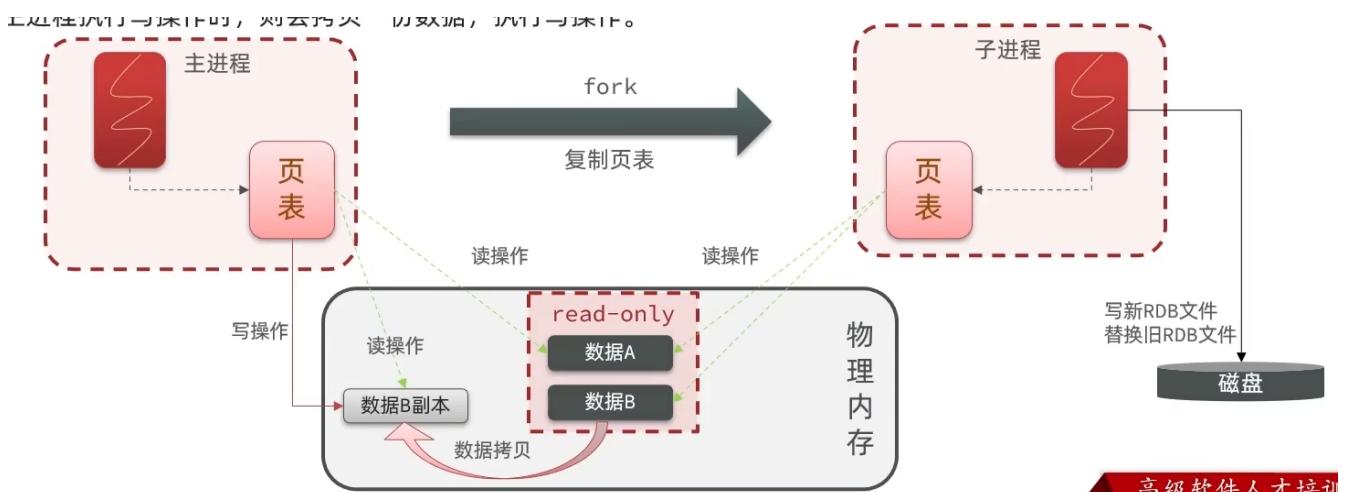


Figure 26: image-20250923160732193

**AOF 持久化** 全称为追加文件，Redis 处理的每一个写命令都会记录在 AOF 文件，可以看作是命令日志文件

**AOF 默认是关闭的，需要修改 redis.conf 配置文件来开启 AOF**

```
## 是否开启 AOF
appendonly yes

## AOF 文件名称
appendfilename "appendonly.aof"
```

**AOF 的命令记录的频率也可以通过 redis.conf 配置文件来指配**

- always: 同步刷盘
- everysec: 每秒刷盘
- no: 操作系统控制

```
## 表示每执行一次写命令，立刻记录到 AOF 文件
appendfsync always

## 写命令执行完先放入 AOF 缓冲区，然后表示每隔 1 秒将缓冲区数据写到 AOF 文件，是默认方案
appendfsync everysec

## 写命令执行完先放入 AOF 缓冲区，由操作系统决定何时将缓冲区内容写到磁盘
appendfsync no
```

	RDB	AOF
<b>持久化方式</b>	定期对整个内存作快照	记录每一次执行的命令
<b>数据完整性</b>	不完整，两次数据备份之间会消失	相对完整，取决于刷盘策略
<b>文件大小</b>	会有压缩，文件体积小	记录命令，文件体积大
<b>宕机恢复速度</b>	很快	慢
<b>数据恢复优先级</b>	低，数据完整性不如 AOF	高，数据完整性高
<b>系统资源占用</b>	高，高内存和高 CPU 占用	低，主要占据磁盘 IO 资源

## Redis 主从

**主节点负责写操作，从节点负责读操作**

**全量同步** 主从的第一次同步是**全量同步**:

1. 第一阶段
  - 从节点向主节点请求数据同步
  - 主节点根据 Replication Id 判断是否是第一次同步
  - 是第一次同步则返回主节点的数据版本信息，包括 Replication Id 和 offset
2. 第二阶段
  - 主节点执行 bgsave，生成 RDB 文件并发送 RDB 文件
  - 主节点会记录 RDB 期间的所有命令到 repl\_backlog 缓冲区中（环形缓冲区），保证数据一致性
3. 第三阶段
  - 主节点发送 repl\_backlog 缓冲区中的所有命令给从节点

## 数据版本信息

- Replication Id: 简称 replid，是数据集的标记，id 一致则说明是同一数据集
  - 每一个 master 都有唯一的 replid，slave 会继承 master 节点的 replid

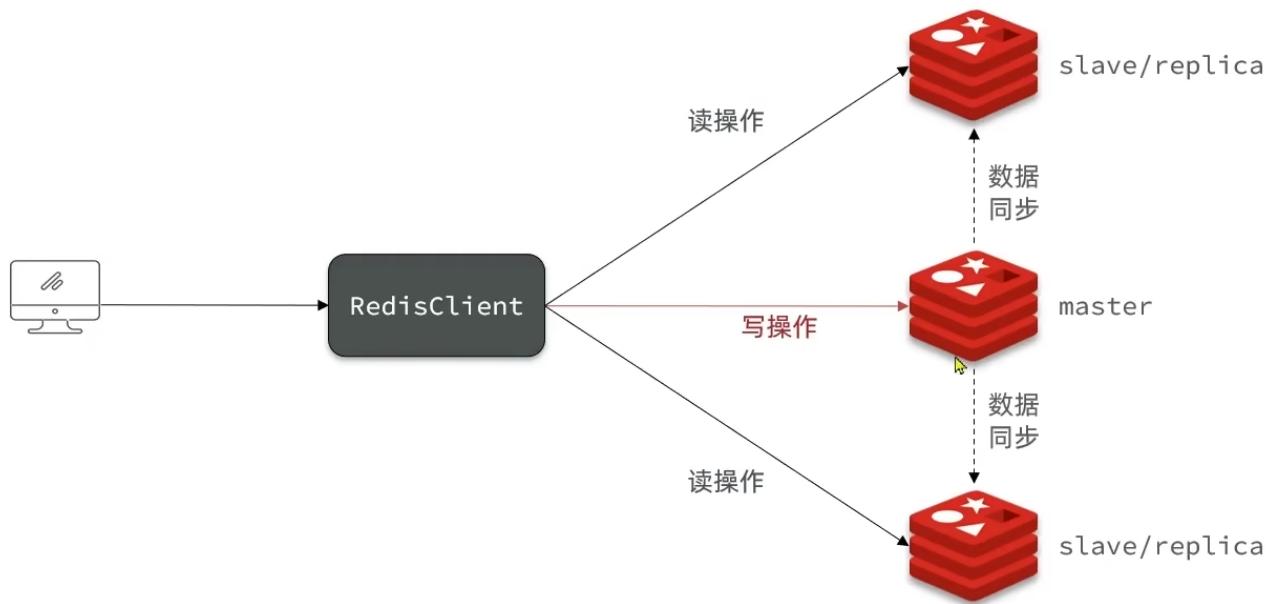


Figure 27: image-20250923162740843



Figure 28: image-20250924121500094

- offset: 偏移量, 随着记录在 `repl_baklog` 中的数据增多而逐渐增大
  - slave 完成同步时也会记录当前同步的 offset, 如果 slave 的 offset 小于 master 的 offset, 说明 slave 数据版本落后于 master, 需要更新

**增量同步** slave 重启后执行增量同步

1. 第一阶段
  - 从节点发送连接请求
  - 主节点判断 `replid` 是否一致
  - 一致, 不是第一次连接, 回复 `continue`
2. 第二阶段
  - 主节点发送从节点 offset 之后的所有新命令

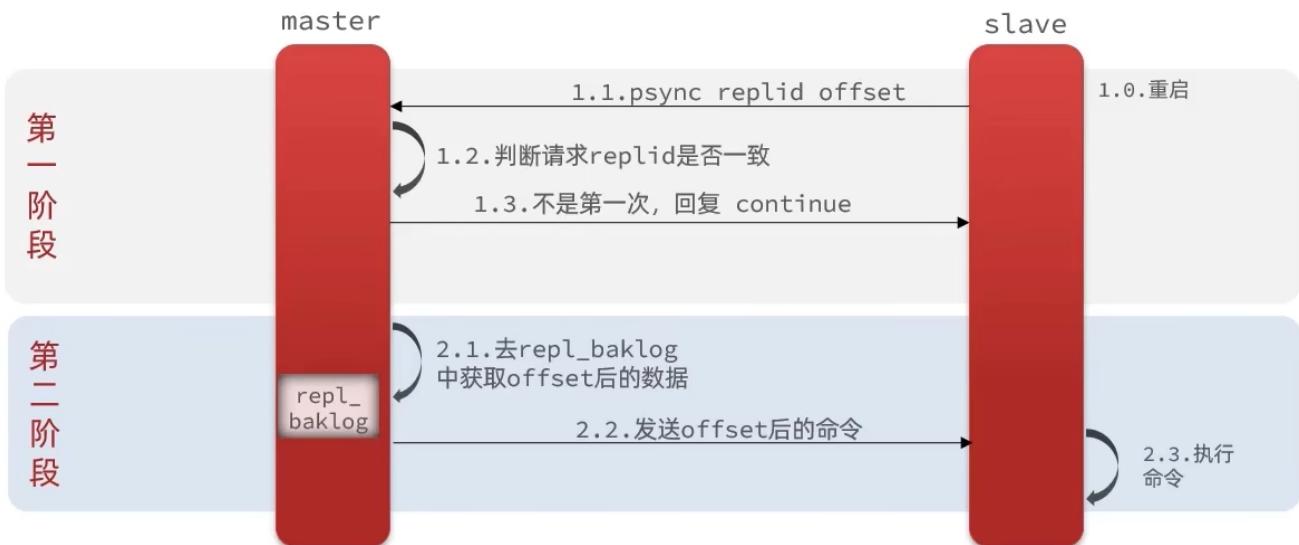


Figure 29: image-20250924122410199

### repl\_baklog 环形缓冲区

- 本质上是一个数组, 但是数组空间可以复用
  - 当环形缓冲区空间已满, 并且 slave 和 master 的数据差异不大, 可以进行数组空间的覆盖
  - 可以根据主节点和从节点的 offset 来判断从节点需要哪些命令进行更新
  - 只要 slave 和 master 的差距不要超过环的存储空间的上限, 从节点就可以实现数据同步

**注意:** `repl_baklog` 大小有上限, 写满后会覆盖最早的数据, 如果 slave 断开时间太久导致尚未备份的数据被覆盖, 就无法基于 log 做同步, 需要进行全量同步

### Redis 哨兵

Redis 基于哨兵机制实现主从集群的自动故障恢复

#### 哨兵的作用

- 监控: Sentinel 会不断检查 master 和 slave 是否按预期工作
- 自动故障恢复: 如果 master 故障, Sentinel 会将一个 slave 提升为 master, 当故障实例恢复后也以新的 master 为主

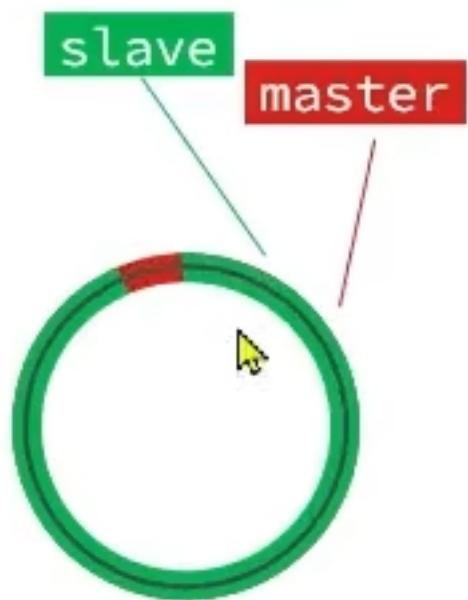


Figure 30: image-20250924122831274

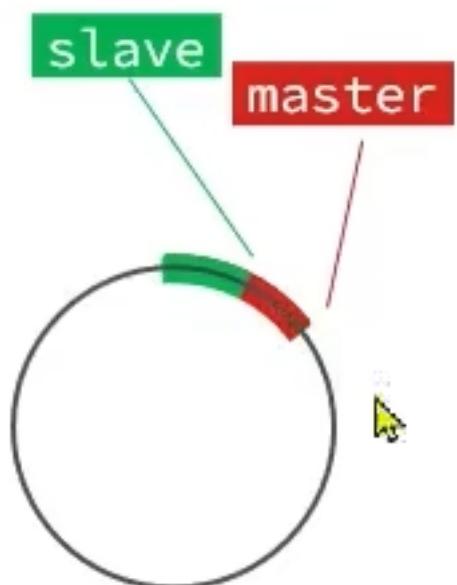


Figure 31: image-20250924122705493



Figure 32: image-20250924123049307

- 通知：Sentinel 充当 Redis 客户端的服务发现来源，当集群发生故障转移时，会将最新信息推送给 Redis 的客户端

Sentinel 基于心跳机制监测服务状态，每隔 1 秒向集群每个实例发送 ping 命令

- 主观下线：如果某哨兵节点发现某实例未在规定时间响应，则认为该实例主观下线
- 客观下线：若超过指定数量（quorum）的哨兵都认为该实例主观下线，则该实例客观下线
  - quorum 最好超过哨兵实例数量的一半

### 选取新的 master

#### 选取新的主节点的依据

- 首先会判断从节点和主节点断开时间的长短，如果断开时间太长超过指定值则会排除该从节点
- 判断从节点的 slave-priority 值，越小优先级越高，如果是 0 则永不参与选举
- 如果 slave-priority 一样，则根据从节点的 offset 值，值越大优先级越高
- 最后判断从节点的允许 id 大小，越小优先级越高

### 故障转移

步骤如下：

- 哨兵给备选的从节点发送 slaveof no one，让该节点称为 master
- 哨兵给其他所有 slave 发送 slvae ip:port，让这些 slave 成为新 master 的从节点，开始从新的 master 上同步数据
- 最后哨兵将故障节点标记为 slave，会强制修改其配置文件，其重启后会自动成为新的 master 的 slave 节点

### Redis 分片集群

主要解决 Redis 高并发写和海量数据存储的问题

#### 特征

- 集群中有多个 master，每个 master 保存不同数据
- 每个 master 可以有多个 slave 节点

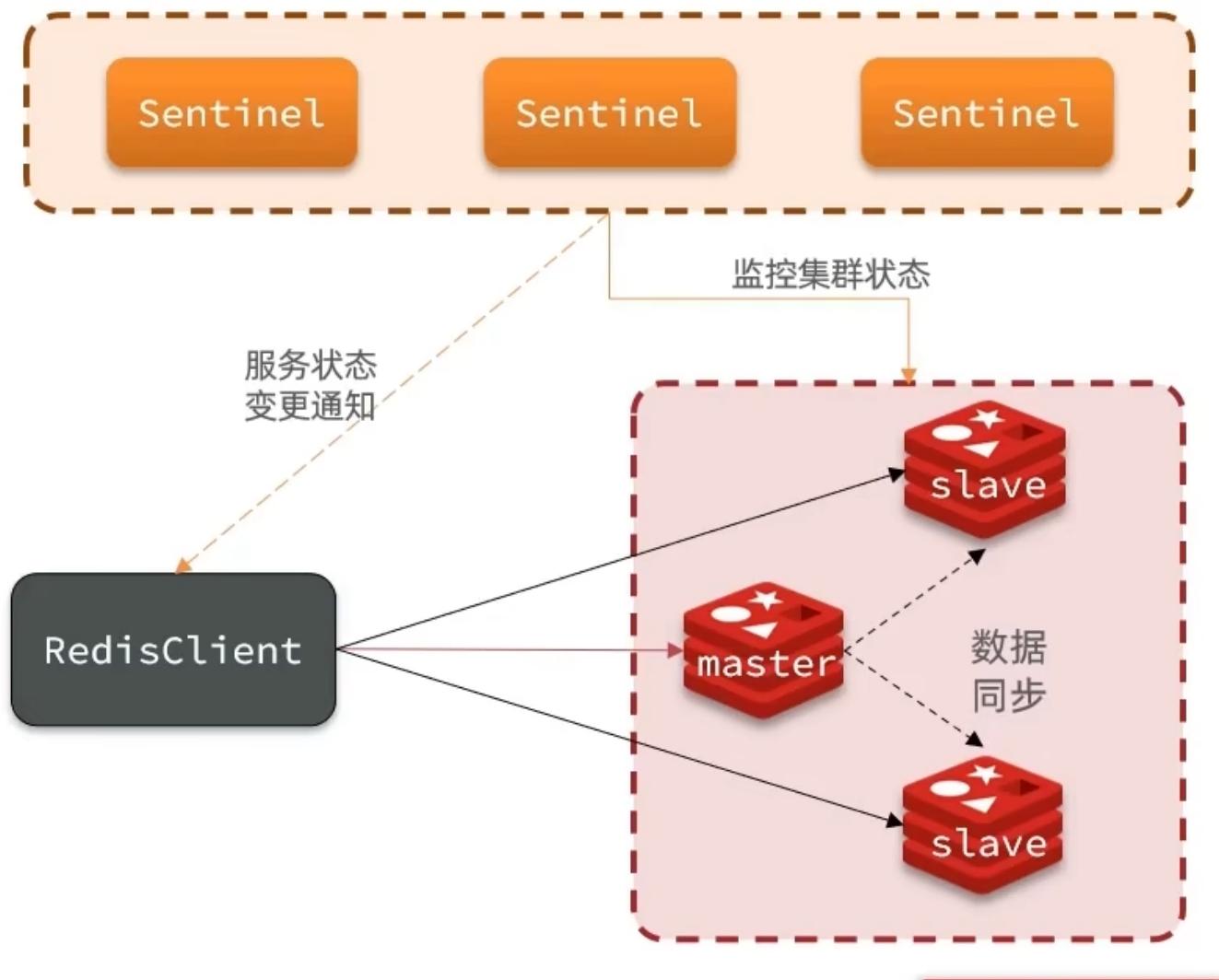


Figure 33: image-20250924161457495

→ ↵ ↶

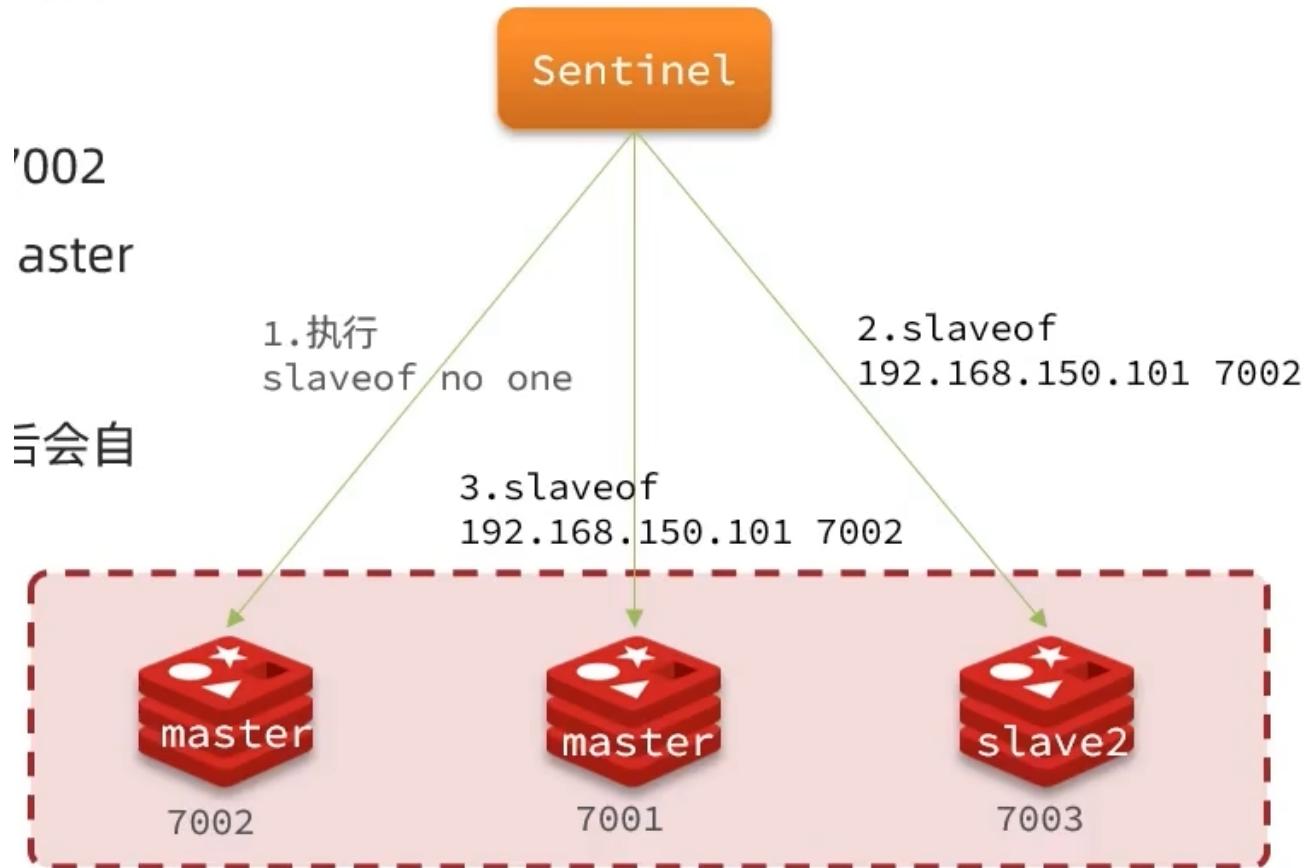


Figure 34: image-20250924170328553

- master 之间通过 ping 监测彼此健康状态
- 客户端可以访问集群任意节点，最终都会被转发至正确的 master 节点

## 启动命令

- redis-cli --cluster: 以集群方式启动 redis

**散列插槽** Redis 会把每一个 master 节点映射到 0-16383 共 16384 个插槽上

数据的 key 不是与节点绑定，而是与插槽绑定。redis 会根据 **key 的有效部分计算插槽值**，有以下两种情况

- key 中包含 {}，且 {} 中至少包含 1 个字符，{} 中的部分是有效部分
- key 中不含 {}，整个 key 都是有效部分

**计算方式** 是利用 CRC16 算法得到一个 hash 值，然后对 16384 取余，得到的结果就是 slot 值

**集群伸缩** redis-cli --cluster add-node: 添加节点

- new\_host:new\_port: 新的节点和端口
- existing\_host:existing\_port: 集群中的已经存在的一个 ip 和端口
- --cluster-save: 使生成的节点变成从节点

redis-cli --cluster reshard: 移动插槽

- host:port: 被移动插槽的节点的 ip 和端口

**故障转移** 当集群中有一个 master 宕机时会自动提升一个 slave 为新的 master

## 数据迁移

利用 cluster failover 命令可以手动让集群中的某个 master 成为从节点，让执行 cluster failover 命令的 slave 节点成为主节点，实现无感知的数据迁移

流程如下：

## 多级缓存

如下所示：

- 浏览器缓存
- Nginx 缓存
- Redis 缓存
- JVM 进程缓存
- 数据库

## JVM 进程缓存

### Nginx 缓存 Lua 语法

- 数据类型
  - nil: 表示一个无效值，条件表达式中相当于 false
  - boolean: false 和 true
  - number: 双精度类型的浮点数
  - string: 字符串
  - function: 由 C 或 lua 编写的函数

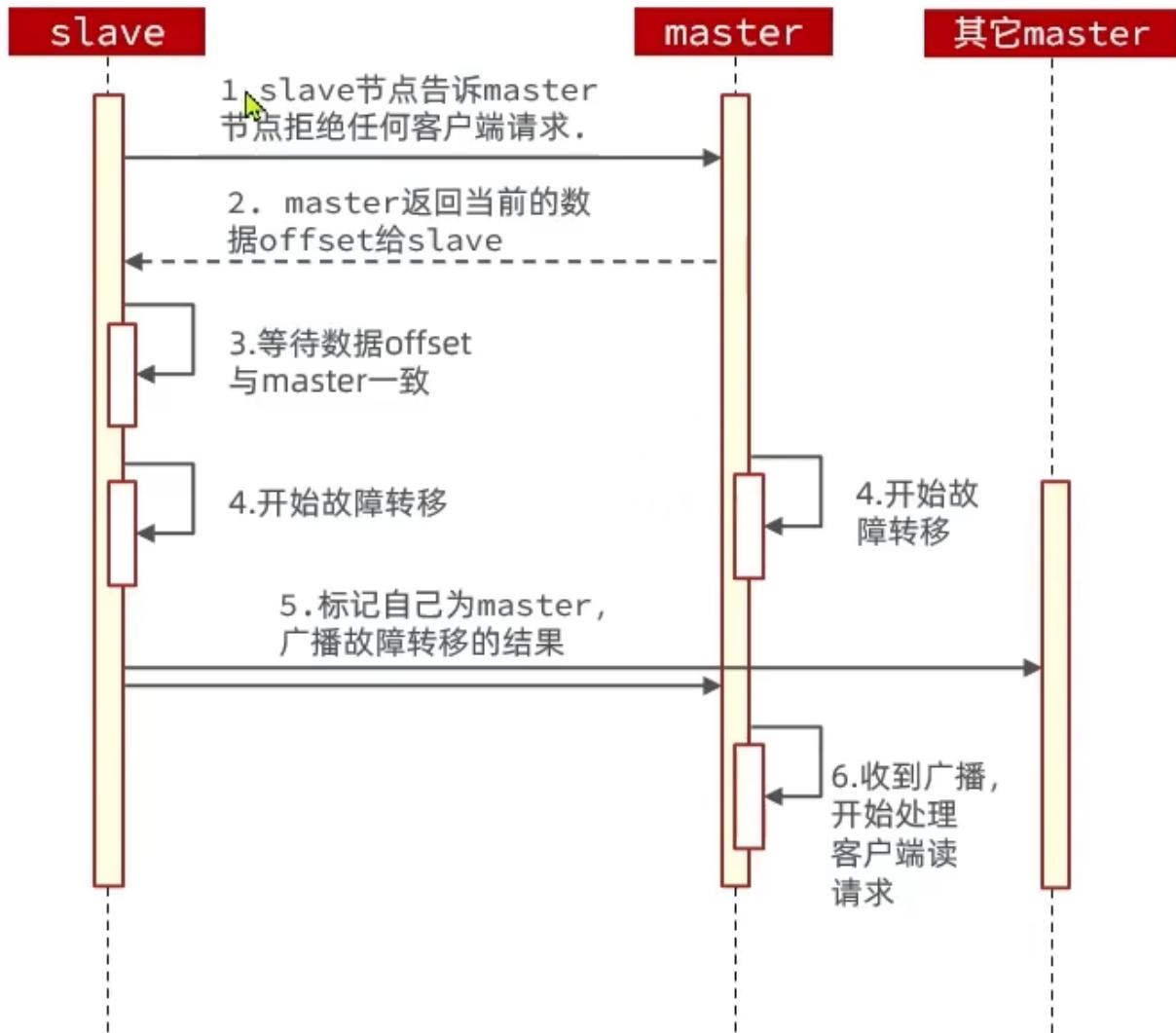


Figure 35: image-20250924204248579

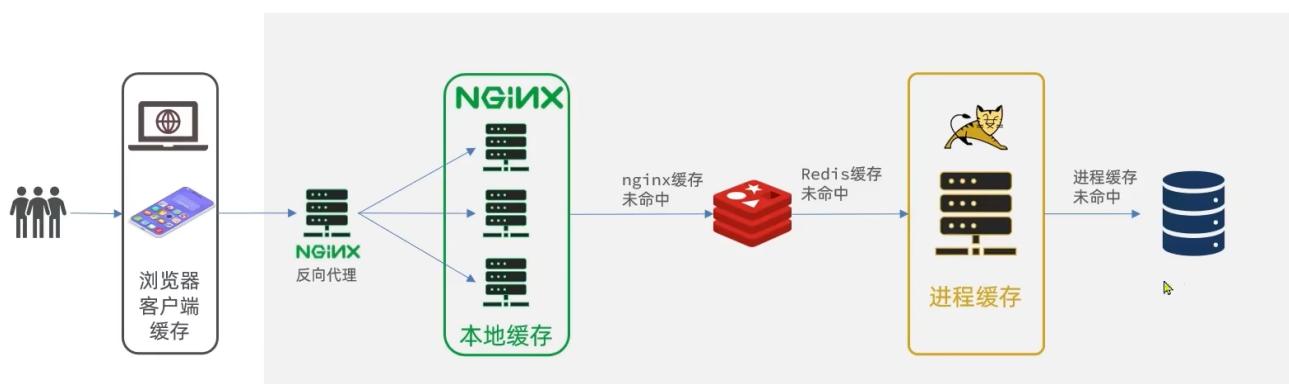


Figure 36: image-20250924205836239

- table: Lua 中的表，数组的索引可以是数字、字符串或者表类型
  - 使用 {} 创建表
- 变量

- 使用 local 声明变量
- 使用 type 获取数据类型

```
-- 声明字符串
local str = 'hello'

-- 声明数字
local num = 21

-- 声明布尔类型
local flag = true

-- 声明数组
local arr = {'java','python','lua'}

-- 声明 table
local map = {name = 'jack',age = 21}
```

- 访问 table:

```
-- 访问数组
print(arr[1])
-- 访问 map
print(map['name'])
print(map.name)
```

- 循环

- 遍历数组

```
-- 声明数组 key 为索引的 table
local arr = {'java','python','lua'}
-- 遍历数组
for index,value in ipairs(arr) do
  print(index,value)
end
```

- 遍历 table

```
-- 声明 table
local map = {name = 'jack',age = 21}
-- 遍历 tabel
for key,value in pairs(map) do
  print(key,value)
end
```

- 函数

- 定义函数的语法

```
function 函数名(argument 1,argument2 ... ,argumentn)
    return 返回值
end
```

- 条件控制
  - and: 与
  - or: 或
  - not: 非

```
if(布尔表达式)
then
    --[ 布尔表达式 true 时执行该语句块 --]
else
    --[ 布尔表达式 false 时执行该语句块 --]
end
```

## 原理

Redis 是基于 c 语言实现的

### 数据结构

**动态字符串 SDS** 可以把 Redis 里面的数据结构都看成是字符串

Redis 构建了一种新的字符串结构，称为**简单动态字符串**，即 **SDS**

- SDS 本质上是一个**结构体**，下面的只是**其中的一种 SDS** (支持 8 个 bit)，Redis 还定义了 16bit、32bit、64bit 的 SDS

```
struct __attribute__((__packed__)) sdshdr8{
    uint8_t len; /* buf 已保存的字符串字节数，不包含结束标识 只有 8 个 bit 字符串最长只能存 255
    ↵ 个字节 */
    uint8_t alloc; /* buf 申请的总的字节数，不包含结束标识 */
    unsigned char flags; /* 不同 SDS 的头类型，用来控制 SDS 的头大小 */
    char buf[];
}
```

- 可以通过 **len** 去决定要读取多长的字节
- 具备**动态扩容**的能力，能够进行**内存预分配**
  - 假设要给 SDS 追加一段字符串，这里会进行内存空间的申请
    - 如果新字符串（**追加后的总长度**）小于 1M，则新空间为新字符串长度（**追加后的总长度**）的 2 倍 +1
    - 如果新字符串（**追加后的总长度**）大于 1M，则新空间为新字符串长度（**追加后的总长度**）的 +1M+1
    - +1 是为了存储 \0，但 SDS 本身不依赖它

**IntSet** 是 Redis 中 **set** 集合的一种实现方式，底层基于整数数组实现，并且具备长度可变、有序等特征，底层采用**二分查找**来查询

结构如下：

例如，一个包含字符串“name”的sds结构如下：



Figure 37: image-20250911123033944

```
typedef struct intset{  
    uint32_t encoding; /* 编码方式，支持存放 16 位、32 位、64 位整数 */  
    uint32_t length; /* 元素个数 */  
    int8_t contents[]; /* 整数数组，保存集合数据，存放数据的字节数由 encoding 决定 */  
}intset;
```

其中 encoding 包含三种模式，表示存储的整数大小不同

```
##define INTSET_ENC_INT16 (sizeof(int16_t)) /* 2 字节整数，范围类似 java 的 short */  
##define INTSET_ENC_INT32 (sizeof(int32_t)) /* 4 字节整数，范围类似 java 的 int */  
##define INTSET_ENC_INT64 (sizeof(int64_t)) /* 8 字节整数，范围类似 java 的 long */
```

Redis 会将 intset 中所有的整数按照升序依次保存在 contents 数组中：

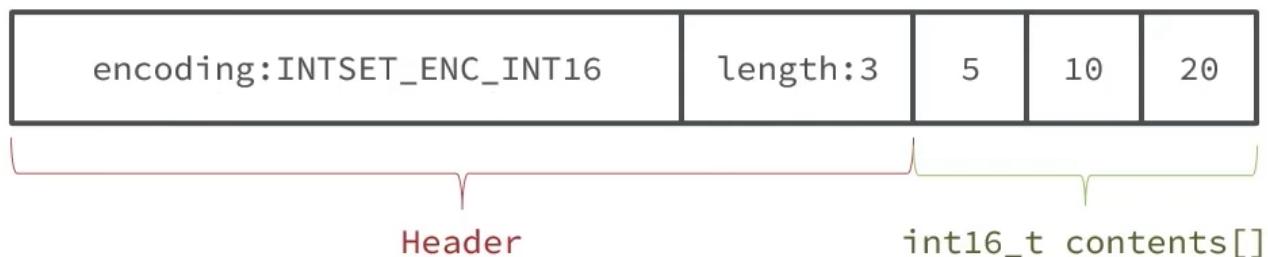


Figure 38: image-20250911193319146

现在数组中每个数字都在 `int16_t` 的范围内，采用的编码方式是 `INTSET_ENC_INT16`，每部分占用的字节大小为

- `encoding`: 4 个字节
- `length`: 4 个字节
- `contents`: 2 字节 \*3=6 字节

寻址公式：`startPtr + (sizeof(int16) * index)`

- `index`: 是元素下标
- `startPtr`: 是数组的地址，也是数组第一个元素的地址
- `sizeof()`: 取决于 `encoding` 的编码方式

### intset 升级

- 当加入了一个较大的数字，intset 会对编码方式进行升级到合适的大小

1. 升级编码方式，并按照新的编码方式及元素个数扩容数组
2. 倒序依次将数组中的元素拷贝到扩容后的正确位置（在原有数组的连续空间上申请新的连续内存）
3. 将待添加的元素放入数组末尾
4. 将 intset 的 encoding 和 length 属性进行更改

**Dict** 由三部分组成：

- 哈希表：底层就是数组，保存的是哈希节点
- 哈希节点
- 字典

## 哈希表

```
typedef struct dictht{
    // entry 数组
    // table 是指针数组的指针，数组元素类型是 dictEntry*
    // 是指向指针数组的指针
    dictEntry **table;
    // 哈希表大小，总是为 2 的 n 次方
    unsigned long size;
    // 哈希表大小的掩码。总等于 size-1
    unsigned long sizemask;
    // entry 个数
    unsigned long used;
}dictht;
```

## 哈希节点

```
typedef struct dictEntry{
    void *key; // 键
    union {
        void *val;
        uint64_t u64;
        int64_t s64;
        double d;
    }v; // 值
    // 下一个 Entry 的指针
    struct dictEntry *next;
} dictEntry;
```

当向 Dict 添加键值对时，Redis 首先根据 key 计算出哈希值 hash (h)，然后利用 h & sizemask 来计算元素应该存储到数组中的哪个索引位置

- 注意是做与运算，原因是 size 总是为 2 的 n 次方，那么 sizemask 的低位总是 1
- 算某个数与 size 的余数其实就是看这个数低位所代表的十进制数

十进制：4

二进制：0000 0000 0100

新数字：7

二进制：0000 0000 0111

与运算后是：

0000 0000 0011

余数是3

## 字典

```
typedef struct dict{  
    dictType *type; // dict 类型，内置不同的哈希函数  
    void *privdata; // 私有数据，在做特殊 hash 运算时使用  
    dictht ht[2]; // 一个 dict 包含两个哈希表，其中一个是当前数据，另一个一般是空，rehash 时使用  
    long rehashidx; //rehash 的进度，-1 表示未进行  
    int16_t pauserehash; //rehash 是否暂停，1 则暂停，0 则继续  
} dict;
```

**Dict 的扩容和收缩** 当集合中的元素较多的时候，会导致哈希冲突的增多，链表过长，查询效率会降低，需要进行扩容

Dict 每次新增键值对的时候都会检查负载因子 ( $\text{LoadFactor} = \text{used}/\text{size}$ )，遇到下面两种情况时会触发哈希表扩容：

- 哈希表的 LoadFactor  $\geq 1$ ，并且服务器没有执行 BGSAVE 或者 BGREWRITEAOF 的后台进程
  - 因为扩容操作比较吃 CPU 资源，如果后台有其它进程，会导致扩容无法及时完成，进而导致主线程阻塞
  - BGSAVE：将 Redis 当前内存中的数据快照保存到磁盘，生成一个 **RDB 文件**
    - RDB 是一种全量备份方式，保存的是某个时间点的完整数据
  - BGREWRITEAOF：对 AOF 日志进行压缩重写，去除冗余命令
    - AOF 记录的是写操作命令
- 哈希表的 LoadFactor  $> 5$

除了扩容以外，每次删除元素的时候也会对负载因子做检查，当  $\text{LoadFactor} < 0.1$  时，会对哈希表进行收缩

**Dict 的 Rehash** 不论是扩容还是收缩，必然会创建新的哈希表，导致哈希表的 size 和 sizemask 发生变化，而 key 的查询和 sizemask 有关。

所以需要对哈希表中的每一个 key 重新计算索引，插入新的哈希表，这个过程称为 **rehash**：

1. 计算新哈希表的 realSize，值取决于当前要做的是扩容还是收缩
  - 如果是扩容，则新 size 为第一个大于等于  $\text{dich}.ht[0].used + 1$  的  $2^n$
  - 如果是收缩，则新 size 为第一个大于等于  $\text{dich}.ht[0].used + 1$  的  $2^n$  (不得小于 4)
2. 按照新的 realSize 申请内存空间，创建 dictht，并赋值给  $\text{dict}.ht[1]$
3. 设置  $\text{dict}.rehashidx = 0$ ，表示开始 rehash
4. 将  $\text{dict}.ht[0]$  中的每一个 dictEntry 都 rehash 到  $\text{dict}.ht[1]$
5. 将  $\text{dict}.ht[1]$  赋值给  $\text{dict}.ht[0]$ ，给  $\text{dict}.ht[1]$  初始化为空的哈希表，释放原来的  $\text{dict}.ht[0]$  内存

## Dict 的渐进式哈希

如果哈希表有数百万的数据，一次性 rehash 会导致**主线程阻塞**

因此 Dict 的 rehash 是分多次，渐进式的完成：

- 每次执行新增、查询、修改、删除操作时，都检查一下  $\text{dict}.rehashidx$  是否大于-1
- 如果是则将  $\text{dict}.ht[0].table.[rehashindex]$  的 entry 链表 rehash 到  $\text{dict}.ht[1]$ ，并将  $\text{rehashidx}++$
- 直到所有的数据都从  $\text{dict}.ht[0]$  被 rehash 到  $\text{dict}.ht[1]$

注意：

- 在 rehash 过程中，新增操作时直接写入  $\text{dict}.ht[1]$ ，查询、修改和删除会在  $\text{dict}.ht[0]$  和  $\text{dict}.ht[1]$  依次查找并执行

**ZipList** 是一种特殊的双端链表，由一系列特殊编码的连续内存块组成。支持在任意一端进行压入、弹出操作

zbytes	zltail	zllen	entry	entry	...	entry	zlend
--------	--------	-------	-------	-------	-----	-------	-------

属性	类型	长度	用途
zbytes	uint32_t	4 字节	记录整个压缩列表占用的内存字节数
zltail	uint32_t	4 字节	记录压缩列表表尾节点距离压缩列表的起始地址有多少字节，通过这个偏移量，可以确定表尾节点的地址。
zllen	uint16_t	2 字节	记录了压缩列表包含的节点数量。最大值为UINT16_MAX（65534），如果超过这个值，此处会记录为65535，但节点的真实数量需要遍历整个压缩列表才能计算得出。
entry	列表节点	不定	压缩列表包含的各个节点，节点的长度由节点保存的内容决定。
zlend	uint8_t	1 字节	特殊值 0xFF（十进制 255），用于标记压缩列表的末端。

Figure 39: image-20250912171110588

- zlend: 结束标识，0xff
- zbytes: 压缩列表的总字节数
- zltail: 尾节点的偏移量，记录压缩列表表尾节点距离压缩列表的起始地址有多少字节
- zllen: entry 节点个数
- entry: 节点
  - entry 不像普通链表那样记录前后节点的指针，因为记录两个指针要占用 16 个字节，浪费内存，而是采用下面结构

previous_entry_length	encoding	content
-----------------------	----------	---------

Figure 40: image-20250913162441318

- previous\_entry\_length: 前一节点的长度，占 **1 个或 5 个字节**
  - 如果前一节点的长度小于 254，则用 **1 个字节**来保存这个长度值
  - 如果前一节点的长度大于 254，则用 **5 个字节**来保存这个长度值，**第一个字节为 0xfe**，后四个字节才是**真实长度数据**
- encoding: 编码属性，记录 content 的数据类型以及**长度**，占用 1 个、2 个或者 5 个字节
  - 以 00、01 或者 10 开头，证明 content 是字符串
  - 以 11 开头证明 content 是整数，且 encoding 固定只占用 1 个字节
    - 如果保存的整数很小，直接在 encoding 后面的 **4 个字节存储保存的整数**
- contents: 负责保存节点的数据，可以是**字符串或者整数**

编码	编码长度	字符串大小
00pppppp	1 bytes	<= 63 bytes
01pppppp qqqqqqqq	2 bytes	<= 16383 bytes
10000000 qqqqqqqq rrrrrrrr ssssssss ttttttt	5 bytes	<= 4294967295 bytes

Figure 41: image-20250913163735100

编码	编码长度	整数类型
11000000	1	int16_t (2 bytes)
11010000	1	int32_t (4 bytes)
11100000	1	int64_t (8 bytes)
11110000	1	24位有符整数(3 bytes)
11111110	1	8位有符整数(1 bytes)
1111xxxx	1	直接在xxxx位置保存数值，范围从0001~1101，减1后结果为实际值

Figure 42: image-20250913165631080

注意：

- ZipList 中所有存储长度的数值均使用小端字节序进行存储，即**低位字节在前，高位字节在后**，使用十六进制进行存储

### ZipList 的连鎖更新问题

- 假设有 N 个连续的、长度为 250-253 字节之间的 entry，因此 entry 的 previous\_entry\_length 属性可用 1 个字节表示
- 但如果此时在头部插入一个 254 字节以上的 entry，会导致后面 entry 的 previous\_entry\_length 需要**扩展为 5 个字节**
- 此时会导致**连鎖反应**，也就意味着**数据要不断往后迁移**（因为是连续内存空间）

连续多次空间扩展操作称为**连鎖更新问题**。新增、删除都可能导致这种问题

**QuickList** 是一个双端链表，只不过链表中的每一个节点都是一个 ZipList

- 可以使用 list-max-ziplist-size 来限制 QuickList 中的每个 ZipList 中 entry 的数量
- 可以使用 list-compress-depth 来控制对节点的 ZipList 做压缩

### QuickList

```
typedef struct quicklist{
    // 头节点指针
    quicklistNode *head;
    // 尾节点指针
    quicklistNode *tail;
    // 所有 ziplist 的 entry 的数量
    unsigned long count;
```

```

// ziplists 总数量
unsigned long len;
// ziplist 的 entry 上限，默认值-2
int fill : QL_FILL_BITS;
// 首尾不压缩的节点数量，可以控制被压缩节点的数量
unsigned int compress : QL_COMP_BITS;
// 内存重分配时的书签数量及数组，一般用不到
unsigned int bookmark_count : QL_BM_BITS;
quicklistBookmark bookmarks[];
}quicklist;

```

## QuickListNode

```

typedef struct quicklistNode{
    // 前一个节点的指针
    struct quicklistNode *prev;
    // 下一个节点的指针
    struct quicklistNode *next;
    // 当前节点的 ZipList 的指针
    unsigned char *zl;
    // 当前节点的 ZipList 的字节大小
    unsigned int sz;
    // 当前节点的 ZipList 的 entry 个数
    unsigned int count : 16;
    // 编码方式: 1, ZipList; 2, lzf 压缩模式
    unsigned int encoding : 2 ;
    // 数据容器类型，方便以后做扩展
    unsigned int container : 2;
    // 是否被解压缩
    unsigned int recompress : 1;
    unsigned int attempted_compress : 1;
    unsigned int extra : 10;
} quicklistNode;

```

## SkipList 跳表首先是链表，与传统链表有差异：

- 元素按照升序进行排列存储
- 节点可能包含多个指针，指针跨度不同

## 跳表

```

typedef struct zskiplist{
    // 头尾节点指针
    struct zskiplistNode *head, *tail;
    // 节点数量
    unsigned long length;
    // 最大的索引层级，默认是 1
    int level;
}zskiplist;

```

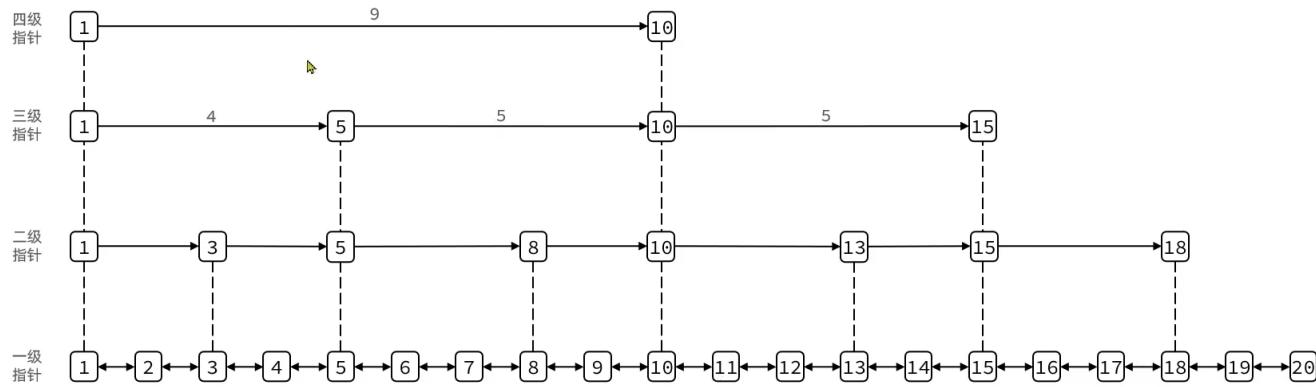


Figure 43: image-20250913171827926

## 跳表节点

```
typedef struct zskiplistNode{
    sds ele; // 节点存储的值
    double score; // 节点分数, 排序, 查找用
    struct zskiplistNode *backward; // 前一个结点的指针
    struct zskiplistLevel{
        struct zskiplistNode *forward;// 下一个节点指针
        unsigned long span; // 索引跨度
    }level [];
}zskiplistNode;
```

特定：

- 跳表是一个双端链表，节点存 score 和 ele 值
- 节点按照 **score** 排序，score 一样则按照 ele 字典序排序
- 每个节点可以包含**多层指针**，层数是 1 到 32 的随机数
- 不同层指针到下一个节点的**跨度不同**，**层级越高跨度越大**

**RedisObject** Redis 中的任意数据类型的键和值都会被封装成一个 **RedisObject**，也叫做 Redis 对象

```
typedef struct redisObject{
    unsigned type:4; // 对象类型, 分别是 string,hash,list,set 和 zset
    unsigned encoding:4; // 底层编码方式, 共有 11 种
    unsigned lru:LRU_BITS; // 记录当前 redis 对象最近被访问的时间
    int refcount; // 对象引用计数器, 计数器为 0 则说明对象无人引用, 可以被回收
    void *str; // 指向存放实际数据的内存空间, 意味着对象和存储的数据内存空间不连续
} robj;
```

**不同数据类型使用的编码：**

- STRING: INT、RAW、EMBSTR
- LIST: LinkedList 和 ZipList (3.2 以前)、QuickList (3.2 以后)
- SET: intset、HT (Hash Table, 也就是 Redis 中的 Dict)
- ZSET: ZipList、HT、SkipList
- HASH: ZipList、HT

## String

- 其基本编码方式是 **RAW**, 基于 SDS 实现, 存储上限为 512MB
- 如果存储的 **SDS 长度 (字符串本身的长度)** 小于 44 字节, 会采用 **EMBSTR** 编码, 此时 RedisObject head 与 SDS 是一段连续空间, 申请内存只需要调用一次内存分配函数

RedisObject					SDS										
type: OBJ_STRING	encoding: OBJ_ENCODING_EMBSTR	lru	refcount	ptr	len:44	alloc:44	flags:1	L	o	n	g	,	.	.	\0
								L	o	n	g	,	.	.	\0

Figure 44: image-20250914145411630

- 存储的字符串是整数值的时候, 并且大小在 **LONG\_MAX** 范围内, 会采用 **INT** 编码, 并且直接将数据保存在 **RedisObject** 的 **ptr** 指针位置 (刚好 8 字节), 不再需要 SDS 了

## List 可以从首尾操作列表中的元素

- 3.2 之前, Redis 采用 ZipList 和 LinkedList, 当元素数量小于 **512** 并且元素大小小于 **64 字节** 采用 ZipList 编码, 超过则用 LinkedList 编码
- 3.2 之后, Redis 统一采用 **QuickList** 来实现 List

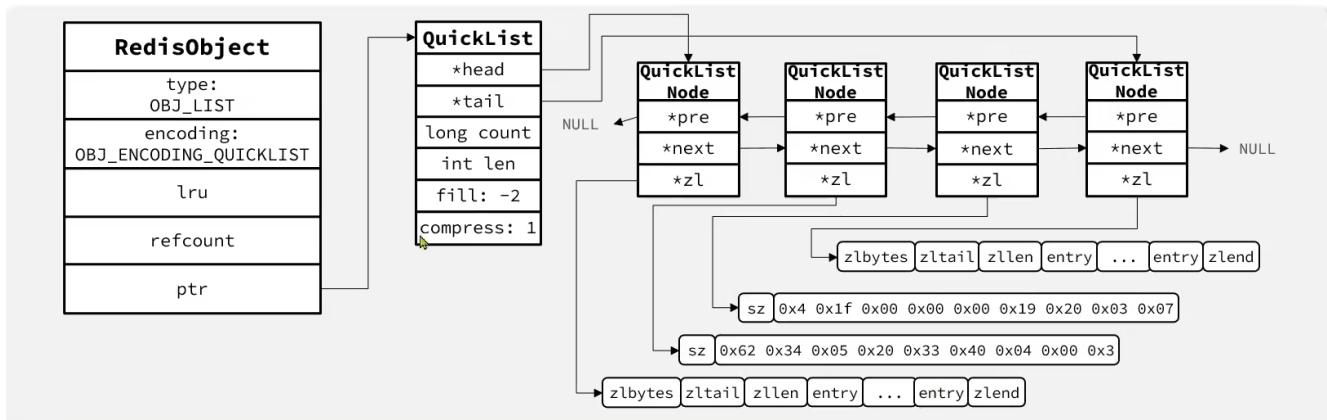


Figure 45: image-20250914152319695

## Set 特点:

- 不保证有序性
- 保证元素唯一 (判断元素是否存在)
- 求交集、并集、差集

### 编码方式:

- Set 会采用 **HT 编码 (Dict)**, Dict 中的 key 用来存储元素, **value** 统一为 **null**
- 当存储的所有数据都为整数, 并且元素数量不超过 **set-max-insert-entries** 时, Set 会采用 **IntSet** 编码, 以节省内存

## ZSet 也就是 SortedSet, 其中每一个元素需要指定一个 score 值和 member 值

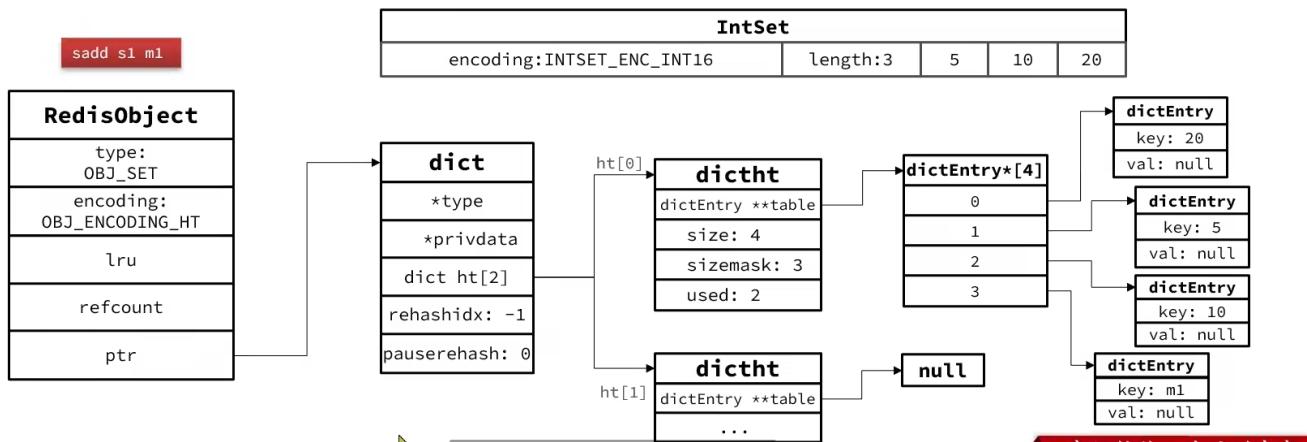


Figure 46: image-20250914153838878

- 根据 score 值排序
- member 必须唯一
- 可以根据 member 查询 score

#### 编码方式：

- SkipList: 可以排序, 可以同时存储 score 和 ele 值
- HT(Dict): 可以键值存储, 可以根据 key 找 value
- RedisObject head 的编码方式写的是 SkipList, 但是 ZSet 底层同时使用了上述两种编码方式
  - 性能很好但是非常占用内存

```

typedef struct zset{
    // Dict 指针
    dict *dict;
    // SkipList 指针
    zskiplist *zsl;
} zset;
    
```

- 当元素数量小于 `zset_max_ziplist_entries` 并且每个元素都小于 `zset_max_ziplist_value` 时才使用 ZipList 结构来存储元素
  - ZipList 是连续内存, 因此 score 和 element 是紧挨在一起的两个 entry
  - score 越小越接近队首, score 越大越接近队尾, 按照 score 升序排列

#### Hash 编码方式：底层采用的编码方式和 Zset 基本保持一致

- Hash 底层默认采用 ZipList, 用以节省内存, 相邻的两个 entry 分别保存 field 和 value
- 当数据量较大时, Hash 结构会转为 HT 编码, 也就是 Dict, 有以下两种触发时机
  - ZipList 中的元素超过了 `hash-max-ziplist-entries`
  - ZipList 中的任意 entry 大小超过了 `hash-max-ziplist-value`

#### Redis 网络模型

**用户空间和内核空间** 32 位操作系统内存上限是  $2^{32}$ , 也就是 4GB

## ZSet

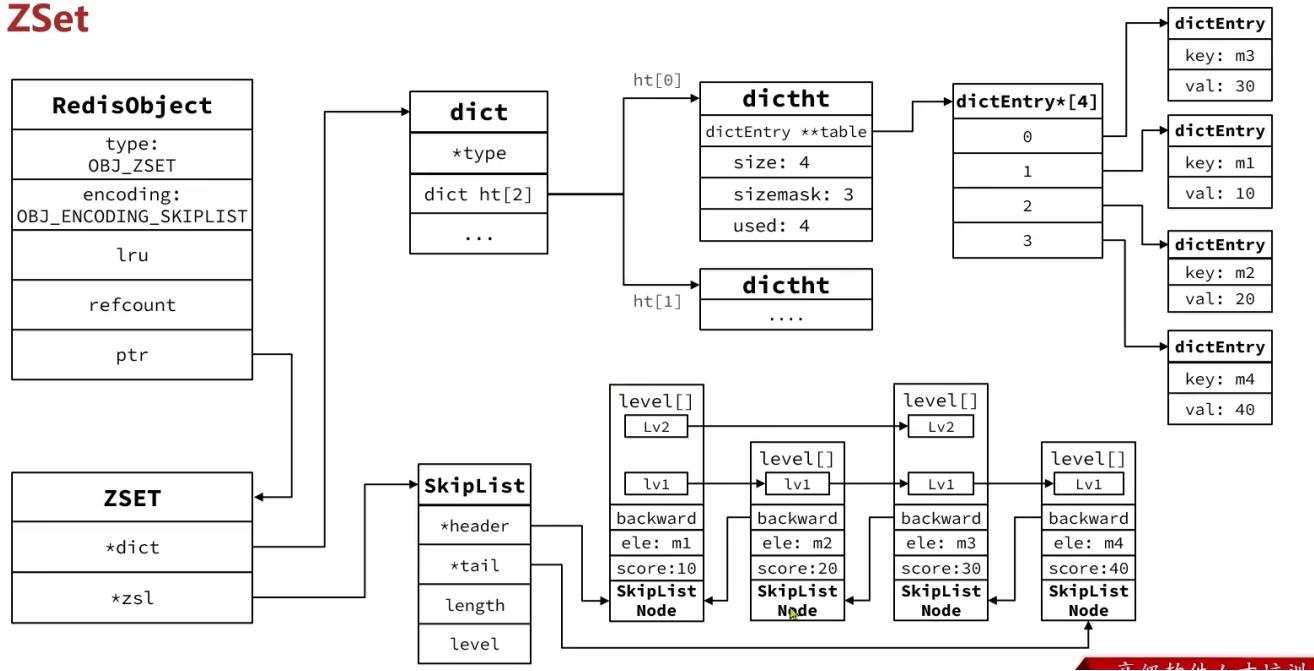


Figure 47: image-20250914161559970

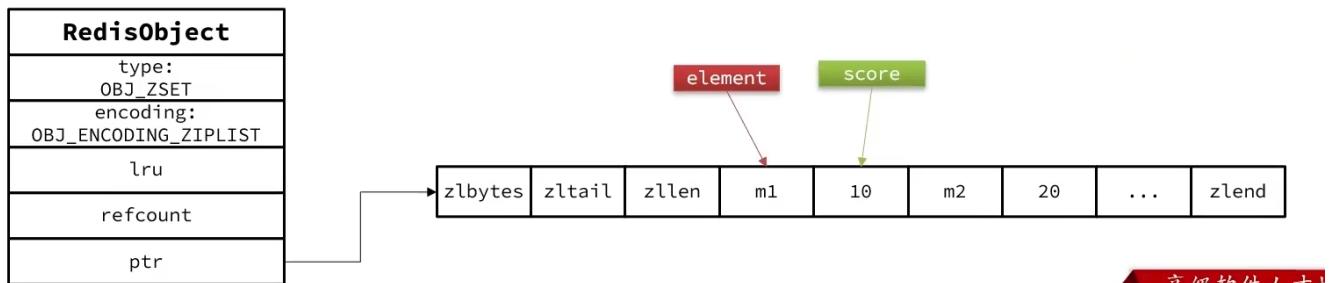


Figure 48: image-20250914162700600

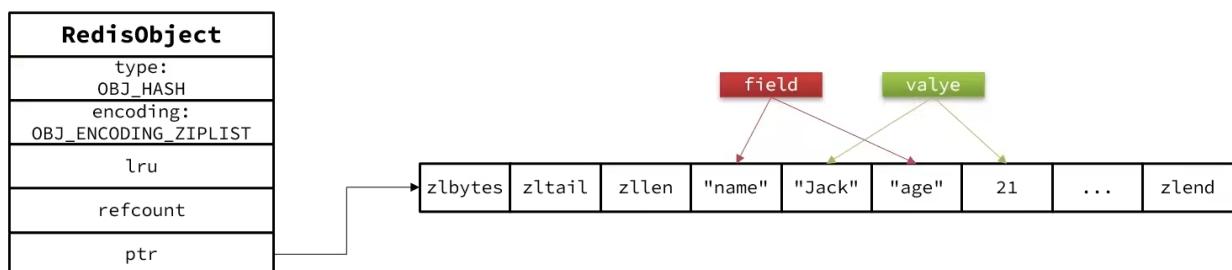


Figure 49: image-20250914163307382

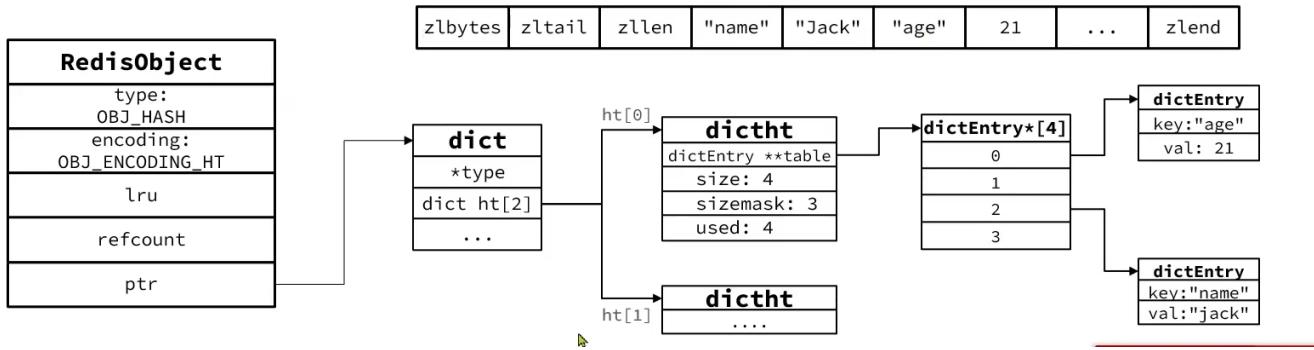


Figure 50: image-20250914163253268

为了避免用户应用导致冲突甚至内核崩溃，**用户应用和内核应用是分离的**（用户和内核能访问的内存是隔离的）

- 进程的寻址空间会划分为两部分：**内核空间、用户空间**
- **用户空间**只能执行受限的命令（Ring3），而不能直接调用系统资源，必须**通过内核提供的接口访问**
- **内核空间**可以执行特权命令（Ring0），调用一切系统资源
- 执行的进程可能会在用户空间和内核空间去做切换，在内核空间执行叫**内核态**，在用户空间执行叫**用户态**

以 **IO 读写**为例子，Linux 为了提高 IO 效率，会在用户空间和内核空间都加入**缓冲区**：

- 写数据时，要把**用户缓冲数据拷贝到内核缓冲区**，然后写入设备
- 读数据时，要从设备**读取数据到内核缓冲区**，然后**拷贝到用户缓冲区**

**提高 IO 效率的两个核心点**

- **减少读操作的无效等待时间**
- **减少用户态和内核态之间的数据拷贝**

**IO 操作数据读取的整个流程**

- **等待数据就绪**，等待硬件获取数据并将其读取到内核空间
- **读取数据**，将数据从内核空间拷贝到用户空间

**阻塞 IO (Blocking IO)** 顾名思义，阻塞 IO 就是**等待数据就绪和读取数据两个流程都需要阻塞等待**

- 用户应用请求获取数据之后，内核没有数据的话会执行等待，直到有数据为止

**非阻塞 IO (Nonblocking IO)** 非阻塞 IO 的 `recvfrom` 操作会**立即返回结果而不是阻塞用户进程**

- 用户应用请求获取数据，内核**如果没有数据会直接返回**，告诉用户应用没有数据
- 如果用户应用拿不到数据，其会**不断发起请求**，直到获取到数据为止
  - 性能并没有得到提高，因为用户应用拿不到数据会不断发送请求，**导致 CPU 使用率暴增**

**IO 多路复用 (IO Multiplexing)** 无论是阻塞 IO 还是非阻塞 IO，用户应用在**等待数据就绪阶段都需要调用 `recvfrom` 来获取数据**，差别在于无数据时的处理方案

- 如果调用 `recvfrom` 时，没有数据，**阻塞 IO** 会使进程阻塞，**非阻塞 IO** 使 CPU 空转，都不能充分发挥 CPU 作用
- 如果调用 `recvfrom` 时，有数据，用户进程可以直接读取并处理数据

**性能问题：**服务端处理客户端 Socket 请求时，单线程情况下只能依次处理每一个 Socket，如果正在处理的 Socket 恰好未就绪，线程就会被阻塞，导致所有的 Socket 请求都没法被处理，性能很差

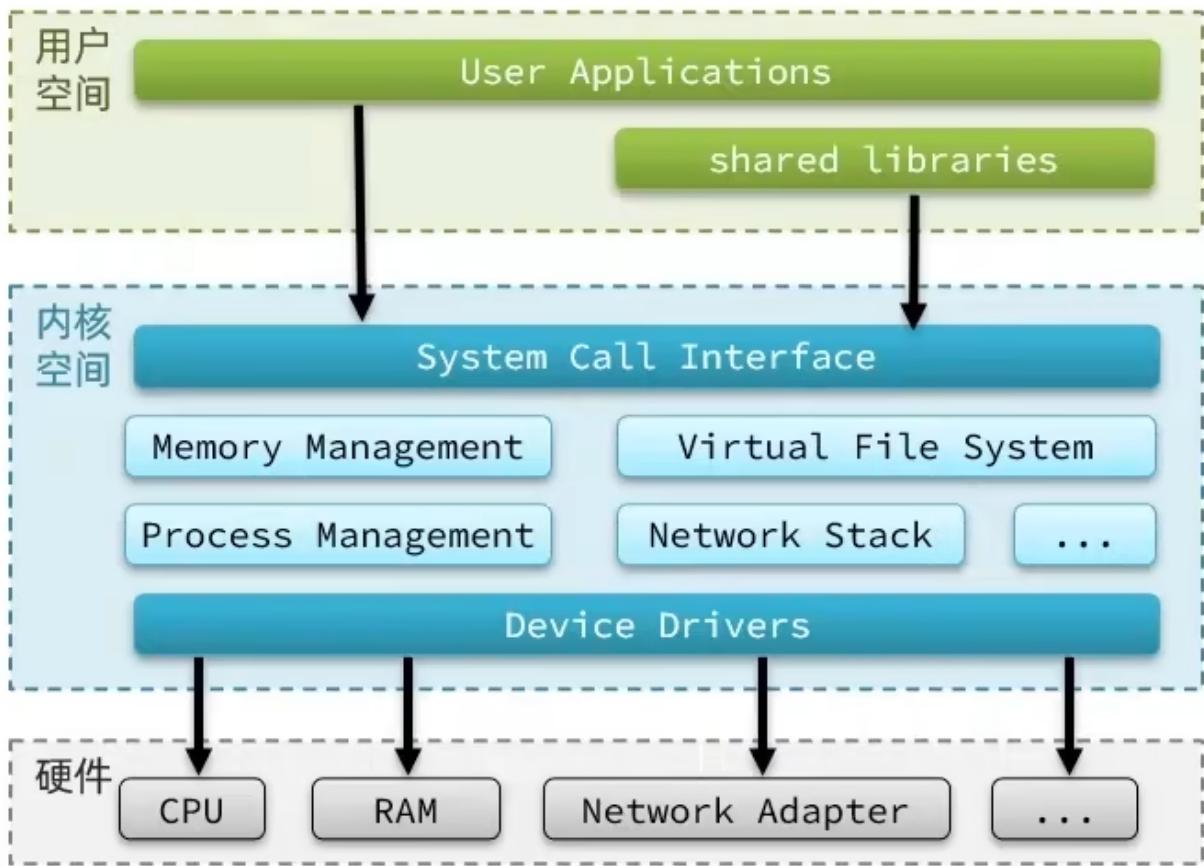


Figure 51: image-20250914165305109

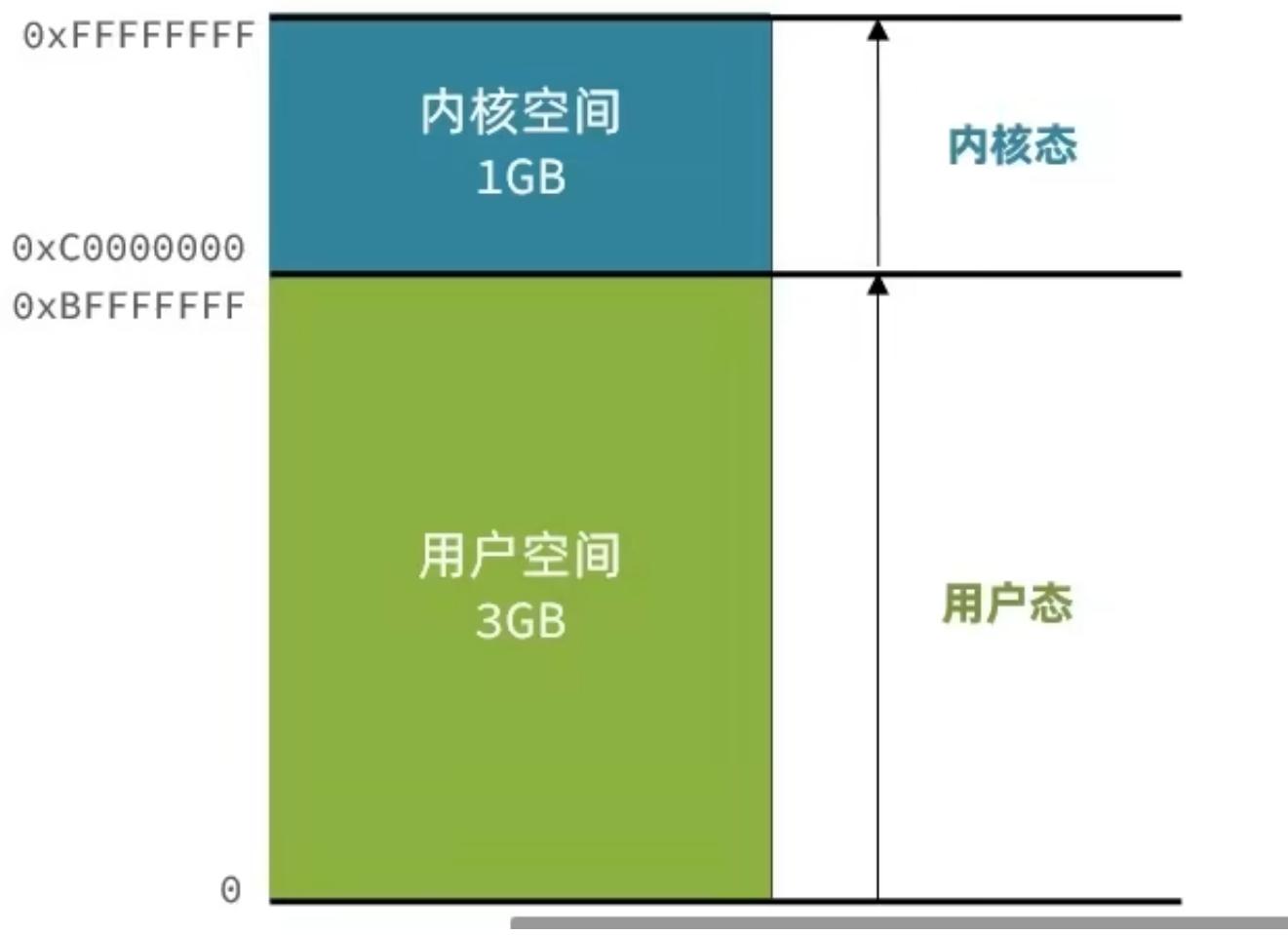


Figure 52: image-20250914165515712

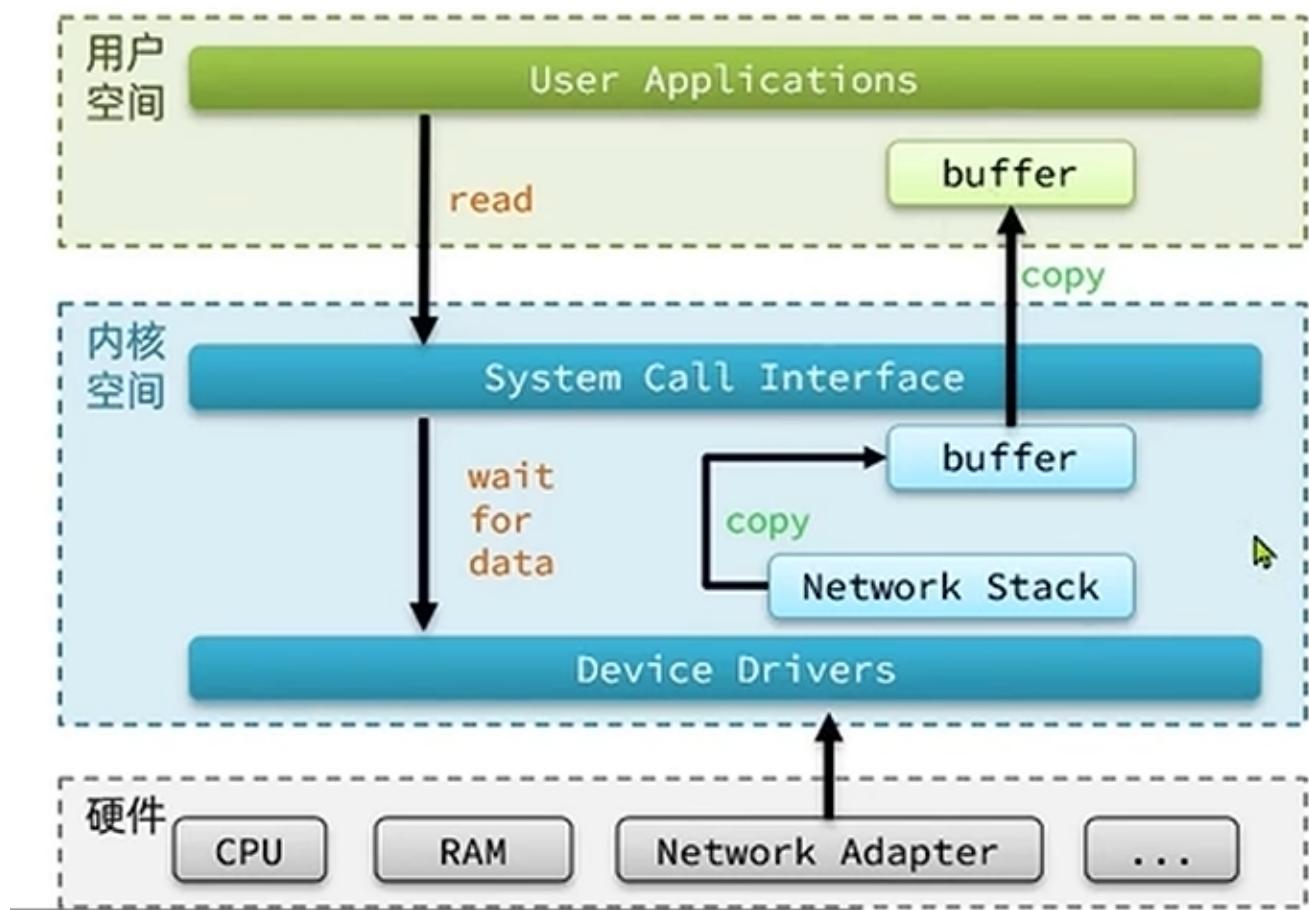


Figure 53: image-20250916121051349

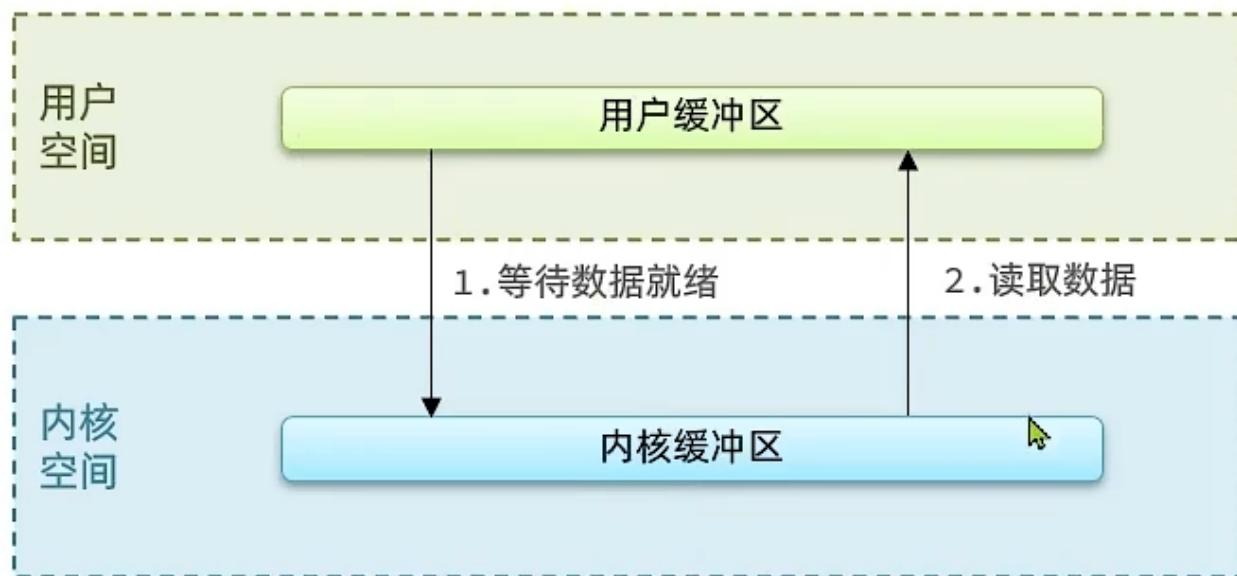


Figure 54: image-20250916121639658

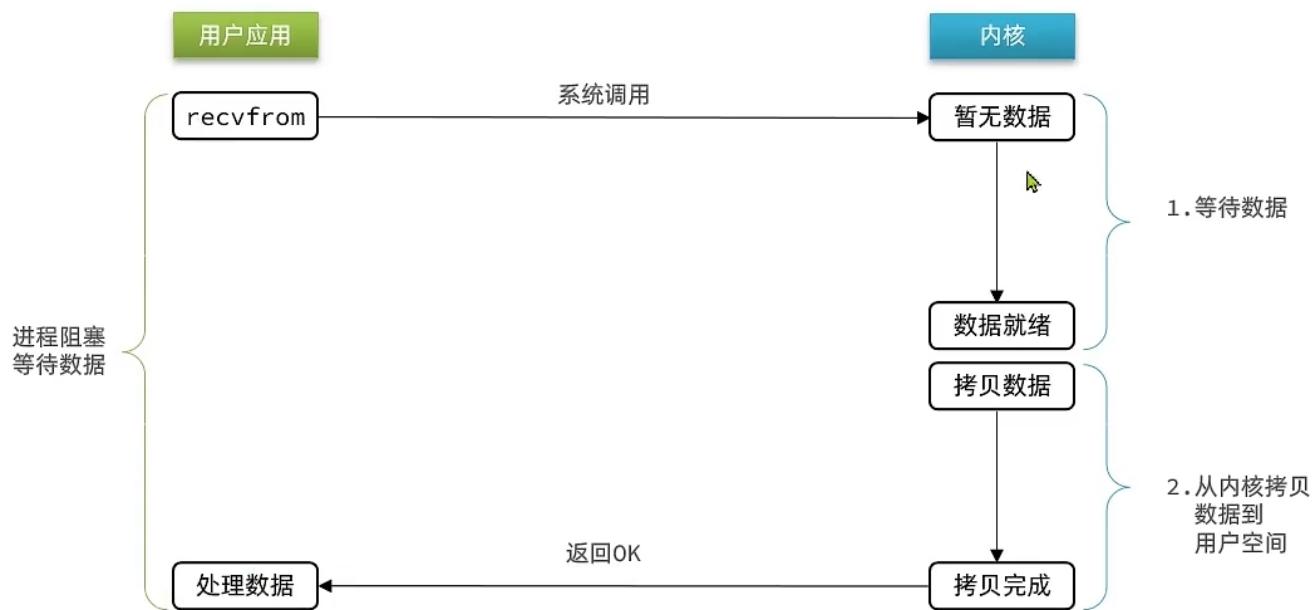


Figure 55: image-20250916121942982

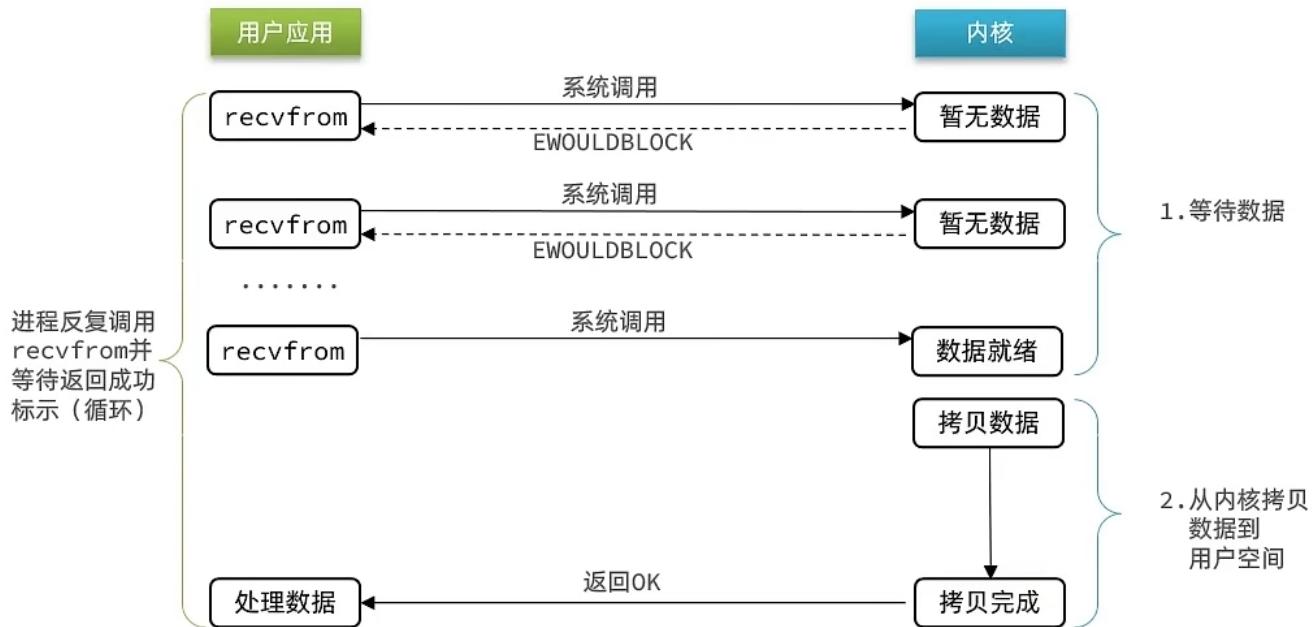


Figure 56: image-20250916122421915

## 方案

- 解决方法：用线程去监听所有的 Socket，当某个 Socket 数据准备就绪了，对应的用户应用就去读取数据
- 文件描述符：简称 FD，是一个从 0 开始递增的无符号整数，用来关联 Linux 的一个文件。在 Linux 中，一切皆文件，包括网络套接字（Socket）
- 核心思想：利用单个线程同时来监听多个 FD，并在某个 FD 可读、可写时得到通知，从而避免无效等待
  - 用户应用会调用 `select` 函数同时监听多个 FD
  - 当 FD 转变状态会返回 `readable` 给用户应用
  - 用户应用立刻调用 `recvfrom` 获取数据

不过监听 FD 的方式、通知的方式又有多种实现，常见的有：

- `select`
- `poll`
- `epoll`

差异：

- `select` 和 `poll` 只会通知用户进程有 FD 就绪，但不确定具体是哪个 FD，需要用户进程逐个遍历 FD 来确认
- `epoll` 会在通知用户进程 FD 就绪的同时，告诉他是哪个 FD 已经准备就绪

`select` 是 Linux 中最早的 I/O 多路复用实现方案

```
// 定义类型别名 __fd_mask, 本质上是 long int(占 4 个字节, 32 个 bit 位)
typedef long int __fd_mask;
```

```
// fd_set 记录要监听的 fd 集合以及对应的状态
typedef struct{
    // fds_bits 是 long int 类型数组, 长度为 1024/32=32
```

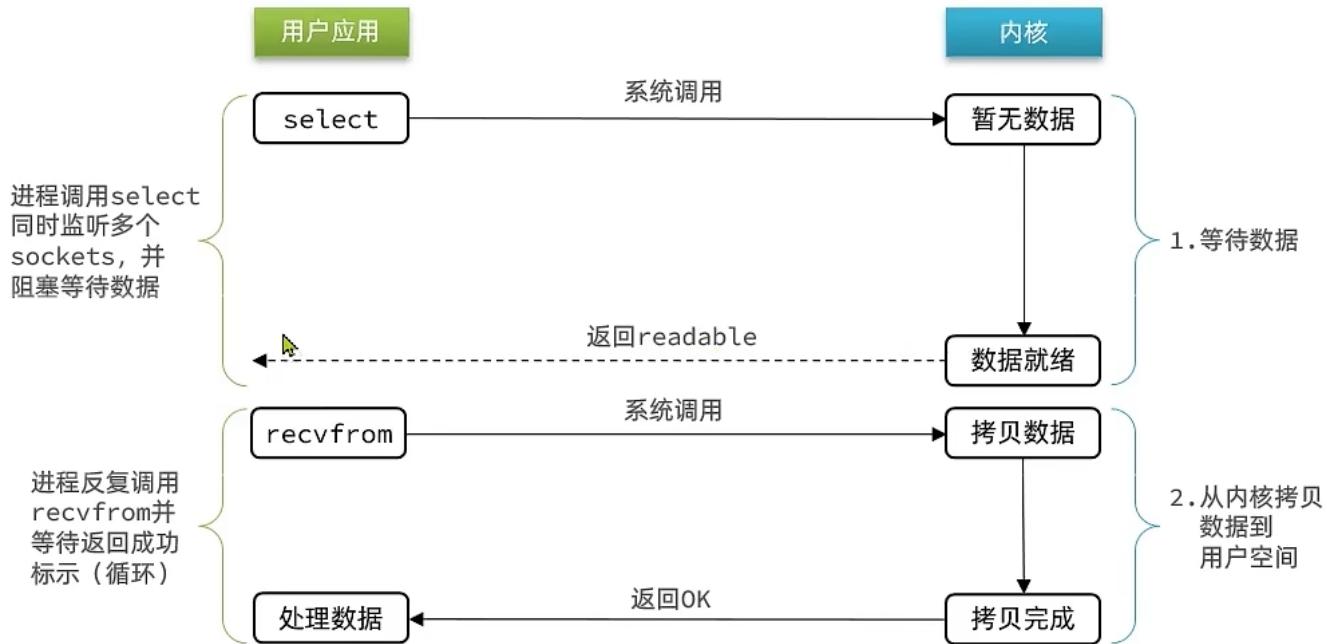


Figure 57: image-20250916134650343

```
// 共 1024 个 bit 位，每个 bit 位代表一个 fd，0 代表未就绪/忽略，1 代表就绪/监听
__fd_mask fds_bits[__FD_SETSIZE / __NFDBITS];
} fd_set;
```

```
// select 函数，用于监听多个 FD 的集合
int select(
    int nfds, // 要监视的 fd_set 的最大 fd+1
    fd_set *readfds, // 要监听读事件的 fd 集合
    fd_set *writefds, // 要监听写事件的 fd 集合
    fd_set *exceptfds, // 要监听异常事件的 fd 集合
    // 超时时间，null-永不超时，0-不阻塞等待，大于 0-固定等待时间
    struct timeval *timeout
)
```

### 工作流程：

1. 用户应用创建 `fd_set`, 将要监听的 `fd` 设置成 1, `nfds` 为要监听的 `fd` 的最大值加 1
2. 执行 `select` 函数, 将 `fd_set` 从用户空间拷贝到内核空间
3. 内核遍历 `fd_set`, 没有就绪线程则休眠等待, 如果有 `fd` 可读线程就会被唤醒, 遍历 `fd_set` 找到和当前已就绪的 `fd` 作比较
4. 就绪的 `fd` 进行保留, 未就绪的 `fd` 改成 0
5. `select` 函数返回就绪的 `fd` 的值, 并且将修改后的 `fd_set` 从内核空间拷贝回用户空间, 覆盖用户空间的 `fd_set` 集合
6. 用户空间遍历 `fd_set` 找到已就绪的 `fd`, 读取其中的数据

### 存在的问题：

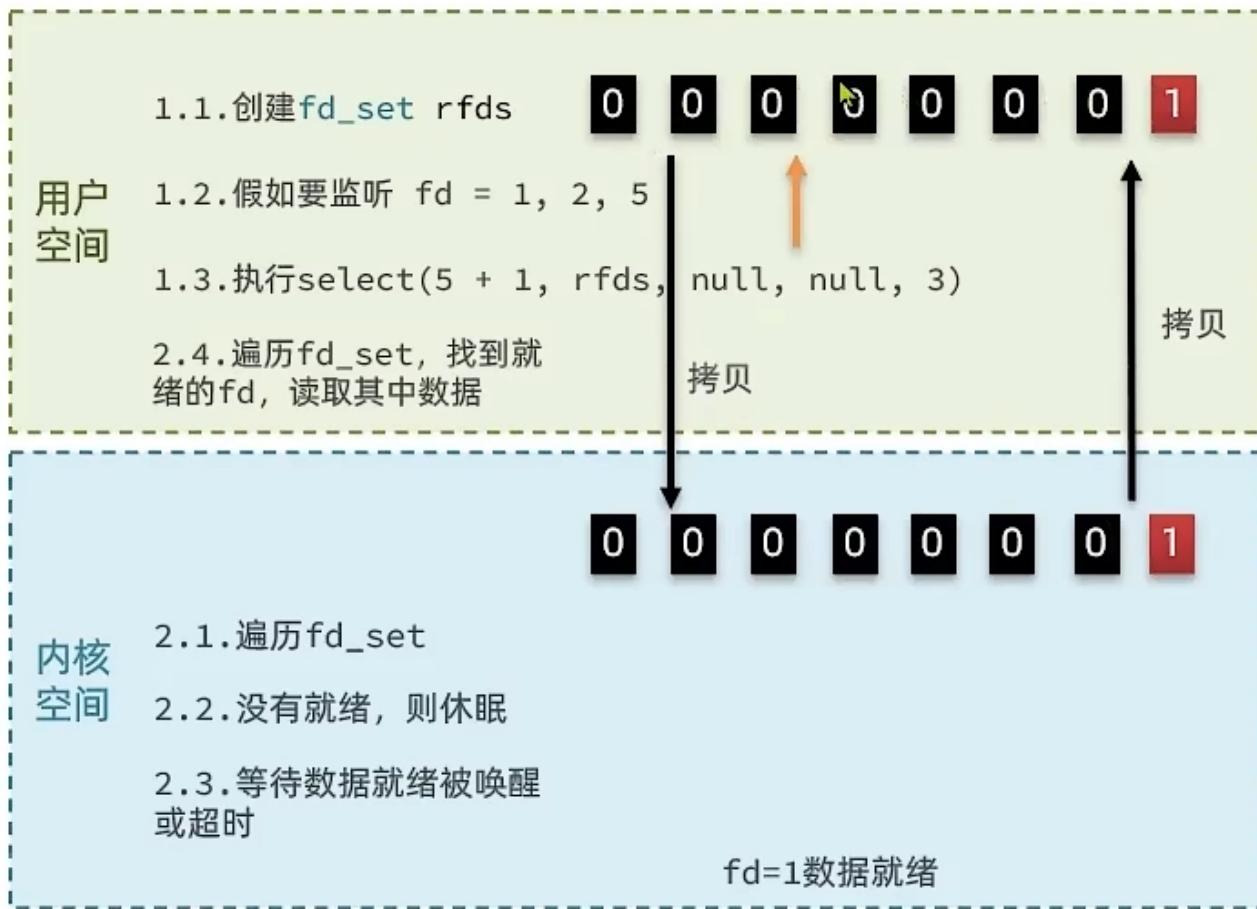


Figure 58: image-20250916143656363

- 执行一次 select 函数需要在用户空间和内核空间执行**两次**数据拷贝
- select 无法得知具体是哪个 fd 就绪，需要遍历整个 fd\_set
- fd\_set 能监听的 fd 数量不多

## poll 源码

```
// pollfd 中的事件类型
#define POLLIN //可读事件
#define POLLOUT //可写事件
#define POLLERR //错误事件
#define POLLNVAL //fd 未打开

// pollfd 结构
struct pollfd {
    int fd; //要监听的 fd */
    short int events; /* 要监听的事件类型：读、写、异常 */
    short int revents; /* 实际发生的事件类型 */
};

// poll 函数
int poll(
    struct pollfd *fds, //pollfd 数组，可以自定义大小
    nfds_t nfds, //数组元素个数
    int timeout //超时时间
);
```

## 工作流程：

1. 创建 pollfd 数组，向其中添加 fd 信息，数组大小自定义
2. 调用 poll 函数，将 pollfd 数组拷贝到内存空间，**转链表存储**
3. 内核遍历 fd，判断是否就绪
4. 数据就绪或超时后，内核在 revents 写入**实际发生的事件类型**（没事件发生则为 0），拷贝 pollfd 数组到用户空间，返回就绪 fd 数量 n
5. 用户进程判断 n 是否大于 0
6. 大于 0 则遍历 pollfd 数组，找到就绪的 fd

## epoll 是对 select 和 poll 的改进，它提供了**三个函数**

```
struct eventpoll{
    struct rb_root rbr; // 一颗红黑树，记录要监听的 FD(epitem)
    struct list_head rdList; // 一个链表，记录就绪的 FD(epitem)
}

// 1. 会在内核创建 eventpoll 结构体，返回对应的句柄 efd
int epoll_create(int size);

// 2. 将一个 FD 添加到 epoll 的红黑树中，并设置 ep_poll_callback 回调函数
// callback 触发时，就把对应的 fd 加入到 rdlist 这个就绪列表中
int epoll_ctl(
    int efd, // epoll 实例的句柄
```

```

int op, // 要执行的操作，包括：ADD、MOD、DEL
int fd, // 要监听的 fd
struct epoll_event *event // 要监听的事件类型：读、写、异常等
);

// 3. 检查 rdlst 列表是否为空，不为空则返回就绪的 FD 的数量
int epoll_wait(
    int epfd, // eventpoll 实例的句柄
    struct epoll_event *events, // 空 event 数组，用于接收就绪的 FD
    int maxevents, // events 数组的最大长度
    int timeout // 超时时间，-1 永不超时，0 不阻塞，大于 0 为阻塞时间
);

```

## 工作流程

1. 用户应用调用 `epoll_create` 方法创建 `eventpoll` 结构体，得到结构体的**句柄**（可以理解成资源的唯一标识，**可以利用其向内核发起调用，但是不直接指向内存地址**）
2. 用户应用把要监听的 FD 通过 `epoll_ctl` 方法加入到 `eventpoll` 结构体中的红黑树中，并设置 `ep_poll_callback` 回调函数，这个回调函数会在 FD 就绪时触发，把就绪的 FD 添加到 `eventpoll` 的链表中
3. 用户应用调用 `epoll_wait` 方法获取就绪的 FD(`epitem`)，通过 `eventpoll` 结构体中的链表获得就绪的 FD(`epitem`)，并把就绪的 FD(`epitem`) 从内核空间拷贝回用户空间的 `epoll_event` 数组
4. 用户空间直接遍历 `epoll_events` 这个数组的 FD(`epitem`)，进行 IO 操作

## 事件通知机制

当 FD 有数据可读时，调用 `epoll_wait` 就可以得到通知，但是事件通知的模式有两种：

- **LevelTriggered**: 简称 **LT**。当 FD 有数据可读时，会**重复通知多次**，直到数据处理完成，是 Epoll 的**默认模式**
- **EdgeTriggered**: 简称 **ET**。当 FD 有数据可读时，**只会被通知一次**，不管数据是否处理完成

## 结论：

- ET 模式避免了 LT 模式可能出现的**惊群现象**
  - 惊群现象就是假设多个进程来监听同一个 FD，LT 模式会给每一个进程都进行通知，但实际上前两个进程就可以把数据处理完，后续被通知到的进程就浪费了
- ET 模式最好结合**非阻塞 IO 读取 FD 数据**，相比 LT 更复杂

## IO 多路复用的 Web 服务流程

- 创建 `serverSocket`，得到其 `fd`，记作 `ssfd`，将其添加到红黑树中，并绑定一个回调函数
- 当客户端尝试连接 `serverSocket`，`ssfd` 会变成可读
- 并且接收客户端 `socket`，得到对应 `fd`，将其加入红黑树

**信号驱动 IO (Signal Driven IO)** 与内核建立 **SIGIO** 的信号关联并设置回调，当内核有 FD 就绪时，会发出 **SIGIO** 信号通知用户，期间用户应用可以执行其它业务

## 缺点：

- 当有大量 IO 操作的时候，信号较多，SIGIO 处理函数不能及时处理可能导致消息队列溢出
- 内核空间和用户空间频繁信号，性能低

**异步 IO (ASynchronous IO)** 整个过程都是**非阻塞的**，用户进程调用完**异步 API** 后就可以去做其他事情，内核等待数据就绪并拷贝到用户空间后才会递交信号，通知用户进程

## 缺点：

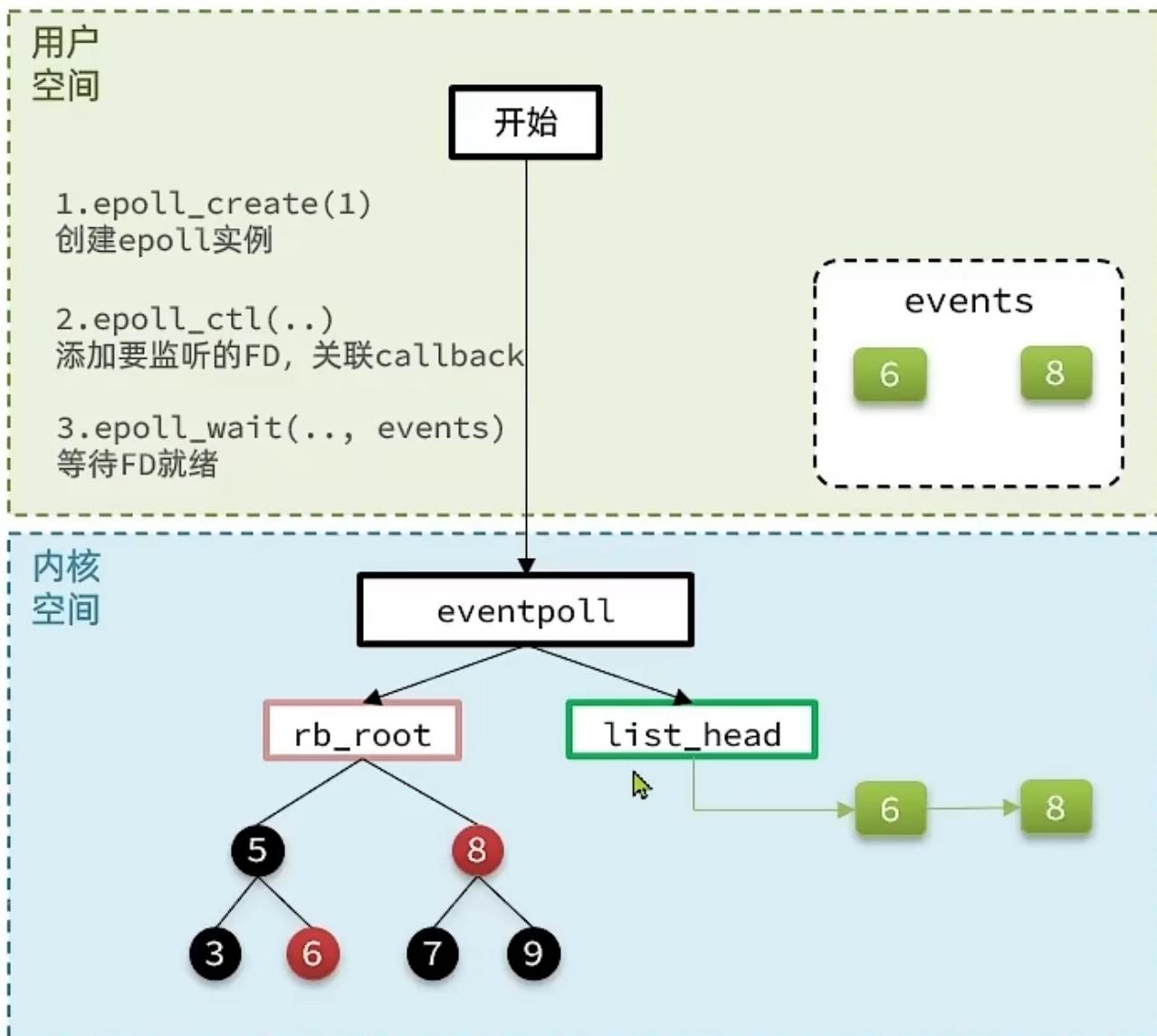


Figure 59: image-20250916152045771

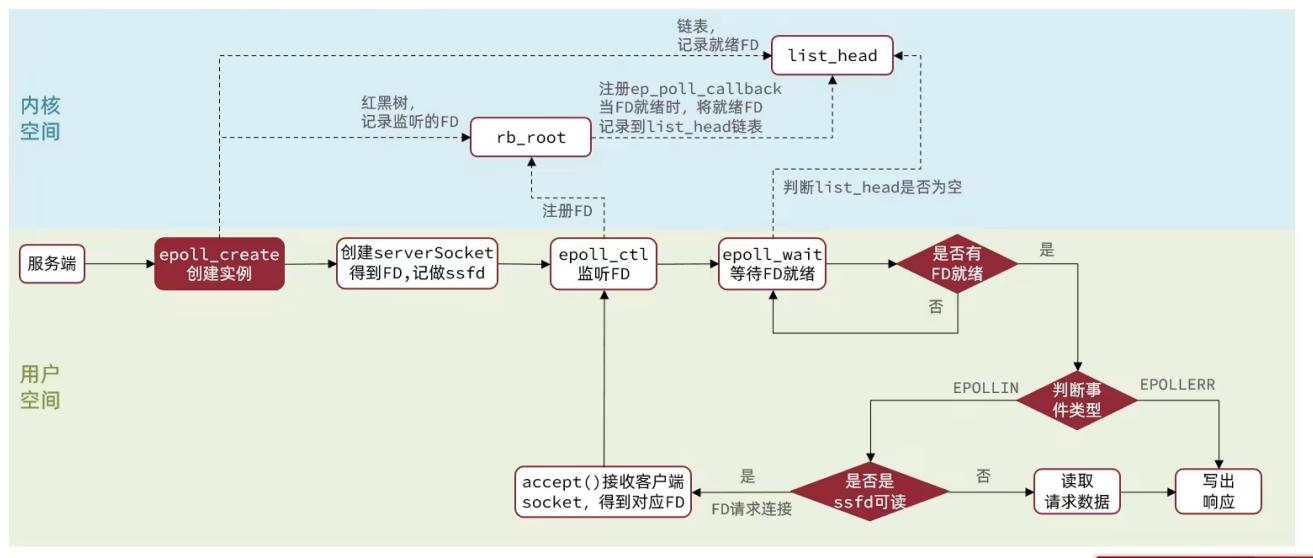


Figure 60: image-20250917182625014

需阻塞等待。

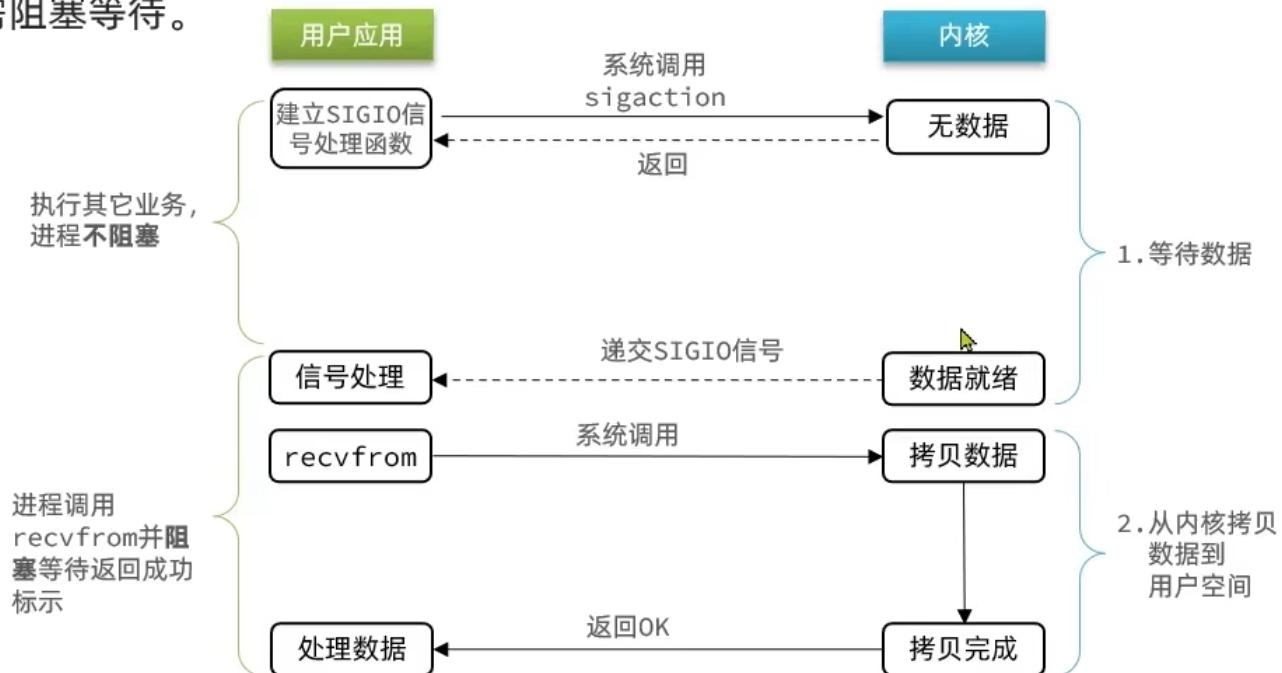


Figure 61: image-20250917183101977

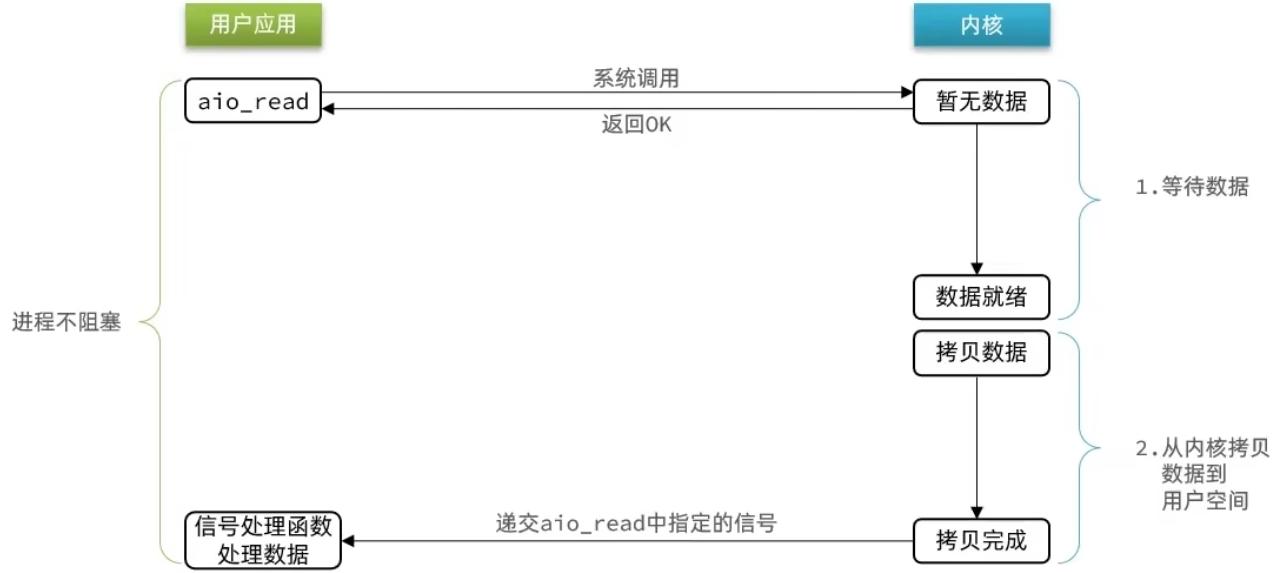


Figure 62: image-20250917183543485

- 高并发下性能不好

**同步和异步** IO 操作是同步还是异步，关键看数据在内核空间和用户空间的拷贝过程是同步还是异步

**单线程和多线程** Redis 是单线程还是多线程？

- 如果是 Redis 的核心业务部分（命令处理），是单线程
- 如果是整个 Redis，那么就是多线程

为什么 Redis 要选择单线程？

- Redis 是纯内存操作，性能瓶颈是网络延迟而不是执行速度
- 多线程会导致过多的上下文切换，造成不必要的开销
- 引入多线程会面临线程安全问题，必然要引入线程锁，实现复杂度会提高

### Redis 网络模型

- 多线程主要在写入请求数据到缓冲区和处理队列中的任务时使用
- 处理请求的流程如下
  - Redis 将请求数据写入查询缓冲区
  - 解析缓冲区字符串，转化为 Redis 命令
  - 主线程执行解析好的 Redis 命令
  - 把结果写到缓冲区或链表（缓冲区存不下）
  - 主线程把结果响应给客户端

### RESP 协议

Redis 是一个 CS 架构的软件，通信一般分为两步

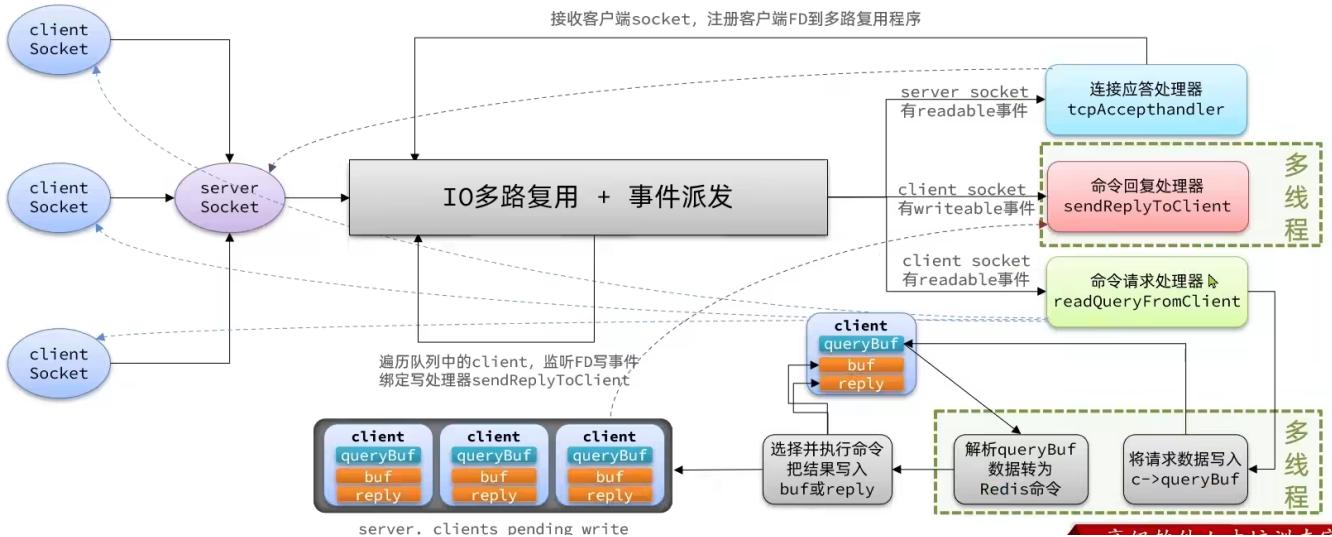


Figure 63: image-20250927102155796

1. 客户端向服务端发送一条命令
2. 服务端解析并执行命令，返回响应结果给客户端

因此客户端发送命令和服务端响应结果的格式的规范就是通信协议

Redis 中采用的是 **RESP 协议**

在 RESP 中，通过首字母的字符来区分不同数据类型，常见的数据类型包括五种

- **单行字符串**: 首字节是'+'，后面跟上单行字符串，以 CRLF("\r\n") 结尾
  - 字符串中不能包含"\r\n"，因此是二进制不安全的
- **错误 (ERRORS)**: 首字节是'-'，与单行字符串格式一样，但字符串是异常信息
- **数值类型**: 首字节是':'，后面跟上数字格式的字符串，以 CRLF 结尾
- **多行字符串**: 首字节是'\$'，表示**二进制安全的字符串**，最大支持 512MB
  - 紧跟字符串长度和字符串数据
  - 如果大小为 0，代表**空字符串**
  - 如果大小为-1，代表**不存在**
- **数组**: 首字节是'\*'，后面跟上数组元素个数，再跟上元素，**元素数据类型不限**

## 内存策略

**内存过期策略** 可以通过 expire 命令给 Redis 的 key 设置 **TTL (存活时间)**

**DB 结构** Redis 本身是一个典型的 **key-value 内存存储数据库**，因此所有的 key、value 都保存在之前学习的 Dict 结构中

不过在其 database 结构体中，有两个 Dict，一个用来记录 **key-value**，另一个用来记录 **key-TTL**

```
typedef struct redisDb{
    dict *dict;                                // 存放所有 key 及 value 的地方
    dict *expires;                             // 存放每一个 key 及其对应的 TTL，只包含设置了 TTL 的 key
```

例

\$5\r\nhello\r\n

字符串占用  
字节大小

真正的字符串数据

Figure 64: image-20250918105136261

例

\*3\r\n

→ 数组元素个数

\$3\r\nset\r\n

\$4\r\nname\r\n

\$6\r\n虎哥\r\n

数组元素

Figure 65: image-20250918105158262

```

dict *blocking_keys;           // 
dict *ready_keys;             // 
dict *watched_keys;           // 
int id;                      // 数据库 ID
long long avg_ttl;           // 记录平均 TTL 时长
unsigned long expires_cursor; // expire 检查时在 dict 中抽样的索引位置
list *defrag_later;          // 等待碎片整理的 key 列表
} redisDb;

```

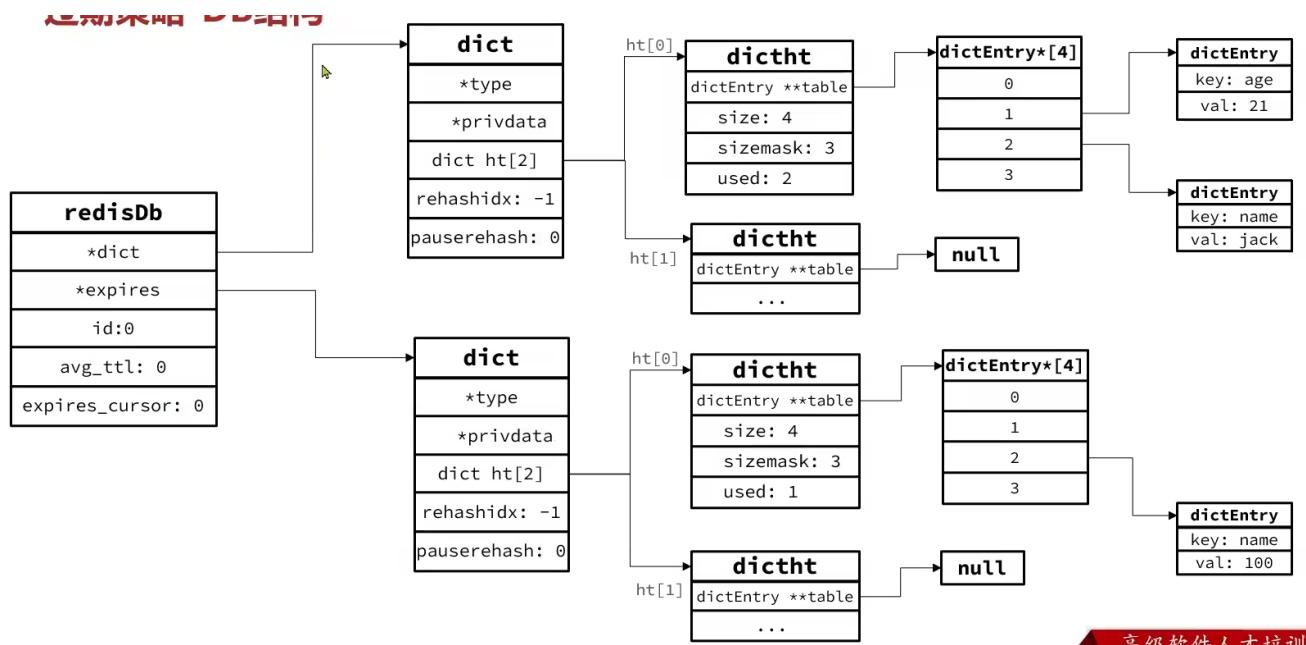


Figure 66: image-20250918105635565

**惰性删除** 并不是在 **TTL 到期后立即删除**，而是在访问一个 **key** 的时候，检查该 **key** 的存活时间，如果已经过期才执行删除

**周期删除** 通过一个定时任务，周期性抽样部分过期的 **key**，然后执行删除

- Redis 会设置一个定时任务 `serverCron()`，按照的频率来执行过期 key 清理，**模式为 SLOW**
  - 执行频率受 `server.hz` 影响，默认为 10，每秒执行 10 次，**每个执行周期 100ms**
  - 执行清理耗时**不超过一次执行周期的 25%**
  - 逐个遍历 db，逐个遍历 db 中的 bucket，抽取 **20 个 key 判断是否过期**
- Redis 的每个事件循环前会调用 `beforeSleep()` 函数，执行过期 key 清理，**模式为 FAST**
  - 执行频率受 `beforeSleep()` 调用频率影响，**两次 FAST 模式间隔不低于 2ms**
  - 执行清理耗时**不超过 1ms**
  - 逐个遍历 db，逐个遍历 db 中的 bucket，抽取 **20 个 key 判断是否过期**

**内存淘汰策略** 内存淘汰：当 Redis 中的内存使用达到设置的阈值时，Redis 主动挑选部分 **key** 删除以释放更多内存的流程

Redis 支持 **8 种不同策略来选择要删除的 key**

- `noeviction`: 不淘汰任何 key，内存满的时候不允许写入新数据
- `volatile-ttl`: 对设置了 TTL 的 key，比较 key 的剩余 TTL 值，TTL 越小越先被淘汰
- `allkeys-random`: 对全体 key，随机进行淘汰
- `volatile-random`: 对设置了 TTL 的 key，随机进行淘汰
- `allkeys-lru`: 对全体 key，基于 **LRU 算法**进行淘汰
- `volatile-lru`: 对设置了 TTL 的 key，基于 **LRU 算法**进行淘汰
- `allkeys-lfu`: 对全体 key，基于 **LFU 算法**进行淘汰
- `volatile-lfu`: 对设置了 TTL 的 key，基于 **LFU 算法**进行淘汰

比较容易混淆的：

- **LRU (Least Recently Used)**: 最少最近使用，用当前时间减去最后一次访问时间，值大的优先淘汰
- **LFU (Least Frequently Used)**: 最少频率使用，会统计每个 key 的访问频率，用 255-LFU 作为值，值小的优先淘汰

Redis 通过 `RedisObject` 来统计最近访问时间和访问频率

```
typedef struct redisObject{
    unsigned type:4; // 对象类型，分别是 string,hash,list,set 和 zset
    unsigned encoding:4; // 底层编码方式，共有 11 种
    unsigned lru:LRU_BITS;// LRU: 以秒为单位记录最近一次访问时间，长度为 24bit
                           // LFU: 高 16 位以分钟为单位记录最近一次访问时间，低 8 位记录逻辑访问次数
    int refcount; // 对象引用计数器，计数器为 0 则说明对象无人引用，可以被回收
    void *str; // 指向存放实际数据的内存空间，意味着对象和存储的数据内存空间不连续
} robj;
```

LFU 的访问次数叫做**逻辑访问次数**，是因为并不是统计 key 被访问的真实次数，而是通过运算：

1. 生成 0~1 的随机数 R
2. 计算  $1 / (\text{旧次数} * \text{lru\_log\_factor} + 1)$ ，记录为 P，`lru_log_factor` 默认为 10
3. 如果  $R < P$ ，则计数器加一，不超过 255
4. 访问次数会随时间衰减，距离上一次访问时间每隔 `lru_decay_time` 分钟（默认 1），计数器减一

## 逻辑

- 如果访问次数多了，生成的 P 会越来越小，则步骤三  $R < P$  的概率就会很小，这样子它记录的就是一个概率性的次数
- 如果 key 一直被访问，计数器的值也是会达到 255 的

## 流程图

## 最佳实践

### 键值设计

Redis 的 key 最好遵循下面的约定：

- 遵循基本格式：[业务名称]:[数据名]:[id]
- 长度不超过 44 字节，防止 String 的**编码类型变成 Raw**
- 不包含特殊字符

要防止出现 BigKey

- 单个 key 的 value 小于 **10kb**
- 对于集合类型的 key，建议元素数量小于 **1000**

### 常见命令

## 淘汰策略

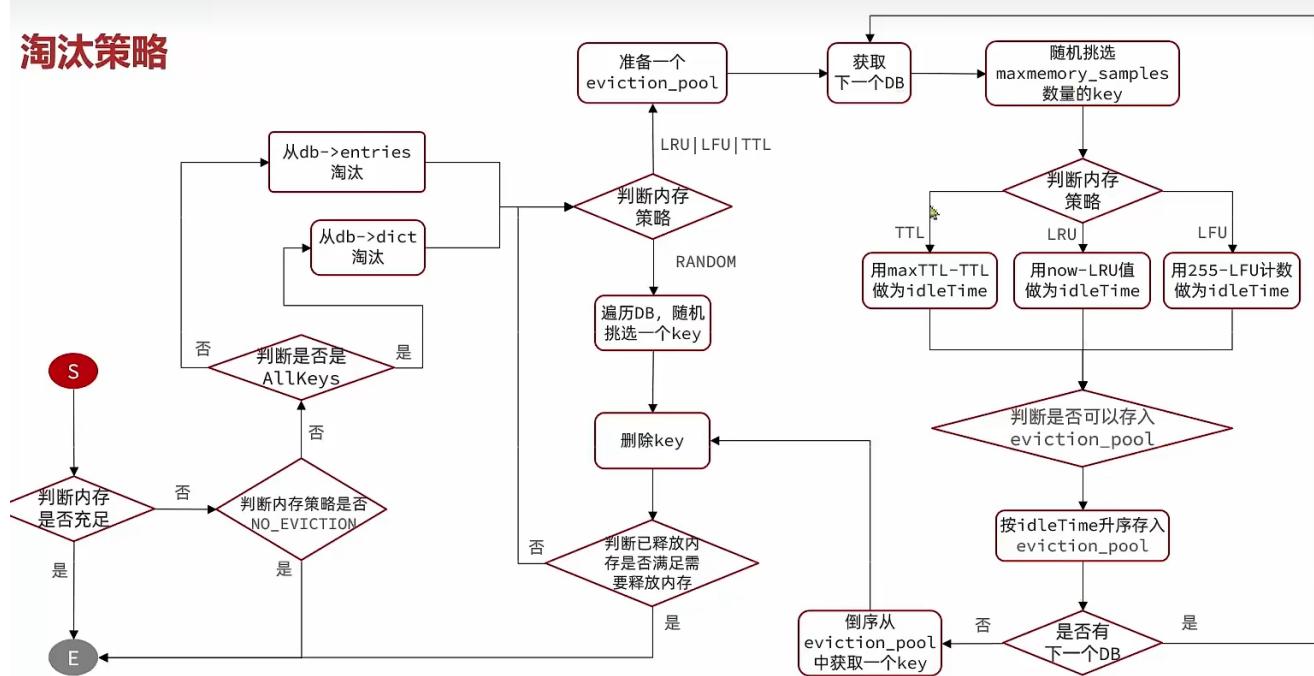


Figure 67: image-20250918113723379

- 使用 redis-cli --bigkeys
  - 返回 Key 的整体统计信息和每个数据的 Top1 的 big key
- 使用 scan 命令扫描所有的 key
  - 不会占用 Redis 的主线程!!
  - 生产环境中不适用 keys \* 查看所有 key, 会阻塞 Redis 主线程
- 使用 unlink 异步删除 big key, 不阻塞主线程

## 批处理优化

Redis 执行命令的速度非常快, 实际读写操作的耗时主要体现在网络延迟上, 可以使用**批处理**减少实际耗时

### 单节点批处理

- 可以使用 mset、hmset 实现**批量插入**
- 可以使用 Pipeline 进行命令**批处理, 非原子性!!!**
- 

### 集群批处理

- 直接使用 Spring 提供的工具类, 默认使用并行 slot 方案

## 服务端优化

### 持久化配置

- 缓存的 Redis 不要开启持久化功能
- 建议使用 AOF 持久化
- 利用脚本定期在 slave 节点做 RDB, 实现数据备份
- 配置 no-appendfsync-on-rewrite=yes, 禁止在 AOF 的 rewrite 或者 RDB 的 fork 期间做 AOF

	串行命令	串行slot	并行slot	hash_tag
实现思路	for循环遍历，依次执行每个命令	在客户端计算每个key的slot，将slot一致分为一组，每组都利用Pipeline批处理。 串行执行各组命令	在客户端计算每个key的slot，将slot一致分为一组，每组都利用Pipeline批处理。 并行执行各组命令	将所有key设置相同的hash_tag，则所有key的slot一定相同
耗时	N次网络耗时 + N次命令耗时	m次网络耗时 + N次命令耗时 $m = \text{key的slot个数}$	1次网络耗时 + N次命令耗时	1次网络耗时 + N次命令耗时
优点	实现简单	耗时较短	耗时非常短	耗时非常短、实现简单
缺点	耗时非常久	实现稍复杂 slot越多，耗时越久	实现复杂	容易出现数据倾斜

Figure 68: image-20250926120009590

**慢查询：**在 Redis 执行时耗时超过某个阈值的命令

- 配置慢查询的阈值：slowlog-log-slower-than，单位是**微妙**

慢查询会被放入慢查询日志中，日志长度可通过配置指定：

- slowlog-max-len：慢查询日志的长度

查看慢查询日志列表：

- slowlog len：查询慢查询日志长度
- slowlog get[n]：读取 n 条慢查询日志
- slowlog reset：清空慢查询列表

## 利用 Redis 入侵服务器

- 如果 Redis 没有设置密码且外网可以访问，黑客可以将他的公钥通过 Redis 存储到服务器硬盘
- 黑客登录 Redis，将公钥存储到 redis，修改其配置文件，更改配置文件的存储目录，执行 RDB，把密钥存储在硬盘
- 利用 ssh 进行免密登陆服务器

## 集群最佳实践

- 如果发现任意一个插槽不可用，则整个集群都会停止服务
  - 修改配置文件内容：cluster-require-full-coverage no
- 如果一个集群包含太多节点，相互之间的 ping 会携带很多的数据量，**所需要的带宽也会非常高**
  - 避免大集群，单个物理机不要运行太多 Redis 实例