# redis五种数据类型

redis本身是一种键值型数据库，键可以简单理解为“永远都是字符串类型”

而值根据类型的不同可分为五种常见的数据类型：string, list, hash, set,和sorted set

## string(字符串)

redis的键都是字符串类型，而值可以是字符串类型。

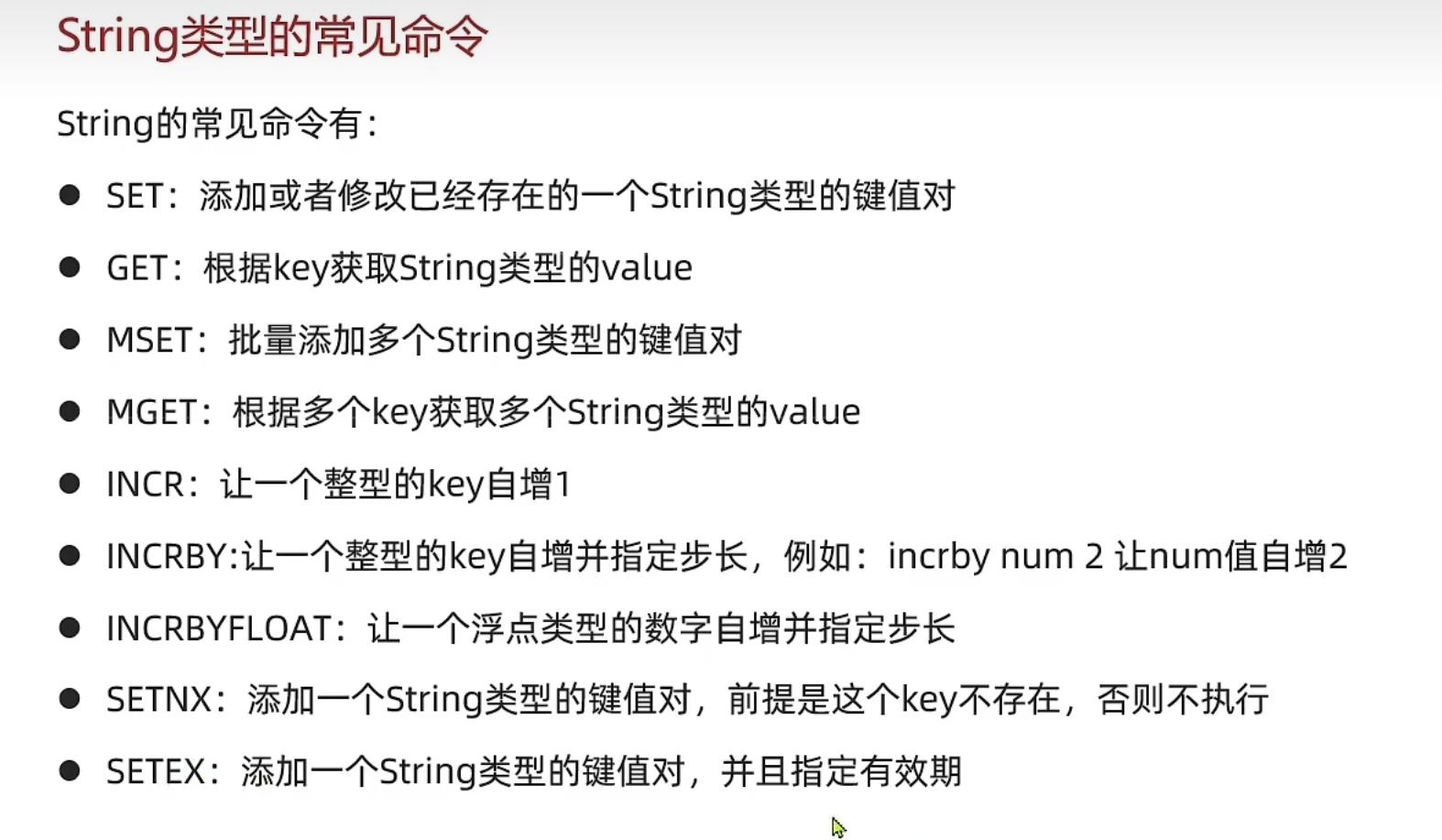
字符串类型按照存储形式又分为三种：1.字符串型2整型.3.浮点型

其中整型和浮点型的值，可以做自增和自减操作

具体解释如下：



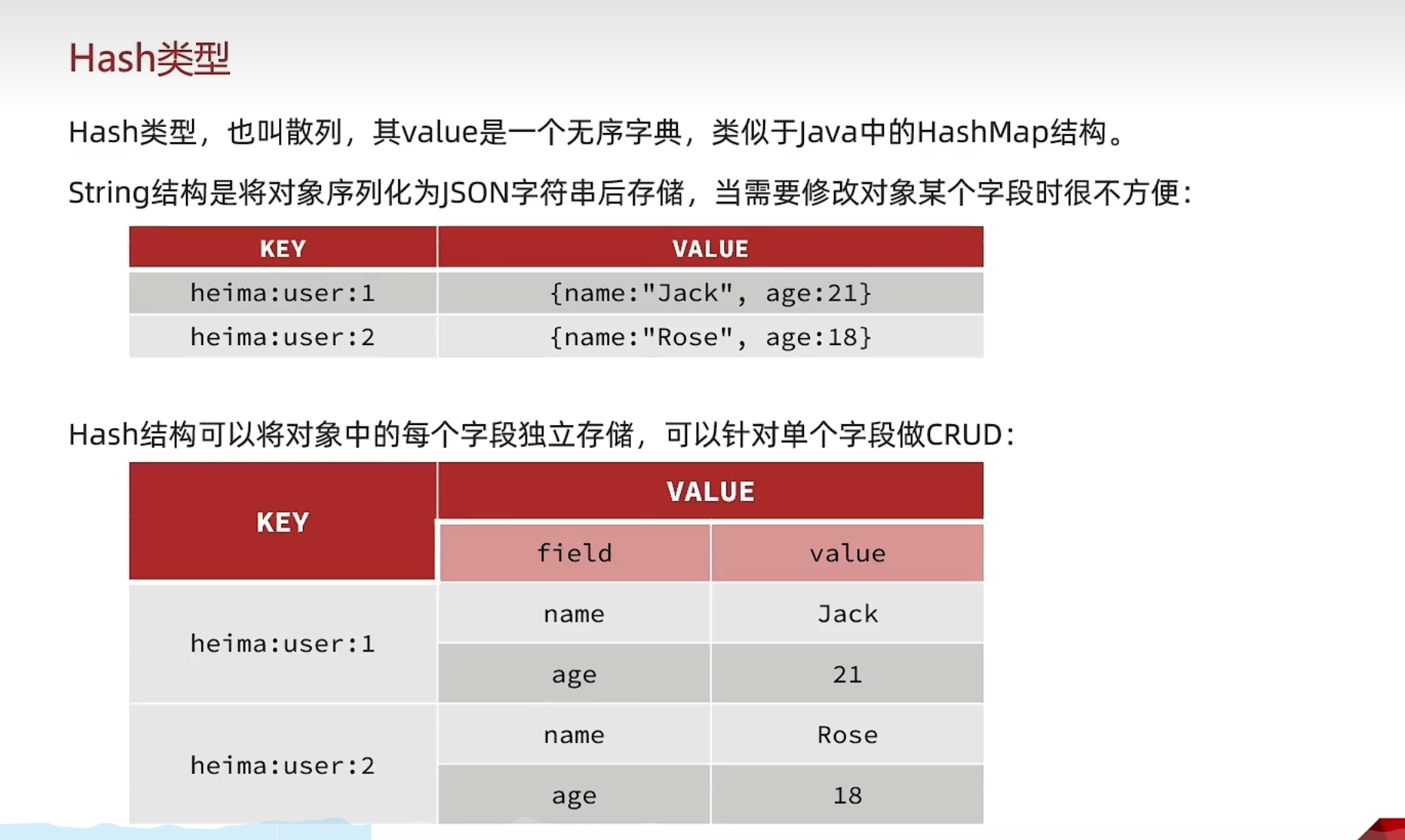
这里面的意思是，redis声称是“字符串”类型（redis里面的字符串），但是编码方式不同，整数和浮点数底层是整数和浮点数的编码方式

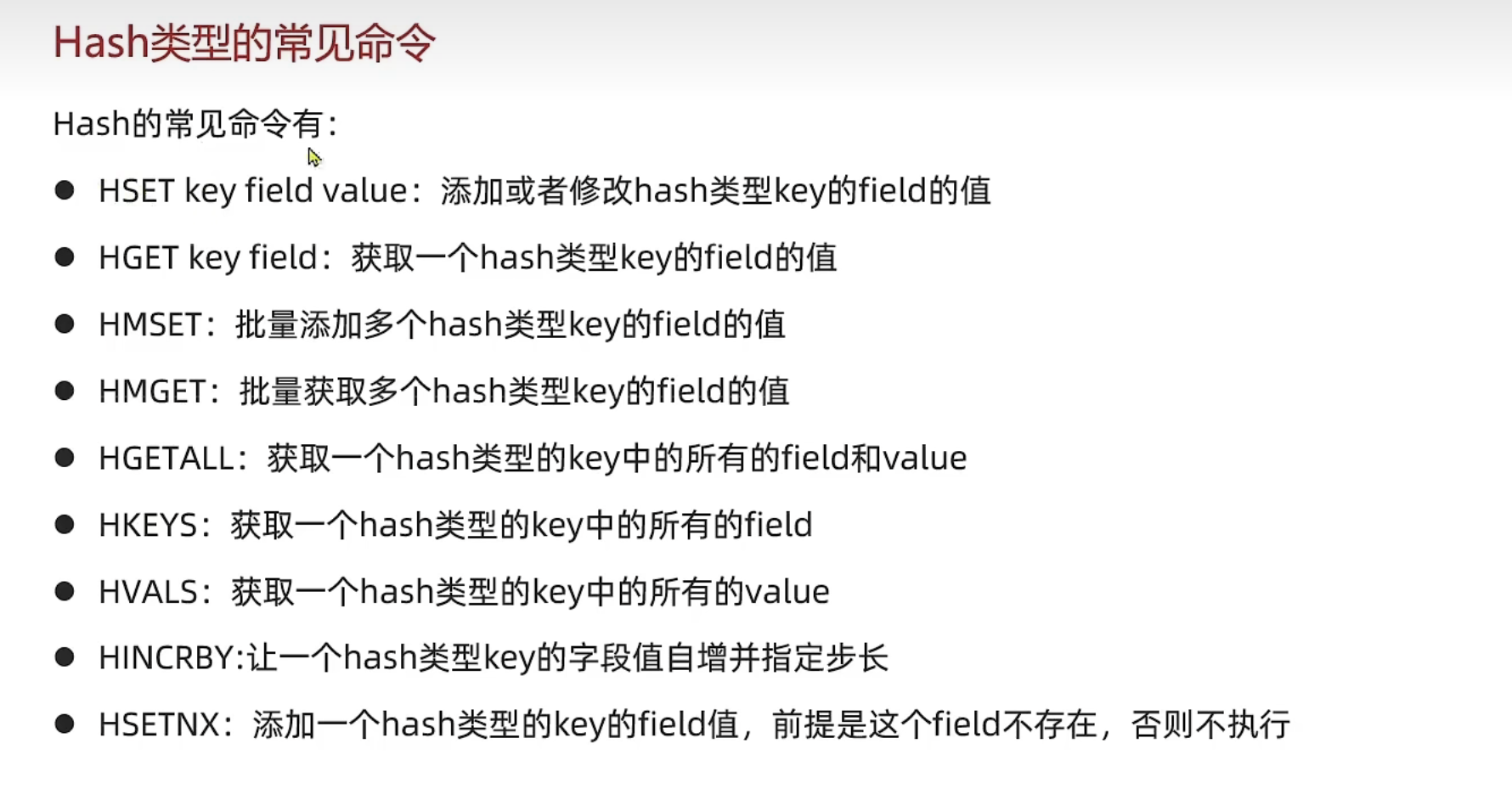


## hash（哈希）

底层实现是哈希表，实现原理等同于C++中的unorder\_map

用法与常见命令如下



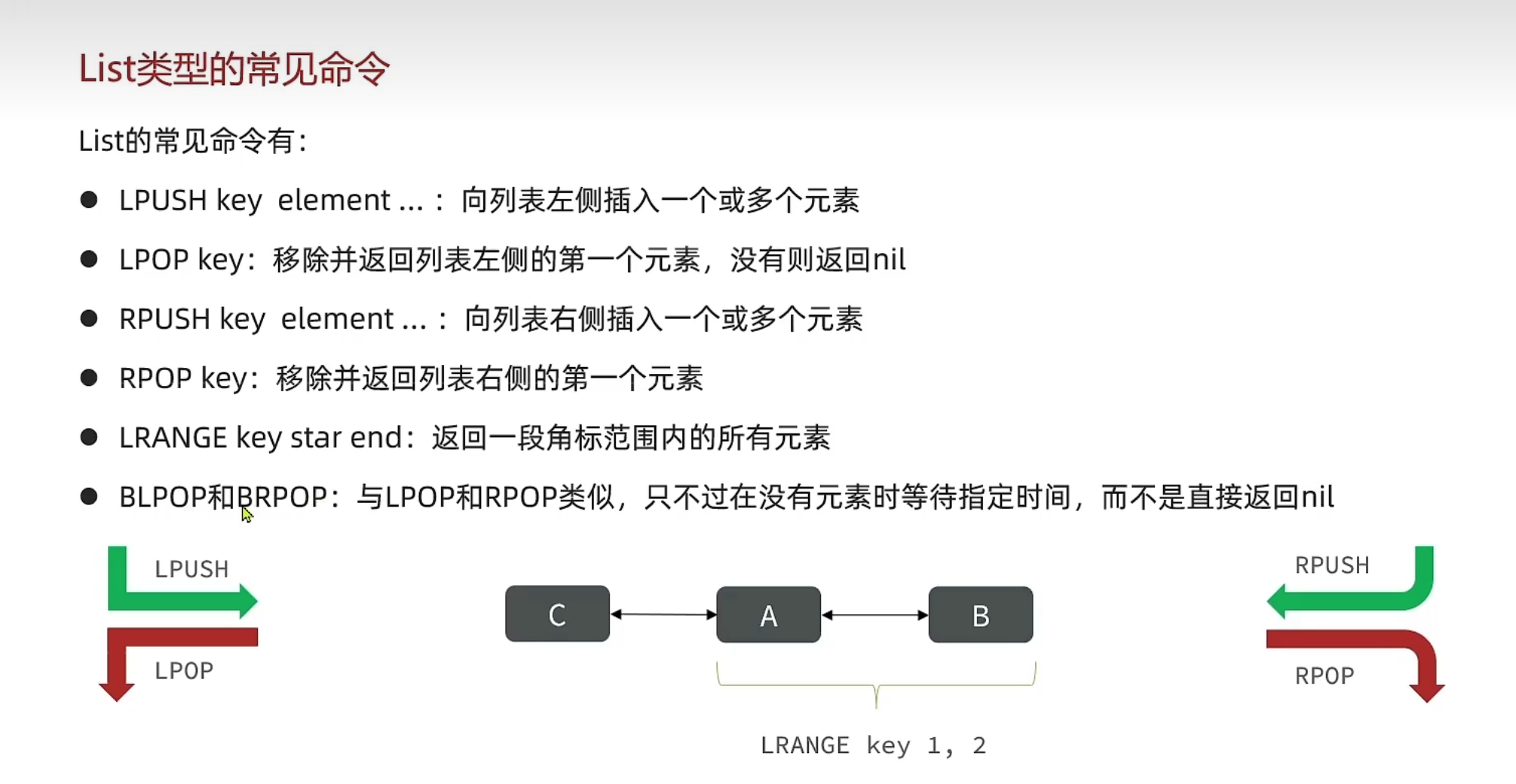


## list（列表）

底层实现是双向链表，实现原理等同于C++中的list

用法如下：





这里的key就是双向链表的key，是redis里面的字符串

而element就是要往双向链表中插入的元素

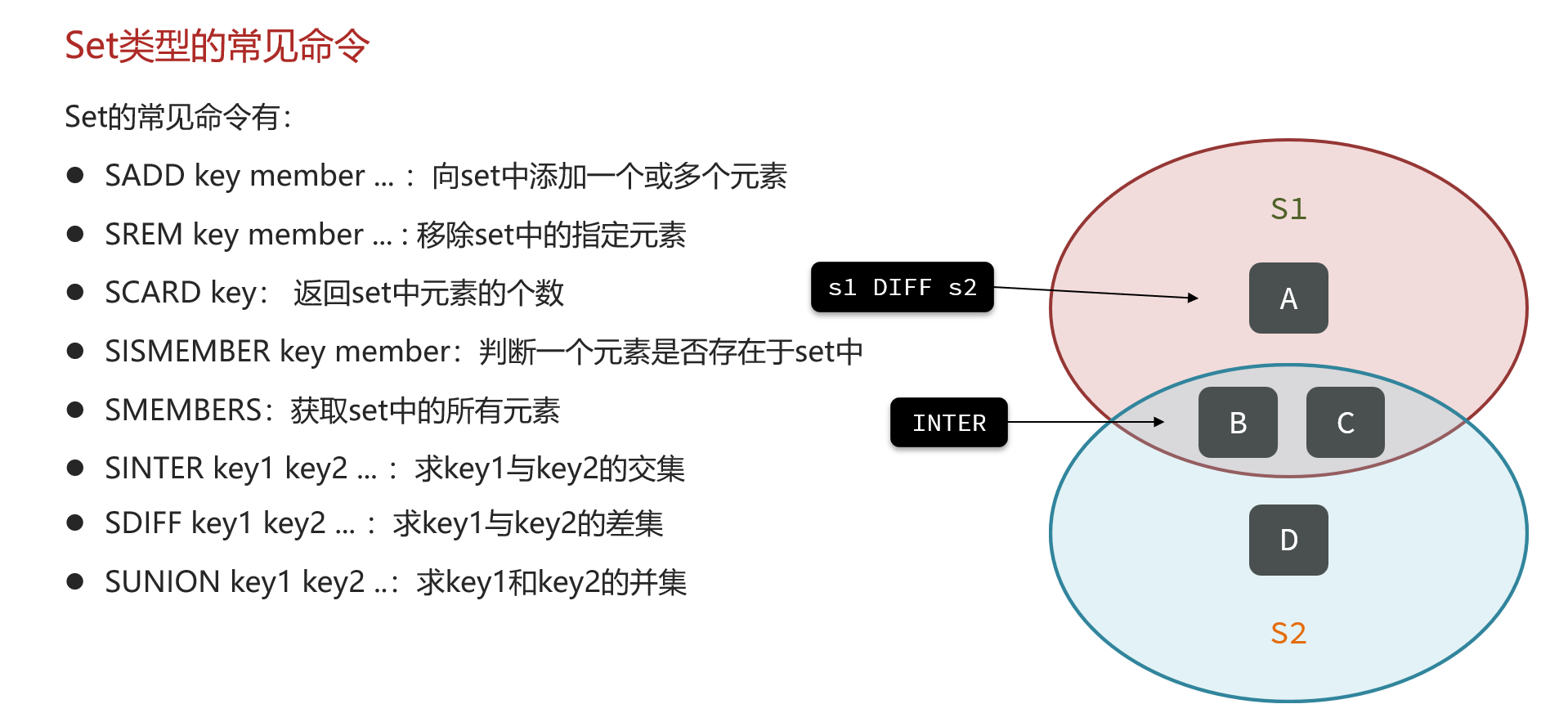
Start 和 end是数字（下标，从0开始）

## set（集合）

底层实现是哈希集合，原理等同于C++中的unordered\_set

用法如下

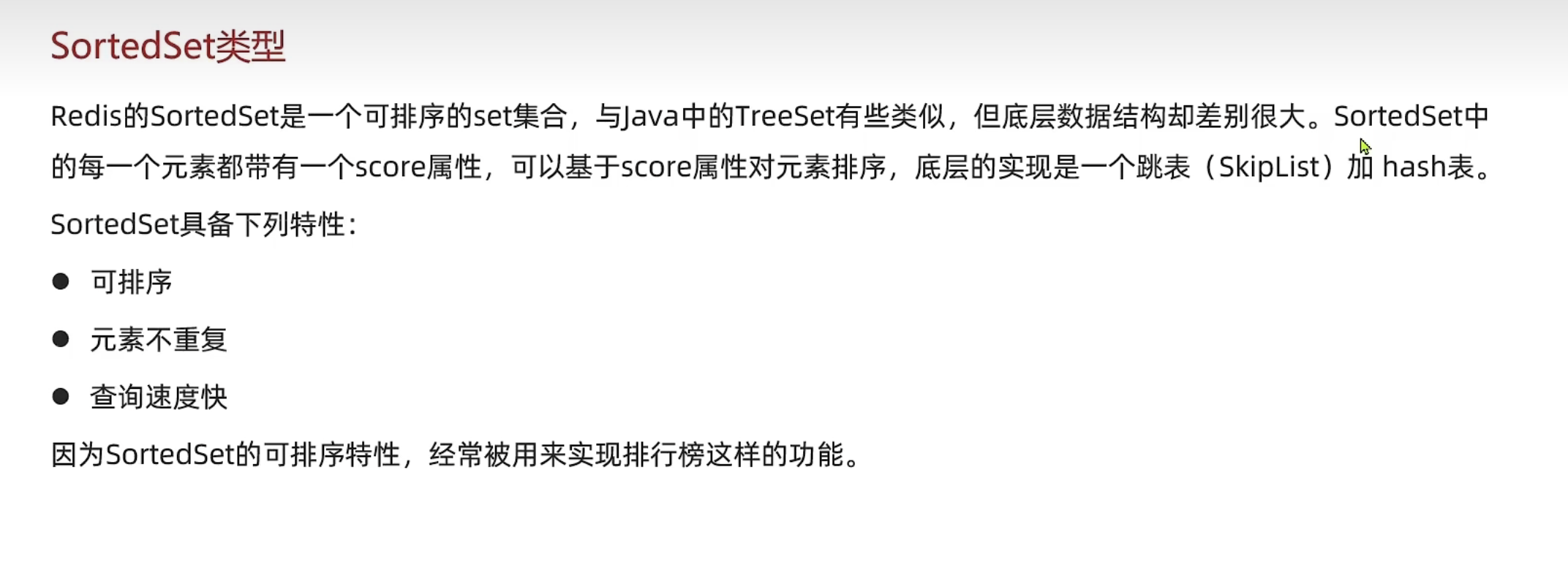




## sorted set（有序集合）

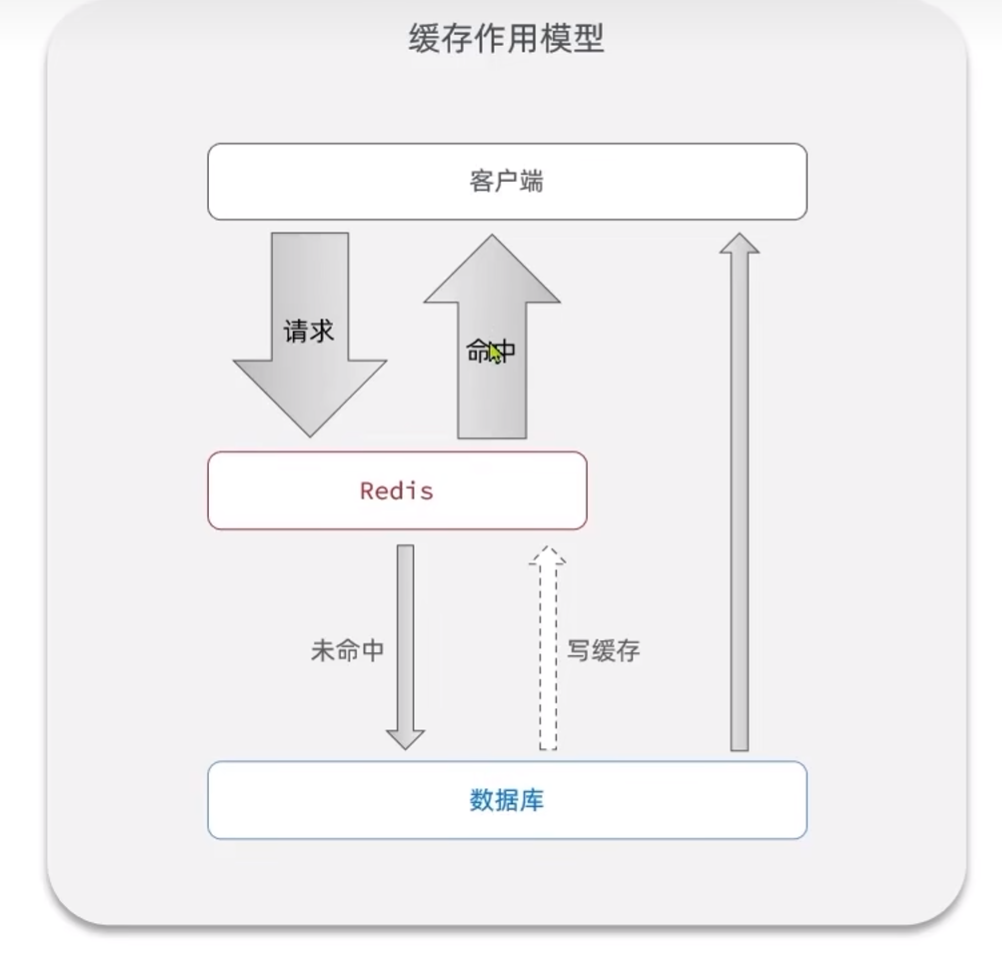
底层实现是跳表加上哈希表。

用法如下：





## redis缓存模型



使用缓存的好处：

1.降低服务器负载压力

2.提高响应时间

使用缓存的代价：

1.数据需要维持一致性

2.代码维护成本

2.为保证复用性，需要建立集群以及设置多级缓存，运维成本。

## 缓存的更新策略

保证数据的一致性与降低维护成本是相互矛盾的，数据一致性要求越高，维护成本越高。

通常来说有三种缓存更新策略：

### 内存淘汰

redis自身有一个内存淘汰机制，主要是为了应对缓存空间不足的问题，当内存不足时，自动淘汰数据，下次查询的时候更新缓存。

这种方法没有维护成本，但是数据一致性也最低

### 超时剔除

给缓存中的每条数据设置一个合适的TTL，时间一到自动被剔除。

这种方法维护成本较低，数据一致性适中

### 主动更新

编写业务逻辑，数据库更新的时候，删除或者更新（优先选择删除）缓存中的相应数据

这种方法维护成本高，数据一致性较好。

#### 主动更新过程中需要保障原子性、一致性和持久性。

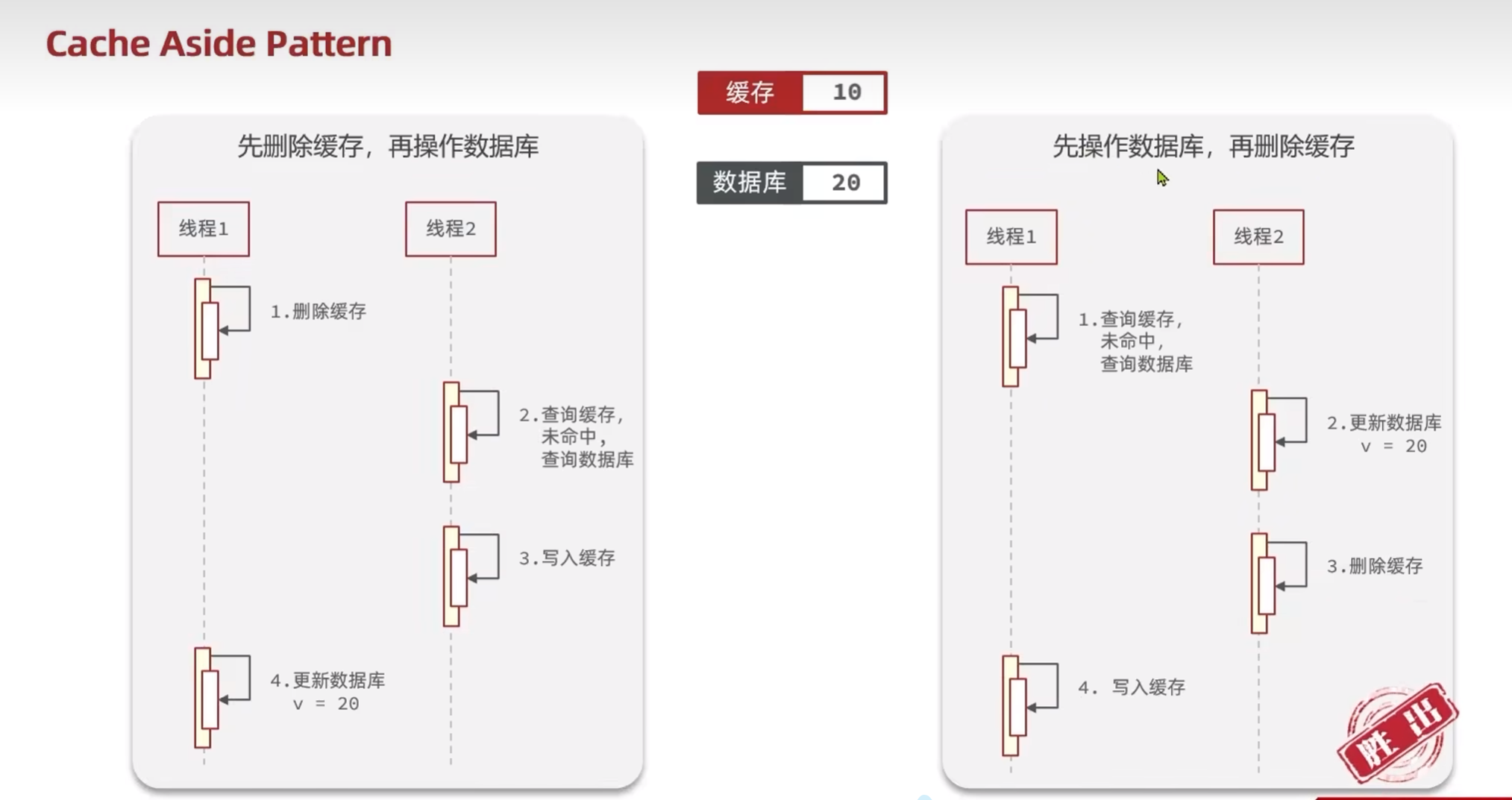
缓存与数据库中的数据更新必须是原子性的，要么同时成功要么同时失败。

因此，应该把这两个操作写在同一个事务之中。

#### 主动更新的线程安全问题：

在实际的过程中，可能有多个线程并发的访问缓存与数据库，在主动更新的过程中，可能产生线程安全问题，导致数据的不一致。

具体过程如下图所示：



#### 保证数据一致性的原理：

这里有两个线程，更新线程和查询线程。

查询线程先查询缓存中的数据，如果没有，再查询数据库，并将数据库里的数据写入缓存

更新线程有两种选择：

1.先更新缓存，再更新数据库。如果这样做，出现线程安全的概率很大。

更新线程先删除缓存，然后查询线程上cpu执行，查询缓存数据一定查不到，因为正好被更新线程删除了，于是去查数据库，并将数据库中的数据写入缓存，这时候更新线程继续执行，更改数据库中的数据。这就出现了数据不一致。

2.而如果先更新数据库，再更新缓存，出现数据不一致性的可能性会小得多。

因为要满足几个条件：

1.查询线程在查询时，缓存中没有对应的数据。

2.查询线程查询时必须在更新线程更新数据库之前，而查询线程更新缓存必须在更新线程删除缓存之后。换言之，查询线程更新缓存这一个操作跨越了更新线程的两个操作。但是更新缓存是一个非常迅速的操作，不太可能跨越更新数据库和删除缓存两个阶段，因此这种情况发生的概率很低。

### 主动更新 + 超时剔除

综上所述，宜采用主动更新策略，将对缓存和数据库的更新写在一个事务中，为了最大可能得避免线程安全问题，采用先更新数据库再删除缓存的策略，并且**以超时剔除作为兜底**。

如果对数据一致性存在更高要求，可以采用锁，来避免线程安全问题,但是会降低并发程度。

## 缓存穿透

### 定义

用户所请求的数据在缓存与数据库中均不存在，不断的发起这样的请求，将给数据库带来巨大的压力

### 解决方法

1.设置布隆过滤器

2.缓存空值，也就是说：即使数据库中不存在该键，也将该键在redis中缓存起来，设置值为空。并且设置TTL。

3.按照隐藏的规律对键进行设定，不符合规则的直接pass

4.用户校验，比如在查询缓存之前，需要密码登录，剔除掉非法用户

### 布隆过滤器

布隆过滤器是一种复杂化的哈希表，只不过哈希表计算一次哈希值，布隆过滤器计算多次哈希值，以减少哈希冲突的可能性。因此布隆过滤器判断存在的某个key可能是由于哈希冲突，实际上在db中并不存在，但是判断不存在的key一定不在db中。 至于布隆过滤器的位置，应该把它放在client和redis之间，但是判断key是否存在用到的是sql里面的所有key值。

这样布局结构就出来了，数据流向为:

客户端->布隆过滤器->redis缓存->mysql数据库

## 缓存雪崩

### 定义

缓存中的数据在同一时间段大量失效，或者缓存服务器宕机，导致大量数据到达数据库，带来巨大压力

### 解决方法

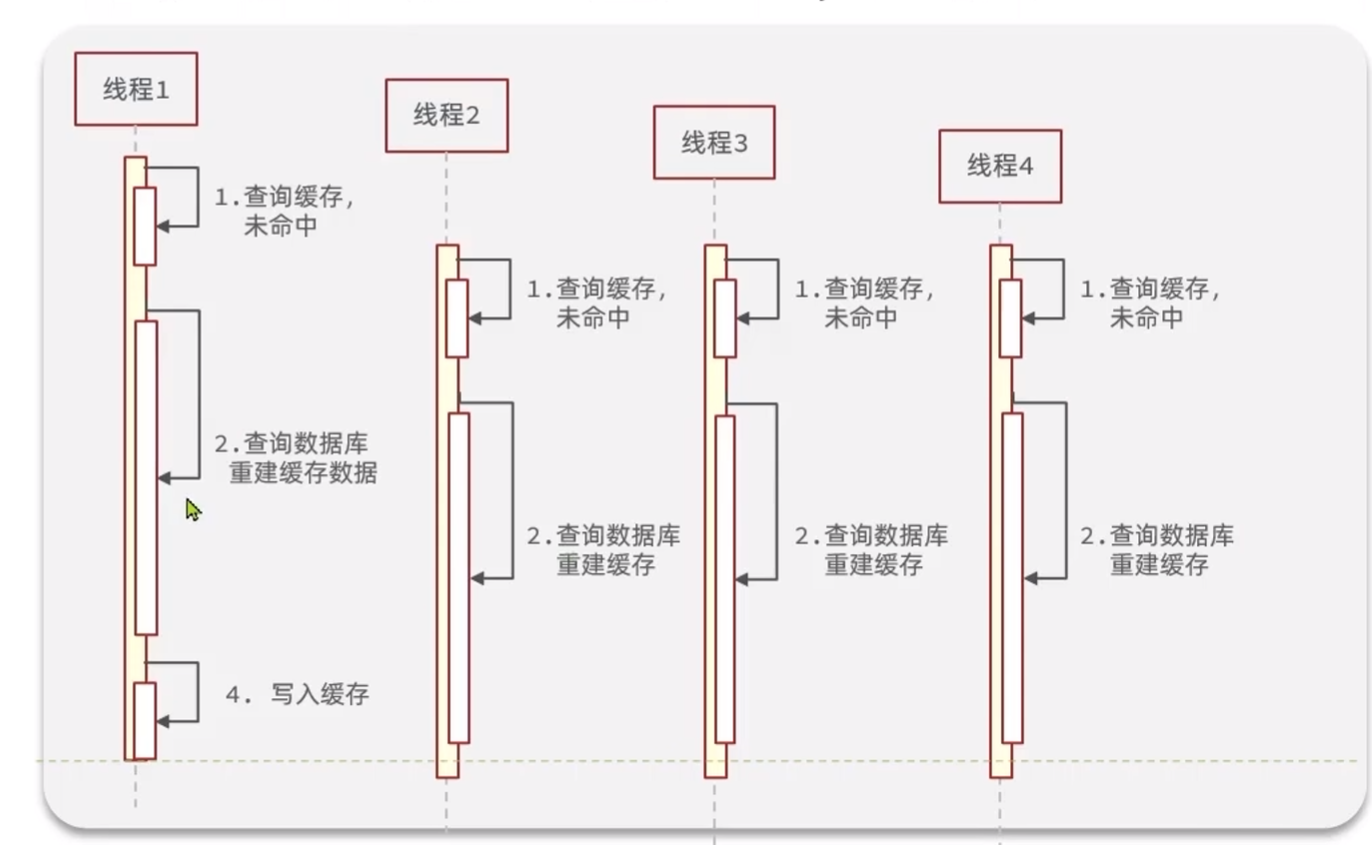
1.在一定范围内，随机的设置TTL

2.设置redis集群与多级缓存来规避缓存服务宕机的风险

## 缓存击穿

### 定义

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，无数的访问请求在瞬间给数据库带来巨大的冲击。



### 解决方法

1.添加互斥锁

查询缓存，有该数据则直接返回，没有则执行下一步;

尝试获取锁；

获取失败，则等待，直到获取锁成功；

查询缓存，若有则返回，没有则访问数据库，将数据库中的数据同步到内存之中。

2.互斥锁+逻辑过期

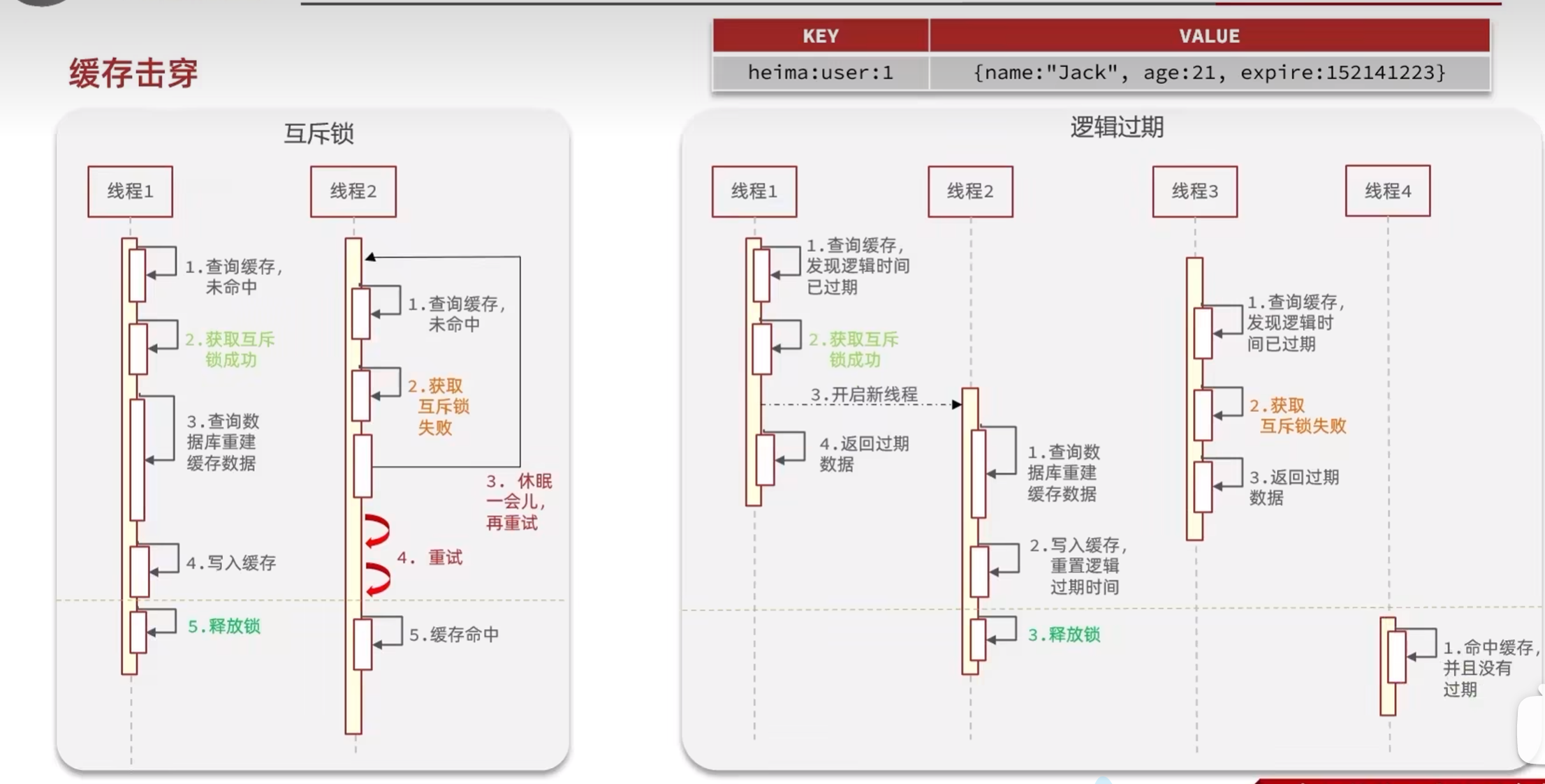
查询缓存，没有逻辑过期则直接返回，逻辑过期则继续执行下一步;

尝试获取锁；

获取失败，则返回逻辑过期的数据，获取成功则继续执行下一步；

访问数据库，将数据库中的数据同步到内存之中。

返回该数据



# 乐观锁与悲观锁

## 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，确保线程同步问题

## 乐观锁

认为线程安全问题不一定会发生，因此不加锁，只在更新数据的时候去判断有没有其它线程对数据做了修改，具体如何判断？有版本号和CAS两种方法

当然默认查询和修改这两个操作本身是原子性的，只是因为它们的顺序不同，才导致了线程安全问题。

比如以下的顺序

线程A查询数据x，得到它的值，发现满足修改条件

线程B查询数据x，得到它的值，发现满足修改条件

线程A修改数据x的值（修改之后，已经不满足线程B的修改条件了）

线程B修改数据x的值

这个时候，线程B的修改覆盖了线程A的修改，但是实际上，线程B的修改应该是非法的。

### 版本号

给数据加上一个版本号字段，每次更新的查询版本有没有发生变化，没有才会继续更新。

### CAS(Compare and Swap)

每次更新的时候，查看数据本身有没有发生变化，没有才会继续更新。

# redis持久化方式

持久化就是把redis内存中的数据保存到磁盘之中，这个过程叫做持久化

## RDB(Redis Database Backup file)

记录下redis所有数据的二进制文件，每隔一段时间写入磁盘（一般是开启一个后台）

## AOF(Append-Only File)

记录下每一条redis的命令操作，类似于逻辑日志

默认情况下，写命令执行完写放入内存缓冲区，每隔1秒，将其追加至磁盘中的AOF文件。

