简介

Redis是一个高性能的key-value数据库。Redis对数据的操作都是原子性的。

优缺点

优点：

1. 基于内存操作，内存读写速度快。
2. Redis是单线程的，避免线程切换开销及多线程的竞争问题。单线程是指在处理网络请求（一个或多个redis客户端连接）的时候只有一个线程来处理，redis运行时不止有一个线程，数据持久化或者向slave同步aof时会另起线程。
3. 支持多种数据类型，包括String、Hash、List、Set、ZSet等
4. 支持持久化。Redis支持RDB和AOF两种持久化机制，持久化功能有效地避免数据丢失问题。
5. redis 采用IO多路复用技术。多路指的是多个socket连接，复用指的是复用一个线程。redis使用单线程来轮询描述符，将数据库的开、关、读、写都转换成了事件。多路复用主要有三种技术：select，poll，epoll。epoll是最新的也是目前最好的多路复用技术。

缺点：对join或其他结构化查询的支持就比较差。

io多路复用

将用户socket对应的文件描述符（file description）注册进epoll，然后epoll帮你监听哪些socket上有消息到达。当某个socket可读或者可写的时候，它可以给你一个通知。只有当系统通知哪个描述符可读了，才去执行read操作，可以保证每次read都能读到有效数据。这样，多个描述符的I/O操作都能在一个线程内并发交替地顺序完成，这就叫I/O多路复用，这里的复用指的是复用同一个线程。

应用场景

1. 缓存热点数据，缓解数据库的压力。
2. 利用Redis中原子性的自增操作，可以用使用实现计算器的功能，比如统计用户点赞数、用户访问数等，这类操作如果用MySQL，频繁的读写会带来相当大的压力。
3. 简单消息队列，不要求高可靠的情况下，可以使用Redis自身的发布/订阅模式或者List来实现一个队列，实现异步操作。
4. 好友关系，利用集合的一些命令，比如求交集、并集、差集等。可以方便搞定一些共同好友、共同爱好之类的功能。
5. 限速器，比较典型的使用场景是限制某个用户访问某个API的频率，常用的有抢购时，防止用户疯狂点击带来不必要的压力。

Memcached和Redis的区别

1. Redis只使用单核，而Memcached可以使用多核。
2. MemCached数据结构单一，仅用来缓存数据，而Redis支持更加丰富的数据类型，也可以在服务器端直接对数据进行丰富的操作，这样可以减少网络IO次数和数据体积。
3. MemCached不支持数据持久化，断电或重启后数据消失。Redis支持数据持久化和数据恢复，允许单点故障。

数据类型

Redis支持五种数据类型：

- string（字符串）
- hash（哈希）
- list（列表）
- set（集合）
- zset(sorted set)

字符串类型

字符串类型的值可以是字符串、数字或者二进制，但值最大不能超过512MB。

常用命令：set, get, incr, incrby, desr, keys, append, strlen

- 赋值和取值

```
SET name tyson
GET name
```

- 递增数字

```
INCR num        //若键值不是整数时，则会提示错误。
INCRBY num 2    //增加指定整数
DESR num        //递减数字
INCRBY num 2.7 //增加指定浮点数
```

- 其他

`keys list*` 列出匹配的key

`APPEND name " dai"` 追加值

`STRLEN name` 获取字符串长度

`MSET name tyson gender male` 同时设置多个值

`MGET name gender` 同时获取多个值

`GETBIT name 0` 获取0索引处二进制位的值

`FLUSHDB` 删除当前数据库所有的key

`FLUSHALL` 删除所有数据库中的key

SETNX和SETEX

`SETNX key value`：当key不存在时，将key的值设为value。若给定的key已经存在，则SETNX不做任何操作。

`SETEX key seconds value`：比SET多了seconds参数，相当于`SET KEY value` + `EXPIRE KEY seconds`，而且SETEX是原子性操作。

keys和scan

redis的单线程的。keys指令会导致线程阻塞一段时间，直到执行完毕，服务才能恢复。scan采用渐进式遍历的方式来解决keys命令可能带来的阻塞问题，每次scan命令的时间复杂度是O（1），但是要真正实现keys的功能，需要执行多次scan。

scan的缺点：在scan的过程中如果有键的变化（增加、删除、修改），遍历过程可能会有以下问题：新增的键可能没有遍历到，遍历出了重复的键等情况，也就是说scan并不能保证完整的遍历出来所有的键。

scan命令用于迭代当前数据库中的数据库键：`SCAN cursor [MATCH pattern] [COUNT count]`

```
scan 0 match * count 10 //返回10个元素
```

SCAN相关命令包括SSCAN 命令、HSCAN 命令和 ZSCAN 命令，分别用于集合、哈希键及有序集合。

expire

```
SET password 666
EXPIRE password 5
TTL password //查看键的剩余生存时间，-1为永不过期
SETEX password 60 123abc //SETEX可以在设置键的同时设置它的生存时间
```

EXPIRE时间单位是秒，PEXPIRE时间单位是毫秒。在键未过期前可以重新设置过期时间，过期之后则键被销毁。

在Redis 2.6和之前版本，如果key不存在或者已过期时返回`-1`。

从Redis2.8开始，错误返回值的结果有如下改变：

- 如果key不存在或者已过期，返回 `-2`
- 如果key存在并且没有设置过期时间（永久有效），返回 `-1` 。

type

TYPE 命令用于返回 key 所储存的值的类型。

```
127.0.0.1:6379> type NEWBLOG
list
```

散列类型

常用命令：hset, hget, hmset, hmget, hgetall, hdel, hkeys, hvals

- 赋值和取值

```
HSET car price 500 //HSET key field value
HGET car price
```

同时设置获取多个字段的值

```
HMSET car price 500 name BMW
HMGET car price name
HGETALL car
```

使用 HGETALL 命令时，如果哈希元素个数比较多，会存在阻塞Redis的可能。如果只需要获取部分field，可以使用hmget，如果一定要获取全部field-value，可以使用hscan命令，该命令会渐进式遍历哈希类型。

`HSETNX car price 400 //当字段不存在时赋值，HSETNX是原子操作，不存在竞态条件`

- 增加数字
  `HINCRBY person score 60`
- 删除字段
  `HDEL car price`

- 其他

```
HKEYS car //获取key
HVALS car //获取value
HLEN car  //长度
```

列表类型

常用命令：lpush, rpush, lpop, rpop, lrange, lrem

添加和删除元素

```
LPUSH numbers 1
RPUSH numbers 2 3
LPOP numbers
RPOP numbers
```

获取列表片段

```
LRANGE numbers 0 2
LRANGE numbers -2 -1 //支持负索引 -1是最右边第一个元素
LRANGE numbers 0 -1
```

向列表插入值

首先从左到右寻找值为pivot的值，向列表插入value

```
LINSERT numbers AFTER 5 8 //往5后面插入8
LINSERT numbers BEFORE 6 9 //往6前面插入9
```

删除元素

`LTRIM numbers 1 2` 删除索引1到2以外的所有元素

LPUSH常和LTRIM一起使用来限制列表的元素个数，如保留最近的100条日志

```
LPUSH logs $newLog
LTRIM logs 0 99
```

删除列表指定的值

`LREM key count value`

```
1. count < 0, 则从右边开始删除前count个值为value的元素
2. count > 0, 则从左边开始删除前count个值为value的元素
3. count = 0, 则删除所有值为value的元素 `LREM numbers 0 2`
```

其他

```
LLEN numbers       //获取列表元素个数
LINDEX numbers -1  //返回指定索引的元素，index是负数则从右边开始计算
LSET numbers 1 7   //把索引为1的元素的值赋值成7
```

集合类型

常用命令：sadd, srem, smembers, scard, sismember, sdiff

集合中不能有相同的元素。

增加/删除元素

```
SADD letters a b c
SREM letters c d
```

获取元素

```
SMEMBERS letters
SCARD letters   //获取集合元素个数
```

判断元素是否在集合中

`SISMEMBER letters a`

集合间的运算

```
SDIFF setA setB  //差集运算
SINTER setA setB //交集运算
SUNION setA setB //并集运算
```

三个命令都可以传进多个键 `SDIFF setA setB setC`

其他

`SDIFFSTORE result setA setB` 进行集合运算并将结果存储

`SRANDMEMBER key count`

> 随机获取集合里的一个元素，count大于0，则从集合随机获取count个不重复的元素，count小于0，则随机获取的count个元素有些可能相同。

`SPOP letters`

有序集合类型

常用命令：zadd, zrem, zscore, zrange

```
zadd zsetkey 50 e1 60 e2 30 e3
```

Zset(sorted set)是string类型的有序集合。zset 和 set 一样也是string类型元素的集合，且不允许重复的成员。不同的是Zset每个元素都会关联一个double（超过17位使用科学计算法表示，可能丢失精度）类型的分数，通过分数来为集合中的成员进行排序。zset的成员是唯一的,但分数(score)可以重复。

有序集合和列表相同点：

1. 都是有序的；
2. 都可以获得某个范围内的元素。

有序集合和列表不同点：

1. 列表基于链表实现，获取两端元素速度快，访问中间元素速度慢；
2. 有序集合基于散列表和跳跃表实现，访问中间元素时间复杂度是OlogN；

3. 列表不能简单的调整某个元素的位置，有序列表可以（更改元素的分数）；
4. 有序集合更耗内存。

增加/删除元素

时间复杂度OlogN。

```
ZADD scoreboard 89 Tom 78 Sophia
ZADD scoreboard 85.5 Tyson      //支持双精度浮点数
ZREM scoreboard Tyson
ZREMRANGEBYRANK scoreboard 0 2  //按照排名范围删除元素
ZREMRANGEBYSCORE scoreboard (80 100 //按照分数范围删除元素，"("代表不包含
```

获取元素分数

时间复杂度O1。

`ZSCORE scoreboard Tyson`

获取排名在某个范围的元素列表

ZRANGE命令时间复杂度是O(log(n)+m)， n是有序集合元素个数，m是返回元素个数。

```
ZRANGE scoreboard 0 2
ZRANGE scoreboard 1 -1  //-1表示最后一个元素
ZRANGE scoreboard 0 -1 WITHSCORES  //同时获得分数
```

获取指定分数范围的元素

ZRANGEBYSCORE命令时间复杂度是O(log(n)+m)， n是有序集合元素个数，m是返回元素个数。

```
ZRANGEBYSCORE scoreboard 80 100
ZRANGEBYSCORE scoreboard 80 (100  //不包含100
ZRANGEBYSCORE scoreboard (60 +inf LIMIT 1 3 //获取分数高于60的从第二个人开始的3个人
```

增加某个元素的分数

时间复杂度OlogN。

`ZINCRBY scoreboard 10 Tyson`

其他

```
ZCARD scoreboard          //获取集合元素个数，时间复杂度O1
ZCOUNT scoreboard 80 100  //指定分数范围的元素个数
ZRANK scoreboard Tyson    //按从小到大的顺序获取元素排名
ZREVRANK scoreboard Tyson //按从大到小的顺序获取元素排名
```

Bitmaps

Bitmaps本身不是一种数据结构，实际上它就是字符串，但是它可以对字符串的位进行操作，可以把Bitmaps想象成一个以位为单位的数组，数组的每个单元只能存储0和1。

bitmap的长度与集合中元素个数无关，而是与基数的上限有关。假如要计算上限为1亿的基数，则需要12.5M字节的bitmap。就算集合中只有10个元素也需要12.5M。

HyperLogLog

HyperLogLog 是用来做基数统计的算法，其优点是，在输入元素的数量或者体积非常非常大时，计算基数所需的空间总是固定的、并且是很小的。

基数：比如数据集 {1, 3, 5, 7, 5, 7, 8}， 那么这个数据集的基数集为 {1, 3, 5 ,7, 8}，基数即不重复元素为 5。

应用场景：独立访客（unique visitor，uv）统计。

数据结构

动态字符串

SDS定义：

```
struct sdshdr {

    // 记录 buf 数组中已使用字节的数量
    // 等于 SDS 所保存字符串的长度
    int len;

    // 记录 buf 数组中未使用字节的数量
    int free;

    // 字节数组，用于保存字符串
    char buf[];

};
```

C 字符串	SDS
获取字符串长度的复杂度为 O(N) 。	获取字符串长度的复杂度为 O(1) 。
API 是不安全的，可能会造成缓冲区溢出。	API 是安全的，不会造成缓冲区溢出。
修改字符串长度 `N` 次必然需要执行 `N` 次内存重分配。	修改字符串长度 `N` 次最多需要执行 `N` 次内存重分配。
只能保存文本数据。	可以保存文本或者二进制数据。
可以使用所有 `<string.h>` 库中的函数。	可以使用一部分 `<string.h>` 库中的函数。

> 优点：
>
> - 获取字符串长度的时间复杂度为O（1）
> - 支持动态扩容
> - 减少内存分配次数
> - 二进制安全

字典

字典使用hashtable作为底层实现。键值对的值可以是一个指针， 或者是一个 uint64_t 整数， 又或者是一个 int64_t 整数。

```
typedef struct dictEntry {
```

```
    // 键
    void *key;

    // 值
    union {
        void *val;
        uint64_t u64;
        int64_t s64;
    } v;

    // 指向下个哈希表节点，形成链表
    struct dictEntry *next;

} dictEntry;
```

整数集合

整数集合（intset）是 Redis 用于保存整数值的集合抽象数据结构， 它可以保存类型为 int16_t 、 int32_t 或者 int64_t 的整数值， 并且保证集合中不会出现重复元素。

压缩列表

ziplist是 Redis 为了节约内存而开发的， 由一系列特殊编码的连续内存块组成的顺序型（sequential）数据结构。每个压缩列表节点都由 previous_entry_length 、 encoding 、 content 三个部分组成。

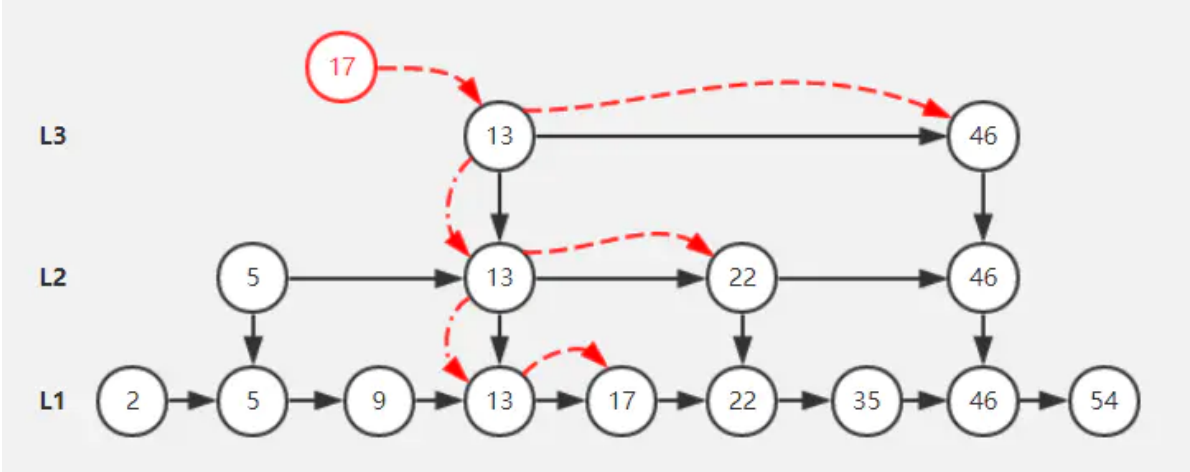
节点的 previous_entry_length 属性以字节为单位， 记录了压缩列表中前一个节点的长度。
节点的 encoding 属性记录了节点的 content 属性所保存数据的类型以及长度。有两种编码方式，字节数组编码和整数编码。

压缩列表的从表尾向表头遍历操作就是使用这一原理实现的： 只要我们拥有了一个指向某个节点起始地址的指针， 那么通过这个指针以及这个节点的 previous_entry_length 属性， 程序就可以一直向前一个节点回溯， 最终到达压缩列表的表头节点。

跳表

跳表可以看成多层链表，它有如下的性质：

- 多层的结构组成，每层是一个有序的链表
- 最底层的链表包含所有的元素
- 跳跃表的查找次数近似于层数，时间复杂度为O(logn)，插入、删除也为 O(logn)



对象

Redis 的对象系统还实现了基于引用计数技术的内存回收机制： 当程序不再使用某个对象的时候， 这个对象所占用的内存就会被自动释放； 另外， Redis 还通过引用计数技术实现了对象共享机制， 这一机制可以在适当的条件下， 通过让多个数据库键共享同一个对象来节约内存。

底层实现

string

字符串对象的编码可以是 int 、 raw 或者 embstr 。

1. 如果一个字符串对象保存的是整数值， 并且这个整数值可以用 long 类型来表示， 那么会将编码设置为 int 。
2. 如果字符串对象保存的是一个字符串值， 并且这个字符串值的长度大于 39 字节， 那么字符串对象将使用一个简单动态字符串（SDS）来保存这个字符串值， 并将对象的编码设置为 raw 。
3. 如果字符串对象保存的是一个字符串值， 并且这个字符串值的长度小于等于 39 字节， 那么字符串对象将使用 embstr 编码的方式来保存这个字符串值。

值	编码
可以用 `long` 类型保存的整数。	`int`
可以用 `long double` 类型保存的浮点数。	`embstr` 或者 `raw`
字符串值， 或者因为长度太大而没办法用 `long` 类型表示的整数， 又或者因为长度太大而没办法用 `long double` 类型表示的浮点数。	`embstr` 或者 `raw`

hash

hash类型内部编码有两种：

1. ziplist，压缩列表。当哈希类型元素个数小于512个，并且所有值都小于64字节时，Redis会使用ziplist作为哈希的内部实现。ziplist使用更加紧凑的结构实现多个元素的连续存储，更加节省内存。
2. hashtable。当哈希类型无法满足ziplist的条件时，Redis会使用hashtable作为哈希的内部实现，因为此时ziplist的读写效率会下降，而hashtable的读写时间复杂度为O（1）。

使用 ziplist 作为 hash 的底层实现时，添加元素的时候，同一键值对的两个节点总是紧挨在一起， 保存键的节点在前， 保存值的节点在后。

使用场景：记录博客点赞数量。`hset MAP_BLOG_LIKE_COUNT blogId likeCount`，key为MAP_BLOG_LIKE_COUNT，field为博客id，value为点赞数量。

list

列表list类型内部编码有两种：

1. ziplist，压缩列表。当列表中的元素个数小于512个，同时列表中每个元素的值都小于64字节时，Redis会选用ziplist来作为列表的内部实现来减少内存的使用。
2. 当列表类型无法满足ziplist的条件时，Redis会使用linkedlist作为列表的内部实现。

Redis3.2版本提供了quicklist内部编码，简单地说它是以一个ziplist为节点的linkedlist，它结合了ziplist和linkedlist两者的优势，为列表类型提供了一种更为优秀的内部编码实现。

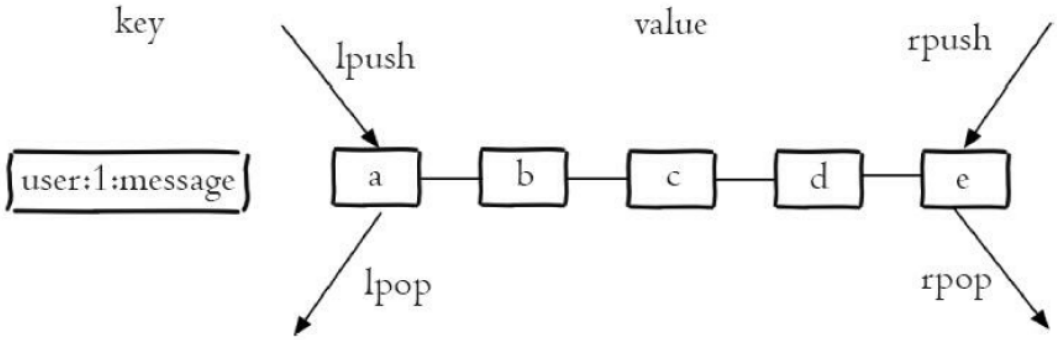


图2-18　列表两端插入和弹出操作

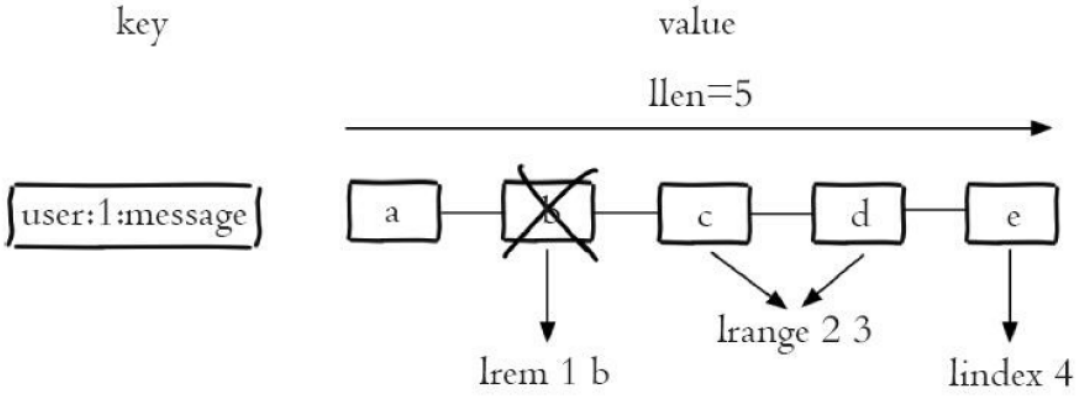


图2-19　子列表获取、删除等操作

使用场景：

1. 消息队列。Redis的lpush+brpop命令组合即可实现阻塞队列。

set

集合对象的编码可以是 intset 或者 hashtable 。

1. intset 编码的集合对象使用整数集合作为底层实现， 集合对象包含的所有元素都被保存在整数集合（数组）里面。
2. hashtable 编码的集合对象使用字典作为底层实现， 字典的每个键都是一个字符串对象， 而字典的值则全部被设置为 NULL 。

zset

有序集合的编码可以是 ziplist 或者 skiplist 。当有序集合的元素个数小于128，同时每个元素的值都小于64字节时，Redis会用ziplist来作为有序集合的内部实现，ziplist可以有效减少内存的使用。否则，使用skiplist作为有序集合的内部实现。

1. ziplist 编码的有序集合对象使用压缩列表作为底层实现， 每个集合元素使用两个紧挨在一起的压缩列表节点来保存， 第一个节点保存元素的成员（member）， 而第二个元素则保存元素的分值（score）。压缩列表内的集合元素按分值从小到大进行排序。
2. skiplist 编码的有序集合对象使用字典和跳跃表实现。使用字典查找给定成员的分值，时间复杂度为O(1)（跳跃表查找时间复杂度为O(logN)）。使用跳跃表可以对有序集合进行范围型操作。

使用场景

string：1、常规key-value缓存应用。常规计数如微博数、粉丝数。2、分布式锁。

hash：存放结构化数据，如用户信息（昵称、年龄、性别、积分等）。

list：热门博客列表、消息队列系统。使用list可以构建队列系统，比如：将Redis用作日志收集器，多个端点将日志信息写入Redis，然后一个worker统一将所有日志写到磁盘。

set：1、好友关系，微博粉丝的共同关注、共同喜好、共同好友等；2、利用唯一性，统计访问网站的所有独立ip 。

zset：1、排行榜；2、优先级队列。

数据库管理

切换数据库：`select 1`。Redis默认配置中是有16个数据库。0号数据库和15号数据库之间的数据没有任何关联，可以存在相同的键。不建议使用Redis多数据库功能，可以在一台机器上部署多个Redis实例，使用端口号区分，实现多数据库功能。

flushdb/flushall命令用于清除数据库，两者的区别的是flushdb只清除当前数据库，flushall会清除所有数据库。如果当前数据库键值数量比较多，flushdb/flushall存在阻塞Redis的可能性，并且这两个命令会将所有数据清除，一旦误操作后果不堪设想。

排序

```
LPUSH myList 4 8 2 3 6
SORT myList DESC
```

```
LPUSH letters f l d n c
SORT letters ALPHA
```

BY参数

```
LPUSH list1 1 2 3
SET score:1 50
SET score:2 100
SET score:3 10
SORT list1 BY score:* DESC
```

GET参数

GET参数命令作用是使SORT命令的返回结果是GET参数指定的键值。

`SORT tag:Java:posts BY post:*->time DESC GET post:*->title GET post:*->time GET #`

GET #返回文章ID。

STORE参数

`SORT tag:Java:posts BY post:*->time DESC GET post:*->title STORE resultCache`

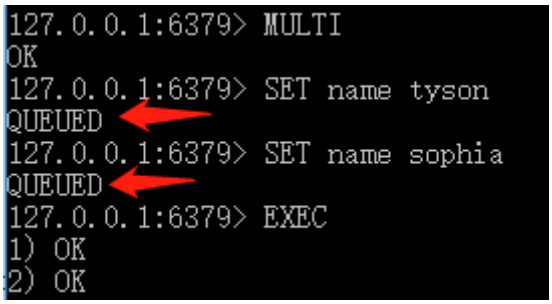
`EXPIRE resultCache 10 //STORE结合EXPIRE可以缓存排序结果`

事务

事务的原理是将一个事务范围内的若干命令发送给Redis，然后再让Redis依次执行这些命令。

事务的生命周期：

1. 使用MULTI开启一个事务
2. 在开启事务的时候，每次操作的命令将会被插入到一个队列中，同时这个命令并不会被真的执行
3. EXEC命令进行提交事务

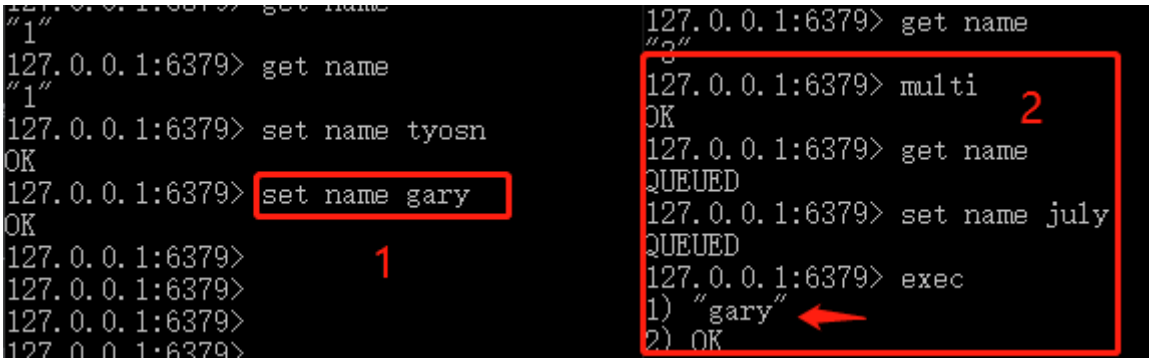


DISCARD：放弃事务，即该事务内的所有命令都将取消

一个事务范围内某个命令出错不会影响其他命令的执行，不保证原子性：

```
127.0.0.1:6379> multi
OK
127.0.0.1:6379> set a 1
QUEUED
127.0.0.1:6379> set b 1 2
QUEUED
127.0.0.1:6379> set c 3
QUEUED
127.0.0.1:6379> exec
1) OK
2) (error) ERR syntax error
3) OK
```

事务里的命令执行时会读取最新的值：



WATCH命令

WATCH命令可以监控一个或多个键，一旦其中有一个键被修改，之后的事务就不会执行（类似于乐观锁）。执行EXEC命令之后，就会自动取消监控。

```
127.0.0.1:6379> watch name
OK
127.0.0.1:6379> set name 1
OK
127.0.0.1:6379> multi
OK
127.0.0.1:6379> set name 2
QUEUED
```

```
127.0.0.1:6379> set gender 1
QUEUED
127.0.0.1:6379> exec
(nil)
127.0.0.1:6379> get gender
(nil)
```

UNWATCH：取消WATCH命令对多有key的监控，所有监控锁将会被取消。

消息队列

使用一个列表，让生产者将任务使用LPUSH命令放进列表，消费者不断用RPOP从列表取出任务。

BRPOP和RPOP命令相似，唯一的区别就是当列表没有元素时BRPOP命令会一直阻塞连接，直到有新元素加入。
`BRPOP queue 0  //0表示不限制等待时间`

优先级队列

`BLPOP queue:1 queue:2 queue:3 0`
如果多个键都有元素，则按照从左到右的顺序取元素

发布/订阅模式

```
PUBLISH channel1 hi
SUBSCRIBE channel1
UNSUBSCRIBE channel1 //退订通过SUBSCRIBE命令订阅的频道。
```

`PSUBSCRIBE channel?*` 按照规则订阅
`PUNSUBSCRIBE channel?*` 退订通过PSUBSCRIBE命令按照某种规则订阅的频道。其中订阅规则要进行严格的字符串匹配，`PUNSUBSCRIBE *`无法退订`channel?*`规则。

缺点：在消费者下线的情况下，生产的消息会丢失。

延时队列

使用sortedset，拿时间戳作为score，消息内容作为key，调用zadd来生产消息，消费者用`zrangebyscore`指令获取N秒之前的数据轮询进行处理。

持久化

Redis支持两种方式的持久化，一种是RDB的方式，一种是AOF的方式。前者会根据指定的规则定时将内存中的数据存储在硬盘上，而后者在每次执行完命令后将命令记录下来。一般将两者结合使用。

RDB方式

RDB 是 Redis 默认的持久化方案。RDB持久化时会将内存中的数据写入到磁盘中，在指定目录下生成一个dump.rdb文件。Redis 重启会加载dump.rdb文件恢复数据。

RDB持久化的过程（执行SAVE命令除外）：

- 创建一个子进程；
- 父进程继续接收并处理客户端的请求，而子进程开始将内存中的数据写进硬盘的临时文件；

- 当子进程写完所有数据后会用该临时文件替换旧的RDB文件。

Redis启动时会读取RDB快照文件，将数据从硬盘载入内存。通过RDB方式的持久化，一旦Redis异常退出，就会丢失最近一次持久化以后更改的数据。

触发RDB快照：

1. 手动触发：
    - 用户执行SAVE或BGSAVE命令。SAVE命令执行快照的过程会阻塞所有来自客户端的请求，应避免在生产环境使用这个命令。BGSAVE命令可以在后台异步进行快照操作，快照的同时服务器还可以继续响应客户端的请求，因此需要手动执行快照时推荐使用BGSAVE命令；
2. 被动触发：
    - 根据配置规则进行自动快照，如`SAVE 300 10`,300秒内至少有10个键被修改则进行快照。
    - 如果从节点执行全量复制操作，主节点自动执行bgsave生成RDB文件并发送给从节点。
    - 默认情况下执行shutdown命令时，如果没有开启AOF持久化功能则自动执行bgsave。
    - 执行debug reload命令重新加载Redis时，也会自动触发save操作。

优点：Redis加载RDB恢复数据远远快于AOF的方式。

缺点：

1. RDB方式数据没办法做到实时持久化/秒级持久化。因为bgsave每次运行都要执行fork操作创建子进程，属于重量级操作，频繁执行成本过高。
2. 存在老版本Redis服务和新版本RDB格式兼容性问题。RDB文件使用特定二进制格式保存，Redis版本演进过程中有多个格式的RDB版本，存在老版本Redis服务无法兼容新版RDB格式的问题。

AOF方式

AOF（append only file）持久化：以独立日志的方式记录每次写命令， Redis重启时会重新执行AOF文件中的命令达到恢复数据的目的。AOF的主要作用是**解决了数据持久化的实时性**，目前已经是Redis持久化的主流方式。

默认情况下Redis没有开启AOF方式的持久化，可以通过appendonly参数启用`appendonly yes`。开启AOF方式持久化后每执行一条写命令，Redis就会将该命令写进aof_buf缓冲区，AOF缓冲区根据对应的策略向硬盘做同步操作。

默认情况下系统每30秒会执行一次同步操作。为了防止缓冲区数据丢失，可以在Redis写入AOF文件后主动要求系统将缓冲区数据同步到硬盘上。可以通过`appendfsync`参数设置同步的时机。

```
appendfsync always //每次写入aof文件都会执行同步，最安全最慢，只能支持几百TPS写入，不建议配置
appendfsync everysec  //保证了性能也保证了安全，建议配置
appendfsync no //由操作系统决定何时进行同步操作
```

重写机制：

随着命令不断写入AOF，文件会越来越大，为了解决这个问题，Redis引入AOF重写机制压缩文件体积。AOF文件重写是把Redis进程内的数据转化为写命令同步到新AOF文件的过程。

优点：
（1）AOF可以更好的保护数据不丢失，一般AOF会每秒去执行一次fsync操作，如果redis进程挂掉，最多丢失1秒的数据。
（2）AOF以appen-only的模式写入，所以没有任何磁盘寻址的开销，写入性能非常高。
缺点
（1）对于同一份文件AOF文件比RDB数据快照要大。

（2）不适合写多读少场景。
（3）数据恢复比较慢。

RDB和AOF如何选择
（1）仅使用RDB这样会丢失很多数据。
（2）仅使用AOF，因为这一会有两个问题，第一通过AOF恢复速度慢；第二RDB每次简单粗暴生成数据快照，更加安全健壮。
（3）综合AOF和RDB两种持久化方式，用AOF来保证数据不丢失，作为恢复数据的第一选择；用RDB来做不同程度的冷备，在AOF文件都丢失或损坏不可用的时候，可以使用RDB进行快速的数据恢复。

集群

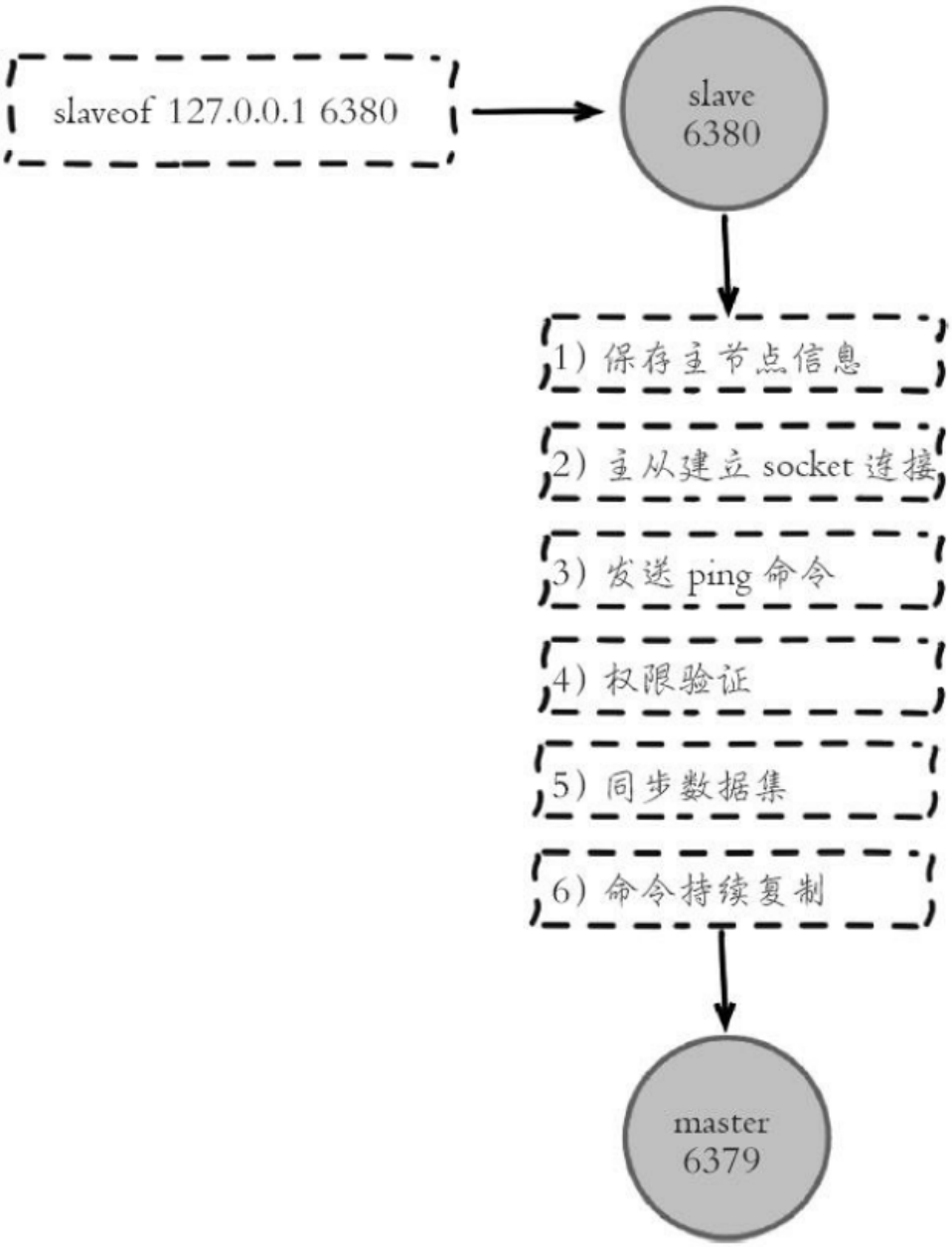
主从复制

redis的复制功能是支持多个数据库之间的数据同步。主数据库可以进行读写操作，当主数据库的数据发生变化时会自动将数据同步到从数据库。从数据库一般是只读的，它会接收主数据库同步过来的数据。一个主数据库可以有多个从数据库，而一个从数据库只能有一个主数据库。

```
redis-server //启动Redis实例作为主数据库
redis-server --port 6380 --slaveof  127.0.0.1 6379  //启动另一个实例作为从数据库
slaveof 127.0.0.1 6379
SLAVEOF NO ONE //停止接收其他数据库的同步并转化为主数据库。
```

同步机制

1. 保存主节点信息。
2. 主从建立socket连接。
3. 从节点发送ping命令进行首次通信，主要用于检测网络状态。
4. 权限认证。如果主节点设置了requirepass参数，则需要密码认证。从节点必须配置masterauth参数保证与主节点相同的密码才能通过验证。
5. 同步数据集。第一次同步的时候，从数据库启动后会向主数据库发送SYNC命令。主数据库接收到命令后开始在后台保存快照（RDB持久化过程），并将保存快照过程接收到的命令缓存起来。当快照完成后，Redis会将快照文件和缓存的命令发送到从数据库。从数据库接收到后，会载入快照文件并执行缓存的命令。以上过程称为复制初始化。
6. 复制初始化完成后，主数据库每次收到写命令就会将命令同步给从数据库，从而实现主从数据库数据的一致性。



Redis在2.8及以上版本使用psync命令完成主从数据同步，同步过程分为：全量复制和部分复制。

全量复制：一般用于初次复制场景，Redis早期支持的复制功能只有全量复制，它会把主节点全部数据一次性发送给从节点，当数据量较大时，会对主从节点和网络造成很大的开销。

部分复制：用于处理在主从复制中因网络闪断等原因造成的数据丢失场景，当从节点再次连上主节点后，如果条件允许，主节点会补发丢失数据给从节点。因为补发的数据远远小于全量数据，可以有效避免全量复制的过高开销。

读写分离

通过redis的复制功能可以实现数据库的读写分离，提高服务器的负载能力。主数据库主要进行写操作，而从数据库负责读操作。很多场景下对数据库的读频率大于写，当单机的Redis无法应付大量的读请求时，可以通过复制功能建立多个从数据库节点，主数据库负责写操作，从数据库负责读操作。这种一主多从的结构很适合读多写少的场景。

从数据库持久化

持久化的操作比较耗时，为了提高性能，可以建立一个从数据库，并在从数据库进行持久化，同时在主数据库禁用持久化。

哨兵Sentinel

当master节点奔溃时，可以手动将slave提升为master，继续提供服务。

- 首先，从数据库使用`SLAVE NO ONE`将从数据库提升为主数据库继续服务；
- 启动奔溃的主数据库，通过`SLAVEOF`命令将其设置为新的主数据库的从数据库，即可将数据同步过来。

通过哨兵机制可以自动切换主从节点。哨兵是一个独立的进程，用于监控redis实例的是否正常运行。

作用

1. 监测redis实例的状态
2. 如果master实例异常，会自动进行主从节点切换

客户端连接redis的时候，先连接哨兵，哨兵会告诉客户端redis主节点的地址，然后客户端连接上redis并进行后续的操作。当主节点宕机的时候，哨兵监测到主节点宕机，会重新推选出某个表现良好的从节点成为新的主节点，然后通过发布订阅模式通知其他的从服务器，让它们切换主机。

定时任务

1. 每隔10s，每个Sentinel节点会向主节点和从节点发送info命令获取最新的拓扑结构。
2. 每隔2s，每个Sentinel节点会去获取其他Sentinel节点对于主节点的判断以及当前Sentinel节点的信息，用于判断主节点是否客观下线和是否有新的Sentinel节点加入。
3. 每隔1s，每个Sentinel节点会向主节点、从节点、其余Sentinel节点发送一条ping命令做一次心跳检测，来确认这些节点是否可达。

工作原理

- 每个Sentinel以每秒钟一次的频率向它所知的Master，Slave以及其他 Sentinel 实例发送一个 PING 命令。
- 如果一个实例距离最后一次有效回复 PING 命令的时间超过指定的值， 则这个实例会被 Sentinel 标记为主观下线。
- 如果一个Master被标记为主观下线，则正在监视这个Master的所有 Sentinel 要以每秒一次的频率确认Master是否真正进入主观下线状态。
- 当有足够数量的 Sentinel（大于等于配置文件指定的值）在指定的时间范围内确认Master的确进入了主观下线状态， 则Master会被标记为客观下线 。若没有足够数量的 Sentinel 同意 Master 已经下线， Master 的客观下线状态就会被移除。 若 Master 重新向 Sentinel 的 PING 命令返回有效回复， Master 的主观下线状态就会被移除。
- 哨兵节点会选举出哨兵领导者，负责故障转移的工作。
- 哨兵领导者会推选出某个表现良好的从节点成为新的主节点，然后通知其他从节点更新主节点。

```
/**
 * 测试Redis哨兵模式
 * @author liu
 */
public class TestSentinels {
    @SuppressWarnings("resource")
    @Test
    public void testSentinel() {
        JedisPoolConfig jedisPoolConfig = new JedisPoolConfig();
```

```
        jedisPoolConfig.setMaxTotal(10);
        jedisPoolConfig.setMaxIdle(5);
        jedisPoolConfig.setMinIdle(5);
        // 哨兵信息
        Set<String> sentinels = new HashSet<>
(Arrays.asList("192.168.11.128:26379",
                "192.168.11.129:26379","192.168.11.130:26379"));
        // 创建连接池
        JedisSentinelPool pool = new JedisSentinelPool("mymaster",
sentinels,jedisPoolConfig,"123456");
        // 获取客户端
        Jedis jedis = pool.getResource();
        // 执行两个命令
        jedis.set("mykey", "myvalue");
        String value = jedis.get("mykey");
        System.out.println(value);
    }
}
```

cluster

集群用于分担写入压力，主从用于灾难备份和高可用以及分担读压力。

主从复制存在不能自动故障转移、达不到高可用的问题。
哨兵模式解决了主从复制不能自动故障转移、达不到高可用的问题，但还是存在主节点的写能力、容量受限于单机配置的问题。
cluster模式实现了Redis的分布式存储，每个节点存储不同的内容，解决主节点的写能力、容量受限于单机配置的问题。

> 总结，主从和哨兵可以饥饿绝高可用、高并发读的问题。但以下两个问题依然没有得到解决：
>
> - 海量数据存储问题
> - 高并发写问题
>
> 使用分片集群可以解决上述问题，分片集群的特征：
>
> - 集群有多个master，每个master保存不同的数据
> - 每个master都可以有多个slave
> - master之间通过ping监测彼此健康状态
> - 客户端请求可以访问集群任意节点，最终都会被转发到正确的节点

哈希分区算法

节点取余分区。使用特定的数据，如Redis的键或用户ID，对节点数量N取余：hash（key）%N计算出哈希值，用来决定数据映射到哪一个节点上。
优点是简单性。扩容时通常采用翻倍扩容，避免数据映射全部被打乱导致全量迁移的情况。

一致性哈希分区：为系统中每个节点分配一个token，范围一般在0~232，这些token构成一个哈希环。数据读写执行节点查找操作时，先根据key计算hash值，然后顺时针找到第一个大于等于该哈希值的token节点。
这种方式相比节点取余最大的好处在于加入和删除节点只影响哈希环中相邻的节点，对其他节点无影响。

Redis Cluser采用虚拟槽分区，所有的键根据哈希函数映射到0~16383整数槽内，计算公式：slot=CRC16（key）&16383。每一个节点负责维护一部分槽以及槽所映射的键值数据。

故障转移

Redis集群内节点通过ping/pong消息实现节点通信，消息不但可以传播节点槽信息，还可以传播其他状态如：主从状态、节点故障等。因此故障发现也是通过消息传播机制实现的，主要环节包括：主观下线（pfail）和客观下线（fail）。

LUA脚本

Redis 通过 LUA 脚本创建具有原子性的命令： 当lua脚本命令正在运行的时候，不会有其他脚本或 Redis 命令被执行，实现组合命令的原子操作。

在Redis中执行Lua脚本有两种方法：eval和evalsha。

eval 命令使用内置的 Lua 解释器，对 Lua 脚本进行求值。

```
//第一个参数是lua脚本，第二个参数是键名参数个数，剩下的是键名参数和附加参数
> eval "return {KEYS[1],KEYS[2],ARGV[1],ARGV[2]}" 2 key1 key2 first second
1) "key1"
2) "key2"
3) "first"
4) "second"
```

evalsha

Redis还提供了evalsha命令来执行Lua脚本。首先要将Lua脚本加载到Redis服务端，得到该脚本的SHA1校验和。Evalsha 命令根据给定的 sha1 校验和，执行缓存在服务器中的脚本。

script load命令可以将脚本内容加载到Redis内存中。

```
redis 127.0.0.1:6379> SCRIPT LOAD "return 'hello moto'"
"232fd51614574cf0867b83d384a5e898cfd24e5a"

redis 127.0.0.1:6379> EVALSHA "232fd51614574cf0867b83d384a5e898cfd24e5a" 0
"hello moto"
```

使用evalsha执行Lua脚本过程如下：

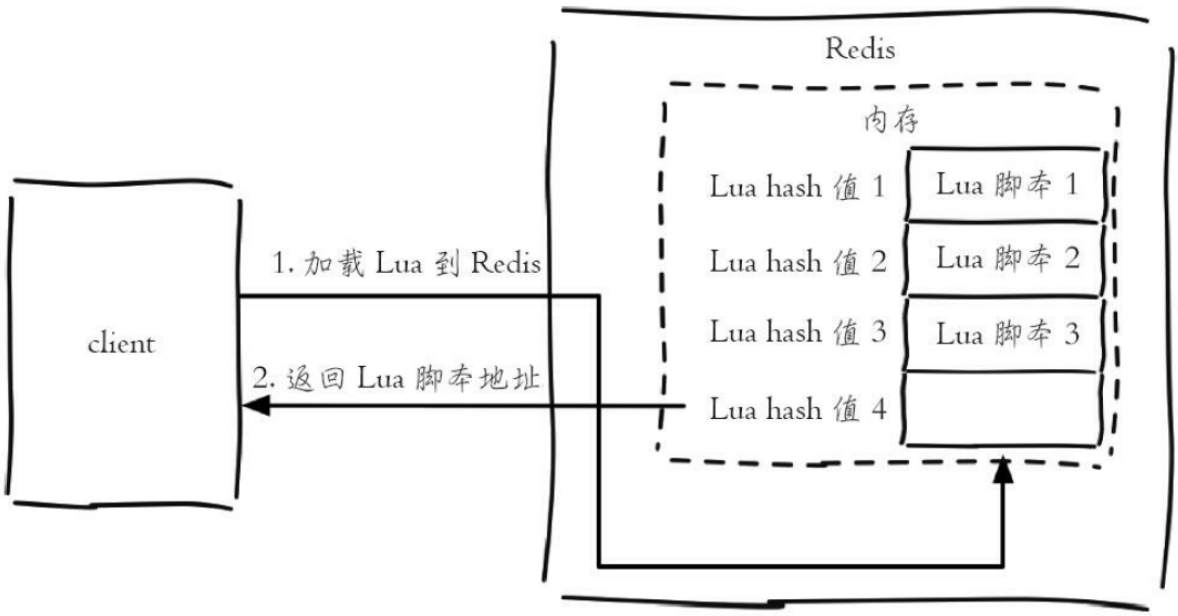


图3-8　使用evalsha执行Lua脚本过程

lua脚本作用

1、Lua脚本在Redis中是原子执行的，执行过程中间不会插入其他命令。

2、Lua脚本可以将多条命令一次性打包，有效地减少网络开销。

应用场景

限制接口访问频率。

在Redis维护一个接口访问次数的键值对，key是接口名称，value是访问次数。每次访问接口时，会执行以下操作：

- 通过aop拦截接口的请求，对接口请求进行计数，每次进来一个请求，相应的接口count加1，存入redis。
- 如果是第一次请求，则会设置count=1，并设置过期时间。因为这里set()和expire()组合操作不是原子操作，所以引入lua脚本，实现原子操作，避免并发访问问题。
- 如果给定时间范围内超过最大访问次数，则会抛出异常。

```
private String buildLuaScript() {
    return "local c" +
        "\nc = redis.call('get',KEYS[1])" +
        "\nif c and tonumber(c) > tonumber(ARGV[1]) then" +
        "\nreturn c;" +
        "\nend" +
        "\nc = redis.call('incr',KEYS[1])" +
        "\nif tonumber(c) == 1 then" +
        "\nredis.call('expire',KEYS[1],ARGV[2])" +
        "\nend" +
        "\nreturn c;";
}

String luaScript = buildLuaScript();
RedisScript<Number> redisScript = new DefaultRedisScript<>(luaScript,
Number.class);
Number count = redisTemplate.execute(redisScript, keys, limit.count(),
limit.period());
```

删除策略

1. 被动删除。在访问key时，如果发现key已经过期，那么会将key删除。
2. 主动删除。定时清理key，每次清理会依次遍历所有DB，从db随机取出20个key，如果过期就删除，如果其中有5个key过期，那么就继续对这个db进行清理，否则开始清理下一个db。
3. 内存不够时清理。Redis有最大内存的限制，通过maxmemory参数可以设置最大内存，当使用的内存超过了设置的最大内存，就要进行内存释放， 在进行内存释放的时候，会按照配置的淘汰策略清理内存，淘汰策略一般有6种，Redis4.0版本后又增加了2种，主要由分为三类：
    - 第一类 不处理 noeviction。发现内存不够时，不删除key，执行写入命令时直接返回错误信息。（默认的配置）
    - 第二类 从所有结果集中的key中挑选，进行淘汰
        - allkeys-random 就是从所有的key中随机挑选key，进行淘汰
        - allkeys-lru 就是从所有的key中挑选最近最少使用的数据淘汰
        - allkeys-lfu 就是从所有的key中挑选使用频率最低的key，进行淘汰。（这是Redis 4.0版本后新增的策略）
    - 第三类 从设置了过期时间的key中挑选，进行淘汰

      这种就是从设置了expires过期时间的结果集中选出一部分key淘汰，挑选的算法有：
        - volatile-random 从设置了过期时间的结果集中随机挑选key删除。
        - volatile-lru 从设置了过期时间的结果集中挑选最近最少使用的数据淘汰
        - volatile-ttl 从设置了过期时间的结果集中挑选可存活时间最短的key开始删除(也就是从哪些快要过期的key中先删除)
        - volatile-lfu 从过期时间的结果集中选择使用频率最低的key开始删除（这是Redis 4.0版本后新增的策略）

其他

客户端

Redis 客户端与服务端之间的通信协议是在TCP协议之上构建的。

Redis Monitor 命令用于实时打印出 Redis 服务器接收到的命令，调试用。

```
redis 127.0.0.1:6379> MONITOR
OK
1410855382.370791 [0 127.0.0.1:60581] "info"
1410855404.062722 [0 127.0.0.1:60581] "get" "a"
```

慢查询

Redis原生提供慢查询统计功能，执行`slowlog get{n}`命令可以获取最近的n条慢查询命令，默认对于执行超过10毫秒的命令都会记录到一个定长队列中，线上实例建议设置为1毫秒便于及时发现毫秒级以上的命令。慢查询队列长度默认128，可适当调大。

Redis客户端执行一条命令分为4个部分：发送命令；命令排队；命令执行；返回结果。慢查询只统计命令执行这一步的时间，所以没有慢查询并不代表客户端没有超时问题。

Redis提供了`slowlog-log-slower-than`（设置慢查询阈值，单位为微秒）和`slowlog-max-len`（慢查询队列大小）配置慢查询参数。

相关命令：

```
showlog get n //获取慢查询日志
slowlog len //慢查询日志队列当前长度
slowlog reset //重置，清理列表
```

慢查询解决方案：

1. 修改为低时间复杂度的命令，如hgetall改为hmget等，禁用keys、sort等命令。
2. 调整大对象：缩减大对象数据或把大对象拆分为多个小对象，防止一次命令操作过多的数据。

pipeline

redis客户端执行一条命令分4个过程： 发送命令 - 〉命令排队 - 〉命令执行 - 〉返回结果。使用Pipeline可以批量请求，批量返回结果，执行速度比逐条执行要快。

使用pipeline组装的命令个数不能太多，不然数据量过大，增加客户端的等待时间，还可能造成网络阻塞，可以将大量命令的拆分多个小的pipeline命令完成。

原生批命令(mset, mget)与Pipeline对比：

1. 原生批命令是原子性，pipeline是非原子性。pipeline命令中途异常退出，之前执行成功的命令不会回滚。
2. 原生批命令只有一个命令, 但pipeline支持多命令。

数据一致性

缓存和DB之间怎么保证数据一致性：
读操作：先读缓存，缓存没有的话读DB，然后取出数据放入缓存，最后响应数据
写操作：先删除缓存，再更新DB
为什么是删除缓存而不是更新缓存呢?

1. 线程安全问题。同时有请求A和请求B进行更新操作，那么会出现(1)线程A更新了缓存(2)线程B更新了缓存(3)线程B更新了数据库(4)线程A更新了数据库，由于网络等原因，请求B先更新数据库，这就导致缓存和数据库不一致的问题。
2. 如果业务需求写数据库场景比较多，而读数据场景比较少，采用这种方案就会导致，数据压根还没读到，缓存就被频繁的更新，浪费性能。
3. 如果你写入数据库的值，并不是直接写入缓存的，而是要经过一系列复杂的计算再写入缓存。那么，每次写入数据库后，都再次计算写入缓存的值，无疑是浪费性能的。

先删除缓存，再更新DB，同样也有问题。假如A先删除了缓存，但还没更新DB，这时B过来请求数据，发现缓存没有，去请求DB拿到旧数据，然后再写到缓存，等A更新完了DB之后就会出现缓存和DB数据不一致的情况了。

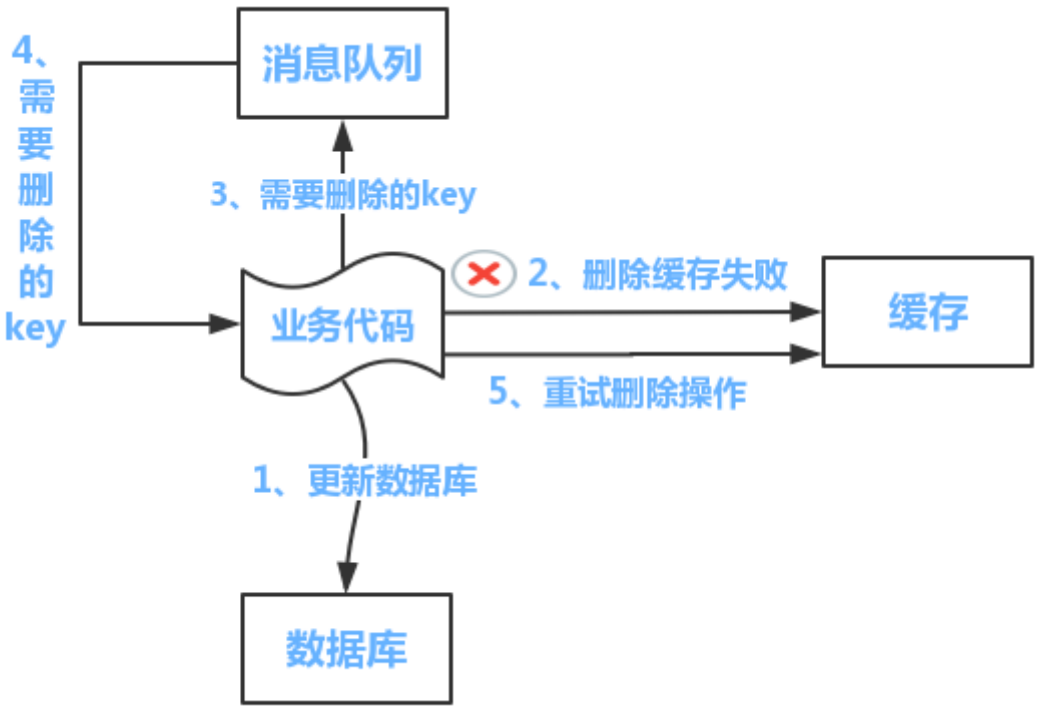
解决方法：采用延时双删策略。更新完数据库之后，延时一段时间，再次删除缓存，确保可以删除读请求造成的缓存脏数据。评估项目的读数据业务逻辑的耗时。然后写数据的休眠时间则在读数据业务逻辑的耗时基础上，加几百ms即可。

```
public void write(String key,Object data){
    redis.delKey(key);
    db.updateData(data);
    Thread.sleep(1000);//确保读请求结束，写请求可以删除读请求造成的缓存脏数据
    redis.delKey(key);
}
```

可以将第二次删除作为异步的。自己起一个线程，异步删除。这样，写的请求就不用沉睡一段时间后了，加大吞吐量。

当删缓存失败时，也会就出现数据不一致的情况。

解决方法：



> 图片来源：https://tech.it168.com