《Redis深度历险》总结

写在开头

《Redis深度历险》感觉一般，感觉《Redis设计与实现》更好一点，quicklist，ziplist和skiplist都是看的《Redis设计与实现》，感觉《Redis设计与实现》讲的比较简单易懂，而《Redis深度历险》讲的东西可能更干一点。

《Redis设计与实现》。额，我又大体看了一下，发现《Redis设计与实现》也感觉一般，还是针对性搜博客看吧，书里面的冗余内容还是比较多，深度也一般。

1.Redis是什么

Redis是互联网领域应用广泛的存储中间件，是“Remote Dictionary Service”(远程字典服务)的缩写;Redis可以看作是一个内存数据库，一种KV的NOSQL样式，它被广泛应用在阿里，百度，美团，滴滴等公司当中。

2.Redis的作用

Redis是一种内存数据库，常常被用作缓存。以滴滴courier-track(骑手轨迹)和courier-lbs(骑手召回)举例来说，首先courier-lbs会从上游courier-dataflow(数据流分发)获得骑手的信息以及坐标数据，这时，可以按照城市id_小时粒度的时间作为redies的key值，而value值是若干骑手id，将key和value存储到redis当中，便可在courier-track当中书写函数，获得某个小时某个城市的所有骑手id。同理，通过骑手id可以获得骑手缓存在redis当中的压缩轨迹，即可以在courier-track当中实现http通过时间和城市获取城市当中所有骑手的轨迹。

3.Redis当中的基础数据结构

1.string(字符串)

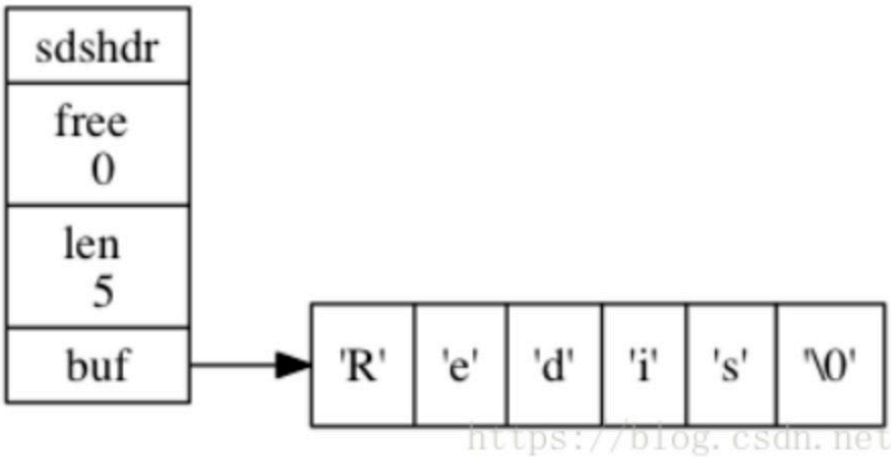
概念：Redis当中的字符串是动态字符串，类似于Java当中的ArrayList，采用预分配冗余空间的方式减少内存的频繁分配。

扩容：字符串长度在小于1MB时扩容时加倍现有的空间；如果字符串长度超过1MB，扩容一次只会多扩1MB的空间，其最大长度为512MB;当字符串长度变小时，并不真正的将空间回收，而是修改free和len的长度。

实现：Redis是使用C语言进行编码，String的实现是采用了一种SDS动态字符串的结构体。

```
struct sdshdr {
  int len//lenkeyvalue
  int free//value
  char buf[]; //len(buf)=len+free+1
}
```

在string当中保存一个value的形式如下：



2.list(列表)

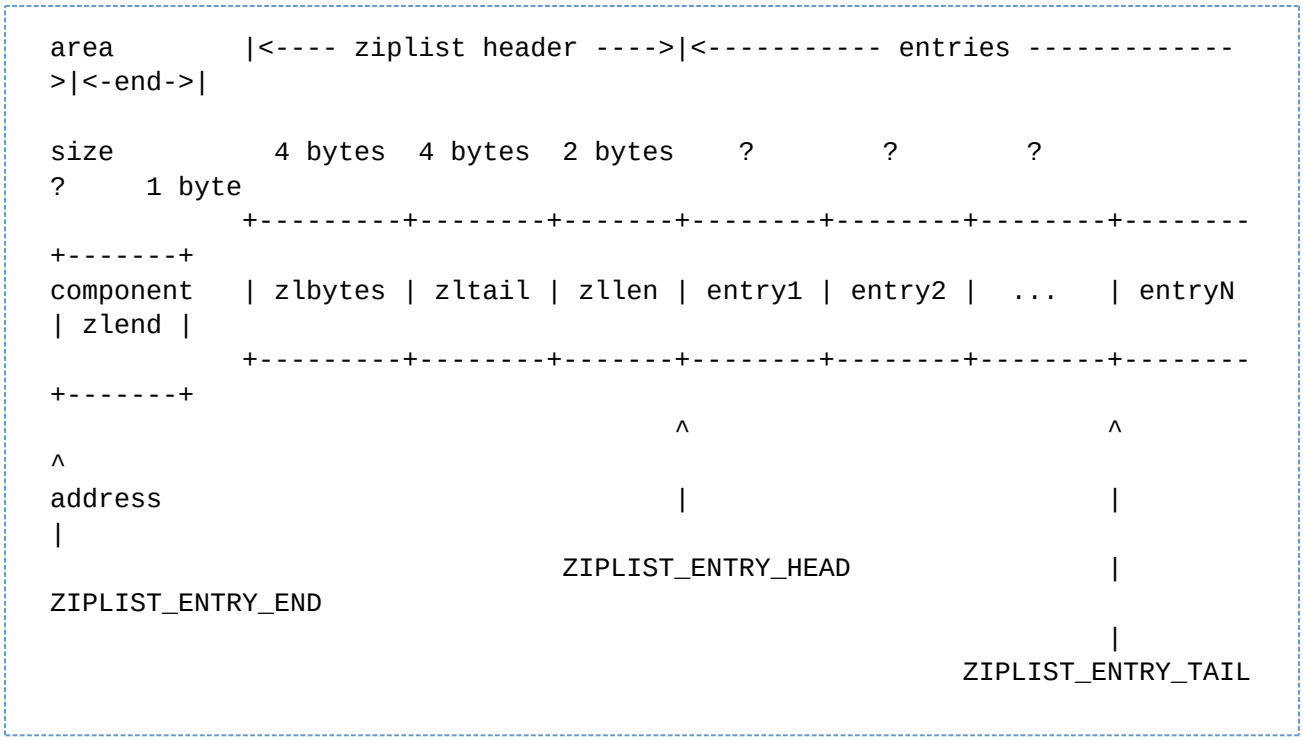
概念：Redis当中的list(列表)类似于Java当中的LinkedList，其插入和删除操作都是比较快的，O(1)复杂度，列表中每个元素都使用双向指针串接，可以同时支持前向后向遍历。

实现：在元素比较少的情况下，list会使用一块连续的内存存储，这个结构就是ziplist;而如果元素比较多时，会使用quicklisl,quicklist是一种将ziplist使用指针链接起来的结构。

lziplist

概念：Ziplist 是由一系列特殊编码的内存块构成的列表， 一个 ziplist 可以包含多个节点（entry）， 每个节点可以保存一个长度受限的字符数组（不以 `\0` 结尾的 `char` 数组）或者整数。

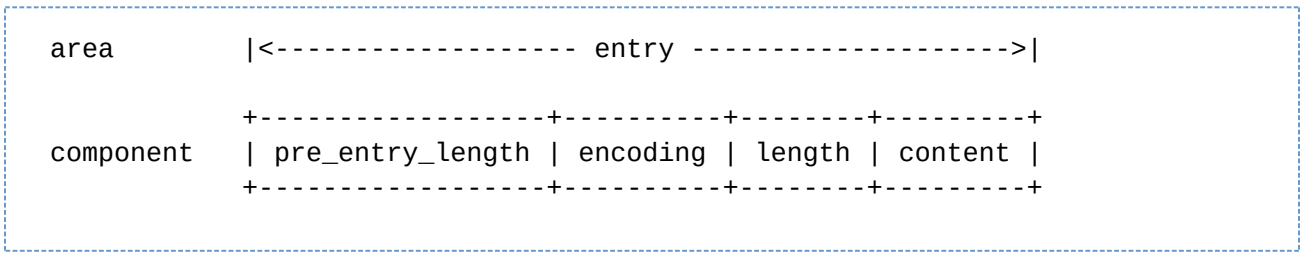
ziplist构成：



图中各个域的作用如下：

域	长度/类型	域的值
zlbytes	uint32_t	整个 ziplist 占用的内存字节数，对 ziplist 进行内存重分配，或者计算末端时使用。
zltail	uint32_t	到达 ziplist 表尾节点的偏移量。 通过这个偏移量，可以在不遍历整个 ziplist 的前提下，弹出表尾节点。
zllen	uint16_t	ziplist 中节点的数量。 当这个值小于 UINT16_MAX （65535）时，这个值就是 ziplist 中节点的数量； 当这个值等于 UINT16_MAX 时，节点的数量需要遍历整个 ziplist 才能计算得出。
entryX	?	ziplist 所保存的节点，各个节点的长度根据内容而定。
zlend	uint8_t	255 的二进制值 1111 1111 （UINT8_MAX），用于标记 ziplist 的末端。

entry构成：



各个域作用如下：

域	作用
pre_entry_length	pre_entry_length 记录了前一个节点的长度，通过这个值，可以进行指针计算，从而跳转到上一个节点。

<table>
<tr><td>encoding 和 length</td><td>encoding 和 length 两部分一起决定了 content 部分所保存的数据的类型（以及长度）

其中， encoding 域的长度为两个 bit ， 它的值可以是 00 、 01 、 10 和 11 ：
<table><tr><th>编码</th><th>编码长度</th><th>content 部分保存的值</th></tr><tr><td>00bbbbbb</td><td>1 byte</td><td>长度小于等于 63 字节的字符数组</td></tr><tr><td>01bbbbbb xxxxxxxx</td><td>2 byte</td><td>长度小于等于 16383 字节的字符数组</td></tr><tr><td>10____ aaaaaaaa bbbbbbbb cccccccc dddddddd</td><td>5 byte</td><td>长度小于等于 4294967295 的字符数组</td></tr><tr><td>11000000</td><td>1 byte</td><td>int16_t 类型的整数</td></tr><tr><td>11010000</td><td>1 byte</td><td>int32_t 类型的整数</td></tr><tr><td>11100000</td><td>1 byte</td><td>int64_t 类型的整数</td></tr><tr><td>11110000</td><td>1 byte</td><td>24 bit 有符号整数</td></tr><tr><td>11111110</td><td>1 byte</td><td>8 bit 有符号整数</td></tr><tr><td>1111xxxx</td><td>1 byte</td><td>4 bit 无符号整数，介于 0 至 12 之间</td></tr></table></td></tr>
<tr><td>content</td><td>content 部分保存着节点的内容，类型和长度由 encoding 和 length 决定</td></tr>
</table>

entry节点也包括插入、删除等操作，具体见：压缩列表——Redis设计与实现

2quicklist

概念：一个以ziplist为节点的双向链表。在redis3.2之后，quicklist取代了压缩列表和linkedlist，成为了列表对象的唯一编码形式

原因：linkedlist由于各个节点都是单独的内存，很容易造成内存碎片；而对于压缩列表，由于其每次修改都会引发内存的重新分配，导致大量的内存拷贝。经过对时间和空间的折中，选择了quicklist这种方法。

节点结构：

```
typedef struct quicklistNode {
    struct quicklistNode *prev; //
    struct quicklistNode *next; //
    unsigned char *zl; // ziplistziplistquicklistLZF
    unsigned int sz;             /* ziplist */
    unsigned int count : 16;     /* ziplistitem*/
    unsigned int encoding : 2;   /* ziplist12*/
    unsigned int container : 2;  /* 2ziplist */
    unsigned int recompress : 1; /* 1*/
    unsigned int attempted_compress : 1; /*  */
    unsigned int extra : 10; /* 32bit */
} quicklistNode;
```

链表结构：

```
typedef struct quicklist {
    quicklistNode *head; //
    quicklistNode *tail; //
    unsigned long count;        /* ziplistitem */
    unsigned long len;          /*  */
    int fill : 16;              /* ziplist */
    unsigned int compress : 16; /*  */
} quicklist;
```

3.hash(字典)

概念：Redis当中的hash(字典)相当于Java当中的HashMap，是一个无序字典，内部存储了很多键值对。

实现：类似于Java当中的HashMap，是数组+链表的形式。但不同之处在于，Redis当中的hash中能存储字符串，并且使用的是渐进rehash。

1完全hash

即暂停线程，等旧的hash完全迁移到新hash再开始工作

2渐进式rehash

渐进式rehash主要维护了新旧两个hash表，设置一个定时任务，将旧的hash表定时定量的向新hash表进行移动，在该过程中，插入hash表的数据都是向新hash表插入，直到旧hash表完全移动结束，删除旧hash表。

在查询过程中，如果需要同时访问新旧两个hash表。

美团针对Redis Rehash机制的探索和实践

4.set(集合)

概念：Redis当中的set相当于Java当中的HashSet，即固定了HashMap当中的value，用key来保存数据

5.zset(有序集合)

概念：Redis当中最具有特色的数据结构，类似于Java当中SortedSet和HashMap的结合体，一方面它是一个set，可以保证内部数据的唯一性；另一方面它的每一个value都具有一个score，可以按照score排序

1skiplist

概念：跳跃表（skiplist）是一种有序数据结构， 它通过在每个节点中维持多个指向其他节点的指针， 从而达到快速访问节点的目的。

实现：skiplist主要由zskiplistNode和zskiplist两个结构实现， 其中 `zskiplistNode` 结构用于表示跳跃表节点， 而 `zskiplist` 结构则用于保存跳跃表节点的相关信息。如图所示：

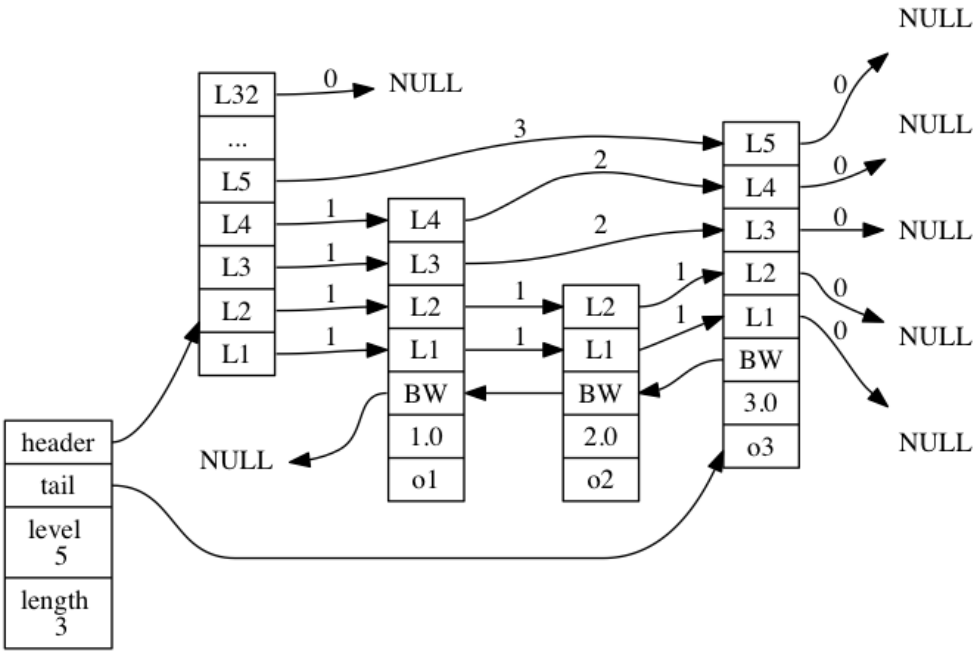


图 5-1　一个跳跃表

位于图片最左边的是 `zskiplist` 结构， 该结构包含以下属性：

- `header` ：指向跳跃表的表头节点。
- `tail` ：指向跳跃表的表尾节点。
- `level` ：记录目前跳跃表内，层数最大的那个节点的层数（表头节点的层数不计算在内）。
- `length` ：记录跳跃表的长度，也即是，跳跃表目前包含节点的数量（表头节点不计算在内）

```
typedef struct zskiplist {
    //
    struct zskiplistNode *header, *tail;
    //
    unsigned long length;
    //
    int level;
} zskiplist;
```

位于 `zskiplist` 结构右方的是四个 `zskiplistNode` 结构， 该结构包含以下属性：

- 层（level）：节点中用 `L1` 、 `L2` 、 `L3` 等字样标记节点的各个层， `L1` 代表第一层， `L2` 代表第二层，以此类推。每个层都带有两个属性：前进指针和跨度。前进指针用于访问位于表尾方向的其他节点，而跨度则记录了前进指针所指向节点和当前节点的距离。在上面的图片中，连线上带有数字的箭头就代表前进指针，而那个数字就是跨度。当程序从表头向表尾进行遍历时，访问会沿着层的前进指针进行。
- 后退（backward）指针：节点中用 `BW` 字样标记节点的后退指针，它指向位于当前节点的前一个节点。后退指针在程序从表尾向表头遍历时使用。
- 分值（score）：各个节点中的 `1.0` 、 `2.0` 和 `3.0` 是节点所保存的分值。在跳跃表中，节点按各自所保存的分值从小到大排列。
- 成员对象（obj）：各个节点中的 `o1` 、 `o2` 和 `o3` 是节点所保存的成员对象。

```
typedef struct zskiplistNode {
    //
    struct zskiplistNode *backward;
    //
    double score;
    //
    robj *obj;
    //
    struct zskiplistLevel {
        //
        struct zskiplistNode *forward;
        //
        unsigned int span;
    } level[];
} zskiplistNode;
```

注意表头节点和其他节点的构造是一样的： 表头节点也有后退指针、分值和成员对象， 不过表头节点的这些属性都不会被用到， 所以图中省略了这些部分， 只显示了表头节点的各个层。

跳跃表的实现

4.Redis当中的高级数据结构

1.HyperLogLog

使用场景：页面的UV数据，即统计页面的访问量,Redis当中的HyperLogLog提供不精确的去重计数方案，标准误差是0.81%。

命令：HyperLogLog提供pfadd,pfcount命令以供统计计数

2.布隆过滤器

使用场景：HyperLogLog只能统计次数，即无法判断某个用户是否访问过该网页，而布隆过滤器则可以提供类似功能。布隆过滤器类似一个不是很精确的set结构，并提供一个contains方法判断某个对象是否存在，但该结构也会有误判几率。

过滤策略：布隆过滤器说某个值存在时，这个值可能是不存在的；当布隆过滤器说某个值不存在时，这个值一定是不存在的。即当布隆过滤器说某个用户访问过某个网页时，这个用户不一定访问过；而当布隆过滤器说某个用户没有访问过某个网页时，该用户一定没有访问过。

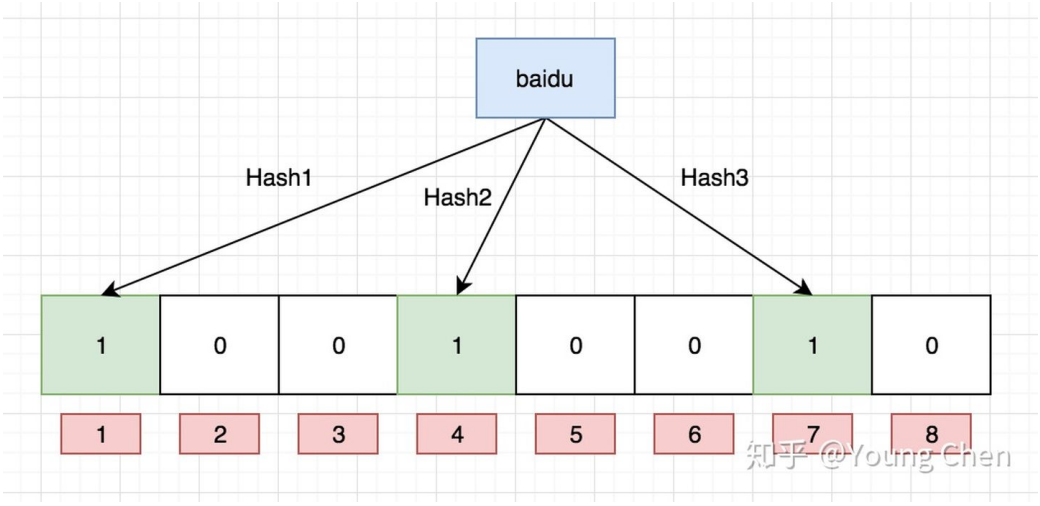
适用场景：用户推荐

命令：bf.add,bf.exists,bf.reserve

注意：布隆过滤器的initial_size如果设置的过大，则会浪费存储空间；如果设置的过小，则会影响准确率。

原理：

布隆过滤器的基本原理是对于输入的key，进行多个无偏hash函数f,g,h计算，计算出多个hash值，将这多个hash值的位置都设置成1，这就完成了add操作;exit操作同理，比较多个hash函数计算出的hash值所在位置是否为1，如果全都为1，表示该key可能被add过;如果不全为1，则表示该key一定不存在。

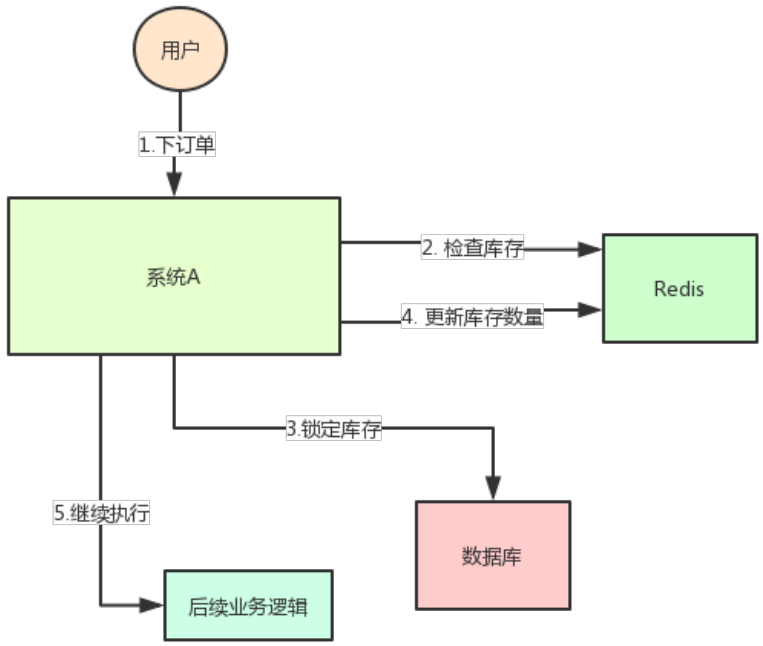


5.Redis分布式锁

1.使用场景

系统A是一个电商系统，目前是一台机器部署，系统中有一个用户下订单的接口，但是用户下订单之前一定要去检查一下库存，确保库存足够了才会给用户下单。

由于系统有一定的并发，所以会预先将商品的库存保存在redis中，用户下单的时候会更新redis的库存。此时系统架构如下：



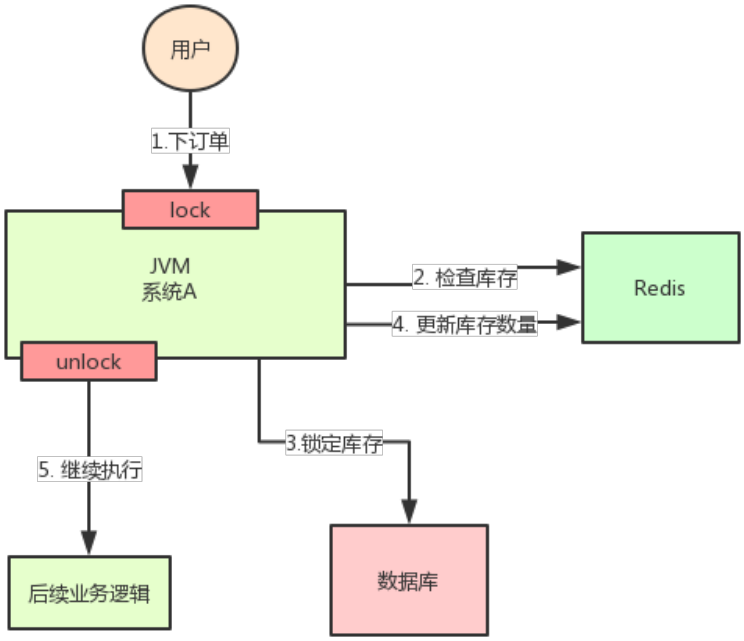
但是这样一来会产生一个问题：假如某个时刻，redis里面的某个商品库存为1，此时两个请求同时到来，其中一个请求执行到上图的第3步，更新数据库的库存为0，但是第4步还没有执行。

而另外一个请求执行到了第2步，发现库存还是1，就继续执行第3步。

这样的结果，是导致卖出了2个商品，然而其实库存只有1个。

这就是典型的问题

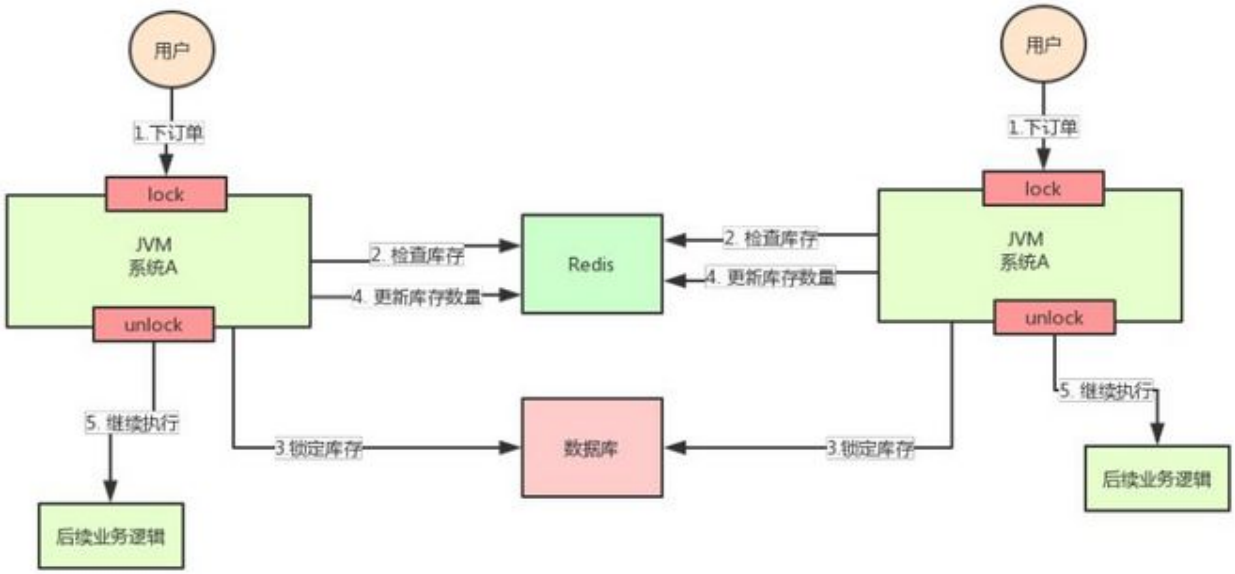
此时，我们很容易想到解决方案：用锁把2、3、4步锁住，让他们执行完之后，另一个线程才能进来执行第2步。



按照上面的图，在执行第2步时，使用`JavasynchronizedReentrantLock`，然后在第4步执行完之后才释放锁。

这样一来，2、3、4 这3个步骤就被“锁”住了，多个线程之间只能串行化执行。

但是好景不长，整个系统的并发飙升，一台机器扛不住了。现在要增加一台机器，如下图：



假设此时两个用户的请求同时到来，但是落在了不同的机器上，那么的问题。

原因：

- 因为上图中的两个A系统，运行在两个不同的JVM里面，他们加的锁只对属于自己JVM里面的线程有效，对于其他JVM的线程是无效的。
- 因此，这里的问题是：Java提供的原生锁机制在多机部署场景下失效了这是因为两台机器加的锁不是同一个锁(两个锁在不同的JVM里面)。

6.Redis限流

1.简单限流

场景：系统要限定用户的某个行为在指定的时间里只能允许发生N次

解决：运用zset维护一个时间窗，key值可以用userid_action代替，value值可以用毫秒时间戳或其它任意唯一值，score用毫秒时间戳。

用户每次action_add，则将zset时间窗以外的记录全部删除，即zremrangeByScore(key,0,nowTs-period*1000)，只保留时间窗内的记录，然后统计个数，判断是否大于N。

2.漏斗限流

3.Redis-cell

概念:Redis4.0中提供的限流模块，该模块也使用了限流算法，并提供了原子限流指令。

命令：cl.throttle

举例：cl.throttle 行为 15 30 60 1

其中，15为漏洞的容量，30 operations/ 60 seconds 表示漏水的速率 1是可选选项

7.Redis的IO机制

Redis是一个单线程的程序

8.Redis的持久化机制

8.1RDB持久化

概念：快照是Redis内存数据的一次全量备份，是内存数据的二进制序列化形式，在存储上非常紧凑。每N分钟数据发送了M次写操作之后，从内存dump数据形成rdb文件，压缩后放在备份目录

实现：Redis内完成RDB持久化的方法有rdbSave和rdbSaveBackground两个函数方法（源码文件rdb.c中），简单说下两者差别：

- rdbSave：是同步执行的，方法调用后就会立刻启动持久化流程。由于Redis是单线程模型，持久化过程中会阻塞，Redis无法对外提供服务；
- rdbSaveBackground：是后台（异步）执行的，该方法会fork出子进程，真正的持久化过程是在子进程中执行的（调用rdbSave），主进程会继续提供服务。其中主要涉及三个参数，saveparams，dirty和lastsave属性。saveparams数组保存了秒和次数两种结构，其意义是当n秒或超过m次写入就会进行bgsave；而dirty计数器和lastsave属性，dirty计数器保存了上一次save命令或bgsave命令后至今的修改次数；lastsave属性记录了上一次save或bgsave的修改时间。

8.2AOF持久化

概念：AOF是连续的增量备份，AOF日志记录的是内存数据修改的指令记录文本。

实现：AOF文件建立可以分为命令追加，文件写入，文件同步三个步骤；当AOF持久化功能处于开启状态时，服务器在执行一个写入操作后，会以协议格式将被执行的写命令追加到服务器状态的aof_buf缓冲区的末尾。而文件的写入和同步是指将缓冲区当中的内容写入到文件当中。而AOF文件建立后，只需要执行一遍AOF文件，则可以还原数据库状态。过程如下：

1）创建一个不带网络连接的伪客户端（fake client）：因为 Redis 的命令只能在客户端上下文中执行，而载入 AOF 文件时所使用的命令直接来源于 AOF 文件而不是网络连接，所以服务器使用了一个没有网络连接的伪客户端来执行 AOF 文件保存的写命令，伪客户端执行命令的效果和带网络连接的客户端执行命令的效果完全一样。

2）从 AOF 文件中分析并读取出一条写命令。

3）使用伪客户端执行被读出的写命令。

4）一直执行步骤 2 和步骤 3，直到 AOF 文件中的所有写命令都被处理完毕为止。

当完成以上步骤之后，AOF 文件所保存的数据库状态就会被完整地还原出来，整个过程如图 11-2 所示。

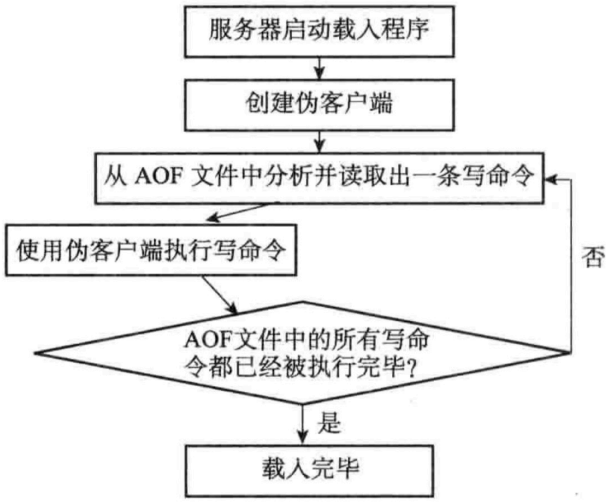


图 11-2　AOF 文件载入过程

然后，AOF文件的体积会不断膨胀，需要一个AOF重写的功能来解决这个问题，具体可以谷歌去看。

9.Redis集群