## 4.1事务

**redis中事务是一组命令的集合**。

**事务同命令一样都是redis的最小执行单位，Redis保证一个事务中的命令要么都执行，要么都不执行**。如果redisClient在发送EXEC命令前掉线，则redis会清空事务队列，事务中的所有命令都不会执行；如果redisClient在发送EXEC命令后掉线，所有的命令依然会被执行，因为redis中已经记录了所有要执行的命令了。

另外，**Redis保证一个事务从开始执行到执行结束期间，不会穿插的执行其它的命令或事务**。

|  |
| --- |
| **redis事务举例** |
|  |

### 4.1.2 redis事务的错误处理

|  |
| --- |
| **命令执行出错的两种情况介绍：** |
|  |
|  |

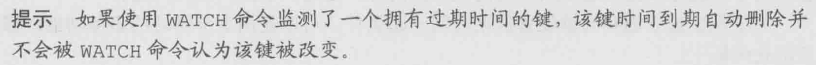
注意：**redis事务没有回滚功能**。对于刚才提到的会导致事务执行失败的两种错误，语法错误完全可以在开发时找出并解决；运行错误可以通过做好规划键名规范等来杜绝（这样就不会出现命令与键类型不匹配的情况了）。

### 4.1.3 Watch与UnWatch

|  |  |
| --- | --- |
| **Watch场景** | |
| 场景：在事务中，有时候需要先获得一条命令的返回值，然后再根据这个值执行下一条命令；例如用事务来实现incr函数以防止竟态条件。  思路：要保证从get获得键值开始到incr函数执行结束为止，该键值不会被其它客户端修改，这样也可以防止竟态条件。  **WATCH命令**：用于监控一个或多个键，一旦其中有一个键被修改或者删除，之后的事务就不会执行（EXEC返回空结果nil）；**监控一直持续到EXEC命令（因为事务中的命令是在EXEC之后执行的，所以在事务中可以修改WATCH监控的键值）**。 | |
| 存在竟态条件的原始伪码 | 用事务改造以防止竟态条件 |
| def incr($key)  $value = GET $key  if not $value  $value = 0  $value = $value + 1  set $key, $value  return $value | def incr($key)  WATCH $key  $value = GET $key  if not $value  $value = 0  $value = $value + 1  MULTI  set $key, $value  result = EXEC  // TODO result为nil则递归调用incr，否则返回结果  return result[0] |
| **UnWatch场景** | |
|  | |

## 4.2 过期时间

|  |  |
| --- | --- |
| expire key seconds | >> 设置（更新）一个键的过期时间，到期后redis会自动删除。  1表示设置成功；  0表示键不存在或设置失败。 |
| **p**expire key mseconds | 毫秒 |
| ttl key | >> 返回一个键还有多久到期(秒)；  如果键不存在则返回-2；  如果键是永久的（即没有为键设置过期时间）则返回-1。 |
| **p**ttl key | 毫秒 |
| persist key | >> 取消键的过期时间设置；  1表示过期时间被成功清除；  否则返回0（键不存在或键本身就是永久的）  **注意：set和getset为键赋值也会同时清除键的过期时间。** |
| expireAt key utcSeconds  pexpireAt key utlMSeconds | 两个不常用的命令，设置键的过期**时刻** |



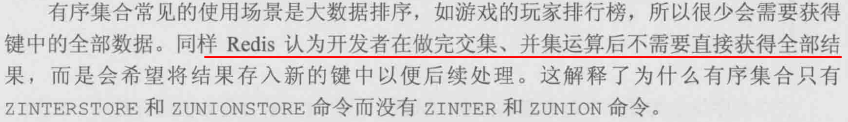
|  |
| --- |
| **实践** |
| **1、实现访问频率限制之方案1**    此方案的问题: 极端情况下，一个ip在第1分钟的第1秒访问了1次博客，建了一个键并设置60秒到期时间，然后在第1分钟最后一秒又访问80次，这样第一分钟内总共访问81次，是可以通过访问频率限制的，此时该键到期被删除。然后在第2分钟第一秒又访问了1次，会再建同名键并设置60秒的到期时间，第二分钟第2秒又访问了80次，依然可以通过访问频率限制。但是**问题**是从第1分钟最后1秒到第2分钟第2秒这3秒时间内访问次数是超过了100次的，**我们需要粒度更小的控制方案**。 |
| **2、实现访问频率限制之方案2（解决方案1的问题）**  如果要精确的保证每个分钟最多访问100次，需要**记录下用户每次访问的时间**。对每个ip，使用一个***列表类型键***来保存他最近100次访问博客的时间。 |
| **3、实现缓存**  当服务器内存有限时，如果大量使用缓存键且过期时间设置过长就会导致redis占满内存，而如果担心redis占用内存过大就将缓存键的过期时间设置过短就会导致缓存命中率过低。**实际开发中一般会限制redis的最大内存，并让redis按照一定规则淘汰不需要的键。**  具体设置方法为，修改配置文件的**maxmemory**参数，以限制redis最大可用内存大小（字节），当超过这个限制时，redis会根据**maxmemory-policy**参数指定的策略来删除不需要的键直到redis占用内存小于指定内存大小。 |

## 4.3 排序

问题：在3.5中我们提到，我们是使用集合类型来存储一个标签下所有文章id的，由于集合类型是无序的，所以查看一个标签下的文章列表时，文章不是按照时间排序的。考虑换成有序集合类型，但是zset支持的集合操作又不如set类型强大，比如说实现获取同属于多个标签下的文章列表，由于zset没有zInter命令，只有zInterStore命令，用zInterStore来实现zInter的效果就有点麻烦，伪代码如下。

|  |
| --- |
| MULTI  zInterStore tempKey …  zRange tempKey …  del tempKey  EXEC |

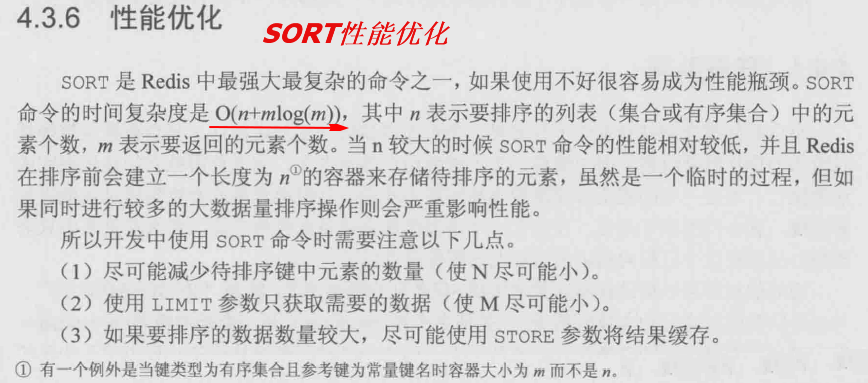
至于为何，zset不提供zInter，zUnion命令，解释如下图：



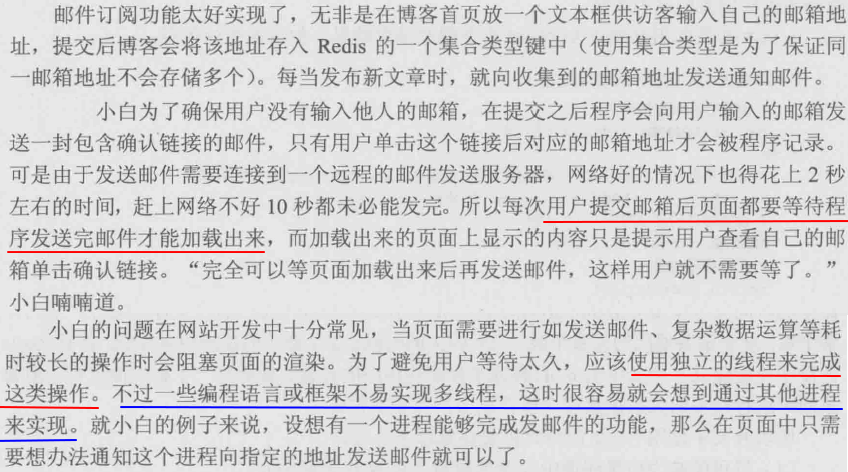
### 4.3.2-5 SORT命令

|  |
| --- |
| **sort key [ALPHA] [DESC] [LIMIT offset count]** |
| 1、sort命令可用于对列表类型键、集合类型键、有序集合类型键进行排序，并且可以完成类似连接查询的功能。  2、在对有序集合类型排序时会忽略元素的分数，只针对元素自身的值进行排序。  3、sort命令默认是对数值元素进行排序的，它会尝试将所有元素转换成double来比较，如果不能转换则报错，通过加**ALPHA参数**可以实现按照字典顺序排列非数字元素。  4、默认是从小到大排序，指定**DESC参数**就是从大到小排序。  5、如果需要分页还可以指定**[LIMIT offset count]参数**。 |
| **BY参数** |
| 格式为**[BY 参考键] ，参考键可以是字符串类型键，或者散列类型键的某个字段(散列键名->字段名)。**  如果提供了BY参数，SORT命令将不再按照元素自身值来进行排序，而是对每个元素依次获取相应参考键的值，然后依据这些参考键的值来进行排序。***对每个元素，通过使用元素的值替换参考键中的第一个”\*”来决定相应的参考键是什么***。  如果**几个元素的对应参考键值相同**，则SORT命令会再比较元素本身的值来决定元素的顺序。当**某个元素对应的参考键不存在**时，会默认参考键的值为0。  若参考键名不包含”\*”（即**常量参考键名**），则所有要比较的值都是一样的，SORT命令将不会执行排序。在不需要排序，但需要借助SORT命令获得与元素相关联的数据时，常量参考键名很有用。  ========================================================================  例如，可通过“sort TAG:java:articleIds BY article:\*->articleTime DESC”来对java标签下的文章按文章发布时间进行排序（这里假设每篇文章都对应了一个散列类型键article:articleId用于存储该文章对象含有的字段，包括发布时间字段articleTime）。  再例如，可通过“sort TAG:java:articleIds BY article:\*: articleTime DESC”来对java标签下的文章按照文章发布时间进行排序（这里假设每篇文章都对应了一个字符串类型键article:articleId:articleTime用于存储该文章的发布时间）; |
| **GET参数** |
| GET参数不影响排序，用于使SORT命令的返回结果不再是元素自身的值，而是GET参数中指定的参数键值。  格式为**[GET 参数键]**，**GET参数的规则和上面介绍的BY参数一样**。要注意：在一个SORT命令中可以有**多个GET参数**，但只能由一个BY参数。  ========================================================================  要实现在排序后直接返回文章标题以及发布时间列表而不是文章id列表，可通过在之前命令的基础上加上GET参数，例如”sort TAG:java:articleIds BY article:\*->articleTime DESC **GET article:\*->articletITLE GET article:\*->articletIME GET #**”，其中”GET #”用于返回元素自身的值。 |
| **STORE参数** |
| 默认情况下，SORT命令会直接返回排序结果，如果**希望保存排序结果**，可以使用STORE参数，格式为**[STORE 目标键名]**，保存后的键的类型为**列表类型**，如果目标键已存在则会被覆盖，**加上STORE参数后SORT命令的返回值**为目标列表类型键中元素个数。例如”sort TAG:java:articleIds BY article:\*->articleTime DESC STORE resultkey”。 |

### 4.3.6 SORT性能分析



## 4.4 消息通知

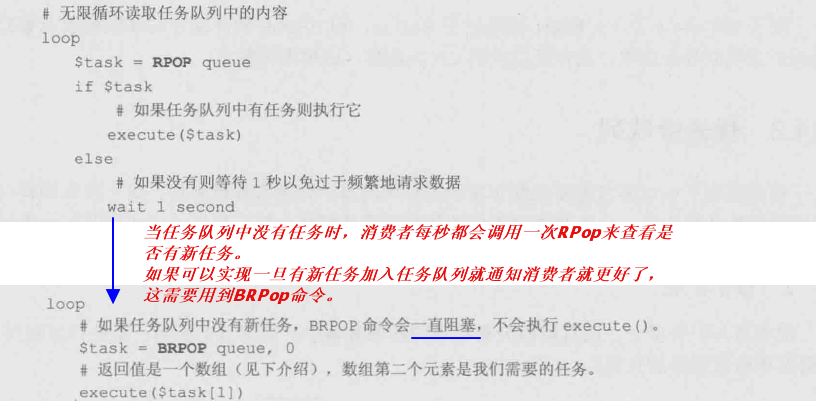


**解决方案**：定义一个任务队列，作为生产者的页面进程负责添加任务到队列中，而作为消费者的邮件发送进程负责不断的从队列中获取任务进行处理。

|  |
| --- |
| **任务队列的好处:** |
| 1. 松耦合。生产者和消费者无需知道彼此的实现细节，只需要约定好任务对象的描述格式。 2. 易于扩展。可增加多个分布在不同服务器上的消费者来降低单台服务器的负载。 |

### 4.4.2 通过redis列表类型实现任务队列

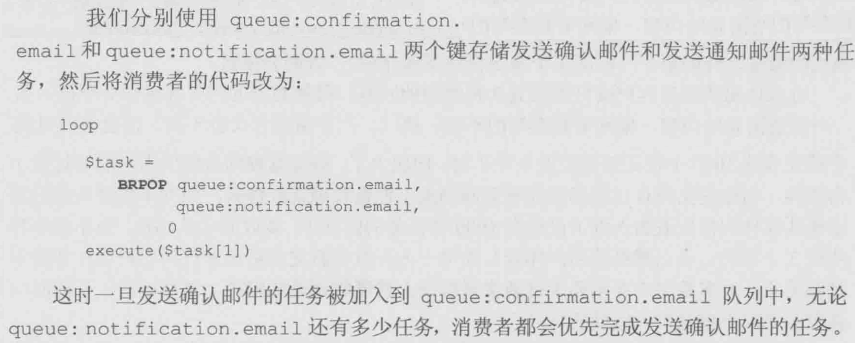
思路：定义一个列表类型键作为任务队列，生产者通过LPush命令添加任务到队列中，消费者不断的使用Rpop命令从队列中取出任务来处理。消费者伪码如下：



|  |  |
| --- | --- |
| BRPop list [list2 …] timeout  BLPop …… | 描述：同时检测多个列表类型键，移出并获取列表的最后一个元素，如果列表中没有元素则会阻塞列表直到等待超时或发现可弹出元素为止；如果多个键都有元素则按照从左到右的顺序取第一个键中的第一个元素。timeout设置为”0”表示不限制等待时间。  返回值：假如在指定时间内没有任何元素被弹出，则返回一个 nil 和等待时长。 反之，返回一个含有两个元素的列表，第一个元素是被弹出元素所属的 key ，第二个元素是被弹出元素的值。 |

### 4.4.3 优先级队列

需求：博客系统存在两种任务，即新邮箱请求订阅时**发送确认邮件的任务**，以及发布新文章后**发送通知邮件给所有已订阅邮箱的任务**。现在需要实现优先级队列，当发送确认邮件任务和发送通知邮件任务同时存在时，优先执行前者。



### 4.4.4 “发布/订阅”模式

|  |
| --- |
| **publish channel msg** |
| 描述：发布者给指定频道发布消息，**发出去的消息不会被持久化**，客户端订阅一个频道后只能收到后续发布到该频道的消息，之前发送的就收不到了。  返回值：接收到这条消息的订阅者数量 |
| **subscribe channel [channel2 …]** |
| 描述：subscribe用于订阅者订阅指定频道，执行subscribe命令后客户端会进入**订阅状态**，处于此状态下的客户端不能使用除了subscribe、unsubscribe、psubscribe、unsubscribe这4个属于”发布/订阅”模式的命令之外的命令，否则会报错。 |
| **unsubscribe [channel channel2 …]** |
| 用于订阅者取消订阅指定频道，如果不指定则会取消订阅所有频道。 |

### 4.4.5 按照规则订阅

|  |
| --- |
| **PSUBSCRIBE pattern [pattern ...]** |
| PSUBSCRIBE命令用于订阅者订阅一个或多个**符合给定模式**的频道。用psubscribe命令进入订阅模式后，如果收到消息，返回值会包含4个值；第一个值为”pmessge”，表示这条消息是通过psubscribe命令订阅频道而收到的；第二个值表示订阅时使用的pattern；第三个值表示实际收到消息的频道名称，第四个值即为消息内容。 |
| **pUNsubscribe [pattern pattern2 …]** |
| PSUBSCRIBE用于退订指定的规则，如果没有指定参数，则会退订所有规则。 |

## 4.5管道

|  |
| --- |
| **往返时延**：redisClient向redis发送命令耗时 + redis向redisClient返回命令执行结果耗时 |
| redis底层对**管道**提供了支持，通过管道可以一次性发送多条命令并在执行完后一次性将结果返回，管道通过减少redisClient与redis的通信次数来实现降低往返时延累计值的目的。当一组命令中所有命令都不依赖于之前命令的执行结果时，就可以将这组命令一起通过管道发出。 |