# 一 事务

## 1 redis事务介绍

与MYSQL一致，Redis中的事务是一组命令的集合，一个事务中的命令要么全部执行，要么都不执行。如果A给B转账，一个命令负责A的金额减少，一个命令负责B的金额增加，两个命令组合为一个事务。

事务的原理是先将属于一个事务的命令发送给redis，然后让redis一次执行。

MULTI 开启事务，返回OK

各种命令 返回值为QUEUED，表示进入队列等待执行。

EXEC 依次执行

如果在发送EXEC命令前客户端断线，那么redis会清空事务队列，所有的命令都不执行，如果发送了EXEC命令，所有命令都会执行，客户端断线也无关，因为redis中已经记录了这些命令。

事务还能保证这些命令在执行的时候不会被别的命令插入干扰。

## 2 事务错误处理

如果有一个命令出现语法错误，如只写了 SET key，没写value，那么EXEC命令后redis就会直接返回错误，语法正确的命令也不执行。（2.65版本前会执行，且返回OK）

如果有一个命令出现运行错误，如使用散列类型命令操作集合类型的键，在执行前redis无法检查到错误，在事务里这样的命令会被redis继续执行，虽然会出现错误，但是后续的正确命令会依然被执行！由于redis不支持回滚..那么需要开发者自己处理。。

## 3 WATCH命令

在事务中只有当所有命令都执行完成后才能得到每个结果的返回值，但是有些情况是需要先获得一个命令的返回值，再根据返回值执行下一条命令：

def incr($key)

$value = GET $key

if not $value

$value=0

$value = $value + 1

SET $key,$value

return $value

如上伪代码，使用GET和SET自己实现incr函数会出现竞态条件，大家可能会想到使用事务实现incr函数防止出现竞态条件，但是因为事务中的每个命令的结果都是最后一期返回的，所以无法将前一条命令的结果作为下一条命令的参数，即在执行SET时，并不知道GET的返回值，也就无法做到+1了。

新思路：

GET获得键值后，保证该键值不会被其他客户端修改，直到函数执行完成后才允许其他客户端修改该键值！！！WATCH可以监控一个或者多个键，一旦有一个键被修改或者删除，之后的事务都不会被执行，监控一直持续到EXEC命令

SET key 1

WATCH key

SET key 2

MULTI

SET key 3

EXEC

GET KEY //得到2

在上述案例中，WATCH后，事务前修改了key值，最后事务的set key 3没有执行，exec返回空结果。

之前案例使用WATCH后的伪代码：

def incr($key)

WATCH $key

$value = GET $key

if not $value

$value=0

$value = $value + 1

MULTI

SET $key,$value

result = EXEC

return result[0]

注意：由于WATCH命令的作用只是当被监控的键被修改后阻止之后的事务执行，但是不能保证其他客户端不修改这一键值，所以需要在EXEC执行失败后重新执行整个函数。

执行EXEC后悔取消对所有键的监控，如果不想执行事务中的命令也可以使用UNWATCH取消监控，比如实现一个hsetxx函数，只有字段存在时才赋值，为了避免竞态条件使用事务来完成：

def hsetxx($key, $field, $value)

WATCH $key

$isFiledExists = HEXISTS $key, $field

if $isFiledExists is 1

MULTI

HSET $key, $field, $value

EXEC

else

UNWATCH

return $isFiledExists

这样做其实已经很麻烦了，我们希望redis能够提供一个命令来实现访问频率限制，并通过一系列参数来设定这个命令，但是redis并没有这样的命令，可以通过编写lua脚本来实现。lua脚本见lua文档。

lua脚本限制访问代码：

local times = redis.call(‘incr’, KEYS[1])

if times = 1 then

redis.call(‘expire’,KEYS[1],ARGV[1]);

end

if times > tonumber(ARGV[2]) then

return 0

end

return 1

将该代码保存为 test.lua，在命令行输入：

redis-cli --eval /path/test.lua rate.limiting:127.0.0.1, 10 3

第一个参数是要操作的键，在脚本中KEYS[1]获取，第二个参数10 3 通过ARGV[1]和ARGV[2]获取。

# 二 过期与缓存

在redis中使用expire命令设置一个键的过期时间，到时间后redis会自动删除。

如：设置session:29e3d键在15分钟后被删除

EXPIRE session:29e3d 900 //返回1则设置成功

TTL session:29e3d //查询该键还要多久会被删除，键不存在返回-2

//没有设置过期时间返回-1，2.6版本中键不存在返回-1

PERSIST //取消过期设置，即设置为永久

当然SET GETSET为键赋值也会同时清除过期时间

注意：

1 EXPIRE参数必须是是正数，所以最小单位是秒。PERSIST的时间单位是毫秒，PTTL用于返回该命令设置的键剩余毫秒数。

2 如果WATCH监测一个设置了过期时间的键，那么键过期后，不会被WATCH视为改变。

3 命令EXPIREAT也可以设置过期时间（单位为毫秒），但是参数使用UNIX时间

实战：经常会遇到限制一个用户在1分钟内可访问的次数，用户每次访问，我们为用户的键，值+1，在1分钟内超过10次，则表明超出限制，提示用户稍后访问，每分钟自动删除，到了下一分钟访问次数重新计算。

这里会遇到2个问题：

问题1：当为用户增加访问此次数到10时，在设置过期时间前程序退出了，那么该键无过期时间，用户永远无法访问了，所以需要事务的控制这2步；

问题2：现在设定用户每分钟只能访问10次，如果一个用户在一分钟内的第一秒访问了1次，最后1秒访问了9次，又在下一分钟的第一秒访问10次，那么该用户在2秒内访问了19次....

所以最好的办法是：使用列表类型的键记录最近10次访问时间，一旦键中的元素超过10个，就判断最早的元素距离现在的时间是否小于1分钟，如果是，则表示用户最近1分钟访问次数超过了10次。

缓存：

为了提高网站负载能力，常常将访问频率高但是对CPU或者IO消耗大的操作结果缓存起来，并希望缓存会自动过期。每次访问数据时，先检查缓存键是否存在，如果存在则直接使用缓存的值，否则重新获取数据并将结果缓存。

但是当服务器内存有限时，大量使用缓存，且过期时间过长则会导致redis占满内存，另一方便如果为了防止redis占用太高，将过期时间缩短，又会造成命中率太低而闲置内存。

为此可以设置一定规则淘汰不需要的缓存键，这种方式在redis做缓存系统时非常实用：

修改配置文件的maxmemory参数，闲置redis最大可用内存大小，单位是字节。超过这个限制时，redis会依据maxmemory-policy参数制定的策略删除不需要的键直到占用内存小于指定内存。

常用规则有：

volatile-lru LRU算法删除一个设置了过期的键

allkeys-lru LRU算法删除一个键

volatile-random 随机删除一个过期键

allkeys-random 随机删除一个键

volatile-ttl 删除过期时间最近的键

noeviction 不删除，只返回错误。

# 三 排序

## 1 有序集合

有序集合常见的应用场景是排序，如游戏玩家的排行榜，所以很少会需要键中的全部数据，redis认为在做完交集、并集运算后不需要直接获取全部结果，而是会希望将结果存入新的键中。所以有序集合只有ZINTERSTORE和ZUNIONSTORE，没有ZINTER和ZUINON的原因。

如果要直接获得结果，使用流程如下：

MULTI

ZINTERSTORE tempkey

ZRANGE tempkey

DEL tempkey

EXEC

## 2 SORT LIMIT

sort可以对集合、列表类型的键进行排序：

SORT tag:ruby:posts

在对有序集合排序时会忽略元素的分数，只针对元素自身的值进行排序。

ALPHA参数：

如果存储的数据时字符，参数 ALPHA可以实现按照字典顺序排列飞数字元素，不带该参数则sort命令会将所有元素转换成双精度浮点数来比较，不能转换则提示错误。

LIMIT参数：

与SQL一样，sort支持limit

SORT myzset DESC LIMIT 1 2

BY参数：

SORT tag:ruby:posts BY post:\*->time DESC

BY的参考键可以是字符串类型键或者散列类型键的某个字段，有BY参数，那么SORT命令不再依据元素自身的值排序，而是对每个元素的值替换参考件的第一个\*并获取值，然后排序。上述案例SORT会依次读取获取到的所有键中time字段的值来决定排序，若参考键值相同，则依据自身值决定排序。

没有 \* 时，则sort命令不会执行排序！！

当某个元素的参考件不存在，则默认使用0

GET参数：

GET不影响排序，他会使SORT返回的结果不再是元素自身的值，而是GET参数中的制定的键值。GET的用法与BY一样，支持字符串和散列类型的键，使用\*作为占位符。

SORT tag:ruby:posts BY post:\*->time DESC GET post:\*->title

注意：在一个SORT中，BY只能有一个，GET可以有多个。

GET # 会返回元素本身

STORE参数：

如果希望SORT结果保存排序结果，则语句后可以添加 STORE sort.result保存在键 sort.result中。保存后的键的类型为列表类型，如果键已经存在则覆盖，且返回值为结果的个数。

优化：

SORT的时间复杂度是O(n+mlog(m)) n表示要排序的列表（即集合和有序集合）中的个数，m表示要返回的元素个数。n越大性能越低，且在排序前，redis会创建一个长度为n的临时容器来存储待排序的元素，会严重影响性能，所以需要：

1 尽可能减少待排序键中元素的数量

2 使用LIMIT参数只获取需要的数据（减少m值）

3 排序数量较大，尽可能使用STORE参数将结果缓存

# 四 消息通知

## 1 任务队列

当页面需要发送如邮件、复杂数据运算等耗时操作时会阻塞页面渲染，为了避免用户等待太久，往往会开启一个独立进程来完成这类操作，在页面中只要想办法通知这个进程向指定的地址发送邮件即可。

通知的过程可以借助任务对垒实现，即传递任务的队列。

生产者将需要处理的任务加入任务队列，消费者不断从任务队列中读入任务并执行。

对于发邮件操作来说：页面程序就是生产者，发邮件的进程是消费者。当需要发送邮件时，页面程序会将收件地址、邮件内容等组成一个任务后存入任务队列中，同时发邮件的进程会不断检查任务队列，一旦发现有新的任务便会将其从对垒中取出并执行，由此实现了进程间的通信。

使用任务队列的好处：

1. 低耦合：生产者和消费者无需知道彼此实现细节
2. 易扩展：消费者可以有多个，可以分布在不同的服务器中

## 2 Redis实现任务队列

使用Redis的列表类型，生产者使用LPUSH将任务加入某个键，消费者不断使用RPOP从该键中取出任务即可。发邮件的伪代码如下：

loop //无线循环读取任务队列内容

$task = RPOP queue

if $task

execute($task) //如果任务队列中有任务则执行该任务

else

wait 1 second //没有任务则等待1秒后继续查询

上述案例并不完美，因为没有任务的时候，消费者仍然在循环读取队列数据，使用BRPOP命令，在没有元素时，可以阻塞连接，直到有新元素加入，上述伪代码可以改写为：

loop

$task = BRPOP queue,0

execute($task[1]);

BRPOP命令会接收2个参数，第一个是键名，第二个是超时时间，单位是秒，超过了超时时间，仍然没有获得新元素，则返回nil，案例中超时时间为0，即不限制等待事假，即如果没有新元素加入列表就会永远阻塞下去。当获得一个元素后，该命令分别返回键名和元素值。

同理，BLPOP会从左边取出元素。

演示：

客户端1 客户端2

BRPOP queque 0 LPUSH queque task

> 1

queue

task

LLEN queue //实例中的元素已经被取走

> 0

## 3 优先级队列

博客在发布文章的时候回向每个订阅者发送邮件，这个步骤可以使用任务队列实现，由于要执行的任务和发送确认邮件一样，所以二者可以使用同一个消费者。但是这时候就会带来问题：

如果有1000个用户订阅，发布一篇文章后向1000个用户发送推送邮件，每次发送消耗10秒，那么这1000个任务要消耗近3个小时，在此期间，如果有个用户想注册账号，同样也会发送邮件，该用户需要等待3个小时才能收到邮件。。。

所以当发送确认邮件、发送通知邮件两种任务同时存在时，应该优先执行前者。

BRPOP可以同时接收多个键，完整格式为：

BLPOP key key key.... timeout

ReadisA RedisB

BLPOP queue:1 queue2 queue:3 0 LPUSH queue:2 task

> 1

queueL2

task

如果多个键都有元素则按照从左到右的顺序取出第一个键中的第一个元素，现在queue:2和queue:3中个加入一个元素：

ReadisA RedisB

LPUSH queue:2 task1 LPUSH queue:3 task2

>1 >1

执行：

BRPOP queque:1 queue:2 queue:3 0

输出 queue:2 task1

## 4 发布订阅模式

除了任务队列，Redis提供了发布订阅模式，同样可以实现进程间传递消息：

发布者可以订阅一个或多个频道，而发布者可以向指定频道发送消息，所有订阅此频道的订阅者都会收到消息。

发布：

PUBLISH channel1.1 hi

1. // 返回值表示接受这小消息订阅者的数量

发布的消息不会被持久化，客户只能收到订阅之后产生的消息。

订阅：

SUBSCRICE channel1.1 channel1.2... //可以订阅多个

执行订阅后，客户端处于订阅状态，此时客户端不能再使用除了发布订阅相关命令外的其他命令，否则会报错。

进入订阅的客户端可能会受到3种类型的回复，每种类型的回复都包含3个值，第一个值时消息的类型，根据消息类型的不同，第二个 三个值的含义不同。

消息类型可能是：

1 subscribe 表示订阅成功反馈的信息，第二个值时订阅成功的频道名称，第三个值是当前客户端订阅的数量。

2 message 表示接收到的消息，第二个值是产生消息的频道，第三个值是消息的内容

3 unsubscribe 表示成功取消订阅某个频道。第二个值是对应的频道名称，第三个值是当前客户端订阅的频道数量，当此值为0时，退出订阅模式。

UNSUBSCRIBE可以取消订阅频道(支持多个)

PSUBSCRIBE命令可以指定订阅的规则：

PSUBSCRIBE channel1.?\* //支持匹配到1.1 1.10 但是不会匹配 1.

注意：PSUBSCRIBE可以重复订阅同1个频道，也会接收到该频道的重复消息；

使用PUNSUBSCRIBE只可以退订通过PSUBSCRIBE订阅的规则。

注意：由于严格的字符串匹配 PUNSUBSCRIBE \* 无法退订 channel.\*规则，必须使用 PUNSUBSCRIBE channel.\*

# 五 管道

Redis的命令需要经过网络传输，发送接收总会有网络延迟，称为往返延时，大致来说本地回环地址的往返延时在数量级上相当于Redis处理一条简单命令的时间，但是命令较多会造成大量延时。

在执行多个命令时，都需要等待上一条命令执行完成，即使不需要上一条命令的执行结果。Redis底层的通信协议对管道提供了支持，通过管道可以一次性发送多条命令并在执行完毕后一次性将结果返回。当一组命令中每条命令都不依赖于之前的命令的执行结果时，就可以将这组命令一起通过管道发出。

管道通过减少客户端与Redis之间的通信次数，降低往返延时。

# 六 优化内存

## 1 精简键名和键值

## 2 内部编码优化

Redis为每种数据类型都提供了两种内部编码方式，例如散列类型是通过散列表实现的，这样就可以实现O（l）时间复杂度的查找、赋值操作，然而键中元素很少的时候，O（l）并不比O（n）有明显的性能提高，所以这种情况下，Redis采用更为紧凑但是性能稍差的O（n）编码方式。当键中元素变多时Redis会自动将编码方式调整为散列表。

OBJECT ENCODING foo //查看foo键的编码

Redis的每个键值都是一个redisObject结构体，定义如下：

typedef struct redisObject {

unsigned type:4;

unsigned notused:2 //not used

unsigned encoding:4;

unsigned lru: 22;

int refcount;

void \*ptr

} robj;

其中type字段表示的是键值的数据类型，取值可以是：

#define REDIS\_STRING 0

#define REDIS\_LIST 1

#define REDIS\_SET 2

#define REDIS\_ZSET 3

#define REDIS\_HASH 4

encoding 字段表示redis编码方式：

#define REDIS\_ENCODING\_RAW 0

#define REDIS\_ENCODING\_INT 1

#define REDIS\_ENCODING\_HT 2

#define REDIS\_ENCODING\_ZIPMAP 3

#define REDIS\_ENCODING\_LINKEDLIST 4

#define REDIS\_ENCODING\_ZIPLIST 5

#define REDIS\_ENCODING\_INSET 6

#define REDIS\_ENCODING\_SKIPLIST 7

#define REDIS\_ENCODING\_EMBSTR 8

各个数据类型可能采用内部编码方式以及相应的OBJECT ENCODING命令执行查看其结果。