**Redis的事务功能详解**

MULTI、EXEC、DISCARD和WATCH命令是Redis事务功能的基础。

Redis事务允许在一次单独的步骤中执行一组命令，并且可以保证如下两个重要事项：

>**Redis会将一个事务中的所有命令序列化，然后按顺序执行。**Redis不可能在一个Redis事务的执行过程中插入执行另一个客户端发出的请求。这样便能保证Redis将这些命令作为**一个单独的隔离操作执行**。 > 在一个Redis事务中，Redis要么执行其中的所有命令，要么什么都不执行。因此，**Redis事务能够保证原子性**。

**EXEC命令会触发执行事务中的所有命令**。因此，当某个客户端正在执行一次事务时，**如果它在调用EXEC命令之前就从Redis服务端断开连接，那么就不会执行事务中的任何操作；相反，如果它在调用EXEC命令之后才从Redis服务端断开连接，那么就会执行事务中的所有操作。（有出错也不会停止了，后面解释）**

当Redis使用只增文件（AOF：Append-only File）时，Redis能够确保使用一个单独的write(2)系统调用，这样便能将事务写入磁盘。然而，如果Redis服务器宕机，或者系统管理员以某种方式停止Redis服务进程的运行，那么Redis很有可能只执行了事务中的一部分操作。Redis将会在重新启动时检查上述状态，然后退出运行，并且输出报错信息。使用redis-check-aof工具可以修复上述的只增文件，这个工具将会从上述文件中删除执行不完全的事务，这样Redis服务器才能再次启动。

从2.2版本开始，除了上述两项保证之外，Redis还能够以乐观锁的形式提供更多的保证，这种形式非常类似于“检查再设置”（CAS：Check And Set）操作。本文稍后会对Redis的乐观锁进行描述。

**一、相关命令**

**1. MULTI**

该命令用来开启事务，它总是返回ok结果，当其执行之后，客户端可以继续发送任意条数量的指令，**这些指令不会立即被执行，而是被放到了队列中**，直到**EXEC被调用**之后，所有命令**才会被序列化执行**。

**2. EXEC**

在一个事务中执行所有先前放入队列的命令，然后恢复正常的连接状态。

当使用WATCH命令时，只有当受监控的键没有被修改时，EXEC命令才会执行事务中的命令，这种方式利用了**检查再设置（CAS）的机制**。

这个命令的运行格式如下所示：

EXEC

这个命令的返回值是一个数组，其中的每个元素分别是原子化事务中的每个命令的返回值。 当使用WATCH命令时，如果事务执行中止，那么EXEC命令就会返回一个Null值。

MULTI开启之后，因为某些原因没有成功执行EXEC，那么事务中所有的命令都不会被执行的。

**3. DISCARD**

**清除所有先前在一个事务中放入队列的命令**，然后恢复正常的连接状态。

如果使用了WATCH命令，那么DISCARD命令就会将当前连接监控的所有键取消监控。

这个命令的运行格式如下所示：

DISCARD

这个命令的返回值是一个简单的字符串，总是OK。

**4. WATCH**

**当某个事务需要按条件执行时，就要使用这个命令将给定的键设置为受监控的**。

这个命令的运行格式如下所示：

WATCH key [key ...]

这个命令的返回值是一个简单的字符串，总是OK。

对于每个键来说，时间复杂度总是O(1)。

NOTE:

A、WATCH使得EXEC命令需要有条件的执行，也就是事务只能在所有被监视的键没有被修改的前提下才能执行。另外，在EXEC被执行之后，所有的WATCH都会被取消。

**5. UNWATCH**

**清除所有先前为一个事务监控的键。**

如果你调用了EXEC或DISCARD命令，那么就不需要手动调用UNWATCH命令。

这个命令的运行格式如下所示：

UNWATCH

这个命令的返回值是一个简单的字符串，总是OK。

时间复杂度总是O(1)。

**二、使用方法及事务内部的错误示范**

使用MULTI命令便可以进入一个Redis事务。这个命令的返回值总是OK。此时，用户可以发出多个Redis命令。Redis会将这些命令放入队列，而不是执行这些命令。一旦调用EXEC命令，那么Redis就会执行事务中的所有命令。

Redis原生使用(Redis-cli)：

127.0.0.1:6379> multi     // 事务开始的动作标志下面即为入队

OK

127.0.0.1:6379> set book-name "Thinking in Java"

QUEUED

127.0.0.1:6379> get book-name

QUEUED

127.0.0.1:6379> sadd tag "java" "Programming""Thinking"

QUEUED

127.0.0.1:6379> smembers tag

QUEUED

127.0.0.1:6379> exec     // 执行事务

1) OK

2) "Thinking in Java"

3) (integer) 3

4) 1) "Thinking"

   2) "Programming"

   3) "java"

127.0.0.1:6379> discard  // 事务已执行完毕 已经自动取消

(error) ERR DISCARD without MULTI

127.0.0.1:6379> multi

OK

127.0.0.1:6379> set book-name "Patterns in Java"

QUEUED

127.0.0.1:6379> get book-name

QUEUED

127.0.0.1:6379> sadd tag "Java" "Thinking""Programming"

QUEUED

127.0.0.1:6379> smembers tag

QUEUED

127.0.0.1:6379> discard  // 事务未执行 可以刷新队列指令状态 取消执行

OK

127.0.0.1:6379> exec     // 事务已经被取消不能再执行

(error) ERR EXEC without MULTI

**四、为什么Redis不支持回滚？**

如果你具备关系型数据库的知识背景，你就会发现一个事实：**在事务运行期间，虽然Redis命令可能会执行失败，但是Redis仍然会执行事务中余下的其他命令，而不会执行回滚操作，你可能会觉得这种行为很奇怪。**

然而，这种行为也有其合理之处：

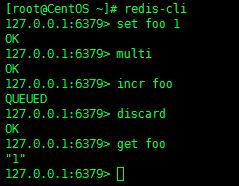
只有当被调用的Redis命令有语法错误时，这条命令才会执行失败（**在将这个命令放入事务队列期间，Redis能够发现此类问题,事务就失败了**），或者对某个键执行不符合其数据类型的操作：实际上，这就意味着**只有程序错误才会导致Redis命令执行失败**，这种错误很有可能在程序开发期间发现，一般很少在生产环境发现。**（语法错误的意思吧）**  
 Redis已经在系统内部进行功能简化，这样可以确保更快的运行速度，因为Redis不需要事务回滚的能力。

对于Redis事务的这种行为，有一个普遍的反对观点，那就是程序有可能会有缺陷（bug）。但是，你应当注意到：**事务回滚并不能解决任何程序错误。**例如，如果某个查询会将一个键的值递增2，而不是1，或者递增错误的键，那么事务回滚机制是没有办法解决这些程序问题的。请注意，没有人能解决程序员自己的错误，这种错误可能会导致Redis命令执行失败。正因为这些程序错误不大可能会进入生产环境，所以我们在开发Redis时选用更加简单和快速的方法，没有实现错误回滚的功能。

**鉴于没有任何机制能避免程序员自己造成的错误， 并且这类错误通常不会在生产环境中出现， 所以 Redis 选择了更简单、更快速的无回滚方式来处理事务。**

**五、丢弃命令队列**

DISCARD命令可以用来中止事务运行。在这种情况下，不会执行事务中的任何命令，并且会将Redis连接恢复为正常状态。示例如下所示：



**六、通过CAS操作实现乐观锁**

1、乐观锁实现

举个例子，假设我们需要原子性为某个键加1操作（假设INCR不存在），那么应该是这样的执行语句：

SET mykey 1

val = GET mykey

val = val + 1

SET mykey ${val}

单个客户端访问操作没有任何问题，如果是多个客户端同时访问mykey，就会产生资源共享访问问题，比如：现在有个两个客户端访问同一个键mykey，那么mykey的可能是2，但是我们期望的值应该是3才对，这个类似于高并发下的sync锁机制，所以我们需要**使用WATCH来监控被共享的键mykey**，如下：

**WATCH mykey（可监控多个键）**

val = GET mykey

val = val + 1

MULTI

SET mykey ${val}

EXEC

NOTE：

虽然大多情况下，多个客户端访问操作同一个键的情况很少或没有，但是不能排除这个特殊情况，所以建议在有可能产生键共享的指令中使用**WATCH**（有点类似JAVA中的**synchronzied**）在**EXEC执行前对其监管。（就是执行过程前，取得这个watch的锁才能继续执行）**

2、Redis不支持回滚(Roll Back)

Redis的事务不支持回滚，这点不同于关系数据库中的事务，所以它的内部保持了简单且快速的特点。另外，Redis不支持回滚是这样考虑的：Redis事务中命令之所以会失败，是由于错误的编程所造成，通过事务回滚是不能回避这个根本问题。

NOTE：

Redis事务中命令执行失败，仍会继续执行后面的执行，在没有特殊干预前提下，直到执行完队列中所有指令为止。

3、使用事务可能遇到的问题

A、事务在执行 [EXEC](http://www.redis.cn/commands/exec.html) 之前，入队的命令可能会出错，举个例子：命令可能会产生语法错误（参数数量错误，参数名错误等），或者其他更严重的错误，比如内存不足（如果服务器使用maxmemory 设置了最大内存限制的话）。

B、事务在执行 [EXEC](http://www.redis.cn/commands/exec.html) 之前，举个例子：事务中的命令可能处理了错误类型的键，比如将列表命令用在了字符串键上面等。

对于发生在 [EXEC](http://www.redis.cn/commands/exec.html) 执行之前的错误，客户端以前的做法是检查命令入队所得的返回值：如果命令入队时返回QUEUED ，那么入队成功；否则，就是入队失败。**如果有命令在入队时失败，那么大部分客户端都会停止并取消这个事务。**

从 Redis 2.6.5 开始，服务器会对命令入队失败的情况进行记录，并在客户端调用 [EXEC](http://www.redis.cn/commands/exec.html) 命令时，**拒绝执行并自动放弃这个事务。**

在 Redis 2.6.5 以前， Redis 只执行事务中那些入队成功的命令，而忽略那些入队失败的命令。

**而新的处理方式则使得在管道技术中包含事务变得简单，因为发送事务和读取事务的回复都只需要和服务器进行一次通讯即可**。

 至于那些在 [EXEC](http://www.redis.cn/commands/exec.html) 命令执行之后所产生的错误，并没有对它们进行特别处理： 即使事务中有某个/某些命令在执行时产生了错误， 事务中的其他命令仍然会继续执行。

**七、WATCH命令详解**

那么WATCH命令实际做了些什么呢？

**这个命令会使得EXEC命令在满足某些条件时才会运行事务：(CAS的意思，**我们要求Redis只有在所有受监控的键都没有被修改时，才会执行事务。也就是说在别的线程里改了这个键，那这个事务就不能执行了**)**

（但是，相同的客户端可能会在事务内部修改这些键，此时这个事务不会中止运行。内部修改的没关系）否则，Redis根本就不会进入事务。（**注意，如果你使用WATCH命令监控一个易失性的键，然后在你监控这个键之后，Redis再使这个键过期，那么EXEC命令仍然可以正常工作。**）

WATCH命令可以被调用多次。简单说来，**所有的WATCH命令都会在被调用之时立刻对相应的键进行监控，直到EXEC命令被调用之时为止**。你可以在单条的WATCH命令之中，使用任意数量的键作为命令参数。

**当调用EXEC命令时，所有的键都会变为未受监控的状态，Redis不会管事务是否被中止。当一个客户单连接被关闭时，所有的键也都会变为未受监控的状态。（就是调用EXEC前，键都是受到WATCH监控，调用后就自动释放监控了）。**

你还可以使用UNWATCH命令（不需要任何参数），这样便能清除所有的受监控键。当我们对某些键施加乐观锁之后，这个命令有时会非常有用。因为，我们可能需要运行一个用来修改这些键的事务，但是在读取这些键的当前内容之后，我们可能不打算继续进行操作，此时便可以使用UNWATCH命令，清除所有受监控的键。在运行UNWATCH命令之后，Redis连接便可以再次自由地用于运行新事务。

如何使用WATCH命令实现ZPOP操作呢？

本文将通过一个示例，说明如何使用WATCH命令创建一个新的原子化操作（Redis并不原生支持这个原子化操作），此处会以实现ZPOP操作为例。这个命令会以一种原子化的方式，从一个有序集合中弹出分数最低的元素。以下源码是最简单的实现方式：

WATCH zset

element = ZRANGE zset 0 0

MULTI

ZREM zset element

EXEC

如果伪码中的EXEC命令执行失败（例如，返回Null值），那么我们只需要重复运行这个操作即可。

**八、Redis脚本和事务**

根据定义，Redis脚本也是事务型的。因此，你可以通过Redis事务实现的功能，同样也可以通过**Redis脚本来实现，而且通常脚本更简单、更快速。**

由于Redis从2.6版本才开始引入脚本特性，而事务特性是很久以前就已经存在的，所以目前的版本才有两个看起来重复的特性。但是，我们不太可能在短时间内移除对事务特性的支持。因为，即使不用求助于Redis脚本，用户仍然能够规避竞争状态，这从语义上来看是适宜的。还有另一个更重要的原因，Redis事务特性的实现复杂度是最小的。

但是，在相当长的一段时间之内，我们不大可能看到整个用户群体都只使用Redis脚本。如果发生这种情况，那么我们可能会废弃，甚至最终移除Redis事务。

脚**本将在下一章节和管道一起描述。**