**1. Redis事务定义**

Redis中的事务（transaction）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，一个事务中的命令要么都执行，要么都不执行。  
事务的原理是先将属于一个事务的命令发送给Redis，然后再让Redis依次执行这些命令。

1. Redis保证一个事务中的所有命令要么都执行，要么都不执行。如果在发送EXEC命令前客户端断线了，则Redis会清空事务队列，事务中的所有命令都不会执行。而一旦客户端发送了EXEC命令，所有的命令就都会被执行，即使此后客户端断线也没关系，因为Redis中已经记录了所有要执行的命令。
2. 除此之外，Redis的事务还能保证一个事务内的命令依次执行而不被其他命令插入。试想客户端A需要执行几条命令，同时客户端B发送了一条命令，如果不使用事务，则客户端B的命令可能会插入到客户端A的几条命令中执行。如果不希望发生这种情况，也可以使用事务。

**2. Redis事务的错误和回滚**

Redis的事务没有隔离级别的概念（CAID中的I），在事务执行前所有的命令都未执行。对于执行过程中的错误按照类型分为两种。

**1. 语法错误**

语法错误指命令不存在或者命令参数的个数不对，这个命令可能会有语法错误（参数的数量错误、命令名称错误，等等），或者可能会有某些临界条件（例如：如果使用maxmemory指令，为Redis服务器配置内存限制，那么就可能会有内存溢出条件）。

可用Redis客户端检测第一种类型的错误，在调用EXEC命令之前，这些客户端可以检查被放入队列的命令的返回值：如果命令的返回值是**QUEUE**字符串，那么就表示已经正确地将这个命令放入队列；否则，Redis将返回一个错误。如果将某个命令放入队列时发生错误，那么大多数客户端将会**中止事务**，并且丢弃这个事务。  
在Redis 2.6.5版本之前，如果发生了上述的错误，那么在客户端调用了EXEC命令之后，Redis还是会运行这个出错的事务，执行已经成功放入事务队列的命令，而不会关心先前发生的错误。从2.6.5版本开始，Redis在遭遇上述错误时，服务器会记住事务积累命令期间发生的错误。然后，Redis会拒绝执行这个事务，在运行EXEC命令之后，便会返回一个错误消息。最后，Redis会自动丢弃这个事务，这样便能轻松地**混合使用事务和管道**。在这种情况下，客户端可以一次性地将整个事务发送至Redis服务器，稍后再一次性地读取所有的返回值。

**2. 运行错误**

运行错误指在命令执行时出现的错误，比如使用散列类型的命令操作集合类型的键，这种错误在实际执行之前Redis是无法发现的，所以在事务里这样的命令是会被Redis接受并执行的。如果事务里的一条命令出现了运行错误，事务里**其他的命令依然会继续执行（包括出错命令之后的命令）**。

**Redis的回滚机制**

Redis的事务没有关系数据库事务提供的**回滚（rollback）功能**。为此开发者必须在事务执行出错后自己收拾剩下的摊子（将数据库复原回事务执行前的状态等,这里我们一般采取**日志记录**然后**业务补偿**的方式来处理，但是一般情况下，在redis做的操作不应该有这种强一致性要求的需求，我们认为这种需求为不合理的设计）。

**3. Redis的乐观锁和Watch**

**Watch命令描述：**  
WATCH命令可以监控**一个或多个键**，一旦其中有一个键被修改（或删除），之后的事务就不会执行。监控一直持续到**EXEC**命令（事务中的命令是在EXEC之后才执行的，所以在MULTI命令后可以修改WATCH监控的键值）

在Redis的事务中，WATCH命令可用于提供**CAS**(check-and-set)功能，且是基于**乐观锁**的思想。假设我们通过WATCH命令在事务执行之前监控了多个Keys，倘若在WATCH之后有任何Key的值发生了变化，**EXEC命令执行的事务都将被放弃**，同时返回**Null multi-bulk**应答以通知调用者事务执行失败。例如，我们再次假设Redis中并未提供incr命令来完成键值的原子性递增，如果要实现该功能，我们只能自行编写相应的代码。其伪码如下：

**[java]** [view plain](https://blog.csdn.net/KingCat666/article/details/77942830) [copy](https://blog.csdn.net/KingCat666/article/details/77942830)

1. val = GET mykey
2. val = val + 1
3. SET mykey $val

以上代码只有在单连接的情况下才可以保证执行结果是正确的，因为如果在同一时刻有多个客户端在同时执行该段代码，那么就会出现多线程程序中经常出现的一种错误场景--竞态争用(race condition)。比如，客户端A和B都在同一时刻读取了mykey的原有值，假设该值为10，此后两个客户端又均将该值加一后set回Redis服务器，这样就会导致mykey的结果为11，而不是我们认为的12。为了解决类似的问题，我们需要借助WATCH命令的帮助，见如下代码：

**[java]** [view plain](https://blog.csdn.net/KingCat666/article/details/77942830) [copy](https://blog.csdn.net/KingCat666/article/details/77942830)

1. WATCH mykey
2. val = GET mykey
3. val = val + 1
4. MULTI
5. SET mykey $val
6. EXEC

和此前代码不同的是，新代码在获取mykey的值之前先通过WATCH命令监控了该键，此后又将set命令包围在事务中，这样就可以有效的保证每个连接在执行EXEC之前，如果当前连接获取的mykey的值被其它连接的客户端修改，那么当前连接的EXEC命令将执行失败。这样调用者在判断返回值后就可以获悉val是否被重新设置成功。

由于WATCH命令的作用只是**当被监控的键值被修改后阻止之后一个事务的执行**，而不能保证其他客户端不修改这一键值，所以在一般的情况下我们需要在**EXEC执行失败后重新执行整个函数。**执行EXEC命令后会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命令来取消监控。

**实现一个hsetNX函数**  
我们实现的hsetNX这个功能是：仅当字段存在时才赋值。为了避免竞态条件我们使用watch和事务来完成这一功能（伪代码）：

**[java]** [view plain](https://blog.csdn.net/KingCat666/article/details/77942830) [copy](https://blog.csdn.net/KingCat666/article/details/77942830)

1. WATCH key
2. isFieldExists = HEXISTS key, field
3. **if** isFieldExists is 1
4. MULTI
5. HSET key, field, value
6. EXEC
7. **else**
8. UNWATCH
9. **return** isFieldExists

在代码中会**判断要赋值的字段是否存在，如果字段不存在的话就不执行事务中的命令**，但需要使用**UNWATCH**命令来保证下一个事务的执行不会受到影响。

4. Jedis的事务

Jedis对Redis的事务机制给出了具体实现，示例代码如下：

**[java]** [view plain](https://blog.csdn.net/KingCat666/article/details/77942830) [copy](https://blog.csdn.net/KingCat666/article/details/77942830)

1. **public** **static** **void** testWach(){
2. Jedis jedis = RedisCacheClient.getInstrance().getClient();
3. String watch = jedis.watch("testabcd");
4. System.out.println(Thread.currentThread().getName()+"--"+watch);
5. Transaction multi = jedis.multi();
6. multi.set("testabcd", "23432");
7. **try** {
8. Thread.sleep(3000);
9. } **catch** (InterruptedException e) {
10. e.printStackTrace();
11. }
12. List<Object> exec = multi.exec();
13. System.out.println("---"+exec);
14. jedis.unwatch();
15. }
16. **public** **static** **void** testWatch2(){
17. Jedis jedis = RedisCacheClient.getInstrance().getClient();
18. String watch = jedis.watch("testabcd2");
19. System.out.println(Thread.currentThread().getName()+"--"+watch);
20. Transaction multi = jedis.multi();
21. multi.set("testabcd", "125");
22. List<Object> exec = multi.exec();
23. System.out.println("--->>"+exec);
24. }

**三 Redis的管道**

Redis是一个响应式的服务，当客户端发送一个请求后，就处于**阻塞状态**等待Redis返回结果。这样一次命令消耗的时间就包括三个部分：**请求从客户端到服务器的时间**、**结果从服务器到客户端的时间**和**命令真正执行时**间，前两个部分消耗的时间总和称为**RTT**（Round Trip Time），当客户端与服务器存在网络延时，RTT就可能会很大，这样就会导致性能问题。

**管道（Pipeline）就是为了改善这个情况的**，利用管道，客户端可以**一次性发送多个请求而不用等待服务器的响应，待所有命令都发送完后再一次性读取服务的响应，这样可以极大的降低RTT时间从而提升性能**。需要注意到是用**pipeline方式打包命令发送**，**redis必须在处理完所有命令前先缓存起所有命令的处理结果。**打包的命令越多，缓存消耗内存也越多。所以并不是打包的命令越多越好。

pipeline和“事务”是两个完全不同的概念，pipeline只是表达“交互”中操作的传递的方向性，pipeline也**可以在事务中运行**，也可以不在。无论如何，pipeline中发送的每个command都会被server立即执行，如果执行失败，将会在此后的相应中得到信息；**也就是pipeline并不是表达“所有command都一起成功”的语义(有的成功有的失败)**；但是如果pipeline的操作被封装在事务中，那么将有事务来确保操作的成功与失败**(只允许成功或者失败)**。Pipeline的示例代码如下：

**[java]** [view plain](https://blog.csdn.net/KingCat666/article/details/77942830) [copy](https://blog.csdn.net/KingCat666/article/details/77942830)

1. **private** **static** **void** usePipeline(**int** count){
2. Jedis jr = **null**;
3. **try** {
4. jr = **new** Jedis("10.10.224.44", 6379);
5. Pipeline pl = jr.pipelined();
6. **for**(**int** i =0; i<count; i++){
7. pl.incr("testKey2");
8. }
9. pl.sync();
10. } **catch** (Exception e) {
11. e.printStackTrace();
12. }
13. **finally**{
14. **if**(jr!=**null**){
15. jr.disconnect();
16. }
17. }
18. }

使pipeline完成操作需要更低的耗时即可。

**四 Redis Lua脚本**

Redis在2.6推出了脚本功能，允许开发者使用**Lua语言编写脚本**传到Redis中执行。使用脚本的好处如下:  
**1.减少网络开销：本来5次网络请求的操作，可以用一个请求完成，原先5次请求的逻辑放在redis服务器上完成。使用脚本，减少了网络往返时延。  
2.原子操作：Redis会将整个脚本作为一个整体执行，中间不会被其他命令插入。  
3.复用：客户端发送的脚本会永久存储在Redis中，意味着其他客户端可以复用这一脚本而不需要使用代码完成同样的逻辑。**

在实际工作过程中，可以使用lua脚本来解决一些需要保证原子性的问题，而且lua脚本可以缓存在redis服务器上，势必会增加性能。

**Lua语法**

见[Lua语言模型 与 Redis应用](http://blog.csdn.net/zjf280441589/article/details/52716720)中关于Lua语法的详述。

**Eval命令**

从Redis2.6.0版本开始，通过内置的Lua解释器，可以使用EVAL命令对Lua脚本进行求值。EVAL命令的格式如下：

**[java]** [view plain](https://blog.csdn.net/KingCat666/article/details/77942830) [copy](https://blog.csdn.net/KingCat666/article/details/77942830)

1. EVAL script numkeys key [key ...] arg [arg ...]

script参数是一段Lua脚本程序，它会被运行在Redis服务器上下文中，这段脚本**不必(也不应该)定义为一个Lua函数**。numkeys参数用于指定键名参数的个数。键名参数 key [key ...] 从EVAL的第三个参数开始算起，表示在脚本中所用到的那些Redis键(key)，这些键名参数可以在 Lua中通过全局变量KEYS数组，用1为基址的形式访问( KEYS[1] ， KEYS[2] ，以此类推)。

在命令的最后，那些不是键名参数的附加参数 arg [arg ...] ，可以在Lua中通过全局变量**ARGV**数组访问，访问的形式和KEYS变量类似( ARGV[1] 、 ARGV[2] ，诸如此类)。例如

**[java]** [view plain](https://blog.csdn.net/KingCat666/article/details/77942830) [copy](https://blog.csdn.net/KingCat666/article/details/77942830)

1. > eval "return {KEYS[1],KEYS[2],ARGV[1],ARGV[2]}" 2 key1 key2 first second
2. 1) "key1"
3. 2) "key2"
4. 3) "first"
5. 4) "second"

其中 "return {KEYS[1],KEYS[2],ARGV[1],ARGV[2]}" 是被求值的Lua脚本，数字2指定了键名参数的数量， key1和key2是键名参数，分别使用 KEYS[1] 和 KEYS[2] 访问，而最后的 first 和 second 则是附加参数，可以通过 ARGV[1] 和 ARGV[2] 访问它们。

**Call和pcall**

在 Lua 脚本中，可以使用两个不同函数来执行Redis命令，它们分别是：

1. redis.call()
2. redis.pcall()

这两个函数的唯一区别在于它们使用不同的方式处理执行命令所产生的错误。当 redis.call() 在执行命令的过程中发生错误时，脚本会停止执行，并返回一个脚本错误，错误的输出信息会说明错误造成的原因，redis.pcall() 出错时并不引发(raise)错误，而是返回一个带 err 域的 Lua 表(table)，用于表示错误。

**Jedis调用**

Jedis中调用脚本需要以字符串形式给出脚本主体，并遵从EVAL的数据规范，示例代码如下：

**[java]** [view plain](https://blog.csdn.net/KingCat666/article/details/77942830) [copy](https://blog.csdn.net/KingCat666/article/details/77942830)

1. String script ="local result={} " +
2. " for i,v in ipairs(KEYS) do " +
3. " result[i] = redis.call('get',v) " +
4. " end " +
5. " return result ";
7. Jedis jedis = **new** Jedis(ip,port);
9. jedis.eval(script,keyCount,String … params);

注意，不要再Lua脚本中出现死循环和耗时的运算，否则redis将不接受其他的命令，这个redis就挂了，只能script kill，如果有写入的话，只能shutdown nosave。 所以**使用时要注意不能出现死循环、耗时的运算。redis是单进程、单线程执行脚本。**

**五 Redis事务、管道和脚本的区别**

1. 事务和脚本从原子性上来说都能满足原子性的要求，其区别在于脚本可借助Lua语言可在**服务器端存储的便利性定制和简化操作，但脚本无法处理长耗时的操作。**

2. 管道是**无状态操作集合**，使用管道可能在效率上比使用script要好，但是有的情况下只能使用script。因为在执行后面的命令时，无法得到前面命令的结果，就像事务一样，所以如果**需要在后面命令中使用前面命令的value等结果，则只能使用script或者事务+watch。**