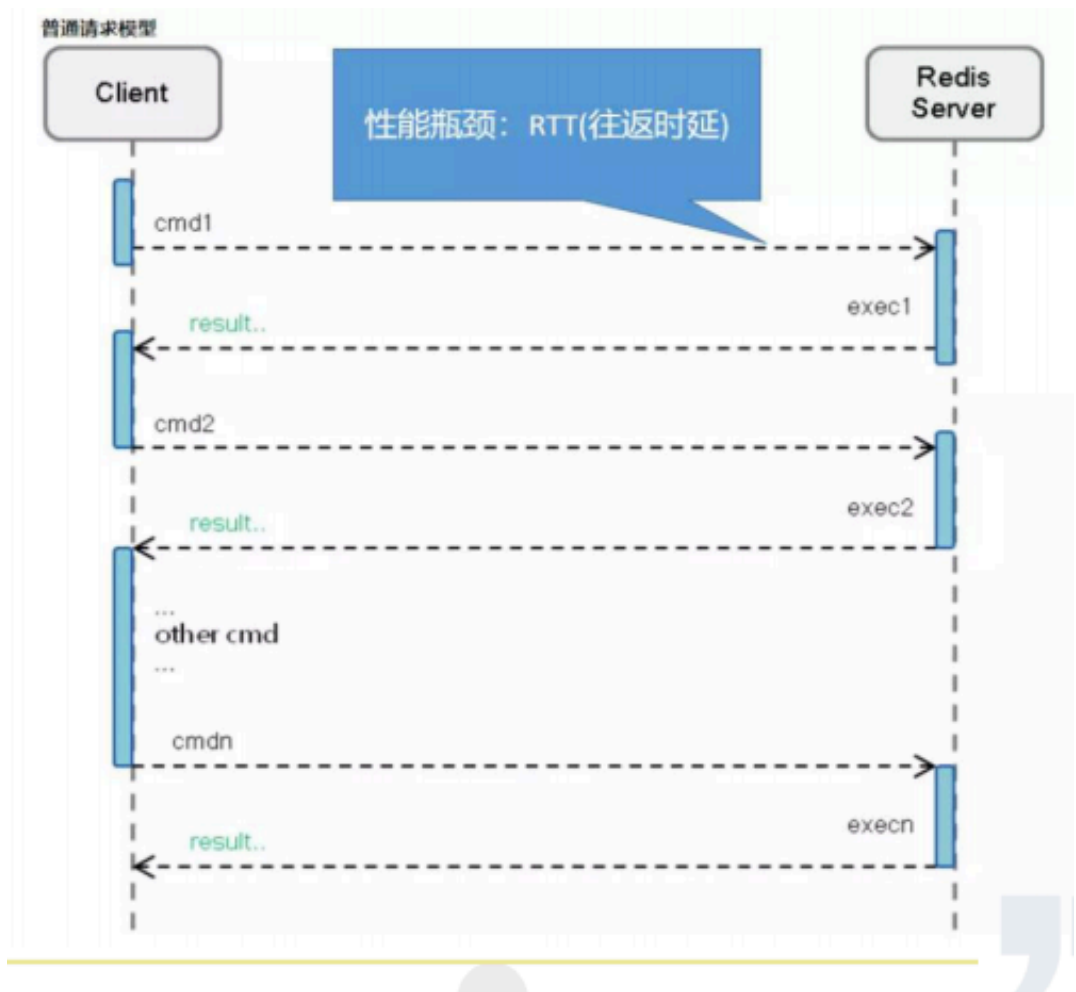


Redis管道与事务

管道（pipeline）

- 1 首先，管道技术是客户端提供的，与服务器无关。通常客户端和服务端请求交互的时候，都是这样的（如下图所示）—— 发送过来一个命令，执行，返回一个结果。这样的话，我们发现，在客户端与服务端交互的过程中，大量的时间用于io的请求传递过程、请求响应过程中。
- 2
- 3 那么我们能不能简化这个流程呢？—— 将多个命令，一起传递给Server，类似于批处理的方式，这样至少减少了多次请求的时间。如果响应的时候也能多次一起返回回来，也就减少了响应的时间。
- 4
- 5 管道的本质，其实就是批处理。让已确定的命令，顺次发送给服务端。这样服务端也就能顺序的处理这些命令了。（形象的说明，就是A、B两点接水，为了将B点的水杯灌满，需要从A点运水。这样就需要A->B、B->A、A->B...不断的往返。但是如果现在有一根管子，那么我们就可以通过这个管子，从A点直接将水灌入管子，通过管子直接运输到B，就不需要自己往返了。我们可以将A点看做是Client，将B点看做是Server）。
- 6
- 7 服务器始终使用，收到-执行-回复的顺序处理消息。
- 8
- 9 而客户端通过对管道中的指令列表改变读写顺序，而节省大幅io时间，指令越多，效果越好。
- 10 管道测试：redis-benchmark (-P)
- 11 管道可以将多个命令打包，一次性的发送给服务器端处理



事务 (transaction)

- 1 一个成熟的数据库，一定要支持事务，以保障多个操作的原子性。
- 2 同时，事务还能保证一个事务中的命令依次执行 不会被其他命令插入。
- 3
- 4 redis支持事务，但是redis只支持部分事务。首先，事务的本质是保证多个命令的原子性。还能够让一个事务中命令依次执行，不被其他命令插队。这些redis都可以支持到。
- 5 但是事务还有一个大的容错方式，就是如果有一个没有执行，那么其他的也都不能执行，可以回滚掉。这是事务的一个非常重要的特性。但是redis不支持。
- 6 redis就相当于，我只帮你执行，出了问题你自己执行。
- 7
- 8 所有事务的基本用法，都是begin、commit、rollback。
- 9 redis事务的指令是，multi、exec、discard，虽然可以使用DISCARD取消事务，但是不支持回滚。
- 10
- 11 当输入MULTI命令后，服务器返回OK表示事务开始成功，然后依次输入需要在本次事务中执行的所有命令，每次输入一个命令服务器并不会马上执行，而是返回“QUEUED”，这表示命令已经被服务器接受并且暂时保存起来，最后输入EXEC命令后，本次事务中的所有命令才会被依次执行。
- 12
- 13 事务错误处理：
- 14 1) 语法错误，全不执行。
- 15 2) 运行错误，出错后仍然继续执行。

事务的指令

```
zhoudw — root@VM-0-10-centos:~ — ssh -q -l root -p 22 121.4.115.241 — 132x39
127.0.0.1:6379> multi multi 多个，表示要开启多个命令的执行 —— 开启事务
OK
127.0.0.1:6379> set k1 v1
QUEUED
127.0.0.1:6379> set k2 v2
QUEUED
127.0.0.1:6379> set k3 v3
QUEUED
127.0.0.1:6379> exec
1) OK
2) OK
3) OK
127.0.0.1:6379> keys *
1) "k2"
2) "k1"
3) "k3"
127.0.0.1:6379> get k1
"v1"
127.0.0.1:6379> 
```

没有真正的执行，还在事务的队列中，排队。—— 顺序排队。

确定只是这三条命令，exec执行，顺序执行。

这个过程可以类比我们生活中，超市购物。

multi 就相当于拿了一个购物车。

set k1 v1
set k2 v2
set k3 v3 这些本质是命令，就相当于商品。
向购物车中，投放商品。

exec 确定了，就买这么多东西，付款。

流程：开启事务(multi) - 多个指令排队 - 执行事务(exec)。

```
127.0.0.1:6379> multi
OK
127.0.0.1:6379> set k4 v5
QUEUED
127.0.0.1:6379> set k5 v5
QUEUED
127.0.0.1:6379> discard 清空所有的命令。也就相当于，我们清空购物车，所有东西都不卖了。
OK
127.0.0.1:6379> get k4
(nil)
127.0.0.1:6379> 
```

需要注意的是，只能在没有执行前，是支持我们清空的。
但是在执行之后，如果出现了问题，是不允许我们回滚的。

```

127.0.0.1:6379> multi
OK
127.0.0.1:6379> set key4 10
QUEUED
127.0.0.1:6379> incr key4
QUEUED
127.0.0.1:6379> set key5 v5
QUEUED
127.0.0.1:6379> incr key5
QUEUED
127.0.0.1:6379> set key6 v6
QUEUED
127.0.0.1:6379> exec
1) OK
2) (integer) 11
3) OK
4) (error) ERR value is not an integer or out of range
5) OK
127.0.0.1:6379> get key6
"v6"
127.0.0.1:6379>

```

Integer类型，可以自增。

string类型，不可以自增。命令没有问题，但是值的类型有问题，string是不可以自增的。问题不在命令本身。

设置key6的值

我们发现，所有的命令都执行了。即使执行过程中出现了错误的值操作（命令是没有问题的）也不影响下面命令的执行。

上面这幅图，演示的是，命令没有语法错误的执行情况，是可以执行的。那么如果现在出现了语法错误，还可以执行吗？

```

127.0.0.1:6379> multi
OK
127.0.0.1:6379> get key4
QUEUED
127.0.0.1:6379> set key7
(error) ERR wrong number of arguments for 'set' command
127.0.0.1:6379> exec
(error) EXECABORT Transaction discarded because of previous errors.
127.0.0.1:6379>

```

出现语法错误，命令都不执行。执行discard，直接清空了所有的命令。

总结：开启事务之后，有两种情况：1.所有指令看起来都是正常的，但是在执行过程中会有问题。在exec的时候仍然会执行，只不过哪条出错了，就给该条指令错误信息，不影响其他指令的执行。这是不回滚的明确体现。2.能够识别出语法错误，这个时候就不再执行了。直接通过调用discard，让事务清空。

```

127.0.0.1:6379> multi
OK
127.0.0.1:6379> get key4
QUEUED
127.0.0.1:6379> set key7
(error) ERR wrong number of arguments for 'set' command
127.0.0.1:6379> exec
(error) EXECABORT Transaction discarded because of previous errors.
127.0.0.1:6379> discard
(error) ERR DISCARD without MULTI

```

multi被清空了，也证明了调用了discard，清空所有的命令了。

通过上述描述，我们发现，对于管道和事务是有共同之处——都是处理多个命令的。而事务一定要处理多个命令，不然就不用事务了。所以说，事务往往是和管道来结合使用。有很多客户端，在使用事务的时候，是要强制使用管道的。这样能够最大节省IO操作的时间。

事务监测

场景模拟

- 1 我们经常说，事务用来解决，形如转账一类的问题。需要确定，银行账户余额，一次转账，只能被改变一次。我们要保证，这一次执行的原子性，怎么保障呢？
- 2 比如说：
- 3 我的余额是，10毛钱。
- 4 甜的余额是，2毛钱。
- 5 我给甜转账，1毛钱。
- 6 ---
- 7 现在：
- 8 我的余额：10-1=9
- 9 甜的余额：2+1=3
- 10 上述算数，就是我们需要获得的变化，但是要保证在变化的过程中，没有其他人再去更改。
- 11 如果甜不知道，我给她转账，自己在这个过程中又花了1毛钱。此时余额就有变化了。
- 12 我们要保证取到甜的余额的时候，余额没有变化。怎么办？——事务提供给我们一个命令：watch（监测），监测当前值是没有变化的，然后我再拿过来改动。

```
127.0.0.1:6379> set wei 10
OK
127.0.0.1:6379> set tian 2
OK
127.0.0.1:6379> multi
OK
127.0.0.1:6379> decr wei
QUEUED
127.0.0.1:6379> incr tian
QUEUED
127.0.0.1:6379> exec
1) (integer) 9
2) (integer) 3
127.0.0.1:6379> █
```

此时是能够拿到正确的结果的，这是因为此时只有一个客户端连接这个服务，不会对tian / wei 进行更改。

但是，事实上，当事务有很多命令，需要执行一小段时间的时候，这段时间 tian / wei 这两个结果是很有可能被更改的。

那么，我们怎么保证在我事务执行期间，这两个结果不会被更改呢？保障我们最终的结果是正确的？

使用watch。

```
127.0.0.1:6379> watch tian
OK
127.0.0.1:6379> watch wei
OK
127.0.0.1:6379> decrby tian 1
(integer) 2
127.0.0.1:6379> multi
OK
127.0.0.1:6379> decr wei
QUEUED
127.0.0.1:6379> incr tian
QUEUED
127.0.0.1:6379> exec
(nil)
127.0.0.1:6379> get tian
"2"
127.0.0.1:6379> get wei
"9"
127.0.0.1:6379> █
```

watch + keyName，监测我们不希望更改的key

现在，我们更改 tain 的值。

事务开启，按照转账的过程，再次输入执行，执行。

发现：返回结果nil，所以我们执行失败了。

因为我们使用watch，监测tian的值不要变化，但是变化了，所以执行失败了。

我们重新查看wei 和 tian 的值，发现，值没有变化。还是原先的。

```
127.0.0.1:6379> watch tian wei
OK
127.0.0.1:6379> multi
OK
127.0.0.1:6379> decr wei
QUEUED
127.0.0.1:6379> incr tian
QUEUED
127.0.0.1:6379> exec
1) (integer) 8
2) (integer) 3
127.0.0.1:6379> █
```

我们重新监测 tian 和 wei，我们希望执行成功，那么我需要保证tian 和 wei，不要出现变化。

所以，我们没有变化这两个键的结果，开启事务，输入指令，执行事务。

看最终结果，正确。

这也说明，我们使用watch就相当于上锁一样，我们可以查看到变化情况。想要执行成功，只要再执行一次就好了。我们可以看到 tian 不仅将钱花出去了，而且，之后转账也成功了。

解释

- 1 将其中一条命令的执行结果作为另一条命令的执行参数，如`i++`，需要使用`watch`命令。
- 2 `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。
- 3 执行`EXEC`命令之后会取消监控使用`WATCH`命令监控的键（实际上，在执行`exec`时，调用了一次`unwatch`），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。
- 4
- 5 使用方式：`watch -> multi -> command -> exec`
- 6
- 7 注意：由于`WATCH`命令的作用只是当被监控的键被修改后取消之后的事务，并不能保证其他客户端不修改监控的值，所以当`EXEC`命令执行失败之后需要手动重新执行整个事务。
- 8
- 9 本质上是一种乐观锁。