小象科技 ChinaHadoop.cn

Redis 从入门到精通

黄健宏（huangz）

事务

事务

Redis 的事务功能允许用户将多个命令包裹起来，然后一次性地、按 顺序地执行被包裹的所有命令。

在事务执行的过程中，服务器不会中断事 务而改去执行其他命令请求，只有在事 务包裹的所有命令都被执行完毕之后，服务器才会去处理其他命令请求。

事务使用示例

现在，让我们假设 SETEX 命令并不存在于 Redis，并且 SET 命令也不支持 EX seconds 参数，如果我们要自己来实现一个 SETEX 命令的话，那么我们可能会使用以下代码：

```
def SETEX(key, seconds, value):
    SET key value
    EXPIRE key seconds
```

在一般情况下，这个自制的 SETEX 命令可以达到设置键值对并设置生产时间的效果，但是这个自制的 SETEX 存在一个缺陷：如果服务器在成功地执行 SET 命令并保存数据之后崩溃，那么 EXPIRE 命令将没办法执行。

这时虽然我们已经设置了键，但并没有为键设置过期时间，如果我们没有发觉的话，那么这个本来应该定期被删除的键就会一直留在数据库里面占用着内存，甚至造成之后的程序出错。

事务命令

为了避免遇上以上所说的情况，我们需要用到 Redis 的事务功能，通过事务，我们可以让 Redis 一次性地执行多个命令，并且确保事务中的命令要么就全部都执行，要么就一个都不执行。

命令	作用
MULTI	开始一个新的事务。
DISCARD	放弃事务。
EXEC	执行事务中的所有命令。

开始事务

MULTI

开始一个事务。

在这个命令执行之后，客户端发送的所有针对数据库或者数据库键的命令都不会被立即执行，而是被放入到一个事务队列里面，并返回 QUEUED 表示命令已入队。

```
redis> MULTI    # 开始一个事务
OK
redis> SET msg "hello world"    # 将这个 SET 命令放入事务队列
QUEUED
redis> EXPIRE msg 10086    # 将这个 SET 命令放入事务队列
QUEUED
```

复杂度为 O(1) 。

放弃事务

小象科技 ChinaHadoop.cn

DISCARD

取消事务，放弃执行事务队列中的所有命令。复 杂度为 O(1)。

```
redis> MULTI
OK
redis> SET msg "hello world"
QUEUED
redis> EXPIRE msg 10086
QUEUED
redis> DISCARD    # 事务已被取消
OK
redis> SET msg "hello world"
OK
```

执行事务

EXEC

按照命令被入队到事务队列中的顺序，执行事务队列中的所有命令。
命令的复杂度为队列中所有命令的复杂度之和。
命令的返回值是一个列表，列表里包含了事务队列中所有被执行命令的返回值。

```
redis> MULTI
OK
redis> SET msg "hello world"
QUEUED
redis> EXPIRE msg 10086
QUEUED
redis> EXEC
1) OK    # SET 命令的返回值
2) (integer) 1    # EXPIRE 命令的返回值
```

使用事务保证操作的安全性

之前的自制 SETEX 的定义，带有安全缺陷：

```
def SETEX(key, seconds, value):
    SET key value
    EXPIRE key seconds    # 如果服务器在 SET 命令执行之后崩溃，那么 EXPIRE 将无法执行
```

使用事务实现的自制 SETEX 的定义，没有安全缺陷，服务器保证要么两个命令都执行，要么就两个命令都不执行：

```
def SETEX(key, seconds, value):
    MULTI
    SET key value
    EXPIRE key seconds
    EXEC
```

流水线和事务的区别

小象科技 ChinaHadoop.cn

功能	性质
流水线	确保多条命令会被一起 发送
事务	确保多条命令会被一起 执行

(PIPELINE_START)
SET msg “hello world” # 这两条命令会被一起 发送至服务器
EXPIRE msg 10086
(PIPELINE_END)

MULTI
SET msg “hello world” # 这两条命令会一起被服 务器执行
EXPIRE msg 10086
EXEC

乐观锁

使用锁来保证数据的正确性

使用乐观锁来保证数据的正确性

通过使用 MULTI 和 EXEC，我们可以将多条命令放到一个事务里面执行，确保事务里面的命令要么全部都被执行，要么就一个都不执行，从而防止数据出错。但是有时候只使用事务还是无法保证数据的正确性，这时候就需要使用 Redis 提供的乐观锁功能（Optimistic Locking）。

举个例子，之前在介绍有序集合的时候我们曾经说过，Redis 只提供了对元素的分值进行自增操作的 ZINCRBY 命令，但是并没有提供相对应的用于对元素的分值进行自减操作的 ZDECRBY 命令，为了实现我们自己的 ZDECRBY 命令，我们可能会写下这样的代码：

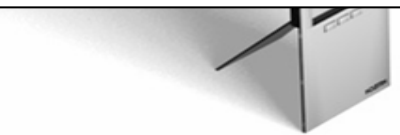
```
def ZDECRBY(key, decrment, member):
    # 取得元素当前的分值
    old_score = ZSCORE key member
    # 使用当前分值减去指定的减量，得出新的分值
    new_score = old_score - decrment
    # 为元素设置新分值，覆盖现有的分值
    ZADD key new_score member
```

ZDECRBY 的竞争条件(1/2)

上面的 ZDECRBY 实现虽然可以实现减少元素分值的效果，但这个实现包含了一个竞争条件，当多个客户端同时对同一个元素调用 ZDECRBY 时，这个竞争条件就可能会出现。

举个例子，假设现在 salary 有序集合 peter 元素的分值为 4000，如果有两个客户端同时对 peter 元素执行 ZDECRBY salary 500 peter 和 ZDECRBY salary 300 peter，就可能会出现下表所示的情况：

时间	客户端 A	客户端 B
T1	执行 ZSCORE salary peter，获得分值 4000，计算 4000 - 500，得出 3500 。	执行 ZSCORE salary peter，获得分值 4000，计算 4000 - 300，得出 3700 。
T2	执行 ZADD salary 3500 peter 。	
T3		执行 ZADD salary 3700 peter 。



ZDECRBY 的竞争条件(2/2)

在正确的情况下，peter 元素原本的分值为 4000，在被扣除 500 和 300 之后，分值应该变为 3200 才对。

但是在上面的这个表中，客户端 A 和客户端 B 先后对 peter 元素的分值进行减法操作，并且客户端 A 的 ZADD 操作比客户端 B 的 ZADD 操作更早执行，当客户端 B 执行 ZADD 时，它不知道 peter 元素当前的分值已经出现了变化，导致它计算出的新分值 3700 已经过期了，而是继续一味地进行设置，使得计算出现了错误。

WATCH 命令

为了消除 ZDECRBY 实现中的竞争条件，我们需要用到 Redis 提供的 WATCH 命令，这个命令需要在开始一个事务之前执行，它接受任意多个键作为参数，并对输入的键进行监视：

WATCH key [key ...]

如果被监视的键在事务提交之前（也即是 EXEC 命令执行之前），已经被其他客户端抢先修改了，那么服务器将拒绝执行客户端提交的事务，并返回 nil 作为事务的回复：

```
redis> WATCH msg    # 监视键 msg
OK
redis> MULTI
OK
redis> SET msg "hello world"
QUEUED
redis> EXEC    # 在这个事务之前，已经有其他客户端对键 msg 进行了修改
(nil)
```

使用 WATCH 来防止竞争条件(1/3)

回到 ZDECRBY 的例子，为了确保 ZDECRBY 执行时，ZADD 设置的新分值是正确的，我们需要在 ZDECRBY 执行一开始时，就使用 WATCH 监视输入的有序集合，并将修改操作 ZADD 添加到事务里面执行，以下是修改后的 ZDECRBY 命令的实现：

```
def ZDECRBY(key, decrment, member):
    # 监视输入的有序集合
    WATCH key
    # 取得元素当前的分值
    old_score = ZSCORE key member
    # 使用当前分值减去指定的减量，得出新的分值
    new_score = old_score - decrment
    # 使用事务包裹 ZADD 命令
    # 确保 ZADD 命令只会在有序集合没有被修改的情况下执行
    MULTI
    ZADD key new_score member    # 为元素设置新分值，覆盖现有的分值
    EXEC
```

使用 WATCH 来防止竞争条件(2/3)

在这个新的 ZDECRBY 执行时，有两种情况可能会出 现：

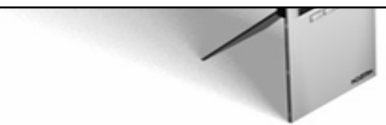
- 如果在程序执行期间，输入的有序集合没有 发生任何变化，那么说明 ZADD 要修改的元素及其分值并没有变化，ZADD 可以正常地更新元素的分 值。

- 如果在程序执行期间，输入的有序集合 发生了变化，那么程序要修改的元素可能已 经发生了变化，这时服务器就会阻止事 务执行，使得 ZADD 无法执行，从而防止竞争条件出现，也避免了元素的分值被设置为错误的值。

使用 WATCH 来防止竞争条件(3/3)

举个例子，下表列出了两个客户端在同时执行 ZDECRBY 命令时，客户端 B 因为 WATCH 命令的效果而导致 ZADD 命令执行失败的情况：

时间	客户端 A	客户端 B
T1	WATCH salary	WATCH salary
T2	执行 ZSCORE salary peter，获得分值 4000，计算 4000 - 500，得出 3500 。	执行 ZSCORE salary peter，获得分值 4000，计算 4000 - 300，得出 3700 。
T3	执行 MULTI 、ZADD salary 3500 peter 和 EXEC 命令，事务执行成功。	
T4		执行 MULTI 、ZADD salary 3700 peter 和 EXEC 命令，因为键 salary 已经被修改，事务执行失败。



和乐观锁有关的其他命令

命令	作用	复杂度
UNWATCH	取消对所有键的监视。	O(1)
DISCARD	放弃执行事务，并且取消对所有键的监视（相当于执行 UNWATCH）。	O(1)

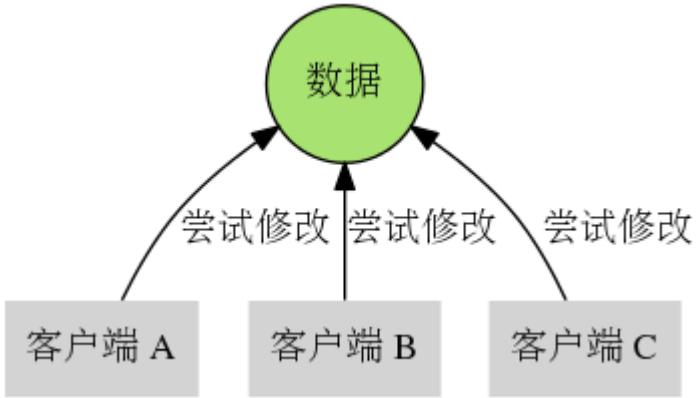
```
redis> WATCH msg name fruits    # 监视三个键
OK

redis> UNWATCH    # 取消对上面三个键的监视
OK
```

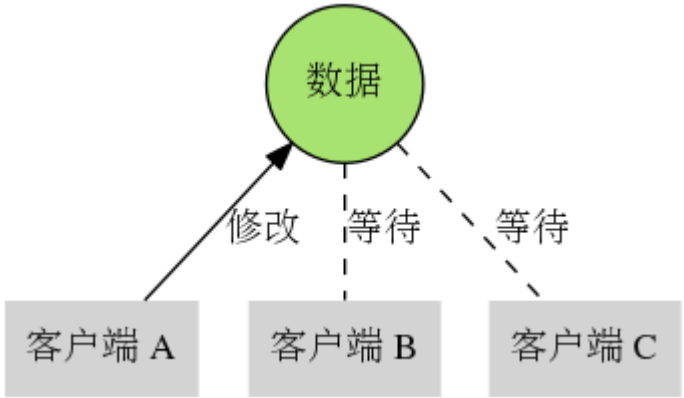
乐观锁和悲观锁的区别

乐观锁会对被加锁的数据进行监视，多个客户端可以同时尝试对数据进行修改，其中最先尝试的客户端会成功，而之后尝试的客户端则会失败。

悲观锁只让一个客户端对数据进行修改，而其他客户端则需要等待正在进行修改的客户端执行完毕之后，才能尝试获得修改权。

对于频繁进行读写操作的 Redis 来说，使用乐观锁可以避免客户端被阻塞：当一个客户端修改数据失败之后，它只要重试就可以了，这个过程不需要进行任何的等待。





复习

回顾一下本节学习的事务相关知识。

复习(1/2)

事务可以让用户将多个命令放入到事 务队列里面，然后一次性地 执行事务队列里面的所有命令，并且事务执行过程中不会被其他命令 请求打断，服务器会等到事 务里面的所有命令都 执行完毕之后，才去处理其他客户端发来的命令请求。

在事务中的命令要么就全部被 执行，要么就一个也不 执行。

命令	作用
MULTI	开始一次新的事 务。
DISCARD	放弃事务。
EXEC	执行事务队列中的所有命令。

复习(2/2)

通过使用 WATCH 命令来监视数据库键，可以防止在执行操作时引入竞争条件。

命令	作用
WATCH key [key …]	监视一个或多个键。
UNWATCH	取消对所有键的监视。
DISCARD	放弃执行事务，并且取消对所有键的监视。