# Redis的SETNX的使用方法

对应给定的keys到他们相应的values上。只要有一个key已经存在，MSETNX一个操作都不会执行。由于这种特性，MSETNX可以实现要么所有的操作都成功，要么一个都不执行，这样可以用来设置不同的key，来表示一个唯一的对象的不同字段。

在 Redis 里，所谓 SETNX，是「SET if Not eXists」的缩写，也就是只有不存在的时候才设置，可以利用它来实现锁的效果，不过很多人没有意识到 SETNX 有陷阱！

比如说：某个查询数据库的接口，因为调用量比较大，所以加了缓存，并设定缓存过期后刷新，问题是当并发量比较大的时候，如果没有锁机制，那么缓存过期的瞬间，大量并发请求会穿透缓存直接查询数据库，造成雪崩效应，如果有锁机制，那么就可以控制只有一个请求去更新缓存，其它的请求视情况要么等待，要么使用过期的缓存。

下面以目前 PHP 社区里最流行的 PHPRedis 扩展为例，实现一段演示代码：

```
<?php

$ok = $redis->setNX($key, $value);

if ($ok) {
    $cache->update();
    $redis->del($key);
}

?>
```

缓存过期时，通过 SetNX 获取锁，如果成功了，那么更新缓存，然后删除锁。看上去逻辑非常简单，可惜有问题：如果请求执行因为某些原因意外退出了，导致创建了锁但是没有删除锁，那么这个锁将一直存在，以至于以后缓存再也得不到更新。于是乎我们需要给锁加一个过期时间以防不测：

```
<?php

$redis->multi();
$redis->setNX($key, $value);
$redis->expire($key, $ttl);
$redis->exec();

?>
```

因为 SetNX 不具备设置过期时间的功能，所以我们需要借助 Expire 来设置，同时我们需要把两者用 Multi/Exec 包裹起来以确保请求的原子性，以免 SetNX 成功了 Expire 却失败了。 可惜还有问题：当多个请求到达时，虽然只有一个请求的 SetNX 可以成功，但是任何一个请求的 Expire 却都可以成功，如此就意味着即便获取不到锁，也可以刷新过期时间，如果请求比较密集的话，那么过期时间会一直被刷新，导致锁一直有效。于是乎我们需要在保证原子性的同时，有条件的执行 Expire，接着便有了如下 Lua 代码：

```
local key   = KEYS[1]
local value = KEYS[2]
local ttl   = KEYS[3]

local ok = redis.call('setnx', key, value)

if ok == 1 then
  redis.call('expire', key, ttl)
end

return ok
```

没想到实现一个看起来很简单的功能还要用到 Lua 脚本，着实有些麻烦。其实 Redis 已经考虑到了大家的疾苦，从 2.6.12 起，SET 涵盖了 SETEX 的功能，并且 SET 本身已经包含了设置过期时间的功能，也就是说，我们前面需要的功能只用 SET 就可以实现。

```
<?php

$ok = $redis->set($key, $value, array('nx', 'ex' => $ttl));
```

```
if ($ok) {
    $cache->update();
    $redis->del($key);
}

?>
```

如上代码是完美的吗？答案是还差一点！设想一下，如果一个请求更新缓存的时间比较长，甚至比锁的有效期还要长，导致在缓存更新过程中，锁就失效了，此时另一个请求会获取锁，但前一个请求在缓存更新完毕的时候，如果不加以判断直接删除锁，就会出现误删除其它请求创建的锁的情况，所以我们在创建锁的时候需要引入一个随机值：

```
<?php

$ok = $redis->set($key, $random, array('nx', 'ex' => $ttl));

if ($ok) {
    $cache->update();
    if ($redis->get($key) == $random) {
        $redis->del($key);
    }
}

?>
```

如此基本实现了单机锁，假如要实现分布锁，请参考：Distributed locks with Redis，这里就不深入讨论了，总结：避免掉入 SETNX 陷阱的最好方法就是永远不要使用它