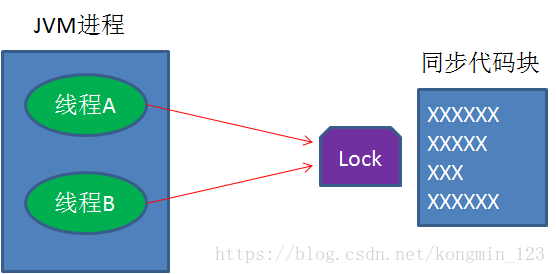
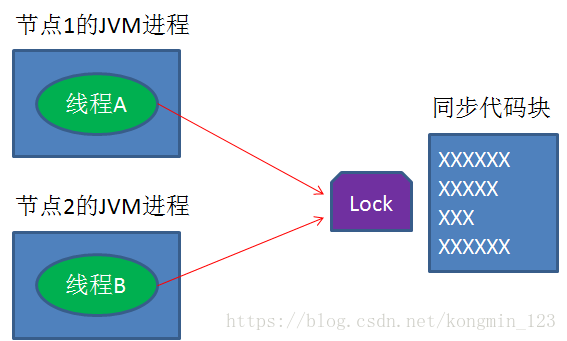
**Redis实现分布式锁**

在多线程并发的情况下，我们可以使用锁来保证一个代码块在同一时间内只能由一个线程访问。比如Java的synchronized关键字和Reentrantlock类等等。

这样子可以保证在同一个JVM进程内的多个线程同步执行。



如果在分布式的集群环境中，如何保证不同节点的线程同步执行呢？



怎么才能在分布式系统中，实现不同线程对代码和资源的同步访问？

对于单进程的并发场景，我们可以使用语言和类库提供的锁。对于分布式场景，我们可以使用分布式锁。

那么怎么才能实现分布式系统中的锁呢？

分布式锁有许多中实现方法，下面简单列举一下。

分布式锁的实现有哪些？

1.Memcached分布式锁

利用Memcached的add命令。此命令是原子性操作，只有在key不存在的情况下，才能add成功，也就意味着线程得到了锁。

2.Redis分布式锁

和Memcached的方式类似，利用Redis的setnx命令。此命令同样是原子性操作，只有在key不存在的情况下，才能set成功。（setnx命令并不完善，后续会介绍替代方案）

3.Zookeeper分布式锁

利用Zookeeper的顺序临时节点，来实现分布式锁和等待队列。Zookeeper设计的初衷，就是为了实现分布式锁服务的。

首先讲一下Redis的分布式锁，这种实现方式比较有代表性。

如何用Redis实现分布式锁？

Redis分布式锁的基本流程并不难理解，但要想写得尽善尽美，也并不是那么容易。在这里，我们需要先了解分布式锁实现的三个核心要素：

1.加锁

最简单的方法是使用setnx命令。key是锁的唯一标识，按业务来决定命名。比如想要给一种商品的秒杀活动加锁，可以给key命名为 “lock\_sale\_商品ID” 。而value设置成什么呢？锁的value值为一个随机生成的UUID。我们可以姑且设置成1。加锁的伪代码如下：

setnx（key，1）

当一个线程执行setnx返回1，说明key原本不存在，该线程成功得到了锁；当一个线程执行setnx返回0，说明key已经存在，该线程抢锁失败。

2.解锁

有加锁就得有解锁。当得到锁的线程执行完任务，需要释放锁，以便其他线程可以进入。释放锁的最简单方式是执行del指令，伪代码如下：

del（key）

释放锁之后，其他线程就可以继续执行setnx命令来获得锁。

3.锁超时

锁超时是什么意思呢？如果一个得到锁的线程在执行任务的过程中挂掉，来不及显式地释放锁，这块资源将会永远被锁住，别的线程再也别想进来。

所以，setnx的key必须设置一个超时时间，单位为second，以保证即使没有被显式释放，这把锁也要在一定时间后自动释放，避免死锁。setnx不支持超时参数，所以需要额外的指令，伪代码如下：

expire（key， 30）

综合起来，我们分布式锁实现的第一版伪代码如下：

if（setnx（key，1） == 1）{

    expire（key，30）

    try {

        do something ......

    } finally {

        del（key）

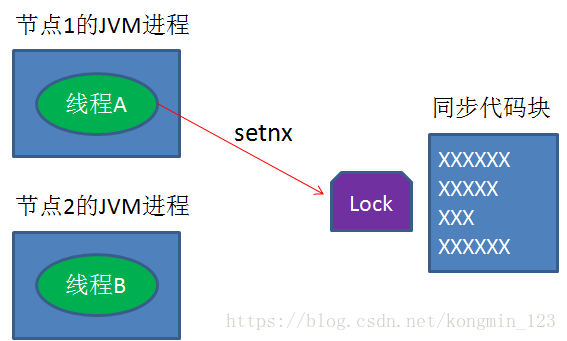
    }

}

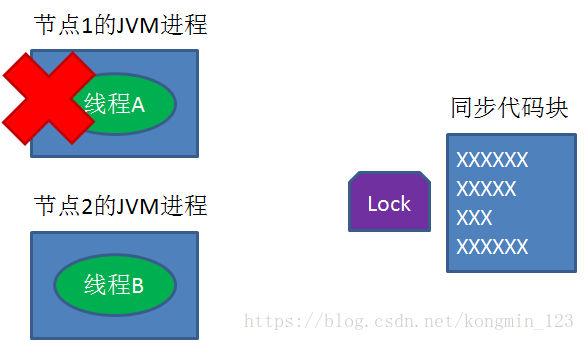
上面的伪代码只是分布式锁的简单实现，结合实际应用场景考虑就会发现上述分布式锁的实现存在着三个致命问题：

1. setnx和expire的非原子性

设想一个极端场景，当某线程执行setnx，成功得到了锁：



setnx刚执行成功，还未来得及执行expire指令，节点1 Duang的一声挂掉了。



这样一来，这把锁就没有设置过期时间，变得“长生不老”，别的线程再也无法获得锁了。

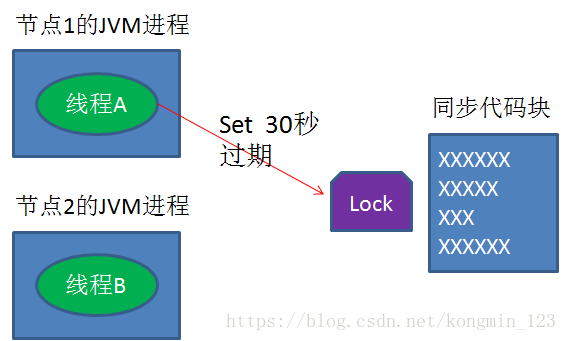
怎么解决呢？setnx指令本身是不支持传入超时时间的，幸好Redis 2.6.12以上版本为set指令增加了可选参数，伪代码如下：

set（key，1，30，NX）

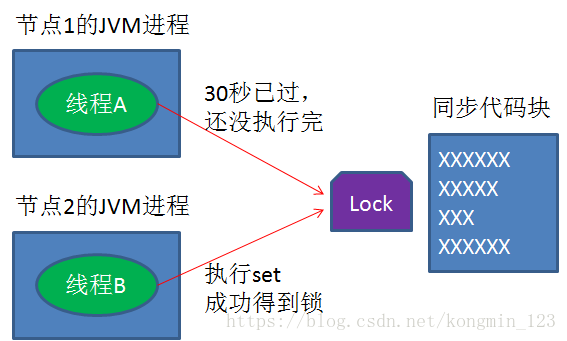
这样就可以取代setnx指令。

2. del 导致误删

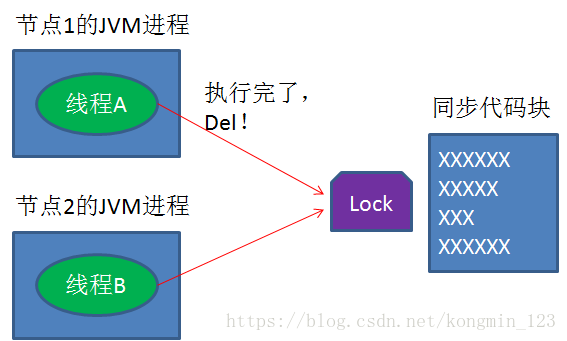
又是一个极端场景，假如某线程成功得到了锁，并且设置的超时时间是30秒。



如果某些原因导致线程A执行的很慢很慢，过了30秒都没执行完，这时候锁过期自动释放，线程B得到了锁。



随后，线程A执行完了任务，线程A接着执行del指令来释放锁。但这时候线程B还没执行完，线程A实际上删除的是线程B加的锁。



怎么避免这种情况呢？可以在del释放锁之前做一个判断，验证当前的锁是不是自己加的锁。

至于具体的实现，可以在加锁的时候把当前的线程ID当做value，并在删除之前验证key对应的value是不是自己线程的ID。

加锁：

String threadId = Thread.currentThread().getId()

set（key，threadId ，30，NX） // 保证原子性操作

解锁：

if（threadId .equals(redisClient.get(key))）{

    del(key)

}

也可以在释放锁的时候，通过锁的默认value值UUID判断是不是该锁，若是该锁，则执行delete进行锁释放。

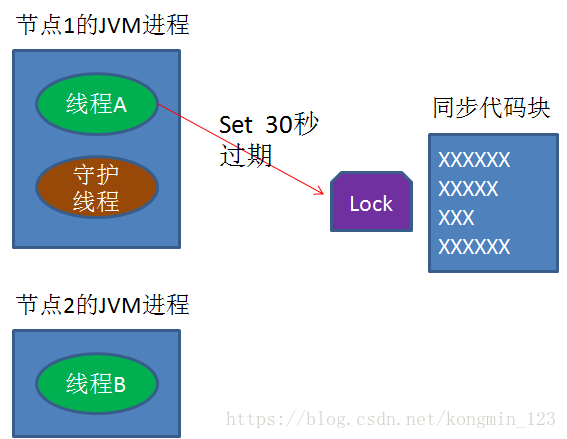
但是，这样做又隐含了一个新的问题，判断和释放锁是两个独立操作，不是原子性的。

要想实现验证和删除过程的原子性，可以使用Lua脚本来实现。这样就能保证验证和删除过程的正确性了。

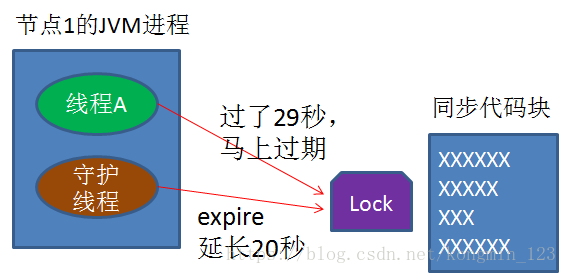
3. 出现并发的可能性

还是刚才第二点所描述的场景，虽然我们避免了线程A误删掉key的情况，但是同一时间有A，B两个线程在访问代码块，仍然是不完美的。

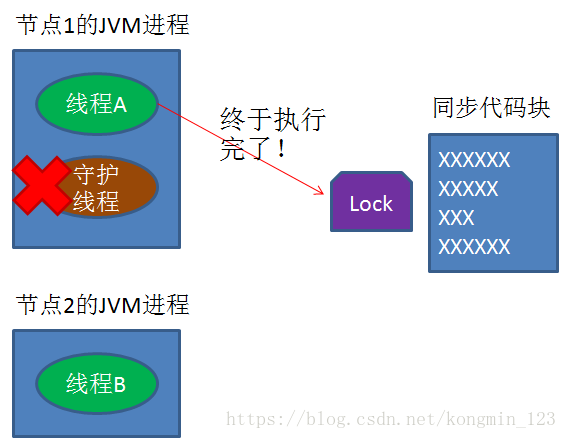
怎么办呢？我们可以让获得锁的线程开启一个守护线程，用来给快要过期的锁“续航”。



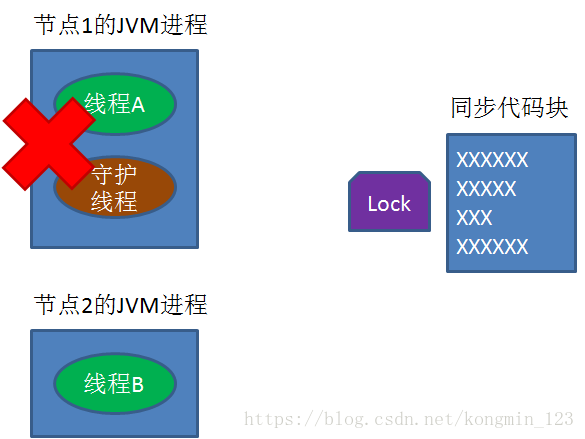
当过去了29秒，线程A还没执行完，这时候守护线程会执行expire指令，为这把锁“续命”20秒。守护线程从第29秒开始执行，每20秒执行一次。



当线程A执行完任务，会显式关掉守护线程。



另一种情况，如果节点1 忽然断电，由于线程A和守护线程在同一个进程，守护线程也会停下。这把锁到了超时的时候，没人给它续命，也就自动释放了。



关于Redis分布式锁的内容就介绍到这里啦。

参考：程序员小灰—《什么是分布式锁》

————————————————

版权声明：本文为CSDN博主「kongmin\_123」的原创文章，遵循CC 4.0 BY-SA版权协议，转载请附上原文出处链接及本声明。

原文链接：https://blog.csdn.net/kongmin\_123/article/details/82080962