第16讲 | synchronized底层如何实现？什么是锁的升级、降级？

2018-06-12 杨晓峰

第16讲 | synchronized底层如何实现？什么是锁的升级、降级？
杨晓峰
00:00 / 11:02

我在上一讲对比和分析了synchronized和ReentrantLock，算是专栏进入并发编程阶段的热身，相信你已经对线程安全，以及如何使用基本的同步机制有了基础，今天我们将深入了解synchronize底层机制，分析其他锁实现和应用场景。

今天我要问你的问题是，synchronized底层如何实现？什么是锁的升级、降级？

典型回答

在回答这个问题前，先简单复习一下上一讲的知识点。synchronized代码块是由一对儿monitorenter/monitorexit指令实现的，Monitor对象是同步的基本实现单元。

在Java 6之前，Monitor的实现完全是依靠操作系统内部的互斥锁，因为需要进行用户态到内核态的切换，所以同步操作是一个无差别的重量级操作。

现代的（Oracle）JDK中，JVM对此进行了大刀阔斧地改进，提供了三种不同的Monitor实现，也就是常说的三种不同的锁：偏斜锁（Biased Locking）、轻量级锁和重量级锁，大大改进了其性能。

所谓锁的升级、降级，就是JVM优化synchronized运行的机制，当JVM检测到不同的竞争状况时，会自动切换到适合的锁实现，这种切换就是锁的升级、降级。

当没有竞争出现时，默认会使用偏斜锁。JVM会利用CAS操作（compare and swap），在对象头上的Mark Word部分设置线程ID，以表示这个对象偏向于当前线程，所以并不涉及真正的互斥锁。这样做的假设是基于在很多应用场景中，大部分对象生命周期中最多会被一个线程锁定，使用偏斜锁可以降低无竞争开销。

如果有另外的线程试图锁定某个已经被偏斜过的对象，JVM就需要撤销（revoke）偏斜锁，并切换到轻量级锁实现。轻量级锁依赖CAS操作Mark Word来试图获取锁，如果重试成功，就使用普通的轻量级锁；否则，进一步升级为重量级锁。

我注意到有的观点认为Java不会进行锁降级。实际上据我所知，锁降级确实是会发生的，当JVM进入安全点（SafePoint）的时候，会检查是否有闲置的Monitor，然后试图进行降级。

考点分析

今天的问题主要是考察你对Java内置锁实现的掌握，也是并发的经典题目。我在前面给出的典型回答，涵盖了一些基本概念。如果基础不牢，有些概念理解起来就比较晦涩，我建议还是尽量理解和掌握，即使有不懂的也不用担心，在后续学习中还会逐步加深认识。

我个人认为，能够基础性地理解这些概念和机制，其实对于大多数并发编程已经足够了，毕竟大部分工程师未必会进行更底层、更基础的研发，很多时候解决的是知道与否，真正的提高还要靠实践踩坑。

后面我会进一步分析：

- 从源码层面，稍微展开一些synchronized的底层实现，并补充一些上面答案中欠缺的细节，有同学反馈这部分容易被问到。如果你对Java底层源码有兴趣，但还没有找到入手点，这里可以成为一个切入点。
- 理解并发包中java.util.concurrent.lock提供的其他锁实现，毕竟Java可不是只有ReentrantLock一种显式的锁类型，我会结合代码分析其使用。

知识扩展

我在上一讲提到过synchronized是JVM内部的Intrinsic Lock，所以偏斜锁、轻量级锁、重量级锁的代码实现，并不在核心类库部分，而是在JVM的代码中。

Java代码运行可能是解释模式也可能是编译模式（如果不记得，请复习专栏第1讲），所以对应的同步逻辑实现，也会分散在不同模块下，比如，解释器版本就是：

src/hotspot/share/interpreter/interpreterRuntime.cpp

为了简化便于理解，我这里会专注于通用的基类实现：

src/hotspot/share/runtime/

另外请注意，链接指向的是最新JDK代码库，所以可能某些实现与历史版本有所不同。

首先，synchronized的行为是JVM runtime的一部分，所以我们需要先找到Runtime相关的功能实现。通过在代码中查询类似“monitor_enter”或“Monitor Enter”，很直观的就可以定位到：

- sharedRuntime.cpp/hpp，它是解释器和编译器运行时的基类。
- synchronizer.cpp/hpp，JVM同步相关的各种基础逻辑。

在sharedRuntime.cpp中，下面代码体现了synchronized的主要逻辑。

```
Handle h_obj(THREAD, obj);
  if (UseBiasedLocking) {
    // Retry fast entry if bias is revoked to avoid unnecessary inflation
    ObjectSynchronizer::fast_enter(h_obj, lock, true, CHECK);
  } else {
    ObjectSynchronizer::slow_enter(h_obj, lock, CHECK);
  }
```

其实现可以简单进行分解：

- UseBiasedLocking是一个检查，因为，在JVM启动时，我们可以指定是否开启偏斜锁。

偏斜锁并不适合所有应用场景，撤销操作（revoke）是比较重的行为，只有当存在较多不会真正竞争的synchronized块儿时，才能体现出明显改善。实践中对于偏斜锁的一直是有争议的，有人甚至认为，当你需要大量使用并发类库时，往往意味着你不需要偏斜锁。从具体选择来看，我还是建议需要在实践中进行测试，根据结果再决定是否使用。

还有一方面是，偏斜锁会延缓JIT 预热的进程，所以很多性能测试中会显式地关闭偏斜锁，命令如下：

```
-XX:-UseBiasedLocking
```

- fast_enter是我们熟悉的完整锁获取路径，slow_enter则是绕过偏斜锁，直接进入轻量级锁获取逻辑。

那么fast_enter是如何实现的呢？同样是通过在代码库搜索，我们可以定位到synchronizer.cpp。 类似fast_enter这种实现，解释器或者动态编译器，都是拷贝这段基础逻辑，所以如果我们修改这部分逻辑，要保证一致性。这部分代码是非常敏感的，微小的问题都可能导致死锁或者正确性问题。

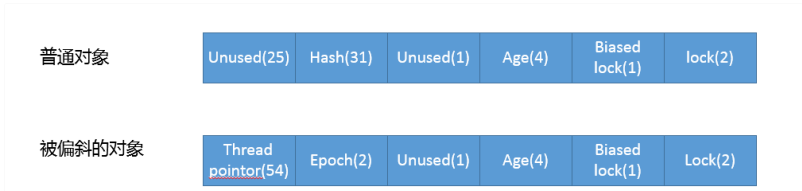
```
void ObjectSynchronizer::fast_enter(Handle obj, BasicLock* lock,
                                    bool attempt_rebias, TRAPS) {
  if (UseBiasedLocking) {
    if (!SafepointSynchronize::is_at_safepoint()) {
      BiasedLocking::Condition cond = BiasedLocking::revoke_and_rebias(obj, attempt_rebias, THREAD);
      if (cond == BiasedLocking::BIAS_REVOKED_AND_REBIASED) {
        return;
      }
    } else {
      assert(!attempt_rebias, "can not rebias toward VM thread");
      BiasedLocking::revoke_at_safepoint(obj);
    }
    assert(!obj->mark()->has_bias_pattern(), "biases should be revoked by now");
  }

  slow_enter(obj, lock, THREAD);
}
```

我来分析下这段逻辑实现：

- biasedLocking定义了偏斜锁相关操作，revoke_and_rebias是获取偏斜锁的入口方法，revoke_at_safepoint则定义了当检测到安全点时的处理逻辑。
- 如果获取偏斜锁失败，则进入slow_enter。
- 这个方法里面同样检查是否开启了偏斜锁，但是从代码路径来看，其实如果关闭了偏斜锁，是不会进入这个方法的，所以算是个额外的保障性检查吧。

另外，如果你仔细查看synchronizer.cpp里，会发现不仅仅是synchronized的逻辑，包括从本地代码，也就是JNI，触发的Monitor动作，全都可以在里面找到（jni_enter/jni_exit）。

关于biasedLocking的更多细节我就不展开了，明白它是通过CAS设置Mark Word就完全够用了，对象头中Mark Word的结构，可以参考下图：



顺着锁升降级的过程分析下去，偏斜锁到轻量级锁的过程是如何实现的呢？

我们来看看slow_enter到底做了什么。

```
void ObjectSynchronizer::slow_enter(Handle obj, BasicLock* lock, TRAPS) {
  markOop mark = obj->mark();
 if (mark->is_neutral()) {
       // 将目前的Mark Word复制到Displaced Header上
    lock->set_displaced_header(mark);
    // 利用CAS设置对象的Mark Word
    if (mark == obj()->cas_set_mark((markOop) lock, mark)) {
      TEVENT(slow_enter: release stacklock);
      return;
    }
    // 检查存在竞争
  } else if (mark->has_locker() &&
             THREAD->is_lock_owned((address)mark->locker())) {
    // 清除
    lock->set_displaced_header(NULL);
    return;
  }

  // 重置Displaced Header
  lock->set_displaced_header(markOopDesc::unused_mark());
  ObjectSynchronizer::inflate(THREAD,
                              obj(),
                              inflate_cause_monitor_enter)->enter(THREAD);
}
```

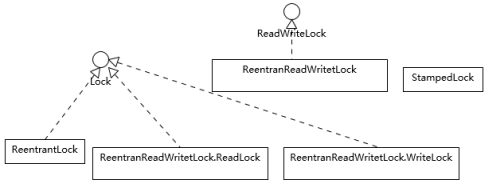
请结合我在代码中添加的注释，来理解如何从试图获取轻量级锁，逐步进入锁膨胀的过程。你可以发现这个处理逻辑，和我在这一讲最初介绍的过程是十分吻合的。

- 设置Displaced Header，然后利用cas_set_mark设置对象Mark Word，如果成功就成功获取轻量级锁。
- 否则Displaced Header，然后进入锁膨胀阶段，具体实现在inflate方法中。

今天就不介绍膨胀的细节了，我这里提供了源代码分析的思路和样例，考虑到应用实践，再进一步增加源代码解读意义不大，有兴趣的同学可以参考我提供的synchronizer.cpp链接，例如：

- deflate_idle_monitors是分析锁降级逻辑的入口，这部分行为还在进行持续改进，因为其逻辑是在安全点内运行，处理不当可能拖长JVM停顿（STW，stop-the-world）的时间。
- fast_exit或者slow_exit是对应的锁释放逻辑。

前面分析了synchronized的底层实现，理解起来有一定难度，下面我们来看一些相对轻松的内容。 我在上一讲对比了synchronized和ReentrantLock，Java核心类库中还有其他一些特别的锁类型，具体请参考下面的图。



你可能注意到了，这些锁竟然不都是实现了Lock接口，ReadWriteLock是一个单独的接口，它通常是代表了一对儿锁，分别对应只读和写操作，标准类库中提供了再入版本的读写锁实现（ReentrantReadWriteLock），对应的语义和ReentrantLock比较相似。

StampedLock竟然也是个单独的类型，从类图结构可以看出它是不支持再入性的语义的，也就是它不是以持有锁的线程为单位。

为什么我们需要读写锁（ReadWriteLock）等其他锁呢？

这是因为，虽然ReentrantLock和synchronized简单实用，但是行为上有一定局限性，通俗点说就是“太霸道”，要么不占，要么独占。实际应用场景中，有的时候不需要大量竞争的写操作，而是以并发读取为主，如何进一步优化并发操作的粒度呢？

Java并发包提供的读写锁等扩展了锁的能力，它所基于的原理是多个读操作是不需要互斥的，因为读操作并不会更改数据，所以不存在互相干扰。而写操作则会导致并发一致性的问题，所以写线程之间、读写线程之间，需要精心设计的互斥逻辑。

下面是一个基于读写锁实现的数据结构，当数据量较大，并发读多、并发写少的时候，能够比纯同步版本凸显出优势。

```
public class RWSample {
    private final Map<String, String> m = new TreeMap<>();
    private final ReentrantReadWriteLock rwl = new ReentrantReadWriteLock();
    private final Lock r = rwl.readLock();
    private final Lock w = rwl.writeLock();
```

```
    public String get(String key) {
        r.lock();
        System.out.println("读锁锁定！");
        try {
            return m.get(key);
        } finally {
            r.unlock();
        }
    }

    public String put(String key, String entry) {
        w.lock();
    System.out.println("写锁锁定！");
            try {
                return m.put(key, entry);
            } finally {
                w.unlock();
            }
        }
    // …
    }
```

在运行过程中，如果读锁试图锁定时，写锁是被某个线程持有，读锁将无法获得，而只好等待对方操作结束，这样就可以自动保证不会读取到有争议的数据。

读写锁看起来比synchronized的粒度似乎细一些，但在实际应用中，其表现也并不尽如人意，主要还是因为相对比较大的开销。

所以，JDK在后期引入了StampedLock，在提供类似读写锁的同时，还支持优化读模式。优化读基于假设，大多数情况下读操作并不会和写操作冲突，其逻辑是先试着修改，然后通过validate方法确认是否进入了写模式，如果没有进入，就成功避免了开销；如果进入，则尝试获取读锁。请参考我下面的样例代码。

```
public class StampedSample {
    private final StampedLock sl = new StampedLock();

    void mutate() {
        long stamp = sl.writeLock();
        try {
            write();
        } finally {
            sl.unlockWrite(stamp);
        }
    }

    Data access() {
        long stamp = sl.tryOptimisticRead();
        Data data = read();
        if (!sl.validate(stamp)) {
            stamp = sl.readLock();
            try {
                data = read();
            } finally {
                sl.unlockRead(stamp);
            }
        }
        return data;
    }
    // …
}
```

注意，这里的writeLock和unLockWrite一定要保证成对调用。

你可能很好奇这些显式锁的实现机制，Java并发包内的各种同步工具，不仅仅是各种Lock，其他的如Semaphore、CountDownLatch，甚至是早期的FutureTask等，都是基于一种AQS框架。

今天，我全面分析了synchronized相关实现和内部运行机制，简单介绍了并发包中提供的其他显式锁，并结合样例代码介绍了其使用方法，希望对你有所帮助。

一课一练

关于今天我们讨论的你做到心中有数了吗？思考一个问题，你知道“自旋锁”是做什么的吗？它的使用场景是什么？

请你在留言区写写你对这个问题的思考，我会选出经过认真思考的留言，送给你一份学习奖励礼券，欢迎你与我一起讨论。

你的朋友是不是也在准备面试呢？你可以“请朋友读”，把今天的题目分享给好友，或许你能帮到他。

公号-Java大后端

2018-06-12

自旋锁:竞争锁的失败的线程，并不会真实的在操作系统层面挂起等待，而是JVM会让线程做几个空循环(基于预测在不久的将来就能获得)，在经过若干次循环后，如果可以获得锁，那么进入临界区，如果还不能获得锁，才会真实的将线程在操作系统层面进行挂起。

适用场景:自旋锁可以减少线程的阻塞，这对于锁竞争不激烈，且占用锁时间非常短的代码块来说，有较大的性能提升，因为自旋的消耗会小于线程阻塞挂起操作的消耗。
如果锁的竞争激烈，或者持有锁的线程需要长时间占用锁执行同步块，就不适合使用自旋锁了，因为自旋锁在获取锁前一直都是占用cpu做无用功，线程自旋的消耗大于线程阻塞挂起操作的消耗，造成cpu的浪费。

作者回复

2018-06-12

不错，自旋是种乐观情况的优化

yearning

2018-06-12

这次原理真的看了很久，一直鼓劲自己，看不懂就是说明自己有突破。

下面看了并发编程对于自旋锁的了解，同时更深刻理解同步锁的性能。

自旋锁采用让当前线程不停循环体内执行实现，当循环条件被其他线程改变时，才能进入临界区。

由于自旋锁只是将当前线程不停执行循环体，不进行线程状态的改变，所以响应会更快。但当线程不停增加时，性能下降明显。
线程竞争不激烈，并且保持锁的时间段。适合使用自旋锁。

为什么会提出自旋锁，因为互斥锁，在线程的睡眠和唤醒都是复杂而昂贵的操作，需要大量的CPU指令。如果互斥仅仅被锁住是一小段时间，
用来进行线程休眠和唤醒的操作时间比睡眠时间还长，更有可能比不上不断自旋锁上轮询的时间长。

当然自旋锁被持有的时间更长，其他尝试获取自旋锁的线程会一直轮询自旋锁的状态。这将十分浪费CPU。

在单核CPU上，自旋锁是无用，因为当自旋锁尝试获取锁不成功会一直尝试，这会一直占用CPU，其他线程不可能运行，
同时由于其他线程无法运行，所以当前线程无法释放锁。

混合型互斥锁， 在多核系统上起初表现的像自旋锁一样， 如果一个线程不能获取互斥锁， 它不会马上被切换为休眠状态，在一段时间依然无法获取锁，进行睡眠状态。

混合型自旋锁，起初表现的和正常自旋锁一样，如果无法获取互斥锁，它也许会放弃该线程的执行，并允许其他线程执行。

切记，自旋锁只有在多核CPU上有效果，单核毫无效果，只是浪费时间。

以上基本参考来源于:
http://ifeve.com/java_lock_see1/
http://ifeve.com/practice-of-using-spinlock-instead-of-mutex/

作者回复

2018-06-12

很不错总结

黑子

2018-06-12

自旋锁 for(;;)结合cas确保线程获取取锁

作者回复

2018-06-12

差不多

sunlight001

2018-06-12

自旋锁是尝试获取锁的线程不会立即阻塞，采用循环的方式去获取锁，好处是减少了上下文切换，缺点是消耗cpu

作者回复

2018-06-12

不错

灰飞灰猪不会灰飞.烟灭

2018-06-12

老师 AQS就不涉及用户态和内核态的切换了 对吧?

作者回复

2018-06-12

我理解是，cas是基于特定指令

jacy

2018-06-19

看了大家对自旋锁的评论，我的收获如下：
1.基于乐观情况下推荐使用，即锁竞争不强，锁等待时间不长的情况下推荐使用
2.单cpu无效，因为基于cas的轮询会占用cpu,导致无法做线程切换
3.轮询不产生上下文切换，如果可估计到睡眠的时间很长，用互斥锁更好

作者回复

2018-06-19

不错

Miaozhe

2018-06-13

杨老师，看到有回复说自旋锁在单核CPU上是无用，感觉这个理论不准确，因为Java多线程在很早时候单核CPC的PC上就能运行，计算机原理中也介绍，控制器会轮巡各个进程或线程。而且多线程是运行在JVM上，跟物理机没有很直接的关系吧？

作者回复

2018-06-13

已回复，我也认为单核无用

齐帜

2018-06-12

老师后面会详细讲 AQS 吗

作者回复

2018-06-12

有的

肖一林

2018-06-12

重量级锁还是互斥锁吗？自旋锁应该是线程拿不到锁的时候，采取重试的办法，适合重试次数不多的场景，如果重试次数过多还是会被系统挂起，这种情况下还不如没有自旋锁。

作者回复

2018-06-12

是的

宾语

2018-07-15

挺不错

刘杰

2018-07-12

偏斜锁和轻量级锁的区别不是很清晰

有福

2018-07-06

自旋锁基本形式就是通过while循环加上cpu指令级别保证的cas原子操作来判断某一个共享变量内存的值是否被其他线程修改，如果没有修改那么就认为获取到了锁。
这个变量需要设置为volatile.否则容易被指令重排引发bug。

之前实际应用的场景就是开发多核处理器下的core to core的高性能无锁队列。

由于是一直while循环，所以cpu在检查锁状态的时候基本上是100%，所以自旋锁基本上是用来判断某个状态是否发生，也就是用来同步的，而不是用来互斥的。

叮叮个呸的

2018-07-03

可是我还是不清楚偏斜锁和轻量级锁的区别

zeusoul

2018-06-23

老师，你好，我想问下，Java 的CAS和volatile对于服务器集群是否支持。

Cui

2018-06-22

基于AQS的锁是属于哪种级别的锁？

Cui

2018-06-22

老师你好 心中一直有个疑问：synchronize和AQS的LockSupport同样起到阻塞线程的作用，这两者的区别是什么？能不能从实现原理和使用效果的角度说说？

作者回复

2018-06-26

LockSupport park是waiting，另一个是blocked；具体底层，马上一篇有说明

Miaozhe

2018-06-15

杨老师，StampedSample这个例子，access方法是不是写错了？
long stamp = sl.tryOptimisticRead();
Data data = read();
应该是先判断tryOptimisticRead的结果，如果获取了所，才进入Read()吧？因为没有获取锁的读，可能是脏读。
自己代码调试，发现即使try Optimistic的结果为0，也会向下执行read().

作者回复

2018-06-17

我记得validate会返回false，如果输入stamp是0，所以程序并没有漏洞

Miaozhe

2018-06-13

关于自旋转锁不适合单核CPU的问题，下来查找了一下资料：
1.JVM在操作系统中是作为一个进程存在，但是OS一般都将将线程作为最小调度单位，进程是资源分配的最小单位。这就是说进程是不活动的，只是作为线程的容器，那么Java的线程是在JVM进程中，也被CPU调度。
2.单核CPU使用多线程时，一个线程被CPU执行，其它处于等待轮巡状态。
3.为什么多线程跑在单核CPU上也比较快呢？是由于这种线程还有其它IO操作(File,Socket)，可以跟CPU运算并行。
4.结论，根据前面3点的分析，与自旋转锁的优点冲突：线程竞争不激烈，占用锁时间短。

作者回复

2018-06-13

自旋是基于乐观假设，就是等待中锁被释放了，单核cpu就自己占着cpu，别人没机会让

I.am DZX

2018-06-13

请问自旋锁和非公平获取锁是不是有点冲突了

作者回复

2018-06-13

我理解非公平是不保证，另外自旋抢到的线程不见得就是等的久的

TWO STRINGS

2018-06-13

StampedLock那里乐观读锁好像是说写操作不需要等待读操作完成，而不是"读操作并不需要等待写完成"吧

作者回复

2018-06-13

非常感谢，这话写的是有问题

食指可爱多

2018-06-13

以前写过自旋锁的实现，当某个线程调用自旋锁实例的lock方法时，使用cas进行设置，cas（lockThread, null, currentThread）,也就是当前无锁定时当前线程会成功,失败则循环尝试直到成功。利用cas保证操作的原子性，成员变量lockThread设置为volatile保证并发时线程间可见性。所以从机制上可以看到，若是在高并发场景，成功拿到锁之外的所有线程会继续努力尝试持有锁，造成CPU资源的浪费。如评论中其它同学所说适合在低并发场景使用。

作者回复

2018-06-13

是的

Geek_e61ae8

2018-06-13

老师讲到读写锁，这里涉及到读并发高，当我更改要加载的数据，这时需要写，读到内存后准备切换，但是一直获取不了写锁。这种采用自己boolean值来控制，让读sleep等待，或者直接返回不进锁（已经获取读锁的线程等处理结束）。写获取锁后更新，替换boolean值。另一种采用公平锁。老师觉得建议那种？

作者回复

2018-06-13

我建议用StampedLock或读写锁

Geek_e61ae8

2018-06-13

这块老师讲了读写锁，如果读并发高，当配置更改，触发了写，但是又获取不了锁，这种情况可以采用boolean值自己控制当写完，替换内存时，让读的线程等待。（已经获取锁的等处理完）没处理的等待。 这种是建议加上公平锁好，还是说自己控制好

作者回复

2018-06-13

没看懂，是说让读线程不停的检查boolean值等待吗？自己控制要达到的目的是什么呢

凡旅

2018-06-12

杨老师，操作系统的互斥锁要怎么理解

作者回复

2018-06-13

这个还是请看操作系统相关代码或资料，原理上mutex和只有0、1值的semaphore是近似的，但现代操作系统怎么实现我真没研究过，谁有空儿补充下？

Miaozhe

2018-06-12

杨老师，偏斜锁有什么作用？还是没有看明白，如果只是被一个线程获取，那么锁还有什么意义？
另外，如果我有两个线程明确定义调用同一个对象的Synchronized块，JVM默认肯定先使用偏斜锁，之后在升级到轻量级所，必须经过撤销Revoke吗？编译的时候不会自动优化？

作者回复

2018-06-12

我理解偏斜锁就是为了优化那些没有并发却写了同步逻辑的代码；javac编译时能判断的是有限的；一旦有另外线程想获取，就会revoke，而且开销明显

张玮(大圣)

2018-06-12

自旋锁类似和忙等待一个套路

作者回复

2018-06-12

嗯，我理解是一个意思

tyson

2018-06-12

简单来说就是while，一直cas直到成功吧。

作者回复

2018-06-12

差不多，也要考虑退出自旋的情况

雷霹雳的爸爸

2018-06-12

今天老师讲这个真够我喝一壶的，而且老师总结的角度启发性很大，最近也再读JCIP，对比起来很有意思，对于自旋锁这个理解，我一直还是蛮肤浅的，顾名思义比较多，就是在那里兜几个圈子——写个循环——试几次，好处是减少线程切换导致的开销，一般也需要有底层有CAS能力的构件支持一下，比如用Atomic开头那些类，当然也未必，比如说nio读不出来东西的时候，也先尝试几次，总之就是暂时不把cpu让度出去，先在占着坑来几次，大概可能这么个意思吧

作者回复

2018-06-12

差不多，算是种乐观主义的"优化"

浩

2018-06-12

自旋就是空转，什么都不干，就在循环等待锁，相当于缓冲一段时间，看能否获得锁，如果此次自旋获得锁，那么下次，会比此次更长时间自旋，增大获得锁的概率，否则，减少自旋次数。

作者回复

2018-06-12

基本正确

极客时间

极客时间