从 JVM 中深入探究 Synchronized

好学的康达姆机... LV.4
2023年01月29日 20:43 · 阅读 4253

开篇语

Synchronized，Java 友好的提供了的一个关键字，它让开发者可以快速的实现同步。它就像一个星星，远远看去就是一个小小的点。但是走近一看，却是一个庞大的蛋糕。而这篇文章就是要将这个巨大的蛋糕切开，吃进肚子里面去。

Synchronized 使用

在 Java 中，如果要实现同步，Java 提供了一个关键词 synchronized 来让开发人员可以快速实现同步代码块。

typescript 复制代码

```
public class Test {

    public static void main(String[] args){
        Object o = new Object();

        Thread thread1 = new Thread(() -> {
            synchronized (o){
                System.out.println("获取锁成功");
            }
        }).start();
    }
}
```

线程 thread1 获取对象 o 的锁，并且输出一句话 “获取锁成功”。

arduino 复制代码

```
public class Test {
```

```
    public synchronized static String get(){
        return "静态方法";
    }

    public void put(){
        synchronized (this){
            System.out.println("同步代码块");
        }
    }
}
```

synchronized 关键字除了可以用于代码块，还可以用于方法上。用于实例方法上时，线程执行该方法之前，会自动获取该对象锁，获取到对象锁之后才会继续执行实例方法中的代码；用于静态方法上时，线程执行该方法之前，会自动获取该对象所属类的锁，获取到类锁之后才会继续执行静态方法中的代码。用于代码块上时，可以传入任意对象作为锁，并且可以控制锁的粒度。

synchronized 实现原理

下面是 Test 类的字节码文件

yaml 复制代码

```
public class Test
  minor version: 0
  major version: 55
  flags: (0x0021) ACC_PUBLIC, ACC_SUPER
  this_class: #7                          // Test
  super_class: #8                         // java/lang/Object
  interfaces: 0, fields: 1, methods: 4, attributes: 1
Constant pool:
   #1 = Methodref          #8.#27         // java/lang/Object."<init>":()V
   #2 = Fieldref           #7.#28         // Test.i:I
   #3 = String             #29            // 静态方法
   #4 = Fieldref           #30.#31        // java/lang/System.out:Ljava/io/PrintStream;
   #5 = String             #32            // 同步代码块
   #6 = Methodref          #33.#34        // java/io/PrintStream.println:(Ljava/lang/String;)V
   #7 = Class              #35            // Test
   #8 = Class              #36            // java/lang/Object
   #9 = Utf8               i
```

```
#15 = Utf8               LocalVariableTable
#16 = Utf8               this
#17 = Utf8               LTest;
#18 = Utf8               set
#19 = Utf8               (I)V
#20 = Utf8               get
#21 = Utf8               ()Ljava/lang/String;
#22 = Utf8               put
#23 = Utf8               StackMapTable
#24 = Class              #37            // java/lang/Throwable
#25 = Utf8               SourceFile
#26 = Utf8               Test.java
#27 = NameAndType        #11:#12        // "<init>":()V
#28 = NameAndType        #9:#10         // i:I
#29 = Utf8               静态方法
#30 = Class              #38            // java/lang/System
#31 = NameAndType        #39:#40        // out:Ljava/io/PrintStream;
#32 = Utf8               同步代码块
#33 = Class              #41            // java/io/PrintStream
#34 = NameAndType        #42:#43        // println:(Ljava/lang/String;)V
#35 = Utf8               Test
#36 = Utf8               java/lang/Object
#37 = Utf8               java/lang/Throwable
#38 = Utf8               java/lang/System
#39 = Utf8               out
#40 = Utf8               Ljava/io/PrintStream;
#41 = Utf8               java/io/PrintStream
#42 = Utf8               println
#43 = Utf8               (Ljava/lang/String;)V
{
  public Test();
    descriptor: ()V
    flags: (0x0001) ACC_PUBLIC
    Code:
      stack=2, locals=1, args_size=1
         0: aload_0
         1: invokespecial #1                  // Method java/lang/Object."<init>":()V
         4: aload_0
         5: iconst_0
         6: putfield      #2                  // Field i:I
         9: return
      LineNumberTable:
        line 5: 0
        line 7: 4
      LocalVariableTable:
```

```
public synchronized void set(int);
  descriptor: (I)V
  flags: (0x0021) ACC_PUBLIC, ACC_SYNCHRONIZED
  Code:
    stack=2, locals=2, args_size=2
       0: aload_0
       1: iload_1
       2: putfield      #2                  // Field i:I
       5: return
    LineNumberTable:
      line 10: 0
      line 11: 5
    LocalVariableTable:
      Start  Length  Slot  Name   Signature
          0       6     0  this   LTest;
          0       6     1     i   I

public static synchronized java.lang.String get();
  descriptor: ()Ljava/lang/String;
  flags: (0x0029) ACC_PUBLIC, ACC_STATIC, ACC_SYNCHRONIZED
  Code:
    stack=1, locals=0, args_size=0
       0: ldc           #3                  // String 静态方法
       2: areturn
    LineNumberTable:
      line 14: 0

public void put();
  descriptor: ()V
  flags: (0x0001) ACC_PUBLIC
  Code:
    stack=2, locals=3, args_size=1
       0: aload_0
       1: dup
       2: astore_1
       3: monitorenter
       4: getstatic     #4                  // Field java/lang/System.out:Ljava/io/PrintStream;
       7: ldc           #5                  // String 同步代码块
       9: invokevirtual #6                  // Method java/io/PrintStream.println:(Ljava/lang/Stri
      12: aload_1
      13: monitorexit
      14: goto          22
      17: astore_2
      18: aload_1
      19: monitorexit
      20: aload_2
```

```
            4    14    17   any
           17    20    17   any
      LineNumberTable:
        line 18: 0
        line 19: 4
        line 20: 12
        line 21: 22
      LocalVariableTable:
        Start  Length  Slot  Name   Signature
            0      23     0  this   LTest;
      StackMapTable: number_of_entries = 2
        frame_type = 255 /* full_frame */
          offset_delta = 17
          locals = [ class Test, class java/lang/Object ]
          stack = [ class java/lang/Throwable ]
        frame_type = 250 /* chop */
          offset_delta = 4
}
```

是通过 ACC_SYNCHRONIZED 这个标志来实现同步的。而作用在代码块时，而且通过指令 monitorenter 和 monitorexit 来实现同步的。monitorenter 是获取锁的指令，monitorexit 则是释放锁的指令。

对象头

通过上文我们已经知道，Java 要实现同步，需要通过获取对象锁。那么在 JVM中，是如何知道哪个线程已经获取到了锁呢?

要解释这个问题，我们首先需要了解一个对象的存储分布由以下三部分组成:

- **对象头（Header）** ：由 **Mark Word** 和 **Klass Pointer** 组成
- **实例数据（Instance Data）** ：对象的成员变量及数据
- **对齐填充（Padding）** ：对齐填充的字节

Mark Word ****记录了对象运行时的数据:

4. **lock 锁状态： 01 无锁/偏向锁；00 轻量级锁；10 重量级锁；11 GC 标志**
5. **偏向线程 ID**

128bit （对象头）	状态			
64bit Mark Word	64bit Klass Poiter			
unused:25	**identity_hashcode:31**	**unused:1**	**age:4**	**biased_lock:1**
threadId:54	**epoch:2**	**unused:1**	**age:4**	**biased_lock:1**
ptr_to_lock_record:62	**lock:2**		轻量级锁	
ptr_to_heavyweight_monitor:62	**lock:2**		重量级锁	
	lock:2		GC 标记	

当线程获取对象锁的时候，需要先通过对象头中的 Mark Word 判断对象锁是否已经被其他线程获取，如果没有，那么线程需要往对象头中写入一些标记数据，用于表示这个对象锁已经被我获取了，其他线程无法再获取到。如果对象锁已经被其他线程获取了，那么线程就需要进入到等待队列中，直到持有锁的线程释放了锁，它才有机会继续获取锁。

当一个线程拥有了锁之后，它便可以多次进入。当然，在这个线程释放锁的时候，那么也需要执行相同次数的释放动作。比如，一个线程先后3次获得了锁，那么它也需要释放3次，其他线程才可以继续访问。这也说明使用 synchronized 获取的锁，都是**可重入锁。**

字节序

我们知道了对象头的内存结构之后，我们还需要了解一个很重要的概念：字节序。它表示每一个字节之间的数据在内存中是如何存放的？如果不理解这个概念，那么在之后打印出对象头时，也会无法跟上述展示的对象头内存结构相互对应上。

> 注意！注意！注意！这里使用了大于，也就是说一个字节内的数据，它的顺序是固定的。

- 大端序（BIG_ENDIAN）：高位字节排在内存的低地址处，低位字节排在内存的高地址处。符合人类的读写顺序
- 小端序（LITTLE_ENDIAN）：高位字节排在内存的高地址处，低位字节排在内存的低地址处。符合计算机的读取顺序

我们来举个例子：

有一个十六进制的数字：0x123456789。

使用大端序阅读：高位字节在前，低位字节在后。

内存地址	**1**	**2**	**3**	**4**
十六进制	0x01	0x23	0x45	0x67
二进制	00000001	00100011	01000101	01100111

使用小端序阅读：低位字节在前，高位字节在后。

内存地址	**1**	**2**	**3**	**4**
十六进制	0x89	0x67	0x45	0x23
二进制	10001001	01100111	01000101	00100011

既然大端序符合人类的阅读习惯，那么统一使用大端序不就好了吗？为什么还要搞出一个小端序来呢？

这是因为计算机都是先从低位开始处理的，这样处理效率比较高，所以计算机内部都是使用小

Java 中的字节序

我们可以通过下面这一段代码打印出 Java 的字节序：

typescript 复制代码

```
public class ByteOrderPrinter {
    public static void main(String[] args){
        System.out.println(ByteOrder.nativeOrder());
    }
}
```

打印的结果为： LITTLE_ENDIAN。

因此，我们可以知道 Java 中的字节序为**小端字节序。**

如何阅读对象头

在理解了字节序之后，我们来看看如何阅读对象头。

首先，我们使用一个第三方类库 jol-core，我使用的是 0.10 版本，帮助我们打印出对象头的数据。

我们可以通过下面这一段代码打印出 Java 的对象头：

csharp 复制代码

```
public class ObjectHeaderPrinter {

    public static void main(String[] args) throws InterruptedException {

        Test test = new Test();

        System.out.println("=====打印匿名偏向锁对象头=====");
        System.out.println(ClassLayout.parseInstance(test).toPrintable());

        synchronized (test){
            System.out.println("=====打印偏向锁对象头=====");
            System.out.println(ClassLayout.parseInstance(test).toPrintable());
        }
    }
```

打印结果如下：

```python
=====打印匿名偏向锁/无锁对象头=====
Test object internals:
 OFFSET  SIZE   TYPE DESCRIPTION                               VALUE
      0     4        (object header)                           05 00 00 00 (00000101 00000000 000000
      4     4        (object header)                           00 00 00 00 (00000000 00000000 000000
      8     4        (object header)                           50 6a 06 00 (01010000 01101010 000001
     12     4    int Test.i                                    0
Instance size: 16 bytes
Space losses: 0 bytes internal + 0 bytes external = 0 bytes total

=====打印偏向锁对象头=====
Test object internals:
 OFFSET  SIZE   TYPE DESCRIPTION                               VALUE
      0     4        (object header)                           05 a0 80 4b (00000101 10100000 100000
      4     4        (object header)                           01 00 00 00 (00000001 00000000 000000
      8     4        (object header)                           50 6a 06 00 (01010000 01101010 000001
     12     4    int Test.i                                    0
Instance size: 16 bytes
Space losses: 0 bytes internal + 0 bytes external = 0 bytes total
```

我们把对象头的内存结构和对象头单独拿出来对照着解释一下：

64bit Mark Word	64bit Klass Poiter			
unused:25	**identity_hashcode: 31**	**unused:1**	**age:4**	**biased_lock:1**
threadId:54	**epoch:2**	**unused:1**	**age:4**	**biased_lock:1**
ptr_to_lock_record: 62	**lock:2**		轻量级锁	
ptr_to_heavyweight_monitor:62	**lock:2**		重量级锁	
	lock:2		GC 标记	

scss 复制代码

```
// 匿名偏向锁/无锁
// 我们给每个字节都标上序号。
                a        b        c        d
05 00 00 00 (00000101 00000000 00000000 00000000) (5)
                e        f        g        h
00 00 00 00 (00000000 00000000 00000000 00000000) (0)
                i        j        k        l
50 6a 06 00 (01010000 01101010 00000110 00000000) (420432)
```

unused：25 位，它实际上的字节应该是：hgf + e 的最高位。

identity_hashcode：31 位，它实际上的字节应该是：e 的低 7 位 + dcb。

unused：1位，它实际上的字节应该是：a 的最高位。

age：4位，它实际上的字节应该是：a的第 4-7 位

biased_lock：1位，它实际上的字节应该是：a的第 3 位

lock：2位，它实际上的字节应该是：a的低 2 位。

hgf + e的最高位	e 的低 7 位 + dcb	a 的最高位	a的第 4-7 位	a的第 3 位
00000000 00000000 00000000 0	0000000 00000000 00000000 00000000	0	0000	1

我们再来看一个加了偏向锁的对象头：

scss 复制代码

```
// 偏向锁
                a        b        c        d
05 90 00 13 (00000101 10010000 00000000 00010011) (318803973)
                e        f        g        h
01 00 00 00 (00000001 00000000 00000000 00000000) (1)
                i        j        k        l
50 6a 06 00 (01010000 01101010 00000110 00000000) (420432)
```

threadId:54	**epoch:2**	**unused:1**	**age:4**	**biased_lock:1**
hgfedc + b 的高 6 位	b的低 2 位	a 的最高位	a的第 4-7 位	a的第 3 位
00000000 00000000 00000000 00000001 00010011 00000000 100100	00	0	0000	1

偏向锁

偏向锁是 Java 为了提高获取锁的效率和降低获取锁的代价，而进行的一个优化。因为 Java 团队发现大多数的锁都只被一个线程获取。基于这种情况，就可以认为锁都只被一个线程获取，那么就不会存在多个线程竞争的条件，因此就可以不需要真正的去获取一个完整的锁。只需要在对象头中写入获取锁的线程 ID，用于表示该对象锁已经被该线程获取。

获取偏向锁，只要修改对象头的标记就可以表示线程已经获取了锁，大大降低了获取锁的代

threadId:54	epoch:2	unused:1	age:4	biased_lock:1	lock:2

threadId：获取了偏向锁的线程 ID

epoch：用于保存偏向时间戳

age：对象 GC 年龄

biased_lock：偏向锁标记，此时为 1

lock：锁标记，此时为 10

获取偏向锁

线程获取对象锁时，首先检查对象锁是否支持偏向锁，即检查 **biased_lock** 是否为 1；如果为 1，那么将会检查**threadId** 是否为 null，如果为 null，将会通过 CAS 操作将自己的线程 ID 写入到对象头中。如果成功写入了线程 ID，那么该线程就获取到了对象的偏向锁，可以继续执行后面的同步代码。

只有匿名偏向的对象才能进入偏向锁模式，即该对象还没有偏向任何一个线程（不是绝对的，存在批量重偏向的情况）。

释放偏向锁

线程是不会主动释放偏向锁的。只有当其它线程尝试竞争偏向锁时，持有偏向锁的线程才会释放偏向锁。

释放偏向锁需要在全局安全点进行。释放的步骤如下：

1. 暂停拥有偏向锁的线程，判断是否处于同步代码块中，如果处于，则进行偏向撤销，并升级为轻量级锁。
2. 如果不处于，则恢复为无锁状态。

偏向撤销主要发生在多个线程存在竞争，不再偏向于任何一个线程了。也就是说偏向撤销之后，将不会再使用偏向锁。具体操作就是将 Mark Work 中的 **biased_lock** 由 1 设置为 0 。偏向撤销需要到达全局安全点才可以撤销，因为它需要修改对象头，并从栈中获取数据。因此偏向撤销也会存在较大的资源消耗。

想要撤销偏向锁，还不能对持有偏向锁的线程有影响，所以就要等待持有偏向锁的线程到达一个 `safepoint 安全点` ,在这个安全点会挂起获得偏向锁的线程。

1. 如果原持有偏向锁的线程依然还在同步代码块中，那么就会将偏向锁升级为轻量级锁。
2. 如果原持有偏向锁的线程已经死亡，或者已经退出了同步代码块，那么直接撤销偏向锁状态即可。

对象的偏向锁被撤销之后，对象在未来将不会偏向于任何一个线程。

批量重偏向

我们可以想象，如果有 100 个对象都偏向于一个线程，此时如果有另外一个线程来获取这些对象的锁，那么这 100 个对象都会发生偏向撤销，而这 100 次偏向撤销都需要在全局安全点下进行，这样就会产生大量的性能消耗。

批量重偏向就是建立在撤销偏向会对性能产生较大影响情况下的一种优化措施。当 JVM 知道有大量对象的偏向锁撤销时，它就知道此时这些对象都不会偏向于原线程，所以会将对象重新偏向于新的线程，从而减少偏向撤销的次数。

当一个类的大量对象被同一个线程 T1 获取了偏向锁，也就是大量对象先偏向于该线程 T1。T1 同步结束后，另一个线程 T2 对这些同一类型的对象进行同步操作，就会让这些对象重新偏向于线程 T2。

> 在了解批量重偏向前，我们需要先了解一点其他知识：
>
> JVM 会给对象的类对象 class 赋予两个属性，一个是偏向撤销计数器，一个是 epoch 值。

我们先来看一个例子：

```
import java.util.ArrayList;
import java.util.List;

/**
*  @author  liuhaidong
*  @date  2023/1/6 15:06
*/
public class ReBiasTest {

    public static void main(String[] args) throws InterruptedException {


        //延时产生可偏向对象
        //默认4秒之后才能进入偏向模式，可以通过参数-XX:BiasedLockingStartupDelay=0设置
        Thread.sleep(5000);

        //创造100个偏向线程t1的偏向锁
        List<Test> listA = new ArrayList<>();
        Thread t1 = new Thread(() -> {
            for (int i = 0; i < 100; i++) {
                Test a = new Test();
                synchronized (a) {
                    listA.add(a);
                }
            }
            try {
                //为了防止JVM线程复用，在创建完对象后，保持线程t1状态为存活
                Thread.sleep(100000);
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
        });
        t1.start();

        //睡眠3s钟保证线程t1创建对象完成
        Thread.sleep(3000);
        System.out.println("打印t1线程，list中第20个对象的对象头：");
        System.out.println((ClassLayout.parseInstance(listA.get(19)).toPrintable()));

        //创建线程t2竞争线程t1中已经退出同步块的锁
        Thread t2 = new Thread(() -> {
            //这里面只循环了30次！！！
            for (int i = 0; i < 30; i++) {
                Test a = listA.get(i);
                synchronized (a) {
```

```
                    if (i == 10) {
                        // 该对象已经是轻量级锁，无法降级，因此只能是轻量级锁
                        System.out.println("第" + (i + 1) + "次偏向结果");
                        System.out.println((ClassLayout.parseInstance(a).toPrintable()));
                    }
                }
            }
            try {
                Thread.sleep(10000);
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
        });
        t2.start();

        Thread.sleep(3000);
        System.out.println("打印list中第11个对象的对象头：");
        System.out.println((ClassLayout.parseInstance(listA.get(10)).toPrintable()));
        System.out.println("打印list中第26个对象的对象头：");
        System.out.println((ClassLayout.parseInstance(listA.get(25)).toPrintable()));
        System.out.println("打印list中第41个对象的对象头：");
        System.out.println((ClassLayout.parseInstance(listA.get(40)).toPrintable()));
    }
}
```

> 在 JDK8 中，-XX:BiasedLockingStartupDelay 的默认值是 4000；在 JDK11 中，-XX:BiasedLockingStartupDelay 的默认值是 0

1. t1 执行完后，100 个对象都会偏向于 t1。
2. t2 执行完毕之后，其中前 19 个对象都会撤销偏向锁，此时类中的偏向撤销计数器为19。但当撤销到第 20 个的时候，偏向撤销计数器为 20，此时达到 `-XX:BiasedLockingBulkRebiasThreshold=20` 的条件，于是将类中的 epoch 值 +1，并在此时找到所有处于同步代码块的对象，并将其 epoch 值等于类对象的 epoch 值。然后进行批量重偏向操作，从第 20 个对象开始，将会比较对象的 epoch 值是否等于类对象的 epoch 值，如果不等于，那么直接使用 CAS 替换掉 Mark Word 中的程 ID 为当前线程的 ID。

- 结论：

2. 第 20 - 30 个对象，依然为偏向锁，偏向于线程 t2。
3. 第 31 - 100 个对象，依然为偏向锁，偏向于线程 t1。

tech.youzan.com/javasuo-yu-...

暂时无法在飞书文档外展示此内容

批量撤销偏向

当偏向锁撤销的数量达到 40 时，就会发生批量撤销。但是，这是在一个时间范围内达到 40 才会发生，这个时间范围通过 `-XX:BiasedLockingDecayTime` 设置，默认值为 25 秒。

也就是在发生批量偏向的 25 秒内，如果偏向锁撤销的数量达到了 40 ，那么就会发生批量撤销，将该类下的所有对象都进行撤销偏向，包括后续创建的对象。如果在发生批量偏向的 25 秒内没有达到 40 ，就会重置偏向锁撤销数量，将偏向锁撤销数量重置为 20。

Hashcode 去哪了

我们通过 Mark Word 知道，在无锁状态下，如果调用对象的 `hashcode()` 方法，就会在 Mark Word 中记录对象的 Hashcode 值，在下一次调用 `hashcode()` 方法时，就可以直接通过 Mark Word 来得知，而不需要再次计算，以此来保证 Hashcode 的一致性。

但是获取了锁之后，就会修改 Mark Word 中的值，那么之前记录下来的 Hashcode 值去哪里了呢?

Lock Record

在解答这个问题之前，我们需要先知道一个东西：Lock Record。

当字节码解释器执行 monitorenter 字节码轻度锁住一个对象时，就会在获取锁的线程栈上显式或者隐式分配一个 Lock Record。换句话说，就是在获取轻量级锁时，会在线程栈上分配一个 Lock Record。这个 Lock Record 说直白一点就是栈上的一块空间，主要用于存储相关信息。

2. 解释器使用 Lock Record 来检测非法的锁状态
3. 隐式地充当锁重入机制的计数器

那么这个 Lock Record 跟 Hashcode 有什么关系呢?

场景 1

我们先来看第一个场景：先获取对象的 hashcode，然后再获取对象的锁。

csharp 复制代码

```
import org.openjdk.jol.info.ClassLayout;

public class TestObject {
    public static void main(String[] args) {
        Test test = new Test();

        // 步骤 1
        System.out.println("=====获取 hashcode 之前=====");
        System.out.println(ClassLayout.parseInstance(test).toPrintable());

        test.hashCode();

        // 步骤 2
        System.out.println("=====获取 hashcode 之后=====");
        System.out.println(ClassLayout.parseInstance(test).toPrintable());

        // 步骤 3
        synchronized (test){
            System.out.println("=====获取锁之后=====");
            System.out.println(ClassLayout.parseInstance(test).toPrintable());
        }

        // 步骤 4
        System.out.println("=====释放锁之后=====");
        System.out.println(ClassLayout.parseInstance(test).toPrintable());
    }
}
```

运行结果:

python 复制代码

```
=====获取 hashcode 之前=====
```

```
      8     4        (object header)                           50 6a 06 00 (01010000 01101010 000001
     12     4    int Test.i                                    0
Instance size: 16 bytes
Space losses: 0 bytes internal + 0 bytes external = 0 bytes total

=====获取 hashcode 之后=====
Test object internals:
 OFFSET  SIZE   TYPE DESCRIPTION                               VALUE
      0     4        (object header)                           01 0c 97 8b (00000001 00001100 100101
      4     4        (object header)                           76 00 00 00 (01110110 00000000 000000
      8     4        (object header)                           50 6a 06 00 (01010000 01101010 000001
     12     4    int Test.i                                    0
Instance size: 16 bytes
Space losses: 0 bytes internal + 0 bytes external = 0 bytes total

=====获取锁之后=====
Test object internals:
 OFFSET  SIZE   TYPE DESCRIPTION                               VALUE
      0     4        (object header)                           90 2a 90 6b (10010000 00101010 100100
      4     4        (object header)                           01 00 00 00 (00000001 00000000 000000
      8     4        (object header)                           50 6a 06 00 (01010000 01101010 000001
     12     4    int Test.i                                    0
Instance size: 16 bytes
Space losses: 0 bytes internal + 0 bytes external = 0 bytes total

=====释放锁之后=====
Test object internals:
 OFFSET  SIZE   TYPE DESCRIPTION                               VALUE
      0     4        (object header)                           01 0c 97 8b (00000001 00001100 100101
      4     4        (object header)                           76 00 00 00 (01110110 00000000 000000
      8     4        (object header)                           50 6a 06 00 (01010000 01101010 000001
     12     4    int Test.i                                    0
Instance size: 16 bytes
Space losses: 0 bytes internal + 0 bytes external = 0 bytes total
```

- 步骤一：未获取对象的 hashcode 值之前，对象处于**匿名偏向锁**状态。锁标记为：101
- 步骤二：获取对象的 hashcode 之后，对象的偏向状态被撤销，处于无锁状态。锁标记为：001。对象头中也存储了 hashcode 值，hashcode 值为 0111011 10001011 10010111 00001100。
- 步骤三：获取锁之后，对象处于轻量级锁状态。锁标记为：00。其余 62 位为指向 Lock Record 的指针。从这里我们可以看到，Mark Word 中已经没有 hashcode 了。整块

JVM 会将 Lock Record 中的值复制回 Mark Word 中，并删除 Lock Record。

结论：

1. 当对象生成 hashcode 之后，会撤销偏向，并将 hashcode 记录在 Mark Word 中。
2. 非偏向的对象获取锁时，会先在栈中生成一个 Lock Record。并将对象的 Mark Word 复制到 Lock Record 中。

场景2

我们现在来看第二个场景：先获取对象的锁，然后在同步代码块中生成 hashcode。

csharp 复制代码

```
import org.openjdk.jol.info.ClassLayout;

public class HashCode2 {
    public static void main(String[] args) {
        Test test = new Test();

        // 步骤一
        System.out.println("=====获取锁之前=====");
        System.out.println(ClassLayout.parseInstance(test).toPrintable());

        synchronized (test){
            // 步骤二
            System.out.println("=====获取锁之后，获取hashcode之前=====");
            System.out.println(ClassLayout.parseInstance(test).toPrintable());

            // 步骤三
            test.hashCode();
            System.out.println("=====获取锁之后，获取hashcode之后=====");
            System.out.println(ClassLayout.parseInstance(test).toPrintable());
        }

        // 步骤四
        System.out.println("=====释放锁之后=====");
        System.out.println(ClassLayout.parseInstance(test).toPrintable());
    }
}
```

运行结果：

```
 OFFSET  SIZE   TYPE DESCRIPTION                               VALUE
      0     4        (object header)                           05 00 00 00 (00000101 00000000 000006
      4     4        (object header)                           00 00 00 00 (00000000 00000000 000006
      8     4        (object header)                           50 6a 06 00 (01010000 01101010 000001
     12     4    int Test.i                                    0
Instance size: 16 bytes
Space losses: 0 bytes internal + 0 bytes external = 0 bytes total

=====获取锁之后，获取hashcode之前=====
Test object internals:
 OFFSET  SIZE   TYPE DESCRIPTION                               VALUE
      0     4        (object header)                           05 90 80 3a (00000101 10010000 100006
      4     4        (object header)                           01 00 00 00 (00000001 00000000 000006
      8     4        (object header)                           50 6a 06 00 (01010000 01101010 000001
     12     4    int Test.i                                    0
Instance size: 16 bytes
Space losses: 0 bytes internal + 0 bytes external = 0 bytes total

=====获取锁之后，获取hashcode之后=====
Test object internals:
 OFFSET  SIZE   TYPE DESCRIPTION                               VALUE
      0     4        (object header)                           02 e8 83 2a (00000010 11101000 100006
      4     4        (object header)                           01 00 00 00 (00000001 00000000 000006
      8     4        (object header)                           50 6a 06 00 (01010000 01101010 000001
     12     4    int Test.i                                    0
Instance size: 16 bytes
Space losses: 0 bytes internal + 0 bytes external = 0 bytes total

=====释放锁之后=====
Test object internals:
 OFFSET  SIZE   TYPE DESCRIPTION                               VALUE
      0     4        (object header)                           02 e8 83 2a (00000010 11101000 100006
      4     4        (object header)                           01 00 00 00 (00000001 00000000 000006
      8     4        (object header)                           50 6a 06 00 (01010000 01101010 000001
     12     4    int Test.i                                    0
Instance size: 16 bytes
Space losses: 0 bytes internal + 0 bytes external = 0 bytes total
```

- 步骤一：未获取对象的 hashcode 值之前，对象处于**匿名偏向锁**状态。锁标记为：101
- 步骤二：进入同步代码块，线程获取了偏向锁。锁标记：101
- 步骤三：对象生成 hashcode，此时锁标记：**10，直接从偏向锁升级为重量级锁。** 其余 62 位为指向 objectMonitor 的指针。

轻量级锁

轻量级锁解决的场景是：任意两个线程交替获取锁的情况。主要依靠 CAS 操作，相比较于使用重量级锁，可以减少锁资源的消耗。

获取轻量级锁

使用轻量级锁的情况有以下几种：

1. 禁用偏向锁。
2. 偏向锁失效，升级为轻量级锁。

禁用偏向锁导致升级

在启动 Java 程序时，如果添加了 JVM 参数 `-XX:-UseBiasedLocking` ， 那么在后续的运行中，就不再使用偏向锁 。

偏向锁失效，升级为轻量级锁

如果对象发生偏向撤销时：

1. 首先会检查持有偏向锁的线程是否已经死亡，如果死亡，则直接升级为轻量级锁，否则，执行步骤2
2. 查看持有偏向锁的线程是否在同步代码块中，如果在，则将偏向锁升级为轻量级锁，否则，执行步骤3
3. 修改 Mark Word 为非偏向模式，设置为无锁状态。

加锁过程

当线程获取轻量级锁时，首先会在线程栈中创建一个 Lock Record 的内存空间，然后拷贝 Mark Word 中的数据到 Lock Record 中。JVM 中将有数据的 Lock Record 叫做 Displated Mark Word。

当数据复制成功之后，JVM 将会使用 CAS 尝试修改 Mark Word 中的数据为指向线程栈中 Displated Mark Word 的指针，并将 Lock Record 中的 owner 指针指向 Mark Word。

如果这两步操作都更新成功了，那么则表示该线程获得轻量级锁成功，设置 Mark Word 中的 lock 字段为 00，表示当前对象为轻量级锁状态。同步，线程可以执行同步代码块。

如果更新操作失败了，那么 JVM 将会检查 Mark Word 是否指向当前线程的栈帧：

- 如果是，则表示当前线程已经获取了轻量级锁，会在栈帧中添加一个新的 Lock Record，这个新 Lock Record 中的 Displated Mark Word 为 null，owner 指向对象。这样的目的是为了统计重入的锁数量，因此，在栈中会有一个 Lock Record 的列表。完成这一步之后就可以直接执行同步代码块。

暂时无法在飞书文档外展示此内容

- 如果不是，那么表示轻量级锁发生竞争，后续将会膨胀为重量级锁。

释放轻量级锁

释放轻量级锁时，会在栈中由低到高，获取 Lock Record。查询到 Lock Record 中的 Displated Mark Word 为 null 时，则表示，该锁是重入的，只需要将 owner 设置为 null 即可，表示已经释放了这个锁。如果 Displated Mark Word 不为 null，则需要通过 CAS 将 Displated Mark Word 拷贝至对象头的 Mark Word 中，然后将 owner 的指针设置为 null，最后修改 Mark Word 的 lock 字段为 01 无锁状态。

重量级锁

重量级锁解锁的场景是：多个线程相互竞争同一个锁。主要通过 `park()` 和 `unpark()` 方法，结合队列来完成。相较于轻量级锁和偏向锁，需要切换内核态和用户态环境，因此获取锁的过程会消耗较多的资源。

获取重量级锁

获取 hashcode，升级为重量级锁

csharp 复制代码

```
import org.openjdk.jol.info.ClassLayout;

public class HashCode2 {
    public static void main(String[] args) {
        Test test = new Test();

        // 步骤一
        System.out.println("=====获取锁之前=====");
        System.out.println(ClassLayout.parseInstance(test).toPrintable());

        synchronized (test){
            // 步骤二
            System.out.println("=====获取锁之后，获取hashcode之前=====");
            System.out.println(ClassLayout.parseInstance(test).toPrintable());

            // 步骤三
            test.hashCode();
            System.out.println("=====获取锁之后，获取hashcode之后=====");
            System.out.println(ClassLayout.parseInstance(test).toPrintable());
        }
    }
}
```

执行后的结果

python 复制代码

```
=====获取锁之前=====
Test object internals:
 OFFSET  SIZE   TYPE DESCRIPTION                               VALUE
      0     4        (object header)                           05 00 00 00 (00000101 00000000 000000
      4     4        (object header)                           00 00 00 00 (00000000 00000000 000000
      8     4        (object header)                           50 6a 06 00 (01010000 01101010 000001
     12     4    int Test.i                                    0
Instance size: 16 bytes
Space losses: 0 bytes internal + 0 bytes external = 0 bytes total

=====获取锁之后，获取hashcode之前=====
Test object internals:
 OFFSET  SIZE   TYPE DESCRIPTION                               VALUE
      0     4        (object header)                           05 90 80 3a (00000101 10010000 100000
```

```
Space losses: 0 bytes internal + 0 bytes external = 0 bytes total

=====获取锁之后，获取hashcode之后=====
Test object internals:
 OFFSET  SIZE   TYPE DESCRIPTION                               VALUE
      0     4        (object header)                           02 e8 83 2a (00000010 11101000 100000
      4     4        (object header)                           01 00 00 00 (00000001 00000000 000000
      8     4        (object header)                           50 6a 06 00 (01010000 01101010 000001
     12     4    int Test.i                                    0
Instance size: 16 bytes
Space losses: 0 bytes internal + 0 bytes external = 0 bytes total
```

我们直接在偏向锁的同步代码块中执行 `hashcode()` ，会发现偏向锁直接膨胀为重量级锁了。我们可以看到 lock 字段为 10。

> 这里有一个疑问，为什么不是升级为轻量级锁呢？轻量级锁也可以在 Lock Record 中存储生成的 hashcode。而膨胀为更为消耗资源的重量级锁。

轻量级锁膨胀为重量级锁

当处于轻量级锁的时候，说明锁已经不再偏向于任何一个线程，但是也没有发生竞争，可以依靠 CAS 获取到轻量级锁。但是当出现 CAS 获取锁失败时，就会直接膨胀为重量级锁。

这里需要注意，只会 CAS 一次，只要一次失败就会直接膨胀为重量级锁，而不是达到自旋次数或者自旋时间才膨胀。

膨胀过程

在膨胀过程中，会有几种标记来表示锁的状态：

- Inflated：膨胀已完成
- Stack-locked：轻量级锁
- INFLATING：膨胀中
- Neutral：无锁

膨胀步骤：

3. 检查是否为轻量级锁，如果是，则执行以下步骤：

    1. 创建一个 ObjectMonitor 对象。
    2. 通过 CAS 设置 Mark Word 为全 0，用以表示 INFLATING 状态。如果失败，则从步骤 1 重新开始执行。
    3. 将 Mark Word 设置到 ObjectMonitor 对象中。
    4. 设置 owner 属性为 Lock Record
    5. 设置 Mark Word 值
    6. 返回

4. 判定为无锁状态，执行以下步骤：

    1. 创建一个 ObjectMonitor 对象。
    2. 通过 CAS 直接设置 Mark Word 值。
    3. 返回

竞争锁过程

我们要理解如何获取重量级锁，需要先了解 ObjectMonitor 对象。顾名思义，这是一个对象监视器。在 Java 中，每个对象都有一个与之对应的 ObjectMonitor 。ObjectMonitor 内部有几个重要的字段：

- cxq：存放被阻塞的线程
- EntryList：存放被阻塞的线程，在释放锁时使用
- WaitSet：获得锁的线程，如果调用 wait() 方法，那么线程会被存放在此处，这是一个双向循环链表
- onwer：持有锁的线程

> cxq，EntryList 均为 ObjectWaiter 类型的单链表。

获取锁过程

1. 通过 CAS 设置 onwer 为当前线程（尝试获取锁），CAS 的原值为 NULL，新值为

3. 判断当前线程是否为之前持有轻量级锁的线程，如果是，直接设置 onwer 为当前线程，表示获得锁。否则执行步骤 4

4. 以上步骤都失败，则尝试一轮自旋来获取锁。如果未获取锁，则执行步骤 5

    1.

5. 使用阻塞和唤醒来控制线程竞争锁

    1. 通过 CAS 设置 owner 为当前线程（尝试获取锁），CAS 的原值为 NULL，新值为 current_thread。如果成功，则表示获得锁。否则执行步骤 b

    2. 通过 CAS 设置 owner 为当前线程（尝试获取锁）CAS 的原值为 DEFLATER_MARKER，新值为 current_thread。如果成功，则表示获得锁。否则执行步骤c。（DEFLATER_MARKER 是一个锁降级的标记，后续会讲解。）

    3. 以上步骤都失败，则尝试一轮自旋来获取锁。如果未获取锁，则执行步骤 d。

    4. 为当前线程创建一个 ObjectWaiter 类型的 node 节点。步骤 i 和 ii 是一个循环，直到一个成功才会跳出这个循环。

        1. 通过 cas 插入 cxq 的头部，如果插入失败，则执行步骤 ii
        2. 通过 CAS 设置 owner 为当前线程（尝试获取锁），CAS 的原值为 NULL，新值为 current_thread。如果失败，则执行 i。

    5. 通过 CAS 设置 owner 为当前线程（尝试获取锁），CAS 的原值为 NULL，新值为 current_thread。如果成功，则表示获得锁。否则执行步骤 f。（该步骤往下开始是一个循环，直到获取到锁为止）

    6. 通过 park()，将线程阻塞。

    7. 线程被唤醒后

        1. 通过 CAS 设置 owner 为当前线程（尝试获取锁），CAS 的原值为 NULL，新值为 current_thread。如果成功，则表示获得锁。否则执行步骤 ii
        2. 通过 CAS 设置 owner 为当前线程（尝试获取锁），CAS 的原值为

> 自适应自旋锁主要是用于重量级锁中，降低阻塞线程概率。而不是用于轻量级锁，这里大家要多多注意。

释放重量级锁

释放锁过程

1. 判断 _owner 字段是否等于 current_thread。如果等于则判断当前线程是否为持有轻量级锁的线程，如果是的话，表示该线程还没有执行 `enter()` 方法，因此，直接设置 _owner 字段为 current_thread。

2. 判断 _recursions，如果大于0，则表示锁重入，直接返回即可，不需要执行后续解锁代码。

3. 设置 _owner 字段为 NULL，解锁成功，后续线程可以正常获取到锁。

4. 唤醒其他正在被阻塞的线程。**在执行以下操作之前需要使用该线程重新获取锁。如果获取锁失败，则表示锁已经被其他线程获取，直接返回，不再唤醒其他线程。（为什么还要获取到锁才可以唤醒其他线程呢？因为唤醒线程时，需要将 cxq 中的节点转移到 EntryList 中，涉及到链表的移动，如果多线程执行，将会出错。）**

    1. 如何 _EntryList 非空，那么取 _EntryList 中的第一个元素，将该元素下的线程唤醒。否则执行步骤 b。

    2. 将 _cxq 设置为空，并将 _cxq 的元素按照原顺序放入 _EntryList 中。然后取 _EntryList 中的第一个元素，将该元素下的线程唤醒。

    3. 线程唤醒

        1. 设置 _owner 字段为 NULL，解锁成功，让后续线程可以正常获取到锁。
        2. 然后调用 `unpark()` 方法，唤醒线程。

我们先来看一个例子：

scss 复制代码

```
public class WaitTest {
    static final Object lock = new Object();

    public static void main(String[] args) {

        new Thread(() -> {
            synchronized (lock){
                log("get lock");
                try {
                    log("wait lock");
                    lock.wait();
                } catch (InterruptedException e) {
                    throw new RuntimeException(e);
                }
                log("get lock again");
                log("release lock");
            }
        }, "thread-A").start();

        sleep(1000);

        new Thread(() -> {
            synchronized (lock){
                log("get lock");
                createThread("thread-C");
                sleep(2000);
                log("start notify");
                lock.notify();
                log("release lock");
            }
        }, "thread-B").start();
    }

    public static void createThread(String threadName) {
        new Thread(() -> {
            synchronized (lock){
                log("get lock");
                log("release lock");
            }
        }, threadName).start();
    }
```

```
            e.printStackTrace();
        }
    }
    private static void log(String desc){
        System.out.println(Thread.currentThread().getName() + " : " + desc);
    }
}
```

最后打印的结果：

arduino 复制代码

```
thread-A : get lock
thread-A : wait lock
thread-B : get lock
thread-B : start notify
thread-B : release lock
thread-A : get lock again
thread-A : release lock
thread-C : get lock
thread-C : release lock
```

1. 线程 A 首先获取到锁，然后通过 `wait()` 方法，将锁释放，并且等待通知。
2. 睡眠 1 S，这里是确保线程 A 可以顺利完成所有操作。
3. 因为 A 释放了锁，所以线程 B 可以获取到锁。然后创建了线程 C。
4. 因为线程 B 睡眠了 2S，依然持有锁，所以线程 C 无法获取到锁，只能继续等待。
5. 线程 B 调用 `notify()` 方法，线程 A 被唤醒，开始竞争锁。
6. 线程 A 和线程 C 竞争锁。

但是根据打印结果，无论执行多少次，都是线程 A 先获取锁。

第一个问题：为什么都是线程 A 先获取锁，而不是线程 C 先获取锁?

第二个问题：为什么 wait 方法并没有生成 monitorenter 指令，也可以获取到锁?

第三个问题：执行 wait 之后，线程去哪里了？它的状态是什么?

为了解答这些问题，我们需要深入到源码中去。但是这里就不放源码了，我只讲一下关键步骤：

3. 将 node 放入 _waitSet 中
4. 释放锁
5. 通过 park() 阻塞 current_thread。

notify()

1. 检查 _waitSet 是否为 null，如果为 null，直接返回
2. 获取 _waitSet 的第一个元素 node，并将其从链表中移除。
3. 此时，存在三个策略：默认使用 policy = 2
    1. 插入到 EntryList 的头部（policy = 1）
    2. 插入到 EntryList 的尾部（policy = 0）
    3. 插入到 cxq 的 头部（policy = 2）
4. 将 node 插入到 cxq 的头部。

notifyAll()

1. 循环检测 _waitSet 是否不为空
    1. 如果不为空，则执行 **notify()** 的步骤。
    2. 否则返回

第一个问题：执行 wait 之后，线程去哪里了？它的状态是什么?

线程 A 调用 wait() 方法后，线程 A 就被 park 了，并被放入到 _waitSet 中。此时他的状态就是 WAITING。如果它从 _waitSet 移除，并被放入到 cxq 之后，那么他的状态就会变为 BLOCKED。如果它竞争到锁，那么他的状态就会变为 RUNNABLE 。

第二个问题：为什么 wait 方法并没有生成 monitorenter 指令，也可以获取到锁?

线程 A 调用 wait() 方法后，线程 A 被放入到 _waitSet 中。直到有其他线程调用 notify() 之后，线程 A 从 _waitSet 移除，并放入到 cxq 中。

移除，放入到 cxq 的头部。因此目前 cxq 的链表结构为：A -> C -> null。接着线程 B 释放锁，会将 cxq 中的元素按照原顺序放入到 EntryList 中，因此目前 cxq 链表结构为：null；EntryList 链表结构为：A -> C -> null。然后唤醒 EntryList 中的第一个线程。

所以，每次都是线程 A 先获取锁。

分类： 后端 标签： Java 后端 源码

评论

全部评论 2

最新 最热

牧子与羊 JY.3 Java开发攻城狮 @ 哪... 1月前

东西是好东西。感觉自己快卷不动了

1 回复

1374685986 JY.4 1月前

量大管饱

1 回复

相关推荐

7月前 后端 Java 源码

多年以后【PageHelper】又深深的给我上了一课！！

1.7w 95 43

6年前 后端 Java 架构

Spring Boot 开箱即用，内藏玄机

6681 79 2

4年前 Java

Java并发编程：synchronized和锁优化

4939 99 2

8月前 后端 Java 源码

Fastjson2你开始使用了吗？来看看源码解析

2.3w 68 38

3年前 Java

Synchronized解析——如果你愿意一层一层剥开我的心

53 2 收藏

2.3w 690 22

25天前 Java 后端 源码

从实现到原理，聊聊Java中的SPI动态扩展

1897 10 2

4年前 Java 后端 编译器

再有人问你synchronized是什么，就把这篇文章发给他。

7421 306 8

3年前 Java

既然synchronized是"万能"的，为什么还需要volatile呢?

1.6w 282 39

5年前 API Java 后端

并发编程的锁机制：synchronized和lock

2.0w 150 6

4年前 Java

再有人问你synchronized是什么，就把这篇文章发给他。

630 24 评论

1年前 后端 Java

(二)彻底理解Java并发编程之Synchronized关键字实现原理剖析

2790 49 70

5年前 API Java 后端

Java并发编程序列之JUC底层AQS(二)

1245 41 2

2年前 JVM

JVM synchronized锁实现原理，看完还不懂算我输！！！

687 点赞 2

3085 18 评论

4年前 Android Java

从 synchronized 到 CAS 和 AQS - 彻底弄懂 Java 各种并发锁

1.5w 191 5

8月前 iOS Xcode

21-探究iOS底层原理|多线程技术【了解iOS中的10个线程锁,与线程锁类型：自旋锁、互斥...

2318 4 评论

2年前 Linux

两个线程，两个互斥锁，怎么形成一个死循环?

889 点赞 评论

1月前 后端 源码 Java

我用笨办法啃下了一个开源项目的源码!

3060 5 1

2年前 Java

我去，你竟然还不会用 synchronized

3380 22 11

神奇的命令行 6年前 后端 Java

死磕 Java 并发 - 深入分析 synchronized 的实现原理

4830 133 1

8小时_2020 3年前 Go

图解Go里面的互斥锁mutex了解编程语言核心实现源码

1145 2 评论

磊叔的技术博客 1年前 后端 Java 源码

log4j2 日志 PatternLayout 配置对 SOFAArk PluginClassLoader 的影响

2407 7 评论

53 2 收藏

1902 22 评论

码农阿宝 4年前 Java

深入理解synchronized关键字

5769 19 2

无聊夫斯基 4年前 面试 后端 Java

从对象头出发了解Synchronized关键字

3796 44 7

Java3y 2年前 Java Java EE

上海某小公司面试题：synchronized锁原理

8828 82 23

Java3y 4年前 后端 Java 源码

Object对象你真理解了吗?

3338 68 5

字母哥哥 2年前 Spring Boot Spring Java

图解进程线程、互斥锁与信号量-看完不懂你来打我

1807 18 7

机器铃砍菜刀 1年前 Go 后端

Go精妙的互斥锁设计

807 1 评论

干货满满张哈希 2年前 Java JVM

JVM相关 - 深入理解 System.gc()

4.4w 21 评论

用嘴写代码 1年前 Java

jvm是怎么做到实现synchronized的?

130 点赞 评论

1.2w 166 25

wenqi42804 4年前 算法 后端 Java

解析Arrays中sort方法的黑科技

2123 22 评论

GDCoder 1年前 iOS

你了解多线程自旋锁、互斥锁、递归锁等锁吗?

2695 14 5

vvsuperman 4年前 后端 Java 源码

JDK源码中的一些“小技巧”

1941 63 4

暮色妖娆丶 1月前 Java 源码 后端

线程局部变量的实现 ThreadLocal

1980 21 4

七淅在学Java 4年前 后端 Java 微信

interrupt(),interrupted() 和 isInterrupted() 的区别

4132 16 1

慕容千语 4年前 源码 后端 Java

干货 | Java 读写锁 ReentrantReadWriteLock 源码分析

3416 50 9

lengendCoder 5年前 Android Java

带你揭开 synchronized 同步机制的神秘面纱

557 21 2

marvin_wjs 2年前 Java

并发[1] - Synchronized的实现原理

376 1 2

2315 23 5

Zack说码 4年前 后端 源码 Spring

从源码入手，一文带你读懂Spring AOP面向切面编程

7052 136 3

胖滚猪学编程 2年前 Java

【漫画】互斥锁ReentrantLock不好用？试试读写锁ReadWriteLock

1557 12 评论

SwiftGG翻译组 4年前 Apple iOS Objective-C

构建一个 @synchronized

1012 16 2

芋道源码_芳芳 5年前 后端 开源 Java

RocketMQ源码解析：定时消息与消息重试

2080 49 6

YKamh 4年前 Android Java

Java基础——Synchronized用法

6945 7 1

做个好人君 4年前 后端 Java JVM

死磕Synchronized底层实现--概论

1.4w 101 8

做个好人君 4年前 后端 Java C++

死磕Synchronized底层实现--偏向锁

7156 33 5

LeopPro 5年前 Java 后端 安全

小豹子带你看源码：Java 线程池（三）提交任务

2306 47 7

2119 点赞 评论

knock_小新 4年前 Java 源码 后端

Java并发（8）- 读写锁中的性能之王：StampedLock

6217 84 4

keithl 3年前 Java

synchronized工作原理(二)

334 点赞 评论

xcgg 3年前 iOS

iOS-锁-@synchronized

4130 1 2

SKjin 3年前 Java

【Java并发】synchronized

1354 5 2

零壹技术栈 4年前 后端 架构 微服务

实战Spring Boot 2.0系列(五) - Listener, Servlet, Filter和Interceptor

1.1w 74 1