liuxiaopeng



# Java并发编程：Synchronized及其实现原理

Java并发编程系列：

- Java 并发编程：核心理论
- Java并发编程：Synchronized及其实现原理
- Java并发编程：Synchronized底层优化（轻量级锁、偏向锁）
- Java 并发编程：线程间的协作(wait/notify/sleep/yield/join)
- Java 并发编程：volatile的使用及其原理

## 一、Synchronized的基本使用

Synchronized是Java中解决并发问题的一种最常用的方法，也是最简单的一种方法。Synchronized的作用主要有三个：（1）确保线程互斥的访问同步代码（2）保证共享变量的修改能够及时可见（3）有效解决重排序问题。从语法上讲，Synchronized总共有三种用法：

（1）修饰普通方法

（2）修饰静态方法

（3）修饰代码块

接下来我就通过几个例子程序来说明一下这三种使用方式（为了便于比较，三段代码除了Synchronized的使用方式不同以外，其他基本保持一致）。

1、没有同步的情况：

代码段一：

```
1 package com.paddx.test.concurrent;
2
3 public class SynchronizedTest {
4     public void method1(){
5         System.out.println("Method 1 start");
6         try {
7             System.out.println("Method 1 execute");
8             Thread.sleep(3000);
9         } catch (InterruptedException e) {
10            e.printStackTrace();
11        }
12        System.out.println("Method 1 end");
13    }
14
15    public void method2(){
16        System.out.println("Method 2 start");
17        try {
```

公告

昵称：liuxiaopeng
园龄：1年6个月
粉丝：110
关注：2
+加关注

< 2017年4月

| 日 | 一 | 二 | 三 | 四 |
|---|---|---|---|---|
| 26 | 27 | 28 | 29 | 30 |
| 2 | 3 | 4 | 5 | 6 |
| 9 | 10 | 11 | 12 | 13 |
| 16 | 17 | 18 | 19 | 20 |
| 23 | 24 | 25 | 26 | 27 |
| 30 | 1 | 2 | 3 | 4 |

搜索

最新随笔

1. Java集合类：AbstractC
码解析
2. Java集合：整体结构
3. Java 并发编程：volatile
理
4. Java 并发编程：线程间的
otify/sleep/yield/join)
5. Java并发编程：Synchro
化（偏向锁、轻量级锁）

```
            System.out.println("Method 2 execute");
            Thread.sleep(1000);
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
        System.out.println("Method 2 end");
    }

    public static void main(String[] args) {
        final SynchronizedTest test = new SynchronizedTest();

        new Thread(new Runnable() {
            @Override
            public void run() {
                test.method1();
            }
        }).start();

        new Thread(new Runnable() {
            @Override
            public void run() {
                test.method2();
            }
        }).start();
    }
}
```

执行结果如下，线程1和线程2同时进入执行状态，线程2执行速度比线程1快，所以线程2先执行完成，这个过程中线程1和线程2是同时执行的。

```
Method 1 start
Method 1 execute
Method 2 start
Method 2 execute
Method 2 end
Method 1 end
```

2、对普通方法同步：

代码段二：

```
package com.paddx.test.concurrent;

public class SynchronizedTest {
    public synchronized void method1(){
        System.out.println("Method 1 start");
        try {
            System.out.println("Method 1 execute");
            Thread.sleep(3000);
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
        System.out.println("Method 1 end");
    }

    public synchronized void method2(){
        System.out.println("Method 2 start");
        try {
            System.out.println("Method 2 execute");
            Thread.sleep(1000);
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
        System.out.println("Method 2 end");
    }

    public static void main(String[] args) {
```

6. Java并发编程：Synchro
现原理
7. Java 并发编程：核心理论
8. 通过反编译深入理解Java
ern
9. Java8内存模型—永久代(
和元空间(Metaspace)
10. 从字节码层面看"HelloW

我的标签
Java(8)
并发编程(4)
集合框架(1)
类加载器(1)
内存模型(1)
双亲委派(1)
字节码(1)
Java 8(1)
JVM(1)
Lambda(1)
更多
随笔档案
2016年6月 (1)
2016年5月 (3)
2016年4月 (4)
2016年3月 (3)
积分与排名

```
        final SynchronizedTest test = new SynchronizedTest();

        new Thread(new Runnable() {
            @Override
            public void run() {
                test.method1();
            }
        }).start();

        new Thread(new Runnable() {
            @Override
            public void run() {
                test.method2();
            }
        }).start();
    }
}
```

执行结果如下，跟代码段一比较，可以很明显的看出，线程2需要等待线程1的method1执行完成才能开始执行method2方法。

```
Method 1 start
Method 1 execute
Method 1 end
Method 2 start
Method 2 execute
Method 2 end
```

3、静态方法（类）同步

代码段三：

```
package com.paddx.test.concurrent;

public class SynchronizedTest {
    public static synchronized void method1(){
        System.out.println("Method 1 start");
        try {
            System.out.println("Method 1 execute");
            Thread.sleep(3000);
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
        System.out.println("Method 1 end");
    }

    public static synchronized void method2(){
        System.out.println("Method 2 start");
        try {
            System.out.println("Method 2 execute");
            Thread.sleep(1000);
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
        System.out.println("Method 2 end");
    }

    public static void main(String[] args) {
        final SynchronizedTest test = new SynchronizedTest();
        final SynchronizedTest test2 = new SynchronizedTest();

        new Thread(new Runnable() {
            @Override
            public void run() {
                test.method1();
            }
        }).start();

        new Thread(new Runnable() {
```

```
            @Override
            public void run() {
                test2.method2();
            }
        }).start();
    }
}
```

执行结果如下，对静态方法的同步本质上是对类的同步（静态方法本质上是属于类的方法，而不是对象上的方法），所以即使test和test2属于不同的对象，但是它们都属于SynchronizedTest类的实例，所以也只能顺序的执行method1和method2，不能并发执行。

```
Method 1 start
Method 1 execute
Method 1 end
Method 2 start
Method 2 execute
Method 2 end
```

4、代码块同步

代码段四：

```
package com.paddx.test.concurrent;

public class SynchronizedTest {
    public void method1(){
        System.out.println("Method 1 start");
        try {
            synchronized (this) {
                System.out.println("Method 1 execute");
                Thread.sleep(3000);
            }
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
        System.out.println("Method 1 end");
    }

    public void method2(){
        System.out.println("Method 2 start");
        try {
            synchronized (this) {
                System.out.println("Method 2 execute");
                Thread.sleep(1000);
            }
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
        System.out.println("Method 2 end");
    }

    public static void main(String[] args) {
        final SynchronizedTest test = new SynchronizedTest();

        new Thread(new Runnable() {
            @Override
            public void run() {
                test.method1();
            }
        }).start();

        new Thread(new Runnable() {
            @Override
            public void run() {
                test.method2();
            }
        }).start();
```

2. Java8内存模型—永久代(和元空间(Metaspace)(858:

3. Java并发编程：Synchro化（偏向锁、轻量级锁）(78

4. Java 并发编程：volatile理(7388)

5. Java 并发编程：核心理论

评论排行榜

1. Java并发编程：Synchro现原理(12)

2. Java 并发编程：volatile理(11)

3. Java 并发编程：线程间的otify/sleep/yield/join)(10

4. 从字节码层面看"HelloWc

5. 通过反编译深入理解Java ern(8)

推荐排行榜

1. 从字节码层面看"HelloWc

2. Java 并发编程：核心理论

3. Java并发编程：Synchro现原理(13)

4. Java集合类：AbstractC码解析(10)

5. Java 并发编程：线程间的otify/sleep/yield/join)(10

```
46     }
47 }
```

执行结果如下，虽然线程1和线程2都进入了对应的方法开始执行，但是线程2在进入同步块之前，需要等待线程1中同步块执行完成。

```
Method 1 start
Method 1 execute
Method 2 start
Method 1 end
Method 2 execute
Method 2 end
```

## 二、Synchronized 原理

如果对上面的执行结果还有疑问，也先不用急，我们先来了解Synchronized的原理，再回头上面的问题就一目了然了。我们先通过反编译下面的代码来看看Synchronized是如何实现对代码块进行同步的：

```
1 package com.paddx.test.concurrent;
2
3 public class SynchronizedDemo {
4     public void method() {
5         synchronized (this) {
6             System.out.println("Method 1 start");
7         }
8     }
9 }
```

反编译结果：

```
liuxpdeMacBook-Pro:classes liuxp$ javap -c com.paddx.test.concurrent.SynchronizedDemo
Compiled from "SynchronizedDemo.java"
public class com.paddx.test.concurrent.SynchronizedDemo {
  public com.paddx.test.concurrent.SynchronizedDemo();
    Code:
       0: aload_0
       1: invokespecial #1                  // Method java/lang/Object."<init>":()V
       4: return

  public void method();
    Code:
       0: aload_0
       1: dup
       2: astore_1
       3: monitorenter
       4: getstatic     #2                  // Field java/lang/System.out:Ljava/io/PrintStream;
       7: ldc           #3                  // String Method 1 start
       9: invokevirtual #4                  // Method java/io/PrintStream.println:(Ljava/lang/String;)V
      12: aload_1
      13: monitorexit
      14: goto          22
      17: astore_2
      18: aload_1
      19: monitorexit
      20: aload_2
      21: athrow
      22: return
```

关于这两条指令的作用，我们直接参考JVM规范中描述：

monitorenter ：

```
Each object is associated with a monitor. A monitor is locked if and only if it has an owner. The thread that executes
monitorenter attempts to gain ownership of the monitor associated with objectref, as follows:
• If the entry count of the monitor associated with objectref is zero, the thread enters the monitor and sets its entry
count to one. The thread is then the owner of the monitor.
• If the thread already owns the monitor associated with objectref, it reenters the monitor, incrementing its entry
count.
• If another thread already owns the monitor associated with objectref, the thread blocks until the monitor's entry
count is zero, then tries again to gain ownership.
```

这段话的大概意思为：

每个对象有一个监视器锁（monitor）。当monitor被占用时就会处于锁定状态，线程执行monitorenter指令时尝试获取monitor的所有权，过程如下：

1、如果monitor的进入数为0，则该线程进入monitor，然后将进入数设置为1，该线程即为monitor的所有者。

2、如果线程已经占有该monitor，只是重新进入，则进入monitor的进入数加1.

3.如果其他线程已经占用了monitor，则该线程进入阻塞状态，直到monitor的进入数为0，再重新尝试获取monitor的所有权。

monitorexit：

```
The thread that executes monitorexit must be the owner of the monitor associated with the instance referenced by objectref.
The thread decrements the entry count of the monitor associated with objectref. If as a result the value of the entry count is zero, the thread exits the monitor and is no longer its owner. Other threads that are blocking to enter the monitor are allowed to attempt to do so.
```

这段话的大概意思为：

执行monitorexit的线程必须是objectref所对应的monitor的所有者。

指令执行时，monitor的进入数减1，如果减1后进入数为0，那线程退出monitor，不再是这个monitor的所有者。其他被这个monitor阻塞的线程可以尝试去获取这个 monitor 的所有权。

通过这两段描述，我们应该能很清楚的看出Synchronized的实现原理，Synchronized的语义底层是通过一个monitor的对象来完成，其实wait/notify等方法也依赖于monitor对象，这就是为什么只有在同步的块或者方法中才能调用wait/notify等方法，否则会抛出java.lang.IllegalMonitorStateException的异常的原因。

我们再来看一下同步方法的反编译结果：

源代码：

```
1 package com.paddx.test.concurrent;
2
3 public class SynchronizedMethod {
4     public synchronized void method() {
5         System.out.println("Hello World!");
6     }
7 }
```

反编译结果：

```
public synchronized void method();
  descriptor: ()V
  flags: ACC_PUBLIC, ACC_SYNCHRONIZED
  Code:
    stack=2, locals=1, args_size=1
       0: getstatic     #2                  // Field java/lang/System.out:Ljava/io/PrintStream;
       3: ldc           #3                  // String Hello World!
       5: invokevirtual #4                  // Method java/io/PrintStream.println:(Ljava/lang/String;)V
       8: return
    LineNumberTable:
      line 5: 0
      line 6: 8
    LocalVariableTable:
      Start  Length  Slot  Name   Signature
          0       9     0  this   Lcom/paddx/test/concurrent/SynchronizedMethod;
```

从反编译的结果来看，方法的同步并没有通过指令monitorenter和monitorexit来完成（理论上其实也可以通过这两条指令来实现），不过相对于普通方法，其常量池中多了ACC_SYNCHRONIZED标示符。JVM就是根据该标示符来实现方法的同步的：当方法调用时，调用指令将会检查方法的 ACC_SYNCHRONIZED 访问标志是否被设置，如果设置了，执行线程将先获取monitor，获取成功之后才能执行方法体，方法执行完后再释放monitor。在方法执行期间，其他任何线程都无法再获得同一个monitor对象。 其实本质上没有区别，只是方法的同步是一种隐式的方式来实现，无需通过字节码来完成。

## 三、运行结果解释

有了对Synchronized原理的认识，再来看上面的程序就可以迎刃而解了。

1、代码段2结果：

虽然method1和method2是不同的方法，但是这两个方法都进行了同步，并且是通过同一个对象去调用的，所以调用之前都需要先去竞争同一个对象上的锁（monitor），也就只能互斥的获取到锁，因此，method1和method2只能顺序的执行。

2、代码段3结果：

虽然test和test2属于不同对象，但是test和test2属于同一个类的不同实例，由于method1和method2都属于静态同步方法，所以调用的时候需要获取同一个类上monitor（每个类只对应一个class对象），所以也只能顺序的执行。

3、代码段4结果：

对于代码块的同步实质上需要获取Synchronized关键字后面括号中对象的monitor，由于这段代码中括号的内容都是this，而method1和method2又是通过同一的对象去调用的，所以进入同步块之前需要去竞争同一个对象上的锁，因此只能顺序执行同步块。

## 四 总结

Synchronized是Java并发编程中最常用的用于保证线程安全的方式，其使用相对也比较简单。但是如果能够深入了解其原理，对监视器锁等底层知识有所了解，一方面可以帮助我们正确的使用Synchronized关键字，另一方面也能够帮助我们更好的理解并发编程机制，有助我们在不同的情况下选择更优的并发策略来完成任务。对平时遇到的各种并发问题，也能够从容的应对。



**标签：** Java， 并发编程





« 上一篇：Java 并发编程：核心理论
» 下一篇：Java并发编程：Synchronized底层优化（偏向锁、轻量级锁）

posted @ 2016-04-19 07:46 liuxiaopeng 阅读(9910) 评论(12) 编辑 收藏

评论列表

#1楼 2016-04-19 08:10 whthomas

赞，喜欢这种深入探究的博文。

支持(1) 反对(0)

#2楼[楼主] 2016-04-19 09:27 liuxiaopeng

@ whthomas
😊，探究一下还是很有好处的~

支持(0) 反对(0)

#3楼 2016-04-19 12:35 那只是一股逆流

好文要顶

支持(0) 反对(0)

#4楼 2016-06-06 03:13 酸酸酸奶

赞， 是并发开速入门的文章惹。但是还是自己多看书会理解的更多惹

支持(0) 反对(0)

#5楼 2016-08-19 10:00 dracularking

ACC_SYNCHRONIZED中的ACC應該是ACCESS吧？

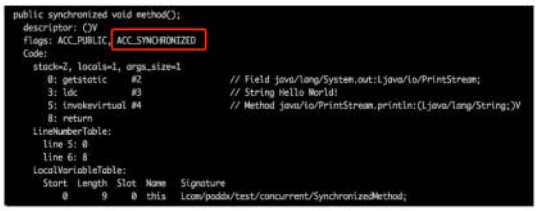
支持(0) 反对(0)

#6楼 2017-01-16 11:34 Dreamer-1

@ 酸酸酸奶
吓 博客园都能遇到巴黎的么 #滑稽.jpg

支持(0) 反对(0)

#7楼 2017-01-16 11:36 Dreamer-1




楼主写得很好，有个问题想问一下，上面这个反编译结果是通过 javap -c 实现的么？还是其他工具？

我用 javap -c 反编译出来的同步方法里没有 flags 标签这些呀？……

支持(0) 反对(0)

#8楼[楼主] 2017-01-17 13:22 liuxiaopeng

@ Dreamer-1
就是通过javap -c反编译的

支持(0) 反对(0)

#9楼 2017-02-13 20:56 曦阳x

1、如果monitor的进入数为0，则该线程进入monitor，然后将进入数设置为1，该线程即为monitor的所有者。

2、如果线程已经占有该monitor，只是重新进入，则进入monitor的进入数加1.
请问楼主：假如一个线程占有了该monitor，此时进入数为1，如果重新进入，则进入数加1变成2，什么情况下会重新进入呢？在同步方法里调用同步代码块会吗？因为此时monitor的进入数为2，如果该线程要退出的话，monitor的进入数直接变为0吗，还是只减1？如果只减1又觉得不合逻辑，按理说线程执行完后，就释放锁了，不应该monitor数还为1啊，除非是嵌套一层一层减的

支持(0) 反对(0)

#10楼[楼主] 2017-02-21 10:54 liuxiaopeng

@ 曦阳x
（1）调用同一个类的其他同步方法就需要重新进入。
（2）每次重新进入都会加1，方法执行完后，减1，当这个值为0是释放锁。确实需要一层一层的减。

支持(0) 反对(0)

#11楼 2017-02-21 10:57 曦阳x

明白了，谢谢楼主

支持(0) 反对(0)

#12楼 2017-03-19 22:52 第一飞哥

@ Dreamer-1
我的环境上也看不到，但是使用javap -verbose SynchronizedMethod.class就能看到了

支持(0) 反对(0)