java synchronized详解

记下来，很重要。

Java语言的关键字，当它用来修饰一个方法或者一个代码块的时候，能够保证在同一时刻最多只有一个线程执行该段代码。

一、当两个并发线程访问同一个对象object中的这个synchronized(this)同步代码块时，一个时间内只能有一个线程得到执行。另一个线程必须等待当前线程执行完这个代码块以后才能执行该代码块。

二、然而，当一个线程访问object的一个synchronized(this)同步代码块时，另一个线程仍然可以访问该object中的非synchronized(this)同步代码块。

三、尤其关键的是，当一个线程访问object的一个synchronized(this)同步代码块时，其他线程对object中所有其它synchronized(this)同步代码块的访问将被阻塞。

四、第三个例子同样适用其它同步代码块。也就是说，当一个线程访问object的一个synchronized(this)同步代码块时，它就获得了这个object的对象锁。结果，其它线程对该object对象所有同步代码部分的访问都被暂时阻塞。

五、以上规则对其它对象锁同样适用.

举例说明：

一、当两个并发线程访问同一个对象object中的这个synchronized(this)同步代码块时，一个时间内只能有一个线程得到执行。另一个线程必须等待当前线程执行完这个代码块以后才能执行该代码块。

```
package ths;

public class Thread1 implements Runnable {
    public void run() {
        synchronized(this) {
            for (int i = 0; i < 5; i++) {

System.out.println(Thread.currentThread().getName() + "
synchronized loop " + i);
            }
        }
    }
```

2011年3月 (8)
2011年2月 (10)
2011年1月 (8)
2010年12月 (10)
2010年11月 (11)
2010年10月 (3)
2010年9月 (7)
2010年8月 (6)
2010年7月 (11)
2010年6月 (4)
2010年4月 (18)
2010年3月 (10)

最新评论

1. Re:java synchronized详解
@lee0oo0确实是写错了...
--famary
2. Re:java synchronized详解
好文好文
--famary
3. Re:java synchronized详解
mark
--微微微疯
4. Re:java synchronized详解
作者是个牛人啊!
--忍辱负重之人
5. Re:java synchronized详解
整篇文章逻辑思路不还很清晰，相同话语也多，希望能够精简。
--王者風範

阅读排行榜

1. java synchronized详解(489528)
2. android 设置Button或者ImageButton的背景透明 半透明 透明(105273)
3. Android中Toast的用法简介(64105)
4. (转) Android 安全机制(35751)
5. (转) Android的Window类(27345)

评论排行榜

1. java synchronized详解(59)
2. 关于WM_NCHITTEST消息(22)
3. (转) Android 安全机制(8)
4. android 中管理短信(8)
5. 写给自己对软件测试经历的总结(7)

推荐排行榜

1. java synchronized详解(135)
2. Android中Toast的用法简介(13)
3. (转) Android 安全机制(7)
4. android 中管理短信(6)
5. 关于WM_NCHITTEST消息(4)

```
    public static void main(String[] args) {
        Thread1 t1 = new Thread1();
        Thread ta = new Thread(t1, "A");
        Thread tb = new Thread(t1, "B");
        ta.start();
        tb.start();
    }
}
```

结果：

```
A synchronized loop 0
A synchronized loop 1
A synchronized loop 2
A synchronized loop 3
A synchronized loop 4
B synchronized loop 0
B synchronized loop 1
B synchronized loop 2
B synchronized loop 3
B synchronized loop 4
```

二、然而，当一个线程访问object的一个synchronized(this)同步代码块时，另一个线程仍然可以访问该object中的非synchronized(this)同步代码块。

```
package ths;

public class Thread2 {
    public void m4t1() {
        synchronized(this) {
            int i = 5;
            while( i-- > 0) {

System.out.println(Thread.currentThread().getName() + " : " + i);
                try {
                    Thread.sleep(500);
                } catch (InterruptedException ie) {
                }
            }
        }
    }
    public void m4t2() {
        int i = 5;
        while( i-- > 0) {
            System.out.println(Thread.currentThread().getName()
+ " : " + i);
            try {
```

```
                Thread.sleep(500);
            } catch (InterruptedException ie) {
            }
        }
    }
    public static void main(String[] args) {
        final Thread2 myt2 = new Thread2();
        Thread t1 = new Thread(  new Runnable() {  public void
run() {  myt2.m4t1();  }  }, "t1"   );
        Thread t2 = new Thread(  new Runnable() {  public void
run() { myt2.m4t2();   }  }, "t2"   );
        t1.start();
        t2.start();
    }
}
```

结果：

```
t1 : 4
t2 : 4
t1 : 3
t2 : 3
t1 : 2
t2 : 2
t1 : 1
t2 : 1
t1 : 0
t2 : 0
```

三、尤其关键的是，当一个线程访问object的一个synchronized(this)同步代码块时，其他线程对object中所有其它synchronized(this)同步代码块的访问将被阻塞。

```
//修改Thread2.m4t2()方法：
public void m4t2() {
    synchronized(this) {
        int i = 5;
        while( i-- > 0) {

System.out.println(Thread.currentThread().getName() + " : " + i);
            try {
                Thread.sleep(500);
            } catch (InterruptedException ie) {
            }
        }
    }

}
```

结果：

t1 : 4
t1 : 3
t1 : 2
t1 : 1
t1 : 0
t2 : 4
t2 : 3
t2 : 2
t2 : 1
t2 : 0

四、第三个例子同样适用其它同步代码块。也就是说，当一个线程访问object的一个synchronized(this)同步代码块时，它就获得了这个object的对象锁。结果，其它线程对该object对象所有同步代码部分的访问都被暂时阻塞。

//修改Thread2.m4t2()方法如下：

```
public synchronized void m4t2() {
    int i = 5;
    while( i-- > 0) {
        System.out.println(Thread.currentThread().getName()
+ " : " + i);
        try {
            Thread.sleep(500);
        } catch (InterruptedException ie) {
        }
    }
}
```

结果：

t1 : 4
t1 : 3
t1 : 2
t1 : 1
t1 : 0
t2 : 4
t2 : 3
t2 : 2
t2 : 1
t2 : 0

五、以上规则对其它对象锁同样适用:

```
package ths;

public class Thread3 {
```

```
    class Inner {
        private void m4t1() {
            int i = 5;
            while(i-- > 0) {

System.out.println(Thread.currentThread().getName() + " :
Inner.m4t1()=" + i);
                try {
                    Thread.sleep(500);
                } catch(InterruptedException ie) {
                }
            }
        }
        private void m4t2() {
            int i = 5;
            while(i-- > 0) {

System.out.println(Thread.currentThread().getName() + " :
Inner.m4t2()=" + i);
                try {
                    Thread.sleep(500);
                } catch(InterruptedException ie) {
                }
            }
        }
    }
    private void m4t1(Inner inner) {
        synchronized(inner) { //使用对象锁
        inner.m4t1();
    }
    private void m4t2(Inner inner) {
        inner.m4t2();
    }
    public static void main(String[] args) {
        final Thread3 myt3 = new Thread3();
        final Inner inner = myt3.new Inner();
        Thread t1 = new Thread( new Runnable() {public void
run() { myt3.m4t1(inner);} }, "t1");
    Thread t2 = new Thread( new Runnable() {public void run() {
myt3.m4t2(inner);} }, "t2");
    t1.start();
    t2.start();
  }
}
```

结果：

尽管线程t1获得了对Inner的对象锁，但由于线程t2访问的是同一个Inner中的非同步部分。所以两个线程互不干扰。

t1：Inner.m4t1()=4
t2：Inner.m4t2()=4
t1：Inner.m4t1()=3
t2：Inner.m4t2()=3
t1：Inner.m4t1()=2
t2：Inner.m4t2()=2
t1：Inner.m4t1()=1
t2：Inner.m4t2()=1
t1：Inner.m4t1()=0
t2：Inner.m4t2()=0

现在在Inner.m4t2()前面加上synchronized：

```
private synchronized void m4t2() {
    int i = 5;
    while(i-- > 0) {
        System.out.println(Thread.currentThread().getName()
+ " : Inner.m4t2()=" + i);
        try {
            Thread.sleep(500);
        } catch(InterruptedException ie) {
        }
    }
}
```

结果：

尽管线程t1与t2访问了同一个Inner对象中两个毫不相关的部分,但因为t1先获得了对Inner的对象锁，所以t2对Inner.m4t2()的访问也被阻塞，因为m4t2()是Inner中的一个同步方法。

t1：Inner.m4t1()=4
t1：Inner.m4t1()=3
t1：Inner.m4t1()=2
t1：Inner.m4t1()=1
t1：Inner.m4t1()=0
t2：Inner.m4t2()=4
t2：Inner.m4t2()=3
t2：Inner.m4t2()=2
t2：Inner.m4t2()=1
t2：Inner.m4t2()=0

第二篇：

synchronized 关键字，它包括两种用法：synchronized 方法和

synchronized 块。
1. synchronized 方法：通过在方法声明中加入 synchronized关键字来声明 synchronized 方法。如：
public synchronized void accessVal(int newVal);
synchronized 方法控制对类成员变量的访问：每个类实例对应一把锁，每个 synchronized 方法都必须获得调用该方法的类实例的锁方能

执行，否则所属线程阻塞，方法一旦执行，就独占该锁，直到从该方法返回时才将锁释放，此后被阻塞的线程方能获得该锁，重新进入可执行

状态。这种机制确保了同一时刻对于每一个类实例，其所有声明为 synchronized 的成员函数中至多只有一个处于可执行状态（因为至多只有

一个能够获得该类实例对应的锁），从而有效避免了类成员变量的访问冲突（只要所有可能访问类成员变量的方法均被声明为 synchronized）

。
在 Java 中，不光是类实例，每一个类也对应一把锁，这样我们也可将类的静态成员函数声明为 synchronized ，以控制其对类的静态成

员变量的访问。
synchronized 方法的缺陷：若将一个大的方法声明为synchronized

将会大大影响效率，典型地，若将线程类的方法 run() 声明为

synchronized ，由于在线程的整个生命期内它一直在运行，因此将导致它对本类任何 synchronized 方法的调用都永远不会成功。当然我们可

以通过将访问类成员变量的代码放到专门的方法中，将其声明为 synchronized ，并在主方法中调用来解决这一问题，但是 Java 为我们提供

了更好的解决办法，那就是 synchronized 块。
2. synchronized 块：通过 synchronized关键字来声明synchronized 块。语法如下：
synchronized(syncObject) {
//允许访问控制的代码
}
synchronized 块是这样一个代码块，其中的代码必须获得对象 syncObject （如前所述，可以是类实例或类）的锁方能执行，具体机

制同前所述。由于可以针对任意代码块，且可任意指定上锁的对象，故灵活性较高。
对synchronized(this)的一些理解
一、当两个并发线程访问同一个对象object中的这个

synchronized(this)同步代码块时，一个时间内只能有一个线程得到执行。另一个线

程必须等待当前线程执行完这个代码块以后才能执行该代码块。
二、然而，当一个线程访问object的一个synchronized(this)同步代码块时，另一个线程仍然可以访问该object中的非synchronized

(this)同步代码块。
三、尤其关键的是，当一个线程访问object的一个synchronized(this)同步代码块时，其他线程对object中所有其它synchronized(this)

同步代码块的访问将被阻塞。
四、第三个例子同样适用其它同步代码块。也就是说，当一个线程访问object的一个synchronized(this)同步代码块时，它就获得了这个

object的对象锁。结果，其它线程对该object对象所有同步代码部分的访问都被暂时阻塞。
五、以上规则对其它对象锁同样适用

http://hi.baidu.com/sunshibing/blog/item/5235b9b731d48ff430add14a.html

java中synchronized用法

打个比方：一个object就像一个大房子，大门永远打开。房子里有很多房间（也就是方法）。

这些房间有上锁的（synchronized方法）， 和不上锁之分（普通方法）。房门口放着一把钥匙（key），这把钥匙可以打开所有上锁的房间。

另外我把所有想调用该对象方法的线程比喻成想进入这房子某个 房间的人。所有的东西就这么多了，下面我们看看这些东西之间如何作用的。

在此我们先来明确一下我们的前提条件。该对象至少有一个synchronized方法，否则这个key还有啥意义。当然也就不会有我们的这个主题了。

一个人想进入某间上了锁的房间，他来到房子门口，看见钥匙在那儿（说明暂时还没有其他人要使用上锁的 房间）。于是他走上去拿到了钥匙

，并且按照自己 的计划使用那些房间。注意一点，他每次使用完一次上锁的房间后会马上把钥匙还回去。即使他要连续使用两间上锁的房间，

中间他也要把钥匙还回去，再取回来。

因此，普通情况下钥匙的使用原则是：“随用随借，用完即还。”

这时其他人可以不受限制的使用那些不上锁的房间，一个人用一间可以，两个人用一间也可以，没限制。但是如果当某个人想要进入上锁

的房

间，他就要跑到大门口去看看了。有钥匙当然拿了就走，没有的话，就只能等了。

要是很多人在等这把钥匙，等钥匙还回来以后，谁会优先得到钥匙?Not guaranteed。象前面例子里那个想连续使用两个上锁房间的家伙，他

中间还钥匙的时候如果还有其他人在等钥匙，那么没有任何保证这家伙能再次拿到。 （JAVA规范在很多地方都明确说明不保证，象

Thread.sleep()休息后多久会返回运行，相同优先权的线程那个首先被执行，当要访问对象的锁被 释放后处于等待池的多个线程哪个会优先得

到，等等。我想最终的决定权是在JVM，之所以不保证，就是因为JVM在做出上述决定的时候，绝不是简简单单根据 一个条件来做出判断，而是

根据很多条。而由于判断条件太多，如果说出来可能会影响JAVA的推广，也可能是因为知识产权保护的原因吧。SUN给了个不保证 就混过去了

。无可厚非。但我相信这些不确定，并非完全不确定。因为计算机这东西本身就是按指令运行的。即使看起来很随机的现象，其实都是有规律

可寻。学过 计算机的都知道，计算机里随机数的学名是伪随机数，是人运用一定的方法写出来的，看上去随机罢了。另外，或许是因为要想弄

的确定太费事，也没多大意义，所 以不确定就不确定了吧。）

再来看看同步代码块。和同步方法有小小的不同。

1.从尺寸上讲，同步代码块比同步方法小。你可以把同步代码块看成是没上锁房间里的一块用带锁的屏风隔开的空间。

2.同步代码块还可以人为的指定获得某个其它对象的key。就像是指定用哪一把钥匙才能开这个屏风的锁，你可以用本房的钥匙；你也可以指定

用另一个房子的钥匙才能开，这样的话，你要跑到另一栋房子那儿把那个钥匙拿来，并用那个房子的钥匙来打开这个房子的带锁的屏风。

记住你获得的那另一栋房子的钥匙，并不影响其他人进入那栋房子没有锁的房间。

为什么要使用同步代码块呢？我想应该是这样的：首先对程序来讲同步的部分很影响运行效率，而一个方法通常是先创建一些局部变

量，再对这些变量做一些 操作，如运算，显示等等；而同步所覆盖

的代码越多，对效率的影响就越严重。因此我们通常尽量缩小其影响范围。

如何做？同步代码块。我们只把一个方法中该同 步的地方同步，比如运算。

另外，同步代码块可以指定钥匙这一特点有个额外的好处，是可以在一定时期内霸占某个对象的key。还记得前面说过普通情况下钥

匙的使用原则吗。现在不是普通情况了。你所取得的那把钥匙不是永远不还，而是在退出同步代码块时才还。

还用前面那个想连续用两个上锁房间的家伙打比方。怎样才能在用完一间以后，继续使用另一间呢。用同步代码块吧。先创建另外

一个线程，做一个同步代码 块，把那个代码块的锁指向这个房子的钥匙。然后启动那个线程。只要你能在进入那个代码块时抓到这房子的钥匙

，你就可以一直保留到退出那个代码块。也就是说 你甚至可以对本房内所有上锁的房间遍历，甚至再sleep(10*60*1000)，而房门口却还有

1000个线程在等这把钥匙呢。很过瘾吧。

在此对sleep()方法和钥匙的关联性讲一下。一个线程在拿到key后，且没有完成同步的内容时，如果被强制sleep()了，那key还一

直在 它那儿。直到它再次运行，做完所有同步内容，才会归还key。记住，那家伙只是干活干累了，去休息一下，他并没干完他要干的事。为

了避免别人进入那个房间 把里面搞的一团糟，即使在睡觉的时候他也要把那唯一的钥匙戴在身上。

最后，也许有人会问，为什么要一把钥匙通开，而不是一个钥匙一个门呢？我想这纯粹是因为复杂性问题。一个钥匙一个门当然更

安全，但是会牵扯好多问题。钥匙 的产生，保管，获得，归还等等。其复杂性有可能随同步方法的增加呈几何级数增加，严重影响效率。这也

算是一个权衡的问题吧。为了增加一点点安全性，导致效 率大大降低，是多么不可取啊。

synchronized的一个简单例子

```
public class TextThread {

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

```
    TxtThread tt = new TxtThread();
    new Thread(tt).start();
    new Thread(tt).start();
    new Thread(tt).start();
    new Thread(tt).start();
}
}

class TxtThread implements Runnable {
int num = 100;
String str = new String();

public void run() {
    synchronized (str) {
     while (num > 0) {

      try {
       Thread.sleep(1);
      } catch (Exception e) {
       e.getMessage();
      }
      System.out.println(Thread.currentThread().getName()
        + "this is " + num--);
     }
    }
}
}
```

上面的例子中为了制造一个时间差,也就是出错的机会,使用了Thread.sleep(10)

Java对多线程的支持与同步机制深受大家的喜爱，似乎看起来使用了synchronized关键字就可以轻松地解决多线程共享数据同步问题。到底如

何？——还得对synchronized关键字的作用进行深入了解才可定论。

总的说来，synchronized关键字可以作为函数的修饰符，也可作为函数内的语句，也就是平时说的同步方法和同步语句块。如果再细的分类，

synchronized可作用于instance变量、object reference（对象引用）、static函数和class literals(类名称字面常量)身上。

在进一步阐述之前，我们需要明确几点：

A．无论synchronized关键字加在方法上还是对象上，它取得的锁都是对象，而不是把一段代码或函数当作锁——而且同步方法很可能还会被其

他线程的对象访问。

B．每个对象只有一个锁（lock）与之相关联。

C．实现同步是要很大的系统开销作为代价的，甚至可能造成死锁，所以尽量避免无谓的同步控制。

接着来讨论synchronized用到不同地方对代码产生的影响：

假设P1、P2是同一个类的不同对象，这个类中定义了以下几种情况的同步块或同步方法，P1、P2就都可以调用它们。

1. 把synchronized当作函数修饰符时，示例代码如下：

Public synchronized void methodAAA()

{

//….

}

这也就是同步方法，那这时synchronized锁定的是哪个对象呢？它锁定的是调用这个同步方法对象。也就是说，当一个对象P1在不同的线程中

执行这个同步方法时，它们之间会形成互斥，达到同步的效果。但是这个对象所属的Class所产生的另一对象P2却可以任意调用这个被加了

synchronized关键字的方法。

上边的示例代码等同于如下代码：

public void methodAAA()

{

synchronized (this)　　// (1)

{

　　//…..

}

}

(1)处的this指的是什么呢？它指的就是调用这个方法的对象，如P1。可见同步方法实质是将synchronized作用于object reference。——那个

拿到了P1对象锁的线程，才可以调用P1的同步方法，而对P2而言，P1这个锁与它毫不相干，程序也可能在这种情形下摆脱同步机制的控制，造

成数据混乱：（

2．同步块，示例代码如下：

public void method3(SomeObject so)

```
{
    synchronized(so)
    {
       //.....
    }
}
```

这时，锁就是so这个对象，谁拿到这个锁谁就可以运行它所控制的那段代码。当有一个明确的对象作为锁时，就可以这样写程序，但当没有明

确的对象作为锁，只是想让一段代码同步时，可以创建一个特殊的instance变量（它得是一个对象）来充当锁：

```
class Foo implements Runnable
{
        private byte[] lock = new byte[0]; // 特殊的instance变量
        Public void methodA()
        {
           synchronized(lock) { //… }
        }
        //.....
}
```

注：零长度的byte数组对象创建起来将比任何对象都经济——查看编译后的字节码：生成零长度的byte[]对象只需3条操作码，而Object lock

= new Object()则需要7行操作码。

3．将synchronized作用于static 函数，示例代码如下：

```
Class Foo
{
    public synchronized static void methodAAA()   // 同步的static函数
    {
        //....
    }
    public void methodBBB()
    {
       synchronized(Foo.class)   // class literal(类名称字面常量)
    }
```

}

代码中的methodBBB()方法是把class literal作为锁的情况，它和同步的static函数产生的效果是一样的，取得的锁很特别，是当前调用这

个方法的对象所属的类（Class，而不再是由这个Class产生的某个具体对象了）。

记得在《Effective Java》一书中看到过将 Foo.class和P1.getClass()用于作同步锁还不一样，不能用P1.getClass()来达到锁这个Class的

目的。P1指的是由Foo类产生的对象。

可以推断：如果一个类中定义了一个synchronized的static函数A，也定义了一个synchronized 的instance函数B，那么这个类的同一对象Obj

在多线程中分别访问A和B两个方法时，不会构成同步，因为它们的锁都不一样。A方法的锁是Obj这个对象，而B的锁是Obj所属的那个Class。

小结如下：

搞清楚synchronized锁定的是哪个对象，就能帮助我们设计更安全的多线程程序。

还有一些技巧可以让我们对共享资源的同步访问更加安全：

1. 定义private 的instance变量+它的 get方法，而不要定义public/protected的instance变量。如果将变量定义为public，对象在外界可以

绕过同步方法的控制而直接取得它，并改动它。这也是JavaBean的标准实现方式之一。

2. 如果instance变量是一个对象，如数组或ArrayList什么的，那上述方法仍然不安全，因为当外界对象通过get方法拿到这个instance对象

的引用后，又将其指向另一个对象，那么这个private变量也就变了，岂不是很危险。 这个时候就需要将get方法也加上synchronized同步，并

且，只返回这个private对象的clone()——这样，调用端得到的就是对象副本的引用了

追究法律责任的权利。

好文要顶 关注我 收藏该文

Gang.Wang
关注 - 20
粉丝 - 265
+加关注

135 推荐　0 反对

« 上一篇：SharedPreferences 的用法
» 下一篇：PowerManager和WakeLock的操作步骤

posted @ 2011-02-27 23:48 Gang.Wang 阅读(489540) 评论(59) 编辑 收藏

< Prev 1 2

评论列表

#51楼 2016-08-28 10:46 花满园

非常清楚，该注意的点以及容易混淆的点

支持(0) 反对(0)

#52楼 2017-02-14 16:07 浅浅love

非常棒！受益匪浅

支持(0) 反对(0)

#53楼 2017-03-10 09:52 NoneNon

受益匪浅。

支持(0) 反对(0)

#54楼 2017-03-16 10:40 paramWei

受益匪浅 谢谢

支持(0) 反对(0)

#55楼 2017-04-24 11:28 王者風範

整篇文章逻辑思路不还很清晰，相同话语也多，希望能够精简。

支持(0) 反对(0)

#56楼 2017-06-29 11:29 忍辱负重之人

作者是个牛人啊！

支持(0) 反对(0)

#57楼 2017-07-18 15:51 微微微疯

mark

支持(0) 反对(0)

#58楼 2017-07-19 12:05 famary

好文好文

支持(0) 反对(0)

#59楼 2017-07-19 12:06 famary

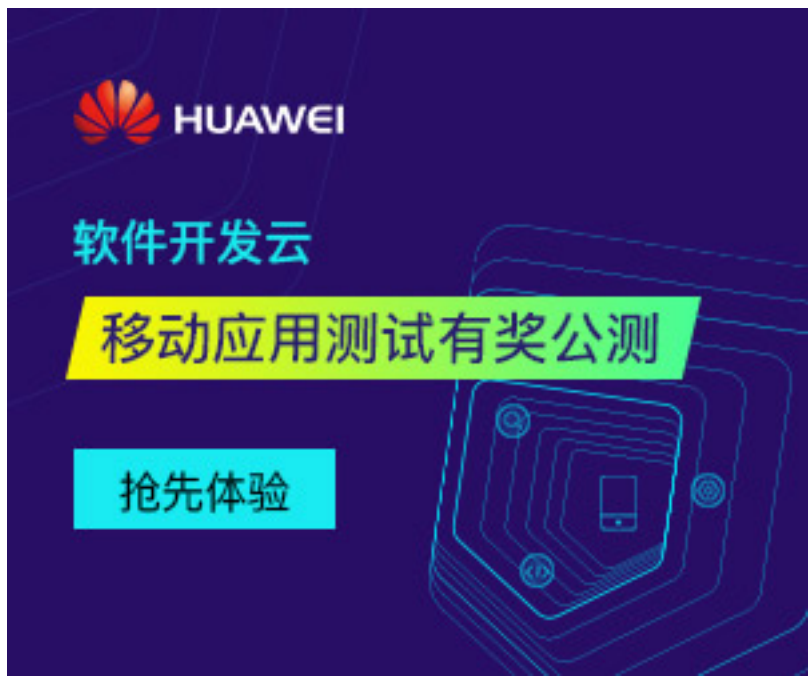
@ lee0oo0
确实是写错了

支持(0) 反对(0)

< Prev 1 2

刷新评论 刷新页面 返回顶部

注册用户登录后才能发表评论，请 登录 或 注册，访问网站首页。



最新IT新闻:
- 小程序打造无人店推社交优惠 万家无人店微信支付可立减
- 百度外卖代理商愤怒喊冤：被规则玩弄 投资破产
- 王健林的新目标
- 背靠阿里又远离阿里，钉钉如何成了马云口中的"惊喜"
- 东芝股东大会批准出售闪存芯片子公司 2万亿日元卖给贝恩财团

» 更多新闻...

最新知识库文章:
- 实用VPC虚拟私有云设计原则
- 如何阅读计算机科学类的书
- Google 及其云智慧
- 做到这一点，你也可以成为优秀的程序员
- 写给立志做码农的大学生

» 更多知识库文章...