转载自：<http://langgufu.iteye.com/blog/2152608>

在java编程中，经常需要用到同步，而用得最多的也许是synchronized关键字了，下面看看这个关键字的用法。

因为synchronized关键字涉及到锁的概念，所以先来了解一些相关的锁知识。

java的内置锁：每个java对象都可以用做一个实现同步的锁，这些锁成为内置锁。线程进入同步代码块或方法的时候会自动获得该锁，在退出同步代码块或方法时会释放该锁。获得内置锁的唯一途径就是进入这个锁的保护的同步代码块或方法。

java内置锁是一个互斥锁，这就是意味着最多只有一个线程能够获得该锁，当线程A尝试去获得线程B持有的内置锁时，线程A必须等待或者阻塞，知道线程B释放这个锁，如果B线程不释放这个锁，那么A线程将永远等待下去。

java的对象锁和类锁：java的对象锁和类锁在锁的概念上基本上和内置锁是一致的，但是，两个锁实际是有很大的区别的，对象锁是用于对象实例方法，或者一个对象实例上的，类锁是用于类的静态方法或者一个类的class对象上的。我们知道，类的对象实例可以有很多个，但是每个类只有一个class对象，所以不同对象实例的对象锁是互不干扰的，但是每个类只有一个类锁。但是有一点必须注意的是，其实类锁只是一个概念上的东西，并不是真实存在的，它只是用来帮助我们理解锁定实例方法和静态方法的区别的

上面已经对锁的一些概念有了一点了解，下面探讨synchronized关键字的用法。

synchronized的用法：synchronized修饰方法和synchronized修饰代码块。

下面分别分析这两种用法在对象锁和类锁上的效果。

对象锁的synchronized修饰方法和代码块：

**Java代码   收藏代码**

1. **public** **class** TestSynchronized
2. {
3. **public** **void** test1()
4. {
5. **synchronized**(**this**)
6. {
7. **int** i = 5;
8. **while**( i-- > 0)
9. {
10. System.out.println(Thread.currentThread().getName() + " : " + i);
11. **try**
12. {
13. Thread.sleep(500);
14. }
15. **catch** (InterruptedException ie)
16. {
17. }
18. }
19. }
20. }
22. **public** **synchronized** **void** test2()
23. {
24. **int** i = 5;
25. **while**( i-- > 0)
26. {
27. System.out.println(Thread.currentThread().getName() + " : " + i);
28. **try**
29. {
30. Thread.sleep(500);
31. }
32. **catch** (InterruptedException ie)
33. {
34. }
35. }
36. }
38. **public** **static** **void** main(String[] args)
39. {
40. **final** TestSynchronized myt2 = **new** TestSynchronized();
41. Thread test1 = **new** Thread(  **new** Runnable() {  **public** **void** run() {  myt2.test1();  }  }, "test1"  );
42. Thread test2 = **new** Thread(  **new** Runnable() {  **public** **void** run() { myt2.test2();   }  }, "test2"  );
43. test1.start();;
44. test2.start();
45. //         TestRunnable tr=new TestRunnable();
46. //         Thread test3=new Thread(tr);
47. //         test3.start();
48. }
50. }

**Java代码   收藏代码**

1. test2 : 4
2. test2 : 3
3. test2 : 2
4. test2 : 1
5. test2 : 0
6. test1 : 4
7. test1 : 3
8. test1 : 2
9. test1 : 1
10. test1 : 0

上述的代码，第一个方法时用了同步代码块的方式进行同步，传入的对象实例是this，表明是当前对象，当然，如果需要同步其他对象实例，也不可传入其他对象的实例；第二个方法是修饰方法的方式进行同步。因为第一个同步代码块传入的this，所以两个同步代码所需要获得的对象锁都是同一个对象锁，下面main方法时分别开启两个线程，分别调用test1和test2方法，那么两个线程都需要获得该对象锁，另一个线程必须等待。上面也给出了运行的结果可以看到：直到test2线程执行完毕，释放掉锁，test1线程才开始执行。（可能这个结果有人会有疑问，代码里面明明是先开启test1线程，为什么先执行的是test2呢？这是因为java编译器在编译成字节码的时候，会对代码进行一个重排序，也就是说，编译器会根据实际情况对代码进行一个合理的排序，编译前代码写在前面，在编译后的字节码不一定排在前面，所以这种运行结果是正常的， 这里是题外话，最主要是检验synchronized的用法的正确性）

如果我们把test2方法的synchronized关键字去掉，执行结果会如何呢？

**Java代码   收藏代码**

1. test1 : 4
2. test2 : 4
3. test2 : 3
4. test1 : 3
5. test1 : 2
6. test2 : 2
7. test2 : 1
8. test1 : 1
9. test2 : 0
10. test1 : 0

上面是执行结果，我们可以看到，结果输出是交替着进行输出的，这是因为，某个线程得到了对象锁，但是另一个线程还是可以访问没有进行同步的方法或者代码。进行了同步的方法（加锁方法）和没有进行同步的方法（普通方法）是互不影响的，一个线程进入了同步方法，得到了对象锁，其他线程还是可以访问那些没有同步的方法（普通方法）。这里涉及到内置锁的一个概念（此概念出自java并发编程实战第二章）：对象的内置锁和对象的状态之间是没有内在的关联的，虽然大多数类都将内置锁用做一种有效的加锁机制，但对象的域并不一定通过内置锁来保护。当获取到与对象关联的内置锁时，并不能阻止其他线程访问该对象，当某个线程获得对象的锁之后，只能阻止其他线程获得同一个锁。之所以每个对象都有一个内置锁，是为了免去显式地创建锁对象。

所以synchronized只是一个内置锁的加锁机制，当某个方法加上synchronized关键字后，就表明要获得该内置锁才能执行，并不能阻止其他线程访问不需要获得该内置锁的方法。

类锁的修饰（静态）方法和代码块：

**Java代码   收藏代码**

1. **public** **class** TestSynchronized
2. {
3. **public** **void** test1()
4. {
5. **synchronized**(TestSynchronized.**class**)
6. {
7. **int** i = 5;
8. **while**( i-- > 0)
9. {
10. System.out.println(Thread.currentThread().getName() + " : " + i);
11. **try**
12. {
13. Thread.sleep(500);
14. }
15. **catch** (InterruptedException ie)
16. {
17. }
18. }
19. }
20. }
22. **public** **static** **synchronized** **void** test2()
23. {
24. **int** i = 5;
25. **while**( i-- > 0)
26. {
27. System.out.println(Thread.currentThread().getName() + " : " + i);
28. **try**
29. {
30. Thread.sleep(500);
31. }
32. **catch** (InterruptedException ie)
33. {
34. }
35. }
36. }
38. **public** **static** **void** main(String[] args)
39. {
40. **final** TestSynchronized myt2 = **new** TestSynchronized();
41. Thread test1 = **new** Thread(  **new** Runnable() {  **public** **void** run() {  myt2.test1();  }  }, "test1"  );
42. Thread test2 = **new** Thread(  **new** Runnable() {  **public** **void** run() { TestSynchronized.test2();   }  }, "test2"  );
43. test1.start();
44. test2.start();
45. //         TestRunnable tr=new TestRunnable();
46. //         Thread test3=new Thread(tr);
47. //         test3.start();
48. }
50. }

**Java代码   收藏代码**

1. test1 : 4
2. test1 : 3
3. test1 : 2
4. test1 : 1
5. test1 : 0
6. test2 : 4
7. test2 : 3
8. test2 : 2
9. test2 : 1
10. test2 : 0

其实，类锁修饰方法和代码块的效果和对象锁是一样的，因为类锁只是一个抽象出来的概念，只是为了区别静态方法的特点，因为静态方法是所有对象实例共用的，所以对应着synchronized修饰的静态方法的锁也是唯一的，所以抽象出来个类锁。其实这里的重点在下面这块代码，synchronized同时修饰静态和非静态方法

**Java代码   收藏代码**

1. **public** **class** TestSynchronized
2. {
3. **public** **synchronized** **void** test1()
4. {
5. **int** i = 5;
6. **while**( i-- > 0)
7. {
8. System.out.println(Thread.currentThread().getName() + " : " + i);
9. **try**
10. {
11. Thread.sleep(500);
12. }
13. **catch** (InterruptedException ie)
14. {
15. }
16. }
17. }
19. **public** **static** **synchronized** **void** test2()
20. {
21. **int** i = 5;
22. **while**( i-- > 0)
23. {
24. System.out.println(Thread.currentThread().getName() + " : " + i);
25. **try**
26. {
27. Thread.sleep(500);
28. }
29. **catch** (InterruptedException ie)
30. {
31. }
32. }
33. }
35. **public** **static** **void** main(String[] args)
36. {
37. **final** TestSynchronized myt2 = **new** TestSynchronized();
38. Thread test1 = **new** Thread(  **new** Runnable() {  **public** **void** run() {  myt2.test1();  }  }, "test1"  );
39. Thread test2 = **new** Thread(  **new** Runnable() {  **public** **void** run() { TestSynchronized.test2();   }  }, "test2"  );
40. test1.start();
41. test2.start();
42. //         TestRunnable tr=new TestRunnable();
43. //         Thread test3=new Thread(tr);
44. //         test3.start();
45. }
47. }

**Java代码   收藏代码**

1. test1 : 4
2. test2 : 4
3. test1 : 3
4. test2 : 3
5. test2 : 2
6. test1 : 2
7. test2 : 1
8. test1 : 1
9. test1 : 0
10. test2 : 0

上面代码synchronized同时修饰静态方法和实例方法，但是运行结果是交替进行的，这证明了类锁和对象锁是两个不一样的锁，控制着不同的区域，它们是互不干扰的。同样，线程获得对象锁的同时，也可以获得该类锁，即同时获得两个锁，这是允许的。

到这里，对synchronized的用法已经有了一定的了解。这时有一个疑问，既然有了synchronized修饰方法的同步方式，为什么还需要synchronized修饰同步代码块的方式呢？而这个问题也是synchronized的缺陷所在

synchronized的缺陷：当某个线程进入同步方法获得对象锁，那么其他线程访问这里对象的同步方法时，必须等待或者阻塞，这对高并发的系统是致命的，这很容易导致系统的崩溃。如果某个线程在同步方法里面发生了死循环，那么它就永远不会释放这个对象锁，那么其他线程就要永远的等待。这是一个致命的问题。

当然同步方法和同步代码块都会有这样的缺陷，只要用了synchronized关键字就会有这样的风险和缺陷。既然避免不了这种缺陷，那么就应该将风险降到最低。这也是同步代码块在某种情况下要优于同步方法的方面。例如在某个类的方法里面：这个类里面声明了一个对象实例，SynObject so=new SynObject()；在某个方法里面调用了这个实例的方法so.testsy();但是调用这个方法需要进行同步，不能同时有多个线程同时执行调用这个方法。

这时如果直接用synchronized修饰调用了so.testsy();代码的方法，那么当某个线程进入了这个方法之后，这个对象其他同步方法都不能给其他线程访问了。假如这个方法需要执行的时间很长，那么其他线程会一直阻塞，影响到系统的性能。

如果这时用synchronized来修饰代码块：synchronized（so）{so.testsy();}，那么这个方法加锁的对象是so这个对象，跟执行这行代码的对象没有关系，当一个线程执行这个方法时，这对其他同步方法时没有影响的，因为他们持有的锁都完全不一样。

不过这里还有一种特例，就是上面演示的第一个例子，对象锁synchronized同时修饰方法和代码块，这时也可以体现到同步代码块的优越性，如果test1方法同步代码块后面有非常多没有同步的代码，而且有一个100000的循环，这导致test1方法会执行时间非常长，那么如果直接用synchronized修饰方法，那么在方法没执行完之前，其他线程是不可以访问test2方法的，但是如果用了同步代码块，那么当退出代码块时就已经释放了对象锁，当线程还在执行test1的那个100000的循环时，其他线程就已经可以访问test2方法了。这就让阻塞的机会或者线程更少。让系统的性能更优越。

一个类的对象锁和另一个类的对象锁是没有关联的，当一个线程获得A类的对象锁时，它同时也可以获得B类的对象锁。

可能上面只有理论和代码，对刚接触的人比较难理解，下面举一个例子，

这个例子是摘自博客：http://www.cnblogs.com/GnagWang/archive/2011/02/27/1966606.html

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，像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。

大概就这么多了。