**Java中synchronized同步锁用法及作用范围**

 https://blog.csdn.net/guyue35/article/details/85052537

Java 中的 synchronized 关键字可以在多线程环境下用来作为线程安全的同步锁。本文主要对 synchronized 的作用，以及其有效范围进行讨论。   
Java中的对象锁和类锁：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。

synchronized 关键字主要有以下几种用法：   
- 非静态方法的同步；   
- 静态方法的同步；   
- 代码块。

下面分对象锁和类锁来分别说明 synchronized 用法：

**对象锁**

非静态方法使用 synchronized 修饰的写法，修饰实例方法时，锁定的是当前对象：

1. public synchronized void test(){
2. *// TODO*
3. }

代码块使用 synchronized 修饰的写法，使用代码块，如果传入的参数是 this，那么锁定的也是当前的对象：

1. public void test(){
2. synchronized (this) {
3. *// TODO*
4. }
5. }

* 1
* 2
* 3
* 4
* 5

下面通过例子来说明对象锁：   
定义一个类，方法如下，将 count 自减，从 5 到 0：

1. public class TestSynchronized {
3. public synchronized void minus() {
4. int count = 5;
5. for (int i = 0; i < 5; i++) {
6. count--;
7. System.out.println(Thread.currentThread().getName() + " - " + count);
8. try {
9. Thread.sleep(500);
10. } catch (InterruptedException e) {
11. }
12. }
13. }
15. }

测试调用方法如下：

1. public class Run {
3. public static void main(String[] args) {
5. final TestSynchronized test = new TestSynchronized();
7. Thread thread1 = new Thread(new Runnable() {
9. @Override
10. public void run() {
11. test.minus();
12. }
13. });
15. Thread thread2 = new Thread(new Runnable() {
17. @Override
18. public void run() {
19. test.minus();
20. }
21. });
23. thread1.start();
24. thread2.start();
26. }
28. }

两个线程 thread1 和 thread2，同时访问对象的方法，由于该方法是 synchronized 关键字修饰的，那么这两个线程都需要获得该对象锁，一个获得后另一个线程必须等待。所以我们可以猜测运行结果应该是，一个线程执行完毕后，另一个线程才开始执行，运行例子，输出打印结果如下：

1. Thread-0 - 4
2. Thread-0 - 3
3. Thread-0 - 2
4. Thread-0 - 1
5. Thread-0 - 0
6. Thread-1 - 4
7. Thread-1 - 3
8. Thread-1 - 2
9. Thread-1 - 1
10. Thread-1 - 0

（另：thread1 和 thread2 谁先执行并不一定）   
本例对于对象锁进行了基础的解释。但是对象锁的范围是怎样的，对象的某个同步方法被一个线程访问后，其他线程能不能访问该对象的其他同步方法，以及是否可以访问对象的其他非同步方法呢，下面对两种进行验证：

对两个同步方法两个线程的验证：   
修改类如下，加入 minus2() 方法，和 minus() 方法一样：

1. package com.test.run;
3. public class TestSynchronized {
5. public synchronized void minus() {
6. int count = 5;
7. for (int i = 0; i < 5; i++) {
8. count--;
9. System.out.println(Thread.currentThread().getName() + " - " + count);
10. try {
11. Thread.sleep(500);
12. } catch (InterruptedException e) {
13. }
14. }
15. }
17. public synchronized void minus2() {
18. int count = 5;
19. for (int i = 0; i < 5; i++) {
20. count--;
21. System.out.println(Thread.currentThread().getName() + " - " + count);
22. try {
23. Thread.sleep(500);
24. } catch (InterruptedException e) {
25. }
26. }
27. }
29. }

测试调用如下，两个线程访问不同的方法：

1. public class Run {
3. public static void main(String[] args) {
5. final TestSynchronized test = new TestSynchronized();
7. Thread thread1 = new Thread(new Runnable() {
9. @Override
10. public void run() {
11. test.minus();
12. }
13. });
15. Thread thread2 = new Thread(new Runnable() {
17. @Override
18. public void run() {
19. test.minus2();
20. }
21. });
23. thread1.start();
24. thread2.start();
25. }
27. }

输出结果如下：

1. Thread-0 - 4
2. Thread-0 - 3
3. Thread-0 - 2
4. Thread-0 - 1
5. Thread-0 - 0
6. Thread-1 - 4
7. Thread-1 - 3
8. Thread-1 - 2
9. Thread-1 - 1
10. Thread-1 - 0

可以看到，某个线程得到了对象锁之后，该对象的其他同步方法是锁定的，其他线程是无法访问的。   
下面看是否能访问非同步方法：   
修改类代码如下，将 minus2() 的 synchronized 修饰去掉，代码如下：

1. public class TestSynchronized {
3. public synchronized void minus() {
4. int count = 5;
5. for (int i = 0; i < 5; i++) {
6. count--;
7. System.out.println(Thread.currentThread().getName() + " - " + count);
8. try {
9. Thread.sleep(500);
10. } catch (InterruptedException e) {
11. }
12. }
13. }
15. public void minus2() {
16. int count = 5;
17. for (int i = 0; i < 5; i++) {
18. count--;
19. System.out.println(Thread.currentThread().getName() + " - " + count);
20. try {
21. Thread.sleep(500);
22. } catch (InterruptedException e) {
23. }
24. }
25. }
27. }

测试调用的类不变，如下：

1. public class Run2 {
3. public static void main(String[] args) {
5. final TestSynchronized test = new TestSynchronized();
7. Thread thread1 = new Thread(new Runnable() {
9. @Override
10. public void run() {
11. test.minus();
12. }
13. });
15. Thread thread2 = new Thread(new Runnable() {
17. @Override
18. public void run() {
19. test.minus2();
20. }
21. });
23. thread1.start();
24. thread2.start();
26. }
28. }

* 执行结果如下：

1. Thread-1 - 4
2. Thread-0 - 4
3. Thread-1 - 3
4. Thread-0 - 3
5. Thread-1 - 2
6. Thread-0 - 2
7. Thread-1 - 1
8. Thread-0 - 1
9. Thread-1 - 0
10. Thread-0 - 0

可以看到，结果是交替的，说明线程是交替执行的，说明如果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）。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。

**类锁**

类锁需要 synchronized 来修饰静态 static 方法，写法如下：

1. public static synchronized void test(){
2. *// TODO*
3. }

或者使用代码块，需引用当前的类：

1. public static void test(){
2. synchronized (TestSynchronized.class) {
3. *// TODO*
4. }
5. }

举例说明类锁的作用：

1. public class TestSynchronized {
3. public static synchronized void minus() {
4. int count = 5;
5. for (int i = 0; i < 5; i++) {
6. count--;
7. System.out.println(Thread.currentThread().getName() + " - " + count);
8. try {
9. Thread.sleep(500);
10. } catch (InterruptedException e) {
11. }
12. }
13. }
15. }

* 测试调用类如下：

1. public class Run {
3. public static void main(String[] args) {
5. Thread thread1 = new Thread(new Runnable() {
7. @Override
8. public void run() {
9. TestSynchronized.minus();
10. }
11. });
13. Thread thread2 = new Thread(new Runnable() {
15. @Override
16. public void run() {
17. TestSynchronized.minus();
18. }
19. });
21. thread1.start();
22. thread2.start();
24. }
26. }

输出结果如下：

1. Thread-0 - 4
2. Thread-0 - 3
3. Thread-0 - 2
4. Thread-0 - 1
5. Thread-0 - 0
6. Thread-1 - 4
7. Thread-1 - 3
8. Thread-1 - 2
9. Thread-1 - 1
10. Thread-1 - 0

可以看到，类锁和对象锁其实是一样的，由于静态方法是类所有对象共用的，所以进行同步后，该静态方法的锁也是所有对象唯一的。每次只能有一个线程来访问对象的该非静态同步方法。   
类锁的作用和对象锁类似，但是作用范围是否和对象锁一致呢，下面看对象锁和类锁是否等同：   
修改类，两个同步方法，其中一个是静态的：

1. public class TestSynchronized {
3. public static synchronized void minus() {
4. int count = 5;
5. for (int i = 0; i < 5; i++) {
6. count--;
7. System.out.println(Thread.currentThread().getName() + " - " + count);
8. try {
9. Thread.sleep(500);
10. } catch (InterruptedException e) {
11. }
12. }
13. }
15. public synchronized void minus2() {
16. int count = 5;
17. for (int i = 0; i < 5; i++) {
18. count--;
19. System.out.println(Thread.currentThread().getName() + " - " + count);
20. try {
21. Thread.sleep(500);
22. } catch (InterruptedException e) {
23. }
24. }
25. }
27. }

测试调用类如下，静态方法直接用类调用，实例方法由对象来调用：

1. public class Run {
3. public static void main(String[] args) {
5. final TestSynchronized test = new TestSynchronized();
7. Thread thread1 = new Thread(new Runnable() {
9. @Override
10. public void run() {
11. TestSynchronized.minus();
12. }
13. });
15. Thread thread2 = new Thread(new Runnable() {
17. @Override
18. public void run() {
19. test.minus2();
20. }
21. });
23. thread1.start();
24. thread2.start();
26. }
28. }

运行结果：

1. Thread-1 - 4
2. Thread-0 - 4
3. Thread-0 - 3
4. Thread-1 - 3
5. Thread-0 - 2
6. Thread-1 - 2
7. Thread-0 - 1
8. Thread-1 - 1
9. Thread-1 - 0
10. Thread-0 - 0

可以看到两个线程是交替进行的，也就是说类锁和对象锁是不一样的锁，是互相独立的。

=================================================================

=================================================================

=================================================================

=================================================================

synchronized的四种用法 - 行走江湖的少侠哥 - CSDN博客  
https://blog.csdn.net/sinat\_32588261/article/details/72880159

**synchronized的四种用法**

**一 修饰方法**

Synchronized修饰一个方法很简单，就是在方法的前面加synchronized，synchronized修饰方法和修饰一个代码块类似，只是作用范围不一样，修饰代码块是大括号括起来的范围，而修饰方法范围是整个函数。

例如：

方法一

1. public synchronized void method()
2. {
3. // todo
4. }

方法二

1. public void method()
2. {
3. synchronized(this) {
4. // todo
5. }
6. }

写法一修饰的是一个方法，写法二修饰的是一个代码块，但写法一与写法二是等价的，都是锁定了整个方法时的内容。

synchronized关键字不能继承。   
虽然可以使用synchronized来定义方法，但synchronized并不属于方法定义的一部分，因此，synchronized关键字不能被继承。如果在父类中的某个方法使用了synchronized关键字，而在子类中覆盖了这个方法，在子类中的这个方法默认情况下并不是同步的，而必须显式地在子类的这个方法中加上synchronized关键字才可以。当然，还可以在子类方法中调用父类中相应的方法，这样虽然子类中的方法不是同步的，但子类调用了父类的同步方法，因此，子类的方法也就相当于同步了。这两种方式的例子代码如下：   
在子类方法中加上synchronized关键字

1. class Parent {
2. public synchronized void method() { }
3. }
4. class Child extends Parent {
5. public synchronized void method() { }
6. }

在子类方法中调用父类的同步方法

1. class Parent {
2. public synchronized void method() { }
3. }
4. class Child extends Parent {
5. public void method() { super.method(); }
6. }

1. 在定义接口方法时不能使用synchronized关键字。
2. 构造方法不能使用synchronized关键字，但可以使用synchronized代码块来进行同步。

**二 修饰一个代码块**

1）一个线程访问一个对象中的synchronized(this)同步代码块时，其他试图访问该对象的线程将被阻塞

注意下面两个程序的区别

1. class SyncThread implements Runnable {
2. private static int count;
4. public SyncThread() {
5. count = 0;
6. }
8. public void run() {
9. synchronized(this) {
10. for (int i = 0; i < 5; i++) {
11. try {
12. System.out.println(Thread.currentThread().getName() + ":" + (count++));
13. Thread.sleep(100);
14. } catch (InterruptedException e) {
15. e.printStackTrace();
16. }
17. }
18. }
19. }
21. public int getCount() {
22. return count;
23. }
24. }
26. public class Demo00 {
27. public static void main(String args[]){
28. //test01
29. // SyncThread s1 = new SyncThread();
30. // SyncThread s2 = new SyncThread();
31. // Thread t1 = new Thread(s1);
32. // Thread t2 = new Thread(s2);
33. //test02
34. SyncThread s = new SyncThread();
35. Thread t1 = new Thread(s);
36. Thread t2 = new Thread(s);
38. t1.start();
39. t2.start();
40. }
41. }

test01的运行结果



test02的运行结果



当两个并发线程(thread1和thread2)访问同一个对象(syncThread)中的synchronized代码块时，在同一时刻只能有一个线程得到执行，另一个线程受阻塞，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。Thread1和thread2是互斥的，因为在执行synchronized代码块时会锁定当前的对象，只有执行完该代码块才能释放该对象锁，下一个线程才能执行并锁定该对象

为什么上面的例子中thread1和thread2同时在执行。这是因为synchronized只锁定对象，每个对象只有一个锁（lock）与之相关联。

2）当一个线程访问对象的一个synchronized(this)同步代码块时，另一个线程仍然可以访问该对象中的非synchronized(this)同步代码块。

例：

1. class Counter implements Runnable{
2. private int count;
4. public Counter() {
5. count = 0;
6. }
8. public void countAdd() {
9. synchronized(this) {
10. for (int i = 0; i < 5; i ++) {
11. try {
12. System.out.println(Thread.currentThread().getName() + ":" + (count++));
13. Thread.sleep(100);
14. } catch (InterruptedException e) {
15. e.printStackTrace();
16. }
17. }
18. }
19. }
21. //非synchronized代码块，未对count进行读写操作，所以可以不用synchronized
22. public void printCount() {
23. for (int i = 0; i < 5; i ++) {
24. try {
25. System.out.println(Thread.currentThread().getName() + " count:" + count);
26. Thread.sleep(100);
27. } catch (InterruptedException e) {
28. e.printStackTrace();
29. }
30. }
31. }
33. public void run() {
34. String threadName = Thread.currentThread().getName();
35. if (threadName.equals("A")) {
36. countAdd();
37. } else if (threadName.equals("B")) {
38. printCount();
39. }
40. }
41. }
43. public class Demo00{
44. public static void main(String args[]){
45. Counter counter = new Counter();
46. Thread thread1 = new Thread(counter, "A");
47. Thread thread2 = new Thread(counter, "B");
48. thread1.start();
49. thread2.start();
50. }
51. }



可以看见B线程的调用是非synchronized,并不影响A线程对synchronized部分的调用。从上面的结果中可以看出一个线程访问一个对象的synchronized代码块时，别的线程可以访问该对象的非synchronized代码块而不受阻塞。

3）指定要给某个对象加锁

1. /\*\*
2. \* 银行账户类
3. \*/
4. class Account {
5. String name;
6. float amount;
8. public Account(String name, float amount) {
9. this.name = name;
10. this.amount = amount;
11. }
12. //存钱
13. public void deposit(float amt) {
14. amount += amt;
15. try {
16. Thread.sleep(100);
17. } catch (InterruptedException e) {
18. e.printStackTrace();
19. }
20. }
21. //取钱
22. public void withdraw(float amt) {
23. amount -= amt;
24. try {
25. Thread.sleep(100);
26. } catch (InterruptedException e) {
27. e.printStackTrace();
28. }
29. }
31. public float getBalance() {
32. return amount;
33. }
34. }
36. /\*\*
37. \* 账户操作类
38. \*/
39. class AccountOperator implements Runnable{
40. private Account account;
41. public AccountOperator(Account account) {
42. this.account = account;
43. }
45. public void run() {
46. synchronized (account) {
47. account.deposit(500);
48. account.withdraw(500);
49. System.out.println(Thread.currentThread().getName() + ":" + account.getBalance());
50. }
51. }
52. }
54. public class Demo00{
56. //public static final Object signal = new Object(); // 线程间通信变量
57. //将account改为Demo00.signal也能实现线程同步
58. public static void main(String args[]){
59. Account account = new Account("zhang san", 10000.0f);
60. AccountOperator accountOperator = new AccountOperator(account);
62. final int THREAD\_NUM = 5;
63. Thread threads[] = new Thread[THREAD\_NUM];
64. for (int i = 0; i < THREAD\_NUM; i ++) {
65. threads[i] = new Thread(accountOperator, "Thread" + i);
66. threads[i].start();
67. }
68. }
69. }

运行结果



在AccountOperator 类中的run方法里，我们用synchronized 给account对象加了锁。这时，当一个线程访问account对象时，其他试图访问account对象的线程将会阻塞，直到该线程访问account对象结束。也就是说谁拿到那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，就可以用类似下面这样的方式写程序。

1. public void method3(SomeObject obj)
2. {
3. //obj 锁定的对象
4. synchronized(obj)
5. {
6. // todo
7. }
8. }

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对象来充当锁：

1. class Test implements Runnable
2. {
3. private byte[] lock = new byte[0]; // 特殊的instance变量
4. public void method()
5. {
6. synchronized(lock) {
7. // todo 同步代码块
8. }
9. }
11. public void run() {
13. }
14. }

本例中去掉注释中的signal可以看到同样的运行结果

**三 修饰一个静态的方法**

Synchronized也可修饰一个静态方法，用法如下：

1. public synchronized static void method() {
2. // todo
3. }

静态方法是属于类的而不属于对象的。同样的，synchronized修饰的静态方法锁定的是这个类的所有对象。

1. /\*\*
2. \* 同步线程
3. \*/
4. class SyncThread implements Runnable {
5. private static int count;
7. public SyncThread() {
8. count = 0;
9. }
11. public synchronized static void method() {
12. for (int i = 0; i < 5; i ++) {
13. try {
14. System.out.println(Thread.currentThread().getName() + ":" + (count++));
15. Thread.sleep(100);
16. } catch (InterruptedException e) {
17. e.printStackTrace();
18. }
19. }
20. }
22. public synchronized void run() {
23. method();
24. }
25. }
27. public class Demo00{
29. public static void main(String args[]){
30. SyncThread syncThread1 = new SyncThread();
31. SyncThread syncThread2 = new SyncThread();
32. Thread thread1 = new Thread(syncThread1, "SyncThread1");
33. Thread thread2 = new Thread(syncThread2, "SyncThread2");
34. thread1.start();
35. thread2.start();
36. }
37. }



syncThread1和syncThread2是SyncThread的两个对象，但在thread1和thread2并发执行时却保持了线程同步。这是因为run中调用了静态方法method，而静态方法是属于类的，所以syncThread1和syncThread2相当于用了同一把锁。

**四  修饰一个类**

Synchronized还可作用于一个类，用法如下：

1. class ClassName {
2. public void method() {
3. synchronized(ClassName.class) {
4. // todo
5. }
6. }
7. }

1. /\*\*
2. \* 同步线程
3. \*/
4. class SyncThread implements Runnable {
5. private static int count;
7. public SyncThread() {
8. count = 0;
9. }
11. public static void method() {
12. synchronized(SyncThread.class) {
13. for (int i = 0; i < 5; i ++) {
14. try {
15. System.out.println(Thread.currentThread().getName() + ":" + (count++));
16. Thread.sleep(100);
17. } catch (InterruptedException e) {
18. e.printStackTrace();
19. }
20. }
21. }
22. }
24. public synchronized void run() {
25. method();
26. }
27. }

本例的的给class加锁和上例的给静态方法加锁是一样的，所有对象公用一把锁

总结

* 1. 无论synchronized关键字加在方法上还是对象上，如果它作用的对象是非静态的，则它取得的锁是对象；如果synchronized作用的对象是一个静态方法或一个类，则它取得的锁是对类，该类所有的对象同一把锁。   
     B. 每个对象只有一个锁（lock）与之相关联，谁拿到这个锁谁就可以运行它所控制的那段代码。   
     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。

# [java 查看线程死锁](https://www.cnblogs.com/ilahsa/archive/2013/06/03/3115410.html)

# https://www.cnblogs.com/ilahsa/archive/2013/06/03/3115410.html

　那我们怎么确定一定是死锁呢？有两种方法。

　　1>使用JDK给我们的的工具JConsole，可以通过打开cmd然后输入jconsole打开。

　　　　1)连接到需要查看的进程。

2）打开线程选项卡，然后点击左下角的“检测死锁”

　　　　3）jconsole就会给我们检测出该线程中造成死锁的线程，点击选中即可查看详情：

　　　　 从上图中我们可以看出：

　　　　　　①在线程Thread-1中，从状态可以看出，它想申请Paper这个资源，但是这个资源已经被Thread-0拥有了，所以就堵塞了。

　　　　　　②在线程Thread-0中，从状态可以看出，它想申请Pen这个资源，但是这个资源已经被Thread-1拥有了，所以就堵塞了。

　　　　Thread-1一直等待paper资源，而Thread--一直等待pen资源，于是这两个线程就这么僵持了下去，造成了死锁。

　　2>直接使用JVM自带的命令

　　　　1）首先通过 jps 命令查看需要查看的Java进程的vmid，如图，我们要查看的进程TestDeadLock的vmid号是7412；

　　　　2）然后利用 jstack 查看该进程中的堆栈情况，在cmd中输入 jstack -l 7412 ，移动到输出的信息的最下面即可得到：

　　　　至此，相信大家都会看了吧，具体就不说啦，根据输出，找到问题所在的代码，开始调试解决即可啦。