## 下面主要对线程安全和线程非安全的情况进行举例，为此加深对java中线程安全代码编写的理解。

## 1 线程非安全代码例子

**public** **class** TraditionalSynchornizedTest {

1. /\*\*
2. \* @param args
3. \*/
4. **public** **static** **void** main(String[] args) {

// Outerput outerput = new Outerput(); is also

ok

1. **new** TraditonalSynchornizedTest().sartThread();
2. }
3. **public** **void** sartThread(){
4. **final** Outerput outerput = **new** Outerput();
5. **new** Thread(**new** Runnable(){
6. @Override
7. **public** **void** run() {
8. **while**(**true**){
9. **try** {
10. Thread.sleep(5);
11. } **catch** (InterruptedException e) {
12. e.printStackTrace();
13. }
14. outerput.print("zhangsanfeng");
15. }
16. }
18. }).start();
19. **new** Thread(**new** Runnable(){
20. @Override
21. **public** **void** run() {
22. **while**(**true**){
23. **try** {
24. Thread.sleep(5);
25. } **catch** (InterruptedException e) {
26. e.printStackTrace();
27. }
28. outerput.print("luxiaofeng");
29. }
30. }
32. }).start();
33. }
34. //内部类
35. **public** **class** Outerput{

public void print(String name){

for(int i = 0;i < name.length(); i++){

System.out.print(name.charAt(i));

printToFile(name.charAt(i));

}

printToFile('\r');

printToFile('\n');

System.out.println();

}

public synchronized void printToFile(char ch) {

try {

File file = new File("outputChar.txt");

// if(!file.exists()){

FileWriter fw = new FileWriter(file, true);

FileOutputStream fos = new FileOutputStream("outputChar2.txt", true);

fos.write(ch);

fos.flush();

// fw.write(ch);

fw.append(ch);

fw.flush();

fw.close();

fos.close();

} catch (IOException e) {

e.printStackTrace();

}

}

}

2. }
3. }

输出的结果中可能出现这种情况：

luxiaofeng

zhangsanfeng

luxiaofeng

luxiazhangsanfeng

ofeng

luxiaofeng

….

原因：以上代码没有对共同持有的对象outerput加锁，所以会出现线程安全问题

**1、对代码块加锁**

对共同持有的对象加锁可以把内部类写成这样的

1. public class Outerput{
2. public void print(String name){
3. synchronized (this) {
4. for(int i = 0;i < name.length(); i++){
5. System.out.print(name.charAt(i));
6. }
7. System.out.println();
8. }
9. }
10. }

**2、对非静态方法加锁，加锁的对象是this**

1. public class Outerput{
2. public synchronized void print(String name){
3. for(int i = 0;i < name.length(); i++){
4. System.out.print(name.charAt(i));
5. }
6. System.out.println();
7. }
8. }

**3、对静态方法加锁的对象到底是谁？**

1. public static synchronized  void print2(String name){
2. for(int i = 0;i < name.length(); i++){
3. System.out.print(name.charAt(i));
4. }
5. System.out.println();
6. }

**4、在线程中的run方法中进行处理加锁 如**

**synchronized** (outerput) {

outerput.print("lisi");

}

其实加锁的对象是字节码对象，Outerput.class

如果和非静态方法同时持有同一个对象时，可以持有同一个字节码对象。

## 2 线程非安全举例2

**package** cn.zhou;

**import** java.util.Arrays;

**import** java.util.concurrent.CountDownLatch;

**public** **class** ThreadUnsafe {

**static** **int** *k*=0;

**static** **int** []*arr*;

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

**final** UnsafeSequence us = **new** UnsafeSequence();

**int** len =100;

//final int []arr = new int[len];

*arr* = **new** **int**[len];

CountDownLatch doneSignal = **new** CountDownLatch(len);

**for** (**int** i = 0; i < len; i++) {

**new** Thread(**new** WorkRunnable(us,doneSignal))

.start();

}

**try** {

doneSignal.await();

} **catch** (InterruptedException e) {

e.printStackTrace();

}

System.*out*.println("thread has finished all");

**for**(**int** m=0;m<*arr*.length;m++){

**for**(**int** n=m+1;n<*arr*.length;n++){

**if**(*arr*[m] == *arr*[n])

{

System.*out*.println("exist same number !,the number is "+*arr*[m]+",index="+m+",index\_other="+n);

**break**;

}

}

}

Arrays.*sort*(*arr*);

**for**(**int** item:*arr*){

System.*out*.print(item+" ");

}

}

**static** **class** WorkRunnable **implements** Runnable{

**private** UnsafeSequence us;

**private** CountDownLatch cdl;

**public** WorkRunnable(UnsafeSequence us,CountDownLatch cdl){

**this**.us = us;

**this**.cdl = cdl;

}

@Override

**public** **void** run() {

**try** {

Thread.*sleep*(5);

} **catch** (InterruptedException e) {

e.printStackTrace();

}

//对k进行加锁 是为了避免对k的读写并发错误，因为我们此刻只关注UnSafeSequence类的并发安全性问题

**synchronized**(Integer.*valueOf*(*k*)){

System.*out*.println(*arr*[*k*++]=us.getNext());

}

cdl.countDown();

}

}

**public** **static** **class** UnsafeSequence{

**private** **int** value;

**public** **int** getNext\_other(){

**int** ret = value;

**for**(**int** i=0;i<100;i++) {

**int** xx = i/21;

}

value++;

**return** ret;

}

**public** **int** getNext(){

**return** value++;

}

}

}

分析：

1）当使用getNext（）时,其中的可能的一次结果如下：

0

2

4

7

6

8

5

3

14

29

30

31

32

37

1

36

35

34

33

28

27

26

25

24

40

23

22

46

21

19

20

51

18

17

52

16

15

13

12

11

10

9

71

70

69

68

67

87

88

66

65

64

63

62

61

60

59

58

57

56

55

54

53

50

49

48

47

45

44

43

42

41

39

38

98

97

96

95

94

93

92

91

90

89

86

85

84

83

82

81

81

80

79

78

77

76

75

74

73

72

thread has finished all

exist same number !,the number is 81,index=81,index\_other=82

0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29 30 31 32 33 34 35 36 37 38 39 40 41 42 43 44 45 46 47 48 49 50 51 52 53 54 55 56 57 58 59 60 61 62 63 64 65 66 67 68 69 70 71 72 73 74 75 76 77 78 79 80 81 81 82 83 84 85 86 87 88 89 90 91 92 93 94 95 96 97 98

UnsafeSequence类在单线程环境中，可以正常工作，但在多线程环境中则不能。其问题在于：如果执行时机不对，那么这两个线程调用getNext()时可能会得到相同的值。虽然递增运算value++看上去是单个操作，但事实上它包含三个独立的操作：读取value值；将value加1；并将计算结果写入value。由于运行时可能将多个线程之间的操作交替执行，因此这两个线程可能同时执行读操作，从而使它们得到相同的值，并都将这个值加1(如下A\B执行流所示)。结果就是 在不同线程的调用中返回的相同的值（如上述结果一样）。

A: value->81 81+1=82 value=82;

B: value->81 81+1=82 value=82

2）当使用getNext\_other时，可能出现的一种结果是：

0

1

2

3

4

5

6

7

8

9

10

11

12

13

14

15

16

17

18

19

20

21

22

23

24

25

26

28

31

30

32

29

33

34

36

41

42

49

50

51

57

58

59

60

61

62

63

27

99

98

97

96

95

94

93

92

91

90

89

88

87

85

85

84

83

82

81

80

79

78

77

76

75

74

73

72

71

70

69

68

67

66

64

64

56

55

54

53

52

48

47

46

45

44

43

40

39

38

37

35

thread has finished all

exist same number !,the number is 64,index=64,index\_other=65

exist same number !,the number is 85,index=85,index\_other=86

0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29 30 31 32 33 34 35 36 37 38 39 40 41 42 43 44 45 46 47 48 49 50 51 52 53 54 55 56 57 58 59 60 61 62 63 64 64 66 67 68 69 70 71 72 73 74 75 76 77 78 79 80 81 82 83 84 85 85 87 88 89 90 91 92 93 94 95 96 97 98 99

很明显，此处的问题和（1）中所述类似。

需要解决上述两个问题，可以采用对共享对象资源进行同步控制，即使用synchronized关键字加锁。如将public int getNext() 改为 public synchronized int getValue()。或者在该方法里面使用synchronized（this）或者在线程体中使用synchronized(us){…}

## 3 线程非安全举例3

ArrayList和Vector有什么区别？HashMap和HashTable有什么区别？StringBuilder和StringBuffer有什么区别？这些都是Java面试中常见的基础问题。面对这样的问题，回答是：ArrayList是非线程安全的，Vector是线程安全的；HashMap是非线程安全的，HashTable是线程安全的；StringBuilder是非线程安全的，StringBuffer是线程安全的。

非线程安全的现象模拟

这里就使用ArrayList和Vector二者来说明。

下面的代码，在主线程中new了一个非线程安全的ArrayList，然后开1000个线程分别向这个ArrayList里面添加元素，每个线程添加100个元素，等所有线程执行完成后，这个ArrayList的size应该是多少？应该是100000个？

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33  34  35  36  37  38  39  40  41  42  43  44  45  46  47  48  49  50  51  52  53  54  55  56  57  58  59  60  61  62  63  64  65  66  67  68 | public class Main  {      public static void main(String[] args)      {          // 进行10次测试          for(int i = 0; i < 10; i++)          {              test();          }      }        public static void test()      {          // 用来测试的List          List<Object> list = new ArrayList<Object>();  //List<Object> list = new Vector<>(); is ok            // 线程数量(1000)          int threadCount = 1000;            // 用来让主线程等待threadCount个子线程执行完毕          CountDownLatch countDownLatch = new CountDownLatch(threadCount);            // 启动threadCount个子线程          for(int i = 0; i < threadCount; i++)          {              Thread thread = new Thread(new MyThread(list, countDownLatch));              thread.start();          }            try          {              // 主线程等待所有子线程执行完成，再向下执行              countDownLatch.await();          }          catch (InterruptedException e)          {              e.printStackTrace();          }            // List的size          System.out.println(list.size());      }  }    class MyThread implements Runnable  {      private List<Object> list;        private CountDownLatch countDownLatch;        public MyThread(List<Object> list, CountDownLatch countDownLatch)      {          this.list = list;          this.countDownLatch = countDownLatch;      }        public void run()      {          // 每个线程向List中添加100个元素          for(int i = 0; i < 100; i++)          {  /\* synchronized (list) { // is ok  list.add(new Object());  }\*/              list.add(new Object());          }            // 完成一个子线程          countDownLatch.countDown();      }  } |

上面进行了10次测试（为什么要测试10次？因为非线程安全并不是每次都会导致问题）。

输出结果：

99946

100000

100000

100000

99998

99959

100000

99975

100000

99996

或者执行结果为

99993

Exception in thread "Thread-100" java.lang.ArrayIndexOutOfBoundsException: 316

at java.util.ArrayList.add(ArrayList.java:441)

at cn.zhou.ConcurrentTest01$MyThread.run(ConcurrentTest01.java:68)

at java.lang.Thread.run(Thread.java:745)

上面的输出结果发现，并不是每次测试结果都是100000，有好几次测试最后ArrayList的size小于100000，甚至时不时会抛出个IndexOutOfBoundsException异常。（如果没有这个现象可以多试几次）

这就是非线程安全带来的问题了。上面的代码如果用于生产环境，就会有隐患就会有BUG了。

再用线程安全的Vector来进行测试，上面代码改变一处，test()方法中

|  |  |
| --- | --- |
| 1 | List<Object> list = new ArrayList<Object>(); |

改成

|  |  |
| --- | --- |
| 1 | List<Object> list = new Vector<Object>(); |

再运行程序。

输出结果：

100000

100000

100000

100000

100000

100000

100000

100000

100000

100000

再多跑几次，发现都是100000，没有任何问题。因为Vector是线程安全的，在多线程操作同一个Vector对象时，不会有任何问题。

再换成LinkedList试试，同样还会出现ArrayList类似的问题，因为LinkedList也是非线程安全的。

二者如何取舍

非线程安全是指多线程操作同一个对象可能会出现问题。而线程安全则是多线程操作同一个对象不会有问题。

线程安全必须要使用很多synchronized关键字来同步控制，所以必然会导致性能的降低。

所以在使用的时候，如果是多个线程操作同一个对象，那么使用线程安全的Vector；否则，就使用效率更高的ArrayList。

非线程安全!=不安全

有人在使用过程中有一个不正确的观点：我的程序是多线程的，不能使用ArrayList要使用Vector，这样才安全。

非线程安全并不是多线程环境下就不能使用。注意我上面有说到：多线程操作同一个对象。注意是同一个对象。比如最上面那个模拟，就是在主线程中new的一个ArrayList然后多个线程操作同一个ArrayList对象。

如果是每个线程中new一个ArrayList，而这个ArrayList只在这一个线程中使用，那么肯定是没问题的。

线程安全的实现

线程安全是通过线程同步控制来实现的，也就是synchronized关键字。

在这里，我用代码分别实现了一个非线程安全的计数器和线程安全的计数器Counter，并对他们分别进行了多线程测试。

非线程安全的计数器：

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33  34  35  36  37  38  39  40  41  42  43  44  45  46  47  48  49  50  51  52  53  54  55  56  57  58  59  60  61  62  63  64  65  66  67  68  69  70  71  72  73  74  75  76  77  78  79  80  81  82  83 | public class Main  {      public static void main(String[] args)      {          // 进行10次测试          for(int i = 0; i < 10; i++)          {              test();          }      }        public static void test()      {          // 计数器          Counter counter = new Counter();            // 线程数量(1000)          int threadCount = 1000;            // 用来让主线程等待threadCount个子线程执行完毕          CountDownLatch countDownLatch = new CountDownLatch(threadCount);            // 启动threadCount个子线程          for(int i = 0; i < threadCount; i++)          {              Thread thread = new Thread(new MyThread(counter, countDownLatch));              thread.start();          }            try          {              // 主线程等待所有子线程执行完成，再向下执行              countDownLatch.await();          }          catch (InterruptedException e)          {              e.printStackTrace();          }            // 计数器的值          System.out.println(counter.getCount());      }  }    class MyThread implements Runnable  {      private Counter counter;        private CountDownLatch countDownLatch;        public MyThread(Counter counter, CountDownLatch countDownLatch)      {          this.counter = counter;          this.countDownLatch = countDownLatch;      }        public void run()      {          // 每个线程向Counter中进行10000次累加          for(int i = 0; i < 10000; i++)          {              counter.addCount();          }            // 完成一个子线程          countDownLatch.countDown();      }  }    class Counter  {      private int count = 0;        public int getCount()      {          return count;      }        public void addCount()      {          count++;      }  } |

面的测试代码中，开启1000个线程，每个线程对计数器进行10000次累加，最终输出结果应该是10000000。

但是上面代码中的Counter未进行同步控制，所以非线程安全。

输出结果：

9963727

9973178

9999577

9987650

9988734

9988665

9987820

9990847

9992305

9972233

稍加修改，把Counter改成线程安全的计数器：

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14 | class Counter  {      private int count = 0;        public int getCount()      {          return count;      }        public synchronized void addCount()      {          count++;      }  } |

上面只是在addCount()方法中加上了synchronized同步控制，就成为一个线程安全的计数器了。再执行程序。

输出结果：

10000000

10000000

10000000

10000000

10000000

10000000

10000000

10000000

10000000

10000000

## 4线程非安全举例4

并发问题再也不是一个只有高级程序员才能接触的问题了，在使用多线程编程的时候，我们更多的将目光放在追求系统的高并发和高吞吐，而这一切的前提是确保程序的正确性。在多线程编程中容易产生的问题有很多，比如线程安全问题、死锁、饥饿等。下面的例子将从线程安全开始，逐步开启并发编程的大门。

[?](http://my.oschina.net/ielts0909/blog/85645)

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9 | @NotThreadSafe  public class UnsafeSequence {      private int value;        /\*\* Returns a unique value. \*/      public int getNext() {          return value++;      }  } |

这个例子来源于《Java Concurrency In Practice》的第一章，一个最最简单，却容易引起线程安全问题的片段。书中给出了如下解释：The problem with UnsafeSequence is that with some unlucky timing, two threads could call getNext and receive the same value. Figure 1.1 shows how this can happen. The increment notation, nextValue++, may appear to be a single operation, but is in fact three separate operations: read the value, add one to it, and write out the new value. Since operations in multiple threads may be arbitrarily interleaved by the runtime, it is possible for two threads to read the value at the same time, both see the same value, and then both add one to it. The result is that the same sequence number is returned from multiple calls in different threads.

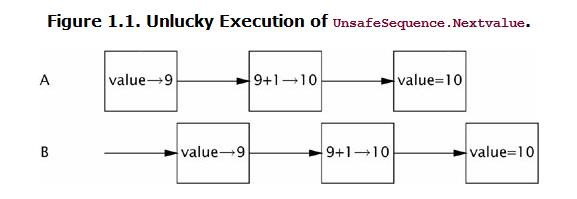
[](http://static.oschina.net/uploads/space/2012/1027/222615_dkKU_589742.jpg)

Figure 1.1的确很明了的告诉我们线程推进的过程，这也给了我们一个寻找线程不安全的方法，通过这样图示法来分析问题。

当然，这里引出了第一个可能会引起线程不安全的因素：程序中有变量的读取、写入或判断操作。

[?](http://my.oschina.net/ielts0909/blog/85645)

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11 | /\*\*例如上述变量的自增\*/      public int getNext() {          return value++;      }  /\*\*例如单例模式中队变量的判断操作\*/     Public Object getInstance(){       If(obj==null){    return new Object();  }  return obj;  }       public class Singleton{             private static Singleton singleton = null;     private Singleton (){}             public static synchronized synchronized getInstance(){                  if(singleton==null){                      singleton = new Singleton();                  }                 return singleton;             }        } |

写一个简单的程序验证一下上述问题，其实线程的学习最好的办法就是举例证明线程的不安全性，然后再想出解决方案，当然这也可能是这部分学习最难所在：

[?](http://my.oschina.net/ielts0909/blog/85645)

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33  34  35  36  37  38  39  40  41  42  43  44  45  46 | package com.a2.concurrency.chapter1;  /\*\*   \* 线程安全第一种因素：程序中有变量的读取、写入或判断操作   \* @author ChenHui   \*   \*/  public class UnsafeSequence {        private int value;        public int getValue() {          return value++;      }        public static void main(String[] args) throws InterruptedException {            final UnsafeSequence us = new UnsafeSequence();          Thread th1 = new Thread("th1") {              @Override              public void run() {                  System.out.println( us.getValue()+" "+super.getName());              }          };            Thread th2 = new Thread("th2") {              @Override              public void run() {                  System.out.println(us.getValue()+" "+super.getName());              }          };            th1.start();          /\*\*           \* 如果不執行Thread.sleep(1000);           \* 偶尔結果为：           \* 0 th2           \* 0 th1           \* 如果执行Thread.sleep(1000);           \* 结果为：           \* 0 th1           \* 1 th2           \*/          //Thread.sleep(1000);          th2.start();      }  } |

对于这种因素产生的问题，我们先给出一种常用解决方案，就是使用同步机制。这里我们先给出最简单，大家也最容易想到的方案，对操作加synchronized关键字：

[?](http://my.oschina.net/ielts0909/blog/85645)

|  |  |
| --- | --- |
| 1  2  3  4 | private volatile int value;      public synchronized int getNext() {          return value++;      } |

在这里使用了synchronized的情况下，是否使用volatile关键字并不是主要的。

这一节的最后例举一下一般会遇到线程安全问题的地方，引用自并发编程书中第一章：

l  Timer

l  Servlet/JSP

l  RMI

l  Swing

l  ……

注：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）。

## 5 servlet是单例的 所以需要线程安全 以及如何实现线程安全

 Servlet/JSP技术和ASP、PHP等相比，由于其多线程运行而具有很高的执行效率。由于Servlet/JSP默认是以多线程模式执行的，所以，在编写代码时需要非常细致地考虑多线程的安全性问题。然而，很多人编写Servlet/JSP程序时并没有注意到多线程安全性的问题，这往往造成编写的程序在少量用户访问时没有任何问题，而在并发用户上升到一定值时，就会经常出现一些莫明其妙的问题。

　　Servlet的多线程机制

　　Servlet体系结构是建立在Java多线程机制之上的，它的生命周期是由Web容器负责的。当客户端第一次请求某个Servlet时，Servlet容器将会根据web.xml配置文件实例化这个Servlet类。当有新的客户端请求该Servlet时，一般不会再实例化该Servlet类，也就是有多个线程在使用这个实例。Servlet容器会自动使用线程池等技术来支持系统的运行，如图1所示。

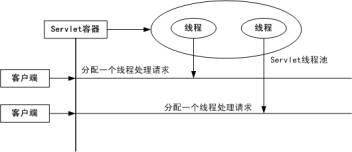


图1 Servlet线程池

　　这样，当两个或多个线程同时访问同一个Servlet时，可能会发生多个线程同时访问同一资源的情况，数据可能会变得不一致。所以在用Servlet构建的Web应用时如果不注意线程安全的问题，会使所写的Servlet程序有难以发现的错误。

　　Servlet的线程安全问题

　　Servlet的线程安全问题主要是由于实例变量使用不当而引起的，这里以一个现实的例子来说明。

Import javax.servlet. \*;

Import javax.servlet.http. \*;

Import java.io. \*;

Public class Concurrent Test extends HttpServlet {

**PrintWriter output;**

Public void service (HttpServletRequest request, HttpServletResponse response) throws ServletException, IOException {

String username;

Response.setContentType ("text/html; charset=gb2312");

Username = request.getParameter ("username");

**Output = response.getWriter ();**

Try {Thread. sleep (5000); //为了突出并发问题，在这设置一个延时

} Catch (Interrupted Exception e){}

output.println("用户名:"+Username+"<BR>");

}

}

　　该Servlet中定义了一个实例变量output，在service方法将其赋值为用户的输出。当一个用户访问该Servlet时，程序会正常的运行，但当多个用户并发访问时，就可能会出现其它用户的信息显示在另外一些用户的[浏览器](http://www.2cto.com/os/liulanqi/)上的问题。这是一个严重的问题。为了突出并发问题，便于测试、观察，我们在回显用户信息时执行了一个延时的操作。假设已在web.xml配置文件中注册了该Servlet，现有两个用户a和b同时访问该Servlet（可以启动两个IE浏览器，或者在两台机器上同时访问）,即同时在浏览器中输入：

　　a： http://localhost: 8080/servlet/ConcurrentTest? Username=a

　　b： http://localhost: 8080/servlet/ConcurrentTest? Username=b

　　如果用户b比用户a回车的时间稍慢一点，将得到如图2所示的输出：



图2 a用户和b用户的浏览器输出

　　从图2中可以看到，Web服务器启动了两个线程分别处理来自用户a和用户b的请求，但是在用户a的浏览器上却得到一个空白的屏幕，用户a的信息显示在用户b的浏览器上。该Servlet存在线程不安全问题。下面我们就从分析该实例的内存模型入手,观察不同时刻实例变量output的值来分析使该Servlet线程不安全的原因。

　　Java的内存模型JMM（Java Memory Model）JMM主要是为了规定了线程和内存之间的一些关系。根据JMM的设计，系统存在一个主内存(Main Memory)，Java中所有实例变量都储存在主存中，对于所有线程都是共享的。每条线程都有自己的工作内存(Working Memory)，工作内存由缓存和堆栈两部分组成，缓存中保存的是主存中变量的拷贝，缓存可能并不总和主存同步，也就是缓存中变量的修改可能没有立刻写到主存中；堆栈中保存的是线程的局部变量，线程之间无法相互直接访问堆栈中的变量。根据JMM，我们可以将论文中所讨论的Servlet实例的内存模型抽象为图3所示的模型。

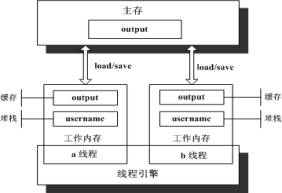


图3 Servlet实例的JMM模型

　　下面根据图3所示的内存模型，来分析当用户a和b的线程（简称为a线程、b线程）并发执行时，Servlet实例中所涉及变量的变化情况及线程的执行情况，如图4所示。

调度时刻 a线程 b线程

T1 访问Servlet页面

T2   访问Servlet页面

T3 output=a的输出username=a休眠5000毫秒，让出CPU

T4   output=b的输出（写回主存）username=b休眠5000毫秒，让出CPU

T5 在用户b的浏览器上输出a线程的username的值,a线程终止。

T6   在用户b的浏览器上输出b线程的username的值,b线程终止。 　　　　　　　　　　　　　　　　　　图4 Servlet实例的线程调度情况

　　从图4中可以清楚的看到，由于b线程对实例变量output的修改覆盖了a线程对实例变量output的修改，从而导致了用户a的信息显示在了用户b的浏览器上。如果在a线程执行输出语句时，b线程对output的修改还没有刷新到主存，那么将不会出现图2所示的输出结果，因此这只是一种偶然现象，但这更增加了程序潜在的危险性。

设计线程安全的Servlet

　　通过上面的分析，我们知道了实例变量不正确的使用是造成Servlet线程不安全的主要原因。下面针对该问题给出了三种解决方案并对方案的选取给出了一些参考性的建议。

　　1、实现 SingleThreadModel 接口

　　该接口指定了系统如何处理对同一个Servlet的调用。如果一个Servlet被这个接口指定,那么在这个Servlet中的service方法将不会有两个线程被同时执行，当然也就不存在线程安全的问题。这种方法只要将前面的Concurrent Test类的类头定义更改为：

Public class Concurrent Test extends HttpServlet implements SingleThreadModel {

…………

}

　　2、同步对共享数据的操作

　　使用synchronized 关键字能保证一次只有一个线程可以访问被保护的区段，在本论文中的Servlet可以通过同步块操作来保证线程的安全。同步后的代码如下：

…………

Public class Concurrent Test extends HttpServlet { …………

Username = request.getParameter ("username");

Synchronized (this){

Output = response.getWriter ();

Try {

Thread. Sleep (5000);

} Catch (Interrupted Exception e){}

output.println("用户名:"+Username+"<BR>");

}

}

}

　　3、避免使用实例变量

　　本实例中的线程安全问题是由实例变量造成的，只要在Servlet里面的任何方法里面都不使用实例变量，那么该Servlet就是线程安全的。

　　修正上面的Servlet代码，将实例变量改为局部变量实现同样的功能，代码如下：

……

Public class Concurrent Test extends HttpServlet {public void service (HttpServletRequest request, HttpServletResponse

Response) throws ServletException, IOException {

Print Writer output;

String username;

Response.setContentType ("text/html; charset=gb2312");

……

}

}

对上面的三种方法进行测试，可以表明用它们都能设计出线程安全的Servlet程序。但是，如果一个Servlet实现了SingleThreadModel接口，Servlet引擎将为每个新的请求创建一个单独的Servlet实例，这将引起大量的系统开销。SingleThreadModel在Servlet2.4中已不再提倡使用；同样如果在程序中使用同步来保护要使用的共享的数据，也会使系统的性能大大下降。这是因为被同步的代码块在同一时刻只能有一个线程执行它，使得其同时处理客户请求的吞吐量降低，而且很多客户处于阻塞状态。另外为保证主存内容和线程的工作内存中的数据的一致性，要频繁地刷新缓存,这也会大大地影响系统的性能。所以在实际的开发中也应避免或最小化 Servlet 中的同步代码；在Serlet中避免使用实例变量是保证Servlet线程安全的最佳选择。从Java 内存模型也可以知道，方法中的临时变量是在栈上分配空间，而且每个线程都有自己私有的栈空间，所以它们不会影响线程的安全。

**对于存在线程不安全的类,如何避免出现线程安全问题:**

**1、采用synchronized同步。缺点就是存在堵塞问题,可以使用一下读锁和写锁。**

**2、使用ThreadLocal(实际上就是一个HashMap),这样不同的线程维护自己的对象，线程之间相互不干扰。**

## 6线程死锁举例

**package** cn.zhou;

**import** java.util.concurrent.CountDownLatch;

**/\*\***

线程死锁等待举例

\* 可以通过JConsole命令工具 去查看

\*\*/

**public** **class** SynAddDeadLock **implements** Runnable{

**private** **int** a,b;

CountDownLatch countDownLatch;

**public** SynAddDeadLock(**int** a,**int** b,CountDownLatch doneSignal){

**this**.a = a;

**this**.b = b;

countDownLatch = doneSignal;

}

@Override

**public** **void** run() {

**synchronized** (Integer.*valueOf*(a)) {

**synchronized** (Integer.*valueOf*(b)) {

System.*out*.println(Thread.*currentThread*().getName()+"---"+(a+b));

}

}

countDownLatch.countDown();

}

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

**final** **int** threadCount = 200;

**boolean** isDeadLock = **false**;

CountDownLatch doneSignal = **new** CountDownLatch(threadCount);

**for** (**int** i = 0; i < threadCount/2; i++) {

**new** Thread(**new** SynAddDeadLock(1, 2,doneSignal)).start();

**new** Thread(**new** SynAddDeadLock(2,1,doneSignal)).start();

}

isDeadLock = **true**;

**try** {

System.*out*.println("has deadlock");

doneSignal.await();

//若所有线程都已经结束 说明不存在死锁

isDeadLock = **false**;

} **catch** (InterruptedException e) {

e.printStackTrace();

}

**if**(!isDeadLock){

System.*out*.println("no deadlock");

}

System.*out*.println("end");

}

}

分析：开启200个线程分别计算1+2以及2+1的值，其实for循环是可以省略的，两个线程也可能会导致死锁，不过那样概率太小，需要尝试很多次才能看到效果。一般的话，带for循环的（如上所述）的版本最多运行2~3次就会遇到死锁，程序无法结束。**造成死锁的原因是：**Integer.valueOf()方法基于减少对象创建次数和节省内存的考虑，【-128,127】之间的数字会被缓存（默认值，实际值取决于java.lang.Integer.IntegerCache.high参数的设置）,当valueOf方法传入参数在这个范围之内，将之间返回缓存中的对象。即 代码中调用了200次的Integer.valueOf（）方法一共只返回了两个不同的对象（1和2）.假如在某个线程的两个synchronized块之间发生了一次线程切换，那就会出现线程A等着被线程B持有的Integer.valueof(1),线程B又等着被线程A持有的Integer.valueOf(2) 结果出现都跑不下去的情景。另外，我们可以可以通过Jconsole jviauslvm等工具查看到线程死锁等待的具体信息。