

多线程进阶---JUC并发编程

1.Lock锁 (重点)

传统 Synchronizd

```
1 package com.godfrey.demo01;
2
3 /**
4  * description : 模拟卖票
5  *
6  * @author godfrey
7  * @since 2020-05-14
8  */
9 public class SaleTicketDemo01 {
10     public static void main(String[] args) {
11         Ticket ticket = new Ticket();
12
13         new Thread(() -> {
14             for (int i = 0; i < 60; i++) {
15                 ticket.sale();
16             }
17         }, "A").start();
18
19         new Thread(() -> {
20             for (int i = 0; i < 60; i++) {
21                 ticket.sale();
22             }
23         }, "B").start();
24
25         new Thread(() -> {
26             for (int i = 0; i < 60; i++) {
27                 ticket.sale();
28             }
29         }, "C").start();
30     }
31 }
32
33 //资源类OOP
34 class Ticket {
35     private int number = 50;
36
37     public synchronized void sale() {
38         if (number > 0) {
39             System.out.println(Thread.currentThread().getName() + "卖出了第"
40             + (50 - (number--)) + "票, 剩余: " + number);
41         }
42     }
43 }
```

Synchronized(本质：队列+锁)和Lock区别

1. Synchronized 是内置关键字， Lock 是一个Java类
2. Synchronized 无法判断锁的状态， Lock 可以判断是否获取到了锁
3. Synchronized 会自动释放锁， Lock 必须手动释放！如果不释放锁， **死锁**
4. Synchronized 线程1（获得锁，阻塞）、线程2（等待，傻傻的等）； Lock 锁就不一定会等待下去（tryLock）
5. Synchronized 可重入锁，不可中断，非公平； Lock 可重入锁，可以判断锁，非公平（可以自己设置）；
6. Synchronized 适合锁少量的代码同步问题， Lock 适合锁大量的同步代码！

锁是什么，如何判断锁的是谁

2.线程之间通信问题：生成者消费者问题

Synchronized版生产者消费者问题

```
1 package proc;
2
3 /**
4 * description : 生产者消费者问题
5 *
6 * @author godfrey
7 * @since 2020-05-14
8 */
9 public class A {
10     public static void main(String[] args) {
11         Data data = new Data();
12
13         new Thread(() -> {
14             for (int i = 0; i < 10; i++) {
15                 try {
16                     data.increment();
17                 } catch (InterruptedException e) {
18                     e.printStackTrace();
19                 }
20             }
21         }, "A").start();
22
23         new Thread(() -> {
24             for (int i = 0; i < 10; i++) {
25                 try {
26                     data.decrement();
27                 } catch (InterruptedException e) {
28                     e.printStackTrace();
29                 }
30             }
31         }, "B").start();
32     }
33 }
```

```

32     }
33 }
34
35 // 判断等待，业务，通知
36 class Data {
37     private int number = 0;
38
39     //+1
40     public synchronized void increment() throws InterruptedException {
41         if (number != 0) {
42             //等待
43             this.wait();
44         }
45         number++;
46         System.out.println(Thread.currentThread().getName() + "=> " +
number);
47         //通知其他线程，我+1完毕了
48         this.notifyAll();
49     }
50
51     //-1
52     public synchronized void decrement() throws InterruptedException {
53         if (number == 0) {
54             //等待
55             this.wait();
56         }
57         number--;
58         System.out.println(Thread.currentThread().getName() + "=> " +
number);
59         //通知其他线程，我-1完毕了
60         this.notifyAll();
61     }
62 }

```

Lock接口

随着这种增加的灵活性，额外的责任。没有块结构化锁定会删除使用synchronized方法和语句发生的锁的自动释放。在大多数情况下，应使用以下惯用语：

加锁  **解锁**

```
Lock l = ... l.lock(); try { // access the resource protected by this lock } finally { l.unlock() }
```

当在不同范围内发生锁定和解锁时，必须注意确保在锁定时执行的所有代码由try-finally或try-catch保护，以确保在必要时释放锁定。

compact1, compact2, compact3
java.util.concurrent.locks

Interface Lock

可重入锁（常用）

所有已知实现类：

ReentrantLock , ReentrantReadWriteLock.ReadLock , ReentrantReadWriteLock.WriteLock

读锁

写锁

```
Creates an instance of ReentrantLock. This is equivalent to
using ReentrantLock(false).

public ReentrantLock() {
    sync = new NonfairSync();           非公平锁
}

Creates an instance of ReentrantLock with the given fairness
policy.

Params: fair - true if this lock should use a fair ordering
policy

public ReentrantLock(boolean fair) {      公平锁
    sync = fair ? new FairSync() : new NonfairSync();
}
```

公平锁：十分公平：可以先来后到
非公平锁：十分不公平：可以插队（默认）

```

35         e.printStackTrace();
36     } finally {
37         lock.unlock();
38     }
39 }
40 }
```

问题存在，ABCD四个线程！怎么解决？

线程也可以唤醒，而不会被通知，中断或超时，即所谓的虚假唤醒。虽然这在实践中很少会发生，但应用程序必须通过测试应该使线程被唤醒的条件来防范，并且如果条件不满足则继续等待。换句话说，等待应该总是出现在循环中，就像这样：

```

synchronized (obj) {
    while (<condition does not hold>
        obj.wait(timeout);
    ... // Perform action appropriate to condition
}
```

注意到，一次性判断应该变为循环判断
if==>while

(有关此主题的更多信息，请参阅Doug Lea的“Java并行编程（第二版）”(Addison-Wesley, 2000) 中的第3.2.3节或Joshua Bloch的“有效Java编程语言指南”(Addison-Wesley, 2001)。

if ==>while

```

1 package com.godfrey.proc;
2
3 /**
4 * description : synchronized版生成者消费者问题
5 *
6 * @author godfrey
7 * @since 2020-05-14
8 */
9 public class A {
10     public static void main(String[] args) {
11         Data data = new Data();
12
13         new Thread(() -> {
14             for (int i = 0; i < 10; i++) {
15                 try {
16                     data.increment();
17                 } catch (InterruptedException e) {
18                     e.printStackTrace();
19                 }
20             }
21         }, "A").start();
22
23         new Thread(() -> {
24             for (int i = 0; i < 10; i++) {
25                 try {
26                     data.increment();
27                 } catch (InterruptedException e) {
28                     e.printStackTrace();
29                 }
30             }
31         }, "B").start();
32
33         new Thread(() -> {
34             for (int i = 0; i < 10; i++) {
35                 try {
```

```

36             data.decrement();
37         } catch (InterruptedException e) {
38             e.printStackTrace();
39         }
40     }
41 }, "C").start();
42
43 new Thread(() -> {
44     for (int i = 0; i < 10; i++) {
45         try {
46             data.decrement();
47         } catch (InterruptedException e) {
48             e.printStackTrace();
49         }
50     }
51 }, "D").start();
52 }
53 }
54
55 // 判断等待, 业务, 通知
56 class Data {
57     private int number = 0;
58
59     //+1
60     public synchronized void increment() throws InterruptedException {
61         while (number != 0) {
62             //等待
63             this.wait();
64         }
65         number++;
66         System.out.println(Thread.currentThread().getName() + "=>" +
number);
67         //通知其他线程, 我+1完毕了
68         this.notifyAll();
69     }
70
71     //-1
72     public synchronized void decrement() throws InterruptedException {
73         while (number == 0) {
74             //等待
75             this.wait();
76         }
77         number--;
78         System.out.println(Thread.currentThread().getName() + "=>" +
number);
79         //通知其他线程, 我-1完毕了
80         this.notifyAll();
81     }
82 }

```

JUC版的生产者和消费者问题

通过Lock

```

class BoundedBuffer {
    final Lock lock = new ReentrantLock();
    final Condition notFull = lock.newCondition();
    final Condition notEmpty = lock.newCondition();

    final Object[] items = new Object[100];
    int putptr, takeptr, count;

    public void put(Object x) throws InterruptedException {
        lock.lock(); try {
            while (count == items.length)
                notFull.await();
            items[putptr] = x;
            if (++putptr == items.length) putptr = 0;
            ++count;
            notEmpty.signal();
        } finally { lock.unlock(); }
    }

    public Object take() throws InterruptedException {
        lock.lock(); try {
            while (count == 0)
                notEmpty.await();
            Object x = items[takeptr];
            if (++takeptr == items.length) takeptr = 0;
            --count;
            notFull.signal();
            return x;
        } finally { lock.unlock(); }
    }
}

```

```

1 package com.godfrey.proc;
2
3 /**
4 * description : Lock版生产者消费者问题
5 *
6 * @author godfrey
7 * @since 2020-05-14
8 */
9
10 import java.util.concurrent.locks.Condition;
11 import java.util.concurrent.locks.Lock;
12 import java.util.concurrent.locks.ReentrantLock;
13
14 public class B {
15     public static void main(String[] args) {
16         Data2 data = new Data2();
17
18         new Thread(() -> {
19             for (int i = 0; i < 10; i++) {

```

```

20         try {
21             data.increment();
22         } catch (InterruptedException e) {
23             e.printStackTrace();
24         }
25     }
26 }, "A").start();
27
28 new Thread(() -> {
29     for (int i = 0; i < 10; i++) {
30         try {
31             data.increment();
32         } catch (InterruptedException e) {
33             e.printStackTrace();
34         }
35     }
36 }, "B").start();
37
38 new Thread(() -> {
39     for (int i = 0; i < 10; i++) {
40         try {
41             data.decrement();
42         } catch (InterruptedException e) {
43             e.printStackTrace();
44         }
45     }
46 }, "C").start();
47
48 new Thread(() -> {
49     for (int i = 0; i < 10; i++) {
50         try {
51             data.decrement();
52         } catch (InterruptedException e) {
53             e.printStackTrace();
54         }
55     }
56 }, "D").start();
57 }
58 }
59
// 判断等待, 业务, 通知
60 class Data2 {
61     private int number = 0;
62
63     private Lock lock = new ReentrantLock();
64     private Condition condition = lock.newCondition();
65
66
67 //+1
68 public void increment() throws InterruptedException {
69     lock.lock();
70     try {
71         while (number != 0) {
72             //等待
73             condition.await();
74         }
75         number++;
76         System.out.println(Thread.currentThread().getName() + "=>" +
77             number);

```

```
77         //通知其他线程，我+1完毕了
78         condition.signalAll();
79     } catch (Exception e) {
80         e.printStackTrace();
81     } finally {
82         lock.unlock();
83     }
84 }
85
86 // -1
87 public void decrement() throws InterruptedException {
88     lock.lock();
89     try {
90         while (number == 0) {
91             //等待
92             condition.await();
93         }
94         number--;
95         System.out.println(Thread.currentThread().getName() + "=>" +
number);
96         //通知其他线程，我-1完毕了
97         condition.signalAll();
98     } catch (Exception e) {
99         e.printStackTrace();
100    } finally {
101        lock.unlock();
102    }
103 }
104 }
```

Condition的优势：精准通知和唤醒线程

The terminal window shows the following output:

```
G:\lib\java-lib\Java\jdk-11\b  
A==>1  
C==>0  
B==>1  
D==>0  
B==>1  
D==>0
```

The code editor shows the following Java code:

```
1 package com.godfrey.proc;  
2  
3 import java.util.concurrent.locks.Condition;  
4 import java.util.concurrent.locks.Lock;  
5 import java.util.concurrent.locks.ReentrantLock;  
6  
7 /**  
8 * description : 按顺序执行 A->B->C  
9 *  
10 * @author godfrey  
11 * @since 2020-05-15  
12 */
```

随机的状态

有序的执行：

A

B

C

D

```
13 public class C {
14     public static void main(String[] args) {
15         Data3 data = new Data3();
16         new Thread(() -> {
17             for (int i = 0; i < 10; i++) {
18                 data.printA();
19             }
20         }, "A").start();
21
22         new Thread(() -> {
23             for (int i = 0; i < 10; i++) {
24                 data.printB();
25             }
26         }, "B").start();
27
28         new Thread(() -> {
29             for (int i = 0; i < 10; i++) {
30                 data.printC();
31             }
32         }, "C").start();
33     }
34 }
35
36 //资源类
37 class Data3 {
38     private Lock lock = new ReentrantLock();
39     private Condition condition1 = lock.newCondition();
40     private Condition condition2 = lock.newCondition();
41     private Condition condition3 = lock.newCondition();
42     private int number = 1; //1A 2B 3C
43
44     public void printA() {
45         lock.lock();
46         try {
47             //业务，判断->执行->通知
48             while (number != 1) {
49                 //等待
50                 condition1.await();
51             }
52             System.out.println(Thread.currentThread().getName() +
"=>AAAAA");
53             //通知指定的人，B
54             number = 2;
55             condition2.signal();
56         } catch (Exception e) {
57             e.printStackTrace();
58         } finally {
59             lock.unlock();
60         }
61     }
62
63     public void printB() {
64         lock.lock();
65         try {
66             //业务，判断->执行->通知
67             while (number != 2) {
68                 //等待
69                 condition2.await();
```

```

70         }
71         System.out.println(Thread.currentThread().getName() +
72             ">>BBBBB");
73             //通知指定的人, C
74             number = 3;
75             condition3.signal();
76         } catch (Exception e) {
77             e.printStackTrace();
78         } finally {
79             lock.unlock();
80         }
81     }
82     public void printC() {
83         lock.lock();
84         try {
85             //业务, 判断->执行->通知
86             while (number != 3) {
87                 //等待
88                 condition3.await();
89             }
90             System.out.println(Thread.currentThread().getName() +
91                 ">>CCCCC");
92                 //通知指定的人, C
93                 number = 1;
94                 condition1.signal();
95             } catch (Exception e) {
96                 e.printStackTrace();
97             } finally {
98                 lock.unlock();
99             }
100        }

```

3.八锁现象

```

1 package com.godfrey.lock8;
2
3 import java.util.concurrent.TimeUnit;
4
5 /**
6 * description : 8锁: 关于锁的8个问题
7 * 1.标准情况下 , 两个线程先打印发短信还是打电话? 1/发短信 2/打电话
8 * 2.sendSMS延时4秒 , 两个线程先打印发短信还是打电话? 1/发短信 2/打电话
9 *
10 * @author godfrey
11 * @since 2020-05-15
12 */
13 public class Test1 {
14     public static void main(String[] args) {
15         Phone phone = new Phone();
16
17         new Thread(() -> {
18             phone.sendSMS();
19         }, "A").start();

```

```

20
21     try {
22         TimeUnit.SECONDS.sleep(1);
23     } catch (InterruptedException e) {
24         e.printStackTrace();
25     }
26
27     new Thread(() -> {
28         phone.call();
29     }, "B").start();
30 }
31
32
33 class Phone {
34     //synchronized 锁的对象是方法的调用者!
35     public synchronized void sendSms() {
36         try {
37             TimeUnit.SECONDS.sleep(4);
38         } catch (InterruptedException e) {
39             e.printStackTrace();
40         }
41         System.out.println("发短信");
42     }
43
44     public synchronized void call() {
45         System.out.println("打电话");
46     }
47 }
```

```

1 package com.godfrey.lock8;
2
3 import java.util.concurrent.TimeUnit;
4
5 /**
6 * description : 8锁: 关于锁的8个问题
7 * 3.增加了一个普通方法后!先执行发短信还是Hello? 普通方法
8 * 4.创建两个对象, !先执行发短信还是打电话? 打电话
9 *
10 * @author godfrey
11 * @since 2020-05-15
12 */
13 public class Test2 {
14     public static void main(String[] args) {
15         //两个对象, 两个调用者, 两把锁
16         Phone2 phone1 = new Phone2();
17         Phone2 phone2 = new Phone2();
18
19         new Thread(() -> {
20             phone1.sendSms();
21         }, "A").start();
22
23         try {
24             TimeUnit.SECONDS.sleep(1);
25         } catch (InterruptedException e) {
26             e.printStackTrace();
27         }
28     }
29 }
```

```

28     new Thread(() -> {
29         //phone1.hello();
30         phone2.call();
31     }, "B").start();
32 }
33 }
34 }
35
36 class Phone2 {
37     //synchronized 锁的对象是方法的调用者!
38     public synchronized void sendSms() {
39         try {
40             TimeUnit.SECONDS.sleep(4);
41         } catch (InterruptedException e) {
42             e.printStackTrace();
43         }
44         System.out.println("发短信");
45     }
46
47     public synchronized void call() {
48         System.out.println("打电话");
49     }
50
51     //这里没有锁! 不是同步方法, 不受锁的影响
52     public void hello() {
53         System.out.println("Hello");
54     }
55 }
```

```

1 package com.godfrey.lock8;
2
3 import java.util.concurrent.TimeUnit;
4
5 /**
6 * description : 8锁: 关于锁的8个问题
7 * 5.增加两个静态的同步方法, 只要一个对象, 先打样发短信还是打电话? 发短信
8 * 6.两个对象, 增加两个静态的同步方法, 只要一个对象, 先打样发短信还是打电话? 发短信
9 *
10 * @author godfrey
11 * @since 2020-05-15
12 */
13 public class Test3 {
14     public static void main(String[] args) {
15         //两个对象, 两个调用者, 两把锁
16         //static 静态方法
17         //类一加载就有了! 锁的是class
18         //Phone3 phone = new Phone3();
19         Phone3 phone1 = new Phone3();
20         Phone3 phone2 = new Phone3();
21
22         new Thread(() -> {
23             //phone.sendSms();
24             phone1.sendSms();
25         }, "A").start();
26
27         try {
```

```

28         TimeUnit.SECONDS.sleep(1);
29     } catch (InterruptedException e) {
30         e.printStackTrace();
31     }
32
33     new Thread(() -> {
34         phone2.call();
35     }, "B").start();
36 }
37 }
38
39 class Phone3 {
40     //synchronized 锁的对象是方法的调用者!
41     public static synchronized void sendSms() {
42         try {
43             TimeUnit.SECONDS.sleep(4);
44         } catch (InterruptedException e) {
45             e.printStackTrace();
46         }
47         System.out.println("发短信");
48     }
49
50     public static synchronized void call() {
51         System.out.println("打电话");
52     }
53 }
```

```

1 package com.godfrey.lock8;
2
3 import java.util.concurrent.TimeUnit;
4
5 /**
6 * description : 8锁: 关于锁的8个问题
7 * 7.一个静态同步方法一个普通方法, 先打样发短信还是打电话? 打电话
8 * 8.一个静态同步方法一个普通方法, 两个对象, 先打样发短信还是打电话? 打电话
9 *
10 * @author godfrey
11 * @since 2020-05-15
12 */
13 public class Test4 {
14     public static void main(String[] args) {
15         //Phone4 phone = new Phone4();
16         Phone4 phone1 = new Phone4();
17         Phone4 phone2 = new Phone4();
18
19         new Thread(() -> {
20             //phone.sendSms();
21             phone1.sendSms();
22         }, "A").start();
23
24         try {
25             TimeUnit.SECONDS.sleep(1);
26         } catch (InterruptedException e) {
27             e.printStackTrace();
28         }
29 }
```

```

30         new Thread() -> {
31             phone2.call();
32         }, "B").start();
33     }
34 }
35
36 class Phone4 {
37     //静态同步方法，锁的对象是Class模板！
38     public static synchronized void sendSms() {
39         try {
40             TimeUnit.SECONDS.sleep(4);
41         } catch (InterruptedException e) {
42             e.printStackTrace();
43         }
44         System.out.println("发短信");
45     }
46
47     //普通同步方法，锁的是调用者
48     public synchronized void call() {
49         System.out.println("打电话");
50     }
51 }
```

小结：看锁的是Class还是对象，看是否同一个调用者

4.集合类不安全

List不安全

```

1 package com.godfrey.unsafe;
2
3 import java.util.*;
4 import java.util.concurrent.CopyOnWriteArrayList;
5
6 /**
7 * description : java.util.ConcurrentModificationException 并发修改异常
8 *
9 * @author godfrey
10 * @since 2020-05-15
11 */
12 public class ListTest {
13     public static void main(String[] args) {
14         //并发下 ArrayList不安全的
15         /**
16          * 解决方案：
17          * 1.List<String> list = new Vector<>();
18          * 2.List<String> list = Collections.synchronizedList(new
19          ArrayList<>());
20          * 3.List<String> list = new CopyOnWriteArrayList<>();
21          */
22     }
23 }
```

```

22     //CopyOnwrite 写入时复制cow 计算机程序设计 领域的一种优化策略
23     //多个线程调用的时候, list, 读取的时候, 固定的, 写入(覆盖)
24     //在写入的时候避免覆盖, 造成数据问题!
25     //读写分离
26     //CopyOnwriteArrayList 比 Vector 牛逼在哪里? CopyOnwriteArrayList用
27     Lock, Vector用Synchronized
28
29     List<String> list = new CopyOnwriteArrayList<>();
30
31     for (int i = 0; i < 10; i++) {
32         new Thread(() -> {
33             list.add(UUID.randomUUID().toString().substring(0, 5));
34             System.out.println(list);
35         }, String.valueOf(i)).start();
36     }
37 }
38

```

Set不安全

```

1 package com.godfrey.unsafe;
2
3 import java.util.set;
4 import java.util.UUID;
5 import java.util.concurrent.CopyOnwriteArrayset;
6
7 /**
8 * description : java.util.ConcurrentModificationException 并发修改异常
9 *
10 * @author godfrey
11 * @since 2020-05-15
12 */
13 public class SetTest {
14     public static void main(String[] args) {
15         //HashSet<String> set = new HashSet<>();
16         //并发下 HashSet不安全的
17         /**
18          * 解决方案:
19          * 1. Set<String> set = Collections.synchronizedSet(new HashSet<>()
20          * );
21          * 2. Set<String> set = new CopyOnwriteArrayset<>();
22          */
23
24         Set<String> set = new CopyOnwriteArrayset<>();
25         for (int i = 0; i < 100; i++) {
26             new Thread(() -> {
27                 set.add(UUID.randomUUID().toString().substring(0, 5));
28                 System.out.println(set);
29             }, String.valueOf(i)).start();
30         }
31     }
32 }

```

问：HashSet的底层是什么？

答：HashMap

```
1 public HashSet() {
2     map = new HashMap<>();
3 }
4
5 //add set的本质是map key
6 public boolean add(E e) {
7     return map.put(e, PRESENT)==null;
8 }
9
10 private static final Object PRESENT = new Object();
```

Map不安全

```
1 package com.godfrey.unsafe;
2
3 import java.util.Map;
4 import java.util.UUID;
5 import java.util.concurrent.ConcurrentHashMap;
6
7 /**
8 * description : java.util.ConcurrentModificationException 并发修改异常
9 *
10 * @author godfrey
11 * @since 2020-05-15
12 */
13 public class MapTest {
14     public static void main(String[] args) {
15         // Map<String, String> map= new HashMap<>();
16         // 等价于 Map<String, String> map = new HashMap<>(16,0.75f); //加载因子, 初始化容量
17
18
19         //并发下 HashMap不安全的
20         /**
21          * 解决方案:
22          * 1.Map<String, String> map = Collections.synchronizedMap(new
23 HashMap<>());
24          * 2.Map<String, String> map = new ConcurrentHashMap<>();
25          */
26         Map<String, String> map = new ConcurrentHashMap<>();
27         for (int i = 0; i < 30; i++) {
28             new Thread(() -> {
29                 map.put(Thread.currentThread().getName(),
30                     UUID.randomUUID().toString().substring(0, 5));
31                 System.out.println(map);
32             }, String.valueOf(i)).start();
33         }
34     }
35 }
```

5.Callable

```
@FunctionalInterface  
public interface Callable<V>
```

返回结果并可能引发异常的任务。实现者定义一个没有参数的单一方法，称为call。

Callable接口类似于Runnable，因为它们都是为其实例可能由另一个线程执行的类设计的。然而，A Runnable不返回结果，也不能抛出被检查的异常。

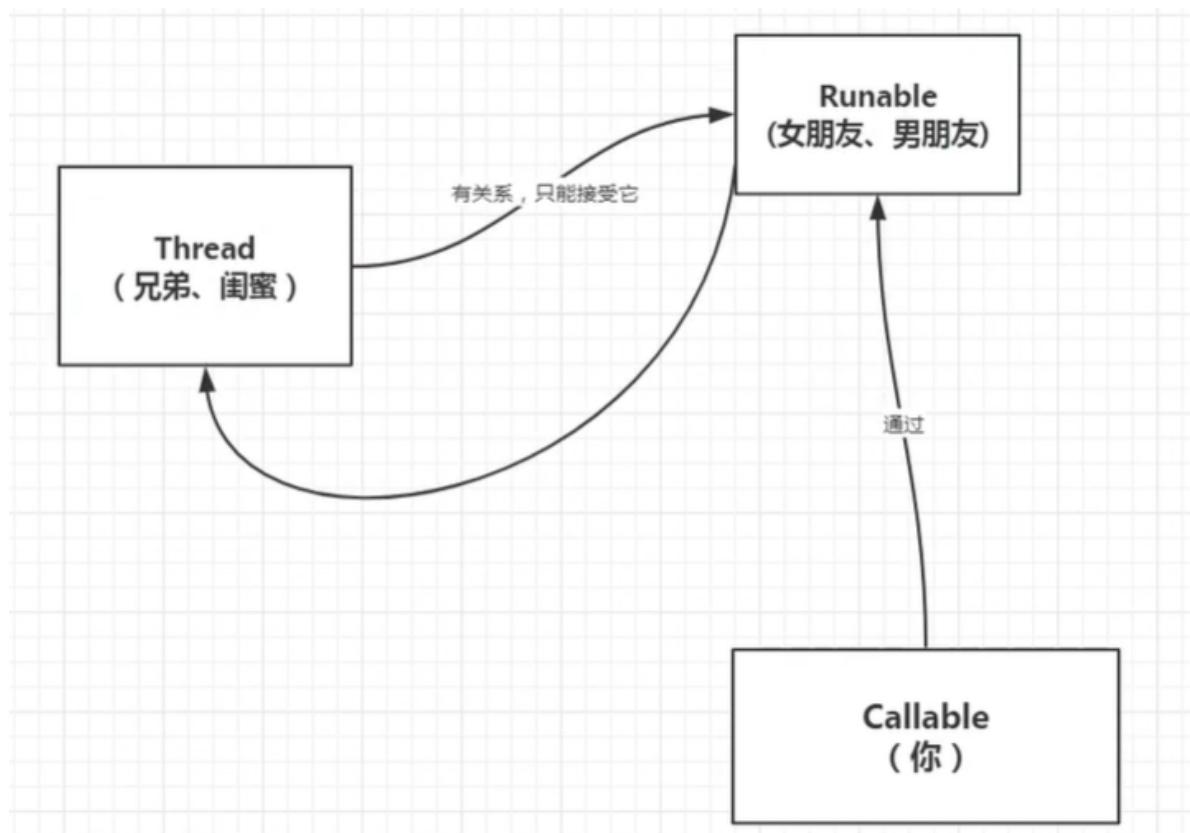
该Executors类包含的实用方法，从其他普通形式转换为Callable类。

从以下版本开始：

1.5

1. 有返回值
2. 可以抛出异常
3. 方法不同，run()/call()

代码测试



All Known Subinterfaces:

RunnableFuture <V>, RunnableScheduledFuture <V>

所有已知实现类：

AsyncBoxView.ChildState, ForkJoinWorkerThread, FutureTask, RenderableImageProducer, SwingWorker, Thread, TimerTask

Functional Interface:

这是一个功能界面，因此可以用作lambda表达式或方法引用的赋值对象。

构造方法摘要

构造方法

Constructor and Description

`FutureTask(Callable<V> callable)`

创建一个 FutureTask，它将在运行时执行给定的 Callable。

`FutureTask(Runnable runnable, V result)`

创建一个 FutureTask，将在运行时执行给定的 Runnable，并安排 get将在成功完成后返回给定的结果。

```
1 package com.godfrey.callable;
2
3 import java.util.concurrent.Callable;
4 import java.util.concurrent.ExecutionException;
5 import java.util.concurrent.FutureTask;
6
7 /**
8 * description : Callable测试
9 *
10 * @author godfrey
11 * @since 2020-05-15
12 */
13 public class CallableTest {
14     public static void main(String[] args) throws ExecutionException,
15     InterruptedException {
16         //new Thread(new Runnable).start();
17         //new Thread(new FutureTask<V>()).start();
18         //new Thread(new FutureTask<V>(callable)).start();
19
20         MyThread thread = new MyThread();
21         FutureTask<Integer> futureTask = new FutureTask<Integer>(thread); //适配类
22
23         new Thread(futureTask, "A").start();
24         new Thread(futureTask, "B").start(); //结果会被缓存，提高效率，最后打印只有一份
25
26         Integer integer = futureTask.get(); //获取Callable的返回结果（get方法可能会产生阻塞【大数据等待返回结果慢】！把它放到最会，或者用异步通信）
27         System.out.println(integer);
28     }
29
30     class MyThread implements Callable<Integer> {
31         @Override
32         public Integer call() {
33             System.out.println("call()");
34             return 1024;
35         }
36     }
}
```

细节：

1. 有缓存
2. get(), 结果可能会等待, 会阻塞

6. 常用辅助类 (必会)

6.1 CountDownLatch

```
public class CountDownLatch  
extends Object
```

允许一个或多个线程等待直到在其他线程中执行的一组操作完成的同步辅助。

A CountDownLatch用给定的计数初始化。await方法阻塞，直到由于countDown()方法的调用而导致当前计数达到零，之后所有等待线程被释放，并且任何后续的await调用立即返回。这是一个一次性的现象 - 计数无法重置。如果您需要重置计数的版本，请考虑使用CyclicBarrier。

A CountDownLatch是一种通用的同步工具，可用于多种用途。一个CountDownLatch为一个计数的CountDownLatch用作一个简单的开/关锁存器，或者门：所有线程调用await在门口等待，直到被调用countDown()的线程打开。一个CountDownLatch初始化N可以用来做一个线程等待，直到N个线程完成某项操作，或某些动作已经完成N次。

CountDownLatch一个有用的属性是，它不要求调用countDown()线程等待计数到达零之前继续，它只是阻止任何线程通过await，直到所有线程可以通过。

示例用法：这是一组类，其中一组工作线程使用两个倒计时锁存器：

- 第一个是启动信号，防止任何工作人员进入，直到驾驶员准备好继续前进；
- 第二个是完成信号，允许司机等到所有的工作人员完成。

```
class Driver { // ... void main() throws InterruptedException { CountDownLatch startSignal = new CountDownLatch(1); CountDownLatch doneSignal = new CountDownLatch(2); }
```

```
1 package com.godfrey.add;  
2  
3 import java.util.concurrent.CountDownLatch;  
4  
5 /**  
6  * description : 减法计数器  
7  *  
8  * @author godfrey  
9  * @since 2020-05-15  
10 */  
11 public class CountDownLatchDemo {  
12     public static void main(String[] args) throws InterruptedException {  
13         //总数是6  
14         CountDownLatch countDownLatch = new CountDownLatch(6);  
15  
16         for (int i = 0; i < 6; i++) {  
17             new Thread(() -> {  
18                 System.out.println(Thread.currentThread().getName() + "\tGo  
Out");  
19                 countDownLatch.countDown(); // -1  
20             }, String.valueOf(i)).start();  
21         }  
22  
23         countDownLatch.await(); // 等待计数器归零，然后再向下执行  
24         System.out.println("Close Door");  
25     }  
26 }
```

原理：

```
countDownLatch.countDown() //数量-1
```

```
countDownLatch.await() //等待计数器归零，然后再向下执行
```

每次有线程调用countDown()数量-1，假设计数器变为0，countDownLatch.await()就会被唤醒，继续执行！

6.2 CyclicBarrier

```
public class CyclicBarrier  
extends Object
```

允许一组线程全部等待彼此达到共同屏障点的同步辅助。循环阻塞在涉及固定大小的线程方的程序中很有用，这些线程必须偶尔等待彼此。屏障被称为 **循环**，因为它可以在等待的线程被释放之后重新使用。

A CyclicBarrier支持一个可选的Runnable命令，每个屏障点运行一次，在派对中的最后一个线程到达之后，但在任何线程释放之前。在任何一方继续进行之前，此屏障操作对更新共享状态很有限。

示例用法：以下是在并行分解设计中使用障碍的示例：

```
class Solver { final int N; final float[][] data; final CyclicBarrier barrier; class Worker implements Runnable { int myRow; Worker(int row) { myRow = r
```

加法计数器

```
1 package com.godfrey.add;  
2  
3 import java.util.concurrent.BrokenBarrierException;  
4 import java.util.concurrent.CyclicBarrier;  
5  
6 /**  
7  * description : 加法计数器  
8  *  
9  * @author godfrey  
10 * @since 2020-05-15  
11 */  
12 public class CyclicBarrierDemo {  
13     public static void main(String[] args) {  
14         /**  
15          * 集齐七颗龙珠召唤神龙  
16          * 集齐龙珠的线程  
17          */  
18         CyclicBarrier cyclicBarrier = new CyclicBarrier(7, () -> {  
19             System.out.println("召唤神龙成功");  
20         });  
21  
22         for (int i = 0; i < 7; i++) {  
23             final int temp = i;//lambda操作不到i  
24             new Thread(() -> {  
25                 System.out.println(Thread.currentThread().getName() + "收集"  
+ temp + "个龙珠");  
26  
27                 try {  
28                     cyclicBarrier.await(); //等待  
29                 } catch (InterruptedException e) {  
30                     e.printStackTrace();  
31                 } catch (BrokenBarrierException e) {  
32                     e.printStackTrace();  
33                 }  
34             }, String.valueOf(i)).start();  
35         }  
36     }  
37 }
```

6.3 Semaphore

Semaphore:信号量

```
public class Semaphore  
extends Object  
implements Serializable
```

一个计数信号量。在概念上，信号量维持一组许可证。如果有必要，每个acquire()都会阻塞，直到许可证可用，然后才能使用它。每个release()添加许可证，潜在地释放阻塞获取方。但是，没有使用实际的许可证对象；Semaphore只保留可用数量的计数，并相应地执行。

信号量通常用于限制线程数，而不是访问某些（物理或逻辑）资源。例如，这是一个使用信号量来控制对一个项目池的访问的类：

```
class Pool { private static final int MAX_AVAILABLE = 100; private final Semaphore available = new Semaphore(MAX_AVAILABLE, true); public Object getItem
```

在获得项目之前，每个线程必须从信号量获取许可证，以确保某个项目可用。当线程完成该项目后，它将返回到池中，并将许可证返回到信号量，允许另一个线程获取该项目。请注意，当调用acquire()时，不会保持同步锁定，因为这将阻止某个项目返回到池中。信号量封装了限制对池的访问所需的同步，与保持池本身一致性所需的任何同步分开。

信号量被初始化为一个，并且被使用，使得它只有至多一个许可证可用，可以用作互斥锁。这通常被称为二进制信号量，因为它只有两个状态：一个许可证可用，或零个许可证可用。当以这种方式使用时，二进制信号量具有属性（与许多Lock实现不同），“锁”可以由除所有者之外的线程释放（因为信号量没有所有权概念）。这在某些专门的上下文中是有用的，例如死锁恢复。

此类的构造函数可选择接受公平参数。当设置为false时，此类不会保证线程获取许可的顺序。特别是，插入是允许的，也就是说，一个线程调用acquire()可以提前已经等待线程分配的许可证，在等待线程队列的头部逻辑新的线程将自己。当公平设置为true时，信号量保证调用acquire方法的线程被选择以按照它们调用这些方法的顺序获得许可（先进先出；FIFO）。请注意，FIFO排序必须适用于这些方法中的特定内部执行点。因此，一个线程可以在另一个线程之前调用acquire，但是在另一个线程之后到达排序点，并且类似地从方法返回。另请注意，未定义的tryAcquire方法不符合公平性设置，但将采取任何可用的许可证。

通常，用于控制资源访问的信号量应该被公平地初始化，以确保线程没有被访问资源。当使用信号量进行其他类型的同步控制时，非正常排序的吞吐量优势往往超过公平性。

抢车位！

```
1 package com.godfrey.add;  
2  
3 import java.util.concurrent.Semaphore;  
4 import java.util.concurrent.TimeUnit;  
5  
6 /**  
7  * description : 信号量  
8  *  
9  * @author godfrey  
10 * @since 2020-05-15  
11 */  
12 public class SemaphoreDemo {  
13     public static void main(String[] args) {  
14         //线程数量：停车位！限流！  
15         Semaphore semaphore = new Semaphore(3);  
16  
17         for (int i = 0; i < 6; i++) {  
18             new Thread(() -> {  
19                 //acquire() 得到  
20                 try {  
21                     semaphore.acquire();  
22                     System.out.println(Thread.currentThread().getName() +  
23                         "抢到车位");  
24                     TimeUnit.SECONDS.sleep(2);  
25                     System.out.println(Thread.currentThread().getName() +  
26                         "离开车位");  
27                 } catch (InterruptedException e) {  
28                     e.printStackTrace();  
29                 } finally {  
29                     //release() 释放  
30                     semaphore.release();  
31                 }  
32             });  
33         }  
34     }  
35 }
```

```

30 }
31
32     }) .start();
33 }
34 }
35 }

```

原理:

semaphore.acquire() 获得, 假设如果已经满了, 等待, 等待被释放为止!

semaphore.release() 释放, 会将当前的信号量释放+ 1, 然后唤醒等待的线程!

作用:

1. 多个共享资源互斥的使用!
2. 并发限流, 控制最大的线程数!

7. 读写锁

ReadWriteLock

```

java.util.concurrent.locks
Interface ReadWriteLock

所有已知实现类:
ReentrantReadWriteLock

```

QQ一群: 732712
QQ二群: 864725
安卓帮助文!

可以被多个线程同时读
只能被一个线程同时写

```

public interface ReadWriteLock

```

A **ReadWriteLock** 维护一对关联的 locks, 一个用于读操作, 一个用于写入。 **read lock** 可以由多个阅读器线程同时进行, 只要没有作者。 **write lock** 是独占的。

所有 **ReadWriteLock** 实现必须保证的存储器同步效应 **writeLock** 操作 (如在指定 Lock 接口) 也保持相对于所述相关联的 **readLock**。 也就是说, 一个线程成功获取读锁定将会看到在之前发布的写锁定所做的所有更新。

读写锁允许访问共享数据时的并发性高于互斥锁所允许的并发性。 它利用了这样一个事实: 一次只有一个线程 (写入线程) 可以修改共享数据, 在许多情况下, 任何数量的线程都可以同时读取数据 (因此读锁是公平的)。 从理论上讲, 通过使用读写锁允许的并发性增加将导致性能改进超过使用互斥锁。 实际上, 并发性的增加只能在多处理器上完全实现, 然后只有在共享数据的访问模式是合适的时才可以。

读写锁是否会提高使用互斥锁的性能取决于数据被读取的频率与被修改的频率相比, 读取和写入操作的持续时间以及数据的争用 - 即是, 将尝试同时读取或写入数据的线程数。 例如, 最初填充数据的集合, 然后经常被修改的频繁搜索 (例如某种目录) 是使用读写锁的理想候选。 然而, 如果更新变得频繁, 那么数据的大部分时间将被写入锁定, 并且并发性增加很少。 此外, 如果读取操作太短, 则读写锁定实现 (其本身比互斥锁更复杂) 的开销可以支配执行成本, 特别是因为许多读写锁定实现仍将序列化所有线程通过小部分代码。 最终, 只有剖析和测量将确定使用读写锁是否适合您的应用程序。

虽然读写锁的基本操作是直接的, 但是执行必须做出许多策略决策, 这可能会影响给定应用程序中读写锁定的有效性。 这些政策的例子包括:

- 在写入器释放写入锁定时, 确定在读取器和写入器都在等待时是否授予读取锁定或写入锁定。 作家偏好是常见的, 因为写作预计会很短, 很少见。 读者喜好不常见, 因为如果读者经常和长期的抒情期, 写作可能导致漫长的延迟。 公平的或“按顺序”的实现也是可能的。
- 确定在读卡器处于活动状态并且写入器正在等待时请求读取锁定的读取器是否被授予读取锁定。 读者的偏好可以无限期地拖延作者, 而对作者的偏好可以减少并发的潜力。
- 确定锁是否可重入: 一个具有写锁的线程是否可以重新获取? 持有写锁可以获取读锁吗? 锁本身是否可重入?
- 写入锁可以降级到读锁, 而不允许插入写者? 读锁可以升级到写锁, 优先于其他等待读者或作者吗?

在评估应用程序的给定实现的适用性时, 应考虑所有这些问题。

从以下版本开始:

1.5

jdk
中
英
文

```

1 package com.godfrey.rw;
2
3 import java.util.HashMap;
4 import java.util.Map;
5 import java.util.concurrent.locks.ReentrantReadWriteLock;
6
7 /**
8 * description : 读写锁
9 * 独占锁 (写锁) 一次只能被一个线程占有
10 * 共享锁 (读锁) 多个线程可以同时占有
11 * ReadWriteLock
12 * 读-读 可以共存!

```

```
13 * 读-写 不能共存!
14 * 写-写 不能共存!
15 *
16 * @author godfrey
17 * @since 2020-05-15
18 */
19 public class ReadwriteLockDemo {
20     public static void main(String[] args) {
21         MyCacheLock myCache = new MyCacheLock();
22
23         //写入
24         for (int i = 0; i < 10; i++) {
25             final int temp = i;
26             new Thread(() -> {
27                 myCache.put(temp + "", temp + "");
28             }, String.valueOf(i)).start();
29         }
30
31         //读取
32         for (int i = 0; i < 10; i++) {
33             final int temp = i;
34             new Thread(() -> {
35                 myCache.get(temp + "");
36             }, String.valueOf(i)).start();
37         }
38     }
39 }
40
41 //加锁的
42 class MyCacheLock {
43     private volatile Map<String, Object> map = new HashMap<>();
44     //读写锁, 更加细粒度的控制
45     private ReentrantReadWriteLock readwriteLock = new
ReentrantReadWriteLock();
46
47     // 存, 写入的时候, 只希望同时只有一个线程写
48     public void put(String key, Object value) {
49         readwriteLock.writeLock().lock();
50         try {
51             System.out.println(Thread.currentThread().getName() + "写入" +
key);
52             map.put(key, value);
53             System.out.println(Thread.currentThread().getName() + "写入OK");
54         } catch (Exception e) {
55             e.printStackTrace();
56         } finally {
57             readwriteLock.writeLock().unlock();
58         }
59     }
60
61     // 取, 读, 所有人都可以
62     public void get(String key) {
63         readwriteLock.readLock().lock();
64         try {
65             System.out.println(Thread.currentThread().getName() + "读入" +
key);
66             Object o = map.get(key);
67         }
```

```

68         System.out.println(Thread.currentThread().getName() + "读入OK");
69     } catch (Exception e) {
70         e.printStackTrace();
71     } finally {
72         readwriteLock.readLock().unlock();
73     }
74 }
75
76 /**
77 * 自定义缓存
78 */
79
80 class MyCache {
81     private volatile Map<String, Object> map = new HashMap<>();
82
83     //存，写
84     public void put(String key, Object value) {
85         System.out.println(Thread.currentThread().getName() + "写入" + key);
86         map.put(key, value);
87         System.out.println(Thread.currentThread().getName() + "写入OK");
88     }
89
90     //取，读
91     public void get(String key) {
92         System.out.println(Thread.currentThread().getName() + "读入" + key);
93         Object o = map.get(key);
94         System.out.println(Thread.currentThread().getName() + "读入OK");
95     }
96 }

```

8.阻塞队列

FIFO

不得不阻塞

写入：如果队列满了，就必须阻塞等待

取：如果是队列是空的，必须阻塞等待生产

阻塞队列：

参数类型

E - 此集合中保存的元素的类型

All Superinterfaces:

Collection <E>, Iterable <E>, Queue <E>

All Known Subinterfaces:

BlockingDeque <E>, TransferQueue <E>

所有已知实现类 :

ArrayBlockingQueue, DelayQueue, LinkedBlockingDeque, LinkedBlockingQueue, LinkedTransferQueue, PriorityBlockingQueue, SynchronousQueue

同步队列

java.util

QQ群 : 8647

Interface Queue<E>

安卓帮助

参数类型

E - 保存在此集合中的元素的类型

All Superinterfaces:

Collection <E>, Iterable <E>

阻塞队列

All Known Subinterfaces:

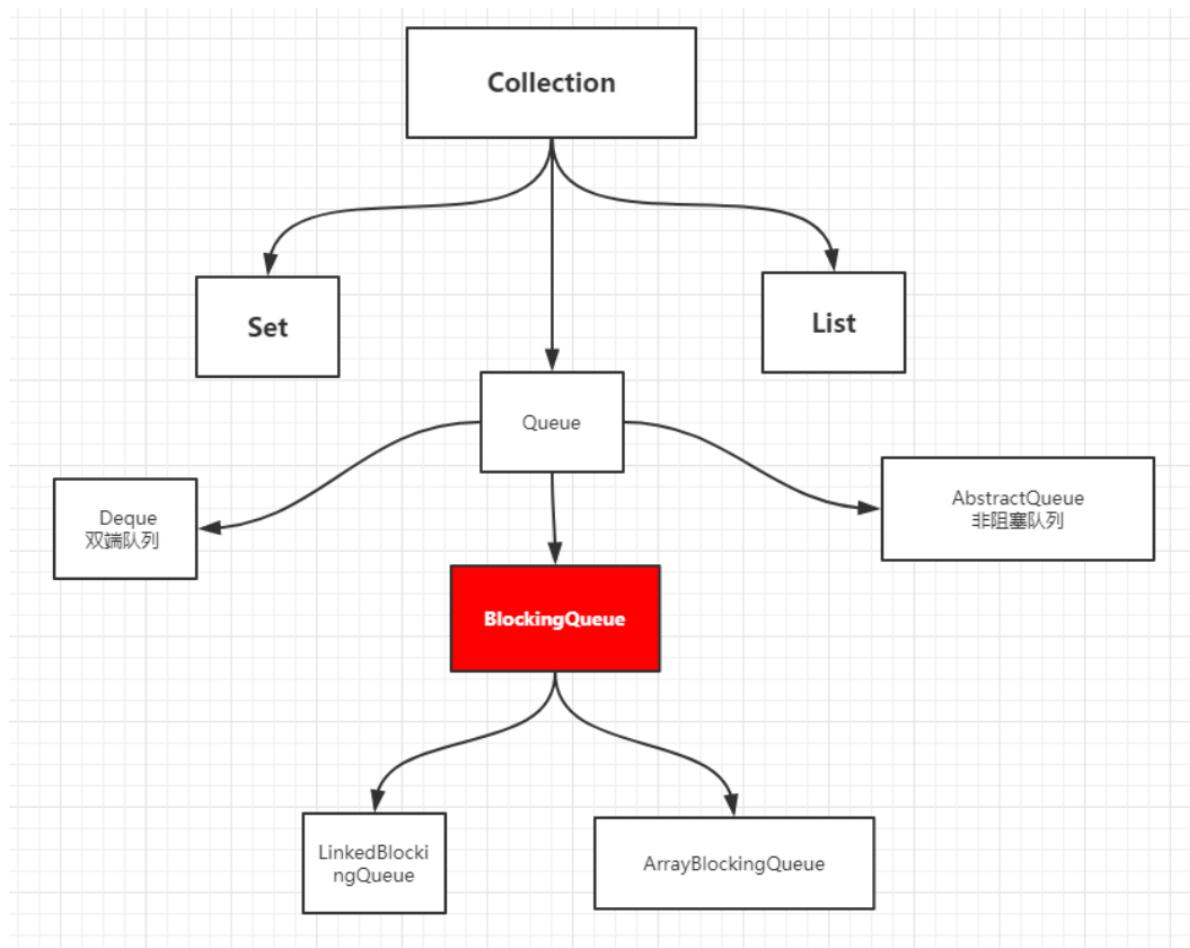
BlockingDeque <E>, BlockingQueue <E>, Deque <E>, TransferQueue <E>

双端队列

所有已知实现类 :

AbstractQueue, ArrayBlockingQueue, ArrayDeque, ConcurrentLinkedDeque, ConcurrentLinkedQueue, DelayQueue, LinkedBlockingDeque, LinkedBlockingQueue, LinkedList, LinkedTransferQueue, PriorityBlockingQueue, PriorityQueue, SynchronousQueue

非阻塞队列



什么情况下我们会使用 阻塞队列：多线程并发处理，线程池！

学会使用队列

添加、移除

四组API

方式	抛出异常	有返回值，不抛出异常	阻塞 等待	超时
添加	add	offer	put	offer
移除	remove	poll	take	poll
判断队列的首部	element	peek	-	-

```
1 /**
2  * 抛出异常
3 */
4 public static void test1() {
5     ArrayBlockingQueue<String> blockingQueue = new ArrayBlockingQueue<String>(3);
6     System.out.println(blockingQueue.add("a"));
7     System.out.println(blockingQueue.add("b"));
8     System.out.println(blockingQueue.add("c"));
9
10    //java.lang.IllegalStateException: Queue full 抛出异常！队列已满
11    //System.out.println(blockingQueue.add("d"));
12
13    System.out.println(blockingQueue.element()); //查看队首元素是谁
14    System.out.println("=====");
15
16    System.out.println(blockingQueue.remove());
17    System.out.println(blockingQueue.remove());
18    System.out.println(blockingQueue.remove());
19    //java.lang.IllegalStateException: Queue empty 抛出异常！队列为空
20    //System.out.println(blockingQueue.remove());
21 }
```

```
1 /**
2  * 有返回值，没有异常
3 */
4 public static void test2() {
5     ArrayBlockingQueue<String> blockingQueue = new ArrayBlockingQueue<String>(3);
6     System.out.println(blockingQueue.offer("a"));
7     System.out.println(blockingQueue.offer("b"));
8     System.out.println(blockingQueue.offer("c"));
9
10    //System.out.println(blockingQueue.offer("d")); // false 不抛出异常！
11
12    System.out.println(blockingQueue.peek()); //查看队首元素是谁
13    System.out.println("=====");
14
15    System.out.println(blockingQueue.poll());
16    System.out.println(blockingQueue.poll());
17    System.out.println(blockingQueue.poll());
18    //System.out.println(blockingQueue.remove()); // null 不抛出异常！
19 }
```

```

1 /**
2 * 等待, 阻塞 (一直阻塞)
3 */
4 public static void test3() throws InterruptedException {
5     // 队列的大小
6     ArrayBlockingQueue<String> blockingQueue = new ArrayBlockingQueue<>(3);
7
8     // 一直阻塞
9     blockingQueue.put("a");
10    blockingQueue.put("b");
11    blockingQueue.put("c");
12    // blockingQueue.put("d"); // 队列没有位置了, 一直阻塞
13    System.out.println(blockingQueue.take());
14    System.out.println(blockingQueue.take());
15    System.out.println(blockingQueue.take());
16    System.out.println(blockingQueue.take()); // 没有这个元素, 一直阻塞
17 }

```

```

1 /**
2 * 等待, 阻塞 (等待超市)
3 */
4 public static void test4() throws InterruptedException {
5     ArrayBlockingQueue<String> blockingQueue = new ArrayBlockingQueue<>(3);
6     blockingQueue.offer("a");
7     blockingQueue.offer("b");
8     blockingQueue.offer("c");
9
10    // blockingQueue.offer("d",2,TimeUnit.SECONDS); // 等待超过2秒就退出
11    System.out.println("=====");
12    System.out.println(blockingQueue.poll());
13    System.out.println(blockingQueue.poll());
14    System.out.println(blockingQueue.poll());
15    blockingQueue.poll(2, TimeUnit.SECONDS); // 等待超过2秒就退出
16 }

```

SynchronousQueue 同步队列

没有容量,
进去一个元素, 必须等待取出来之后, 才能再往里面放一个元素!
put、take

```

1 package com.godfrey.bq;
2
3 import java.util.concurrent.BlockingQueue;
4 import java.util.concurrent.SynchronousQueue;
5 import java.util.concurrent.TimeUnit;
6 /**
7 * description :
8 *
9 * @author godfrey
10 * @since 2020-05-15
11 */
12
13 /**
14 * 同步队列

```

```

15 * 和其他的BlockingQueue 不一样， SynchronousQueue 不存储元素
16 * put了一个元素， 必须从里面先take取出来， 否则不能在put进去值！
17 */
18 public class SynchronousQueueDemo {
19     public static void main(String[] args) {
20         BlockingQueue<String> blockingQueue = new SynchronousQueue<>(); //同步队列
21
22         new Thread(() -> {
23             try {
24                 blockingQueue.put("1");
25                 System.out.println(Thread.currentThread().getName() + " put
26 1");
27                 blockingQueue.put("2");
28                 System.out.println(Thread.currentThread().getName() + " put
29 2");
30                 blockingQueue.put("3");
31                 System.out.println(Thread.currentThread().getName() + " put
32 3");
33             } catch (InterruptedException e) {
34                 e.printStackTrace();
35             }
36         }, "T1").start();
37
38         new Thread(() -> {
39             try {
40                 TimeUnit.SECONDS.sleep(3);
41                 System.out.println(Thread.currentThread().getName() + "=>" +
42 blockingQueue.take());
43                 TimeUnit.SECONDS.sleep(3);
44                 System.out.println(Thread.currentThread().getName() + "=>" +
45 blockingQueue.take());
46                 TimeUnit.SECONDS.sleep(3);
47                 System.out.println(Thread.currentThread().getName() + "=>" +
48 blockingQueue.take());
49             } catch (InterruptedException e) {
50                 e.printStackTrace();
51             }
52         }, "T2").start();
53     }
54 }

```

9.线程池（重点）

线程池：三大方法、7大参数、4种拒绝策略

池话技术

程序的运行，本质：占用系统的资源！优化资源的使用！=>池化技术

线程池、连接池、内存池、对象池//..... 创建、销毁。十分浪费资源

池化技术：事先准备好一些资源，有人要用，就来我这里拿，用完之后还给我

线程池的好处：

1. 降低资源的消耗
2. 提高响应的速度
3. 方便管理

线程复用、可以控制最大并发数、管理线程

三大方法

4. 【强制】线程池不允许使用 `Executors` 去创建，而是通过 `ThreadPoolExecutor` 的方式，这样的处理方式让写的同学更加明确线程池的运行规则，规避资源耗尽的风险。

说明： `Executors` 返回的线程池对象的弊端如下：

1) `FixedThreadPool` 和 `SingleThreadPool`: → 约为21亿

允许的请求队列长度为 `Integer.MAX_VALUE`, 可能会堆积大量的请求，从而导致 OOM。

2) `CachedThreadPool` 和 `ScheduledThreadPool`:

允许的创建线程数量为 `Integer.MAX_VALUE`, 可能会创建大量的线程，从而导致 OOM。

```

1 package com.godfrey.pool;
2
3 import java.util.concurrent.ExecutorService;
4 import java.util.concurrent.Executors;
5
6 /**
7 * description : Executors 工具类、3大方法
8 *
9 * @author godfrey
10 * @since 2020-05-15
11 */
12 public class Demo01 {
13     public static void main(String[] args) {
14         ExecutorService threadPool = Executors.newSingleThreadExecutor(); // 单个线程
15         //ExecutorService threadPool = Executors.newFixedThreadPool(5); // 创建一个固定的线程池的大小
16         //ExecutorService threadPool = Executors.newCachedThreadPool(); // 可伸缩的，遇强则强，遇弱则弱
17
18         try {
19             for (int i = 0; i < 100; i++) {
20                 threadPool.execute(() -> {
21                     System.out.println(Thread.currentThread().getName() +
22                         "\tok");
23                 });
24             } catch (Exception e) {
25                 e.printStackTrace();
26             } finally {
27                 // 线程池用完，程序结束，关闭线程池
28                 threadPool.shutdown();
29             }
30         }
31     }

```

七大参数

源码分析：

```
1 public static ExecutorService newSingleThreadExecutor() {
2     return new FinalizableDelegatedExecutorService
3         (new ThreadPoolExecutor(1, 1,
4             0L, TimeUnit.MILLISECONDS,
5             new LinkedBlockingQueue<Runnable>()));
6 }
7
8 public static ExecutorService newFixedThreadPool(int nThreads) {
9     return new ThreadPoolExecutor(nThreads, nThreads,
10         0L, TimeUnit.MILLISECONDS,
11         new LinkedBlockingQueue<Runnable>());
12 }
13
14 public static ExecutorService newCachedThreadPool() {
15     return new ThreadPoolExecutor(0, Integer.MAX_VALUE,
16         60L, TimeUnit.SECONDS,
17         new SynchronousQueue<Runnable>());
18 }
19
20
21 //本质ThreadPoolExecutor()
22
23
24 public ThreadPoolExecutor(int corePoolSize, // 核心线程池大小
25                           int maximumPoolSize, // 最大核心线程池大小
26                           long keepAliveTime, // 超时了没有人调用就会释放
27                           TimeUnit unit, // 超时单位
28                           BlockingQueue<Runnable> workQueue, // 阻塞队列
29                           ThreadFactory threadFactory, // 线程工厂：创建线程的
一般不用动
30                           RejectedExecutionHandler handler // 拒绝策略) {
31     if (corePoolSize < 0 ||
32         maximumPoolSize <= 0 ||
33         maximumPoolSize < corePoolSize ||
34         keepAliveTime < 0)
35         throw new IllegalArgumentException();
36     if (workQueue == null || threadFactory == null || handler == null)
37         throw new NullPointerException();
38     this.corePoolSize = corePoolSize;
39     this.maximumPoolSize = maximumPoolSize;
40     this.workQueue = workQueue;
41     this.keepAliveTime = unit.toNanos(keepAliveTime);
42     this.threadFactory = threadFactory;
43     this.handler = handler;
44 }
```

4. 【强制】线程池不允许使用 `Executors` 去创建，而是通过 `ThreadPoolExecutor` 的方式，这样的处理方式让写的同学更加明确线程池的运行规则，规避资源耗尽的风险。

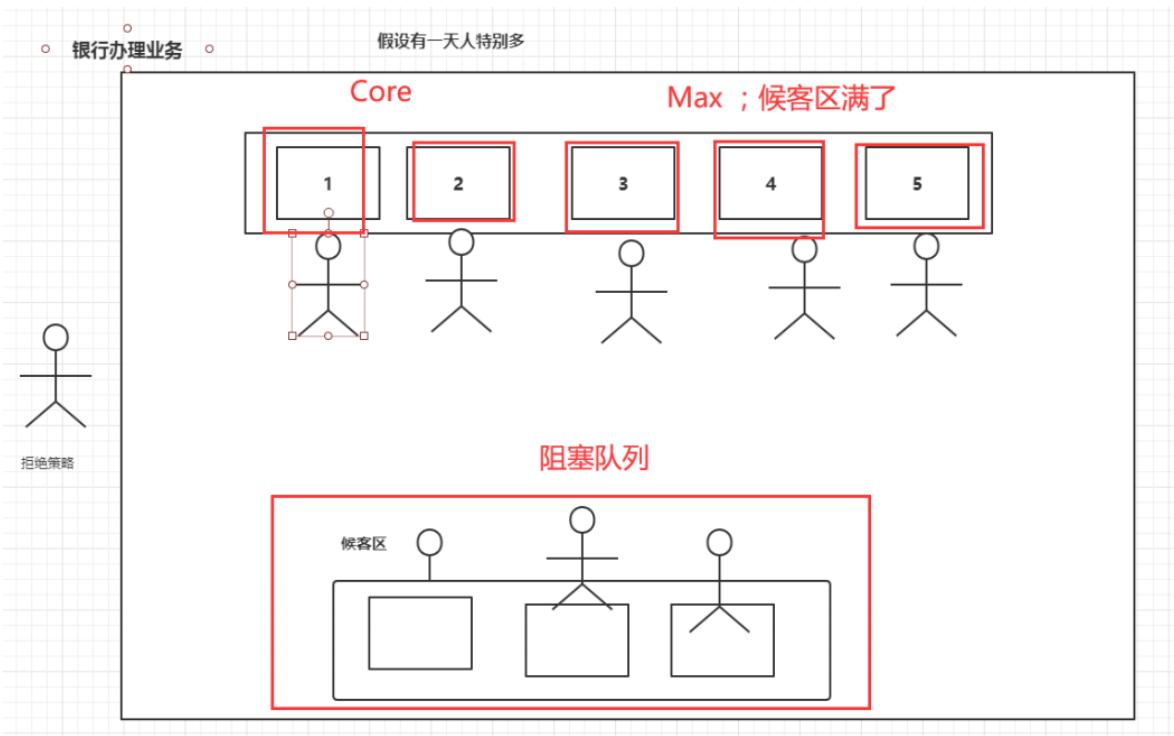
说明：`Executors` 返回的线程池对象的弊端如下：

1) `FixedThreadPool` 和 `SingleThreadPool`:

允许的请求队列长度为 `Integer.MAX_VALUE`，可能会堆积大量的请求，从而导致 OOM。

2) `CachedThreadPool` 和 `ScheduledThreadPool`:

允许的创建线程数量为 `Integer.MAX_VALUE`，可能会创建大量的线程，从而导致 OOM。



手写一个线程池

```
1 package com.godfrey.pool;
2
3 import java.util.concurrent.*;
4
5 /**
6  * description : 七大参数与四种拒绝策略
7  * 四种拒绝策略：
8  * AbortPolicy(默认)：队列满了，还有任务进来，不处理这个任务的，直接抛出
9  RejectedExecutionException异常！
10 * CallerRunsPolicy: 哪来的回哪里！
11 * DiscardOldestPolicy: 队列满了，抛弃队列中等待最久的任务，然后把当前任务加入队列中尝试
再次提交
12 * DiscardPolicy(): 队列满了，直接丢弃任务，不予任何处理也不抛出异常.如果允许任务丢失，这
是最好的拒绝策略！
13 *
14 * @author godfrey
15 * @since 2020-05-15
16 */
17 public class Demo02 {
18     public static void main(String[] args) {
```

```

18     ExecutorService threadPool = new ThreadPoolExecutor(
19         //模拟银行业务办理
20         2,      //常驻核心线程数      办理业务窗口初始数量
21         5, //线程池能够容纳同时执行的最大线程数,此值大于等于1,    办理业务窗口
22             最大数量
23             3, //多余的空闲线程存活时间,当空间时间达到keepAliveTime值时,多余的线
24             程会被销毁直到只剩下corePoolSize个线程为止      释放后窗口数量会变为常驻核心数
25             TimeUnit.SECONDS, //超时单位
26             new LinkedBlockingDeque<>(3), //任务队列,被提交但尚未被执行的任
27             务. 候客区座位数量
28             Executors.defaultThreadFactory(), //线程工厂: 创建线程的,一般不
29             用动
30             new ThreadPoolExecutor.DiscardOldestPolicy()); //拒绝策略,表示
31             当线程队列满了并且工作线程大于等于线程池的最大显示 数(maximumPoolSize)时如何来拒绝
32
33     try {
34         for (int i = 0; i < 10; i++) {
35             threadPool.execute(() -> {
36                 System.out.println(Thread.currentThread().getName() +
37                     "\tok");
38             });
39         }
40     } finally {
41         // 线程池用完, 程序结束, 关闭线程池
42         threadPool.shutdown();
43     }
44 }

```

四种拒绝策略

1. AbortPolicy(默认): 队列满了, 还有任务进来, 不处理这个任务的, 直接抛出 RejectedExecution 异常
2. CallerRunsPolicy: 哪来的回哪里!
3. DiscardOldestPolicy: 队列满了, 抛弃队列中等待最久的任务,然后把当前任务加入队列中尝试再次提交
4. DiscardPolicy(): 队列满了, 直接丢弃任务,不予任何处理也不抛出异常.如果允许任务丢失,这是最好的拒绝策略!

小结和拓展

池的最大的大小如何去设置!

获取CPU核数 `System.out.println(Runtime.getRuntime().availableProcessors())`

了解: 用来 (调优)

- CPU密集型: CPU核数+1
- IO密集型:
 - CPU核数*2

- CPU核数/1.0-阻塞系数 阻塞系数在0.8~0.9之间

10.四大函数式接口（必需掌握）

新时代的程序员：lambda表达式、链式编程、函数式接口、Stream流式计算

函数式接口：只有一个方法的接口

```
1  @FunctionalInterface
2  public interface Runnable {
3      public abstract void run();
4 }
```

Interfaces

BiConsumer
BiFunction
BinaryOperator
BiPredicate
BooleanSupplier
Consumer
DoubleBinaryOperator
DoubleConsumer
DoubleFunction
DoublePredicate
DoubleSupplier
DoubleToIntFunction
DoubleToLongFunction
DoubleUnaryOperator
Function
IntBinaryOperator
IntConsumer
IntFunction
IntPredicate
IntSupplier
IntToDoubleFunction
IntToLongFunction
IntUnaryOperator
LongBinaryOperator
LongConsumer
LongFunction
LongPredicate
LongSupplier
LongToDoubleFunction
LongToIntFunction
LongUnaryOperator
ObjDoubleConsumer
ObjIntConsumer
ObjLongConsumer
Predicate
Supplier
ToDoubleBiFunction

四大函数式
接口

代码测试

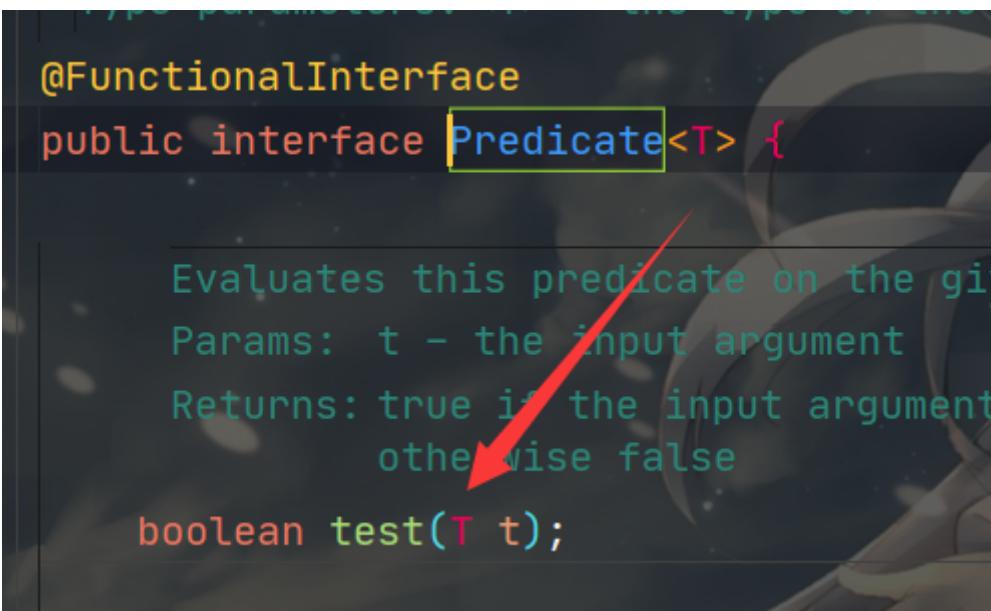
函数式接口

```
@FunctionalInterface  
public interface Function<T, R> {  
  
    Applies this function to the given argument.  
    Params: t - the function argument  
    Returns: the function result  
  
    R apply(T t);  传入参数  
}
```

返回结果

```
1 package com.godfrey.function;  
2  
3 import java.util.function.Function;  
4  
5 /**  
6 * description : Function 函数式接口,有一个输入参数,有一个输出  
7 * 只要是 函数型接口 可以 用 Lambda表达式简化  
8 *  
9 * @author godfrey  
10 * @since 2020-05-16  
11 */  
12 public class Demo01 {  
13     public static void main(String[] args) {  
14         //Function function = new Function<String, String>(){  
15         //    @Override  
16         //    public String apply(String o) {  
17         //        return o;  
18         //    }  
19         //};  
20  
21         Function function = str->{return str;};  
22  
23         System.out.println(function.apply("123"));  
24     }  
25 }
```

断定型接口：有一个输入参数，返回值只能是 布尔值！



```
1 package com.godfrey.function;
2
3 import java.util.function.Predicate;
4
5 /**
6  * description : 断定型接口,有一个输入参数,返回值只能是布尔值!
7  *
8  * @author godfrey
9  * @since 2020-05-16
10 */
11 public class Demo02 {
12     public static void main(String[] args) {
13         //判断字符串是否为空
14         //Predicate<String> predicate = new Predicate<String>() {
15         //    @Override
16         //    public boolean test(String str) {
17         //        return str.isEmpty();
18         //    }
19         //};
20
21         Predicate<String> predicate = str -> { return str.isEmpty(); };
22         System.out.println(predicate.test(""));
23     }
24 }
```

Consumer 消费型接口

```
@FunctionalInterface  
public interface Consumer<T> {  
  
    Performs this operation on the given  
    Params: t - the input argument  
    void accept(T t); 只有输入，没有返回值
```

```
1 package com.godfrey.function;  
2  
3 import java.util.function.Consumer;  
4  
5 /**  
6  * description : Consumer 消费型接口,只有输入, 没有返回值  
7  *  
8  * @author godfrey  
9  * @since 2020-05-16  
10 */  
11 public class Demo03 {  
12     public static void main(String[] args) {  
13         //Consumer<String> consumer = new Consumer<String>() {  
14         //    @Override  
15         //    public void accept(String str) {  
16         //        System.out.println(str);  
17         //    }  
18         //};  
19         Consumer<String> consumer = str -> System.out.println(str);  
20         consumer.accept("godfrey");  
21     }  
22 }
```

Supplier 供给型接口

```
@FunctionalInterface  
public interface Supplier<T> {  
  
    Gets a result.  
    Returns: a result  
    T get(); 没有参数, 只有返回值  
}
```

```
1 package com.godfrey.function;
```

```
2  
3 import java.util.function.Supplier;  
4  
5 /**  
6  * description : Supplier 供给型接口,没有参数, 只有返回值  
7  *  
8  * @author godfrey  
9  * @since 2020-05-16  
10 */  
11 public class Demo04 {  
12     public static void main(String[] args) {  
13         //Supplier supplier = new Supplier<Integer>() {  
14             //    @Override  
15             //    public Integer get() {  
16                 //        System.out.println("get()");  
17                 //        return 1024;  
18             //    }  
19             //};  
20         Supplier supplier = () -> { return 1024;};  
21         System.out.println(supplier.get());  
22     }  
23 }
```

11.Stream流式计算

什么是Stream流式计算

大数据：存储 + 计算
集合、MySQL 本质就是存储东西的；
计算都应该交给流来操作！

java.util.stream

Interface Stream<T>

参数类型

T - 流元素的类型

All Superinterfaces:

AutoCloseable , BaseStream <T, Stream <T>>

```
1 package com.godfrey.stream;  
2  
3 import java.util.Arrays;  
4 import java.util.List;  
5  
6 /**  
7  * description : 一分钟内完成此题, 只能用一行代码实现!  
8  * 现在有5个用户! 篩选:  
9  * 1、ID 必须是偶数
```

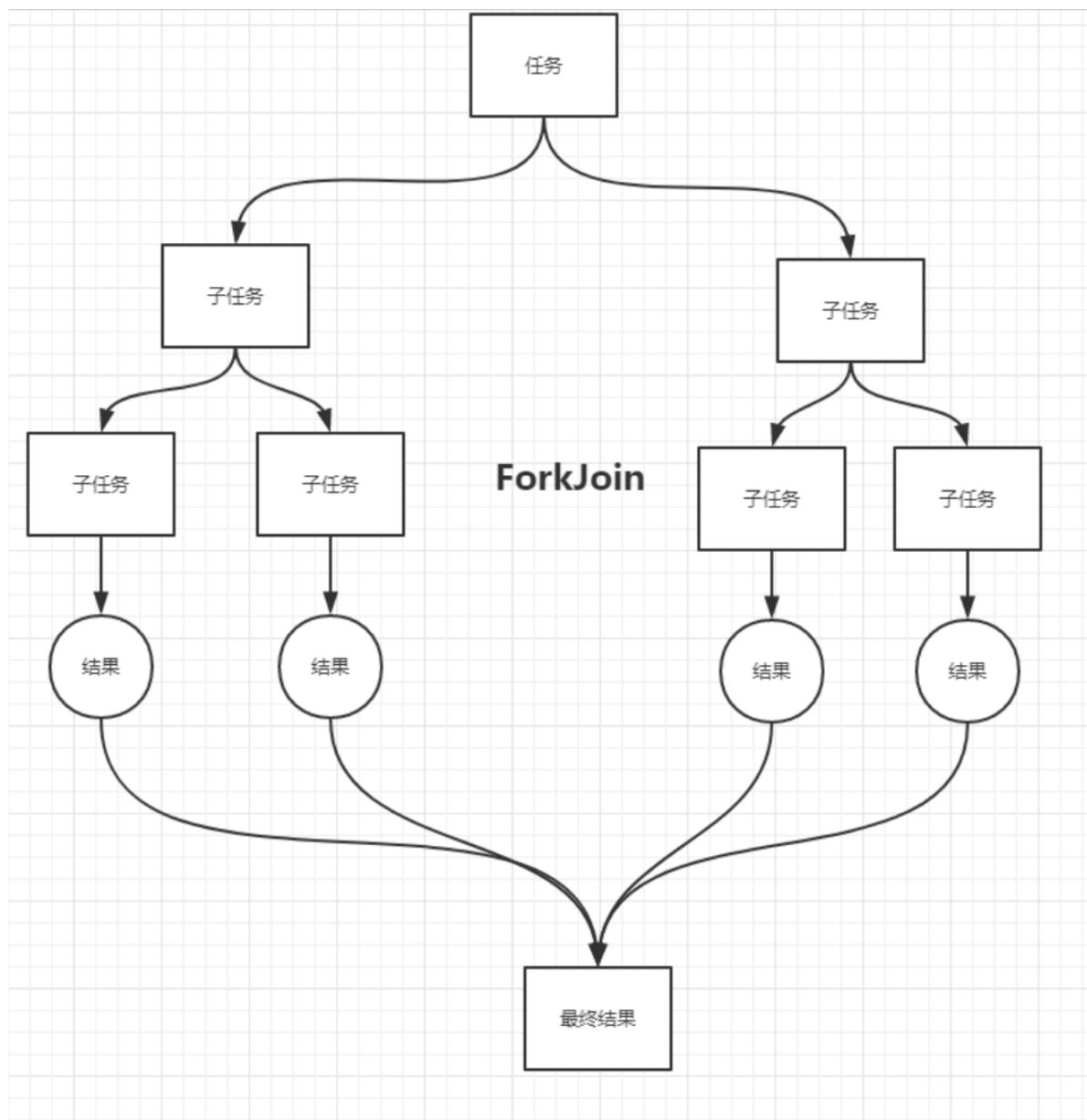
```
10 * 2、年龄必须大于23岁
11 * 3、用户名转为大写字母
12 * 4、用户名字母倒着排序
13 * 5、只输出一个用户！
14 *
15 * @author godfrey
16 * @since 2020-05-16
17 */
18 public class Test {
19     public static void main(String[] args) {
20         User u1 = new User(1, "a", 21);
21         User u2 = new User(2, "b", 22);
22         User u3 = new User(3, "c", 23);
23         User u4 = new User(4, "d", 24);
24         User u5 = new User(6, "e", 25);
25
26         //集合就算存储
27         List<User> list = Arrays.asList(u1, u2, u3, u4, u5);
28
29         //计算交给Stream流
30         list.stream()
31             .filter(u -> { return u.getId() % 2 == 0; })
32             .filter(u->{return u.getAge()>23;})
33             .map(u->{return u.getName().toUpperCase();})
34             .sorted((uu1,uu2)->{return uu2.compareTo(uu1);})
35             .limit(1)
36             .forEach(System.out::println);
37     }
38 }
```

12.ForkJoin

什么是 ForkJoin

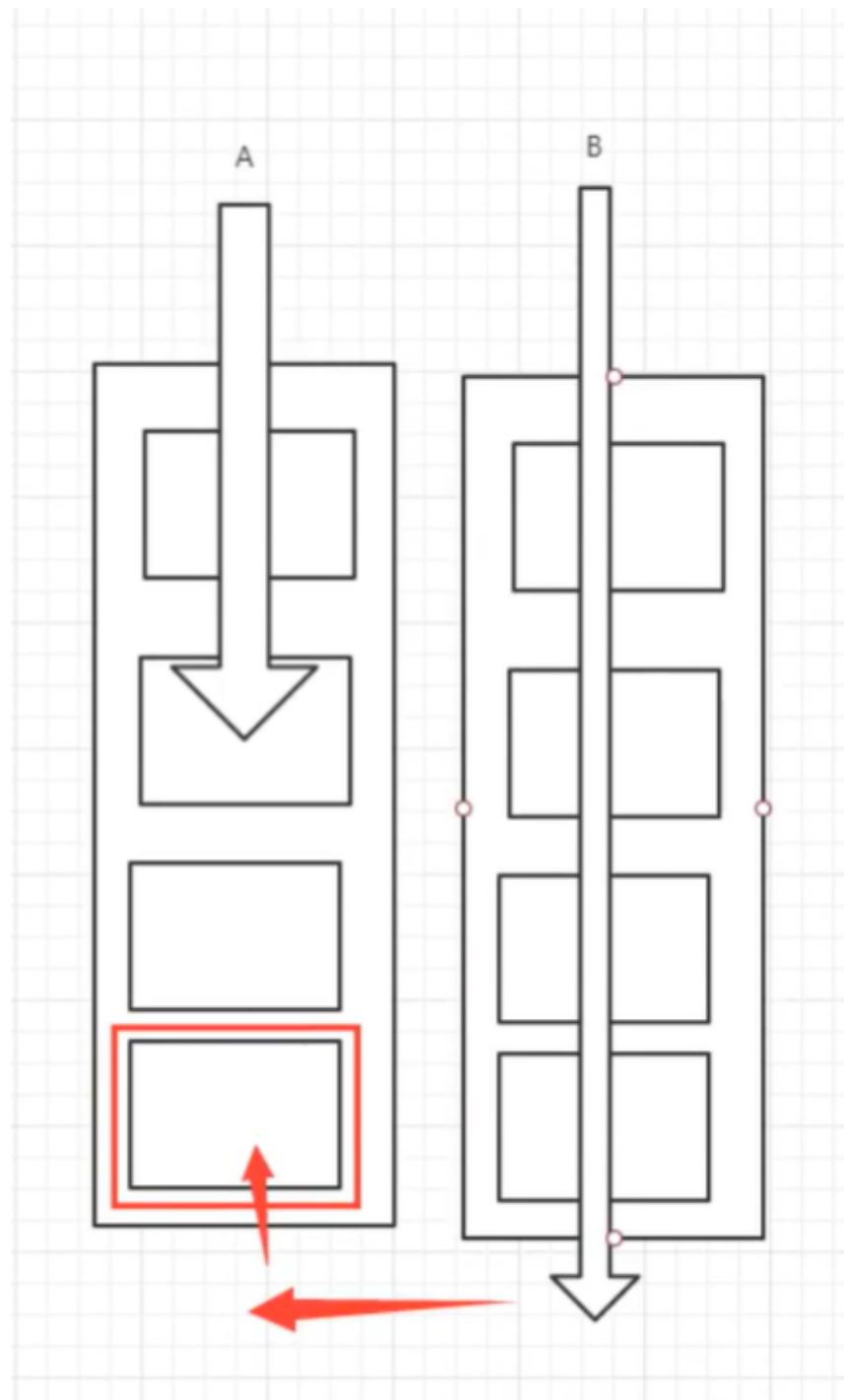
ForkJoin 在 JDK 1.7 , 并行执行任务! 提高效率。大数据量!

大数据: Map Reduce (把大任务拆分为小任务)



| ForkJoin 特点：工作窃取

这个里面维护的都是双端队列



Forkjoin

void

`execute(ForkJoinTask<?> task)`

为异步执行给定任务的排列。

compact1, compact2, compact3
java.util.concurrent

Class ForkJoinTask<V>

java.lang.Object

java.util.concurrent.ForkJoinTask<V>

递归事件

没有返回值

递归任务

有返回值的

All Implemented Interfaces:

Serializable , Future <V>

已知直接子类：

CountedCompleter , RecursiveAction , RecursiveTask

```
1 package com.godfrey.forkjoin;
2
3 import java.util.concurrent.RecursiveTask;
4
5 /**
6  * 求和计算的任务！
7  * 如何使用 forkjoin
8  * 1、forkjoinPool 通过它来执行
9  * 2、计算任务 forkjoinPool.execute(ForkJoinTask task)
10 * 3、计算类要继承 ForkJoinTask
11 *
12 * @author godfrey
13 * @since 2020-05-16
14 */
15 public class ForkJoinDemo extends RecursiveTask<Long> {
16     private Long start;
17     private Long end;
18
19     //临界值
20     private Long temp = 10000L;
21
22     public ForkJoinDemo(Long start, Long end) {
23         this.start = start;
24         this.end = end;
25     }
26
27
28     //计算方法
29     @Override
30     protected Long compute() {
31         if ((end - start) > temp) {
32             Long sum = 0L;
33             for (Long i = start; i < end; i++) {
34                 sum += i;
35             }
36             return sum;
37         } else { // forkjoin 递归
38             Long middle = (start + end) / 2;//中间值
39             ForkJoinDemo task1 = new ForkJoinDemo(start, middle);
40             task1.fork(); // 拆分任务，把任务压入线程队列
41         }
42     }
43 }
```

```

41         ForkJoinDemo task2 = new ForkJoinDemo(middle + 1, end);
42         task2.fork(); // 拆分任务，把任务压入线程队列
43         return task1.join() + task2.join();
44     }
45 }
46 }
```

测试：

```

1 package com.godfrey.forkjoin;
2
3 import java.util.concurrent.ExecutionException;
4 import java.util.concurrent.ForkJoinPool;
5 import java.util.concurrent.ForkJoinTask;
6 import java.util.stream.LongStream;
7
8 /**
9  * description : 效率测试
10 *
11 * @author godfrey
12 * @since 2020-05-16
13 */
14 public class Test {
15     public static void main(String[] args) throws ExecutionException,
16     InterruptedException {
17         //test1(); //sum=499999999500000000 时间: 7192
18         //test2(); //sum=499999999500000000 时间: 6949
19         test3(); //sum=500000000500000000 时间: 526
20     }
21
22     // 普通程序员
23     public static void test1() {
24         Long sum = 0L;
25         Long start = System.currentTimeMillis();
26         for (Long i = 0L; i < 10_0000_0000L; i++) {
27             sum += i;
28         }
29         Long end = System.currentTimeMillis();
30         System.out.println("sum=" + sum + " 时间: " + (end - start));
31     }
32
33     //ForkJoin
34     public static void test2() throws ExecutionException,
35     InterruptedException {
36         Long start = System.currentTimeMillis();
37
38         ForkJoinPool forkJoinPool = new ForkJoinPool();
39         ForkJoinTask<Long> task = new ForkJoinDemo(0L, 10_0000_0000L);
40         ForkJoinTask<Long> submit = forkJoinPool.submit(task);
41         Long sum = submit.get();
42
43         Long end = System.currentTimeMillis();
44         System.out.println("sum=" + sum + " 时间: " + (end - start));
45     }
46
47     //Stream并行流
48     public static void test3() {
49         Long start = System.currentTimeMillis();
50     }
```

```
48         Long sum = LongStream.rangeClosed(0L,
49                                         10_0000_0000L).parallel().reduce(0, Long::sum);
50         Long end = System.currentTimeMillis();
51         System.out.println("sum=" + sum + " 时间: " + (end - start));
52     }
53 }
```

13. 异步回调

Future 设计的初衷：对将来的某个事件的结果进行建模

Class CompletableFuture<T>

```
java.lang.Object
    java.util.concurrent.CompletableFuture<T>
```

All Implemented Interfaces:

CompletionStage <T>, Future <T>

```
1 package com.godfrey.future;
2
3 import java.util.concurrent.CompletableFuture;
4 import java.util.concurrent.ExecutionException;
5 import java.util.concurrent.TimeUnit;
6
7 /**
8 * description : 异步回调
9 *
10 * @author godfrey
11 * @since 2020-05-16
12 */
13 public class Demo01 {
14     public static void main(String[] args) throws ExecutionException,
15     InterruptedException {
16         //test1();
17         test02();
18     }
19
20     // 没有返回值的 runAsync 异步回调
21     public static void test1() throws InterruptedException,
22     ExecutionException {
23
24         CompletableFuture<Void> completableFuture =
25         CompletableFuture.runAsync(() -> {
26             try {
27                 TimeUnit.SECONDS.sleep(2);
28             } catch (InterruptedException e) {
29                 e.printStackTrace();
30             }
31         });
32     }
33 }
```

```

28         System.out.println(Thread.currentThread().getName() +
29             "runAsync=>void");
30     });
31     System.out.println("1111");
32     completableFuture.get(); // 获取阻塞执行结果
33 }
34
35 // 有返回值的 supplyAsync 异步回调
36 // ajax, 成功和失败的回调
37 // 返回的是错误信息:
38 public static void test02() throws InterruptedException,
39 ExecutionException {
40
41     CompletableFuture<Integer> completableFuture =
42         CompletableFuture.supplyAsync(() -> {
43             System.out.println(Thread.currentThread().getName() +
44                 "supplyAsync=>Integer");
45             int i = 10 / 0;
46             return 1024;
47         });
48     System.out.println(completableFuture.whenComplete((t, u) -> {
49         System.out.println("t=>" + t); // 正常的返回结果
50         System.out.println("u=>" + u); // 错误信息:
51         java.util.concurrent.CompletionException:java.lang.ArithmetricException: / by
52             zero
53
54     }).exceptionally((e) -> {
55         System.out.println(e.getMessage());
56         return 233; // 可以获取到错误的返回结果
57     }).get());
58     /**
59      * succee Code 200
60      * error Code 404 500
61     */
62 }

```

14.JMM

请你谈谈你对 Volatile 的理解

Volatile 是 Java 虚拟机提供轻量级的同步机制

1. 保证可见性
2. 不保证原子性
3. 禁止指令重排

什么是JMM

JMM : Java内存模型, 不存在的东西, 概念! 约定!

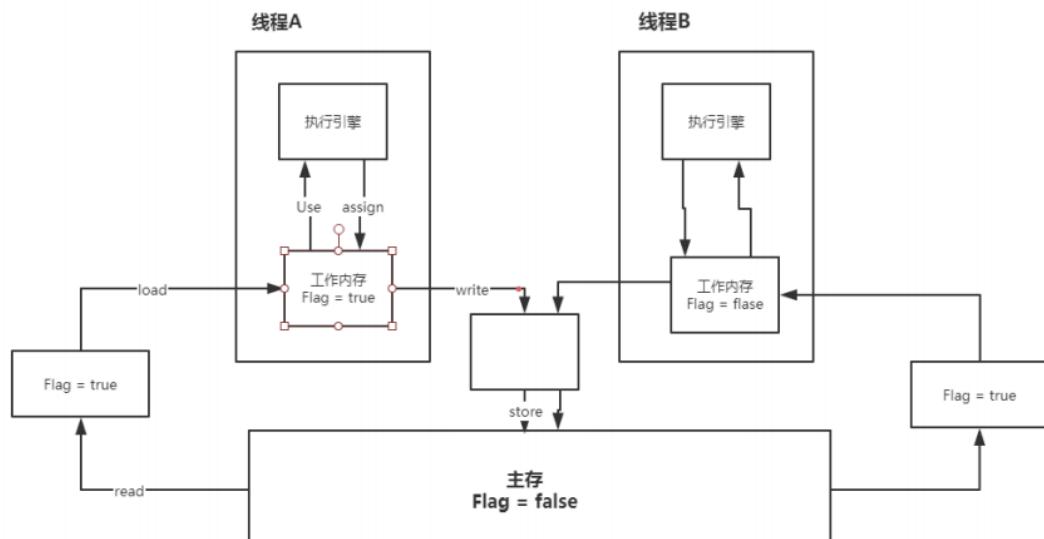
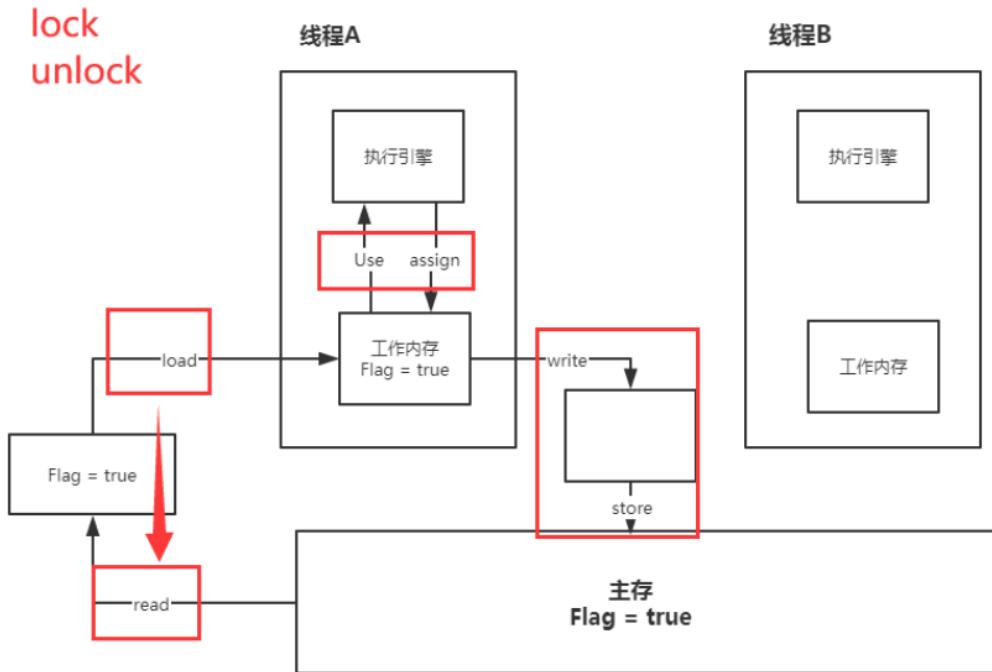
关于JMM的一些同步的约定:

1. 线程解锁前, 必须把共享变量立刻刷回主存

2. 线程加锁前，必须读取主存中的最新值到工作内存中！
3. 加锁和解锁是同一把锁

线程 工作内存、主内存

8种操作：



问题，线程B修改了值，但是线程A不能及时可见

内存交互操作有8种，虚拟机实现必须保证每一个操作都是原子的，不可再分的（对于double和long类型的变量来说，load、store、read和write操作在某些平台上允许例外）

- lock (锁定)：作用于主内存的变量，把一个变量标识为线程独占状态
- unlock (解锁)：作用于主内存的变量，它把一个处于锁定状态的变量释放出来，释放后的变量才可以被其他线程锁定

- **read** (读取) : 作用于主内存变量, 它把一个变量的值从主内存传输到线程的工作内存中, 以便随后的load动作使用
- **load** (载入) : 作用于工作内存的变量, 它把read操作从主存中变量放入工作内存中
- **use** (使用) : 作用于工作内存中的变量, 它把工作内存中的变量传输给执行引擎, 每当虚拟机遇到一个需要使用到变量的值, 就会使用到这个指令
- **assign** (赋值) : 作用于工作内存中的变量, 它把一个从执行引擎中接受到的值放入工作内存的变量副本中
- **store** (存储) : 作用于主内存中的变量, 它把一个从工作内存中一个变量的值传送到主内存中, 以便后续的write使用
- **write** (写入) : 作用于主内存中的变量, 它把store操作从工作内存中得到的变量的值放入主内存的变量中

JMM对这八种指令的使用, 制定了如下规则:

- 不允许read和load、 store和write操作之一单独出现。即使用了read必须load, 使用了store必须write
- 不允许线程丢弃他最近的assign操作, 即工作变量的数据改变了之后, 必须告知主存
- 不允许一个线程将没有assign的数据从工作内存同步回主内存
- 一个新的变量必须在主内存中诞生, 不允许工作内存直接使用一个未被初始化的变量。就是对变量实施use、 store操作之前, 必须经过assign和load操作
- 一个变量同一时间只有一个线程能对其进行lock。多次lock后, 必须执行相同次数的unlock才能解锁
- 如果对一个变量进行lock操作, 会清空所有工作内存中此变量的值, 在执行引擎使用这个变量前, 必须重新load或assign操作初始化变量的值
- 如果一个变量没有被lock, 就不能对其进行unlock操作。也不能unlock一个被其他线程锁住的变量
- 对一个变量进行unlock操作之前, 必须把此变量同步回主内存

15.Volatile

1.保证可见性

```

1 package com.godfrey.tvolatile;
2
3 import java.util.concurrent.TimeUnit;
4
5 /**
6  * description : volatile 保证可见性
7  *
8  * @author godfrey
9  * @since 2020-05-16
10 */
11 public class JMMDemo {
12     // 不加 volatile 程序就会死循环!
13     // 加 volatile 可以保证可见性
14     private volatile static int num = 0;
15
16     public static void main(String[] args) { // main
17         new Thread(() -> { // 线程 1 对主内存的变化不知道的
18             while (num == 0) {
19                 }
20             }).start();
21         try {

```

```
22     TimeUnit.SECONDS.sleep(1);
23 } catch (InterruptedException e) {
24     e.printStackTrace();
25 }
26 num = 1;
27 System.out.println(num);
28 }
29 }
```

2.不保证原子性

原子性：不可分割

线程A在执行任务的时候，不能被打扰的，也不能被分割。要么同时成功，要么同时失败

```
1 package com.godfrey.tvolatile;
2
3 /**
4 * description : volatile 不保证原子性
5 *
6 * @author godfrey
7 * @since 2020-05-16
8 */
9 public class JMMDemo {
10     private volatile static int num = 0;
11
12     public static void add() {
13         num++;
14     }
15
16     public static void main(String[] args) {
17         //理论上num结果应该为 2 万
18         for (int i = 0; i < 20; i++) {
19             new Thread(() -> {
20                 for (int j = 0; j < 1000; j++) {
21                     add();
22                 }
23             }).start();
24         }
25         while (Thread.activeCount() > 2) { // main gc
26             Thread.yield();
27         }
28         System.out.println(Thread.currentThread().getName() + " " + num);
29     }
30 }
```

如果不加 lock 和 synchronized，怎么样保证原子性

```

7  * @since 2020-05-16
8  */
9  public class VDemo02 {
10    private volatile static int num = 0;
11
12    public static void add() {
13      num++;    1.获得这个值
14      2.+1
15      3.写回这个值
16    }
17    //理论num结果应该为 2 方
18    public static void main(String[] args) {
19      for (int i = 0; i < 20; i++) {
20        new Thread(()->{
21          for (int j = 0; j < 1000; j++) {
22            add();
23          }
24        }).start();
25      }
26      while (Thread.activeCount() > 2) { // main_gc
27        Thread.yield();
28      }
29      System.out.println(Thread.currentThread()
30      .getName() + " " + num);
31    }
32  }

```

```

Run: javap -c VDemo02
Code:
  0: aload_0
  1: invokespecial #1
  4: return
// Method java/lang/Object/toString()

public static void add();
Code:
  0: getstatic   #2
  3: iconst_1
  4: iadd
  5: putstatic   #2
  8: return
// Field num:I

public static void main(java.lang.String[]);
Code:
  0: iconst_0
  1: istore_1
  2: iload_1
  3: bipush      20
  5: if_icmpge   29
  8: new         #3
 11: dup
 12: invokesynamic #4,  0
 17: invokespecial #5
 20: invokevirtual #6
// class java/lang/Object
// InvokeDynamic
// Method java/lang/Object/toString()
// Method java/lang/Object/toString()

```

使用原子类，解决原子性问题

java.util.concurrent
java.util.concurrent.atomic
 java.util.concurrent.locks
 java.util.function
 java.util.jar
 java.util.logging
 java.util.prefs
 java.util.regex

java.util.concurrent.atomic

Classes

- AtomicBoolean**
- AtomicInteger**
- AtomicIntegerArray
- AtomicIntegerFieldUpdater
- AtomicLong**
- AtomicLongArray
- AtomicLongFieldUpdater
- AtomicMarkableReference
- AtomicReference**
- AtomicReferenceArray
- AtomicReferenceFieldUpdater
- AtomicStampedReference
- DoubleAccumulator

```

1 package com.godfrey.tvolatile;
2
3 import java.util.concurrent.atomic.AtomicInteger;
4
5 /**
6  * description : volatile 不保证原子性
7  *
8  * @author godfrey
9  * @since 2020-05-16
10 */

```

```

11 public class VDemo02 {
12     // volatile 不保证原子性
13     // 原子类的 Integer
14     private volatile static AtomicInteger num = new AtomicInteger();
15
16     public static void add() {
17         // num++; // 不是一个原子性操作
18         num.getAndIncrement(); // AtomicInteger + 1 方法, CAS
19     }
20
21     public static void main(String[] args) {
22         //理论上num结果应该为 2 万
23         for (int i = 0; i < 20; i++) {
24             new Thread(() -> {
25                 for (int j = 0; j < 1000; j++) {
26                     add();
27                 }
28             }).start();
29         }
30         while (Thread.activeCount() > 2) { // main gc
31             Thread.yield();
32         }
33         System.out.println(Thread.currentThread().getName() + " " + num);
34     }
35 }
```

这些类的底层都直接和操作系统挂钩！在内存中修改值！Unsafe类是一个很特殊的存在！

指令重排

什么是 指令重排：你写的程序，计算机并不是按照你写的那样去执行的

源代码-->编译器优化的重排--> 指令并行也可能会重排--> 内存系统也会重排--> 执行

处理器在进行指令重排的时候，考虑：数据之间的依赖性！

```

1 int x = 1; // 1
2 int y = 2; // 2
3 x = x + 5; // 3
4 y = x * x; // 4
5 我们所期望的：1234 但是可能执行的时候会变成 2134 1324
6 可能是 4123!
```

可能造成影响的结果：a b x y 这四个值默认都是 0；

线程A	线程B
x=a	y=b
b=1	a=2

正常的结果：x = 0; y = 0; 但是可能由于指令重排

线程A	线程B
b=1	a=2
x=a	y=b

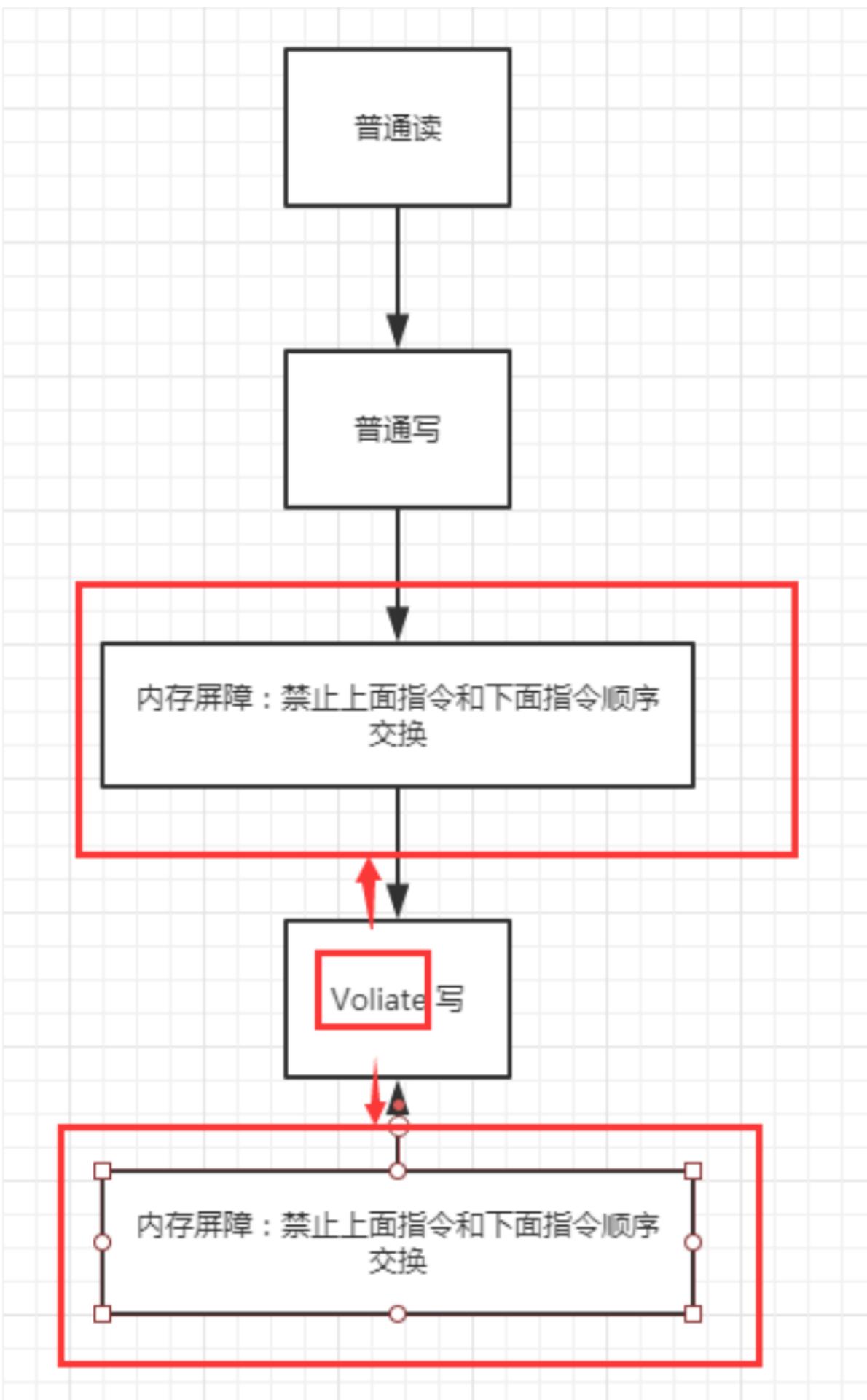
指令重排导致的诡异结果： $x = 2; y = 1;$

禁止指令重排

volatile可以禁止指令重排：

内存屏障。CPU指令。作用：

1. 保证特定的操作的执行顺序！
2. 可以保证某些变量的内存可见性（利用这些特性volatile实现了可见性）



Volatile 是可以保持 可见性。不能保证原子性，由于内存屏障，可以保证避免指令重排的现象产生！

16. 彻底玩转单例模式

饿汉式 DCL 懒汉式，深究！

饿汉式

```
1 package com.godfrey.single;
2
3 /**
4 * description : 饿汉式单例
5 *
6 * @author godfrey
7 * @since 2020-05-16
8 */
9 public class Hungry {
10     private Hungry() {
11
12     }
13
14     public final static Hungry HUNGRY = new Hungry();
15
16     public static Hungry getInstance() {
17         return HUNGRY;
18     }
19 }
```

静态内部类

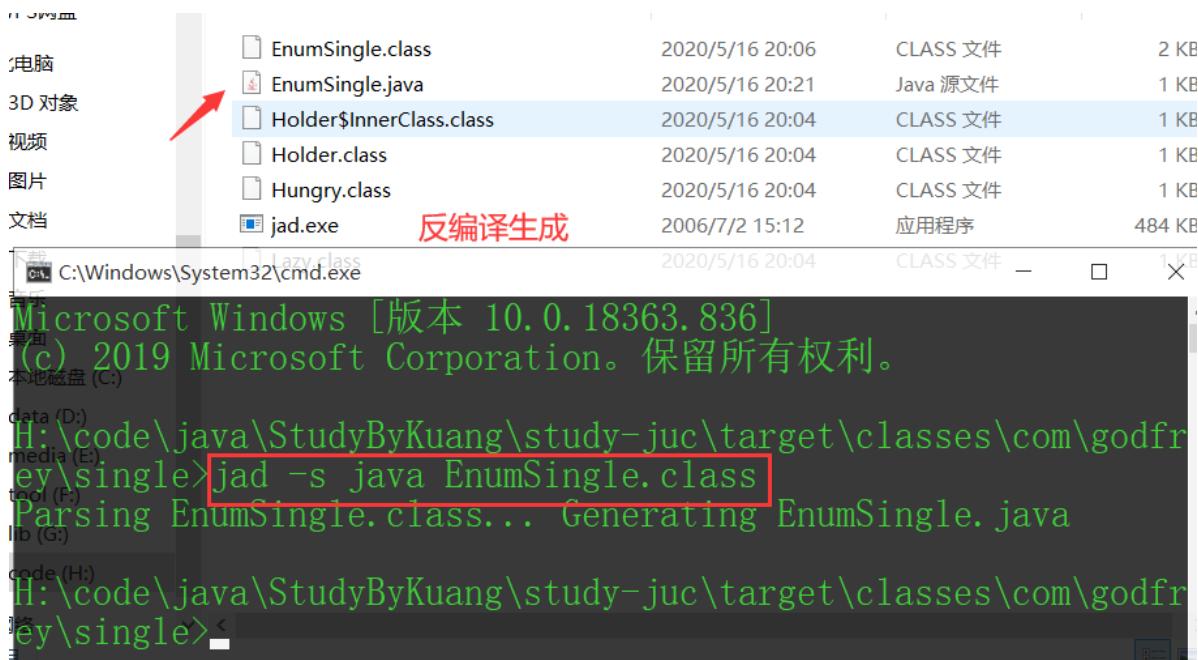
```
1 package com.godfrey.single;
2
3 /**
4 * description : 静态内部类单例
5 *
6 * @author godfrey
7 * @since 2020-05-16
8 */
9 public class Holder {
10     private Holder() {
11
12     }
13
14     private static class InnerClass {
15         private final static Holder HOLDER = new Holder();
16     }
17
18     public static Holder getInstance() {
19         return InnerClass.HOLDER;
20     }
21 }
```

单例不安全，反射

枚举

```
1 package com.godfrey.single;
2
3 /**
4 * description : 枚举单例
5 *
6 * @author godfrey
7 * @since 2020-05-16
8 */
9 public enum EnumSingle {
10     INSTANCE;
11 }
```

通过class文件反编译得到Java文件：



```
1 // Decompiled by Jad v1.5.8g. Copyright 2001 Pavel Kouznetsov.
2 // Jad home page: http://www.kpdus.com/jad.html
3 // Decompiler options: packimports(3)
4 // Source File Name:   EnumSingle.java
5
6 package com.godfrey.single;
7
8 import java.io.PrintStream;
9
10 public final class EnumSingle extends Enum
11 {
12
13     public static EnumSingle[] values()
14     {
15         return (EnumSingle[])$VALUES.clone();
16     }
17
18     public static EnumSingle valueof(String name)
```

```

19     {
20         return (EnumSingle)Enum.valueOf(com/godfrey/single/EnumSingle,
21             name);
22     }
23
24     private EnumSingle(String s, int i)
25     {
26         super(s, i);
27     }
28
29     public static void main(String args[])
30     {
31         System.out.println(INSTANCE);
32     }
33
34     public static final EnumSingle INSTANCE;
35     private static final EnumSingle $VALUES[];
36
37     static
38     {
39         INSTANCE = new EnumSingle("INSTANCE", 0);
40         $VALUES = (new EnumSingle[] {
41             INSTANCE
42         });
43     }

```

可以发现枚举的单例构造器是有参的

17.深入理解CAS

什么是 CAS

```

1 package com.godfrey.cas;
2
3 import java.util.concurrent.atomic.AtomicInteger;
4
5 /**
6  * description : CAS compareAndSet : 比较并交换
7  *
8  * @author godfrey
9  * @since 2020-05-16
10 */
11 public class CASDemo {
12     public static void main(String[] args) {
13         AtomicInteger atomicInteger = new AtomicInteger(2020);
14
15         // 期望、更新
16         // public final boolean compareAndSet(int expectedValue, int
17         // newValue)
18         // 如果我期望的值达到了，那么就更新，否则，就不更新，CAS 是CPU的并发原语！
19         System.out.println(atomicInteger.compareAndSet(2020, 2021));

```

```

20     System.out.println(atomicInteger.get());
21     atomicInteger.getAndIncrement();
22     System.out.println(atomicInteger.compareAndSet(2020, 2021));
23     System.out.println(atomicInteger.get());
24 }
25 }
```

Unsafe 类

```

/*
public class AtomicInteger extends Number implements java.io.Serializable {
    private static final long serialVersionUID = 6214790243416807050L;

    // setup to use Unsafe compareAndSwapInt for updates
    private static final Unsafe unsafe = Unsafe.getUnsafe();
    private static final long valueOffset;

    static {
        try {
            valueOffset = unsafe.objectFieldOffset(
                AtomicInteger.class.getDeclaredField( name: "value"));
        } catch (Exception ex) { throw new Error(ex); }
    }

    private volatile int value;
    */

    public native public final int getAndIncrement() {
        return unsafe.getAndAddInt( o: this, valueOffset, i: 1);
    }
}
```

Java 无法操作内存
Java 可以调用c++ native
c++ 可以操作内存
Java 的后门 . 可以通过这个类操

```

public native void putOrderedLong(Object v)
public native void unpark(Object var1);
public native void park(boolean var1, long var2);
public native int getLoadAverage(double var1, int var2);

```

```

public final int getAndAddInt(Object var1, long var2, int var4) {
    int var5; 获得内存地址中的值
    do {
        var5 = this.getIntVolatile(var1, var2);
    } while(!this.compareAndSwapInt(var1, var2, var5, var5: var5 + var4));

    return var5;
}

```

内存操作 , 效率很高

```

public final int getAndAddInt(Object var1, long var2, int var4) {
    int var5;
    do {
        var5 = this.getIntVolatile(var1, var2);
    } while(!this.compareAndSwapInt(var1, var2, var5, var5: var5 + var

```

```

public final int getAndAddInt(Object var1, long var2, int var4) {
    int var5;
    do {
        var5 = this.getIntVolatile(var1, var2);
    } while(!this.compareAndSwapInt(var1, var2, var5, var5: var5 + var4));

    return var5;
}

```

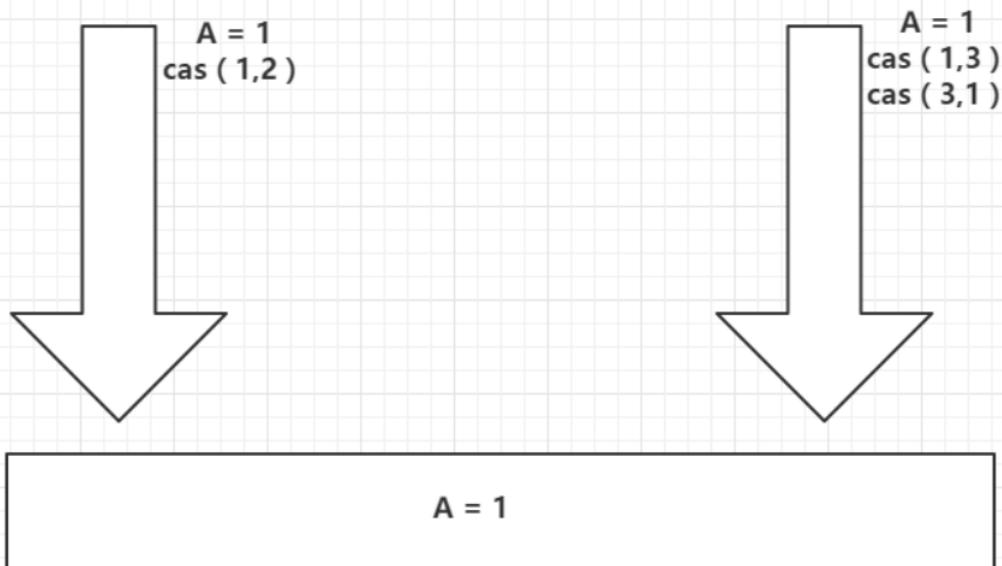
自旋锁

CAS：比较当前工作内存中的值和主内存中的值，如果这个值是期望的，那么则执行操作！如果不是就一直循环！

缺点：

1. 循环会耗时
2. 一次性只能保证一个共享变量的原子性
3. ABA问题

CAS：ABA问题（狸猫换太子）



```

1 package com.godfrey.cas;
2
3 import java.util.concurrent.atomic.AtomicInteger;
4
5 /**
6 * description : CAS问题: ABA (狸猫换太子)
7 *
8 * @author godfrey
9 * @since 2020-05-16
10 */
11 public class ABADemo {
12     // CAS compareAndSet : 比较并交换!
13     public static void main(String[] args) {
14         AtomicInteger atomicInteger = new AtomicInteger(2020);
15

```

```

16     // 期望、更新
17     // public final boolean compareAndSet(int expect, int update)
18     // 如果我期望的值达到了，那么就更新，否则，就不更新，CAS是CPU的并发原语！
19     // ===== 捣乱的线程 =====
20     System.out.println(atomicInteger.compareAndSet(2020, 2021));
21     System.out.println(atomicInteger.get());
22     System.out.println(atomicInteger.compareAndSet(2021, 2020));
23     System.out.println(atomicInteger.get());
24     // ===== 期望的线程 =====
25     System.out.println(atomicInteger.compareAndSet(2020, 6666));
26     System.out.println(atomicInteger.get());
27 }
28 }
```

18.原子引用

解决ABA问题，引入原子引用！对应的思想：乐观锁

带版本号的原子操作！

```

1 package com.godfrey.cas;
2
3 import java.util.concurrent.TimeUnit;
4 import java.util.concurrent.atomic.AtomicStampedReference;
5
6 /**
7 * description : 原子引用解决ABA问题
8 *
9 * @author godfrey
10 * @since 2020-05-16
11 */
12 public class AtomicStampedReferenceDemo {
13     //AtomicStampedReference 注意，如果泛型是一个包装类，注意对象的引用问题
14     static AtomicStampedReference<Integer> atomicStampedReference = new
AtomicStampedReference<>(1, 1);
15
16     public static void main(String[] args) {
17         new Thread(() -> {
18             int stamp = atomicStampedReference.getStamp(); // 获得版本号
19             System.out.println("a1=>" + stamp);
20
21             try {
22                 TimeUnit.SECONDS.sleep(1);
23             } catch (InterruptedException e) {
24                 e.printStackTrace();
25             }
26             atomicStampedReference.compareAndSet(1, 2,
atomicStampedReference.getStamp(), atomicStampedReference.getStamp() + 1);
27             System.out.println("a2=>" + atomicStampedReference.getStamp());
28
29             System.out.println(atomicStampedReference.compareAndSet(2, 1,
atomicStampedReference.getStamp(), atomicStampedReference.getStamp() + 1));
30             System.out.println("a3=>" + atomicStampedReference.getStamp());
31         }, "a").start();
32     }
33 }
```

```

32
33     // 乐观锁的原理相同!
34     new Thread(() -> {
35         int stamp = atomicStampedReference.getStamp(); // 获得版本号
36         System.out.println("b1=>" + stamp);
37
38         try {
39             TimeUnit.SECONDS.sleep(2);
40         } catch (InterruptedException e) {
41             e.printStackTrace();
42         }
43         System.out.println(atomicStampedReference.compareAndSet(1, 6,
44 stamp, stamp + 1));
45         System.out.println("b2=>" + atomicStampedReference.getStamp());
46     }, "b").start();
47 }

```

注意：

`Integer` 使用了对象缓存机制，默认范围是 -128 ~ 127，推荐使用静态工厂方法 `valueOf` 获取对象实例，而不是 `new`，因为 `valueOf` 使用缓存，而 `new` 一定会创建新的对象分配新的内存空间；

7. 【强制】所有整型包装类对象之间值的比较，全部使用 `equals` 方法比较。

说明：对于 `Integer var = ?` 在 -128 至 127 之间的赋值，`Integer` 对象是在 `IntegerCache.cache` 产生，会复用已有对象，这个区间内的 `Integer` 值可以直接使用 `==` 进行判断，但是这个区间之外的所有数据，都会在堆上产生，并不会复用已有对象，这是一个大坑，推荐使用 `equals` 方法进行判断。

19. 各种锁的理解

1. 公平锁、非公平锁

公平锁：非常公平，不能够插队，必须先来后到！

非公平锁：非常不公平，可以插队（默认都是非公平）

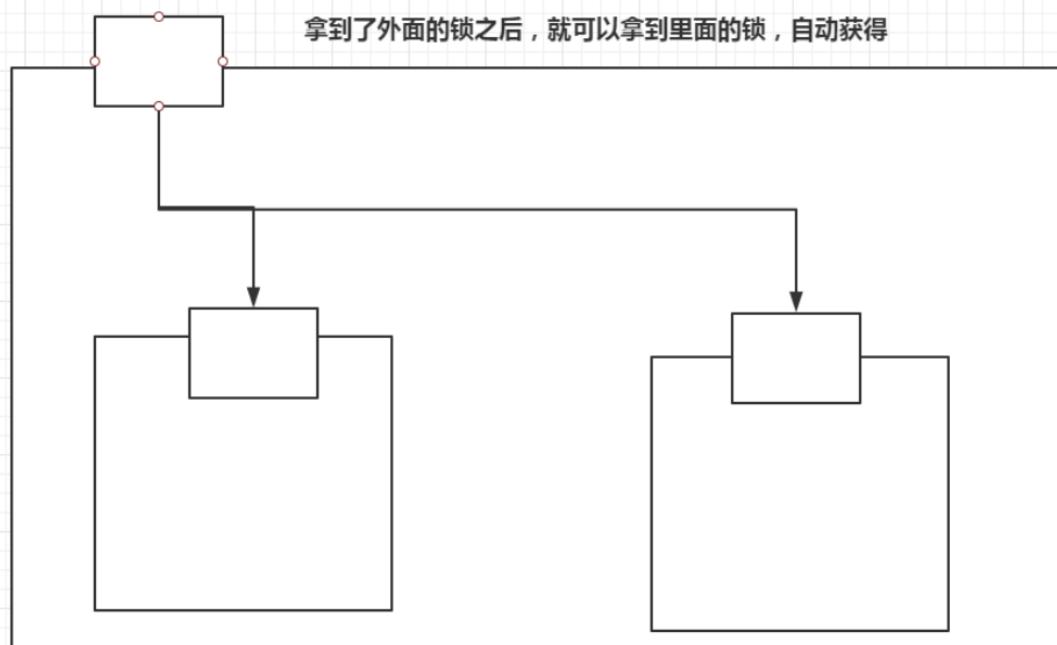
```

1 public ReentrantLock() {
2     sync = new NonfairSync();
3 }
4 public ReentrantLock(boolean fair) {
5     sync = fair ? new FairSync() : new NonfairSync();
6 }

```

2. 可重入锁

可重入锁（递归锁）



Synchronized

```

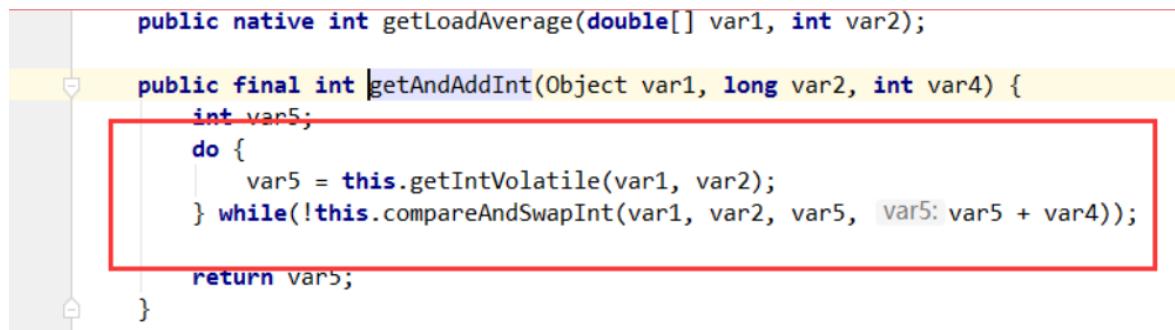
1 package com.godfrey.lock;
2
3 /**
4 * description : synchronized
5 *
6 * @author godfrey
7 * @since 2020-05-16
8 */
9 public class Demo01 {
10     public static void main(String[] args) {
11         Phone phone = new Phone();
12
13         new Thread(() -> {
14             phone.sms();
15         }, "A").start();
16
17         new Thread(() -> {
18             phone.sms();
19         }, "B").start();
20     }
21 }
22
23 class Phone {
24     public synchronized void sms() {
25         System.out.println(Thread.currentThread().getName() + "sms");
26         call(); // 这里也有锁
27     }
28
29     public synchronized void call() {
30         System.out.println(Thread.currentThread().getName() + "call");
31     }
32 }
```

Lock 版

```
1 package com.godfrey.lock;
2
3 import java.util.concurrent.locks.Lock;
4 import java.util.concurrent.locks.ReentrantLock;
5
6 /**
7 * description :
8 *
9 * @author godfrey
10 * @since 2020-05-16
11 */
12 public class Demo02 {
13     public static void main(String[] args) {
14         Phone2 phone = new Phone2();
15         new Thread(() -> {
16             phone.sms();
17         }, "A").start();
18
19         new Thread(() -> {
20             phone.sms();
21         }, "B").start();
22     }
23 }
24
25 class Phone2 {
26     Lock lock = new ReentrantLock();
27
28     public void sms() {
29         lock.lock();
30         try {
31             System.out.println(Thread.currentThread().getName() + "sms");
32             call(); // 这里也有锁
33         } catch (Exception e) {
34             e.printStackTrace();
35         } finally {
36             lock.unlock();
37         }
38     }
39
40     public void call() {
41         lock.lock();
42         try {
43             System.out.println(Thread.currentThread().getName() + "call");
44         } catch (Exception e) {
45             e.printStackTrace();
46         } finally {
47             lock.unlock();
48         }
49     }
50 }
```

3.自旋锁

spinlock



我们来自定义一个锁测试

```
1 package com.godfrey.lock;
2
3 import java.util.concurrent.atomic.AtomicReference;
4
5 /**
6 * description : 自旋锁
7 *
8 * @author godfrey
9 * @since 2020-05-16
10 */
11 public class SpinlockDemo {
12
13     AtomicReference<Thread> atomicReference = new AtomicReference<>();
14
15     // 加锁
16     public void myLock() {
17         Thread thread = Thread.currentThread();
18         System.out.println(Thread.currentThread().getName() + "=> mylock");
19
20         // 自旋锁
21         while (!atomicReference.compareAndSet(null, thread)) {
22             }
23     }
24
25     // 解锁
26     public void myUnLock() {
27         Thread thread = Thread.currentThread();
28         System.out.println(Thread.currentThread().getName() + "=>
myUnLock");
29         atomicReference.compareAndSet(thread, null);
30     }
31 }
```

测试

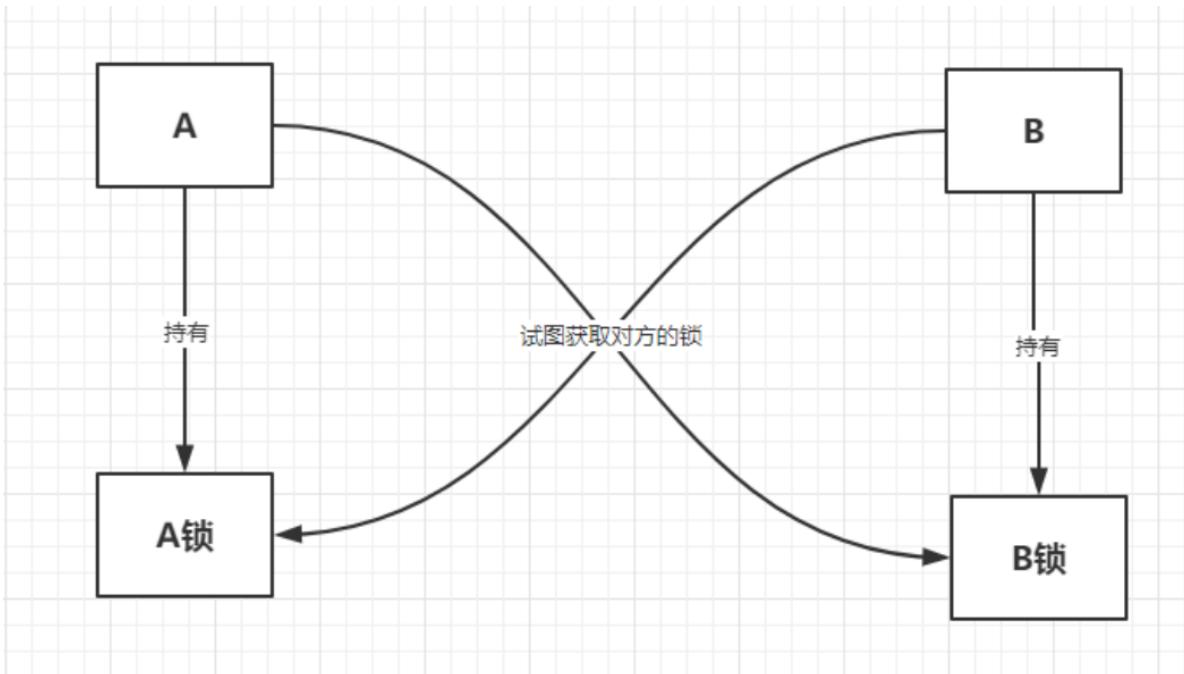
```
1 package com.godfrey.lock;
2
3 import java.util.concurrent.TimeUnit;
4
5 /**
6 * description : 测试自定义CAS实现的自旋锁
7 *
```

```
7  *
8  * @author godfrey
9  * @since 2020-05-16
10 */
11 public class TestSpinLock {
12     public static void main(String[] args) throws InterruptedException {
13         // 底层使用的自旋锁CAS
14         SpinlockDemo lock = new SpinlockDemo();
15
16         new Thread(() -> {
17             lock.myLock();
18             try {
19                 TimeUnit.SECONDS.sleep(5);
20             } catch (Exception e) {
21                 e.printStackTrace();
22             } finally {
23                 lock.myUnLock();
24             }
25         }, "T1").start();
26
27         TimeUnit.SECONDS.sleep(1);
28         new Thread(() -> {
29             lock.myLock();
30             try {
31                 TimeUnit.SECONDS.sleep(1);
32             } catch (Exception e) {
33                 e.printStackTrace();
34             } finally {
35                 lock.myUnLock();
36             }
37         }, "T2").start();
38     }
39 }
```

```
G:\lib\java-lib\Java\jdk-11\bin\java.exe "-javaagent"
T1==> mylock
T2==> mylock
T1==> myUnlock
T2==> myUnlock
```

4.死锁

死锁是什么



死锁测试，怎么排除死锁：

```

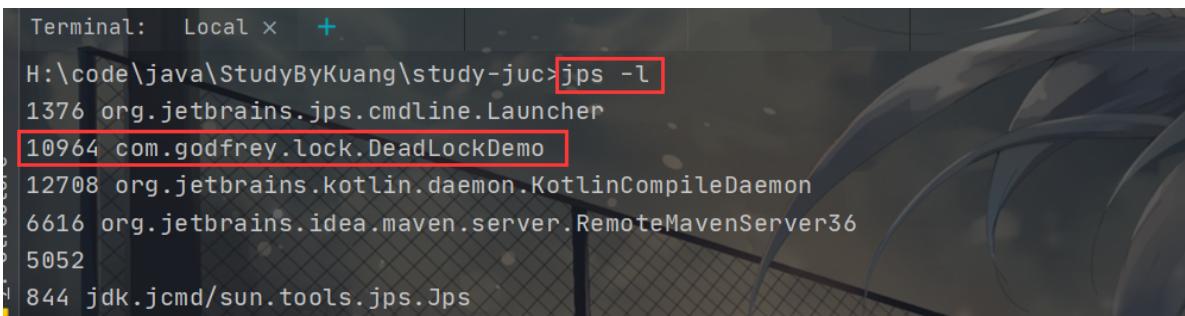
1 package com.godfrey.lock;
2
3 import java.util.concurrent.TimeUnit;
4
5 /**
6  * description : 死锁Demo
7  *
8  * @author godfrey
9  * @since 2020-05-16
10 */
11 public class DeadLockDemo {
12     public static void main(String[] args) {
13         String lockA = "lockA";
14         String lockB = "lockB";
15         new Thread(new MyThread(lockA, lockB), "T1").start();
16         new Thread(new MyThread(lockB, lockA), "T2").start();
17     }
18 }
19
20 class MyThread implements Runnable {
21     private String lockA;
22     private String lockB;
23
24     public MyThread(String lockA, String lockB) {
25         this.lockA = lockA;
26         this.lockB = lockB;
27     }
28
29     @Override
30     public void run() {
31         synchronized (lockA) {
32             System.out.println(Thread.currentThread().getName() +
33                     "lock:" + lockA + "=>get" + lockB);
34             try {
35                 TimeUnit.SECONDS.sleep(2);
36             } catch (InterruptedException e) {

```

```
37         e.printStackTrace();
38     }
39     synchronized (lockB) {
40         System.out.println(Thread.currentThread().getName() +
41             "lock:" + lockB + ">get" + lockA);
42     }
43 }
44 }
45 }
```

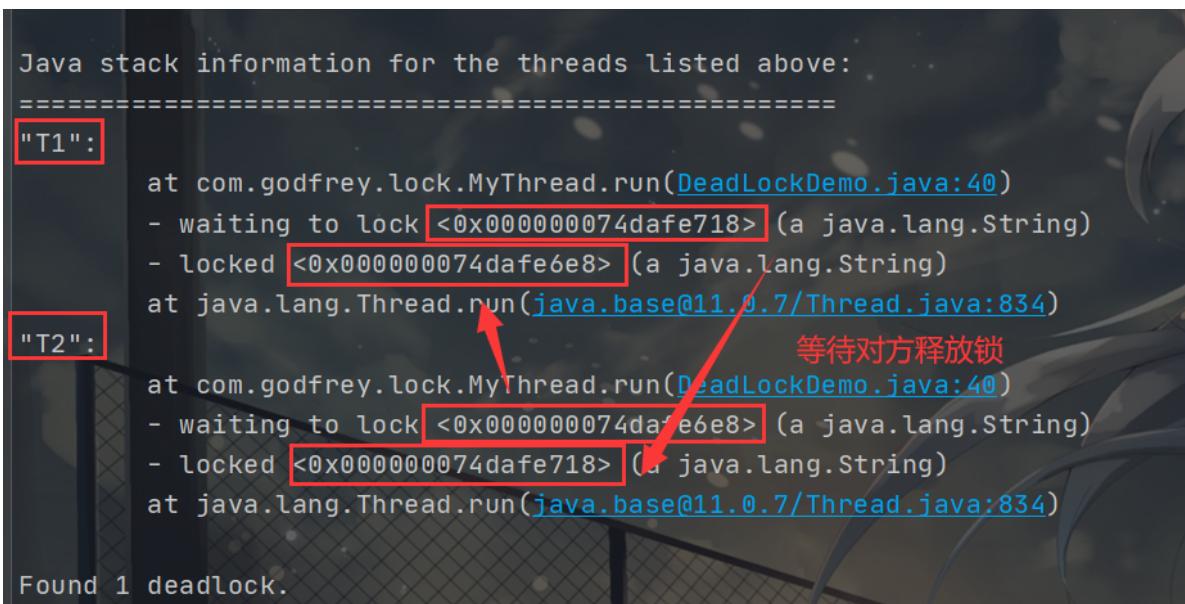
解决问题

1. 使用 `jps -l` 定位进程号



```
Terminal: Local x +
H:\code\java\StudyByKuang\study-juc>jps -l
1376 org.jetbrains.jps.cmdline.Launcher
10964 com.godfrey.lock.DeadLockDemo
12708 org.jetbrains.kotlin.daemon.KotlinCompileDaemon
6616 org.jetbrains.idea.maven.server.RemoteMavenServer36
5052
844 jdk.jcmd/sun.tools.jps.Jps
```

2. 使用 `jstack 进程号` 找到死锁问题



```
Java stack information for the threads listed above:
=====
[T1]:
at com.godfrey.lock.MyThread.run(DeadLockDemo.java:40)
- waiting to lock <0x000000074dafe718> (a java.lang.String)
- locked <0x000000074dafe6e8> (a java.lang.String)
at java.lang.Thread.run(java.base@11.0.7/Thread.java:834)

[T2]:
at com.godfrey.lock.MyThread.run(DeadLockDemo.java:40)
- waiting to lock <0x000000074dafe6e8> (a java.lang.String)
- locked <0x000000074dafe718> (a java.lang.String)
at java.lang.Thread.run(java.base@11.0.7/Thread.java:834)

Found 1 deadlock.
```