

ps: 这节课需要回顾一下操作系统的知识

## 多线程

### 多线程的创建和启动

★ 创建多线程可以通过 `Tread` 来创建, 而 `Tread` 方法需要传入一个 `Runnable` 的接口实现(使用内部类)

#### 1 `Tread` 的构建方法

```
1  /**
2   * Allocates a new {@code Thread} object. This constructor has the same
3   * effect as {@linkplain #Thread(ThreadGroup,Runnable,String) Thread}
4   * {@code (null, target, gname)}, where {@code gname} is a newly generated
5   * name. Automatically generated names are of the form
6   * {@code "Thread-"}+<i>n</i>, where <i>n</i> is an integer.
7   *
8   * @param target
9   *         the object whose {@code run} method is invoked when this thread
10   *         is started. If {@code null}, this classes {@code run} method does
11   *         nothing.
12   */
13 public Thread(Runnable target) {
14     init(null, target, "Thread-" + nextThreadNum(), 0);
15 }
```

#### 2 `Runnable` 在线程中是一个需要去实现的方法, 就是实现就是线程需要执行的操作

```
1  @FunctionalInterface
2  public interface Runnable {
3      /**
4       * When an object implementing interface <code>Runnable</code> is used
5       * to create a thread, starting the thread causes the object's
6       * <code>run</code> method to be called in that separately executing
7       * thread.
8       * <p>
9       * The general contract of the method <code>run</code> is that it may
10      * take any action whatsoever.
11      *
12      * @see      java.lang.Thread#run()
13      */
14      public abstract void run(); //通过实现run()方法来实现线程的操作
15 }
```

3 演示执行计算1-100的和，这只是创建一个线程，规定线程的作用，但是并没有开始运行线程

```
1 Thread thread = new Thread(new Runnable() { //这是一个匿名内部类
2     @Override
3     public void run() {
4         int sum = 0;
5         for (int i = 1; i <= 100; i++) {
6             sum += i;
7         }
8         System.out.println(sum);
9     }
10 });
11 }
12
13
14 //替换成lambda
15 public static void main(String[] args) {
16     Thread thread = new Thread(() -> {
17         int sum = 0;
18         for (int i = 1; i <= 100; i++) {
19             sum += i;
20         }
21         System.out.println(sum);
22     });
23 }
```

4 使用 start 方法开始执行 thread

```
1 public static void main(String[] args) {
2     Thread thread = new Thread(() -> {
3         int sum = 0;
4         for (int i = 1; i <= 100; i++) {
5             sum += i;
6         }
7         System.out.println(sum);
8     });
9     thread.start();
10 }
11
12 //输出：5050
```

5 主线程和子线程之间演示，主线程和子线程一起在计算，并不是按照顺序来执行的，互补干扰

```
1 public static void main(String[] args) {
2     Thread t = new Thread(() -> {
3         System.out.println("我是线程: "+Thread.currentThread().getName());
4         System.out.println("A我正在计算 0-10000 之间所有数的和...");
5         int sum = 0;
6         for (int i = 0; i <= 10000; i++) {
7             sum += i;
8         }
9     });
10 }
```

```

8         }
9         System.out.println("A结果: "+sum);
10    });
11    t.start();
12    System.out.println("我是主线程! ");
13    System.out.println("B我正在计算 0-10000 之间所有数的和...");
14    int sum = 0;
15    for (int i = 0; i <= 10000; i++) {
16        sum += i;
17    }
18    System.out.println("B结果: "+sum);
19 }
20
21
22 //输出:
23 我是主线程!
24 B我正在计算 0-10000 之间所有数的和...
25 我是线程: Thread-0
26 A我正在计算 0-10000 之间所有数的和...
27 B结果: 50005000
28 A结果: 50005000

```

#### 6 还有一个案例

```

1 public static void main(String[] args) {
2     Thread t1 = new Thread(() -> {
3         for (int i = 0; i < 50; i++) {
4             System.out.println("我是一号线程: "+i);
5         }
6     });
7     Thread t2 = new Thread(() -> {
8         for (int i = 0; i < 50; i++) {
9             System.out.println("我是二号线程: "+i);
10        }
11    });
12    t1.start();
13    t2.start();
14 }

```

★ 除了第一个构建方法外，还有一些构建方法，比如这个在传入一个参数给线程自定义名字的

```

1 /**
2  * Allocates a new {@code Thread} object. This constructor has the same
3  * effect as {@linkplain #Thread(ThreadGroup,Runnable,String) Thread}
4  * {@code (null, target, name)}.
5  *
6  * @param target
7  *        the object whose {@code run} method is invoked when this thread
8  *        is started. If {@code null}, this thread's run method is invoked.
9  *
10     * @param name

```

```

11     *         the name of the new thread
12     */
13     public Thread(Runnable target, String name) {
14         init(null, target, name, 0);
15     }

```

1 代码演示，在 `Runnable` 后面传入一个参数改名字

```

1     public static void main(String[] args) {
2         Thread t = new Thread() -> {
3             System.out.println("我是线程: "+Thread.currentThread().getName());
4             //Thread.currentThread后面会介绍
5             System.out.println("我正在计算 0-10000 之间所有数的和...");
6             int sum = 0;
7             for (int i = 0; i <= 10000; i++) {
8                 sum += i;
9             }
10            System.out.println("结果: "+sum);
11        }, "子线程A");
12        t.start();
13        System.out.println("我是主线程! ");
14    }
15    //输出:
16    我是主线程!
17    我是线程: 子线程A //输出名字改了
18    我正在计算 0-10000 之间所有数的和...
19    结果: 50005000

```

2 如果不写名字默认有方法自动改名字

```

1     /**
2      * Allocates a new {@code Thread} object. This constructor has the same
3      * effect as {@linkplain #Thread(ThreadGroup,Runnable,String) Thread}
4      * {@code (group, target, gname)} ,where {@code gname} is a newly generated
5      * name. Automatically generated names are of the form
6      * {@code "Thread-"}+<i>n</i>, where <i>n</i> is an integer.
7      *
8      * @param group
9      *         the thread group. If {@code null} and there is a security
10     *         manager, the group is determined by {@linkplain
11     *         SecurityManager#getThreadGroup SecurityManager.getThreadGroup()}.
12     *         If there is not a security manager or {@code
13     *         SecurityManager.getThreadGroup()} returns {@code null}, the group
14     *         is set to the current thread's thread group.
15     *
16     * @param target
17     *         the object whose {@code run} method is invoked when this thread
18     *         is started. If {@code null}, this thread's run method is invoked.
19     *
20     * @throws SecurityException

```

```

21         *           if the current thread cannot create a thread in the specified
22         *           thread group
23         */
24     public Thread(ThreadGroup group, Runnable target) {
25         init(group, target, "Thread-" + nextThreadNum(), 0);
26     }

```

★ 执行方法还有一个 `run`，`run` 它是只在当前线程去执行，并不是单开一个线程，

1 他是直接调用传入的 `Runnable`

```

1 //实现代码
2     @Override
3     public void run() {
4         if (target != null) {
5             target.run();
6         }
7     }

```

2 下列代码实例会发现是运行完t1在运行t2

```

1     public static void main(String[] args) {
2         Thread t1 = new Thread(() -> {
3             for (int i = 0; i < 50; i++) {
4                 System.out.println("我是一号线程: "+i);
5             }
6         });
7         Thread t2 = new Thread(() -> {
8             for (int i = 0; i < 50; i++) {
9                 System.out.println("我是二号线程: "+i);
10            }
11        });
12        t1.run();
13        t2.run();
14    }
15
16    //会发现是运行完t1在运行t2

```

★ 进程会有很多运行状态：等待，运行阻塞

1. **就绪态 (ready)**：一个进程已经具备运行条件，但由于某些事情从而不能运行的状态，当他调度给它占用CPU时，立即可以运行。一个进程获得除处理机之外的一切所需资源时，它就会位于“就绪队列中”
2. **执行态 (Running state)**：进程占有了包括CPU在内的全部资源，正在CPU上运行；再单机的环境下，每一时刻最多只有一个进程处于运行状态
3. **等待态**：也叫阻塞态，指因等待某种事件发生而暂停运行的状态；会位于等待队列中



#### 1 进程切换

##### 1. 运行 → 就绪:

1. 运行进程用完了时间片，不得不让出来（被动）
2. 运行进程被更高优先级的进程中断，所以当前进程被迫处于就绪状态

##### 3. 运行 → 阻塞: 进程用"系统调用"的方式申请某种操作系统资源，或者请求等待某个事件发生

##### 4. 阻塞 → 就绪: 当进程所等待的事件发生时，就会进入就绪队列，重新等待处理机的调度

★ 使用 `Thread.currentThread` 来获取当前进程对象，在进程内部

#### 1 获取 Main 线程对象

```
1 public static void main(String[] args) {
2     Thread t1 = new Thread(() -> {
3         for (int i = 0; i < 50; i++) {
4             System.out.println("我是一号线程: "+i);
5             //获取当前进程对象
6             Thread thread = Thread.currentThread();
7             System.out.println(thread.getName());
8         }
9     });
10    //获取main进程对象
11    Thread main = Thread.currentThread();
12    System.out.println(main.getName());
13 }
```

★ 可以用 `stop` 来终止进程，会发现`stop`，打了一个`@Deprecated`注解，这个要么是弃用，要么是要被移除

```
1  @Deprecated
2      public final void stop() {
3          SecurityManager security = System.getSecurityManager();
4          if (security != null) {
5              checkAccess();
6              if (this != Thread.currentThread()) {
7
8                  security.checkPermission(SecurityConstants.STOP_THREAD_PERMISSION);
9              }
10             }
11             // A zero status value corresponds to "NEW", it can't change to
12             // not-NEW because we hold the lock.
13             if (threadStatus != 0) {
14                 resume(); // Wake up thread if it was suspended; no-op otherwise
15             }
16
17             // The VM can handle all thread states
18             stop0(new ThreadDeath());
19     }
```

#### 1 关门`main`方法(自己搞着玩的)

```
1      public static void main(String[] args) {
2          //获取main进程对象
3          Thread main = Thread.currentThread();
4          main.stop();
5          System.out.println(main.getName());
6
7      }
```

#### 2 关闭线程

```
1      public static void main(String[] args) {
2          Thread t1 = new Thread(() -> {
3              for (int i = 0; i < 50; i++) {
4                  System.out.println("我是一号线程: "+i);
5                  //获取当前进程对象
6                  Thread thread = Thread.currentThread();
7                  if(i==50) thread.stop();
8              }
9          });
10         t1.start();
11     }
```

## 线程的休眠和中断



★ 我们前面提到，一个线程处于运行状态下，线程的下一个状态会出现以下情况：

- 当CPU给予的运行时间结束时，会从运行状态回到就绪（可运行）状态，等待下一次获得CPU资源。
- 当线程进入休眠 / 阻塞(如等待IO请求) / 手动调用 `wait()` 方法时，会使得线程处于等待状态，当等待状态结束后会回到就绪状态。
- 当线程出现异常或错误 / 被 `stop()` 方法强行停止 / 所有代码执行结束时，会使得线程的运行终止。

★ 可以使用 `sleep` 方法让程序进入阻塞状态

### 1 代码实例

```
1 public static void main(String[] args) throws InterruptedException {  
2     System.out.print("H");  
3     Thread.sleep(1000); //停止1秒  
4     System.out.print("e");  
5     Thread.sleep(1000);  
6     System.out.print("l");  
7     Thread.sleep(1000);  
8     System.out.print("l");  
9     Thread.sleep(1000);  
10    System.out.println("o");  
11 }
```



★ 在使用 `sleep` 的时候会抛出一个 `InterruptedException` 的异常，这个异常为中断异常，当支持中断操作的方法，都会抛出中断异常，这个异常可以在调用 `interrupt` 方法来终止支持中断异常的方法来发生，`interrupt` 它不会和 `stop` 一样直接强制终止，它会给指定线程添加一个中断标记以告知线程需要立即停止运行或是进行其他操作，由线程来响应此中断并进行相应的处理，也就是告诉你你要中断了，让线程先去处理中断前的操作

```
1 //操作
2 public static native void sleep(long millis) throws InterruptedException; //支持中
   断操作
```

ps: 为什么不推荐使用 `stop` 方法呢？因为它是将程序强制直接终止的，之前学过流的概念，在操作完成一个文件后，需要使用 `close` 关闭，你直接 `stop` 了就表示直接终止进程，各种后事都不会执行，就会导致这个文件一直被占用，是一件很危险的事情

```
1 public static void main(String[] args) {
2     Thread t = new Thread() -> {
3         try {
4             Thread.sleep(10000); //休眠10秒
5         } catch (InterruptedException e) {
6             e.printStackTrace();
7         }
8     };
9     t.start();
10    try {
11        Thread.sleep(3000); //休眠3秒，一定比线程t先醒来
12        t.interrupt(); //调用t的interrupt方法
13    } catch (InterruptedException e) {
14        e.printStackTrace();
15    }
16 }
```

这里会抛出一个中断异常

1 使用 `isInterrupted()`，方法来判断进程是否被打上了中断标记，然后在进行结束前的前置操作

```
1 public static void main(String[] args) {
2     Thread t = new Thread() -> {
3         System.out.println("线程开始运行！");
4         while (true){ //无限循环
5             if(Thread.currentThread().isInterrupted()){ //判断是否存在中断标
   志
6                 System.out.println("我被打了中断标记需要立即停止");
7                 break; //响应中断
8             }
9         }
10        System.out.println("线程被中断了！");
11    };
12    t.start();
13    try {
14        Thread.sleep(3000); //休眠3秒，一定比线程t先醒来
```

```

15         t.interrupt();    //调用t的interrupt方法
16     } catch (InterruptedException e) {
17         e.printStackTrace();
18     }
19 }

```

★ 如果只是想让进程在接受到一个中断标记之后，**通知进程**切换到进程内另外一个操作也可以使用 `interrupt` 方法判断，不过需要搭配 `interrupted` 来重置中断标记

```

1  import java.util.TreeMap;
2
3  public class Main {
4
5      public static void main(String[] args) {
6          Thread t = new Thread(() -> {
7              System.out.println("线程开始运行!");
8              while (true){    //无限循环
9                  if(Thread.currentThread().isInterrupted()){    //判断是否存在中断标
志
10                     System.out.println("我被打断了中断标记需要切换到下一个循环");
11                     break;    //响应中断
12                 }
13             }
14             Thread.interrupted(); //重置中断标记
15             while (true){    //无限循环
16                 if(Thread.currentThread().isInterrupted()){    //判断是否存在中断标
志
17                     System.out.println("我被打断了中断标记需要立即停止!!");
18                     break;    //响应中断
19                 }
20             }
21             System.out.println("线程被中断了!");
22         });
23         t.start();
24         try {
25             Thread.sleep(3000);    //休眠3秒，一定比线程t先醒来
26             t.interrupt();    //调用t的interrupt方法
27             Thread.sleep(3000);    //让t进程第一个循环先执行
28             t.interrupt();    //再次打上中断标记
29         } catch (InterruptedException e) {
30             e.printStackTrace();
31         }
32     }
33 }
34
35 //输出:
36 线程开始运行!
37 我被打断了中断标记需要切换到下一个循环
38 我被打断了中断标记需要立即停止!!
39 线程被中断了!
40

```

★ 让线程暂停 `suspend`，让线程恢复 `resume`

1 暂停进程，会发现进程会一直卡住

```
1 public static void main(String[] args) {
2     Thread t = new Thread() -> {
3         System.out.println("线程开始运行!");
4         Thread.currentThread().suspend(); //暂停此线程
5         System.out.println("线程被继续运行!");
6     };
7     t.start();
8     try {
9         Thread.sleep(3000); //休眠3秒，一定比线程t先醒来
10    } catch (InterruptedException e) {
11        e.printStackTrace();
12    }
13 }
```

2 只有当其他进程使用 `resum` 方法通知他可以继续运行它才可以运行

```
1 public static void main(String[] args) {
2     Thread t = new Thread() -> {
3         System.out.println("线程开始运行!");
4         Thread.currentThread().suspend(); //暂停此线程
5         System.out.println("线程被继续运行!");
6     };
7     t.start();
8     try {
9         Thread.sleep(3000); //休眠3秒，一定比线程t先醒来
10        t.resume();
11    } catch (InterruptedException e) {
12        e.printStackTrace();
13    }
14 }
```

## 线程的优先级

实际上，Java程序中的每个线程并不是平均分配CPU时间的，为了使得线程资源分配更加合理，Java采用的是抢占式调度方式，优先级越高的线程，优先使用CPU资源！我们希望CPU花费更多的时间去处理更重要的任务，而不太重要的任务，则可以先让出一部分资源。线程的优先级一般分为以下三种：

- `MIN_PRIORITY` 最低优先级
- `MAX_PRIORITY` 最高优先级
- `NOM_PRIORITY` 常规优先级

```

1 public static void main(String[] args) {
2     Thread t = new Thread() -> {
3         System.out.println("线程开始运行!");
4     };
5     t.start();
6     t.setPriority(Thread.MIN_PRIORITY); //通过使用setPriority方法来设定优先级
7 }

```

优先级越高的线程，获得CPU资源的概率会越大，并不是说一定优先级越高的线程越先执行！

## 线程的礼让和加入

★ 使用 `yield` 让位操作，将当前 CPU 资源让位给同等优先级的线程，并不是直接暂停当前让位的线程，而是尽可能的让其他线程拿到资源的时间比让位的进程多(我是这样理解的)

1 让位之后执行 `t2` 执行一段时间之后在回到 `t1`，然后继续满足条件继续让位，在让位之后，尽可能多的在执行线程2的内容

```

1 public static void main(String[] args) {
2     Thread t1 = new Thread() -> {
3         System.out.println("线程1开始运行!");
4         for (int i = 0; i < 50; i++) {
5             if(i % 5 == 0) {
6                 System.out.println("让位!");
7                 Thread.yield();
8             }
9             System.out.println("1打印: "+i);
10        }
11        System.out.println("线程1结束!");
12    };
13    Thread t2 = new Thread() -> {
14        System.out.println("线程2开始运行!");
15        for (int i = 0; i < 50; i++) {
16            System.out.println("2打印: "+i);
17        }
18    };
19    t1.start();
20    t2.start();
21 }

```

★ 当一个进程需要等待另外一个进程执行完毕后在操作可以使用 `join`，也就是将被等待进程加入到等待进程，线程的加入只是等待另一个线程的完成，并不是将另一个线程和当前线程合并！

```

1 public static void main(String[] args) {
2     Thread t1 = new Thread() -> {
3         System.out.println("线程1开始运行!");
4         for (int i = 0; i < 50; i++) {

```

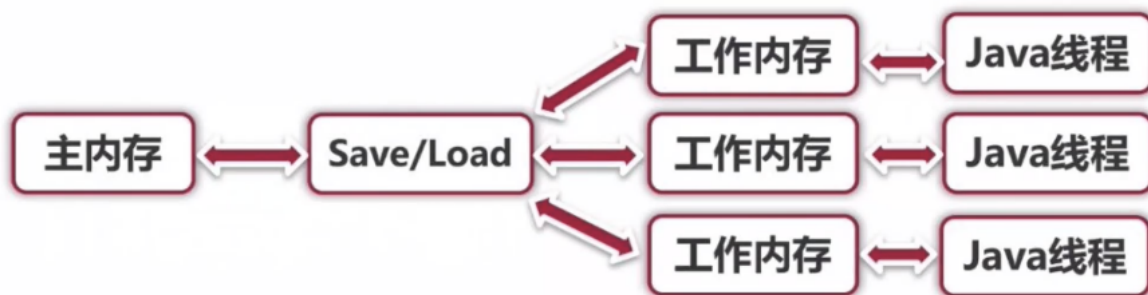
```

5         System.out.println("1打印: "+i);
6     }
7     System.out.println("线程1结束! ");
8 });
9     Thread t2 = new Thread(() -> {
10         System.out.println("线程2开始运行! ");
11         for (int i = 0; i < 50; i++) {
12             System.out.println("2打印: "+i);
13             if(i == 10){
14                 try {
15                     System.out.println("线程1加入到此线程! ");
16                     t1.join();    //在i==10时, 让线程1加入, 先完成线程1的内容, 在继续当
前内容
17                 } catch (InterruptedException e) {
18                     e.printStackTrace();
19                 }
20             }
21         }
22     });
23     t1.start();
24     t2.start();
25 }

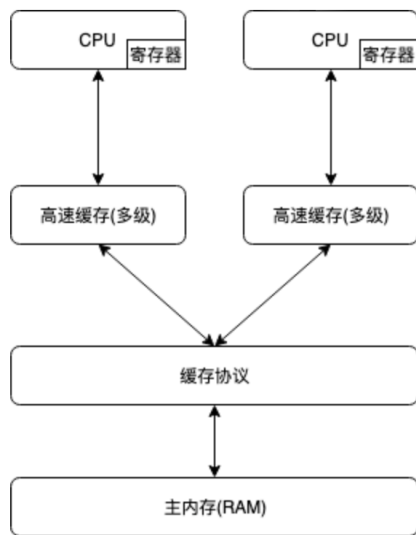
```

## 线程锁和线程同步

★ ? 如果让两个线程，都访问一个变量，并且线程内部都是将这个变量执行 10000 次，会发生什么操作？



线程之间的共享变量（比如之前悬念中的value变量）存储在主内存（main memory）中，每个线程都有一个私有的工作内存（本地内存），工作内存中存储了该线程以读/写共享变量的副本。它类似于我们在 [计算机组成原理](#) 中学习的多核心处理器高速缓存机制：



高速缓存通过保存内存中数据的副本来提供更加快速的数据访问，但是如果多个处理器的运算任务都涉及同一块内存区域，就可能导致各自的高速缓存数据不一致，在写回主内存时就会发生冲突，这就是引入高速缓存引发的新问题，称之为：缓存一致性。

实际上，Java的内存模型也是这样类似设计的，当我们同时去操作一个共享变量时，如果仅仅是读取还好，但是如果同时写入内容，就会出现问题！好比说一个银行，如果我和我的朋友同时在银行取我账户里面的钱，难道取1000还可能吐2000出来吗？我们需要一种更加安全的机制来维持秩序，保证数据的安全性！

**1** 会发现按照道理来说应该是20000，但是有些时候可能输出不到20000，当然有些情况如果电脑情况比较好，可能 `t1` 执行完成，`t2` 才执行，可能就会出现20000

这是因为每个线程在运行的时候会得到一个工作内存，而变量 `i` 是存储在主内存中的，它们在对 `i` 进行操作的时候，就会复制一个 `i` 到线程里面去执行线程里面的 `i`，执行完毕才会将线程里面的 `i` 的值返回给主内存中的 `i`，**这就会出现异步问题**

```

1      private static int i = 0;
2      public static void main(String[] args) throws InterruptedException {
3          Thread t1 = new Thread(() -> {
4              for (int j = 0; j < 10000; j++) {
5                  i++;
6              }
7          });
8          Thread t2 = new Thread(() -> {
9              for (int j = 0; j < 10000; j++) {
10                 i++;
11             }
12         });
13         t1.start();
14         t2.start();
15         Thread.sleep(2000); // 暂停两秒，让线程执行
16         System.out.println(i); // 输出 i
17     }

```

★ 解决办法是，在某个进程使用主内存变量或者全局变量的时候，就给这个变量加个锁，同一时间只能由一个线程访问和操作

**1** 同步代码块 `synchronized` ,这下无论你怎么执行都会是2000了,因为在同步代码块执行过程中,拿到了我们传入对象或类的锁(传入的如果是对象,就是对象锁,不同的对象代表不同的对象锁,如果是类,就是类锁,类锁只有一个,实际上类锁也是对象锁(入 `Main.class`),是Class类实例,但是Class类实例同样的类无论怎么获取都是同一个),但是注意两个线程必须使用同一把锁!

当一个线程进入到同步代码块时,会获取到当前的锁,而这时如果其他使用同样的锁的同步代码块也想执行内容,就必须等待当前同步代码块的内容执行完毕,在执行完毕后会自动释放这把锁,而其他的线程才能拿到这把锁并开始执行同步代码块里面的内容(实际上 `synchronized` 是一种悲观锁,随时都认为有其他线程在对数据进行修改;

★ 必须要是同一把锁才可以

```
1 private static int i = 0;
2 public static void main(String[] args) throws InterruptedException {
3     Object o = new Object(); //创建一个对象使用它的锁
4     Thread t1 = new Thread(() -> {
5         for (int j = 0; j < 10000; j++) {
6             synchronized (o) { //使用synchronized创建同步代码块
7                 i++;
8             }
9         }
10    });
11    Thread t2 = new Thread(() -> {
12        for (int j = 0; j < 10000; j++) {
13            synchronized (o) {
14                i++;
15            }
16        }
17    });
18    t1.start();
19    t2.start();
20    Thread.sleep(2000); //暂停两秒,让线程执行
21    System.out.println(i); //输出i
22 }
```

**2** 有个更加直观的例子

```
1 public static void main(String[] args) throws InterruptedException {
2     Object o = new Object(); //创建一个对象使用它的锁
3     Thread t1 = new Thread(() -> {
4         System.out.println("线程1开始执行");
5         synchronized (o) { //使用synchronized创建同步代码块
6             for (int j = 0; j < 5; j++) {
7                 try {
8                     Thread.sleep(1000);
9                 } catch (InterruptedException e) {
10                    throw new RuntimeException(e);
11                }
12                System.out.println("线程1运行中...");
13            }
14            System.out.println("线程1结束");
15        }
16    });
17    t1.start();
18 }
```

```

15         }
16     });
17     Thread t2 = new Thread(() -> {
18         System.out.println("线程2开始执行");
19         synchronized (o) { //使用synchronized创建同步代码块
20             for (int j = 0; j < 5; j++) {
21                 try {
22                     Thread.sleep(1000);
23                 } catch (InterruptedException e) {
24                     throw new RuntimeException(e);
25                 }
26                 System.out.println("线程2运行中...");
27             }
28             System.out.println("线程2结束");
29         }
30     });
31     t1.start();
32     t2.start();
33 }
34
35 //输出:
36 线程1开始执行
37 线程2开始执行
38 线程1运行中...
39 线程1运行中...
40 线程1运行中...
41 线程1运行中...
42 线程1运行中...
43 线程1结束
44 线程2运行中...//线程二等待线程一
45 线程2运行中...
46 线程2运行中...
47 线程2运行中...
48 线程2运行中...
49 线程2结束

```

### 3 如果不是同一把锁

```

1 public static void main(String[] args) throws InterruptedException {
2     Object o = new Object(); //创建一个对象使用它的锁
3     Thread t1 = new Thread(() -> {
4         System.out.println("线程1开始执行");
5         synchronized (o) { //使用synchronized创建同步代码块
6             for (int j = 0; j < 5; j++) {
7                 try {
8                     Thread.sleep(1000);
9                 } catch (InterruptedException e) {
10                    throw new RuntimeException(e);
11                }
12                System.out.println("线程1运行中...");
13            }

```



```

14         System.out.println("线程1结束");
15     }
16 });
17 Thread t2 = new Thread(() -> {
18     System.out.println("线程2开始执行");
19     synchronized (new Object()) { //不是同一把锁
20         for (int j = 0; j < 5; j++) {
21             try {
22                 Thread.sleep(1000);
23             } catch (InterruptedException e) {
24                 throw new RuntimeException(e);
25             }
26             System.out.println("线程2运行中...");
27         }
28     }
29     System.out.println("线程2结束");
30 }
31 t1.start();
32 t2.start();
33 }

```

★ `synchronized` 还可以锁方法，这个方法只能拿有一个线程调用

```

1     private static int value = 0;
2
3     private static synchronized void add(){ //直接使用的方法锁，静态方法
4         value++;
5     }
6
7     public static void main(String[] args) throws InterruptedException {
8         Thread t1 = new Thread(() -> {
9             for (int i = 0; i < 10000; i++) add();
10            System.out.println("线程1完成");
11        });
12        Thread t2 = new Thread(() -> {
13            for (int i = 0; i < 10000; i++) add();
14            System.out.println("线程2完成");
15        });
16        Thread t3 = new Thread(() -> {
17            for (int i = 0; i < 10000; i++)
18                synchronized(Main.class){ //因为这里是静态方法所以说是当前这个类的锁
19                    value++;
20                }
21            System.out.println("线程3完成");
22        });
23        t1.start();
24        t2.start();
25        t3.start();
26        Thread.sleep(1000); //主线程停止1秒，保证两个线程执行完成
27        System.out.println(value);
28    }

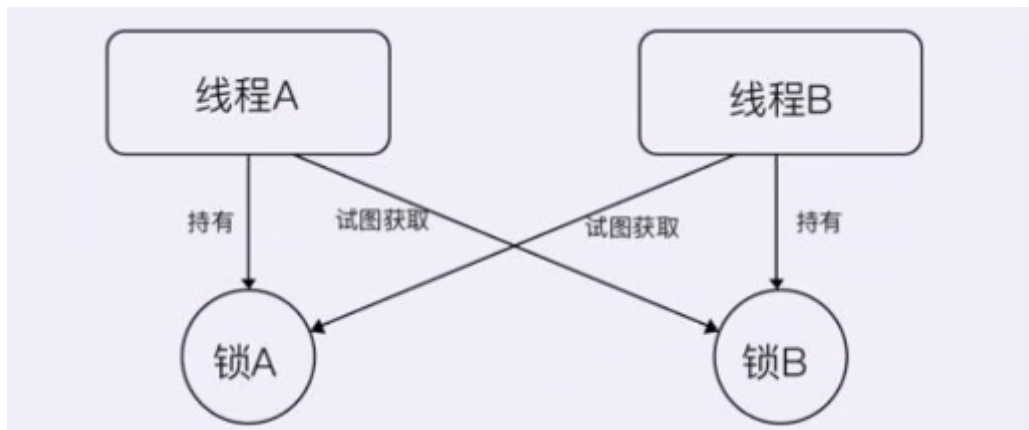
```

1 如果不是静态方法就需要创建对象，使用对象锁

```
1 private static int value = 0;
2
3 private synchronized void add(){ //直接使用的方法锁，成员方法
4     value++;
5 }
6
7 public static void main(String[] args) throws InterruptedException {
8     Main main = new Main();
9     Thread t1 = new Thread(() -> {
10         for (int i = 0; i < 10000; i++) main.add();
11         System.out.println("线程1完成");
12     });
13     Thread t2 = new Thread(() -> {
14         for (int i = 0; i < 10000; i++) main.add();
15         System.out.println("线程2完成");
16     });
17     Thread t3 = new Thread(() -> {
18         for (int i = 0; i < 10000; i++)
19             synchronized(main){ //这个是成员方法使用对象锁
20                 value++;
21             }
22         System.out.println("线程3完成");
23     });
24     t1.start();
25     t2.start();
26     t3.start();
27     Thread.sleep(1000); //主线程停止1秒，保证两个线程执行完成
28     System.out.println(value);
29 }
```

## 死锁

★其实死锁的概念在操作系统中也有提及，它是指两个线程相互持有对方需要的锁，但是又迟迟不释放，导致程序卡住：



1 程序A那拿到了锁A，程序B拿到了锁B，程序A执行一段时间后，有段代码需要拿到锁B才可以继续执行，而程序B也是如此，需要拿到锁A才可以继续运行，它们两互相僵持不下，本来可以退一步海阔天空，但是它们一直僵持

程序演示

```
1 public static void main(String[] args) throws InterruptedException {
2     Object o1 = new Object(); //锁1
3     Object o2 = new Object(); //锁2
4     Thread t1 = new Thread(() -> {
5         synchronized (o1){ //先拿到锁一
6             try {
7                 Thread.sleep(1000);
8                 synchronized (o2){ //需要锁二才可以执行，而锁2需要等待t2运行完成才可以释
放
9                     System.out.println("线程1");
10                }
11            } catch (InterruptedException e) {
12                e.printStackTrace();
13            }
14        }
15    });
16    Thread t2 = new Thread(() -> {
17        synchronized (o2){ //先拿到锁2
18            try {
19                Thread.sleep(1000);
20                synchronized (o1){ //需要锁1才可以继续执行，而锁1需要等待t1运行完成才可
以释放
21                    System.out.println("线程2");
22                }
23            } catch (InterruptedException e) {
24                e.printStackTrace();
25            }
26        }
27    });
28    t1.start();
29    t2.start();
30 }
```

★ 通过 jstack 来检查java程序中是否有死锁， jps 会显示 java 进程的进程号

1 先运行上面的程序使用 jps 查看进程号

```
1 31936
2 21444 Main
3 26812 Launcher
4 9692 Jps
```

2 使用 jstack 查看

```
1 # jstack 21444
```

```
2
3 2024-04-25 21:06:36
4 Full thread dump Java HotSpot(TM) 64-Bit Server VM (25.131-b11 mixed mode):
5
6 "DestroyJavaVM" #22 prio=5 os_prio=0 tid=0x000000003003800 nid=0x704c waiting
on condition [0x0000000000000000]
7     java.lang.Thread.State: RUNNABLE
8
9 "Thread-1" #21 prio=5 os_prio=0 tid=0x0000000021312000 nid=0x5ff8 waiting for
monitor entry [0x0000000021b8f000]
10     java.lang.Thread.State: BLOCKED (on object monitor)
11         at Main.lambda$main$1(Main.java:24)
12         - waiting to lock <0x0000000076e3a1f78> (a java.lang.Object)
13         - locked <0x0000000076e3a1f88> (a java.lang.Object)
14         at Main$$Lambda$2/1324119927.run(Unknown Source)
15         at java.lang.Thread.run(Thread.java:748)
16
17 "Thread-0" #20 prio=5 os_prio=0 tid=0x000000002130e800 nid=0x6110 waiting for
monitor entry [0x0000000021a8f000]
18     java.lang.Thread.State: BLOCKED (on object monitor)
19         at Main.lambda$main$0(Main.java:12)
20         - waiting to lock <0x0000000076e3a1f88> (a java.lang.Object)
21         - locked <0x0000000076e3a1f78> (a java.lang.Object)
22         at Main$$Lambda$1/295530567.run(Unknown Source)
23         at java.lang.Thread.run(Thread.java:748)
24
25 "Service Thread" #19 daemon prio=9 os_prio=0 tid=0x000000001e785000 nid=0x7fa0
runnable [0x0000000000000000]
26     java.lang.Thread.State: RUNNABLE
27
28 "C1 CompilerThread11" #18 daemon prio=9 os_prio=2 tid=0x000000001e6b6000
nid=0x6818 waiting on condition [0x0000000000000000]
29     java.lang.Thread.State: RUNNABLE
30
31 "C1 CompilerThread10" #17 daemon prio=9 os_prio=2 tid=0x000000001e6b7800
nid=0x5c08 waiting on condition [0x0000000000000000]
32     java.lang.Thread.State: RUNNABLE
33
34 "C1 CompilerThread9" #16 daemon prio=9 os_prio=2 tid=0x000000001e6b5800
nid=0x71d8 waiting on condition [0x0000000000000000]
35     java.lang.Thread.State: RUNNABLE
36
37 "C1 CompilerThread8" #15 daemon prio=9 os_prio=2 tid=0x000000001e6b4000
nid=0x179c waiting on condition [0x0000000000000000]
38     java.lang.Thread.State: RUNNABLE
39
40 "C2 CompilerThread7" #14 daemon prio=9 os_prio=2 tid=0x000000001e6b4800
nid=0x4070 waiting on condition [0x0000000000000000]
41     java.lang.Thread.State: RUNNABLE
42
43 "C2 CompilerThread6" #13 daemon prio=9 os_prio=2 tid=0x000000001e6a8800
nid=0x7804 waiting on condition [0x0000000000000000]
```

```
44     java.lang.Thread.State: RUNNABLE
45
46 "C2 CompilerThread5" #12 daemon prio=9 os_prio=2 tid=0x000000001e69e000
nid=0x77d8 waiting on condition [0x0000000000000000]
47     java.lang.Thread.State: RUNNABLE
48
49 "C2 CompilerThread4" #11 daemon prio=9 os_prio=2 tid=0x000000001e696000
nid=0x7d48 waiting on condition [0x0000000000000000]
50     java.lang.Thread.State: RUNNABLE
51
52 "C2 CompilerThread3" #10 daemon prio=9 os_prio=2 tid=0x000000001e695800
nid=0x39b8 waiting on condition [0x0000000000000000]
53     java.lang.Thread.State: RUNNABLE
54
55 "C2 CompilerThread2" #9 daemon prio=9 os_prio=2 tid=0x000000001e694800
nid=0x6a00 waiting on condition [0x0000000000000000]
56     java.lang.Thread.State: RUNNABLE
57
58 "C2 CompilerThread1" #8 daemon prio=9 os_prio=2 tid=0x000000001e694000
nid=0x5378 waiting on condition [0x0000000000000000]
59     java.lang.Thread.State: RUNNABLE
60
61 "C2 CompilerThread0" #7 daemon prio=9 os_prio=2 tid=0x000000001e682800
nid=0x50d4 waiting on condition [0x0000000000000000]
62     java.lang.Thread.State: RUNNABLE
63
64 "Monitor Ctrl-Break" #6 daemon prio=5 os_prio=0 tid=0x000000001e67f800
nid=0x5f8c runnable [0x000000002028e000]
65     java.lang.Thread.State: RUNNABLE
66         at java.net.SocketInputStream.socketRead0(Native Method)
67         at java.net.SocketInputStream.socketRead(SocketInputStream.java:116)
68         at java.net.SocketInputStream.read(SocketInputStream.java:171)
69         at java.net.SocketInputStream.read(SocketInputStream.java:141)
70         at sun.nio.cs.StreamDecoder.readBytes(StreamDecoder.java:284)
71         at sun.nio.cs.StreamDecoder.implRead(StreamDecoder.java:326)
72         at sun.nio.cs.StreamDecoder.read(StreamDecoder.java:178)
73         - locked <0x000000076e27d6d0> (a java.io.InputStreamReader)
74         at java.io.InputStreamReader.read(InputStreamReader.java:184)
75         at java.io.BufferedReader.fill(BufferedReader.java:161)
76         at java.io.BufferedReader.readLine(BufferedReader.java:324)
77         - locked <0x000000076e27d6d0> (a java.io.InputStreamReader)
78         at java.io.BufferedReader.readLine(BufferedReader.java:389)
79         at
com.intellij.rt.execution.application.AppMainV2$1.run(AppMainV2.java:53)
80
81 "Attach Listener" #5 daemon prio=5 os_prio=2 tid=0x000000001e5eb000 nid=0x7c6c
waiting on condition [0x0000000000000000]
82     java.lang.Thread.State: RUNNABLE
83
84 "Signal Dispatcher" #4 daemon prio=9 os_prio=2 tid=0x000000001e640800
nid=0x49ac runnable [0x0000000000000000]
85     java.lang.Thread.State: RUNNABLE
```

```
86
87 "Finalizer" #3 daemon prio=8 os_prio=1 tid=0x000000001e5c6800 nid=0x7c28 in
Object.wait() [0x000000001ff2e000]
88   java.lang.Thread.State: WAITING (on object monitor)
89     at java.lang.Object.wait(Native Method)
90     - waiting on <0x0000000076e108ec8> (a
java.lang.ref.ReferenceQueue$Lock)
91     at java.lang.ref.ReferenceQueue.remove(ReferenceQueue.java:143)
92     - locked <0x0000000076e108ec8> (a java.lang.ref.ReferenceQueue$Lock)
93     at java.lang.ref.ReferenceQueue.remove(ReferenceQueue.java:164)
94     at java.lang.ref.Finalizer$FinalizerThread.run(Finalizer.java:209)
95
96 "Reference Handler" #2 daemon prio=10 os_prio=2 tid=0x000000001cb6f800
nid=0xc10 in Object.wait() [0x000000001fe2f000]
97   java.lang.Thread.State: WAITING (on object monitor)
98     at java.lang.Object.wait(Native Method)
99     - waiting on <0x0000000076e106b68> (a java.lang.ref.Reference$Lock)
100    at java.lang.Object.wait(Object.java:502)
101    at java.lang.ref.Reference.tryHandlePending(Reference.java:191)
102    - locked <0x0000000076e106b68> (a java.lang.ref.Reference$Lock)
103    at java.lang.ref.Reference$ReferenceHandler.run(Reference.java:153)
104
105 "VM Thread" os_prio=2 tid=0x000000001e5a4800 nid=0x7130 runnable
106
107 "GC task thread#0 (ParallelGC)" os_prio=0 tid=0x0000000003018000 nid=0x6ba0
runnable
108
109 "GC task thread#1 (ParallelGC)" os_prio=0 tid=0x000000000301a000 nid=0x1f28
runnable
110
111 "GC task thread#2 (ParallelGC)" os_prio=0 tid=0x000000000301b800 nid=0x7738
runnable
112
113 "GC task thread#3 (ParallelGC)" os_prio=0 tid=0x000000000301e000 nid=0x7184
runnable
114
115 "GC task thread#4 (ParallelGC)" os_prio=0 tid=0x0000000003020000 nid=0x686c
runnable
116
117 "GC task thread#5 (ParallelGC)" os_prio=0 tid=0x0000000003021000 nid=0x6d1c
runnable
118
119 "GC task thread#6 (ParallelGC)" os_prio=0 tid=0x0000000003024800 nid=0x6880
runnable
120
121 "GC task thread#7 (ParallelGC)" os_prio=0 tid=0x0000000003025800 nid=0x640
runnable
122
123 "GC task thread#8 (ParallelGC)" os_prio=0 tid=0x0000000003026800 nid=0x7808
runnable
124
```


```

125 "GC task thread#9 (ParallelGC)" os_prio=0 tid=0x000000003028000 nid=0x615c
runnable
126
127 "GC task thread#10 (ParallelGC)" os_prio=0 tid=0x000000003029000 nid=0x70bc
runnable
128
129 "GC task thread#11 (ParallelGC)" os_prio=0 tid=0x00000000302c000 nid=0x3178
runnable
130
131 "GC task thread#12 (ParallelGC)" os_prio=0 tid=0x00000000302d800 nid=0x7e98
runnable
132
133 "VM Periodic Task Thread" os_prio=2 tid=0x000000001e7ab800 nid=0x7e9c waiting
on condition
134
135 JNI global references: 318
136
137 //发现了一个死锁
138 Found one Java-level deadlock:
139 =====
140 "Thread-1":
141   waiting to lock monitor 0x000000001cb761a8 (object 0x0000000076e3a1f78, a
java.lang.Object),
142   which is held by "Thread-0"
143 "Thread-0":
144   waiting to lock monitor 0x000000001cb73918 (object 0x0000000076e3a1f88, a
java.lang.Object),
145   which is held by "Thread-1"
146
147 //死锁发生位置
148 Java stack information for the threads listed above:
149 =====
150 "Thread-1":
151   at Main.lambda$main$1(Main.java:24)
152   - waiting to lock <0x0000000076e3a1f78> (a java.lang.Object)
153   - locked <0x0000000076e3a1f88> (a java.lang.Object)
154   at Main$$Lambda$2/1324119927.run(Unknown Source)
155   at java.lang.Thread.run(Thread.java:748)
156 "Thread-0":
157   at Main.lambda$main$0(Main.java:12)
158   - waiting to lock <0x0000000076e3a1f88> (a java.lang.Object)
159   - locked <0x0000000076e3a1f78> (a java.lang.Object)
160   at Main$$Lambda$1/295530567.run(Unknown Source)
161   at java.lang.Thread.run(Thread.java:748)
162
163 Found 1 deadlock.
164
165

```

3 还可以使用 jconsole 来查看，它是一个图形化界面

JConsole: 新建连接

 新建连接

☒ 本地进程 (L) :

名称	PID
	31936
Main	21444
jdk.jconsole/sun.tools.jconsole.JConsole	27336
org.jetbrains.jps.cmdline.Launcher D:/soft...	26812

注: 将对此进程启用管理代理。

☐ 远程进程 (R) :

用法: <hostname>:<port> 或 service:jmx:<protocol>:<sap>

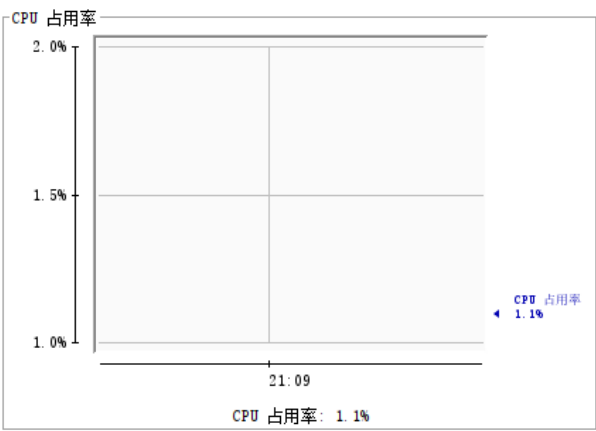
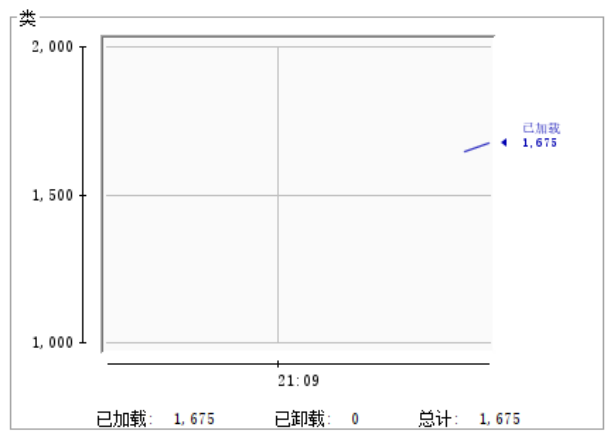
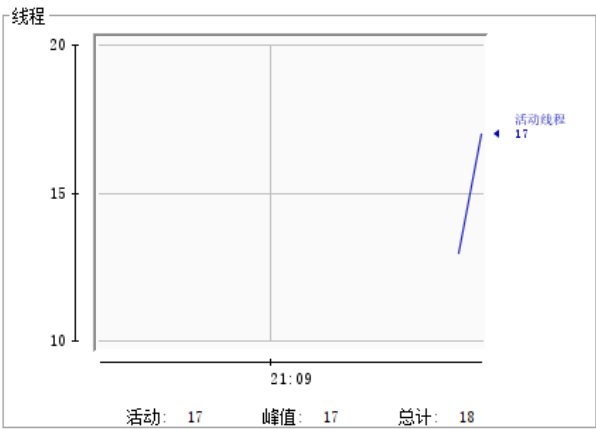
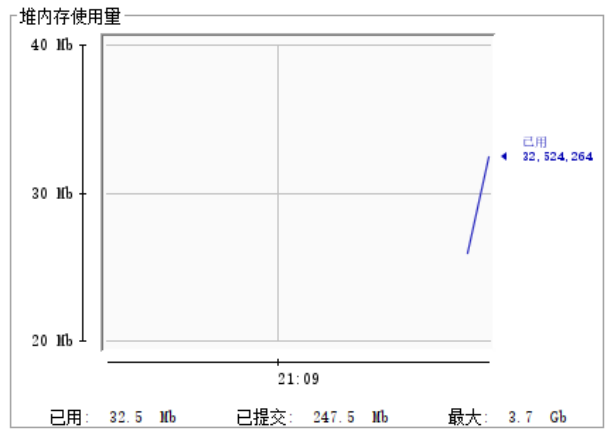
用户名 (U):  口令 (P):

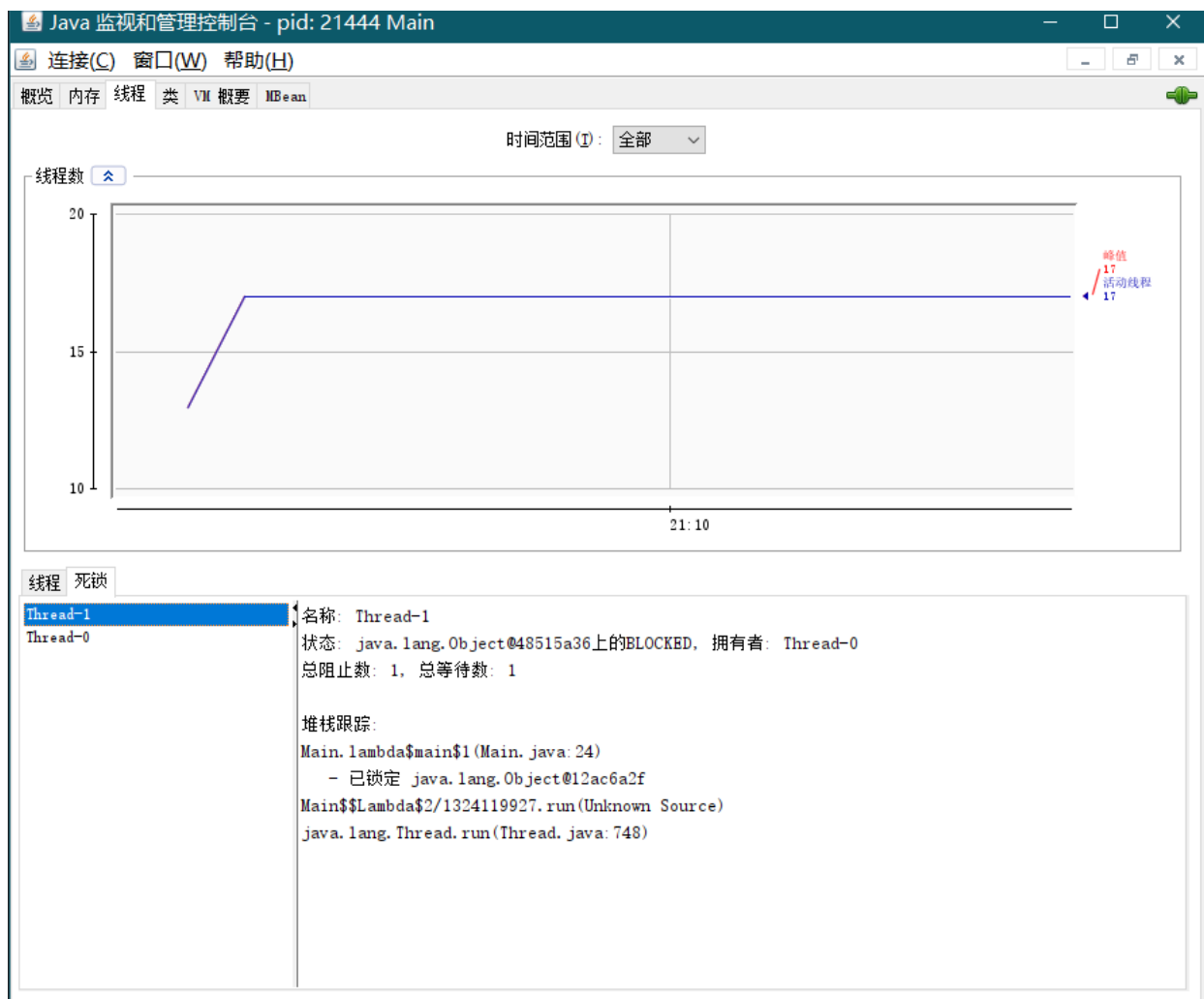
连接 (C)

取消



时间范围 (T): 全部





因此, 前面说不推荐使用 `suspend()` 去挂起线程的原因, 是因为 `suspend()` 在使线程暂停的同时, 并不会去释放任何锁资源。其他线程都无法访问被它占用的锁。直到对应的线程执行 `resume()` 方法后, 被挂起的线程才能继续, 从而其它被阻塞在这个锁的线程才可以继续执行。但是, 如果 `resume()` 操作出现在 `suspend()` 之前执行, 那么线程将一直处于挂起状态, 同时一直占用锁, 这就产生了死锁。

## wait和notify方法

操作系统中的PV操作

★ `Object`类还有三个方法我们从来没有使用过, 分别是 `wait()`、`notify()` 以及 `notifyAll()`, 他们其实是需要配合 `synchronized` 来使用的 (实际上锁就是依附于对象存在的, 每个对象都应该有针对于锁的一些操作, 所以说就这样设计了) 当然, 只有在同步代码块中才能使用这些方法, 正常情况下会报错, 我们来看看他们的作用是什么

★ `wait` 相当于暂停这个线程, 而且释放了它持有的锁, 使得其他线程可以获取到它持有的锁, 当其他线程调用 `notify` 方法后, 就会唤醒 `wait` 的线程(但是不会立即释放锁), 但是必须要等到线程执行结束才释放

```
1 import java.util.TreeMap;
2
3 public class Main {
4     public static void main(String[] args) throws InterruptedException {
```

```

5      Object o1 = new Object(); //锁1
6      Object o2 = new Object(); //锁2
7      Thread t1 = new Thread() -> {
8          synchronized (o1){ //先拿到锁一
9              try {
10                  Thread.sleep(1000);
11                  System.out.println("线程一开始拿到o1锁");
12                  System.out.println("开始等待");
13                  o1.wait(); //暂停进程，并释放锁
14                  synchronized (o2){ //需要锁二才可以执行，而锁2需要等待t2运行完成
15                      System.out.println("线程1拿到o2锁");
16                  }
17              } catch (InterruptedException e) {
18                  e.printStackTrace();
19              }
20          }
21          System.out.println("线程一结束");
22      });
23      Thread t2 = new Thread() -> {
24          synchronized (o2){ //先拿到锁2
25              try {
26                  Thread.sleep(1000);
27                  System.out.println("线程二开始拿到o2锁");
28                  synchronized (o1){ //需要锁1才可以继续执行，而锁1需要等待t1运行完成
29                      System.out.println("线程2拿到o1锁");
30                      o1.notify();
31                  }
32              } catch (InterruptedException e) {
33                  e.printStackTrace();
34              }
35          }
36          System.out.println("线程二结束");
37      });
38      t1.start();
39      t2.start();
40  }
41 }
42
43
44 //执行：
45 线程一开始拿到o1锁
46 开始等待
47 线程二开始拿到o2锁
48 线程2拿到o1锁
49 线程二结束
50 线程1拿到o2锁
51 线程一结束

```

★ 必须是在持有锁的情况下使用(同步代码块内)，否则会抛出异常，这个也很好理解，就是你都没拿到这个锁，你肯定就不能执行啊



```

26         // 消费完成，通知生产者可以继续生产
27         monitor.notifyAll();
28         System.out.println("Notified producer.");
29     }
30 }
31 });
32
33 producer.start();
34 // 确保生产者线程先开始等待
35 try {
36     Thread.sleep(100);
37 } catch (InterruptedException e) {
38     e.printStackTrace();
39 }
40 consumer.start();
41 }
42 }

```

**注意：**在实际应用中，直接使用 `Object` 的等待/通知机制需要非常小心，因为不当的使用很容易导致死锁或不响应。通常推荐使用 `java.util.concurrent` 包下的高级并发工具类，如 `Semaphore`，`CountDownLatch`，`CyclicBarrier` 等，它们提供了更强大且易于控制的同步机制。

★ `wait` 也是可以响应异常操作的

```

1     public static void main(String[] args) throws InterruptedException {
2         Object o1 = new Object(); //锁1
3         Thread t1 = new Thread() -> {
4             synchronized (o1){ //先拿到锁一
5                 try {
6                     System.out.println("线程一开始拿到o1锁");
7                     System.out.println("开始等待");
8                     o1.wait(); //暂停进程，并释放锁
9                 } catch (InterruptedException e) {
10                     System.out.println("等待的过程中被中断了");
11                     e.printStackTrace();
12                 }
13             }
14             System.out.println("线程一结束");
15         });
16         t1.start();
17         t1.interrupt();
18     }

```

🔗 下面这个代码我刚刚开始写的时候看他输出笑了一下

从您提供的执行输出顺序来看，线程3的执行情况并未完整展示，但我们可以基于现有信息推测可能的原因：

1. **线程调度**: 线程的执行顺序受到操作系统线程调度策略的影响, 可能导致线程3在等待执行机会时, 其他线程 (特别是线程1和线程2) 已经完成了它们的任务并退出。如果在您的观察时段内, 线程调度器没有给线程3足够的CPU时间片, 它可能还没来得及完成其执行流程。
2. **锁的获取**: 根据输出, 线程3在尝试获取 o1 锁之后进入了等待状态。如果在线程1和线程2执行完毕后, 没有其他线程持有或再次请求了 o1 锁, 理论上线程3应该能够被唤醒并继续执行, 获取 o2 锁, 然后结束。如果输出没有显示线程3的后续步骤, 可能是因为在您观测时点, 它还在等待被唤醒, 或者唤醒后因某种原因 (如再次尝试获取锁失败、线程被中断等) 未能顺利执行至结束。
3. **潜在的死锁或竞态条件**: 虽然根据您给出的输出, 这次执行似乎避免了预期的死锁, 但代码的设计中仍存在死锁的风险, 尤其是当线程调度的时序稍有不同, 可能导致线程3在等待 o1 或 o2 锁时陷入死锁状态。如果线程2和线程1的执行顺序恰好导致了 o1 和 o2 锁的永久锁定, 那么线程3将无法继续执行。
4. **输出同步问题**: 如果输出语句没有使用同步机制 (如 `synchronized` 块), 在多线程环境下可能存在输出混乱, 导致某些线程的输出没有完整打印出来。

为确保线程3能够执行完毕并打印结束信息, 建议检查和调整代码中的同步逻辑, 以及考虑使用更高级的并发工具来管理线程间的协调, 同时确保输出逻辑的同步以准确反映线程执行情况。

```
1  import java.util.TreeMap;
2
3  public class Main {
4      public static void main(String[] args) throws InterruptedException {
5          Object o1 = new Object(); //锁1
6          Object o2 = new Object(); //锁2
7          Thread t1 = new Thread(() -> {
8              synchronized (o1){ //先拿到锁一
9                  try {
10                      Thread.sleep(1000);
11                      System.out.println("线程一开始拿到o1锁");
12                      System.out.println("开始等待");
13                      o1.wait(); //暂停进程, 并释放锁
14                      synchronized (o2){ //需要锁二才可以执行, 而锁2需要等待t2运行完成才
可以释放
15                          System.out.println("线程1拿到o2锁");
16                      }
17                  } catch (InterruptedException e) {
18                      e.printStackTrace();
19                  }
20              }
21              System.out.println("线程一结束");
22          });
23          Thread t2 = new Thread(() -> {
24              synchronized (o2){ //先拿到锁2
25                  try {
26                      Thread.sleep(1000);
27                      System.out.println("线程二开始拿到o2锁");
28                      synchronized (o1){ //需要锁1才可以继续执行, 而锁1需要等待t1运行完成
才可以释放
29                          System.out.println("线程2拿到o1锁");
30                          o1.notify();
31                      }

```

```

32         } catch (InterruptedException e) {
33             e.printStackTrace();
34         }
35     }
36     System.out.println("线程二结束");
37 });
38
39     Thread t3 = new Thread() -> {
40         synchronized (o1){ //等待进程拿到锁一
41             try {
42                 Thread.sleep(1000);
43                 System.out.println("线程3开始拿到o1锁");
44                 System.out.println("开始等待");
45                 o1.wait(); //暂停进程，并释放锁
46                 synchronized (o2){
47                     System.out.println("线程3拿到o2锁");
48                 }
49             } catch (InterruptedException e) {
50                 e.printStackTrace();
51             }
52         }
53         System.out.println("线程3结束");
54     });
55     t1.start();
56     t2.start();
57     t3.start();
58 }
59 }
60
61
62 //执行
63 线程一开始拿到o1锁
64 开始等待
65 线程二开始拿到o2锁
66 线程3开始拿到o1锁
67 开始等待
68 线程2拿到o1锁
69 线程二结束
70 线程1拿到o2锁
71 线程一结束

```

## ThreadLocal的使用

★ 每个线程都有工作内存，使用 ThreadLocal 类来在工作内存中创建线程供呈现出自己使用

```

1 public static void main(String[] args) throws InterruptedException {
2     ThreadLocal<String> local = new ThreadLocal<>();
3     local.set("Hello world");
4     System.out.println(local.get()); //在当前线程是可以取到数据
5
6     new Thread(() -> System.out.println(local.get())).start(); //在其他线程取
    不到
7 }
8
9 //输出
10 Hello world
11 null

```

```

1 public static void main(String[] args) throws InterruptedException {
2     ThreadLocal<String> local = new ThreadLocal<>(); //注意这是一个泛型类，存储类型
    为我们要存放的变量类型
3     Thread t1 = new Thread(() -> {
4         local.set("lbwnb"); //将变量的值给予ThreadLocal
5         System.out.println("线程1变量值已设定！");
6         try {
7             Thread.sleep(2000); //间隔2秒
8         } catch (InterruptedException e) {
9             e.printStackTrace();
10        }
11        System.out.println("线程1读取变量值：");
12        System.out.println(local.get()); //尝试获取ThreadLocal中存放的变量
13    });
14    Thread t2 = new Thread(() -> {
15        local.set("yyds"); //将变量的值给予ThreadLocal
16        System.out.println("线程2变量值已设定！");
17    });
18    t1.start();
19    Thread.sleep(1000); //间隔1秒
20    t2.start();
21 }

```

★ 希望父线程的 ThreadLocal 设置的变量可以给子线程使用我们可以使用 InheritableThreadLocal 来解决：

```

1 public static void main(String[] args) {
2     ThreadLocal<String> local = new InheritableThreadLocal<>();
3     Thread t = new Thread(() -> {
4         local.set("lbwnb");
5         new Thread(() -> {
6             System.out.println(local.get());
7         }).start();
8     });
9     t.start();
10 }

```



## 定时器

★ 我们有时候会有这样的需求，我希望定时执行任务，比如3秒后执行，其实我们可以通过使用 `Thread.sleep()` 来实现：

2 我们通过自行封装一个 `TimerTask` 类，并在启动时，先休眠3秒钟，再执行我们传入的内容。那么现在我们希望，能否循环执行一个任务呢？比如我希望每隔1秒钟执行一次代码，这样该怎么做呢？

```
1 public static void main(String[] args) {
2     new TimerTask(() -> System.out.println("我是定时任务! "), 3000).start();    //
    创建并启动此定时任务
3 }
4
5 //自定义了类，在类中执行star方法执行休眠
6 static class TimerTask{
7     Runnable task;
8     long time;
9
10    public TimerTask(Runnable runnable, long time){
11        this.task = runnable;
12        this.time = time;
13    }
14
15    public void start(){
16        new Thread(() -> {
17            try {
18                Thread.sleep(time);
19                task.run();    //休眠后再运行
20            } catch (InterruptedException e) {
21                e.printStackTrace();
22            }
23        }).start();
24    }
25 }
```

```
1 public static void main(String[] args) {
2     new TimerLoopTask(() -> System.out.println("我是定时任务! "), 3000).start();
    //创建并启动此定时任务
3 }
4
5 static class TimerLoopTask{
6     Runnable task;
7     long loopTime;
8
9     public TimerLoopTask(Runnable runnable, long loopTime){
10        this.task = runnable;
11        this.loopTime = loopTime;
12    }
13 }
```

```

14     public void start(){
15         new Thread(() -> {
16             try {
17                 while (true){    //无限循环执行
18                     Thread.sleep(loopTime);
19                     task.run();    //休眠后再运行
20                 }
21             } catch (InterruptedException e) {
22                 e.printStackTrace();
23             }
24         }).start();
25     }
26 }

```

★ 但是上面两种方法终究都是自己定义了，很多方面可能没有考虑全面，Java 为我们自己提供了一个框架，用于处理定时任务

1 使用 `Timer` 创建一个定时对象,使用 `schedule` 来调度一个任务，需要传入一个 `TimerTask` 类，我们可以通过此对象来创建任意类型的定时任务，包延时任务、循环定时任务等

```

1 public static void main(String[] args) {
2     Timer timer = new Timer();    //创建定时器对象
3     timer.schedule(new TimerTask() {    //注意这个是一个抽象类，不是接口，无法使用lambda
        表达式简化，只能使用匿名内部类
4         @Override
5         public void run() {
6             System.out.println(Thread.currentThread().getName());    //打印当前线程
        名称
7         }
8     }, 1000);    //执行一个延时任务，设定延迟1s
9 }

```

2 会发现线程并没有终止，在 Java 中需要非守护线程全部结束，运行才会结束，并不是主线程结束了才结束，因为 `Timer` 内存维护了一个任务队列和一个工作线程，

```

1 public class Timer {
2     /**
3      * The timer task queue. This data structure is shared with the timer
4      * thread. The timer produces tasks, via its various schedule calls,
5      * and the timer thread consumes, executing timer tasks as appropriate,
6      * and removing them from the queue when they're obsolete.
7      */
8     private final TaskQueue queue = new TaskQueue(); //任务队列
9
10    /**
11     * The timer thread.
12     */
13    private final TimerThread thread = new TimerThread(queue); //任务线程
14
15    ...
16 }

```

3 如果我想要这个程序每隔一段时间执行一次，可以实现下面这个构建方法，在传入一个每隔一段时间执行一次

```
1 //源代码
2 public void schedule(TimerTask task, long delay, long period) {
3     if (delay < 0)
4         throw new IllegalArgumentException("Negative delay.");
5     if (period <= 0)
6         throw new IllegalArgumentException("Non-positive period.");
7     sched(task, System.currentTimeMillis()+delay, -period);
8 }
```

```
1 public static void main(String[] args) {
2     Timer timer = new Timer(); //创建定时器对象
3     timer.schedule(new TimerTask() {
4         @Override
5         public void run() {
6             System.out.println(Thread.currentThread().getName()); //打印当前线程名称
7         }
8     }, 1000, 1000); //执行一个延时任务,延迟1s执行,且每1s执行一次
9 }
```

4 使用 cancel 方法终止 timer

```
1 public static void main(String[] args) throws InterruptedException {
2     Timer timer = new Timer(); //创建定时器对象
3     timer.schedule(new TimerTask() { //注意这个是一个抽象类,不是接口,无法使用
4         //lambda表达式简化,只能使用匿名内部类
5         @Override
6         public void run() {
7             System.out.println(Thread.currentThread().getName()); //打印当前线程名称
8         }
9     }, 1000, 1000); //执行一个延时任务,延迟1s执行,且每1s执行一次
10    Thread.sleep(5000); //五秒后终止
11    timer.cancel();
12 }
```

★ 在什么的 Timer 源码中,会有个 TimerThread, 它继承自 Thread 是新创建的线程,在构造时候自动启动

```
1 public Timer(String name) {
2     thread.setName(name);
3     thread.start(); //启动线程
4 }
```

1 而它的run方法会循环地读取队列中是否还有任务，如果有任务依次执行，没有的话就暂时处于休眠状态：

```
1 public void run() {
2     try {
3         mainLoop();
4     } finally {
5         // Someone killed this Thread, behave as if Timer cancelled
6         synchronized(queue) {
7             newTasksMayBeScheduled = false;
8             queue.clear(); // Eliminate obsolete references
9         }
10    }
11 }
12
13 /**
14  * The main timer loop. (See class comment.)
15  */
16 private void mainLoop() {
17     try {
18         TimerTask task;
19         boolean taskFired;
20         synchronized(queue) {
21             // wait for queue to become non-empty
22             while (queue.isEmpty() && newTasksMayBeScheduled) //当队列为空同时没有
                被关闭时，会调用wait()方法暂时处于等待状态，当有新的任务时，会被唤醒。
23                 queue.wait();
24             if (queue.isEmpty())
25                 break; //当被唤醒后都没有任务时，就会结束循环，也就是结束工作线程
26             ...
27     }
```

2 newTasksMayBeScheduled 实际上就是标记当前定时器是否关闭，当它为false时，表示已经不会再有新的任务到来，也就是关闭，我们可以通过调用 cancel() 方法来关闭它的工作线程：

```
1 public void cancel() {
2     synchronized(queue) {
3         thread.newTasksMayBeScheduled = false;
4         queue.clear();
5         queue.notify(); //唤醒wait使得工作线程结束
6     }
7 }
```

## 守护线程

⚠ 不要把操作系统重的守护进程和守护线程相提并论！

★ 守护进程在后台运行运行，不需要和用户交互，本质和普通进程类似。而守护线程就不一样了，当其他所有的非守护线程结束之后，守护线程自动结束，也就是说，Java中所有的线程都执行完毕后，守护线程自动结束，因此守护线程不适合进行IO操作，只适合打打杂，因为分配给守护进程的资源一般都比较小；

1 使用 `setDaemon` 设置守护线程，设置了守护线程后，`main` 线程（也就是主线程），主线程结束后，守护线程就会自动结束，即便他还在继续运行，如果不设置为守护线程，那么主线程结束后，子线程还会继续运行

```
1 public static void main(String[] args) throws InterruptedException{
2     Thread t = new Thread(() -> {
3         while (true){
4             try {
5                 System.out.println("程序正常运行中...");
6                 Thread.sleep(1000);
7             } catch (InterruptedException e) {
8                 e.printStackTrace();
9             }
10        }
11    });
12    t.setDaemon(true);    //设置为守护线程（必须在开始之前，中途是不允许转换的）
13    t.start();
14    for (int i = 0; i < 5; i++) {
15        Thread.sleep(1000);
16    }
17 }
```

★ 在守护线程中产生的新线程也是守护的：

```
1 public static void main(String[] args) throws InterruptedException{
2     Thread t = new Thread(() -> {
3         Thread it = new Thread(() -> {
4             while (true){
5                 try {
6                     System.out.println("程序正常运行中...");
7                     Thread.sleep(1000);
8                 } catch (InterruptedException e) {
9                     e.printStackTrace();
10                }
11            }
12        });
13        it.start();
14    });
15    t.setDaemon(true);    //设置为守护线程（必须在开始之前，中途是不允许转换的）
16    t.start();
17    for (int i = 0; i < 5; i++) {
18        Thread.sleep(1000);
19    }
```

## 集合的多线程

★ 之前在学习集合的时候，提到过有个 `splititerator` 和 `parallelStream`，当时只是提了一下

可拆分迭代器（`Splitable Iterator`）和 `Iterator` 一样，`Splititerator` 也用于遍历数据源中的元素，但它是为了并行执行而设计的。Java 8 已经为集合框架中包含的所有数据结构提供了一个默认的 `Splititerator` 实现。在集合跟接口 `Collection` 中提供了一个 `splititerator()` 方法用于获取可拆分迭代器

```

1 //与迭代器作用相同，但是是并行执行的，我们会在下一章多线程部分中进行介绍
2 @Override
3 default Spliterator<E> spliterator() {
4     return Spliterators.splititerator(this, 0);
5 }
6
7 //生成当前集合的流，我们会在后面进行讲解
8 default Stream<E> stream() {
9     return StreamSupport.stream(spliterator(), false);
10 }
11
12 //生成当前集合的并行流，我们会在下一章多线程部分中进行介绍
13 default Stream<E> parallelStream() {
14     return StreamSupport.stream(spliterator(), true);
15 }

```

★ `parallelStream` 就是利用了可拆分迭代器进行多线程操作，就是一个多线程执行的流，它通过默认的 `ForkJoinPool` 实现，它可以提高你的多线程任务的速度

1 下面这个使用并行流代码，使用 `forEach` 方法，发现输出顺序并不是按照列表里面来排的，而且输出的进程名也不一样，其实他就是开了多个线程去执行这个代码

```

1 public static void main(String[] args) {
2     List<Integer> list = new ArrayList<>(Arrays.asList(1, 4, 5, 2, 9, 3, 6, 0));
3     list
4         .parallelStream() //获得并行流
5         .forEach(i -> System.out.println(Thread.currentThread().getName()+"
-> "+i)); //打印线程名字
6 }
7
8 //输出：
9 main -> 3
10 main -> 9
11 main -> 6
12 ForkJoinPool.commonPool-worker-2 -> 0
13 ForkJoinPool.commonPool-worker-11 -> 2
14 ForkJoinPool.commonPool-worker-9 -> 5
15 ForkJoinPool.commonPool-worker-4 -> 1

```

## 2 如果你想顺序输出就可以调用 `forEachOrdered` 方法

```
1 public static void main(String[] args) {
2     List<Integer> list = new ArrayList<>(Arrays.asList(1, 4, 5, 2, 9, 3, 6, 0));
3     list
4         .parallelStream()    //获得并行流
5         .forEachOrdered(System.out::println);
6 }
```

★ 在 `Arrays` 数组工具类中，也包含了很多 `parallel` 方法

方法名	描述	使用示例
<code>parallelSort(int[] a)</code>	并行地对整型数组进行排序。	<code>java\nint[] arr = {2, 5, 1, 7, 6};\nArrays.parallelSort(arr);\nsystem.out.println(Arrays.toString(arr));\n</code>
<code>parallelSort(int[] a, int fromIndex, int toIndex)</code>	并行地对整型数组的一部分进行排序 (从 <code>fromIndex</code> 到 <code>toIndex-1</code> )。	<code>java\nint[] arr = {2, 5, 1, 7, 6};\nArrays.parallelSort(arr, 1, 4);\nsystem.out.println(Arrays.toString(arr));\n</code>
<code>parallelSort(long[] a)</code>	并行地对长整型数组进行排序。	<code>java\nlong[] arr = {2L, 5L, 1L, 7L, 6L};\nArrays.parallelSort(arr);\nsystem.out.println(Arrays.toString(arr));\n</code>
<code>parallelSort(long[] a, int fromIndex, int toIndex)</code>	并行地对长整型数组的一部分进行排序。	<code>java\nlong[] arr = {2L, 5L, 1L, 7L, 6L};\nArrays.parallelSort(arr, 1, 4);\nsystem.out.println(Arrays.toString(arr));\n</code>
<code>parallelSort(double[] a)</code>	并行地对双精度浮点数数组进行排序。	<code>java\ndouble[] arr = {2.2, 5.5, 1.1, 7.7, 6.6};\nArrays.parallelSort(arr);\nsystem.out.println(Arrays.toString(arr));\n</code>
<code>parallelSort(double[] a, int fromIndex, int toIndex)</code>	并行地对双精度浮点数数组的一部分进行排序。	<code>java\ndouble[] arr = {2.2, 5.5, 1.1, 7.7, 6.6};\nArrays.parallelSort(arr, 1, 4);\nsystem.out.println(Arrays.toString(arr));\n</code>
<code>parallelSort(Object[] a)</code>	并行地对对象数组进行排序，要求数组元素实现 <code>Comparable</code> 接口。	<code>java\nString[] arr = {"banana", "apple", "cherry"};\nArrays.parallelSort(arr);\nsystem.out.println(Arrays.toString(arr));\n</code>
<code>parallelSort(Object[] a, int fromIndex, int toIndex)</code>	并行地对对象数组的一部分进行排序。	<code>java\nString[] arr = {"banana", "apple", "cherry"};\nArrays.parallelSort(arr, 0, 2);\nsystem.out.println(Arrays.toString(arr));\n</code>
<code>parallelSort(T[] a, Comparator&lt;? super T&gt; cmp)</code>	并行地对具有指定比较器的对象数组进行排序。	<code>java\nString[] arr = {"banana", "apple", "cherry"};\nArrays.parallelSort(arr, String.CASE_INSENSITIVE_ORDER);\nsystem.out.println(Arrays.toString(arr));\n</code>
<code>parallelSort(T[] a, int fromIndex, int toIndex, Comparator&lt;? super T&gt; cmp)</code>	并行地对对象数组的一部分进行排序，使用指定的比较器。	<code>java\nString[] arr = {"Banana", "apple", "Cherry"};\nArrays.parallelSort(arr, 0, 3, String.CASE_INSENSITIVE_ORDER);\nsystem.out.println(Arrays.toString(arr));\n</code>

```
1 public static void main(String[] args) {
2     // List<Integer> list = new ArrayList<>(Arrays.asList(1, 4, 5, 2, 9, 3, 6, 0));
3     int[] arr = new int[]{1,2,3,5,6,7,0,8};
4     Arrays.parallelSort(arr);
5     system.out.println(Arrays.toString(arr));
6 }
```

★ 学习了线程之后，之前学习的线程可能会有点不适用了，因为在线程中可能要对同一个对象来进行操作，之前学习的集合类都是基于单线程设计的，如果同时操作这个对象，就会出现异步的问题

1 两个线程都操纵一个 `List`，得到的结果往往不是 30000，而且还可能会报错

```
1 public static void main(String[] args) throws InterruptedException {
2     List<Integer> list = new ArrayList<>();
3     new Thread(() -> {
4         for (int i = 0; i < 10000; i++) {
5             list.add(i);    //两个线程同时操作集合类进行插入操作
6         }
7     }).start();
8     new Thread(() -> {
9         for (int i = 1000; i < 20000; i++) {
10             list.add(i);
11         }
12     }).start();
13 }
```

```

12     }).start();
13     Thread.sleep(2000);
14     System.out.println(list.size());
15 }

```

```

D:\software\Java\jdk1.8.0_131\bin\java.exe ...
Exception in thread "Thread-1" java.lang.ArrayIndexOutOfBoundsException: Create breakpoint : 4164
    at java.util.ArrayList.add(ArrayList.java:459)
    at Main.lambda$main$1(Main.java:13) <1 个内部行>
11054

```

❷ 报错原因是，因为之前的集合类，并没有考虑到多线程运行的情况，如果两个线程同时执行，那么有可能两个线程同一时间都执行同一个方法，这种情况下就很容易出问题

```

1 public boolean add(E e) {
2     ensureCapacityInternal(size + 1); // 当数组容量更好还差一个满的时候，这个时候两个线程同时走到了这里，因为都判断为没满，所以说没有进行扩容，但是实际上两个线程都要插入一个元素进来
3     elementData[size++] = e; //当两个线程同时在这里插入元素，直接导致越界访问
4     return true;
5 }

```

★ 在 Java 早期，还有一些老的集合，它们都是线程安全的，因为它们在每个方法都加了个锁，但是这样会导致运行速度变慢，开销更大，但是它们都不会在使用了

```

1 public static void main(String[] args) throws InterruptedException {
2     Vector<Integer> list = new Vector<>(); //我们可以使用Vector代替List使用
3     //Hashtable<Integer, String> 也可以使用Hashtable来代替Map
4     new Thread(() -> {
5         for (int i = 0; i < 1000; i++) {
6             list.add(i);
7         }
8     }).start();
9     new Thread(() -> {
10        for (int i = 1000; i < 2000; i++) {
11            list.add(i);
12        }
13    }).start();
14
15    Thread.sleep(1000);
16    System.out.println(list.size());
17 }

```

❶ 它们的一些源代码

```

1 public synchronized void copyInto(Object[] anArray) {
2     System.arraycopy(elementData, 0, anArray, 0, elementCount);
3 }
4
5 /**
6  * Trims the capacity of this vector to be the vector's current
7  * size. If the capacity of this vector is larger than its current

```



```

8      * size, then the capacity is changed to equal the size by replacing
9      * its internal data array, kept in the field {@code elementData},
10     * with a smaller one. An application can use this operation to
11     * minimize the storage of a vector.
12     */
13     public synchronized void trimToSize() {
14         modCount++;
15         int oldCapacity = elementData.length;
16         if (elementCount < oldCapacity) {
17             elementData = Arrays.copyOf(elementData, elementCount);
18         }
19     }
20
21     /**
22      * Increases the capacity of this vector, if necessary, to ensure
23      * that it can hold at least the number of components specified by
24      * the minimum capacity argument.
25      *
26      * <p>If the current capacity of this vector is less than
27      * {@code minCapacity}, then its capacity is increased by replacing its
28      * internal data array, kept in the field {@code elementData}, with a
29      * larger one. The size of the new data array will be the old size plus
30      * {@code capacityIncrement}, unless the value of
31      * {@code capacityIncrement} is less than or equal to zero, in which case
32      * the new capacity will be twice the old capacity; but if this new size
33      * is still smaller than {@code minCapacity}, then the new capacity will
34      * be {@code minCapacity}.
35      *
36      * @param minCapacity the desired minimum capacity
37     */
38     public synchronized void ensureCapacity(int minCapacity) {
39         if (minCapacity > 0) {
40             modCount++;
41             ensureCapacityHelper(minCapacity);
42         }
43     }

```

★ 在 `Java.util.concurrent` 有很多专门用于并发操作的集合类

```
public class Main {  
    public static void main(String[] args) throws InterruptedException {  
        java.util.concurrent.  
    }  
}
```

- ConcurrentSkipListMap<K, V> java.util.concurrent
- ConcurrentSkipListSet<E> java.util.concurrent
- CopyOnWriteArrayList<E> java.util.concurrent
- CopyOnWriteArraySet<E> java.util.concurrent
- CountDownLatch java.util.concurrent
- CountedCompleter<T> java.util.concurrent
- CyclicBarrier java.util.concurrent
- Delayed java.util.concurrent
- DelayQueue<E> java.util.concurrent
- Exchanger<V> java.util.concurrent
- ExecutionException java.util.concurrent
- Executor java.util.concurrent

按 Enter 插入, 按 Tab 替换 下一提示