

# JUC

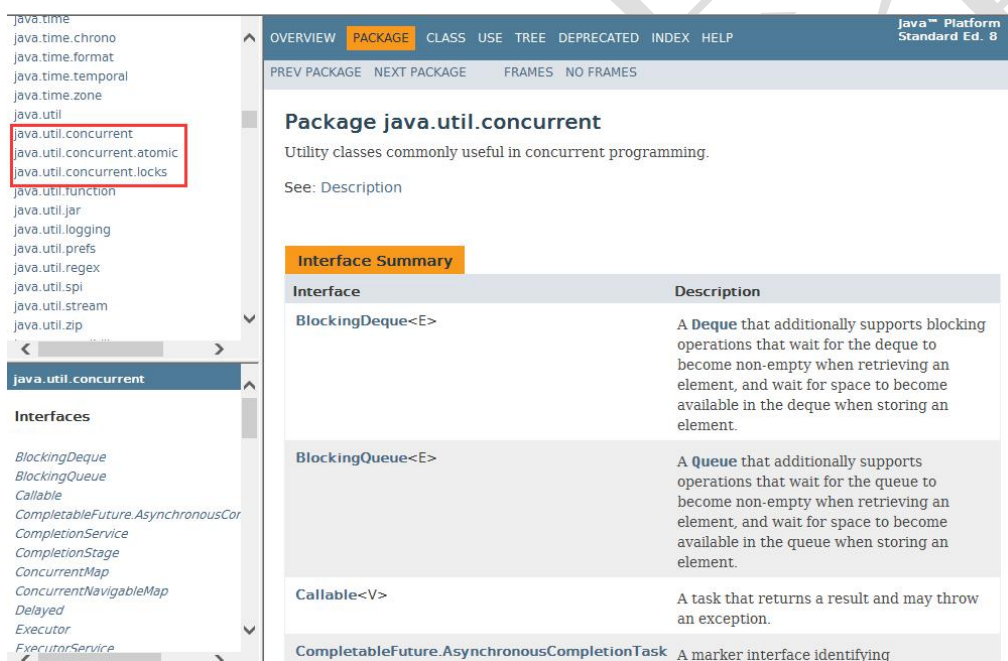
尚硅谷 JAVA 研究院

版本：V1.1

## 第 1 章 JUC 简介

### 1. 是什么

java.util.concurrent 在并发编程中使用的工具类



Package java.util.concurrent

Utility classes commonly useful in concurrent programming.

See: Description

Interface	Description
BlockingDeque<E>	A Deque that additionally supports blocking operations that wait for the deque to become non-empty when retrieving an element, and wait for space to become available in the deque when storing an element.
BlockingQueue<E>	A Queue that additionally supports operations that wait for the queue to become non-empty when retrieving an element, and wait for space to become available in the queue when storing an element.
Callable<V>	A task that returns a result and may throw an exception.
CompletableFuture, AsynchronousCompletionTask	A marker interface identifying

### 2. 进程/线程回顾

#### 2.1 进程/线程

**进程：**进程是一个具有一定独立功能的程序关于某个数据集合的一次运行活动。它是操作系统动态执行的基本

单元，在传统的操作系统中，进程既是基本的分配单元，也是基本的执行单元。

**线程：**通常在一个进程中可以包含若干个线程，当然一个进程中至少有一个线程，不然没有存在的意义。线程可以利用进程所拥有的资源，在引入线程的操作系统中，通常都是把进程作为分配资源的基本单位，而把线程作为独立运行和独立调度的基本单位，由于线程比进程更小，基本上不拥有系统资源，故对它的调度所付出的开销就会小得多，能更高效的提高系统多个程序间并发执行的程度。

## 2.2 进程/线程例子

使用 QQ，查看进程一定有一个 QQ.exe 的进程，我可以用 qq 和 A 文字聊天，和 B 视频聊天，给 C 传文件，给 D 发一段语言，QQ 支持录入信息的搜索。

上学的时候写论文，用 word 写论文，同时用 QQ 音乐放音乐，同时用 QQ 聊天，多个进程。

word 如没有保存，停电关机，再通电后打开 word 可以恢复之前未保存的文档，word 也会检查你的拼写，两个线程：容灾备份，语法检查

## 第 2 章 Lock 接口

### 1. 复习 Synchronized

#### 1.1 多线程编程模板上

- (1) 线程 操作 资源类
- (2) 高内聚低耦合

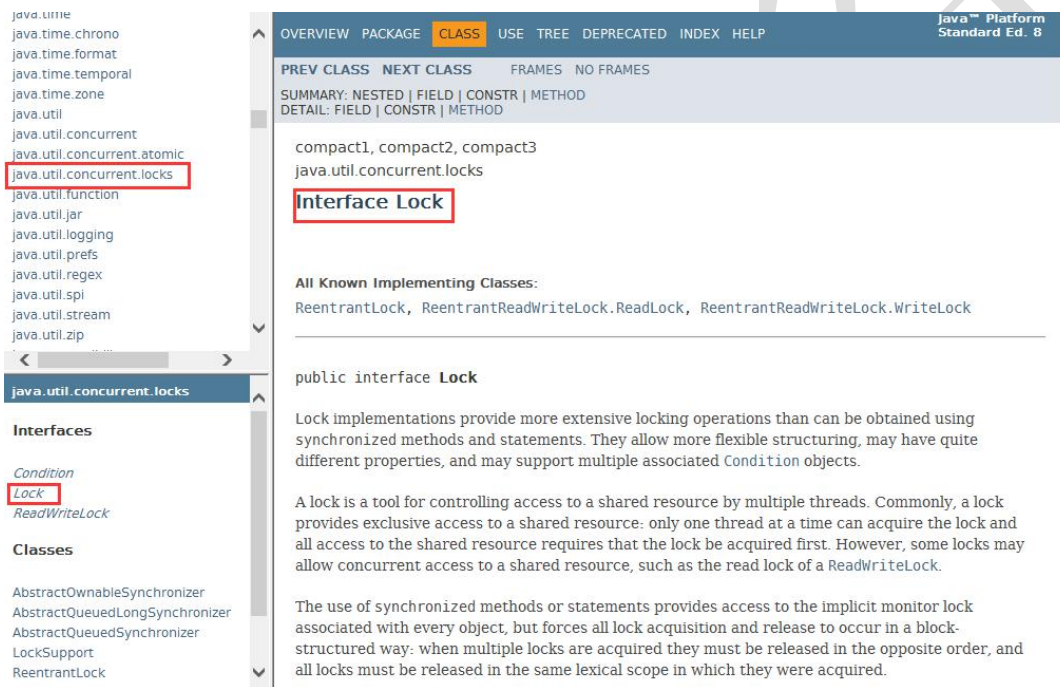
#### 1.2 实现步骤

- (1) 创建资源类
- (2) 资源类里创建同步方法、同步代码块

## 2. Lock

### 2.1 是什么

参考 Java8API



The screenshot displays the Java Platform Standard Ed. 8 API documentation for the `Lock` interface. The left sidebar shows the package hierarchy with `java.util.concurrent.locks` selected. The main content area shows the `Interface Lock` with its description and implementing classes.

**Interface Lock**

All Known Implementing Classes:  
`ReentrantLock`, `ReentrantReadWriteLock.ReadLock`, `ReentrantReadWriteLock.WriteLock`

public interface **Lock**

Lock implementations provide more extensive locking operations than can be obtained using synchronized methods and statements. They allow more flexible structuring, may have quite different properties, and may support multiple associated `Condition` objects.

A lock is a tool for controlling access to a shared resource by multiple threads. Commonly, a lock provides exclusive access to a shared resource: only one thread at a time can acquire the lock and all access to the shared resource requires that the lock be acquired first. However, some locks may allow concurrent access to a shared resource, such as the read lock of a `ReadWriteLock`.

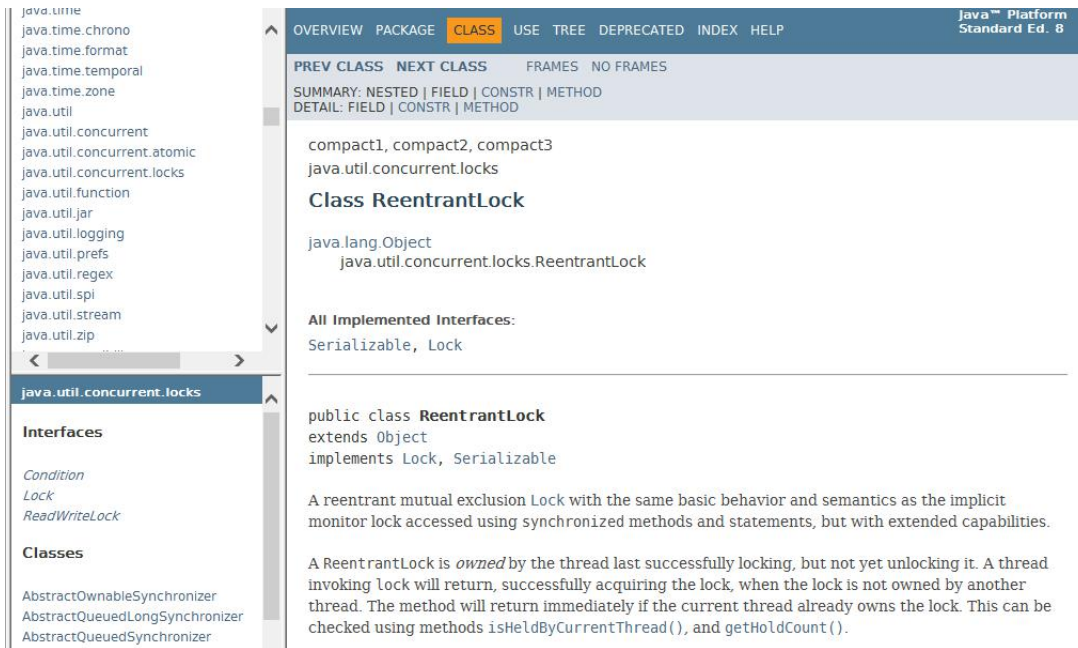
The use of synchronized methods or statements provides access to the implicit monitor lock associated with every object, but forces all lock acquisition and release to occur in a block-structured way: when multiple locks are acquired they must be released in the opposite order, and all locks must be released in the same lexical scope in which they were acquired.

Lock implementations provide more extensive locking operations than can be obtained using synchronized methods and statements. They allow more flexible structuring, may have quite different properties, and may support multiple associated `Condition` objects.

`Lock` 实现提供更广泛的锁定操作可以比使用 `synchronized` 获得方法和声明更好。他们允许更灵活的结构，可以有完全不同的特性，可以支持多个相关的 `Condition` 对象。

### 2.2 Lock 接口的实现

ReentrantLock 可重入锁，参考 Java8API



## 2.3 创建线程方式

### (1) 继承 Thread

例如：

```
public class SaleTicket extends Thread
```

Java 是单继承，资源宝贵，要用接口方式

### (2) new Thread()

例如：

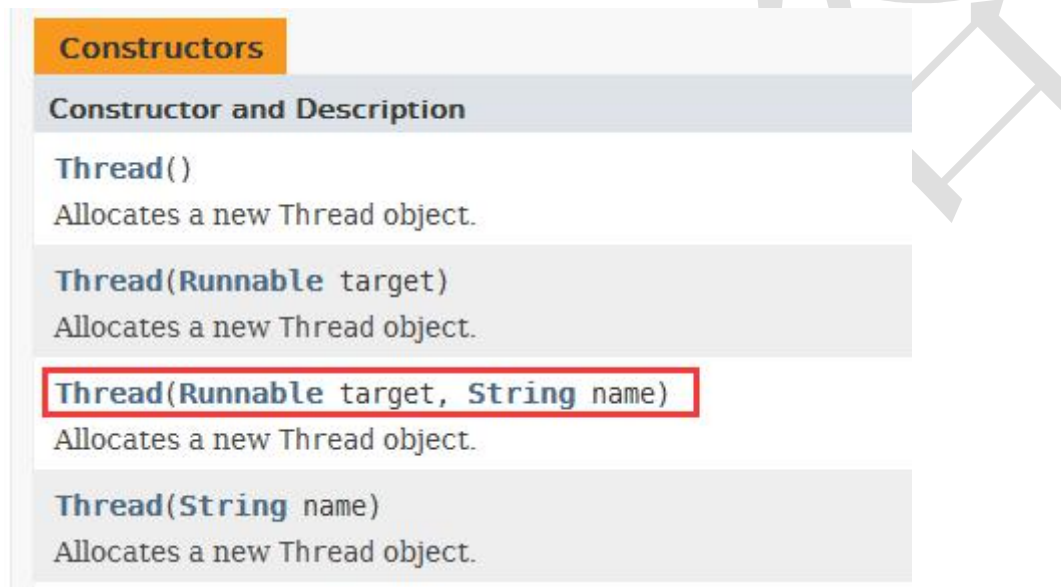
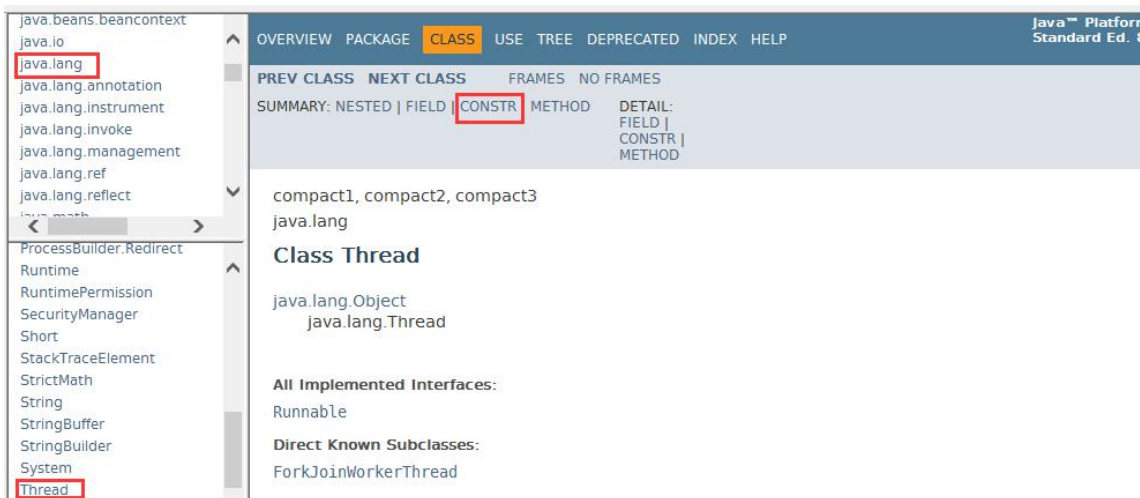
```
Thread t1 = new Thread();
```

```
t1.start();
```

不能这样实现

### (3) Thread(Runnable target, String name)

参考 Java8API



## 2.4 实现线程方法

(1) 新建类实现 runnable 接口

```
class MyThread implements Runnable//新建类实现 runnable 接口
new Thread(new MyThread,...)
```

这种方法会新增类，有更新更好的方法

(2) 匿名内部类

```
new Thread(new Runnable() {
```

```
@Override  
  
public void run() {  
  
}  
  
}, "your thread name").start();
```

这种方法不需要创建新的类，可以 new 接口

### (3) lambda 表达式

```
new Thread(() -> {  
  
}, "your thread name").start();
```

这种方法代码更简洁精炼

## 2.5 程序代码

```
package com.atguigu.thread;  
  
import java.util.concurrent.locks.Lock;  
import java.util.concurrent.locks.ReentrantLock;  
  
class Ticket //实例例 eld +method  
{  
    private int number=30;  
    /* //1 同步 public synchronized void sale()  
    {  
        //2 同步 synchronized(this) {}  
        if(number > 0) {  
            System.out.println(Thread.currentThread().getName()+"卖出"+(number--)+"\t 还剩 number);  
        }  
    }  
}
```

```
*/

// Lock implementations provide more extensive locking operations
// than can be obtained using synchronized methods and statements.

private Lock lock = new ReentrantLock();//List list = new ArrayList()

public void sale()
{
    lock.lock();

    try {
        if(number > 0) {
            System.out.println(Thread.currentThread().getName()+"卖出" +(number--)+"\t 还剩 number);
        }
    } catch (Exception e) {
        e.printStackTrace();
    } finally {
        lock.unlock();
    }

}

}

/**
 *
```

\* @Description:卖票程序个售票出 0 张票

@author xiale

\* 笔记: J 里面如何 1 多线程编-上

1.1 线程 (资里源类 \* 1.2 高内聚 /

```
public class SaleTicket
```

```
{
```

```
    public static void main(String[] args)//main 所有程序
```

```
        Ticket ticket = new Ticket();
```

```
        //Thread(Runnable target, String name) Allocates a new Thread object.
```

```
        new Thread(() -> {for (int i = 1; i < 40; i++)ticket.sale();}, "AA").start();
```

```
        new Thread(() -> {for (int i = 1; i < 40; i++)ticket.sale();}, "BB").start();
```

```
        new Thread(() -> {for (int i = 1; i < 40; i++)ticket.sale();}, "CC").start();
```

```
/* new Thread(new Runnable() {
```

```
    @Override
```

```
    public void run()
```

```
    {
```

```
        for (int i = 1; i <=40; i++)
```

```
        {
```

```
            ticket.sale();
```

```
        }
```

```
    }
```

```
}, "AA").start();
```



```
new Thread(new Runnable() {  
    @Override  
    public void run()  
    {  
        for (int i = 1; i <=40; i++)  
        {  
            ticket.sale();  
        }  
    }  
}, "BB").start();  
new Thread(new Runnable() {  
    @Override  
    public void run()  
    {  
        for (int i = 1; i <=40; i++)  
        {  
            ticket.sale();  
        }  
    }  
}, "CC").start();  
*/  
}  
}
```

## 3. java8 新特性

### 3.1 lambda 表达式

#### (1) 要求

lambda 表达式，如果一个接口只有一个方法，我可以把方法名省略

```
Foo foo = () -> {System.out.println("****hello lambda");};
```

#### (2) 编写规则

拷贝小括号 ( )，写死右箭头->，落地大括号{...}

#### (3) 函数式接口

lambda 表达式，必须是函数式接口，必须只有一个方法

如果接口只有一个方法 java 默认它为函数式接口。

为了正确使用 Lambda 表达式，需要给接口加个注解：@FunctionalInterface

如有两个方法，立刻报错

Runnable 接口为什么可以用 lambda 表达式？

```
// (version 1.8 : 52.0, no super bit)
@java.lang.FunctionalInterface
public abstract interface java.lang.Runnable {

    // Method descriptor #1 0V
    public abstract void run();

}
```

### 3.2 接口里的实现方法

#### (1) default 方法

接口里在 java8 后容许有接口的实现，default 方法默认实现

```
default int div(int x,int y) {  
    return x/y;  
}
```

接口里 **default** 方法可以有几个？

可以有多个，没有限制

## （2） 静态方法实现

静态方法实现：接口新增

```
public static int sub(int x,int y){  
    return x-y;  
}
```

接口里静态方法可以有几个？

可以有多个，没有限制

注意静态的叫类方法，能用 **foo** 去调吗？要改成 **Foo**

## 3.3 程序代码

```
package com.atguigu.thread;  
  
@FunctionalInterface  
interface Foo{  
  
    // public void sayHello() ;  
    // public void say886() ;  
  
    public int add(int x,int y);  
  
    default int div(int x,int y) {  
        return x/y;  
    }  
}
```

```
}

public static int sub(int x,int y) {

    return x-y;

}

}

/**
 *
 * @Description: Lambda Express-----> 函数式编程
 * @author xialei
 * 1 拷贝小括号(形参列表)，写死右箭头 ->，落地大括号 {方法实现}
 * 2 有且只有一个 public 方法@FunctionalInterface 注解增强定义
 * 3 default 方法默认实现
 * 4 静态方法实现
 */
public class LambdaDemo
{

    public static void main(String[] args)

    {

//    Foo foo = new Foo() {

//        @Override

//        public void sayHello() {

//            System.out.println("Hello!!");

//        }

    }
```

```
//  
  
// @Override  
// public void say886() {  
//     // TODO Auto-generated method stub  
//  
// }  
// };  
// foo.sayHello();  
// System.out.println("=====");  
// foo = ()-> {System.out.println("Hello!! lambda !!");};  
// foo.sayHello();  
  
Foo foo = (x,y)->{  
    System.out.println("Hello!! lambda !!");  
    return x+y;  
};  
  
int result = foo.add(3,5);  
System.out.println("*****result="+result);  
System.out.println("*****result div="+foo.div(10, 2));  
System.out.println("*****result sub="+foo.sub(10, 2));  
  
}  
}
```

## 第 3 章 Callable 接口

### 1. 是什么

#### 1.1 面试题：获得多线程的方法几种？

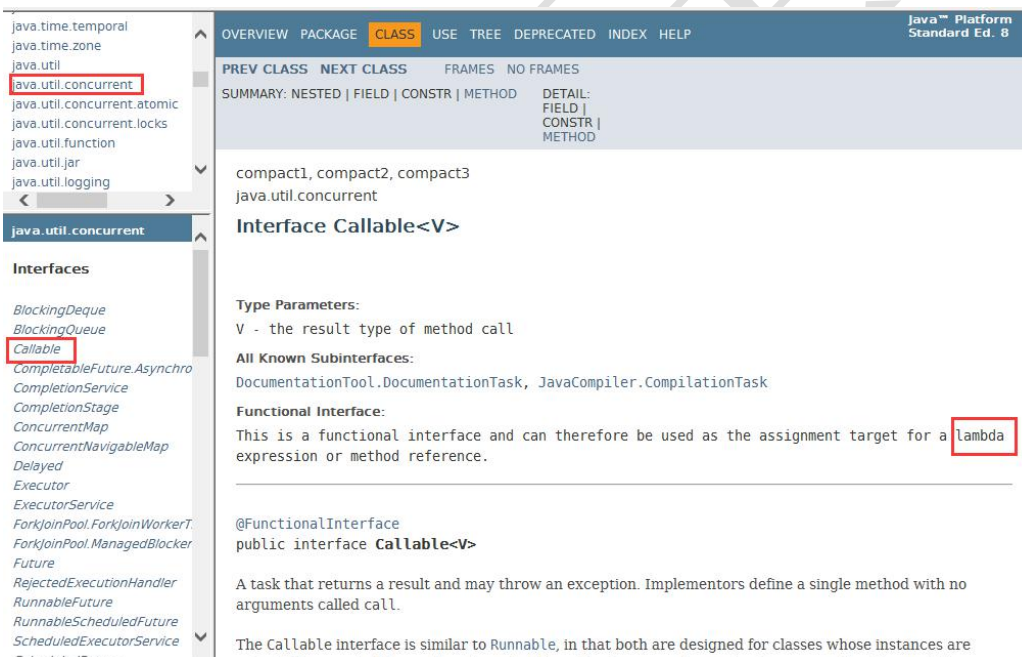
错误回答：

(1) 继承 thread 类 (2) runnable 接口如果只回答这两个你连被问到 juc 的机会都没有。

正确回答：

传统的是继承 thread 类和实现 runnable 接口，java5 以后又有实现 callable 接口和 java 的线程池获得。

#### 1.2 功能接口



这是一个功能接口，因此可以用作 lambda 表达式或方法引用的赋值对象。

## 2. 与 runnable 对比

创建新类 MyThread 实现 runnable 接口

```
class MyThread implements Runnable{  
    @Override  
    public void run() {  
    }  
}
```

新类 MyThread2 实现 callable 接口

```
class MyThread2 implements Callable<Integer>{  
    @Override  
    public Integer call() throws Exception {  
        return 200;  
    }  
}
```

面试题:callable 接口与 runnable 接口的区别?

- 答: (1) 是否有返回值  
(2) 是否抛异常  
(3) 落地方法不一样, 一个是 run, 一个是 call

## 3. 怎么用

### 3.1 直接替换 runnable 是否可行?



```
new Thread(new MyThread(), "AA").start();
```

不可行, 因为: thread 类的构造方法根本没有 Callable

**Thread()**  
Allocates a new Thread object.

**Thread(Runnable target)**  
Allocates a new Thread object.

**Thread(Runnable target, String name)**  
Allocates a new Thread object.

**Thread(String name)**  
Allocates a new Thread object.

**Thread(ThreadGroup group, Runnable target)**  
Allocates a new Thread object.

**Thread(ThreadGroup group, Runnable target, String name)**  
Allocates a new Thread object so that it has target as its run object, has the specified name as its name, and belongs to the thread group referred to by group.

**Thread(ThreadGroup group, Runnable target, String name, long stackSize)**  
Allocates a new Thread object so that it has target as its run object, has the specified name as its name, and belongs to the thread group referred to by group, and has the specified *stack size*.

**Thread(ThreadGroup group, String name)**  
Allocates a new Thread object.

### 3.2 认识陌生人找中间人

这像认识一个不认识的同学，我可以找中间人介绍。

中间人是什么？**java 多态，一个类可以实现多个接口！！**

java.lang

Interfaces

Appendable  
AutoCloseable  
CharSequence  
Cloneable  
Comparable  
Iterable  
Readable  
Runnable

Interface Runnable

All Known Subinterfaces:  
RunnableFuture<V>, RunnableScheduledFuture<V>

All Known Implementing Classes:  
AsyncBoxView.ChildState, ForkJoinWorkerThread, **FutureTask**, RenderableImageProducer, SwingWorker, Thread, TimerTask

Functional Interface:  
This is a functional interface and can therefore be used as the assignment target for a lambda expression or method reference.

```
FutureTask<Integer> ft = new FutureTask<Integer>(new MyThread());

new Thread(ft, "AA").start();
```

运行成功后如何获得返回值？

java.util

**FutureTask**  
LinkedBlockingDeque  
LinkedBlockingQueue  
LinkedTransferQueue  
Phaser  
PriorityBlockingQueue

All Methods Instance Methods Concrete Methods

Modifier and Type	Method and Description
boolean	<b>cancel</b> (boolean mayInterruptIfRunning) Attempts to cancel execution of this task.
protected void	<b>done</b> () Protected method invoked when this task transitions to state isDone (whether normally or via cancellation).
V	<b>get</b> () Waits if necessary for the computation to complete, and then retrieves its result.

```
ft.get();
```



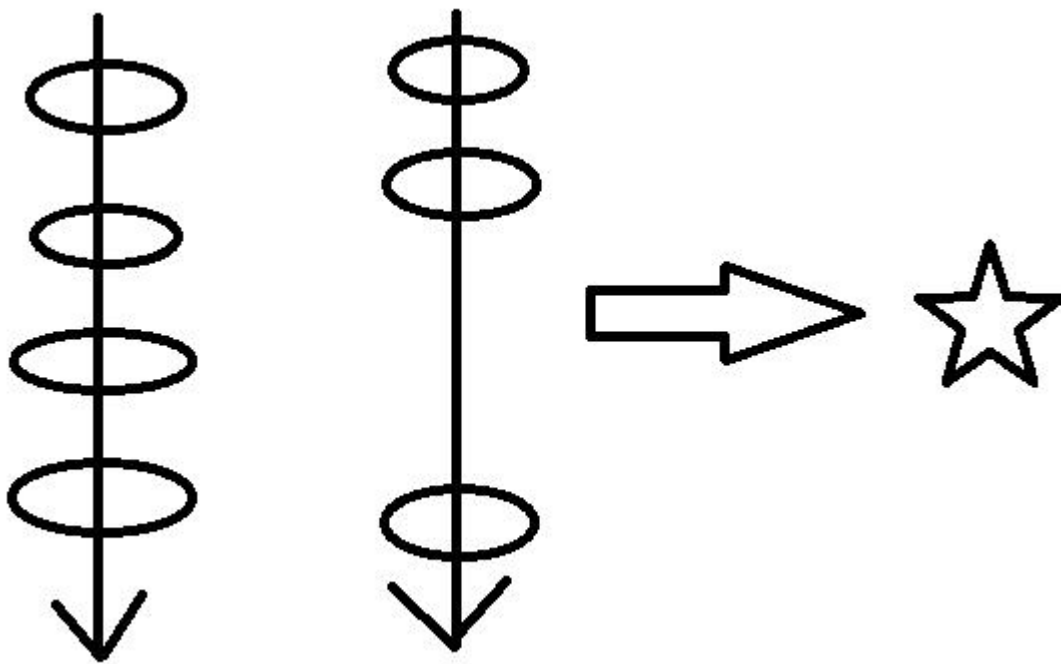
### 3.3 FutureTask

#### (1) 是什么

未来的任务，用它就干一件事，异步调用

main 方法就像一个冰糖葫芦，一个个方法由 main 串起来。

但解决不了一个问题：正常调用挂起堵塞问题



例子：

- 1、老师上着课，口渴了，去买水不合适，讲课线程继续，我可以单起个线程找班长帮忙买水，水买回来了放桌上，我需要的时候再去 get。
- 2、4 个同学，A 算  $1+20$ , B 算  $21+30$ , C 算  $31*$  到 40, D 算  $41+50$ ，是不是 C 的计算量有点大啊，FutureTask 单起个线程给 C 计算，我先汇总 ABD，最后等 C 计算完了再汇总 C，拿到最终结果
- 3、高考：会做的先做，不会的放在后面做

#### (2) 原理

在主线程中需要执行比较耗时的操作时，但又不想阻塞主线程时，可以把这些作业交给 Future 对象在后台完成，当主线程将来需要时，就可以通过 Future 对象获得后台作业的计算结果或者执行状态。

一般 `FutureTask` 多用于耗时的计算，主线程可以在完成自己的任务后，再去获取结果。

仅在计算完成时才能检索结果；如果计算尚未完成，则阻塞 `get` 方法。一旦计算完成，就不能再重新开始或取消计算。`get` 方法而获取结果只有在计算完成时获取，否则会一直阻塞直到任务转入完成状态，然后会返回结果或者抛出异常。

只计算一次

`get` 方法放到最后

### (3) 程序代码

```
package com.atguigu.thread;

import java.util.concurrent.Callable;
import java.util.concurrent.ExecutionException;
import java.util.concurrent.FutureTask;

class MyThread implements Callable<Integer>
{
    @Override
    public Integer call() throws Exception
    {
        Thread.sleep(4000);

        System.out.println(Thread.currentThread().getName()+" *****come in call");

        return 200;
    }
}
```

```
/**
 *
 * @Description: Callable 接口获得多线程
 * @author xialei
 * @date
 * 笔记结论见最后
 */

public class CallableDemo
{
    public static void main(String[] args) throws InterruptedException, ExecutionException
    {

        FutureTask<Integer> ft = new FutureTask<Integer>(new MyThread());
        new Thread(ft, "AA").start();

        /*FutureTask<Integer> ft2 = new FutureTask<Integer>(new MyThread());
        new Thread(ft2, "BB").start();*/

        System.out.println(Thread.currentThread().getName()+"-----main");

        Integer result = ft.get();

        //Integer result2 = ft2.get();
    }
}
```

```
System.out.println("*****result: "+result);
```

```
}
```

```
}
```

## 第 4 章 线程间通信

### 1. 线程间通信

- (1) 生产者+消费者
- (2) 通知等待唤醒机制

#### 1.1 多线程编程模板下

- (1) 判断
- (2) 干活
- (3) 通知

### 2. synchronized 实现

#### 2.1 程序代码

```
package com.atguigu.thread;
```

```
import java.util.concurrent.locks.Condition;

import java.util.concurrent.locks.Lock;

import java.util.concurrent.locks.ReentrantLock;


import org.omg.IOP.Codec;


class ShareDataOne//资源类
{
    private int number = 0;//初始值为零的一个变量

    public synchronized void increment() throws InterruptedException
    {
        //1 判断
        if(number !=0 ) {
            this.wait();
        }
        //2 干活
        ++number;

        System.out.println(Thread.currentThread().getName()+"\t"+number);

        //3 通知
        this.notifyAll();
    }

    public synchronized void decrement() throws InterruptedException
    {

```

```
// 1 判断
if (number == 0) {
    this.wait();
}

// 2 干活
--number;

System.out.println(Thread.currentThread().getName() + "\t" + number);

// 3 通知
this.notifyAll();
}
}

/**
 *
 * @Description:
 * 现在两个线程，
 * 可以操作初始值为零的一个变量，
 * 实现一个线程对该变量加 1，一个线程对该变量减 1，
 * 交替，来 10 轮。
 * @author xialei
 *
 * * 笔记：Java 里面如何进行工程级别的多线程编写
 * 1 多线程变成模板（套路）-----上
 *     1.1 线程    操作    资源类
 *     1.2 高内聚  低耦合
 * 2 多线程变成模板（套路）-----下
 *     2.1 判断
```

\* 2.2 干活

\* 2.3 通知

\*/

```
public class NotifyWaitDemoOne
```

```
{
```

```
    public static void main(String[] args)
```

```
    {
```

```
        ShareDataOne sd = new ShareDataOne();
```

```
        new Thread(() -> {
```

```
            for (int i = 1; i < 10; i++) {
```

```
                try {
```

```
                    sd.increment();
```

```
                } catch (InterruptedException e) {
```

```
                    // TODO Auto-generated catch block
```

```
                    e.printStackTrace();
```

```
                }
```

```
            }
```

```
        }, "A").start();
```

```
        new Thread(() -> {
```

```
            for (int i = 1; i < 10; i++) {
```

```
                try {
```

```
                    sd.decrement();
```

```
                } catch (InterruptedException e) {
```

```
                    // TODO Auto-generated catch block
```

```
                    e.printStackTrace();
```

```
                }
```

```
    }  
    }, "B").start();  
}  
}
```

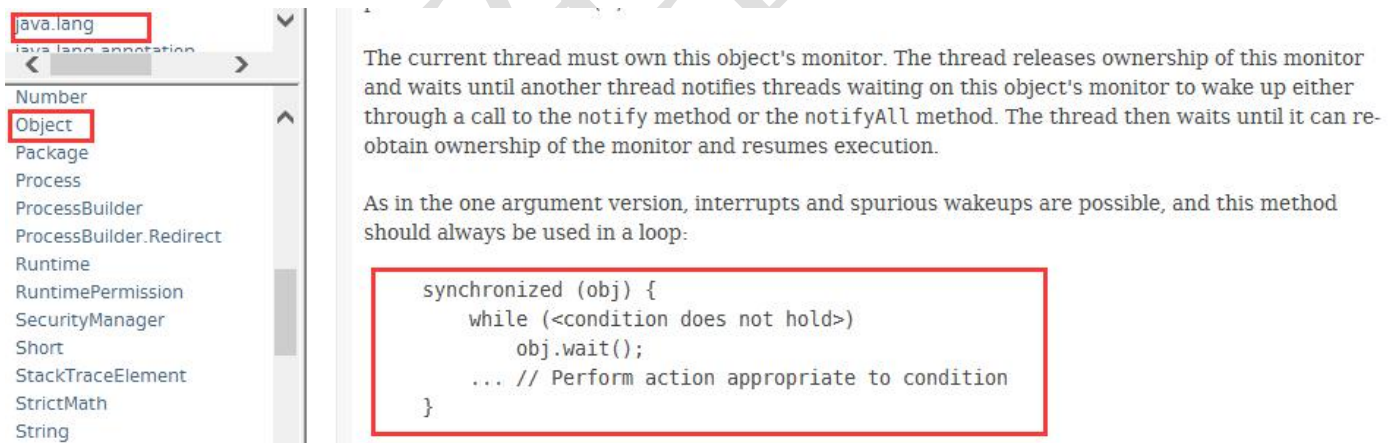
## 2.2 换成 4 个线程

换成 4 个线程会导致错误，虚假唤醒

原因：在 java 多线程判断时，不能用 if，程序出事出在了判断上面，突然有一天加的线程进到 if 了，突然中断了交出控制权，没有进行验证，而是直接走下去了，加了两次，甚至多次

## 2.3 虚假唤醒解决办法

解决虚假唤醒：查看 API，java.lang.Object



The current thread must own this object's monitor. The thread releases ownership of this monitor and waits until another thread notifies threads waiting on this object's monitor to wake up either through a call to the notify method or the notifyAll method. The thread then waits until it can re-obtain ownership of the monitor and resumes execution.

As in the one argument version, interrupts and spurious wakeups are possible, and this method should always be used in a loop:

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

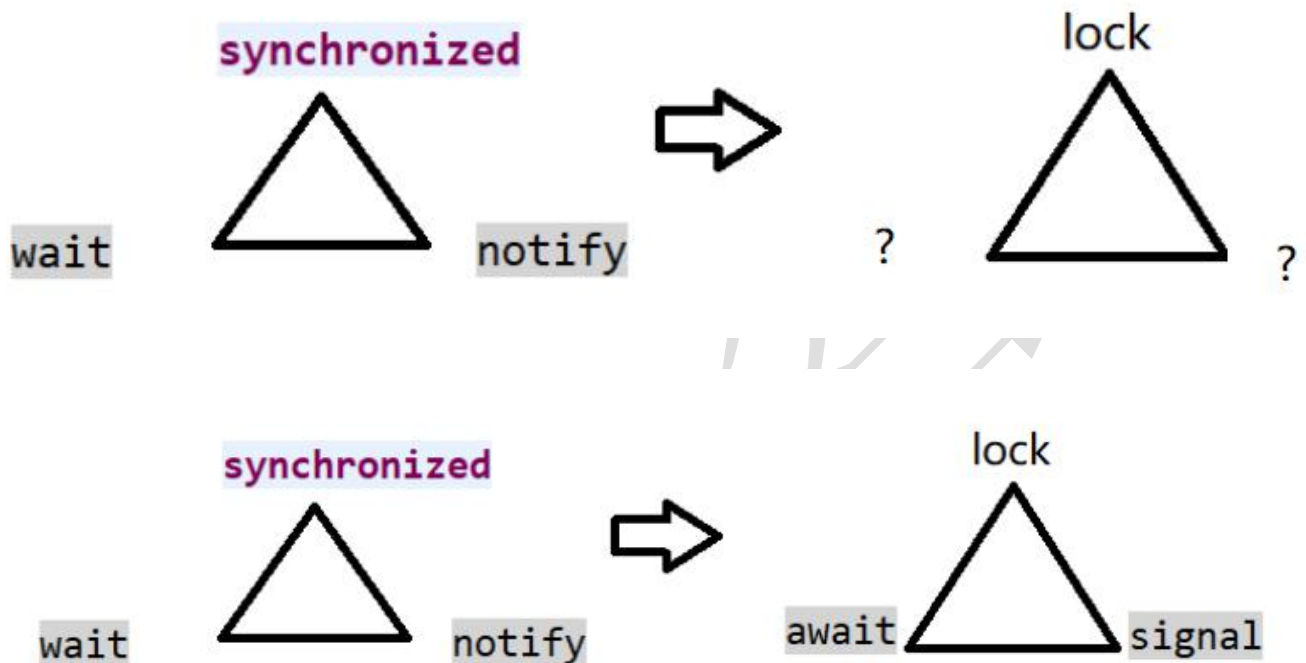
中断和虚假唤醒是可能产生的，所以要用 loop 循环，if 只判断一次，while 是只要唤醒就要拉回来再判断一次。

if 换成 while



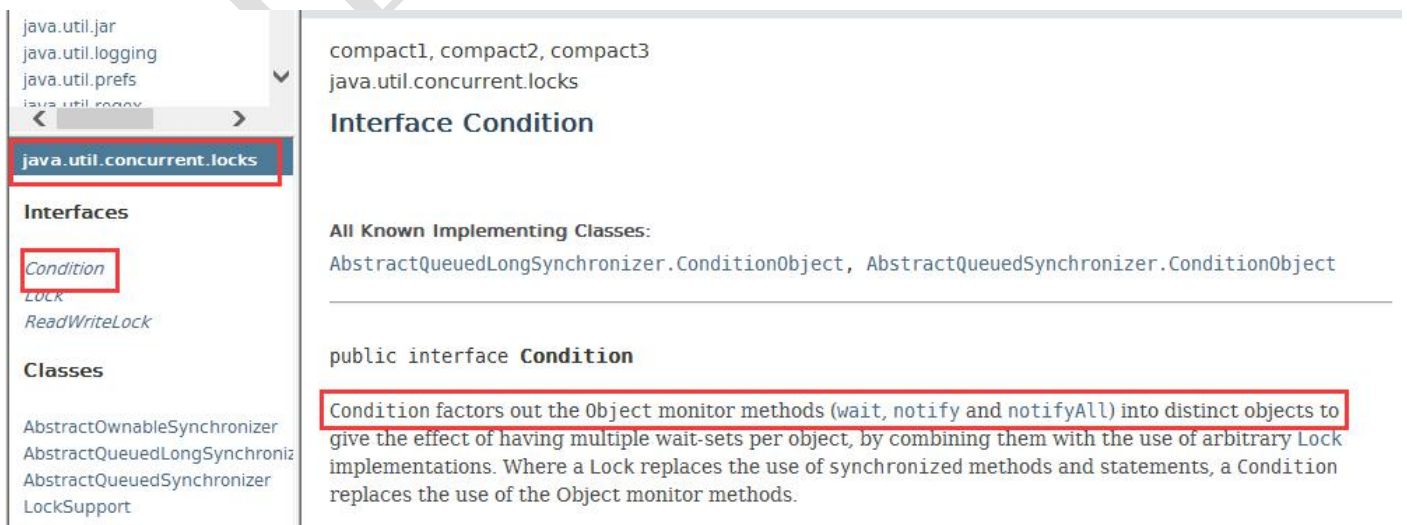
### 3. java8 新版实现

#### 3.1 对标实现



#### 3.2 Condition

Condition: 查看 API, [java.util.concurrent](#)



The screenshot shows the Java API documentation for the `Condition` interface in the `java.util.concurrent.locks` package. The package name is highlighted in the left sidebar. The `Condition` interface is listed under the "Interfaces" section and is also highlighted. The main content area shows the following details:

- compact1, compact2, compact3**
- java.util.concurrent.locks**
- Interface Condition**
- All Known Implementing Classes:** `AbstractQueuedLongSynchronizer.ConditionObject`, `AbstractQueuedSynchronizer.ConditionObject`
- public interface Condition**
- Condition factors out the Object monitor methods (`wait`, `notify` and `notifyAll`) into distinct objects to give the effect of having multiple wait-sets per object, by combining them with the use of arbitrary Lock implementations. Where a Lock replaces the use of synchronized methods and statements, a Condition replaces the use of the Object monitor methods.**

API 例子:

```
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();  
        }  
    }  
}
```

### 3.3 程序代码

```
package com.atguigu.thread;
```

```
import java.util.concurrent.locks.Condition;

import java.util.concurrent.locks.Lock;

import java.util.concurrent.locks.ReentrantLock;


import org.omg.IOP.Codec;


class ShareData//资源类
{
    private int number = 0;//初始值为零的一个变量

    private Lock lock = new ReentrantLock();
    private Condition condition = lock.newCondition();

    public void increment() throws InterruptedException
    {
        lock.lock();
        try {
            //判断
            while(number!=0) {
                condition.await();
            }

            //干活
            ++number;

            System.out.println(Thread.currentThread().getName()+" \t "+number);

            //通知
```

```
        condition.signalAll();

    } catch (Exception e) {

        e.printStackTrace();

    } finally {

        lock.unlock();

    }

}

public void decrement() throws InterruptedException
{

    lock.lock();

    try {

        //判断
        while(number!=1) {

            condition.await();

        }

        //干活
        --number;

        System.out.println(Thread.currentThread().getName()+" \t "+number);

        //通知
        condition.signalAll();

    } catch (Exception e) {

        e.printStackTrace();

    } finally {
```

```
        lock.unlock();
    }

}

/*public synchronized void increment() throws InterruptedException
{
    //判断
    while(number!=0) {
        this.wait();
    }
    //干活
    ++number;
    System.out.println(Thread.currentThread().getName()+"\t "+number);
    //通知
    this.notifyAll();
}

public synchronized void decrement() throws InterruptedException
{
    //判断
    while(number!=1) {
        this.wait();
    }
    //干活
    --number;
    System.out.println(Thread.currentThread().getName()+"\t "+number);
}
```

```
//通知
this.notifyAll();

}*/
}

/**
 *
 * @Description:
 * 现在两个线程，
 * 可以操作初始值为零的一个变量，
 * 实现一个线程对该变量加 1，一个线程对该变量减 1，
 * 交替，来 10 轮。
 * @author xialei
 *
 * * 笔记：Java 里面如何进行工程级别的多线程编写
 * 1 多线程变成模板（套路）-----上
 *     1.1 线程    操作    资源类
 *     1.2 高内聚 低耦合
 * 2 多线程变成模板（套路）-----下
 *     2.1 判断
 *     2.2 干活
 *     2.3 通知
 *
 */
public class NotifyWaitDemo
{
    public static void main(String[] args)
```

```
{  
  
    ShareData sd = new ShareData();  
  
    new Thread(() -> {  
  
        for (int i = 1; i <= 10; i++) {  
  
            try {  
  
                sd.increment();  
  
            } catch (InterruptedException e) {  
  
                e.printStackTrace();  
  
            }  
  
        }  
  
    }, "A").start();  
  
    new Thread(() -> {  
  
        for (int i = 1; i <= 10; i++) {  
  
            try {  
  
                sd.decrement();  
  
            } catch (InterruptedException e) {  
  
                e.printStackTrace();  
  
            }  
  
        }  
  
    }, "B").start();  
  
    new Thread(() -> {  
  
        for (int i = 1; i <= 10; i++) {  
  
            try {
```

```
        sd.increment();
    } catch (InterruptedException e) {
        e.printStackTrace();
    }
}
}, "C").start();
new Thread(() -> {

    for (int i = 1; i <= 10; i++) {
        try {
            sd.decrement();
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
    }
}, "D").start();

}

}
```



## 第 5 章 线程间定制化调用通信

### 1. 实现步骤

- (1) 有顺序通知，需要有标识位
- (2) 有一个锁 Lock，3 把钥匙 Condition
- (3) 判断标志位
- (4) 输出线程名+第几次+第几轮
- (5) 修改标志位，通知下一个

### 2. 程序代码

```
package com.atguigu.thread;

import java.util.concurrent.locks.Condition;
import java.util.concurrent.locks.Lock;
import java.util.concurrent.locks.ReentrantLock;

class ShareResource
{
    private int number = 1;//1:A 2:B 3:C

    private Lock lock = new ReentrantLock();

    private Condition c1 = lock.newCondition();
    private Condition c2 = lock.newCondition();
    private Condition c3 = lock.newCondition();
```

```
public void print5(int totalLoopNumber)
{
    lock.lock();

    try
    {
        //1 判断
        while(number != 1)
        {
            //A 就要停止
            c1.await();
        }
        //2 干活
        for (int i = 1; i <=5; i++)
        {
            System.out.println(Thread.currentThread().getName()+"\t"+i+"\t"
totalLoopNumber:
"+totalLoopNumber);
        }
        //3 通知
        number = 2;
        c2.signal();
    } catch (Exception e) {
        e.printStackTrace();
    } finally {
        lock.unlock();
    }
}

public void print10(int totalLoopNumber)
```

```
{  
    lock.lock();  
    try  
    {  
        //1 判断  
        while(number != 2)  
        {  
            //A 就要停止  
            c2.await();  
        }  
        //2 干活  
        for (int i = 1; i <=10; i++)  
        {  
            System.out.println(Thread.currentThread().getName()+"\t"+i+"\t"  
totalLoopNumber:  
"+totalLoopNumber);  
        }  
        //3 通知  
        number = 3;  
        c3.signal();  
    } catch (Exception e) {  
        e.printStackTrace();  
    } finally {  
        lock.unlock();  
    }  
}  
  
public void print15(int totalLoopNumber)
```

```
{
    lock.lock();

    try
    {
        //1 判断
        while(number != 3)
        {
            //A 就要停止
            c3.await();
        }
        //2 干活
        for (int i = 1; i <=15; i++)
        {
            System.out.println(Thread.currentThread().getName()+"\t"+i+"\t"
totalLoopNumber:
"+totalLoopNumber);
        }
        //3 通知
        number = 1;
        c1.signal();
    } catch (Exception e) {
        e.printStackTrace();
    } finally {
        lock.unlock();
    }
}
```

```
/**
 *
 * @Description:
 * 多线程之间按顺序调用，实现 A->B->C
 * 三个线程启动，要求如下：
 *
 * AA 打印 5 次，BB 打印 10 次，CC 打印 15 次
 * 接着
 * AA 打印 5 次，BB 打印 10 次，CC 打印 15 次
 * .....来 10 轮
 * @author xialei
 *
 */
public class ThreadOrderAccess
{
    public static void main(String[] args)
    {
        ShareResource sr = new ShareResource();

        new Thread(() -> {
            for (int i = 1; i <=10; i++)
            {
                sr.print5(i);
            }
        }, "AA").start();

        new Thread(() -> {
```

```
        for (int i = 1; i <=10; i++)  
        {  
            sr.print10(i);  
        }  
    }, "BB").start();  
    new Thread(() -> {  
        for (int i = 1; i <=10; i++)  
        {  
            sr.print15(i);  
        }  
    }, "CC").start();  
}  
}
```

## 第 6 章多线程锁

### 1. 程序代码

```
package com.atguigu.thread;
```

```
import java.util.concurrent.TimeUnit;

class Phone
{

    public synchronized void sendSMS() throws Exception
    {

        System.out.println("-----sendSMS");
    }

    public synchronized void sendEmail() throws Exception
    {

        System.out.println("-----sendEmail");
    }

    public void getHello()
    {

        System.out.println("-----getHello");
    }

}

/**
 *
 * @Description: 8 锁
 * @author xialei
```

- \*
  - 1 标准访问，先打印短信还是邮件
  - 2 停 4 秒在短信方法内，先打印短信还是邮件
  - 3 新增普通的 hello 方法，是先打短信还是 hello
  - 4 现在有两部手机，先打印短信还是邮件
  - 5 两个静态同步方法，1 部手机，先打印短信还是邮件
  - 6 两个静态同步方法，2 部手机，先打印短信还是邮件
  - 7 1 个静态同步方法,1 个普通同步方法，1 部手机，先打印短信还是邮件
  - 8 1 个静态同步方法,1 个普通同步方法，2 部手机，先打印短信还是邮件

\* -----

\*

\*/

```
public class Lock_8
{
    public static void main(String[] args) throws Exception
    {

        Phone phone = new Phone();
        Phone phone2 = new Phone();

        new Thread(() -> {
            try {
                phone.sendSMS();
            } catch (Exception e) {
                e.printStackTrace();
            }
        }, "AA").start();
    }
}
```



```
Thread.sleep(100);

new Thread(() -> {
    try {
        phone.sendEmail();
        //phone.getHello();
        //phone2.sendEmail();
    } catch (Exception e) {
        e.printStackTrace();
    }
}, "BB").start();
}
```

## 2. 锁的 8 个问题

- (1) 标准访问，先打印短信还是邮件
- (2) 停 4 秒在短信方法内，先打印短信还是邮件
- (3) 普通的 hello 方法，是先打短信还是 hello
- (4) 现在有两部手机，先打印短信还是邮件
- (5) 两个静态同步方法，1 部手机，先打印短信还是邮件
- (6) 两个静态同步方法，2 部手机，先打印短信还是邮件

(7) 1 个静态同步方法，1 个普通同步方法，1 部手机，先打印短信还是邮件

(8) 1 个静态同步方法，1 个普通同步方法，2 部手机，先打印短信还是邮件

### 3. 锁的 8 个问题分析

一个对象里面如果有多个 `synchronized` 方法，某一个时刻内，只要一个线程去调用其中的一个 `synchronized` 方法了，其它的线程都只能等待，换句话说，某一个时刻内，只能有唯一一个线程去访问这些 `synchronized` 方法锁的是当前对象 `this`，被锁定后，其它的线程都不能进入到当前对象的其它的 `synchronized` 方法

加个普通方法后发现和同步锁无关

换成两个对象后，不是同一把锁了，情况立刻变化。

`synchronized` 实现同步的基础：Java 中的每一个对象都可以作为锁。

具体表现为以下 3 种形式。

对于普通同步方法，锁是当前实例对象。

对于静态同步方法，锁是当前类的 `Class` 对象。

对于同步方法块，锁是 `Synchronized` 括号里配置的对象

当一个线程试图访问同步代码块时，它首先必须得到锁，退出或抛出异常时必须释放锁。

也就是说如果一个实例对象的非静态同步方法获取锁后，该实例对象的其他非静态同步方法必须等待获取锁的方法释放锁后才能获取锁，可是别的实例对象的非静态同步方法因为跟该实例对象的非静态同步方法用的是不同的锁，所以毋须等待该实例对象已获取锁的非静态同步方法释放锁就可以获取他们自己的锁。

所有的静态同步方法用的也是同一把锁——类对象本身，这两把锁是两个不同的对象，所以静态同步方法与非

静态同步方法之间是不会有竞态条件的。但是一旦一个静态同步方法获取锁后，其他的静态同步方法都必须等待该方法释放锁后才能获取锁，而不管是同一个实例对象的静态同步方法之间，还是不同的实例对象的静态同步方法之间，只要它们同一个类的实例对象！

## 第 7 章 JUC 强大的辅助类讲解

### 1. ReentrantReadWriteLock 读写锁

类似软件：红蜘蛛

程序代码：

```
package com.atguigu.thread;

import java.util.concurrent.locks.ReentrantReadWriteLock;

class MyQueue
{

    private Object obj;

    private ReentrantReadWriteLock rwLock = new ReentrantReadWriteLock();

    public void readObj()
    {
        rwLock.readLock().lock();

        try
        {
```

```
        System.out.println(Thread.currentThread().getName()+"\t"+obj);
    } catch (Exception e) {
        e.printStackTrace();
    } finally {
        rwLock.readLock().unlock();
    }
}

public void writeObj(Object obj)
{
    rwLock.writeLock().lock();
    try
    {
        this.obj = obj;
        System.out.println(Thread.currentThread().getName()+"writeThread:\t"+obj);
    } catch (Exception e) {
        e.printStackTrace();
    } finally {
        rwLock.writeLock().unlock();
    }
}

/**
 *
 * @Description: 一个线程写入,100 个线程读取
 */
```

```
* @author xialei
*
*/
public class ReadWriteLockDemo
{
    public static void main(String[] args) throws InterruptedException
    {
        MyQueue q = new MyQueue();

        new Thread(() -> {
            q.writeObj("ClassName1221");
        }, "AAAAA").start();

        for (int i = 1; i <=100; i++)
        {
            new Thread(() -> {
                q.readObj();
            },String.valueOf(i)).start();
        }

    }
}
```

## 2. CountdownLatch 减少计数

### 2.1 原理

CountDownLatch 主要有两个方法，当一个或多个线程调用 `await` 方法时，这些线程会阻塞。

其它线程调用 `countDown` 方法会将计数器减 1(调用 `countDown` 方法的线程不会阻塞)，当计数器的值变为 0 时，因 `await` 方法阻塞的线程会被唤醒，继续执行。

### 2.2 程序代码

```
package com.atguigu.thread;

import java.util.concurrent.CountDownLatch;

/**
 *
 * @Description:
 * * 让一些线程阻塞直到另一些线程完成一系列操作后才被唤醒。
 *
 * CountdownLatch 主要有两个方法，当一个或多个线程调用 await 方法时，这些线程会阻塞。
 * 其它线程调用 countDown 方法会将计数器减 1(调用 countDown 方法的线程不会阻塞)，
 * 当计数器的值变为 0 时，因 await 方法阻塞的线程会被唤醒，继续执行。
 *
 * 解释：6 个同学陆续离开教室后值班同学才可以关门。
 *
 * main 主线程必须要等前面 6 个线程完成全部工作后，自己才能开干
 * @author xialei
 */
```

```
public class CountdownLatchDemo
{
    public static void main(String[] args) throws InterruptedException
    {
        CountdownLatch countDownLatch = new CountdownLatch(6);

        for (int i = 1; i <= 6; i++) //6 个上自习的同学，各自离开教室的时间不一致
        {
            new Thread(() -> {
                System.out.println(Thread.currentThread().getName()+"\t 号同学离开教室");
                countDownLatch.countDown();
            }, String.valueOf(i)).start();
        }
        countDownLatch.await();
        System.out.println(Thread.currentThread().getName()+"\t***** 班长关门走人，main 线程是班长");
    }
}
```

## 3. CyclicBarrier 循环栅栏

### 3.1 原理

CyclicBarrier 的字面意思是可循环（Cyclic）使用的屏障（Barrier）。它要做的事情是，让一组线程到达一个屏障（也可以叫同步点）时被阻塞,直到最后一个线程到达屏障时，屏障才会开门，所有被屏障拦截的线程才会继续干活。线程进入屏障通过 CyclicBarrier 的 await()方法。

### 3.1 程序代码

```
package com.atguigu.thread;

import java.util.concurrent.BrokenBarrierException;
import java.util.concurrent.CyclicBarrier;

/**
 *
 * @Description: TODO(这里用一句话描述这个类的作用)
 * @author xialei
 *
 * CyclicBarrier
 * 的字面意思是可循环（Cyclic）使用的屏障（Barrier）。它要做的事情是，
 * 让一组线程到达一个屏障（也可以叫同步点）时被阻塞，
 * 直到最后一个线程到达屏障时，屏障才会开门，所有
 * 被屏障拦截的线程才会继续干活。
 * 线程进入屏障通过 CyclicBarrier 的 await()方法。
 *
 * 集齐 7 颗龙珠就可以召唤神龙
 */
public class CyclicBarrierDemo
{
    private static final int NUMBER = 7;
```



```
public static void main(String[] args)
{
    //CyclicBarrier(int parties, Runnable barrierAction)

    CyclicBarrier cyclicBarrier = new CyclicBarrier(NUMBER, ()->{System.out.println("*****集齐 7 颗龙珠就可以召唤神
龙");});

    for (int i = 1; i <= 7; i++) {
        new Thread(() -> {
            try {
                System.out.println(Thread.currentThread().getName()+"\t 星龙珠被收集 ");
                cyclicBarrier.await();
            } catch (InterruptedException | BrokenBarrierException e) {
                // TODO Auto-generated catch block
                e.printStackTrace();
            }
        }, String.valueOf(i)).start();
    }
}
}
```

## 4. Semaphore 信号灯

### 3.2 原理

在信号量上我们定义两种操作：

**acquire**（获取） 当一个线程调用 **acquire** 操作时，它要么通过成功获取信号量（信号量减 1），要么一直等下去，直到有线程释放信号量，或超时。

**release**（释放）实际上会将信号量的值加 1，然后唤醒等待的线程。信号量主要用于两个目的，一个是用于多个共享资源的互斥使用，另一个用于并发线程数的控制。

### 4.1 程序代码

```
package com.atguigu.thread;

import java.util.Random;
import java.util.concurrent.Semaphore;
import java.util.concurrent.TimeUnit;

/**
 *
 * @Description: TODO(这里用一句话描述这个类的作用)
 * @author xialei
 *
 * 在信号量上我们定义两种操作：
 *
 * acquire（获取） 当一个线程调用 acquire 操作时，它要么通过成功获取信号量（信号量减 1），
 *
 * 要么一直等下去，直到有线程释放信号量，或超时。
 *
 * release（释放）实际上会将信号量的值加 1，然后唤醒等待的线程。
 *
 * 信号量主要用于两个目的，一个是用于多个共享资源的互斥使用，另一个用于并发线程数的控制。
 */
```

```
*/  
public class SemaphoreDemo  
{  
    public static void main(String[] args)  
    {  
        Semaphore semaphore = new Semaphore(3);//模拟 3 个停车位  
  
        for (int i = 1; i <=6; i++) //模拟 6 部汽车  
        {  
            new Thread(() -> {  
                try  
                {  
                    semaphore.acquire();  
                    System.out.println(Thread.currentThread().getName()+"\t 抢到了车位");  
                    TimeUnit.SECONDS.sleep(new Random().nextInt(5));  
                    System.out.println(Thread.currentThread().getName()+"\t----- 离开");  
                } catch (InterruptedException e) {  
                    e.printStackTrace();  
                } finally {  
                    semaphore.release();  
                }  
            }, String.valueOf(i)).start();  
        }  
    }  
}
```

尚硅谷