# JUC

尚硅谷大数据

V1.0

# 第一章 JUC简介

在 Java 5.0 提供了 java.util.concurrent （简称 JUC ）包，在此包中增加了在并发编程中很常用的实用工具类，用于定义类似于线程的自定义子系统，包括线程池、异步 IO 和轻量级任务框架。提供可调的、灵活的线程池。还提供了设计用于多线程上下文中的 Collection 实现等。

# 第二章 多线程回顾

## 1. 线程和进程

**程序(program)**是为完成特定任务、用某种语言编写的一组指令的集合。即指一段静态的代码。

**进程(process)**是程序的一次执行过程，或是正在运行的一个程序。进程是一个动态过程，

即有它自身的产生、存在和消亡的过程。每个Java程序都有一个隐含的主程序，即main方法。

**线程(thread)**线程是进程内部的一条具体的执行路径。若一个程序可同一时间执行多个线程，

就是支持多线程的。

**总结：**程序是静态的，程序运行后变为一个进程，一个进程内部可以有多个线程同时执行。

进程是所有线程的集合，每一个线程是进程中的一条执行路径

## 2. 多线程的使用优势

①提高应用程序的响应。对图形化界面更有意义，可增强用户体验。

②提高计算机系统CPU的利用率

③改善程序结构。将既长又复杂的进程分为多个线程，独立运行，利于理解和修改

④使用线程可以耗时任务放到后台去处理，例如等待用户输入、文件读写和网络收发数据等。

## 3. 多线程的创建方式

### 3.1 继承Thread类

①定义子类继承Thread类。

②子类中重写Thread类中的run方法。

③创建Thread子类对象，即创建了线程对象。

④调用线程对象start方法启动线程，默认调用run方法。

注意：如果只是调用run方法，则此时会在调用该方法的线程中来执行，而不是另启动一个线程。

示例：

|  |
| --- |
| **package** com.atguigu.thread.main;  **class** Thread1 **extends** Thread{  @Override  **public** **void** run() {  **for** (**int** i = 0; i < 100; i++) {  **if** (i%2==0) {  System.***out***.println(getName()+":------>"+i);  }  }  }  }  **public** **class** TheadDemo1 {  **public** **static** **void** main(String[] args)  **new** Thread1().start();  **for** (**int** i = 0; i < 100; i++) {  **if** (i%2==1) {  System.***out***.println("main:------>"+i);  }  }  }  } |

### 3.2 实现Runable接口

①定义子类，实现Runnable接口。

②子类中重写Runnable接口中的run方法。

③通过Thread类含参构造器创建线程对象，将Runnable接口的子类对象作为实际参数传递给

      Thread类的构造方法中。

④调用Thread类的start方法启动线程，其最终调用Runnable子类接口的run方法。

|  |
| --- |
| **class** Thread2 **implements** Runnable{  @Override  **public** **void** run() {  **for** (**int** i = 0; i < 100; i++) {  **if** (i%2==0) {  System.***out***.println(Thread.*currentThread*().getName()+":------>"+i);  }  }  }  }  **public** **class** TheadDemo1 {  **public** **static** **void** main(String[] args) {  **new** Thread(**new** Thread2()).start();  **for** (**int** i = 0; i < 100; i++) {  **if** (i%2==1) {  System.***out***.println("main:------>"+i);  }  }  }  } |

两种方式的区别：

继承Thread:线程代码存放Thread子类run方法中。

实现Runnable：线程代码存在接口的子类的run方法中。

实现Runnable接口避免了单继承的局限性，多个线程可以共享同一个接口子类的对象，非常适合多个相同线程来处理同一份资源。

优先使用实现接口的方式!

### 3.4 使用匿名内部类创建线程

|  |
| --- |
| **public** **class** TheadDemo1 {  **public** **static** **void** main(String[] args) {  **new** Thread(**new** Runnable() {  @Override  **public** **void** run() {  **for** (**int** i = 0; i < 100; i++) {  **if** (i%2==0) {  System.***out***.println(Thread.*currentThread*().getName()+":------>"+i);  }  }  }  }).start();;  **for** (**int** i = 0; i < 100; i++) {  **if** (i%2==1) {  System.***out***.println("main:------>"+i);  }  }  }  } |

如果线程只需要创建一次，那么可以使用匿名内部类的方式创建。

### 3.5 使用Callable接口

Callable接口作为JDK1.5新增的接口，与使用Runnable相比其功能更强大些。

* 相比run()方法，可以有返回值
* 方法可以抛出异常
* 支持泛型的返回值
* 需要借助FutureTask类，比如获取返回结果。

Callable接口一般用于配合ExecutorService使用。

Future接口：



* 可以对具体Runnable、Callable任务的执行结果进行取消、查询是否完成、获取结果等。
* FutrueTask是Futrue接口的实现类
* FutureTask 同时实现了Runnable, Future接口。它既可以作为Runnable被线程执行，又可以作为Future得到Callable的返回值。
* 多个线程同时执行一个FutureTask，只要一个线程执行完毕，其他线程不会再执行其call()方法。

|  |
| --- |
| **class** Thread06 **implements** Callable<Integer>{  @Override  **public** Integer call() **throws** Exception {  **int** num=0;  **for** (; num < 50; num++) {  System.***out***.println(Thread.*currentThread*().getName()+"======》"+num);    }  **return** num;  }  }  **public** **class** CallableThread {  **public** **static** **void** main(String[] args) **throws** InterruptedException, ExecutionException {  Thread06 thread06 = **new** Thread06();  FutureTask<Integer> futureTask = **new** FutureTask<>(thread06);  Thread thread = **new** Thread(futureTask);  thread.start();  Integer integer = futureTask.get();  System.***out***.println(integer);  System.***out***.println("主线程结束！");  }  } |

注意： get()方法会阻塞当前线程！

### 3.6 使用线程池

经常创建和销毁、使用量特别大的资源，比如并发情况下的线程，对性能影响很大。因此提前创建好多个线程，放入线程池中，使用时直接获取，使用完放回池中。可以避免频繁创建销毁、实现重复利用。

优势：

* 提高响应速度（减少了创建新线程的时间）
* 降低资源消耗（重复利用线程池中线程，不需要每次都创建）
* 便于线程管理

相关属性举例：

corePoolSize：核心池的大小

maximumPoolSize：最大线程数

keepAliveTime：线程没有任务时最多保持多长时间后会终止

JDK 5.0起提供了ExecutorService 和 Executors来实现线程池。

**ExecutorService**：真正的线程池接口。常见子类ThreadPoolExecutor。

void execute(Runnable command) ：执行任务/命令，没有返回值，一般用来执行Runnable

<T> Future<T> submit(Callable<T> task)：执行任务，有返回值，一般又来执行Callable

void shutdown() ：关闭连接池

**Executors**：工具类、线程池的工厂类，用于创建并返回不同类型的线程池

Executors.newCachedThreadPool()： 创建一个可缓存线程池，如果线程池长度超过处理需要，可灵活回收空闲线程，若无可回收，则新建线程。

Executors.newFixedThreadPool(n); 创建一个可定长线程数的线程池，可控制最大并发数，超出的线程会在队列中等待分配。

Executors.newSingleThreadExecutor() ：创建一个只有一个线程的线程池，单一线程可以保证所有的任务按照指定的顺序执行。

Executors.newScheduledThreadPool(n)：创建一个线程池，它可安排在给定延迟后运行命令或者定期地执行。

|  |
| --- |
| **public** **static** **void** main(String[] args) **throws** InterruptedException, ExecutionException {  ExecutorService executorService = Executors.*newFixedThreadPool*(10);  executorService.submit(**new** Thread06());  executorService.shutdown();  } |

## 4. 线程的常用方法

|  |  |
| --- | --- |
| **Thread中常用api方法** | |
| start() | 启动线程； |
| currentThread() | 获取当前线程对象 |
| getID() | 获取当前线程ID |
| getName() | 获取当前线程名称，Thread-编号  该编号从0开始 |
| setName() | 设置当前线程的名字 |
| sleep(long mill) | 休眠线程 |
| stop（） | 停止线程 |
| yield（） | 释放cpu的操作 |
| join （） | 加塞，谁调join()，谁先执行，当前线程被阻塞，直到 join() 方法加入的 join 线程执行完为止。 |
| isAlive() | 判断线程是否还活着 |
| **常用线程构造函数** | |
| Thread（） | 创建一个新的 Thread 对象 |
| Thread（String name） | 创建一个新的 Thread对象，具有指定的 name正如其名。 |
| Thread（Runable r） | 创建一个新的 Thread对象 |
| Thread（Runable r, String name） | 创建一个新的 Thread对象 |

需要注意的是，如果同一个线程对象，执行两次start()方法，那么会报错java.lang.IllegalThreadStateException。

## 5. 线程的控制

线程的调度策略有两种，一种是基于时间片的调度策略，一种是抢占式的调度策略，抢占式的调度策略则是高优先级的线程会抢占CPU。

在java中，同优先级线程组成先进先出队列（先到先服务），使用时间片策略，对高优先级的线程，使用优先调度的抢占式策略。

## 6. 线程的优先级

线程的优先级控制：

* **MAX\_PRIORITY（10）**
* **MIN \_PRIORITY （1）**
* **NORM\_PRIORITY （5）**

常用方法：

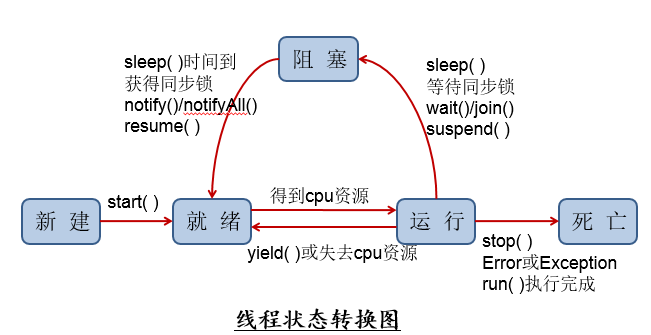
* **getPriority() ：**返回线程优先级
* **setPriority(int newPriority) ：**设置线程的优先级

子线程创建时继承父线程的优先级。

## 7. 线程的生命周期

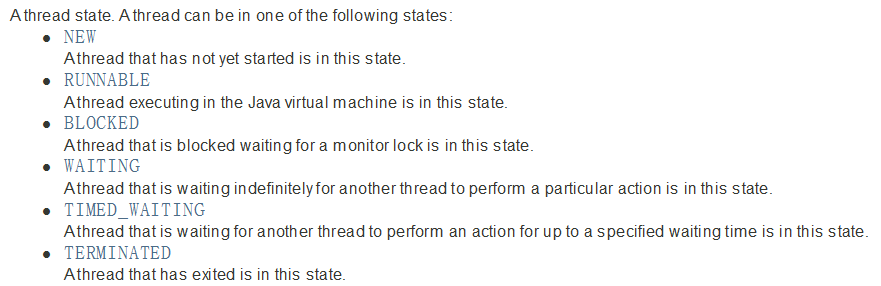
在一个完整的线程生命周期中通常要经历如下的五种状态：

* 新建： 当一个Thread类或其子类的对象被声明并创建时。
* 就绪： 处于新建状态的线程被start()后，将进入线程队列等待CPU时间片，此时它已具备了运行的条件。
* 运行： 当就绪的线程被调度并获得处理器资源时,便进入运行状态，run()方法定义了线程的操作和功能。
* 阻塞：在某种特殊情况下，被人为挂起或执行输入输出操作时，让出 CPU 并临时中止自己的执行，进入阻塞状态。
* 死亡：线程完成了它的全部工作或线程被提前强制性地中止 。



## 8. 线程的状态

JDK中用Thread.State枚举表示了线程的几种状态：



## 9. 线程的分类

Java中的线程分为两类：一种是守护线程（后台线程），一种是用户线程（前台线程）。它们在几乎每个方面都是相同的，唯一的区别是判断JVM何时离开。

守护线程是用来服务用户线程的，通过在start()方法前调用thread.setDaemon(true)可以把一个用户线程变成一个守护线程。当主线程不存在或主线程停止时，守护线程也会停止。

Java垃圾回收就是一个典型的守护线程。若JVM中都是守护线程，当前JVM将退出。

## 10. 线程的停止

* + 使用退出标志，使线程正常退出，也就是当run方法完成后线程终止。
  + 使用stop方法强行终止线程（已过时）。
  + 使用interrupt方法中断线程。

# 第三章 线程的安全问题

## 1. 线程安全

### 1.1 什么是线程安全

当多个线程同时共享，同一个全局变量或静态变量，做写的操作时，可能会发生数据冲突问题，也就是线程安全问题。但是做读操作是不会发生数据冲突问题。

### 1.2 线程安全的解决方式

使用多线程之间同步或使用锁(lock)可以解决线程安全问题。其核心在于将可能会发生数据冲突问题(线程不安全问题)，只能让当前一个线程进行执行。代码执行完成后释放锁，让后才能让其他线程进行执行。这样的话就可以解决线程不安全问题。

### 1.3 什么是多线程之间的同步

多个线程共享同一个资源的环境下,每个线程工作时不会受到其他线程的干扰称之为多线程之间的同步。

## 2. 解决线程安全

### 2.1 使用同步代码块

synchronized(同一个对象){

可能会发生线程冲突问题

}

注意：在同步代码块中，多个线程必须使用的是同一把锁，即同一个对象。

一般情况下，在使用Runnable实现的线程类中，我们会使用this作为锁对象。

在使用Thread继承的线程类中，一般会使用其Class对象（Class对象在JVM中只会创建一次）。

### 2.2 使用同步方法

使用synchronized修饰的方法称为同步方法。

|  |
| --- |
| **private** **synchronized** **boolean** sell()  **if** (tickets > 0) {  System.***out***.println(Thread.*currentThread*().getName()+"售出："+tickets+" 号车票！");  tickets--;  **return** **true**;  }**else** {  **return** **false**;  }    } |

注意：如果使用Thread继承的方式实现多线程，那么同步方法需要是一个静态的方法！

### 2.3 常见问题

①什么是静态同步函数，它使用什么锁？

使用static和synchronized 关键字同时修饰的函数为静态同步函数。静态同步函数使用的锁是该函数所属字节码文件对象，即类.class。

②同步代码块与同步函数区别？

同步代码使用自定义锁(明锁)，同步函数使用this锁。

③什么是多线程中的死锁？请举例说明？

死锁指不同的线程分别占用对方需要的同步资源不放弃，都在等待对方放弃自己需要的同步资源，就形成了线程的死锁。出现死锁后，不会出现异常，不会出现提示，只是所有的线程都处于阻塞状态，无法继续。

死锁一般在同步代码块中嵌套同步代码块时出现。

死锁示例：

|  |
| --- |
| **public** **class** DeadLock {  **private** **static** StringBuffer *s1*=**new** StringBuffer();  **private** **static** StringBuffer *s2*=**new** StringBuffer();  **public** **static** **void** main(String[] args) {  **new** Thread(**new** Runnable() {  @Override  **public** **void** run() {  **synchronized** (*s1*) {  *s1*.append("a");  *s2*.append(1);  **try** {  Thread.*sleep*(100);  } **catch** (InterruptedException e) {  e.printStackTrace();  }  **synchronized** (*s2*) {  *s1*.append("b");  *s2*.append(2);  System.***out***.println(*s1*);  System.***out***.println(*s2*);  }  }  }  }).start();    **new** Thread(**new** Runnable() {  @Override  **public** **void** run() {    **synchronized** (*s2*) {  *s1*.append("c");  *s2*.append(3);  **try** {  Thread.*sleep*(100);  } **catch** (InterruptedException e) {  e.printStackTrace();  }  **synchronized** (*s1*) {  *s1*.append("d");  *s2*.append(4);  System.***out***.println(*s1*);  System.***out***.println(*s2*);  }  }  }  }).start();  }  } |

在多线程编程中，尽量避免同步代码块的嵌套，避免使用同样的锁来避免死锁。

### 2.4 同步锁的作用域

不同线程访问同一个对象的不同同步方法，那么使用的锁都是这个对象，因此这个对象的所有同步方法共同使用同一把锁。

不同线程访问同一个类的多个静态同步方法，那么使用的锁都是这个对象的类对象，因此这个对象的所有静态同步方法共同使用同一把锁。

同步方法和普通方法互不干扰！

### 2.5 使用Lock解决线程安全

从JDK 5.0开始，Java提供了更强大的线程同步机制——通过显式定义同步锁对象来实现同步。同步锁使用Lock对象充当。java.util.concurrent.locks.Lock接口是控制多个线程对共享资源进行访问的工具。锁提供了对共享资源的独占访问，每次只能有一个线程对Lock对象加锁，线程开始访问共享资源之前应先获得Lock对象。

ReentrantLock 类实现了 Lock ，它拥有与 synchronized 相同的并发性和内存语义，在实现线程安全的控制中，比较常用的是ReentrantLock，可以显式加锁、释放锁。

使用示例：

|  |
| --- |
| **class** Thread1 **implements** Runnable{  **private** ReentrantLock lock =**new** ReentrantLock(**true**);  @Override  **public** **void** run() {  **try** {  lock.lock();  **//操作资源**  } **finally** {  lock.unlock();  }  }  } |

相比synchronized的由系统自动获取锁和释放锁，Lock需要自己手动实现加锁和释放锁！因此也更加灵活！

# 第四章 线程的通信

## 1. 什么是线程的通信

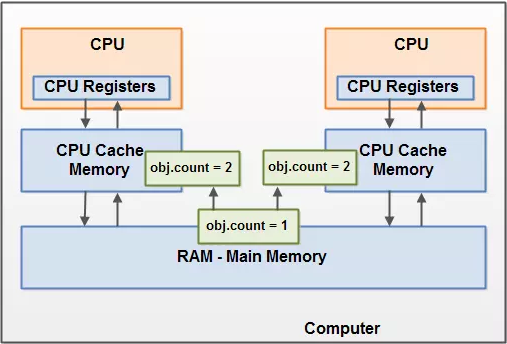
线程的通信是指线程之间以何种机制来交换信息。在编程中，线程之间的通信机制有两种，共享内存和消息传递。

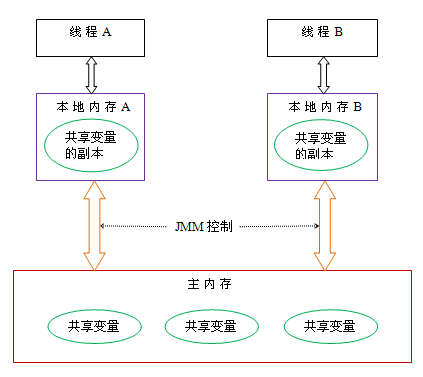
在共享内存的并发模型里，线程之间共享程序的公共状态，线程之间通过写-读内存中的公共状态来隐式进行通信，典型的共享内存通信方式就是通过共享对象进行通信。

在消息传递的并发模型里，如果线程之间没有公共状态，线程之间必须通过明确的发送消息来显式进行通信，例如使用 wait()和notify()。

## 2. Volatile

JMM定义了Java 虚拟机(JVM)在计算机内存(RAM)中的工作方式。JVM是整个计算机虚拟模型，所以JMM是隶属于JVM的。从抽象的角度来看，JMM定义了线程和主内存之间的抽象关系：线程之间的共享变量存储在主内存（Main Memory）中，每个线程都有一个私有的本地内存（Local Memory），本地内存中存储了该线程以读/写共享变量的副本。本地内存是JMM的一个抽象概念，并不真实存在。它涵盖了缓存、写缓冲区、寄存器以及其他的硬件和编译器优化。





从上图来看，线程A与线程B之间如要通信的话，必须要经历下面2个步骤：首先，线程A把本地内存A中更新过的共享变量刷新到主内存中去。然后，线程B到主内存中去读取线程A之前已更新过的共享变量。

但是如果线程A和B是并发操作的，那么就会出现数据不安全。例如：

|  |
| --- |
| **class** Thread01 **extends** Thread {  **public** **boolean** flag = **true**;  @Override  **public** **void** run() {  System.***out***.println("开始执行子线程....");  **while** (flag) {  }  System.***out***.println("线程停止");  }  **public** **void** setRuning(**boolean** flag) {  **this**.flag = flag;  }  }  **public** **class** VolatileDemo {  **public** **static** **void** main(String[] args) **throws** InterruptedException {  Thread01 thread01 = **new** Thread01();  thread01.setName("线程1");  thread01.start();  Thread.*sleep*(3000);  thread01.setRuning(**false**);  System.***out***.println("flag 已经设置成false");  Thread.*sleep*(1000);  System.***out***.println(thread01.flag);  }  } |

结果：当在主线程中设置了thread01对象的flag属性时，分线程并未立刻停止！

原因：线程之间是不可见的，分线程读取的是副本，而没有及时读取到主内存已经修改的结果。

解决：使用Volatile关键字将解决线程之间可见性, 强制线程每次读取该值的时候都去“主内存”中取值。

Volatile虽然可以保证线程之间的可见性，但是并不能保证一些变量操作的原子性，例如：

|  |
| --- |
| **class** Thread2 **extends** Thread{  **private** **static** **volatile** **int** *count*;  @Override  **public** **void** run() {  **for** (**int** i = 0; i < 100; i++) {  *count*++;  }  System.***out***.println(getName()+"----->"+*count*);  }  }  **public** **class** VolatileDemo2 {  **public** **static** **void** main(String[] args) {  **for** (**int** i = 0; i < 10; i++) {  Thread2 thread2 = **new** Thread2();  thread2.start();  }  }  } |

解决： 使用AtomicInteger的incrementAndGet()方法解决。

## 3. 常用的线程通信方法

wait()：令当前线程挂起并放弃CPU、同步资源并等待，使别的线程可访问并修改共享资源，而当前线程排队等候其他线程调用notify()或notifyAll()方法唤醒，唤醒后等待重新获得对监视器的所有权后才能继续执行。

notify()：唤醒正在排队等待同步资源的线程中优先级最高者结束等待。

notifyAll ()：唤醒正在排队等待资源的所有线程结束等待。

注意：

①这三个方法只有在synchronized方法或synchronized代码块中才能使用，否则会报java.lang.IllegalMonitorStateException异常。也就是说只有在已经获取锁的时候，才可以使用！

②这三个方法的调用者必须是同步代码块或同步方法中的同步监视器。

③这三个方法是在Object类中已经定义的。

面试题：sleep()和wait()的异同？

同：这两个方法都可以使线程进入阻塞状态。

异：① 两个方法声明的位置不同，sleep()是Thread类中声明的，而wait()是Object类中声明的。

② 调用的要求不同，sleep()可以在任何场景调用，而wait()只能在同步代码块或同步方法中使用。

③ sleep()不会释放同步锁，wait()会释放锁。

## 4. 使用Lock后线程通信

Condition的功能类似于在传统的线程技术中的,Object.wait()和Object.notify()的功能。

Condition.await()类似于wait.。

Condition.signal()类似于notify。

|  |
| --- |
| **class** Thread01 **implements** Runnable{  **private** **int** num=1;  **private** ReentrantLock lock=**new** ReentrantLock(**true**);  **private** Condition condition=lock.newCondition();  @Override  **public** **void** run() {  **while**(**true**){  **try** {  lock.lock();  condition.signal();  **if** (num<=100) {  System.***out***.println(Thread.*currentThread*().getName()+"---->"+num);  num++;  **try** {  condition.await();  } **catch** (InterruptedException e1) {  e1.printStackTrace();  }  }**else** {  **break**;  }  } **finally** {  lock.unlock();  }  }  }  } |

# 第五章 JUC工具类

## 1. ReentrantReadWriteLock

现实中有这样一种场景：对共享资源有读和写的操作，且写操作没有读操作那么频繁。在没有写操作的时候，多个线程同时读一个资源没有任何问题，所以应该允许多个线程同时读取共享资源；但是如果一个线程想去写这些共享资源，就不应该允许其他线程对该资源进行读和写的操作了。

针对这种场景，JAVA的并发包提供了读写锁ReentrantReadWriteLock，它表示两个锁，一个是读操作相关的锁，称为共享锁；一个是写相关的锁，称为排他锁。

当没有其他线程的写锁时，线程进入读锁。当没有其他线程的读锁和写锁时，才会进入当前线程的写锁！

举例：看电影。

## 2. CountDownLatch

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。

举例： 促销活动。

## 3. CyclicBarrier

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。

举例： 军训阅兵。

## 4. Semaphore

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。

举例： 饭店等位。

## 5. 锁的总结

①特质上分：共享锁和排他锁

②用途上分： 读锁和写锁

③数据库： 表锁和行锁

④世界观划分： 悲观锁（真锁）和乐观锁（假锁）

⑤是否明显： 显式锁和隐式锁