

Java并发问题

1. 什么是原子操作？在Java Concurrency API中有哪些原子类(atomic classes)？

原子操作是指一个不受其他操作影响的操作任务单元。原子操作是在多线程环境下避免数据不一致必须的手段。

int++并不是一个原子操作，所以当一个线程读取它的值并加1时，另外一个线程有可能会读到之前的值，这就会引发错误。

为了解决这个问题，必须保证增加操作是原子的，在JDK1.5之前我们可以使用同步技术来做到这一点。到JDK1.5，java.util.concurrent.atomic包提供了int和long类型的装类，它们可以自动的保证对于他们的操作是原子的并且不需要使用同步。可以阅读这篇文章来了解Java的atomic类。

2. Java Concurrency API中的Lock接口(Lock interface)是什么？对比同步它有什么优势？

Lock接口比同步方法和同步块提供了更具扩展性的锁操作。他们允许更灵活的结构，可以具有完全不同的性质，并且可以支持多个相关类的条件对象。

它的优势有：

可以使锁更公平

可以使线程在等待锁的时候响应中断

可以让线程尝试获取锁，并在无法获取锁的时候立即返回或者等待一段时间

可以在不同的范围，以不同的顺序获取和释放锁

阅读更多关于锁的例子

3. 什么是Executors框架？

Executor框架同java.util.concurrent.Executor 接口在Java 5中被引入。Executor框架是一个根据一组执行策略调用，调度，执行和控制的异步任务的框架。

无限制的创建线程会引起应用程序内存溢出。所以创建一个线程池是个更好的的解决方案，因为可以限制线程的数量并且可以回收再利用这些线程。利用Executors框架可以非常方便的创建一个线程池，阅读这篇文章可以了解如何使用Executor框架创建一个线程池。

4. 什么是阻塞队列？如何使用阻塞队列来实现生产者-消费者模型？

java.util.concurrent.BlockingQueue的特性是：当队列是空的时，从队列中获取或删除元素的操作将会被阻塞，或者当队列是满时，往队列里添加元素的操作会被阻塞。

阻塞队列不接受空值，当你尝试向队列中添加空值的时候，它会抛出NullPointerException。

阻塞队列的实现都是线程安全的，所有的查询方法都是原子的并且使用了内部锁或者其他形式的并发控制。

BlockingQueue 接口是java collections框架的一部分，它主要用于实现生产者-消费者问题。

阅读这篇文章了解如何使用阻塞队列实现生产者-消费者问题。

5. 什么是Callable和Future?

Java 5在concurrency包中引入了java.util.concurrent.Callable 接口，它和Runnable接口很相似，但它可以返回一个对象或者抛出一个异常。

Callable接口使用泛型去定义它的返回类型。Executors类提供了一些有用的方法去在线程池中执行Callable内的任务。由于Callable任务是并行的，我们必须等待它返回的结果。java.util.concurrent.Future对象为我们解决了这个问题。在线程池提交Callable任务后返回了一个Future对象，使用它我们可以知道Callable任务的状态和得到Callable返回的执行结果。Future提供了get()方法让我们可以等待Callable结束并获取它的执行结果。

阅读这篇文章了解更多关于Callable，Future的例子。

6. 什么是FutureTask?

FutureTask是Future的一个基础实现，我们可以将它同Executors使用处理异步任务。通常我们不需要使用FutureTask类，单当我们打算重写Future接口的一些方法并保持原来基础的实现是，它就变得非常有用。我们可以仅仅继承于它并重写我们需要的方法。阅读Java FutureTask例子，学习如何使用它。

7.什么是并发容器的实现？

Java集合类都是快速失败的，这就意味着当集合被改变且一个线程在使用迭代器遍历集合的时候，迭代器的next()方法将抛出ConcurrentModificationException异常。

并发容器支持并发的遍历和并发的更新。

主要的类有ConcurrentHashMap, CopyOnWriteArrayList 和CopyOnWriteArraySet，阅读这篇文章了解如何避免ConcurrentModificationException。

8. Executors类是什么？

Executors为Executor，ExecutorService，ScheduledExecutorService，ThreadFactory和Callable类提供了一些工具方法。

Executors可以用于方便的创建线程池。

//////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////

/////////////////////////////////////////////////////////////////////////////////////////////////////////////

Java 中的编译期常量是什么？使用它有什么风险？

类加载过程有三个，加载、连接、初始化。

thread-斯

currentThread-

Gc回收的是堆内存，不是常量池内存，也不是栈内存

使用继承Thread类的方式创建新线程时，最大的局限是不支持多继承，Thread的构造可以传Runnable对象，也可以传Thread对象，Thread实现了Runnable接口

在主线程中直接调用代码thread.run()是在主线程中执行的，不是异步，执行start()方法的顺序不代表线程的启动顺序

在多数的jvm中，i--,为非原子性操作，要分成三步，

1，取得原有的值， 2，计算i-1, 3，对进行赋值

在这三个步骤中，如果有多个线程同时访问，那么一定会出现线程安全的问题

虽然print()方法在内部是同步的，但i++的操作却是在进人print()之前发生的，所以有发生非线程安全问题的概率

synchronized可以在任意对象和方法加锁。而加锁的代码称为“互斥区”或“临界区”

当一个线程想要执行同步方法里面的代码时，线程首先尝试同步去拿这把锁，如果能够拿到这把锁，那么这个线程就可以执行synchronized里面的代码。如果不能拿到这把锁，那么这个线程就会不断地尝试拿这把锁，直到能够拿到为止，而且是有多个线程同时去争抢这把锁非线程安全主要是指多个线程对同一个对象中的同一个实例变量进行操作时会出现值被更改、值不同步的情况，进而影响程序执行流程

currentThread()方法返回代码段正在执行的线程的信息。线程的名称可以相同

方法isAlive()的作用是测试线程是否处下活动状态。什么是活动状态呢?活动状态就是线程已经启动且尚未终止，线程处于正在运行或准备开始运行的状态，就认为线程是”存活状态”，sleep时时在此中么

Thread.currentThread()和this.currentThread()是有区别的

在java中有三种方法可以终止正在运行的线程

1，使用退出标志，使线程正常退出，也就是run方法完成后线程终止。

2，使用stop()方法强行终止线程，但是不推荐使用这个方法，因为stop和suspend及resume一样。都是作废过期的方法，使用它们可能产生不可预料的结果。

3，使用interrupt方法终止线程

this.interrupted()测试当前线程是否已经是中断状态，执行后具有将状态标志置清除为false的功能。

this.islnterrupted()测试线程Thread对象是否已经是中断状态，但不清除状态标志。

如果在sleep状态下停止某一个线程，会进入catch语句，并且清除停止状态，使之变成false

方法stop()已经被作废，因为如果强制让线程停止可能使一些清理行、性的工作得不到完成。另外一个情况就是对锁定的对象进行了“解锁”，导致数据得不到同步的处理，出现数据不一致的问题。

由于stop()方法已经在jdk中被标明是“作废、过期”的方法，显然他在功能上具有缺陷，所以不建议在程序中使用stop()方法

不过还是建议使用抛异常的方式来使线程停止，因为在catch块中还可以使异常向上抛，使异常得以传播。

suspend与resume方法的缺点，独占，不同步

yield方法的作用是放弃当前的cpu资源，将它让给其他的任务去占用cpu执行时间，但放弃的时间不确定，有可能刚刚放弃，马上又获得cpu的时间片

在操作系统中，线程可以划分优先级，优先级较高的线程得到的cpu的资源较多，也就是cpu优先执行优先级较高的线程对象中的任务，设置线程优先级有助于帮“线程规划器”确定在下一次选择哪一个线程来优先执行。

在java中，线程的优先级分为1-10这10个等级，如果小于1或者大于10，则jdk抛出异常 throw new IllehalArgumentException()

在java中，线程的优先级具有“继承性”，比如A线城启动B线程，则B线程的优先级与A线按的优先级一致。

高优先级的线程总是大部分先执行完，但并不代表高优先级的线程全部先执行完，另外，不要以为MyThread1线程先被main线程所调用就会先执行完，出现这样的结果全是因为MyThread1线程的优先级是最高值为10造成的，当线程优先级的等级差距很大时，谁先完成和代码的调用顺序无关，

cpu尽量将执行资源让给优先级比较高的线程，

优先级具有随机性

优先级高的运行的快，运行的多

在java线程中有两种线程，一种是用户线程，一种是守护线程，守护线程是一种特殊的线程，它的特性有“

陪伴”的含义，当进程中没有了非守护线程，则守护线程自动销毁。典型的守护线程就是垃圾回收线程，当进程中没有非守护线程了，则垃圾回收线程也就没有必要存在了，自动销毁，用个比较通俗的比喻来解释一下“守护线程”：任何一个守护线程都是jvm中所有非守护线程的保姆，只要当前的jvm实例中存在一个非守护线程没有结束，守护线程就在工作，只有当最后一个非守护线程的运行结束时，守护线程才随着jvm一同结束工作，Daemon的作用是为其他线程的运行提供便利服务。

“脏读”也就是读取到的数据是被更改过的

“非线程安全”问题存在实例变量中

在两个线程访问同一个对象的同步方法时一定是线程安全的

关键字synchronized取到的锁是对象锁，而不是把一段代码或方法当做锁，所以在上面的实例中，那个线程先执行带synchronized关键字的方法，那个线程就持有该方法所属对象的锁Lock,那么其他线程只能呈现等待状态，前提是多个线程访问同一个对象

同步单词是synchronized。异步的单词是asynchronized

调用synchronized声明的方法一定是排队运行的，另外需要牢牢记住“共享”这两个字，只有共享资源的读写访问才需要同步化，如果不是共享资源，那么根本就没有同步的必要。

A线城先持有object对象的Look锁，B线程可以以异步的方式调用object对象中的非synchronized类型的方法。

A线城先持有object对象的Look锁，B线程如果在这时调用object对象中的synchronized类型的方法则需要等待，也就是同步。

synchronized方法/块的内部调用本类的其他synchronized方法/块时，时可以永远得到锁的。

“可重入锁”的概念是，自己可以再次获取自己的内部锁，不如有一个线程获得了某个对象的锁，此时这个对象的锁还没有释放，当其再次想要获得这个对象的锁的时候是可以获取的，如果不可锁重入的话，机会在成死锁，

可重入锁支持在父子类继承的环境中。当存在父子类继承关系时子类完全可以通过“可重入锁”调用父类的同步方法

当一个线程执行的代码出现异常时，其所持有的锁会自动释放

不同不能继承，所以还得再子类的方法中添加synchronized的声明

当两个并发线程同时访问同一个对象object中的synchronized(this)同步代码时，一段时间内只能有一个线程被执行，另一个线程必须等待当前线程执行完成这个代码块以后才能执行该代码块，

不在synchronized块中及时异步执行，在synchronized块中就是同步执行

在使用同步synchronized(this)代码时需要注意的是，当一个线程访问一个synchronized(this)同步代码时，其他线程对同一个object中所有其他的synchronized(this)同步代码块的访问将被阻塞，这也说明synchronized使用的“对象监视器”是一个。

和synchronized方法一样，synchronized(this)代码块也是锁定当前对象的

多个线程调用同一个对象中不同名称的synchronized的同步方法或synchronized(this)同步代码块，调用的效果就是按顺序执行，也就是同步的，阻塞的。

但如果使用同步代码块锁非this对象，则synchronized(非this)代码块中的程序与同步方法时异步的，捕鱼其他锁this同步方法挣抢this锁，则可以大大提高运行的效率。

同步代码块放在非同步synchronized方法中进行声明，并不能保证调用方法的线程执行同步、顺序性，也就是线程调用方法的顺序是无序的，虽然在同步块中执行的顺序是同步的，这样极易出现“脏读”问题。

synchronized(非this对象x)格式的写法是将x对象本身作为“对象监视器”，这样有3个结论，

1，当多个下城同时执行synchronized(x)同步代码块时呈现同步效果。

2，当其他线程执行x对象中的synchronized同步方法和synchronized(this)代码块，时呈现同步效果。

synchronized还可以应用在静态的方法上面，如果这样写，那就是对当前的.java文件对应的class类进行持锁。而synchronized加到非静态方法上是给对象加锁

Runnable的run方法可以加synchronized，

在jvm中具有string常量缓存的功能

出现这样的情况就是因为String的两个值都是AA，两个线程持有相同的锁，所以造成线程B不能执行，这就是String常量池多带来的问题，因此在大多数情况下，同步代码都不使用String作为锁对象，而改用其他，比如new Object（）实例化一个Object对象，但它并不放在缓存中。

java线程死锁是一个经典的多线程问题，因为不同的线程都在等待根本不可能释放的锁，从而导致所有的任务都无法继续完成，在多线程技术中，“死锁”是必须避免的，因为这样会造成线程的“假死”

可以使用jdk自带的工具来检测是否有死锁现象，首先进入cmd工具，在进入jdk的安装文件夹的bin目录，执行jps命令。

死锁是程序设计的bug，在设计程序是就要避免双方相互持有对方的锁的情况，只要互相等待对方释放锁就有可能出现死锁。

线程A和线程B持有的锁都是“123”，虽然将锁改成了“456”，但结果还是同步的，因为A和B共同争抢的锁是“123”，

只要对象不变，即使对象得到属性改变了，运行的结果还是同步的，string看的是内容。

关键字volatile的主要作用是使变量在多个线程见可见，其强制从公共堆栈中取得变量的值，而不是从线程私有的数据栈(cup缓存,连续操作不会更新，中断时可能会更新，cpu，4核心，256m，理论同时跑4个线程，每个线程的缓存是64m)中取得变量的值。

在jvm被设为-service 模式时为了线程运行的效率，线程一直在私有堆栈中取得变量的值，而其他线程设置变量的值设置的是公共堆栈中的变量的值

这个问题其实就是私有堆栈中的值和公共堆栈中的值不同步造成的，解决这样的问题就要使用volatile关键字了，他主要的作用就是当前线程访问的数据，强制性从公共堆栈中进行取值。

关键字volatile是线程同步的轻量化实现，所以violate的性能肯定比synchronized要好，并且volatile只能修饰变量，而synchronized可以修饰方法，代码块，随之jdk新版本的发布，synchronized的执行效率得到了很大提升，在开发中使用synchronized的比率还是较大的，

多线程访问volatile不会发生阻塞，而synchronized会出现阻塞。

volatile能保证数据的可见行，但不能保证原子性，而synchronized可以保证原子性，也可以间接保证可见行，因为它会将私有内存和公共内存中的数据做同步，

volatile解决的是变量在多个线程见的可见行，而synchronized解决的是多个线程间访问资源的同步，线程安全包含原子性和可见性，java的同步机制都是围绕这两个方面来确保线程安全的。

如果使用synchronized来声明方法，也就没有必要使用volatile来声明其中使用的变量了。

volatile声明变量，jvm虚拟机这是保证从祝内存加载到线程工作内存的值是新的，就是说解决的是“读”的可见性，但不保证原子性，对于多个多个线程访问同一个实例变量还是需要加锁同步，

原子操作是不能分割的整体，没有其他线程能够中断或检查正在原子操作中的变量，一个原子类型就是原子操作可用的类型，它可以在没有锁的情况下做到线程安全。

方法和方法间的调用却不是原子性的，

多线程着重“外练互斥”，内修可见

//--------

方法wait()的作用是当前执行代码的线程进行等待，wait方法是Object类的方法，该方法用来将当前线程置入“预备执行队列”中，并且在wait所在的代码行处停止执行，直到接到通知或被中断为止，在调用wait方法之前，线程必须获得该对象的对象级别锁，即只能在同步方法或同步代码块中调用wait方法，在执行wait方法后，当前线程释放锁，在从wait返回前，线程与其他线程竞争重新获得锁，如果调用wait时没有所得适当锁，则抛出异常是一个运行时异常，不需要try-catch语句进行捕捉异常，

方法notify也要在同步方法或同步代码块中调用，即在调用前，线程也必须获得该对象的对象级别的锁，该方法用来通知那些可能等待该对象的对象锁的其他线程，如果有多个线程等待，则由线程规划器随机挑选出其中一个呈wait状态的线程，对其发出通知notify，病使他等待获取对象的对象锁，需要说明的是，在执行notify方法后，当前的线程不会马上释放该对象锁，呈wait状态的线程也并不能马上获得该对象锁，要等到执行notify方法的线程将程序执行完，也就是退出synchronized代码块后，当前线程才会释放，而呈wait状态所在的线程才可以获得对象锁，当第一个或得了该对象锁的wait线程运行完毕以后，他会释放该对象锁，此时如果该对象没有再次使用notify语句，则即使该对象已经空闲了，其他wait状态等待的线程由于没有得到该对象的通知，还是会继续阻塞在wait状态，直到这个对象发出一个notify或notifyAll。

一言以蔽之，wait使线程停止运行，而notify使停止运行的线程继续运行。

notify操作可以唤醒一个因调用了wait操作而处于阻塞状态的线程，使其进入就绪状态，如果发出去的notify操作时没有处于阻塞状态中的线程，那么该线程就被忽略，

notifyAll方法可以使所有正在等待队列中等待同一个资源的“全部”线程从等待状态退出，进入可运行状态，此时，优先级最高的那个线程最先执行，但也有可能是随机执行的，因为这要取决于JVM虚拟机的实现。

每个锁对象都有连个队列，一个是阻塞队列，一个是就绪队列，就绪队列存储了将要获得锁的线程，阻塞队列存储了被阻塞的线程，一个线程被唤醒后，才会进入就绪队列，等待cpu的调度，一个线程被wait后，就会进入阻塞队列，等待下一次被唤醒。

当方法wait被执行后，锁自动释放，但执行完notify方法，锁却不一定自动释放，必须执行完notify方法所在的同步代码后才释放锁。

1，执行完同步代码块会释放对象锁

2，在执行同步代码块的过程中，遇到了异常导致线程终止，锁也会被释放。

3，在执行同步代码块过程中，执行了所对象的wait方法，这个线程会释放对象锁，而此线程对象会进入线程池中，等待被唤醒。

带一个参数的wait方法的功能是等待某一时间内是否有线程对象锁进行唤醒，如果超过这个时间则自动唤醒。

如果通知过早，则会打乱程序的正常运行逻辑。

假死出现的主要原因是有可能连续唤醒同类，解决这样的问题是改notify为notifyAll来使用就可，他的原理就是不光通知同类线程，也包括异类线程，这样就不至于出现假死的状态了，程序会一直的执行下去。

join方法的租用是等待线程对象销毁

方法join的作用是使所属的线程对象x正常执行run方法中的任务，而是当前的线程z进行无限期的阻塞，等待线程x销毁后再继续执行z后面的代码，

方法join具有使线程排队运行的作用，有些类似同步的运行效果，join与synchronized的区别是，join在内部使用wait方法进行等待，而synchronized关键字使用的是“对象监视器”原理作为同步。

带参数的join中的参数是设定的等待时间

方法join(long)d的功能在内部是使用wait(long)方法来实现的，所以join(long)方法具有释放锁的特点，而Thread。Sleep(long)方法却不释放锁。

ThreadLocal主要解决的就是每个线程绑定自己的值，可以将ThreadLocal比喻成全局存放数据的盒子，盒子中可以存储每个线程的私有数据。

ThreadLocal解决的是变量在不同线程间的隔离性，也就是不同线程拥有自己的值，不同的线程中的值是可以放入ThreadLocal中进行保存的。覆盖initialValue()方法具有初始值，提供get()的首次返回值，

InheritableThreadLocal可以在子线程中取得父线程继承下来的值，

但注意，如果子线程取得值的同时，主线程将InheritableThreadLocal中的值进行更改，那么子线程取得到的值还是旧值。

调用ReentratLock对象的lock（）方法获取锁，调用unlock()方法释放锁。

调用lock。Lock()代码的线程就持有了“对象监视器”，其他线程只有等待锁被释放时再次挣枪，效果和使用synchronized关键字一样，线程之间还是顺序执行的。

Condition类是jdk5中出现的技术，使用它有更好的灵活性，比如可以实现多路通知功能，也就是在一个Lock对象里面可以创建多个Condition(即对象监视器)实例，线程对象可以注册在指定的Condition中，从而可以有选择性的进行线程通知，在调度线程上更加的灵活，

在使用notify和notifyAll方法进行通知时，被通知的线程时jvm随机选择的，但是使用ReentrantLock结合Condition类是可以实现“选择性通知”的，这个功能是非常重要的，而且在Condition类中是默认提供的。

而synchronized就相当于整个Lock对象中只有一个单一的Condition对象，所有线程都注册在它的一个对象上，线程开始notifyAll时，需要通知所有的waitting的线程，没有选择权，会出现相当大的效率问题。

报错的异常信息是监视器出错，解决方法就是必须在condition。await方法调用之前用lock。lock方法获取同步监视器。

Object类中wait方法相当于Condition类中的await方法。

Object类中的waiting(long timeout)方法相当于Condition类中的await(long time,TimeUnit unit)方法。

Object类中的notify方法相当于Condition类中的signal方法

Object类中的notifyAll方法相当于Condition类中的signalAll方法。

Condition对象可以唤醒指定线程，尤祖宇提升程序的效率，可以新对线程进行分组，然后再唤醒指定组中的线程。

使用ReentrantLock对象可以唤醒指定种类的线程，这是控制部分线程行为的方便方式。

notifyAll的解决方案可以使用signalAll方法来解决

锁Lock分为公平锁和费非公平锁，公平锁表示线程获取锁的顺序是按照线程加锁的顺序来分配的，即先来先得的FIFO先进先出的顺序，而非公平锁就是一种获取锁的抢占机制，是随机获得锁，和公平锁不一样就是先来的不一定先得到所，这个方式可能造成某些线程一直拿不到锁，结果也就是不公平了。非公平锁的运行结果基本上是乱序的，说明先start启动的线程不代表先获得锁。

lock的getHoldCount方法的作用是查询当前线程保持此锁的个数，也就是调用lock方法的次数。

lock的getQueueLength方法的作用是返回正等待获取次锁定的线程的估计数，比如有5个线程，1个线程首先执行了await方法，那么在调用getQueueLength方法返回的值是4，说明有4个线程同时在等待lock的释放。

lock的getWaitQueueLengthCondition方法的作用是返回等待与此锁相关的给定条件的condition的线程的估计数量。

lock的hasQueueThread方法的作用是查询指定的线程是否在等待获取此锁定。

lock的hasQueueThreads方法的作用是查询是否有线程正在等待获取此锁定。

lock的hasWaiters的作用是查询是否有线程正在等待与此锁定有关的condition条件。

lock的isFair方法的作用是判断是不是公平锁。早默认情况下，ReentrantLock类使用的是非公平锁。

lock的isHeldByCurrentThread的方法的作用是查询当前线程是否保持此锁定。

lock的isLocked方法的作用是查询此锁定是否由任意线程保持。

lock的lockInterruptibly的作用是，如果当前线程未被中断，则获取锁定，如果已经中断则抛出异常，单纯的调用lock的lock方法不会抛出异常，

lock的tryLock方法的作用是，仅在调用时锁定未被另一个线程保持的情况下，才获取锁定。

lock的tryLock(long timeout,TimeUnit unit)的作用是，如果锁定在给定等待时间内没有被另一个线程保持，且当前线程未被中断，则获取该锁定。

lock的awaitUntil方法，在线程等待时间到达前，可被其他线程唤醒。

lock的awaitUninterruptibly是使用，线程状态不响应interrupt方法，只有通过condition.singal或者singalAll才能唤醒线程。意味着如果线程通过调用awaitUninterruptibly从而使得线程状态为waiting，并不会因为调用interrupt()方法而中断。

类ReentrantLock具有完全互斥排他的效果，即同一时间只有一个线程在执行ReentrantLock。lock方法后面的任务，这样做虽然保证了实例变量的线程的安全性，但效率也是非常低的，所以在jdk中提供了一种读写锁ReentrantReadWriteLock类，使用它可以加快效率，在某些不需要操作实例变量的方法中，完全可以使用读写锁，ReentrantReadWriteLock来提升该方法的代码的运行速度。

读写锁表示有两个锁，一个是读操作相关的锁，也成为共享锁，另一个是写操作的相关的锁，也被称作排它锁，也就是多个读锁之间不互斥，读锁与写锁互斥，写锁与写锁互斥，在没有线程Thread进行写入操作时，进行读取操作的多个Thread都可以获取读取锁，而进行写操作的Thread只有在获得写锁后才能进行写操作。即多个Thread可以同时进行读取操作，但是同一时刻只允许一个Thread进行写操作。

完全可以使用Lock对象将synchronized关键字替换掉，而且其具有的独特功能也是synchronized所不具有的，在学习并发时，Lock时synchronized关键字的进阶，掌握Lock有助于学习并发包中源代码的实现原理，在并发包中大量的类使用了Lock接口作为同步的处理方式。

自己的理解和总结 Timer

schedule和

scheduleAtFixedRate具有“追赶”的特点，指定时间已过，执行的任务超时，都会追加“任务”，追赶，当达到间隔时间和执行任务的总时间与当前“平衡”时，便不追赶了

在不需要追赶的情况下，schedule和scheduleAtFixedRate的表现一样。

在需要追赶的情况下，如果任务执行的时间（总是）超过period，二者的表现也是一样的，因为scheduleAtFixedRated没有多余的时间去追赶。

抛去追赶的特点，二者都是以任务开始执行的时间为参考点