学习多线程，要学会“外练互斥，内修可见”。

1. Volatile关键字

Volatile主要使用的场合是在多个线程之间感知实例变量的修改，并且可以获得最新值使用，当线程想要访问volatile修饰的变量时强制从公共堆栈中进行读取。

Volatile可以保证每次线程从主内存中刷新到最新的变量值，但是不能保证变量值被加载到线程内存之后对该变量做的修改操作是原子性的。Volatile变量在线程内存中被修改之后要立即同步回主内存中，以保证其他线程使用该volatile关键字修饰的变量时获取到的是最新的变量值。

也就是说，volatile关键字保证的是变量在不同线程之间的可见性，但是无法保证原子性，对于多个线程访问同一个实例变量还是需要加锁同步。

1. Volatile变量有两个特性

① 保证可见性：此变量对于所有线程的可见性，指的是当一个线程修改了这个变量的值，新值对于其他线程来说是立即可见的。

② 禁止指令重排序优化（内存屏障，也会牺牲掉一些性能）

1. 某个变量定义为Volatile的应用场景

① 运算结果不依赖于变量的当前值或者能够确保只有一个线程修改这个变量的值；

② 变量不需要与其他的状态变量共同参与不变约束。

1. Java内存模型对volatile变量定义的特殊规则

① 在工作内存中每次使用变量前都需要从主内存中刷新最新值；

② 每次修改变量的值之后都必须立刻同步到主内存中；

③ 要求volatile修饰的变量不会被指令重新排序。

1. 不支持原子性，非线程安全

对一个volatile修饰的变量的读写操作不是原子性的。因为如果在第一个线程加载某个volatile修饰的变量值到工作内存之后有其他线程修改了这个变量值，那么第一个线程是感知不到这个值的变化的。这个时候就会出现线程安全的问题，所以为了保证线程安全问题还是需要synchronized关键字。

PS:指令重排优化：在保证可以得到程序正确执行结果的前提下，CPU允许将多条指令不按程序规定的顺序分开发送到各个相应电路单元处理。

1. Volatile变量的使用

并不建议过多地依赖volatile变量。如果在代码中过多地依赖volatile变量来控制状态可见性，通常比使用锁的代码更脆弱，也更难以理解。

仅当volatile变量能简化代码的实现以及对同步策略的验证，才应使用volatile变量。如果在验证正确性时需要对可见性进行复杂的判断，那么就不要使用volatile变量。Volatile变量的正确使用方式包括：①确保它们自身状态的可见性；②确保它们所引用的对象的状态的可见性；③标识一些重要的程序生命周期事件的发生(比如初始化或关闭)。

1. Synchronized关键字

Synchronized关键字保证在同一时刻，只有一个线程可以执行某个对象内某一个方法或某一段代码块。包含两个特征：互斥性和可见性。Synchronized可以解决一个线程看到对象处于不一致的状态，可以保证进入同步方法或者同步代码块的每个线程都可以看到由同一个锁保护之前所有的修改效果。

|  |  |  |
| --- | --- | --- |
| 方法 | static方法 | 锁的是class对象，该类的所有对象访问这个static方法都被阻塞。 |
| 非static方法 | 锁的是对象，必须持有该对象锁才能访问对象内的同步方法。 |
| 代码块 | this对象 | 与锁非static方法一样，都是给对象上锁。 |
| 非this对象x | 给对象x上锁，想访问这个代码段必须持有这个对象x的锁才可以。 |
| class对象 | 与给static方法加同步关键字一样，锁的都是class对象。 |

注意：class锁和对象锁不是同一种锁。持有class锁则可以访问同步的static方法和锁定class对象的代码段；持有对象锁则可以访问同步的非static方法和锁定this对象的代码段。只有多个线程持有同一个对象的锁时，访问该对象内的同步方法才会被阻塞，如果持有的不是同一个对象的锁则异步执行。

Synchronized关键字可以用于同步方法和同步代码块。同步方法又可分为同步static方法和非static方法，① 如果是给static方法加上synchronized关键字，则说明同步的是当前类的.class类，那么后面所有对这个static方法的访问都会被阻塞，但是此时可以访问其他没有加Synchronized关键字的方法或者是加了Synchronized关键字的非static方法；② 如果给非static方法加上Synchronized关键字，则同步的是当前对象，这样的话其他想访问同一个对象下的Synchronized同步方法就会被阻塞，但是不影响访问Synchronized同步的static方法，因为Synchronized非static方法是某个对象实例加锁，而Synchronized static方法是给.class对象加锁，但是Class锁是对类的所有对象都有效，也就是说如果现在有个static方法加上了Synchronized关键字，则这个类的所有对象都会对这个方法进行同步操作。

Synchronized同步代码块的时候分为Synchronized(this 对象)、Synchronized(非this 对象)、Synchronized(class对象)。① Synchronized(this 对象)同步的也是当前对象，而Synchronized(非this 对象)则是对某个非this对象进行同步即锁定。② Synchronized(非this 对象)同步代码块的方法在进行同步操作时，对象监视器必须是同一个对象。如果不是同一个对象监视器，运行的结果就是异步调用了。③ Synchronized(class对象)与给static方法加上Synchronized关键字是一样的。

另外使用Synchronized需要注意的地方：

① Synchronized锁可重入：持有某个对象的锁可继续访问需要持有该对象锁才可访问的方法和代码段；

② 可重入锁也支持在父子类继承的环境中：调用子类中同步的方法时也可以访问父类中需要对象锁的方法和代码段；

③ 同步方法内出现异常时将会自动释放持有的对象锁；

④ 同步不能继承，所以还需要在子类中需要同步的方法上加同步关键字。

1. Volatile和Synchronized的区别

① 关键字volatile是线程同步的轻量级实现，volatile只能用于同步变量；而Synchronized可以用于修饰方法和代码块；

② 多线程访问volatile不会发生阻塞，而Synchronized会发生阻塞；

③ volatile可保证数据的可见性，但是不能保证数据的一致性(原子性)，而Synchronized可以保证原子性，也可以间接保证可见性，因为它会将私有内存和公共内存中的数据做同步。

④ volatile关键字解决的是多线程之间的可见性，而Synchronized解决的是多个线程之间访问资源的同步性。

1. 并发编程中的三个概念(原子、可见、有序)

Java内存模型是围绕着在并发过程中如何处理原子性、可见性和有序性三个特征建立的。

① 原子性：一个操作要么全部执行完毕 ，要么根本就不执行。Java内存模型直接保证的原子性变量操作有read、load、assign、use、store和write。

② 可见性：多个线程访问同一个变量时一个线程修改了这个变量的值，其他线程能够立即看得到修改的值。Java语言中的volatile、synchronized和final三个关键字都可保证操作时变量的可见性。

③ 有序性：即程序执行的顺序按照代码的先后顺序执行。Java语言提供了volatile和synchronized两个关键字来保证线程之间操作的有序性。

总之，要想让并发程序正确地执行，必须要保证原子性、可见性以及有序性。只要有一个没有被保证，就有可能会导致程序运行不正确。

1. 先行发生原则

先行发生是Java内存模型中定义的两项操作之间的偏序关系，如果说操作A先行发生于操作B，那么操作A产生的影响将会被操作B观察到，“影响”包括修改了内存中共享变量的值、发送了消息、修改了变量等。

Java内存模型中定义的一些“天然的”先行发生关系：① 程序次序原则(控制流中的先后顺序)、② 管程锁定原则(同一个锁的unlock发生在下一次lock)、③ volatile变量规则(禁止指令重排优化)、④ 线程启动原则(线程的start操作优先于线程内部的其他所有操作)、④ 线程终止规则(线程内部的所有操作优先于线程终止操作)、⑥ 线程中断原则(对线程interrupt()方法的调用先行发生于被中断检测代码检测到中断的发生)、⑦ 对象终结原则(对象初始化操作先行于finalize操作的发生)、⑧ 传递性(A先行发生于B，B先行发生于C，则A先行发生于C)。

时间先后顺序与先行发生原则之间基本没有太大关系，衡量并发问题的时候不要受到时间上先后发生的干扰，一切以先行发生原则为准。

1. Compare and swap(CAS)

当前的处理器基本都支持CAS，只不过每个厂家所实现的算法并不一样罢了，每一个CAS操作过程都包含三个运算符：一个内存地址V，一个期望的值A和一个新值B，操作的时候如果这个地址上存放的值等于这个期望的值A，则将地址上的值赋为新值B，否则不做任何操作。CAS的基本思路就是，如果这个地址上的值和期望的值相等，则给其赋予新值，否则不做任何事儿，但是要返回原值是多少。处理过程是一个原子操作。

这样怎么实现线程安全呢？其实在语言层面是没有做任何同步的操作的，源码上也没有任何锁加在上面，可它为什么是线程安全的呢？这就是Atomic包下这些类的奥秘：语言层面不做处理，我们将其交给硬件—CPU和内存，利用CPU的多处理能力，实现硬件层面的阻塞，再加上volatile变量的特性即可实现基于原子操作的线程安全。所以说，CAS并不是无阻塞，只是阻塞并非在语言、线程方面，而是在硬件层面，所以无疑这样的操作会更快更高效！

虽然基于CAS的线程安全机制很好很高效，但要说的是，并非所有线程安全都可以用这样的方法来实现，这只适合一些粒度比较小，形如计数器这样的需求用起来才有效，否则也不会有锁的存在了。

CAS带来的三大问题：

① ABA问题：因为CAS需要在操作值的时候检查下值有没有发生变化，如果没有发生变化则更新，但是如果一个值原来是A，变成了B，又变成了A，那么使用CAS进行检查时会发现它的值没有发生变化，但是实际上却变化了。JUC包解决ABA问题的解决思路就是使用版本号。提供了一个带有标记的原子引用类”AtomicStampedReference”，在变量前面追加上版本号，每次变量更新的时候把版本号加一，那么A－B－A 就会变成1A-2B－3A。AtomicStampedReference类中的compareAndSet方法的作用是首先检查当前引用和预期引用是否相等，并且检查当前值和预期值是否相等，如果都相等，则以原子形式将该引用和该标志的值设置为给定的更新值。

② 循环时间长开销大：自旋CAS如果长时间不成功，会给CPU带来非常大的执行开销。如果JVM能支持处理器提供的pause指令那么效率会有一定的提升，pause指令有两个作用，第一它可以延迟流水线执行指令（de-pipeline）,使CPU不会消耗过多的执行资源，延迟的时间取决于具体实现的版本，在一些处理器上延迟时间是零。第二它可以避免在退出循环的时候因内存顺序冲突（memory order violation）而引起CPU流水线被清空（CPU pipeline flush），从而提高CPU的执行效率。

③ 只能保证一个共享变量的原子操作。当对一个共享变量执行操作时，我们可以使用循环CAS的方式来保证原子操作，但是对多个共享变量操作时，循环CAS就无法保证操作的原子性，这个时候就可以用锁。

1. 线程调度

线程调度指的就是给线程分配使用处理器的过程。主要的调度方式有两种：协同式调度和抢占式调度。

协同式调度指的是线程完成自己的任务之后主动通知系统切换到另一个线程上。优点是实现简单，线程对于自己的切换是已知的，不存在线程同步的问题；缺点是如果一个线程一直阻塞占用处理器，则其他线程都会被阻塞。

抢占式调度是由系统来为线程分配使用处理器的时间片。优点：线程的执行时间是系统可控的，不会出现一个线程导致整个进程阻塞的问题。

Java使用的线程调度就是抢占式调度。

1. 公平锁与非公平锁

公平锁是多个线程在等待同一个锁时，必须按照申请锁的时间顺序来一次获得锁(默认的是不公平锁)；

非公平锁就是一种获取锁的抢占机制，是随机获得锁的，和公平锁不一样的就是先来的不一定先得到锁。

1. 介绍一下ThreadLocal，为什么要使用ThreadLocal

ThreadLocal类解决的是变量在不同线程间的隔离性，每个线程拥有自己的不同值。Java.lang.ThreadLocal类来实现线程本地存储的功能。ThreadLocal为每个使用该变量的线程中都创建了一个副本，那么每个线程可以访问自己内部的副本变量，每个线程内部都会有一个该变量，且在线程内部任何地方都可以使用，线程之间互不影响，这样一来就不存在线程安全问题，也不会严重影响程序执行性能。

每一个线程的Thread对象中都有一个ThreadLocalMap对象，这个对象存储了一组以ThreadLocal.threadLocalHashCode为键，以本地线程变量为值的K-V值对。ThreadLocal对象就是当前线程的ThreadLocalMap的访问入口，每一个ThreadLocal对象都包含了一个独一无二的threadLocalHashCode的值，使用这个值就可以在线程K-V值对中找回对应的本地线程变量。

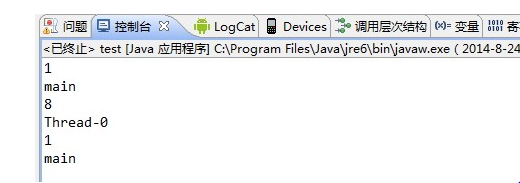
get()方法是用来获取ThreadLocal在当前线程中保存的变量副本，set()用来设置当前线程中变量的副本，remove()用来移除当前线程中变量的副本，initialValue()是一个protected方法，一般是用来在使用时进行重写的，它是一个延迟加载方法。操作的步骤都是先取得当前线程t，然后通过getMap(t)方法获取到一个map，map的类型为ThreadLocalMap。然后接着下面获取到<key,value>键值对。

总结：实际上通过ThreadLocal创建的副本是存储在每个线程自己内ThreadLocalMap类型对象中的。ThreadLocalMap的键值为ThreadLocal对象的原因是每个线程中可有多个ThreadLocal变量。<http://www.cnblogs.com/dolphin0520/p/3920407.html>

最常见的ThreadLocal使用场景是用来解决数据库连接、Session管理等。

|  |
| --- |
| public class Test {  ThreadLocal<Long> longLocal = new ThreadLocal<Long>();  ThreadLocal<String> stringLocal = new ThreadLocal<String>();      public void set() {  longLocal.set(Thread.currentThread().getId());  stringLocal.set(Thread.currentThread().getName());  }    public long getLong() {  return longLocal.get();  }    public String getString() {  return stringLocal.get();  }    public static void main(String[] args) throws InterruptedException {  final Test test = new Test();      test.set();  System.out.println(test.getLong());  System.out.println(test.getString());  Thread thread1 = new Thread(){  public void run() {  test.set();  System.out.println(test.getLong());  System.out.println(test.getString());  };  };  thread1.start();  thread1.join();    System.out.println(test.getLong());  System.out.println(test.getString());  }  } |

输出结果如下：



解释：在这段代码中由两个ThreadLocal变量分别是longLocal和stringLocal，这两个变量在thread1和main线程中都保存了自己的变量副本，且变量的值在两个线程之间相互隔离在每个线程内部，把自己线程内部的longLocal变量和stringLocal变量存储在该线程内部的ThreadLocalMap对象中，在这个ThreadLocalMap对象中存储的内容是ThreadLocal.threadLocalHashCode为键，以本地线程变量为值的K-V值对，例如在上面的例子中thread1内的ThreadLocalMap对象存储的内容是<longLocal对象的hashcode值，thread1线程内的longLocal的值8>，<stringLocal对象的hashcode值，thread1线程内的stringLoccal的值Thread-0>，在main线程内的ThreadLocalMap对象存储的内容是<longLocal对象的hashcode值，main线程内的longLocal的值1>，<stringLocal对象的hashcode值，thread1线程内的stringLoccal的值main>。每个线程内部可以有多个threadLocal变量，在ThreadLocalMap中以ThreadLocal变量的hash值为键，以该值在该线程内部的变量为值的<key,value>对。

补充：使用InheritableThreadLocal类可以让子线程从父线程中取得值，但是继承之后在父线程或子线程中再修改就互不影响了。《Java多线程编程核心技术》

1. 乐观锁（CAS 可以举原子类中的例子）和悲观锁（synchronized）

悲观锁(Pessimistic Lock), 顾名思义，就是很悲观，每次去拿数据的时候都认为别人会修改，所以每次在拿数据的时候都会上锁，这样别人想拿这个数据就会阻塞直到它拿到锁。传统的关系型数据库里边就用到了很多这种锁机制，比如行锁，表锁等，读锁，写锁等，都是在做操作之前先上锁。

乐观锁(Optimistic Lock), 顾名思义，就是很乐观，每次去拿数据的时候都认为别人不会修改，所以不会上锁，但是在更新的时候会判断一下在此期间别人有没有去更新这个数据，可以使用版本号等机制。乐观锁适用于多读的应用类型，这样可以提高吞吐量，像数据库如果提供类似于write\_condition机制的其实都是提供的乐观锁。乐观锁是基于冲突检测的乐观并发策略。

Synchronized原始采用的就是CPU悲观锁机制，即线程获取的是独占锁。独占锁意味着其他线程只能依靠阻塞来等待线程释放锁。Lock采用的就是乐观锁方式。所谓乐观锁就是，每次不加锁而是假设没有冲突而去完成某项操作，如果因为冲突失败就重试，直到成功为止。乐观锁实现的机制就是CAS操作。

1. Java内存模型 JMM

Java内存模型的主要目标是定义程序中各个变量的访问规则，即在虚拟机中将变量存储到内存和从内存中取出变量这样的底层细节。

Java内存模型规定了所有的变量都存储在主内存中，但是每个线程还是有自己的工作内存，线程的工作内存中保存了被该线程使用到的变量的主内存副本拷贝，线程对变量的所有操作都必须在工作内存中进行，而不能直接读写主内存中的变量。不同的线程之间也无法直接访问对方工作内存中的变量，线程间变量值的传递均需要通过主内存来完成。

1. 锁优化

Java虚拟机团队在JDK1.5到JDK1.6之间对高效并发进行了重要改进，有以下五种技术，都是为了在线程之间更高效地共享数据，以及解决竞争问题，从而提高程序的执行效率。

① 自旋锁和自适应锁

为了让线程等待，我们只需让线程执行一个忙循环（自旋），这就是所谓的自旋。

自旋是有限定的次数的，在jdk1.6引入了自适应自旋。自适应意味着自旋的时间不再固定了，而是由前一次在同一个锁上的自旋时间以及锁的拥有者的状态来决定。

② 锁消除

指在虚拟机即时编译器(JIT编译器)在运行时，对一些代码上要求同步，但是被检测到不可能存在共享数据竞争的锁进行消除。

③ 锁粗化

如果虚拟机探测到一连串零碎的操作都属于同一个对象加锁，将会把加锁同步的范围扩展到整个操作序列的外部。

④ 轻量级锁

并不是用来代替重量级锁的，它的本意是在没有多线程竞争的前提下减少传统重量级锁使用操作系统互斥量产生的性能消耗。轻量级锁能提升程序同步性能的依据是“对于绝大部分的锁，在整个同步周期内都是不存在竞争的”。在无竞争的情况下使用CAS操作避免了使用互斥量的开销，但是如果存在锁竞争，除了互斥量的开销外，还额外发生了CAS操作，因此在有竞争的情况下，轻量级锁会比传统的重量级锁更慢。

适用场景：追求响应时间，同步块执行速度非常快。

PS:使用操作系统互斥量来实现的传统锁称为是“重量级”锁。

⑤ 偏向锁

目的是消除数据在无竞争情况下的同步原语，进一步提高程序的运行性能。偏向锁会偏向于第一个获得它的线程，如果在后面的执行过程中，该锁没有被其他的线程获取，则持有偏向锁的线程将永远不需要再进行同步。

偏向锁可以提高带有同步但无竞争的程序性能。但是如果程序中大多数的锁总是被多个不同的线程访问，那偏向模式就是多余的。

适用场景：只有一个线程访问同步块场景。

1. 什么是线程安全

多个线程在访问同一个对象的时候不需要其他额外的同步手段或措施就能保证该对象被正确的访问并产生正确的执行结果。那么这个对象就是线程安全的。

线程安全的代码必须具备一个特征：代码本身封装了所有必要的正确性保障手段(如互斥同步)，使用该代码的开发人员无需关心多线程的问题也不用自己采用任何措施来保证多线程的正确调用。

线程不安全的代码在多个线程中使用时必须作同步处理，否则可能产生不可预期的后果。

1. Java语言中的线程安全

可以将Java语言中各种操作共享的数据分为以下五类：不可变、绝对线程安全、相对线程安全、线程兼容和线程对立。

① 不可变

如果共享数据是一个基本数据类型，只需要在定义的时候声明为final即可；如果是共享数据是一个对象，则需要保证对象的行为不会对其状态产生任何影响才行(最简单的做法就是把对象中带有状态的变量都声明为final)。

② 绝对线程安全

不管运行环境如何，调用者都不需要任何额外的同步措施的类可以称作是绝对线程安全的，但是这通常是需要付出相对较大的代价的。

③ 相对线程安全

对这个对象单独的操作是线程安全，在调用单个操作的时候不需要做其他额外的保障措施，但是对于一些特定顺序的连续调用，就可能需要在调用端使用额外的同步手段来保证调用的正确性。总之相对线程安全就是多个对象对这个对象单独操作的时候是线程安全的，但是如果多个线程操作这个对象的不同行为时就需要调用端使用同步的手段来保证调用的正确顺序了。

在Java语言中，大部分的线程安全类都是属于这种类型，例如Vector、HashTable、Collections的synchronizedCollection()等。

④ 线程兼容

线程兼容是指对象本身并不是线程安全的，但是可以通过在调用端正确地使用同步手段来保证对象在并发环境中可以安全地使用。如Vector和HashTable相对应的集合类ArrayList和HashMap等。

⑤ 线程对立

线程对立是指物理调用端是否采取了同步措施，都无法再多线程环境中并发使用的代码。

1. 停止线程

Java中有三种方法：

* 1. 使用退出标志，是线程正常退出，也就是当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、resume、suspend都是作废过期的方法，使用它们会产生不可预料的结果；如果使用stop强制停止线程，则有可能使一些清理工作无法完成，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，处理数据不一致的问题。使用suspend，在调用后线程不会保证线程的资源（比如）正常释放，而是占着资源进入睡眠状态，容易发生死锁，比如suspend方法不会释放锁，这就会导致如果用suspend挂起一个线程，那么程序就会发生死锁。
  3. 使用interrupt()方法结合异常来中断线程，这样在catch块中还可以将异常向上抛，使线程停止的事件得以传播。推荐这个！！
  4. 使用return方法停止线程。

1. java中的线程池

(1). Executor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，事先创建若干个可执行的线程放入一个容器中，需要的时候从容器中获取不需要自行创建，不用的时候再放回到容器中，从而减少创建和销毁线程对象的开销。



这里有一个非常巧妙的设计方式，假如我们来设计线程池，可能会有一个任务分派线程，当发现有线程空闲时，就从任务缓存队列中取一个任务交给空闲线程执行。但是在这里，并没有采用这样的方式，因为这样会要额外地对任务分派线程进行管理，无形地会增加难度和复杂度，这里直接让执行完任务的线程去任务缓存队列里面取任务来执行。

(2). 线程池状态

在ThreadPoolExecutor中定义了一个volatile变量，另外定义了几个static final变量表示线程池的各个状态：

volatile int runState;// 表示当前线程池的状态，它是一个volatile变量用来保证线程之间的可见性

static final int RUNNING  = 0;// 当创建线程池后，初始时，线程池处于RUNNING状态

static final int SHUTDOWN  = 1;// 如果调用了shutdown()方法，则线程池处于SHUTDOWN状态，此时线程池不能够接受新的任务，它会等待所有任务执行完毕；

static final int STOP     = 2;// 如果调用了shutdownNow()方法，则线程池处于STOP状态，此时线程池不能接受新的任务，并且会去尝试终止正在执行的任务

static final int TERMINATED = 3;// 当线程池处于SHUTDOWN或STOP状态，并且所有工作线程已经销毁，任务缓存队列已经清空或执行结束后，线程池被设置为TERMINATED状态

(3). corePoolSize、maximumPoolSize、keepAliveTime

① corePoolSize：核心池的大小，这个参数跟后面讲述的线程池的实现原理有非常大的关系。在创建了线程池后，默认情况下，线程池中并没有任何线程，而是等待有任务到来才创建线程去执行任务，除非调用了prestartAllCoreThreads() 或者prestartCoreThread()方法，从这2个方法的名字就可以看出，是预创建线程的意思，即在没有任务到来之前就创建corePoolSize个线程或者一个线程。默认情况下，在创建了线程池后，线程池中的线程数为0，当有任务来之后，就会创建一个线程去执行任务，即使其他空闲的基本线程能够执行新任务也会创建线程，当线程池中的线程数目达到corePoolSize后，就会把到达的任务放到缓存队列当中；

② maximumPoolSize：线程池最大线程数，这个参数也是一个非常重要的参数，它表示在线程池中最多能创建多少个线程；

③ keepAliveTime：表示线程没有任务执行时最多保持多久时间会终止。默认情况下，只有当线程池中的线程数大于corePoolSize时，keepAliveTime才会起作用，直到线程池中的线程数不大于corePoolSize，即当线程池中的线程数大于corePoolSize时，如果一个线程空闲的时间达到keepAliveTime，则会终止，直到线程池中的线程数不超过corePoolSize。但是如果调用了allowCoreThreadTimeOut(boolean)方法，在线程池中的线程数不大于corePoolSize时，keepAliveTime参数也会起作用，直到线程池中的线程数为0；unit：参数keepAliveTime的时间单位

(4). Worker的作用

线程池创建线程时，会将线程封装成工作线程worker，worker在执行完任务后，还会循环获取工作队列里的任务来执行。

(5). 任务提交给线程池之后的处理策略

要知道任务提交给线程池之后的处理策略，这里总结一下主要有4点：

* 如果当前线程池中的线程数目小于corePoolSize，则每来一个任务，就会创建一个线程去执行这个任务；
* 如果当前线程池中的线程数目>=corePoolSize，则每来一个任务，会尝试将其添加到任务缓存队列当中，若添加成功，则该任务会等待空闲线程将其取出去执行；
* 若添加失败（一般来说是任务缓存队列已满），则会尝试创建新的线程去执行这个任务；
* 如果当前线程池中的线程数目达到maxinumPoolSize，则会采取任务拒绝策略进行处理；
* 如果线程池中的线程数量大于 corePoolSize时，如果某线程空闲时间超过keepAliveTime，线程将被终止，直至线程池中的线程数目不大于corePoolSize；如果允许为核心池中的线程设置存活时间，那么核心池中的线程空闲时间超过keepAliveTime，线程也会被终止。

(6). 任务缓存队列及排队策略

　在前面我们多次提到了任务缓存队列，即workQueue，它用来存放等待执行的任务。workQueue的类型为BlockingQueue<Runnable>，通常可以取下面三种类型：

① ArrayBlockingQueue：基于数组的先进先出队列，此队列创建时必须指定大小；

② LinkedBlockingQueue：基于链表的先进先出队列，如果创建时没有指定此队列大小，则默认为Integer.MAX\_VALUE；

　③ synchronousQueue：这个队列比较特殊，不存储元素的阻塞队列，每个插入操作必须等到另一个线程调用移除操作，否则插入操作一直处于阻塞状态，吞吐量高于LinkedBlockingQueue，静态工厂方法Exectors.newCachedThreadPool使用了这个队列。

(7). 任务拒绝策略

　当线程池的任务缓存队列已满并且线程池中的线程数目达到maximumPoolSize，如果还有任务到来就会采取任务拒绝策略，通常有以下四种策略：

① ThreadPoolExecutor.AbortPolicy:丢任务并抛RejectedExecutionException 异常。

② ThreadPoolExecutor.DiscardPolicy：也是丢弃任务，但是不抛出异常。

③ ThreadPoolExecutor.DiscardOldestPolicy：丢弃队列最前面的任务，然后重新尝试执行任务（重复此过程）

④ ThreadPoolExecutor.CallerRunsPolicy：由调用线程处理该任务

(8). 线程池的关闭

　ThreadPoolExecutor提供了两个方法，用于线程池的关闭，分别是shutdown()和shutdownNow()，其中：

* shutdown()：不会立即终止线程池，而是要等所有任务缓存队列中的任务都执行完后才终止，但再也不会接受新的任务
* shutdownNow()：立即终止线程池，并尝试打断正在执行的任务，并且清空任务缓存队列，返回尚未执行的任务

(9). 线程池容量的动态调整

　　ThreadPoolExecutor提供了动态调整线程池容量大小的方法：setCorePoolSize()和setMaximumPoolSize()，

* setCorePoolSize：设置核心池大小
* setMaximumPoolSize：设置线程池最大能创建的线程数目大小

　当上述参数从小变大时，ThreadPoolExecutor进行线程赋值，还可能立即创建新的线程来执行任务。

(10). 如何合理配置线程池的大小

调整线程池的大小基本上就是避免两类错误：线程太少或线程太多。幸运的是，对于大多数应用程序来说，太多和太少之间的余地相当宽。

线程池过小则达不到线程复用的作用，并且会有太多的任务阻塞在缓存队列中。

如果线程池太大，那么被那些线程消耗的资源可能严重地影响系统性能。在线程之间进行切换将会浪费时间，而且使用超出比实际需要的线程可能会引起资源匮乏 问题，因为池线程正在消耗一些资源，而这些资源可能会被其它任务更有效地利用。

一般需要根据任务的类型来配置线程池大小：

如果是CPU密集型任务，就需要尽量压榨CPU，参考值可以设为 NCPU+1。

如果是IO密集型任务，参考值可以设置为2\*NCPU

当然，这只是一个参考值，具体的设置还需要根据实际情况进行调整，比如可以先将线程池大小设置为参考值，再观察任务运行情况和系统负载、资源利用率来进行适当调整。

(11). Java中四种线程池的使用

不过在java doc中，并不提倡我们直接使用ThreadPoolExecutor，而是经常使用Executors类中提供的几个静态方法来创建线程池。从它们的具体实现来看，它们实际上也是调用了ThreadPoolExecutor，只不过参数都已配置好了。

|  |
| --- |
| Executors.newCachedThreadPool();        //创建一个缓冲池，缓冲池容量大小为Integer.MAX\_VALUE  Executors.newFixedThreadPool(int);    //创建固定容量大小的缓冲池  Executors.newSingleThreadExecutor();   //创建容量为1的缓冲池  Executors.newScheduledThreadPool(corePoolSize);// 创建一个定长线程池，支持定时及周期性任务执行。 |

|  |
| --- |
| public static ExecutorService newCachedThreadPool() {      return new ThreadPoolExecutor(0, Integer.MAX\_VALUE,                                    60L, TimeUnit.SECONDS,                                    new SynchronousQueue<Runnable>());  }  public static ExecutorService newFixedThreadPool(int nThreads) {      return new ThreadPoolExecutor(nThreads, nThreads,                                    0L, TimeUnit.MILLISECONDS,                                    new LinkedBlockingQueue<Runnable>());  }  public static ExecutorService newSingleThreadPool() {      return new FinalizableDelegatedExecutorService          (new ThreadPoolExecutor(1, 1,                                  0L, TimeUnit.MILLISECONDS,                                  new LinkedBlockingQueue<Runnable>()));  }  public static ScheduledExecutorService newScheduledThreadPool(int corePoolSize) {  return new ScheduledThreadPoolExecutor(corePoolSize);  } |

① 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。实现原理将corePoolSize设置为0，将maximumPoolSize设置为Integer.MAX\_VALUE，使用的SynchronousQueue(无界)，也就是说来了任务就创建线程运行，当线程空闲超过60秒，就销毁线程。是大小无界的线程池，适用于执行很多的短期异步任务的小程序，或者是负载较轻的服务器。

② newFixedThreadPool 创建一个定长线程池，可控制线程最大并发数，超出的线程会在队列中等待。这个创建的线程池corePoolSize和maximum PoolSize 值是相等的，它使用的LinkedBlockingQueue(无界队列)。适用于为了满足资源管理要求，而需要限制当前线程数量的应用场景。

③ newSingleThreadExecutor 创建一个单线程化的线程池，它只会用唯一的工作线程来执行任务，保证所有任务按照指定顺序(FIFO, LIFO, 优先级)执行。适用于需要保证顺序地执行各个任务，并且在任意时间点，不会有多个线程是活动的应用场景。它使用的LinkedBlockingQueue(无界队列)。

④ newScheduledThreadPool 创建一个定长线程池，支持定时及周期性任务执行。适用于多个后台线程执行周期任务，同时为了满足资源管理的需求而限制后台线程的数量的应用场景。

⑤ newSingleThreadScheduledExector只包含一个线程的定时及周期线程池。

(12). 为什么使用线程池

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。线程池为线程生命周期开销问题和资源不足问题提供了解决方案。

① 通过对多个任务重用线程，线程创建的开销被分摊到了多个任务上。

② 因为在请求到达时线程已经存在，所以无意中也消除了线程创建所带来的延迟。这样，就可以立即为请求服务，使应用程序响应更快。

③ 通过适当地调整线程池中的线程数目，也就是当请求的数目超过某个阈值时，就强制其它任何新到的请求一直等待，直到获得一个线程来处理为止，从而可以防止资源不足。

(13). 使用线程池需要注意的地方

① 合理设置线程池的核心数量大小；

② 不要对那些同步等待其他任务结果的任务排队，以免引起死锁；

③ 在为时间可能很长的操作使用合用的线程时要小心，避免阻塞其他线程。

④ 和其他资源一样，线程池在使用完毕后也需要释放，用shutdown()方法可以关闭线程池，如果当时池里还有没有被执行的任务，它会等待任务执行完毕，在等待期间试图进入线程池的任务将被拒绝。也可以用shutdownNow()来关闭线程池，它会立刻关闭线程池，没有执行的任务作为返回值返回。

⑤ 执行时间不同的任务可以交给不同规模的线程池来处理，或者可以使用优先级队列，让执行时间短的任务先执行。

(14). 线程池的监控

可以通过线程池提供的参数进行监控，在监控线程池的时候可以使用一下属性：

TaskCount：线程池需要执行的任务数量；

completedTaskCount：线程池在运行过程中已完成的任务数量，小于等于taskCount；

largestPoolSize：线程池里曾经创建过的最大线程数量。通过这个数据可以知道线程池是否曾经满过；

getPoolSize：线程池的线程数量。

GetActiveCount：获取活动的线程数。

通过扩展线程池进行监控，可以继承线程池来自定义线程池，重写线程池的beforeExecute、afterExecute和terminated方法。

(14). 线程池中的角色

线程管理器、工作线程、任务接口、任务队列。

1. ReentrantLock(可重入锁)

可重入锁的三个特性：

① 等待可中断：当持有锁的线程长期不释放锁的时候，正在等待的线程可以选择放弃等待，改为处理其他事情。可中断特性对处理执行时间长的同步块很有帮助。

② 公平锁与非公平锁：ReentrantLock默认是非公平锁，但是可以通过带布尔值的构造函数要求使用公平锁。

公平锁是多个线程在等待同一个锁时，必须按照申请锁的时间顺序来一次获得锁(默认的是不公平锁)；

非公平锁就是一种获取锁的抢占机制，是随机获得锁的，和公平锁不一样的就是先来的不一定先得到锁。

③ 锁可以绑定多个条件condition，可以选择性通知指定的某个种类的condition。

1. 如何实现Java多线程

① 继承Thread类，重写run()方法；

② 实现Runnable接口，并实现该接口的run()方法；推荐这个！！(实现这个接口的类还可以继承自其它的类，用第一方法就没有办法再去继承别的类)

Step1:自定义类并实现Rubbale接口，实现run()方法；

Step2:创建Thread对象，用实现Runnable接口的对象作为参数实例化Thread对象；

Step3:调用Thread的start()方法。

③ 实现Callable接口，重写call()方法。

Callabe接口实际上是属于Executor框架中的功能类，Callable接口与Runnable接口的功能类似，提供了比Runnable更强大的功能，主要表现有三点：a.Callable可以在任务结束之后提供一个返回值，Runnable则不能；b.Callable接口中的call()方法可以抛出异常，而Runnable()的run()方法不能抛出异常；c.运行Callable可以拿到一个Future对象，Future对象表示异步计算的结果，提供了检查计算是否完成的方法。，当调用future的get()方法以获取结果时，当前线程就会阻塞，直到call()方法结束返回结果。

|  |
| --- |
| public static class CallableTest implements Callable<String>{  @Override  public String call() throws Exception {  return "hello word";  }  }  public static void main(String[] args) {  ExecutorService threadPool = Executors.newSingleThreadExecutor();//1.创建线程池  Future<String> future = threadPool.submit(new CallableTest());//2.提交任务  System.out.println("waiting thread to finish");  try {  System.out.println(future.get());//3.获取返回值  } catch (InterruptedException | ExecutionException e) {  e.printStackTrace();  } |



1. 为什么wait, notify 和 notifyAll这些方法不在thread类里面？

　　简单的说，由于wait，notify和notifyAll都是锁级别的操作，把它们定义在Object类中因为锁属于对象。一个很明显的原因是JAVA提供的锁是对象级的而不是线程级的，每个对象都有锁，通过线程获得。如果线程需要等待某些锁那么调用对象中的wait()方法就有意义了。如果wait()方法定义在Thread类中，线程正在等待的是哪个锁就不明显了。

1. 怎么检测一个线程是否拥有锁？

　　在java.lang.Thread中有一个方法叫holdsLock()，它返回true如果当且仅当当前线程拥有某个具体对象的锁

1. 生产/消费者模型

生产/消费者问题是个非常典型的多线程问题，涉及到的对象包括“生产者”、“消费者”、“仓库”和“产品”。他们之间的关系如下：

① 生产者仅仅在仓储未满时候生产，仓满则停止生产。

② 消费者仅仅在仓储有产品时候才能消费，仓空则等待。

③ 当消费者发现仓库没产品可消费时候会通知生产者生产。

④ 生产者在生产出可消费产品时候，应该通知等待的消费者去消费。

用wait/notify/notifyAll实现和用Lock的Condition实现。

用wait/notify/notifyAll 实现生产者消费者模型：

方法一：用五个类来实现，分别为Produce(实现生产过程), Consumer(实现消费过程), ProduceThread(实现生产者线程),ConsumeThread(实现消费者线程), Main等。需要注意的是有两个地方。

① 用while判断当前list是否为空；

② 调用的是object的notifyAll()方法而不是notify()方法。

方法二：用四个类实现，分别为MyService(实现生产和消费过程用synchronized关键字实现同步)，ProduceThread(实现生产者线程),ConsumeThread(实现消费者线程), Main。需要注意的也是方法一中的两个地方while和notifyAll()。

用Lock和Condition实现。共有四个类，分别是分别为MyService(实现生产和消费过程，用lock实现线程间同步)，ProduceThread(实现生产者线程),ConsumeThread(实现消费者线程), Main。需要注意的也是方法一中的两个地方while和signalAll()。

|  |
| --- |
| package ProduceConsumer;  import java.util.ArrayList;  public class Produce {    public Object object;  public ArrayList<Integer> list;//用list存放生产之后的数据，最大容量为1    public Produce(Object object,ArrayList<Integer> list ){  this.object = object;  this.list = list;  }    public void produce() {    synchronized (object) {  /\*只有list为空时才会去进行生产操作\*/  try {  while(!list.isEmpty()){  System.out.println("生产者"+Thread.currentThread().getName()+" waiting");  object.wait();  }  int value = 9999;  list.add(value);  System.out.println("生产者"+Thread.currentThread().getName()+" Runnable");  object.notifyAll();//然后去唤醒因object调用wait方法处于阻塞状态的线程  }catch (InterruptedException e) {  e.printStackTrace();  }  }  }  }  package ProduceConsumer;  import java.util.ArrayList;  public class Consumer {    public Object object;  public ArrayList<Integer> list;//用list存放生产之后的数据，最大容量为1    public Consumer(Object object,ArrayList<Integer> list ){  this.object = object;  this.list = list;  }    public void consmer() {    synchronized (object) {  try {  /\*只有list不为空时才会去进行消费操作\*/  while(list.isEmpty()){  System.out.println("消费者"+Thread.currentThread().getName()+" waiting");  object.wait();  }  list.clear();  System.out.println("消费者"+Thread.currentThread().getName()+" Runnable");  object.notifyAll();//然后去唤醒因object调用wait方法处于阻塞状态的线程    }catch (InterruptedException e) {  e.printStackTrace();  }  }  }    }  package ProduceConsumer;  public class ProduceThread extends Thread {  private Produce p;  public ProduceThread(Produce p){  this.p = p;  }  @Override  public void run() {  while (true) {  p.produce();  }  }  }  package ProduceConsumer;  public class ConsumeThread extends Thread {  private Consumer c;  public ConsumeThread(Consumer c){  this.c = c;  }  @Override  public void run() {  while (true) {  c.consmer();  }  }  }  public class Main {  public static void main(String[] args) {  Object object = new Object();  ArrayList<Integer> list = new ArrayList<Integer>();    Produce p = new Produce(object, list);  Consumer c = new Consumer(object, list);    ProduceThread[] pt = new ProduceThread[2];  ConsumeThread[] ct = new ConsumeThread[2];    for(int i=0;i<2;i++){  pt[i] = new ProduceThread(p);  pt[i].setName("生产者 "+(i+1));  ct[i] = new ConsumeThread(c);  ct[i].setName("消费者"+(i+1));  pt[i].start();  ct[i].start();  }  }  } |

方法二：

|  |
| --- |
| package ProduceConsumer2;  import java.util.ArrayList;  public class MyService {    public ArrayList<Integer> list = new ArrayList<Integer>();//用list存放生产之后的数据，最大容量为1  synchronized public void produce() {    try {  /\*只有list为空时才会去进行生产操作\*/  while(!list.isEmpty()){  System.out.println("生产者"+Thread.currentThread().getName()+ " waiting");  this.wait();  }  int value = 9999;  list.add(value);  System.out.println("生产者"+Thread.currentThread().getName()+" Runnable");  this.notifyAll();//然后去唤醒因object调用wait方法处于阻塞状态的线程  }catch (InterruptedException e) {  e.printStackTrace();  }  }    synchronized public void consmer() {  try {  /\*只有list不为空时才会去进行消费操作\*/  while(list.isEmpty()){  System.out.println("消费者"+Thread.currentThread().getName()+" waiting");  this.wait();  }  list.clear();  System.out.println("消费者"+Thread.currentThread().getName()+" Runnable");  this.notifyAll();//然后去唤醒因object调用wait方法处于阻塞状态的线程  } catch (InterruptedException e) {  e.printStackTrace();  }  }    }  package ProduceConsumer2;  public class ProduceThread extends Thread {  private MyService p;  public ProduceThread(MyService p){  this.p = p;  }  @Override  public void run() {  while (true) {  p.produce();  }  }  }  package ProduceConsumer2;  public class ConsumeThread extends Thread {  private MyService c;  public ConsumeThread(MyService c){  this.c = c;  }  @Override  public void run() {  while (true) {  c.consmer();  }  }  }  package ProduceConsumer2;  public class Main {  public static void main(String[] args) {  MyService service = new MyService();    ProduceThread[] pt = new ProduceThread[2];  ConsumeThread[] ct = new ConsumeThread[2];    for(int i=0;i<2;i++){  pt[i] = new ProduceThread(service);  pt[i].setName("生产者 "+(i+1));  ct[i] = new ConsumeThread(service);  ct[i].setName("消费者"+(i+1));  pt[i].start();  ct[i].start();  }  }  } |

用Lock和Condition实现

|  |
| --- |
| package ConditionProduceConsumer;  import java.util.concurrent.locks.Condition;  import java.util.concurrent.locks.ReentrantLock;  public class MyService {    private ReentrantLock lock = new ReentrantLock();  private Condition condition = lock.newCondition();  private boolean hasValue = false;      public void produce() {  lock.lock();  try {  /\*只有list为空时才会去进行生产操作\*/  while(hasValue == true){  System.out.println("生产者"+Thread.currentThread().getName()+" waiting");  condition.await();  }  hasValue = true;  System.out.println("生产者"+Thread.currentThread().getName()+" Runnable");  condition.signalAll();//然后去唤醒因object调用wait方法处于阻塞状态的线程  } catch (InterruptedException e) {  e.printStackTrace();  }finally{  lock.unlock();  }    }    public void consmer() {  lock.lock();  try {  /\*只有list为空时才会去进行生产操作\*/  while(hasValue == false){  System.out.println("消费者"+Thread.currentThread().getName()+" waiting");  condition.await();  }  hasValue = false;  System.out.println("消费者"+Thread.currentThread().getName()+" Runnable");  condition.signalAll();//然后去唤醒因object调用wait方法处于阻塞状态的线程  } catch (InterruptedException e) {  e.printStackTrace();  }finally{  lock.unlock();  }    }  }  package ConditionProduceConsumer;  public class ProduceThread extends Thread {  private MyService p;  public ProduceThread(MyService p){  this.p = p;  }  @Override  public void run() {  while (true) {  p.produce();  }  }  }  package ConditionProduceConsumer;  public class ConsumeThread extends Thread {  private MyService c;  public ConsumeThread(MyService c){  this.c = c;  }  @Override  public void run() {  while (true) {  c.consmer();  }  }  }  package ConditionProduceConsumer;  public class Main {  public static void main(String[] args) {  MyService service = new MyService();    ProduceThread[] pt = new ProduceThread[2];  ConsumeThread[] ct = new ConsumeThread[2];    for(int i=0;i<1;i++){  pt[i] = new ProduceThread(service);  pt[i].setName("Condition 生产者 "+(i+1));  ct[i] = new ConsumeThread(service);  ct[i].setName("Condition 消费者"+(i+1));  pt[i].start();  ct[i].start();  }  }  } |

1. Lock与synchronized的区别

① 是否为语言内置：

Lock不是Java语言内置的，synchronized是Java语言的关键字，因此是内置特性。Lock是一个类，通过这个类可以实现同步访问；

② 用法

Synchronzied可以用于去修饰方法或代码块，而lock则需要在代码中显示的通过调用lock()和unlock()方法来指定起始位置和终止位置。Synchronize是托管给JVM的，而Lock的锁定是通过代码实现的，他有比synchronized更精确的线程语义。

③ 锁机制

Lock和synchronized有一点非常大的不同，采用synchronized不需要用户去手动释放锁，当synchronized方法或者synchronized代码块执行完之后，系统会自动让线程释放对锁的占用；而Lock则必须要用户去手动释放锁，如果没有主动释放锁，就有可能导致出现死锁现象；

④ 并发量对性能的影响

并发数量多的时候synchronized性能会下降很厉害，但是对lock影响不大。

⑤ Lock可以提高多个线程进行读操作的效率。

1. JUC包

* Java.unit.concurrent.atomic

Java从JDK1.5开始提供Java.unit.concurrent.atomic(简称Atomic包)，这个包中的原子操作类提供了一种用法简单、性能高效、线程安全地更新一个变量的方式。在Atomic包中一共提供了13个类，属于4种类型的原子更新方式，分别是原子更新基本类型、原子更新数组、原子更新引用和原子更新属性(字段)。

比如在AtomicInteger类中value值是volatile属性的，并且在对value进行加或减的操作是需要进行CAS比较。

比如AtomicInteger对象的addAndGet(int delta):以原子方式将输入的数值与实例中的值相加，并返回结果。

|  |
| --- |
| static AtomicInteger ai = new AtomicInteger(1);  public static void main(String[] args) {  System.out.println(ai.addAndGet(3));  System.out.println(ai.get());  }  /\*\*  \* Atomically adds the given value to the current value.  \*  \* @param delta the value to add  \* @return the updated value  \*/  public final int addAndGet(int delta) {  for (;;) {  int current = get();//Step1  int next = current + delta;//Step2  if (compareAndSet(current, next))//Step3  return next;  }  }  public final boolean compareAndSet(int expect, int update) {  return unsafe.compareAndSwapInt(this, valueOffset, expect, update);  } |

Step1：先取得AtomicInteger里存储的数值；

Step2：对AtomicInteger中的当前数值进行加法运算；

Step3：调用compareAndSet方法来进行原子更新操作，该方法先检查当前数值是否等于current，等于意味着AtomicInteger的当前数值更新成next的值，如果不等compareAndSet方法会返回false，程序会进入for循环重新进行compareAndSet操作。

1. 对象的发布与逸出

发布：是对象能够在当前作用域之外的代码中使用，如一个指向该对象的引用保存到其他代码可以访问的地方，或者在一个非私有的方法返回该引用，或者将引用传递到其他类的方法中。

1. Thread类的sleep()方法和对象的wait()方法都可以让线程暂停使用，它们有什么区别？

①方法所属的类

sleep()方法是Thread类的静态方法，调用此方法是线程用来控制自身的执行流程，可能是在执行过程中需要有一段时间不执行任何操作；而wait()方法则是Object类的方法，用于进行间通信，调用wait()方法则会让当前拥有该对象锁的线程等待，直到其他线程调用notify()或者notifyAll()方法，不过开发人员也可以设定一个时间，让其自动醒来；

②是否释放锁

Sleep方法不涉及进程间通信，因此调用sleep方法时不会释放对象锁，但是如果一个线程调用了sleep对象，则会进入阻塞状态，CPU的拥有权则会被撤去，直到sleep时间到才会进入到就绪状态，等待下次获得CPU执行权限再去执行。

Wait方法涉及到进程间通信，因此调用某个对象的wait()方法的线程就会释放该对象的锁，从而使线程所在对象中的其他synchronized方法可以被别的线程使用；

③怎么苏醒

调用sleep方法时的参数则是该线程sleep的时间，如果没有遇到异常，则时间到了就会进入到线程的就绪状态；

调用某个对象的wait()方法的线程则会在其他线程调用该对象的notify或者notifyAll方法时被唤醒。

④使用的区域

Sleep方法可以放在任何位置使用，obj.wait()方法由于其进行间通信的特殊意义，则必须放在synchronized同步的obj的临界区中使用。

⑤是否会抛出异常

Sleep方法不会释放锁标志，容易导致死锁问题的发生，因此，一般情况下，不推荐使用sleep()方法，而是推荐使用wait()方法。

1. 线程的sleep()方法和yield()方法有什么区别？

答：

① sleep()方法给其他线程运行机会时不考虑线程的优先级，因此会给低优先级的线程以运行的机会；yield()方法只会给相同优先级或更高优先级的线程以运行的机会；

② 线程执行sleep()方法后转入阻塞（blocked）状态，而执行yield()方法后转入就绪（ready）状态；

③ sleep()方法声明抛出InterruptedException，而yield()方法没有声明任何异常；

④ sleep()方法比yield()方法（跟操作系统CPU调度相关）具有更好的可移植性。

1. wait() 、notify()和notifyAll()

这三种方法被调用的地方必须都是synchronized修饰的方法或代码块中。Ps, 这个地方理解与《Java多线程编程核心技术》中P141理解的稍有一点不一样。书上提到调用notifyAll()之后所有的相关线程都处于就绪状态，但是我觉得应该是所有的线程都处于唤醒状态，只有获取到对象锁的线程才是处于就绪状态。

- wait()：在线程的代码流程中如果某个synchronized方法或者代码块执行了某个对象的wait()方法则该对象处于wait状态，释放持有的对象锁，直到该对象调用notify或者notifyAll方法才能被唤醒。正在被执行的线程就会释放该对象的锁，进入到阻塞状态。

- notify()：唤醒一个处于等待状态的线程，当然在调用此方法的时候，并不能确切的唤醒某一个等待状态的线程，而是由JVM确定唤醒哪个线程，而且与优先级无关； 唤醒的线程可以获得对象锁，然后进入到就绪状态，等待系统调度然后去执行。

- notityAll()：唤醒所有处于等待状态的线程，该方法并不是将对象的锁给所有线程。所有的线程被唤醒后，接着进行竞争，获取对象锁的线程进入到就绪态，等待系统调度去执行，执行结束后就会释放对象锁，之前竞争失败处于被唤醒阶段的线程就会继续重复刚才的动作，直到所有被唤醒的线程都执行结束。

注意：在执行notify()方法之后，当前线程不会马上释放该对象锁，呈wait状态的线程也不能马上获取该对象锁，要等到执行notify()方法的线程将程序执行完，也就是退出synchronized代码块之后，当前线程才会释放锁，呈wait状态所在的线程才可以获取该对象锁。

1. 死锁和解决办法

死锁：互相等待对方的资源。

解决：避免一个synchronized块内再去获取其他的对象锁；让线程按照相同的顺序去获取共享资源的锁；尽量保证每个锁只占用同一个资源。

七、 java内存模型

造成不一致的原因有：java的内存模型和指令的重排序？？？？？

1. Java内存模型的主要目标是定义程序中各个变量的访问规则，即在虚拟机中将变量存储到内存和从内存中取出变量这样的底层细节。

Java内存模型规定了所有的变量都存储在主内存中，但是每个线程还是有自己的工作内存，线程的工作内存中保存了被该线程使用到的变量的主内存副本拷贝，线程对变量的所有操作都必须在工作内存中进行，而不能直接读写主内存中的变量。不同的线程之间也无法直接访问对方工作内存中的变量，线程间变量值的传递均需要通过主内存来完成。

1. 原子性、可见性、有序性

Java的内存模型是围绕着在并发过程中如何处理原子性、可见性和有序性3个特征建立的。