Java 多线程学习笔记

创建线程的3种方法：

1、创建直接继承自Thread类创建线程子类。

  步骤如下：a 定义一个子类，同时要继承Thread类。

            b 在这个子类中重写run方法。

            c 在需要的地方创建该子类的实例，即创建一个线程实例。

            d 调用start方法来启动线程。

2、创建实现Runnable接口的类。

   步骤如下：a 定义一个类，并且实现了Runnable接口。

             b 在子类中实现接口的run方法，并且重写run方法。

             c 创建该子类的一个实例，并将该实例作为参数传入到Thread类的构造函数   中。Thread  t1=new Thread(new Runnable()).

             d 最后调用线程的start方法来启动线程。即t1.start()。

3、对于应用程序可以使用Excutor框架来创建线程。主要是使用Callable接口和Future接口。其中Callable接口和Runnable接口很像，但Callable接口会更加的抽象，而Future接口会返回线程异步计算的结果。

------------------------------------------------------------------------------

1. 多线程程序执行时速度不一定会比串行执行速度要快，这是因为多线程有创建线程和上下文切换的开销。如何减少上下文切换：无锁并发编程、CAS算法、使用最少线程、协程。
2. 避免死锁的几个常见的方法：避免一个线程获取多个锁；避免一个线程在锁内同时占用多个资源，尽量保证每个锁只占用一个资源；尝试使用定时锁，使用lock.tryLock(timeout)来替代使用内部锁机制；对于数据库锁，加锁和解锁必须在一个数据库连接里，否则会出现解锁失败的情况。
3. Synchronized实现同步的基础：Java中的每一个对象都可以作为锁。具体表现为以下三种形式：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；对于同步方法快，锁是Synchronized括号里配置的对象。

-------------------------------------------------------------------------------

Synchronized与Static Synchronized的区别：

Synchronized(实例锁)：锁在每一个对象上。如果该类是一个单例（比如单例模式），那么它就具有全局锁的概念。

Static Synchronized(全局锁)：该锁针对的是类的，无论实例化了多少个对象，那么线程都是共享这个锁的。

public Class Something{

public synchronized void sysA(){  }

public synchronized void sysB(){  }

public static synchronized void C\_sysA(){  }

public static synchronized void C\_sysB(){  }

}

现在假如Something有两个实例x和y，下面我们来分析下面几种情况获取锁的结果：

（1）x.sysA()和x.sysB()

（2）x.sysA()和y.sysB()

（3）x.C\_sysA()和y.C\_sysB()

（4）x.sysA()和Something.C\_sysA()

现在我们一一分析：

1、这两个方法是不能同时被调用的，因为使用的是同一个对象锁x。

2、这两个方法是可以同时被调用的，因为使用的不是同一个对象锁。

3、这两个方法是不能同时被调用的。因为C\_sysA()和C\_sysB()都使用的是全局锁。而x和y是属于同一个类（Something），所以这两个方法是共用一个锁的。

4、这两个方法是可以同时执行的。首先sysA()方法使用的是对象锁x，而C\_sysA()方法使用的是类的锁。所以使用的锁是不一样的。

-------------------------------------------------------------------------------

synchronized关键字的作用域有二种：

1）是某个对象实例内，synchronized aMethod(){}可以防止多个线程同时访问这个对象的synchronized方法（如果一个对象有多个synchronized方法，只要一个线程访问了其中的一个synchronized方法，其它线程不能同时访问这个对象中任何一个synchronized方法）。这时，不同的对象实例的synchronized方法是不相干扰的。也就是说，其它线程照样可以同时访问相同类的另一个对象实例中的synchronized方法；

2）是某个类的范围，synchronized static aStaticMethod{}防止多个线程中不同的实例对象（或者同一个实例对象）同时访问这个类中的synchronized static 方法。它可以对类的所有对象实例起作用。

3）除了方法前用synchronized关键字，synchronized关键字还可以用于方法中的某个区块中，表示只对这个区块的资源实行互斥访问。用法是: synchronized(this){/\*区块\*/}（或者synchronized(obj){/\*区块\*/}），它的作用域是当前对象/括号中指定对象

4）synchronized关键字是不能继承的，也就是说，基类的方法synchronized f(){} 在继承类中并不自动是synchronized f(){}，而是变成了f(){}。继承类需要你显式的指定它的某个方法为synchronized方法；

-------------------------------------------------------------------------------

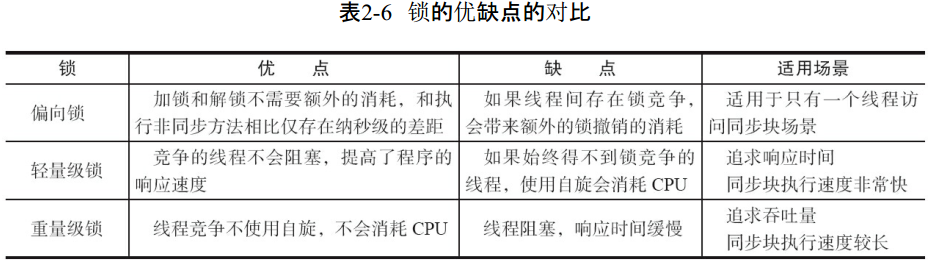
注意：

在使用synchronized关键字时候，应该尽可能避免在synchronized方法或synchronized块中使用sleep或者yield方法，因为synchronized程序块占有着对象锁，你休息那么其他的线程只能一边等着你醒来执行完了才能执行。不但严重影响效率，也不合逻辑。

同样，在同步程序块内调用yeild方法让出CPU资源也没有意义，因为你占用着锁，其他互斥线程还是无法访问同步程序块。当然与同步程序块无关的线程可以获得更多的执行时间。

-------------------------------------------------------------------------------

1. JavaSE 6 为了减少获得锁和释放锁带来的性能小号，引入了“偏向锁”和“轻量级锁“，在JavaSE 6 中，锁一共有4中状态，级别从低到高分别为：无锁状态、偏向锁状态、轻量级状态和重量级状态，这几个状态会随着竞争情况逐渐升级。锁可以升级但是不可以降级，意味着偏向锁升级成轻量级以后不能降级为偏向锁。这种只能升级不能降级的策略，目的是为了提高获得锁和释放锁的效率。偏向锁在Java 6和Java 7里是默认启用的，但是它在应用程序启动几秒钟之后才激活，如  
   有必要可以使用JVM参数来关闭延迟：-XX:BiasedLockingStartupDelay=0。如果你确定应用程  
   序里所有的锁通常情况下处于竞争状态，可以通过JVM参数关闭偏向锁：-XX:-  
   UseBiasedLocking=false，那么程序默认会进入轻量级锁状态。



1. 在Java并发包中有一些并发框架也使用了自旋CAS的方式来实现原子操作，比如LinkedTransferQueue类的Xfer方法。CAS虽然很高效地解决了原子操作，但是CAS仍然存在三大问题。ABA问题；时间长开销大，以及只能保证一个共享变量的原子操作。

-------------------------------------------------------------------------------

1）ABA问题。因为CAS需要在操作值的时候，检查值有没有发生变化，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。ABA问题的解决思路就是使用版本号。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1，那么A→B→A就会变成1A→2B→3A。从Java 1.5开始，JDK的Atomic包里提供了一个类AtomicStampedReference来解决ABA问题。这个类的compareAndSet方法的作用是首先检查当前引用是否等于预期引用，并且检查当前标志是否等于预期标志，如果全部相等，则以原子方式将该引用和该标志的值设置为给定的更新值。

public boolean compareAndSet(

V expectedReference, // 预期引用

V newReference, // 更新后的引用

int expectedStamp, // 预期标志

int newStamp // 更新后的标志

)

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。还有一个取巧的办法，就是把多个共享变量合并成一个共享变量来操作。比如，有两个共享变量i＝2，j=a，合并一下ij=2a，然后用CAS来操作ij。从Java 1.5开始，JDK提供了AtomicReference类来保证引用对象之间的原子性，就可以把多个变量放在一个对象里来进行CAS操作。

4）使用锁机制实现原子操作锁机制保证了只有获得锁的线程才能够操作锁定的内存区域。JVM内部实现了很多种锁机制，有偏向锁、轻量级锁和互斥锁。有意思的是除了偏向锁，JVM实现锁的方式都用了循环CAS，即当一个线程想进入同步块的时候使用循环CAS的方式来获取锁，当它退出同步块的时候使用循环CAS释放锁。

-------------------------------------------------------------------------------

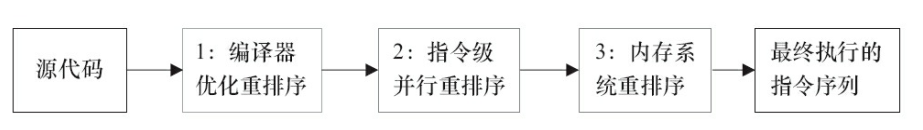
1. 在执行程序时，为了提高性能，编译器和处理器常常会对指令做重排序。重排序分3种类型。

a编译器优化的重排序。编译器在不改变单线程程序语义的前提下，可以重新安排语句的执行顺序。

b指令级并行的重排序。现代处理器采用了指令级并行技术（Instruction-LevelParallelism，ILP）来将多条指令重叠执行。如果不存在数据依赖性，处理器可以改变语句对应机器指令的执行顺序。

c内存系统的重排序。由于处理器使用缓存和读/写缓冲区，这使得加载和存储操作看上去可能是在乱序执行。

从Java源代码到最终实际执行的指令序列，会分别经历下面3种重排序，如图



1. 与程序员密切相关的happens-before规则如下：

·程序顺序规则：一个线程中的每个操作，happens-before于该线程中的任意后续操作。

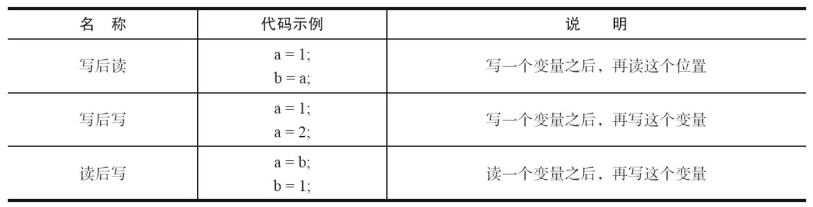
·监视器锁规则：对一个锁的解锁，happens-before于随后对这个锁的加锁。

·volatile变量规则：对一个volatile域的写，happens-before于任意后续对这个volatile域的读。

·传递性：如果A happens-before B，且B happens-before C，那么A happens-before C。

注意：两个操作之间具有happens-before关系，并不意味着前一个操作必须要在后一个操作之前执行！happens-before仅仅要求前一个操作（执行的结果）对后一个操作可见，且前一个操作按顺序排在第二个操作之前（the first is visible to and ordered before the second）。

1. 如果两个操作访问同一个变量，且这两个操作中有一个为写操作，此时这两个操作之间就存在数据依赖性。数据依赖分为下列3种类型，如表



上面3种情况，只要重排序两个操作的执行顺序，程序的执行结果就会被改变。

前面提到过，编译器和处理器可能会对操作做重排序。编译器和处理器在重排序时，会遵守数据依赖性，编译器和处理器不会改变存在数据依赖关系的两个操作的执行顺序。

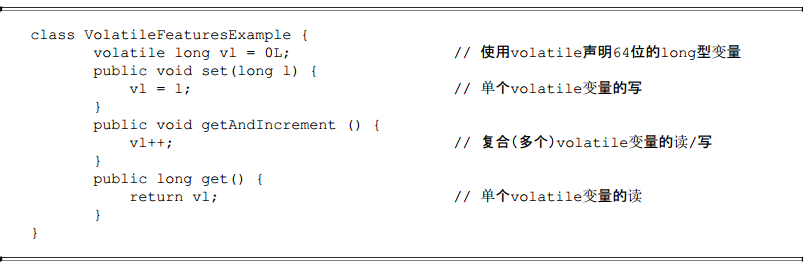
这里所说的数据依赖性仅针对单个处理器中执行的指令序列和单个线程中执行的操作，不同处理器之间和不同线程之间的数据依赖性不被编译器和处理器考虑。

-------------------------------------------------------------------------------

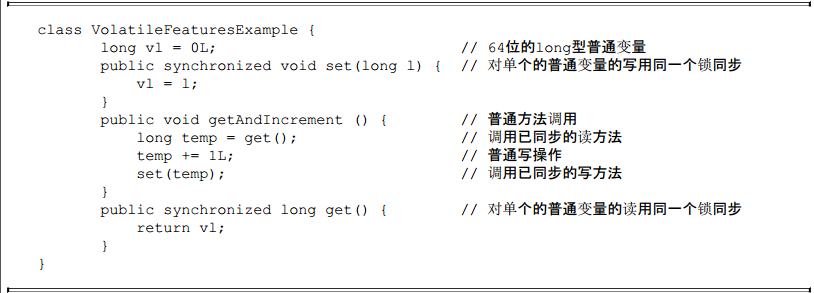
在单线程程序中，对存在控制依赖的操作重排序，不会改变执行结果（这也是as-if-serial语义允许对存在控制依赖的操作做重排序的原因）；但在多线程程序中，对存在控制依赖的操作重排序，可能会改变程序的执行结果

-------------------------------------------------------------------------------

1. 理解volatile特性的一个好方法是把对volatile变量的单个读/写，看成是使用同一个锁对这些单个读/写操作做了同步。下面通过具体的示例来说明，示例代码如下



假设有多个线程分别调用上面程序的3个方法，这个程序在语义上和下面程序等价。



如上面示例程序所示，一个volatile变量的单个读/写操作，与一个普通变量的读/写操作都是使用同一个锁来同步，它们之间的执行效果相同。锁的happens-before规则保证释放锁和获取锁的两个线程之间的内存可见性，这意味着对一个volatile变量的读，总是能看到（任意线程）对这个volatile变量最后的写入。

锁的语义决定了临界区代码的执行具有原子性。这意味着，即使是64位的long型和double型变量，只要它是volatile变量，对该变量的读/写就具有原子性。如果是多个volatile操作或类似于volatile++这种复合操作，这些操作整体上不具有原子性。

简而言之，volatile变量自身具有下列特性。

·可见性。对一个volatile变量的读，总是能看到（任意线程）对这个volatile变量最后的写入。

·原子性：对任意单个volatile变量的读/写具有原子性，但类似于volatile++这种复合操作不具有原子性。

-------------------------------------------------------------------------------

volatile写的内存语义如下：

当写一个volatile变量时，JMM会把该线程对应的本地内存中的共享变量值刷新到主内存。

volatile读的内存语义如下：

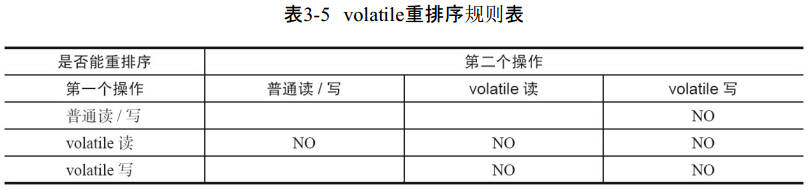
当读一个volatile变量时，JMM会把该线程对应的本地内存置为无效。线程接下来将从主内存中读取共享变量。

下面对volatile写和volatile读的内存语义做个总结：

·线程A写一个volatile变量，实质上是线程A向接下来将要读这个volatile变量的某个线程发出了（其对共享变量所做修改的）消息。

·线程B读一个volatile变量，实质上是线程B接收了之前某个线程发出的（在写这个volatile变量之前对共享变量所做修改的）消息。

·线程A写一个volatile变量，随后线程B读这个volatile变量，这个过程实质上是线程A通过主内存向线程B发送消息。



详情在Java并发编程与艺术3.4.4

-------------------------------------------------------------------------------

对比锁释放-获取的内存语义与volatile写-读的内存语义可以看出：锁释放与volatile写有相同的内存语义；锁获取与volatile读有相同的内存语义。

下面对锁释放和锁获取的内存语义做个总结。

·线程A释放一个锁，实质上是线程A向接下来将要获取这个锁的某个线程发出了（线程A对共享变量所做修改的）消息。

·线程B获取一个锁，实质上是线程B接收了之前某个线程发出的（在释放这个锁之前对共享变量所做修改的）消息。

·线程A释放锁，随后线程B获取这个锁，这个过程实质上是线程A通过主内存向线程B发送消息。

-------------------------------------------------------------------------------

1. 于final域，编译器和处理器要遵守两个重排序规则。

1）在构造函数内对一个final域的写入，与随后把这个被构造对象的引用赋值给一个引用变量，这两个操作之间不能重排序。

2）初次读一个包含final域的对象的引用，与随后初次读这个final域，这两个操作之间不能重排序。

-------------------------------------------------------------------------------

1. happens-before规则。

1）程序顺序规则：一个线程中的每个操作，happens-before于该线程中的任意后续操作。

2）监视器锁规则：对一个锁的解锁，happens-before于随后对这个锁的加锁。

3）volatile变量规则：对一个volatile域的写，happens-before于任意后续对这个volatile域的读。

4）传递性：如果A happens-before B，且B happens-before C，那么A happens-before C。

5）start()规则：如果线程A执行操作ThreadB.start()（启动线程B），那么A线程的ThreadB.start()操作happens-before于线程B中的任意操作。

6）join()规则：如果线程A执行操作ThreadB.join()并成功返回，那么线程B中的任意操作happens-before于线程A从ThreadB.join()操作成功返回。

-------------------------------------------------------------------------------

1. 初始化一个类，包括执行这个类的静态初始化和初始化在这个类中声明的静态字段。根据Java语言规范，在首次发生下列任意一种情况时，一个类或接口类型T将被立即初始化。

1）T是一个类，而且一个T类型的实例被创建。

2）T是一个类，且T中声明的一个静态方法被调用。

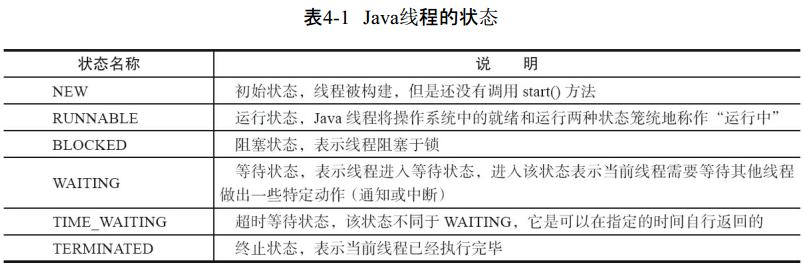
3）T中声明的一个静态字段被赋值。

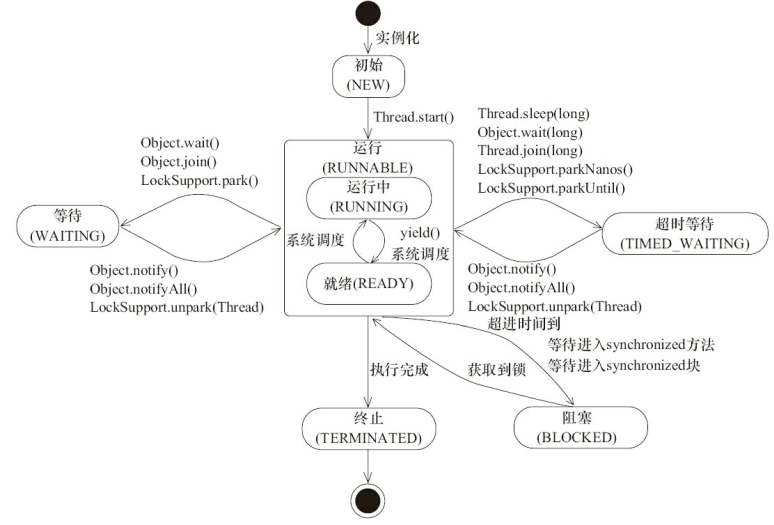
4）T中声明的一个静态字段被使用，而且这个字段不是一个常量字段。

5）T是一个顶级类（Top Level Class，见Java语言规范的§7.6），而且一个断言语句嵌套在T内部被执行

-------------------------------------------------------------------------------

1. 线程状态





Java将操作系统中的运行和就绪两个状态合并称为运行状态。阻塞状态是线程阻塞在进入synchronized关键字修饰的方法或代码块（获取锁）时的状态，但是阻塞在java.concurrent包中Lock接口的线程状态却是等待状态，因为java.concurrent包中Lock接口对于阻塞的实现均使用了LockSupport类中的相关方法。

-------------------------------------------------------------------------------

1. 守护线程

Daemon（守护）线程是一种支持型线程，因为它主要被用作程序中后台调度以及支持性工作。这意味着，当一个Java虚拟机中不存在非Daemon线程的时候，Java虚拟机将会退出。可以通过调用Thread.setDaemon(true)将线程设置为Daemon线程。

**注意**：Daemon属性需要在启动线程之前设置，不能在启动线程之后设置。Daemon线程被用作完成支持性工作，但是在Java虚拟机退出时Daemon线程中的finally块并不一定会执行。

main线程（非Daemon线程）在启动了线程DaemonRunner之后随着main方法执行完毕而终止，而此时Java虚拟机中已经没有非Daemon线程，虚拟机需要退出。Java虚拟机中的所有Daemon线程都需要立即终止，因此DaemonRunner立即终止，但是DaemonRunner中的finally块并没有执行。

**注意**：在构建Daemon线程时，不能依靠finally块中的内容来确保执行关闭或清理资源的逻辑。

------------------------------------------------------------------------------

1. 使用Thread.interrupt()中断线程

正如Listing A中所描述的，Thread.interrupt()方法不会中断一个正在运行的线程。这一方法实际上完成的是，在线程受到阻塞时抛出一个中断信号，这样线程就得以退出阻塞的状态。更确切的说，如果线程被Object.wait, Thread.join和Thread.sleep三种方法之一阻塞，那么，它将接收到一个中断异常（InterruptedException），从而提早地终结被阻塞状态。

因此，如果线程被上述几种方法阻塞，正确的停止线程方式是设置共享变量，并调用interrupt()（注意变量应该先设置）。如果线程没有被阻塞，这时调用interrupt()将不起作用；否则，线程就将得到异常（该线程必须事先预备好处理此状况），接着逃离阻塞状态。在任何一种情况中，最后线程都将检查共享变量然后再停止。

-------------------------------------------------------------------------------

1. join方法的作用

join()的作用是：“等待该线程终止”，这里需要理解的就是该线程是指的主线程等待子线程的终止。也就是在子线程调用了join()方法后面的代码，只有等到子线程结束了才能执行。

-------------------------------------------------------------------------------

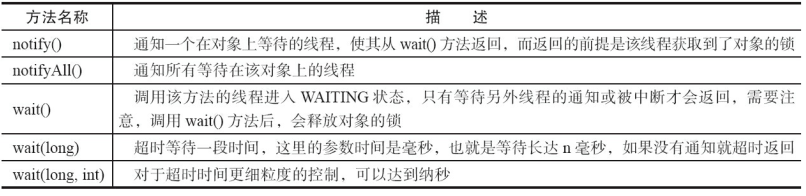
1. suspend()、resume()和stop()

suspend()、resume()和stop()方法完成了线程的暂停、恢复和终止工作，而且非常“人性化”。但是这些API是过期的，也就是不建议使用的。不建议使用的原因主要有：以suspend()方法为例，在调用后，线程不会释放已经占有的资源（比如锁），而是占有着资源进入睡眠状态，这样容易引发死锁问题。同样，stop()方法在终结一个线程时不会保证线程的资源正常释放，通常是没有给予线程完成资源释放工作的机会，因此会导致程序可能工作在不确定状态下。

**注意**：正因为suspend()、resume()和stop()方法带来的副作用，这些方法才被标注为不建议使用的过期方法，而暂停和恢复操作可以用后面提到的等待/通知机制来替代。

-------------------------------------------------------------------------------

1. wait()和notify()



注意：

1）wait方法会释放对象的锁。

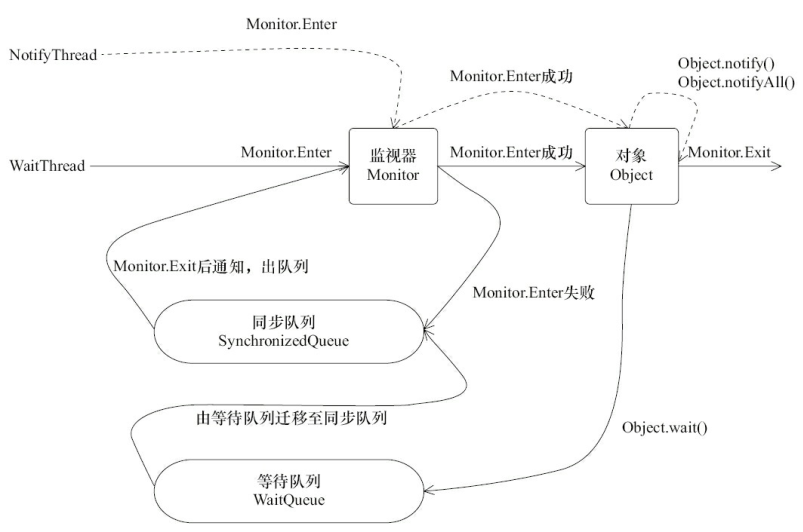
2）notify时不会释放lock的锁，直到当前线程释放了lock后，线程才能从wait方法中返回。

调用wait()、notify()以及notifyAll()时需要注意的细节，如下。

1）使用wait()、notify()和notifyAll()时需要先对调用对象加锁。

2）调用wait()方法后，线程状态由RUNNING变为WAITING，并将当前线程放置到对象的等待队列。

3）notify()或notifyAll()方法调用后，等待线程依旧不会从wait()返回，需要调用notify()或notifAll()的线程释放锁之后，等待线程才有机会从wait()返回。



WaitThread首先获取了对象的锁，然后调用对象的wait()方法，从而放弃了锁并进入了对象的等待队列WaitQueue中，进入等待状态。由于WaitThread释放了对象的锁，NotifyThread随后获取了对象的锁，并调用对象的notify()方法，将WaitThread从WaitQueue移到SynchronizedQueue中，此时WaitThread的状态变为阻塞状态。NotifyThread释放了锁之后，WaitThread再次获取到锁并从wait()方法返回继续执行。

WaitNotify示例中可以提炼出等待/通知的经典范式，该范式分为两部分，分别针对等待方（消费者）和通知方（生产者）。等待方遵循如下原则。

1）获取对象的锁。

2）如果条件不满足，那么调用对象的wait()方法，被通知后仍要检查条件。

3）条件满足则执行对应的逻辑。对应的伪代码如下。

synchronized(对象) {

while(条件不满足) {

对象.wait();

}

对应的处理逻辑

}

通知方遵循如下原则。

1）获得对象的锁。

2）改变条件。

3）通知所有等待在对象上的线程。对应的伪代码如下。

synchronized(对象) {

改变条件

对象.notifyAll();

}

1. 等待超时模式代码示例

前面的章节介绍了等待/通知的经典范式，即加锁、条件循环和处理逻辑3个步骤，而这种范式无法做到超时等待。而超时等待的加入，只需要对经典范式做出非常小的改动，改动内容如下。

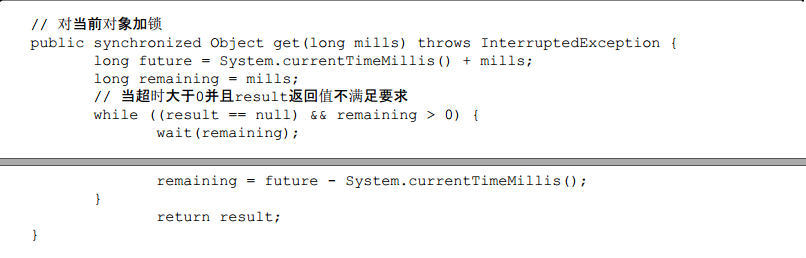
假设超时时间段是T，那么可以推断出在当前时间now+T之后就会超时。定义如下变量。

·等待持续时间：REMAINING=T。

·超时时间：FUTURE=now+T。

这时仅需要wait(REMAINING)即可，在wait(REMAINING)返回之后会将执行：REMAINING=FUTURE–now。如果REMAINING小于等于0，表示已经超时，直接退出，否则将继续执行wait(REMAINING)。wait(REMAINING)可通过notify()唤醒 或 notifyAll()唤醒，或超过3000ms延时；然后才被唤醒。

上述描述等待超时模式的伪代码如下。

可以看出，等待超时模式就是在等待/通知范式基础上增加了超时控制，这使得该模式相比原有范式更具有灵活性，因为即使方法执行时间过长，也不会“永久”阻塞调用者，而是会按照调用者的要求“按时”返回。

-------------------------------------------------------------------------------

1. CountDownLatch是什么

CountDownLatch是在java1.5被引入的，跟它一起被引入的并发工具类还有CyclicBarrier、Semaphore、[ConcurrentHashMap](http://howtodoinjava.com/2013/05/27/best-practices-for-using-concurrenthashmap/)和[BlockingQueue](http://howtodoinjava.com/2012/10/20/how-to-use-blockingqueue-and-threadpoolexecutor-in-java/)，它们都存在于java.util.concurrent包下。CountDownLatch这个类能够***使一个线程等待其他线程完成各自的工作后再执行***。

CountDownLatch如何工作

CountDownLatch.java类中定义的构造函数：

//Constructs a CountDownLatch initialized with the given count.

public void CountDownLatch(int count) {...}

构造器中的计数值（count）实际上就是闭锁需要等待的线程数量。这个值只能被设置一次，而且CountDownLatch没有提供任何机制去重新设置这个计数值。

与CountDownLatch的第一次交互是主线程等待其他线程。主线程必须在启动其他线程后立即调用CountDownLatch.await()方法。这样主线程的操作就会在这个方法上阻塞，直到其他线程完成各自的任务。

其他N 个线程必须引用闭锁对象，因为他们需要通知CountDownLatch对象，他们已经完成了各自的任务。这种通知机制是通过 CountDownLatch.countDown()方法来完成的；每调用一次这个方法，在构造函数中初始化的count值就减1。所以当N个线程都调 用了这个方法，count的值等于0，然后主线程就能通过await()方法，恢复执行自己的任务。

在实时系统中的使用场景:

让我们尝试罗列出在java实时系统中CountDownLatch都有哪些使用场景。我所罗列的都是我所能想到的。如果你有别的可能的使用方法，请在留言里列出来，这样会帮助到大家。

1实现最大的并行性：有时我们想同时启动多个线程，实现最大程度的并行性。例如，我们想测试一个单例类。如果我们创建一个初始计数为1的CountDownLatch，并让所有线程都在这个锁上等待，那么我们可以很轻松地完成测试。我们只需调用 一次countDown()方法就可以让所有的等待线程同时恢复执行。

2开始执行前等待n个线程完成各自任务：例如应用程序启动类要确保在处理用户请求前，所有N个外部系统已经启动和运行了。

3死锁检测：一个非常方便的使用场景是，你可以使用n个线程访问共享资源，在每次测试阶段的线程数目是不同的，并尝试产生死锁。

1. 线程池中线程数量并不是越多越好，具体的数量需要评估每个任务的处理时间，以及当前计算机的处理器能力和数量。使用的线程过少，无法发挥处理器的性能；使用的线程过多，将会增加系统的无故开销，起到相反的作用.
2. 在finally块中释放锁，目的是保证在获取到锁之后，最终能够被释放。不要将获取锁的过程写在try块中，因为如果在获取锁（自定义锁的实现）时发生了异常，异常抛出的同时，也会导致锁无故释放。



1. 重入锁

重入锁ReentrantLock，顾名思义，就是支持重进入的锁，它表示该锁能够支持一个线程对资源的重复加锁。除此之外，该锁的还支持获取锁时的公平和非公平性选择。

ReentrantLock虽然没能像synchronized关键字一样支持隐式的重进入，但是在调用lock()方法时，已经获取到锁的线程，能够再次调用lock()方法获取锁而不被阻塞。

这里提到一个锁获取的公平性问题，如果在绝对时间上，先对锁进行获取的请求一定先被满足，那么这个锁是公平的，反之，是不公平的。公平的获取锁，也就是等待时间最长的线程最优先获取锁，也可以说锁获取是顺序的。ReentrantLock提供了一个构造函数，能够控制锁是否是公平的。事实上，公平的锁机制往往没有非公平的效率高，但是，并不是任何场景都是以TPS作为唯一的指标，公平锁能够减少“饥饿”发生的概率，等待越久的请求越是能够得到优先满足。

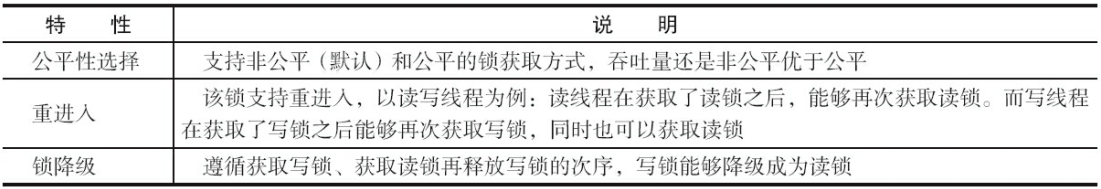
当一个线程请求锁时，只要获取了同步状态即成功获取锁。在这个前提下，刚释放锁的线程再次获取同步状态的几率会非常大，使得其他线程只能在同步队列中等待。

非公平性锁可能使线程“饥饿”，为什么它又被设定成默认的实现呢？再次观察上表的结果，如果把每次不同线程获取到锁定义为1次切换，公平性锁在线程切换的次数上较非公平性锁切换的次数更多，这说明非公平性锁的开销更小，因为它减小了线程切换的开销。

1. 读写锁ReentrantReadWriteLock

读写锁在同一时刻可以允许多个读线程访问，但是在写线程访问时，所有的读线程和其他写线程均被阻塞。读写锁维护了一对锁，一个读锁和一个写锁，通过分离读锁和写锁，使得并发性相比一般的排他锁有了很大提升。

读写锁ReentrantReadWriteLock特性如下：



写锁是一个支持重进入的排它锁。如果当前线程已经获取了写锁，则增加写状态。如果当前线程在获取写锁时，读锁已经被获取（读状态不为0）或者该线程不是已经获取写锁的线程，则当前线程进入等待状态。

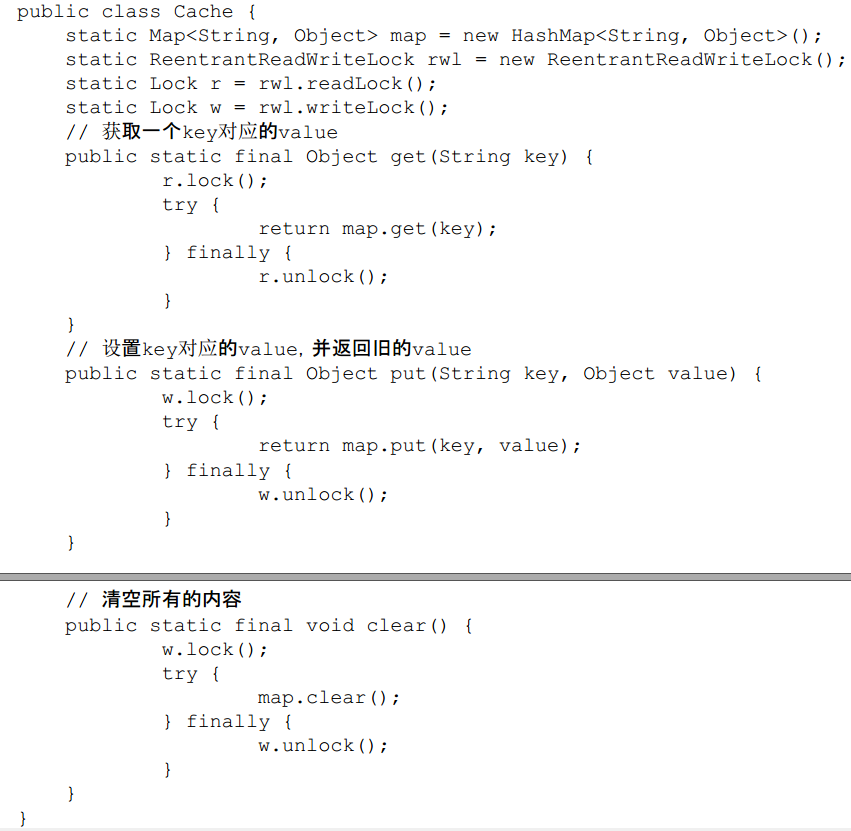
读锁是一个支持重进入的共享锁，它能够被多个线程同时获取，在没有其他写线程访问（或者写状态为0）时，读锁总会被成功地获取，而所做的也只是（线程安全的）增加读状态。如果当前线程已经获取了读锁，则增加读状态。如果当前线程在获取读锁时，写锁已被其他线程获取，则进入等待状态

锁降级指的是写锁降级成为读锁。如果当前线程拥有写锁，然后将其释放，最后再获取读锁，这种分段完成的过程不能称之为锁降级。锁降级是指把持住（当前拥有的）写锁，再获取到读锁，随后释放（先前拥有的）写锁的过程。

锁降级中读锁的获取是否必要呢？答案是必要的。主要是为了保证数据的可见性，如果当前线程不获取读锁而是直接释放写锁，假设此刻另一个线程（记作线程T）获取了写锁并修改了数据，那么当前线程无法感知线程T的数据更新。如果当前线程获取读锁，即遵循锁降级的步骤，则线程T将会被阻塞，直到当前线程使用数据并释放读锁之后，线程T才能获取写锁进行数据更新。

RentrantReadWriteLock不支持锁升级（把持读锁、获取写锁，最后释放读锁的过程）。目的也是保证数据可见性，如果读锁已被多个线程获取，其中任意线程成功获取了写锁并更新了数据，则其更新对其他获取到读锁的线程是不可见的。

代码示例：



上述示例中，Cache组合一个非线程安全的HashMap作为缓存的实现，同时使用读写锁的读锁和写锁来保证Cache是线程安全的。在读操作get(String key)方法中，需要获取读锁，这使得并发访问该方法时不会被阻塞。写操作put(String key,Object value)方法和clear()方法，在更新HashMap时必须提前获取写锁，当获取写锁后，其他线程对于读锁和写锁的获取均被阻塞，而只有写锁被释放之后，其他读写操作才能继续。Cache使用读写锁提升读操作的并发性，也保证每次写操作对所有的读写操作的可见性，同时简化了编程方式。

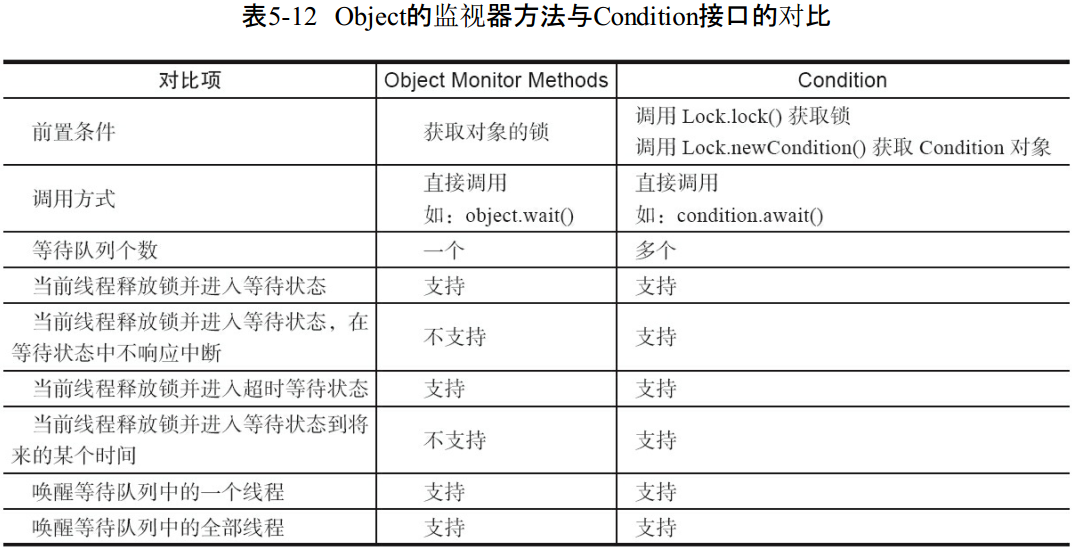
1. LockSupport线程阻塞

在Java 6中，LockSupport增加了park(Object blocker)、parkNanos(Object blocker,long nanos)和parkUntil(Object blocker,long deadline)3个方法，用于实现阻塞当前线程的功能，其中参数blocker是用来标识当前线程在等待的对象（以下称为阻塞对象），该对象主要用于问题排查和系统监控。对比Java 5推出的parkNanos(long nanos)方法和parkNanos(Object blocker,long nanos)方法由于在Java 5之前，当线程阻塞（使用synchronized关键字）在一个对象上时，通过线程dump能够查看到该线程的阻塞对象，方便问题定位，而Java 5推出的Lock等并发工具时却遗漏了这一点，致使在线程dump时无法提供阻塞对象的信息。因此，在Java 6中，LockSupport新增了上述3个含有阻塞对象的park方法，用以替代原有的park方法。

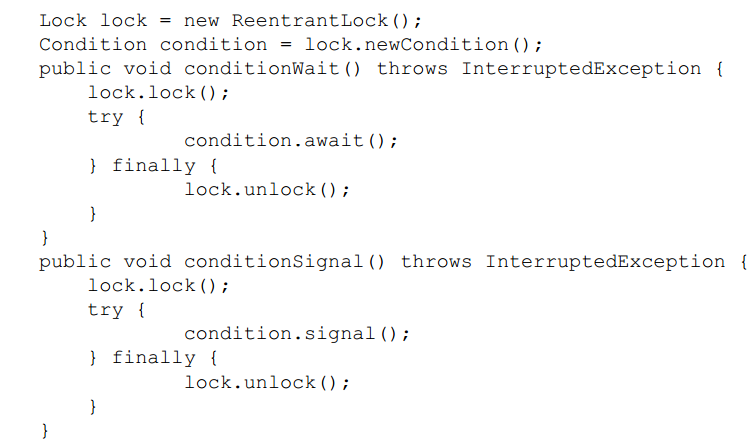
1. Condition接口

任意一个Java对象，都拥有一组监视器方法（定义在java.lang.Object上），主要包括wait()、wait(long timeout)、notify()以及notifyAll()方法，这些方法与synchronized同步关键字配合，可以实现等待/通知模式。Condition接口也提供了类似Object的监视器方法，与Lock配合可以实现等待/通知模式，但是这两者在使用方式以及功能特性上还是有差别的。

通过对比Object的监视器方法和Condition接口，可以更详细地了解Condition的特性，对比项与结果如表所示。



Condition定义了等待/通知两种类型的方法，当前线程调用这些方法时，需要提前获取到Condition对象关联的锁。Condition对象是由Lock对象（调用Lock对象的newCondition()方法）创建出来的，换句话说，Condition是依赖Lock对象的，Condition的使用方式比较简单，需要注意在调用方法前获取锁。

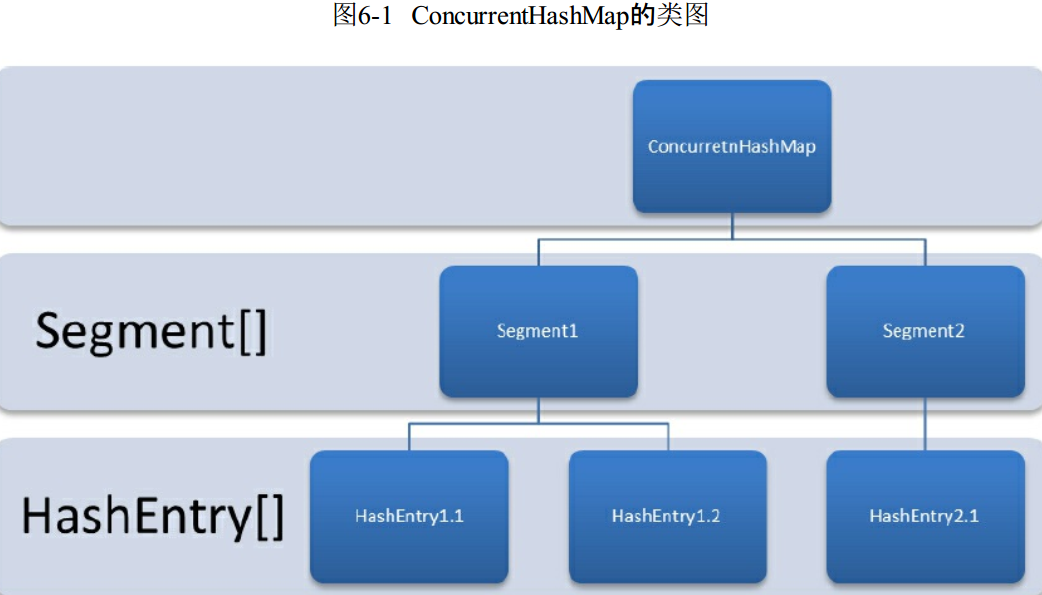


如示例所示，一般都会将Condition对象作为成员变量。当调用await()方法后，当前线程会释放锁并在此等待，而其他线程调用Condition对象的signal()方法，通知当前线程后，当前线程才从await()方法返回，并且在返回前已经获取了锁。

1. ConcurrentHashMap

HashTable容器在竞争激烈的并发环境下表现出效率低下的原因是所有访问HashTable的线程都必须竞争同一把锁，假如容器里有多把锁，每一把锁用于锁容器其中一部分数据，那么当多线程访问容器里不同数据段的数据时，线程间就不会存在锁竞争，从而可以有效提高并发访问效率，这就是JDK5~JDK7 ConcurrentHashMap所使用的锁分段技术。首先将数据分成一段一段地存储，然后给每一段数据配一把锁，当一个线程占用锁访问其中一个段数据的时候，其他段的数据也能被其他线程访问。

JDK5~JDK7 ConcurrentHashMap是由Segment数组结构和HashEntry数组结构组成。Segment是一种可重入锁（ReentrantLock），在ConcurrentHashMap里扮演锁的角色；HashEntry则用于存储键值对数据。一个ConcurrentHashMap里包含一个Segment数组。Segment的结构和HashMap类似，是一种数组和链表结构。一个Segment里包含一个HashEntry数组，每个HashEntry是一个链表结构的元素，每个Segment守护着一个HashEntry数组里的元素，当对HashEntry数组的数据进行修改时，必须首先获得与它对应的Segment锁。



ConcurrentHashMap的get操作的高效之处在于整个get过程不需要加锁，除非读到的值是空才会加锁重读。我们知道HashTable容器的get方法是需要加锁的，那么ConcurrentHashMap的get操作是如何做到不加锁的呢？原因是它的get方法里将要使用的共享变量都定义成volatile类型，如用于统计当前Segement大小的count字段和用于存储值的HashEntry的value。定义成volatile的变量，能够在线程之间保持可见性，能够被多线程同时读，并且保证不会读到过期的值，但是只能被单线程写（有一种情况可以被多线程写，就是写入的值不依赖于原值），在get操作里只需要读不需要写共享变量count和value，所以可以不用加锁。之所以不会读到过期的值，是因为根据Java内存模型的happen before原则，对volatile字段的写入操作先于读操作，即使两个线程同时修改和获取volatile变量，get操作也能拿到最新的值，这是用volatile替换锁的经典应用场景。

put方法里需要对共享变量进行写入操作，所以为了线程安全，在操作共享变量时必须加锁。put方法首先定位到Segment，然后在Segment里进行插入操作。插入操作需要经历两个步骤，第一步判断是否需要对Segment里的HashEntry数组进行扩容，第二步定位添加元素的位置，然后将其放在HashEntry数组里。

在扩容的时候，首先会创建一个容量是原来容量两倍的数组，然后将原数组里的元素进行再散列后插入到新的数组里。为了高效，ConcurrentHashMap不会对整个容器进行扩容，而只对某个segment进行扩容。

如果要统计整个ConcurrentHashMap里元素的大小，就必须统计所有Segment里元素的大小后求和。Segment里的全局变量count是一个volatile变量，那么在多线程场景下，是不是直接把所有Segment的count相加就可以得到整个ConcurrentHashMap大小了呢？不是的，虽然相加时可以获取每个Segment的count的最新值，但是可能累加前使用的count发生了变化，那么统计结果就不准了。所以，最安全的做法是在统计size的时候把所有Segment的put、remove和clean方法全部锁住，但是这种做法显然非常低效。

因为在累加count操作过程中，之前累加过的count发生变化的几率非常小，所以ConcurrentHashMap的做法是先尝试2次通过不锁住Segment的方式来统计各个Segment大小，如果统计的过程中，容器的count发生了变化，则再采用加锁的方式来统计所有Segment的大小。

那么ConcurrentHashMap是如何判断在统计的时候容器是否发生了变化呢？使用modCount变量，在put、remove和clean方法里操作元素前都会将变量modCount进行加1，那么在统计size前后比较modCount是否发生变化，从而得知容器的大小是否发生变化。

-------------------------------------------------------------------------------

在JDK1.8中ConcurrentHashMap摒弃了Segment（锁段）的概念，而是启用了一种全新的方式实现，同HashMap一样底层采用数组+链表+红黑树的存储结构,利用CAS算法实现无锁并发，从而大大提高了ConcurrentHashMap的整体性能

ConcurrentHashMap的第一次初始化并不是在创建队形的时候，而是第一次执行put操作得时候。并且在某节点的连表长度大于8时，并不会首先考虑链表转红黑树，而是首先判断map长度是否小于某一阈值（64），如果小于此值，则进行扩容，而不是链表转红黑树。若大于此值则链表转红黑树。

有一个最重要的不同点就是ConcurrentHashMap不允许key或value为null值。另外由于涉及到多线程，put方法就要复杂一点。在多线程中可能有以下两个情况：

1）如果一个或多个线程正在对ConcurrentHashMap进行扩容操作，当前线程也要进入扩容的操作中。这个扩容的操作之所以能被检测到，是因为transfer方法中在空结点上插入forward节点，如果检测到需要插入的位置被forward节点占有，就帮助进行扩容；

2）如果检测到要插入的节点是非空且不是forward节点，就对这个节点加锁，这样就保证了线程安全。尽管这个有一些影响效率，但是还是会比hashTable的synchronized要好得多。

putIfAbsent方法与put方法：put方法在面对插入存在的key/value的时候会直接覆盖原值，并返回原值。putIfAbsent方法在面对插入存在的key/value的时候不会覆盖原来的值，并且返回现有的值。

-------------------------------------------------------------------------------

1. ConcurrentLinkedQueue

ConcurrentLinkedQueue是一个基于链接节点的无界线程安全队列，它采用先进先出的规则对节点进行排序，当我们添加一个元素的时候，它会添加到队列的尾部；当我们获取一个元素时，它会返回队列头部的元素。它采用了CAS算法来实现。

ConcurrentLinkedQueue由head节点和tail节点组成，每个节点（Node）由节点元素（item）和指向下一个节点（next）的引用组成，节点与节点之间就是通过这个next关联起来，从而组成一张链表结构的队列。默认情况下head节点存储的元素为空，tail节点等于head节点。

从单线程的角度入队主要做两件事情：

第一是将入队节点设置成当前队列尾节点的下一个节点；

第二是更新tail节点，如果tail节点的next节点不为空，则将入队节点设置成tail节点，如果tail节点的next节点为空，则将入队节点设置成tail的next节点，所以tail节点不总是尾节点

从源代码角度来看，整个入队过程主要做两件事情：第一是定位出尾节点；第二是使用CAS算法将入队节点设置成尾节点的next节点，如不成功则重试。

让tail节点永远作为队列的尾节点，这样实现代码量非常少，而且逻辑清晰和易懂。但是，这么做有个缺点，每次都需要使用循环CAS更新tail节点。如果能减少CAS更新tail节点的次数，就能提高入队的效率，所以doug lea使用hops变量来控制并减少tail节点的更新频率，并不是每次节点入队后都将tail节点更新成尾节点，而是当tail节点和尾节点的距离大于等于常量HOPS的值（默认等于1）时才更新tail节点，tail和尾节点的距离越长，使用CAS更新tail节点的次数就会越少，但是距离越长带来的负面效果就是每次入队时定位尾节点的时间就越长，因为循环体需要多循环一次来定位出尾节点，但是这样仍然能提高入队的效率，因为从本质上来看它通过增加对volatile变量的读操作来减少对volatile变量的写操作，而对volatile变量的写操作开销要远远大于读操作，所以入队效率会有所提升。

注意：入队方法永远返回true，所以不要通过返回值判断入队是否成功。

并不是每次出队时都更新head节点，当head节点里有元素时，直接弹出head节点里的元素，而不会更新head节点。只有当head节点里没有元素时，出队操作才会更新head节点。这种做法也是通过hops变量来减少使用CAS更新head节点的消耗，从而提高出队效率。

通过源码可知，出队时首先获取头节点的元素，然后判断头节点元素是否为空，如果为空，表示另外一个线程已经进行了一次出队操作将该节点的元素取走，如果不为空，则使用CAS的方式将头节点的引用设置成null，如果CAS成功，则直接返回头节点的元素，如果不成功，表示另外一个线程已经进行了一次出队操作更新了head节点，导致元素发生了变化，需要重新获取头节点。

-------------------------------------------------------------------------------

1. JDK 7提供了7个阻塞队列，如下。

·ArrayBlockingQueue：一个由数组结构组成的有界阻塞队列。此队列按照先进先出（FIFO）的原则对元素进行排序。默认情况下不保证线程公平的访问队列。

·LinkedBlockingQueue：一个由链表结构组成的有界阻塞队列。此队列的默认和最大长度为Integer.MAX\_VALUE。此队列按照先进先出的原则对元素进行排序。

·PriorityBlockingQueue：一个支持优先级排序的无界阻塞队列。默认情况下元素采取自然顺序升序排列。也可以自定义类实现compareTo()方法来指定元素排序规则，或者初始化PriorityBlockingQueue时，指定构造参数Comparator来对元素进行排序。需要注意的是不能保证同优先级元素的顺序。

·DelayQueue：一个使用优先级队列实现的无界阻塞队列。DelayQueue是一个支持延时获取元素的无界阻塞队列。队列使用PriorityQueue来实现。队列中的元素必须实现Delayed接口，在创建元素时可以指定多久才能从队列中获取当前元素。只有在延迟期满时才能从队列中提取元素。DelayQueue非常有用，可以将DelayQueue运用在以下应用场景。

·缓存系统的设计：可以用DelayQueue保存缓存元素的有效期，使用一个线程循环查询DelayQueue，一旦能从DelayQueue中获取元素时，表示缓存有效期到了。

·定时任务调度：使用DelayQueue保存当天将会执行的任务和执行时间，一旦从DelayQueue中获取到任务就开始执行，比如TimerQueue就是使用DelayQueue实现的。

·SynchronousQueue：一个不存储元素的阻塞队列。每一个put操作必须等待一个take操作，否则不能继续添加元素。它支持公平访问队列。默认情况下线程采用非公平性策略访问队列。

·LinkedTransferQueue：一个由链表结构组成的无界阻塞队列。相对于其他阻塞队列，LinkedTransferQueue多了tryTransfer和transfer方法。

transfer方法

如果当前有消费者正在等待接收元素（消费者使用take()方法或带时间限制的poll()方法时），transfer方法可以把生产者传入的元素立刻transfer（传输）给消费者。如果没有消费者在等待接收元素，transfer方法会将元素存放在队列的tail节点，并等到该元素被消费者消费了才返回。

tryTransfer方法

tryTransfer方法是用来试探生产者传入的元素是否能直接传给消费者。如果没有消费者等待接收元素，则返回false。和transfer方法的区别是tryTransfer方法无论消费者是否接收，方法立即返回，而transfer方法是必须等到消费者消费了才返回。对于带有时间限制的tryTransfer（E e，long timeout，TimeUnit unit）方法，试图把生产者传入的元素直接传给消费者，但是如果没有消费者消费该元素则等待指定的时间再返回，如果超时还没消费元素，则返回false，如果在超时时间内消费了元素，则返回true。

·LinkedBlockingDeque：一个由链表结构组成的双向阻塞队列。所谓双向队列指的是可以从队列的两端插入和移出元素。双向队列因为多了一个操作队列的入口，在多线程同时入队时，也就减少了一半的竞争。相比其他的阻塞队列，LinkedBlockingDeque多了addFirst、addLast、offerFirst、offerLast、peekFirst和peekLast等方法，以First单词结尾的方法，表示插入、获取（peek）或移除双端队列的第一个元素。以Last单词结尾的方法，表示插入、获取或移除双端队列的最后一个元素。在初始化LinkedBlockingDeque时可以设置容量防止其过度膨胀。另外，双向阻塞队列可以运用在“工作窃取”模式中。

1. Fork/Join框架

Fork/Join框架是Java 7提供的一个用于并行执行任务的框架，是一个把大任务分割成若干个小任务，最终汇总每个小任务结果后得到大任务结果的框架。

工作窃取（work-stealing）算法是指某个线程从其他队列里窃取任务来执行。那么，为什么需要使用工作窃取算法呢？假如我们需要做一个比较大的任务，可以把这个任务分割为若干互不依赖的子任务，为了减少线程间的竞争，把这些子任务分别放到不同的队列里，并为每个队列创建一个单独的线程来执行队列里的任务，线程和队列一一对应。比如A线程负责处理A队列里的任务。但是，有的线程会先把自己队列里的任务干完，而其他线程对应的队列里还有任务等待处理。干完活的线程与其等着，不如去帮其他线程干活，于是它就去其他线程的队列里窃取一个任务来执行。而在这时它们会访问同一个队列，所以为了减少窃取任务线程和被窃取任务线程之间的竞争，通常会使用双端队列，被窃取任务线程永远从双端队列的头部拿任务执行，而窃取任务的线程永远从双端队列的尾部拿任务执行。

工作窃取算法的优点：充分利用线程进行并行计算，减少了线程间的竞争。

工作窃取算法的缺点：在某些情况下还是存在竞争，比如双端队列里只有一个任务时。并且该算法会消耗了更多的系统资源，比如创建多个线程和多个双端队列。

ForkJoinTask在执行的时候可能会抛出异常，但是我们没办法在主线程里直接捕获异常，所以ForkJoinTask提供了isCompletedAbnormally()方法来检查任务是否已经抛出异常或已经被取消了，并且可以通过ForkJoinTask的getException方法获取异常。使用如下代码。

if(task.isCompletedAbnormally())

{

System.out.println(task.getException());

}

getException方法返回Throwable对象，如果任务被取消了则返回CancellationException。如果任务没有完成或者没有抛出异常则返回null。

1. Java中的源自操作类

Java从JDK 1.5开始提供了java.util.concurrent.atomic包（以下简称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。

因为变量的类型有很多种，所以在Atomic包里一共提供了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）。Atomic包里的类基本都是使用Unsafe实现的包装类。

使用原子的方式更新基本类型，Atomic包提供了以下3个类。

·AtomicBoolean：原子更新布尔类型。

·AtomicInteger：原子更新整型。

·AtomicLong：原子更新长整型。

Atomic包提供了3种基本类型的原子更新，但是Java的基本类型里还有char、float和double等。那么问题来了，如何原子的更新其他的基本类型呢？Atomic包里的类基本都是使用Unsafe实现的，看一下Unsafe的源码我们发现Unsafe只提供了3种CAS方法：compareAndSwapObject、compareAndSwapInt和compareAndSwapLong，再看AtomicBoolean源码，发现它是先把Boolean转换成整型，再使用compareAndSwapInt进行CAS，所以原子更新char、float和double变量也可以用类似的思路来实现。

通过原子的方式更新数组里的某个元素，Atomic包提供了以下3个类。

·AtomicIntegerArray：原子更新整型数组里的元素。

·AtomicLongArray：原子更新长整型数组里的元素。

·AtomicReferenceArray：原子更新引用类型数组里的元素。

原子更新基本类型的AtomicInteger，只能更新一个变量，如果要原子更新多个变量，就需要使用这个原子更新引用类型提供的类。Atomic包提供了以下3个类。

·AtomicReference：原子更新引用类型。

·AtomicReferenceFieldUpdater：原子更新引用类型里的字段。

·AtomicMarkableReference：原子更新带有标记位的引用类型。可以原子更新一个布尔类型的标记位和引用类型。构造方法是AtomicMarkableReference（V initialRef，booleaninitialMark）。

如果需原子地更新某个类里的某个字段时，就需要使用原子更新字段类，Atomic包提供了以下3个类进行原子字段更新。

·AtomicIntegerFieldUpdater：原子更新整型的字段的更新器。

·AtomicLongFieldUpdater：原子更新长整型字段的更新器。

·AtomicStampedReference：原子更新带有版本号的引用类型。该类将整数值与引用关联起来，可用于原子的更新数据和数据的版本号，可以解决使用CAS进行原子更新时可能出现的ABA问题。

要想原子地更新字段类需要两步。第一步，因为原子更新字段类都是抽象类，每次使用的时候必须使用静态方法newUpdater()创建一个更新器，并且需要设置想要更新的类和属性。第二步，更新类的字段（属性）必须使用public volatile修饰符。

1. 同步屏障CyclicBarrier

CyclicBarrier的字面意思是可循环使用（Cyclic）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运行。

CyclicBarrier默认的构造方法是CyclicBarrier（int parties），其参数表示屏障拦截的线程数量，每个线程调用await方法告诉CyclicBarrier我已经到达了屏障，然后当前线程被阻塞。当达到屏障的线程数量和CyclicBarrier设定的屏障数量相等的时候，将同时释放所有屏障，线程继续执行。

CyclicBarrier还提供一个更高级的构造函数CyclicBarrier（int parties，Runnable barrierAction），用于在线程到达屏障时，优先执行barrierAction，方便处理更复杂的业务场景。

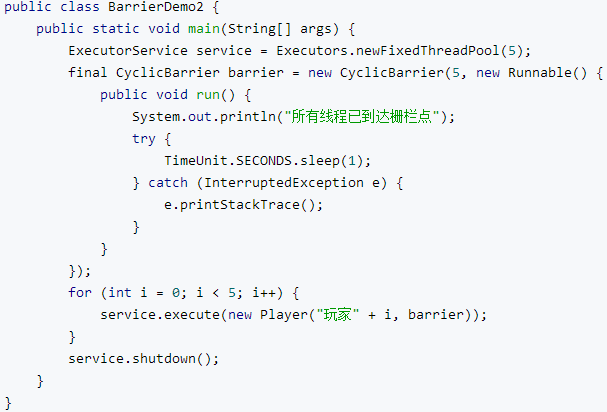
CyclicBarrier的应用场景：

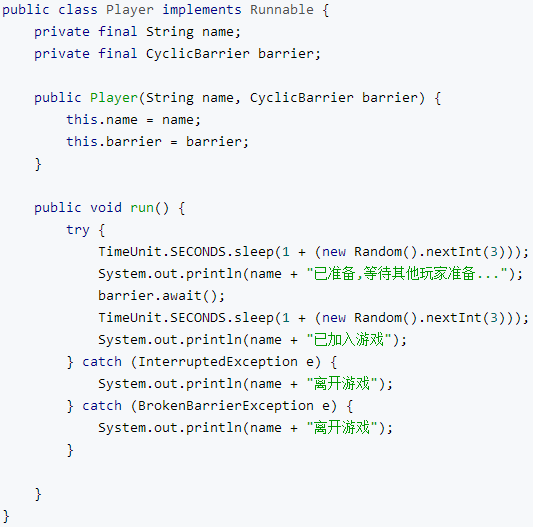
1. CyclicBarrier可以用于多线程计算数据，最后合并计算结果的场景。例如，用一个Excel保存了用户所有银行流水，每个Sheet保存一个账户近一年的每笔银行流水，现在需要统计用户的日均银行流水，先用多线程处理每个sheet里的银行流水，都执行完之后，得到每个sheet的日均银行流水，最后，再用barrierAction用这些线程的计算结果，计算出整个Excel的日均银行流水。





1. 对战平台中玩家需要完全准备好了,才能进入游戏的场景。





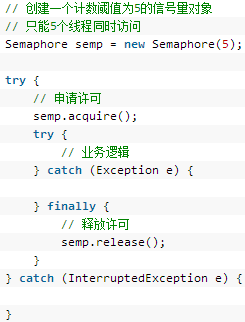
CountDownLatch的计数器只能使用一次，而CyclicBarrier的计数器可以使用reset()方法重置。所以CyclicBarrier能处理更为复杂的业务场景。例如，如果计算发生错误，可以重置计数器，并让线程重新执行一次。CyclicBarrier还提供其他有用的方法，比如getNumberWaiting方法可以获得Cyclic-Barrier阻塞的线程数量。isBroken()方法用来了解阻塞的线程是否被中断。

1. 控制并发线程数的Semaphore

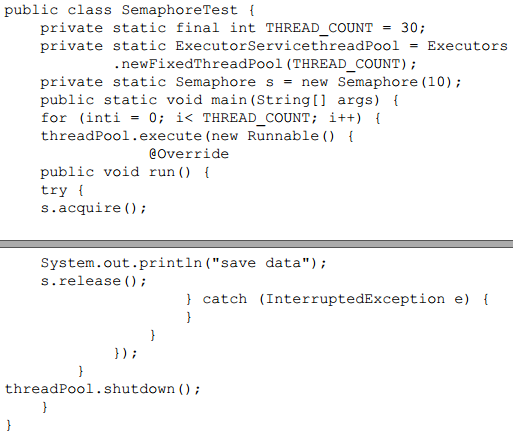
Semaphore（信号量）是用来控制同时访问特定资源的线程数量，它通过协调各个线程，以保证合理的使用公共资源。

Semaphore可以用于做流量控制，特别是公用资源有限的应用场景，比如数据库连接。假如有一个需求，要读取几万个文件的数据，因为都是IO密集型任务，我们可以启动几十个线程并发地读取，但是如果读到内存后，还需要存储到数据库中，而数据库的连接数只有10个，这时我们必须控制只有10个线程同时获取数据库连接保存数据，否则会报错无法获取数据库连接。这个时候，就可以使用Semaphore来做流量控制。

它的用法如下：



具体实例代码：



在代码中，虽然有30个线程在执行，但是只允许10个并发执行。Semaphore的构造方法Semaphore（int permits）接受一个整型的数字，表示可用的许可证数量。Semaphore（10）表示允许10个线程获取许可证，也就是最大并发数是10。Semaphore的用法也很简单，首先线程使用Semaphore的acquire()方法获取一个许可证，使用完之后调用release()方法归还许可证。还可以用tryAcquire()方法尝试获取许可证。

Semaphore还提供一些其他方法，具体如下。

·intavailablePermits()：返回此信号量中当前可用的许可证数。

·intgetQueueLength()：返回正在等待获取许可证的线程数。

·booleanhasQueuedThreads()：是否有线程正在等待获取许可证。

·void reducePermits（int reduction）：减少reduction个许可证，是个protected方法。

·Collection getQueuedThreads()：返回所有等待获取许可证的线程集合，是个protected方法。

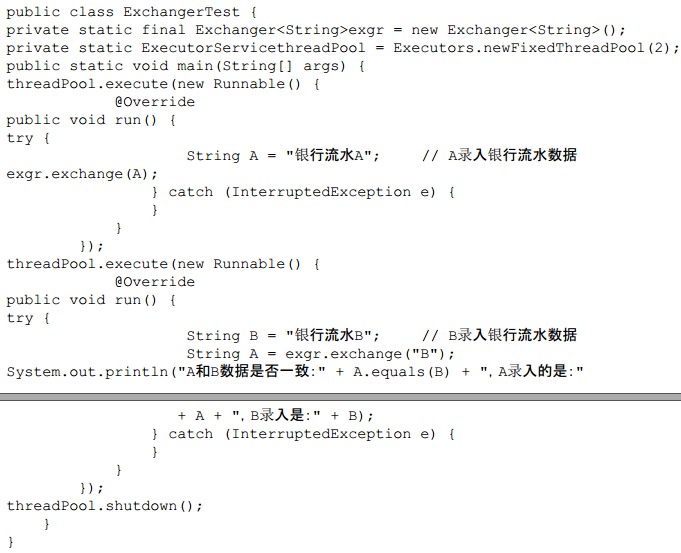
1. 线程间交换数据的Exchanger

Exchanger（交换者）是一个用于线程间协作的工具类。Exchanger用于进行线程间的数据交换。它提供一个同步点，在这个同步点，两个线程可以交换彼此的数据。这两个线程通过exchange方法交换数据，如果第一个线程先执行exchange()方法，它会一直等待第二个线程也执行exchange方法，当两个线程都到达同步点时，这两个线程就可以交换数据，将本线程生产出来的数据传递给对方。

下面来看一下Exchanger的应用场景：

Exchanger可以用于遗传算法，遗传算法里需要选出两个人作为交配对象，这时候会交换两人的数据，并使用交叉规则得出2个交配结果。

Exchanger也可以用于校对工作，比如我们需要将纸制银行流水通过人工的方式录入成电子银行流水，为了避免错误，采用AB岗两人进行录入，录入到Excel之后，系统需要加载这两个Excel，并对两个Excel数据进行校对，看看是否录入一致。



如果两个线程有一个没有执行exchange()方法，则会一直等待，如果担心有特殊情况发生，避免一直等待，可以使用exchange（V x，longtimeout，TimeUnit unit）设置最大等待时长。

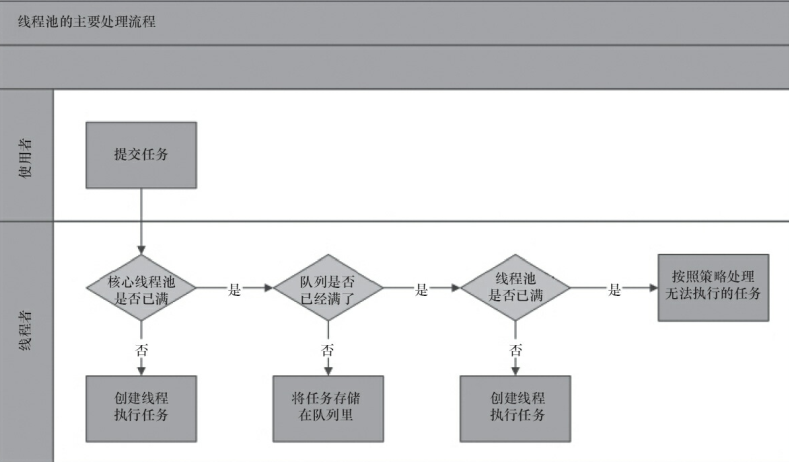
1. Java线程池的基本原理以及使用方法

Java中的线程池是运用场景最多的并发框架，几乎所有需要异步或并发执行任务的程序都可以使用线程池。在开发过程中，合理地使用线程池能够带来3个好处。

第一：降低资源消耗。通过重复利用已创建的线程降低线程创建和销毁造成的消耗。

第二：提高响应速度。当任务到达时，任务可以不需要等到线程创建就能立即执行。

第三：提高线程的可管理性。线程是稀缺资源，如果无限制地创建，不仅会消耗系统资源，还会降低系统的稳定性，使用线程池可以进行统一分配、调优和监控。但是，要做到合理利用线程池，必须对其实现原理了如指掌。



当提交一个新任务到线程池时，线程池的处理流程如下。

1）线程池判断核心线程池里的线程是否都在执行任务。如果不是，则创建一个新的工作线程来执行任务。如果核心线程池里的线程都在执行任务，则进入下个流程。

2）线程池判断工作队列是否已经满。如果工作队列没有满，则将新提交的任务存储在这个工作队列里。如果工作队列满了，则进入下个流程。

3）线程池判断线程池的线程是否都处于工作状态。如果没有，则创建一个新的工作线程来执行任务。如果已经满了，则交给饱和策略来处理这个任务。

-------------------------------------------------------------------------------

ThreadPoolExecutor执行execute方法分下面4种情况。

1）如果当前运行的线程少于corePoolSize，则创建新线程来执行任务（注意，执行这一步骤需要获取全局锁）。

2）如果运行的线程等于或多于corePoolSize，则将任务加入BlockingQueue。

3）如果无法将任务加入BlockingQueue（队列已满），则创建新的线程来处理任务（注意，执行这一步骤需要获取全局锁）。

4）如果创建新线程将使当前运行的线程超出maximumPoolSize，任务将被拒绝，并调用RejectedExecutionHandler.rejectedExecution()方法。

-------------------------------------------------------------------------------

线程池的创建：

new ThreadPoolExecutor(int corePoolSize, int maximumPoolSize, long keepAliveTime, TimeUnit unit, BlockingQueue<Runnable> workQueue, ThreadFactory threadFactory, RejectedExecutionHandler handler)

创建一个线程池时需要输入几个参数，如下。

1）corePoolSize（线程池的基本大小）：当提交一个任务到线程池时，线程池会创建一个线程来执行任务，即使其他空闲的基本线程能够执行新任务也会创建线程，等到需要执行的任务数大于线程池基本大小时就不再创建。如果调用了线程池的prestartAllCoreThreads()方法，线程池会提前创建并启动所有基本线程。

2）runnableTaskQueue（任务队列）：用于保存等待执行的任务的阻塞队列。可以选择以下几个阻塞队列。

·ArrayBlockingQueue：是一个基于数组结构的有界阻塞队列，此队列按FIFO（先进先出）原则对元素进行排序。

·LinkedBlockingQueue：一个基于链表结构的阻塞队列，此队列按FIFO排序元素，吞吐量通常要高于ArrayBlockingQueue。静态工厂方法Executors.newFixedThreadPool()使用了这个队列。

·SynchronousQueue：一个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，吞吐量通常要高于Linked-BlockingQueue，静态工厂方法Executors.newCachedThreadPool使用了这个队列。

·PriorityBlockingQueue：一个具有优先级的无限阻塞队列。

3）maximumPoolSize（线程池最大数量）：线程池允许创建的最大线程数。如果队列满了，并且已创建的线程数小于最大线程数，则线程池会再创建新的线程执行任务。值得注意的是，如果使用了无界的任务队列这个参数就没什么效果。

4）ThreadFactory：用于设置创建线程的工厂，可以通过线程工厂给每个创建出来的线程设置更有意义的名字。使用开源框架guava提供的ThreadFactoryBuilder可以快速给线程池里的线程设置有意义的名字，代码如下。new ThreadFactoryBuilder().setNameFormat("XX-task-%d").build();

5）RejectedExecutionHandler（饱和策略）：当队列和线程池都满了，说明线程池处于饱和状态，那么必须采取一种策略处理提交的新任务。这个策略默认情况下是AbortPolicy，表示无法处理新任务时抛出异常。在JDK 1.5中Java线程池框架提供了以下4种策略。

·AbortPolicy：直接抛出异常。

·CallerRunsPolicy：只用调用者所在线程来运行任务。

·DiscardOldestPolicy：丢弃队列里最近的一个任务，并执行当前任务。

·DiscardPolicy：不处理，丢弃掉。当然，也可以根据应用场景需要来实现RejectedExecutionHandler接口自定义策略。如记录日志或持久化存储不能处理的任务。

6）keepAliveTime（线程活动保持时间）：线程池的工作线程空闲后，保持存活的时间。所以，如果任务很多，并且每个任务执行的时间比较短，可以调大时间，提高线程的利用率。

7) TimeUnit（线程活动保持时间的单位）：可选的单位有天（DAYS）、小时（HOURS）、分钟（MINUTES）、毫秒（MILLISECONDS）、微秒（MICROSECONDS，千分之一毫秒）和纳秒（NANOSECONDS，千分之一微秒）。



-------------------------------------------------------------------------------

可以使用两个方法向线程池提交任务，分别为execute()和submit()方法。execute()方法用于提交不需要返回值的任务，所以无法判断任务是否被线程池执行成功。submit()方法用于提交需要返回值的任务。线程池会返回一个future类型的对象，通过这个future对象可以判断任务是否执行成功，并且可以通过future的get()方法来获取返回值，get()方法会阻塞当前线程直到任务完成，而使用get（long timeout，TimeUnit unit）方法则会阻塞当前线程一段时间后立即返回，这时候有可能任务没有执行完。

可以通过调用线程池的shutdown或shutdownNow方法来关闭线程池。它们的原理是遍历线程池中的工作线程，然后逐个调用线程的interrupt方法来中断线程，所以无法响应中断的任务可能永远无法终止。但是它们存在一定的区别，shutdownNow首先将线程池的状态设置成STOP，然后尝试停止所有的正在执行或暂停任务的线程，并返回等待执行任务的列表，而shutdown只是将线程池的状态设置成SHUTDOWN状态，然后中断所有没有正在执行任务的线程。只要调用了这两个关闭方法中的任意一个，isShutdown方法就会返回true。当所有的任务都已关闭后，才表示线程池关闭成功，这时调用isTerminaed方法会返回true。至于应该调用哪一种方法来关闭线程池，应该由提交到线程池的任务特性决定，通常调用shutdown方法来关闭线程池，如果任务不一定要执行完，则可以调用shutdownNow方法。

-------------------------------------------------------------------------------

要想合理地配置线程池，就必须首先分析任务特性，可以从以下几个角度来分析。

·任务的性质：CPU密集型任务、IO密集型任务和混合型任务。

·任务的优先级：高、中和低。

·任务的执行时间：长、中和短。

·任务的依赖性：是否依赖其他系统资源，如数据库连接。

性质不同的任务可以用不同规模的线程池分开处理。CPU密集型任务应配置尽可能小的线程，如配置Ncpu+1个线程的线程池。由于IO密集型任务线程并不是一直在执行任务，则应配置尽可能多的线程，如2\*Ncpu。混合型的任务，如果可以拆分，将其拆分成一个CPU密集型任务和一个IO密集型任务，只要这两个任务执行的时间相差不是太大，那么分解后执行的吞吐量将高于串行执行的吞吐量。如果这两个任务执行时间相差太大，则没必要进行分解。可以通过Runtime.getRuntime().availableProcessors()方法获得当前设备的CPU个数。优先级不同的任务可以使用优先级队列PriorityBlockingQueue来处理。它可以让优先级高的任务先执行。

注意：如果一直有优先级高的任务提交到队列里，那么优先级低的任务可能永远不能执行。

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。依赖数据库连接池的任务，因为线程提交SQL后需要等待数据库返回结果，等待的时间越长，则CPU空闲时间就越长，那么线程数应该设置得越大，这样才能更好地利用CPU。

建议使用有界队列。有界队列能增加系统的稳定性和预警能力，可以根据需要设大一点儿，比如几千。有一次，我们系统里后台任务线程池的队列和线程池全满了，不断抛出抛弃任务的异常，通过排查发现是数据库出现了问题，导致执行SQL变得非常缓慢，因为后台任务线程池里的任务全是需要向数据库查询和插入数据的，所以导致线程池里的工作线程全部阻塞，任务积压在线程池里。如果当时我们设置成无界队列，那么线程池的队列就会越来越多，有可能会撑满内存，导致整个系统不可用，而不只是后台任务出现问题。当然，我们的系统所有的任务是用单独的服务器部署的，我们使用不同规模的线程池完成不同类型的任务，但是出现这样问题时也会影响到其他任务。

-------------------------------------------------------------------------------

线程池的监控

如果在系统中大量使用线程池，则有必要对线程池进行监控，方便在出现问题时，可以根据线程池的使用状况快速定位问题。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以下属性。

·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，小于或等于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。如该数值等于线程池的最大大小，则表示线程池曾经满过。

·getPoolSize：线程池的线程数量。如果线程池不销毁的话，线程池里的线程不会自动销毁，所以这个大小只增不减。

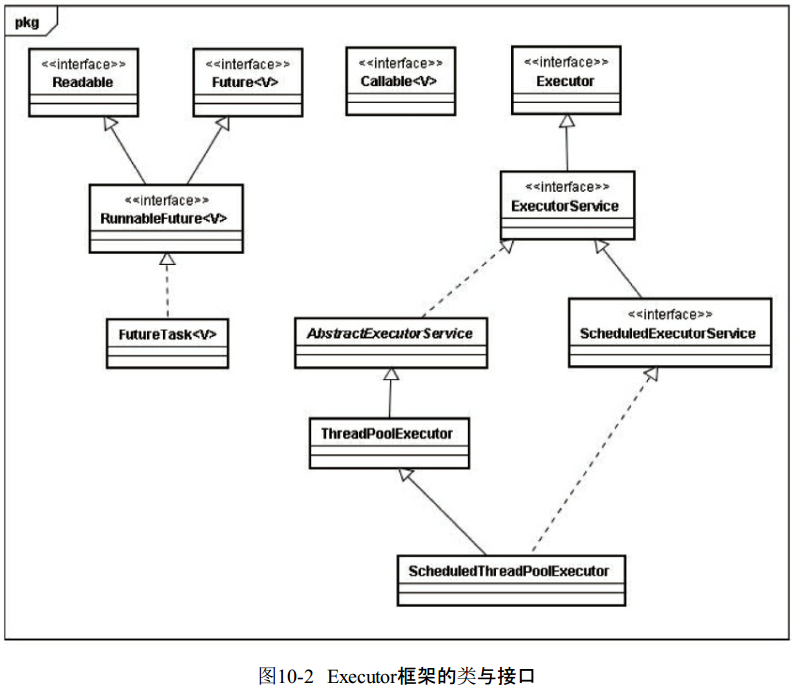
·getActiveCount：获取活动的线程数。

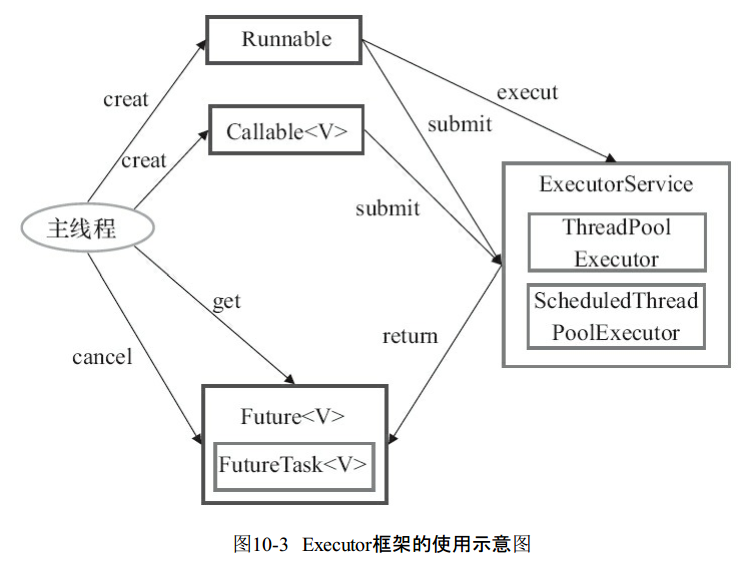
通过扩展线程池进行监控。可以通过继承线程池来自定义线程池，重写线程池的beforeExecute、afterExecute和terminated方法，也可以在任务执行前、执行后和线程池关闭前执行一些代码来进行监控。例如，监控任务的平均执行时间、最大执行时间和最小执行时间等。这几个方法在线程池里是空方法。

-------------------------------------------------------------------------------

1. Executor框架介绍

Executor框架的主要成员：ThreadPoolExecutor、ScheduledThreadPoolExecutor、Future接口、Runnable接口、Callable接口和Executors。

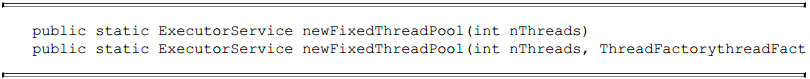




（1）ThreadPoolExecutor

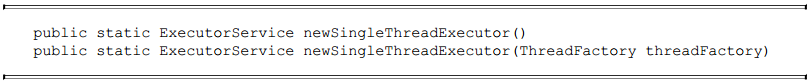
ThreadPoolExecutor通常使用工厂类Executors来创建。Executors可以创建3种类型的ThreadPoolExecutor：SingleThreadExecutor、FixedThreadPool和CachedThreadPool。下面分别介绍这3种ThreadPoolExecutor。

FixedThreadPool。下面是Executors提供的，创建使用固定线程数的FixedThreadPool的API。



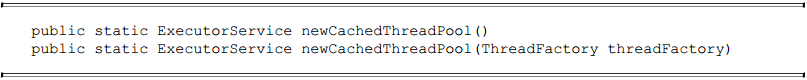
FixedThreadPool适用于为了满足资源管理的需求，而需要限制当前线程数量的应用场景，它适用于负载比较重的服务器。

2）SingleThreadExecutor。下面是Executors提供的，创建使用单个线程的SingleThreadExecutor的API。



SingleThreadExecutor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。

CachedThreadPool。下面是Executors提供的，创建一个会根据需要创建新线程的CachedThreadPool的API。



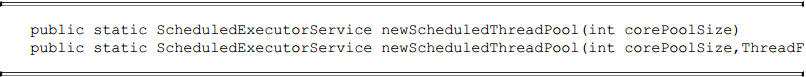
CachedThreadPool是大小无界的线程池，适用于执行很多的短期异步任务的小程序，或者是负载较轻的服务器。

（2）ScheduledThreadPoolExecutor

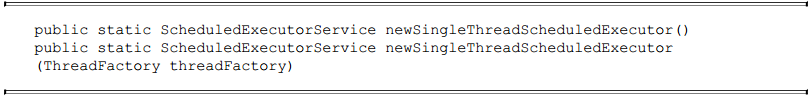
ScheduledThreadPoolExecutor通常使用工厂类Executors来创建。Executors可以创建2种类型的ScheduledThreadPoolExecutor，如下。

·ScheduledThreadPoolExecutor。包含若干个线程的ScheduledThreadPoolExecutor。

·SingleThreadScheduledExecutor。只包含一个线程的ScheduledThreadPoolExecutor。下面分别介绍这两种ScheduledThreadPoolExecutor下面是工厂类Executors提供的，创建固定个数线程的ScheduledThreadPoolExecutor的API。



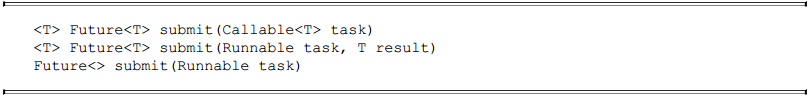
ScheduledThreadPoolExecutor适用于需要多个后台线程执行周期任务，同时为了满足资源管理的需求而需要限制后台线程的数量的应用场景。下面是Executors提供的，创建单个线程的SingleThreadScheduledExecutor的API。



SingleThreadScheduledExecutor适用于需要单个后台线程执行周期任务，同时需要保证顺序地执行各个任务的应用场景。

（3）Future接口

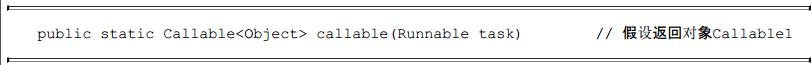
Future接口和实现Future接口的FutureTask类用来表示异步计算的结果。当我们把Runnable接口或Callable接口的实现类提交（submit）给ThreadPoolExecutor或ScheduledThreadPoolExecutor时，ThreadPoolExecutor或ScheduledThreadPoolExecutor会向我们返回一个FutureTask对象。下面是对应的API。



有一点需要读者注意，到目前最新的JDK 8为止，Java通过上述API返回的是一个FutureTask对象。但从API可以看到，Java仅仅保证返回的是一个实现了Future接口的对象。在将来的JDK实现中，返回的可能不一定是FutureTask。

（4）Runnable接口和Callable接口

Runnable接口和Callable接口的实现类，都可以被ThreadPoolExecutor或ScheduledThreadPoolExecutor执行。它们之间的区别是Runnable不会返回结果，而Callable可以返回结果。除了可以自己创建实现Callable接口的对象外，还可以使用工厂类Executors来把一个Runnable包装成一个Callable。下面是Executors提供的，把一个Runnable包装成一个Callable的API。



下面是Executors提供的，把一个Runnable和一个待返回的结果包装成一个Callable的API。



前面讲过，当我们把一个Callable对象（比如上面的Callable1或Callable2）提交给ThreadPoolExecutor或ScheduledThreadPoolExecutor执行时，submit（…）会向我们返回一个FutureTask对象。我们可以执行FutureTask.get()方法来等待任务执行完成。当任务成功完成后FutureTask.get()将返回该任务的结果。例如，如果提交的是对象Callable1，FutureTask.get()方法将返回null；如果提交的是对象Callable2，FutureTask.get()方法将返回result对象。

-------------------------------------------------------------------------------

1. ThreadPoolExecutor详解

Executor框架最核心的类是ThreadPoolExecutor，它是线程池的实现类，主要由下列4个组件构成。

·corePool：核心线程池的大小。

·maximumPool：最大线程池的大小。

·BlockingQueue：用来暂时保存任务的工作队列。

·RejectedExecutionHandler：当ThreadPoolExecutor已经关闭或ThreadPoolExecutor已经饱和时（达到了最大线程池大小且工作队列已满），execute()方法将要调用的Handler。

·通过Executor框架的工具类Executors，可以创建3种类型的ThreadPoolExecutor。

·FixedThreadPool。

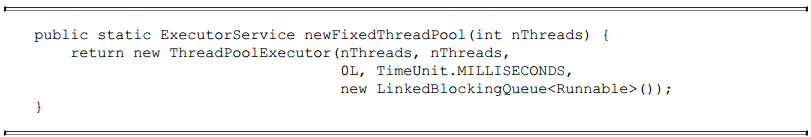
·SingleThreadExecutor。

·CachedThreadPool。

下面将分别介绍这3种ThreadPoolExecutor。

-------------------------------------------------------------------------------

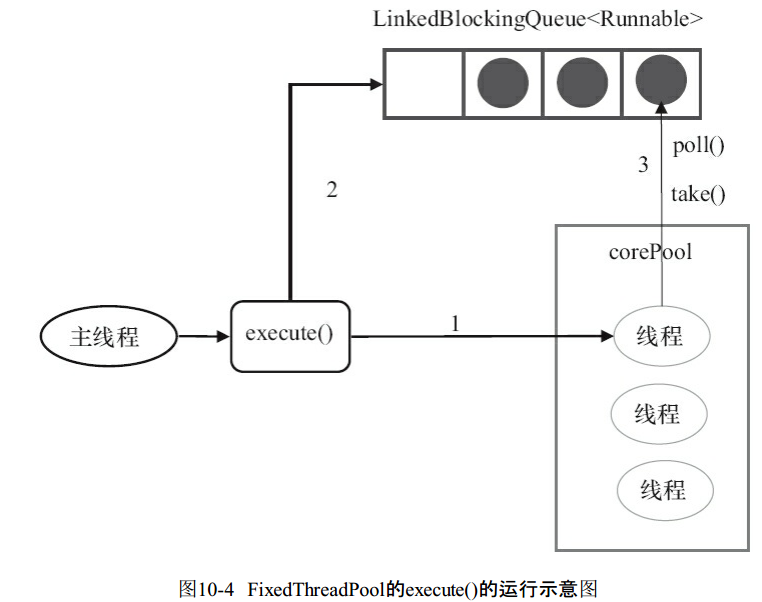
FixedThreadPool被称为可重用固定线程数的线程池。下面是FixedThreadPool的源代码实现。



FixedThreadPool的corePoolSize和maximumPoolSize都被设置为创建FixedThreadPool时指定的参数nThreads。

当线程池中的线程数大于corePoolSize时，keepAliveTime为多余的空闲线程等待新任务的最长时间，超过这个时间后多余的线程将被终止。这里把keepAliveTime设置为0L，意味着多余的空闲线程会被立即终止。

FixedThreadPool的execute()方法的运行示意图如图



1）如果当前运行的线程数少于corePoolSize，则创建新线程来执行任务。

2）在线程池完成预热之后（当前运行的线程数等于corePoolSize），将任务加入LinkedBlockingQueue。

3）线程执行完1中的任务后，会在循环中反复从LinkedBlockingQueue获取任务来执行。

FixedThreadPool使用无界队列LinkedBlockingQueue作为线程池的工作队列（队列的容量为Integer.MAX\_VALUE）。使用无界队列作为工作队列会对线程池带来如下影响。

1）当线程池中的线程数达到corePoolSize后，新任务将在无界队列中等待，因此线程池中的线程数不会超过corePoolSize。

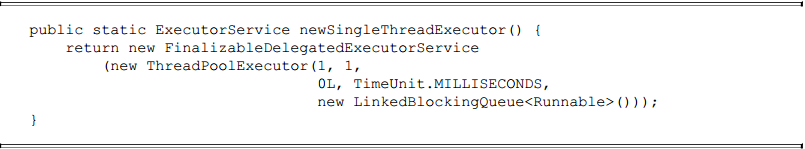
2）由于1，使用无界队列时maximumPoolSize将是一个无效参数。

3）由于1和2，使用无界队列时keepAliveTime将是一个无效参数。

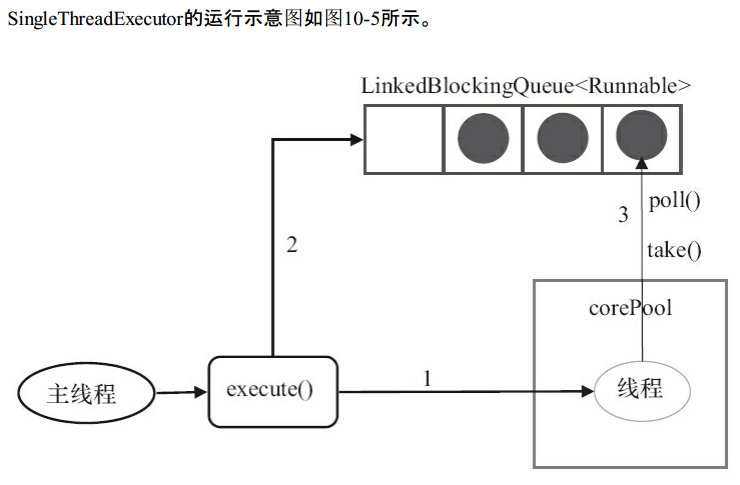
4）由于使用无界队列，运行中的FixedThreadPool（未执行方法shutdown()或shutdownNow()）不会拒绝任务（不会调用RejectedExecutionHandler.rejectedExecution方法）。

-------------------------------------------------------------------------------

SingleThreadExecutor是使用单个worker线程的Executor。下面是SingleThreadExecutor的源代码实现。



SingleThreadExecutor的corePoolSize和maximumPoolSize被设置为1。其他参数与FixedThreadPool相同。SingleThreadExecutor使用无界队列LinkedBlockingQueue作为线程池的工作队列（队列的容量为Integer.MAX\_VALUE）。SingleThreadExecutor使用无界队列作为工作队列对线程池带来的影响与FixedThreadPool相同，这里就不赘述了。

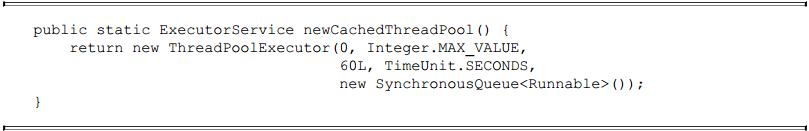


1）如果当前运行的线程数少于corePoolSize（即线程池中无运行的线程），则创建一个新线程来执行任务。

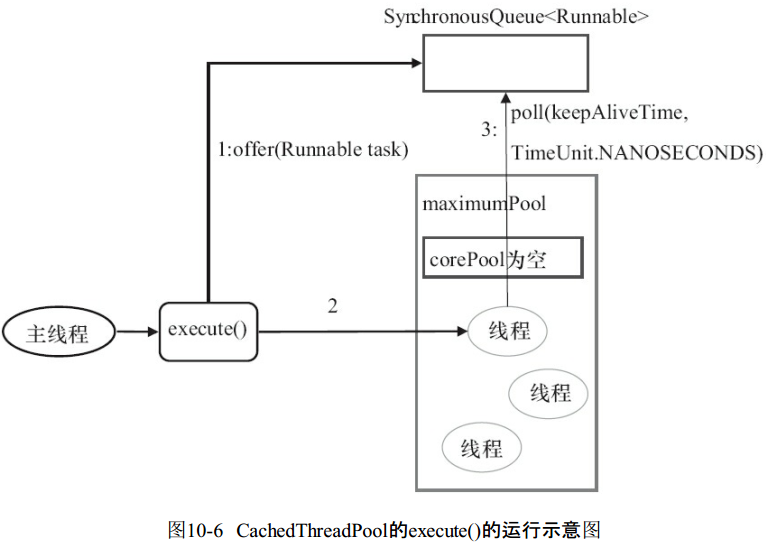
2）在线程池完成预热之后（当前线程池中有一个运行的线程），将任务加入LinkedBlockingQueue。

3）线程执行完1中的任务后，会在一个无限循环中反复从LinkedBlockingQueue获取任务来执行。  
-------------------------------------------------------------------------------

CachedThreadPool是一个会根据需要创建新线程的线程池。下面是创建CachedThreadPool的源代码。



CachedThreadPool的corePoolSize被设置为0，即corePool为空；maximumPoolSize被设置为Integer.MAX\_VALUE，即maximumPool是无界的。这里把keepAliveTime设置为60L，意味着CachedThreadPool中的空闲线程等待新任务的最长时间为60秒，空闲线程超过60秒后将会被终止。FixedThreadPool和SingleThreadExecutor使用无界队列LinkedBlockingQueue作为线程池的工作队列。CachedThreadPool使用没有容量的SynchronousQueue作为线程池的工作队列，但CachedThreadPool的maximumPool是无界的。这意味着，如果主线程提交任务的速度高于maximumPool中线程处理任务的速度时，CachedThreadPool会不断创建新线程。极端情况下，CachedThreadPool会因为创建过多线程而耗尽CPU和内存资源。



1）首先执行SynchronousQueue.offer（Runnable task）。如果当前maximumPool中有空闲线程正在执行SynchronousQueue.poll（keepAliveTime，TimeUnit.NANOSECONDS），那么主线程执行offer操作与空闲线程执行的poll操作配对成功，主线程把任务交给空闲线程执行，execute()方法执行完成；否则执行下面的步骤2）。

2）当初始maximumPool为空，或者maximumPool中当前没有空闲线程时，将没有线程执行SynchronousQueue.poll（keepAliveTime，TimeUnit.NANOSECONDS）。这种情况下，步骤1）将失败。此时CachedThreadPool会创建一个新线程执行任务，execute()方法执行完成。

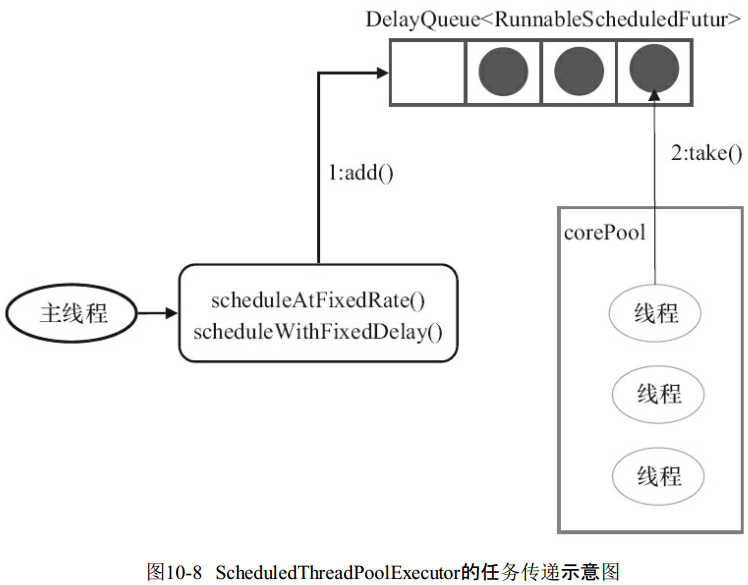
3）在步骤2）中新创建的线程将任务执行完后，会执行SynchronousQueue.poll（keepAliveTime，TimeUnit.NANOSECONDS）。这个poll操作会让空闲线程最多在SynchronousQueue中等待60秒钟。如果60秒钟内主线程提交了一个新任务（主线程执行步骤1）），那么这个空闲线程将执行主线程提交的新任务；否则，这个空闲线程将终止。由于空闲60秒的空闲线程会被终止，因此长时间保持空闲的CachedThreadPool不会使用任何资源。前面提到过，SynchronousQueue是一个没有容量的阻塞队列。每个插入操作必须等待另一个线程的对应移除操作，反之亦然。CachedThreadPool使用SynchronousQueue，把主线程提交的任务传递给空闲线程执行。

-------------------------------------------------------------------------------

1. ScheduledThreadPoolExecutor详解

ScheduledThreadPoolExecutor继承自ThreadPoolExecutor。它主要用来在给定的延迟之后运行任务，或者定期执行任务。ScheduledThreadPoolExecutor的功能与Timer类似，但ScheduledThreadPoolExecutor功能更强大、更灵活。Timer对应的是单个后台线程，而ScheduledThreadPoolExecutor可以在构造函数中指定多个对应的后台线程数。

ScheduledThreadPoolExecutor的执行示意图如下（本文基于JDK 6）



DelayQueue是一个无界队列，所以ThreadPoolExecutor的maximumPoolSize在ScheduledThreadPoolExecutor中没有什么意义（设置maximumPoolSize的大小没有什么效果）。ScheduledThreadPoolExecutor的执行主要分为两大部分。

1)当调用ScheduledThreadPoolExecutor的scheduleAtFixedRate()方法或者scheduleWithFixedDelay()方法时，会向ScheduledThreadPoolExecutor的DelayQueue添加一个实现了RunnableScheduledFutur接口的ScheduledFutureTask。

2）线程池中的线程从DelayQueue中获取ScheduledFutureTask，然后执行任务。

ScheduledThreadPoolExecutor的实现

前面我们提到过，ScheduledThreadPoolExecutor会把待调度的任务（ScheduledFutureTask）放到一个DelayQueue中。ScheduledFutureTask主要包含3个成员变量，如下。

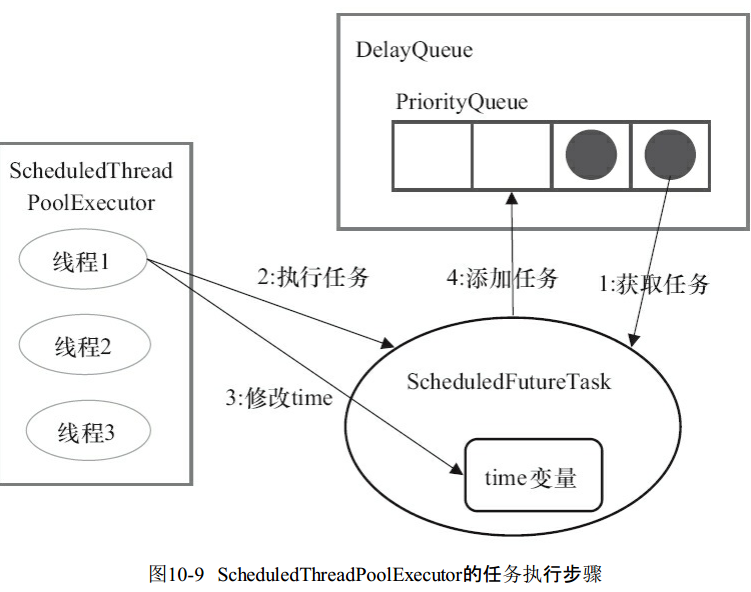
·long型成员变量time，表示这个任务将要被执行的具体时间。

·long型成员变量sequenceNumber，表示这个任务被添加到ScheduledThreadPoolExecutor中的序号。

·long型成员变量period，表示任务执行的间隔周期。

DelayQueue封装了一个PriorityQueue，这个PriorityQueue会对队列中的ScheduledFutureTask进行排序。排序时，time小的排在前面（时间早的任务将被先执行）。

如果两个ScheduledFutureTask的time相同，就比较sequenceNumber，sequenceNumber小的排在前面（也就是说，如果两个任务的执行时间相同，那么先提交的任务将被先执行）。首先，让我们看看ScheduledThreadPoolExecutor中的线程执行周期任务的过程。



下面是对这4个步骤的说明。

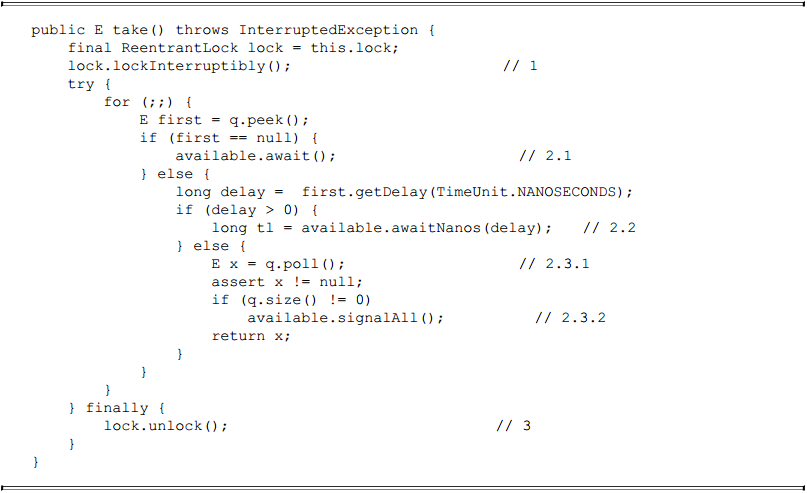
1）线程1从DelayQueue中获取已到期的ScheduledFutureTask（DelayQueue.take()）。到期任务是指ScheduledFutureTask的time大于等于当前时间。

2）线程1执行这个ScheduledFutureTask。

3）线程1修改ScheduledFutureTask的time变量为下次将要被执行的时间。

4）线程1把这个修改time之后的ScheduledFutureTask放回DelayQueue中（DelayQueue.add()）。

接下来，让我们看看上面的步骤1）获取任务的过程。下面是DelayQueue.take()方法的源代码实现。



获取任务分为3大步骤。

1）获取Lock。

2）获取周期任务。

·如果PriorityQueue为空，当前线程到Condition中等待；否则执行下面的2.2。

·如果PriorityQueue的头元素的time时间比当前时间大，到Condition中等待到time时间；否则执行下面的2.3。

·获取PriorityQueue的头元素（2.3.1）；如果PriorityQueue不为空，则唤醒在Condition中等待的所有线程（2.3.2）。

3）释放Lock。

ScheduledThreadPoolExecutor在一个循环中执行步骤2，直到线程从PriorityQueue获取到一个元素之后（执行2.3.1之后），才会退出无限循环（结束步骤2）。

-------------------------------------------------------------------------------

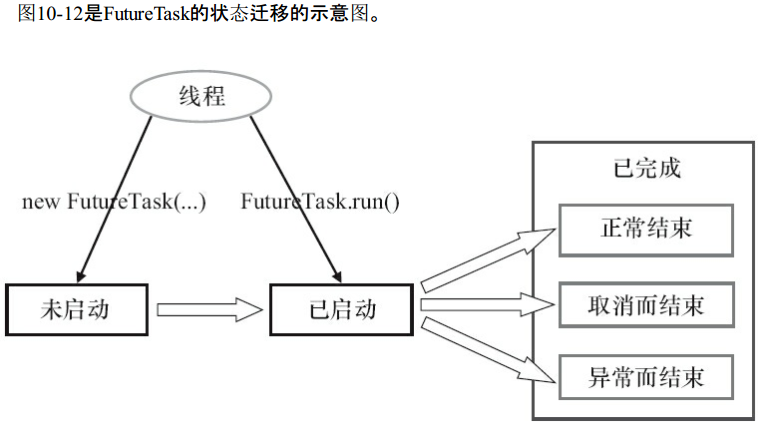
1. FutureTask简介

FutureTask除了实现Future接口外，还实现了Runnable接口。因此，FutureTask可以交给Executor执行，也可以由调用线程直接执行（FutureTask.run()）。根据FutureTask.run()方法被执行的时机，FutureTask可以处于下面3种状态。

1）未启动。FutureTask.run()方法还没有被执行之前，FutureTask处于未启动状态。当创建一个FutureTask，且没有执行FutureTask.run()方法之前，这个FutureTask处于未启动状态。

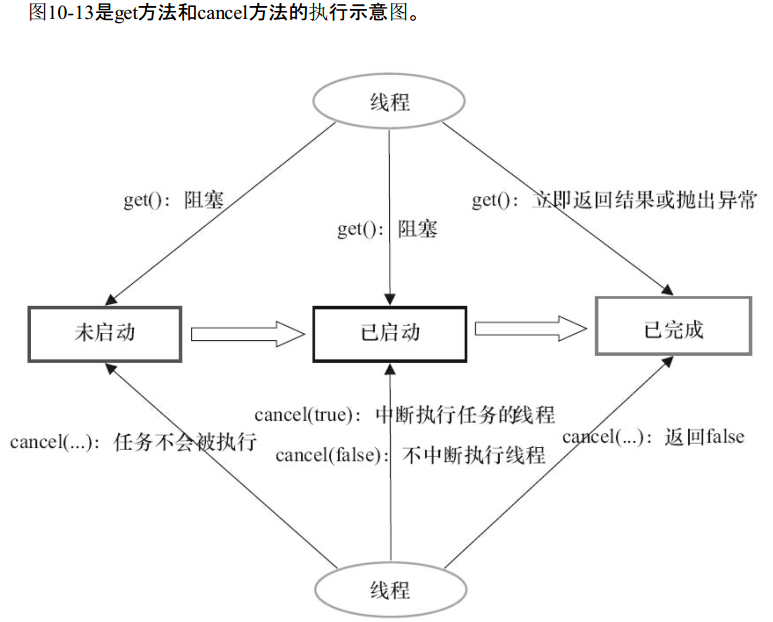
2）已启动。FutureTask.run()方法被执行的过程中，FutureTask处于已启动状态。

3）已完成。FutureTask.run()方法执行完后正常结束，或被取消（FutureTask.cancel（…）），或执行FutureTask.run()方法时抛出异常而异常结束，FutureTask处于已完成状态。



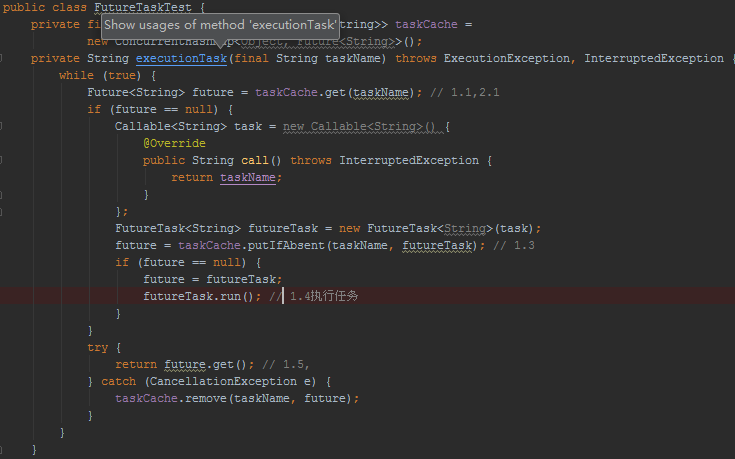
当FutureTask处于未启动或已启动状态时，执行FutureTask.get()方法将导致调用线程阻塞；当

FutureTask处于已完成状态时，执行FutureTask.get()方法将导致调用线程立即返回结果或抛出异常。当FutureTask处于未启动状态时，执行FutureTask.cancel()方法将导致此任务永远不会被执行；当FutureTask处于已启动状态时，执行FutureTask.cancel（true）方法将以中断执行此任务线程的方式来试图停止任务；当FutureTask处于已启动状态时，执行FutureTask.cancel（false）方法将不会对正在执行此任务的线程产生影响（让正在执行的任务运行完成）；当FutureTask处于已完成状态时，执行FutureTask.cancel（…）方法将返回false。



可以把FutureTask交给Executor执行；也可以通过ExecutorService.submit（…）方法返回一个FutureTask，然后执行FutureTask.get()方法或FutureTask.cancel（…）方法。除此以外，还可以单独使用FutureTask。

当一个线程需要等待另一个线程把某个任务执行完后它才能继续执行，此时可以使用FutureTask。假设有多个线程执行若干任务，每个任务最多只能被执行一次。当多个线程试图同时执行同一个任务时，只允许一个线程执行任务，其他线程需要等待这个任务执行完后才能继续执行。下面是对应的示例代码。



当两个线程试图同时执行同一个任务时，如果Thread 1执行1.3后Thread 2执行2.1，那么接下来Thread 2将在2.2等待，直到Thread 1执行完1.4后Thread 2才能从2.2（FutureTask.get()）返回。

FutureTask的实现基于AbstractQueuedSynchronizer（以下简称为AQS）。java.util.concurrent中的很多可阻塞类（比如ReentrantLock）都是基于AQS来实现的。AQS是一个同步框架，它提供通用机制来原子性管理同步状态、阻塞和唤醒线程，以及维护被阻塞线程的队列。JDK 6中AQS被广泛使用，基于AQS实现的同步器包括：ReentrantLock、Semaphore、ReentrantReadWriteLock、CountDownLatch和FutureTask。

-------------------------------------------------------------------------------

1. 解决Hash冲突的两种方法：

在理想状态下，哈希函数可以将关键字均匀的分散到数组的不同位置，不会出现两个关键字散列值相同（假设关键字数量小于数组的大小）的情况。但是在实际使用中，经常会出现多个关键字散列值相同的情况（被映射到数组的同一个位置），我们将这种情况称为散列冲突。为了解决散列冲突，主要采用下面两种方式：

分离链表法（separate chaining）

开放定址法（open addressing）

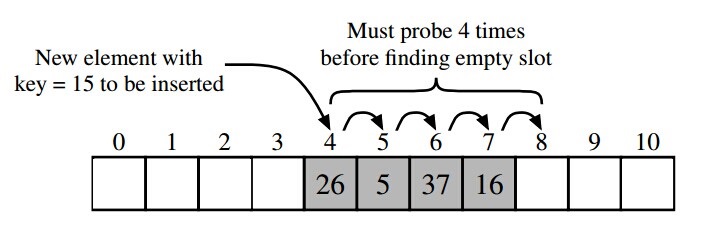
分离链表法

分散链表法使用链表解决冲突，将散列值相同的元素都保存到一个链表中。当查询的时候，首先找到元素所在的链表，然后遍历链表查找对应的元素。下面是一个示意图：



开放定址法

开放定址法不会创建链表，当关键字散列到的数组单元已经被另外一个关键字占用的时候，就会尝试在数组中寻找其他的单元，直到找到一个空的单元。探测数组空单元的方式有很多，这里介绍一种最简单的 -- 线性探测法。线性探测法就是从冲突的数组单元开始，依次往后搜索空单元，如果到数组尾部，再从头开始搜索（环形查找）。如下图所示：



ThreadLocalMap 中使用开放地址法来处理散列冲突，而 HashMap 中使用的分离链表法。之所以采用不同的方式主要是因为：在 ThreadLocalMap 中的散列值分散的十分均匀，很少会出现冲突。并且 ThreadLocalMap 经常需要清除无用的对象，使用纯数组更加方便。

-------------------------------------------------------------------------------

1. CompletionService和ExecutorCompletionService异步任务线程池

使用ExecutorCompletionService管理异步任务

1. Java中的ExecutorCompletionService<V>本身有管理任务队列的功能

\* i. ExecutorCompletionService内部维护列一个队列, 用于管理已完成的任务

\* ii. 内部还维护列一个Executor, 可以执行任务

2. ExecutorCompletionService内部维护了一个BlockingQueue, 只有完成的任务才被加入到队列中

3. 任务一完成就加入到内置管理队列中, 如果队列中的数据为空时, 调用take()就会阻塞 (等待任务完成)

\* i. 关于完成任务是如何加入到完成队列中的, 请参考ExecutorCompletionService的内部类QueueingFuture的done()方法

4. ExecutorCompletionService的take/poll方法是对BlockingQueue对应的方法的封装, 关于BlockingQueue的take/poll方法:

\* i. take()方法, 如果队列中有数据, 就返回数据, 否则就一直阻塞;

\* ii. poll()方法: 如果有值就返回, 否则返回null

\* iii. poll(long timeout, TimeUnit unit)方法: 如果有值就返回, 否则等待指定的时间; 如果时间到了如果有值, 就返回值, 否则返回null

解决了已完成任务得不到及时处理的问。

CompletionService与ExecutorService最主要的区别在于

         \*前者submit的task不一定是按照加入时的顺序完成的。CompletionService对ExecutorService进行了包装，

         \*内部维护一个保存Future对象的BlockingQueue。

         \*只有当这个Future对象状态是结束的时候，才会加入到这个Queue中，take()方法其实就是Producer-Consumer中的Consumer。

         \*它会从Queue中取出Future对象，如果Queue是空的，就会阻塞在那里，直到有完成的Future对象加入到Queue中。

         \*所以，先完成的必定先被取出。这样就减少了不必要的等待时间。

-------------------------------------------------------------------------------

1. ThreadGroup使用方法

线程组ThreadGroup表示一组线程的集合,一旦一个线程归属到一个线程组之中后，就不能再更换其所在的线程组。那么为什么要使用线程组呢？个人认为有以下的好处：方便统一管理，线程组可以进行复制，快速定位到一个线程，统一进行异常设置等。ThreadGroup它其实并不属于Java并发包中的内容，它是java.lang中的内容。

基本用法:

1 获取当前线程组名

Thread.currentThread().getThreadGroup().getName() ;

2 将线程放入到一个线程组中去

ThreadGroup threadGroup1 = new ThreadGroup("group1");

ThreadGroup threadGroup2 = new ThreadGroup("group2");

Thread thread1 =new Thread(threadGroup1, "group1's member");

Thread thread2 =new Thread(threadGroup2, "group2's member");

3 复制线程组

Thread[] threads = new Thread[threadGroup.activeCount()];

threadGroup.enumerate(threads);

4 未捕获异常处理

ThreadGroup中有一个uncaughtException()方法。当线程组中某个线程发生Unchecked exception异常时，由执行环境调用此方法进行相关处理，如果有必要，可以重新定义此方法。

**(1)线程组也可以包含其他线程组。**

**(2)线程组构成一棵树，在树中，除了初始线程组外，每个线程组都有一个父线程组。**

-------------------------------------------------------------------------------

1. 帖子摘抄

有三种方式可以降低锁的竞争程度：减少锁的持有时间、降低锁的请求频率、使用带有协调机制的独占锁，这些机制允许更高的并发性。

减少锁的竞争：缩小锁的范围（快进快出）、减小锁的粒度（锁分解）、锁分段、避免热点域、一些替代古老锁的方法（使用并发容器、读写锁、不可变对象、院子变量）。

由于使用线程常常是为了充分利用多个处理器的计算能力，因此在并发程序性能的讨论中，通常更多地将侧重点放在吞吐量和可伸缩性上，而不是服务时间。Amdahl定律告诉我们，程序的可伸缩性取决于在所有代码中必须被串行执行的代码比例。因为Java程序中串行操作的主要来源是独占方式的资源，因此通常可以通过以下方式来提升可伸缩性：减少锁的持有时间，降低锁的粒度，以及采用非独占的锁或非阻塞锁来代替独占锁。

在一些内置所无法满足需求的情况下，ReentrantLock可以作为一种高级工具。当需要一些高级功能时才应该使用ReentrantLock，这些功能包括：可定时的、可轮询的与可中断的锁获取操作，公平队列，以及非块结构的锁。否则，还是应该优先使用synchronized。

在Condition对象中，与wait、notify、notifyAll方法对应的分别是await、signal、signalAll。但是，Condition对Object进行了子类扩展，因而它也包含wait、notify方法。一定要确保使用正确的版本-await、signal。signal比signalAll更高效，它能极大的减少在每次缓存操作中发生的上下文切换和锁请求的次数。

一个很管用的经验法则是：在大多数处理器上，在无竞争的锁获取和释放的“快速代码路径”上的开销，大约是CAS开销的两倍。

非阻塞算法：如果在某种算法中，一个线程的失败或挂起不会导致其他线程失败或挂起，那么这种算法就被称为非阻塞算法。如果在算法的每个步骤都存在某个线程能够执行下去，那么这种算法也称为无锁算法。如果在算法中仅将CAS用作线程之间的协调，并且能正确实现，那么它既是一种无阻塞算法，也是一种无锁算法。非阻塞算法的所有特性：某项工作的完成具有不确定性，必须重新执行。CAS使用的基本模式：在更新某个值存在不确定性，以及在更新失败时重新尝试。构建非阻塞算法的技巧在于：将执行原子修改的范围缩小到单个变量上。

解决ABA问题相对简单的解决方案：不是更新某个引用的值，而是更新两个值，包括一个引用和一个版本号。即使这个值由A变为B，又由B变成A，版本号也将使不同的。AtomicStampedReference以及AtomicMarkableReference支持在两个变量上执行原子条件的更新。AtomicStampedReference将更新一个“对象-引用”二元组，通过在引用上加上版本从而避免ABA问题。类似的，AtomicMarkableReference将更新一个“对象引用-布尔值”二元组，在某些算法中将通过这种二元组使节点保存在链表中同时又将其标记为“已删除的节点“。

volatile、synchronized、final都能保证变量可见性