# 第2章：Java并行程序基础

## 2.2 初始线程：线程的基本操作

### 2.2.1 新建线程

注意：不要用run()来开启新线程。它只会在当前线程中，串行执行run()中的代码。

\* 一个线程如果调用两次start()方法，会有一次执行成功，另一次抛出异常java.lang.IllegalThreadStateException

为什么stop()被废弃而不推荐使用呢？原因是stop()方法太过于暴力，强行把执行到一半的线程终止，可能会引起一些数据不一致的问题。

### 2.2.2 终止线程

Thread.stop()方法在结束线程时，会产生数据不一致问题。

### 2.2.3 线程中断

在Java中，线程中断是一种重要的线程协作机制。从表面上理解，中断就是让目标线程停止执行的意思，实际上并非完全如此。

严格地讲，线程中断并不会是线程立即退出，而是给线程发送一个通知，告知目标线程，有人希望你退出。至于目标线程接到通知后如何处理，则完全由目标线程自行决定。

### 2.2.4 等待(wait)和通知(notify)

如果一个线程调用了object.wait()，那么它就会进入object对象的等待队列。这个等待队列中，可能会有多个线程，因为系统运行多个线程同时等待某一个对象。当object.notify()被调用时，它就会从这个等待队列中，随机选择一个线程，并将其唤醒。这个选择并不是公平的，并不是先等待的线程会优先被选择，这个选择完全是随机的。

除了notify()方法外，Object对象还有一个类似的notifyAll()方法，它和notify()的功能基本一致，但不同的是，它会唤醒在这个等待队列中所有等待的线程，而不是随机选择一个。

注意：Object.wait()和Thread.sleep()方法都可以让线程等待若干时间。除了wait()可以被唤醒外，另外一个主要区别就是wait()方法会释放目标对象的锁，而Thread.sleep()方法不会释放任何资源。

### 2.2.5 挂起(suspend)和继续执行(resume)线程

JDK有关的Thread类的API文档中，还有两个被标注为废弃的方法，即线程挂起(suspend)和继续执行(resume)，并不推荐使用。

不推荐使用suspend()去挂起线程的原因，是因为suspend()在导致线程暂停的同时，并不会去释放任何锁资源。此时，其他任何线程想要访问它占用的锁时，都会被牵连，导致无法正常继续运行。直到对应的线程上进行了resume()操作，被挂起的线程才能继续，从而其他所有阻塞在相关锁上的线程也可以继续执行。但是，如果resume()操作意外地在suspend()前就执行了，那么被挂起的线程可能很难有机会被继续执行。并且，更严重的是：它所占用的锁不会被释放，因此可能会导致整个系统工作不正常。而且，对于被挂起的线程，从它的线程状态上看，居然还是Runnable，这也会严重影响我们对系统当前状态的判断。

### 2.2.6 等待线程结束(join)和谦让(yield)

很多时候，一个线程的输入可能非常依赖于另外一个或者多个线程的输出，此时，这个线程就需要等待依赖线程执行完毕，才能继续执行。JDK提供了join()操作来实现这个功能。

join()的API有两个：

public final void join() throws InterruptedException

public final synchronized void join(long millis) throws InterruptedException

第一个join()方法表示无限等待，它会一直阻塞当前线程，直到目标线程执行完毕。第二个方法给出了一个最大等待时间，如果超过给定时间目标线程还在执行，当前线程也会因为“等不及了”，而继续往下执行。

join()的本质是让调用线程wait()在当前线程对象实例上。下面是JDK中join()实现的核心代码片段：

while(isAlive()) {

wait(0);

}

可以看到，它让调用线程在当前线程对象上进行等待。当线程执行完成后，被等待的线程会在退出前调用notifyAll()通知所有的等待线程继续执行。因此，值得注意的一点是：不要在应用程序中，在Thread对象实例上使用类似wait()或者notify()等方法，因为这很有可能会影响系统API的工作，或者被系统API所影响。

另外一个比较有趣的方法，是Thread.yield()，它的定义如下：

public static native void yield();

这是一个静态方法，一旦执行，它会使当前线程让出CPU。但要注意，让出CPU并不表示当前线程不执行了。当前线程在让出CPU后，还会进行CPU资源的争夺，但是是否能够再次被分配到，就不一定了。因此，对Thread.yield()的调用就好像是在说：我已经完成一些最重要的工作了，我应该是可以休息一下了，可以给其他线程一些工作机会啦！

## 2.3 volatile与Java内存模型(JMM)

volatile并不能代替锁，它也无法保证一些复合操作的原子性。

## 2.4 分门别类的管理：线程组

在一个系统中，如果线程数量很多，而且线程数量很多，而且功能分配比较明确，就可以将相同功能的线程放置在一个线程组里。

## 2.5 驻守后台：守护线程(Daemon)

守护线程是一种特殊的线程，就和它的名字一样，它是系统的守护者，在后台默默地完成一些系统性的服务，比如垃圾回收线程、JIT线程就可以理解为守护线程。与之相对应的是用户线程，用户线程可以认为是系统的工作线程，它会完成这个程序应该要完成的业务操作。如果用户线程全部结束，这也意味着这个程序实际上无事可做了。守护线程要守护的对象已经不存在了，那么整个应用程序就自然应该结束。因此，当一个Java应用内，只有守护线程时，虚拟机就会自然退出。

## 2.6 先干重要的事：线程优先级

Java中的线程可以有自己的优先级。优先级高的线程在竞争资源时会更有优势，更可能抢占资源，当然，这只是一个概率问题。如果运气不好，高优先级线程可能也会抢占失败。由于线程的优先级调度和底层操作系统有密切的关系，在各个平台上表现不一，并且这种优先级产生的后果也可能不容易预测，无法精准控制，比如一个低优先级的线程可能一直抢占不到资源，从而始终无法运行，而产生饥饿。因此，在要求严格的场合，还是需要自己在应用层解决线程调度问题。

在Java中，使用1到10表示线程优先级。一般可以使用内置的三个静态标量表示：

public final static int MIN\_PRIORITY = 1;

public final static int NORM\_PRIORITY = 5;

public final static int MAX\_PRIORITY = 10;

数字越大则优先级越高，但有效范围在1到10之间。

## 2.7 线程安全的概念与synchronized

关键字synchronized的作用是实现线程间的同步。它的工作是对同步的代码加锁，使得每一次，只能有一个线程进入同步块，从而保证线程间的安全性。

关键字synchronized可以有多种用法。这里做一个简单的整理。

1)指定加锁对象：对给定对象加锁，进入同步代码前要获得给定对象的锁。

2)直接作用于实例方法：相当于对当前实例加锁，进入同步代码前要获得当前实例的锁。

3)直接作用于静态方法：相当于对当前类加锁，进入同步代码前要获得当前类的锁。

## 2.8 程序中的幽灵：隐蔽的错误

最可怕的情况是：系统没有任何异常表现，没有日志，也没有堆栈，但是却给出了一个错误的执行结果，这种情况下，才真会让你抓狂。

### 2.8.1 无提示的错误案例

典型的例子是整型溢出。

### 2.8.2 并发下的ArrayList

ArrayList在扩容的过程中，内部一致性被破坏，但由于没有锁的保护，另外一个线程访问到了不一致的内部状态，导致出现越界问题。

### 2.8.3 并发下诡异的HashMap

HashMap同样不是线程安全的。当你使用多线程访问HashMap时，也可能会遇到意想不到的错误。

### 2.8.4 初学者常见问题：错误的加锁

加锁必须是合理的。

# 第3章：JDK并发包

## 3.1 多线程的团队协作：同步控制

### 3.1.1 synchronized的功能扩展：重入锁

重入锁可以完全替代synchronized关键字。在JDK 5.0的早起版本中，重入锁的性能远远好于synchronized，但从JDK 6.0开始，JDK在synchronized上做了大量的优化，使得两者的性能差距并不大。

重入锁使用java.util.concurrent.locks.ReentrantLock类来实现。

与synchronized相比，重入锁有着显式的操作过程。开发人员必须手动指定何时加锁，何时释放锁。也正因为这样，重入锁对逻辑控制的灵活性要远远好于synchronized。但值得注意的是，在退出临界区时，必须记得释放锁。

可重入锁是可以反复进入的，当然，这里的反复仅仅局限于一个线程。下面的代码依然成立：

lock.lock();

lock.lock();

try {

i ++;

} finally {

lock.unlock();

lock.unlock();

}

在这种情况下，一个线程连续两次获得同一把锁。这是允许的！如果不允许这么操作，那么同一个线程在第2次获得锁时，将会和自己产生死锁。程序就会“卡死”在第2次申请锁的过程中。但需要注意的是，如果同一个线程多次获得锁，那么在释放锁的时候，也必须释放相同次数。如果释放的次数多，那么会得到一个java.lang.IllegalMonitorStateException异常，反之，如果释放锁的次数少了，那么相当于线程还持有这个锁，因此，其他线程也无法进入临界区。

**中断响应**

对于synchronized来说，如果一个线程在等待锁，那么结果只有两种情况，要么它获得这把锁继续执行，要么它就保持等待。而使用重入锁，则提供另外一种可能，那就是线程可以被中断。也就是在等待锁的过程中，程序可以根据需要取消对锁的请求。

中断正式提供了一套类似的机制。如果一个线程正在等待锁，那么它依然可以收到一个通知，被告知无须再等待，可以停止工作了。这种情况对于处理死锁是有一定帮助的。

**锁申请等待限时**

除了等待外部通知之外，要避免死锁还有另外一种方法，那就是限时等待。通常，我们无法判断为什么一个线程迟迟拿不到锁。也许是因为死锁了，也许是因为产生了饥饿。但如果给定一个等待时间，让线程自动放弃，那么对系统来说是有意义的。我们可以使用tryLock()方法进行一次限时的等待。

ReentrantLock.tryLock()方法也可以不带参数直接运行。在这种情况下，当前线程会尝试获得锁，如果锁并未被其他线程占用，则申请锁会成功，并立即返回true。如果锁被其他线程占用，则当前线程不会进行等待，而是立即返回false。

**公平锁**

在大多数情况下，锁的申请都是非公平的。也就是说，线程1首先请求了锁A，接着线程2也请求了锁A。那么当锁A可用时，是线程1可以获得锁还是线程2可以获得锁呢？这是不一定的。系统只会从这个锁的等待队列中随机挑选一个。因此不能保证其公平性。如果我们使用synchronized关键字进行锁控制，那么产生的锁就是非公平的。而重入锁允许我们对其公平性进行设置。它有一个如下的构造函数：

public ReentrantLock(Boolean fair)

当参数fair为true时，表示锁是公平的。公平锁看起来很优美，但是要实现公平锁必然要求系统维护一个有序队列，因此公平锁的实现成本比较高，性能相对也非常低下，因此，默认情况下，锁是非公平的。如果没有特别的需求，也不需要使用公平锁。公平锁和非公平锁在线程调度表现上也是非常不一样的。

**重入锁总结：**

就重入锁的实现来看，它主要集中在Java层面。在重入锁的实现中，主要包含三个要素：

第一，是原子状态。原子状态使用CAS操作来存储当前锁的状态，判断锁是否已经被别的线程持有。

第二，是等待队列。所有没有请求到锁的线程，会进入等待队列进行等待。待有线程释放锁后，系统就能从等待队列中唤醒一个线程，继续工作。

第三，是阻塞原语pack()和unpark()，用来挂起和恢复线程。没有得到锁的线程将会被挂起。

### 3.1.2 重入锁的好搭档：Condition条件

通过Lock接口(重入锁就实现了这一接口)的Condition newCondition()方法可以生成一个与当前重入锁绑定的Condition实例。利用Condition对象，我们就可以让线程在合适的时间等待，或者在某一个特定的时刻得到通知，继续执行。

### 3.1.3 允许多个线程同时访问：信号量(Semaphore)

信号量可以指定多个线程，同时访问某一个资源。

### 3.1.4 ReadWriteLock读写锁

读写锁互斥规则：

1)读-读不互斥：读读之间不阻塞。

2)读-写互斥：读阻塞写，写也会阻塞读。

3)写-写互斥：写写阻塞。

### 3.1.5 倒计时器：CountDownLatch

让所有的线程达到指定点之后再开始执行。

### 3.1.6 循环栅栏：CyclicBarrier

比CountDownLatch略微强大一些，CyclicBarrier可以接收一个参数作为barrierAction。所谓barrierAction就是当计数器一次计数完成后，系统会执行的动作。

### 3.1.7 线程阻塞工具类：LockSupport

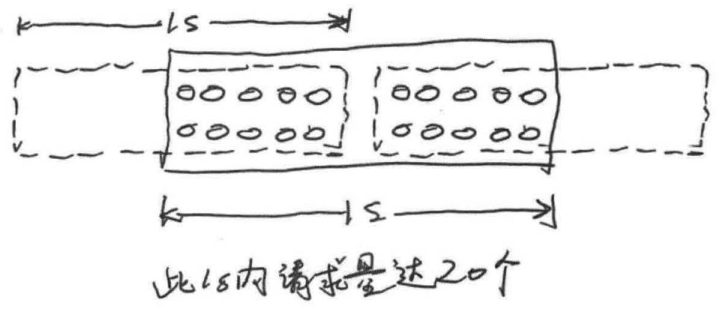
LockSupport是一个非常方便实用的线程阻塞工具，它可以在线程内任意位置让线程阻塞。和Thread.suspend()相比，它弥补了由于resume()在前发生，导致线程无法继续执行的情况。和Object.wait()相比，它不需要先获得某个对象的锁，也不会抛出InterruptedException异常。

LockSupport的静态方法park()可以阻塞当前线程，类似的还有parkNanos()、parkUntil()等方法。它们实现了一个限时的等待。

### 3.1.8 Guava和RateLimiter限流

任何应用和模块组件都有一定的访问速率上限，如果请求速率突破了这个上限，不但多余的请求无法处理，甚至会压垮系统使所有的请求均无法有效处理。因此，对请求进行限流是非常必要的。RateLimiter正是这么一款限流工具。

一种简单的限流算法就是给出一个单位时间，然后使用一个计数器counter统计单位时间内收到的请求数量，当请求数量超过门限时，余下的请求丢弃或者等待。但这种简单的算法有一个严重的问题，就是很难控制边界时间上的请求。假设单位时间是1秒，每秒请求不超过10个。如果在这一秒的前半秒没有请求，而后半秒有10个请求，下一秒的前半秒又有10个请求，那么在这中间的一秒内，就会合理处理20个请求，而这明显违反了限流的基本需求。这是一种简单粗暴的总数量限流而不是平均限流。



计数器限流算法的问题

因此，更为一般化的限流算法有两种：漏桶算法和令牌桶算法。

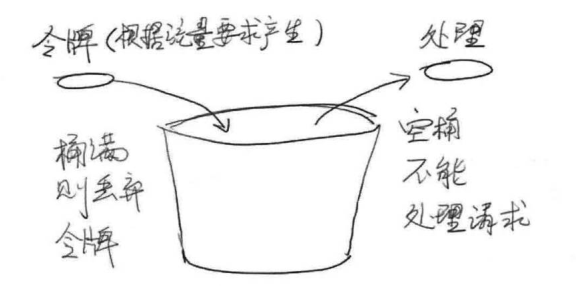
漏桶算法的基本思想是：利用一个缓冲区，当有请求进入系统时，无论请求的速率如何，都先在缓存区内保存，然后以固定的流速流出缓存区进行处理。如下图：



漏桶算法示意图

漏桶算法的特点是无论外部请求压力如何，漏桶算法总是以固定的流速处理数据。漏桶的容积和流出速率是该算法的两个重要参数。

令牌桶算法是一种反向的漏桶算法。在令牌桶算法中，桶中存放的不再是请求，而是令牌。处理程序只有拿到令牌后，才能对请求进行处理。如果没有令牌，那么处理程序要么丢弃请求，要么等待可用的令牌。为了限制流速，该算法在每个单位时间产生一定量的令牌存入桶中。比如，要限定应用每秒只能处理一个请求，那么令牌桶就会每秒产生1个令牌。通常，桶的容量是有限的，比如，当令牌没有被消耗掉时，只能累计有限单位时间内的令牌数量，其基本原理如下：



RateLimiter正是采用了令牌桶算法。

## 3.2 线程复用：线程池

注意：在实际生产环境中，线程的数量必须得到控制。盲目的大量创建线程对系统性能是有伤害的。

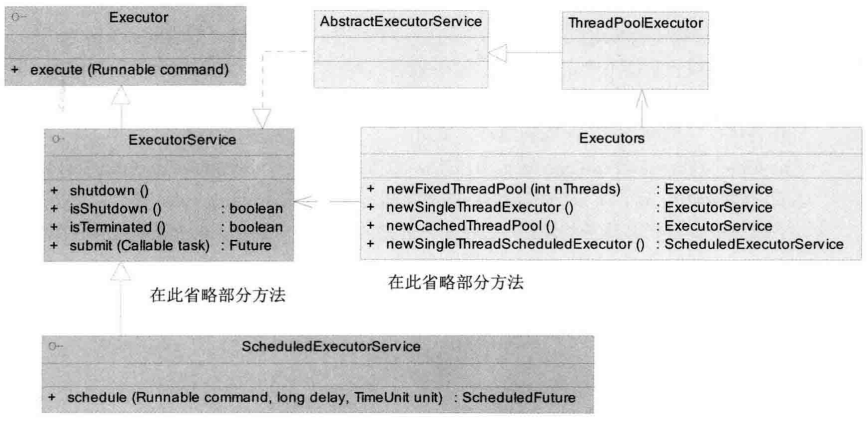
### 3.2.1 什么是线程池？

线程池中，总有那么几个活跃线程。当你需要使用线程时，可以从池子中随便拿一个空闲线程，当完成工作时，当完成工作时，并不急着关闭线程，而是将这个线程退回到池子，方便其他人使用。

简而言之，在使用线程后，创建线程变成了从线程池获得空闲线程，关闭线程变成了向线程池归还线程。

### 3.2.2 不要重复发明轮子：JDK对线程池的支持

JDK提供的Executor框架。



以上成员均在java.util.concurrent包中，是JDK并发包的核心类。其中ThreadPoolExecutor表示一个线程池。Executors类则扮演着线程池工厂的角色，通过Executors可以取得一个拥有特定功能的线程池。从UML图中亦可知，ThreadPoolExecutor类实现了Executor接口，因此通过这个接口，任何Runnable的对象都可以被ThreadPoolExecutor线程池调度。

Executor框架提供了各种类型的线程池，主要有以下方法：

public static ExecutorService newFixedThreadPool(int nThreads)

public static ExecutorService newSingleThreadExecutor()

public static ExecutorService newCachedThreadPool()

public static ScheduleExecutorService newSingleThreadScheduledExecutor()

public static ScheduleExecutorService newScheduledThreadPool(int corePoolSize)

以上工厂方法分别返回具有不同工作特性的线程池。这些线程池工厂方法的具体说明如下。

1)newFixedThreadPool()方法：该方法返回一个固定线程数量的线程池。该线程池中的线程数量始终不变。当有一个新的任务提交时，线程池中若有空闲线程，则立即执行。若没有，则新的任务会被暂存在一个任务队列中，待有线程空闲时，便处理在任务队列中的任务。

2)newSingleThreadExecutor()方法：该方法返回一个只有一个线程的线程池。若多余一个任务被提交到该线程池，任务会被保存在一个任务队列中，待线程空闲，按先入先出的顺序执行队列中的任务。

3)newCachedThreadPool()方法：该方法返回一个可根据实际情况调整线程数量的线程池。线程池的线程数量不确定，但若有空闲线程可以复用，则会优先使用可复用的线程。若所有线程均在工作，又有新的任务提交，则会创建新的线程处理任务。所有线程在当前任务执行完毕后，将返回线程池进行复用。

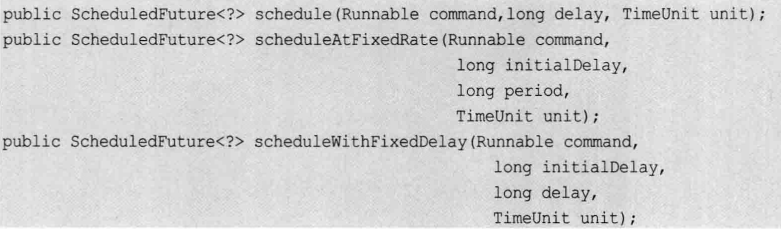
4)newSingleThreadScheduledExecutor()方法：该方法返回一个ScheduledExecutorService对象，线程池大小为1。ScheduledExecutorService接口在ExecutorService接口之上扩展了在给定时间执行某任务的功能，如在某个固定的延时之后执行，或者周期性执行某个任务。

5)newScheduledThreadPool()方法：该方法也返回一个ScheduledExecutorService对象，但该线程池可以指定线程数数量。

1. 固定大小的线程池

2. 计划任务

另外一个值得注意的方法是newScheduledThreadPool()。它返回一个ScheduledExecutorService对象，可以根据时间需要对线程进行调度。它的一些主要方法如下：



与其他几个线程池不同，ScheduledExecutorService并不一定会立即安排执行任务。它其实是起到了计划任务的作用。它会在指定的时间，对任务进行调度。

方法schedule()会在给定时间，对任务进行一次调度。方法scheduleAtFixedRate()和scheduleWithFixedDelay()会对任务进行周期性的调度。但是两者有一点小小的区别，对于FixedRate方式来说，任务调度的频率是一定的。它是以上一个任务开始执行时间为起点，之后的period时间，调度下一次任务。而FixDelay则是在上一个任务结束后，再经过delay时间进行任务调度。

如果任务的执行时间超过调度时间，会发生什么情况呢？比如调度周期是2秒，如果任务的执行时间是8秒，是不是会出现多个任务堆叠在一起呢？

实际上，ScheduledExecutorService不会让任务堆叠出现。如果将执行时间修改为8秒，就会发现任务的执行周期不再是2秒，而是变成了8秒。

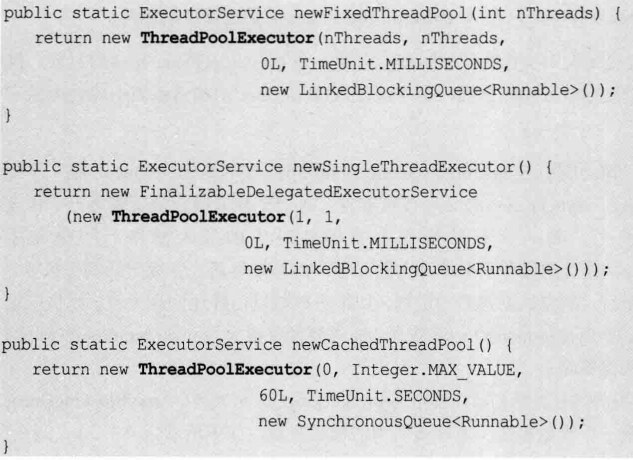
也就是说，周期如果太短，那么任务就会在上一个任务结束后，立即被调用。可以想象，如果采用scheduleWithFixedDelay()，并且按照修改8秒，调度周期2秒计，那么任务的实际间隔将是10秒，大家可以自行尝试。

另一个值得注意的问题是，调度程序实际上并不保证任务会无限期的持续调用。如果任务本身抛出了异常，那么后续的所有执行都会被中断，因此，如果你想让你的任务持续稳定的执行，那么做好异常处理就非常重要，否则，你很有可能观察到你的调度器无疾而终。

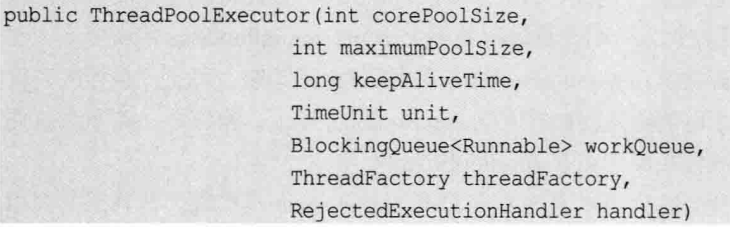
注意：如果任务遇到异常，那么后续的所有子任务都会停止调度，因此，必须保证异常被及时处理，为周期性任务的稳定调度提供条件。

### 3.2.3 刨根究底：核心线程池的内部实现

对于核心的几个线程池，无论是newFixedThreadPool()方法、newSingleThreadExecutor()还是newCachedThreadPool()方法，虽然看起来创建的线程有着完全不同的功能特点，但其内部实现均使用了ThreadPoolExecutor实现。下面给出了这三个线程池的实现方式：



由以上线程池的实现代码可以看到，它们都只是ThreadPoolExecutor类的封装。为何ThreadPoolExecutor有如此强大的功能呢？来看一下ThreadPoolExecutor最重要的构造函数：



函数的参数含义如下：

1)corePoolSize：指定了线程池中的线程数量。

2)maximumPoolSize：指定了线程池中的最大线程数量。

3)keepAliveTime：当线程池线程数量超过corePoolSize时，多余的空闲线程的存活时间。即，超过corePoolSize的空闲线程，在多长时间内，会被销毁。

4)unit：keepAliveTime的单位。

5)workQueue：任务队列，被提交但尚未被执行的任务。

6)threadFactory：线程工厂，用于创建线程，一般用默认的即可。

7)handler：拒绝策略。当任务太多来不及处理，如何拒绝任务。

以上参数，大部分都很简单，只有workQueue和handler需要进行详细说明。

参数workQueue指被提交但未被执行的任务队列，它是一个BlockingQueue接口的对象，仅用于存放Runnable对象。根据队列功能分类，在ThreadPoolExecutor的构造函数中可使用以下几种BlockingQueue。

1)直接提交的队列：该功能由SynchronousQueue对象提供。SynchronousQueue是一个特殊的BlockingQueue。SynchronousQueue没有容量，每一个插入操作都要等待一个相应的删除操作，反之，每一个删除操作都要等待对应的插入操作。如果使用SynchronousQueue，提交的任务不会被真实的保存，而总是将新任务提交给线程执行，如果没有空闲的进程，则尝试创建新的进程，如果进程数量已经达到最大值，则执行拒绝策略。因此，使用SynchronousQueue队列，通常要设置很大的maximumPoolSize值，否则很容易执行拒接策略。

2)有界的任务队列：有界的任务队列可以使用ArrayBlockingQueue实现。ArrayBlockingQueue的构造函数必须带一个容量参数，表示该队列的最大容量，如下所示。

public ArrayBlockingQueue(int capacity)

当使用有界的任务队列时，若有新的任务需要执行，如果线程池的实际线程数小于corePoolSize，则会优先创建新的线程，若大于corePoolSize，则会将新任务加入等待队列。若等待队列已满，无法加入，则在总线程数不大于maximumPoolSize的前提下，创建新的进程执行任务。若大于maximumPoolSize，则执行拒绝策略。可见，有界队列仅当在任务队列装满时，才可能将线程数提升到corePoolSize以上，换言之，除非系统非常繁忙，否则确保核心线程数维持在corePoolSize。

3)无界的任务队列：无界任务队列可以通过LinkedBlockingQueue类实现。与有界队列相比，除非系统资源耗尽，否则无界的任务队列不存在任务入队失败的情况。当有新的任务到来，系统的线程数小于corePoolSize时，线程池会生成新的线程执行任务，但当系统的线程数达到corePoolSize后，就不会继续增加。若后续仍有新的任务加入，而又没有空闲的线程资源，则任务直接进入队列等待。若任务创建和处理速度差异很大，无界队列会保持快速增长，直到耗尽系统内存。

4)优先任务队列：优先任务队列是带有执行优先级的队列。它通过PriorityBlockingQueue实现，可以控制任务的执行先后顺序。它是一个特殊的无界队列。无论是有界队列ArrayBlockingQueue，还是未指定大小的无界队列LinkedBlockingQueue都是按照先进先出算法处理任务的。而PriorityBlockQueue则可以根据任务自身的优先级顺序先后执行，在确保系统性能的同时，也能有很好的质量保证(总是确保高优先级的任务先执行)。

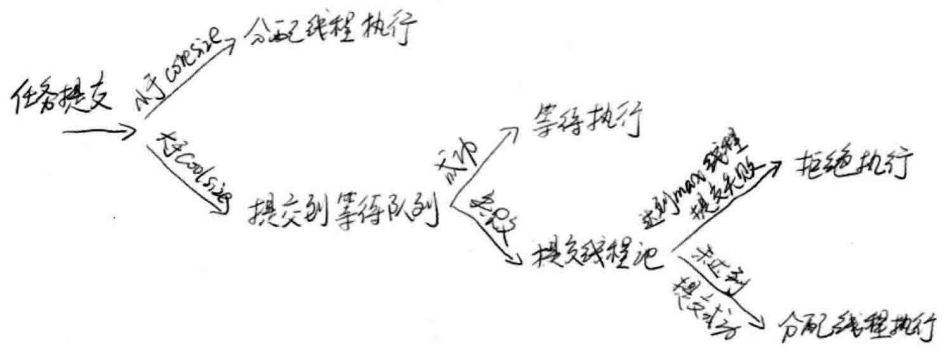
newFixedThreadPool()方法的实现。它返回了一个corePoolSize和maximumPoolSize大小一样的，并且使用了LinkedBlockingQueue任务队列的线程池。因为对于固定大小的线程池而言，不存在线程数量的动态变化，因此corePoolSize和maximumPoolSize可以相等。同时，它使用无界队列存放无法立即执行的任务，当任务提交非常频繁的时候，该队列可能迅速膨胀，从而耗尽系统资源。

newSingleThreadExecutor()返回的单线程线程池，是newFixedThreadPool()方法的一种退化，只是简单的将线程池线程数量设置为1。

newCachedThreadPool()方法返回corePoolSize为0，maximumPoolSize无穷大的线程池，这意味着在没有任务时，该线程池内无线程，而当任务被提交时，该线程池会使用空闲的线程执行任务，若无空闲线程，则将任务加入SynchronousQueue队列，而SynchronousQueue队列是一种直接提交的队列，它总会迫使线程池增加新的线程执行任务。当任务执行完毕后，由于corePoolSize为0，因此空闲线程又会在指定时间内(60秒)被回收。

对于newCachedThreadPool()，如果同时有大量任务被提交，而任务的执行又不那么快时，那么系统便会开启等量的线程处理，这样的做法可能会很快消耗系统的资源。

注意：使用自定义线程池时，要根据应用的具体情况，选择合适的并发队列作为任务的缓冲。当线程资源紧张时，不同的并发队列对系统行为和性能的影响均不同。



ThreadPoolExecutor的任务调度逻辑

### 3.2.4 超负载了怎么办：拒绝策略

ThreadPoolThread的最后一个参数指定了拒绝策略。也就是当任务数量超过系统实际承载能力时，该如何处理呢？这时就要用到拒绝策略了。拒绝策略可以说是系统超负荷运行时的补救措施，通常由于压力太大而引起的，也就是线程池中的线程已经用完了，无法继续为新任务服务，同时，等待队列中也已经排满了，再也塞不下任务了。这时，我们就需要有一套机制，合理地处理这个问题。

JDK内置提供了四种拒绝策略：

1)AbortPolicy策略：该策略会直接抛出异常，阻止系统正常工作；

2)CallerRunsPolicy策略：只要线程池未关闭，该策略直接在调用者线程中，运行当前被丢弃的任务。显然这样做不会真的丢弃任务，但是，任务提交线程的性能极有可能会急剧下降。

3)DiscardOldestPolicy策略：该策略将丢弃最老的一个请求，也就是即将被执行的一个任务，并尝试再次提交当前任务。

4)DiscardPolicy策略：该策略默默地丢弃无法处理的任务，不予任何处理。如果允许任务丢失，这可能是最好的一种方案了。

以上内置的策略均实现了RejectedExecutionHandler接口，若以上策略仍无法满足实际应用需要，完全可以自己扩展RejectedExecutionHandler接口。RejectedExecutionHandler的定义如下：

public interface RejectedExecutionHandler {

void rejectedExecution(Runnable r, ThreadPoolExecutor executor);

}

其中r为请求执行的任务，executor为当前的线程池。

### 3.2.5 自定义线程创建：ThreadFactory

线程池中的线程从哪里来呢？

线程池的主要作用是为了线程复用，也就是避免了线程的频繁创建。但是，最开始的那些线程从何而来呢？答案就是ThreadFactory。

ThreadFactory是一个接口，它只是一个方法，用来创建线程：

Thread newThread(Runnable r);

当线程池需要新建线程时，就会调用这个方法。

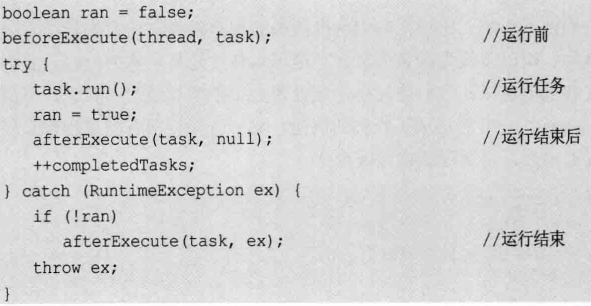
自定义线程池可以帮助我们做不少事。比如，我们可以跟踪线程池究竟在何时创建了多少线程，也可以自定义线程的名称、组以及优先级等信息，甚至可以任性地将所有的线程设置为守护线程。总之，使用自定义线程池可以让我们更加自由地设置池子中所有线程的状态。

### 3.2.6 我的应用我做主：扩展线程池

有时候我们需要对线程池做一些扩展，比如，我们想监控每个任务执行的开始和结束时间，或者其他一些自定义的增强功能，这时候该怎么办呢？

一个好消息是：ThreadPoolExecutor也是一个可以扩展的线程池。它提供了beforeExecute()、afterExecute()和terminate()三个接口对线程池进行监控。

以beforeExecute()、afterExecute()为例，在ThreadPoolExecutor.Worker.runTask()方法内部提供了这样的实现：



ThreadPoolExecutor.Worker是ThreadPoolExecutor的内部类，它是一个实现了Runnable接口的类。ThreadPoolExecutor线程池中的工作线程也正是Worker实例。Worker.runTask()方法会被线程池以多线程模式异步调用，即Worker.runTask()会同时被多个线程访问。因此其beforeExecute()、afterExecute()接口也将同时多线程访问。

在默认的ThreadPoolExecutor实现中，提供了空的beforeExecute()和afterExecute()实现。在实际应用中，可以对其进行扩展来实现对线程池运行状态的跟踪，输出一些有用的调试信息，以帮助系统故障诊断，这对于多线程程序错误排查是很有帮助的。

### 3.2.7 合理的选择：优化线程池线程数量

线程池的大小对系统的性能有一定的影响。过大或者过小的线程数量都无法发挥最优的系统性能，但是线程池大小的确也不需要做得非常精确，因为只要避免极大和极小两种情况，线程池的大小对系统的性能并不会影响太大。一般来说，确定线程池的大小需要考虑CPU数量、内存大小等因素。在《Java Concurrency in Practice》一书中给出了一个估算线程池大小的经验公式：

Ncpu = CPU的数量

Ucpu = 目标CPU的使用率，0<=Ucpu<=1

W/C = 等待时间与计算时间的比率

为了保持处理器达到期望的使用率，最优的池的大小等于：

Nthreads = Ncpu \* Ucpu \* (1 + W/C)

在Java中，可以通过：

Runtime.getRuntime().availableProcessors()

取得可用的CPU数量。

### 3.2.8 堆栈去哪里了：在线程池中寻找堆栈

如果使用了多线程，诡异的错误会变得更加常见。

使用线程池虽然是好事，但是还是得出处留意这些“坑”。线程池很有可能会“吃”掉程序抛出的异常，导致我们对程序的错误一无所知。

通过扩展ThreadPoolExecutor线程池的方式，让它在调度任务之前，先保存一下提交任务线程的堆栈信息。

### 3.2.9 分而治之：Fork/Join框架

“分而治之”一直是一个非常有效地处理大量数据的方法。著名的MapReduce也是采取了分而治之的思想。

在JDK中，给出了一个ForkJoinPool线程池，对于fork()方法并不急着线程，而是提交给ForkJoinPool线程池进行处理，以节省系统资源。

由于线程池的优化，提交的任务和线程数量并不是一对一的关系。在绝大多数情况下，一个物理线程实际上是需要处理多个逻辑任务的。因此，每个线程必然需要拥有一个任务队列。因此，在实际执行过程中，可能遇到这么一种情况：线程A就会“帮助”线程B，从线程B的任务队列中拿一个任务过来处理，尽可能地达到平衡。一个值得注意的地方是，当线程试图帮助别人时，总是从任务队列的底部开始拿数据，而线程试图执行自己的任务时，则是从相反的顶部开始拿。因此，这种行为也十分有利于避免数据竞争。

ForkJoinPool有一个重要的接口：

public <T> ForkJoinTask<T> submit(ForkJoinTask<T> task)

你可以向ForkJoinPool线程池提交一个ForkJoinTask任务。所谓ForkJoinTask任务就是支持fork()分解以及join()等待的任务。ForkJoinTask有两个重要的子类，RecursiveAction和RecursiveTask。它们分别表示没有返回值的任务和可以携带返回值的任务。

在使用ForkJoin时需要注意，如果任务的划分层次很深，一直得不到返回，那么可能出现两种情况：第一，系统内的线程数量越积越多，导致性能严重下降。第二，函数的调用层次变得很深，最终导致栈溢出。

ForkJoin线程池使用一个无锁的栈来管理空闲线程。如果一个工作线程暂时取不到可用的任务，则可能会被挂起，挂起的线程将会被压入由线程池维护的栈中。待将来有任务可用时，再从栈中唤醒这些线程。

### 3.2.10 Guava中对线程池的扩展

除了JDK内置的线程池以外，Guava对线程池也进行了一定的扩展，主要体现在MoreExecutors工具类中。

1. 特殊的DirecExecutor线程池

在MoreExecutors中，提供了一个简单但是非常重要的线程池实现，即DirectExecutor线程池。DirectExecutor线程池很简单，它并没有真的创建或者使用额外线程，它总是将任务在当前线程中直接执行。读者也许会觉得奇怪，为什么需要这么一个线程池呢？这是软件设计上的需要。

从软件设计的角度上说，抽象是软件设计的根本和精髓。将不同业务的共同属性提取并抽象成模型非常有利于对不同业务的统一处理。我们总是希望并且倾向于使用通用的代码来处理不同的场景，因此，这就需要不同场景进行统一的抽象和建模。

对于线程池来说，其技术目的是为了复用线程以提高运行效率，但其业务需求却是去异步执行一段业务指令。但是有时候，异步并不是必要的。因此，当我们剥去线程池的技术细节，仅关注其使用场景便不难发现，任何一个可以运行Runnable实例的模块都可以被视为线程池，即便它没有真正创建线程。这样就可以将异步执行和同步执行进行统一，使用统一的编码风格来处理同步和异步调用，进而简化设计。

**2. Daemon线程池**

此外，在MoreExecutors中，还提供了将普通线程池转为Daemon线程池的方法。在很多场合，我们并不希望后台线程池阻止程序的退出，当系统执行完成后，即便有线程池存在，依然希望进程结束执行。此时，就可以使用MoreExecutors.getExitingExecutorService()方法。

3. 对Future模式的扩展

放在5.5节扩展。

## 3.3 不要重复发明轮子：JDK的并发容器

### 3.3.1 超好用的工具类：并发集合简介

JDK提供的这些容器大部分在java.util.concurrent包中。

1)ConcurrentHashMap：这是一个高效的并发HashMap。你可以理解为一个线程安全的HashMap。

2)CopyOnWriteArrayList：这是一个List，从名字看就是和ArrayList是一族的。在读多写少的场合，这个List的性能非常好，远远好于Vector。

3)ConcurrentLinkedQueue：高效的并发队列，使用链表实现。可以看做一个线程安全的LinkedList。

4)BlockingQueue：这是一个接口，JDK内部通过链表、数组等方式实现了这个接口。表示阻塞队列，非常适合用于作为数据共享的通道。

5)ConcurrentSkipListMap：跳表的实现。这是一个Map，使用跳表的数据结构进行快速查找。

除了以上并发包中的专有数据结构外，java.util下的Vector是线程安全的(虽然性能和上述专用工具没得比)，另外Collections工具类可以帮助我们将任意集合包装成线程安全的集合。

### 3.3.2 线程安全的HashMap

如果需要一个线程安全的HashMap应该怎么做呢？一种可行的方法是使用Collections.synchronized()方法包装我们的HashMap。如下代码，产生的HashMap就是线程安全的：

public static Map m = Collections.synchronizedMap(new HashMap());

Collections.synchronizedMap()会生成一个名为SynchronizedMap的Map。它使用委托，将自己所有Map相关的功能交给传入的HashMap实现，而自己则主要负责保证线程安全。

如果并发级别不高，该方式是够用的。但是，在高并发环境中，我们也有必要寻求新的解决方案。

一个更加专业的并发HashMap是ConcurrentHashMap。它位于java.util.concurrent包内。它专门为并发进行了性能优化，因此，更加适合多线程的场合。

### 3.3.3 有关List的线程安全

队列、链表之类的数据结构也是极其常见的，几乎所有的应用程序都会与之相关。在Java中，ArrayList和Vector都是使用数组作为其内部实现。两者最大的不同在于Vector是线程安全的，而ArrayList不是。此外，LinkedList使用链表的数据结构实现了List。但是很不幸，LinkedList并不是线程安全的，不过参考前面对HashMap的包装，在这里我们也可以使用Collections.synchronizedList()方法来包装任意List，如下所示：

public static List<String> l = Collections.synchronizedList(new LinkedList<>());

此时生成的List对象就是线程安全的。

### 3.3.4 高效读写的队列：深度剖析ConcurrentLinkedQueue

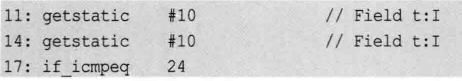
队列Queue也是常用的数据结构之一。在JDK中提供了一个ConcurrentLinkedQueue类用来实现高并发的队列。从名字可以看到，这个队列使用链表作为其数据结构。有关ConcurrentLinkedQueue的性能测试，大家可以自行尝试。ConcurrentLinkedQueue应该算是高并发环境中性能最好的队列就可以了。它之所以能有很好的性能，是因为其内部复杂的实现。

其他内容看书，下面是重点：

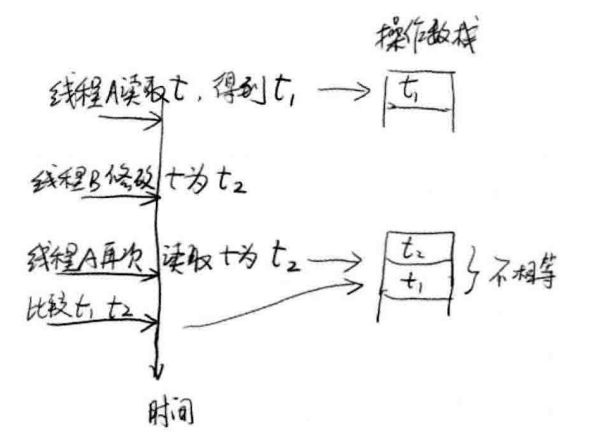
p = (t != (t = tail)) ? t : head;

这句话虽然只有短短一行，但是包含的信息比较多。首先“!=”并不是原子操作，它是可以被中断的。也就是说，在执行“!=”时，程序会先取得t的值，再执行t = tail，并取得新的t的值。然后比较这两个值是否相等。在单线程时，t != t这种语句显然不会成立。但是在并发环境中，有可能在获得左边的t值后，右边的t值被其他线程修改。这样，t != t就可能成立。这里就是这种情况。如果在比较过程中，tail被其他线程修改，当它再次赋值给t时，就会导致等式左边的t和右边的t不同。如果两个t不相同，表示tail在中途被其他线程篡改。这时，我们就可以用新的tail作为链表末尾，也就是这里等式右边的t。但如果tail没有被修改，则返回head，要求从头部开始，重新查找尾部。

作为简化题，我们将考察t != t的字节码(注意这里假设t为静态整型变量)：



可以看到，在字节码层面，t被先后取了两次，在多线程环境下，我们自然无法保证两次对t的取值会是相同的，如下图，显示了这种情况：



### 3.3.5 高效读取：不变模式下的CopyOnWriteArrayList

在很多应用场景中，读操作可能会远远大于写操作。比如，有些系统级别的信息，往往只需要加载或者修改很少的次数，但是会被系统内所有的模块频繁的访问。对于这种场景，我们最希望看到的就是读操作可以尽可能地快，而写即使慢一些也没有太大关系。

由于读操作根本不会修改原有的数据，因此对于每次读取都进行加锁其实是一种资源浪费。

为了将读取的性能发挥到极致，JDK中提供了CopyOnWriteArrayList类。对它来说，读取是完全不用加锁的，并且更好的消息是：写入也不会阻塞读取操作。只有写入和写入之间需要进行同步等待。这样一来，读操作的性能就会大幅度提升。

所谓CopyOnWrite就是在写入操作时，进行一次自我复制。换句话说，当这个List需要修改时，我并不修改原有的内容(这对于保证当前在读线程的数据一致性非常重要)，而是对原有的数据进行一次复制，将修改的内容写入副本中。写完之后，再将修改完的副本替换原来的数据。这样就可以保证写操作不会影响读了。

要注意的是：读取代码没有任何同步操作和锁操作，理由就是内部数组array不会发生修改，只会被另外一个array替换，因此可以保证数据安全。

### 3.3.6 数据共享通道：BlockingQueue

我们既希望线程A能够通知线程B，又希望线程A不知道线程B的存在。这样，如果将来进行重构或者升级，我们完全可以不修改线程A，而直接把线程B升级为线程C，保证系统的平滑过渡。

与之前提到的ConcurrentLinkedQueue或者CopyOnWriteArrayList不同，BlockingQueue是一个接口，并非一个具体的实现。

这里主要介绍ArrayBlockingQueue和LinkedBlockingQueue。

ArrayBlockingQueue是基于数组实现的，更适合做有界队列，因为队列中可容纳的最大元素需要在队列创建时指定(毕竟数组的动态扩展不太方便)；LinkedBlockingQueue基于链表，适合做无界队列，或者那些边界值非常大的队列，因为其内部元素可以动态增加，它不会因为初值容量很大，而一口气吃掉你一大半的内存。

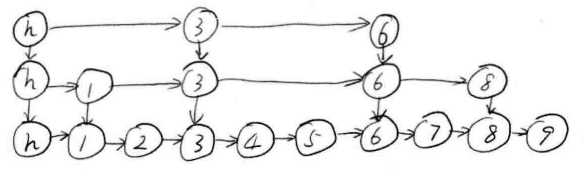
BlockingQueue之所以适合作为数据共享的通道，其关键还在于Blocking上。Blocking是阻塞的意思，当服务线程(服务线程指不断获取队列中的消息，进行处理的线程)处理完成队列中所有的消息后，它如何知道下一条消息何时到来呢？

BlockingQueue会让服务线程在队列为空时，进行等待，当有新的消息进入队列后，自动将线程唤醒。

### 3.3.7 随机数据结构：跳表(SkipList)

在JDK的并发包中，除了常用的哈希表外，还实现了一种有趣的数据结构----跳表。跳表是一种可以用来快速查找的数据结构，有点类似于平衡树。它们都可以对元素进行快速的查找。但是一个重要的区别是：对平衡树的插入和删除往往很可能导致平衡树进行一次全局的调整。而对跳表的插入和删除只需要对整个数据结构的局部进行操作即可。这样带来的好处是：在高并发的情况下，你会需要一个全局锁来保证整个平衡树的线程安全。而对于跳表，你只需要部分锁即可。这样，在高并发环境下，你就可以拥有更好的性能。而就查询的性能而言，跳表的时间复杂度也是O(log n)。所以在并发数据结构中，JDK使用跳表来实现一个Map。

跳表的另外一个特点是随机算法。跳表的本质是同时维护了多个链表，并且链表是分层的，如下图：



跳表结构示意图

最底层的链表维护了跳表内所有的元素，每上面一层链表都是下面一层的子集，一个元素插入哪些层完全是随机的。因此，如果你运气不好的话，你可能会得到一个性能很糟糕的结构。但是在实际工作中，它的表现是非常好的。

跳表内的所有链表的元素都是排序的。查找时，可以从顶级链表开始找。一旦发现被查找的元素大于当前链表中的取值，就会转入下一层链表继续找。这也就是说在查找过程中，搜索是跳跃式的。

使用跳表实现Map和使用哈希算法实现Map的另外一个不同之处是：哈希并不会保存元素的顺序，而跳表内所有的元素都是排序的。因此在对跳表进行遍历时，你会得到一个有序的结果。

## 3.4 使用JMH进行性能测试

由于业务原因引入的多线程有可能因为线程并发控制导致性能损耗，因此要评估损耗的比重是否可以接受。无论出自何种原因需要进行性能评估，量化指标总是必要的。在大部分场合，简单地回答谁快谁慢是远远不够的，如何将程序性能量化呢？这就是本节要介绍的Java微基准测试框架JMH。

### 3.4.1 什么是JMH

JMH(Java Microbenchmark Harness)是一个在OpenJDK项目中发布的，专门用于性能测试的框架，其精度可以达到毫秒级。通过JMH可以对多个方法的性能进行定量分析。比如，当要知道执行一个函数需要多少时间，或者当对一个算法有多种不同的实现时，需要选取性能最好的那个。

### 3.4.2 Hello JMH

### 3.4.3 JMH的基本概念和配置

为了能够更好地使用JMH的各项功能，首先需要对JMH的基本概念有所了解。

**1. 模式(Mode)**

Mode表示JMH的测量方式和角度，共有4种。

1)Throughput: 整体吞吐量，表示1秒内可以执行多少次调用。

2)AverageTime: 调用的平均时间，指每一次调用所需要的时间。

3)SampleTime: 随机取样，最后输出取样结果的分布，例如“99%的调用在xxx毫秒以内，99.99%的调用在xxx毫秒以内”。

4)SingleShotTime: 以上模式都是默认一次Iteration是1秒，唯有SingleShotTime只运行一次。往往同时把warmup次数设为0，用于测试冷启动时的性能。

**2. 迭代(Iteration)**

迭代是JMH的一次测试单位。在大部分测量模式下，一次迭代表示1秒。在这一秒内会不间断调用被测试方法，并采样计算吞吐量、平均时间等。

**3. 预热(Warmup)**

由于Java虚拟机的JIT的存在，同一个方法在JIT编译前后的时间将会不同。通常只考虑方法在JIT编译后的性能。

**4. 状态(State)**

通过State可以指定一个对象的作用范围，范围主要有两种。一种为线程范围，也就是一个对象只会被一个线程访问。在多线程池测试时，会为每一个线程生成一个对象。另一种是基准测试范围(Benchmark)，即多个线程共享一个实例。

**5. 配置类(Options/OptionsBuilder)**

在测试开始前，首先要对测试进行配置。通常需要指定一些参数，比如制定测试类(include)、使用的进程个数(fork)、预热迭代次数(warmupIterations)。在配置启动测试时，需要使用配置类。

### 3.4.4 理解JMH中的Mode

### 3.4.5 理解JMH中的State

JMH中的State可以理解为变量或者数据模型的作用域，通常包括整个Benchmark级别和Thread线程级别。

### 3.4.6 有关性能测试的一些思考

在大部分场景中，并没有绝对的快或者慢。性能需要从不同角度、不同场景进行评估和取舍。一个典型的例子就是时间复杂度和空间复杂度的关系。

对性能的优化和研究就是需要在各种不同的场景下，对组件进行全方位的性能分析，并结合实际应用情况进行取舍和权衡。

### 3.4.7 CopyOnWriteArrayList类与ConcurrentLinkedQueue类

# 第4章：锁的优化及注意事项

在多核时代，使用多线程可以明显地提高系统的性能。但事实上，使用多线程的方式会额外增加系统的开销。

## 4.1 有助于提高“锁”性能的几点建议

### 4.1.1 减少锁持有时间

对于使用锁进行并发控制的应用程序而言，在锁竞争过程中，单个线程对锁的持有时间与系统性能有着直接的关系。如果线程持有锁的时间很长，那么相对地，锁的竞争程度也就越激烈。

在开发程序的时候，应该尽可能地减少对某个锁的占有时间，以减少线程间互斥的可能。

一个较为优化的解决方案是，只在必要时进行同步，这样就能明显减少线程持有锁的时间，提高系统的吞吐量。

### 4.1.2 减少锁粒度

减小锁粒度也是一种削弱多线程锁竞争的有效手段。这种技术典型的使用场景就是ConcurrentHashMap类的实现。

对于ConcurrentHashMap，它内部进一步细分了若干个小的HashMap，称之为段(SEGMENT)。默认情况下，一个ConcurrentHashMap被进一步细分为16个段。

如果需要在ConcurrentHashMap中增加一个新的表项，并不是将整个HashMap加锁，而是首先根据hashcode得到该表项应该被存放到那个段中，然后对该段加锁，并完成put()操作。在多线程环境中，如果多个线程同时进行put()操作，只要被加入的表项不存放在同一个段中，则线程间便可以做到真正的并行。

由于默认有16个段，因此，如果够幸运的话，ConcurrentHashMap可以同时接受16个线程同时插入(如果都插入不同的段中)，从而大大提升其吞吐量。

### 4.1.3 读写分离锁来替换独占锁

使用读写分离锁来替代独占锁是减少锁粒度的一种特殊情况。在读多写少的场合，读写锁对系统性能是很有好处的。因为如果系统在读写数据时均只使用独占锁，那么读操作和写操作间、读操作和读操作间、写操作和写操作间均不能做到真正的并发，而且需要相互等待。而读操作本身不会影响数据完整性和一致性。因此，理论上讲，在大部分情况下，应该可以允许多线程同时读，读写锁正是实现了这种功能。

### 4.1.4 锁分离

在LinkedBlockingQueue的实现中，take()函数和put()函数分别实现了从队列中取得数据和往队列中增加数据的功能。虽然两个函数都对当前队列进行了修改操作，但由于LinkedBlockingQueue是基于链表的，因此，两个操作分别作用于队列的前端和尾端，从理论上说，两者并不冲突。

如果使用独占锁，则要求在两个操作进行时获取当前队列的独占锁，那么take()和put()操作就不可能真正的并发，在运行时，它们会彼此等待对方释放锁资源。在这种情况下，锁竞争会相对比较激烈，从而影响程序在高并发时的性能。

通过takeLock和putLock两个锁，LinkedBlockingQueue实现了取数据和写数据的分离，使两者在真正意义上称为可并发的操作。

### 4.1.5 锁粗化

如果对同一个锁不停地进行请求、同步和释放，其本身也会消耗系统宝贵的资源，反而不利于性能的优化。

为此，虚拟机在遇到一连串连续地对同一锁不断进行请求和释放的操作时，便会把所有的锁操作整合成对锁的一次请求，从而减少对锁的请求同步失败，这个操作叫做锁的粗话。

## 4.2 Java虚拟机对锁优化所做的努力

### 4.2.1 偏向锁

偏向锁是一种针对加锁操作的优化手段。它的核心思想是：如果一个线程获得了锁，那么锁就进入偏向模式。当这个线程再次请求锁时，无须再做任何同步操作。这样就节省了大量有关锁申请的操作，从而提高了程序性能。因此，对于几乎没有锁竞争的场合，偏向锁有比较好的优化效果，因为连续多次极有可能是同一个线程请求相同的锁。而对于锁竞争比较激烈的场合，其效果不佳。因为在竞争激烈的场合，最有可能的情况是每次都是不同的线程来请求相同的锁。这样偏向模式会失效，因此还不如不启用偏向锁。使用Java虚拟机参数-XX:+UseBiasedLocking可以开启偏向锁。

### 4.2.2 轻量级锁

如果偏向锁失败，虚拟机并不会立即挂起线程。它还会使用一种称为轻量级锁的优化手段。轻量级锁的操作也很轻便，它只是简单地将对象头部作为指针，指向持有锁的线程堆栈的内部，来判断一个线程是否持有对象锁。如果线程获得轻量级锁成功，则可以顺利进入临界区如果轻量级锁加锁失败，则表示其他线程抢险争夺到了锁，那么当前线程的锁请求就会膨胀为重量级锁。

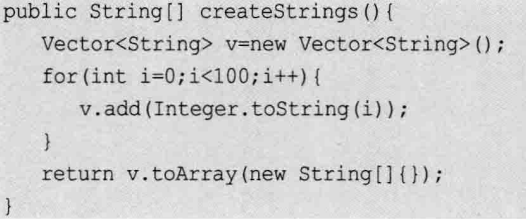
### 4.2.3 自旋锁

锁膨胀后，虚拟机为了避免线程真实地在操作系统层面挂起，虚拟机还会再做最后的努力---自旋锁。由于当前线程暂时无法获得锁，但是什么时候可以获得锁是一个未知数。也许在几个CPU时钟周期后，就可以得到锁。如果这样，简单粗暴地挂起线程可能是一种得不偿失的操作。因此，系统会进行一次赌注：它会假设在不久的将来，线程可以得到这把锁。因此，虚拟机会让当前线程做几个空循环(这也是自旋的含义)，在经过若干次循环后，如果可以得到锁，那么就顺利进入临界区。如果还不能获得锁，才会真实地将线程在操作系统层面挂起。

### 4.2.4 锁消除

锁消除是一种更彻底的锁优化。Java虚拟机在JIT编译时，通过对运行上下文的扫描，取出不可能存在共享资源竞争的锁。通过锁消除，可以节省毫无意义的请求锁时间。

分析下面的代码：



注意上述代码中的Vector，由于变量v只在createStrings()函数中使用，因此，它只是一个单纯的局部变量。局部变量是在线程栈上分配的，属于线程私有的数据，因此不可能被其他线程访问。所以，在这种情况下，Vector内部所有加锁同步都是没有必要的。如果虚拟机检测到这种情况，就会将这些无用的锁操作去除。

锁消除涉及的一项关键技术为逃逸分析。所谓逃逸分析就是观察某一个变量是否会逃出某一个作用域。在本例中，变量v显然没有逃出createStrings()函数之外。以此为基础，虚拟机才可以大胆地将v内部的加锁操作去除。如果createStrings()返回的不是String数组，而是v本身，那么久认为变量v逃逸出了当前函数，也就是说v有可能被其他线程访问。如果是这样，虚拟机就不可能消除v中的锁操作。

逃逸分析必须在-server模式下，可以使用-XX:+DoEscapeAnalysis参数打开逃逸分析。使用-XX:+EliminateLocks参数可以打开锁消除。

## 4.3 人手一支笔：ThreadLocal

除了控制资源的访问外，我们还可以通过增加资源来保证所有对象的线程安全。

### 4.3.1 ThreadLocal的简单使用

注意：为每一个线程分配不同的对象，需要在应用层面保证。ThreadLocal只是起到了简单的容器作用。

### 4.3.2 ThreadLocal的实现原理

原理找博客详细解释。

### 4.3.3 对性能有何帮助

为每一个线程分配一个独立的对象对系统性能也许是有帮助的。当然了，这也不一定，这完全取决于共享对象的内部逻辑。如果共享对象对于竞争的处理容易引起性能损失，我们还是应当考虑使用ThreadLocal为每个线程分配单独的对象。

## 4.4 无锁

无锁的策略使用一种叫做比较交换的技术(CAS Compare And Swap)来鉴别线程冲突，一旦检查到冲突产生，就重试当前操作直到没有冲突为止。

### 4.4.1 与众不同的并发策略：比较并交换(CAS)

与锁相比，使用比较交换(下文简称CAS)会使程序看起来更加复杂一些。但由于其阻塞性，它对死锁问题天生免疫，并且，线程间的相互影响也远远比基于锁的方式更小。更为重要的是，使用无锁的方式完全没有锁竞争带来的系统开销，也没有线程间频繁调度带来的开销，因此，它要比基于锁的方式拥有更优越的性能。

CAS算法的过程是这样：它包含三个参数CAS(V, E, N)。V表示要更新的变量，E表示预期值，N表示新值。仅当V值等于E值时，才会将V的值设置为N，如果V值和E值不同，则说明已经有其他线程做了更新，则当前线程什么都不做。最后，CAS返回当前V的真实值。CAS操作是抱着乐观的态度进行的，它总是认为自己可以成功完成操作。当多个线程同时使用CAS操作一个变量时，只有一个会胜出，并成功更新，其余均会失败。失败的线程不会被挂起，仅是被告知失败，并且允许再次尝试，当然也允许失败的线程放弃操作。基于这样的原理，CAS操作即便没有锁，也可以发现其他线程对当前线程的干扰，并进行恰当的处理。

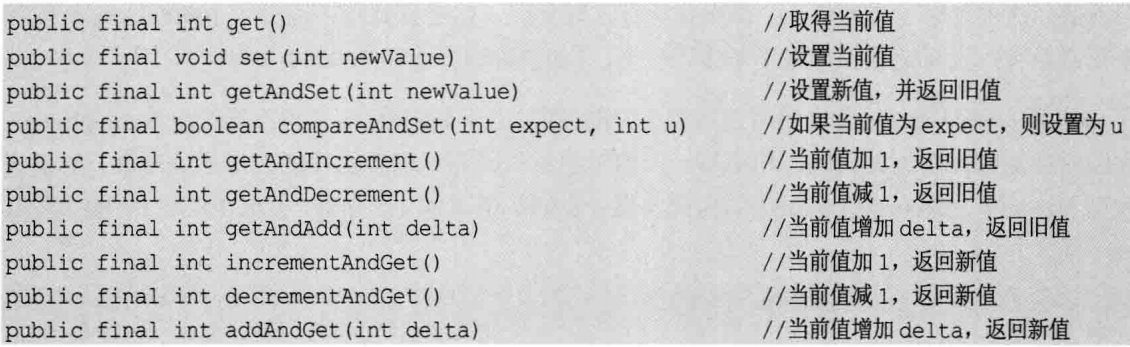
简单地说，CAS需要你额外给出一个期望值，也就是你认为这个变量现在应该是什么样子的。如果变量不是你想象的那样，那说明它已经被别人修改过了。你就重新读取，再次尝试修改就好了。

在硬件层面，大部分的现代处理器都已经支持原子化的CAS指令。在JDK5.0以后，虚拟机便可以使用这个指令来实现并发操作和并发数据结构，并且，这种操作在虚拟机中可以说是无处不在。

### 4.4.2 无锁线程安全整数

为了让Java程序员能够受益于CAS等CPU指令，JDK并发包中有一个atomic包，里面实现了一些直接使用CAS操作的线程安全的类型。

其中，最常用的一个类，应该就是AtomicInteger。你可以把它看做是一个整数。但是与Integer不同，它是可变的，并且线程安全的。对其进行修改等任何操作，都是用CAS指令进行的。这里简单列举一下AtomicInteger的一些主要方法，对于其他原子类，操作也是非常类似的：



就内部实现上来说，AtomicInteger中保存一个核心字段：

private volatile int value;

它就代表了AtomicInteger的当前实际取值。此外还有一个：

private static final long valueOffset;

它保存着value字段在AtomicInteger对象中的偏移量。后面你会看到，这个偏移量是实现AtomicInteger的关键。

### 4.4.3 Java中的指针：Unsafe类

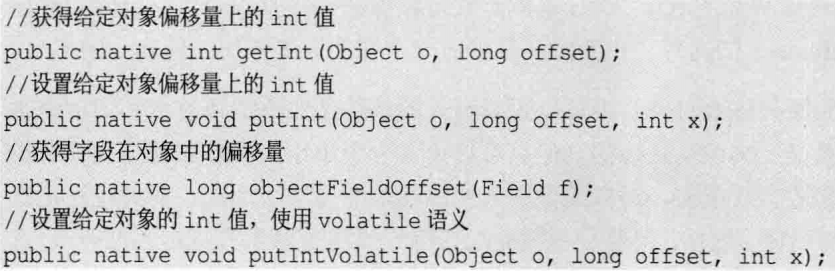
AtomicInteger中的incrementAndGet()方法是使用sun.misc.Unsafe类型中的compareAndSwapInt()实现的。

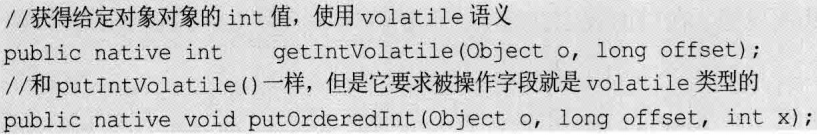
compareAndSwapInt()方法是一个native方法，它的几个参数含义如下：

public final native boolean compareAndSwapInt(Object o, long offset, int expected, int x);

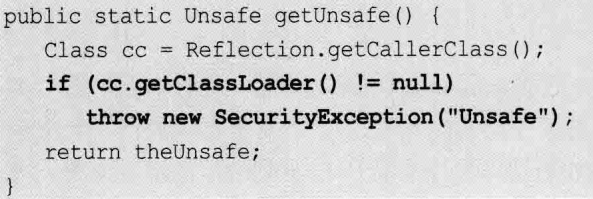
第一个参数o为给定的对象，offset为对象内的偏移量(其实就是一个字段到对象头部的偏移量，通过这个字段可以快速定位字段)，expected表示期望值，x表示要设置的值。如果指定的字段的值等于expected，那么就会把它设置为x。

不难看出，compareAndSwapInt()方法的内部，必然是使用CAS原子指令来完成的。此外，Unsafe类还提供了一些方法，主要有以下几个(以Int操作为例，其他数据类型是相似的)：





获得Unsafe实例的方法是调用其工厂方法getUnsafe()。但是，它的实现却是这样：



注意加粗部分的代码，它会检查调用getUnsafe()函数的类，如果这个类的ClassLoader不为null，就直接抛出异常，拒绝工作。因此，这也使得我们自己的应用程序无法直接使用Unsafe类。它是一个JDK内部使用的专属类。

注意：根据Java类加载器的工作原理，应用程序的类由App Loader加载。而系统核心类，如rt.jar中的类由Bootstrap类加载器加载。Bootstrap加载器是没有Java对象的对象，因此试图获得这个类加载器会返回null。所以，当一个类的类加载器为null时，说明它是由Bootstrap加载的，而这个类也极有可能是rt.jar中的类。

### 4.4.4 无锁的对象引用：AtomicReference

AtomicReference和AtomicInteger非常类似，不同之处就在于AtomicInteger是对整数的封装，而AtomicReference则对应普通的对象引用。也就是它可以保证你在修改对象引用时的线程安全性。

原子操作逻辑上的不足：ABA问题

### 4.4.5 带有时间戳的对象引用：AtomicStampedReference

AtomicReference无法解决上述问题的根本原因是对象在修改过程中，丢失了状态信息。对象值本身与状态被画上了等号。

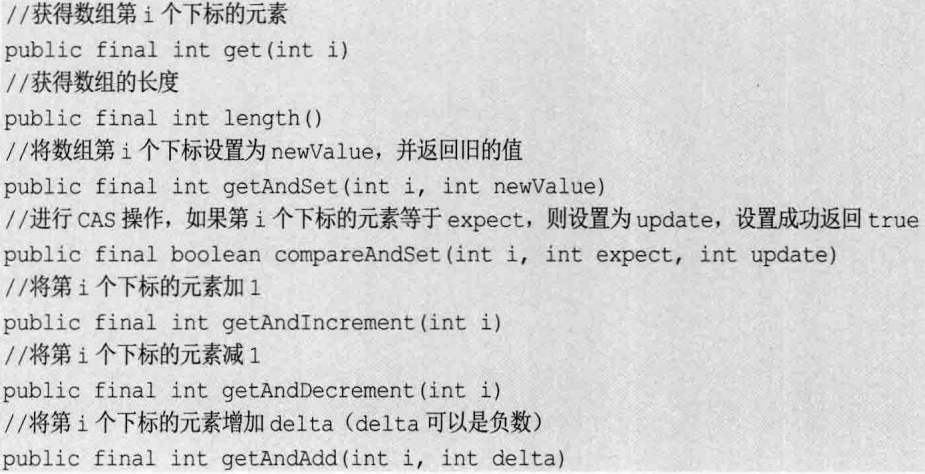
AtomicStampedReference解决了上述问题，它内部不仅维护了对象值，还维护了一个时间戳(我这里把它称为时间戳，实际上它可以使用任何一个整数来表示状态值)。当AtomicStampedReference对应的数值被修改时，除了更新数据本身外，还必须要更新时间戳。当AtomicStampedReference设置对象值时，对象值以及时间戳都必须满足期望值，写入才会成功。因此，即使对象值被反复读写，写回原值，只要时间戳发生变化，就能防止不恰当的写入。

### 4.4.6 数组也能无锁：AtomicIntegerArray

除了提供基本数据类型外，JDK还为我们准备了数组等复合结构。当前可用的原子数组有：AtomicIntegerArray、AtomicLongArray和AtomicReferenceArray，分别表示整数数组、long型数组和普通的对象数组。

这里以AtomicIntegerArray为例，展示原子数组的使用方式：

AtomicIntegerArray本质上是对int[]类型的封装，使用Unsafe类通过CAS的方式控制int[]在多线程下的安全性。它提供了以下几个核心API：



### 4.4.7 让普通变量也享受原子操作：AtomicIntegerFieldUpdater

AtomicIntegerFieldUpdater可以让你在不改动(或者极少改动)原有代码的基础上，让普通的变量也享受CAS操作带来的线程安全性，这样你可以修改极少的代码，来获得线程安全的保证。

根据数据类型不同，这个Updater有三种，分别是AtomicIntegerFieldUpdater、AtomicLongFieldUpdater和AtomicReferenceFieldUpdater。顾名思义，它们分别可以对int、long和普通对象进行CAS修改。

使用AtomicIntegerFieldUpdater的几个注意事项：

第一，Updater只能修改它可见范围内的变量。因为Updater使用反射得到这个变量。如果变量不可见，就会出错。比如如果score声明为private，就是不可行的。

第二，为了确保变量被正确的读取，它必须是volatile类型的。如果我们原有代码中未声明这个类型，那么简单地声明一下就行，这不会引起什么问题。

第三，由于CAS操作会通过对象实例中的偏移量直接进行赋值，因此，它不支持static字段(Unsafe.objectFieldOffset()不支持静态变量)。

### 4.4.8 挑战无锁算法：无锁的Vector实现

相对于有锁的方法，使用无锁的方式编程更加考验一个程序员的耐心和智力。但是，无锁带来的好处也是显而易见的，第一，在高并发的情况下，它比有锁的程序拥有更好的性能；第二，它天生就是死锁免疫的。

### 4.4.9 让线程之间互相帮助：细看SynchronousQueue的实现

SynchronousQueue

源码分析

## 4.5 有关死锁的问题

什么是死锁？通俗的说，死锁就是两个或者多个线程，相互占用对方需要的资源，而都不进行释放，导致彼此之间都相互等待对方释放资源，产生了无限制等待的现象。死锁一旦发生，如果没有外力介入，这种等待将永远存在，从而对程序产生严重的影响。

# 第5章：并行模式与算法

## 5.1 探讨单例模式

在Java中，单例模式能带来两大好处：

1)对于频繁使用的对象，可以省略new操作花费的时间，这对于那些重量级对象而言，是非常可观的一笔系统开销；

2)由于new操作的次数减少，因而对系统内存的使用频率也会降低，这将减轻GC压力，缩短GC停顿时间。

对于静态成员instance，它会在类第一次初始化的时候被创建。这个时刻并不一定是getInstance()方法第一次被调用的时候。

各种单例模式的写法。

## 5.2 不变模式

为了尽可能地去除同步操作，提高并行程序性能，可以使用一种不可改变的对象，依靠对象的不变性，可以确保其在没有同步操作的多线程环境中依然始终保持内部状态的一致性和正确性。这就是不变模式。

不变模式天生就是多线程友好的，它的核心思想是，一个对象一旦被创建，则它的内部状态将永远不会发生改变。所以，没有一个线程可以修改其内部状态和数据，同时其内部状态也绝不会自行发生改变。基于这些特性，对不变对象的多线程操作不需要进行同步控制。

同时还需要注意，不变模式和只读属性是有一定的区别的。不变模式是比只读属性具有更强的一致性和不变性。对只读属性的对象而言，对象本身不能被其他线程修改，但是对象的自身状态却可能自行修改。

不变模式的主要使用场景需要满足以下2个条件：

1)当对象创建后，其内部状态和数据不再发生任何变化；

2)对象需要被共享，被多线程频繁访问。

在Java语言中，不变模式的实现很简单。为确保对象被创建后，不发生任何改变，并保证不变模式正常工作，只需要注意一下4点：

1)去除setter方法以及所有修改自身属性的方法；

2)将所有属性设置为私有，并用final标记，确保其不可修改；

3)确保没有子类可以重载修改它的行为；

4)有一个可以创建完整对象的构造函数。

在不变模式的实现中，final关键字起到了重要的作用。对属性的final定义确保所有数据只能在对象被构造时赋值1次。之后，就永远不再发生改变。而对class的final确保了类不会有子类。根据里氏替换原则，子类可以完全的替代父类。如果父类是不变的，那么子类也必须是不可变的，但实际上我们并无法约束这点。为了防止子类作出一些意外的行为，这里就干脆把子类都禁用了。

在JDK中，不变模式的应用非常广泛。其中，最为典型的就是java.lang.String类。此外，所有的元数据类包装类，都是使用不变模式实现的。主要的不变模式类型如下：

java.lang.String

java.lang.Boolean

java.lang.Byte

java.lang.Character

java.lang.Double

java.lang.Float

java.lang.Integer

java.lang.Long

java.lang.Short

由于基本数据类型和String类型在实际的软件开发中应用极其广泛，使用不变模式后，所有实例的方法均不需要进行同步操作，保证了它们在多线程环境下的性能。

注意：不变模式通过回避问题而不是解决问题的态度来处理多线程并发访问控制。不变对象是不需要进行同步操作的。由于并发同步会对性能产生不良的影响，因此，在需求允许的情况下，不变模式可以提高系统的并发性和并发量。

## 5.3 生产者-消费者模式

在生产者-消费者模式中，通常有两类线程，即若干个生产者线程和若干个消费者线程。生产者线程负责提交用户请求，消费者线程则负责具体处理生产者提交的任务。生产者和消费者之间则通过共享内存缓冲区进行通信。

注意：生产者-消费者模式中的内存缓存区的主要功能是数据在多线程间的共享，此外，通过该缓冲区，可以缓解生产者和消费者间的性能差。

生产者-消费者模式的核心组件是共享内存缓存区，它作为生产者和消费者的通信桥梁，避免了生产者和消费者的直接通信，从而将生产者和消费者进行解耦。生产者不需要知道消费者的存在，消费者也不需要知道生产者的存在。

同时，由于内存缓冲区的存在，允许生产者和消费者在执行速度上存在时间差，无论是生产者在某一局部时间内速度高于消费者，还是消费者在局部时间内高于生产者，都可以通过共享内存缓冲区得到缓解，确保系统正常运行。

## 5.4 高性能的生产者-消费者：无锁的实现

BlockingQueue用于实现生产者和消费者是一个不错的选择。它可以很自然地实现作为生产者和消费者的内存缓冲区。但是BlockingQueue并不是一个高性能的实现，它完全使用锁和阻塞等待来实现线程间的同步。在高并发场合，它的性能并不是特别的优越。就像之前我已经提过的：ConcurrentLinkedQueue是一个高性能的队列，但是BlockingQueue只是为了方便数据共享。

而ConcurrentLinkedQueue的秘诀就在于大量使用了无锁的CAS操作。同理，如果我们使用CAS来实现生产者-消费者模式，也同样可以获得可观的性能提升。不过正如大家所见，使用CAS进行编程是非常困难的，但是有一个好消息是，目前有一个现成的Disruptor框架，它已经帮助我们实现了这一功能。

### 5.4.1 无锁的缓存框架：Disruptor

### 5.4.2 用Disruptor实现生产者-消费者案例

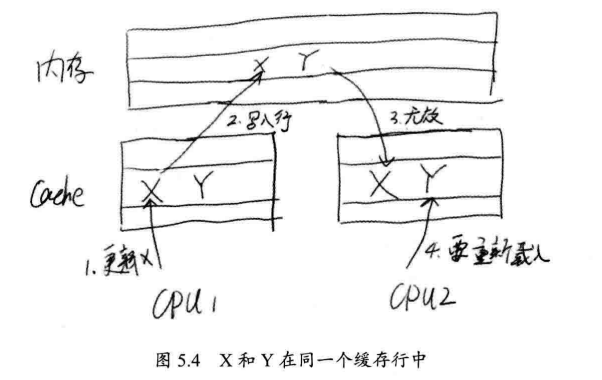
### 5.4.3 提高消费者的响应时间：选择合适的策略

### 5.4.4 CPU Cache的优化：解决伪共享问题

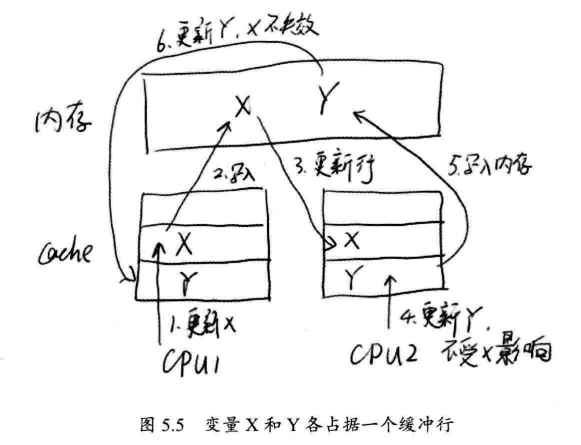
除了使用CAS和提供了各种不同的等待策略来提高系统的吞吐量外。Disruptor大有将优化进行到底的气势，它甚至尝试解决CPU缓存的伪共享问题。

什么是伪共享问题呢？我们知道，为了提高CPU的速度，CPU有一个高速缓存Cache。在高速缓存中，读写数据的最小单位为缓存行(Cache Line)，它是从主存(memory)复制到缓存(Cache)的最小单位，一般为32字节到128字节。

如果两个变量存放在一个缓存行中时，在多线程访问中，可能会相互影响彼此的性能。如下图所示，假设X和Y在同一个缓存行。运行在CPU1上的线程更新了X，那么CPU2上的缓存行就会失效，同一行的Y即使没有修改也会变成无效，导致Cache无法命中。接着，如果在CPU2上的线程更新了Y，则导致CPU1上的缓存行又失效(此时，同一行的X又变得无法访问)。这种情况反反复复发生，无疑是一个潜在的性能杀手。如果CPU经常不能命中缓存，那么系统的吞吐量就会急剧下降。



为了使这种情况不发生，一种可行的做法就是在X变量的前后空间都先占据一定的位置(把它叫做padding吧，用来填充用的)。这样，当内存被读入缓存时，这个缓存行中，只有X一个变量实际是有效的，因此就不会发生多个线程同时修改缓存行中不同变量而导致变量全体失效的情况，如下图所示。



## 5.5 Future模式

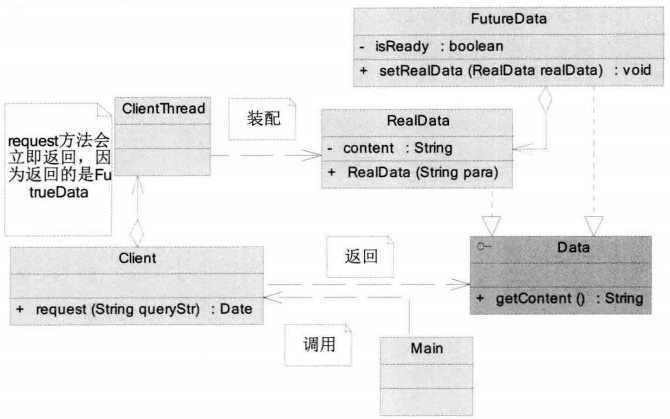
Future模式是多线程开发中非常常见的一种设计模式，它的核心思想是异步调用。当我们需要调用一个函数方法时，如果这个函数执行很慢，那么我们就要进行等待。但有时候，我们可能并不急着要结果。因此，我们可以让被调用者立即返回，让它在后台慢慢处理这个请求。对于调用者来说，则可以先处理一些其他任务，在真正需要数据的场合再去尝试获得需要的数据。

对于Future模式来说，虽然它无法立即给出你需要的数据。但是，它会返回给你一个契约，将来，你可以凭借着这个契约去重新获取你需要的信息。

### 5.5.1 Future模式的主要角色



Future模式的主要参与者



Future模式结构图

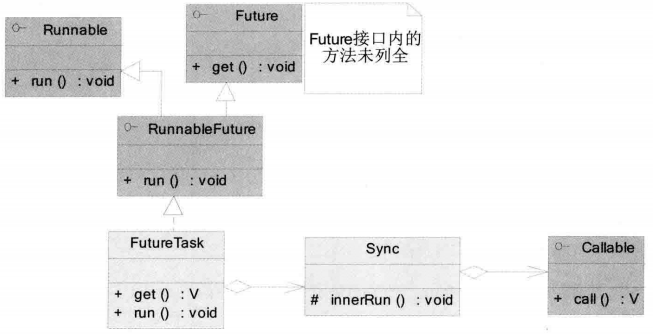
### 5.5.2 Future模式的简单实现

在这个实现中，有一个核心接口Data，这就是客户希望获取的数据。在Future模式中，这个Data接口有两个重要的实现，分别是RealData，也就是真实数据，这就是我们最终需要获得的，有价值的信息。另外一个就是FutureData，它就是用来提取RealData的一个“订单”。因此FutureData是可以立即返回得到的。

### 5.5.3 JDK中的Future模式

Future模式是如此常用，因此JDK内部已经为我们准备好了一套完整的实现。

RunnableFuture继承了Future和Runnable两个接口，其中run()方法用于构造真实的数据。它有一个具体的实现FutureTask类。FutureTask有一个内部类Sync，一些实质性的工作，会委托Sync类实现。而Sync类最终会调用Callable接口，完成实际数据的组装工作。



Callable接口只有一个方法call()，它会返回需要构造的实际数据。这个Callable接口也是这个Future框架和应用程序之间的重要接口。如果我们要实现自己的业务系统，通常需要实现自己的Callable对象。此外，FutureTask类也与应用密切相关，通常，我们会使用Callable示例构造一个FutureTask示例，并将它提交给线程池。

## 5.6 并行流水线

并发算法虽然可以充分发挥多核CPU的性能。但不幸的是，并非所有的计算都可以改造成并发的形式。那什么样的算法是无法使用并发进行计算的呢？简单来说，执行过程中有数据相关性的运算都是无法完美并行化的。

## 5.7 并行搜索

搜索是几乎每一个软件都必不可少的功能。对于有序数据，通常可以采用二分查找法。对于无序数据，则只能挨个查找。

如果使用并行方式，则需要额外增加一些线程间的通信机制，使各个线程可以有效地运行。

一种简单的策略就是将原始数据集合按照期望的线程数进行分割。如果我们计划使用两个线程进行搜索，那么就可以把一个数组或集合分割成两个。每个线程各自独立搜索，当其中有一个线程找到数据后，立即返回结果即可。

## 5.8 并行排序

对于大部分排序算法来说，都是串行执行的。当排序元素很多时，若使用并行算法代替串行算法，显然可以更有效地利用CPU。

### 5.8.1 分离数据相关性：奇偶交换排序

### 5.8.2 改进的插入排序：希尔排序

插入排序也是一种很常见的排序算法。它的基本思想是：一个未排序的数组(当然可以是链表)可以分为两个部分，前半部分是已经排序的，后半部分是未排序的。在进行排序时，只需要在未排序的部分中选择一个元素，将其插入前面有序的数组中即可。最终，未排序的部分会越来越少，直到为0，那么排序就完成了。初始时，可以假设已排序部分就是第一个元素。

简单的插入排序时很难并行化的。因为这一次的数据插入依赖于上一次的得到的有序序列，所以多个步骤之间无法并行。谓词，我们可以对插入排序进行扩展，这就是希尔排序。

希尔排序的一个主要优点是，即使一个较小的元素在数组的末尾，由于每次元素移动都以h为间隔进行，因此数组末尾的小元素可以在很少的交换次数下，就被置换到最接近元素最终位置的地方。

## 5.9 并行算法：矩阵乘法

在矩阵乘法中，第一个矩阵的列数和第二个矩阵的行数必须是相同的。

## 5.10 准备好了再通知我：网络NIO

博文视点社区

Java NIO中涉及的基础内容有通道(Channel)、缓冲区(Buffer)、

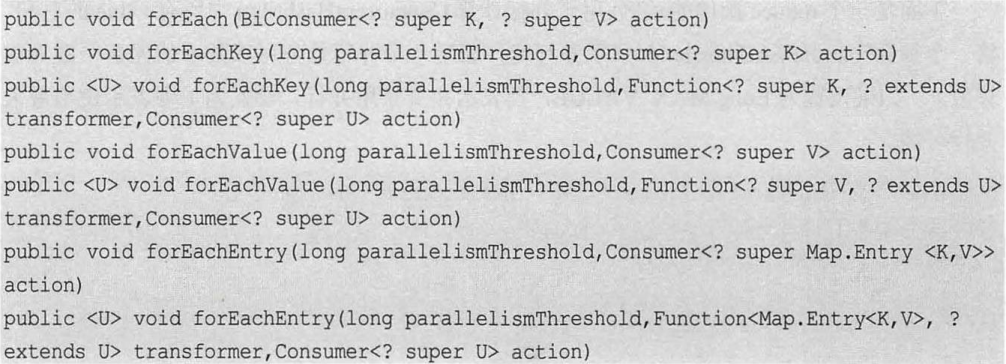
### 6.7.2 LongAdder功能的增强版：LongAccumulator

LongAccumulator是LongAdder的“亲兄弟”，它们有公共的父类Striped64。因此，LongAccumulator内部的优化方式和LongAdder的是一样的。它们都是将一个long型整数分割，并存储在不同的变量中，以防止多线程竞争。两者的主要逻辑是类似的，但是LongAccumulator是LongAdder的功能扩展。对于LongAdder来说，它只是每次对给定的整数执行一次加法，而LongAccumulator则可以实现任意函数操作。

## 6.8 ConcurrentHashMap增强

### 6.8.1 forearch操作

新版本的ConcurrentHashMap增加了一些foreach操作，如下所示。



这些foreach操作的接口是一个Consumer或者BiConcumer，用于对Map的数据进行消费。

### 6.8.2 reduce操作

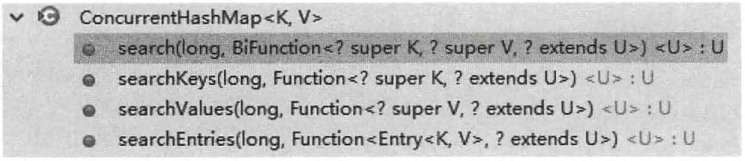
和foreach操作类似，reduce操作对Map的数据进行处理的同时会将其转为另一种形式。可以认为这是foreach操作的Function版本。

### 6.8.3 条件插入

在应用开发中，一个十分常见的场景是条件插入，即当元素不存在时需要创建并且将对象插入Map中，而当Map中已经存在该元素时，则直接获得当前在Map中的元素，从而避免多次创建。这样可以起到对象复用的功能，对于大型重量级对象有很好的优化效果。

### 6.8.4 search操作

基于ConcurrentHashMap还可以做并发搜索，下图中有几个搜索函数。



### 6.8.5 其他新方法

1. mappingCount()方法

返回Map中的条目总数。有别于size()方法，该方法返回的是long型数据。因此，当元素超过整数最大值时，应该使用这个方法。同时，该方法并不返回精确值，如果在执行该方法时，同时存在并发的插入或者删除操作，则结果是不准确的。

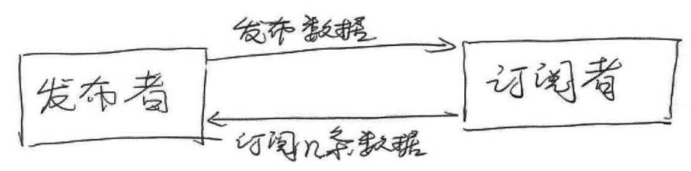
2. newKeySet()方法

在JDK中，Set的实现依附于Map，实际上，Set是Map的一种特殊情况。如果需要一个线程安全的高效并发HashSet，那么基于ConcurrentHashMap的实现是最好的选择。该方法是一个静态工厂方法，返回一个线程安全的Set。

## 6.9 发布和订阅模式

在JDK 9中，引入了一种新的并发编程架构----反应式编程。那么什么是反应式编程呢？反应式编程用于处理异步流中的数据。每当应用收到数据项，便会对它进行处理。反应式编程以流的形式处理数据，因此其内存使用效率会更高。

在反应式编程中，核心的两个组件是Publisher和Subscriber。Publisher将数据发布到流中，Subscriber则负责处理这些数据，如图6.8所示。



### 6.9.1 简单的发布订阅例子

Todo

# 第7章：使用Akka构建高并发程序

Akka是用Scala创建的。

使用Akka能够给我们带来什么好处呢？

首先，Akka提供了一种名为Actor的并发模型，其粒度比线程小，这意味着你可以在系统中启用大量的Actor。

其次，Akka中提供了一套容错机制，允许在Actor出现异常时，进行一些恢复或者重置操作。

再次，通过Akka不仅可以在单机上构建高并发程序，也可以在网络中构建分布式程序，并提供位置透明的Actor定位服务。

## 7.1 新并发模型：Actor

对于并发程序来说，线程始终是并发程序的基本执行单元。但在Akka中，你可以完全忘记线程了。当你使用Akka时，你就有一个全新的执行单元----Actor。Actor是什么呢？

传统Java并行程序是完全基于面向对象的方法。我们还是通过对象的方法调用进行信息的传递。这时，如果对象的方法会修改对象本身的状态，那么在多线程情况下，就有可能出现对象状态的不一致，所以我们必须对这类方法调用进行同步。当然，同步往往是以牺牲性能为代价的。

在Actor模型中，我们失去了对象的方法调用，我们并不是通过调用Actor对象的某一个方法来告诉Actor你需要做什么，而是**给Actor发送一条消息。当一个Actor收到消息后，它有可能会根据消息的内容做出某些行为，包括更改自身状态。**但是，在这种情况下，这个状态的更改是Actor自己进行的，并不是由外界被迫进行的。

## 7.2 Akka之Hello World

当使用Actor进行并行程序开发时，我们的关注点已经不在线程上了。实际上，线程的调度已经被Akka框架封装了，我们只需要关注Actor对象即可。而Actor对象之间的交流和普通的对象的函数调用有明显区别，它们是通过显式的消息发送来传递信息的。

当系统内有多个Actor存在时，Akka会自动在线程池中选择线程来执行我们的Actor。因此，多个不同的Actor有可能会被同一个线程执行，同时，一个Actor也有可能被不同线程执行。因此，一个值得注意的地方是：不要在一个Actor中执行耗时的代码，这样可能会导致其他Actor的调度出现问题。

## 7.3 有关消息投递的一些说明

整个Akka应用是由消息驱动的，消息是除Actor之外最重要的核心组件。作为并发程序中的核心组件，在Actor之间传递的消息应该满足不可变性，也就是不变模式。因为可变的消息无法高效的在并发环境中使用。理论上Akka中的消息可以使用任何对象实例，但在实际使用中，强烈推荐使用不可变的对象。

实际上，对于消息投递，我们可以有三种不同的策略。

第一种，称为至多一次投递。在这种策略中，每一条消息最多会被投递一次。在这种情况下，可能偶尔会出现消息投递失败，而导致消息丢失。

第二种，称为至少一次投递。在这种策略中，每一条消息至少会被投递一次，直到成功为止。因此在一些偶然的场合，接受者可能会收到重复的消息，但不会发生消息丢失。

第三种，称为精确的消息投递。也就是所有的消息保证被精确地投递并成功接收一次。既不会有丢失，也不会有重复接收。

很明显，第一种策略是最高性能、最低成本的。因为系统只要负责把消息送出去就可以了，不需要关注是否成功。第二种策略则需要保存消息投递的状态并不断充实。第三种策略则是成本最高且最不容易实现的。

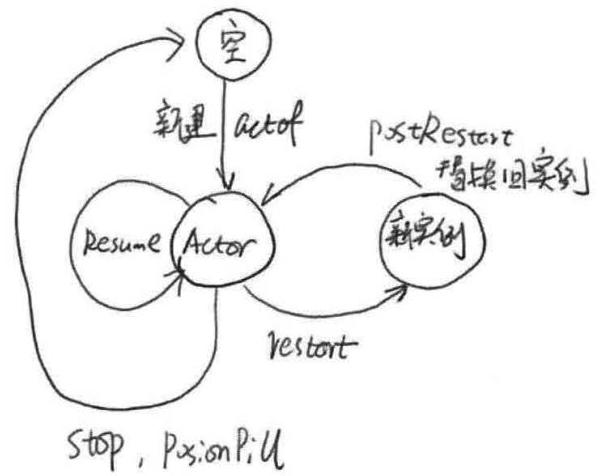
那我们是否真的需要保证消息投递的可靠性呢？

答案是否定的。实际上，我们没有必要在Akka层保证消息的可靠性。这样做成本太高了，也是没有必要的。消息的可靠性更应该从应用的业务层去维护，因为也许在有些时候，丢失一些消息完全是符合应用要求的。因此，在使用Akka时，需要在业务层对此进行保证。

值得注意的一点是，这种消息投递规则不具备可传递性。

## 7.4 Actor的生命周期

Actor在系统中产生后，也存在着“生老病死”的活动周期。Akka框架提供了若干回调函数，让我们得以在Actor的活动周期内进行一些业务相关的行为。Actor的生命周期如下所示：



## 7.5 监督策略

在Akka框架内，父Actor可以对子Actor进行监督，监控Actor的行为是否有异常。监督策略可以分为两种：一种是OneForOneStrategy策略的监督，另外一种是AllForOneStrategy策略的监督。

对于OneForOneStrategy策略，父Actor只会对出问题的子Actor进行处理，比如重启或者停止，而对于AllForOneStrategy策略，父Actor会对出问题的子Actor及它所有的兄弟都进行处理。很显然，对于AllForOneStrategy策略，它更加适合各个Actor联系非常紧密的场景，多个Actor间只要有一个Actor出现故障，就宣告整个任务失败。否则，在更多的场景中，应该使用OneForOneStrategy策略。当然了，OneForOneStrategy策略也是Akka的默认策略。

在一个指定的策略中，我们可以对Actor的失败情况进行相应的处理，比如当失败时，我们可以无视这个错误，继续执行Actor，就像什么事都没发生过一样。或者可以重启Actor，甚至可以让Actor彻底停止工作。要指定这些监督行为，只要构造一个自定义的监督策略即可。

## 7.6 选择Actor

在一个ActorSystem中，可能存在大量的Actor。如何才能有效地对大量Actor进行批量的管理和通信呢？Akka为我们提供了一个ActorSelection类，用来进行批量消息发送。

## 7.7 消息收件箱(Inbox)

所有Actor之间的通信都是通过消息来进行的。这是否意味着我们必须构建一个Actor来控制整个系统呢？答案是否定的，我们并不一定要这么做，Akka框架已经为我们准备了一个叫做“收件箱”的组件，使用它可以很方便地对Actor进行消息发送和接收，大大方便了应用程序与Actor之间的交互。

## 7.8 消息路由

Akka提供了非常灵活的消息发送机制。有时候，我们也许会使用一组Actor而不是一个Actor来提供一项服务。这一组Actor中所有的Actor都是对等的，也就是说你可以找任何一个Actor来为你服务。这种情况下，如何才能快速有效地找到合适的Actor呢？或者说如何调度这些消息，才可以使负载更为均衡地分配在这一组Actor中。

为了解决这个问题，Akka使用一个路由器组件(Router)来封装消息的调度。系统提供了几种实用的消息路由策略，比如，轮询选择Actor进行消息发送，随机消息发送，将消息发送给最空闲的Actor，甚至是在组内广播消息。

## 7.9 Actor的内置状态转换

一个Actor内部消息处理函数可以拥有多个不同的状态，在特定的状态下，可以对同一消息进行不同的处理，状态之间也可以任意切换。

## 7.10 询问模式：Actor中的Future

由于Actor之间都是通过异步消息通信的。当你发送一条消息给一个Actor后，你通常只能等待Actor的返回。与同步方法不同，在你发送异步消息后，接受消息的Actor可能根本来不及处理你的消息，调用方就已经返回了。

这种模式与我们之前提到的Future模式非常相像。不同之处只是在传统的异步调用中，我们进行的是函数调用，但在这里，我们发送了一条消息。

由于两者的行为方式如此相像，因此我们就会很自然地想到，当我们需要一个有返回值的调用时，Actor是不是也应该给我们一个契约(Future)呢？这样，就算我们当下没有办法立即获得Actor的处理结果，在将来，通过这个契约还是可以追踪到我们的请求的。

## 7.11 多个Actor同时修改数据：Agent

在Actor的编程模型中，Actor之间主要通过消息进行信息传递。因此，很少发生多个Actor需要访问同一个共享变量的情况。但在实际开发中，这种情况很难完全避免。如果多个Actor需要对同一个共享变量进行读写时，如何保证线程安全呢？

在Akka中，使用一种叫作Agent的组件来实现这个功能。一个Agent提供了对一个变量的异步更新。当一个Actor希望改变Agent的值时，它会向Agent下发一个动作(action)。当多个Actor同时改变Agent时，这些action将会在ExecutionContext中被并发调度执行。在任意时刻，一个Agent最多只能执行一个action，对于某一个线程来说，它执行action的顺序与它的发生顺序一致，但对于不同线程来说，这些action可能会交织在一起。

Agent的修改可以使用两种方法：send()和alter()。它们都可以向Agent发送一个修改动作。但是send()方法没有返回值，而alter()方法会返回一个Future对象，便于跟踪Agent的执行。

## 7.12 像数据库一样操作内存数据：软件事务内存

在一些函数式编程语言中，支持一种叫作软件事务内存(STM)的技术。什么是软件事务内存呢？这里的事务和数据库中所说的事务非常类似，具有隔离性、原子性和一致性。与数据库事务不同的是，内存事务不具备持久性(很显然内存数据库不会保存下来)。

在很多场合，某一项工作可能要由多个Actor协作完成。在这种协作事务中，如果一个Actor处理失败，那么根据事务的原子性，其他Actor所进行的操作必须要回滚。

## 7.13 一个有趣的例子：并发粒子群的发现

粒子群算法(PSO)是一种进化算法。它与大名鼎鼎的遗传算法非常类似，可以用来解决一些优化问题。大家知道，一些优化问题(比如旅行商问题TSP)都属于NP问题。它们的时间复杂度可能会达到O(n!)或者O(2n)，这种在多项式时间内不可解的问题总是会让人望而生畏。而以**PSO算法**为代表的进化算法，往往可以将这些NP问题转变为一个多项式问题。但这种转变是有代价的，进化算法往往都不保证你可以从结果中得到最优解。

既然最优的方案很难得到，那么我们就想办法以很低的成本获得一个还算过得去的方案，也不失为良策。在很多情况下，虽然进化算法无法让你获得最优解，也无法证明它得到的解与最优解到底有多少差距，但在实际生活中，通过进化算法搜索到的满意解很可能与最优解已经非常接近了。

### 7.13.1 什么是粒子群算法

粒子群算法是一种进化计算技术，最早由Kenny与Eberhart于1995年提出。它源于对鸟群捕食行为的研究，与遗传算法相似，是一种基于迭代的优化算法，广泛应用于函数优化和神经网络训练等方面。与遗传算法相比，PSO的实现简单得多，参数配置也相对较少，对使用人员的经验要求不高，因此更加易于在实际工程中应用。

### 7.13.2 粒子群算法的计算过程

粒子群算法的步骤如下：

(1)初始化所有粒子，粒子的位置随机生成。计算每个粒子当前的适应度，并将此设为当前粒子的个体最优解(记为pBest)。

(2)所有粒子将自己的个体最优值发送给管理者Master。Master获得所有粒子的信息后，筛选出全局最优解(记为gBest)。

(3)Master将gBest通知所有粒子，所有粒子便知道全局最优点的位置。

(4)所有粒子根据自己的个体最优解和全局最优解，更新自己的速度，在有了速度后，再更新自己的位置。

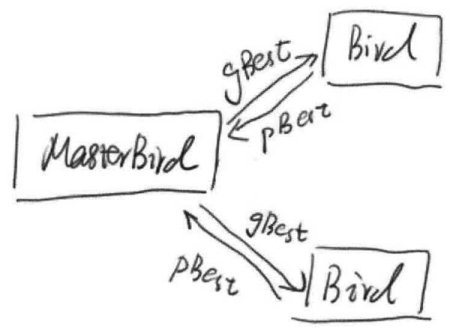
Vk+1 = c0 × rand() × vK + c1 × rand() × (pBestk - xk) + c2 × rand() × (gBestk - xk)

xk+1 = xk + vk+1

其中，rand()函数产生一个(0, 1)之间的随机数。c0 = 1、c1 = 2、c2 = 2，k表示进化的代数。vk表示当前速度，pBestk和gBestk表示个体最优解和全局最优解。当然，对于每一个维度上的速度分量，我们都可以限定一个最大值。确保“小鸟”不会飞得太快，错过了重要的信息。

(5)如果粒子产生了新的个体最优解，则发送给Master，在此转到步骤(2)。

整体过程的示意图如下：



PSO算法示意图

从这个计算步骤中可以看到，计算过程拥有一定的随机性。但由于我们可以启用大量的例子，因此其计算效果在统计学意义上是稳定的。在这个标准的粒子群算法中，由于所有粒子都会向全局最优靠拢，因此其跳出局部最优的能力并不算太强。因此，我们也可以想办法对标准的粒子群算法进行一些优化。比如，允许各个粒子随机移动，甚至逆向移动来试图突破局部最优。为简单起见，我不打算做这些复杂的实现。

### 7.13.3 粒子群算法能做什么

粒子群算法能为我们做什么呢？它应用最多的场景是进行最优化计算。实际上，以粒子群算法为代表的进化计算，可以说是最优化计算中的通用方法。几乎一切最优化问题都可以通过这种随机搜索的模式解决，其成本低、难度小、效果好，因此颇受欢迎。

### 7.13.4 使用Akka实现粒子群

# 第8章：并行程序调试

在本章中，简单介绍一下有关并行程序调试的一些技巧和经验。

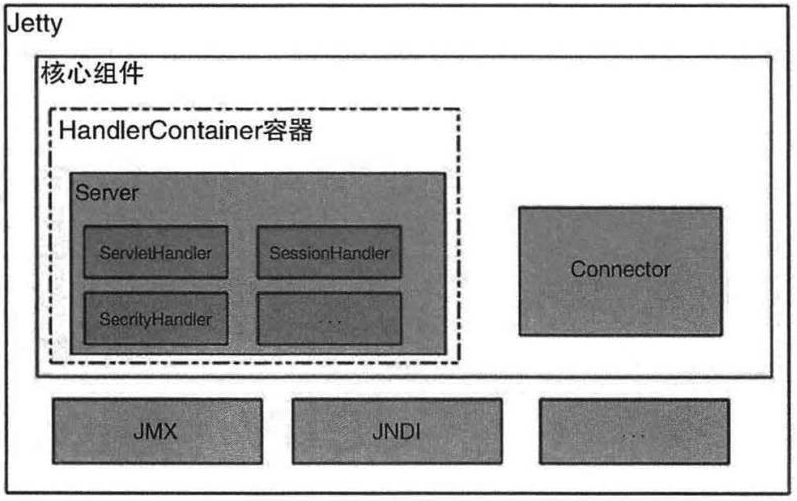
IDEA多线程调试技巧。

# 第9章：多线程优化示例----Jetty核心代码分析

Jetty是一个基于Java实现的、免费的HTTP服务器和Servlet容器。该项目成立于1995年，到目前为止Jetty已经成为与Tomcat齐名的、使用最为广泛的Java Web容器之一。大量成功应用都基于Jetty开发，如Apache Geromino、JBoss、IBM Tivoli、Cisco SESM等。

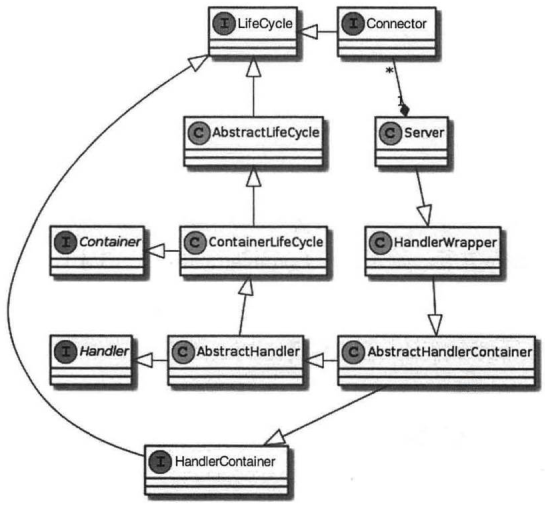
## 9.1 Jetty简介与架构

整个Jetty的核心组件由Server和Connector两个组件构成，整个Server组件是基于Handler容器工作的，它类似于Tomcat的Container容器。Jetty中的Connector组件负责接收客户端的连接请求，并将请求分配给一个处理队列去执行。



Jetty架构图

其中，Server可以说是整个Jetty的核心元素，大量的工作都围绕着Server展开。Server与其他核心组件的关系，如下图：



Server与其他核心组件的关系

Server内维护着一组Connector，每个Connector表示一个可用的服务，每一个客户端连接都是针对一个Connector发起的。Container接口表示可以被JMX管理的对象。LifeCycle接口定义了具有可管理生命周期的对象。Jetty的所有组件的生命周期管理基于观察者模式，在LifeCycle接口中，又定义了LifeCycle.Listener接口作为观察者对象。

## 9.2 Jetty服务器初始化

### 9.2.1 初始化线程池

Jetty服务器使用了独立的线程池处理用户请求，也就是执行“qtp”的那些线程(如果你曾经尝试dump Jetty线程，那么你应该会看到一些qtp开头的线程)。为了满足自己特殊的需求及管理的需要，Jetty并没有直接使用JDK提供的线程池，而是完全独立开发了一套线程池。在初始化线程池时，使用的是QueuedThreadPool。

QueuedThreadPool是一个可以设置最大和最小线程数，以及线程空闲退出时间的线程池。在默认情况下，最大线程数为200，最小线程数为8，线程空闲退出时间为1分钟。也就是说，线程池启动时开启8个线程。在执行过程中，如果线程数不够用，则有机会向上扩展到最多200个线程。当空闲后，如果1个线程1分钟没有任务处理，则结束线程，但始终在线程池中保持8个活动线程。

与JDK自带线程池最大的不同是：QueuedThreadPool没有核心线程数的概念，在不超过最大线程数的前提下，只要没有空闲线程处理新任务，它就会立即开启新的线程。

类似于JDK自带线程池，QueuedThreadPool依然将任务放在BlockingQueue中。但出于优化的目的，Jetty实现了自己的BlockingQueue----BlockingArrayQueue。

BlockingArrayQueue是一个基于数组的队列。它可以是有界的，也可以是无界的，也可以是无界动态扩展的。在QueuedThreadPool中，默认使用无界的BlockingArrayQueue。与LinkedBlockingQueue相比，BlockingArrayQueue采用数组形式，没有next指针，因此会比LinkedBlockingQueue更加节省内存(大量的next指针会占据较多的内存空间)。

BlockingArrayQueue有3个重要的参数：初始化大小、扩展增量和最大容量。当所需容量超过初始化大小时，会以扩展增量来扩大容量。

Object[] elements = new Object[capacity + \_growCapacity];

可以看到，和Vector的扩容机制不同，BlockingArrayQueue的扩容相对保守，并不会成倍扩容，以确保内存空间使用的合理性。