## 基础概念

### CPU核心数和线程数的关系

多核心:也指单芯片多处理器( Chip Multiprocessors,简称CMP),CMP是由美国斯坦福大学提出的,其思想是将大规模并行处理器中的SMP(对称多处理器)集成到同一芯片内,各个处理器并行执行不同的进程。这种依靠多个CPU同时并行地运行程序是实现超高速计算的一个重要方向,称为并行处理

多线程: Simultaneous Multithreading.简称SMT.SMT可通过复制处理器上的结构状态,让同一个处理器上的多个线程同步执行并共享处理器的执行资源可最大限度地实现宽发射、乱序的超标量处理,提高处理器运算部件的利用率,缓和由于数据相关或 Cache未命中带来的访问内存延时。

核心数、线程数:目前主流CPU有双核、三核和四核,六核也在2010年发布。增加核心数目就是为了增加线程数,因为操作系统是通过线程来执行任务的,一般情况下它们是1:1对应关系,也就是说四核CPU一般拥有四个线程。但 Intel引入超线程技术后,使核心数与线程数形成1:2的关系

### CPU时间片轮转机制

我们平时在开发的时候，感觉并没有受cpu核心数的限制，想启动线程就启动线程，哪怕是在单核CPU上，为什么？这是因为操作系统提供了一种CPU时间片轮转机制。

时间片轮转调度是一种最古老、最简单、最公平且使用最广的算法,又称RR调度。每个进程被分配一个时间段,称作它的时间片,即该进程允许运行的时间。

百度百科对CPU时间片轮转机制原理解释如下:

如果在时间片结束时进程还在运行,则CPU将被剥夺并分配给另一个进程。如果进程在时间片结束前阻塞或结来,则CPU当即进行切换。调度程序所要做的就是维护一张就绪进程列表,当进程用完它的时间片后,它被移到队列的末尾

时间片轮转调度中唯一有趣的一点是时间片的长度。从一个进程切换到另一个进程是需要定时间的,包括保存和装入寄存器值及内存映像,更新各种表格和队列等。假如进程切( processwitch),有时称为上下文切换( context switch),需要5ms,再假设时间片设为20ms,则在做完20ms有用的工作之后,CPU将花费5ms来进行进程切换。CPU时间的20%被浪费在了管理开销上了。

为了提高CPU效率,我们可以将时间片设为5000ms。这时浪费的时间只有0.1%。但考虑到在一个分时系统中,如果有10个交互用户几乎同时按下回车键,将发生什么情况?假设所有其他进程都用足它们的时间片的话,最后一个不幸的进程不得不等待5s才获得运行机会。多数用户无法忍受一条简短命令要5才能做出响应,同样的问题在一台支持多道程序的个人计算机上也会发

结论可以归结如下:时间片设得太短会导致过多的进程切换,降低了CPU效率:而设得太长又可能引起对短的交互请求的响应变差。将时间片设为100ms通常是一个比较合理的折衷。

在CPU死机的情况下,其实大家不难发现当运行一个程序的时候把CPU给弄到了100%再不重启电脑的情况下,其实我们还是有机会把它KⅢ掉的,我想也正是因为这种机制的缘故。

### 什么是进程和线程

**进程是程序运行资源分配的最小单位**

进程是操作系统进行资源分配的最小单位,其中资源包括:CPU、内存空间、磁盘10等,同一进程中的多条线程共享该进程中的全部系统资源,而进程和进程之间是相互独立的。进程是具有一定独立功能的程序关于某个数据集合上的一次运行活动,进程是系统进行资源分配和调度的一个独立单位。

进程是程序在计算机上的一次执行活动。当你运行一个程序,你就启动了一个进程。显然,程序是死的、静态的,进程是活的、动态的。进程可以分为系统进程和用户进程。凡是用于完成操作系统的各种功能的进程就是系统进程,它们就是处于运行状态下的操作系统本身,用户进程就是所有由你启动的进程。

**线程是cPU调度的最小单位,必须依赖于进程而存在**

线程是进程的一个实体,是CPU调度和分派的基本单位,它是比进程更小的、能独立运行的基本单位。线程自己基本上不拥有系统资源,只拥有一点在运行中必不可少的资源(如程序计数器,一组寄存器和栈),但是它可与同属一个进程的其他的线程共享进程所拥有的全部资源。

**线程无处不在**

任何一个程序都必须要创建线程,特别是Java不管任何程序都必须启动一个main函数的主线程; Java Web开发里面的定时任务、定时器、JSP和 Servlet、异步消息处理机制,远程访问接口RM等,任何一个监听事件, onclick的触发事件等都离不开线程和并发的知识。

### 澄清并行和并发

我们举个例子,如果有条高速公路A上面并排有8条车道,那么最大的**并行**车辆就是8辆此条高速公路A同时并排行走的车辆小于等于8辆的时候,车辆就可以并行运行。CPU也是这个原理,一个CPU相当于一个高速公路A,核心数或者线程数就相当于并排可以通行的车道;而多个CPU就相当于并排有多条高速公路,而每个高速公路并排有多个车道。

当谈论**并发**的时候一定要加个单位时间,也就是说单位时间内并发量是多少?离开了单位时间其实是没有意义的。

俗话说,一心不能二用,这对计算机也一样,原则上一个CPU只能分配给一个进程,以便运行这个进程。我们通常使用的计算机中只有一个CPU,也就是说只有一颗心,要让它一心多用同时运行多个进程,就必须使用并发技术。实现并发技术相当复杂,最容易理解的是“时间片轮转进程调度算法”。

综合来说：

并发:指应用能够交替执行不同的任务,比如单CPU核心下执行多线程并非是同时执行多个任务,如果你开两个线程执行,就是在你几乎不可能察觉到的速度不断去切换这两个任务,已达到"同时执行效果",其实并不是的,只是计算机的速度太快,我们无法察觉到而已.

并行:指应用能够同时执行不同的任务,例:吃饭的时候可以边吃饭边打电话,这两件事情可以同时执行

两者区别:一个是交替执行,一个是同时执行.

### 高并发编程的意义、好处和注意事项

由于多核多线程的CPU的诞生,多线程、高并发的编程越来越受重视和关注。多线程可以给程序带来如下好处。

(1)充分利用CPU的资源

从上面的CPU的介绍,可以看的出来,现在市面上没有CPU的内核不使用多线程并发机制的,特别是服务器还不止一个CPU,如果还是使用单线程的技术做思路,明显就out了。因为程序的基本调度单元是线程,并且一个线程也只能在一个CPU的一个核的一个线程跑,如果你是个i3的CPU的话,最差也是双核心4线程的运算能力:如果是一个线程的程序的话,那是要浪费3/4的CPU性能:如果设计一个多线程的程序的话,那它就可以同时在多个CPU的多个核的多个线程上跑,可以充分地利用CPU,减少CPU的空闲时间,发挥它的运算能力,提高并发量。

就像我们平时坐地铁一样,很多人坐长线地铁的时候都在认真看书,而不是为了坐地铁而坐地铁,到家了再去看书,这样你的时间就相当于有了两倍。这就是为什么有些人时间很充裕,而有些人老是说没时间的一个原因,工作也是这样,有的时候可以并发地去做几件事情,充分利用我们的时间,CPU也是一样,也要充分利用。

(2)加快响应用户的时间

比如我们经常用的迅雷下载,都喜欢多开几个线程去下载,谁都不愿意用一个线程去下载,为什么呢?答案很简单,就是多个线程下载快啊。

我们在做程序开发的时候更应该如此,特别是我们做互联网项目,网页的响应时间若提升1s,如果流量大的话,就能增加不少转换量。做过高性能web前端调优的都知道,要将静态资源地址用两三个子域名去加载,为什么?因为每多一个子域名,浏览器在加载你的页面的时候就会多开几个线程去加载你的页面资源,提升网站的响应速度。多线程,高并发真的是无处不在。

(3)可以使你的代码模块化,异步化,简单化

例如我们在做 Android程序开发的时候,主线程的UI展示部分是一块主代码程序部分,但是UI上的按钮用相应事件的处理程序就可以做个单独的模块程序拿出来。这样既增加了异步的操,又使程序模块化,清晰化和简单化。

时下最流行的异步程序处理机制,正是多线程、并发程序最好的应用例子。

多线程应用开发的好处还有很多,大家在日后的代码编写过程中可以慢慢体会它的魅力。

多线程程序需要注意事项

(1)线程之间的安全性

从前面的章节中我们都知道,在同一个进程里面的多线程是资源共享的,也就是都可以访问同一个内存地址当中的一个变量。例如:若每个线程中对全局变量、静态变量只有读操作,而无写操作,一般来说,这个全局变量是线程安全的:若有多个线程同时执行写操作,一般都需要考虑线程同步,否则就可能影响线程安全。

(2)线程之间的死循环过程

为了解决线程之间的安全性引入了Java的锁机制,而一不小心就会产生Java线程死锁的多线程问题,因为不同的线程都在等待那些根本不可能被释放的锁,从而导致所有的工作都无法完成。假设有两个线程,分别代表两个饥饿的人,他们必须共享刀叉并轮流吃饭。他们都需要获得两个锁:共享刀和共享叉的锁。

假如线程A获得了刀,而线程B获得了叉。线程A就会进入阻塞状态来等待获得叉,而线程B则阻塞来等待线程A所拥有的刀。这只是人为设计的例子,但尽管在运行时很难探测到,这类情况却时常发生

(3)线程太多了会将服务器资源耗尽形成死机当机

线程数太多有可能造成系统创建大量线程而导致消耗完系统内存以及CPU的“过渡切换”,造成系统的死机,那么我们该如何解决这类问题呢?

某些系统资源是有限的,如文件描述符。多线程程序可能耗尽资源,因为每个线程都可能希望有一个这样的资源。如果线程数相当大,或者某个资源的侯选线程数远远超过了可用的资源数则最好使用资源池。一个最好的示例是数据库连接池。只要线程需要使用一个数据库连接,它就从池中取出一个,使用以后再将它返回池中。资源池也称为资源库。这里先有一个概念,后面会

多线程应用开发的注意事项很多,希望大家在日后的工作中可以慢慢体会它的危险所在。

## 认识Java里的线程

### Java里的程序天生就是多线程的

一个Java程序从main()方法开始执行，然后按照既定的代码逻辑执行，看似没有其他线程参与，但实际上Java程序天生就是多线程程序，因为执行main()方法的是一个名称为main的线程。

[6] Monitor Ctrl-Break //监控Ctrl-Break中断信号的

[5] Attach Listener //内存dump，线程dump，类信息统计，获取系统属性等

[4] Signal Dispatcher // 分发处理发送给JVM信号的线程

[3] Finalizer // 调用对象finalize方法的线程

[2] Reference Handler//清除Reference的线程

[1] main //main线程，用户程序入口

### 线程的启动与中止

#### 启动

启动线程的方式有：

1、X extends Thread;，然后X.run

2、X implements Runnable；然后交给Thread运行

3、X implements Callable；然后交给Thread运行

第1、2方式都有一个缺陷就是：在执行完任务之后无法获取执行结果。从Java 1.5开始，就提供了Callable和Future，通过它们可以在任务执行完毕之后得到任务执行结果。

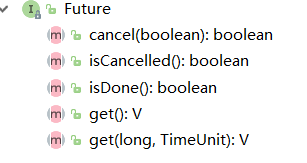
参见代码：cn.enjoyedu.concurrent.NewThread

##### Callable、Future和FutureTask

Runnable是一个接口，在它里面只声明了一个run()方法，由于run()方法返回值为void类型，所以在执行完任务之后无法返回任何结果。

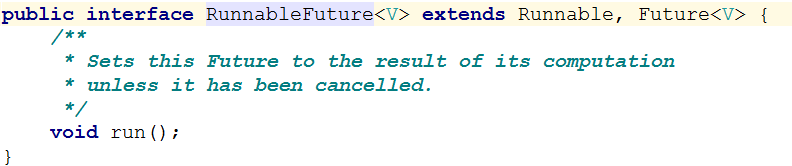
Callable位于java.util.concurrent包下，它也是一个接口，在它里面也只声明了一个方法，只不过这个方法叫做call()，这是一个泛型接口，call()函数返回的类型就是传递进来的V类型。

Future就是对于具体的Runnable或者Callable任务的执行结果进行取消、查询是否完成、获取结果。必要时可以通过get方法获取执行结果，该方法会阻塞直到任务返回结果。



因为Future只是一个接口，所以是无法直接用来创建对象使用的，因此就有了下面的FutureTask。

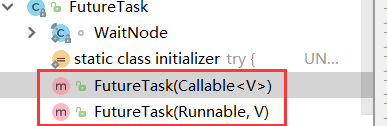




FutureTask类实现了RunnableFuture接口，RunnableFuture继承了Runnable接口和Future接口，而FutureTask实现了RunnableFuture接口。所以它既可以作为Runnable被线程执行，又可以作为Future得到Callable的返回值。

事实上，FutureTask是Future接口的一个唯一实现类。

要new一个FutureTask的实例，有两种方法



#### 中止

线程自然终止：要么是run执行完成了，要么是抛出了一个未处理的异常导致线程提前结束。

手动中止

暂停、恢复和停止操作对应在线程Thread的API就是**suspend()、resume()和stop()**。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。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。

安全的中止则是其他线程通过调用某个线程A的**interrupt()**方法对其进行中断操作, 中断好比其他线程对该线程打了个招呼，“A，你要中断了”，不代表线程A会立即停止自己的工作，同样的A线程完全可以不理会这种中断请求。因为java里的线程是协作式的，不是抢占式的。线程通过检查自身的中断标志位是否被置为true来进行响应，线程通过方法**isInterrupted()**来进行判断是否被中断，也可以调用静态方法**Thread.interrupted()**来进行判断当前线程是否被中断，不过Thread.interrupted()会同时将中断标识位改写为false。

如果一个线程处于了阻塞状态（如线程调用了thread.sleep、thread.join、thread.wait、），则在线程在检查中断标示时如果发现中断标示为true，则会在这些阻塞方法调用处抛出InterruptedException异常，并且在抛出异常后会立即将线程的中断标示位清除，即重新设置为false。

不建议自定义一个取消标志位来中止线程的运行。因为run方法里有阻塞调用时会无法很快检测到取消标志，线程必须从阻塞调用返回后，才会检查这个取消标志。这种情况下，使用中断会更好，因为，一、一般的阻塞方法，如sleep等本身就支持中断的检查，二、检查中断位的状态和检查取消标志位没什么区别，用中断位的状态还可以避免声明取消标志位，减少资源的消耗。

**注意：处于死锁状态的线程无法被中断**

## 对Java里的线程再多一点点认识

### 深入理解run()和start()

Thread类是Java里对线程概念的抽象，可以这样理解：我们通过new Thread()其实只是new出一个Thread的实例，还没有操作系统中真正的线程挂起钩来。只有执行了start()方法后，才实现了真正意义上的启动线程。

start()方法让一个线程进入就绪队列等待分配cpu，分到cpu后才调用实现的run()方法，start()方法不能重复调用。

而run方法是业务逻辑实现的地方，本质上和任意一个类的任意一个成员方法并没有任何区别，可以重复执行，可以被单独调用。

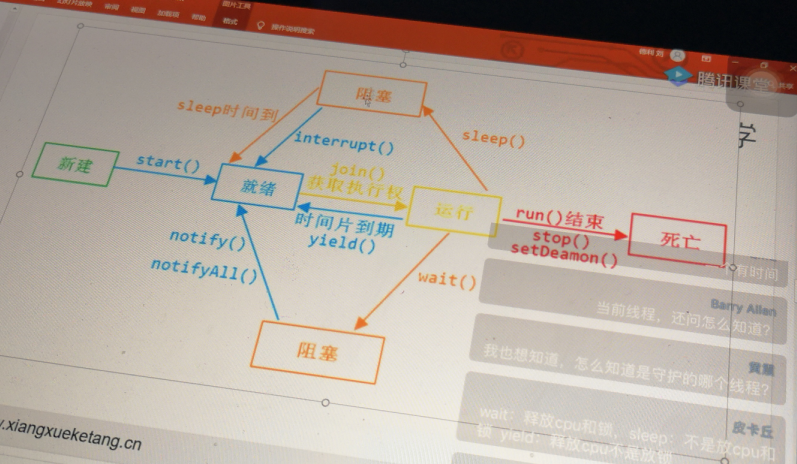
### 其他的线程方法

yield()方法：使当前线程让出CPU占有权，但让出的时间是不可设定的。也不会释放锁资源，所有执行yield()的线程有可能在进入到可执行状态后马上又被执行。

join方法：把指定的线程加入到当前线程，可以将两个交替执行的线程合并为顺序执行的线程。比如在线程B中调用了线程A的Join()方法，直到线程A执行完毕后，才会继续执行线程B。

wait()/notify()/notifyAll()：后面会单独讲述

## 线程的大致的流程



一个线程新建，到start（）-就绪了，如果获取join（）执行权就会运行，时间片到期 yield就会从运行状态转化到就绪

运行的状态，可以通过sleep（） 到阻塞状态，同时run（）到结束，也是就是一个死亡的状态。Stop不建议去使用，因为这样子会导致线程有很多的碎片化的空间，线程中的任务没有去执行完成，也可以通过wait 去到阻塞的情况

线程阻塞可以通过 notify notifyAll去唤醒他，同时sleep时间到也可以到就绪的状态，如果一个线程处于了阻塞状态（如线程调用了thread.sleep、thread.join、thread.wait、），则在线程在检查中断标示时如果发现中断标示为true，则会在这些阻塞方法调用处抛出InterruptedException异常，并且在抛出异常后会立即将线程的中断标示位清除，即重新设置为false，所以要去处理异常的情况才可以

join来控制，让两外一个线程获取到自行权 看Demo