1、线程基础、线程之间的共享和协作

基础概念

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 IO 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

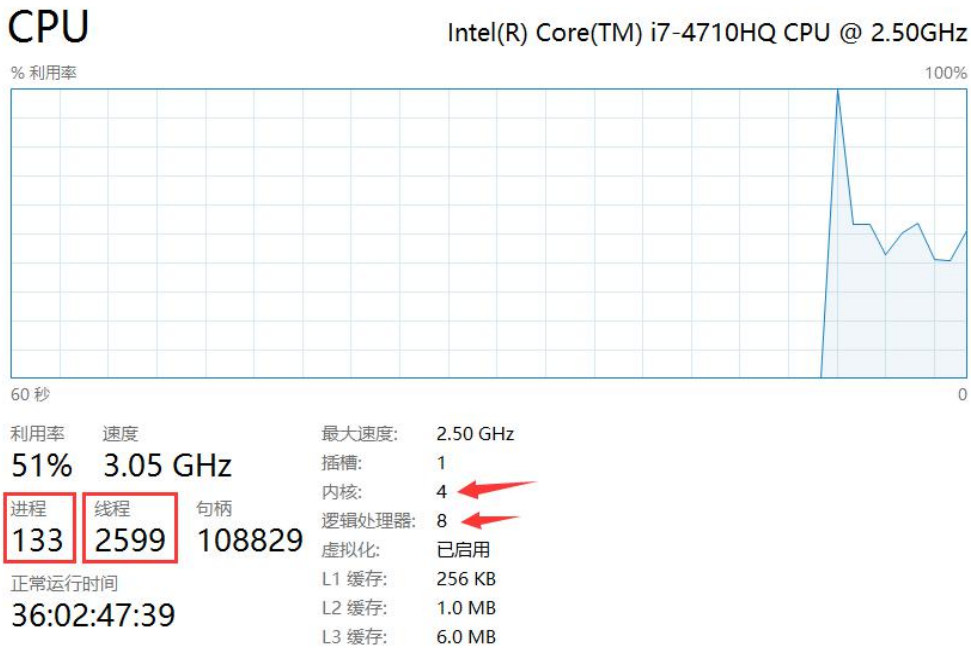
任何一个程序都必须要创建线程,特别是 Java 不管任何程序都必须启动一个 main 函数的主线程; Java Web 开发里面的定时任务、定时器、JSP 和 Servlet、异步消息处理机制,远程访问接口 RM 等,任何一个监听事件, onclick 的触发事件等都离不开线程和并发的知识。

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.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。

核心数、线程数:目前主流 CPU 都是多核的。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。

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。

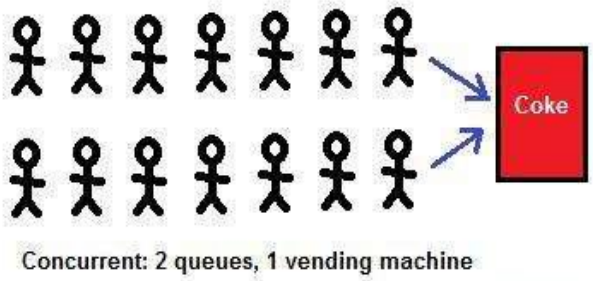
俗话说,一心不能二用,这对计算机也一样,原则上一个 CPU 只能分配给一个进程,以便运行这个进程。我们通常使用的计算机中只有一个 CPU,也就是说只有一颗心,要让它一心多用同时运行多个进程,就必须使用并发技术。实现并发技术相当复杂,最容易理解的是“时间片轮转进程调度算法”。

综合来说:

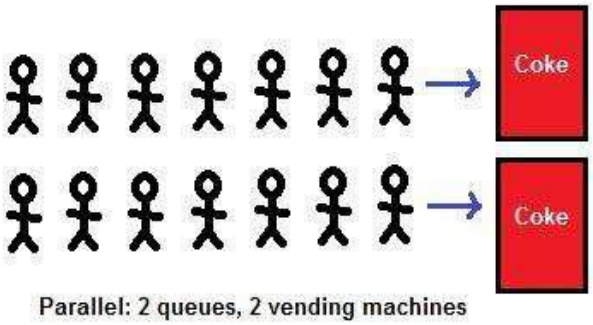
并发:指应用能够交替执行不同的任务,比如单 CPU 核心下执行多线程并非是同时执行多个任务,如果你开两个线程执行,就是在你几乎不可能察觉到的速度不断去切换这两个任务,已达到"同时执行效果",其实并不是的,只是计算机的速度太快,我们无法察觉到而已.

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.



Concurrent: 2 queues, 1 vending machine



Parallel: 2 queues, 2 vending machines

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

例如我们实现电商系统，下订单和给用户发送短信、邮件就可以进行拆分，将给用户发送短信、邮件这两个步骤独立为单独的模块，并交给其他线程去执行。这样既增加了异步的操作，提升了系统性能，又使程序模块化,清晰化和简单化。

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)线程之间的死锁

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。

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 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.start

2、X implements　Runnable；然后交给 Thread 运行

参见代码：cn.enjoyedu.ch1.base.NewThread

Thread 和 Runnable 的区别

Thread 才是 Java 里对线程的唯一抽象，Runnable 只是对任务（业务逻辑）的抽象。Thread 可以接受任意一个 Runnable 的实例并执行。

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。

stop

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。

中断

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 等），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，如果重复调用会抛出异常。

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，也可以被单独调用。

其他的线程相关方法

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。注意：并不是每个线程都需要这个锁的，而且执行 yield( )的线程不一定就会持有锁，我们完全可以在释放锁后再调用 yield 方法。

所有执行 yield()的线程有可能在进入到就绪状态后会被操作系统再次选中马上又被执行。

wait()/notify()/notifyAll()：后面会单独讲述

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。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。(此处为常见面试考点)

线程的优先级

在 Java 线程中，通过一个整型成员变量 priority 来控制优先级，优先级的范围从 1~10，在线程构建的时候可以通过 setPriority(int)方法来修改优先级，默认优先级是 5，优先级高的线程分配时间片的数量要多于优先级低的线程。

设置线程优先级时，针对频繁阻塞（休眠或者 I/O 操作）的线程需要设置较高优先级，而偏重计算（需要较多 CPU 时间或者偏运算）的线程则设置较低的优先级，确保处理器不会被独占。在不同的 JVM 以及操作系统上，线程规划会存在差异，有些操作系统甚至会忽略对线程优先级的设定。

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。我们一般用不上，比如垃圾回收线程就是 Daemon 线程。

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。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。

线程间的共享和协作

线程间的共享

synchronized 内置锁

线程开始运行，拥有自己的栈空间，就如同一个脚本一样，按照既定的代码一步一步地执行，直到终止。但是，每个运行中的线程，如果仅仅是孤立地运行，那么没有一点儿价值，或者说价值很少，如果多个线程能够相互配合完成工作，包括数据之间的共享，协同处理事情。这将会带来巨大的价值。

Java 支持多个线程同时访问一个对象或者对象的成员变量，关键字 synchronized 可以修饰方法或者以同步块的形式来进行使用，它主要确保多个线程在同一个时刻，只能有一个线程处于方法或者同步块中，它保证了线程对变量访问的可见性和排他性，又称为内置锁机制。

对象锁和类锁：

对象锁是用于对象实例方法，或者一个对象实例上的，类锁是用于类的静态方法或者一个类的 class 对象上的。我们知道，类的对象实例可以有很多个，但是每个类只有一个 class 对象，所以不同对象实例的对象锁是互不干扰的，但是每个类只有一个类锁。

但是有一点必须注意的是，其实类锁只是一个概念上的东西，并不是真实存在的，类锁其实锁的是每个类的对应的 class 对象。类锁和对象锁之间也是互不干扰的。

对象锁和类锁，以及锁 static 变量之间的运行情况，请参考包 cn.enjoyedu.ch1.syn 下的代码。