
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

Intel(R) Core(TM) i7-4710HQ CPU @ 2.50GHz



利用率 51% 速度 3.05 GHz

进程 133 线程 2599 句柄 108829

正常运行时间
36:02:47:39

最大速度: 2.50 GHz

插槽: 1

内核: 4

逻辑处理器: 8

虚拟化: 已启用

L1 缓存: 256 KB

L2 缓存: 1.0 MB

L3 缓存: 6.0 MB

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束,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(processswitch),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 才能获得运行机会。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。

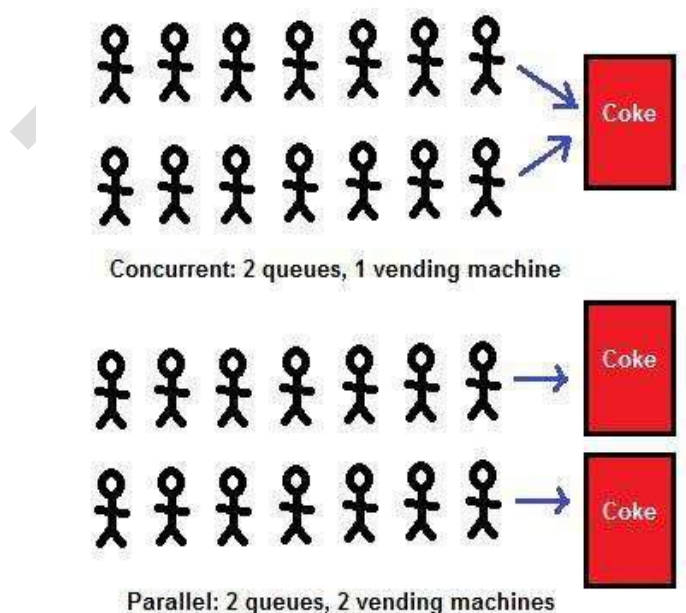
俗话说,一心不能二用,这对计算机也一样,原则上一个 CPU 只能分配给一个进程,以便运行这个进程。我们通常使用的计算机中只有一个 CPU,也就是说只有一颗心,要让它一心多用同时运行多个进程,就必须使用并发技术。实现并发技术相当复杂,最容易理解的是“时间片轮转进程调度算法”。

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,就能增加不少访问量。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其中的一个变量。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,或者某个资源的候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 线程完全可以不理睬这种中断请求。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。

错误的加锁和原因分析

参见代码 `cn.enjoyedu.ch1.syn.TestIntegerSyn`

原因：虽然我们对 i 进行了加锁，但是

```
private Integer i;

public Worker(Integer i) {
    this.i=i;
}

@Override
public void run() {
    synchronized (i) {
        Thread thread=Thread.currentThread();
        System.out.println(thread.getName()+"--@"+System
        i++;
        System.out.println(thread.getName()+"-----"+i+
        try {
            Thread.sleep( millis: 3000);
        } catch (InterruptedException e) {
            e.printStackTrace();
        }

        System.out.println(thread.getName()+"-----"+i+
    }
```

但是当我们反编译这个类的 class 文件后，可以看到 i++ 实际是，

```
= Thread.currentThread();
ntln(thread.getName() + "--@" + System.identityHashCode(this.i));
nteger1 = this.i; Integer localInteger2 = this.i = Integer.valueOf(this.i.intValue() + 1);
ntln(thread.getName() + "-----" + this.i + "--@" + System.identityHashCode(this.i));
```

```
public static Integer valueOf(int i) {
    if (i >= IntegerCache.low && i <= IntegerCache.high)
        return IntegerCache.cache[i + (-IntegerCache.low)];
    return new Integer(i);
}
```

本质上是返回了一个新的 Integer 对象。也就是每个线程实际加锁的是不同的 Integer 对象。

volatile , 最轻量的同步机制

volatile 保证了不同线程对这个变量进行操作时的可见性，即一个线程修改了某个变量的值，这新值对其他线程来说是立即可见的。参见代码：

cn.enjoyedu.ch1.vola.VolatileCase

```
public class VolatileCase {
    private static boolean ready;
    private static int number;
```

不加 volatile 时，子线程无法感知主线程修改了 ready 的值，从而不会退出循环，而加了 volatile 后，子线程可以感知主线程修改了 ready 的值，迅速退出循环。

但是 volatile 不能保证数据在多个线程下同时写时的线程安全，参见代码：

cn.enjoyedu.ch1.vola.NotSafe

volatile 最适用的场景：一个线程写，多个线程读。

ThreadLocal 辨析

与 Synchronized 的比较

ThreadLocal 和 Synchronized 都用于解决多线程并发访问。可是 ThreadLocal 与 synchronized 有本质的差别。synchronized 是利用锁的机制，使变量或代码块在某一时刻仅仅能被一个线程访问。而 ThreadLocal 为每个线程都提供了变量的副本，使得每个线程在某一时间访问到的并非同一个对象，这样就隔离了多个线程对数据的数据共享。

Spring 的事务就借助了 ThreadLocal 类。Spring 会从数据库连接池中获得一个 connection，然会把 connection 放进 ThreadLocal 中，也就和线程绑定了，事务需要提交或者回滚，只要从 ThreadLocal 中拿到 connection 进行操作。为何 Spring 的事务要借助 ThreadLocal 类？

以 JDBC 为例，正常的事务代码可能如下：

```
dbc = new DataBaseConnection();//第 1 行
Connection con = dbc.getConnection();//第 2 行
con.setAutoCommit(false);//第 3 行
con.executeUpdate(...);//第 4 行
con.executeUpdate(...);//第 5 行
con.executeUpdate(...);//第 6 行
con.commit();//第 7 行
```

上述代码，可以分成三个部分：

事务准备阶段：第 1～3 行

业务处理阶段：第 4～6 行

事务提交阶段：第 7 行

可以很明显的看到，不管我们开启事务还是执行具体的 sql 都需要一个具体的数据库连接。

现在我们开发应用一般都采用三层结构，如果我们控制事务的代码都放在 DAO(DataAccessObject)对象中，在 DAO 对象的每个方法当中去打开事务和关闭事务，当 Service 对象在调用 DAO 时，如果只调用一个 DAO，那我们这样实现则效果不错，但往往我们的 Service 会调用一系列的 DAO 对数据库进行多次操作，那么，这个时候我们就无法控制事务的边界了，因为实际应用当中，我们的 Service 调用的 DAO 的个数是不确定的，可根据需求而变化，而且还可能出现 Service 调用 Service 的情况。

如果不使用 ThreadLocal，代码大概就会是这个样子：

```
public void serviceMethod(){
    Connection conn=null;
    try{
        Connection conn=getConnection();
        conn.setAutoCommit(false);
        Dao1 dao1=new Dao1(conn);
        dao1.doSomething();
        Dao2 dao2=new Dao2(conn);
        dao2.doSomething();
        Dao3 dao3=new Dao3(conn);
        dao3.doSomething();
        conn.commit();
    }
}

Class Dao1{
    private Connection conn=null;
    public Dao1(Connection conn){
        this.conn=conn;
    }
    public void doSomething(){
        PreparedStatement pstmt=null;
        try{
            pstmt=conn.prepareStatement(sql);
            pstmt.execute...
        }
        ...
    }
}
```

但是需要注意一个问题，如何让三个 DAO 使用同一个数据源连接呢？我们就必须为每个 DAO 传递同一个数据库连接，要么就是在 DAO 实例化的时候作为构造方法的参数传递，要么在每个 DAO 的实例方法中作为方法的参数传递。这两种方式无疑对我们的 Spring 框架或者开发人员来说都不合适。为了让这个数据库连接可以跨阶段传递，又不显示的进行参数传递，就必须使用别的办法。

Web 容器中，每个完整的请求周期会由一个线程来处理。因此，如果我们将一些参数绑定到线程的话，就可以实现在软件架构中跨层次的参数共享（是隐式的共享）。而 JAVA 中恰好提供了绑定的方法--使用 ThreadLocal。

结合使用 Spring 里的 IOC 和 AOP，就可以很好的解决这一点。

只要将一个数据库连接放入 ThreadLocal 中，当前线程执行时只要有使用数据库连接的地方就从 ThreadLocal 获得就行了。

ThreadLocal 的使用

ThreadLocal 类接口很简单，只有 4 个方法，我们先来了解一下：

- void set(Object value)

设置当前线程的线程局部变量的值。

- public Object get()

该方法返回当前线程所对应的线程局部变量。

- public void remove()

将当前线程局部变量的值删除，目的是为了减少内存的占用，该方法是 JDK 5.0 新增的方法。需要指出的是，当线程结束后，对应该线程的局部变量将自动被垃圾回收，所以显式调用该方法清除线程的局部变量并不是必须的操作，但可以加快内存回收的速度。

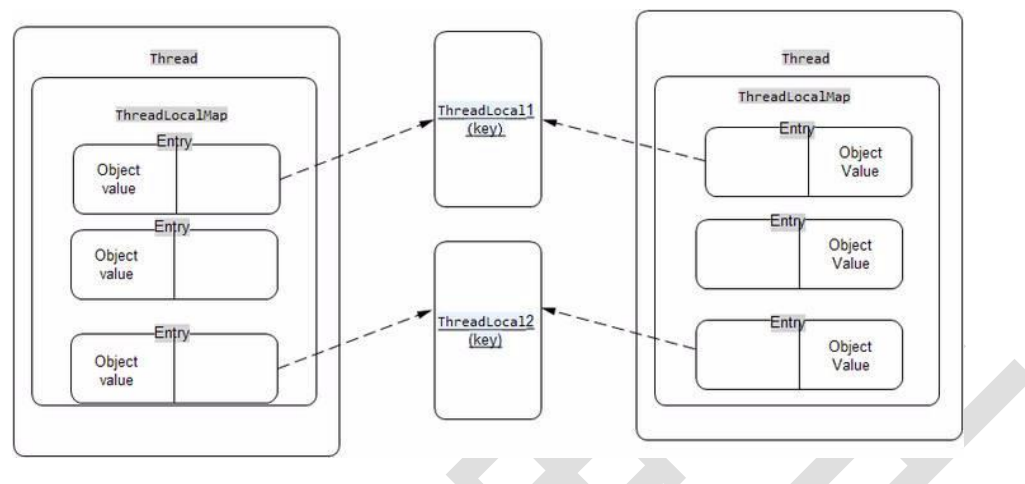
- protected Object initialValue()

返回该线程局部变量的初始值，该方法是一个 protected 的方法，显然是为了让子类覆盖而设计的。这个方法是一个延迟调用方法，在线程第 1 次调用 get() 或 set(Object) 时才执行，并且仅执行 1 次。ThreadLocal 中的缺省实现直接返回一个 null。

```
public final static ThreadLocal<String> RESOURCE = new  
ThreadLocal<String>();
```

RESOURCE 代表一个能够存放 String 类型的 ThreadLocal 对象。此时不论什么一个线程能够并发访问这个变量，对它进行写入、读取操作，都是线程安全的。

实现解析



```
public T get() {  
    Thread t = Thread.currentThread();  
    ThreadLocalMap map = getMap(t);  
    if (map != null) {  
        ThreadLocalMap.Entry e = map.getEntry(key: this);  
        if (e != null) {  
            /unchecked/  
            T result = (T) e.value;  
            return result;  
        }  
    }  
    return setInitialValue();  
}
```

```
ThreadLocalMap getMap(Thread t) {  
    return t.threadLocals;  
}
```

java × OnlyMain.java × ThreadPoolExecutor.java × ThreadLocal.java × Thread.java ×

```
ThreadLocal.ThreadLocalMap threadLocals = null;
```

上面先取到当前线程，然后调用 getMap 方法获取对应的 ThreadLocalMap，ThreadLocalMap 是 ThreadLocal 的静态内部类，然后 Thread 类中有一个这样类型成员，所以 getMap 是直接返回 Thread 的成员。

看下 ThreadLocal 的内部类 ThreadLocalMap 源码：


```

static class ThreadLocalMap {
    /**
     * The entries in this hash map extend WeakReference, using
     * its main ref field as the key (which is always a
     * ThreadLocal object). Note that null keys (i.e. entry.get()
     * == null) mean that the key is no longer referenced, so the
     * entry can be expunged from table. Such entries are referred to
     * as "stale entries" in the code that follows.
     */
    static class Entry extends WeakReference<ThreadLocal<?>> {
        /** The value associated with this ThreadLocal. */
        Object value;

        Entry(ThreadLocal<?> k, Object v) {
            super(k);
            value = v;
        }
    }

    /**
     * The initial capacity -- MUST be a power of two.
     */
    private static final int INITIAL_CAPACITY = 16;

    /**
     * The table, resized as necessary.
     * table.length MUST always be a power of two.
     */
    private Entry[] table;
}

```

类似于map的key, value结构
key就是ThreadLocal, Value就
需要隔离访问的变量

用数组保存了Entry, 因为可能有多个变量
需要线程隔离访问

可以看到有个 Entry 内部静态类, 它继承了 WeakReference, 总之它记录了两个信息, 一个是 ThreadLocal<?>类型, 一个是 Object 类型的值。getEntry 方法则是获取某个 ThreadLocal 对应的值, set 方法就是更新或赋值相应的 ThreadLocal 对应的值。

```

private Entry getEntry(ThreadLocal<?> key) {
    int i = key.threadLocalHashCode & (table.length - 1);
    Entry e = table[i];

    private void set(ThreadLocal<?> key, Object value) {
        // We don't use a fast path as with get() because

```

回顾我们的 get 方法, 其实就是拿到**每个线程独有的 ThreadLocalMap**

然后再用 ThreadLocal 的当前实例, 拿到 Map 中的相应的 Entry, 然后就可以拿到相应的值返回出去。当然, 如果 Map 为空, 还会先进行 map 的创建, 初始化等工作。

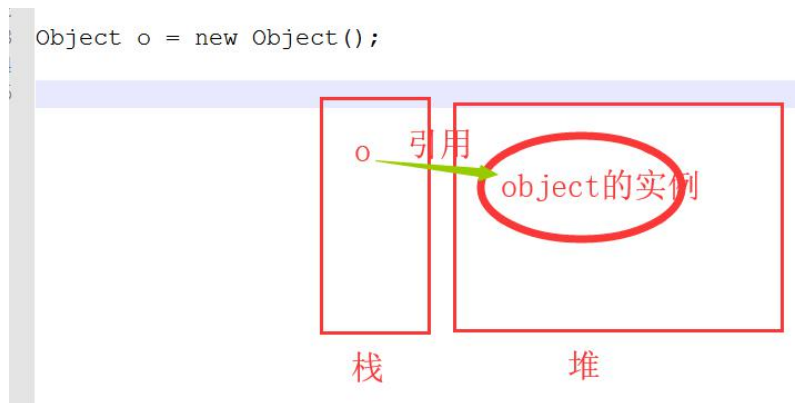
引发的内存泄漏分析

预备知识

引用

Object o = new Object();

这个 `o`，我们可以称之为对象引用，而 `new Object()` 我们可以称之为在内存中产生了一个对象实例。



当写下 `o=null` 时，只是表示 `o` 不再指向堆中 `object` 的对象实例，不代表这个对象实例不存在了。

强引用就是指在程序代码之中普遍存在的，类似“`Object obj=new Object()`”这类的引用，只要强引用还存在，垃圾收集器永远不会回收掉被引用的对象实例。

软引用是用来描述一些还有用但并非必需的对象。对于软引用关联着的对象，在系统将要发生内存溢出异常之前，将会把这些对象实例列进回收范围之中进行第二次回收。如果这次回收还没有足够的内存，才会抛出内存溢出异常。在 JDK 1.2 之后，提供了 `SoftReference` 类来实现软引用。

弱引用也是用来描述非必需对象的，但是它的强度比软引用更弱一些，被弱引用关联的对象实例只能生存到下一次垃圾收集发生之前。当垃圾收集器工作时，无论当前内存是否足够，都会回收掉只被弱引用关联的对象实例。在 JDK 1.2 之后，提供了 `WeakReference` 类来实现弱引用。

虚引用也称为幽灵引用或者幻影引用，它是最弱的一种引用关系。一个对象实例是否有虚引用的存在，完全不会对其生存时间构成影响，也无法通过虚引用来取得一个对象实例。为一个对象设置虚引用关联的唯一目的就是能在这个对象实例被收集器回收时收到一个系统通知。在 JDK 1.2 之后，提供了 `PhantomReference` 类来实现虚引用。

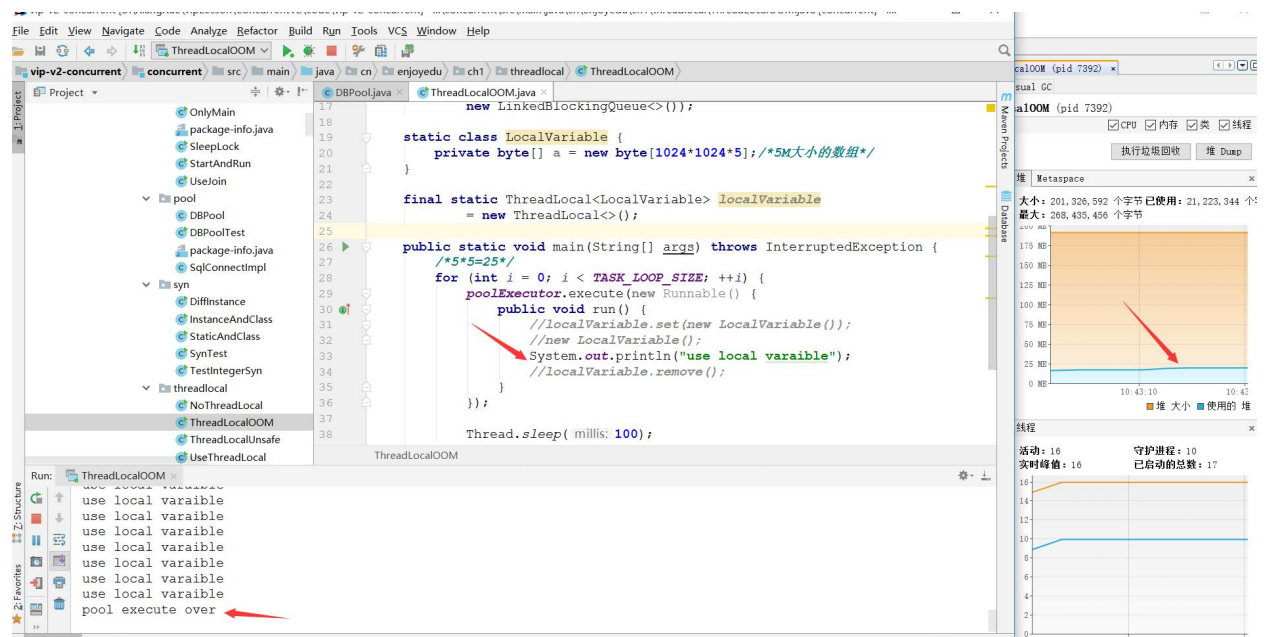
内存泄漏的现象

执行 `cn.enjoyedu.ch1.threadlocal` 下的 `ThreadLocalOOM`，并将堆内存大小设置为 `-Xmx256m`，

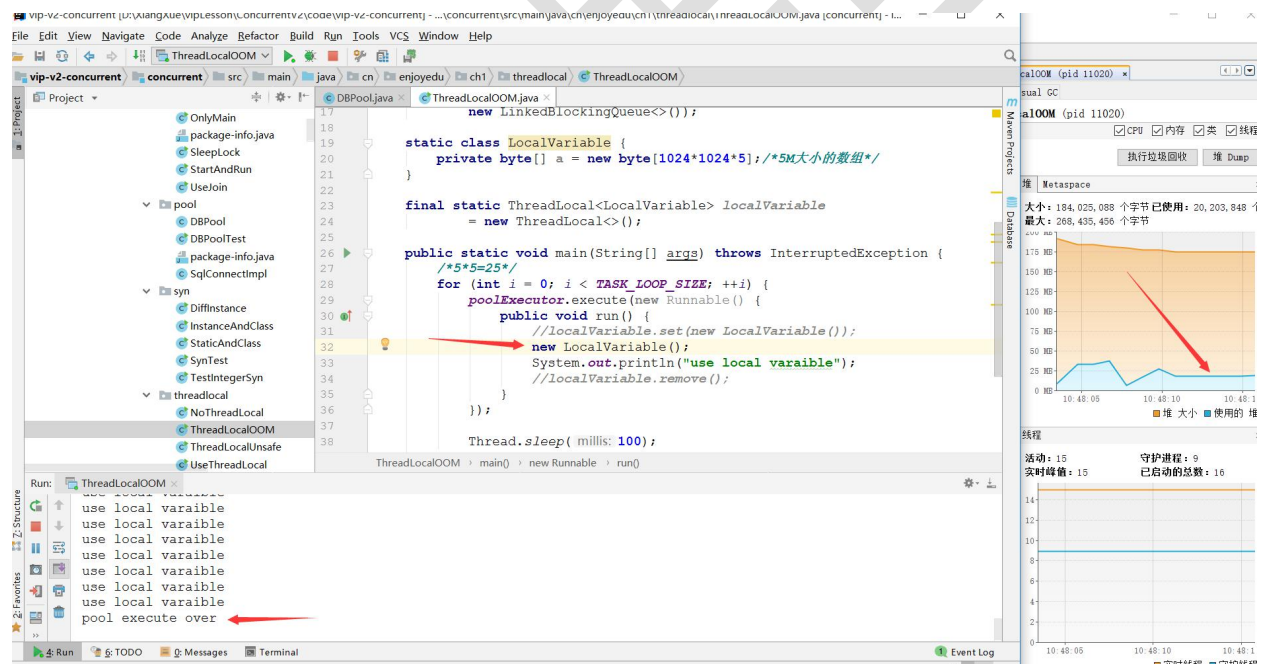
我们启用一个线程池，大小固定为 5 个线程

```
final static ThreadPoolExecutor poolExecutor
    = new ThreadPoolExecutor( corePoolSize: 5, maximumPoolSize: 5, keepAliveTime: 1,
        TimeUnit.MINUTES,
        new LinkedBlockingQueue<>());
```

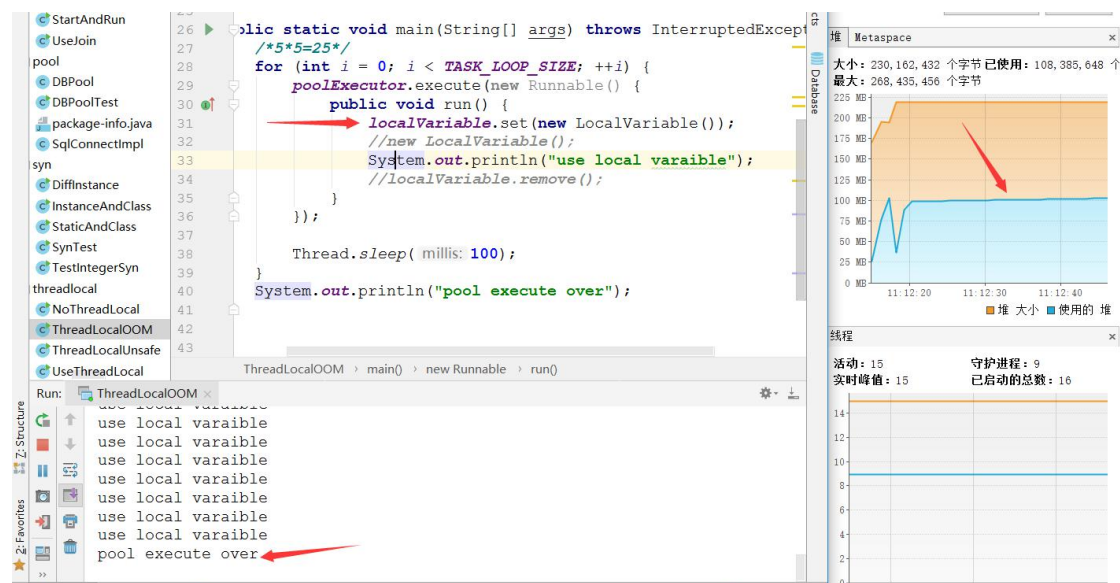
场景 1，首先任务中不执行任何有意义的代码，当所有的任务提交执行完成后，可以看见，我们这个应用的内存占用基本上为 25M 左右



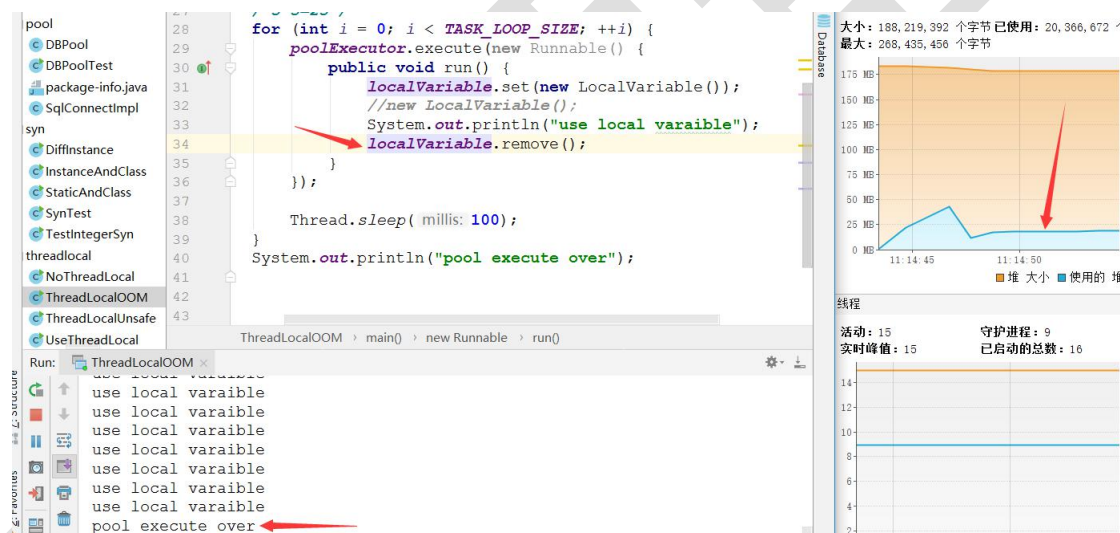
场景 2，然后我们只简单的在每个任务中 new 出一个数组，执行完成后我们可以看见，内存占用基本和场景 1 同



场景 3，当我们启用了 ThreadLocal 以后：



执行完成后我们可以看见，内存占用变为了 100M 左右
场景 4，于是，我们加入一行代码，再执行，看看内存情况：



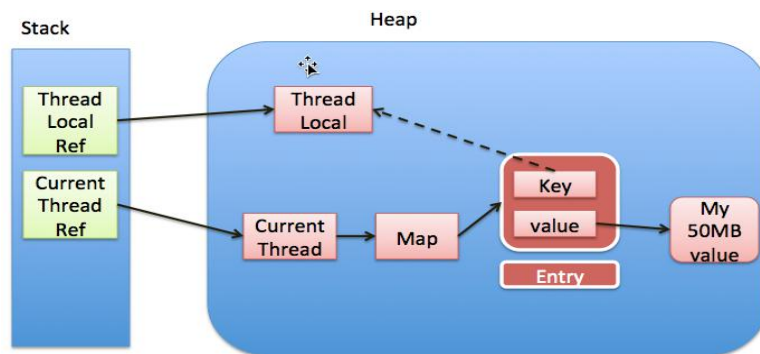
可以看见，内存占用基本和场景 1 同。

这就充分说明，场景 3，当我们启用了 ThreadLocal 以后确实发生了内存泄漏。

分析

根据我们前面对 ThreadLocal 的分析，我们可以知道每个 Thread 维护一个 ThreadLocalMap，这个映射表的 key 是 ThreadLocal 实例本身，value 是真正需要存储的 Object，也就是说 ThreadLocal 本身并不存储值，它只是作为一个 key 来让线程从 ThreadLocalMap 获取 value。仔细观察 ThreadLocalMap，这个 map 是使用 ThreadLocal 的弱引用作为 Key 的，弱引用的对象在 GC 时会被回收。

因此使用了 ThreadLocal 后，引用链如图所示



图中的虚线表示弱引用。

这样，当把 `threadlocal` 变量置为 `null` 以后，没有任何强引用指向 `threadlocal` 实例，所以 `threadlocal` 将会被 gc 回收。这样一来，`ThreadLocalMap` 中就会出现 `key` 为 `null` 的 `Entry`，就没有办法访问这些 `key` 为 `null` 的 `Entry` 的 `value`，如果当前线程再迟迟不结束的话，这些 `key` 为 `null` 的 `Entry` 的 `value` 就会一直存在一条强引用链：`Thread Ref -> Thread -> ThreaLocalMap -> Entry -> value`，而这块 `value` 永远不会被访问到了，所以存在着内存泄露。

只有当前 `thread` 结束以后，`current thread` 就不会存在栈中，强引用断开，`Current Thread`、`Map value` 将全部被 GC 回收。最好的做法是不在需要使用 `ThreadLocal` 变量后，都调用它的 `remove()` 方法，清除数据。

所以回到我们前面的实验场景，场景 3 中，虽然线程池里面的任务执行完毕了，但是线程池里面的 5 个线程会一直存在直到 JVM 退出，我们 `set` 了线程的 `localVariable` 变量后没有调用 `localVariable.remove()` 方法，导致线程池里面的 5 个线程的 `threadLocals` 变量里面的 `new LocalVariable()` 实例没有被释放。

其实考察 `ThreadLocal` 的实现，我们可以看见，无论是 `get()`、`set()` 在某些时候，调用了 `expungeStaleEntry` 方法用来清除 `Entry` 中 `Key` 为 `null` 的 `Value`，但是这是不及时的，也不是每次都会执行的，所以一些情况下还是会发生内存泄露。只有 `remove()` 方法中显式调用了 `expungeStaleEntry` 方法。

从表面上看内存泄漏的根源在于使用了弱引用，但是另一个问题也同样值得思考：为什么使用弱引用而不是强引用？

下面我们分两种情况讨论：

key 使用强引用：引用 `ThreadLocal` 的对象被回收了，但是 `ThreadLocalMap` 还持有 `ThreadLocal` 的强引用，如果没有手动删除，`ThreadLocal` 的对象实例不会被回收，导致 `Entry` 内存泄漏。

key 使用弱引用：引用的 `ThreadLocal` 的对象被回收了，由于 `ThreadLocalMap` 持有 `ThreadLocal` 的弱引用，即使没有手动删除，`ThreadLocal` 的对象实例也会被回收。`value` 在下次 `ThreadLocalMap` 调用 `set`，`get`，`remove` 都有机会被回收。

比较两种情况，我们可以发现：由于 `ThreadLocalMap` 的生命周期跟 `Thread` 一样长，如果都没有手动删除对应 `key`，都会导致内存泄漏，但是使用弱引用可以多一层保障。

因此，ThreadLocal 内存泄漏的根源是：由于 ThreadLocalMap 的生命周期跟 Thread 一样长，如果没有手动删除对应 key 就会导致内存泄漏，而不是因为弱引用。

总结

JVM 利用设置 ThreadLocalMap 的 Key 为弱引用，来避免内存泄露。

JVM 利用调用 remove、get、set 方法的时候，回收弱引用。

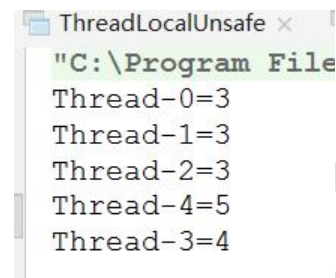
当 ThreadLocal 存储很多 Key 为 null 的 Entry 的时候，而不再去调用 remove、get、set 方法，那么将导致内存泄漏。

使用线程池+ ThreadLocal 时要小心，因为这种情况下，线程是一直在不断的重复运行的，从而也就造成了 value 可能造成累积的情况。

错误使用 ThreadLocal 导致线程不安全

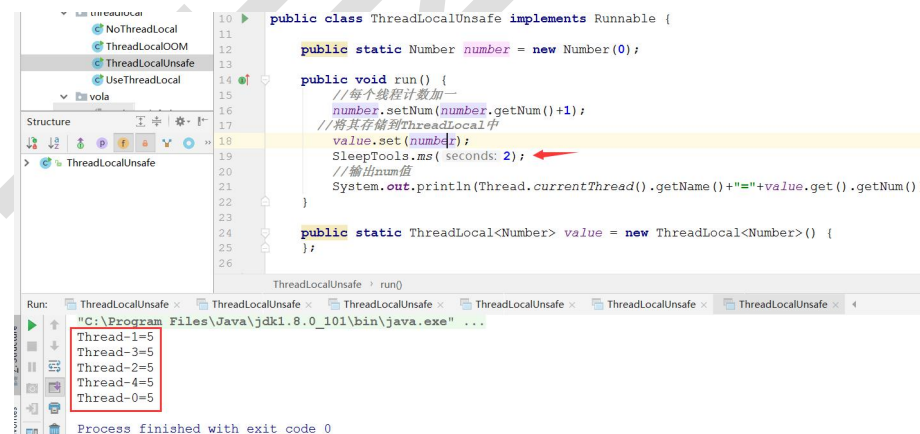
参见代码 cn.enjoyedu.ch1.threadlocal. ThreadLocalUnsafe

运行后的结果为



```
"C:\Program File
Thread-0=3
Thread-1=3
Thread-2=3
Thread-4=5
Thread-3=4
```

如果我们加入 SleepTools.ms(2);会看的更明显



为什么每个线程都输出 5？难道他们没有独自保存自己的 Number 副本吗？为什么其他线程还是能够修改这个值？仔细考察 ThreadLocal 和 Thead 的代码，我们发现 ThreadLocalMap 中保存的其实是对象的一个引用，这样的话，当有其他线程对这个引用指向的对象实例做修改时，其实也同时影响了所有的线程持有的对象引用所指向的同一个对象实例。这也就是为什么上面的程序为什么会输出一样的结果：5 个线程中保存的是同一 Number 对象的引用，在线程睡眠的时候，其他线程将 num 变量进行了修改，而修改的对象 Number 的实例是同一份，因此它们最终输出的结果是相同的。

而上面的程序要正常的工作，应该的用法是让每个线程中的 `ThreadLocal` 都应该持有一个新的 `Number` 对象。

线程间的协作

线程之间相互配合，完成某项工作，比如：一个线程修改了一个对象的值，而另一个线程感知到了变化，然后进行相应的操作，整个过程开始于一个线程，而最终执行又是另一个线程。前者是生产者，后者就是消费者，这种模式隔离了“做什么”（`what`）和“怎么做”（`How`），简单的办法是让消费者线程不断地循环检查变量是否符合预期在 `while` 循环中设置不满足的条件，如果条件满足则退出 `while` 循环，从而完成消费者的工作。却存在如下问题：

1) 难以确保及时性。

2) 难以降低开销。如果降低睡眠的时间，比如休眠 1 毫秒，这样消费者能更加迅速地发现条件变化，但是却可能消耗更多的处理器资源，造成了无端的浪费。

等待/通知机制

是指一个线程 A 调用了对象 O 的 `wait()` 方法进入等待状态，而另一个线程 B 调用了对象 O 的 `notify()` 或者 `notifyAll()` 方法，线程 A 收到通知后从对象 O 的 `wait()` 方法返回，进而执行后续操作。上述两个线程通过对象 O 来完成交互，而对象上的 `wait()` 和 `notify/notifyAll()` 的关系就如同开关信号一样，用来完成等待方和通知方之间的交互工作。

`notify()`:

通知一个在对象上等待的线程,使其从 `wait` 方法返回,而返回的前提是该线程获取到了对象的锁，没有获得锁的线程重新进入 `WAITING` 状态。

`notifyAll()`:

通知所有等待在该对象上的线程

`wait()`

调用该方法的线程进入 `WAITING` 状态,只有等待另外线程的通知或被中断才会返回.需要注意,调用 `wait()` 方法后,会释放对象的锁

`wait(long)`

超时等待一段时间,这里的参数时间是毫秒,也就是等待长达 n 毫秒,如果没有通知就超时返回

`wait (long,int)`

对于超时时间更细粒度的控制,可以达到纳秒

等待和通知的标准范式

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。

```
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。

```
synchronized(对象) {  
    改变条件  
    对象.notifyAll();  
}
```

在调用 `wait()`、`notify()` 系列方法之前，线程必须要获得该对象的对象级锁，即只能在同步方法或同步块中调用 `wait()` 方法、`notify()` 系列方法，进入 `wait()` 方法后，当前线程释放锁，在从 `wait()` 返回前，线程与其他线程竞争重新获得锁，执行 `notify()` 系列方法的线程退出调用了 `notifyAll()` 的 `synchronized` 代码块的时候后，他们就会去竞争。如果其中一个线程获得了该对象锁，它就会继续往下执行，在它退出 `synchronized` 代码块，释放锁后，其他的已经被唤醒的线程将会继续竞争获取该锁，一直进行下去，直到所有被唤醒的线程都执行完毕。

notify 和 notifyAll 应该用谁

尽可能用 `notifyAll()`，谨慎使用 `notify()`，因为 `notify()` 只会唤醒一个线程，我们无法确保被唤醒的这个线程一定就是我们需要唤醒的线程，具体表现参见代码：包 `cn.enjoyedu.ch1.wn` 下

等待超时模式实现一个连接池

调用场景：调用一个方法时等待一段时间（一般来说是给定一个时间段），如果该方法能够在给定的时间段之内得到结果，那么将结果立刻返回，反之，超时返回默认结果。

假设等待时间段是 `T`，那么可以推断出在当前时间 `now+T` 之后就会超时
等待持续时间：`REMAINING=T`。

• 超时时间：`FUTURE=now+T`。

// 对当前对象加锁

```
public synchronized Object get(long mills) throws InterruptedException {  
    long future = System.currentTimeMillis() + mills;  
    long remaining = mills;  
    // 当超时大于 0 并且 result 返回值不满足要求  
    while ((result == null) && remaining > 0) {
```

```
        wait(remaining);
        remaining = future - System.currentTimeMillis();
    }
    return result;
}
```

具体实现参见：包下 `cn.enjoyedu.ch1.pool` 的代码

客户端获取连接的过程被设定为等待超时的模式，也就是在 1000 毫秒内如果无法获取到可用连接，将会返回给客户端一个 `null`。设定连接池的大小为 10 个，然后通过调节客户端的线程数来模拟无法获取连接的场景。

它通过构造函数初始化连接的最大上限，通过一个双向队列来维护连接，调用方需要先调用 `fetchConnection(long)` 方法来指定在多少毫秒内超时获取连接，当连接使用完成后，需要调用 `releaseConnection(Connection)` 方法将连接放回线程池

面试题

调用 `yield()`、`sleep()`、`wait()`、`notify()` 等方法对锁有何影响？

`yield()`、`sleep()` 被调用后，都不会释放当前线程所持有的锁。

调用 `wait()` 方法后，会释放当前线程持有的锁，而且当前被唤醒后，会重新去竞争锁，锁竞争到后才会执行 `wait` 方法后面的代码。

调用 `notify()` 系列方法后，对锁无影响，线程只有在 `syn` 同步代码执行完后才会自然而然的释放锁，所以 `notify()` 系列方法一般都是 `syn` 同步代码的最后一行。

2、线程的并发工具类

Fork-Join

java 下多线程的开发可以我们自己启用多线程，线程池，还可以使用 `forkjoin`，`forkjoin` 可以让我们不去了解诸如 `Thread`、`Runnable` 等相关的知识，只要遵循 `forkjoin` 的开发模式，就可以写出很好的多线程并发程序，

分而治之

同时 `forkjoin` 在处理某一类问题时非常的有用，哪一类问题？分而治之的问题。十大计算机经典算法：快速排序、堆排序、归并排序、二分查找、线性查找、

深度优先、广度优先、`Dijkstra`、动态规划、朴素贝叶斯分类，有几个属于分而治之？3 个，快速排序、归并排序、二分查找，还有大数据中 `M/R` 都是。

分治法的设计思想是：将一个难以直接解决的大问题，分割成一些规模较小的相同问题，以便各个击破，分而治之。

分治策略是：对于一个规模为 n 的问题，若该问题可以容易地解决（比如说规模 n 较小）则直接解决，否则将其分解为 k 个规模较小的子问题，**这些子问题互相独立且与原问题形式相同(子问题相互之间有联系就会变为动态规范算法)**，递归地解这些子问题，然后将各子问题的解合并得到原问题的解。这种算法设计策略叫做分治法。

归并排序

归并排序是建立在归并操作上的一种有效的排序算法。该算法是采用分治法的一个非常典型的应用。将已有序的子序列合并，得到完全有序的序列；即先使每个子序列有序，再使子序列段间有序。

若将两个有序表合并成一个有序表，称为 2-路归并，与之对应的还有多路归并。

对于给定的一组数据，利用递归与分治技术将数据序列划分成为越来越小的半子表，在对半子表排序后，再用递归方法将排好序的半子表合并成为越来越大的有序序列。

为了提升性能，有时我们在半子表的个数小于某个数（比如 15）的情况下，对半子表的排序采用其他排序算法，比如插入排序。

归并排序（降序）示例



先将数组划分为左右两个子表：



然后继续左右两个子表拆分：



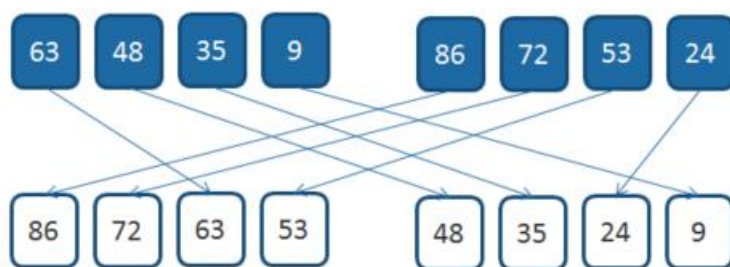
对最后的拆分的子表，两两进行排序



对有序的子表进行排序和比较合并

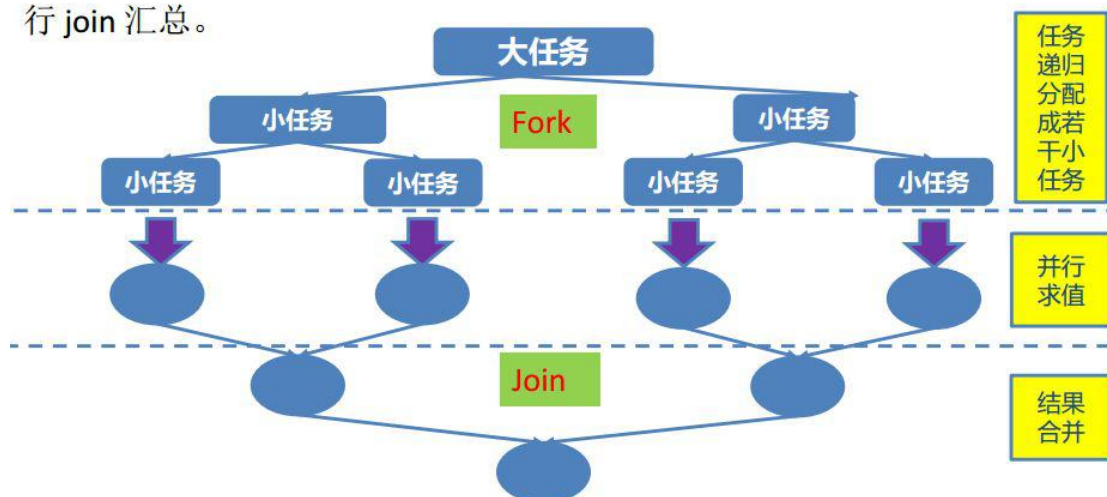


对合并后的子表继续比较合并



Fork-Join 原理

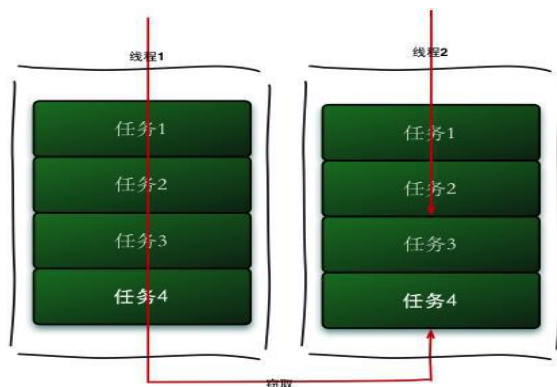
Fork/Join 框架：就是在必要的情况下，将一个大任务，进行拆分(fork)成若干个小任务（拆到不可再拆时），再将一个个的小任务运算的结果进行 join 汇总。



工作窃取

即当前线程的 Task 已经全被执行完毕，则自动取到其他线程的 Task 池中取出 Task 继续执行。

ForkJoinPool 中维护着多个线程（一般为 CPU 核数）在不断地执行 Task，每个线程除了执行自己任务内的 Task 之外，还会根据自己工作线程的闲置情况去获取其他繁忙的工作线程的 Task，如此一来就能减少线程阻塞或是闲置的时间，提高 CPU 利用率。



Fork/Join 实战

Fork/Join 使用的标准范式

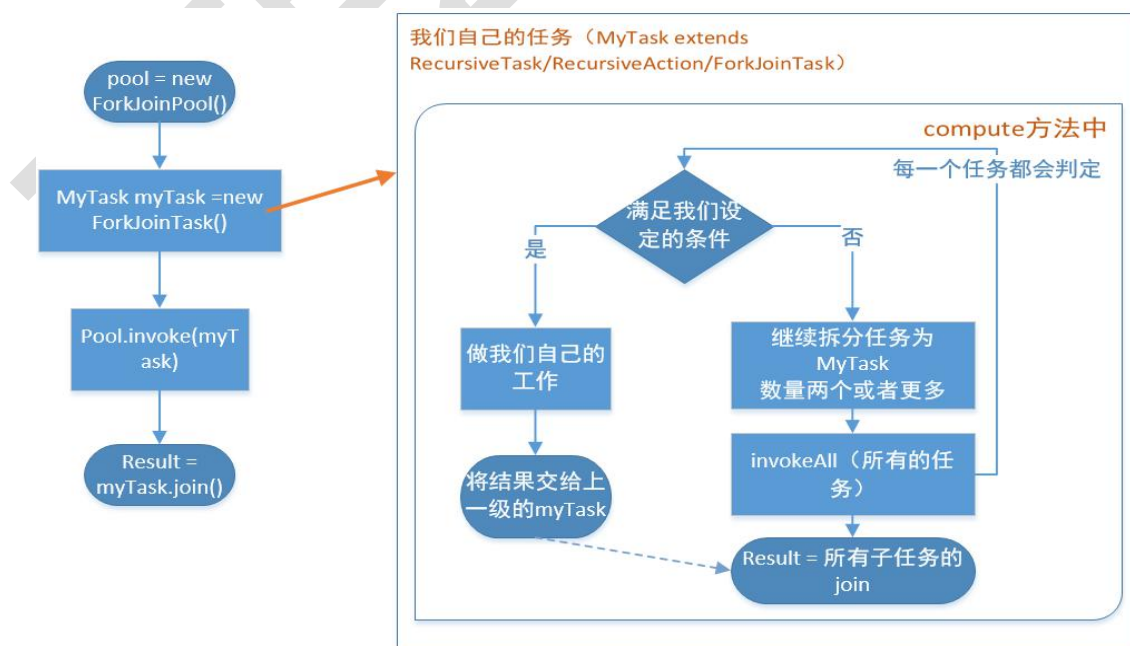
我们要使用 ForkJoin 框架，必须首先创建一个 ForkJoin 任务。它提供在任务中执行 fork 和 join 的操作机制，通常我们不直接继承 ForkJoinTask 类，只需要直接继承其子类。

1. RecursiveAction，用于没有返回结果的任务

2. RecursiveTask，用于有返回值的任务

task 要通过 ForkJoinPool 来执行，使用 submit 或 invoke 提交，两者的区别是：invoke 是同步执行，调用之后需要等待任务完成，才能执行后面的代码；submit 是异步执行。

join()和 get 方法当任务完成的时候返回计算结果。



在我们自己实现的 compute 方法里，首先需要判断任务是否足够小，如果足够小就直接执行任务。如果不够小，就必须分割成两个子任务，每个子任务

在调用 `invokeAll` 方法时，又会进入 `compute` 方法，看看当前子任务是否需要继续分割成孙任务，如果不需要继续分割，则执行当前子任务并返回结果。使用 `join` 方法会等待子任务执行完并得到其结果。

Fork/Join 的同步用法和异步用法

参见代码包 `cn.enjoyedu.ch2.forkjoin` 下

CountDownLatch

闭锁，`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。例如，应用程序的主线程希望在负责启动框架服务的线程已经启动所有的框架服务之后再执行。

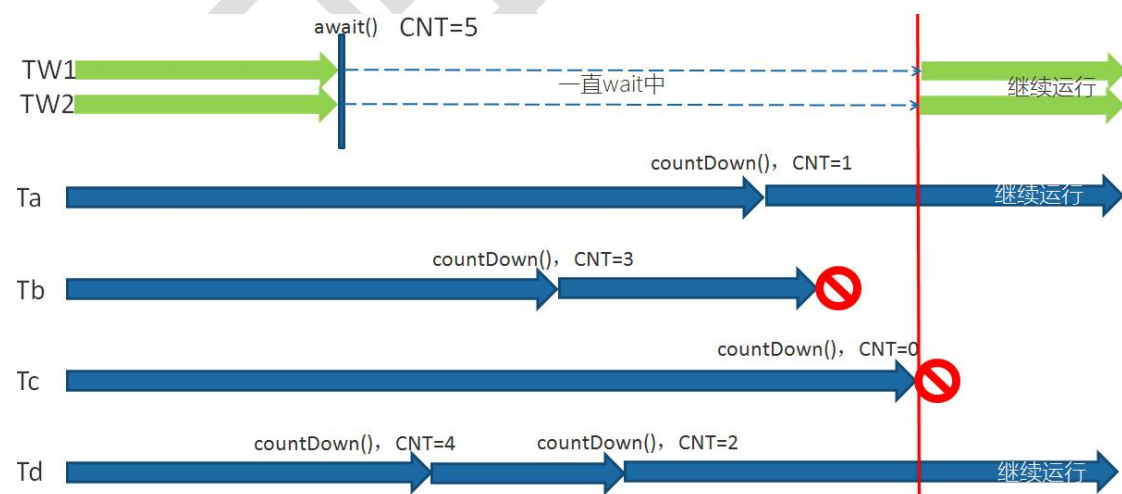
`CountDownLatch` 是通过一个计数器来实现的，计数器的初始值为初始任务的数量。每当完成了一个任务后，计数器的值就会减 1

（`CountDownLatch.countDown()` 方法）。当计数器值到达 0 时，它表示所有的已经完成了任务，然后在闭锁上等待 `CountDownLatch.await()` 方法的线程就可以恢复执行任务。

应用场景：

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。

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，例如处理 excel 中多个表单。



参见代码包 `cn.enjoyedu.ch2.tools` 下