**Thread以及安全场景**

## 一、进程和线程

**1 进程：每个进程都有独立的代码和数据空间（进程上下文）**，进程间的切换会有较大的开销，一个进程包含1--n个线程。（进程是资源分配的最小单位）

线程：同一类线程共享代码和数据空间，每个线程有独立的运行栈和程序计数器(PC)，线程切换开销小。（线程是cpu调度的最小单位）

线程和进程一样分为五个阶段：**创建、就绪、运行、阻塞、终止**。

多进程是指操作系统能同时运行多个任务（程序），多线程是指在同一程序中有多个顺序流在执行。创建线程类的手段（继续Thread类；实现Runable接口；实现Callable接口，并与Future、线程池结合使用）

**2 start()方法的调用后并不是立即执行多线程代码**，而是使得该线程变为可运行态（Runnable），什么时候运行是由操作系统决定的。Thread.sleep()方法调用目的是不让当前线程独自霸占该进程所获取的CPU资源，以留出一定时间给其他线程执行的机会。

实际上所有的多线程代码执行顺序都是不确定的，每次执行的结果都是随机的。

**3 start()方法来启动一个线程**，这时此线程处于就绪（可运行）状态，并没有运行，一旦得到cpu时间片，就开始执行run()方法。

**4 run()称为线程体**，它包含了要执行的这个线程的内容，Run方法运行结束，此线程随即终止，只是类的一个**普通方法而**已，如果直接调用Run方法，程序中依然只有主线程这一个线程，其程序执行路径还是只有一条，还是要顺序执行。

**5 实现Runnable接口比继承Thread类所具有的优势：**

1）：资源共享，适合多个相同的程序代码的线程去处理同一个资源

2）：可以避免java中的单继承的限制

3）：增加程序的健壮性，代码可以被多个线程共享，代码和数据独立

4）：线程池只能放入实现Runable或callable类线程，不能直接放入继承Thread的类

## 二、线程状态

**1、新建状态（New）：新创建**了一个线程对象。

**2、就绪状态（Runnable）：**线程对象创建后，其他线程调用了该对象的**start**()方法。该状态的线程位于可运行线程池中，变得可运行，等待获取CPU的使用权。

**3、运行状态（Running）**：就绪状态的线程**获取了CPU**，执行程序代码。

**4、阻塞状态（Blocked）：**阻塞状态是线程因为某种原因放弃CPU使用权，暂时停止运行。直到线程进入就绪状态，才有机会转到运行状态。阻塞的情况分三种：

（一）、等待阻塞：运行的线程执行**wait()**方法，JVM会把该线程放入等待池中。(wait会释放持有的锁) 。在获取对象的同步锁时，若该同步锁被别的线程占用，则JVM会把该线程放入锁池中。

（二）、同步阻塞：运行的线程

（三）、其他阻塞：运行的线程执行sleep(不会释放持有的锁)或join()方法，或者发出了I/O请求时，JVM会把该线程置为阻塞状态。当sleep()超时、join()、I/O处理完毕时，线程重新转入就绪状态。

**5、死亡状态（Dead）：**线程执行完了或者因异常退出了run()方法，该线程结束生命周期。

## 三、常用的API

**1、让出CPU：**yield():暂停当前正在执行的线程对象，并执行其他线程。Thread.yield()方法作用是：暂停当前正在执行的线程对象，并执行其他线程。

**2、线程睡眠**：Thread.sleep(long millis)方法，使线程转到阻塞状态。millis参数设定睡眠的时间，以毫秒为单位。当睡眠结束后，就转为就绪（Runnable）状态。sleep()平台移植性好。Thread.sleep()与Object.wait()二者都可以暂停当前线程，释放CPU控制权，主要的区别在于Object.wait()在释放CPU同时，释放了对象锁的控制。

**3、线程等待**：Object类中的wait()方法，导致当前的线程等待，直到其他线程调用此对象的 notify() 方法或 notifyAll() 唤醒方法。**一般出现在判断条件中，比如消费者和生产者，如果生产队列满了，就wait（），一旦发现判断没有满，就 notifyAll()** Obj.wait()，与Obj.notify()必须要与synchronized(Obj)一起使用，也就是wait,与notify是针对已经获取了Obj锁进行操作，从语法角度来说就是Obj.wait(),Obj.notify必须在synchronized(Obj){...}语句块内。

**4、线程让步：**Thread.yield() 方法，暂停当前正在执行的线程对象，把执行机会让给相同或者更高优先级的线程。

**5、线程加入：**join()方法，在当前线程中调用另一个线程的join()方法，则当前线程转入阻塞状态，直到另一个进程运行结束，当前线程再由阻塞转为就绪状态。通常是串行化，等待另一个执行完在执行当前的线程**，可以获取另一个线程的结果**。

**6、线程唤醒：**Object类中的notify()方法，唤醒在此对象监视器上等待的单个线程。选择唤醒其中一个线程。

**7、所有唤醒：**notifyAll()，唤醒在此对象监视器上等待的所有线程。

**8、中断信号：**interrupt()，不要以为它是中断某个线程！它只是线线程发送一个中断信号，让线程在无限等待时（如死锁时）能抛出抛出，从而结束线程，但是如果你吃掉了这个异常，那么这个线程还是不会中断的！

\* 这种场景是 独立一个复合对象（自定义后者jdk里面的）

\* 初始化**一个线程类**，构造好参数，用同一个对象去创建2个线程。分别start ...

\* 这样多个线程共享资源了

\* 一般2种情况，1是分布式处理各自的任务，1是共享资源

\* 还有一种情况是继承Thread，new出的对象不一样，只能处理各自的任务

\* 或者new出两个 Account，也不是供向资源

## 死锁

**两个线程互相等待对方释放同步监视器，就发生了死锁。**

死锁的产生是有规律可循的，只有同时满足以下四个条件，死锁才会产生。

**1.互斥条件：**一个资源每次只能被一个进程使用。独木桥每次只能通过一个人。

**2.请求与保持条件：**一个进程因请求资源而阻塞时，对已获得的资源保持不放。乙不退出桥面，甲也不退出桥面。

**3.不剥夺条件:** 进程已获得的资源，在未使用完之前，不能强行剥夺。甲不能强制乙退出桥面，乙也不能强制甲退出桥面。

**4.循环等待条件：**若干进程之间形成一种头尾相接的循环等待资源关系。如果乙不退出桥面，甲不能通过，甲不退出桥面，乙不能通过。

**Jdk并发包**

一、JUC（java.util.concurrent）

Java 5 添加了一个java.util.concurrent 包。这个包包含有一系列能够让 Java 的并发编程变得更加简单轻松的类。在这个包被添加以前，你需要自己去动手实现自己的相关工具类。

用于定义类似于线程的自定义子系统，包括线程池、异步 IO 和轻量级任务框架。提供可调的、灵活的线程池。还提供了设计用于多线程上下文中的 Collection 实现等。

**Volatile**

## 内存可见性（Memory Visibility）

**1 指当某个线程正在使用对象状态而另一个线程在同时修改该状态，需要确保当一个线程修改了对象状态后，其他线程能够看到发生的状态变化。**

可见性错误是指当读操作与写操作在不同的线程中执行时，我们无法确保执行读操作的线程能适时地看到其他线程写入的值，有时甚至是根本不可能的事情。**可以通过同步来保证对象被安全地发布。除此之外也可以使用一种更加轻量级的 volatile 变量。**

**2 volatile : 用**来确保将变量的更新操作通知到其他线程，可以保证内存中的数据可见。可以将 volatile 看做一个轻量级的锁，但是又与锁有些不同：对于多线程，不是一种互斥关系; 能保证变量状态的“原子性操作”

**3 在并发编程原子性，可见性，有序性**

**1）.原子性**

原子性：即一个操作或者多个操作 要么全部执行并且执行的过程不会被任何因素打断，要么就都不执行。

一个很经典的例子就是银行账户转账问题：

比如从账户A向账户B转1000元，那么必然包括2个操作：从账户A减去1000元，往账户B加上1000元。

试想一下，如果这2个操作不具备原子性，会造成什么样的后果。假如从账户A减去1000元之后，操作突然中止。然后又从B取出了500元，取出500元之后，再执行 往账户B加上1000元 的操作。这样就会导致账户A虽然减去了1000元，但是账户B没有收到这个转过来的1000元。

**Java内存模型只保证了基本读取和赋值是原子性操作，如果要实现更大范围操作的原子性，可以通过synchronized和Lock来实现。由于synchronized和Lock能够保证任一时刻只有一个线程执行该代码块，那么自然就不存在原子性问题了，从而保证了原子性。**

**2）可见性**

可见性是指当多个线程访问同一个变量时，一个线程修改了这个变量的值，其他线程能够立即看得到修改的值。

举个简单的例子，看下面这段代码：

//线程1执行的代码

int i = 0;

i = 10;

//线程2执行的代码

j = i;

假若执行线程1的是CPU1，执行线程2的是CPU2。由上面的分析可知，当线程1执行 i =10这句时，会先把i的初始值加载到CPU1的高速缓存中，然后赋值为10，那么在CPU1的高速缓存当中i的值变为10了，却没有立即写入到主存当中。

此时线程2执行 j = i，它会先去主存读取i的值并加载到CPU2的缓存当中，注意此时内存当中i的值还是0，那么就会使得j的值为0，而不是10.

这就是可见性问题，线程1对变量i修改了之后，线程2没有立即看到线程1修改的值。

**当一个共享变量被volatile修饰时，它会保证修改的值会立即被更新到主存，当有其他线程需要读取时，它会去内存中读取新值。而普通的共享变量不能保证可见性，因为普通共享变量被修改之后，什么时候被写入主存是不确定的，当其他线程去读取时此时内存中可能还是原来的旧值，因此无法保证可见性。另外，通过synchronized和Lock也能够保证可见性，synchronized和Lock能保证同一时刻只有一个线程获取锁然后执行同步代码，并且在释放锁之前会将对变量的修改刷新到主存当中。因此可以保证可见性。**

**3）.有序性**

有序性：即程序执行的顺序按照代码的先后顺序执行。举个简单的例子，看下面这段代码：

int i = 0;

boolean flag = false;

i = 1; //语句1

flag = true; //语句2

上面代码定义了一个int型变量，定义了一个boolean类型变量，然后分别对两个变量进行赋值操作。从代码顺序上看，语句1是在语句2前面的，那么JVM在真正执行这段代码的时候会保证语句1一定会在语句2前面执行吗？不一定，为什么呢？这里可能会发生指令重排序（Instruction Reorder）。

下面解释一下什么是**指令重排序**，一般来说，处理器为了提高程序运行效率，可能会对输入代码进行优化，它不保证程序中各个语句的执行先后顺序同代码中的顺序一致，但是它会保证程序最终执行结果和代码顺序执行的结果是一致的。

比如上面的代码中，语句1和语句2谁先执行对最终的程序结果并没有影响，那么就有可能在执行过程中，语句2先执行而语句1后执行。

但是要注意，虽然处理器会对指令进行重排序，但是它会保证程序最终结果会和代码顺序执行结果相同，那么它靠什么保证的呢？再看下面一个例子：

int a = 10; //语句1

int r = 2; //语句2

a = a + 3; //语句3

r = a\*a; //语句4

这段代码有4个语句，那么可能的一个执行顺序是：

那么可不可能是这个执行顺序呢： 语句2 语句1 语句4 语句3

不可能，因为处理器在进行重排序时是会考虑指令之间的数据依赖性，如果一个指令Instruction 2必须用到Instruction 1的结果，那么处理器会保证Instruction 1会在Instruction 2之前执行。

虽然重排序不会影响单个线程内程序执行的结果，但是多线程呢？下面看一个例子：

//线程1:

context = loadContext(); //语句1

inited = true; //语句2

//线程2:

while(!inited ){

sleep()

}

doSomethingwithconfig(context);

上面代码中，由于语句1和语句2没有数据依赖性，因此可能会被重排序。假如发生了重排序，在线程1执行过程中先执行语句2，而此是线程2会以为初始化工作已经完成，那么就会跳出while循环，去执行doSomethingwithconfig(context)方法，而此时context并没有被初始化，就会导致程序出错。

**从上面可以看出，指令重排序不会影响单个线程的执行，但是会影响到线程并发执行的正确性。也就是说，要想并发程序正确地执行，必须要保证原子性、可见性以及有序性。只要有一个没有被保证，就有可能会导致程序运行不正确。**

x = 2;        //语句1

y = 0;        //语句2

flag = true;  //语句3

x = 4;         //语句4

y = -1;       //语句5

**由于flag变量为volatile变量，那么在进行指令重排序的过程的时候，不会将语句3放到语句1、语句2前面，也不会讲语句3放到语句4、语句5后面。但是要注意语句1和语句2的顺序、语句4和语句5的顺序是不作任何保证的。**

**Volatile可以保证可见性（立即写入内存通知其他线程）和有序性（阻止cpu进行指令重排，但是仅仅针对该变量），但是没有原子性 。**

**使用场景：**

**1.状态标记量**

volatile boolean flag = false;

while(!flag){

doSomething();

}

public void setFlag() {

flag = true;

volatile boolean inited = false;

//线程1:

context = loadContext();

inited = true;

//线程2:

while(!inited ){

sleep()

}

doSomethingwithconfig(context);

**2.double check**

class Singleton{

private volatile static Singleton instance = null;

private Singleton() {

}

public static Singleton getInstance() {

if(instance==null) {

synchronized (Singleton.class) {

if(instance==null)

instance = new Singleton();

}

}

return instance;

}

}

**BlockingQuence**

## 一、基础知识

**1 Queue**

1）当Queue为空时，获取元素线程被阻塞直到Queue变为非空；

2）当Queue满时，添加元素线程被阻塞直到Queue不满。BlockingQueue不允许元素为null，如果入队一个null元素，会抛NullPointerException。常用于生产者消费者模式。

**2 BlockingQueue对于不能满足条件的操作，提供了四种处理方式：**

1）直接抛异常，抛出异常。如果队列已满，添加元素会抛出IllegalStateException异常；如果队列为空，获取元素会抛出NoSuchElementException异常；

2）返回一个特殊值（null或false）；

3）在满足条件之前，无限期的阻塞当前线程，当队列满足条件或响应中断退出；

4）在有限时间内阻塞当前线程，超时后返回失败。

**3 JDK提供的阻塞队列：**

**ArrayBlockingQueue：一个由数组结构组成的有界阻塞队列，遵循FIFO原则。**

**LinkedBlockingQueue：一个由链表结构组成的有界阻塞队列，遵循FIFO原则，默认和最大长度为Integer.MAX\_VALUE。**

**PriorityBlockingQueue：一个支持优先级排序的无界阻塞队列。**

**DelayQueue：一个使用优先级队列实现的无界阻塞队列。**

**SynchronousQueue：一个不存储元素的阻塞队列。**

**LinkedTransferQueue：一个由链表结构组成的无界阻塞队列。**

**LinkedBlockingDeque：一个由链表结构组成的双向阻塞队列。**

**4 当队列满时，生产者会一直阻塞，**当消费者从队列中取出元素时，如何通知生产者队列可以继续，以ArrayBlockingQueue和LinkedBlockingQueue为例，

分析源代码如何实现阻塞队列。它们的阻塞机制都是基于**Lock和Condition**实现，其中LinkedBlockingQueue还用到了原子变量类。

**5 LinkedBlockingQueue**

**问题一：为什么使用两把锁？**

为了提高并发度和吞吐量，使用两把锁，takeLock只负责出队，putLock只负责入队，入队和出队可以同时进行，提高入队和出队操作的效率，增大队列的吞吐量。LinkedBlockingQueue队列的吞吐量通常要高于ArrayBlockingQueue队列，但是在高并发条件下可预测性降低。

**问题二：ArrayBlockingQueue中的count是一个普通的int型变量，LinkedBlockingQueue的count为什么是AtomicInteger类型的？**

因为ArrayBlockingQueue的入队和出队操作使用同一把锁，对count的修改都是在处于线程获取锁的情况下进行操作，因此不会有线程安全问题。而LinkedBlockingQueue的入队和出队操作使用的是不同的锁，会有对count变量并发修改的情况，所以使用原子变量保证线程安全。

**问题三：像notEmpty、takeLock、count域等都声明为final型，final成员变量有什么特点？**

1）对于一个final变量，如果是基本数据类型的变量，则其数值一旦在初始化之后便不能更改；如果是引用类型的变量，则在对其初始化之后便不能再让其指向另一个对象。

2）对于一个final成员变量，必须在定义时或者构造器中进行初始化赋值，而且final变量一旦被初始化赋值之后，就不能再被赋值了。只要对象是正确构造的（被构造对象的引用在构造函数中没有“逸出”），那么不需要使用同步（指lock和volatile的使用）就可以保证任意线程都能看到这个final域在构造函数中被初始化之后的值。

**LocalThread**

## ThreadLocal含义

**1 首先ThreadLocal和本地线程没有一毛钱关系**，更不是一个特殊的Thread，它只是一个线程的局部变量(其实就是一个Map),ThreadLocal会为每个使用该变量的线程提供独立的变量副本，所以每一个线程都可以独立地改变自己的副本，而不会影响其它线程所对应的副本。这样做其实就是以空间换时间的方式(与synchronized相反)，以耗费内存为代价，单大大减少了线程同步(如synchronized)所带来性能消耗以及减少了线程并发控制的复杂度。

**2 多个模块在同一个线程中运行时要共享同一份数据**，实现线程范围内的数据共享可以用上一节中所用的方法。JDK1.5提供了ThreadLocal类来方便实现线程范围内的数据共享，它的作用就相当于上一节中的Map。ThreadLocal的作用和目的：**用于实现线程内的数据共享，即对于相同的程序代码，多个模块在同一个线程中运行时要共享一份数据，而在另外线程中运行时又共享另外一份数据。**

每个线程调用全局ThreadLocal对象的set方法，就相当于往其内部的map集合中增加一条记录，key就是各自的线程，value就是各自的set方法传进去的值。

在线程结束时可以调用ThreadLocal.clear()方法用来更快释放内存，也可以不调用，因为线程结束后也可以自动释放相关的ThreadLocal变量。

一个ThreadLocal对象只能记录一个线程内部的一个共享变量，需要记录多个共享数据，可以创建多个ThreadLocal对象，或者将这些数据进行封装，将封装后的数据对象存入ThreadLocal对象中。

**3 在同步机制中，通过对象的锁机制保证同一时间只有一个线程访问变量。**这时该变量是多个线程共享的，使用同步机制要求程序慎密地分析什么时候对变量进行读写，什么时候需要锁定某个对象，什么时候释放对象锁等繁杂的问题，程序设计和编写难度相对较大。

**4 而ThreadLocal则从另一个角度来解决多线程的并发访问。**ThreadLocal会为每一个线程提供一个独立的变量副本，从而隔离了多个线程对数据的访问冲突。因为每一个线程都拥有自己的变量副本，从而也就没有必要对该变量进行同步了。ThreadLocal提供了线程安全的共享对象，在编写多线程代码时，可以把不安全的变量封装进ThreadLocal。

**5 概括起来说，对于多线程资源共享的问题，同步机制采用了“以时间换空间”的方式，而ThreadLocal采用了“以空间换时间”的方式。**前者仅提供一份变量，让不同的线程排队访问，而后者为每一个线程都提供了一份变量，因此可以同时访问而互不影响

**6 Spring:** 只有无状态的Bean才可以在多线程环境下共享，在Spring中，绝大部分Bean都可以声明为singleton作用域。就是因为Spring对一些Bean（如**RequestContextHolder、TransactionSynchronizationManager、LocaleContextHolder**等）中非线程安全状态采用ThreadLocal进行处理，让它们也成为线程安全的状态，因为有状态的Bean就可以在多线程中共享了。

**7 线程安全问题都是由全局变量及静态变量引起的**

SimpleDateFormate sdf = new SimpleDateFormat();

使用sdf.parse(dateStr);sdf.format(date);

在sdf内有一个对Caleadar对象的引用，在源码sdf.parse(dateStr);源码中calendar.clear();和calendar.getTime(); // 获取calendar的时间

如果 线程A 调用了 sdf.parse(), 并且进行了 calendar.clear()后还未执行calendar.getTime()的时候,线程B又调用了sdf.parse(), 这时候线程B也执行了sdf.clear()方法, 这样就导致线程A的的calendar数据被清空了;

**8 ThreadLocal是使用空间换时间，synchronized是使用时间换空间**

**9 比如说有一个流程业务类ProcessBusiness是单例的，所有并发公用一个pb实例，要使用同步方法，用锁的形式。ThreadLocal可以封装一个pb实例跟着线程走。**

**注意：只能局限在各自使用的层级，如果涉及多个线程共同卖票不能够这么做。**

**闭锁、栅栏、交换机、信号量**

**1.闭锁(CountDownLatch)**

通过一个计数器实现，原理就是不停地检查join线程是否存活，如果存活则一直等待，作用是m(m>=1)个线程等待其他的线程执行玩之后然后继续往下执行，**类似join()方法，但功能更强大**，可以操作更多的线程。当计数器的值减少为0的时候,Waiter线程就能够不再阻塞，继续往下运行了。**计数器初始值设置为3，调用await的地方需要等待0的时候执行。原则是多个线程公用一个 CountDown。**

public class TestTemp {

public static void main(String[] args) throws InterruptedException {

CountDownLatch latch = new CountDownLatch(3);//计数器初始值设置为3

Waiter waiter = new Waiter(latch);

Decrementer decrementer = new Decrementer(latch);

new Thread(waiter)

.start();

new Thread(decrementer).start();

Thread.sleep(4000);

}

}

class Waiter implements Runnable{

CountDownLatch latch = null;

public Waiter(CountDownLatch latch) {

this.latch = latch;

}

public void run() {

try {

latch.await();//阻塞在这边

} catch (InterruptedException e) {

e.printStackTrace();

}

System.out.println("Waiter Released");

}

}

class Decrementer implements Runnable {

CountDownLatch latch = null;

public Decrementer(CountDownLatch latch) {

this.latch = latch;}

public void run() {

try {

Thread.sleep(1000);

this.latch.countDown();//计数器减1

Thread.sleep(1000);

this.latch.countDown();//计数器减1

Thread.sleep(1000);

this.latch.countDown();//计数器减1

} catch (InterruptedException e) {

e.printStackTrace();

}

}

}

**2.栅栏（CyclicBarrier ）**

使用场景:所有线程相互等待，直到所有线程都到达某一点时才打开栅栏，然后线程继续执行。可以看到触发的条件仅仅和两个线程本身有关，而闭锁和外部的条件(计数器)来触发，它们的触发条件是不同的。

package com.hiya.concurrent.cyclicBarrier;

public class ActionAdd implements Runnable

{

@Override

public void run()

{

System.out.println("ActionAdd......");

}

}

package com.hiya.concurrent.cyclicBarrier;

public class ActionDelete implements Runnable

{

@Override

public void run()

{

System.out.println("ActionDelete......");

}

}

package com.hiya.concurrent.cyclicBarrier;

public class ActionUpdate implements Runnable

{

@Override

public void run()

{

System.out.println("ActionUpdate......");

}

}

package com.hiya.concurrent.cyclicBarrier;

import java.util.concurrent.CyclicBarrier;

public class CyclicBarrierClient

{

public static void main(String[] args) throws InterruptedException

{

Runnable actionAdd = new ActionAdd();

Runnable actionUpdate = new ActionUpdate();

Runnable actionDelete = new ActionDelete();

CyclicBarrier barrier1 = new CyclicBarrier(3, actionAdd);// 有3个线程等待就执行actionAdd

CyclicBarrier barrier2 = new CyclicBarrier(2, actionUpdate);// 有2个线程等待就执行actionUpdate

CyclicBarrier barrier3 = new CyclicBarrier(1, actionDelete);// 有2个线程等待就执行actionUpdate

CyclicBarrierRunnable barrierRunnable1 = new CyclicBarrierRunnable(barrier1, barrier2);

CyclicBarrierRunnable barrierRunnable2 = new CyclicBarrierRunnable(barrier1, barrier2);

CyclicBarrierRunnable barrierRunnable3 = new CyclicBarrierRunnable(barrier1, barrier3);

new Thread(barrierRunnable1).start();

new Thread(barrierRunnable2).start();

new Thread(barrierRunnable3).start();

}

}

package com.hiya.concurrent.cyclicBarrier;

import java.util.concurrent.BrokenBarrierException;

import java.util.concurrent.CyclicBarrier;

public class CyclicBarrierRunnable implements Runnable

{

CyclicBarrier barrier1 = null;

CyclicBarrier barrier2 = null;

public CyclicBarrierRunnable(CyclicBarrier barrier1, CyclicBarrier barrier2)

{

this.barrier1 = barrier1;

this.barrier2 = barrier2;

}

public void run()

{

try

{

Thread.sleep(1000);

System.out.println(Thread.currentThread().getName() + " waiting at barrier 1");

this.barrier1.await();// 等在这里呢

Thread.sleep(1000);

System.out.println(Thread.currentThread().getName() + " waiting at barrier 2");

this.barrier2.await();

System.out.println(Thread.currentThread().getName() + " done!");

} catch (InterruptedException e)

{

e.printStackTrace();

} catch (BrokenBarrierException e)

{

e.printStackTrace();

}

}

}

**3.交换机**

使用场景:用户线程之间的数据交互,Exchanger类源于java.util.concurrent包，它可以在两个线程之间传输数据，Exchanger中的public V exchange(V x)方法被调用后等待另一个线程到达交换点（如果当前线程没有被中断），然后将已知的对象传给它，返回接收的对象。

如果另外一个线程已经在交换点等待，那么恢复线程计划并接收通过当前线程传给的对象

**两个线程的值已经发生了交换。**

package com.hiya.concurrent.exchanger;

import java.util.concurrent.Exchanger;

public class BikeExchangerRunnable implements Runnable

{

private Exchanger<String> exchanger;

public BikeExchangerRunnable(Exchanger<String> exchanger)

{

super();

this.exchanger = exchanger;

}

@Override

public void run()

{

try

{

System.out.println(Thread.currentThread().getName() + "-BikeExchangerRunnable: " + exchanger.exchange("Bike"));

} catch (InterruptedException e)

{

e.printStackTrace();

}

}

}

package com.hiya.concurrent.exchanger;

import java.util.concurrent.Exchanger;

public class CarExchangerRunnable implements Runnable

{

private Exchanger<String> exchanger;

public CarExchangerRunnable(Exchanger<String> exchanger)

{

super();

this.exchanger = exchanger;

}

@Override

public void run()

{

try

{

System.out.println(Thread.currentThread().getName() + "-CarExchangerRunnable: " + exchanger.exchange("Car"));

} catch (InterruptedException e)

{

e.printStackTrace();

}

}

}

package com.hiya.concurrent.exchanger;

import java.util.concurrent.Exchanger;

public class ExchangerClient

{

public static void main(String[] args) throws InterruptedException

{

Exchanger<String> exchanger = new Exchanger<>();

CarExchangerRunnable car = new CarExchangerRunnable(exchanger);

BikeExchangerRunnable bike = new BikeExchangerRunnable(exchanger);

new Thread(car).start();

new Thread(bike).start();

System.out.println("Main end!");

}

}

**4.信号量**

1. **保护一个重要(代码)部分防止一次超过 N 个线程进入**。

2. 在两个线程之间发送信号。同一个时刻，只有N个线程可以执行acquire和release部分的代码

package com.hiya.concurrent.semaphore;

public class SemaphoreClient

{

public static void main(String[] args) throws InterruptedException

{

Service service = new Service();

ThreadA a = new ThreadA(service);

a.setName("A");

ThreadB b = new ThreadB(service);

b.setName("B");

ThreadC c = new ThreadC(service);

c.setName("C");

a.start();

b.start();

c.start();

}

}

package com.hiya.concurrent.semaphore;

import java.util.concurrent.Semaphore;

public class Service

{

private Semaphore semaphore = new Semaphore(1);// 同一个时刻，只有一个线程可以执行acquire和release部分的代码

public void testMethod()

{

try

{

semaphore.acquire();

System.out.println(Thread.currentThread().getName() + " begin timer=" + System.currentTimeMillis());

Thread.sleep(5000);

System.out.println(Thread.currentThread().getName() + " end timer=" + System.currentTimeMillis());

semaphore.release();

} catch (InterruptedException e)

{

e.printStackTrace();

}

}

}

package com.hiya.concurrent.semaphore;

public class ThreadA extends Thread

{

private Service service;

public ThreadA(Service service)

{

super();

this.service = service;

}

@Override

public void run()

{

service.testMethod();

}

}

package com.hiya.concurrent.semaphore;

public class ThreadB extends Thread

{

private Service service;

public ThreadB(Service service)

{

super();

this.service = service;

}

@Override

public void run()

{

service.testMethod();

}

}

package com.hiya.concurrent.semaphore;

public class ThreadC extends Thread

{

private Service service;

public ThreadC(Service service)

{

super();

this.service = service;

}

@Override

public void run()

{

service.testMethod();

}

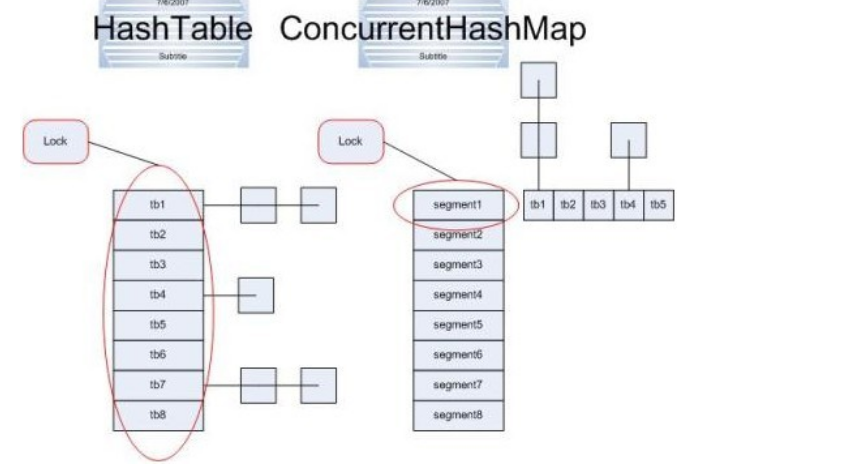
}

**ConcurrentHashMap**

## 一、ConcurrentHashMap概述

**1 HashMap 线程不安全，Hashtable 内部采用独占锁，线程安全，但效率低。**

ConcurrentHashMap同步容器类是java5 新增的一个线程安全的哈希表，效率介于HashMap和Hashtable之间。**内部采用“锁分段”机制。**



**2 并发编程实践中，ConcurrentHashMap是一个经常被使用的数据结构**，相比于Hashtable以及Collections.synchronizedMap()，ConcurrentHashMap在线程安全的基础上提供了更好的写**并发能力，但同时降低了对读一致性的要求**。设计与实现非常精巧，大量的利用了volatile，final，CAS等lock-free技术来减少锁竞争对于性能的影响。**诸如get/put/remove操作只需要锁着需要的单个桶即可。ConcurrentHashMap只有在size等操作的时候才会锁住整个Hash表。**

**3 java.util.concurrent 包还提供了设计用于多线程上下文中的Collection实现：**

当期望许多线程访问一个给定 collection 时，

ConcurrentHashMap 通常优于同步的 HashMap，

ConcurrentSkipListMap 通常优于同步的 TreeMap

ConcurrentSkipListSet通常优于同步的 TreeSet.

**4 当期望的读数和遍历远远大于列表的更新数时，**

CopyOnWriteArrayList 优于同步的 ArrayList。因为每次添加时都会进行复制，开销非常的大，并发迭代操作多时。

**5 ConcurrentHashMap采用了分段锁的设计**，只有在同一个分段内才存在竞态关系，不同的分段锁之间没有锁竞争。相比于对整个Map加锁的设计，分段锁大大的提高了高并发环境下的处理能力。但同时，由于不是对整个Map加锁，导致一些需要扫描整个Map的方法（如size(), containsValue()）需要使用特殊的实现，另外一些方法（如clear()）甚至放弃了对一致性的要求

**6 ConcurrentHashMap中的分段锁称为Segment**，它即类似于HashMap的结构，即内部拥有一个Entry数组，数组中的每个元素又是一个链表；同时又是一个ReentrantLock（**Segment继承了ReentrantLock**）。ConcurrentHashMap中的HashEntry相对于HashMap中的Entry有一定的差异性：HashEntry中的value以及next都被volatile修饰，这样在多线程读写过程中能够保持它们的可见性，代码如下：

static final class HashEntry<K,V> {

final int hash;

final K key;

volatile V value;

volatile HashEntry<K,V> next;

**7 并发度**

可以理解为程序运行时能够同时更新ConccurentHashMap且不产生锁竞争的最大线程数，实际上就是ConcurrentHashMap中的分段锁个数，即Segment[]的数组长度。ConcurrentHashMap默认的并发度为16，

但用户也可以在构造函数中设置并发度。当用户设置并发度时，ConcurrentHashMap会使用大于等于该值的最小2幂指数作为实际并发度（假如用户设置并发度为17，实际并发度则为32）。

如果并发度设置的过小，会带来严重的锁竞争问题；如果并发度设置的过大，原本位于同一个Segment内的访问会扩散到不同的Segment中CPUcache命中率会下降，从而引起程序性能下降。在hashMap的基础上，ConcurrentHashMap将数据分为多个segment，默认16个（concurrency level），然后每次操作对一个segment加锁，避免多线程锁得几率，提高并发效率。

**8 线程不安全的HashMap**

因为多线程环境下，使用Hashmap进行put操作会引起死循环，导致CPU利用率接近100%，所以在并发情况下不能使用HashMap。

**9 效率低下的HashTable容器**

HashTable容器使用synchronized来保证线程安全，但在线程竞争激烈的情况下HashTable的效率非常低下。因为当一个线程访问HashTable的同步方法时，其他线程访问HashTable的同步方法时，可能会进入阻塞或轮询状态。如线程1使用put进行添加元素，线程2不但不能使用put方法添加元素，并且也不能使用get方法来获取元素，所以竞争越激烈效率越低。

**10 锁分段技术**

HashTable容器在竞争激烈的并发环境下表现出效率低下的原因，是因为所有访问HashTable的线程都必须竞争同一把锁，那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，从而可以有效的提高并发访问效率，这就是ConcurrentHashMap所使用的锁分段技术，首先将数据分成一段一段的存储，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。有些方法需要跨段，比如size()和containsValue()，它们可能需要锁定整个表而而不仅仅是某个段，这需要按顺序锁定所有段，操作完毕后，又按顺序释放所有段的锁。这里“按顺序”是很重要的，否则极有可能出现死锁，在ConcurrentHashMap内部，段数组是final的，并且其成员变量实际上也是final的，但是，仅仅是将数组声明为final的并不保证数组成员也是final的，这需要实现上的保证。这可以确保不会出现死锁，因为获得锁的顺序是固定的。

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守护者一个HashEntry数组里的元素,当对HashEntry数组的数据进行修改时，必须首先获得它对应的Segment锁。

**11 CopyOnWriteArrayList确实是线程安全的**

运行上面的代码,没有报出

java.util.ConcurrentModificationException

CopyOnWriteArrayList使用了一种叫**写时复制**的方法，当有新元素添加到CopyOnWriteArrayList时，先从原有的数组中拷贝一份出来，然后在新的数组做写操作，写完之后，再将原来的数组引用指向到新数组。

**12 CopyOnWriteArrayList的整个add操作都是在锁的保护下进行的。**

这样做是为了避免在多线程并发add的时候，复制出多个副本出来,把数据搞乱了，导致最终的数组数据不是我们期望的。

由于所有的写操作都是在新数组进行的，这个时候如果有线程并发的写，则通过锁来控制，如果有线程并发的读，则分几种情况，可见CopyOnWriteArrayList的读操作是可以不用加锁的。

**1）、如果写操作未完成，那么直接读取原数组的数据；**

**2）、如果写操作完成，但是引用还未指向新数组，那么也是读取原数组数据；**

**3）、如果写操作完成，并且引用已经指向了新的数组，那么直接从新数组中读取数据**

**13 CopyOnWriteArrayList 合适读多写少的场景，不过这类慎用**

1、由于写操作的时候，需要拷贝数组，会消耗内存，如果原数组的内容比较多的情况下，可能导致young gc或者full gc

2、不能用于实时读的场景，像拷贝数组、新增元素都需要时间，所以调用一个set操作后，读取到数据可能还是旧的,虽然CopyOnWriteArrayList 能做到最终一致性,但是还是没法满足实时性要求；

**ExecutorService线程池**

一、Java通过Executors提供四种线程池  
**1 newCachedThreadPool**创建一个可缓存线程池，如果线程池长度超过处理需要，可灵活回收空闲线程，若无可回收，则新建线程。  
**2 newFixedThreadPool** 创建一个定长线程池，可控制线程最大并发数，超出的线程会在队列中等待。  
**3 newScheduledThreadPool** 创建一个定长线程池，支持定时及周期性任务执行。  
**4 newSingleThreadExecutor** 创建一个单线程化的线程池，它只会用唯一的工作线程来执行任务，保证所有任务按照指定顺序(FIFO, LIFO, 优先级)执行。

**5 注意：线程池只是为了控制应用中处理某项业务中防止高并发问题带来的线程不安全的发生的概率。在我目前的测试用，还没有发现线程可以重用这个概念，因为线程开启后，用完就关闭了，不可以再次开启的，查看源码发现会每次新创建一个线程用来处理业务。我们可以通过线程池指定处理这项业务最大的同步线程数，比如：Executors.newFixedThreadPool(3);在线程池中保持三个线程可以同时执行，但是注意，并不是说线程池中永远都是这三个线程，只是说可以同时存在的线程数，当某个线程执行结束后，会有新的线程进来。newFixedThreadPool.execute(new ThreadForpools());这句话的含义并不是添加新的线程，而是添加新的处理业务请求进来。至少我当前是这么理解的，没有发现线程可以重复使用。**

**6 代码解释**

\* 一、线程池：提供了一个线程队列，队列中保存着所有等待状态的线程。避免了创建与销毁额外开销，提高了响应的速度。

\* 二、线程池的体系结构：

\* java.util.concurrent.Executor:负责线程的使用与调度的根接口

\* |--\*\*ExecutorService 子接口：线程池的主要接口

\* |--ThreadPoolExecutor 线程池的实现类

\* |--ScheduledExecutorService 子接口：负责线程的调度

\* |--ScheduledThreadPoolExecutor:继承ThreadPoolExecutor,实现ScheduledExecutorService接口

\* 三、工具类：Executors

\* 方法有：

\* ExecutorService newFixedThreadPool(): 创建固定大小的线程池

\* ExecutorService newCachedThreadPool():缓存线程池，线程池的数量不固定，可以根据需要自动的更改数量。

\* ExecutorService newSingleThreadExecutor():创建单个线程池。线程池中只有一个线程

\* ScheduledExecutorService newScheduledThreadPool():创建固定大小的线程，可以延迟或定时的执行任务

**Atomic和CAS原语**

## CAS (Compare-And-Swap)

**1 一种底层硬件对并发的支持**，针对多处理器操作而设计的处理器中的一种特殊指令，用于管理对共享数据的并发访问。一种**无锁的非阻塞算法**的实现。**相当于版本控制的乐观锁**。 java.util.concurrent包借助CAS实现了区别于synchronized同步锁的一种乐观锁。乐观锁就是每次去取数据的时候都乐观的认为数据不会被修改，所以不会上锁，但是在更新的时候会判断一下在此期间数据有没有更新。CAS有3个操作数：内存值V，旧的预期值A，要修改的新值B。当且仅当预期值A和内存值V相同时，将内存值V修改为B，否则什么都不做。CAS的关键点在于，系统在硬件层面保证了比较并交换操作的原子性，处理器使用基于对缓存加锁或总线加锁的方式来实现多处理器之间的原子操作。

**当且仅当V==A时，B的值才更新给A，否则将不做任何操作。**

**2 jdk1.5 后 java.util.concurrent.atomic 类的小工具包**，支持在单个变量上解除锁的线程安全编程，包下提供了常用的原子变量：

- AtomicBoolean 、AtomicInteger 、AtomicLong 、 AtomicReference

- AtomicIntegerArray 、AtomicLongArray

- AtomicMarkableReference

- AtomicReferenceArray

- AtomicStampedReference

**3 在Java语言中，i++这类的操作不是原子操作**，并非是线程安全的，使用的时候需要用到synchronized关键字进行同步。**而AtomicInteger则通过一种线程安全的操作接口自动实现同步，不需要再人为的增加同步控制。**

**4 AtomicInteger源码分析**

下面通过AtomicInteger的源码来看一下是怎么在没有锁的情况下保证数据正确性。首先看一下incrementAndGet方法的实现

public final int incrementAndGet() {

for (;;) {

int current = get();

int next = current + 1;

if (compareAndSet(current, next))

return next;

}

}

方法中采用了CAS操作，每次从内存中读取数据然后将此数据和+1后的结果进行CAS操作，如果成功就返回结果，否则重试直到成功为止。而compareAndSet利用JNI来完成CPU指令的操作。整体的过程就是这样子的，**利用CPU的CAS指令，同时借助JNI来完成Java的非阻塞算法。JNI是Java Native Interface的缩写,中文为JAVA本地调用**

**5 cas问题**

1、ABA问题。CAS在操作值的时候检查值是否已经变化，没有变化的情况下才会进行更新。但是如果一个值原来是A，变成B，又变成A，那么CAS进行检查时会认为这个值没有变化，但是实际上却变化了。

ABA问题的解决方法是使用版本号。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一，那么A－B－A 就变成1A-2B－3A。从Java1.5开始JDK的atomic包里提供了一个类AtomicStampedReference来解决ABA问题。

2、并发越高，失败的次数会越多，CAS如果长时间不成功，会极大的增加CPU的开销。因此CAS不适合竞争十分频繁的场景。

3、只能保证一个共享变量的原子操作。当对多个共享变量操作时，CAS就无法保证操作的原子性，这时就可以用锁，或者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Java1.5开始JDK提供了AtomicReference类来保证引用对象的原子性，你可以把多个变量放在一个对象里来进行CAS操作

**同步策略和秒杀系统**

**1. 超发的原因**

假设某个抢购场景中，我们一共只有100个商品，在最后一刻，我们已经消耗了99个商品，仅剩最后一个。这个时候，系统发来多个并发请求，这批请求读取到的商品余量都是99个，然后都通过了这一个余量判断，最终导致超发。

在上面的这个图中，就导致了并发用户B也“抢购成功”，多让一个人获得了商品。这种场景，在高并发的情况下非常容易出现。

**2. 悲观锁思路**

解决线程安全的思路很多，可以从“悲观锁”的方向开始讨论。

悲观锁，也就是在修改数据的时候，采用锁定状态，排斥外部请求的修改。遇到加锁的状态，就必须等待。

虽然上述的方案的确解决了线程安全的问题，但是，别忘记，我们的场景是“高并发”。也就是说，会很多这样的修改请求，每个请求都需要等待“锁”，某些线程可能永远都没有机会抢到这个“锁”，

这种请求就会死在那里。同时，这种请求会很多，瞬间增大系统的平均响应时间，结果是可用连接数被耗尽，系统陷入异常。

**3. FIFO队列思路**

那好，那么我们稍微修改一下上面的场景，我们直接将请求放入队列中的，采用FIFO（First Input First Output，先进先出），这样的话，我们就不会导致某些请求永远获取不到锁。看到这里，是不是有点强行将多线程变成单线程。

然后，我们现在解决了锁的问题，全部请求采用“先进先出”的队列方式来处理。那么新的问题来了，高并发的场景下，因为请求很多，很可能一瞬间将队列内存“撑爆”，然后系统又陷入到了异常状态。或者设计一个极大的内存队列，也是一种方案，但是，系统处理完一个队列内请求的速度根本无法和疯狂涌入队列中的数目相比。也就是说，队列内的请求会越积累越多，最终Web系统平均响应时候还是会大幅下降，系统还是陷入异常。

**4. 乐观锁思路**

这个时候，我们就可以讨论一下“乐观锁”的思路了。乐观锁，是相对于“悲观锁”采用更为宽松的加锁机制，大都是采用带版本号（Version）更新。实现就是，这个数据所有请求都有资格去修改，但会获得一个该数据的版本号，只有版本号符合的才能更新成功，其他的返回抢购失败。这样的话，我们就不需要考虑队列的问题，不过，它会增大CPU的计算开销。但是，综合来说，这是一个比较好的解决方案。

有很多软件和服务都“乐观锁”功能的支持，例如Redis中的watch就是其中之一。通过这个实现，我们保证了数据的安全。

1. **concurrentHashMap思路**

保证了同步和效率。

package com.hiya.concurrent.safety.method;

import java.util.Date;

import java.util.Iterator;

import java.util.Map;

import java.util.Map.Entry;

import java.util.Set;

import java.util.concurrent.atomic.AtomicInteger;

import java.util.concurrent.locks.Lock;

import java.util.concurrent.locks.ReentrantLock;

public class ThreadSafety

{

/\*\*

\* 普通的线程不安全，不能保证原子性，可见性

\* 几个线程分别进行了一次自增操作后，inc只增加了1

\* 由于不能保证可见性，所以数值比incVolatile要小

\*/

public int incCommon = 0;

Lock lock = new ReentrantLock();

public AtomicInteger incAtomic = new AtomicInteger(0);

/\*\*

\* 假如某个时刻变量incVolatile的值为10，

线程1对变量进行自增操作，线程1先读取了变量inc的原始值，然后线程1被阻塞了；

然后线程2对变量进行自增操作，线程2也去读取变量inc的原始值，由于线程1只是对变量inc进行读取操作，而没有对变量进行修改操作，所以不会导致线程2的工作内存中缓存变量inc的缓存行无效，

所以线程2会直接去主存读取inc的值，发现inc的值时10，然后进行加1操作，并把11写入工作内存，最后写入主存。

然后线程1接着进行加1操作，由于已经读取了inc的值，注意此时在线程1的工作内存中inc的值仍然为10，所以线程1对inc进行加1操作后inc的值为11，然后将11写入工作内存，最后写入主存。

那么两个线程分别进行了一次自增操作后，inc只增加了1。

解释到这里，可能有朋友会有疑问，不对啊，前面不是保证一个变量在修改volatile变量时，会让缓存行无效吗？然后其他线程去读就会读到新的值，对，这个没错。这个就是上面的happens-before

规则中的volatile变量规则，但是要注意，线程1对变量进行读取操作之后，被阻塞了的话，并没有对inc值进行修改。然后虽然volatile能保证线程2对变量inc的值读取是从内存中读取的，但是线程1没

有进行修改，所以线程2根本就不会看到修改的值。

根源就在这里，自增操作不是原子性操作，而且volatile也无法保证对变量的任何操作都是原子性的。

\*/

public volatile int incVolatile= 0;

/\*\*

\* 线程不安全

\*/

public void increase()

{

incCommon++;

}

/\*\*

\* 保证可见性（立即写入内存通知其他线程）和有序性（阻止cpu进行指令重排，但是仅仅针对该变量），但是没有原子性

\*/

public void increaseVolatile()

{

incVolatile++;

}

/\*\*

\* 线程安全，传统办法

\*/

public synchronized void increaseSynchronized()

{

incCommon++;

}

/\*\*

\* 线程安全，lock方法，速度快一倍

\*/

public void increaseLock()

{

lock.lock();

try

{

incCommon++;

}

finally

{

lock.unlock();

}

}

/\*\*

\* 在java 1.5的java.util.concurrent.atomic包下提供了一些原子操作类，即对基本数据类型的 自增（加1操作），自减（减1操作）、以及加法操作（加一个数），

\* 减法操作（减一个数）进行了封装，保证这些操作是原子性操作。atomic是利用CAS来实现原子性操作的（Compare And Swap），CAS实际上是利用处理器提供的

\* CMPXCHG指令实现的，而处理器执行CMPXCHG指令是一个原子性操作。

\*/

public void increaseAtomic()

{

incAtomic.getAndIncrement();

}

public static void doBusiness(String type)

{

System.out.println("Begin time:"+new Date(System.currentTimeMillis()));

final ThreadSafety test = new ThreadSafety();

for (int i = 0; i < 10; i++)

{

new Thread()

{

public void run()

{

for (int j = 0; j < 10000000; j++)

{

if("Common".equals(type))

{

test.increase();

}

else if("Volatile".equals(type))

{

test.increaseVolatile();

}

else if("Synchronized".equals(type))

{

test.increaseSynchronized();

}

else if("Lock".equals(type))

{

test.increaseLock();

}

else if("Atomic".equals(type))

{

test.increaseAtomic();

}

else

{

test.increase();

}

}

};

}.start();

}

while (true)

{

int activeCount = Thread.activeCount() ;

//System.out.println("activeCount="+activeCount );

if(activeCount >1)

{

Map<Thread, StackTraceElement[]> map = Thread.getAllStackTraces();

Set<Entry<Thread, StackTraceElement[]>> entry = map.entrySet();

Iterator<Map.Entry<Thread, StackTraceElement[]>> ite = entry.iterator();

while(ite.hasNext())

{

Map.Entry<Thread, StackTraceElement[]> en = ite.next();

Thread key = en.getKey();

System.out.println("当前线程："+key.getName() );

}

}

//只有主线程 main

if(activeCount == 1)

{

System.out.println("incCommon="+test.incCommon);

System.out.println("incVolatile="+test.incVolatile);

System.out.println("incAtomic="+test.incAtomic);

System.out.println("End time:"+new Date(System.currentTimeMillis()));

break;

}

}

}

}

**ReentrantLock和Condition**

## 一、lock 概述

**1 java.util.concurrent.lock 中的 Lock 框架是锁定的一个抽象**，它允许把锁定的实现作为 Java 类，而不是作为语言的特性来实现。这就为 Lock 的多种实现留下了空间，各种实现可能有不同的调度算法、性能特性或者锁定语义。 ReentrantLock 类实现了 Lock ，**它拥有与 synchronized 相同的并发性和内存语义**，**但是添加了类似锁投票、定时锁等候和可中断锁等候的一些特性**。此外，它还提供了在激烈争用情况下更佳的性能。（换句话说，当许多线程都想访问共享资源时，JVM 可以花更少的时候来调度线程，把更多时间用在执行线程上。**）lock 必须在 finally 块中释放。否则，如果受保护的代码将抛出异常，锁就有可能永远得不到释放！**

**2 reentrant 锁意味着什么呢**？简单来说，它有一个与锁相关的获取计数器，如果拥有锁的某个线程再次得到锁，那么获取计数器就加1，然后锁需要被释放两次才能获得真正释放。

**3 ReentrantLock与synchronized的比较**

@相同：ReentrantLock提供了synchronized类似的功能和内存语义。

@不同：

1. ReentrantLock功能性方面更全面，比如时间锁等候，可中断锁等候，锁投票等，因此更有扩展性。在多个条件变量和高度竞争锁的地方，用ReentrantLock更合适，ReentrantLock还提供了Condition，对线程的等待和唤醒等操作更加灵活，一个ReentrantLock可以有多个Condition实例，所以更有扩展性。

（2）ReentrantLock 的性能比synchronized会好点，同样的测试，时间缩短一半。

（3）ReentrantLock提供了可轮询的锁请求，他可以尝试的去取得锁，如果取得成功则继续处理，取得不成功，可以等下次运行的时候处理，tryLock所以不容易产生死锁，而synchronized则一旦进入锁请求要么成功，要么一直阻塞，所以更容易产生死锁。

**4 在内部锁中，死锁是致命的——唯一的恢复方法是重新启动程序**，唯一的预防方法是在构建程序时不要出错。**而可轮询的锁获取模式具有更完善的错误恢复机制，可以规避死锁的发生。 如果你不能获得所有需要的锁，那么使用可轮询的获取方式使你能够重新拿到控制权，它会释放你已经获得的这些锁，然后再重新尝试。**可轮询的锁获取模式，由tryLock()方法实现。此方法仅在调用时锁为空闲状态才获取该锁。如果锁可用，则获取锁，并立即返回值true。如果锁不可用，则此方法将立即返回值false。

**tryLock(long, TimeUnit) 实现可定时的锁请求**

**lockInterruptibly()方法能够使你获得锁的时候响应中断**

Lock lock = ...;

if (lock.tryLock()) {

try {

} finally {

lock.unlock();

}

} else {

}

**5 Condition条件变量是为了解决Object.wait/notify/notifyAll难以使用的问题**

条件变量需要与锁绑定，而且多个Condition需要绑定到同一锁上。获取一个条件变量的方法是Lock.newCondition()。

void await() throws InterruptedException;

void awaitUninterruptibly();

long awaitNanos(long nanosTimeout) throws InterruptedException;

boolean await(long time, TimeUnit unit) throws InterruptedException;

boolean awaitUntil(Date deadline) throws InterruptedException;

void signal();

void signalAll();

以上是Condition接口定义的方法，await\*对应于Object.wait，signal对应于Object.notify，signalAll对应于Object.notifyAll。特别说明的是Condition的接口改变名称就是为了避免与Object中的wait/notify/notifyAll的语义和使用上混淆，因为Condition同样有wait/notify/notifyAll方法。每一个Lock可以有任意数据的Condition对象，Condition是与Lock绑定的，所以就有Lock的公平性特性：如果是公平锁，线程为按照FIFO的顺序从Condition.await中释放，如果是非公平锁，那么后续的锁竞争就不保证FIFO顺序了。

**6 代码举例**

package com.hiya.concurrent.condition;

import java.util.concurrent.locks.Condition;

import java.util.concurrent.locks.Lock;

import java.util.concurrent.locks.ReentrantLock;

public class ProductQueue<T>

{

private final T[] items;

private final Lock lock = new ReentrantLock();

private Condition notFull = lock.newCondition();

private Condition notEmpty = lock.newCondition();

private int head, tail, count;

@SuppressWarnings("unchecked")

public ProductQueue(int maxSize)

{

items = (T[]) new Object[maxSize];

}

public ProductQueue()

{

this(10);

}

/\*\*

\* 生产put()需要队列不满，如果满了就挂起（await()），直到收到notFull的信号。

\* @param t

\* @throws InterruptedException

\*/

public void put(T t) throws InterruptedException

{

lock.lock();

try

{

while (count == getCapacity())

{

//在进入lock.lock()后唯一可能释放锁的操作就是await()了。也就是说await()操作实际上就是释放锁，然后挂起线程，一旦条件满足就被唤醒，再次获取锁

notFull.await();

}

items[tail] = t;

if (++tail == getCapacity())

{

tail = 0;

}

++count;

notEmpty.signalAll();

} finally

{

lock.unlock();

}

}

/\*\*

\* 消费take()需要 队列不为空，如果为空就挂起（await()），直到收到notEmpty的信号；

\* @return

\* @throws InterruptedException

\*/

public T take() throws InterruptedException

{

lock.lock();

try

{

while (count == 0)

{

//在进入lock.lock()后唯一可能释放锁的操作就是await()了。也就是说await()操作实际上就是释放锁，然后挂起线程，一旦条件满足就被唤醒，再次获取锁

notEmpty.await();

}

T ret = items[head];

items[head] = null;// GC

//

if (++head == getCapacity())

{

head = 0;

}

--count;

notFull.signalAll();

return ret;

} finally

{

lock.unlock();

}

}

public int getCapacity()

{

return items.length;

}

public int size()

{

lock.lock();

try

{

return count;

} finally

{

lock.unlock();

}

}

}