一、线程基本概念

1. 线程的创建

线程表示一条单独的执行流，它有自己的程序执行计数器，有自己的栈。

在Java中创建线程有两种方式，一种是继承Thread，另外一种是实现Runnable接口。

1.1 继承Thread类

Java中java.lang.Thread这个类表示线程，一个类可以继承Thread并重写其run方法来实现一个线程

(1) 代码：

//继承Thread类，定义线程类

public class HelloThread extends Thread {

@Override

public void run() {

System.out.println("hello");

}

}

//启动线程，main为一个线程，相当于同时运行两个线程

public static void main(String[] args) {

Thread thread = new HelloThread();

thread.start();

}

(2) start() 和 run() 的区别

① start表示启动该线程，使其成为一条单独的执行流，背后，操作系统会分配线程相关的资源，**每个线程会有单独的程序执行计数器和栈**，操作系统会把这个线程作为一个独立的个体进行调度，分配时间片让它执行，执行的起点就是run方法。

② 屏幕的输出并不会发生变化，但并不会启动一条单独的执行流，run方法的代码依然是在main线程中执行的，run方法只是main方法调用的一个普通方法。

(3) 查看当前线程的方法 Thread类的currentThread()方法

public static native Thread currentThread();

(4) 线程类Thread的属性 – id和name

public long getId()

public final String getName()

1.2 实现Runnable接口

(1) Runnable接口的定义

public interface Runnable {

public abstract void run();

}

(2) 线程实现

public class HelloRunnable implements Runnable {

@Override

public void run() {

System.out.println("hello");

}

}

(3) 实现Runnable接口和继承Thread类的线程启动不一样

public static void main(String[] args) {

//创建一个Thread对象，传递一个Runnable对象

Thread helloThread = new Thread(new HelloRunnable());

helloThread.start();

}

2. 线程（Tread）的基本属性和方法

(1) id和name

每个线程都有一个id和name，id是一个递增的整数，每创建一个线程就加一。

name的默认值是"Thread-"后跟一个编号，name可以在Thread的构造方法中进行指定，也可以通过setName方法进行设置。

(2) 优先级

线程有一个优先级的概念，在Java中，优先级从1到10，默认为5。

public final void setPriority(int newPriority)

public final int getPriority()

注意：这个优先级会被映射到操作系统中线程的优先级，不过，因为操作系统各不相同，不一定都是10个优先级，Java中不同的优先级可能会被映射到操作系统中相同的优先级，另外，优先级对操作系统而言更多的是一种建议和提示，而非强制，简单的说，在编程中，不要过于依赖优先级。

(3) 状态

① 线程有一个状态的概念，Thread有一个方法用于获取线程的状态：

public State getState()；

返回值类型为Thread.State，它是一个枚举类型，有如下值：

public enum State {

NEW, //没有调用start的线程状态为NEW

RUNNABLE, //线程运行结束后状态为TERMINATED

BLOCKED, //表示线程被阻塞了，在等待一些条件

WAITING, //表示线程被阻塞了，在等待一些条件

TIMED\_WAITING, //表示线程被阻塞了，在等待一些条件

TERMINATED; //调用start后线程在执行run方法且没有阻塞时状态为RUNNABLE，

不过，RUNNABLE不代表CPU一定在执行该线程的代码 , 可能正在执行也可能在等待操作系统分配时间片，只是它没有在等待其他条件

}

② Thread还有一个方法，返回线程是否活着

public final native boolean isAlive()；

线程被启动后，run方法运行结束前，返回值都是true。

(4) 是否daemo线程（守护线程）

① Thread有一个是否daemo线程的属性，相关方法是：

public final void setDaemon(boolean on)；//设置守护线程

public final boolean isDaemon()； //返回是否守护线程

② daemo线程有什么用呢？**它一般是其他线程的辅助线程**，在它辅助的主线程退出的时候，它就没有存在的意义了。在我们运行一个即使最简单的"hello world"类型的程序时，实际上，Java也会创建多个线程，除了main线程外，至少还有一个负责垃圾回收的线程，这个线程就是daemo线程，在main线程结束的时候，垃圾回收线程也会退出。

③ 启动线程会启动一条单独的执行流，整个程序只有在所有线程都结束的时候才退出，但daemo线程是例外，当整个程序中剩下的都是daemo线程的时候，程序就会退出。

(4) Thread类sleep()方法

① Thread的静态的sleep方法，调用该方法会让当前线程睡眠指定的时间，单位是毫秒：

//会抛出中断异常，sleep会让出CPU，同时也会放弃锁

public static native void sleep(long millis) throws InterruptedException;

② 调用sleep会让出锁，睡眠期间，该线程会让出CPU，但睡眠的时间不一定是确切的给定毫秒数，可能有一定的偏差，偏差与系统定时器和操作系统调度器的准确度和精度有关。

③ 睡眠期间，线程可以被中断，如果被中断，sleep会抛出InterruptedException，关于中断以及中断处理，我们后续章节再介绍。

(5) Thread类yield ()方法 – 放弃CPU

public static native void yield();

一个静态方法，调用该方法，是告诉操作系统的调度器，我现在不着急占用CPU，你可以先让其他线程运行。不过，这对调度器也仅仅是建议，调度器如何处理是不一定的，它可能完全忽略该调用。

(6) Thread类join ()方法 – 并行执行变为串行执行

① Thread类中的join方法的主要作用就是同步，它可以使得线程之间的并行执行变为串行执行。

//当前线程等待调用join方法线程结束的过程中如果被中断，会抛出

InterruptedException异常

public final void join() throws InterruptedException

② join方法还有一个变体，可以限定等待的最长时间，单位为毫秒，如果为0，表示无期限等待：

public final synchronized void join(long millis) throws InterruptedException

③ 实例：

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

ThreadJoinTest t1 = new ThreadJoinTest("小明");

ThreadJoinTest t2 = new ThreadJoinTest("小东");

t1.start();

//当前线程为main和t1并行执行，t1调用join表示，main放弃当前CPU交给t1

//执行，直到t1执行完，main线程再继续执行

t1.join();

t2.start();

}

(7) 过时方法 -- 不能使用

public final void stop()；

public final void suspend()；

public final void resume()；

3. 共享内存

3.1 共享内存基础

(1) 线程、内存和程序代码之间的关系

① 不同的线程可以访问和操作相同的变量。

② 不同的线程可以执行相同的程序代码。

③ 当多个线程执行相同的程序代码时，**每个线程都有单独的栈**，方法中的参数和局部变量都有自己的一份。

3.2 竞争条件

(1) 计算机运行过程中，并发、无序、大量的进程在使用有限、独占、不可抢占的资源，由于进程无限，资源有限，产生矛盾，这种矛盾称为竞争（Race）。

(2) 由于两个或者多个进程竞争使用不能被同时访问的资源，使得这些进程有可能因为时间上推进的先后原因而出现问题，这叫做竞争条件（Race Condition）。

(3) 竞争条件分为两类:

① -Mutex（互斥）：两个或多个进程彼此之间没有内在的制约关系，但是由于要抢占使用某个临界资源（不能被多个进程同时使用的资源，如打印机，变量）而产生制约关系。

② -Synchronization（同步）：两个或多个进程彼此之间存在内在的制约关系（前一个进程执行完，其他的进程才能执行），如严格轮转法。

(4) 解决竞争条件所采用的同步方法：

使用synchronized关键字

使用显式锁

使用原子变量

4. 内存的可见性

4.1 实例

多个线程可以共享访问和操作相同的变量，但一个线程对一个共享变量的修改，另一个线程不一定马上就能看到，甚至永远也看不到。

public class VisibilityDemo {

//设置shutdown为false

private static boolean shutdown = false;

//run方法，当shutdown为false时进入死循环

static class HelloThread extends Thread {

@Override

public void run() {

while(!shutdown){

// do nothing

}

System.out.println("exit hello");

}

}

//启动主线程

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

new HelloThread().start(); //启动新线程

Thread.sleep(1000);

//shutdown设为true，但是新线程中不会读取到改变后的值，从而不会退出

shutdown = true;

System.out.println("exit main");

}

}

4.2 问题说明：

在计算机系统中，除了内存，数据还会被缓存在CPU的寄存器以及各级缓存中，当访问一个变量时，可能直接从寄存器或CPU缓存中获取，而不一定到内存中去取，当修改一个变量时，也可能是先写到缓存中，而稍后才会同步更新到内存中。在单线程的程序中，这一般不是个问题，但在多线程的程序中，尤其是在有多CPU的情况下，这就是个严重的问题。一个线程对内存的修改，另一个线程看不到，一是修改没有及时同步到内存，二是另一个线程根本就没从内存读。

4.3 解决方法：

使用volatile关键字

使用synchronized关键字或显式锁同步

5. 线程小结

5.1 线程优点

(1) 充分利用多CPU的计算能力，单线程只能利用一个CPU，使用多线程可以利用多CPU的计算能力。

(2) 充分利用硬件资源，CPU和硬盘、网络是可以同时工作的，一个线程在等待网络IO的同时，另一个线程完全可以利用CPU，对于多个独立的网络请求，完全可以使用多个线程同时请求。

(3) 在用户界面(GUI)应用程序中，保持程序的响应性，界面和后台任务通常是不同的线程，否则，如果所有事情都是一个线程来执行，当执行一个很慢的任务时，整个界面将停止响应，也无法取消该任务。

(4) 简化建模及IO处理，比如，在服务器应用程序中，对每个用户请求使用一个单独的线程进行处理，相比使用一个线程，处理来自各种用户的各种请求，以及各种网络和文件IO事件，建模和编写程序要容易的多。

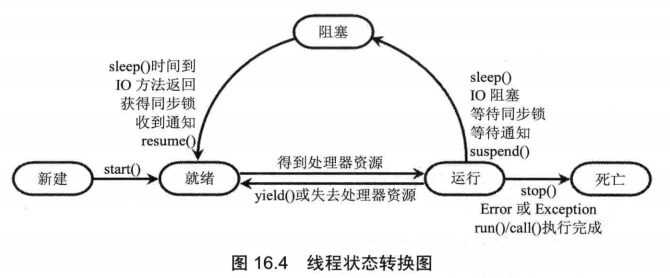
5.2 成本

(1) 创建线程需要消耗操作系统的资源，操作系统会为每个线程创建必要的数据结构、栈、程序计数器等，创建也需要一定的时间。

(2) 线程调度和切换也是有成本的，当有当量可运行线程的时候，操作系统会忙于调度，为一个线程分配一段时间，执行完后，再让另一个线程执行，一个线程被切换出去后，操作系统需要保存它的当前上下文状态到内存，上下文状态包括当前CPU寄存器的值、程序计数器的值等，而一个线程被切换回来后，操作系统需要恢复它原来的上下文状态，整个过程被称为上下文切换，这个切换不仅耗时，而且使CPU中的很多缓存失效，是有成本的。

(3) 如果执行的任务都是CPU密集型的，即主要消耗的都是CPU，那创建超过CPU数量的线程就是没有必要的，并不会加快程序的执行。

6. 线程的状态转换



二、同步关键字-Synchronized

1. 基本用法

synchronized可以用于修饰类的实例方法、静态方法和代码块。

1.2 synchronized实例方法实际保护的是**同一个对象的方法调用**，确保同时只能有一个线程执行。再具体来说，synchronized实例方法保护的是当前实例对象，即this，this对象有一个锁和一个等待队列，锁只能被一个线程持有，其他试图获得同样锁的线程需要等待，执行synchronized实例方法的过程大概如下：

(1) 尝试获得锁，如果能够获得锁，继续下一步，否则加入等待队列，阻塞并等待唤醒

(2) 执行实例方法体代码

(3) 释放锁，如果等待队列上有等待的线程，从中取一个并唤醒，如果有多个等待的线程，唤醒哪一个是不一定的，不保证公平性。

注意：

(1) synchronized保护的是对象而非代码，只要访问的是同一个对象的synchronized方法，即使是不同的代码，也会被同步顺序访问。

(2) 需要说明的，synchronized方法不能防止非synchronized方法被同时执行。比如有一个同步方法和一个非同步方法同时访问一个变量，此时不能保证该变量是线程安全的。

1.3 synchronized的其他用法

(1) 修饰静态方法

synchronized保护的是对象，对实例方法，保护的是当前实例对象this。对静态方法，是类对象，这里是StaticCounter.class，实际上，每个对象都有一个锁和一个等待队列，类对象也不例外。

(2) 修饰代码块

① 实例代码：

public void incr(){

synchronized(this){

count ++;

}

}

等效于：

public synchronized void incr(){

count ++;

}

说明：synchronized关键字是对象锁，this表示当前对象，当所需要当前对象的所有方法都

需要同步执行，可以将锁设为Object或者类对象（静态方法）。

2. 特性

2.1 可重入性

(1) 定义

对同一个执行线程，它在获得了锁之后，在调用其他需要同样锁的代码时，可以直接调用，比如说，在一个synchronized实例方法内，可以直接调用其他synchronized实例方法。

(2) 可重入是通过记录锁的持有线程和持有数量来实现的，当调用被synchronized保护的代码时，检查对象是否已被锁，如果是，再检查是否被当前线程锁定，如果是，增加持有数量，如果不是被当前线程锁定，才加入等待队列，当释放锁时，减少持有数量，当数量变为0时才释放整个锁。

2.2 内存可见性

(1) 作用：

保证内存可见性，在释放锁时，所有写入都会写回内存，而获得锁后，都会从内存中读最新数据。

(2) volatile关键字 – 用于保证内存可见性

public class Switcher {

//操作该变量时，确保读写到内存最新值

private volatile boolean on;

public boolean isOn() {

return on;

}

public void setOn(boolean on) {

this.on = on;

}

}

2.3 死锁

(1) 定义

所谓死锁就是类似这种现象，比如， 有a, b两个线程，a持有锁A，在等待锁B，而b持有锁B，在等待锁A，a,b陷入了互相等待，最后谁都执行不下去。

(2) 注意

应该尽量避免在持有一个锁的同时去申请另一个锁，如果确实需要多个锁，所有代码都应该按照相同的顺序去申请锁。

3. 同步容器

3.1 基本介绍

public static <T> Collection<T> synchronizedCollection(Collection<T> c)

public static <T> List<T> synchronizedList(List<T> list)

public static <K,V> Map<K,V> synchronizedMap(Map<K,V> m)

给所有容器方法都加上synchronized来实现安全的，加了synchronized，所有方法调用变成了原子操作。线程安全针对的是容器对象，指的是当多个线程并发访问同一个容器对象时，不需要额外的同步操作，也不会出现错误的结果。

3.2 其他情况 – 造成线程不安全

(1) 复合操作，比如先检查再更新

public V putIfAbsent(K key, V value){

V old = map.get(key);

//当对应键值为空时，添加value值

if(old!=null){

return old;

}

map.put(key, value);

return null;

}

线程不安全情况：

当多个线程同时检查完后，再一起调用put方法会造成线程不安全。因为即使map类是线程安全，先检查后更新的复合操作得不到该操作所期望的值。

(2) 伪同步

//synchronized同步的对象的是当前方法所在的对象

public synchronized V putIfAbsent(K key, V value){

//需要同步的操作，需要同步的对象是map

//所以同步对象不一样还是会存在问题

V old = map.get(key);

if(old!=null){

return old;

}

map.put(key, value);

return null;

}

原因：锁住的对象不一样会产生伪同步的问题。

(3) 迭代

synchronized(list){

for (String str : list) {

}

当一个线程修改list，一个线程遍历list，会出现线程安全的问题。

在遍历的时候给整个容器对象加锁可以解决这个问题。

4. 并发容器

(1) 背景

同步容器的性能也是比较低的，当并发访问量比较大的时候性能很差。

(2) 并发容器类，这些容器类都是线程安全的，但都没有使用synchronized、没有迭代问题、直接支持一些复合操作、性能也高得多。

CopyOnWriteArrayList

ConcurrentHashMap

ConcurrentLinkedQueue

ConcurrentSkipListSet

三、线程的基本协作机制

1. Object类方法 wait/notify

1.1 wait方法

//不带时间参数，表示无限期等待，实际就是调用wait(0)。

public final void wait() throws InterruptedException

//带时间参数，单位是毫秒，表示最多等待这么长时间，参数为0表示无限期等待。

//在等待期间都可以被中断，如果被中断，会抛出InterruptedException

public final native void wait(long timeout) throws InterruptedException;

(1) 作用：

导致当前线程等待，直到另一个线程调用该对象的 notify()方法或 notifyAll()方法。

sleep()不会让出锁，wait()会让出锁进入阻塞状态。

1.2 notify方法

//唤醒等待该对象锁的随机线程，和线程的优先级有关，进入可运行状态

public final native void notify();

//唤醒所有等待该对象锁的线程，进入可运行状态

public final native void notifyAll();

(1) 实际上，wait/notify方法只能在synchronized代码块内被调用，如果调用wait/notify方法时，当前线程没有持有对象锁，会抛出异常java.lang.IllegalMonitorStateException。

2. 协作场景

2.1 生产者/消费者模式

(1) 假设生产和消费都在同一个队列

//向队尾添加元素

public synchronized void put(E e) throws InterruptedException {

//当队列长度达到限定值时，进入阻塞状态，等待消费唤醒

while (queue.size() == limit) {

wait();

}

//添加元素，并且唤醒所有需要该锁对象的阻塞线程

queue.add(e);

notifyAll();

}

//从队首取出一个元素

public synchronized E take() throws InterruptedException {

//当队列为空时，进入阻塞状态，等待生产唤醒

while (queue.isEmpty()) {

wait();

}

E e = queue.poll();

notifyAll();

return e;

}

2.2 多个线程同时开始

(1) 共享变量是Fireflag

static class FireFlag {

private volatile boolean fired = false;

//当fired为false进入阻塞状态

public synchronized void waitForFire() throws InterruptedException {

while (!fired) {

wait();

}

}

//将fired设为true，唤醒全部需要该对象锁的线程

public synchronized void fire() {

this.fired = true;

notifyAll();

}

}

(2) 多个线程

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

int num = 10;

FireFlag fireFlag = new FireFlag();

//新建多个线程，并且逐个启动

//由于线程调用了fireFlag对象，统一进入阻塞状态

Thread[] racers = new Thread[num];

for (int i = 0; i < num; i++) {

racers[i] = new Racer(fireFlag);

racers[i].start();

}

Thread.sleep(1000);

//当该fireFlag对象唤醒全部所需该对象锁的线程时，所有线程同时进入可运行状态。

fireFlag.fire();

}

2.3 等待结束

(1) 背景：

主从协作模式也是一种常见的协作模式，主线程将任务分解为若干个子任务，为每个子任务创建一个线程，主线程在继续执行其他任务之前需要等待每个子任务执行完毕。

2.4 异步结果

(1) 背景：

在主从协作模式中，主线程手工创建子线程的写法往往比较麻烦，一种常见的模式是将子线程的管理封装为异步调用，异步调用马上返回，但返回的不是最终的结果，而是一个一般称为Promise或Future的对象，通过它可以在随后获得最终的结果。

2.5 集合点

(1) 背景：

类似于学校或公司组团旅游，在旅游过程中有若干集合点，比如出发集合点，每个人从不同地方来到集合点，所有人到齐后进行下一项活动，在一些程序，比如并行迭代计算中，每个线程负责一部分计算，然后在集合点等待其他线程完成，所有线程到齐后，交换数据和计算结果，再进行下一次迭代。

四、线程的中断机制

1. 背景 – 非强制

在Java中，停止一个线程的主要机制是中断，中断并不是强迫终止一个线程，它是一种协作机制，是给线程传递一个取消信号，但是由线程来决定如何以及何时退出。

2. 中断方法（Thread类）

//实例方法。返回对应线程的中断标志位是否为true。

public boolean isInterrupted()；

//实例方法。表示中断对应的线程，

public void interrupt()；

//静态方法。返回当前线程的中断标志位是否为true，但它还有一个重要的副作用，就是清空中断标志位，也就是说，连续两次调用interrupted()，第一次返回的结果为true，第二次一般就是false (除非同时又发生了一次中断)。

public static boolean interrupted()；

3. 线程对中断的反应

3.1 线程的状态

RUNNABLE：运行/可运行状态

WAITING/TIMED\_WAITING：线程在等待某个条件或超时

BLOCKED：阻塞状态

NEW/TERMINATED：线程刚创建还未启动/线程已经死亡

(1) RUNNABLE

如果线程在运行中，且没有执行IO操作，interrupt()只是会设置线程的中断标志位，没有任何其它作用。线程应该在运行过程中合适的位置检查中断标志位。

public class InterruptRunnableDemo extends Thread {

@Override

public void run() {

//当中断标志位真的时候，直接跳出循环，执行完线程代码

while (!Thread.currentThread().isInterrupted()) {

// ... 单次循环代码

}

System.out.println("done ");

}

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

Thread thread = new InterruptRunnableDemo();

thread.start();

Thread.sleep(1000);

thread.interrupt();

}

}

(2) WAITING/TIMED\_WAITIN --G -- 等待被唤醒状态，和阻塞差不多

① 线程执行如下方法会进入WAITING状态：

public final void join() throws InterruptedException

public final void wait() throws InterruptedException

② 执行如下方法会进入TIMED\_WAITING状态：

public final native void wait(long timeout) throws InterruptedException;

public static native void sleep(long millis) throws InterruptedException;

public final synchronized void join(long millis) throws InterruptedException

注意：该状态是通过这五个方法进入的，等待获得锁，可以统一理解为阻塞状态。

③ 在这些状态时，对线程对象调用interrupt()会使得该线程抛出InterruptedException，需要注意的是，抛出异常后，中断标志位会被清空，而不是被设置。

(3) BLOCKED -- 阻塞状态，这里特指自然等待的队列

如果线程在等待锁，对线程对象调用interrupt()只是会设置线程的中断标志位，线程依然会处于BLOCKED状态，也就是说，interrupt()并不能使一个在等待锁的线程真正"中断"。

① 在使用synchronized关键字获取锁的过程中不响应中断请求，这是synchronized的局限性。

② 显式锁Lock接口，支持以响应中断的方式获取锁。

(4) NEW/TERMINATE – 新创建还未启动/已经死亡

如果线程尚未启动(NEW)，或者已经结束(TERMINATED)，则调用interrupt()对它没有任何效果，中断标志位也不会被设置。

(5) IO操作 - 如果线程在等待IO操作，尤其是网络IO，则会有一些特殊的处理

① 如果IO通道是可中断的，即实现了InterruptibleChannel接口，则线程的中断标志位会被设置，同时，线程会收到异常ClosedByInterruptException。

② 如果线程阻塞于Selector调用，则线程的中断标志位会被设置，同时，阻塞的调用会立即返回。

③ InputStream的read调用，该操作是不可中断的，如果流中没有数据，read会阻塞 (但线程状态依然是RUNNABLE)，且不响应interrupt()。

4. 小结

(1) interrupt方法不一定会真正"中断"线程，它只是一种协作机制，如果不明白线程在做什么，不应该贸然的调用线程的interrupt方法，以为这样就能取消线程。

(2) 对于以线程提供服务的程序模块而言，它应该封装取消/关闭操作，提供单独的取消/关闭方法给调用者，类似于InterruptReadDemo中演示的cancel方法，外部调用者应该调用这些方法而不是直接调用interrupt。

五、原子变量和CAS

1. 原子变量类型 -- 实现原子操作的变量

(1) 基本原子变量类型

AtomicBoolean ：原子Boolean类型

AtomicInteger ：原子Integer类型

AtomicLong ：原子Long类型

AtomicReference ：原子引用类型

(2) 数组类型

AtomicIntegerArray ：原子Integer数组类型

AtomicLongArray ：原子Long数组类型

AtomicReferenceArray ：原子引用类型数组

(3) 没有针对char, short, float, double类型的原子变量，可以通过转换类型使用。

2. AtomicInteger

(1) 构造方法：

public AtomicInteger(int initialValue) //给定初始值

public AtomicInteger() //初始值为0

(2) 设置和获取

public final int get()

public final void set(int newValue)

(3) 以原子方式实现组合操作的方法 – 被称为原子变量的原因

//以原子方式获取旧值并设置新值

public final int getAndSet(int newValue)

//以原子方式获取旧值并给当前值加1

public final int getAndIncrement()

等等。。。

(4) 核心方法

public final boolean compareAndSet(int expect, int update) //比较并设置，简称为CAS

① 作用：如果当前值等于expect，则更新为update，否则不更新，如果更新成功，返回true，否则返回false。

(5) 更新方式的区别

① synchronized代表一种阻塞式算法，得不到锁的时候，进入锁等待队列，等待其他线程唤醒，有上下文切换开销。（悲观锁）

② 原子变量的更新逻辑是非阻塞式的，更新冲突的时候，它就重试，不会阻塞，不会有 上下文切换开销。（乐观锁）

3. AtomicBoolean -可以用来在程序中表示一个标志位

public final boolean compareAndSet(boolean expect, boolean update)

public final boolean getAndSet(boolean newValue)

4. AtomicLong --可以用来在程序中生成唯一序列号

5. 原子数组

(1) 构造方法

//创建一个长度为length的空数组

public AtomicIntegerArray(int length)

//接受一个已有的数组，但不会直接操作该数组，而是会创建一个新数组，只是拷

//贝参数数组中的内容到新数组。

public AtomicIntegerArray(int[] array)

(2) 更新方法

//比较并设置

public final boolean compareAndSet(int i, int expect, int update)

public final int getAndIncrement(int i)

public final int getAndAdd(int i, int delta)

6. CAS方法的ABA问题

(1) 背景

一个线程开始看到的值是A，随后使用CAS进行更新，它的实际期望是没有其他线程修改过才更新，但普通的CAS做不到，因为可能在这个过程中，已经有其他线程修改过了，比如先改为了B，然后又改回为了A。

(2) 解决方法

使用AtomicStampedReference，在修改值的同时附加一个时间戳，只有值和时间戳都相同才进行修改。

六、显式锁

1. 基本介绍

Java并发包中的显式锁接口和类位于包java.util.concurrent.locks下，主要接口和类有：

锁接口Lock，主要实现类是ReentrantLock。

读写锁接口ReadWriteLock，主要实现类是ReentrantReadWriteLock。

2. 接口Lock

public interface Lock {

//普通的获取锁和释放锁方法，lock()会阻塞直到成功。

void lock();

void unlock();

//与lock()的不同是，它可以响应中断，如果被其他线程中断了，抛出InterruptedException。

void lockInterruptibly() throws InterruptedException;

//只是尝试获取锁，立即返回，不阻塞，如果获取成功，返回true，否则返回false。

boolean tryLock();

//先尝试获取锁，如果能成功则立即返回true，否则阻塞等待，但等待的最长时间

//定的参数，在等待的同时响应中断，如果发生了中断，抛出InterruptedException，

//在等待的时间内获得了锁，返回true，否则返回false。

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

//新建一个条件，一个Lock可以关联多个条件

Condition newCondition();

}

3. 可重入锁

(1) 作用：

Lock接口的主要实现类是ReentrantLock，它的基本用法lock/unlock实现了与synchronized一样的语义，包括：

① 可重入，一个线程在持有一个锁的前提下，可以继续获得该锁。

② 可以解决竞态条件问题。

③ 可以保证内存可见性。

(2) 构造方法

public ReentrantLock()

//参数fair表示是否保证公平，不指定的情况下，默认为false，表示不保证公平。所谓

//公平是指，等待时间最长的线程优先获得锁。保证公平会影响性能，一般也不需要，

//所以默认不保证，synchronized锁也是不保证公平的，

public ReentrantLock(boolean fair)

3.1 基本用法

使用显式锁，一定要记得调用unlock，一般而言，应该将lock之后的代码包装到try语句内，在finally语句内释放锁。

(1) 具体代码：

public class Counter {

private final Lock lock = new ReentrantLock();

private volatile int count; //保证内存可见性

public void incr() {

lock.lock(); //获取锁

try {

count++;

} finally {

lock.unlock(); //释放锁

}

}

public int getCount() {

return count;

}

}

3.2 tryLock()用法 – 避免死锁

使用tryLock()，可以避免死锁。在持有一个锁，获取另一个锁，获取不到的时候，可以释放已持有的锁，给其他线程机会获取锁，然后再重试获取所有锁。

(1) 实例：

if(from.tryLock()){

try(to.tryLock()){ //尝试获取锁

……..

} finally {

to.unlock(); //如果获取不到锁，则释放已持有的锁，避免死锁

}

}

3.3 获取锁信息

//锁是否被持有，只要有线程持有就返回true，不一定是当前线程持有

public boolean isLocked()

//锁是否被当前线程持有

public boolean isHeldByCurrentThread()

//锁被当前线程持有的数量，0表示不被当前线程持有

public int getHoldCount()

//锁等待策略是否公平

public final boolean isFair()

//是否有线程在等待该锁

public final boolean hasQueuedThreads()

//指定的线程thread是否在等待该锁

public final boolean hasQueuedThread(Thread thread)

//在等待该锁的线程个数

public final int getQueueLength()

4. ReentrantLock和synchronized区别

(1) 相比synchronized，ReentrantLock可以实现与synchronized相同的语义，但还支持以非阻塞方式获取锁、可以响应中断、可以限时等，更为灵活。

(2) synchronized代表一种声明式编程，程序员更多的是表达一种同步声明，由Java系统负责具体实现，程序员不知道其实现细节，显式锁代表一种命令式编程，程序员实现所有细节。

(3) 声明式编程的好处除了简单，还在于性能，在较新版本的JVM上，ReentrantLock和synchronized的性能是接近的，但Java编译器和虚拟机可以不断优化synchronized的实现，比如，自动分析synchronized的使用，对于没有锁竞争的场景，自动省略对锁获取/释放的调用。

简单总结，能用synchronized就用synchronized，不满足要求，再考虑ReentrantLock。

七、显式条件

1. 基本概念

锁用于解决竞态条件问题，条件是线程间的协作机制。显式锁与synchronzied相对应，而显式条件与wait/notify相对应。wait/notify与synchronized配合使用，显式条件与显式锁配合使用。

2. 创建方法 – 通过lock接口的方法创建

条件与锁相关联，创建条件变量需要通过显式锁，Lock接口定义了创建方法：

Condition newCondition()；

2.1 Condition接口

// await()对应于Object的wait()，signal()对应于notify，signalAll()对应于notifyAll()，语

//义也是一样的。

public interface Condition {

void await() throws InterruptedException;

//该方法不会由于中断结束，但当它返回时，如果等待过程中发生了中断，中断标志

//位会被设置。

void awaitUninterruptibly();

// await方法都是响应中断的，如果发生了中断，会抛出InterruptedException，但中

//断标志位会被清空。

//等待时间也是相对时间，但参数单位是纳秒，返回值是nanosTimeout减去实际等

//待的时间

long awaitNanos(long nanosTimeout) throws InterruptedException;

//等待时间是相对时间，如果由于等待超时返回，返回值为false，否则为true

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

//等待时间是绝对时间，如果由于等待超时返回，返回值为false，否则为true

boolean awaitUntil(Date deadline) throws InterruptedException;

void signal();

void signalAll();

}

(1) 一般而言，与Object的wait方法一样，调用await方法前需要先获取锁，如果没有锁，会抛出异常IllegalMonitorStateException。await在进入等待队列后，会释放锁，释放CPU，当其他线程将它唤醒后，或等待超时后，或发生中断异常后，它都需要重新获取锁，获取锁后，才会从await方法中退出。

(2) 一般而言，signal/signalAll与notify/notifyAll一样，调用它们需要先获取锁，如果没有锁，会抛出异常IllegalMonitorStateException。signal与notify一样，挑选一个线程进行唤醒，signalAll与notifyAll一样，唤醒所有等待的线程，但这些线程被唤醒后都需要重新竞争锁，获取锁后才会从await调用中返回。

3. 实例代码：

private Lock lock = new ReentrantLock();

private Condition condition = lock.newCondition(); //lock获得显式条件

@Override

public void run() {

try {

lock.lock();

try {

while (!fire) {

condition.await(); //显式条件调用await，进入等待状态

}

} finally {

lock.unlock();

}

System.out.println("fired");

} catch (InterruptedException e) {

Thread.interrupted();

}

}

(1) 不要将signal/signalAll与notify/notifyAll混淆，notify/notifyAll是Object中定义的方法，Condition对象也有，稍不注意就会误用。运行时会抛出IllegalMonitorStateException

(2) 显式条件与显式锁配合，wait/notify与synchronized配合。