线程与进程区别

总结:进程是所有线程的集合，每一个线程是进程中的一条执行路径。

Java中线程指的就是Thread

多线程的目的

守护线程：

所谓守护线程是指在程序运行的时候在后台提供一种通用服务的线程，比如垃圾回收（gc）线程就是一个很称职的守护者，并且这种线程并不属于程序中不可或缺的部分。因 此，当所有的非守护线程结束时，程序也就终止了，同时会杀死进程中的所有守护线程。反过来说，只要任何非守护线程还在运行，程序就不会终止。

守护线程和用户线程的没啥本质的区别：唯一的不同之处就在于虚拟机的离开：如果用户线程已经全部退出运行了，只剩下守护线程存在了，虚拟机也就退出了。 因为没有了被守护者，守护线程也就没有工作可做了，也就没有继续运行程序的必要了。

将线程转换为守护线程可以通过调用Thread对象的setDaemon(true)方法来实现。

提高程序的效率

多线程创建方式

继承Thread（已经实现了Runnable）

class CreateThread extends Thread {  
 public void run() {  
 }  
 }

实现Runnable

class CreateThread implements Runnable{  
 public void run() {  
 }  
 }

匿名内部类方式

new Thread(new Runnable() {  
 @Override  
 public void run() {  
   
 }  
});

实现Callable（有返回值）

public class Main {  
 public static void main(String[] args) {  
 ThreadDemo td = new ThreadDemo();  
 // 1.执行Callable方式，需要FutureTask实现类的支持，用于接收运算结果  
 FutureTask<Integer> result = new FutureTask<>(td);  
 new Thread(result).start();  
 try {  
 //等所有线程执行完，获取值，因此FutureTask 可用于闭锁  
 Integer sum = result.get();  
 } catch (InterruptedException e) {  
 e.printStackTrace();  
 } catch (ExecutionException e) {  
 e.printStackTrace();  
 }  
 }  
  
}  
  
class ThreadDemo implements Callable<Integer> {  
 @Override  
 public Integer call() throws Exception {  
 return null;  
 }  
}

线程池的方式

ExecutorService pool= Executors.newFixedThreadPool(10);  
Callable task=new Callable() {  
 @Override  
 public Object call() throws Exception {  
 return null;  
 }  
};

pool.submit(task);

实现接口的方式好点，灵活可再继承（继承也有继承的有点如不需要重复实现）

实际上都要创建Thread线程实例，调用它的start方法，不同创建方式只是重写run方法的不同方式，也就是创建Thread类中target方式不同。

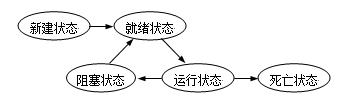
Thread类

线程中最重要的一个类，查看源码仔细阅读。

start方法和run方法

看源码可知start0方法是本地方法（将对硬件的操作交给第三方库），cup会调用run方法，如果直接调用run方法，那只是调用一个普通类实例的一个方法，没有开启多线程。

多线程运行状态



查看源码可知Thread有private volatile int threadStatus = 0,表示线程的状态。在新建、运行等操作时会判断并且改变状态值，如start时if (threadStatus != 0)  
 throw new IllegalThreadStateException();

停止线程思路

1. 使用退出标志，使线程正常退出，也就是当run方法完成后线程终止。

2. 使用stop方法强行终止线程（这个方法不推荐使用，因为stop和suspend、resume一样，也可能发生不可预料的结果）。

3. 使用interrupt方法中断线程。

join()

t.join()表示t线程走完再走其他线程。

wait与sleep区别?

对于sleep()方法，我们首先要知道该方法是属于Thread类中的。而wait()方法，则是属于Object类中的。

sleep()方法导致了程序暂停执行指定的时间，让出cpu该其他线程，但是他的监控状态依然保持者，当指定的时间到了又会自动恢复运行状态。

在调用sleep()方法的过程中，线程不会释放对象锁。

而当调用wait()方法的时候，线程会放弃对象锁，进入等待此对象的等待锁定池，只有针对此对象调用notify()方法后本线程才进入对象锁定池准备

获取对象锁进入运行状态

Wait通常被用于线程间交互，sleep通常被用于暂停执行。

线程安全问题

当多个线程同时共享，同一个全局变量或静态变量，做写的操作时，可能会发生数据冲突问题，也就是线程安全问题。但是做读操作是不会发生数据冲突问题。

案例:需求现在有100张火车票，有两个窗口同时抢火车票，请使用多线程模拟抢票效果

public class ThreadTest implements Runnable {  
  
 private int count=100;  
  
 @Override  
 public void run() {  
 while (count>0){  
 try {  
 Thread.sleep(50);  
 sale();  
 } catch (InterruptedException e) {  
 e.printStackTrace();  
 }  
 }  
  
 }  
 public void sale() {  
 System.out.println(Thread.currentThread().getName() + ",出售第" + (100 - count + 1) + "票");  
 count--;  
 }  
}  
class Main{  
 public static void main(String[] args) {  
  
 //threadTest为共享资源，一定要是同一个对象，不然就不存在什么线程安全了  
 ThreadTest threadTest=new ThreadTest();  
 Thread t1=new Thread(threadTest);  
 Thread t2=new Thread(threadTest);  
 t1.start();  
 t2.start();  
 }  
}

运行结果：

Thread-0,出售第98票

Thread-1,出售第99票

Thread-0,出售第100票

Thread-1,出售第101票

解决多线程之间线程安全问题

同步synchronized或使用锁(lock)

同步代码块

synchronized(obj){

可能会发生线程冲突问题

}

obj对象监视器必须是同一个对象。如果不是同一个对象监视器，运行的结果就是异步调用了，就会交叉运行

public void sale() {  
 synchronized (obj) {  
 System.out.println(Thread.currentThread().getName() + ",出售第" + (100 - count + 1) + "票");  
 count--;  
   
 }  
}

同步函数 在方法上修饰synchronized 称为同步函数

public synchronized void sale() {  
  
 System.out.println(Thread.currentThread().getName() + ",出售第" + (100 - count + 1) + "票");  
 count--;  
   
  
}

同步函数使用this锁。

静态同步函数 方法上加上static关键字，使用synchronized 关键字修饰 或者使用类.class文件。

静态的同步函数使用的锁是 该函数所属字节码文件对象

@Override  
public void run() {  
  
 synchronized (ThreadTest.class) {  
   
 while (count > 0) {  
 try {  
 Thread.sleep(50);  
 sale();  
 } catch (InterruptedException e) {  
 e.printStackTrace();  
 }  
 }  
 }  
}

Lock

在 jdk1.5 之后，并发包中新增了 Lock 接口(以及相关实现类)用来实现锁功能，Lock 接口提供了与 synchronized 关键字类似的同步功能，但需要在使用时手动获取锁和释放锁。

写法

Lock lock = new ReentrantLock();

lock.lock();

try{

//可能会出现线程安全的操作

}finally{

//一定在finally中释放锁

//也不能把获取锁在try中进行，因为有可能在获取锁的时候抛出异常

lock.unlock();

}

Lock 接口更灵活。

Lock是一个接口，而synchronized是Java中的关键字，synchronized是内置的语言实现；

Lock 接口可以尝试非阻塞地获取锁 当前线程尝试获取锁。如果这一时刻锁没有被其他线程获取到，则成功获取并持有锁。  
Lock 接口能被中断地获取锁 与 synchronized 不同，获取到锁的线程能够响应中断，当获取到的锁的线程被中断时，中断异常将会被抛出，同时锁会被释放。

Lock 接口在指定的截止时间之前获取锁，如果截止时间到了依旧无法获取锁，则返回。

ReentrantLock

意思是“可重入锁”，关于可重入锁的概念在下一节讲述。ReentrantLock是唯一实现了Lock接口的类，并且ReentrantLock提供了更多的方法

public class Test {

    private ArrayList<Integer> arrayList = new ArrayList<Integer>();

    private Lock lock = new ReentrantLock();    //注意这个地方

    public static void main(String[] args)  {

        final Test test = new Test();

        new Thread(){

            public void run() {

                test.insert(Thread.currentThread());

            };

        }.start();

        new Thread(){

            public void run() {

                test.insert(Thread.currentThread());

            };

        }.start();

    }

    public void insert(Thread thread) {

        lock.lock();

        try {

            System.out.println(thread.getName()+"得到了锁");

            for(int i=0;i<5;i++) {

                arrayList.add(i);

            }

        } catch (Exception e) {

            // TODO: handle exception

        }finally {

            System.out.println(thread.getName()+"释放了锁");

            lock.unlock();

        }

    }

}

ReadWriteLock

读写锁，实现类ReentrantReadWriteLock

public class Test {

    private ReentrantReadWriteLock rwl = new ReentrantReadWriteLock();

    public static void main(String[] args)  {

        final Test test = new Test();

        new Thread(){

            public void run() {

                test.get(Thread.currentThread());

            };

        }.start();

        new Thread(){

            public void run() {

                test.get(Thread.currentThread());

            };

        }.start();

    }

    public void get(Thread thread) {

        rwl.readLock().lock();

        try {

            long start = System.currentTimeMillis();

            while(System.currentTimeMillis() - start <= 1) {

                System.out.println(thread.getName()+"正在进行读操作");

            }

            System.out.println(thread.getName()+"读操作完毕");

        } finally {

            rwl.readLock().unlock();

        }

    }

}

Java锁的深度化

悲观锁与乐观锁

悲观锁:悲观锁悲观的认为每一次操作都会造成更新丢失问题，在每次查询时加上排他锁。

每次去拿数据的时候都认为别人会修改，所以每次在拿数据的时候都会上锁，这样别人想拿这个数据就会block直到它拿到锁。传统的关系型数据库里边就用到了很多这种锁机制，比如行锁，表锁等，读锁，写锁等，都是在做操作之前先上锁。

Select \* from xxx for update;

乐观锁:乐观锁会乐观的认为每次查询都不会造成更新丢失,利用版本字段控制

在前面介绍了Lock的基本使用，这一节来介绍一下与锁相关的几个概念。

　　1.可重入锁

　　如果锁具备可重入性，则称作为可重入锁。像synchronized和ReentrantLock都是可重入锁，可重入性在我看来实际上表明了锁的分配机制：基于线程的分配，而不是基于方法调用的分配。举个简单的例子，当一个线程执行到某个synchronized方法时，比如说method1，而在method1中会调用另外一个synchronized方法method2，此时线程不必重新去申请锁，而是可以直接执行方法method2。

　　看下面这段代码就明白了：

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9 | class MyClass {      public synchronized void method1() {          method2();      }        public synchronized void method2() {        }  } |

 　　上述代码中的两个方法method1和method2都用synchronized修饰了，假如某一时刻，线程A执行到了method1，此时线程A获取了这个对象的锁，而由于method2也是synchronized方法，假如synchronized不具备可重入性，此时线程A需要重新申请锁。但是这就会造成一个问题，因为线程A已经持有了该对象的锁，而又在申请获取该对象的锁，这样就会线程A一直等待永远不会获取到的锁。

　　而由于synchronized和Lock都具备可重入性，所以不会发生上述现象。

　　2.可中断锁

　　可中断锁：顾名思义，就是可以相应中断的锁。

　　在Java中，synchronized就不是可中断锁，而Lock是可中断锁。

　　如果某一线程A正在执行锁中的代码，另一线程B正在等待获取该锁，可能由于等待时间过长，线程B不想等待了，想先处理其他事情，我们可以让它中断自己或者在别的线程中中断它，这种就是可中断锁。

　　在前面演示lockInterruptibly()的用法时已经体现了Lock的可中断性。

　　3.公平锁

　　公平锁即尽量以请求锁的顺序来获取锁。比如同是有多个线程在等待一个锁，当这个锁被释放时，等待时间最久的线程（最先请求的线程）会获得该所，这种就是公平锁。

　　非公平锁即无法保证锁的获取是按照请求锁的顺序进行的。这样就可能导致某个或者一些线程永远获取不到锁。

　　在Java中，synchronized就是非公平锁，它无法保证等待的线程获取锁的顺序。

而对于ReentrantLock和ReentrantReadWriteLock，它默认情况下是非公平锁，但是可以设置为公平锁。

　4.读写锁

　　读写锁将对一个资源（比如文件）的访问分成了2个锁，一个读锁和一个写锁。

　　正因为有了读写锁，才使得多个线程之间的读操作不会发生冲突。

　　ReadWriteLock就是读写锁，它是一个接口，ReentrantReadWriteLock实现了这个接口。

　　可以通过readLock()获取读锁，通过writeLock()获取写锁。

CAS（Compare-And-Swap比较并交换）无锁机制



（1）与锁相比，使用比较交换（下文简称CAS）会使程序看起来更加复杂一些。但由于其非阻塞性，它对死锁问题天生免疫，并且，线程间的相互影响也远远比基于锁的方式要小。更为重要的是，使用无锁的方式完全没有锁竞争带来的系统开销，也没有线程间频繁调度带来的开销，因此，它要比基于锁的方式拥有更优越的性能。

（2）无锁的好处：

第一，在高并发的情况下，它比有锁的程序拥有更好的性能；

第二，它天生就是死锁免疫的。

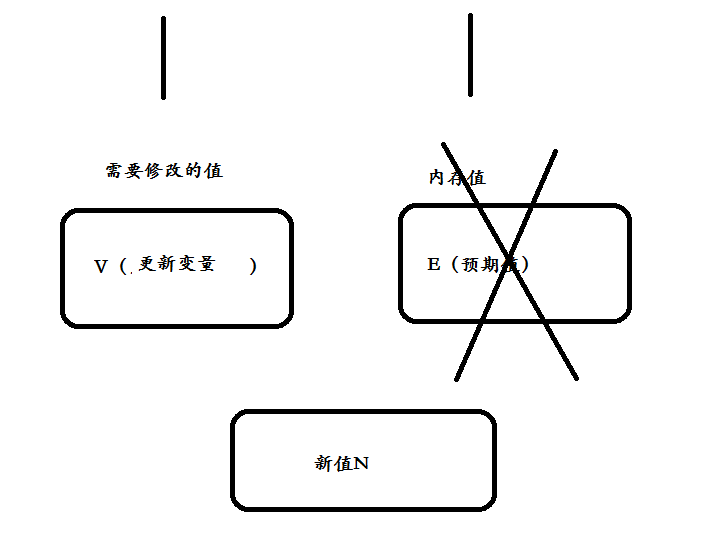
就凭借这两个优势，就值得我们冒险尝试使用无锁的并发。

（3）CAS算法的过程是这样：它包含三个参数CAS(V,E,N): V表示要更新的变量，E表示预期值，N表示新值。仅当V值等于E值时，才会将V的值设为N，如果V值和E值不同，则说明已经有其他线程做了更新，则当前线程什么都不做。最后，CAS返回当前V的真实值。

（4）CAS操作是抱着乐观的态度进行的，它总是认为自己可以成功完成操作。当多个线程同时使用CAS操作一个变量时，只有一个会胜出，并成功更新，其余均会失败。失败的线程不会被挂起，仅是被告知失败，并且允许再次尝试，当然也允许失败的线程放弃操作。基于这样的原理，CAS操作即使没有锁，也可以发现其他线程对当前线程的干扰，并进行恰当的处理。

（5）简单地说，CAS需要你额外给出一个期望值，也就是你认为这个变量现在应该是什么样子的。如果变量不是你想象的那样，那说明它已经被别人修改过了。你就重新读取，再次尝试修改就好了。

（6）在硬件层面，大部分的现代处理器都已经支持原子化的CAS指令。在JDK 5.0以后，虚拟机便可以使用这个指令来实现并发操作和并发数据结构，并且，这种操作在虚拟机中可以说是无处不在。



|  |
| --- |
| /\*\*  \* Atomically increments by one the current value.  \*  \* @return the updated value  \*/  public final int incrementAndGet() {  for (;;) {  //获取当前值  int current = get();  //设置期望值  int next = current + 1;  //调用Native方法compareAndSet，执行CAS操作  if (compareAndSet(current, next))  //成功后才会返回期望值，否则无线循环  return next;  }  } |

CAS在Java中的应用，即并发包中的原子操作类(Atomic系列)，从JDK 1.5开始提供了java.util.concurrent.atomic包，在该包中提供了许多基于CAS实现的原子操作类，用法方便，性能高效

自旋锁

自旋锁是采用让当前线程不停地的在循环体内执行实现的，当循环的条件被其他线程改变时 才能进入临界区。如下

|  |
| --- |
| public class SpinLock {  private AtomicReference<Thread> sign =new AtomicReference<>();  public void lock() {  Thread current = Thread.currentThread();  while (!sign.compareAndSet(null, current)) {  }  }  public void unlock() {  Thread current = Thread.currentThread();  sign.compareAndSet(current, null);  }  } |
|  |

|  |
| --- |
| public class Test implements Runnable {  static int sum;  private SpinLock lock;  public Test(SpinLock lock) {  this.lock = lock;  }  /\*\*  \* @param args  \* @throws InterruptedException  \*/  public static void main(String[] args) throws InterruptedException {  SpinLock lock = new SpinLock();  for (int i = 0; i < 100; i++) {  Test test = new Test(lock);  Thread t = new Thread(test);  t.start();  }  Thread.currentThread().sleep(1000);  System.out.println(sum);  }  @Override  public void run() {  this.lock.lock();  this.lock.lock();  sum++;  this.lock.unlock();  this.lock.unlock();  }  } |

当一个线程 调用这个不可重入的自旋锁去加锁的时候没问题，当再次调用lock()的时候，因为自旋锁的持有引用已经不为空了，该线程对象会误认为是别人的线程持有了自旋锁

使用了CAS原子操作，lock函数将owner设置为当前线程，并且预测原来的值为空。unlock函数将owner设置为null，并且预测值为当前线程。

当有第二个线程调用lock操作时由于owner值不为空，导致循环一直被执行，直至第一个线程调用unlock函数将owner设置为null，第二个线程才能进入临界区。

由于自旋锁只是将当前线程不停地执行循环体，不进行线程状态的改变，所以响应速度更快。但当线程数不停增加时，性能下降明显，因为每个线程都需要执行，占用CPU时间。如果线程竞争不激烈，并且保持锁的时间段。适合使用自旋锁。

分布式锁

如果想在不同的jvm中保证数据同步，使用分布式锁技术。

有数据库实现、缓存实现、Zook

Condition用法

Condition condition = lock.newCondition();

res. condition.await(); 类似wait

res. Condition. Signal() 类似notify

。

死锁： 同步中嵌套同步,导致锁无法释放

死锁解决办法:不要在同步中嵌套同步

死锁示例：

public class ThreadTest implements Runnable {  
  
 private int count = 100;  
 public boolean flag = true;  
 private Object obj = new Object();  
  
 @Override  
 public void run() {  
 if (flag) {  
 while (true){  
 synchronized (obj){  
 sale();  
 }  
 }  
 } else {  
 while (true) {  
 sale();  
 }  
 }  
 }  
  
  
 public synchronized void sale() {  
 synchronized (obj) {  
 if(count>0){  
 try {  
 Thread.sleep(40);  
 } catch (Exception e) {  
  
 }  
 System.out.println(Thread.currentThread().getName() + ",出售第" + (100 - count + 1) + "票");  
 count--;  
 }  
 }  
  
 }  
  
  
}  
  
class Main {  
 public static void main(String[] args) throws InterruptedException {  
 //threadTest为共享资源，一定要是同一个对象，不然就不存在什么线程安全了  
 ThreadTest threadTest = new ThreadTest();  
 Thread t1 = new Thread(threadTest);  
 Thread t2 = new Thread(threadTest);  
 t1.start();  
 Thread.sleep(40);  
 threadTest.flag = false;  
 t2.start();  
 }  
}

多线程有三大特性

原子性、可见性、有序性

什么是原子性

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。这2个操作必须要具备原子性才能保证不出现一些意外的问题。

我们操作数据也是如此，比如i = i+1；其中就包括，读取i的值，计算i，写入i。这行代码在[Java](http://lib.csdn.net/base/java" \o "Java 知识库)中是不具备原子性的，则多线程运行肯定会出问题，所以也需要我们使用同步和lock这些东西来确保这个特性了。

原子性其实就是保证数据一致、线程安全一部分，

什么是可见性

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。

若两个线程在不同的cpu，那么线程1改变了i的值还没刷新到主存，线程2又使用了i，那么这个i值肯定还是之前的，线程1对变量的修改线程没看到这就是可见性问题。

什么是有序性

程序执行的顺序按照代码的先后顺序执行。

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。如下：

int a = 10; //语句1

int r = 2; //语句2

a = a + 3; //语句3

r = a\*a; //语句4

则因为重排序，他还可能执行顺序为 2-1-3-4，1-3-2-4  
但绝不可能 2-1-4-3，因为这打破了依赖关系。  
显然重排序对单线程运行是不会有任何问题，而多线程就不一定了，所以我们在多线程编程时就得考虑这个问题了。

Java内存模型

共享内存模型指的就是Java内存模型(简称JMM)，JMM决定一个线程对共享变量的写入时,能对另一个线程可见。从抽象的角度来看，JMM定义了线程和主内存之间的抽象关系：线程之间的共享变量存储在主内存（main memory）中，每个线程都有一个私有的本地内存（local memory），本地内存中存储了该线程以读/写共享变量的副本。本地内存是JMM的一个抽象概念，并不真实存在。它涵盖了缓存，写缓冲区，寄存器以及其他的硬件和编译器优化。



从上图来看，线程A与线程B之间如要通信的话，必须要经历下面2个步骤：

1. 首先，线程A把本地内存A中更新过的共享变量刷新到主内存中去。

2. 然后，线程B到主内存中去读取线程A之前已更新过的共享变量。

下面通过示意图来说明这两个步骤：   


如上图所示，本地内存A和B有主内存中共享变量x的副本。假设初始时，这三个内存中的x值都为0。线程A在执行时，把更新后的x值（假设值为1）临时存放在自己的本地内存A中。当线程A和线程B需要通信时，线程A首先会把自己本地内存中修改后的x值刷新到主内存中，此时主内存中的x值变为了1。随后，线程B到主内存中去读取线程A更新后的x值，此时线程B的本地内存的x值也变为了1。

从整体来看，这两个步骤实质上是线程A在向线程B发送消息，而且这个通信过程必须要经过主内存。JMM通过控制主内存与每个线程的本地内存之间的交互，来为java程序员提供内存可见性保证。

总结：什么是Java内存模型：java内存模型简称jmm，定义了一个线程对另一个线程可见。共享变量存放在主内存中，每个线程都有自己的本地内存，当多个线程同时访问一个数据的时候，可能本地内存没有及时刷新到主内存，所以就会发生线程安全问题。

Volatile

Volatile 关键字的作用是变量在多个线程之间可见，每次读取变量从主类存中读取。

public class ThreadVolatileDemo extends Thread {  
 public boolean flag = true;  
  
 @Override  
 public void run() {  
 System.out.println("开始执行子线程....");  
 while (flag) {  
 }  
 System.out.println("线程停止");  
 }  
  
 public void setRuning(boolean flag) {  
 this.flag = flag;  
 }  
  
 public static void main(String[] args) throws InterruptedException {  
 ThreadVolatileDemo threadVolatileDemo = new ThreadVolatileDemo();  
 threadVolatileDemo.start();  
 Thread.sleep(3000);  
 threadVolatileDemo.setRuning(false);  
   
 }  
}

运行结果

开始执行子线程....

子线程没有结束

已经将结果设置为fasle为什么？还一直在运行呢。

原因:线程之间是不可见的，读取的是副本，没有及时读取到主内存结果。

解决办法使用Volatile关键字将解决线程之间可见性, 强制线程每次读取该值的时候都去“主内存”中取值

将 public boolean flag = true 改为public volatile boolean flag = true;

运行结果

开始执行子线程....

线程停止

Volatile 只是使变量可见性，但具有原子性

示例

public class VolatileNoAtomic extends Thread {  
 private static volatile int count;  
 private static void addCount() {  
 for (int i = 0; i < 1000; i++) {  
 count++;  
 // count.incrementAndGet();  
 }  
 System.out.println(count);  
 }  
  
 public void run() {  
 addCount();  
 }  
  
 public static void main(String[] args) {  
 VolatileNoAtomic[] arr = new VolatileNoAtomic[100];  
 for (int i = 0; i < 10; i++) {  
 arr[i] = new VolatileNoAtomic();  
 }  
  
 for (int i = 0; i < 10; i++) {  
 arr[i].start();  
 }  
 }  
}

运行结果

1000

2084

3672

2672

4672

5672

6886

7089

9089

8089

结果发现 数据不同步，因为Volatile不用具备原子性

count++其实是有三个操作组成 1.从主存拿共享变量count  2.进行count++ 3.把count写进 内存

比如 count=5时，线程A和线程B做了以下操作

线程A读取 count的值 5

线程B读取 count的值 5

线程B执行+1操作 6

线程B 写入最新count的值 6

线程A执行+1操作 6

线程A 写入最新count的值 6

导致2个线程操作完后count值为6而不是为7.

解决办法

保证原子性，1：使用锁 synchronized 、lock

2：使用原子类AtomicInteger

AtomicInteger示例

public class VolatileNoAtomic extends Thread {  
  
  
 private static AtomicInteger atomicInteger = new AtomicInteger(0);  
 private static void addCount() {  
  
 for (int i = 0; i < 1000; i++) {  
 //等同于i++  
 atomicInteger.incrementAndGet();  
  
 }  
 System.out.println(atomicInteger);  
 }  
  
 public void run() {  
 addCount();  
 }  
  
 public static void main(String[] args) {  
 VolatileNoAtomic[] volatileNoAtomic = new VolatileNoAtomic[10];  
 for (int i = 0; i < 10; i++) {  
 volatileNoAtomic[i] = new VolatileNoAtomic();  
 }  
  
 for (int i = 0; i < 10; i++) {  
 volatileNoAtomic[i].start();  
 }  
 }  
  
}

Volatile使用场景

要在多线程中使用volatile必须满足

对变量的写入操作不依赖其当前值

.不满足：number++、count=count\*5

.满足：boolean变量、记录温度变化的变量等

该变量没有包含在具有其他变量的不变式中

.不满足 ：不变式 low<up

volatile与synchronized区别

仅靠volatile不能保证线程的安全性。（原子性）

①volatile轻量级，只能修饰变量。synchronized重量级，还可修饰方法

②volatile只能保证数据的可见性，不能用来同步，因为多个线程并发访问volatile修饰的变量不会阻塞。

synchronized不仅保证可见性，而且还保证原子性，因为，只有获得了锁的线程才能进入临界区，从而保证临界区中的所有语句都全部执行。多个线程争抢synchronized锁对象时，会出现阻塞。

线程安全性

线程安全性包括两个方面，①可见性。②原子性。

从上面自增的例子中可以看出：仅仅使用volatile并不能保证线程安全性。而synchronized则可实现线程的安全性。

ThreadLocal

ThreadLocal的实例代表了一个线程局部的变量，每条线程都只能看到自己的值，并不会意识到其它的线程中也存在该变量。它采用采用空间来换取时间的方式，解决多线程中相同变量的访问冲突问题。

public class ThreadLocaDemo extends Thread {  
 private Res res;  
 public ThreadLocaDemo(Res res) {  
 this.res = res;  
 }  
 @Override  
 public void run() {  
 for (int i = 0; i < 3; i++) {  
 System.out.println(Thread.currentThread().getName() + "---" + "i---" + i + "--num:" + res.getNum());  
 }  
 }  
 public static void main(String[] args) {  
 Res res = new Res();  
 ThreadLocaDemo threadLocaDemo1 = new ThreadLocaDemo(res);  
 ThreadLocaDemo threadLocaDemo2 = new ThreadLocaDemo(res);  
 ThreadLocaDemo threadLocaDemo3 = new ThreadLocaDemo(res);  
 threadLocaDemo1.start();  
 threadLocaDemo2.start();  
 threadLocaDemo3.start();  
 }  
}  
class Res {  
 // 生成序列号共享变量  
 public static Integer count = 0;  
 public static ThreadLocal<Integer> threadLocal = new ThreadLocal<Integer>() {  
 protected Integer initialValue() {  
 return 0;  
 };  
  
 };  
 public Integer getNum() {  
 int count = threadLocal.get() + 1;  
 threadLocal.set(count);  
 return count;  
 }  
}

运行结果

Thread-0---i---0--num:1

Thread-0---i---1--num:2

Thread-0---i---2--num:3

Thread-2---i---0--num:1

Thread-2---i---1--num:2

Thread-2---i---2--num:3

Thread-1---i---0--num:1

Thread-1---i---1--num:2

Thread-1---i---2--num:3

实现原理：

ThreadLocal通过map集合

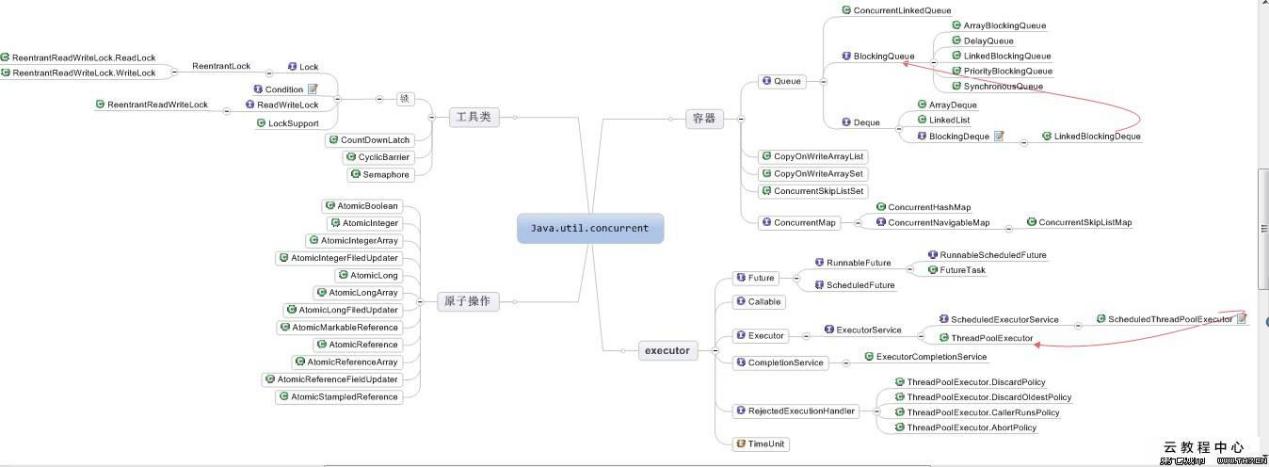
Map.put(“当前线程”,值)；

源码为

public void set(T value) {  
 Thread t = Thread.currentThread();  
 ThreadLocalMap map = getMap(t);  
 if (map != null)  
 map.set(this, value);  
 else  
 createMap(t, value);  
}

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

concurrent并发包结构图

IMG_256

java并发包&线程池原理分析&锁的深度化

并发包

同步容器类

Vector与ArrayList区别

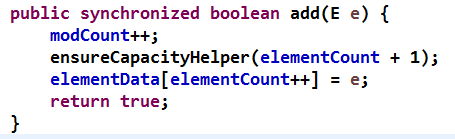
1.ArrayList是最常用的List实现类，内部是通过数组实现的，它允许对元素进行快速随机访问。数组的缺点是每个元素之间不能有间隔，当数组大小不满足时需要增加存储能力，就要讲已经有数组的数据复制到新的存储空间中。当从ArrayList的中间位置插入或者删除元素时，需要对数组进行复制、移动、代价比较高。因此，它适合随机查找和遍历，不适合插入和删除。

2.Vector与ArrayList一样，也是通过数组实现的，不同的是它支持线程的同步，即某一时刻只有一个线程能够写Vector，避免多线程同时写而引起的不一致性，但实现同步需要很高的花费，因此，访问它比访问ArrayList慢

注意: Vector线程安全、ArrayList

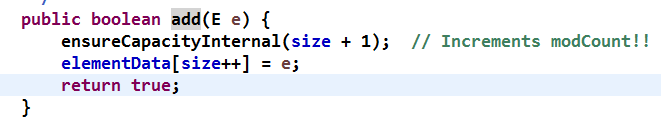
Vector源码类

Add方法源码类



Arraylist源码

Add方法源码



HasTable与HasMap

1.HashMap不是线程安全的

HastMap是一个接口 是map接口的子接口，是将键映射到值的对象，其中键和值都是对象，并且不能包含重复键，但可以包含重复值。HashMap允许null key和null value，而hashtable不允许。

2.HashTable是线程安全的一个Collection。

3.HashMap是Hashtable的轻量级实现（非线程安全的实现），他们都完成了Map接口，主要区别在于HashMap允许空（null）键值（key）,由于非线程安全，效率上可能高于Hashtable。  
HashMap允许将null作为一个entry的key或者value，而Hashtable不允许。  
HashMap把Hashtable的contains方法去掉了，改成containsvalue和containsKey。

注意: HashTable线程安全，HashMap线程不安全。

源码分析

synchronizedMap

Collections.synchronized\*(m) 将线程不安全额集合变为线程安全集合

ConcurrentHashMap

ConcurrentMap接口下有俩个重要的实现 :  
ConcurrentHashMap  
ConcurrentskipListMap (支持并发排序功能。弥补ConcurrentHas hMa p)  
ConcurrentHashMap内部使用段(Segment)来表示这些不同的部分，每个段其实就是一个  
小的HashTable,它们有自己的锁。只要多个修改操作发生在不同的段上，它们就可以并  
发进行。把一个整体分成了16个段(Segment.也就是最高支持16个线程的并发修改操作。  
这也是在重线程场景时减小锁的粒度从而降低锁竞争的一种方案。并且代码中大多共享变  
量使用volatile关键字声明，目的是第一时间获取修改的内容，性能非常好。

CountDownLatch  
CountDownLatch类位于java.util.concurrent包下，利用它可以实现类似计数器的功能。比如有一个任务A，它要等待其他4个任务执行完毕之后才能执行，此时就可以利用CountDownLatch来实现这种功能了。

|  |
| --- |
| public class Test002 {  public static void main(String[] args) throws InterruptedException {  System.out.println("等待子线程执行完毕...");  CountDownLatch countDownLatch = new CountDownLatch(2);  new Thread(new Runnable() {  @Override  public void run() {  System.out.println("子线程," + Thread.currentThread().getName() + "开始执行...");  countDownLatch.countDown();// 每次减去1  System.out.println("子线程," + Thread.currentThread().getName() + "结束执行...");  }  }).start();  new Thread(new Runnable() {  @Override  public void run() {  System.out.println("子线程," + Thread.currentThread().getName() + "开始执行...");  countDownLatch.countDown();  System.out.println("子线程," + Thread.currentThread().getName() + "结束执行...");  }  }).start();  countDownLatch.await();// 调用当前方法主线程阻塞 countDown结果为0, 阻塞变为运行状态  System.out.println("两个子线程执行完毕....");  System.out.println("继续主线程执行..");  }  } |

CyclicBarrier

CyclicBarrier初始化时规定一个数目，然后计算调用了CyclicBarrier.await()进入等待的线程数。当线程数达到了这个数目时，所有进入等待状态的线程被唤醒并继续。

 CyclicBarrier就象它名字的意思一样，可看成是个障碍， 所有的线程必须到齐后才能一起通过这个障碍。

CyclicBarrier初始时还可带一个Runnable的参数， 此Runnable任务在CyclicBarrier的数目达到后，所有其它线程被唤醒前被执行。

|  |
| --- |
| class Writer extends Thread {  private CyclicBarrier cyclicBarrier;  public Writer(CyclicBarrier cyclicBarrier){  this.cyclicBarrier=cyclicBarrier;  }  @Override  public void run() {  System.out.println("线程" + Thread.currentThread().getName() + ",正在写入数据");  try {  Thread.sleep(3000);  } catch (Exception e) {  // TODO: handle exception  }  System.out.println("线程" + Thread.currentThread().getName() + ",写入数据成功.....");    try {  cyclicBarrier.await();  } catch (Exception e) {  }  System.out.println("所有线程执行完毕..........");  }  }  public class Test001 {  public static void main(String[] args) {  CyclicBarrier cyclicBarrier=new CyclicBarrier(5);  for (int i = 0; i < 5; i++) {  Writer writer = new Writer(cyclicBarrier);  writer.start();  }  }  } |

CyclicBarrier和CountDownLatch的区别

CountDownLatch的计数器只能使用一次，而CyclicBarrier的计数器可以使用reset()方法重置，可以使用多次，所以CyclicBarrier能够处理更为复杂的场景；

CyclicBarrier还提供了一些其他有用的方法，比如getNumberWaiting()方法可以获得CyclicBarrier阻塞的线程数量，isBroken()方法用来了解阻塞的线程是否被中断；

CountDownLatch允许一个或多个线程等待一组事件的产生，而CyclicBarrier用于等待其他线程运行到栅栏位置。

Semaphore

Semaphore是一种基于计数的信号量。它可以设定一个阈值，基于此，多个线程竞争获取许可信号，做自己的申请后归还，超过阈值后，线程申请许可信号将会被阻塞。Semaphore可以用来构建一些对象池，资源池之类的，比如数据库连接池，我们也可以创建计数为1的Semaphore，将其作为一种类似互斥锁的机制，这也叫二元信号量，表示两种互斥状态。它的用法如下：

availablePermits函数用来获取当前可用的资源数量

wc.acquire(); //申请资源

wc.release();// 释放资源

|  |
| --- |
| // 创建一个计数阈值为5的信号量对象  // 只能5个线程同时访问  Semaphore semp = new Semaphore(5);    try {  // 申请许可  semp.acquire();  try {  // 业务逻辑  } catch (Exception e) {    } finally {  // 释放许可  semp.release();  }  } catch (InterruptedException e) {    } |

案例:

需求: 一个厕所只有3个坑位，但是有10个人来上厕所，那怎么办？假设10的人的编号分别为1-10，并且1号先到厕所，10号最后到厕所。那么1-3号来的时候必然有可用坑位，顺利如厕，4号来的时候需要看看前面3人是否有人出来了，如果有人出来，进去，否则等待。同样的道理，4-10号也需要等待正在上厕所的人出来后才能进去，并且谁先进去这得看等待的人是否有素质，是否能遵守先来先上的规则。

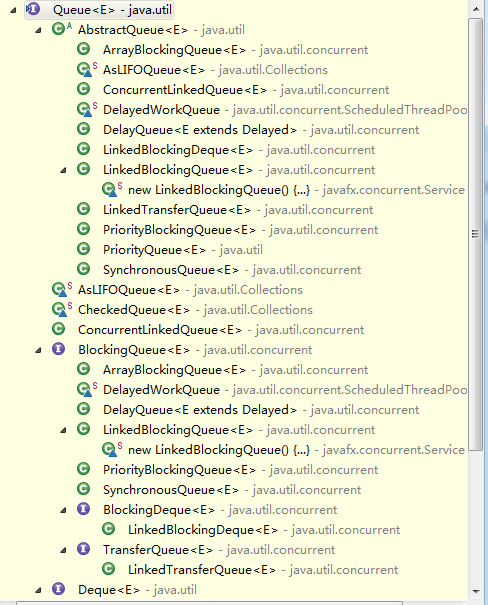
代码:

|  |
| --- |
| class Parent implements Runnable {  private String name;  private Semaphore wc;  public Parent(String name,Semaphore wc){  this.name=name;  this.wc=wc;  }  @Override  public void run() {  try {  // 剩下的资源(剩下的茅坑)  int availablePermits = wc.availablePermits();  if (availablePermits > 0) {  System.out.println(name+"天助我也,终于有茅坑了...");  } else {  System.out.println(name+"怎么没有茅坑了...");  }  //申请茅坑 如果资源达到3次，就等待  wc.acquire();  System.out.println(name+"终于轮我上厕所了..爽啊");  Thread.sleep(new Random().nextInt(1000)); // 模拟上厕所时间。  System.out.println(name+"厕所上完了...");  wc.release();    } catch (Exception e) {  }  }  }  public class TestSemaphore02 {  public static void main(String[] args) {  // 一个厕所只有3个坑位，但是有10个人来上厕所，那怎么办？假设10的人的编号分别为1-10，并且1号先到厕所，10号最后到厕所。那么1-3号来的时候必然有可用坑位，顺利如厕，4号来的时候需要看看前面3人是否有人出来了，如果有人出来，进去，否则等待。同样的道理，4-10号也需要等待正在上厕所的人出来后才能进去，并且谁先进去这得看等待的人是否有素质，是否能遵守先来先上的规则。  Semaphore semaphore = new Semaphore(3);  for (int i = 1; i <=10; i++) {  Parent parent = new Parent("第"+i+"个人,",semaphore);  new Thread(parent).start();  }  }  } |

并发队列

在并发队列上JDK提供了两套实现，一个是以ConcurrentLinkedQueue为代表的高性能队

列，一个是以BlockingQueue接口为代表的阻塞队列，无论哪种都继承自Queue。



ConcurrentLinkedDeque  
ConcurrentLinkedQueue : 是一个适用于高并发场景下的队列，通过无锁的方式，实现  
了高并发状态下的高性能，通常ConcurrentLinkedQueue性能好于BlockingQueue.它  
是一个基于链接节点的无界线程安全队列。该队列的元素遵循先进先出的原则。头是最先  
加入的，尾是最近加入的，该队列不允许null元素。  
ConcurrentLinkedQueue重要方法:  
add 和offer() 都是加入元素的方法(在ConcurrentLinkedQueue中这俩个方法没有任何区别)  
poll() 和peek() 都是取头元素节点，区别在于前者会删除元素，后者不会。

|  |
| --- |
| ConcurrentLinkedDeque q = new ConcurrentLinkedDeque();  q.offer("余胜军");  q.offer("码云");  q.offer("蚂蚁课堂");  q.offer("张杰");  q.offer("艾姐");  //从头获取元素,删除该元素  System.out.println(q.poll());  //从头获取元素,不刪除该元素  System.out.println(q.peek());  //获取总长度  System.out.println(q.size()); |

BlockingQueue

阻塞队列（BlockingQueue）是一个支持两个附加操作的队列。这两个附加的操作是：

在队列为空时，获取元素的线程会等待队列变为非空。  
当队列满时，存储元素的线程会等待队列可用。

阻塞队列常用于生产者和消费者的场景，生产者是往队列里添加元素的线程，消费者是从队列里拿元素的线程。阻塞队列就是生产者存放元素的容器，而消费者也只从容器里拿元素。

BlockingQueue即阻塞队列，从阻塞这个词可以看出，在某些情况下对阻塞队列的访问可能会造成阻塞。被阻塞的情况主要有如下两种：

1. 当队列满了的时候进行入队列操作

2. 当队列空了的时候进行出队列操作

因此，当一个线程试图对一个已经满了的队列进行入队列操作时，它将会被阻塞，除非有另一个线程做了出队列操作；同样，当一个线程试图对一个空队列进行出队列操作时，它将会被阻塞，除非有另一个线程进行了入队列操作。

在Java中，BlockingQueue的接口位于java.util.concurrent 包中(在Java5版本开始提供)，由上面介绍的阻塞队列的特性可知，阻塞队列是线程安全的。

在新增的Concurrent包中，BlockingQueue很好的解决了多线程中，如何高效安全“传输”数据的问题。通过这些高效并且线程安全的队列类，为我们快速搭建高质量的多线程程序带来极大的便利。本文详细介绍了BlockingQueue家庭中的所有成员，包括他们各自的功能以及常见使用场景。

认识BlockingQueue

阻塞队列，顾名思义，首先它是一个队列，而一个队列在数据结构中所起的作用大致如下图所示：

从上图我们可以很清楚看到，通过一个共享的队列，可以使得数据由队列的一端输入，从另外一端输出；

常用的队列主要有以下两种：（当然通过不同的实现方式，还可以延伸出很多不同类型的队列，DelayQueue就是其中的一种）

　　先进先出（FIFO）：先插入的队列的元素也最先出队列，类似于排队的功能。从某种程度上来说这种队列也体现了一种公平性。

　　后进先出（LIFO）：后插入队列的元素最先出队列，这种队列优先处理最近发生的事件。

多线程环境中，通过队列可以很容易实现数据共享，比如经典的“生产者”和“消费者”模型中，通过队列可以很便利地实现两者之间的数据共享。假设我们有若干生产者线程，另外又有若干个消费者线程。如果生产者线程需要把准备好的数据共享给消费者线程，利用队列的方式来传递数据，就可以很方便地解决他们之间的数据共享问题。但如果生产者和消费者在某个时间段内，万一发生数据处理速度不匹配的情况呢？理想情况下，如果生产者产出数据的速度大于消费者消费的速度，并且当生产出来的数据累积到一定程度的时候，那么生产者必须暂停等待一下（阻塞生产者线程），以便等待消费者线程把累积的数据处理完毕，反之亦然。然而，在concurrent包发布以前，在多线程环境下，我们每个程序员都必须去自己控制这些细节，尤其还要兼顾效率和线程安全，而这会给我们的程序带来不小的复杂度。好在此时，强大的concurrent包横空出世了，而他也给我们带来了强大的BlockingQueue。（在多线程领域：所谓阻塞，在某些情况下会挂起线程（即阻塞），一旦条件满足，被挂起的线程又会自动被唤醒）

下面两幅图演示了BlockingQueue的两个常见阻塞场景：

ArrayBlockingQueue

ArrayBlockingQueue是一个有边界的阻塞队列，它的内部实现是一个数组。有边界的意思是它的容量是有限的，我们必须在其初始化的时候指定它的容量大小，容量大小一旦指定就不可改变。

ArrayBlockingQueue是以先进先出的方式存储数据，最新插入的对象是尾部，最新移出的对象是头部。下面

是一个初始化和使用ArrayBlockingQueue的例子：

|  |
| --- |
| ArrayBlockingQueue<String> arrays = new ArrayBlockingQueue<String>(3);  arrays.add("李四");  arrays.add("张军");  arrays.add("张军");  // 添加阻塞队列  arrays.offer("张三", 1, TimeUnit.SECONDS); |

LinkedBlockingQueue

LinkedBlockingQueue阻塞队列大小的配置是可选的，如果我们初始化时指定一个大小，它就是有边界的，如果不指定，它就是无边界的。说是无边界，其实是采用了默认大小为Integer.MAX\_VALUE的容量 。它的内部实现是一个链表。

和ArrayBlockingQueue一样，LinkedBlockingQueue 也是以先进先出的方式存储数据，最新插入的对象是尾部，最新移出的对象是头部。下面是一个初始化和使LinkedBlockingQueue的例子：

|  |
| --- |
| LinkedBlockingQueue linkedBlockingQueue = new LinkedBlockingQueue(3);  linkedBlockingQueue.add("张三");  linkedBlockingQueue.add("李四");  linkedBlockingQueue.add("李四");  System.out.println(linkedBlockingQueue.size()); |

PriorityBlockingQueue

PriorityBlockingQueue是一个没有边界的队列，它的排序规则和 java.util.PriorityQueue一样。需要注

意，PriorityBlockingQueue中允许插入null对象。

所有插入PriorityBlockingQueue的对象必须实现 java.lang.Comparable接口，队列优先级的排序规则就

是按照我们对这个接口的实现来定义的。

另外，我们可以从PriorityBlockingQueue获得一个迭代器Iterator，但这个迭代器并不保证按照优先级顺

序进行迭代。

下面我们举个例子来说明一下，首先我们定义一个对象类型，这个对象需要实现Comparable接口：

SynchronousQueue

SynchronousQueue队列内部仅允许容纳一个元素。当一个线程插入一个元素后会被阻塞，除非这个元素被另一个线程消费。

使用BlockingQueue模拟生产者与消费者

|  |
| --- |
| class ProducerThread implements Runnable {  private BlockingQueue queue;  private volatile boolean flag = true;  private static AtomicInteger count = new AtomicInteger();  public ProducerThread(BlockingQueue queue) {  this.queue = queue;  }  @Override  public void run() {  try {  System.out.println("生产线程启动...");  while (flag) {  System.out.println("正在生产数据....");  String data = count.incrementAndGet()+"";  // 将数据存入队列中  boolean offer = queue.offer(data, 2, TimeUnit.SECONDS);  if (offer) {  System.out.println("生产者,存入" + data + "到队列中,成功.");  } else {  System.out.println("生产者,存入" + data + "到队列中,失败.");  }  Thread.sleep(1000);  }  } catch (Exception e) {  } finally {  System.out.println("生产者退出线程");  }  }  public void stop() {  this.flag = false;  }  }  class ConsumerThread implements Runnable {  private BlockingQueue<String> queue;  private volatile boolean flag = true;  public ConsumerThread(BlockingQueue<String> queue) {  this.queue = queue;  }  @Override  public void run() {  System.out.println("消费线程启动...");  try {  while (flag) {  System.out.println("消费者,正在从队列中获取数据..");  String data = queue.poll(2, TimeUnit.SECONDS);  if (data != null) {  System.out.println("消费者,拿到队列中的数据data:" + data);  Thread.sleep(1000);  } else {  System.out.println("消费者,超过2秒未获取到数据..");  flag = false;  }      }  } catch (Exception e) {  e.printStackTrace();  } finally {  System.out.println("消费者退出线程...");  }    }  }  public class ProducerAndConsumer {  public static void main(String[] args) throws InterruptedException {  BlockingQueue<String> queue = new LinkedBlockingQueue<String>(10);  ProducerThread producerThread1 = new ProducerThread(queue);  ProducerThread producerThread2 = new ProducerThread(queue);  ConsumerThread consumerThread1 = new ConsumerThread(queue);  Thread t1 = new Thread(producerThread1);  Thread t2 = new Thread(producerThread2);  Thread c1 = new Thread(consumerThread1);  t1.start();  t2.start();  c1.start();  // 执行10s  Thread.sleep(10 \* 1000);  producerThread1.stop();  producerThread2.stop();    }  } |

线程池

什么是线程池

Java中的线程池是运用场景最多的并发框架，几乎所有需要异步或并发执行任务的程序  
都可以使用线程池。在开发过程中，合理地使用线程池能够带来3个好处。  
第一：降低资源消耗。通过重复利用已创建的线程降低线程创建和销毁造成的消耗。  
第二：提高响应速度。当任务到达时，任务可以不需要等到线程创建就能立即执行。  
第三：提高线程的可管理性。线程是稀缺资源，如果无限制地创建，不仅会消耗系统资源，  
还会降低系统的稳定性，使用线程池可以进行统一分配、调优和监控。但是，要做到合理利用  
线程池，必须对其实现原理了如指掌。

线程池作用

线程池是为突然大量爆发的线程设计的，通过有限的几个固定线程为大量的操作服务，减少了创建和销毁线程所需的时间，从而提高效率。

如果一个线程的时间非常长，就没必要用线程池了(不是不能作长时间操作，而是不宜。)，况且我们还不能控制线程池中线程的开始、挂起、和中止。

线程池的分类

ThreadPoolExecutor

Java是天生就支持并发的语言，支持并发意味着多线程，线程的频繁创建在高并发及大数据量是非常消耗资源的，因为java提供了线程池。在jdk1.5以前的版本中，线程池的使用是及其简陋的，但是在JDK1.5后，有了很大的改善。JDK1.5之后加入了java.util.concurrent包，java.util.concurrent包的加入给予开发人员开发并发程序以及解决并发问题很大的帮助。这篇文章主要介绍下并发包下的Executor接口，Executor接口虽然作为一个非常旧的接口（JDK1.5 2004年发布），但是很多程序员对于其中的一些原理还是不熟悉，因此写这篇文章来介绍下Executor接口，同时巩固下自己的知识。如果文章中有出现错误，欢迎大家指出。

Executor框架的最顶层实现是ThreadPoolExecutor类，Executors工厂类中提供的newScheduledThreadPool、newFixedThreadPool、newCachedThreadPool方法其实也只是ThreadPoolExecutor的构造函数参数不同而已。通过传入不同的参数，就可以构造出适用于不同应用场景下的线程池，那么它的底层原理是怎样实现的呢，这篇就来介绍下ThreadPoolExecutor线程池的运行过程。

corePoolSize： 核心池的大小。 当有任务来之后，就会创建一个线程去执行任务，当线程池中的线程数目达到corePoolSize后，就会把到达的任务放到缓存队列当中  
maximumPoolSize： 线程池最大线程数，它表示在线程池中最多能创建多少个线程；  
keepAliveTime： 表示线程没有任务执行时最多保持多久时间会终止。  
unit： 参数keepAliveTime的时间单位，有7种取值，在TimeUnit类中有7种静态属性：

线程池四种创建方式

Java通过Executors（jdk1.5并发包）提供四种线程池，分别为：  
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。

案例演示:

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。  
newScheduledThreadPool 创建一个定长线程池，支持定时及周期性任务执行。  
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。

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。示例代码如下：

|  |
| --- |
| // 无限大小线程池 jvm自动回收  ExecutorService newCachedThreadPool = Executors.newCachedThreadPool();  for (int i = 0; i < 10; i++) {  final int temp = i;  newCachedThreadPool.execute(new Runnable() {  @Override  public void run() {  try {  Thread.sleep(100);  } catch (Exception e) {  // TODO: handle exception  }  System.out.println(Thread.currentThread().getName() + ",i:" + temp);  }  });  } |

总结: 线程池为无限大，当执行第二个任务时第一个任务已经完成，会复用执行第一个任务的线程，而不用每次新建线程。

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。示例代码如下：

|  |
| --- |
| ExecutorService newFixedThreadPool = Executors.newFixedThreadPool(5);  for (int i = 0; i < 10; i++) {  final int temp = i;  newFixedThreadPool.execute(new Runnable() {  @Override  public void run() {  System.out.println(Thread.currentThread().getId() + ",i:" + temp);  }  });  } |

总结:因为线程池大小为3，每个任务输出index后sleep 2秒，所以每两秒打印3个数字。

定长线程池的大小最好根据系统资源进行设置。如Runtime.getRuntime().availableProcessors()

newScheduledThreadPool

创建一个定长线程池，支持定时及周期性任务执行。延迟执行示例代码如下：

|  |
| --- |
| ScheduledExecutorService newScheduledThreadPool = Executors.newScheduledThreadPool(5);  for (int i = 0; i < 10; i++) {  final int temp = i;  newScheduledThreadPool.schedule(new Runnable() {  public void run() {  System.out.println("i:" + temp);  }  }, 3, TimeUnit.SECONDS);  } |

表示延迟3秒执行。

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。示例代码如下：

|  |
| --- |
| ExecutorService newSingleThreadExecutor = Executors.newSingleThreadExecutor();  for (int i = 0; i < 10; i++) {  final int index = i;  newSingleThreadExecutor.execute(new Runnable() {  @Override  public void run() {  System.out.println("index:" + index);  try {  Thread.sleep(200);  } catch (Exception e) {  // TODO: handle exception  }  }  });  } |

注意: 结果依次输出，相当于顺序执行各个任务。

线程池原理剖析

提交一个任务到线程池中，线程池的处理流程如下：

1、判断线程池里的核心线程是否都在执行任务，如果不是（核心线程空闲或者还有核心线程没有被创建）则创建一个新的工作线程来执行任务。如果核心线程都在执行任务，则进入下个流程。

2、线程池判断工作队列是否已满，如果工作队列没有满，则将新提交的任务存储在这个工作队列里。如果工作队列满了，则进入下个流程。

3、判断线程池里的线程是否都处于工作状态，如果没有，则创建一个新的工作线程来执行任务。如果已经满了，则交给饱和策略来处理这个任务。

合理配置线程池

要想合理的配置线程池，就必须首先分析任务特性，可以从以下几个角度来进行分析：

任务的性质：CPU密集型任务，IO密集型任务和混合型任务。

任务的优先级：高，中和低。

任务的执行时间：长，中和短。

任务的依赖性：是否依赖其他系统资源，如数据库连接。

任务性质不同的任务可以用不同规模的线程池分开处理。CPU密集型任务配置尽可能少的线程数量，如配置Ncpu+1个线程的线程池。IO密集型任务则由于需要等待IO操作，线程并不是一直在执行任务，则配置尽可能多的线程，如2\*Ncpu。混合型的任务，如果可以拆分，则将其拆分成一个CPU密集型任务和一个IO密集型任务，只要这两个任务执行的时间相差不是太大，那么分解后执行的吞吐率要高于串行执行的吞吐率，如果这两个任务执行时间相差太大，则没必要进行分解。我们可以通过Runtime.getRuntime().availableProcessors()方法获得当前设备的CPU个数。

优先级不同的任务可以使用优先级队列PriorityBlockingQueue来处理。它可以让优先级高的任务先得到执行，需要注意的是如果一直有优先级高的任务提交到队列里，那么优先级低的任务可能永远不能执行。

执行时间不同的任务可以交给不同规模的线程池来处理，或者也可以使用优先级队列，让执行时间短的任务先执行。

依赖数据库连接池的任务，因为线程提交SQL后需要等待数据库返回结果，如果等待的时间越长CPU空闲时间就越长，那么线程数应该设置越大，这样才能更好的利用CPU。

一般总结哦，有其他更好的方式，希望各位留言，谢谢。

CPU密集型时，任务可以少配置线程数，大概和机器的cpu核数相当，这样可以使得每个线程都在执行任务

IO密集型时，大部分线程都阻塞，故需要多配置线程数，2\*cpu核数

操作系统之名称解释：

某些进程花费了绝大多数时间在计算上，而其他则在等待I/O上花费了大多是时间，

前者称为计算密集型（CPU密集型）computer-bound，后者称为I/O密集型，I/O-bound。

什么是Disruptor