**线程与并发**

# 线程与进程区别

每个正在系统上运行的程序都是一个进程。每个进程包含一到多个线程。线程是一组指令的集合，或者是程序的特殊段，它可以在程序里独立执行。也可以把它理解为代码运行的上下文。所以线程基本上是轻量级的进程，它负责在单个程序里执行多任务。通常由操作系统负责多个线程的调度和执行。

使用线程可以把占据时间长的程序中的任务放到后台去处理，程序的运行速度可能加快，在一些等待的任务实现上如用户输入、文件读写和网络收发数据等，线程就比较有用了。在这种情况下可以释放一些珍贵的资源如内存占用等等。

如果有大量的线程,会影响性能，因为操作系统需要在它们之间切换，更多的线程需要更多的内存空间，线程的中止需要考虑其对程序运行的影响。通常块模型数据是在多个线程间共享的，需要防止线程死锁情况的发生。

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。

# 为什么要使用多线程

思考生活问题？

现在有一千克的水，目前只有小明一个人去打水，但是小明每小时打水200千克，现在要求一小时内把水全部打完，请问怎么解决？

如果小明一个人将水全部打完需要五个小时。

解决办法:

在加四个人同时打水，分别为小军、小红、小玲、小小，加上小明一共五个人同时去打水，五个人每小时打水200千克，那么一小时后就可以完成打完一顿水。

总结:多线程的好处提高程序的效率。

# 多线程应用场景

答:主要能体现到多线程提高程序效率。

举例: 迅雷多线程下载、分批发送短信等、定时任务大量数据处理

# 守护线程

Java分为两种线程：用户线程和守护线程

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。在使用守护线程时需要注意一下几点：

(1) thread.setDaemon(true)必须在thread.start()之前设置，否则会跑出一个IllegalThreadStateException异常。你不能把正在运行的常规线程设置为守护线程。

(2) 在Daemon线程中产生的新线程也是Daemon的。

(3) 守护线程应该永远不去访问固有资源，如文件、数据库，因为它会在任何时候甚至在一个操作的中间发生中断。

# 多线程创建方式

## 第一种继承Thread类 重写run方法

|  |
| --- |
| 代码: |
| class CreateThread extends Thread {  // run方法中编写 多线程需要执行的代码  publicvoid run() {  for (inti = 0; i< 10; i++) {  System.*out*.println("i:" + i);  }  }  }  publicclass ThreadDemo {  publicstaticvoid main(String[] args) {  System.*out*.println("-----多线程创建开始-----");  // 1.创建一个线程  CreateThread createThread = new CreateThread();  // 2.开始执行线程 注意 开启线程不是调用run方法，而是start方法  System.*out*.println("-----多线程创建启动-----");  createThread.start();  System.*out*.println("-----多线程创建结束-----");  }  } |

允许结果:



调用start方法后，代码并没有从上往下执行，而是有一条新的执行分之

。

注意：画图演示多线程不同执行路径。

## 第二种实现Runnable接口,重写run方法

代码:

|  |
| --- |
| class CreateRunnable implements Runnable {  @Override  publicvoid run() {  for (inti = 0; i< 10; i++) {  System.*out*.println("i:" + i);  }  }  }  publicclass ThreadDemo2 {  publicstaticvoid main(String[] args) {  System.*out*.println("-----多线程创建开始-----");  // 1.创建一个线程  CreateRunnable createThread = new CreateRunnable();  // 2.开始执行线程 注意 开启线程不是调用run方法，而是start方法  System.*out*.println("-----多线程创建启动-----");  Thread thread = new Thread(createThread);  thread.start();  System.*out*.println("-----多线程创建结束-----");  }  } |

## 第三种使用匿名内部类方式

|  |
| --- |
| System.*out*.println("-----多线程创建开始-----");  Thread thread = new Thread(new Runnable() {  public void run() {  for (int i = 0; i< 10; i++) {  System.*out*.println("i:" + i);  }  }  });  thread.start();  System.*out*.println("-----多线程创建结束-----"); |

## 使用继承Thread类还是使用实现Runnable接口好？

使用实现实现Runnable接口好，原因实现了接口还可以继续继承，继承了类不能再继承。

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。

## 启动线程是使用调用start方法还是run方法？

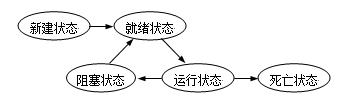
开始执行线程 注意 开启线程不是调用run方法，而是start方法

调用run知识使用实例调用方法。

# 获取线程对象以及名称

|  |  |
| --- | --- |
| 常用线程api方法 | |
| start() | 启动线程 |
| currentThread() | 获取当前线程对象 |
| getID() | 获取当前线程ID Thread-编号  该编号从0开始 |
| getName() | 获取当前线程名称 |
| sleep(long mill) | 休眠线程 |
| Stop（） | 停止线程 |
| 常用线程构造函数 | |
| Thread（） | 分配一个新的 Thread 对象 |
| Thread（String name） | 分配一个新的 Thread对象，具有指定的 name正如其名。 |
| Thread（Runable r） | 分配一个新的 Thread对象 |
| Thread（Runable r, String name） | 分配一个新的 Thread对象 |

# 多线程运行状态



线程从创建、运行到结束总是处于下面五个状态之一：新建状态、就绪状态、运行状态、阻塞状态及死亡状态。

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();

## 新建状态

   当用new操作符创建一个线程时， 例如new Thread(r)，线程还没有开始运行，此时线程处在新建状态。 当一个线程处于新生状态时，程序还没有开始运行线程中的代码

## 就绪状态

一个新创建的线程并不自动开始运行，要执行线程，必须调用线程的start()方法。当线程对象调用start()方法即启动了线程，start()方法创建线程运行的系统资源，并调度线程运行run()方法。当start()方法返回后，线程就处于就绪状态。

     处于就绪状态的线程并不一定立即运行run()方法，线程还必须同其他线程竞争CPU时间，只有获得CPU时间才可以运行线程。因为在单CPU的计算机系统中，不可能同时运行多个线程，一个时刻仅有一个线程处于运行状态。因此此时可能有多个线程处于就绪状态。对多个处于就绪状态的线程是由[Java](http://lib.csdn.net/base/java" \t "_blank" \o "Java 知识库)运行时系统的线程调度程序(*thread scheduler*)来调度的。

## 运行状态

当线程获得CPU时间后，它才进入运行状态，真正开始执行run()方法.

## 阻塞状态

    线程运行过程中，可能由于各种原因进入阻塞状态:  
        1>线程通过调用sleep方法进入睡眠状态；  
        2>线程调用一个在I/O上被阻塞的操作，即该操作在输入输出操作完成之前不会返回到它的调用者；  
        3>线程试图得到一个锁，而该锁正被其他线程持有；  
        4>线程在等待某个触发条件；

## 死亡状态

有两个原因会导致线程死亡：  
  1) run方法正常退出而自然死亡，  
   2) 一个未捕获的异常终止了run方法而使线程猝死。  
  为了确定线程在当前是否存活着（就是要么是可运行的，要么是被阻塞了），需要使用isAlive方法。如果是可运行或被阻塞，这个方法返回true； 如果线程仍旧是new状态且不是可运行的， 或者线程死亡了，则返回false.

# 多线程分批处理数据

需求:目前蚂蚁课堂有10万个用户，现在蚂蚁课堂需要做活动，给每一个用户发送一条祝福短信。

为了提高数程序的效率，请使用多线程技术分批发送据。

每开一个线程，都会占用CPU资源

服务器（电脑）配置 CPU 核数

## 新建用户实体类

|  |
| --- |
| publicclass UserEntity {  private String userId;  private String userName;  public String getUserId() {  returnuserId;  }  publicvoid setUserId(String userId) {  this.userId = userId;  }  public String getUserName() {  return userName;  }  publicvoid setUserName(String userName) {  this.userName = userName;  }  } |

## 建立多线程UserThread 执行发送短信

|  |
| --- |
| Class UserThread extends Thread {  private List<UserEntity>list;  /\*\*  \* 通过构造函数 传入每个线程需要执行的发送短信内容  \* @param list  \*/  public UserThread(List<UserEntity>list) {  this.list = list;  }  publicvoid run() {  for (UserEntity userEntity : list) {  System.*out*.println("threadName:" + Thread.*currentThread*().getName() + "-学员编号:" + userEntity.getUserId()  + "---学员名称:" + userEntity.getUserName());  // 调用发送短信具体代码  }  }  } |

## 初始化数据

|  |
| --- |
| publicstatic List<UserEntity> init() {  List<UserEntity>list = new ArrayList<UserEntity>();  for (inti = 1; i<= 140; i++) {  UserEntity userEntity = new UserEntity();  userEntity.setUserId("userId" + i);  userEntity.setUserName("userName" + i);  list.add(userEntity);  }  returnlist;  } |

## 计算分页工具类

|  |
| --- |
| Public class ListUtils {  /\*\*  \* @methodDesc: 功能描述:(list 集合分批切割)  \* @param: @param list  \* @param: @param pageSize  \* @param: @return  \* @createTime:2017年8月7日 下午9:30:59  \* @returnType:@param list 切割集合  \* @returnType:@param pageSize 分页长度  \* @returnType:@return List<List<T>> 返回分页数据  \*/  Static public<T> List<List<T>> splitList(List<T>list, intpageSize) {  intlistSize = list.size();  intpage = (listSize + (pageSize - 1)) / pageSize;  List<List<T>>listArray = new ArrayList<List<T>>();  for (inti = 0; i<page; i++) {  List<T>subList = new ArrayList<T>();  for (intj = 0; j<listSize; j++) {  intpageIndex = ((j + 1) + (pageSize - 1)) / pageSize;  if (pageIndex == (i + 1)) {  subList.add(list.get(j));  }  if ((j + 1) == ((j + 1) \* pageSize)) {  break;  }  }  listArray.add(subList);  }  returnlistArray;  }  } |

## 实现发送短信

|  |
| --- |
| Public staticvoid main(String[] args) {  // 1.初始化用户数据  List<UserEntity>listUserEntity = *init*();  // 2.计算创建创建多少个线程并且每一个线程需要执行“分批发送短信用户”  // 每一个线程分批跑多少  intuserThreadPage = 50;  // 计算所有线程数  List<List<UserEntity>>splitUserList = ListUtils.*splitList*(listUserEntity, userThreadPage);  intthreadSaze = splitUserList.size();  for (inti = 0; i<threadSaze; i++) {  List<UserEntity>list = splitUserList.get(i);  UserThread userThread = new UserThread(list);  // 3.执行任务发送短信  userThread.start();  }  } |

# 什么是线程安全

## 为什么有线程安全问题

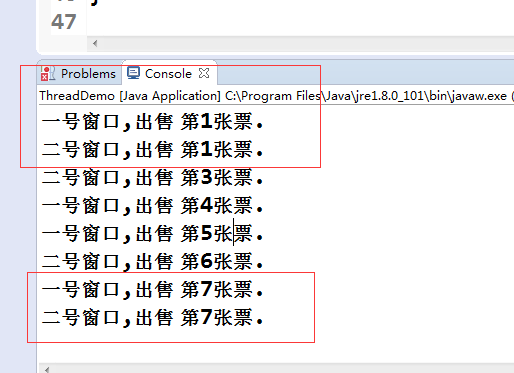
当多个线程同时共享，同一个全局变量或静态变量，做写的操作时，可能会发生数据冲突问题，也就是线程安全问题。但是做读操作是不会发生数据冲突问题。

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。

代码:

|  |
| --- |
| class ThreadTrain1 implements Runnable {  private int count = 100;  private static Object *oj* = new Object();  @Override  public void run() {  while (count > 0) {  try {  Thread.*sleep*(50);  } catch (Exception e) {  // TODO: handle exception  }  sale();  }  }  public void sale() {  // 前提 多线程进行使用、多个线程只能拿到一把锁。  // 保证只能让一个线程 在执行 缺点效率降低  // synchronized (oj) {  // if (count > 0) {  System.*out*.println(Thread.*currentThread*().getName() + ",出售第" + (100 - count + 1) + "票");  count--;  // }  // }  }  }  public class ThreadDemo {  public static void main(String[] args) {  ThreadTrain1 threadTrain1 = new ThreadTrain1();  Thread t1 = new Thread(threadTrain1, "①号窗口");  Thread t2 = new Thread(threadTrain1, "②号窗口");  t1.start();  t2.start();  }  } |

运行结果:



一号窗口和二号窗口同时出售火车第一张和第七张,部分火车票会重复出售。

结论发现，多个线程共享同一个全局成员变量时，做写的操作可能会发生数据冲突问题。

# 线程安全解决办法:

问:如何解决多线程之间线程安全问题?

答:使用多线程之间同步synchronized或使用锁(lock)。

问:为什么使用线程同步或使用锁能解决线程安全问题呢？

答:将可能会发生数据冲突问题(线程不安全问题)，只能让当前一个线程进行执行。代码执行完成后释放锁，让后才能让其他线程进行执行。这样的话就可以解决线程不安全问题。

问:什么是多线程之间同步？

答:当多个线程共享同一个资源,不会受到其他线程的干扰。

## 同步代码块

什么是同步代码块？

答:就是将可能会发生线程安全问题的代码，给包括起来。

synchronized(同一个数据){

可能会发生线程冲突问题

}

就是同步代码块

synchronized(对象)//这个对象可以为任意对象

{

    需要被同步的代码

}

对象如同锁，持有锁的线程可以在同步中执行

没持有锁的线程即使获取CPU的执行权，也进不去

同步的前提：

1，必须要有两个或者两个以上的线程

2，必须是多个线程使用同一个锁

必须保证同步中只能有一个线程在运行

好处：解决了多线程的安全问题

弊端：多个线程需要判断锁，较为消耗资源、抢锁的资源。

代码样例:

|  |
| --- |
| private static Object oj = new Object();  public void sale() {  // 前提 多线程进行使用、多个线程只能拿到一把锁。  // 保证只能让一个线程 在执行 缺点效率降低  synchronized (*oj*) {  if (count > 0) {  System.*out*.println(Thread.*currentThread*().getName() + ",出售第" + (100 - count + 1) + "票");  count--;  }  }  } |

## 同步函数

什么是同步函数？

答：在方法上修饰synchronized 称为同步函数

代码样例

|  |
| --- |
| public synchronized void sale() {  if (trainCount > 0) {  try {  Thread.*sleep*(40);  } catch (Exception e) {  }  System.*out*.println(Thread.*currentThread*().getName() + ",出售 第" + (100 - trainCount + 1) + "张票.");  trainCount--;  }  } |

思考问题？同步函数用的是什么锁？

答：同步函数使用this锁。

证明方式: 一个线程使用同步代码块(this明锁),另一个线程使用同步函数。如果两个线程抢票不能实现同步，那么会出现数据错误。

代码:

|  |
| --- |
| package com.itmayiedu;  class ThreadTrain2 implements Runnable {  private int count = 100;  public boolean flag = true;  private static Object *oj* = new Object();  @Override  public void run() {  if (flag) {  while (count > 0) {  synchronized (this) {  if (count > 0) {  try {  Thread.*sleep*(50);  } catch (Exception e) {  // TODO: handle exception  }  System.*out*.println(Thread.*currentThread*().getName() + ",出售第" + (100 - count + 1) + "票");  count--;  }  }  }  } else {  while (count > 0) {  sale();  }  }  }  public synchronized void sale() {  // 前提 多线程进行使用、多个线程只能拿到一把锁。  // 保证只能让一个线程 在执行 缺点效率降低  // synchronized (oj) {  if (count > 0) {  try {  Thread.*sleep*(50);  } catch (Exception e) {  // TODO: handle exception  }  System.*out*.println(Thread.*currentThread*().getName() + ",出售第" + (100 - count + 1) + "票");  count--;  }  // }  }  }  public class ThreadDemo2 {  public static void main(String[] args) throws InterruptedException {  ThreadTrain2 threadTrain1 = new ThreadTrain2();  Thread t1 = new Thread(threadTrain1, "①号窗口");  Thread t2 = new Thread(threadTrain1, "②号窗口");  t1.start();  Thread.*sleep*(40);  threadTrain1.flag = false;  t2.start();  }  } |

## 静态同步函数

答：什么是静态同步函数？

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

可以用 getClass方法获取，也可以用当前 类名.class 表示。

代码样例:

|  |
| --- |
| synchronized (ThreadTrain.class) {  System.*out*.println(Thread.*currentThread*().getName() + ",出售 第" + (100 - *trainCount* + 1) + "张票.");  *trainCount*--;  try {  Thread.*sleep*(100);  } catch (Exception e) {  }  } |

总结：

synchronized 修饰方法使用锁是当前this锁。

synchronized 修饰静态方法使用锁是当前类的字节码文件

# 多线程死锁

## 什么是多线程死锁？

答:同步中嵌套同步,导致锁无法释放

代码:

|  |
| --- |
| package com.itmayiedu;  class ThreadTrain6 implements Runnable {  // 这是货票总票数,多个线程会同时共享资源  private int trainCount = 100;  public boolean flag = true;  private Object mutex = new Object();  @Override  public void run() {  if (flag) {  while (true) {  synchronized (mutex) {  // 锁(同步代码块)在什么时候释放？ 代码执行完， 自动释放锁.  // 如果flag为true 先拿到 obj锁,在拿到this 锁、 才能执行。  // 如果flag为false先拿到this,在拿到obj锁，才能执行。  // 死锁解决办法:不要在同步中嵌套同步。  sale();  }  }  } else {  while (true) {  sale();  }  }  }  /\*\*  \* @methodDesc: 功能描述:(出售火车票)  \*/  public synchronized void sale() {  synchronized (mutex) {  if (trainCount > 0) {  try {  Thread.*sleep*(40);  } catch (Exception e) {  }  System.*out*.println(Thread.*currentThread*().getName() + ",出售 第" + (100 - trainCount + 1) + "张票.");  trainCount--;  }  }  }  }  public class DeadlockThread {  public static void main(String[] args) throws InterruptedException {  ThreadTrain6 threadTrain = new ThreadTrain6(); // 定义 一个实例  Thread thread1 = new Thread(threadTrain, "一号窗口");  Thread thread2 = new Thread(threadTrain, "二号窗口");  thread1.start();  Thread.*sleep*(40);  threadTrain.flag = false;  thread2.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

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

## 什么是Volatile

Volatile 关键字的作用是变量在多个线程之间可见。

代码:

|  |
| --- |
| 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 class ThreadVolatile {  public static void main(String[] args) throws InterruptedException {  ThreadVolatileDemo threadVolatileDemo = new ThreadVolatileDemo();  threadVolatileDemo.start();  Thread.*sleep*(3000);  threadVolatileDemo.setRuning(false);  System.*out*.println("flag 已经设置成false");  Thread.*sleep*(1000);  System.*out*.println(threadVolatileDemo.flag);  }  } |

运行结果:

C:\Users\ADMINI~1\AppData\Local\Temp\ksohtml\wpsB78E.tmp.jpg

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

## Volatile非原子性

注意: Volatile非原子性

|  |
| --- |
| public class VolatileNoAtomic extends Thread {  private static volatile int *count*;  // private static AtomicInteger count = new AtomicInteger(0);  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:

|  |
| --- |
| C:\Users\ADMINI~1\AppData\Local\Temp\ksohtml\wpsB79F.tmp.jpg |

结果发现 数据不同步，因为Volatile不用具备原子性。

## 使用AtomicInteger原子类

AtomicInteger是一个提供原子操作的Integer类，通过线程安全的方式操作加减。

|  |
| --- |
| public class VolatileNoAtomic extends Thread {  static int *count* = 0;  private static AtomicInteger *atomicInteger* = new AtomicInteger(0);  @Override  public void run() {  for (int i = 0; i < 1000; i++) {  //等同于i++  *atomicInteger*.incrementAndGet();  }  System.*out*.println(*count*);  }  public static void main(String[] args) {  // 初始化10个线程  VolatileNoAtomic[] volatileNoAtomic = new VolatileNoAtomic[10];  for (int i = 0; i < 10; i++) {  // 创建  volatileNoAtomic[i] = new VolatileNoAtomic();  }  for (int i = 0; i < volatileNoAtomic.length; i++) {  volatileNoAtomic[i].start();  }  }  } |

## 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

## 什么是ThreadLocal

ThreadLocal提供一个线程的局部变量，访问某个线程拥有自己局部变量。

当使用ThreadLocal维护变量时，ThreadLocal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。

ThreadLocal的接口方法

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，但它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。

案例:创建三个线程，每个线程生成自己独立序列号。

代码:

|  |
| --- |
| 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;  }  }  public class ThreadLocaDemo2 extends Thread {  private Res res;  public ThreadLocaDemo2(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();  ThreadLocaDemo2 threadLocaDemo1 = new ThreadLocaDemo2(res);  ThreadLocaDemo2 threadLocaDemo2 = new ThreadLocaDemo2(res);  ThreadLocaDemo2 threadLocaDemo3 = new ThreadLocaDemo2(res);  threadLocaDemo1.start();  threadLocaDemo2.start();  threadLocaDemo3.start();  }  } |

## ThreadLoca实现原理

ThreadLocl通过map集合

Map.put(“当前线程”,值)；

多线程之间实现通讯

# 多线程之间如何实现通讯

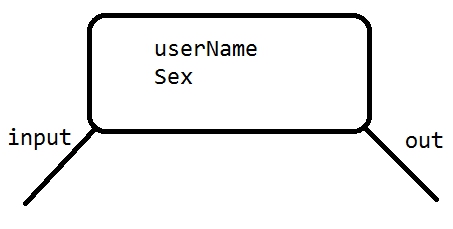
## 什么是多线程之间通讯？

多线程之间通讯，其实就是多个线程在操作同一个资源，但是操作的动作不同。

画图演示

## 多线程之间通讯需求

需求:第一个线程写入(input)用户，另一个线程取读取(out)用户.实现读一个，写一个操作。



### 共享资源源实体类

|  |
| --- |
| class Res {  public String userSex;  public String userName;  } |

### 输入线程资源

|  |
| --- |
| class IntThrad extends Thread {  private Res res;  public IntThrad(Res res) {  this.res = res;  }  @Override  public void run() {  int count = 0;  while (true) {  if (count == 0) {  res.userName = "余胜军";  res.userSex = "男";  } else {  res.userName = "小紅";  res.userSex = "女";  }  count = (count + 1) % 2;  }  }  } |

### 输出线程

|  |
| --- |
| class OutThread extends Thread {  private Res res;  public OutThread(Res res) {  this.res = res;  }  @Override  public void run() {  while (true) {  System.*out*.println(res.userName + "--" + res.userSex);  }  }  } |

### 运行代码

|  |
| --- |
| Res res = new Res();  IntThrad intThrad = new IntThrad(res);  OutThread outThread = new OutThread(res);  intThrad.start();  outThread.start(); |

### 运行代码

|  |
| --- |
| C:\Users\ADMINI~1\AppData\Local\Temp\ksohtml\wps9026.tmp.jpg |

注意：数据发生错乱，造成线程安全问题

### 解决线程安全问题

#### IntThrad 加上synchronized

|  |
| --- |
| class IntThrad extends Thread {  private Res res;  public IntThrad(Res res) {  this.res = res;  }  @Override  public void run() {  int count = 0;  while (true) {  synchronized (res) {  if (count == 0) {  res.userName = "余胜军";  res.userSex = "男";  } else {  res.userName = "小紅";  res.userSex = "女";  }  count = (count + 1) % 2;  }  }  }  } |

#### 输出线程加上synchronized

|  |
| --- |
| class Res {  public String userName;  public String sex;  }  class InputThread extends Thread {  private Res res;  public InputThread(Res res) {  this.res = res;  }  @Override  public void run() {  int count = 0;  while (true) {   synchronized (res) {  if (count == 0) {  res.userName = "余胜军";  res.sex = "男";  } else {  res.userName = "小红";  res.sex = "女";  }  count = (count + 1) % 2;  }  }  }  }  class OutThrad extends Thread {  private Res res;  public OutThrad(Res res) {  this.res = res;  }  @Override  public void run() {  while (true) {  synchronized (res) {  System.*out*.println(res.userName + "," + res.sex);  }  }  }  }  public class ThreadDemo01 {  public static void main(String[] args) {  Res res = new Res();  InputThread inputThread = new InputThread(res);  OutThrad outThrad = new OutThrad(res);  inputThread.start();  outThrad.start();  }  } |

# wait()、notify、notifyAll()方法

wait()、notify()、notifyAll()是三个定义在Object类里的方法，可以用来控制线程的状态。

这三个方法最终调用的都是jvm级的native方法。随着jvm运行平台的不同可能有些许差异。

如果对象调用了wait方法就会使持有该对象的线程把该对象的控制权交出去，然后处于等待状态。

如果对象调用了notify方法就会通知某个正在等待这个对象的控制权的线程可以继续运行。

如果对象调用了notifyAll方法就会通知所有等待这个对象控制权的线程继续运行。

注意:一定要在线程同步中使用,并且是同一个锁的资源

|  |
| --- |
| class Res {  public String userSex;  public String userName;  //线程通讯标识  public boolean flag = false;  } |

|  |
| --- |
| class IntThrad extends Thread {  private Res res;  public IntThrad(Res res) {  this.res = res;  }  @Override  public void run() {  int count = 0;  while (true) {  synchronized (res) {  if (res.flag) {  try {  // 当前线程变为等待，但是可以释放锁  res.wait();  } catch (Exception e) {  }  }  if (count == 0) {  res.userName = "余胜军";  res.userSex = "男";  } else {  res.userName = "小紅";  res.userSex = "女";  }  count = (count + 1) % 2;  res.flag = true;  // 唤醒当前线程  res.notify();  }  }  }  } |

|  |
| --- |
| class OutThread extends Thread {  private Res res;  public OutThread(Res res) {  this.res = res;  }  @Override  public void run() {  while (true) {  synchronized (res) {  if (!res.flag) {  try {  res.wait();  } catch (Exception e) {  // TODO: handle exception  }  }  System.*out*.println(res.userName + "--" + res.userSex);  res.flag = false;  res.notify();  }  }  }  } |

|  |
| --- |
| public class ThreaCommun {  public static void main(String[] args) {  Res res = new Res();  IntThrad intThrad = new IntThrad(res);  OutThread outThread = new OutThread(res);  intThrad.start();  outThread.start();  }  } |

# 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。

# 生产者与消费者模式

todo

# JDK1.5-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。

## Lock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.ublock();  } |

## Lock 接口与 synchronized 关键字的区别

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。

## Condition用法

Condition的功能类似于在传统的线程技术中的,Object.wait()和Object.notify()的功能。

### 代码

|  |
| --- |
| Condition condition = lock.newCondition();  res. condition.await(); 类似wait  res. Condition. Signal() 类似notify |

|  |
| --- |
| class Res {  public String userName;  public String sex;  public boolean flag = false;  Lock lock = new ReentrantLock();  }  class InputThread extends Thread {  private Res res;  Condition newCondition;  public InputThread(Res res, Condition newCondition) {  this.res = res;  this.newCondition=newCondition;  }  @Override  public void run() {  int count = 0;  while (true) {  // synchronized (res) {  try {  res.lock.lock();  if (res.flag) {  try {  // res.wait();  newCondition.await();  } catch (Exception e) {  // TODO: handle exception  }  }  if (count == 0) {  res.userName = "余胜军";  res.sex = "男";  } else {  res.userName = "小红";  res.sex = "女";  }  count = (count + 1) % 2;  res.flag = true;  // res.notify();  newCondition.signal();  } catch (Exception e) {  // TODO: handle exception  }finally {  res.lock.unlock();  }  }  // }  }  }  class OutThrad extends Thread {  private Res res;  private Condition newCondition;  public OutThrad(Res res,Condition newCondition) {  this.res = res;  this.newCondition=newCondition;  }  @Override  public void run() {  while (true) {  // synchronized (res) {  try {  res.lock.lock();  if (!res.flag) {  try {  // res.wait();  newCondition.await();  } catch (Exception e) {  // TODO: handle exception  }  }  System.*out*.println(res.userName + "," + res.sex);  res.flag = false;  // res.notify();  newCondition.signal();  } catch (Exception e) {  // TODO: handle exception  }finally {  res.lock.unlock();  }  // }  }  }  }  public class ThreadDemo01 {  public static void main(String[] args) {  Res res = new Res();  Condition newCondition = res.lock.newCondition();  InputThread inputThread = new InputThread(res,newCondition);  OutThrad outThrad = new OutThrad(res,newCondition);  inputThread.start();  outThrad.start();  }  } |

# 如何停止线程？

## 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。

代码:

|  |
| --- |
| class StopThread implements Runnable {  private boolean flag = true;  @Override  public synchronized void run() {  while (flag) {  try {  wait();  } catch (Exception e) {  //e.printStackTrace();  stopThread();  }  System.*out*.println("thread run..");  }  }  public void stopThread() {  flag = false;  }  }    public class StopThreadDemo {  public static void main(String[] args) {  StopThread stopThread1 = new StopThread();  Thread thread1 = new Thread(stopThread1);  Thread thread2 = new Thread(stopThread1);  thread1.start();  thread2.start();  int i = 0;  while (true) {  System.*out*.println("thread main..");  if (i == 300) {  // stopThread1.stopThread();  thread1.interrupt();  thread2.interrupt();  break;  }  i++;  }  }  } |

* 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，但它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。

案例:创建三个线程，每个线程生成自己独立序列号。

代码:

|  |
| --- |
| 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;  }  }  public class ThreadLocaDemo2 extends Thread {  private Res res;  public ThreadLocaDemo2(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();  ThreadLocaDemo2 threadLocaDemo1 = new ThreadLocaDemo2(res);  ThreadLocaDemo2 threadLocaDemo2 = new ThreadLocaDemo2(res);  ThreadLocaDemo2 threadLocaDemo3 = new ThreadLocaDemo2(res);  threadLocaDemo1.start();  threadLocaDemo2.start();  threadLocaDemo3.start();  }  } |

# (屏障)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();  }  }  } |

# (计数器)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。CountDownLatch是通过一个计数器来实现的，计数器的初始值为线程的数量。每当一个线程完成了自己的任务后，计数器的值就会减1。当计数器值到达0时，它表示所有的线程已经完成了任务，然后在闭锁上等待的线程就可以恢复执行任务。

|  |
| --- |
| public static void main(String[] args) throws InterruptedException {  CountDownLatch countDownLatch = new CountDownLatch(2);  new Thread(new Runnable() {  @Override  public void run() {  System.out.println(Thread.currentThread().getName() + ",子线程开始执行...");  countDownLatch.countDown();  System.out.println(Thread.currentThread().getName() + ",子线程结束执行...");  }  }).start();    new Thread(new Runnable() {  @Override  public void run() {  System.out.println(Thread.currentThread().getName() + ",子线程开始执行...");  countDownLatch.countDown();//计数器值每次减去1  System.out.println(Thread.currentThread().getName() + ",子线程结束执行...");  }  }).start();  countDownLatch.await();// 減去为0,恢复任务继续执行  System.out.println("两个子线程执行完毕....");  System.out.println("主线程继续执行.....");  for (int i = 0; i <10; i++) {  System.out.println("main,i:"+i);  }  } |

# (计数信号量)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 ThradDemo001 extends Thread {  private String name;  private Semaphore wc;  public ThradDemo001(String name, Semaphore wc) {  this.name = name;  this.wc = wc;  }  @Override  public void run() {  // 剩下的资源  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...");  }  try {  // 申请资源  wc.acquire();  } catch (InterruptedException e) {  }  System.out.println(name + "终于上厕所啦.爽啊" + ",剩下厕所:" + wc.availablePermits());  try {  Thread.sleep(new Random().nextInt(1000));  } catch (Exception e) {  // TODO: handle exception  }  System.out.println(name + "厕所上完啦!");  // 释放资源  wc.release();  }  }  public class TestSemaphore {  public static void main(String[] args) {  Semaphore semaphore = new Semaphore(3);  for (int i = 1; i <= 10; i++) {  ThradDemo001 thradDemo001 = new ThradDemo001("第" + i + "个人", semaphore);  thradDemo001.start();  }  }  } |

# 同步容器类

## Vector与ArrayList区别

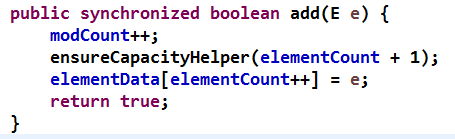
1.ArrayList是最常用的List实现类，内部是通过数组实现的，它允许对元素进行快速随机访问。数组的缺点是每个元素之间不能有间隔，当数组大小不满足时需要增加存储能力，就要讲已经有数组的数据复制到新的存储空间中。当从ArrayList的中间位置插入或者删除元素时，需要对数组进行复制、移动、代价比较高。因此，它适合随机查找和遍历，不适合插入和删除。

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

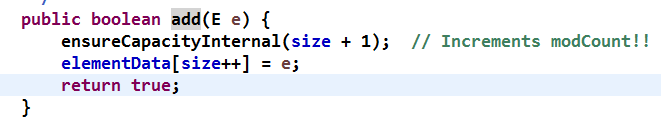
#### Vector源码类

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。

# 并发队列

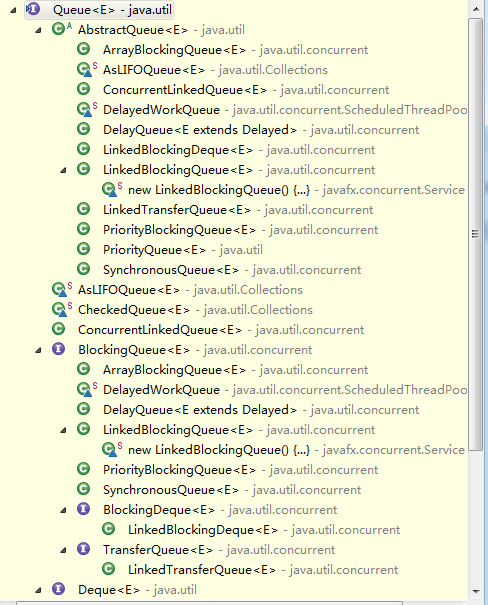
在并发队列上JDK提供了两套实现，一个是以ConcurrentLinkedQueue为代表的高性能队

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。

## 阻塞队列与非阻塞队

阻塞队列与普通队列的区别在于，当队列是空的时，从队列中获取元素的操作将会被阻塞，或者当队列是满时，往队列里添加元素的操作会被阻塞。试图从空的阻塞队列中获取元素的线程将会被阻塞，直到其他的线程往空的队列插入新的元素。同样，试图往已满的阻塞队列中添加新元素的线程同样也会被阻塞，直到其他的线程使队列重新变得空闲起来，如从队列中移除一个或者多个元素，或者完全清空队列.

1.ArrayDeque, （数组双端队列）   
2.PriorityQueue, （优先级队列）   
3.ConcurrentLinkedQueue, （基于链表的并发队列）   
4.DelayQueue, （延期阻塞队列）（阻塞队列实现了BlockingQueue接口）   
5.ArrayBlockingQueue, （基于数组的并发阻塞队列）   
6.LinkedBlockingQueue, （基于链表的FIFO阻塞队列）   
7.LinkedBlockingDeque, （基于链表的FIFO双端阻塞队列）   
8.PriorityBlockingQueue, （带优先级的无界阻塞队列）   
9.SynchronousQueue （并发同步阻塞队列）



## ConcurrentLinkedQueue

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）

## 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。

# Callable

在Java中，创建线程一般有两种方式，一种是继承Thread类，一种是实现Runnable接口。然而，这两种方式的缺点是在线程任务执行结束后，无法获取执行结果。我们一般只能采用共享变量或共享存储区以及线程通信的方式实现获得任务结果的目的。  
不过，Java中，也提供了使用Callable和Future来实现获取任务结果的操作。Callable用来执行任务，产生结果，而Future用来获得结果。

Callable接口与Runnable接口是否相似，查看源码，可知Callable接口的定义如下：

|  |
| --- |
| @FunctionalInterface  public interface Callable<V> {  /\*\*  \* Computes a result, or throws an exception if unable to do so.  \*  \* @return computed result  \* @throws Exception if unable to compute a result  \*/  V call() throws Exception;  } |

可以看到，与Runnable接口不同之处在于，call方法带有泛型返回值V。

## Future常用方法

V get() ：获取异步执行的结果，如果没有结果可用，此方法会阻塞直到异步计算完成。

V get(Long timeout , TimeUnit unit) ：获取异步执行结果，如果没有结果可用，此方法会阻塞，但是会有时间限制，如果阻塞时间超过设定的timeout时间，该方法将抛出异常。

boolean isDone() ：如果任务执行结束，无论是正常结束或是中途取消还是发生异常，都返回true。

boolean isCanceller() ：如果任务完成前被取消，则返回true。

boolean cancel(boolean mayInterruptRunning) ：如果任务还没开始，执行cancel(...)方法将返回false；如果任务已经启动，执行cancel(true)方法将以中断执行此任务线程的方式来试图停止任务，如果停止成功，返回true；当任务已经启动，执行cancel(false)方法将不会对正在执行的任务线程产生影响(让线程正常执行到完成)，此时返回false；当任务已经完成，执行cancel(...)方法将返回false。mayInterruptRunning参数表示是否中断执行中的线程。

通过方法分析我们也知道实际上Future提供了3种功能：（1）能够中断执行中的任务（2）判断任务是否执行完成（3）获取任务执行完成后额结果。

我们通过简单的例子来体会使用Callable和Future来获取任务结果的用法。

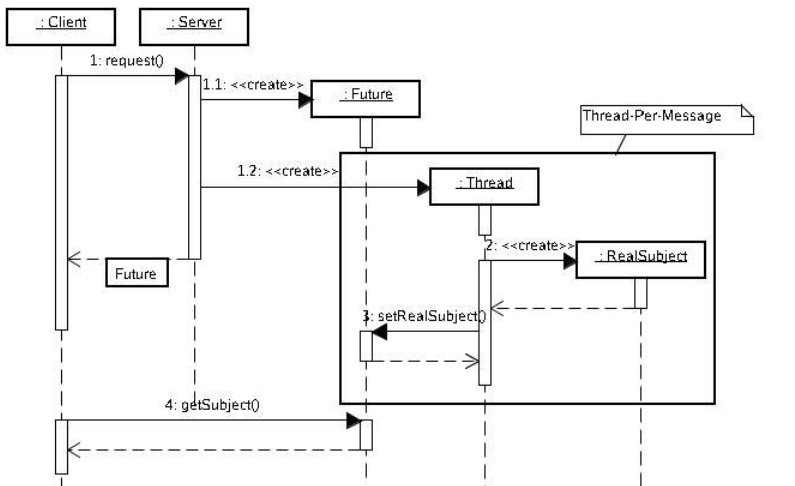
|  |
| --- |
| public class TestMain {  public static void main(String[] args) throws InterruptedException, ExecutionException {  ExecutorService executor = Executors.*newCachedThreadPool*();  Future<Integer> future = executor.submit(new AddNumberTask());  System.*out*.println(Thread.*currentThread*().getName() + "线程执行其他任务");  Integer integer = future.get();  System.*out*.println(integer);  // 关闭线程池  if (executor != null)  executor.shutdown();  }  }  class AddNumberTask implements Callable<Integer> {  public AddNumberTask() {  }  @Override  public Integer call() throws Exception {  System.*out*.println("####AddNumberTask###call()");  Thread.*sleep*(5000);  return 5000;  }  } |

# Future模式

Future模式的核心在于：去除了主函数的等待时间，并使得原本需要等待的时间段可以用于处理其他业务逻辑

Futrure模式:对于多线程，如果线程A要等待线程B的结果，那么线程A没必要等待B，直到B有结果，可以先拿到一个未来的Future，等B有结果是再取真实的结果。

　在多线程中经常举的一个例子就是：网络图片的下载，刚开始是通过模糊的图片来代替最后的图片，等下载图片的线程下载完图片后在替换。而在这个过程中可以做一些其他的事情。



首先客户端向服务器请求RealSubject，但是这个资源的创建是非常耗时的，怎么办呢？这种情况下，首先返回Client一个FutureSubject,以满足客户端的需求，于此同时呢，Future会通过另外一个Thread 去构造一个真正的资源，资源准备完毕之后，在给future一个通知。如果客户端急于获取这个真正的资源，那么就会阻塞客户端的其他所有线程，等待资源准备完毕。

公共数据接口，FutureData和RealData都要实现。

|  |
| --- |
| public interface Data {  public abstract String getRequest();  } |

FutureData,当有线程想要获取RealData的时候，程序会被阻塞。等到RealData被注入才会使用getReal()方法。

|  |
| --- |
| public class FurureData implements Data {  public volatile static boolean *ISFLAG* = false;  private RealData realData;  public synchronized void setRealData(RealData realData) {  // 如果已经获取到结果，直接返回  if (*ISFLAG*) {  return;  }  // 如果没有获取到数据,传递真是对象  this.realData = realData;  *ISFLAG* = true;  // 进行通知  notify();  }  @Override  public synchronized String getRequest() {  while (!*ISFLAG*) {  try {  wait();  } catch (Exception e) {  }  }  // 获取到数据,直接返回  return realData.getRequest();  }  } |

真实数据RealData

|  |
| --- |
| public class RealData implements Data {  private String result;  public RealData(String data) {  System.*out*.println("正在使用data:" + data + "网络请求数据,耗时操作需要等待.");  try {  Thread.*sleep*(3000);  } catch (Exception e) {  }  System.*out*.println("操作完毕,获取结果...");  result = "余胜军";  }  @Override  public String getRequest() {  return result;  } |

FutureClient 客户端

|  |
| --- |
| public class FutureClient {  public Data request(String queryStr) {  FurureData furureData = new FurureData();  new Thread(new Runnable() {  @Override  public void run() {  RealData realData = new RealData(queryStr);  furureData.setRealData(realData);  }  }).start();  return furureData;  }  } |

调用者：

|  |
| --- |
| public class Main {  public static void main(String[] args) {  FutureClient futureClient = new FutureClient();  Data request = futureClient.request("请求参数.");  System.*out*.println("请求发送成功!");  System.*out*.println("执行其他任务...");  String result = request.getRequest();  System.*out*.println("获取到结果..." + result);  }  } |

调用者请求资源，client.request("name"); 完成对数据的准备

当要获取资源的时候，data.getResult() ，如果资源没有准备好isReady = false;那么就会阻塞该线程。直到资源获取然后该线程被唤醒。

# 原子类

java.util.concurrent.atomic包：原子类的小工具包，支持在单个变量上解除锁的线程安全编程

## 原子变量类相当于一种泛化的 volatile 变量，能够支持原子的和有条件的读-改-写操作。AtomicInteger 表示一个int类型的值，并提供了 get 和 set 方法，这些 Volatile 类型的int变量在读取和写入上有着相同的内存语义。它还提供了一个原子的 compareAndSet 方法（如果该方法成功执行，那么将实现与读取/写入一个 volatile 变量相同的内存效果），以及原子的添加、递增和递减等方法。AtomicInteger 表面上非常像一个扩展的 Counter 类，但在发生竞争的情况下能提供更高的可伸缩性，因为它直接利用了硬件对并发的支持。

## 为什么会有原子类

## CAS：Compare and Swap，即比较再交换。

## jdk5增加了并发包java.util.concurrent.\*,其下面的类使用CAS算法实现了区别于synchronouse同步锁的一种乐观锁。JDK 5之前Java语言是靠synchronized关键字保证同步的，这是一种独占锁，也是是悲观锁。

## 如果同一个变量要被多个线程访问，则可以使用该包中的类

AtomicBoolean

AtomicInteger

AtomicLong

AtomicReference

## CAS无锁模式

### 什么是CAS

CAS：Compare and Swap，即比较再交换。

### jdk5增加了并发包java.util.concurrent.\*,其下面的类使用CAS算法实现了区别于synchronouse同步锁的一种乐观锁。JDK 5之前Java语言是靠synchronized关键字保证同步的，这是一种独占锁，也是是悲观锁。

### CAS算法理解

（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。

常用原子类

Java中的原子操作类大致可以分为4类：原子更新基本类型、原子更新数组类型、原子更新引用类型、原子更新属性类型。这些原子类中都是用了无锁的概念，有的地方直接使用CAS操作的线程安全的类型。

AtomicBoolean

AtomicInteger

AtomicLong

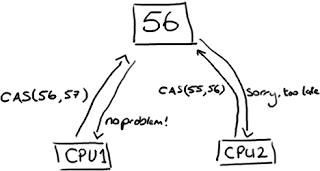
AtomicReference

|  |
| --- |
| public class Test0001 implements Runnable {  private static Integer *count* = 1;  private static AtomicInteger *atomic* = new AtomicInteger();  @Override  public void run() {  while (true) {  int count = getCountAtomic();  System.*out*.println(count);  if (count >= 150) {  break;  }  }  }  public synchronized Integer getCount() {  try {  Thread.*sleep*(50);  } catch (Exception e) {  // TODO: handle exception  }  return *count*++;  }  public Integer getCountAtomic() {  try {  Thread.*sleep*(50);  } catch (Exception e) {  // TODO: handle exception  }  return *atomic*.incrementAndGet();  }  public static void main(String[] args) {  Test0001 test0001 = new Test0001();  Thread t1 = new Thread(test0001);  Thread t2 = new Thread(test0001);  t1.start();  t2.start();  }  } |

### CAS（乐观锁算法）的基本假设前提

CAS比较与交换的伪代码可以表示为：

do{     
       备份旧数据；    
       基于旧数据构造新数据；    
}while(!CAS( 内存地址，备份的旧数据，新数据 ))

[](https://images0.cnblogs.com/blog/28306/201402/191145372245230.png)

（上图的解释：CPU去更新一个值，但如果想改的值不再是原来的值，操作就失败，因为很明显，有其它操作先改变了这个值。）

就是指当两者进行比较时，如果相等，则证明共享数据没有被修改，替换成新值，然后继续往下运行；如果不相等，说明共享数据已经被修改，放弃已经所做的操作，然后重新执行刚才的操作。容易看出 CAS 操作是基于共享数据不会被修改的假设，采用了类似于数据库的 commit-retry 的模式。当同步冲突出现的机会很少时，这种假设能带来较大的性能提升。

|  |
| --- |
| public final int getAndAddInt(Object o, long offset, int delta) {  int v;  do {  v = getIntVolatile(o, offset);  } while (!compareAndSwapInt(o, offset, v, v + delta));  return v;  } |

|  |
| --- |
| /\*\*  \* 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缺点

## CAS存在一个很明显的问题，即ABA问题。

## 问题：如果变量V初次读取的时候是A，并且在准备赋值的时候检查到它仍然是A，那能说明它的值没有被其他线程修改过了吗？

## 如果在这段期间曾经被改成B，然后又改回A，那CAS操作就会误认为它从来没有被修改过。针对这种情况，java并发包中提供了一个带有标记的原子引用类AtomicStampedReference，它可以通过控制变量值的版本来保证CAS的正确性。

# Java锁的深度化

## 悲观锁、乐观锁、排他锁

场景

当多个请求同时操作数据库时，首先将订单状态改为已支付，在金额加上200，在同时并发场景查询条件下，会造成重复通知。

SQL:

Update

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

## 重入锁

锁作为并发共享数据，保证一致性的工具，在JAVA平台有多种实现(如 synchronized 和 ReentrantLock等等 ) 。这些已经写好提供的锁为我们开发提供了便利。

重入锁，也叫做递归锁，指的是同一线程 外层函数获得锁之后 ，内层递归函数仍然有获取该锁的代码，但不受影响。  
在JAVA环境下 ReentrantLock 和synchronized 都是 可重入锁

|  |
| --- |
| public class Test implements Runnable {  public synchronized void get() {  System.*out*.println("name:" + Thread.*currentThread*().getName() + " get();");  set();  }  public synchronized void set() {  System.*out*.println("name:" + Thread.*currentThread*().getName() + " set();");  }  @Override  public void run() {  get();  }  public static void main(String[] args) {  Test ss = new Test();  new Thread(ss).start();  new Thread(ss).start();  new Thread(ss).start();  new Thread(ss).start();  }  } |

|  |
| --- |
| public class Test02 extends Thread {  ReentrantLock lock = new ReentrantLock();  public void get() {  lock.lock();  System.*out*.println(Thread.*currentThread*().getId());  set();  lock.unlock();  }  public void set() {  lock.lock();  System.*out*.println(Thread.*currentThread*().getId());  lock.unlock();  }  @Override  public void run() {  get();  }  public static void main(String[] args) {  Test ss = new Test();  new Thread(ss).start();  new Thread(ss).start();  new Thread(ss).start();  }  } |

## 读写锁

相比[Java中的锁(Locks in Java)](http://ifeve.com/locks/" \t "_blank)里Lock实现，读写锁更复杂一些。假设你的程序中涉及到对一些共享资源的读和写操作，且写操作没有读操作那么频繁。在没有写操作的时候，两个线程同时读一个资源没有任何问题，所以应该允许多个线程能在同时读取共享资源。但是如果有一个线程想去写这些共享资源，就不应该再有其它线程对该资源进行读或写（译者注：也就是说：读-读能共存，读-写不能共存，写-写不能共存）。这就需要一个读/写锁来解决这个问题。Java5在java.util.concurrent包中已经包含了读写锁。尽管如此，我们还是应该了解其实现背后的原理。

|  |
| --- |
| public class Cache {  static Map<String, Object> *map* = new HashMap<String, Object>();  static ReentrantReadWriteLock *rwl* = new ReentrantReadWriteLock();  static Lock *r* = *rwl*.readLock();  static Lock *w* = *rwl*.writeLock();  // 获取一个key对应的value  public static final Object get(String key) {  *r*.lock();  try {  System.*out*.println("正在做读的操作,key:" + key + " 开始");  Thread.*sleep*(100);  Object object = *map*.get(key);  System.*out*.println("正在做读的操作,key:" + key + " 结束");  System.*out*.println();  return object;  } catch (InterruptedException e) {  } finally {  *r*.unlock();  }  return key;  }  // 设置key对应的value，并返回旧有的value  public static final Object put(String key, Object value) {  *w*.lock();  try {  System.*out*.println("正在做写的操作,key:" + key + ",value:" + value + "开始.");  Thread.*sleep*(100);  Object object = *map*.put(key, value);  System.*out*.println("正在做写的操作,key:" + key + ",value:" + value + "结束.");  System.*out*.println();  return object;  } catch (InterruptedException e) {  } finally {  *w*.unlock();  }  return value;  }  // 清空所有的内容  public static final void clear() {  *w*.lock();  try {  *map*.clear();  } finally {  *w*.unlock();  }  }  public static void main(String[] args) {  new Thread(new Runnable() {  @Override  public void run() {  for (int i = 0; i < 10; i++) {  Cache.*put*(i + "", i + "");  }  }  }).start();  new Thread(new Runnable() {  @Override  public void run() {  for (int i = 0; i < 10; i++) {  Cache.*get*(i + "");  }  }  }).start();  }  } |

## CAS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。

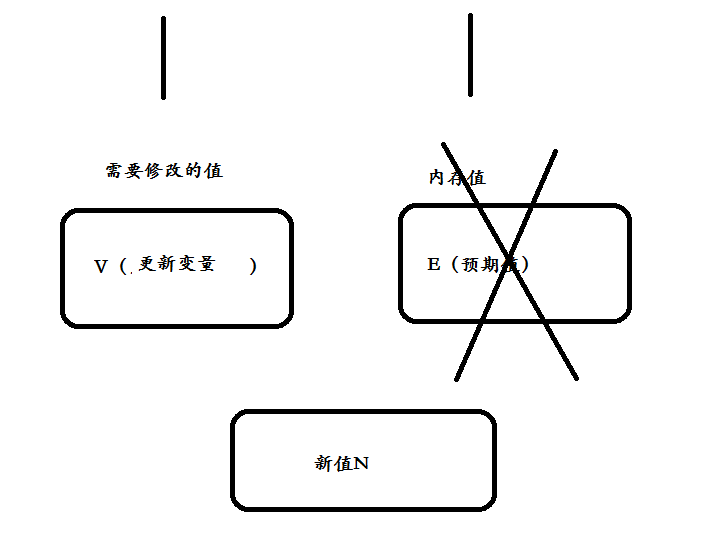
就凭借这两个优势，就值得我们冒险尝试使用无锁的并发。

（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;  }  } |

## 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

|  |
| --- |
| 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、Zookeeper分布式锁