## 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。

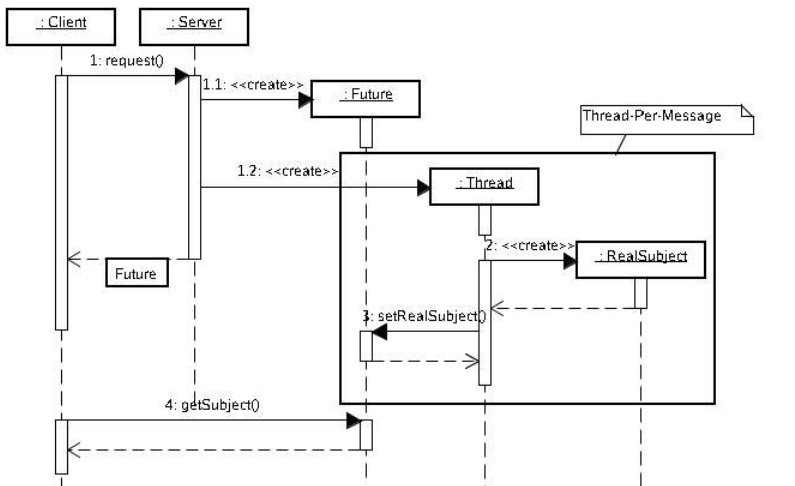
|  |
| --- |
| **public** **class** TestMain {  **public** **static** **void** main(String[] args) **throws** InterruptedException, ExecutionException {  ExecutorService executor = Executors.*newCachedThreadPool*();  Future<Integer> future = executor.submit(**new** AddNumberTask());  System.***out***.println(Thread.*currentThread*().getName() + "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。

# 一、线程

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。

# 线程与进程

为什么要使用多线程？

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、数据库连接池、分批发送短信等。

1.1线程分类：用户线程，守护线程，主线程，子线程，GC线程

1.2什么是线程：线程是一条执行路径，每个线程都互不影响。

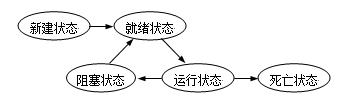
1.3什么是多线程：一个进程中，有多条不同的执行路径，并行执行。目的是提高程序效率。

1.4一个进程中一定会有一个主线程。

1.5创建线程的方式：

1. 继承Thread类
2. 实现runable接口
3. 使用匿名内部类
4. 使用线程池管理

# 2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。

## 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.

3.守护线程

会在主线程中设置一个子线程作为守护线程，当守护线程执行结束，守护线程无论任务是否结束，都跟着结束。setDaemon(true)

# java内存模型

1.1多线程特性：原子性、可见性、有序性

1.2内存模型：jmm 多线程相关。内存结构：jvm内存结构。

Jmm:决定一个线程对共享变量的写入时，决定另一个线程是否可见。

主内存：共享变量

本地内存：共享变量的副本

2.1线程安全：多个线程共享一个全局变量，做些的操作时，可能会收到其他线程的干扰。读是不会有线程安全问题。

2.1.2解决方案：

同步方式：同步代码块，或同步方法（synchronized）

同步方法: 静态同步，非静态同步。如果是new的多个对象，需要把共享参数和锁设置成静态的。

锁静态方法，相当于。锁的.class文件

3.1Threadlocal

会把一个共享的局部变量，变成本地的变量计算，线程间互不影响。

4.只要是全局共享变量，都加上volatile

5.重排序（只有在多线程的情况下才会出现，本身就是为了内部优化）

Cpu会对代码执行实现优化，不会对有依赖关系做重排序，代码执行顺序可能改变，但是执行结果不会发生变化。

Int a=1;

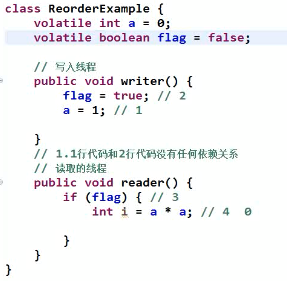
Int b=2;

Int c=a\*b;

可能a和b位置会发生变化，但是c和前面2个有依赖，就不会有变化。

下面的代码，把读和写的依赖关系隐藏了，所有要加volatile（加了它就不会改变代码顺序）

synchronized不会禁重排序，但是它是同步的，也不会有问题。volatile禁重排序。



# Java锁的深度化

## 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（重量级） 和 ReentrantLock(轻量级)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();  }  } |

## 悲观锁、乐观锁

### 乐观锁

总是认为不会产生并发问题，每次去取数据的时候总认为不会有其他线程对数据进行修改，因此不会上锁，但是在更新时会判断其他线程在这之前有没有对数据进行修改，一般会使用版本号机制或CAS操作实现。

version方式：一般是在数据表中加上一个数据版本号version字段，表示数据被修改的次数，当数据被修改时，version值会加一。当线程A要更新数据值时，在读取数据的同时也会读取version值，在提交更新时，若刚才读取到的version值为当前数据库中的version值相等时才更新，否则重试更新操作，直到更新成功。

核心SQL语句

update table set x=x+1, version=version+1 where id=#{id} and version=#{version};

CAS操作方式：即compare and swap 或者 compare and set，涉及到三个操作数，数据所在的内存值，预期值，新值。当需要更新时，判断当前内存值与之前取到的值是否相等，若相等，则用新值更新，若失败则重试，一般情况下是一个自旋操作，即不断的重试。

### 悲观锁

总是假设最坏的情况，每次取数据时都认为其他线程会修改，所以都会加锁（读锁、写锁、行锁等），当其他线程想要访问数据时，都需要阻塞挂起。可以依靠数据库实现，如行锁、读锁和写锁等，都是在操作之前加锁，在Java中，synchronized的思想也是悲观锁。

## 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

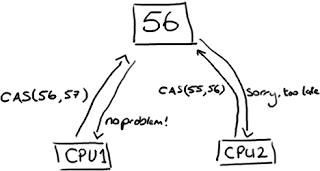
AtomicReference

|  |
| --- |
| **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。

## 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