**多线程小结**

之前简单的程序都是一个main方法作为入口，从上到下执行，这类型的程序可以称为单线程程序，类似于只有一个服务员的小店。

接下来所说的Java多线程程序，类似于有多个服务员的店，可以同时让多个用户（客户-服务员线程）访问同一个资源（厨房）。

首先要了解线程和进程的区别。

线程，指的是一个程序不同的执行顺序，类似于上面例子的客户-服务员-厨房。

进程，指的是一个正在运行中的应用程序，拥有独立的内存和地址，执行独立的功能。

然后要区别并发性和并行性。

并发指的是同一时刻，最多只有一条指令在运行，但多个进程指令被快速轮换执行。

并行指的是同一时刻，有多条指令在多个处理器上同时运行。

线程也被称为轻量级进程，线程在进程中，是独立、并发的，各自拥有自己的执行顺序。

多线程有其独特的优势

线程之间可以共享内存，共享资源

创建线程比创建进程的资源耗费小得多，使用多线程来实现多任务并发，比使用多进程效率要高

Java内置多线程功能，简单易用

**线程的创建和启动**，一般有3种方式：

1. 继承Thread类，创建线程类

A、定义Thread类的子类，重写run()方法，其方法体就是需要运行的线程内容

B、创建Thread子类的实例--线程对象

C、调用线程对象的start()方法，启动该线程

示例代码如下**package** thread;

**public** **class** **Demo1** **extends** **Thread**

{

**private** **int** i;

***@Override***

**public** **void** **run**() {

**for**(;i<5;i++)

{

**System**.***out***.println(**Thread**.*currentThread*().getName()+"\*\*\*"+i);

}

}

**public** **static** **void** **main**(**String**[] args) {

//Demo1 d1 = new Demo1();

**new** Demo1().start();

**new** Demo1().start();

**new** Demo1().start();

}

}

输出结果如下：

Thread-1\*\*\*0

Thread-0\*\*\*0

Thread-2\*\*\*0

Thread-0\*\*\*1

Thread-1\*\*\*1

Thread-0\*\*\*2

Thread-2\*\*\*1

Thread-0\*\*\*3

Thread-1\*\*\*2

Thread-0\*\*\*4

Thread-2\*\*\*2

Thread-1\*\*\*3

Thread-2\*\*\*3

Thread-1\*\*\*4

Thread-2\*\*\*4

1. 实现Runnable接口创建线程类（静态代理设计模式）
2. 定义Runnable接口的实现类，重写run()方法，
3. 创建实现类的实例，并以此实例作为Thread的target来创建Thread对象，该Thread对象才是真正的线程对象
4. 调用线程对象的start()方法，启动线程

示例代码如下：

**package** thread;

**public** **class** **Demo2** **implements** Runnable

{

**private** **int** i;

***@Override***

**public** **void** **run**() {

// **TODO** Auto-generated method stub

**for**(;i<5;i++)

{

**System**.***out***.println(**Thread**.*currentThread*().getName()+"\*\*\*"+i);

}

}

**public** **static** **void** **main**(**String**[] args) {

**Demo2** **d2** = **new** Demo2();

**new** Thread(d2,"线程1").start();

**new** Thread(d2,"线程2").start();

}

}

结果如下

线程2\*\*\*0

线程1\*\*\*0

线程2\*\*\*1

线程1\*\*\*2

线程2\*\*\*3

线程1\*\*\*4

1. **使用Callable和Future创建线程**

与继承Thread类以及实现Runnable接口创建线程相比，使用这个方法，可以有返回值，可以声明异常，总之，这个方法麻烦但是实用。

实现了Callable接口的对象其实和实现了Runnable接口的对象类似，都是作为Thread对象的target，在启动线程时，对应Runnable的run()方法,则是Callable接口的call()方法。但是，由于Callable是新增的接口，并不能直接成为Thread对象的target，需要和Future接口相配合使用，就是将实现了Callable接口的类包裹在FutureTask类中。步骤如下

1. 创建实现了Callable接口的类，重写其call()方法作为线程执行体，而且可以有返回值
2. 用FutureTask对象的实现类包裹实现了Callable接口的类，作为Thread类的target
3. 使用Thread的start()方法
4. 调用FutureTask的get()方法，获得子进程完成后的返回值

示例代码如下

**package** thread;

**import** java.util.concurrent.Callable;

**import** java.util.concurrent.ExecutionException;

**import** java.util.concurrent.FutureTask;

**public** **class** **CallableDemo2** **implements** Callable<Integer>

{

***@Override***

**public** **Integer** **call**() **throws** **Exception** {

**int** **i** = 0;

**for**(;i<5;i++)

{

**System**.***out***.println(**Thread**.*currentThread*().getName()+"\*\*\*"+i);

}

**return** i;

}

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

**CallableDemo2** **cd2** = **new** CallableDemo2();

**FutureTask**<Integer> **ft** = **new** FutureTask<Integer>(cd2);

//将实现了Callable接口的类包裹其中，作为Thread的target

**FutureTask**<Integer> **ft2** = **new** FutureTask<Integer>(cd2);

**new** Thread(ft,"xc1").start();

**new** Thread(ft2,"xc2").start();

**System**.***out***.println("\*\*\*\*\*\*\*");

//获取返回值

**System**.***out***.println(ft.get());

**System**.***out***.println(ft2.get());

}

}

结果如下

\*\*\*\*\*\*\*

xc2\*\*\*0

xc2\*\*\*1

xc2\*\*\*2

xc2\*\*\*3

xc2\*\*\*4

xc1\*\*\*0

xc1\*\*\*1

xc1\*\*\*2

xc1\*\*\*3

xc1\*\*\*4

5

5

一个线程的生命周期，可以分为New，Runnable(就绪),Running,Blockd和Dead。

Java的线程支持提供了一些方法来控制线程的执行，例如

Join进程，a.Join()，则相当于a插队了，要先运行了a才可以运行后面的语句。

Daemon Thread,后台线程，语法为DaemonThread dt = new DaemonThread()，dt.setDaemon(true),一般来说，守护线程会在所有的前台方法都实行完之后自动退出，相对的，后台线程需要在所有前台线程start()之前启动。

线程睡眠sleep，语法Thread类.sleep(毫秒),指定的线程暂停，并进入阻塞状态。

线程让步yield，语法与Thread类.yield()，指定线程让出控制权，让其他线程先执行。

改变线程优先级，例如改变当前线程的优先级,Thread.currentThread.setPriority(xx)，xx可以是数字1-10，但是一般来说，使用一些默认的静态常量MAX\_PRIORITY,值是10;MIN\_PRIORITY，值是1;NORM\_PRIORITY,值是5。

**线程同步**:当多个线程访问同一个资源的时候，就可能发生数据不正确的问题，线程安全问题。

例如甲和乙分别在两个客户端抢票，当剩下一张票的时候，很有可能都显示自己抢到票了，可是付款却发现，其中一个人没有票等等的情况。

要解决这类的问题，一个方法是对资源加锁，每次只能一个线程访问，甲线程访问完之后，乙线程才能获得访问权。使用的关键字是synchronized(obj)｛同步代码块｝，synchronized也可以锁定方法。**要十分注意的是，同步代码块的范围和逻辑要正确，否则同步锁发挥不了作用。**

另外也可以使用显示定义--使用lock对象来设置同步锁，个人觉得，使用显式方法更加易懂。例如

Public void test()

{//加锁

lock.lock();

try{

Do something;

}

finally{//使用try finall块，保证解锁

lock.unlock();

}

**线程通信：**不同线程之间交换信息，保障线程协调运行。

对于被**synchronized修饰**的同步方法，由于默认监视器的是this，类本身，所以可以在同步方法中直接调用以下3个方法：

1. wait()，当前线程等待，直至其他线程调用它的notify()或者notifyAll()方法释放等待为止。方法中可以带参数，表示等待的时长。
2. notify(),唤醒这个同步监视器this，上面等待的单个线程。如果有多个，则是随机唤醒-\_-!! 只有当前线程放弃对该同步监视器的锁定后，才能执行被唤醒的线程，也就是说，被唤醒的线程只是出于Runnable 就绪状态。
3. notifyAll()，唤醒所有wait()线程，同时让被唤醒的线程处于就绪状态。、

使用的时候，**可以设置boolean flag作为信号灯**，辅助设计。   

使用**lock对象显式**锁定的场合，可以使用Condition控制线程通信。方法其实和synchronized的类似，使用的Condition对象由lock对象的newCondition()方法生成，使用Condition.await()等方法，如下：

1. await(),类似于wait()
2. signal()，类似于notify()
3. signalAll(),类似于notifyAll()

   

使用**阻塞队列(BlockingQueue)**控制线程通信（生产者-消费者设计模式）

容器的使用，BlockingQueue有不同的子类，对应不同的数据结构，该容器只能交替地进行放入、取出元素，方法有这些：

1. put(E e) 从队尾加入元素，如满，阻塞队列
2. take()从对头拿出元素，如空，阻塞队列



**线程组和未处理的异常**：

**线程组：**Java使用TreadGroup来表示线程组，可以对一批线程进行管理。一般情况之下，创建的线程都是属于默认线程组。在执行期间，线程所属不能更改，所以线程组对象没有set方法，但是有get方法返回所属的线程组。Thread类有一些构造器，可以将指定的线程构造为指定线程组所属。ThreadGroup类提供ThreadGroup(String name)和ThreadGroup(ThreadGroup parent,String name)方法来创建新的线程组。ThreadGroup类提供以下方法来操作**整个线程组里面的所有线程：**

1. int activeCount(),返回此线程组中活动线程的数目
2. Interrupt(),中断指定线程组的所有线程
3. isDaemon(),判断是否为后台线程
4. setDaemon(boolean daemon),将线程组设置为后台线程组
5. setMaxPriority(int pri),设置线程组的优先级



**未处理异常：**Java5开始，JVM在结束线程之前，会检查是否有对应的Thread.UncaughtExceptionHandler对象，如果有，则抛出该对象的uncaughtException(Thread t,Throwable e)异常。

Thread.UncaughtExceptionHandler是Thread类的一个静态接口，只有一个对应的抛出异常方法，提供了一些方法来设置异常处理器。

ThreadGroup类实现了Thread.UncaughtExceptionHandler接口，所以每个线程所属的线程组将会作为默认的异常处理器。一般的原则是

1. 如果线程组有父线程组，则使用父线程组的uncaughtException
2. 如果线程实例所属的线程类有默认的异常处理器，则调用之
3. 如果该异常对象是ThreadDeath对象（线程进入死亡阶段），则不作处理



**线程池：**系统启动一个线程，非常耗时，尤其是一些线程只是生存一小段时间的时候，使用线程池，可以比较好地提高系统性能。

与数据库的连接池类似，线程池在系统启动的时候创建大量空闲的线程，程序有需要启动线程的时候，则使用这些资源，在执行完run()方法或者start()方法后，线程并没有死亡，而是再次回到线程池，等待下一个Runnable对象或者Callable对象的run()方法或者start()方法。

Java5开始，系统内建线程池。