**JAVA多线程基础**

Java支持多线程，线程对于进程来说，创建的代价比较小，上下文切换比较迅速，线程是进程的执行单元，是进程的组成部分，一个进程可以拥有多个线程，一个线程必须要拥有一个父进程。一个线程可以创建和销毁另一个线程，同一个进程中的多个线程可以并发执行。

执行一个程序，操作系统会分配一个进程，在这个进程中至少会新建一个线程来作为程序执行的入口点。

进程与进程之间不共享内存，而进程中的线程可以共享系统分配给这个进程的内存空间，线程不仅可以共享进程的内存空间，还有一个属于自己的内存空间，这段内存空间叫做线程栈。

# 1、线程的创建与启动

## 1.1、继承Thread类

继承Thread类就可以创建一个线程类，run方法就是线程执行体，调用线程对象的start方法就可以启动线程。

示例代码：

public class ThreadTest1 extends Thread{

public void run(){

System.out.println(getName() + " 线程启动 ");

}

public static void main(String[] args){

ThreadTest1 tt1 = new ThreadTest1();

tt1.start();

}

}

通过继承Thread类的方式创建的线程实例之间无法共享变量。

## 1.2、实现Runnable接口

定义Runnable接口的实现类，重写run方法，run方法就是线程的执行体。Thread类中定义了属性名为target的Runnable类型的变量，我们可以创建Runnable接口实现类的实例，并以此实例作为Thread对象的target来创建线程对象，并调用线程对象的start方法启动线程。

示例代码：

public class ThreadTest2 {

public static void main(String[] args){

ThreadTest2 tt2 = new ThreadTest2();

new Thread(tt2.new RunnableThread(),"Runnable实现线程").start();

}

class RunnableThread implements Runnable{

public void run(){

System.out.println(Thread.currentThread().getName() + " 线程开始启动！");

}

}

}

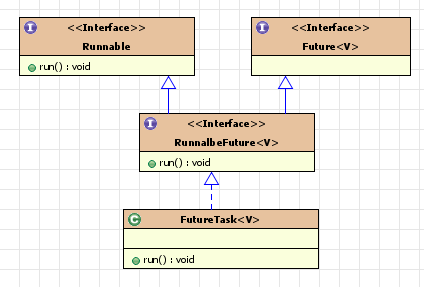
用实现Runnable接口的方式创建的线程类可以共享线程类的实例属性，因为在这种方式下创建的Runnable对象只是线程类的target属性，而多个线程可以共享一个target对象。

## 1.3、使用Callable接口和FutureTask类创建

之前两种线程的创建方式都无法实现线程完成后的返回值，java 5之后提供了Callable可以实现当线程结束后有返回值和抛出异常。创建Callable接口的实现类，重写call方法，call方法就是线程的实现类，Callable接口并不是Runnable接口的子接口，所以无法作为Thread类的target。

Java5之后提供了FutureTask类，该类实现了RunnableFuture接口，RunnableFuture接口继承了Runnable和Future接口，所以FutureTask类可以作为Thread的target。

类图如下：



Callable接口有泛型形参限制，该类型和call方法返回值类型相同。

public class ThreadTest3 {

class CallableThread implements Callable<Integer>{

public Integer call(){

int k =0;

for(int i=0; i<100; i++){

System.out.println(Thread.currentThread().getName() + " 线程开启 i = " + i);

k++;

}

return k;

}

}

public static void main(String[] args){

ThreadTest3 tt3 = new ThreadTest3();

FutureTask<Integer> task = new FutureTask<Integer>(tt3.new CallableThread());

new Thread(task,"callable线程").start();

try {

System.out.println("子线程的返回值：" + task.get().toString());

} catch (Exception e) {

e.printStackTrace();

}

}

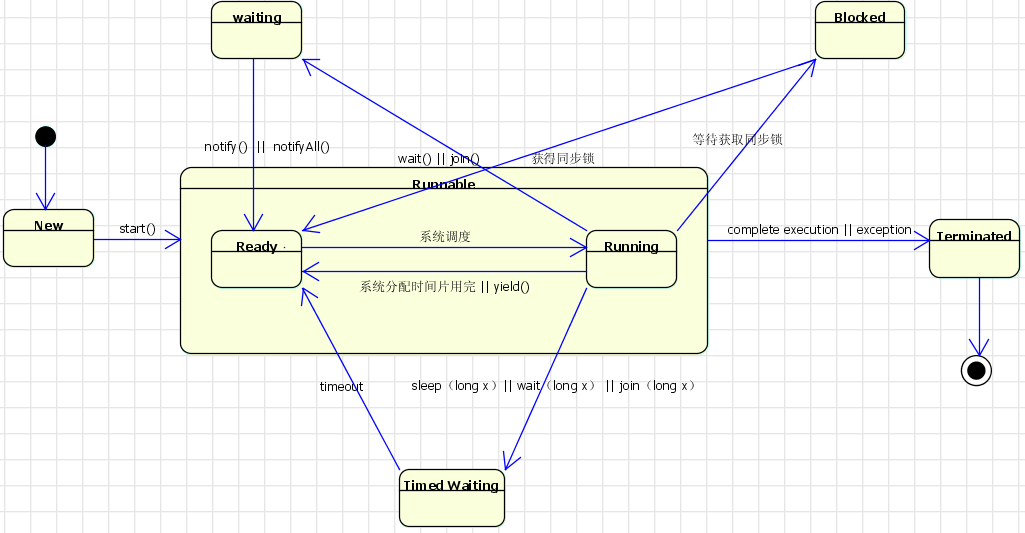
}

调用FutureTask类的get方法会导致线程阻塞，直到call方法有返回值。

# 2、线程的生命周期

线程有5种状态，分别是新建（New）、就绪（Runnable）、运行（Running）、阻塞（Blocking）、死亡（Dead）。

状态图如下：



当主线程结束后，其他线程不受任何影响，并不会随之结束，一旦子线程启动后就拥有了和主线程相同的地位，不会受主线程的影响。

# 3、控制线程

## 3.1、join方法

让一个线程等待另一个线程执行完的方法，当在程序执行流中调用其他线程的join方法，那么当前线程就会被阻塞，直到被调用join方法的线程执行完成。

join方法通常由使用线程的程序调用，一般将大问题划分成一些小问题，每个小问题分配一个线程，当所有的小问题 、执行好之后，在继续主线程的操作。

## 3.2、sleep方法

让当前正在执行的线程暂停一段时间。

## 3.3、yield方法

跟sleep方法有点类似，都是暂停当前线程，只是调用yield方法的线程不会进入阻塞状态，而是进入就绪状态。

当某个线程调用yield方法暂停后，只有优先级与当前线程相同，或者优先级高于当前线程并且处于就绪状态的线程才会被执行。

# 4、线程同步

## 4.1、同步代码块

使用synchronized关键字，示例如下：

synchronized(obj){ //obj是同步监视器

……

}

## 4.2、同步方法

使用synchronized修饰方法，这样具有隐式的同步监视器，就是对象本身。synchronized关键字只能修饰方法和代码块。

## 4.3、同步锁

Java5开始可以通过显示的定义同步锁对象来实现同步。锁提供了对共享资源的独占访问，每次只有一个线程对Lock对象加锁。

比较常用的是ReentrantLock（可重入锁）是lock接口的实现类，ReentrantLock是可重入锁，当被加锁的情况下也还可以被加锁，

示例代码：

public class ThreadTest4 {

private final ReentrantLock lock = new ReentrantLock();

public void test(){

lock.lock();

try{

//do sth

}finally{

lock.unlock();

}

}

}

# 5、线程通信

## 5.1、调用同步监视器方法

对于通过synchronized修饰的同步代码块或者同步方法，可以使用wait、notify、notifyAll方法来实现线程之间的通信。

wait： 导致当前线程等待，直到其他线程调用同步监视器的notify或notifyAll方法。

notify: 唤醒在该同步监视器上等待的单个线程。

notifyAll: 唤醒在该同步监视器上等待的所有线程。

## 5.2、使用Condition控制线程通信

当使用lock来实现同步机制的时候，并不存在隐式的同步监视器，所以需要使用Condition类来实现线程之间的通信。

通过lock.newCondition()来获取lock的同步监视器，condition类中有如下三个方法进行线程之间的通信。

await：相当于wait

signal: 相当于notify

signalAll：相当于notifyAll

## 5.3、使用阻塞队列（BlockingQueue）

BlockingQueue接口提供了两个线程阻塞的方法：

put: 如果队列已满就阻塞当前线程。

take：从队列头部取出元素，如果队列为空，则阻塞当前线程。

ArrayBlockingQueue和ListBlockingQueue分别是BlockingQueue的两个比较常用的实现类。

# 6、线程池

启动一个新线程成本比较高，因为涉及到操作系统的交互，所以使用线程池可以很好的提高性能。线程池在系统启动的时候创建大量的空闲线程，程序将Runnable或Callable对象传给线程池，线程池就分配线程执行线程执行体，如果执行体执行完成后线程并不死亡而是处于空闲状态，直到线程池分配新的线程执行体。

可以通过Executors工厂类来产生线程池，该类中有以下几个常用的静态方法：

newCachedThreadPool: 创建具有缓存功能的线程池

newFixedThreadPool(int poolsize): 创建一个固定个数，可重用的线程池

newSingleThreadExecutor：创建一个只有单线程的线程池

ExecutorService代表尽快执行的线程池，只需将Runnable或Callable对象提交给线程池就可执行任务。有如下几个方法可以将任务提交给线程池：

Future<?> submit(Runnable task): 因为run方法无返回值，所以执行好后Future为null。

Future<T> submit(Runnable task, T result): result为显示的指定返回值。

Future<T> submit(Callable<T> task): Future就是call方法的返回值的封装类。

代码示例：

public class ThreadTest5 {

class RunnableThread implements Runnable{

public void run(){

System.out.println(Thread.currentThread().getName() + " 开始执行");

}

}

public static void main(String[] args){

ThreadTest5 tt5 = new ThreadTest5();

ExecutorService pool = Executors.newFixedThreadPool(1);

Future future = pool.submit(tt5.new RunnableThread(),"test");

try {

System.out.println("future = " + future.get());

} catch (Exception e) {

e.printStackTrace();

}

pool.shutdown();

}

}

输出：

pool-1-thread-1 开始执行

future = test

示例代码：

public class ThreadTest6 {

class CallableThread implements Callable<String>{

public String call(){

System.out.println(Thread.currentThread().getName() + " 开始启动");

return "run call method";

}

}

public static void main(String[] args){

ThreadTest6 tt6 = new ThreadTest6();

ExecutorService pool = Executors.newFixedThreadPool(1);

Future future = pool.submit(tt6.new CallableThread());

try {

System.out.println("Callable result = " + future.get().toString());

} catch (Exception e) {

e.printStackTrace();

}

pool.shutdown();

}

}

代码输出：

pool-1-thread-1 开始启动

Callable result = run call method

# 7、线程安全的集合类

（1）使用Collections类下以synchronized开头的静态方法包装不安全的线程集合

（2）concurrent包下以Concurrent开头的集合

（3）以CopyOnWrite开头的集合