1.使用Callable接口实现多线程求和的返回值判断线程的各自的执行加值的顺序

源码：

**package** xhu.edu.callable;

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

//使用通过实现Callable接口，每个线程执行一次sum++,最后得到值判断多个线程的执行顺序

**public** **class** Sumnum **implements** Callable<Integer> {

**private** **int** sum = 0;

@Override

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

**synchronized** (**this**) {

sum += 10;

}

**return** sum;

}

}

**package** xhu.edu.callable;

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

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

**public** **class** Test {

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

//新建一个Sumnum对象

Sumnum sumnum=**new** Sumnum();

//使用该对象创建三个FutureTask对象，因为使用的是一个Sumnum对象创建FutureTask，所以相当于使用的同一个

//实现Runnable接口的对象创建三个独立的线程，所以Sumnum对象是共享的，它里面的成员变量也是共享的

FutureTask<Integer> task1=**new** FutureTask<Integer>(sumnum);

FutureTask<Integer> task2=**new** FutureTask<Integer>(sumnum);

FutureTask<Integer> task3=**new** FutureTask<Integer>(sumnum);

//创建三个线程

**new** Thread(task1).start();

**new** Thread(task2).start();

**new** Thread(task3).start();

//通过FutureTask对象获取返回值,通过各自的值即可判断这些线程谁先谁后执行

System.***out***.println("task1:"+task1.get());

System.***out***.println("task2:"+task2.get());

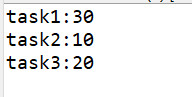
System.***out***.println("task3:"+task3.get());

}

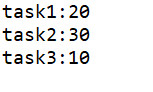
}

运行截图：

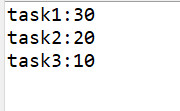
第1次执行：



第2次执行：



第3次执行：



总结：

实现Callable接口，可以带一个返回值，而这个值是需要通过实现Future接口的类的对象进行调用，一般使用FutureTask，因为要创建Thread的构造函数中没有Callable做参数的，所以需要进行将Callable转化为Runnable，而FutureTask恰好是其子类，且其构造函数又是Callable作为参数的方法，所以通过实现Callable接口的类的对象去构造FutureTask对象，再通过FutureTask对象构造线程对象。（注意：按理说是可以通过一个FutureTask去构造多个线程，但是因为一般我们要不同的线程返回各自的值，但是FutureTask只能get()到一个值所以，选择创建多个FutureTask对象。而且get()方法是阻塞方法，如果对应的线程没有执行完成，那么调用get()方法所在的线程就会被阻塞，所以需要将其放在较后的位置。）

2.线程池的使用：

源码：

**package** xhu.edu.threadpool;

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

**public** **class** DemoC **implements** Callable<String> {

@Override

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

**return** "this is Callable:" + Thread.*currentThread*().getName();

}

}

**package** xhu.edu.threadpool;

**public** **class** DemoR **implements** Runnable {

@Override

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

System.***out***.println("this is Runnable:"+Thread.*currentThread*().getName());

}

}

**package** xhu.edu.threadpool;

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

**import** java.util.concurrent.ExecutorService;

**import** java.util.concurrent.Executors;

**import** java.util.concurrent.Future;

**public** **class** Test {

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

//通过Executors创建一个线程池

ExecutorService executorService=Executors.*newFixedThreadPool*(2);

//执行一个实现Callable接口的线程

Future future=executorService.submit(**new** DemoC());

//执行一个实现Runnable接口的线程

executorService.submit(**new** DemoR());

System.***out***.println((String)future.get());

//再执行一个实现Callable接口的线程

Future future2=executorService.submit(**new** DemoC());

//再执行一个实现Runnable接口的线程

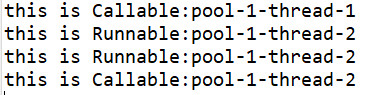
executorService.submit(**new** DemoR());

System.***out***.println((String)future2.get());

}

}

运行截图：



总结：

由运行结果可以知道：

每次调用线程池执行的线程是随机的，也证明了线程池中的线程无论是实现Runnable和Callable接口的类的对象都可以进行执行。还说明了线程池中的线程确实没有在执行完后就死亡，而是进入了休眠状态。