链滴

线程创建的四种方式详解

作者：lyz1996
原文链接：https://ld246.com/article/1599488529392
来源网站：链滴

线程创建的四种方式详解

概述：

Java使用Thread类代表线程，所有的线程对象都必须是Thread类或其子类的实例。Java可以用四种式来创建线程，如下所示：

Java使用Thread类代表线程，所有的线程对象都必须是Thread类或其子类的实例。Java可以用四种式来创建线程，如下所示：
Java使用Thread类代表线程，所有的线程对象都必须是Thread类或其子类的实例。Java可以用四种式来创建线程，如下所示：

- 继承Thread类创建线程
- 实现Runnable接口创建线程
- 使用Callable和Future创建线程
- 使用线程池例如用Executor框架

详解

继承Thread类创建线程

主要步骤为：

- 定义Thread类的子类，并重写该类的run()方法，该方法的方法体就是线程需要完成的业务代码任，run()方法也称为线程的执行体
- 创建Thread子类的实例，也就是创建了线程对象
- 启动线程，及调用线程的start()方法

代码实例：

```
public class MyThread extends Thread {
    public void run() {
        // ...
    }
}
public static void main(String[] args) {
    MyThread mt = new MyThread();
    mt.start();
}
```

实现Runnable接口创建线程

主要步骤为：

- 定义Runnable接口的实现类，一样要重写run()方法，这个run()方法和Thread()中的run()方法一样线程的执行体
- 创建Runnable实现类的实例额，并用这个实例作为Thread的target来创建Thread对象，这个Threa

对象才是真正的线程对象

- 通过调用线程对象的start()房发来启动线程

代码实现：

```
public class MyRunnable implements Runnable {
    @Override
    public void run() {
        // ...
    }
}

public static void main(String[] args) {
    MyRunnable instance = new MyRunnable();
    Thread thread = new Thread(instance);
    thread.start();
}
```

使用Callable和Future创建线程

注意事项：

和Runnable接口不一样，Callable接口提供了一个call（）方法作为线程执行体，call()方法比run()法功能要强大。

和Runnable接口不一样，Callable接口提供了一个call（）方法作为线程执行体，call()方法比run()法功能要强大。
和Runnable接口不一样，Callable接口提供了一个call（）方法作为线程执行体，call()方法比run()法功能要强大。

- call()方法可以有返回值
- call()方法可以声明抛出异常

Java5提供了**Future**接口来代表**Callable**接口里**call()**方法的返回值，并且为Future接口提供了一实现类FutureTask，这个实现类既实现了Future接口，还实现了Runnable接口，因此可以作为Threa类的target。在Future接口里定义了几个公共方法来控制它关联的Callable任务。

- boolean cancel(boolean mayInterruptIfRunning)：视图取消该Future里面关联的Callable任务
- V get()：返回Callable里call()方法的返回值，调用这个方法会导致程序**阻塞**，必须等到子线程结束才会得到返回值
- V get(long timeout,TimeUnit unit)：返回Callable里call()方法的返回值，最多阻塞timeout时间经过指定时间没有返回抛出TimeoutException
- boolean isDone()：若Callable任务完成，返回True
- boolean isCancelled()：如果在Callable任务正常完成前被取消，返回True

主要步骤：

- 创建Callable接口的实现类，并实现call()方法，然后创建该实现类的实例（从java8开始可以直接用Lambda表达式创建Callable对象）。
- 使用FutureTask类来包装Callable对象，该FutureTask对象封装了Callable对象的call()方法的返回值

- 使用FutureTask对象作为Thread对象的target创建并启动线程（因为FutureTask实现了Runnable口）
- 调用FutureTask对象的get()方法来获得子线程执行结束后的返回值

代码实例：

```
public class MyCallable implements Callable<Integer> {
    public Integer call() {
        return 123;
    }
}
public static void main(String[] args) throws ExecutionException, InterruptedException {
    MyCallable mc = new MyCallable();
    FutureTask<Integer> ft = new FutureTask<>(mc);
    Thread thread = new Thread(ft);
    thread.start();
    // ft.get() 可以获取返回值
    System.out.println(ft.get());
}

// Lambda 表达式方法
public class Test {
      public static void main(String[] args){
         MyThread3 th=new MyThread3();
         //使用Lambda表达式创建Callable对象
        //使用FutureTask类来包装Callable对象
         FutureTask<Integer> future=new FutureTask<Integer>(
            (Callable<Integer>)()->{
                  return 5;
            }
        );
    //实质上还是以Callable对象来创建并启动线程
         new Thread(task,"有返回值的线程").start();
        try{
       //get()方法会阻塞，直到子线程执行结束才返回
            System.out.println("子线程的返回值："+future.get());
        }catch(Exception e){
            ex.printStackTrace();
         }
      }
}
```

使用线程池例如用Executor框架

1.5后引入的Executor框架的最大优点是把任务的提交和执行解耦。要执行任务的人只需把Task描述楚，然后提交即可。这个Task是怎么被执行的，被谁执行的，什么时候执行的，提交的人就不用关心。具体点讲，提交一个Callable对象给ExecutorService（如最常用的线程池ThreadPoolExecutor）将得到一个Future对象，调用Future对象的get方法等待执行结果就好了。Executor框架的内部使用线程池机制，它在java.util.cocurrent 包下，通过该框架来控制线程的启动、执行和关闭，可以简化发编程的操作。因此，在Java 5之后，通过Executor来启动线程比使用Thread的start方法更好，除更易管理，效率更好（用线程池实现，节约开销）外，还有关键的一点：有助于避免this逃逸问题—如果我们在构造器中启动一个线程，因为另一个任务可能会在构造器结束之前开始执行，此时可能会问到初始化了一半的对象用Executor在构造器中。

Executor框架包括：线程池，Executor，Executors，ExecutorService，CompletionService，Futu e，Callable等。

Executor接口中之定义了一个方法execute（Runnable command），该方法接收一个Runable实 ，它用来执行一个任务，任务即一个实现了Runnable接口的类。ExecutorService接口继承自Executo 接口，它提供了更丰富的实现多线程的方法，比如，ExecutorService提供了关闭自己的方法，以及 为跟踪一个或多个异步任务执行状况而生成 Future 的方法。 可以调用ExecutorService的shutdown ）方法来平滑地关闭 ExecutorService，调用该方法后，将导致ExecutorService停止接受任何新的 务且等待已经提交的任务执行完成(已经提交的任务会分两类：一类是已经在执行的，另一类是还没有 始执行的)，当所有已经提交的任务执行完毕后将会关闭ExecutorService。因此我们一般用该接口来 现和管理多线程。

ExecutorService的生命周期包括三种状态：运行、关闭、终止。创建后便进入运行状态，当调用了sh tdown（）方法时，便进入关闭状态，此时意味着ExecutorService不再接受新的任务，但它还在执 已经提交了的任务，当素有已经提交了的任务执行完后，便到达终止状态。如果不调用shutdown（ 方法，ExecutorService会一直处在运行状态，不断接收新的任务，执行新的任务，服务器端一般不 要关闭它，保持一直运行即可。

Executors提供了一系列工厂方法用于创先线程池，返回的线程池都实现了ExecutorService接口。

具体对比图片如下：

Executor执行Callable任务

在Java 5之后，任务分两类：一类是实现了Runnable接口的类，一类是实现了Callable接口的类。 者都可以被ExecutorService执行，但是Runnable任务没有返回值，而Callable任务有返回值。并且Ca lable的call()方法只能通过ExecutorService的submit(Callable<T> task) 方法来执行，并且返回一个 T>Future<T>，是表示任务等待完成的 Future。

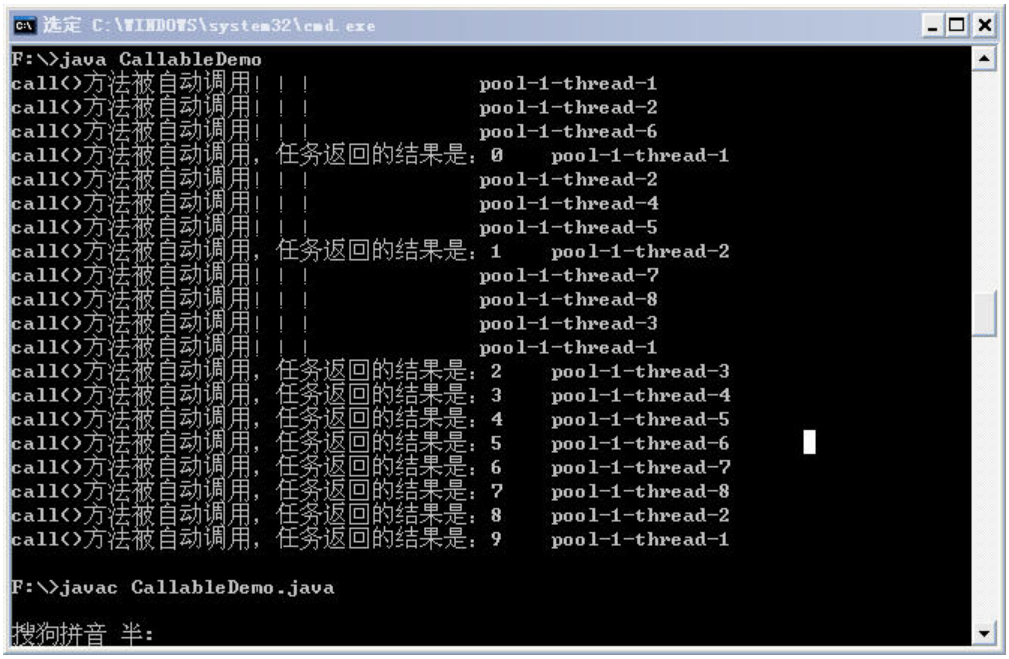
Callable接口类似于Runnable，两者都是为那些其实例可能被另一个线程执行的类设计的。但是 Run able 不会返回结果，并且无法抛出经过检查的异常而Callable又返回结果，而且当获取返回结果时可 会抛出异常。Callable中的call()方法类似Runnable的run()方法，区别同样是有返回值，后者没有。

当将一个Callable的对象传递给ExecutorService的submit方法，则该call方法自动在一个线程上执行 并且会返回执行结果Future对象。同样，将Runnable的对象传递给ExecutorService的submit方法 则该run方法自动在一个线程上执行，并且会返回执行结果Future对象，但是在该Future对象上调用ge 方法，将返回null。

```
import java.util.ArrayList;
import java.util.List;
import java.util.concurrent.*;

public class CallableDemo{
    public static void main(String[] args){
        ExecutorService executorService = Executors.newCachedThreadPool();
        List<Future<String>> resultList = new ArrayList<Future<String>>();

        //创建10个任务并执行
        for (int i = 0; i < 10; i++){
            //使用ExecutorService执行Callable类型的任务，并将结果保存在future变量中
            Future<String> future = executorService.submit(new TaskWithResult(i));
            //将任务执行结果存储到List中
            resultList.add(future);
```

```
        }

        //遍历任务的结果
        for (Future<String> fs : resultList){
            try{
                while(!fs.isDone);//Future返回如果没有完成，则一直循环等待，直到Future返回完成
                System.out.println(fs.get());     //打印各个线程（任务）执行的结果
            }catch(InterruptedException e){
                e.printStackTrace();
            }catch(ExecutionException e){
                e.printStackTrace();
            }finally{
                //启动一次顺序关闭，执行以前提交的任务，但不接受新任务
                executorService.shutdown();
            }
        }
    }
}


class TaskWithResult implements Callable<String>{
    private int id;

    public TaskWithResult(int id){
        this.id = id;
    }

    /**
     * 任务的具体过程，一旦任务传给ExecutorService的submit方法，
     * 则该方法自动在一个线程上执行
     */
    public String call() throws Exception {
        System.out.println("call()方法被自动调用！！！    " + Thread.currentThread().getName());
        //该返回结果将被Future的get方法得到
        return "call()方法被自动调用，任务返回的结果是：" + id + "    " + Thread.currentThread().g
tName();
    }
}
```

执行结果：



从结果中可以同样可以看出，submit也是首先选择空闲线程来执行任务，如果没有，才会创建新的线来执行任务。另外，需要注意：如果Future的返回尚未完成，则get（）方法会阻塞等待，直到Futur完成返回，可以通过调用isDone（）方法判断Future是否完成了返回。

总结

四种创建线程方法对比

实现Runnable和实现Callable接口的方式基本相同，不过是后者执行call()方法有返回值，后者线程行体run()方法无返回值，因此可以把这两种方式归为一种这种方式与继承Thread类的方法之间的差如下：

1. 线程只是实现Runnable或实现Callable接口，还可以继承其他类。
2. 这种方式下，多个线程可以共享一个target对象，非常适合多线程处理同一份资源的情形。
3. 但是编程稍微复杂，如果需要访问当前线程，必须调用Thread.currentThread()方法。
4. 继承Thread类的线程类不能再继承其他父类（Java单继承决定）。
5. 前三种的线程如果创建关闭频繁会消耗系统资源影响性能，而使用线程池可以不用线程的时候放回程池，用的时候再从线程池取，项目开发中主要使用线程池