链滴

java 多线程学习

作者：Zl992532172

原文链接：https://ld246.com/article/1503388911457

来源网站：链滴

一、如何实现多线程

1. 实现Runnable接口

```
 public static void main(String[] args) {
    MyThread myThread = new MyThread();// 一个实现了Runnable接口的类
    Thread t = new Thread(myThread);// 声明一个线程
    t.start();// 启动线程
}

public class MyThread implements Runnable {
    @Override
    public void run() {// 启动线程后执行的方法
        System.out.print("run the \"run()\" method!");
    }
}
```

2. 继承Thread类

```
public static void main(String[] args) {
    MyThread myThread = new MyThread();// 一个继承了Thread类的类
    myThread.start();// 启动线程
}

public class extends Thread {
    @Override
    public void run() {// 启动线程后执行的方法
        System.out.print("run the \"run()\" method!");
    }
}
```

3. 使用ExecutorService、Future和Callable创建有返回值的线程

```
public static void main(String[] args) {
    // 创建一个ExecutorService
    ExecutorService executorService =
            Executors.newCachedThreadPool();
    // new一个MyThread线程,并交给executorService执行,
    // 通过Future接收返回结果
    Future future =
            executorService.submit(new MyThread());
    try {
        // 从future中获取返回值
        Object result = future.get();
        System.out.println(result.toString());
    } catch (Exception e) {
        e.printStackTrace();
    }
}

// 一个有返回值的线程
public class MyThread implements Callable {
    @Override
    public String call() {
```

```
        System.out.print("run the \"call()\" method!");
        return "test";
    }
}
```

二、实质

- Runnable、Callable接口和多线程的实现没有关系

接口的作用是约束行为,Runnable接口的作用是指定一个协议,规定所有继承这些接口的类都要有一个参无返回值的方法

- 多线程的实现是由类来完成的

java中所有的多线程最终都通过Thread类来实现,线程的资源分配、线程启动这些工作都是在start()方中完成的,严格来讲,在start0()方法中完成的。start0()是一个native方法,并不是用java语言实现的。

1. 可以通过两种方式来定义多线程中要完成的工作

- 实现Runnable接口

Runnable接口中的run()方法,没有返回值

- 实现Callable接口

Callable接口中的call()方法,有返回值

2. 需要对线程进行管理(提交、启动、终止)

    1. Thread类

        - Thread类有一个构造函数可以传递一个Runnable类型的参数target,可以通过target来提交要执行的任务(实现run()方法,然后传给Thread实例)
        - Thread类本身实现了Runnable接口,也可以直接通过实现Thread类来提交想要执行的任务(写run()方法)
        - Thread类的start()方法负责启动执行线程,当start()方法执行时
            - 若已经重写了run()方法来执行任务,则会执行该方法
            - 若传入了target参数,则会调用target中的run()方法
        - Thread类中有stop方法负责停止线程,但是已经弃用
        - 可以通过interrupt()方法中断线程

    2. ExecutorService接口

        - 可以通过executor()方法或submit()方法提交任务
        - submit()方法提交的任务会返回一个Future对象,可以通过这个对象来获取返回结果
        - shutdown()和shutdownNow()方法可以用来停止线程池

三、详解

1. 线程的生命周期

- new thread(新建)：创建一个线程实例,比如通过new操作创建一个Thread类的实例,此时线程未

启动

- runnable(可运行)：一个线程创建好之后,需要通知cpu这个线程可以开始执行了,比如thread类的tart()方法执行后,此时线程在就绪队列中等待cpu分配资源
- running(运行中)：线程获得cpu资源后开始运行,比如运行run()方法中的逻辑,此时除非线程自动弃cpu资源或者有优先级更高的线程进入,否则将执行到线程结束
- dead(死亡)：线程正常执行结束,或者被kill调,此时线程将不会再次被执行
- block(阻塞)：线程主动让出cpu使用权、其它更高优先级的线程进入、该线程的时间片用完,此时该线程还没有执行完成,都会使线程进入block状态,进入block状态的线程还可以回到就绪队列中待再次执行。

2. Thread类中的方法

- start：启动一个线程,这个方法会是线程进入Runnable状态,等待执行
- isAlive：判断线程是否处于活动状态（Runnable或running）
- sleep：强制让线程放弃当前时间片进入休眠状态一定时间,此时线程会进入block状态,直到休眠时间结束,再进入Runnable状态。sleep是静态方法,只能控制所在线程。sleep(0)会直接触发下一次cp竞争,如果没有优先级更高的线程,则会继续工作
- wait（override Object）：放弃对象锁,进入等待池,只有针对此对象调用notify()方法之后,才会次进入Runnable状态
- join：阻塞等待线程结束,可以接收参数millis和nanos指定等待的最大时间
- interrupt:中断线程,这个方法并不能中断正在运行的线程,运行该方法后,只有当线程被join(),sleep)和wait()方法所阻塞时,才会被interrupted方法所中断,并抛出一个InterruptedException异常
- static yield：主动放弃cpu使用权,回到Runnable状态

四、Tips

1. block状态

1. 等待阻塞：运行线程执行了wait方法,该线程会释放占用的所有资源包括对象锁,进入等待队列中进入等在队列的线程是不能自动唤醒的,必须依靠其它线程调用notify()、notifyAll()来进行唤醒（该状下线程会释放对象锁）

2. 同步阻塞：运行的线程在获取对象同步锁时,同步锁已被其它线程占用,则该线程会进入锁队列等获取同步锁,直到获取到同步锁之后回再次进入Runnable状态（该状态下线程还没有获得对象锁）

3. 其它阻塞：运行的线程调用了sleep()或join()方法,或者发出IO请求,该线程会进入阻塞状态,知道slep超时、join所等待的线程结束或是IO操作完成,则会再次进入Runnable状态（该状态下线程只会放弃pu而不会释放对象锁）

2. sleep(0)

sleep(0)会重新触发一次cpu竞争,当Runable队列中有大于或等于当前线程优先级的线程时,当前线程进入Runnable队列将cpu的使用权让出,否则会继续运行

3. sleep()和wait()

- sleep方法会让出cpu,但不会释放对象锁,等到sleep超时之后会自动进入Runnable队列
- wait方法会让出cpu,并释放对象锁,需要其它线程调用notify()、notifyAll()才能重新进入Runnabl队列

4. interrupt()

interrupt方法的作用更倾向于告诉线程,你可以结束了,而不是直接地中断线程,知道线程进入阻塞状态时才能中断线程。对于陷入死循环、IO等待等难以进入阻塞状态的线程来说,interrupt方法是不能有效断的。

5. sleep()和yield()

这两个方法都会让出cpu使用权,sleep会进入block状态,而yield会直接进入Runnable状态