链滴

并发编程

作者：468336329Zc
原文链接：https://ld246.com/article/1608992331470
来源网站：链滴

1.线程基础

用户线程和主线程main线程的区别:

- 主线程停掉之后，用户线程不会跟着停掉。
- 守护线程会随着主线程停掉之后 停掉。

不需要结果返回就叫异步执行，需要结果的返回叫同步（需要步调一致）。

==多核cpu可以并行执行多个线程，提升效率。但线程太多，会影响cpu资源调度，会影响效率。==

==单核cpu是串行执行，会比较浪费时间。单核cpu上运行多个程序，实际上是并不是并行执行，而cpu进行快速的切换，让我们觉得是并行运行的。==

1.创建线程

- 继承Thread方式
- 实现Runable方式
- FutureTask方式

1)一般使用runabble方式创建任务对象传参给Thread构造线程。

不使用继承的方式创建类。

```
//1.使用RUnnable创建任务对象runableThread  配合Thread创建线程
    runableThread runableThread = new runableThread ();
    Thread t1 = new Thread (runableThread);
    t1.start ();//开始执行。。
```

2)可得返回结果的Thread创建方式

实现callable，将对象传给FutureTask实现 线程创建。

```
//2.使用FutureTask配合calable 创建任务对象，传参给Thread构造线程  这种方式能获取返回结果
    callable callable = new callable ();
    FutureTask<String> task = new FutureTask<String> (callable);
    Thread t2 = new Thread (task);//task对象传参给Thread构造线程
    t2.start ();//开始执行
    System.out.println (task.get ());

//callable类
 static class callable implements Callable{

    @Override
    public String call() throws Exception {
        try{
            Thread.sleep (3000);
        }catch(Exception e){
```

```
        }
        return "正在运行，";
    }

}
```

2. join方法（同步等待）

join方法就是暂停 同步的意思，

t1.join等待t1完成之后，才往后面运行。

t1.join(2000); 2s后就会停止等待，不过当t1提前结束后，join就会立刻停止

3.interrupt方法详解

打断线程，并设置打断标记。

打断sleep，wait，join等状态的线程，会抛出InterruptException异常，打断标记会被清除，isInterrupted判断是不是被打断，根据这个可自己控制什么时候退出线程。

打断正常状态的线程，标记为true

```
boolean interrupted = Thread.currentThread ().isInterrupted ();
```

4.线程安全

- 多线程对共享资源操作，线程出现==交错==情况下，就会出现问题。
- 一块代码存在对共享资源的 **多线程读写**操作，就是临界区。

（共享资源 在几个线程都操作）

解决办法：

- 阻塞时方式：synchronzied 加 锁
- 非阻塞时方式：原子操作

对临界区的共享资源synchronzied枷锁，使之互斥。

1.synchronzied加锁实质是保护了synchronzied内的代码块的原子性，这些操作不会被多个线程分。

2.synchronzied加在方法上，等同于所在当前对象上。锁只能锁对象。

```
public  void sell(int count){
    synchronzied(this){
        this.amount-=count;
    }
```

```
}
```

3. 如果出现两个不同的临界区，需要给两个分别加上锁.

```
public  void sell(int count,Student s){
        synchronzied(obj1){
            this.amount-=count;
        }
        synchronzied(obj2)
        {
         s.count++;
        }


}
```

但可找出两个临界区都共享的对象，进行枷锁。

```
public  void sell(int count,Student s){
        synchronzied(People){
         this.amount-=count;
         s.count++;
        }
}
```