day18【线程池、死锁、线程状态、等待与唤醒】

今日内容

- 死锁
- 线程池
- 线程状态
- 等待与唤醒

教学目标

- [ ] 能够描述Java中线程池运行原理
- [ ] 能够描述死锁产生的原因
- [ ] 能够说出线程6个状态的名称
- [ ] 能够理解等待唤醒案例

第一章 线程池方式

1.1 线程池的思想

我们使用线程的时候就去创建一个线程，这样实现起来非常简便，但是就会有一个问题：

如果并发的线程数量很多，并且每个线程都是执行一个时间很短的任务就结束了，这样频繁创建线程就会大大降低系统的效率，因为频繁创建线程和销毁线程需要时间。

那么有没有一种办法使得线程可以复用，就是执行完一个任务，并不被销毁，而是可以继续执行其他的任务？

在Java中可以通过线程池来达到这样的效果。今天我们就来详细讲解一下Java的线程池。

1.2 线程池概念

- **线程池**：其实就是一个容纳多个线程的容器，其中的线程可以反复使用，省去了频繁创建线程对象的操作，无需反复创建线程而消耗过多资源。

由于线程池中有很多操作都是与优化资源相关的，我们在这里就不多赘述。我们通过一张图来了解线程池的工作原理：

合理利用线程池能够带来三个好处：

1. 降低资源消耗。减少了创建和销毁线程的次数，每个工作线程都可以被重复利用，可执行多个任务。

2. 提高响应速度。当任务到达时，任务可以不需要的等到线程创建就能立即执行。
3. 提高线程的可管理性。可以根据系统的承受能力，调整线程池中工作线线程的数目，防止因为消耗过多的内存，而把服务器累趴下(每个线程需要大约1MB内存，线程开的越多，消耗的内存也就越大，最后死机)。

1.3 线程池的使用

Java里面线程池的顶级接口是`java.util.concurrent.Executor`，但是严格意义上讲`Executor`并不是一个线程池，而只是一个执行线程的工具。真正的线程池接口是`java.util.concurrent.ExecutorService`。

要配置一个线程池是比较复杂的，尤其是对于线程池的原理不是很清楚的情况下，很有可能配置的线程池不是较优的，因此在`java.util.concurrent.Executors`线程工厂类里面提供了一些静态工厂，生成一些常用的线程池。官方建议使用Executors工程类来创建线程池对象。

Executors类中有个创建线程池的方法如下：

- `public static ExecutorService newFixedThreadPool(int nThreads)`：返回线程池对象。(创建的是有界线程池,也就是池中的线程个数可以指定最大数量)

获取到了一个线程池ExecutorService 对象，那么怎么使用呢，在这里定义了一个使用线程池对象的方法如下：

- `public Future<?> submit(Runnable task)`:获取线程池中的某一个线程对象，并执行

  > Future接口：用来记录线程任务执行完毕后产生的结果。

使用线程池中线程对象的步骤：

1. 创建线程池对象。
2. 创建Runnable接口子类对象。(task)
3. 提交Runnable接口子类对象。(take task)
4. 关闭线程池(一般不做)。

Runnable实现类代码：

```
public class MyRunnable implements Runnable {
    @Override
    public void run() {
        System.out.println("我要一个教练");
        try {
            Thread.sleep(2000);
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
        System.out.println("教练来了： " + Thread.currentThread().getName());
        System.out.println("教我游泳,教完后，教练回到了游泳池");
    }
}
```

线程池测试类：

```
public class ThreadPoolDemo {
    public static void main(String[] args) {
        // 创建线程池对象
```

```
        ExecutorService service = Executors.newFixedThreadPool(2);//包含2个线程对
象
        // 创建Runnable实例对象
        MyRunnable r = new MyRunnable();

        //自己创建线程对象的方式
        // Thread t = new Thread(r);
        // t.start(); ---> 调用MyRunnable中的run()

        // 从线程池中获取线程对象,然后调用MyRunnable中的run()
        service.submit(r);
        // 再获取个线程对象，调用MyRunnable中的run()
        service.submit(r);
        service.submit(r);
        // 注意：submit方法调用结束后，程序并不终止，是因为线程池控制了线程的关闭。
        // 将使用完的线程又归还到了线程池中
        // 关闭线程池
        //service.shutdown();
    }
}
```

Callable测试代码:

- `<T> Future<T> submit(Callable<T> task)` : 获取线程池中的某一个线程对象，并执行.

  Future : 表示计算的结果.
- `V get()` : 获取计算完成的结果。

```
public class ThreadPoolDemo2 {
    public static void main(String[] args) throws Exception {
        // 创建线程池对象
      ExecutorService service = Executors.newFixedThreadPool(2);//包含2个线程对象

        // 创建Runnable实例对象
        Callable<Double> c = new Callable<Double>() {
            @Override
            public Double call() throws Exception {
                return Math.random();
            }
        };

        // 从线程池中获取线程对象,然后调用Callable中的call()
        Future<Double> f1 = service.submit(c);
        // Futur 调用get() 获取运算结果
        System.out.println(f1.get());

        Future<Double> f2 = service.submit(c);
        System.out.println(f2.get());

        Future<Double> f3 = service.submit(c);
        System.out.println(f3.get());
    }
}
```

1.4 线程池的练习

需求: 使用线程池方式执行任务,返回1-n的和

分析: 因为需要返回求和结果,所以使用Callable方式的任务

代码:

```
public class Demo04 {
    public static void main(String[] args) throws ExecutionException,
InterruptedException {
        ExecutorService pool = Executors.newFixedThreadPool(3);

        SumCallable sc = new SumCallable(100);
        Future<Integer> fu = pool.submit(sc);
        Integer integer = fu.get();
        System.out.println("结果: " + integer);

        SumCallable sc2 = new SumCallable(200);
        Future<Integer> fu2 = pool.submit(sc2);
        Integer integer2 = fu2.get();
        System.out.println("结果: " + integer2);

        pool.shutdown();
    }
}
```

SumCallable.java

```
public class SumCallable implements Callable<Integer> {
    private int n;

    public SumCallable(int n) {
        this.n = n;
    }

    @Override
    public Integer call() throws Exception {
        // 求1-n的和
        int sum = 0;
        for (int i = 1; i <= n; i++) {
            sum += i;
        }
        return sum;
    }
}
```

第二章 死锁

2.1 什么是死锁

在多线程程序中,使用了多把锁,造成线程之间相互等待.程序不往下走了。

2.2 产生死锁的条件

1.有多把锁 2.有多个线程 3.有同步代码块嵌套

2.3 死锁代码

```
public class Demo05 {
    public static void main(String[] args) {
        MyRunnable mr = new MyRunnable();

        new Thread(mr).start();
        new Thread(mr).start();
    }
}

class MyRunnable implements Runnable {
    Object objA = new Object();
    Object objB = new Object();

    /*
    嵌套1 objA
    嵌套1 objB
    嵌套2 objB
    嵌套1 objA
     */
    @Override
    public void run() {
        synchronized (objA) {
            System.out.println("嵌套1 objA");
            synchronized (objB) {// t2, objA, 拿不到B锁,等待
                System.out.println("嵌套1 objB");
            }
        }

        synchronized (objB) {
            System.out.println("嵌套2 objB");
            synchronized (objA) {// t1 , objB, 拿不到A锁,等待
                System.out.println("嵌套2 objA");
            }
        }
    }
}
```

注意:**我们应该尽量避免死锁**

第三章 线程状态

3.1 线程状态概述

线程由生到死的完整过程：技术素养和面试的要求。

当线程被创建并启动以后，它既不是一启动就进入了执行状态，也不是一直处于执行状态。在线程的生命周期中，有几种状态呢？在API中`java.lang.Thread.State`这个枚举中给出了六种线程状态：

这里先列出各个线程状态发生的条件，下面将会对每种状态进行详细解析

线程状态	导致状态发生条件
NEW(新建)	线程刚被创建，但是并未启动。还没调用start方法。MyThread t = new MyThread只有线程对象，没有线程特征。
Runnable(可运行)	线程可以在java虚拟机中运行的状态，可能正在运行自己代码，也可能没有，这取决于操作系统处理器。调用了t.start()方法：就绪（经典教法）
Blocked(锁阻塞)	当一个线程试图获取一个对象锁，而该对象锁被其他的线程持有，则该线程进入Blocked状态；当该线程持有锁时，该线程将变成Runnable状态。
Waiting(无限等待)	一个线程在等待另一个线程执行一个（唤醒）动作时，该线程进入Waiting状态。进入这个状态后是不能自动唤醒的，必须等待另一个线程调用notify或者notifyAll方法才能够唤醒。
Timed Waiting(计时等待)	同waiting状态，有几个方法有超时参数，调用他们将进入Timed Waiting状态。这一状态将一直保持到超时期满或者接收到唤醒通知。带有超时参数的常用方法有Thread.sleep 、Object.wait。
Teminated(被终止)	因为run方法正常退出而死亡，或者因为没有捕获的异常终止了run方法而死亡。

我们不需要去研究这几种状态的实现原理，我们只需知道在做线程操作中存在这样的状态。那我们怎么去理解这几个状态呢，新建与被终止还是很容易理解的，我们就研究一下线程从Runnable（可运行）状态与非运行状态之间的转换问题。

3.2 睡眠sleep方法

我们看到状态中有一个状态叫做计时等待，可以通过Thread类的方法来进行演示.

`public static void sleep(long time)` 让当前线程进入到睡眠状态，到毫秒后自动醒来继续执行

```
public class Test{
    public static void main(String[] args){
        for(int i = 1;i<=5;i++){
            Thread.sleep(1000);
            System.out.println(i)
        }
    }
}
```

这时我们发现主线程执行到sleep方法会休眠1秒后再继续执行。

3.3 等待和唤醒

Object类的方法

`public void wait()` : 让当前线程进入到等待状态 此方法必须锁对象调用.

```
public class Demo1_wait {
    public static void main(String[] args) throws InterruptedException {
       // 步骤1 : 子线程开启,进入无限等待状态, 没有被唤醒,无法继续运行.
        new Thread(() -> {
            try {

                System.out.println("begin wait ....");
                synchronized ("") {
                    "".wait();
                }
                System.out.println("over");
            } catch (Exception e) {
            }
        }).start();
    }
```

`public void notify()` : 唤醒当前锁对象上等待状态的线程 此方法必须锁对象调用.

```
public class Demo2_notify {
    public static void main(String[] args) throws InterruptedException {
       // 步骤1 : 子线程开启,进入无限等待状态, 没有被唤醒,无法继续运行.
        new Thread(() -> {
            try {

                System.out.println("begin wait ....");
                synchronized ("") {
                    "".wait();
                }
                System.out.println("over");
            } catch (Exception e) {
            }
        }).start();

        //步骤2:  加入如下代码后, 3秒后,会执行notify方法, 唤醒wait中线程.
        Thread.sleep(3000);
        new Thread(() -> {
            try {
                synchronized ("") {
                    System.out.println("唤醒");
                    "".notify();
                }
            } catch (Exception e) {
            }
        }).start();
    }
}
```

3.4 等待唤醒案例（包子铺卖包子）

定义一个集合，包子铺线程完成生产包子，包子添加到集合中；吃货线程完成购买包子，包子从集合中移除。
1. 当包子没有时（包子状态为false），吃货线程等待.
2. 包子铺线程生产包子（即包子状态为true），并通知吃货线程（解除吃货的等待状态）

代码示例：

生成包子类：

```
public class BaoZiPu extends Thread{
    private List<String> list ;
    public BaoZiPu(String name,ArrayList<String> list){
        super(name);
        this.list = list;
    }
    @Override
    public void run() {
        int i = 0;
        while(true){
            //list作为锁对象
            synchronized (list){
                if(list.size()>0){
                    //存元素的线程进入到等待状态
                    try {
                        list.wait();
                    } catch (InterruptedException e) {
                        e.printStackTrace();
                    }
                }

                //如果线程没进入到等待状态 说明集合中没有元素
                //向集合中添加元素
                list.add("包子"+i++);
                System.out.println(list);
                //集合中已经有元素了 唤醒获取元素的线程
                list.notify();
            }
        }
    }
    }
}
```

消费包子类：

```
public class ChiHuo extends Thread {

    private List<String> list ;
    public ChiHuo(String name,ArrayList<String> list){
        super(name);
        this.list = list;
    }

    @Override
    public void run() {
```

```
        //为了能看到效果 写个死循环
        while(true){
            //由于使用的同一个集合 list作为锁对象
            synchronized (list){
                //如果集合中没有元素 获取元素的线程进入到等待状态
                if(list.size()==0){
                    try {
                        list.wait();
                    } catch (InterruptedException e) {
                        e.printStackTrace();
                    }
                }
                //如果集合中有元素 则获取元素的线程获取元素(删除)
                list.remove(0);
                //打印集合 集合中没有元素了
                System.out.println(list);
                //集合中已经没有元素 则唤醒添加元素的线程 向集合中添加元素
                list.notify();
            }
        }
    }
    }
}
```

测试类：

```
public class Demo {
    public static void main(String[] args) {
        //等待唤醒案例
        List<String> list = new ArrayList<>();
        // 创建线程对象
         BaoZiPu bzp = new BaoZiPu("包子铺",list);
        ChiHuo ch = new ChiHuo("吃货",list);
        // 开启线程
        bzp.start();
        ch.start();
    }
}
```

第四章 定时器

4.1 定时器概述

定时器，可以设置线程在某个时间执行某件事情，或者某个时间开始，每间隔指定的时间反复的做某件事情

4.2 定时器Timer类

java.util.Timer类：线程调度任务以供将来在后台线程中执行的功能。 任务可以安排一次执行，或者定期重复执行。

> 1.构造方法

```
public Timer()：构造一个定时器
```

> 2.成员方法

返回值	方法名	说明
void	schedule(TimerTask task, long delay)	在指定的延迟之后安排指定的任务执行。
void	schedule(TimerTask task, long delay, long period)	在指定 *的延迟*之后开始，重新 *执行固定延迟执行*的指定任务。
void	schedule(TimerTask task, Date time)	在指定的时间安排指定的任务执行。
void	schedule(TimerTask task, Date firstTime, long period)	从指定的时间开始，对指定的任务执行重复的 *固定延迟执行* 。

```
public class Test{
    public static void main(String[] args){
        //1.设置一个定时器，2秒后启动，只执行一次
        Timer t = new Timer();
        t.schedule(new TimerTask(){
            @Override
            public void run(){
                for(int i = 10;i >= 0 ; i--){
                    System.out.println("倒数：" + i);
                    try{
                        Thread.sleep(1000);
                    }catch(Exception e){}
                }
                System.out.println("嘭......嘭......");
                //任务执行完毕，终止计时器
                t.cancel();
            }
        },2000);

        //2.设置一个定时器，5秒后开始执行，每一秒执行一次
        Timer t2 = new Timer();
        t2.schedule(new TimerTask(){
            @Override
            public void run(){
                SimpleDateFormat sdf = new SimpleDateFormat("yyyy-MM-dd
HH:mm:ss");
                System.out.println(sdf.format(new Date()));
            }
        },5000,1000);

        //3.设置一个定时器，在2030年01月01日零时开始执行，每隔24小时执行一次
        Timer t3 = new Timer();
        Calendar c = new GregorianCalendar(2030, 1-1, 1, 0, 0, 0);
        Date d = c.getTime();

        t3.schedule(new TimerTask(){
            @Override
```

```
            public void run(){
                System.out.println("搜索全盘......");
            }
        },d,1000 * 3600 * 24);
    }
}
```