# 多线程-2

## 一、学习目标

**理解多线程常用方法**

**理解** 线程池 会

**理解** CAS自旋锁

**理解** 乐观锁和悲观锁原理

## 二、学习内容

### 1、多线程的常用方法

#### 1.1 设置和获取线程名字

方法

|  |  |
| --- | --- |
| 方法名 | 说明 |
| void setName(String name) | 将此线程的名称更改为等于参数name |
| String getName() | 返回此线程的名称 |
| Thread currentThread() | 返回对当前正在执行的线程对象的引用 |

代码示例

线程类：  
public class MyThread extends Thread {  
  
 public MyThread() {  
  
 }  
   
 public MyThread(String name) {  
 super(name);  
 }  
  
 @Override  
 public void run() {  
 for (int i = 1; i <= 100; i++) {  
 System.out.println(getName()+":"+i);  
 }  
 }  
}  
  
测试类：  
public class MyThreadDemo {  
 public static void main(String[] args) {  
 MyThread m1 = new MyThread();  
 MyThread m2 = new MyThread();  
 //void setName(String name)：将此线程的名称更改为等于参数 name  
 m1.setName("高铁");  
 m2.setName("飞机");  
   
 m1.start();  
 m2.start();  
  
 //Thread(String name) 通过构造方法把线程名称设置进行  
 MyThread my1 = new MyThread("高铁");  
 MyThread my2 = new MyThread("飞机");  
   
 my1.start();  
 my2.start();  
  
 //static Thread currentThread() 返回对当前正在执行的线程对象的引用  
 System.out.println(Thread.currentThread().getName());  
 }  
}

#### 1.2 线程休眠

|  |  |
| --- | --- |
| 方法名 | 说明 |
| static void sleep(long millis) | 使当前正在执行的线程停留（暂停执行）指定的毫秒数 |

代码示例

public class ThreadSleepDemo {  
  
 public static void main(String[] args) throws InterruptedException {  
 for (int i = 1; i <=100 ; i++) {  
 System.out.println("每隔一秒执行一次"+i);  
  
 //sleep(long millis) 使当前正在执行的线程停留（暂停执行）指定的毫秒数  
 //参数为毫秒级， 1秒=1000毫秒  
 Thread.sleep(1000);  
 }  
 }  
}

#### 1.3 线程优先级

* 每一个 Java 线程都有一个优先级，这样有助于操作系统确定线程的调度顺序
* Java 线程的优先级是一个整数，其取值范围是 1 - 10
* 每一个线程都会分配一个默认优先级 5
* 具有较高优先级的线程对程序更重要，并且应该在低优先级的线程之前分配处理器资源。但是，线程优先级不能保证线程执行的顺序，而且非常依赖于平台
* 具有很强的随机性，假如计算机只有一个 CPU，那么 CPU 在某一个时刻只能执行一条指令，线程只有得到CPU时间片，也就是使用权，才可以执行指令。所以说多线程程序的执行是有随机性，因为谁抢到CPU的使用权是不一定的

优先级相关方法

|  |  |
| --- | --- |
| 方法名 | 说明 |
| final int getPriority() | 返回此线程的优先级 |
| final void setPriority(int newPriority) | 设置此线程的优先级，线程默认优先级是5；线程优先级的范围是：1-10 |

代码示例

public class MyThread implements Runnable {  
 @Override  
 public void run() {  
 for (int i = 1; i <=100 ; i++) {  
 System.out.println(Thread.currentThread().getName()+"抢到了执行权"+i);  
 }  
 }  
}  
  
public class MyThreadDemo {  
 public static void main(String[] args) {  
 //创建MyThread对象，线程干活的类  
 MyThread mt = new MyThread();  
 //创建Thread线程对象  
 Thread t1 = new Thread(mt, "线程一");  
 Thread t2 = new Thread(mt, "线程二");  
  
 //设置线程一的优先级为10 最高  
 t1.setPriority(10);  
 //启动线程  
 t1.start();  
 t2.start();  
 }  
}

#### 1.4 守护线程

Java分为两种线程：

1. 用户线程
2. 守护线程；

所谓守护线程是指在程序运行的时候在后台提供一种通用服务的线程，当所有的非守护线程结束时，程序也就终止了，同时会杀死进程中的所有守护线程。反过来说，只要任何非守护线程还在运行，程序就不会终止。

守护线程和用户线程的没什么本质的区别：如果用户线程已经全部退出运行了，只剩下守护线程存在了， 因为没有了被守护者，守护线程也就没有工作可做了，也就没有继续运行程序的必要了。

设置守护线程方法

|  |  |
| --- | --- |
| 方法名 | 说明 |
| void setDaemon(boolean on) | 将此线程标记为守护线程，当运行的线程都是守护线程时，Java虚拟机将退出 |

代码示例

public class MyThread1 extends Thread {  
 @Override  
 public void run() {  
 for (int i = 1; i <= 10; i++) {  
 System.out.println(getName() + "在执行" + i);  
 }  
 }  
}  
  
public class MyThread2 extends Thread {  
 @Override  
 public void run() {  
 for (int i = 1; i <= 100; i++) {  
 System.out.println(getName() + "在守护" + i);  
 }  
 }  
}  
  
public class MyThreadDemo {  
 public static void main(String[] args) {  
 //创建线程对象  
 MyThread1 m1 = new MyThread1();  
 MyThread2 m2 = new MyThread2();  
 //设置线程名称  
 m1.setName("女神");  
 m2.setName("备胎");  
 //设置守护线程，当普通线程执行完成之后，守护线程也没有执行下去的必要了  
 m2.setDaemon(true);  
 //启动线程  
 m1.start();  
 m2.start();  
 }  
}

#### 1.5 小结

设置线程名称

setName(String name)

获取线程名称

getName()

线程休眠

sleep(long millis)

### 2.线程池

#### 2.1 线程池概述

##### 2.1.1 概述

线程池：Java中开辟出了一种管理线程的概念，这个概念叫做线程池。

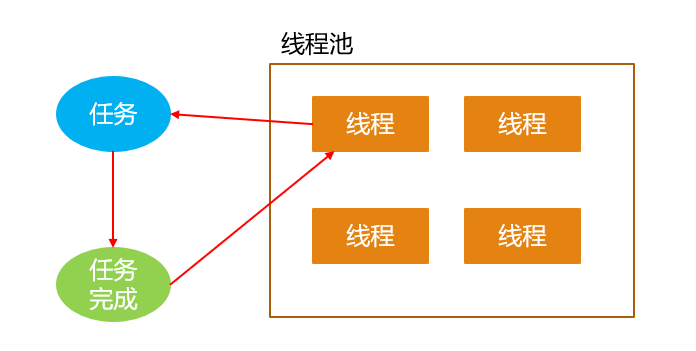
举个例子：我们吃饭用的碗，把它当做一个线程，用来装碗的碗柜就是一个线程池。我们每次吃完饭后不会把碗扔掉，而是把碗放在碗柜方便下次再次使用。如果把碗扔掉了，下次再吃饭就必须得再买碗，成本会很高。所以碗柜的作用就体现出来了， 就是存放碗，管理碗，降低成本，减少消耗。线程池就是这个碗柜，线程池的好处，就是可以方便的管理线程，也可以减少内存的消耗。

##### 2.1.2 线程池存在的意义

系统创建一个线程的成本是比较高的，因为它涉及到与操作系统交互，当程序中需要创建大量生存期很短暂的线程时，频繁的创建和销毁线程对系统的资源消耗有可能大于业务处理是对系统资源的消耗，这样就有点"舍本逐末"了。针对这一种情况，为了提高性能，我们就可以采用线程池。线程池在启动的时，会创建大量空闲线程，当我们向线程池提交任务的时，线程池就会启动一个线程来执行该任务。等待任务执行完毕以后，线程并不会死亡，而是再次返回到线程池中称为空闲状态。等待下一次任务的执行。

##### 2.1.3 线程池的设计思路

1. 准备一个任务容器
2. 一次性启动多个消费者线程
3. 刚开始任务容器是空的，所以线程都在wait
4. 直到一个外部线程向这个任务容器中扔了一个"任务"，就会有一个消费者线程被唤醒
5. 这个消费者线程取出"任务"，并且执行这个任务，执行完毕后，继续等待下一次任务的到来

* 

#### 2.2 线程池使用方式及详细讲解

概述 : JDK对线程池也进行了相关的实现，在真实企业开发中我们也很少去自定义线程池，而是使用JDK中自带的线程池。

##### 2.2.1 线程池之Executors默认线程池

Executors方法

|  |  |
| --- | --- |
| 方法 | 说明 |
| static ExecutorService newCachedThreadPool() | 创建一个默认的线程池 |
| static ExecutorService newFixedThreadPool(int nThreads) | 创建一个指定最多线程数量的线程池 |

ExecutorService方法

|  |  |
| --- | --- |
| 方法 | 说明 |
| submit(Runnable r) | 调取线程执行任务 |
| shutdown() | 关闭线程池 |

线程池操作步骤

1. 创建线程池对象
2. 定义任务类，创建线程任务对象
3. 调取线程执行任务
4. 关闭线程池

代码示例

public class ExecutorsDemo01 {  
 public static void main(String[] args) throws InterruptedException {  
 //1.创建线程池对象  
 ExecutorService executorService1 = Executors.newCachedThreadPool();  
 //2.创建线程任务对象  
 MyRunnable mr = new MyRunnable();  
 //3.从线程池中调取线程执行任务  
 executorService1.submit(mr);  
// Thread.sleep(1000);  
 executorService1.submit(mr);  
 executorService1.submit(mr);  
 //4.关闭线程池  
 executorService1.shutdown();  
  
 }  
}  
public class ExecutorsDemo02 {  
 public static void main(String[] args) {  
 //1.创建线程池对象  
 ExecutorService executorService2 = Executors.newFixedThreadPool(2);  
 //2.创建线程任务对象  
 MyRunnable mr = new MyRunnable();  
 //3.从线程池中调取线程执行任务  
 executorService2.submit(mr);  
 executorService2.submit(mr);  
 executorService2.submit(mr);  
 //4.关闭线程池  
 executorService2.shutdown();  
 }  
}

##### 2.2.2 线程池之ThreadPoolExecutor

ThreadPoolExecutor对象构造方法

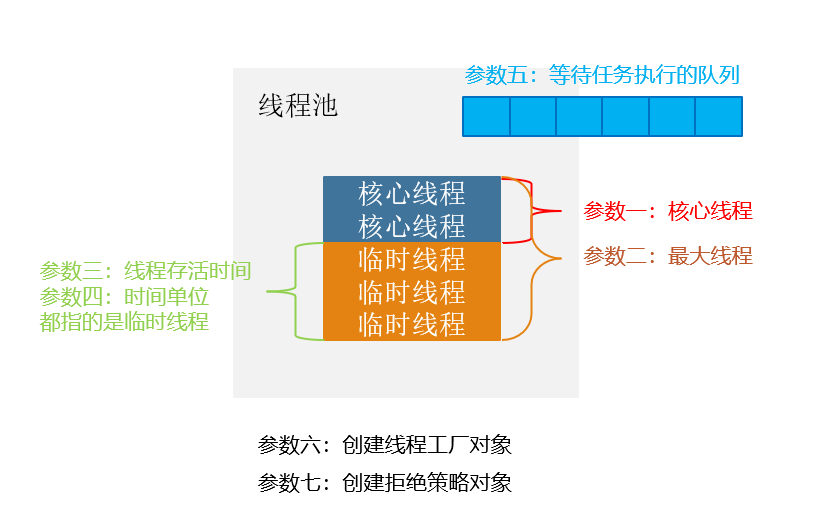
|  |  |
| --- | --- |
| 构造方法 | 说明 |
| public ThreadPoolExecutor(  int corePoolSize,   int maximumPoolSize,  long keepAliveTime,  TimeUnit unit,  BlockingQueue workQueue,  ThreadFactory threadFactory,  RejectedExecutionHandler handler) | 创建一个线程池对象，参数： 核心线程数量， 最大线程数量， 空闲线程最大存活时间， 时间单位， 任务队列， 创建线程工厂， 任务的拒绝策略 |

代码示例

public class ThreadPoolExecutorDemo01 {  
  
 /\*  
 参数一：核心线程数量  
 参数二：最大线程数  
 参数三：空闲线程最大存活时间  
 参数四：时间单位  
 参数五：任务队列  
 参数六：创建线程工厂  
 参数七：任务的拒绝策略  
 \*/  
 public static void main(String[] args) {  
 ThreadPoolExecutor pool = new ThreadPoolExecutor(  
 2,  
 5,  
 2,  
 TimeUnit.SECONDS,  
 new ArrayBlockingQueue<>(10),  
 Executors.defaultThreadFactory(),  
 new ThreadPoolExecutor.AbortPolicy());  
  
 MyRunnable mr = new MyRunnable();  
 pool.submit(mr);  
 pool.submit(mr);  
  
 pool.shutdown();  
 }  
}

##### 2.2.3 ThreadPoolExecutor参数详解

public ThreadPoolExecutor(int corePoolSize,  
 int maximumPoolSize,  
 long keepAliveTime,  
 TimeUnit unit,  
 BlockingQueue<Runnable> workQueue,  
 ThreadFactory threadFactory,  
 RejectedExecutionHandler handler)  
   
corePoolSize： 核心线程的最大值，不能小于0  
maximumPoolSize：最大线程数，不能小于等于0，maximumPoolSize >= corePoolSize  
keepAliveTime： 空闲线程最大存活时间,不能小于0  
unit： 时间单位  
workQueue： 任务队列，不能为null  
threadFactory： 创建线程工厂,不能为null   
handler： 任务的拒绝策略,不能为null



##### 2.2.4 线程池之拒绝策略

线程池中，有三个重要的参数，决定影响了拒绝策略： 核心线程数、任务队列 、 最大线程数。

当提交任务数大于**核心线程数**的时候，会优先将任务放到**任务队列**中。当**任务队列**饱和后，会扩充线程池中线程数，直到达到最大线程数配置。此时，再多余的任务，则会触发线程池的拒绝策略了。

总结起来就是 **当提交的任务数大于（任务队列+最大线程数 ），就会触发线程池的拒绝策略**。

RejectedExecutionHandler是jdk提供的一个任务拒绝策略接口，它下面存在4个子类

|  |  |
| --- | --- |
| 策略 | 说明 |
| ThreadPoolExecutor.AbortPolicy | 丢弃任务并抛出RejectedExecutionException异常。是默认的策略。 |
| ThreadPoolExecutor.DiscardPolicy | 丢弃任务，但是不抛出异常(不推荐) |
| ThreadPoolExecutor.DiscardOldestPolicy | 抛弃队列中等待最久的任务 然后把当前任务加入队列中 |
| ThreadPoolExecutor.CallerRunsPolicy | 调用任务的run()方法绕过线程池直接执行 |

#### 2.3 小结

线程池存在的意义

* 方便管理线程
* 减少内存的内耗

### 3、线程的原子性

#### 3.1 原子性概述

所谓的原子性是指在一次操作或者多次操作中，要么所有的操作全部都得到了执行并且不会受到任何因素的干扰而中断，要么所有的操作都不执行，多个操作是一个不可以分割的整体。

代码示例

public class MyThread implements Runnable {  
 private int count = 0; //送冰淇淋的数量  
  
 @Override  
 public void run() {  
 for (int i = 0; i < 100; i++) {  
 //1,从共享数据中读取数据到本线程栈中.  
 //2,修改本线程栈中变量的值  
 //3,会把本线程栈中变量的值赋值给共享数据.  
 count++;  
 System.out.println("已经送了" + count + "个冰淇淋");  
 **try** {  
 Thread.*sleep*(200);  
 } **catch** (InterruptedException e) {  
 e.printStackTrace();  
 }

}  
 }  
}  
  
public class Demo01 {  
 public static void main(String[] args) {  
 MyThread atom = new MyThread();  
  
 for (int i = 0; i < 3; i++) {  
 //每次循环都开启一条线程  
 new Thread(atom).start();  
 }  
 }  
}

**出现问题：**count++ 不是一个原子性操作, 他在执行的过程中,有可能被其他线程打断

#### 3.2 AtomicInteger对象的原子性

##### 3.2.1 AtomicInteger介绍

java从JDK1.5开始提供了java.util.concurrent.atomic包(简称Atomic包)，这个包中的原子操作类提供了一种用法简单，性能高效，线程安全地更新一个变量的方式。因为变量的类型有很多种，所以在Atomic包里一共提供了13个类，属于4种类型的原子更新方式，分别是原子更新基本类型、原子更新数组、原子更新引用和原子更新属性(字段)。我们以AtomicInteger为例进行讲解

##### 3.2.2 AtomicInteger构造方法

构造方法

|  |  |
| --- | --- |
| 构造方法 | 说明 |
| public AtomicInteger() | 初始化一个默认值为0的原子型Integer |
| public AtomicInteger(int initialValue) | 初始化一个指定值的原子型Integer |

代码示例

public class AtomicIntegerDemo01 {  
 public static void main(String[] args) {  
 //public AtomicInteger() 初始化一个默认值为0的原子型Integer  
 AtomicInteger ac = new AtomicInteger();  
 System.out.println(ac);  
  
 //public AtomicInteger(int initialValue) 初始化一个指定值的原子型Integer  
 AtomicInteger ac2 = new AtomicInteger(10);  
 System.out.println(ac2);  
 }  
}

##### 4.2.3 AtomicInteger常用方法

常用方法

|  |  |
| --- | --- |
| 方法 | 说明 |
| int get() | 获取值 |
| int getAndIncrement() | 以原子方式将当前值加1，注意，这里返回的是自增前的值 |
| int incrementAndGet() | 以原子方式将当前值加1，注意，这里返回的是自增后的值 |
| int addAndGet(int data) | 以原子方式将输入的数值与实例中的值（AtomicInteger里的value）相加，并返回结果。 |
| int getAndSet(int value) | 以原子方式设置为newValue的值，并返回旧值 |

代码示例

public class AtomicIntegerDemo02 {  
  
 public static void main(String[] args) {  
 AtomicInteger ac1 = new AtomicInteger(10);  
 //int get() 获取值  
 System.out.println(ac1.get());  
  
 AtomicInteger ac2 = new AtomicInteger(10);  
 //int getAndIncrement() 以原子方式将当前值加1，注意，这里返回的是自增前的值  
 int andIncrement = ac2.getAndIncrement();  
 System.out.println(andIncrement);  
 System.out.println(ac2.get());  
  
 AtomicInteger ac3 = new AtomicInteger(10);  
 //int incrementAndGet() 以原子方式将当前值加1，注意，这里返回的是自增后的值。  
 int incrementAndGet = ac3.incrementAndGet();  
 System.out.println(incrementAndGet);//自增后的值  
 System.out.println(ac3.get());  
  
 AtomicInteger ac4 = new AtomicInteger(10);  
 //int addAndGet(int data) 以原子方式将参数与对象中的值相加，并返回结果。  
 int addAndGet = ac4.addAndGet(20);  
 System.out.println(addAndGet);  
 System.out.println(ac4.get());  
  
 AtomicInteger ac5 = new AtomicInteger(100);  
 //int getAndSet(int value) 以原子方式设置为newValue的值，并返回旧值。  
 int andSet = ac5.getAndSet(20);  
 System.out.println(andSet);  
 System.out.println(ac5.get());  
 }  
}

##### 3.2.3 使用AtomicInteger解决上述原子性问题

代码示例

public class MyThread implements Runnable {  
// private int count = 0; //送冰淇淋的数量  
 private AtomicInteger atom = new AtomicInteger();  
  
 @Override  
 public void run() {  
 for (int i = 0; i < 100; i++) {  
// count++;  
 int count = atom.incrementAndGet();  
 System.out.println("已经送了" + count + "个冰淇淋");  
 **try** {  
 Thread.*sleep*(200);  
 } **catch** (InterruptedException e) {  
 e.printStackTrace();  
 }

}  
 }  
}  
  
public class Demo01 {  
 public static void main(String[] args) {  
 MyThread atom = new MyThread();  
  
 for (int i = 0; i < 3; i++) {  
 //每次循环都开启一条线程  
 new Thread(atom).start();  
 }  
 }  
}

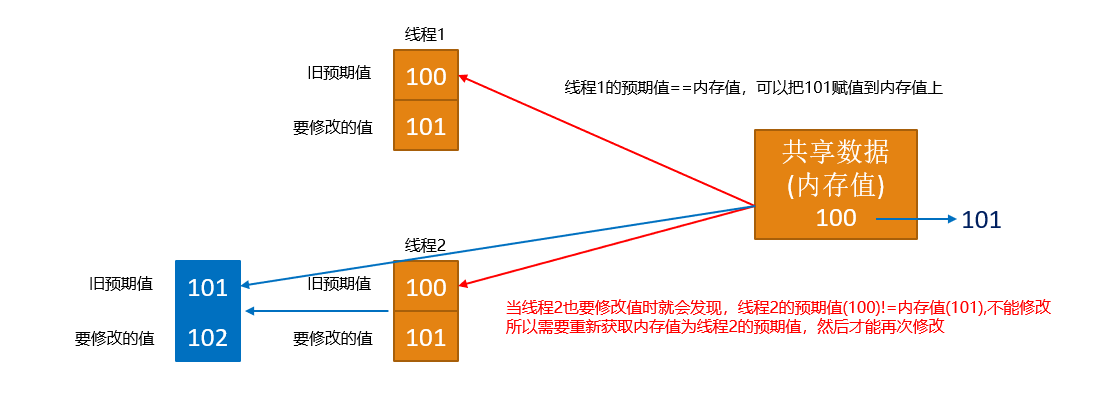
#### 3.3 CAS自旋锁

##### 3.3.1 AtomicInteger原理

**AtomicInteger原理 :** 自旋锁 + CAS 算法

**CAS算法：**

* 有3个操作数（内存值V， 旧的预期值A，要修改的值B）
* 当旧的预期值A == 内存值 此时修改成功，将V改为B
* 当旧的预期值A！=内存值 此时修改失败，不做任何操作
* 并重新获取现在的最新值（这个重新获取的动作就是自旋）



##### 3.3.2 AtomicInteger源码解析

源码解析

//先自增，然后获取自增后的结果  
public final int incrementAndGet() {  
 //+ 1 自增后的结果  
 //this 就表示当前的atomicInteger（值）  
 //1 自增一次  
 return U.getAndAddInt(this, VALUE, 1) + 1;  
}  
  
public final int getAndAddInt(Object o, long offset, int delta) {  
 //v 旧值  
 int v;  
 //自旋的过程  
 do {  
 //不断的获取旧值  
 v = getIntVolatile(o, offset);  
 //如果这个方法的返回值为false，那么继续自旋  
 //如果这个方法的返回值为true，那么自旋结束  
 //o 表示的就是内存值  
 //v 旧值  
 //v + delta 修改后的值  
 } while (!weakCompareAndSetInt(o, offset, v, v + delta));  
 //作用：比较内存中的值，旧值是否相等，如果相等就把修改后的值写到内存中，返回true。表示修改成功。  
 //如果不相等，无法把修改后的值写到内存中，返回false。表示修改失败。  
 //如果修改失败，那么继续自旋。  
 return v;  
}

#### 3.4 悲观锁和乐观锁

**synchronized(悲观锁)和CAS(乐观锁)的区别:**

**相同点：**在多线程情况下，都可以保证共享数据的安全性。

**不同点：**

* synchronized总是从最坏的角度出发，认为每次获取数据的时候，别人都有可能修改。所以在每 次操作共享数据之前，都会上锁。（悲观锁）
* cas是从乐观的角度出发，假设每次获取数据别人都不会修改，所以不会上锁。只不过在修改共享数据的时候，会检查一下，别人有没有修改过这个数据
  + 如果别人修改过，那么我再次获取现在最新的值
  + 如果别人没有修改过，那么我现在直接修改共享数据的值.(乐观锁）