# 1.线程

## 1.1知识引导

1）学习String时知道，StringBuffer是线程安全的，StringBuilder是线程不安全的。

2）学习集合时知道，ArrayList，LinkedList是线程不安全的，vectory是线程安全的，Set集合，hashSet是线程不安全的。

3）学习映射的时候，知道hashMap是线程不安全的，hashTable是线程安全的

## 1.2进程引入

当我们使用QQ时，我们会点击QQ，此时计算机问我们QQ就是创建了进程，进程也就是我们想启动一个软件，打开这个软件，我们就相应的启动了一个进程。

通过任务管理器可以看到我们启动的进程，如图1

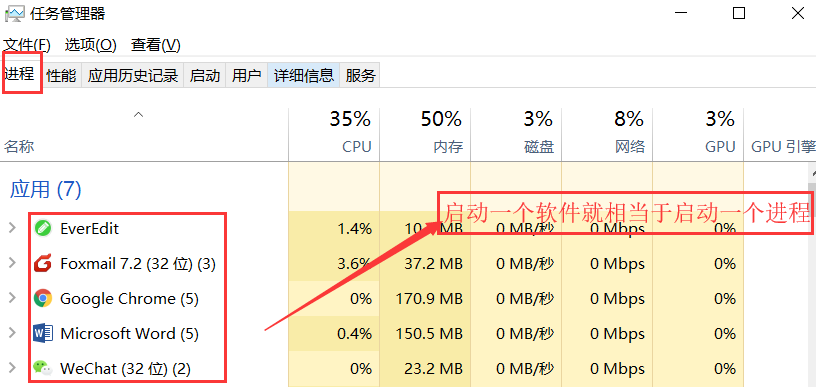


图-1

1.2.1进程概念

进程就是计算机在执行的任务。

## 1.3线程引入

启动QQ以后，我们启动一个聊天进程，我们就可以使用QQ跟我们的好友进行聊天，我们知道QQ可以同时跟很多好友聊天，那我们现在只启动了一个聊天进程，怎么实现跟多个好友聊天呢？其实我们计算机其实把和多个人聊天分成小任务，我们的每个小任务就是使用我们的线程来实现的，每一个好友的聊天窗口就是我们的一个小任务。

1.3.1线程概念

线程：进程中执行的某段逻辑或者任务。

1.3.2多线程引入

使用QQ跟多个好友聊天时，聊天进程中会创建多个线程执行聊天任务。

1.3.3多线程实例

微信：在语音聊天时，可以跟其他好友发图片，发文字

迅雷：我们可以同时下载多部电影，多部电影同时下载

## 1.4线程的创建方式

线程创建方式有3种，分别是继承线程的顶级父类；实现Runnable接口；实现callable接口。

1.4.1继承Thread

继承线程的顶级父类，继承后这个类就变成了线程类，继承后需要实现run()方法。

代码实现

public class ThreadDemo {

public static void main(String[] args) {

TDemo td = new TDemo();

// 启动这个线程---表示这个线程是一个单独的逻辑

td.start();

for (int i = 0; i < 10; i++) {

System.out.println("main:" + i);

}

}

}

class TDemo extends Thread {

//线程将要执行的任务

@Override

public void run() {

for (int i = 0; i < 10; i++) {

System.out.println("Thread:" + i);

}

}

1.4.2实现Runnable接口

实现Runnable，重写run方法，需要利用Thread类来启动这个线程

代码实现

public class RunnableDemo {

public static void main(String[] args) {

RDemo rd = new RDemo();

Thread t = new Thread(rd);

t.start();

}

}

class RDemo implements Runnable {

//线程将要执行的任务

@Override

public void run() {

for (int i = 0; i < 10; i++) {

System.out.println(i);

}

}

}

1.4.3实现callable接口

实现Callable<R>，重写call方法---现阶段要求了解即可。

代码实现

import java.util.concurrent.Callable;

import java.util.concurrent.ExecutorService;

import java.util.concurrent.Executors;

import java.util.concurrent.Future;

public class CallableDemo {

public static void main(String[] args) throws Exception {

ExecutorService es = Executors.newCachedThreadPool();

Future<String> f = es.submit(new CDemo());

System.out.println(f.get());

}

}

class CDemo implements Callable<String> {

@Override

public String call() throws Exception {

return "hahah~~~";

}

}

## 1.5 run方法和start方法

我们知道要把执行的任务逻辑放到run方法中，调用start方法可以开启线程，现在在main方法中使用start方法来启动线程和直接调用run方法有什么区别

代码实现

public class ThreadDemo {

public static void main(String[] args) {

TDemo td = new TDemo();

// 启动这个线程---表示这个线程是一个单独的逻辑

td.start();

//td.run();

for (int i = 0; i < 10; i++) {

System.out.println("main:" + i);

}

}

}

class TDemo extends Thread {

@Override

public void run() {

for (int i = 0; i < 10; i++) {

System.out.println("Thread:" + i);

}

}

直接调用run方法，现在只启动了main线程，我们自定义的线程并没有启动，执行结果如下



## 1.6多线程引入

问题描述：现有一个公园开业，进入公园需要购票，今日开四个窗口售票，有四个售票员同时售票，公园门票只有100张。

代码实现

public class SellDemo {

public static void main(String[] args) {

Ticket t = new Ticket();

t.setCount(100);

Seller s1 = new Seller(t);

new Thread(s1, "售票员1号").start();

Seller s2 = new Seller(t);

new Thread(s2, "售票员2号").start();

Seller s3 = new Seller(t);

new Thread(s3, "售票员3号").start();

Seller s4 = new Seller(t);

new Thread(s4, "售票员4号").start();

}

}

class Ticket {

private int count;

public int getCount() {

return count;

}

public void setCount(int count) {

this.count = count;

}

}

class Seller implements Runnable {

private Ticket t;

public Seller(Ticket t) {

this.t = t;

}

@Override

public void run() {

while (t.getCount() >0) {

try {

Thread.sleep(5);

} catch (InterruptedException e) {

e.printStackTrace();

}

t.setCount(t.getCount()-1);

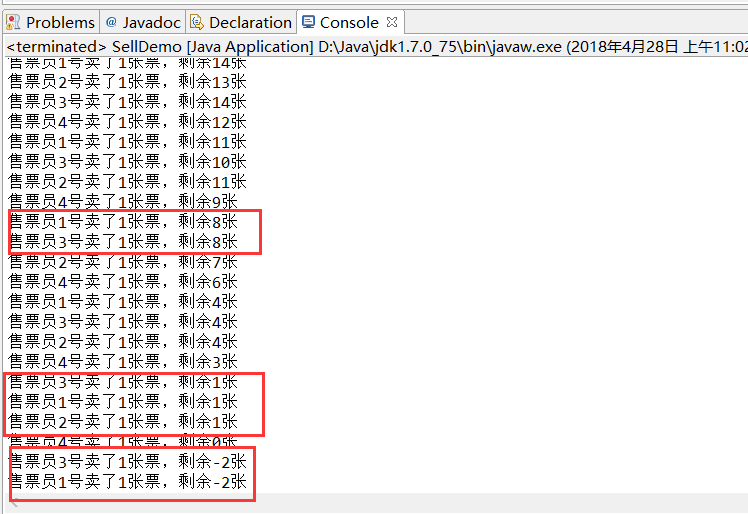
System.out.println(Thread.currentThread().getName() + "卖了1张票，剩余" + t.getCount()+"张");

}

}

}

执行结果：

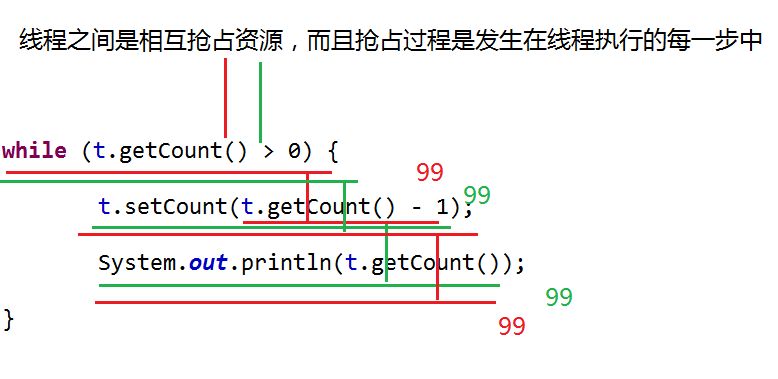


在上述执行结果总出现了重复数字问题，出现负数的情况，这就是我们在有多个线程并发安全的问题。

## 1.7多线程的并发安全问题

1.7.1并发安全问题分析

**线程执行顺序：**线程的执行不存在先后，相互抢占执行的，抢占并不是在发生在线程执行的开始，而是发生在线程执行的每一步过程中。



**说明：**现在有小红和小绿两条线程，两条线程同时到大while循环，满足条件进入循环后，两条线程拿到的值都是100，进行减1操作后，值都为99，接着两条线程去执行，输出语句，所以就会产生打印数字重复，这就是我们线程安全的问题。

## 1.8同步锁机制

使用同步锁机制解决多线程安全问题，synchronized修饰的同步代码块，同步代码块锁对象必须是所有线程所共享的，共享资源，类的字节码和this关键字，使用this关键字锁对象需要在特殊的情况下下才能保证安全问题。

synchronized也可以修饰方法，修饰方法的锁对象为this。

代码实现

public class SellDemo {

public static void main(String[] args) {

Ticket t = new Ticket();

t.setCount(100);

Seller s1 = new Seller(t);

new Thread(s1, "售票员1号").start();

Seller s2 = new Seller(t);

new Thread(s2, "售票员2号").start();

Seller s3 = new Seller(t);

new Thread(s3, "售票员3号").start();

Seller s4 = new Seller(t);

new Thread(s4, "售票员4号").start()；

// Seller s = new Seller(t);

// new Thread(s).start();

// new Thread(s).start();

// new Thread(s).start();

// new Thread(s).start();

}

}

class Ticket {

private int count;

public int getCount() {

return count;

}

public void setCount(int count) {

this.count = count;

}

}

class Seller implements Runnable {

private Ticket t;

public Seller(Ticket t) {

this.t = t;

}

@Override

public void run() {

while (true) {

synchronized (t) {

// synchronized (Seller.class) {

// synchronized (Date.class) {

// synchronized (this) {

if (t.getCount() <= 0) {

break;

}

try {

Thread.sleep(5);

} catch (InterruptedException e) {

e.printStackTrace();

}

t.setCount(t.getCount() - 1);

System.out.println(Thread.currentThread().getName() + "卖了1张票，剩余" + t.getCount());

}

}

}

}

## 1.9 死锁

由于多个线程执行并且线程之间的锁形成嵌套而导致程序无法继续运行的情况—死锁

死锁代码

public class DeadLockDemo {

static Printer p = new Printer();

static Scanner s = new Scanner();

public static void main(String[] args) {

new Thread(new Runnable() {

@Override

public void run() {

synchronized (s) {

s.scan();

try {

Thread.sleep(10);

} catch (InterruptedException e) {

e.printStackTrace();

}

synchronized (p) {

p.print();

}

}

}

}).start();

new Thread(new Runnable() {

@Override

public void run() {

synchronized (p) {

p.print();

try {

Thread.sleep(10);

} catch (InterruptedException e) {

e.printStackTrace();

}

synchronized (s) {

s.scan();

}

}

}

}).start();

}

}

class Scanner {

public void scan() {

System.out.println("扫描仪在吱呦吱呦的扫描~~~");

}

}

class Printer {

public void print() {

System.out.println("打印机在哼哧哼哧的打印~~~");

}

}

1.9.1死锁产生的原因：

多线程，锁对象不统一、共享资源比较多、锁嵌套

1.9.2 避免死锁

减少线程数量、减少共享资源、统一锁对象减少锁嵌套

## 1.10 同步/异步

1.10.1同步/异步

同步：在同一个时间段内只允许一个线程执行任务。

异步：在同一个时间段内允许多个线程去抢占执行任务。

1.10.2思考

同步一定是线程安全的，线程安全不一定是同步，线程不安全一定异步，异步的不一定线程不安全

## 1.11 wait，notify和notifyALl方法

wait()、notify()和notifyAll()是Object类中的方法：

从这三个方法的文字描述可以知道以下几点信息：

1）wait()、notify()和notifyAll()方法是本地方法，并且为final方法，无法被重写。

2）调用某个对象的wait()方法能让当前线程阻塞，并且当前线程必须拥有此对象的monitor（即锁）

3）调用某个对象的notify()方法能够唤醒一个正在等待这个对象的monitor的线程，如果有多个线程都在等待这个对象的monitor，则只能唤醒其中一个线程；

4）调用notifyAll()方法能够唤醒所有正在等待这个对象的monitor的线程；

WaitNotify代码实现

public class WaitNotifyDemo {

public static void main(String[] args) {

Student s = new Student();

s.name = "酸菜";

s.gender = '男';

new Thread(new Ask(s)).start();

new Thread(new Change(s)).start();

}

}

class Ask implements Runnable {

private Student s;

public Ask(Student s) {

this.s = s;

}

@Override

public void run() {

synchronized (s) {

while (true) {

if (s.flag == false) {

try {

// 让当前线程陷入等待

s.wait();

} catch (InterruptedException e) {

e.printStackTrace();

}

}

System.out.println(s.name + ":" + s.gender);

System.out.println("问了一个问题~~~");

s.flag = false;

s.notify();

}

}

}

}

class Change implements Runnable {

private Student s;

public Change(Student s) {

this.s = s;

}

@Override

public void run() {

synchronized (s) {

while (true) {

if (s.flag == true) {

try {

s.wait();

} catch (InterruptedException e) {

e.printStackTrace();

}

}

if (s.gender == '男') {

s.name = "翠花";

s.gender = '女';

} else {

s.name = "酸菜";

s.gender = '男';

}

s.flag = true;

// 唤醒在等待的线程

s.notify();

}

}

}

}

class Student {

String name;

char gender;

// 标记位

// 当程序中依靠已有的变量无法确定执行顺序后者作为控制条件的时候，需要第三方变量来作为开关使用

// 这个开关称之为标记位

boolean flag = true;

}

WaitNotifyAll代码实现

public class WaitNotifyAllDemo {

public static void main(String[] args) {

Student s = new Student();

s.name = "酸菜";

s.gender = '男';

new Thread(new Ask2(s)).start();

new Thread(new Ask2(s)).start();

new Thread(new Change2(s)).start();

new Thread(new Change2(s)).start();

}

}

class Ask2 implements Runnable {

private Student s;

public Ask2(Student s) {

this.s = s;

}

@Override

public void run() {

synchronized (s) {

while (true) {

while (s.flag == false) {

try {

// 让当前线程陷入等待

s.wait();

} catch (InterruptedException e) {

e.printStackTrace();

}

}

System.out.println(s.name + ":" + s.gender);

System.out.println("问了一个问题~~~");

s.flag = false;

// 表示唤醒所有的线程

s.notifyAll();

}

}

}

}

class Change2 implements Runnable {

private Student s;

public Change2(Student s) {

this.s = s;

}

@Override

public void run() {

synchronized (s) {

while (true) {

while (s.flag == true) {

try {

s.wait();

} catch (InterruptedException e) {

e.printStackTrace();

}

}

if (s.gender == '男') {

s.name = "翠花";

s.gender = '女';

} else {

s.name = "酸菜";

s.gender = '男';

}

s.flag = true;

// 唤醒在等待的线程

s.notifyAll();

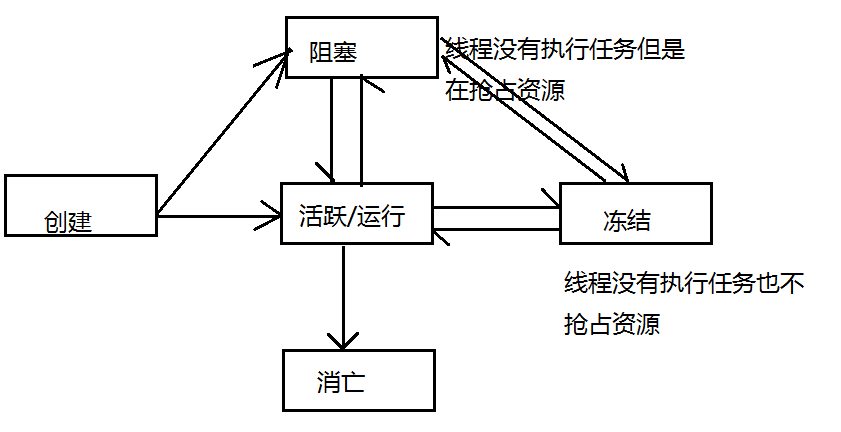
}

}

}

}

## 1.11线程的状态



## 面试题总结：sleep和wait有什么区别？

sleep需要指定一个睡眠时间，到点自然醒，释放执行权，不释放锁，被设计在了Thread上，是一个静态方法。

wait可以指定等待时间，也可以不指定，需要被唤醒，释放执行权，释放锁，被设计在了Object上，是一个普通方法

## 1.12守护线程

当线程被设置为守护线程的时候，其他的线程就默认是被守护状态。一旦被守护的线程结束，守护线程无论完成与否都会随时结束

## 1.13线程的优先级

线程一共有1-10这10个优先级。理论上，数字越大表示这个线程优先级越高，抢占到资源的几率就越大。实际上，相邻的两个优先级的差异性不明显，当优先级差到5个单位以上的时候，才相对有一点明显

设计模式---在软件开发过程中常见的一些方式

## 1.14单例模式

在全局过程中只存在一个唯一实例。

需要把构造函数私有化，提供一个本类的静态变量，并且通过对应的静态方法来获取这个唯一的静态变量

饿汉式不存在多线程并发安全问题，可能会导致内存的浪费

懒汉式相对节省内存，存在线程安全问题