# Day24 多线程

## 线程的基础

### 常识问题

#### 24.01\_多线程(多线程的引入)

**1.什么是线程**

**线程是程序执行的一条路径, 一个进程中可以包含多条线程**

**多线程并发执行可以提高程序的效率, 可以同时完成多项工作**

2.多线程的应用场景

红蜘蛛同时共享屏幕给多个电脑

迅雷开启多条线程一起下载

QQ同时和多个人一起视频

服务器同时处理多个客户端请求

#### 24.02\_多线程(多线程并行和并发的区别)

**并行就是两个任务同时运行，就是甲任务进行的同时，乙任务也在进行。(需要多核CPU,就是多个服务器,一个cpu就是一个服务器)**

**并发是指两个任务都请求运行，而处理器只能按受一个任务，就把这两个任务安排轮流进行，由于时间间隔较短，使人感觉两个任务都在运行。**

比如我跟两个网友聊天，左手操作一个电脑跟甲聊，同时右手用另一台电脑跟乙聊天，这就叫并行。如果用一台电脑我先给甲发个消息，然后立刻再给乙发消息，然后再跟甲聊，再跟乙聊。这就叫并发。

**理解:一段代码放到内存中,而内存是共享的,不管是那个CPU来的线程,甚至是多线程来自不同的CPU也是只能有先后执行的.**

通过多线程实现并发/并行：

1.**Thread类定义实现了多线程，通过多线程可以实现并发或者并行**。

2.在**CPU比较繁忙资源不足**的时候，操作系统只为一个含有多线程的进程分配仅有的CPU资源，这些**线程就会自己争夺时间片。这就是多线程实现并发**，线程之间争夺CPU资源获得执行机会。

3.在**CPU资源较足**的时候，一个进程内的多线程，可以被分配到不同的CPU资源，这就是多线程实现并行。

4.至于**多线程实现的究竟是并发还是并行？前面两点提到的说明这是都有可能的**。因为**多线程可能被分配到一个CPU内核执行，也可能被分配到不同的CPU资源中执行**。**然而这个过程是操作系统干的，不关我的事**。所以我也不确定最后究竟是并发还是并行。

5.**不管是并发还是并行，都提高了对CPU资源的利用率，最大限度的利用了CPU资源**。

#### 24.03\_多线程(Java程序运行原理和JVM的启动是多线程的吗)

A:Java程序运行原理

Java命令会启动java虚拟机，启动JVM，等于启动了一个应用程序，也就是启动了一个进程。该进程会自动启动一个 “主线程” ，然后主线程去调用某个类的 main 方法。

B:JVM的启动是多线程的吗

**JVM启动至少启动了垃圾回收线程和主线程，所以是多线程的。**

**垃圾回收的启动:System.gc();**

**总结:单核中多线程是为了不阻碍主线程产生ANR问题.但是其他的情境下单线程可能要比多线程快因为创建和切换都需要时间,多核中那就更简单了有多个cpu执行多个线程执行多条路效率更多了.多线程下载开启了多条路并且抢占了服务器的带宽.**

#### 24.04 高并发

是指短时间内大量的访问和请求，例如双十一。想系统能够适应高并发状态，则需要全面优化优化，包括，硬件、网络、系统架构、开发语言的选取、数据结构的运用、算法优化、数据库优化……而多线程只是其中解决方法之一。

多线程在高并发问题中起到的作用是使计算机资源在每一时刻能达到最大利用率，不至于浪费计算机资源使其闲置。

### 两大方式

#### 重点 24.04\_多线程(多线程程序实现的方式1,创建子类复写run方法)(掌握)

**线程是程序中的执行线程,Java虚拟机是允许应用程序并发的运行多个执行线程.**

1.继承Thread **涉及到多线程的时候应该将公共的部分提取出来以便复用**

第一步：定义类继承Thread

第二步：重写run方法

第三步：把新线程要做的事写在run方法中

第四步：创建线程对象

第五步：开启新线程, 内部会自动执行run方法

---------------------------------------------------------------------------------------------------------------------------

public class Demo2\_Thread {

public static void main(String[] args) {

MyThread mt = new MyThread(); //4,创建自定义类的对象

mt.start(); //5,开启线程

for(int i = 0; i < 3000; i++) {

System.out.println("bb");

}

}

}

class MyThread extends Thread { //1,定义类继承Thread

public void run() { //2,重写run方法

for(int i = 0; i < 3000; i++) { //3,将要执行的代码,写在run方法中

System.out.println("aaaaaaaaaaaaaaaaaaaaaaaaaaaa");

}

}

}

**注意：**

1，上边的打印肯定是先打印“bb”，因为在开启线程的一刹那间CPU分配给了主方法线程。

2,主线程执行完毕其他线程还会继续执行,**都是平行的关系**.只是一个先后的关系.

3,线程不能重复开启,只能一次.多次会报错.每次new出来的线程是可以的.

4,继承之后不是必须重写run()方法的,不写就是null,什么也没有,输出为空.

---------------------------------------------------------------------------------------------------------------------------

#### 重点 24.05\_多线程(多线程程序实现的方式2,更加的面向对象)(掌握)

2.实现Runnable

定义类实现Runnable接口

实现run方法

把新线程要做的事写在run方法中

创建自定义的Runnable的子类对象

创建Thread对象, 传入Runnable

调用start()开启新线程, 内部会自动调用Runnable的run()方法

---------------------------------------------------------------------------------------------------------------------------

public class Demo3\_Runnable {

/\*\*

@param args

\*/

public static void main(String[] args) {

MyRunnable mr = new MyRunnable(); //4,创建自定义类对象

Thread t = new Thread(mr); //5,将其当作参数传递给Thread的构造函数

t.start(); //6,开启线程

for(int i = 0; i < 3000; i++) {

System.out.println("bb");

}

}

}

class MyRunnable implements Runnable { //1,自定义类实现Runnable接口

@Override

public void run() { //2,重写run方法

for(int i = 0; i < 3000; i++) { //3,将要执行的代码,写在run方法中

System.out.println("aaaaaaaaaaaaaaaaaaaaaaaaaaaa");

}

}

}

**总结:若是两个都在一个Thread中执行,肯定是复写的执行了,编译看父类执行看子类.**

#### ----24.06\_多线程(实现Runnable的原理)(了解)

**底层是有判断的,thread中的start调用了tun方法的**

查看源码

1,看Thread类的构造函数,传递了Runnable接口的引用

2,通过init()方法找到传递的target给成员变量的target赋值

3,查看run方法,发现run方法中有判断,如果target不为null就会调用Runnable接口子类对象的run方法

#### 面试 24.07\_多线程(两种方式的区别)(掌握)

**面试正确能说出源码中的区别**

查看源码的区别:

a.继承Thread : 由于子类重写了Thread类的run(), 当调用start()时, 直接找子类的run()方法(人家就是这么的规定的)

b.实现Runnable : 构造函数中传入了Runnable的引用, 成员变量记住了它, start()调用run()方法时内部判断成员变量Runnable的引用是否为空, 不为空编译时看的是Runnable的run(),运行时执行的是子类的run()方法

**继承Thread(直接性很强,主要推荐方式,两种方式还是的看具体的需求的)**

**好处是:可以直接使用Thread类中的方法,代码简单**

**弊端是:如果已经有了父类,就不能用这种方法**

**实现Runnable接口(可以认为是继承thread的一种补充,扩展性好)**

**好处是:即使自己定义的线程类有了父类也没关系,因为有了父类也可以实现接口,而且接口是可以多实现的**

**弊端是:不能直接使用Thread中的方法需要先获取到线程对象后,才能得到Thread的方法,代码复杂**

#### ----(匿名内部类实现)24.08\_多线程(线程的两种方式)

**继承Thread类(匿名内部类,匿名子类对象)**

---------------------------------------------------------------------------------------------------------------------------

new Thread() { //1,new 类(){}继承这个类

public void run() { //2,重写run方法

for(int i = 0; i < 3000; i++) { //3,将要执行的代码,写在run方法中

System.out.println("aaaaa");

}

}

}.start();

**实现Runnable接口**

new Thread(new Runnable(){ //1,new 接口(){}实现这个接口

public void run() { //2,重写run方法

for(int i = 0; i < 3000; i++) { //3,将要执行的代码,写在run方法中

System.out.println("bb");

}

}

}).start();

### 几个简单方法

#### 24.09\_多线程

  3、**通过Thread.currentThread().setName()方法**

    2、**通过Thread.currentThread().getName()方法**

#### 24.10\_多线程(获取当前线程的对象)(掌握)

**Thread.currentThread(), 主线程也可以获取**

#### 24.11\_多线程(休眠线程)(掌握),Thread.sleep()方法即可

Thread.sleep(毫秒,纳秒)

## 线程的几大特性

### 24.12\_多线程(守护线程)(掌握)

**setDaemon(), 设置一个线程为守护线程, 该线程不会单独执行, 当其他非守护线程都执行结束后, 自动退出,可能会用一份缓冲的时间就是延迟**

--------------------------------------------------------------------------------------------------------------------------

Thread t1 = new Thread() {

public void run() {

for(int i = 0; i < 50; i++) {

System.out.println(getName() + "...aaaaaaaaaaaaaaaaaaaaaa");

try {

Thread.sleep(10);

} catch (InterruptedException e) {

e.printStackTrace();

}

}

}

};

Thread t2 = new Thread() {

public void run() {

for(int i = 0; i < 5; i++) {

System.out.println(getName() + "...bb");

try {

Thread.sleep(10);

} catch (InterruptedException e) {

e.printStackTrace();

}

}

}

};

**t1.setDaemon(true); //将t1设置为守护线程,肯定是先设置而后启动的,顺序不能搞反的.**

t1.start();

t2.start();

**应用:聊天的界面是守护而qq总界面是非守护的,qq解密啊你关掉随之聊天界面也会挂掉的.**

### 24.13\_多线程(加入线程)(掌握)

**join(), 当前线程暂停, 等待指定的线程执行结束后, 当前线程再继续**

**join(int), 可以等待指定的毫秒之后继续**

---------------------------------------------------------------------------------------------------------------------------

---------------------------------------------------------------------------------------------------------------------------

final Thread t1 = new Thread() {

public void run() {

for(int i = 0; i < 50; i++) {

System.out.println(getName() + "...aaaaaaaaaaaaaaaaaaaaaa");

try {

Thread.sleep(10);

} catch (InterruptedException e) {

e.printStackTrace();

}

}

}

};

Thread t2 = new Thread() {

public void run() {

for(int i = 0; i < 50; i++) {

if(i == 2) {

try {

//t1.join(); //插队,加入

**t1.join(30); //加入,有固定的时间,过了固定时间,继续交替执行**

Thread.sleep(10);

} catch (InterruptedException e) {

e.printStackTrace();

}

}

System.out.println(getName() + "...bb");

}

}

};

t1.start();

t2.start();

### 24.14\_多线程(礼让线程)(了解)

yield让出cpu

### 24.15\_多线程(设置线程的优先级)(了解)

setPriority()设置线程的优先级

## 线程的同步,安全,死锁问题

### 面试重点:24.16\_多线程(同步代码块)(掌握)

**1.什么情况下需要同步**

**当多线程并发, 有多段代码同时执行时, 我们希望某一段代码执行的过程中CPU不要切换到其他线程工作(多核中也就是说不能多个CPU每人都带领着一个线程来执行同一段代码). 这时就需要同步.**

**如果两段代码是同步的, 那么同一时间只能执行一段, 在一段代码没执行结束之前, 不会执行另外一段代码.**

**就是不让CPU的使用权让给别人,你一进来就得执行完毕,那样我的数据才不会出错**

2.同步代码块

使用synchronized关键字加上一个锁对象来定义一段代码, 这就叫同步代码块

多个同步代码块如果使用相同的锁对象, 那么他们就是同步的

**锁对象:**

**同步代码块**:不能是匿名对象,只能是同一个对象(可以另外单建立一个对象或者是本类的对象),就是同一个地址值.当然可以是本类的字节码文件即.class文件(反射讲,他就是一个对象)也可以是this.(当然在静态的时候是不能用this的,而.class一直都是可以用的)

**同步方法**:静态最好是本类的.class文件,非静态可以是this.因为方法内部是隐藏了this的,但是静态并没有.(总结,其实方法上锁无所谓静态与非静态,都是必须一次性的执行完事的,只是锁不同而已)

代码块讲解

package com.example.will.thread\_knowledge;  
  
public class Code {  
  
 /\*\*  
 \* 普通代码块：在方法或语句中出现的{}就称为普通代码块。  
 \* 普通代码块和一般的语句执行顺序由他们在代码中出现的次序决定--“先出现先执行”  
 \*/  
 public void init() {  
 {  
 System.out.println("普通代码块中的普通代码块");  
 }  
 System.out.println("普通代码块");  
 }  
  
 /\*\*  
 \* 构造块：直接在类中定义且没有加static关键字的代码块称为{}构造代码块。  
 \* 构造代码块在创建对象时被调用，每次创建对象都会被调用，并且构造代码块的执行次序优先于类构造函数  
 \*/ {  
 System.out.println("Code的构造块");  
 }  
  
 /\*\*  
 \* 静态代码块:在java中使用static关键字声明的代码块。静态块用于初始化类，为类的属性初始化。每个静态代码块只会执行一次。  
 \* 由于JVM在加载类时会执行静态代码块，所以静态代码块先于主方法执行。如果类中包含多个静态代码块，那么将按照"先定义的代码先执行，后定义的代码后执行"。  
 \* 注意：1 静态代码块不能存在于任何方法体内。2 静态代码块不能直接访问静态实例变量和实例方法，需要通过类的实例对象来访问。  
 \*/  
 static {  
 System.out.println("Code的静态代码块");  
 }  
  
 public Code() {  
 System.out.println("Code的构造方法");  
 }  
}

---------------------------------------------------------------------------------

### 24.17\_多线程(同步方法)(掌握)

使用synchronized关键字修饰一个方法, 该方法中所有的代码都是同步的

---------------------------------------------------------------------------------------------------------------------------

**class** Printer2 {

Demo d = **new** Demo();

**//非静态的同步方法的锁对象是神马?**

**//答:非静态的同步方法的锁对象是this,就是指的是所在类的对象,而并非是字节码对象.**

**//静态的同步方法的锁对象是什么?**

**//是该类的字节码对象,并不是该类对象**

**//底下验证的是静态的方法,若是非静态直接将两把锁换成this即可,代表的是大类Printer2的对象**

**public** **static** **synchronized** **void** print1() { //同步方法只需要在方法上加synchronized关键字即可

System.***out***.print("黑");

System.***out***.print("马");

System.***out***.print("程");

System.***out***.print("序");

System.***out***.print("员");

System.***out***.print("\r\n");

}

// static类加载的时候创建,那么就不能使用this,super方法了.

**public** **static** **void** print2() {

//synchronized(new Demo()) { //锁对象不能用匿名对象,因为匿名对象不是同一个对象

**synchronized**(Printer2.**class**) {

System.***out***.print("传");

System.***out***.print("智");

System.***out***.print("播");

System.***out***.print("客");

System.***out***.print("\r\n");

}

----------------------------------------------------------------

### 24.18\_多线程(线程安全问题)(掌握)

**多线程并发操作同一数据(static)时, 就有可能出现线程安全问题**

**使用同步技术可以解决这种问题, 把操作数据的代码进行同步, 不要多个线程一起操作**

---------------------------------------------------------------------------------------------------------------------------

public class Demo2\_Synchronized {

/\*\*

@param args

需求:铁路售票,一共100张,通过四个窗口卖完.

\*/

public static void main(String[] args) {

TicketsSeller t1 = new TicketsSeller();

TicketsSeller t2 = new TicketsSeller();

TicketsSeller t3 = new TicketsSeller();

TicketsSeller t4 = new TicketsSeller();

t1.setName("窗口1");

t2.setName("窗口2");

t3.setName("窗口3");

t4.setName("窗口4");

t1.start();

t2.start();

t3.start();

t4.start();

}

}

class TicketsSeller extends Thread {

**private static int tickets = 100; //不是静态的话,四个线程每人都会有一个成员变量,下边的锁对象也是一样,必须是四个对象所公共的,最直接的锁对象是Ticks.class.尽量使用字节码文件作为锁对象.**

**static Object obj = new Object();**

public TicketsSeller() {

super();

}

public TicketsSeller(String name) {

super(name);

}

public void run() {

while(true) {

**synchronized(obj) {**

if(tickets <= 0)

break;

try {

Thread.sleep(10);//线程1睡,线程2睡,线程3睡,线程4睡,最后按照睡觉的先后顺序醒来执行下边的数据.模拟其他操作的.

} catch (InterruptedException e) {

e.printStackTrace();

}

System.out.println(getName() + "...这是第" + tickets-- + "号票");

}

}

}

}

### ----24.19\_多线程(火车站卖票的例子用实现Runnable接口)(掌握)

MyTicket mt = **new** MyTicket(); **//创建一个对象**

**new** Thread(mt).start();

**new** Thread(mt).start();

**new** Thread(mt).start();

**new** Thread(mt).start();

/\*Thread t1 = new Thread(mt); **//多次启动一个线程是非法的**

t1.start();

t1.start();

t1.start();

t1.start();\*/

class Ticks2 implements Runnable {

private static int ticks = 100;

public void run() {

while (true) {

synchronized (Ticks2.class) {

if (ticks <= 0) {

break;

}

try {

Thread.sleep(100);

} catch (InterruptedException e) {

e.printStackTrace();

}

System.out.println(Thread.currentThread().getName() + "正在卖" + ticks-- + "号票");

}

}

}

}

### 24.20\_多线程(死锁)

**多线程同步的时候, 如果同步代码嵌套, 使用相同锁, 就有可能出现死锁**

**尽量不要嵌套使用**

---------------------------------------------------------------------------------------------------------------------------

private static String s1 = "筷子左";

private static String s2 = "筷子右";

public static void main(String[] args) {

new Thread() {

public void run() {

while(true) {

synchronized(s1) {

System.out.println(getName() + "...拿到" + s1 + "等待" + s2);

synchronized(s2) {

System.out.println(getName() + "...拿到" + s2 + "开吃");

}

}

}

}

}.start();

new Thread() {

public void run() {

while(true) {

synchronized(s2) {

System.out.println(getName() + "...拿到" + s2 + "等待" + s1);

synchronized(s1) {

System.out.println(getName() + "...拿到" + s1 + "开吃");

}

}

}

}

}.start();

}

## 掌握:24.21\_多线程(以前的线**程**安全的类回顾)(掌握)

A:回顾以前说过的线程安全问题

看源码：Vector,StringBuffer,Hashtable,Collections.synchroinzed(xxx)

Vector是线程安全的,ArrayList是线程不安全的(是可以使用synchronize方法使之安全)

StringBuffer是线程安全的,StringBuilder是线程不安全的

Hashtable是线程安全的,HashMap是线程不安全.

集合中是有工具类将不安全集合转变为安全集合的.

**看不安全看的就是添加方法是否有synchroinzed.**

24.22\_多线程(总结)

# Day25 多线程

## 重点:25.01\_多线程(单例设计模式)(懒加载应该加锁)

单例设计模式：保证类在内存中只有一个对象。

如何保证类在内存中只有一个对象呢？

**(1)控制类的创建，不让其他类来创建本类的对象。private构造方法**

**(2)在本类中定义一个本类的对象。Singleton s;**

**(3)提供公共的访问方式。 public static Singleton getInstance(){return s}**

单例写法两种：

(1)饿汉式 开发用这种方式。

//饿汉式

class Singleton {

//1,私有构造函数

private Singleton(){}

//2,创建本类对象

private static Singleton s = new Singleton();

//3,对外提供公共的访问方法

public static Singleton getInstance() { **//引用的时候都是static,私有了**

return s;

}

public static void print() {

System.out.println("11111111111");

}

}

(2)懒汉式 面试写这种方式。多线程的问题

//懒汉式,单例的延迟加载模式

class Singleton {

//1,私有构造函数

private Singleton(){}

//2,声明一个本类的引用

private static Singleton s;

//3,对外提供公共的访问方法

public static Singleton getInstance() {

synchronized (Singletonn.class) {

if(s == null)

**//线程1,线程2,多线程的访问数据是有问题.可以将之sync**

**hroized锁起来,同步代码块或者是方法的锁,都是判断为空的,所以都要创建对象**

s = new Singleton();

}

return s;

}

public static void print() {

System.out.println("11111111111");

}

}

(3)第三种格式

class Singleton {

private Singleton() {}

public static final Singleton s = new Singleton();//final是最终的意思,被final修饰的变量不可以被更改

}

**总结:懒汉式和饿汉式的区别.**

**1,懒汉式是在空间换时间,当然是时间重要,应该空间换时间用饿汉式.**

**2,懒汉式会存在多线程的访问安全问题.**

## 25.02\_多线程(Runtime类)

Runtime类是一个单例类,就是封闭了一个对象,验证上边的单例设计模式

Runtime r = Runtime.getRuntime();

//r.exec("shutdown -s -t 300"); //300秒后关机

r.exec("shutdown -a"); //取消关机

## 重点:25.03\_多线程(Timer)(掌握)

Timer类:计时器,指定时间来做指定的事情.

public class Demo5\_Timer {

/\*\*

@param args

计时器

@throws InterruptedException

\*/

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

**Timer t = new Timer(); //指定的时间做指定的事情.Task直接在这里写就行**

**t.schedule(new MyTimerTask(), new Date(114,9,15,10,54,20),3000); //看API,有特殊的规定年月日,指定的时间之后多久,后边的3000是间隔3s进行连环执行.**

while(true) {

System.out.println(new Date());

Thread.sleep(1000);

}

}

}

class MyTimerTask extends TimerTask {

@Override

public void run() {

System.out.println("起床背英语单词");

}

}

## 掌握:25.04\_多线程(两个线程间的通信)

**1.什么时候需要通信**

多个线程并发执行时, 在默认情况下CPU是随机切换线程的

如果我们希望他们有规律的执行, 就可以使用通信, 例如每个线程执行一次打印

**2.怎么通信**

如果希望线程等待, 就调用wait()

如果希望唤醒等待的线程, 就调用notify();

**这两个方法必须在同步代码中执行, 并且使用同步锁对象来调用**

package com.heima.thread2;

public class Demo1\_Notify {

/\*\*

@param args

等待唤醒机制

\*/

public static void main(String[] args) {

final Printer p = new Printer();

new Thread() {

public void run() {

while(true) {

try {

**p.print1(); 将方法进行提取出来之后进行封闭**

} catch (InterruptedException e) {

e.printStackTrace();

}

}

}

}.start();

new Thread() {

public void run() {

while(true) {

try {

**p.print2();**

} catch (InterruptedException e) {

e.printStackTrace();

}

}

}

}.start();

}

}

//等待唤醒机制

class Printer {

private int flag = 1;

public void print1() throws InterruptedException {

synchronized(**this**) {

**//重新开启线程后,子线程并不持有类对象,就是Printer对象,所以应该使用this,用什么锁就用什么notify()**

**//java.lang.IllegalMonitorStateException**

**if(flag != 1) {**

**this.wait(); //当前线程等待**

**}**

System.out.print("黑");

System.out.print("马");

System.out.print("程");

System.out.print("序");

System.out.print("员");

System.out.print("\r\n");

**flag = 2;**

**this.notify(); //随机唤醒单个等待的线程**

}

}

public void print2() throws InterruptedException {

synchronized(this) {

**if(flag != 2) {**

**this.wait();**

**}**

System.out.print("传");

System.out.print("智");

System.out.print("播");

System.out.print("客");

System.out.print("\r\n");

**flag = 1;**

**this.notify();**

}

}

}

## 25.05\_多线程(三个或三个以上间的线程通信)

多个线程通信的问题

notify()方法是随机唤醒一个线程(**其实等待的是等待队列,是有先后顺序的,先进先出)**

**notifyAll()方法是唤醒所有线程**

**JDK5之前无法唤醒指定的一个线程**

**1,在同步代码块中,用哪个对象锁,就用哪个对象调用wait方法**

2,为什么wait方法和notify方法定义在Object这类中?

因为锁对象可以是任意对象,Object是所有的类的基类,所以wait方法和notify方法需要定义在Object这个类中

**3,sleep方法和wait方法的区别?**

a,sleep方法必须传入参数,参数就是时间,时间到了自动醒来

wait方法可以传入参数也可以不传入参数,传入参数就是在参数的时间结束后等待,不传入参数就是直接等待

b,sleep方法在同步函数或同步代码块中,不释放锁,睡着了也抱着锁睡

wait方法在同步函数或者同步代码块中,释放锁

## 掌握:25.06\_多线程(JDK1.5的新特性互斥锁)(掌握)

1.同步

**使用ReentrantLock类的lock()和unlock()方法进行同步,就是建立锁对象**

2.通信

**使用ReentrantLock类的newCondition()方法可以获取Condition对象**

**需要等待的时候使用Condition的await()方法, 唤醒的时候用signal()方法**

**不同的线程使用不同的Condition, 这样就能区分唤醒的时候找哪个线程了**

**主要就是为了替换之前版本中的notify()和wait()的.**

----------------------------------------------------------------------------------------------------

package com.heima.thread2;

import java.util.concurrent.locks.Condition;

import java.util.concurrent.locks.ReentrantLock;

public class Demo3\_ReentrantLock {

/\*\*

\* @param args

\*/

public static void main(String[] args) {

final Printer3 p = new Printer3();

new Thread() {

public void run() {

while(true) {

try {

p.print1();

} catch (InterruptedException e) {

e.printStackTrace();

}

}

}

}.start();

new Thread() {

public void run() {

while(true) {

try {

p.print2();

} catch (InterruptedException e) {

e.printStackTrace();

}

}

}

}.start();

new Thread() {

public void run() {

while(true) {

try {

p.print3();

} catch (InterruptedException e) {

e.printStackTrace();

}

}

}

}.start();

}

}

class Printer3 {

**private ReentrantLock r = new ReentrantLock();**

**private Condition c1 = r.newCondition();**

**private Condition c2 = r.newCondition();**

**private Condition c3 = r.newCondition();**

private int flag = 1;

public void print1() throws InterruptedException {

**r.lock(); //获取锁**

if(flag != 1) {

**c1.await();**

}

System.out.print("黑");

System.out.print("马");

System.out.print("程");

System.out.print("序");

System.out.print("员");

System.out.print("\r\n");

flag = 2;

**//this.notify(); //随机唤醒单个等待的线程**

**c2.signal(); //指定的线程所持有的锁**

**r.unlock(); //释放锁**

}

public void print2() throws InterruptedException {

r.lock();

if(flag != 2) {

c2.await();

}

System.out.print("传");

System.out.print("智");

System.out.print("播");

System.out.print("客");

System.out.print("\r\n");

flag = 3;

//this.notify();

c3.signal();

r.unlock();

}

public void print3() throws InterruptedException {

r.lock();

if(flag != 3) {

c3.await();

}

System.out.print("i");

System.out.print("t");

System.out.print("h");

System.out.print("e");

System.out.print("i");

System.out.print("m");

System.out.print("a");

System.out.print("\r\n");

flag = 1;

c1.signal();

r.unlock();

}

}

----------------------------------------------------------------------------------------------------

## 25.07\_多线程(线程组的概述和使用)(可以成组的管理,更智能的就是线程池)

A:线程组概述

Java中使用ThreadGroup来表示线程组，它可以对一批线程进行分类管理，Java允许程序直接对线程组进行控制。

**默认情况下，所有的线程都属于主线程组。**

public final ThreadGroup getThreadGroup()//通过线程对象获取他所属于的组

public final String getName()//通过线程组对象获取他组的名字

我们也可以给线程设置分组

1,ThreadGroup(String name) 创建线程组对象并给其赋值名字

2,创建线程对象

3,Thread(ThreadGroup?group, Runnable?target, String?name)

4,设置整组的优先级或者守护线程

B:案例演示

线程组的使用,默认是主线程组

-----------------------------------------------------------------------------------------------------------------------

package com.heima.thread2;

public class Demo4\_ThreadGroup {

/\*\*

\* @param args

\* ThreadGroup

\*/

public static void main(String[] args) {

//demo1();

ThreadGroup tg = new ThreadGroup("我是一个新的线程组"); //创建新的线程组

MyRunnable mr = new MyRunnable(); //创建Runnable的子类对象

**Thread t1 = new Thread(tg, mr, "张三"); //将线程t1放在组中**

**Thread t2 = new Thread(tg, mr, "李四"); //将线程t2放在组中**

System.out.println(t1.getThreadGroup().getName()); //获取组名

System.out.println(t2.getThreadGroup().getName());

tg.setDaemon(true);

}

public static void demo1() {

MyRunnable mr = new MyRunnable();

Thread t1 = new Thread(mr, "张三");

Thread t2 = new Thread(mr, "李四");

ThreadGroup tg1 = t1.getThreadGroup();

ThreadGroup tg2 = t2.getThreadGroup();

System.out.println(tg1.getName()); //默认的是主线程

System.out.println(tg2.getName());

}

}

class MyRunnable implements Runnable {

@Override

public void run() {

for(int i = 0; i < 1000; i++) {

System.out.println(Thread.currentThread().getName() + "...." + i);

}

}

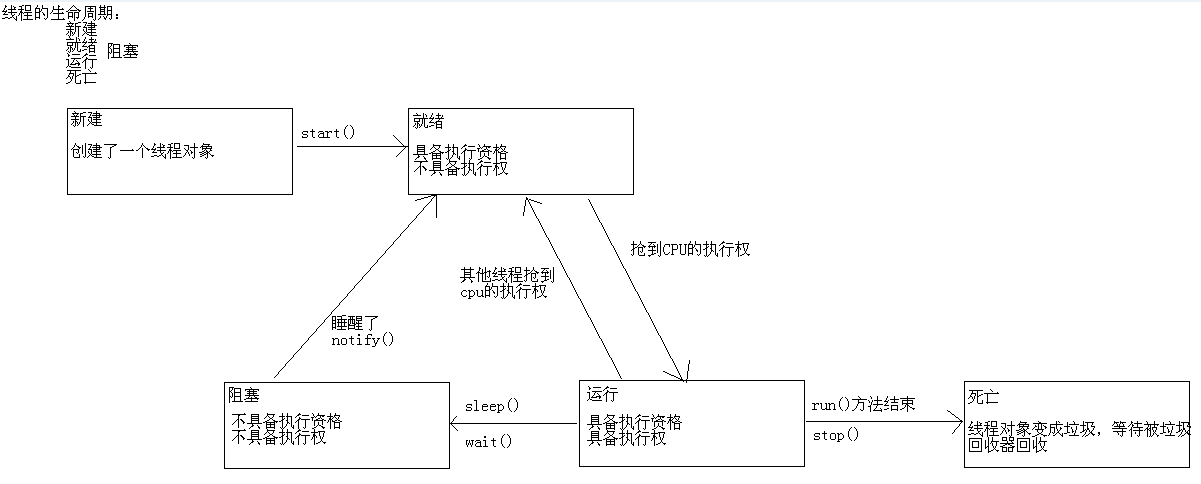
}

----------------------------------------------------------------------------------------------------------------------

## (面试)25.08\_多线程(线程的五种状态)

看图说话

新建,就绪,运行,阻塞,死亡



## 25.09\_多线程(线程池的概述和使用)

A:线程池概述

**程序启动一个新线程成本是比较高的，因为它涉及到要与操作系统进行交互。而使用线程池可以很好的提高性能，尤其是当程序中要创建大量生存期很短的线程时，更应该考虑使用线程池。线程池里的每一个线程代码结束后，并不会死亡，而是再次回到线程池中成为空闲状态，等待下一个对象来使用。在JDK5之前，我们必须手动实现自己的线程池，从JDK5开始，Java内置支持线程池**

B:内置线程池的使用概述

JDK5新增了一个Executors工厂类来产生线程池，有如下几个方法

public static ExecutorService **newFixedThreadPool(int nThreads)**

public static ExecutorService **newSingleThreadExecutor()**

这些方法的返回值是ExecutorService对象，该对象表示一个线程池，可以执行Runnable对象或者Callable对象代表的线程。它提供了如下方法

Future<?> submit(Runnable task)

<T> Future<T> submit(Callable<T> task)

使用步骤：

创建线程池对象

创建Runnable实例

提交Runnable实例

关闭线程池

C:案例演示

提交的是Runnable

---------------------------------------------------------------------------------------------------------------

public static void main(String[] args) {

**ExecutorService pool = Executors.newFixedThreadPool(2);//创建线程池**

**pool.submit(new MyRunnable()); //将线程放进池子里并执行**

**pool.submit(new MyRunnable());**

**pool.shutdown(); //关闭线程池**

}

class MyRunnable implements Runnable {

@Override

public void run() {

for(int i = 0; i < 1000; i++) {

System.out.println(Thread.currentThread().getName() + "...." + i);

}

}

}

------------------------------------------------------------------------------------------------------------------

## 知道:25.10\_多线程(多线程程序实现的方式3)(Callable有返回值并且可以抛异常)

提交的是Callable

// 创建线程池对象

ExecutorService pool = Executors.newFixedThreadPool(2);

// 可以执行Runnable对象或者Callable对象代表的线程

Future<Integer> f1 = pool.submit(new MyCallable(100));

Future<Integer> f2 = pool.submit(new MyCallable(200));

// V get()

Integer i1 = f1.get();

Integer i2 = f2.get();

System.out.println(i1);

System.out.println(i2);

// 结束

pool.shutdown();

public class MyCallable implements Callable<Integer> {

private int number;

public MyCallable(int number) {

this.number = number;

}

@Override

public Integer call() throws Exception {

int sum = 0;

for (int x = 1; x <= number; x++) {

sum += x;

}

return sum;

}

}

多线程程序实现的方式3的好处和弊端

好处：

可以有返回值

可以抛出异常

弊端：

代码比较复杂，所以一般不用

## 重点:25.11\_设计模式(简单工厂模式概述和使用)

A:简单工厂模式概述

**又叫静态工厂方法模式，它定义一个具体的工厂类负责创建一些类的实例,就和线程池一样的.**

B:优点

客户端不需要在负责对象的创建，从而明确了各个类的职责

C:缺点

**这个静态工厂类负责所有对象的创建，如果有新的对象增加，或者某些对象的创建方式不同，就需要不断的修改工厂类，不利于后期的维护**

D:案例演示

动物抽象类：public abstract Animal { public abstract void eat(); }

具体狗类：public class Dog extends Animal {}

具体猫类：public class Cat extends Animal {}

动物工厂:

开始，在测试类中每个具体的内容自己创建对象，但是，创建对象的工作如果比较麻烦，就需要有人专门做这个事情，所以就知道了一个专门的类来创建对象。

public class AnimalFactory {

private AnimalFactory(){}

//public static Dog createDog() {return new Dog();}

//public static Cat createCat() {return new Cat();}

//改进

public **static** Animal **createAnimal**(String animalName) {

If(animalName == null){

return null;

}

if(“dog”.equals(animalName)) {

return new Dog();

}

else if(“cat”.equals(animale)) {

return new Cat();

}else {

return null;

}

}

}

## 知道:25.12\_设计模式(工厂方法模式的概述和使用)

**总结:其实简单工厂用的多,就是因为代码简单.**

**简易工厂模式与工厂设计模式区别:**

**动物抽象类抽象方法/普通类继承抽象类重写抽象方法这是共同点,工厂类与测试类/工厂接口方法,每一个动物工厂实现工厂的接口实现方法.**

A:工厂方法模式概述

工厂方法模式中抽象工厂类负责定义创建对象的接口，具体对象的创建工作由继承抽象工厂的具体类实现。

B:优点

客户端不需要在负责对象的创建，从而明确了各个类的职责，如果有新的对象增加，只需要增加一个具体的类和具体的工厂类即可，不影响已有的代码，后期维护容易，增强了系统的扩展性

C:缺点

需要额外的编写代码，增加了工作量

D:案例演示

动物抽象类：public abstract Animal { public abstract void eat(); }

工厂接口：public interface Factory {public abstract Animal createAnimal();}

具体狗类：public class Dog extends Animal {}

具体猫类：public class Cat extends Animal {}

开始，在测试类中每个具体的内容自己创建对象，但是，创建对象的工作如果比较麻烦，就需要有人专门做这个事情，所以就知道了一个专门的类来创建对象。发现每次修改代码太麻烦，用工厂方法改进，针对每一个具体的实现提供一个具体工厂。

狗工厂：public class DogFactory implements Factory {

public Animal createAnimal() {…}

}

猫工厂：public class CatFactory implements Factory {

public Animal createAnimal() {…}

}

## 25.13\_GUI(如何创建一个窗口并显示)

Graphical User Interface(图形用户接口)。

Frame f = new Frame(“my window”);

f.setLayout(new FlowLayout());//设置布局管理器

f.setSize(500,400);//设置窗体大小

f.setLocation(300,200);//设置窗体出现在屏幕的位置

f.setIconImage(Toolkit.getDefaultToolkit().createImage("qq.png"));

f.setVisible(true);

## 25.14\_GUI(布局管理器)

FlowLayout（流式布局管理器）

从左到右的顺序排列。

Panel默认的布局管理器。

BorderLayout（边界布局管理器）

东，南，西，北，中

Frame默认的布局管理器。

GridLayout（网格布局管理器）

规则的矩阵

CardLayout（卡片布局管理器）

选项卡

GridBagLayout（网格包布局管理器）

非规则的矩阵

## 25.15\_GUI(窗体监听)

Frame f = new Frame("我的窗体");

//事件源是窗体,把监听器注册到事件源上

//事件对象传递给监听器

f.addWindowListener(new WindowAdapter() {

public void windowClosing(WindowEvent e) {

//退出虚拟机,关闭窗口

System.exit(0);

}

});

## 25.16\_GUI(鼠标监听)

## 25.17\_GUI(键盘监听和键盘事件)

## 重点:25.18\_GUI(动作监听)

就是要掌握那个Listen的方式.

## 重点:25.19\_设计模式(适配器设计模式)(掌握)

a.什么是适配器

在使用监听器的时候, 需要定义一个类事件监听器接口.

通常接口中有多个方法, 而程序中不一定所有的都用到, 但又必须重写, 这很繁琐.

适配器简化了这些操作, 我们定义监听器时只要继承适配器, 然后重写需要的方法即可.

b.适配器原理

**适配器就是一个类, 实现了监听器接口, 所有抽象方法都重写了, 但是方法全是空的.**

适配器类需要定义成抽象的,因为创建该类对象,调用空方法是没有意义的

目的就是为了简化程序员的操作, 定义监听器时继承适配器, 只重写需要的方法就可以了.

## 25.20\_GUI(需要知道的)

事件处理

事件: 用户的一个操作

事件源: 被操作的组件

监听器: 一个自定义类的对象, 实现了监听器接口, 包含事件处理方法,把监听器添加在事件源上, 当事件发生的时候虚拟机就会自动调用监听器中的事件处理方法

## 25.21\_day25总结

把今天的知识点总结一遍。