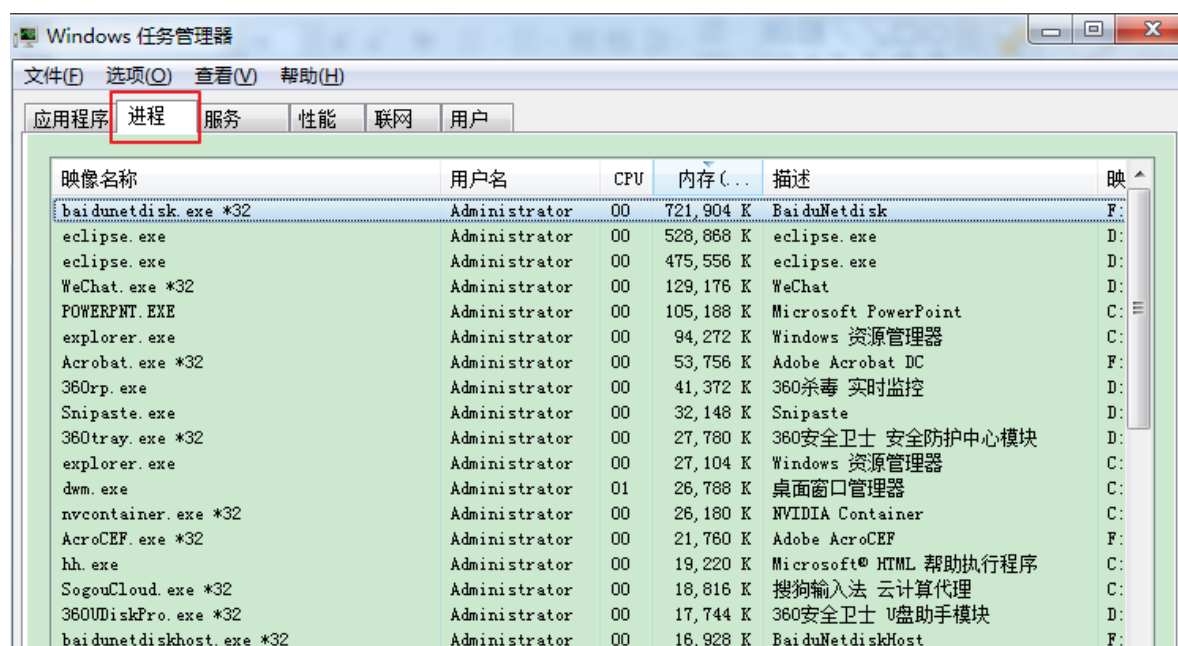


一、基本概念：程序、进程、线程

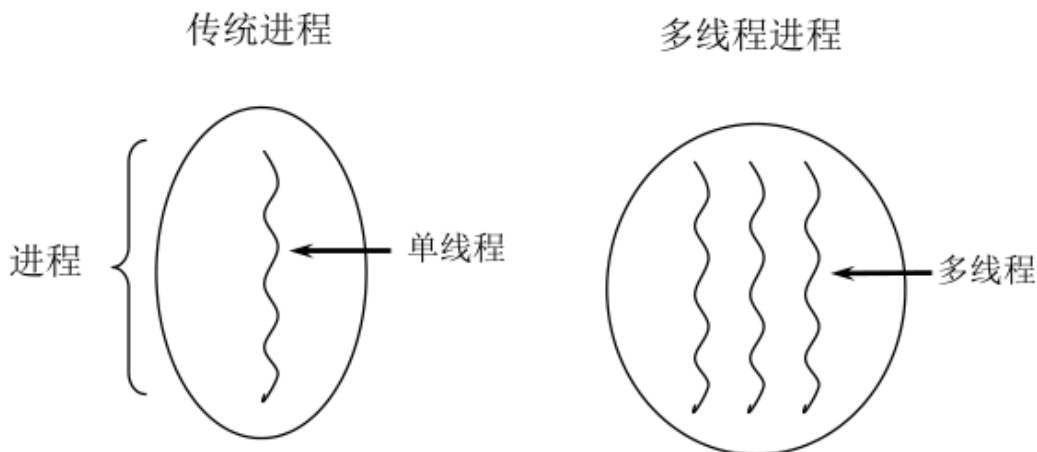
1、程序、进程、线程的概念

- **程序(program)**是为完成特定任务、用某种语言编写的一组指令的集合。即指一段静态的代码，静态对象。
- **进程(process)**是程序的一次执行过程，或是正在运行的一个程序。是一个动态的过程：有它自身的产生、存在和消亡的过程。——生命周期
 - 如：运行中的QQ，运行中的MP3播放器
 - 程序是静态的，进程是动态的
 - 进程作为资源分配的单位，系统在运行时会为每个进程分配不同的内存区域
- **线程(thread)**，进程可进一步细化为线程，是一个程序内部的一条执行路径。
 - 若一个进程同一时间并行执行多个线程，就是支持多线程的
 - 线程作为调度和执行的单位，每个线程拥有独立的运行栈和程序计数器(pc)，线程切换的开销小
 - 一个进程中的多个线程共享相同的内存单元/内存地址空间→它们从同一堆中分配对象，可以访问相同的变量和对象。这就使得线程间通信更简便、高效。但多个线程操作共享的系统资源可能就会带来安全的隐患。



映像名称	用户名	CPU	内存	描述
baidunetdisk.exe *32	Administrator	00	721,904 K	BaiduNetdisk
eclipse.exe	Administrator	00	528,868 K	eclipse.exe
eclipse.exe	Administrator	00	475,556 K	eclipse.exe
WeChat.exe *32	Administrator	00	129,176 K	WeChat
POWERPNT.EXE	Administrator	00	105,188 K	Microsoft PowerPoint
explorer.exe	Administrator	00	94,272 K	Windows 资源管理器
Acrobat.exe *32	Administrator	00	53,756 K	Adobe Acrobat DC
360rp.exe	Administrator	00	41,372 K	360杀毒 实时监控
Snipaste.exe	Administrator	00	32,148 K	Snipaste
360tray.exe *32	Administrator	00	27,780 K	360安全卫士 安全防护中心模块
explorer.exe	Administrator	00	27,104 K	Windows 资源管理器
dwm.exe	Administrator	01	26,788 K	桌面窗口管理器
nvcontainer.exe *32	Administrator	00	26,180 K	NVIDIA Container
AcroCEF.exe *32	Administrator	00	21,760 K	Adobe AcroCEF
hh.exe	Administrator	00	19,220 K	Microsoft® HTML 帮助执行程序
SogouCloud.exe *32	Administrator	00	18,816 K	搜狗输入法 云计算代理
360UDiskPro.exe *32	Administrator	00	17,744 K	360安全卫士 U盘助手模块
baidunetdiskhost.exe *32	Administrator	00	16,928 K	BaiduNetdiskHost

进程与线程



2、单核CPU和多核CPU的理解

- 单核CPU，其实是一种假的多线程，因为在一个时间单元内，也只能执行一个线程的任务。例如：虽然有多车道，但是收费站只有一个工作人员在收费，只有收了费才能通过，那么CPU就好比收费人员。如果有某个人不想交钱，那么收费人员可以把他“挂起”（晾着他，等他想通了，准备好了钱，再去收费）。但是因为CPU时间单元特别短，因此感觉不出来。
- 如果是多核的话，才能更好的发挥多线程的效率。（现在的服务器都是多核的）
- 一个Java应用程序java.exe，其实至少有三个线程：main()主线程，gc()垃圾回收线程，异常处理线程。当然如果发生异常，会影响主线程。

3、并行与并发

- 并行：多个CPU同时执行多个任务。比如：多个人同时做不同的事。
- 并发：一个CPU(采用时间片)同时执行多个任务。比如：秒杀、多个人做同一件事。

4、使用多线程的优点

背景：以单核CPU为例，只使用单个线程先后完成多个任务（调用多个方法），肯定比用多个线程来完成用的时间更短，为何仍需多线程呢？

多线程程序的优点：

1. 提高应用程序的响应。对图形化界面更有意义，可增强用户体验。
2. 提高计算机系统CPU的利用率
3. 改善程序结构。将既长又复杂的进程分为多个线程，独立运行，利于理解和修改

5、何时需要多线程

- 程序需要同时执行两个或多个任务。
- 程序需要实现一些需要等待的任务时，如用户输入、文件读写操作、网络操作、搜索等。
- 需要一些后台运行的程序时。

二、线程的创建和使用(2种)

1、线程的创建和启动

- Java语言的JVM允许程序运行多个线程，它通过**java.lang.Thread**类来体现。
- Thread类的特性
 - 每个线程都是通过某个特定Thread对象的run()方法来完成操作的，经常把run()方法的主体称为**线程体**
 - 通过该Thread对象的start()方法来启动这个线程，而非直接调用run()

2、Thread类构造器

- 构造器
 - **Thread():** 创建新的Thread对象
 - **Thread(String threadname):** 创建线程并指定线程实例名
 - **Thread(Runnable target):** 指定创建线程的目标对象，它实现了Runnable接口中的run方法
 - **Thread(Runnable target, String name):** 创建新的Thread对象

3、API 中创建线程的两种方式

- JDK1.5之前创建新执行线程有两种方法：
 - 继承Thread类的方式
 - 实现Runnable接口的方式

(1) 方式一：继承Thread类

- 1) 定义子类继承Thread类。
- 2) 子类中重写Thread类中的run方法。
- 3) 创建Thread子类对象，即创建了线程对象。
- 4) 调用线程对象start方法：启动线程，调用run方法。

● 注意点：

1. 如果自己手动调用run()方法，那么就只是普通方法，没有启动多线程模式。
2. run()方法由JVM调用，什么时候调用，执行的过程控制都有操作系统的CPU调度决定。
3. 想要启动多线程，必须调用start方法。
4. 一个线程对象只能调用一次start()方法启动，如果重复调用了，则将抛出以上的异常“IllegalThreadStateException”。

```
package atguigu.java;

/**
 * 多线程的创建，方式一：继承于Thread类
 * 1. 创建一个继承于Thread类的子类
 * 2. 重写Thread类的run() --> 将此线程执行的操作声明在run()中
 * 3. 创建Thread类的子类的对象
 * 4. 通过此对象调用start()
 * <p>
 * 例子：遍历100以内的所有的偶数
 *
 * @author shkstart
 * @create 2019-02-13 上午 11:46
 */

//1. 创建一个继承于Thread类的子类
class MyThread extends Thread {
    //2. 重写Thread类的run()
    @Override
    public void run() {
        for (int i = 0; i < 100; i++) {
            if(i % 2 == 0){
                System.out.println(Thread.currentThread().getName() + ":" + i);
            }
        }
    }
}

public class ThreadTest {
    public static void main(String[] args) {
        //3. 创建Thread类的子类的对象
        MyThread t1 = new MyThread();
    }
}
```

```

//4.通过此对象调用start():①启动当前线程 ② 调用当前线程的run()
t1.start();
//问题一：我们不能通过直接调用run()的方式启动线程。
//      t1.run();

//问题二：再启动一个线程，遍历100以内的偶数。不可以还让已经start()的线程去执行。会报
//IllegalThreadStateException
//      t1.start();
//我们需要重新创建一个线程的对象
MyThread t2 = new MyThread();
t2.start();

//如下操作仍然是在main线程中执行的。
for (int i = 0; i < 100; i++) {
    if(i % 2 == 0){
        System.out.println(Thread.currentThread().getName() + ":" + i +
"*****main()*****");
    }
}
}
}
}

```

(2) 方式二：实现Runnable 接口

- 1) 定义子类，实现Runnable接口。
- 2) 子类中重写Runnable接口中的run方法。
- 3) 通过Thread类含参构造器创建线程对象。
- 4) 将Runnable接口的子类对象作为实际参数传递给Thread类的构造器中。
- 5) 调用Thread类的start方法：开启线程，调用Runnable子类接口的run方法。

```

package atguigu.java;

/**
 * 创建多线程的方式二：实现Runnable接口
 * 1. 创建一个实现了Runnable接口的类
 * 2. 实现类去实现Runnable中的抽象方法：run()
 * 3. 创建实现类的对象
 * 4. 将此对象作为参数传递到Thread类的构造器中，创建Thread类的对象
 * 5. 通过Thread类的对象调用start()
 *
 *
 * 比较创建线程的两种方式。
 * 开发中：优先选择：实现Runnable接口的方式
 * 原因：1. 实现的方式没有类的单继承性的局限性
 *        2. 实现的方式更适合来处理多个线程有共享数据的情况。
 *
 * 联系：public class Thread implements Runnable
 * 相同点：两种方式都需要重写run(),将线程要执行的逻辑声明在run()中。
 *
 * @author shkstart

```

```

* @create 2019-02-13 下午 4:34
*/
//1. 创建一个实现了Runnable接口的类
class MThread implements Runnable{

    //2. 实现类去实现Runnable中的抽象方法: run()
    @Override
    public void run() {
        for (int i = 0; i < 100; i++) {
            if(i % 2 == 0){
                System.out.println(Thread.currentThread().getName() + ":" + i);
            }
        }
    }
}

public class ThreadTest1 {
    public static void main(String[] args) {
        //3. 创建实现类的对象
        MThread mThread = new MThread();
        //4. 将此对象作为参数传递到Thread类的构造器中, 创建Thread类的对象
        Thread t1 = new Thread(mThread);
        t1.setName("线程1");
        //5. 通过Thread类的对象调用start():① 启动线程 ②调用当前线程的run()-->调用了
        Runnable类型的target的run()
        t1.start();

        //再启动一个线程, 遍历100以内的偶数
        Thread t2 = new Thread(mThread);
        t2.setName("线程2");
        t2.start();
    }
}

```

(3) 继承方式和实现方式的联系与区别

```
public class Thread extends Object implements Runnable
```

● 区别

- 继承Thread: 线程代码存放Thread子类run方法中。
- 实现Runnable: 线程代码存在接口的子类的run方法。

● 实现方式的好处

- 避免了单继承的局限性
- 多个线程可以共享同一个接口实现类的对象, 非常适合多个相同线程来处理同一份资源。

4、Thread 类的有关方法

- void start():** 启动线程，并执行对象的run()方法
- run():** 线程在被调度时执行的操作
- String getName():** 返回线程的名称
- void setName(String name):** 设置该线程名称
- static Thread currentThread():** 返回当前线程。在Thread子类中就是this，通常用于主线程和Runnable实现类
- static void yield():** 线程让步
 - 暂停当前正在执行的线程，把执行机会让给优先级相同或更高的线程
 - 若队列中没有同优先级的线程，忽略此方法
- join() :** 当某个程序执行流中调用其他线程的 join() 方法时，调用线程将被阻塞，直到 join() 方法加入的 join 线程执行完为止
 - 低优先级的线程也可以获得执行
- static void sleep(long millis):** (指定时间:毫秒)
 - 令当前活动线程在指定时间段内放弃对CPU控制,使其他线程有机会被执行,时间到后重排队。
 - 抛出InterruptedException异常
- stop():** 强制线程生命期结束，不推荐使用
- boolean isAlive():** 返回boolean，判断线程是否还活着

5、线程的调度

●调度策略

- 时间片



- **抢占式：高优先级的线程抢占CPU**

●Java的调度方法

- 同优先级线程组成先进先出队列（先到先服务），使用时间片策略
- 对高优先级，使用优先调度的抢占式策略

6、线程的优先级

● 线程的优先级等级

- **MAX_PRIORITY: 10**
- **MIN_PRIORITY: 1**
- **NORM_PRIORITY: 5**

● 涉及的方法

- **getPriority()** : 返回线程优先值
- **setPriority(int newPriority)** : 改变线程的优先级

● 说明

- 线程创建时继承父线程的优先级
- 低优先级只是获得调度的概率低，并非一定是在高优先级线程之后才被调用

```
package atguigu.java;

/**
 * 测试Thread中的常用方法:
 * 1. start():启动当前线程;调用当前线程的run()
 * 2. run(): 通常需要重写Thread类中的此方法,将创建的线程要执行的操作声明在此方法中
 * 3. currentThread():静态方法,返回执行当前代码的线程
 * 4. getName():获取当前线程的名字
 * 5. setName():设置当前线程的名字
 * 6. yield():释放当前cpu的执行权
 * 7. join():在线程a中调用线程b的join(),此时线程a就进入阻塞状态,直到线程b完全执行完以后,线程a才
 *     结束阻塞状态。
 * 8. stop():已过时。当执行此方法时,强制结束当前线程。
 * 9. sleep(long millitime):让当前线程“睡眠”指定的millitime毫秒。在指定的millitime毫秒
 *     时间内,当前
 *     线程是阻塞状态。
 * 10. isAlive():判断当前线程是否存活
 *
 * 线程的优先级:
 * 1.
 * MAX_PRIORITY: 10
 * MIN_PRIORITY: 1
 * NORM_PRIORITY: 5 -->默认优先级
 * 2.如何获取和设置当前线程的优先级:
 *     getPriority():获取线程的优先级
 *     setPriority(int p):设置线程的优先级
 *
 * 说明:高优先级的线程要抢占低优先级线程cpu的执行权。但是只是从概率上讲,高优先级的线程高概
 * 率的情况下
 *     被执行。并不意味着只有当高优先级的线程执行完以后,低优先级的线程才执行。
 *
 * @author shkstart
 * @create 2019-02-13 下午 2:26
 */
class HelloThread extends Thread{
    @Override
    public void run() {
        for (int i = 0; i < 100; i++) {
            if(i % 2 == 0){
```



```

//            try {
//                sleep(10);
//            } catch (InterruptedException e) {
//                e.printStackTrace();
//            }

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

//            if(i % 20 == 0){
//                yield(); // 释放当前线程cpu的执行权
//            }

    }

}

public HelloThread(String name){
    super(name);
}

}

public class ThreadMethodTest {
    public static void main(String[] args) {

        HelloThread h1 = new HelloThread("Thread: 1");

//        h1.setName("线程一");
//        设置分线程的优先级
        h1.setPriority(Thread.MAX_PRIORITY);

        h1.start();

//        给主线程命名
        Thread.currentThread().setName("主线程");
        Thread.currentThread().setPriority(Thread.MIN_PRIORITY);

        for (int i = 0; i < 100; i++) {
            if(i % 2 == 0){
                System.out.println(Thread.currentThread().getName() + ":" +
Thread.currentThread().getPriority() + ":" + i);
            }

//            if(i == 20){
//                try {
//                    h1.join(); // 主线程阻塞
//                } catch (InterruptedException e) {
//                    e.printStackTrace();
//                }
//            }

        }

//        System.out.println(h1.isAlive());

```

```
}  
}
```

7、线程的分类：守护线程和用户线程

Java中的线程分为两类：一种是**守护线程**，一种是**用户线程**。

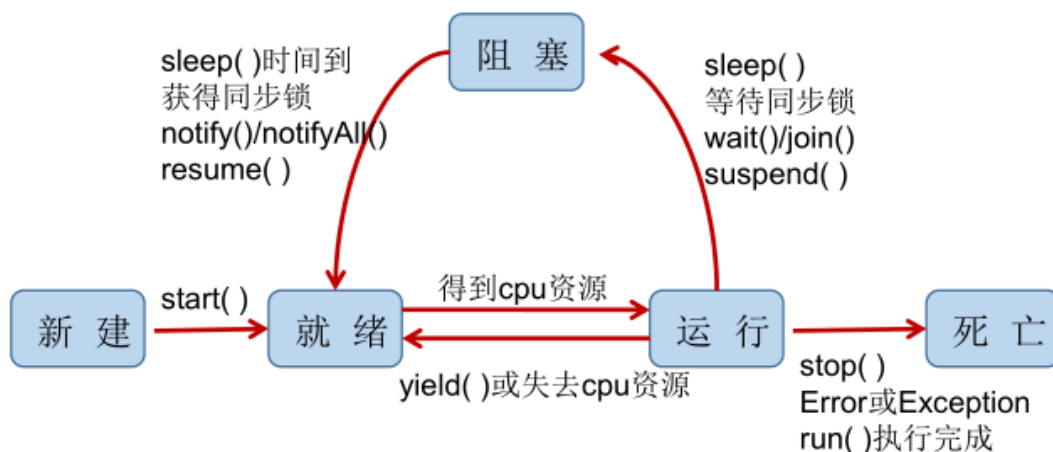
- 它们在几乎每个方面都是相同的，唯一的区别是判断JVM何时离开。
- 守护线程是用来服务用户线程的，通过在**start()**方法前调用**thread.setDaemon(true)**可以把一个用户线程变成一个守护线程。
- Java垃圾回收就是一个典型的守护线程。
- 若JVM中都是守护线程，当前JVM将退出。
- 形象理解：兔死狗烹，鸟尽弓藏

三、线程的生命周期

● JDK中用Thread.State类定义了线程的几种状态

要想实现多线程，必须在主线程中创建新的线程对象。Java语言使用Thread类及其子类的对象来表示线程，在它的一个完整生命周期中通常要经历如下的五种状态：

- **新建**：当一个Thread类或其子类的对象被声明并创建时，新生的线程对象处于新建状态
- **就绪**：处于新建状态的线程被**start()**后，将进入线程队列等待CPU时间片，此时它已具备了运行的条件，只是没分配到CPU资源
- **运行**：当就绪的线程被调度并获得CPU资源时，便进入运行状态，**run()**方法定义了线程的操作和功能
- **阻塞**：在某种特殊情况下，被人为挂起或执行输入输出操作时，让出CPU并临时中止自己的执行，进入阻塞状态
- **死亡**：线程完成了它的全部工作或线程被提前强制性地中止或出现异常导致结束

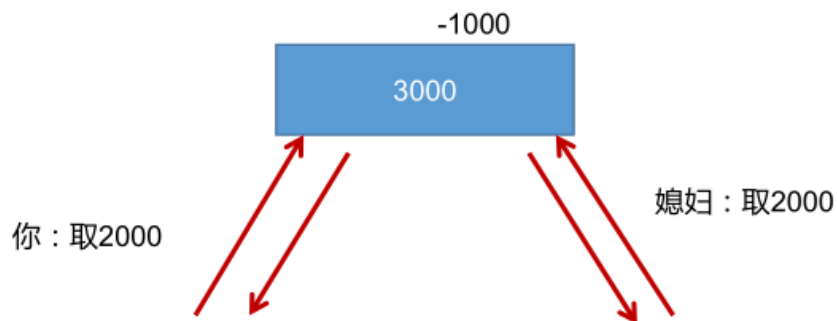


线程状态转换图

四、线程的同步和安全

1、理解线程的安全问题

- 多个线程执行的不确定性引起执行结果的不稳定
- 多个线程对账本的共享，会造成操作的不完整性，会破坏数据。



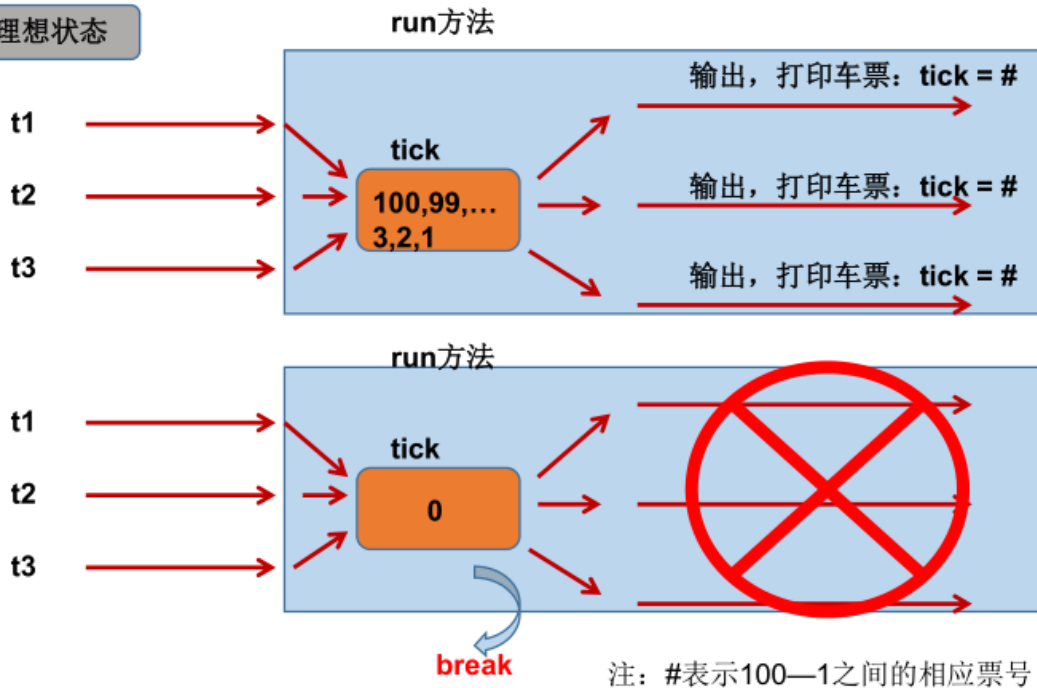
2、线程安全问题的举例和解决措施

模拟火车站售票程序，开启三个窗口售票：

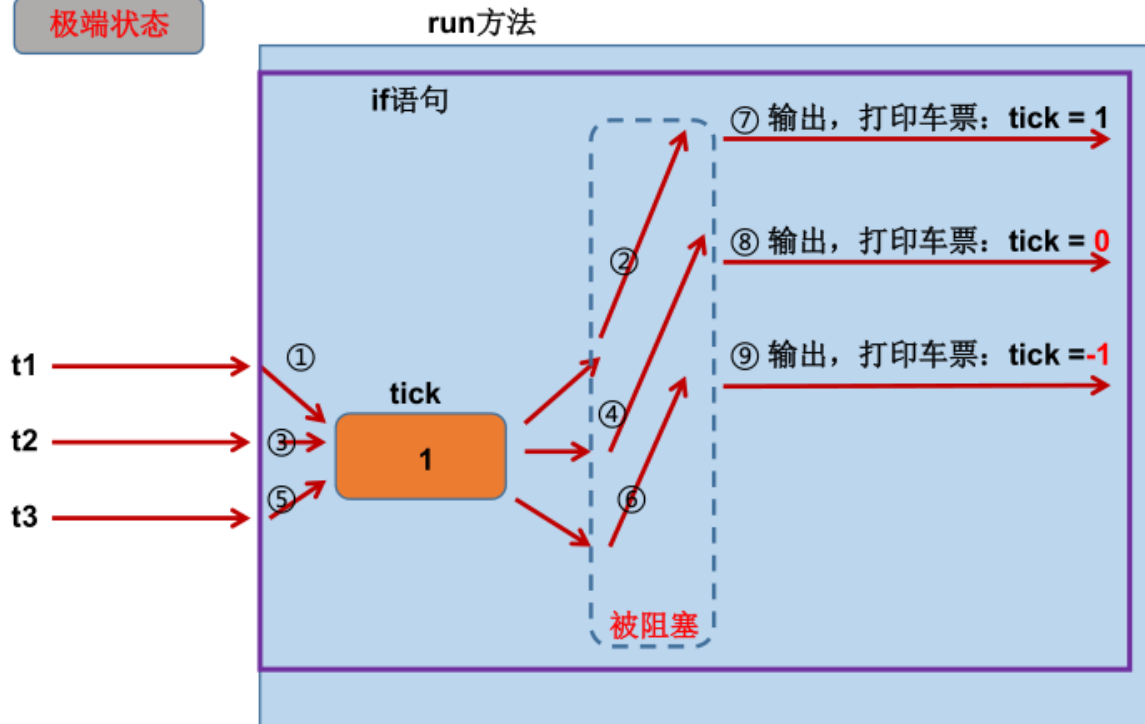
```
class Ticket implements Runnable {  
    private int tick = 100;  
  
    public void run() {  
        while (true) {  
            if (tick > 0) {  
                System.out.println(Thread.currentThread()  
                    ().getName() + "售出车票，tick号为：" +  
                    tick--);  
            } else  
                break;  
        }  
    }  
}
```

```
class TicketDemo {  
    public static void main(String[] args) {  
  
        Ticket t = new Ticket();  
  
        Thread t1 = new Thread(t);  
        Thread t2 = new Thread(t);  
        Thread t3 = new Thread(t);  
        t1.setName("t1窗口");  
        t2.setName("t2窗口");  
        t3.setName("t3窗口");  
        t1.start();  
        t2.start();  
        t3.start();  
    }  
}
```

理想状态



极端状态



```
private int tick = 100;
public void run(){
    while(true){
        if(tick>0){
            try{
                Thread.sleep(10);
            }catch(InterruptedException e){ e.printStackTrace();}
            System.out.println(Thread.currentThread().getName()+"售出车票, tick号为: "+tick--);
        }
    }
}
```

1. 多线程出现了安全问题

2. 问题的原因:

当多条语句在操作同一个线程共享数据时, 一个线程对多条语句只执行了一部分, 还没有执行完, 另一个线程参与进来执行。导致共享数据的错误。

3. 解决办法:

对多条操作共享数据的语句, 只能让一个线程都执行完, 在执行过程中, 其他线程不可以参与执行。

3、Synchronized同步机制解决线程的安全问题

- Java对于多线程的安全问题提供了专业的解决方式：**同步机制**

(1) 同步代码块

1. 同步代码块：
synchronized (对象){
// 需要被同步的代码；
}

说明：

- 1.操作共享数据的代码，即为需要被同步的代码。-->不能包含代码多了，也不能包含代码少了。
- 2.共享数据：多个线程共同操作的变量。比如：ticket就是共享数据。
- 3.同步监视器，俗称：锁。任何一个类的对象，都可以充当锁。
- 4.要求：多个线程必须要共用同一把锁。

补充：在实现Runnable接口创建多线程的方式中，我们可以考虑使用this充当同步监视器。

① 使用同步代码块解决实现Runnable接口的线程安全问题

```
package com.atguigu.java;

/**
 * 例子：创建三个窗口卖票，总票数为100张.使用实现Runnable接口的方式
 *
 * 1.问题：卖票过程中，出现了重票、错票 -->出现了线程的安全问题
 * 2.问题出现的原因：当某个线程操作车票的过程中，尚未操作完成时，其他线程参与进来，也操作车票。
 * 3.如何解决：当一个线程a在操作ticket的时候，其他线程不能参与进来。直到线程a操作完ticket时，其他
    线程才可以开始操作ticket。这种情况即使线程a出现了阻塞，也不能被改变。
 *
 * 4.在Java中，我们通过同步机制，来解决线程的安全问题。
 *
 * 方式一：同步代码块
 *
 * synchronized(同步监视器){
 *     //需要被同步的代码
 * }
 * 说明：1.操作共享数据的代码，即为需要被同步的代码。 -->不能包含代码多了，也不能包含代码少了。
 *
 * 2.共享数据：多个线程共同操作的变量。比如：ticket就是共享数据。
 * 3.同步监视器，俗称：锁。任何一个类的对象，都可以充当锁。
 *     要求：多个线程必须要共用同一把锁。
 *
 * 补充：在实现Runnable接口创建多线程的方式中，我们可以考虑使用this充当同步监视器。
 * 方式二：同步方法。
 *     如果操作共享数据的代码完整的声明在一个方法中，我们不妨将此方法声明同步的。
 *
 * 
```

```

*
* 5. 同步的方式，解决了线程的安全问题。---好处
* 操作同步代码时，只能有一个线程参与，其他线程等待。相当于是一个单线程的过程，效率低。 ---
局限性
*
* @author shkstart
* @create 2019-02-13 下午 4:47
*/
class Window1 implements Runnable{

    private int ticket = 100;
    // Object obj = new Object();
    // Dog dog = new Dog();
    @Override
    public void run() {
        // Object obj = new Object();
        while(true){
            synchronized (this){//此时的this:唯一的Window1的对象 //方式二:
synchronized (dog) {

                if (ticket > 0) {

                    try {
                        Thread.sleep(100);
                    } catch (InterruptedException e) {
                        e.printStackTrace();
                    }

                    System.out.println(Thread.currentThread().getName() + ":卖
票，票号为: " + ticket);

                    ticket--;
                } else {
                    break;
                }
            }
        }
    }
}

public class WindowTest1 {
    public static void main(String[] args) {
        Window1 w = new Window1();

        Thread t1 = new Thread(w);
        Thread t2 = new Thread(w);
        Thread t3 = new Thread(w);

        t1.setName("窗口1");
        t2.setName("窗口2");
        t3.setName("窗口3");

        t1.start();
        t2.start();
        t3.start();
    }
}

```

```

}

class Dog{

}

```

② 使用同步代码块解决继承Thread类的方式的线程安全问题

```

package com.atguigu.java;

/**
 * @author shkstart
 * @create 2019-02-15 上午 11:15
 */
/**
 * 使用同步代码块解决继承Thread类的方式的线程安全问题
 *
 * 例子：创建三个窗口卖票，总票数为100张.使用继承Thread类的方式
 *
 * 说明：在继承Thread类创建多线程的方式中，慎用this充当同步监视器，考虑使用当前类充当同步监视器。
 */
class Window2 extends Thread{

    private static int ticket = 100;

    private static Object obj = new Object();

    @Override
    public void run() {

        while(true){
            //正确的
            // synchronized (obj){
            synchronized (Window2.class){//Class clazz =
            Window2.class,Window2.class只会加载一次
            //错误的方式： this代表着t1,t2,t3三个对象
            // synchronized (this){

                if(ticket > 0){

                    try {
                        Thread.sleep(100);
                    } catch (InterruptedException e) {
                        e.printStackTrace();
                    }

                    System.out.println(getName() + ": 卖票, 票号为: " + ticket);
                    ticket--;
                }
            }
        }
    }
}

```


* @create 2019-02-15 上午 11:35

*/

```
class Window3 implements Runnable {

    private int ticket = 100;

    @Override
    public void run() {
        while (true) {

            show();

        }
    }

    private synchronized void show(){//同步监视器: this
        //synchronized (this){

            if (ticket > 0) {

                try {
                    Thread.sleep(100);
                } catch (InterruptedException e) {
                    e.printStackTrace();
                }

                System.out.println(Thread.currentThread().getName() + ":卖票, 票号
为: " + ticket);

                ticket--;
            }
        }
    }

}

public class WindowTest3 {
    public static void main(String[] args) {
        Window3 w = new Window3();

        Thread t1 = new Thread(w);
        Thread t2 = new Thread(w);
        Thread t3 = new Thread(w);

        t1.setName("窗口1");
        t2.setName("窗口2");
        t3.setName("窗口3");

        t1.start();
        t2.start();
        t3.start();
    }
}
```

② 使用同步方法处理继承Thread类的方式中的线程安全问题

```
package com.atguigu.java;

/**
 * 使用同步方法处理继承Thread类的方式中的线程安全问题
 *
 * @author shkstart
 * @create 2019-02-15 上午 11:43
 */
class Window4 extends Thread {

    private static int ticket = 100;

    @Override
    public void run() {

        while (true) {

            show();

        }

    }

    private static synchronized void show(){//同步监视器: window4.class
        //private synchronized void show(){ //同步监视器: t1,t2,t3。此种解决方式是错误的
        if (ticket > 0) {

            try {
                Thread.sleep(100);
            } catch (InterruptedException e) {
                e.printStackTrace();
            }

            System.out.println(Thread.currentThread().getName() + ": 卖票, 票号为: " + ticket);
            ticket--;

        }

    }

}

public class WindowTest4 {
    public static void main(String[] args) {
        Window4 t1 = new Window4();
        Window4 t2 = new Window4();
        Window4 t3 = new Window4();

        t1.setName("窗口1");
        t2.setName("窗口2");
        t3.setName("窗口3");

        t1.start();
    }
}
```

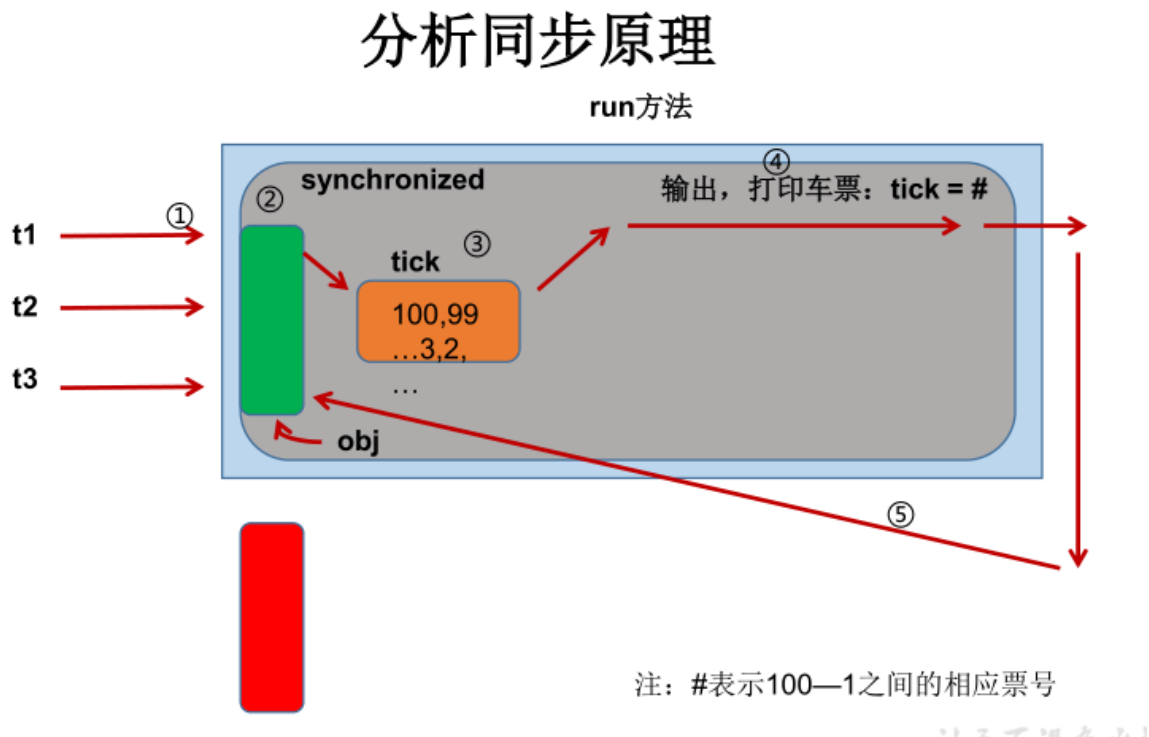
```

        t2.start();
        t3.start();

    }
}

```

(3) 分析同步原理



4、同步机制中的锁

同步机制中的锁

● 同步锁机制:

在《Thinking in Java》中，是这么说的：对于并发工作，你需要某种方式来防止两个任务访问相同的资源（其实就是共享资源竞争）。防止这种冲突的方法就是当资源被一个任务使用时，在其上加锁。第一个访问某项资源的任务必须锁定这项资源，使其他任务在其被解锁之前，就无法访问它了，而在其被解锁之时，另一个任务就可以锁定并使用它了。

● synchronized的锁是什么？

- 任意对象都可以作为同步锁。所有对象都自动含有单一的锁（监视器）。
- 同步方法的锁：静态方法（类名.class）、非静态方法（this）
- 同步代码块：自己指定，很多时候也是指定为this或类名.class

● 注意：

- 必须确保使用同一个资源的**多个线程共用一把锁**，这个非常重要，否则就无法保证共享资源的安全
- 一个线程类中的所有静态方法共用同一把锁（类名.class），所有非静态方法共用同一把锁（this），同步代码块（指定需谨慎）

让天下没有难学的技

5、同步的范围

1、如何找问题，即代码是否存在线程安全？（非常重要）

- (1) 明确哪些代码是多线程运行的代码
- (2) 明确多个线程是否有共享数据
- (3) 明确多线程运行代码中是否有多条语句操作共享数据

2、如何解决呢？（非常重要）

对多条操作共享数据的语句，只能让一个线程都执行完，在执行过程中，其他线程不可以参与执行。

即所有操作共享数据的这些语句都要放在同步范围中

3、切记：

- 范围太小：没锁住所有有安全问题的代码
- 范围太大：没发挥多线程的功能。

6、释放锁的操作

- 当前线程的同步方法、同步代码块执行结束。
- 当前线程在同步代码块、同步方法中遇到`break`、`return`终止了该代码块、该方法的继续执行。
- 当前线程在同步代码块、同步方法中出现了未处理的`Error`或`Exception`，导致异常结束。
- 当前线程在同步代码块、同步方法中执行了线程对象的`wait()`方法，当前线程暂停，并释放锁。

7、不会释放锁的操作

- 线程执行同步代码块或同步方法时，程序调用`Thread.sleep()`、`Thread.yield()`方法暂停当前线程的执行
- 线程执行同步代码块时，其他线程调用了该线程的`suspend()`方法将该线程挂起，该线程不会释放锁（同步监视器）。
 - 应尽量避免使用`suspend()`和`resume()`来控制线程

8、单例设计模式之懒汉式(线程安全)

```
class Singleton {
    private static Singleton instance = null;

    private Singleton() {
    }

    public static Singleton getInstance() {
        if (instance == null) {
            synchronized (Singleton.class) {
                if (instance == null) {
                    instance = new Singleton();
                }
            }
        }
    }
}
```

```

        return instance;
    }
}

public class SingletonTest {
    public static void main(String[] args) {
        Singleton s1 = Singleton.getInstance();
        Singleton s2 = Singleton.getInstance();
        System.out.println(s1 == s2);
    }
}

```

9、线程的死锁问题

●死锁

- 不同的线程分别占用对方需要的同步资源不放弃，都在等待对方放弃自己需要的同步资源，就形成了线程的死锁
- 出现死锁后，不会出现异常，不会出现提示，只是所有的线程都处于阻塞状态，无法继续

●解决方法

- 专门的算法、原则
- 尽量减少同步资源的定义
- 尽量避免嵌套同步

```

package com.atguigu.java1;

//死锁的演示
class A {
    public synchronized void foo(B b) { //同步监视器：A类的对象：a
        System.out.println("当前线程名：" + Thread.currentThread().getName()
            + " 进入了A实例的foo方法"); // ①
        try {
            Thread.sleep(200);
        } catch (InterruptedException ex) {
            ex.printStackTrace();
        }
        System.out.println("当前线程名：" + Thread.currentThread().getName()
            + " 企图调用B实例的last方法"); // ③
        b.last();
    }

    public synchronized void last() { //同步监视器：A类的对象：a
        System.out.println("进入了A类的last方法内部");
    }
}

class B {
    public synchronized void bar(A a) { //同步监视器：b
        System.out.println("当前线程名：" + Thread.currentThread().getName()
            + " 进入了B实例的bar方法"); // ②
        try {

```

```

        Thread.sleep(200);
    } catch (InterruptedException ex) {
        ex.printStackTrace();
    }
    System.out.println("当前线程名: " + Thread.currentThread().getName()
        + " 企图调用A实例的last方法"); // ④
    a.last();
}

public synchronized void last() { //同步监视器: b
    System.out.println("进入了B类的last方法内部");
}
}

public class DeadLock implements Runnable {
    A a = new A();
    B b = new B();

    public void init() {
        Thread.currentThread().setName("主线程");
        // 调用a对象的foo方法
        a.foo(b);
        System.out.println("进入了主线程之后");
    }

    @Override
    public void run() {
        Thread.currentThread().setName("副线程");
        // 调用b对象的bar方法
        b.bar(a);
        System.out.println("进入了副线程之后");
    }

    public static void main(String[] args) {
        DeadLock dl = new DeadLock();
        new Thread(dl).start();

        dl.init();
    }
}

```

10、Lock锁方式解决线程安全问题

- 从JDK 5.0开始，Java提供了更强大的线程同步机制——通过显式定义同步锁对象来实现同步。同步锁使用Lock对象充当。
- **java.util.concurrent.locks.Lock**接口是控制多个线程对共享资源进行访问的工具。锁提供了对共享资源的独占访问，每次只能有一个线程对Lock对象加锁，线程开始访问共享资源之前应先获得Lock对象。
- ReentrantLock 类实现了 Lock ，它拥有与 synchronized 相同的并发性和内存语义，在实现线程安全的控制中，比较常用的是ReentrantLock，可以显式加锁、释放锁。

```
class A{
    private final ReentrantLock lock = new ReentrantLock();
    public void m(){
        lock.lock();
        try{
            //保证线程安全的代码;
        }
        finally{
            lock.unlock();
        }
    }
}
```

注意：如果同步代码有异常，要将unlock()写入finally语句块

```
package com.atguigu.java1;

import java.util.concurrent.locks.ReentrantLock;

/**
 * 解决线程安全问题的方式三：Lock锁 --- JDK5.0新增
 *
 * 1. 面试题：synchronized 与 Lock的异同？
 * 相同：二者都可以解决线程安全问题
 * 不同：synchronized机制在执行完相应的同步代码以后，自动的释放同步监视器
 * Lock需要手动的启动同步（lock()），同时结束同步也需要手动的实现（unlock()）
 *
 * 2. 优先使用顺序：
 * Lock □ 同步代码块（已经进入了方法体，分配了相应资源） □ 同步方法（在方法体之外）
 *
 * 面试题：如何解决线程安全问题？有几种方式
 * @author shkstart
 * @create 2019-02-15 下午 3:38
 */
class Window implements Runnable{

    private int ticket = 100;
    //1.实例化ReentrantLock
    private ReentrantLock lock = new ReentrantLock();

    @Override
    public void run() {
```

```

        while(true){
            try{

                //2.调用锁定方法lock()
                lock.lock();

                if(ticket > 0){

                    try {
                        Thread.sleep(100);
                    } catch (InterruptedException e) {
                        e.printStackTrace();
                    }

                    System.out.println(Thread.currentThread().getName() + ": 售票, 票号为: " + ticket);
                    ticket--;
                }else{
                    break;
                }
            }finally {
                //3.调用解锁方法: unlock()
                lock.unlock();
            }
        }
    }
}

public class LockTest {
    public static void main(String[] args) {
        Window w = new Window();

        Thread t1 = new Thread(w);
        Thread t2 = new Thread(w);
        Thread t3 = new Thread(w);

        t1.setName("窗口1");
        t2.setName("窗口2");
        t3.setName("窗口3");

        t1.start();
        t2.start();
        t3.start();
    }
}

```

11、synchronized 与 Lock 的对比

1. **Lock**是显式锁（手动开启和关闭锁，别忘记关闭锁），**synchronized**是隐式锁，出了作用域自动释放
2. **Lock**只有代码块锁，**synchronized**有代码块锁和方法锁
3. 使用**Lock**锁，JVM将花费较少的时间来调度线程，性能更好。并且具有更好的扩展性（提供更多的子类）

优先使用顺序：

Lock → 同步代码块（已经进入了方法体，分配了相应资源） → 同步方法（在方法体之外）

五、线程的通信

1、线程通信的例题

```
package com.atguigu.java2;

/**
 * 线程通信的例子：使用两个线程打印 1-100。线程1，线程2 交替打印
 *
 * 涉及到的三个方法：
 * wait():一旦执行此方法，当前线程就进入阻塞状态，并释放同步监视器。
 * notify():一旦执行此方法，就会唤醒被wait的一个线程。如果有多个线程被wait，就唤醒优先级高的那个。
 * notifyAll():一旦执行此方法，就会唤醒所有被wait的线程。
 *
 * 说明：
 * 1.wait(), notify(), notifyAll()三个方法必须使用在同步代码块或同步方法中。
 * 2.wait(), notify(), notifyAll()三个方法的调用者必须是同步代码块或同步方法中的同步监视器。
 * 否则，会出现IllegalMonitorStateException异常
 * 3.wait(), notify(), notifyAll()三个方法是定义在java.lang.Object类中。
 *
 * 面试题：sleep() 和 wait()的异同？
 * 1.相同点：一旦执行方法，都可以使得当前的线程进入阻塞状态。
 * 2.不同点：1) 两个方法声明的位置不同：Thread类中声明sleep()，Object类中声明wait()
 * 2) 调用的要求不同：sleep()可以在任何需要的场景下调用。wait()必须使用在同步代码块或同步方法中
 * 3) 关于是否释放同步监视器：如果两个方法都使用在同步代码块或同步方法中，sleep()不会释放锁，wait()会释放锁。
 *
 * @author shkstart
 * @create 2019-02-15 下午 4:21
 */
class Number implements Runnable{
    private int number = 1;
    private Object obj = new Object();
    @Override
    public void run() {

        while(true){

            synchronized (obj) {
```

```

        obj.notify();

        if(number <= 100){

            try {
                Thread.sleep(10);
            } catch (InterruptedException e) {
                e.printStackTrace();
            }

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

            number++;

            try {
                //使得调用如下wait()方法的线程进入阻塞状态
                obj.wait();
            } catch (InterruptedException e) {
                e.printStackTrace();
            }

            }else{
                break;
            }
        }
    }
}

public class CommunicationTest {
    public static void main(String[] args) {
        Number number = new Number();
        Thread t1 = new Thread(number);
        Thread t2 = new Thread(number);

        t1.setName("线程1");
        t2.setName("线程2");

        t1.start();
        t2.start();
    }
}

```

2、wait() 与 notify() 和 notifyAll()

- **wait()**: 令当前线程挂起并放弃CPU、同步资源并等待，使别的线程可访问并修改共享资源，而当前线程排队等候其他线程调用**notify()**或**notifyAll()**方法唤醒，唤醒后等待重新获得对监视器的所有权后才能继续执行。
- **notify()**: 唤醒正在排队等待同步资源的线程中优先级最高者结束等待
- **notifyAll ()**: 唤醒正在排队等待资源的所有线程结束等待。
- 这三个方法只有在**synchronized**方法或**synchronized**代码块中才能使用，否则会报 **java.lang.IllegalMonitorStateException** 异常。
- 因为这三个方法必须有锁对象调用，而任意对象都可以作为**synchronized**的同步锁，因此这三个方法只能在**Object**类中声明。

wait() 方法

- 在当前线程中调用方法： 对象名.wait()
- 使当前线程进入等待（某对象）状态，直到另一线程对该对象发出 **notify** (或**notifyAll**) 为止。
- 调用方法的必要条件：当前线程必须具有对该对象的监控权（加锁）
- 调用此方法后，当前线程将释放对象监控权，然后进入等待
- 在当前线程被**notify**后，要重新获得监控权，然后从断点处继续代码的执行。

notify()/notifyAll()

- 在当前线程中调用方法： 对象名.notify()
- 功能：唤醒等待该对象监控权的一个/所有线程。
- 调用方法的必要条件：当前线程必须具有对该对象的监控权（加锁）

3、经典例题：生产者/消费者问题

- 生产者(**Productor**)将产品交给店员(**Clerk**)，而消费者(**Customer**)从店员处取走产品，店员一次只能持有固定数量的产品(比如:20)，如果生产者试图生产更多的产品，店员会叫生产者停一下，如果店中有空位放产品了再通知生产者继续生产；如果店中没有产品了，店员会告诉消费者等一下，如果店中有产品了再通知消费者来取走产品。
- 这里可能出现两个问题：
 - 生产者比消费者快时，消费者会漏掉一些数据没有取到。
 - 消费者比生产者快时，消费者会取相同的数据。

```
package com.atguigu.java2;

/**
 * 线程通信的应用：经典例题：生产者/消费者问题
 *
 * 生产者(Productor)将产品交给店员(Clerk)，而消费者(Customer)从店员处取走产品，
```

```

* 店员一次只能持有固定数量的产品(比如:20)，如果生产者试图生产更多的产品，店员
* 会叫生产者停一下，如果店中有空位放产品了再通知生产者继续生产；如果店中没有产品
* 了，店员会告诉消费者等一下，如果店中有产品了再通知消费者来取走产品。
*
* 分析：
* 1. 是否是多线程问题？是，生产者线程，消费者线程
* 2. 是否有共享数据？是，店员（或产品）
* 3. 如何解决线程的安全问题？同步机制,有三种方法
* 4. 是否涉及线程的通信？是
*
* @author shkstart
* @create 2019-02-15 下午 4:48
*/
class Clerk{

    private int productCount = 0;
    //生产产品
    public synchronized void produceProduct() {

        if(productCount < 20){
            productCount++;
            System.out.println(Thread.currentThread().getName() + ":开始生产第" +
productCount + "个产品");

            notify();

        }else{
            //等待
            try {
                wait();
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
        }

    }
    //消费产品
    public synchronized void consumeProduct() {
        if(productCount > 0){
            System.out.println(Thread.currentThread().getName() + ":开始消费第" +
productCount + "个产品");
            productCount--;

            notify();
        }else{
            //等待
            try {
                wait();
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
        }

    }

}

class Producer extends Thread{//生产者

```

```

        private Clerk clerk;

        public Producer(Clerk clerk) {
            this.clerk = clerk;
        }

        @Override
        public void run() {
            System.out.println(getName() + ":开始生产产品.....");

            while(true){

                try {
                    Thread.sleep(10);
                } catch (InterruptedException e) {
                    e.printStackTrace();
                }

                clerk.produceProduct();
            }
        }
    }

    class Consumer extends Thread{//消费者
        private Clerk clerk;

        public Consumer(Clerk clerk) {
            this.clerk = clerk;
        }

        @Override
        public void run() {
            System.out.println(getName() + ":开始消费产品.....");

            while(true){

                try {
                    Thread.sleep(20);
                } catch (InterruptedException e) {
                    e.printStackTrace();
                }

                clerk.consumeProduct();
            }
        }
    }

    public class ProductTest {

        public static void main(String[] args) {
            Clerk clerk = new Clerk();

            Producer p1 = new Producer(clerk);
            p1.setName("生产者1");

            Consumer c1 = new Consumer(clerk);
            c1.setName("消费者1");
        }
    }

```



```
Consumer c2 = new Consumer(c1.clerk);
c2.setName("消费者2");

p1.start();
c1.start();
c2.start();

}

}
```

4、面试题：sleep() 和 wait()的异同

相同点：一旦执行方法，都可以使得当前的线程进入阻塞状态。

不同点：

- 1) 两个方法声明的位置不同：Thread类中声明sleep(), Object类中声明wait()
- 2) 调用的要求不同：sleep()可以在任何需要的场景下调用。wait()必须使用在同步代码块或同步方法中
- 3) 关于是否释放同步监视器：如果两个方法都使用在同步代码块或同步方法中，sleep()不会释放锁，wait()会释放锁。

六、JDK5.0新增线程创建方式(2种)

1、新增方式一：实现Callable

● 与使用Runnable相比， Callable功能更强大些

- 相比run()方法，可以有返回值
- 方法可以抛出异常
- 支持泛型的返回值
- 需要借助FutureTask类，比如获取返回结果

● Future接口

- 可以对具体Runnable、Callable任务的执行结果进行取消、查询是否完成、获取结果等。
- FutureTask是Future接口的唯一的实现类
- FutureTask 同时实现了Runnable, Future接口。它既可以作为Runnable被线程执行，又可以作为Future得到Callable的返回值

```
package com.atguigu.java2;
```

```

import java.util.concurrent.Callable;
import java.util.concurrent.ExecutionException;
import java.util.concurrent.FutureTask;

/**
 * 创建线程的方式三：实现Callable接口。 --- JDK 5.0新增
 *
 *
 * 如何理解实现Callable接口的方式创建多线程比实现Runnable接口创建多线程方式强大？
 * 1. call()可以有返回值的。
 * 2. call()可以抛出异常，被外面的操作捕获，获取异常的信息
 * 3. Callable是支持泛型的
 *
 * @author shkstart
 * @create 2019-02-15 下午 6:01
 */
//1. 创建一个实现Callable的实现类
class NumThread implements Callable{
    //2. 实现call方法，将此线程需要执行的操作声明在call()中
    @Override
    public Object call() throws Exception {
        int sum = 0;
        for (int i = 1; i <= 100; i++) {
            if(i % 2 == 0){
                System.out.println(i);
                sum += i;
            }
        }
        return sum;
    }
}

public class ThreadNew {
    public static void main(String[] args) {
        //3. 创建Callable接口实现类的对象
        NumThread numThread = new NumThread();
        //4. 将此Callable接口实现类的对象作为传递到FutureTask构造器中，创建FutureTask的对象
        FutureTask futureTask = new FutureTask(numThread);
        //5. 将FutureTask的对象作为参数传递到Thread类的构造器中，创建Thread对象，并调用start()
        new Thread(futureTask).start();

        try {
            //6. 获取Callable中call方法的返回值
            //get()返回值即为FutureTask构造器参数Callable实现类重写的call()的返回值。
            Object sum = futureTask.get();
            System.out.println("总和为: " + sum);
        } catch (InterruptedException e) {
            e.printStackTrace();
        } catch (ExecutionException e) {
            e.printStackTrace();
        }
    }
}

```

2、新增方式二：使用线程池

- **背景：**经常创建和销毁、使用量特别大的资源，比如并发情况下的线程，对性能影响很大。
- **思路：**提前创建好多个线程，放入线程池中，使用时直接获取，使用完放回池中。可以避免频繁创建销毁、实现重复利用。类似生活中的公共交通工具。
- **好处：**
 - 提高响应速度（减少了创建新线程的时间）
 - 降低资源消耗（重复利用线程池中线程，不需要每次都创建）
 - 便于线程管理
 - ✓ `corePoolSize`：核心池的大小
 - ✓ `maximumPoolSize`：最大线程数
 - ✓ `keepAliveTime`：线程没有任务时最多保持多长时间后会终止
 - ✓ ...

线程池相关API

- JDK 5.0起提供了线程池相关API：**ExecutorService** 和 **Executors**
- **ExecutorService**：真正的线程池接口。常见子类`ThreadPoolExecutor`
 - `void execute(Runnable command)`：执行任务/命令，没有返回值，一般用来执行`Runnable`
 - `<T> Future<T> submit(Callable<T> task)`：执行任务，有返回值，一般用来执行`Callable`
 - `void shutdown()`：关闭连接池
- **Executors**：工具类、线程池的工厂类，用于创建并返回不同类型的线程池
 - `Executors.newCachedThreadPool()`：创建一个可根据需要创建新线程的线程池
 - `Executors.newFixedThreadPool(n)`：创建一个可重用固定线程数的线程池
 - `Executors.newSingleThreadExecutor()`：创建一个只有一个线程的线程池
 - `Executors.newScheduledThreadPool(n)`：创建一个线程池，它可安排在给定延迟后运行命令或者定期地执行。

让天下没有难学的技术

```
package com.atguigu.java2;

import java.util.concurrent.ExecutorService;
import java.util.concurrent.Executors;
import java.util.concurrent.ThreadPoolExecutor;

/**
 * 创建线程的方式四：使用线程池
 *
 * 好处：
 * 1. 提高响应速度（减少了创建新线程的时间）
 * 2. 降低资源消耗（重复利用线程池中线程，不需要每次都创建）
 * 3. 便于线程管理
 *
 *     corePoolSize：核心池的大小
 *     maximumPoolSize：最大线程数
 *     keepAliveTime：线程没有任务时最多保持多长时间后会终止
 *
 */
```

```

* 面试题：创建多线程有几种方式？四种！
* @author shkstart
* @create 2019-02-15 下午 6:30
*/

class NumberThread implements Runnable{

    @Override
    public void run() {
        for(int i = 0;i <= 100;i++){
            if(i % 2 == 0){
                System.out.println(Thread.currentThread().getName() + ": " + i);
            }
        }
    }
}

class NumberThread1 implements Runnable{

    @Override
    public void run() {
        for(int i = 0;i <= 100;i++){
            if(i % 2 != 0){
                System.out.println(Thread.currentThread().getName() + ": " + i);
            }
        }
    }
}

public class ThreadPool {

    public static void main(String[] args) {

        //1. 提供指定线程数量的线程池
        ExecutorService service = Executors.newFixedThreadPool(10);
        ThreadPoolExecutor service1 = (ThreadPoolExecutor) service;
        //设置线程池的属性
        //    System.out.println(service.getClass());
        //    service1.setCorePoolSize(15);
        //    service1.setKeepAliveTime();

        //2. 执行指定的线程的操作。需要提供实现Runnable接口或Callable接口实现类的对象
        service.execute(new NumberThread());//适合适用于Runnable
        service.execute(new NumberThread1());//适合适用于Runnable

        //    service.submit(Callable callable);//适合使用于Callable
        //3. 关闭连接池
        service.shutdown();
    }
}

```

七、每日练习

1、谈谈你对程序、进程、线程的理解？

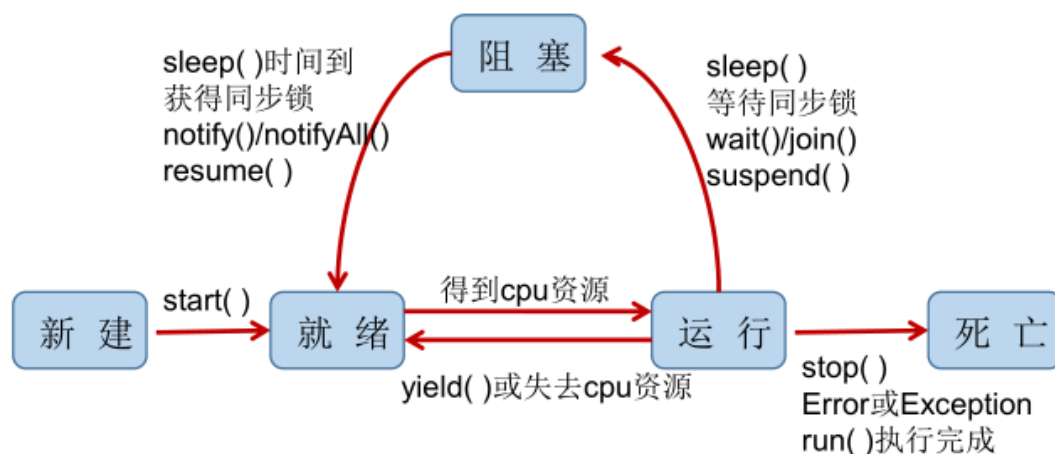
- **程序(program)**是为完成特定任务、用某种语言编写的一组指令的集合。即指一段静态的代码，静态对象。
- **进程(process)**是程序的一次执行过程，或是正在运行的一个程序。是一个动态的过程：有它自身的产生、存在和消亡的过程。——生命周期
 - 如：运行中的QQ，运行中的MP3播放器
 - 程序是静态的，进程是动态的
 - 进程作为资源分配的单位，系统在运行时会为每个进程分配不同的内存区域
- **线程(thread)**，进程可进一步细化为线程，是一个程序内部的一条执行路径。
 - 若一个进程同一时间并行执行多个线程，就是支持多线程的
 - 线程作为调度和执行的单位，每个线程拥有独立的运行栈和程序计数器(pc)，线程切换的开销小
 - 一个进程中的多个线程共享相同的内存单元/内存地址空间→它们从同一堆中分配对象，可以访问相同的变量和对象。这就使得线程间通信更简便、高效。但多个线程操作共享的系统资源可能就会带来安全的隐患。

2、对比继承和实现的两种多线程创建方式？

public class Thread extends Object implements Runnable

- **区别**
 - 继承Thread：线程代码存放Thread子类run方法中。
 - 实现Runnable：线程代码存在接口的子类的run方法。
- **实现方式的好处**
 - 避免了单继承的局限性
 - 多个线程可以共享同一个接口实现类的对象，非常适合多个相同线程来处理同一份资源。

3、画图说明线程的生命周期，以及各状态切换使用到的方法等？



4、同步代码块中涉及到同步监视器和共享数据，谈谈你对同步监视器和共享数据的理解，以及注意点？

共享数据：多个线程共同操作的变量。

同步监视器，俗称：锁。任何一个类的对象，都可以充当锁。

- **synchronized的锁是什么？**

- 任意对象都可以作为同步锁。所有对象都自动含有单一的锁（监视器）。
- 同步方法的锁：静态方法（类名.class）、非静态方法（this）
- 同步代码块：自己指定，很多时候也是指定为this或类名.class

- **注意：**

- 必须确保使用同一个资源的**多个线程共用一把锁**，这个非常重要，否则就无法保证共享资源的安全
- 一个线程类中的所有静态方法共用同一把锁（类名.class），所有非静态方法共用同一把锁（this），同步代码块（指定需谨慎）
让天下没有难学的技术。

5、sleep()和wait()的区别？

相同点：一旦执行方法，都可以使得当前的线程进入阻塞状态。

不同点：

- 1) 两个方法声明的位置不同：Thread类中声明sleep()，Object类中声明wait()
- 2) 调用的要求不同：sleep()可以在任何需要的场景下调用。wait()必须使用在同步代码块或同步方法中
- 3) 关于是否释放同步监视器：如果两个方法都使用在同步代码块或同步方法中，sleep()不会释放锁，wait()会释放锁。

6、写一个线程安全的懒汉式？

```
class Singleton {
    private static Singleton instance = null;

    private Singleton() {
    }

    public static Singleton getInstance() {
        if (instance == null) {
            synchronized (Singleton.class) {
                if (instance == null) {
                    instance = new Singleton();
                }
            }
        }
        return instance;
    }
}

public class SingletonTest {
    public static void main(String[] args) {
        Singleton s1 = Singleton.getInstance();
        Singleton s2 = Singleton.getInstance();
        System.out.println(s1 == s2);
    }
}
```

7、创建多线程有哪几种方式？

4种。

- 继承Thread类
- 实现Runnable接口
- 实现Callable接口
- 线程池（响应速度提高了，提高了资源的重用率，便于管理）

8、java中有几种方法可以实现一个线程(jdk5.0之前)? 用什么关键字修饰同步方法? stop()和suspend()方法为何不推荐使用?

有两种实现方法，分别是继承Thread类与实现Runnable接口。

用synchronized关键字修饰同步方法。

反对使用stop()，是因为它不安全。它会解除由线程获取的所有锁定，而且如果对象处于一种不连贯状态，那么其他线程能在那种状态下检查和修改它们。结果很难检查出真正的问题所在。

suspend()方法容易发生死锁。调用suspend()的时候，目标线程会停下来，但却仍然持有在这之前获得的锁定。此时，其他任何线程都不能访问锁定的资源，除非被"挂起"的线程恢复运行。对任何线程来说，如果它们想恢复目标线程，同时又试图使用任何一个锁定的资源，就会造成死锁。所以不应该使用suspend()，而应在自己的Thread类中置入一个标志，指出线程应该活动还是挂起。若标志指出线程应该挂起，使用wait()命其进入等待状态。若标志指出线程应当恢复，则用一个notify()重新启动线程。

9、同步和异步有何异同，在什么情况下分别使用他们?

如果数据将在线程间共享。例如正在写的数据以后可能被另一个线程读到，或者正在读的数据可能已经被另一个线程写过了，那么这些数据就是共享数据，必须进行同步存取。

当应用程序在对象上调用了需要一个花费很长时间来执行的方法，并且不希望让程序等待方法的返回时，就应该使用异步编程，在很多情况下采用异步途径往往更有效率。

10、启动一个线程是用run()还是start()?

启动一个线程是调用start()方法，使线程所代表的虚拟处理机处于可运行状态，这意味着它可以由JVM调度并执行。这并不意味着线程就会立即运行。run()方法就是正常的对象调用方法的执行，并不是使用分线程来执行的。

11、当一个线程进入一个对象的一个synchronized方法后，其它线程是否可进入此对象的其它方法?

不能，一个对象的一个synchronized方法只能由一个线程访问。

12、请说出你所知道的线程同步的方法?

wait():使一个线程处于等待状态，并且释放所持有的对象的lock。

sleep():使一个正在运行的线程处于睡眠状态，是一个静态方法，调用此方法要捕捉InterruptedException异常。

notify():唤醒一个处于等待状态的线程，注意的是在调用此方法的时候，并不能确切的唤醒某一个等待状态的线程，而是由JVM确定唤醒哪个线程，而且不是按优先级。

notifyAll():唤醒所有处于等待状态的线程，注意并不是给所有唤醒线程一个对象的锁，而是让它们竞争。

13、同步有几种实现方法,都是什么？

同步的实现方面有两种，分别是synchronized,wait与notify

14、线程的基本概念、线程的基本状态以及状态之间的关系？

线程指在程序执行过程中，能够执行程序代码的一个执行单位，每个程序至少都有一个线程，也就是程序本身。

Java中的线程有四种状态分别是：创建、就绪、运行、阻塞、结束。

15、简述synchronized和java.util.concurrent.locks.Lock的异同？

主要相同点：Lock能完成synchronized所实现的所有功能

主要不同点：Lock有比synchronized更精确的线程语义和更好的性能。synchronized会自动释放锁，而Lock一定要求程序员手工释放，并且必须在finally从句中释放。

16、Java为什么要引入线程机制，线程、程序、进程之间的关系是怎样的？

线程可以彼此独立的执行，它是一种实现并发机制的有效手段，可以同时使用多个线程来完成不同的任务，并且一般用户在使用多线程时并不考虑底层处理的细节。

程序是一段静态的代码，是软件执行的蓝本。

进程是程序的一次动态执行过程，即是处于运行过程中的程序。

线程是比进程更小的程序执行单位，一个进程可以启动多个线程同时运行，不同线程之间可以共享相同的内存区域和数据。

17、Runnable接口包括哪些抽象方法？Thread类有哪些主要域和方法？

Runnable接口中仅有run()抽象方法。

Thread类主要域有：MAX_PRIORITY,MIN_PRIORITY,NORM_PRIORITY。

主要方法有start(),run(),sleep(),currentThread(),setPriority(),getPriority(),join()等。

18、创建线程有哪两种方式(jdk5.0之前)？试写出每种的具体流程。比较两种创建方式的不同，哪个更优。

1—继承Thread类

1) 定义类继承Thread类。

2) 覆盖Thread类中的run方法。

3) 创建Thread子类对象，即创建了线程对象。

4) 调用线程对象start方法：启动线程，调用run方法。

2—实现Runnable接口

1) 定义类，实现Runnable接口。

2) 覆盖Runnable接口中的run方法。

3) 通过Thread类建立线程对象。

4) 将Runnable接口的子类对象作为实际参数传递给Thread类的构造方法中。

5) 调用Thread类的start方法：开启线程，调用Runnable子类接口的run方法。

【区别】

继承Thread: 线程代码存放Thread子类run方法中。

实现Runnable: 线程代码存在接口的子类的run方法。

【实现方法的好处】

1) 避免了单继承的局限性

2) 多个线程可以共享同一个接口子类的对象，非常适合多个相同线程来处理同一份资源。

19、 判断题

C 和 Java 都是多线程语言？

答案：错误

知识点：C 是单线程语言

如果线程死亡，它便不能运行？

答案：正确

知识点：线程死亡就意味着它不能运行。

在 Java 中，高优先级的可运行线程会抢占低优先级线程？

答案：正确

知识点：线程优先级的使用。

程序开发者必须创建一个线程去管理内存的分配？

答案：错误

知识点：Java 提供了一个系统线程来管理内存的分配。

一个线程在调用它的 start 方法，之前，该线程将一直处于出生期？

答案：正确

知识点：出生期的概念

当调用一个正在进行线程的 stop()方法时，该线程便会进入休眠状态？

答案：错误

知识点：应该是 sleep 方法。

如果线程的 run 方法执行结束或抛出一个不能捕获的例外，线程便进入等待状态？

答案：错误

知识点：如果线程的 run 方法执行结束或抛出一个不能捕获的例外，线程便进入死亡状态。

一个线程可以调用 yield 方法使其他线程有机会运行？

答案：正确

知识点：yield 方法总是让高优先级的就绪线程先运行。

20、选择题

Java 语言中提供了一个__线程，自动回收动态分配的内存。

A 异步

B 消费者

C 守护

D 垃圾收集

答案：D

知识点：垃圾线程的使用。

当__方法终止时，能使线程进入死亡状态。

A run

B setPriority

C yield

D sleep

答案：A

知识点：run 方法的使用。

用__方法可以改变线程的优先级。

A run

B setPriority

C yield

D sleep

答案：B

知识点：setPriority 方法的使用。

线程通过__方法可以使具有相同优先级线程获得处理器。

A run

B setPriority

C yield

D sleep

答案：C

知识点：yield 方法的使用。

线程通过__方法可以休眠一段时间，然后恢复运行。

A run

B setPriority

C yield

D sleep

答案：D

知识点：sleep 方法的使用。

__方法使对象等待队列的第一个线程进入就绪状态。

A run

B notify

C yield

D sleep

答案：B

知识点：notify 方法的使用。

方法 resume()负责重新开始__线程的执行。

A 被 stop()方法停止

B 被 sleep()方法停止

C 被 wait()方法停止

D 被 suspend()方法停止

答案：D

知识点：一个线程被用 suspend()方法，将该线程挂起。并通过调用 resume()方法来重新开始线程的执行。

但是该方法容易导致死锁，应尽量避免使用。

__方法可以用来暂时停止当前线程的运行。

A stop()

B sleep()

C wait()

D suspend()

答案：BCD

知识点：当调用 stop()方法后，当前的线程不能重新开始运行