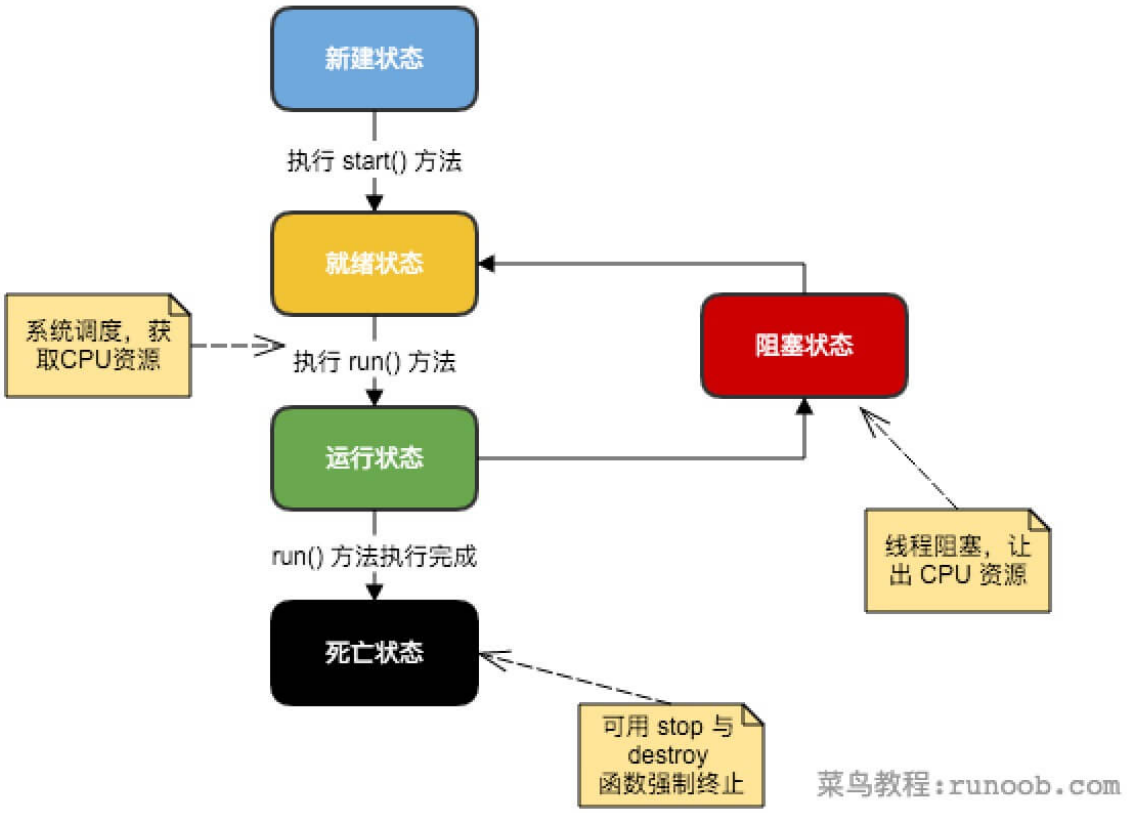
- 面试必问（多线程）

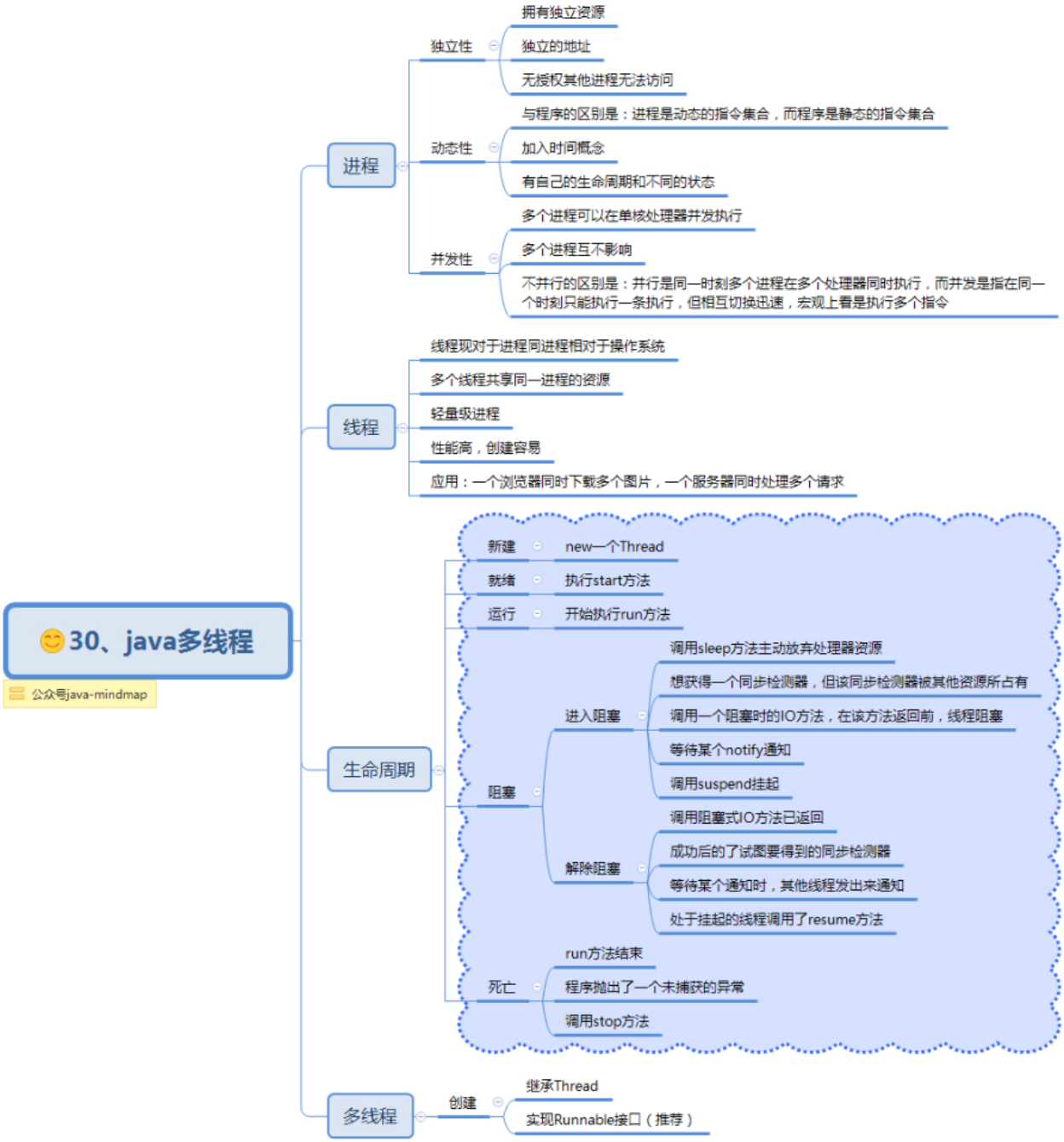
一个线程的生命周期



> - **新建状态**: 使用**new**关键字和**Thread类或其子类**建立一个线程对象后，该线程对象就处于新建状态。它保持这个状态直到程序 start() 这个线程。
> - **就绪状态**: 当线程对象调用了Start()方法后，该线程就进入了就绪状态，处于线程就绪队列中，等待JVM的线程调度器调度。
> - **运行状态**：如果就绪状态的线程获取CPU资源，就可以执行run（）方法了，此时线程就处于运行状态，此时的线程状态最为复杂，他可以变成死亡状态、阻塞状态，就绪状态。
> - **阻塞状态**: 如果一个线程执行了sleep（睡眠）或者suspend（挂起），等方法，那么线程失去占用资源后，就从运行状态进入了阻塞状态。在睡眠时间已到或获得设备资源后可以重新进入就绪状态。可以分为三种：
>   - **等待阻塞**: 运行状态中的线程执行 wait() 方法，使线程进入到等待阻塞状态。
>   - **同步阻塞**: 线程在获取 synchronized 同步锁失败(因为同步锁被其他线程占用)。
>   - **其他阻塞**：通过调用线程的 sleep() 或 join() 发出了 I/O 请求时，线程就会进入到阻塞状态。当sleep() 状态超时，join() 等待线程终止或超时，或者 I/O 处理完毕，线程重新转入就绪状态。
> - **死亡状态**：一个运行状态的线程完成任务或者其他终止条件发生时，该线程就切换到终止状态。

java中的进程（线程依赖于进程而存在）

- 进程：一个进程包括由操作系统分配的内存空间，包含一个或多个线程。一个线程不能独立的存在，它必须是进程的一部分。一个进程一直运行，直到所有的**非守护线程**都结束运行后才能结束。
    - 进程，就是指正在运行的程序称为一个进程，具有独立的内存空间和系统资源。
    - 单进程：指计算机只可以执行一个程序，当然现在的计算机都是是多进程的。（同理而言）
    - 多进程;：即在同一个时间段内执行多个程序，进程越多cpu占用越大



java 多线程编程（优先级越高，cup占用越大，同一个线程不能被多次启动）

- 定义：java 给多线程编程提供了内置的支持。一条线程指的是进程中一个单一顺序的控制流，一个进程中可以并发多个线程，每条线程并行执行不同的任务。
- 多线程是多任务的一种特别形式，但是线程使用了更小的资源开销
- 多线程能够满足程序员编写高效率的程序来达到充分利用CPU的目的,线程是CPU调度和任务

分配的基本单位，是程序中最小的单位，因此正在占用CPU的是线程。
- 单线程：如果程序在执行过程中只有一条执行路径，那么该程序为单线程执行，多条路径则为多线程。
- 多线程具有随机性，（下载电视、音乐等），为了增加使用率，可以多开几个线程，多线程会抢占CPU的资源。多线程的意义在于提高应用程序的使用效率（不会提高程序的执行速度，但会提高使用cup资源的概率），当然者具有随机性，我们无法确保那个进程或者线程优先占用cup。

如何实现多线程：java提供了一个类来通过调用c/c++来调用系统功能来创建一个类。

- 使用线程的前提是必须有一个进程。因为线程依赖进程而存在，因此首先我们需要系统功能创建一个进程，==而 java不能调用系统功能直接实现多线程程序,但是java提供了一个类库**Tread**,他既可以被继承有可以被实现，所以我们通过java调用c/c++，然后由c/c++去调用系统功能创建进程，并提供一些类，从而实现多线程程序。==
- 这个类位于java.lang.Tread类：两种方式：
  - 1、继承Tread类 步骤：自定义类并继承Tread类---->自定义类需要重写run（）方法---创建对象--->start（）启动线程 注意：如果直接调用run()方法，相当于普通方法。
  - 2、方式二：实现java。lang.runable接口

线程的优先级

- 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==（Thread.MIN_PRIORITY ） - ==10==（Thread.MAX_PRIORITY ）==。**默认**情况下，每一个线程都会分配一个优先级 ==NORM_PRIORITY（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。

创建线程(三种方式)

- 1、实现Runnable接口；
- 2、继承Tread类本身；
- 3 、通过callable和Future创建线程。

1、通过实现 Runnable 接口来创建线程:

- 创建一个线程最简单的方式就是实现一个Runnable接口，只需要调用run()方法
  - 重写该方法，重要的是理解的run()可以调用其他方法使用其他类，并声明变量，就像主线程一样。
  - 在创建一个实现 Runnable 接口的类之后，你可以在类中实例化一个线程对象。Thread 定义了几个构造方法，下面的这个是我们经常使用的：

```
Thread(Runnable threadOb,String threadName);
这里，threadOb 是一个实现Runnable接口的类的实例,并且threadName指定新线程的名字。
新线程创建之后，你调用它的 start() 方法它才会运行。
```

- 下面是实现Runnable线程实例

```
class RunnableDemo implements Runnable(){
```

```
    private Tread t;
    private String threadName;

    //构造方法
    RunnableDemo(String name){
        threadName = name;
        sysout("Creating"+threadName);
    }
    //重写run () 方法
    public void run(){
        sysout("Running"+ threadName);
        try{
            for(int i=4;i>0;i--){
                sysout("Thread:"+threadName+","+i)
                // 调用sleep () 方法，让线程睡一会儿
                sleep(50);
            }
     }catch(InterruptedException e){
          System.out.println("Thread " +  threadName + " interrupted.");
        }
          System.out.println("Thread " +  threadName + " exiting.");
    }
    public void start(){
        sysout("starting"+threadName);
        if(t==null){
            t= new Thread(this,threadName);
            t.start();
        }
    }
}
//测试
public class TestThread{
    main(){
        RunnableDemo r1 = new RunnableDemo(Thread-1);
        r1.start();
        RunnableDemo r2 = new RunnableDemo(Thread-2)
        r2.start();
    }
}
```

2、通过继承Thread来创建线程

> - 创建一个线程的第二种方法是创建一个新的类，该类继承 Thread 类，然后创建一个该类的实例。
> - 继承类必须重写 run() 方法，该方法是新线程的入口点。它也必须调用 start() 方法才能执行。本质上也是实现了 Runnable 接口的一个实例

```
public class ThreadDemo extends Thread{
    private Thread t;  //成员变量t，线程类。、
```

```
    private String threadName;

    //构造方法
    ThreadDemo(String name ){
        threadName = name;
        sysout("Creating"+threadName);
    }
    //重写run方法
    public void run(){
        sysout("Running"+threadName);
        try{
         for(int i=4;i>0;i--){
            //输出正在执行的线程
            sysout("Thread"+threadName+","+i);
            //让线程sleep一会儿
            Thread.sleep(50);
         }
        }catch (InterruptedException e) {
            sysout("Thread " +  threadName + " interrupted.");
        }
        sysout("Thread " +  threadName + " exiting.");
    }
   //线程启动方法
   public void start(){
       sysout("Starting"+threadName);
       t.start();
    }
  }
}
//测试类:
public class TestThread{
    main(){

    //创建两个线程
    ThreadDemo  T1 = new ThreadDemo("Thread-1");
    T1.start();
    ThreadDemo  T2 = new ThreadDemo("Thread-2");
    T2.start();
    }
}
```

Thread的方法

> - 下面方法是被Thread对象调用
>   - **public void start()**:使线程开始执行，java虚拟机调用该程序的run（）方法
>   - **public void run()**: 如果该线程是实现接口Runnable,那么可以直接调用runnable的run方法，否则，该方法不执行任何操作并返回。
>   - **public final void setName(String name)**: 改变线程的名称，使之与参数name相同。

    - **public final void setPriority(int priority)**: 改变线程的优先级
    - **public final void setDaemon(boolean on)**： 将线程标记为守护线程或者用户线程。
    - **public final void join(long millisec)**：等待该线程终止的时间最长为 millis 毫秒。
    - **public void interrupt()**： 中断线程。
    - **public final boolean isAlive()**：测试线程是否处于活动状态。
- Thread的静态方法
    - public static void yield() :暂停当前正在执行的线程对象，并执行其他线程。
    - public static void sleep(long millisec) :在指定毫秒内让当前正在执行的线程休眠（暂停执行），此操作受到系统计时器和调度程序的精确度和准确性的影响。
    - public static boolean holdsLock(Object x) :当且仅当当前线程在指定的对象上保持监视器锁时，才返回true
    - public static Thread currentThread() : 返回当前正在执行的线程对象的引用。
    - public static void dumpStack() :将当前线程的堆栈跟踪打印至标准错误流。

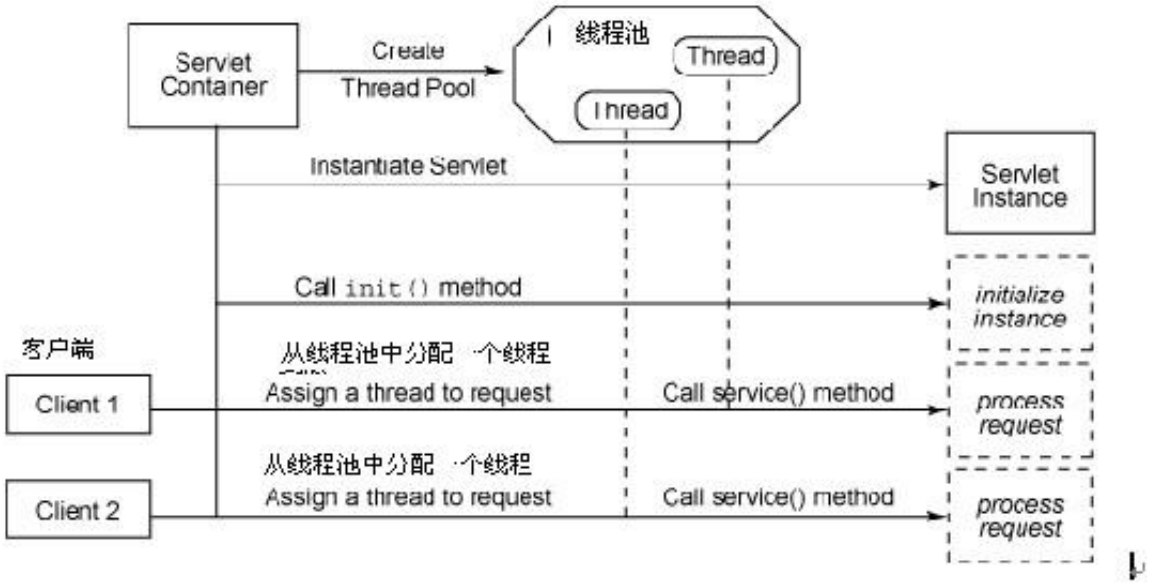
```
// 文件名 : DisplayMessage.java
// 通过实现 Runnable 接口创建线程
 public class DisplayMessage implements Runnable(){
     private String message;
     public DisplayMessage(String message){
         this.message = message;
     }
     public void run(){
         while(true){
             Sysout(message);
         }
     }
 }
```

```
// 文件名 : GuessANumber.java
// 通过继承 Thread 类创建线程
public class GuessANumber extends Thread {
    private int number;
    public GuessANumber(int number){
        this.number = number;
    }
    public void run(){
        int counter = 0;
        int guess = 0;
        do {
            guess = (int)(Math.random()*100+1);
            sysout(this.getName()+"guesses"+guess);
            counter++;
        }while(guess != number);
        sysout("**Correct!"+this.getName()+"in"+counter+"guesses.**");
    }
    }
```

```
// 文件名 : ThreadClassDemo.java 测试
//hread.setDaemon(true)必须在thread.start()之前设置，否则会跑出一个
IllegalThreadStateException异常。你不能把正在运行的常规线程设置为守护线程。 守护线程为用
户线程服务
public class ThreadClassDemio{
    main(){
        Runnable hello = new DisplayMessage("hello");
        Thread thread1 = new Thread(hello);     //接口实例化实现的是他的子类。
        thread1.setDaemon(true);  //thread1标记为守护线程
        thread1.setName("hello");
        sysout(starting hello thread...);
        thread1.start();
        Runnable bye = new DisplayMessage("goodbye");
        Thread thread2 = new Thread(bye
        thread2.setPriority(Thread.MIN_PRIORITY) ; // 设置线程的优先级为最低。
        thread2.setDaemon(true);  // 将线程标记为守护线程
        sysout("Starting  goodbye thread...");
        thread2.start();
        sysout("Starting  thread3...");
        Thread thread3 = new GuessANumber(27);
        thread3.start();
        try{
         thread3.join(); //  等待该线程终止的时间最长为 millis 毫秒
        }catch(InterruptedException e){
           sysout("Thread intterrupted.");
        }
       sysout("Starting  thread4...")
       Thread thread4 = new GuessaNumber(75);
       thread4.start();
       sysout(main() is ending...);
    }
}
```

---

web服务器中，容器启动时后台初始化一个服务线程。



通过 Callable 和 Future 创建线程

- 创建 Callable 接口的实现类，并实现 call() 方法，该 call() 方法将作为线程执行体，并且有返回值。
- 创建 Callable 实现类的实例，使用 FutureTask 类来包装 Callable 对象，该 FutureTask 对象封装了该 Callable 对象的 call() 方法的返回值
- 使用 FutureTask 对象作为 Thread 对象的 target 创建并启动新线程。
- 调用 FutureTask 对象的 get() 方法来获得子线程执行结束后的返回值。

```
public class CallableThreadTest implements Callable<Integer>{
    main(){
        CallableThreadTest ctt = new CallableThreadTest();  //接口callable的实现类
        FutureTask<Integer>  ft = new FutureTest<>(ctt);
        for(int i=0;i<100;i++){
            sysout(Thread.currentThread().getName()+"为当前线程的名字，他的循环变量i="+i);
            if(i==20){
                //创建一个有返回值的线程
                new Thread(ft,"有返回值的线程".start());
            }
        }
        try{
           sysout("子线程的返回值：“+ft.get());
        }catch (InterruptedException e) {

             e.printStackTrace();
        }
    }
    public Integer call() throws Exception  // @Override
    {
    int i = 0;
```

```
    for(;i<100;i++){
       System.out.println(Thread.currentThread().getName()+" "+i);
    }
    return i;
 }
}
```

创建线程的三种方式的对比

> - 采用实现Runnable和callable时，线程类只是实现类接口，还可以继承其他类。
> - 继承Thread类，编写简单，如果访问当前线程时，无需使用Thread.currentThread()方法，直接使用this就可以获得当前线程。

几个问题

> - **后台线程**：
>   - public final void setDaemon(boolean on):标记是否为守护线程，守护线程为用户线程服务，且必须在用户线程启动前调用，如果用户线程全部退出，只剩下守护线程时，JVM退出。
> - **为何是run（）方法呢?**
>   - 因为创建的类中只有部分代码需要被多线程执行，为了便于区分，Thread类中用来包含那些被线程执行的代码,就是说要被线程执行的代码放到run方法里面。
> - **为什么直接调用run方法是单线程呢?** ：
>   - run方法直接调用相当于普通方法，因此是单线程.
> - **run()和start()方法的区别**：
>   - run方法里面放的是线程需要执行的普通代码块，直接调用相当于普通方法。
>   - start方法首先启动线程，然后由JVM去调用该线程的run方法。
> - **多线程并非将start()调用多次，这样会出现非法线程状态异常**:
>   - 多次调用start（）方法，这相当于同一个线程启动了多次，然而这并不是多线程，只用创建多个对象后使用start（）方法，才能是多线程。
> - **如何设置和获取线程名称?**
>   - public final String getName():获取线程名称。
>   - public static Thread currentThread():返回当前正在执行的线程对象 ,然后输出时： ---->System.out.println(Thread.currentThread().getName());
>   - 设置线程对象的名称有两种方法
>   - public final void setName(String name):设置线程名称
>   - 使用有参构造方法设置线程名称

> - **什么是java程序运行原理?**
>   - java虚拟机启动的时候是一个多线程，主线程启动的时候同时启动垃圾回收线程。因为容易出现内存溢出。
>   - java虚拟机启动后，相当于启动了一个程序，也就是一个进程，该进程会启动一个“主线程”然后主线程会去调用一个类的main方法，所以main方法运行在主线程。在此之前的所有程序都是单线程。

> - **线程调度、优先级获取和设置**: 优先级高仅仅表示他获得的CPU几率大。

- ==线程调度的 两种模型==(java 使用的是抢占调度模型，取决于线程本身的优先级)
  - 分时调度：程序轮流使用cpu资源。
  - 抢占调度模型:取决于线程的优先级，优先级相同，**会随机选择一个执行**，优先级高的获取cpu的机会多一点，但是不一定先完成。

- **如何设置和获取线程的优先级?**
  - public final int getPriority(): 获取优先级
  - public final int setPriority(); 设置优先级（优先级1--10，默认为5）

- **线程的控制**：
  - ==休眠==： public static void **sleep**(long millis);在指定毫秒数内让当前的线程休眠（暂停执行），此操作受到系统计时器和调度程序精度和准确性的影响
  - ==加入==： public void join（）：等待该线程终止，即当前执行线程终止后别人才能执行
  - ==礼让==： public static void yield();暂停当前正在执行的的线程对象，并执行其他线程。
  - ==终止==：
    - public final void stop(): //不过这个方法现在已经过时，他会直接将线程强行终止，且无异常抛出，后续程序无法再执行。这会造成线程死锁。
    - public void interrupt()，//如果当前线程没有中断它自己（这在任何情况下都是允许的），则该线程的 checkAccess 方法就会被调用，这可能抛出 SecurityException。

```
public class ThreadStop extends Thread{
public void run(){
   System.out.println(“开始执行：”+new Date());
   try{     //亲爱的女神，我需要休息5秒钟，请不要打扰我
      Thread.sleep(50000000000000L);
}catch(InterruptedException e){
      System.out.println(“线程终止了”);
}
System.out.println(“结束执行：”+new Date());
}
}
public class ThreadStopDemo{
   public static void main(String[] args){
      ThreadStop ts = new ThreadStop();
      ts.start();
      Thread.sleep(3000); //你超过3s不醒来，我就打死你。
      ts.stop();//此方法已过时，并且比较暴力，不建议使用
}
}
```

- **礼让（yield）和休眠(sleep)的区别**：
  - yield（礼让）只是让当前线程重新回到可执行状态，所以执行yield（）后，线程可能会马上又被执行，只能是同优先级的线程有执行的机会。==同样, yield()也不会释放锁

资源==。
    - sleep()可以使优先级低的线程得到执行的机会,而yield()与同级不同级优先级的线程都有执行的机会。

线程的并发和并行以及如何处理。

- **并行**：逻辑上同时发生，在==某一个时间段内同时==在线多个程序。
- **并发**：物理上同时发生，在==某一个时间点同时==运行多个程序。

1、线程案例分析之继承Thread 类的方式卖电影票

- ++恒大影城目前正在热映《比得兔》，共有100张票，而他有2个售票窗口售票， 请设计一个程序模拟电影院售票++
- 实现Runnable 接口
    - 重写run()方法
    - 创建MyRunnable 类的对象
    - 创建Thread类的对象，并把上一步骤的对象作为构造参数传递(可以直接在里面实例化MyRunnable类)
- 继承Tread类
    - 自定义类X，然后继承Tread类
    - 在类X,中重写run()方法
    - 创建X，类的对象
    - start()启动线程对象。

问题：卖票的功能感觉已实现，但貌似会出现卖同票或负票的问题?

```
public class SellTicket extends Thread{
    private static int ticket =100;  //创建存储10张电影票的变量，且将其定义为成员变量，static为使多个线程共享100张电影票，
    public void run(){
       while (true) {
         if(ticket>0){
             sysout(getaName()+"正在出售第"+ (ticket--) +"张票");
         }
       }

    }
public class SellTiketDemo{
    main(){
        SellTicket st1 = new SellTicket();
        SellTicket st2 = new SellTicket();
        //设置线程名称
        st1.setName("窗口1");
        st2.setName("窗口2");
    }
  }
}
```

- 多线程(**买电影票出现了同票和负数票的原因分析**)：

- 出售同票问题: cpu的一次执行具有原子性
- 出售负票问题:线程的随机或延迟性

- 多线程(同步代码块的方式解决线程安全问题):
  - 我们发现三个窗口共享100张票，想法设法把共享内容包起来，此时java提供了同步机制。
- 多线程(同步的特点及好处和弊端):
  - 多个线程
  - 多个线程共享同一个锁
  - 解决了安全问题(优点)
  - 当线程相当多时，其运行效率较低(缺点)

同步代码块:

```
synchronized (对象) {
    //需要同步的代码
}
```

A:对象是什么呢?

可尝试任意创建一个对象(new Object())，还是未解决问题，试想2个窗口需要几把锁，一定是一把锁，否则失去意义。所以需要在线程类中创建锁的对象，让2个窗口共享一把锁。Object obj = new Object();

B:需要同步的代码是哪些呢?

把共享数据的代码包起来

代码实现:

```
public class SellTicket extends Thread{
     private int ticket = 100;//定义用于存储100张票的变量
     private Object obj = new Object();
     public void run(){
        while(true){
      synchronized(){
            if(ticket>0){
               try{
                   Thread.sleep(1000);
             }catch(InterruptedException e){
                  e.printStackTrace();
             }
               System.out.println(Thread.currentThread().getName()+"正在出售第"+
(ticket--)+"张票");
              }
            }
           }
         }
public class SellTicketDemo{
    public static void main(String[] args){
        SellTicket st1= new SellTicket();
        SellTicket st2= new SellTicket();
        //设置线程名称
        st1.setName("窗口1");
        st2.setName("窗口2");
```

```
        //启动线程
        st1.start();
        st2.start();
}
}
}
```

- 多线程(**同步代码、方法的应用和锁的问题**):
  - 同步代码块的同步锁对象可以是任意一个对象。
  - 如果一个方法中其方法体中的内容都被同步了，那么就可以直接将此方法变为同步方法。
  - 语法：将同步关键字加在方法上，然后删除同步代码块的内容。
  - 同步方法的锁是谁呢?
    - this
  - 静态同步方法的锁是谁呢?
    - 一定不会是this，因为static与this无关，其随着类的加载而加载。当前类的class文件（反射会讲）

- 多线程(**JDK5之后的Lock锁的概述和使用**):
  - 为了更清晰的让我们看到在哪里加的锁，哪里开的锁
  - Java 中提供了Lock接口：
    - public void lock()；获取锁
    - publc void unlock();释放锁
    - 实现类：ReentrantLock

- 多线程(**死锁问题概述和使用**):
  - 同步弊端：效率低,如果出现同步嵌套，会出现死锁问题
  - 死锁问题及其代码：是指两个或两个以上的线程在执行的过程中，因争夺资源产生的一种互相等待现象。

举例：
    中美两国人吃饭案例。
    正常情况：
        中国人：筷子两支
        美国人：刀和叉
    现有状况：
        中国人：筷子一支，刀一把
        美国人：筷子一支，叉一把

```
/**
* 一个简单的死锁类
* 当DeadLock类的对象flag==1时（td1），先锁定t1,睡眠500毫秒
* 而td1在睡眠的时候另一个flag==0的对象（td2）线程启动，先锁定o2,睡眠500毫秒
* td1睡眠结束后需要锁定t2才能继续执行，而此时t2已被td2锁定；
```

```
* td2睡眠结束后需要锁定t1才能继续执行，而此时t1已被td1锁定；
* td1、td2相互等待，都需要得到对方锁定的资源才能继续执行，从而死锁。
*/
public class DeadLock implements Runnable{
    private boolean flag;
    public DeadLock(boolean flag){
        this.flag = flag;
    }
  //静态对象是类的所有对象共享的
  private static Object t1 = new Object(),t2 = new Object();
   @Override
   public void run() {
        System.out.println("flag=" + flag);
    if(flag){
        synchronized(t1){
            try{
                Thread.sleep(500);
            }catch (Exception e) {
               e.printStackTrace();
            }
            synchronized(t2){
                e.printStackTrace();
            }
        }
    }else{
        synchronized(t2){
            try{
                Thread.sleep(500);
            }catch(Exception e){
                e.printStackTrace();
            }
             synchronized (o1) {
                    System.out.println("0");
            }
        }
    }
 public static void main(String[] args){
     DeadLock td1 = new DeadLock(true);
     DeadLock td2 = new DeadLock(false);
      //td1,td2都处于可执行状态，但JVM线程调度先执行哪个线程是不确定的。
    //td2的run()可能在td1的run()之前运行
    new Thread(td1).start();
    new Thread(td2).start();
    }
}
```

- 多线程(如**何解决死锁问题-线程通信**)：\

生产者和消费者的问题:不同种类的线程操作同一个资源

代码实现：

```
     共享资源类：SharedData
     生产者：Producer
     消费者：Consumer
     测试类：CommunicationDemo

     Public class SharedData{
    String name;
    Double price;
    }
    public class Producer implements Runnable{

    public void run(){
        SharedData s = new SharedData();
        s.name = “武大郎”;
        s.price = 5.0;
}
}
//消费者
public class Consumer implements Runnable{
    public void run(){
        Student s = new Student();
        System.out.println(s.name+”------”+s.price);
}
}
//测试类
public class SharedDataDemo{
     public static void main(String[] args){
         Producer p = new Producer();
         Consumer c = new Consumer();

         Thread th1 = new Thread(p);
         Thread th2 = new Thread(c);

         th1.start();
         th2.start();
}
}
```

- 问题1：看着思路写代码，出现了数据为null null的情况
    - 原因：我们在每个线程中都创建了新的资源，而我们要求设置和获取的是共享资源
- 如何实现呢?
    - 在外界把这个数据创建出来，通过构造方法传递给其他的类。

```
public class Producer implements Runnable{
    private SharedData s;
    public Producer(SharedData s){
       this.s = s;
```

```
}
public void run(){
        //SharedData s = new SharedData();
        s.name = “武大郎”;
        s.price = 5.0;
}
}
//消费者
public class Consumer implements Runnable{
    private SharedData s;
    public Consumer (SharedData s){
       this.s = s;
}

    public void run( ){
        //Student s = new Student();
        System.out.println(s.name+”------”+s.price);
}
}
    public class SharedDataDemo{
     public static void main(String[] args){
         SharedData s = new SharedData();
         Producer p = new Producer(s);
         Consumer c = new Consumer(s);

         Thread th1 = new Thread(p);
         Thread th2 = new Thread(c);

         th1.start();
         th2.start();
}
}
```

- 问题，出现了null-----5.0

```
问题：为了数据多效果好一些，加入了循环，给出不同的值，这个时候产生了新的问题
    A：同一个数据出现多次
    B：数据不匹配
原因：
    A:同一数据出现多次
       获取CPU一点点执行权，就足够执行多次
   B:数据不匹配
       线程的随机性
public class Producer implements Runnable{
    private SharedData s;
    private int I = 0;
    public Producer(SharedData s){
       this.s = s;
}
```

```
    public void run(){
       while(true){
           if(i%2==0){
              s.name = “武大郎”;
              s.price = 5.0;
}else{
              s.name = “袁记”;
              s.price = 5.0;

}
I++;
}
}
}
```

- **线程安全问题**：

```
线程安全问题：
    A:是否是多线程坏境
    B:是否有共享数据
    C:是否有多条语句操作共享数据
解决方案：线程同步，即加锁，
注意：A:不同种类的线程都要加锁   B:不同种类线程的锁必须是同一把锁。
public class Producer implements Runnable{
    private SharedData s;
    private int I = 0;
    public Producer(SharedData s){
       this.s = s;
}
    public void run(){
       while(true){
          //加锁
           if(i%2==0){
              s.name = “武大郎”;
              s.price = 5.0;
}else{
              s.name = “袁记”;
              s.price = 5.0;

}
}
}
}
public class Consumer implements Runnable{
    private SharedData s;
    public Consumer (SharedData s){
       this.s = s;
}
```

```
    public void run(){
            //加锁
        System.out.println(s.name+"------"+s.price);
}
}
```

- **等待唤醒机制**：如果消费者先抢到执行权，生产者还未生产，如何消费？如果生产者先抢到执行权，就会生产数据，但是产完数据后还继续拥有执行权，他又继续生产烧饼？没有消费者消费，生产将无意义。
  - 正常思路：
    - A:生产者：先看是否有数据，有就等待，没有就生产，生产完就通知消费者消费
    - B:消费者;先看是否有数据，有就消费，没有就等待，没有就通知生产者生产
- Object 类中提供了三个方法：
  - wait():等待
  - notify();唤醒单个线程
  - notifyAll()：唤醒所有线程
- 问题：**为什么这些方法不定义在Thread类中呢**?
  - 答案：这些方法的调用必须通过锁对象调用，而我们使用的锁对象是任意锁对象,所以，这些方法必须定义在Object类中

```
public class SharedData {
private String name;
private Double price;
        Boolean flag; //默认情况下表示没有数据 如果为true，表示有数据
}
```

```
public class Producer  implements Runnable{
    private SharedData s;
    private int i =0;
    public Producer(SharedData s){
     this.s = s;
    }
@Override
public void run() {
   while(true){
synchronized (s) {
               if(s.flag){
                  s.wait();//等待
}
   if(i%2==0){
        s.setName("武大郎烧饼");
        s.setPrice(5.0);
    }else{
        s.setName("王程烧饼");
```

```
            s.setPrice(3.0);
        }
            i++;
                        //此时修改标记
            s.flag = true;
                        //唤醒 s.notify();
            }
           }
        }
}
```

```
public class Consumer implements Runnable {
    private SharedData s;
    public Consumer(SharedData s){
     this.s = s;
    }
@Override
public void run() {
 while(true){
        synchronized (s) {
                        if(!s.flag){
                           s.wait();
        }
    System.out.println(s.getName()+"-------"+s.getPrice());
                        //修改标记
          s.flag = false;
                        //唤醒
          s.notify();
        }
     }
   }
}
```