多线程

今日学习内容：

- 线程的创建和启动
- 线程的生命周期
- 线程生命周期中的各个状态
- 操作线程的多个方法
- 线程的同步代码块
- 线程的同步方法

今日学习目标：

- 了解进程和线程的区别
- 认识主线程和后台线程
- 掌握使用继承方式编写多线程
- 掌握使用实现方式编写多线程
- 区分继承和实现方式编写多线程的优劣
- 掌握线程的生命周期图
- 掌握线程多个状态之间的转换关系
- 了解操作线程的方法
- 掌握使用同步代码块和同步方法实现线程的同步操作

24. 线程与并发

24.1. 理解进程和线程的区别（了解）

进程：是指一个内存中运行的应用程序（程序的一次运行就产生一个进程），每个进程都有自己独立的一块内存空间，比如在Windows的任务管理器中，一个运行的xx.exe就是一个进程。

多进程操作系统

假定 A,B,C 三个程序开始运行，A、B、C 产生进程 Pa、Pb、Pc。Pa、Pb、Pc 排队轮流使用 CPU。假设 Pa 先抢占到 CPU，Pa 执行，执行过程中 Pa 需要等待数据输入(I/O 操作(例如用户输入)、请求网络资源)， 此时 CPU 空转，为了提高 CPU 利用率，Pa 被切换出去，Pa 保存当前执行状态,Pa 挂起。Pb 抢占 CPU，Pb 开始执行，Pb 如果没有数据输入，Pb 也可能被切换出去(CPU 时间片到了)，Pb 挂起;Pc 抢占到 CPU，开始执行，如果 Pc 有数据输入，Pc 保存当前状态并挂起。此时 3 个进程都挂起，CPU 空闲。CPU 挑选一个进程运行，根据 CPU 执行原则，选中 Pb，Pb 继续执行。

CPU 通过时间片实现多任务，这样的操作系统称为多任务操作系统，但同一时刻还是只有一个进程执行。

并行和并发

并行： 同一时间点执行多个任务

并发：同一时间段中执行多个任务

线程：是指进程中的一个执行任务(控制单元)，一个进程中可以运行多个线程，多个线程可共享进程的数据。

线程的出现为了解决实时性问题。

线程是进程的细分，通常，在实时性操作系统中，进程会被划分为多个可以独立运行的子任务， 这些子任务被称为线程，多个线程配合完成一个进程的任务。

假设 P 进程抢占 CPU 后开始执行，此时如果 P 进行正在进行获取网络资源的操作时，用户进行 UI 操作，此时 P 进程不会响应 UI 操作。可以把 P 进程可以分为 Ta、Tb 两个线程。Ta 用于获取网络资源，Tb 用于响应 UI 操作。此时如果 Ta 正在执行获取网络资源时、用户进行 UI 操作， 为了做到实时性，Ta 线程暂时挂起，Tb 抢占 CPU 资源，执行 UI 操作，UI 操作执行完成后让出CPU，Ta 抢占 CPU 资源继续执行请求网络资源。

多线程：在同一个进程中并发运行的多个子任务。

一个进程至少有一个线程，为了提高CPU的效率，可以在一个进程中开启多个控制单元，这就是多线程。

24.2. 主线程 main（了解）

在运行一个简单的Java程序的时候，就已经存在了两个线程，一个是主线程，一个是后台线程——维护的垃圾回收。主线程很特殊，在启动VM的时候自动启动的。

如果一个进程没有任何线程，我们成为单线程应用程序；如果一个进程有多个线程存在， 我们成为多线程应用程序。进程执行时一定会有一个主线程(main 线程)存在，主线程有能力创建其他线程。

多个线程抢占CPU，导致程序的运行轨迹不确定。多线程的运行结果也不确定，多线程的程序复杂度提高很多。

24.3. 线程的创建和启动（掌握）

方式一，继承Thread类：

- 自定义类继承Thread
- 覆写run方法
- 创建自定义类对象
- 自定义类对象调用start方法

```
class MyThread extends Thread {

    public void run() {
        //线程体，线程启动时，会自动调用本方法，所有这里是我们写代码的主体部分
    }
}

public class ExceptionDemo {
    public static void main(String[] args) {
        MyThread t = new MyThread();
        t.start();// 调用Thread的start方法，JVM会自动调用run方法。
    }
}
```

方式二，实现Runnable 接口

- 自定义类实现Runnable接口
- 覆写run方法
- 创建自定义类对象
- 把自定类的对象作为Thread类构造器参数，并调用Thread对象start方法

```
class MyRunnable implements Runnable {
    public void run() {
        //线程体，线程启动时，会自动调用本方法，所有这里是我们写代码的主体部分
    }
}

public class ThreadDemo2 {
    public static void main(String[] args) {
        MyRunnable target = new MyRunnable();
        Thread t = new Thread(target);
        t.start();
    }
}
```

第一种使用起来方便，启动一个线程也方便，很多功能都在Thread类中定义好了；

第二种方式启动得依赖于Thread，因为本身Runnable中只有run方法，请看Thread的构造方法。后期的功能拓展有优势

24.3.1. 线程体-run方法（掌握）

不管哪种方式创建的线程，都得覆写run 方法，因为这是线程体方法，该方法在线程启动之后会自动被调用。

```
public void run() {
    //线程体，线程启动时，会自动调用本方法，所有这里是我们写代码的主体部分
}
```

线程的执行随机性：

一旦一个线程启动之后就是一个独立的线程，等待CPU的调度分配资源，不会因为启动它的外部线程结束而结束。

```
class MyThread extends Thread {
    public void run() {
        //自定义线程中的for循环打印i,打印顺序是完全随机的。
        for (int i = 0; i < 10; i++) {
            System.out.println("MyThread  ==> " + i);
        }
    }
}

public class Demo {
    public static void main(String[] args) {
        MyThread mt = new MyThread();
        mt.start();
        //主线程中的for循环打印i
        for (int i = 0; i < 10; i++) {
            System.out.println("main  ==> " + i);
        }
    }
}
```

多次运行该程序，观察每次运行的结果。

24.3.2. 线程的启动（掌握）

启动线程必须调用线程类Thread中的start方法，该方法应该由Thread类的一个实例来调用，下面是方法签名：

```
public void start()
```

底层会调用该线程的 run 方法。

只有调用了线程对象的start方法才会开启一个新的线程，如果是直接调用对象的run方法不会开启新的线程，只是一个单线程。

注意：启动一个新线程，不能使用run()方法，只能使用start方法。

24.3.2 继承方式VS实现方式（掌握）

当多线程并发访问同一个资源时，会导致线程出现安全性的原因，看案例。

案例：现有50张票，现在有三个窗口(A、B、C)卖这50张票。

```
因为A、B、C三个窗口可以同时卖票，此时得使用多线程技术来实现这个案例。
分析：可以定义三个线程对象，并启动线程.
第一步：每一个窗口买票的时候：展示自己买出一张票，
第二步：还剩xx张票
```

使用继承方式

```
public class TicketWindow extends Thread{

    private int count = 50;

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

    @Override
    public void run() {

        // 模拟10个人买票
        for( int i = 0;i < 10;i++ ){
            if( TicketThread.count > 0 ){
                count--;
                System.out.println(super.getName() + "卖出一张票，还剩" + count +
"张");
            }
        }
    }
}
-------------------------------------------------
public class Test01Ticket {
    public static void main(String[] args) {
        // 模拟买票过程。共有 5 张票，多线程模拟卖票的过程。
        TicketWindow ta = new TicketWindow("窗口A");
        TicketWindow tb = new TicketWindow("窗口B");
        TicketWindow tc = new TicketWindow("窗口C");
```

```
        ta.start();
        tb.start();
        tc.start();
    }
}
```

使用继承方式完成该案例的时候，会发现A、B、C都各自卖了50张票，为何?

使用实现方式

```
public class Ticket implements Runnable{

    private int count = 5;

    @Override
    public void run() {
        // 模拟10个人买票
        for( int i = 0;i < 10;i++ ){

            if( this.count > 0 ){
                count--;
                System.out.println(Thread.currentThread().getName() + "卖出一张票，
还剩" + count + "张");
            }

        }
    }
}

-------------------------------------------------
public class TestO1Ticket {
    public static void main(String[] args) {

        // 1>创建一个Runnable实现类
        Ticket ticket = new Ticket();

        // 2> 让myRun对象的线程体(run方法)跑在4个线程中
        Thread t1 = new Thread(ticket,"窗口A");
        Thread t2 = new Thread(ticket,"窗口B");
        Thread t3 = new Thread(ticket,"窗口C");

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

在使用实现方式的时候，我们发现A、B、C一共卖了50张票，为何?

通过买票案例，分析继承方式和实现方式的区别:

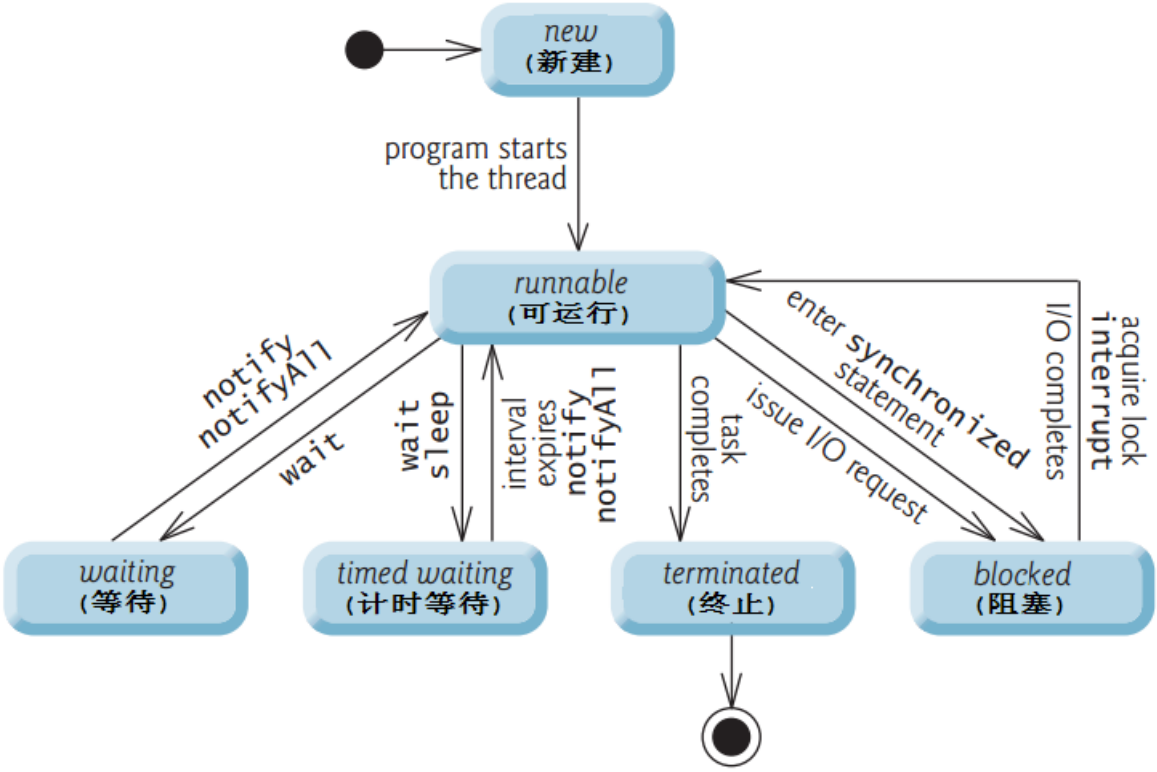
继承方式:

- Java中类是单继承的，如果继承了Thread了，该类就不能再有其他的直接父类了。
- 从操作上分析，继承方式更简单，获取线程名字也简单。
- 从多线程共享同一个资源上分析，继承方式不能多个线程共享同一个资源。

实现方式：

- Java中类可以多实现接口，此时该类还可以继承其他类，并且还可以实现其他接口（设计上,更优雅）。
- 从操作上分析，获取线程名字也比较复杂，得使用Thread.currentThread()来获取当前线程的引用。
- 从多线程共享同一个资源上分析，实现方式可以多线程共享同一个资源。

24.4. 线程生命周期和状态（掌握）



- 新生状态：用 new 关键字建立一个线程后，该线程对象就处于新生状态。
  处于新生状态的线程有自己的内存空间，通过调用 start()方法进入就绪状态。
- 就绪状态
  处于就绪状态线程具备了运行条件，但还没分配到 CPU，处于线程就绪队列，等待系统为其分配 CPU。当系统选定一个等待执行的线程后，它就会从就绪状态进入执行状态，该动作称为“CPU 调度”。
- 运行状态
  在运行状态的线程执行自己的run 方法中代码,直到等待某资源而阻塞或完成任务而死亡。如果在给定的时间片内没有执行结束，就会被系统给换下来回到等待执行状态（就绪）。
- 阻塞状态
  处于运行状态的线程在某些情况下，如执行了 sleep(睡眠)方法，或等待 I/O
  设备等资源，将让出 CPU 并暂时停止自己运行，进入阻塞状态。
  在阻塞状态的线程不能进入就绪队列。只有当引起阻塞的原因消除时，如睡眠时间已到，或等待的 I/O 设备空闲下来，线程便转入就绪状态，重新到就绪队列中排队等待，被系统选中后从原来停止的位置开始继续执行。
- 死亡状态
  死亡状态是线程生命周期中的最后一个阶段。线程死亡的原因有三个，一个是正常运行的线程完成了它的全部工作；二是线程抛出未捕获的Exception或Error，三是线程被强制性地终止，如通过 stop 方法来终止一个线程【易导致死锁，不推荐】

24.5. 操作线程的方法（掌握）

24.5.1. join方法（了解）

join方法的主要作用就是同步，它可以使得线程之间的并发执行变为串行执行。

比如在A线程中调用了B线程的join()方法时，表示只有当B线程执行完毕时，A线程才能继续执行。

```
public class JoinThread extends Thread{
    public JoinThread() {}
    public JoinThread(String name) {
        super(name);
    }

    @Override
    public void run() {

        for(int i = 0;i < 10;i++){
            System.out.println(super.getName() + i);
        }
    }
}
```

测试join方法

```
public class Test01Join {
    public static void main(String[] args) throws InterruptedException{

        JoinThread ta = new JoinThread("线程A");
        ta.start();

        // main thread
        for(int i = 0;i < 10;i++){
            if( i == 2 ){
                // t1强制执行直到执行结束。
                ta.join();
            }
            System.out.println("main:" + i);
        }
    }
}
```

线程A的join方法表示线程A的强制执行，其他线程都阻塞，直到线程A执行完成，其他线程才会被执行。

24.5.2.sleep方法（了解）

sleep方法让正在执行的线程暂停一段时间，进入阻塞状态，常常用来模拟网络延迟等。

```
sleep(long milllis) throws InterruptedException：毫秒为单位
```

调用sleep()后，在指定时间段之内，该线程不会获得执行的机会

```
public class SleepThread extends Thread{
    public SleepThread() {}
```

```
    public SleepThread(String name) {
        super(name);
    }

    @Override
    public void run() {
        System.out.println("线程即将执行");

        try {
            Thread.sleep(10000);
        } catch (InterruptedException e) {
            System.out.println("我被中断了");
        }

        System.out.println("线程完成");
    }
}
```

测试 sleep 方法

```
public class Test01Sleep {
    public static void main(String[] args) throws InterruptedException{

        SleepThread ta = new SleepThread("线程A");
        ta.start();

        System.out.println("主线程进入休眠5s");
        Thread.sleep(5000);
        System.out.println("5s后主线程自动唤醒");

        // 企图去中断ta线程
        // ta.interrupt();
    }
}
```

24.5.3. 线程的优先级（了解）

每个线程都有优先级，优先级的高低只和线程获得执行机会的次数多少有关。并不是说优先级高的就一定先执行，哪个线程的先运行取决于CPU的调度；

Thread对象的setPriority(int x)和getPriority()用来设置和获得优先级。

24.5.4. 后台线程（了解）

所谓后台线程，一般用于为其他线程提供服务。也称为守护线程。JVM的垃圾回收就是典型的后台线程。

特点：若所有的前台线程都死亡，后台线程自动死亡。

Thread对象setDaemon(true)用来设置后台线程。

setDaemon(true)必须在start()调用前，否则抛IllegalThreadStateException异常。

25. 线程安全性

25.2. 线程同步（掌握）

当多线程并发访问同一个资源对象的时候,可能出现线程不安全的问题。

但是，分析打印的结果，有时候发现没有问题：

意识：看不到问题，不代表没有问题，可能是我们经验不够，或者说问题出现的不够明显。

那么可以使用线程休眠来模拟网络延迟，让问题来得更明显一些：

```
Thread.sleep(10);//当前线程睡10毫秒,当前线程休息着,让其他线程去抢资源.
```

在程序中并不是使用Thread.sleep(10)之后程序才出现问题，而是使用之后，问题更明显，休眠的时间越久问题越明显，一般用10或100即可，具体根据情况而定。

```
public class MyRun implements Runnable {

    private int count = 50;

    @Override
    public void run() {
        // 模拟10个人买票
        for (int i = 0; i < 10; i++) {
            // 模拟询问过程
            try {
                Thread.sleep(1000);
            } catch (InterruptedException e) {
                e.printStackTrace();
            }

            if (this.count > 0) {
                count--;
                System.out.println(Thread.currentThread().getName() + "卖出一张票，
还剩" + count + "张");
            }
        }
    }
}
```

分析运行结果，为什么有错误的结果。

在这里，总数减1操作和打印输出剩余操作，应该是一个原子操作，也就说是一个不能分割的操作，两个步骤之间不能被其他线程插一脚。

对于原子性操作，要么都不执行，要么都执行完成，0% / 100%

```
第一步：count--;
第二步：System.out.println(Thread.currentThread().getName() + "卖出一张票，还剩" +
count + "张");
```

解决方案：保证票总数减1操作和打印输出剩余操作，必须同步完成。

解决思路：A线程获得同步锁进入操作的时候，B和C线程只能在外等着，A操作结束，释放同步锁。A和B和C才有机会去抢同步锁（谁获得同步锁，谁才能执行代码）。

通俗例子：A、B、C三个人去抢厕所的雅间，为了保证安全规定谁抢到了必须上锁，把其他人排除外雅间外面。若A抢到了，进入后应该立马上锁，B和C只能在外等着，当A释放锁出来的时候，A、B、C又开始尝试抢资源。

- 方式1：同步代码块
- 方式2：同步方法

25.2.1. 同步代码块（掌握）

同步代码块语法:

```
synchronized(同步锁){
     //需要同步操作的代码
}
```

同步锁，又称之为同步监听对象/同步锁/同步监听器/互斥锁：

为了保证每个线程都能正常执行原子操作，Java引入了线程同步机制。

对象的同步锁只是一个概念，可以想象为在对象上标记了一个锁。

Java程序允许使用任何对象作为同步监听对象，一般的，我们把当前并发访问的共享资源作为同步监听对象，比如此时三个线程的共同资源Ticket对象。

注意：在任何时候，最多允许一个线程拥有同步锁，谁拿到锁就执行，其他的线程只能在代码块外等着。

```
public class Ticket implements Runnable {

    private int count = 50;

    @Override
    public void run() {
        // 模拟10个人买票
        for (int i = 0; i < 10; i++) {

            synchronized (this) {
                // 业务上不可分割的逻辑单元
                try {
                    Thread.sleep(1000);
                } catch (InterruptedException e) {
                    e.printStackTrace();
                }

                if (this.count > 0) {
                    count--;

                    System.out.println(Thread.currentThread().getName() + "卖出一张票，还剩" + count + "张");
                }
            }

        }
    }
}
```

此时的同步锁this表示Ticket对象，而程序中Ticket对象只有一份，故可以作为同步锁。

25.2.2. 同步方法（掌握）

使用synchronized修饰的方法，就叫做同步方法。保证A线程执行该方法的时候，其他线程只能在方法外等着。

```
synchronized  public  void  doWork(){
     ///TODO
}
```

此时同步锁是谁——其实就是，调用当前同步方法的对象：

- 对于非static方法，同步锁就是this。
- 对于static方法，同步锁就是当前方法所在类的字节码对象。

```
class Ticket implements Runnable {
    private int count = 50;

    public void run() {
        for (int i = 0; i < 50; i++) {
            this.saleTicket();
        }
    }

    public synchronized void saleTicket(){
        // 业务上不可分割的逻辑单元
        try {
            Thread.sleep(1000);
        } catch (InterruptedException e) {
            e.printStackTrace();
        }

        if (this.count > 0) {
            count--;
            System.out.println(Thread.currentThread().getName() + "卖出一张票，还
剩" + count + "张");
        }
    }
}
```

25.2.3. synchronized的优劣（掌握）

好处：保证了多线程并发访问时的同步操作，避免线程的安全性问题。

缺点：使用synchronized的方法/代码块的性能要低一些。

建议：尽量减小synchronized的作用域。

面试题:

1. StringBuilder和StringBuffer的区别
2. 说说ArrayList和Vector的区别
3. HashMap和Hashtable的区别

通过源代码会发现，主要就是方法有没有使用synchronized的区别，比如StringBuilder和StringBuffer。

StringBuffer类

```
public synchronized StringBuffer append(Object obj) {
    super.append(String.valueOf(obj));
    return this;
}
```

StringBuilder类

```
public StringBuilder append(Object obj) {
    return append(String.valueOf(obj));
}
```

因此得出结论：使用synchronized修饰的方法性能较低，但是安全性较高，反之则反。