第十五章 线程和多线程

讲授思路

- 线程简介
- Java中创建多线程
- 多线程的同步和死锁

讲授思路-线程简介

- 程序、进程、线程
- 线程的应用场景
- 线程的生命周期

程序、进程、线程

- 程序、进程、线程
  - 程序是一段静态的代码，是应用软件执行的蓝本。
  - 进程是程序的一次动态执行过程，它对应了**从代码加载、执行至执行完毕的一个完整过程**，这个过程也是进程本身从产生、发展至消亡的过程。
  - 线程是比进程更小的执行单位。进程在其执行过程中，可以产生多个线程，形成多条执行线索，每条线索，即每个线程也有它自身的产生、存在和消亡的过程，也是一个动态的概念。

线程和进程的区别

进程

一个应用程序一个进程

独立功能的程序

通过多个线程占据系统资源

进程之间数据状态完全独立

线程

一个进程可以有多个线程

执行程序的最小单元

占用CPU的基本单位

线程间共享一块内存空间

Java中的线程

- Java中的线程
  - 每个Java程序都有一个默认的主线程
  - 当JVM加载代码发现main方法之后，就会立即启动一个线程，这个线程称为主线程
- 主线程的特点：
  - 是产生其他子线程的线程
  - 不一定是最后完成执行的线程

Java中的线程

- Java中的线程
  - 如果main方法中没有创建其他的线程，那么当main方法执行完最后一个语句，JVM就会结束Java应用程序——单线程。
  - 如果main方法中又创建了其他线程，那么JVM就要在主线程和其他线程之间轮流切换，JVM要等程序中所有线程都结束之后才结束程序——多线程。

多线程的优势

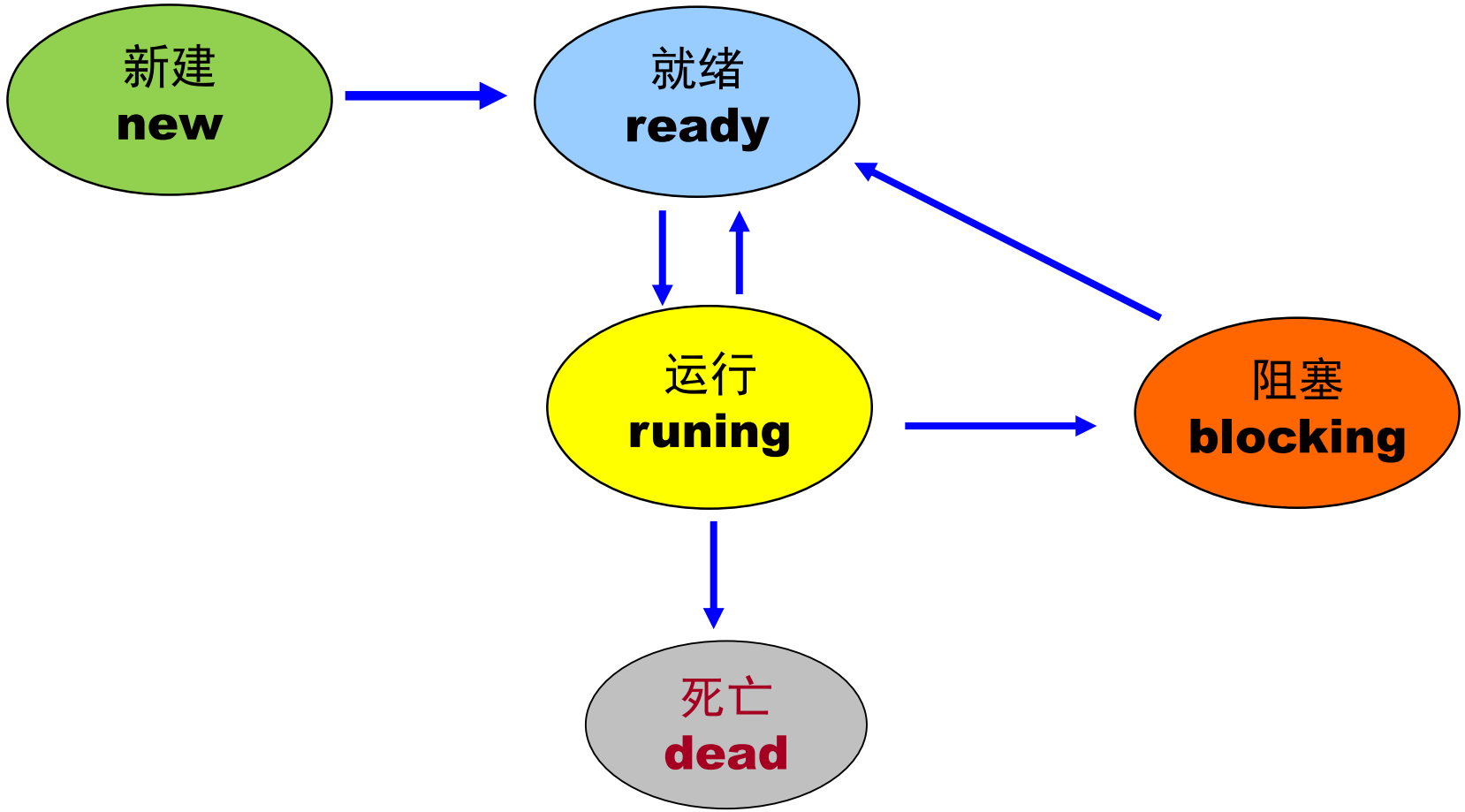
- 多线程的优势:
  - 减轻编写交互频繁、涉及面多的程序的困难。
  - 程序的吞吐量会得到改善。
  - 由多个处理器的系统，可以并发运行不同的线程.(否则,任何时刻只有一个线程在运行)
  - “同时”执行是人的感觉，在线程之间实际上轮换执行。

多线程的应用场景

- 多线程什么场合下被使用呢？
  - 想要同时处理多件事：单线程处理不了的，必须使用多线程。（类似于分身术）
  - 多个线程分解大任务：用单线程可以做，但是使用多线程可以更快。（类似于左右开弓）

线程的生命周期

- 线程完整的生命周期包括五个状态：新建、就绪、运行、阻塞和死亡。



线程状态介绍

- 新建状态：线程对象**已经创建**，还没有在其上调用start()方法。
- 可运行状态：当线程调用start方法，但调度程序还**没有把它选定为运行线程**时线程所处的状态。
- 运行状态：线程调度程序从可运行池中选择一个线程作为**当前线程**时线程所处的状态。这也是线程进入运行状态的唯一方式。

线程状态介绍

- 等待/阻塞/睡眠状态：其共同点是：线程仍旧是活的，但是当前**没有条件运行**。它是可运行的，当某件事件出现，他可能返回到可运行状态。
- 死亡状态：当线程的**run()方法完成**时就认为它死去。线程一旦死亡，就不能复生。 一个死去的线程上调用start()方法，会抛出java.lang.IllegalThreadStateException异常。

讲授思路-Java中创建多线程

- 定义任务
- 线程中的常用方法
- 线程的优先级

定义任务

- Java中两种创建线程的方式：
  - 继承Thread类
    - 重写run() 方法
    - new一个线程对象
    - 调用对象的 start() 启动线程
  - 实现Runnable接口
    - 实现run() 方法
    - 创建一个Runnable类的对象r，new MyRunnable()
    - 创建Thread类对象并将Runnable对象作为参数，new Thread(r)
    - 调用Thread对象的start()启动线程

创建多线程示例

继承Thread类

```
class Handler extends Thread{
    public void  run(){
      //方法重写
    }
    public static void main(String[] args){
        Thread thread = new Handler();
        //创建线程对象
        thread.start();//启动线程
    }
}
```

实现Runnable接口

```
class Handler implements Runnable{
    public void  run(){
      //方法实现
    }
    public static void main(String[] args){
        Handler handler = new Handler();
        Thread thread = new Thread(handler);
        //创建线程对象
        thread.start();//启动线程
    }
}
```

创建线程的方式对比

- 继承Thread类实现多线程
  - 优点：编写简单，如果需要访问当前线程直接使用this即可获得当前线程.
  - 缺点：因为线程类已经继承了Thread类，不能再继承其他的父类.
- 实现Runnable接口
  - 优点：线程类只实现了Runable接口，还可以继承其他的类. 这种方式可以多个线程共享同一个目标(target)对象，非常适合多个相同线程来处理同一份资源的情况，从而可以将代码和数据分开，形成清晰的模型，较好地体现了面向对象的思想.
  - 缺点：编程稍微复杂，如果需要访问当前线程，必须使用Thread.currentThread()方法。

线程创建的问题

- 线程的名字，一个运行的线程总有一个名字，JVM给的名字或者我们自定义的名字，通过setName方法设置。
- 获取当前线程对象的方法:Thread.currentThread()。
- 在一个程序里多个线程只能保证其开始时间，而无法保证其结束时间，执行顺序也无法确定。

线程创建的问题

- 一个线程的run方法执行结束后，该线程结束。
- 一个线程只能被启动一次。
- 线程的调度是JVM的一部分，在一个CPU的机器上，一次只能运行一个线程。

  JVM线程调度程序决定实际运行哪个处于可运行状态的线程。采用队列形式。

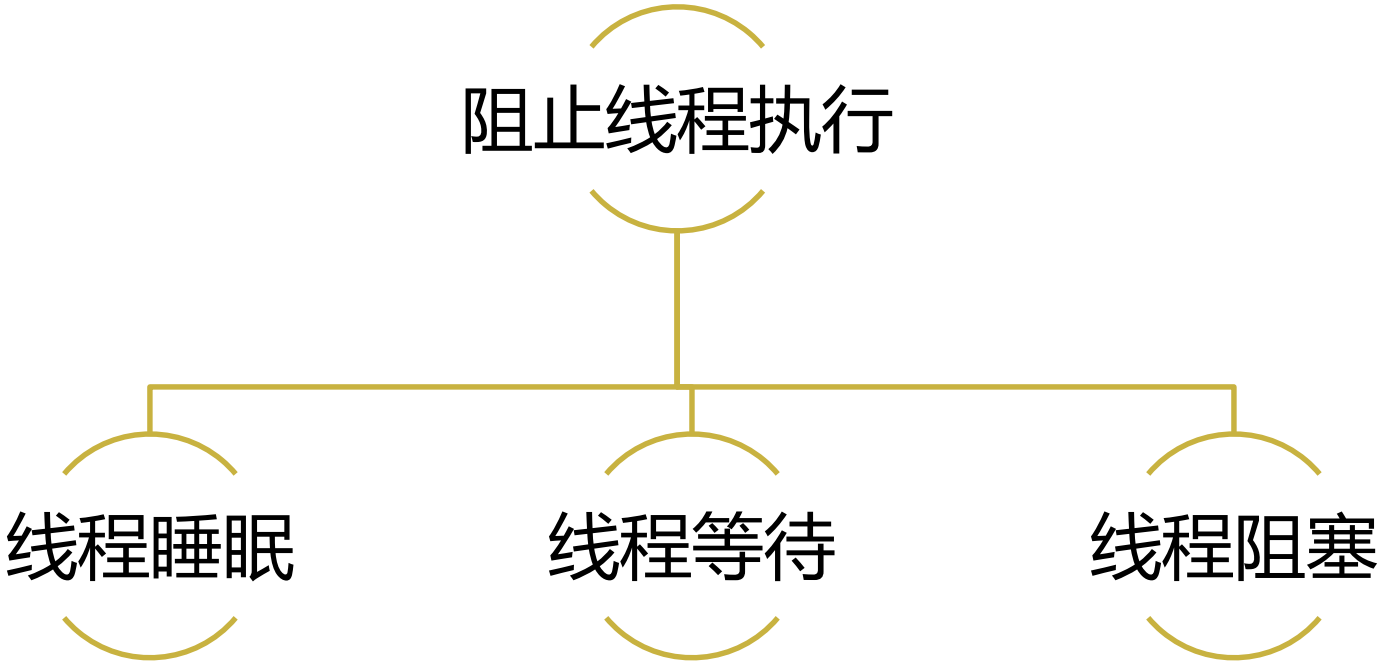
线程中的常用方法

- start():启动线程，让线程从新建状态进入就绪队列排队;
- run():线程对象被调度之后所执行的操作;
- sleep():暂停线程的执行，让当前线程休眠若干毫秒;
- currentThread():返回对当前正在执行的线程对象的引用;
- isAlive():测试线程的状态，新建、死亡状态的线程返回false。

线程中的常用方法

- interrupt():“吵醒”休眠的线程，唤醒“自己”；
- yield():暂停正在执行的线程，让同等优先级的线程运行;
- join():当前线程等待调用该方法的线程结束后，再排队等待CPU资源;
- stop():终止线程。

阻止线程执行的方法



线程睡眠

- Java中通过Thread的静态方法sleep来实现线程的睡眠。当线程睡眠时，它暂停执行，在苏醒之前不会返回到可运行状态。当睡眠时间到期，则返回到**可运行状态**。
- 使用场景：
  - 线程执行太快
  - 需要强制设定为下一轮执行
- 实现方法：在run方法中加入如下代码：

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

线程睡眠注意事项

- 线程睡眠是帮助其他线程获得运行机会的最好方法。
- 线程睡眠到期自动苏醒，并返回到可运行状态，不是运行状态。
- sleep()中指定的时间是线程不会运行的最短时间。因此，sleep()方法不能保证该线程睡眠到期后就开始执行。
- sleep()是静态方法，只能控制当前正在运行的线程。

课堂练习

- 计数器，从1-100之间计数，每隔1秒计数一次。

```
public class MyThread extends Thread {
    public void run() {
        for (int i = 0; i < 100; i++) {
            System.out.print(i);
            try {
                Thread.sleep(1000);
                System.out.print("    线程睡眠1000毫秒！\n");
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
        }
    }
    public static void main(String[] args) {
        new MyThread().start();
    }
}
```

线程的优先级

- 多线程运行时，JVM的调度策略为按优先级调度，级别相同时由操作系统按时间片来分配.
  - 线程优先级通常表示为1~10的数字
- 设置线程优先级
  - 线程默认优先级是创建它的执行线程的优先级.
  - 通过Thread实例调用setPriority()方法设置线程优先级
    - Thread.MIN_PRIORITY (1)
    - Thread.NORM_PRIORITY (5)
    - Thread.MAX_PRIORITY (10)
  - 通过Thread示例调用getPriority()方法得到线程优先级

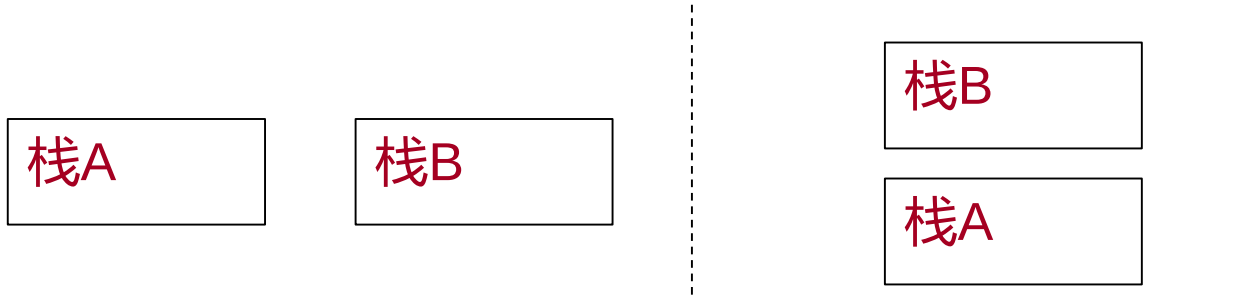
线程优先级及线程让步

- 当线程池中线程都具有相同的优先级，调度程序的操作有两种可能：
  - 一是选择一个线程运行，直到它阻塞或者运行完成为止。
  - 二是时间分片，为池内的每个线程提供均等的运行机会。
- 线程让步通过yield方法来实现，暂停当前正在执行的线程对象，并执行**同等优先级**的其他线程。
- 但yield()将导致线程从运行状态转到可运行状态，有可能没有效果无法保证yield()达到让步目的，因为让步的线程还有可能被线程调度程序再次选中。

```
Thread.yield();
```

线程阻塞

- 在线程A中调用B.join()。让一个线程A“加入”到线程B的尾部。在B执行完毕之前，A不能工作。
- join()方法还有带超时限制的重载版本。




```
Runnable r = new ThreadTest();
Thread t = new Thread(r);
t.start();
t.join();
t.join(100);
```

小结

- 线程离开运行状态的方法：
  - 调用Thread.sleep()：使当前线程睡眠至少多少毫秒（尽管它可能在指定的时间之前被中断）。
  - 调用Thread.yield()：不能保障太多事情，尽管通常它会让当前运行线程回到可运行性状态，使得有相同优先级的线程有机会执行
  - 调用join()方法：保证当前线程停止执行，直到调用join方法的线程完成为止。然而，如果调用join的线程没有存活，则当前线程不需要停止
  - 线程的run()方法完成。
  - 在对象上调用wait()方法（不是在线程上调用）。
  - 线程不能在对象上获得锁定，它正试图运行该对象的方法代码。
  - 线程调度程序可以决定将当前运行状态移动到可运行状态，以便让另一个线程获得运行机会，而不需要任何理由。

讲授思路-多线程的同步和死锁

- 多线程的问题
- 资源协调
- 多线程同步
- 死锁

多线程问题

- 多线程程序在设计上最大的困难在于各个线程的控制流彼此独立，使得各个线程之间的代码是乱序执行的，而且各个线程共享资源，所以多线程会带来线程调度、同步、死锁等一系列的问题。

例

一个工资管理人员正在修改雇员的工资表，
而一些雇员同时正在领取工资，
如果容许这样做，必然会引起发放工资的混乱。

多线程问题——资源协调

```
class Stack{
        int idx=0;
        char[ ] data = new char[6];
        public void push(char c){
             data[idx] = c;
              idx++;
        }
 }
```

```
public char pop(){
    idx--;
    char c = data[idx];
    data[idx] = ' ';
    return c;
}
```

- 两个线程A和B在同时使用Stack的同一个实例对象，A正在往堆栈里push一个数据，B则要从堆栈中pop一个数据。

多线程问题——资源协调

- 操作之前 data = | p | q | | | | | idx=2
- A执行push中的第一个语句，将r推入堆栈；

  data = | p | q | r | | | | idx=2
- A还未执行idx++语句，A的执行被B中断，B执行pop方法，返回q： data = | p | q | r | | | | idx=1
- A继续执行push的第二个语句： data = | p | q | r | | , | | idx=2；最后的结果相当于r没有入栈。

- 产生问题的原因在于对共享数据访问的操作的不完整性。
- Java引入了对象互斥锁的概念，来保证共享数据操作的完整性，从而避免上述问题。

资源同步——对象互斥锁

- Java中每个对象都对应一个称为“互斥锁”的标记
  - 关键字synchronized 与对象互斥锁联合起来使用保证对象在任意时刻只能由一个线程访问;

```
class Stack{
        int idx=0;
        char[ ] data = new char[6];
        public void push(char c){
            synchronized(this){
                data[idx] = c;
                 idx++;
            }
        }
}
```

```
public char pop(){
    synchronized(this){
        idx--;
        char c = data[idx];
        data[idx] = ' ';
        return c;
    }
}
```

资源同步——对象互斥锁

- synchronized可以修饰方法，表示这个方法在任意时刻只能由一个线程访问;
- synchronized可以修饰类，则表明该类的所有对象共用一把锁。

```
class Stack{
        int idx=0;
        char[ ] data = new char[6];
        public synchronized void push(char c){
              data[idx] = c;
              idx++;
        }
        public synchronized char pop(){
              ……
        }
}
```

```
class Stack{
        int idx=0;
        char[ ] data = new char[6];
        public void push(char c){
              synchronized(Stack.class){
                    data[idx] = c;
                     idx++;
               }
        }
        public char pop(){
              ……
        }
}
```

当多个线程共享一个资源的时候需要进行同步，但是过多的同步可能导致死锁

多线程同步模型

- 为了更好的解决多个交互线程之间的运行进度（即同步问题），Java引入了多线程同步的模型
  - 生产者——消费者示例

```
class SyncStack{
    private int index = 0;
    private char []buffer = new char[6];
    public synchronized void push(char c){
        while(index == buffer.length){
                try{
                    this.wait();
                }catch(InterruptedException e){
                }
        }
        this.notify();
        buffer[index] = c;
        index++;
    }
    }
```

多线程同步模型

```
        public synchronized char pop(){
                while(index ==0){
                        try{
                                this.wait();
                        }catch(InterruptedException e){
                        }
                }
                this.notify();
                index--;
                return buffer[index];
        }
}
```

```
class  Producer implements Runnable{
        SyncStack theStack;
        public Producer(SyncStack s){    theStack = s;  }
        public void run(){
                char c;
                for(int i=0; i<20; i++){
                        c =(char)(Math.random()*26+'A');
                        theStack.push(c);
                        System.out.println("Produced: "+c);
                        try{
                            Thread.sleep((int)(Math.random()*100));
                        }catch(InterruptedException e){}
                }
        }
}
```

```
class Consumer implements Runnable{
        SyncStack theStack;
        public Consumer(SyncStack s){
                theStack = s;
        }
        public void run(){
                char c;
                for(int i=0;i<20;i++){
                        c = theStack.pop();
                        System.out.println("Consumed: "+c);
                        try{
                            Thread.sleep((int)(Math.random()*1000));
                        }catch(InterruptedException e){}
                }
        }
}
```

多线程同步模型

```
public class SyncTest{
        public static void main(String args[]){
                SyncStack stack = new SyncStack();
                Runnable source=new Producer(stack);
                Runnable sink = new Consumer(stack);
                Thread t1 = new Thread(source);
                Thread t2 = new Thread(sink);
                t1.start();
                t2.start();
        }
}
```

多线程问题——死锁

- 当两个或两个以上的线程在执行过程中，因争夺资源而造成了互相等待，并且若无外力作用，它们都将无法推进下去的现象称为系统处在死锁状态或系统产生了死锁.
  - 资源占用是互斥的，当某个线程提出申请资源后，使得有关线程在无外力协助下，永远分配不到必需的资源而无法继续运行
- 产生死锁的必要条件
  - 互斥条件：指线程对所分配到的资源进行排它性使用.
  - 请求和保持条件：指线程已经保持至少一个资源，但又提出了新的资源请求.
  - 不可剥夺条件：进程已获得的资源，在未使用完之前，不能被剥夺，只能在使用完时由自己释放.
  - 环路等待条件:指在发生死锁时，必然存在一个线程—资源的环形链.

多线程问题——死锁

- 出现死锁的情况

相互排斥	• 一个线程永远占用某一共享资源
循环等待	• 线程A在等待线程B，线程B在等待线程C，线程C在等待线程A
部分分配	• 线程A得到了资源1，线程B得到了资源2，两个线程都不能得到全部的资源
缺少优先权	• 一个线程访问了某资源，但一直不释放该资源，即使该线程处于阻塞状态

总结

- 线程简介
  - 程序、进程、线程
  - 线程的应用场景
  - 线程的生命周期
- Java中创建多线程
  - 定义任务
  - 线程中的常用方法
  - 线程的优先级
- 多线程的同步和死锁
  - 多线程的问题
  - 资源协调
  - 多线程同步
  - 死锁

Thank You