



Java程序设计

2019春季

彭启民

pengqm@aliyun.com

■ 对象序列化

- 将对象转换为字节序列，并能够在以后将这个字节序列完全恢复为原来的对象。
- 可以将程序中的对象写入文件，之后再从文件中把对象读出来重新建立。
- 通过对象序列化将对象在不同的主机间传输，给分布式计算带来的方便

■ 对象序列化

- 实现**Serializable**接口，JVM就能自动来将类实现序列化。**Serializable**接口没有任何接口函数，只是一个空接口，唯一的作用就是标志类是可以序列化。
- 如果希望指定序列化的内容，则可以实现**Externalizable**接口

■ 对象序列化

```
public interface Externalizable extends Serializable {  
    public void writeExternal(ObjectOutput out) throws IOException ;  
    public void readExternal(ObjectInput in) throws IOException,  
    ClassNotFoundException ;  
}
```

■ 对象输入/输出流

- 把对象保存到外存，称为永久化。
- 实现`java.io.Serializable`接口类的对象可以被输入/输出。
- 只有对象的数据被保存，方法与构造函数不被保存。
- 以`transient`关键字标记的数据不被保存。

```
import java.io.Serializable;

public class MyClass implements Serializable {
    public transient Thread myThread ;
    Private transient String customerID;
    private int total;
    ...
}
```

- 对象序列化建立了一张对象网，将当前要序列化的对象中所持有的引用指向的对象都包含起来一起写入到文件
- 序列化类的所有成员变量也是可以序列化的。
 - 类A有个成员变量对象是类B，A能序列化，要求B也要能够序列化。
 - 如果你一次序列化了好几个对象，它们中相同的内容将会被共享写入

- 序列化对象都是通过Java的ObjectInputStream和ObjectOutputStream来实现的。

写:

```
ObjectOutputStream oos = new ObjectOutputStream(os);  
oos.writeObject(A);
```

读:

```
ObjectInputStream ios = new ObjectInputStream(is);  
A a = (A) ios.readObject();
```

```
public static void writeObjectToFile(Object obj)
{
    File file =new File("objects.dat");
    FileOutputStream out;

    Object obj = new someTh();//.....

    try {
        out = new FileOutputStream(file);
        ObjectOutputStream objOut=new ObjectOutputStream(out);
        objOut.writeObject(obj);
        objOut.flush();
        objOut.close();
        System.out.println("write object success!");
    } catch (IOException e) {
        System.out.println("write object failed");
        e.printStackTrace();
    }
}
```




```
public static Object readObjectFromFile()
```

```
{
    Object temp=null;
    File file =new File("objects.dat");
    FileInputStream in;
    try {
        in = new FileInputStream(file);
        ObjectInputStream objIn=new ObjectInputStream(in);
        temp=objIn.readObject();
        objIn.close();
        System.out.println("read object success!");
    } catch (IOException e) {
        System.out.println("read object failed");
        e.printStackTrace();
    } catch (ClassNotFoundException e) {
        e.printStackTrace();
    }
    return temp;
}
```

```
import java.io.Serializable;
import java.io.File;
import java.io.FileOutputStream;
import java.io.IOException;
import java.io.ObjectOutputStream;
```

Properties类

`java.util.Properties` 类表示了一个持久的属性集，继承于 **Hashtable**，是一个集合类，以集合的方式读写。

Properties 可保存在流中或从流中加载。属性列表中每个键及其对应值都是一个**字符串**。

Properties是Java中读写资源文件最重要的类,功能:

1. 读写**Properties**文件
2. 读写**XML**文件
3. 读写符合**key=value**格式的其它格式文件如**txt**等。

Properties属性文件用于保存**JAVA**应用程序配置信息，是文本文件。数据少，不必使用数据库管理，通过**File**操作处理不便。**Properties**文件为键值对。

属性文件中的每一行都是一个键值对应，代表了一个属性对象，以键和值的关系存储到**Properties**中。

Properties类的方法

除了从Hashtable中集成而来的put()等方法，还定义了以下方法：

序号	方法描述
1	String getProperty(String key) 用指定的键在此属性列表中搜索属性。
2	String getProperty(String key, String defaultProperty) 用指定的键在属性列表中搜索属性。
3	void list(PrintStream streamOut) 将属性列表输出到指定的输出流。
4	void list(PrintWriter streamOut) 将属性列表输出到指定的输出流。
5	void load(InputStream streamIn) throws IOException 从输入流中读取属性列表（键和元素对）。
6	Enumeration propertyNames() 按简单的面向行的格式从输入字符流中读取属性列表（键和元素对）。
7	Object setProperty(String key, String value) 调用 Hashtable 的方法 put。
8	void store(OutputStream streamOut, String description) 以适合使用 load(InputStream)方法加载到 Properties 表中的格式，将此 Properties 表中的属性列表（键和元素对）写入输出流。

例：已有属性文件prop.properties，内容如下：

```
item=test
```

```
url=www.ucas.ac.cn
```

首先将文件读取到Properties类对象中：

```
Properties prop = new Properties();//属性集合对象
```

```
FileInputStream fis = new FileInputStream("prop.properties");//属性文件流
```

```
prop.load(fis);//将属性文件流装载到Properties对象中
```

读取属性：，getProperty(String key)方法用来通过键名读取键值，若key不存在则赋予一个默认值时，可以使用public String getProperty(String key, String defaultValue)方法


用指定的键在属性列表中搜索属性。如果在属性列表中未找到该键，则接着递归检查默认属性列表及其默认值。如果未找到属性，则此方法返回默认值变量。

```
//获取item属性值
```

```
System.out.println("获取属性值： item=" + prop.getProperty("item"));
```

//获取属性值，comments未在文件中定义，则将返回一个默认值，并不修改属性文件

```
System.out.println("获取属性值： comments=" + prop.getProperty("comments", "未定义"));
```



修改/添加新的属性到属性集合方法: `setProperty(String key, String value)`
当属性集合中存在指定的key时，就修改这个key的值，否则新建一个key

//修改item的属性值

```
prop.setProperty("item", "Address");
```

//添加一个新的属性mail

```
prop.setProperty("mail", "postcode ");
```

保存到属性文件方法: `public void store(OutputStream out, String comments)`
将属性集合写到一个OutputStream流中。

//文件输出流

```
FileOutputStream fos = new FileOutputStream("prop.properties");
```

//将Properties集合保存到流中

```
prop.store(fos, "Properties demonstration");
```

```
fos.close();//关闭流
```

```

import java.io.FileInputStream;
import java.io.FileOutputStream;
import java.util.Properties;
public class PropertyDemo{
public static void main(String[] args) throws Exception {
    Properties prop = new Properties();// 属性集合对象
    FileInputStream fis = new FileInputStream("prop.properties");// 属性文件输入流
    prop.load(fis);// 将属性文件流装载到Properties对象中
    fis.close();// 关闭流
    // 获取已有属性值item
    System.out.println("获取属性值: item=" + prop.getProperty("item"));
    // 获取未定义属性值comments
    System.out.println("获取属性值: comments=" + prop.getProperty(" comments ",
“未定义”));
    prop.setProperty("item", "Address"); // 修改item的属性值
    prop.setProperty("mail", "postcode"); // 添加一个新的属性mail

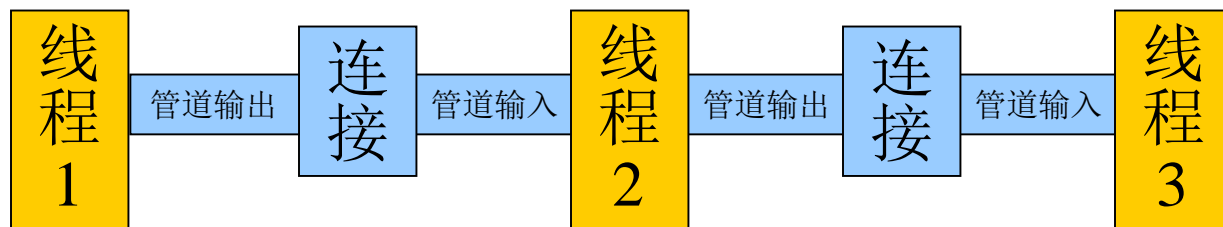
    FileOutputStream fos = new FileOutputStream("prop.properties"); // 文件输出流
    prop.store(fos, " Properties demonstation "); // 将Properties集合保存到
    fos.close();// 关闭流
}
}

```

流中

■ 管道流

- 管道用来把一个线程的输出连接到另一个线程的输入，实现线程间的通讯。
- PipedReader/PipedInputStream实现管道的输入端；
- PipedWriter/PipedOutputStream实现管道的输出端。



■ 管道流的创建

- 将一个线程的输出流直接挂在另一个线程的输入流，建立管道，实现线程间数据交换。

```
PipedInputStream pin= new PipedInputStream( );  
PipedOutputStream pout = new  
    PipedOutputStream(pin);
```

```
PipedInputStream pin= new PipedInputStream( );  
PipedOutputStream pout = new PipedOutputStream();  
pin.connect(pout); //或pout.connect(pin)
```


- 为了达到某个目的，需要包装好几层。

```
PrintWriter out1 = new PrintWriter (  
    new BufferedWriter (new FileWriter (  
        "IODemo.out") ) ) )
```

```
import java.io.BufferedReader;
import java.io.FileNotFoundException;
import java.io.FileReader;
import java.io.IOException;
```

```
public class datainput {
    public static void main(String[] args) throws IOException {
        BufferedReader br = null;
        String lineContent = "";
        String s = new String();
        Double[][] dat = new Double[100][100];
        int lin=0;

        try {
            br = new BufferedReader(new FileReader("record.txt"));
            while ((s = br.readLine()) != null) { // 判断是否读到了最后一行
                String arr[]=s.split("[\\t \\n]+");
                for (int i = 0; i < arr.length; i++)
                    dat[lin][i]=java.lang.Double.valueOf(arr[i]);
                lin++;
            }
        } catch (FileNotFoundException e) {
            e.printStackTrace();
        }
    }
}
```

■ 流API的演化（IO->NIO）

- JDK1.4引入NIO即New IO。NIO和IO有相同的作用和目的，但实现方式不同，NIO主要用到的是块，所以NIO的效率要比IO高很多。Java API中提供了两套NIO：针对标准输入输出NIO以及网络编程NIO
- Java 7中，NIO2又在NIO的基础上，引入了对异步IO的支持。
- Java 8.0引入了函数式编程的证明，Stream API提供了元素流的函数式操作，包括list、set、map等，还支持过滤filtering、映射mapping、移除集合中的重复元素等，可以从集合、数组、读缓冲区等获取流Stream。
-

Java中的XML解析

- XML（eXtensible Markup Language，可扩展标记语言）是一种通用的数据交换格式
- XML要求所有的标记必须成对出现，可以自己定义标签，大小敏感，要求嵌套、配对，并遵循DTD的树形结构
- XML的平台无关性、语言无关性、系统无关性、给数据集成与交互带来了极大的方便。

```
<?xml version="1.0" encoding="UTF-8"?>
<booklist>
  <book id="1">
    <name>book1</name>
    <language>English</language>
    <author>A</author>
    <year>2018</year>
    <price>68</price>
  </book>
  <book id="2">
    <name>book 2</name>
    <year>2019</year>
    <price>92</price>
  </book>
</booklist>
```

books.xml

Java中的XML解析

■ XML的常规解析方式

- DOM解析
- SAX解析
- JDOM解析
- DOM4J解析

其中前两种属于基础方法，是官方提供的平台无关的解析方式；后两种属于java平台的扩展方法

。

```
public class DOM4JTest {

    public static void main(String[] args) {
        // 创建SAXReader的对象reader解析books.xml文件
        SAXReader reader = new SAXReader();
        try {
            Document document = reader.read(new File("books.xml")); // 加载books.xml文件,获取document对象
            Element books = document.getRootElement(); // 通过document对象获取根节点
            Iterator it = books.elementIterator(); // 通过element对象的elementIterator方法获取迭代器
            while (it.hasNext()) { // 遍历迭代器, 获取根节点中的信息
                Element book = (Element) it.next();
                List<Attribute> bookAttrs = book.attributes(); // 获取book的属性名以及 属性值
                for (Attribute attr : bookAttrs) {
                    System.out.println("属性名: " + attr.getName() + "--属性值: " + attr.getValue());
                }
                Iterator itt = book.elementIterator();
                while (itt.hasNext()) {
                    Element bookChild = (Element) itt.next();
                    System.out.println("节点名: " + bookChild.getName() + "--节点值: " + bookChild.getStringValue());
                }
            }
        } catch (DocumentException e) {
            e.printStackTrace();
        }
    }
}
```

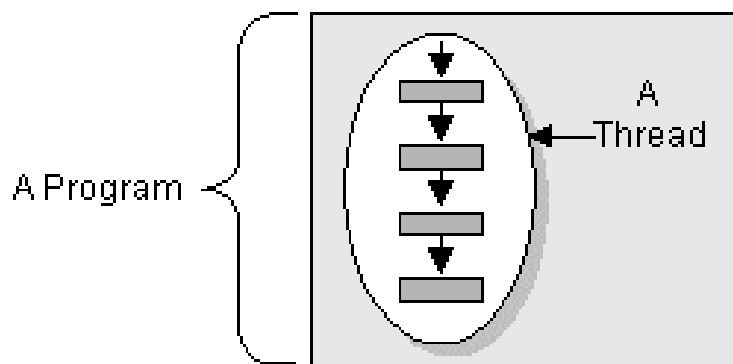


多线程

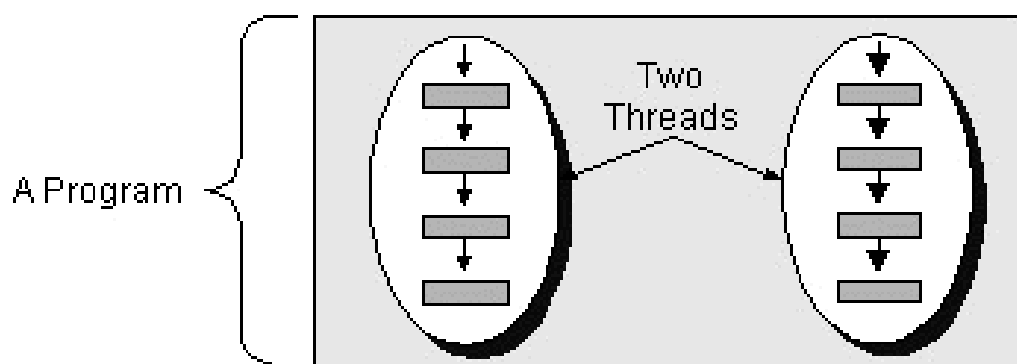
进程

- 进程是正在运行的一个程序
- 进程是分配资源的最小单位，是程序的一次运行
- 程序：静态对象——进程：动态过程
- 操作系统为每个进程分配一段内存空间，包括：代码、数据以及堆栈等资源
- 多任务的操作系统（OS）中，进程切换对CPU资源消耗较大

- 线程是程序中的单个执行流，多线程是一个程序 中包含的多个同时运行的执行流。




传统进程



多线程进程


线程与进程

- 进程：内核级的实体。包含虚存映象、文件指示符，用户ID等。这些结构都在内核空间中，用户程序只有通过系统调用才能访问与改变。
- 线程：用户级的实体。线程结构驻留在用户空间中，能够被普通的用户级函数组成的线程库直接访问。
 - 寄存器（栈指针，程序计数器）是线程专有的成分。
 - 一个进程中的所有线程共享该进程的状态。

- 
- 线程是比进程更小一级的执行单元
 - 线程不能独立存在，必须存在于进程中，各线程间共享进程空间的数据
 - 线程是程序运行的最小单位
 - 通常需要将一个程序转换成多个独立运行的子任务，每个子任务都叫做一个线程（Thread）

- 线程本身的数据通常只有**寄存器**数据，以及一个程序执行时使用的**堆栈**，所以线程的创建、销毁和切换的负荷远小于进程，又称为轻量级进程（**lightweight process**）。
- 多个进程的**内部数据**和**状态**都是完全独立的，而多线程是共享一块内存空间和一组系统资源，有可能互相影响



- 
- 多线程是实现并发的一种有效手段。
 - 一个进程可以通过运行多个线程来并发地执行多项任务。
 - 多个线程如何调度执行由系统来实现。

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,充分发挥多处理器的优势

创建线程

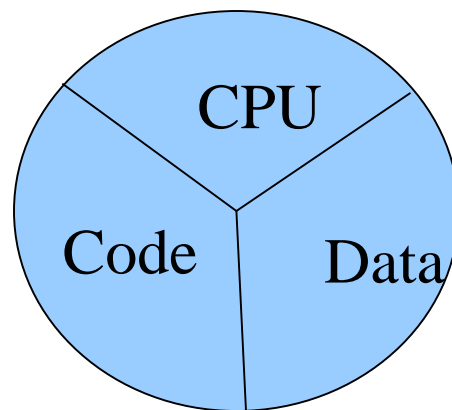
- 在**JAVA**中建立线程（**Thread**对象）：执行的代码、代码所操作的数据和执行代码的虚拟**CPU**。
 - 程序代码为类的成员方法。
 - 数据只能作为方法中的自动（或本地）变量或类的成员存在（对象）。
 - 虚拟**CPU**包装在**Thread**类的实例中。

■ JAVA的多线程机制

- 在Java编程中，每实例化一个线程对象，就创建一个虚拟的CPU，由虚拟CPU处理本线程数据
- 每个Java程序都有一个主线程，即由main()方法所对应的线程。对于applet，浏览器即是主线程
- 除主线程外，线程无法自行启动，必须通过其他程序来启动它

■ Java 中线程被认为是一个CPU、程序代码、和数据封装体。

- 一个虚拟的CPU
- 该CPU执行的代码
- 代码所操作的数据



注：代码与数据是相互独立的，代码与数据均可以与其它线程共享。



■ 对线程的综合支持是Java技术的一个重要特色

- 提供了**thread**类、监视器和条件变量的技术，对数据同步的支持更充分。
- 线程间的执行是相互独立的。
- 线程独立于启动它的线程（或程序）。

线程的构造

- **Java.lang.Thread**类使用户可以创建和控制自己的线程
- 在**Java**中，虚拟**CPU**是自动封装进**Thread**类的实例中，而**Code**和**Data**要通过一个对象传给**Thread**类的构造函数
- 线程的**Code**和**Data**构成线程体。线程体决定了线程的行为。
 - 同步性
 - 互斥性
 - 优先级

■ 实现多线程的两种编程方法

□ 实现 **Runnable** 接口

- **Runnable** 接口只提供了一个 `public void run()` 方法。

□ 继承 **Thread** 类

- `start()`
- `stop()`
- `run()`

■ Runnable接口

- 任何线程类都必须要实现的一个接口，Thread类也不例外
- run()方法

■ Runnable接口的使用

- 自定义类实现Runnable接口
- 将该类的实例作为参数传给Thread类的一个构造函数，从而创建一个线程。
 - Thread(Runnable, String)
- 使用start()启动线程

■ 例:

```
class A implements Runnable{
    public void run(){....}
}
class B {
    public static void main(String[] arg){
        Runnable a=new A(); //A a=new A();
        Thread t=new Thread(a);
        t.start();
    }
}
```

Thread类

- java.lang包
- 构造函数
 - Thread(); 无参数
 - Thread(String threadname); 指定线程实例名
 - Thread(Runnable,String); 指定线程实例名



■ Thread类

- Thread 类本身实现了Runnable接口。

- 基本步骤

 - 继承Thread类


 - 重写其中的run()方法定义线程体

 - 定义线程的具体操作

 - 系统调度此线程时自动执行

 - 初始时无具体操作内容


 - 创建该子类的对象创建线程




```
public class Counter extends Thread{  
    public void run( ){  
        ...  
    }  
}
```

■ 创建与运行线程:

```
Counter ThreadCounter = new Counter( );  
ThreadCounter.Start( );
```



```
public class SimpleThread extends Thread {  
    public SimpleThread(String str) {  
        super(str);  
    }  
    public void run() {  
        for (int i = 0; i < 10; i++) {  
            System.out.println(i + " " + getName());  
            try {  
                sleep((int)(Math.random() * 1000));  
            } catch (InterruptedException e) {}  
        }  
        System.out.println("DONE! " + getName());  
    }  
}
```



```
public class TwoThreadsTest {  
    public static void main (String[] args) {  
        new SimpleThread("Beijing??").start();  
        new SimpleThread("Shanghai!!").start();  
    }  
}
```

线程两种创建方法比较

- 实现**Runnable**接口的优势：

- 符合OO设计的思想。
- 便于用**extends**继承其它类。

- 采用继承**Thread**类方法的优点：

- 程序代码更简单。

（提倡采用第一种方式实现多线程）

线程的运行

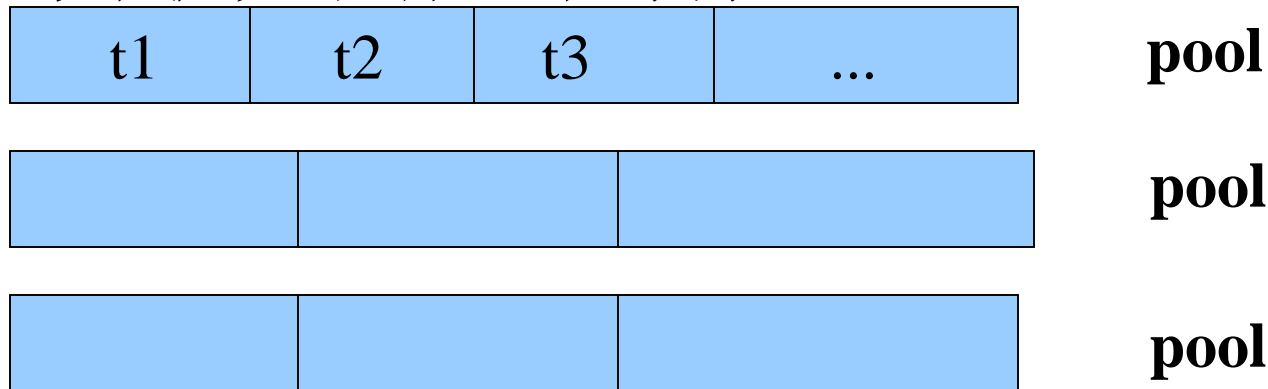
- 新创建的线程不会自动运行。必须调用线程的**start()**方法,把嵌入在线程中的虚拟CPU置为可运行（**Runnable**）状态。
 - **Runnable**状态意味着该线程可以参加调度，被JVM运行，并不意味着线程会立即执行。

■ 线程的优先级

□ 每个线程都有优先级

- 有缺省值，可用**SetPriority()**方法改变。

□ 每个优先级有一个等待池





■ 线程的优先级控制

- 三个常量:

 - MAX(10),MIN(1),NORM_PRIORITY(5)

- getPriority(),setPriority(int nP)

- 线程创建时继承父线程的优先级



■ Java的调度方法

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

■ Java中线程抢占式调度模式

- 许多线程可能是可运行的，但只能有一个线程在运行。
 -
- 该线程将持续运行，直到它自行中止或出现高优先级
- 线程成为可运行的，则该低优先级线程被高优先级线程强占运行。

- 线程中止的原因可能有多种，如执行 `Thread.sleep()` 调用，或等待访问共享的资源。

线程并发引起的问题

- 多个线程相对执行的顺序是不确定的。
- 线程执行顺序的不确定性会产生执行结果的不确定性。
- 在多线程对共享数据 操作时常常会产生这种不确定性。

■ 问题的解决

- 同步: 用 **synchronized** 关键字前缀给针对共享资源的操作加锁; 同步方法、同步块

`synchronized void push()`

`synchronized int pop()`

- 管程: 具有互斥锁访问性质的一种程序结构, 其中的多个线程互斥访问共享资源。

■ 对象锁

- Java中每个对象都带有一个monitor标志，相当于一个锁。
- **synchronized**关键字用来给对象加上独占的排它锁

■ 返还对象的monitor

- 当synchronized(){ }语句块执行完毕后。
- 当在synchronized(){ }语句块中出现exception.
- 当调用该对象的wait()方法。将该线程放入对象的wait pool中，等待某事件的发生。

- 对共享数据的所有访问都必须使用 **synchronized**.
- 用 **synchronized** 保护的共享数据必须是私有的，使线程不能直接访问这些数据，必须通过对象的方法。
- 如果一个方法的整体都在 **synchronized** 块中，则可以把 **synchronized** 关键字放于方法定义的头部的：

- **Java**运行系统允许已经拥有某个对象锁的线程再次获得该对象的锁

```
public class Test {  
    public synchronized void a() {  
        b();  
        System.out.println("here I am, in a()");  
    }  
    public synchronized void b() {  
        System.out.println("here I am, in b()");  
    }  
}
```

线程的死锁

■ 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：两个线程同时等待对方持有的锁
- 哲学家进餐问题(thinking or eating)

■ 避免死锁

□ 死锁的避免完全由程序控制。

- 如果要访问多个共享数据对象，则要从全局考虑定义
- 一个获得封锁的顺序，并在整个程序中都遵守这个顺序。释放锁时，要按加锁的反序释放。

□ 专门的算法、原则

□ 尽量减少同步资源的定义

■ 不建议使用的方法

- `stop()`：线程强行终止，容易造成数据的不一致。
- `suspend()` 和 `resume()`：使一个线程A可以通过调用`B.suspend()`直接控制B的运行。`Suspend()`方法将不使B释放锁。容易发生死锁。

■ 建议使用

■ daemon线程

- 守护线程是为其它线程提供服务的线程
- 守护线程一般应该是一个独立的线程,它的run()方法是一个无限循环.
- 守护线程与其它线程的区别是,如果守护线程是唯一运行着的线程,程序会自动退出
- 在客户/服务器模式下,服务器的作用是等待用户发来请求,并按请求完成客户的工作

■ 几种基本状态

- **Newborn:** 线程已创建，但尚未执行

- **Runnable:** （就绪）

线程已被调度，按优先级和先到先服务原则在队列中排队等待CPU时间片资源

- **Runnnig:** 正在运行

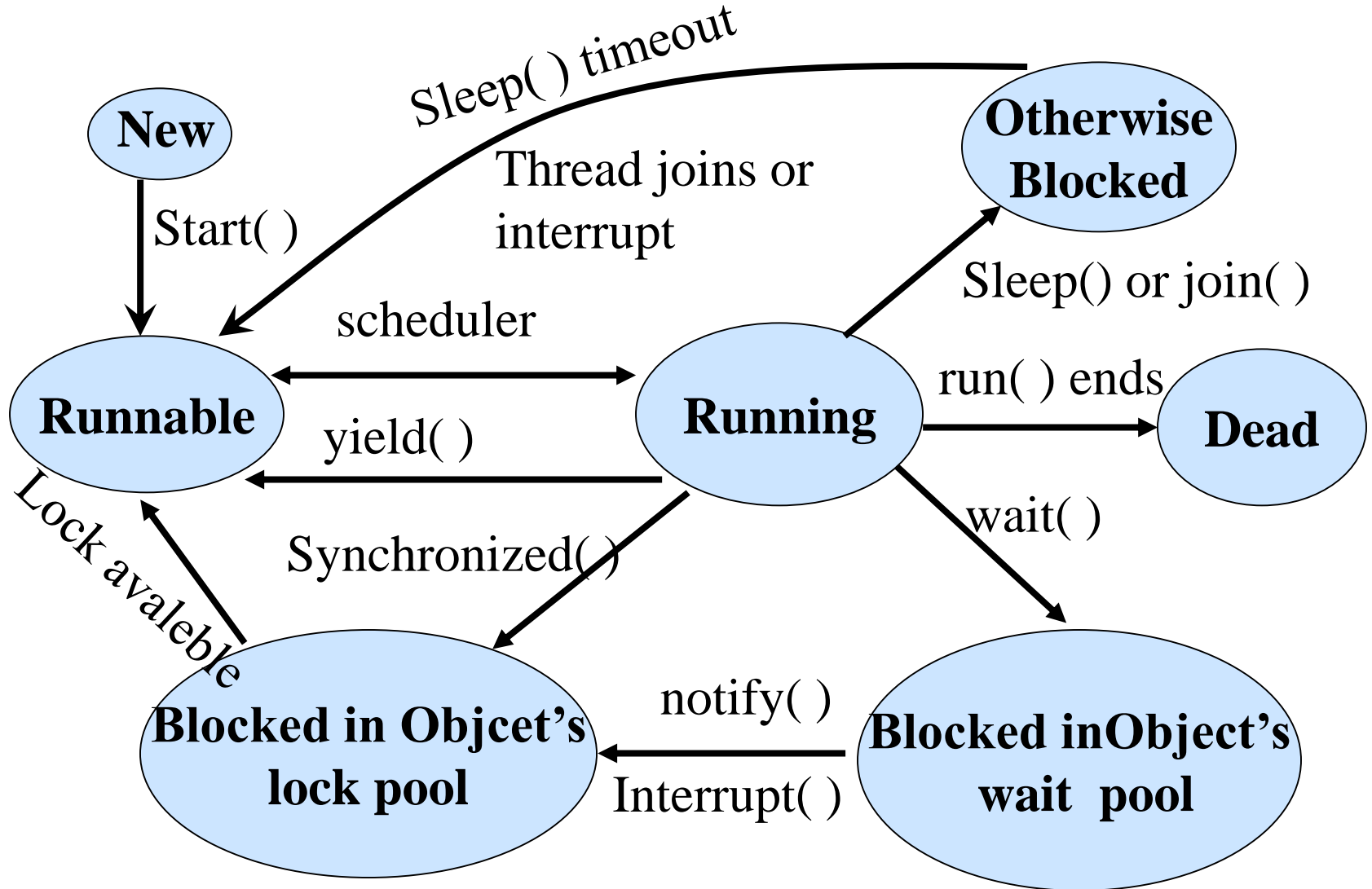
- **Blocked:** （阻塞）

因某事件或睡眠而被暂时性地挂起

- **Dead:** 正常/强行中断，退出运行状态

- 线程状态由线程控制方法，如`sleep()`, `join()`或线程同步控制方法引起变化。

线程状态与生命周期



线程的基本控制

■ 获取当前线程

- Thread 类的静态方法 `currentThread()` 返回当前线程

■ 测试线程

- 线程的状态未知时，用 `isAlive()` 确定线程是否活着。返回 `true` 意味着线程已经启动，但还没有运行结束

■ `start()`: 启动线程

■ `run()`: 线程在被调度时执行的操作

■ 线程的基本控制

□ sleep()

- 该方法用来使一个线程放弃对**CPU**控制, 暂停运行一段固定的时间, 使其他线程有机会被执行（在线程睡眠时间内, 将运行别的线程）。
- 时间到, 线程将进入**Runnable**状态, 重新排队。
- **sleep**要丢出异常**InterruptedException**, 必须捕获.

```
try{sleep(100)}catch(InterruptedException e){ }
```

动画

- 即若干幅相近的图片接连显示
- 例： 旋转的地球 RollEarth.java



```
public class RollEarth extends Applet implements Runnable {
    ....
    public void run() {
        ....
        while (true) {
            try {
                displayImage(m_Graphics);
                m_nCurrImage++;
                if (m_nCurrImage == NUM_IMAGES)
                    m_nCurrImage = 0;
                Thread.sleep(50);
            } catch (InterruptedException e) {
            }
        }
        ....
    }
}
```


- **suspend()** : 挂起线程, 处于阻塞状态
- **resume()**: 恢复挂起的线程, 重新进入就绪队列排队

应用: 可控制某线程的暂停与继续

方法: 设一状态变量 `suspendStatus=false` (初始)

暂停: `if(!suspendStatus)`
 `{T.suspend(); suspendStatus=true;}`

继续: `if(suspendStatus)`
 `{T.resume(); suspendStatus=false;}`

■ yield()

- 对正在执行的线程
- 若就绪队列中有与当前线程同优先级的排队线程, 则当前线程让出CPU控制权, 移到队尾
- 若队列中没有同优先级的线程, 忽略此方法

■ stop()

- 强制线程生命期结束

■ isAlive()

- 返回boolean, 表明是否还存在

■ 线程的基本控制

□ join()

- 线程a.join()方法使当前的线程等待，直到线程a结束为止，当前线程恢复到runnable状态。要求捕获异常。

```
Public void doTask(){
    TimerThread tt= new TimerThread(100) ;
    tt.start( ) ;
    ...
    // Do stuff in parallel with the other thread for a while
    ...
    // Wait here for the timer thread to finish
    try{
        tt.join( );
    } catch( InterruptedException e){ // tt came back early }
    ...
    // continue in this thread
    ...
}
```

■ 线程的基本控制


□ yield()

- 调用该方法将**CPU**让给具有与当前线程相同优先级的线程。
- 如果没有同等优先级的线程是**Runnable**状态，**yield()**方法将什么也不做。
- **yield()** 方法与**sleep()** 方法相似，只是它不能由用户指定线程暂停多长时间。
 - **SUN**: **sleep**方法可以使低优先级的线程得到执行的机会，当然也可以让同优先级和高优先级的线程有执行的机会。而**yield()**方法只能使同优先级的线程有执行的机会。

■ 线程的基本控制

□ 结束线程

- 线程完成运行并结束后，将不能再运行。
- 除正常运行结束外，还可用其他方法控制使其停止。
 - 用`stop()`方法强行终止线程
 - 容易造成线程的不一致，不再提倡！
 - 设置`flag` 指明`run()`方法应该结束



```
class Xyz implements Runnable{
    private boolean  timeToQuit = false;
    public void run( ){
        while(!timeToQuit){ ... }
        // clean up before run() ends
        ...
    }
    public void stopRunning( ){
        timeToQuit = true ;
    }
}

public class ControlThread{
    public void main(String [ ] args){
        Runnable r = new Xyz( ) ;
        Thread t = new Thread( r) ;
        t.start();
        r.stopRunning( );
    }
}
```

线程的协调与通讯

- 生产者/消费者问题

- **wait()**与**notify()**

- 是Object类的方法： **public final void**，与**sleep()**和**wakeup()**等价，但在同步方法中使用时，不受竞争条件约束。

- 线程**sleep()**的时候并不释放对象的锁，但是**wait()**的时候却会释放对象的锁。

- 在线程**wait()**期间，别的线程可以调用它的**synchronized**方法。

- **wait()**: 令当前线程挂起并放弃管程，同步资源解锁，使别的线程可访问并修改共享资源，而当前线程排队等候再次对资源的访问
- **notify()**唤醒正在排队等待资源管程的线程中优先级最高者，使之执行并拥有资源的管程
- **wait() + notify() + 标志变量**: 可协调、同步不同线程的工作

■ Wait() 和 notify()

- 线程在 **synchronized** 块中调用 **x.wait()** 等待共享数据的某种状态。该线程将放入对象 **x** 的 **wait pool**, 并且将释放 **x** 的 **monitor**。
- 线程在改变共享数据的状态后, 调用 **x.notify()**, 则对象的 **wait pool** 中的一个线程将移入 **lock pool**, 等待 **x** 的 **monitor**, 一旦获得便可以运行。

■ Notifyall() 把对象 **wait pool** 中的所有线程都移入 **lock pool**。

多线程程序设计

■ 任何事物都有其两面性

- 一方面多线程与单线程相比有许多优越的性能；
- 另一方面编写多线程程序比较复杂，也容易出错。其中有线程的同步（线程之间的配合），互斥（一线程等待另一线程的运行），优先级的设置，以及多线程引起的死锁等。

■ 是否需要多线程？何时需要多线程？

- 多线程的核心在于多个代码块并发执行
- 本质特点在于各代码块之间的代码是乱序执行的（即执行的顺序不可预测）。
- 程序是否需要多线程，要看其内在特点。

多线程：并发容器类

- 串行访问容器状态以实现线程安全，当多线程竞争容器锁时，吞吐量会下降。
- Java1.5
 - ConcurrentHashMap //Hashmap
 - CopyOnWriteArrayList //List
 - Queue和BlockingQueue接口 //Collection
 - ConcurrentLinkedQueue
- Java1.6
 - ConcurrentSkipListMap //SortedMap
 - ConcurrentSkipListSet //SortedSet

作业2

- 编写可读取如下格式xml文档的程序。
 - （5月11日24时前课程网提交源程序，注明姓名学号）


```
<?xml version="1.0" encoding="UTF-8"?>
<books>
  <book id="001">
    <title>Harry Potter</title>
    <author>J K. Rowling</author>
    <price>$50.2</price>
  </book>
  <book id="002">
    <title>Learning XML</title>
    <author>Erik T. Ray</author>
    <price>$90</price>
  </book>
</books>
```

作业3


运行如下四段代码并解释结果

```
import java.io.*;
//implements Runnable
public class ThreadTest1 extends Thread {
    public void run() {
        for (int i = 0; i < 10; i++) System.out.print(" " + i);
    }

    public static void main(String[] args) {
        new ThreadTest1().start();
        new ThreadTest1().start();
    }
}
```




```
import java.io.*;
public class ThreadTest2 implements Runnable {
    public synchronized void run() {
        for (int i = 0; i < 10; i++)    System.out.print(" " + i);
    }
    public static void main(String[] args) {
        Runnable r1 = new ThreadTest2();
        Runnable r2 = new ThreadTest2();
        Thread t1 = new Thread(r1);
        Thread t2 = new Thread(r2);
        t1.start();
        t2.start();
    }
}
```



```
import java.io.*;
public class ThreadTest3 implements Runnable {
    public synchronized void run() {
        for (int i = 0; i < 10; i++) System.out.print(" " + i);
    }

    public static void main(String[] args) {
        Runnable r = new ThreadTest3();
        Thread t1 = new Thread(r);
        Thread t2 = new Thread(r);
        t1.start();
        t2.start();
    }
}
```



```
import java.io.*;
public class ThreadTest4 implements Runnable {
    public void run() {
        synchronized (this) {
            for (int i = 0; i < 10; i++)    System.out.print(" " + i);
        }
    }
}
```

```
public static void main(String[] args) {
    Runnable r = new ThreadTest4();
    Thread t1 = new Thread(r);
    Thread t2 = new Thread(r);
    t1.start();
    t2.start();
}
}
```


要求：

- 5月11日24时前课程网提交。
- 文本文件形式（.txt），注明姓名学号，说明各程序运行情况，并简要分析原因。