Java Application Design

By 一二的夏季

I/O

总览

java中的字符使用unicode编码，一个字符两个字节

类	说明
File	文件类，代表一个特定文件信息，或代表一个目录下的一组文件
InputStream	字节输入流接口，表示从不同数据源产生输入的类
OutputStream	字节输出流接口，表示数据输出的目标
Read	字符输入流接口
Writer	字符输出流接口

File

File类是对文件系统中文件以及文件夹进行封装的对象，可以通过对象的思想来操作文件和文件夹。File类保存文件或目录的各种元数据信息，包括文件名getName()、文件长度length()、最后修改时间lastModified()、是否可读写canRead()、canWrite()、获取文件的路径getPath()，判断文件是否存在exists()、获得当前目录中的文件列表list()、创建mkdirs()、删除delete()等方法

```
package io.iosystem;

import java.io.File;
import java.io.FilenameFilter;

public class DirList {
    public static void main(String[] args) {
        File path = new File(".");
        System.out.println("绝对路径: " + path.getAbsolutePath());
        System.out.println("可读写: " + path.canRead() + " " + path.canWrite());
        System.out.println("文件名: " + path.getName());
        System.out.println("最后修改时间: " + path.lastModified());
        // list()方法可接受一个FilenameFilter对象，
        // 作为策略模式，list会回调FilenameFilter.accept()方法，允许我们对文件夹中的子文件名称做额外的处理
        // 如下，只会返回文件名小于10字符的文件名组
        String[] list = path.list(new FilenameFilter() {
            @Override
            public boolean accept(File dir, String name) {
                return name.length() < 5 ? true : false;
            }
        });
        for (String s : list) {
```

```
            System.out.println(s);
        }
    }
}

/*output:
绝对路径: D:\IdeaProjects\Test\thinkingInJava\.
可读写: true true
文件名: .
最后修改时间: 1548920209829
file
out
src
*/
```

流

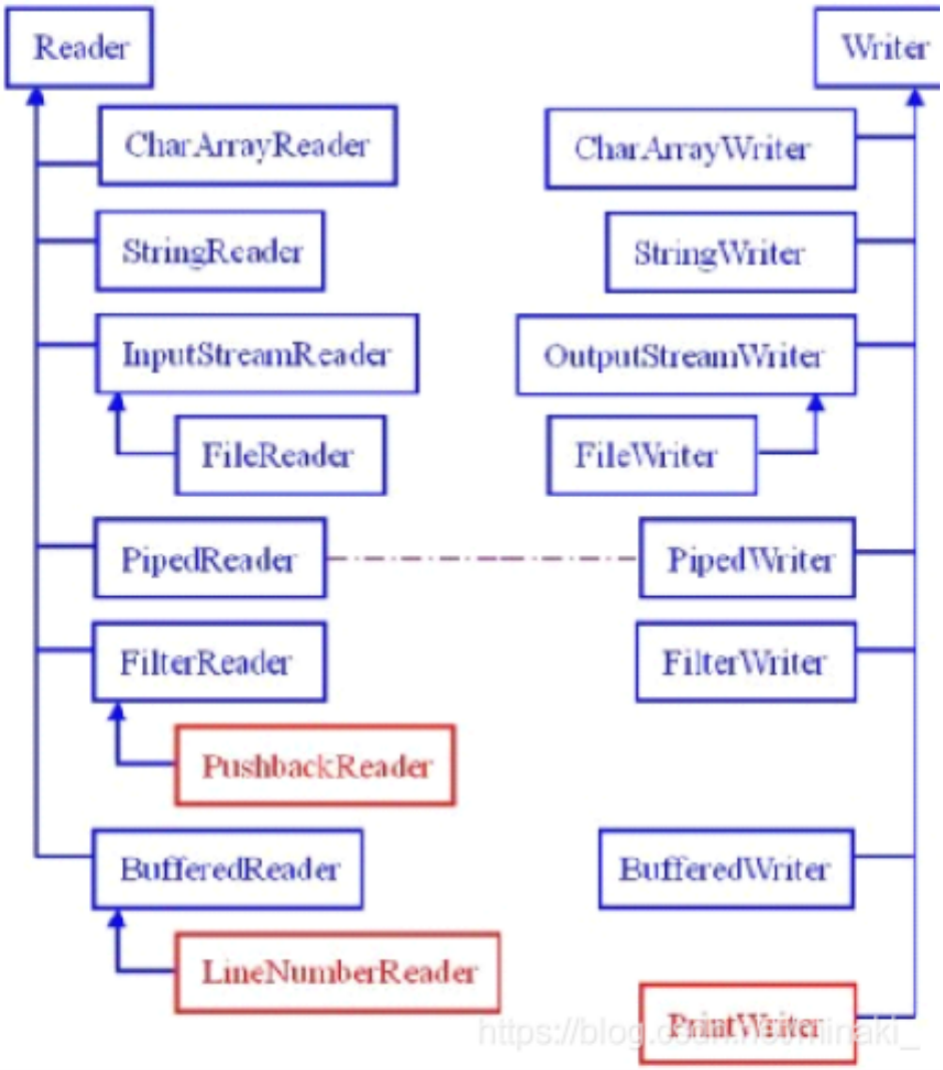
- Java中所有的I/O都是通过流来实现的。
- 根据处理数据类型的不同分为：字符流和字节流；他们的区别是：读写单位不同：字节流以字节（8bit）为单位，字符流以字符为单位，根据码表映射字符，一次可能读多个字节；处理对象不同：字节流能处理所有类型的数据（如图片、视频等），而字符流只能处理字符类型的数据。
- 根据数据流向不同分为：输入流和输出流。
- Java通过系统类System实现标准输入/输出的功能，定义了3个流变量:in,out,和err.这3个流在Java中都定义为静态变量，可以直接通过System类进行调用。System.in表示标准输入，通常指从键盘输入数据；System.out表示标准输出，通常指把数据输出到控制台或者屏幕。System.err表示标准错误输出，通常指把数据输出到控制台或者屏幕，后两者都是标准输出对象。

字节流（8bit，Stream class）

- 图中蓝色为输入输出对应部分，红色为不对应的部分，绿色部分已被弃用Deprecated不推荐使用，紫色虚线部分代表这些流一般需要搭配使用。其中，InputStream类和OutputStream类是所有二进制I/O的根类。
- 要求写一段代码读取一个文本文件，使用FileInputStream
- PipedOutputStream和PipedInputStream：它们分别是管道输出流和管道输入流，必须配套使用。

  当我们在线程A中向PipedOutputStream中写入数据，这些数据会自动的发送到与PipedOutputStream对应的PipedInputStream中，进而存储在PipedInputStream的缓冲中；线程B通过读取PipedInputStream中的数据。**PipedOutputstream必须基于PipedInputStream才能建立**
- DataInputStream参数可以使用FilterInputStream

字符流（16bit,reader和writer）



例题：After executing the function, the size of the file “data.out” is:

```
void write() throws IOException {
        DataOutputStream out = new DataOutputStream(
            new FileOutputStream( new File("bin")));
        out.writeInt(1);
        out.write(1);
        out.close();
}
//答案是5，原因是write(int) 继承自父类OutputStream只写入int的最低位的一个字节，
out.writeInt则是写入了32位，最后输出的是字节数
```

序列化

定义

- 很多时候，传输的数据是特有的类对象，而类对象仅仅在当前jvm是有效的，传递给别的jvm或者传递给别的语言的时候，是无法直接识别类对象的，那么，我们需要多个服务之间交互或者不同语言交互，该怎么办？这个时候我们就需要通过固定的协议，传输固定的数据格式，而这个数据传输的协议称之为序列化
- 在java语言中实例对象想要序列化传输，需要实现Serializable 接口，只有当前接口修饰定义的类对象才可以按照指定的方式传输对象。而传输的过程中，需要使用java.io.ObjectOutputStream 和java.io.ObjectInputStream 来实现对象的序列化和数据写入

```
public class User implements Serializable{}
//编写发送对象(序列化)的实现:
public class OutPutMain
{
   public static void main( String[] args ) throws UnknownHostException,
IOException
   {
       Socket socket = new Socket("localhost",8080);
       try(ObjectOutputStream outputStream = new
ObjectOutputStream(socket.getOutputStream())){
           User user = new User().setAge(10).setId(10).setName("张
三").setSex((byte)0);
           outputStream.writeObject(user);
           outputStream.flush();
           System.out.println("对象已经发送：--->"+user);
       }catch (Exception e) {
           e.getStackTrace();
           System.err.println("对象发送失败:--->");
       }finally{
           if(!socket.isClosed()){
               socket.close();
           }
       }
   }
}
//定义读取实体(反序列化)的代码:
public class InputMain {
    public static void main(String[] args) throws IOException {
        ServerSocket serverSocket = new ServerSocket(8080);
        Socket socket = serverSocket.accept();
        try(ObjectInputStream inputStream = new
ObjectInputStream(socket.getInputStream())){
            User user = (User) inputStream.readObject();
            System.out.println(user);
        }catch (Exception e) {
            e.getStackTrace();
        }finally {
            if(!serverSocket.isClosed()){
                serverSocket.close();
            }
        }
    }
}
```

serialVersionUID

- jdk推荐我们实现序列化接口后，让我们再去生成一个固定的序列化id--serialVerionUID,而这个id的作用是用来作为传输/读取双端进程的版本是否一致的，防止我们因为版本不一致导致的序列化失败。
- UID的选取，一种是固定的用1L，一种是根据类的各种性质动态生成

Transient关键字

- 这个是java针对序列化出的关键字，修饰在指定字段上，可以在序列化的时候，排除当前关键字修饰的字段，仅序列化其他字段，当我们反序列化的时候，可以看到基础类型为默认值，引用类型则为null

```java
public class User implements Serializable{
    private static final long serialVersionUID = 2L;
    //不序列化id字段
    private transient Integer id;
    private String name;
    private Byte sex;
    private Integer age;


//对象已经发送：--->User [id=10, name=张三, sex=0, age=10]
//User [id=null, name=张三, sex=0, age=10]
```

- **序列化不是万能的，除了transient关键字外，如果某个属性存在static关键字修饰，那么无论是否有transient修饰，都不能参与序列化**

GUI-The Model and Design

历史

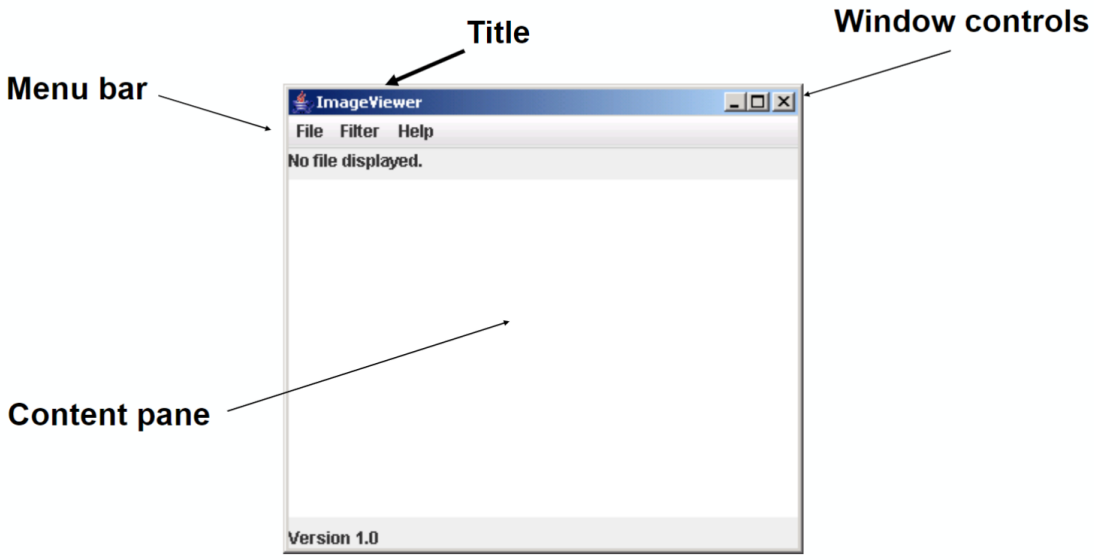
嵌入式：Android>MiniGUI（需要高度定制，工业上优势小）>QT

桌面：Windows更新后，程序员放弃同步更新，可以跨平台的QT成为选择（**pyQT**）

考试不考GUI具体的内容，会考GUI的设计模式

AWT and Swing

整体界面



区别联系

- AWT和Swing之间的区别：

  1)AWT 是基于本地方法的C/C++程序，其运行速度比较快；Swing是基于AWT的Java程序，其运行速度比较慢。

  2)AWT的控件在不同的平台可能表现不同，而Swing在所有平台表现一致

3)Swing是对AWT出现问题的改进，但是并不是子集

基础概念

- System.out.println()还是在console中输出的，因此可以保证好的程序调试
- 随着窗口的放大缩小不断重画，最小化重画，最大化重画，但是拖动窗口不会重画repaint
- 矩形的右下角是getwidth,getheight,左上角是0,0
- 组件支持叠加，就是A组件里可以多个B组件，B组件又可以包含多个C组件
- Java可以设置程序的界面外观，即可以让程序在不同操作系统下按照系统特有的外观风格显示，也可以将风格统一。
- Font的style，3种（中文里的汉字其实不支持斜体，因为无这样的斜体）。英文有衬线（Serif，对应“SansSerif”）的字体-对应中文的宋体（有衬线可以有效消除视觉疲劳，因此出版物的正文都是宋体）

代码概念

- Jframe是最大的快，在其上增加Jpanel（四周和中间的）、menuBar（菜单的）
- 当使用BorderLayout去完成布局时，东南西北的添加顺序就无关紧要了，每个区域一个组件，多个组件可以包含在Jpanel内。如果一个区域加了多个组件，有效的是最后一个。
- Jpanel的缺省布局管理器是FlowLayout（帮助维持button'大小），其中可以包含多个组件
- Menubar里可以添加多个JMenu，JMenu里可以添加多个JMenuItem。JMenuItem和普通按钮JButton具有相似的行为，它们具有相同的父类AbstractButton。**当菜单被点击，触发的事件是ActionEvent**
- 每个组件都可以添加多个Listener触发；When an event occurs, the source object notices all the registered listeners；A registered listener is able to be de-registered from a source object dynamically；One listener can be registered at more than one source object
- Label组件用来显示文本
- 要设置位置的原因是，为了不让组件在拉动窗口的时候有所变化
- 事件监听上，如果是鼠标移动触发的是事件MouseMotionListener.

User Action	*Source Object*	*Event Type Fired*
Click a button	`JButton`	`ActionEvent`
Press return on a text field	`JTextField`	`ActionEvent`
Select a new item	`JComboBox`	`ItemEvent`, `ActionEvent`
Select item(s)	`JList`	`ListSelectionEvent`
Click a check box	`JCheckBox`	`ItemEvent`, `ActionEvent`
Click a radio button	`JRadioButton`	`ItemEvent`, `ActionEvent`
Select a menu item	`JMenuItem`	`ActionEvent`
Move the scroll bar	`JScrollBar`	`AdjustmentEvent`
Move the scroll bar	`JSlider`	`ChangeEvent`
Window opened, closed, iconified, deiconified, or closing	`Window`	`WindowEvent`
Mouse pressed, released, clicked, entered, or exited	`Component`	`MouseEvent`
Mouse moved or dragged	`Component`	`MouseEvent`
Key released or pressed	`Component`	`KeyEvent`
Component added or removed from the container	`Container`	`ContainerEvent`
Component moved, resized, hidden, or shown	`Component`	`ComponentEvent`
Component gained or lost focus	`Component`	`FocusEvent`

错题

- J20 In the model-view-controller (MVC), the controller can be implemented as a listener in the event delegation model. False
- Java GUI程序中，“模式”形式的对话框在关闭前主窗口不能接收任何形式的输入。 True
- Java GUI程序中,当关闭框架JFrame时，缺省地也会关闭整个应用程序。 False（设置参数）
- The statement for registering a listener for processing list view item change is lv.getSelectionModel().selectedItemProperty().addListener(e -> {processStatements});
- How to add a separator to a Menu?　myMenu.addSeparator();
- Which method below is to be executed only once during a lifecycle of an Applet——Init（）

线程（超级重要）

- 没有指针，传入传出的是类对象。只要你new出了一个，你可以在几个.java文件里不断的用=传这个对象，然后修改这个对象。
- 习题：2-8

基础知识

接口

Runnable只有一个方法，就是run

start（），sleep（）是thread的

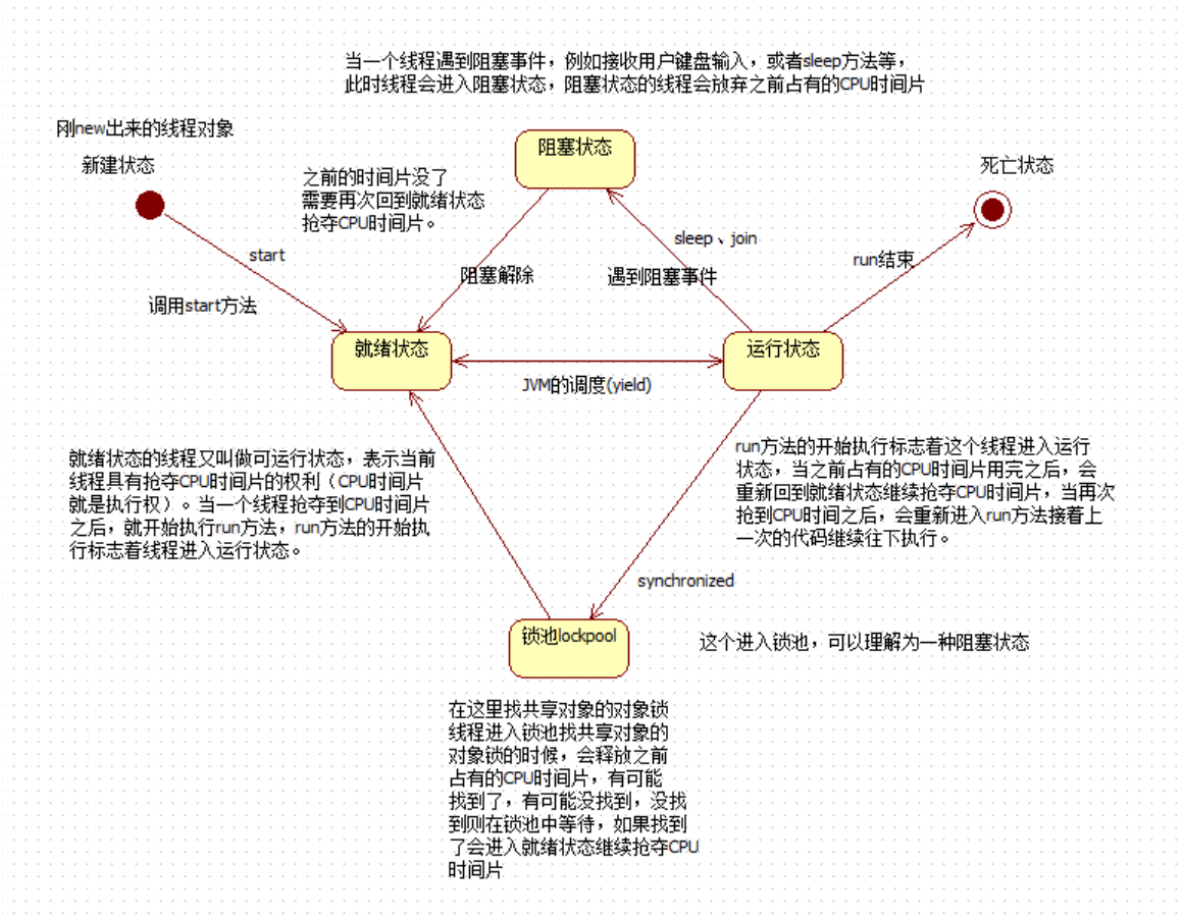
wait、notify是object的

内存

- 线程A和线程B，`堆内存` 和 `方法区` 内存共享。但是 `栈内存` 独立，一个线程一个栈。

- Java 提供了一个**系统线程**来管理内存的分配，程序员没必要创建一个线程去管理内存的分配。

运行过程



线程名

```
Thread currentThread = Thread.currentThread();//currentThread获得当前线程对象，前面固定跟Thread.currentThread
System.out.println(currentThread.getName() + "-->" + i);//getName获得当前线程名
currentThread.setName("tRY")//修改线程名
```

创建线程

一共有两个办法，第一个是**extending Thread class**，第二个是**implementing Runnable interface**。但是实现Runnable接口比继承Thread类创建线程的方式扩展性更好。

方法一

- 编写一个类，直接 继承 Thread类，重写 `run方法`。但是如果已经有父类就不能这样继承了，java只能继承一个类。
- run方法一定是void类型的。

1. 怎么创建线程对象？ **new**继承线程的类。
2. 怎么启动线程呢？ 调用线程对象的 `start()` 方法。

```
public class ThreadTest02 {
    public static void main(String[] args) {
```

```
        MyThread t = new MyThread();
        // 启动线程
        //t.run(); // 不会启动线程，不会分配新的分支栈。（这种方式就是单纯调用了函数。）
        t.start();//这时才启动子线程，并且立刻执行run
        // 这里的代码还是运行在主线程中。
        for(int i = 0; i < 1000; i++){
            System.out.println("主线程--->" + i);
        }
    }
}

class MyThread extends Thread {
    @Override
    public void run() {
        // 编写程序，这段程序运行在分支线程中（分支栈）。
        for(int i = 0; i < 1000; i++){
            System.out.println("分支线程--->" + i);
        }
    }
}
```

方法二

- 编写一个类，实现 `java.lang.Runnable` 接口，实现 `run方法`。2-9
    1. 怎么创建线程对象？ **new**线程类传入可运行的类/接口。
    2. 怎么启动线程呢？ 调用线程对象的 `start()` 方法。
    3. 更为实用，可以去继承其他的类

```
public class ThreadTest03 {
    public static void main(String[] args) {
        Thread t = new Thread(new MyRunnable());
        // 启动线程
        t.start();

        for(int i = 0; i < 100; i++){
            System.out.println("主线程--->" + i);
        }
    }
}

// 这并不是一个线程类，是一个可运行的类。它还不是一个线程。
class MyRunnable implements Runnable {
    @Override
    public void run() {
        for(int i = 0; i < 100; i++){
            System.out.println("分支线程--->" + i);
        }
    }
}
```

```
public class ThreadTest04 {
    public static void main(String[] args) {
        // 创建线程对象，采用匿名内部类方式。
        Thread t = new Thread(new Runnable(){//这里就是new的时候直接重写run
```

```
            @Override
            public void run() {
                for(int i = 0; i < 100; i++){
                    System.out.println("t线程---> " + i);
                }
            }
        });

        // 启动线程
        t.start();

        for(int i = 0; i < 100; i++){
            System.out.println("main线程---> " + i);
        }
    }
}
```

关闭线程

- 如果线程死亡，它便不能运行。
- 使用开关，可以关闭线程。

```
public class ThreadTest10 {
    public static void main(String[] args) {
        MyRunable4 r = new MyRunable4();
        Thread t = new Thread(r);
        t.setName("t");
        t.start();

        // 模拟5秒
        try {
            Thread.sleep(5000);
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
        // 终止线程
        // 你想要什么时候终止t的执行，那么你把标记修改为false，就结束了。
        r.run = false;
    }
}

class MyRunable4 implements Runnable {

    // 打一个布尔标记
    boolean run = true;

    @Override
    public void run() {
        for (int i = 0; i < 10; i++){
            if(run){
                System.out.println(Thread.currentThread().getName() + "--->" +
i);
                try {
                    Thread.sleep(1000);
```

```
                } catch (InterruptedException e) {
                    e.printStackTrace();
                }
            }else{
                // return就结束了，你在结束之前还有什么没保存的。
                // 在这里可以保存呀。
                //save....

                //终止当前线程
                return;
            }
        }
    }
}
```

- 调用stop（），可以停止线程，但是由于不安全已被放弃。

调度知识

设置线程优先级

方法名	作用
int getPriority()	获得线程优先级
void setPriority(int newPriority)	设置线程优先级
static void yield()	让位方法，当前线程暂停，回到就绪状态，让给其它线程。
void join()	将一个线程合并到当前线程中，当前线程受阻塞，加入的线程执行直到结束

- 最低优先级1，默认优先级是5，最高优先级10
- 优先级比较高的获取CPU时间片可能会多一些，**但不必然多。**所以，在Java中，高优先级的可运行线程不必然会抢占低优先级线程
- main线程的默认优先级也是5，不一定就是最大的。

休眠(sleep)，让步(yield)，插队(join)

sleep休眠与interrupt中断sleep

- static void sleep(long millis)，让当前线程**休眠**millis毫秒，它可以让当前线程进入休眠，进入“**阻塞状态**”，**放弃占有CPU时间片**，让给其它线程使用（**不是终止线程**），sleep方法不会释放锁
- 看下面的例子，以下会睡眠的是主线程。如果想让t线程睡眠，sleep必须写在t的run内

```
Thread t = new Thread() {
    void run() {
        for (;;) System.out.println();
    }
};
t.start();
t.sleep(1000);
```

- interrupt：在一个线程中调用另一个线程的interrupt()方法，即会向那个线程发出信号，线程中断，停止sleep。被中止sleep的线程不会报错误信号。

```
public class InterruptionInJava implements Runnable{
    private volatile static boolean on = false;
    public static void main(String[] args) throws InterruptedException {
        Thread testThread = new Thread(new
InterruptionInJava(),"InterruptionInJava");
        //start thread
        testThread.start();
        Thread.sleep(1000);
        InterruptionInJava.on = true;
        testThread.interrupt();

        System.out.println("main end");

    }

    @Override
    public void run() {
        while(!on){
            try {
                Thread.sleep(10000000);
            } catch (InterruptedException e) {
                System.out.println("caught exception right now: "+e);
            }
        }
    }
}
```

yield ()

- yield()方法不是阻塞方法。让当前线程让步，让给其它线程使用。yield()方法的执行会让当前线程从“**运行状态**”回到“**就绪状态**”。在回到就绪之后，让具有同等优先级的线程执行，但是有可能还会再次抢到。

join ()

- join方法的主要作用就是同步，它可以使得线程之间的并行执行变为串行执行，让join的线程必须先执行完才能正常执行此时的线程。并且，join方法必须在线程start方法调用之后调用才有意义，在前这条语句将毫无作用

```
public class JoinTest {
    public static void main(String [] args) throws InterruptedException {
        ThreadJoinTest t1 = new ThreadJoinTest("小明");
        ThreadJoinTest t2 = new ThreadJoinTest("小东");
```

```
        t1.start();
        /**join的意思是使得放弃当前线程的执行，并返回对应的线程，例如下面代码的意思就是：
         程序在main线程中调用t1线程的join方法，则main线程放弃cpu控制权，并返回t1线程继续执行直到线程t1执行完毕
         所以结果是t1线程执行完后，才到主线程执行，相当于在main线程中同步t1线程，t1执行完了，main线程才有执行的机会
         */
        /**join方法可以传递参数，join(10)表示main线程会等待t1线程10毫秒，10毫秒过去后，
         * main线程和t1线程之间执行顺序由串行执行变为普通的并行执行
         */
        t1.join(10);
        t2.start();
    }

}
class ThreadJoinTest extends Thread{
    public ThreadJoinTest(String name){
        super(name);
    }
    @Override
    public void run(){
        for(int i=0;i<1000;i++){
            System.out.println(this.getName() + ":" + i);
        }
    }
}
```

suspend与resume

suspend()

- 当某个线程的suspend()方法被调用时，该线程会被挂起。如果该线程占有了锁，则它不会释放锁。即，线程在挂起的状态下还持有锁。
- suspend之后的代码都不会执行

resume（）

- 恢复suspend的线程

死锁

异步编程模型：

线程t1和线程t2，各自执行各自的，t1不管t2，t2不管t1，谁也不需要等谁，这种编程模型叫做：异步编程模型。

其实就是：多线程并发（效率较高。）

异步就是并发。

同步编程模型：

线程t1和线程t2，在线程t1执行的时候，必须等待t2线程执行结束，或者说在t2线程执行的时候，必须等待t1线程执行结束，两个线程之间发生了等待关系，这就是同步编程模型。

效率较低。线程排队执行。

同步就是排队。

synchronized

定义

- synchronized关键字可以作为函数的修饰符，也可作为函数内的语句，synchronized可作用于instance变量、object reference（对象引用）、static函数和class literals(类名称字面常量)身上。
- 无论synchronized关键字加在方法上还是对象上，它取得的锁都是对象，而不是把一段代码或函数当作锁。
- 每个对象只有一个锁（lock）与之相关联。

作用

- 某个对象实例内，synchronized a Method(){}可以防止多个线程同时访问这个对象的synchronized方法（如果一个对象有多个synchronized方法，只要一个线程访问了其中的一个synchronized方法，其它线程不能同时访问这个对象中任何一个synchronized方法）。
- 某个类的范围，synchronized static a StaticMethod{}防止多个线程同时访问这个类中的synchronized static 方法。它可以对类的所有对象实例起作用

```
public synchronized void withdraw(double money){
        double before = this.getBalance();
        double after = before - money;
        try {
            Thread.sleep(1000);
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
        this.setBalance(after);
    }
```

变量知识

1. 实例变量在堆中，堆只有1个。
2. 静态变量在方法区中，方法区只有1个。
3. 临时变量在栈里，每个线程自己有一个栈
4. 堆和方法区都是多线程共享的，所以可能存在线程安全问题。

总结：

- 局部变量+常量：不会有线程安全问题。
- 成员变量（实例+静态）：可能会有线程安全问题

变量选择

- 如果使用局部变量的话，**StringBuilder**。因为局部变量不存在线程安全问题。反之，使用StringBuffer，它线程安全。
- ArrayList是非线程安全的。
- Vector是线程安全的。
- HashMap HashSet是非线程安全的。
- Hashtable是线程安全的

特殊：Object类的wait()、notify()、notifyAll()方法

方法名	作用
void wait()	让活动在当前对象的线程无限等待（释放之前占有的锁）
void notify()	唤醒当前对象正在等待的线程（只提示唤醒，不会释放锁）
void notifyAll()	唤醒当前对象全部正在等待的线程（只提示唤醒，不会释放锁）

1. wait和notify方法**不是线程对象的方法**，是java中任何一个java对象都有的方法，因为这两个方法是 **Object类**中自带 的。

wait方法和notify方法不是通过线程对象调用

```
Object o = new Object();

o.wait();
//o上线程等待
Object o = new Object();

o.notify();
//唤醒正在o对象上等待的线程。
Object o = new Object();

o.notifyAll();
//这个方法是唤醒o对象上处于等待的所有线程。
```

2. wait和notify实现生产者-消费者模式

```
public class ThreadTest16 {
    public static void main(String[] args) {
        // 创建1个仓库对象，共享的。
        List list = new ArrayList();
        // 创建两个线程对象
        // 生产者线程
        Thread t1 = new Thread(new Producer(list));
        // 消费者线程
        Thread t2 = new Thread(new Consumer(list));

        t1.setName("生产者线程");
        t2.setName("消费者线程");

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

// 生产线程
class Producer implements Runnable {
    // 仓库
    private List list;

    public Producer(List list) {
        this.list = list;
    }
```

```
    @Override
    public void run() {
        // 一直生产（使用死循环来模拟一直生产）
        while(true){
            // 给仓库对象list加锁。
            synchronized (list){
                if(list.size() > 0){ // 大于0，说明仓库中已经有1个元素了。
                    try {
                        // 当前线程进入等待状态，并且释放Producer之前占有的list集合的
锁。
                        list.wait();
                    } catch (InterruptedException e) {
                        e.printStackTrace();
                    }
                }
                // 程序能够执行到这里说明仓库是空的，可以生产
                Object obj = new Object();
                list.add(obj);
                System.out.println(Thread.currentThread().getName() + "--->" +
obj);
                // 唤醒消费者进行消费
                list.notifyAll();
            }
        }
    }
}

// 消费线程
class Consumer implements Runnable {
    // 仓库
    private List list;

    public Consumer(List list) {
        this.list = list;
    }

    @Override
    public void run() {
        // 一直消费
        while(true){
            synchronized (list) {
                if(list.size() == 0){
                    try {
                        // 仓库已经空了。
                        // 消费者线程等待，释放掉list集合的锁
                        list.wait();
                    } catch (InterruptedException e) {
                        e.printStackTrace();
                    }
                }
                // 程序能够执行到此处说明仓库中有数据，进行消费。
                Object obj = list.remove(0);
                System.out.println(Thread.currentThread().getName() + "--->" +
obj);
                // 唤醒生产者生产。
                list.notifyAll();
            }
        }
```

```
    }
}
```

Socket知识

格式

- Socket套接字中的端口号是一个2个字节的整数，16位，0 ~ 65535（计网中划分了端口范围的公有私有）
- 通信时，通过Socket套接字的getInputStream()返回此套接字的输入流，getOutputStream()返回套接字的输出流。通信的双方通过输入、输出流读写数据
- shutdownInput和shutdownOutput的作用是关闭相应的输入、输出流，不关闭网络连接
- TCP协议开发网络程序时，需要使用2个类SocketServer Socket
- 使用UDP协议开发网络程序时，需要使用2个类DatagramSocket DatagramPacket。其中，使用DatagramPacket把要发送的数据打包，并且，用该类对象接收数据
- DatagramSocket的发生阻塞

使用UDP协议进行通信，在程序运行时，DatagramSocket的哪个方法会发生阻塞？（）

- ○ A. send()
- ◉ B. receive()
- ○ C. close()
- ○ D. connect()

数据库知识

- ResultSet接口通过列名或者是第几列来获取数据
- 三种Java事务：

  1、JDBC事务控制的局限性在一个数据库连接内，但是其使用简单。
  2、JTA事务的功能强大，事务可以跨越多个数据库或多个DAO，使用也比较复杂。
  3、容器事务，主要指的是J2EE应用服务器提供的事务管理，局限于EJB应用使用。
- JDBC是一套用于执行SQL语句的API
- PrepareStatement对象继承了Statement接口，而且访问数据库的速度比Statement对象快。
- 在编写JDBC程序时，必须要把所使用的数据库驱动程序或类库加载到项目的（classpath）变量所指定的类路径下。

函数式编程

常见编程范式

命令式编程：

命令式编程的主要思想是关注计算机执行的步骤，即一步一步告诉计算机先做什么再做什么。代表语言有：C, C++, Java, Javascript，BASIC,Ruby等多为老牌语言

声明式编程：

声明式编程是以数据结构的形式来表达程序执行的逻辑。它的主要思想是告诉计算机应该做什么，但不指定具体要怎么做。代表语言有：SQL，HTML，CSS

函数式编程：

函数式编程将函数作为编程中的“一等公民”，关注于流程而非具体实现。可以将函数作为参数或返回值。所有数据的操作都通过函数来实现。可以理解为数学中的函数。较新的语言基本上追求语法上的简洁基本都有支持。代表语言有：JAVA（8以上），js（ES6），C#，Scala，python等

介绍

- 函数式编程有它自己适合的应用场景，比如科学计算、数据处理、统计分析等。在这些领域，程序往往比较容易用数学表达式来表示，比起非函数式编程，实现同样的功能，函数式编程可以用很少的代码就能搞定。
- 函数式编程的编程单元是无状态函数。就是说，函数的执行结果只与入参有关，跟其他任何外部变量无关。同样的入参，不管怎么执行，得到的结果都是一样的
- **Stream类**：Stream类上的操作有两种：中间操作和终止操作。中间操作返回的仍然是Stream类对象，而终止操作返回的是确定的值结果。
- **Lambda表达式**：Lambda表达式包括三部分：输入、函数体、输出。以下的是标准写法，除此之外，还有很多简化写法。比如，如果输入参数只有一个，可以省略 ()，直接写成 a->{...}；如果没有入参，可以直接将输入和箭头都省略掉，只保留函数体；如果函数体只有一个语句，那可以将{}省略掉；如果函数没有返回值，return语句就可以不用写了。

```
(a, b) -> { 语句1；语句2；...; return 输出; } //a,b是输入参数
```