Java 复习

Tags	
Last edited time	@January 6, 2024 9:49 PM

一、Java入门

JDK的发展

- 2004：JDK1.5（JDK5.0）
- 2014：JDK8.0

Java工作方式/开发环境

- Java虚拟机（JVM）
- Java开发工具包（JDK）
- Java运行环境（JRE）

Java程序结构

```
public class HelloWorld{
    public static void main(String[] args){
        System.out.println("Hello World");
    }
}
```

▼ **注意：**

- **一个源文件中最多只能有一个public类。其它类的个数不限，如果源文件包含一个public类，则文件名必须按该类名命名。**
- **关键字static允许在不创建类的实例的情况下，调用main函数**

二、Java基础语法

变量

- 命名规则：首字符（**字母，下划线，$**）

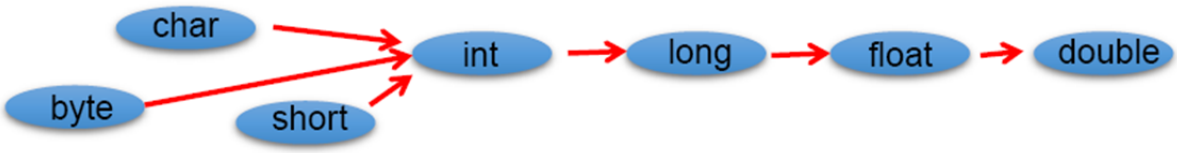
数据类型

- 基本数据类型
- 引用数据类型：**类、接口、数组、字符串**

数据类型转换

- 自动类型转换
  - 相互转换的两种数据类型要兼容，数值类型（整型和浮点型）互相兼容
  - **低精度的值可以直接赋给高精度变量**，直接转换为高精度

> **byte < short < char < int < long < float < double**



  - **byte、short、char之间不会相互转换，三者在计算时首先转换为int类型！**
  - **高精度的值不可以直接赋给低精度变量，需进行强制转换**
  - 不同类型变量混合运算后的结果是**精度最高的类型**
- 强制类型转换
  - **（类型名）表达式**

运算符与表达式

- &与&&，|与||之间的区别（&&，||：**前项表达式满足不运行后项**）
- 运算符优先级

优先级	运　算　符

1	括号: ()和[]
2	一元运算符：-、++（前置）、--（前置）、！
3	算术运算符：(*、/、%)、(+和-)
4	关系运算符：(>、>=、<、<=)、(==和!=)
5	逻辑运算符：(&)、(|)、(&&)和(||)
6	条件运算符：? :
7	赋值运算符：=、*=、/=、%=、+=和-=

- 运算符的结合性
  - Java 语言中除了单目运算符、赋值运算符和三目运算符是从右向左结合之外，其他运算符均是从左向右结合。
  - 当有多中运算符参与运算时，先考虑优先级，只有在优先级相同时，再来根据结合性决定运算顺序。

数组

- 声明：**数据类型[ ] 数组名;** 或者 **数据类型 数组名[ ];**
- 分配空间：**数据类型 数组名 = new 数据类型[大小] ;**
- 赋值
  - **int**[ ] score = {89, 79, 76};
  - **int**[ ] score = **new int**[ ]{89, 79, 76};
- 属性
  - length

三、面向对象编程

- 面向对象的特征：**抽象、封装、继承、多态**

类与对象

- 创建对象：**类名 对象名 = new 类名();**

函数重载

- 定义：若两个函数的**名称相同**，但**参数不一致**，则可以说一个函数是另一个函数的重载。**不能根据方法的返回值来区分重载方法**
- **方法的重载可以发生在子类中**

构造函数

- 没有显式定义构造函数，系统自动隐含定义无参构造函数，将成员变量默认初始化为0

静态变量（类变量）

static修饰符

- static修饰的成员被所有的对象所共享；
- static修饰的成员多了一种调用方式，就可以直接被类名所调用。(**类名.静态成员**)

静态函数

- **静态函数不能引用实例函数**
- **静态函数不能访问实例变量**

静态代码块

```
static {
//代码块
}
```

- 只需要在项目启动时执行一次的代码，如，对所有对象的共同信息进行一次性初始化；
- 静态代码块属于类，只需加载类就能运行；
- 静态代码块**不能存在任何方法体中**；
- 静态代码块**不能访问普通变量**；
- 只在类加载时运行一次，且优先于各种其他代码块以及构造函数。

> **执行顺序：静态代码块 > 构造代码块 > 构造函数 > 普通代码块**

封装

- 访问控制修饰符
  - **private：**表示私有，只能被**该类自身**方法访问
  - **public：**表示公有，可被该**项目的所有类**访问
  - **default：**缺省修饰符，表示只能被**同一个包中的类**访问和引用
  - **protected：**表示受保护，可被**同一包中的类以及其他包中该类的子类**访问
- this关键字
  - 调用属性/函数/构造函数：**this.xxx；this()；**
  - 如果本类没有，则从父类中继续查找
- 包
  1. 为什么要用包？
  - **允许类组成较小的单元（类似文件夹），易于找到和使用相应的文件**
  - **防止命名冲突**
  - **更好的保护类、属性和方法**
  2. 包的创建
  - **使用package声明包，以分号结尾**
  - **若有包的声明，一定作为Java源代码的第一条语句**

```
package com.hz.classandobject;   //声明包
public class AccpSchool{
    //……
    public String toString(){
       //……
    }
```

  - 命名规范
    - 包名由小写字母组成，不能以圆点开头或结尾

    - 包名之前最好加上唯一的前缀，通常使用组织倒置的网络域名。如：域名javagroup.net

3. 包的使用

- **包名 . 类名**
- **import关键字**
    - import PackageName.*; //导入包中的所有类
    - import PackageName.ClassaName; //导入包中的指定类

- 内部类
    - **成员内部类可以无条件访问外部类的所有成员属性和成员方法（包括private成员和静态成员）**
    - **外部类中如果要访问成员内部类的成员，必须先创建一个成员内部类的对象，再通过指向这个对象的引用来访问**
    - **成员内部类是依附外部类而存在的，所以，如果要创建成员内部类的对象，前提是必须存在一个外部类的对象并通过该外部类对象来创建**

```
class Outter {
    private Inner inner = null;
    public Outter() { ... ... }
    public Inner getInnerInstance() {
        if(inner == null)
            inner = new Inner();
        return inner;
    }
    class Inner {
        public Inner() { ... ... }
    }
}

public class Test {
    public static void main(String[] args)  {
        //第一种方式：
        Outter outter = new Outter();
```

```
        Outter.Inner inner = outter.new Inner();

        //第二种方式：
        Outter.Inner inner1 = outter.getInnerInstance();
    }
}
```

- **内部类可以拥有private、protected、public及包访问权限。**
- **当内部类和外部类拥有同名的成员变量或者方法时，会发生隐藏现象，即默认情况下访问的是成员内部类的成员。若要访问外部类的同名成员，需以下面形式进行访问：**

```
外部类.this.成员变量
外部类.this.成员方法
```

- 匿名内部类
  - 匿名内部类不能定义任何静态成员、方法；
  - 匿名内部类中的方法不能是抽象的；
  - 匿名内部类必须实现接口或抽象父类的所有抽象方法；
  - 当匿名内部类和外部类有同名变量（方法）时，默认访问的是匿名内部类的变量（方法），要访问外部类的变量（方法）则需要加上外部类的类名。

继承

- 继承分为单继承和多重继承。Java中的类只支持单继承，而多继承的功能是通过接口（interface）来间接实现的。
- 语法格式

```
<修饰词> class <派生类名> extends <父类名> {
    <派生类成员表>
}
```

- 继承的特点

    - 子类**只能直接继承一个父类**，而不允许继承多个父类，Java的类支持单继承和多层继承，不允许多重继承。
    - 有些父类成员不能继承
        - private成员
        - 子类与父类不在同包，使用默认访问权限的成员
        - 构造函数（**只可调用，不能继承！**）
    - **子类自己定义了构造函数，则在创建新对象时，它将先执行继承自父类的无参数构造函数，然后再执行自己的构造函数**
- super关键字（使用**super**关键字来调用父类中的指定操作。）
    - super.xxx；
    - 当子父类出现同名成员时，可用super表明调用的是父类成员
- **重写（override）【多态性】**

  保持与父类完全相同的方法头部声明，即应与父亲有**完全相同的方法名和参数列表**。
    - 重写以继承为前提，父类成员方法只能被其子类重写
    - **子类重写的方法使用的访问权限不能小于父类被重写的方法的访问权限**
    - 返回类型与被重写方法的返回类型可以不相同，但是必须是父类返回值的派生类
        - 父类被重写方法返回值类型是void，子类重写方法的返回值类型也应是void
        - 父类被重写方法返回值类型是基本数据类型，子类重写方法的返回值类型是相同的基本数据类型
    - **声明为private和final的方法不能被重写**
    - 声明为 static 的方法不能被重写，但可被再次声明
    - **构造方法不能被重写**

多态

多态是同一个行为具有多个不同表现形式或形态的能力。

- Java中的多态性
    - 编译时多态

主要指**方法的重载**，它根据参数列表的不同来区分不同的方法。通过编译后变成不同的方法。

- 运行时多态

  即**动态绑定**，指在执行期间（非编译期间）判断引用对象的实际类型，根据实际类型判断并调用相应的属性和方法。主要用于继承父类和实现接口时，父类引用指向子类对象。

```
public static void main(String[] args) {
    Printer p = new ColorPrinter();
    p.print();
    p = new BlackPrinter(); // 更换对象
    p.print();
}
```

- 多态的实现
  - **子类对象的多态性，并不适用于属性。因此输出的是父类对象的属性值。**（当子类和父类有相同属性时，父类还是会执行自己所拥有的属性，若父类中没有的属性子类中有，当父类对象指向子类引用时(向上转型)，在编译时期就会报错）
  - 父类引用指向子类对象时，只会**调用父类的静态方法**。**所以，静态方法并不具有多态性。**
- 多态的转型
  - 向上转型

    父类类型 变量名 = new 子类类型();

    即子类对象p2自动转型为父类对象，使得该对象可访问子类从父类继承或重写的方法。
  - 向下转型

    子类类型 变量名 =（子类类型） 父类类型的变量;

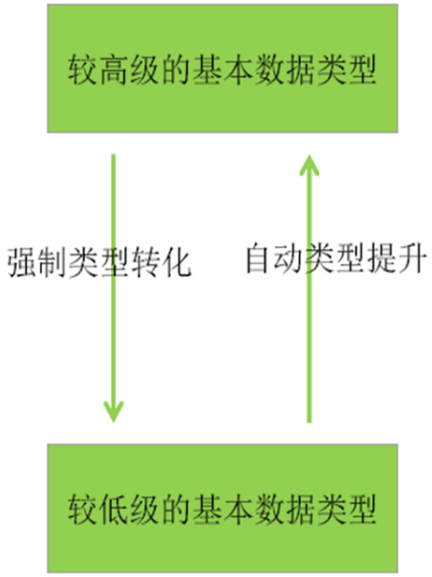
    向下转型后，可以调用子类中特有的方法

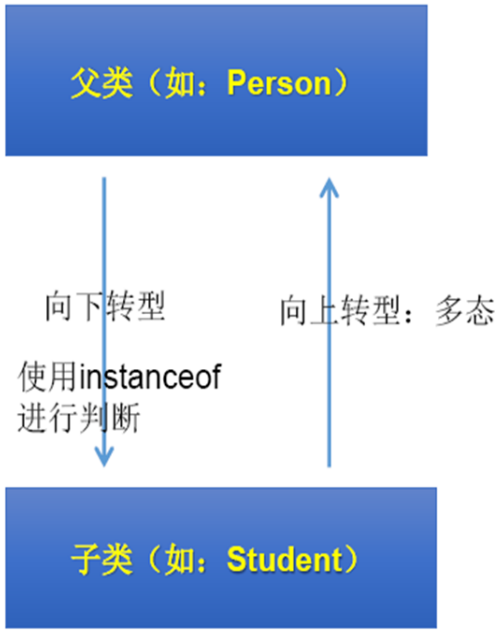
    向下转型一般连着向上转型
  - instanceof关键字

- 对象x **instanceof** 类A
- **作用：**判断某个对象是否属于某种数据类型，返回值为boolean型。

对Java对象的强制类型转换称为**造型：**

- 从子类到父类的类型转换可以自动进行（实际就是多态）
- 从父类到子类的类型转换须通过造型(强制类型转换)实现
- 无继承关系的引用类型间的转换是非法的
- 在造型前可以使用instanceof操作符测试一个对象的类型





抽象

- 语法

  abstract class <类名> {

  abstract <返回值类型> <方法名> (参数列表);

  }

- 使用规则
  - 抽象类和抽象方法都要使用 abstract 关键字声明；
  - 抽象类不能实例化，抽象类是用来被继承的；

- 抽象类的子类必须重写父类的抽象方法，并提供方法体。若没有重写全部的抽象方法，则仍为抽象类；
- Abstract不能修饰变量、代码块、构造函数；
- Abstract不能修饰私有方法、静态方法、final方法、final类；

▼ **注意：**

- **拥有一个抽象方法并不是构建一个抽象类充分必要条件。**
- **普通类中不允许有抽象方法。**

final关键字

- 使用方法
  - 使用final关键字可以将变量声明为**常量**，一旦被赋值之后就**不能再次修改它的值**。常量通常使用大写字母表示，并**在声明时进行初始化**。
  - final关键字可以用于变量、方法和类。如果一个类被声明为final，则表示它是一个**最终类，不能被继承**。如果一个方法被声明为final，则表示它是**最终方法，不能被覆盖**。
- final标记的类
  - String类、System类、StringBuffer类
- static final修饰属性：全局常量
- 与static比较
  - **static作用于成员变量用来表示只保存一份副本，而final的作用是用来保证变量不可变。**

接口

- 定义

  [访问修饰符] **interface** 接口名称 [extends 其他的接口名] {

  // 声明变量

  // 抽象方法

  }

- 特点

- 所有成员变量都默认是**public static final**修饰（全局常量）
- 所有抽象方法都默认是由**public abstract**修饰
- 没有构造函数
- 支持多继承机制

- 接口的实现
  - 一个类通过**继承接口**的方式，来继承接口的抽象方法。
  - 接口中的方法只能由实现接口的类来实现。
  - 除非实现接口的类是抽象类，否则**该类要定义接口中的所有方法**。
  - 一个类可以**实现多个接口**，接口也可以继承其它接口。
  - 与继承关系类似，接口与实现类之间存在**多态性**。

```
[public] interface interface_name [extends interface1_name[,
    //接口体，其中可以定义常量和声明方法
}

class SubClass implements InterfaceA [InterfaceB, InterfaceC,
      //类主体
}

class SubClass extends superclass implements InterfaceA{
      //类主体
}
```

▼ **注意：**

- **无论何时实现一个由接口定义的方法，它都必须实现为 public，因为接口中的所有成员都显式声明为 public**
- **接口中的成员变量默认为static final,必须在定义时进行初始化！**

- 接口和类的区别
  - 接口不能用于实例化对象。
  - 接口没有构造方法。

- 接口中所有的方法必须是抽象方法。
- 接口不能包含成员变量，除了 static 和 final 变量。
- 接口不是被类继承了，而是要被类实现。
- 接口支持多继承。

Object类

Object类存储在java.lang包中，是所有java类(Object类除外)的终极父类。

如果在类的声明中未使用extends关键字显式指明其父类，则默认父类为java.lang.Object类。

- clone()——深拷贝

```
Person p1 = p; //引用的复制。p和p1只是对象的引用，具有相同的地址！
Person p1 = (Person) p.clone(); //创建了一个新的对象，二者具有不同
```

- String toString()
  - 在进行String与其它类型数据的连接操作时，会自动调用toString()方法
  - 基本类型数据转换为String类型时会调用对应包装类的toString()方法
- boolean equals(Object obj)
  - 调用方法的对象和传入的对象的**引用是否相等**。即用来**判断两个对象引用的是否是同一对象**。

四、常用API

Scanner类

- **hasNextInt()：判断用户从键盘输入的字符是否是合法的数字**

String类

- 特点
  - java.lang.String是一个final类，也不能有子类

- 属于常量，用""引起来表示。**创建后其值不能更改**

- 方法

序号	方法	功能描述
1	public char charAt(int index)	返回index所指位置的字符
2	public String concat(String str)	将两字符串连接
3	public boolean startsWith(String str)/endsWith(String str)	测试字符串是否以str开始/结尾
4	public boolean equals(Object obj)	比较两对象
5	public char[] getBytes()/getBytes(String str)	将字符串转换成字符数组返回
6	public int indexOf(String str)/indexOf(String str, int fromIndex)	返回字符串在串中位置
7	public int length()	返回字符串的长度
8	public String replace(char old ,char new)	将old用new替代
9	public char[] toCharArray ()	将字符串转换成字符数组
10	public String substring(int start)/substring(int start, int end)	截取字符串内某段并返回
11	public String toLowerCase()/toUpperCase()	转小写/大写
12	public String trim()	去掉两边空格
13	public static String valueOf(各种类型)	将各种类型转为字符串

  - **public int compareTo(String anotherString)**

该方法是对字符串内容按字典顺序进行大小比较。若当前对象比参数大则返回正整数，反之返回负整数，相等返回0。

  - **public boolean equals(Object obj)**

    - 基本数据类型（也称原始数据类型）：byte,short,char,int,long,float,double,boolean。他们之间的比较，应用双等号（==）,**比较的是他们的值**。

        - 引用数据类型：当他们用（==）进行比较的时候，比较的是他们在内存中的存放地址（确切的说，是**堆内存**地址）。

    - **public String[] split(String str)**

    该方法将str作为分隔符进行字符串分解，分解后的字符串在字符串数组中返回。

StringBuffer类

- 特点
    - StringBuffer又称可变字符序列，是一个类似于 String 的字符串缓冲区;
    - 提供了String不支持的添加、插入、修改和删除之类的操作。
- 方法
    - **public int capacity()**

    该方法返回StringBuffer的当前容量。

五、多线程

基本概念

- 并行与并发
    - 并行：指两个或多个事件在**同一时刻**发生（同时发生）；
    - 并发：指两个或多个事件在**一个时间段内**发生。
- 程序、进程与线程
    - 程序：是为完成特定任务，用某种语言编写的一组计算机指令的集合，即指一段**静态的代码**；
    - 进程：指程序的一次执行活动，是一个动态的过程。它是**资源申请、调度和独立运行**的单位，使用系统中的运行资源，具有生命周期；
    - 线程：**进程内部的一个独立执行单元**（路径），**一个进程可以同时并发运行多个线程**。

Thread类

- **Thread.currentThread()：**
  - **是一个静态方法，返回正在执行的线程对象的引用。**
  - **输出线程对象将产生一个数组输出，其格式为：[线程名称，优先级别，线程组名]。**

```
public class MainThreadDemo  {
    public static void main(String args[])  {
        Thread t = Thread.currentThread();
        System.out.println("当前线程名称是: " + t.getName());
        System.out.println("输出当前线程: " + t);
    }
}
```

多线程的创建

- **继承java.lang.Thread 类**

```
class Thread1 extends Thread{
    // 重写Thread类的run()方法
    public void run(){
        for(int i=0; i<10; i++){
            System.out.println("play music  " + i);
        }
    }
}
public class ThreadDemo1  {
    public static void main(String args[])  {
        for(int i=0; i<10; i++){
            System.out.println("play games  " + i);
            if(i==3){
                Thread1 th1 = new Thread1();
                th1.start();
            }
        }
```

```
    }
}
```

> **多线程的机制实际上相当于CPU交替分配给不同的代码段来运行。默认情况下，分配工作由操作系统决定。所以，不仅启动顺序可能是随机的，运行完毕的顺序也不一定是启动的顺序**

- **实现Runnable接口**

```
class Runnable1 implements Runnable{
    //重写Runnable接口的run()方法
    public void run() {
        for(int i = 0 ; i < 10 ;i++){
            System.out.println("play music " + i);
        }
    }
}
public class ThreadDemo3 {
    public static void main(String[] args) {
        for(int i = 0 ; i < 10 ; i++){
            System.out.println(“play games " + i);
            if(i==3){
                Thread th1 = new Thread(new Runnable1());
                th1.start();
            }
        }
    }
}
```

- 两种方法的比较
  - 继承Thread类
    - 易受JAVA
    单继承特点的限制；
    - 每个对象都是一个线程，均具有各自的成员变量。

- 实现Runnable接口
  - 对象可自由地继承自另一个类；
  - 对象不是线程，须将其作为参数传入Thread对象才能运行；
  - 线程间可共享同一个接口实现类的对象，更适合多个线程**共享数据**的情况。

Thread类的常用方法

- **void start()**

  启动线程，并执行对象的run()方法。

- **void run()**

  线程在被调度时执行的操作。

- **static Thread currentThread()**

  返回当前线程。在Thread子类中就是this，通常用于主线程和Runnable实现类。

- **String getName()**

  返回线程的名称。

- **void setName(String name)**

  设置该线程名称。

- **static void yield()**

  暂停当前的执行线程，把执行机会让给优先级相同或更高的线程。

- **final void join()**

  当线程A中调用线程B的join() 方法时，线程A将被阻塞，直到线程B执行完为止，线程A才结束阻塞。

- **static void sleep(long millitime)**

  让当前线程“睡眠”指定的时长。在该时间内线程处于阻塞状态

- **final void setPriority(int priority))**

  设置当前线程的优先级。

- **final int getPriority()**

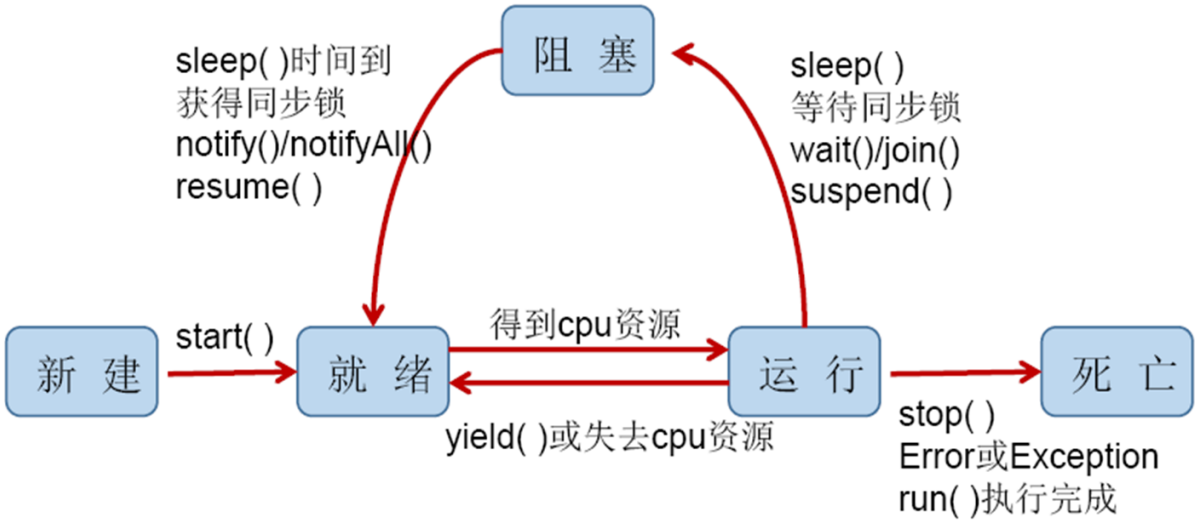
  获取当前线程的优先级。

- **final boolean isAlive()**

  判断当前线程是否存活。

线程的生命周期

线程从**创建、启动到终止**的整个过程，叫一个生命周期。在其间的任何一个时刻，线程总是处于某个特定的状态。



线程的调度策略

- **分时调度模型**

  指让所有的线程轮流获得CPU的使用权，并且平均分配每个线程占用CPU的时间片。

- **抢占式调度模型**

  让可运行迟中优先级高的线程优先占用CPU，而对于优先级相同的线程，随机选择一个线程使其占用CPU，当它失去了CPU的使用权后，再随机选择其它线程获取CPU的使用权。

> 线程的优先级用1~10之间的整数来表示，数字越大则表示优先级越高。

多线程的同步

- 在Java中通过**互斥锁标志synchronized关键字**的运用来实现同步。

```
//在要标志为同步的方法前加上synchronized关键字
synchronized void method( ) {
      //同步的方法
}
```

- **使用同步代码块**

  这种需要分离出来进行互斥的代码段被称为“**临界区**”。

```
//用synchronized来指定某个对象，此对象的锁被用来对花括号内的代码进行同步控
synchronized(object) {
      //同步的语句
}
```

- **使用Lock锁机制**

```
//创建ReentrantLock对象，在同步方法中，调用 lock() 方法后，
//将同步代码块置于try中，然后在finally中调用 unlock()以防死锁
class X {
    Lock lock = new ReentrantLock();
    public void m() {
        lock.lock();
        try {
            // ... method body
        } finally {
            lock.unlock();
        }
    }
}
```

- **synchronized 与Lock 的对比**
  - Lock是显式锁（**需手动开启和关闭锁**）；synchronized是隐式锁，出了作用域自动释放；

- Lock只有代码块锁，synchronized有代码块锁和方法锁；
- 使用Lock锁，JVM将花费较少的时间来调度线程，性能更好且具有更好的扩展性（提供更多的子类）；

> 优先使用顺序：**Lock** > **同步代码块**（已经进入了方法体，分配了相应资源）> **同步方法**（在方法体之外）

六、IO流

IO流的分类

- 字节流
  - 操作的数据单元是8位的字节。以InputStream、OutputStream作为抽象基类；
- 字符流
  - 操作的数据单元是16位的字符。以Writer、Reader作为抽象基类。

File类

java.io.File类：文件和文件目录路径的抽象表示形式，与平台无关。

```
File f1 = new File("E:\\a.txt");
System.out.println(f1);     // E:\a.txt

File f2 = new File("a.txt");
System.out.println(f2);     // a.txt

File file = new File("C:\\","a.txt");
System.out.println(file);       //c:\a.txt

File parent = new File("c:\\");
File file = new File(parent,"hello.java");
System.out.println(file);      //c:\hello.java
```

- 创建功能
  - public boolean createNewFile()
    创建文件。若文件存在，则不创建，返回false。
  - public boolean mkdir()
    创建文件目录。如果此文件目录存在，就不创建了。如果此文件目录的上层目录不存在，也不创建。
  - public boolean mkdirs()
    创建文件目录。如果上层文件目录不存在，一并创建。
- 删除功能
  - public boolean delete()
    删除文件或者文件夹。

节点流（文件流）的读写

```
//读取文件
public static void readFile() {
        System.out.println("===读取文件====");
        File src = new File("E:/xp/test/临时.txt");
        InputStream is = null;
        try {
```

```
            is = new FileInputStream(src);
            byte[] car = new byte[124];//缓冲数组
            int len = 0;//接收实际读取大小
            try {
                while (-1 != (len = is.read(car))) {
                    String info = new String(car, 0, len);
                    System.out.println(info);
                }
            } catch (IOException e) {
                e.printStackTrace();
                System.out.println("文件不存在");
            }
        } catch (FileNotFoundException e) {
            System.out.println("读取文件失败");
            e.printStackTrace();
        } finally {
            if (null != is) {
                try {
                    is.close();//close关闭
                } catch (IOException e) {
                    System.out.println("关闭文件输入流失败");
                    e.printStackTrace();
                }
            }
        }
    }

//写入文件
public static void writeFile() {
        System.out.println(“ == = 写出文件 ====");
        File src = new File("E:/xp/test/临时.txt");
        OutputStream os = null;
        try {
            os = new FileOutputStream(src, true);  //追加
            String str = “\nHello\r\n”;
            byte[] data = str.getByte();//字符串转换为字节数组
```

```
            os.write(data, data.length);
            os.flush();//强制刷新
        } catch (FileNotFoundException e) {
            System.out.println(“文件未找到");
                    e.printStackTrace();
        } catch (IOException e) {
            System.out.println(“文件写出失败");
                    e.printStackTrace();
        } finally {
            if (null != os) {
                try {
                    os.close();//close关闭
                } catch (IOException e) {
                    System.out.println(“关闭文件输出流失败");
                            e.printStackTrace();
                }
            }
        }
    }
```

- 完整代码

```
import java.io.*;

public class CopyFile {
    public static void main(String[] args) {
        int dt;
        FileInputStream fis;
        FileOutputStream fos;
        try {
            fis = new FileInputStream("c:\\file\\file1.doc");
        } catch (FileNotFoundException e) {
            System.out.println("源文件未找到");
            return;
        }
```

```
        try {
            fos = new FileOutputStream("c:\\file\\file2.doc");
        } catch (FileNotFoundException e) {
            System.out.println("目标文件打开失败");
            return;
        }
        try {
            while ((dt = fis.read()) != -1)
                fos.write(dt);
        } catch (IOException e) {
            System.out.println("文件读写出错");
        } finally {
            try {
                fis.close();
                fos.close();
            } catch (IOException e) {
                e.printStackTrace();
            } finally {
                System.out.println("文件读写完毕");
            }
        }
    }
}
```

标准输入输出流

示例代码：

```
import java.util.Scanner;
class MyIO {
    public static void main(String []args) {
        int count = 0;
        String str = "";
        Scanner rd = new Scanner(System.in);//JDK5提供Scanner
        System.out.println("请输入数据：");
        while(rd.hasNext()) {
```

```
            str = rd.next();         //读入字符串数据
            System.out.print(str);       //显示刚输入字符串
            count += str.length();       //统计以读入的字符数目
        }
        System.out.println("\n共输入了" + count + "个字符");
    }
}
```

- **next()与 nextLine()的区别**
  - next()
    - 一定要读取到有效字符后才可以结束输入
    - 对输入有效字符之前遇到的空白，next() 方法会自动将其去掉
    - 只有输入有效字符后才将其后输入的空白作为分隔符或者结束符
    - next() 不能得到带有空格的字符串
  - nextLine()
    - 以Enter为结束符,也就是说nextLine()方法返回的是输入回车之前的所有字符
    - 可以获得带空白的字符串