JAVA精简面试题

JAVA概述

何为编程?

编程就是让计算机为解决某个问题而使用某种程序设计语言编程程序代码，并最终得到结果的过程。

为了使计算机能够理解人的意图，人类就必须将需要解决的问题的思路，方法，和手段通过计算机能够理解的形式告诉计算机，使得计算机能够根据人的指令一步步的去工作，完成某种特定的任务。

这种人和计算机之间交流的过程就是编程。

什么是*Java*?

Java是一门面向对象编程语言，不经吸收了C++语言的各种优点，还摒弃了C++里难以理解的多继承，指针等概念，因此Java语言具有功能强大和简单易用两个特征。

Java语言作为静态面向对象编程语言的代表，极好的实现了面向对象的理论，允许程序员以优雅的思维方式进行复炸的编程。

*JDK1.5*之后的三大版本

JavaSE(J2SE, Java Platform Standard Edition, 标准版)

Java SE 以前称为 J2SE。它允许开发和部署在桌面、服务器、嵌入式环境和实时环境中使用的 Java 应用程序。Java SE 包含了支持 Java Web 服务开发的类，并为Java EE和Java ME提供基础。

Java EE（J2EE，Java 2 Platform Enterprise Edition，企业版）

Java EE 以前称为 J2EE。企业版本帮助开发和部署可移植、健壮、可伸缩且安全的服务器端Java 应用程序。Java EE 是在 Java SE 的基础上构建的，它提供 Web 服务、组件模型、管理和通信 API，可以用来实现企业级的面向服务体系结构（service-oriented architecture，SOA）和 Web2.0应用程序。2018年2月，Eclipse 宣布正式将 JavaEE 更名为 JakartaEE

Java ME（J2ME，Java 2 Platform Micro Edition，微型版）

Java ME 以前称为 J2ME。Java ME 为在移动设备和嵌入式设备（比如手机、PDA、电视机顶盒和打印机）上运行的应用程序提供一个健壮且灵活的环境。Java ME 包括灵活的用户界面、健壮的安全模型、许多内置的网络协议以及对可以动态下载的连网和离线应用程序的丰富支持。基于 Java ME 规范的应用程序只需编写一次，就可以用于许多设备，而且可以利用每个设备的本机功能。

*JVM, JRE*和*JDK*的关系

JVM：

Java Virtual Machine 是Java程序需要允许在虚拟机上，不同的平台有自己的虚拟机，因此Java语言可以实现跨平台。

JRE：

Java Runtime Environment 包括Java虚拟机和Java程序所需的核心类库等。

核心类库主要是java.lang包：

包含了允许Java程序必不可少的系统类。

如：

基本数据类型

基本数学函数

字符串处理

线程

异常处理类

系统缺省加载这个包

JDK：

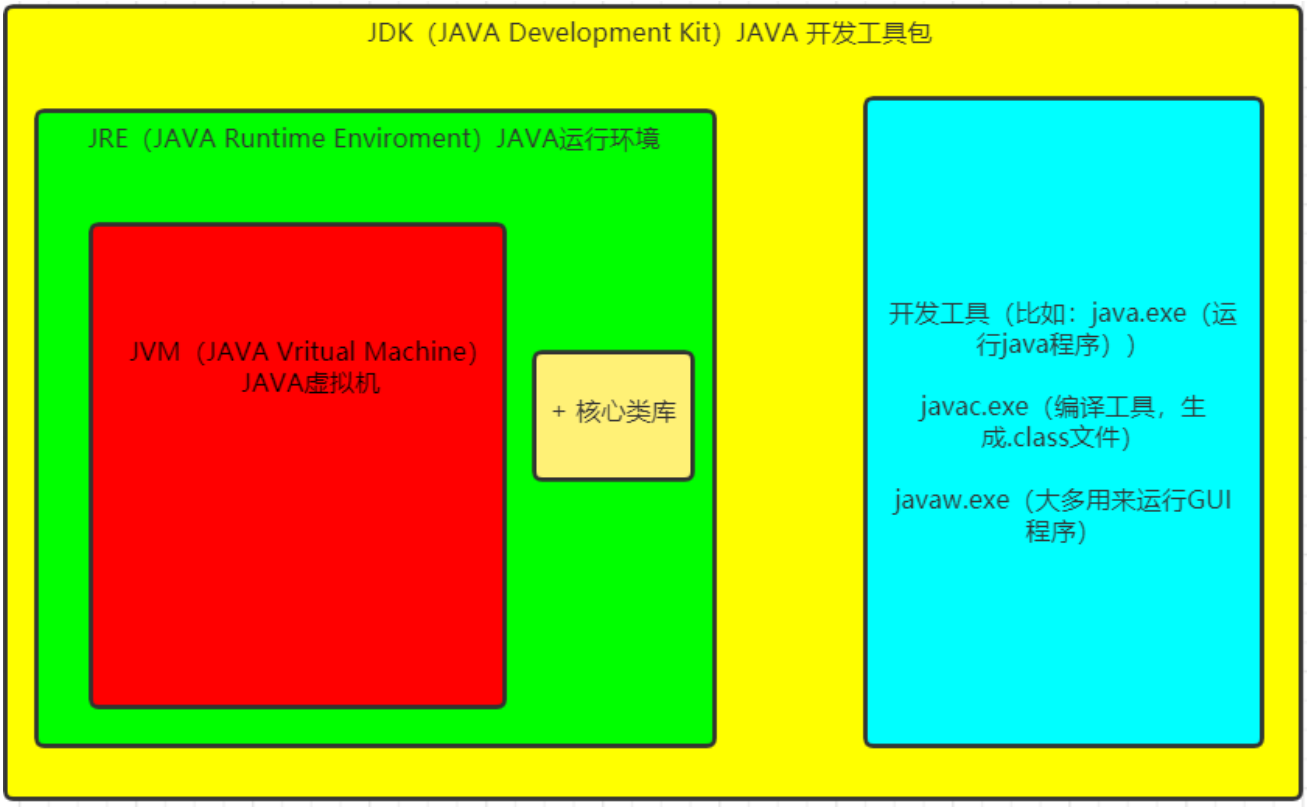
Java Development Kit是提供给Java开发人员使用的，其中包含了Java的开发工具，也包括了JRE。所以，安装了JDK，就无需单独安装JRE

其中的开发工具：

编译工具：javac.exe

打包工具：jar.exe

JVM&JRE&JDK关系图：



什么是跨平台性?

所谓跨平台性，是指java语言编写的程序，一次编译后，可以在多个系统平台上运行。

原理是什么?

实现原理：java程序是通过java虚拟机在系统平台上运行的，只要该系统可以安装想要的java虚拟机，该系统就可以运行java程序。

*Java*语言有哪些特点?

1.简单以学（Java语言的语法与C语言和C++语言很接近）

2.面向对象编程（封装， 继承， 多态）

3.平台无关性（Java虚拟机实现平台无关性）

4.支持网络编程并且很方便（Java语言诞生本身就是为了简化网络编程设计的）

5.支持多线程（多线程机制使应用程序在统一时间并行执行多项任务）

6.健壮性（Java语言的强类型机制，异常处理，垃圾的自动收集等）

7.安全性

采用字节码的好处

Java语言通过字节码的方式，在一定程度上解决了传统解释型语言执行效率低的问题，同时又保留了解释型可移植的特点。

所以，Java程序运行时比较高效而且，由于字节码并不专对一种特定的机器，音系，Java程序无序重新编译便可在多种不同的计算机上运行。

*Java*中的编译器和解释器：

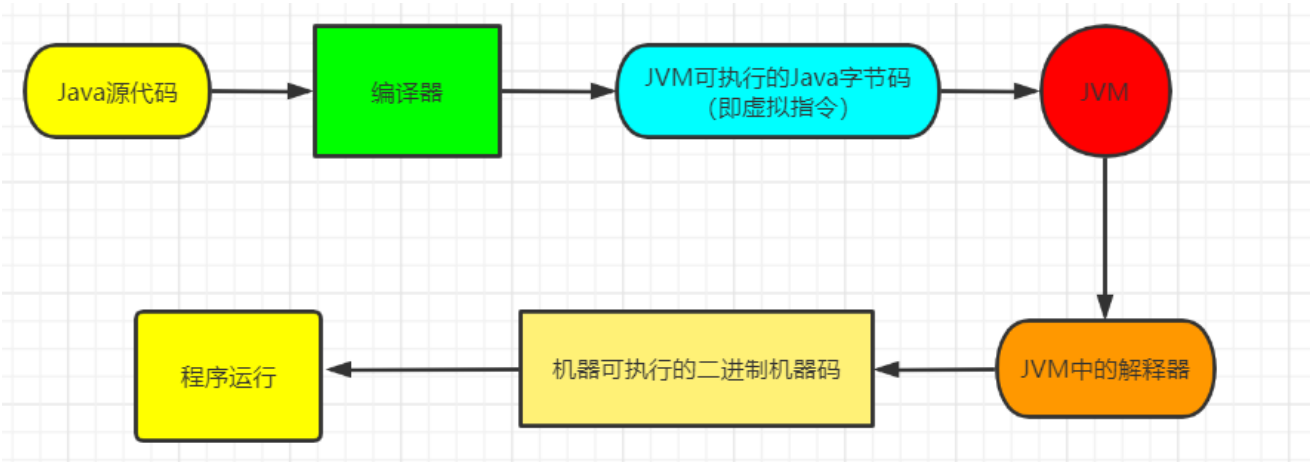
Java中引入了虚拟机的概念，即在机器和编译程序之间加入了一层抽象的虚拟机器。

这台虚拟的机器在任何平台上都是提供编译程序一个的共同的接口。

编译程序只需要面向虚拟机，生成虚拟机能够理解的代码，然后由解释器来将虚拟机代码转换为特定紫铜的机器码执行。

在Java中，这种提供虚拟机理解的代码叫做字节码（即扩展为.class的文件），它步面向任何特定的处理器，只面向虚拟机。每一种平台的解释器是不同的，但是实现的虚拟机是相同的。

Java源程序经过编译器编译后编程字节码，字节码由虚拟机解释执行，虚拟机将每一条要执行的字节码送给解释器，解释器将其翻译成特定机器上的机器码，然后在特定的机器上进行运行，这就是上面提到的Java的特定的编译与解释并存的解释。



什么是*JAVA*程序的主类？应用程序和小程序的主类有何不同？

一个程序可以有多个类，但只能有一个类是主类。在JAVA应用程序中，这个朱磊是指包含main()方法的类。

而在JAVA小程序中，这个朱磊是一个继承自系统类JApplet或Allpet的子类。

应用程序的主类不一定要求是public类，但小程序的朱磊要求必须是public类。

主类是JAVA程序执行的入口点。

*JAVA*应用程序与小程序之间有哪些差别？

简单来说应用程序是从主线程启动（也就是main()方法）。

Applet小程序是没有main方法，是嵌套在浏览器页面上运行（调用init()或者run()来启动），嵌入浏览器这点跟flash的小游戏类似。

*JAVA*和*C++*的区别？

- 都是面向对象的语言，都支持封装，继承，多态
- JAVA不提供指针来直接访问内存，程序内存更加安全
- JAVA的类是单继承，C++支持多重继承，索然JAVA的类不可以多继承，但是接口可以多继承。
- JAVA有自动内存管理机制，不需要程序员手动释放无用内存

Oracle JDK 和 *OpenJDK* 的对比

Oracle JDK版本将每三年发布一次，而OpenJDK版本每三个月发布一次；

OpenJDK 是一个参考模型并且是完全开源的，而Oracle JDK是OpenJDK的一个实现，并不是完全开源的；

Oracle JDK 比 OpenJDK 更稳定。OpenJDK和Oracle JDK的代码几乎相同，但Oracle JDK有更多的类和一些错误修复。因此，如果您想开发企业/商业软件，我建议您选择Oracle JDK，因为它经过了彻底的测试和稳定。某些情况下，有些人提到在使用OpenJDK 可能会遇到了许多应用程序崩溃的问题，但是，只需切换到Oracle JDK就可以解决问题；

在响应性和JVM性能方面，Oracle JDK与OpenJDK相比提供了更好的性能；

Oracle JDK不会为即将发布的版本提供长期支持，用户每次都必须通过更新到最新版本获得支持来获取最新版本；

Oracle JDK根据二进制代码许可协议获得许可，而OpenJDK根据GPL v2许可获得许可。

基础语法：

- JAVA有哪些数据类型：

定义：JAVA语言是强类型语言，对于每一种数据都定义了明确的具体的数据类型，在内存中分配了不同大小的内存空间。

分类：

基本数据类型：

整数类型（byte, short, init, long）

浮点类型（float, double）

数值类型

字符类型（char）

布尔类型（boolean）

引用数据类型：

类（class）

接口（interface）

数组（[]）

String

*switch*是否作用在*byte*上，是否作用在*long*上，是否作用在*String*上？

在JAVA5以前，switch(expr)中，expr只能是byte,short,char,int,从JAVA5开始，JAVA中引入了枚举类型，expr也可以是enum类型，

从JAVA7开始，expr还可以是字符串（String），但是长整型（long）在目前所有的版本中都是不可以的。

用最有效的方法计算2乘以8：

2<<3

（左移3位相当于乘以2的3次方，右移3位相当于除以2的3次方）

Math.round(11.5)等于多少？Math.round(-11.5)等于多少？

Math.round(11.5)的返回值是12，Math.round(-11.5)的返回值是-11

四舍五入的雨里是在参数上加入0.5然后，向下取整。

Float f = 3.4; 是否正确？

不正确。

3.4是双精度，将双精度型（double）赋值给浮点数（float）属于向下转型（down-casting,也称为窄化）会造成精度损失，因此需要强制类型转换 float f = (float)3.4;

或者写成 float f = 3.4Fl4;

*Short s1 = 1; s1 = s1 + 1;*有错吗？

对于short s1 = 1; s1 = s1 + 1;由于1是int类型，英雌s1+1运算也是int型，需要强制类型转换才能赋值给short型。

*Short s1 = 1; s1 += 1;*有错吗？

Short s1 = 1; s1 += 1; 可以正确编译，因此 s1 += 1;相当于s1 = (short(s1 + 1));

其中有隐含的强制类型转换。

*JAVA*语言采用何种编码方案？有何特点？

JAVA语言采用Unicode编码标准，Unicode（标准编码），它位每个字符制定了一个唯一的数值，因此，在任何的语言，平台，程序都可以放心的使用。

什么是*JAVA*注释？

定义：用于解释说明程序的文字

分类：

单行注释

格式：// 注释文字

多行注释

格式：/* 注释文字 */

文档注释

格式：/** 注释文字 */

作用：

在测试程序中，尤其是复杂的程序中，适当地加入注释可以增加程序地可读性，有利于程序地修改，调试，交流。注释地内容在程序编译地时候会被忽略，不会产生目标代码，注释地部分不会对程序地执行结果产生任何影响。

注意事项：

多行和文档注释都不能嵌套使用。

访问修饰符

访问修饰符: public, private, protected,以及不写（默认）时的区别

JAVA中，可以使用访问修饰符来对类，编程，方法，和构造方法地访问。

JAVA支持4中不同的访问权限。

分类：

Private:在同一类内可见。

使用对象：变量，方法。注意：不能修饰类（外部类）

Default(即缺省， 说明也不用写，不使用任何关键字)

在同一包内可见，不使用任何修饰符。

使用对象：类，接口，变量，方法。

Protected:对同一包内的类所以子类可见。

使用对象：变量，方法。注意：不能修饰类（外部类）

Public：对所有类可见。

使用对象：类，接口，变量，方法。

访问修饰符图：

修饰符	当前包	同包	子类	其他包
Public	✓	✓	✓	✓
Protected	✓	✓	✓	✗
Default	✓	✓	✗	✗
private	✓	✗	✗	✗

运算符：

&和&&的区别：

&运算符有两种用法：

- 按位与
- 逻辑与

&&运算符是短路与运算。

逻辑与跟短路与的差别是分层巨大的，索然二者都要求运算符作用两端的布尔都是true整个表达式的值才是true。

&&之所以称为短路运算。是因为如果&&左边的表达式的值是false，左边的表达式会被直接短路掉，不会再进行运算。

注意：逻辑或运算（|）和短路或运算符（||）的差别也是这样。

关键字：

JAVA有没有goto?

goto是java中的保留字，再目前版本的java中没有使用。

*flinal*有什么用?

用于修饰类，属性，和方法?

- 被final修饰的类不可以被继承
- 被flian修饰的方法不可以被重写
- 被final修饰的变量不可以被改变
- 被final修饰不可变的是变量的引用，而不是引用指向的内容，引用指向的内容是可以改变的。

*Final finally finalize*区别?

- final可以被修饰类，变量，方法，修饰类表示该类不能被继承，修饰的方法表示该方法不能被重写，修饰变量表示该变量是一个常量不能被重新赋值。

- finally一般作用再try-catch代码块中，再处理异常的时候，通常我们将一定要执行的代码方法写在finally代码块中，表示不管是否出现异常，该代码块都会执行，一般用来存放一些关闭资源的代码。

- finalize是一个方法，属于object类的一个方法，而object类是所有类的父类，该方法一般由垃圾回收来调用，当我们调用System.gc()方法的时候，由垃圾回收调用finalize(),回收垃圾，一个对象是否可以回收的最后判断。

*this*关键字的用法：

this是自身的一个对象，代表对象本身，可以理解位：指向对象本身的一个指针。

this的用法在java中大体可以分为3种：

- 普通的直接引用，this相当于是指向当前对象本身
- 形参与成员名字重名，用this来区分：

```
Public Person(String name, int age) {
    This.name = name;
    This.age = age;
}
```

- 引用本类的构造函数

```
Class Person {
    Private String name;
    Private int age;
    Public Person() {
    }
    Public Person(String name) {
        This.name = name
    }
    Public Person(String name, int age) {
        This(name);
        This.age = age;
    }
}
```

*super*关键字的用法：

Super可以理解为是指向自己超（父）类对象的一个指针，而这个超累指的是离自己最近的一个父类。

super也用三种用法：

1.普通的直接引用:

与this类似，super相当于是指向当前对象的父类的引用，这样就可以用super.xxx来引用父类的成员。

2.子类中的成员变量或方法与父类中的成员变量或方法同名时，用super来进行区分。

```
Class Person {
    Protected String name;
    Public Person(String name) {
        This.name = name;
    }
}
Class Student extends Person {
    Private String name;
    Public Student(String name, String name1) {
        Super(name);
        This.name = name1;
    }
    Public void getInfo() {
        System.out.println(this.name); //Child
        System.out.println(super.name); //Father
    }
}
    Public class Test {
    Public static void main(String[] args) {
        Student s1 new Student(“Father”, “Child”);
        S1.getInfo();
    }
}
```

引用父类构造函数

Super（参数）：调用父类中的某一个构造函数（应该为构造函数中的第一条语句）

This（参数）：调用本类中另一种形式的构造函数（应该为构造函数种的第一条语句）

*This*与*super*的区别：

Super:它引用当前对象的直接父类中的成员（直接访问父类中被隐藏的父类成员数据或函数，积累与派生类中有相同的成员定义时如：super.变量名 super.成员函数名（实参））

This:它代表当前对象（在程序中易产生二义性指出，应使用this来指明当前对象：如果函数的形参与类中的成员数据同名，这时需要this来指明成员变量名）

super()和this()类似,区别是，super()在子类中调用父类的构造方法，this()在本类内调用本类的其它构造方法

super()和this()均需放在构造方法内第一行

尽管可以用this调用一个构造器，但却不能调用两个。

this和super不能同时出现在一个构造函数里面，因为this必然会调用其它的构造函数，其它的构造函数必然也会有super语句的存在，所以在同一个构造函数里面有相同的语句，就失去了语句的意义，编译器也不会通过。

this()和super()都指的是对象，所以，均不可以在static环境中使用。包括：static变量,static方法，static语句块。

从本质上讲，this是一个指向本对象的指针, 然而super是一个Java关键字。

*static*存在的主要意义：

static的主要意义时在于创建独立于具用对象的域变量或者方法，以至于即使没有创建对象，也能使用属性和调用方法！

static关键字还有一个比较关键的作用就是，用来形成静态diamagnetic块以优化程序性能。

static块可以置于类中的任何地方，类中可以有多个static块。

在类初次被加载的时候，会按照static块的顺序来执行每个static块，并且只会执行一次。

为什么说*static*块可以用来优化程序的性能?

因为它的特性:只会在类加载的时候执行。因此，很多时候会将一些只需要进行一次的初始化操作都放在static代码块中进行。

static的独特之处：

1.被static修饰的变量或者方法时独立于该类的任何对象，也就是说，这些变量和方法不属于任何一个实列对象，二十倍类的实列对象所共享

怎么理解 “被类的实例对象所共享” 这句话呢?

就是说，一个类的静态成员，它是属于大伙的【大伙指的是这个类的多个对象实例，我们都知道一个类可以创建多个实例！】，所有的类对象共享的，不像成员变量是自个的【自个指的是这个类的单个实例对象】

2.在该类被第一次加载的时候，就会去加载被static修饰的部分，而且旨在类的第一次使用时加载并进行初始化，注意这是第一次用就要初始化，后面根据需要时可以再次赋值的。

3.static变量值在类加载的时候分配空间，以后创建类对象的时候，不会重新分配。

4.赋值的话，是可以任意赋值的!

5.被static修饰的变量或者方法优先于对象存在的，u而就是说当前一个类加载完毕忠厚，即便没有创建对象，也是可以去访问的。

*Static*应用场景：

因为static是被类的实列对象所共享，因此如果某个成员变量是被对象所共享的，那么这个成员变量就应该定义为静态变量。

比较常见的static应用场景：

1.修饰成员变量

2.修饰成员方法

3.静态代码块

4.修饰类（只能修饰内部类u而就是静态内部类）

5.静态导包

static注意事项：

1.静态只能访问静态

2.非静态即可以方位非静态，也可以访问静态的

流程控制语句：

Break, continue, return的区别及作用：

break跳出总上一层循环，不再执行循环（结束当前的循环体）

continue跳出本次循环，继续执行洗一次循环（结束正在执行的循环体，进入下一个循环条件）

return程序返回，不再执行下面的代码（结束当前的方法，直接返回）

在JAVA中，如何跳出当前多重嵌套循环

在JAVA中，要想跳出多重循环，可以在外面的循环语句前定义一个标号，然后在里层循环体的代码块使用待用标号的break语句，即可跳出外层村换。例如：

```
Public static void main(String[] args) {
    Ok;
    For (int i = 0; i < 10; i++) {
        For (int j = 0; j < 10; j++) {
            System.out.println(“i = ” + i + “j = ” + j);
                If (j == 5) {
                Break ok;
            }
        }
    }
}
```

面向对象：

面向对象和面向过程的区别：

面向过程：

优点：性能比面向对象高，因为类调用时需要实列化，开销比较大，比较消耗资源。

比如：单片机，嵌入式开发，linux/unix等一般采用面向过程开发，性能是最重要的因素。

缺点：没有面向对象易维护，易复用，易扩展

面向对象：

优点：易维护，易复用，易扩展，由于面向对象有封装，继承，多态性的特征，可以设计出低耦合的系统，使系统更加的灵活，更加易于维护。

缺点：性能比面向过程低

面向过程是具体化的。流程化，解决一个问题，你需要一步一步的分析，一步一步的实现。

面向对象是模型化的，只需要抽象出一个类，这是一个封闭的盒子，在这里你拥有数据也拥有解决问题的方法。

需要说明功能直接使用就可以，不必要去一步一步的是实现，至于这个功能是如何实现，不需要我们知道，和我们没有关系，我们会使用就可以了。

面向对象的底层其实还是面向过程，把面向过程抽象成类，然后封装，方便我们使用的就是面向对象。

面向对象的三大特征：

抽象：

抽象就是将一类对象的共同特征总结出来构造类的过程，包括数据抽象和行为抽象两方法。抽象只关注有哪些属性和行为，并不关注这些行为的细节。

封装：

封装就是把一个对象的属性私有化，同时提供一些可以被外界访问的属性的方法，如果属性不想被外界访问，我们大可不必提供方法给外界访问，但是如果一个类没有提供给外界访问的方法，那么这个类也没有什么实际的意义！

继承：

继承就是使用已存在的类定义作为基础建立新类的技术，新类的定义可以增加新的数据或新的功能，也可以用父类的功能，但不能选择性地继承父类。通过使用继承我们能够非常方便地复用以前的代码。

关于继承如下3点：（重点）

1.子类拥有父类非private的属性和方法。

2.子类可以拥有自己属性和方法，即子类可以对父类进行扩展

3.子类可以用自己的方法实现父类的方法

多态：

所谓多态就是指程序中定义的引用变量所指向的具体类型和通过该引用变量发出的方法调用在编程时并不确定，而是在册灰姑娘徐运行期间才确定，即一个引用变量到底会指向哪个类的实列对象，该引用发出的方法调用到底是哪个类中实现的方法，必须在由程序运行期间才能决定。

在JAVA中两种形式可以实现多态：继承（多个子类对同一个方法的重写）和接口（实现接口并覆盖接口中同一方法）。

其中JAVA中面向对象的三大特性：封装，继承，多态

封装：隐藏对象的属性和实现细节，仅对外提供公共访问方式，将变化隔离，便于，提高复用性和安全性。

继承：继承就是使用已存在的类的定义作为基础建立新类的技术。

新类的定义可以增加新的数据或新的功能，也可以用父类的功能，但不能选择性的继承父类。

通过使用继承可以提高代码的复用性。继承是多态的前提。

关于继承如下3点（重点）：

1.子类拥有父类分private的属性和方法。

2.子类可以拥有自己属性和方法，即子类可以对父类进行扩展

3.子类可以用自己的方式实现父类的方法

多态性：父类或接口定义的引用变量可以指向子类或具体实现类的实列对象。提高了程序的扩展性。

在JAVA中有两种形式可以实现多态：

继承（多个子类对同一个方法的重写）

接口（实现接口并覆盖接口中同一方法）

方法重载（overload）实现的是编译时的多态性（也成为前绑定）

方法重写（override）实现的是运行时的多态性（也成为后绑定）

一个引用变量到底会指向那个类的实列对象，该引用变量发出的方法调用到底时哪个类中实现的方法，必须在由程序运行期间才能决定。运行时的多态时面向对象最精髓的东西，要是实现多态需要做两件事：

1.方法重写（子类继承父类并重写父类中已有的或抽象的方法）

2.对象造型（用父类型引用子类型对象，这样同一的引用调用同一的方法就会分局子类对象的不同而表现出不同的行为）

什么时多态机制？*JAVA*语言如何实现多态的？

所谓多态就是指程序中定义的引用变量所指向的具体类型和通过该引用变量发出的方法调用在编程时并不确定，而是在程序运行期间才确定，即，一个引用变量到底会指向那个类的实列对象，该该引用变量发出的方法调用到底是哪个类中实现的方法，必须在由程序运行期间才能决定。因为在程序运行时才确定具体的类，这样，不用修改源程序代码，就可以让引用变量绑定到各种不同的类实现上，从而导致该引用调用的具体方法随之改变，即不修改程序代码就可以改变程序运行时所绑定的具体代码，让程序可以选择多个运行状态，这就是多态性。

多态：

多态分为编译时多态和运行时多态。

其中编译时多态时静态的，主要是指方法的重载，它是根据参数列表的不同来区分不同的函数，通过编译之后会变成两个不同的函数，在运行时谈不上多态。而运行时多态是动态的，它是通过动态来绑定实现的，也就是我们所说的多态性。

多态的实现

JAVA实现多态的三个必要条件：继承，重写，向上转型

1.继承：在多态中必须存在继承关系的子类和父类。

2.重写：子类对父类中某些方法进行重新定义，在调用这些方法时就会调用子类的方法。

3.向上转型：在多态中需要将子类的引用赋给父类对象，只有这样该应用才能够具备技能调用父类的方法和子类的方法。

只有满足了上述三个条件：才能在同一个机车呢个结构中使用统一的逻辑是西安代码处理不同的对象，从而达到执行不同的行为。

对于JAVA而言：它多态的实现机制遵循一个原则：

当超累对象引用变量引用子类对象时，被引用对象的类型而不是引用变量的类型决定了调用谁的成员方法，但是这个被调用的方法必须时在超类中定义过的，也就是说被子类覆盖的方法。

面向对象五大基本原则是什么？（可选）

1.单一职责原则SRP（Singele Responsibility Principle）

2.类的功能要单一，不能保罗万象，跟杂货铺似的。

3.开放封闭原则OCP（Open-close Principle）

4.一个模块对于扩展是开放的，对于修改是封闭的，想要增加功能热烈欢迎，想要修改功能，一万个不乐意。

5.里式替换原则LSP(the Liskov Substitution Principle LSP)

6.子类可以替换父类出现在父类能够出现的任何地方。比如你能代表你爸去你姥姥家干活。

7.依赖倒置原则DIP(the Dependency Inversion Principle DIP)

8.高层次的模块不应该依赖于低层次的模块，他们都应该依赖于抽象。抽象不应该依赖于具体实现，具体实现应该依赖于抽象。就是你出国要说你是中国人，而不能说你是哪个村子的。比如说中国人是抽象的，下面有具体的xx省，xx市，xx县。你要依赖的抽象是中国人，而不是你是xx村的

9.接口分离原则ISP(the Interface Segregation Principle ISP)设计时采用多个与特定客户类有关的接口比采用一个通用的接口要好。就比如一个手机拥有打电话，看视频，玩游戏等功能，把这几个功能拆分成不同的接口，比在一个接口里要好的多。

类于接口：

抽象类和接口的对比

1.抽象类就是用来捕捉子类的通用特性的

2.接口是抽象方法的集合

从设计层面来说，

抽象类是对类的抽象，是一种模板设计

接口是行为的抽象，是一种行为的规范

相同点：

接口和抽象类都不能实列化

都位于继承的顶端，用于被其他实现或继承

都包含抽象方法，其子类都必须覆写这些抽象方法

不同点：

参数	抽象类	接口
声明	抽象类使用 abstract 关键字声明	接口使用 interface 关键字声明
实现	子类使用 extends 关键字来继承抽象类。如果子类不是抽象类的话，它需要提供抽象类中所有的方法实现	子类使用 implements 关键字来实现接口。它需要提供接口中所以声明的方法的实现
构造器	抽象类可以有构造器	接口不能有构造器
访问修饰符	抽象类中的方法可以是任意访问修饰符	接口方法默认修饰符是 public。并且不允许定义为 private 或者 protected
多继承	一个类最多只能继承一个抽象类	一个类可以实现多个接口
字段声明	抽象类的字段声明可以是任意的	接口的字段默认是 static 和 final

备注：Java8中接口中引入默认方法和静态方法，以此来减少抽象类和接口之间的差异。

现在，我们可以为接口提供默认实现的方法了，并且不用强制子类来实现它

接口和抽象类各有优缺点，在接口和抽象类的选择上，必须遵守这样一个原则：

1.行为模型应该总是通过接口而不是抽象类定义，所以通常是优先选用接口，尽量少用抽象类。

2.选择抽象类的时候通常是如下情况：需要定义子类的行为，又要为子类提供通用的功能。

普通类和抽象类有哪些区别?

1.普通类不能包含抽象方法，抽象类可以包含抽象方法。

2.抽象类不能直接实例化，普通类可以直接实例化。

抽象类能使用 *final* 修饰吗?

不能，定义抽象类就是让其他类继承的，如果定义为 final 该类就不能被继承，这样彼此就会产生矛盾，所以 final 不能修饰抽象类

创建一个对象用什么关键字？对象实例与对象引用有何不同？

new关键字

new创建对象实例（对象实例在堆内存中），对象引用指向对象实例（对象引用存放在栈内存中）。一个对象引用可以指向0个或1个对象（一根绳子可以不系气球，也可以系一个气球）;一个对象可以有n个引用指向它（可以用n条绳子系住一个气球）

成员变量与局部变量的区别有哪些？

变量:

在程序执行的过程中，在某个范围内其值可以发生改变的量。从本质上讲，变量其实是内存中的一小块区域

成员变量:

方法外部，类内部定义的变量

局部变量:

类的方法中的变量。

成员变量和局部变量的区别

作用域

成员变量：针对整个类有效。局部变量：只在某个范围内有效。

(一般指的就是方法,语句体内)

存储位置:

成员变量：随着对象的创建而存在，随着对象的消失而消失，存储在堆内存中。

局部变量：在方法被调用，或者语句被执行的时候存在，存储在栈内存中。当方法调用完，或者语句结束后，就自动释放。

生命周期

成员变量：随着对象的创建而存在，随着对象的消失而消失

局部变量：当方法调用完，或者语句结束后，就自动释放。

初始值

成员变量：有默认初始值。

局部变量：没有默认初始值，使用前必须赋值。

使用原则

在使用变量时需要遵循的原则为：就近原则

首先在局部范围找，有就使用；接着在成员位置找。

在*Java*中定义一个不做事且没有参数的构造方法的作用

Java程序在执行子类的构造方法之前，如果没有用super()来调用父类特定的构造方法，则会调用父类中“没有参数的构造方法”。因此，如果父类中只定义了有参数的构造方法，而在子类的构造方法中又没有用super()来调用父类中特定的构造方法，则编译时将发生错误，因为Java程序在父类中找不到没有参数的构造方法可供执行。解决办法是在父类里加上一个不做事且没有参数的构造方法。

在调用子类构造方法之前会先调用父类没有参数的构造方法，其目的是?

帮助子类做初始化工作。

一个类的构造方法的作用是什么？若一个类没有声明构造方法，改程序能正确执行吗？为什么?

主要作用是完成对类对象的初始化工作。可以执行。因为一个类即使没有声明构造方法也会有默认的不带参数的构造方法。

构造方法有哪些特征：

名字与类相同

没有返回值，但是不能使用void声明构造函数

生成类的对象时自动执行，无序调用

静态变量和实列变量区别：

静态变量：

静态变量由于不属于任何实列对象，属于类，所有在内存中只会有一份，在类的加载过程中，JVM只为静态变量分配一次内存空间。

实列变量：

每次创建对象，都会为每个对象分配成员变量内存空间

实列变量是属于实列对象的，再内存中，创建几次对象，就会有几份成员变量。

静态变量于普通变量的区别：

static变量也成为静态变量，静态变量和非静态变量的区别就是：

静态变量被所有的对象共享，在内存中只有一个副本，它当且仅当再类初次加载时会被初始化。

而分静态变量时对象所拥有的，再创建对象的时候被初始化，存在多个副本，各个对象拥有的副本互不影响。

静态方法和实列方法有什么不同？

静态方法和实列方法的却别主要体现在两个方面：

在外部调用静态方法时，可以使用”类名.方法名”的方式，也可以使用”对象.方法名”的方式。

而实列方法只有后面这种方法。也就是说，调用静态方法可以无需创建对象。

静态方法在访问本类的成员时，只允许访问静态成员（即静态成员变量和静态方法）而不允许访问实列成员变量和实列方法，实列方法则无此限制

在一个静态方法内调用一个非静态成员为什么是非法的?

由于静态方法可以不通过对象进行调用，因此在静态方法里，不饿能调用其他非静态变量，也可以访问非静态变量成员。

什么是方法的返回值？返回值的作用是什么?

方法的返回值：

是指我们获取到的某个方法体中的代码执行后产生的结果！

（前提是该方法可能产生结果）

返回值的作用：

接受出结果，是的它可以用于其他的操作

什么是内部类?

在JAVA中，可以将一个类定义放在另一个类的定义内部，这就是内部类。

内部类本身就是类的一个属性，与其他属性定义方式一致

内部类的分类哪些?

内部类可以分为四种：

成员内部类

局部内部类

匿名内部类

静态内部类

静态内部类：

定义在类内部的静态类，就是静态内部类。

```
Public class Outer {
    Private static int radius = 1;
    Static class StaticInner {
        Public void visit() {
        System,out.println(“ static variable: ” + radius);
        }
    }
}
```

静态内部类可以访问外部类所有的静态变量，而不可以访问外部类的非静态变量；

静态内部类的创建方式，new 外部类.静态内部类()：

```
Outer.StaticInner inner = new Outer.StaticInner();
Inner.visit();
```

成员内部类：

定义在内部类，成员位置上的非静态类，就是成员内部类。

```
Public class Outer {
    Private static int radius = 1;
    Private int count = 2;
    Class Inner {
        Public void visit() {
            System.out.println(“static variable:” + radius);
            System.out.println(“visit outer variable:” + count);
        }
    }
}
```

成员内部类可以访问外部类所有的变量和方法，包括静态和非静态，私有和公有。

成员内部类依赖外部类的实列，它的创建方式 外部类实列.new 内部类().

```
Outer outer  = new Outer();
Outer.Inter inner = outer.new Inner();
Inner.visit();
```

局部内部类：

定义在方法的内部类，就是局部内部类。

```
Public class Outer {
Private int out_a = 1;
Private static int STATIC_b = 2;
Public void testFunctionClass() {
    Int inner_c = 3;
    Class Inner {
    Private void fun() {
```

```
        System.out.println(out_a);
                    System.out.println(STATIC_b);
                    System.out.println(inner_c);
        }
    }
    Inner  inner = new Inner();
    inner.fun();
    }
    public static void testStaticFunctionClass(){
        int d =3;
        class Inner {
            private void fun(){
// System.out.println(out_a); 编译错误，定义在静态方法中的局部类不可以访问外部类的实例变量
                System.out.println(STATIC_b);
                System.out.println(d);
            }
        }
        Inner  inner = new Inner();
        inner.fun();
    }
}
```

定义在实列的方法中的局部类可以访问外部类的所有变量和方法，定义在静态方法中的局部类只能访问外部类的静态变量和方法。

局部内部类的创建方式，对应方法内，new 内部类()

```
Public static void testStaticFunctionClass() {
   Class Inner {
     Inner inner = new Inner();
    }
}
```

匿名内部类：

匿名内部类就是没有名字的内部类，日常开发中使用比较多

```
Public class Outer {
    Private void test(final int i) {
        New Service() {
            Public void method() {
                For (int j = 0; j < i; j++) {
                    System.out.println(“匿名内部类”);
                }
            }
        }.method();
    }
}
Interface Service {
  Void method();
}
```

除了没有名字，匿名内部类还有以下特点：

匿名内部类必须继承一个抽象类或者实现一个接口。

匿名内部类不能定义任何静态成员和静态方法

当所在的方法的形参需要被匿名内部类使用时，必须声明为final。

匿名内部类不能是抽象的，它必须是实现继承的类或者实现接口的所有抽象方法

匿名内部类创建方式

New 类/接口{

//匿名内部类实现部分

}

内部类的优点：

我们为什么要使用内部类呢？

因为它有一个优点：

一个内部类对象可以访问创建它的外部类对象的内容，包括私有数据！

内部类不为同一包的其他类所见，具有很好的封装性

内部类有效实现了”多重继承”，优化java单继承的缺陷

匿名内部类可以很方便的定义回调

内部类有安歇用于场景

一些多算法场合

解决一些非面向对象的语句块

适当使用内部类，是的代码更加灵活和富有扩展性

当某个除了它的外部类，不再被其他的类使用时

局部内部类和匿名内部类局部变量的时候，为什么变量必须要加上final？

局部内部类和匿名内部类访问局部变量的时候，为什么变量必须要加上final呢？

它的内部原理是什么？

代码示例：

```
Public class Outer {
    Void outMethod() {
        Final int a = 10;
            Class Inner {
                Void innerMethod() {
                    System.out.println(a);
            }
        }
    }
}
```

以上案列：为什么要加final呢?

是因为生命周期不一致，局部变量直接存储在栈中，当方法执行结束后，非final的局部变量就被销毁。

而局部内部类堆局部变量的引用依然存在，如果局部内部类要调用局部变量时，就会出错，加了final，可以确保局部内部类使用的变量与外部的局部变量区分开，解决了这个问题。

内部类相关，看程序输出运行结果：

```
public class Outer {
  private int age = 12;

  class Inner {
    private int age = 13;
    public void print() {
      int age = 14;
      System.out.println("局部变量: " + age);
      System.out.println("内部类变量: " + this.age);
      System.out.println("外部类变量: " + Outer.this.age);
    }
  }

  public static void main(String[] args) {
    Outer.Inner in = new Outer().new Inner();
    in.print();
  }

}
```

运行结果：

局部变量：14
内部类变量：13
外部类变量：12

重写与重载

构造器（constructot）是否可以被重写（override）

构造器不能被继承，因此不能被重写，但可以被重载

重载（Overload）和重写（Override）的区别?

重载的方法是否根据返回值类型进行区分?

方法的重载和重写都是实现多态的方式，区别在于前者实现的是编译时的多态性

而后者实现的是运行时的多态性。

重载：发生在同一个类中，方法名相同参数列表不同（参数类型不同，个数不同，顺序不同）

与方法返回值和访问修饰无关，即重载的方法不能根据返回类型进行区分。

重写：发生在父子类中，方法名，参数列表必须相同，返回值小于等于父类，抛出的异常小于等于父类，访问修饰大于等于父类（里斯代换原则）

如果父类方法访问修饰符为private则，子类中就不是重写。

== 和*equals*的区别是什么?

==：

它的作用时判断两个对象的地址是不是相等，即：判断两个对象是不是同一个对象。（基本数据类型 == 比较的是值， 引用数据类型 == 比较的是内存地址）

Equals():

它的作用也是判断两个对象是否相等

但是它一般有两种使用情况

情况1：类没有覆盖equals()方法，则通过equals()比较该类的两个对象时，等价于通过 “==”比较两个对象。

情况2：类覆盖了equals()方法。

一般，我们都覆盖equals()方法来两个对象的内容相等，若它们的内容相等，则返回true（即，认为这两个对象相等）

举个案列：

```
public class test1 {
    public static void main(String[] args) {
        String a = new String("ab"); // a 为一个引用
        String b = new String("ab"); // b为另一个引用,对象的内容一样
        String aa = "ab"; // 放在常量池中
        String bb = "ab"; // 从常量池中查找
        if (aa == bb) // true
            System.out.println("aa==bb");
        if (a == b) // false，非同一对象
            System.out.println("a==b");
        if (a.equals(b)) // true
            System.out.println("aEQb");
        if (42 == 42.0) { // true
            System.out.println("true");
        }
    }
```

```
}
```

说明：

String中的equals方法是被重写的。因为object的equals方法是比较的对象的内存地址，而String的equals放啊比较的是对象的值。

当创建String类型的对象时，虚拟机会在常量池中查找有没有已经存在的值和要创建的值相同的对象，如果有就把它赋给当前引用。如果没有就在常量池中重新创建一个String对象。

HashCode 与*equals(*重要*)*

HashSet如何检查重复？

两个对象的hashcode（）形同，则equals（）也一定为true，对？

hashcode和equals方法的关系

面试官可能会问：

“你重写过hashcode和equals么”，为什么重写equals时必须重写hashcode方法？

Hashcode()介绍：

Hashcode()的作用时获取哈希码，也就是散列码：

它实际上就是返回一个int整数。

这个哈希码的作用是确定该对象在哈希表中的索引位置。

Hashcode()定义了JDK的Object.java中，这意味着java中的任何类都包含有hashcode()函数

散列表存储的是键值对（key-value）

它的特点是：

能根据”键”快速的检索出对应的”值”,这其中就利用了散列码！（可以快速找到所需要的对象）

为什么要有*hashcode*?

我们以”hashset如何检查重复”为列子为毛要有hashcode:

当你把对象加入hashset时，hashset会先计算对象的hashcode值来进行判断对象加入的位置，同时有人会与其他已经加入的对象的hashcode值进行比较，如果没有相符的hashcode，hashset会假设对象没有重复出现。但是如果没有发现相同的hashcode值的对象，这是就会调用equals()方法来检查hashcode相等的对象是否真的相同。

如果两者相同，hashset就不会让其加入操作成功，如果不同的话，就会重新散列到其他的位置。这样我们就大大的减少了equals的次数，相应的大大提高了执行的速度。

*Hashcode()*与*equals()*的相关规定：

如果两个对象相等，则hashcode一定也是相同的

两个对象相等，对两个对象分别调用equals方法都返回true

两个对象有相同的hashcode值，它们也不一定是相等的

因此,equals方法被覆盖过，则hashcode方法也必须被覆盖

Hashcode()的默认行为是对堆上的对象产生独特值。

如果没有重写hashcode(),则该class的两个对象无论如何都不会相等（即使两个对象指向相同的数据）

对象的相等与指向他们的引用相等，两者有何不同？

对象的相等比的是内存中存放的内容是否相等

引用相等比较的是它们指向的内存地址是否相等

值传递;

当一个对象被当作参数传递到一个方法后，此方法可以改变这个对象的属性，并可返回后的结果，那么这里到底是是传递花式引用传递？是值传递。

java语言的方法调用只支持参数的值传递。

当一个对象实列作为一个参数被传递到方法中时，参数的值就是对该对象的引用。

对象的属性可以被调用过程中被改变，但是对象引用的改变是不会影响到调用者的。

为什么 *Java* 中只有值传递?

首先回顾一下程序设计语言中有关将参数传递给方法（或函数）的一些专业术语。

按值调用(class by value)表示方法接收的是调用这提供的值，而按引用调用(call by reference)表示方法接收的是调用者提供的变量地址。

一个方法可以修改传递所对应的变量值，而布恩那个修改传递至调用所对应的变量值。

它用来描述各种程序设计（不只是java）中的方法参数传递方式。

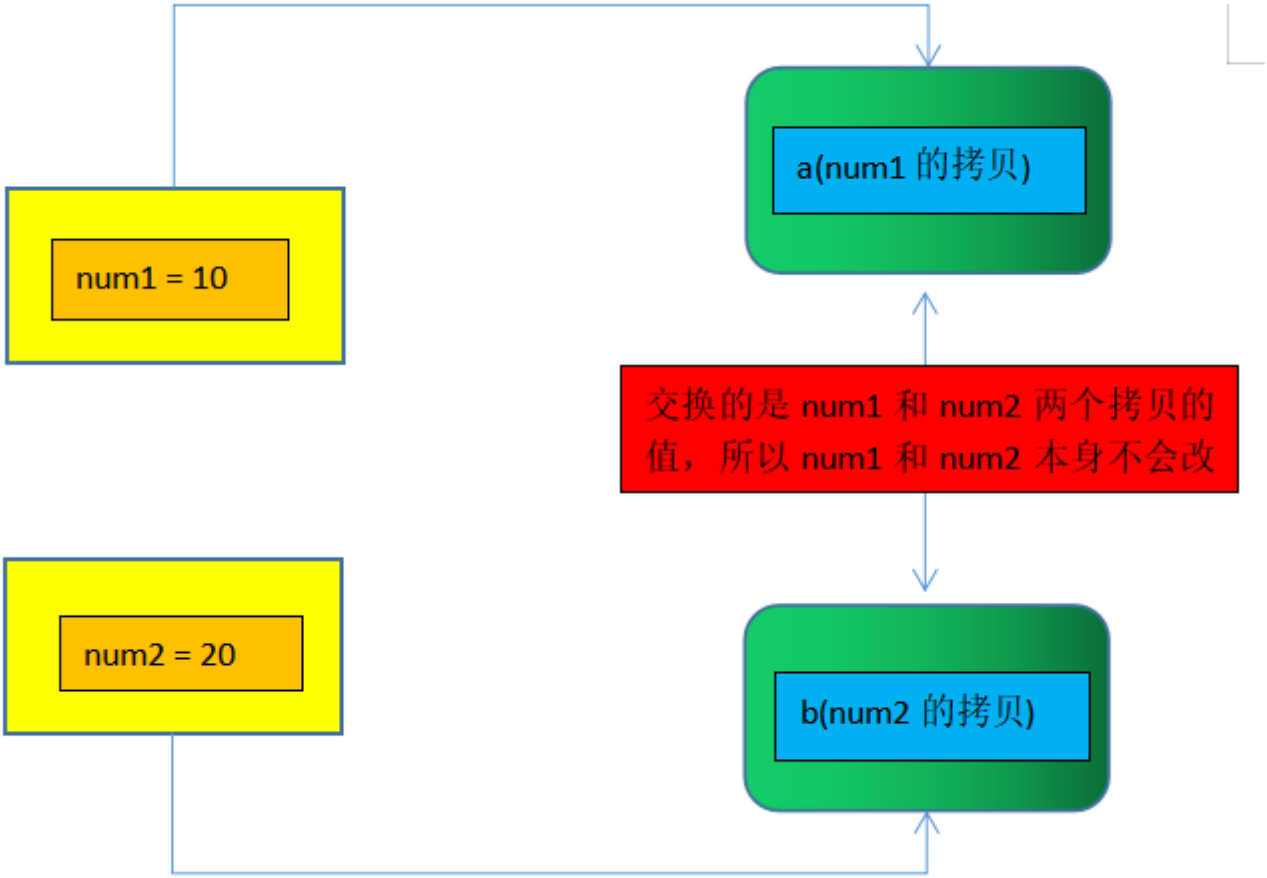
JAVA程序设计语言总是采用按值调用。

也就是说，方法得到的是所有参数值的一个拷贝，也就是说，方法不能修改传递给它的任何参数变量的内容。

下面通过 3 个例子来给大家说明

```
example 1

public static void main(String[] args) {
    int num1 = 10;
    int num2 = 20;

    swap(num1, num2);

    System.out.println("num1 = " + num1);
    System.out.println("num2 = " + num2);
}

public static void swap(int a, int b) {
    int temp = a;
    a = b;
    b = temp;

    System.out.println("a = " + a);
    System.out.println("b = " + b);
}
```

结果:
a = 20
b = 10
num1 = 10
num2 = 20



在swap方法中

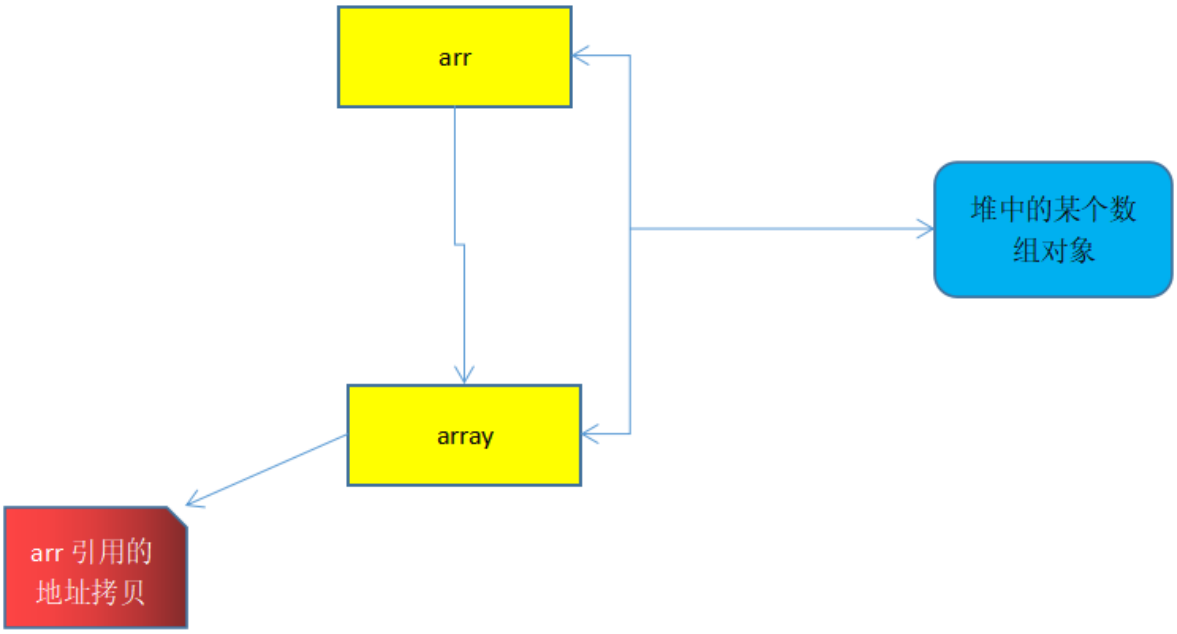
a、b的值进行交换，并不会影响到 num1、num2。因为，a、b中的值，只是从 num1、num2 的复制过来的。也就是说，a、b相当于num1、num2 的副本，副本的内容无论怎么修改，都不会影响到原件本身。

通过上面例子，我们已经知道了一个方法不能修改一个基本数据类型的参数，而对象引用作为参数就不一样，请看example2.

```
example 2

    public static void main(String[] args) {
        int[] arr = { 1, 2, 3, 4, 5 };
        System.out.println(arr[0]);
        change(arr);
        System.out.println(arr[0]);
    }

    public static void change(int[] array) {
        // 将数组的第一个元素变为0
        array[0] = 0;
    }
```

结果：

1
0
1
2



array 被初始化 arr 的拷贝也就是一个对象的引用，也就是说 array 和 arr 指向的时同一个数组对象。 因此，外部对引用对象的改变会反映到所对应的对象上。

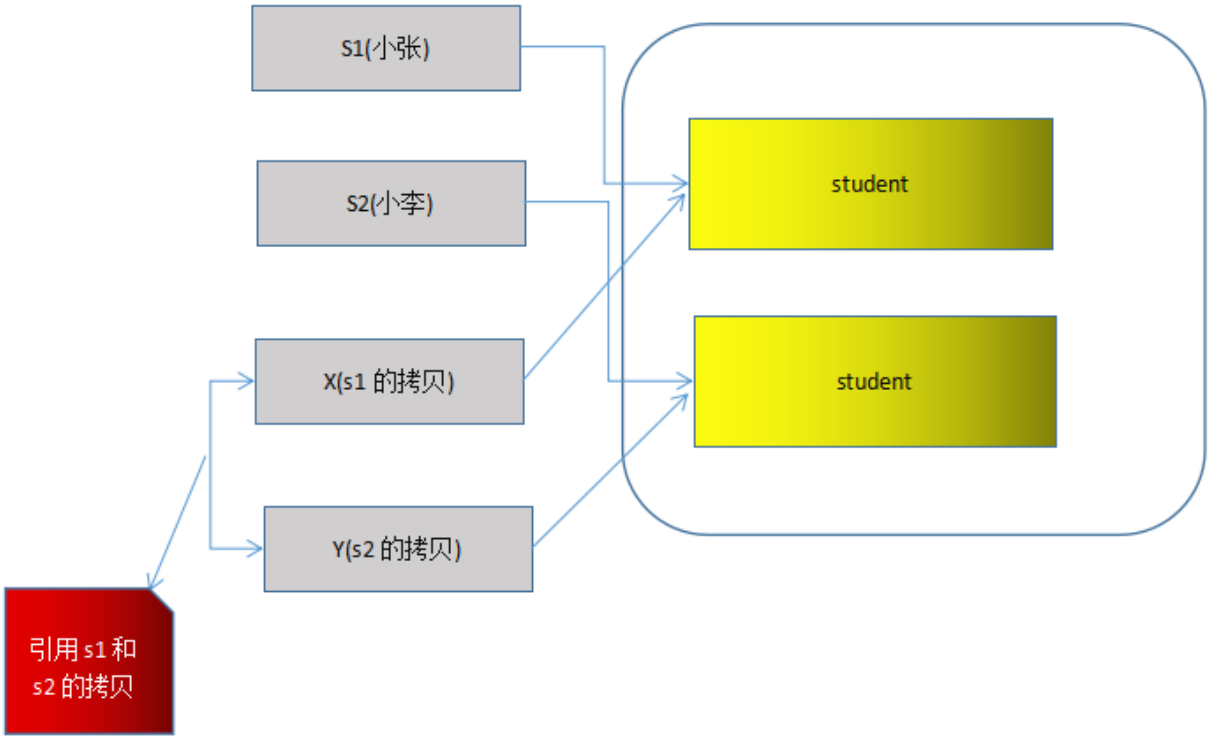
通过 example2 我们已经看到，实现一个改变对象参数状态的方法并不是一件难事。理由很简单，方法得到的是对象引用的拷贝，对象引用及其他的拷贝同时引用同一个对象。

很多程序设计语言（特别是，C++和Pascal)提供了两种参数传递的方式：值调用和引用调用。有些程序员（甚至本书的作者）认为Java程序设计语言对对象采用的是引用调用，实际上，这种理解是不对的。由于这种误解具有一定的普遍性，所以下面给出一个反例来详细地阐述一下这个问题。

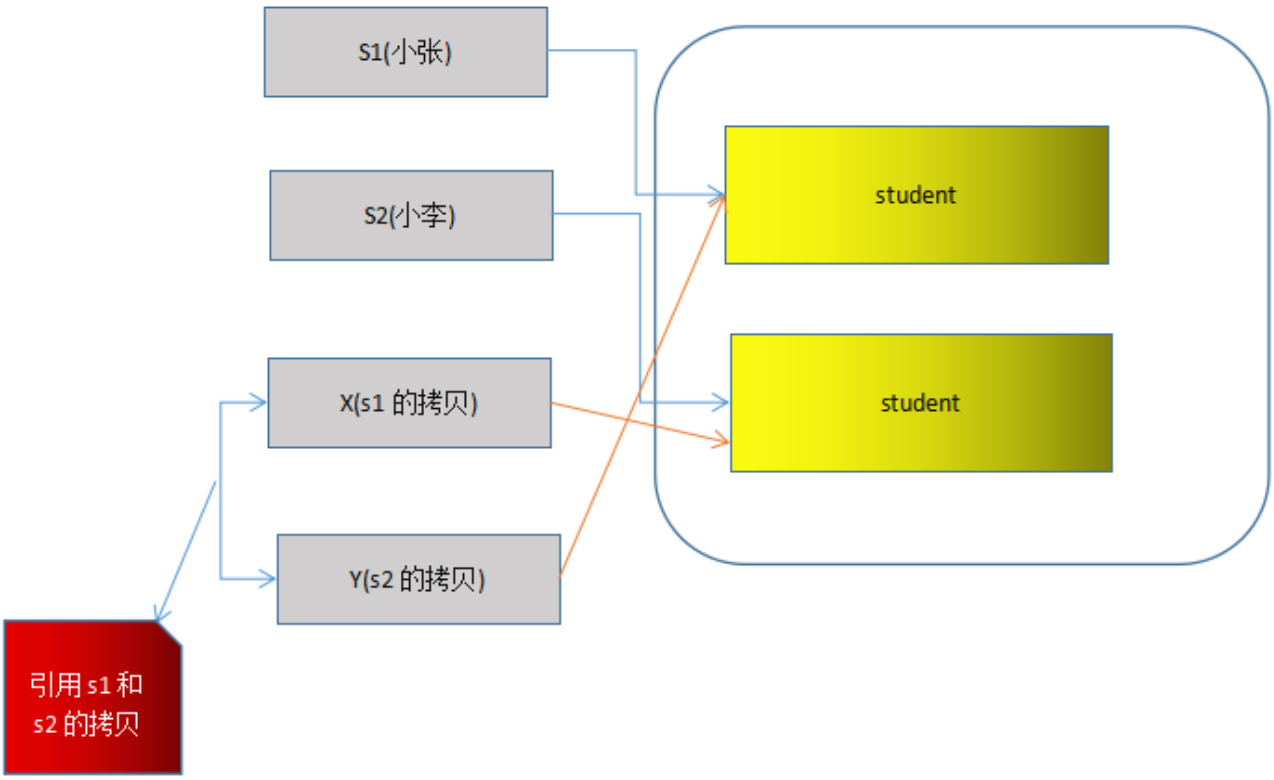
```
example 3

public class Test {

    public static void main(String[] args) {
        // TODO Auto-generated method stub
        Student s1 = new Student("小张");
        Student s2 = new Student("小李");
        Test.swap(s1, s2);
        System.out.println("s1:" + s1.getName());
        System.out.println("s2:" + s2.getName());
    }

    public static void swap(Student x, Student y) {
        Student temp = x;
        x = y;
```

```
        y = temp;
        System.out.println("x:" + x.getName());
        System.out.println("y:" + y.getName());
    }
}
```

解析：交换之前：



交换之后：



通过上面两张图可以很清晰的看出：方法并没有改变存储在变量 s1 和 s2 中的对象引用。swap方法的参数x和y被初始化为两个对象引用的拷贝，这个方法交换的是这两个拷贝

总结：

JAVA程序设计语言对对象采用的不是引用调用，实际上，对象引用是按值传递的。

总结一下JAVA中方法参数的使用情况：

一个方法不能修改一个基本数据类型的参数（即数值型或布尔型）

一个方法可以改变对象的参数的状态

一个方法不能让对象参数引用一个新的的对象

值传递和引用传递有什么区别?

值传递：

指的是在方法调用时，传递的参数是按值的拷贝传递，传递的是值的拷贝，也就是说传递过后就互不相关了。

引用传递：

指的是在方法调用时，传递的参数是按引用进行传递，其实传递的是引用的地址，也就是变量所对应的内存空间的地址。传递的是值的引用，也就是说传递和传递后都指向同一个引用（也就是同一个内存空间）。

JDK 中常用的包有哪些*?*

java.lang：这个是系统的基础类；

java.io：这里面是所有输入输出有关的类，比如文件操作等；

java.nio：为了完善 io 包中的功能，提高 io 包中性能而写的一个新包；

java.net：这里面是与网络有关的类；

java.util：这个是系统辅助类，特别是集合类；

java.sql：这个是数据库操作的类.

*import java*和*javax*有什么区别?

刚开始的时候 JavaAPI 所必需的包是 java 开头的包，javax 当时只是扩展 API 包来说使用。然而随着时间的推移，javax 逐渐的扩展成为 Java API 的组成部分。但是，将扩展从 javax 包移动到 java 包将是太麻烦了，最终会破坏一堆现有的代码。因此，最终决定 javax 包将成为标准API的一部分。

所以，实际上java和javax没有区别。这都是一个名字。

java 中 *IO* 流分为几种*?*

按照流的流向分，可以分为输入流和输出流；

按照操作单元划分，可以划分为字节流和字符流；

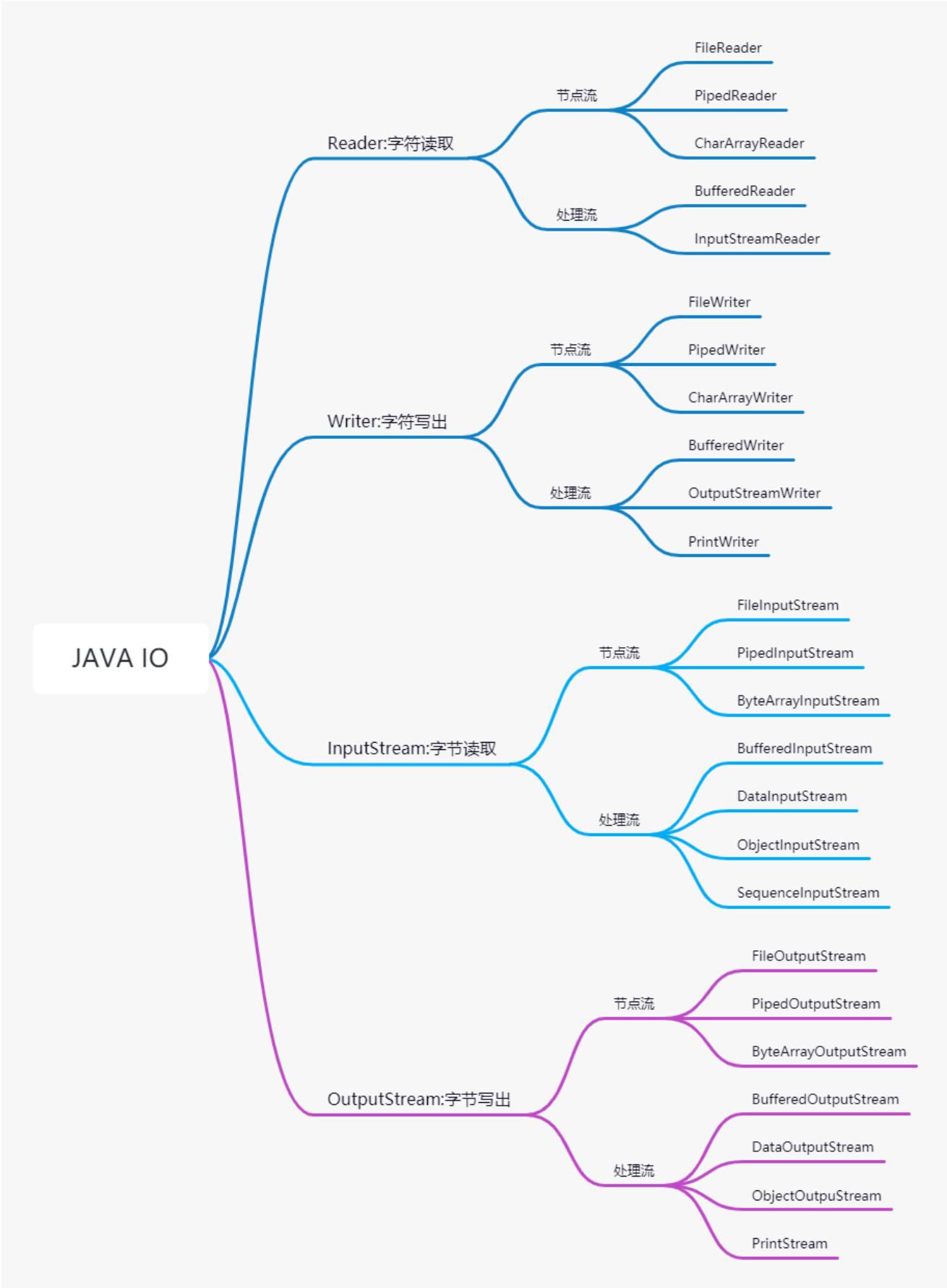
按照流的角色划分为节点流和处理流。

Java Io流共涉及40多个类，这些类看上去很杂乱，但实际上很有规则，而且彼此之间存在非常紧密的联系， Java I0流的40多个类都是从如下4个抽象类基类中派生出来的。

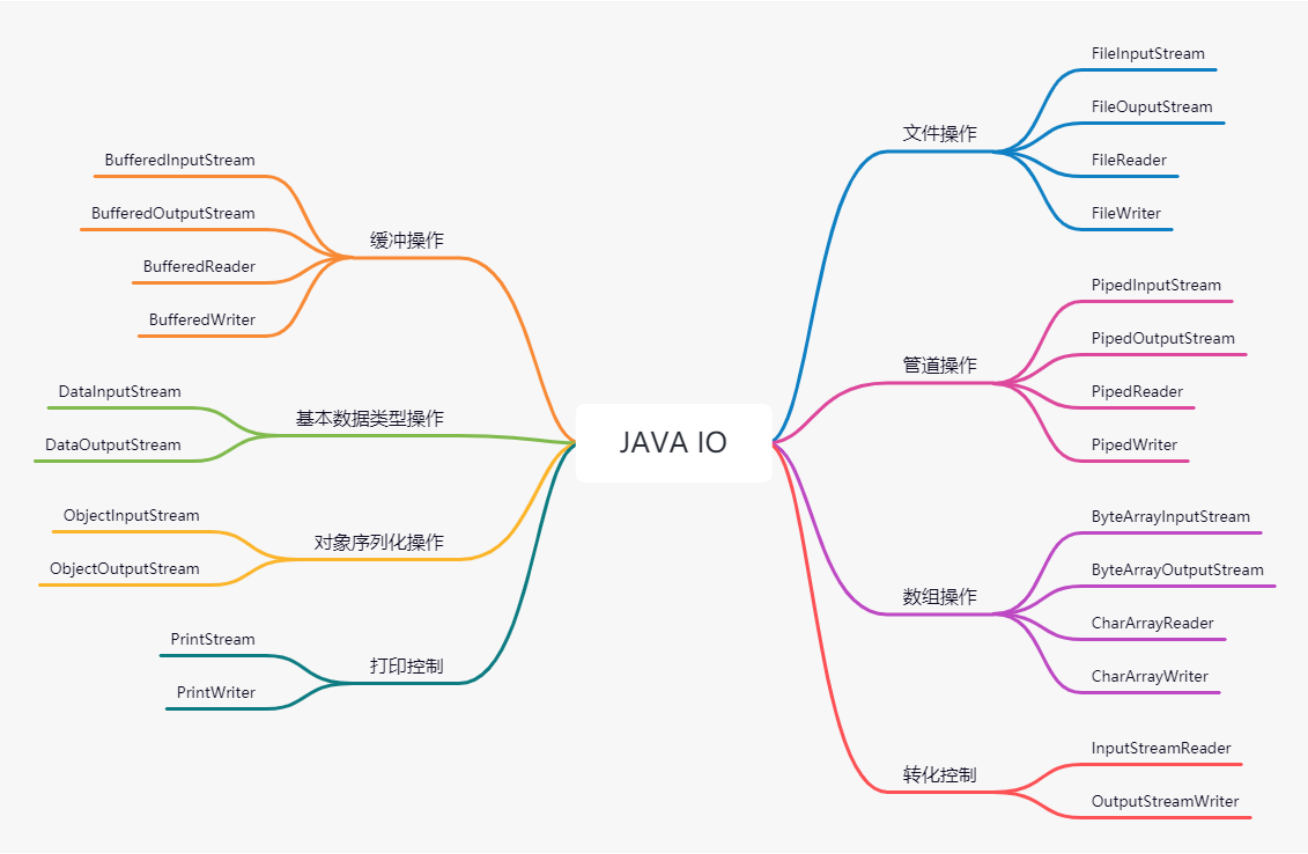
InputStream/Reader: 所有的输入流的基类，前者是字节输入流，后者是字符输入流。

OutputStream/Writer: 所有输出流的基类，前者是字节输出流，后者是字符输出流。

按操作方式分类结构图：


JAVA IO
Reader:字符读取
节点流
FileReader
PipedReader
CharArrayReader
处理流
BufferedReader
InputStreamReader
Writer:字符写出
节点流
FileWriter
PipedWriter
CharArrayWriter
处理流
BufferedWriter
OutputStreamWriter
PrintWriter
InputStream:字节读取
节点流
FileInputStream
PipedInputStream
ByteArrayInputStream
处理流
BufferedInputStream
DataInputStream
ObjectInputStream
SequenceInputStream
OutputStream:字节写出
节点流
FileOutputStream
PipedOutputStream
ByteArrayOutputStream
处理流
BufferedOutputStream
DataOutputStream
ObjectOutpuStream
PrintStream

按操作对象分类结构图：



BIO,NIO,AIO 有什么区别*?*

BIO：Block IO 同步阻塞式 IO，就是我们平常使用的传统 IO，它的特点是模式简单使用方便，并发处理能力低。

NIO：Non IO 同步非阻塞 IO，是传统 IO 的升级，客户端和服务器端通过 Channel（通道）通讯，实现了多路复用。

AIO：Asynchronous IO 是 NIO 的升级，也叫 NIO2，实现了异步非堵塞 IO ，异步 IO 的操作基于事件和回调机制。

详细回答

BIO (Blocking I/O): 同步阻塞I/O模式，数据的读取写入必须阻塞在一个线程内等待其完成。在活动连接数不是特别高（小于单机1000）的情况下，这种模型是比较不错的，可以让每一个连接专注于自己的 I/O 并且编程模型简单，也不用过多考虑系统的过载、限流等问题。线程池本身就是一个天然的漏斗，可以缓冲一些系统处理不了的连接或请求。但是，当面对十万甚至百万级连接的时候，传统的 BIO 模型是无能为力的。因此，我们需要一种更高效的 I/O 处理模型来应对更高的并发量。

NIO (New I/O): NIO是一种同步非阻塞的I/O模型，在Java 1.4 中引入了NIO框架，对应 java.nio 包，提供了 Channel , Selector，Buffer等抽象。NIO中的N可以理解为Non-blocking，不单纯是New。它支持面向缓冲的，基于通道的I/O操作方法。NIO提供了与传统BIO模型中的 Socket 和 ServerSocket 相对应的 SocketChannel 和 ServerSocketChannel 两种不同的套接字通道实现,两种通道都支持阻塞和非阻塞两种模式。阻塞模式使用就像传统中的支持一样，比较简单，但是性能和可靠性都不好；非阻塞模式正好与之相反。对于低负载、低并发的应用程序，可以使用同步阻塞I/O来提升开发速率和更好的维护性；对于高负载、高并发的（网络）应用，应使用 NIO 的非阻塞模式来开发

AIO (Asynchronous I/O): AIO 也就是 NIO 2。在 Java 7 中引入了 NIO 的改进版 NIO 2,它是异步非阻塞的IO模型。异步 IO 是基于事件和回调机制实现的，也就是应用操作之后会直接返回，不会堵塞在那里，当后台处理完成，操作系统会通知相应的线程进行后续的操作。AIO 是异步IO的缩写，虽然 NIO 在网络操作中，提供了非阻塞的方法，但是 NIO 的 IO 行为还是同步的。对于 NIO 来说，我们的业务线程是在 IO 操作准备好时，得到通知，接着就由这个线程自行进行 IO 操作，IO操作本身是同步的。查阅网上相关资料，我发现就目前来说 AIO 的应用还不是很广泛，Netty 之前也尝试使用过 AIO，不过又放弃了。

*Files*的常用方法都有哪些?

Files. exists()：检测文件路径是否存在。

Files. createFile()：创建文件。

Files. createDirectory()：创建文件夹。

Files. delete()：删除一个文件或目录。

Files. copy()：复制文件。

Files. move()：移动文件。

Files. size()：查看文件个数。

Files. read()：读取文件。

Files. write()：写入文件。

什么是反射机制?

JAVA反射机制是在运行状态中，对于任意一个类，都能够知道这个类的所有属性和方法；对于任意一个对象，都能够调用它的任意一个方法和属性；这种动态获取的信息以及动态调用对象的方法的功能称为java语言的反射机制。

静态编译和动态编译

静态编译：在编译时确定类型，绑定对象

动态编译：运行时确定类型，绑定对象

反射机制优缺点

优点：运行期类型的判断，动态加载类，提高代码灵活度。

缺点：性能瓶颈：反射相当于一系列解释操作，通知 JVM 要做的事情，性能比直接的java代码要慢很多。

反射机制的应用场景有哪些?

反射是框架设计的灵魂。

在我们平时的项目开发过程中，基本上很少会直接使用到反射机制，但这不能说明反射机制没有用，实际上有很多设计、开发都与反射机制有关，例如模块化的开发，通过反射去调用对应的字节码；动态代理设计模式也采用了反射机制，还有我们日常使用的 Spring / Hibernate 等框架也大量使用到了反射机制。

举例：①我们在使用JDBC连接数据库时使用Class.forName()通过反射加载数据库的驱动程序；②Spring框架也用到很多反射机制，最经典的就是xml的配置模式。Spring 通过 XML 配置模式装载 Bean 的过程：1) 将程序内所有 XML 或 Properties 配置文件加载入内存中; 2)Java类里面解析xml或properties里面的内容，得到对应实体类的字节码字符串以及相关的属性信息; 3)使用反射机制，根据这个字符串获得某个类的Class实例; 4)动态配置实例的属性

Java获取反射的三种方法

1.通过new对象实现反射机制

2.通过路径实现反射机制

3.通过类名实现反射机制

```
public class Student {
    private int id;
    String name;
    protected boolean sex;
    public float score;
}

public class Get {
    //获取反射机制三种方式
    public static void main(String[] args) throws ClassNotFoundException {
        //方式一(通过建立对象)
        Student stu = new Student();
        Class classobj1 = stu.getClass();
        System.out.println(classobj1.getName());
        //方式二（所在通过路径-相对路径）
        Class classobj2 = Class.forName("fanshe.Student");
        System.out.println(classobj2.getName());
        //方式三（通过类名）
        Class classobj3 = Student.class;
        System.out.println(classobj3.getName());
    }
}
```

网络编程

网络编程的面试题可以查看我的这篇文章重学TCP/IP协议和三次握手四次挥手，内容不仅包括TCP/IP协议和三次握手四次挥手的知识，还包括计算机网络体系结构，HTTP协议，get请求和post请求区别，session和cookie的区别等，欢迎大家阅读。

常用API

String相关：

字符型常量和字符串常量的区别？

形式上：

字符串常量是单引号引起的一个字符，字符串常量是双引号引起的若干个字符。

含义上：

字符串相当于一个整型值（ASCII值），可以参加表达式运算，字符串常量代表一个地址值（该地址值字符串在内存中存放的位置）

占内存大小：字符常量只占一个字节，字符串常量占若干个字节（至少一个字符结束标志）

什么是字符串常量池？

字符串常量池位于堆内存中，专门用来存储字符串常量，可以提高内存的使用率，避免开辟多块存储相同的字符串，在创建字符串时JVM会首先检查字符串常量池，如果该字符串已经存在池中，则返回它的引用，如果不存在，则实列化一个字符串放到池中，并返回其引用。

*String*是最基本的数据类型吗？

不是。

JAVA中的基本数据类型只有8个：

Byte, short, int, long, float, double, char, boolean

除了基本数据类型（primitive type）剩下的都是引用类型（reference type）

JAVA5以后引入的美剧类型也算是一种比较特俗的引用类型。

这是很基础的东西，但是很多初学者却容易忽视，Java 的 8 种基本数据类型中不包括 String，基本数据类型中用来描述文本数据的是 char，但是它只能表示单个字符，比如 ‘a’,‘好’ 之类的，如果要描述一段文本，就需要用多个 char 类型的变量，也就是一个 char 类型数组，比如“你好” 就是长度为2的数组 char[] chars = {‘你’,‘好’};

但是使用数组过于麻烦，所以就有了 String，String 底层就是一个 char 类型的数组，只是使用的时候开发者不需要直接操作底层数组，用更加简便的方式即可完成对字符串的使用。

*String*有哪些特性?

不变性。

String是只读字符串，是一个典型的immutable对象，堆它进行任何操作，其实都是创建一个新的对象，再把引用指向该对象。不变模式的主要作用在于当一个对象需要被多线程共享并平凡访问时，可以确保数据的一致性。

常量池的优化：

String对象创建之后，会在字符串常量池中进行缓存，如果下次创建同样的对象时，会直接返回缓存的引用。

Final：使用final来定义String类，表示String类不能被继承，提高了系统的安全性。

*String*为什么是不可变的吗?

简单来说就是String类利用了final修饰的char类型数组存储字符，源码如下图所以：

```
/** The value is used for character storage. */
private final char value[];
```

String真的是不可变的吗?

我觉得如果别人问这个问题的话，回答不可变就可以了。下面只是给大家看两个有代表性的例子：

String不可变但不代表引用不可以变

```
String str = "Hello";
str = str + " World";
System.out.println("str=" + str);
```

结果：

str=Hello World

解析：

实际上，原来String的内容是不变的，只是str由原来指向"Hello"的内存地址转为指向"Hello World"的内存地址而已，也就是说多开辟了一块内存区域给"Hello World"字符串。

通过反射是可以修改所谓的“不可变”对象

```
// 创建字符串"Hello World"， 并赋给引用s
String s = "Hello World";

System.out.println("s = " + s); // Hello World

// 获取String类中的value字段
Field valueFieldOfString = String.class.getDeclaredField("value");

// 改变value属性的访问权限
valueFieldOfString.setAccessible(true);

// 获取s对象上的value属性的值
char[] value = (char[]) valueFieldOfString.get(s);

// 改变value所引用的数组中的第5个字符
value[5] = '_';

System.out.println("s = " + s); // Hello_World
```

结果：

s = Hello World
s = Hello_World

解析：

用反射可以访问私有成员， 然后反射出String对象中的value属性， 进而改变通过获得的value引用改变数组的结构。但是一般我们不会这么做，这里只是简单提一下有这个东西。

是否可以继承 String 类?

String 类是 final 类，不可以被继承。

String str="i"与 String str=new String(“i”)一样吗?

不一样，因为内存的分配方式不一样。String str="i"的方式，java 虚拟机会将其分配到常量池中；而 String str=new String(“i”) 则会被分到堆内存中。

String s = new String(“xyz”);创建了几个字符串对象

两个对象，一个是静态区的"xyz"，一个是用new创建在堆上的对象。

```java
String str1 = "hello"; //str1指向静态区
String str2 = new String("hello"); //str2指向堆上的对象
String str3 = "hello";
String str4 = new String("hello");
System.out.println(str1.equals(str2)); //true
System.out.println(str2.equals(str4)); //true
System.out.println(str1 == str3); //true
System.out.println(str1 == str2); //false
System.out.println(str2 == str4); //false
System.out.println(str2 == "hello"); //false
str2 = str1;
System.out.println(str2 == "hello"); //true
```

如何将字符串反转?

使用 StringBuilder 或者 stringBuffer 的 reverse() 方法。

示例代码:

```java
// StringBuffer reverse
StringBuffer stringBuffer = new StringBuffer();
stringBuffer. append("abcdefg");
System. out. println(stringBuffer. reverse()); // gfedcba
// StringBuilder reverse
StringBuilder stringBuilder = new StringBuilder();
stringBuilder. append("abcdefg");
System. out. println(stringBuilder. reverse()); // gfedcba
```

数组有没有 *length()*方法? *String* 有没有 *length()*方法?

数组没有 length()方法 ，有 length 的属性。String 有 length()方法。JavaScript中，获得字符串的长度是通过 length 属性得到的，这一点容易和 Java 混淆。

String 类的常用方法都有那些?

```
indexOf()：返回指定字符的索引。
charAt()：返回指定索引处的字符。
replace()：字符串替换。
trim()：去除字符串两端空白。
split()：分割字符串，返回一个分割后的字符串数组。
getBytes()：返回字符串的 byte 类型数组。
length()：返回字符串长度。
toLowerCase()：将字符串转成小写字母。
toUpperCase()：将字符串转成大写字符。
substring()：截取字符串。
equals()：字符串比较。
```

在使用 *HashMap* 的时候，用 *String* 做 *key* 有什么好处?

HashMap 内部实现是通过 key 的 hashcode 来确定 value 的存储位置，因为字符串是不可变的，所以当创建字符串时，它的 hashcode 被缓存下来，不需要再次计算，所以相比于其他对象更快。

String和StringBuffer、StringBuilder的区别是什么？String为什么是不可变的 可变性

String类中使用字符数组保存字符串，private final char value[]，所以string对象是不可变的。StringBuilder与StringBuffer都继承自AbstractStringBuilder类，在AbstractStringBuilder中也是使用字符数组保存字符串，char[] value，这两种对象都是可变的。

线程安全性

String中的对象是不可变的，也就可以理解为常量，线程安全。AbstractStringBuilder是StringBuilder与StringBuffer的公共父类，定义了一些字符串的基本操作，如expandCapacity、append、insert、indexOf等公共方法。StringBuffer对方法加了同步锁或者对调用的方法加了同步锁，所以是线程安全的。StringBuilder并没有对方法进行加同步锁，所以是非线程安全的。

性能

每次对String 类型进行改变的时候，都会生成一个新的String对象，然后将指针指向新的String 对象。StringBuffer每次都会对StringBuffer对象本身进行操作，而不是生成新的对象并改变对象引用。相同情况下使用StirngBuilder 相比使用StringBuffer 仅能获得10%~15% 左右的性能提升，但却要冒多线程不安全的风险。

对于三者使用的总结

如果要操作少量的数据用 = String

单线程操作字符串缓冲区 下操作大量数据 = StringBuilder

多线程操作字符串缓冲区 下操作大量数据 = StringBuffer

Date相关

包装类相关

自动装箱与拆箱

装箱：将基本类型用它们对应的引用类型包装起来；

拆箱：将包装类型转换为基本数据类型；

int 和 *Integer* 有什么区别?

Java 是一个近乎纯洁的面向对象编程语言，但是为了编程的方便还是引入了基本数据类型，但是为了能够将这些基本数据类型当成对象操作，Java 为每一个基本数据类型都引入了对应的包装类型（wrapper class），int 的包装类就是 Integer，从 Java 5 开始引入了自动装箱/拆箱机制，使得二者可以相互转换。

Java 为每个原始类型提供了包装类型：

原始类型: boolean，char，byte，short，int，long，float，double

包装类型：Boolean，Character，Byte，Short，Integer，Long，Float，Double

Integer a= 127 与 Integer b = 127相等吗

对于对象引用类型：==比较的是对象的内存地址。

对于基本数据类型：==比较的是值。

如果整型字面量的值在-128到127之间，那么自动装箱时不会new新的Integer对象，而是直接引用常量池中的Integer对象，超过范围 a1==b1的结果是false

```
public static void main(String[] args) {
    Integer a = new Integer(3);
    Integer b = 3;  // 将3自动装箱成Integer类型
    int c = 3;
    System.out.println(a == b); // false 两个引用没有引用同一对象
    System.out.println(a == c); // true a自动拆箱成int类型再和c比较
    System.out.println(b == c); // true

    Integer a1 = 128;
    Integer b1 = 128;
    System.out.println(a1 == b1); // false

    Integer a2 = 127;
    Integer b2 = 127;
    System.out.println(a2 == b2); // true
}
```