面向对象和面向过程

- 面向过程 ：面向过程**性能**比面向对象**高。** 因为类调用时需要实例化，开销比较大，比较消耗资源，所以当性能是最重要的考量因素的时候，比如单片机、嵌入式开发、Linux/Unix等一般采用面向过程开发。但是，面向过程没有面向对象易维护、易复用、易扩展。
- 面向对象 ：面向对象**易维护、易复用、易扩展。** 因为面向对象有封装、继承、多态性的特性，所以可以设计出**低耦合**的系统，使系统更加灵活、更加易于维护。但是，面向对象性能比面向过程低。

Java 语言特点

1. 简单易学；
2. 面向对象（封装，继承，多态）；
3. 平台无关性（ Java 虚拟机实现平台无关性）；
4. 可靠性；
5. 安全性；
6. 支持多线程（ C++ 语言没有内置的多线程机制，因此必须调用操作系统的多线程功能来进行多线程程序设计，而 Java 语言却提供了多线程支持）；
7. 支持网络编程并且很方便（ Java 语言诞生本身就是为简化网络编程设计的，因此 Java 语言不仅支持网络编程而且很方便）；
8. 编译与解释并存；

Java和C++的区别

- 都是面向对象的语言，都支持封装、继承和多态
- Java 不提供指针来直接访问内存，程序内存更加安全
- Java 的类是单继承的，C++ 支持多重继承；虽然 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

应用程序和小程序

- 在 Java 应用程序中，主类是指包含 main（ ）方法的类。而在 Java 小程序中，主类是一个继承自系统类 JApplet 或 Applet 的子类。应用程序的主类不一定要求是 public 类，但小程序的主类要求必须是 public 类。主类是 Java 程序执行的入口点。
- 应用程序是从主线程启动(也就是 main() 方法)。applet 小程序没有 main() 方法，主要是嵌在浏览器页面上运行(调用init()或者run()来启动)

字符型常量和字符串常量

1. 形式上: 字符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
2. 含义上: 字符常量相当于一个整型值(ASCII 值),可以参加表达式运算; 字符串常量代表一个地址值(该字符串在内存中存放位置)
3. 占内存大小: 字符常量只占2个字节; 字符串常量占若干个字节(至少一个字符结束标志) (注意： char在Java中占两个字节)

三大特性:

封装

封装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意义了。

继承

继承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。
记住继承 3 点：

1. 子类拥有父类对象所有的属性和方法（包括私有属性和私有方法），但是父类中的私有属性和方法子类是无法访问，只是拥有。
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。（以后介绍）。

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，而是在程序运行期间才确定，即一个引用变量到底会指向哪个类的实例对象，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。
在Java中有两种形式可以实现多态：继承（多个子类对同一方法的重写）和接口（实现接口并覆盖接口中同一方法）。

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

可变性

简单的来说：String 类中使用 final 关键字修饰字符数组来保存字符串：private final char value[]，所以 String 对象是不可变的。而 StringBuilder 与 StringBuffer 都继承自AbstractStringBuilder类（StringBuilder与 StringBuffer 的构造方法都是调用父类构造方法），在 AbstractStringBuilder 中也是使用字符数组保存字符串 char[]value 但是没有用 final 关键字修饰，所以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，可以理解为常量，线程安全。AbstractStringBuilder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

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 StringBuilder 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。

总结：

1. 操作少量的数据: 适用String
2. 单线程操作字符串缓冲区下操作大量数据: 适用StringBuilder
3. 多线程操作字符串缓冲区下操作大量数据: 适用StringBuffer

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有参构造方法，则编译时将发生错误。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。

import java和javax有什么区别？

最初 JavaAPI 所必需的包是 java 开头的包，javax 只是扩展 API 包来使用。后来javax 逐渐地扩展成为 Java API 的组成部分。但是，将扩展从 javax 包移动到 java 包确实太麻烦了，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。

*接口和抽象类的区别是什么？

1. 接口的方法默认是 public，所有方法在接口中不能有实现(Java 8 开始接口方法可以有默认实现），而抽象类可以有非抽象的方法。
2. 接口中除了static、final变量，不能有其他变量，而抽象类中则不一定。
3. 一个类可以实现多个接口，但只能实现一个抽象类。接口自己本身可以通过extends关键字扩展多个接口。
4. 接口方法默认修饰符是public，抽象方法可以有public、protected和default这些修饰符（抽象方法就是为了被重写所以不能使用private关键字修饰！）。

5. 从设计层面来说，抽象是对类的抽象，是一种模板设计，而接口是对行为的抽象，是一种行为的规范。

备注：在JDK8中，接口也可以定义静态方法，可以直接用接口名调用。实现类和实现是不可以调用的。如果同时实现两个接口，接口中定义了一样的默认方法，则必须重写，不然会报错。

成员变量与局部变量的区别有那些？

1. 从语法形式上看:成员变量是属于类的，而局部变量是在方法中定义的变量或是方法的参数；成员变量可以被 public,private,static 等修饰符所修饰，而局部变量不能被访问控制修饰符及 static 所修饰；但成员变量和局部变量都能被 final 所修饰。

2. 从变量在内存中的存储方式来看:如果成员变量是使用static修饰的，那么这个成员变量是属于类的，如果没有使用static修饰，这个成员变量是属于实例的。而对象存在于堆内存，局部变量则存在于栈内存。

3. 从变量在内存中的生存时间上看:成员变量是对象的一部分，它随着对象的创建而存在，而局部变量随着方法的调用而自动消失。

4. 成员变量如果没有被赋初值:则会自动以类型的默认值而赋值（一种情况例外:被 final 修饰的成员变量也必须显式地赋值），而

局部变量则不会自动赋值。

创建一个对象用什么运算符?对象实体与对象引用有何不同?

new运算符，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）;一个对象可以有n个引用指向它。

构造方法有哪些特性？

1. 名字与类名相同。
2. 没有返回值，但不能用void声明构造函数。
3. 生成类的对象时自动执行，无需调用。

静态方法和实例方法有何不同

1. 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"对象名.方法名"的方式。而实例方法只有后面这种方式。也就是说，调用静态方法可以无需创建对象。
2. 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），而不允许访问实例成员变量和实例方法；实例方法则无此限制。

对象的相等与指向他们的引用相等,两者有什么不同?

对象的相等，比的是内存中存放的内容是否相等。而引用相等，比较的是他们指向的内存地址是否相等。

== 与 equals(重要)

== ：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() ：判断两个对象是否相等。但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“==”比较这两个对象。
- 情况2：类覆盖了 equals() 方法。一般，覆盖 equals() 方法来比较两个对象的内容是否相等；若它们的内容相等，则返回 true (即，认为这两个对象相等)。

覆盖String的equals：

```
public boolean equals(Object anObject)
{
     if (this == anObject) {
        return true;
     }
     if (anObject instanceof String) {
        String anotherString = (String)anObject;
        int n = value.length;
        if (n == anotherString.value.length) {
            char v1[] = value;
            char v2[] = anotherString.value;
            int i = 0;
            while (n-- != 0) {
                if (v1[i] != v2[i]) return false;
                i++;
            }
            return true;
        }
    }
    return false;
}
```

- **使用==操作符检查“参数是否为这个对象的引用”：如果是对象本身，则直接返回，拦截了对本身调用的情况，算是一种性能优化。**
- **使用instanceof操作符检查“参数是否是正确的类型”：如果不是，就返回false，正如对称性和传递性举例子中说得，不要想着兼容别的类型，很容易出错。在实践中检查的类型多半是equals所在类的类型，或者是该类实现的接口的类型，比如Set、List、Map这些集合接口。**
- **把参数转化为正确的类型： 经历了上一步的检测，基本会成功。**
- **对于该类中的“关键域”，检查参数中的域是否与对象中的对应域相等：基本类型的域就用==比较，float域用Float.compare方法，double域用Double.compare方法，至于别的引用域，我们一般递归调用它们的equals方法比较，加上判空检查和对自身引用的检查，一般会写成这样：**`(field == o.field || (field != null && field.equals(o.field)))`**,而上面的String里使用的是数组，所以只要把数组中的每一位拿出来比较就可以了。**
- **编写完成后思考是否满足上面提到的对称性，传递性，一致性等等。**

还有一些注意点。

覆盖equals时一定要覆盖hashCode

equals函数里面一定要是Object类型作为参数

hashCode 与 equals（重要）

hashCode（）介绍

hashCode() 的作用是获取哈希码（也称为散列码，实际上是返回一个int整数）确定该对象在哈希表中的索引位置。

hashCode() 定义在JDK的Object中，即Java中的任何类都包含有hashCode() 函数。但仅仅当创建“类的散列表”（散列表指的是：Java集合（数据结构）中本质是散列表的类，如HashMap，Hashtable，HashSet）时，该类的hashCode() 才有用；其它情况下(创建类的单个对象，或者创建类的对象数组等等)，是没有作用。

散列表本质是通过数组实现的，以键值对的形式存储。“键”对应的散列码计算得到数组索引再获取对应的“值”。特点：能根据“键”（使用散列码）快速的检索出对应的“值”。在散列表中：

1、如果两个对象相等，即通过equals()比较两对象返回true，那么它们的hashCode()值一定要相同；

2、如果两个对象hashCode()相等，即两个键值对的哈希值相等，但它们并不一定相等。这是哈希冲突，在这种情况下。若要判断两个对象是否相等，除了要覆盖equals()之外，也必须要覆盖hashCode()函数。否则，equals()无效。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（即使这两个对象指向相同的数据）

为什么要有 hashCode

以“HashSet 如何检查重复”为例子来说明为什么要有 hashCode： 当对象加入 HashSet 时，HashSet 会先计算对象的 hashcode 值来判断对象加入的位置，同时也会与其他已经加入的对象的 hashcode 值作比较，如果没有相符的hashcode，HashSet会假设对象没有重复出现。但是如果发现有相同 hashcode 值的对象，这时会调用 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位置。这样我们就大大减少了 equals 的次数，相应就大大提高了执行速度。

例如在String类中定义的hashcode()方法如下：

```
public int hashCode() {
    int h = hash;
    if (h == 0) {
        int off = offset;
```

```
        char val[] = value;
        int len = count;

        for (int i = 0; i < len; i++) {
            h = 31 * h + val[off++];
        }
        hash = h;
    }
    return h;
}
```

获取用键盘输入常用的的两种方法

方法1：通过 Scanner

```
Scanner input = new Scanner(System.in);
String s  = input.nextLine();
input.close();
```

方法2：通过 BufferedReader

```
BufferedReader input = new BufferedReader(new
InputStreamReader(System.in));
String s = input.readLine();
```

Java 中的异常处理

Java序列化中如果有些字段不想进行序列化，怎么办？

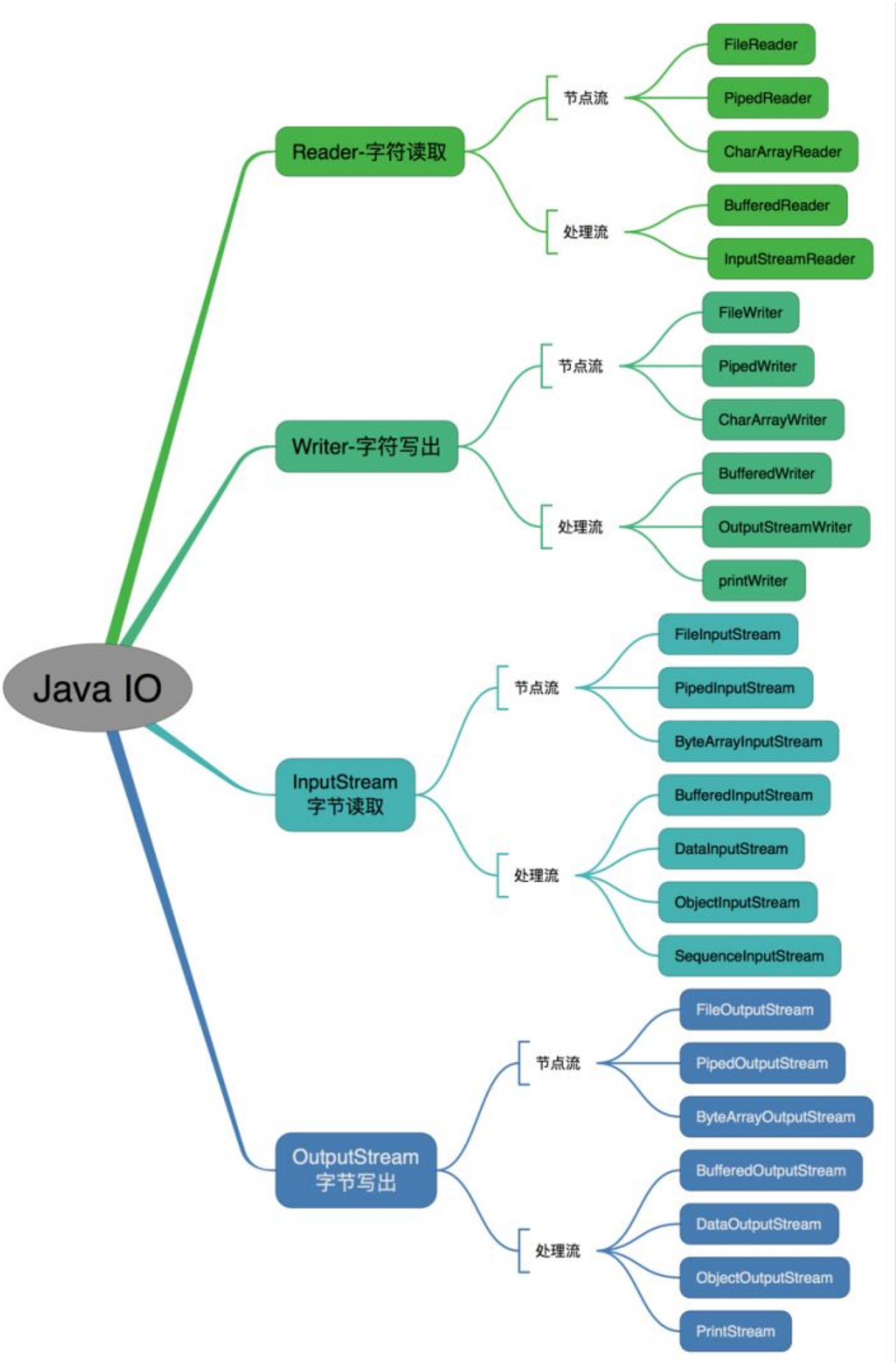
使用transient关键字修饰。

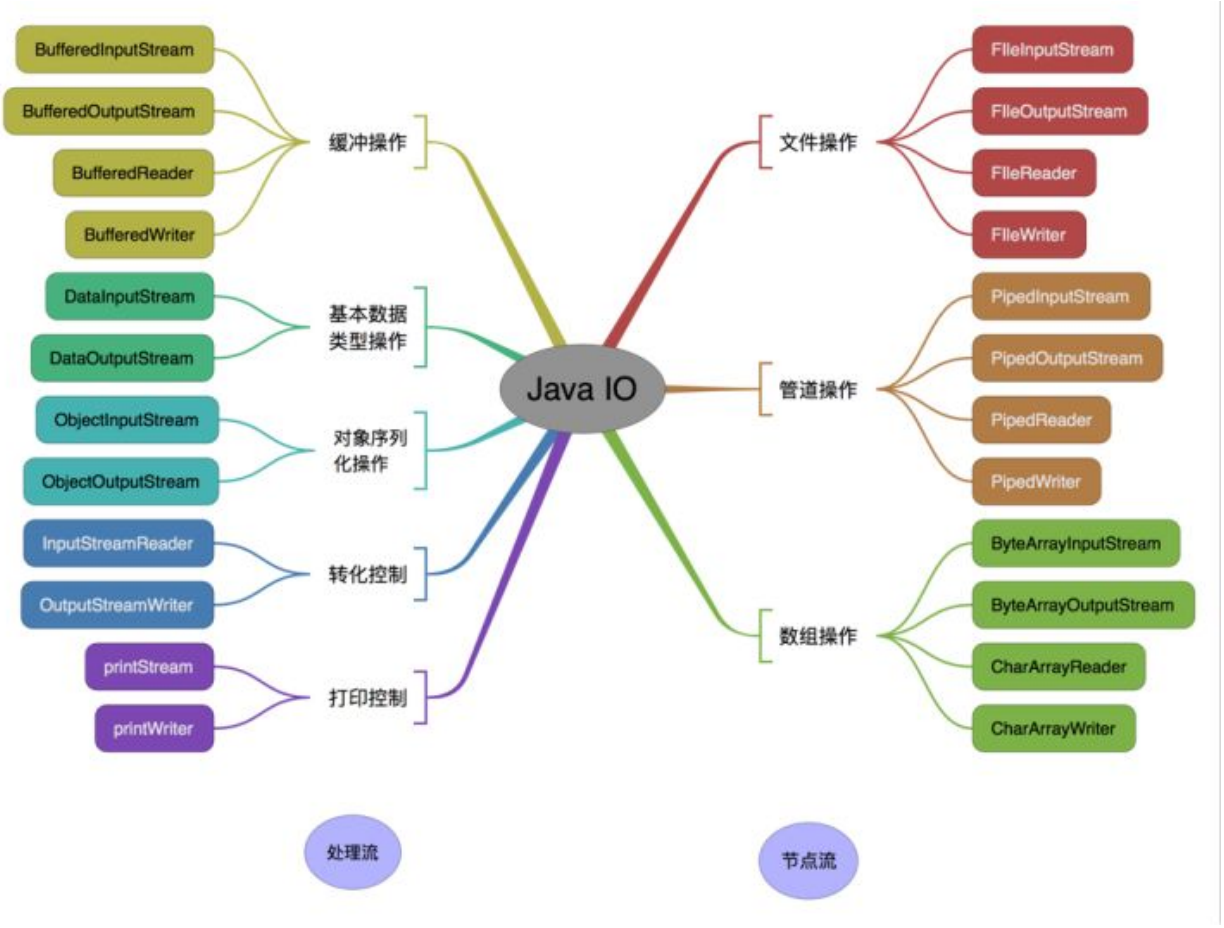
Java 中 IO 流分为几种?BIO,NIO,AIO 有什么区别?

- 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。**

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。


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
ObjectOutputStream
PrintStream



BIO,NIO,AIO 有什么区别?

- 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): 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。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，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。