码农求职小助手：Java基础

笔记本： 1-Java基础
创建时间： 2019/9/7 21:13 **更新时间：** 2019/9/7 21:13
作者： pc941206@163.com

更多资料请关注微信公众号：**码农求职小助手**

一、Java 语言基本特性

1、三大特性概述

1、**封装：** 通常认为封装是把数据和操作数据的方法封装起来，对数据的访问只能通过已定义的接口。

2、**继承：** 继承是从已有类得到继承信息创建新类的过程。提供继承信息的类被称为父类（超类/基类），得到继承信息的被称为子类（派生类）。

> 关于继承的几点补充：
>
> 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。**因为在一个子类被创建的时候，首先**

> **会在内存中创建一个父类对象，然后在父类对象外部放上子类独有的属性，两者合起来形成一个子类的对象；**
>
> 2、子类可以拥有自己属性和方法；
>
> 3、子类可以用自己的方式实现父类的方法。（重写）

3、**多态：** 分为编译时多态（方法重载）和运行时多态（方法重写）。要实现多态需要做两件事：一是子类继承父类并重写父类中的方法，二是用父类型引用子类型对象，这样同样的引用调用同样的方法就会根据子类对象的不同而表现出不同的行为。

2、重载（Overload）和重写（Override）的区别？【重点】

重载：编译时多态、同一个类中、同名的方法具有不同的参数列表、不能根据返回类型进行区分【因为：函数调用时不能指定类型信息，编译器不知道你要调哪个函数】；

重写（又名覆盖）：运行时多态、子类与父类之间、子类重写父类的方法具有相同的返回类型、更好的访问权限。

3、Java 中是否可以重写（Override）一个 private 或者 static 方法?

Java 中 static 方法不能被覆盖，因为方法覆盖是基于运行时动态绑定的，而 static 方法是编译时静态绑定的。static 方法跟类的任何实例都不相关，所以概念上不适用。

> **静态的方法可以被继承，但是不能重写。**如果父类和子类中存在同样名称和参数的静态方法，那么该**子类的方法会把原来继承过来的父类的方法隐藏，而不是重写。通俗的讲就是父类的方法和子类的方法是两个没有关系的方法，具体调用哪一个方法是看是哪个对象的引用；这种父子类方法也不在存在多态的性质** 【即 Father father = new child(); father 这个引用调用的是父类里的静态方法。如果是多态，那么 father 应该调用的子类的方法】

Java 中也不可以覆盖 private 的方法，因为 private 修饰的变量和方法只能在当前类中使用， 如果是其他的类继承当前类是不能访问到 private 变量或方法的，当然也不能覆盖。

4、Java 中创建对象的几种方式?

1、使用 new 关键字；

2、使用 Class 类的 newInstance 方法，该方法调用无参的构造器创建对象（反射）：Class.forName.newInstance()；

3、使用 clone() 方法；

4、反序列化，比如调用 ObjectInputStream 类的 readObject() 方法。

5、抽象类和接口有什么不同？【重点】

不同点：

1、抽象类中可以定义构造函数，接口不能定义构造函数；

2、抽象类中可以有抽象方法和具体方法，而接口中只能有抽象方法（public abstract）；

3、抽象类中的成员权限可以是 public、默认、protected（抽象类中抽象方法就是为了重写，所以不能被 private 修饰），而接口中的成员只可以是 public（方法默认：public abstrat、成员变量默认：public static final）；

> 在 Java8 中，允许在接口中包含带有具体实现的方法，使用 default 修饰，这类方法就是默认方法。

4、抽象类中可以包含静态方法，而接口中不可以包含静态方法；

> 抽象类中可以包含静态方法，在 JDK1.8 之前接口中不能包含静态方法，JDK1.8 以后可以包含。之前不能包含是因为，接口不可以实现方法，只可以定义方法，所以不能使用静态方法（因为静态方法必须实现）。现在可以包含了，只能直接用接口调用静态方法。1.8 仍然不可以包含静态代码块。
>
> 接口的成员变量默认是 public static final，static 是为了保证变量只有一份，因为一个类可以实现多个接口，定义为 static 后，如果出现重名，那么存储在静态存储区的时候就会报错，因为静态存储区已经有一份了，你改名吧（成员变量可以通过接口和实现接口的类来调用）。final 是因为接口的东西是大家共用的，不能随便修改。

相同点：

1、都不能被实例化；

2、可以将抽象类和接口作为引用类型；

3、一个类如何继承了某个抽象类或者实现了某个接口，就必须对其中所有的抽象方法全部进行实现，否则该类仍然需要被声明为抽象类。

5、抽象类

- **普通类和抽象类有哪些区别?**

  普通类：不能包含抽象方法，抽象类可以包含抽象方法。

  抽象类：不能直接实例化，普通类可以直接实例化。

- **抽象类必须要有抽象方法吗?**

不需要，抽象类不一定非要有抽象方法。

- **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。

7、静态变量和实例变量的区别?

静态变量： 是被 static 修饰的变量，也称为类变量，它属于类，因此不管创建多少个对象，静态变量在内存中有且仅有一个拷贝；静态变量可以实现让多个对象共享内存。

实例变量： 属于某一实例，需要先创建对象，然后通过对象才能访问到它。

8、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；但是，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。

9、构造器 Constructor 是否可被 override?

在讲继承的时候我们就知道父类的私有属性和构造方法并不能被继承，所以 Constructor 也就不能被 override（重写），但是可以 overload（重载）,所以你可以看到一个类中有多个构造函数的情况。

10、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、成类的对象时自动执行，无需调用。

11、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。

二、八种基本数据类型

Java 是一种强类型语言。变量的值占据一定的内存空间。不同类型的变量占据不同的大小。Java 中共有 8 种基本数据类型：包括 4 种整型、2 种浮点型、1 种字符型和 1 种布尔型。

1、byte：8位，最大存储数据量是255，存放的数据范围是-128~127之间。

2、short：16位，最大数据存储量是65536，数据范围是-32768~32767之间。

3、int：32位，最大数据存储容量是2的32次方减1，数据范围是负的2的31次方到正的2的31次方减1。

4、long：64位，最大数据存储容量是2的64次方减1，数据范围为负的2的63次方到正的2的63次方减1。

5、float：32位，数据范围在3.4e-45~1.4e38，直接赋值时必须在数字后加上 f 或 F。

6、double：64位，数据范围在4.9e-324~1.8e308，赋值时可以加 d 或 D 也可以不加。

7、boolean：只有 true 和 false 两个取值。

8、char：16位，存储 Unicode 码，用单引号赋值。

1、short s1 = 1；s1 = s1 + 1；有什么错？那么 short s1 = 1; s1 += 1；呢？有没有错误?

对于 short s1 = 1; s1 = s1 + 1; 来说，在 s1 + 1 运算时会自动提升表达式的类型为 int ，那么将 int 型值赋值给 short 型变量，**s1 会出现类型转换错误。**

对于 short s1 = 1; s1 += 1; 来说，+= 是 Java 语言规定的运算符，Java 编译器会对它进行特殊处理，因此可以正确编译。

2、Integer 和 int 的区别?

1、int 是 Java 的八种基本数据类型之一，而 Integer 是 Java 为 int 类型提供的封装类；

2、int 型变量的默认值是 0，Integer 变量的默认值是 null，这一点说明 Integer 可以区分出未赋值和值为 0 的区分；

3、Integer 变量必须实例化后才可以使用，而 int 不需要。

2.1 延申：关于 Integer 和 int 的比较：

1、由于 Integer 变量实际上是对一个 Integer 对象的引用，所以两个通过 new 生成的 Integer 变量永远是不相等的，因为其内存地址是不同的；

2、Integer 变量和 int 变量比较时，只要两个变量的值是相等的，则结果为 true。因为包装类 Integer 和基本数据类型 int 类型进行比较时，Java 会自动拆包装类为 int，然后进行比较，实际上就是两个 int 型变量在进行比较；

3、非 new 生成的 Integer 变量和 new Integer() 生成的变量进行比较时，结果为 false。因为非 new 生成的 Integer 变量指向的是 Java 常量池中的对象，而 new Integer() 生成的变量指向堆中新建的对象，两者在内存中的地址不同；

4、对于两个非 new 生成的 Integer 对象进行比较时，如果两个变量的值在区间 [-128, 127] 之间，则比较结果为 true，否则为 false。

这里解释下第 4 条的原因：Java 在编译 Integer i = 100 时，会编译成 Integer i = Integer.valueOf(100)，而 Integer 类型的 valueOf 的源码如下所示：

```
public static Integer valueOf(int var0) {
        return var0 >= -128 && var0 <= Integer.IntegerCache.high ?
Integer.IntegerCache.cache[var0 + 128] : new Integer(var0);
}
```

从上面的代码中可以看出：Java 对于 [-128, 127] 之间的数会进行缓存，比如：Integer i = 127，会将 127 进行缓存，下次再写 Integer j = 127 的时候，就会直接从缓存中取出，而对于这个区间之外的数就需要 new 了。

> https://www.cnblogs.com/javatech/p/3650460.html
> **包装类的缓存：**
>
> Boolean：全部缓存
>
> Byte：全部缓存
>
> Character：<= 127 缓存
>
> Short：-128 — 127 缓存
>
> Long：-128 — 127 缓存
>
> Integer：-128 — 127 缓存
>
> **Float：没有缓存**
>
> **Doulbe：没有缓存**

3、装箱和拆箱

自动装箱是 Java编译器在基本数据类型和对应得包装类之间做的一个转化。比如：把 int 转化成 Integer，double 转化成 Double 等等。反之就是自动拆箱。
原始类型：boolean、char、byte、short、int、long、float、double
封装类型：Boolean、Character、Byte、Short、Integer、Long、Float、Double

4、char 类型变量能不能存储一个中文的汉字，为什么?

char 类型变量是用来存储 Unicode 编码的字符的，Unicode 字符集包含了中文，所以 char 类型变量当然可以存储汉字。但是如果某个生僻字没有包含在 Unicode 编码的字符集中，那么 char 就不能存储该生僻字。

> 补充：常用中文字符用 utf-8 编码占用 3 个字节（大约 2 万多字），但超大字符集中的更大多数汉字要占 4 个字节。GBK、GB2312 收编的汉字占 2 个字节。

5、switch 语句能否作用在 byte 上，能否作用在 long 上，能否作用在 String 上?

在 switch(expr 1) 中，expr1 只能是一个整数表达式或者枚举常量。而整数表达式可以是 int 基本数据类型或者 Integer 包装类型。由于，**byte、short、char 都可以隐式转换为 int，所以，这些类型以及这些类型的包装类型也都是可以的。而 long 和 String 类型都不符合 switch 的语法规定，并且不能被隐式的转换为 int 类型，所以，它们不能作用于 switch 语句中。**不过，需要注意的是在 **JDK1.7 版本之后 switch 就可以作用在 String 上了。**

6、字节和字符的区别?

字节是存储容量的基本单位；

字符是数字、字母、汉字以及其他语言的各种符号；

1 字节 = 8 个二进制单位，一个字符由一个字节或多个字节的二进制单位组成。

7、Java 基本类型和引用类型的区别?

基本类型保存原始值，引用类型保存的是引用值（引用值就是指对象在堆中所处的位置）。

三、String

1、String 概述

String 被声明为 final，因此它不可被继承。

String 在 Java8 中，String 内部使用 char 数组存储数据。

```
public final class String implements java.io.Serializable,
Comparable<String>, CharSequence {
    /** The value is used for character storage. */
    private final char value[ ];
}
```

String 在 Java 9 之后，String 类的实现改用 byte 数组存储字符串，同时使用 coder 来标识使用了哪种编码。

```
public final class String implements java.io.Serializable,
Comparable<String>, CharSequence {
    /** The value is used for character storage. */
    private final byte[] value;

    /** The identifier of the encoding used to encode the bytes in {@code
value}. */
    private final byte coder;
}
```

value 数组被声明为 final，这意味着 value 数组初始化之后就不能再引用其它数组。并且 String 内部没有改变 value 数组的方法，因此可以保证 String 不可变。

而 StringBuilder 与 StringBuffer 都继承自 AbstractStringBuilder 类，在 AbstractStringBuilder 中的字符数组 char[] value 没有用 final 关键字修饰，所以这两种对象都是可变的。

```
abstract class AbstractStringBuilder implements Appendable, CharSequence {
        char[] value;
        int count;
        AbstractStringBuilder() {
        }
        AbstractStringBuilder(int capacity) {
                value = new char[capacity];
        }
```

2、String、StringBuilder、StringBuffer的区别【重点】？

11 1、String：用于字符串操作，属于不可变类；【补充：String 不是基本数据类型，是引用类型，底层用 char 数组实现的】

2、StringBuffer：也用于字符串操作，不同之处是 StringBuffer 属于可变类，对方法加了同步锁，线程安全；

说明：StringBuffer 中并不是所有方法都使用了 Synchronized 修饰来实现同步：

```
@Override
public StringBuffer insert(int dstOffset, CharSequence s) {
    // Note, synchronization achieved via invocations of other StringBuffer methods
    // after narrowing of s to specific type
    // Ditto for toStringCache clearing
    super.insert(dstOffset, s);
    return this;
}
```

上诉方法通过调用超类 AbstractStringBuilder 中的方法，将 s 转换为 String。

3、StringBuilder：与 StringBuffer 类似，都是字符串缓冲区，但线程不安全。

执行效率：StringBuilder > StringBuffer > String

String 字符串修改实现的原理：当用 String 类型来对字符串进行修改时，其实现方法是首先创建一个 StringBuffer，其次调用 StringBuffer 的 append() 方法，最后调用 StringBuffer 的 toString() 方法把结果返回。

3、String 与基本数据类型直接的转换?

字符串转换为基本数据类型：调用基本数据类型对应包装类中的方法 parseXXX(String) 或者 valueOf(String) 即可返回相应的基本数据类型；

基本数据类型转换为字符串：一种方法是将基本数据类型与空字符串（” ”）连接（+）即可获得其对应的字符串；另一种方法是调用 String 类中的 vauleOf() 方法返回相应字符串。

4、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。

详细讲解：https://blog.csdn.net/pcwl1206/article/details/81985828

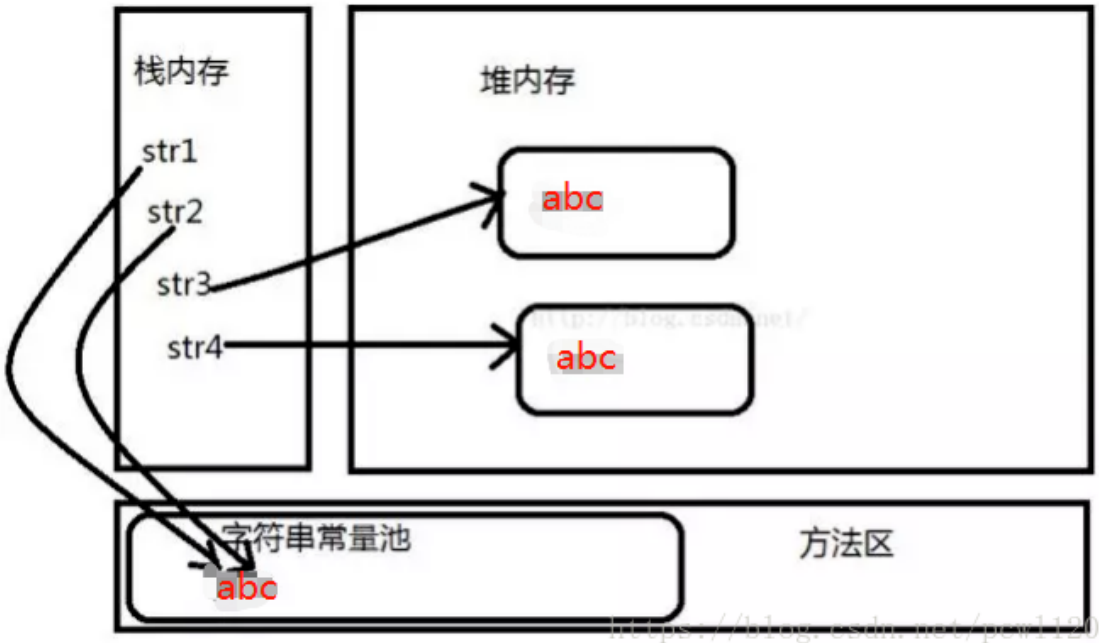
```
public static void main(String[] args){

        String str1 = "abc";
        String str2 = "abc";
        String str3 = new String("abc");
        String str4 = new String("abc");

        System.out.println(str1 == str2);          //true
        System.out.println(str1 == str3);          //false
        System.out.println(str3 == str4);          //false
        System.out.println(str3.equals(str4));     //true
}
```

在执行 String str1 = "abc" 的时候，JVM 会首先检查字符串常量池中是否已经存在该字符串对象，如果已经存在，那么就不会再创建了，直接返回该字符串在字符串常量池中的内存地址；如果该字符串还不存在字符串常量池中，那么就会在字符串常量池中创建该字符串对象，然后再返回。所以在执行 String str2 = "abc" 的时候，因为字符串常量池中已经存在“abc”字符串对象了，就不会在字符串常量池中再次创建了，所以栈内存中 str1 和 str2 的内存地址都是指向 "abc" 在字符串常量池中的位置，所以 str1 = str2 的运行结果为 true。

而在执行 String str3 = new String("abc") 的时候，JVM 会首先检查字符串常量池中是否已经存在“abc”字符串，如果已经存在，则不会在字符串常量池中再创建了；**如果不存在，则就会在字符串常量池中创建"abc"字符串对象，然后再到堆内存中再创建一份字符串对象，把字符串常量池中的"abc"字符串内容拷贝到内存中的字符串对象中，然后返回堆内存中该字符串的内存地址，即栈内存中存储的地址是堆内存中对象的内存地址。** String str4 = new String("abc")是在堆内存中又创建了一个对象，所以 str 3 == str4 运行的结果是 false。str1、str2、str3、str4 在内存中的存储状况如下图所示：



5、 如何将字符串反转?

使用 StringBuilder 或者 stringBuffer 的 reverse() 方法。

6、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。

7、final 修饰 StringBuffer 后还可以 append 吗?

可以。final 修饰的是一个引用变量，那么这个引用始终只能指向这个对象，但是这个对象内部的属性是可以变化的。

> once a final variable has been assigned, it always contains the same value. If a final variable holds a reference to an object, then the state of the object may be changed by operations on the object, but the variable will always refer to the same object.

四、关键字

1、final

1. 数据

声明数据为常量，可以是编译时常量，也可以是在运行时被初始化后不能被改变的常量。

对于基本类型： final 使数值不变；

对于引用类型： final 使引用不变，也就不能引用其它对象，但是被引用的对象本身是可以修改的。

```
final int x = 1;
// x = 2;    // cannot assign value to final variable 'x'final A y = new A();
y.a = 1;
```

2. 方法声明

方法不能被子类重写。

private 方法隐式地被指定为 final，如果在子类中定义的方法和基类中的一个 private 方法签名相同，此时子类的方法不是重写基类方法，而是在子类中定义了一个新的方法。

3. 类

声明类不允许被继承。

1.1、final、finally、finalize 的区别?

final： 用于声明属性、方法和类，分别表示属性不可变、方法不可覆盖、被其修饰的类不可继承；

finally： 异常处理语句结构的一部分，表示总是执行；

finallize： Object类的一个方法，在垃圾回收时会调用被回收对象的finalize。

2、finally

非常好的文章：https://blog.csdn.net/dove_knowledge/article/details/71077512

- **finally 块中的代码什么时候被执行?**

在 Java 语言的异常处理中，finally 块的作用就是为了保证无论出现什么情况，finally 块里的代码一定会被执行。由于程序执行 return 就意味着结束对当前函数的调用并跳出这个函数体，因此任何语句要执行都只能在 return 前执行（除非碰到 exit 函数），**因此 finally 块里的代码也是在 return 之前执行的。**

此外，如果 try-finally 或者 catch-finally 中都有 return，那么 finally 块中的 return 将会覆盖别处的 return 语句，最终返回到调用者那里的是 finally 中 return 的值。

程序在执行到 return 时会首先将返回值存储在一个指定的位置，其次去执行 finally 块，最后再返回。因此，对基本数据类型，在 finally 块中改变 return 的值没有任何影响，直接覆盖掉；而对引用类型是有影响的，返回的是在 finally 对 前面 return 语句返回对象的修改值。

- **finally 是不是一定会被执行到?**

不一定。下面列举两种执行不到的情况：

1、当程序进入 try 块之前就出现异常时，会直接结束，不会执行 finally 块中的代码；

2、当程序在 try 块中强制退出时也不会去执行 finally 块中的代码，比如在 try 块中执行 exit 方法。

- **try-catch-finally 中那个部分可以省略?**

catch 可以省略。try 只适合处理运行时异常，try+catch 适合处理运行时异常+普通异常。也就是说，如果你只用 try 去处理普通异常却不加以 catch 处理，编译是通不过的，因为编译器硬性规定，普通异常如果选择捕获，则必须用 catch 显示声明以便进一步处理。**而运行时异常在编译时没有如此规定，所以 catch 可以省略，你加上 catch 编译器也觉得无可厚非。**

3、static

3.1、静态变量

静态变量： 又称为类变量，也就是说这个变量属于类的，类所有的实例都共享静态变量，可以直接通过类名来访问它。静态变量在内存中只存在一份。

实例变量： 每创建一个实例就会产生一个实例变量，它与该实例同生共死。

3.2、静态方法

静态方法在类加载的时候就存在了，它不依赖于任何实例。所以**静态方法必须有实现，也就是说它不能是抽象方法。**只能访问所属类的静态字段和静态方法，方法中不能有 this 和 super 关键字。

3.3、静态语句块

静态语句块在类初始化时运行一次。

3.4、静态内部类

非静态内部类依赖于外部类的实例，而静态内部类不需要。静态内部类不能访问外部类的非静态的变量和方法。

3.5、初始化顺序

静态变量和静态语句块优先于实例变量和普通语句块，静态变量和静态语句块的初始化顺序取决于它们在代码中的顺序。

```
public static String staticField = "静态变量";

        static {
            System.out.println("静态语句块");   ---- 1
        }

        public String field = "实例变量";       ---- 2

        {
            System.out.println("普通语句块");  ----3
        }

        最后才是构造函数的初始化。public InitialOrderTest() {
            System.out.println("构造函数");    ----4
}
```

存在继承的情况下，初始化顺序为：

1. 父类（静态变量、静态语句块）
2. 子类（静态变量、静态语句块）
3. 父类（实例变量、普通语句块）
4. 父类（构造函数）
5. 子类（实例变量、普通语句块）
6. 子类（构造函数）

4、super

访问父类的构造函数： 可以使用 super() 函数访问父类的构造函数，从而委托父类完成一些初始化的工作。

访问父类的成员： 如果子类重写了父类的某个方法，可以通过使用 super 关键字来引用父类的方法实现。

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。

5、transient --> Java 序列化中如果有些字段不想进行序列化，怎么办?

对于不想进行序列化的变量，使用 transient 关键字修饰。

transient 关键字的作用是：阻止实例中那些用此关键字修饰的的变量序列化。当对象被反序列化时，被 transient 修饰的变量值不会被持久化和恢复。 transient 只能修饰变量，不能修饰类和方法。

五、运算

1、== 和 equals 的区别?

==： 如果比较的对象是基本数据类型，则比较的是数值是否相等；如果比较的是引用数据类型，则比较的是对象的地址值是否相等；

equals方法： 用来比较两个对象的内容是否相等；注意：**equals 方法不能用于比较基本数据类型的变量。**如果没有对 equals 方法进行重写，则比较的是引用类型的变量所指向的对象的地址。（很多类重新了 equals 方法，比如 String、Integer 等把它变成了值比较，所以一般情况下 equals 比较的是值是否相等）

2、两个对象的 hashCode() 相同，则 equals() 也一定为 true 吗?

两个对象的 hashCode() 相同，equals() 不一定为 true。因为在散列表中，hashCode() 相等即两个键值对的哈希值相等，然而哈希值相等，并不一定能得出键值对相等。【散列冲突】

3、为什么重写 equals() 就一定要重写 hashCode() 方法?

> 这个问题应该是有个前提，就是你需要用到 HashMap、HashSet 等 Java 集合。用不到哈希表的话，其实仅仅重写 equals() 方法也可以吧。而工作中的场景是常常用到 Java 集合，所以 Java 官方建议重写 equals() 就一定要重写 hashCode() 方法。

对于对象集合的判重，如果一个集合含有 10000 个对象实例，仅仅使用 equals() 方法的话，那么对于一个对象判重就需要比较 10000 次，随着集合规模的增大，时间开销是很大的。**但是同时使用哈希表的话，就能快速定位到对象的大概存储位置，并且在定位到大概存储位置后，后续比较过程中，如果两个对象的 hashCode 不相同，也不再需要调用 equals() 方法，从而大大减少了 equals() 比较次数。**

所以从程序实现原理上来讲的话，既需要 equals() 方法，也需要 hashCode() 方法。那么既然重写了 equals()，那么也要重写 hashCode() 方法，以保证两者之间的配合关系。

- **hashCode（）与equals（）的相关规定：**

18 1、如果两个对象相等，则 hashCode 一定也是相同的；

2、两个对象相等，对两个对象分别调用 equals 方法都返回 true；

3、两个对象有相同的 hashCode 值，它们也不一定是相等的；

4、因此，equals 方法被覆盖过，则 hashCode 方法也必须被覆盖；

5、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）

4、& 和 && 的区别?

Java中 && 和 & 都是表示与的逻辑运算符，都表示逻辑运输符 and，当两边的表达式都为 true 的时候，整个运算结果才为 true，否则为 false。

&&： 有短路功能，当第一个表达式的值为 false 的时候，则不再计算第二个表达式；

&： 不管第一个表达式结果是否为 true，第二个都会执行。除此之外，& 还可以用作位运算符：当 & 两边的表达式不是 Boolean 类型的时候，& 表示按位操作。

5、参数传递

https://blog.csdn.net/pcwl1206/article/details/86550268

Java 的参数是以值传递的形式传入方法中，而不是引用传递。

当传递方法参数类型为基本数据类型（数字以及布尔值）时，一个方法是不可能修改一个基本数据类型的参数。

当传递方法参数类型为引用数据类型时，一个方法将修改一个引用数据类型的参数所指向对象的值。

即使 Java 函数在传递引用数据类型时，也只是拷贝了引用的值罢了，之所以能修改引用数据是因为它们同时指向了一个对象，但这仍然是按值调用而不是引用调用。

6、Java 中的 Math.round(-1.5) 等于多少?

等于 -1，因为在数轴上取值时，中间值（0.5）**向右取整**，所以正 0.5 是往上取整，负 0.5 是直接舍弃。

7、两个数的异或结果是什么?

问题答案：两个二进制数异或结果 是 这两个二进制数差的绝对值，即表达为如下：

a^b = |a-b| （按位相减取绝对值，再按位累加）

解答过程：

二进制数a与b异或，即a和b两个数按位进行，如果对应位相同，即为0（这个时候相当于对应位算术相减），如果不相同，即为1（这个时候相当于对应位算术相减的绝对值）。由于二进制每个位只有两种状态，要么是0，要么是1，则按位异或操作可以表达为按位相减取绝对值，再按位累加。

两个二进制数异或的结果是多少?

六、异常

1、error 和 exception 的区别?

Error 类和 Exception 类的父类都是 Throwable 类。主要区别如下：

Error类： 一般是指与虚拟机相关的问题，如：系统崩溃，虚拟机错误，内存空间不足，方法调用栈溢出等。这类错误将会导致应用程序中断，仅靠程序本身无法恢复和预防；

Exception 类：分为运行时异常和受检查的异常。

运行时异常：【如空指针异常、指定的类找不到、数组越界、方法传递参数错误、数据类型转换错误】可以编译通过，但是一运行就停止了，程序不会自己处理；

受检查异常：要么用 try ... catch... 捕获，要么用 throws 声明抛出，交给父类处理。

2、throw 和 throws 的区别?

20 **throw：**

throw 在方法体内部，表示抛出异常，由方法体内部的语句处理；

throw 是具体向外抛出异常的动作，所以它抛出的是一个异常实例；

throws：

throws 在方法声明后面，表示如果抛出异常，由该方法的调用者来进行异常的处理；

表示出现异常的可能性，并不一定会发生这种异常。

3、常见的异常类有哪些?

NullPointerException：当应用程序试图访问空对象时，则抛出该异常。

SQLException：提供关于数据库访问错误或其他错误信息的异常。

IndexOutOfBoundsException：指示某排序索引（例如对数组、字符串或向量的排序）超出范围时抛出。

FileNotFoundException：当试图打开指定路径名表示的文件失败时，抛出此异常。

IOException：当发生某种 I/O 异常时，抛出此异常。此类是失败或中断的 I/O 操作生成的异常的通用类。

ClassCastException：当试图将对象强制转换为不是实例的子类时，抛出该异常。

IllegalArgumentException：抛出的异常表明向方法传递了一个不合法或不正确的参数。

4、主线程可以捕获到子线程的异常吗?

- 线程设计的理念：“**线程的问题应该线程自己本身来解决，而不要委托到外部**”。

正常情况下，如果不做特殊的处理，在主线程中是不能够捕获到子线程中的异常的。如果想要在主线程中捕获子线程的异常，我们可以用如下的方式进行处理，使用 Thread 的静态方法。

```
Thread.setDefaultUncaughtExceptionHandler(new MyUncaughtExceptionHandle());
```

六、泛型

泛型提供了编译期的类型安全，确保你只能把正确类型的对象放入集合中，避免了在运行时出现 ClassCastException。

1、Java 的泛型是如何工作的？什么是类型擦除？

泛型是通过类型擦除来实现的，编译器在编译时擦除了所有类型相关的信息，所以在运行时不存在任何类型相关的信息。例如：List<String> 在运行时仅用一个 List 来表示。这样做的目的，是确保能和 Java 5 之前的版本开发二进制类库进行兼容。

2、类型擦除

类型擦除：泛型信息只存在于代码编译阶段，在进入 JVM 之前，与泛型相关的信息会被擦除掉，专业术语叫做类型擦除。

在泛型类被类型擦除的时候，之前泛型类中的类型参数部分如果没有指定上限，如 <T> 则会被转译成普通的 Object 类型，如果指定了上限如 <T extends String> 则类型参数就被替换成类型上限。

```
List<String>  list  =  new  ArrayList<String>();
```

1、两个 String 其实只有第一个起作用，后面一个没什么卵用，只不过 JDK7 才开始支持 List<String>list = new ArrayList<>; 这种写法。

2、第一个 String 就是告诉编译器，List 中存储的是 String 对象，也就是起类型检查的作用，之后编译器会擦除泛型占位符，以保证兼容以前的代码。

3、什么是泛型中的限定通配符和非限定通配符？

限定通配符对类型进行了限制。有两种限定通配符，一种是<? extends T> 它通过确保类型必须是 T 的子类来设定类型的上界，另一种是<? super T>它通过确保类型必须是 T 的父类来设定类型的下界。泛型类型必须用限定内的类型来进行初始化，否则会导致编译错误。另一方面 <?> 表示了非限定通配符，因为 <?> 可以用任意类型来替代。

4、List<? extends T> 和 List <? super T> 之间有什么区别？

这两个 List 的声明都是限定通配符的例子，List<? extends T> 可以接受任何继承自 T 的类型的 List，而List <? super T> 可以接受任何 T 的父类构成的 List。例如 List<? extends Number> 可以接受 List<Integer> 或 List<Float>。

Array 不支持泛型，要用 List 代替 Array，因为 List 可以提供编译器的类型安全保证，而 Array却不能。

七、克隆和序列化

1、如何实现对象的克隆?

两种方式：

1、实现 Cloneable 接口并重写 Object 类中的 clone() 方法；

2、实现 Serializable 接口，通过对象的序列化和反序列化实现克隆，可以实现真正的深克隆。

注意：深克隆和浅克隆的区别：

1、**浅克隆**：拷贝对象和原始对象的引用类型引用同一个对象。浅克隆只是复制了对象的引用地址，两个对象指向同一个内存地址，所以修改其中任意的值，另一个值都会随之变化，这就是浅克隆（例：assign()）。

2、**深克隆**：拷贝对象和原始对象的引用类型引用不同对象。深拷贝是将对象及值复制过来，两个对象修改其中任意的值另一个值不会改变，这就是深拷贝（例：JSON.parse() 和 JSON.stringify()，但是此方法无法复制函数类型）。

> 深克隆的实现就是在引用类型所在的类实现 Cloneable 接口，并使用 public 访问修饰符重写 clone 方法；
>
> Java 中定义的 clone 没有深浅之分，都是统一的调用 Object 的 clone 方法。为什么会有深克隆的概念？是由于我们在实现的过程中刻意的嵌套了 clone 方法的调用。也就是说深克隆就是在需要克隆的对象类型的类中重新实现克隆方法 clone()。

2、什么是 Java 的序列化，如何实现 Java 的序列化?

对象序列化是一个用于将对象状态转换为字节流的过程，可以将其保存到磁盘文件中或通过网络发送到任何其他程序。从字节流创建对象的相反的过程称为反序列化。而创建的字节流是与平台无关的，在一个平台上序列化的对象可以在不同的平台上反序列化。序列化是为了解决在对象流进行读写操作时所引发的问题。

序列化的实现：将需要被序列化的类实现 Serializable 接口，该接口没有需要实现的方法，只是用于标注该对象是可被序列化的，然后使用一个输出流（如：FileOutputStream）来构造一个 ObjectOutputStream 对象，接着使用 ObjectOutputStream 对象的 writeObject(Object obj) 方法可以将参数为 obj 的对象写出，要恢复的话则使用输入流。

- 什么情况下需要序列化：

a）当你想把的内存中的对象状态保存到一个文件中或者数据库中时候；

b）当你想用套接字在网络上传送对象的时候；

c）当你想通过 RMI 传输对象的时候。

八、反射

每个类都有一个 Class 对象，包含了与类有关的信息。当编译一个新类时，会产生一个同名的 .class 文件，该文件内容保存着 Class 对象。

类加载相当于 Class 对象的加载，类在第一次使用时才动态加载到 JVM 中。也可以使用 Class.forName("com.mysql.jdbc.Driver") 这种方式来控制类的加载，该方法会返回一个 Class 对象。

反射可以提供运行时的类信息，并且这个类可以在运行时才加载进来，甚至在编译时期该类的 .class 不存在也可以加载进来。

Class 和 java.lang.reflect 一起对反射提供了支持，java.lang.reflect 类库主要包含了以下三个类：

1、Field ： 可以使用 get() 和 set() 方法读取和修改 Field 对象关联的字段；

2、Method ： 可以使用 invoke() 方法调用与 Method 对象关联的方法；

3、Constructor ： 可以用 Constructor 创建新的对象。

- 应用举例：

比如工厂模式：使用反射机制，根据全限定类名获得某个类的 Class 实例。

1、反射的优点：

- 面试回答：

优点：**运行期类型的判断，class.forName()动态加载类，提高代码的灵活度。**

> 静态加载类：编译时刻加载的类，例如：用 new 关键字创建的对象要通过编译器的静态检查，如果编译时该类不存在，那么使用该对象的类也无法通过编译。
>
> 动态加载类：**运行时刻加载的类，在编译的时候不会进行判断，只有在运行时才会进行判断，假设该类不存在，在运行时才会报错。**
>
> 反射使用的前提条件：必须先得到代表的字节码的 Class，Class 类用于表示 .class 文件（字节码）。在运行期间，一个类，只有一个 Class 对象产生。

- 官方回答：

可扩展性： 应用程序可以利用全限定名创建可扩展对象的实例，来使用来自外部的用户自定义类；

类浏览器和可视化开发环境： 一个类浏览器需要可以枚举类的成员。可视化开发环境（如 IDE）可以从利用反
射中可用的类型信息中受益，以帮助程序员编写正确的代码。

调试器和测试工具： 调试器需要能够检查一个类里的私有成员。测试工具可以利用反射来自动地调用类里定义的可被发现的 API 定义，以确保一组测试中有较高的代码覆盖率。

2、反射的缺点：

尽管反射非常强大，但也不能滥用。如果一个功能可以不用反射完成，那么最好就不用。在我们使用反射技术时，下面几条内容应该牢记于心。

性能开销： 反射涉及了动态类型的解析，所以 JVM 无法对这些代码进行优化。因此，反射操作的效率要比那些非反射操作低得多。我们应该避免在经常被执行的代码或对性能要求很高的程序中使用反射。

安全限制： 使用反射技术要求程序必须在一个没有安全限制的环境中运行。如果一个程序必须在有安全限制的环境中运行，如 Applet，那么这就是个问题了。

内部暴露： 由于反射允许代码执行一些在正常情况下不被允许的操作（比如：访问私有的属性和方法），所以使用反射可能会导致意料之外的副作用，这可能导致代码功能失调并破坏可移植性。反射代码破坏了抽象性，因此当平台发生改变的时候，代码的行为就有可能也随着变化。

九、动态代理

1、动态代理是什么？有哪些应用？

- **动态代理：**

当想要给实现了某个接口的类中的方法，加一些额外的处理。比如说加日志，加事务等。可以给这个类创建一个代理，故名思议就是创建一个新的类，这个类不仅包含原来类方法的功能，而且还在原来的基础上添加了额外处理的新功能。**这个代理类并不是定义好的，是动态生成的。**具有解耦意义，灵活，扩展性强。

- **动态代理的应用：**

  Spring 的 AOP 、加事务、加权限、加日志。

2、怎么实现动态代理?

首先必须定义一个接口，还要有一个 **InvocationHandler**（将实现接口的类的对象传递给它）处理类。再有一个**工具类 Proxy**（习惯性将其称为代理类，因为调用它的 newInstance() 可以产生代理对象，其实它只是一个产生代理对象的工具类）。利用到 InvocationHandler，拼接代理类源码，将其编译生成代理类的二进制码，利用加载器加载，并将其实例化产生代理对象，最后返回。

每一个动态代理类都必须要实现 InvocationHandler 这个接口，并且每个代理类的实例都关联到了一个 handler，当我们通过代理对象调用一个方法的时候，这个方法的调用就会被转发为由 InvocationHandler 这个接口的 invoke 方法来进行调用。我们来看看 InvocationHandler 这个接口的唯一一个方法 invoke 方法：

```
Object invoke(Object proxy, Method method, Object[] args) throws Throwable
```

> proxy: 指代我们所代理的那个真实对象
>
> method: 指代的是我们所要调用真实对象的某个方法的 Method 对象
>
> args: 指代的是调用真实对象某个方法时接受的参数

Proxy 类的作用是动态创建一个代理对象的类。它提供了许多的方法，但是我们用的最多的就是 newProxyInstance 这个方法：

```
public static Object newProxyInstance(ClassLoader loader, Class<?>[]
interfaces,  InvocationHandler handler)  throws IllegalArgumentException
```

> loader： 一个 ClassLoader 对象，定义了由哪个 ClassLoader 对象来对生成的代理对象进行加载；
>
> interfaces：一个 Interface 对象的数组，表示的是我将要给我需要代理的对象提供一组什么接口，如果我提供了一组接口给它，那么这个代理对象就宣称实现了该接口(多态)，这样我就能调用这组接口中的方法了

> handler：一个 InvocationHandler 对象，表示的是当我这个动态代理对象在调用方法的时候，会关联到哪一个 InvocationHandler 对象上。

通过 Proxy.newProxyInstance 创建的代理对象是在 Jvm 运行时动态生成的一个对象，它并不是我们的 InvocationHandler 类型，也不是我们定义的那组接口的类型，而是在运行是动态生成的一个对象。

十、IO 流

推荐阅读：极客时间：https://time.geekbang.org/column/article/99478

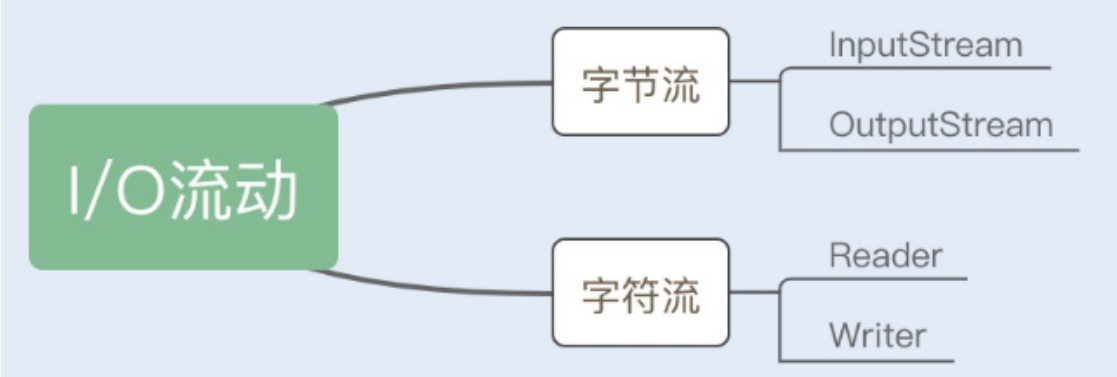
1、Java 中 IO 流分为几种?

- 按功能来分：**输入流（input）、输出流（output）。**
- 按类型来分：**字节流 和 字符流。**

字节流和字符流的区别是：

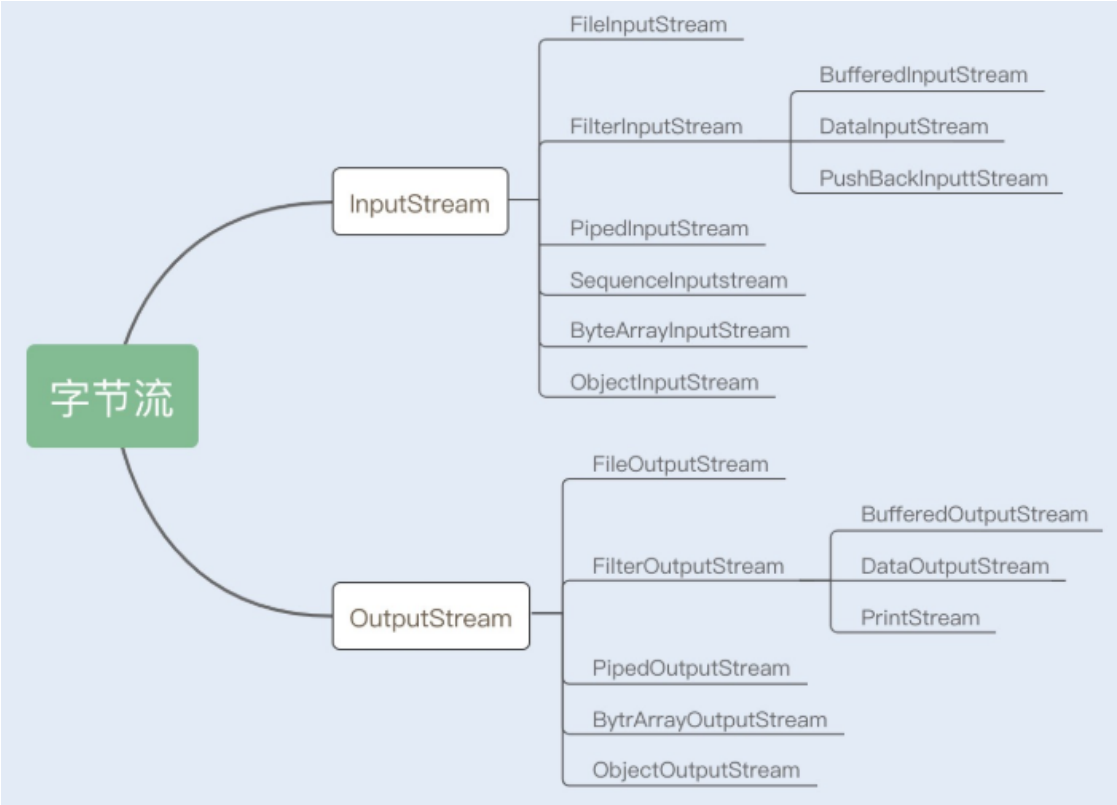
字节流按 8 位传输，以字节为单位输入输出数据，字符流按 16 位传输，以字符为单位输入输出数据。

但是不管文件读写还是网络发送接收，信息的最小存储单元都是字节。



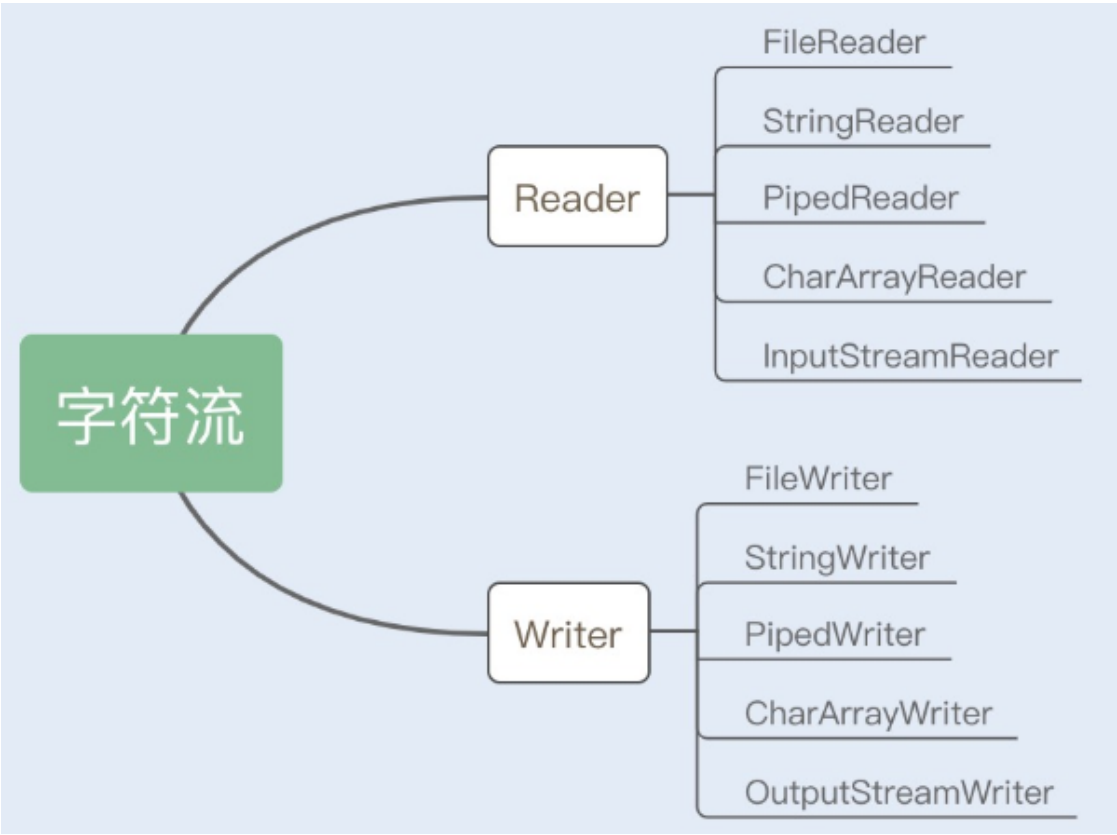
2、字节流

InputStream/OutputStream 是字节流的抽象类，这两个抽象类又派生了若干子类，不同的子类分别处理不同的操作类型。具体子类如下所示：



3、字符流

Reader/Writer 是字符的抽象类，这两个抽象类也派生了若干子类，不同的子类分别处理不同的操作类型。



4、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。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 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。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 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。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。

5、NIO

新的输入/输出 (NIO) 库是在 JDK 1.4 中引入的，弥补了原来的 I/O 的不足，提供了高速的、面向块的 I/O。

- **流与块**

I/O 与 NIO 最重要的区别是数据打包和传输的方式，I/O 以流的方式处理数据，而 NIO 以块的方式处理数据。

面向流的 I/O 一次处理一个字节数据：一个输入流产生一个字节数据，一个输出流消费一个字节数据。为流式数据创建过滤器非常容易，链接几个过滤器，以便每个过滤器只负责复杂处理机制的一部分。不利的一面是，面向流的 I/O 通常相当慢。

面向块的 I/O 一次处理一个数据块，按块处理数据比按流处理数据要快得多。但是面向块的 I/O 缺少一些面向流的 I/O 所具有的优雅性和简单性。

I/O 包和 NIO 已经很好地集成了，java.io.* 已经以 NIO 为基础重新实现了，所以现在它可以利用 NIO 的一些特性。例如，java.io.* 包中的一些类包含以块的形式读写数据的方法，这使得即使在面向流的系统中，处理速度也会更快。

- **通道与缓冲区**

1. 通道

通道 Channel 是对原 I/O 包中的流的模拟，可以通过它读取和写入数据。

通道与流的不同之处在于，流只能在一个方向上移动(一个流必须是 InputStream 或者 OutputStream 的子类)，而通道是双向的，可以用于读、写或者同时用于读写。

通道包括以下类型：

> FileChannel：从文件中读写数据；
>
> DatagramChannel：通过 UDP 读写网络中数据；
>
> SocketChannel：通过 TCP 读写网络中数据；
>
> ServerSocketChannel：可以监听新进来的 TCP 连接，对每一个新进来的连接都会创建一个 SocketChannel。

2. 缓冲区

发送给一个通道的所有数据都必须首先放到缓冲区中，同样地，从通道中读取的任何数据都要先读到缓冲区中。也就是说，不会直接对通道进行读写数据，而是要先经过缓冲区。

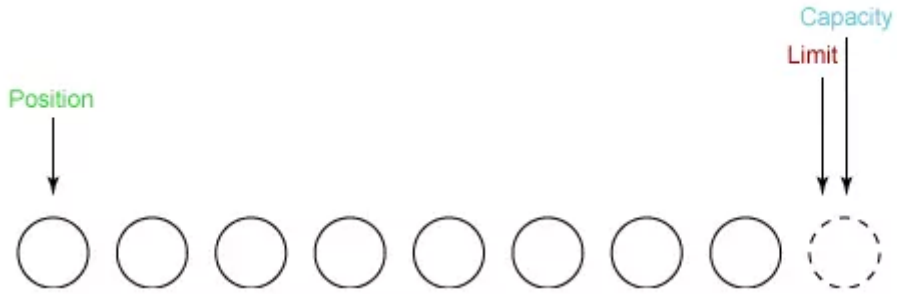
缓冲区实质上是一个数组，但它不仅仅是一个数组。缓冲区提供了对数据的结构化访问，而且还可以跟踪系统的读/写进程。缓冲区包括以下类型：**ByteBuffer、CharBuffer、ShortBuffer、IntBuffer、LongBuffer、FloatBuffer、DoubleBuffer。**

- **缓冲区状态变量**

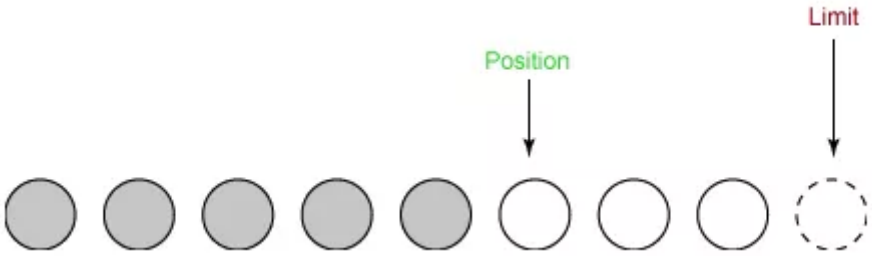
> **capacity**：最大容量；
>
> **position**：当前已经读写的字节数；
>
> **limit**：还可以读写的字节数。

状态变量的改变过程举例：

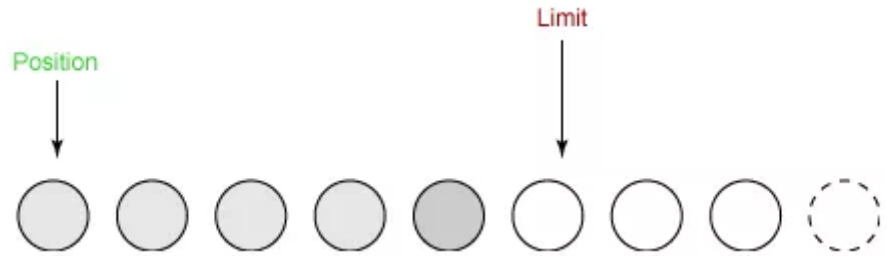
① 新建一个大小为 8 个字节的缓冲区，此时 position 为 0，而 limit = capacity = 8。capacity 变量不会改变，下面的讨论会忽略它。



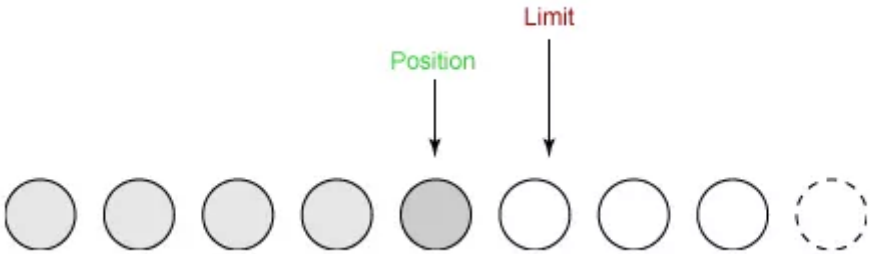
② 从输入通道中读取 5 个字节数据写入缓冲区中，此时 position 为 5，limit 保持不变。



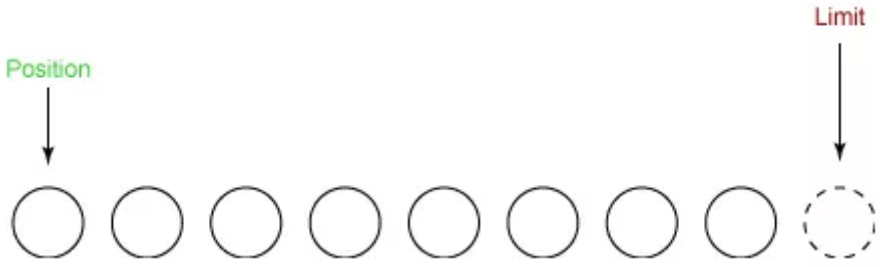
③ 在将缓冲区的数据写到输出通道之前，需要先调用 flip() 方法，这个方法将 limit 设置为当前 position，并将 position 设置为 0。



④ 从缓冲区中取 4 个字节到输出缓冲中，此时 position 设为 4。



⑤ 最后需要调用 clear() 方法来清空缓冲区，此时 position 和 limit 都被设置为最初位置。



- **选择器**

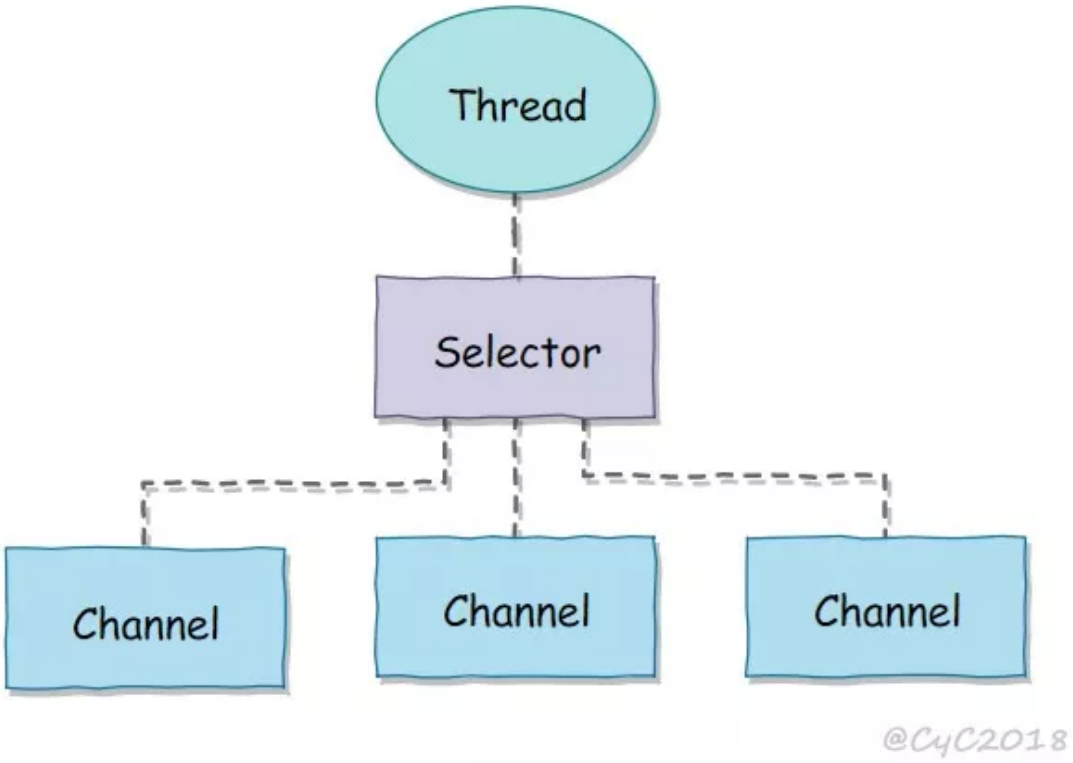
NIO 常常被叫做非阻塞 IO，主要是因为 NIO 在网络通信中的非阻塞特性被广泛使用。

NIO 实现了 IO 多路复用中的 Reactor 模型，**一个线程 Thread 使用一个选择器 Selector 通过轮询的方式去监听多个通道 Channel 上的事件，从而让一个线程就可以处理多个事件。**

通过配置监听的通道 Channel 为非阻塞，那么当 Channel 上的 IO 事件还未到达时，就不会进入阻塞状态一直等待，而是继续轮询其它 Channel，找到 IO 事件已经到达的 Channel 执行。

因为创建和切换线程的开销很大，因此使用一个线程来处理多个事件而不是一个线程处理一个事件，对于 IO 密集型的应用具有很好地性能。

应该注意的是，只有套接字 Channel 才能配置为非阻塞，而 FileChannel 不能，因为 FileChannel 配置非阻塞也没有意义。



1. 创建选择器

```
Selector selector = Selector.open();
```

2. 将通道注册到选择器上

```
ServerSocketChannel ssChannel = ServerSocketChannel.open();
ssChannel.configureBlocking(false);
ssChannel.register(selector, SelectionKey.OP_ACCEPT);
```

通道必须配置为非阻塞模式，否则使用选择器就没有任何意义了，因为如果通道在某个事件上被阻塞，那么服务器就不能响应其它事件，必须等待这个事件处理完毕才能去处理其它事件，显然这和选择器的作用背道而驰。

在将通道注册到选择器上时，还需要指定要注册的具体事件，主要有以下几类：

1. SelectionKey.OP_CONNECT
2. SelectionKey.OP_ACCEPT
3. SelectionKey.OP_READ
4. SelectionKey.OP_WRITE

它们在 SelectionKey 的定义如下：

```
public static final int OP_READ = 1 &lt;&lt; 0;
public static final int OP_WRITE = 1 &lt;&lt; 2;
public static final int OP_CONNECT = 1 &lt;&lt; 3;
public static final int OP_ACCEPT = 1 &lt;&lt; 4;
```

可以看出每个事件可以被当成一个位域，从而组成事件集整数。例如：

```
int interestSet = SelectionKey.OP_READ | SelectionKey.OP_WRITE;
```

3. 监听事件

```
int num = selector.select();
```

使用 select() 来监听到达的事件，它会一直阻塞直到有至少一个事件到达。

4. 获取到达的事件

```
Set<SelectionKey> keys = selector.selectedKeys();
Iterator<SelectionKey> keyIterator = keys.iterator();
while (keyIterator.hasNext()) {
        SelectionKey key = keyIterator.next();
        if (key.isAcceptable()) {
                // ...
        } else if (key.isReadable()) {
                // ...
        }
        keyIterator.remove();
}
```

5. 事件循环

因为一次 select() 调用不能处理完所有的事件，并且服务器端有可能需要一直监听事件，因此服务器端处理事件的代码一般会放在一个死循环内。

```
while (true) {
        int num = selector.select();
        Set<SelectionKey> keys = selector.selectedKeys();
        Iterator<SelectionKey> keyIterator = keys.iterator();
        while (keyIterator.hasNext()) {
                SelectionKey key = keyIterator.next();
                if (key.isAcceptable()) {
                        // ...
                } else if (key.isReadable()) {
                        // ...
                }
                keyIterator.remove();
        }
}
```

- **内存映射文件**

内存映射文件 I/O 是一种读和写文件数据的方法，它可以比常规的基于流或者基于通道的 I/O 快得多。

向内存映射文件写入可能是危险的，只是改变数组的单个元素这样的简单操作，就可能会直接修改磁盘上的文件。修改数据与将数据保存到磁盘是没有分开的。

下面代码行将文件的前 1024 个字节映射到内存中，map() 方法返回一个 MappedByteBuffer，它是 ByteBuffer 的子类。因此，可以像使用其他任何 ByteBuffer 一样使用新映射的缓冲区，操作系统会在需要时负责执行映射。

```
MappedByteBuffer mbb = fc.map(FileChannel.MapMode.READ_WRITE, 0, 1024);
```

- **对比**

NIO 与普通 I/O 的区别主要有以下两点：

> NIO 是非阻塞的；
>
> NIO 面向块，I/O 面向流。

十一、其他面试题

1、二维数组判空

二维数组为空，需要检查三个部分：

1、数组的首地址是否为空；

2、数组是：{}，即：array.length == 0 的情况；

3、数组是：{{}}，即：array.length() == 1，但是 array[0].length == 0。满足其中一个条件就直接返回 false 了；

- 综上：

```
if(arrray == null || array.length() == 0 || array.length == 1 && array[0].length == 0){
        return false;
}
```

2、Comparable 和 Comparator 区别比较

1、Comparable 相当于“内部比较器”，而 Comparator 相当于“外部比较器”；

2、Comparable 和 Comparator 是用来实现集合中元素的比较、排序的；

3、Comparable 是在集合内部定义的方法实现的排序，位于 java.lang 下；Comparator 在集合外部实现的排序，位于 java.util 下；

4、Comparable 是一个对象本身就已经支持自比较所需要实现的接口，如 String、Integer 自己就实现了 Comparable 接口，可完成比较大小操作。自定义类要在加入 list 容器中后能够排序，也可以实现 Comparable 接口，在用 Collections 类的 sort 方法排序时若不指定 Comparator，那就以自然顺序排序。所谓自然顺序就是实现 Comparable 接口设定的排序方式。

5、Comparator 是一个专用的比较器，当这个对象不支持自比较或者自比较函数不能满足要求时，可写一个比较器来完成两个对象之间大小的比较。

6、Comparator 体现了一种策略模式 (strategy design pattern)，就是不改变对象自身，而用一个策略对象 (strategy object) 来改变它的行为。

总而言之 Comparable 是自己完成比较，Comparator 是外部程序实现比较。