# java基础

## 重载和重写的区别

### 区别：

**重载**：Java中的方法重载发生在**同一个类**里面两个或者是多个方法的**方法名相同**但是**参数不同**的情况。

**重写：**方法重写是说**子类**重新定义了**父类的方法**。方法重写必须有相同的方法名，参数列表和返回类型。重写者可能不会限制它所重写的方法的访问。

### 重写的规则：

1）参数列表必须完全与被重写的方法相同，否则不能称其为重写而是重载。

2）返回的类型必须一直与被重写的方法的返回类型相同，否则不能称其为重写而是重载。

3）访问修饰符的限制一定要大于被重写方法的访问修饰符（public>protected>default>private）

4）重写方法一定不能抛出新的检查异常或者比被重写方法申明更加宽泛的检查型异常。例如：父类的一个方法申明了一个检查异常IOException，在重写这个方法是就不能抛出Exception,只能抛出IOException的子类异常，可以抛出非检查异常。

### 重载的规则：

1）必须具有不同的参数列表；

2）可以有不同的返回类型，只要参数列表不同就可以了；

3）可以有不同的访问修饰符；

4）可以抛出不同的异常；

## 接口和抽象类的区别

Java 提供和支持创建抽象类和接口。它们的实现有共同点，不同点在于：

1.接口中所有的**方法**隐含的都是抽象的。而抽象类则可以同时包含抽象和非抽象的方法。

2.类可以实现很多个接口，但是只能**继承**一个抽象类。

3.类如果要**实现**一个接口，它必须要实现接口声明的所有方法。但是，类可以不实现抽象类声明的所有方法，当然，在这种情况下，类也必须得声明成是抽象的。

4.抽象类可以在不提供接口方法实现的情况下实现接口。

5.Java 接口中声明的**变量**默认都是 public static final 的。抽象类可以包含非 final 的变量。

6.Java 接口中的**成员函数**默认是 public 的。抽象类的成员函数可以是 private，protected 或者是 public 。

7.接口是绝对抽象的，不可以被实例化。抽象类也不可以被实例化，但是，实例化子类的时候，就会初始化父类，不管父类是不是抽象类都会调用父类的构造方法，初始化一个类，先初始化父类。

## 抽象类有构造方法吗

//什么是构造方法？

A.方法名与类名相同；

B.没有返回类型（例如return、void等）；

C.不能被static、final、native、abstract和synchronized修饰，不能被子类继承。

D.父类的构造方法不能被子类调用，可以通过super语句调用父类的构造方法。

E.构造方法可以重载，以参数的个数，类型，顺序，分为空参构造方法和有参构造方法。//

在接口中不可以有构造方法，在抽象类中可以有构造方法。

在抽象类中可以有构造方法，只是不能直接创建抽象类的实例对象，但实例化子类的时候，就会初始化父类，不管父类是不是抽象类都会调用父类的构造方法，初始化一个类，先初始化父类。

## 抽象方法和类方法的区别？

1.抽象方法：

**抽象方法：在类中没有方法体的方法，就是抽象方法。**

2.类方法：

用static修饰的方法，也叫静态方法。

由于类方法是属于整个类的，所以类方法的方法体中不能有与类的对象有关的内容。

即类方法体有如下限制：

1.类方法中不能引用对象变量；

2.类方法中不能调用类的对象方法；

3.在类方法中不能调使用super，this关键字；

4.类方法不能被覆盖。

## 说一说Java的多态

<https://www.cnblogs.com/chenssy/p/3372798.html>

多态的定义：指允许不同类的对象对同一消息做出响应。即同一消息可以根据发送对象的不同而采用多种不同的行为方式。（发送消息就是函数调用）。

多态性：子类继承了来自父级类中的属性和方法，并对其中部分方法进行重写。于是多个子类中虽然都具有同一个方法，但是这些子类实例化的对象调用这些相同的方法后却可以获得完全不同的结果，这种技术就是多态性。多态性增强了软件的灵活性。

Java实现多态有三个必要条件：继承、重写、向上转型。

继承：在多态中必须存在有继承关系的子类和父类。

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。

向上转型：

如果我们用一个父类型去引用子类对象时，会先访问到子类中重写的父类方法（父类的方法不会再执行），如果子类没有重写父类的方法，才会执行父类中的重写办法。同时，子类中没有继承到父类的部分，是不能被执行的。

在多态中需要将子类的引用赋给父类对象，只有这样该引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，我们才能够在同一个继承结构中使用统一的逻辑实现代码处理不同的对象，从而达到执行不同的行为。

酒 a = 剑南春

      酒 b = 五粮液

      酒 c = 酒鬼酒

在上面的喝酒例子中，酒（Wine）是父类，剑南春（JNC）、五粮液（WLY）、酒鬼酒（JGJ）是子类。我们定义如下代码：

     JNC a = new  JNC();

     对于这个代码我们

      Wine a = new JNC();

      在这里我们这样理解，这里定义了一个Wine 类型的a，它指向JNC对象实例。由于JNC是继承与Wine，所以JNC可以自动向上转型为Wine，所以a是可以指向JNC实例对象的。这样做存在一个非常大的好处，在继承中我们知道子类是父类的扩展，它可以提供比父类更加强大的功能，如果我们定义了一个指向子类的父类引用类型，那么它除了能够引用父类的共性外，还可以使用子类强大的功能。

      但是向上转型存在一些缺憾，那就是它必定会导致一些方法和属性的丢失，而导致我们不能够获取它们。所以父类类型的引用可以调用父类中定义的所有属性和方法，对于只存在与子类中的方法和属性它就望尘莫及了。



  从程序的运行结果中我们发现，a.fun1()首先是运行父类Wine中的fun1().然后再运行子类JNC中的fun2()。

      分析：在这个程序中子类JNC重载了父类Wine的方法fun1()，重写fun2()，而且重载后的fun1(String a)与 fun1()不是同一个方法，由于父类中没有该方法，向上转型后会丢失该方法，所以执行JNC的Wine类型引用是不能引用fun1(String a)方法。而子类JNC重写了fun2() ，那么指向JNC的Wine引用会调用JNC中fun2()方法。

指向子类的父类引用由于向上转型了，它只能访问父类中拥有的方法和属性，而对于子类中存在而父类中不存在的方法，该引用是不能使用的，尽管是重载该方法。若子类重写了父类中的某些方法，在调用该些方法的时候，必定是使用子类中定义的这些方法（动态连接、动态调用）。

向下转型：

已经转为父类型的子类对象，再转回成子类型，才叫做向下转型。

**多态进行一个总结**：1、不管是向上转型，还是向下转型，我们涉及到的都是**子类和父类**的问题，也就是说多态必须存在继承关系；

2、正如上面所说，多态就是多种形态。也就是说，当我们需要实现多态的时候，就需要有**父类的方法被子类重写**。否则，如果没有重写的方法，就看不出多态的特性，一切按照父类的方法来，还不如不要继承，直接在父类中添加相应的方法，然后在实例化好了；

3、不管是向上转型还是向下转型，都有一个共性，就是**父类指向子类对象**。如果没有这个，也就没有了多态。

而关于多态的表现形式，分为两类：   
 **静态分派，即方法的重载**。表现为方法同名不同参，可以存在多个同名的方法，只要参数不一致就行，我们可以根据实际情况调用其中一个，所以称为静态分派。   
 **动态分派，即方法的重写。**表现为同名同参不同的执行操作。只能出现一个方法，而方法内部的操作时动态更改的。每次能且仅能调用一个方法。因为它并没有被人选择地特性，所以为动态。

## final，finally和finalize的区别

### 1.final

final关键字可以用于**类，方法，变量前**，用来表示该关键字修饰的类，方法，变量**具有不可变的特性**。

如果一个类被声明为final，意味着它不能再派生出新的子类，即不能被继承，因此它和 abstract 是反义词。

被声明为 final 的方法也同样只能使用，不能在子类中被重写。

将变量声明为 final，可以保证它们在使用中不被改变，被声明为 final 的变量必须在声明时给定初值，而在以后的引用中只能读取不可修改。

### 2.finally

通常放在 try…catch 的后面构造总是执行代码块，这就意味着程序无论正常执行还是发生异常，这里的代码只要 JVM 不关闭都能执行，可以将释放外部资源的代码写在 finally 块中。

### 3.finalize

Object 类中定义的方法，Java 中允许使用 finalize() 方法在垃圾收集器将对象从内存中清除出去之前做必要的清理工作。这个方法是由垃圾收集器在销毁对象时调用的，通过重写finalize() 方法可以整理系统资源或者执行其他清理工作。

## String、、StringBuilder和StringBuffer的区别

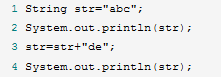
　这三个类之间的区别主要是在两个方面，即运行速度和线程安全这两方面。、

1.运行速度方面

运行速度快慢为：StringBuilder > StringBuffer > String；

String最慢的原因：

String为字符串常量，而StringBuilder和StringBuffer均为字符串变量，即String对象一旦创建之后该对象是不可更改的，但后两者的对象是变量，是可以更改的。



//如果运行这段代码会发现先输出“abc”，然后又输出“abcde”，好像是str这个对象被更改了，其实，这只是一种假象罢了，JVM对于这几行代码是这样处理的，首先创建一个String对象str，并把“abc”赋值给str，然后在第三行中，其实JVM又创建了一个新的对象也名为str，然后再把原来的str的值和“de”加起来再赋值给新的str，而原来的str就会被JVM的垃圾回收机制（GC）给回收掉了，所以，str实际上并没有被更改，也就是前面说的String对象一旦创建之后就不可更改了。所以，Java中对String对象进行的操作实际上是一个不断创建新的对象并且将旧的对象回收的一个过程，所以执行速度很慢。

2. 线程安全

　　在线程安全上，StringBuilder是线程不安全的，而StringBuffer是线程安全的

如果一个StringBuffer对象在字符串缓冲区被多个线程使用时，StringBuffer中很多方法可以带有synchronized关键字，所以可以保证线程是安全的，但StringBuilder的方法则没有该关键字，所以不能保证线程安全，有可能会出现一些错误的操作。所以如果要进行的操作是多线程的，那么就要使用StringBuffer，但是在单线程的情况下，还是建议使用速度比较快的StringBuilder。

　3. 总结一下  
　　String：适用于少量的字符串操作的情况

　　StringBuilder：适用于单线程下在字符缓冲区进行大量操作的情况

　　StringBuffer：适用多线程下在字符缓冲区进行大量操作的情况

## static的理解

static修饰的成员**变量**和成员**方法独立于**该类的任何**对象**。也就是说，它不依赖类特定的实例，**被类的所有实例共享**。

只要**这个类被加**载，Java虚拟机就能**根据类名**在运行时数据区的方法区内定找到他们。因此，static对象可以在它的任何对象创建之前访问，无需引用任何对象。static方法是编译时静态绑定。

static可以用来修饰类的成员方法、类的成员变量，另外可以编写static代码块来优化程序性能。

1）static变量

　static变量也称作静态变量，静态变量和非静态变量的区别是：静态变量被所有的对象所共享，在内存中只有一个副本，它当且仅当在类初次加载时会被初始化。而非静态变量是对象所拥有的，在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。

　　static成员变量的初始化顺序按照定义的顺序进行初始化。

2）static方法

static方法一般称作静态方法，由于静态方法不依赖于任何对象就可以进行访问，因此对于静态方法来说，是没有this的，因为它不依附于任何对象，既然都没有对象，就谈不上this了。并且由于这个特性，在静态方法中不能访问类的非静态成员变量和非静态成员方法，因为非静态成员方法/变量都是必须依赖具体的对象才能够被调用。

但是要注意的是，虽然在静态方法中不能访问非静态成员方法和非静态成员变量，但是在非静态成员方法中是可以访问静态成员方法/变量的。

3）static代码块

static代码块也叫静态代码块，是在类中独立于类成员的static语句块，可以有多个，位置可以随便放，它不在任何的方法体内，JVM加载类时会执行这些静态的代码块，如果static代码块有多个，JVM将按照它们在类中出现的先后顺序依次执行它们，每个代码块只会被执行一次，所以说static块可以用来优化程序性能。

stati**c方法块**和static**方法**的**区别**：

静态**代码块**是**自动执行的**;

静态**方法**是**被调用的时候才执行**的。

## java如何判断两个对象是否相等

1.java中的基本数据类型

（int、double、short、long、byte、float、boolean、char）判断是否相等，直接使用"=="就行了，相等返回true，否则，返回false

2.java中的引用类型的对象

假设有两个引用对象obj1,obj2,obj1==obj2 判断是obj1,obj2这两个引用变量是否相等，即它们所指向的对象是否为同一个对象。言外之意就是要求两个变量所指内存地址相等的时候，才能返回true，每个对象都有自己的一块内存，因此必须指向同一个对象才返回ture。

3.如果想要自定义两个对象

（不是一个对象，即这两个对象分别有自己的一块内存）是否相等的规则，那么必须在对象的类定义中重写equals()方法，如果不重写equals()方法的话，默认的比较方式是比较两个对象是否为同一个对象==。

## 八．==和equals的区别(比较方式)

==操作符专门用来比较两个基本类型变量的值是否相等

==用来比较两个引用类型的变量的内存地址是否相等，也就是是否为同一个对象。

equals方法是用于比较两个独立对象的内容是否相等。如果类没有重写equals方法，那么默认是Object类的equals方法，也就是==。

## 对Java的参数传递是怎么理解的

Java中没有真正的引用传递 只有值传递！

1..按值传递：

指的是在方法调用时，传递的参数是按值的拷贝传递。按值传递重要特点：传递的是值的拷贝，也就是说传递后就互不相关了。示例如下：



2.按引用传递：

指的是在方法调用时，传递的参数是按引用进行传递，其实传递的引用的地址，也就是变量所对应的内存空间的地址。

按引用传递的重要特点：传递的是值的引用，也就是说传递前和传递后都指向同一个引用（也就是同一个内存空间）。



（1）“在Java里面参数传递都是按值传递”这句话的意思是：按值传递是传递的值的拷贝，按引用传递其实传递的是引用的地址值，所以统称按值传递。

（2）在Java里面只有基本类型和按照下面这种定义方式的String是按值传递，其它的都是按引用传递。就是直接使用双引号定义字符串方式：String str = “Java私塾”;

## 子类继承父类一定要继承父类的构造方法吗

不是。构造方法是一个类独有的方法不会被它的子类继承，但是子类在实例化对象时：如果子类的构造器没有显示的调用父类的构造器，则将自动调用父类默认的构造器（无参）。相当于默认省略了super()。

关于子类调用父类的构造方法：

子类实例化对象时，会先调用父类的构造器；

如果子类的构造器没有显示的调用父类的构造器 ，默认调用super();

子类要使用父类有参的构造器，使用super（参数）形式，且super必须是子类构造方法中的头一条语句；

如果父类没有不带参数的构造器，且子类的构造器中又没显示的调用父类其他构造器，则Java编译器将报告错误。

## 说下Java的克隆体系

提供了**Cloneable接口。**在Java中所有类都实现了clone()方法，因为Java中的所有类的**父类Object具有clone()方法**，而所有Java类都是Object类的子类。

某个对象如果想要能够**实现克隆**，那么该对象的类**必须实现cloneable接口**。Cloneable接口仅仅只是一个标记接口，不包含任何需要实现的方法。如果某个对象的类没有实现Cloneable接口而调用clone()方法，那么会抛出CloneNotSupportedException。

**深克隆&浅克隆**

**所谓浅克隆是指复制一个对象的实例，但是这个对象中包含的其它的对象还是共用的。**

**所谓深克隆是指复制一个对象的实例，而且这个对象中包含的其它的对象也要复制一份。**

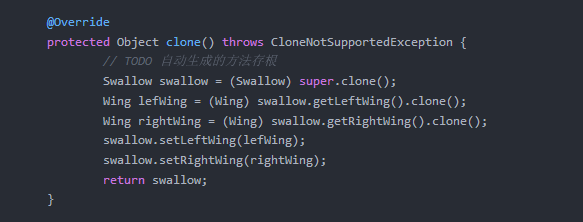
如何实现对象的克隆？分三步：

1.对象的类实现Cloneable接口；

2.覆盖Object类的clone()方法；

3.在clone()方法中调用super.clone()；

把浅克隆改成深克隆的方法也很简单，就是在clone方法中把引用类型的成员变量单独调一次clone方法：

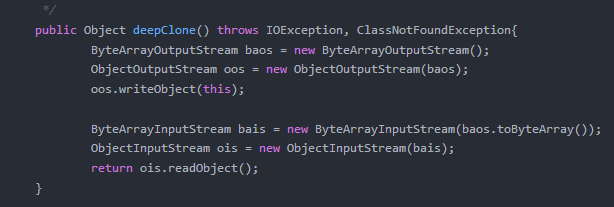


深克隆的问题：

对象包含多个引用类型的成员，同时这些成员又包含了引用类型的成员，那层层克隆岂不是相当繁琐，并且维护不便。

通过序列化(Serialization)方式来实现

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



## 反射

### 1.反射的概述

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。

反射就是把java类中的各种成分映射成一个个的Java对象。

### 2.反射机制的作用

a.在运行时判断任意一个对象所属的类；

b.在运行时**判断**任意**一个类**所具有的**成员变量和方法**。

c.在运行时获取类的对象；

d.在运行时访问java对象的属性，方法，构造方法等。

### 3.反射机制的优缺点？

**静态编译**：在编译时确定类型，绑定对象，即通过。   
**动态编译**：运行时确定类型，绑定对象。动态编译最大限度发挥了java的灵活性，体现了多态的应用，有以降低类之间的藕合性。

反射机制的优点：可以实现动态创建对象和编译，体现出很大的灵活性。通过反射机制我们可以获得类的各种内容，进行了反编译。对于JAVA这种先编译再运行的语言来说，反射机制可以使代码更加灵活，更加容易实现面向对象。

　　比如，一个大型的软件，不可能一次就把把它设计的很完美，当这个程序编译后，发布了，当发现需要更新某些功能时，我们不可能要用户把以前的卸载，再重新安装新的版本，假如这样的话，这个软件肯定是没有多少人用的。采用静态的话，需要把整个程序重新编译一次才可以实现功能的更新，而采用反射机制的话，它就可以不用卸载，只需要在运行时才动态的创建和编译，就可以实现该功能。

反射机制的缺点：对性能有影响。使用反射基本上是一种解释操作，我们可以告诉JVM，我们希望做什么并且它 满足我们的要求。这类操作总是慢于只直接执行相同的操作。

### 4.哪里用到反射机制？

jdbc中有一行代码：Class.forName('com.[MySQL](http://lib.csdn.net/base/mysql).jdbc.Driver.class').newInstance();那个时候只知道生成驱动对象实例，后来才知道，这就是反射，现在很多框架都用到反射机制，[hibernate](http://lib.csdn.net/base/javaee)，struts都是用反射机制实现的。

Spring的IOC中应用了反射机制。

Spring使用步骤：

1.找到配置文件

2.加载配置文件

3.解析配置文件中的bean元素，并识别id和class

4**.通过反射（Class.forName().newInstance()）创建这个bean的实例**

5.将id作为key、实例作为value存放进Spring容器中

6.getBean取出实例

### JDBC的反射？？

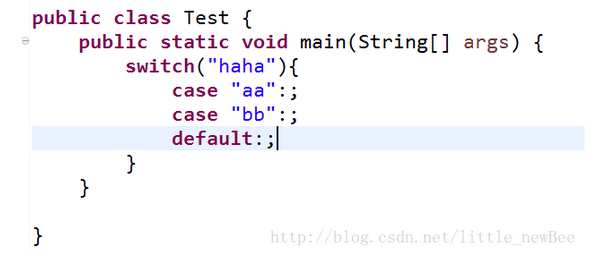
## 抽象类中可以有静态方法吗？

[Java抽象类](https://www.baidu.com/s?wd=Java%E6%8A%BD%E8%B1%A1%E7%B1%BB&tn=SE_PcZhidaonwhc_ngpagmjz&rsv_dl=gh_pc_zhidao)中不能有静态的抽象方法。  
原因：抽象类是不能实例化的，即不能被[分配内存](https://www.baidu.com/s?wd=%E5%88%86%E9%85%8D%E5%86%85%E5%AD%98&tn=SE_PcZhidaonwhc_ngpagmjz&rsv_dl=gh_pc_zhidao);而static修饰的方法在类实例化之前就已经别分配了内存，这样一来矛盾就出现了：抽象类不能被[分配内存](https://www.baidu.com/s?wd=%E5%88%86%E9%85%8D%E5%86%85%E5%AD%98&tn=SE_PcZhidaonwhc_ngpagmjz&rsv_dl=gh_pc_zhidao)，而static方法必须被[分配内存](https://www.baidu.com/s?wd=%E5%88%86%E9%85%8D%E5%86%85%E5%AD%98&tn=SE_PcZhidaonwhc_ngpagmjz&rsv_dl=gh_pc_zhidao)。所以抽象类中不能有静态的抽象方法。  
另外，定义抽象方法的目的是重写此方法，但如果定义成静态方法就不能被重写。

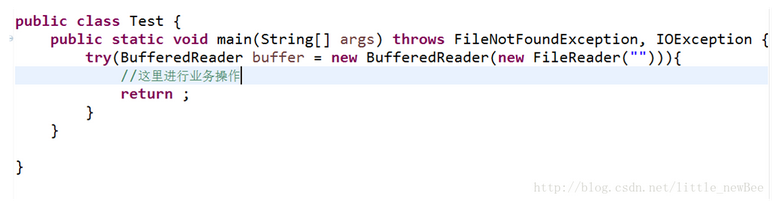
## 对Java7、8的新特性有什么了解

### java7新特性:

1.Switch中支持string类型啦。

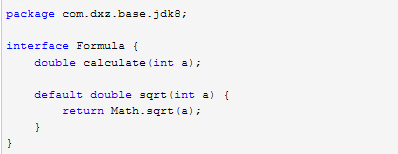


2. 支持try语句块申请更多空间，自动释放资源。



### Java8新特性

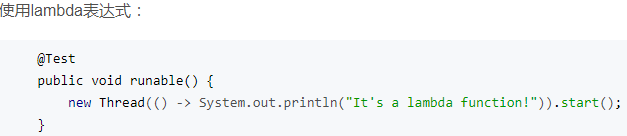
1.接口的默认方法Java 8允许我们给接口添加一个非抽象的方法实现，只需要使用 default关键字即可，这个特征又叫做扩展方法，示例如下：



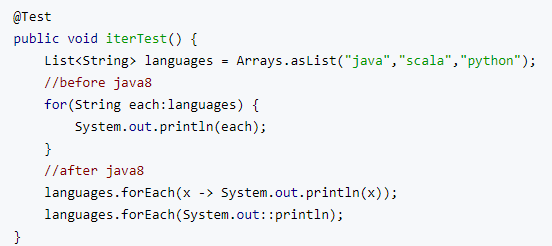
#### 2.****Lambda 表达式****

允许我们将行为传到函数中

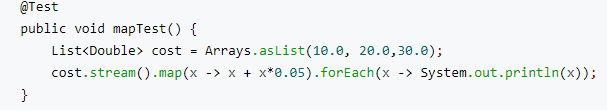
**1）替代匿名内部类**



**2）使用lambda表达式对集合进行迭代**



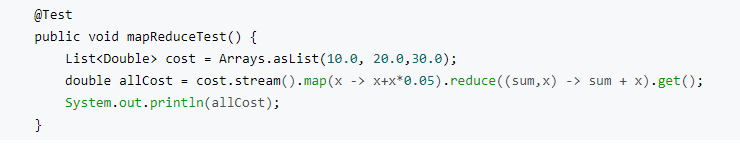
**3）用lambda表达式实现map**



map的作用是将一个对象变换为另外一个

**4）用lambda表达式实现map与reduce**

reduce实现的则是将所有值合并为一个



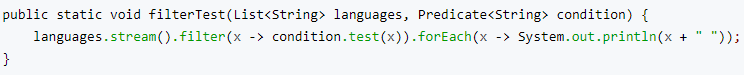
**5）filter操作**

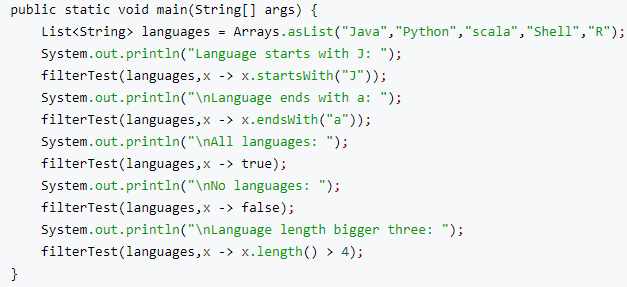
原始的集合中过滤掉一部分元素



**6）与函数式接口Predicate配合**

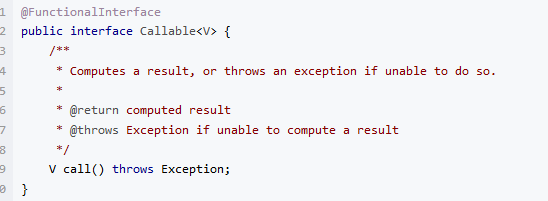
Java 8也添加了一个包，叫做 **java.util.function**。它包含了很多类，用来支持Java的函数式编程。其中一个便是Predicate，使用 java.util.function.Predicate 函数式接口以及lambda表达式，可以向API方法添加逻辑，用更少的代码支持更多的动态行为。Predicate接口非常适用于做过滤。





#### 3.函数式接口

为了更友好方便 使用Lambda表达式，JDK设计了函数式接口。函数式接口就是一个具有一个方法的普通接口。函数式接口可以被隐式转换为lambda表达式。java.lang.Runnable与java.util.concurrent.Callable是函数式接口最典型的两个例子。在实际使用过程中，函数式接口是容易出错的：**如有某个人在接口定义中增加了另一个方法，这时，这个接口就不再是函数式的了，并且编译过程也会失败**。为了克服函数式接口的这种脆弱性并且能够明确声明接口作为函数式接口的意图，Java 8增加了一种特殊的注解@FunctionalInterface（Java 8中所有类库的已有接口都添加了@FunctionalInterface注解）



3.Stream

Stream 是java8最重要的新特新之一，真正的函数式编程风格引入到Java中，它大大简化了我们的编码量，让程序员写出高效率、干净、简洁的代码。

**几个重要的方法：**

1. **stream** 返回顺序流，集合作为其源。
2. **parallelStream** 返回并行数据流， 集合作为其源。
3. **filter** 方法用于消除基于标准元素
4. **map** 方法用于映射每个元素对应的结果
5. **forEach** 方法遍历该流中的每个元素
6. **limit** 方法用于减少流的大小
7. **sorted** 方法用来流排序
8. **collect** 方法是终端操作，这是通常出现在管道传输操作结束标记流的结束。

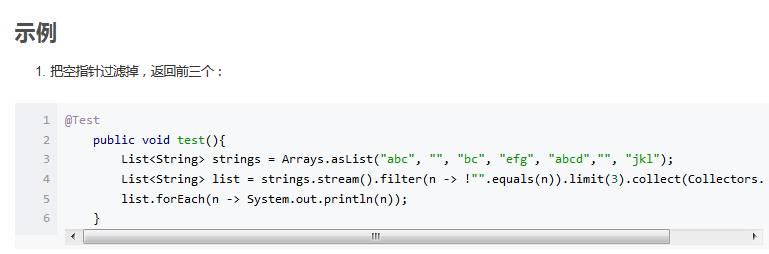
Stream 的方法分为两类。一类叫惰性求值，一类叫及早求值。

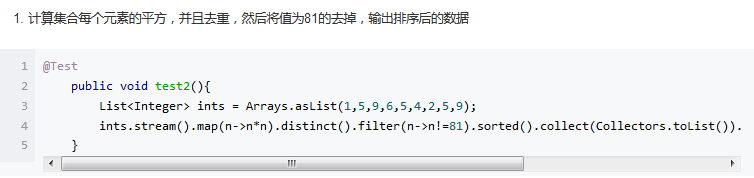
如果返回值是 Stream，那么是惰性求值。其实可以这么理解，如果调用惰性求值方法，Stream 只是记录下了这个惰性求值方法的过程，并没有去计算，等到调用及早求值方法后，就连同前面的一系列惰性求值方法顺序进行计算，返回结果。

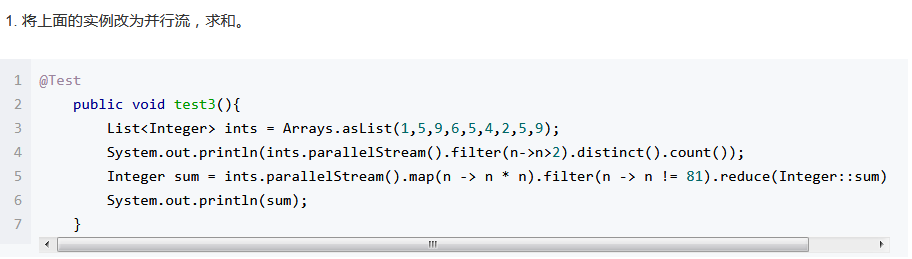
Stream.惰性求值.惰性求值. ... .惰性求值.及早求值

**collect(toList())**

collect(toList()) 方法由 Stream 里的值生成一个列表，是一个及早求值操作。可以理解为 Stream 向 Collection 的转换.

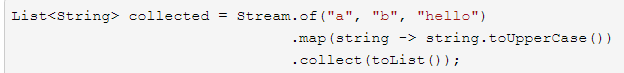






**map**

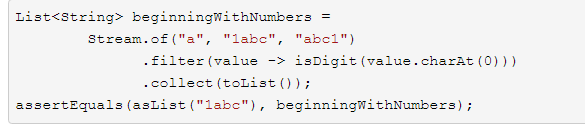
如果有一个函数可以将一种类型的值转换成另外一种类型，map 操作就可以使用该函数，将一个流中的值转换成一个新的流。



 map 方法就是接受的一个 Function 的匿名函数类，进行的转换。



**filter**

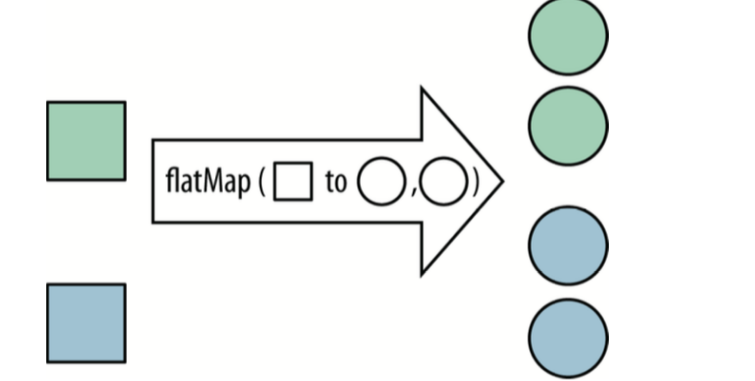


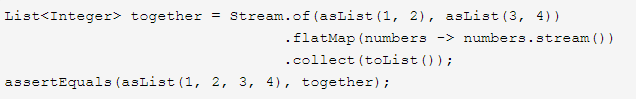
filter 方法就是接受的一个 Predicate 的匿名函数类，判断对象是否符合条件，符合条件的才保留下来



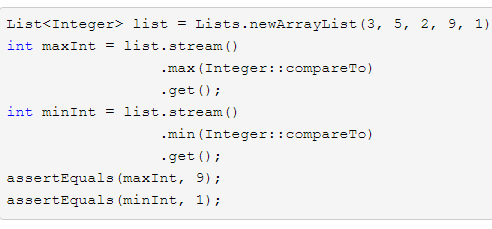
**flatMap**

flatMap 方法可用 Stream 替换值，然后将多个 Stream 连接成一个 Stream





**max和min**

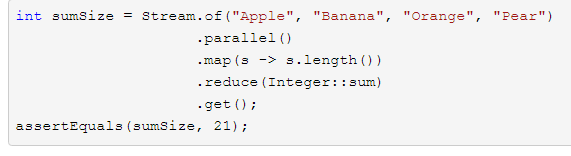


Integer::compareTo 也是属于 Java 8 引入的新特性，叫做 **方法引用（Method References）**。在这边，其实就是 (int1, int2) -> int1.compareTo(int2) 的简写。

**数据并行化操作**

Stream 的并行化也是 Java 8 的一大亮点。数据并行化是指将数据分成块，为每块数据分配单独的处理单元。这样可以充分利用多核 CPU 的优势。

并行化操作流只需改变一个方法调用。如果已经有一个 Stream 对象，调用它的 parallel() 方法就能让其拥有并行操作的能力。如果想从一个集合类创建一个流，调用 parallelStream() 就能立即获得一个拥有并行能力的流。



数据并行化会先对数据进行分块，然后对每块数据开辟线程进行运算，这些地方会花费额外的时间。并行化操作只有在 **数据规模比较大** 或者 **数据的处理时间比较长** 的时候才能体现出优势，所以并不是每个地方都需要让数据并行化，应该具体问题具体分析。

**stream迭代的优势和区别**

stream迭代可以发挥多核优势，可以并行处理，而不像外部迭代需要串行处理。

## construct的调用顺序（子类父类之间）

初始化**父类的静态**成员

初始化**父类的静态**代码块

静态成员、静态代码块

初始化**子类的静态**成员

初始化**子类的静态**代码块

————————————

初始化**父类的非静态**成员

初始化**父类的非静态**代码块

非静态成员、代码块、构造方法

初始化**父类**的**构造方法**

初始化**子类的非静态**成员

初始化**子类的非静态**代码块

初始化**子类**的构造方法

## StringBuffer和StringBuilder有什么区别，底层实现上呢

**区别**

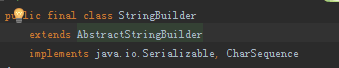
1、StringBuffer 与 StringBuilder 中的方法和功能完全是等价的。

2、只是StringBuffer 中的方法大都采用了 synchronized 关键字进行修饰，因此是线程安全的，而 StringBuilder 没有这个修饰，可以被认为是线程不安全的。

3、在单线程程序下，StringBuilder效率更快，因为它不需要加锁，不具备多线程安全而StringBuffer则每次都需要判断锁，效率相对更低。

**实现原理**

StringBuffer类继承自AbstractStringBuilder抽象类，实现Serializable序列化接口和CharSequence接口。



AbstractStringBuilder抽象类实现Appendabel，CharSequence接口。

**StringBuffer初始化及扩容机制**

1.StringBuffer()的初始容量可以容纳16个字符，当该对象的实体存放的字符的长度大于16时，实体容量就自动增加。StringBuffer对象可以通过length()方法获取实体中存放的字符序列长度，通过capacity()方法来获取当前实体的实际容量。

2.StringBuffer(int size)可以指定分配给该对象的实体的初始容量参数为参数size指定的字符个数。当该对象的实体存放的字符序列的长度大于size个字符时，实体的容量就自动的增加。以便存放所增加的字符。

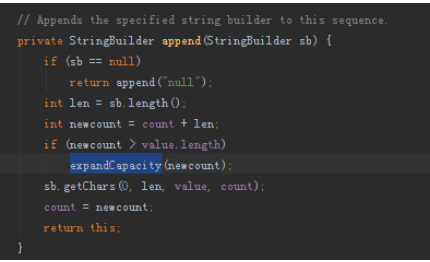
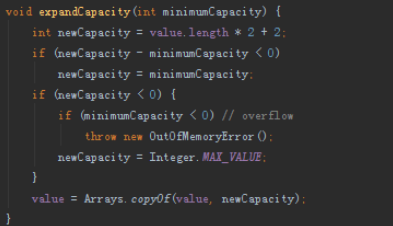
3.StringBuffer(String s)可以指定给对象的实体的初始容量为参数字符串s的长度额外再加16个字符。当该对象的实体存放的字符序列长度大于size个字符时，实体的容量自动的增加，以便存放所增加的字符。

**扩容算法：**

使用append()方法在字符串后面追加东西的时候，如果长度超过了该字符串存储空间大小了就需要进行扩容：构建新的存储空间更大的字符串，将旧的复制过去；

在进行字符串append添加的时候，会先计算添加后字符串大小，然后使用ensureCapacityInternal 这个方法进行是否扩容的判断，需要扩容就调用expandCapacity方法进行扩容

尝试将新容量扩为大小变成2倍+2   ，if 判断一下 容量如果不够，直接扩充到需要的容量大小。

## Reader和InputStream区别

java.io下面有两个抽象类：InputStream和Reader。  
InputStream是表示字节输入流的所有类的超类；  
Reader是用于读取字符流的抽象类；  
InputStream提供的是字节流的读取，而非文本读取，这是和Reader类的根本区别。  
即用Reader读取出来的是char数组或者String ，使用InputStream读取出来的是byte数组。

### 1.InputStream

   | \_\_FileInputStream   
**1.1 FileInputStream**

FileInputStream从文件系统中的某个文件中获得输入字节。  
构造方法摘要  :  
FileInputStream (File  file)   
          通过打开一个到实际文件的连接来创建一个 FileInputStream ，该文件通过文件系统中的 File 对象 file 指定。   
FileInputStream (FileDescriptor  fdObj)   
          通过使用文件描述符 fdObj 创建一个 FileInputStream ，该文件描述符表示到文件系统中某个实际文件的现有连接。   
FileInputStream (String  name)   
          通过打开一个到实际文件的连接来创建一个 FileInputStream ，该文件通过文件系统中的路径名 name 指定。

2.Reader  
  
   |——BufferedReader   
   |\_\_\_InputStreamReader   
         |\_\_FileReader

**2.1 BufferedReader**

从字符输入流中读取文本，缓冲各个字符，从而实现字符、数组和行的高效读取。

构造方法摘要  :  
BufferedReader (Reader  in)   
          创建一个使用默认大小输入缓冲区的缓冲字符输入流。   
BufferedReader (Reader  in, int sz)   
          创建一个使用指定大小输入缓冲区的缓冲字符输入流。   
BufferedReader (Java Platform SE 6)   
BufferedReader的最大特点就是缓冲区的设置。通常Reader 所作的每个读取请求都会导致对底层字符或字节流进行相应的读取请求，如果没有缓冲，则每次调用  read() 或 readLine() 都会导致从文件中读取字节，并将其转换为字符后返回，而这是极其低效的。   
使用BufferedReader可以指定缓冲区的大小，或者可使用默认的大小。大多数情况下，默认值就足够大了。   
因此，建议用 BufferedReader 包装所有其 read() 操作可能开销很高的 Reader（如 FileReader 和InputStreamReader）。例如，   
 BufferedReader in  
   = new BufferedReader(new FileReader("foo.in"));  
 将缓冲指定文件的输入。

**2.2 InputStreamReader**

InputStreamReader是字节流通向字符流的桥梁：它使用指定的 charset 读取字节并将其解码为字符。它使用的字符集可以由名称指定或显式给定，或者可以接受平台默认的字符集。

每次调用 InputStreamReader 中的一个 read() 方法都会导致从底层输入流读取一个或多个字节。要启用从字节到字符的有效转换，可以提前从底层流读取更多的字节，使其超过满足当前读取操作所需的字节。

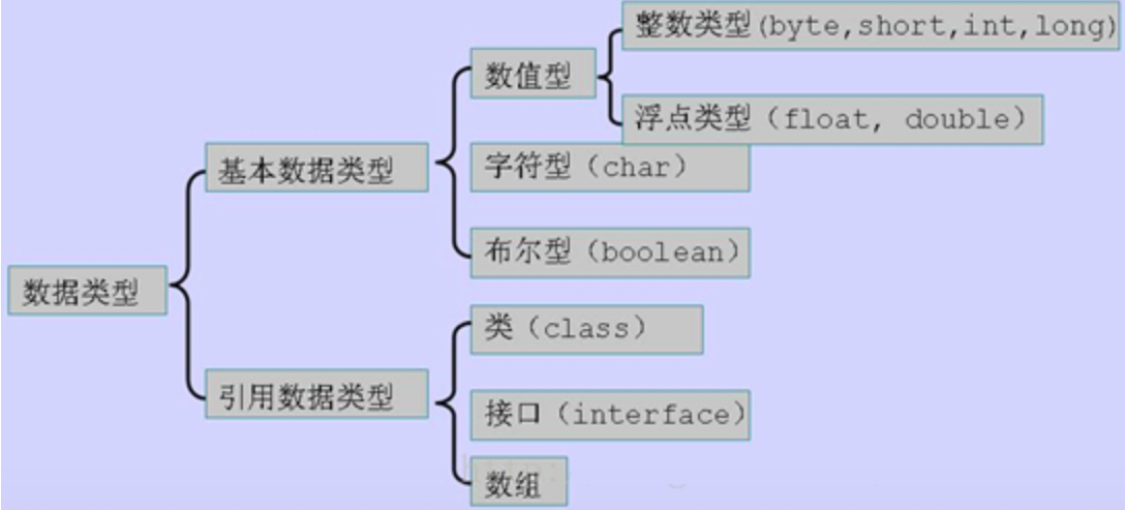
为了达到最高效率，可要考虑在 BufferedReader 内包装 InputStreamReader。例如：   
BufferedReader in = new BufferedReader(new InputStreamReader(System.in));  
InputStreamReader最大的特点是可以指转换的定编码格式，这是其他类所不能的，从构造方法就可看出，这一点在读取中文字符时非常有用。

**2.3FileReader**  
1）FileReader类介绍：  
InputStreamReader类的子类，所有方法（read（）等）都从父类InputStreamReader中继承而来；  
2）与InputStreamReader类的区别：  
构造方法摘要  ：  
FileReader (File  file)   
          在给定从中读取数据的 File 的情况下创建一个新 FileReader 。   
FileReader (FileDescriptor  fd)   
          在给定从中读取数据的 FileDescriptor 的情况下创建一个新 FileReader 。   
FileReader (String  fileName)   
          在给定从中读取数据的文件名的情况下创建一个新 FileReader

该类与它的父类InputStreamReader的主要不同在于构造函数，主要区别也就在于构造函数！从InputStreamReader的构造函数中看到，参数为InputStream和编码方式，可以看出，当要指定编码方式时，必须使用InputStreamReader类；而FileReader构造函数的参数与FileInputStream同，为File对象或表示path的String，可以看出，当要根据File对象或者String读取一个文件时，用FileReader。

**联系与区别**  
（1）字符与字节：   
FileInputStream 类以二进制输入/输出，I/O速度快且效率搞，但是它的read（）方法读到的是一个字节（二进制数据），很不利于人们阅读，而且无法直接对文件中的字符进行操作，比如替换，查找（必须以字节形式操作）；  
而Reader类弥补了这个缺陷，可以以文本格式输入/输出，非常方便；比如可以使用while((ch = filereader.read())!=-1 )循环来读取文件；可以使用BufferedReader的readLine()方法一行一行的读取文本。  
（2）编码  
InputStreamReader ，它是字节转换为字符的桥梁。 你可以在构造器重指定编码的方式，如果不指定的话将采用底层操作系统的默认编码方式，例如GBK等。   
FileReader与InputStreamReader 涉及编码转换(指定编码方式或者采用os默认编码)，可能在不同的平台上出现乱码现象！而FileInputStream 以二进制方式处理，不会出现乱码现象.   
因此要指定编码方式时，必须使用InputStreamReader 类，所以说它是字节转换为字符的桥梁；  
(3) 缓存区  
    BufferReader类用来包装所有其 read() 操作可能开销很高的 Reader（如 FileReader 和InputStreamReader）。

## Java基础的数据类型和占多少字节



| 类型 | 字节数 | 取值范围 |
| --- | --- | --- |
| byte | 1 | −27——27−1 |
| short | 2 | −215——215 −1 |
| int | 4 | −231−231——231−1 |
| long | 8 | −263——263−1 |
| float | 4 | 与double的区别在于float类型有效小数点只有6~7位） |
| double | 8 | 范围更大，精度更高 |
| char | 2 | \u0000~\uFFFF |
| boolean | 1 | 前七位是0 |

## Java中的数据类型与内存的关系

1）基本数据类型的存储原理：所有的简单数据类型不存在“引用”的概念，基本数据类型都是直接存储在内存中的内存栈上的，数据本身的值就是存储在栈空间里面，而Java语言里面八种数据类型是这种存储模型；

2）引用类型的存储原理:引用类型继承于Object类（也是引用类型）都是按照Java里面存储对象的内存模型来进行数据存储的，使用Java内存堆和内存栈来进行这种类型的数据存储，简单地讲，“引用”是存储在有序的内存栈上的，而对象本身的值存储在内存堆上的。

**区别:基本数据类型和引用类型的区别主要在于基本数据类型是分配在栈上的，而引用类型是分配在堆上的（需要java中的栈、堆概念）**。

## OOP（Object Oriented Programming）特性

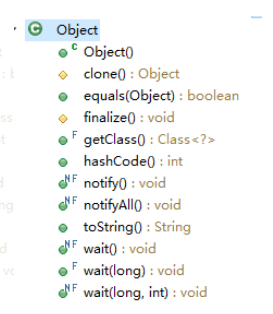
OOP具有三大特点 ：

1、封装性：也称为信息隐藏，就是将一个类的使用和实现分开，只保留部分接口和方法与外部联系，或者说只公开了一些供开发人员使用的方法。于是开发人员只需要关注这个类如何使用，而不用去关心其具体的实现过程，这样就能实现MVC分工合作，也能有效避免程序间相互依赖，实现代码模块间松藕合。

2、继承性：就是子类自动继承其父级类中的属性和方法，并可以添加新的属性和方法或者对部分属性和方法进行重写。继承增加了代码的可重用性。PHP只支持单继承，也就是说一个子类只能有一个父类。

3、多态性：子类继承了来自父级类中的属性和方法，并对其中部分方法进行重写。于是多个子类中虽然都具有同一个方法，但是这些子类实例化的对象调用这些相同的方法后却可以获得完全不同的结果，这种技术就是多态性。多态性增强了软件的灵活性。

## object类有哪些方法



Object类有12个成员方法：

* protected Object **clone()**创建并返回此对象的一个副本。
* public boolean **equals(Object obj)**指示其他某个对象是否与此对象“相等”。
* protected void **finalize()**当**垃圾回收器确定不存在对该对象的更多引用时**，由**对象的垃圾回收器调用此方法**。
* public final **native Class< ? > getClass()** 返回此 Object 的运行时类。
* public int **hashCode()**返回该对象的**哈希码值**。
* public void **notify()**唤醒在**此对象监视器**上**等待的单个线程**。
* public void **notifyAll()**唤醒在此对象监视器上**等待的所有线程**。
* public String **toString()**返回该**对象的字符串**表示。
* public void **wait()**在其他线程调用此对象的 **notify()** 方法或 notifyAll() 方法前，导致**当前线程等待**。
* public void **wait(long timeout)**在其他线程调用此对象的 notify() 方法或 notifyAll() 方法，或者超过指定的时间量前，导致当前线程等待。
* public void **wait(long timeout, int nanos)** 在其他线程调用此对象的 notify() 方法或 notifyAll() 方法，或者其他某个线程中断当前线程，或者已超过某个实际时间量前，导致当前线程等待。

## 泛型在编译期和运行期的作用

泛型的本质是为了参数化类型（在不创建新的类型的情况下，通过泛型指定的不同类型来控制形参具体限制的类型）。操作的数据类型被指定为一个参数，这种参数类型可以用在类、接口和方法中，分别被称为泛型类、泛型接口、泛型方法。

泛型值存在于java的编译期，编译后生成字节码文件泛型是被擦除的。

泛型只是在java的编译期会产生错误但是在java的运行期（已经生成字节码文件后）是会被擦除的，这个期间并没泛型的存在。

## 怎么一行一行的读文件



## BufferedReader读取文件乱码怎么解决

解決BufferedReader读取UTF-8文件中文乱码  
读取txt文件乱码   
【Java代码】  
　　BufferedReader read = new BufferedReader(new FileReader(new File(filename)));  
解决办法：   
【Java代码】  
　　InputStreamReader isr = new InputStreamReader(new FileInputStream(file), "UTF-8");  
　　BufferedReader read = new BufferedReader(isr);  
因为InputStreamReader和BufferedReader 都继承自Reader,而BufferedReader 的构造器又是Reader.   
Writer也是一样的处理   
【Java代码】  
　　FileOutputStream writerStream = new FileOutputStream(filePath, true);  
　　BufferedWriter oWriter = new BufferedWriter(new OutputStreamWriter(writerStream, "UTF-8"));



1. Reader用到什么设计模式
2. 一致性哈希是干嘛的？
3. split的源码，split("a|b|c");得出多少个数组
4. servlet生命周期，是否单例，为什么是单例。
5. 对静态变量的理解