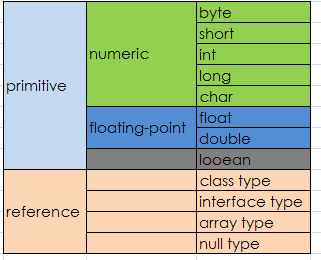
## 数据类型



### 容量单位

计算机中表储存容量大小的单位 bit、B（byte）、KB、MB、GB、TB（类似质量单位g、kg）在计算机术语中，bit的中文含义是位。在计算机中的二进制数系统中，位，简记为b,也称为比特，每个0或1就是一个位(bit)。计算机中的CPU位数指的是CPU一次能处理的最大位数。

内存单位的换算：

|  |
| --- |
| * 1B（byte，字节）= 8 bit * 1KB（Kilobyte，千字节）=1024B= 10^3 B * 1MB（Megabyte，兆字节，百万字节，简称“兆”）=1024KB= 10^6 B * 1GB（Gigabyte，吉字节，十亿字节，又称“千兆”）=1024MB= 10^9 B * 1TB（Terabyte，万亿字节，太字节）=1024GB= 10^12 B |

### 基本数据类型

变量就是申请内存来存储值。也就是说，当创建变量的时候，需要在内存中申请空间。内存管理系统根据变量的类型为变量分配存储空间，分配的空间只能用来储存该类型数据。为了合理利用内存空间，所以用到数据类型。如int a=2;不管a的是多少，都会向内存申请32bit的内存，如果a的是定值且很小在byte范围内，那可以用byte类型占8位内存节省空间。

Java语言提供了八种基本类型。六种数字类型（四个整数型，两个浮点型），一种字符类型，还有一种布尔型。Java代码中数字默认都是int类型，十进制,int类型/操作时得到的是只舍不进的整数。

|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 类型 | byte | short | int | long | float | double | boolean | char |
| 位数 | 8bit | 16bit | 32bit | 64bit | 32bit | 64bit | 1bit | 16bit |
| 默认值 | 0 | 0 | 0 | 0L | 0.0f | 0.0d | false | null |

byte

字节，计算机文件在计算机中都是以字节形式存储，其他类型长度都是字节长度的倍数

整数型以byte为例，数据类型是8位、有符号的，以二进制补码表示的整数

|  |
| --- |
| 原码:用二进制表示的原始编码 反码:除了符号位外，其它位取反 补码:正数的补码等于原码，负数的补码等于反码加1 |

示例：byte a=5,b=-5的二进制表示的整数，根据以上规则得到5的原码：00000101，反码：11111010，补码：11111011，整数是原码那么5的二进制表示为00000101，负数为补码二进制表示为11111011。

int a=5的二进制表示00000000000000000000000000000101，int a=-5的二进制表示1111111111111111111111111111111011

byte的最大值为01111111即2^7-1=127最小值为最大值的负数再减1即-2^7-1=-128.计算机中采用补码表示负数11111111：-1,10000000：-128，正数第一位是0，负数为1。

注意：

byte a=127;  
byte b=4;  
byte c= (byte) (a+b);  
System.out.println(c);

结果：-125。代码中2个正整数相加变为负数就是因为二进制储存位数原因

long类型java中十进制最大数2^32-1，十进制为19位长度，而在js中为2^53十进制为16位长度,所以后台长度过长，那么前台会精度丢失，解决方法转为String类型，手动转换和第三方包注解转如jackson.databind。

浮点型与二进制转换方法：整数除以2，商继续除以2，得到0为止，将余数逆序排列，小数乘以2，取整，小数部分继续乘以2，取整，得到小数部分0为止，将整数顺序排列。

示例：十进制22.8125等于二进制10110.1101的算法

22 / 2 11 余 0

11 / 2 5 余 1

5 / 2 2 余 1

2 / 2 1 余 0

1 / 2 0 余 1

0结束

所以22的二进制是10110

小数乘以2，取整，小数部分继续乘以2，取整，得到小数部分0为止，将整数顺序排列

0.8125x2=1.625 取整1，小数部分是0.625

0.625x2=1.25 取整1，小数部分是0.25

0.25x2=0.5 取整0，小数部分是0.5

0.5x2=1.0 取整1，小数部分是0，结束

所以0.8125的二进制是0.1101

浮点型表示一个小数的时候存在精度不准确的问题，由上可知整数部分除以2结果一定为0余0或1，即整数部分永远可以用二进制精确表示，但小数部分就不一定了

如0.9用二进制表示

0.9\*2=1.8 取整数部分 1

0.8(1.8的小数部分)\*2=1.6 取整数部分 1

0.6\*2=1.2 取整数部分 1

0.2\*2=0.4 取整数部分 0

0.4\*2=0.8 取整数部分 0

0.8\*2=1.6 取整数部分 1

0.6\*2=1.2 取整数部分 0

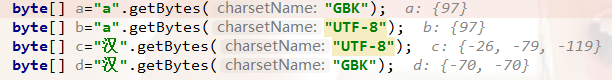
......... 0.9二进制表示为(从上往下): 1100100100100......

上面的计算过程循环了，也就是说\*2永远不可能消灭小数部分，这样算法将无限下去。很显然，小数的二进制表示有时是不可能精确的,实际应用中对小数精度要求比较的如金额要用BigDecimal类。

boolean数据类型表示一位的信息，只有两个取值：true 和 false，默认值是 false，二进制储存ture为1，false为0。

char类型是一个单一的 16 位 Unicode 字符，数据类型可以储存任何字符，如char a=’a’占8位，二进制一般为0000000010110001，char a=‘汉’占16位,而int a=‘汉’则会占32位，二进制一般为0100101010101110，即一个汉字=2个英文字母=2字节，一个字符储存什么样的二进制则需要一套编码机制，每一个字符都对应一个唯一的二进制数，乱码的原因就是字符编码和解码用的不是同一套编码，假如‘汉’用gbk编码储存为010，UTF-8编码为001，编码时用gbk，解码时用utf导致乱码。有的编码不是包含所有汉字，所以有的特殊生僻字没有对应的二进制导致乱码。

一个英文字符占一个字节，中文字符占2-4个字节，如下图知，一个英文字符不管用什么编码转为成字节长度都为1且值一样，而汉字字符则为3或2且值不同。





reference类型在32位系统上每个占用4bytes, 在64位系统上每个占用8bytes。

基本类型的包装类

虽然 Java 语言是典型的面向对象编程语言，但其中的八种基本数据类型并不支持面向对象编程，基本类型的数据不具备“对象”的特性——不携带属性、没有方法可调用。 沿用它们只是为了迎合人类根深蒂固的习惯，并的确能简单、有效地进行常规数据处理。这种借助于非面向对象技术的做法有时也会带来不便，比如引用类型数据均继承了 Object 类的特性，要转换为 String 类型（经常有这种需要）时只要简单调用 Object 类中定义的toString()即可，而基本数据类型转换为 String 类型则要麻烦得多。为解决此类问题 ，Java为每种基本数据类型分别设计了对应的类，称之为包装类(Wrapper Classes)，也有教材称为外覆类或数据类型类。

包装类是引用类型所以默认值为null，如int 默认为0，Integer 默认null，Integer可以区分出未赋值和值为0的区别，int则无法表达出未赋值的情况

比如在查询条件时，

<if test="taskStatus!=null">

AND tsm.task\_status = #{taskStatus,jdbcType=INTEGER}

</if>

int类型不可以，Integer则可以，包装类型里包含了基本类型的各种属性，如Integer.SIZE、Integer.MAX\_VALUE、Integer.MIN\_VALUE

|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 基本数据类型 | byte | short | int | long | char | float | double | boolean |
| 对应的包装类 | Byte | Short | Integer | Long | Character | Float | Double | Boolean |

基本类型和对应的包装类可以相互装换：装箱、拆箱

由基本类型向对应的包装类转换称为装箱，例如把 int 包装成 Integer 类的对象；

包装类向对应的基本类型转换称为拆箱，例如把 Integer 类的对象重新简化为 int。

int m = 500;

Integer obj = new Integer(m); // 手动装箱

int n = obj.intValue(); // 手动拆箱

Java 1.5 之后可以自动拆箱装箱，也就是在进行基本数据类型和对应的包装类转换时，系统将自动进行，这将大大方便程序员的代码书写

int m = 500;

Integer obj = m; // 自动装箱

int n = obj; // 自动拆箱

自动装箱和拆箱的原理

自动装箱时编译器调用valueOf将原始类型值转换成对象，同时自动拆箱时，编译器通过调用类似intValue(),doubleValue()这类的方法将对象转换成原始类型值。明白自动装箱和拆箱的原理后，我们带着上面的疑问进行分析下Integer的自动装箱的实现源码。如下：

public static Integer valueOf(int i) {

  //判断i是否在-128和127之间，如果不在此范围，则从IntegerCache中获取包装类的实例。否则new一个新实例

  if (i >= IntegerCache.low && i <= IntegerCache.high)

    return IntegerCache.cache[i + (-IntegerCache.low)];

  return new Integer(i);

}

//使用享元模式，来减少对象的创建（亨元设计模式大家有必要了解一下，我认为是最简单的设计模式，也许大家经常在项目中使用，不知道他的名字而已

private static class IntegerCache {

  static final int low = -128;

  static final int high;

  static final Integer cache[];

  //静态方法，类加载的时候进行初始化cache[],静态变量存放在常量池中

  static {

    // high value may be configured by property

    int h = 127;

    String integerCacheHighPropValue =

      sun.misc.VM.getSavedProperty("java.lang.Integer.IntegerCache.high");

    if (integerCacheHighPropValue != null) {

      try {

        int i = parseInt(integerCacheHighPropValue);

        i = Math.max(i, 127);

        // Maximum array size is Integer.MAX\_VALUE

        h = Math.min(i, Integer.MAX\_VALUE - (-low) -1);

      } catch( NumberFormatException nfe) {

        // If the property cannot be parsed into an int, ignore it.

      }

    }

    high = h;

    cache = new Integer[(high - low) + 1];

    int j = low;

    for(int k = 0; k < cache.length; k++)

      cache[k] = new Integer(j++);

    // range [-128, 127] must be interned (JLS7 5.1.7)

    assert IntegerCache.high >= 127;

  }

  private IntegerCache() {}

}

Integer i1 = 40; 自动装箱，相当于调用了Integer.valueOf(40);方法。首先判断i值是否在-128和127之间，如果在-128和127之间则直接从IntegerCache.cache缓存中获取指定数字的包装类；不存在则new出一个新的包装类。IntegerCache内部实现了一个Integer的静态常量数组，在类加载的时候，执行static静态块进行初始化-128到127之间的Integer对象，存放到cache数组中。cache属于常量，存放在java的方法区中。

### 引用类型

Java中除了8种基本数据类型，其他都是引用类型。数组、String、所有的类变量都是引用类型，默认值都为null，理论上Object类是所有引用类型的父类，即直接或间接的继承java.lang.Object类。由于所有的类包括数组都继承了Object类，因此省略了extends Object关键字，引用类型都具有Object类的的方法，其中String类是一种很特殊的引用类，具有很多基本类型的特性，对Object类的学习参考源码及注释。

## 异常

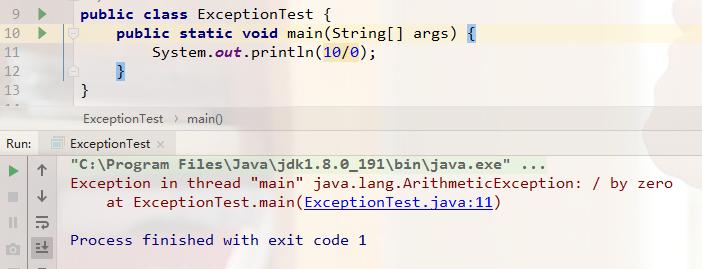
### 概述

异常机制已经成为判断一门编程语言是否成熟的标准，异常机制可以使程序中异常处理代码和正常业务代码分离，保证程序代码更加优雅，并提高程序健壮性，在开发工作中遇到报错问题时，异常信息则是解决问题的入口。

### 异常语法

Java异常处理机制：在程序运行出现意外时，系统会生成一个Exception对象，来通知程序，从而实现将“业务功能实现代码”和“错误处理代码”分离，提供更好的可读性。如果执行try块里的业务逻辑代码时出现异常，jvm会自动生成一个异常对象，该异常对象被提交给运行环境，这个过程被称为抛出（throw）异常。Java环境收到异常对象时，会寻找合适的catch块，如果异常被捕捉到了，那么会从发生异常的地方直接跳到捕捉到异常的catch块执行代码，如果找不到，则会传给jvm处理，jvm处理异常默认方案是向控制台打印异常信息，然后退出程序即执行system.exit(1),java运行环境就会终止，java程序将退出。

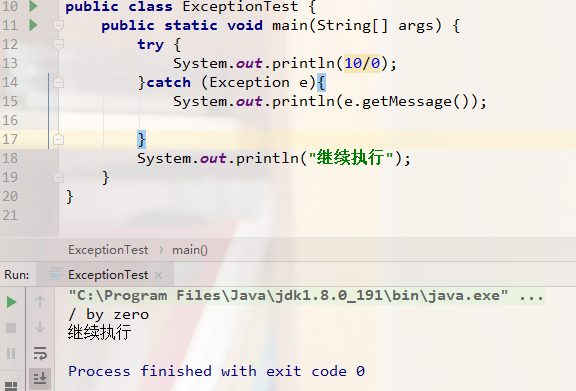
1、异常未被捕获代码示例



当程序执行10/0时发生运行时异常，jvm根据异常情况创建ArithmeticException（“/by zero”）并抛出（throw），运行环境守到异常对象寻找适合的catch块，由于代码没有捕获最终交给jvm处理，jvm默认显示异常信息是通过调用ThreadGroup.uncaughtException(..)方法如下，当执行完后再执行System.exit(1)显示Process finished with exit code 1程序终止。

|  |
| --- |
| public void uncaughtException(Thread t, Throwable e) {  if (parent != null) {  parent.uncaughtException(t, e);  } else {  Thread.UncaughtExceptionHandler ueh =  Thread.getDefaultUncaughtExceptionHandler();  if (ueh != null) {  ueh.uncaughtException(t, e);  } else if (!(e instanceof ThreadDeath)) {  System.err.print("Exception in thread \""  + t.getName() + "\" ");  e.printStackTrace(System.err);  }  } } |

2、异常被捕获代码示例



当程序执行10/0时发生运行时异常，jvm根据异常情况创建ArithmeticException（“/by zero”）并抛出（throw），运行环境守到异常对象寻找适合的catch块，由于代码中有捕获的catch块所以被捕获到进入catch块执行后继续执行以下代码，当所有代码执行完后再执行System.exit(1)显示Process finished with exit code 1程序终止。

Java异常机制主要依赖于try、catch、finally、throw、throws五个关键字。

* try：捕获可能引发异常的代码,只有代码发生异常throw出来才能捕获到，如果被上一级try捕获到了并没有再throw出来，那么下级是捕获不到的，捕获最后一层。
* catch：后面对应异常类型和一个代码块，用于表明该catch块用于处理这种类型的代码块，可以有多个catch块，先捕获小的异常，再捕获大的异常。
* finally：主要用于回收在try块里打开的物力资源（如数据库连接、网络连接和磁盘文件），异常机制总是保证finally块总是被执行，除非在try块或者catch块中调用了退出虚拟机的方法即System.exit(1)。只有finally块，执行完成之后，才会回来执行try或者catch块中的return或者throw语句，如果finally中使用了return或者throw等终止方法的语句，则就不会跳回执行，直接停止，所以一般情况下，不要再finally块中使用renturn或throw等导致方法终止的语句，因为一旦使用，将会导致try块、catch块中的return、throw语句失效。
* throw：用于抛出一个实际的异常，可以单独作为语句使用，抛出一个具体的异常对象给上级捕获，如throw new NullPointerException()运行时异常，则方法中不用加throws，throw new FileNotFoundException()，则方法中必须加上throws,因为检查性异常必须处理，要么当前方法处理要么被调用的上级处理或上级的上级处理，如果一直没有被捕获最终由jvm处理则会按照jvm默认处理方式处理。
* throws：检查性异常必须被捕获，如果在当前方法中不想捕获处理可以在方法签名中使用throws，用于声明该方法可能抛出的异常，如果某段代码调用了一个带throws声明的方法，该方法声明抛出了checked异常，这表明该方法希望它的调用者来处理该异常。那么这段代码要么放在try块中显示捕获该异常，要么这段代码处于另一个带throws声明抛出的方法中，如检查性异常io异常，使用的方法后有throws 。

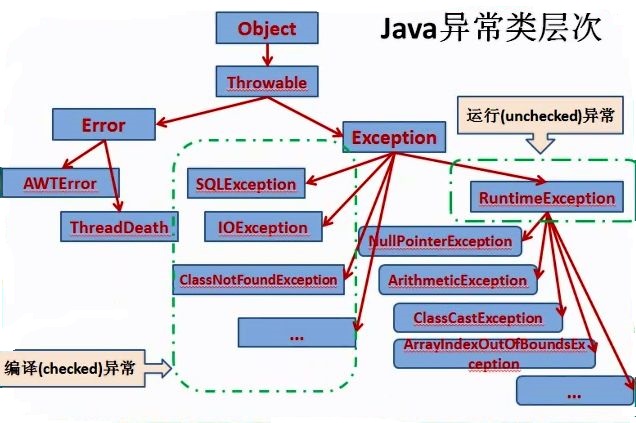
### 异常对象

java是面向对象编程，面向对象思想，所以在java同样会将错误、异常抽象封装为不同的类，从而更方便在程序中获取处理异常的信息。java程序发生异常情况一般有错误，检查性异常，非检查性异常。

* 检查性异常：编译时异常，最具代表的检查性异常是用户错误或问题引起的异常，这是程序员无法预见的。例如要打开一个不存在文件时，一个异常就发生了，这些异常在编译时不能被简单地忽略，必须被捕获。
* 非检查异常：运行时异常，可能被程序员避免的异常。与检查性异常相反，运行时异常可以在编译时被忽略。
* 错误：不是异常，而是脱离程序员控制的问题。错误在代码中通常被忽略。例如，当栈溢出时，一个错误就发生了，它们在编译也检查不到的。



java将上述分类分别封装为了Throwable、 Error、Exception、RuntimeException来表示不同的异常及它们之间的关系。



* Throwable:java语言中所有错误(Error)或异常(Exception)的超类
* Error:一般是指java虚拟机相关的问题，如系统崩溃、虚拟机出错误、动态链接失败等，这种错误无法恢复或不可能捕获，将导致应用程序中断，通常应用程序无法处理这些错误，因此应用程序不应该捕获Error对象，也无须在其throws子句中声明该方法抛出任何Error或其子类。
* Exception：所有异常类的父类
* RuntimeException:运行时异常类的父类

下面的列表是 Throwable 类的主要方法:

|  |  |
| --- | --- |
| 1 | public String getMessage() 只会获得具体的异常名称. 如/by zero,只显示名称。这个消息在Throwable 类的构造函数中初始化了。 |
| 2 | public Throwable getCause() 返回一个Throwable 对象代表异常原因。 |
| 4 | public void printStackTrace() 会打出详细异常,异常名称,出错位置,便于调试用。如下，具体执行原理查看源码  Exception in thread "main" java.lang.ArithmeticException: / by zero  at ExceptionTest.main(ExceptionTest.java:13) |

自定义异常类

自定义检查性异常直接继承Exception基类，如果希望自定义运行时异常则应该继承RuntimeException基类。

### 事务中的应用

原理：通过异常机制，当发生异常时抛出异常被捕获然后回滚

|  |
| --- |
| Connection conn = getConnection();  try {  conn.commit();1  } catch (Exception e) {  conn.rollback();2  }finally {  conn .close() ;3  } |

原理是1处操作发生异常，程序抛出一个异常被2处扑捉到进行回滚。如果在1处再套一层try catch如下,这样2处则不会执行

|  |
| --- |
| Connection conn = getConnection();  try {  try{  conn.commit();1  }catch(Exception e){  e.printStackTrace();  }  } catch (Exception e) {  conn.rollback();2  }finally {  conn .close() ;3  } |

在实际开发中一般用spring声明式事务，1处的操作就是service层代码，为了事务的正常生效，当有try catch后捕获了异常，事务不会回滚，如果不得不在service层写try catch 并且当发生时希望回滚则需要catch后 throw new RuntimeException() 让事务回滚,如下

|  |
| --- |
| Connection conn = getConnection();  try {  try{  conn.commit();1  }catch(Exception e){  throw new  [Global](C:/Program%20Files/Dict/8.5.1.0/resultui/html/index.html" \l "/javascript:;)Exception(“统一异常处理”)  }  } catch (Exception e) {  conn.rollback();2  }finally {  conn .close() ;3  } |

spring通过异常进行事务回滚的机制：

1、spring 的默认事务机制，当出现unchecked异常时候回滚，checked异常的时候不会回滚,因为一般检查性异常都已经处理了并且不会再抛出异常让上级捕获；

2、我们有时为了打印日志，会在service层抓住Exception并打印日志，这时我们的所有异常都会被认为成checked异常。

3、为了事务的正常生效：当有try catch后捕获了异常，事务不会回滚，如果不得不在service层写try catch 需要catch后 throw new RuntimeException 让事务回滚；

例子：

try {

} catch (Exception e) {

e.printStackTrace();

logger.error("发生异常");

throw new RuntimeException();

}

如何保证在有返回值的方法中进行异常的捕获？

方法1：因为要保证当前方法具有返回值，在前端界面展示操作状态。所以我们service层在添加了try catch日志打印后，抛出的runtime类异常需要在controller层进行捕获，捕获之后，在catch中编写操作失败后返回值的信息。

方法2：不依赖于spring的异常捕获机制进行事务回滚，通过手动的session.rollback进行异常捕获后回滚事务也可。

|  |
| --- |
| ------------update by qiyun on 20191206 |

## 修饰符

修饰符用来定义类、方法或者变量，通常放在语句的最前端

### 访问控制修饰符

| **修饰符** | **当前类** | **同包** | **子类** | **其他包** |
| --- | --- | --- | --- | --- |
| public | √ | √ | √ | √ |
| protected | √ | √ | √ | × |
| default | √ | √ | × | × |
| private | √ | × | × | × |

这个范围指的是实例所在的位置，如下person在Main中，所有person的private不可调用，public在哪都可以。

|  |
| --- |
| public class Main {  public static void main(String[] args) {  Person person = new Person();  person.say1();  person.say2();//编译无法通过  int age1 = person.age1;  int age2 = person.age2;//编译无法通过  }  }  class Person {  private String name;  public int age1;  private int age2;  public void say1() {  System.out.println("say1");  }  private void say2() {  System.out.println("say2");  }  public String getName() {  return name;  }  public void setName(String name) {  this.name = name;  }  } |

Jdk中有的类的构造器是用private修饰的，目的就是不让用户在其他地方实例化，如Math类，这种类要么是其中的方法都是静态的直接调用不需要实例化，要么就是为了保证单例， 通过类中的单例模式获取实例。

访问控制和继承

1. 父类中声明为public的方法在子类中也必须为public。
2. 父类中声明为protected的方法在子类中要么声明为protected，要么声明为public。不能声明为private。很少用protectedObject类中对clone方法的声明即用到了protected访问修饰符，这是因为Object类的clone方法只能实现浅克隆，而并不能实现常使用的深克隆，这就要求子类在需要克隆对象时尽量重写clone方法，此时即声明为protected的，以保证在需要克隆对象时，必须要求待克隆对象所在的类实现Cloneable接口并重写clone方法。（浅：引用，深：新的对象）
3. 父类中声明为private的方法，不能够被继承。

### 非访问修饰符

为了实现一些其他的功能，Java也提供了许多非访问修饰符

#### static

static方法就是没有this的方法。在static方法内部不能调用非静态方法，反过来是可以的。而且可以在没有创建任何对象的前提下，仅仅通过类本身来调用static方法。这实际上正是static方法的主要用途。这段话虽然只是说明了static方法的特殊之处，但是可以看出static关键字的基本作用，简而言之，一句话来描述就是：方便在没有创建对象的情况下来进行调用（方法/变量）。很显然，被static关键字修饰的方法或者变量不需要依赖于对象来进行访问，只要类被加载了，就可以通过类名去进行访问。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。

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。而对于非静态成员方法，它访问静态成员方法/变量显然是毫无限制的。因此，如果说想在不创建对象的情况下调用某个方法，就可以将这个方法设置为static。我们最常见的static方法就是main方法，至于为什么main方法必须是static的，现在就很清楚了。因为程序在执行main方法的时候没有创建任何对象，因此只有通过类名来访问。

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是不允许用来修饰局部变量。不要问为什么，这是Java语法的规定。static成员变量的初始化顺序按照定义的顺序进行初始化。

static代码块

static关键字还有一个比较关键的作用就是用来形成静态代码块以优化程序性能。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性能，是因为它的特性:只会在类加载的时候执行一次，如spring、mybatis中启动加载的时候很多只加载一次的初始化都用static块实现。

static加载顺序问题

public class Test {  
 Person person = new Person("Test");  
 static{  
 System.out.println("test static");  
 }  
 public Test() {  
 System.out.println("test constructor");  
 }  
 public static void main(String[] args) {  
 new MyClass();  
 }  
}  
class Person{  
 static{  
 System.out.println("person static");  
 }  
 public Person(String str) {  
 System.out.println("person "+str);  
 }  
}  
class MyClass extends Test {  
 Person person = new Person("MyClass");  
 static{  
 System.out.println("myclass static");  
 }  
 public MyClass() {  
 System.out.println("myclass constructor");  
 }  
}

分析：执行Test类中main方法，首先加载Test类，因此会执行Test类中的static块。接着执行new MyClass()，而MyClass类还没有被加载，因此需要加载MyClass类。在加载MyClass类的时候，发现MyClass类继承自Test类，但是由于Test类已经被加载了，所以只需要加载MyClass类，那么就会执行MyClass类的中的static块。在加载完之后，就通过构造器来生成对象。而在生成对象的时候，必须先初始化父类的成员变量，因此会执行Test中的Person person = new Person()，而Person类还没有被加载过，因此会先加载Person类并执行Person类中的static块，接着执行父类的构造器，完成了父类的初始化，然后就来初始化自身了，因此会接着执行MyClass中的Person person = new Person()，最后执行MyClass的构造器。

#### final

A、修饰类(class)。   
     1、该类不能被继承。   
     2、类中的方法不会被覆盖，因此默认都是final的。   
     3、用途：设计类时，如果该类不需要有子类，不必要被扩展，类的实现细节不允许被改变，那么就设计成final类

B、修饰方法(method)   
     1、该方法可以被继承，但是不能被覆盖。   
     2、用途：一个类不允许子类覆盖该方法，则用final来修饰   
     3、好处：可以防止继承它的子类修改该方法的意义和实现；更为高效，编译器在遇到调用fianal方法转入内嵌机制，提高了执行效率。   
     4、注意：父类中的private成员方法不能被子类覆盖，因此，private方法默认是final型的(可以查看编译后的class文件)

C、修饰变量(variable)   
 1、用final修饰后变为常量。包括静态变量、实例变量和局部变量这三种。   
 2、特点：可以先声明，不给初值，这种叫做final空白。但是使用前必须被初始化,可以在域的定义处或每个构造器中用表达式为final赋值， 一旦被赋值，将不能再被改变。

3、注意：基本数据类型是不可以被修改了,包括String，对于其他refer 型的变量，可以修改这种对像里面的内容，也就是说final引用对象 的引用不能改变，但是里面的值可以改变，.如Map,List等,即使你用final修饰了,依然可以put和add

D、修饰参数(arguments)   
     1、用final修饰参数时，可以读取该参数，但是不能对其作出修改。

#### abstract

1、抽象类

抽象类不能用来实例化对象，声明抽象类的唯一目的是为了将来对该类进行扩充。抽象类是存在抽象方法的，如果能让抽象类创建对象的话，那么使用抽象的对象调用抽象方法是没有任何意义的，一个类不能同时被abstract和final修饰。如果一个类包含抽象方法，那么该类一定要声明为抽象类，否则将出现编译错误。抽象类可以包含抽象方法和非抽象方法。

2、抽象方法

抽象方法是一种没有任何实现的方法，该方法的的具体实现由子类提供。抽象方法不能被声明成final和static。任何继承抽象类的子类必须实现父类的所有抽象方法，除非该子类也是抽象类。如果一个类包含若干个抽象方法，那么该类必须声明为抽象类。抽象类可以不包含抽象方法。

#### synchronized

出现的目的：解决多线程安全问题，保证①可见性。②原子性。

什么是多线程安全问题：如当多个线程同时共享，同一个全局变量或静态变量，做写的操作时，可能会发生数据冲突问题，这时就发生了线程安全问题。线程安全问题体现在多线程编程各个方面，关键是在编程的时候保证多线程的三大特性可见性、原子性、有序性。synchronized则是解决线程安全问题的一种方式。

同步，Java语言的关键字，可用来给对象和方法或者代码块加锁。当两个并发线程访问同一个对象object中的这个加锁同步代码块时，一个时间内只能有一个线程得到执行。

1. 普通同步方法（实例方法），锁是当前实例对象 ，进入同步代码前要获得当前实例的锁this锁
2. 静态同步方法，锁是当前类的class对象 ，进入同步代码前要获得当前类对象的锁
3. 同步方法块，锁是括号里面的对象，对给定对象加锁，进入同步代码库前要获得给定对象的锁。

|  |
| --- |
| //共享对象属性，当多个线程同时共享，同一个全局变量或静态变量，做写的操作时可能会发生数据冲突问题，如果是不同实例则没有线程安全问题 public int a = 0; // 当前实例锁即this锁,如果不同线程拥有的实例不同，那么互相不影响，不会有线程安全问题 public synchronized void syn1() {  for (int i = 0; i < 1000; i++) {  System.out.println(a++);  }  //this.wait()当前线程释放当前对象的锁 } //当前class锁即SynchronizedTest.class锁，因为class唯一一个，所以多线程相互等待执行 public static synchronized void syn2() {  //XXX.class.wait()显示的释放类锁  } //给定对象的锁，更具多线程中传入的对象而定，如果是同一个实例则获取锁执行 public void syn3(Object obj) {  synchronized (obj) {  for (int i = 0; i < 1000; i++) {  System.out.println(a++);  }  //obj.wait()当前线程释放传入的obj对象的锁  } } |

this锁和obj锁都是对象锁，释放直接调用其wait方法(Object方法即所有类都有的方法)即可，类锁释放xxx.class.wait(),xxx.class也是一种对象

#### volatile

volatile修饰的成员变量在每次被线程访问时，都强迫从共享内存中重读该成员变量的值。而且，当成员变量发生变化时，强迫线程将变化值回写到共享内存。这样在任何时刻，两个不同的线程总是看到某个成员变量的同一个值。即保证①可见性。



其实在main线程中修改game属性的时候，在game线程中其实并不知道它的属性被修改了，所以导致一直停不下，多线程Java内存模型(简称JMM)



从上图来看，线程A与线程B之间如要通信的话，必须要经历下面2个步骤：

1. 首先，线程A把本地内存A中更新过的共享变量刷新到主内存中去。

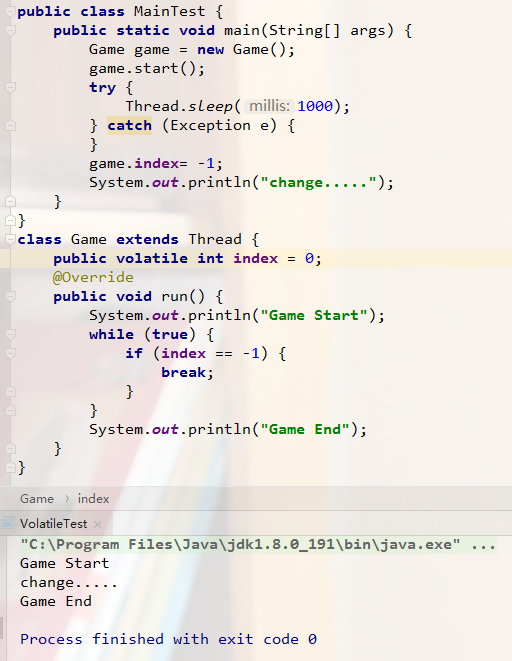
2. 然后，线程B到主内存中去读取线程A之前已更新过的共享变量。

下面通过示意图来说明这两个步骤：   


从整体来看，这两个步骤实质上是线程A在向线程B发送消息，而且这个通信过程必须要经过主内存。JMM通过控制主内存与每个线程的本地内存之间的交互，来为java程序员提供内存可见性保证。当多个线程同时访问一个数据的时候，可能本地内存没有及时刷新到主内存，所以就会发生线程安全问题。

具体解决方法

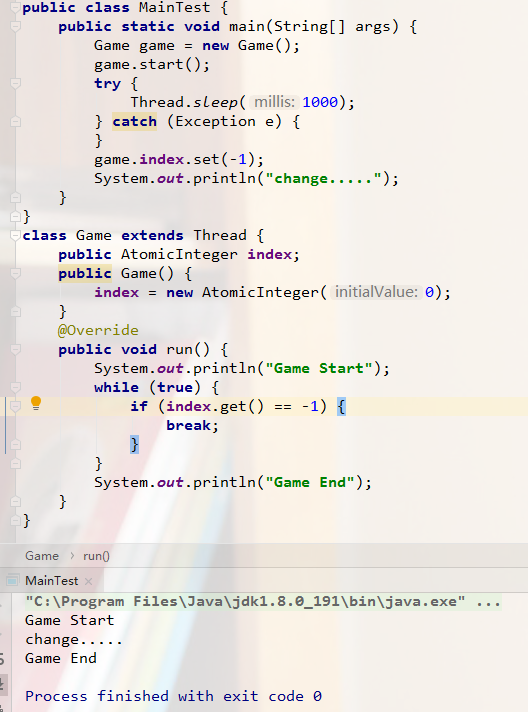
1. 使用volatile关键字



1. 使用synchronized



1. 使用原子类



volatile（不稳定的）和synchronized（同步）的区别

1. volatile本质是在告诉jvm当前变量在寄存器（工作内存）中的值是不确定的，需要从主存中读取，synchronized则是锁定当前变量，只有当前线程可以访问该变量，其他线程被阻塞住。
2. volatile仅能使用在变量级别，synchronized则可以使用在变量、方法、和类级别的
3. volatile仅能实现变量的修改可见性，不能保证原子性，而synchronized则可以保证变量的修改可见性和原子性
4. volatile不会造成线程的阻塞，synchronized可能会造成线程的阻塞。
5. volatile标记的变量不会被编译器优化，synchronized标记的变量可以被编译器优化

volatile使用场景

volatile没有保证原子性，所以没法保证线程安全，但是如果多个变量之间或者某个变量的当前值与修改后值之间没有约束，出于简易性或可伸缩性的考虑，可能倾向于使用 volatile 变量而不是锁。只能在有限的一些情形下使用 volatile 变量替代锁。要使 volatile 变量提供理想的线程安全，必须同时满足下面两个条件：

* 对变量的写操作不依赖于当前值。
* 该变量没有包含在具有其他变量的不变式中。

第一个条件的限制使 volatile 变量不能用作线程安全计数器。虽然增量操作（x++）看上去类似一个单独操作，实际上它是一个由（读取－修改－写入）操作序列组成的组合操作，必须以原子方式执行，而 volatile 不能提供必须的原子特性。实现正确的操作需要使x 的值在操作期间保持不变，而 volatile 变量无法实现这点。（然而，如果只从单个线程写入，那么可以忽略第一个条件。）

一般可以使用的场景有：

模式 #1：状态标志

也许实现 volatile 变量的规范使用仅仅是使用一个布尔状态标志，用于指示发生了一个重要的一次性事件，例如完成初始化或请求停机。

模式 #2：一次性安全发布（one-time safe publication）

模式 #3：独立观察（independent observation）

安全使用 volatile 的另一种简单模式是：定期 “发布” 观察结果供程序内部使用。【例如】假设有一种环境传感器能够感觉环境温度。一个后台线程可能会每隔几秒读取一次该传感器，并更新包含当前文档的 volatile 变量。然后，其他线程可以读取这个变量，从而随时能够看到最新的温度值

模式 #4“volatile bean” 模式

volatile bean 模式的基本原理是：很多框架为易变数据的持有者（例如 HttpSession）提供了容器，但是放入这些容器中的对象必须是线程安全的。

#### transient

序列化的对象包含被transient修饰的实例变量时，java虚拟机(JVM)跳过该特定的变量。该修饰符包含在定义变量的语句中，用来预处理类和变量的数据类型。

#### native

java本地接口，如Object类中 public final native Class<?> getClass();public native int hashCode();这些方法是调用外部的方法，一般是c语言写的。

## 枚举

在[数学](https://baike.baidu.com/item/%E6%95%B0%E5%AD%A6/107037" \t "https://baike.baidu.com/item/%E6%9E%9A%E4%B8%BE/_blank)和[计算机科学](https://baike.baidu.com/item/%E8%AE%A1%E7%AE%97%E6%9C%BA%E7%A7%91%E5%AD%A6/9132" \t "https://baike.baidu.com/item/%E6%9E%9A%E4%B8%BE/_blank)理论中，一个集的枚举是列出某些有穷序列集的所有成员的程序，或者是一种特定类型对象的计数。这两种类型经常（但不总是）重叠。是一个被命名的整型常数的集合，枚举在日常生活中很常见，例如表示星期的SUNDAY、MONDAY、TUESDAY、WEDNESDAY、THURSDAY、FRIDAY、SATURDAY就是一个枚举。在枚举出现之前，如果想要表示一组特定的离散值，往往使用一些常量。例如：

public static final int VIDEO = 1;//视频

public static final int AUDIO = 2;//音频

使用枚举则会限定了值域，更安全了，方便管理常量。

### java中的枚举

enum 的全称为 enumeration， 是 JDK 1.5 中引入的新特性。在Java中，被 enum 关键字修饰的类型就是枚举类型。形式如下：enum Color { RED, GREEN, BLUE }如果枚举不添加任何方法，枚举值默认为从0开始的有序数值。以 Color 枚举类型举例，它的枚举常量依次为 RED：0，GREEN：1，BLUE：2。

枚举的好处：可以将常量组织起来，统一进行管理。

枚举的典型应用场景：错误码、状态机等。

### 枚举类型本质

尽管 enum 看起来像是一种新的数据类型，事实上，enum是一种受限制的类，并且具有自己的方法,除了不能继承，基本上可以将 enum 看做一个常规的类。创建enum时，编译器会为你生成一个相关的类，这个类继承自 java.lang.Enum而不是Object,此类是单例私有构造器，只能由jvm生成，程序员直接调用类即可。在enum中，提供了一些基本方法：

values()：返回 enum 实例的数组，而且该数组中的元素严格保持在 enum 中声明时的顺序。

name()：返回实例名。

ordinal()：返回实例声明时的次序，从0开始。

getDeclaringClass()：返回实例所属的 enum 类型。

equals() ：判断是否为同一个对象。可以使用 == 来比较enum实例。

此外，java.lang.Enum实现了Comparable和 Serializable 接口，所以也提供 compareTo() 方法。

**示例**

public enum WeekEnum {

MON, TUE, WED, THU, FRI, SAT, SUN;

}

解释：创建枚举类型要使用 enum 关键字，隐含了所创建的类型都是 java.lang.Enum 类的子类（java.lang.Enum 是一个抽象类）。枚举类型符合通用模式 Class Enum<E extends Enum<E>>，而 E 表示枚举类型的名称。枚举类型的每一个值都将映射到 protected Enum(String name, int ordinal) 构造函数中，在这里，每个值的名称都被转换成一个字符串，并且序数设置表示了此设置被创建的顺序。上面那段代码声明的enum对象实际上调用了7次Enum(String name, int ordinal) 这个构造函数。（name 是常量，上面是MON,TUE等，ordinal是被创建顺序）new Enum<EnumTest>("MON",0);new Enum<EnumTest>("TUE",1);new Enum<EnumTest>("WED",2);。。。。。。

可以在枚举中添加变量如下，成员变量必须在声明实例后面

public enum BondTypeEnum {  
 BONDType2010("2010", "国债"),  
 BONDType2020("2020", "央行票据"),  
 BONDType2030("2030", "金融债"),  
 // 成员变量  
 private String code;  
 private String name;  
 // 构造方法  
 private BondTypeEnum(String code, String name) {  
 this.code = code;  
 this.name = name;  
 }  
 // 普通方法  
 public static String getName(String code) {  
 for (BondTypeEnum c : BondTypeEnum.values()) {  
 if (c.getCode() == code) {  
 return c.name;  
 }  
 }  
 return null;  
 }  
 // get set 方法   
}

## 泛型

### 概述

泛型，即“参数化类型”。一提到参数，最熟悉的就是定义方法时有形参，然后调用此方法时传递实参。那么参数化类型怎么理解呢？顾名思义，就是将类型由原来的具体的类型参数化，类似于方法中的变量参数，此时类型也定义成参数形式（可以称之为类型形参），然后在使用/调用时传入具体的类型（类型实参）。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）。也就是说在泛型使用过程中，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。

### 特性

泛型只在编译阶段有效,在编译之后程序会采取去泛型化的措施,在编译过程中，正确检验泛型结果后，会将泛型的相关信息擦出，并且在对象进入和离开方法的边界处添加类型检查和类型转换的方法。也就是说，泛型信息不会进入到运行时阶段。

public class TestGenericity {  
 public static void main(String[] args) {  
 Map map1=new HashMap();  
 Map<String ,Integer> map2=new HashMap();  
 map1.put("a",new BigDecimal(2));  
 map2=map1;  
 Integer a = map2.get("a");  
 }  
}

编译后为class源码

public class TestGenericity {  
 public TestGenericity() {  
 }  
 public static void main(String[] args) {  
 Map map1 = new HashMap();  
 new HashMap();  
 map1.put("a", new BigDecimal(2));  
 Integer a = (Integer)map1.get("a");  
 }  
}

运行时报错java.lang.ClassCastException: java.math.BigDecimal cannot be cast to java.lang.Integer

注意：当debug单独执行源码Integer a = map2.get("a")时是可以执行的，但是执行整个代码是报错，原因就是debug单独执行的是map2.get("a")，而真正运行时执行的是class文件中的 (Integer)map1.get("a")，所以会报错

### 泛型的使用

泛型类

//此处T可以随便写为任意标识，常见的如T、E、K、V等形式的参数常用于表示泛型//在实例化泛型类时，必须指定T的具体类型

public class Generic<T>{

//key这个成员变量的类型为T,T的类型由外部指定

private T key;

public Generic(T key) { //泛型构造方法形参key的类型也为T，T的类型由外部指定

this.key = key;

}

public T getKey(){ //泛型方法getKey的返回值类型为T，T的类型由外部指定

return key;

}

}

注意：

泛型的类型参数只能是类类型，不能是简单类型。

不能对确切的泛型类型使用instanceof操作。如下面的操作是非法的，编译时会出错。if(ex\_num instanceof Generic<Number>){ }

泛型接口

//定义一个泛型接口

public interface Generator<T> {

public T next();

}

/\*\*

\* 未传入泛型实参时，与泛型类的定义相同，在声明类的时候，需将泛型的声明也一起加到类中

\* 即：class FruitGenerator<T> implements Generator<T>{

\* 如果不声明泛型，如：class FruitGenerator implements Generator<T>，编译器会报错："Unknown class"

\*/

class FruitGenerator<T> implements Generator<T>{

@Override

public T next() {

return null;

}

}

\*\*

\* 传入泛型实参时：

\* 定义一个生产器实现这个接口,虽然我们只创建了一个泛型接口Generator<T>

\* 但是我们可以为T传入无数个实参，形成无数种类型的Generator接口。

\* 在实现类实现泛型接口时，如已将泛型类型传入实参类型，则所有使用泛型的地方都要替换成传入的实参类型

\* 即：Generator<T>，public T next();中的的T都要替换成传入的String类型。

\*/public class FruitGenerator implements Generator<String> {

private String[] fruits = new String[]{"Apple", "Banana", "Pear"};

@Override

public String next() {

Random rand = new Random();

return fruits[rand.nextInt(3)];

}

}

泛型方法

泛型类的定义非常简单，但是泛型方法就比较复杂了。尤其是我们见到的大多数泛型类中的成员方法也都使用了泛型，有的甚至泛型类中也包含着泛型方法，这样在初学者中非常容易将泛型方法理解错了。

泛型类，是在实例化类的时候指明泛型的具体类型；泛型方法，是在调用方法的时候指明泛型的具体类型 。

/\*\*

\* 泛型方法的基本介绍

\* @param tClass 传入的泛型实参

\* @return T 返回值为T类型

\* 说明：

\* 1）public 与 返回值中间<T>非常重要，可以理解为声明此方法为泛型方法。

\* 2）只有声明了<T>的方法才是泛型方法，泛型类中的使用了泛型的成员方法并不是泛型方法。

\* 3）<T>表明该方法将使用泛型类型T，此时才可以在方法中使用泛型类型T。

\* 4）与泛型类的定义一样，此处T可以随便写为任意标识，常见的如T、E、K、V等形式的参数常用于表示泛型。

\*/public <T> T genericMethod(Class<T> tClass)throws InstantiationException ,

IllegalAccessException{

T instance = tClass.newInstance();

return instance;

}

public class GenericTest {

//这个类是个泛型类，在上面已经介绍过

public class Generic<T>{

private T key;

public Generic(T key) {

this.key = key;

}

//我想说的其实是这个，虽然在方法中使用了泛型，但是这并不是一个泛型方法。

//这只是类中一个普通的成员方法，只不过他的返回值是在声明泛型类已经声明过的泛型。

//所以在这个方法中才可以继续使用 T 这个泛型。

public T getKey(){

return key;

}

/\*\*

\* 这个方法显然是有问题的，在编译器会给我们提示这样的错误信息"cannot reslove symbol E"

\* 因为在类的声明中并未声明泛型E，所以在使用E做形参和返回值类型时，编译器会无法识别。

public E setKey(E key){

this.key = keu

}

\*/

}

/\*\*

\* 这才是一个真正的泛型方法。

\* 首先在public与返回值之间的<T>必不可少，这表明这是一个泛型方法，并且声明了一个泛型T

\* 这个T可以出现在这个泛型方法的任意位置.

\* 泛型的数量也可以为任意多个

\* 如：public <T,K> K showKeyName(Generic<T> container){

\* ...

\* }

\*/

public <T> T showKeyName(Generic<T> container){

System.out.println("container key :" + container.getKey());

//当然这个例子举的不太合适，只是为了说明泛型方法的特性。

T test = container.getKey();

return test;

}

//这也不是一个泛型方法，这就是一个普通的方法，只是使用了Generic<Number>这个泛型类做形参而已。

public void showKeyValue1(Generic<Number> obj){

Log.d("泛型测试","key value is " + obj.getKey());

}

//这也不是一个泛型方法，这也是一个普通的方法，只不过使用了泛型通配符?

//同时这也印证了泛型通配符章节所描述的，?是一种类型实参，可以看做为Number等所有类的父类

public void showKeyValue2(Generic<?> obj){

Log.d("泛型测试","key value is " + obj.getKey());

}

/\*\*

\* 这个方法是有问题的，编译器会为我们提示错误信息："UnKnown class 'E' "

\* 虽然我们声明了<T>,也表明了这是一个可以处理泛型的类型的泛型方法。

\* 但是只声明了泛型类型T，并未声明泛型类型E，因此编译器并不知道该如何处理E这个类型。

public <T> T showKeyName(Generic<E> container){

...

}

\*/

/\*\*

\* 这个方法也是有问题的，编译器会为我们提示错误信息："UnKnown class 'T' "

\* 对于编译器来说T这个类型并未项目中声明过，因此编译也不知道该如何编译这个类。

\* 所以这也不是一个正确的泛型方法声明。

public void showkey(T genericObj){

}

\*/

### 泛型通配符

public void showKeyValue1(Generic<Number> obj){

Log.d("泛型测试","key value is " + obj.getKey());

}

 Generic<Integer> gInteger = new Generic<Integer>(123);

Generic<Number> gNumber = new Generic<Number>(456);

showKeyValue(gNumber);

// showKeyValue这个方法编译器会为我们报错：Generic<java.lang.Integer> // cannot be applied to Generic<java.lang.Number>// showKeyValue(gInteger);

public void showKeyValue1(Generic<?> obj){

Log.d("泛型测试","key value is " + obj.getKey());

}

类型通配符一般是使用？代替具体的类型实参，注意了，此处’？’是类型实参，而不是类型形参，可以解决当具体类型不确定的时候，这个通配符就是 ?  ；当操作类型时，不需要使用类型的具体功能时，只使用Object类中的功能。那么可以用 ? 通配符来表未知类型。

### 泛型上下边界

在使用泛型的时候，我们还可以为传入的泛型类型实参进行上下边界的限制，如：类型实参只准传入某种类型的父类或某种类型的子类。

为泛型添加上边界，即传入的类型实参必须是指定类型的子类型

public void showKeyValue1(Generic<? extends Number> obj){

Log.d("泛型测试","key value is " + obj.getKey());

}

Generic<String> generic1 = new Generic<String>("11111");

Generic<Integer> generic2 = new Generic<Integer>(2222);

Generic<Float> generic3 = new Generic<Float>(2.4f);

Generic<Double> generic4 = new Generic<Double>(2.56);

//这一行代码编译器会提示错误，因为String类型并不是Number类型的子类//showKeyValue1(generic1);

showKeyValue1(generic2);

showKeyValue1(generic3);

showKeyValue1(generic4);

## 注解

## 运算符