目录

[assert断言的用法 1](#_Toc528869746)

[迭代器 2](#_Toc528869747)

[方法的不定长参数 2](#_Toc528869748)

[函数指针与遍历访问 2](#_Toc528869749)

[Java语言先编译后解释 2](#_Toc528869750)

[native关键字 3](#_Toc528869751)

[强引用、弱引用、软引用、虚引用 4](#_Toc528869752)

[深克隆与浅克隆 9](#_Toc528869753)

[static块 9](#_Toc528869754)

[static静态类和其使用方法 10](#_Toc528869755)

[无界和限界通配符<?> 11](#_Toc528869756)

[序列化（串行化）Serialization与transient关键字 11](#_Toc528869757)

[引用，指针和参数传递 11](#_Toc528869758)

[字符串格式化输出String Format 12](#_Toc528869759)

java.util工具包  
java.sql数据库包  
java.io输入输出流包  
java.net网络包  
java.lang基础包

## assert断言的用法

assert关键字是在JDK1.4才被添加，用于程序的调试。用法有两种

1.assert expression

expression是逻辑运算表达式，如果expression为true，表示断言成功，程序继续执行。如果为false，会抛出AssertionError，AssertionError继承Error类，Error类与我们经常用到的Exception类都是集成于Throwable类。

2.assert expression：expression2

用法与1相同，添加了expression2，expression2是一个字符串，当AssertionError发生时，会同AssertionError信息后一起打印出。

使用assert 我们不用再调试时使用if或System.out来输出调试信息，当程序正式发布就不用再去删除这些调试信息了。而assert可以通过参数-da（默认）使程序不进行断言判断。

注意：编译器默认不适用assert检测，所以要使用时要添加参数虚拟机启动参数-ea

如 java -ea

在eclipse当中：run configurations->arguments->vm arguments中添加 -ea

## 迭代器

在对向量、列表和序列进行处理（比如，查找某一特定的元素）时，一种典型的操作就是依次访问或修改其中的各个元素。迭代器是软件设计的一种模式，它是对“逐一访问所有元素”这类操作的抽象。迭代器本身也是一个序列S，在任何时候，迭代器中都有唯一的当前元素。迭代器还必须提供某种机制，使得我们可以不断转向S中的下一元素，并将其置为新的当前元素。与位置ADT(Position)相比，迭代器是对前者的扩展与推广。实际上，一个位置本身已经可以被看作是一个迭代器了，只不过它还不能不断更新。总之，所谓迭代器，就是对一组对象之间的位置、直接后继等关系的集成。也因此，它的ADT必须具备一些固定的方法。

## 方法的不定长参数

函数或方法可以使用形如“类型… 参数名”的参数，称为不定长参数，而用户可以以两种方式为这样的形参提供参数。第一种是提供一个数组（长度显然是任意的），第二种是提供一串参数，每个实参以逗号相隔，实参的个数任意但是类型必须一致满足形参类型。

## 函数指针与遍历访问

由于Java中无法使用函数指针，在数据结构的内部提供函数指针来遍历访问该数据结构不可行。只能先将遍历的结果序列输出，可制成迭代器，然后再在函数中调用迭代器或序列来执行目标操作。

## Java语言先编译后解释

1.编译型语言与解释型语言的区别

计算机是不能理解高级语言，当然也就不能直接执行高级语言了。计算机只能直接理解机器语言，所以任何语言，都必须将其翻译成机器语言，计算机才能运行高级语言编写的程序。翻译的方式有两种，一个是编译，一个是解释。两种方式只是翻译的时间不同。

编译型语言：程序在被执行之前，需要一个专门的编译过程，把程序编译成为机器语言的文件，比如exe文件，以后要运行的话就不用重新翻译了，直接使用编译的结果就行了（exe文件），因为翻译只做了一次，运行时不需要翻译，所以编译型语言的程序执行效率高。

解释型语言：程序不需要编译，省了道工序，解释性语言在运行程序的时候才翻译，比如解释性basic语言，专门有一个解释器能够直接执行basic程序，每个语句都是执行的时候才翻译。这样解释性语言每执行一次就要翻译一次，效率比较低。

C/C++ 等都是编译型语言，而Java，C#等都是解释型语言。

虽然Java程序在运行之前也有一个编译过程，但是并不是将程序编译成机器语言，而是将它编译成字节码（可以理解为一个中间语言）。在运行的时候，由JVM将字节码再翻译成机器语言。

2.Java语言要先编译后解释

Java是一种特殊的高级语言，其既具有编译型语言的特征，又具有解释型语言的特征，因为Java语言要经过先编译、后解释才能被执行。

前面介绍了编译型和解释型语言，Java语言比较特殊，由其编写的程序需要先编译，但此编译不会生成特定平台的机器语言文件，而是生成一种和平台无关的字节码文件，也就是\*.class文件，这种字节码文件不是可执行文件，它必须使用特定平台的解释器来解释执行。

先编译成字节码文件，也即\*.class文件：程序员使用Java语言编写的\*.java源文件，需要使用javac编译器编译成字节码文件，也即\*.class文件。注意：该字节码文件与平台无关，可被特定平台的Java解释器执行。

然后根据不同平台的Java解释器，将字节码文件解释成特定平台的机器文件：Java语言中负责解释字节码文件的是Java虚拟机，即JVM（Java Virtual Machine）。不同平台，各自实现了其JVM，JVM向编译器提供相同的编程接口，所以可解释编译器生成的字节码文件，将其解释成特定平台的机器语言文件。

## native关键字

native关键字说明其修饰的方法是一个原生态方法，方法对应的实现不是在当前文件，而是在用其他语言（如C和C++）实现的文件中。Java语言本身不能对操作系统底层进行访问和操作，但是可以通过JNI接口调用其他语言来实现对底层的访问。

JNI是Java本机接口（Java Native Interface），是一个本机编程接口，它是Java软件开发工具箱（Java Software Development Kit，SDK）的一部分。JNI允许Java代码使用以其他语言编写的代码和代码库。Invocation API（JNI的一部分）可以用来将Java虚拟机（JVM）嵌入到本机应用程序中，从而允许程序员从本机代码内部调用Java代码。

不过，对Java外部的调用通常不能移植到其他平台，在applet中还可能引发安全异常。实现本地代码将使您的Java应用程序无法通过100%纯Java测试。但是，如果必须执行本地调用，则要考虑几个准则：

1.将您的所有本地方法都封装到一个类中，这个类调用单个的DLL。对每一种目标操作系统平台，都可以用特定于适当平台的版本的DLL。这样可以将本地代码的影响减少到最小，并有助于将以后所需要的移植问题考虑在内。

2.本地方法尽量简单。尽量使您的本地方法对第三方（包括Microsoft）运行时DLL的依赖减少到最小。使您的本地方法尽量独立，以将加载您的DLL和应用程序所需的开销减少到最小。如果需要运行时DLL，必须随应用程序一起提供。

JNI的书写步骤如下：

a.编写带有native声明的方法的Java类

b.使用javac命令编译编写的Java类

c.使用java -jni \*\*\*\*来生成后缀名为.h的头文件

d.使用其他语言（C、C++）实现本地方法

e.将本地方法编写的文件生成动态链接库

以下是一个在Java中调用本地C程序的简单的例子：

a.编写HelloWorld.java类

class HelloWorld{

public native void hello();

static{  
System.loadLibrary("hello");  
}

public static void main(String[] args){

new HelloWorld().hello();  
}

}

b.编译

javac HelloWorld.java

c.生成.h文件

javah -jni HelloWorld

生成内容如下：

/\* DO NOT EDIT THIS FILE - it is machine generated \*/  
#include <jni.h>  
/\* Header for class HelloWorld \*/

#ifndef \_Included\_HelloWorld  
#define \_Included\_HelloWorld  
#ifdef \_\_cplusplus  
extern "C" {  
#endif  
/\*  
\* Class: HelloWorld  
\* Method: hello  
\* Signature: ()V  
\*/  
JNIEXPORT void JNICALL Java\_HelloWorld\_hello  
(JNIEnv \*, jobject);

#ifdef \_\_cplusplus  
}  
#endif  
#endif

第一个参数是调用JNI方法时使用的JNI Environment指针。第二个参数是指向在此Java代码中实例化的Java对象HelloWorld的一个句柄。其他参数是方法本身的参数

d.c实现

#include <jni.h>  
#include "HelloWorld.h"  
#include <stdio.h>  
JNIEXPORT void JNICALL Java\_HelloWorld\_hello(JNIEnv \*env,jobject obj){  
printf("Hello World!\n");  
return;  
}

其中，第一行是将jni.h文件引入（在%JAVA\_HOME%\include目录下），里边有JNIEnv和jobject的定义。

e.编译c实现

这里以在Windows中为例，需要生成dll文件。在保存HelloWorldImpl.c文件夹下面，使用VC的编译器cl成。  
cl -I%java\_home%\include -I%java\_home%\include\win32 -LD HelloWorldImp.c -Fehello.dll  
注意：生成的dll文件名在选项-Fe后面配置，这里是hello，因为在HelloWorld.java文件中我们loadLibary的时候使用的名字是hello。当然这里修改之后那里也需要修改。另外需要将-I%java\_home%\include -I%java\_home%\include\win32参数加上，因为在第四步里面编写本地方法的时候引入了jni.h文件。  
6) 运行程序  
java HelloWorld就ok了！

## 强引用、弱引用、软引用、虚引用

Java 7中引入的多种引用类型。

1、强引用（StrongReference）  
         强引用是使用最普遍的引用。如果一个对象具有强引用，那垃圾回收器绝不会回收它。如下：

1. Object o=**new** Object();   //  强引用

      当内存空间不足，Java虚拟机宁愿抛出OutOfMemoryError错误，使程序异常终止，也不会靠随意回收具有强引用的对象来解决内存不足的问题。如果不使用时，要通过如下方式来弱化引用，如下：

1. o=**null**;     // 帮助垃圾收集器回收此对象

      显式地设置o为null，或超出对象的生命周期范围，则gc认为该对象不存在引用，这时就可以回收这个对象。具体什么时候收集这要取决于gc的算法。

举例：

1. **public** **void** test(){
2. Object o=**new** Object();
3. // 省略其他操作
4. }

在一个方法的内部有一个强引用，这个引用保存在栈中，而真正的引用内容（Object）保存在堆中。当这个方法运行完成后就会退出方法栈，则引用内容的引用不存在，这个Object会被回收。

但是如果这个o是全局的变量时，就需要在不用这个对象时赋值为null，因为强引用不会被垃圾回收。

强引用在实际中有非常重要的用处，举个ArrayList的实现源代码：

1. **private** **transient** Object[] elementData;
2. **public** **void** clear() {
3. modCount++;
4. // Let gc do its work
5. **for** (**int** i = 0; i < size; i++)
6. elementData[i] = **null**;
7. size = 0;
8. }

在ArrayList类中定义了一个私有的变量elementData数组，在调用方法清空数组时可以看到为每个数组内容赋值为null。不同于elementData=null，强引用仍然存在，避免在后续调用 add()等方法添加元素时进行重新的内存分配。使用如clear()方法中释放内存的方法对数组中存放的引用类型特别适用，这样就可以及时释放内存。

2、软引用（SoftReference）

        如果一个对象只具有软引用，则内存空间足够，垃圾回收器就不会回收它；如果内存空间不足了，就会回收这些对象的内存。只要垃圾回收器没有回收它，该对象就可以被程序使用。软引用可用来实现内存敏感的高速缓存。

1. String str=**new** String("abc");                                     // 强引用
2. SoftReference<String> softRef=**new** SoftReference<String>(str);     // 软引用

      当内存不足时，等价于：

1. If(JVM.内存不足()) {
2. str = **null**;  // 转换为软引用
3. System.gc(); // 垃圾回收器进行回收
4. }

虚引用在实际中有重要的应用，例如浏览器的后退按钮。按后退时，这个后退时显示的网页内容是重新进行请求还是从缓存中取出呢？这就要看具体的实现策略了。

（1）如果一个网页在浏览结束时就进行内容的回收，则按后退查看前面浏览过的页面时，需要重新构建

（2）如果将浏览过的网页存储到内存中会造成内存的大量浪费，甚至会造成内存溢出

这时候就可以使用软引用

1. Browser prev = **new** Browser();               // 获取页面进行浏览
2. SoftReference sr = **new** SoftReference(prev); // 浏览完毕后置为软引用
3. **if**(sr.get()!=**null**){
4. rev = (Browser) sr.get();           // 还没有被回收器回收，直接获取
5. }**else**{
6. prev = **new** Browser();               // 由于内存吃紧，所以对软引用的对象回收了
7. sr = **new** SoftReference(prev);       // 重新构建
8. }

这样就很好的解决了实际的问题。

       软引用可以和一个引用队列（ReferenceQueue）联合使用，如果软引用所引用的对象被垃圾回收器回收，Java虚拟机就会把这个软引用加入到与之关联的引用队列中。

3、弱引用（WeakReference）

      弱引用与软引用的区别在于：只具有弱引用的对象拥有更短暂的生命周期。在垃圾回收器线程扫描它所管辖的内存区域的过程中，一旦发现了只具有弱引用的对象，不管当前内存空间足够与否，都会回收它的内存。不过，由于垃圾回收器是一个优先级很低的线程，因此不一定会很快发现那些只具有弱引用的对象。

1. String str=**new** String("abc");
2. WeakReference<String> abcWeakRef = **new** WeakReference<String>(str);
3. str=**null**;

当垃圾回收器进行扫描回收时等价于：

1. str = **null**;
2. System.gc();

   如果这个对象是偶尔的使用，并且希望在使用时随时就能获取到，但又不想影响此对象的垃圾收集，那么你应该用 Weak Reference 来记住此对象。

   下面的代码会让str再次变为一个强引用：

1. String  abc = abcWeakRef.get();

弱引用可以和一个引用队列（ReferenceQueue）联合使用，如果弱引用所引用的对象被垃圾回收，Java虚拟机就会把这个弱引用加入到与之关联的引用队列中。

当你想引用一个对象，但是这个对象有自己的生命周期，你不想介入这个对象的生命周期，这时候你就是用弱引用。

这个引用不会在对象的垃圾回收判断中产生任何附加的影响。

1. **public** **class** ReferenceTest {
3. **private** **static** ReferenceQueue<VeryBig> rq = **new** ReferenceQueue<VeryBig>();
5. **public** **static** **void** checkQueue() {
6. Reference<? **extends** VeryBig> ref = **null**;
7. **while** ((ref = rq.poll()) != **null**) {
8. **if** (ref != **null**) {
9. System.out.println("In queue: " + ((VeryBigWeakReference) (ref)).id);
10. }
11. }
12. }
14. **public** **static** **void** main(String args[]) {
15. **int** size = 3;
16. LinkedList<WeakReference<VeryBig>> weakList = **new** LinkedList<WeakReference<VeryBig>>();
17. **for** (**int** i = 0; i < size; i++) {
18. weakList.add(**new** VeryBigWeakReference(**new** VeryBig("Weak " + i), rq));
19. System.out.println("Just created weak: " + weakList.getLast());
21. }
23. System.gc();
24. **try** { // 下面休息几分钟，让上面的垃圾回收线程运行完成
25. Thread.currentThread().sleep(6000);
26. } **catch** (InterruptedException e) {
27. e.printStackTrace();
28. }
29. checkQueue();
30. }
31. }
33. **class** VeryBig {
34. **public** String id;
35. // 占用空间,让线程进行回收
36. **byte**[] b = **new** **byte**[2 \* 1024];
38. **public** VeryBig(String id) {
39. **this**.id = id;
40. }
42. **protected** **void** finalize() {
43. System.out.println("Finalizing VeryBig " + id);
44. }
45. }
47. **class** VeryBigWeakReference **extends** WeakReference<VeryBig> {
48. **public** String id;
50. **public** VeryBigWeakReference(VeryBig big, ReferenceQueue<VeryBig> rq) {
51. **super**(big, rq);
52. **this**.id = big.id;
53. }
55. **protected** **void** finalize() {
56. System.out.println("Finalizing VeryBigWeakReference " + id);
57. }
58. }

最后的输出结果为：

1. Just created weak: com.javabase.reference.VeryBigWeakReference@1641c0
2. Just created weak: com.javabase.reference.VeryBigWeakReference@136ab79
3. Just created weak: com.javabase.reference.VeryBigWeakReference@33c1aa
4. Finalizing VeryBig Weak 2
5. Finalizing VeryBig Weak 1
6. Finalizing VeryBig Weak 0
7. In queue: Weak 1
8. In queue: Weak 2
9. In queue: Weak 0

4、虚引用（PhantomReference）

     “虚引用”顾名思义，就是形同虚设，与其他几种引用都不同，虚引用并不会决定对象的生命周期。如果一个对象仅持有虚引用，那么它就和没有任何引用一样，在任何时候都可能被垃圾回收器回收。

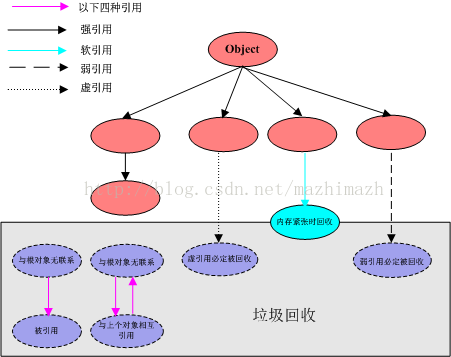
    虚引用主要用来跟踪对象被垃圾回收器回收的活动。虚引用与软引用和弱引用的一个区别在于：虚引用必须和引用队列 （ReferenceQueue）联合使用。当垃圾回收器准备回收一个对象时，如果发现它还有虚引用，就会在回收对象的内存之前，把这个虚引用加入到与之 关联的引用队列中。

5、总结

 Java4种引用的级别由高到低依次为：

**强引用  >  软引用  >  弱引用  >  虚引用**

通过图来看一下他们之间在垃圾回收时的区别：



当垃圾回收器回收时，某些对象会被回收，某些不会被回收。垃圾回收器会从根对象Object来标记存活的对象，然后将某些不可达的对象和一些引用的对象进行回收

通过表格来说明一下，如下：

|  |  |  |  |
| --- | --- | --- | --- |
| **引用类型** | **被垃圾回收时间** | **用途** | **生存时间** |
| 强引用 | 从来不会 | 对象的一般状态 | JVM停止运行时终止 |
| 软引用 | 在内存不足时 | 对象缓存 | 内存不足时终止 |
| 弱引用 | 在垃圾回收时 | 对象缓存 | gc运行后终止 |
| 虚引用 | Unknown | Unknown | Unknown |

## 深克隆与浅克隆

在使用克隆时，我们需要知道使用的目的：就是为了快速构造一个和已有对象相同的副本。如果克隆对象，一般需要先创建一个对象，然后将原对象中的数据导入到新创建的对象中去,而不用根据已有对象进行手动赋值操作。

protected native Object clone() throws CloneNotSupportedException;

有如下3点需要提示：

1）它是一个protected修饰的native方法，因此它的实现是取决于本地代码。native方法的效率一般来说都是远高于java中的非native方法。

2）Object中的clone方法是protected的，所以要使用clone就必须继承Object类（默认）。并且为了可以使其它类调用该方法，覆写克隆方法时必须将其作用域设置为public.

3）克隆方法返回的是一个Object对象，所以必须要经过强制类型转换。

通常克隆对象都是通过调用super.clone()方法来获取克隆对象的，所以任何克隆的过程最终都将到达java.lang.Object 的clone()方法。但是在覆写clone()方法时，这个类需要继承Clonable接口，这个接口中没有定义方法，他类似于RandomAccess这些接口类，只做为一种标识存在。如果 clone 类没有实现 Cloneable 接口，并调用了 Object 的 clone() 方法，那么就会抛出 CloneNotSupportedException 异常。

通过从源码的注释中可以看出，克隆的一些特性：

1）x.clone() != x 必须为真，也就是对于基础类型来说，其克隆后在堆中有两个独立且内容相同的内存区域。而对于引用类型来说，其引用也不相同。也就是说克隆对象和原始对象在java 堆(heap)中是两个独立的对象

2）x.clone().getClass() == x.getClass()  他们所属的类是同一个

3） x.clone().equals(x)   所比较的对象内容相同

1、浅克隆（shadow clone）

直接调用Object类中的clone()方法，亦即super.clone()方法即可使用浅克隆，或者说浅克隆是已经定义好的。但是浅克隆只适合基本类型和字符串类型的数据，对于引用类型或自定义类型，无法实现改变克隆而不影响本体的要求。

2、深克隆（deep clone）

深克隆可使用两种方法：重写clone()方法和使用序列化技术。后者略过，在前者中，使用super.clone()获得浅克隆对象后，对于其引用类型的成员进行单独克隆处理，若该成员为自定义类型，可以继承Cloneable接口，用浅克隆来完成其基本类型成员的克隆处理（亦即是一种递归操作）。

## static块

用static修饰的代码块表示静态代码块，当Java虚拟机（JVM）加载类时，就会执行该代码块（用处非常大，呵呵）。关于直接单独使用。就类似这样

static{…}

这是静态块或者静态区，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。

一般这样的static块可用于static成员变量的初始化，static的本质含义就是独立于所有非静态成员、可提前被执行无需个别实例化的内容。

## static静态类和其使用方法

在Java世界里，经常被提到静态这个概念，static作为静态成员变量和成员函数的修饰符，意味着它为该类的所有实例所共享，也就是说当某个类的实例修改了该静态成员变量，其修改值为该类的其它所有实例所见。最近一个项目里频繁用到static修饰的内部类，再读了一下《Effective Java》才明白为什么会用static来修饰一个内部类也就是本文的中心——静态类。

如果一个类要被声明为static的，只有一种情况，就是静态内部类。如果在外部类声明为static，程序会编译都不会过。在一番调查后个人总结出了3点关于内部类和静态内部类（俗称：内嵌类）

1.静态内部类跟静态方法一样，只能访问静态的成员变量和方法，不能访问非静态的方法和属性，但是普通内部类可以访问任意外部类的成员变量和方法

2.静态内部类可以声明普通成员变量和方法，而普通内部类不能声明static成员变量和方法。

3.静态内部类可以单独初始化:

Inner i = new Outer.Inner();

普通内部类初始化：

Outer o = new Outer();

Inner i = o.new Inner();

 静态内部类使用场景一般是当外部类需要使用内部类，而内部类无需外部类资源，并且内部类可以单独创建的时候会考虑采用静态内部类的设计，在知道如何初始化静态内部类，在《Effective Java》第二章所描述的静态内部类builder阐述了如何使用静态内部类：

public class Outer {

private String name;

private int age;

public static class Builder {

private String name;

private int age;

public Builder(int age) {

this.age = age;

}

public Builder withName(String name) {

this.name = name;

return this;

}

public Builder withAge(int age) {

this.age = age;

return this;

}

public Outer build() {

return new Outer(this);

}

}

private Outer(Builder b) {

this.age = b.age;

this.name = b.name;

}

}

静态内部类调用外部类的构造函数，来构造外部类，由于静态内部类可以被单独初始化说有在外部就有以下实现：

public Outer getOuter(){

Outer outer = new Outer.Builder(2).withName("Yang Liu").build();

return outer;}

对于静态类总结是：

1.如果类的构造器或静态工厂中有多个参数，设计这样类时，最好使用Builder模式，特别是当大多数参数都是可选的时候。

2.如果现在不能确定参数的个数，最好一开始就使用构建器即Builder模式。

## 无界和限界通配符<?>

Java代码中可以使用“？”作为数据类型通配符，也称作wildcard。一般只能在不确定类型的已定义泛型上使用，用以代表一个不确定的类型。单独的<?>是无界通配符，<? extends XXX>或者<? super XXX>称作有界或限界通配符。其中extends代表通配符限定于XXX本体和其子类。通配符的存在一般代表着类型推断，其实际类型由赋值等操作的对象来决定。

## 序列化（串行化）Serialization与transient关键字

使用Java的序列化和反序列化可以实现信息的持久存储、或者也可以实现Java对象的深克隆。序列化实际上就是将对象的内容存储在硬盘文件中，反序列化即读取这个对象的信息。二者的具体操作由ObjectOutputStream（序列化）和ObjectInputStream（反序列化）两个类和其方法writeObject(Object o)和readObject()实现。

下面来具体讲解一下序列化。能够进行序列化的类必须要实现Serializable接口，这个接口仅用作一个标识，表示这个类可以进行序列化和反序列化。

在每个要序列化的类中加入private static final long的serialVersionUID，这样，即使在某个对象被序列化之后，它所对应的类被修改了，该对象也依然可以被正确地反序列化。

如果一个类中有引用类型的实例变量（或自定义类型的变量成员），这个引用类型也要实现Serializable接口。否则就会抛出异常。或者在不实例Serializable接口时，为这个引用变量加上transient关键字即可。一旦变量被transient修饰，变量将不再是对象持久化的一部分，该变量内容在序列化后无法获得访问。被transient关键字修饰的变量不再能被序列化，一个静态变量不管是否被transient修饰，均不能被序列化。

如果一个类实现了Serializable接口，但是它的父类没有实现，那么这个类也可以序列化，但是其继承的成员不能被序列化。需要提醒的是Object超类没有实现这个序列化的接口。为什么呢？？因为如果一个类没有实现Serializable接口，但是它的父类实现了，那么这个类也可以序列化。Object实现序列化接口意味着所有的类都可以被序列化，这一概念变得无意义，现时还会造成各种问题。在进行反序列化过程中还是遵守类的初始化规则，也就是先父类后子类，但是子类的初始化并不是通过调用构造函数等手段，而是从持久化文件中存储的信息进行初始化的。

序列化过程可以指定对象的存储格式，如xml等。

## 引用，指针和参数传递

Java中不能交给用户使用指针，即可以理解为没有指针工具，另外常用的Java语句（Java7中有较为复杂的引用类型）中也没有类似C++的比较易用的引用机制。根据数据类型的区别，即基本数据类型的变量直接存储的是数据本身的值，而String和其他class的对象储存的是其实际的内存地址；还有参数传递实际上就是形参与实参的赋值传递；因此对于基本类型都是值传递，而class的对象都是引用传递或者称为地址传递。

## 字符串格式化输出String Format

字符串格式化输出可以转换原有的字符串的格式或按照一定格式转换非字符串类型数据，格式定义的核心是format specifier或称placeholder等。其通用书写格式为：

%[argument\_index$][flags][width][.precision]conversion

[]：表示可选的内容

argument\_index$：占位符的编号

flags：对输出字符串格式进行特殊加工的选项

width：The width is the minimum number of characters to be written to the output. For the line separator conversion, width is not applicable; if it is provided, an exception will be thrown.

precision：For general argument types, the precision is the maximum number of characters to be written to the output.超出字符串本身长度的部分用空格补位

conversion：对字符串输出格式的基本选项（详情见Java document中Formatter类）