(最终还是决定重新写一份Java基础相关的内容，原来因为在写这一个章节的时候没有考虑到会坚持往后边写，这次应该是更新该内容。而且很讨厌写基础的东西，内容比较琐碎，而且整理起来总会很多，有可能会打散成两个章节，但是我不保证，有可能一个章节就写完了，所以有时候希望基础的很多内容还是读者自己去看看，我基本保证把基础的内容全部都写出来，见谅。这一个章节写了过后我会把前边那个关于基础类型的章节从目录里面删除掉，以保证教材的完整性和唯一性，防止有人收藏过链接，我会继续保留在BLOG地址上边不删除，所以请读者见谅！初学者可以从本章的第三小节开始看，因为前两个章节内容也是比较概念性的，刚开始可以不去理解。这里直接从Java语法开始，不提及Java的历史以及Java的前序发展，如果有什么笔误，就发我Email：silentbalanceyh@126.com)

本章目录

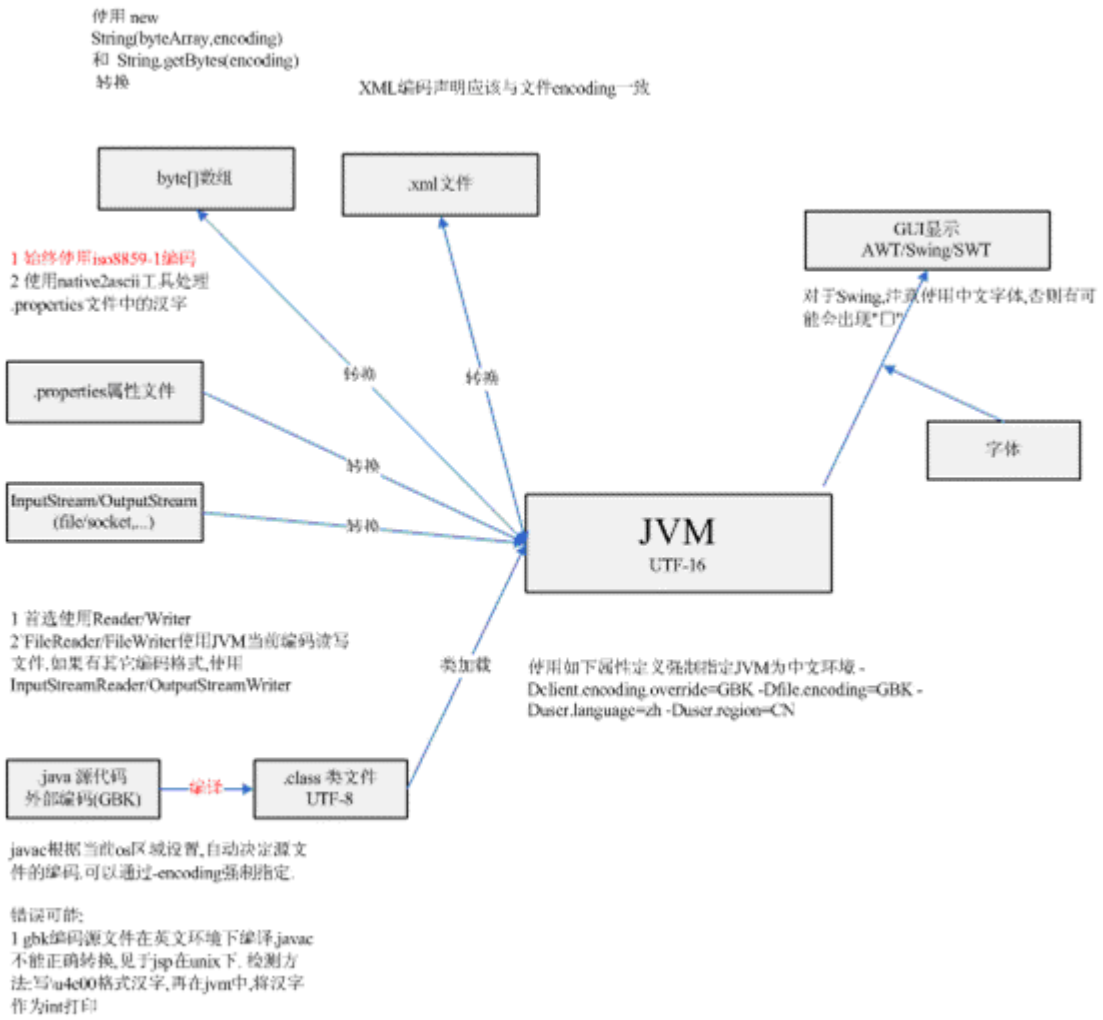
1.概念以及提纲

Java技术是一种高级的面向对象的编程语言，也是一个平台，Java技术是基于**Java虚拟机（Java Virtual Machine，JVM）**的概念——这是语言和底层软件和硬件之间的一种转换器，Java语言的所有实现都是基于JVM的，从而使Java程序可以在有JVM的任何系统上运行。

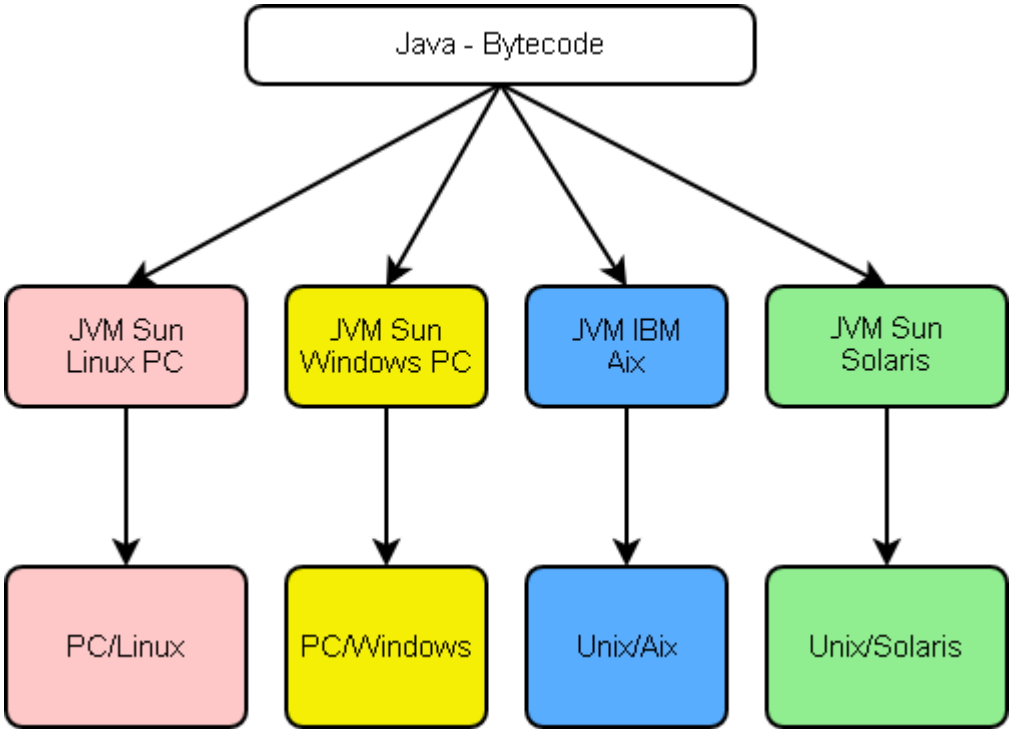
i.JVM详细介绍：

1)JVM执行原理

JVM可以称为软件模拟的计算机，它可以在任何处理器安全地兼容并且执行.class字节码。其实JVM兼容的**二进制字节码**和操作系统的**本地机器码**有一定的区别，只是针对JVM上层的调用程序而言，执行过程效果一样，所以我们一般理解就是说直接用JVM来执行二进制码，实际上二者本质有一定的**差异**，但是这一点可以理解JVM具有跨平台性。一般情况下，编程人员都是直接编写**.java**的源文件，然后用Java编译器（javac命令）对源文件进行编译，生成**.class**文件，生成的.class文件就是我们平时所说的包含了"**机器码**"的文件，实际上JVM在编译和运行过程做了两件事，先是直接将源文件编译成二进制字节码.class文件，然后进行第二次处理：解释器负责将这些二进制字节码根据本地操作系统宿主环境生成相应的本地机器码解释执行。所以可以理解的一点是为什么Java语言具有跨平台性，因为JVM提供了Java运行的一个中间层，使得操作系统和上层应用相互之间是依靠JVM中间层进行通信的，也就是说Java编写的程序是**运行在JVM**上的；再者尽管Java确实可以做到"一次编译，多处运行"，但是在不同结构的操作系统平台生成的.class文件真正在执行的时候是存在一定差异的，只是JVM本身会根据安装的不同版本进行不同的操作系统平台下本地机器码的生成及运行，所以虽然我们在Sun公司官方网站可以下载到很多不同操作系统版本的JDK，但是执行效果一样。而且还有一点，这一步操作针对开发人员是透明的，所以真正在开发过程可以放心的是不需要去担心这种问题，只要下载的版本是和符合我们运行的操作系统的，只管进行普通编程的编译、解释运行操作就可以了。



Java语言既是编译型语言，也是解释型语言，在.class文件生成之前，JVM通过javac命令对源代码进行编译操作，然后用JVM根据包含了二进制码的.class生成机器码并且解释执行。所以Java程序的跨平台特性主要是指字节码文件可以在任何具有Java虚拟机的计算机或者电子设备上运行，Java虚拟机中的Java解释器负责将字节码文件解释成为特定的机器码进行运行。Java虚拟机的建立需要针对不同的软硬件平台来实现，既要考虑处理器的型号，也要考虑操作系统的种类。由此在SPARC结构、X86结构、MIPS和PPC等嵌入式处理芯片上，在UNIX、Linux、Windows和部分实时操作系统上都可实现Java虚拟机，这也是为了在运行过程生成本地机器码而考虑的，使得JVM可以兼容不同的软硬件平台。



2)JVM的安全检查机制【参考链接：http://galaxystar.javaeye.com/blog/225615】

JVM在执行字节码的时候需要经过下边的步骤：

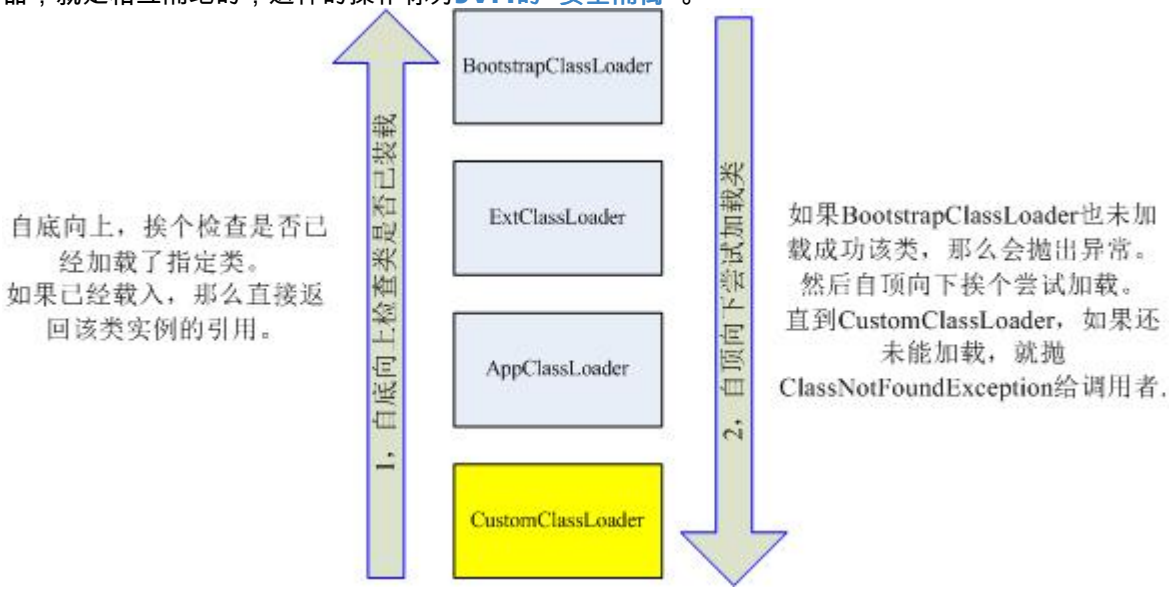
- 由类加载器（Class Loader）负责把类文件加载到Java虚拟机中（.class），在这个过程需要校验该类文件是否符合类文件规范
- 字节码校验器（Bytecode Verifier）检查该类文件的代码中是否存在着某些非法操作
- 如果字节码校验器校验通过，就由Java解释器负责把该类文件解释成机器码进行执行

JVM在上边操作过程使用了"沙箱"模型，即把Java程序的代码和数据都限制起来放在一定的内存空间执行，不允许程序访问该内存空间以外的内存。这种访问过程不仅仅是本地的，也可以是远程的，最明显的体验是使用RMI的时候。

——[$]Java的"沙箱"详解——

[1]步骤一："双亲委派类加载模型"：

双亲委派方式，指的是优先从顶层启动类加载器，自定向下的方式加载类模型，这种方式在"沙箱"安全模型里面做了第一道安全保障；而且这样的方式使得底层的类加载器加载的类和顶层的类加载器的类不能相互调用，哪怕两种类加载器加载的是同一个包里面的类，只要加载的时候不属于同一个类加载器，就是相互隔绝的，这样的操作称为**JVM的"安全隔离"**。



[2]步骤二：字节码校验：

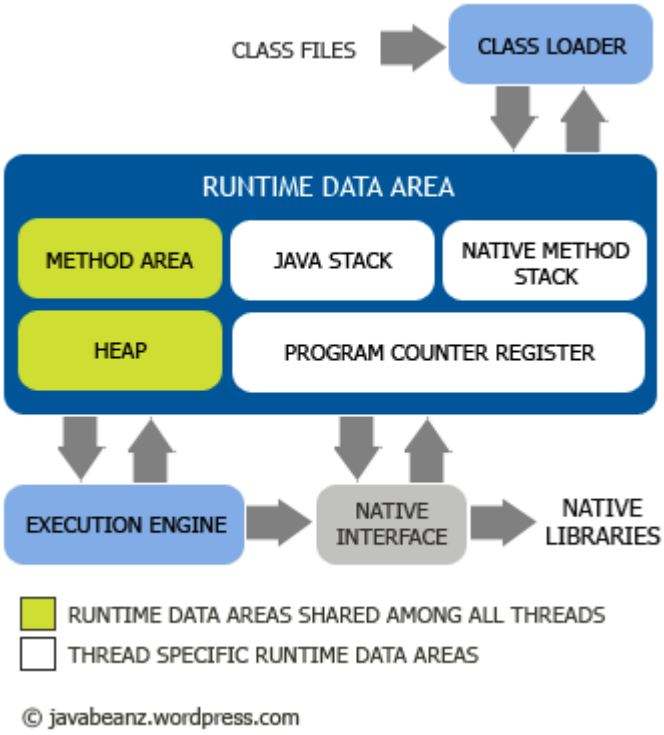
字节码校验过程需要经过四个步骤：

- **检查class文件的内部结构是否正确**，主要是检查该文件是否以某个内存地址打头一般为0xCAFEBABE，但是可能**32bit和64bit**存在一定的差异，通过这样的检查可以使得被损坏的class文件或者伪装的class文件不能够被加载，可以标记为不安全的。第一趟扫描的主要目的是保证这个字节序列正确的定义了一个类型，它必须遵从java class文件的固定格式，这样它才能被编译成在方法区中的（基于实现的）内部数据结构。
- **检查是否符合JVM语言特性里面的编译规则**，因为Java里面所有的Class都是从Object类继承过来的，一旦发现这种不安全的类存在，就直接抛异常。这次检查，class文件检验器不需要查看字节码，也不需要查看和装载任何其他类型。在这趟扫描中，检验器查看每个组成部分，确认它们是否是其所属类型的实例，它们结构是否正确。比如，方法描述符（它的返回类型，以及参数的类型和个数）在class文件中被存储为一个字符串，这个字符串必须符合特定的上下文无关文法。另外，还会检查这个类本身是否符合特定的条件，它们是由java编程语言规定的。比如，除Object外，所有类都必须要有一个超类，final的类不能被子类化，final方法也没有被覆盖，检查常量池中的条目是合法的，而且常量池的所有索引必须指向正确类型的常量池条目。
- **检查字节码是否导致JVM崩溃掉**，这里存在一个JVM中断的问题，编程过程无法捕捉Error，同样的没有办法判断程序是否因为执行的class字节码中断或者崩溃。字节码流代表了java的方法，它是由被称为操作码的单字节指令组成的序列，每一个操作码后都跟着一个或多个操作数。执行字节码时，依次执行操作码，这就在java虚拟机内构成了执行的线程，每一个线程被授予自己的java栈，这个栈是由不同的栈帧构成的，每一个方法调用将获得一个自己的栈帧——栈帧其实就是一个内存片段，其中存储着局部变量和计算的中间结果，用于存储中间结果的部分被称为操作数栈。字节码检验器要进行大量的检查，以确保采用任何路径在字节码流中都得到一个确定的操作码，确保操作数栈总是包含正确的数值以及正确的类型。它必须保证局部变量在赋予合适的值以前不能被访问，而且类的字段中必须总是被赋予正确类型的值，类的方法被调用时总是传递正确数值和类型的参数。字节码检验器还必须保证每个操作码都是合法的，即都有合法的操作数，以及对每一个操作码，合适类型的数值位于局部变量中或是在操作数栈中。这些仅仅是字节码检验器所做的大量检验工作中的一小部分，在整个检验过程通过后，它就能保证这个字节码流可以被java虚拟机安全的执行。
- **检查符号引用验证**，一个类文件，它会包含它引用的其他**类的全名和描述符**，并跟他们建立符号引用（一种虚拟的，非物理连接的方式）。当程序第一次执行到需要符号引用的位置时，JVM会检查这个符号链接的正确性，然后建立真正的物理引用(直接引用)。java虚拟机将追踪那些引用——从被验证的class文件到被引用的class文件，以确保这个引用是正确的。这次扫描可能要装载新的类。考虑到虚拟机实现上的差别，第四趟扫描可能紧随第三趟扫描发生，也有可能在第三趟扫描之后很久，当字节码被执行时才执行。动态连接是一个将符号引用解析为直接引用的过程。当java虚拟机执行字节码时，如果它遇到一个操作码，这个操作码第一次使用一个指向另一个类的符号引用，那么虚拟机就必须解析这个符号引用。在解析时，虚拟机执行两个基本任务：

  ——查找被引用的类（如果必要的话就装载它）

  ——将符号引用替换为直接引用，例如指向一个类、字段或方法的指针或偏移量

  虚拟机必须记住这个直接引用，这样当它以后再次遇到同样的引用时，就可以直接使用，而不需要重新解析该符号引用了。



[3]步骤三：内置于JVM的安全特性：

在执行期，除了符号引用验证，JVM还会对一些内建的安全特性进行检查

- 类型安全的引用转化
- 结构化的内存访问（非指针算法）
- GC
- 数组边界检查
- 空引用检查（NullPointer）

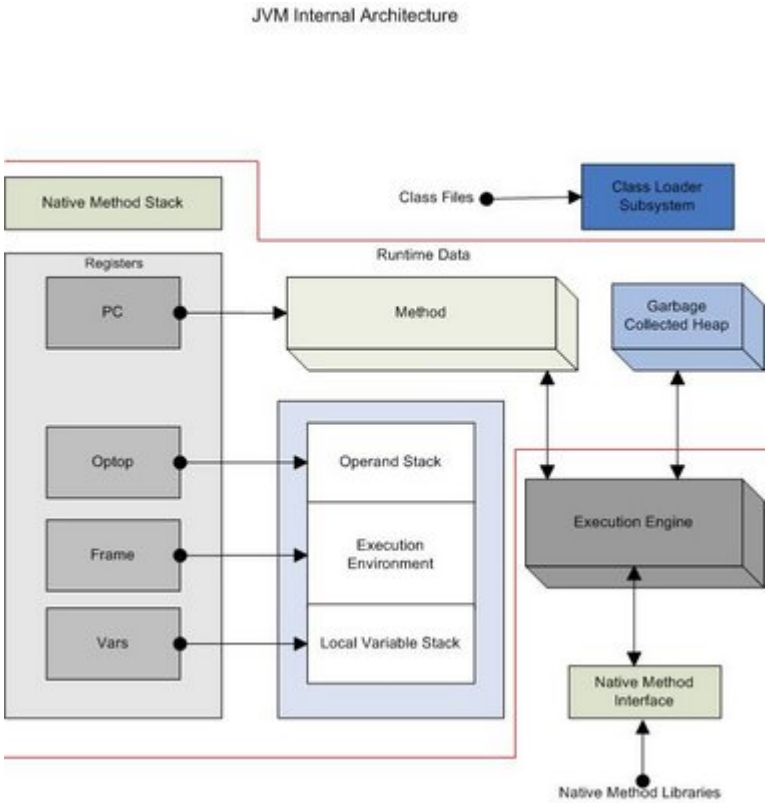
强制内存的结构话访问，避免了恶意用户在了解内存分布的情况下通过指针对JVM的内部结构进行破外。当然要了解JVM的内存分布也不是易事。JVM对运行时数据空间的分配是一个黑盒过程，完全由JVM自己决定如何分配，在class中没有任何相关的信息。

字节码检查的局限就是对于那些不经过字节码检查的方法（**如本地方法：native method**）无法验证其安全性，所以这里采用的是对动态链接库的访问控制，对于那些足有足够可信度的代码才被允许访问本地方法。具体的实现是，由安全管理器来决定代码是否有调用动态链接库的权限，因为调用本地方法必须调用动态链接库。这样一来，不可信的代码将无法通过调用本地方法来绕过字节码检查。

[4]步骤四：安全管理器SecurityManager：

这一步是编程中最接近程序的一个环节，SecurityManager存在于JavaAPI里面，是可以通过编程来完成安全策略管理的，默认情况下，Java应用是不设置SecurityManager实例的，但是这个实例却需要我们在启动的时候通过System.setSecurityManager来进行设置。一般情况下，调用了SecurityManager.checkPermission(Permission perm)来完成的，外部程序可以创建一个权限实例Permission来进行Check操作。主要应用为：

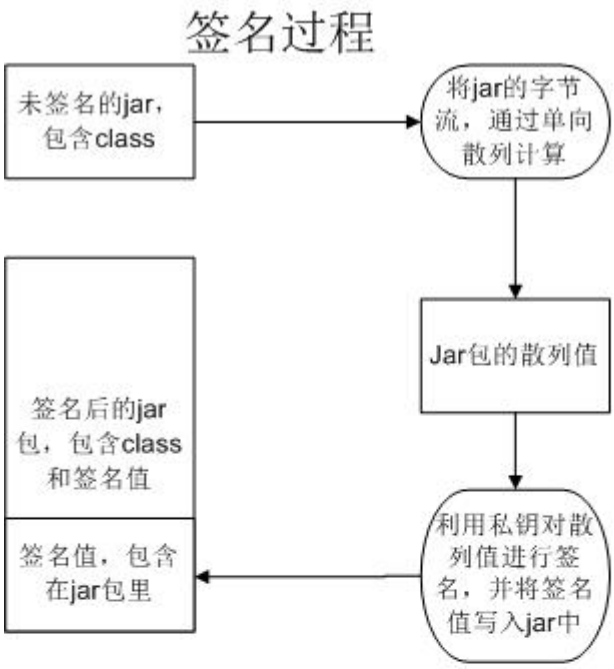
- 默认安全管理器：java.lang.SecurityManager
- 代码签名和认证
- 策略：java.security.Policy
- 权限：java.security.Permission
- 策略文件
- 保护域：CodeSource,PersimissionCollection,ProtectionDomain
- 访问控制器：java.security.AccessController

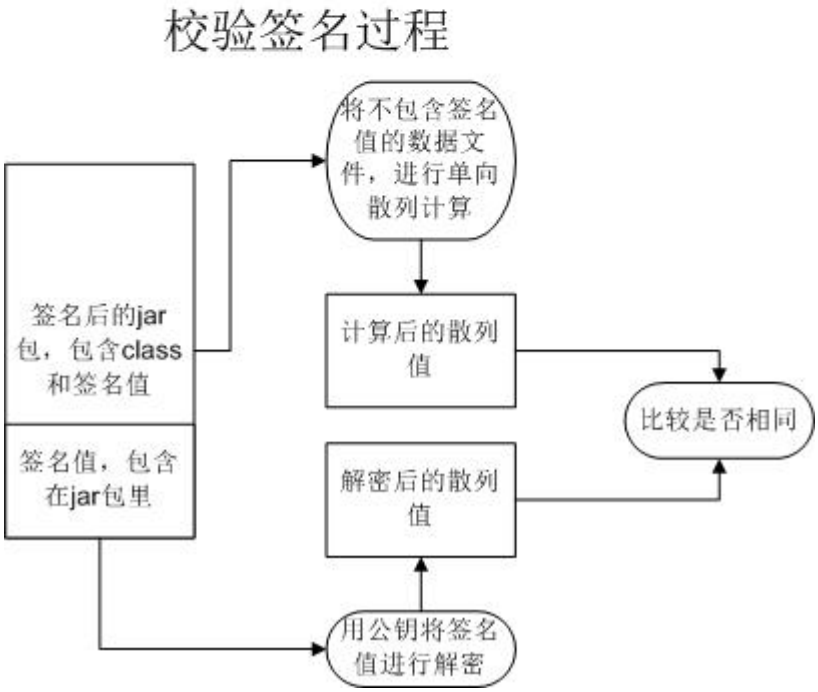


【这一步内容比较多，这里不详细讲解，带过】

[5]步骤五：Java签名/证书机制：

Java签名机制使得使用者可以安全调用外部提供的jar文件，签名必须是基于jar包的，也就是如果要使用安全模型必须要将提供的class文件打包成jar，然后使用**JDK工具（jarsigner）**对jar进行签名。新安全平台中对足够信任度的代码放宽了限制，要获得充分信任须通过代码签名来实现，若我们对某一签名团体足够信任，比如SUN，那么具有该团体签名的代码将被给予充分信任。一个基本的思路大致为，代码发布者创建私钥/公钥对，然后利用私钥对发布代码进行签名，代码使用方在获得代码发布者提供的公钥后对代码进行验证，确认代码确为该提供者提供以及在发布后未经非法修改。这其中存在一些潜在的危险，既是公钥是否是该代码发布者提供的，恶意用户可能替换掉合法的公钥，这会导致用户将给与恶意用户发布的代码以充分信任，目前的常见做法是通过一些权威的证书机构来发布证书而不是公钥，代码发布者被证书发布机构认证合格后，可以将自己的公钥交付证书发布机构，证书发布机构再通过私钥加密该公钥，从而生成证书序列。这样替换公钥的可能性就变得微乎其微。不过世上无绝对安全之事，通过证书机构发布的证书也**不是绝对安全**的途径。





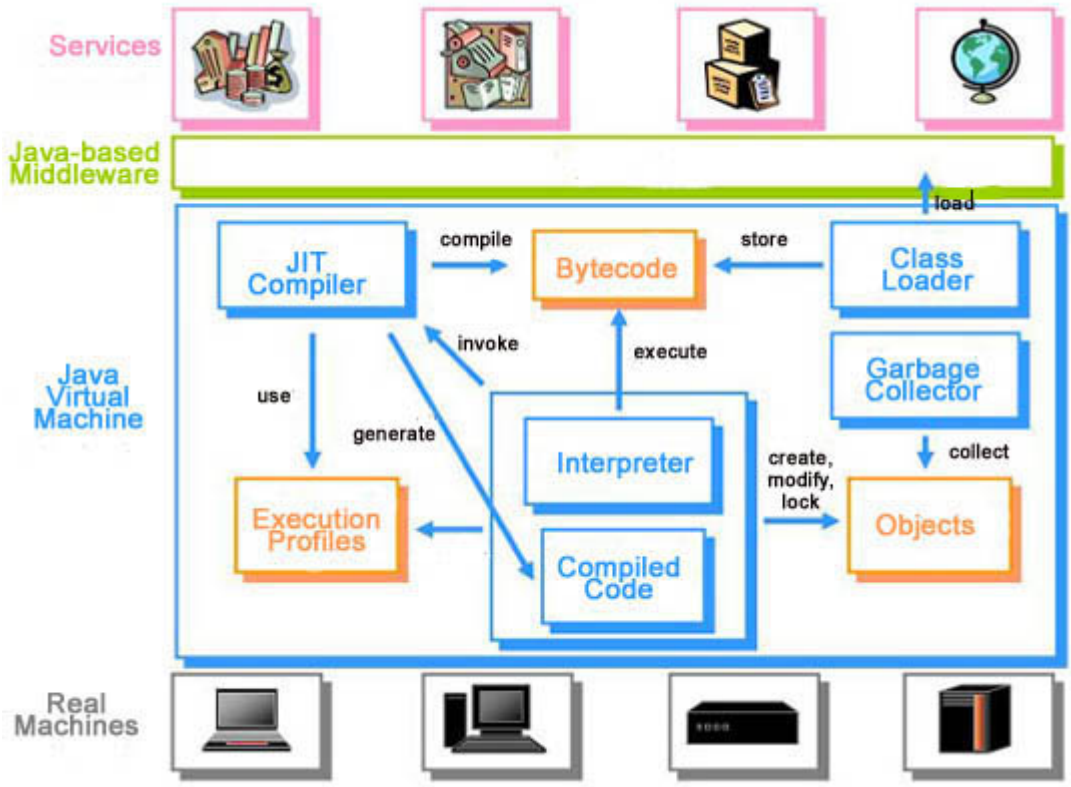
证书是在签名基础上，对签名值，再进一步做一次加密。而这次加密使用的私钥和公钥都是证书机构提供的。这种方式，是为了防止，有些恶意用户，在公钥发到你手上前，就对其做了手脚，然后再发一个动过手脚的jar给你，用动过手脚的公钥解动过手脚的jar包，是可以解开的。而使用证书后，它会对已经加密的签名值，再做一层加密，这样，到你手里，你只需要通过证书机构的公钥进行解密，然后再用jar包发布者的公钥解密就行了。(只能在一定程度上，提供一些安全性)

3)JVM的内部结构：

JVM是Java平台的核心，为了让编译产生的字节码能够更好的解释和执行，JVM主要分为6个部分【这里只是带过，想要了解JVM整体执行原理的读者可以去参考《Inside JVM》】：

- **JVM解释器**：即这个虚拟机处理字段码的CPU。
- **JVM指令系统**：该系统与计算机很相似，一条指令由操作码和操作数两部分组成。操作码为8位二进制数，主要是为了说明一条指令的功能，操作数可以根据需要而定，JVM有多达256种不同的操作指令。

- 寄存器：JVM有自己的虚拟寄存器，这样就可以快速地与JVM的解释器进行数据交换。为了功能的需要，JVM设置了4个常用的32位寄存器：pc（程序计数器）、optop（操作数栈顶指针）、frame（当前执行环境指针）和vars（指向当前执行环境中第一个局部变量的指针）。
- JVM栈：指令执行时数据和信息存储的场所和控制中心，它提供给JVM解释器运算所需要的信息。
- 存储区：JVM存储区用于存储编译过后的字节码等信息。
- 碎片回收区：JVM碎片回收是指将使用过的Java类的具体实例从内存进行回收，这就使得开发人员免去了自己编程控制内存的麻烦和危险。随着JVM的不断升级，其碎片回收的技术和算法也更加合理。JVM 1.4.1版后产生了一种叫分代收集技术，简单来说就是利用对象在程序中生存的时间划分成代，以此为标准进行碎片回收。



ii.Java平台介绍：

Java平台（Java Platform）是一种纯软件平台，它可以在各种基于硬件的平台运行，它有三个版本（**JavaSE**,**JavaME**,**JavaEE**），它由**Java应用程序接口（Java Application Programming Interface，API）和JVM**组成，Java API是一个现成的软件组件集合，可以简化Applet和应用程序的开发部署，包括健壮、安全且可互操作的企业应用程序。它涵盖了从基本对象到连网和安全性，再到XML生成和Web服务的所有东西，Java API组织成相关类和接口的库，库也可以成为包。

每个包实现包括：

- 用来编译、运行、监视、调试应用程序以及简历应用程序文档的开发工具
- 用来部署应用程序的标准机制
- 用来创建复杂的图形用户界面（GUI）的用户界面工具包
- 用来启用数据库访问和操作远程对象的集成库

1)Java平台的版本：

Java平台有三个版本，使得软件开发人员、服务提供商和设备生产商可以针对特定的市场进行开发：

[1]Java SE（Java Platform，Standard Edition）：Java SE 以前称为 J2SE。它允许开发和部署在桌面、服务器、嵌入式环境和实时环境中使用的 Java 应用程序。Java SE 包含了支持 Java Web 服务开发的类，并为 Java Platform，Enterprise Edition（Java EE）提供基础。大多数 Java 开发人员使用 Java SE 5，也称为 Java 5.0 或“Tiger”（2006 年 6 月，Java SE 6 或“Mustang”发布了 beta 版。）

[2]Java EE（Java Platform，Enterprise Edition）：这个版本以前称为 J2EE。企业版本帮助开发和部署可移植、健壮、可伸缩且安全的服务器端 Java 应用程序。Java EE 是在 Java SE 的基

础上构建的，它提供 Web 服务、组件模型、管理和通信 API，可以用来实现企业级的面向服务体系结构（service-oriented architecture，SOA）和 Web 2.0 应用程序。

[3]Java ME（Java Platform，Micro Edition）：这个版本以前称为 J2ME。Java ME 为在移动设备和嵌入式设备（比如手机、PDA、电视机顶盒和打印机）上运行的应用程序提供一个健壮且灵活的环境。Java ME 包括灵活的用户界面、健壮的安全模型、许多内置的网络协议以及对可以动态下载的连网和离线应用程序的丰富支持。基于 Java ME 规范的应用程序只需编写一次，就可以用于许多设备，而且可以利用每个设备的本机功能。

2)Java组件技术提纲：【写此提纲的目的是防止学习过程术语混乱，这里提供的分类不标准，但是覆盖了Java里面大部分我们开发过程会碰到的规范和术语】

- **Java SE中的规范：**

JavaBeans Component Architecture【JavaBeans】：一个为Java平台定义可重用软件组件的框架，可以在图形化构建工具中对这些组件进行设计

Java Foundation Classes(Swing)【JFC】：一套Java的类库，支持为基于Java的客户端应用程序构建GUI和图形化功能

JavaHelp：一个独立于平台的可扩展帮助系统，开发人员和做着可以使用它将在线帮助集成到Applet、组件、应用程序、操作系统和设备中，还可以提供Web的在线文档

Java Native Interface【JNI】：使JVM中运行的Java代码可以与用其他编程语言编写的应用程序和库进行互操作，很多时候实现部分用**C**来写，有兴趣的读者可以写一个简单的Java版任务管理器，嘿嘿，写这个可以收获很多哦！

Java Platform Debugger Architecture【JPDA】：用于Java SE的调试支持的基础结构

Java 2D API【Java 2D】：是一套用于高级2D图形和图像的类、一套提供了精确控制颜色空间定义和转换的类以及一套面向显示图像的操作符

Java Web Start：允许用户通过一次点击下载并启动特性完整的网络应用程序，而不需要安装，从而简化了Java的部署工作

Certification Path API：提供了一套用于创建、构建和检验认证路径（也称为“认证链”）的 API，可以安全地建立公共密钥到主体的映射。

Java Advanced Imaging【JAI】：提供一套面向对象的接口，这些接口支持一个简单的高级编程模型，使开发人员能够轻松地操作图像。

Java Cryptography Extension【JCE】：提供用于加密、密钥生成和协商以及 Message Authentication Code（MAC）算法的框架和实现，它提供对对称、不对称、块和流密码的加密支持，它还支持安全流和密封的对象。

Java Data Objects【JDO】：是一种基于标准接口的持久化 Java 模型抽象，使程序员能够将 Java 领域模型实例直接存储进持久化存储（数据库）中，这可以替代直接文件 I/O、串行化、JDBC 以及 **EJB Bean Managed Persistence（BMP）**或 **Container Managed Persistence（CMP）**实体 bean 等方法。

Java Management Extensions【JMX】提供了用于构建分布式、基于 Web、模块化且动态的应用程序的工具，这些应用程序可以用来管理和监视设备、应用程序和服务驱动的网络。

Java Media Framework【JMF】：可以将音频、视频和其他基于时间的媒体添加到 Java 应用程序和 applet 中。

Java Secure Socket Extensions【JSSE】：支持安全的互联网通信，实现了 SSL（Secure Sockets Layer）和 TLS（Transport Layer Security）的 Java 版本，包含数据加密、服务器身份验证、消息完整性和可选的客户机身份验证等功能。

Java Speech API【JSAPI】：包含 Grammar Format【**JSGF**】和 Markup Language【**JSML**】规范，使 Java 应用程序能够将语音技术集成到用户界面中。JSAPI 定义一个跨平台的 API，支持命令和控制识别器、听写系统和语音识别器。

Java 3D API【Java 3D】：它提供一套面向对象的接口，这些接口支持一个简单的高级编程模型，开发人员可以使用这个 API 轻松地将可伸缩的独立于平台的 **3D 图形**集成到 Java 应用程序中。

Metadata Facility【JMI】：允许给类、接口、字段和方法标上特定的属性，从而使开发工具、部署工具和运行时库能够以特殊方式处理它们，Java元数据接口（使用**Annotation**实现）

Java Content Repository API：用于访问 Java SE 中独立于实现的内容存储库的 API。内容存储库是一个高级信息管理系统，它是传统数据存储库的超集。

Concurrency Utilities：一套中级实用程序，提供了并发程序中常用的功能。

Juxtapose【JXTA】：一个用来解决P2P计算的开放的网络计算平台。JXTA P2P平台使开发者在其上建立PtoP的应用。

- **Java EE中的规范：**

Java Database Connectivity【JDBC】：使用户能够从 Java 代码中访问大多数表格式数据源，提供了对许多 SQL 数据库的跨 DBMS 连接能力，并可以访问其他表格式数据源，比如电子表格或平面文件。

Java Naming and Directory Interface【JNDI】：为 Java 应用程序提供一个连接到企业中的多个命名和目录服务的**统一接口**，可以无缝地连接结构不同的企业命名和目录服务。

Enterprise JavaBeans【EJB】：J2EE技术之所以赢得某体广泛重视的原因之一就是EJB。它们提供了一个框架来开发和实施分布式商务逻辑，由此很显著地简化了具有可伸缩性和高度复杂的企业级应用的开发。EJB规范定义了EJB组件在何时如何与它们的容器进行交互作用。容器负责提供公用的服务，例如目录服务、事务管理、安全性、资源缓冲池以及容错性。但这里值得注意的是，EJB并不是实现J2EE的唯一途径。正是由于J2EE的开放性，使得有的厂商能够以一种和EJB平行的方式来达到同样的目的。

Remote Method Invoke【RMI】：正如其名字所表示的那样，RMI协议调用远程对象上方法。它使用了序列化方式在客户端和服务器端传递数据。RMI是一种被EJB使用的更底层的协议。

Java IDL/CORBA：在Java IDL的支持下，开发人员可以将Java和CORBA集成在一起。他们可以创建Java对象并使之可在CORBA ORB中展开，或者他们还可以创建Java类并作为和其它ORB一起展开的CORBA对象的客户。后一种方法提供了另外一种途径，通过它Java可以被用于将你的新的应用和旧的系统相集成。

Portlet Specification：定义了一套用于 Java 门户计算的 API，可以解决聚合、个人化、表示和安全性方面的问题。

JavaMail：提供了一套对邮件系统进行建模的抽象类。

Java Message Service【JMS】：为所有与 JMS 技术兼容的消息传递系统定义一套通用的消息概念和编程策略，从而支持开发可移植的基于消息的 Java 应用程序。

JavaServer Faces【JSF】：提供一个编程模型，帮助开发人员将可重用 UI 组件组合在页面中，将这些组件连接到应用程序数据源，将客户机生成的事件连接到服务器端事件处理程序，从而轻松地组建 Web 应用程序。

JavaServer Pages【JSP】：允许 Web 开发人员快速地开发和轻松地维护动态的独立于平台的 Web 页面，并将用户界面和内容生成隔离开，这样设计人员就能够修改页面布局而不必修改动态内容。这种技术使用类似 XML 的标记来封装为页面生成内容的逻辑。

Java Servlets：提供一种基于组件的独立于平台的方法，可以构建基于 Web 的应用程序，同时避免 CGI 程序的性能限制，从而扩展并增强 Web 服务器的功能。

J2EE Connector Architecture【JCA】：为将 J2EE 平台连接到各种结构的 Enterprise Information Systems (EIS) 定义了一个标准的体系结构，它定义了一套可伸缩的安全的事务性机制，使 EIS 厂商能够提供标准的资源适配器，可以将这些资源适配器插入应用服务器中。

J2EE Management Specification【JMX】：为 J2EE 平台定义了一个信息管理模型。根据其设计，J2EE Management Model 可与多种管理系统和协议进行互操作；包含模型到 Common Information Model (CIM) 的标准映射，CIM 是一个 SNMP Management Information Base (MIB)；还可以通过一个驻留在服务器上的 EJB 组件 —— **J2EE Management EJB Component【MEJB】** —— 映射到 Java 对象模型。

Java Transaction API【JTA】：是一个独立于实现和协议的高级 API，它使应用程序和应用服务器可以访问事务。

Java Transaction Service【JTS】：JTS是CORBA OTS事务监控的基本的实现。JTS规定了事务管理器的实现方式。该事务管理器是在高层支持Java Transaction API (JTA)规范，并且在较底层实现OMG OTS specification的Java映像。JTS事务管理器为应用服务器、资源管理器、独立的应用以及通信资源管理器提供了事务服务.....

Java Authentication and Authorization Service【JAAS】：实现了标准的 Pluggable Authentication Module (PAM) 框架的 Java 版本并支持基于用户的授权，使服务能够对用户进行身份验证和访问控制。

JavaBeans Activation Framework【JAF】：JavaMail利用JAF来处理MIME编码的邮件附件。MIME的字节流可以被转换成Java对象，或者转换自Java对象。大多数应用都可以不需要直接使用JAF。

Java API for XML Web Service【JAX-WS】：一组XML web services的JAVA API。JAX-WS允许开发者可以选择RPC-oriented或者message-oriented 来实现自己的web services。

Java API for XML Processing【JAXP】：允许 Java 应用程序独立于特定的 XML 处

理实现对 XML 文档进行解析和转换，允许灵活地在 XML 处理程序之间进行切换，而不需要修改应用程序代码。
Java API for XML Binding【JAXB】：允许在 XML 文档和 Java 对象之间进行自动的映射。
SOAP with Attachments API for Java【SAAJ】：使开发人员能够按照 SOAP 1.1 规范和 SOAP with Attachments note 生成和消费消息。
Streaming API for XML【StAX】：就提供了两种方法来处理 XML：文档对象模型（DOM）方法是用标准的对象模型表示 XML 文档；Simple API for XML (SAX) 方法使用应用程序提供的事件处理程序来处理 XML。JSR-173 提出了一种面向流的新方法：Streaming API for XML (StAX)。其最终版本于 2004 年 3 月发布，并成为了 JAXP 1.4（将包含在即将发布的 Java 6 中）的一部分。
Java API for XML Registries【JAXR】：提供了一种统一和标准的Java API，用于访问不同类型的基于XML的元数据注册中心。JAXR目前实现支持ebXML Registry2.0版和UDDI2.0版。未来可以支持更多的类似的注册中心。JAXR为客户端提供了API，与XML注册中心进行交互，同时为服务中心提供者提供了服务提供者接口（SPI），这样，注册中心的实现是可插拔的。JAXR API将应用代码与下层的注册中心机制相隔离。当编写基于JAXR的客户端浏览和操作注册中心时，当注册中心更换时，比如从UDDI更换为ebXML，代码不需要进行修改。
Java Persistence API【JPA】：通过JDK 5.0注解或XML描述对象 - 关系表的映射关系，并将运行期的实体对象持久化到数据库中，一个标准的**ORM框架**。
Java Authorization Service Provider Contract for Containers【JACC】：在J2EE应用服务器和特定的授权认证服务器之间定义了一个连接的协约，以便将各种授权认证服务器插入到J2EE产品中去。
Java API for RESTful Web Services【JAX-RS】：定义一个统一的规范，使得 Java 程序员可以使用一套固定的接口来开发 REST 应用，避免了依赖于第三方框架。同时，JAX-RS 使用 POJO 编程模型和基于标注的配置，并集成了 JAXB，从而可以有效缩短 REST 应用的开发周期。

- **Java ME中的规范：**
  Connected Limited Device Configuration【CLDC】：是组成资源有限的移动信息设备的 Java 运行时环境的两种配置之一。CLDC 描述最基本的库和虚拟机特性，所有包含 K 虚拟机（K virtual machine，KVM）的 J2ME 环境实现中都必须提供这些库和特性。
  Mobile Information Device Profile【MIDP】：是组成资源有限的移动信息设备的 Java 运行时环境的两种配置之一。MIDP 提供核心应用程序功能，包括用户界面、网络连接、本地数据存储和应用程序生命周期管理。
  Connected Device Configuration【CDC】：是一个基于标准的框架，用来构建和交付可以跨许多连接网络的消费类设备和嵌入式设备共享的应用程序。
  Mobile 3D Graphics API for J2ME【M3G】：是一种轻量的交互式 3D 图形 API，它作为可选的包与 J2ME 和 MIDP 结合使用。

【*：上边这份清单基本上涵盖了Java平台所有的术语以及相关介绍，因为WebBeans是Java EE 6的新规范，而且JBI和SCA具体是为ESB开发存在的，这三个规范我没有归纳进来，有兴趣的读者可以自己去查阅一下，还有一点需要说明JXTA以前不属于Java规范里面的，也是因为该组件用起来比较方便貌似现在纳入JSR规范了，具体情况如何我**没有查证**，见谅，若有遗漏掉的规范提纲，也请有心的读者来Email告知，谢谢：silentbalanceyh@126.com。这套教程我在书写的时候我只会写一部分与我自己开发相关的使用过的组件技术，其他我不熟悉的可能不会涉及到。而且需要提醒一下读者的是：**SSH（Struts+Spring+Hibernate）**框架虽然很流行，但是不属于Sun公司Java EE里面的官方标准规范，这三个框架是三个开源框架，所以：**JavaEE的规范定义里面并不包含SSH！**】

2.语言基础【*：本教材不会讲JDK的环境配置，环境配置以后一必要我在BLOG的配置分类里面单独讲解】

i.开发第一个Java程序：

步骤一：书写.java源代码
步骤二：使用javac命令编译.java源代码，将该代码编译成.class字节码
步骤三：使用java命令运行.class字节码
接着按照步骤，书写第一个Java程序：

```
/**
 *我的第一个Java程序
 **/
class Hello{
```

```
}

public class HelloJava {
    public static void main(String[] args) {
        System.out.println("Hello Java");
    }
}
```

将上边的代码保存成HelloJava.java文件名，放于D盘根目录。【*：这里的放置位置根据读者爱好自己决定】

第二步：使用命令行工具：

```
javac HelloJava.java
```

第三步：运行.class文件

```
java HelloJava
```

输出应该为：

```
Hello Java
```

针对这样一个过程有几点需要说明：

[1]关于Java源文件：Java的源文件是以**.java结尾**的，每一个源文件里面可以定义很多class，但是只能定义一个public的class，如果不定义public的class，文件名必须和其中至少一个class的名称一致；若定义了一个public的class，文件名**必须**和定义的public的class的类名**一致**。【*：这点是Java的语言规范，但是从语言设计上来讲这是没有必要的，实际上编译器不应该把文件名和类名关联，但是Java语言是一个特殊，学过Scala和C#的读者应该就明白这一点，所以这一点在Java使用的时候需要特别小心。】

[2]关于函数入口：一个Java程序可能由**一个或者多个**class构成，上边使用了javac过后，同一个目录下可以生成两个.class文件，分别是Hello.class和HelloJava.class，不论怎样必须在运行的类里面（使用java命令的时候后边跟的类名的类）定义一个Java应用**程序的入口**：public static void main(String args[])，在这里，该定义成为Java语言的主函数入口，也是程序运行的起点，上边的程序的入口就在HelloJava类里面。main()方法的签名是不可以更改的，因此下边的函数入口的写法都是**不对**的：

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

使用上边三种入口函数的定义都会使得JVM没有办法找到主函数入口，后两种定义会抛出下边的异常，第一种情况比较特殊，JVM会提示：Main method not public，也就是说第一种情况JVM能够找到主函数，但是该主函数**不符合**Java应用程序的运行规范。

```
Exception in thread "Main Thread" java.lang.NoSuchMethodError: main
```

[3]在上边的程序里面，HelloJava的类可以不是public的时候，这里需要说明的是，不是说因为某个类是public主函数入口就必须放在该类里面，读者可以试试下边的代码段：

```
class HelloTester{
    public static void main(String[] args){
        System.out.println("Hello World");
    }
}

public class MainTester {

}
```

当我们使用javac MainTester.java的时候，编译是可以通过的，而且会生成两个类文件：HelloTester.class和MainTester.class。注意这里的代码结构，按照第一点提到的，这个.java文件只能保存为MainTester.java，如果这段代码保存为HelloTester.java文件，编译的时候会抛出下边的异常：

```
HelloTester.java:8 : class MainTester is public,should be declared in a file named
MainTester.java
```

也就是说一旦在**一个.java源文件里面定义了一个类为public，那么这个源文件的名称就被限制了必须和public的类保持一致**。上边的代码经过编译过后使用下边的命令读者可以测试以下：

```
java HelloTester【这种做法会输出：Hello World，因为在这个类里面找到了主函数的入口】
java MainTester【这种做法不会成功，因为在执行类MainTester里面，JVM找不到主函数的入口】
```

[4]main函数的参数：

对于java里面的主函数入口方法main，JVM有一定的限制，首先是不可以有返回值，因此返回值为void，而且必须有一个参数，类型为String[]的，这也是Java语言里面的规范，这个参数会匹配命令行里面的参数，至于参数名是里面唯一可以更改的，所以上边的定义可以为：

```
public static void main(String[] myInputValue)
```

一般情况下，约定使用args的名称，而这里的args代表了命令行的输入参数，这一点可以提供一个带命令行参数的例子：

```
public class MainTester {
   public static void main(String args[])
   {
      System.out.println(args[0]);
      System.out.println(args[1]);
   }
}
```

把上边的代码保存为MainTester.java，然后使用javac命令编译，这一步之前所有的的内容都不用担心，和前边一样，但是运行的时候可能需要下边这种格式：

```
java MainTester hello1 hello2
```

上边这段代码会输出为：

```
hello1
hello2
```

也就是说，该参数读取的是在java ClassName之后的参数，按照空白为分隔符，然后输入的内容就为参数的值，这些值构成了一个String[]的参数列表，上边代码里面args.length的值为2，第一个参数为hello1，第二个参数为hello2，这就是主函数参数列表里面的参数的使用

小节一下上边几点：

- 使用javac命令的时候：如果.java里面没有public的class，直接针对class的定义进行编译操作，这种情况下，.java源文件的文件名只要和里面的任何一个类的类名一致就可以了；如果.java里面定义了一个类是public的，这种情况下，.java源文件的名称只能和这个类的类名保持一致，否则编译不会通过。
- 使用java命令的时候：在使用java命令的时候，后边敲入命令不需要填写文件全名【不需要java ClassName.class格式】，只是需要填写类名就可以了，这种情况下，JVM会在ClassName对应的类定义里面去寻找主函数入口，如果找不到就会抛出异常，找到过后就可以执行，也可以这样讲，包含主函数的类不一定在一个public的类里面可以是非public的。
- Java应用程序在执行的时候，必须有一个主函数的入口【*：这里不包括Applet的运行，因为Applet里面有特殊的入口以及特殊的运行机制】，而且该主函数的入口必须定义为：public static void main(String args[])
- 这里还有一点需要说明：main函数是可以抛异常的，JVM在检查方法签名的时候，不会检查main后边带的异常定义，比如下边这种方式也是定义函数入口的正确方式，也就是说带异常抛出的时候仅仅检测方法签名就可以了：
  public static void main(String args[]) throws Exception
- 注意主函数传入参数的使用，args参数实际上就是在使用java命令的时候传入的参数的值的使用，这些使用可以通过写几个简单的程序来实现，在参数使用的过程里面，一般情况下在编写程序的时候针对参数进行一定的检测，上边使用参数的代码实际上是会抛异常的，因为会存在数组越界的问题，为了保证数据不会越界，一般可以通过代码if(args.length > 0)来对传入的参数进行判断，而且把使用参数的代码放在try-catch块里面进行数组越界异常的捕捉。

ii.Java的注释：

在Java里面主要有三种注释：**行注释、段落注释、文档注释**

1)行注释：行注释也成为单行注释，行注释使用“//注释文字”的格式来对某一行的代码进行注释或者加以说明

```
public class LineComment
{
   //这是单行注释的范例
   public static void main(String args[])
   {
      //这只是一个单行注释的例子
      System.out.println("Single Line Comment");
   }
}
```

上边代码里面//后边的文字就是行注释的一个例子

2)段注释：段注释也成为多行注释，通常是当说明文字比较长的时候的注释方法

```
public class MultiCommont
{
   /*
    *这是段注释的一个简单的例子
    *这里是函数入口main方法
    */
   public static void main(String args[])
   {
      System.out.println("Multi Lines Comments");
   }
}
```

3)文档注释：文档注释是Java里面的一个比较厉害的功能，它可以用于注释类、属性、方法等说明，而且通过JDK工具javadoc直接生成相关文档，文档注释的基本格式为"/**...*/"，不仅仅如此，文档注释本身还存在语法

[1]文档和文档注释的格式化：

生成的文档是HTML格式的，而这些HTML格式的标识符并不是javadoc加的，而是我们在写注释的时候写上去的。因此在格式化文档的时候需要适当地加入HTML标签，例如：

```
/**
 *This is first line.<br/>
 *This is second line.<br/>
 *This is third line.<br/>
 **/
```

[2]文档注释的三部分：

根据在文档中显示的效果，文档注释可以分为三个部分，这里举个例子：

```
/**
 *testDoc方法的简单描述
 *<p>testDoc方法的详细说明</p>
 *@param testInput String 打印输入的字符串
 *@return 没有任何返回值
 **/
public void testDoc(String testInput)
{
   System.out.println(testInput);
}
```

简述：文档中，对于属性和方法都是现有一个列表，然后才在后面一个一个的详细说明，列表中属性名或者方法名后面那段说明都是简述。

详细说明：该部门对属性或者方法进行了详细说明，在格式上不需要特殊的要求，可以写成前边讲的HTML的格式，如同上边第二行的内容。【*：这里有个技巧就是简述和详细说明尽量不要重复，在简述中出现过的内容不需要在详细说明中进行第二次描述，可以理解为详细说明是简述的一种扩展。后边章节的概念说明代码大部分我都没有写注释，也请各位读者见谅！】

特殊说明：除开上边的两部分，最后一个部分就是特殊说明部分，特殊说明部分使用JavaDoc标记进行，这些都是JavaDoc工具能够解析的特殊标记，这一点下边章节将会讲到

[3]使用javadoc标记：

javadoc标记是java插入文档注释中的特殊标记，它们用于识别代码中的特殊引用。javadoc标记由"@"以及其后跟着的标记类型和专用注释引用组成。它的格式包含了三个部分：

@、标记类型、专用注释引用

有时候我们也可以直接理解为两个方面：@和标记类型、专用注释引用；Javadoc工具可以解析Java文档注释中嵌入的特殊标记，这些文档标记可以帮助自动从源代码生成完整的格式化API，标记用"@"符号开头，区分大小写，必须按照正确的大小写字母输入。标记必须从一行的开头开始，否则会被视为普通文本，而且按照规定应将相同名字的标记放一起，比如所有的@see标记应该放到一起，接下来看一看每一种标记的含义。

@author(**1.0**)：**语法[**@author name-text**]**

当使用-author选项的时候，用指定的name-text在生成文档的时候添加"Author"项，文档注释可以包含多个@author标记，可以对每个@author指定一个或者多个名字。

@deprecated(**1.0**)：**语法[**@deprecated deprecated-text**]**

添加注释，只是不应该再使用该API（尽管是可用的），Javadoc工具会将deprecated-text移动到描述前面，用斜体显示，并且在它前边添加粗体警告："不鼓励使用"。deprecated-text的首句至少应该告诉用户什么时候开始不鼓励使用该API以及使用什么替代它。Javadoc仅将首句复制到概览部分

和索引中，后面的语句还可解释为什么不鼓励使用，应该还包括一个指向替代API的{@link}标记
【1.2版本用法】

- 对于Javadoc 1.2，使用{@link}标记，这将在需要的地方创建内嵌链接，如：
  /**
  *@deprecated 在JDK X.X中，被{@link #methodName(paramList)}取代
  **/
  【*：在上边的标记里面X.X指代JDK的版本，后边的链接链接的是方法的签名，methodName为方法名，paramList为参数列表】
- 对于Javadoc 1.1，标准格式是为每个@deprecated标记创建@see标记（它不可内嵌）

@exception(**1.0**)：**语法[**@exception class-name description**]**
@throws(**1.2**)：**语法[**@throws class-name description**]**
以上两个标记是同义词，用class-name和description文本给生成的文档添加"抛出"子标题，其中class-name是该方法可抛出的异常名。
{@link}(**1.2**)：**语法[**{@link name label}**]**
出入指向指定name的内嵌链接，该标记中name和label的语法与@see标记完全相同，如下所述，但是产生的内嵌链接而不是在"参见"部分防止链接。该标记用花括号开始并用花括号结束，以使它区别于其他内嵌文本，如果需要在标签内使用"}"，直接使用HTML的实体表示法：**}**
@param(**1.0**)：**语法[**@param parameter-name description**]**
给"参见"部分添加参数，描述可以继续到下一行进行操作，主要是提供了一些参数的格式以及描述
@return(**1.0**)：**语法[**@return description**]**
用description本文添加"返回"部分，该文本应描述值的返回类型和容许范围
@see(1.0)：**语法[**@see reference**]**
该标记是一个相对复杂的标记，添加"参见"标题，其中有指向reference的链接或者文本项，文档注释可包含任意数目的@see标记，它们都分组在相同的标题下，@see有三种格式：

- **@see "string" 注**：该形式在JDK 1.2中没有用，它不打印引用文本
  为string添加文本项，不产生链接，string是通过该URL不可用的书籍或者其他信息引用，Javadoc通过查找第一个字符为双引号（ " ）的情形来区分它前边的情况，比如：
  @see "这是Java教材，主要是提供给初学者"
  上边的注释将会生成：
  参见：
  "这是Java教材，主要是提供给初学者"
- **@see Java某章节**
  添加URL#value定义的链接，其中URL#value是相对URL或者绝对URL，JavaDoc工具通过查找第一个字符小写符号（ < ）区分它与其他情况，比如：
  @see 测试规范
  上边的注释将会生成：
  参见：
  测试规范
- **@see package.class#member label【比较常用的一种格式】**
  添加带可见文本label的链接，它指向Java语言中指定名字的文档。其中label是可选的，如果省略，则名字作为可见文本出现，而且嵌在<code>HTML标记中，当想要缩写可见文本或不同于名字的可见文本的时候，可以使用label。
  ——package.class#member是Java语言中的任何有效名字——包名、类名、接口名、构造函数名、方法名或域名——除了用hash字符(#)取代成员名前面的点之外，如果该名字位于带文档的类中，则Javadoc将自动创建到它的链接，要创建到外部的引用链接，可使用-link选项，使用另外两种@see形式中的任何一种引用不属于引用类的名字的文档。
  ——label是可选文本，它是链接的可见标签，label可包含空白，如果省略label，则将显示package.class.member，并相对于当前类和包适当缩短
  ——空格是package.class#member和label之间的分界符，括号内的空格不表示标签的开始，因此在方法各参数之间可使用空格

@see package.class#member的典型形式
引用当前类的成员 @see #field @see #method(Type,Type,...) @see #method(Type argname,Type argname,...) **引用当前包或导入包中的其他类**

```
@see Class#field
@see Class#method(Type,Type,...)
@see Class#method(Type argname,Type argname,...)
@see Class

引用其他包（全限定）
@see package.Class#field
@see package.Class#method(Type,Type,...)
@see package.Class#method(Type argname,Type argname,...)
@see package.Class
@see package
```

简单说明一下：

——第一套形式（没有类和包）将导致 Javadoc 仅搜索当前类层次。它将查找当前类或接口、其父类或超接口、或其包含类或接口的成员。它不会搜索当前包的其余部分或其他包（搜索步骤 4-5）。

——如果任何方法或构造函数输入为不带括号的名字，例如 getValue，且如果没有具有相同名字的域，则 Javadoc 将正确创建到它的链接，但是将显示警告信息，提示添加括号和参数。如果该方法被重载，则 Javadoc 将链接到它搜索到的第一个未指定方法。

——对于所有形式，内部类必须指定为 outer.inner，而不是简单的 inner。

——如上所述，hash 字符（#）而不是点（.）用于分隔类和成员。这使 Javadoc 可正确解析，因为点还用于分隔类、内部类、包和子包。当 hash 字符（#）是第一个字符时，它是绝对不可少的。但是，在其他情况下，Javadoc 通常不严格，并允许在不产生歧义时使用点号，但是它将显示警告。

@see标记的搜索次序——JavaDoc将处理出现在**源文件（.java）**、**包文件（package.html）**或**概述文件（overview.html）**中的@see标记，在后两种文件中，必须完全限定使用@see提供的名字，在源文件中，可指定全限定或部分限定的名字，@see的搜索顺序为：

- 当前类或接口
- 任何包含类和接口，先搜索最近的
- 任何父类和超接口，先搜索最近的
- 当前包
- 任何导入包、类和接口，按导入语句的次序搜索

@since(**1.1**)：**语法[**@since since-text**]**

用since-text指定的内容给生成文档添加"Since"标题，该文本没有特殊内部结构，该标记表示该改变或功能自since-text所指定的软件件版本后就存在了，例如：@since JDK 1.4

@serial(**1.2**)：**语法[**@serial field-description**]**

用于缺省的可序列化域的文档注释中，可选的field-description增强了文档注释对域的描述能力，该组合描述必须解释该域的意义并列出可接受值，如果有需要描述可以多行，应该对自Serializable类的最初版本之后添加的每个可序列化的域添加@since标记。

@serialField(**1.2**)：**语法[**@serialField field-name field-type field-description**]**

简历Serializable类的serialPersistentFields成员的ObjectStreamField组件的文档，应该对每个ObjectStreamField使用一个@serialField标记

@serialData(**1.2**)：**语法[**@serialData data-description**]**

data-description建立数据（尤其是writeObject方法所写入的可选数据和Externalizable.writeExternal方法写入的全部数据）序列和类型的文档，@serialData标记可用于writeObject、readObject、writeExternal和readExternal方法的文档注释中

@version(**1.0**)：**语法[**@version version-text**]**

当使用-version选项时用version-text指定的内容给生成文档添加"版本"子标题，该文本没有特殊内部结构，文档注释最多只能包含一个@version标记。

{@code}(**1.5**)：**语法[**{@code text}**]**

该标签等同于<code>{@literal}</code>，里面可以直接过滤掉HTML的标签，可以不用<和>来显示（<和>），在这个代码块里面的text部分，可以直接书写代码，即使使用了Hello，在HTML里面也不会识别成为加粗的Hello，而还是原来的代码段Hello的格式输出

{@docRoot}(**1.3**)：**语法[**{@docRoot}**]**

代表相对路径生成的文件的（目标）的根从任何产生的网页目录，当您要包括如版权页或公司徽标文件的时候它是有用的，你要引用所生成的网页，链接从每个页面底部的版权页面是常见的。（@docRoot将标记可用于在命令行，并在两个文档注释：这个标记是有效的，在所有文档注释：概述、包装类、接

口、构造、方法和领域，包括任何标记文本的一部分（如@return，@param和@deprecated的使用）。

比如：

```
/**
 * See the <a href="{@docRoot}/copyright.html">Copyright</a>.
 **/
```

{@inheritDoc}(**1.4**)：**语法[**{@inheritDoc}**]**

继承（拷贝）文档从最近的"继承类或可执行的接口"到当前在这个标签的位置的文档注释内容，这使您可以编写更多与继承相关的一般性文档

下边的情况比较适合：

- 在一个方法的主要描述块，在这种情况下，主要描述是整个继承树结构里面的类和接口的一个拷贝。
- 在文本参数返回的@return @param和@throws等方法标记，在这种情况下，文本标记是整个层次结构里面标签的拷贝。

{@linkplain}(**1.4**)：**语法[**{@linkplain package.class#member label}**]**

和{@link}类似，除非链接的label是用普通文本的格式显示的，当文本是普通文本的时候该标签就能够起作用，例如：

```
Refer to {@linkplain add() the overridden method}
```

这个会显示成：

Refer to the overridden method

{@value}(**1.4**)：**语法[**{@value package.class#field}**]**

主要用来显示静态字段的值：

```
/**
 * The value of this constant is {@value}
 **/
public static final String SCRIPT_START = "script";
```

[4]JavaDoc标记的举例：

——[$]一个使用JavaDoc标记的例子——

```
/**
 * @author Lang Yu
 * @version 1.2
 */
public class JavaDocBasic {
   /**
    * @see "Main Function JavaDoc"
    * @since JDK 1.4
    * @param args The params of console
    **/
   public static void main(String args[]){
      System.out.println("Hello World!");
   }
}
```

例如有这样一小段代码，在我机器上我放在了**D:\Source\work**下，然后进入该目录下，使用下边的命令：

```
D:\Source\work>javadoc -d doc JavaDocBasic.java
```

通过这样的命令使用，在执行该命令了过后电脑上有下边这样的输出，而且去目录下边可以看到一个doc文件夹，用浏览器打开里面的index.html就可以看到生成的文档的内容：

```
Loading source file JavaDocBasic.java...
Constructing Javadoc information...
Standard Doclet version 1.6.0_16
Building tree for all the packages and classes...
Generating doc\JavaDocBasic.html...
Generating doc\package-frame.html...
Generating doc\package-summary.html...
Generating doc\package-tree.html...
Generating doc\constant-values.html...
Building index for all the packages and classes...
Generating doc\overview-tree.html...
```

```
Generating doc\index-all.html...
Generating doc\deprecated-list.html...
Building index for all classes...
Generating doc\allclasses-frame.html...
Generating doc\allclasses-noframe.html...
Generating doc\index.html...
Generating doc\help-doc.html...
Generating doc\stylesheet.css...
```

——[$]一个使用{@link}的例子——

```
/**
 * @author Lang Yu
 * @see java.lang.String
 */
public class JavaDocLink {
   private int a;
   private int b;
   /**
    * {@link #getAdd() getAdd()} Method
    * @return the result of (a + b)
    **/
   public int getAdd(){
      return a + b;
   }
}
```

同样的方式生成对应的文档，就可以看到@link的效果了，讲到这里Java注释相关的内容就介绍了一半了，后边一般是与JavaDoc工具相关的，这个留到命令行工具里面去介绍。

iii.JDK和JRE的区别：

JDK（Java Development Kit的缩写），主要是指Java的开发环境和运行环境的一个集成开发包，这是面向开发人员的。

JRE（Java Runtime Enviroment的缩写），是指Java的运行环境，这一部分不包括开发环境，这是面向用户的。

JDK主要目的是为了开发、修改、调试、部署作为主体用途，因为有了JDK才能使用它提供的工具进行Java程序开发以及相关操作，这一开发包主要是针对开发人员的。JRE只是一个Java的运行环境，这里有一点需要说明的是，当一个程序开发完了需要在PC上运行的时候，该PC必须要提供一个JRE，也就是Java的运行环境，否则使用Java开发的软件是不能够直接运行在PC机上的，任何一个系统如果要运行Java的应用程序是必须依赖一个JRE的，有了这个运行时，才使得Java开发的软件可以无缝运行在操作系统上，而且对于像Tomcat这种服务器而言，JRE还远远不够，必须要JDK才能起到一定的作用，【*：这里提一下，Tomcat安装在搜索JDK的时候必须是JDK的路径，而不是JRE的路径，也就是说如果依赖Tomcat进行开发不能单纯配置一个JRE环境。】

其他相关知识参考一下网上的文档：

[1]为什么Sun要让JDK安装两套相同的JRE？这是因为JDK里面有很多用Java所编写的开发工具（如javac.exe、jar.exe等），而且都放置在 \lib\tools.jar 里。从下面例子可以看出，先将tools.jar改名为tools1.jar，然后运行javac.exe，显示如下结果： Exception in thread "main" java.lang.NoClassDefFoundError: com/sun/tools/javac /Main 这个意思是说，你输入javac.exe与输入 java -cp c:\jdk\lib\tools.jar com.sun.tools.javac.Main 是一样的，会得到相同的结果。从这里我们可以证明javac.exe只是一个包装器（Wrapper），而制作的目的是为了让开发者免于输入太长的指命。而且可以发现\lib目录下的程序都很小，不大于2 9K，从这里我们可以得出一个结论。就是JDK里的工具几乎是用Java所编写，所以也是Java应用程序，因此要使用JDK所附的工具来开发Java程序，也必须要自行附一套JRE才行，所以位于C:\Program Files\Java目录下的那套JRE就是用来运行一般Java程序用的。

[2]如果一台电脑安装两套以上的JRE，谁来决定呢？这个重大任务就落在java.exe身上。Java.exe的工作就是找到合适的JRE来运行Java程序。 Java.exe依照底下的顺序来查找JRE：自己的目录下有没有JRE；父目录有没有JRE；查询注册表： [HKEY_LOCAL_MACHINE\SOFTWARE\JavaSoft\Java Runtime Environment] 所以java.exe的运行结果与你的电脑里面哪个JRE被执行有很大的关系。

[3]介绍JVM JRE目录下的Bin目录有两个目录：server与client。这就是真正的jvm.dll所在。jvm.dll无法单独工作，当jvm.dll启动后，会使用explicit的方法（就是使用Win32 API之中的LoadLibrary()与GetProcAddress()来载入辅助用的动态链接库），而这些辅助用的动态链接库

（.dll）都必须位于jvm.dll所在目录的父目录之中。因此想使用哪个JVM，只需要设置PATH，指向JRE所在目录底下的jvm.dll。

iv.Java标识符：

标识符是指可被用来为类、变量或方法等命名的字符序列，换言之，标识符就是用户自定义的名称来标识类、变量或方法等。更简单的说，标识符就是一个名字。

标识符的选择并不是任意的，Java语言规定标识符由**字母、数字、下划线和美元符号（$）**组成，并且第一个字符不能是数字，例如以下都是合法的标识符：

num、user3、price$、book_name、MIN_VALUE

Java标识符中的字符是区分大小写的，如name和Name是两个不同的标识符。Java中所谓的字母并不只包含英文字母、数字及一些常用符号。Java语言使用的是Unicode标准字符集中的字符，Unicode字符集最多可以识别65536个字符，其前128个字符与ASCII码表中的字符对应。其余的字符中就包含了世界上大部分语言中的"字母表"中的字母，大部分国家"字母表"中的字母都是Unicode字符集中的一个字符，如汉字中的"礼"字就是Unicode字符集中的第31036个字符。因此，Java可使用的字符不仅可以是英文字母等，也可以是汉字、朝鲜文、俄文、希腊字母以及其他许多语言中的文字。

总结起来：

[1]Java标识符只能由数字、字母、下划线"_"或"$"符号以及Unicode字符集组成

[2]Java标识符必须以字母、下划线"_"或"$"符号以及Unicode字符集开头

[3]Java标识符不可以是Java关键字、保留字（const、goto）和字面量（true、false、null）

[4]Java标识符区分大小写，是**大小写敏感**的

合法标识符举例：

_哈哈、哈哈、_name、name、$dollor、$123、哈1哈

非法标识符举例：

1name、1Hello、 H%llo

——[$]标识符举例——

```
package org.susan.java.basic;

public class IdentityTester {
   public static void main(String args[]){
      // 因为中文是Unicode字符集，所以是可以作为标识符的
      String _哈哈 = "Hello";
      String 哈哈 = "You are very good";
      int _name = 0;
      int name = 0;
      float $dollor = 0.0f;
      String 哈1哈 = "Hello One Hello";
      //String 1Hello = "";
      //String H%llo = "";
   }
}
```

上边的代码段里面可以知道，合法标识符和非合法表示符的区别，这里特别需要注意的是中文汉字、朝鲜文、俄文、希腊字母以及其他文字，只要这些文字是属于Unicode字符集的，那么这些都是可以定义为标识符的，上边就是一个最简单的例子。

3.数据类型

数据类型在计算机语言里面，是对内存位置的一个抽象表达方式，可以理解为针对内存的一种抽象的表达方式。接触每种语言的时候，都会存在数据类型的认识，有复杂的、简单的。各种数据类型都需要在学习初期去了解，Java是强类型语言，所以Java对于数据类型的规范相对严格。数据类型是语言的抽象原子概念，可以说是语言中最基本的单元定义，在Java里面，本质上数据类型分为两种：简单类型和复杂类型。

简单类型：简单数据类型是不能简化的、内置的数据类型、由编程语言本身定义，它表示了真实的数字、字符和整数。

复杂类型：Java语言本身不支持C++中的结构（struct）或联合（union）数据类型，它的复合数据类型一般都是通过类或接口进行构造，类提供了捆绑数据和方法的方式，同时可以针对程序外部进行信息隐藏。

i.Java中的基本类型：

1)概念：

Java中的简单类型从概念上分为四种：实数、整数、字符、布尔值。但是有一点需要说明的是，Java里面有八种原始类型，其列表如下：

实数：double、float
整数：byte、short、int、long
字符：char
布尔值：boolean
复杂类型和基本类型的内存模型本质上不一样，简单数据类型的存储原理：所有的简单数据类型**不存在"引用"**概念，简单数据类型直接存储在内存中的内存栈上，数据本身的值存储在栈空间里面，而Java语言里面只有这八种数据类型是这种存储模型；而其他的只要是继承于Object类的复杂数据类型都是按照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。

2)原始类型特征：

Java的简单数据讲解列表如下：

int：int为整数类型，在存储的时候，用**4个字节**存储，范围为**-2,147,483,648到2,147,483,647**，在变量初始化的时候，int类型的默认值为**0**。

short：short也属于整数类型，在存储的时候，用**2个字节**存储，范围为**-32,768到32,767**，在变量初始化的时候，short类型的默认值为**0**，一般情况下，因为Java本身转型的原因，可以直接写为**0**。

long：long也属于整数类型，在存储的时候，用**8个字节**存储，范围为**-9,223,372,036,854,775,808到9,223,372,036, 854,775,807**，在变量初始化的时候，long类型的默认值为**0L或0l**，也可直接写为**0**。

byte：byte同样属于整数类型，在存储的时候，用**1个字节**来存储，范围为**-128到127**，在变量初始化的时候，byte类型的默认值也为**0**。

float：float属于实数类型，在存储的时候，用**4个字节**来存储，范围为**32位IEEEE 754单精度**范围，在变量初始化的时候，float的默认值为**0.0f或0.0F**，在初始化的时候可以写**0.0**。

double：double同样属于实数类型，在存储的时候，用**8个字节**来存储，范围为**64位IEEE 754双精度**范围，在变量初始化的时候，double的默认值为**0.0**。

char：char属于字符类型，在存储的时候用**2个字节**来存储，因为Java本身的字符集不是用ASCII码来进行存储，是使用的16位Unicode字符集，它的字符范围即是**Unicode的字符范围**，在变量初始化的时候，char类型的默认值为**'u0000'**。

boolean：boolean属于布尔类型，在存储的时候不使用字节，仅仅使用**1位**来存储，范围仅仅为**0和1**，其字面量为**true和false**，而boolean变量在初始化的时候变量的默认值为**false**。

——[$]提供一个字面量赋值的例子——

```
package org.susan.java.basic;

public class AssignTester {
   public static void main(String args[]){
      int x,y;//定义x，y变量
      float f = 12.34f; //定义float类型的变量并赋值
      double w = 1.234;//定义double类型变量并且赋值
      boolean flag = true ; //指定变量flag为boolean型，且赋初值为true
      char c ; //定义字符型变量c
      String str ; //定义字符串变量str
      String str1 = " Hi " ; //指定变量str1为String型，且赋初值为Hi
      c = 'A' ; //给字符型变量c赋值'A'
      str = " bye " ; //给字符串变量str赋值"bye"
      x = 12 ; //给整型变量x赋值为12
      y = 300; //给整型变量y赋值为300
   }
}
```

3)自动拆箱（AutoBox）：

Java里面，每一种原始类型都对应着相应的包装类型，在JDK1.5之前（不包含JDK1.5），当包装类和原始类型进行相互转换的时候，需要调用包装类型的方法进行转换，不能通过操作符进行直接的计算。下边是一个原始类型和包装类型的一个对应表：

原始类型	对应的包装类型	默认值	存储格式	数据范围
short	java.lang.**Short**	0	2个字节	-32,768到32767
int	java.lang.**Integer**	0	4个字节	-2,147,483,648到2,147,483,647
byte	java.lang.**Byte**	0	1个字节	-128到127

char	java.lang.**Character**	\u0000	2个字节	Unicode的字符范围
long	java.lang.**Long**	0L或0l	8个字节	-9,223,372,036,854,775,808到9,223,372,036,854,775,807
float	java.lang.**Float**	0.0F或0.0f	4个字节	32位IEEEE 754单精度范围
double	java.lang.**Double**	0.0或0.0D(d)	8个字节	64位IEEE 754双精度范围
boolean	java.lang.**Boolean**	false	1位	true(1)或false(0)

简单看看下边这段代码：

```
package org.susan.java.basic;

public class AutoBoxTester {
   public static void main(String args[]){
      Integer integer = new Integer(12);
      int integer2 = 33;
      System.out.println(integer + integer2);
   }
}
```

这段代码在JDK 1.5版本以上可以通过编译，而且不会报错，运行结果如下输出：

```
45
```

但是如果这段代码在JDK 1.4上边编译就会有问题了，因为在JDK 1.4的规范里面Integer属于一个包装类型，而int是原始类型，如果一个包装类型和原始类型要进行想对应的运算的时候，需要进行转换操作，直接将Integer类型转换称为原始类型操作，否则二者是不允许相加的，可以试试将上边代码用1.4版本进行编译：

```
javac -source 1.4 AutoBoxTester.java
```

就会收到下边的异常：

```
AutoBoxTester.java:5: operator + cannot be applied to java.lang.Integer,int
    System.out.println(integer + integer2);
```

为什么呢？其实编译器给的信息很明显，使用JDK 1.5进行编译可以直接通过而且不会报错，是因为JDK 1.5提供了自动拆箱和自动装箱的功能，而JDK 1.4里面如果要使得上边的代码段可以编译通过，必须做一个简单的修改：

```
public class AutoBoxTester {
   public static void main(String args[]){
      Integer integer = new Integer(12);
      int integer2 = 33;
      System.out.println(integer.intValue() + integer2);
   }
}
```

改成上边代码段了过后，在JDK 1.4平台下就可以得到输出：

```
45
```

从上边的例子可以看出，在JDK 1.5之前，如果要针对包装类进行数值计算，必须要将包装类直接转化称为原始类型，否则操作符本身是不会支持包装类的操作的，但是在JDK 1.5以及以后就没有这个限制了。

【简单总结：自动拆箱的意思就是不需要经过用户手工编程，编译器会直接识别包装类和原始类型相互之间的转换以及运算，并且把包装类型拆成原始类型进行代码里面规定的数值运算或者其他操作，这功能JDK的最低支持版本是1.5。其实对Java语言本身而言，Integer这种封装类实际上就是Java里面继承于Object的类的对象实例，只是在1.4之前，必须调用方法xxxValue()来完成手工拆箱的操作，只是这个在JDK 1.5不会有此限制。】

4)类型转换：

Java里面的类型转换包括两种：**自动转换（隐式转换）；强制转换（显示转换）**

[1]自动转换：

条件：**A.**这两种类型是兼容的；**B.**目的类型数的范围（位数）比来源类型的大

当以上2个条件都满足的时候，**拓宽转换（widening conversion）**就会自动发生，例如，int类型范围比所有byte类型的合法范围大，因此不要求显示的强制转换语句。对于拓宽转换，兼容程度可以看下边的继承树：

```
java.lang.Object
    |—java.lang.Boolean
```

|—java.lang.**Character**
|—java.lang.**Number**
|—java.lang.**Byte**
|—java.lang.**Float**
|—java.lang.**Integer**
|—java.lang.**Long**
|—java.lang.**Short**
|—java.lang.**Double**

从上边的继承树可以可以看到，**Boolean类型、Character类型、Number类型**是两两不兼容的，所以在隐式转换的过程，不兼容的类型是不能进行自动类型转换的

[2]强制转换：

尽管自动类型转换是很有帮助的，但并不能满足所有的编程需要。如果2种类型是兼容的，那么Java将自动地进行转换。例如，把int 类型的值赋给long 类型的变量，总是可行的。然而，不是所有的类型都是兼容的，因此，不是所有的类型转换都是可以隐式实现的。例如，没有将double 型转换为byte型的定义。幸好，获得不兼容的类型之间的转换仍然是可能的。要达到这个目的，你必须使用一个强制类型转换，它能完成两个不兼容的类型之间的显式变换。它的通用格式如下：

(target-type)value

[3]关于类型的自动提升，遵循下边的规则：

所有的byte、short、char类型的值将提升为int类型；

如果有一个操作数是long类型，计算结果是long类型；

如果有一个操作数是float类型，计算结果是float类型；

如果有一个操作数是double类型，计算结果是double类型；

自动类型转换图如下：

byte->short(char)->int->long->float->double

如果是强制转换的时候，就将上边的图反过来

[4]转换附加：

当两个类型进行自动转换的时候，需要满足条件：【1】这两种类型是兼容的，【2】目的类型的数值范围应该比源转换值的范围要大。（上边已经讲过了）而拓展范围就遵循上边的自动类型转换树，当这两个条件都满足的时候，拓展转换才会发生，而对于几个原始类型转换过程，根据兼容性boolean和char应该是独立的，而其他六种类型是可以兼容的，在强制转换过程，唯独可能特殊的是char和int是可以转换的，不过会使用char的ASCII码值比如：

int a = (int)'a';

a的值在转换过后输出的话，值为97；

[5]基础类型的几个常见的代码例子：

——[$]附加转换和精度丢失——

```
package org.susan.java.basic;

public class CastingNumbers {
   public static void main(String args[]){
      double above = 1.7;
      double below = 0.4;
      System.out.println("above:" + above);
      System.out.println("below:" + below);
      System.out.println("(int)above:" + (int)above);
      System.out.println("(int)below:" + (int)below);
      System.out.println("(char)('a' + above):" + (char)('a' + above));
      System.out.println("(char)('a' + below):" + (char)('a' + below));
   }
}
```

可以分析上边这段代码的输出：

```
above:1.7
below:0.4
(int)above:1
(int)below:0
(char)('a' + above):b
(char)('a' + below):a
```

从上边的输出结果可以知道，在double到int的强制转换中，会发现精度丢失的情况，而且这种精度丢失是不考虑四舍五入的，是直接将浮点部分舍弃掉，从上边的输出就可以知道了；而且当int以及

double转化称为char的时候，会使用上边提及的附加转换的结果，当'a'加了1过后进行char转换会生成b。

——[$]进制转换——

```
package org.susan.java.basic;
/**
 *进制相互转换的例子，以十进制为基础
 **/
public class SystemTester {
    public static void main(String args[]){
        //二进制和十进制相互转换
        int integer = 117;
        String binary = Integer.toBinaryString(integer);
        System.out.println("Binary value of " + integer + " is " + binary);
        String inputBinary = "100010011";
        int bResult = Integer.parseInt(inputBinary,2);
        System.out.println("Integer value of '" + inputBinary + "' is " + bResult);

        //十进制和八进制的相互转换
        int integer2 = 1024;
        String octal = Integer.toOctalString(integer2);
        System.out.println("Octal value of " + integer2 + " is " + octal);
        String inputOctal = "712364";
        int oResult = Integer.parseInt(inputOctal, 8);
        System.out.println("Integer value of " + inputOctal + " is " + oResult);
        //十进制和十六进制的相互转换
        int integer3 = 4096;
        String hex = Integer.toHexString(integer3);
        System.out.println("Hex value of " + integer3 + " is " + hex);
        String inputHex = "FEDD34";
        int hResult = Integer.parseInt(inputHex, 16);
        System.out.println("Integer value of " + inputHex + " is " + hResult);
    }
}
```

上边这段代码的输出为：

```
Binary value of 117 is 1110101
Integer value of '100010011' is 275
Octal value of 1024 is 2000
Integer value of 712364 is 234740
Hex value of 4096 is 1000
Integer value of FEDD34 is 16702772
```

——[$]字符类型例子——

```
package org.susan.java.basic;
/**
 *字符类型的例子
 **/
public class CharTester {
    public static void main(String args[]){
        //Char和整数之间的相互转换
        char ch1 = 'a';
        char ch2 = 66;
        System.out.println("---------------");
        System.out.println("ch1 is " + ch1);
        System.out.println("ch2 is " + ch2);

        //Character里面的方法
        System.out.println("---------------");
        System.out.println("isLowerCase：Z——" + Character.isLowerCase('Z'));
        System.out.println("isUpperCase：A——" + Character.isUpperCase('A'));
```

```
        System.out.println("isLetter : A——" + Character.isLetter('A'));
        System.out.println("isLetter : 2——" + Character.isLetter('3'));
        System.out.println("isLetterOrDigit : %——" + Character.isLetterOrDigit('%'));
        System.out.println("isDigit : %——" + Character.isDigit('%'));
        System.out.println("isDigit : 0——" + Character.isDigit('0'));
        System.out.println("isWhitespace: ' '——" + Character.isWhitespace(' '));
        System.out.println("isWhitespace: A——" + Character.isWhitespace('A'));

        //从字符串里面抽取Char
        System.out.println("---------------");
        String inputValue = "123456";
        System.out.println("Get 3 " + inputValue.charAt(2));
        System.out.println(containsOnlyNumbers("123456"));
        System.out.println(containsOnlyNumbers("123abc456"));
    }
    //检测某个字符串是否全部是数字
    private static boolean containsOnlyNumbers(String str){
        for(int i = 0; i < str.length(); i++ ){
            if(!Character.isDigit(str.charAt(i))){
                return false;
            }
        }
        return true;
    }
}
```

上边这段代码提取了Char类里面的一部分方法，其他的方法可以去查阅API的内容，输出为下：

```
---------------
ch1 is a
ch2 is B
---------------
isLowerCase: Z——false
isUpperCase: A——true
isLetter: A——true
isLetter: 2——false
isLetterOrDigit: %——false
isDigit: %——false
isDigit: 0——true
isWhitespace: ' '——true
isWhitespace: A——false
---------------
Get 3 3
true
false
```

Java的基本类型里面有几点需要说明：

[1]char类型是无符号16位整数，子面值必须用单引号括起来，如:'a'
[2]String在Java里面是类，直接父类是java.lang.**Object**，所以String不属于Java里面的原始类型
[3]长整数字有一个后缀为"L"或者"l"，八进制数字前缀为"0"，十六进制的前缀为"0x"
[4]默认的基本浮点类型为double
[5]float数据类型有一个后缀为"F"或"f"，Double数据类型后边可以跟"D"或者"d"，也可以不跟
[6]char类型可以使用通用的转义字符，但是不是ASCII码，应该是Unicode格式的如'\u0000'

5)Java的浮点精度：

[1]精确的浮点运算：

在Java里面，有时候为了保证数值的准确性需要精确的数据，先提供一个例子就可以发现问题了：

```
package org.susan.java.basic;

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

```
        System.out.println(0.05+0.01);
        System.out.println(1.0 - 0.42);
        System.out.println(4.015 * 100);
        System.out.println(123.3 / 100);
    }
}
```

按照我们的期待，上边应该是什么结果呢，但是看输出我们就会发现问题了：

```
0.060000000000000005
0.5800000000000001
401.49999999999994
1.2329999999999999
```

这样的话这个问题就相对严重了，如果我们使用123.3元交易，计算机却因为1.2329999999999999而拒绝了交易，岂不是和实际情况大相径庭

[2]四舍五入：

另外的一个计算问题，就是四舍五入。但是Java的计算本身是不能够支持四舍五入的，比如：

```
package org.susan.java.basic;

public class GetThrowTester {
    public static void main(String args[]){
        System.out.println(4.015 * 100.0);
    }
}
```

这个输出为：

```
401.49999999999994
```

所以就会发现这种情况并不能保证四舍五入，如果要四舍五入，只有一种方法java.text.**DecimalFormat**：

```
package org.susan.java.basic;

import java.text.DecimalFormat;

public class NumberFormatMain {
    public static void main(String args[]){
        System.out.println(new DecimalFormat("0.00").format(4.025));
        System.out.println(new DecimalFormat("0.00").format(4.024));
    }
}
```

上边代码输出为：

```
4.02
4.02
```

发现问题了么？因为DecimalFormat使用的舍入模式，关于DecimalFormat的舍入模式在格式化一章会讲到

[3]浮点输出：

Java浮点类型数值在大于9999999.0就自动转化成为科学计数法，看看下边的例子：

```
package org.susan.java.basic;

public class FloatCounter {
    public static void main(String args[]){
        System.out.println(9969999999.04);
        System.out.println(199999999.04);
        System.out.println(1000000011.01);
        System.out.println(9999999.04);
    }
}
```

输出结果为：

```
9.96999999904E9
1.9999999904E8
1.00000001101E9
9999999.04
```

但是有时候我们不需要科学计数法，而是转换成为字符串，所以这样可能会有点麻烦。

[4]BigInteger类和BigDecimal介绍：

BigInteger用法：

```
package org.susan.java.basic;

import java.math.BigInteger;

public class BigIntTester {
   public static void main(String args[]){
      BigInteger leftInteger = new BigInteger("1234567890123400");
      BigInteger rightInteger = BigInteger.valueOf(123L);
      //大数加法
      BigInteger resultInteger = leftInteger.add(rightInteger);
      System.out.println("Add:" + resultInteger);
      //大数除法
      resultInteger = leftInteger.divide(rightInteger);
      System.out.println("Divide:" + resultInteger);
      //大数减法
      resultInteger = leftInteger.subtract(rightInteger);
      System.out.println("Substract:" + resultInteger);
      //大数乘法
      resultInteger = leftInteger.multiply(rightInteger);
      System.out.println("Multiply:" + resultInteger);
   }
}
```

上边的输出为：

```
Add:1234567890123523
Divide:10037137318076
Substract:1234567890123277
Multiply:151851850485178200
```

在BigInteger里面有很多运算，这里只是提供了普通的+、-、*、/的运算，可以参考以下BigInteger的方法列表：

BigInteger abs()：返回BigInteger的绝对值
BigInteger add(BigInteger val)：返回为（this + val）的BigInteger，做加法
BigInteger and(BigInteger val)：返回为（this & val）的BigInteger，进行位与运算
BigInteger andNot(BigInteger val)：返回为（this & ~val）的BigInteger
int bitCount()：返回此BigInteger的二进制补码表示形式中与符号不同的位的常量
int bitLength()：返回此BigInteger的最小的二进制补码表示形式的位数，不包括符号位
BigInteger clearBit(int n)：返回其值与清除了指定位的此BigInteger等效的BigInteger
int compareTo(BigInteger val)：将此BigInteger与指定的BigInteger进行比较
BigInteger divide(BigInteger val)：返回值为（this/val）的BigInteger，做除法
BigInteger[] divideAndRemainder(BigInteger val)：包含返回（this/val）后跟（this & val）的两个BigInteger的数组
double doubleValue()：将此BigInteger转换为double
boolean equals(Object s)：比较此BigInteger与指定的Object的相等性
BigInteger flipBit(int n)：返回其值与对此BigInteger进行指定位翻转过后的值等效的BigInteger
float floatValue()：将此BigInteger转换为float
BigInteger gcd(BigInteger val)：返回一个BigInteger，值为abs(this)和abs(val)的最大公约数
int getLowestSetBit()：返回此 BigInteger 最右端（最低位）1 比特的索引（即从此字节的右端开始到本字节中最右端 1 比特之间的 0 比特的位数）
int hashCode()：返回此BigInteger的哈希码
int intValue()：将此BigInteger转换为int
boolean isProbablePrime(int certaintry)：如果此 BigInteger 可能为素数，则返回 true，如果它一定为合数，则返回 false
long longValue()：将此BigInteger转换为long
BigInteger max(BigInteger val)：返回this和val的最大值

BigInteger min(BigInteger val)：返回this和val的最小值
BigInteger mod(BigInteger val)：返回this mod val的结果，取模运算
BigInteger modInverse(BigInteger val)：返回其值为 $(this^{-1} \bmod val)$ 的 BigInteger
BigInteger modPow(BigInteger exponent,BigInteger val)：返回其值为 $(this^{exponent} \bmod val)$ 的 BigInteger
BigInteger multiply(BigInteger val)：返回值为（this*val）的BigInteger
BigInteger negate()：返回值是（-this）的BigInteger
BigInteger nextProbablePrime()：返回大于此BigInteger的可能为素数的第一个整数
BigInteger not()：返回其值为（~this）的BigInteger
BigInteger or(BitInteger val)：返回值为（this|val）的BigInteger
BigInteger pow(int exponent)：返回其值为 $(this^{exponent})$ 的 BigInteger
static BigInteger probablePrime(int bitLength,Random rnd)：返回有可能是素数的、具有指定长度的正 BigInteger
BigInteger remainder(BigInteger val)：返回其值为 (this % val) 的 BigInteger
BigInteger setBit(int n)：返回其值与设置了指定位的此 BigInteger 等效的 BigInteger
BigInteger shiftLeft(int n)：返回其值为 (this << n) 的 BigInteger
BigInteger shiftRight(int n)：返回其值为 (this >> n) 的 BigInteger
int signum()：返回此 BigInteger 的正负号函数
BigInteger substract(BigInteger val)：返回其值为 (this - val) 的 BigInteger
BigInteger xor(BigInteger val)：返回其值为 (this ^ val) 的 BigInteger
BigDecimal用法：
BigDecimal是Java提供的一个不变的、任意精度的有符号十进制数对象。它提供了四个构造器，有两个是用BigInteger构造，在这里我们不关心，我们重点看用double和String构造的两个构造器：
BigDecimal(double val)：把一个double类型十进制数构造为一个BigDecimal对象实例
BigDecimal(String val)：把一个以String表示的BigDecimal对象构造为BigDecimal对象实例
用一段代码来说明两个构造函数的区别：

```
package org.susan.java.basic;

import java.math.BigDecimal;

public class BigDecimalMain {
   public static void main(String args[]){
      System.out.println(new BigDecimal(123456789.01).toString());
      System.out.println(new BigDecimal("123456789.01").toString());
   }
}
```

看了输出结果过后，两种构造函数的区别一目了然：

```
123456789.010000005364418029785156 25
123456789.01
```

BigDecimal舍入模式介绍：
舍入模式在java.math.**RoundingMode**里面：
RoundingMode.*CEILING*：向正无限大方向舍入的舍入模式。如果结果为正，则舍入行为类似于 RoundingMode.UP；如果结果为负，则舍入行为类似于 RoundingMode.DOWN。注意，此舍入模式始终不会减少计算值

输入数字	使用CEILING舍入模式将数字舍入为一位数
5.5	6
2.5	3
1.1	2
1.0	1
-1.0	-1
-1.1	-1
-1.6	-1
-2.5	-2

-5.5	-5

RoundingMode.*DOWN*：向零方向舍入的舍入模式。从不对舍弃部分前面的数字加 1（即截尾）。注意，此舍入模式始终不会增加计算值的绝对值

输入数字	使用**DOWN**舍入模式将数字舍入为一位数
5.5	5
2.5	2
1.1	1
-1.0	-1
-1.6	-1
-2.5	-2
-5.5	-5

RoundingMode.*FLOOR*：向负无限大方向舍入的舍入模式。如果结果为正，则舍入行为类似于 RoundingMode.DOWN；如果结果为负，则舍入行为类似于 RoundingMode.UP。注意，此舍入模式始终不会增加计算值

输入数字	使用**FLOOR**舍入模式将输入数字舍入为一位
5.5	5
2.3	2
1.6	1
1.0	1
-1.1	-2
-2.5	-3
-5.5	-6

RoundingMode.*HALF_DOWN*：向最接近数字方向舍入的舍入模式，如果与两个相邻数字的距离相等，则向下舍入。如果被舍弃部分 > 0.5，则舍入行为同 RoundingMode.UP；否则舍入行为同 RoundingMode.DOWN

输入数字	使用**HALF_DOWN**输入模式舍入为一位
5.5	5
2.5	2
1.6	2
1.0	1
-1.1	-1
-1.6	-2
-2.5	-2
-5.5	-5

RoundingMode.*HALF_EVEN*：向最接近数字方向舍入的舍入模式，如果与两个相邻数字的距离相等，则向相邻的偶数舍入。如果舍弃部分左边的数字为奇数，则舍入行为同 RoundingMode.HALF_UP；如果为偶数，则舍入行为同 RoundingMode.HALF_DOWN。注意，在重复进行一系列计算时，此舍入模式可以在统计上将累加错误减到最小。此舍入模式也称为"银行家舍入法"，主要在美国使用。此舍入模式类似于 Java 中对 float 和 double 算法使用的舍入策略

输入数字	使用**HALF_EVEN**舍入模式将输入舍为一位
5.5	6
2.5	2
1.6	2
1.1	1

-1.0	-1
-1.6	-2
-2.5	-2
-5.5	-6

RoundingMode.*HALF_UP*：向最接近数字方向舍入的舍入模式，如果与两个相邻数字的距离相等，则向上舍入。如果被舍弃部分 >= 0.5，则舍入行为同 RoundingMode.UP；否则舍入行为同 RoundingMode.DOWN。注意，此舍入模式就是通常学校里讲的四舍五入

输入数字	使用**HALF_UP**舍入模式舍入为一位数
5.5	6
2.5	3
1.6	2
1.0	1
-1.1	-1
-1.6	-2
-2.5	-3
-5.5	-6

RoundingMode.*UNNECESSARY*：用于断言请求的操作具有精确结果的舍入模式，因此不需要舍入。如果对生成精确结果的操作指定此舍入模式，则抛出 ArithmeticException

输入数字	使用**UNNECESSARY**模式
5.5	抛出 ArithmeticException
2.5	抛出 ArithmeticException
1.6	抛出 ArithmeticException
1.0	1
-1.0	-1.0
-1.1	抛出 ArithmeticException
-1.6	抛出 ArithmeticException
-2.5	抛出 ArithmeticException
-5.5	抛出 ArithmeticException

RoundingMode.*UP*：远离零方向舍入的舍入模式。始终对非零舍弃部分前面的数字加 1。注意，此舍入模式始终不会减少计算值的绝对值

输入数字	使用UP舍入模式将输入数字舍入为一位数
5.5	6
1.6	2
1.1	2
1.0	1
-1.1	-2
-1.6	-2
-2.5	-3
-5.4	-6

——[$]示例代码：——

```
package org.susan.java.basic;

import java.math.BigDecimal;
```

```
import java.text.DecimalFormat;
/**
 *使用舍入模式的格式化操作
 **/
public class DoubleFormat {
    public static void main(String args[]){
        DoubleFormat format = new DoubleFormat();
        System.out.println(format.doubleOutPut(12.345, 2));
        System.out.println(format.roundNumber(12.335, 2));
    }
    public String doubleOutPut(double v,Integer num){
        if( v == Double.valueOf(v).intValue()){
            return Double.valueOf(v).intValue() + "";
        }else{
            BigDecimal b = new BigDecimal(Double.toString(v));
            return b.setScale(num,BigDecimal.ROUND_HALF_UP).toString();
        }
    }
    public String roundNumber(double v,int num){
        String fmtString = "0000000000000000"; //16bit
        fmtString = num>0 ? "0." + fmtString.substring(0,num):"0";
        DecimalFormat dFormat = new DecimalFormat(fmtString);
        return dFormat.format(v);
    }
}
```

这段代码的输出为：

```
12.35
12.34
```

ii.数组、复杂类型

1)数组的定义、初始化：

在Java语言里面使用下边的方式定义数组：

```
Type[] arrayone;
Type arrayone[];
```

这里的Type可以是基本数据类型，也可以是Object类的子类的引用，arrayone在这里为一个Java合法的标识符

——[$]这里提供一个参考代码进行数组的定义和初始化——

```
package org.susan.java.basic;

import java.util.Arrays;
/**
 *数组的定义和初始化过程
 **/
public class ArraysMain {
    public static void main(String args[]){
        String[] arrays1 = new String[3];
        String[] arrays2 = new String[]{"A","B","C"};
        String[] arrays3 = {"A","B","C","D"};
        System.out.println("Array 1:" + Arrays.asList(arrays1));
        System.out.println("Array 2:" + Arrays.asList(arrays2));
        System.out.println("Array 3:" + Arrays.asList(arrays3));
    }
}
```

先看上边代码段的输出：

```
Array 1:[null, null, null]
Array 2:[A, B, C]
Array 3:[A, B, C, D]
```

这里简单总结一下Java里面定义、创建、初始化数组的一些细节问题：

- 定义数组一般用**Type[] arrays**或者**Type arrays[]**两种格式，Java里面推荐使用Type[] arrays这种方式定义数组
- 创建数组的时候可以使用三种方式创建，上边的arrays1创建了一个长度为3的字符串数组，但是未被初始化
- 初始化过程有两种方式，上边arrays2和arrays3是直接赋值，这种情况不仅仅创建了数组，而且进行了初始化的操作，还有一种方式像下边这样针对每个元素初始化

——[$]单项初始化——

```
package org.susan.java.basic;

import java.util.Arrays;

public class ArrayItemMain {
    public static void main(String args[]){
        String[] arrayStrings = new String[3];
        for(int i = 0; i < arrayStrings.length; i++){
            arrayStrings[i] = new String("String" + i);
        }
        System.out.println(Arrays.asList(arrayStrings));
    }
}
```

上边的输出为：

```
[String0, String1, String2]
```

由此可以知道，在这样的循环里面已经将定义的长度为3的字符串数组的每一项都初始化了，而且值分别为String0,String1,String2，而且从上边可以知道，数组的元素访问使用索引进行访问，格式为：

arrays[index]，这里需要特殊说明的是索引的范围：**0 <= index < arrays.length**；关于索引的范围需要特别注意，**最小的index是0，最大的index值是length-1**

2)多维数组：

Java里面不仅仅可以定义一纬数组，还可以定义多维数组，多维数组的定义如下：

```
int[][] arrays1 = new int[3][2];
int[][] arrays2 = new int[3][];
```

不合法的定义方式为：

```
int[][] arrays3 = new int[][2];
```

【*：这里定义的多维数组为两维数组，不仅仅如此，根据需要可以定义三维数组或者多维数组，其实在Java里面，二维数组就是数组的数组，也就是说在一纬数组的基础之上每一项又保存了一个数组的引用。】

——[$]提供一个多维数组的例子——

```
package org.susan.java.basic;

import java.util.Arrays;
import java.util.Random;

public class ArrayMultiMain {
    public static void main(String args[]){
        // 二维规则数组
        int[][] arrays = new int[3][3];
        Random random = new Random();
        for( int i = 0; i < arrays.length; i++){
            for( int j = 0; j < arrays[i].length; j++){
                arrays[i][j] = random.nextInt(40);
            }
        }
        printArray(arrays);
        System.out.println("------------------------");
        // 二维不规则数组
        int[][] arrays1 = new int[3][];
        for( int i = 0; i < arrays1.length; i++){
            int innerLength = random.nextInt(6) + 1;
```

```
            arrays1[i] = new int[innerLength];
            for( int j = 0; j < arrays1[i].length; j++){
                arrays1[i][j] = random.nextInt(40);
            }
        }
        //System.out.println(Arrays.asList(arrays1))
        printArray(arrays1);
        System.out.println("-----------------------");
        int[][] arrays2 = {{12,33},{45,44,46,47},{56,12,9}};
        printArray(arrays2);
    }
    private static void printArray(int [][] arrays){
        for(int[] array:arrays){
            for(int item:array){
                System.out.print("[" + item + "],");
            }
            System.out.println();
        }
    }
}
```

这里运行的输出为：

```
[8],[16],[4],
[36],[2],[34],
[20],[35],[37],
-----------------------
[4],
[8],[38],[37],[7],
[18],[17],[36],
-----------------------
[12],[33],
[45],[44],[46],[47],
[56],[12],[9],
```

针对**输出内容**需要**说明**的有几点：

- 第一个数组的行列是定死的，但是里面的数据是随机的，所以每次运行可能里面盛放的项不一样
- 第二个数组为不规则的二维数组，行市定死的，但是每行的列数十不定的，每次运行可能都不一样
- 第三个数组在创建和初始化的时候因为是直接赋值，是一个死数组，所以第三那个数组每次输出都一样

3)Java中的动态数组和静态数组：【摘录自：http://developer.51cto.com/art/200905/122774.htm】

我们学习的数组都是静态数组，其实在很多的时候，静态数组根本不能满足我们编程的实际需要，比方说我需要在程序运行过程中动态的向数组中添加数据，这时我们的静态数组大小是固定的，显然就不能添加数据，要动态添加数据必须要用到动态数组，动态数组中的各个元素类型也是一致的，不过这种类型已经是用一个非常大的类型来揽括—Object类型。Object类是java.lang包中的顶层超类。所有的类型都可以与Object类型兼容，所以我们可以将任何Object类型添加至属于Object类型的数组中，能添加Object类型的的集合有**ArrayList、Vector**及**LinkedList**，它们对数据的存放形式仿造于数组，属于集合类，下面是他们的特点：

特点一、容量扩充性

从内部实现机制来讲ArrayList和Vector都是使用Objec的数组形式来存储的。当你向这两种类型中增加元素的时候，如果元素的数目超出了内部数组目前的长度它们都需要扩展内部数组的长度，Vector缺省情况下自动增长原来一倍的数组长度，ArrayList是原来的50%,所以最后你获得的这个集合所占的空间总是比你实际需要的要大。所以如果你要在集合中保存大量的数据那么使用Vector有一些优势，因为你可以通过设置集合的初始化大小来避免不必要的资源开销。

特点二、同步性

ArrayList,LinkedList是不同步的，而Vestor是的。所以如果要求线程安全的话，可以使用ArrayList或LinkedList，可以节省为同步而耗费开销。但在多线程的情况下，有时候就不得不使用

Vector了。当然，也可以通过一些办法包装ArrayList,LinkedList，使他们也达到同步，但效率可能会有所降低。

特点三、数据操作效率

ArrayList和Vector中，从指定的位置(用index)检索一个对象，或在集合的末尾插入、删除一个对象的时间是一样的，可表示为O(1)。但是，如果在集合的其他位置增加或移除元素那么花费的时间会呈线形增长：O(n-i)，其中n代表集合中元素的个数，i代表元素增加或移除元素的索引位置。为什么会这样呢?以为在进行上述操作的时候集合中第i和第i个元素之后的所有元素都要执行(n-i)个对象的位移操作。LinkedList中，在插入、删除集合中任何位置的元素所花费的时间都是一样的—O(1)，但它在索引一个元素的时候比较慢，为O(i),其中i是索引的位置。所以，如果只是查找特定位置的元素或只在集合的末端增加、移除元素，那么使用Vector或ArrayList都可以。如果是对其它指定位置的插入、删除操作，最好选择LinkedList、ArrayList 和Vector是采用数组方式存储数据，此数组元素数大于实际存储的数据以便增加和插入元素，都允许直接序号索引元素，但是插入数据要设计到数组元素移动等内存操作，所以索引数据快插入数据慢，Vector由于使用了synchronized方法(线程安全)所以性能上比ArrayList要差，LinkedList使用双向链表实现存储，按序号索引数据需要进行向前或向后遍历，但是插入数据时只需要记录本项的前后项即可，所以插入数度较快。

4.操作符

操作	优先级	结合
后缀运算符	[] . () 函数调用	从左
单目运算符	! ~ ++ -- +(单操作符) -(单操作符)	从右
创建	new	从左
乘除	* / %	从左
加减	+ -	从左
移位	<< >> >>>	从左
关系	< <= > >= instanceof	从左
相等	== !=	从左
按位与	&	从左
按位异或	^	从左
按位或	|	从左
逻辑与	&&	从左
逻辑或	||	从左
条件	? :	从右
赋值	= += -= *= /= %= ^= <<= >>= >>>=	从右

几乎所有运算符都只能操作“基本类型”（Primitives），只是=、==、!=可以操作所有对象，除此String类型可以支持+、+=操作符：

i.赋值操作符：

一般情况下基本类型可以直接使用=进行赋值操作，看简单的一份代码

```
package org.susan.java.basic;
/**
 *赋值操作符的使用
 **/
public class AssignMain {
    public static void main(String args[]){
        int i = 5;
        int j = 10;
        i += 5;
        j -= 2;
        System.out.println("i = " + i);
        System.out.println("j = " + j);
    }
```

```
}
```

这段代码的输出为：

```
i = 10
j = 8
```

赋值操作符一般使用=或者带操作的=进行，带操作的等号比如上边使用的+=、-=等。针对数值操作符而言：

i += b; 等价于 i = i + b;

所以可以理解上边的两句话：

```
i += 5;
j -= 2;
```

这种操作符就是带操作的赋值，这里需要说明的主要是赋值操作符=

[1]针对**原始类型**的=操作符而言，比如a=b【a和b为两个变量】，在赋值的过程里面，这句话进行了下边的操作：JVM在内存里面为b保存一个栈地址用来存储真实的b的值，然后在赋值过程中，JVM创建一个**b的副本**，然后把b的副本的值赋值到a里面，也就是说在这种情况下如果**a的值发生了变化的话，b本身的值是不会受到影响**的，而b在赋值过程的副本也是瞬间的，当a一旦接受到b的值过后，b的副本就从内存里面清除掉，就使得a和b各自有了自己的栈地址

[2]如果是**非原始类型**：比如有两个对象的引用a和b，如果进行了a=b的操作，这句话就进行了下边的操作：JVM会**拷贝一个b的引用**，然后把引用副本赋值到a，使得a引用和b引用指向同一个对象，**当a引用的对象发生改变的时候，b引用指向的对象也会发生同样的改变**

——[$]一个赋值的概念说明例子——

```
package org.susan.java.basic;
/**
 *针对赋值符号的概念说明代码，主要是区分上边的不同类型进行赋值操作
 **/
public class AssignObject {
   public static void main(String args[]){
      // Primitives类型
      System.out.println("--------------------");
      int bNumber = 1;
      int aNumber = bNumber;
      System.out.println("Before assign aNumber is " + aNumber);
      System.out.println("Before assign bNumber is " + bNumber);
      aNumber = 12;
      System.out.println("After assign aNumber is " +aNumber);
      System.out.println("After assign bNumber is " + bNumber);

      // Reference类型
      System.out.println("--------------------");
      StringBuffer bBuffer = new StringBuffer("b");
      StringBuffer aBuffer = bBuffer;
      System.out.println("Before assign aBuffer is " + aBuffer);
      System.out.println("Before assgin bBuffer is " + bBuffer);
      aBuffer.append(" assign a");
      System.out.println("Before assign aBuffer is " + aBuffer);
      System.out.println("Before assgin bBuffer is " + bBuffer);
   }
}
```

上边程序的输出为：

```
--------------------
Before assign aNumber is 1
Before assign bNumber is 1
After assign aNumber is 12
After assign bNumber is 1
--------------------
Before assign aBuffer is b
Before assgin bBuffer is b
Before assign aBuffer is b assign a
Before assgin bBuffer is b assign a
```

【*:上边的程序输出刚好说明了上边的两点内容，这里有一点需要注意的是，在使用引用赋值的时候，所说的修改引用指代对象的内容的时候，这里不能使用String类型，因为使用了String类型就会出现很特殊的情况，这一点在String的说明里面我会特别讲述，需要重视的一点是：修改对象内容。这里针对StringBuffer而言，append方法就修改了StringBuffer对象的内容，所以这个程序这样说明是没有问题的，这里再次提醒一下：String类型的对象属于不可变对象，在针对不可变对象的时候，修改对象内容的原理和可变对象修改对象内容的原理不一样的，所以这里不能使用String，演示的代码使用的是StringBuffer。】

简单总结：

- **leftVarl op= rightVal**格式的表达式等同于**leftVal = leftVal op rightVal**
- 原始类型赋值操作和引用类型的赋值操作有本质的区别
- 可变对象和不可变对象对内容的更改本质也不一样的

ii.算术运算符：

Java的算数运算符：+、-、/、*、%，优先级【仅针对算术运算符】：

[1]/、*、%是第一优先级的，这三个操作符的运算是从左到右

[2]+、-操作符是第二优先级的，其运算也是从左到右

——[$]浮点取模、整数取模——

```
package org.susan.java.basic;
/**
 *普通运算的优先级，整数取模和浮点取模
 **/
public class NumberOpTester {
   public static void main(String args[]){
      int a = 2;
      int b = 4;
      int c = 3;
      int result = c + b / a;
      System.out.println("c + b / a = " + result);
      result = c * b / a;
      System.out.println("c * b / a = " + result);

      double b1 = 4.5f;
      double a1 = 2;
      double result1 = b1 % a1;
      System.out.println(" b % c = " + (b % c));
      System.out.println(" b1 % a1 = " + result1);
   }
}
```

上边的输出为：

```
c + b / a = 5
c * b / a = 6
b % c = 1
b1 % a1 = 0.5
```

根据上边的运算符，需要说明一点：

在**Java运算符**里面，不仅仅像C++一样支持整数取模，而且还支持**浮点类型的取模**，就像上边的最后一个表达式result1 = b1 % a1，就是进行的浮点取模操作；而运算符的优先级，上边代码已经一目了然了这里就不重复了

iii.自动递增、递减运算：

Java和C++一样存在前递增和前递减（++A和--A），这种情况先进行运算，再生成值；后递增和后递减（A++和A--），会先生成值，再执行运算；因为++和--操作一样，所以提供一段说明代码：

```
package org.susan.java.basic;
/**
 *自动递增递减的代码示例
 **/
public class AutoIncreasement {
   public static void main(String args[]){
      int i = 3;
      int j = 3;
```

```
        System.out.println("++i:" + (++i));
        System.out.println("j++:" + (j++));
        System.out.println("i:" + i);
        System.out.println("j:" + j);

        System.out.println(i==(i++));
        System.out.println(i==(++i));

        float floatValue = 0.3F;
        System.out.println(floatValue++);
        System.out.println(floatValue);
    }
}
```

上边代码的输出为：

```
++i:4
j++:3
i:4
j:4
true
false
0.3
1.3
```

需要说明的是：**A++**和**A = A + 1**是等效的，**++A**的最终执行效果和 **A = A + 1**也是等效的，二者只有一个最简单的区别，上边代码可以看出来

[1]前递增（递减）和后递增（递减）就是上边表达的那种情况，但是有一点特殊就是前递增（递减）是**先改变变量**的值，而后递增（递减）是在进行运算过**后修改变量**的值

[2]而且在Java里面，自动递增递减是**支持浮点类型**的，最后那一些代码可以说明这一点

注意：可以这样来理解，i,i++,++i代表三个不同的变量，如果i的值是a，那么i++本身的值是在递增之前返回的，i++的判断是和i一样的，而++i本身的值是在递增之后返回的，++i和i的判断却是不一样的，所以这里应该理解的是变量是**什么时候**进行更改的。主要参考上边的代码：

```
System.out.println(i==(i++));
System.out.println(i==(++i));
```

根据这段代码的输出就可以知道（更改变量的时间）在自增自减的运算中的重要性了，提供一个更加迷惑的例子：

```
package org.susan.java.basic;
/**
 *一个比较迷惑的自增自减的例子
 **/
public class AutoIncreaseMain {
    public static void main(String args[]){
        int i = 4;
        int j = 6;
        int result = i+++ ++j + i+++ ++i + i + ++i;
        System.out.println("result:" + result);
        int a = 5;
        int result1 = a+++a+++a+++a++;
        System.out.println("result1:" + result1);
    }
}
```

先看输出：

```
result:38
result1:26
```

仔细分析一下上边的输出：

在分析过程模拟一个堆栈，把每次的结果都压入堆栈进行运算：

第一个表达式result的结果：

[1]最先压入栈的是i++，这个地方i的值为4，那么i++返回值就是4，但是运算过后i的值变成了5，**结果为：4 +**

[2]然后压入栈的是++j，这个地方j的值为6，那么++j返回值就是7，但是运算过后同样使得j变成了7，**结果为：4 + 7 +**
[3]然后压入栈的同样是i++，但是这个时候i的值为5，那么i++也是5，运算过后i的值变成了6，**结果为：4 + 7 + 5 +**
[4]按照这种方式运算下去，最终的**结果是：4 + 7 + 5 + 7 + 7 + 8**，所以result的值为**38**
按照同样的方式可以运算result1的结果：result1的最终结果为26；至于这种操作的表达式写法有一定的规则，下边的是合法和不合法的例子：
合法的写法：

```
i+++ ++i;
i+++i++;
i+ ++i;
i+++i;
i++ +(++i);
```

不合法的写法：

```
i+++++i;
i++ +++i;
```

至于其他的写法对与不对可以直接在Eclipse里面进行编译来判断表达式的正确与否

iv.关系运算符：

关系运算符包括<、>、<=、>=、==、!=：
关于关系运算符需要说明的是：

- ==和!=是适合所有的基本类型（Primitives）的，但是其他关系运算符不能比较boolean
- 如果是针对两个引用指向的对象内容进行比较，必须用特殊的操作方法equals()
- **==和!=在比较两个引用**的时候，比较的是引用的指向，而不是对象的内容信息

——[$]仅提供一个例子说明——

```
package org.susan.java.basic;

public class CompareMain {
   public static void main(String args[]){
      int i = 4;
      System.out.println((i==4));
      System.out.println((i!=4));
      System.out.println((i >= 3));
      System.out.println((i < 4));
      System.out.println((i > 6));
      System.out.println((i <= 4));
   }
}
```

上边的输出为：

```
true
false
true
false
false
true
```

v.逻辑运算符和按位运算符：

逻辑运算符有**&&、||、!**能够结合表达式生成一个boolean返回值
按位运算符**AND(&)、OR(|)、XOR(^)、NOT(~)**运算
下边是所有的逻辑关系表：
非关系表：

A	!A
true	false
false	true

逻辑与关系：

A	B	A && B
false	false	false

true	false	false
false	true	false
true	true	true

逻辑或关系：

A	B	A || B
false	false	false
true	false	true
false	true	true
true	true	true

下边是按位运算关系表：

非位运算符：

A	~A
1	0
0	1

与位运算符：

A	B	A & B
1	1	1
1	0	0
0	1	0
0	0	0

或位运算符：

A	B	A | B
1	1	1
1	0	1
0	1	1
0	0	0

异或运算符：

A	B	A ^ B
1	1	0
1	0	1
0	1	1
0	0	0

关于逻辑运算符一定要理解的是**"短路"**现象，比如下边的表达式：

```
if( 1==1 && 1==2 && 1==3)
```

代码从左到右开始执行，指定第一个表达式1==1是为true，因为是为true，然后就开始运算1==2，运算了第二个表达式过后结果如下：

true && false && 1==3

到了这个时候就不再运算下去了，因为在这种运算里面第二个表达式为false，这种情况就短路了，因为**不论最后一个表达式为什么，最终返回就已经是false**，短路现象在&&和||中都是常见的，只有逻辑关系运算符存在短路现象，位逻辑运算符中是不存在短路现象的

——[$]提供一个运算的例子——

```
package org.susan.java.basic;

public class ShortTruFalseTester {
    public static void main(String args[]){
        boolean a = true;
```

```
        boolean b = false;
        System.out.println(!a);
        System.out.println(a && b);
        System.out.println(a || b);

        int aNumber = 15;
        int bNumber = 5;
        System.out.println(aNumber & bNumber);
        System.out.println(aNumber | bNumber);
        System.out.println(aNumber ^ bNumber);
        System.out.println(~aNumber);
    }
}
```

输出为：

```
false
false
true
5
15
10
-16
```

逻辑运算符这里不做任何解释，但是另外三个计算机底层的位运算符是需要简单解释一下：

aNumber=15中，aNumber转换成为二进制位为：**1111**

bNumber=5中，bNumber转换成为二进制位为：**0101**

1111&0101的结果每一位进行&运算可以得到**1111&0101=0101**，运算规则就为每一位1&1=1,1&0和0&1为0，0&0=0，最后结果转换为**0101=5**

1111|0101的结果是每一位进行|运算**1111|0101=1111**，运算规则为1|1=1，1|0=1和0|1=1，0|0=0，最后结果为**1111=15**

1111^0101的结果是每一位进行^运算**1111^0101=1010**，运算规则为1^1和0^0为0，1^0和0^1为1，最后结果为**1010=10**

~aNumber在这个地方相对复杂：

其实在32bit系统中，aNumber的真实数值为：

0000 0000 0000 0000 0000 0000 0000 1111

按位运算结果为：

1111 1111 1111 1111 1111 1111 1111 0000

转化成为数值就为：-16

vi.三元操作符：

三目运算符是一个特殊的运算符，它的语法形式如下：

布尔表达式?表达式1:表达式2

运算过程，如果布尔表达式的值为true就返回表达式1的值，如果为false返回表达式2的值，简单用一个例子说明：

```
package org.susan.java.basic;
/**
 *三元操作符
 **/
public class ThirdMain {
    public static void main(String args[]){
        int sum = 90;
        String resultValue = (sum > 90)?"Bigger than 90":"Smaller than 90";
        String resultValue1 = (sum < 100)?"Smaller than 100":"Smaller than 100";
        System.out.println(resultValue);
        System.out.println(resultValue1);
    }
}
```

输出结果为：

```
Smaller than 90
Smaller than 100
```

vii.移位运算符：

移位运算符操作的对象是二进制位，可以单独用移位运算符来处理int类型的整数，下边是概念表：

运算符	含义	例子
<<	左移运算符，将运算符左边的对象向左异动运算符右边指定的位（在低位补0）	x << 3
>>	"有符号"右移运算 符，将运算符左边的对象向右移动运算符右边指定的位数。使用符号扩展机制，也就是说，如果值为正，则在高位补0，如果值为负，则在高位补1。	x >> 3
>>>	"无符号"右移运算 符，将运算符左边的对象向右移动运算符右边指定的位数。采用0扩展机制，也就是说，无论值的正负，都在高位补0。	x >>> 3

1)计算机里面的原码、反码、补码

在计算机里面所有的数据都是按照字节来存储的，1个字节等于8位（1Byte=8bit），而计算机**只能识别0和1**，所以根据排列，1个字节可以存储256种不同的信息，就是2^8(0和1两种可能，8位排列)，比如定义一个字节大小的无符号整数，那么表示的就是0~255(0~2^8-1)，这些数一共是256个数。这里只是标识了一个无符号的整数，那么需要进一步剖析这个数，0是这些数字中最小的一个：00000000，在计算机里面，无符号整数就是这样的方式来进行整数的存储，也就是说如果知道了一个无符号整数的二进制码就可以直接计算该整数的值了。而计算机里面有两外一种存储整数的方式，这种方式是针对有符号的整数而言的。

只有有符号的整数才有原码、反码和补码，其他的类型一概都没有，而且不需要。虽然我们也可以用二进制中最小的数去对应最小的负数，最大的也对应最大的，但是那样不科学，计算机本身却不使用这样的方式，二是有另外的存储方式。如果1个字节不管怎样还是只能标识256位，那么带符号的整数就应该是：**-128~127**，这种情况在计算机里面怎么存储呢？其实该字节的最高位是符号位，比如：

0 1111111

在上边的8位里面，最高位是0，代表的是符号位，在计算机里面0代表+，1代表-，这样的存储就可以知道实际只有255个数。那么计算机的存储原理是什么呢？

可以这样理解，用最高位表示符号位，**如果是0表示正数，如果是1表示负数**，剩下的7位用来储存数的绝对值的话，能表示2^7个数的绝对值，再考虑正负两种情况，2^7*2还是256个数。首先定义0在计算机中储存为00000000，对于正数我们依然可以像无符号数那样换算，从00000001到01111111依次表示1到127。那么这些数对应的二进制码就是这些数的原码。到这里很多人就会想，那负数是不是从10000001到11111111依次表示-1到-127，那你发现没有，如果这样的话那么一共就只有255个数了，因为10000000的情况没有考虑在内。实际上，10000000在计算机中表示最小的负整数，就是这里的-128，而且实际上并不是从10000001到11111111依次表示-1到-127，而是刚好相反的，从10000001到11111111依次表示-127到-1。负整数在计算机中是以补码形式储存的，补码是怎么样表示的呢，这里还要引入另一个概念——反码，所谓反码就是把负数的原码除符号位（**负数的原码除符号位和它的绝对值所对应的原码相同，简单的说就是绝对值相同的数原码相同**）各个位按位取反，是1就换成0，是0就换成1，如-1的原码是0000001（**注意这里只有7位，不看符号位，我这里所说的负数符号位都是1**），和1的原码相同，那么-1的反码就是1111110（**这也是7位，后面加上了符号位都是8位了**），而补码就是在反码的基础上加1，即-1的补码是11111110+1=11111111，因此我们可以算出-1在计算机中是按11111111储存的。**总结一下，计算机储存有符号的整数时，是用该整数的补码进行储存的，0的原码、补码都是0，正数的原码、补码可以特殊理解为相同，负数的补码是它的反码加1。**

举个简单的例子：

有符号的整数	原码	反码	补码
47	00101111	00101111	00101111（正数补码和原码、反码面的值进行理解）
-47	10101111	11010000	11010001（负数补码是在反

简单讲：

整数的补码和原码相同

负数的补码则是符号位为"1"，数值部分按位取反过后在末位加1，也就是"反码+1"

【*：这里只是简单介绍一下计算机存储的码，如果读者需要更加详细的资料，可以去查阅，这里讲这些是为了方便初学者能够理解Java里面的移位运算符】

2)移位运算例子

这里提供简单的例子：

-5>>3=-1

1111 1111 1111 1111 1111 1111 1111 1011

1111 1111 1111 1111 1111 1111 1111 1111
其结果与 Math.floor((double)-5/(2*2*2)) 完全相同。
-5<<3=-40
1111 1111 1111 1111 1111 1111 1111 1011
1111 1111 1111 1111 1111 1111 1101 1000
其结果与 -5*2*2*2 完全相同。
5>>3=0
0000 0000 0000 0000 0000 0000 0000 0101
0000 0000 0000 0000 0000 0000 0000 0000
其结果与 5/(2*2*2) 完全相同。
5<<3=40
0000 0000 0000 0000 0000 0000 0000 0101
0000 0000 0000 0000 0000 0000 0010 1000
其结果与 5*2*2*2 完全相同。
-5>>>3=536870911
1111 1111 1111 1111 1111 1111 1111 1011
0001 1111 1111 1111 1111 1111 1111 1111
无论正数、负数，它们的右移、左移、无符号右移 32 位都是其本身，比如 -5<<32=-5、-5>>32=-5、-5>>>32=-5。
一个有趣的现象是，把 1 左移 31 位再右移 31 位，其结果为 -1。
0000 0000 0000 0000 0000 0000 0000 0001
1000 0000 0000 0000 0000 0000 0000 0000
1111 1111 1111 1111 1111 1111 1111 1111
小结：到这里基本的Java运算符就已经讲完了，唯一没有讲到的就是instanceof操作符以及关于String的一些操作符，instanceof已经在《类和对象》章节里面讲到了，而String的操作会在String的专程章节里面讲述。

5.控制流程

Java程序和普通程序运行流程一样，有三种运行顺序：顺序、循环和选择，接下来介绍以下Java里面的这三种程序运行流程【本章节的概念代码可能更加基础，而且这一小节不讲保留字goto】：

1)顺序运行：

顺序运行是最简单的Java程序，基本不需要任何关键字就可以操作，提供一段简单的代码读者就可以明白了：

```
package org.susan.java.basic;

public class CommonFlow {
   public static void main(String args[]){
      System.out.println("Step One");
      System.out.println("Step Two");
      System.out.println("Step Three");
   }
}
```

上边的输出这里就不列出了，它会按照顺序打印Step One——>Step Two——>Step Three

2)选择运行：

选择运行需要使用到Java里面的条件语句，条件语句主要包括：

[1]if语句；
[2]if...else语句；
[3]switch语句：

if语句：

if语句的语法有两种：

if(boolean表达式) 语句1;
if(boolean表达式){ 语句1;语句2;}

它所意味着的含义为如果boolean表达式的返回为true就执行语句1或者执行语句1所在的语句块{}内的内容：

```
package org.susan.java.basic;

import java.util.Random;
```

```
public class IfSingleMain {
   public static void main(String args[]){
      Random random = new Random();
      int result = random.nextInt(3);
      if( result == 2){
         System.out.print("This is if block,");
         System.out.println("Test block flow.");
      }
      if( result == 1)
         System.out.println("This is if single flow.");
         System.out.println("This is Inner or Outer."); //这句的写法是故意的
      System.out.println("Main flow.");
   }
}
```

上边这段代码的输出是不固定的，主要是看result返回的值是多少，它会随机生成三个整数值：1、2、3

result值为2的时候直接是块语句，所以不需要讲解什么，但是需要知道if还支持直接的不带块的语句，所以不论任何值生成，都会有下边这两句输出：

This is Inner or Outer.
Main flow.

这里提醒读者的只有一个关键点：在使用if语句的时候，如果后边的语句块不带{}，那么它能产生的效果只有紧跟着if的后边一句话（单条语句），这种情况下，其他的语句都会视为和if无关的语句

if-else语句

该语句的语法为：

```
if(布尔表达式1)
   {语句执行块1}
   else if(布尔表达式2)
   {语句执行块2}
   else
   {语句执行块3}
```

针对该语句，简单修改一下上边的程序：

```
package org.susan.java.basic;

import java.util.Random;

public class IfSingleMain {
   public static void main(String args[]){
      Random random = new Random();
      int result = random.nextInt(3);
      if( result == 2){
         System.out.print("This is if block,");
         System.out.println("Test block flow.");
      }else if( result == 1){
         System.out.println("This is if single flow.");
         System.out.println("This is Inner or Outer.");
      }else {
         System.out.print("This is other flow.");
         System.out.println("This is single else flow.");
      }
      System.out.println("Main flow.");
   }
}
```

这是一个简单的if-else语句，这里只用了else的块语句，没有使用else的单条语句，如果使用else的单条语句，和if是一样的规则：

如果result随机生成的值是0【*：这个地方**result只可能有三个值就是0，1，2**】，就会得到下边的结果：

This is other flow.This is single else flow.
Main flow.

如果去掉最后一个else后边的{}后会有什么效果呢，去掉了最后一个括号过后，下边的语句不论result为任何值的时候都会输出：

```
This is single else flow.
Main flow.
```

原因和if语句一样，如果紧随其后的不是{}的语句块，只能影响一行完整的语句，也就是当去掉上边代码最后一个else后边的{}过后，else只能影响语句：

```
System.out.print("This is other flow.");
```

switch语句：

该语句的语法为：

```
switch(输入因子){
    case 匹配因子1:{执行语句块1;}break;
    caes 匹配因子2:{执行语句块2;}break;
    default:{执行语句块3;}break;
    }
```

当编程过程需要多个分支语句的时候，就需要使用到switch语句，也就是进行多项选择的语句，这里同样提供一个概念说明例子：

```
package org.susan.java.basic;

import java.util.Random;

public class SwitchTester {
   public static void main(String args[]){
      Random random = new Random();
      int result = random.nextInt(4);
      switch (result) {
      case 1:
         System.out.println("Result is 1");
         break;
      case 2:
         System.out.println("Result is 2");
      case 3:
         System.out.println("Result is 3");
         break;
      default:
         System.out.println("Result is 4");
         break;
      }
   }
}
```

当result的值为各种值的时候输出为：

```
Result is 1【*：result的值为1的输出】
Result is 2
Result is 3【*：result的值为2的输出】
Result is 3【*：result的值为3的输出】
Result is 4【*：result的值为0的输出】
```

针对switch语句有几点需要说明：

- switch后边的括号里面的输入因子有类型限制的，只能为：
  [1]int类型的变量
  [2]short类型的变量
  [3]char类型的变量
  [4]byte类型的变量
  [5]enum的枚举类型的变量【**JDK1.5**过后支持】
  其他的变量都是不能传入switch的输入因子的
- 关于case语句的一点点限制：
  [1]case语句后边必须是一个整型的**常量**或者**常量表达式**
  [2]如果使用常量表达式，表达式中的每个变量，必须是**final**的
  [3]case后边不能是非final的变量，比如使用一个case val这种情况将会直接通不过编译，

因为val不是final类型的变量，只是一个普通变量
[4]case后边的常量和常量表达式**不能使用相同的值**

- 关于语句的执行顺序：
  当接收到合法的输入因子过后，JVM会去寻找和系统输入因子匹配的case语句，如果没有任何一个case语句匹配的话就直接执行default语句；当匹配到case语句过后，会执行该case到紧接着代码里面写的该case的下一个case之间的语句块，一旦执行完该语句块过后，如果没有遇到return或者break就继续执行下一个case，执行的流程和该case的执行流程是一样的，知道碰到最终的case为止。所以就可以理解为什么上边的代码当result的值为2的时候会输出：
  Result is 2
  Result is 3

【*：这里思考一个问题，如果把default语句写在case语句的前边会发生什么情况呢？稍稍改动一下上边的代码，就会发现其实default的位置不重要，关键是最终满足的case是否能够匹配，可以这样认为：当所有case条件都不满足的时候选择执行default里面的语句块，当然前提条件是所有的case语句都跟随了一个break语句。一般编程过程中都是直接使case和break配对出现的，因为一旦case执行完了过后，遇到break语句就直接跳出switch语句块了，不会再执行其他的case，如果没有break，还会继续往下一个case执行。而且需要注意一点，执行完匹配的case而没有遇到break或者return的时候，再往下执行就不需要再匹配case条件了，这个时候不论接下来指定的case是否匹配都会执行下去，也就是说，所有的输入因子在进入switch语句过后只匹配一次。还有一点需要提醒一下，当default放在前面的时候，如果default没有匹配的break语句，同样执行完default对应的语句块过后需要执行紧跟着default下边的case语句块，直到遇到break语句。】

3)循环运行
[1]while和do-while循环
[2]for循环
[3]中断循环

while和do-while：

while循环和do-while循环称为条件循环，也就是循环的终止条件为当判断语句为false的时候就终止循环，while和do-while的区别在于不论条件判断返回false还是true，do-while语句至少执行一次，而while语句必须是当条件返回true的时候才会一次次执行

语法规则为：

```
while(布尔表达式){执行语句块}
    do{执行语句块}while(布尔表达式);
```

下边是一个while和do-while循环的例子：

```
package org.susan.java.basic;

public class WhileLoopMain {
   public static void main(String args[]){
      int i = 4;
      System.out.println("While loop:");
      while(i < 4){
         System.out.println(i);
         i++;
      }
      i = 4;
      System.out.println("Do While loop:");
      do{
         System.out.println(i);
         i++;
      }while(i < 4);
   }
}
```

上边是一个很极端的例子，先看输出，然后再分析结果：

```
While loop:
Do While loop:
4
```

这里可以知道的是在while语句里面，先判断i<4，因为i的初始值是4，所以该条件不成立，所以while语句里面直接跳过语句执行块，不打印任何内容；但是针对do-while语句而言，虽然i的初始值也是4，但是i<4是在进行了一次运行过后才比较的，实际上细心的读者会发现，两个循环返回false的

条件不一样。第一个while语句是因为4<4返回false的，而do-while语句却是因为5<4返回false的，这里可以看出while和do-while的细微差异

for循环：

语法规则为：

for(初始条件;判断条件;变化条件){执行语句块;}

for(每一项:包含项的列表)【*：等效于C#里面的foreach语句，而且是**JDK1.5**里面才支持的功能】

同样用一段代码来说明用法：

```
package org.susan.java.basic;

public class ForLoopMain {
   public static void main(String args[]){
      String[] arrayStrings = {"A","B","C"};
      //进行arrayStrings的遍历
      for( int i= 0; i < arrayStrings.length; i++){
         System.out.println(arrayStrings[i]);
      }
      System.out.println("---------------");
      for(String item:arrayStrings){
         System.out.println(item);
      }
   }
}
```

上边代码的输出为：

```
A
B
C
---------------
A
B
C
```

这两种都可以进行循环遍历操作，简单总结以下前两种循环：

- 一般情况下，**while和do-while**循环主要用于**无限循环**，就是不知道循环次数，但是当终止条件满足的时候就退出循环
- **for**循环一般是用于**有限循环**，就是已经知道了循环次数，当到达某种条件的时候退出

这两种循环没有本质的区别，针对哪种循环的选择主要是在于用户在写程序的时候如何设计的问题

中断循环：

循环中断一般有三种方式：**break、continue、标签中断**

break的特征为：当循环遇到了break语句的时候，直接跳出**本循环**；

continue的特征为：当循环遇到了continue语句的时候，直接跳出**本轮循环**，进入**下一次循环**；

标签中断：可以在循环里面使用标签让break或者continue的时候直接从标签位置继续，这种情况有时候用于嵌套循环的一些内容；

——[$]break和continue——

```
package org.susan.java.basic;

public class BreakContinueMain {
   public static void main(String args[]){
      System.out.println("Break Loop");
      for( int i = 0; i < 4; i++){
         if( i == 2)
            break;
         System.out.println("Loop " + i);
      }
      System.out.println("Continue Loop");
      for( int i = 0; i < 4; i++){
         if( i == 2)
            continue;
         System.out.println("Loop " + i);
```

```
        }
    }
}
```

上边这段代码的输出为：

```
Break Loop
Loop 0
Loop 1
Continue Loop
Loop 0
Loop 1
Loop 3
```

【*：读者仔细思考一下，如果没有break和continue语句，应该依次打印i的值为0,1,2,3的每一句话，第一个循环是Break循环，当i=2的时候直接跳出了该循环，按照break的语法是直接跳出循环，所以只打印了i=0和1的时候的情况。而第二个循环是continue循环，当i=2的时候使用了continue，一旦使用了continue过后，该次循环就不执行了，直接进入下一轮循环，所以在continue循环的语句里面只有i=2的语句没有打印出来。】

——[$]使用标签——

```
package org.susan.java.basic;

public class LabelLoopMain {
    public static void main(String args[]){
        outer:for(int i=1; i < 4; i++){
            inner:for(int j =0; j < 5; j++){
            if( j == 2 )
                continue inner;
            System.out.println("i + j = " + (i+j));
            if( j == 4)
                break outer;
            }
        }
        other:for(int i = 0; i < 4; i++){
            if( i == 3){
                break other;
            }
            System.out.println("i = " + i);
        }
    }
}
```

上边这段代码定义了三个循环标签分别为**outer**，**inner**，**other**，输出为：

```
i + j = 1
i + j = 2
i + j = 4
i + j = 5
i = 0
i = 1
i = 2
```

这里仅仅讲解一下简单的标签的语意，这段程序的详细逻辑留给读者自行去分析

- **break labelname**：跳出该标签指代的循环，标签指代的循环就是标签的:后边的内容
- **continue labelname**：继续标签指代的循环，标签指代的循环就是标签的:后边的内容

【*：这里需要说明的是，没有特殊情况可以不用考虑标签循环，标签中断循环主要是设计来为嵌套循环的相互之间的跳转用的，比如有三层循环，在某种条件满足的情况下仅仅想从内层循环跳到中层循环，这种情况下就可以使用标签，直接使用break labelmiddle的方式，labelmiddle就为中间那一层的循环前边定义的标签的名称，其他正常的编程情况能不使用标签的情况可以避免使用标签。】

6.关键字清单

对Java初学者而言，记忆对应的关键字也是一个不错的学习方式，这里提供一个与Java关键字相关的查询文档：

i.关键字、字面量、保留字

在Java里，按照使用的方式把所有可以被IDE着色的Java字分为三种类型：

- **关键字**：关键字就是目前正在使用的Java语法
- **字面量**：在Sun公司的官方规范里面有明确的说明，**false、true和null**这三个虽然IDE提供了语法着色，但是这三个不能称为关键字，而是使用的时候提供的字面量
- **保留字**：保留字在Java里面称为以后可能会用到的语法，就是定义标识符的时候不能使用，主要有**const和goto**

ii.关键字清单以及说明

- **[3]访问控制**：private、protected、public
- **[13]类、方法和变量修饰符**：abstract、class、extends、final、implements、interface、native、new、static、strictfp、synchronized、transient、volatile
- **[12]程序控制语句**：break、continue、return、do、while、if、else、for、instanceof、switch、case、default
- **[5]错误处理**：catch、finally、throw、throws、try
- **[2]包相关**：import、package
- **[8]基本类型**：boolean、byte、char、double、float、int、long、short
- **[3]变量引用**：super、this、void
- **[1]JDK1.5新关键字**：enum
- **[2]其他**：new、assert

【*：需要说明的是const和goto从概念上来将属于Java里面的保留字，不应该列入关键字之列；**true、false和null**三个属于字面量，虽然属于Java里面的关键字，但是在规范里面不列入关键字之列。这里解释一下，很多资料都说Java里面有51个关键字，数数上边就会发现**只有49个**，51个关键字的来历是什么呢？51个关键在是在上边关键字的基础上计算了**false、true和null**，而且是JDK1.4的说法，还去掉了新关键字**enum**，所以很多资料都记载了51个关键字。】

7.小节

到这里，整个Java语言最基础的语法、流程、关键字、标识符等各种前边的基础知识就讲完了，没有设计到的内容就是JDK的工具以及JDK环境的配置，基本上用这一个章节直接替代掉原来写的Java基础类型的章节应该方便初学者进行学习，至于目前还没有涉及的IO、格式化、序列化、多线程以及JavaSE的基础部分的内容我会在后边的章节慢慢写上来，只是完成这样一篇BLOG可能花费的时间很长，所以产量比较慢希望读者能够见谅。而且每一次写完了我自己要阅读几遍才能帮着读者标注整个这篇文章的重点以及代码着色。若有什么笔误，请来Email告知，谢

谢：**silentbalanceyh@126.com**