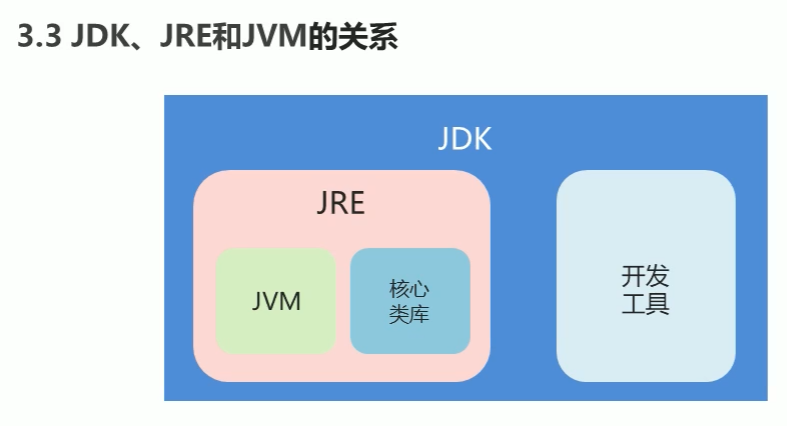
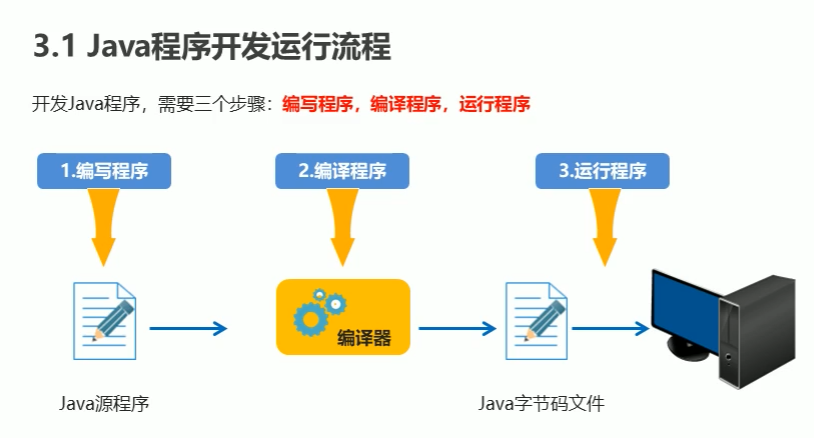
Java之父：詹姆斯\*高斯林

Java语言跨平台原理：在需要运行Java应用程序的操作系统上，安装一个与操作系统对应的Java虚拟机JVM (Java Virtual Machine)即可。

JRE和JDK：JRE (Java Runtime Environment)是Java程序的运行时环境，包含JVM和运行时所需要的核心类库。想要**运行**一个已有的Java程序，只需安装JRE即可。JDK (Java Development Kit)是Java程序开发工具包，包含JRE和开发人员使用的工具，其中开发工具包括：编译工具(javac.exe)和运行工具(java.exe)。想要开发一个全新的Java程序，就必须安装JDK。

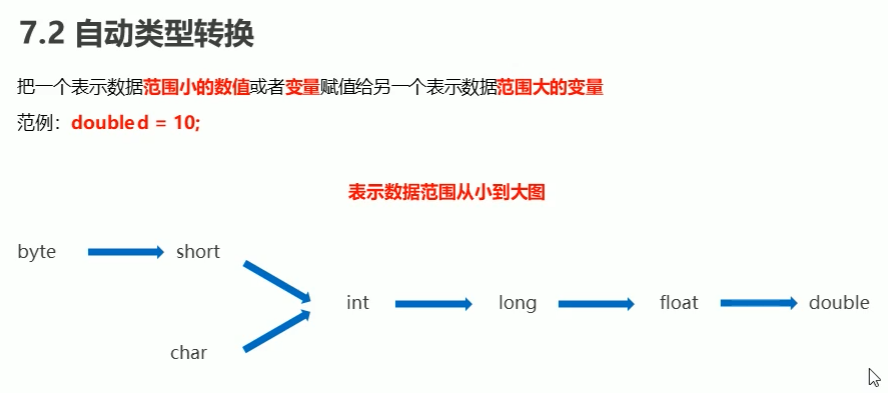


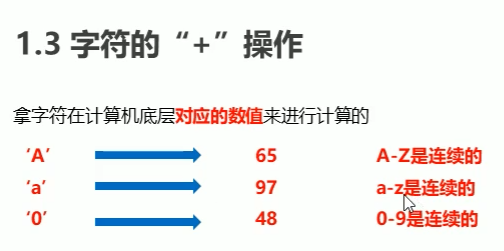


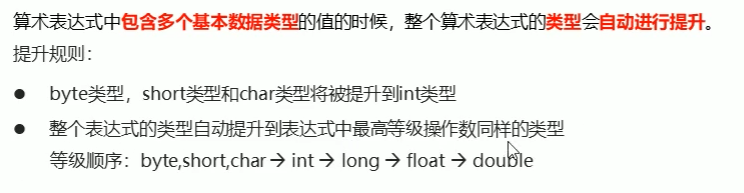








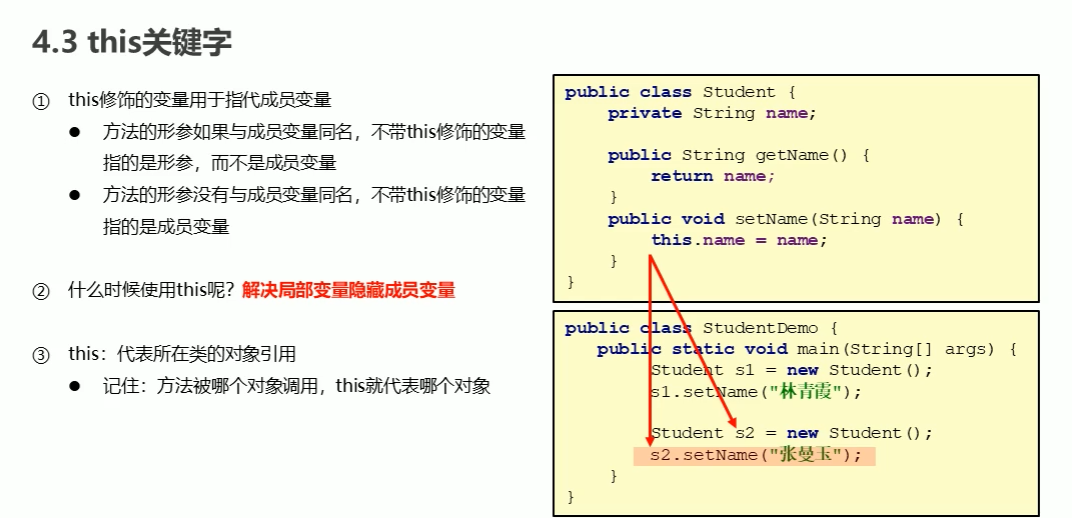


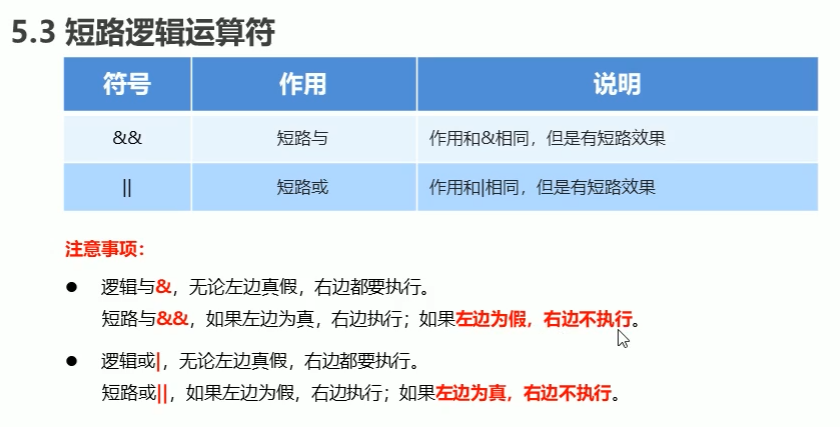




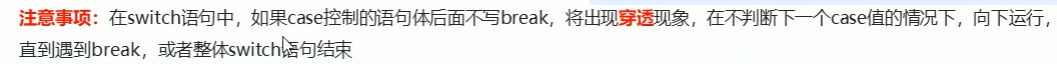


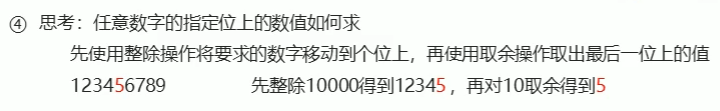


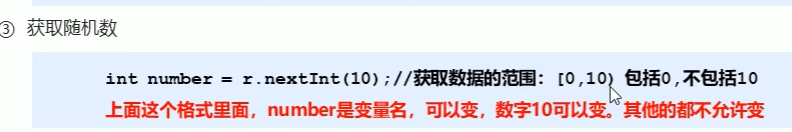


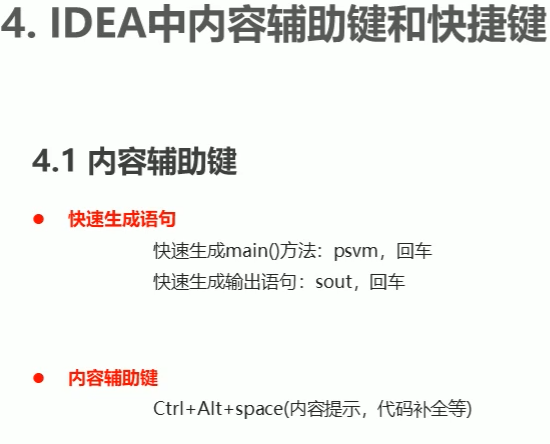


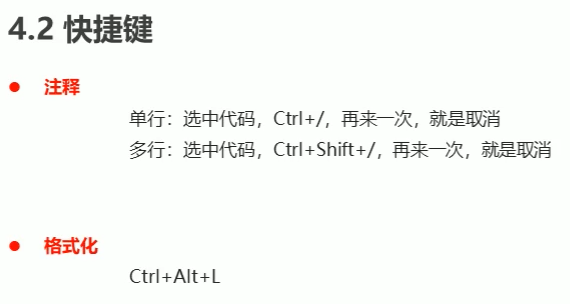
Switch中的case穿透现象

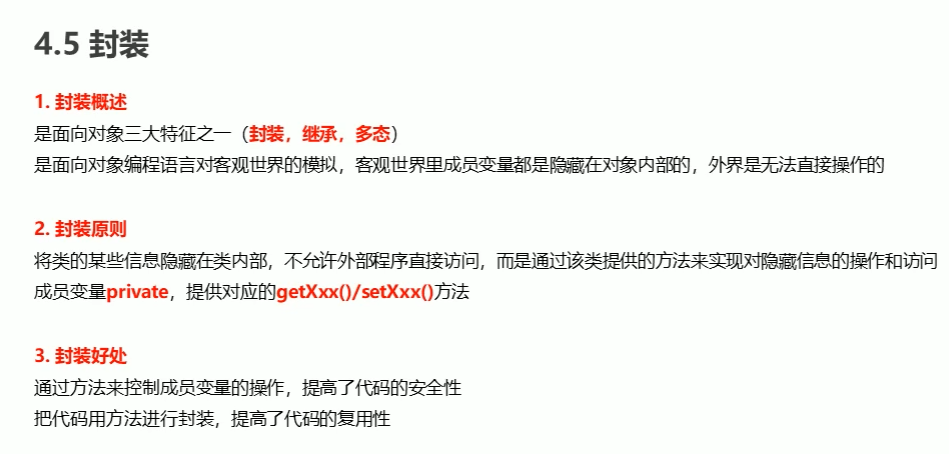


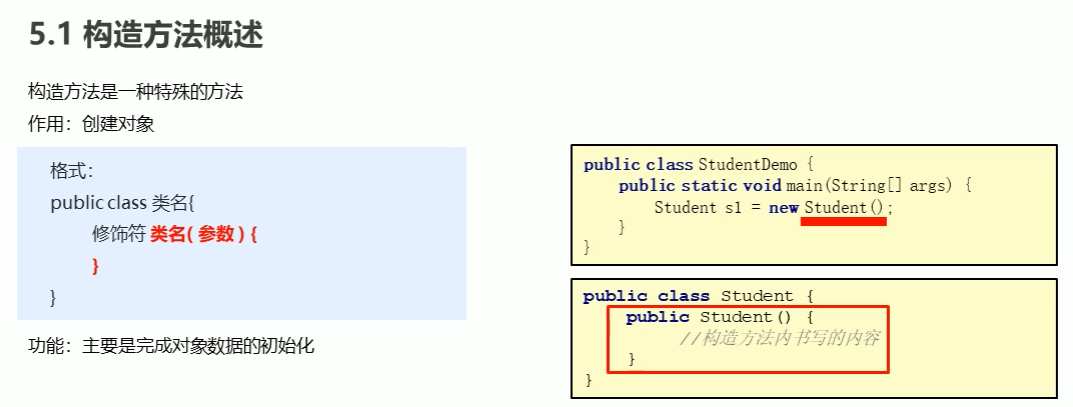


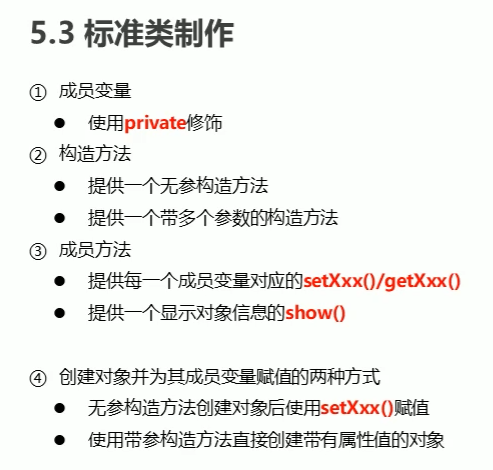


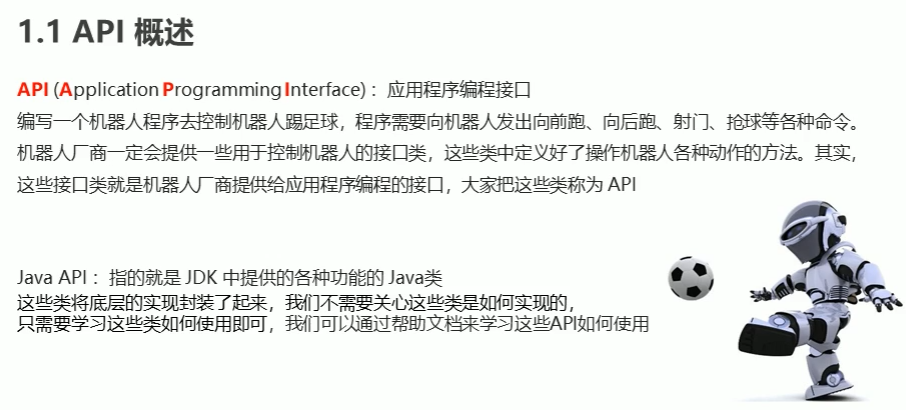




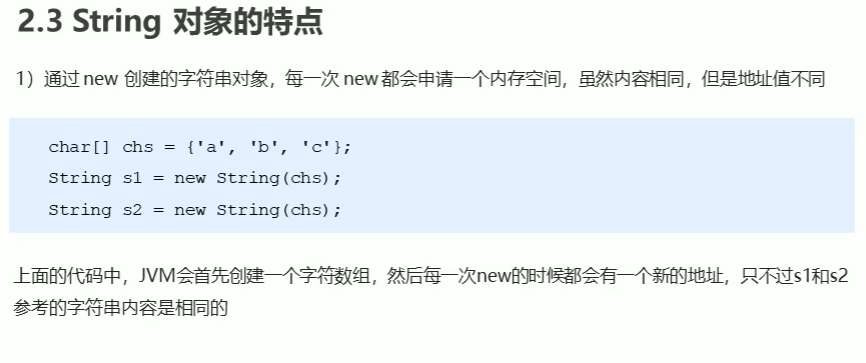


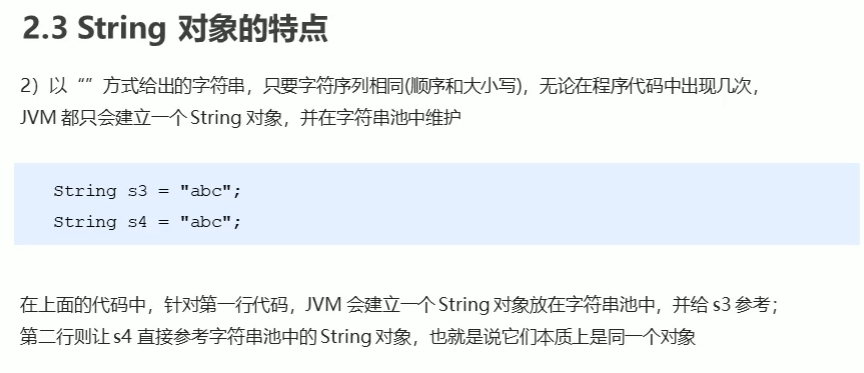








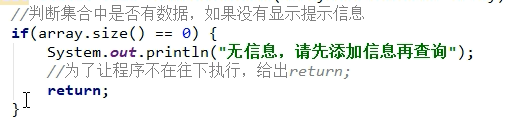


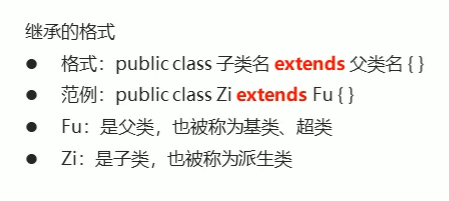




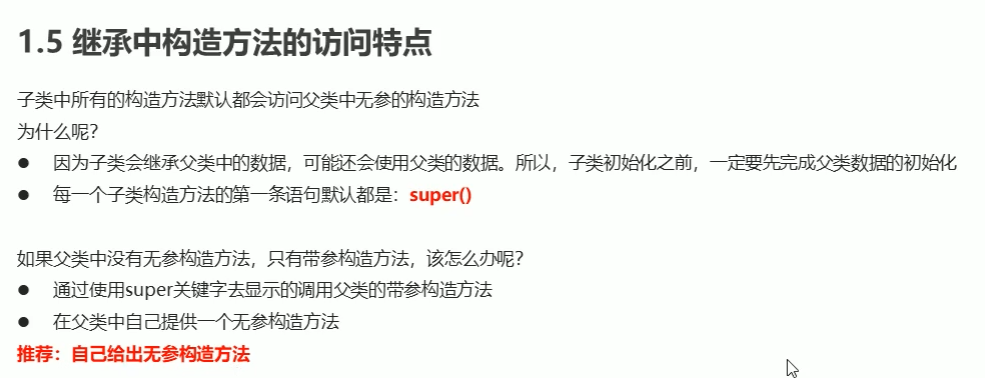


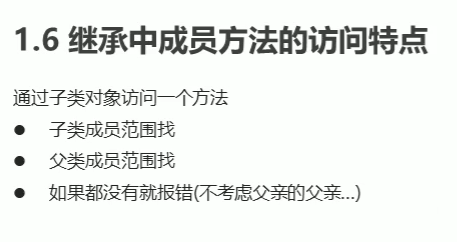
System.exit(0); //JVM虚拟机退出。

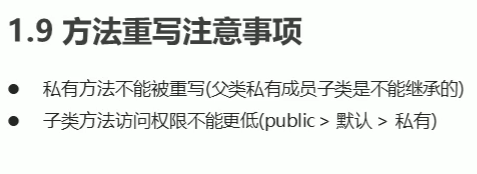






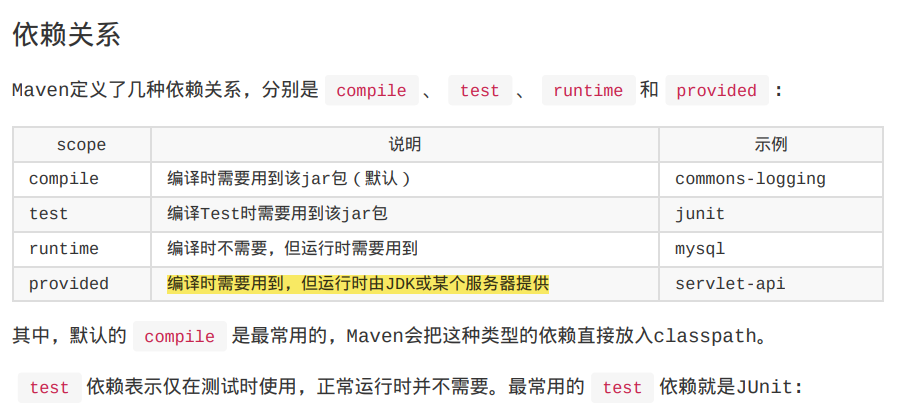






了解即可：





Maven通过解析依赖关系确定项目所需的jar包，常用的4种scope有：compile（默 认），test，runtime和provided ；

runtime 依赖表示编译时不需要，但运行时需要。最典型的 runtime 依赖是JDBC驱动，例如 MySQL驱动。

provided 依赖表示编译时需要，但运行时不需要。最典型的 provided 依赖是Servlet API， 编译的时候需要，但是运行时，Servlet服务器内置了相关的jar，所以运行期不需要。

**为啥需要Unicode**

      刚开始的时候有ASCII字符集(American Standard Code for Information Interchange，美国信息交换标准码),它使用7 bits(剩下的1位二进制为0)来表示一个字符,总共表示128个字符,我们一般都是用字节(byte,即8个01串)来作为基本单位，后来IBM公司在此基础上进行了扩展，用8bit来表示一个字符，总共可以表示256个字符.也就是当第一个bit是0时仍表示之前那些常用的字符.当为1时就表示其他补充的字符.

        任何一个字符只对应一个确定的数字.ISO取的名字叫UCS(Universal Character Set),Unicode取的名字就叫unicode了.

      总结起来为啥需要Unicodey就是为了适应全球化的发展,便于不同语言之间的兼容交互,而ASCII不再能胜任此任务了.

**Unicode详细介绍**

**1.容易产生后歧义的两字节**

        unicode的第一个版本是用两个字节(16bit)来表示所有字符

       我们总觉得两个字节就代表保存在计算机中时是两个字节.于是任何字符如果用unicode表示的话保存下来都占两个字节.其实这种说法是错误的.

     其实Unicode涉及到两个步骤,首先是定义一个规范,给所有的字符指定一个唯一对应的数字,这完全是数学问题,可以跟计算机没半毛钱关系.第二步才是怎么把字符对应的数字保存在计算机中,这才涉及到实际在计算机中占多少字节空间.

     所以我们也可以这样理解,Unicode是用0至65535之间的数字来表示所有字符.其中0至127这128个数字表示的字符仍然跟ASCII完全一样.65536是2的16次方.这是第一步.第二步就是怎么把0至65535这些数字转化成01串保存到计算机中.这肯定就有不同的保存方式了.于是出现了UTF(unicode transformation format),有UTF-8,UTF-16.

**Unicode** 只是一种**字符码表**， 而在计算机中进行存储时，必须指定一种具体的存储方式。常见的如**utf8**, **utf16**, **utf32**等。

UTF-8编码的最大长度是4个字节，即可以容纳21位二进制数字。而 **Unicode** 的最大码位0x10FFFF也只有21位

**2.UTF-8 与UTF-16的区别**

    UTF-16比较好理解,就是任何字符对应的数字都用两个字节来保存.我们通常对Unicode的误解就是把Unicode与UTF-16等同了.但是很显然如果都是英文字母这做有点浪费.明明用一个字节能表示一个字符为啥整两个啊.

   于是又有个UTF-8,这里的8非常容易误导人,8不是指一个字节,难道一个字节表示一个字符? 实际上不是.当用UTF-8时表示一个字符是可变的,有可能是用一个字节表示一个字符,也可能是两个,三个..反正是根据字符对应的数字大小来确定.

   于是UTF-8和UTF-16的优劣很容易就看出来了.如果全部英文或英文与其他文字混合,但英文占绝大部分,用UTF-8就比UTF-16节省了很多空间.而如果全部是中文这样类似的字符或者混合字符中中文占绝大多数.UTF-16就占优势了,可以节省很多空间.另外还有个容错问题,等会再讲

   看的有点晕了吧,举个例子.假如中文字"汉"对应的unicode是6C49(这是用十六进制表示,用十进制表示是27721为啥不用十进制表示呢?很明显用十六进制表示要短点.其实都是等价的没啥不一样.就跟你说60分钟和1小时一样.).你可能会问当用程序打开一个文件时我们怎么知道那是用的UTF-8还是UTF-16啊.自然会有点啥标志,在文件的开头几个字节就是标志.

EF BB BF 表示UTF-8

FE FF 表示UTF-16.

 用UTF-16表示"汉"

假如用UTF-16表示的话就是01101100   01001001(共16 bit,两个字节).程序解析的时候知道是UTF-16就把两个字节当成一个单元来解析.这个很简单.

用UTF-8表示"汉"

用UTF-8就有复杂点.因为此时程序是把一个字节一个字节的来读取,然后再根据字节中开头的bit标志来识别是该把1个还是两个或三个字节做为一个单元来处理.

0xxxxxxx,如果是这样的01串,也就是以0开头后面是啥就不用管了XX代表任意bit.就表示把一个字节做为一个单元.就跟ASCII完全一样.

110xxxxx 10xxxxxx.如果是这样的格式,则把两个字节当一个单元

1110xxxx 10xxxxxx 10xxxxxx 如果是这种格式则是三个字节当一个单元.

这是约定的规则.你用UTF-8来表示时必须遵守这样的规则.我们知道UTF-16不需要用啥字符来做标志,所以两字节也就是2的16次能表示65536个字符.

而UTF-8由于里面有额外的标志信息,所有一个字节只能表示2的7次方128个字符,两个字节只能表示2的11次方2048个字符.而三个字节能表示2的16次方,65536个字符.

由于"汉"的编码27721大于2048了所有两个字节还不够,只能用三个字节来表示.

所有要用1110xxxx 10xxxxxx 10xxxxxx这种格式.把27721对应的二进制从左到右填充XXX符号(实际上不一定从左到右,也可以从右到左,这是涉及到另外一个问题.等会说.

刚说到填充方式可以不一样,于是就出现了Big-Endian,Little-Endian的术语.Big-Endian就是从左到右,Little-Endian是从右到左.

由上面我们可以看出UTF-8在局部的字节错误（丢失、增加、改变）不会导致连锁性的错误，因为 UTF-8 的字符边界很容易检测出来，所以容错性较高。

**Unicode版本2**

    前面说的都是unicode的第一个版本.但65536显然不算太多的数字,用它来表示常用的字符是没一点问题.足够了,但如果加上很多特殊的就也不够了.于是从1996年开始又来了第二个版本.用四个字节表示所有字符.这样就出现了UTF-8,UTF16,UTF-32.原理和之前肯定是完全一样的,UTF-32就是把所有的字符都用32bit也就是4个字节来表示.然后UTF-8,UTF-16就视情况而定了.UTF-8可以选择1至8个字节中的任一个来表示.而UTF-16只能是选两字节或四字节..由于unicode版本2的原理完全是一样的,就不多说了.

前面说了要知道具体是哪种编码方式,需要判断文本开头的标志,下面是所有编码对应的开头标志

EF BB BF　　　 UTF-8  
FE FF　　　　　UTF-16/UCS-2, little endian  
FF FE　　　　　UTF-16/UCS-2, big endian  
FF FE 00 00　　UTF-32/UCS-4, little endian.  
00 00 FE FF　　UTF-32/UCS-4, big-endian.

其中的UCS就是前面说的ISO制定的标准,和Unicode是完全一样的,只不过名字不一样.ucs-2对应utf-16,ucs-4对应UTF-32.UTF-8是没有对应的UCS

**UTF-16 并不是一个完美的选择，它存在几个方面的问题：**

1. UTF-16 能表示的字符数有 6 万多，看起来很多，但是实际上目前 Unicode 5.0 收录的字符已经达到 99024 个字符，早已超过 UTF-16 的存储范围；这直接导致 UTF-16 地位颇为尴尬——如果谁还在想着只要使用 UTF-16 就可以高枕无忧的话，恐怕要失望了
2. UTF-16 存在大小端字节序问题，这个问题在进行信息交换时特别突出——如果字节序未协商好，将导致乱码；如果协商好，但是双方一个采用大端一个采用小端，则必然有一方要进行大小端转换，性能损失不可避免（大小端问题其实不像看起来那么简单，有时会涉及硬件、操作系统、上层软件多个层次，可能会进行多次转换）
3. 另外，容错性低有时候也是一大问题——局部的字节错误，特别是丢失或增加可能导致所有后续字符全部错乱，错乱后要想恢复，可能很简单，也可能会非常困难。（这一点在日常生活里大家感觉似乎无关紧要，但是在很多特殊环境下却是巨大的缺陷）

目前支撑我们继续使用 UTF-16 的理由主要是考虑到它是双字节的，在计算字符串长度、执行索引操作时速度很快。当然这些优点 UTF-32 都具有，但很多人毕竟还是觉得 UTF-32 太占空间了。

**UTF—16**(*16-bit Unicode Transformation Format*)，以16位无符号整数为单位对**Unicode**进行编码。

**UTF-16** 也是一种变长编码，对于一个Unicode的字符被编码成1~2个code unit，每个code unit使用16位 （2个字节）。

**反过来 UTF-8 也不完美，也存在一些问题：**

1. 文化上的不平衡——对于欧美地区一些以英语为母语的国家 UTF-8 简直是太棒了，因为它和 ASCII 一样，一个字符只占一个字节，没有任何额外的存储负担；但是对于中日韩等国家来说，UTF-8 实在是太冗余，一个字符竟然要占用 3 个字节，存储和传输的效率不但没有提升，反而下降了。所以欧美人民常常毫不犹豫的采用 UTF-8，而我们却老是要犹豫一会儿
2. 变长字节表示带来的效率问题——大家对 UTF-8 疑虑重重的一个问题就是在于其因为是变长字节表示，因此无论是计算字符数，还是执行索引操作效率都不高。为了解决这个问题，常常会考虑把 UTF-8 先转换为 UTF-16 或者 UTF-32 后再操作，操作完毕后再转换回去。而这显然是一种性能负担。

**当然，UTF-8 的优点也不能忘了：**

1. 字符空间足够大，未来 Unicode 新标准收录更多字符，UTF-8 也能妥妥的兼容，因此不会再出现 UTF-16 那样的尴尬
2. 不存在大小端字节序问题，信息交换时非常便捷
3. 容错性高，局部的字节错误（丢失、增加、改变）不会导致连锁性的错误，因为 UTF-8 的字符边界很容易检测出来，这是一个巨大的优点（正是为了实现这一点，咱们中日韩人民不得不忍受 3 字节 1 个字符的苦日子）

**那么到底该如何选择呢？**  
因为无论是 UTF-8 和 UTF-16/32 都各有优缺点，因此**选择的时候应当立足于实际的应用场景**。例如在我的习惯中，存储在磁盘上或进行网络交换时都会采用 UTF-8，而在程序内部进行处理时则转换为 UTF-16/32。对于大多数简单的程序来说，这样做既可以保证信息交换时容易实现相互兼容，同时在内部处理时会比较简单，性能也还算不错。（基本上只要你的程序不是 I/O 密集型的都可以这么干，当然这只是我粗浅的认识范围内的经验，很可能会被无情的反驳）  
**稍微再展开那么一点点……**  
在一些特殊的领域，字符编码的选择会成为一个很关键的问题。特别是一些高性能网络处理程序里更是如此。这时采用一些特殊的设计技巧，可以缓解性能和字符集选择之间的矛盾。例如对于内容检测/过滤系统，需要面对任何可能的字符编码，这时如果还采用把各种不同的编码都转换为同一种编码后再处理的方案，那么性能下降将会很显著。而如果采用多字符编码支持的有限状态机方案，则既能够无需转换编码，同时又能够以极高的性能进行处理。当然如何从规则列表生成有限状态机，如何使得有限状态机支持多编码，以及这将带来哪些限制，已经又成了另外的问题了。

InputStream is = BaseDao.class.getClassLoader().getResourceAsStream("db.properties");

class是指当前类的class对象，getClassLoader()是获取当前的类加载器，什么是类加载器？简单点说，就是用来加载java类的,类加载器负责把class文件加载进内存中，并创建一个java.lang.Class类的一个实例，也就是class对象，并且每个类的类加载器都不相同。getResourceAsStream(path)是用来获取资源的，而类加载器默认是从classPath下获取资源的，因为这下面有class文件吗，所以这段代码总的意思是通过类加载器在classPath目录下获取资源.并且是以流的形式。

我们知道在Java中所有的类都是通过加载器加载到虚拟机中的，而且类加载器之间存在父子关系，就是子知道父，父不知道子，这样不同的子加载的类型之间是无法访问的（虽然它们都被放在方法区中），所以在这里通过当前类的加载器来加载资源也就是保证是和类类型同一个加载器加载的。

Javaweb重定向和转发的路径问题：

1）重定向因为是从客户端发来的，所以只知道发到那个服务器，不知道发给那个项目，所以重定向的"/"表示"http:服务器ip:8080/"

2）请求转发因为服务器内部自己转发，因此知道发给那个项目，所以请求转发的"/"表示"http:服务器ip:8080/项目名/"

3）重定向没有"/"表示在当前目录，请求转发没有"/"表示相对于当前资源的相对路径

注意：相对路径使用“/”字符作为目录的分隔字符，而绝对路径可以使用“\”或“/”字符作为目录的分隔字符。

在相对路径里常使用“../”来表示上一级目录。如果有多个上一级目录，可以使用多个“../”

Java关键字 transient：

1）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。

2）transient关键字只能修饰变量，而不能修饰方法和类。注意，本地变量是不能被transient关键字修饰的。变量如果是用户自定义类变量，则该类需要实现Serializable接口。

3）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。

第三点可能有些人很迷惑，因为发现在User类中的username字段前加上static关键字后，程序运行结果依然不变，即static类型的username也读出来为“Alexia”了，这不与第三点说的矛盾吗？实际上是这样的：第三点确实没错（一个静态变量不管是否被transient修饰，均不能被序列化），反序列化后类中static型变量username的值为当前JVM中设置的对应static变量的值，这个值是JVM中的不是反序列化得出的

transient使用细节——被transient关键字修饰的变量真的不能被序列化吗？

|  |
| --- |
| public class ExternalizableTest implements Externalizable {  private transient String content = "是的，我将会被序列化，不管我是否被transient关键字修饰";  @Override  public void writeExternal(ObjectOutput out) throws IOException {  out.writeObject(content);  }  @Override  public void readExternal(ObjectInput in) throws IOException,  ClassNotFoundException {  content = (String) in.readObject();  }  public static void main(String[] args) throws Exception {    ExternalizableTest et = new ExternalizableTest();  ObjectOutput out = new ObjectOutputStream(new FileOutputStream(  new File("test")));  out.writeObject(et);  ObjectInput in = new ObjectInputStream(new FileInputStream(new File(  "test")));  et = (ExternalizableTest) in.readObject();  System.out.println(et.content);  out.close();  in.close();  }  } |
| 运行结果是：是的，我将会被序列化，不管我是否被transient关键字修饰 |

在Java中，对象的序列化可以通过实现两种接口来实现，若实现的是Serializable接口，则所有的序列化将会自动进行，若实现的是Externalizable接口，则没有任何东西可以自动序列化，需要在writeExternal方法中进行手工指定所要序列化的变量，这与是否被transient修饰无关。因此输出的是变量content初始化的内容，而不是null；

当Java虚拟机要加载一个类时，到底派出哪个类加载器去加载呢？

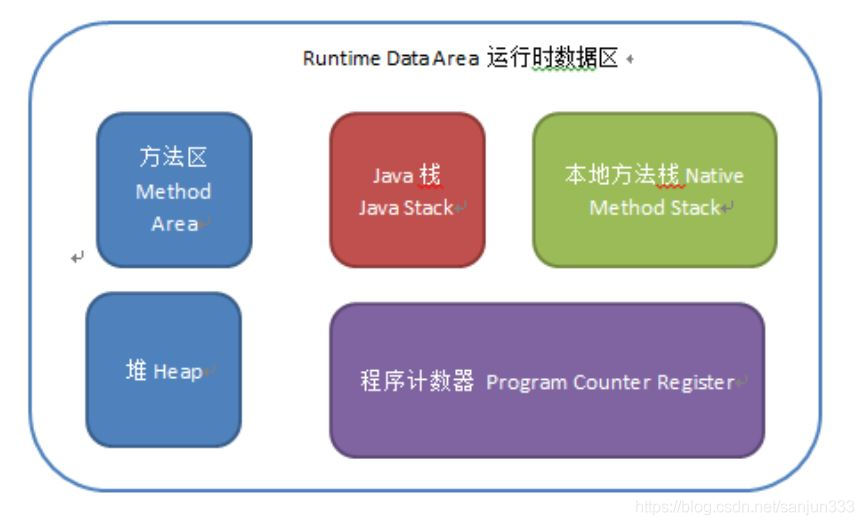
首先当前线程的类加载器去加载线程中的第一个类（假设为类A）。  
注：当前线程的类加载器可以通过Thread类的getContextClassLoader()获得，也可以通过setContextClassLoader()自己设置类加载器。

如果类A中引用了类B，Java虚拟机将使用加载类A的类加载器去加载类B。

还可以直接调用ClassLoader.loadClass()方法来指定某个类加载器去加载某个类。

**Java内存结构**

Java内存结构是在说明运行时的数据区域的划分，JVM在执行Java程序时，会把它管理的内存划分为几个不同的数据区域，这些区域都有个自的用途、创建时间、销毁时间。



1、PC寄存器/程序计数器（Program Counter Register）

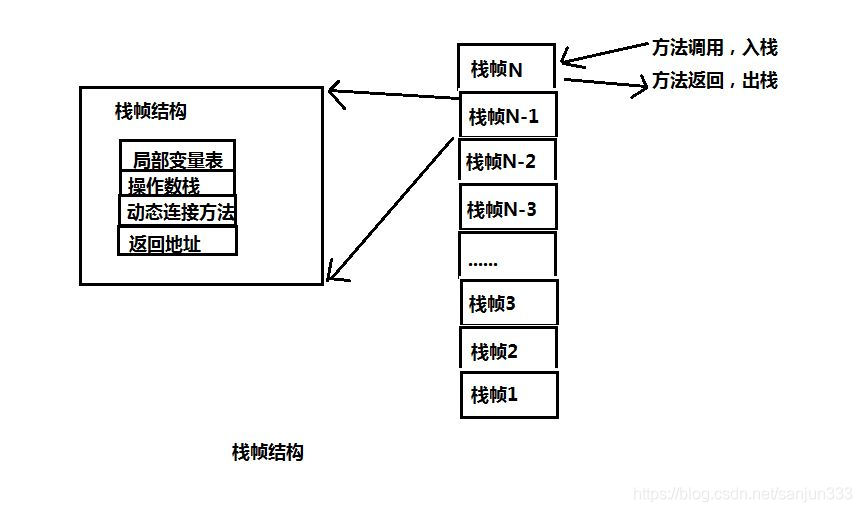
严格来说是一个数据结构，用于保存当前正在执行的程序的内存地址，由于Java是支持多线程执行的，所以程序执行的轨迹不可能一直都是线性执行。当有多个线程交叉执行时，被中断的线程的程序当前执行到哪条内存地址必然要保存下来，以便用于被中断的线程恢复执行时再按照被中断时的指令地址继续执行下去。为了线程切换后能恢复到正确的执行位置，每个线程都需要有一个独立的程序计数器，各个线程之间计数器互不影响，独立存储，我们称这类内存区域为“线程私有”的内存,这在某种程度上有点类似于“ThreadLocal”，是线程安全的。

2、Java栈（Java Stack）

Java栈总是与线程关联在一起的，每当创建一个线程，JVM就会为该线程创建对应的Java栈，在这个Java栈中又会包含多个栈帧(Stack Frame)，这些栈帧是与每个方法关联起来的，每运行一个方法就创建一个栈帧，每个栈帧会含有一些局部变量、操作栈和方法返回值等信息。每当一个方法执行完成时，该栈帧就会弹出栈帧的元素作为这个方法的返回值，并且清除这个栈帧，Java栈的栈顶的栈帧就是当前正在执行的活动栈，也就是当前正在执行的方法，PC寄存器也会指向该地址。只有这个活动的栈帧的本地变量可以被操作栈使用，当在这个栈帧中调用另外一个方法时，与之对应的一个新的栈帧被创建，这个新创建的栈帧被放到Java栈的栈顶，变为当前的活动栈。同样现在只有这个栈的本地变量才能被使用，当这个栈帧中所有指令都完成时，这个栈帧被移除Java栈，刚才的那个栈帧变为活动栈帧，前面栈帧的返回值变为这个栈帧的操作栈的一个操作数。

由于Java栈是与线程对应起来的，Java栈数据不是线程共有的，所以不需要关心其数据一致性，也不会存在同步锁的问题。

在Java虚拟机规范中，对这个区域规定了两种异常状况：如果线程请求的栈深度大于虚拟机所允许的深度，将抛出StackOverflowError异常；如果虚拟机可以动态扩展，如果扩展时无法申请到足够的内存，就会抛出OutOfMemoryError异常。在Hot Spot虚拟机中，可以使用-Xss参数来设置栈的大小。栈的大小直接决定了函数调用的可达深度。



3.堆 (Heap)

堆是JVM所管理的内存中最大的一块，是被所有Java线程所共享的，不是线程安全的，在JVM启动时创建。堆是存储Java对象的地方，这一点Java虚拟机规范中描述是：所有的对象实例以及数组都要在堆上分配。Java堆是GC管理的主要区域，从内存回收的角度来看，由于现在GC基本都采用分代收集算法，所以Java堆还可以细分为：新生代和老年代；新生代再细致一点有Eden空间、From Survivor空间、To Survivor空间等。

4.方法区(Method Area)

方法区存放了要加载的类的信息（名称、修饰符等）、类中的静态常量、类中定义为final类型的常量、类中的Field信息、类中的方法信息，当在程序中通过Class对象的getName.isInterface等方法来获取信息时，这些数据都来源于方法区。方法区是被Java线程锁共享的，不像Java堆中其他部分一样会频繁被GC回收，它存储的信息相对比较稳定，在一定条件下会被GC，当方法区要使用的内存超过其允许的大小时，会抛出OutOfMemory的错误信息。方法区也是堆中的一部分，就是我们通常所说的Java堆中的永久区 Permanet Generation，大小可以通过参数来设置,可以通过-XX:PermSize指定初始值，-XX:MaxPermSize指定最大值。

5.常量池(Constant Pool)

常量池本身是方法区中的一个数据结构。常量池中存储了如字符串、final变量值、类名和方法名常量。常量池在编译期间就被确定，并保存在已编译的.class文件中。一般分为两类：字面量和应用量。字面量就是字符串、final变量等。类名和方法名属于引用量。引用量最常见的是在调用方法的时候，根据方法名找到方法的引用，并以此定位到函数体进行函数代码的执行。引用量包含：类和接口的全限定名、字段的名称和描述符，方法的名称和描述符。

6.本地方法栈(Native Method Stack)

本地方法栈和Java栈所发挥的作用非常相似，区别不过是Java栈为JVM执行Java方法服务，而本地方法栈为JVM执行Native方法服务。本地方法栈也会抛出StackOverflowError和OutOfMemoryError异常。

static静态代码块。静态代码块的作用也是完成一些初始化工作。首先执行静态块，然后执行构造方法。静态代码块在类被加载的时候执行，而构造方法是在生成对象的时候执行；要调用某个类来生成对象，首先需要将类加载到Java虚拟机上(JVM)，然后由JVM加载这个类来生成对象。

类的静态代码块只会执行一次，是在类被加载的时候执行的，因为每个类只会加载一个，所以静态代码块也只会被执行一次；而构造方法则不然，每次生成一个对象的时候都会调用类的构造方法，所以new一次就会调用构造方法一次。

final修饰方法：当一个方法被final所修饰的时，表示该方法是一个终态方法，即不能被重写(Override)。

final修饰类：当一个类被final所修饰时，表示该类是一个终态类，即不能被继承。

static：强调唯一性，它的使用无需建立对象，也就是一个类中所有static修饰的量或方法就是唯一的，对象的不同不会影响到他，甚至不建立对象都可以使用static修饰的量或方法。声明时未赋值会赋默认值。static可以修饰内部类，修饰以后的内部类可以像普通外部类一样调用。

final：强调不变性，但是必须建立对象才能调用，随着对象的不同，final修饰的量也可以不同，所以他的初始化可以在声明时进行，也可以在构造函数中进行，当然他修饰的类不可以继承，修饰的量不可以改变。

static final：结合上述两者特性，使用无需实例化，声明必须初始化。

如果继承体系既有构造方法，又有静态代码块，那么首先执行最顶层的类的静态代码块，一直执行到最底层类的静态代码块，然后再去执行最顶层类的构造方法，一直执行到最底层的构造方法，注意：静态代码块只会执行一次。

访问final修饰的静态变量时，不会触发类加载，因为在编译期已经将此常量放在常量池了。

访问类的静态变量会触发类加载。 new创建实例对象，会触发类加载进行。

访问类的静态方法会触发类加载。反射会触发类加载。

类的实例化顺序：先父类静态变量/静态代码块-> 再子类静态变量/静态代码块->父类构造器->子类构造器。

定义了main()方法的那个类会先触发类加载。

静态代码块是自动执行的; 静态方法是被调用的时候才执行的。

静态代码块只在第一次被类加载器加载时执行一次，之后不再执行，而非静态代码块在每new一次就执行一次。

\* 静态函数（main方法）不能直接调用非静态成员属性和成员方法;

\* 静态函数（main方法）可以通过创建实例对象来调用非静态的成员属性与成员方法;

在继承中代码的执行顺序为：

1.父类静态对象，父类静态代码块

2.子类静态对象，子类静态代码块

3.父类非静态对象，父类非静态代码块

4.父类构造函数

5.子类非静态对象，子类非静态代码块

6.子类构造函数

|  |
| --- |
| package com.pattern.test;  */\*执行顺序：父类静态代码块、静态变量 -> 子类静态代码块、静态变量 -> 父类非静态代码块、成员变量 -> 父类构造函数-> 子类非静态代码块、成员变量-> 子类构造函数\*/* public class Child extends Father{  static {  System.*out*.println("child-->static");  }   private int n = 20;   {  System.*out*.println("Child Non-Static n=" + n);  n = 30;  }   public int x = 200;   public Child() {  this("The other constructor");  System.*out*.println("child constructor body: " + n);  }   public Child(String s) {  System.*out*.println("child's s=" + s);  System.*out*.println("child's n=" + n);  }   public void age() {  System.*out*.println("age=" + n);  }   public void printX() {  System.*out*.println("x=" + x);  }   public static void main(String[] args) {  new Child().printX();  }  }  class Father {  static {  *//System.out.println("n+"+n);  //当n定义在下面时，会提示Cannot reference a field before it is defined，  //所以必须把n定义移到上面才可以输出* System.*out*.println("super-->static");   }   public static int *n* = 10;   public int x = 100;   public Father() {  System.*out*.println("super's x=" + x);  age();  }   {  System.*out*.println("Father Non-Static");  System.*out*.println("super's x=" + x);  System.*out*.println("super's n=" + *n*);  x = 50;  }   public void age(){  System.*out*.println("nothing");  }  } |

在有继承关系时，创建子类对象，无论调用无参还有带参构造，都会调用父类的构造器，对父类对象初始化（但是在初始化父类对象前，先会执行子类中重载构造器中this()调用，执行后才会调用父类构造器，默认没有this()的构造器有隐式super(),当然也可以指定构带参构造器）。初始化父类对象，默认进入父类的空参构造器中（此时子类构造器中有个隐匿surper()），但如果调用子类构造器中第一行已经明确选择父类带参构造器，则进入父类带参构造器，super调用和this调用都只能在构造器中使用，而且super和this的调用必须作为构造器的第一行代码。

|  |
| --- |
| package com.pattern.test;   public class Son1 extends Father11 {   int age;   public Son1(int age) {  *// super(); 此处为隐式* System.*out*.println("son带参构造");   }   public Son1() {  this(3);  System.*out*.println("son空参构造");  }   public static void main(String[] args) {  Son1 son = new Son1();  System.*out*.println(son);  } }  class Father11 {  int age;   public Father11(int age) {  super(); *//隐式* this.age = age;  System.*out*.println("我是father带参空构造");  }   public Father11() {  this(39);  System.*out*.println("我是father空构造");   }  } |

还有一点也至关重要，当父类中有带参构造器，无空参构造，且子类中没有明确指出使用父类带参构造生成父类对象时，程序会默认使用super()空参构造生成父类对象，此时程序会报错。原因很简单：当一个类手动加入了带参构造器，系统将不会再为你自动生成空参构造。这时就需要我们为父类添加空参构造。

this()和super()都指的是对象，所以，均不可以在static环境中使用。包括：static变量,static方法，static语句块。

java中，在使用new操作符创建一个类的实例对象的时候，开始分配空间并将成员变量初始化为默认数值，注意这里并不是指将变量初始化为在变量定义处的初始值，而是给整型赋值0，给字符串赋值null 这一点于C++不同，(student.name = null , student.age = 0 )，然后在进入类的构造函数。

在构造函数里面，首先要检查是否有this或者super调用，this调用是完成本类本身的构造函数之间的调用，super调用是完成对父类的调用。二者只能出现一个，并且只能作为构造函数的第一句出现，第一句的this和super可以分布在同一个类的不同构造函数中。在调用this和super的时候实现程序的跳转，转而执行被调用的this构造函数或者super构造函数。

在super执行完毕之后，this执行完之前，程序转而执行在类定义的时候进行的变量初始化工作。这个执行完毕，才是构造函数中剩下的代码的执行。  
double weight = 2.3；

实际上会被分成如下2次执行：

（1）double weight;：创建Java对象时系统根据该语句为该对象分配内存。

（2）weight = 2.3;：这条语句将会被提取到Java类的构造器中执行。

继承成员变量和继承成员方法的区别：

|  |
| --- |
| package com.pattern.test;  public class FiledAndMethodTest {  public static void main(String[] args) {  Base bd = new Driver();  System.*out*.println(bd.count); *// 2 实例变量看声明类型（前半部分Base bd）* bd.display(); *// 20 引用方法看引用类型（后半部分new Driver()）* } }  class Base {  int count = 2;   public Base(){  System.*out*.println("super's count=" + this.count);  }   public void display() {  System.*out*.println(this.count);  }  }  class Driver extends Base {  int count = 20;   public Driver(){  System.*out*.println("Driver's count=" + this.count);  }   @Override  public void display() {  System.*out*.println(this.count);  } } |

解释，直接通过db访问count实例变量，输出的将是Base（声明时类型）对象的count实例变量的值；如果通过db来调用display()方法，该方法将表现出Derive（运行时类型）对象的行为方式。

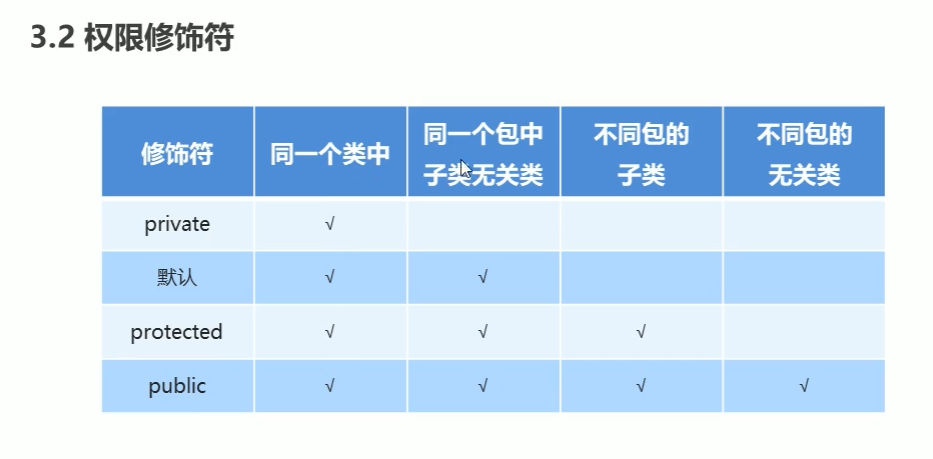
原因：如果子类重写了父类方法，就意味着子类里定义的方法彻底覆盖了父类里的同名方法，系统将不可能把父类的方法转移到子类中。对于实例变量则不存在这样的现象，即使子类中定义的与父类完全同名的实例变量，这个实例变量依然不可能覆盖父类中定义的实例变量。因为继承成员变量和继承方法之间存在这样的差异，所以对于一个引用类型的变量而言，当通过该变量访问它所引用的对象的实例变量时，该变量的值取决于声明该变量时类型；当通过该变量来调用它所引用的对象的方法时，该方法取决于它所实际引用的对象的类型。

首先应该知道，当程序创建一个子类对象时，系统不仅会为该类中定义的实例变量分配内存，也会为其父类中定义的所有实例变量分配内存，即使子类定义与父类中同名实例变量。也就是说，当系统创建一个Java对象的时候，如果该Java类有两个父类（一个直接父类A，一个间接父类B），假设A类中定义了2个实例变量，B类中定义了3个实例变量，当前类中定义了2个实例变量，那这个Java对象将会保存2+3+2个实例变量。

如果在子类里定义了与父类中已有变量同名的变量，那么子类中定义的变量会隐藏父类中定义的变量。注意不是完全覆盖，因此系统为创建子类对象时，依然会为父类中定义的、被隐藏的变量分配内存空间。为了在子类方法中访问父类中定义的、被隐藏的实例变量，或者为了在子类方法中调用父类中定义的、被覆盖（Override）的方法，可以通过super.作为限定来修饰这些实例变量和实例方法。

内部类访问的局部变量都要使用final修饰：

原因：对于普通局部变量，其作用域就是停留在该方法内，当方法执行结束，该局部变量也随之消失，但内部类则可能产生隐式的“闭包”，闭包将使得局部变量脱离它所在的方法继续存在。



**default在annotation中用来定义默认值：**

|  |
| --- |
| 默认值不能是null public @interface Test1 {  String value() default ""; } |

关于default的几种用法：



<? extends T> 表示类型的上界，表示参数化类型的可能是T 或是 T的子类;

<? super T> 表示类型下界（Java Core中叫超类型限定），表示参数化类型是此类型的超类型（父类型），直至Object;

List<? extends Father> 表示 “具有任何从Father继承类型的列表”，编译器无法确定List所持有的类型，所以无法安全的向其中添加对象。可以添加null,因为null 可以表示任何类型。所以List 的add 方法不能添加任何有意义的元素，但是可以接受现有的子类型List 赋值。

|  |
| --- |
| package com.pattern.test;  import java.util.LinkedList; import java.util.List;  public class test {  public static void main(String[] args) {  List<? extends Father1> list = new LinkedList<LeiFeng>();  list.add(null);  } }  class Human{ } class Father1 extends Human{ } class Son extends Father1{ } class LeiFeng extends Father1 { } |

List<? extends Father> list不能进行add，但是，这种形式还是很有用的，虽然不能使用add方法，但是可以在初始化的时候一个<? extends T>指定不同的类型。比如：

List<? extends Father> list1 = getFatherList();//getFatherList方法会返回一个Father的子类的list

另外，由于我们已经保证了List中保存的是Father类或者他的某一个子类，所以，可以用get方法直接获得值：

List<? extends Father> list1 = new ArrayList<>();

Father father = list1.get(0);//读取出来的东西只能存放在Father或它的基类里。

Object object = list1.get(0);//读取出来的东西只能存放在Father或它的基类里。

Human human = list1.get(0);//读取出来的东西只能存放在Father或它的基类里。

Son son = (Son)list1.get(0);

//super只能添加Father和Father的子类，不能添加Father的父类,读取出来的东西只能存放在Object类里

List<? super Father> list = new ArrayList<>();

list.add(new Father());

list.add(new Human());//compile error

list.add(new Son());

Father person1 = list.get(0);//compile error

Son son = list.get(0);//compile error

Object object1 = list.get(0);

因为下界规定了元素的最小粒度的下限，实际上是放松了容器元素的类型控制。既然元素是Father的基类，那往里存粒度比Father小的都可以。出于对类型安全的考虑，我们可以加入Father对象或者其任何子类（如Son）对象，但由于编译器并不知道List的内容究竟是Father的哪个超类，因此不允许加入特定的任何超类（如Human）。而当我们读取的时候，编译器在不知道是什么类型的情况下只能返回Object对象，因为Object是任何Java类的最终祖先类。但这样的话，元素的类型信息就全部丢失了。

**PECS原则：**

最后看一下什么是PECS（Producer Extends Consumer Super）原则，已经很好理解了：

* 频繁往外读取内容的，适合用上界Extends。
* 经常往里插入的，适合用下界Super。

**<? extends T>和<? super T>总结：**

* extends 可用于返回类型限定，不能用于参数类型限定（换句话说：? extends xxx 只能用于方法返回类型限定，jdk能够确定此类的最小继承边界为xxx，只要是这个类的父类都能接收，但是传入参数无法确定具体类型，只能接受null的传入）。
* super 可用于参数类型限定，不能用于返回类型限定（换句话说：? supper xxx 只能用于方法传参，因为jdk能够确定传入为xxx的子类，返回只能用Object类接收）。
* ? 既不能用于方法参数传入，也不能用于方法返回。

**<? extends SomeClass>与<T extends SomeClass>的区别：**

首先是名词对应表，不需要记住右边的名字，但需要知道左边的各种用法

List<String> —- 参数化的类型   
List<E> —- 泛型   
List<?> —- 无限制通配符类型   
<E extends SomeClass> —- 有限制类型参数   
List <? extends SomeClass>—- 有限制通配符类型   
<T extends Comparable<T>> —– 递归类型限制   
static <E> List<E> asList(E[] a) —- 泛型方法

**<E extends ClassA> 与 <? extends ClassA>有什么区别？**

答：当我第一次接触这两名词时，感觉他们的功能是一样的，T可以代表任意的子类，?也可以代表任意的子类。   
首先我们明确一下两边的名字，限制类型 & 通配符类型，<E extends ClassA>表示后续都只能使用E进行某些判断或操作，而<? extends ClassA>？表示后续使用时可以是任意的。   
举个<E extends ClassA>最常见的例子，用于比较操作，比如返回“最大值”，“最大值”的定义为：整型、浮点型返回最大值，字符串返回字典序最大者，由于想调用compareTo函数，我们让所有参数都继承Compareble,即T extends Comparable<T>，整个测试代码如下：

package test;

/\*\*

\* 定义了 <T extends someClass>,

\* 里面的代码便只能用somClass的子类T进行比较或其他操作。

\*/

public class MaximumTest {

// determines the largest of three Comparable objects

public static <T extends Comparable<T>> T maximum(T x, T y, T z) {

T max = x; // assume x is initially the largest

if ( y.compareTo( max ) > 0 ) {

max = y; // y is the largest so far

}

if ( z.compareTo( max ) > 0 ) {

max = z; // z is the largest now

}

return max; // returns the largest object

}

public static void main(String args[]) {

System.out.printf( "Max of %d, %d and %d is %d\n\n",

3, 4, 5, maximum( 3, 4, 5 ) );

System.out.printf( "Maxm of %.1f,%.1f and %.1f is %.1f\n\n",

6.6, 8.8, 7.7, maximum( 6.6, 8.8, 7.7 ) );

System.out.printf( "Max of %s, %s and %s is %s\n","pear",

"apple", "orange", maximum( "pear", "apple", "orange" ) );

}

}

注释里已经写清楚了，我们只能用T类型来进行一些操作，我们不能把T替换成?，因为？并不是一个类名，它只是一个通配符，然后举个<? extends ClassA>的例子。

比如我们有一个Stack类，类里提供一个pullAll方法，我们想把一系列元素全部放到堆栈中，如下方法：

// Stack定义

public class Stack<E> {

public Stack();

public E pop();

public boolean isEmpty();

}

// ...

public <E> void pushAll(Iterable<E> src) {

for(E e : src) {

push(e);

}

}

这个方法编译时没问题，Iterable src的元素类型与堆栈的类型完全匹配就没有问题。但是假如有一个Stack<Number>调用了push(intVal)，这里的intVal是Integer类型，这是可以的，因为Integer是Number的一个子类型，但下面的代码会报编译错误，

Stack<Number> numberStack = new Stack<Number>();

Iterable<Integer> integers = "...";

numberStack.pushAll(integers);

因为在Java中，参数化类型是不可变的。所以现在我们的通配符类型就派上用场了，代码如下

public void pushAll(Iterable<? extends E> scr) {

for( E e : src) {

push(e);

}

}

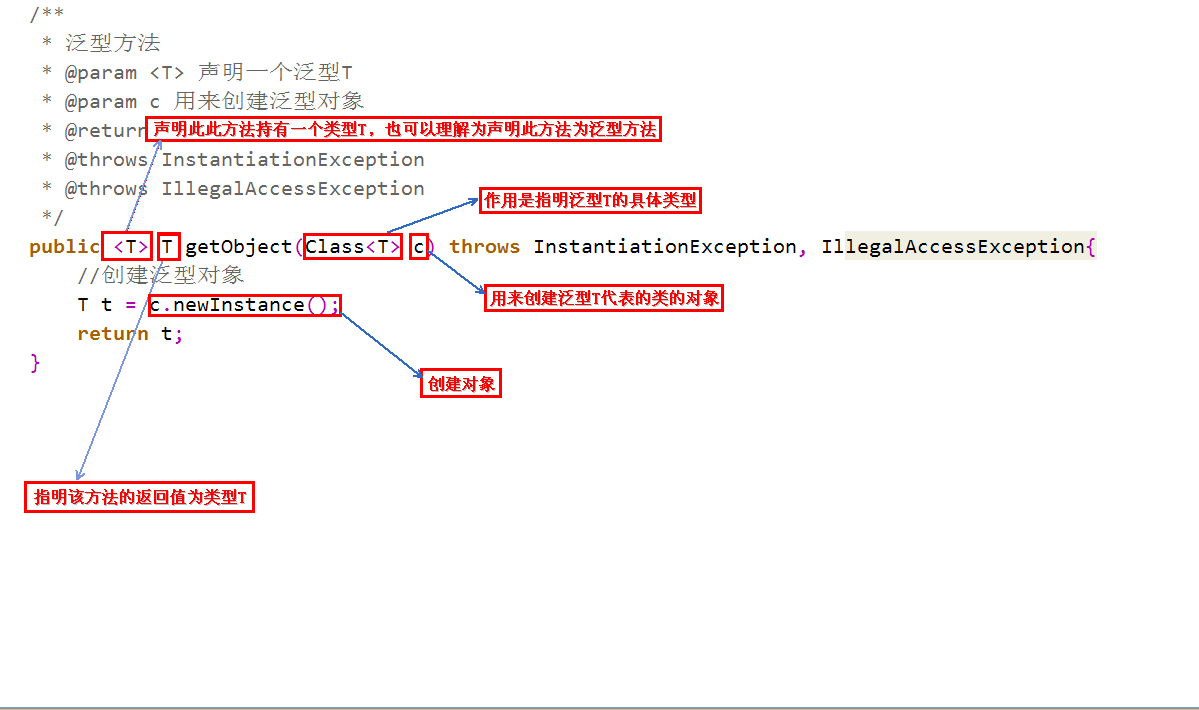
此处就必须用通配符?，代表泛型的泛指“E的某个子类型的Iterator接口”。

泛型是什么意思在这就不多说了，而Java中泛型类的定义也比较简单，例如：public class Test<T>{}。这样就定义了一个泛型类Test，在实例化该类时，必须指明泛型T的具体类型，例如：Test<Object> t = new Test<Object>();，指明泛型T的类型为Object。

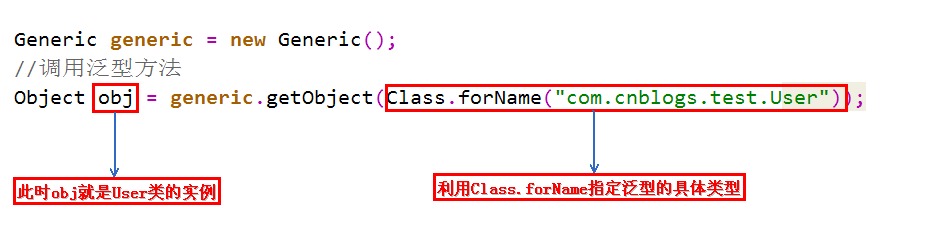
但是Java中的泛型方法就比较复杂了。

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

**定义泛型方法**语法格式如下：



**调用泛型方法**语法格式如下：



说明一下，定义泛型方法时，必须在返回值前边加一个<T>，来声明这是一个泛型方法，持有一个泛型T，然后才可以用泛型T作为方法的返回值。

       Class<T>的作用就是指明泛型的具体类型，而Class<T>类型的变量c，可以用来创建泛型类的对象。

       为什么要用变量c来创建对象呢？既然是泛型方法，就代表着我们不知道具体的类型是什么，也不知道构造方法如何，因此没有办法去new一个对象，但可以利用变量c的newInstance方法去创建对象，也就是利用反射创建对象。

       泛型方法要求的参数是Class<T>类型，而Class.forName()方法的返回值也是Class<T>，因此可以用Class.forName()作为参数。其中，forName()方法中的参数是何种类型，返回的Class<T>就是何种类型。在本例中，forName()方法中传入的是User类的完整路径，因此返回的是Class<User>类型的对象，因此调用泛型方法时，变量c的类型就是Class<User>，因此泛型方法中的泛型T就被指明为User，因此变量obj的类型为User。

       当然，泛型方法不是仅仅可以有一个参数Class<T>，可以根据需要添加其他参数。

       为什么要使用泛型方法呢？因为泛型类要在实例化的时候就指明类型，如果想换一种类型，不得不重新new一次，可能不够灵活；而泛型方法可以在调用的时候指明类型，更加灵活。

注解（Annontion）是Java5开始引入的新特征。它提供了一种安全的类似注释的机制，用来将任何的信息或元数据（metadata）与程序元素（类、方法、成员变量等）进行关联。

Javac负责将Java源代码编译为字节码文件；

解可以标记在包、类、属性、方法，方法参数以及局部变量上，且同一个地方可以同时标记多个注解。

除了直接使用JDK 定义好的注解，我们还可以自定义注解，在JDK 1.5中提供了4个标准的用来对注解类型进行注解的注解类，我们称之为 meta-annotation（元注解），他们分别是：

@Target

@Retention

@Documented

@Inherited

**@Target注解**

Target注解的作用是：描述注解的使用范围（即：被修饰的注解可以用在什么地方） 。

Target注解用来说明那些被它所注解的注解类可修饰的对象范围：注解可以用于修饰 packages、types（类、接口、枚举、注解类）、类成员（方法、构造方法、成员变量、枚举值）、方法参数和本地变量（如循环变量、catch参数），在定义注解类时使用了@Target 能够更加清晰的知道它能够被用来修饰哪些对象，它的取值范围定义在ElementType 枚举中。

|  |
| --- |
| public enum ElementType {   *TYPE*, *// 类、接口、枚举类   FIELD*, *// 成员变量（包括：枚举常量）   METHOD*, *// 成员方法   PARAMETER*, *// 方法参数   CONSTRUCTOR*, *// 构造方法   LOCAL\_VARIABLE*, *// 局部变量   ANNOTATION\_TYPE*, *// 注解类   PACKAGE*, *// 可用于修饰：包   TYPE\_PARAMETER*, *// 类型参数，JDK 1.8 新增   TYPE\_USE // 使用类型的任何地方，JDK 1.8 新增* } |

**@Retention注解**

Reteniton注解的作用是：描述注解保留的时间范围（即：被描述的注解在它所修饰的类中可以被保留到何时） 。

Reteniton注解用来限定那些被它所注解的注解类在注解到其他类上以后，可被保留到何时，一共有三种策略，定义在RetentionPolicy枚举中。

RetentionPolicy.SOURCE：注解只保留在源码中，在编译时会被编译器丢弃

RetentionPolicy.CLASS：(默认的保留策略) 注解会被保留在Class文件中，但不会被加载到虚拟机中，运行时无法获得

RetentionPolicy.RUNTIME：注解会被保留在Class文件中，且会被加载到虚拟机中，可以在运行时获得

**@Documented**

用于将注解包含在javadoc中

默认情况下，javadoc是不包括注解的，但如果使用了@Documented注解，则相关注解类型信息会被包含在生成的文档中

用于描述其它类型的annotation应该作为被标注的程序成员的公共API，因此可以被例如javadoc此类的工具文档化。它是一个标记注解，没有成员。

**@Inherited**

用于指明父类注解会被子类继承得到

|  |
| --- |
| @Target(ElementType.TYPE) @Retention(RetentionPolicy.RUNTIME) @Inherited public @interface ParentAnnotation {   String key();   String value(); }  @ParentAnnotation(key = "key", value = "value") public class Parent {  }  public class Son extends Parent {   public static void main(String[] args) {  Annotation[] annotations = Son.class.getAnnotations();  *// [@com.ParentAnnotation(key=key, value=value)]* System.*out*.println(Arrays.toString(annotations));  } } |

**@Repeatable**

用于声明标记的注解为可重复类型注解，可以在同一个地方多次使用

|  |
| --- |
| @Target(ElementType.TYPE) @Retention(RetentionPolicy.RUNTIME) @Repeatable(RepeatableAnnotation.class) public @interface TestAnnotation {   String key();   String value(); }  @Target(ElementType.TYPE) @Retention(RetentionPolicy.RUNTIME) @interface RepeatableAnnotation {  TestAnnotation[] value(); }  @TestAnnotation(key = "key1", value = "value1") @TestAnnotation(key = "key2", value = "value2") public class Test {   public static void main(String[] args) {  RepeatableAnnotation annotation = Test.class.getAnnotation(RepeatableAnnotation.class);  TestAnnotation[] testAnnotations = annotation.value();  for (TestAnnotation testAnnotation : testAnnotations) {  *// @com.TestAnnotation(key=key1, value=value1)  // @com.TestAnnotation(key=key2, value=value2)* System.*out*.println(testAnnotation);  }  } } |

**@interface 是用来修饰 Annotation （注解）的，它不是 interface。这个关键字声明隐含了一个信息：它是继承了java.lang.annotation.Annotation 接口，而不是声明了一个 interface。**

关于包扫描的介绍.(@ComponentScan)

在以往采用配置文件,配置扫描包时,会使用<context:component-scan base-package="需要扫描的包名前缀">,即可,之后Spring会自动加载类上带有@Controller,@Service,@Repository,@Component的类;

在Spring注解开发中,只需要在主配置类上添加@ComponentScan(value="需要扫描的包名前缀")注解即可;示例代码如下：

|  |
| --- |
| @Configuration*//标记此类为配置类* @ComponentScan(value="com.cnblogs") public class MainConfig {  } |

如果想排除或者只包含指定类型的文件,可使用incloudFiltes和excludeFilters属性进行配置.

1:excludeFilters属性可以接收数组类型,数组里面存放的是@Filter注解入:

|  |
| --- |
| @Configuration*//标记此类为配置类* @ComponentScan(value="com.cnblogs",  excludeFilters= { @Filter(type=FilterType.ANNOTATION,classes= {Controller.class}) })  public class MainConfig {  } |

以上代码,指定容器中排除加载带有@Controller注解的类.@Filter注解的type属性,注明按照什么方式进行过滤.常用的为FilterType.ANNOTATION(按照注解的方式进行过滤).class属性中存放需要排除的bean.这里只是初步的对@Filter注解进行介绍.下一篇中会有详细的介绍;

2,includeFilters属性,让容器实现只加指定的类.这里需要说一下@ComponentScan注解默认会加载所有带有@Controller,@Service,@Repository,@Component的类到容器中,所以要想实现加载指定的类,需要把@ComponentScan注解的默认加载所有类关掉.即useDefaultFilters = false. 代码示例如下:

|  |
| --- |
| @Configuration*//标记此类为配置类* @ComponentScan(value="com.cnblogs",includeFilters= {  @Filter(type=FilterType.ANNOTATION,classes= {Controller.class}) },useDefaultFilters = false) public class MainConfig {  } |

以上代码为只加载带有@Controller注解的类

注:如果使用的是jdk1.8,@ComponentScan可以在一个类上配置多个:

|  |
| --- |
| @Configuration*//标记此类为配置类* @ComponentScan(...) @ComponentScan(...) public class MainConfig {  } |

如果不是jdk1.8可使用@ComponentScans注解里面配置多个@ComponentScan,代码如下:

|  |
| --- |
| @Configuration*//标记此类为配置类* @ComponentScans(  value={  @ComponentScan(...),@ComponentScan(...)  } ) public class MainConfig {  } |

上面主要针对@ComponentScan注解做了一些说明,本文主要对@Filter的扫描条件,再做一些详细的介绍

1,FilterType.ANNOTATION 按照注解的方式进行扫描.后面classes属性,为注解的类型,如:

|  |
| --- |
| @Configuration*//标记此类为配置类* @ComponentScan(value="com.wxj",excludeFilters= {  @Filter(type=FilterType.ANNOTATION,classes= {Controller.class}) }) public class MainConfig {  *//将标有@Controller注解的类排除在外不会加载到容器中来* } |

2,type=FilterType.ASSIGNABLE\_TYPE,按照指定的类,进行过滤,后面的classes属性的值为"类名.class".如:

|  |
| --- |
| @Configuration*//标记此类为配置类* @ComponentScan(value="com.wxj",  includeFilters= {  @Filter(type=FilterType.ASSIGNABLE\_TYPE,classes= {BookService.class})  },useDefaultFilters=false) public class MainConfig {  *//只会加载BookService,以及BookService的子类或者其实现类* } |

3,FilterType.CUSTOM,按照自己自定义的方式来进行过滤和筛选(使用此过滤类型,虽然比较繁琐,但是使用起来完全可以由自己来定义扫描的规则)

首先定义@Filter注解的类型

|  |
| --- |
| @Configuration*//标记此类为配置类* @ComponentScan(value="com.wxj",  includeFilters= { @Filter(type=FilterType.CUSTOM,classes{MyTypeFilter.class})  },useDefaultFilters=false) public class MainConfig { } |

MyTypeFilter即为自己定义的匹配方法,其中MyTypeFilter类中的match方法的返回值为true时,为符合过滤条件,如果返回为false,则不符合过滤条件,代码如下:

|  |
| --- |
| public class MyTypeFilter implements TypeFilter {   */\*\*  \* MetadataReader metadataReader:获取当前正在扫描的类的信息  \* MetadataReaderFactory metadataReaderFactory,获取带其他任何类的信息  \*  \*/* public boolean match(MetadataReader metadataReader, MetadataReaderFactory metadataReaderFactory) throws IOException {  AnnotationMetadata annotationMetadata = metadataReader.getAnnotationMetadata();*//获取当前类的注解信息* ClassMetadata classMetadata = metadataReader.getClassMetadata();*//获取当前扫描的类信息* Resource resource = metadataReader.getResource();*//获取当前扫描的资源信息* String name = classMetadata.getClassName();*//获取类的名字* if(name.contains("er")) {  return true;*//如果类的名字中带有"er",则符合过滤的要求* }  return false;  } } |

使用@Configuration 来注解类表示类可以被 Spring 的 IOC 容器所使用

@ComponentScan根据定义的扫描路径，把符合扫描规则的类装配到spring的bean容器中。

**表达式中的陷阱：**

1、StringBuilder和StringBuffer

进行字符串拼接时，通常应该优先考虑StringBuilder，StringBuilder和StringBuffer唯一的区别在于，StringBuffer是线程安全的，也就是说StringBuffer类里绝大数方法都增加了synchronized修饰符，以使得保证该方法线程安全，但会降低该方法的执行效率。在没有多线程的环境下，应该优先使用StringBuilder来表示字符串。

2、复合赋值运算符的陷阱

复合赋值运算符包含了一个隐式的类型转换，也就是说，下面两句并不等价。

a = a + 5;

a += 5;

实际上，a += 5;等价于a = (a的类型)(a + 5);，也就是说，复合赋值运算符会自动将它计算的结果值强制类型转换成其左侧变量的的类型。如果结果的类型与该变量的类型相同，那么这个转型不会造成任何影响。如果结果值的类型比该变量的类型要大，那么复合赋值运算符将会执行一次强制类型转换，这个强制类型转换将有可能导致高位“截断”，如下程序：

|  |
| --- |
| public class CompositeAssign {   public static void main(String[] args) {  short st = 5;  st += 10;*// 没有任何问题，系统执行隐式的类型转换* System.*out*.println(st);  st += 90000;*// 出现问题，隐式类型转换，将会发生溢出* System.*out*.println(st);  }  } |

执行第一个+=后，st的值是15，执行第二个+=会引起高位“截断”，第二个+=行代码相当于：

st = (short)(st + 90000);，即st = (short)90015;。问题是：short类型的变量只能接受-32768—32767之间的整数，因此上面程序会将90015高位“截断”，程序最后输出是24479。

容易导致计算结果被高位“截断”的情况有以下几种，需要注意：

将复合赋值运算符运用于btye，short或char等类型的变量。

将复合赋值运算符运用于int类型的变量，而表达式右侧是long、float、double类型的值。

将复合赋值运算符运用于float类型的变量，而表达式右侧是double类型的值。

3、多线程的陷阱

（1）线程的创建和启动

从JDK1.5开始，Java提供了3种方式创建、启动多线程：

继承Thread类来创建线程类，重写run()方法作为线程执行体；

实现Runnable接口来创建线程类，重写run()方法作为线程执行体；

实现Callable接口来创建线程类，重写call()方法作为线程执行体。

其中，第一种方式最差，它有两种不足：

线程类继承了Thread类，无法继承其他类。

因为每条线程都是一个Thread子类的实例，因此多个线程之间共享数据比较麻烦。

对于第二种、第三种方式，它们的本质是一样的，只是Callable接口里包含的call()方法既可以声明抛出异常，也可以拥有返回值。

（2）静态的同步方法

根据java虚拟机规范，所有java虚拟机实现必须在每个类或接口被java程序首次**主动使用**时才初始化。

主动使用有以下6种：  
1) 创建类的实例  
2) 访问某个类或者接口的静态变量，或者对该静态变量赋值（如果访问静态编译时常量(即编译时可以确定值的常量)不会导致类的初始化）  
3) 调用类的静态方法  
4) 反射（Class.forName(xxx.xxx.xxx)）  
5) 初始化一个类的子类（相当于对父类的主动使用），不过直接通过子类引用父类元素，不会引起子类的初始化  
6) Java虚拟机被标明为启动类的类（包含main方法的）

类与接口的初始化不同，如果一个类被初始化，则其父类或父接口也会被初始化，但如果一个接口初始化，则不会引起其父接口的初始化。

|  |
| --- |
| class Toy {  static {  System.*out*.println("Initializing");*// 静态子句，只在类第一次被加载并初始化时执行一次，而且只执行一次* }  public static final int price=10;  Toy() {  System.*out*.println("Building");*// 构造方法，在每次声明新对象时加载* } }  public class TestToy {  public static void main(String[] args) {  *// try {  // Class c = Class.forName("myblog.classloader.Toy");  // } catch (ClassNotFoundException e) {  // e.printStackTrace();  // }* Class c = Toy.class; *// 不会输出任何值* } } |

用**类字面常量** .class和Class.forName都可以创建对类的应用，但是不同点在于，用Gum.class创建Class对象的应用时，**不会自动初始化**该Class对象（static子句不会执行）

如果该main方法是直接写在Toy类中，那么调用Toy.class，会引起初始化，并输出Initializing，原因并不是Toy.class引起的，而是该类中含有启动方法main，该方法会导致Toy的初始化。

使用Toy.class是在编译期执行的，因此在编译时必须已经有了Toy的.class文件，不然会编译失败，这与 Class.forName("myblog.classloader.Toy")不同，后者是运行时动态加载。

**编译时常量**。回到完整的类Toy，如果直接输出：System.out.println(Toy.price)，会发现static子句和构造方法都没有被执行，这是因为Toy中，常量price被static final限定，这样的常量叫做**编译时常量**，对于这种常量，不需要初始化就可以读取。编译时常量必须满足3个条件：static的，final的，常量。

每个类编译后产生一个Class对象，存储在.class文件中，JVM使用类加载器（Class Loader）来加载类的字节码文件（.class）。

java 类和对象的初始化

**Java类的装载**

  在Java中，类装载器把java类装载到虚拟机中，经过装载，链接和初始化三个步骤来完成。

其中链接中包括 校验、准备和解析。

下面对这些概念进行解析：

装载：查找和导入类或接口的二进制数据，常用的是根据类的路径加载，还有根据网络的地址加载。

链接：执行校验、准备和解析步骤，其中解析步骤是可以选择的；

      校验：检查导入类或接口的二进制数据的正确性；

      准备：给类的静态变量分配并初始化存储空间；

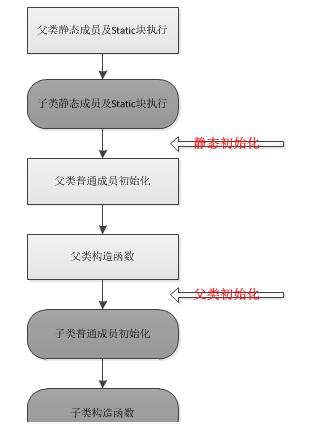
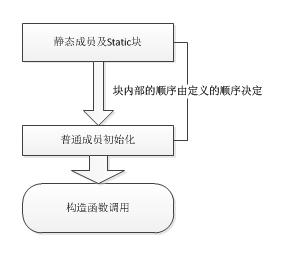
      解析：将符号引用转成直接引用；

初始化：\*\*类的静态变量的初始化Java代码和静态Java代码块。

其中 初始化（initialization）包含两部分：

1.类的初始化（initialization class & interface）---初始化静态变量和静态Java代码块

2.对象的创建（creation of new class instances）----new class()。



**初始化**

1.类的初始化，也就是静态变量以及静态块的初始化

    1.1 如果有继承父类有静态变量或者静态块的，先执行初始化。

    1.2 如果是final static ,那么就不会初始化类

1.3 如果执行的是父类的静态变量，那么子类不会初始化

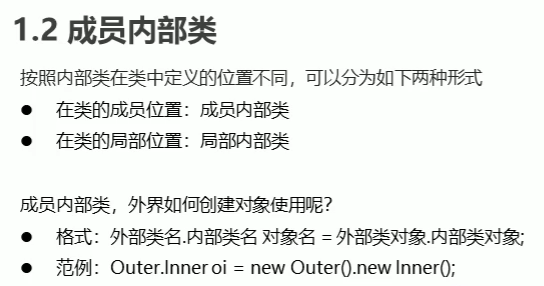
|  |
| --- |
| public class SuperTmp {  public static int *a* = 10;  public final static int *b* = 20;  public int c = 10;  static{  System.*out*.println("init super calss...");  }  public SuperTmp (){  System.*out*.println("super init constructor:");  } } public class Sub extends SuperTmp {  public static final int *aa* = 30;  public static int *bb* = 40;  public int cc = 20;  static {  System.*out*.println("init sub class.....");  }  public Sub (){   System.*out*.println("sub init constructor:" );  } } public class Play {  public static void main(String[] args) {  */\*\*输出  \* init super calss...  \* 10  \* 不会初始化子类 Sub  \*/* System.*out*.println(Sub.*a*);  */\*\*\*\*\*\*  \* 输出 b 不会初始化任何类  \*/* System.*out*.println(Sub.*b*);  */\*\*\*  \* 输出  \* init super calss...  \* init sub class.....  \* 40  \*/* System.*out*.println(Sub.*bb*);  */\*对象初始化  \*init super calss...  init sub class.....  \*super init constructor  sub init constructor  \*/* Sub sb = new Sub()  } } |

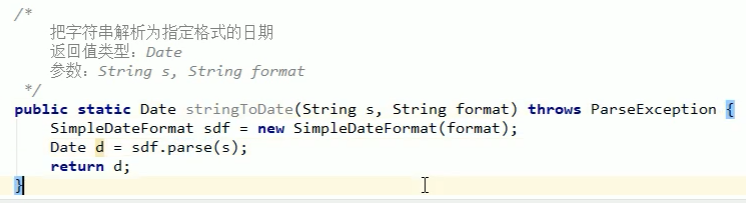
2.创建class的对象

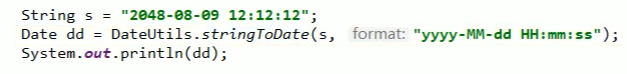
    2.1.先为Sub类和其父类SuperTmp类分配内存空间,父类和子类的变量都初始化值，对象类型的为null，基本数据类型的为默认的值，例如 int 类型的为初始化 0

   2.2  调用构造函数Sub()(不执行), 调用父类的构造函数(不执行)，先初始化superTmp的成员变量,，再执行父类构造函数里面的语句输出“super init constructor”，接着初始化Sub()类的成员变量，再执行Sub()构造函数，输出“sub init constructor”。

在使用引用类型之前，最好判断一下这个引用是否为null





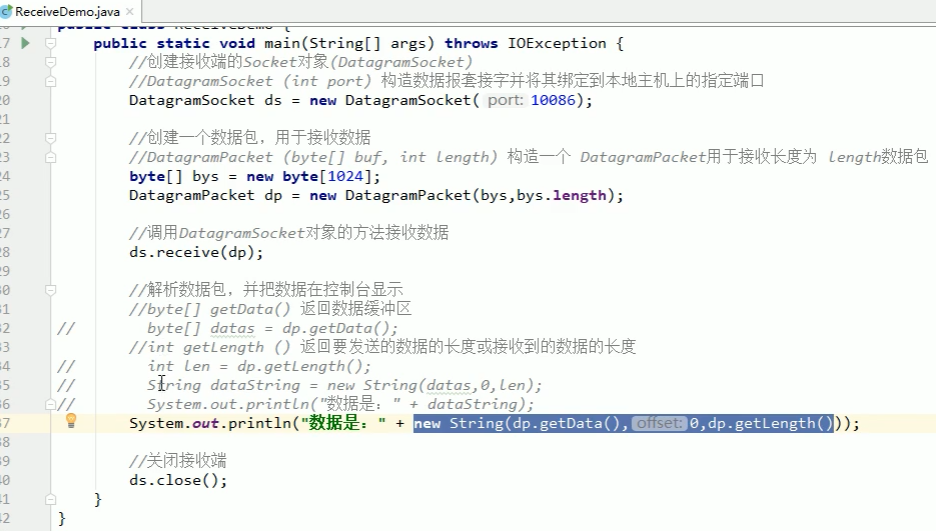




网络编程：







TCP通信程序：









1. 被transient修饰的变量不参与序列化和反序列化

2. transient一般在实现了Serializable接口的类中使用**。**

