**方法的静态、动态链接**：

1、在jvm中，将**符号引用（常量池中的#数字）**转换为调用方法的**直接引用（具体的值就是#数字后面的值 ）**与方法的绑定机制相关。

2、静态链接：（比如我创建一个类的实例，我能明确的知道我创建的哪个类实例，那么这代码一写了，那就是死代码，不会变了，所以后面调用方法我能直接知道实体在哪里，）

当一个字节码文件被装载进JVM内部时，如果被调用的目标方法**在编译期可知**，且运行期间保持不变时。这种情况下将调用方法的符号引用转换为直接引用的过程称之为静态链接。

3、动态链接：（比如上转型对象配上工厂方法，那你不可能知道具体创建的是什么对象，但是我偏偏能通过上转型进行方法的调用，那只有在具体运行时才去找具体的方法实体）

如果被调用的方法在**编译期无法被确定下**来，也就是说，只能够在程序运行期将调用方法的符号引用转换为直接引用由于这种引用转换过程具备动态性，因此也就被称之为动态链接。

**静态链接与动态链接对应的绑定机制**：（也就是说在编译期间会确定方法调用的绑定机制，根据绑定的机制可以确定常量池中符号引用以及引用了常量池中符号引用的引用转换为直接引用的方式。）

1、方法的绑定机制为：早期绑定（Early Binding）和晚期绑定（Late Binding）。**绑定是一个字段、方法 或者类在符号引用被替换为直接引用的过程，这仅仅发生一次。（也就是动静态链接只是针对与方法，而绑定范围更广，所以绑定方式确定方法的链接方式）**

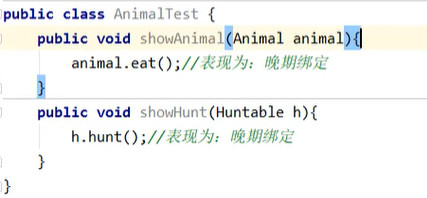
2、早期绑定：

早期绑定就是指被调用的目标方法如果在编译期可知，且运行期保持不变时，即可将这个方法与所属的类型进行绑定，这样一来，由于明确了被调用的目标方法究竟时哪一个，因此也就可以使用静态链接的方法将符号引用转换为直接引用。

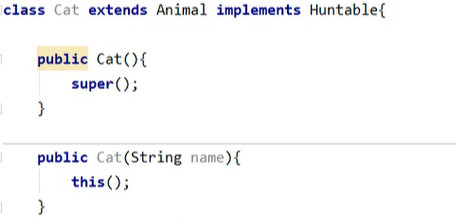
3、晚期绑定：

如果被调用的方法在编译期无法被确定下来，只能够在程序运行期间根据实际的类型绑定相关的方法，这种绑定方式也就被称之为晚期绑定。

晚期绑定：



早期绑定：



（

注意：（**面向过程的语言只有早期绑定**）

1、随着高级语言的横空出世，类似于Java一样的基于面向对象的编程语言越来越多，尽管这类编程语言在语法风格上存在差异，但是他们的理念是由共性的，呢就是都支持封装、继承和多态（允许父类指针指向子类实例）等面向对象的特性，**既然这一类的编程语言具备多态的特性，那么自然也就具备早期绑定和晚期绑定两种绑定方式。**

2、Java中任何一个普通的方法其实都具备虚函数的特征，他们相当于C++语言的虚函数（C++中则需要使用关键字virtual来显式定义）。如果在Java程序中不希望某个方法拥有虚函数的特征时，则可以使用，关键字final来标记这个方法。

）

**虚方法与非虚方法：**

**非虚方法：（不涉及到多态的使用）**

1、如果方法**在编译期**就确定了具体的调用版本，这个版本在运行时是不可变的，这样的方法称为非虚方法。

2、静态方法、私有方法、final方法、实例构造器、super、this直接调用都是非虚方法。



3、其他方法称为虚方法。

JVM中提供的方法调用指令：

一、普通调用指令：

1、**invokestatic：调用静态方法，解析阶段确定唯一方法版本**。

2、**invokespecial：调用<init>方法、私有及父类方法，解析阶段确定唯一方法版本**

3、invokevirtual：调用所有的虚方法

4、invokeinterface：调用接口方法。

二、动态调用指令：(jdk1.7)

1、invokedynamic：动态解析出需要调用的方法、然后执行。

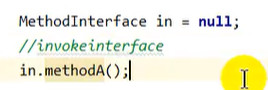
（注意：

前四条指令固化在虚拟机内部，方法的调用执行不可人为干预，而invokedynamic指令则支持由用户确定方法版本，其中**invokestatic指令和invokespecial指令调用的方法称为非虚方法,但是final修饰的方法却不是由他们来调用的**，其余的，final修饰的除外称为虚方法。

）







**关于动态语言与静态语言**：（想想js也是一种弱类型语言）

1、静态语言是指在编译时变量的数据类型即可确定的语言，多数静态类型语言要求在使用变量之前必须声明数据类型。例如：C++、Java、Delphi、C#等。  
  
2、动态语言是在运行时确定数据类型的语言。变量使用之前不需要类型声明，通常变量的类型是被赋值的那个值的类型。例如：Python、Ruby、Perl等。

**关于invokedynamic指令：**

1、jvm字节码指令集一直比较稳定，一直到Java7中才增加了一个invokedynamic指令，这是Java为了实现【动态类型语言】支持而做的一种改进。

2、但是在Java7中并没有提供直接生成invokedynamic指令的方法，需要借助ASM这种底层字节码工具来产生invokedynamic指令。直到Java8的Lambda白哦大是的出现，invokedynamic指令的生成，在Java中才有了直接的生成方式。

3、Jav7中增加的动态语言类型支持的本质是对Java虚拟机规范的修改，而不是对Java语言规则的修改，这一块相对来讲比较复杂，增加了虚拟机中的方法调用，最直接的受益者就是运行在Java平台的动态语言的编译器。

**Java语言中重写的本质**：（这里的描述还没有涉及到多态上转型问题）

1、找到操作数栈顶的第一个元素所执行的对象的实际类型，记作C。

（**问题1**：**如果是在多态情况下，这个是怎么知道实际类型的？**

**问题2：执行方法时，我们是通过对象的引用进行方法的调用，但是对象的引用仅仅只是指向的堆区的中属性地址，并没有指向Class的方法？**

**答：我们来理一理方法中调用另一个对象的方法的过程：**

**方法的指令在编译期就已经确定好了，比如我声明了**

）

2、如果在**类型C**中找到与常量中描述相符合简单名称都相符的方法，则进行访问权限校验，如果通过则返回这个方法的直接引用，查找过程结束；如果不通过，则返回java.lang.illegalAccessError异常。

3、否则，按照继承关系从下往上依次对C的各个父类进行第2步的搜索和验证过程。如果始终没有找到合适的方法，则抛出java.lang.AbstractMethodError异常。

(

illegalAccessError:程序试图访问或修改一个属性或调用一个方法，这个属性或方法，你没有权限访问。一般的，这个会引起编译器异常。这个错误如果发生在运行时，就说明一个类发生了不兼容的改变。

)