多态

1.将一个方法调用同一个方法主体关联起来叫做绑定，若在程序执行前绑定称为前期绑定或静态绑定；而程序运行时才绑定的称为动态绑定或运行时绑定；

在java中除了static和final（private是属于final）都是后期绑定；所以final的用处就是在于关闭了动态绑定，不需要覆盖；

2.只有普通方法的调用才有多态，而域或者说常量是没有多态的；所有对于域、static、final修饰的来说看类型不看引用；

3.在构造器里唯一能安全调用的那些方法是基类中final方法，这些方法不能被覆盖，不会出现变量为初始化问题；

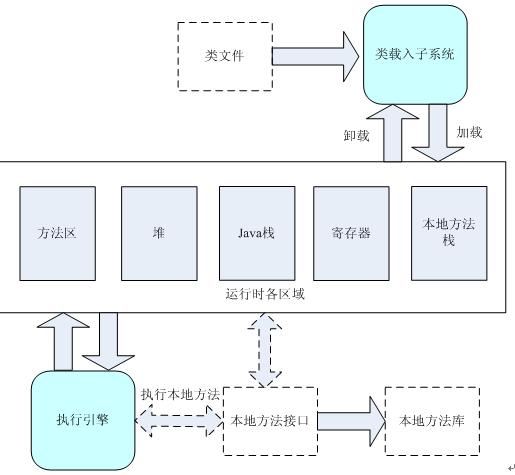
**Java 的实现方式**

Java 对于方法调用动态绑定的实现主要依赖于方法表，但通过类引用调用和接口引用调用的实现则有所不同。总体而言，当某个方法被调用时，JVM 首先要查找相应的常量池，得到方法的符号引用，并查找调用类的方法表以确定该方法的直接引用，最后才真正调用该方法。以下分别对该过程中涉及到的相关部分做详细介绍。

**JVM 的结构**

典型的 Java 虚拟机的运行时结构如下图所示

**图 1.JVM 运行时结构**



此结构中，我们只探讨和本文密切相关的方法区 (method area)。当程序运行需要某个类的定义时，载入子系统 (class loader subsystem) 装入所需的 class 文件，并在内部建立该类的类型信息，这个类型信息就存贮在方法区。类型信息一般包括该类的方法代码、类变量、成员变量的定义等等。可以说，类型信息就是类的 Java 文件在运行时的内部结构，包含了改类的所有在 Java 文件中定义的信息。

注意到，该类型信息和 class 对象是不同的。class 对象是 JVM 在载入某个类后于堆 (heap) 中创建的代表该类的对象，可以通过该 class 对象访问到该类型信息。比如最典型的应用，在 Java 反射中应用 class 对象访问到该类支持的所有方法，定义的成员变量等等。可以想象，JVM 在类型信息和 class 对象中维护着它们彼此的引用以便互相访问。两者的关系可以类比于进程对象与真正的进程之间的关系。

**Java 的方法调用方式**

Java 的方法调用有两类，动态方法调用与静态方法调用。静态方法调用是指对于类的静态方法的调用方式，是静态绑定的；而动态方法调用需要有方法调用所作用的对象，是动态绑定的。类调用 (invokestatic) 是在编译时刻就已经确定好具体调用方法的情况，而实例调用 (invokevirtual) 则是在调用的时候才确定具体的调用方法，这就是动态绑定，也是多态要解决的核心问题。

JVM 的方法调用指令有四个，分别是 invokestatic，invokespecial，invokesvirtual 和 invokeinterface。前两个是静态绑定，后两个是动态绑定的。本文也可以说是对于 JVM 后两种调用实现的考察。

**常量池（constant pool）**

常量池中保存的是一个 Java 类引用的一些常量信息，包含一些字符串常量及对于类的符号引用信息等。Java 代码编译生成的类文件中的常量池是静态常量池，当类被载入到虚拟机内部的时候，在内存中产生类的常量池叫运行时常量池。

常量池在逻辑上可以分成多个表，每个表包含一类的常量信息，本文只探讨对于 Java 调用相关的常量池表。

CONSTANT\_Utf8\_info

字符串常量表，该表包含该类所使用的所有字符串常量，比如代码中的字符串引用、引用的类名、方法的名字、其他引用的类与方法的字符串描述等等。其余常量池表中所涉及到的任何常量字符串都被索引至该表。

CONSTANT\_Class\_info

类信息表，包含任何被引用的类或接口的符号引用，每一个条目主要包含一个索引，指向 CONSTANT\_Utf8\_info 表，表示该类或接口的全限定名。

CONSTANT\_NameAndType\_info

名字类型表，包含引用的任意方法或字段的名称和描述符信息在字符串常量表中的索引。

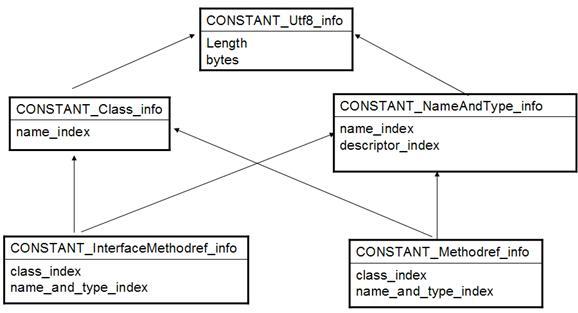
CONSTANT\_InterfaceMethodref\_info

接口方法引用表，包含引用的任何接口方法的描述信息，主要包括类信息索引和名字类型索引。

CONSTANT\_Methodref\_info

类方法引用表，包含引用的任何类型方法的描述信息，主要包括类信息索引和名字类型索引。

**图 2. 常量池各表的关系**



可以看到，给定任意一个方法的索引，在常量池中找到对应的条目后，可以得到该方法的类索引（class\_index）和名字类型索引 (name\_and\_type\_index), 进而得到该方法所属的类型信息和名称及描述符信息（参数，返回值等）。注意到所有的常量字符串都是存储在 CONSTANT\_Utf8\_info 中供其他表索引的。

**方法表与方法调用**

方法表是动态调用的核心，也是 Java 实现动态调用的主要方式。它被存储于方法区中的类型信息，包含有该类型所定义的所有方法及指向这些方法代码的指针，注意这些具体的方法代码可能是被覆写的方法，也可能是继承自基类的方法。

如有类定义 Person, Girl, Boy,

**清单 1**

class Person {

public String toString(){

return "I'm a person.";

}

public void eat(){}

public void speak(){}

}

class Boy extends Person{

public String toString(){

return "I'm a boy";

}

public void speak(){}

public void fight(){}

}

class Girl extends Person{

public String toString(){

return "I'm a girl";

}

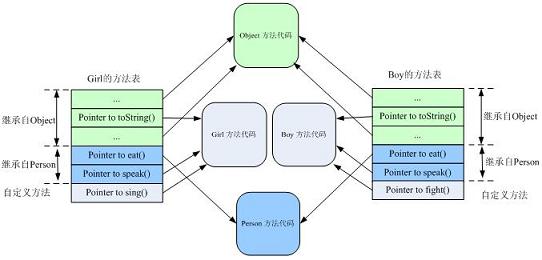
public void speak(){}

public void sing(){}

}

当这三个类被载入到 Java 虚拟机之后，方法区中就包含了各自的类的信息。Girl 和 Boy 在方法区中的方法表可表示如下：

**图 3.Boy 和 Girl 的方法表**



可以看到，Girl 和 Boy 的方法表包含继承自 Object 的方法，继承自直接父类 Person 的方法及各自新定义的方法。注意方法表条目指向的具体的方法地址，如 Girl 的继承自 Object 的方法中，只有 toString() 指向自己的实现（Girl 的方法代码），其余皆指向 Object 的方法代码；其继承自于 Person 的方法 eat() 和 speak() 分别指向 Person 的方法实现和本身的实现。

Person 或 Object 的任意一个方法，在它们的方法表和其子类 Girl 和 Boy 的方法表中的位置 (index) 是一样的。这样 JVM 在调用实例方法其实只需要指定调用方法表中的第几个方法即可。

如调用如下：

**清单 2**

class Party{

…

void happyHour(){

Person girl = new Girl();

girl.speak();

…

}

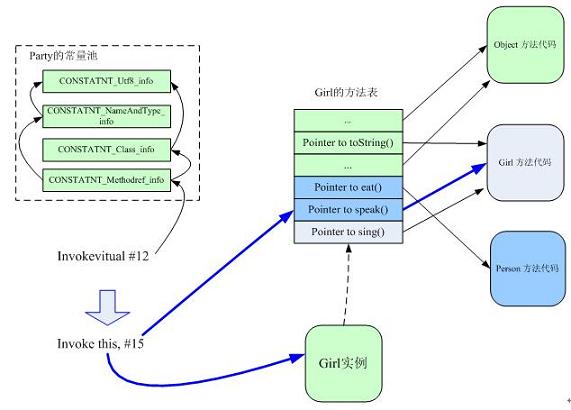
}

当编译 Party 类的时候，生成 girl.speak()的方法调用假设为：

Invokevirtual #12

设该调用代码对应着 girl.speak(); #12 是 Party 类的常量池的索引。JVM 执行该调用指令的过程如下所示：

**图 4. 解析调用过程**



JVM 首先查看 Party 的常量池索引为 12 的条目（应为 CONSTANT\_Methodref\_info 类型，可视为方法调用的符号引用），进一步查看常量池（CONSTANT\_Class\_info，CONSTANT\_NameAndType\_info ，CONSTANT\_Utf8\_info）可得出要调用的方法是 Person 的 speak 方法（注意引用 girl 是其基类 Person 类型），查看 Person 的方法表，得出 speak 方法在该方法表中的偏移量 15（offset），这就是该方法调用的直接引用。

当解析出方法调用的直接引用后（方法表偏移量 15），JVM 执行真正的方法调用：根据实例方法调用的参数 this 得到具体的对象（即 girl 所指向的位于堆中的对象），据此得到该对象对应的方法表 (Girl 的方法表 )，进而调用方法表中的某个偏移量所指向的方法（Girl 的 speak() 方法的实现）。

**接口调用**

因为 Java 类是可以同时实现多个接口的，而当用接口引用调用某个方法的时候，情况就有所不同了。Java 允许一个类实现多个接口，从某种意义上来说相当于多继承，这样同样的方法在基类和派生类的方法表的位置就可能不一样了。

**清单 3**

interface IDance{

void dance();

}

class Person {

public String toString(){

return "I'm a person.";

}

public void eat(){}

public void speak(){}

}

class Dancer extends Person

implements IDance {

public String toString(){

return "I'm a dancer.";

}

public void dance(){}

}

class Snake implements IDance{

public String toString(){

return "A snake.";

}

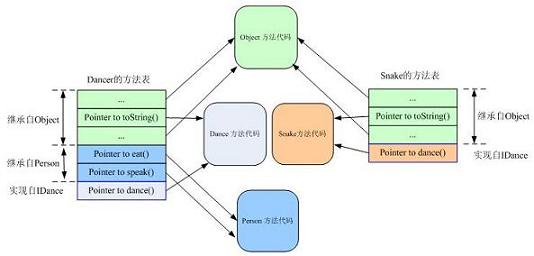
public void dance(){

//snake dance

}

}

**图 5.Dancer 的方法表（**[**查看大图**](https://www.ibm.com/developerworks/cn/java/j-lo-polymorph/image011.jpg)**）**



可以看到，由于接口的介入，继承自于接口 IDance 的方法 dance（）在类 Dancer 和 Snake 的方法表中的位置已经不一样了，显然我们无法通过给出方法表的偏移量来正确调用 Dancer 和 Snake 的这个方法。这也是 Java 中调用接口方法有其专有的调用指令（invokeinterface）的原因。

Java 对于接口方法的调用是采用搜索方法表的方式，对如下的方法调用

invokeinterface #13

JVM 首先查看常量池，确定方法调用的符号引用（名称、返回值等等），然后利用 this 指向的实例得到该实例的方法表，进而搜索方法表来找到合适的方法地址。

因为每次接口调用都要搜索方法表，所以从效率上来说，接口方法的调用总是慢于类方法的调用的。