方法调用并不等于方法执行，方法调用阶段唯一的任务就是确定被调用方法的版本（即调用哪一个方法），暂时还不涉及方法内部的具体运行过程。

在程序运行时，进行方法调用是最普遍、最频繁的操作，但是Class文件的编译过程不包括传统编译中的***连接***步骤，***一切方法调用在Class文件里面存储的都只是符号引用***，而不是方法在实际运行时内存布局中的入口地址（相对于之前说的直接引用）。这个特性给Java带来了更强大的动态扩展能力，但也使得Java方法调用过程变得相对复杂起来，需要在***类加载期间***，甚至到***运行期间***才能确定目标方法的直接引用。

解析

所有方法调用中的目标方法在Class文件里面都是一个***常量池中的引用***，在类加载的***解析阶段***，会将其中的一部分符号引用转化为直接引用。这种解析能成立的***前提***是：方法在程序真正执行之前就有一个可确定的调用版本，并且这个方法的调用版本在运行期是不可改变的。换句话说，调用目标在程序代码写好、编译器进行编译时就必须确定下来，这类方法的调用称为***解析***。

在Java语言中符合“***编译期可知，运行期不可变***”这个要求的方法，主要包括***静态方法***和***私有方法***两大类，前者与类型直接关联，后者在外部不可被访问，这两种方法各自的特点决定了他们不可能通过继承或别的方式重写其他版本，因此他们适合在类加载阶段进行解析。

***静态方法***、***私有方法***、***实例构造器***、***父类方法***。这些方法称为***非虚方法***，它们在类加载的时候就会把***符号引用***解析为该方法的***直接引用***。与之相反，其他方法称为***虚方法***（除去final方法）。

分派

静态分派

1. public class StaticDispatch {
2. static abstract class Human{
3. }
4. static class Man extends Human{
5. }
6. static class Woman extends Human{
7. }
8. public static void sayHello(Human guy){
9. System.out.println("hello,guy!");
10. }
11. public static void sayHello(Man guy){
12. System.out.println("hello,gentlemen!");
13. }
14. public static void sayHello(Woman guy){
15. System.out.println("hello,lady!");
16. }
18. public static void main(String[] args) {
19. Human man=new Man();
20. Human woman=new Woman();
21. sayHello(man);
22. sayHello(woman);
23. }
24. }

输出：

hello,guy!  
hello,guy!

Human man=new Man();  
我们把“Human”称为变量的***静态类型***，后面的“Man”称为变量的***实际类型***，静态类型和实际类型在程序中都可以发生一些变化，区别是静态类型的变化仅仅在使用时发生，***变量***本身的静态类型不会被改变，并且最终的静态类型在编译器可知；而实际类型变化的结果在运行期才确定，编译器在编译期并不知道一个对象的实际类型是什么。

1. Human man=new Man();
2. sayHello(man);
3. sayHello((Man)man);*//类型转换，静态类型变化，我们知道转型后的静态类型一定是Man*
4. man=new Woman(); *//实际类型变化,实际类型却是不确定的*
5. sayHello(man);
6. sayHello((Woman)man);*//类型转换，静态类型变化*

输出：

hello,guy!  
hello,gentlemen!  
hello,guy!  
hello,lady!

编译器在***重载***时是通过***参数的静态类型***而不是实际类型作为判定的依据。并且静态类型在编译期可知，因此，编译阶段，Javac编译器会根据***参数的静态类型***决定使用哪个重载版本。

所有依赖静态类型来定位方法执行版本的分派动作称为***静态分派***。静态分派的典型应用就是***方法重载***。

静态分派发生在编译阶段，因此确定静态分派的动作实际上不是由虚拟机来执行的，而是由编译器来完成。

但是，***字面量没有显示的静态类型***，它的静态类型只能通过语言上的规则去理解和推断。

1. public class LiteralTest {
2. */\*\*/*
3. *public static void sayHello(char arg){*
4. *System.out.println("hello char");*
5. *}*
6. *public static void sayHello(int arg){*
7. *System.out.println("hello int");*
8. *}*
10. *public static void sayHello(long arg){*
11. *System.out.println("hello long");*
12. *}*
14. *public static void sayHello(Character arg){*
15. *System.out.println("hello Character");*
16. *}*
17. *public static void main(String[] args) {*
18. *sayHello('a');*
19. *}*
20. *}*

输出：

hello char  
将重载方法从上向下依次注释，将会得到不同的输出。

***如果编译器无法确定要自定转型为哪种类型，会提示类型模糊，拒绝编译。***

1. import java.util.Random;
3. public class LiteralTest {
4. */\*\*/*
5. *public static void sayHello(String arg){//新增重载方法*
6. *System.out.println("hello String");*
7. *}*
8. *public static void sayHello(char arg){*
9. *System.out.println("hello char");*
10. *}*
11. *public static void sayHello(int arg){*
12. *System.out.println("hello int");*
13. *}*
15. *public static void sayHello(long arg){*
16. *System.out.println("hello long");*
17. *}*
19. *public static void sayHello(Character arg){*
20. *System.out.println("hello Character");*
21. *}*
22. *public static void main(String[] args) {*
23. *Random r=new Random();*
24. *String s="abc";*
25. *int i=0;*
26. *sayHello(r.nextInt()%2!=0?s:i);//编译错误*
27. *sayHello(r.nextInt()%2!=0?'a':false);//编译错误*
28. *}*
29. *}*

动态分派

1. public class DynamicDispatch {
2. static abstract class Human{
3. protected abstract void sayHello();
4. }
5. static class Man extends Human{
6. @Override
7. protected void sayHello() {
8. System.out.println("man say hello!");
9. }
10. }
11. static class Woman extends Human{
12. @Override
13. protected void sayHello() {
14. System.out.println("woman say hello!");
15. }
16. }
17. public static void main(String[] args) {
19. Human man=new Man();
20. Human woman=new Woman();
21. man.sayHello();
22. woman.sayHello();
23. man=new Woman();
24. man.sayHello();
25. }
26. }

输出：  
man say hello!  
woman say hello!  
woman say hello!

显然，这里不可能再根据静态类型来决定，因为静态类型同样是Human的两个变量man和woman在调用sayHello()方法时执行了不同的行为，并且变量man在两次调用中执行了不同的方法。导致这个现象的原因很明显，是这两个变量的实际类型不同，Java虚拟机是如何根据实际类型来分派方法执行版本的呢？  
我们从invokevirtual指令的多态查找过程开始说起，***invokevirtual指令***的运行时解析过程大致分为以下几个步骤：

1、找到操作数栈顶的第一个元素所指向的对象的***实际类型***，记作***C***。  
2、如果在***类型C***中找到与常量中的描述符和简单名称相符合的方法，然后进行访问权限验证，如果验证通过则返回这个方法的直接引用，查找过程结束；如果验证不通过，则抛出java.lang.IllegalAccessError异常。  
3、否则未找到，就按照继承关系***从下往上***依次对***类型C***的各个***父类***进行第2步的搜索和验证过程。  
4、如果始终没有找到合适的方法，则跑出java.lang.AbstractMethodError异常。

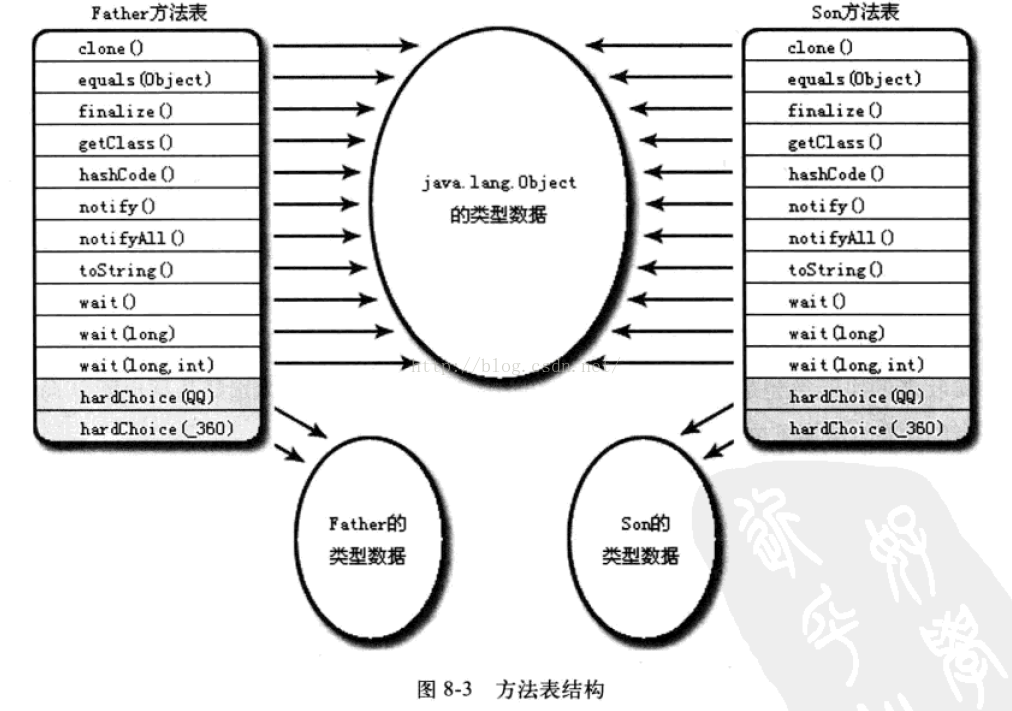
由于invokevirtual指令执行的第一步就是在运行期确定接收者的***实际类型***，所以两次调用中的invokevirtual指令把常量池中的***类方法符号引用***解析到了***不同***的***直接引用***上，这个过程就是Java语言方法***重写***的本质。我们把这种在运行期根据实际类型确定方法执行版本的分派过程称为***动态分派***。

虚拟机动态分派的实现

前面介绍的分派过程，作为对虚拟机概念模型的解析基本上已经足够了，它已经解决了虚拟机在分派中"会做什么"这个问题。

但是，虚拟机”具体是如何做到的“，可能各种虚拟机实现都会有些差别。

由于动态分派是非常频繁的动作，而且动态分派的方法版本选择过程需要运行时在类的方法元数据中搜索合适的目标方法，因此虚拟机的实际实现中基于性能的考虑，大部分实现都不会真正的进行如此频繁的搜索。面对这种情况，最常用的***”稳定优化“***手段就是为类在方法区中建立一个虚方法表（Virtual Method Table，也称为vtable），使用***虚方法表索引***来***代替***元数据***查找***以提高性能。



虚方法表中存放着各个方法的***实际入口地址***。如果某个方法在子类中没有被重写，那子类的虚方法表里面的地址入口和父类相同方法的地址入口是一致的，都是指向父类的实际入口。如果子类中重写了这个方法，子类方法表中的地址将会替换为指向子类实际版本的入口地址。

为了程序实现上的方便，具有相同签名的方法，在父类、子类的虚方法表中具有***一样的索引序号***，这样当类型变换时，仅仅需要变更查找的方法表，就可以从不同的虚方法表中按索引转换出所需要的入口地址。

方法表一般在类加载阶段的***连接阶段***进行初始化，准备了类的变量初始值后，虚拟机会把该类的方法表也初始化完毕。