**[深入理解java字节码](http://kingj.iteye.com/blog/1555046)**

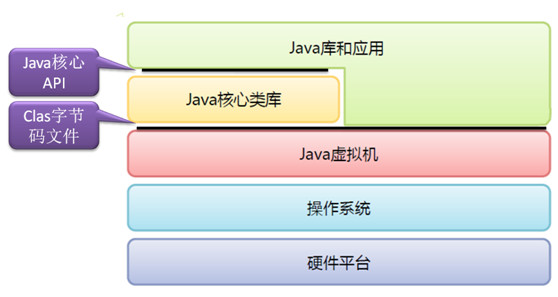
**博客分类：**

* [深入java虚拟机，字节码](http://kingj.iteye.com/category/208991)

**深入理解JVM—字节码执行引擎**

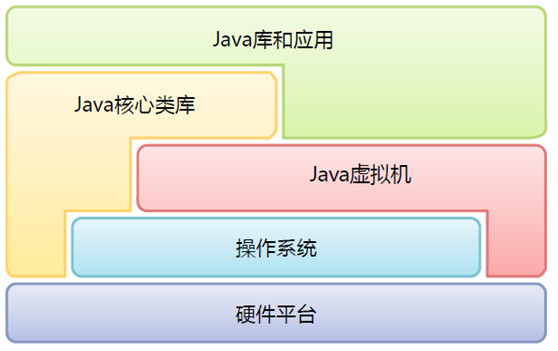
2012-03-20 16:10:17|  分类： [JVM](http://yhjhappy234.blog.163.com/blog/#m=0&t=1&c=fks_084065093084089067087080094095087087087064086094085065) |  标签：[jvm](http://yhjhappy234.blog.163.com/blog/" \l "m=0&t=3&c=jvm)  [字节码](http://yhjhappy234.blog.163.com/blog/#m=0&t=3&c=%E5%AD%97%E8%8A%82%E7%A0%81)  [执行引擎](http://yhjhappy234.blog.163.com/blog/#m=0&t=3&c=%E6%89%A7%E8%A1%8C%E5%BC%95%E6%93%8E)  [class](http://yhjhappy234.blog.163.com/blog/#m=0&t=3&c=class)  [分派](http://yhjhappy234.blog.163.com/blog/#m=0&t=3&c=%E5%88%86%E6%B4%BE)  |字号 订阅

    前面我们不止一次的提到，Java是一种跨平台的语言，为什么可以跨平台，因为我们编译的结果是中间代码—字节码，而不是机器码，那字节码在整个Java平台扮演着什么样的角色的呢？JDK1.2之前对应的结构图如下所示：



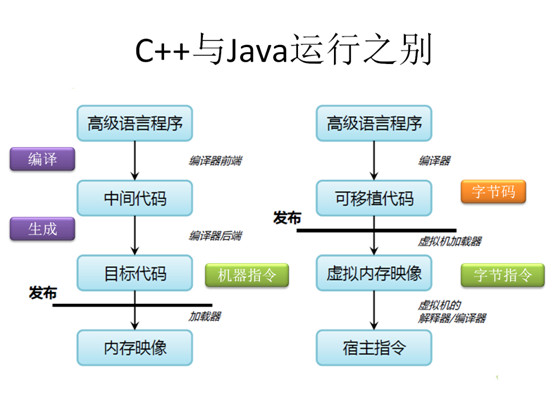
 从

JDK1.2开始，迫于Java运行始终笔C++慢的压力，JVM的结构也慢慢发生了一些变化，JVM在某些场景下可以操作一定的硬件平台，一些核心的Java库甚至也可以操作底层的硬件平台，从而大大提升了Java的执行效率，在前面JVM内存模型和垃圾回收中也给大家演示了如何操作物理内存，下图展示了JDK1.2之后的JVM结构模型。



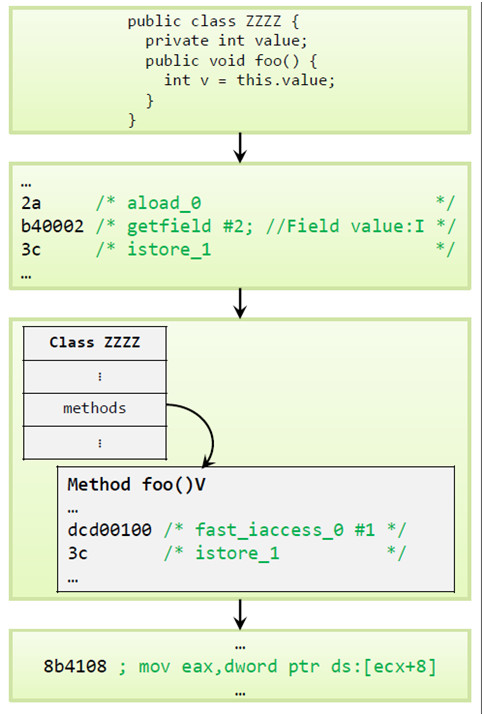
 那

C++和Java在编译和运行时到底有啥不一样？为啥Java就能跨平台的呢？



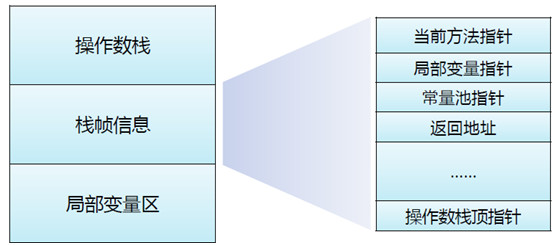
 我们从上图可以看出。

C++发布的就是机器指令，而Java发布的是字节码，字节码在运行时通过JVM做一次转换生成机器指令，因此能够更好的跨平台运行。如图所示，展示了对应代码从编译到执行的一个效果图。



 我们知道

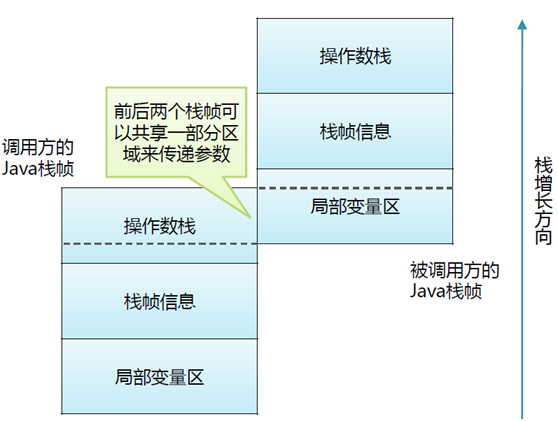
JVM是基于栈执行的，每个线程会建立一个操作栈，每个栈又包含了若干个栈帧， 每个栈帧包含了局部变量、操作数栈、动态连接、方法的返回地址信息等。其实在我们编译的时候，需要多大的局部变量表、操作数深度等已经确定并写入了Code属性，因此运行时内存消耗的大小在启动时已经已知。



 在栈帧中，最小的单位为变量槽

(Variable Slot),其中每个Slot占用32个字节。在32bit的JVM中32位的 数据类型占用1个Slot，64bit数据占用2个Slot；在64bit中使用64bit字节填充来模拟32bit（又称补位），因此我们可以得出结论：64bit的JVM比32bit的更消耗内存，但是又出32bit机器的内存上限限制，有时候牺牲一部分还是值得的。Java的基本数据类型中，除了long、double两种数据类型为64bit以外，boolean、byte、char、int、float、reference等都是32bit的数据类型。

在栈帧中，局部变量表中的Slot是可以复用的，如在一个方法返回给上一个方法是就可以通过公用Slot的方法来节约内存控件，但这在一定程度省会影响垃圾回收，因此JVM不确定这块Slot空间是否还需要复用。



 Slot

复用会给JVM的垃圾回收带来一定影响，如下代码：

**package** com.yhj.jvm.byteCode.slotFree;

/\*\*

 \* **@Described**：Slot局部变量表 没有破坏GCRoot情况演示

 \* **@VM** params :-XX:+PrintGCDetails -verbose:gc

 \* **@author** YHJ create at 2012-2-22 下午04:37:29

 \* **@FileNmae** com.yhj.jvm.byteCode.slotFree.SlotFreeTestCase.java

 \*/

**public** **class** SlotFreeTestCase {

    /\*\*

     \* **@param** args

     \* **@Author** YHJ create at 2012-2-22 下午04:37:25

     \*/

    @SuppressWarnings("unused")

**public** **static** **void** main(String[] args) {

       //case 1

**byte**[] testCase = **new** **byte**[10\*1024\*1024];

       System.*gc*();

//

//     //case 2

//     {

//         byte[] testCase = new byte[10\*1024\*1024];

//     }

//     System.gc();

//

//     //case 3

//     {

//         byte[] testCase = new byte[10\*1024\*1024];

//     }

//     int a = 0;

//     System.gc();

//

//     //case 5

//     byte[] testCase = new byte[10\*1024\*1024];

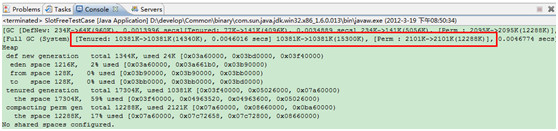
//     testCase=null;

//     System.gc();

    }

}

如上所示，当我们执行这段代码的时候并不会引发GC的回收，因为很简单，我的testCase对象还在使用中，生命周期并未结束，因此运行结果如下



 但是我们换下面的

case2这种写法呢？

       //case 2

       {

**byte**[] testCase = **new** **byte**[10\*1024\*1024];

       }

       System.*gc*();

这种写法，testCase在大括号中生命周期已经结束了，会不会引发GC的呢？我们来看结果：



  我们可以看到仍然没有进行回收。

那我变通一下，再定义一个变量会怎么样的呢？

       //case 3

       {

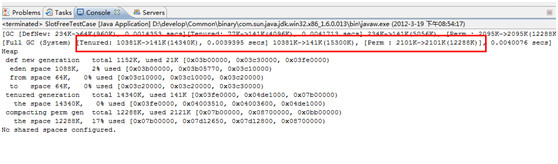
**byte**[] testCase = **new** **byte**[10\*1024\*1024];

       }

**int** a = 0;

       System.*gc*();

这下我们貌似看到奇迹了



 没错，

JVM做了回收操作，因为JVM在做下面的操作时并没有发现公用的Slot，因此该内存区域被回收。但是我们这样写代码会让很多人感到迷惑，我们应该怎样写才能更好一点让人理解的呢？

       //case 5

**byte**[] testCase = **new** **byte**[10\*1024\*1024];

       testCase=**null**;

       System.gc();

无疑，这样写才是最好的，这也是书本effective Java中强调了很多遍的写法，随手置空不用的对象。

我们知道private int a；这么一个语句在一个类中的话他的默认值是0，那么如果是在局部变量中的呢？我们开看这样一段代码：

**package** com.yhj.jvm.byteCode.localVariableInit;

/\*\*

 \* **@Described**：局部变量拒绝默认初始化

 \* **@author** YHJ create at 2012-2-24 下午08:40:34

 \* **@FileNmae** com.yhj.jvm.byteCode.localVariableInit.LocalVariableInit.java

 \*/

**public** **class** LocalVariableInit {

    /\*\*

     \* **@param** args

     \* **@Author** YHJ create at 2012-2-22 下午05:12:06

     \*/

    @SuppressWarnings("unused")

**public** **static** **void** main(String[] args) {

**int** a;

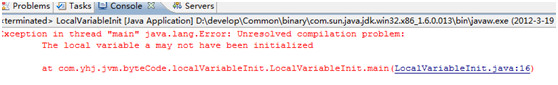
       System.*out*.println(a);

    }

}

这段代码的运营结果又是什么的呢？

很多人会回答0.我们来看一下运行结果：



 没错，就是报错了，如果你使用的是

Eclipse这种高级一点的IDE的话，在编译阶段他就会提示你，该变量没有初始化。原因是什么的呢？原因就是，局部变量并没有类实例变量那样的连接过程，前面我们说过，类的加载分为加载、连接、初始化三个阶段，其中连接氛围验证、准备、解析三个阶段，而验证是确保类加载的正确性、准备是为类的静态变量分配内存，并初始化为默认值、解析是把类中的符号引用转换为直接引用。而外面的初始化为类的静态变量赋值为正确的值。而局部变量并没有连接的阶段，因此没有赋值为默认值这一阶段，因此必须自己初始化才能使用。

我们在类的加载中提到类的静态连接过程，但是还有一部分类是需要动态连接的，其中以下是需要动态连接的对象

1、  实例变量（类的变量或者局部变量）

2、  通过其他荣报告期动态注入的变量(*IOC*)

3、  通过代码注入的对象(void setObj(Object obj))

所有的动态连接都只有准备和解析阶段，没有再次校验(校验发生在连接前类的加载阶段)，其中局部变量不会再次引发准备阶段。

前面我们提到JVM的生命周期，在以下四种情况下会引发JVM的生命周期结束

1、  执行了System.exit()方法

2、  程序正常运行结束

3、  程序在执行过程中遇到了异常或者错误导致异常终止

4、  由于操作系统出现错误而导致JVM进程终止

同样，在以下情况下会导致一个方法调用结束

1、  执行引擎遇到了方法返回的字节码指令

2、  执行引擎在执行过程中遇到了未在该方法内捕获的异常

这时候很多人会有一个疑问：当程序返回之后它怎么知道继续在哪里执行？这就用到了我们JVM内存模型中提到了的PC计数器。方法退出相当于当前栈出栈，出栈后主要做了以下事情：

1、  回复上层方法的局部变量表

2、  如果有返回值的话将返回值压入到上层操作数栈

3、  调整PC计数器指向下一条指令

除了以上信息以外，栈帧中还有一些附加信息，如预留一部分内存用于实现一些特殊的功能，如调试信息，远程监控等信息。

接下来我们要说的是方法调用，方法调用并不等于方法执行，方法调用的任务是确定调用方法的版本(调用哪一个方法)，在实际过程中有可能发生在加载期间也有可能发生在运行期。Class的编译过程并不包含类似C++的连接过程，只有在类的加载或者运行期才将对应的符号引用修正为真正的直接引用，大大的提升了Java的灵活性，但是也大大增加了Java的复杂性。

在类加载的第二阶段连接的第三阶段解析，这一部分是在编译时就确定下来的，属于编译期可知运行期不可变。在字节码中主要包含以下两种

1、  invokestatic 主要用于调用静态方法，属于绑定类的调用

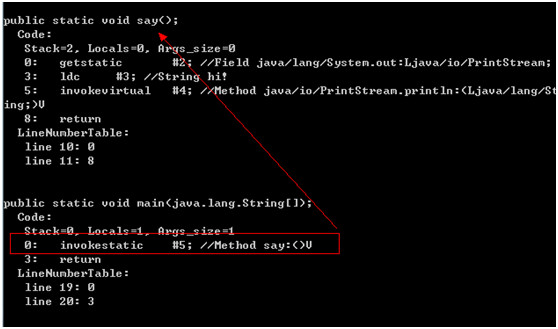
2、  invokespecial 主要用于调用私有方法，外部不可访问，绑定实例对象

还有一种是在运行时候解析的，只有在运行时才能确定下来的，主要包含以下两方面

1、  invokevirtual 调用虚方法，不确定调用那一个实现类

2、  invokeinterface 调用接口方法，不确定调用哪一个实现类

我们可以通过javap的命令查看对应的字节码文件方法调用的方式，如下图所示



 Java

方法在调用过程中，把invokestatic和invokespecial定义为非虚方法的调用，非虚方法的调用都是在编译器已经确定具体要调用哪一个方法，在类的加载阶段就完成了符号引用到直接引用的转换。除了非虚方法以外，还有一种被final修饰的方法，因被final修饰以后调用无法通过其他版本来覆盖，因此被final修饰的方法也是在编译的时候就已知的废墟方法。

除了解析，Java中还有一个概念叫分派，分派是多态的最基本表现形式，可分为单分派、多分派两种；同时分派又可以分为静态分派和动态分派，因此一组合，可以有四种组合方式。其实最本质的体现就是方法的重载和重写。我们来看一个例子

**package** com.yhj.jvm.byteCode.staticDispatch;

/\*\*

 \* **@Described**：静态分配

 \* **@author** YHJ create at 2012-2-24 下午08:20:06

 \* **@FileNmae** com.yhj.jvm.byteCode.staticDispatch.StaticDispatch.java

 \*/

**public** **class** StaticDispatch {

**static** **abstract** **class** Human{};

**static** **class** Man **extends** Human{} ;

**static** **class** Woman **extends** Human{} ;

**public** **void** say(Human human) {

       System.*out*.println("hi,you are a good human!");

    }

**public** **void** say(Man human) {

       System.*out*.println("hi,gentleman!");

    }

**public** **void** say(Woman human) {

       System.*out*.println("hi,yong lady!");

    }

    /\*\*

     \* **@param** args

     \* **@Author** YHJ create at 2012-2-24 下午08:20:00

     \*/

**public** **static** **void** main(String[] args) {

       Human man = **new** Man();

       Human woman = **new** Woman();

       StaticDispatch dispatch = **new** StaticDispatch();

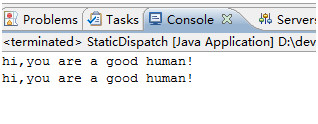
       dispatch.say(man);

       dispatch.say(woman);

    }

}

这个例子的执行结果会是什么呢？我们来看一下结果



 和你的预期一致么？这个其实是一个静态分派的杯具，

man和woman两个对象被转型以后，通过特征签名匹配，只能匹配到对应的父类的重载方法，因此导致最终的结构都是执行父类的代码。因为具体的类是在运行期才知道具体是什么类型，而编译器只确定是Human这种类型的数据。

这种写法曾经在我们项目中也发生过一次。如下代码所示

**package** com.yhj.jvm.byteCode.staticDispatch;

**import** java.util.ArrayList;

**import** java.util.List;

/\*\*

 \* **@Described**：蝌蚪网曾经的杯具

 \* **@author** YHJ create at 2012-2-26 下午09:43:20

 \* **@FileNmae** com.yhj.jvm.byteCode.staticDispatch.CothurnusInPassport.java

 \*/

**public** **class** CothurnusInPassport {

    /\*\*

     \* 主函数入口

     \* **@param** args

     \* **@Author** YHJ create at 2012-2-26 下午09:48:02

     \*/

**public** **static** **void** main(String[] args) {

       List<CothurnusInPassport> inPassports = **new** ArrayList<CothurnusInPassport>();

       inPassports.add(**new** CothurnusInPassport());

       String xml = XML\_Util.*createXML*(inPassports);

       System.*out*.println(xml);

    }

}

**class** XML\_Util{

**public** **static** String createXML(Object obj){

**return**  。。。// ... 通过反射遍历属性 生成对应的XML节点

    }

**public** **static** String createXML(List<Object> objs){

       StringBuilder sb = **new** StringBuilder();

**for**(Object obj : objs)

           sb.append(*createXML*(obj));

**return** **new** String(sb);

    }

}

当时我们项目组写了以恶搞XML\_Util的一个类用于生成各种XML数据，其中一个实例传入的参数是Object，一个是一个List类型的数据，如上面代码所示，我的调用结果会执行哪一个的呢？结果大家已经很清楚了，他调用了createXML(Object obj)这个方法，因此生成过程中老是报错，原因很简单，就是因为我叼用的时候泛型 不匹配，进行了隐式的类型转换，因此无法匹配到对应的List《Object》最终调用了createXML(Object obj)这个方法。

下面我们来看一道恶心的面试题，代码如下：

**package** com.yhj.jvm.byteCode.polymorphic;

**import** java.io.Serializable;

/\*\*

 \* **@Described**：重载测试

 \* **@author** YHJ create at 2012-2-24 下午08:41:12

 \* **@FileNmae** com.yhj.jvm.byteCode.polymorphic.OverLoadTestCase.java

 \*/

**public** **class** OverLoadTestCase {

**public** **static** **void** say(Object obj){ System.*out*.println("Object"); }

**public** **static** **void** say(**char** obj){ System.*out*.println("char"); }

**public** **static** **void** say(**int** obj){ System.*out*.println("int"); }

**public** **static** **void** say(**long** obj){ System.*out*.println("long"); }

**public** **static** **void** say(**float** obj){ System.*out*.println("float"); }

**public** **static** **void** say(**double** obj){ System.*out*.println("double"); }

**public** **static** **void** say(Character obj){ System.*out*.println("Character"); }

**public** **static** **void** say(Serializable obj){ System.*out*.println("Serializable"); }

**public** **static** **void** say(**char**... obj){ System.*out*.println("char..."); }

**public** **static** **void** main(String[] args) {

       OverLoadTestCase.*say*('a');

    }

}

这样的代码会执行什么呢？这个很简单的了，是char，那如果我注释掉char这个方法，再执行呢？是int，继续注释，接下来是什么的呢？大家可以自己测试一下，你会发现这段代码有多么的恶心。

我们接下来再看一段代码：

**package** com.yhj.jvm.byteCode.dynamicDispatch;

/\*\*

 \* **@Described**：动态分派测试

 \* **@author** YHJ create at 2012-2-26 下午10:05:43

 \* **@FileNmae** com.yhj.jvm.byteCode.dynamicDispatch.DynamicDispatch.java

 \*/

**public** **class** DynamicDispatch {

**static** **abstract** **class** Human{

**public** **abstract** **void** say();

    };

**static** **class** Man **extends** Human{

       @Override

**public** **void** say(){

           System.*out*.println("hi,you are a good man!");

       }

    } ;

**static** **class** Woman **extends** Human{

       @Override

**public** **void** say(){

           System.*out*.println("hi,young lady!");

       }

    } ;

    //主函数入口

**public** **static** **void** main(String[] args) {

       Human man = **new** Man();

       Human woman = **new** Woman();

       man.say();

       woman.say();

       woman = **new** Man();

       woman.say();

    }

}

这段代码执行的结果会是什么的呢？这个不用说了吧？企业级的应用经常会使用这些的方法重写，这是动态分配的一个具体体现，也就是说只有运行期才知道具体执行的是哪一个类，在编译期前并不知道会调用哪一个类的这个方法执行。

我们再来看一段代码，这段代码被称为“一个艰难的决定”

//动态单分派静态多分派    宗量选择

**package** com.yhj.jvm.byteCode.dynamicOneStaticMoreDispatch;

/\*\*

 \* **@Described**：一个艰难的决定

 \* **@author** YHJ create at 2012-2-24 下午09:23:26

 \* **@FileNmae** com.yhj.jvm.byteCode.dynamicOneStaticMore.OneHardMind.java

 \*/

**public** **class** OneHardMind {

**static** **class** QQ{}                //腾讯QQ

**static** **class** \_360{}             //360安全卫士

**static** **class** QQ2011 **extends** QQ{} //腾讯QQ2011

**static** **class** QQ2012 **extends** QQ{} //腾讯QQ2012

    //百度

**static** **class** BaiDu{

**public** **static** **void** choose(QQ qq){ System.*out*.println("BaiDu choose QQ"); }

**public** **static** **void** choose(QQ2011 qq){ System.*out*.println("BaiDu choose QQ2011"); }

**public** **static** **void** choose(QQ2012 qq){ System.*out*.println("BaiDu choose QQ2012"); }

**public** **static** **void** choose(\_360 \_){ System.*out*.println("BaiDu choose 360 safe"); }

    }

    //迅雷

**static** **class** Thunder{

**public** **static** **void** choose(QQ qq){ System.*out*.println("Thunder choose QQ"); }

**public** **static** **void** choose(QQ2011 qq){ System.*out*.println("Thunder choose QQ2011"); }

**public** **static** **void** choose(QQ2012 qq){ System.*out*.println("Thunder choose QQ2012"); }

**public** **static** **void** choose(\_360 qq){ System.*out*.println("Thunder choose 360 safe"); }

    }

    //主函数入口

    @SuppressWarnings("static-access")

**public** **static** **void** main(String[] args) {

       BaiDu baiDu = **new** BaiDu();

       Thunder thunder = **new** Thunder();

       QQ qq = **new** QQ();

       \_360 \_360\_safe = **new** \_360();

       baiDu.*choose*(qq);

       thunder.*choose*(\_360\_safe);

       qq = **new** QQ2011();

       baiDu.*choose*(qq);

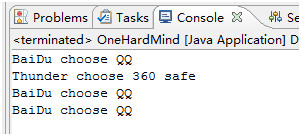
       qq = **new** QQ2012();

       baiDu.*choose*(qq);

    }

}

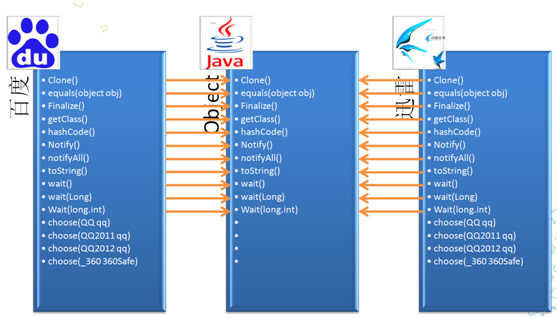
这段代码的执行结果又是什么？现在可以很简单的说出对应的结果了吧！



 从这个例子我们可以看出，

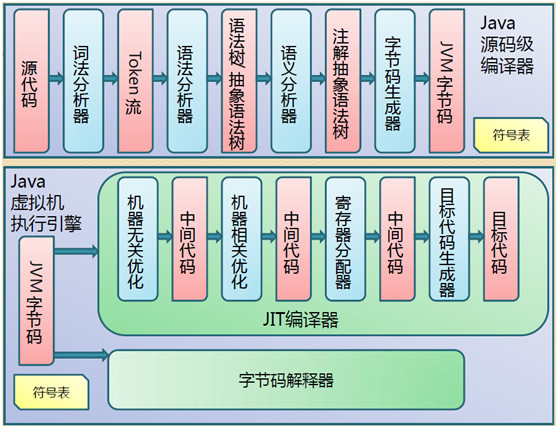
Java是静态多分派动态单分派的 同理，C#3.0 、C++也是静态多分配，动态单分派的 C#4.0后引入类型dynamic可以实现动态多分派，sun公司在JSR-292中提出了动态多分派的实现，规划在JDK1.7推出，但是被oracle收购后，截至目前，JDK1.7已经不发布了多个版本，但尚未实现动态多分派。至于动态多分派究竟是怎么样子的？我们可以参考[Python的多分派实例](http://www.ibm.com/developerworks/cn/linux/l-pydisp/)。

那虚拟机为什么能够实现不同的类加载不同的方法，什么时候使用静态分派？什么时候又使用动态分派呢？我们把上面的示例用一个图来表示，大家就很清楚了！



 当子类有重写父类的方法时，在系统进行解析的时候，子类没有重写的方法则将对应的符号引用解析为父类的方法的直接引用，否则解析为自己的直接引用，因此重写永远都会指向自己的直接引用，但是重载在解析时并不知道具体的直接引用对象是哪一个？所以只能解析为对应的表象类型的方法。

我们在前面已经提到，新的Java编译器已经不会纯粹的走解释执行之路，在一些情况下还会走编译之路。如下图所示：



 我们知道，程序之所以能运行，是因为有指令集，而

JVM主要是基于栈的一个指令集，而还有一部分程序是基于寄存器的指令集，两者有什么区别的呢？

基于栈的指令集有接入简单、硬件无关性、代码紧凑、栈上分配无需考虑物理的空间分配等优势，但是由于相同的操作需要更多的出入栈操作，因此消耗的内存更大。而基于寄存器的指令集最大的好处就是指令少，速度快，但是操作相对繁琐。下面我们来看一段代码，看一下同样一段代码在不同引擎下的执行效果有啥不同。

**public** **class** Demo {

**public** **static** **void** foo() {

**int** a = 1;

**int** b = 2;

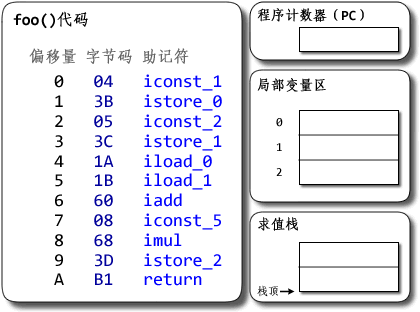
**int** c = (a + b) \* 5;

    }

}

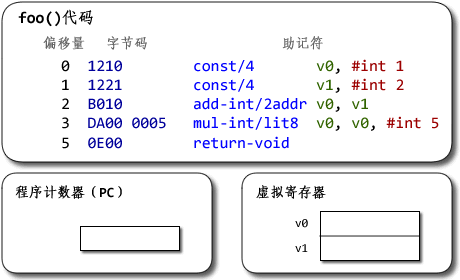
在Client/Server VM的模式下，我们可以使用javap –verbose ${ClassName}的方式来查看对应的字节码，而基于java的DalvikVM亦可以通过platforms\android-1.6\tools目录中的dx工具查看对应的字节码。具体命令为dx --dex –verbose --dump-to=packageName --dump-method=Demo.foo --verbose-dump Demo.class 。

基于栈的Hotspot的执行过程如下：

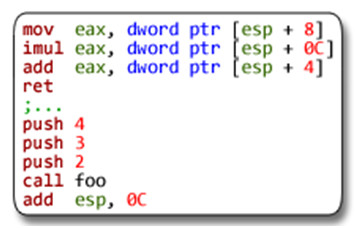


 基于栈的

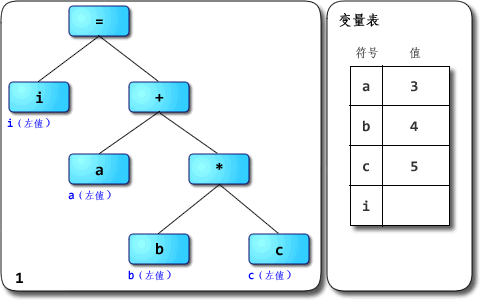
DalvikVM执行过程如下所示：



 而基于汇编语言的展示就是这样的了



附：基于JVM的逻辑运算模型如下图所示



 因此执行到

JVM上的过程就是下面的形式

