

第8章

类方法解析

本章摘要

- ◎ Java 方法签名解析
- ◎ Java 方法的 code 属性解析
- ◎ LVT 与 LVTT
- ◎ method 创建
- ◎ Java 方法的字节码指令解析
- ◎ <clinit>()方法与<init>()方法
- ◎ 使用 HSDB 查看运行时的字节码指令
- ◎ vtable 的概念与机制

前面分析了 HotSpot 解析常量池和类变量的详细过程，这一章接着探讨类方法的解析。作为一门面向对象的语言，每一个 Java 类都有属性和行为这两个基本要素，而 Java 又作为一门解释性的语言，由 JVM 虚拟机负责解释执行。JVM 执行的正是 Java 类的“行为”——Java 方法，而执行之前，必须要先对方法进行解析，毕竟 JVM 虚拟机没有真正的运算能力，最终必须依靠物理 CPU 完成 Java 字节码的运算。

Java 方法的解析大体上可以分为 3 道工序：

- (1) 在 Java 类源代码编译期间，编译器负责将 Java 类源代码翻译为对应的字节码指令，同时完成的工作还有 Java 方法局部变量表的计算，以及最大操作数栈的计算。
- (2) 在 JVM 运行期间，JVM 加载类型，调用 classFileParser::parseClassFile() 函数对 Java class 字节码文件进行解析，在这一步将会完成 Java 方法的分析、字节码指令存放、父类与接口类方

法继承与重载等一系列逻辑。

(3) 在调用系统加载器 System Class Loader (SCL) 对应用程序的 Java 类进行加载的过程中, 完成方法符号链接、验证, 最重要的是完成 vtable 与 itable 的构建, 从而支持在 JVM 运行期的方法动态绑定(也叫晚绑定)。当然, Java 技术体系不仅提供了 SCL, 还提供了其他类加载器用于加载 Java 类, 开发者也可以自定义类加载器来加载, 关于类加载器的话题会在本书后续章节详细讨论。

经过这 3 道层层推进的工序, 最终才能完成 Java 类方法的解析工作, 等这一切都完成后, JVM 才能在运行期通过 invoke_virtual 等字节码指令, 完成 Java 方法的调用和执行。

本章主要讲述 Java 类方法解析的第 2 道工序, 该工序主要在 classFileParser::parseClassFile() 函数中完成。classFileParser::parseClassFile() 函数主要完成 Java 类 class 字节码文件的解析, 其中不仅仅包含 Java 方法的解析, 还包含其他步骤, 前面几个步骤在前文都已进行了详细分析。这里还是按照惯例给出这个函数的总体“地图”(见图 8.1), 以理顺思路。

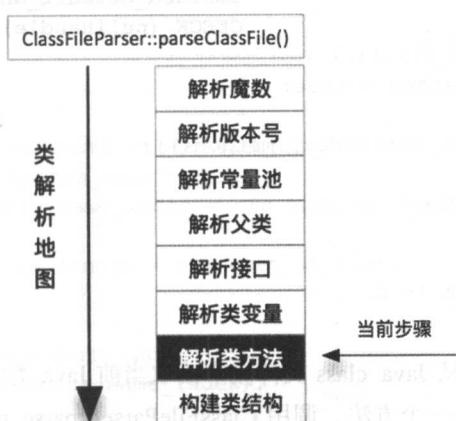


图 8.1 类解析的当前步骤——类方法解析

classFileParser::parseClassFile() 函数通过调用 ClassFileParser::parse_methods() 函数完成 Java 类方法解析的第 2 道工序。

ClassFileParser::parse_methods() 函数主要逻辑如下(仅保留主要逻辑, 不相关的代码皆已去除, 以免浪费纸张):

清单: /src/share/vm/classfile/classFileParser.cpp

作用: parse_methods() 方法主要逻辑

```

objArrayHandle ClassFileParser::parse_methods(constantPoolHandle cp, bool
is_interface,
  
```

```

AccessFlags* promoted_flags,
...) {
    ClassFileStream* cfs = stream();
    objArrayHandle nullHandle;
    typeArrayHandle method_annotations;
    typeArrayHandle method_parameter_annotations;
    typeArrayHandle method_default_annotations;
    u2 length = cfs->get_u2_fast(); // 获取方法长度
    if (length == 0) {
        return objArrayHandle(THREAD, Universe::the_empty_system_obj_array());
    } else {
        objArrayOop m = oopFactory::new_system_objArray(length, CHECK_(nullHandle));
        objArrayHandle methods(THREAD, m);
        for (int index = 0; index < length; index++) {
            methodHandle method = ClassFileParser::parse_method(cp, is_interface,
                promoted_flags,
                &method_annotations,
                &method_parameter_annotations,
                &method_default_annotations,
                CHECK_(nullHandle));
            if (method->is_final()) {
                *has_final_method = true;
            }
            methods->obj_at_put(index, method());
        }
    }
}

return methods;
} // --end if length != 0
}

```

这段逻辑很简单，首先从 Java class 文件流中读取当前 Java 类中所定义的全部方法数量，接着循环遍历 Java 类中的每一个方法，调用 ClassFileParser::parse_method() 函数对 Java 方法进行逐个解析。

ClassFileParser::parse_method() 函数所包含的逻辑颇为复杂，毕竟这是 Java 类的核心所在。不过虽然复杂，却是条缕分明，层次清晰，总体看来，其源码“骨架”如下：

清单：/src/share/vm/classfile/classFileParser.cpp

作用：parse_method()方法主要逻辑

```

methodHandle ClassFileParser::parse_method(constantPoolHandle cp, bool
is_interface,
                                            AccessFlags *promoted_flags,
                                            typeArrayHandle* method_annotations,
                                            typeArrayHandle*
method_parameter_annotations,

```

```

typeArrayHandle*
method_default_annotations,
TRAPS) {
    ClassFileStream* cfs = stream();
    methodHandle nullHandle;
    ResourceMark rm(THREAD);

    // ①. 读取和验证 Java 方法的访问标识、名称
    int flags = cfs->get_u2_fast();
    u2 name_index = cfs->get_u2_fast();
    // ...

    // 定义方法相关的内部属性，解析时会用到
    u2 max_stack = 0;
    u2 max_locals = 0;
    u4 code_length = 0;
    u1* code_start = 0;
    u2 exception_table_length = 0;
    // ...

    // ②. 解析方法的属性
    u2 method_attributes_count = cfs->get_u2_fast();
    while (method_attributes_count--) {
        u2 method_attribute_name_index = cfs->get_u2_fast(); // 获取 Java 方法当前
        属性的名称索引
        u4 method_attribute_length = cfs->get_u4_fast(); // 获取 Java 方法当前属
        性在字节码文件中所占的长度
        Symbol* method_attribute_name =
        cp->symbol_at(method_attribute_name_index);

        // Java 方法的 code 属性
        if (method_attribute_name == vmSymbols::tag_code()) {

            // ...
            while (code_attributes_count--) {

                // ...
                // 解析栈深度、局部变量表深度等

                // Java 方法源代码行号
                if (LoadLineNumberTables &&
                    cp->symbol_at(code_attribute_name_index) ==
                    vmSymbols::tag_line_number_table()) {
                        // Parse and compress line number table

```

```

// ...

// Java 方法局部变量表
} else if (LoadLocalVariableTables &&
           cp->symbol_at(code_attribute_name_index) ==
           vmSymbols::tag_local_variable_table()) {
    // Parse local variable table
    // ...

// Java 方法局部变量类型表
} else if (LoadLocalVariableTypeTables &&
           _major_version >= JAVA_1_5_VERSION &&
           cp->symbol_at(code_attribute_name_index) ==
           vmSymbols::tag_local_variable_type_table()) {
    // ...
}

// ...

// 异常属性
} else if (method_attribute_name == vmSymbols::tag_exceptions()) {
// ...

// 编译器生成属性--synthetic
} else if (method_attribute_name == vmSymbols::tag_synthetic()) {
// ...

// 方法废弃属性
} else if (method_attribute_name == vmSymbols::tag_DEPRECATED()) { // 4276120
    if (method_attribute_length != 0) {
        classfile_parse_error(
            "Invalid Deprecated method attribute length %u in class file %s",
            method_attribute_length, CHECK_(nullHandle));
    }
}
// ...

}

// ③. 创建 methodOop
methodOop m_oop = oopFactory::new_method(code_length, access_flags,
linenumber_table_length,
                                         total_lvt_length,
checked_exceptions_length,
                                         oopDesc::IsSafeConc,

```

```

CHECK_(nullHandle));
methodHandle m (THREAD, m_oop);

// ...

// ④. 复制字节码
m->set_code(code_start);

// ...

return m;
}

```

笔者使用①、②、③这样的标记将 parse_method()方法分成了几个大的步骤，这几个步骤分别是：

- (1) 读取和验证 Java 方法的访问标识、名称。
- (2) 解析方法的属性。
- (3) 创建 methodOop。
- (4) 复制字节码。

本文将 HotSpot 解析 Java 方法的步骤分为这 4 大步，虽然 HotSpot 其实并不仅仅只做了这 4 件事，例如解析方法的注解等，只是这 4 个步骤是理解 Java 方法解析的精髓所在。

8.1 方法签名解析与校验

Java class 字节码文件中的方法属性部分的解析，从方法的 flags 标识的索引（指该标识在 Java class 内部常量池的索引号，下同）开始。flags 标识占用 2 字节长度，并且其后连续跟了 4 字节，分别是方法名称索引和方法描述索引，方法名称索引和描述索引各占 2 字节长度。Java 方法的签名信息主要由 3 部分组成：

- ◎ 方法的标识，public、private、static、final、synchronized、native 等
- ◎ 方法的名称
- ◎ 方法的描述，描述方法的返回值类型和入参信息，例如()V 标识无人参的 void 类型方法

这 3 种信息共同组成 Java 方法的签名信息。由于每种信息中所存储的都是指向常量池的索引号，因此只需要 2 字节，Java 方法签名信息总共需要 6 字节长度。

在 parse_method()方法一开始，连续调用 3 次 cfs->get_u2_fast()，分 3 次从 Java class 字节码文件中读取出 Java 方法签名的 3 部分索引，并逐一进行校验。

在解析 Java 方法名称时，调用了 classFileParser.cpp::verify_legal_method_name()方法校验 Java 方法名称的有效性，verify_legal_method_name()主要逻辑如下：

清单：/src/share/vm/classfile/classFileParser.cpp

作用：verify_legal_method_name()方法主要逻辑

```
void ClassFileParser::verify_legal_method_name(Symbol* name, TRAPS) {
    // ...
    char* bytes = name->as_utf8_flexible_buffer(THREAD, buf, fixed_buffer_size);
    unsigned int length = name->utf8_length();
    bool legal = false;

    if (length > 0) {
        if (bytes[0] == '<') {
            if (name == vmSymbols::object_initializer_name() || name ==
                vmSymbols::class_initializer_name()) {
                legal = true;
            }
        } else if (_major_version < JAVA_1_5_VERSION) {
            // ...
        } else {
            legal = verify_unqualified_name(bytes, length, LegalMethod);
        }
    }
    // ...
}
```

在这段逻辑中，首先判断 Java 方法的第一个字符是否是“<”，如果是该字符，则接着判断当前 Java 方法是否是编译器自动生成的<init>和<clinit>这两种方法，如果不是，则校验不通过。所以，开发者所定义的 Java 方法，其名不能以“<”开始。

如果 Java 方法名不以“<”开始，则 verify_legal_method_name()会调用 verify_unqualified_name()函数继续校验 Java 方法中是否包含特殊的字符，verify_unqualified_name()函数主要逻辑如下：

清单：/src/share/vm/classfile/classFileParser.cpp

作用：verify_unqualified_name()方法主要逻辑

```
bool ClassFileParser::verify_unqualified_name(
    char* name, unsigned int length, int type) {
    jchar ch;

    for (char* p = name; p != name + length; ) {
        ch = *p;
```

```

        if (ch < 128) {
            p++;
            if (ch == '.' || ch == ';' || ch == '[') {
                return false; // do not permit '.', ';', or '['
            }
            if (type != LegalClass && ch == '/') {
                return false; // do not permit '/' unless it's class name
            }
            if (type == LegalMethod && (ch == '<' || ch == '>')) {
                return false; // do not permit '<' or '>' in method names
            }
        } else {
            char* tmp_p = UTF8::next(p, &ch);
            p = tmp_p;
        }
    } else {
        return true;
    }
}

```

这段逻辑表明，Java 方法名中不能包含如下特殊字符：

“.,;、[、/、<、>”

8.2 方法属性解析

在 ClassFileParser::parse_method()函数中，解析和校验完方法的签名信息，接着就开始解析 Java 方法的属性。在前文详细分析 Java class 字节码文件的结构时讲到，在字节码文件中使用几个大的属性来描述 Java 方法的方方面面的信息，这些属性包括 code、exception、line number table 等。

Java 方法的几大属性并不一定每个都会在字节码文件中出现，例如，如果 Java 方法没有抛出异常，则不会有异常表产生。同时，字节码文件中描述属性的字节码区域之间并不是有序的，因此，字节码文件中仅存储 Java 方法的属性的总数量，在 ClassFileParser::parse_method()函数中根据属性的总数量，通过 while 循环来逐个读取 Java 方法的当前属性，通过属性名来判断当前究竟是哪个属性，然后根据不同属性所承载的信息与结构之不同，分别使用不同的策略进行解析。

8.2.1 code 属性解析

Java 方法的 code 属性解析都集中在 ClassFileParser::parse_method()函数的 while 循环下的 if (method_attribute_name == vmSymbols::tag_code()) {} 块中。

Java 方法的 code 属性主要包含属性的总长度、最大栈深度、局部变量表数量、字节码指令，除此以外，code 属性本身是一个复合属性，其下面还包含几个子属性，例如行号表、局部变量表等。

Java 方法的 code 属性起始于属性名称的常量池索引号，索引号之后所跟的是属性长度，所以在 ClassFileParser::parse_method() 中主要解析 Java 方法几大属性的 while 循环中，首先便是执行 u2 method_attribute_name_index = cfs->get_u2_fast() 和 u4 method_attribute_length = cfs->get_u4_fast() 来分别获取当前属性的索引号和总长度。Java 方法的几大属性的索引号的数据宽度为 2 字节，总数据宽度为 4 字节。

紧跟在 code 属性的总长度后面的是 3 个属性：max_stack、max_locals 和 code_length。所以在 if(method_attribute_name == vmSymbols::tag_code()) {} 块中连续调用 3 次 get_u*_fast() 来分别获取这 3 个属性数据。在不同的 JVM 版本中，max_stack、max_locals 和 code_length 所占用的数据宽度不同，HotSpot 对此进行了区分，逻辑如下：

清单：/src/share/vm/classfile/classFileParser.cpp

作用：parse_method() 处理不同版本 JVM 的数据宽度

```
if (_major_version == 45 && _minor_version <= 2) {
    cfs->guarantee_more(4, CHECK_(nullHandle));
    max_stack = cfs->get_u1_fast();
    max_locals = cfs->get_u1_fast();
    code_length = cfs->get_u2_fast();
} else {
    cfs->guarantee_more(8, CHECK_(nullHandle));
    max_stack = cfs->get_u2_fast();
    max_locals = cfs->get_u2_fast();
    code_length = cfs->get_u4_fast();
}
```

在 Java 方法的 code 属性中，紧跟在 code_length 之后的就是 Java 源码所对应的字节码指令了，这部分指令最终会被从 Java class 字节码文件复制到内存中，具体而言是复制到 Java 方法在 JVM 内部所对应的 methodOop 对象的内存区域。由于当前阶段仍在 Java 方法属性的解析阶段，尚未创建 methodOop 对象，因此在这一步不会进行复制。但是 HotSpot 却通过 code_start = cfs->get_u1_buffer() 将字节码的第一条指令在 Java class 字节码文件中的位置记录下来，保存到 code_start 变量中。在前面解析出的 code_length 最终将会在后续创建 methodOop 时作为参数传递进去，最终 HotSpot 将依据 code_start 和 code_length 这两个数据确定从 Java class 字节码文件中要复制的字节码指令区域。具体复制的方式及复制的目标位置在后文再讲解，此处先略过不提。

注意：HotSpot 调用 cfs->get_u1_buffer() 函数来获取第一条字节码指令在字

节码文件中的位置，而非该位置处的值。在 HotSpot 内部，获取字节码文件某个位置的值，通常调用诸如 `get_u1_fast()` 这样的方法。

8.2.2 LVT&LVTT

在 Java 方法的属性中，有一种属性是局部变量表——LocalVariableTable，在 JVM 内部简写为 LVT。

LocalVariableTable 属性用于描述 Java 方法栈帧中局部变量表中的变量与 Java 源代码定义的变量之间的关系，这种关系并非运行时必需，所以默认情况下不会生成到 class 文件中。若想生成到 class 字节码文件中，则可以通过在 javac 命令中使用`-g:vars` 选项生成这项信息。

如果 Java class 字节码文件中没有生成局部变量表，则在调试 Java 程序时，无法看到源码中所定义的参数名称，IDE 可能使用 `arg0`、`arg1` 占位符替代原来的参数，这对程序运行没有任何影响，但会影响使用体验。

在 Java class 字节码文件中，LocalVariableTable 属性表的结构如表 8.1 所示。

表 8.1 LocalVariableTable 属性表结构

类 型	名 称	数 量
u2	attribute_name_index	1
u4	attribute_length	1
u2	local_variable_table_length	1
local_variable_info	local_variable_table	local_variable_table_length

注：表中的 `local_variable_info` 类型是一种特殊的复合数据结构，该复合结构用于描述 Java 方法栈帧与源代码中局部变量的关联，其结构如表 8.2 所示。

表 8.2 local_variable_info 类型的数据结构

类 型	名 称	数 量
u2	start_pc	1
u2	length	1
u2	name_index	1
u2	descriptor_index	1
u2	index	1

这 5 种属性的含义如下：

- ◎ start_pc，表示当前局部变量的生命周期开始的字节码偏移量。
- ◎ length，表示当前局部变量的作用范围覆盖长度，和 start_pc 一起就表示局部变量在字节码中的作用范围。
- ◎ name_index，当前局部变量的名称所对应的常量池的索引号。
- ◎ descriptor_index，当前局部变量的描述信息所对应的常量池的索引号。
- ◎ index，当前局部变量在栈帧局部变量中 slot 的位置，如果数据类型是 long 或 double (64 位)，slot 的位置为 index 和 index + 1。

研究局部变量表对于 Java 应用程序开发者而言没有具体的意义，但是这关乎 Java 程序调试的一些工程实现策略，也就是说，这种策略不仅在 Java 中有应用，在其他编程语言中也有类似实现。在调试过程中，如何让 IDE 在面对编译后的毫无意义的栈帧占位符的背景下，能够为开发者呈现出这些占位符所对应的源码中的原始变量名，是所有编程语言都需要具备的能力。既然 HotSpot 开放源代码，我们不妨顺道研究一番，虽然这个话题也许与主题关联不是很大。

HotSpot 的局部变量表的分析逻辑在 Java 编译器中实现，Java 编译器会对 Java 源码进行语法解析，分析 Java 方法的栈帧结构，并据此分析各个变量的作用域。分析的结果被以特定的组织结构存储在 Java class 字节码文件中，具体就是表 8.1 和表 8.2 所列。在 HotSpot 加载某个 Java 类时，会按照这种特定的组织结构还原出编译期所分析的结果，从而在调试期间能够据此显示出局部变量的原始名称。在 HotSpot 中，局部变量表还原的逻辑封装在 classFileParser.cpp::parse_localvariable_table() 函数中，如下所示：

清单：/src/share/vm/classfile/classFileParser.cpp

作用：parse_localvariable_table() 解析局部变量表

```
u2* ClassFileParser::parse_localvariable_table(u4 code_length,
                                                u2 max_locals,
                                                u4 code_attribute_length,
                                                constantPoolHandle cp,
                                                u2* localvariable_table_length,
                                                bool isLVT,
                                                TRAPS) {
    *localvariable_table_length = cfs->get_u2(CHECK_NULL);
    unsigned int size = (*localvariable_table_length) *
sizeof(Classfile_LVT_Element) / sizeof(u2);
    u2* localvariable_table_start = cfs->get_u2_buffer();
    if (!_need_verify) {
        cfs->skip_u2_fast(size);
    } else {
        cfs->guarantee_more(size * 2, CHECK_NULL);
```

```

for(int i = 0; i < (*localvariable_table_length); i++) {
    u2 start_pc = cfs->get_u2_fast();
    u2 length = cfs->get_u2_fast();
    u2 name_index = cfs->get_u2_fast();
    u2 descriptor_index = cfs->get_u2_fast();
    u2 index = cfs->get_u2_fast();
    u4 end_pc = (u4)start_pc + (u4)length;
}
return localvariable_table_start;
}

```

这段逻辑很简单，通过 `cfs->get_*_fast()` 函数从 Java class 字节码文件中读取连续的数据，分别获取局部变量表的总长度、`start_pc`、`length`、`name_index` 等数据。不过这段逻辑仅仅是将这些数据读取出来，进行了简单的校验便结束。真正的逻辑在 `classFileParser.cpp::parse_method()` 函数中：

清单：/src/share/vm/classfile/classFileParser.cpp

作用：parse_method()复制局部变量表

```

//初始化 hash 表，存储局部变量表，校验局部变量表是否存在重复
initialize_hashtable(lvt_Hash);
Classfile_LVT_Element* cf_lvt;

//局部变量表信息存储在 constMethodOop 对象实例的内存区域的末尾，在 byte code 之后
LocalVariableTableElement* lvt = m->localvariable_table_start();

for (tbl_no = 0; tbl_no < lvt_cnt; tbl_no++) {
    cf_lvt = (Classfile_LVT_Element *) localvariable_table_start[tbl_no];
    for (idx = 0; idx < localvariable_table_length[tbl_no]; idx++, lvt++) {

        //将局部变量表信息从 Java class 字节码文件中复制到 constMethodOop 对象实例的内存区域中
        copy_lvt_element(&cf_lvt[idx], lvt);

        //将当前局部变量表保存到 hash 表中，若保存失败则抛异常，表示有重复的局部变量表
        if (LVT_put_after_lookup(lvt, lvt_Hash) == false
            && _need_verify
            && _major_version >= JAVA_1_5_VERSION ) {
            clear_hashtable(lvt_Hash);
            classfile_parse_error("Duplicated LocalVariableTable attribute "
                "entry for '%s' in class file %s",
                cp->symbol_at(lvt->name_cp_index)->as_utf8(),
                CHECK_(nullHandle));
        }
    }
}

```

FileParser.cpp::parse_method()函数的这段逻辑，将局部变量表从 Java class 字节码文件复制到 Java 方法在 JVM 内部所对应的 constMethodOop 这个内部对象的内存区域，这个对象的内存布局结构在下文介绍。

在 JVM 运行期，在方法中打上断点，当 JVM 运行到断点处会执行中断而暂停，此时 JVM 能够通过栈帧上指向 methodOop 的指针定位到对应的 constMethodOop，由于局部变量表就保存在 constMethodOop 的内存区域的末尾位置，因此 JVM 能够基于 constMethodOop 进一步获取当前 Java 方法所有的局部变量表，从而在 IDE 中将方法入参和局部变量以原始的变量名显示出来。

在编译 Java 类时，是否启用生成局部变量表并放到字节码文件的选项，不仅会影响 IDE 调试时的体验，还会影响 Java 类库使用方的体验。在提供 class 文件给第三方使用时（不包含 Java 源码），如果没有开启-g:vars 选项，不生成局部变量表，则第三方将不会看到 Java 方法入参的原始名称。例如下面这个类：

清单：/Test.java

作用：演示字节码文件的局部变量表

```
public class Test{
    private long l;

    public void add(int aaa, int bbb){
        int z = aaa + bbb;
        long y = z + 5;
        int x = 3 + z;
    }

    public static void main(String[] args) throws Exception{
        Test test = new Test();
        test.add(2, 3);
    }
}
```

使用 javac 命令进行编译，不带-g:vars 选项，使用 eclipse 打开编译后生成的 class 文件，内容如下：

清单：/Test.class

作用：演示字节码文件的局部变量表

```
public class Test {
    private long l;

    public Test() {
    }
```

```

public void add(int var1, int var2) {
    int var3 = var1 + var2;
    long var4 = var3 + 5;
    int var6 = 3 + var3;
}

public static void main(String[] var0) throws Exception {
    Test var1 = new Test();
    var1.add(2, 3);
}
}
}

```

可以看到,无论是 add()方法还是 main()方法,其内部的变量名都变成了以 var 开头的名字,这是 IDE 自己的命名。IDE 根据字节码指令进行逆向编译时,只能分析出 java 方法包含几个人参和几个局部变量,但是由于字节码文件中没有存储变量名,因此 IDE 无法还原出源码中真实的变量名。

注意观察 IDE 自动生成的这几个 var 变量名,2 个人参被命名为 var1 和 var2,3 个局部变量被分别命名为 var3、var4 和 var6。中间貌似有不连贯的地方,缺失了 var5。IDE 为何要跳过 var5 直接将最后一个变量命名为 var6 呢?不难猜测,IDE 对变量进行自动命名的规则是 var 拼接上入参或变量所对应的 slot 索引号。由于 add()是 Java 类成员方法,因此其第 1 个人参是隐藏的 this,所以其源码中显式声明的两个人参才分别被命名为 var1 和 var2。而 add()方法中的变量 y 的类型是 long,JVM 规范规定 long 类型的数据在 slot 中占用 2 个槽位,所以最后一个变量 x 的 slot 索引号自然就变成了 6。再观察 main()方法,由于这是一个静态方法,所以并不包含 this 这个隐藏的入参,所以 main()方法中显式声明的第一个入参 String[] 被 IDE 自动命名成 var0。这里并不是要各位道友去研究 Java 逆向编译工程原理,只是要你明白,处处留心皆学问。

而带上-g:vars 选项进行 javac 编译后,使用 eclipse 打开编译后的 class 文件,所看到的方法内部的变量名与 Java 源码中的变量名完全一致。之所以会这样,是因为使用-g:vars 选项进行编译时,Java 方法内部的变量名称也会在字节码文件的常量池中存储,这是必须的,因为局部变量表中并没有直接存储变量名,而是将变量名存储到常量池并引用常量池的索引号。

下面举例对局部变量表加以说明,Java 示例便是上面所举的 Test 类,以 Test.add()方法作为分析对象。使用-g:vars 选项对 Test 类进行编译,并使用 javap 命令分析字节码文件,得到 add()方法的分析结果如下:

```

public void add(int, int);
Code:
Stack=2, Locals=7, Args_size=3
0:  iload_1
1:  iload_2
2:  iadd

```

```

3:  istore_3
4:  iload_3
5:  iconst_5
6:  iadd
7:  i2l
8:  lstore_4
10: iconst_3
11: iload_3
12: iadd
13: istore_6
15: return
LocalVariableTable:
Start  Length   Slot    Name     Signature
0      16       0       this    LTest;
0      16       1       aaa     I
0      16       2       bbb     I
4      12       3       z       I
10     6        4       y       J
15     1        6       x       I

```

先看局部变量表中的 this、aaa 和 bbb 这 3 个入参，Start 都是 0，Length 都是 16。这里的 0 表示这 3 个参数的作用域从第 1 个字节码指令就开始生效，length 为 16 是因为 add()方法字节码指令的总长度为 16，对于 Java 方法的入参，在方法内部任何地方都能引用到，所以其作用域的长度自然等于整个字节码指令的总长度。

接着看变量 z，其 Start 为 4，Length 为 12。在 add()方法中，变量 z 的声明与初始化被合二为一了，其声明语句为 int z = aaa + bbb，其对应的字节码指令包含 4 条，分别是列表中 bci(byte code index，字节码指令偏移量) 等于 0、1、2、3 的 4 条指令。在这 4 条指令之后的所有字节码指令都能引用变量 z，所以变量 z 的作用域就从 4 开始，而其作用范围自然是后续所有字节码指令的长度，为 $(16-4)=12$ 。

同样的机制，变量 y 的声明语句的字节码指令到 bci=8 的位置结束，bci=8 的指令是 lstore 4，一共占 2 字节长度，所以下一条字节码指令的 bci 为 10，这正是变量 y 的作用域开始的位置，所以变量 y 的 Start=10。

JDK 1.5 引入了泛型之后，为 LocalVariableTable 属性添加了一个“姐妹属性”：LocalVariableTypeTable。在 JVM 内部，这个属性简称为 LVTT，该属性的结构和 LocalVariableTable 相似，仅仅把记录的字段描述符的 descriptor_index 替换成字段特征签名(signature)。限于篇幅，本书不再描述该属性，有兴趣的道友可以自行研究。

8.3 创建 methodOop

完成对 Java 方法的各项属性解析之后，HotSpot 开始在内存中创建一个与 Java 方法对等的内部对象——methodOop。methodOop 包含 Java 方法的一切信息，例如方法名、返回值类型、入参、字节码指令、栈深、局部变量表、行号表等。其实一言以蔽之，HotSpot 通过 methodOop，将 Java class 字节码文件中的方法信息存储到了内存中，并且这片内存区域是结构化的，使得可以在 JVM 运行期方便地访问 Java 方法的各种属性信息。

ClassFileParser::parse_method() 函数中通过调用 oopFactory::new_method() 函数完成 methodOop 对象的创建，且看 oopFactory::new_method() 的实现：

清单：/src/share/vm/memory/oopFactory.cpp

作用：new_method()逻辑

```
methodOop oopFactory::new_method(int byte_code_size, AccessFlags access_flags,
                                 int compressed_line_number_size,
                                 int localvariable_table_length,
                                 int checked_exceptions_length,
                                 bool is_conc_safe,
                                 TRAPS) {
    methodKlass* mk = methodKlass::cast(Universe::methodKlassObj());
    assert(!access_flags.is_native() || byte_code_size == 0,
           "native methods should not contain byte codes");
    constMethodOop cm = new_constMethod(byte_code_size,
                                         compressed_line_number_size,
                                         localvariable_table_length,
                                         checked_exceptions_length,
                                         is_conc_safe, CHECK_NULL);
    constMethodHandle rw(THREAD, cm);
    return mk->allocate(rw, access_flags, CHECK_NULL);
}
```

oopFactory::new_method() 函数里面实际上创建了两个对象，分别是 methodOop 和 constMethodOop。其中 methodOop 通过调用(methodKlass*)->allocate()函数创建，而 constMethodOop 通过调用 oopFactory::new_constMethod()函数创建。

这里需要说明一下 constMethod 和 methodOop 的关系。无论在 JDK 6 还是最新的 JDK 8 中，都保留了这两个对象。这两个对象的作用不同，methodOop 主要存储 Java 方法的名称、签名、访问标识、解释入口等信息，而 constMethodOop 则用于存储方法的字节码指令、行号表、异常表等信息。

对于 JDK 6 和 JDK 8 而言，这两者有微小的变化。在 JDK 6 中，Java 方法的最大栈深度和

局部变量表数量存储在 methodOop 中，而到了 JDK 8，这两个数据则被存储到了 constMethodOop 中。这种变化仅影响到 HotSpot 为 Java 方法创建栈帧时读取这两个变量的逻辑。

创建 methodOop 的逻辑如下：

清单：/src/share/vm/oops/methodKlass.cpp

作用：allocate()逻辑

```
methodOop methodKlass::allocate(constMethodHandle xconst,
                                 AccessFlags access_flags, TRAPS) {
    int size = methodOopDesc::object_size(access_flags.is_native());
    KlassHandle h_k(THREAD, as_klassOop());
    methodOop m = (methodOop)CollectedHeap::permanent_obj_allocate(h_k, size,
CHECK_NULL);
    assert(!m->is_parsable(), "not expecting parsability yet.");

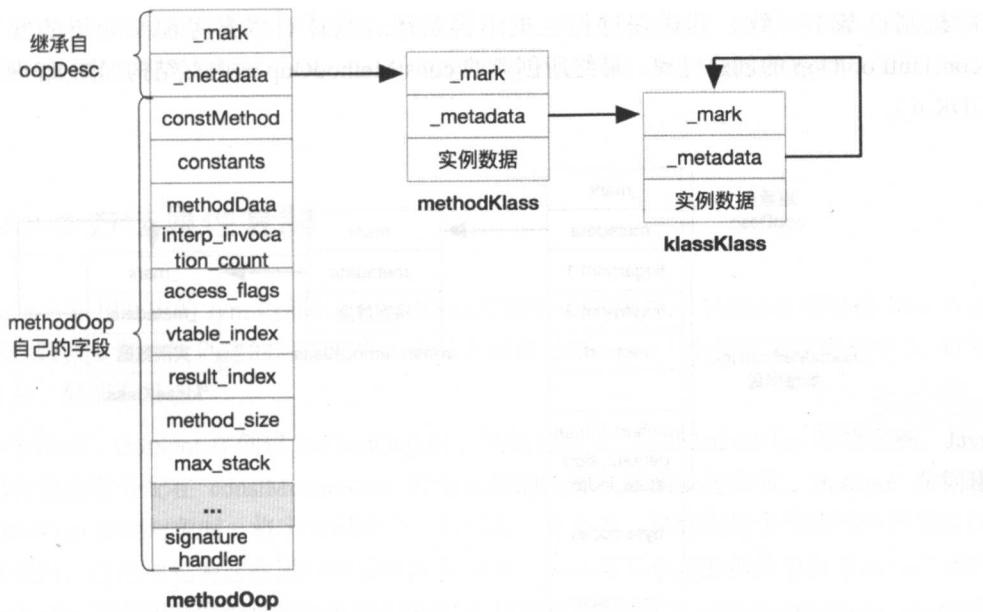
    No_Safepoint_Verifier no_safepoint; // until m becomes parsable below
    m->set_constMethod(xconst());
    m->set_access_flags(access_flags);
    //...m的一系列初始化

    xconst->set_method(m);
    return m;
}
```

前文曾经详细分析过常量池对象 constantPoolOop 的创建过程，在 classFileParser::parseClassFile()方法中调用 oopFactory::new_constantPool()函数创建常量池对象，后者调用 constantPoolKlass::allocate()函数完成常量池的创建。

仔细比较 constantPoolKlass::allocate()函数与这里的 methodKlass::allocate()函数，会发现两者逻辑基本一致，都是先求取 methodOop 的大小 size，接着调用 CollectedHeap::permanent_obj_allocate()函数在 perm 区（对于 JDK 8 而言则在 metaSpace 区）为所创建的对象分配内存。最后再调用一系列的 setter()方法初始化所创建的对象。

创建 methodOop 的详细过程这里不再赘述，与 constantPoolOop 的创建基本一致。最终创建出来的 methodOop 对象的内存布局如图 8.2 所示（基于 JDK 6 的内存模型）。

图 8.2 `methodOop` 对象的内存布局

虽然方法对象 `methodOop` 与常量池对象 `constMethodOop` 的创建机制基本相同，但是两者的数据结构仍然有所不同。`methodOop` 的内存结构与其类型定义的结构保持一致，而常量池 `constantPoolOop` 实例对象的内存的末尾还跟着常量池的元素数据。HotSpot 内部并没有为常量池的元素数组专门定义一种数据结构，这是因为不同 Java 类编译后所得到的常量池数组大小不同，并且各个元素成员的内存也完全不同，无法抽象成专门的数据结构，所以 HotSpot 只能将其分配到 `constantPoolOop` 实例对象的内存的末尾区域。但是 `methodOop` 与 `constantPoolOop` 一样，类型本身的字段并不足以保存 Java 方法的全部信息，例如 Java 方法的字节码指令、行号表等信息，在 `methodOop` 类型中并没有专门的字段存储这些信息，所以按理说 `methodOop` 也应该像 `constantPoolOop` 那样，将这些信息分配到 `methodOop` 实例对象的内存的末尾区域，但是 `methodOop` 并没有这么做。这是因为 HotSpot 为此专门另外定义了一种数据结构——`constMethodOop`，HotSpot 将 Java 方法的字节码指令及行号表等信息分配到了 `constMethodOop` 实例对象的内存的末尾区域。

在 `classFileParser::parseClassFile()` 函数调用 `oopFactory::new_method()` 函数创建 Java 方法对象的过程中，后者调用 `oopFactory::new_constMethod()` 函数完成 `constMethodOop` 的创建。在 `oopFactory::new_constMethod()` 函数中实际是调用 `constMethodKlass::allocate()` 函数完成 `constMethodOop` 的创建。`constMethodKlass::allocate()` 函数与 `constantPoolKlass::allocate()` 函数

的机制也基本一致，其详细过程这里不再赘述，具体可以参考前文介绍的常量池对象 constantPoolOop 的创建过程。最终所创建的 constMethodOop 的内存结构如图 8.3 所示（基于 JDK 6）。

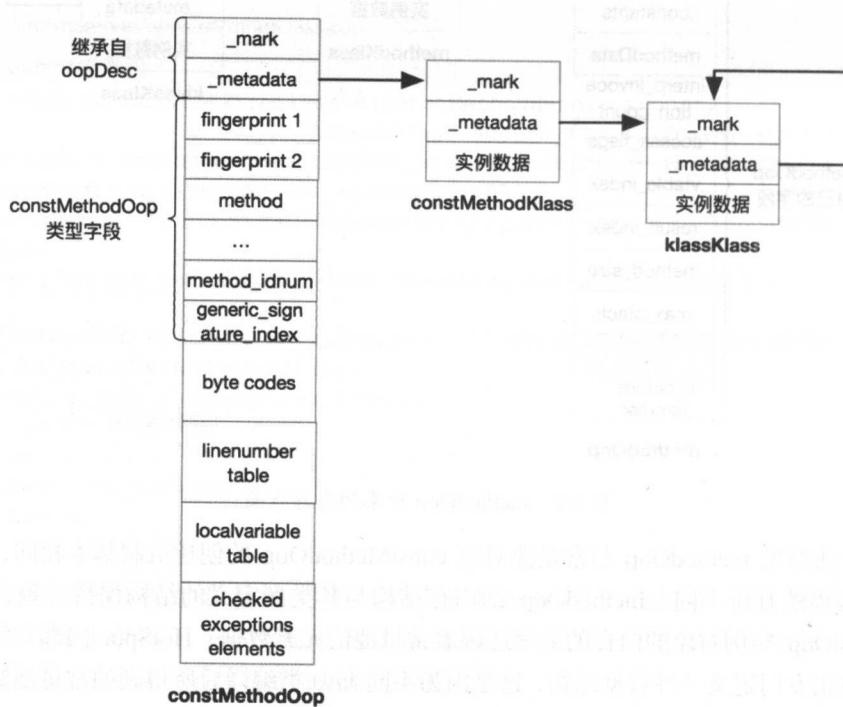


图 8.3 constMethodOop 内存布局

当 `oopFactory::new_constMethod()` 函数调用 `constMethodKlass::allocate()` 函数时，`constMethodKlass::allocate()` 函数的入参如下：

```
constMethodOop constMethodKlass::allocate(int byte_code_size,
                                         int compressed_line_number_size,
                                         int localvariable_table_length,
                                         int checked_exceptions_length,
                                         bool is_conc_safe,
                                         TRAPS)
```

可以看到，`constMethodKlass::allocate()` 函数入参包含字节码大小、行号表大小、局部变量表大小等信息，在 `constMethodKlass::allocate()` 函数内部所计算出的内存大小，是 `constMethodOop` 本身的大小与字节码大小、行号表大小等的总和，因为 HotSpot 将字节码指令、行号表、局部变量表等信息分配在 `constMethodOop` 对象实例的内存的末尾区域，所以在创建 `constMethodOop`

对象时，就将末尾区域所需要的内存提前申请和分配好。后续 HotSpot 会执行逻辑，将字节码指令、行号表、异常表等数据从 Java class 字节码文件中复制到 constMethodOop 末尾的这段内存中。

8.4 Java 方法属性复制

与 Java 方法相对应的 Jvm 内部的 methodOop 对象创建完成之后，HotSpot 需要将 Java 方法的几大属性数据复制进所创建的对象之中，这几大属性是指 code（主要是字节码指令）、行号表、异常表、局部变量表等。

上一节讲过，HotSpot 在创建 methodOop 时，顺便创建了 constMethodOop 对象实例，Java 方法的属性信息被分配在 constMethodOop 对象实例的内存区域的末尾位置。HotSpot 在创建 constMethodOop 实例对象时，将字节码指令、行号表、异常表、局部变量表等属性所需的空间大小计算在内，已经预先为这些属性申请好内存空间，所以接下来需要做的事就是将这些属性信息从 Java class 字节码文件中复制到申请的内存中。这段逻辑位于 ClassFileParser::parse_method() 函数中，如下：

清单：/src/share/vm/classfile/classFileParser.cpp

作用：复制 Java 方法属性的逻辑

```
//...
//复制异常表
m->set_exception_table(exception_handlers());

//复制字节码指令
m->set_code(code_start);

//复制行号表
if (linenumber_table != NULL) {
    memcpy(m->compressed_linenumber_table(),
           linenumber_table->buffer(), linenumber_table_length);
}
//...
```

这些逻辑大同小异，都是将前面步骤已经解析好的属性信息复制到对应的内存位置，其中局部变量表的复制逻辑已经在前文讲解 LVT 时进行过描述。这里重点关注字节码指令的处理，相信这也是很多道友所关心的问题。

字节码指令的处理主要是调用 m->set_code(address code_start) 函数，入参 code_start 在前置流程中已经解析出来，其值是当前 Java 方法的第一条字节码指令在读入内存中的字节码文件流

中的内存位置。

methodOop::set_code(address)函数逻辑如下：

清单：/src/share/vm/oops/methodOop.hpp

作用：复制 Java 方法字节码指令

```
void set_code(address code) {  
    return constMethod() -> set_code(code);  
}
```

本函数里面调用了 constMethod() -> set_code(address) 函数，该函数逻辑如下：

清单：/src/share/vm/oops/constMethodOop.hpp

作用：复制 Java 方法字节码指令

```
void set_code(address code) {  
    if (code_size() > 0) {  
        memcpy(code_base(), code, code_size());  
    }  
}
```

由此可见，constMethodOop::set_code(address) 函数最终调用了 memcpy() 函数，memcpy() 函数有 3 个形式入参，分别表示目的内存首地址、复制源内存首地址、复制长度。在复制 Java 方法的字节码指令时，第 2 个人参 code 已经包含明确的值，就是当前 Java 方法的第一条字节码指令在读入内存中的字节码文件流中的内存位置。而 constMethodOop::set_code(address) 函数第一个人参是 code_base() 函数的返回值，该函数逻辑如下：

清单：/src/share/vm/oops/constMethodOop.hpp

作用：复制 Java 方法字节码指令

```
address code_base() const {  
    return (address) (this+1);  
}
```

address 是指针类型，因此 this+1 表示 constMethodOop 实例对象的末尾位置的下一位，这正是 Java 方法字节码指令的存放位置。

到此为止，Java 方法字节码指令复制的实现逻辑便清楚了，字节码指令最终从 Java class 字节码文件中复制到了 constMethodOop 对象实例的内存的末尾位置。在 Java 程序运行期，HotSpot 将根据所调用的目标函数，找到该目标 Java 方法在内存中所对应的 methodOop 对象实例，并根据 methodOop 对象实例找到对应的 constMethodOop，最终基于 constMethodOop 定位到目标 Java 方法所对应的字节码指令，并将首个字节码指令的内存位置保存到目标 Java 方法的栈帧中，HotSpot 通过 JMP 硬件指令跳转到这个位置开始执行 Java 方法所对应的字节码指令，从而完成 Java 方法逻辑。

8.5 <clinit>与<init>

1. 初识<clinit>与<init>

在 Java 中，有两种特殊的方法，分别是<clinit>和<init>。这两个方法并非由 Java 开发者所定义，而是由 Java 编译器自动生成。当 Java 类中存在用 static 修饰的静态类型字段，或者存在使用 static{}块包裹的逻辑时，编译器会自动生成<clinit>方法。而当 Java 类定义了构造函数，或者其非 static 类成员变量被赋予了初始值时，编译器会自动生成<init>方法。看下面 Java 示例：

清单：Test.java

作用：演示<init>与<clinit>

```
public class Test{
    private Integer i = 3;
    private static int a = 90;

    public void add(int a, int b){
        Test test = this;
        int z = a + b;
        int x = 3;
    }

    public static void main(String[] args){
        Test test = new Test();
        test.add(2, 3);
    }
}
```

将该类编译后，使用javap命令查看编译后的class文件，显示如下：

```
$ javap -verbose Test
Compiled from "Test.java"
public class Test extends java.lang.Object
  SourceFile: "Test.java"
  minor version: 0
  major version: 50
  Constant pool:
const #1 = Method   #8.#24; //  java/lang/Object."<init>":()V
const #2 = Method   #25.#26; //  java/lang/Integer.valueOf:(I)Ljava/lang/Integer;
const #3 = Field    #4.#27; //  Test.i:Ljava/lang/Integer;
//...
const #11 = Asciz   a;
const #12 = Asciz   I;
```

```

const #13 = Asciz <init>;
const #14 = Asciz ()V;
//...
const #19 = Asciz main;
const #20 = Asciz ([Ljava/lang/String;)V;
const #21 = Asciz <<clinit>;
const #22 = Asciz SourceFile;
//...
const #33 = Asciz valueOf;
const #34 = Asciz (I)Ljava/lang/Integer;;

{
public Test();
Code:
Stack=2, Locals=1, Args_size=1
0:  aload_0
1:  invokespecial #1; //Method java/lang/Object."<init>":()V
4:  aload_0
5:  iconst_3
6:  invokestatic #2; //Method java/lang/Integer.valueOf:(I)Ljava/lang/Integer;
9:  putfield #3; //Field i:Ljava/lang/Integer;
12: return

public void add(int, int);
Code:
Stack=2, Locals=6, Args_size=3
0:  aload_0
//...

public static void main(java.lang.String[]);
Code:
Stack=3, Locals=2, Args_size=1
0:  new #4; //class Test
//...

static {};
Code:
Stack=1, Locals=0, Args_size=0
0:  bipush 90
2:  putstatic #7; //Field a:I
5:  return
}

```

通过 javap 命令可以看到，Test 类一共包含 4 个方法，其中 main() 和 add() 方法在 Test 类中被显式定义，然而另外 2 个方法却并未在 Test 类中进行过声明，分别是 public Test() 和 static {}，其实这 2 个方法便是由编译器自动生成的。由于 Test 类中有一个被 static 修饰的成员变量，因

此编译器自动生成了 static {}这样的方法，同时由于 Test 类的成员变量 i 在声明时就被赋予了初值，因此编译器自动生成了 public Test()这样的构造函数。

其实，这里的 public Test()和 static {}这两个方法，方法名分别是<init>和<clinit>，使用 javap 命令解析 Java class 文件时，在常量池中便有这两个方法的名称，索引号分别为 13 和 21，如下：

```
const #13 = Asciz  <init>;
const #21 = Asciz  <clinit>;
```

只是 javap 命令并没有像 main()和 add()方法那样直接将<init>和<clinit>这 2 个方法的名称显示出来，而是以 public Test()和 static {}这样的形式加以显示。

2. 使用 HSDB 查看<clinit>和<init>

通过 HSDB 这个神器，可以直观地显示出<init>和<clinit>这 2 个方法。使用 JDB 在 Test.add()方法内部打上断点并启动调试，然后启动 HSDB 并使其连上 Test 进程，接着单击 HSDB 的 Tool->Class browser 工具按钮，选择 Test 类，就能查看 Test 类中的全部方法（关于 JDB 工具和 HSDB 工具的使用，前文已有详细介绍，这里不再赘述），如图 8.4 所示。

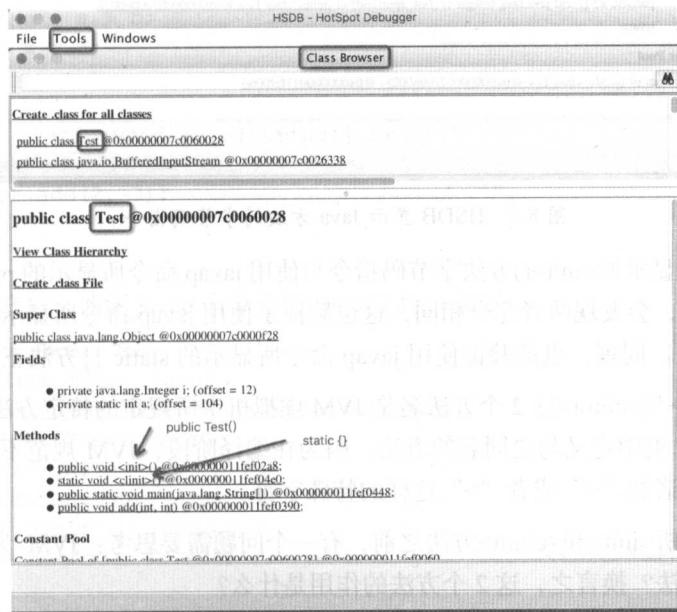


图 8.4 使用 HSDB 显示 Test 类的全部方法

图 8.4 显示，Test 类中的确存在名为<init>和<clinit>的两个方法，而这两个方法正对应于用 javap 命令所显示出来的 public Test()和 static {}方法。

在如图 8.4 所显示的 HSDB 的 Class Browser 视窗中单击某个方法，HSDB 会显示该方法的字节码指令，例如单击 public void <init>()方法名，显示的字节码指令如图 8.5 所示。

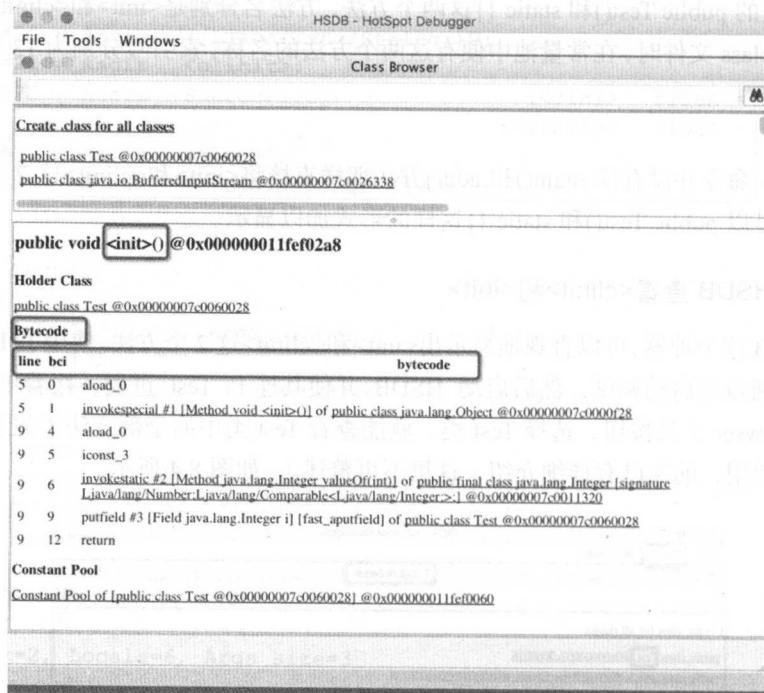


图 8.5 HSDB 显示 Java 方法的字节码指令

将 HSDB 中所显示的<init>()方法字节码指令与使用 javap 命令所显示的 public Test()方法字节码指令进行对比，会发现两者完全相同，这也验证了使用 javap 命令所显示的 public Test()方法正是<init>()方法。同理，也能验证使用 javap 命令所显示的 static {}方法正是<clinit>()方法。

事实上，<init>与<clinit>这 2 个方法名是 JVM 虚拟机中所规定的特定方法名，Java 应用开发者并不能在 Java 类中定义与之同名的方法，因为在编译阶段，JVM 规范不允许 Java 源程序中的方法名中存在诸如“<”或者“>”这样的特殊字符。

在继续深入分析<init>和<clinit>方法之前，有一个问题需要思考：JVM 为什么需要编译器自动生成这 2 个方法？换言之，这 2 个方法的作用是什么？

其实<init>方法好说，等同于类的构造函数，因此在遇到诸如 new 等指令时，自然会调用该方法。而当 Java 类中存在 static 成员变量，或者存在 static {}包裹的代码块时，则 JVM 加载 Java 类时，会触发<clinit>方法，完成 static 成员变量的初始化，或者执行被 static {}包裹的代码段的逻辑。

3. <init>详解

在上面的 Test 类的例子中，并没有在 Test 类中显式定义构造函数，但是编译器还是默认生成了一个构造函数，那么如果开发者显式定义一个无参的构造函数，编译器会如何处理呢？且举一例进行说明，将上述 Test 类实例稍微做下修改，添加一个无参的构造函数，修改后的 Test 类如下：

清单：Test.java

作用：演示<init>与<clinit>

```
public class Test{
    private Integer i = 3;
    private static int a = 90;

    public Test(){
        short s = 8;
    }

    public void add(int a, int b){
        Test test = this;
        int z = a + b;
        int x = 3;
    }

    public static void main(String[] args){
        Test test = new Test();
        test.add(2, 3);
    }
}
```

编译 Test 类之后，使用 javap 命令分析 class 文件，如下：

```
$ javap -verbose Test
Compiled from "Test.java"
public class Test extends java.lang.Object
  SourceFile: "Test.java"
  minor version: 0
  major version: 50
  Constant pool:
const #1 = Method    #8.#24; //  java/lang/Object."<init>":()V
//...
const #33 = Asciz   valueOf;
const #34 = Asciz   (I)Ljava/lang/Integer;;
{

public Test();
Code:
```

```
Stack=2, Locals=2, Args_size=1
0:  aload_0
1:  invokespecial #1; //Method java/lang/Object."<init>":()V
2:  aload_0
3:  iconst_3
4:  invokestatic #2; //Method java/lang/Integer.valueOf:(I)Ljava/lang/Integer;
5:  putfield      #3; //Field i:Ljava/lang/Integer;
6:  bipush       8
7:  istore_1
8:  return
9:  public void add(int, int);
10:  Code:
11:  //...
12:  
```

本次为 Test 类显式定义了一个无参的构造函数，这种改动主要影响<init>()方法。使用 javap 命令分析 Test.class 文件时，只需要观察 public Test()这部分的字节码指令的变化。在上面这段字节码指令中，能够看到如下指令：

```
5:  iconst_3
6:  invokestatic #2; //Method java/lang/Integer.valueOf:(I)Ljava/lang/Integer;
7:  putfield      #3; //Field i:Ljava/lang/Integer;
8:  bipush       8
9:  istore_1
10: 
```

其中 bci 等于 5、6、9 的字节码指令对应的 Java 源码是 Test 类的成员变量 i 的定义和赋值：private Integer i = 3。接着 bci 等于 12 和 14 的字节码指令对应的 Java 源码是 Test 构造函数中的 short s=8。由此可见，即使人为定义了默认的无参构造函数，编译器仍然会修改构造函数的字节码指令，将 Java 类成员变量的初始化指令都插入到构造函数<init>()方法的字节码指令中。这一点是<init>()方法与其他人为定义的 Java 类方法的最大之不同，对于人为定义的 Java 类方法，其字节码指令一定与 Java 源码保持一致，编译器不会随便往里加入其他逻辑。

前文讲过，当 Java 类的成员变量存在初始化行为时，其初始化的逻辑就会被嵌入到<init>()方法中。其实，还有一种情况也会如此，那就是在 Java 类中使用{}包裹的代码逻辑，也会被嵌入到<init>()方法中。修改 Test 类，如下：

清单：Test.java

作用：演示<init>与<clinit>

```
public class Test{
    private Integer i = 3;

    {
        System.out.println("i=" + i);
    }
}
```

```

public static void main(String[] args) {
    Test test = new Test();
}
}
}

```

编译后使用 javap 命令分析 Test.class 字节码文件，输出如下：

```

public Test();
Code:
Stack=3, Locals=1, Args_size=1
0:   aload_0
1:   invokespecial #1; //Method MyClass."<init>":()V
4:   aload_0
5:   iconst_3
6:   invokestatic #2; //Method java/lang/Integer.valueOf:(I)Ljava/lang/Integer;
9:   putfield      #3; //Field i:Ljava/lang/Integer;
12:  getstatic     #4; //Field java/lang/System.out:Ljava/io/PrintStream;
15:  new #5; //class java/lang/StringBuilder
18:  dup
19:  invokespecial #6; //Method java/lang/StringBuilder."<init>":()V
22:  ldc #7; //String i=
24:  invokevirtual #8; //Method
java/lang/StringBuilder.append:(Ljava/lang/String;)Ljava/lang/StringBuilder;
27:  aload_0
28:  getfield      #3; //Field i:Ljava/lang/Integer;
31:  invokevirtual #9; //Method
java/lang/StringBuilder.append:(Ljava/lang/Object;)Ljava/lang/StringBuilder;
34:  invokevirtual #10; //Method
java/lang/StringBuilder.toString:()Ljava/lang/String;
37:  invokevirtual #11; //Method
java/io/PrintStream.println:(Ljava/lang/String;)V
40:  return
LineNumberTable:
line 4: 0
line 9: 4
line 13: 12
line 14: 40

```

可以看出，上面有一大段字节码指令在处理对应的 Test 类源码中被 {} 包裹的块逻辑中的 System.out.println("i=" + i)。

现在再对 Test 类进行微调，刚才显式定义了一个无参的默认构造函数，现在则定义含入参的非默认的构造函数，同时将刚才无参的默认构造函数删除，修改后的 Test 类如下：

清单：Test.java

作用：演示<init>与<clinit>

```
public class Test {
```

```
private Integer i = 3;
private static int a = 90;

public Test(int x){
    x = 12;
}

public void add(int a, int b){
    Test test = this;
    int z = a + b;
    int x = 3;
}

public static void main(String[] args){
    Test test = new Test(10);
    test.add(2, 3);
}
}
```

编译 Test 类并使用 javap 命令分析 Test.class 文件，此时得到的字节码指令如下：

```
$ javap -verbose Test
Compiled from "Test.java"
public class Test extends java.lang.Object
  SourceFile: "Test.java"
  minor version: 0
  major version: 50
  Constant pool:
const #1 = Method    #8.#25; //  java/lang/Object."<init>":()V
//...
const #35 = Asciz   valueOf;
const #36 = Asciz   (I)Ljava/lang/Integer;;
{
public Test(int);
  Code:
    Stack=2, Locals=2, Args_size=2
    0:  aload_0
    1:  invokespecial #1; //Method java/lang/Object."<init>":()V
    4:  aload_0
    5:  iconst_3
    6:  invokestatic   #2; //Method java/lang/Integer.valueOf:(I)Ljava/lang/
Integer;
    9:  putfield      #3; //Field i:Ljava/lang/Integer;
    12: bipush     12
    14: istore_1
    15: return
```

```
public void add(int, int);
Code:
//...
```

可以看到,现在得到了 public Test(int)方法及其字节码指令,其字节码指令中的 bci 等于 5、6、9 的这 3 条字节码指令仍然对应 Test 类中的 private Integer i = 3 这一条源代码,然后 bci 等于 12、14 的这两条字节码指令对应 Test(int)这个含入参的构造函数中的 x = 12 这一句源代码。由此可见,当为 Java 类定义非默认的构造函数时,Java 编译器仍然会“擅自”修改构造函数的逻辑,将 Java 类成员的初始化逻辑所对应的字节码指令插入到构造函数所对应的字节码指令中。

到这里可以看出来,<init>()方法还是有点意思的,看不见的背后,其实编译器为我们做了很多工作。不过<init>()方法还有更多值得玩味的特性。刚才分别为 Test 类显式定义了一个无参的默认构造函数和一个含入参的构造函数,现在看看如果为一个 Java 类同时定义两个乃至多个构造函数,编译器如何处理。修改 Test 类,变成如下:

清单: Test.java

作用: 演示<init>与<clinit>

```
public class Test{
    private Integer i = 3;
    private static int a = 90;

    public Test(){
        short s = 8;
    }

    public Test(int x){
        x = 12;
    }

    public void add(int a, int b){
        Test test = this;
        int z = a + b;
        int x = 3;
    }

    public static void main(String[] args){
        Test test = new Test();
        test.add(2, 3);
    }
}
```

可以看到,现在 Test 类包含 2 个显式定义的构造函数,编译后使用 javap 命令分析 Test.class

文件，显示如下：

```
$ javap -verbose Test
Compiled from "Test.java"
public class Test extends java.lang.Object
  SourceFile: "Test.java"
  minor version: 0
  major version: 50
  Constant pool:
    const #1 = Method    #8.#25; //  java/lang/Object."<init>":()V
    //...
    const #35 = Asciz   valueOf;
    const #36 = Asciz   (I)Ljava/lang/Integer;;
  {
    public Test();
      Code:
        Stack=2, Locals=2, Args_size=1
        0:  aload_0
        1:  invokespecial #1; //Method java/lang/Object."<init>":()V
        4:  aload_0
        5:  iconst_3
        6:  invokestatic   #2; //Method java/lang/Integer.valueOf:(I)Ljava/lang/Integer;
        9:  putfield       #3; //Field i:Ljava/lang/Integer;
       12:  bipush     8
       14:  istore_1
       15:  return

    public Test(int);
      Code:
        Stack=2, Locals=2, Args_size=2
        0:  aload_0
        1:  invokespecial #1; //Method java/lang/Object."<init>":()V
        4:  aload_0
        5:  iconst_3
        6:  invokestatic   #2; //Method java/lang/Integer.valueOf:(I)Ljava/lang/Integer;
        9:  putfield       #3; //Field i:Ljava/lang/Integer;
       12:  bipush     12
       14:  istore_1
       15:  return

    public void add(int, int);
      Code:
        //...
```

仔细观察使用 javap 命令得到的输出中的 Test() 和 Test(int) 这两个方法的字节码，会发现这两个方法都包含 bci 等于 5、6、9 的这 3 条字节码指令，并且这 3 条字节码指令的含义相同，

都对应 Test 类中的 private Integer i = 3 这一句源代码。而这两个方法中，在 bci 为 9 之后的字节码指令则不再相同，因为两个构造函数内部的逻辑不同。由此可见，当为 java 类定义多个构造函数时，编译器会将类成员变量的初始化逻辑的字节码指令编排到每一个构造函数的字节码指令中。而道理其实很简单，因为不管是通过 new Test()去实例化 Test 类，还是通过 new Test(10)去实例化 Test 类，所实例化出的 Test 类的成员变量 i 都应该具有初始值 3，所以编译器必须将类成员变量的初始化逻辑原封不动地复制到每一个构造函数里去。

现在转换下视角，从继承的角度看<init>()方法。让 Test 类继承一个父类 MyClass，MyClass 定义如下：

清单： MyClass.java

作用：演示<init>与<clinit>

```
public abstract class MyClass {
    protected long plong = 12L;
}
```

修改 Test 类，使其继承 MyClass。为了节省篇幅，这里不贴出 Test 类。由于 Test 类继承了 MyClass 类，因此 Test 类便理所当然地继承了 MyClass 类中的成员变量 plong。当实例化 Test 类时，JVM 除了要完成 Test 类的成员变量的赋初值逻辑，也要完成继承自 MyClass 父类的成员变量的赋初值逻辑。因此，Java 编译器必然要将 MyClass 父类成员变量的初始化逻辑也嵌入到 Test 类的全部构造函数之中。

基于这种理论猜测来做验证——编译 Test 类，并使用 javap 命令分析 Test.class 字节码文件，javap 命令的输出如下：

```
$ javap -verbose Test
Compiled from "Test.java"
public class Test extends MyClass
  SourceFile: "Test.java"
    minor version: 0
    major version: 50
    Constant pool:
const #1 = Method    #8.#25; // MyClass."<init>":()V
//...
const #35 = Asciz   valueOf;
const #36 = Asciz   (I)Ljava/lang/Integer;;

{
public Test();
  Code:
    Stack=2, Locals=2, Args_size=1
    0:   aload_0
    1:   invokespecial #1; //Method MyClass."<init>":()V
```

```
4:  aload_0
5:  iconst_3
6:  invokestatic #2; //Method java/lang/Integer.valueOf:(I)Ljava/lang/Integer;
9:  putfield    #3; //Field i:Ljava/lang/Integer;
12: bipush     8
14: istore_1
15: return

public Test(int);
Code:
Stack=2, Locals=2, Args_size=2
0:  aload_0
1:  invokespecial #1; //Method MyClass."<init>":()V
4:  aload_0
5:  iconst_3
6:  invokestatic #2; //Method java/lang/Integer.valueOf:(I)Ljava/lang/Integer;
9:  putfield    #3; //Field i:Ljava/lang/Integer;
12: bipush     12
14: istore_1
15: return

public void add(int, int);
Code:
/...
```

观察 javap 命令的输出，在 public Test() 和 public Test(int) 这 2 个方法中，bci=1 的指令都是：

```
invokespecial #1; //Method MyClass."<init>":()V
```

可见，在 Test 类的两个构造函数中自动调用了父类的默认构造函数。Java 编译器将 MyClass 父类的成员变量的初始化逻辑封装到了 MyClass 父类自己的默认构造函数中，并通过将调用父类默认构造函数的逻辑嵌入到子类的各个构造函数中，从而在子类构造函数中完成父类成员变量的初始化逻辑。关于 MyClass.<init>() 方法，各位道友可以自行使用 javap 命令分析 MyClass.<init>() 方法的字节码指令，验证该方法有无嵌入 MyClass 类成员变量的初始化逻辑。

到了这里，笔者有个担心，如果父类有多个构造函数，那么编译器会让子类的构造函数调用父类的哪个构造函数呢？各位道友可以大胆猜测一把，这里直接通过试验进行验证。修改上述 MyClass 类，在其中定义两个构造函数，修改后的 MyClass 类如下：

清单：MyClass.java

作用：演示<init>与<clinit>

```
public abstract class MyClass {
    protected long plong = 12L;

    public MyClass() {
```

```

    plong = (long)'C';
}

public MyClass(int x){
    x = 32;
}
}
}

```

注意，上述类文件中为 MyClass 类定义了一个默认的无参构造函数，另一个则是含入参的非默认构造函数。

Test 类还是继承 MyClass 类，为节省篇幅，这里不贴出 Test 类。编译 Test 类并使用 javap 命令分析 Test.class 字节码文件，输出如下（且看子类 Test 的两个构造函数到底会调用父类的哪个构造函数）：

```

$ javap -verbose Test
Compiled from "Test.java"
//...
{
public Test();
Code:
Stack=2, Locals=2, Args_size=1
0:  aload_0
1:  invokespecial #1; //Method MyClass."<init>":()V
4:  aload_0
5:  iconst_3
//...
}

public Test(int);
Code:
Stack=2, Locals=2, Args_size=2
0:  aload_0
1:  invokespecial #1; //Method MyClass."<init>":()V
4:  aload_0
5:  iconst_3
//...
//...

```

javap 命令的输出显示，Test 的两个构造函数中被嵌入了调用 MyClass.<init>()这个默认的构造函数的逻辑。

上面举了多个例子，全方位分析了 Java 类的<init>()方法（即构造函数）的生成规则，这些规则可以总结为如下几点：

- ◎ 无论一个 Java 类有无定义构造函数，编译器都会自动生成一个默认的构造函数<init>()。可以使用javap命令或者HSDB来验证。
- ◎ <init>()方法主要完成Java类的成员变量的初始化逻辑，同时会执行Java类中被{}所包裹的块逻辑。如果Java类中的成员变量没有被赋初值，则在<init>()方法中不会对其进行初始化。
- ◎ 如果为Java类显式定义了多个构造函数，无论是否是默认的无参构造函数，Java编译器都会将Java类成员变量的初始化逻辑嵌入到每一个构造函数中，并且嵌入的位置在各构造函数自身逻辑之前。
- ◎ 当Java类显式继承了父类时，则Java编译器会让子类的各个构造函数调用父类的默认构造函数<init>()，从而在子类实例化时完成父类成员变量的初始化逻辑。
- ◎ 当父类中定义了多个构造函数时，子类构造函数会调用父类默认构造函数。
- ◎ 子类构造函数对父类默认构造函数的调用顺序，位于子类各个构造函数自身逻辑之前。

可以将上述6点总结成一句话：

一个类的各个构造函数的处理逻辑是，调用父类默认构造函数，完成类自身成员变量的初始化逻辑和被{}包裹的块逻辑，调用各构造函数自身逻辑。

4. <clinit>详解

前面详细分析了<init>()方法，而Java编译器除了会自动生成<init>()方法，还会自动生成<clinit>()方法。前文讲过，当Java类中存在static字段，或者被static{}包裹的代码逻辑时，就会自动生成<clinit>()方法。

关于<clinit>()方法也有很多有趣的特性，限于篇幅，本书不一一分析，各位道友可以按照前文分析<init>()方法的思路和角度来逐一分析<clinit>()方法的特性。这里仅举一复杂示例，直接得出结论。

示例如下：

清单：MyClass.java

作用：演示<init>与<clinit>

```
public abstract class MyClass {  
    protected long plong = 12L;  
  
    static {  
        Integer i = 30;  
    }  
}
```

```

public class Test extends MyClass{
    private Integer i = 3;
    private static int a = 156;

    static{
        int y = 128;
    }

    public void add(int a, int b){
        Test test = this;
        int z = a + b;
        int x = 3;
    }

    public static void main(String[] args){
        Test test = new Test();
        test.add(2, 3);
    }
}

```

上述代码中定义了两个类，Test 类继承了 MyClass 类。编译 Test 类，先使用 javap 命令分析 MyClass.class 字节码文件，输出如下：

```

$ javap -verbose MyClass
Compiled from "MyClass.java"
public abstract class MyClass extends java.lang.Object
  SourceFile: "MyClass.java"
  minor version: 0
  major version: 50
  Constant pool:
const #1 = Method  #7.#17; //  java/lang/Object."<init>":()V
//...
const #25 = Asciz  (I)Ljava/lang/Integer;;
{
  LineNumberTable:
    line 11: 0
static {};
Code:
Stack=1, Locals=1, Args_size=0
0:  bipush 30
2:  invokestatic #5; //Method java/lang/Integer.valueOf:(I)Ljava/lang/Integer;
5:  astore_0
6:  return
LineNumberTable:
line 11: 0

```

```
    line 12: 6  
}  
}
```

可以看到，MyClass 的 static {}方法中使用了 invokestatic 字节码指令完成变量 i 的初始化。
接着使用 javap 命令分析 Test.class 字节码文件，输出如下：

```
$ javap -verbose Test  
Compiled from "Test.java"  
public class Test extends MyClass  
  SourceFile: "Test.java"  
  minor version: 0  
  major version: 50  
  Constant pool:  
const #1 = Method    #8.#24; //  MyClass."<init>":()V  
//...  
const #34 = Asciz    (I)Ljava/lang/Integer;;  
  
{  
//...  
  
static {};  
Code:  
  Stack=1, Locals=1, Args_size=0  
  0:  sipush  156  
  3:  putstatic #7; //Field a:I  
  6:  sipush  128  
  9:  istore_0  
 10:  return  
}
```

可以看到，Test 类的 static {}方法中分别完成了成员变量 a 和局部变量 y 的初始化。由此可见，<clinit>()方法不具有继承性，原因其实也很简单，因为<clinit>()方法是在类加载过程中被调用，而父类与子类是分别加载的，当父类加载完之后，父类中的 static 成员变量初始化和被 static {} 所包裹的块逻辑已经执行完成，没必要在子类加载时再执行一次，所以子类只需完成自身 static 成员变量初始化以及被 static {} 所包裹的块逻辑即可。

5. init 与 clinit 执行顺序

<clinit>()方法在 Java 类第一次被 Jvm 加载时调用，而<init>()方法则在 Java 类被实例化时调用。由于类的加载一定位于类实例化之前，因此<clinit>()方法一定在<init>()方法之前被触发。并且每一次实例化 Java 类都会调用<init>()方法，而<clinit>()仅在类第一次加载时被调用，以后再加载时不会重复调用。

下面举例说明。

清单： MyClass.java

作用：演示<init>与<clinit>

```
public abstract class MyClass {
    protected long plong = 12L;

    {
        plong++;
        System.out.println("father.{}...plong=" + plong);
    }

    static {
        Integer i = 30;
        System.out.println("father.static{}...i=" + i);
    }

    public MyClass() {
        plong = (long)'C';
        System.out.println("father.constructor()...plong=" + plong);
    }
}

public class Test extends MyClass{
    private Integer i = 3;
    private static int a = 156;

    {
        i--;
        System.out.println("son.{}...i=" + i);
        System.out.println("son.{}...a=" + a);
    }

    static{
        a--;
        System.out.println("son.static{}...a=" + a);
    }

    public static void main(String[] args) {
        Test test = new Test();
    }
}
```

上面定义了 MyClass 和 Test 两个类，Test 类继承自 MyClass 类。并且 Test 和 MyClass 类中都包含{}块逻辑和 static {}块逻辑。运行 Test 类的 main()方法，输出如下：

father.static{}...i=30

```
son.static{}...a=155  
father.{...plong=13  
father.constructor()...plong=67  
son.{...i=2  
son.{...a=155
```

根据该输出顺序可知，JVM 先加载了父类 MyClass，调用了父类的 static {} 块逻辑，因此首先输出 “father.static {}...i=30”。接着又加载了子类 Test，因此调用了子类的 {} 块逻辑，因此接着输出 “son.static {}...a=155”。接着实例化 Test 类，前文总结过，类的构造函数的执行顺序是“调用父类的默认构造函数->执行类成员变量初始化逻辑和被 {} 包裹的块逻辑->执行构造函数自身逻辑”，由于 Test 类继承了 MyClass 类，因此 Test 的构造函数先执行父类的构造函数。而父类 MyClass 中包含被 {} 包裹的块逻辑，因此父类构造函数先执行块逻辑，再执行自身逻辑，所以接下来的输出便是按照这种顺序执行的结果。

后续章节会从源码的角度分析 jvm 先执行类的加载逻辑再调用类的实例化的逻辑，此处先略过不提。

6. {} 和 static{} 的作用域

在 Java 类源码中可以使用 {} 和 static {} 来包裹块逻辑，前面也有多个例子进行了演示。不过这两者的作用域却是各不相同。

被 {} 所包裹的块，能够访问 Java 类的非静态成员变量和静态成员变量，但是其内部所定义的变量不能被外部所访问，这一点与 Java 类的普通函数（非 static）的作用域完全相同。而在编译阶段，就连 {} 块中的局部变量也与 Java 类中的非 static 方法的局部变量一样，被组织成局部变量表。且看下面示例：

清单：Test.java

作用：演示 {} 作用域

```
public class Test{  
    private Integer i = 51;  
  
    {  
        int a = 1;  
        int b = 2;  
        int c = a + b;  
    }  
}
```

编译 Test 类，并使用 javap 命令分析 Test.class 字节码文件，输出的<init>() 构造函数的字节码指令如下：

```

public Test();
Code:
Stack=2, Locals=4, Args_size=1
0:  aload_0
1:  invokespecial #1; //Method java/lang/Object."<init>":()V
4:  aload_0
5:  bipush 51
7:  invokestatic #2; //Method java/lang/Integer.valueOf:(I)Ljava/lang/Integer;
10: putfield #3; //Field i:Ljava/lang/Integer;
13:  iconst_1
14:  istore_1
15:  iconst_2
16:  istore_2
17:  iload_1
18:  iload_2
19:  iadd
20:  istore_3
21:  return

```

其中, bci 等于 4、5、7、10 的这 4 条字节码指令的作用是完成 Test 类成员变量 i 的初始化。而从 bci=13 的字节码指令开始, 执行 Test 类中被 {} 包裹的逻辑。注意 bci=14 的字节码指令是 istore_1, 这表示将数字 1 赋值给 {} 块中的 a 变量, 这正暗示出 Java 编译器将 {} 块中的变量 a 处理成了局部变量表中的第 2 个局部变量(即第 2 个 slot)。同理, bci=16 和 bci=20 的两条字节码指令 istore_2 和 istore_3 也表明编译器将 {} 块中的第 2 和第 3 个局部变量 b 和 c 分别处理成了局部变量表中第 3 和第 4 个变量。

既然 Java 编译器将 {} 块中的局部变量处理成了局部变量表, 而局部变量表是函数相关的, 但是 {} 块逻辑并不是一个函数, 那么问题来了, 这里的局部变量表是谁的呢? 很显然, 由于 Java 编译器将 {} 块中的逻辑嵌入到了<init>() 构造函数中, 因此这里的局部变量表自然就是 Java 类的构造函数的了。

不过当一个 Java 类中有多个 {} 块逻辑时, 问题将变得更加有趣。将上述 Test 类稍加改造, 如下:

清单: Test.java

作用: 演示 {} 作用域

```

public class Test{
    private Integer i = 51;

    {
        int a = 1;
        int b = 2;
        int c = a + b;
    }
}

```

```

    {
        int x = 11;
        int y = 12;
        int z = x + y;
    }
}

```

现在 Test 类中有两个{}块逻辑。前面刚刚讲过，Java 编译器会将{}块逻辑中的局部变量处理成构造函数中的局部变量表，现在有两个块逻辑，那么一个关键的问题就随之出现了：这两个{}块逻辑中的变量在局部变量表中的编号会是怎样的？是否相关？

带着这个疑问，编译 Test 类并使用 javap 命令分析 Test.class 字节码文件，输出的<init>() 构造函数的字节码指令如下所示：

```

public Test();
Code:
Stack=2, Locals=4, Args_size=1
0:  aload_0
1:  invokespecial #1; //Method java/lang/Object."<init>":()V
4:  aload_0
5:  bipush 51
7:  invokestatic #2; //Method java/lang/Integer.valueOf:(I)Ljava/lang/Integer;
10: putfield #3; //Field i:Ljava/lang/Integer;
13:  iconst_1
14:  istore_1          //int a=1
15:  iconst_2
16:  istore_2          //int b=2
17:  iload_1
18:  iload_2
19:  iadd
20:  istore_3          //int c=a+b
21:  bipush 11
23:  istore_1          //int x=11
24:  bipush 12
26:  istore_2          //int y=12
27:  iload_1
28:  iload_2
29:  iadd
30:  istore_3          //int z=x+y
31:  return

```

上面将字节码指令与 Java 代码之间做了一一映射，Test 类中的 2 个{}块逻辑代码所对应的字节码指令都被嵌入到构造函数之中。注意观察这 2 个{}块逻辑中的局部变量在局部变量表中的编号（slot 索引），都是从 1 开始，也即第一个{}块逻辑中的 a 变量的 slot 索引是 1，而第二

个{}块逻辑中的 b 变量的 slot 索引也是 1。相应地，这 2 个块逻辑中的后续变量的 slot 索引编号都从 1 开始递增。由此可见，当 Java 类中存在多个{}块逻辑时，Java 编译器并不是简单地将其合并到构造函数之中，各个块中的局部变量的 slot 索引之间并不存在任何关联，唯一存在关联的就是各个{}块逻辑中的第一个局部变量的 slot 索引号相同。其实道理也很简单，由于 Java 类中的{}块就相当于一个独立的普通的 Java 函数，因此多个{}块中的局部变量之间彼此没有任何联系，相互不能引用，所以即使 Java 编译器将多个{}块逻辑都统一嵌入到构造函数之中，但是仍然保留了各个{}块逻辑的独立性。{}块中的局部变量既不能被其他{}块引用，更不会被构造函数引用到，因此当{}块逻辑执行完了之后，其局部变量表立即就被回收，被回收的局部变量表空间便被其他{}块的局部变量表所复用（其实并没有回收的动作，被复用就是被回收）。

讲完了{}块，接下来就轮到 static {}上场了。既然{}块可以被当作一个普通的 Java 方法处理，那么 static {}也自然可以被当作一个 static 修饰的 Java 静态方法处理，而事实上 JVM 也的确是这么规定的，所唯一不同的只是 static {}块逻辑是在 Java 类被加载时就执行，这是与其他 Java 类中静态方法的最大不同。

既然 static {}块可以被当作一个 Java 静态方法处理，那么这个块的作用域自然也就十分明了：

- ◎ 仅能访问 Java 类中的静态成员变量，而不能访问非静态成员变量。
- ◎ static {}块中所定义的局部变量不能被外部访问。

关于 static {}的以上这两条特性，诸位道友可以自行写示例程序进行验证，这里就不浪费纸张了。

7. <init>与<clinit>的访问标识

<init>()方法包括其一系列的重载方法（即 Java 类中所定义的非默认构造函数），直接被编译器处理成 public 类型，这从前文一系列 Java 示例程序的 javap 命令的输出中可以看出来。同时由于是构造函数，本身就没有返回类型。

而对于<clinit>()方法，在 HotSpot 源码中的 classFileParser.cpp::parse_method()方法中存在如下逻辑：

清单：/src/share/vm/classfile/classFileParser.cpp

作用：parse_method()设置<clinit>()方法签名

```
AccessFlags access_flags;
if (name == vmSymbols::class_initializer_name()) {
    if (_major_version < 51) { // backward compatibility
        flags = JVM_ACC_STATIC;
    } else if ((flags & JVM_ACC_STATIC) == JVM_ACC_STATIC) {
```

```

        flags &= JVM_ACC_STATIC | JVM_ACC_STRICT;
    }
} else {
    verify_legal_method_modifiers(flags, is_interface, name, CHECK_(nullHandle));
}

```

这段代码中的 `vmSymbols::class_initializer_name()` 就返回 `<clinit>`。通过这段源码可知，当 HotSpot 所解析的当前 Java 方法名是 `<clinit>` 时，就直接设置其访问标识是 `ACC_STATIC`。

8.6 查看运行时字节码指令

Java 方法经编译后就生成了字节码指令，在运行期字节码指令会被加载到 JVM 内存中，使用 HSDB 可以观察运行期的字节码指令，相信各位道友对此也非常感兴趣。

先准备一个示例程序：

清单：/Test.java

作用：观察运行期字节码指令

```

public class Test{
    private Integer i = 51;

    public void add(int a, int b){
        Test test = this;
        int z = a + b;
        int x = 3;
    }

    public static void main(String[] args){
        Test test = new Test();
        test.add(2, 3);
    }
}

```

使用 JDB 在 `Test.add()` 方法处打上断点并调试运行 `Test` 程序，运行至断点处，然后中断程序，接着使用 HSDB 连接上 `Test` 进程，观察内存。

注意，使用 JDB 调试 `Test` 程序时，必须加上 `-XX:-UseCompressedOops` 选项，这样迫使 JVM 放弃默认的指针压缩功能，否则无法显示内存对象的真实地址，给内存观察带来不便。

字节码指令被分配在 `constMethodOop` 对象的内存区域的末尾，因此只要在 HSDB 中能够查到 `add()` 方法所对应的 `constMethodOop` 所在的内存位置，就能基于此定位到字节码指令的内存位置，并进一步查看内存中的字节码指令长成啥样。

在 HSDB 中单击工具栏的 Tools->Class Browser 按钮，搜索 Test 类，可以看到该类的所有方法，如图 8.6 所示。

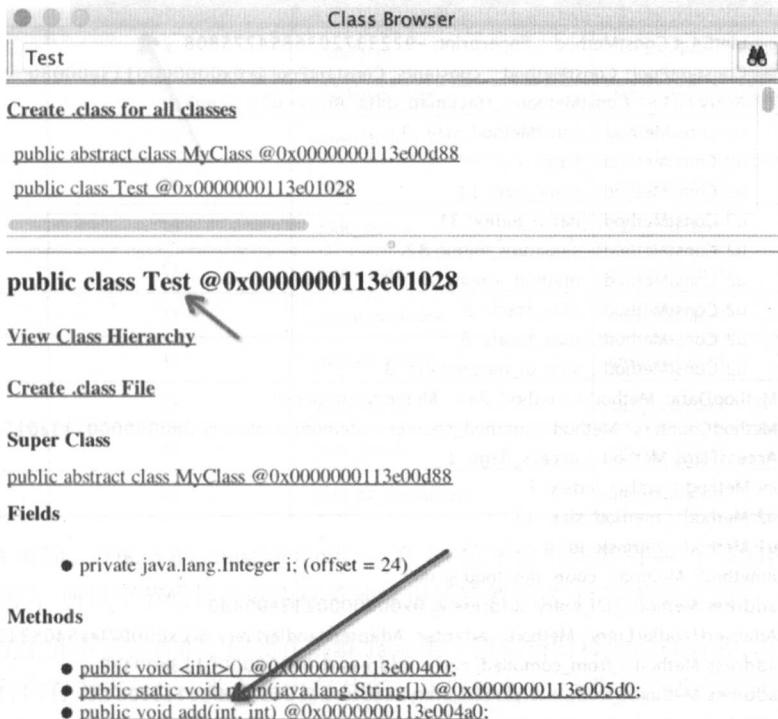


图 8.6 在 HSDB 中查看 Test 类的所有方法

图 8.6 中显示出 Test.add() 方法，HSDB 显示出该方法的签名和地址，其地址是 0x0000000113e004a0。

接着单击 HSDB 工具栏的 Tools->Inspector 按钮，输入 Test.add()方法的地址并回车。Inspector 工具将分析 Test.add()方法在 JVM 内部所对应的 MethodOop 对象实例的内存结构以及各个字段的内存地址，其中就包括我们关心的 constMethodOop 字段的内存地址。如图 8.7 所示。

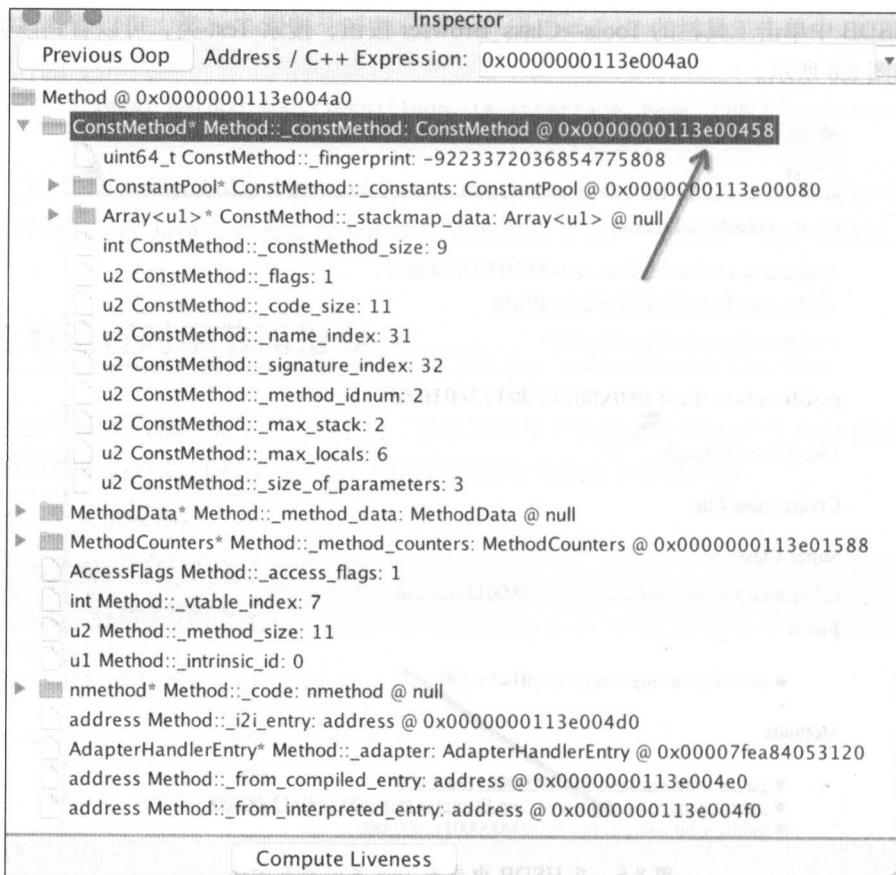


图 8.7 使用 HSDB 分析 Java 方法的内部结构

由于 MethodOop 类型包含 constMethodOop 的引用，因此在图 8.7 中可以看到 constMethodOop 字段的内存地址，该地址为 0x0000000113e00458。这个地址正是 Test.add()方法在 JVM 内部所对应的 constMethodOop 对象实例的内存地址，而 Test.add()方法的字节码指令就跟在 constMethodOop 对象实例的内存后面。HSDB 没有直接给出字节码指令的查看工具，因此只能通过计算字节码指令的起始内存地址并使用 Mem 工具去查看字节码指令。字节码指令的起始内存地址的计算很简单，只需要使用 constMethodOop 的内存首地址加上该对象实例所占内存空间的大小，就得到字节码指令的内存首地址。

本示例运行在 64 位平台上，并且使用 JDB 工具进行调试时添加了-XX:-UseCompressedOops 选项，因此 constMethodOop 的所有指针类型的字段都占用 8 字节的内存空间，constMethodOop 对象内各个字段的偏移量如表 8.3 所示（基于 JDK 8）。

表 8.3 constMethodOop 字段偏移量

占用内存	偏 移	字 段	类 型
8	0	_fingerprint	volatile unsigned __int64
8	8	_constants	ConstantPool *
8	16	_stackmap_data	Array<unsigned char> *
4	24	_constMethod_size	int
2	28	_flags	unsigned short
2	30	_code_size	unsigned short
2	32	_name_index	unsigned short
2	34	_signature_index	unsigned short
2	36	_method_idnum	unsigned short
2	38	_max_stack	unsigned short
2	40	_max_locals	unsigned short
2	42	_size_of_parameters	unsigned short

由表 8.3 可知，JDK 8 的 constMethodOop 在 64 位平台上共占用 44 字节的内存空间（关闭指针压缩功能），所以字节码指令的内存首地址很显然就在这 44 字节的后面。

但是事实上并不是这样的，前面在讲解 Java 类字段解析时提到内存对齐，不仅 Java 类的各个字段会进行内存对齐，C++类也会内存对齐，并且整个 C++类也需要内存对齐。C++类对齐的其中一条规则就是整个类型实例所占的内存空间必须是类型中宽度最大的字段所占内存的整数倍。由于 constMethodOop 中包含指针，因此该类实例对象所占的内存空间必须是 8 字节的整数倍，因此最终 constMethodOop 所占的内存大小应该是 48 字节。

constMethodOop 的大小计算出来之后，便可以计算字节码指令的内存起始地址，计算公式很简单：

$$0x00000000113e00458 + 0x30 = 0x00000000113e00488$$

注：48 对应的十六进制数是 0x30。

计算出字节码指令的起始地址后，可以使用 HSDB 所提供的 mem 工具查看这个地址处的内容。激动人心的时刻就要来临了！

单击 HSDB 工具栏的 Windows->Command Line 按钮打开 HSDB 的命令行工具，在里面输入 mem 0x00000000113e00488 2，输出如下内容：

```
hsdb> mem 0x00000000113e00488 2
```

```
0x0000000113e00488: 0x060436601c1b4eca  
0x0000000113e00490: 0x29116800ffb10536
```

上面这个 mem 命令的最后加了一个选项 2，表示从 0x0000000113e00488 这个内存位置开始，连续输出两行内容，在 64 位平台上，一行显示 128 字节内容。

这一串数字看不懂？很正常。对于不正常的事物，需要分析其规律，然后才能分析出其本质。JVM 的每个字节码指令都占 1 字节长度，并且 Test.add()方法里的局部变量的值也都小于 255，因此每个变量的值也都只占 1 字节的长度，这些变量将作为带参数的字节码指令的操作数，例如 istore 4，istore 指令后面紧跟的 4，在内存中仅使用 1 字节来表示，并且作为字节码指令的操作数，其内存位置与字节码指令所在的内存相邻，这一点很好理解。

弄清楚这一规律，现在来分析 mem 0x0000000113e00488 2 命令的输出结果。输出结果一共 2 行，每行按照从高位到低位的顺序显示，但是字节码指令在内存中的分配顺序实际上是从低位到高位，即 Java 方法的第一条字节码指令所在的内存地址，一定是在最低位，而 Java 方法的最后一条字节码指令，则一定在最高位。所以这里首先要将 mem 的输出结果的顺序进行逆转，将其按照从低位到高位的顺序重新排列，这样才符合人类的阅读习惯。对于 Test.add()方法，其每个字节码指令及操作数皆使用 1 字节表示，因此在对 mem 输出结果的顺序进行逆转时，也以 1 字节为单位进行逆转。内存重排后的内容如下：

```
0x0000000113e00488: ca 4e 1b 1c 60 36 04 06  
0x0000000113e00490: 36 05 b1 ff 00 68 11 29
```

内存重排后，面对这一串数字，还是看不懂？别着急，先来看 Test.add()方法所对应的字节码指令的十六进制数字。使用 javap 命令分析编译后的 Test.class 文件，得到如下结果：

```
public void add(int, int);  
  Code:  
    Stack=2, Locals=6, Args_size=3  
0:   aload_0  
1:   astore_3  
2:   iload_1  
3:   iload_2  
4:   iadd  
5:   istore_4  
7:   iconst_3  
8:   istore_5  
10:  return
```

使用 javap 命令只能得到字节码指令的名称，但是内存里实际存储的是字节码的十六进制编码。对照字节码指令集的十六进制编码表（这张编码表就不在本书贴出来了，从网络可以搜索到），将上述字节码指令名称及操作数逐个翻译成十六进制编码，翻译后的结果如下：

```

public void add(int, int);
Code:
Stack=2, Locals=6, Args_size=3
0:   aload_0           2a
1:   astore_3          4e
2:   iload_1           1b
3:   iload_2           1c
4:   iadd              60
5:   istore 4          36 04
7:   iconst_3          06
8:   istore 5          36 05
10:  return             b1

```

Test.add()方法的第一条字节码指令是 `aload_0`, 其十六进制编码是 `2a`, 该字节码指令的内存地址就是 `add()`方法的字节码指令在 `constMethodOop` 之后的首地址, 因此其内存地址就是上文所分析得到的 `0x0000000113e00488`, 也即上面使用 `mem` 命令所观察的内存地址。以 `aload_0` 指令的十六进制编码为开始, 将 `add()`方法的字节码指令的十六进制编码以字节为单位顺序编排, 如下:

```

0x0000000113e00488: 2a 4e 1b 1c 60 36 04 06
0x0000000113e00490: 36 05 b1

```

将这个结果与上面使用 `mem` 命令所查看到的内存内容进行比较, 从第一个字节到 `add()`方法所对应的最后一条字节码指令的十六进制编码 `b1` 逐一比较, 会发现除了第一个字节之外, 其余的都是完全相同的。字节码指令存储的奥秘到此便完全揭晓。

不过仍然有一个疑问, 为何第一个字节码的内容不一样呢? 这是因为为了使用 HSDB 观察 Test 程序的运行时内存结构, 使用了 JDB 进行调试, 并且就在 `Test.add()`方法内部打上了断点, 所以 JVM 将 `add()`方法的入口地址改写成断点指令的。等 `add()`方法从中断恢复之后, 第一条字节码指令会恢复正常。

其实对于字节码指令的内存位置的计算, 除了上面所述的通过 `constMethodOop` 内存首地址加上该对象实例所占的内存大小的方法之外, 还有一个更简单的办法, 那就是直接查看堆栈(前面章节已经详细讲解过堆栈的结构)。Java 的每一个方法都会在 JVM 的内存中被分配一个栈帧空间, 并且栈帧中会保存一个指针指向 Java 方法的字节码指令的内存首地址, 而 HSDB 可以观察到运行期的 Java 方法的堆栈内容, 因此可以使用这种方式直接定位到 Java 方法的字节码指令所在的内存地址。

使用 JDB 在 `Test.add()`方法内打断点并调试运行至断点处, 接着使用 HSDB 连接 Test 程序。HSDB 刚连上 Test 程序时会显示 JVM 的线程, 其中包含主线程, 如图 8.8 所示。

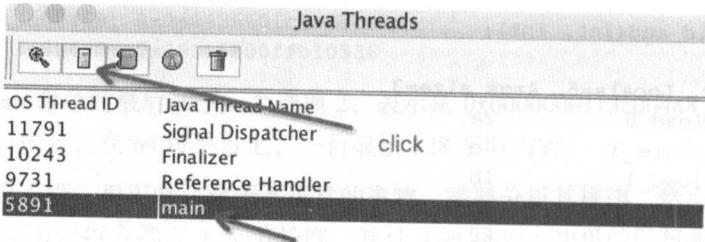


图 8.8 在 HSDB 中显示 Java 进程的主线程

选择主线程 main，并单击工具栏中的第 2 个按钮，HSDB 便会显示这个主线程的堆栈结构。由于 JDB 在 Test.add()方法内部打了断点，并且 Test 程序已经运行至 Test.add()方法，因此 Test 主线程的堆栈会包含 add()方法的栈帧结构。而 add()方法的栈帧中有一项是一个指针，该指针指向 add()方法所对应的 Java 字节码指令的内存地址，如图 8.9 所示。

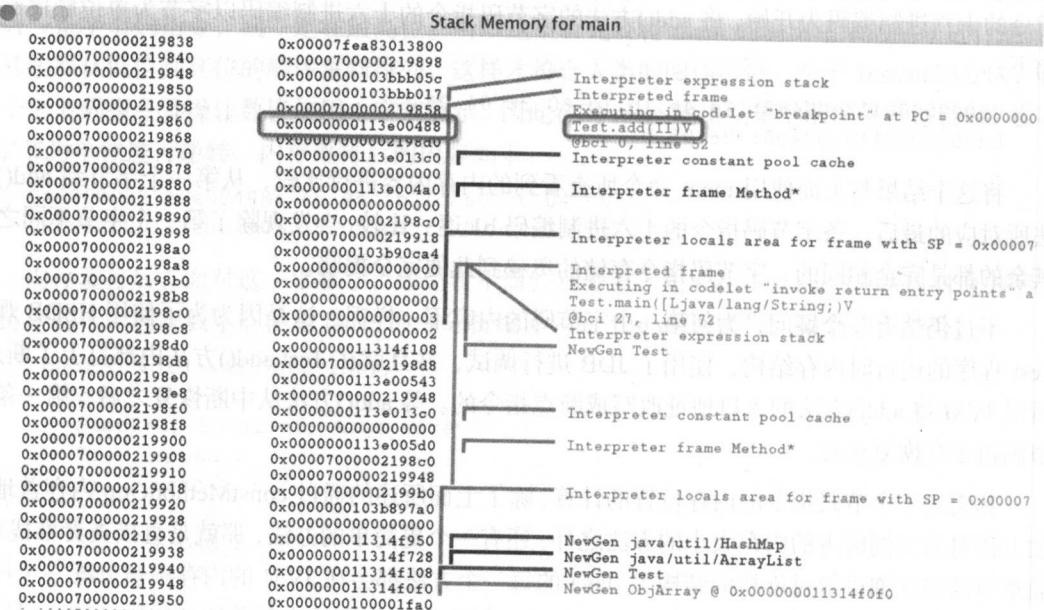


图 8.9 使用 HSDB 查看 Java 方法堆栈

图 8.9 中所框住的内容便是 Test.add()方法的字节码指令的内存首地址。这个地址与上面所计算出来的地址完全相同。通过本示例也可以猜想 JVM 的垃圾回收机制，如果一个内存对象没有被栈帧引用或者被栈帧所引用的对象的堆所引用，则必定是垃圾对象，则 JVM 的垃圾回收器便可以进行回收。不过本书限于篇幅，不会讲解垃圾回收的机制和实现，将放到下一本书中进行分析。

8.7 vtable

本节开始讲解 Java 多态中的一个十分关键的技术——vtable。

8.7.1 多态

Java 是一门面向对象的编程语言，面向对象的一大特色便是多态。多态的具体体现便是在运行期能够根据对象实例的不同而执行不同的接口方法，换成业界对多态的标准定义便是：允许不同类的对象对同一消息做出响应，即同一消息可以根据发送对象的不同而采用多种不同的行为方式（发送消息就是函数调用）。多态是面向对象编程的特性，而这种特性并不仅仅是喊喊口号就算的，而是必须使用特定的机制或技术去实现。实现多态的技术称为动态绑定（dynamic binding），是指在执行期间判断所引用对象的实际类型，根据其实际的类型调用其相应的方法。在 Java 中，动态绑定也叫“晚绑定”，这是因为在 Java 中还有一类绑定是在编译期间便能确定，所以所谓的晚绑定的概念，是相对于编译期绑定而言的。

面向对象编程语言之所以要实现多态这一特性，最主要的目的就是为了消除类型之间的耦合关系，通俗地讲就是解耦。从计算机软件一产生，“解耦”便是一切计算机程序所要重点考虑的原则之一。其实何止是软件，计算机硬件之间也是以解耦为主要原则的，这类例子举不胜举，例如内存插槽、IO 接口之类，都是实现解耦的手段。

解耦的最大好处在于，一旦系统发生了变化，能够将变化降低到最小，仅变化新增的部件，而对于已经存在的部件，则尽量保持不变。所以一个优秀的系统设计师总是想办法设计拥有良好兼容性和扩展性的架构，而面向对象语言的多态性，则是从语言特性上直接实现对象的解耦，这极大地提升了面向对象编程语言构建一套高内聚、低耦合系统的能力。

由于多态通过“动态绑定”的方式得以实现，而绑定通俗一点讲就是让不同的对象对同一个函数进行调用，或者反过来讲就是将同一个函数与不同的对象绑定起来，所以多态性得以实现的一个大前提就是，编程语言必须是面向对象的，否则哪来的函数与对象相互绑定一说呢？同时，函数与对象相互绑定，意味着函数也属于对象的一部分，这便具备了封装的特性。因为有了封装，才有了对象。有了对象才叫作面向对象编程。同时，一个函数能够绑定多个不同的对象，意味着多个不同的对象都具有相同的行为，这是继承的含义。因此，面向对象编程语言的三大特性——封装、继承与多态，其中前两个特性“封装”与“继承”其实就是为了第三个特性“多态”而准备的，或者说“封装”与“继承”成全了“多态”，为“多态”做了嫁衣。

下面是一个简单的动态绑定的示例程序：

清单：/Test.java

作用：演示动态绑定

```
public abstract class Animal {  
    abstract void say();  
  
    public static void main(String[] args){  
        Animal animal = new Dog();  
        run(animal);  
  
        animal = new Cat();  
        run(animal);  
    }  
  
    public static void run(Animal animal){  
        animal.say();  
    }  
}  
  
class Dog extends Animal{  
    @Override  
    void say() {  
        System.out.println("I'm a dog");  
    }  
}  
  
class Cat extends Animal{  
    @Override  
    void say() {  
        System.out.println("I'm a cat");  
    }  
}
```

本示例程序中定义了虚类 Animal，同时定义了 2 个子类 Dog 和 Cat，这 2 个子类都重写了基类中的 say()方法。在 main()函数中，将 animal 实例引用分别指向 Dog 和 Cat 的实例，并分别调用 run(Animal)方法。在本示例中，当在 Animal.run(Animal)方法中执行 animal.say()时，因为在编译期并不知道 animal 这个引用到底指向哪个实例对象，所以编译期无法进行绑定，必须等到运行期才能确切知道最终调用哪个子类的 say()方法，这便是动态绑定，也即晚绑定，这是 Java 语言以及绝大多数面向对象语言的动态机制最直接的体现。

8.7.2 C++中的多态与 vtable

JVM 实现晚绑定的机制基于 vtable，即 virtual table，也即虚方法表。JVM 通过虚方法表在运行期动态确定所调用的目标类的目标方法。在讲解 JVM 的 vtable 概念之前，先一起品味 C++ 中虚方法表的实现机制，这两者有很紧密的联系。

有如下 C++类：

清单：/Test.cpp

作用：演示 C++的动态绑定

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

class CPLUS{
public:
    short x;
public:
    void run(){
        this->x=2;
    }
};

int main(int argc, char const *argv[])
{
    CPLUS cplus;
    printf("sizeof(CPLUS)= %lu\n", sizeof(CPLUS));
    printf("&cplus= %p\n", &cplus);
    printf("&(cplus.x)= %p\n", &(cplus.x));

    return 0;
}
```

这个 C++示例很简单，类中包含一个 short 类型的变量和一个 run()方法，在 main()函数中打印 3 个信息：CPLUS 类型宽度、其实例的内存首地址和其变量 x 的内存地址。编译并执行，输出结果如下：

```
sizeof(CPLUS)= 2
&cplus= 0x7fff5ef57998
&(cplus.x)= 0x7fff5ef57998
```

由于 CPLUS 类中仅包含 1 个 short 类型的变量，因此该类型的数据宽度自然是 2。另外注意观察结果中的 cplus 实例和其变量 x 的内存地址，两者是相等的。

现在将 C++类中的 run()方法修改一下，变成虚方法，修改后的程序如下：

清单：/Test.cpp

作用：演示 C++ 的动态绑定

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

class CPLUS{
public:
    short x;
public:
    virtual void run() {
        this->x=2;
    }
};
```

main() 函数内容不变，因此这里不重复贴出来。编译并运行程序，现在输出变成如下所示：

```
sizeof(CPLUS)= 16
&cplus= 0xffff5694d990
&(cplus.x)= 0xffff5694d998
```

注意看，现在 sizeof(CPLUS) 的值变成 16 了，并且 cplus 实例和其变量 x 的内存地址也不再相等了。这是怎么回事呢？这是因为当 C++ 类中出现虚方法时，表示该方法拥有多态性，此时会根据类型指针所指向的实际对象而在运行期调用不同的方法，这与 Java 中的多态在语义上是完全一致的。

C++ 为了实现多态，就在 C++ 类实例对象中嵌入虚函数表 vtalbe，通过虚函数表来实现运行期的方法分派。C++ 中所谓虚函数表，其实就是一个普通的表，表中存储的是方法指针，方法指针会指向目标方法的内存地址。所以虚函数表就是一堆指针的集合而已。

对于大部分 C++ 编译器而言，其实现虚函数表的机制都大同小异，都会将虚函数表分配在 C++ 对象实例的起始位置，当 C++ 类中出现虚函数表时，其内存分配就是先分配虚函数表，再分配类中的字段空间。以本示例程序而言，CPLUS 的实例对象 cplus 的实际内存结构如图 8.10 所示。

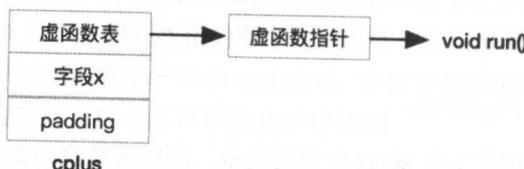


图 8.10 cplus 实例内存结构布局

由于 CPLUS 类中仅包含一个虚函数，因此虚函数表中只有一个指针。

注意，在 cplus 实例的末尾有一段补白空间，这是因为 C++ 编译器会对类型做对齐处理，整个 C++ 类实例对象所占的内存空间必须是其中宽度最大的字段所占内存空间的整数倍，而 CPLUS 类中由于嵌入了虚函数表，表中元素是指针类型，在 64 位平台上，1 个指针占 8 字节内存空间，因此 CPLUS 类实例对象所占的内存空间就是 16 字节，这就是上面运行示例程序后输出结果中的 sizeof(CPLUS) 的值变成 16 的原因所在。同时，字段 x 被安排在虚函数表之后，因此 x 的内存地址也不再与 cplus 实例对象的内存首地址相等，并且根据上述程序运行结果可见，这两者的内存地址相差 8 字节，这正好是一个指针的宽度。

8.7.3 Java 中的多态实现机制

Java 的多态机制并没有跳出这个圈，也采用了 vtable 来实现动态绑定。Java 类在 JVM 内部对应的对象是 instanceKlassOop (JDK 8 中是 instanceKlass)。在 JVM 加载 Java 类的过程中，JVM 会动态解析 Java 类的方法及其对父类方法的重写，进而构建出一个 vtable，并将 vtable 分配到 instanceKlassOop 内存区的末尾，从而支持运行期的方法动态绑定。

JVM 的 vtable 机制与 C++ 的 vtable 机制之间最大之不同在于，C++ 的 vtable 在编译期间便由编译器完成分析和模型构建，而 JVM 的 vtable 则在 JVM 运行期、Java 类被加载时进行动态构建。其实也可以认为 JVM 在运行期做了 C++ 编译器在静态编译期做的事情。关于 C++ 编译器在编译期间构建 vtable 的机制，不同的编译器具体的实现方式不尽相同，但基本都没跳出这个框框。各位有兴趣的道友如果想对此进行研究，可以分析 C++ 类编译后的汇编脚本。

JVM 在第一次加载 Java 类时会调用 classFileParser.cpp::parseClassFile() 函数对 Java class 字节码进行解析，上文刚刚讲过，在 parseClassFile() 函数中会调用 parse_methods() 函数解析 Java 类中的方法，parse_methods() 函数执行完之后，会继续调用 klassVtable::compute_vtable_size_and_num_mirandas() 函数，计算当前 Java 类的 vtable 大小。下面便一起来围观该方法实现的主要逻辑：

清单：/src/share/vm/oops/klassVtable.cpp

作用：verify_legal_method_name() 方法主要逻辑

```
void klassVtable::compute_vtable_size_and_num_mirandas(int &vtable_length,
                                                       int &num_miranda_methods,
                                                       klassOop super,
                                                       objArrayOop methods,
                                                       AccessFlags class_flags,
                                                       Handle classloader,
                                                       Symbol* classname,
```

```

    objArrayOop local_interfaces,
    TRAPS
) {
    vtable_length = 0;
    num_miranda_methods = 0;

    instanceKlass* sk = (instanceKlass*)super->klass_part();
    vtable_length = super == NULL ? 0 : sk->vtable_length();

    int len = methods->length();
    for (int i = 0; i < len; i++) {
        methodHandle mh(THREAD, methodOop(methods->obj_at(i)));

        if (needs_new_vtable_entry(mh, super, classloader, classname,
        class_flags, THREAD)) {
            vtable_length += vtableEntry::size(); // we need a new entry
        }
    }

    num_miranda_methods = get_num_mirandas(super, methods, local_interfaces);
    vtable_length += (num_miranda_methods * vtableEntry::size());
}

//...
}

```

从这段代码可以看出计算 vtable 的思路主要分为两步：

(1) 获取父类 vtable 的大小，并将当前类的 vtable 的大小设置为父类 vtable 的大小。

(2) 循环遍历当前 Java 类的每一个方法，调用 needs_new_vtable_entry() 函数进行判断，如果判断的结果是 true，则将 vtable 的大小增 1。

关于父类 vtable 的大小需要从 Java 类的顶级父类 java.lang.Object 开始算起，这个一会儿再讲，现在重点看第 2 步——needs_new_vtable_entry() 函数的实现逻辑：

清单：/src/share/vm/oops/klassVtable.cpp

作用：needs_new_vtable_entry() 方法主要逻辑

```

bool klassVtable::needs_new_vtable_entry(methodHandle target_method,
                                         klassOop super,
                                         Handle classloader,
                                         Symbol* classname,
                                         AccessFlags class_flags,
                                         TRAPS) {
    //如果 Java 方法被 final、static 修饰，或者 Java 类被 final 修饰，或者 Java 方法是构造
    //函数，则返回 false
    if ((class_flags.is_final() || target_method()->is_final()) ||
        (target_method()->is_static()) ||

```

```

        (target_method()->name() == vmSymbols::object_initializer_name())
    )
    return false;
}
//...
//如果 Java 方法的访问权限是 private 则返回 true
if (target_method()->is_private()) {
    return true;
}

//遍历父类中同名、签名也完全相同的方法，如果父类方法的访问权限是 public 或者 protected，并且没有 static 或 private 修饰，则说明子类重写了父类的方法，此时返回 false
ResourceMark rm;
Symbol* name = target_method()->name();
Symbol* signature = target_method()->signature();
klassOop k = super;
methodOop super_method = NULL;
instanceKlass *holder = NULL;
methodOop recheck_method = NULL;
while (k != NULL) {
    super_method = instanceKlass::cast(k)->lookup_method(name, signature);
    if (super_method == NULL) {
        break;
    }

    instanceKlass* superk = instanceKlass::cast(super_method->method_holder());
    if ((!super_method->is_static()) &&
        (!super_method->is_private())) {
        if (superk->is_override(super_method, classloader, classname, THREAD)) {
            return false;
        }
    }
    k = superk->super();
}

//处理 miranda 方法
instanceKlass *sk = instanceKlass::cast(super);
if (sk->has_miranda_methods()) {
    if (sk->lookup_method_in_all_interfaces(name, signature) != NULL) {
        return false;
    }
}
return true;
}

```

在分析这段逻辑之前，有必要稍微解释一下 vtable 的机制。Java 类在运行期进行动态绑定

的方法，一定会被声明为 public 或者 protected 的，并且没有 static 和 final 修饰，且 Java 类上也没有 final 修饰。道理很简单，阐述如下：

- ◎ 如果一个 Java 方法被 static 修饰，则压根儿不会参与到整个 Java 类的继承体系中，所以静态的 Java 方法不会参与到运行期的动态绑定机制。所谓的动态绑定，是指将 Java 类实例与 Java 方法搭配，而静态方法的调用压根儿不需要经过类实例，只需要有类型名即可。
- ◎ 如果一个 Java 方法被 private 修饰，则外部根本无法调用该方法，只能被该类内部的其他方法所调用，因此也不会参与到运行期的动态绑定。
- ◎ 如果一个 Java 方法被 final 修饰，则其子类无法重写该方法（如果非要重写，则 IDE 会报错的，并且编译也不会通过）。既然子类无法重写该方法，则该方法仅为 Java 类所固有，不会出现多态性。
- ◎ 如果一个 Java 类被 final 修饰，则该 Java 类中的所有非静态方法都会隐式地被 final 修饰，参考第 3 条，则该 Java 类中的所有非静态方法都不会被子类重写，因此都不会出现多态性，不会发生运行期动态绑定。

只有满足以上 4 个条件的 Java 方法，才有可能参与到运行期的动态绑定，而满足了以上 4 个条件的 Java 方法，其一定只被 public 或者 protected 关键字修饰。而满足了这 4 个条件的 Java 方法只是有可能参与动态绑定，这是因为仅仅满足了这 4 个条件还不够，还得有别的条件，其余的条件包括以下：

- ◎ 父类中必须包含名称相同的 Java 方法。这里所谓的父类，并不一定是 Java 类的直接父类，也可能是间接的父类。例如 A 类继承了 B 类，B 类继承了 C 类，则 A 类间接继承了 C 类。
- ◎ 父类中名称相同的 Java 类，其签名也必须完全一致。

以上这两点其实换而言之，就是 Java 类中的方法必须重写了父类的 Java 方法，这样的 Java 类方法才会参与到运行期动态绑定。而这其实正是多态的含义：父类与子类中包含一个完全一样的行为（即 Java 方法的名称和签名完全相同），在运行期这一行为将根据不同的条件与具体的类型对象相绑定（父类或子类实例）。

而父类与子类都同时拥有的相同的行为，从继承的角度看，就是子类对父类的重写，并且重写的前提是，这一行为不能是 private 的，不能是 static 的，不能是 final 的，否则父类的行为无法被子类继承，所谓的重写更无从谈起。所以上面这段代码的逻辑就是在进行这一系列的判断，只有最终满足条件的，才会返回 true。

在 klassVtable.cpp::compute_vtable_size_and_num_mirandas()函数中,如果 needs_new_vtable_entry()函数返回 true, 将会对 vtable_length 增加 1。

现在我们描述 JVM 内部 vtable 的实现机制。每一个 Java 类在 JVM 内部都有一个对应的 instanceKlassOop, vtable 就被分配在这个 oop 内存区域的后面。vtable 表中的每一个位置存放一个指针, 指向 Java 方法在内存中所对应的 methodOop 的内存首地址。如果一个 Java 类继承了父类, 则该 Java 类就会直接继承父类的 vtable。若该 Java 类中声明了一个非 private、非 final、非 static 的方法, 若该方法是对父类方法的重写, 则 JVM 会更新父类 vtable 表中指向父类被重写的方法的指针, 使其指向子类中该方法的内存地址。若该方法并不是对父类方法的重写, 则 JVM 会向该 Java 类的 vtable 中插入一个新的指针元素, 使其指向该方法的内存位置。

相信很多人对这段文字看得有些晕, 还是举个例子。假设有下面一个超类:

清单: /A.java

作用: Java 的多态机制

```
class A {
    public void print(){
        System.out.println("A.print()");
    }
}
```

接着定义一个子类:

清单: /B.java

作用: Java 的多态机制

```
class B extends A{
    public void newFun(){
        System.out.println("B.newFun()");
    }

    public void print(){
        System.out.println("B.print()");
    }
}
```

子类 B 继承于类 A, 并且重写了类 A 的 void print()方法。由于类 B 和类 A 中的 print()方法名称相同, 签名也完全相同, 并且都没有 private、static、final 这 3 个关键字修饰, 因此该方法将会在运行期进行动态绑定。

当 HotSpot 在运行期加载类 A 时, 其 vtable 中将会有两个指针元素指向其 void print()方法在 HotSpot 内部的内存首地址。当 HotSpot 加载类 B 时, 首先类 B 完全继承其父类 A 的 vtable, 因此类 B 便也有一个 vtable, 并且 vtable 里有一个指针指向类 A 的 print()方法的内存地址。

HotSpot 遍历类 B 的所有方法，并发现 print()方法是 public 的，并且没有被 static、final 修饰，于是 HotSpot 去搜索其父类中名称相同、签名也相同的方法（即上文所讲的 `klassVtable.cpp::needs_new_vtable_entry()` 函数），结果发现父类中存在一个完全一样的方法，于是 HotSpot 就会将类 B 的 vtable 中原本指向类 A 的 print()方法的内存地址的指针值修改成指向类 B 自己的 print()方法所在的内存地址。

而当 HotSpot 解析类 B 的 newFun()方法时，由于该方法并没有在父类中出现，并且也是 public 的，同时没有 static 和 final 修饰，满足 vtable 的条件，于是 HotSpot 将类 B 原本继承于 A 的 vtable 的长度增 1，并将新增的 vtable 的指针元素指向 newFun()方法在内存中的位置。

在 `classFileParser.cpp::parseClassFile()` 函数中，执行完 `klassVtable.cpp::compute_vtable_size_and_num_mirandas()` 函数后，会得到当前 Java 类的 vtable 的大小，虚函数表的大小被保存在 `classFileParser.cpp::parseClassFile()` 函数的局部变量 `vtable_size` 中，该变量值将在后续创建 Java 类所对应的 `instanceKlassOop` 对象时被保存到该对象中的 `_vtable_len` 字段中，而该字段可以通过 HSDB 进行查看，并进而验证 HotSpot 计算虚函数表大小的逻辑。

写个测试类调用上述示例中的类 A 或 B 中的方法并设断点，让程序运行到断点处中断，然后使用 HSDB 连上测试程序（由于测试程序很简单，因此这里就不贴出来），单击 HSDB 工具栏上的 Tools->Class Browser 按钮，会看到类 A 和类 B，如图 8-11 所示。

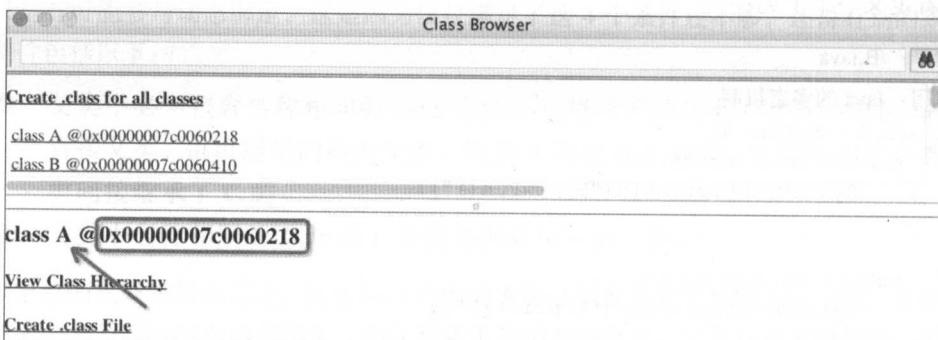


图 8.11 使用 HSDB 查看类 A 的内存地址

复制图 8.11 中的类 A 的地址 `0x00000007c0060218`，单击 HSDB 工具栏的 Tools->Inspector 按钮，将该地址复制进去，可以查看类 A 在 JVM 内部所对应的 `instanceKlassOop` 的内部结构，其中就有类 A 所对应的 vtable 虚函数表的大小，如图 8.12 所示。

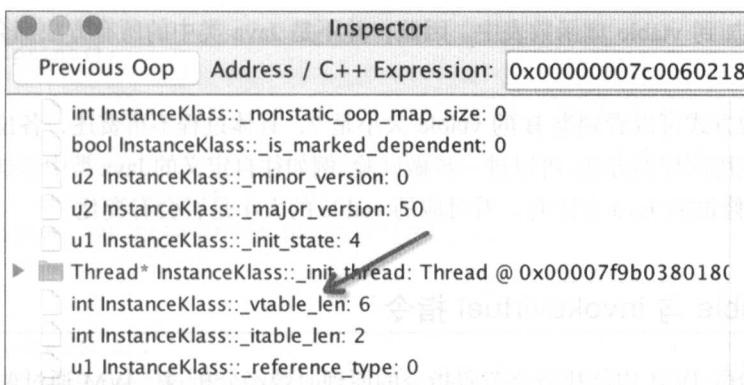


图 8.12 使用 HSDB 查看类 A 所对应的 instenceKlassOop 的 vtable 大小

如图 8.12 中箭头所示,类 A 的 vtable 大小为 6。这里有个疑问,类 A 中只定义了一个方法,按理说其 vtable 大小应该是 1,可是为何 HSDB 里却显示 6 呢?其实,在 Java 语言中,所有 Java 类都隐式继承了顶级父类 java.lang.Object,类 A 的 vtable 的另外 5 个方法指针元素便指向 java.lang.Object 中的 5 个方法。java.lang.Object 中一共定义了如下 12 个方法:

```
protected void finalize()
public final void wait(long, int)
public final native void wait(long)
public final void wait()
public boolean equals(java.lang.Object)
public java.lang.String toString()
public native int hashCode()
public final native java.lang.Class getClass() [signature ()Ljava.lang.
Class<*>;]
protected native java.lang.Object clone()
private static native void registerNatives()
public final native void notify()
public final native void notifyAll()
```

而其中只有如下 5 个方法可以被子类重写,因此 java.lang.Object 类的 vtable 大小为 5,这 5 个方法如下:

```
protected void finalize()
public boolean equals(java.lang.Object)
public java.lang.String toString()
public native int hashCode()
protected native java.lang.Object clone()
```

这 5 个方法都是 public 或者 protected 的,并且没有用 static 和 final 修饰,而 java.lang.Object 中的其他方法要么被 final 修饰,要么被 static 修饰,因此都不能被子类继承,所以其方法指针

不会被 JVM 添加到 vtable 虚函数表中。因此，并不是 Java 类中的所有方法都会被放入 vtable 中。

使用同样的方式可以看到类 B 的 vtable 大小是 7，具体过程不再赘述，各位道友可自行试验分析。同时，使用同样的办法，可以进一步做试验，例如往自定义的 Java 类中添加使用 private、static 或者 final 修饰的 Java 方法时，看对应的 vtable 的大小是否会有变化。

8.7.4 vtable 与 invokevirtual 指令

本书一开始讲 JVM 内部执行字节码指令的原理时曾经分析过，JVM 通过调用 CallStub 例程开始调用 Java 程序的主函数 main()，在 CallStub 例程内部最终调用了 zero_locals 这个例程，而在 zero_locals 例程中最终跳转到 Java 程序的 main() 主函数的第一条字节码指令并开始执行 Java 程序。那么在 Java 程序内部，当一个 Java 方法调用了另一个 Java 方法时，是如何实现 Java 方法的调用的？这得从 Java 的字节码指令开始说起。

Java 的字节码指令中方法的调用实现分为 4 种指令：

- ◎ Invokevirtual，为最常见的情况，包含 virtual dispatch（虚方法分发）机制。
- ◎ Invokespecial，调用 private 和构造方法，绕过了 virtual dispatch。
- ◎ Invokeinterface，其实现与 invokevirtual 类似。
- ◎ Invokestatic，调用静态方法。

其中最复杂的要属 invokevirtual 指令，它涉及多态的特性，凡是 Java 类中需要在运行期动态绑定的方法调用，都通过 invokevirtual 指令，该指令将实现方法的分发，因此 vtable 与该指令之间有莫大的联系，而事实上，在 HotSpot 执行 invokevirtual 指令的过程中，最终会读取被调用的类的 vtable 虚函数表，并据此决定真实的目标调用方法。JVM 内部实现 virtual dispatch 机制时，会首先从 receiver（被调用方法的对象）的类的实现中查找对应的方法，如果没找到，则去父类查找，直到找到函数并实现调用，而不是依赖于引用的类型。这里还是先来看一个例子，感受一下上面这 4 个指令的用法：

清单：/Test.java

作用：vtable 与 invokevirtual 字节码指令验证

```
public class Test {  
    public static void main(String[] args) {  
        A a = new A();  
        a.print();  
  
        B b = new B();  
    }  
}
```

```

        b.newFun();
        b.print();
    }
}

class A {
    public void print(){
        System.out.println("A.print()");
    };
}

class B extends A{
    public void newFun(){
        System.out.println("B.newFun()");
    }

    public void print(){
        System.out.println("B.print()");
        newFun();
        privateFun();
    }

    private void privateFun(){ }
}

```

编译这段程序，并使用javap命令分别分析Test.main()函数中的3个方法调用的字节码指令，以及类B.print()方法中调用newFun()和privateFun()时所使用的字节码指令。

首先看B类的print()方法调用newFun()和privateFun()时的字节码指令，javap分析的结果如下所示：

```

public void print();
Code:
Stack=2, Locals=1, Args_size=1
0:  getstatic      #2; //Field java/lang/System.out:Ljava/io/PrintStream;
3:  ldc #5; //String B.print()
5:  invokevirtual   #4; //Method
java/io/PrintStream.println:(Ljava/lang/String;)V
8:  aload_0
9:  invokevirtual   #6; //Method newFun:()V
12:  aload_0
13:  invokespecial  #7; //Method privateFun:()V
16:  return

```

由javap分析的结果可知，在B.print()方法中调用newFun()和privateFun()方法时所使用的字节码指令分别是invokevirtual和invokespecial，为何在类B内部调用自己的这两个方法，所

使用的指令竟然还有所不同呢？

这是因为 newFun() 是 public 的，并且没有 static 和 final 修饰，因此这个方法是可以被继承的，并且是可以被子类重写的。而编译器在编译期间并不知道类 B 有没有子类，因此这里只能使用 invokevirtual 指令去调用 newFun() 方法，从而使 newFun() 方法支持在运行期进行动态绑定。虽然编译器在编译期间可以分析整个工程以确定类 B 到底有无子类，但是别忘了 JVM 可是能够支持在运行期动态创建新的类型（例如，使用 cglib）的，编译器根本无法得知在运行期会不会突然冒出个类去继承类 B 并重写类 B 的 newFun() 方法。

而 privateFun() 方法则不一样，其为类 B 的私有方法，就算有子类继承于类 B，也无法重写该方法，因此该方法不需要参与动态绑定，在编译期间便能直接确定其调用者，所以其对应的字节码指令是 invokespecial。

与 B.print() 方法中调用 newFun() 方法使用 invokevirtual 指令同样的道理，在 Test 类的 main() 主函数中调用 a.print()、b.print() 和 b.newFun() 这 3 个方法时，所对应的字节码指令也都是 invokevirtual，这是因为这 3 个方法都是可以被子类所重写的，所以编译器在编译期间无法确定其真实的调用方到底是谁，只能通过 invokevirtual 指令在运行期进行动态绑定。

8.7.5 HSDB 查看运行时 vtable

在 HotSpot 中，Java 类在 JVM 内部所对应的类型是 instanceKlassOop（JDK 8 中的类型名是 instanceKlass），vtable 便分配在 instanceKlass 对象实例的内存末尾。instanceKlass 对象实例在内存中所占内存大小是 0x1b8 字节，换算成十进制是 440。根据这个特点，可以使用 HSDB 获取到 Java 类所对应的 instanceKlass 在内存中的首地址，然后加上 0x1b8，就得到 vtable 的内存地址，如此便可以查看这个内存位置上的 vtable 成员数据。

还是以上一节中的 class A 作为示例程序，类 A 中仅包含 1 个 Java 方法，因此类 A 的 vtable 长度一共是 6，另外 5 个是超类 java.lang.Object 中的 5 个方法。使用 JDB 基于 JDK 8 调试（关闭 JDK 的指针压缩选项），并运行至断点处使程序暂停，然后使用 HSDB 连接上测试程序，打开 HSDB 的 Tools->Class Browser 功能，就能看到类 A 在 JVM 内部所对应的 instanceKlass 对象实例的内存地址，如图 8.13 所示。

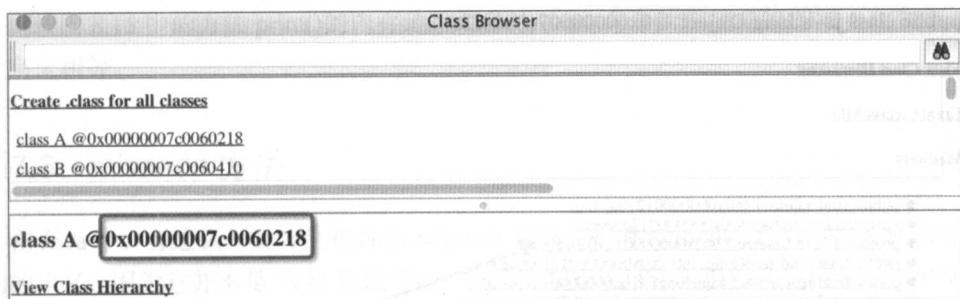


图 8.13 使用 HSDB 查看类 A 的内存地址

由图 8.13 可知,类 A 在 JVM 内部所对应的 instanceKlass 的内存首地址是 0x00000007c0060218,由于 vtable 被分配在 instanceKlass 的末尾位置,因此 vtable 的内存首地址是:

$$0x00000007c0060218 + 0x1b8 = 0x00000007c00603d0$$

这里的 0x1b8 是 instanceKlass 对象实例所占的内存空间大小。得到 vtable 内存地址后,便可以使用 HSDB 的 mem 工具来查看这个地址处的内存数据。单击 HSDB 工具栏上的 Windows->Console 按钮,打开 HSDB 的终端控制台,按回车键,然后输入“mem 0x00000007c00603d0 6”命令,就可以查看从 vtable 内存首地址开始的连续 6 个双字内容,如下所示:

```
hsdb> mem 0x00000007c00603d0 6
0x00000007c00603d0: 0x00000001210a0c10
0x00000007c00603d8: 0x00000001210a06e8
0x00000007c00603e0: 0x00000001210a0840
0x00000007c00603e8: 0x00000001210a0640
0x00000007c00603f0: 0x00000001210a0778
0x00000007c00603f8: 0x00000001214a05d8
```

在 64 位平台上,一个指针占 8 字节,而 vtable 里的每一个成员元素都是一个指针,因此这里 mem 所输出的 6 行,正好是类 A 的 vtable 里的 6 个方法指针,每一个指针指向 1 个方法在内存中的位置。类 A 的 vtable 总长度是 6,其中前面 5 个是基类 java.lang.Object 中的 5 个方法的指针。在 HSDB 中可以查看 Java 方法在 JVM 内部所对应的 method 对象实例的内存地址,单击 HSDB 工具栏上的 Tools->Class Browser 按钮,选择某个类,HSDB 会显示这个类中的所有方法及其内存地址。图 8.14 显示了使用 HSDB 查看 java.lang.Object 中的方法的内存地址。

```
public class java.lang.Object @0x00000007c0000f28

View Class Hierarchy

Create .class File

Methods



- public void <init>() @0x00000001210a0480;
- static void <clinit>() @0x00000001210a0ca8;
- protected void finalize() @0x00000001210a0c10;
- public final void wait(long, int) @0x00000001210a0ae0;
- public final native void wait(long) @0x00000001210a0a00;
- public final void wait() @0x00000001210a0b78;
- public boolean equals(java.lang.Object) @0x00000001210a06e8;
- public java.lang.String toString() @0x00000001210a0840;
- public native int hashCode() @0x00000001210a0640;
- public final native java.lang.Class getClass() [signature OLjava.lang.Class;<*>] @0x00000001210a05a8;
- protected native java.lang.Object clone() @0x00000001210a0778;
- private static native void registerNatives() @0x00000001210a0508;
- public final native void notify() @0x00000001210a08c8;
- public final native void notifyAll() @0x00000001210a0960;

```

图 8.14 使用 HSDB 查看 java.lang.Object 中的 5 个动态方法的内存地址

图 8.14 标识出了 java.lang.Object 中 5 个可被继承的 Java 方法的内存地址。既然类 A 的 vtable 的 5 个成员指针指向 java.lang.Object 中的这 5 个方法，则 vtable 中的前 5 个指针的值必定就是 java.lang.Object 类中这 5 个方法的内存地址。比较上面 mem 命令的输出结果与该图中的这 5 个地址的值，果然发现 mem 命令所输出结果的前 5 行与 java.lang.Object 中的这 5 个方法的内存地址是一一对应的。这一方面证明 vtable 数据的确是被分配在 instanceKlass 对象实例的内存区域的后面，另一方面也说明 vtable 中所存储的的确是指向方法内存的指针。

上面 mem 命令所输出的第 6 行的指针，一定就是指向类 A 自己的方法的内存地址。使用 HSDB 查看类 A 的方法的内存地址，如图 8.15 所示。

```
class A @0x00000007c0060218

View Class Hierarchy

Create .class File

Super Class
public class java.lang.Object @0x00000007c0000f28

Methods



- void <init>() @0x00000001214a0530;
- public void print() @0x00000001214a05d8;

```

图 8.15 使用 HSDB 查看类 A 的 print()方法的内存地址

比较图 8.15 中的方法 print() 的内存地址与上述 mem 命令所输出结果的第 6 行数据，会发现两者完全相等。

8.7.6 miranda 方法

在 Java 中，有这么一类方法被称为 miranda 方法，所谓 miranda 方法，在 JVM 内部并没有专门的定义，因为它并不是 JVM 规范里的一部分。根据 HotSpot 里的文档描述，在早期的虚拟机里有一个 bug，那就是 JVM 在遍历解析 Java 类的方法时，仅仅会遍历 Java 类及其所有父类的方法，但是并不会去查找 Java 类所实现的接口 interface 里的方法，这会导致这样一种结果：如果 Java 类没有实现接口里的方法，则接口里的方法将不会被虚拟机查到。为了解决这个问题，编译器引入了一个相反的办法，那就是在编译期往 Java 类中插入接口里所定义的方法，这些方法就是所谓的 miranda 方法。但是这种解决办法也是有问题的，因为 miranda 方法并不是 Java 规范的一部分，所以从某种意义上说，这其实是另一种 bug。

按照上面所描述的，首先有一个疑问：JVM 规定，对于接口里所定义的接口方法，Java 类必须全部实现这些接口类，那么哪里还会出现什么 bug 呢？但是有一种情况允许 Java 类可以不实现接口方法，只需要在类名前面加上 abstract 修饰符，如下例所示。

先定义一个接口：

清单：/IA.java

作用：miranda 方法

```
public interface IA {
    void test();
}
```

接着定义一个 abstract 的类：

清单：/MyClass.java

作用：miranda 方法

```
public abstract class MyClass implements IA{
    public MyClass(){
        test();
    }
}
```

MyClass 类继承了 IA 接口，但是并没有实现接口方法 test()，这种情况也是能够编译通过的。但是在 MyClass 的构造函数里，可以调用 test() 这个接口方法。如果编译器没有 miranda 机制，则在 MyClass 类的构造函数里调用 test() 接口方法时，肯定会编译报错，因为这个方法只存在于接口类中，而不存在于 MyClass 的任何父类中。

事实上，miranda 方法中的“Miranda”一词是有典故的，这个典故来源于法庭宣判。miranda 其实是一个法律术语，即“米兰达原则”，简单来说，米兰达原则（Miranda Rule）要求警察告诉被拘捕的犯罪嫌疑人，他可以对警察保持缄默，他有权要求有律师在场。

关于这个典故各位道友可以自行在网络上搜索，本书不多讲了。总之，JVM 借鉴了法律上的这一充满人道主义关怀的原则，将其运用于接口类的方法实现中。如果一个 Java 类无法提供接口类方法的实现，那么编译器将会为其提供一个方法实现，这个方法就叫作 miranda 方法。这如同法律审判一样，如果一个犯罪嫌疑人没有能力请一名律师辩护，那么法庭将为其提供一个。从这个角度来重新审视程序，会发现程序设计原来和生活法则都是相通的，创意来源于生活，设计来源于生活，一切都离不开生活。

软件程序使用符号作为程序世界的规则，但是依然存在不遵循规则的现象，在程序的世界里，当出现不遵循规则的现象时，设计者总是倾向于来弥补缺陷，使程序的世界尽量完美，少一些“悲剧”，这何尝不是每一个人所梦想并为之奋斗的社会愿景？

在 Java 类中调用 miranda 方法时，实现的是动态绑定策略，而非早绑定。还是以上面的 MyClass 类为例，在其构造函数中调用 test()方法时，生成的汇编指令是 invokevirtual，如下：

```
public MyClass();
Code:
Stack=1, Locals=1, Args_size=1
0:   aload_0
1:   invokespecial #1; //Method java/lang/Object."<init>":()V
4:   aload_0
5:   invokevirtual #2; //Method test1:()V
8:   return
```

既然 miranda 方法被实现为“晚绑定”机制，那么按照上面所讲的 vtable 的原理，miranda 方法将会被加入到 vtable 中，而 HotSpot 内部的确也是这么处理的。在 classFileParser.cpp::parseClassFile() 函数中调用 klassVtable.cpp::compute_vtable_size_and_num_mirandas() 函数计算 vtable 长度时，其实已经包含了对 miranda 方法的处理，如下：

清单：/src/share/vm/oops/klassVtable.cpp

作用：在 compute_vtable_size_and_num_mirandas() 函数中处理 miranda 方法

```
void klassVtable::compute_vtable_size_and_num_mirandas(int &vtable_length,
                                                       int &num_miranda_methods,
                                                       klassOop super,
                                                       objArrayOop methods,
                                                       AccessFlags class_flags,
                                                       Handle classloader,
                                                       Symbol* classname,
                                                       objArrayOop local_interfaces,
```

```

        TRAPS
    }

//...

// compute the number of mirandas methods that must be added to the end
num_miranda_methods = get_num_mirandas(super, methods, local_interfaces);
vtable_length += (num_miranda_methods * vtableEntry::size());

//...
}

```

在这个逻辑中，HotSpot 先计算出当前类的所有 miranda 方法的总数，并将其增加到 vtable 的长度变量里。而 miranda 方法的总数的计算逻辑就是搜索当前类所实现的所有接口类，以及各个接口所继承的父类接口类中的方法（别忘了接口类是可以继承接口类的），如果这些接口类方法并没有在当前类中实现，则会被当作 miranda 方法。各位道友可以自行编写一个用 abstract 修饰的 Java 类，使其实现某个接口类，同时不实现接口方法，然后借助于 HSDB 来观察 Java 类在 JVM 内部所对应的 instanceKlassOop 对象实例的 _vtable_len 大小，从而验证 HotSpot 上面这段逻辑。

在 klassVtable.cpp::compute_vtable_size_and_num_mirandas() 函数中调用 klassVtable::get_num_mirandas() 函数，而后者调用 klassVtable::get_mirandas() 函数完成所有 miranda 方法的搜索和判断。在该函数中分别对当前 Java 类的全部接口类进行遍历，而在遍历每一个接口类时，又对该接口类所继承的父类接口类进行遍历，如此一层一层往上搜索全部 miranda 方法：

清单：/src/share/vm/oops/klassVtable.cpp

作用：get_mirandas()方法

```

void klassVtable::get_mirandas(GrowableArray<methodOop>* mirandas,
                                klassOop super, objArrayOop class_methods,
                                objArrayOop local_interfaces) {
    assert((mirandas->length() == 0), "current mirandas must be 0");

    // 遍历当前 Java 类所实现的全部接口类
    int num_local_ifs = local_interfaces->length();
    for (int i = 0; i < num_local_ifs; i++) {
        instanceKlass *ik = instanceKlass::cast(klassOop(local_interfaces->obj_at(i)));
        add_new_mirandas_to_list(mirandas, ik->methods(), class_methods, super);

        // 遍历每一个接口类所继承的父接口类
        objArrayOop super_ifs = ik->transitive_interfaces();
        int num_super_ifs = super_ifs->length();
        for (int j = 0; j < num_super_ifs; j++) {
            instanceKlass *sik =

```

```
instanceKlass::cast(klassOop(super_ifs->obj_at(j)));
    add_new_mirandas_to_list(mirandas, sik->methods(), class_methods, super);
}
}
```

miranda 方法其实并不是 JVM 的一个标准规范，但是笔者本人实在是被这个机制所感动，冰冷的机器里面也是充满情义的，因此忍不住将其介绍给大家。

对于接口方法，Java 类中还会保存一种内部结构——itable，顾名思义，就是接口方法表。itable 主要用于接口类方法的分发，其机制与 vtable 类似，都会在运行期进行方法的动态绑定。各位有兴趣的道友可以自行研究。

8.7.7 vtable 特点总结

前文对 vtable 进行了比较全面的研究和验证，这里再次总结下其特点：

- ◎ vtable 分配在 instanceKlassOop 对象实例的内存末尾。
 - ◎ 所谓 vtable，可以看作是一个数组，数组中的每一项成员元素都是一个指针，指针指向 Java 方法在 JVM 内部所对应的 method 实例对象的内存首地址。
 - ◎ vtable 是 Java 实现面向对象的多态性的机制，如果一个 Java 方法可以被继承和重写，则最终通过 invokevirtual 字节码指令完成 Java 方法的动态绑定和分发。事实上，很多面向对象的语言都基于 vtable 机制去实现多态性，例如 C++。
 - ◎ Java 子类会继承父类的 vtable。
 - ◎ Java 中所有类都继承自 java.lang.Object，java.lang.Object 中有 5 个虚方法（可被继承和重写）：

```
void finalize()  
boolean equals(Object)  
String toString()  
int hashCode()  
Object clone()
```

因此，如果一个 Java 类中不声明任何方法，则其 `vtalbe` 的长度默认为 5。

- Java 类中不是每一个 Java 方法的内存地址都会保存到 vtable 表中。只有当 Java 子类中声明的 Java 方法是 public 或者 protected 的，且没有 final、static 修饰，并且 Java 子类中的方法并非对父类方法的重写时，JVM 才会在 vtable 表中为该方法分配一个槽位。

法增加一个引用。

- ◎ 如果 Java 子类某个方法重写了父类方法，则子类的 vtable 中原本对父类方法的指针引用会被替换为对子类的方法引用。

8.7.8 vtable 机制逻辑验证

上文一直在讲 Java 中实现多态是通过 vtable 这个机制，但是 vtable 到底是如何实现多态机制的呢？这在 JVM 内部颇为复杂，这里先对其进行逻辑上的推理，让各位道友可以在不用知道具体源码实现细节的前提下，也能够知道如何“玩转”多态。

其实，多态实现的机制也很简单，就是通过 vtable（貌似这句话白说了哈哈）。文字描述起来够费劲，各位道友理解起来也费劲，不如举个例子：

清单：Animal.java

作用：vtable 机制验证

```
public class Animal {
    public void say(){
        System.out.println("I'm animal");
    }

    public static void main(String[] args){
        Animal animal = new Dog();
        run(animal);

        animal = new Animal();
        run(animal);
    }

    public static void run(Animal animal){
        animal.say();
    }
}

class Dog extends Animal{
    @Override
    public void say(){
        System.out.println("I'm a dog");
    }
}
```

本示例在本节开始讲解 vtable 的时候提到过，不过稍微做了一点修改。在本示例中，有父类 Animal 和子类 Dog，子类 Dog 重写了父类的 say()方法。

在 main() 主函数中，将 animal 引用分别指向 2 个不同的实例，并调用 run(Animal) 方法。运行这个程序，结果也如预测的那样，如下：

```
I'm a dog
I'm animal
```

这样的输出结果充分演绎了面向对象语言所具有的多态特性，而多态得以实现的奥秘，其实就隐藏在 vtable 中。

根据前文所讲的 vtable 的构成原理，类 Animal 的 vtable 的长度应该为 6，除了所继承的 java.lang.Object 中的 5 个虚方法（即可被重写的方法）外，其自身仅包含 1 个虚方法。并且其 vtable 中的第 6 个指针元素指向 say() 方法在 JVM 内部所对应的 method 实例对象的内存地址。同理，子类 Dog 的 vtable 的长度也应该等于 6，因为前文讲过，子类会完全继承父类的 vtable，并且如果子类重写了父类的方法，则 JVM 会将子类 vtable 中原本指向父类方法的指针成员修改成重新指向子类的方法。

类 Animal 和子类 Dog 的 vtable 结构分别如图 8.16 中左图和右图所示。

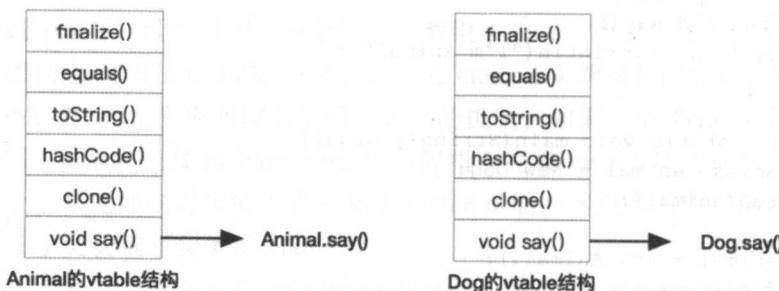


图 8.16 Animal 和 Dog 类的 vtable 结构

在 JVM 运行期，会根据对象引用所执行的实际的实例调用实例的方法。不过，这首先得从编译器说起。上面示例中 Animal.run(Animal) 方法所对应的字节码指令如下（使用 javap 命令显示）：

```
public static void run(Animal);
Code:
Stack=1, Locals=1, Args_size=1
0:   aload_0
1:   invokevirtual    #10; //Method say:()V
4:   return
```

Animal.run(Animal) 方法的字节码指令主要的逻辑包含两步，第一步是 `aload_0`，第二步是 `invokevirtual`。字节码指令 `aload_0` 表示从第 0 个 slot 位置加载 Java 引用对象。由于

Animal.run(Animal)方法是一个 static 静态方法，其入参并没有隐式的 this 指针，所以 slot 中的第一个局部变量就是 Animal.run(Animal)的第一个入参 animal 引用对象。接着第二步执行 invokevirtual 指令，Java 多态的秘密就隐藏在该指令后面所跟的操作数 operand 中。invokevirtual 指令后面的操作数是常量池的索引值，该值为 10，javap 命令显示索引为 10 的常量池所代表的字符串是 Method say:()V，这表示 invokevirtual 在运行期调用的方法是 void say()。在运行期，JVM 将首先确定被调用的方法所属的 Java 类实例对象，JVM 会读取被调用的方法的堆栈，并获取堆栈中的局部变量表的第 0 个 slot 位置的数据，该数据一定是指向被调用的方法所属的 Java 类实例，原因很简单，凡是所对应的字节码指令为 invokevirtual 的 Java 方法，其必定是 Java 类的成员方法，而非静态方法，而 Java 类的成员方法的第一个入参一定是隐式的 this 指针，该指针就指向 Java 类的对象实例。同时，Java 类的第一个入参一定位于局部变量表的第 0 个位置，因此 JVM 可以从被 invokevirtual 所调用的方法的局部变量表中读取到 this 指针，从而得知被调用的 Java 类实例到底是哪一个。

以本示例为例，当 animal 引用变量指向 new Dog() 对象实例时，则 say() 方法的局部变量表第一个入参便指向 new Dog() 这个对象实例。同理，当 animal 引用变量指向 new Animal() 对象实例时，则 say() 方法的局部变量表的第一个入参便指向 new Animal() 实例对象。

JVM 获取到被 invokevirtual 指令所调用的方法所属的实际的类对象时，接着便能够通过对对象获取到其对应的 vtable 方法分发表。vtable 表中保存当前类的每一个方法的指针。JVM 会遍历 vtable 中的每一个指针成员，并根据指针读取到其对应的 method 对象，判断 invokevirtual 指令所调用的方法名称和签名与 vtable 表中指针所指向的方法的名称和签名是否一致，如果方法名称和签名完全一致，则算是找到了 invokevirtual 所实际调用的目标方法，于是 JVM 定位到目标方法的第一条字节码指令并开始执行。如此便完成了方法在运行期的动态分发和执行。

还是以本示例为例说明，当 animal 引用变量指向 new Dog() 对象实例时，JVM 会遍历 Dog 类所对应的 vtable 表，并搜索其中名称为“say”且签名为“void ()”的方法，很显然，Dog 类中存在该方法，于是 JVM 最终执行 Dog 类的 say() 方法。同理，当 animal 引用变量指向 new Animal() 对象实例时，JVM 最终执行的就是 Animal 类中的 say() 方法。

事实上，C++ 的虚方法分发的原理与此完全类似，所不同的是 C++ 的 vtable 由编译器在编译期间完成构建，而 Java 类的 vtable 则由 JVM 在运行期进行构建。

8.8 本章总结

本章主要描述了 Java 方法解析的技术实现。相比于前面章节所讲解的 Java 类字段的解析，

Java 类方法解析明显要复杂得多。这种复杂性体现在 Java 方法属性本身拥有众多信息，尤其是字节码指令部分。除了字节码指令，还有 LVT、miranda 方法等，存储格式比较复杂，并且概念理解起来也并不是一件轻松的事情。JVM 为 Java 方法在内存中所构建的对等体也明显更加复杂。

同时，Java 方法的解析还承担了一部分实现面向对象机制的责任，其核心技术便是 vtable。只有真正理解了 vtable 的实现机制，才能真正理解 Java 面向对象与多重继承的原理。