JVM分享

Java Program in Action

——Java程序的编译、加载与执行

v0.1 2010-02-04
v0.2 2010-04-24
v0.3 2010-06-21
v0.4 2010-12-28
v0.5 2011-03-31
v0.6 in progress
-- always in alpha

莫枢（撒迦）
Kris Mok
http://rednaxelafx.iteye.com

关于我...

- 莫枢（撒迦）
- 2009年7月本科毕业自南京大学软件学院
- 同年10月加入淘宝
- 目前在核心系统研发部专用计算组参与JVM优化与工具制作
- 对编程语言的设计、实现非常感兴趣，希望能够多交流
  - Blog: http://rednaxelafx.iteye.com
  - Twitter: @rednaxelafx
  - 新浪微博: @RednaxelaFX
  - 高级语言虚拟机圈子: http://hllvm.group.iteye.com/
- 加入淘宝前常用的语言是C#、Ruby与Java
  - 也稍微玩过JavaScript、F#/OCaml、Scala、Haskell、Python、Scheme、Go等
- 希望能与大家多多交流！ ^_^

分享安排

- 语言处理器的基本结构
- Java语言与平台
- Java源码级编译器（javac）
- Class文件
- 虚拟机与JVM
- HotSpot VM
- HotSpot VM的运行时支持
- HotSpot VM的内存管理
- HotSpot与解释器
- HotSpot与JIT编译器
- HotSpot VM与JSR 292

友情提示：
许多参考资料可以从该演示稿里嵌的链接找到

希望留下的观点...

- An implementation can be very different from the “mental model”
  - You can cheat as long as you don't get caught
- Don’t trust microbenchmarks
  - Profile your typical application

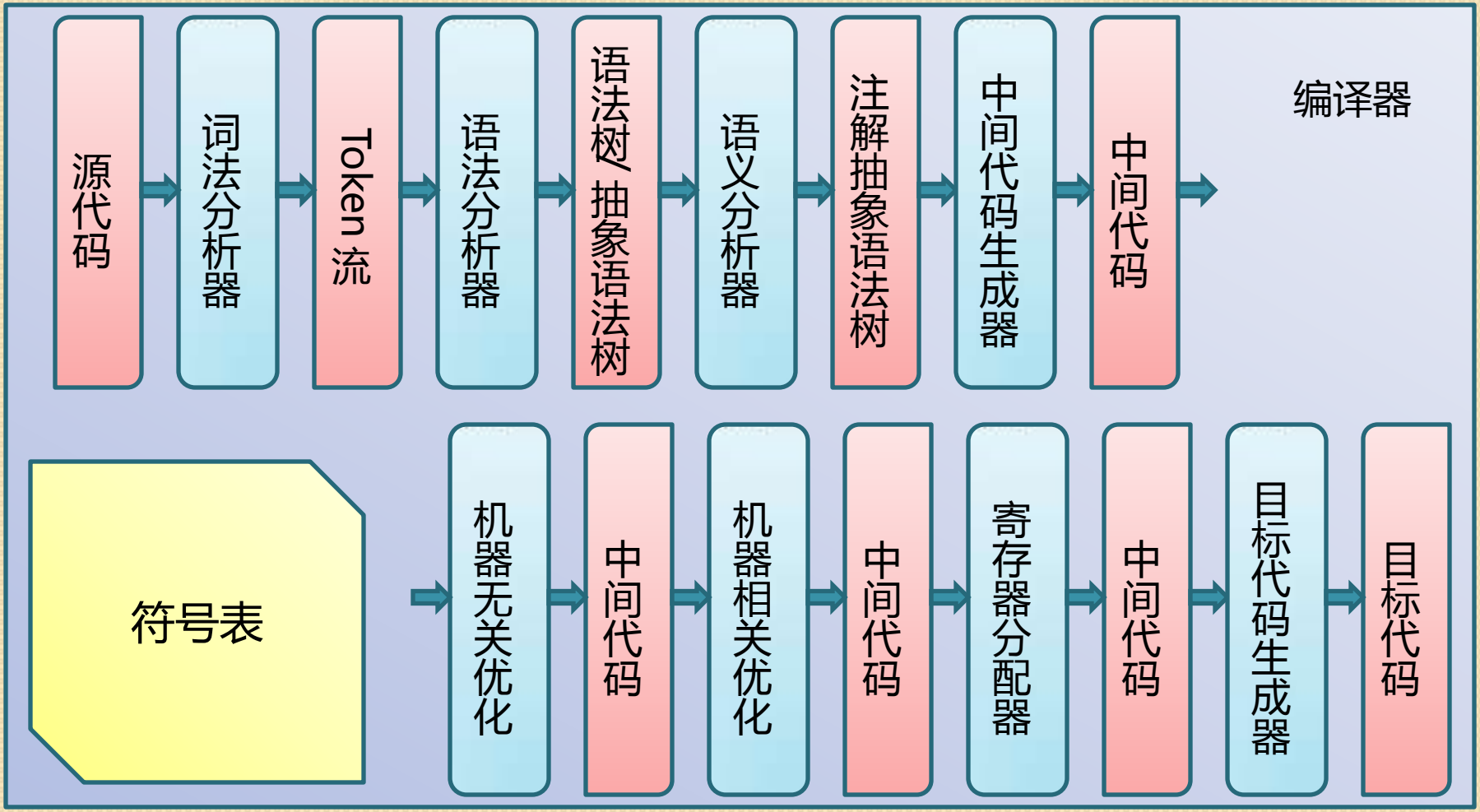
语言处理器的基本结构

语言处理器

- 语言处理器的种类
  - 编译器，如gcc、javac
  - 解释器，如Ruby、Python等的一些实现
  - IDE，如Eclipse、NetBeans等
  - 代码分析器，如FindBugs等
  - 反编译器，如JD、Jad、Reflector.NET等
  - etc ...

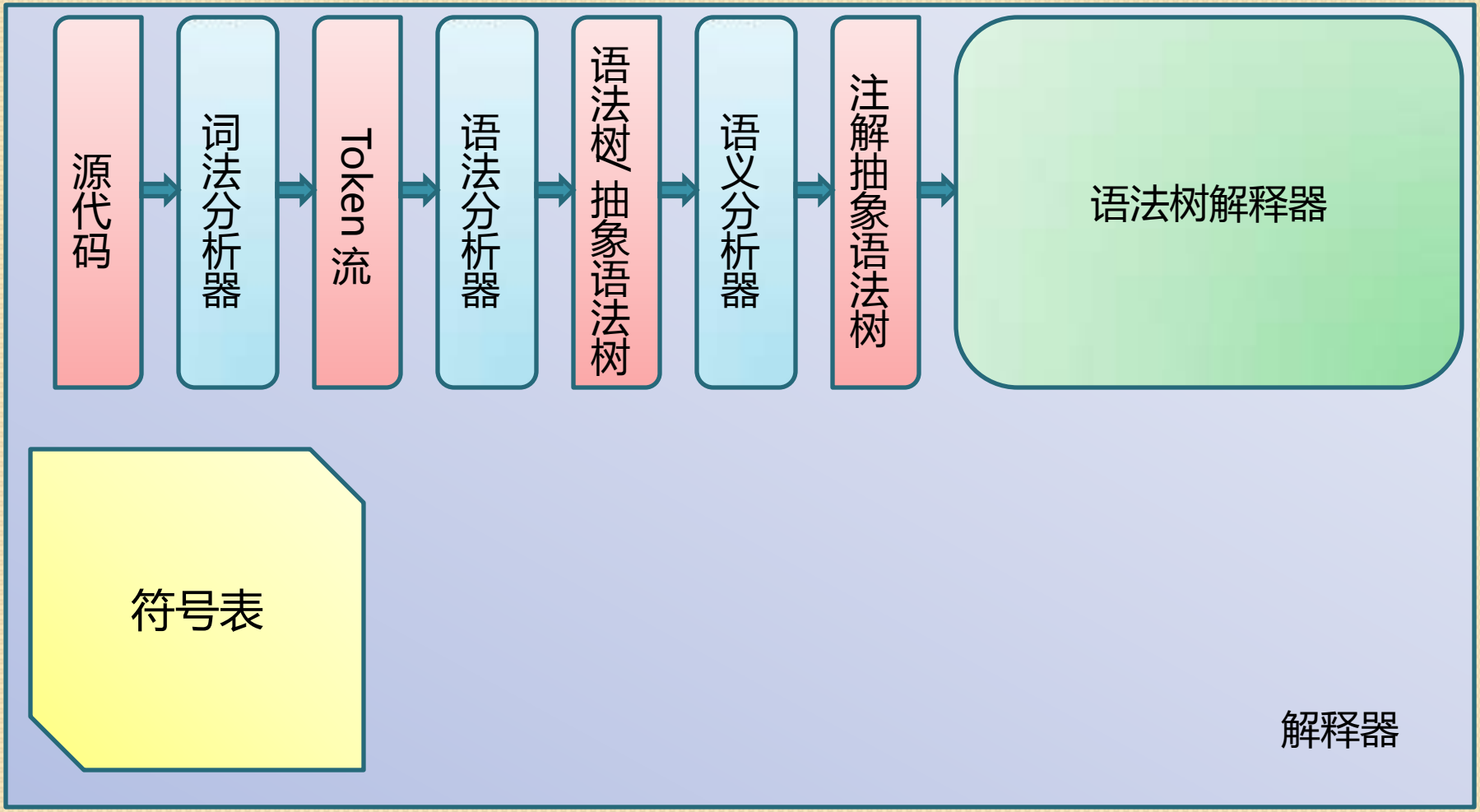
语言处理器的基本结构

语言处理器的重要形式——编译器的基本结构



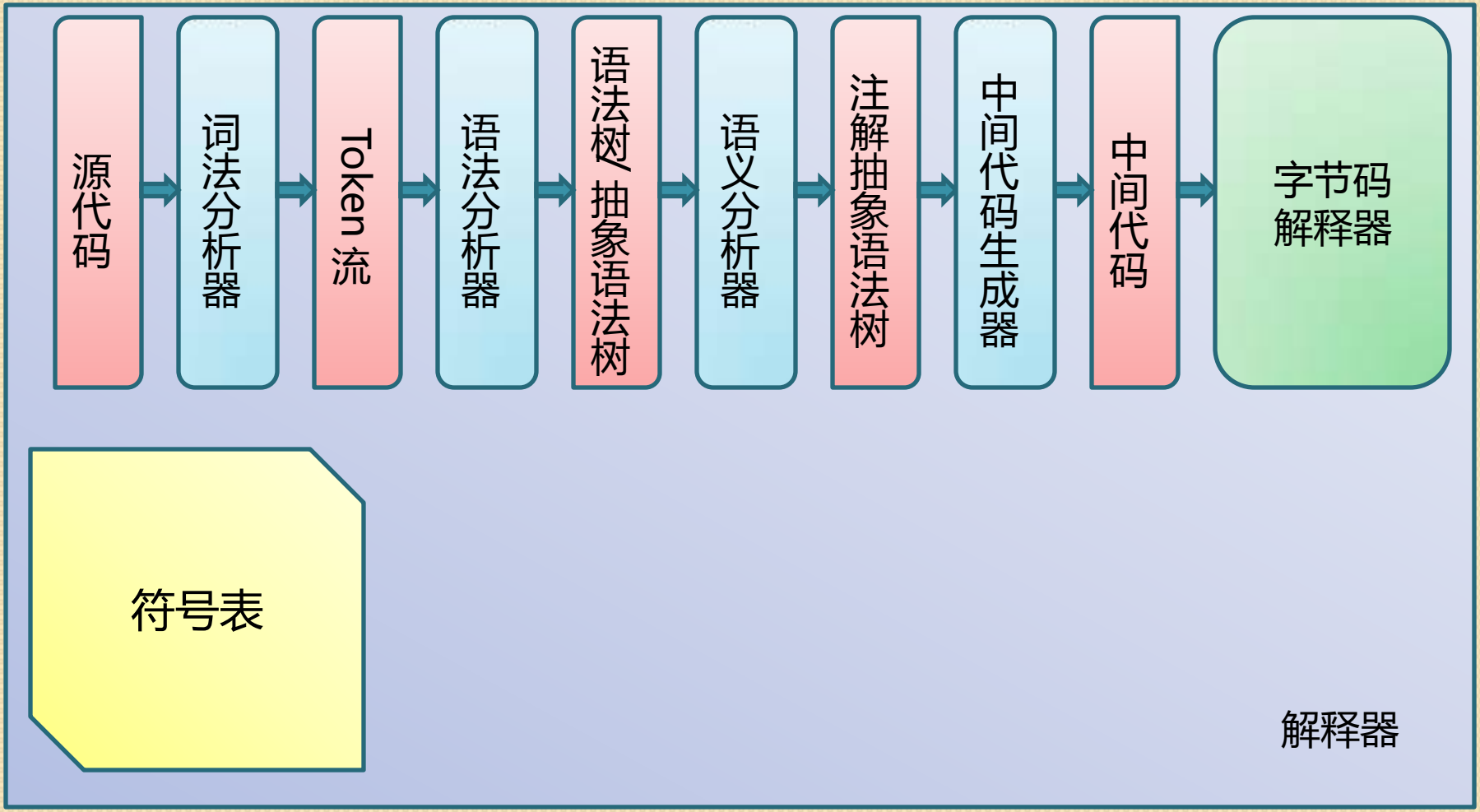
语言处理器的基本结构

语言处理器的主要形式，解释器的基本结构的一种可能



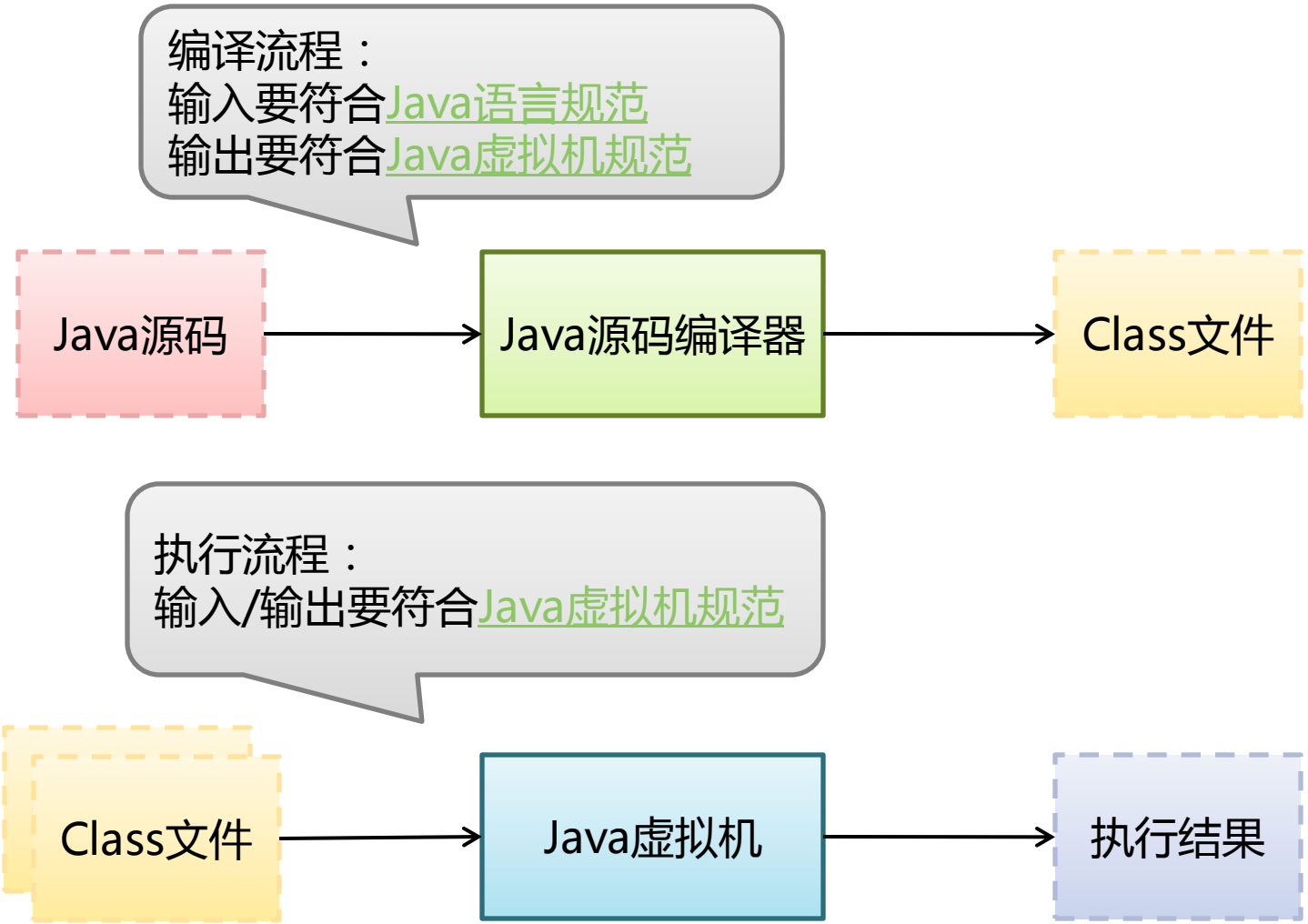
语言处理器的基本结构

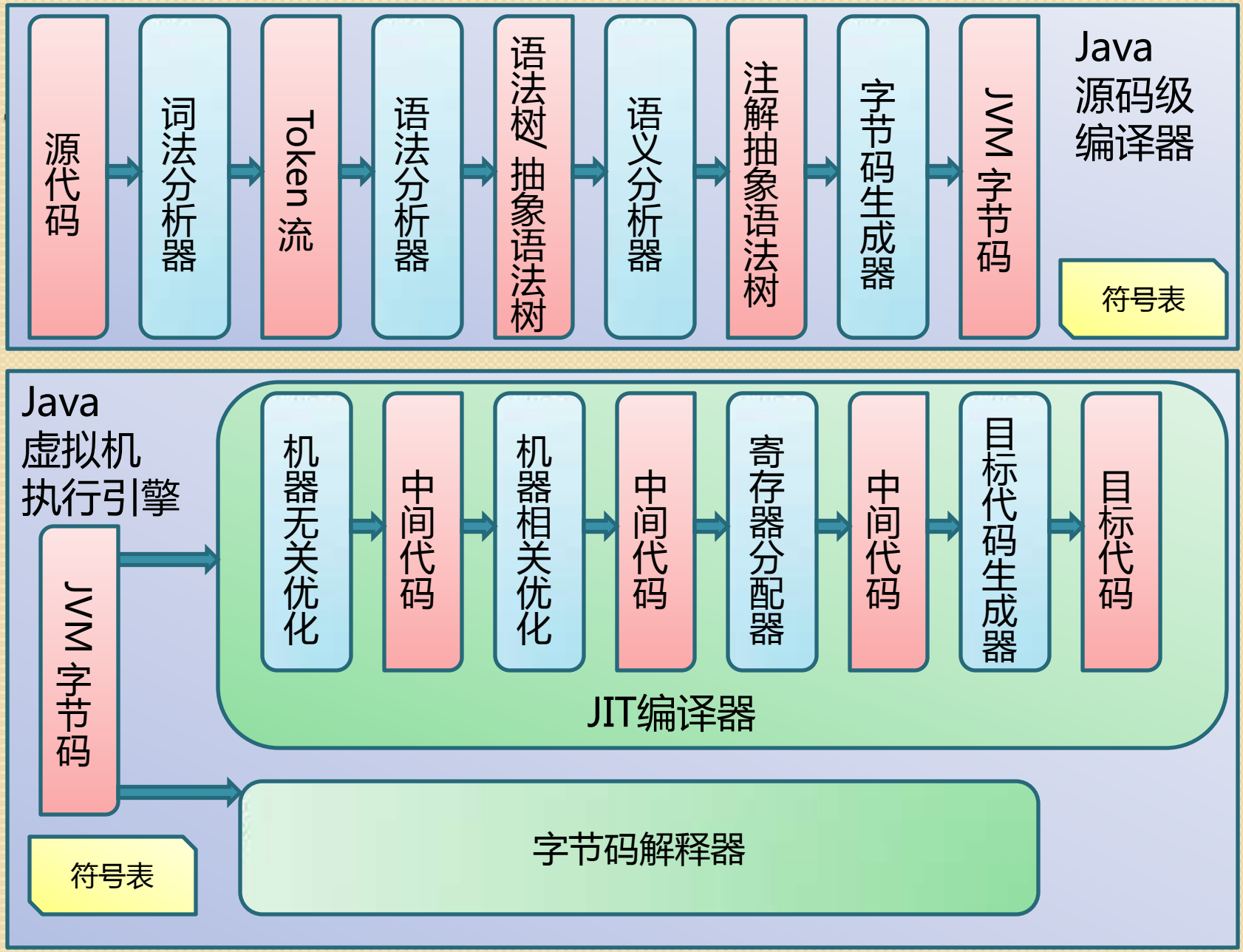
语言处理器的主要形式，解释器的基本结构的一种可能



Sun JDK是如何实现Java语言的？

Java从源码到执行流经何处？




源代码
词法分析器
Token 流
语法分析器
语法树/ 抽象语法树
语义分析器
注解抽象语法树
字节码生成器
JVM 字节码
Java
源码级
编译器
符号表
Java
虚拟机
执行引擎
JVM 字节码
机器无关优化
中间代码
机器相关优化
中间代码
寄存器分配器
中间代码
目标代码生成器
目标代码
JIT编译器
字节码解释器
符号表

JAVA语言与平台

Java语言的基本特征

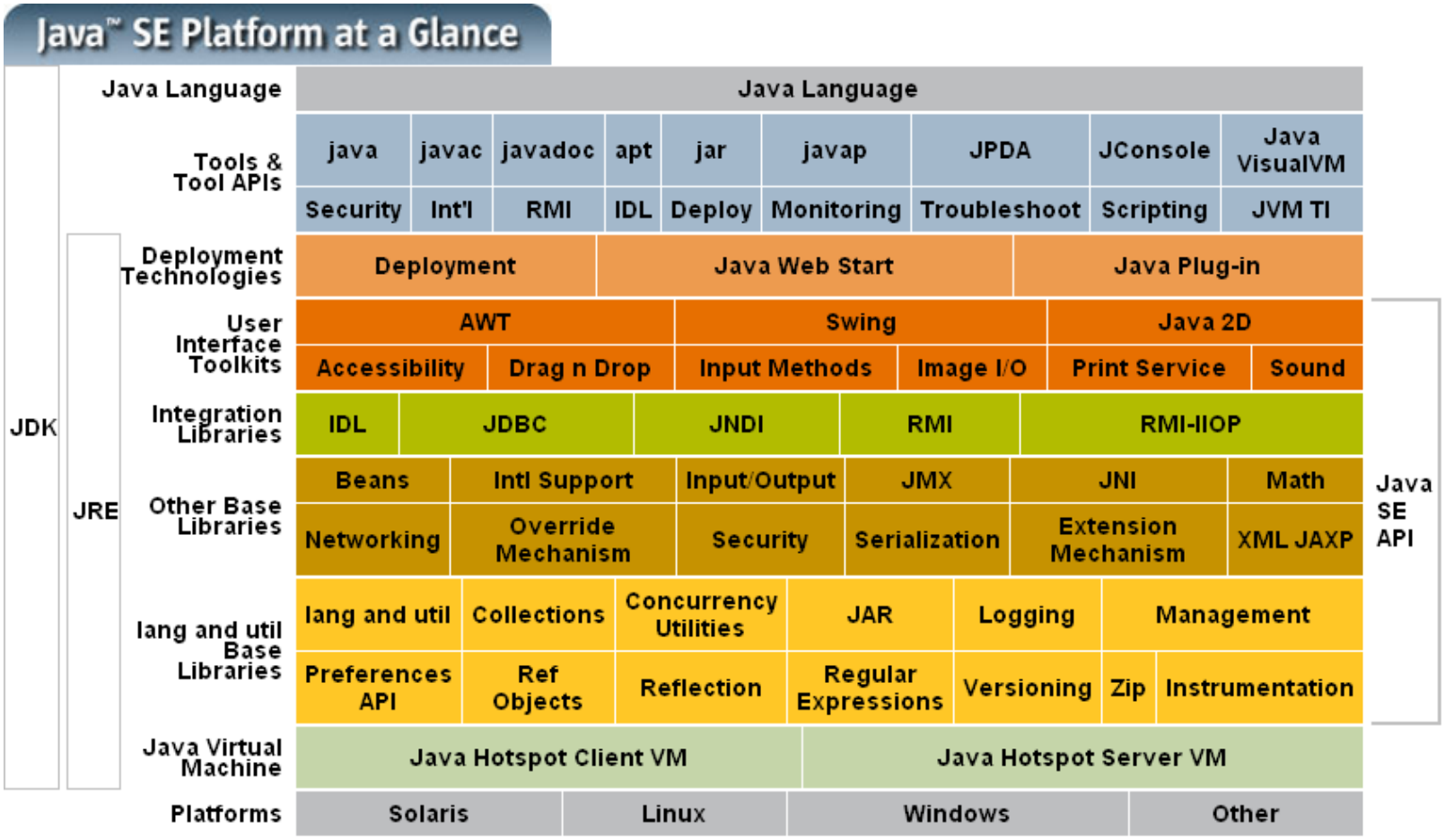
- 类似C++的语法
  - 但是去掉了指针运算
- 带有面向对象特征的静态类型系统
  - 单继承，但可以有多个超类型
  - 运行时一个类型的超类型关系不可变
    - 意味着继承深度不变
- 早绑定与迟绑定的方法分派
- 反射
- 自动内存管理
- 多线程
- 泛型编程
  - JSR 14（Java 5）
- 用户自定义的注解及其处理
  - JSR 175（Java 5）
  - JSR 269（Java 6）

Java语言的类型系统

Java语言是...

- 解释型语言？虚拟机语言？
  - *并非一定要在“JVM”上运行，只要程序能满足Java语言规范中定义的语法和语义即可
  - 从Java源码直接编译到本地代码的编译器
    - 如 GCJ， Excelsior JET等
    - 本次分享接下来的部分将不涉及这类编译器

Java平台

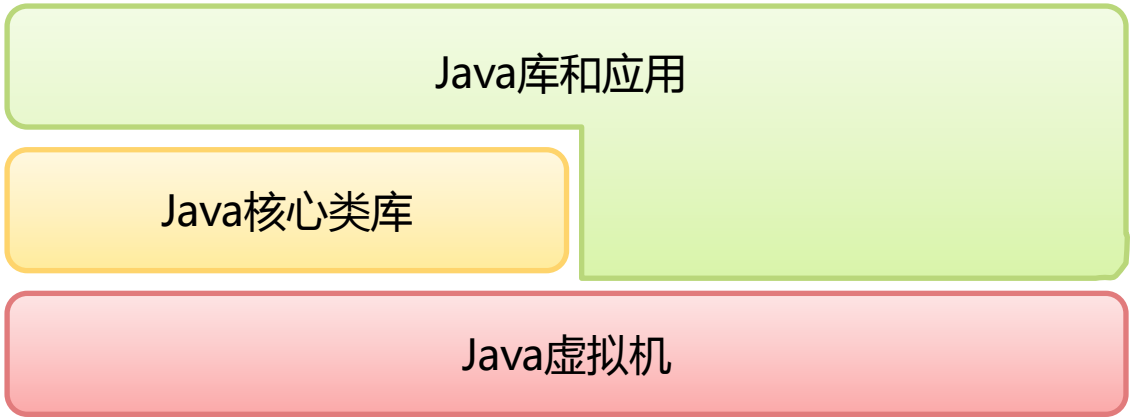

Java™ SE Platform at a Glance
Java Language
Java Language
Tools & Tool APIs
java
javac
javadoc
apt
jar
javap
JPDA
JConsole
Java VisualVM
Security
Int'l
RMI
IDL
Deploy
Monitoring
Troubleshoot
Scripting
JVM TI
Deployment Technologies
Deployment
Java Web Start
Java Plug-in
User Interface Toolkits
AWT
Swing
Java 2D
Accessibility
Drag n Drop
Input Methods
Image I/O
Print Service
Sound
Integration Libraries
IDL
JDBC
JNDI
RMI
RMI-IIOP
Other Base Libraries
Beans
Intl Support
Input/Output
JMX
JNI
Math
Networking
Override Mechanism
Security
Serialization
Extension Mechanism
XML JAXP
lang and util Base Libraries
lang and util
Collections
Concurrency Utilities
JAR
Logging
Management
Preferences API
Ref Objects
Reflection
Regular Expressions
Versioning
Zip
Instrumentation
Java Virtual Machine
Java Hotspot Client VM
Java Hotspot Server VM
Platforms
Solaris
Linux
Windows
Other
JDK
JRE
Java SE API

图片来源

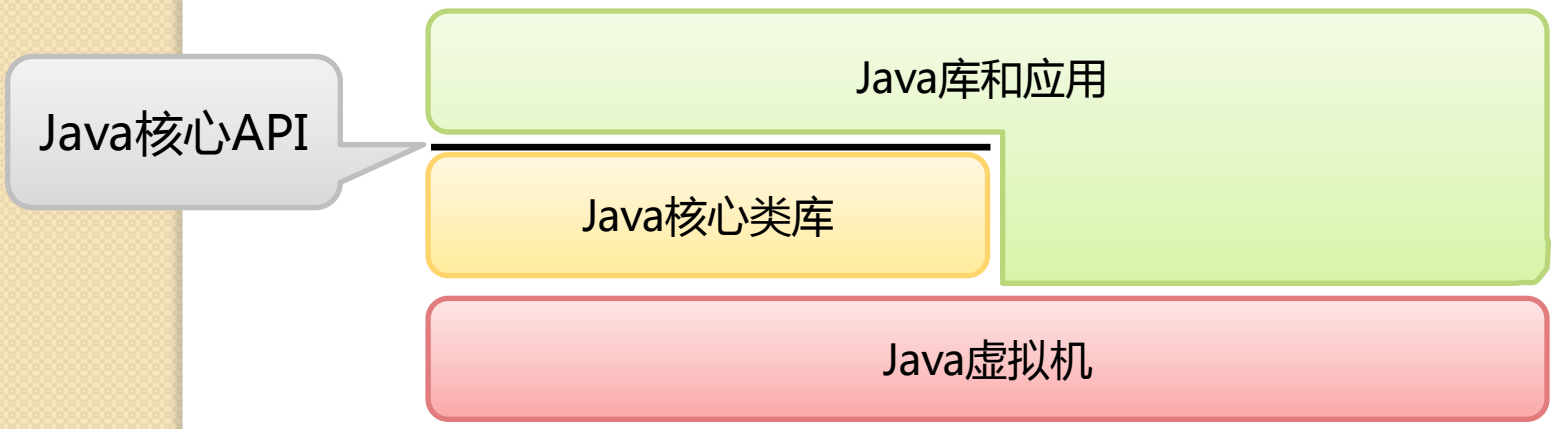
Java平台

- JVM与JRE、JDK的关系
  - JVM：Java Virtual Machine
    - 负责执行符合规范的Class文件
  - JRE：Java Runtime Environment
    - 包含JVM与类库
  - JDK：Java Development Kit
    - 包含JRE与一些开发工具，如javac

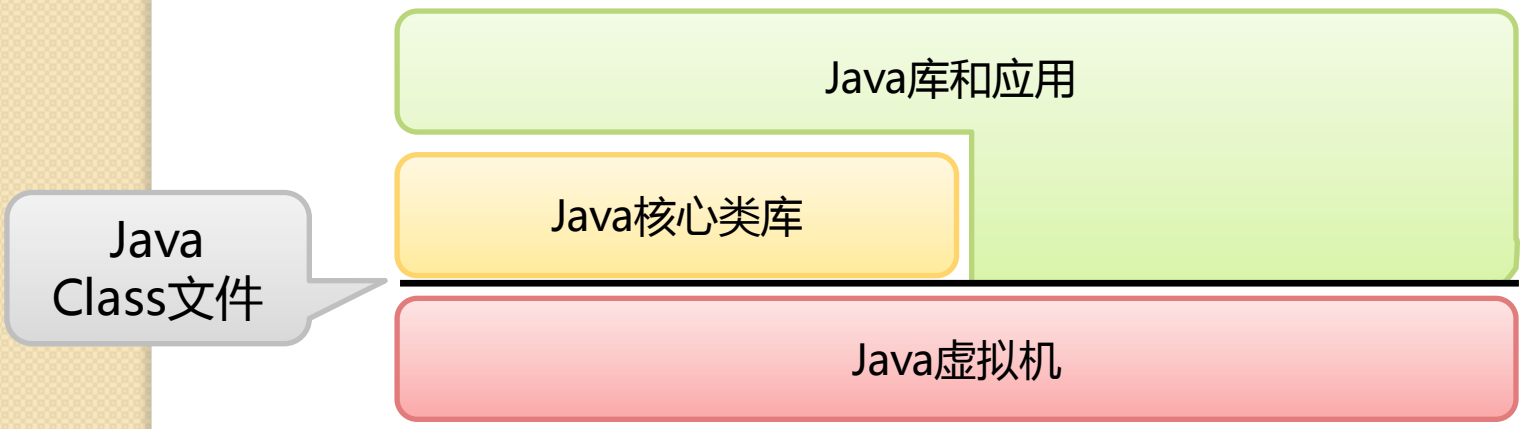
Java平台



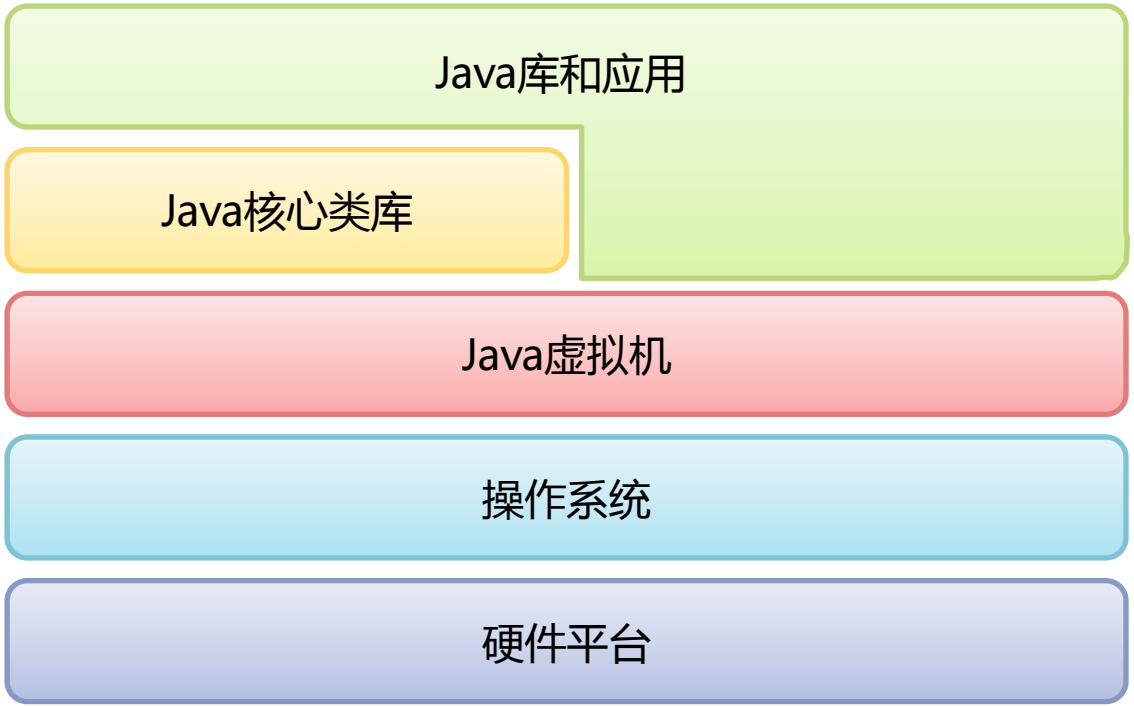
Java平台



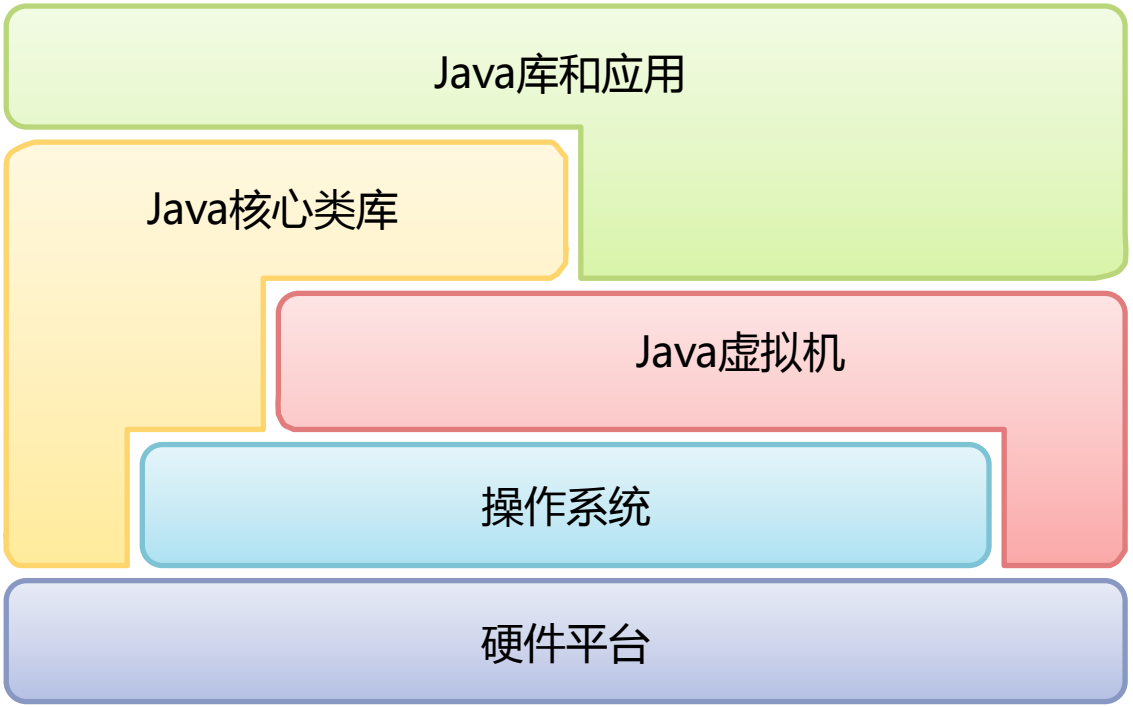
Java平台



Java平台

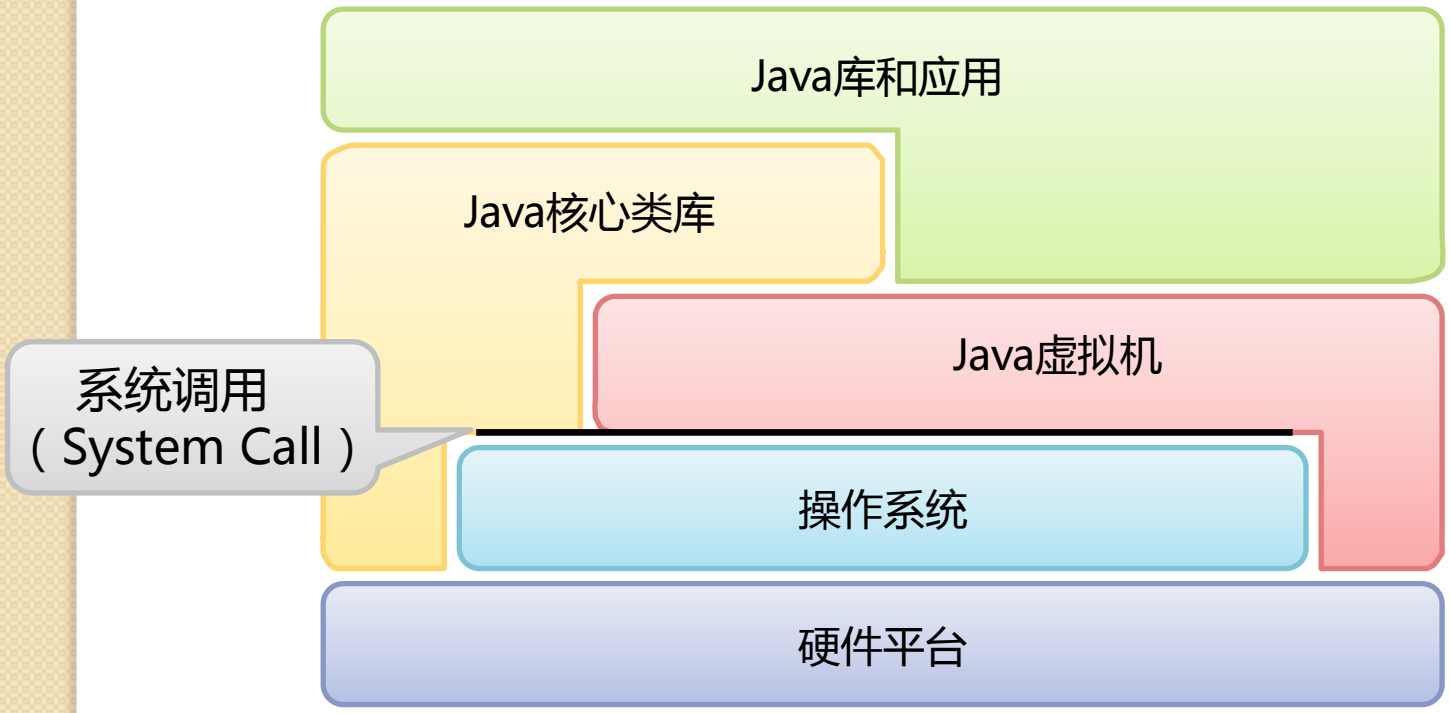


Java平台

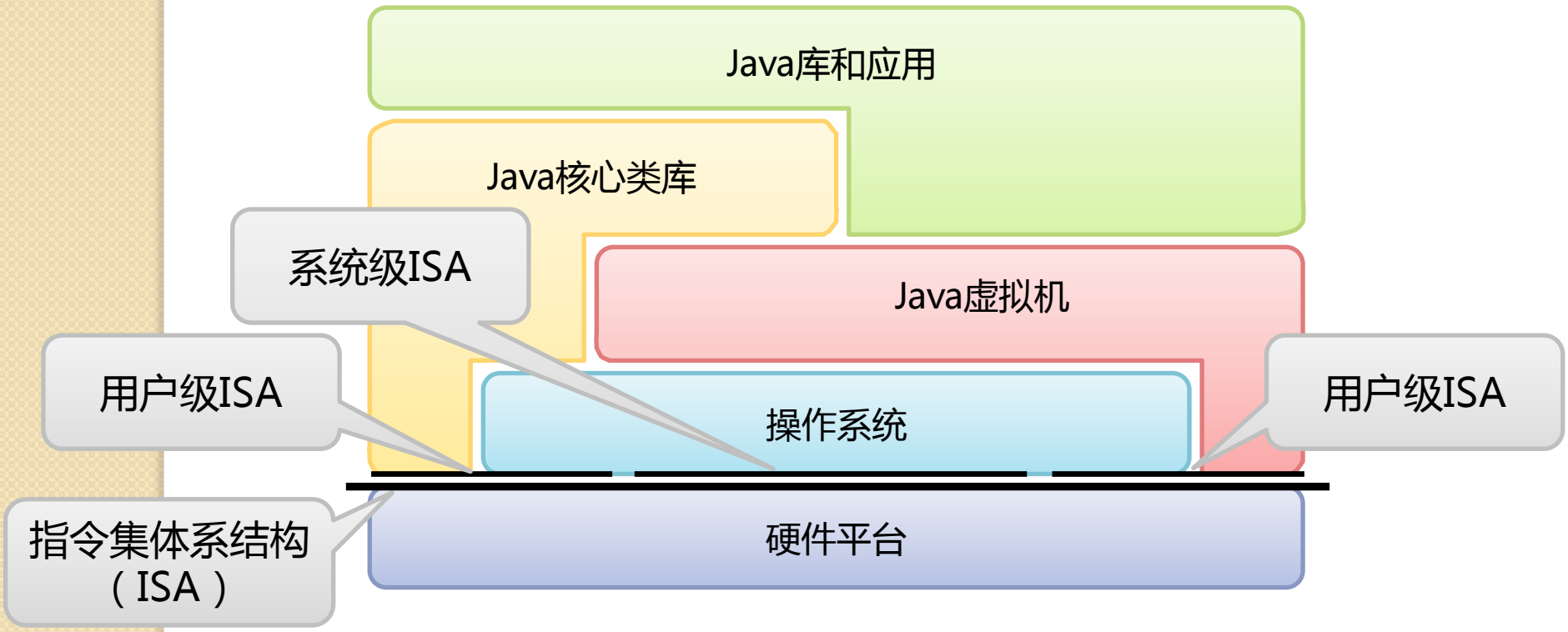


（图片参考自《虚拟机：系统与进程的通用平台》图1.4）

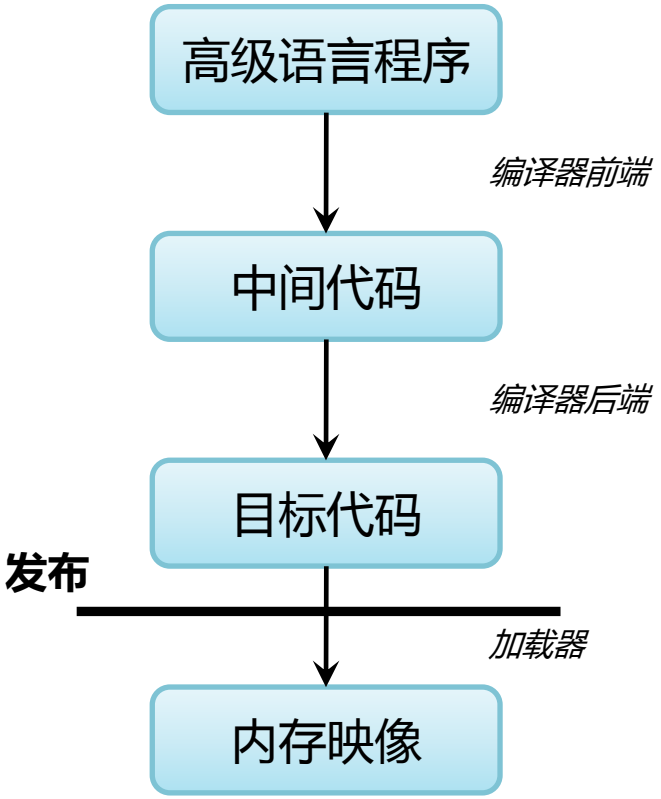
Java平台



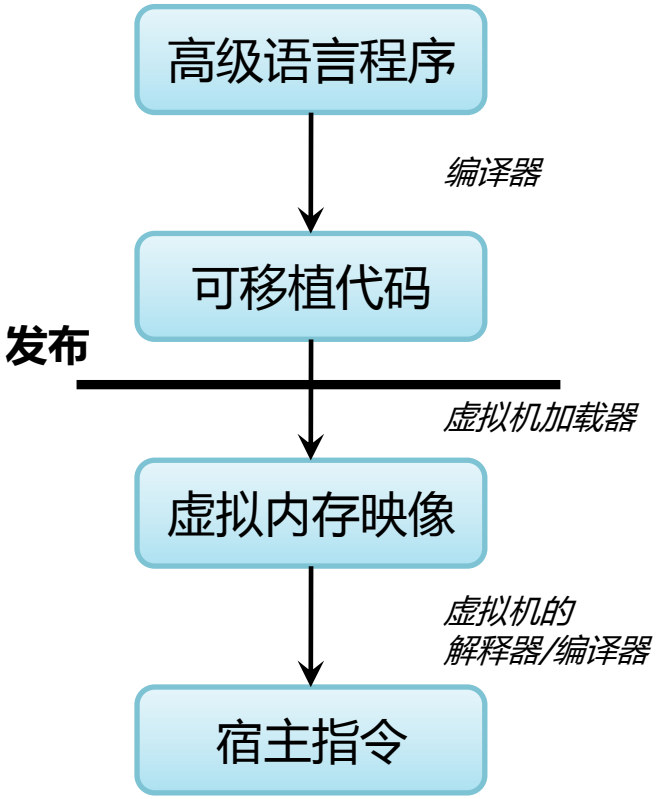
Java平台


Java库和应用
Java核心类库
系统级ISA
Java虚拟机
用户级ISA
操作系统
用户级ISA
指令集体系结构
（ISA）
硬件平台

高级语言程序的处理环境



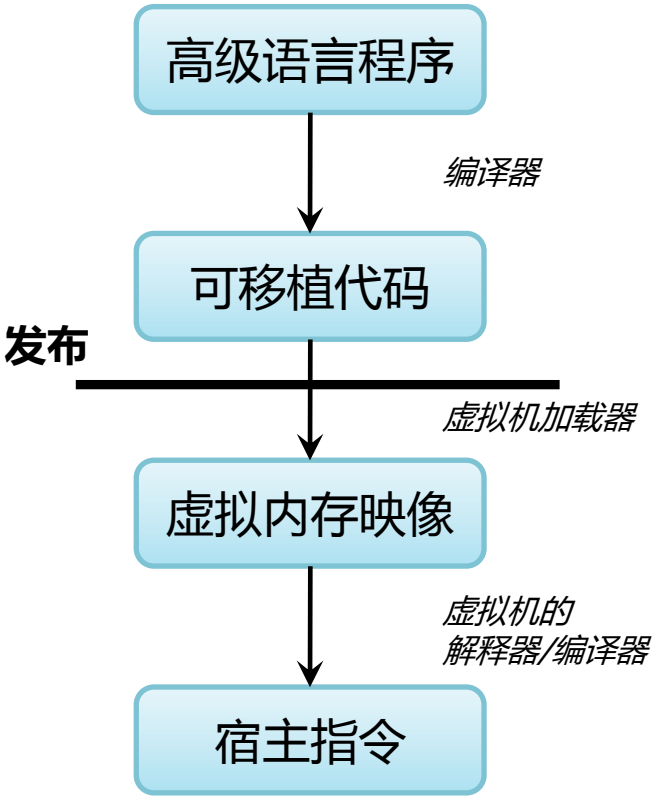
(a) 传统的高级语言实现方式，
发布的是平台相关的目标代码，
装载后可以直接在机器上执行

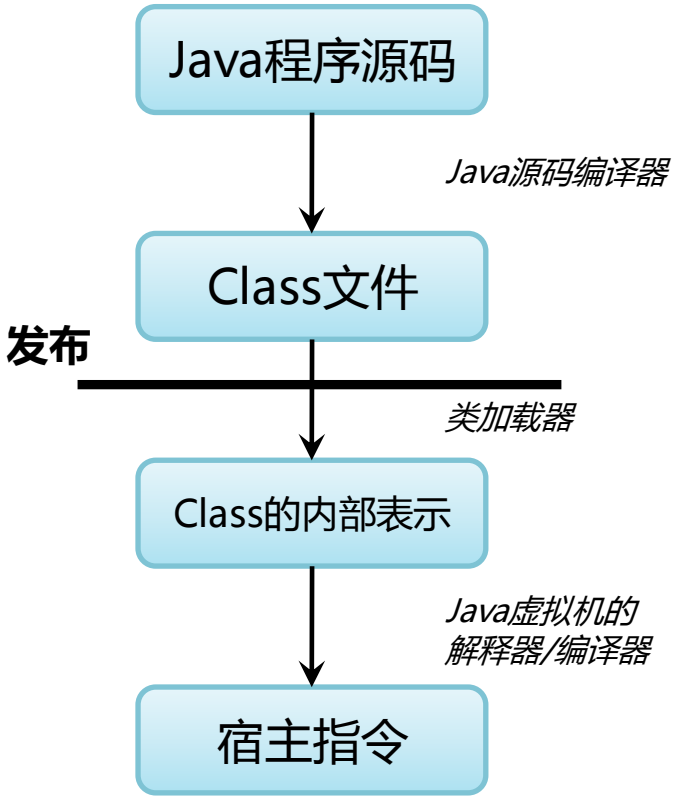


(b) 高级语言虚拟机环境，
发布的是平台无关的可移植代码，
由平台相关的虚拟机实现来“执行”

（图片参考自《虚拟机：系统与进程的通用平台》图1.10）

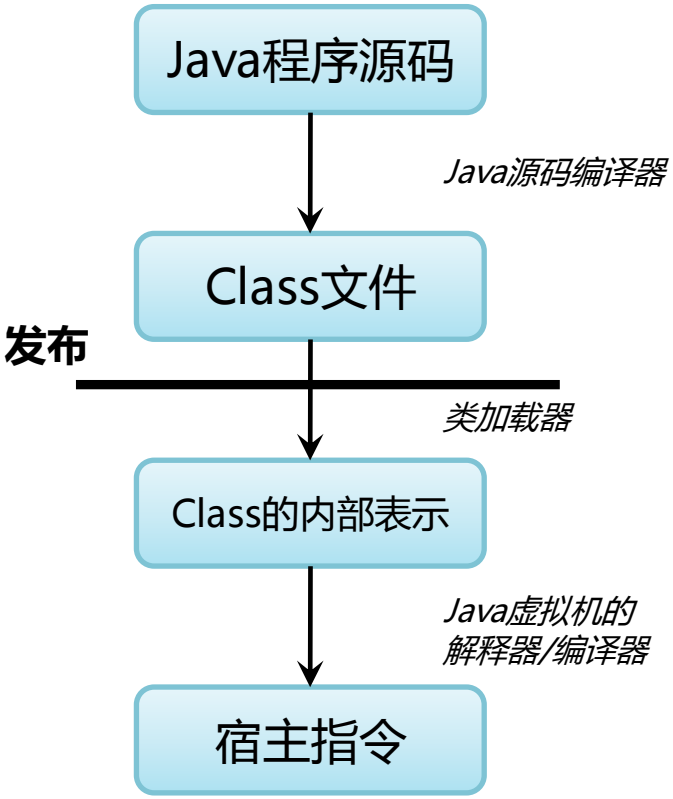
——具体到Java的情况





Java的处理环境在实际
硬件上的可能处理方式

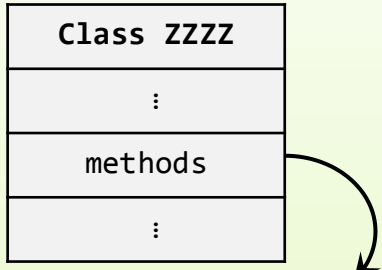
可能的实现方式



Java的处理环境在实际
硬件上的可能处理方式

```
public class ZZZZ {
  private int value;
  public void foo() {
    int v = this.value;
  }
}
```

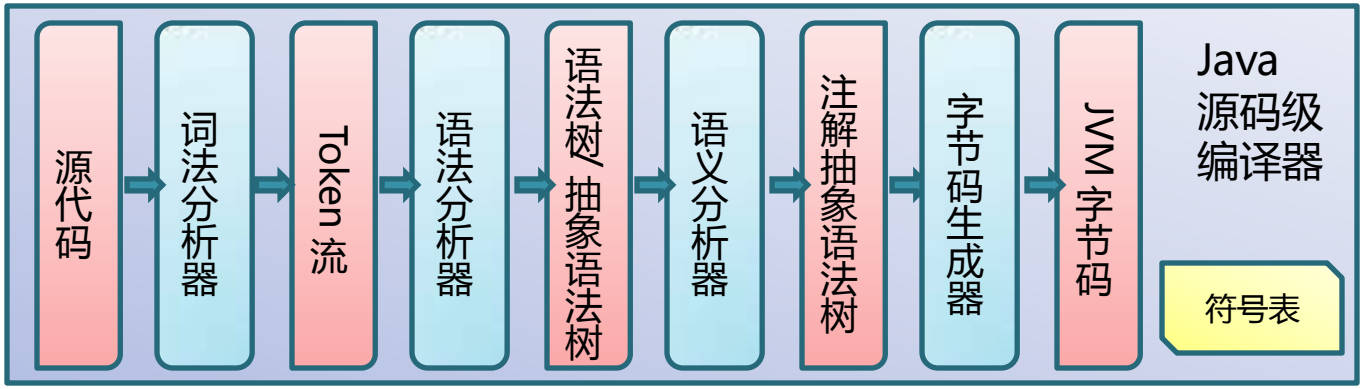
```
…
2a      /* aload_0                          */
b40002  /* getfield #2; //Field value:I */
3c      /* istore_1                         */
…
```




```
Method foo()V
…
dcd00100 /* fast_iaccess_0 #1 */
3c       /* istore_1          */
…
```

```
…
8b4108 ; mov eax,dword ptr ds:[ecx+8]
…
```

JAVA源码级编译器（JAVAC）



Java源码级编译器

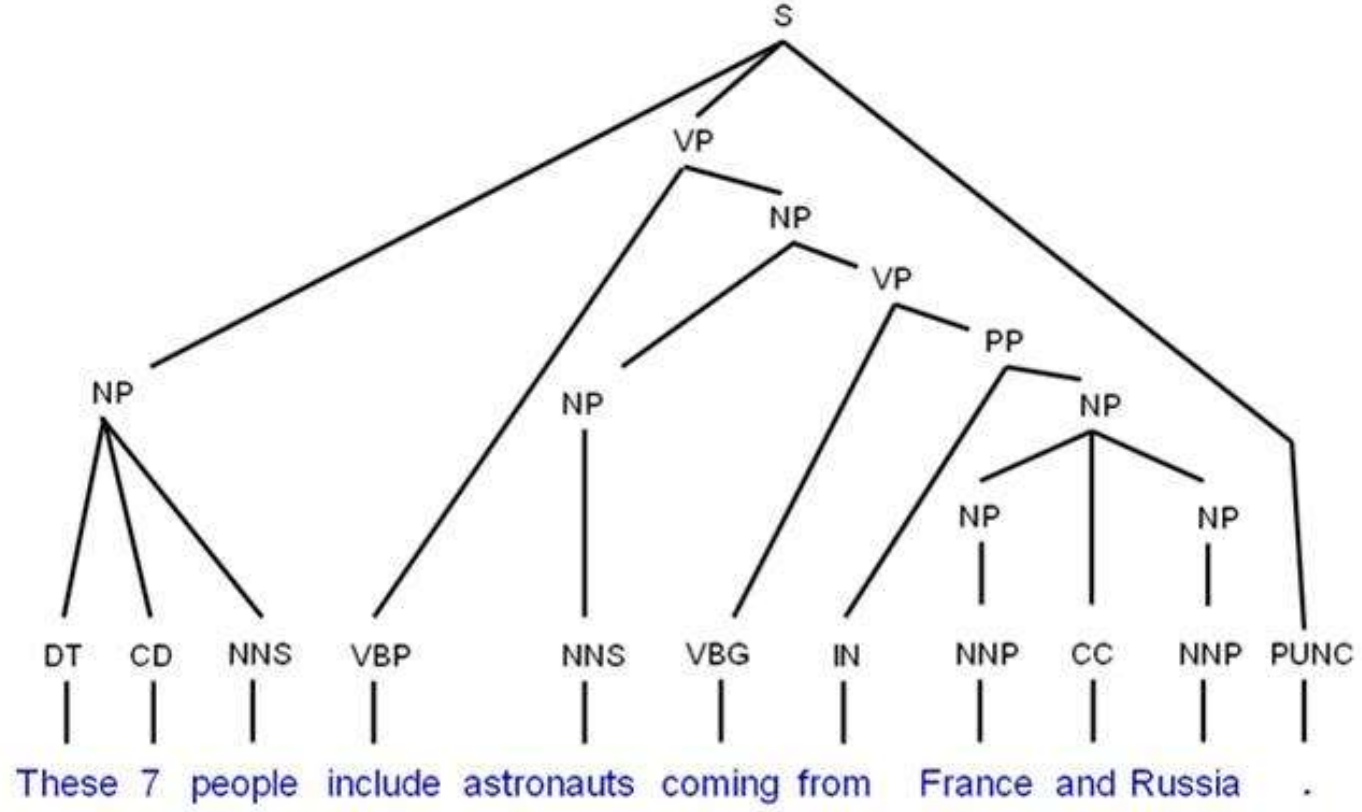
- 任务：将符合Java语言规范的源码编译为符合Java虚拟机规范的Class文件
  - 若输入的Java源码不符合规范则需要报告错误
- Sun的JDK中
  - Java源码级编译器是javac
  - 该编译器是用Java编写的
    - 某种程度上实现了Java的自举（bootstrap）
    - 真正实现Java自举还得结合Java写的Java虚拟机
- 其它Java源码级编译器
  - ECJ：Eclipse Compiler for Java
  - Jikes

一些历史

- 从JDK 1.3开始，javac忽略-O（优化）参数
  - 除了Java语言规范规定的常量折叠和条件编译外，基本上不做优化
  - 因为javac相当于传统编译器的前端，而JVM中的JIT编译器相当于后端，把优化都交给后端做也是合理的
- JDK 1.4.2开始javac不再通过jsr/ret实现finally子句
- Java SE 7开始JVM拒绝加载含有jsr/jsr_w/ret等指令的方法的类型
- JDK 1.5开始javac中泛型的实现部分来自GJC（Generic Java Compiler）

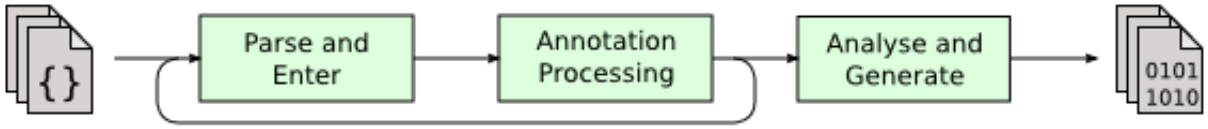
语法树

- 根据语法规则，将语言中隐含的结构表现出来
- 例子：一个英语句子的语法树：



图片来源

javac工作流程



com.sun.tools.javac.main.JavaCompiler中，

compile():

```
initProcessAnnotations(processors);

// These method calls must be chained to avoid memory leaks
delegateCompiler =
processAnnotations(enterTrees(stopIfError(parseFiles(sourceFileObjects))),
                                                        classnames);
delegateCompiler.compile2();
```

compile2():

```
while (todo.nonEmpty())
    generate(desugar(flow(attribute(todo.next()))));
```

javac工作流程

- 解析（parse）与输入到符号表（enter）
- 注解处理（annotation processing）
- 分析与代码生成
  - 属性标注与检查（Attr与Check）
  - 数据流分析（Flow）
  - 将泛型类型转换为裸类型（TransType）
  - 解除语法糖（Lower）
  - 生成Class文件（Gen）

raw type

参考资料

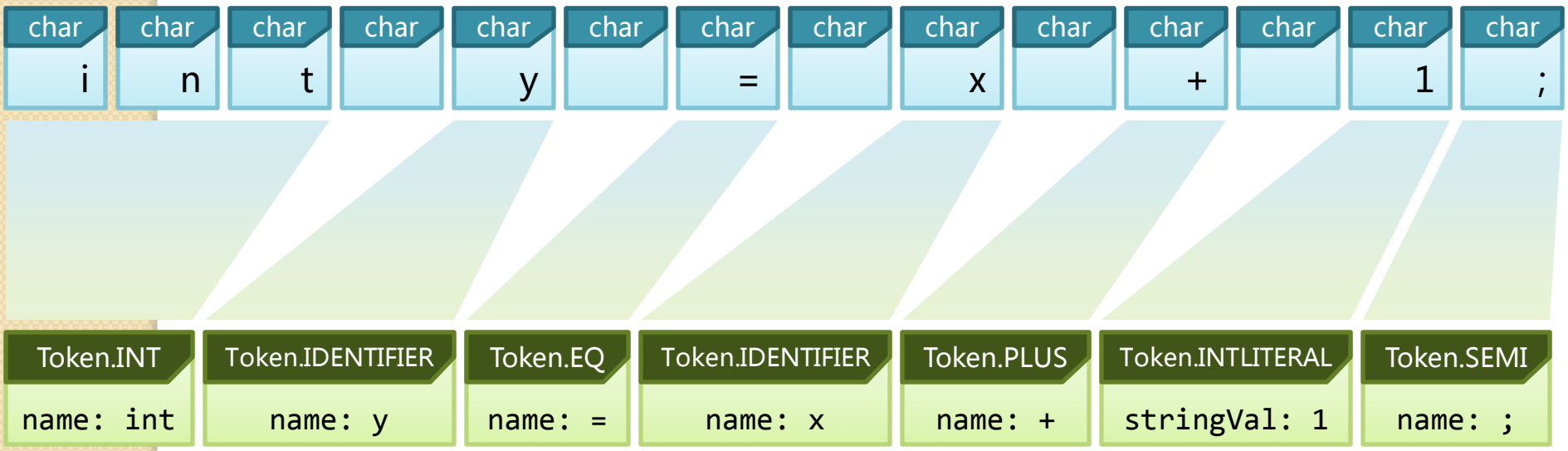
解析（parse）

- 词法分析
  - com.sun.tools.javac.parser.Scanner
  - 手写的ad-hoc方式构造的词法分析器
  - 根据词法将字符序列转换为token序列

- 语法分析
  - com.sun.tools.javac.parser.Parser
  - 手写的递归下降+运算符优先级式语法分析器
  - 根据语法由token序列生成抽象语法树
  - 语法分析后的所有步骤都在抽象语法树上进行
    - 后续如注解处理等步骤都不是简单的“文本替换”

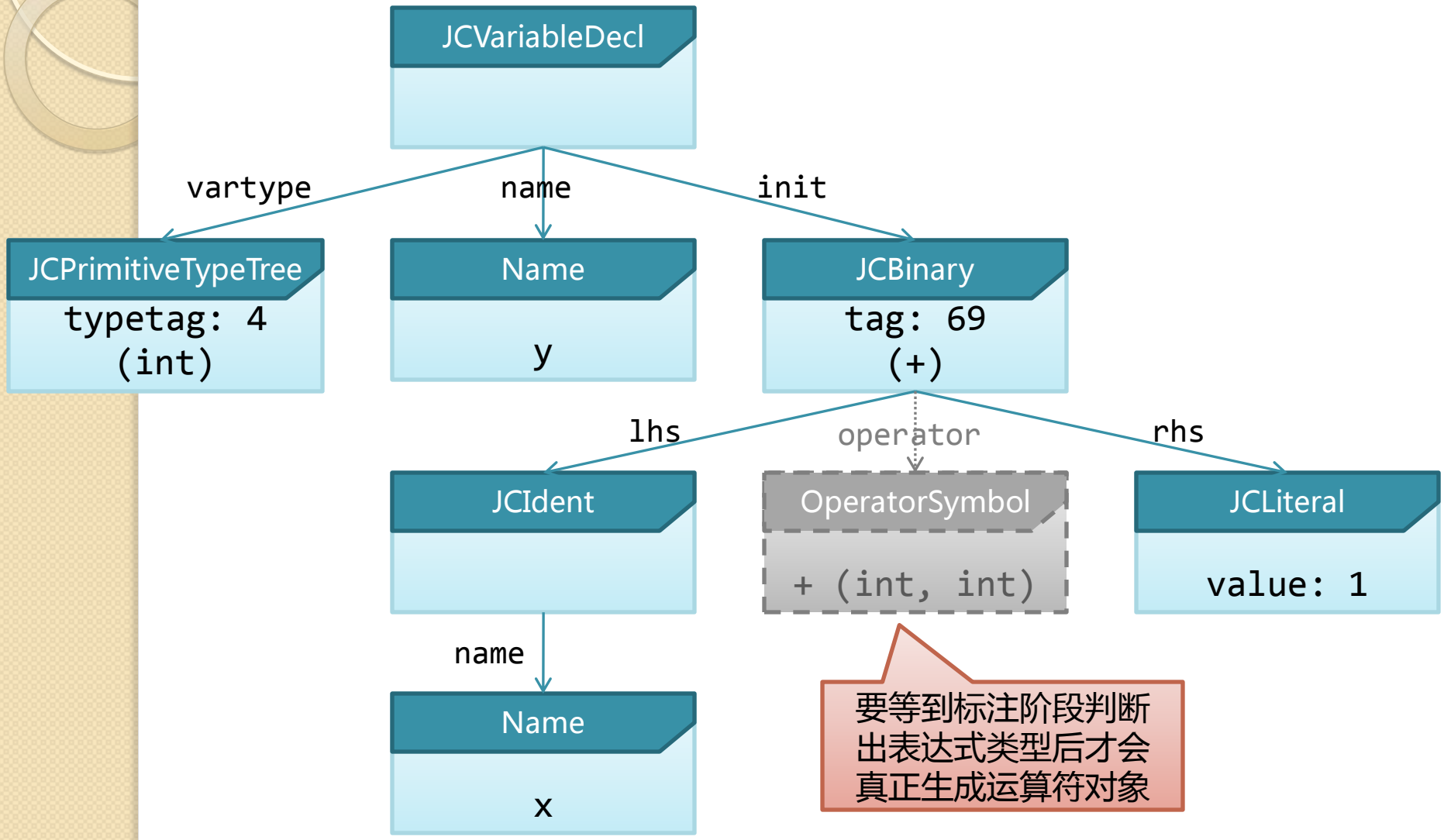
词法分析

```
int y = x + 1;
```


char i
char n
char t
char
char y
char
char =
char
char x
char
char +
char
char 1
char ;
Token.INT
name: int
Token.IDENTIFIER
name: y
Token.EQ
name: =
Token.IDENTIFIER
name: x
Token.PLUS
name: +
Token.INTLITERAL
stringVal: 1
Token.SEMI
name: ;

语法分析

```
int y = x + 1;
```


JCVariableDecl
vartype
name
init
JCPrimitiveTypeTree
typetag: 4
(int)
Name
y
JCBinary
tag: 69
(+)
lhs
operator
rhs
JCIdent
OperatorSymbol
+ (int, int)
JCLiteral
value: 1
name
Name
x
要等到标注阶段判断
出表达式类型后才会
真正生成运算符对象

将符号输入到符号表（enter）

- com.sun.tools.javac.comp.Enter
- 每个编译单元的抽象语法树的顶层节点都先被放到待处理列表中
- 逐个处理列表中的节点
- 所有类符号被输入到外围作用域的符号表中
- 若找到package-info.java，将其顶层树节点加入到待处理列表中
- 确定类的参数（对泛型类型而言）、超类型和接口
- 根据需要添加默认构造器
- 将类中出现的符号输入到类自身的符号表中
- 分析和校验代码中的注解（annotation）

完成类定义前

```
public class CompilerTransformationDemo {
}
```

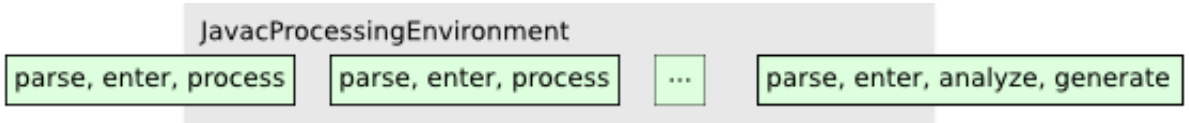
完成类定义后

添加了默认构造器

```
public class CompilerTransformationDemo {

    public CompilerTransformationDemo() {
        super();
    }
}
```

注解处理（annotation processing）

- com.sun.tools.javac.processing.JavacProcessingEnvironment
- 支持用户自定义的注解处理
  - JSR 269（Java 6）
  - 可以读取语法树中任意元素
    - 包括注释（comment）
  - 可以改变类型的定义
  - 可以创建新的类型
  - … 发挥你的想象力：Project Lombok



注解处理前

```
// user code
public @Data class LombokPojoDemo {
    private String name;
}


// from project lombok
@Target(ElementType.TYPE)
@Retention(RetentionPolicy.SOURCE)
public @interface Data {
    String staticConstructor() default "";
}
```

注解处理后

@Data注解已经去掉

默认构造器是在输入符号表的时候添加的

由Lombok的注解处理器添加

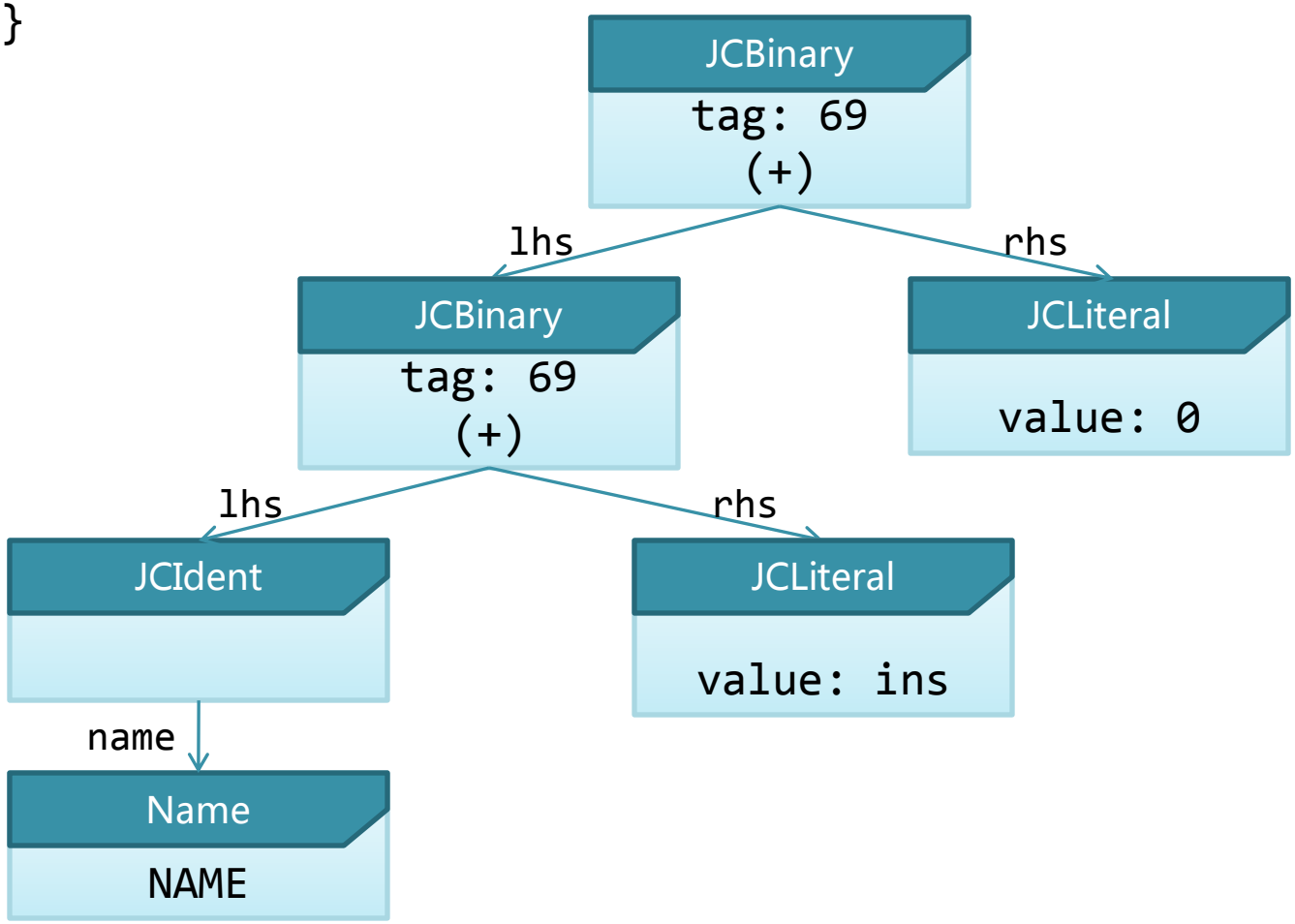
```
public class LombokPojoDemo {
    private String name;

    public LombokPojoDemo() {
    }

    public String getName() {
        return name;
    }

    public void setName(final String name) {
        this.name = name;
    }

    @java.lang.Override
    public boolean equals(final java.lang.Object o) {
        if (o == this) return true;
        if (o == null) return false;
        if (o.getClass() != this.getClass()) return false;
        final LombokPojoDemo other = (LombokPojoDemo)o;
        if (this.name == null ? other.name != null : !this.name.equals(other.name)) return false;
        return true;
    }

    @java.lang.Override
    public int hashCode() {
        final int PRIME = 31;
        int result = 1;
        result = result * PRIME + (this.name == null ? 0 : this.name.hashCode());
        return result;
    }

    @java.lang.Override
    public java.lang.String toString() {
        return "LombokPojoDemo(name=" + name + ")";
    }
}
```

标注（Attr）和检查（Check）

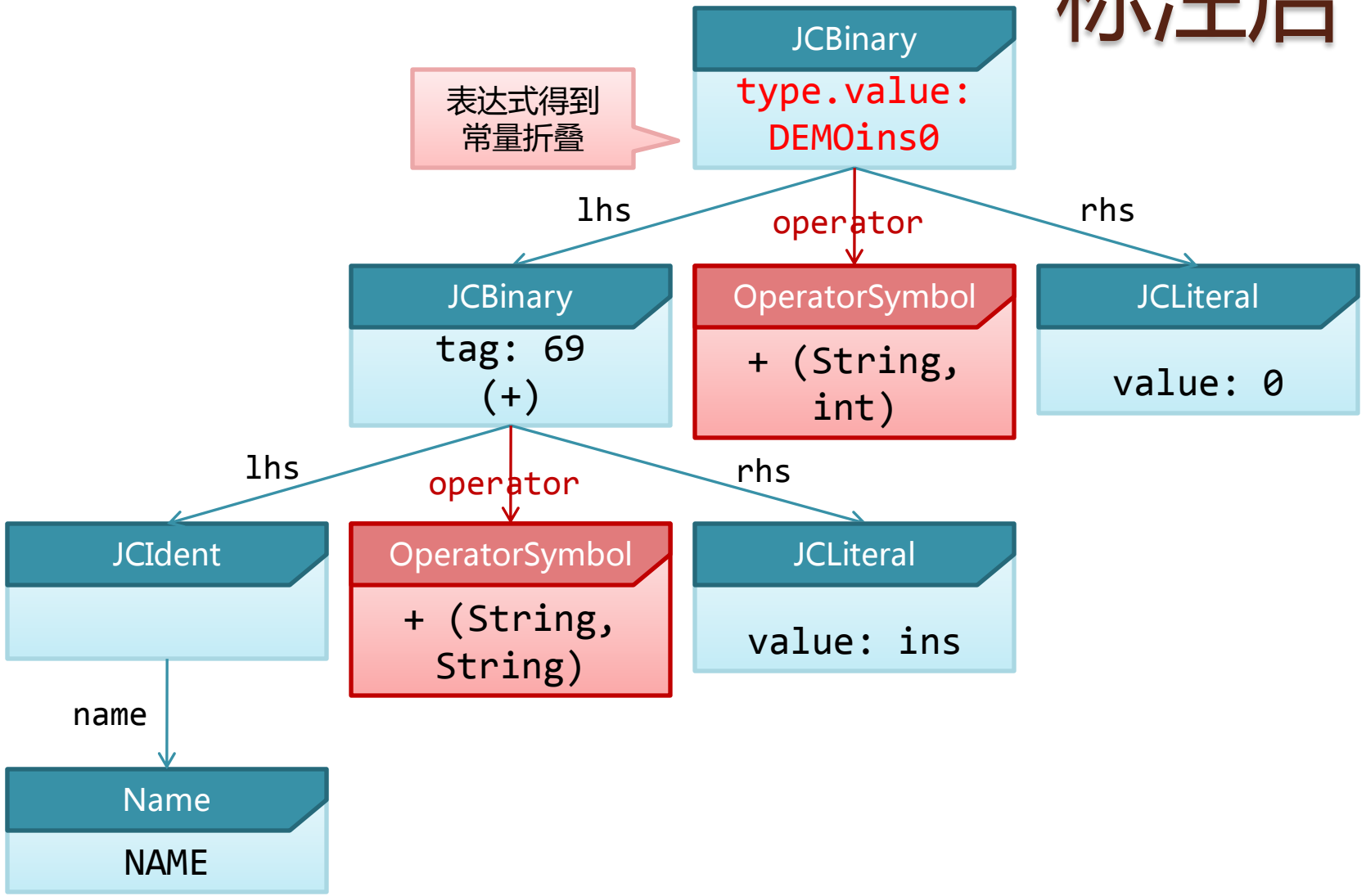
- com.sun.tools.javac.comp.Attr
与 com.sun.tools.javac.comp.Check
- 语义分析的一个步骤
- 将语法树中名字、表达式等元素与变量、方法、类型等联系到一起
- 检查变量使用前是否已声明
- 推导泛型方法的类型参数
- 检查类型匹配性
- 进行常量折叠

标注前

```
public class CompilerTransformationDemo {
    private static final String NAME = "DEMO";
    private String instanceName = NAME + "ins" + 0;
}
```



标注后


JCBinary
type.value:
DEMOins0
表达式得到
常量折叠
lhs
operator
rhs
JCBinary
tag: 69
(+)
OperatorSymbol
+ (String,
int)
JCLiteral
value: 0
lhs
operator
rhs
JCIdent
OperatorSymbol
+ (String,
String)
JCLiteral
value: ins
name
Name
NAME

数据流分析（Flow）

- com.sun.tools.javac.comp.Flow
- 语义分析的一个步骤
- 检查所有语句都可到达
- 检查所有checked exception都被捕获或抛出
- 检查变量的确定性赋值
  - 所有局部变量在使用前必须确定性赋值
  - 有返回值的方法必须确定性返回值
- 检查变量的确定性不重复赋值
  - 为保证final的语义

转换类型（TransTypes）

- com.sun.tools.javac.comp.TransTypes
- 解除语法糖的一个步骤
- 将泛型Java转换为普通Java
  - 同时插入必要的类型转换代码

转换类型前

```
public void desugarGenericToRawAndCheckcastDemo() {
    List<Integer> list = Arrays.asList(1, 2, 3);
    list.add(4);
    int i = list.get(0);
}
```

转换类型后

将泛型类型转换为raw type

```
public void desugarGenericToRawAndCheckcastDemo() {
    List list = Arrays.asList(1, 2, 3);
    list.add(4);
    int i = (Integer)list.get(0);
}
```

添加了强制类型转换来保持泛型的语义

解除语法糖（Lower）

- com.sun.tools.javac.comp.Lower
- 解除语法糖的一个步骤
- 削除`if (false) { … }`形式的无用代码
- 满足下述所有条件的代码被认为是条件编译的无用代码
  - if语句的条件表达式是Java语言规范定义的常量表达式
  - 并且常量表达式值为false则then块为无用代码；反之则else块为无用代码
- 将含有语法糖的语法树改写为含有简单语言结构的语法树
  - 具名内部类/匿名内部类/类字面量
  - 断言（assertion）
  - 自动装箱/拆箱
  - foreach循环
  - enum类型的switch
  - String类型的switch（Java 7）
  - etc …

前一个例子Lower后

```
public void desugarGenericToRawAndCheckcastDemo() {
    List list = Arrays.asList(
        new Integer[]{
            Integer.valueOf(1),
            Integer.valueOf(2),
            Integer.valueOf(3)
        }
    );
    list.add(Integer.valueOf(4));
    int i = ((Integer)list.get(0)).intValue();
}
```

可变长参数的数组包装
原始类型的自动装箱

自动装箱代码

自动拆箱代码

Lower前（再举一例）

```
public class CompilerTransformationDemo {

    public CompilerTransformationDemo() {
        super();
    }

    public void desugarDemo() {
        Integer[] array = {1, 2, 3};
        for (int i : array) {
            System.out.println(i);
        }
        assert array[0] == 1;
    }
}
```

```
public class CompilerTransformationDemo {
    /*synthetic*/ static final boolean $assertionsDisabled =
        !CompilerTransformationDemo.class.desiredAssertionStatus();

    public CompilerTransformationDemo() {
        super();
    }

    public void desugarDemo() {
        Integer[] array = {
      Integer.valueOf(1), Integer.valueOf(2), Integer.valueOf(3)};
        for (Integer[] arr$ = array, len$ = arr$.length, i$ = 0;
             i$ < len$; ++i$) {
            int i = arr$[i$].intValue();
            {
                System.out.println(i);
            }
        }
        if (!$assertionsDisabled && !(array[0].intValue() == 1))
            throw new AssertionError();
    }
}
```

Lower后

生成Class文件（Gen）

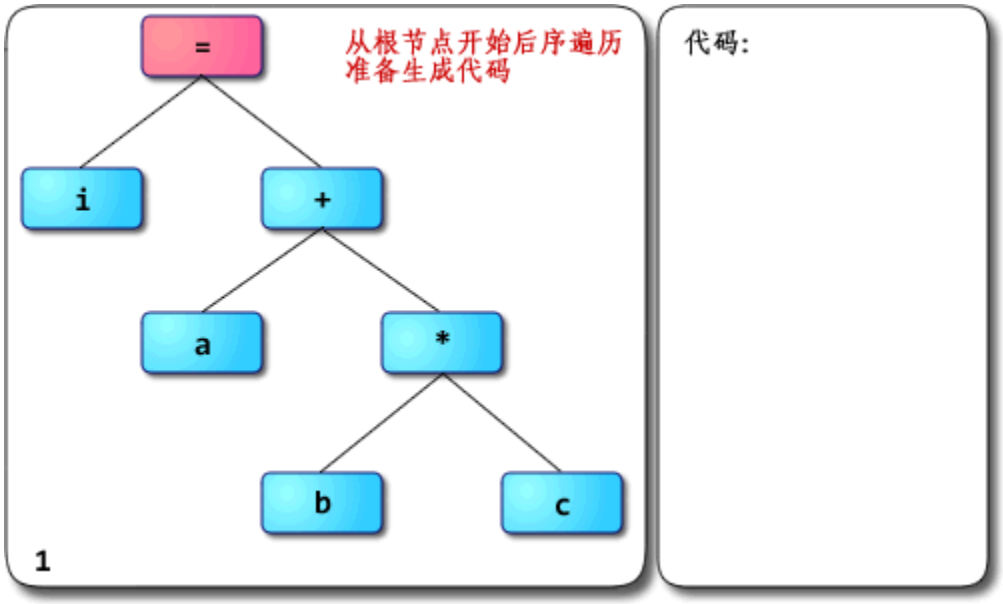
- com.sun.tools.javac.jvm.Gen

- 将实例成员初始化器收集到构造器中成为<init>()
- 将静态成员初始化器收集为<clinit>()

- 从抽象语法树生成字节码
  - 后序遍历语法树
  - 进行最后的少量代码转换
    - String的+被生成为StringBuilder操作
    - x++/x--在条件允许时被优化为++x/--x
    - etc ...

- 从符号表生成Class文件
  - 生成Class文件的结构信息
  - 生成元数据（包括常量池）

生成Java虚拟机字节码



原帖链接

Class文件就是字节码么？

CLASS文件

Class文件所记录的信息

- 结构信息
  - Class文件格式版本号
  - 各部分的数量与大小
- 元数据
  - 类 / 继承的超类 / 实现的接口的声明信息
  - 域与方法声明信息
  - 常量池
  - 用户自定义的、 RetentionPolicy为CLASS或RUNTIME的注解
  - ——对应Java源代码中“声明”与“常量”对应的信息
- 方法信息
  - 字节码
  - 异常处理器表
  - 操作数栈与局部变量区大小
  - 操作数栈的类型记录（StackMapTable，Java 6开始）
  - 调试用符号信息（如LineNumberTable、LocalVariableTable）
  - ——对应Java源代码中“语句”与“表达式”对应的信息

Class文件所记录的信息

- 结构信息
  - Class文件格式版本号
  - 各部分的数量与大小
- 元数据
  - 类 / 继承的超类 / 实现的接口的声明信息
  - 域与方法声明信息
  - 常量池
  - 用户自定义的、 RetentionPolicy为CLASS或RUNTIME的注解
  - ——对应Java源代码中“声明”与“常量”对应的信息
- 方法信息
  - 字节码
  - 异常处理器表
  - 操作数栈与局部变量区大小
  - 操作数栈的类型记录（StackMapTable，Java 6开始）
  - 调试用符号信息（如LineNumberTable、LocalVariableTable）
  - ——对应Java源代码中“语句”与“表达式”对应的信息

字节码只代表程序逻辑，
只是Class文件众多组成部分其中之一

Class文件例子

```
import java.io.Serializable;

public class Foo implements Serializable {
    public void bar() {
        int i = 31;
        if (i > 0) {
            int j = 42;
        }
    }
}
```

结构：
声明与常量

代码：
语句与表达式

输出调试符号信息

编译Java源码

```
javac -g Foo.java
```

反编译Class文件

```
javap -c -s -l -verbose Foo
```

Class文件例子

类声明

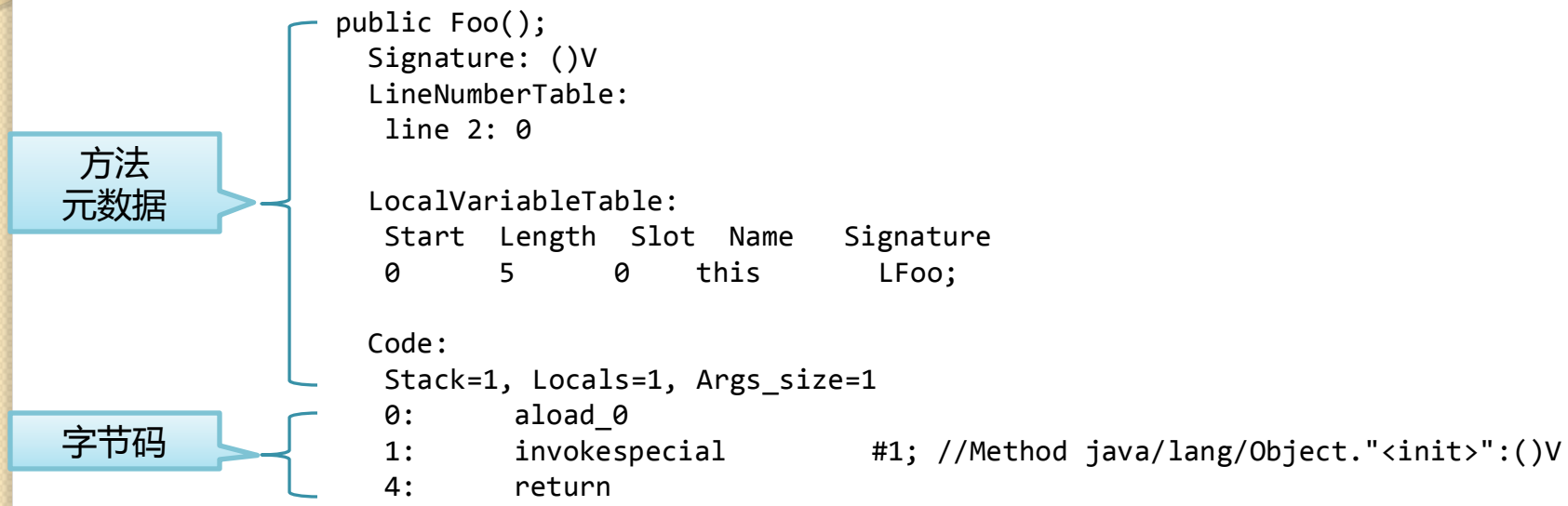
源文件名

Class文件结构信息

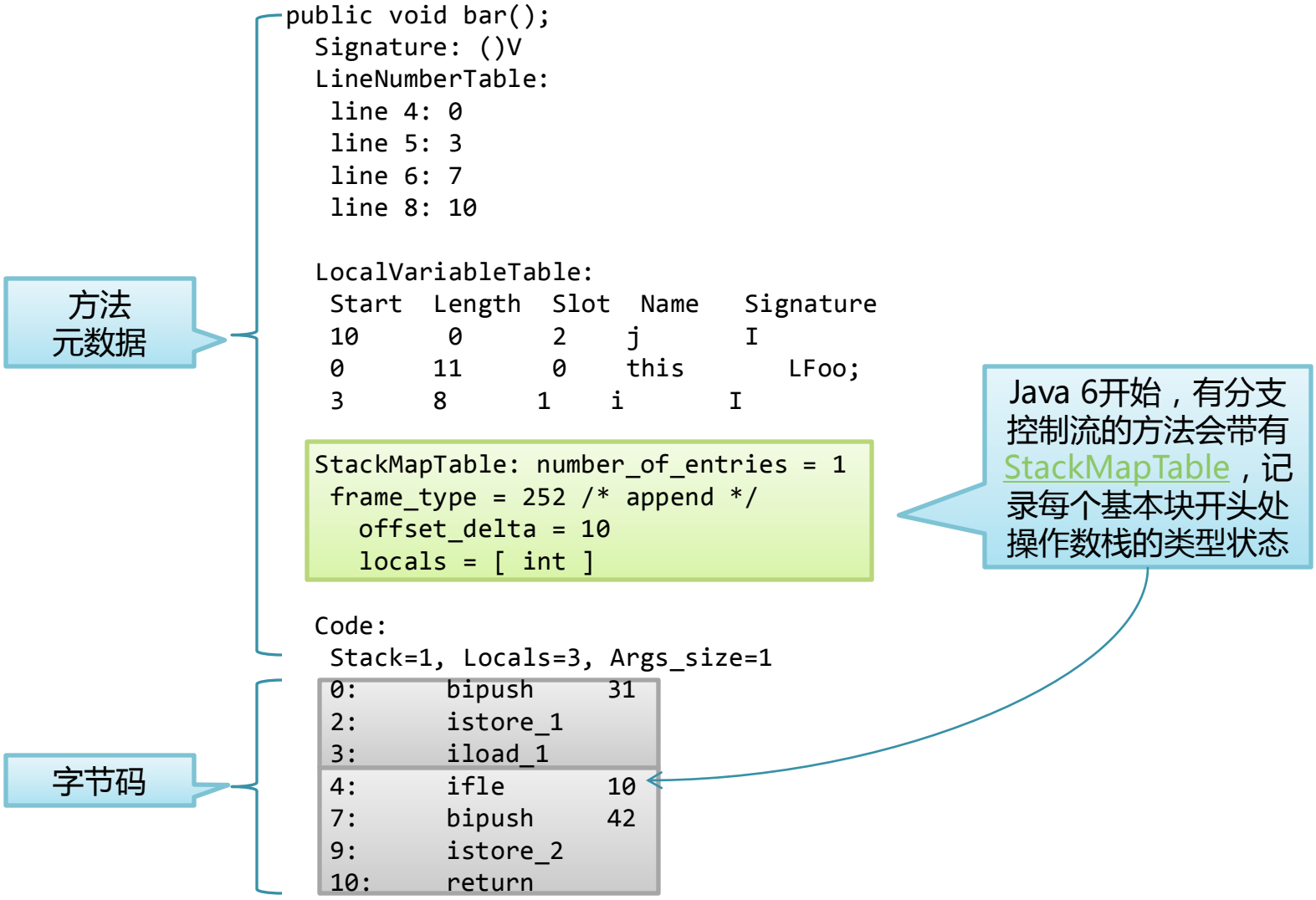
常量池

```
public class Foo extends java.lang.Object implements java.io.Serializable
  SourceFile: "Foo.java"
  minor version: 0
  major version: 50
  Constant pool:
const #1 = Method     #3.#19;    //  java/lang/Object."<init>":()V
const #2 = class      #20;       //  Foo
const #3 = class      #21;       //  java/lang/Object
const #4 = class      #22;       //  java/io/Serializable
const #5 = Asciz      <init>;
const #6 = Asciz      ()V;
const #7 = Asciz      Code;
const #8 = Asciz      LineNumberTable;
const #9 = Asciz      LocalVariableTable;
const #10 = Asciz     this;
const #11 = Asciz     LFoo;;
const #12 = Asciz     bar;
const #13 = Asciz     j;
const #14 = Asciz     I;
const #15 = Asciz     i;
const #16 = Asciz     StackMapTable;
const #17 = Asciz     SourceFile;
const #18 = Asciz     Foo.java;
const #19 = NameAndType          #5:#6;//  "<init>":()V
const #20 = Asciz     Foo;
const #21 = Asciz     java/lang/Object;
const #22 = Asciz     java/io/Serializable;
```

Class文件例子


方法
元数据
字节码
public Foo();
Signature: ()V
LineNumberTable:
line 2: 0
LocalVariableTable:
Start Length Slot Name Signature
0 5 0 this LFoo;
Code:
Stack=1, Locals=1, Args_size=1
0: aload_0
1: invokespecial #1; //Method java/lang/Object."<init>":()V
4: return

Class文件例子


public void bar();
Signature: ()V
LineNumberTable:
line 4: 0
line 5: 3
line 6: 7
line 8: 10
LocalVariableTable:
Start Length Slot Name Signature
10 0 2 j I
0 11 0 this LFoo;
3 8 1 i I
StackMapTable: number_of_entries = 1
frame_type = 252 /* append */
offset_delta = 10
locals = [int]
Code:
Stack=1, Locals=3, Args_size=1
0: bipush 31
2: istore_1
3: iload_1
4: ifle 10
7: bipush 42
9: istore_2
10: return
方法
元数据
字节码
Java 6开始，有分支控制流的方法会带有StackMapTable，记录每个基本块开头处操作数栈的类型状态

Class文件与JDK的版本

JDK版本	Class文件版本（major.minor）
1.0	45.3
1.1	45.3
1.2	46.0
1.3	47.0
1.4	48.0
5	49.0
6	50.0
7	51.0

您有没有想过……

- 为什么ASM无法从接口类型上的方法取得参数的名称？
  - 参数名跟局部变量名都保存在LocalVariableTable；这个属性表跟方法体联系在一起，而接口的方法没有方法体。
- import时每个类型名都写出来与使用*通配符有什么不同？
  - 从Class文件来看没有任何不同，只不过使用通配符增加了源码中名字冲突的可能性而已。

JAVA虚拟机

什么是虚拟机？

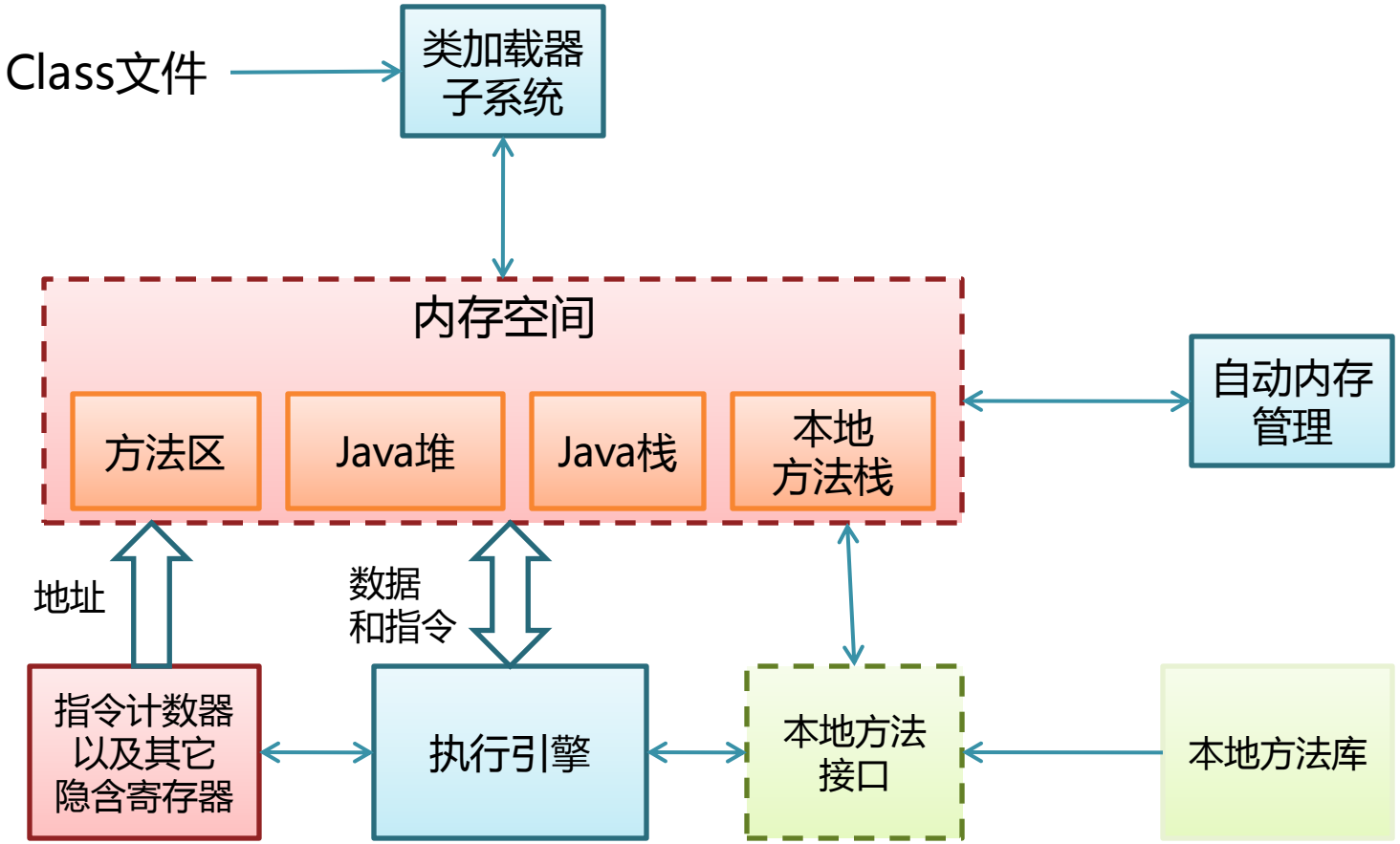
- 多种分类
  - 进程虚拟机
    - 高级语言虚拟机
  - 系统虚拟机
  - 协设计虚拟机
- 模拟执行某种指令集体系结构的软件

什么是Java虚拟机？

- 多层含义
  - 一套规范：Java虚拟机规范
    - 定义概念上Java虚拟机的行为表现
  - 一种实现：例如HotSpot，J9，JRockit
    - 需要实现JVM规范
    - 但具体实现方式不需要与“概念中”的JVM一样
    - **注意**：只有通过JCK测试的才可以合法的称为Java™ VM
  - 一个运行中的实例
    - 某个JVM实现的某次运行的实例
- 只要输入为符合规范的Class文件即可执行
- 并非一定要执行“Java”程序
  - 可以支持其它语言
    - Scala、Clojure、Groovy、Fantom、Fortress、Nice、Jython、JRuby、Rhino、Ioke、Jaskell、（C、Fortran …）…

本节讨论的都是“概念中的JVM”
只是“心理模型”
不代表实现方法

概念中Java虚拟机的基本结构



Java虚拟机的类型系统

Java虚拟机的基本特征

- 基于栈的体系结构
- 动态加载程序
- 安全性
- 自动内存管理
- 多线程支持
- 与本地库的交互

Java虚拟机字节码

- 相当于传统编译器中的中间代码
- 基于栈的指令集体系结构
  - 代码紧凑
  - 方便实现高可移植性的解释器
  - 直观方式解释较基于寄存器的体系结构慢
- 对应Java源代码中语句与表达式的后缀记法（逆波兰记法）
- 指令不定长，在1～4字节间变动

参考资料

Java虚拟机字节码设计目标

- 易于校验（安全性）
- 易于编译（高性能）
- 易于解释执行（实现门槛低）
- 易于移植
- 包含大量类型信息

Java虚拟机字节码指令类别

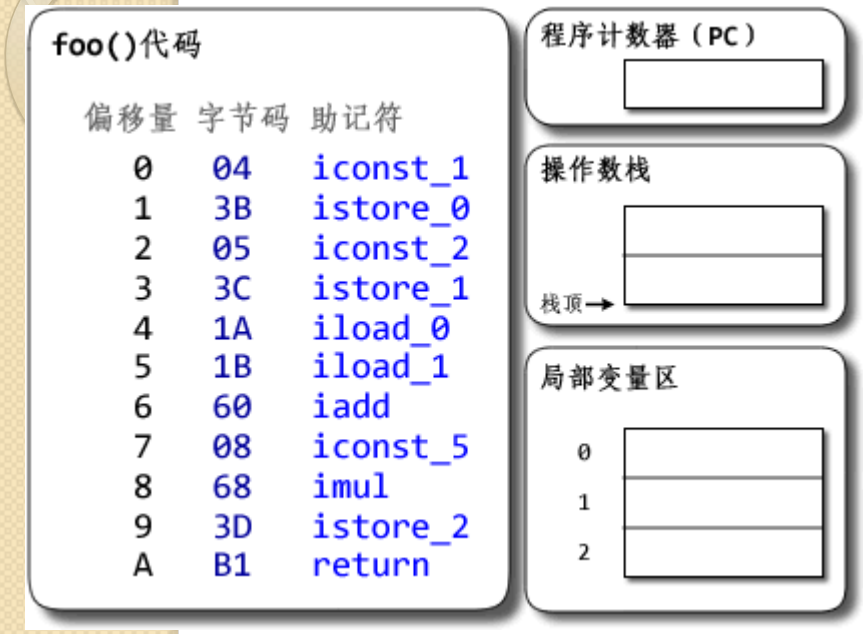
- 局部变量读/写
- 算术与类型转换
- 条件/无条件跳转
- 对象创建和操作
- 数组创建和操作
- 方法调用
- 栈操作（操作数栈）

基于栈与基于寄存器的体系结构的区别

- 保存临时值的位置不同
  - 基于栈：将临时值保存在“求值栈”上
  - 基于寄存器：将临时值保存在寄存器中
- 代码所占的体积不同
  - 基于栈：代码紧凑，体积小，但所需代码条数多
  - 基于寄存器：代码相对大些，但所需代码条数少
- “基于栈”中的“栈”指的是“求值栈”
- 而不是“调用栈”
  - 某些情况下两者是同一个栈
  - 但在JVM中两者是不同的概念
  - JVM中“求值栈”被称为“操作数栈”（operand stack）
  - HotSpot把该栈称为“表达式栈”（expression stack）

参考资料

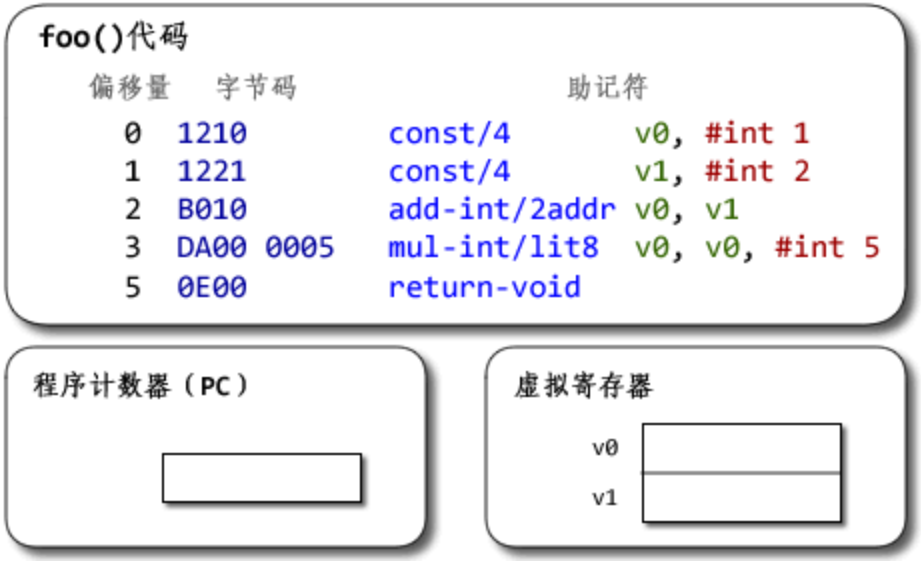
基于栈与基于寄存器的体系结构的区别



概念中的Java虚拟机

```
public class Demo {
    public static void foo() {
        int a = 1;
        int b = 2;
        int c = (a + b) * 5;
    }
}
```

概念中的Dalvik虚拟机



[原帖链接]

JVM的方法调用栈

- 每个Java线程有一个Java方法调用栈
  - 该栈不与其它线程共享
- 每个方法每次被调用时都会在方法调用栈上分配一个栈帧
  - 作为该方法调用的“活动记录”
  - 存储局部数据（包括参数）、临时值、返回值等
  - 也用于动态链接、分派异常等用途
  - 栈帧大小在从Java源码编译为Class文件时已可以确定
    - 记录在Class文件中每个方法的Code属性中
    - max_stack与max_locals
    - 栈帧大小不需要在方法的执行当中变化
- 方法的一次调用结束时，对应的栈帧自动被撤销
  - 无论是正常返回还是抛出异常

参考资料

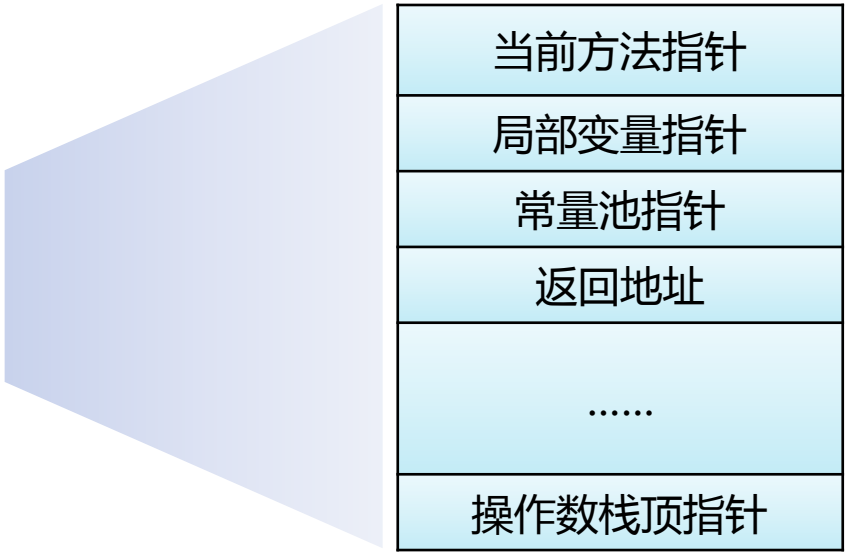
JVM的方法调用栈

- 每个Java栈帧包括：
  - 局部变量区
    - 保存参数与局部变量
  - 操作数栈
    - 保存表达式计算过程中的临时值
  - 指向方法已解析的常量池的引用
    - 用于动态链接、查找常量
  - 其它一些VM内部实现需要的数据

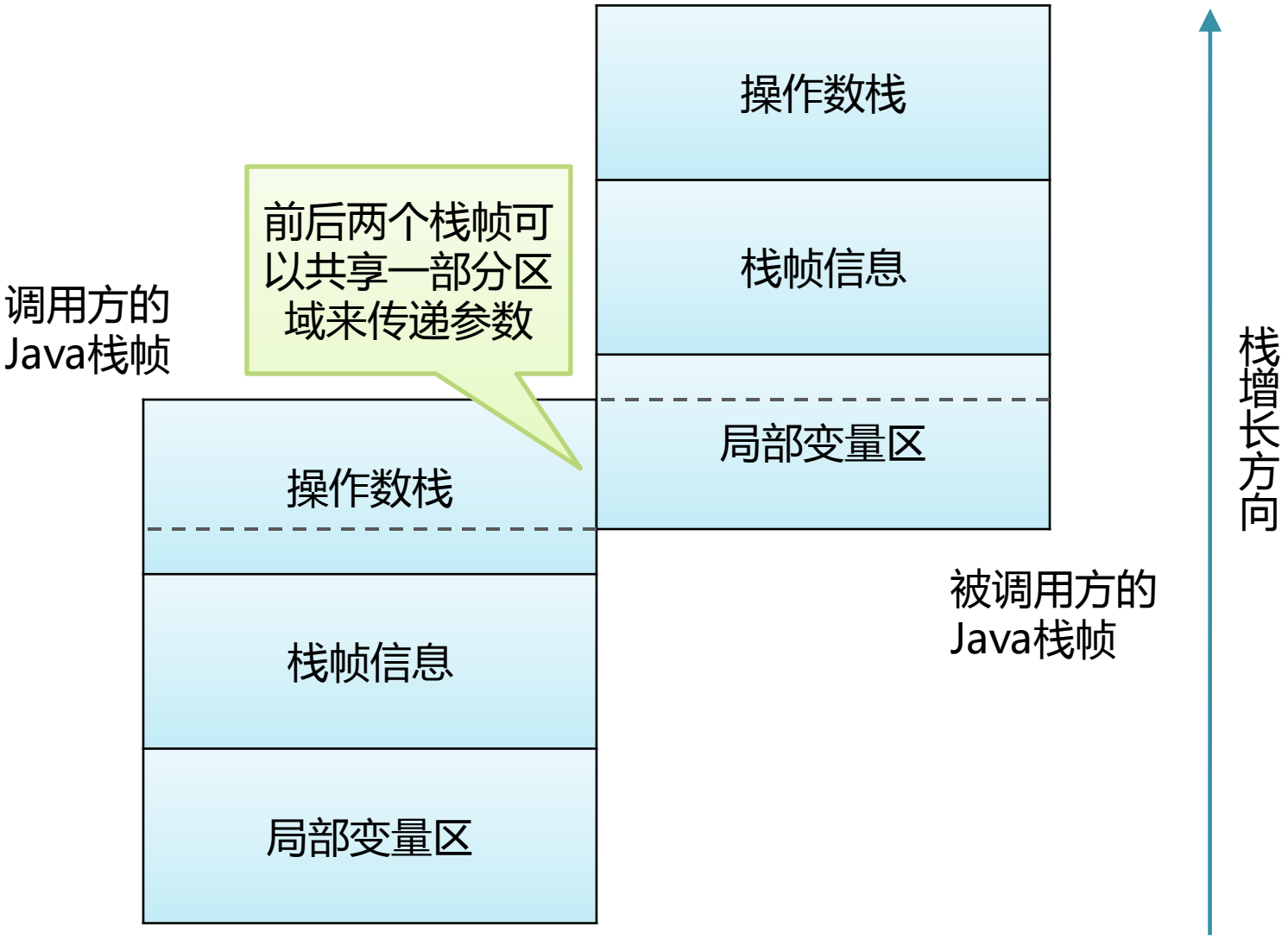
JVM的方法调用栈

一个Java栈帧

栈帧信息中的
一些可能数据



JVM的方法调用栈

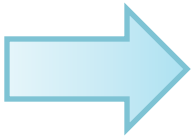

操作数栈
栈帧信息
局部变量区
前后两个栈帧可以共享一部分区域来传递参数
调用方的Java栈帧
操作数栈
栈帧信息
局部变量区
被调用方的Java栈帧
栈增长方向

栈帧中局部变量区的slot的复用

- 局部变量区在每个Java栈帧里都有一份
- 用于保存参数与局部变量
  - 也可能用于临时保存返回值（finally）
  - 也可能保持ret指令的返回地址
  - 以slot为单位，每个slot至少32位宽
  - double与long占两个slot，其它占一个
- 一个slot在一个方法中可以分配给多个变量使用
  - 只要这些变量的作用域不重叠
  - 变量类型是否相同没有关系

栈帧中局部变量区的slot的复用

```
public class LocalVariableDemo {
    public int demo(int i, Object o) {
        int j = i;
        {
            long l = i + j;
        }
        while (j-- > 0) {
            int k = i * 3;
        }
        return j;
    }
}
```



```
Code:
  Stack=2, Locals=6, Args_size=3
   0:   iload_1
   1:   istore_3
   2:   iload_1
   3:   iload_3
   4:   iadd
   5:   i2l
   6:   lstore  4
   8:   iload_3
   9:   iinc    3, -1
  12:   ifle    23
  15:   iload_1
  16:   iconst_3
  17:   imul
  18:   istore  4
  20:   goto    8
  23:   iload_3
  24:   ireturn

StackMapTable: number_of_entries = 2
  frame_type = 252 /* append */
    offset_delta = 8
    locals = [ int ]
  frame_type = 14 /* same */
```

参数与局部变量使用的slot在javac编译时已经确定

栈帧中局部变量区的slot的复用

```
public class LocalVariableDemo {
    public int demo(int i, Object o) {
        int j = i;
        {
            long l = i + j;
        }
        while (j-- > 0) {
            int k = i * 3;
        }
        return j;
    }
}
```

局部变量区：
固定大小为6

0	
1	
2	
3	
4	
5	

栈帧中局部变量区的slot的复用

```
public class LocalVariableDemo {
    public int demo(int i, Object o) {
        int j = i;
        {
            long l = i + j;
        }
        while (j-- > 0) {
            int k = i * 3;
        }
        return j;
    }
}
```

0	this
1	i
2	o
3	
4	
5	

栈帧中局部变量区的slot的复用

```
public class LocalVariableDemo {
    public int demo(int i, Object o) {
        int j = i;
        {
            long l = i + j;
        }
        while (j-- > 0) {
            int k = i * 3;
        }
        return j;
    }
}
```

0	this
1	i
2	o
3	j
4 5	l

保存long型变量时，相邻两个slot合并为一个使用。

栈帧中局部变量区的slot的复用

```
public class LocalVariableDemo {
    public int demo(int i, Object o) {
        int j = i;
        {
            long l = i + j;
        }
        while (j-- > 0) {
            int k = i * 3;
        }
        return j;
    }
}
```

0	this
1	i
2	o
3	j
4 5	l

栈帧中局部变量区的slot的复用

```
public class LocalVariableDemo {
    public int demo(int i, Object o) {
        int j = i;
        {
            long l = i + j;
        }
        while (j-- > 0) {
            int k = i * 3;
        }
        return j;
    }
}
```

0	this
1	i
2	o
3	j
4	
5	

栈帧中局部变量区的slot的复用

```
public class LocalVariableDemo {
    public int demo(int i, Object o) {
        int j = i;
        {
            long l = i + j;
        }
        while (j-- > 0) {
            int k = i * 3;
        }
        return j;
    }
}
```

0	this
1	i
2	o
3	j
4	k
5	

栈帧中局部变量区的slot的复用

```
public class LocalVariableDemo {
    public int demo(int i, Object o) {
        int j = i;
        {
            long l = i + j;
        }
        while (j-- > 0) {
            int k = i * 3;
        }
        return j;
    }
}
```

0	this
1	i
2	o
3	j
4	
5	

栈帧中局部变量区的slot的复用

```
public class LocalVariableDemo {
    public int demo(int i, Object o) {
        int j = i;
        {
            long l = i + j;
        }
        while (j-- > 0) {
            int k = i * 3;
        }
        return j;
    }
}
```

0	this
1	i
2	o
3	j
4	
5	

线程与栈

参考资料

类与类加载

参考资料

Class文件校验

- format checking (JVMS3 4.8)
- bytecode verification (JVMS3 4.10)

参考资料

Class文件校验——类型层次

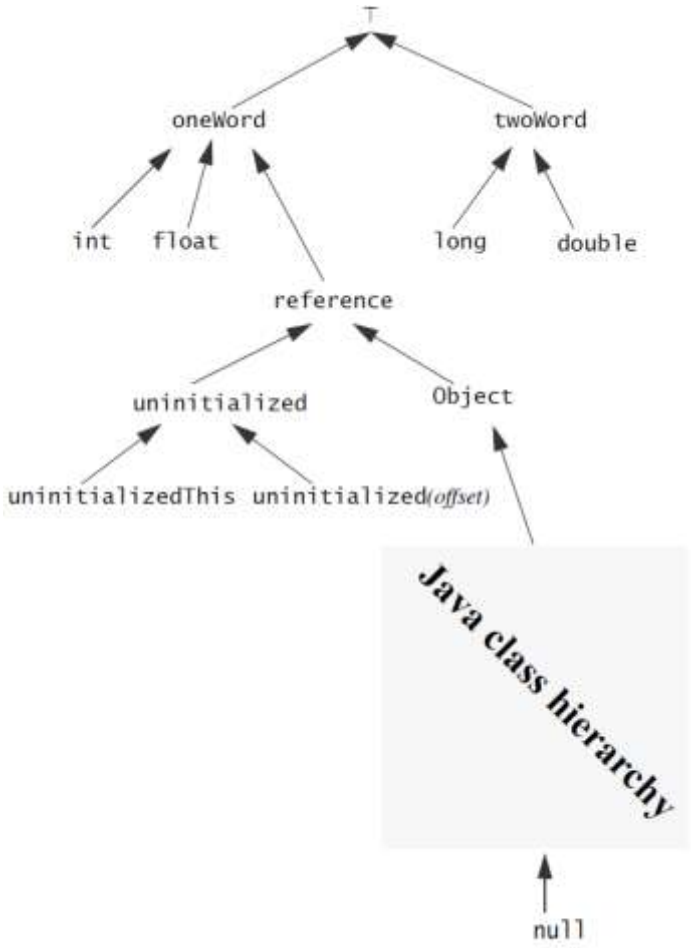
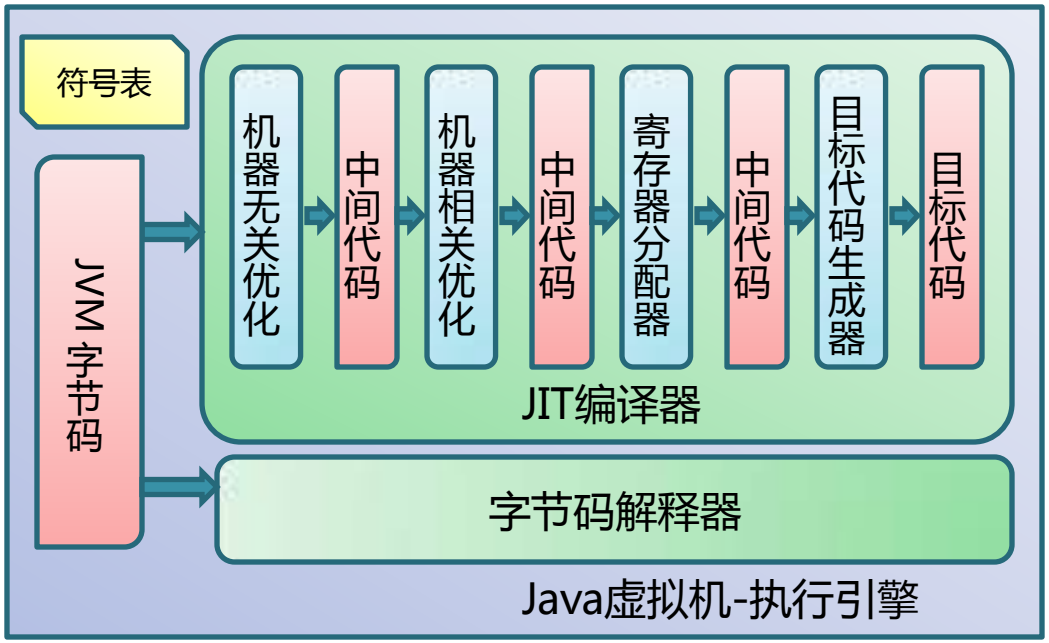


Figure 1: The verification type Hierarchy

参考资料

SUN HOTSPOT VM



Sun HotSpot虚拟机

- Oracle(Sun)实现的一种桌面版JVM
- 从JDK 1.3开始成为Sun JDK的默认VM
- 主要使用C++实现，JNI接口部分用C实现
- 先进的解释器、动态编译器与GC
  - 解释器与编译器结合的混合执行模式
  - 默认启动时解释执行，对执行频率高的代码（热点）做动态编译
  - 故名之“HotSpot VM”
- 2006年底开源
  - 包含在OpenJDK中（GPL 2.0）
- Oracle JDK官方支持的硬件平台
  - Java SE: x86(i386)/x86_64/Itanium(ia64)(*)/SPARC
  - Java SE Embedded(*): x86/ARM/PowerPC
  - 带(*)的没有开源，因而不包含在OpenJDK中

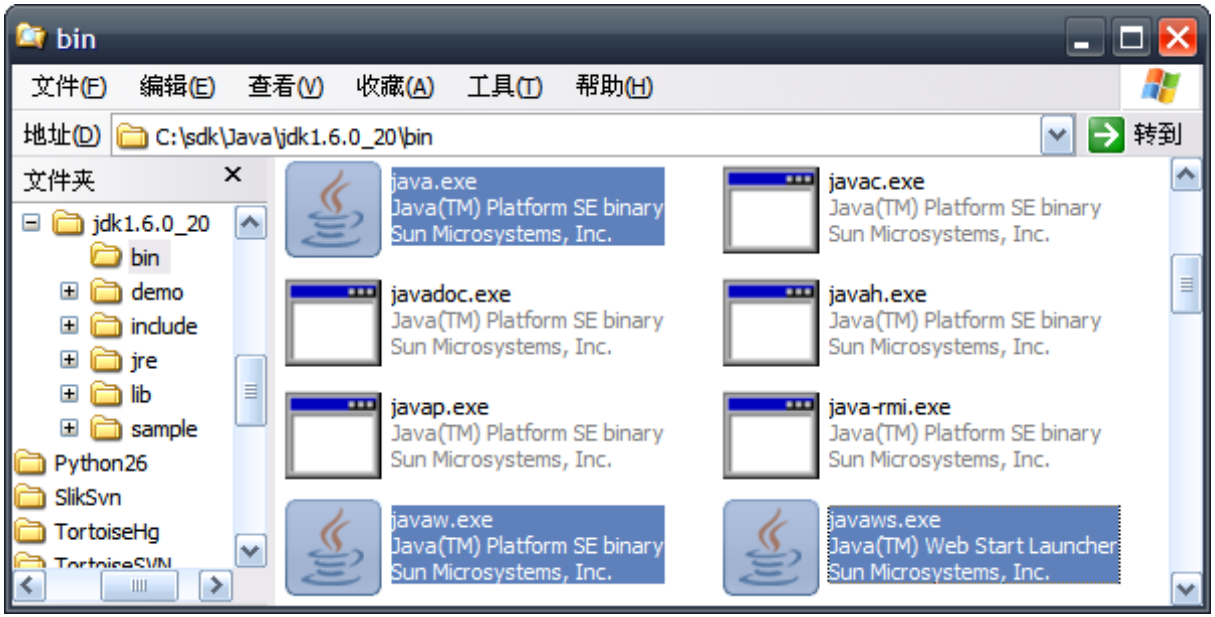
HotSpot虚拟机的前世今生

- 技术源于Smalltalk/Self的实现经验
  - Animorphic Strongtalk
- 1999年4月27日Sun发表Java HotSpot Performance Engine的开发消息
- 经过十多年发展，如今已成为针对Java应用而高度优化的JVM
- Oracle收购Sun后，HotSpot将与JRockit融合，发生较大的设计变更

JVM在哪里？

JVM不在哪里？

- 以下是启动程序（launcher），不是JVM自身
  - `%JAVA_HOME%\bin\java.exe`　　`$JAVA_HOME/bin/java`
  - `%JAVA_HOME%\bin\javaw.exe`　　`$JAVA_HOME/bin/javaws`
  - `%JAVA_HOME%\bin\javaws.exe`
  - `%WINDIR%\system32\java.exe`
- 根据启动参数来选择并配置JVM来执行Java程序



JVM在哪里？

- HotSpot VM的实体（Windows例）
  - %JAVA_HOME%\jre\bin\client\jvm.dll
  - %JAVA_HOME%\jre\bin\server\jvm.dll

※ 留意到client目录比server目录多了个classes.jsa么？
这就是Class Data Sharing的数据文件

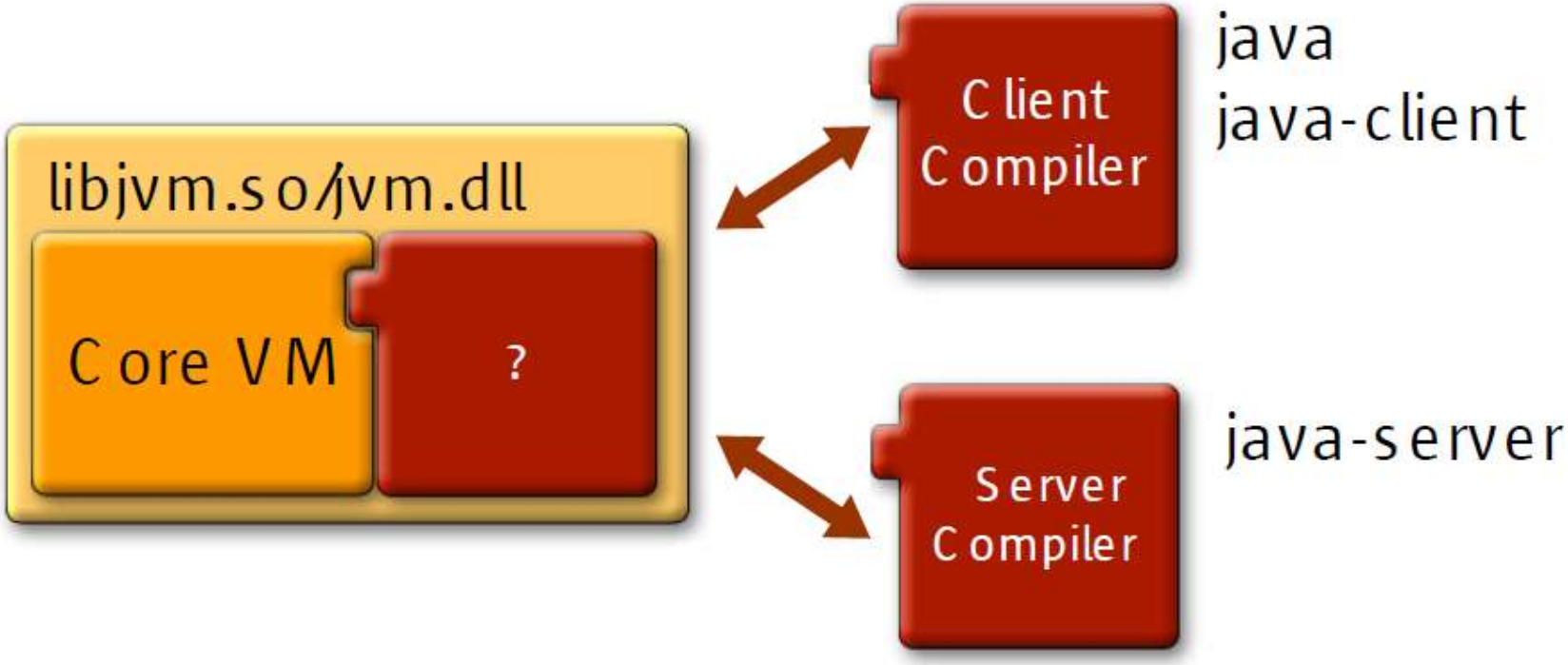
JDK/JRE的文件目录布局：Solaris Linux Windows

HotSpot VM的基本结构

- 一套运行时环境
  - 同一个解释器
    - 高度优化的解释器，执行启动阶段代码及不常用代码
    - 解释代码与编译后代码、本地代码共用同一个栈
  - 同一组GC
    - 有多种GC算法实现可供选择
  - 除编译器外，其它部分上同
    - 不过client与server模式有不同的默认配置
- 两个编译器
  - Client编译器（C1）
    - 轻量，只做少量性能开销比高的优化，占用内存较少
    - 适用于桌面交互式应用，如GUI应用
  - Server编译器（C2，或称为Opto）
    - 重量，大量应用传统编译优化技巧，占用内存相对多些
    - 顶峰速度高
    - 适用于服务器端应用

HotSpot白皮书

HotSpot VM的基本结构



图片来源：JavaOne 2002

HotSpot VM的版本信息

```
java -version
```

```
java version "1.6.0_18"
Java(TM) SE Runtime Environment (build 1.6.0_18-b07)
Java HotSpot(TM) Client VM (build 16.0-b13, mixed mode, sharing)
```

HotSpot VM自身的版本号：
version 16.0, build 13

```
java -server -version
```

```
java version "1.6.0_18"
Java(TM) SE Runtime Environment (build 1.6.0_18-b07)
Java HotSpot(TM) Server VM (build 16.0-b13, mixed mode)
```

环境：32位Windows XP SP3上的Sun JDK 1.6.0 update 18
（注意要用JDK下的java，不要用默认公共JRE下的java）

HotSpot VM的版本信息

```
java version "1.6.0_18"
Java(TM) SE Runtime Environment (build 1.6.0_18-b07)
Java HotSpot(TM) Client VM (build 16.0-b13, mixed mode, sharing)
```

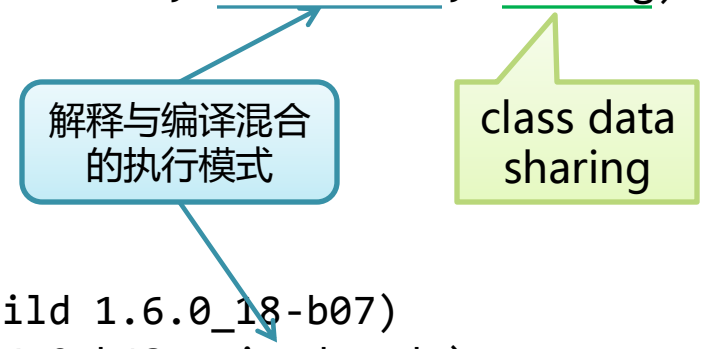
```
java version "1.6.0_18"
Java(TM) SE Runtime Environment (build 1.6.0_18-b07)
Java HotSpot(TM) Server VM (build 16.0-b13, mixed mode)
```

从Java 5开始，Sun HotSpot VM可以根据环境自动选择启动参数，在“服务器级”机器上会自动选用-server模式（但在32位Windows上总是默认使用-client模式）。

“服务器级”指CPU为2核或以上（或者2 CPU或以上），并且内存有2GB或更多的机器。

HotSpot VM的版本信息

```
java version "1.6.0_18"
Java(TM) SE Runtime Environment (build 1.6.0_18-b07)
Java HotSpot(TM) Client VM (build 16.0-b13, mixed mode, sharing)
```




```
java version "1.6.0_18"
Java(TM) SE Runtime Environment (build 1.6.0_18-b07)
Java HotSpot(TM) Server VM (build 16.0-b13, mixed mode)
```

HotSpot VM的版本信息

```
java -Xint -version
```

```
java version "1.6.0_18"
Java(TM) SE Runtime Environment (build 1.6.0_18-b07)
Java HotSpot(TM) Client VM (build 16.0-b13, interpreted mode, sharing)
```

纯解释模式
（禁用JIT编译）

```
java –Xcomp -version
```

```
java version "1.6.0_18"
Java(TM) SE Runtime Environment (build 1.6.0_18-b07)
Java HotSpot(TM) Client VM (build 16.0-b13, compiled mode, sharing)
```

纯编译模式
（但遇到无法编译的方法时，会退回到解释模式执行无法编译的方法）

默认使用的是-Xmixed，
也就是混合模式执行

HOTSPOT VM的运行时支持

启动程序（launcher）

- Windows: java.exe / POSIX: java
- 载入JVM（jvm.dll / libjvm.so）
- 设置启动参数
- 初始化JVM
- 找到主类，main方法，检查signature
- 通过JNI调用main方法

启动程序（launcher）

- 设置类路径
  - -Djava.class.path
  - Windows上设置的类路径顺序与字符串顺序相同，Linux上则相反（实现细节）

HotSpot中的Java对象布局

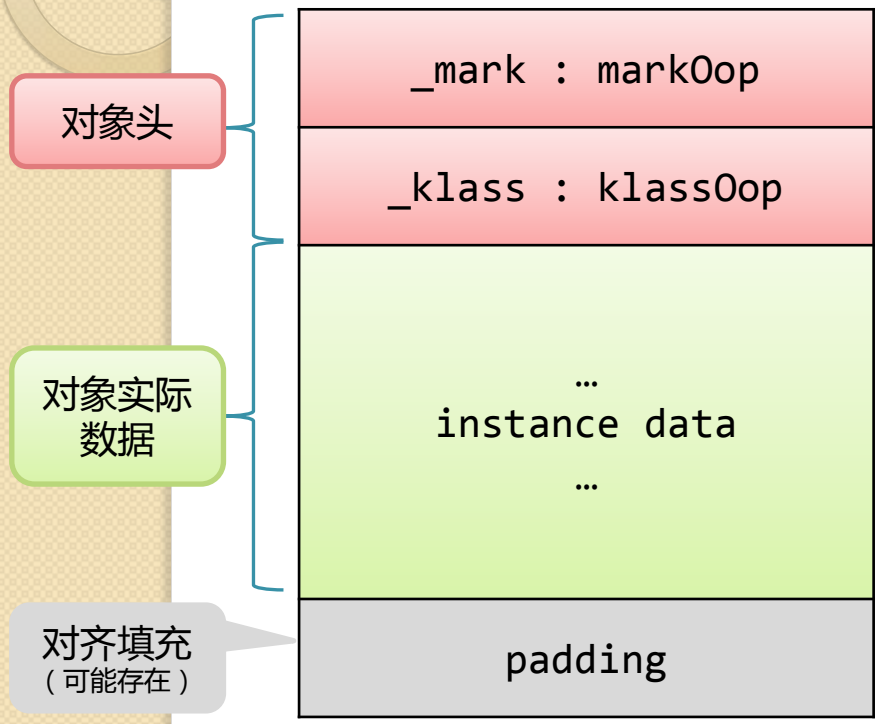
- 怎么介绍好？

参考资料

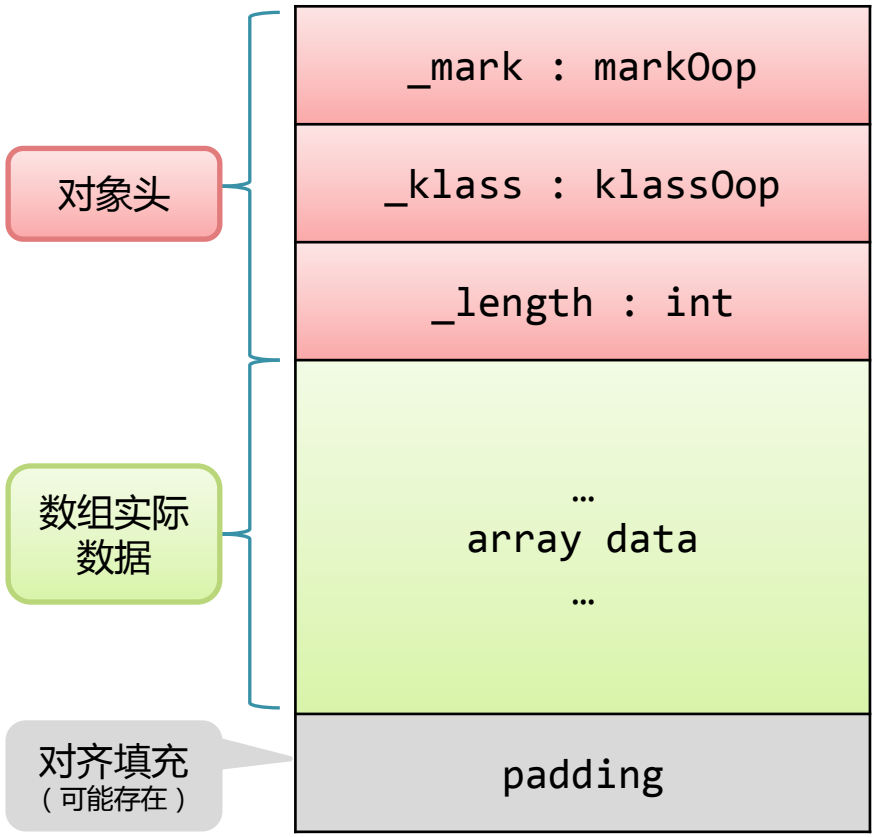
HotSpot中的Java对象布局

- 由两个参数控制：
  - FieldsAllocationStyle
    - 0 - oops, longs/doubles, ints/floats, shorts/chars, bytes/booleans
    - 1（默认） - longs/doubles, ints/floats, shorts/chars, bytes/booleans, oops
  - CompactFields
    - true（默认）尝试将实例变量分配在前面的实例变量之间的空隙中
    - false

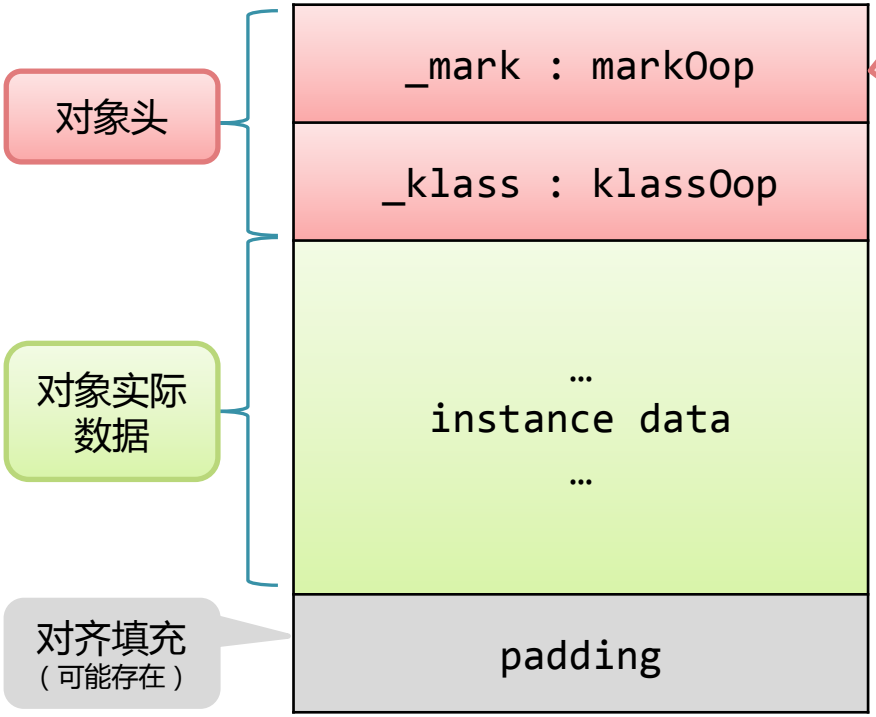
Java对象实例
（instanceOopDesc）



Java数组实例
（objArrayOopDesc
或
typeArrayOopDesc）



Java对象实例
（instanceOopDesc）



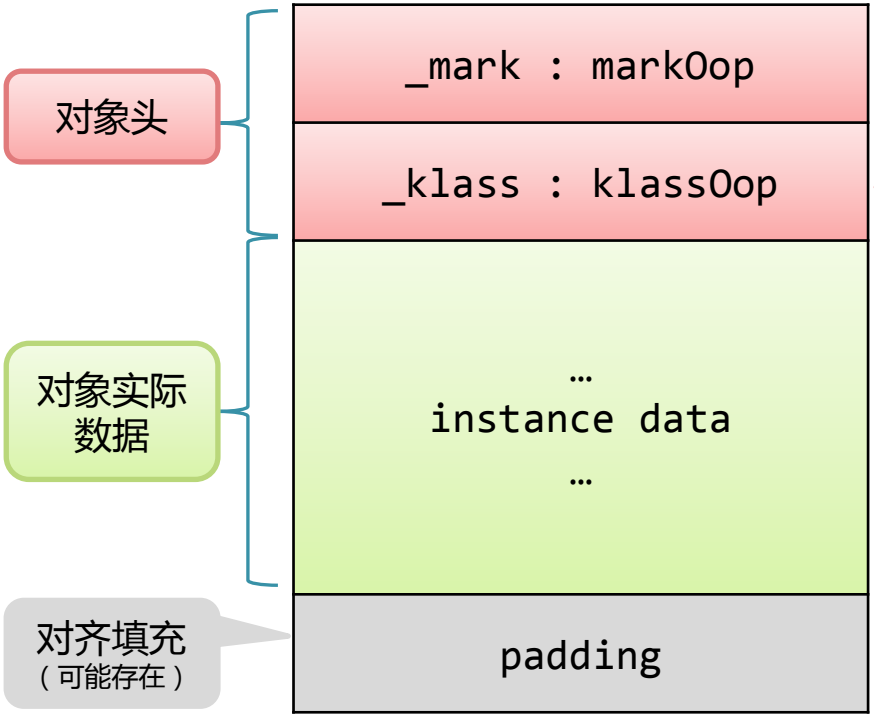
对象状态信息

HotSpot里，GC堆上的对象需要维持一些状态信息，诸如：

- 身份哈希码（identity hash code）
- 当前是否已被GC标记（只在GC过程中需要）
- 当前对象年龄（经历的GC次数）
- 当前是否被当作锁同步
- 最近持有该对象锁的线程ID（用于偏向锁）
- ……等

该字段并非一口气记录上述所有信息，而是根据对象状态有选择性的记录其中一些

Java对象实例
（instanceOopDesc）

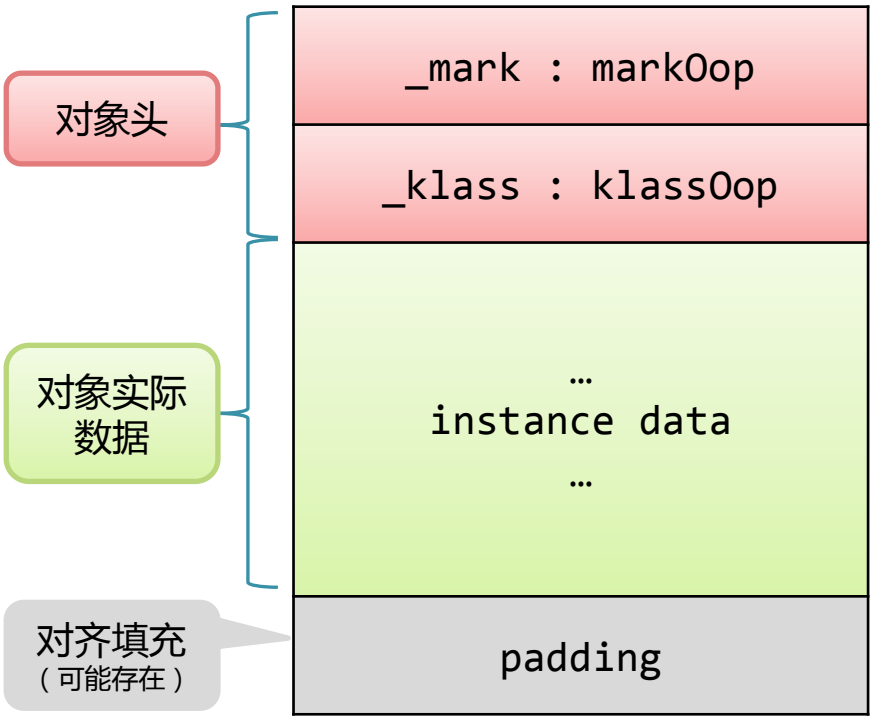


类元数据指针

HotSpot里，所有存储在由GC管理的堆（包括Java堆与PermGen）中的对象从C++看都是oopDesc的子类的实例。每个这样的对象都有一个_klass字段，指向一个描述自身的元数据的对象。

※ Java对象与数组的klass并不是Java语言级的java.lang.Class。Klass用于运行而java.lang.Class只用于Java一侧的反射API；前者中有_java_mirror字段指向后者。

Java对象实例
（instanceOopDesc）



实例数据

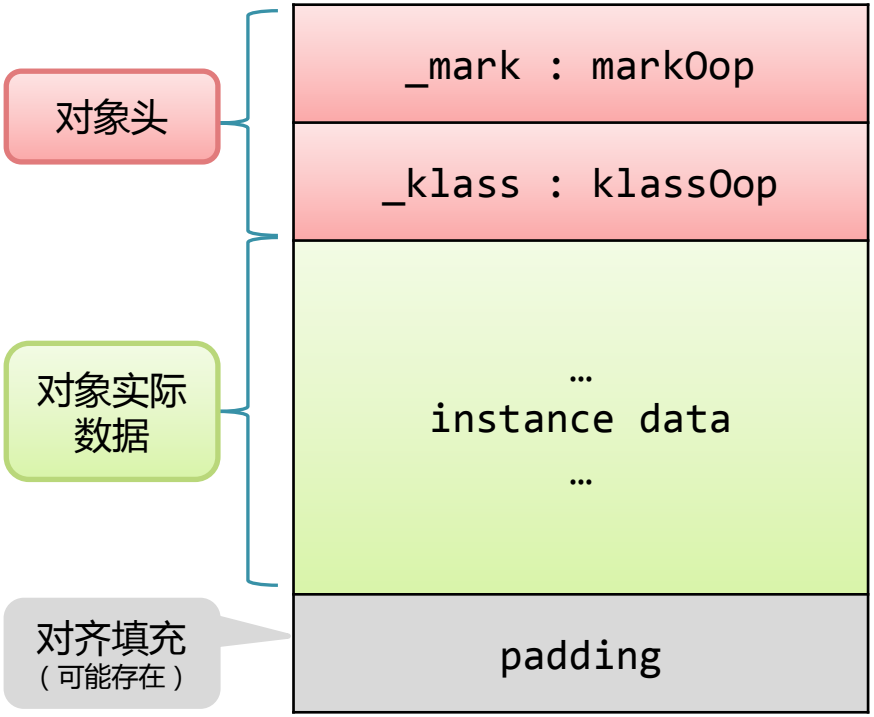
HotSpot里，对象实例数据紧跟在对象头后分配空间。

字段的分配顺序受源码中声明的顺序以及HotSpot的分配策略的影响。
无论是哪种分配策略，宽度相同的字段总是相邻分配的；不同宽度间的字段可能存在对齐填充（padding）。

笼统的说，基类声明的实例字段会出现在派生类声明的实例字段之前。但开启压缩分配模式时，派生类声明的较窄的字段可能会插入到基类的实例字段之间的对齐填充部分。

相关参数：
FieldsAllocationStyle
CompactFields

Java对象实例
（instanceOopDesc）

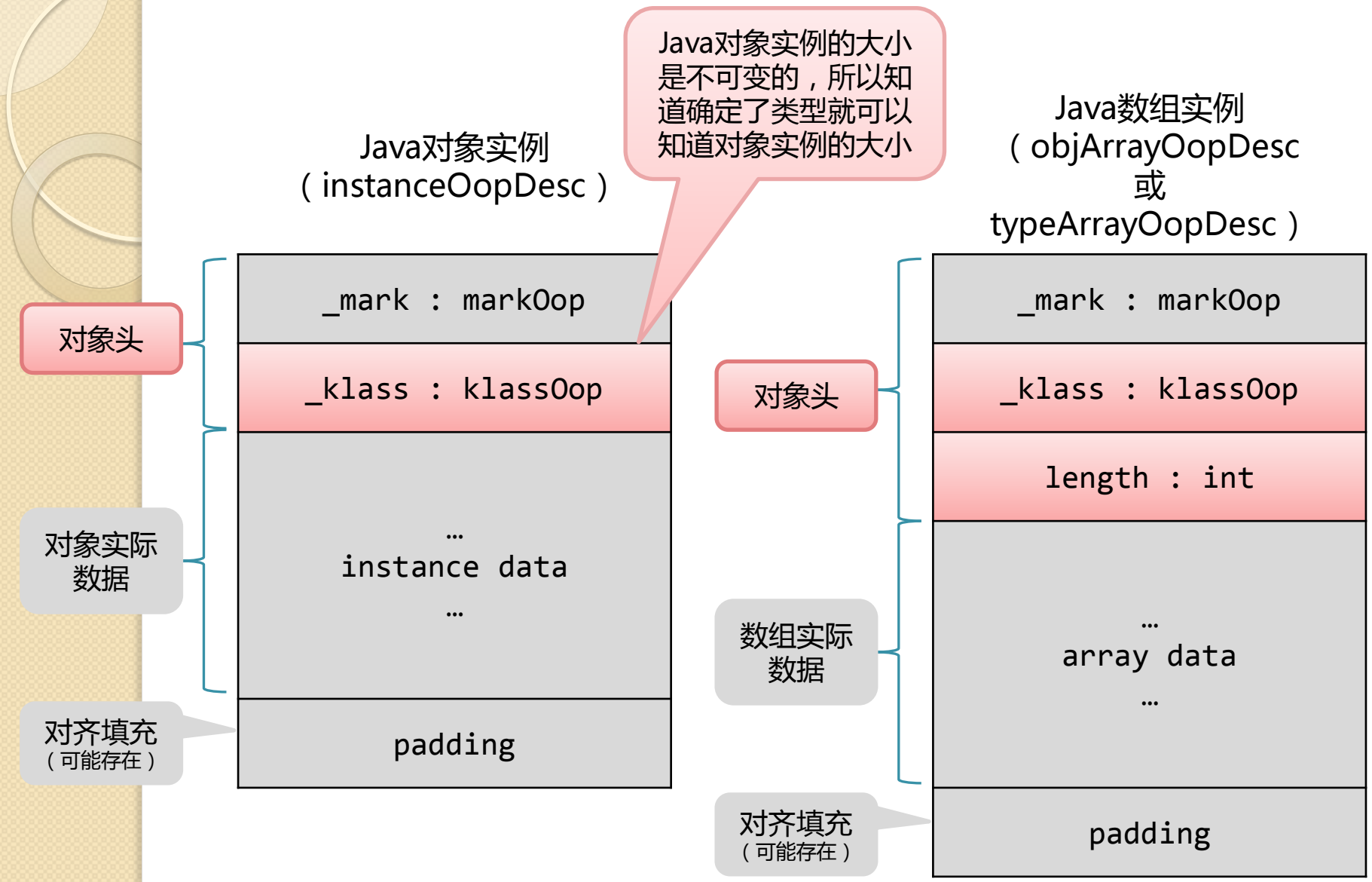


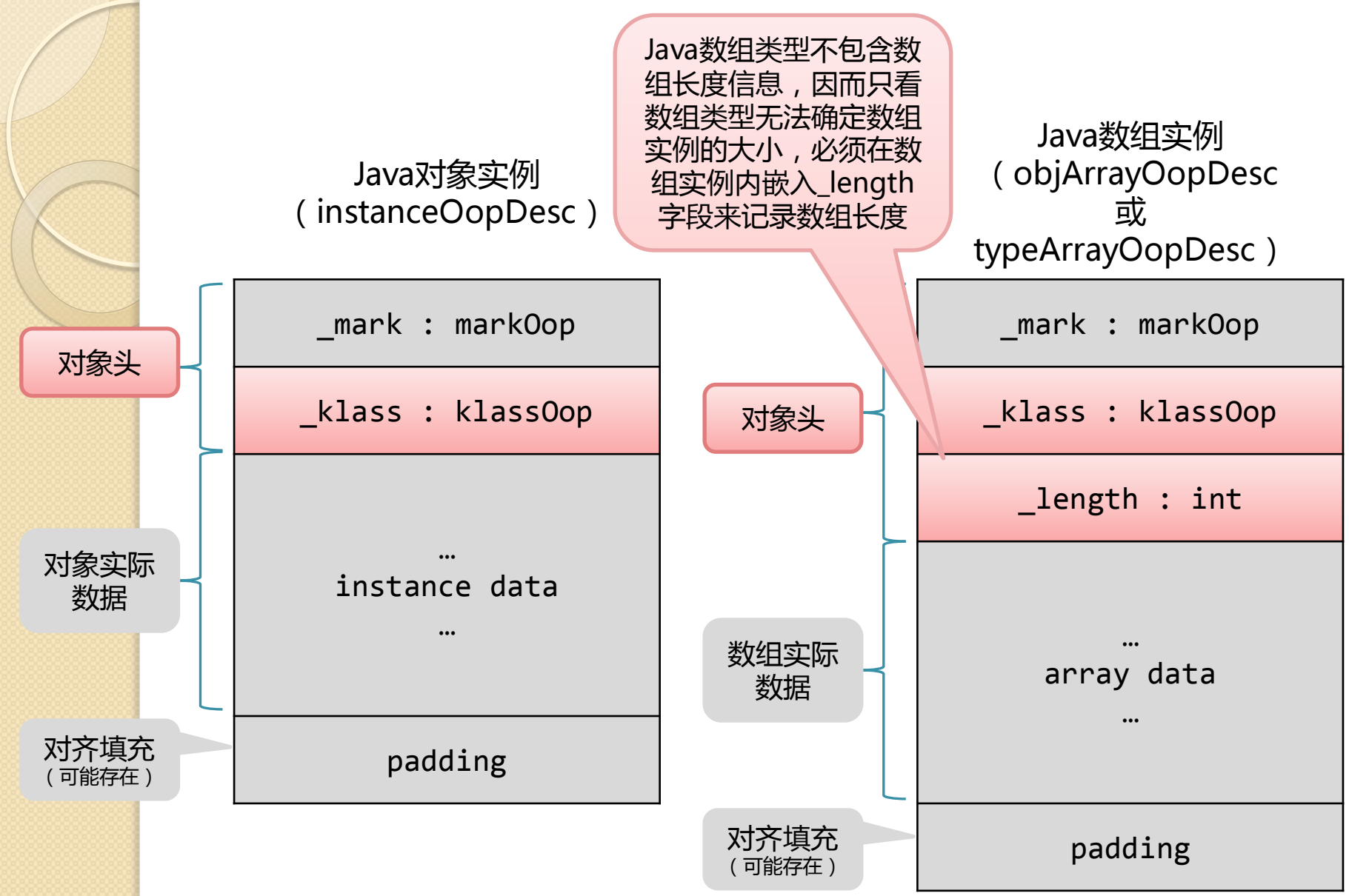
对齐填充（padding）

HotSpot里，GC堆上的对象要求在8字节边界上分配；也就是说，对象的起始地址必须是8的倍数，对象占用的空间也必须是8的倍数。

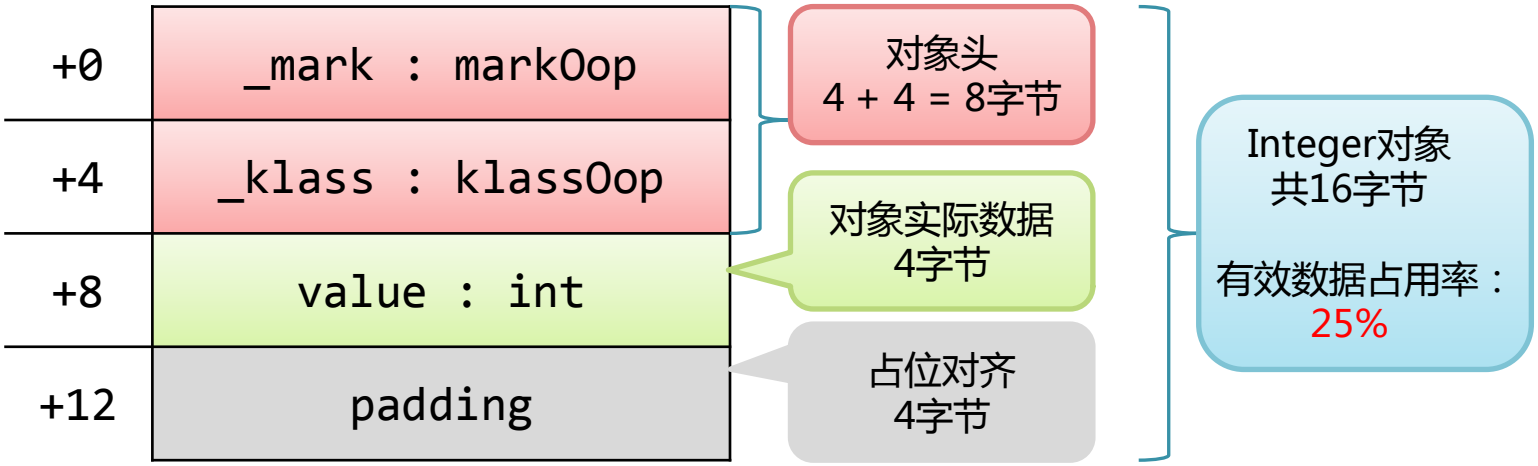
若对象实际需要的大小不足8的倍数，则用0填充不足的部分，直到8字节边界为止。

对齐填充可能出现在不同宽度的字段之间，也可能出现在对象的末尾；或者当不存在未对齐数据时则不会出现对齐填充。

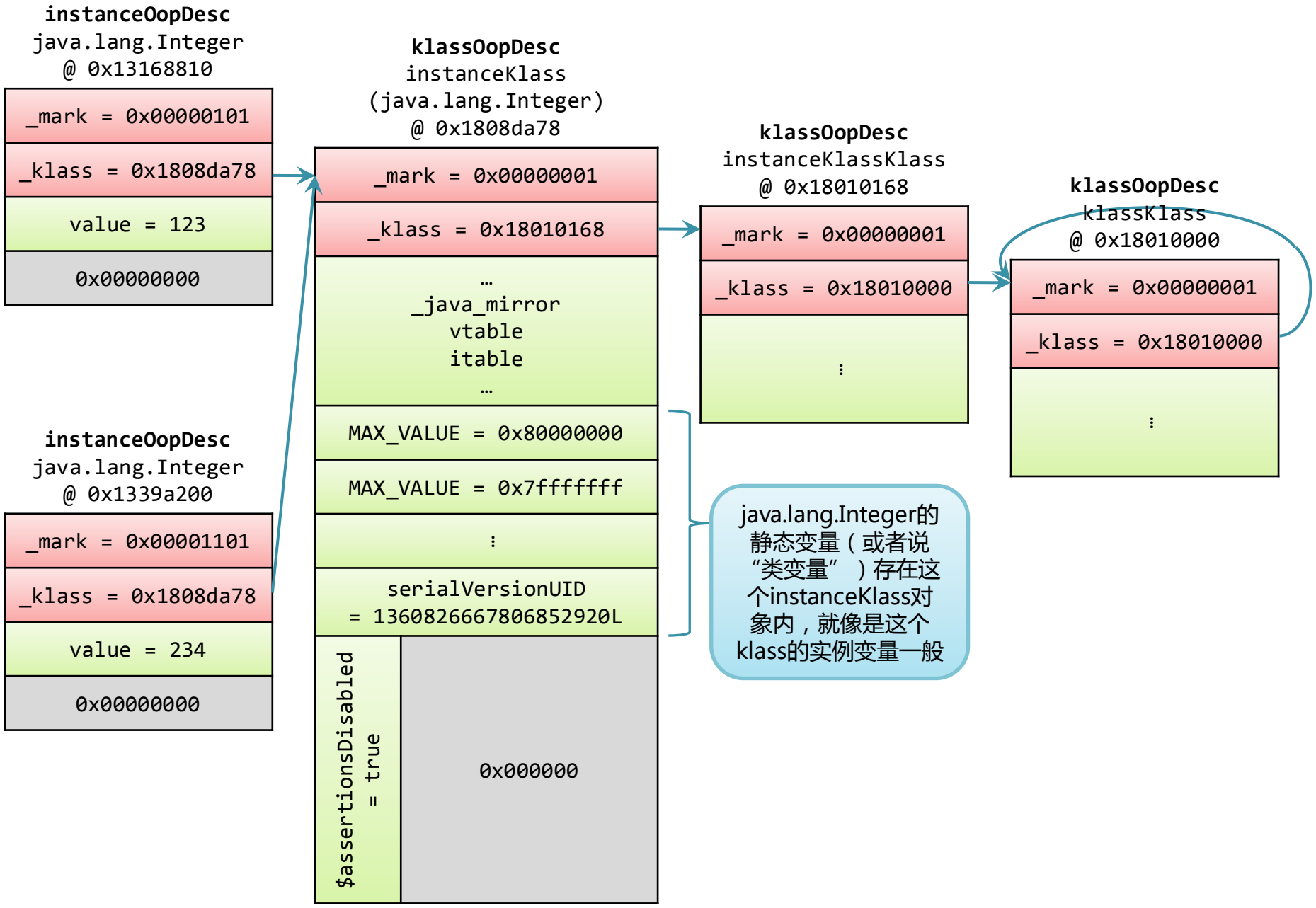

Java对象实例的大小是不可变的，所以知道确定了类型就可以知道对象实例的大小
Java对象实例
（instanceOopDesc）
Java数组实例
（objArrayOopDesc
或
typeArrayOopDesc）
对象头
_mark : markOop
_klass : klassOop
对象实际数据
…
instance data
…
对齐填充
（可能存在）
padding
对象头
_mark : markOop
_klass : klassOop
length : int
数组实际数据
…
array data
…
对齐填充
（可能存在）
padding

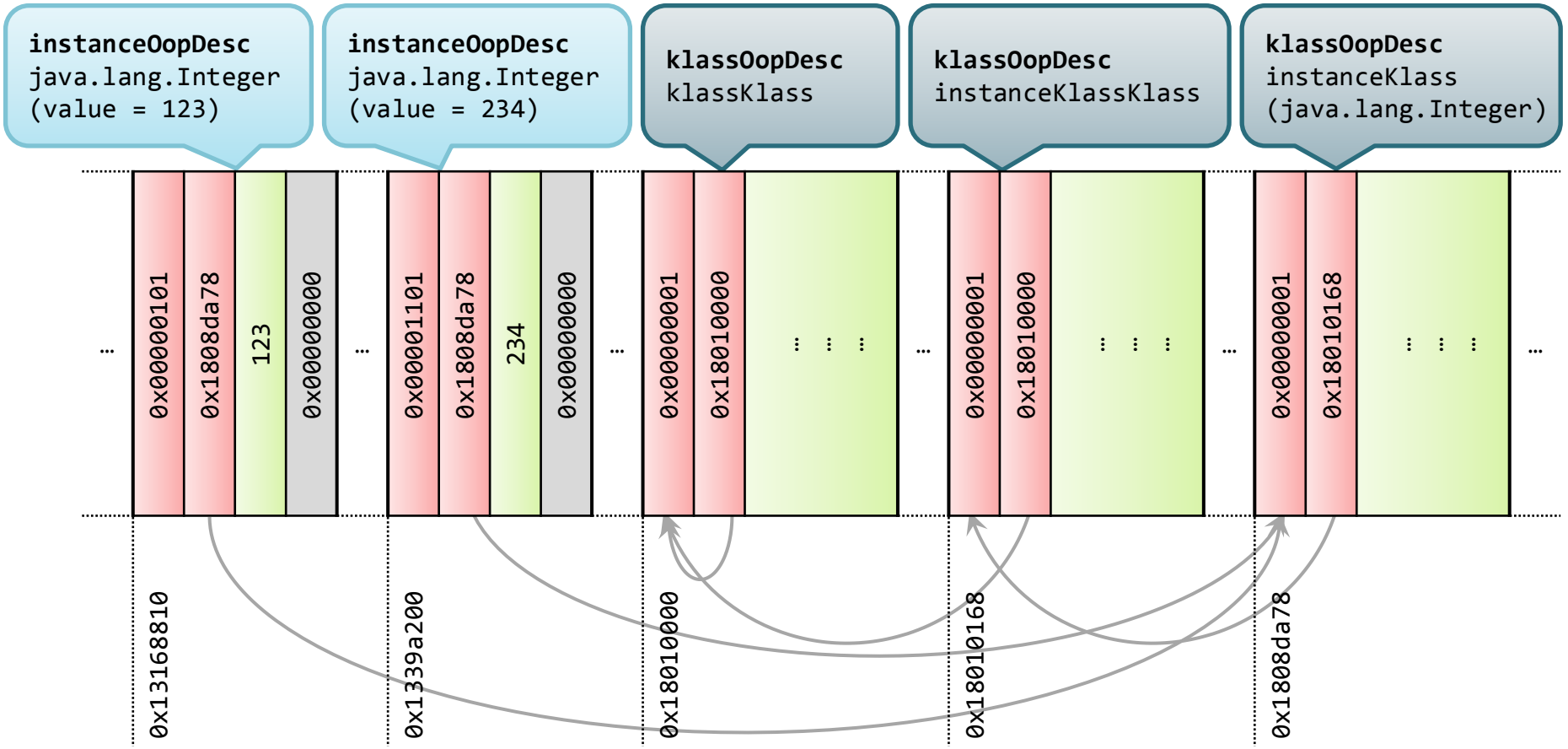

Java对象实例
（instanceOopDesc）
Java数组类型不包含数组长度信息，因而只看数组类型无法确定数组实例的大小，必须在数组实例内嵌入_length字段来记录数组长度
Java数组实例
（objArrayOopDesc
或
typeArrayOopDesc）
对象头
_mark : markOop
_klass : klassOop
对象实际数据
…
instance data
…
对齐填充
（可能存在）
padding
对象头
_mark : markOop
_klass : klassOop
_length : int
数组实际数据
…
array data
…
对齐填充
（可能存在）
padding

java.lang.Integer 32位HotSpot

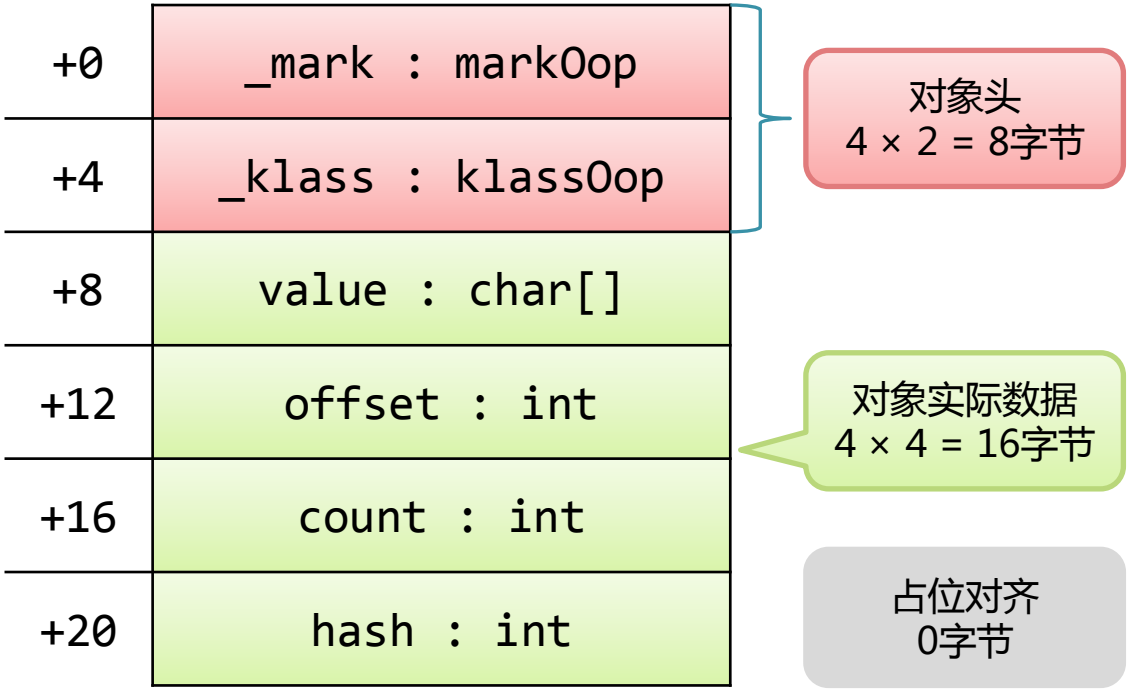



```
public final class Integer
        extends Number
        implements Comparable<Integer> {

    private final int value;
    // ...
}
```


instanceOopDesc
java.lang.Integer
@ 0x13168810
_mark = 0x00000101
_klass = 0x1808da78
value = 123
0x00000000
instanceOopDesc
java.lang.Integer
@ 0x1339a200
_mark = 0x00001101
_klass = 0x1808da78
value = 234
0x00000000
klassOopDesc
instanceKlass
(java.lang.Integer)
@ 0x1808da78
_mark = 0x00000001
_klass = 0x18010168
…
_java_mirror
vtable
itable
…
MAX_VALUE = 0x80000000
MAX_VALUE = 0x7fffffff
⋮
serialVersionUID
= 1360826667806852920L
$assertionsDisabled
= true
0x000000
klassOopDesc
instanceKlassKlass
@ 0x18010168
_mark = 0x00000001
_klass = 0x18010000
⋮
klassOopDesc
klassKlass
@ 0x18010000
_mark = 0x00000001
_klass = 0x18010000
⋮
java.lang.Integer的静态变量（或者说“类变量”）存在这个instanceKlass对象内，就像是这个klass的实例变量一般

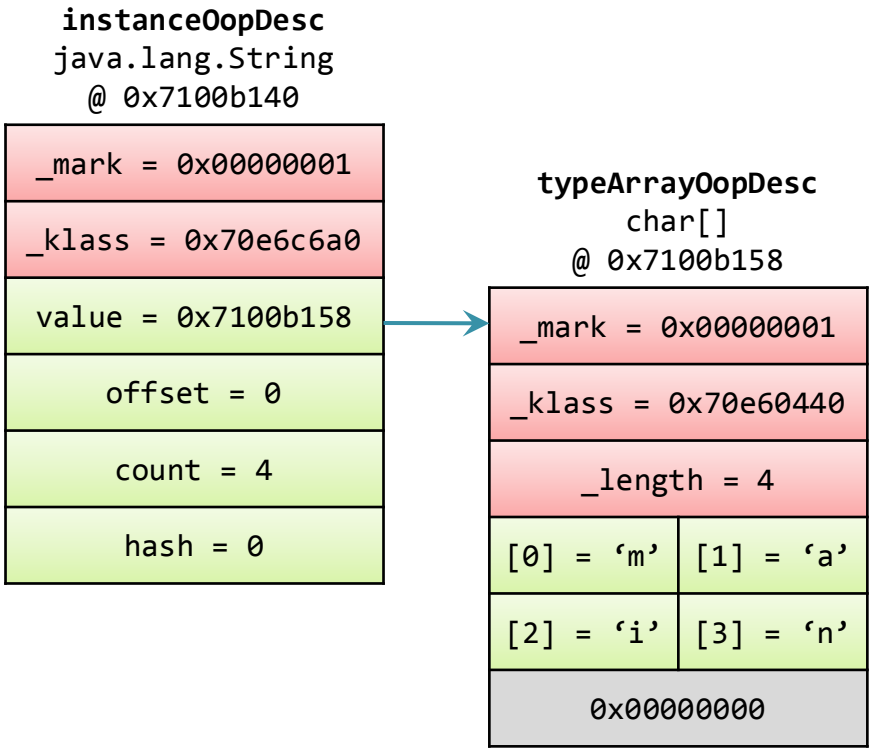

instanceOopDesc
java.lang.Integer
(value = 123)
instanceOopDesc
java.lang.Integer
(value = 234)
klassOopDesc
klassKlass
klassOopDesc
instanceKlassKlass
klassOopDesc
instanceKlass
(java.lang.Integer)
0x00000101
0x1808da78
123
0x00000000
0x00001101
0x1808da78
234
0x00000000
0x00000001
0x18010000
0x00000001
0x18010000
0x00000001
0x18010168
0x13168810
0x1339a200
0x18010000
0x18010168
0x1808da78

java.lang.String 32位HotSpot

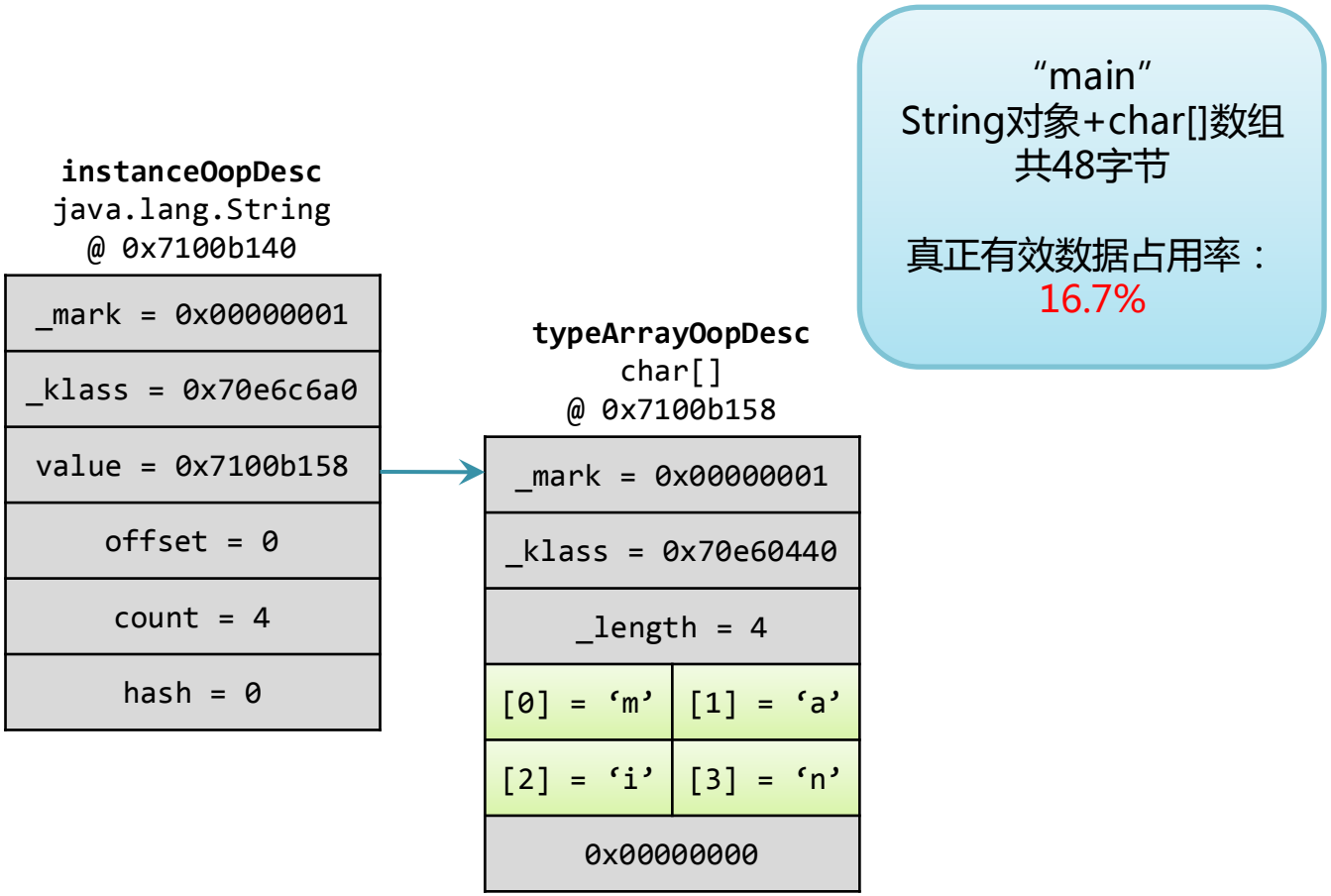



```
public final class String
    implements java.io.Serializable, Comparable<String>, CharSequence
{
    private final char value[];
    private final int offset;
    private final int count;
    private int hash; // Default to 0

    // ...
}
```

java.lang.String “main” 32位HotSpot



java.lang.String “main” 32位HotSpot

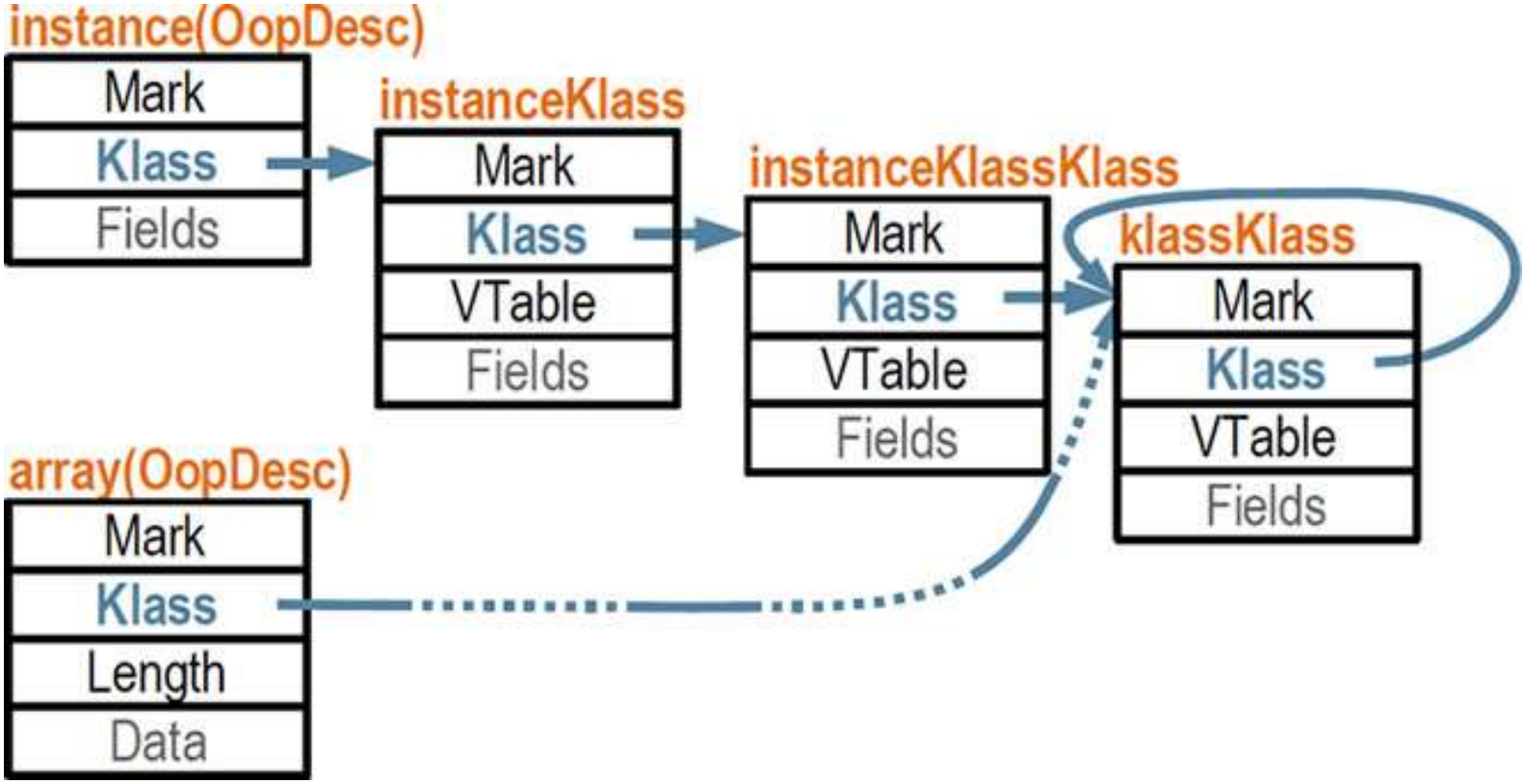


oop

- ordinary object pointer
  - 也有说法是object oriented pointer
- 参考Strongtalk中的oop
- Self -> Strongtalk -> HotSpot

参考资料

oop与klass



参考资料

markOop

Bitfields			Tag	State
Hashcode	Age	0	01	Unlocked
Lock record address			00	Light-weight locked
Monitor address			10	Heavy-weight locked
Forwarding address, etc.			11	Marked for GC
Thread ID	Age	1	01	Biased / biasable

Compressed oop

- -XX:+UseCompressedOops
  - 默认为false
- 在64位HotSpot中使用32位指针
  - 3段式，根据-Xmx/-XX:MaxHeapSize、-XX: HeapBaseMinAddress以及进程环境自动选择：
    - < 4GB
    - ~ 26GB
    - ~ 32GB

```
// Narrow Oop encoding mode:
// 0 - UnscaledNarrowOop
//     - Use 32-bits oops without encoding when
//       HeapBaseMinAddress + heap_size < 4Gb
// 1 - ZeroBasedNarrowOop
//     - Use zero based compressed oops with encoding when
//       HeapBaseMinAddress + heap_size < 32Gb
// 2 - HeapBasedNarrowOop
//     - Use compressed oops with heap base + encoding.
```

参考文档

constantPoolCache

类加载与类加载器

类加载器查找到类的方式

参考资料

类加载与SystemDictionary

- 记录着所有当前已加载的类
  - 记录的映射关系是
    - 从Java层面看：

      ```
      (class name, classloader) => class
      ```

    - 从VM层面看：

      ```
      (symbolOop, oop) => klassOop
      ```

    - 反映出Java/JVM中一个Java类是由它的全限定名与加载它的类加载器共同确定的，而不能只依靠类的全限定名确定

类加载与对象布局

- Java对象布局是在类加载过程中由类加载器计算出来的，布局信息保存在klass对象中

字节码校验器与Class文件格式校验器

split verifier

- JSR 202（Java 6）
- 先在CLDC投入使用

参考资料

老字节码校验器

抽象解释（1）

HotSpot中的Java方法相关对象

- 图！

HotSpot中的Java方法相关对象

- methodOop
  - 方法对象
- methodKlass
  - methodOop的klass对象
- InvocationCounter（×2）
  - 调用与回边计数器
- methodDataOop
  - methodOop的profile信息
- nmethod
  - 由JIT编译器编译得到的结果对象
- ……还有许多关联对象
- methodHandle
  - methodOop的handle
- constMethodOop
  - 一个Java方法中固定不变的信息
    - 代码大小
    - 操作数栈大小
    - 局部变量区大小
    - 参数格式
    - 可访问性
    - 字节码
    - StackMapTable
    - Exceptions
    - LineNumberTable
    - … etc
  - 不包括常量池内容
- constantPoolOop
  - Java方法的常量池

constMethodOop

- First, methodOops in the server VM are larger than client. On 1.4.2 client, methodOops are 108 bytes in size, on server, they are 132 bytes in size. In Tiger, part of the methodOop has been split off into the constantMethodOop, but the total space required for a method grows to 140 and 156 bytes, respectively. We can't "fix" this problem.

参考文章

methodOop的多个入口

- from_interpreted_entry
  - 从解释器调用当前方法
    - （从native调用Java方法也经过该入口）
  - 若当前方法已被标准编译，则经过i2c进入编译后代码
  - 若当前方法目前没有可用的已编译代码，则直接进入解释器
    - 可能是尚未被编译，也可能编译后被逆优化
- from_compiled_entry
  - 从编译后代码调用当前方法
  - 若当前方法已被标准编译，则直接调到编译后代码
  - 若当前方法目前没有可用的已编译代码，则经过c2i进入解释器

适配器栈帧（adapter frame）

- i2c
- c2i
- c2n
- code patching / JIT direct call

调用约定与栈帧适配器

- 适配器用于调整参数位置

参考文章（1）（2）

统一的调用栈

- 同一线程中的Java方法与本地方法共用一个调用栈
  - 同一线程中，解释器中运行的Java方法与编译后的Java方法也一样共用一个调用栈
- 节省了一个栈指针寄存器
  - x86上只需要ESP，不需要额外的寄存器来记录Java求值栈指针
- TODO:其它特征

参考资料

统一的调用栈

- 图！

快速子类型检查（fast subtype-checking）

- Java程序中含有许多需要检查子类型的情况
  - instanceof运算符
    - 对应instanceof字节码
  - 强制类型转换
    - 以及调用返回值是泛型参数的方法
    - 以及访问类型为泛型参数的成员变量
    - 对应checkcast字节码
  - 引用类型数组元素赋值
    - Java的数组是协变的
      - 在引用类型数组元素赋值时类型不安全
      - 需要运行时的类型检查
    - 对应aastore字节码
- 需要能够快速检查子类型关系
  - 尽量不要递归搜索

快速子类型检查（fast subtype-checking）

- 优化方式
  - Java类型的超类型在运行时不可变
  - 一个类在类层次结构的“深度”也不变
  - 所以可以通过“区头向量”（display）来快速验证子类型关系
- （部分情况可以由JIT进一步削除）

快速子类型检查（fast subtype-checking）

- klass中存的相关数据

```
class Klass : public Klass_vtbl {
 protected:
  enum { _primary_super_limit = 8 };

  // Where to look to observe a supertype (it is &_secondary_super_cache for
  // secondary supers, else is &_primary_supers[depth()].
  juint       _super_check_offset;

 protected:
  // Cache of last observed secondary supertype
  klassOop    _secondary_super_cache;
  // Array of all secondary supertypes
  objArrayOop _secondary_supers;
  // Ordered list of all primary supertypes
  klassOop    _primary_supers[_primary_super_limit];
};
```

快速子类型检查（fast subtype-checking）

- 原本的优化算法

```
S.is_subtype_of(T) := {
  int off = T.offset;
  if (T == S[off])    return true;
  if (off != &cache)  return false;
  if (S == T)         return true;
  if ( S.scan_secondary_subtype_array(T) ) {
    S.cache = T;
    return true;
  }
  return false;
}
```

参考资料

快速子类型检查（fast subtype-checking）

- 在Java 6中实现的算法

```
S.is_subtype_of(T) := {
  int off = T.offset;
  if (S == T)          return true;
  if (S[off] == T)     return true;
  if (off != &cache)   return false;
  if ( S.scan_secondary_subtype_array(T) ) {
    S.cache = T;
    return true;
  }
  return false;
}
```

快速子类型检查（fast subtype-checking）

- 图！Subtype lattice!

HotSpot里的线程

- Java应用线程
- 系统线程
  - VMThread
  - Finalizer
  - ReferenceHandler
  - CompilerThread
  - WatcherThread
  - Signal Dispatcher
  - Attach Listener
  - Low Memory Detector
  - Surrogate Locker Thread (CMS)
  - ConcurrentMarkSweepThread
  - GC Daemon
  - DestroyJavaVM
  - JDWP Event Helper Thread
  - JDWP Transport Listener
  - ...等

```
// Class hierarchy
// - Thread
//   - NamedThread
//     - VMThread
//     - WorkerThread
//       - GangWorker
//       - GCTaskThread
//     - ConcurrentGCThread
//       - ConcurrentMarkSweepThread
//   - WatcherThread
//   - JavaThread
//     - CompilerThread
//     - SurrogateLockerThread
//     - JvmtiAgentThread
//   - ServiceThread
```

HotSpot里的线程

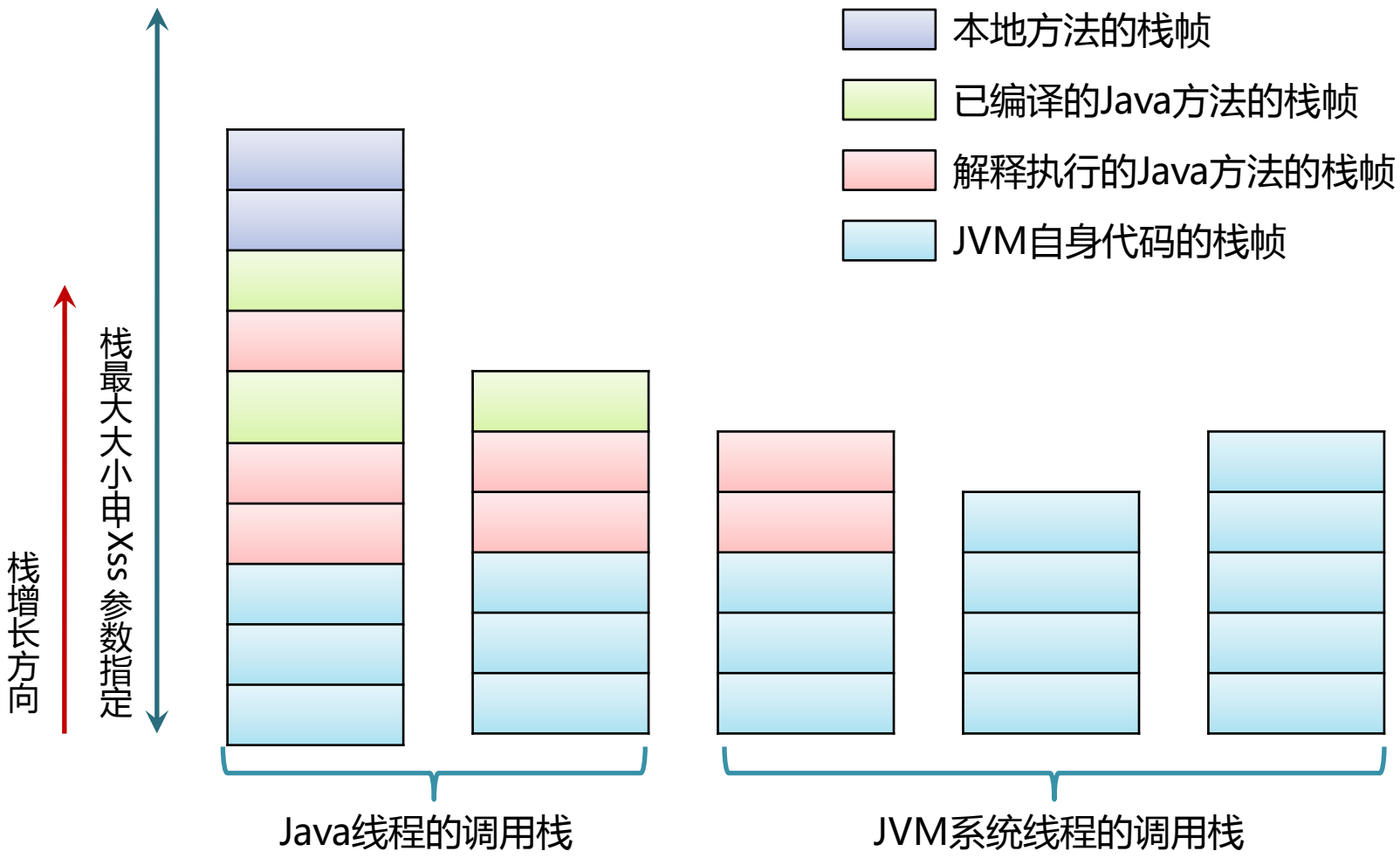
- Java应用线程
- 系统线程
  - VMThread
  - Finalizer
  - ReferenceHandler
  - CompilerThread
  - WatcherThread
  - Signal Dispatcher
  - Attach Listener
  - Low Memory Detector
  - Surrogate Locker Thread (CMS)
  - ConcurrentMarkSweepThread
  - GC Daemon
  - DestroyJavaVM
  - JDWP Event Helper Thread
  - JDWP Transport Listener
  - ...等

这两个线程是用Java代码实现的

HotSpot里的线程

- 实验（1）：
  - jstack <pid>
  - 观察运行在HotSpot上的Java进程的线程状况，及其对应的栈
- 实验（2）：
  - jstack -m <pid> | c++filt
  - （Linux/Solaris）观察HotSpot中的Java线程中，Java栈帧与本地栈帧是如何交错出现的

HotSpot里的线程



参考资料：图8

安全的停止线程——safepoint

参考资料(1) (2)

Lightweight Locking

Biased Locking

类数据共享（class data sharing)

- TODO：写点什么？
  - 从Java SE 5.0开始引入
  - TODO！
  - -Xshare:auto | on | off | dump
- IBM SDK for Java v6的类数据共享更加激进

参考资料（1）（2）

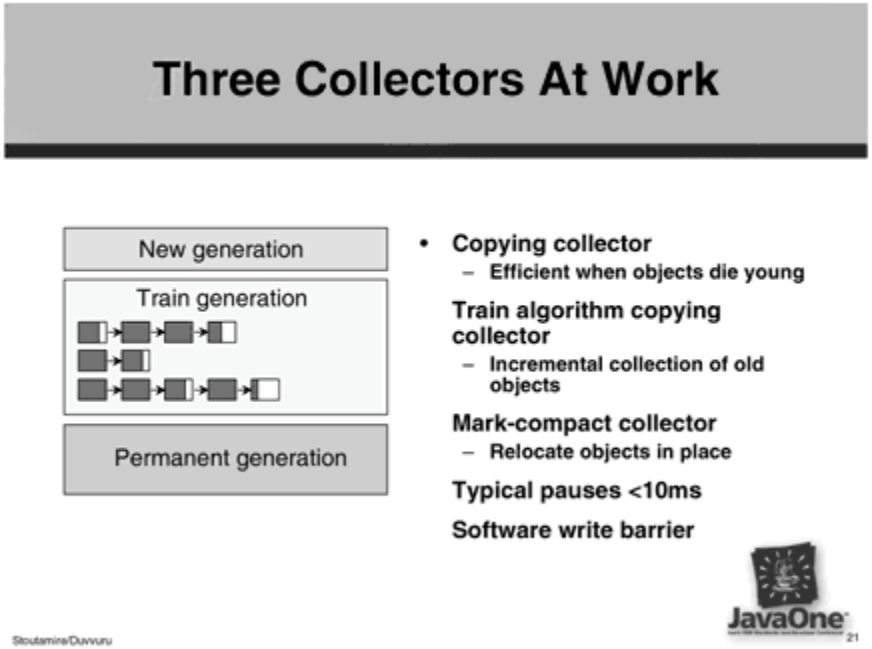
HOTSPOT VM的内存管理

分代式收集的堆

- David Ungar, Generation Scavenging: A Non-disruptive High Performance Storage Reclamation Algorithm (1984)
  - 奠定了HotSpot的GC堆的基本布局
    - HotSpot的GC当中，只有G1不使用该布局
  - 衍生自Berkeley Smalltalk -> Self -> Strongtalk VM
  - 该论文2008年获得Retrospective ACM SIGSOFT Impact Paper Award

老版本HotSpot VM用过的GC

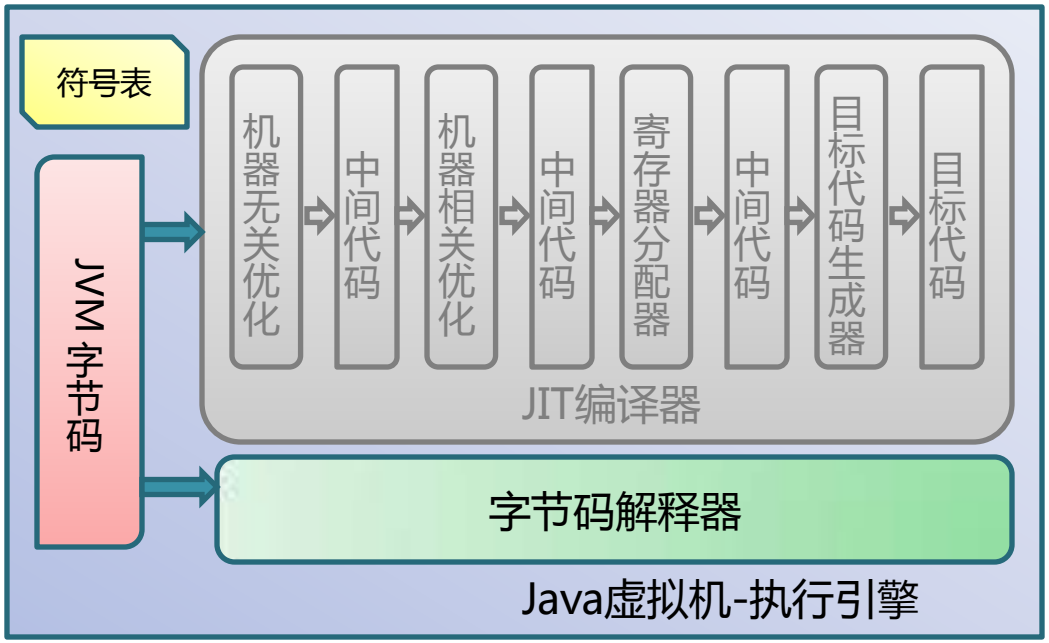
- 老的HotSpot VM曾经提供过-Xincgc参数，用于指定使用incremental “train” collection



GCHisto

- http://java.net/projects/gchisto/

HOTSPOT与解释器



实现高效解释器的一种思路

存储程序计算机的核心循环——FDX

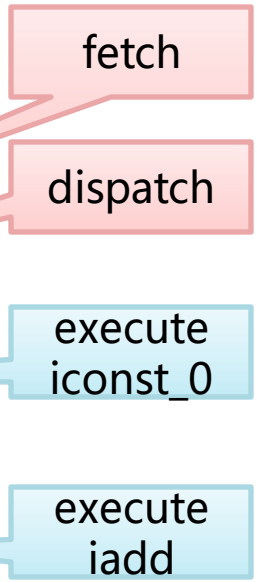
- F: fetch
  - 获取下一条指令，并让指令计数器加一
- D: decode
  - 对指令解码
  - 读取指令的操作数
- (D: dispatch)
  - 分派到具体的执行单元去处理
- X: execute
  - 执行指令的逻辑

参考Wikipedia

实现简单的解释器貌似很容易

FDX循环的直观实现——switch threading

```
#define FETCH_OPCODE (opcode = bytecode[pc++])
#define PUSH(val)    (*++sp = (val))
#define POP          (*sp--)

int execute() {
    // ... declarations
    while (true) {
        FETCH_OPCODE;
        switch (opcode) {
        case ICONST_0:
            PUSH(0);
            break;
        case IADD:
            y = POP;
            x = POP;
            PUSH(x + y);
            break;
        // ...
        }
    }
}
```



Sun JDK 1.0系的JVM的
解释器采用这种方式实现

FDX循环的直观实现——switch threading

- 内存占用量：低
- 启动性能：高
- 稳定状态性能：低

- 每轮FDX循环有多个直接/间接跳转
- 字节码指令在数据缓存（d-cache）中，而没有利用上CPU为处理指令而设计的指令缓存（i-cache）
- 可移植性：这种方式容易用高级语言实现，所以容易达到高可移植性

token-threading

```
#define FETCH_OPCODE (opcode = bytecode[pc++])
#define DISPATCH     goto *handlers[opcode]
#define PUSH(val)    (*++sp = (val))
#define POP          (*sp--)

int execute() {
    // ... declarations
    ICONST_0: { // first-class label
        PUSH(0);
        FETCH_OPCODE;
        DISPATCH;
    }
    IADD: {
        y = POP;
        x = POP;
        PUSH(x + y);
        FETCH_OPCODE;
        DISPATCH;
    }
}
```

token-threading

- 内存占用量比switch方式稍多（fetch与dispatch代码冗余）
- 启动性能：基本上与switch方式一样
- 稳定状态性能：比switch方式稍好
  - 减少了跳转的次数
  - 跳转比switch方式的容易预测一些
- 可移植性：易用C实现，可达到高可移植

参考资料

通过汇编精确控制寄存器的使用

栈顶缓存（top-of-stack caching）

参考资料

操作数栈的栈顶状态（TosState）

- atos – Object, Array（TOS在EAX）
- btos – byte, boolean（TOS在EAX）
- ctos – char（TOS在EAX）
- stos – short（TOS在EAX）
- itos – int（TOS在EAX）
- ltos – long（TOS在EDX/EAX）
- ftos – float
- dtos – double
- vtos – void（栈顶没有缓存在寄存器中）

基于模板的解释器（template based interpreter）

- HotSpot的解释器不是直接用汇编写的
- HotSpot内部有一个汇编库，用于在运行时生成机器码
- 通过其中的InterpreterMacroAssembler，HotSpot在启动时根据预置的汇编模板、VM启动参数以及硬件条件来生成解释器
  - x86通用寄存器很少，在解释器的核心循环需要人工精确调优
  - 栈顶值缓存在EAX或EDX/EAX对中，大大减少操作数栈操作开销
  - 达到了近似条件编译的效果，只在需要解释器的某些功能时在为其生成代码，如debug、profile、trace等
  - 可以最大限度利用机器提供的新指令集，同时兼容老指令集
  - 方便VM将函数或变量的地址、偏移量等信息直接写入解释器中
- 这样可以既可以精确控制生成的机器码，又不会受不同汇编器的语法和功能的差异的困扰，减少了外部依赖

参考论文： Generation of Virtual Machine Code at Startup

一个简单的指令模板——iconst

指定该指令预期的
进入TOS状态
与离开的TOS状态

根据条件生成
实际处理逻辑

```
void TemplateTable::iconst(int value) {
  transition(vtos, itos);
  if (value == 0) {
    __ xorptr(rax, rax);
  } else {
    __ movptr(rax, value);
  }
}
```

所谓指令模板，其实就是普通的C++方法而已

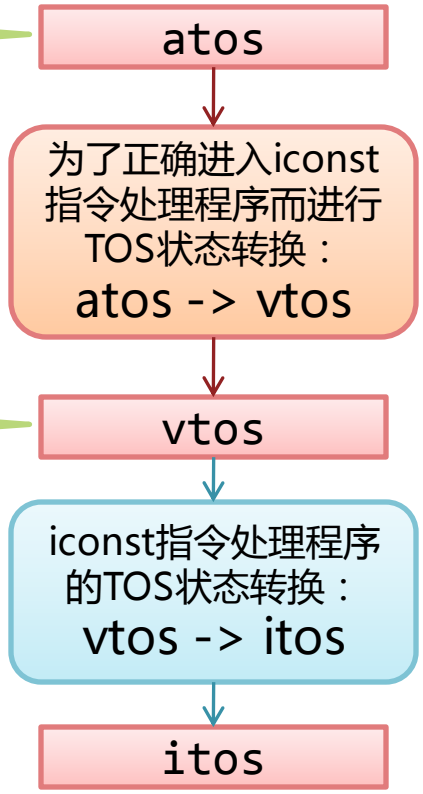
一个简单的指令模板——iconst

```
transition(vtos, itos);
```

如果前一条指令结束时TOS状态是**atos**，则从这里进入iconst的处理

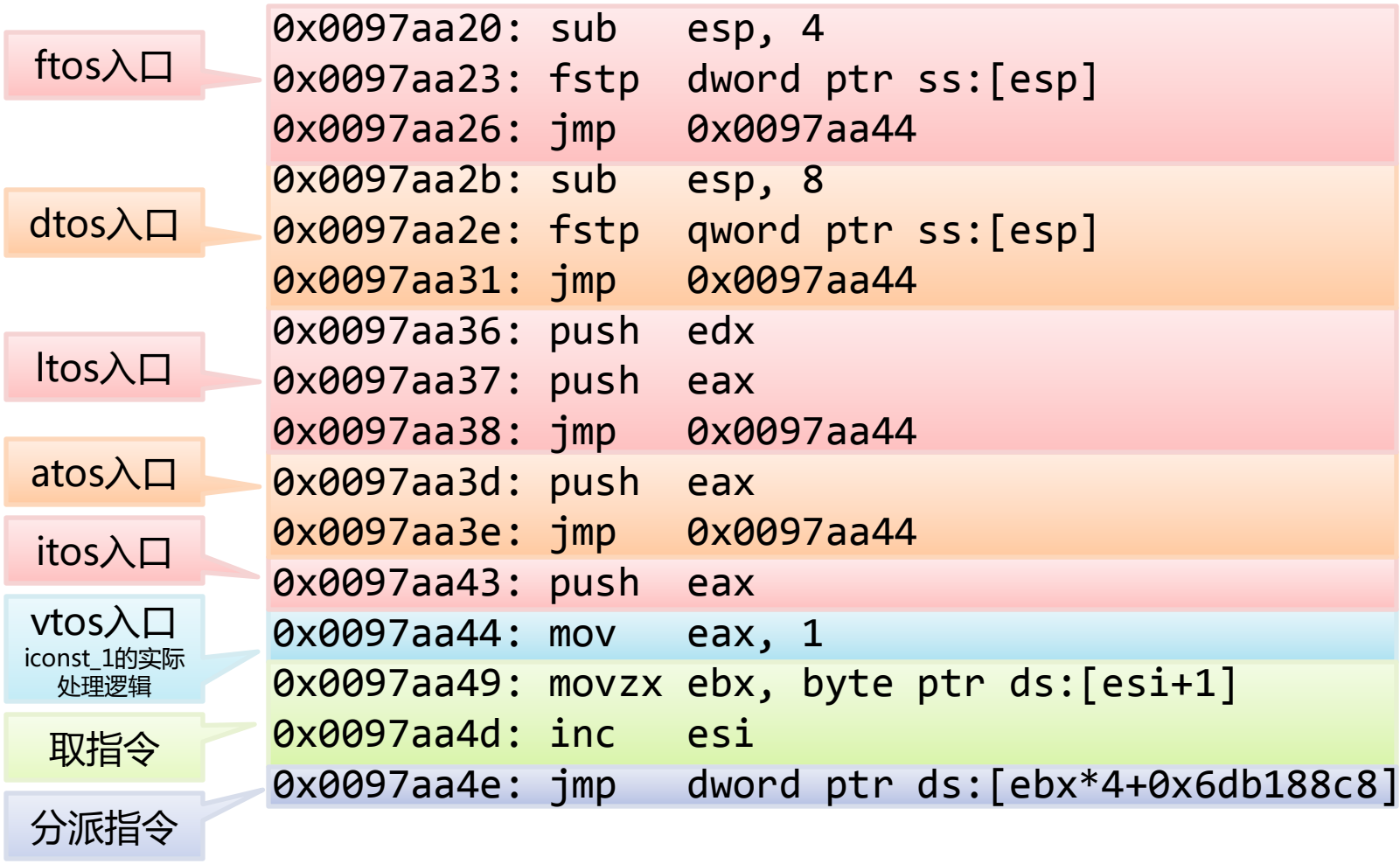
如果前一条指令结束时TOS状态是**vtos**，则从这里进入iconst的处理

其它指令、其它进入TOS状态的情况依次类推



iconst_1在client模式默认生成的代码

```
def(Bytecodes::_iconst_1, ____|____|____|____, vtos, itos, iconst, 1);
```



超级指令（superinstruction）

- 将多条字节码指令融合为一条
- 节省取指令和指令分派的开销
- 解释器占用的空间会稍微增加
- 在HotSpot解释器中有所应用
  - 由-XX:+RewriteFrequentPairs参数控制
    - client模式默认关闭，server模式默认开启
  - 某些模式的字节码被首次执行时会被改写为对应的超级指令
    - 例如将两个连续的iload改写为一个fast_iload2
  - 后续执行则使用超级指令版本处理

参考论文： Superinstructions and replication in the Cacao JVM interpreter

HotSpot中一条超级指令——fast_iload2

第一条iload的处理

第二条iload的处理

```
void TemplateTable::fast_iload2() {
  transition(vtos, itos);
  locals_index(rbx);
  __ movl(rax, iaddress(rbx));
  debug_only(__ verify_local_tag(frame::TagValue, rbx));
  __ push(itos);
  locals_index(rbx, 3);
  __ movl(rax, iaddress(rbx));
  debug_only(__ verify_local_tag(frame::TagValue, rbx));
}
```

两条iload指令之间的取指令与分派指令的开销被削除了

HotSpot的解释器与safepoint

- 在正常执行状态中的解释器不包含检查safepoint的代码，减少执行开销
- 需要进入safepoint的时候，解释器的指令分派表（dispatchTable）会从正常版切换到safepoint版
  - 也就是说解释器中有两套dispatchTable
  - safepoint版dispatchTable指向的指令处理程序有检查safepoint的逻辑
- safepoint过后再切换回正常版

解释器到编译器构成一个连续体

从解释器到编译器

纯解释器

- 源码级解释器
- 树遍历解释器
- 虚拟指令（字节码）解释器
  - switch-threading
  - indirect-threading
  - token-threading
  - direct-threading
  - subroutine-threading
  - inline-threading
  - context-threading
  - …

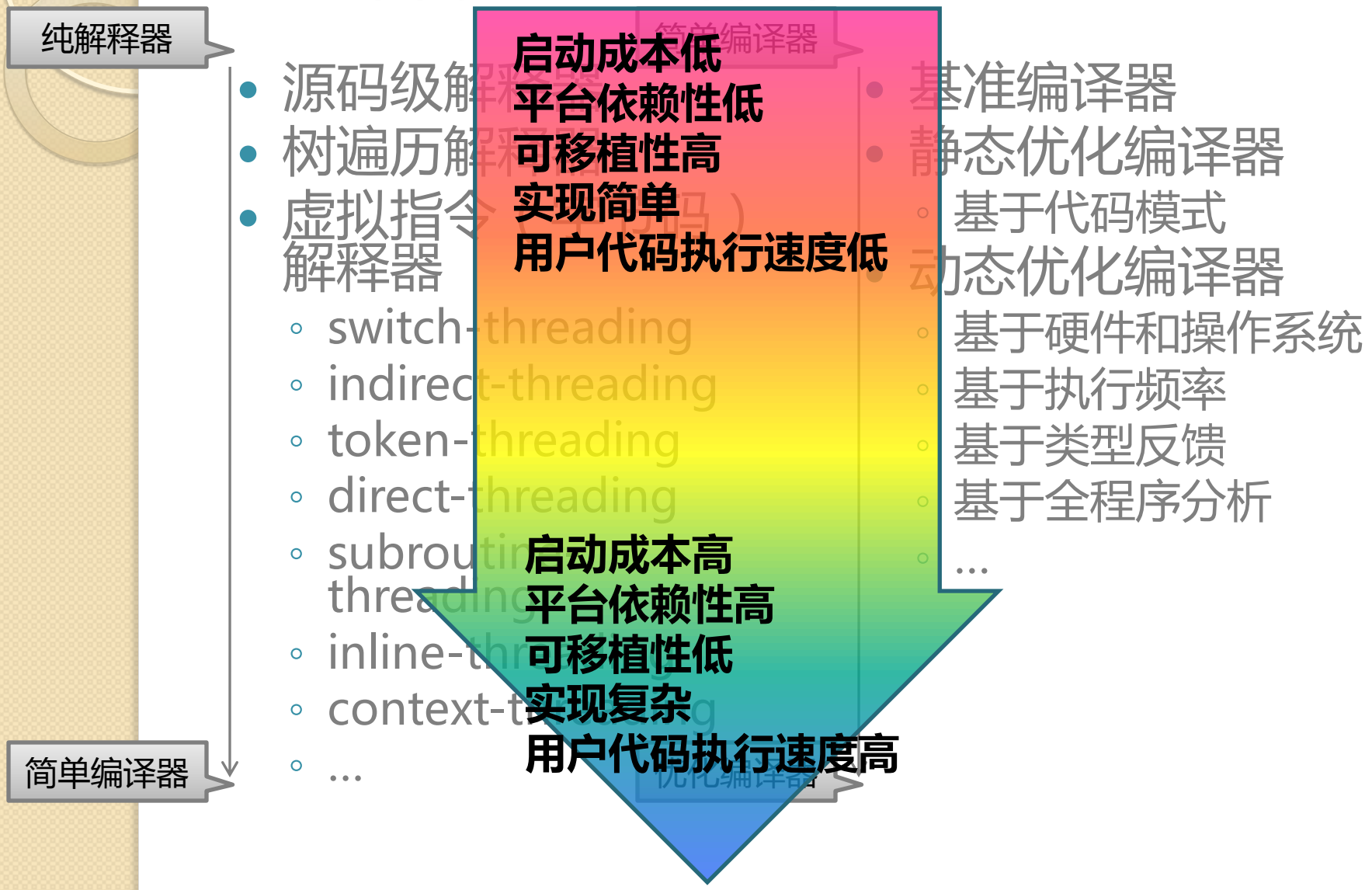
简单编译器

简单编译器

- 基准编译器
- 静态优化编译器
  - 基于代码模式
- 动态优化编译器
  - 基于硬件和操作系统
  - 基于执行频率
  - 基于类型反馈
  - 基于整程序分析
  - …

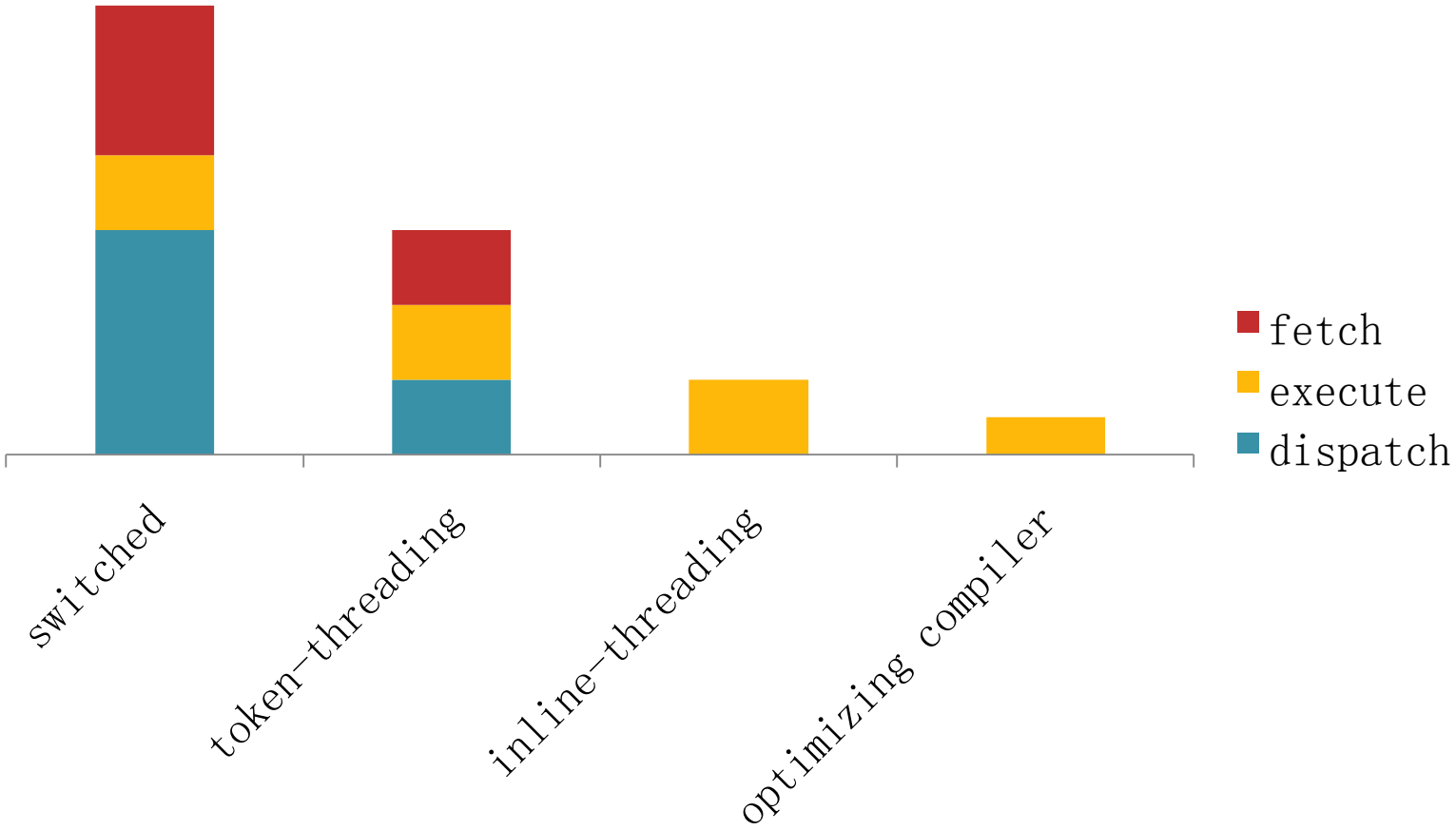
优化编译器

从解释器到编译器



- 源码级解释器
- 树遍历解释器
- 虚拟指令（字节码）解释器
  - switch-threading
  - indirect-threading
  - token-threading
  - direct-threading
  - subroutine-threading
  - inline-threading
  - context-threading
  - …
- 基准编译器
- 静态优化编译器
  - 基于代码模式
- 动态优化编译器
  - 基于硬件和操作系统
  - 基于执行频率
  - 基于类型反馈
  - 基于全程序分析
  - …

解释执行的开销


fetch
execute
dispatch
switched
token-threading
inline-threading
optimizing compiler

解释执行的开销（例）

Java源代码：

```
public class TOSDemo {
    public static int test() {
        int i = 31;
        int j = 42;
        int x = i + j;
        return x;
    }
}
```

由javac编译

JVM字节码：

```
iload_0
iload_1
iadd
istore_2
```

执行？

```
iload_0
iload_1
iadd
istore_2
```

由Sun JDK 1.0.2的
JVM解释执行
经过的x86指令序列

```
;;------------iload_0------------
mov ecx,dword ptr ss:[esp+1C]
mov edx,dword ptr ss:[esp+14]
add dword ptr ss:[esp+14],4
mov eax,dword ptr ds:[ecx]
inc dword ptr ss:[esp+10]
mov dword ptr ds:[edx+10],eax
jmp javai.1000E182
mov eax,dword ptr ss:[esp+10]
mov bl,byte ptr ds:[eax]
xor eax,eax
mov al,bl
cmp eax,0FF
ja  short javai.1000E130
jmp dword ptr
;;------------
mov ecx,dword
mov edx,dword
add dword ptr
mov eax,dword
inc dword ptr
mov dword ptr
jmp javai.1000
mov eax,dword
mov bl,byte pt
xor eax,eax
mov al,bl
cmp eax,0FF
ja  short java
jmp dword ptr
```

```
;;-------------iadd--------------
mov ecx,dword ptr ss:[esp+14]
inc dword ptr ss:[esp+10]
sub dword ptr ss:[esp+14],4
mov edx,dword ptr ds:[ecx+C]
add dword ptr ds:[ecx+8],edx
jmp javai.1000E182
mov eax,dword ptr ss:[esp+10]
mov bl,byte ptr ds:[eax]
xor eax,eax
mov al,bl
cmp eax,0FF
ja  short javai.1000E130
jmp dword ptr ds:[eax*4+10011B54]
;;-----------istore_2------------
mov eax,dword ptr ss:[esp+14]
mov ecx,dword ptr ss:[esp+1C]
sub dword ptr ss:[esp+14],4
mov edx,dword ptr ds:[eax+C]
inc dword ptr ss:[esp+10]
mov dword ptr ds:[ecx+8],edx
jmp javai.1000E182
mov eax,dword ptr ss:[esp+10]
mov bl,byte ptr ds:[eax]
xor eax,eax
mov al,bl
cmp eax,0FF
ja  short javai.1000E130
jmp dword ptr ds:[eax*4+10011B54]
```

由Sun JDK 1.1.8的
JVM解释执行
经过的x86指令序列

```
;;-------------iload_0-------------
movzx eax,byte ptr ds:[esi+1]
mov   ebx,dword ptr ss:[ebp]
inc   esi
jmp   dword ptr ds:[eax*4+1003FBD4]
;;-------------iload_1-------------
movzx eax,byte ptr ds:[esi+1]
mov   ecx,dword ptr ss:[ebp+4]
inc   esi
jmp   dword ptr ds:[eax*4+1003FFD4]
;;--------------iadd---------------
add ebx,ecx
movzx eax,byte ptr ds:[esi+1]
inc   esi
jmp   dword ptr ds:[eax*4+1003FBD4]
;;------------istore_2-------------
movzx eax,byte ptr ds:[esi+1]
mov   dword ptr ss:[ebp+8],ebx
inc   esi
jmp   dword ptr ds:[eax*4+1003F7D4]
```

由Sun JDK 6u18的
HotSpot解释执行
经过的x86指令序列

```
;;-------------iload_0-------------
mov   eax,dword ptr ds:[edi]
movzx ebx,byte ptr ds:[esi+1]
inc   esi
jmp   dword ptr ds:[ebx*4+6DB188C8]
;;-------------iload_1-------------
push  eax
mov   eax,dword ptr ds:[edi-4]
movzx ebx,byte ptr ds:[esi+1]
inc   esi
jmp   dword ptr ds:[ebx*4+6DB188C8]
;;--------------iadd---------------
pop   edx
add   eax,edx
movzx ebx,byte ptr ds:[esi+1]
inc   esi
jmp   dword ptr ds:[ebx*4+6DB188C8]
;;------------istore_2-------------
mov   dword ptr ds:[edi-8],eax
movzx ebx,byte ptr ds:[esi+1]
inc   esi
jmp   dword ptr ds:[ebx*4+6DB19CC8]
```

例：指令序列（instruction traces）

解释执行的开销（例）

```
iload_0
iload_1
iadd
istore_2
```

由HotSpot解释执行

```
;;-------------iload_0------------
mov   eax,dword ptr ds:[edi]
movzx ebx,byte ptr ds:[esi+1]
inc   esi
jmp   dword ptr ds:[ebx*4+6DB188C8]
;;-------------iload_1------------
push  eax
mov   eax,dword ptr ds:[edi-4]
movzx ebx,byte ptr ds:[esi+1]
inc   esi
jmp   dword ptr ds:[ebx*4+6DB188C8]
;;--------------iadd--------------
pop   edx
add   eax,edx
movzx ebx,byte ptr ds:[esi+1]
inc   esi
jmp   dword ptr ds:[ebx*4+6DB188C8]
;;------------istore_2------------
mov   dword ptr ds:[edi-8],eax
movzx ebx,byte ptr ds:[esi+1]
inc   esi
jmp   dword ptr ds:[ebx*4+6DB19CC8]
```

假想：
消除指令分派开销
（inline-threading
可达到此效果）

```
;;-------------iload_0------------
mov   eax,dword ptr ds:[edi]
;;-------------iload_1------------
push  eax
mov   eax,dword ptr ds:[edi-4]
;;--------------iadd--------------
pop   edx
add   eax,edx
;;------------istore_2------------
mov   dword ptr ds:[edi-8],eax
```

假想：
进行常量传播/折叠、
寄存器分配等优化

```
mov  eax,ecx
```

inline-threading

- 除了HotSpot的解释器用的token-threading外，还有许多种“threading”
  - 由于HotSpot本身含有优化JIT编译器，解释器就不追求高的稳定执行速度，而追求高启动速度、低内存占用，因而不采用direct-threading、subroutine-threading或inline-threading等解释器优化技巧
- 其中，inline-threading/context-threading都通过复制、融合字节码的处理程序来消除字节码指令分派的开销
  - 也被称为code-copying JIT compiler
- 性能与简单的JIT编译器相似
  - 都消除了软件实现取指令与指令分派的开销
- SableVM、JamVM等一些JVM使用了该技巧
- 诸如Nitro、JaegerMonkey等JavaScript引擎也使用了该技巧

使用解释器的好处

- 启动速度快
- 启动阶段占用内存空间小

- 保持程序的高响应度
  - JIT编译可以在后台进行，同时解释器在前台继续执行程序，不会因编译而停顿
  - 这样也为JIT编译提供了更多时间，使其不必过分为了追求生成代码的速度而放弃生成的代码的质量

- 易于处理带状态的指令
  - 触发类加载
  - 触发类初始化
    - getstatic / putstatic / invokestatic / new 等

- 易于跟踪和收集程序行为（trace & profile）

HotSpot解释器的栈帧布局

```
// A frame represents a physical stack frame (an activation).  Frames can be
// C or Java frames, and the Java frames can be interpreted or compiled.
// In contrast, vframes represent source-level activations, so that one physical frame
// can correspond to multiple source level frames because of inlining.
// A frame is comprised of {pc, fp, sp}
// ------------------------------ Asm interpreter ----------------------------------------
// Layout of asm interpreter frame:
//    [expression stack      ] * <- sp
//    [monitors              ]   \
//     ...                        | monitor block size
//    [monitors              ]   /
//    [monitor block size    ]
//    [byte code index/pointr]                  = bcx()                bcx_offset
//    [pointer to locals     ]                  = locals()             locals_offset
//    [constant pool cache   ]                  = cache()              cache_offset
//    [methodData            ]                  = mdp()                mdx_offset
//    [methodOop             ]                  = method()             method_offset
//    [last sp               ]                  = last_sp()            last_sp_offset
//    [old stack pointer     ]                    (sender_sp)          sender_sp_offset
//    [old frame pointer     ]   <- fp          = link()
//    [return pc             ]
//    [oop temp              ]                    (only for native calls)
//    [locals and parameters ]
//                               <- sender sp
// ------------------------------ Asm interpreter ----------------------------------------
```

HotSpot解释器的栈帧布局

```
//-----------------------------------------------------------------------------
// Entry points
//
// Here we generate the various kind of entries into the interpreter.
// The two main entry type are generic bytecode methods and native call method.
// These both come in synchronized and non-synchronized versions but the
// frame layout they create is very similar. The other method entry
// types are really just special purpose entries that are really entry
// and interpretation all in one. These are for trivial methods like
// accessor, empty, or special math methods.
//
// When control flow reaches any of the entry types for the interpreter
// the following holds ->
//
// Arguments:
//
// rbx,: methodOop
// rcx: receiver
//
//
// Stack layout immediately at entry
//
// [ return address     ] <--- rsp
// [ parameter n        ]
//   ...
// [ parameter 1        ]
// [ expression stack   ] (caller's java expression stack)
```

```
// Assuming that we don't go to one of the trivial specialized
// entries the stack will look like below when we are ready to execute
// the first bytecode (or call the native routine). The register usage
// will be as the template based interpreter expects (see interpreter_x86.hpp).
//
// local variables follow incoming parameters immediately; i.e.
// the return address is moved to the end of the locals).
//
// [ monitor entry      ] <--- rsp
//   ...
// [ monitor entry      ]
// [ expr. stack bottom ]
// [ saved rsi          ]
// [ current rdi        ]
// [ methodOop          ]
// [ saved rbp,          ] <--- rbp,
// [ return address     ]
// [ local variable m   ]
//   ...
// [ local variable 1   ]
// [ parameter n        ]
//   ...
// [ parameter 1        ] <--- rdi
```

概念上与HotSpot解释器中栈帧的关系

- 栈帧布局/传参方式

参考文章

HotSpot解释器中的方法调用

解释器中的invokestatic

解释器中的invokespecial

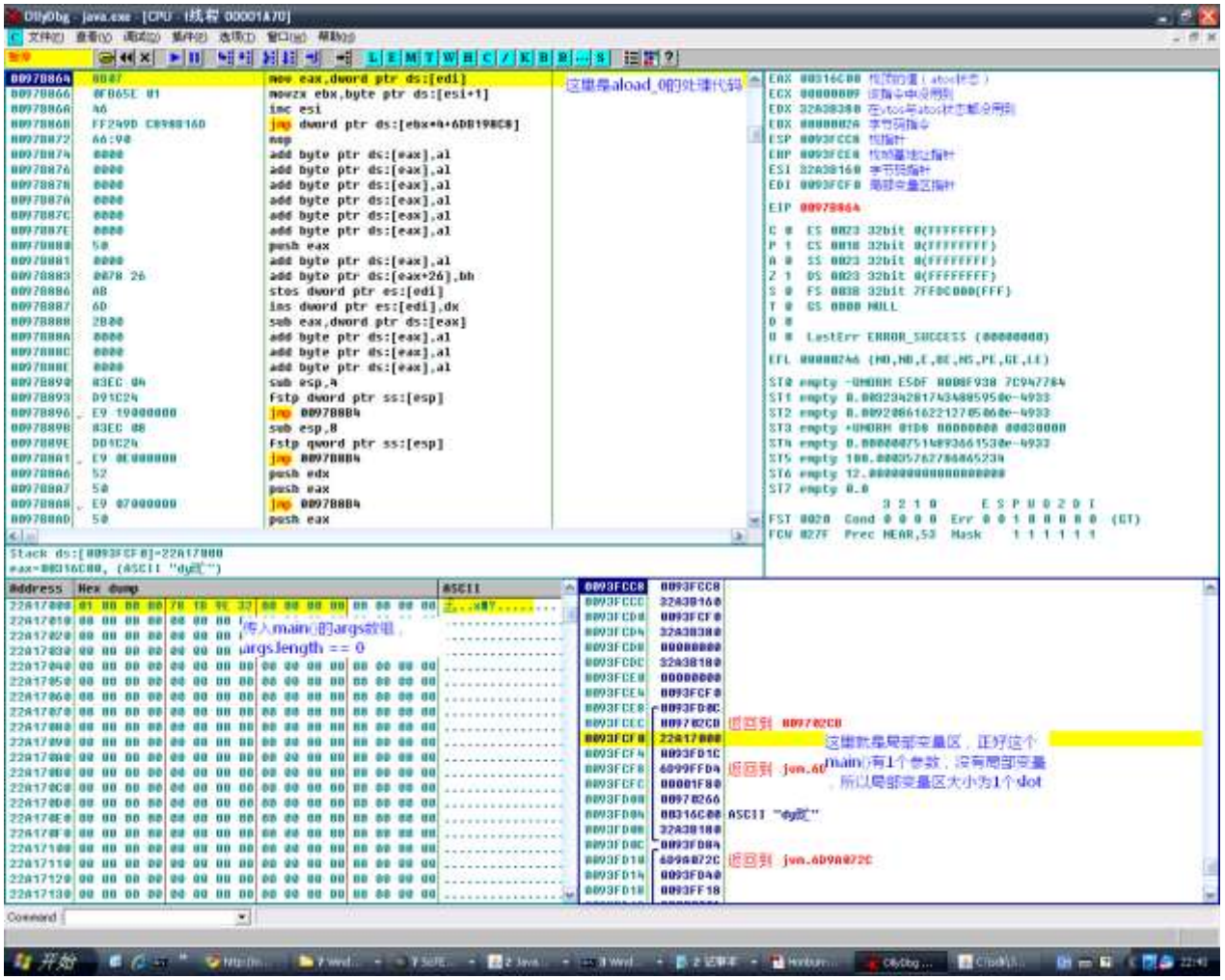
解释器中的invokevirtual

- 3次间接
  - 从对象找到klass
  - 从klass找到vtable里的methodOop
  - 从methodOop里找到 from_interpreted_entry
- final的处理！

解释器中的invokeinterface

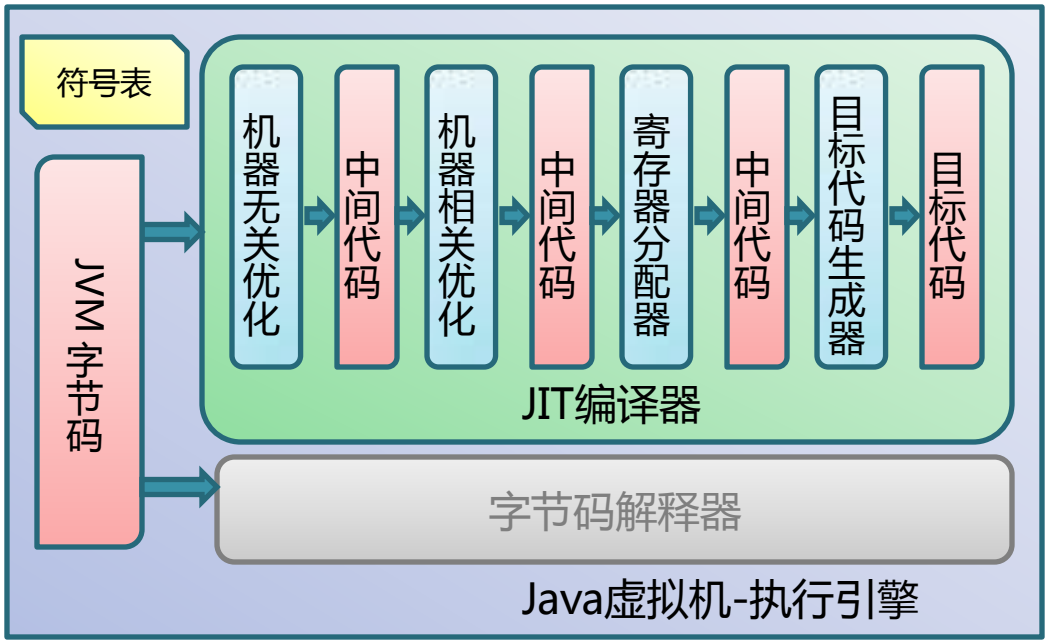
演示：
HotSpot interpreter in action!

HotSpot interpreter in action!



更多图片

HOTSPOT与JIT编译器



JVM的常见优化

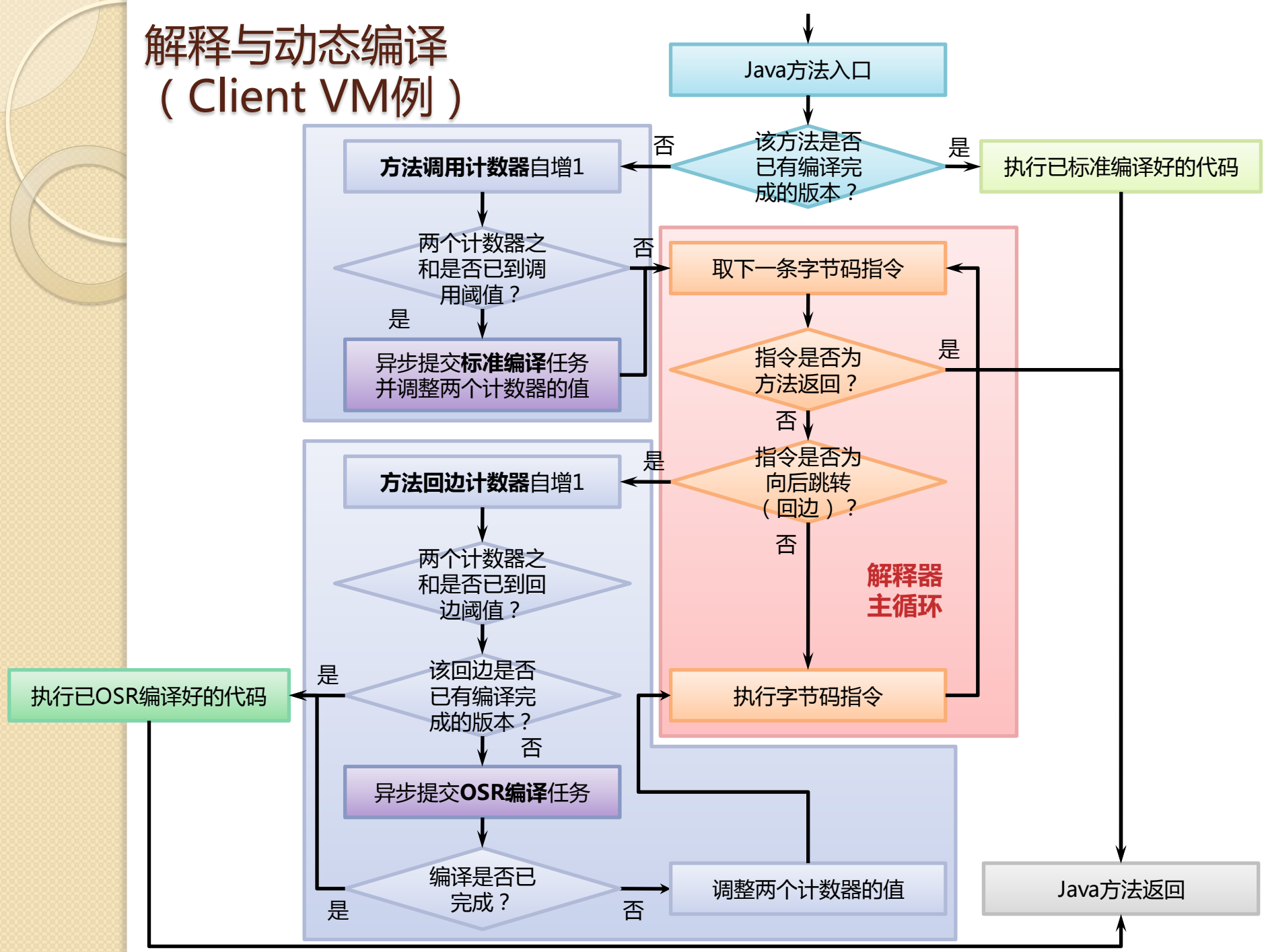
- compiler tactics
  - delayed compilation
  - tiered compilation
  - on-stack replacement
  - delayed reoptimization
  - program dependence graph representation
  - static single assignment representation
- proof-based techniques
  - exact type inference
  - memory value inference
  - memory value tracking
  - constant folding
  - reassociation
  - operator strength reduction
  - null check elimination
  - type test strength reduction
  - type test elimination
  - algebraic simplification
  - common subexpression elimination
  - integer range typing
- flow-sensitive rewrites
  - conditional constant propagation
  - dominating test detection
  - flow-carried type narrowing
  - dead code elimination
- language-specific techniques
  - class hierarchy analysis
  - devirtualization
  - symbolic constant propagation
  - autobox elimination
  - escape analysis
  - lock elision
  - lock fusion
  - de-reflection
- speculative (profile-based) techniques
  - optimistic nullness assertions
  - optimistic type assertions
  - optimistic type strengthening
  - optimistic array length strengthening
  - untaken branch pruning
  - optimistic N-morphic inlining
  - branch frequency prediction
  - call frequency prediction
- memory and placement transformation
  - expression hoisting
  - expression sinking
  - redundant store elimination
  - adjacent store fusion
  - card-mark elimination
  - merge-point splitting
- loop transformations
  - loop unrolling
  - loop peeling
  - safepoint elimination
  - iteration range splitting
  - range check elimination
  - loop vectorization
- global code shaping
  - inlining (graph integration)
  - global code motion
  - heat-based code layout
  - switch balancing
  - throw inlining
- control flow graph transformation
  - local code scheduling
  - local code bundling
  - delay slot filling
  - graph-coloring register allocation
  - linear scan register allocation
  - live range splitting
  - copy coalescing
  - constant splitting
  - copy removal
  - address mode matching
  - instruction peepholing
  - DFA-based code generator

参考资料

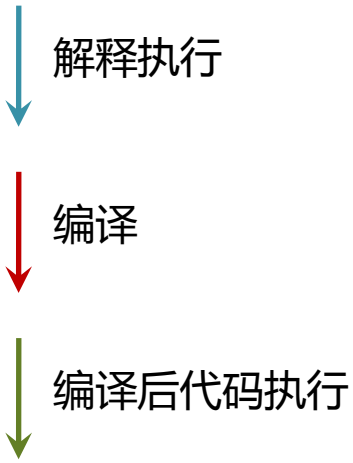
JIT编译器的优化思路

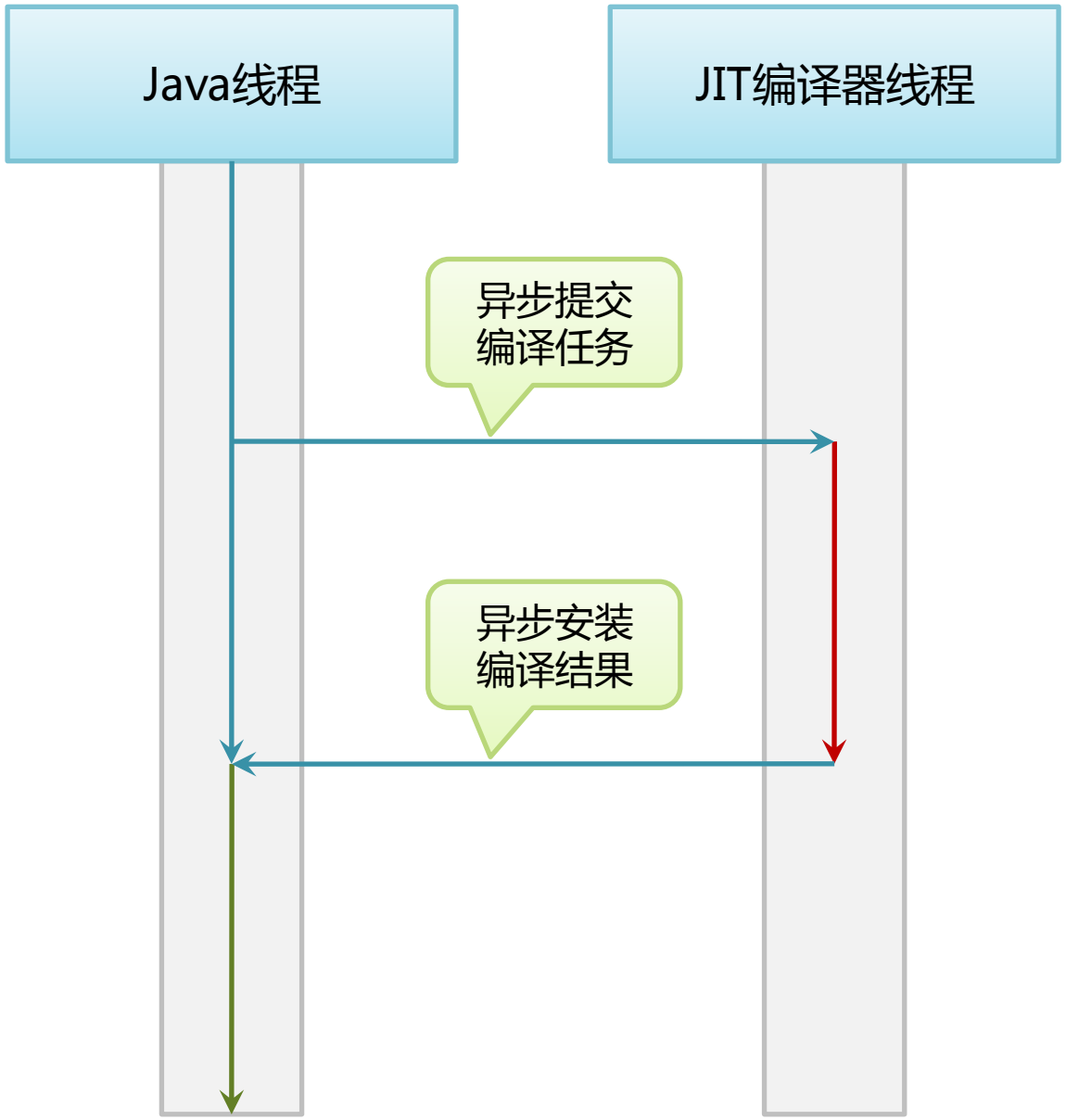
- 有选择性的优化
  - 选择要优化的代码
    - 其中，选择要优化的代码路径
  - 选择优化的程度
    - 追求交互性的时候只选用效费比高的优化算法
    - 追求顶峰性能时选择合适的代码进行高度优化
      - 传统的编译器优化算法都可以有选择性得到应用
- 为常见情况而优化
  - 避开非常见情况，留下“逃生门”

解释与动态编译（Client VM例）



解释与动态编译——后台编译

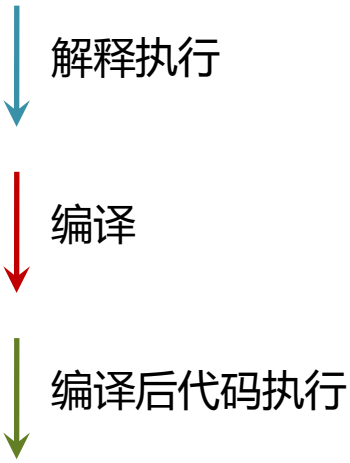


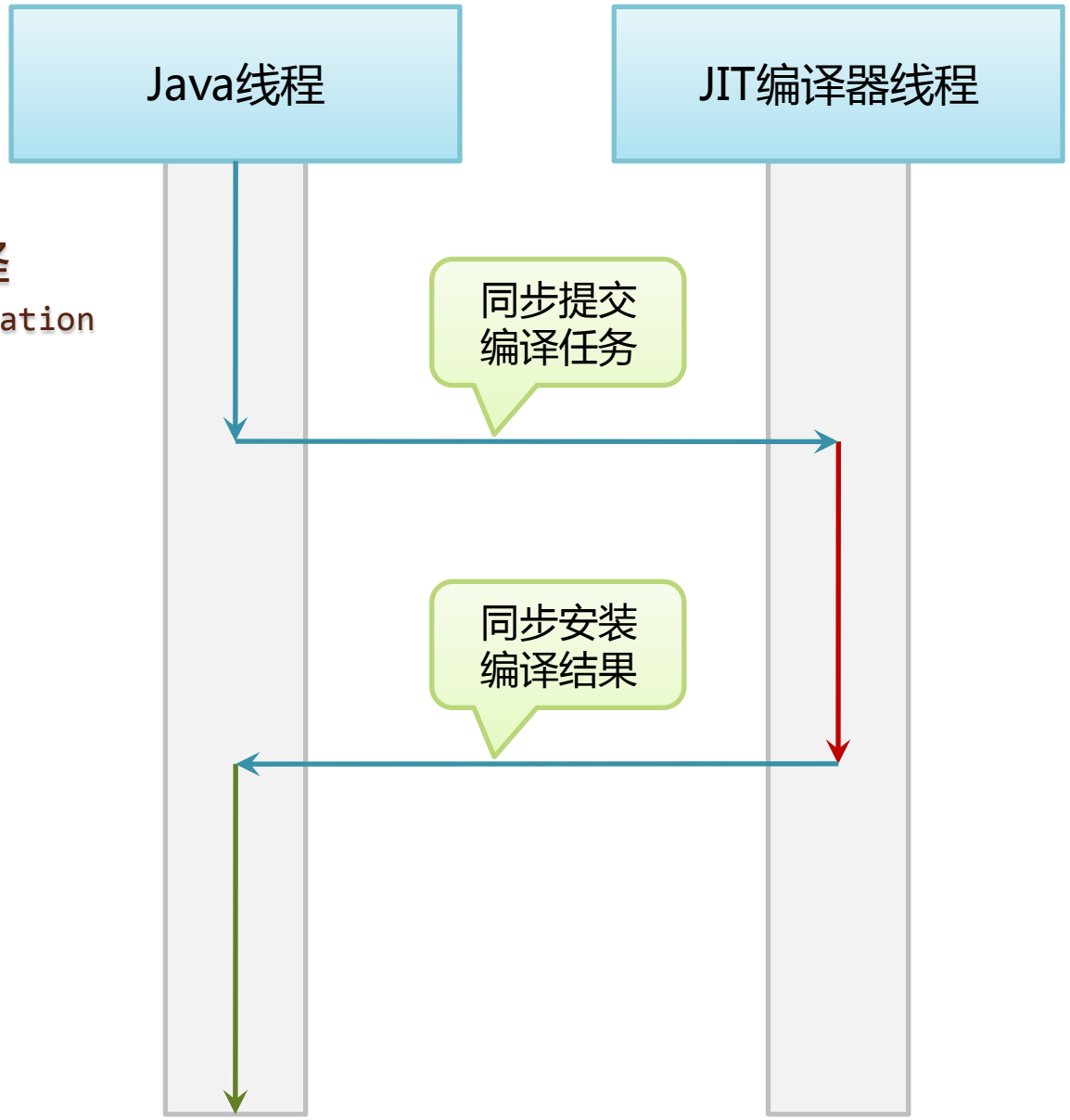


参考资料：图10

解释与动态编译
——禁用后台编译

（-XX:-BackgroundCompilation或-Xbatch）





关于后台编译的一篇早期论文

从解释转入编译

- JIT编译默认在后台的编译器线程进行
  - 主要由BackgroundCompilation参数控制
- 方法入口处的计数器溢出触发的编译后代码会在下次该方法被调用时投入使用
  - 这样触发的编译称为标准编译，其入口位于方法的开头
  - 编译完成后，对应的methodOop的方法入口会对应修改
- 方法回边处的计数器溢出触发的编译后代码会在编译完成后某次再到回边检查的时候投入使用
  - 这样触发的编译称为OSR编译，其入口位于某个回边处
  - 编译完成后，对应的methodOop的入口不变
  - 回边计数器溢出后，如果是后台编译的话，会调用计数器调整到溢出状态；同时把把回边计数器的值降低一些，让循环继续执行若干轮之后再次溢出，然后检查编译是否完成；若未完成则再次将计数器值降低然后重复上述步骤直到编译完成后转入编译后代码（或者循环结束了就不在本次调用转入编译后代码）

根据计数器触发编译的条件

```
class InvocationCounter {
 private:                                // bit no: |31  3|  2  | 1 0 |
  unsigned int _counter;                 // format: [count|carry|state]
  enum PrivateConstants {
    number_of_state_bits = 2,
    number_of_carry_bits = 1,
    number_of_noncount_bits = number_of_state_bits + number_of_carry_bits,
    number_of_count_bits = BitsPerInt - number_of_noncount_bits,
    // ...
  };
  // ...
};

void InvocationCounter::reinitialize(bool delay_overflow) {
  InterpreterInvocationLimit =
    CompileThreshold << number_of_noncount_bits;
  InterpreterProfileLimit =
    ((CompileThreshold * InterpreterProfilePercentage) / 100)<< number_of_noncount_bits;
  // ...
  if (ProfileInterpreter) {
    InterpreterBackwardBranchLimit =
      (CompileThreshold * (OnStackReplacePercentage - InterpreterProfilePercentage)) / 100;
  } else {
    InterpreterBackwardBranchLimit =
      ((CompileThreshold * OnStackReplacePercentage) / 100) << number_of_noncount_bits;
  }
}
```

由计数器触发的JIT编译

- 方法调用计数器（ >= CompileThreshold ）
  - CompileThreshold
    - client: 1500
    - server: 10000
- 回边计数器
  - BackEdgeThreshold
    - client: 100000
    - server: 100000

目前HotSpot中并没有实际使用该参数

- 触发OSR编译（ >= InterpreterBackwardBranchLimit ）
  - OnStackReplacePercentage
    - client: 933
    - server: 140
  - InterpreterProfilePercentage
    - server: 33

由计数器触发的JIT编译

- 方法调用计数器（ >= CompileThreshold ）
  - CompileThreshold
    - client: 1500
    - server: 10000
- 回边计数器
  - BackEdgeThreshold
    - client: 100000
    - server: 100000
- 触发OSR编译（ >= InterpreterBackwardBranchLimit ）
  - OnStackReplacePercentage
    - client: 933
    - server: 140
  - InterpreterProfilePercentage
    - server: 33

计数器的“衰减”（decay）

- 方法调用计数器会随着时间推移做指数衰减
  - 这样，方法调用计数器实际记录的不是准确的累计调用次数，而是方法的“热度”
  - 回边计数器不做衰减

- 相关参数：
  - UseCounterDecay = true
  - CounterHalfLifeTime = 30（秒）
  - CounterDecayMinIntervalLength = 500（毫秒）

解释器可以收集性能剖析信息（profiling）

- 在-client模式中解释器默认不收集profile（ProfileInterpreter为false）
- 在-server模式中，解释器会包含有profiling逻辑，供C2用作优化依据
  - 分支的跳转/不跳转的频率
  - 某条指令上出现过的类型
  - 是否出现过空值
  - 是否出现过异常
  - etc …

解释器可以收集性能剖析信息（profiling）

- 收集profile时间越长越能精确了解程序行为特性，让C2编译器可以生成更优质的代码
  - 例如将执行频率高的代码路径安排在连续的直线上，将其它代码安排在后面
- 但收集profile过程中解释速度较慢，在这个模式停留太久反而得不偿失
- 权衡！

固有函数（intrinsics）

- 有些Java方法在HotSpot中直接用硬件的特殊指令来实现
  - 如Math.sin()、Unsafe.compareAndSwapInt()之类
- 调用这些方法，在解释模式有可能还是调用了本地方法，而在被JIT编译的代码中则直接把其中的操作内联了进来

隐式异常处理

- 对空指针、栈溢出等异常情况不显式检查，而是直接生成不检查错误的代码
  - 在正常执行时不会因为要检查异常条件而带来任何开销
  - 在异常发生时速度比显式检查异常条件慢
  - 若检测到异常经常发生则重新编译，生成显式检查异常条件的代码

逃逸分析（escape analysis）

- 检查某个对象的使用是否只在某个受限范围内（方法内/线程内）
- 可以为其它优化提供机会
  - 标量替换
  - 栈上分配
  - 锁削除

参考资料

标量替换（scalar replacement）

- (Point的例子)

栈上分配（stack object allocation）

- HotSpot目前没有使用该优化

锁削除（lock elision）

- 对无法逃逸到别的线程的对象做同步不会有任何效果
- 但是做了这种优化需要考虑与Java Memory Model的可能冲突
  - monitorenter与monitorexit都是full memory barrier

增大锁粒度（lock coarsening）

JIT编译器不只要生成代码

- 还需要生成：
  - ExceptionHandlerTable / ImplicitExceptionTable
  - oopMap / relocInfo 等
  - 代码中使用的一些不直接插在代码中的常量
  - write barrier
    - 为了支持分代式GC/区域式GC
    - HotSpot目前没有使用read barrier
  - safepoint
    - 轮询一个特殊的内存页看是否需要把线程停下
    - GC只能在safepoint将线程停下
    - safepoint被插入到回边或方法返回处

JIT编译器生成的nmethod对象

```
// A nmethod contains:
//  - header                 (the nmethod structure)
//  [Relocation]
//  - relocation information
//  - constant part          (doubles, longs and floats used in nmethod)
//  [Code]
//  - code body
//  - exception handler
//  - stub code
//  [Debugging information]
//  - oop array
//  - data array
//  - pcs
//  [Exception handler table]
//  - handler entry point array
//  [Implicit Null Pointer exception table]
//  - implicit null table array
```

虚方法分派（virtual method dispatch）

虚方法分派的实质

单根继承的面向对象类型系统的例子

类型 \ 方法	m	n	o	p
A	A.m	A.n		
B	A.m	B.n	B.o	B.p
C	A.m	C.n	B.o	C.p

按行看就是附加在每个类型上的虚方法表
按列看就是附加在每个调用点（callsite）上的inline-cache

方法内联（method inlining）

- 面向对象编程习惯倾向使用大量小的虚方法
- 方法内联可以消除调用虚方法开销
  - 当调用的目标方法代码比调用序列代码小时，内联使生成的代码更小
  - 通常情况下内联会使生成的代码比不内联有所膨胀
- 更重要的是它可以增大其它优化的作用域
  - 使过程内分析可以达到部分过程间分析的效果
- 对提高Java程序性能至关重要

参考论文

虚方法分派——C++的常见做法

虚方法分派——C++的常见做法

- C++的编译器实现可以根据分析决定内联非成员函数，或者非虚成员方法调用
- 但一般无法内联通过指针对虚方法的调用
  - 存在例外，但非常少见：e.g.

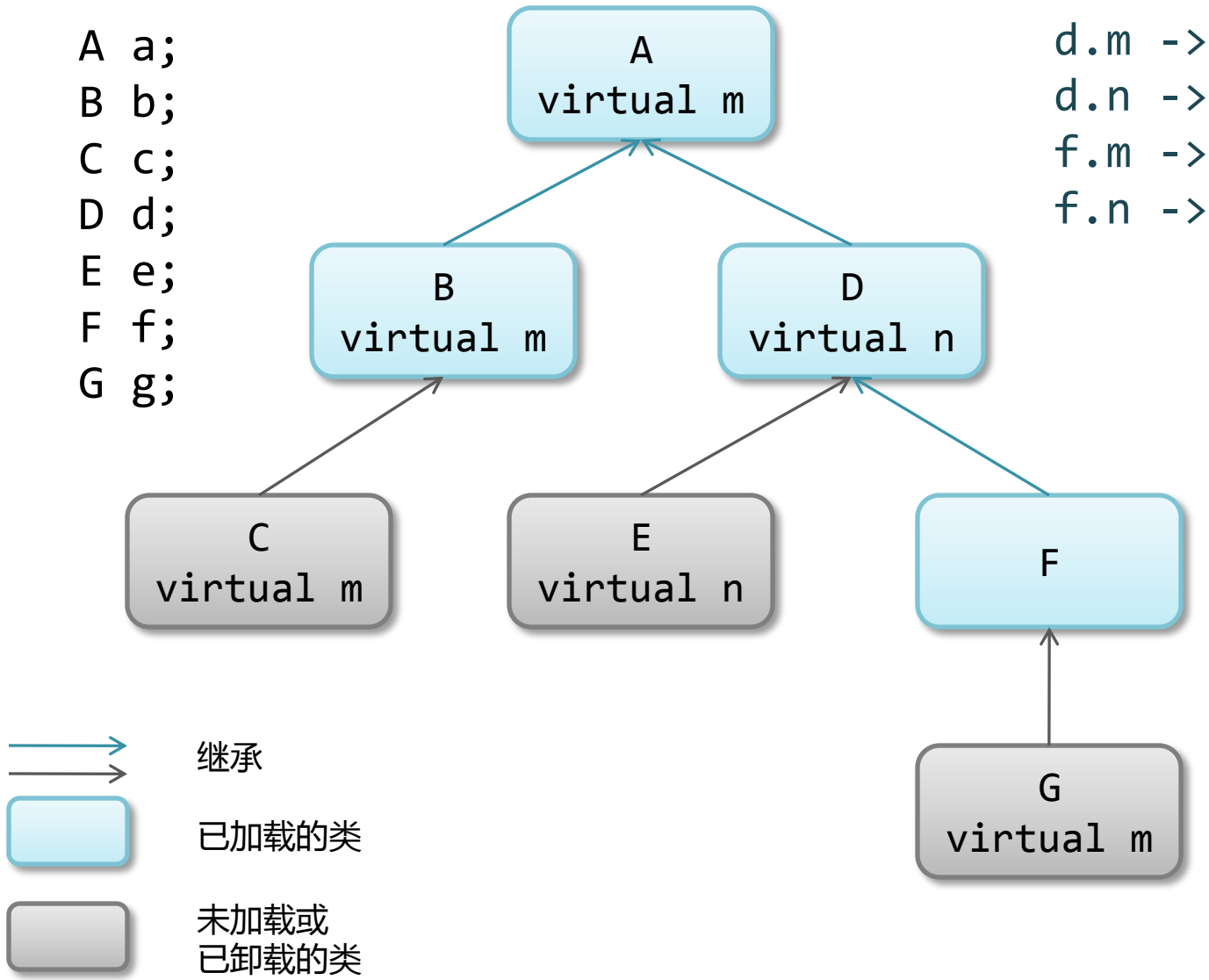
类层次分析（CHA）

- class hierarchy analysis
- Java程序的一种全程序分析
  - 静态编译器无法做这种分析，因为Java允许动态加载和链接，无法静态发现全程序的所有代码
  - 动态编译器则可以激进的以当前已加载的信息为依据做优化，当类层次发生变化时还可以找“退路”

类层次分析（CHA）

```
A a;
B b;
C c;
D d;
E e;
F f;
G g;
```

```
a.m -> A.m, B.m
b.m -> B.m
d.m -> A.m
d.n -> D.n
f.m -> A.m
f.n -> D.n
```



图片来源

去虚拟化（devirtualization）

- CHA能证明单调用目标时
  - 直接调用
  - 直接内联
- CHA未能证明单调用目标时
  - 带guard的调用（guarded inline-cache）
  - 带guard的内联
- …还有啥，补上！

inline cache

- 目标方法的正常入口之前
- 状态转换
  - 初始（clean）
  - 单态（？

inline cache的状态转换

```
//-----------------------------------------------------------------------------
// The CompiledIC represents a compiled inline cache.
//
// In order to make patching of the inline cache MT-safe, we only allow the following
// transitions (when not at a safepoint):
//
//
//         [1] --<--  Clean -->---  [1]
//            /       (null)      \
//           /                     \      /-<-\
//          /          [2]          \    /     \
//      Interpreted  ---------> Monomorphic     | [3]
//  (compiledICHolderOop)       (klassOop)     |
//          \                       /   \     /
//       [4] \                     / [4] \->-/
//            \->-  Megamorphic -<-/
//                  (methodOop)
//
// The text in paranteses () refere to the value of the inline cache receiver (mov instruction)
//
// The numbers in square brackets refere to the kind of transition:
// [1]: Initial fixup. Receiver it found from debug information
// [2]: Compilation of a method
// [3]: Recompilation of a method (note: only entry is changed. The klassOop must stay the same)
// [4]: Inline cache miss. We go directly to megamorphic call.
//
// The class automatically inserts transition stubs (using the InlineCacheBuffer) when an MT-unsafe
// transition is made to a stub.
```

Self-modifying code

HotSpot实现的虚方法分派

- monomorphic inline cache

参考文章

栈上替换（OSR）

- on-stack replacement
- 由回边计数器溢出引发JIT编译后，在循环中从解释器转入编译后代码
- 发生逆优化时要从编译后代码转回到解释器中
- PrintCompilation的日志中以%表示发生了OSR编译

HotSpot从版本号2.0开始支持OSR

方法内联（method inlining）

内联带来的代码膨胀问题

- 内联可能引致冗余，因而容易导致生成的代码比不内联有所膨胀
- 但调用方法的代码序列也有一定长度
- 因而如果被内联的方法比调用方法的代码序列短，则内联完全有益而无害
- 若被内联的方法长、被调用次数少且未能为其它优化带来更多优化机会，则内联反而可能给性能带来负面影响

HotSpot的内联选项

- 权衡！选择要内联的方法
- HotSpot在产品中的参数：
  - -XX:+Inlining
  - -XX:+ClipInlining
  - -XX:InlineSmallCode=1000
  - -XX:MaxInlineSize=35
  - -XX:FreqInlineSize=325
- HotSpot的诊断选项：
  - -XX:+PrintInlining
- HotSpot在开发模式下的参数：
  - -XX:+InlineAccessors
  - -XX:MaxTrivialSize=6
  - -XX:MinInliningThreshold=250
  - -XX:MaxInlineLevel=9
  - -XX:MaxRecursiveInlineLevel=1
  - -XX:DesiredMethodLimit=8000
  - ……

逆优化（deoptimization）

- 当激进优化的前提假设不再成立时
  - 加载了新的类，改变了类层次
  - 执行到了uncommon trap
  - 程序运行中接入了调试器
  - etc …

安全性

- Java的安全设计依赖stack walking
- 但方法内联后，实际的调用栈可能缺失了部分栈帧
- 为了维持安全性检查的需求，HotSpot通过vframe维护了一套虚拟栈帧，当需要逆优化或做安全性检查的时候可以用上

实验：HotSpot的启动参数

- -Xint
- -Xcomp
- -XX:+UnlockDiagnosticVMOptions
- -XX:+PrintInterpreter
- -XX:+PrintCompilation
- -XX:+PrintAssembly
- -XX:+PrintInlining
- -XX:+PrintIntrinsics
- -XX:+PrintNMethods
- -XX:-BackgroundCompilation
- -XX:+TraceItables
- -XX:+TraceICs
- …

更详细的参数列表，请参考OpenJDK 6的下列源文件：

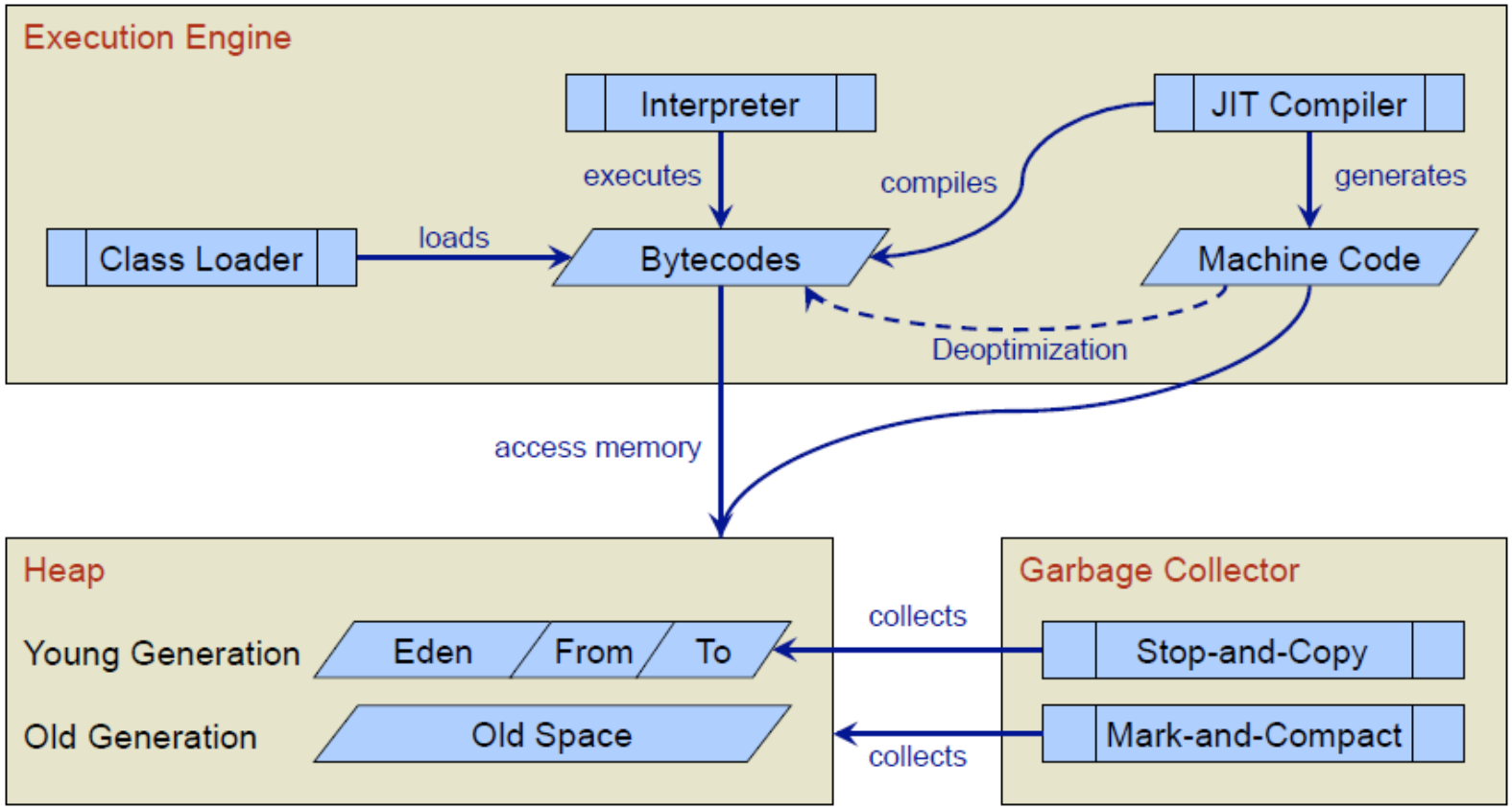
globals.hpp
globals_extension.hpp
c1_globals.hpp
c2_globals.hpp
globals_x86.hpp
c1_globals_x86.hpp
c2_globals_x86.hpp
globals_windows.hpp
globals_windows_x86.hpp
globals_linux.hpp
globals_linux_x86.hpp
arguments.cpp

-XX:+PrintAssembly

演示：
java -XX:+PrintCompilation

HotSpot Client Compiler（C1）

HotSpot Client VM



图片来源

C1的工作流程

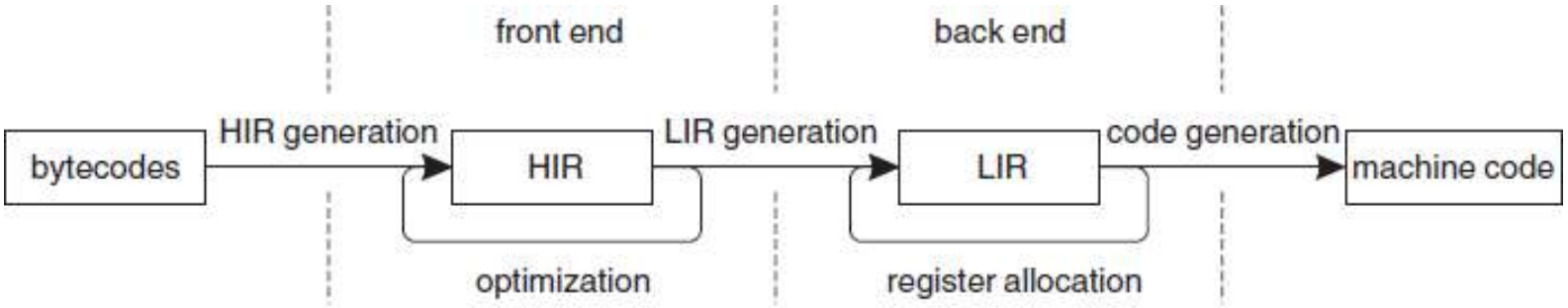


Fig. 2. Structure of the Java HotSpot™ client compiler.

HIR

- High-level Intermediate Representation
- 按基本块组织的控制流图
- 基本块内是SSA形式的图

SSA

- static single-assignment form

抽像解释（abstract interpretation）

- 也称为符号执行（symbolic execution）

抽象解释（2）

方法内联（method inlining）

- 从字节码转换为HIR的过程中完成
- 基于CHA和方法的字节码大小
- 不基于动态收集的profile信息

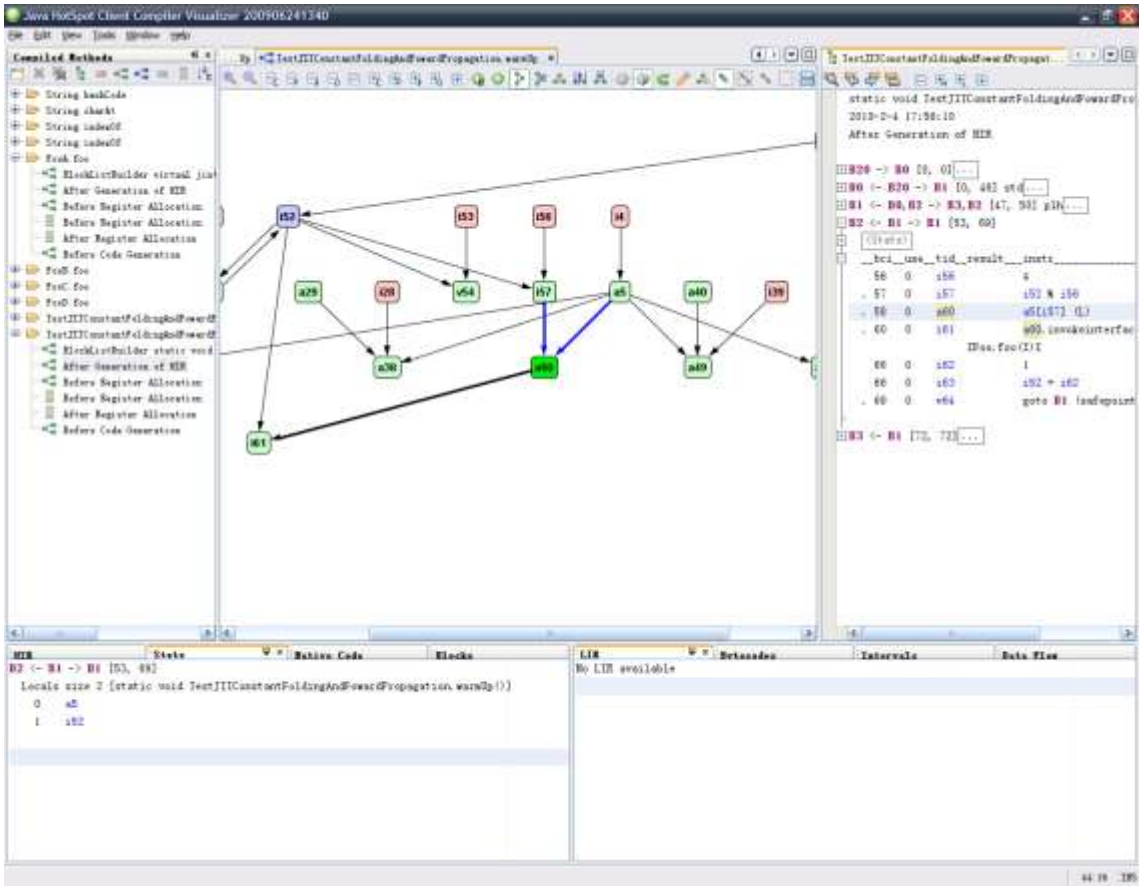
LIR

- Low-level Intermediate Representation

线性扫描寄存器分配器

参考资料

c1visualizer



c1visualizer

- 将C1编译器在编译各阶段的数据结构可视化展现出来的工具
  - 原始字节码
  - HIR生成
  - LIR生成
  - 寄存器分配
  - 最终代码生成
- 可以展示控制流、数据流、寄存器分配区间等
- 配合JDK 6或7的debug版HotSpot使用
  - 使用-XX:+PrintCFGToFile选项获取output.cfg

- 官方网站： http://java.net/projects/c1visualizer/

HotSpot Server Compiler（C2）

一些参数

- develop(intx, ImplicitNullCheckThreshold, 3,
  \
- "Don't do implicit null checks if NPE's in a method exceeds limit")

Sea-of-nodes

- Program Dependence Graph
- 不以基本块为单元来组织控制流，而是将控制依赖与数据依赖统一在一起，把所有节点组成一大图
- SSA形式的中间代码

参考资料

全局值编号（GVN）

- global value numbering

循环不变量代码外提（LICM）

- loop invariant code motion

循环展开（loop unrolling）

削除冗余数组边界检查（RCE）

- Range Check Elimination

uncommon trap

- 不为执行频率低的路径生成代码，而是生成一个uncommon trap
  - 避免影响类型判断
  - 降低编译的时间/空间开销
- 真的执行到uncommon trap时，退回到解释模式或者重新编译

自底向上改写系统（BURS）

- bottom-up rewrite system

图着色寄存器分配器

参考资料

最终的机器码生成

- 每个平台有对应的描述文件
  - Architecture Description
  - x86_32.ad
  - x86_64.ad
  - sparc.ad
- -XX:+PrintOptoAssembly在此实现

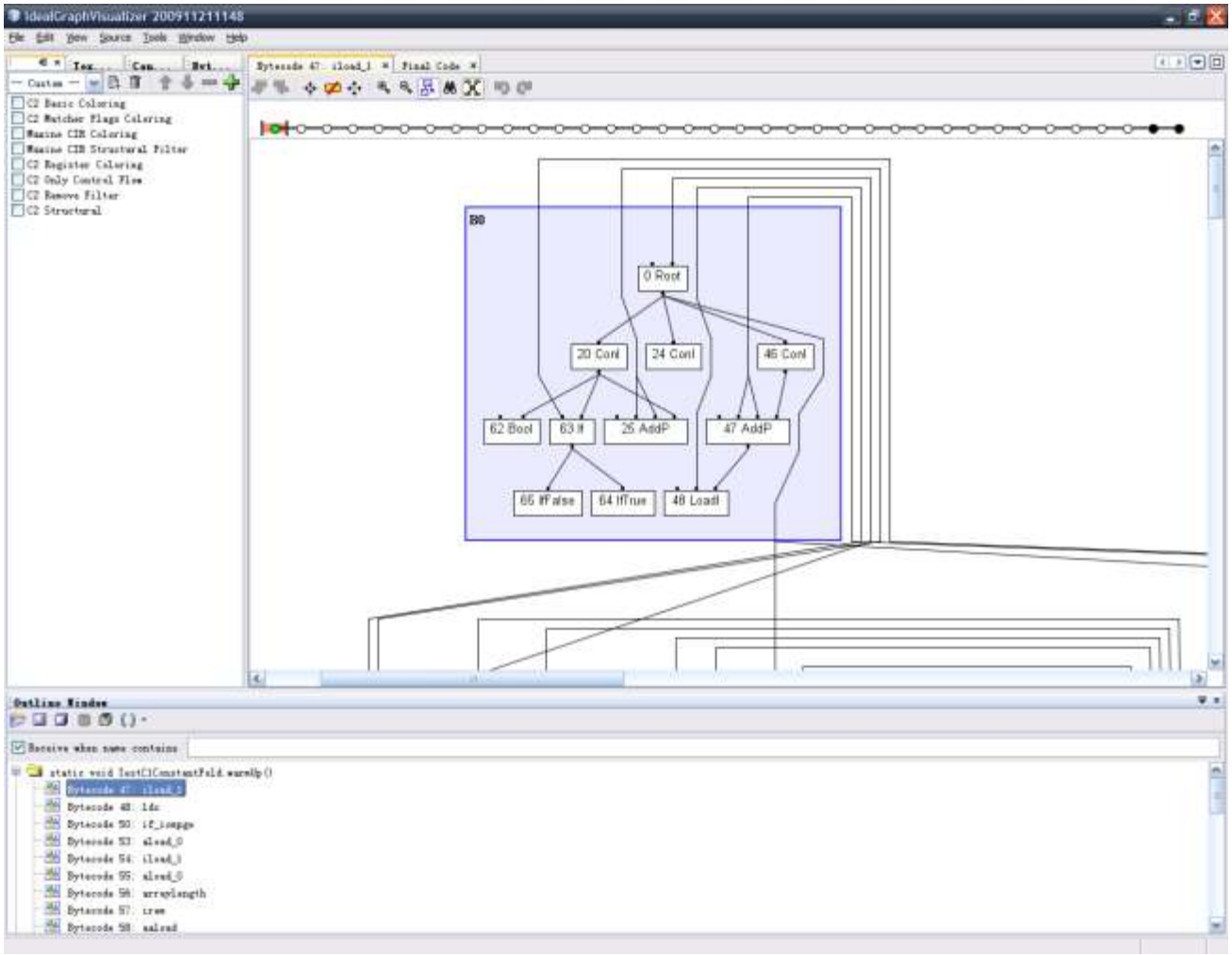
-XX:+PrintOptoAssembly

多层编译（tiered-compilation）

- 多层编译的混合执行模式
  - -server将C1编译器也包含进来
  - 第0层：解释器不承担profile任务，只用于启动并触发C1编译
  - 第1层：C1编译，生成不带有profile逻辑的代码
  - 第2层：C1编译，生成带有方法级计数器profile的代码
  - 第3层：C1编译，生成带有字节码指令级profile的代码
  - 第4层：C2编译，利用C1编译的代码提供的profile信息进行激进优化
    - 当激进优化的假设不再成立时，可以退回到解释器或者第1层的代码
- 默认工作方式（JDK 6 update 25为准）：
  - 优化：第0层 -> 第3层 -> 第4层
  - 去优化：第4层 -> 第0层 / 第一层
  - 未使用：第2层
- 新的HotSpot的多层编译策略参考AdvancedThresholdPolicy
- ~~Java 7中该模式将成为HotSpot -server的默认模式~~
  - Oracle JDK7的首个发布版里还未将其设置为默认模式
- J9与JRockit也使用多层编译方式执行Java代码

参考资料

Ideal Graph Visualizer



Ideal Graph Visualizer

- 将C2编译器在编译各阶段的数据结构可视化展现出来的工具
- 可配合用debug版HotSpot使用
  - -XX:PrintIdealGraphLevel=n -XX:PrintIdealGraphFile=filename
  - n为0-4，filename为指定输出log的文件路径
- 已集成进OpenJDK7
- 官方网站：http://ssw.jku.at/General/Staff/TW/igv.html

HOTSPOT VM的查看工具

参考资料

JConsole

VisualVM

Serviceability Agent

- 进程外审视工具
- Windows上JDK 7-b64之后才可以用

HOTSPOT VM未来展望

参考资料

版本51.0的Class文件

- 禁用jsr与jsr_w字节码指令（6911922）
- 必须用类型检查式校验器（6693236）
- method handle常量（6939203）

MethodHandle

invokedynamic

HotSwap

- Sun JDK从1.4.0开始提供HotSwap功能
- 但老的HotSwap实现……？（性能不好？功能太少？）
- 新的HotSwap实现（DCEVM）：减少限制，可以动态修改类的内容，甚至可以修改继承关系，而且在swap前后没有额外运行开销

参考资料（1）（2）（3）

协程（coroutine）

参考资料

Automatic object inlining

参考资料

Multi-Tasking Virtual Machine (MVM)

- JSR 121

参考资料

—— 除HotSpot外，Sun还有其它JVM吗？
—— 有，很多

Sun的其它JVM（1）——早期时代

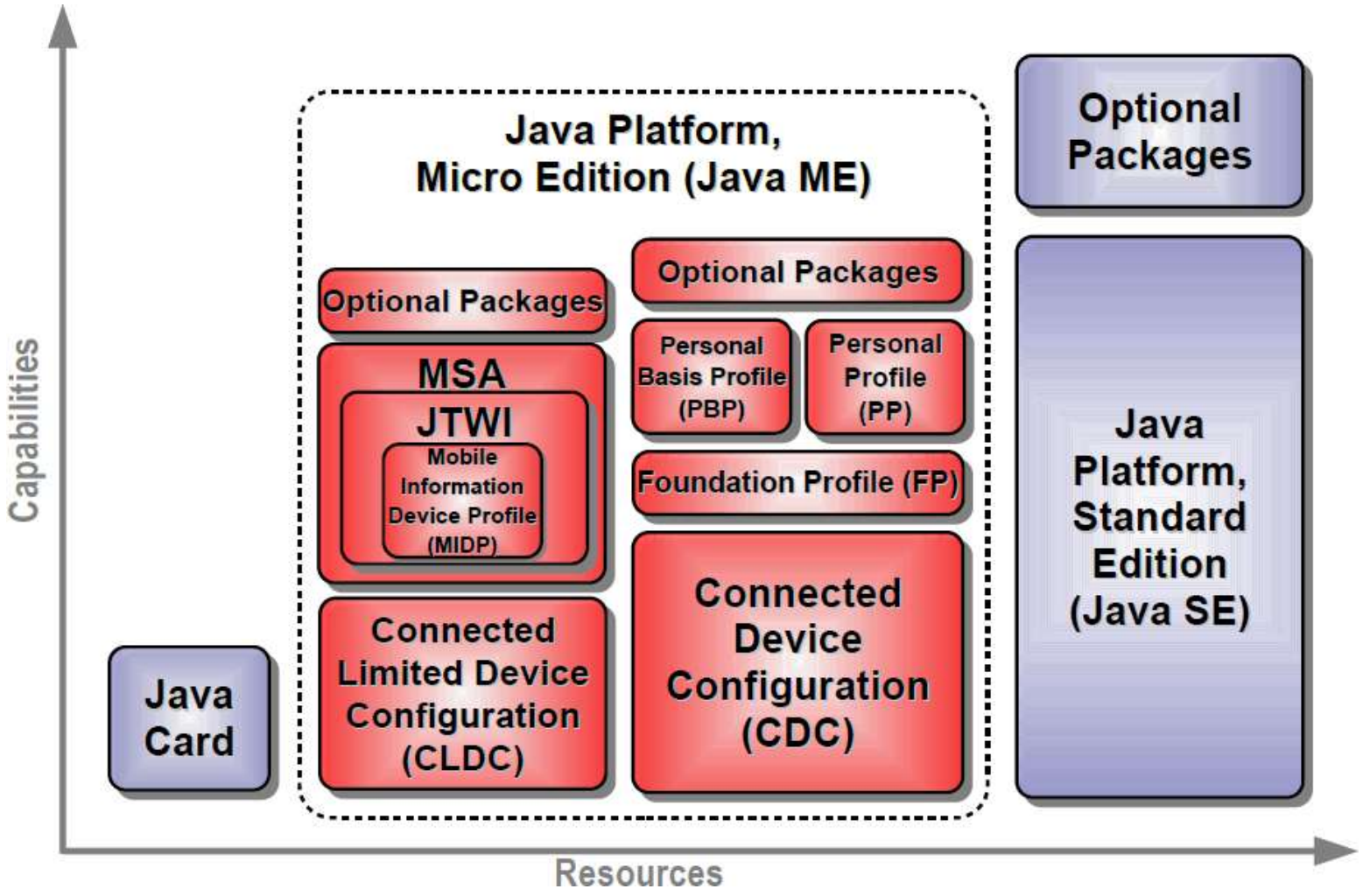
- 始祖JVM
  - Classic VM
    - 纯解释
    - 可从外部插入JIT编译器（例如Symantec JIT编译器）
      - 但插入后解释器就默认不工作了
      - 全靠外部提供的JIT编译器来生成执行代码
      - 1996年Borland和Symantec相继发布JIT编译器
      - Sun购买了Symantec的JIT编译器的许可证
      - 同年10月25日Sun发布第一个配套使用的JIT编译器
    - 从J2SDK 1.4.0开始不再随JDK发布（Solaris上从 1.3.0开始）

- 与HotSpot起源时间相近的JVM
  - Exact VM（EVM）
    - 最初衍生自Classic VM，但得到了大量改进
    - 整合了解释器与两个JIT编译器
      - 实现了混合模式执行以及多层编译
    - 开发目的是研究高性能准确式GC
    - 在JDK 1.2时代也投入到实际使用（Solaris上）
    - 与HotSpot争夺Sun的产品JVM地位的时候失败，项目被终止
    - 其Research VM的地位后来由Maxine VM替代

Sun的其它JVM（2）——嵌入式

- KVM
  - ‘k’ for “kilobyte”
  - 简单，轻量，高度可移植，但比较慢
  - 由1998年的Spotless研究项目衍生而来，于1999年的JavaOne上发表
  - 是CLDC的参考实现（reference implementation）；得到过广泛部署
- CDC HotSpot Implementation （GPL 2.0）
  - 也称为phoneME Advanced VM或CVM (C Virtual Machine)
  - 大部分功能用C实现，也用到了一些汇编（百行左右）
  - 针对相对高端的嵌入式设备设计的高性能JVM
- CLDC HotSpot Implemantation （GPL 2.0）
  - 也称为phoneME Feature VM
  - 针对相对低端的嵌入式设备设计的JVM
  - 替代KVM
- Squawk VM （GPL 2.0）
  - 用于Sun SPOT
  - VM的很大部分是用Java实现的（可以算元循环VM的一种）
  - 部署前先将程序翻译为C，将生成的C代码编译后再部署到设备
- Java SE Embedded
  - 其中的JVM基本上就是桌面版HotSpot，专门为嵌入式应用调优
  - HotSpot的新开发也有考虑到这方面需求，如G1-Lite低延迟GC

Sun的其它JVM（2）——嵌入式



参考资料

Sun的其它JVM（3）——元循环VM

- 用Java写的JVM
  - meta-circular VM
    - 用一种语言实现自身的运行环境
- JavaInJava（1997-1998）
  - 纯Java实现，只有解释器
  - 必须在一个宿主JVM上运行，非常慢
- Maxine（2005-现在）（GPL 2.0）
  - 几乎纯Java实现
    - 只有用于启动JVM的加载器用C编写
  - 有先进的JIT编译器与GC，没有解释器
  - 可在宿主模式或独立模式执行
  - 速度接近client模式的HotSpot
  - 通过接近90%的JCK（Java Compatibility Kit）测试
  - 吸收了别的同类JVM的经验：Jikes RVM、Joeq、Moxie等
    - 还大量应用了一个元循环Self虚拟机的经验：Klein VM
- Graal（2010-现在）（GPL 2.0）
  - Maxine VM中C1X编译器的进化版
  - 纯Java写的动态编译器，生成x64机器码
  - 能插入到HotSpot VM中使用
  - 良好的可扩展性，能让用户为自己的库写特定优化扩展

Sun的其它JVM（4）——实时Java

- Project Mackinac
  - Sun的第一个RTSJ（JSR 01）实现
  - 基于HotSpot VM

参考资料

Sun的其它JVM（5）——硬件实现

- picoJava
  - 参考论文
- picoJava-II
- MicroJava 701
- UltraJava

- 这些尝试并没有获得成功
- 但其它一些公司的Java芯片产品成功打入过低端、低功耗市场，如aJile、 Jazelle

其它JVM（及JIT）的部分列表

- IBM J9
- Oracle JRockit
- Apache Harmony
- JamVM
- cacaovm
- SableVM
- Kaffe
- Jelatine JVM
- NanoVM
- Symantec JIT
- MRP
- Moxie JVM
- Jikes RVM
- ORP
- joeq
- VMKit
- Ovm
- kissme
- shuJIT
- OpenJIT

其它JVM的执行引擎的技术方案

Azul JVM

- 源自Sun HotSpot VM
- 在GC、多核并发方面做了大量改进
  - Generational Pauseless GC
  - 生产环境中可以在864核/768GB内存的Azul Vega上高效运行
  - 目前也有基于x86的版本，Zing

参考资料

IBM J9

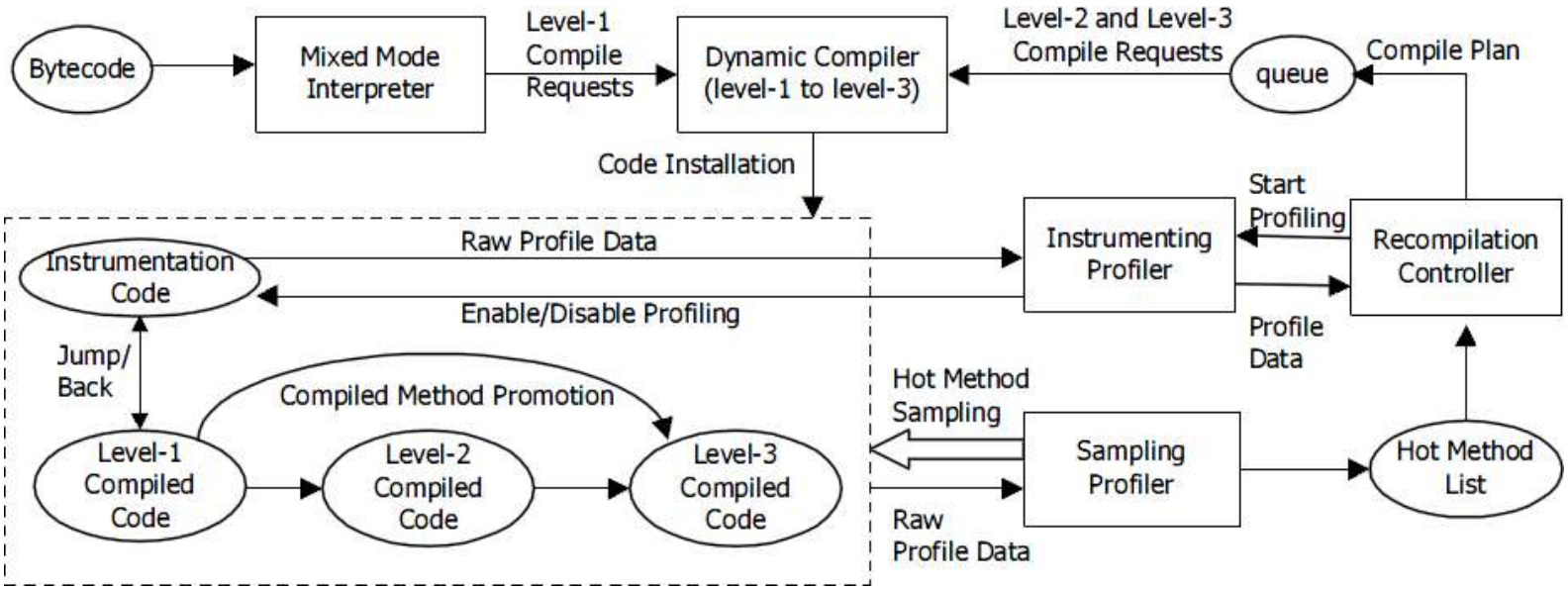
- （参考Pro (IBM) WebSphere Application Server 7 Internals
- 但这本书的描述有陷阱，要留心）
- 发展历程
  - OTI Smalltalk (K8) -> UVM -> J9

- 参考资料

IBM Sovereign JVM

- 发展历程与Sun Exact VM类似
- 最初源自Sun Classic VM，但做了大量改进
- 随早期IBM JDK发布，直到IBM JDK5开始被J9替代

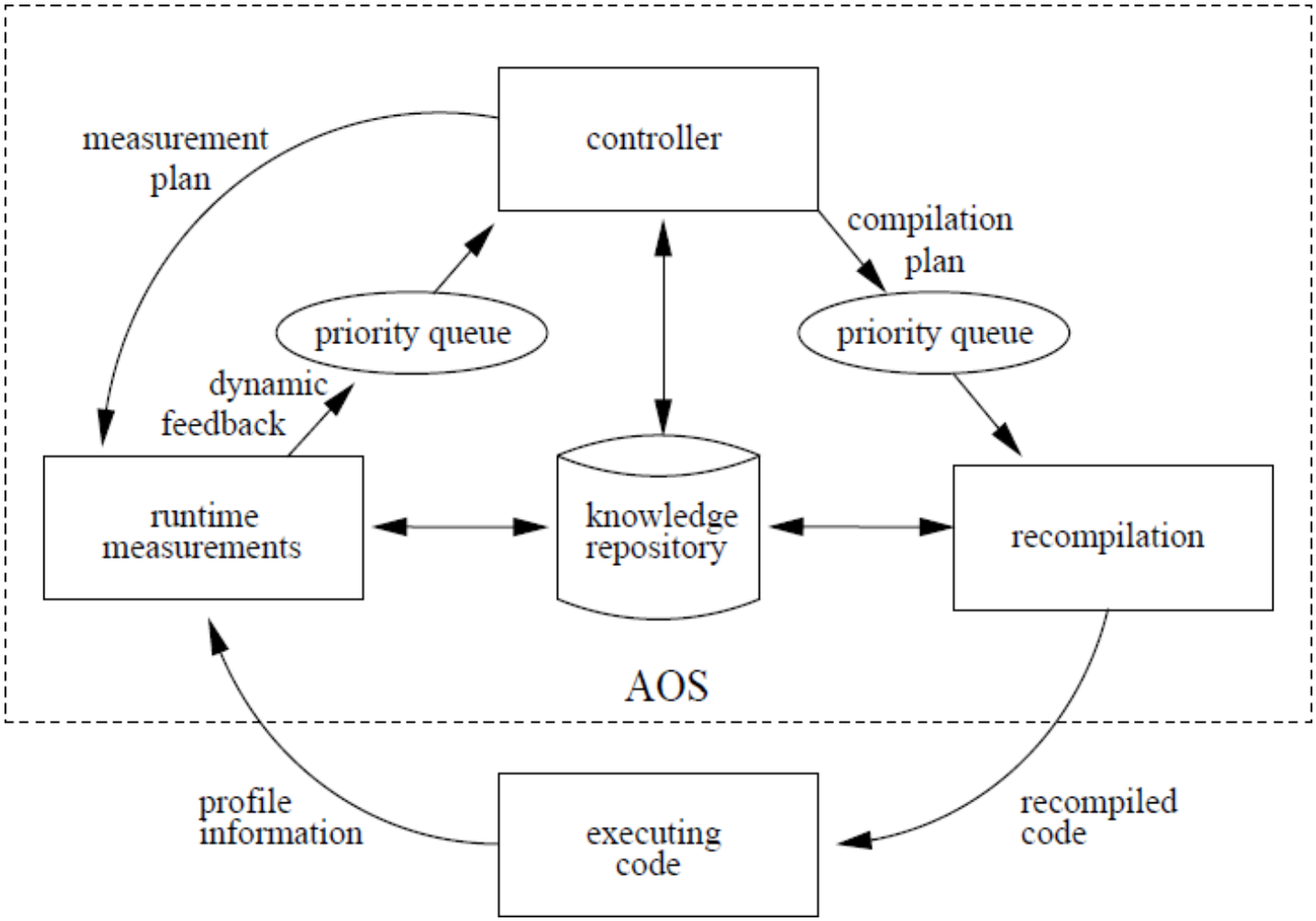
IBM JIT Compiler Dynamic Optimization Framework



参考资料

Jikes RVM

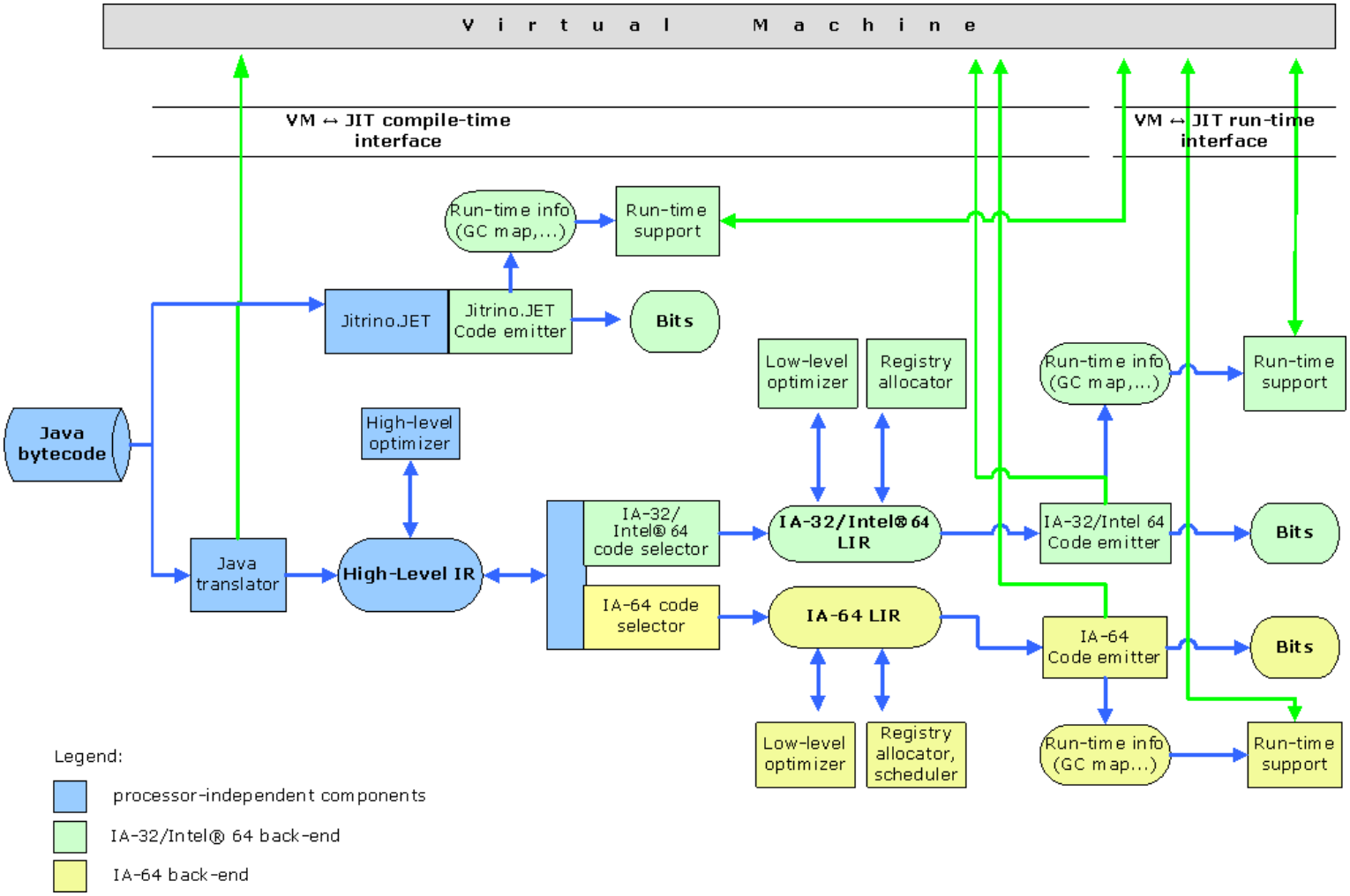
Jikes RVM Adaptive Optimization System



参考资料

Apache Harmony

Apache Harmony Jitrino Compiler Architecture



参考资料

joeq

joeq

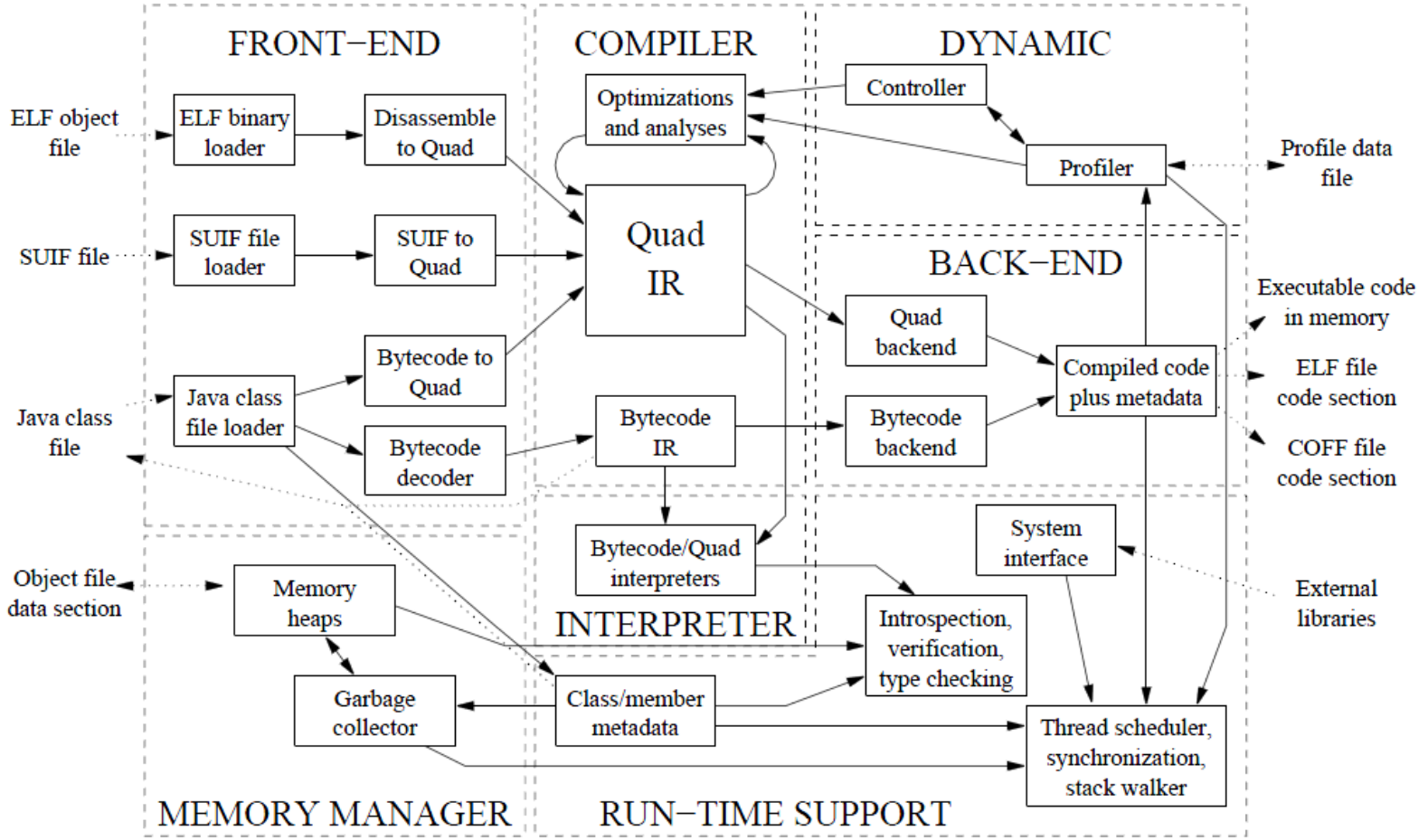
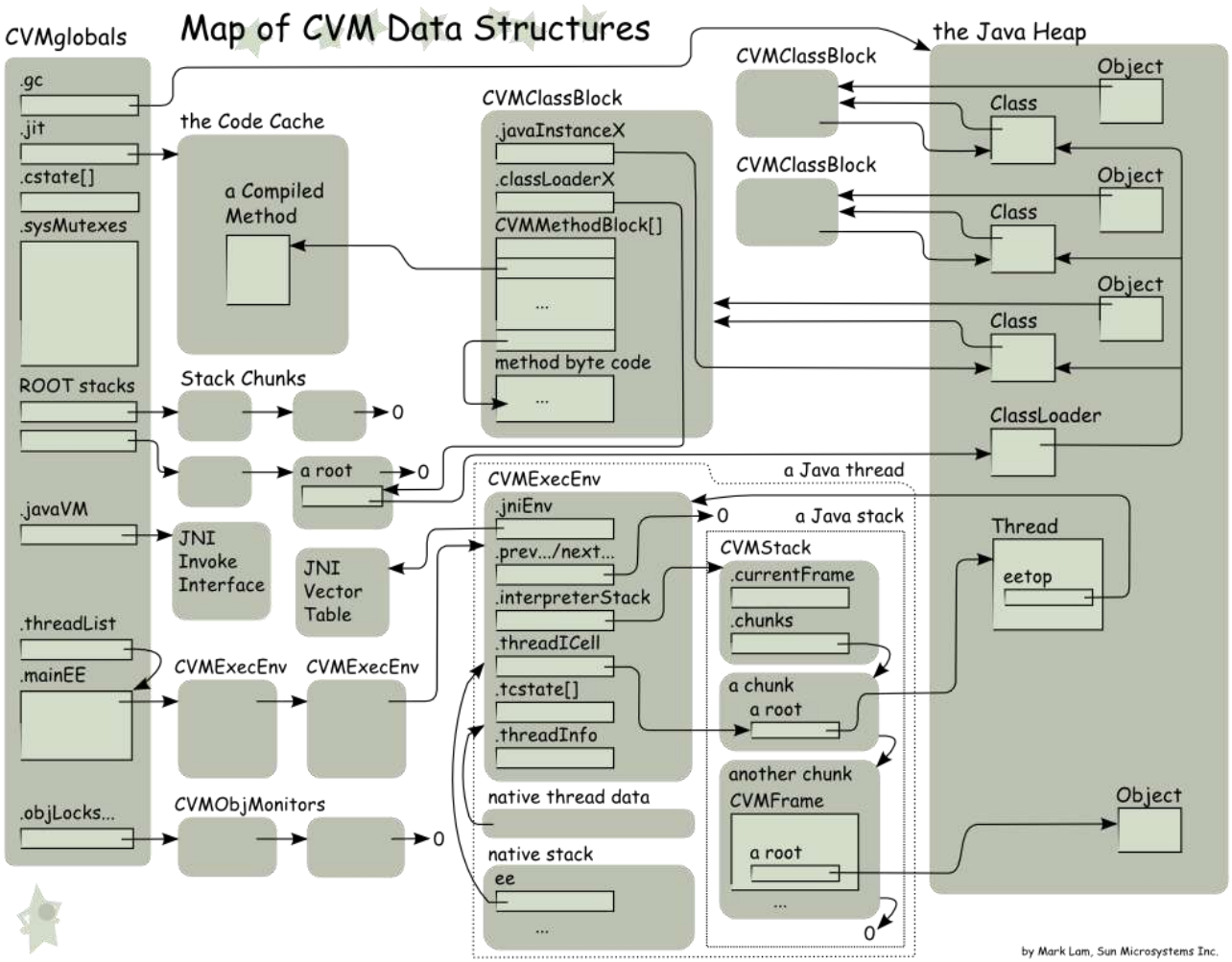


Figure 1. Overview of the Joeq system. Arrows between blocks signify either the flow of data between components, or the fact that one component uses another component.

参考资料

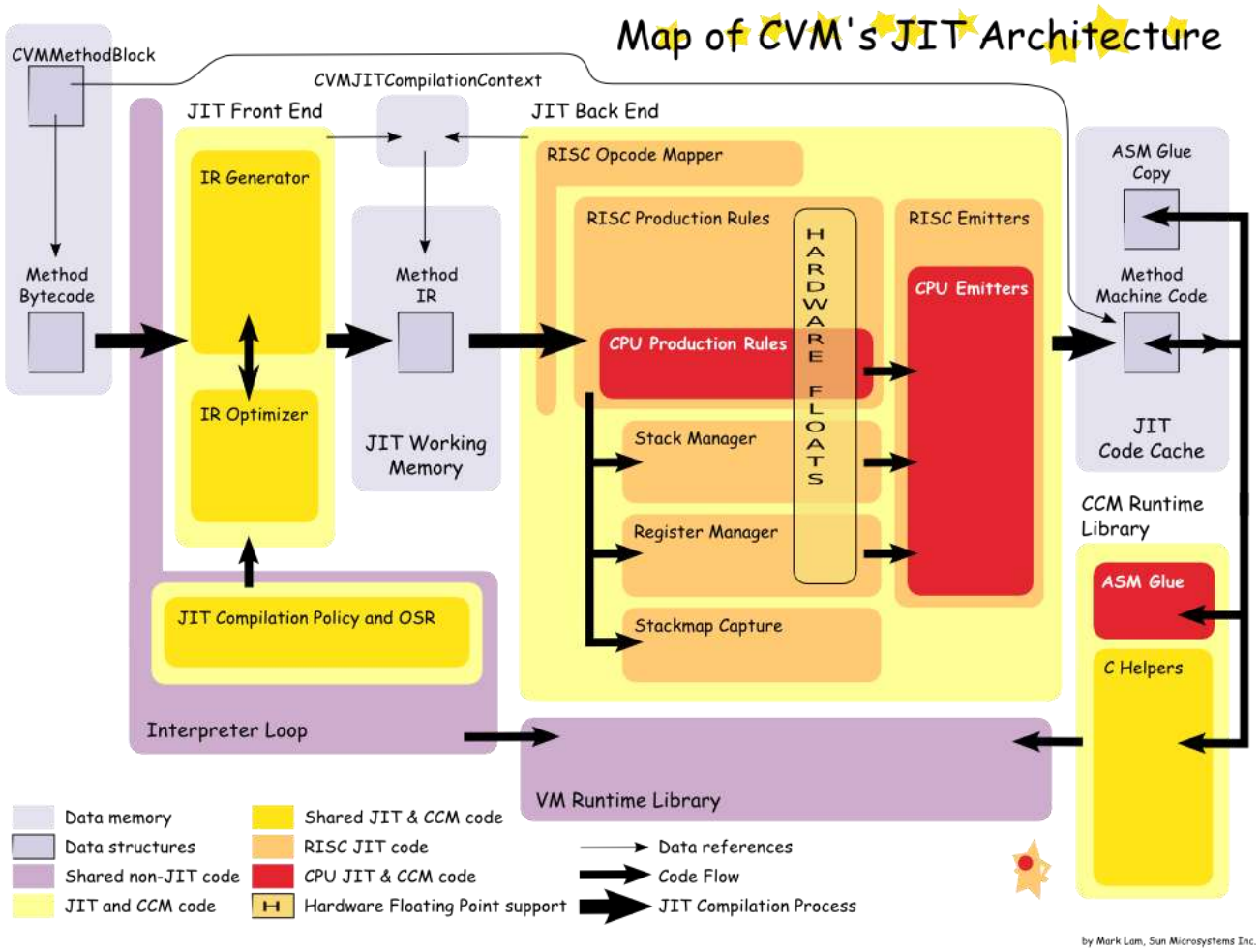
CVM

CVM


Map of CVM Data Structures
CVMglobals
.gc
.jit
.cstate[]
.sysMutexes
ROOT stacks
.javaVM
.threadList
.mainEE
.objLocks...
the Code Cache
a Compiled Method
Stack Chunks
a root
JNI Invoke Interface
JNI Vector Table
CVMExecEnv
CVMExecEnv
CVMObjMonitors
CVMClassBlock
.javaInstanceX
.classLoaderX
CVMMethodBlock[]
method byte code
the Java Heap
CVMClassBlock
CVMClassBlock
Object
Class
ClassLoader
a Java thread
CVMExecEnv
.jniEnv
.prev.../next...
.interpreterStack
.threadICell
.tcstate[]
.threadInfo
native thread data
native stack
ee
a Java stack
CVMStack
.currentFrame
.chunks
a chunk
a root
another chunk
CVMFrame
a root
Thread
eetop
Object
by Mark Lam, Sun Microsystems Inc.

参考资料

CVM


Map of CVM's JIT Architecture
CVMMethodBlock
CVMJITCompilationContext
JIT Front End
JIT Back End
IR Generator
IR Optimizer
Method Bytecode
Method IR
JIT Working Memory
RISC Opcode Mapper
RISC Production Rules
HARDWARE FLOATS
RISC Emitters
CPU Production Rules
CPU Emitters
Stack Manager
Register Manager
Stackmap Capture
ASM Glue Copy
Method Machine Code
JIT Code Cache
CCM Runtime Library
ASM Glue
C Helpers
JIT Compilation Policy and OSR
Interpreter Loop
VM Runtime Library
Data memory
Data structures
Shared non-JIT code
JIT and CCM code
Shared JIT & CCM code
RISC JIT code
CPU JIT & CCM code
Hardware Floating Point support
Data references
Code Flow
JIT Compilation Process
by Mark Lam, Sun Microsystems Inc.

参考资料

推荐阅读

推荐书籍

- The Java Language Specification
- The Java Virtual Machine Specification
  - 针对Java 5的修订
  - 针对Java 6的Class文件格式的修订
  - JVM规范第三版草案（2011/03/01）（到Java 6为止所有JVM规范更新的整合版）

- Oracle JRockit: The Definitive Guide

- Virtual Machines: Versatile Platforms for Systems and Processes
  - 中文版：虚拟机——系统与进程的通用平台
- Compilers: Principles, Techniques, and Tools (2nd Edition)
  - 中文版：编译原理（原书第2版）
- Advanced Compiler Design and Implementation
  - 中文版：高级编译器设计与实现
- Principles of Computer Organization and Assembly Language, Using the Java Virtual Machine
  - 中文版：计算机组成及汇编语言原理

较老书籍

- Java and the Java Virtual Machine
- Programming for the Java Virtual Machine
- Virtual Machines (Iain D. Craig著)
- Inside the Java Virtual Machine (2nd Edition)
  - 中文版：深入Java虚拟机（原书第2版）

推荐网站与博客

- HotSpot Publications
- JVM Language Summit
- OpenJDK
- The HotSpot Group
- javac Group
- the Da Vinci Machine Project
- The Java HotSpot Performance Engine Architecture
- Java™ Virtual Machine Technology – JDK 7
- HotSpot Internals for OpenJDK
- Da Vinci Machine Project Wiki
- Publications of the Institute for System Software
- IBM Research: Java JIT compiler and related publications
- Jikes RVM

- Cliff Click
- John Rose
- Mark Lam
- Fredrik Öhrström
- Ian Rogers
- The JRockit Blog
- Christian Thalinger
- Lukas Stadler
- Gary Benson
- Steve Goldman
- Xiao-Feng Li
- Christian Wimmer
- Maurizio Cimadamore
- Joseph D. Darcy
- Ben L. Titzer
- Steve Blackburn
- ……

谢谢！
期待以后继续交流 ^_^

这组演示稿会持续演化、完善……

附录A：Sun JDK历史与JVM

Sun JDK版本

- JDK 1.0
  - JDK 1.0　　1996-01-23
  - JDK 1.0.1
  - JDK 1.0.2

参考资料

Sun JDK版本

- JDK 1.1
  - JDK 1.1.0 1997-02-18
  - JDK 1.1.1
  - JDK 1.1.2
  - JDK 1.1.3
  - JDK 1.1.4 Sparkler 1997-09-12
  - JDK 1.1.5 Pumpkin 1997-12-03
  - JDK 1.1.6 Abigail 1998-04-24
  - JDK 1.1.7 Brutus 1998-09-28
  - JDK 1.1.8 Chelsea 1999-04-08

参考资料

Sun JDK版本

- J2SE 1.2
  - J2SE 1.2.0 Playground 1998-12-04
  - J2SE 1.2.1 (none) 1999-03-30
  - J2SE 1.2.2 Cricket 1999-07-08
    - J2SE 1.2.2 update 17

Sun JDK版本

- J2SE 1.3
  - J2SE 1.3.0 Kestrel 2000-05-08
    - J2SE 1.3.0 [1.3.0] (HotSpot 1.3.0-C)
    - J2SE 1.3.0 update 1 [1.3.0_01]
    - J2SE 1.3.0 update 2 [1.3.0_02]
    - J2SE 1.3.0 update 3 [1.3.0_03]
    - J2SE 1.3.0 update 4 [1.3.0_04]
    - J2SE 1.3.0 update 5 [1.3.0_05]
  - J2SE 1.3.1 Ladybird 2001-05-17
    - J2SE 1.3.1 [1.3.1]
    - J2SE 1.3.1 update 1 [1.3.1_01]
    - J2SE 1.3.1 update 1a [1.3.1_01a]
    - J2SE 1.3.1 update 2 [1.3.1_02]
    - J2SE 1.3.1 update 3 [1.3.1_03]
    - J2SE 1.3.1 update 4 [1.3.1_04]
    - J2SE 1.3.1 update 5 [1.3.1_05]
    - J2SE 1.3.1 update 6 [1.3.1_06]
    - J2SE 1.3.1 update 7 [1.3.1_07-b02]
    - J2SE 1.3.1 update 8 [1.3.1_08]
    - J2SE 1.3.1 update 9 [1.3.1_09]
    - J2SE 1.3.1 update 10 [1.3.1_10]
    - J2SE 1.3.1 update 11 [1.3.1_11]
    - J2SE 1.3.1 update 12 [1.3.1_12]

Sun JDK版本

- J2SE 1.4
  - J2SE 1.4.0 Merlin 2002-02-13
    - J2SE 1.4.0 [1.4.0]
    - J2SE 1.4.0 Update 1 [1.4.0_01]
    - J2SE 1.4.0 Update 2 [1.4.0_02]
    - J2SE 1.4.0 Update 3 [1.4.0_03]
    - J2SE 1.4.0 Update 4 [1.4.0_04-b04]
  - J2SE 1.4.1 Hopper 2002-09-16
    - J2SE 1.4.1 [1.4.1-b21]
    - J2SE 1.4.1 Update 1 [1.4.1_01-b01]
    - J2SE 1.4.1 Update 2 [1.4.1_02]
    - J2SE 1.4.1 Update 3 [1.4.1_03]
    - J2SE 1.4.1 Update 4 [1.4.1_04]
    - J2SE 1.4.1 Update 5 [1.4.1_05]
    - J2SE 1.4.1 Update 6 [1.4.1_06]
    - J2SE 1.4.1 Update 7 [1.4.1_07-b02]
  - J2SE 1.4.2 Mantis 2003-06-26
    - J2SE 1.4.2 [1.4.2-b28]
    - J2SE 1.4.2 Update 1 [1.4.2_01]
    - J2SE 1.4.2 Update 2 [1.4.2_02]
    - J2SE 1.4.2 Update 3 [1.4.2_03]
    - J2SE 1.4.2 Update 4 [1.4.2_04]
    - J2SE 1.4.2 Update 5 [1.4.2_05]
    - J2SE 1.4.2 Update 6 [1.4.2_06]
    - J2SE 1.4.2 Update 7 [1.4.2_07]
    - J2SE 1.4.2 Update 8 [1.4.2_08-b03]
    - J2SE 1.4.2 Update 9 [1.4.2_09-b05]
    - J2SE 1.4.2 Update 10 [1.4.2_10-b03]
    - J2SE 1.4.2 Update 11 [1.4.2_11-b06]
    - J2SE 1.4.2 Update 12 [1.4.2_12-b03]
    - J2SE 1.4.2 Update 13 [1.4.2_13-b03]
    - J2SE 1.4.2 Update 14 [1.4.2_14-b05]
    - J2SE 1.4.2 Update 15 [1.4.2_15-b02]
    - J2SE 1.4.2 Update 16 [1.4.2_16-b01]
    - J2SE 1.4.2 Update 17 [1.4.2_17-b06]
    - J2SE 1.4.2 Update 18 [1.4.2_18-b06]
    - J2SE 1.4.2 Update 19 [1.4.2_19-b04]

Sun JDK版本

- J2SE 5.0
  - J2SE 5.0 (1.5.0) Tiger 2004-09-29
    - J2SE 5.0 [1.5.0-b64]
    - J2SE 5.0 Update 1 [1.5.0_01]
    - J2SE 5.0 Update 2 [1.5.0_02-b09]
    - J2SE 5.0 Update 3 [1.5.0_03-b07]
    - J2SE 5.0 Update 4 [1.5.0_04-b05]
    - J2SE 5.0 Update 5 [1.5.0_05-b05]
    - J2SE 5.0 Update 6 [1.5.0_06-b05]
    - J2SE 5.0 Update 7 [1.5.0_07-b03]
    - J2SE 5.0 Update 8 [1.5.0_08-b03]
    - J2SE 5.0 Update 9 [1.5.0_09-b03]
    - J2SE 5.0 Update 10 [1.5.0_10-b02]
    - J2SE 5.0 Update 11 [1.5.0_11-b03]
    - J2SE 5.0 Update 12 [1.5.0_12-b04]
    - J2SE 5.0 Update 13 [1.5.0_13-b01]
    - J2SE 5.0 Update 14 [1.5.0_14-b03]
    - J2SE 5.0 Update 15 [1.5.0_15-b04]
    - J2SE 5.0 Update 16 [1.5.0_16-b02]
    - J2SE 5.0 Update 17 [1.5.0_17-b04]
    - J2SE 5.0 Update 18 [1.5.0_18-b02]
    - J2SE 5.0 Update 19 [1.5.0_19-b02]
    - J2SE 5.0 Update 20 [1.5.0_20-b02]
    - J2SE 5.0 Update 21 [1.5.0_21-b01]
    - J2SE 5.0 Update 22 [1.5.0_22-b03]
  - J2SE 5.1 (1.5.1) Dragonfly 未发布

Oracle/Sun JDK版本

- Java SE 6
  - Java SE 6 (1.6.0) Tiger 2006-12

Java SE 6	[1.6.0-b105]	(HotSpot 1.6.0-b105)	2006
Java SE 6 Update 1	[1.6.0_01-b06]	(HotSpot 1.6.0_01-b06)	2007
Java SE 6 Update 2	[1.6.0_02-b05/b06]	(HotSpot 1.6.0_02-b05/b06)	2007
Java SE 6 Update 3	[1.6.0_03-b05]	(HotSpot 1.6.0_03-b05)	2007
Java SE 6 Update 4	[1.6.0_04-b12]	(HotSpot 10.0-b19)	2007
Java SE 6 Update 5	[1.6.0_05-b13]	(HotSpot 10.0-b19)	2008
Java SE 6 Update 6	[1.6.0_06-b02]	(HotSpot 10.0-b22)	2008
Java SE 6 Update 7	[1.6.0_07-b06]	(HotSpot 10.0-b23)	2008
Java SE 6 Update 10	[1.6.0_10-b33]	(HotSpot 11.0-b15)	2008-10-15
Java SE 6 Update 11	[1.6.0_11-b03]	(HotSpot 11.0-b16)	2008
Java SE 6 Update 12	[1.6.0_12-b04]	(HotSpot 11.2-b01)	2009-02-02
Java SE 6 Update 13	[1.6.0_13-b03]	(HotSpot 11.3-b02)	2009
Java SE 6 Update 14	[1.6.0_14-b08]	(HotSpot 14.0-b16)	2009-05-28
Java SE 6 Update 15	[1.6.0_15-b03]	(HotSpot 14.1-b02)	2009-08-04
Java SE 6 Update 16	[1.6.0_16-b01]	(HotSpot 14.2-b01)	2009
Java SE 6 Update 17	[1.6.0_17-b04]	(HotSpot 14.3-b01)	2009
Java SE 6 Update 18	[1.6.0_18-b07]	(HotSpot 16.0-b13)	2010-01-13
Java SE 6 Update 19	[1.6.0_19-b04]	(HotSpot 16.2-b04)	2010-03
Java SE 6 Update 20	[1.6.0_20-b02]	(HotSpot 16.3-b01)	2010-04-15
Java SE 6 Update 21	[1.6.0_21-b06/07]	(HotSpot 17.0-b17)	2010-07-10
Java SE 6 Update 22	[1.6.0_22-b04]	(HotSpot 17.1-b03)	2010-10-22
Java SE 6 Update 23	[1.6.0_23-b05]	(HotSpot 19.0-b09)	2010-12-07
Java SE 6 Update 24	[1.6.0_24-b07]	(HotSpot 19.1-b02)	2011-02-16
Java SE 6 Update 25	[1.6.0_25-b06]	(HotSpot 20.0-b11)	2011-04-21
Java SE 6 Update 26	[1.6.0_26-b03]	(HotSpot 20.1-b02)	2011-06-08

  - HotSpot VM的开发周期从JDK 6 Update 3之后与JDK开始分离，两者不再维持版本号同步

Oracle JDK版本

- Java SE 7
  - Java SE 7 (1.7.0) Dolphin 2011-07-28
    - `Java SE 7 [1.7.0-b147] (HotSpot 21.0-b17) 2011-07-28`

OpenJDK与Oracle/Sun JDK版本的对应关系

- https://gist.github.com/925323
- http://dbhole.wordpress.com/2011/05/27/why-do-xx-and-yy-in-jdk6-uxx-and-openjdk-byy-differ/