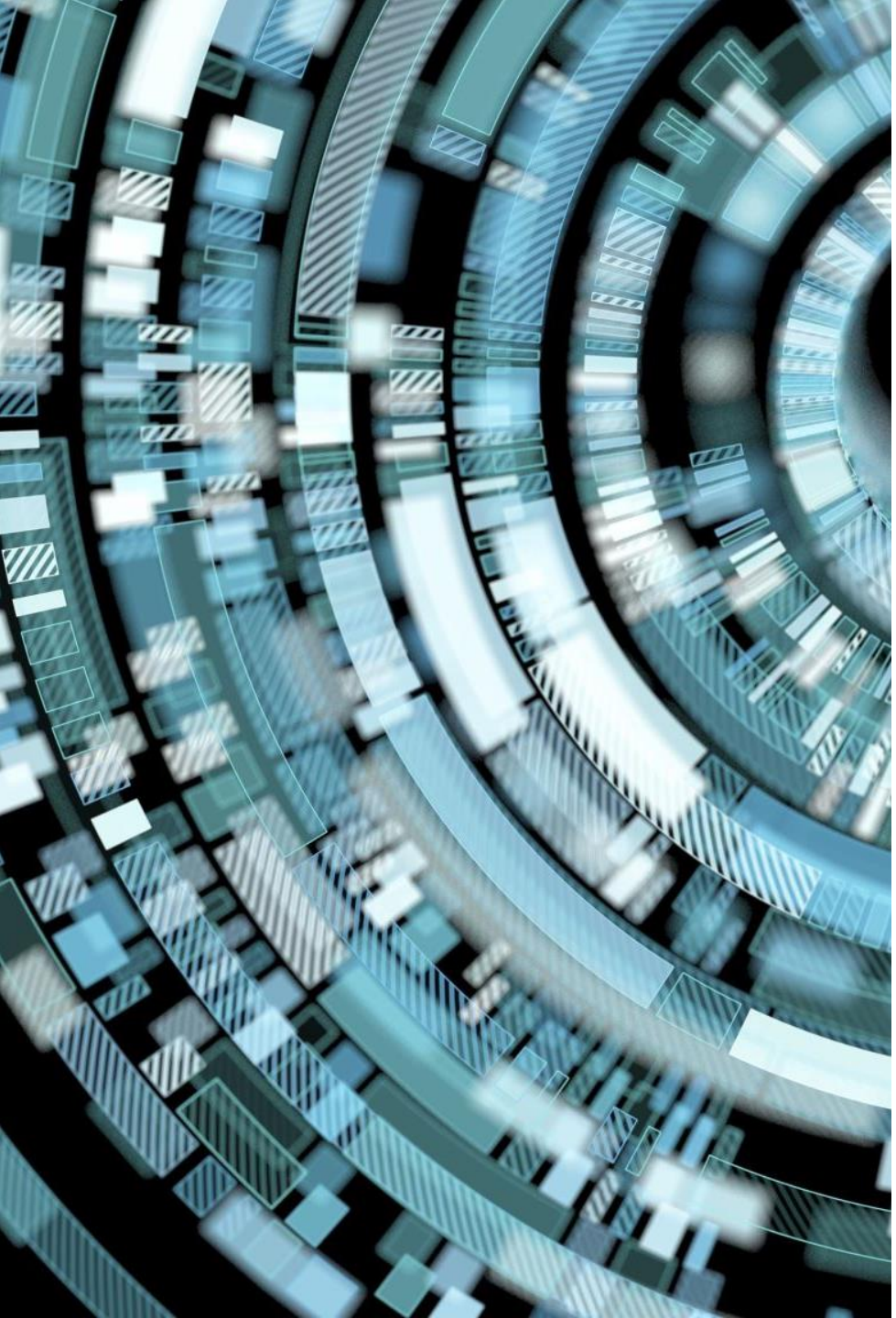
Computer System Design & Application

计算机系统设计与应用A

陶伊达 （TAO Yida）
taoyd@sustech.edu.cn



Lecture 14

- `.class` structure
- JVM

`.class`: Platform Independence

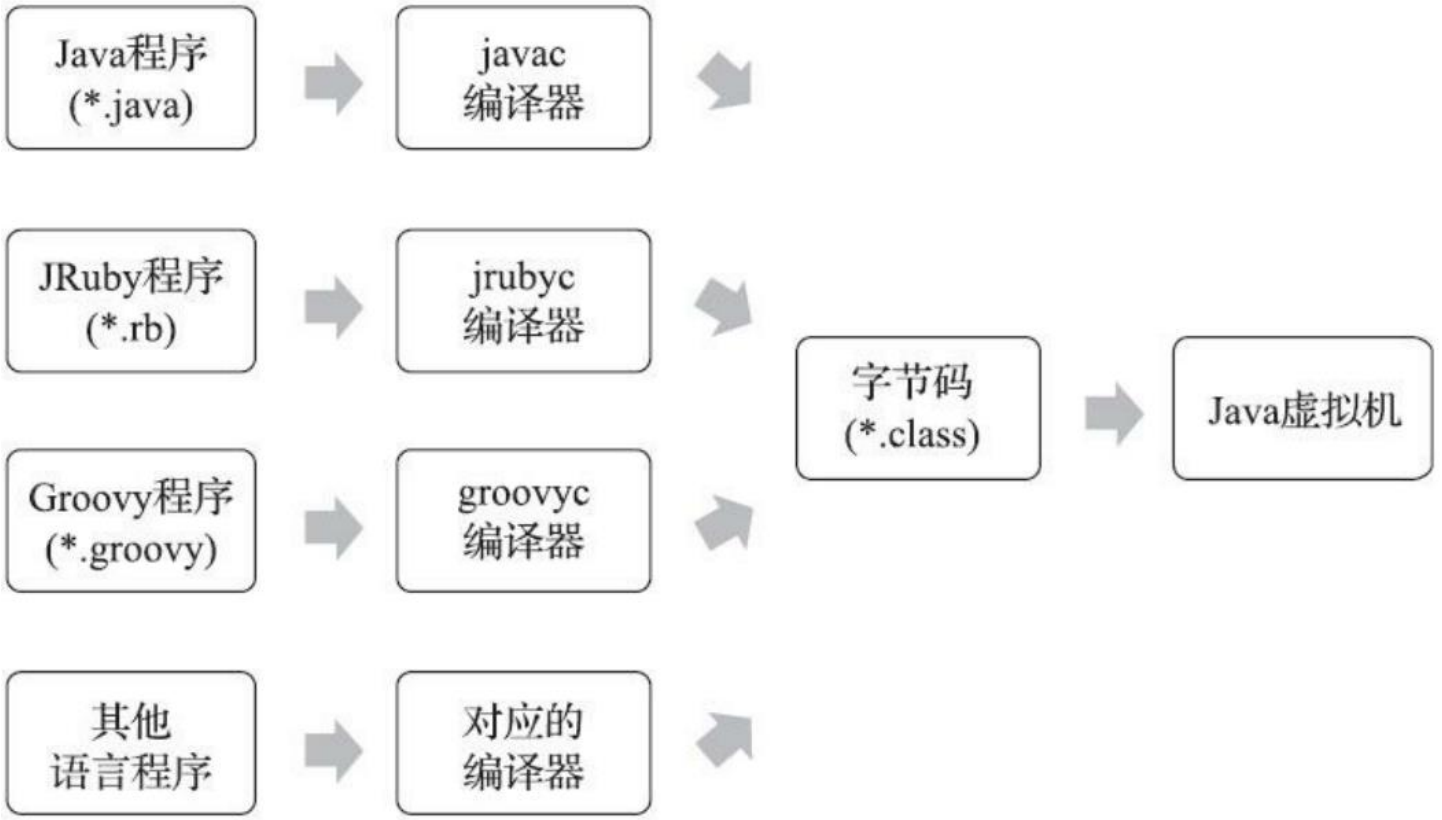
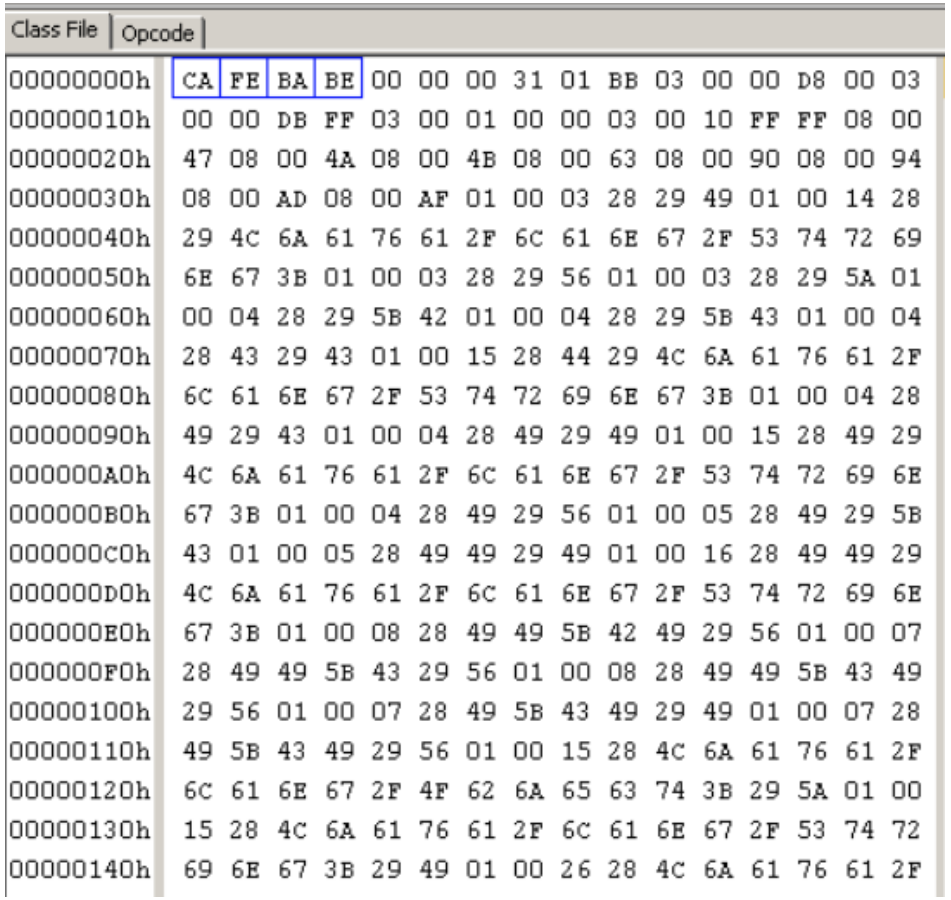
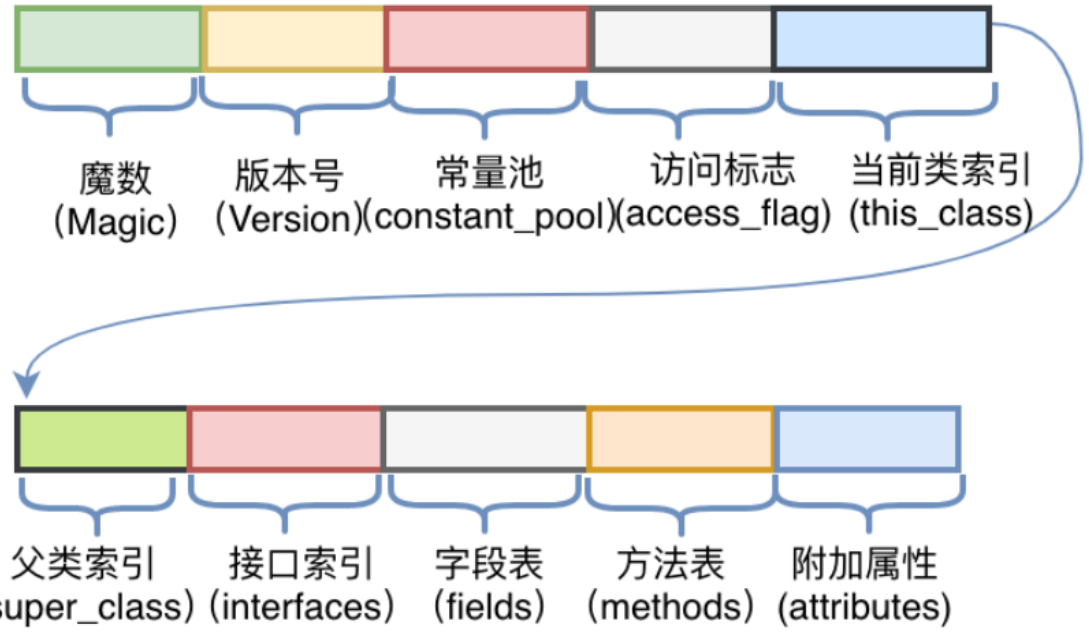


Image source: 《深入理解Java虚拟机》第三版，周志明

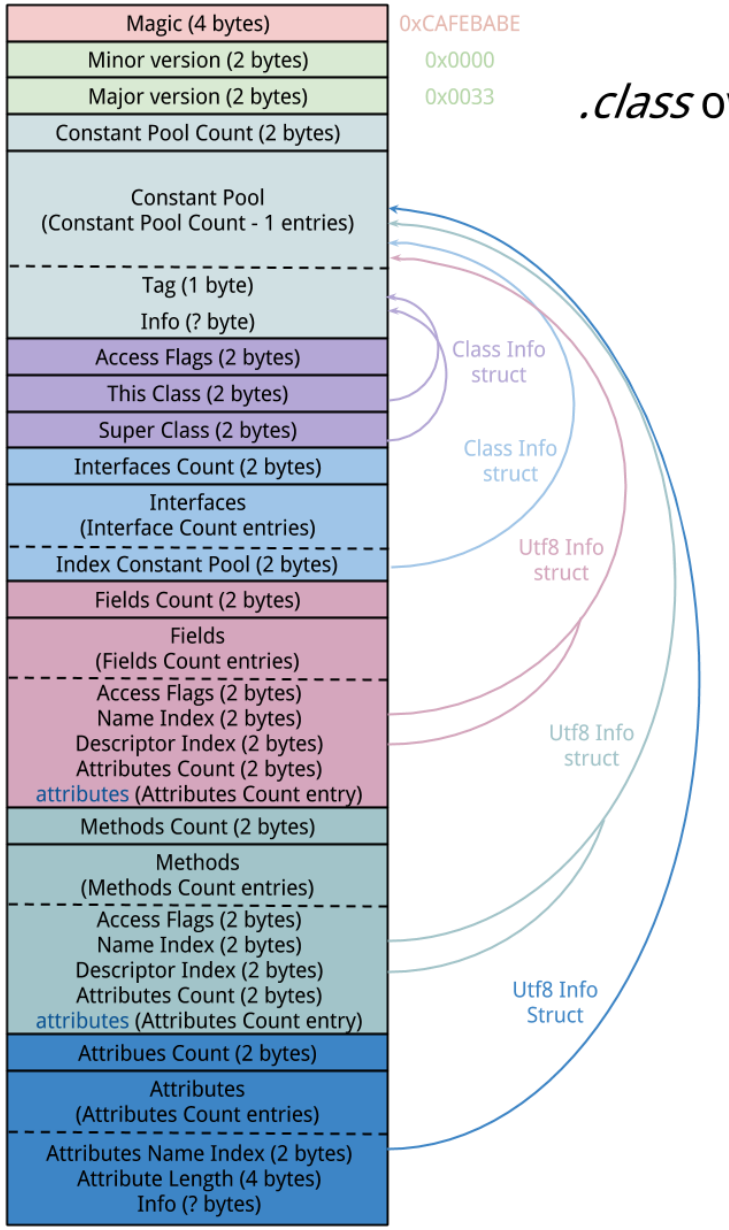
`.class`: Platform Independence



`.class` file format



https://github.com/likuisuper/Java-Notes/



https://blog.lse.epita.fr/2014/04/28/0xcafebabe-java-class-file-format-an-overview.html

Version Number

Java is backward compatible: you can run Java 15 compiled jar on JRE 16 but not vice versa.

`java.lang.UnsupportedClassVersionError (version 60.0)` happens because of a higher JDK (JDK 16) during compile time and lower JDK during runtime.

Java SE	Major Version
18	62
17	61
16	60
15	59
14	58
13	57
12	56
11	55
10	54
9	53
8	52
7	51
6.0	50
5.0	49
1.4	48
1.3	47
1.2	46
1.1	45

Constant Pool

常量池存放的是**字面量**和**符号引用**。

- 字面量接近于java语言中被final修饰的常量、文本字符
- 符号引用如类和接口的全限定名、字段的名称和描述符、方法的名称和描述符等

https://github.com/likuisuper/Java-Notes/

```
Constant pool:
   #1 = Methodref          #6.#23         // java/lang/Object."<init>":()V
   #2 = Fieldref           #24.#25        // java/lang/System.out:Ljava/io/PrintStream;
   #3 = String             #26            // hello word
   #4 = Methodref          #27.#28        // java/io/PrintStream.println:(Ljava/lang/String;)V
   #5 = Class              #29            // com/cxylk/partthree/ClassFileDemo
   #6 = Class              #30            // java/lang/Object
   #7 = Utf8               v9
   #8 = Utf8               [I
   #9 = Utf8               v10
  #10 = Utf8               [Ljava/lang/String;
  #11 = Utf8               <init>
  #12 = Utf8               ()V
  #13 = Utf8               Code
  #14 = Utf8               LineNumberTable
  #15 = Utf8               LocalVariableTable
  #16 = Utf8               this
  #17 = Utf8               Lcom/cxylk/partthree/ClassFileDemo;
  #18 = Utf8               main
  #19 = Utf8               ([Ljava/lang/String;)V
  #20 = Utf8               args
  #21 = Utf8               SourceFile
  #22 = Utf8               ClassFileDemo.java
  #23 = NameAndType        #11:#12        // "<init>":()V
  #24 = Class              #31            // java/lang/System
  #25 = NameAndType        #32:#33        // out:Ljava/io/PrintStream;
  #26 = Utf8               hello word
  #27 = Class              #34            // java/io/PrintStream
  #28 = NameAndType        #35:#36        // println:(Ljava/lang/String;)V
  #29 = Utf8               com/cxylk/partthree/ClassFileDemo
  #30 = Utf8               java/lang/Object
```

Constant Pool

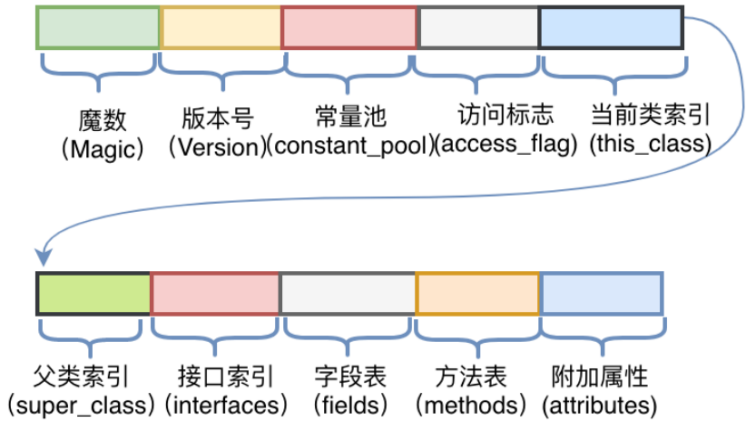
```
public class ClassFileDemo {
    int[] v9;
    String[] v10;

    public static void main(String[] args) {
        int c;
        System.out.println("hello word");
    }
}
```

https://github.com/likuisuper/Java-Notes/

```
Constant pool:
   #1 = Methodref          #6.#23         // java/lang/Object."<init>":()V
   #2 = Fieldref           #24.#25        // java/lang/System.out:Ljava/io/PrintStream;
   #3 = String             #26            // hello word
   #4 = Methodref          #27.#28        // java/io/PrintStream.println:(Ljava/lang/String;)V
   #5 = Class              #29            // com/cxylk/partthree/ClassFileDemo
   #6 = Class              #30            // java/lang/Object
   #7 = Utf8               v9
   #8 = Utf8               [I
   #9 = Utf8               v10
  #10 = Utf8               [Ljava/lang/String;
  #11 = Utf8               <init>
  #12 = Utf8               ()V
  #13 = Utf8               Code
  #14 = Utf8               LineNumberTable
  #15 = Utf8               LocalVariableTable
  #16 = Utf8               this
  #17 = Utf8               Lcom/cxylk/partthree/ClassFileDemo;
  #18 = Utf8               main
  #19 = Utf8               ([Ljava/lang/String;)V
  #20 = Utf8               args
  #21 = Utf8               SourceFile
  #22 = Utf8               ClassFileDemo.java
  #23 = NameAndType        #11:#12        // "<init>":()V
  #24 = Class              #31            // java/lang/System
  #25 = NameAndType        #32:#33        // out:Ljava/io/PrintStream;
  #26 = Utf8               hello word
  #27 = Class              #34            // java/io/PrintStream
  #28 = NameAndType        #35:#36        // println:(Ljava/lang/String;)V
  #29 = Utf8               com/cxylk/partthree/ClassFileDemo
  #30 = Utf8               java/lang/Object
```

Access Flags



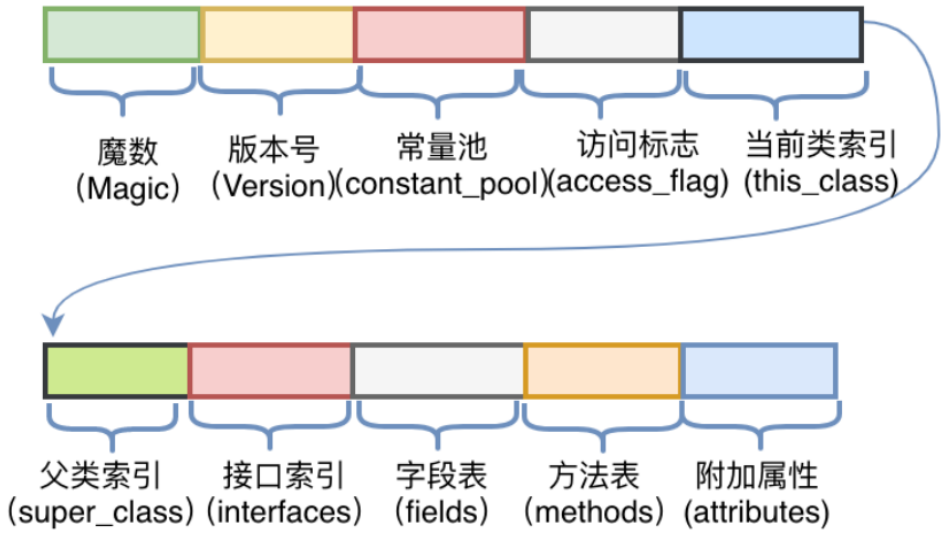
Class Flags
ACC_PUBLIC (0x0001)
ACC_FINAL (0x0010)
ACC_SUPER (0x0020)
ACC_INTERFACE (0x0200)
ACC_ABSTRACT (0x0400)
ACC_SYNTHETIC (0x1000)
ACC_ANNOTATION (0x2000)
ACC_ENUM (0x4000)

Fields Flags
ACC_PUBLIC (0x0001)
ACC_PRIVATE (0x0002)
ACC_PROTECTED (0x0004)
ACC_STATIC (0x0008)
ACC_FINAL (0x0010)
ACC_VOLATILE (0x0040)
ACC_TRANSIENT (0x0080)
ACC_SYNTHETIC (0x1000)
ACC_ENUM (0x4000)

Methods Flags
ACC_PUBLIC (0x0001)
ACC_PRIVATE (0x0002)
ACC_PROTECTED (0x0004)
ACC_FINAL (0x0010)
ACC_SYNCHRONIZED (0x0020)
ACC_BRIDGE (0x0040)
ACC_VARARGS (0x0080)
ACC_NATIVE (0x0100)
ACC_ABSTRACT (0x0400)
ACC_STRICT (0x0800)
ACC_SYNTHETIC (0x1000)

https://blog.lse.epita.fr/2014/04/28/0xcafebabe-java-class-file-format-an-overview.html

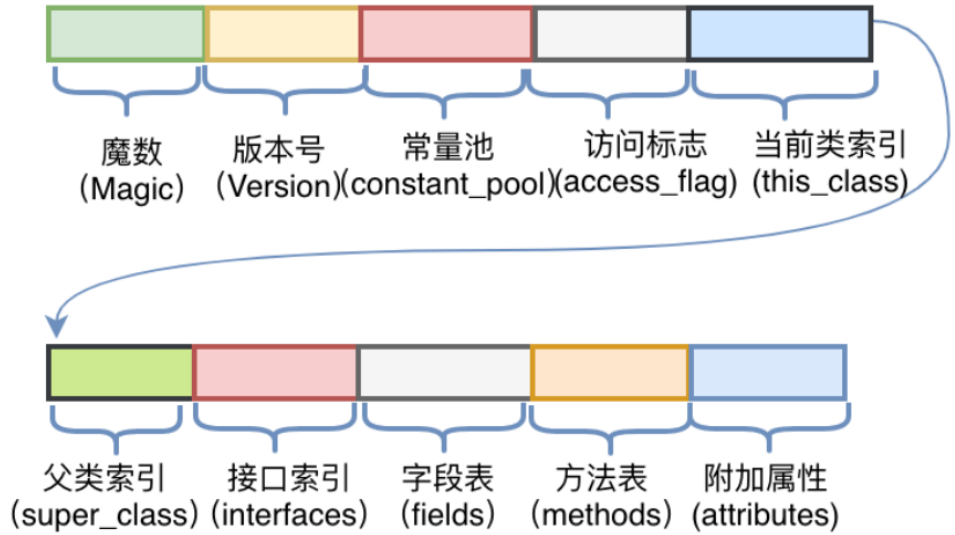
Class, Superclass, Interfaces

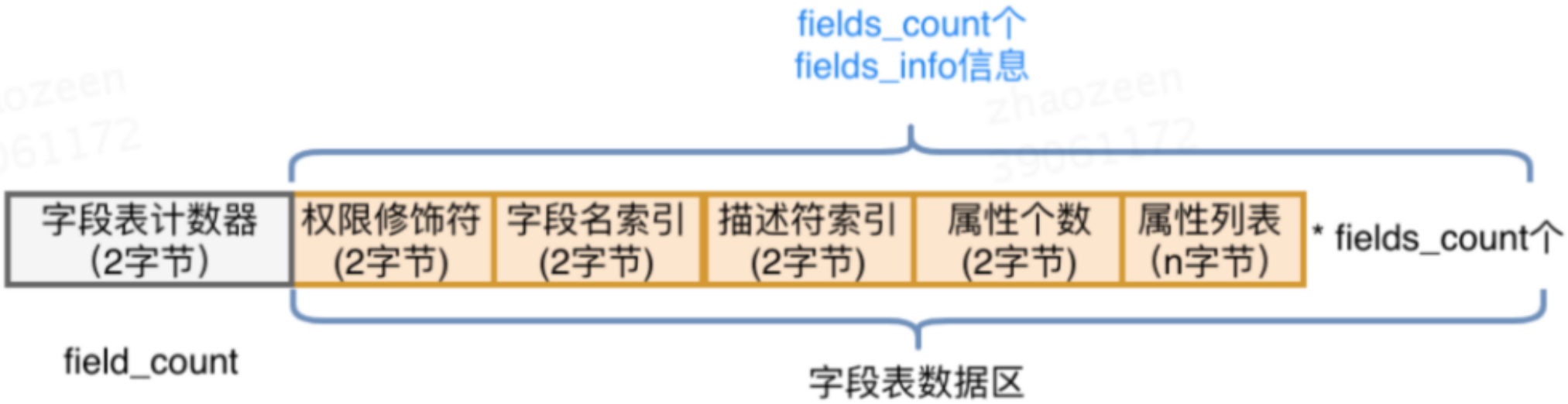


https://github.com/likuisuper/Java-Notes/

```
Constant pool:
   #1 = Methodref          #6.#23         // java/lang/Object."<init>":()V
   #2 = Fieldref           #24.#25        // java/lang/System.out:Ljava/io/PrintStream;
   #3 = String             #26            // hello word
   #4 = Methodref          #27.#28        // java/io/PrintStream.println:(Ljava/lang/String;)V
   #5 = Class              #29            // com/cxylk/partthree/ClassFileDemo
   #6 = Class              #30            // java/lang/Object
   #7 = Utf8               v9
   #8 = Utf8               [I
   #9 = Utf8               v10
  #10 = Utf8               [Ljava/lang/String;
  #11 = Utf8               <init>
  #12 = Utf8               ()V
  #13 = Utf8               Code
  #14 = Utf8               LineNumberTable
  #15 = Utf8               LocalVariableTable
  #16 = Utf8               this
  #17 = Utf8               Lcom/cxylk/partthree/ClassFileDemo;
  #18 = Utf8               main
  #19 = Utf8               ([Ljava/lang/String;)V
  #20 = Utf8               args
  #21 = Utf8               SourceFile
  #22 = Utf8               ClassFileDemo.java
  #23 = NameAndType        #11:#12        // "<init>":()V
  #24 = Class              #31            // java/lang/System
  #25 = NameAndType        #32:#33        // out:Ljava/io/PrintStream;
  #26 = Utf8               hello word
  #27 = Class              #34            // java/io/PrintStream
  #28 = NameAndType        #35:#36        // println:(Ljava/lang/String;)V
  #29 = Utf8               com/cxylk/partthree/ClassFileDemo
  #30 = Utf8               java/lang/Object
```

Fields



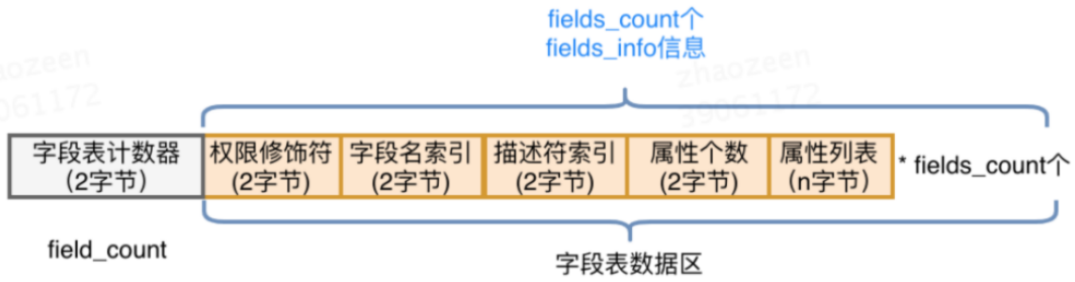


https://github.com/likuisuper/Java-Notes/

Fields

```
public class ClassFileDemo {
    int[] v9;
    String[] v10;

    public static void main(String[] args) {
        int c;
        System.out.println("hello word");
    }
}
```

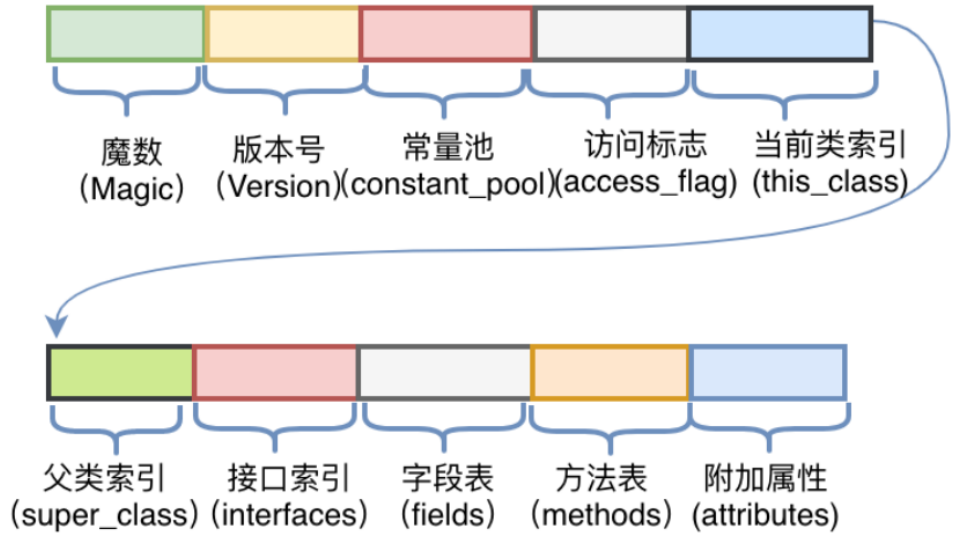


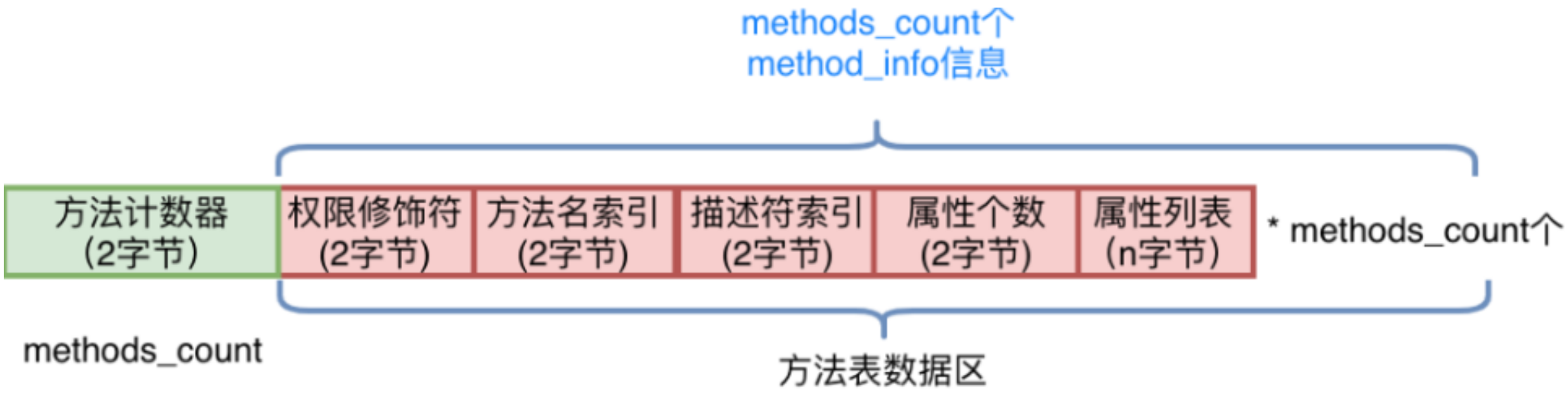
0002	0000	0007	0008	0000
字段表计数器，2个字段	访问标志，因为没有使用到标志位，所以为0	字段名称，#7，查常量表得v9	字段描述，#8，查常量表为[I	字段属性个数为0

```
Constant pool:
   #1 = Methodref          #6.#23         // java/lang/Object."<init>":()V
   #2 = Fieldref           #24.#25        // java/lang/System.out:Ljava/io/PrintStream;
   #3 = String             #26            // hello word
   #4 = Methodref          #27.#28        // java/io/PrintStream.println:(Ljava/lang/String;)V
   #5 = Class              #29            // com/cxylk/partthree/ClassFileDemo
   #6 = Class              #30            // java/lang/Object
   #7 = Utf8               v9
   #8 = Utf8               [I
   #9 = Utf8               v10
  #10 = Utf8               [Ljava/lang/String;
  #11 = Utf8               <init>
  #12 = Utf8               ()V
  #13 = Utf8               Code
  #14 = Utf8               LineNumberTable
  #15 = Utf8               LocalVariableTable
  #16 = Utf8               this
  #17 = Utf8               Lcom/cxylk/partthree/ClassFileDemo;
  #18 = Utf8               main
  #19 = Utf8               ([Ljava/lang/String;)V
  #20 = Utf8               args
  #21 = Utf8               SourceFile
  #22 = Utf8               ClassFileDemo.java
  #23 = NameAndType        #11:#12        // "<init>":()V
  #24 = Class              #31            // java/lang/System
  #25 = NameAndType        #32:#33        // out:Ljava/io/PrintStream;
  #26 = Utf8               hello word
  #27 = Class              #34            // java/io/PrintStream
  #28 = NameAndType        #35:#36        // println:(Ljava/lang/String;)V
  #29 = Utf8               com/cxylk/partthree/ClassFileDemo
  #30 = Utf8               java/lang/Object
```

https://github.com/likuisuper/Java-Notes/

Methods





https://github.com/likuisuper/Java-Notes/

Methods

```
public class ClassFileDemo {
    int[] v9;
    String[] v10;

    public static void main(String[] args) {
        int c;
        System.out.println("hello word");
    }
}
```

```
Constant pool:
   #1 = Methodref          #6.#23         // java/lang/Object."<init>":()V
   #2 = Fieldref           #24.#25        // java/lang/System.out:Ljava/io/PrintStream;
   #3 = String             #26            // hello word
   #4 = Methodref          #27.#28        // java/io/PrintStream.println:(Ljava/lang/String;)V
   #5 = Class              #29            // com/cxylk/partthree/ClassFileDemo
   #6 = Class              #30            // java/lang/Object
   #7 = Utf8               v9
   #8 = Utf8               [I
   #9 = Utf8               v10
  #10 = Utf8               [Ljava/lang/String;
  #11 = Utf8               <init>
  #12 = Utf8               ()V
  #13 = Utf8               Code
  #14 = Utf8               LineNumberTable
  #15 = Utf8               LocalVariableTable
  #16 = Utf8               this
  #17 = Utf8               Lcom/cxylk/partthree/ClassFileDemo;
  #18 = Utf8               main
  #19 = Utf8               ([Ljava/lang/String;)V
  #20 = Utf8               args
  #21 = Utf8               SourceFile
  #22 = Utf8               ClassFileDemo.java
  #23 = NameAndType        #11:#12        // "<init>":()V
  #24 = Class              #31            // java/lang/System
```

0002	0001	000B	000C	0001	000D
方法个数为2	由上图可知01表示public	方面名称，索引是#11，对应值为 `<init>`	方法描述，索引是#12，对应值为()V	1个属性	attribute_name_index，索引为#13，对应值为Code

https://github.com/likuisuper/Java-Notes/

Attributes

Each field, method and class have other information, which are contained in attributes:

- Code
- Local variables, constant value, information about the stack and exceptions
- Inner Classes, Bootstrap Methods
- Annotations
- Complementary information (Deprecated, Signature…)
- ……

https://github.com/likuisuper/Java-Notes/

```
public static void main(java.lang.String[]);
  descriptor: ([Ljava/lang/String;)V
  flags: ACC_PUBLIC, ACC_STATIC
  Code:
    stack=2, locals=2, args_size=1
       0: getstatic     #2                  // Field
       3: ldc           #3                  // String
       5: invokevirtual #4                  // Method
       8: return
    LineNumberTable:
      line 25: 0
      line 26: 8
    LocalVariableTable:
      Start  Length  Slot  Name   Signature
          0       9     0  args   [Ljava/lang/String;
```

Code Attributes

```java
public class Example {
    public static void main(String[] args) {
        int a = 1;
        a++;
        int b = a * a;
        int c = b - a;
        System.out.println(c);
    }
}
```

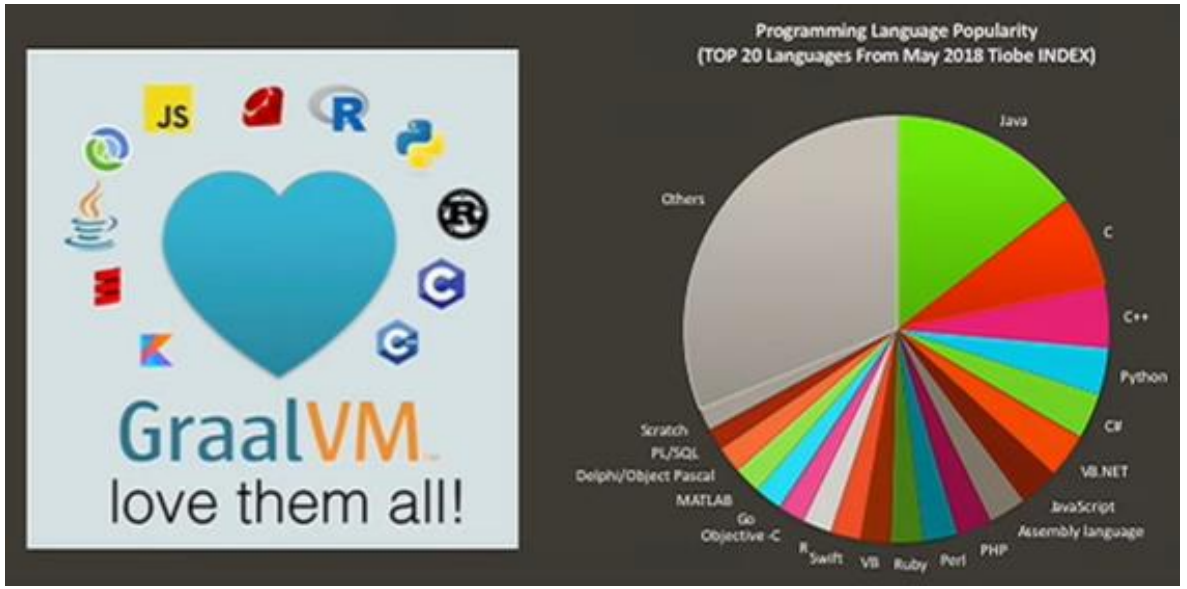
```
    Code:
      stack=2, locals=4, args_size=1
         0: iconst_1
         1: istore_1
         2: iinc          1, 1
         5: iload_1
         6: iload_1
         7: imul
         8: istore_2
         9: iload_2
        10: iload_1
        11: isub
        12: istore_3
        13: getstatic     #2                  // Field
java/lang/System.out:Ljava/io/PrintStream;
        16: iload_3
        17: invokevirtual #3                  // Method
java/io/PrintStream.println:(I)V
        20: return
```

attribute_name_index	attribute_length	max_stack	max_locals	Code_length	code
0x000D	0x0000002F	0x0001	0x0001	0x00000005	2A 87 00 01 B1
固定值="Code"	属性长度	操作数栈最大深度	局部变量表所需存储空间	存储生成的字节码指令	字节码指令

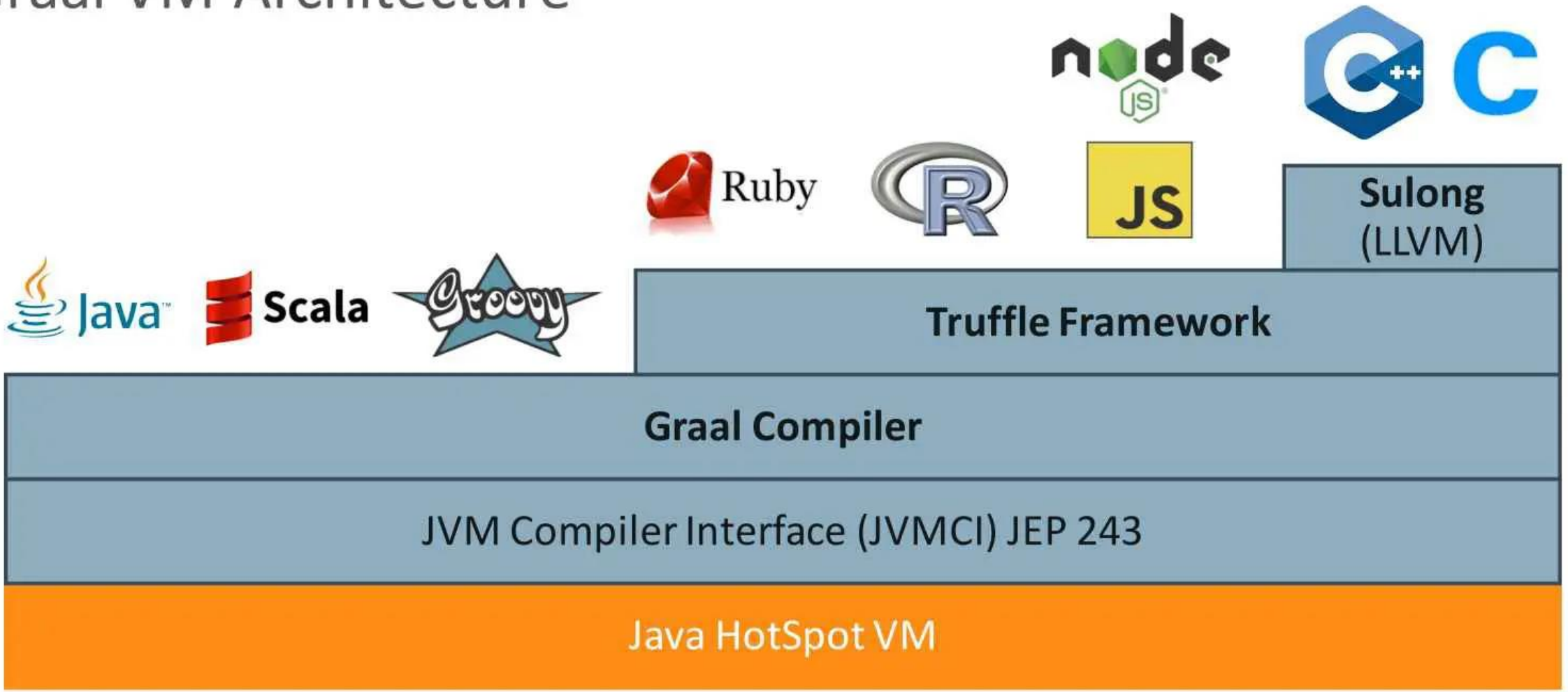
https://github.com/likuisuper/Java-Notes/

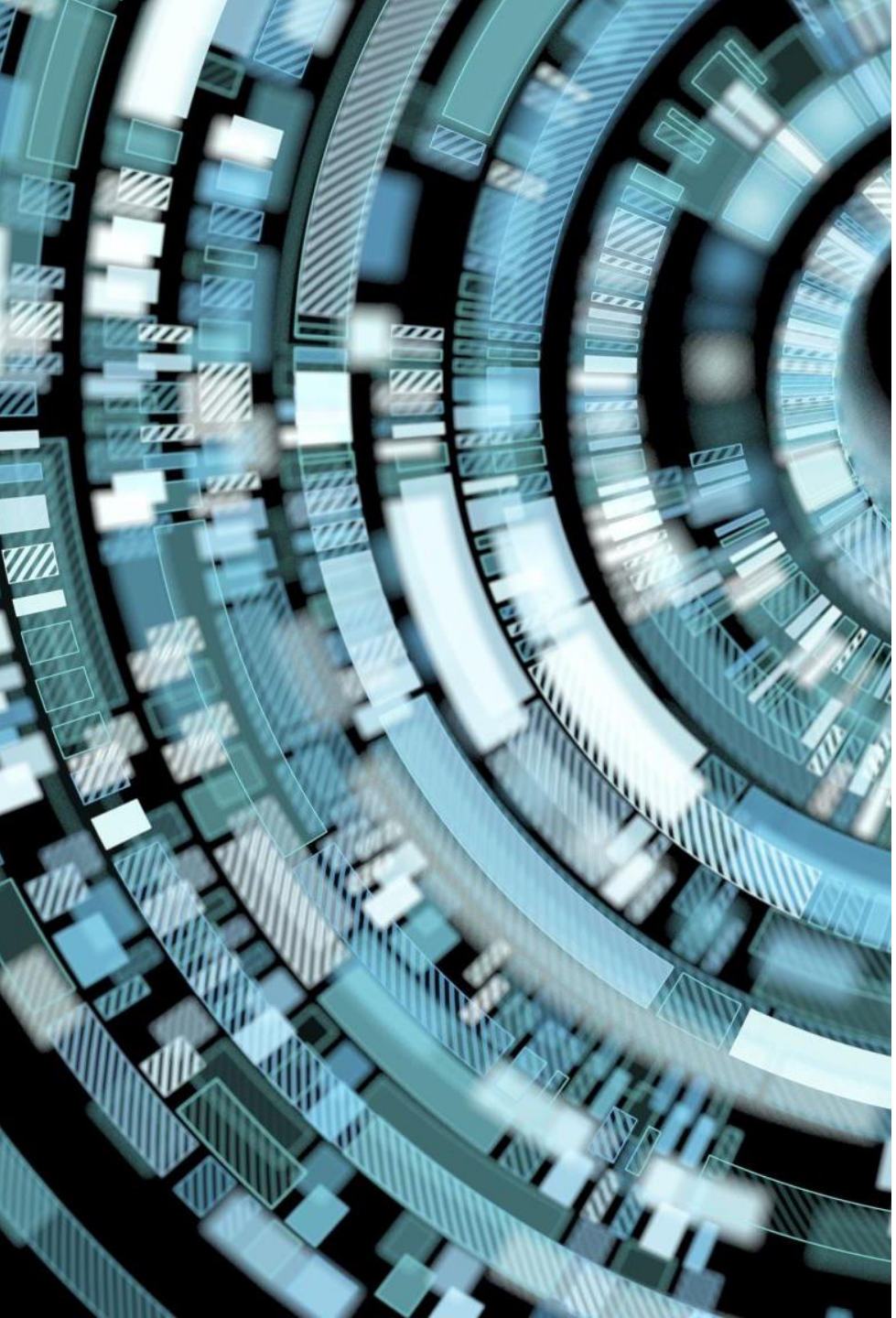
GraalVM


~8 years ago, we started a little research project...
One VM to Rule Them All
Thomas Würthinger* Christian Wimmer* Andreas Wöß† Lukas Stadler†
Gilles Duboscq† Christian Humer† Gregor Richards§ Doug Simon* Mario Wolczko*
*Oracle Labs †Institute for System Software, Johannes Kepler University Linz, Austria §S3 Lab, Purdue University


JS
GraalVM
love them all!
Programming Language Popularity
(TOP 20 Languages From May 2018 Tiobe INDEX)
Java
C
C++
Python
C#
VB.NET
JavaScript
Assembly language
PHP
Perl
Ruby
VB
Swift
R
Objective-C
Go
MATLAB
Delphi/Object Pascal
PL/SQL
Scratch
Others

Graal VM Architecture





Lecture 14

- `.class` structure
- JVM

JVM Architecture

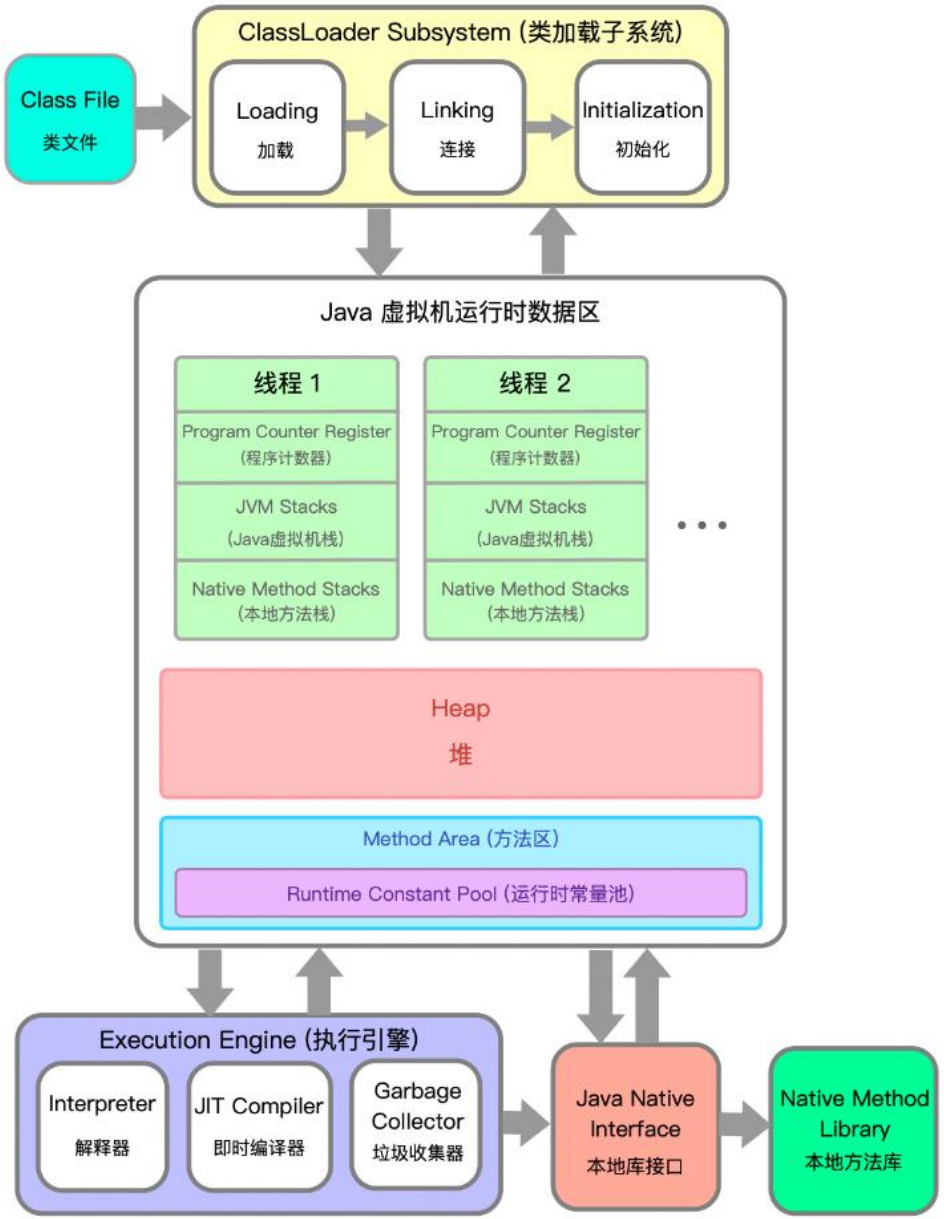
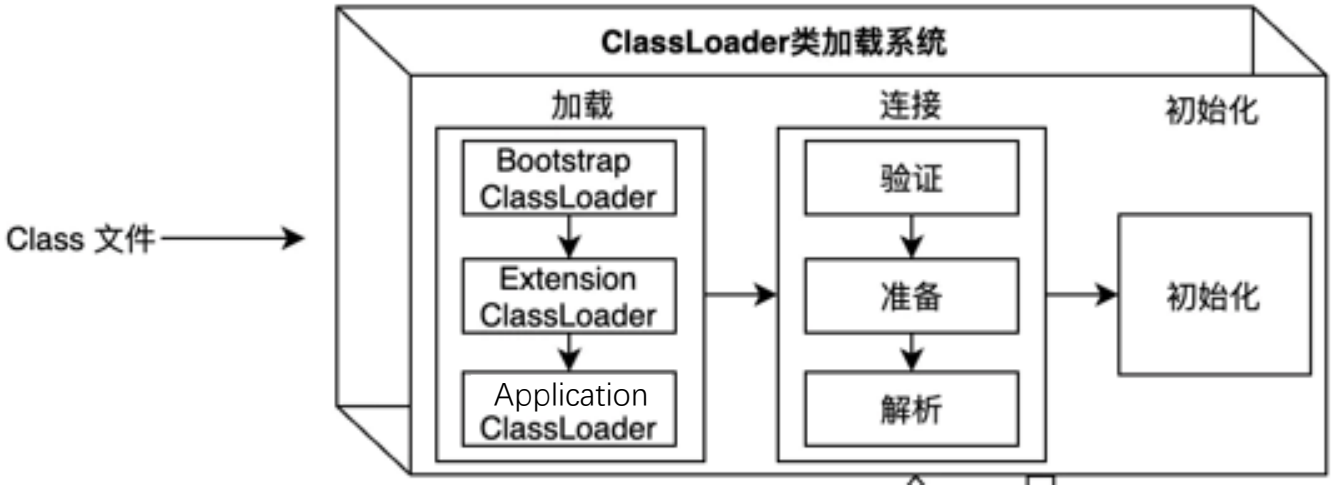


Image source: https://www.cnblogs.com/hynblogs/p/12275957.html

ClassLoader: Loading

- 在加载阶段，Java虚拟机需要完成以下三件事情：
  - 通过一个类的全限定名来获取定义此类的二进制字节流。
  - 将这个字节流所代表的静态存储结构转化为方法区的运行时数据结构。
  - 在内存中生成一个代表这个类的java.lang.Class对象，作为方法区这个类的各种数据的访问入口。



《深入理解Java虚拟机》第三版，周志明

ClassLoader: Loading

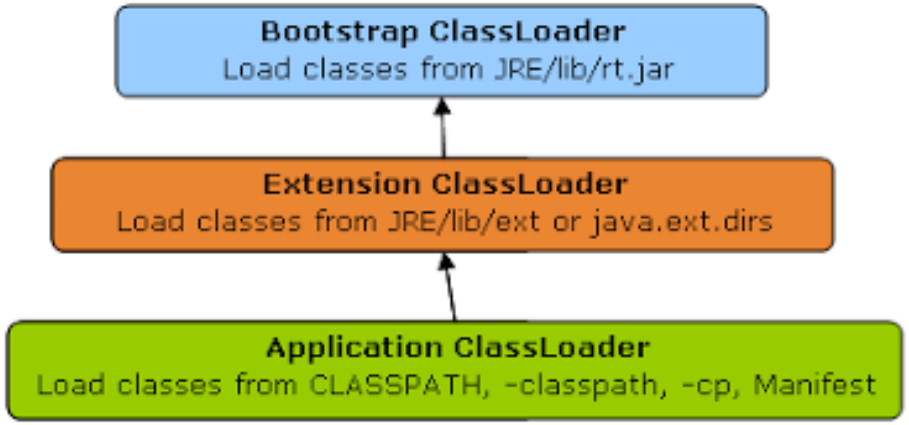
```java
public class HelloApp {
   public static void main(String argv[]) {
      System.out.println("Aloha! Hello and Bye");
   }
}
```

```
prmpt>java -verbose:class HelloApp


[Opened C:\Program Files\Java\jre1.5.0\lib\rt.jar]
[Opened C:\Program Files\Java\jre1.5.0\lib\jsse.jar]
[Opened C:\Program Files\Java\jre1.5.0\lib\jce.jar]
[Opened C:\Program Files\Java\jre1.5.0\lib\charsets.jar]
[Loaded java.lang.Object from shared objects file]
[Loaded java.io.Serializable from shared objects file]
[Loaded java.lang.Comparable from shared objects file]
[Loaded java.lang.CharSequence from shared objects file]
[Loaded java.lang.String from shared objects file]
[Loaded java.lang.reflect.GenericDeclaration from shared objects file]
[Loaded java.lang.reflect.Type from shared objects file]
[Loaded java.lang.reflect.AnnotatedElement from shared objects file]
[Loaded java.lang.Class from shared objects file]
[Loaded java.lang.Cloneable from shared objects file]
[Loaded java.lang.ClassLoader from shared objects file]
[Loaded java.lang.System from shared objects file]
[Loaded java.lang.Throwable from shared objects file]
.
.
.
[Loaded java.security.BasicPermissionCollection from shared objects file]
[Loaded java.security.Principal from shared objects file]
[Loaded java.security.cert.Certificate from shared objects file]
[Loaded HelloApp from file:/C:/classes/]
Aloha! Hello and Bye
[Loaded java.lang.Shutdown from shared objects file]
[Loaded java.lang.Shutdown$Lock from shared objects file]
```

Image source: https://stackoverflow.com/questions/2424604/what-is-a-java-classloader

ClassLoader: Loading



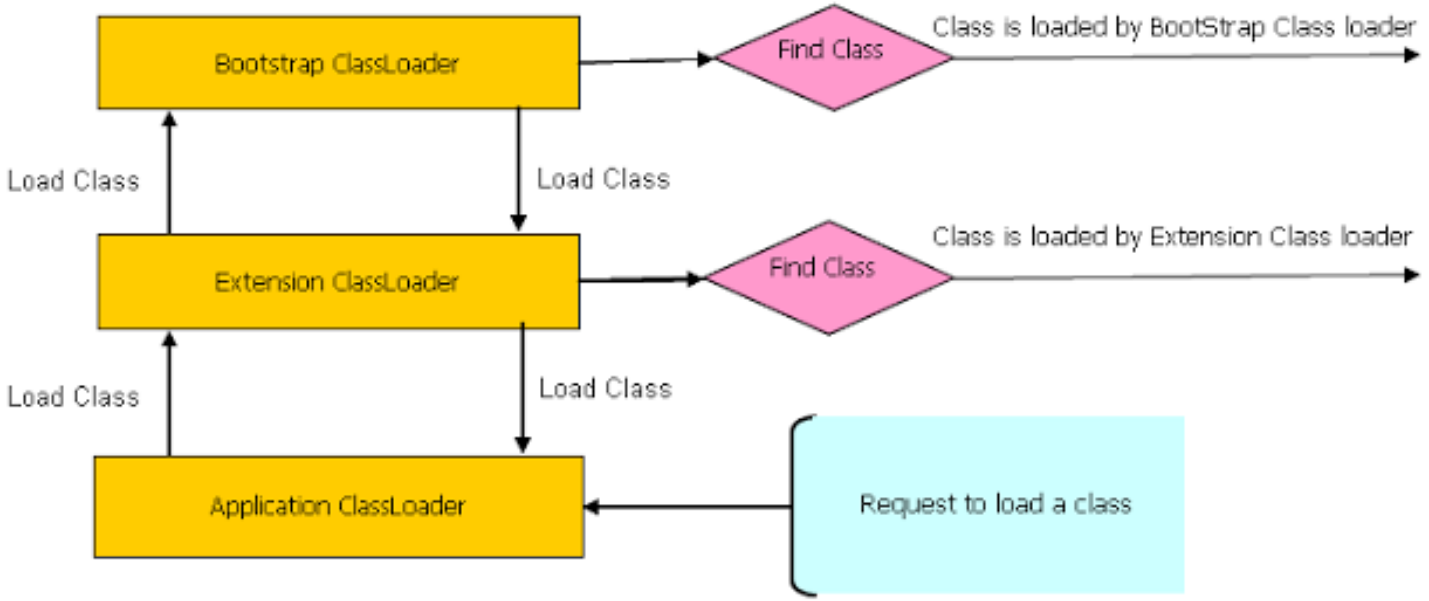


Image source: https://stackoverflow.com/questions/2424604/what-is-a-java-classloader

JVM Runtime Data Areas

JVM defines various run-time data areas that are used during execution of a program

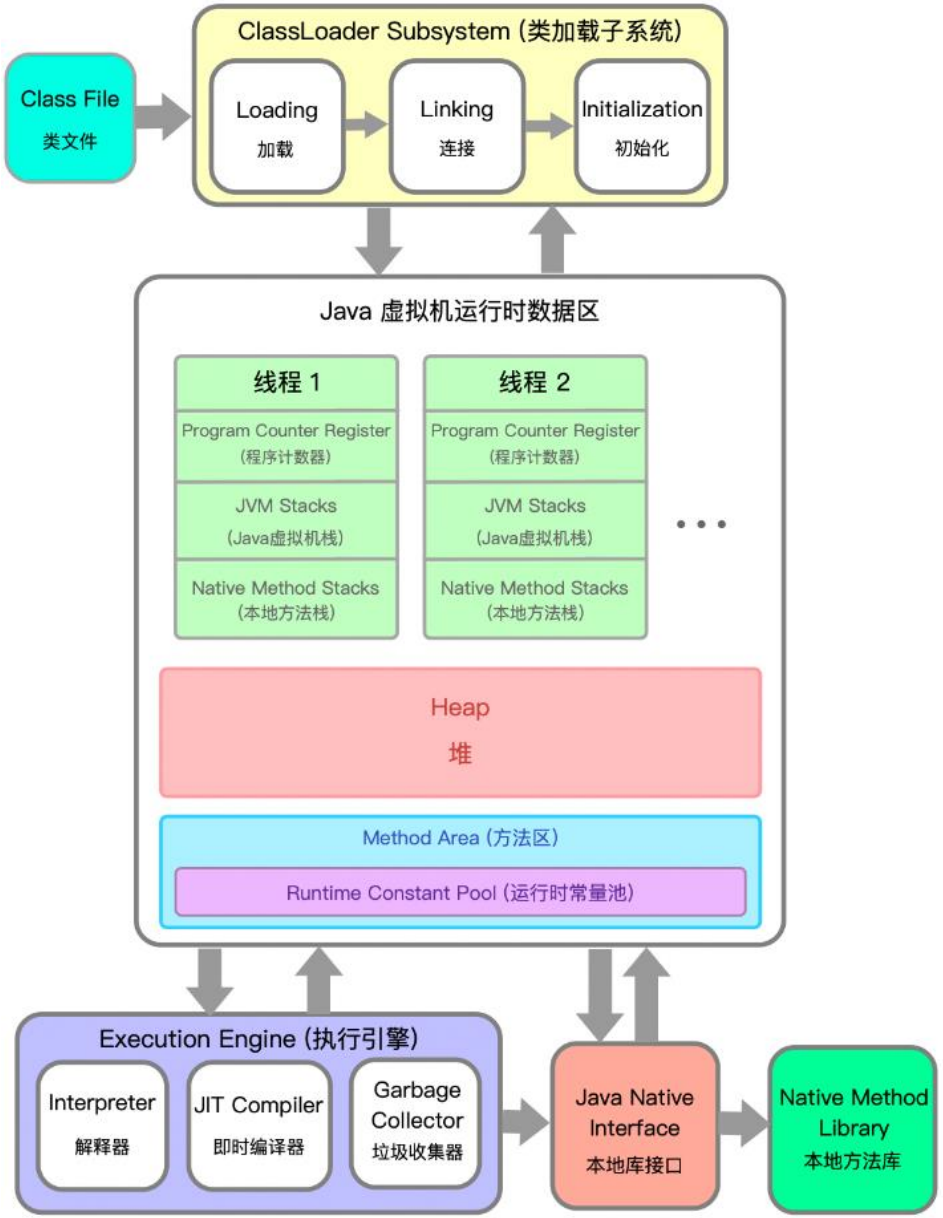
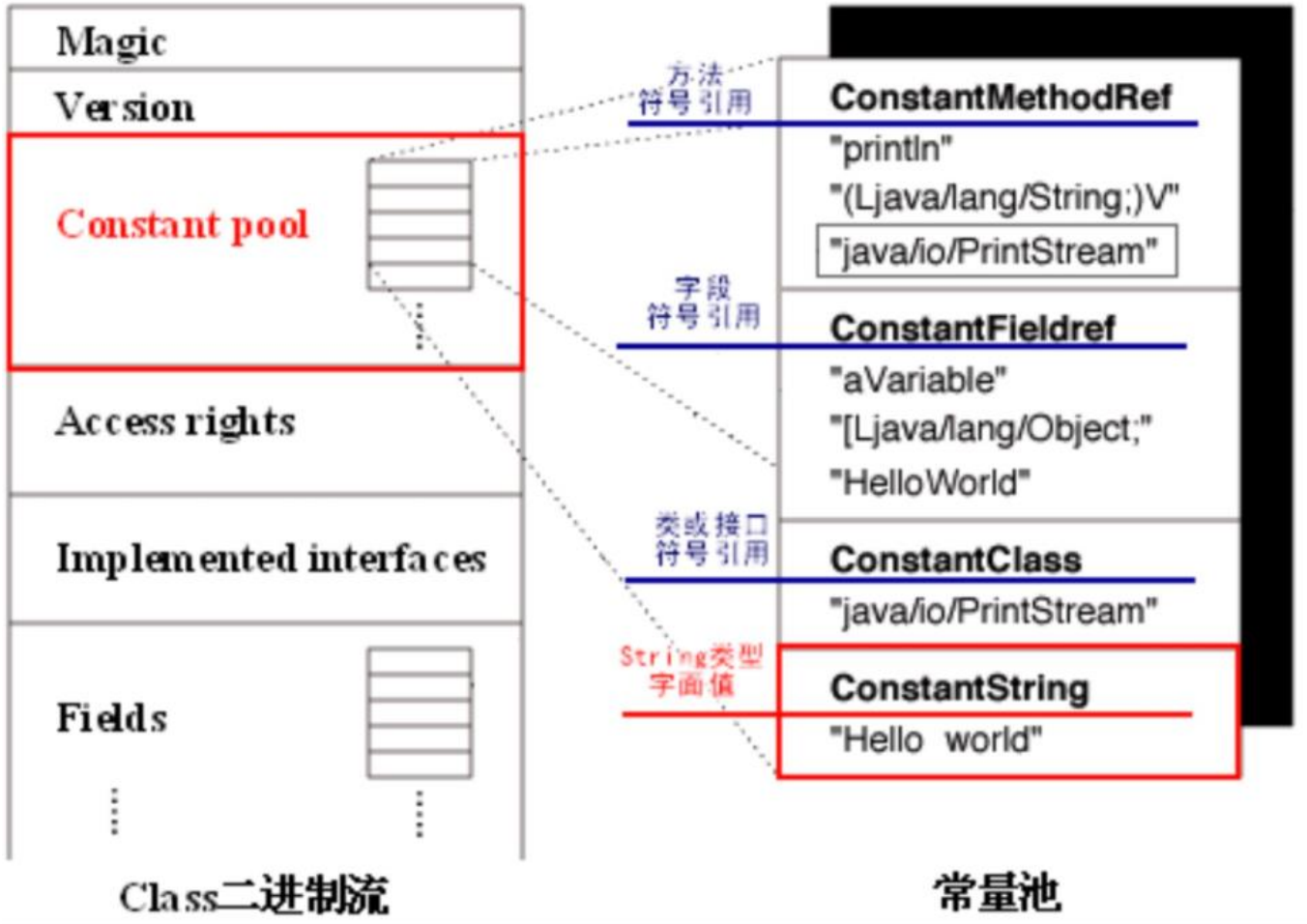


Image source: https://www.cnblogs.com/hynblogs/p/12275957.html

Method Area

- 类型信息: 对每个加载的类型（class、interface、enum、annotation），JVM 方法区中存储以下类型信息：
  - 这个类型的完整有效名称（全名=包名.类名）
  - 这个类型直接父类的完整有效名（对于 interface 或是 java.lang.Object，都没有父类）
  - 这个类型的修饰符（public, abstract, final 的某个子集）
  - 这个类型直接接口的一个有序列表
- 域（Field) 信息
  - JVM 必须在方法区中保存类型的所有域的相关信息以及域的声明顺序。
  - 域的相关信息包括：域名称、域类型、域修饰符（public,private,protected,static,final,volatile,transient 的某个子集）
- 方法（Method) 信息：JVM 必须保存所有方法的以下信息，同域信息一样包括声明顺序：
  - 方法名称
  - 方法的返回类型
  - 方法参数的数量和类型（按顺序）
  - 方法的修饰符（public ,private, protected , static ,final, synchronized, native,abstract 的一个子集）
  - 方法的字节码（bytecodes)、操作数栈、局部变量表及大小（abstract 和 native方法除外）
  - 异常表 （abstract 和 native 方法除外）

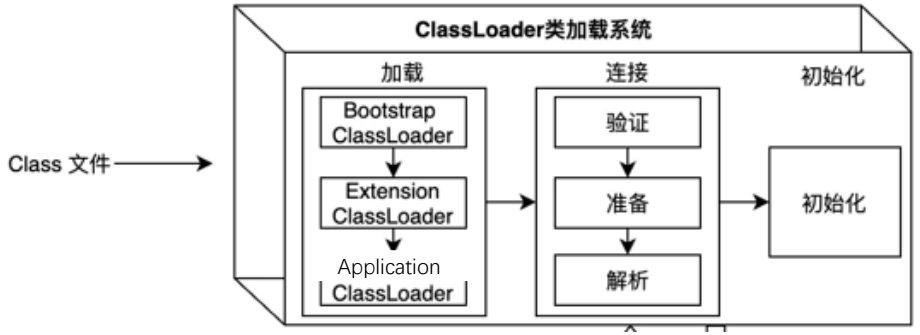
Runtime Constant Pool

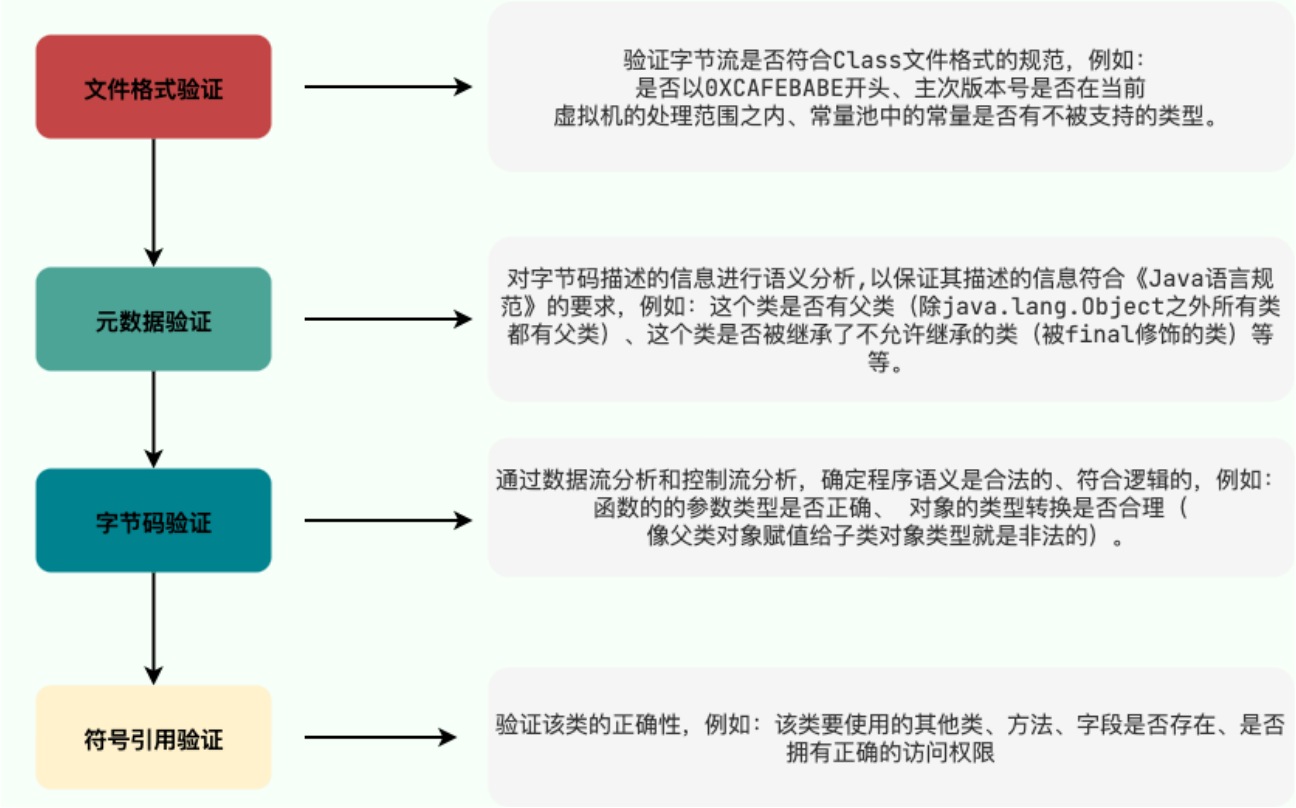


https://www.cnblogs.com/better-farther-world2099/articles/13889075.html

ClassLoader: Linking

- 验证
  - 文件格式验证（e.g., 是否以CAFEBABE开头）
  - 元数据验证（e.g., 类是否有父类）
  - 字节码验证（e.g., 任何跳转指令都不会跳转到方法体以外的字节码指令上）
  - 符号引用验证（e.g., 符号引用中通过字符串描述的fully qualified name是否能够找到对应的类）
- 准备
  - 为类的static变量分配内存并且设置初始值
- 解析
  - Resolution：解析阶段是虚拟机将常量池内的符号引用替换为直接引用的过程



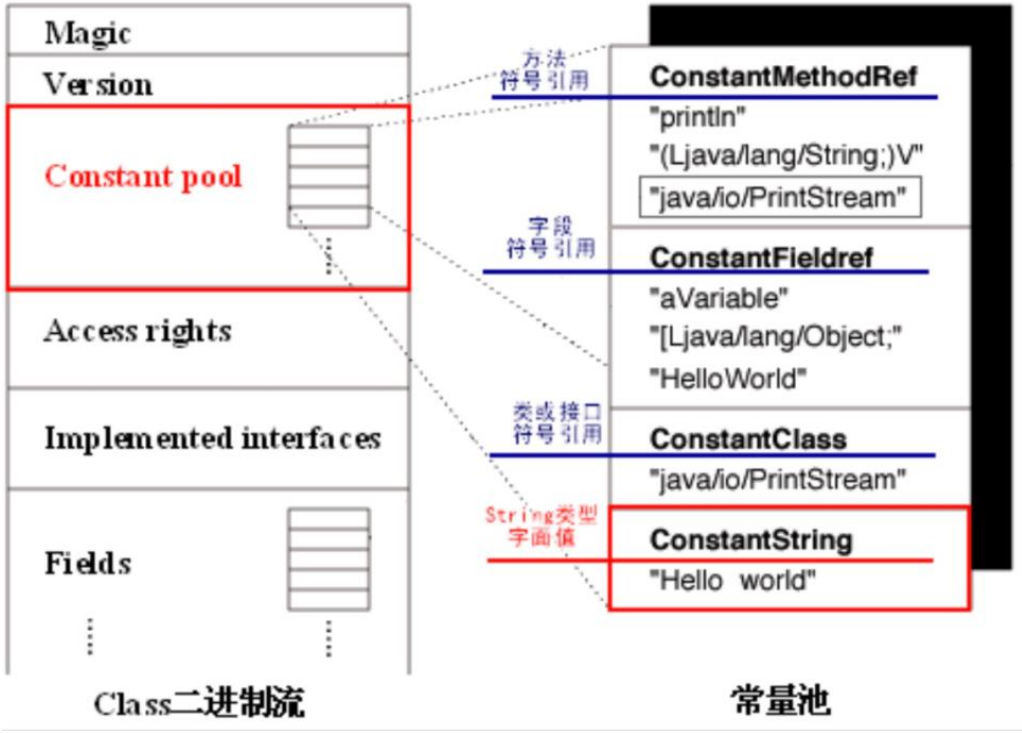


https://javaguide.cn/java/jvm/class-loading-process.html#%E9%AA%8C%E8%AF%81

《深入理解Java虚拟机》第三版，周志明

Resolution (解析)

- **符号引用**（Symbolic References）：符号引用以一组符号来描述所引用的目标，符号可以是任何形式的字面量，只要使用时能无歧义地定位到目标即可。
- **直接引用**（Direct References）：直接引用是可以直接指向目标的指针或相对偏移量。直接引用是和虚拟机实现的内存布局直接相关的，同一个符号引用在不同虚拟机实例上翻译出来的直接引用一般不会相同。
- 解析动作主要针对类或接口、字段、类方法、接口方法、方法类型等

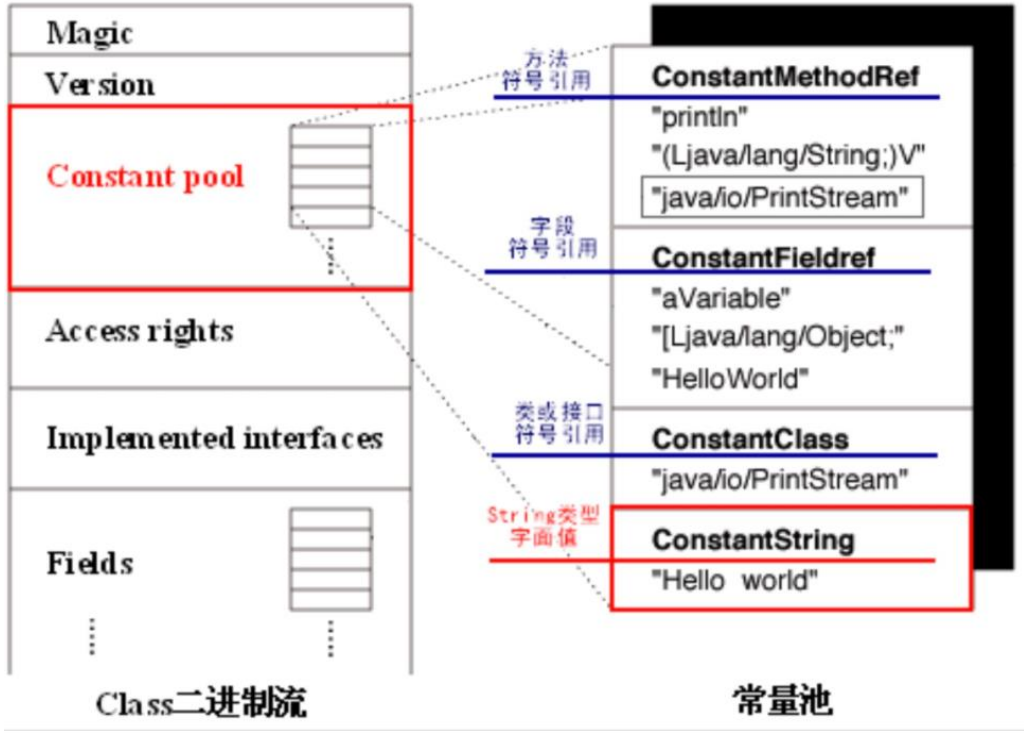


《深入理解Java虚拟机》第三版，周志明

Resolution (解析)

- 这种解析能够成立的前提是：方法在程序真正运行之前就有一个可确定的调用版本，并且这个方法的调用版本在运行期是不可改变的。即，调用目标在程序代码写好、编译器进行编译那一刻就已经确定下来。这类方法的调用被称为解析（Resolution）。

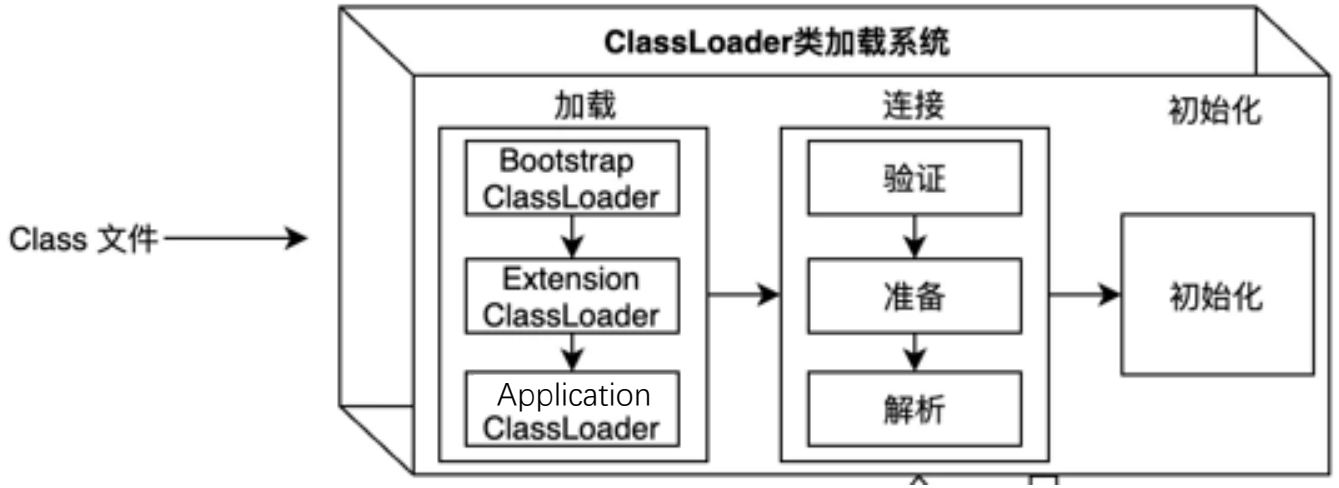
- 在Java语言中符合“编译期可知，运行期不可变”这个要求的方法，主要有静态方法和私有方法两大类，前者与类型直接关联，后者在外部不可被访问，这两种方法各自的特点决定了它们都不可能通过继承或别的方式重写出其他版本，因此它们都适合在类加载阶段进行解析。



ClassLoader: Initialization

- **初始化**：初始化类变量与静态语句块

- 准备阶段变量赋过系统要求的初始零值；在初始化阶段，JVM才真正开始执行类中编写的程序代码，根据程序员通过编程制定的主观计划去初始化类变量和其它资源



《深入理解Java虚拟机》第三版，周志明

Execution Engine

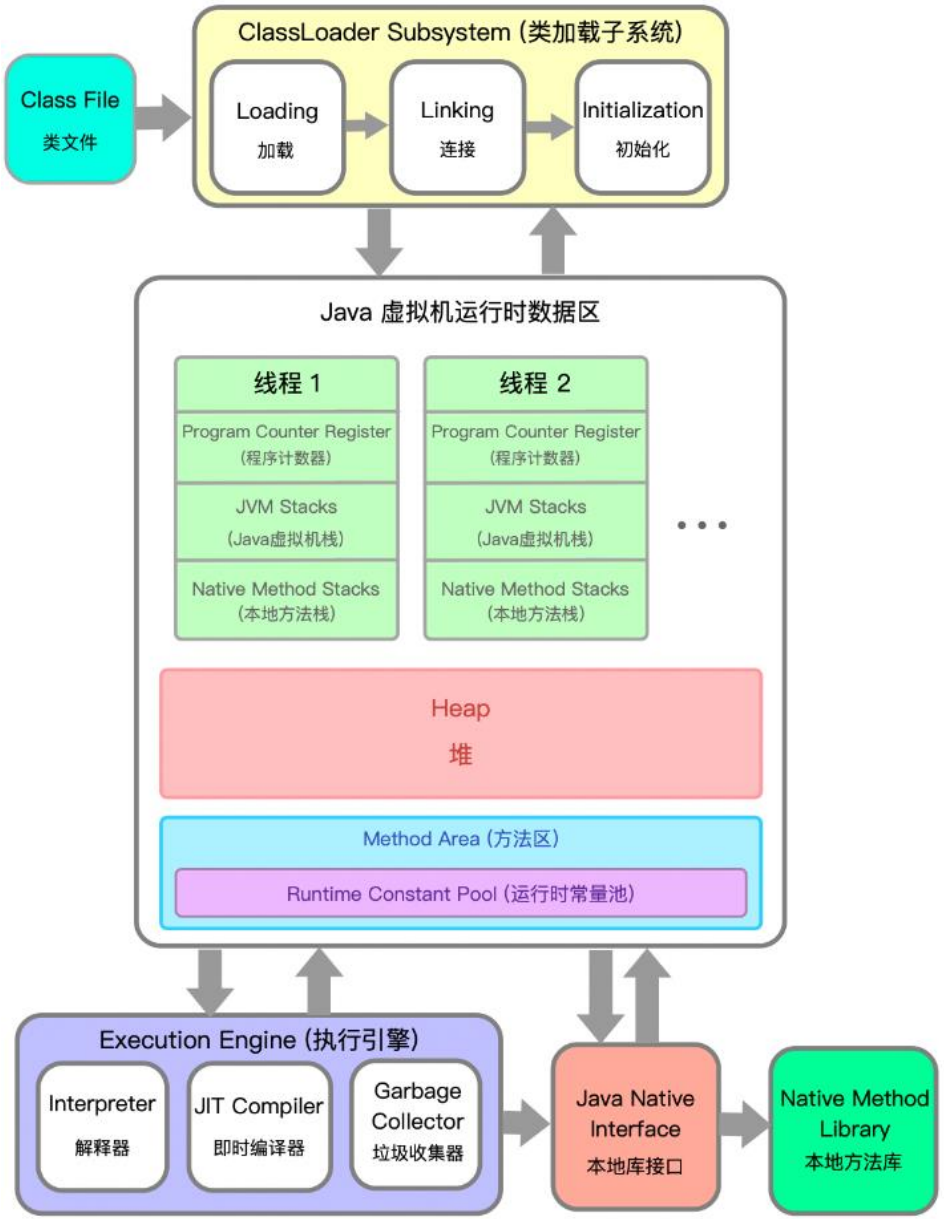
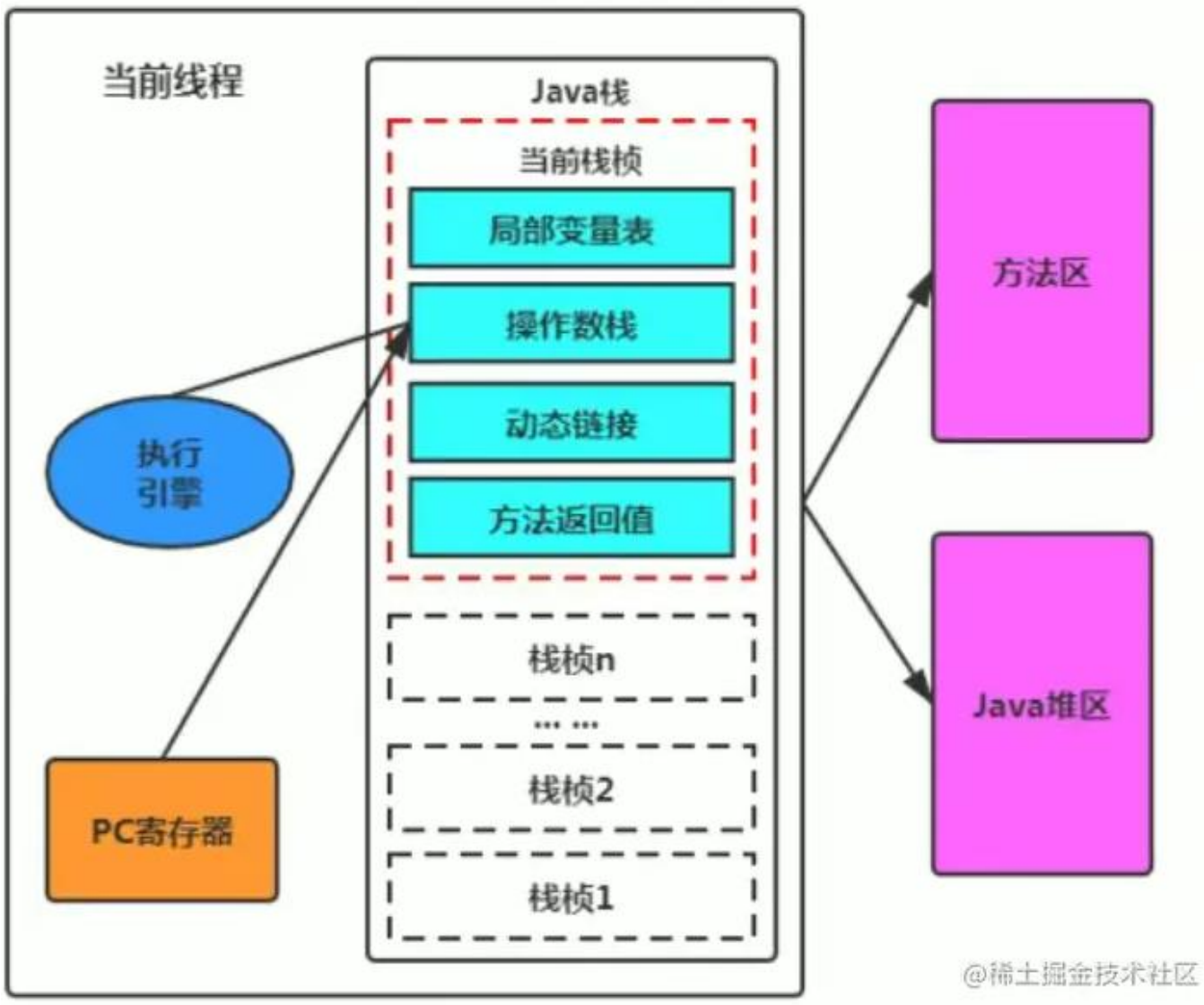


Image source: https://www.cnblogs.com/hynblogs/p/12275957.html

Execution Engine

- Java虚拟机以方法作为最基本的执行单元
- 栈帧（Stack Frame）是用于支持虚拟机进行方法调用和方法执行背后的数据结构，每一个方法从调用开始至执行结束的过程，都对应着一个栈帧在虚拟机栈里面从入栈到出栈的过程。

- 栈帧存储了方法的局部变量表、操作数栈、动态连接和方法返回地址等信息



Stack-based Opcodes & Interpreter

```
public int calc() {
    int a = 100;
    int b = 200;
    int c = 300;
    return (a + b) * c;
}
```

```
public int calc();
    Code:
        Stack=2, Locals=4, Args_size
         0:   bipush  100
         2:   istore_1
         3:   sipush  200
         6:   istore_2
         7:   sipush  300
        10:  istore_3
        11:  iload_1
        12:  iload_2
        13:  iadd
        14:  iload_3
        15:  imul
        16:  ireturn
}
```

图8-5 执行偏移地址为0的指令的情况

首先，执行偏移地址为0的指令，Bipush指令的作用是将单字节的整型常量值（-128～127）推入操作数栈顶，跟随有一个参数，指明推送的常量值，这里是100。

《深入理解Java虚拟机》第三版，周志明

Stack-based Opcodes & Interpreter

```
public int calc() {
    int a = 100;
    int b = 200;
    int c = 300;
    return (a + b) * c;
}
```

```
public int calc();
    Code:
        Stack=2, Locals=4, Args_size=1
         0:   bipush  100
         2:   istore_1
         3:   sipush  200
         6:   istore_2
         7:   sipush  300
        10:  istore_3
        11:  iload_1
        12:  iload_2
        13:  iadd
        14:  iload_3
        15:  imul
        16:  ireturn
}
```

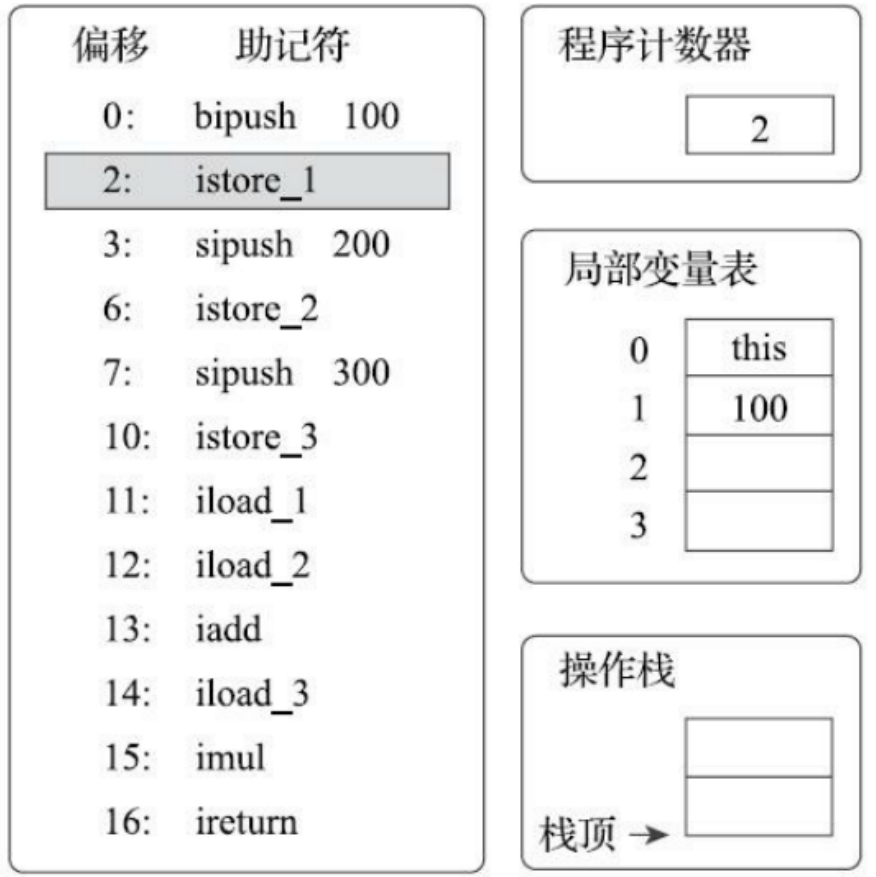


图8-6 执行偏移地址为1的指令的情况

执行偏移地址为2的指令，istore_1指令的作用是将操作数栈顶的整型值出栈并存放到第1个局部变量槽中。后续4条指令（直到偏移为11的指令为止）都是做一样的事情，也就是在对应代码中把变量a、b、c赋值为100、200、300。这4条指令的图示略过。

《深入理解Java虚拟机》第三版，周志明

Stack-based Opcodes & Interpreter

```
public int calc() {
    int a = 100;
    int b = 200;
    int c = 300;
    return (a + b) * c;
}
```

```
public int calc();
    Code:
        Stack=2, Locals=4, Args_size=1
         0:   bipush  100
         2:   istore_1
         3:   sipush  200
         6:   istore_2
         7:   sipush  300
        10:  istore_3
        11:  iload_1
        12:  iload_2
        13:  iadd
        14:  iload_3
        15:  imul
        16:  ireturn
}
```

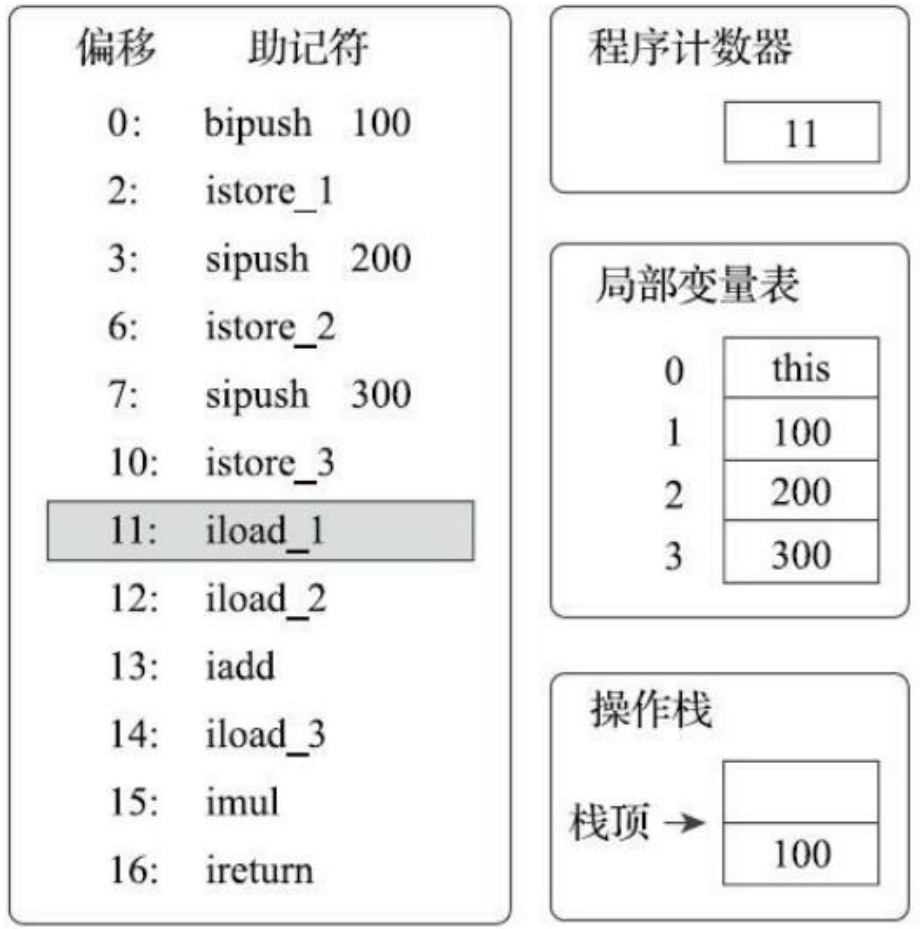


图8-7 执行偏移地址为11的指令的情况

执行偏移地址为11的指令，iload_1指令的作用是将局部变量表第1个变量槽中的整型值复制到操作栈顶。

《深入理解Java虚拟机》第三版，周志明

Stack-based Opcodes & Interpreter

```
public int calc() {
    int a = 100;
    int b = 200;
    int c = 300;
    return (a + b) * c;
}
```

```
public int calc();
    Code:
        Stack=2, Locals=4, Args_size=1
         0:   bipush  100
         2:   istore_1
         3:   sipush  200
         6:   istore_2
         7:   sipush  300
        10:  istore_3
        11:  iload_1
        12:  iload_2
        13:  iadd
        14:  iload_3
        15:  imul
        16:  ireturn
}
```

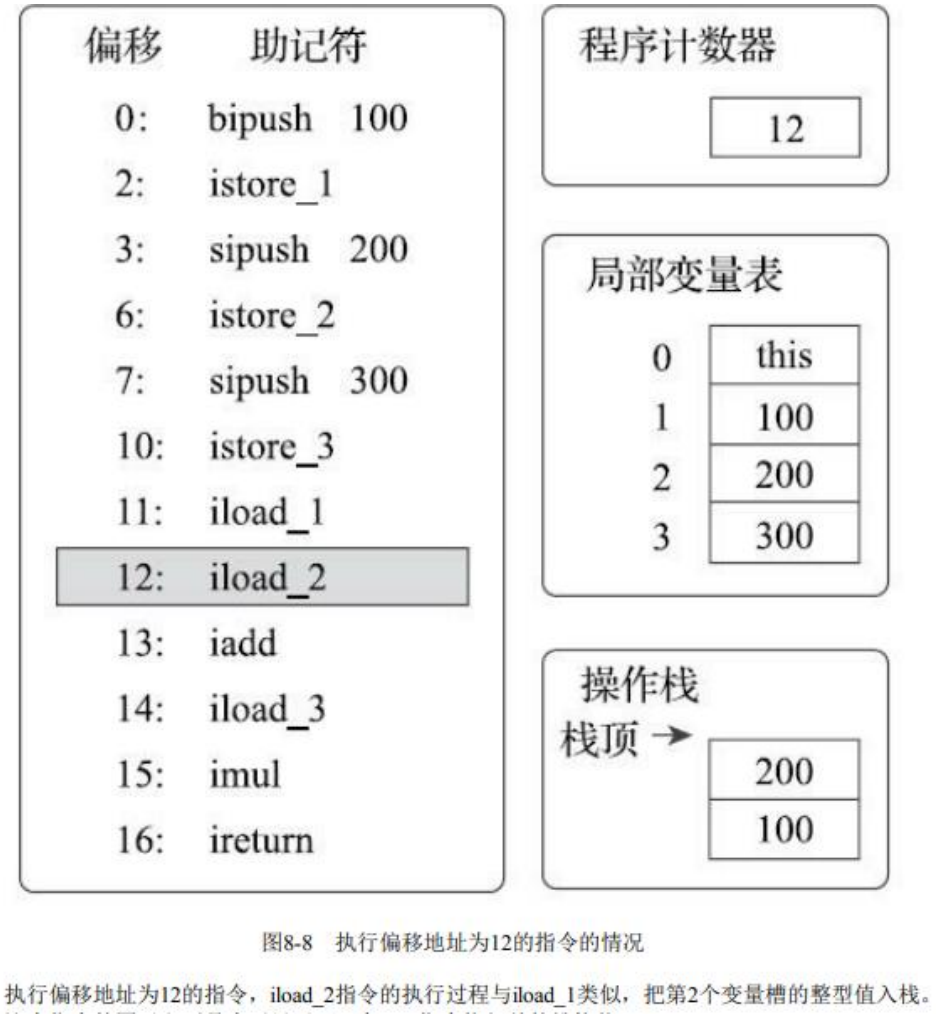


图8-8 执行偏移地址为12的指令的情况

执行偏移地址为12的指令，iload_2指令的执行过程与iload_1类似，把第2个变量槽的整型值入栈。

《深入理解Java虚拟机》第三版，周志明

Stack-based Opcodes & Interpreter

```
public int calc() {
    int a = 100;
    int b = 200;
    int c = 300;
    return (a + b) * c;
}
```

```
public int calc();
    Code:
        Stack=2, Locals=4, Args_size=1
         0:   bipush  100
         2:   istore_1
         3:   sipush  200
         6:   istore_2
         7:   sipush  300
        10:  istore_3
        11:  iload_1
        12:  iload_2
        13:  iadd
        14:  iload_3
        15:  imul
        16:  ireturn
}
```

图8-9 执行偏移地址为13的指令的情况

执行偏移地址为13的指令，iadd指令的作用是将操作数栈中头两个栈顶元素出栈，做整型加法，然后把结果重新入栈。在iadd指令执行完毕后，栈中原有的100和200被出栈，它们的和300被重新入栈。

《深入理解Java虚拟机》第三版，周志明

Stack-based Opcodes & Interpreter

```
public int calc() {
    int a = 100;
    int b = 200;
    int c = 300;
    return (a + b) * c;
}
```

```
public int calc();
    Code:
        Stack=2, Locals=4, Args_size=1
         0:   bipush  100
         2:   istore_1
         3:   sipush  200
         6:   istore_2
         7:   sipush  300
        10:  istore_3
        11:  iload_1
        12:  iload_2
        13:  iadd
        14:  iload_3
        15:  imul
        16:  ireturn
}
```

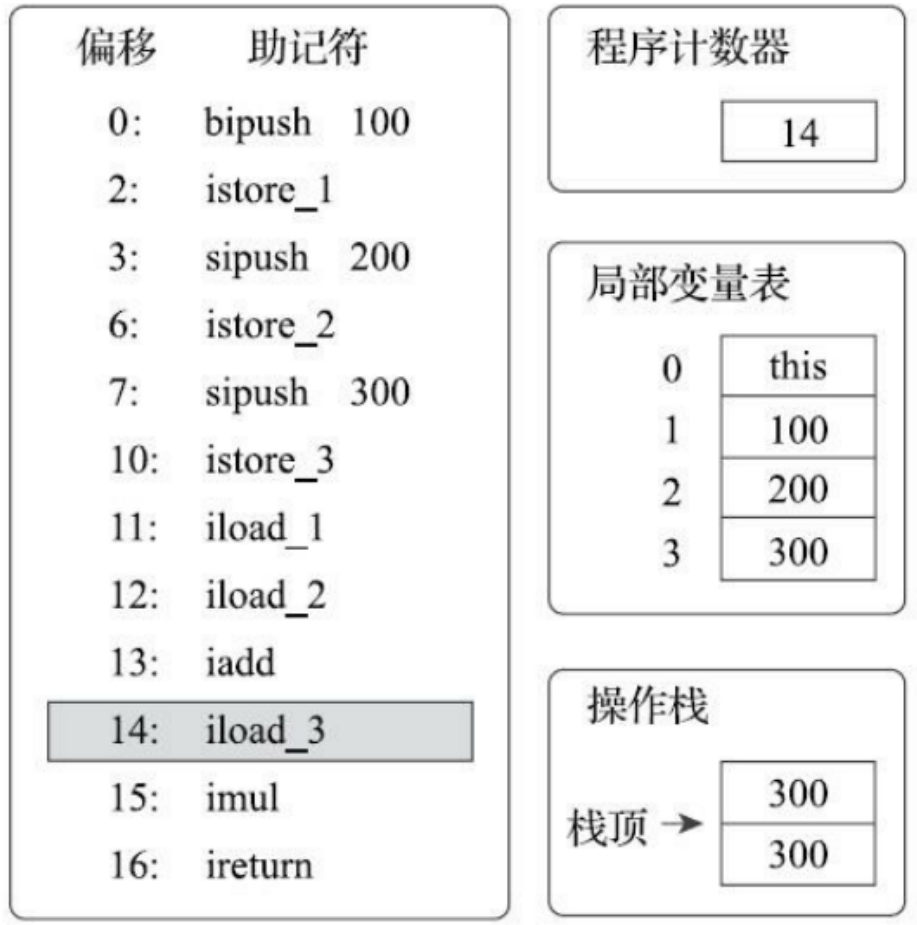


图8-10 执行偏移地址为14的指令的情况

执行偏移地址为14的指令，iload_3指令把存放在第3个局部变量槽中的300入栈到操作数栈中。这时操作数栈为两个整数300。下一条指令imul是将操作数栈中头两个栈顶元素出栈，做整型乘法，然后把结果重新入栈，与iadd完全类似，所以笔者省略图示。

《深入理解Java虚拟机》第三版，周志明

Stack-based Opcodes & Interpreter

```
public int calc() {
    int a = 100;
    int b = 200;
    int c = 300;
    return (a + b) * c;
}
```

```
public int calc();
    Code:
        Stack=2, Locals=4, Args_size=1
         0:   bipush  100
         2:   istore_1
         3:   sipush  200
         6:   istore_2
         7:   sipush  300
        10:  istore_3
        11:  iload_1
        12:  iload_2
        13:  iadd
        14:  iload_3
        15:  imul
        16:  ireturn
}
```

图8-11 执行偏移地址为16的指令的情况

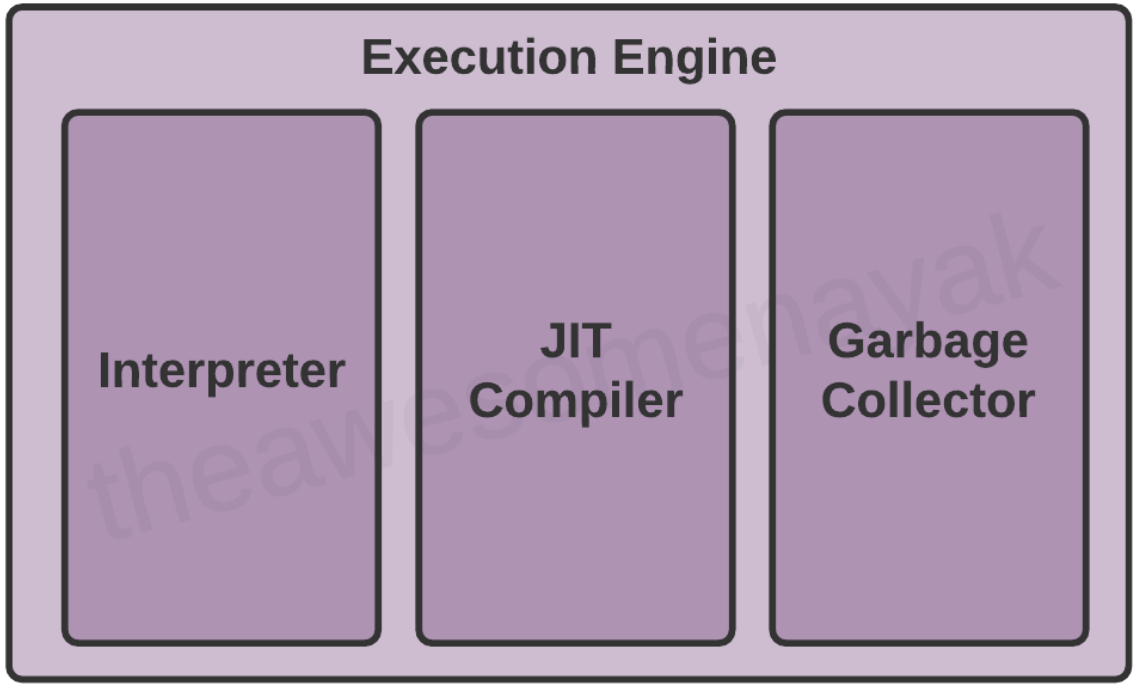
执行偏移地址为16的指令，ireturn指令是方法返回指令之一，它将结束方法执行并将操作数栈顶的整型值返回给该方法的调用者。到此为止，这段方法执行结束。

《深入理解Java虚拟机》第三版，周志明

JIT Compiler

- 目前主流的商用Java虚拟机（HotSpot）里，Java程序最初都是通过解释器（Interpreter）进行解释执行的

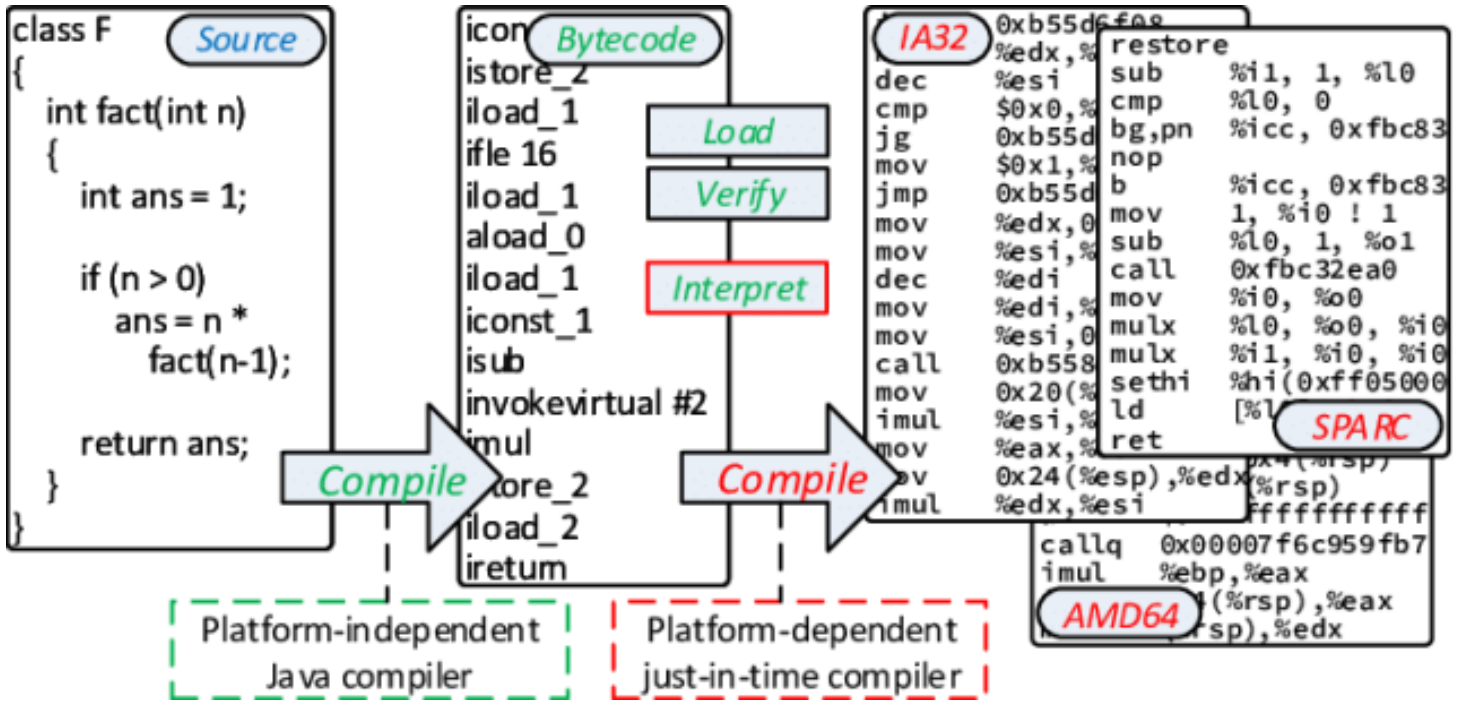
- 当虚拟机发现某个方法或代码块的运行特别频繁，就会把这些代码认定为“热点代码”（Hot Spot Code）
  - 被多次调用的方法
  - 被多次执行的循环体



《深入理解Java虚拟机》第三版，周志明

JIT Compiler

为了提高热点代码的执行效率，在运行时，虚拟机将会把这些代码编译成本地机器码，并以各种手段尽可能地进行代码优化，运行时完成这个任务的后端编译器被称为即时编译器(Just In Time, JIT)。

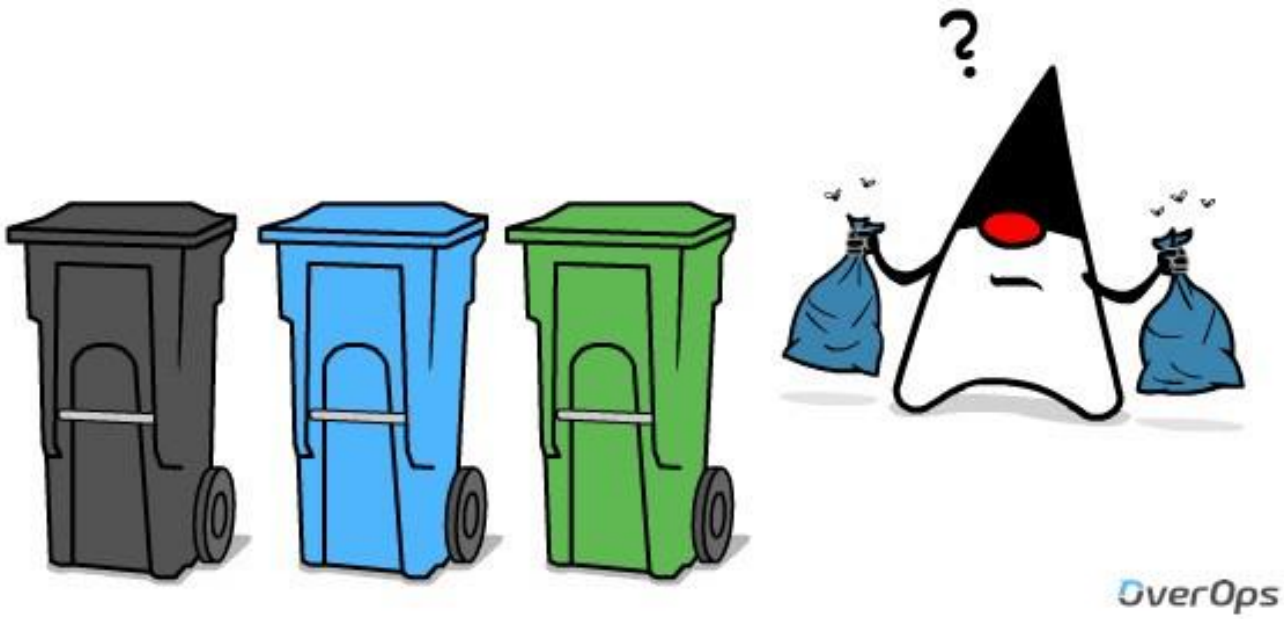


https://www.cesarsotovalero.net/blog/aot-vs-jit-compilation-in-java.html

《深入理解Java虚拟机》第三版，周志明

Garbage Collection

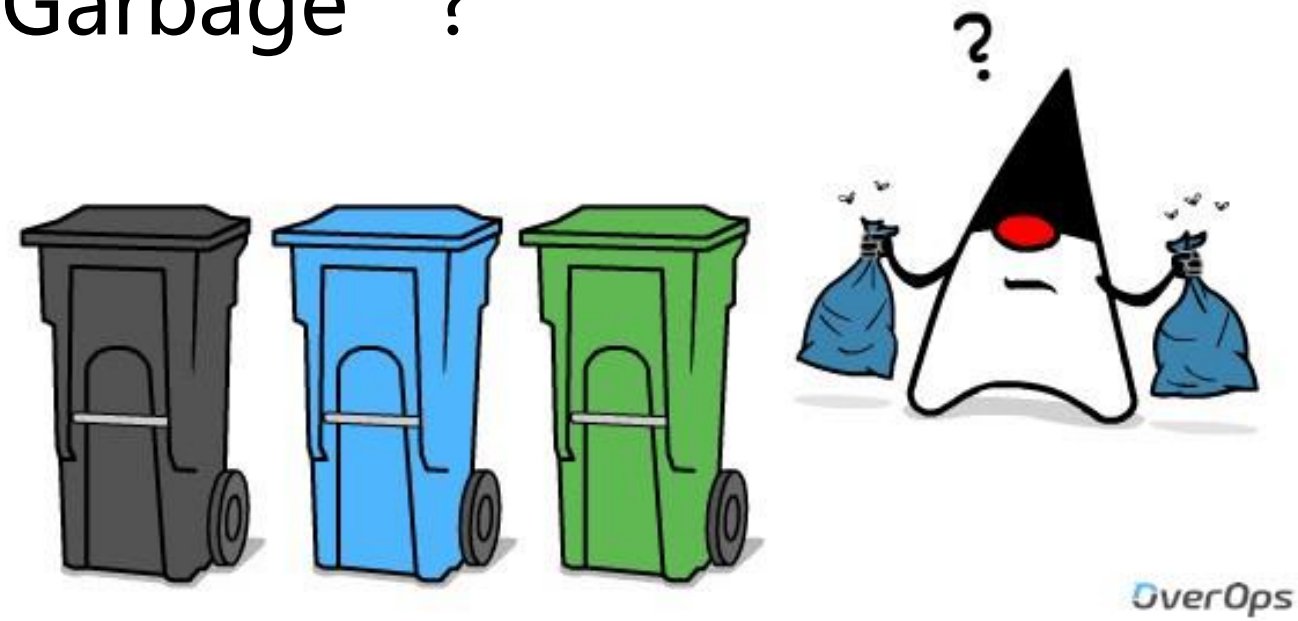
- 栈的每一个栈帧中分配多少内存基本上（可能被优化）是在类结构确定下来时就已知的，即内存分配和回收都具有确定性。
- Java堆和方法区这两个区域则有着很显著的不确定性：比如一个方法所执行的不同条件分支所需要的内存也可能不一样
- 只有处于运行期间，我们才能知道程序究竟会创建哪些对象，创建多少个对象，这部分内存的分配和回收是动态的。Java的垃圾回收关注的是这部分内存如何管理。



《深入理解Java虚拟机》第三版，周志明

Which Objects are “Garbage”？

- 引用计数算法
  (Reference Counting)
- 可达性分析算法
  (Reachability Analysis)



Reference Counting

Reference counting GC algorithms associate a reference count with each object.

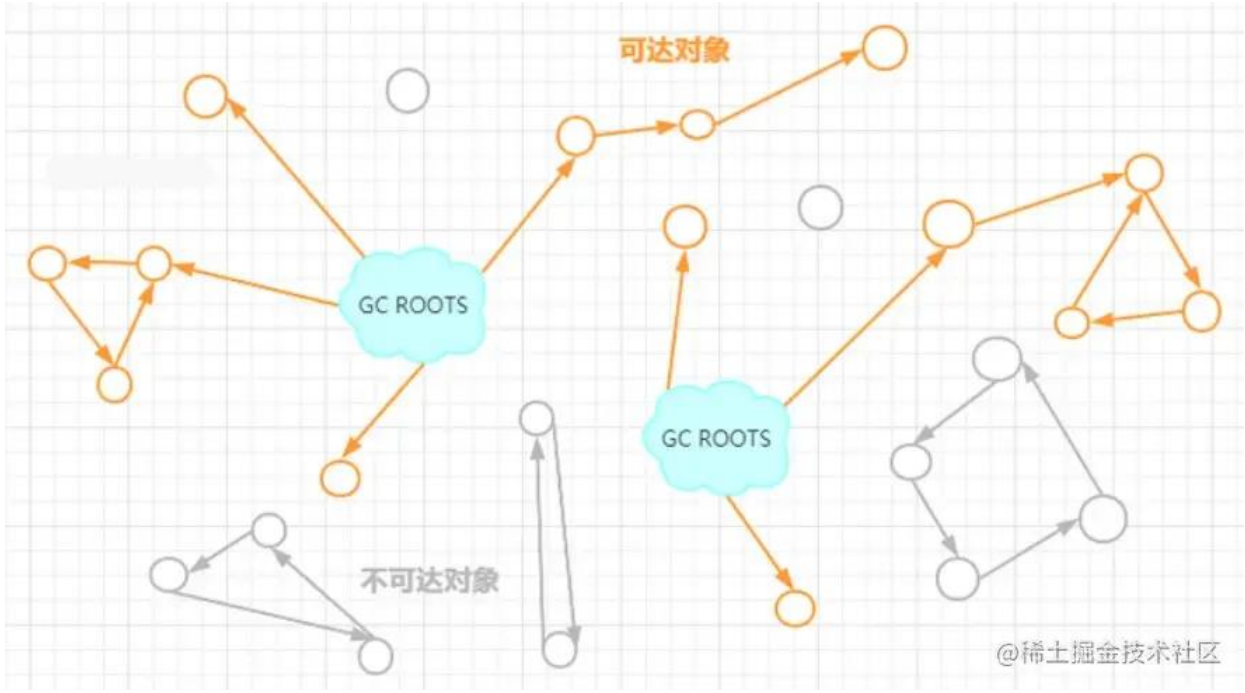
These algorithms consider an object to be alive as long as the number of references to that object is greater than zero.

What to do for Cyclic References ?

Reachability Analysis

从某一些指定的根对象（GC Roots）出发，一步步遍历找到和这个根对象具有引用关系的对象，然后再从这些对象开始继续寻找，从而形成一个个的引用链

不在这些引用链上面的对象便被标识为引用不可达对象（垃圾），需要回收掉。



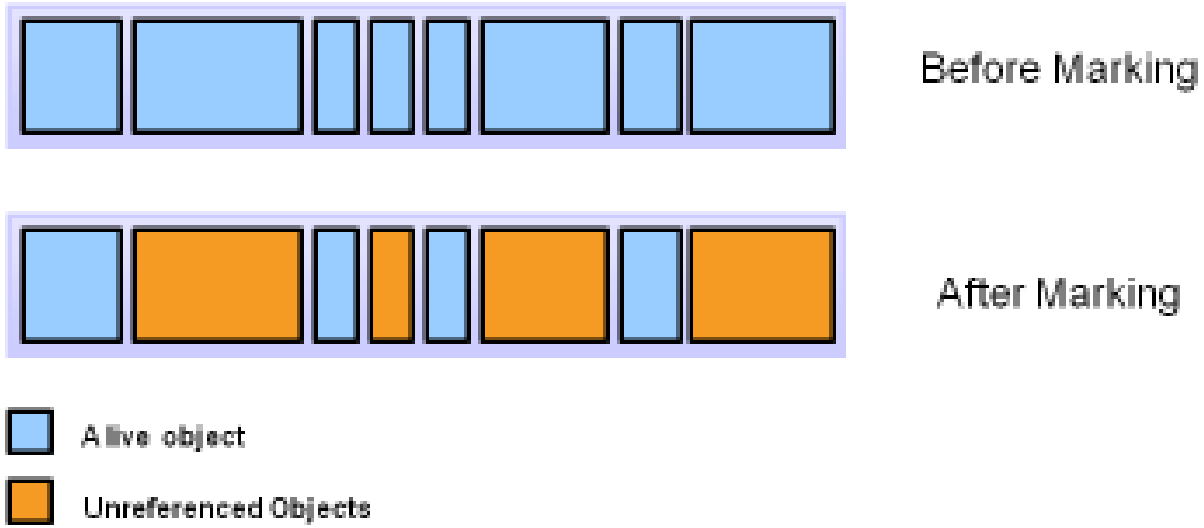
https://juejin.cn/post/7123853933801373733

Garbage Collection Algorithm

Naïve Algorithm: 标记-清除算法：首先标记出所有需要回收的对象，在标记完成后，统一回收掉所有被标记的对象。

缺点

- 大量标记和清除动作，效率低
- 内存空间碎片化



https://www.oracle.com/webfolder/technetwork/tutorials/obe/java/gc01/index.html

《深入理解Java虚拟机》第三版，周志明

Garbage Collection Algorithm

经验法则

- 弱分代假说：绝大多数对象都是朝生夕灭的
- 强分代假说：熬过越多次GC过程的对象就越难以消亡
- 跨代引用假说：跨代引用相对于同代引用来说仅占极少数

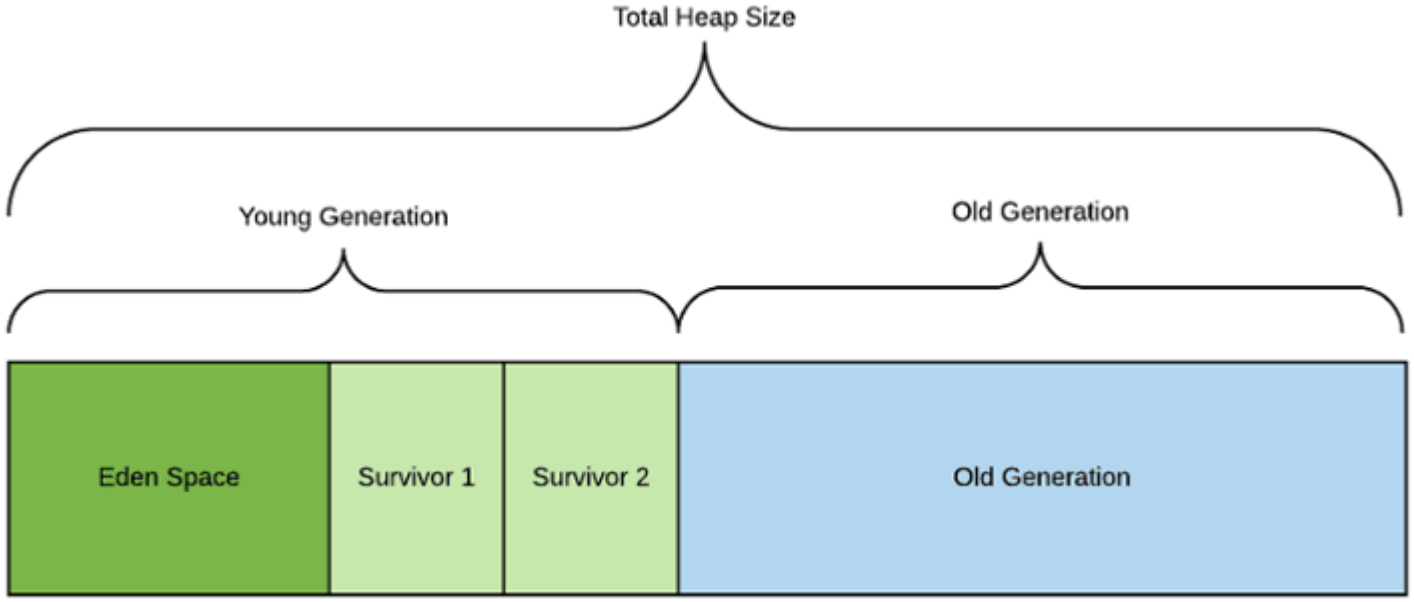


Image source: https://backstage.forgerock.com/knowledge/kb/article/a75965340

《深入理解Java虚拟机》第三版，周志明

Garbage Collection Algorithm

The Young Generation is where all new objects are allocated and aged.

When the young generation fills up, this causes a minor garbage collection.

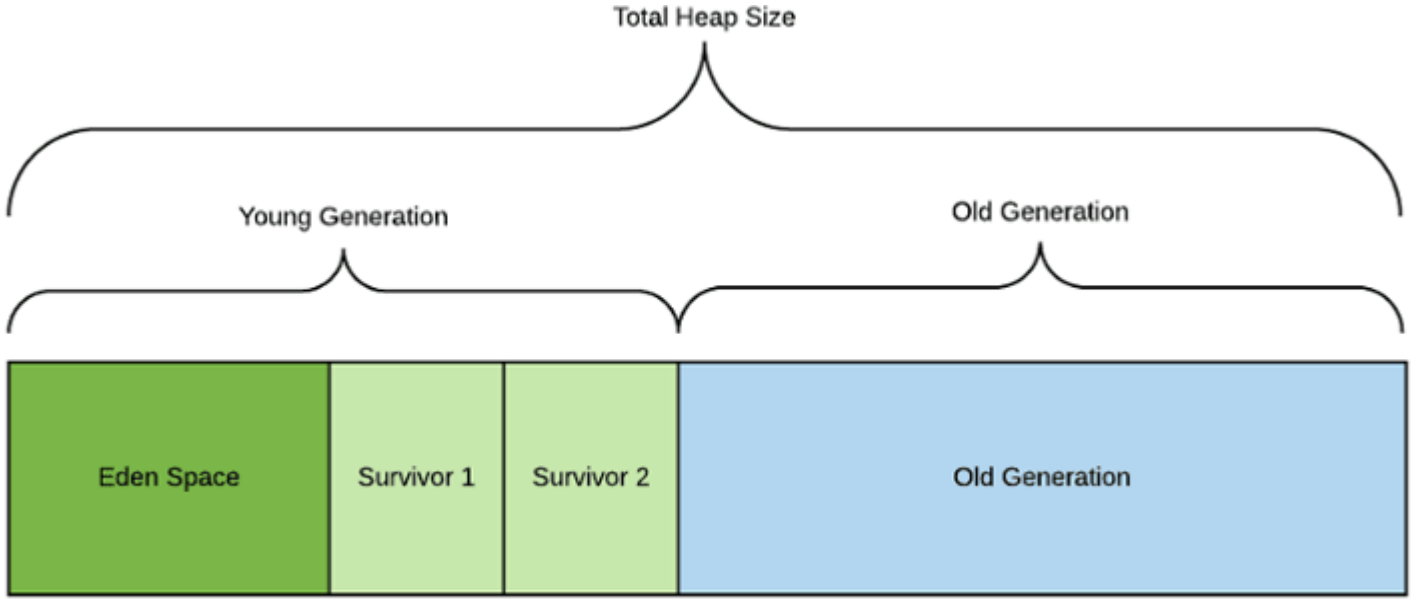


Image source: https://backstage.forgerock.com/knowledge/kb/article/a75965340

Garbage Collection Algorithm

The Old Generation is used to store long surviving objects.

Typically, a threshold is set for young generation object and when that age is met, the object gets moved to the old generation.

Eventually the old generation needs to be collected. This event is called a major garbage collection.

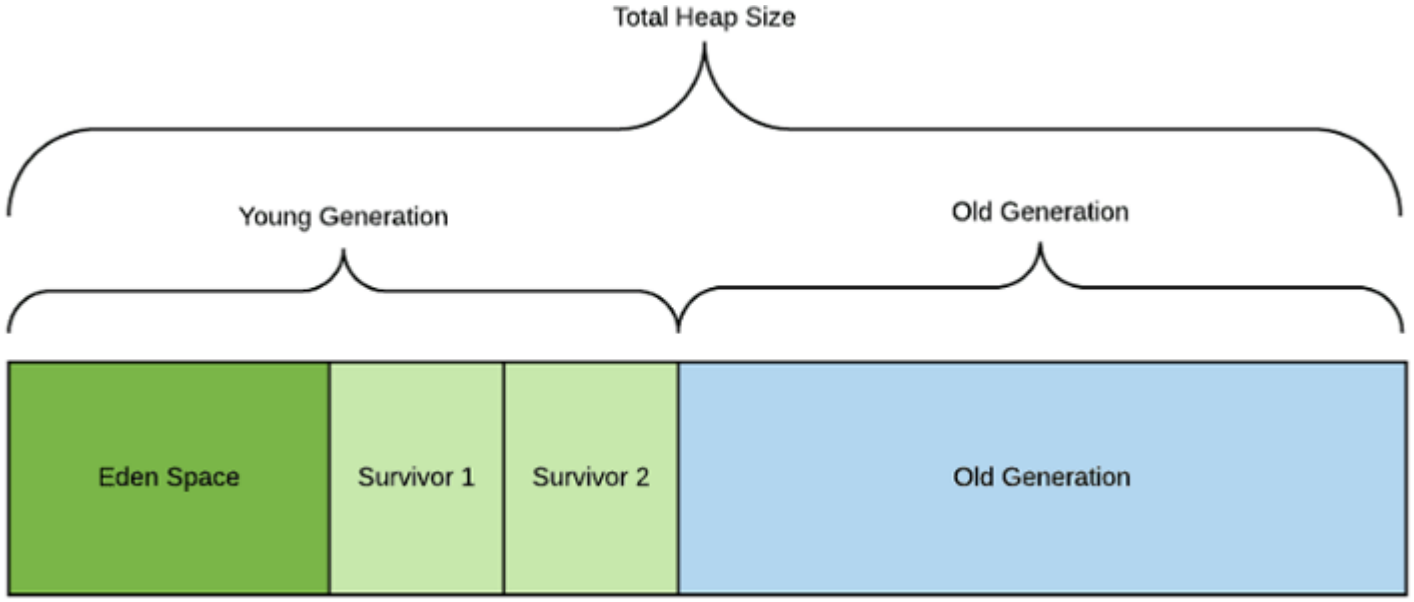


Image source: https://backstage.forgerock.com/knowledge/kb/article/a75965340

Garbage Collection Algorithm

The Old Generation is used to store long surviving objects.

Typically, a threshold is set for young generation object and when that age is met, the object gets moved to the old generation.

Eventually the old generation needs to be collected. This event is called a major garbage collection.

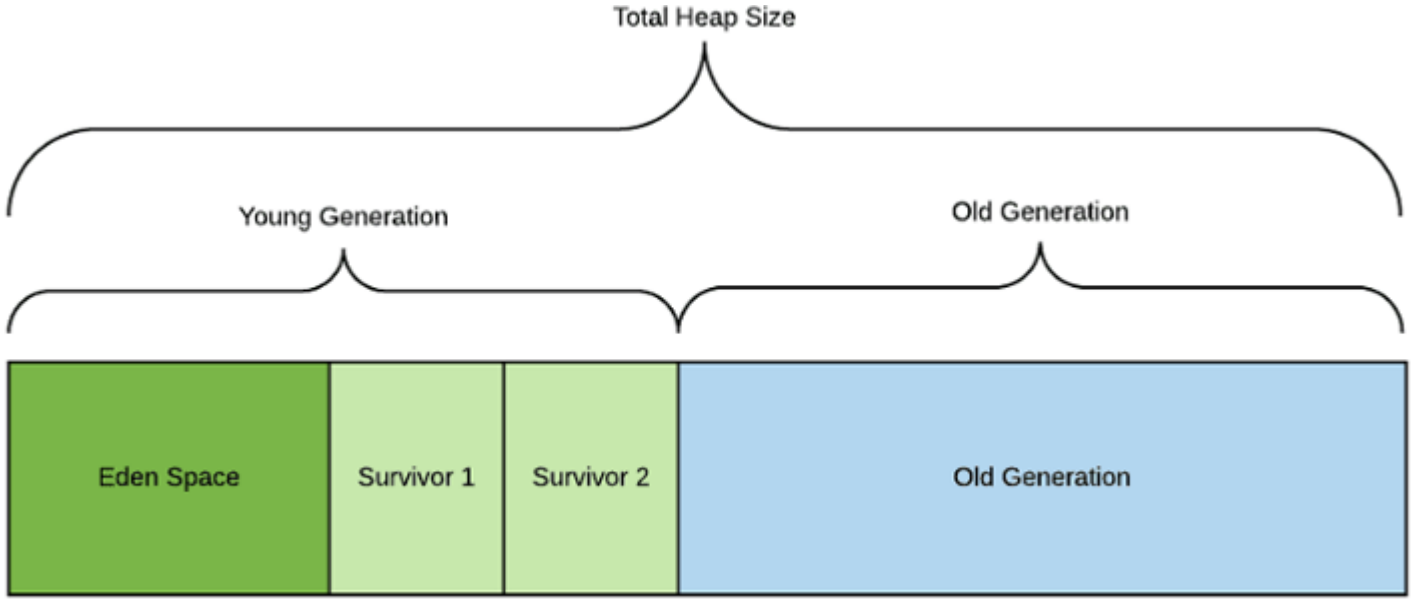


Image source: https://backstage.forgerock.com/knowledge/kb/article/a75965340

Animation: https://speakerdeck.com/deepu105/jvm-minor-gc

Next Lecture

- Project Presentation
- Course Review