JAVA虚拟机

尹洪亮

Kevin.Yin

认识JVM

JVM是什么

- JVM 是Java虚拟机的缩写， 英文为Java Virtual Machine
- JVM是一种用于计算设备的规范，它是一个虚构出来的计算机
- JVM包括一套字节码指令集、寄存器、栈、堆、方法区等

跨平台特性（分层、适配）

- JVM屏蔽了与具体操作系统平台相关的信息
- JAVA程序只需生成在字节码就可以虚拟机上运行,从而可以在任何平台上不加修改的运行
- JVM在执行字节码时，实际上最终还是把字节码解释成具体平台上的机器指令执行
- 通过JVM实现了JAVA的跨平台特性，一次编译到处运行

有哪些JVM

Sun Classic VM

1、世界上第一款商用的Java虚拟机
2、只能使用纯解释器的方式来执行Java代码

Exact VM

1、Exact Memory Management准确式内存管理
2、编译器和解释器混合工作以及两级即时编译器
3、只在Solaris平台发布了，还没来得及在Linux和Windows发布就被后面HotSpot的取代了

HotSpot VM

1、它是Sun/Oracle JDK和Open JDK中默认携带的虚拟机
2、起初是一家小公司开发的，后来被Sun收购，Sun又被Oracle收购
3、HotSpot指的就是它的热点代码探测技术、Exact VM其实也有相近的技术
4、目前使用范围最广的Java虚拟机
5、java –version 即可查看使用的虚拟机

KVM

1、简单，轻量，高度可移植
2、嵌入式虚拟机产品，在手机平台运行

有哪些JVM

JRockit

1、JRockit是BEA公司研发的。
2、世界上最快的java虚拟机。
3、专注服务器端应用。
4、优势：垃圾收集器；MissionControl服务套件

J9

1、IBM公司研发了。它最开始的名字不叫J9，叫IBM Technology for Java virtual Machine ----IT4j
2、类似于HotSpot，他不仅可以用于服务器端，还可以用于桌面应用，嵌入式；它开发是为了IBM产品的各种java平台

Dalvik

1、它不是java虚拟机，因为它没有遵循java虚拟机的规范，它是不能直接执行编译后的class文件的
2、它使用的是寄存器架构，而不是常用的栈架构。
3、它所执行的是Dex-dalvik Executalbe文件,这个文件可以通过class文件转化而来。
4、用于移动端

有哪些JVM

Microsoft JVM

1、一看就知道是微软开发的，也是为了自家软件与java兼容

TaobaoVM

1、淘宝根据Hotspot进行深度定制的的虚拟机

2、对硬件的依赖性够，牺牲了兼容性。

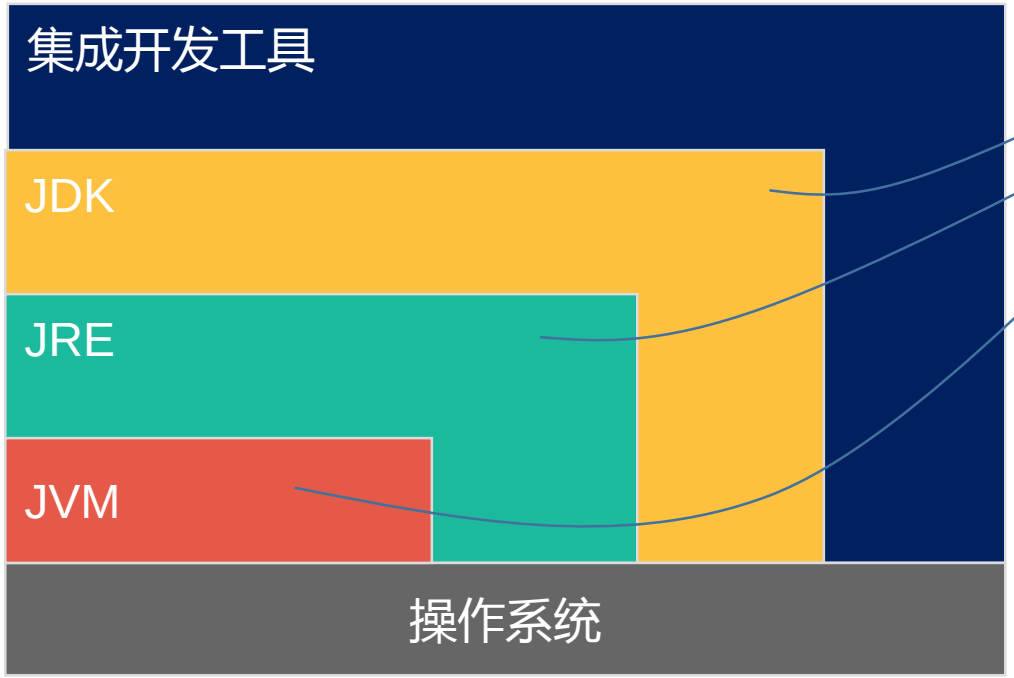
Azul VM

Azul VM是Azul Systems 公司在HotSpot基础上进行大量改进，运行于Azul Systems公司的专有硬件Vega系统上的Java虚拟机

Liquid VM

Liquid VM即是现在的JRockit VE（Virtual Edition），它是BEA公司开发的，可以直接运行在自家Hypervisor系统上的JRockit VM的虚拟化版本，Liquid VM不需要操作系统的支持，或者说它自己本身实现了一个专用操作系统的必要功能，如文件系统、网络支持等。由虚拟机越过通用操作系统直接控制硬件可以获得很多好处，如在线程调度时，不需要再进行内核态/用户态的切换等，这样可以最大限度地发挥硬件的能力，提升Java程序的执行性能。

JDK、JRE、JVM的关系



- JDK包含JRE、JRE包含JVM
- JRE由JVM和核心类库组成,程序运行只要有JRE就够了
- JVM与操作系统打交道，没有JVM就没有跨平台

- 不同版本的JDK目录结构不同、

JDK9新特性-非JVM相关15项

1、目录结构
2、模块化系统
3、jshell
4、多版本兼容JAR
5、接口的私有方法
6、改进try-with-resourcs
7、改进砖石操作符
8、限制使用单独下划线标识符
9、String存储结构变更
10、快速创建只读结合
11、增强Stream API
12、改进Optional 类
13、多分辨率图像 API
14、全新 HTTP客服端API
15、javadoc 的 HTML5 支持

JDK9新特性-JVM相关3项

1、智能JAVA 编译工具

智能 java 编译工具(sjavac)，用于在多核处理器情况下提升 JDK 的编译速度。其目的是改进 Java 编译工具，并取代目前 JDK 编译工具 javac，JDK 9 还更新了 javac 编译器以便能够将 java 9 代码编译运行在低版本 Java 中

2、统一JVM 日志系统

对所有的 JVM 组件引入一个单一的系统，这些 JVM 组件支持细粒度的和易配置的 JVM 日志

3、java 动态编译

JIT（Just-in-time）编译器可以在运行时将热点编译成本地代码，速度很快。但是 Java 项目现在变得很大很复杂，因此 JIT 编译器需要花费较长时间才能热身完，而且有些 Java 方法还没法编译，性能方面也会下降。AoT 编译就是为了解决这些问题而生的。在 JDK 9 中， AOT（JEP 295: Ahead-of-Time Compilation）作为实验特性被引入进来，开发者可以利用新的 jaotc 工具将重点代码转换成类似类库一样的文件。虽然仍处于试验阶段，但这个功能使得 Java应用在被虚拟机启动之前能够先将 Java 类编译为原生代码。此功能旨在改进小型和大型应用程序的启动时间，同时对峰值性能的影响很小。

JDK10新特性-非JVM相关7项

1、局部变量类型推断
2、将JDK多存储库合并为单储存库
3、应用数据共享
4、线程局部管控
5、Unicode 标签扩展
6、Root 证书
7、基于时间的版本控制
8、移除Native-Header Generation Tool（javah）

JDK10新特性-JVM相关4项

1、垃圾回收接口

是一个在 JVM 源代码中的允许另外的垃圾回收器快速方便的集成的接口。垃圾回收接口为HotSpot的GC代码提供更好的模块化；在不影响当前代码的基础情况下，将GC添加到HotSpot变的更简单；更容易从JDK构建中排除GC

2、并行Full GC 的G1

G1垃圾收集器在JDK 9中是默认的。以前的默认值并行收集器中有一个并行的Full GC。为了尽量减少对使用GC用户的影响，G1的Full GC也应该并行。G1垃圾收集器的设计目的是避免Full收集，但是当集合不能足够快地回收内存时，就会出现完全GC。目前对G1的Full GC的实现使用了单线程标记-清除-压缩算法。JDK10 使用并行化标记-清除-压缩算法，并使用Young和Mixed收集器相同的线程数量。线程的数量可以由-XX:ParallelGCThreads选项来控制，但是这也会影响用Young和Mixed收集器的线程数量。

3、备用内存设备上分配堆内存

启用HotSpot VM以在用户指定的备用内存设备上分配Java对象堆。随着廉价的NV-DIMM内存的可用性，未来的系统可能配备了异构的内存架构。这种技术的一个例子是英特尔的3D XPoint。这样的体系结构，除了DRAM之外，还会有一种或多种类型的非DRAM内存，具有不同的特征。具有与DRAM具有相同语义的可选内存设备，包括原子操作的语义，因此可以在不改变现有应用程序代码的情况下使用DRAM代替DRAM。所有其他的内存结构，如代码堆、metaspace、线程堆栈等等，都将继续驻留在DRAM中。

JDK10新特性-JVM相关4项

4、基于实验JAVA 的JIT 编译器

启用基于Java的JIT编译器Graal，将其用作Linux / x64平台上的实验性JIT编译器。Graal是一个基于Java的JIT编译器,它是JDK 9中引入的Ahead-of-Time（AOT）编译器的基础。使它成为实验性JIT编译器是Project Metropolis的一项举措，它是下一步是研究JDK的基于Java的JIT的可行性。

使Graal可用作实验JIT编译器，从Linux / x64平台开始。Graal将使用JDK 9中引入的JVM编译器接口（JVMCI）。Graal已经在JDK中，因此将它作为实验JIT将主要用于测试和调试工作。要启用Graal作为JIT编译器，请在java命令行上使用以下选项：-XX:+UnlockExperimentalVMOptions -XX:+UseJVMCICompiler

JDK9~12版本差异

JDK Release-notes

https://www.oracle.com/technetwork/java/javase/jdk-relnotes-index-2162236.html

JDK9新功能

https://docs.oracle.com/javase/9/whatsnew/toc.htm#JSNEW-GUID-C23AFD78-C777-460B-8ACE-58BE5EA681F6

JDK10新功能

https://www.oracle.com/technetwork/java/javase/10-relnote-issues-4108729.html

JDK11新功能

https://www.oracle.com/technetwork/java/javase/11-relnote-issues-5012449.html

JDK12新功能

https://www.oracle.com/technetwork/java/javase/12-relnote-issues-5211422.html

JAVA_HOME\bin 标准工具

- JDK提供的各种标准工具，在不同的版本中都会提供，可能存在差异

基础类工具

工具	说明
appletviewer	在没有web浏览器的情况下运行和调试applet
extcheck	检查Jar冲突的工具
jar	创建和管理Jar文件
java	Java运行工具，用于运行.class字节码文件或.jar文件
javac	用于Java编程语言的编译器
javadoc	API文档生成器
javah	C头文件和stub函数生成器，用于编写native方法
javap	类文件反汇编器，解析字节码文件
jdb	Java调试器(Java Debugger)
jdeps	Java类依赖性分析器

JAVA_HOME\bin 标准工具

安全类工具

工具	说明
keytool	管理密钥库和证书。主要用于获取或缓存Kerberos协议的票据授权票据。允许用户查看本地凭据缓存和密钥表中的条目。Kerberos密钥表管理工具，允许用户管理存储于本地密钥表中的主要名称和服务密钥。
jarsigner	生成并验证JAR签名
policytool	管理策略文件的GUI工具，用于管理用户策略文件(.java.policy)

国际化工具(i18n)

工具	说明
native2ascii	本地编码到ASCII编码的转换器(Native-to-ASCII Converter)，用于“任意受支持的字符编码”和与之对应的“ASCII编码和(或)Unicode转义”之间的相互转换。

JAVA_HOME\bin 标准工具

远程方法调用类工具(RMI)

工具	说明
rmic	Java RMI 编译器，为使用JRMP或IIOP协议的远程对象生成stub、skeleton、和tie类，也用于生成OMG IDL。
rmiregistry	远程对象注册表服务，用于在当前主机的指定端口上创建并启动一个远程对象注册表。
rmid	启动激活系统守护进程，允许在虚拟机中注册或激活对象。
serialver	生成并返回指定类的序列化版本ID

JAVA_HOME\bin 标准工具

Java IDL 与 RMI-IIOP类工具

工具	说明
rmic	提供对命名服务的访问
idlj	IDL转Java编译器(IDL-to-Java Compiler)，生成映射OMG IDL接口的.java文件，并启用以Java编程语言编写的使用CORBA功能的应用程序的.java文件。IDL意即接口定义语言(Interface Definition Language)。
orbd	对象请求代理守护进程(Object Request Broker Daemon)，提供从客户端查找和调用CORBA环境服务端上的持久化对象的功能。使用ORBD代替瞬态命名服务tnameserv。ORBD包括瞬态命名服务和持久命名服务。ORBD工具集成了服务器管理器，互操作命名服务和引导名称服务器的功能。当客户端想进行服务器时定位，注册和激活功能时，可以与servertool一起使用。
servertool	为应用程序注册，注销，启动和关闭服务器提供易用的接口

JAVA_HOME\bin 标准工具

发布类工具

工具	说明
javapackager	打包、签名Java和JavaFX应用程序
pack200	使用Java gzip压缩器将JAR文件转换为压缩的pack200文件。压缩的压缩文件是高度压缩的JAR，可以直接部署，节省带宽并减少下载时间。
unpack200	将pack200生成的打包文件解压提取为JAR文件

WEB启动类工具

工具	说明
javaws	启动Java Web Start并设置各种选项的工具

JAVA_HOME\bin 标准工具

诊断监控类工具

工具	说明
jcmd	JVM诊断命令工具，将诊断命令请求发送到正在运行的Java虚拟机。
jconsole	用于监控Java虚拟机的使用JMX规范的图形工具。它可以监控本地和远程JVM。它还可以监控和管理应用程序。
jmc	Java任务控制客户端（JMC，Java Mission Control），包含用于监控和管理Java应用程序的工具，而不会引入与这些工具相关联的性能开销。开发者可以使用jmc命令来创建JMC工具。
jvisualvm	一种图形化工具，可在Java虚拟机中运行时提供有关基于Java技术的应用程序（Java应用程序）的详细信息。 Java VisualVM提供内存和CPU分析，堆转储分析，内存泄漏检测，MBean访问和垃圾收集。

JAVA_HOME\bin 实验性工具

- 由HotSpot JDK提供了，但是可能在之后的某个版本就不不提供了，但是这些工具极其重要，要学习使用。

监控类工具

工具	说明
jps	JVM进程状态工具(JVM Process Status Tool)，在目标系统上列出HotSpot Java虚拟机进程的描述信息
jstat	JVM统计监控工具(JVM Statistics Monitoring Tool)，根据参数指定的方式收集和记录指定的jvm进程的性能统计信息。
jstatd	JVM jstat守护程序，启动一个RMI服务器应用程序，用于监视测试的HotSpot Java虚拟机的创建和终止，并提供一个界面，允许远程监控工具附加到在本地系统上运行的Java虚拟机。

JAVA_HOME\bin 实验性工具

故障排查类工具

工具	说明
jinfo	Java的配置信息工具(Java Configuration Information)，用于打印指定Java进程、核心文件或远程调试服务器的配置信息。
jhat	Java堆分析工具(Java Heap Analysis Tool)，用于分析Java堆内存中的对象信息。
jmap	Java内存映射工具(Java Memory Map)，主要用于打印指定Java进程、核心文件或远程调试服务器的共享对象内存映射或堆内存细节。
jsadebugd	适用于Java的可维护性代理调试守护程序(Java Serviceability Agent Debug Daemon)，主要用于附加到指定的Java进程、核心文件，或充当一个调试服务器。
jstack	Java的堆栈跟踪工具，主要用于打印指定Java进程、核心文件或远程调试服务器的Java线程的堆栈跟踪信息

JAVA_HOME\bin 实验性工具

脚本类工具

工具	说明
jjs	对Nashorn引擎的调用。Nashorn是基于Java实现一个轻量级高性能的JavaScript运行环境。
jrunscript	Java命令行脚本外壳工具(command line script shell)，主要用于解释执行javascript、groovy、ruby等脚本语言。

JAVA_HOME\db

- Java DB 是 Oracle 支持的 Apache Derby 开源数据库的发行版本。它通过 JDBC 和 Java EE API 支持标准的 ANSI/ISO SQL，Java DB 包括在 JDK 中。

简介：

- 功能齐备、易于使用
- 事务保护的和崩溃可恢复
- 可嵌入到应用中
- 纯 Java、可移植、跨 CDC FP 1.1、Java 5、Java 6 和 Java 7（可运行于从平板电脑到大型机的任何机器上）
- 包括在 JDK 中
- 紧凑型 (2.6 MB)

官网地址

- https://www.oracle.com/technetwork/cn/java/javadb/overview/index.html

JAVA_HOME\include & JAVA_HOME\lib

JAVA_HOME\include

- C 语言头文件 支持 用Java本地接口和Java虚拟机接口 来本机代码编程

JAVA_HOME\lib

- Java开发工具要用的类库文件

Jar包	说明
dt.jar	关于运行环境的类库,主要是swing的类库
ant-javafx.jar、javafx-mx.jar	JavaFX的类库
jconsole.jar	jconsole工具的类库
tools.jar	包含了java、javac、jar等等工具的类库

JAVA_HOME\jre\bin & JAVA_HOME\jre\lib

JAVA_HOME\jre\bin

- 开发工具可执行文件(部分与JAVA_HOME\bin重复)、JVM实现、

JAVA_HOME\jre\lib

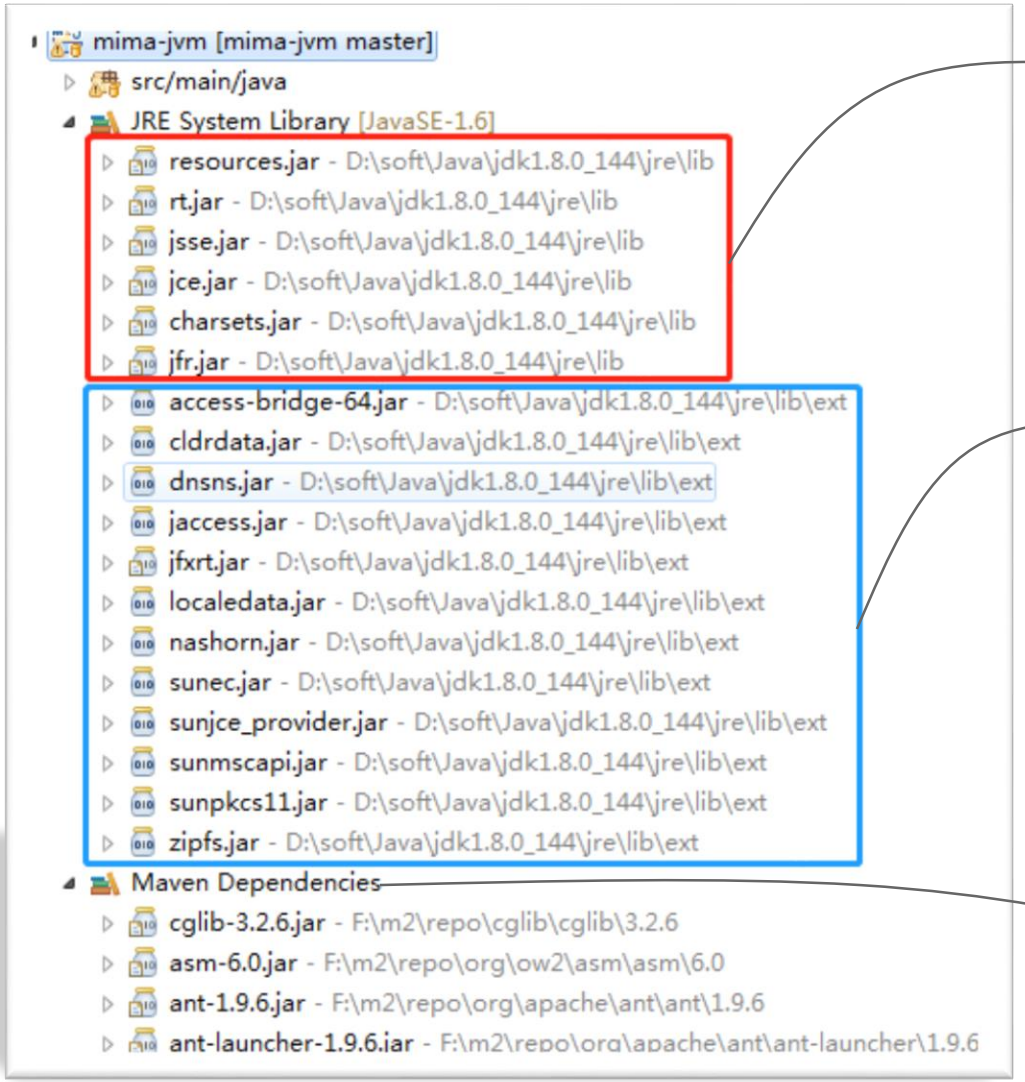
- JRE要用的核心类库，属性设置，资源文件等

目录	说明
\ext\	Java平台扩展类库
\applet\	Java applets 要的Jar包，可以放到lib/applet/目录，节省 applet 类装载器从本地文件系统装载
\fonts\	平台所需的TrueType字体文件
\images\	图片资源
\security\	安全策略配置
\management\	管理配置，JMX配置
\deploy\	和部署相关的i18n文件、logo图片

JAVA_HOME\jre\lib

目录	说明
rt.jar	Java核心类库，rt.jar 默认就在Bootstrap Classloader的加载路径里面的，同时jre/lib目录下的jce.jar、jsse.jar、charsets.jar、resources.jar都在Bootstrap Classloader中
resources.jar	资源文件,包含图片、properties配置文件等
jsse.jar、jce.jar	加解密和安全相关jar包
charsets.jar	字符集定义
jfr.jar	JAVA飞行记录仪，JVM性能分析工具，Java Flight Recorder

项目加载的类库说明

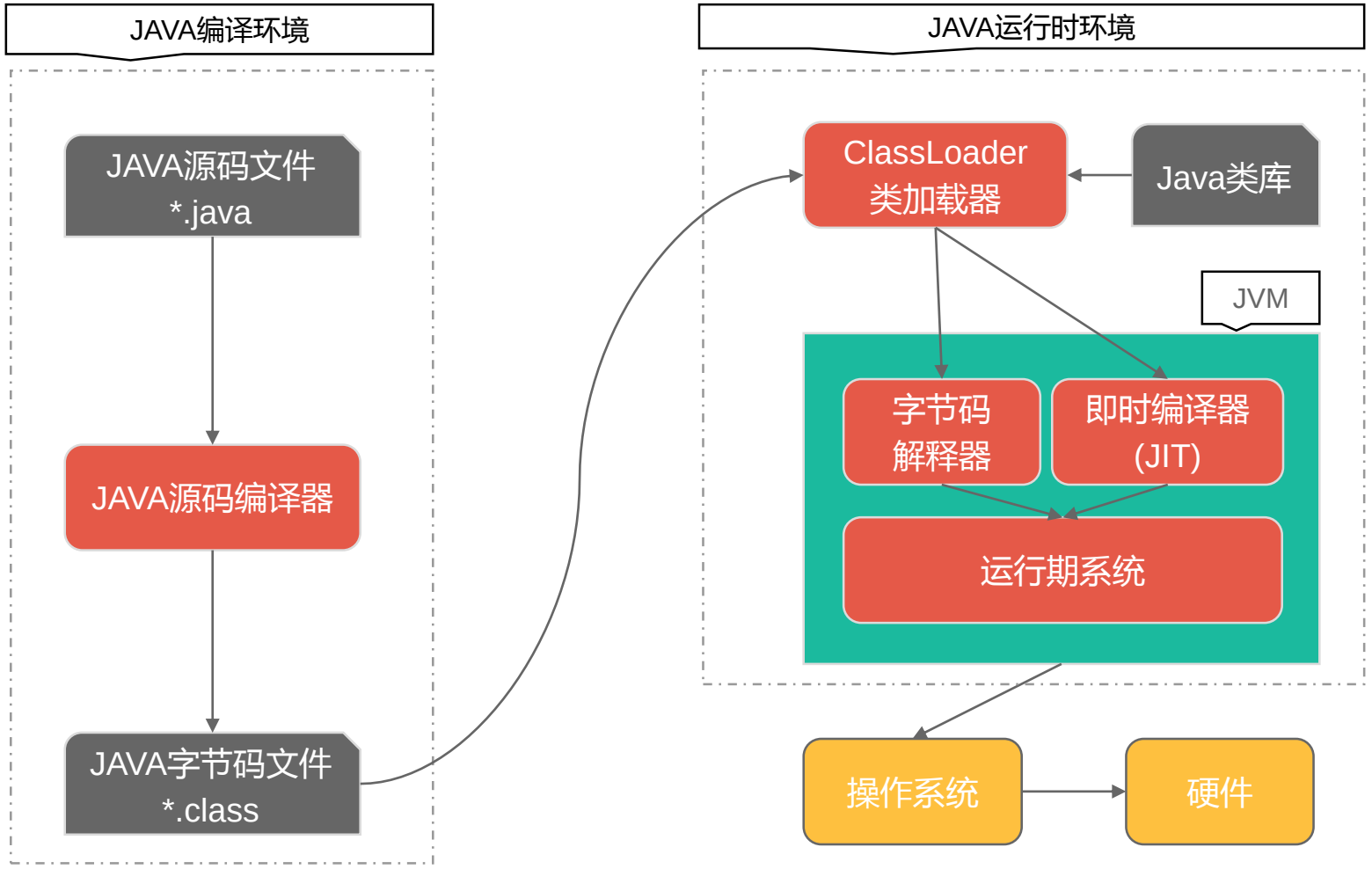


JRE加载的java平台核心类库

JRE加载的java平台扩展类库

第三方类库

JAVA代码是如何执行的



JAVA源码编译器

- **词法分析：**首先将源代码按照字节的方式读取，然后找出定义的语法关键字（if/else/for等），然后判断哪些关键字是符合java语言规范的，经过整理分析返回一些规范化的Token流
- **语法分析：**对Token流进行分析。分析出这些关键字组合在一起是否符合java语言规范，经过分析之后会产生一个符合java规范的抽象语法树。抽象语法树就是一个结构化的语法表达式，作用就是将词法用一个结构化的形式组合在一起。
- **语义分析：**对生成的抽象结构树进一步分析，将复杂的语法结构转换为简单的，易于理解和阅读的语法结构。例如：将增强for循环foreach转换为for循环结构。经过语义分析之后会产生一个更加具体的抽象结构树。

Class文件解析 - javap命令

javap查看class字节码文件

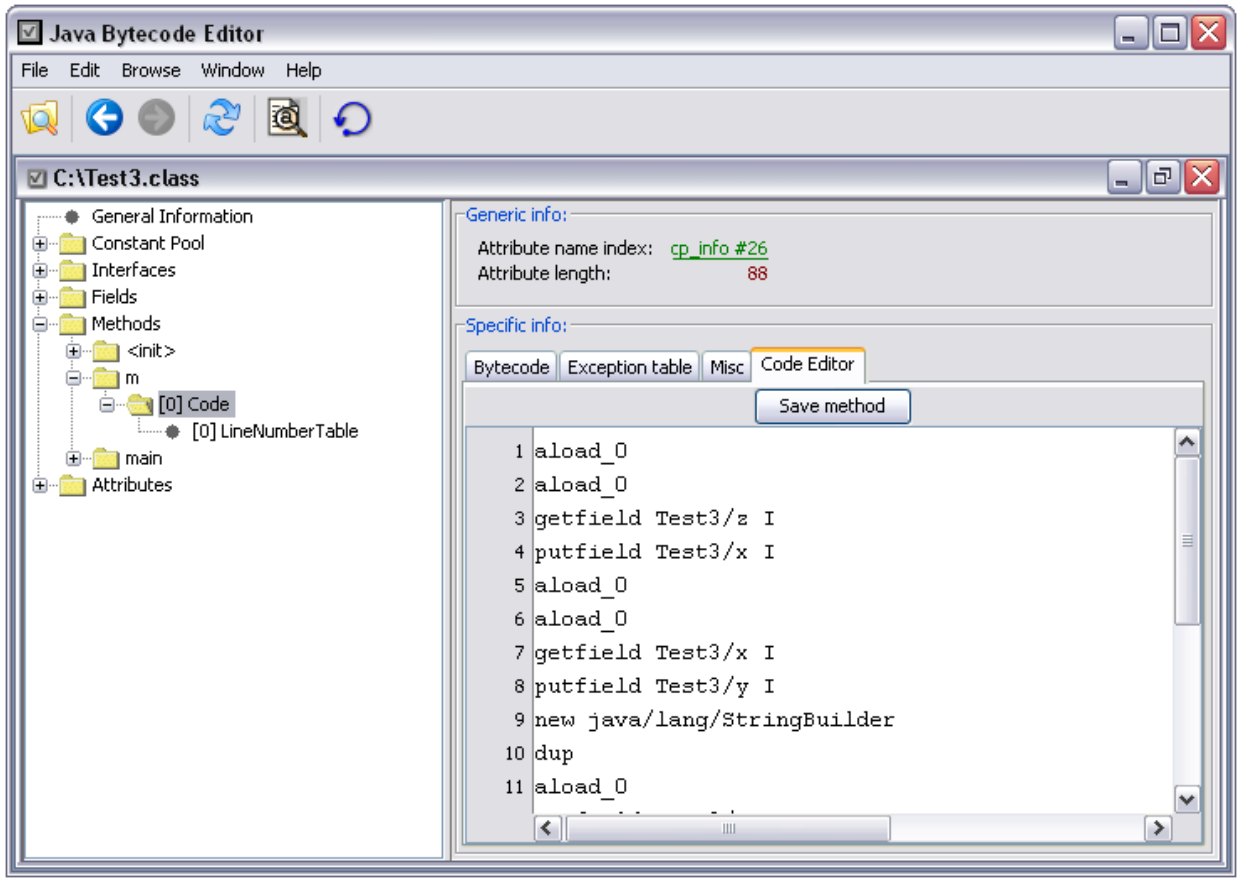
- javap是JDK提供的一个命令行工具,javap能对给定的class文件提供的字节代码进行解析、反汇编。

```
$ javap --help
用法: javap <options> <classes>
其中, 可能的选项包括:
  -help  --help  -?        输出此用法消息
  -version                 版本信息
  -v  -verbose             输出附加信息
  -l                       输出行号和本地变量表
  -public                  仅显示公共类和成员
  -protected               显示受保护的/公共类和成员
  -package                 显示程序包/受保护的/公共类
                           和成员 (默认)
  -p  -private             显示所有类和成员
  -c                       对代码进行反汇编
  -s                       输出内部类型签名
  -sysinfo                 显示正在处理的类的
                           系统信息 (路径, 大小, 日期, MD5 散列)
  -constants               显示最终常量
  -classpath <path>        指定查找用户类文件的位置
  -cp <path>               指定查找用户类文件的位置
  -bootclasspath <path>    覆盖引导类文件的位置
```

Class文件解析 - JBE

JBE： Java Bytecode Editor（Java字节码编辑器） 2M大小 解压即用

- 官网及下载地址：http://www.cs.ioc.ee/~ando/jbe/



Class文件详解-构成

构成

- Class文件是**一组以8位字节为基础单位的二进制流**，各个数据项目严格**按照顺序紧凑地排列**在Class文件之中，中间**没有添加任何分隔符**
- 根据Java虚拟机规范的规定，Class文件结构采用一种类似于C语言结构体的伪结构来存储数据，这种伪结构**只有两种数据类型：无符号数和表**

  - **无符号数**属于基本的数据类型，以u1、u2、u4、u8来表示1个字节、2个字节、4个字节和8个字节的无符号数，无符号数可以用来描述数字、索引引用、数量值或者按照UTF-8编码构成的字符串值

  - **表**是由多个无符号数或者其他表作为数据项构成的符合数据类型，所有表都习惯性地以“_info”结尾。表用于描述有层次关系的复合结构的数据，整个Class文件本质上就是一张表，它由下表所示的数据项构成

16进制查看class文件

- 本课程使用，使用Notepad++的HEX-Editor插件，以16进制的方式查看class文件。 (也可使用其他软件)
- 插件安装方法：https://blog.csdn.net/nameisbill/article/details/54616311

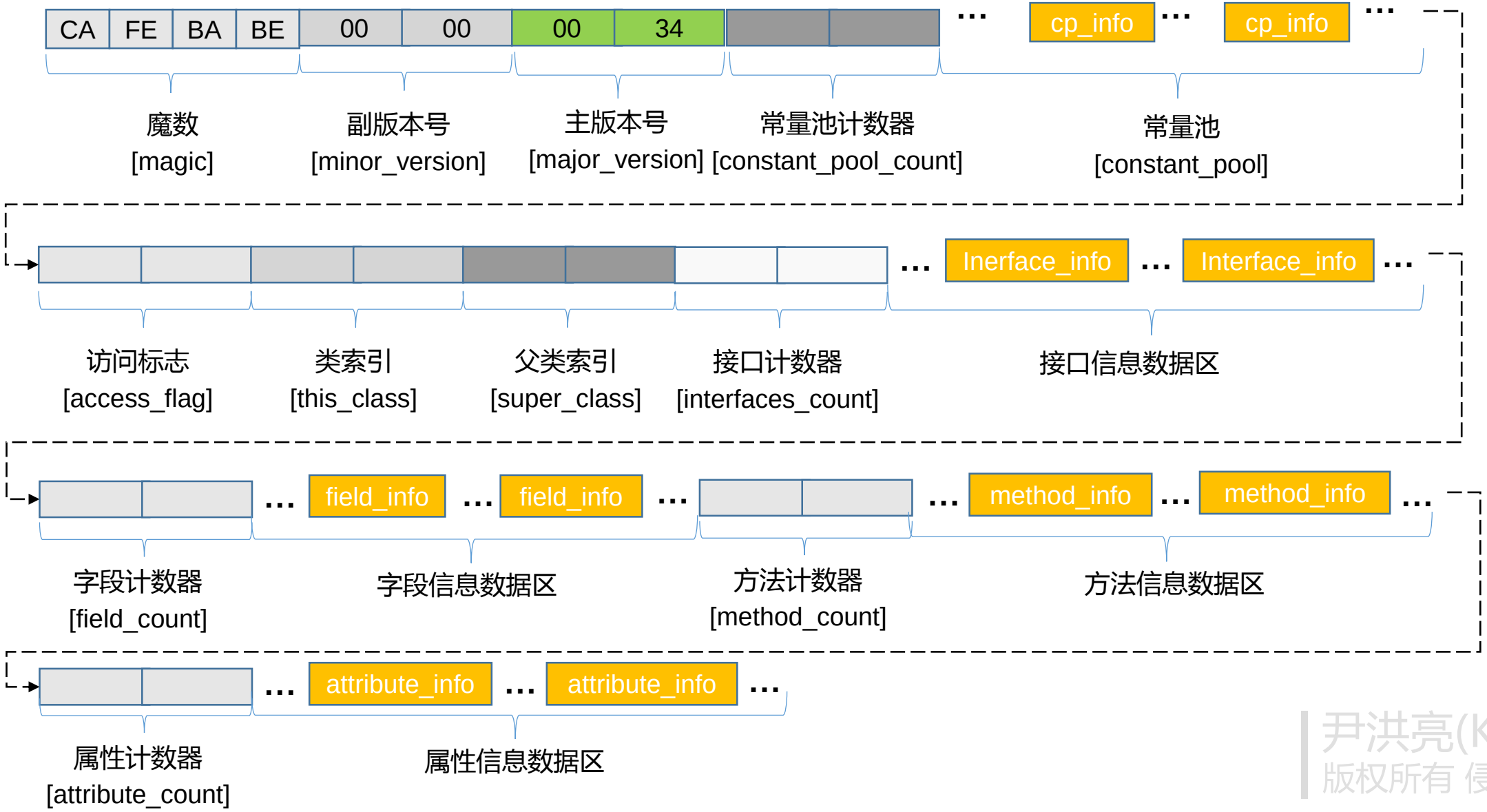
Class文件详解-构成

```
ClassFile {
    u4          magic;
    u2          minor_version;
    u2          major_version;
    u2          constant_pool_count;
    cp_info     constant_pool[constant_pool_count-1];
    u2          access_flags;
    u2          this_class;
    u2          super_class;
    u2          interfaces_count;
    u2          interfaces[interfaces_count];
    u2          fields_count;
    field_info  fields[fields_count];
    u2          methods_count;
    method_info methods[methods_count];
    u2          attributes_count;
    attribute_info attributes[attributes_count];
}
```

Class文件详解-构成

类　型	名　称	数　量
u4	magic	1
u2	minor_version	1
u2	major_version	1
u2	constant_pool_count	1
cp_info	constant_pool	constant_pool_count-1
u2	access_flags	1
u2	this_class	1
u2	super_class	1
u2	interfaces_count	1
u2	interfaces	interfaces_count
u2	fields_count	1
field_info	fields	fields_count
u2	methods_count	1
method_info	methods	methods_count
u2	attributes_count	1
attribute_info	attributes	attributes_count

Class文件详解-结构



Class文件详解-魔数

- 每一个class文件的头4个字节称为魔数，它唯一的作用是确定这个文件是否为一个能被虚拟机接受的Class文件。
- 非常多文件存储标准中都使用魔数来进行身份识别。譬如图片格式gif、jpeg等。使用魔数而不是拓展名来进行识别主要是基于安全方面的考虑，由于文件扩展名能够任意修改。

Person.class

Address	0	1	2	3	4	5	6	7	8	9	a	b	c	d	e	f	Dump
00000000	ca	fe	ba	be	00	00	00	34	00	25	07	00	02	01	00	22	澉壕...4.%.....”[
00000010	63	6f	6d	2f	6d	69	6d	61	78	75	65	79	75	61	6e	2f	com/mimaxueyuan/
00000020	6a	76	6d	2f	61	65	6e	74	69	74	79	2f	50	65	72	73	jvm/aentity/Pers
00000030	6f	6e	07	00	04	01	00	10	6a	61	76	61	2f	6c	61	6e	on.....java/lan
00000040	67	2f	4f	62	6a	65	63	74	01	00	03	61	67	65	01	00	g/Object...age..

Class文件详解-版本号

- Minor Version 次版本号 Major Version 主版本号
- **16进制文件**

Person.class

次版本 主版本

Address	0	1	2	3	4	5	6	7	8	9	a	b	c	d	e	f	Dump
00000000	ca	fe	ba	be	00	00	00	34	00	25	07	00	02	01	00	22	漱壕..4.%....."[
00000010	63	6f	6d	2f	6d	69	6d	61	78	75	65	79	75	61	6e	2f	com/mimaxueyuan/
00000020	6a	76	6d	2f	61	65	6e	74	69	74	79	2f	50	65	72	73	jvm/aentity/Pers
00000030	6f	6e	07	00	04	01	00	10	6a	61	76	61	2f	6c	61	6e	on.....java/lan
00000040	67	2f	4f	62	6a	65	63	74	01	00	03	61	67	65	01	00	g/Object...age..
00000050	01	49	01	00	04	6e	61	6d	65	01	00	12	4c	6a	61	76	.I...name...Ljav

- **javap解析 （0000转10进制=0 0034转10进制=52 ）**

```
F:\GIT\mima-jvm\target\classes\com\mimaxueyuan\jvm\aentity>javap -v Person.class
Classfile /F:/GIT/mima-jvm/target/classes/com/mimaxueyuan/jvm/aentity/Person.class
  Last modified 2019-1-11; size 936 bytes
  MD5 checksum 7e5fe3b58e58e57135a262ee7ddeef10
  Compiled from "Person.java"
public class com.mimaxueyuan.jvm.aentity.Person
  minor version: 0
  major version: 52
  flags: ACC_PUBLIC, ACC_SUPER
Constant pool:
```

Class文件详解- Major Version 与JDK版本对应关系

对照表

JDK版本	Major Version
1.1	45
1.2	46
1.3	47
1.4	48
1.5	49
1.6	50
1.7	51
1.8	52

- 异常：Unsupported major.minor version 51.0 ，文件的编译版本是JDK1.7，但是JRE版本小于1.7

Class文件详解-常量池-结构

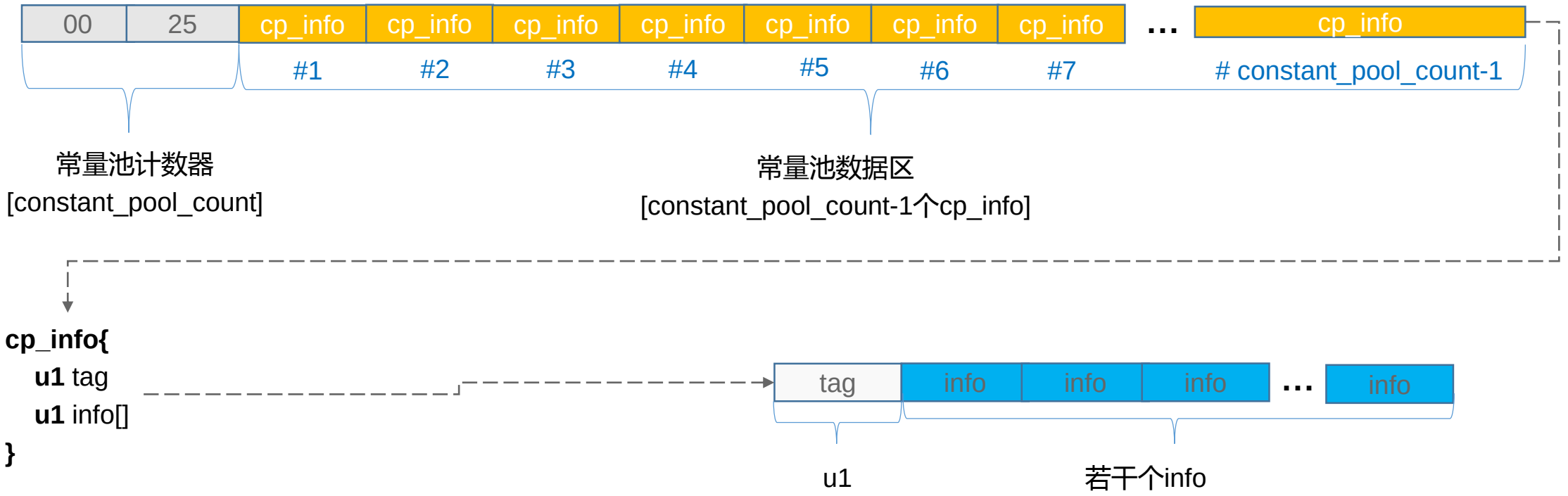
- 紧接着版本号之后的是常量池入口，常量池可以理解为Class文件之中的资源仓库，它是Class文件结构中与其他项目关联最多的数据项，也是占用Class文件空间最大的数据项目之一。常量池中存储**字面量**和**符号引用**。
- 由于常量池中的常量数量不固定，所以在常量池的入口需要放置一项u2类型的数据，代表容量池容量计数值constant_pool_count

Person.class

Address	0	1	2	3	4	5	6	7	8	9	a	b	c	d	e	f	Dump
00000000	ca	fe	ba	be	00	00	00	32	00	25	07	00	02	01	00	22	澈壕...2.%....."[
00000010	63	6f	6d	2f	6d	69	6d	61	78	75	65	79	75	61	6e	2f	com/mimaxueyuan/
00000020	6a	76	6d	2f	61	65	6e	74	69	74	79	2f	50	65	72	73	jvm/aentity/Pers
00000030	6f	6e	07	00	04	01	00	10	6a	61	76	61	2f	6c	61	6e	on......java/lan
00000040	67	2f	4f	62	6a	65	63	74	01	00	03	61	67	65	01	00	g/Object...age..

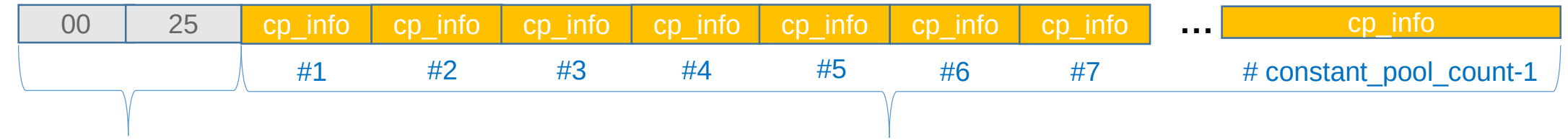
- 0026转十进制=37, 代表常量池数据区中有37-1个cp_info
- constant_pool[contstant_pool_count-1]
- 可以使用javap命令解析class文件查看

Class文件详解-常量池-结构



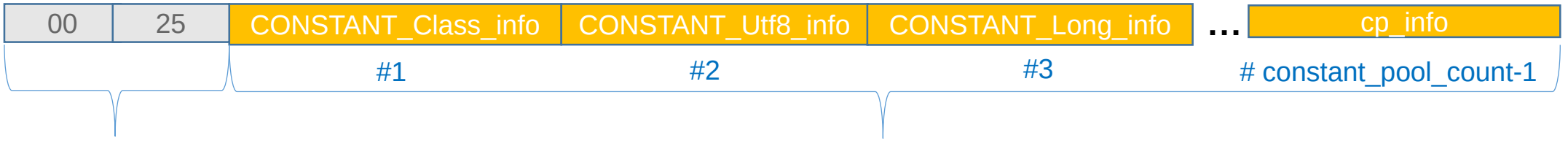
- 0025转十进制=37, 代表常量池数据区中有37-1=36个cp_info
- cp_info的个数为contstant_pool_count-1个，因为class文件规范将第0项常量空了出来，为了满足某些指向常量池的索引值的数据 “不引用任何一个常量池项”的情况。
- cp_info只有两种：字面量、
- 可以使用javap命令解析class文件查看, demo: /mima-jvm/src/main/java/com/mimaxueyuan/jvm/aentity/Person1.java

Class文件详解-常量池



常量池计数器
[constant_pool_count]

常量池数据区
[constant_pool_count-1个cp_info]



常量池计数器
[constant_pool_count]

常量池数据区
[constant_pool_count-1个cp_info]

Class文件详解-常量池-结构类型

- cp_info可以细分为多种类型，标志=tag，15、16、18的常量项是用来支持动态语言调用的（jdk1.7时才加入的）详细如下

类　　型	标　　志	描　　述
CONSTANT_Utf8_info	1	UTF-8 编码的字符串
CONSTANT_Integer_info	3	整型字面量
CONSTANT_Float_info	4	浮点型字面量
CONSTANT_Long_info	5	长整型字面量
CONSTANT_Double_info	6	双精度浮点型字面量
CONSTANT_Class_info	7	类或接口的符号引用
CONSTANT_String_info	8	字符串类型字面量
CONSTANT_Fieldref_info	9	字段的符号引用
CONSTANT_Methodref_info	10	类中方法的符号引用
CONSTANT_InterfaceMethodref_info	11	接口中方法的符号引用
CONSTANT_NameAndType_info	12	字段或方法的部分符号引用
CONSTANT_MethodHandle_info	15	表示方法句柄
CONSTANT_MethodType_info	16	标识方法类型
CONSTANT_InvokeDynamic_info	18	表示一个动态方法调用点

字面量

符号引用

Class文件详解-常量池-字面量和符号引用

字面量

- 文本字符串
- 被声明为final的常量值
- 基本数据类型的值

符号引用

- 类和结构的全限定名
- 字段名称和描述符
- 方法名称和描述符

Class文件详解-常量池

常　量	项　目	类　型	描　述
CONSTANT_Utf8_info	tag	u1	值为 1
	length	u2	UTF-8 编码的字符串占用的字节数
	bytes	u1	长度为 length 的 UTF-8 编码的字符串
CONSTANT_Integer_info	tag	u1	值为 3
	bytes	u4	按照高位在前存储的 int 值
CONSTANT_Float_info	tag	u1	值为 4
	bytes	u4	按照高位在前存储的 float 值
CONSTANT_Long_info	tag	u1	值为 5
	bytes	u8	按照高位在前存储的 long 值
CONSTANT_Double_info	tag	u1	值为 6
	bytes	u8	按照高位在前存储的 double 值
CONSTANT_Class_info	tag	u1	值为 7
	index	u2	指向全限定名常量项的索引
CONSTANT_String_info	tag	u1	值为 8
	index	u2	指向字符串字面量的索引

Class文件详解-常量池

CONSTANT_Fieldref_info	tag	u1	值为 9
	index	u2	指向声明字段的类或者接口描述符 CONSTANT_Class_info 的索引项
	index	u2	指向字段描述符 CONSTANT_NameAndType 的索引项
CONSTANT_Methodref_info	tag	u1	值为 10
	index	u2	指向声明方法的类描述符 CONSTANT_Class_info 的索引项
	index	u2	指向名称及类型描述符 CONSTANT_NameAndType 的索引项
CONSTANT_Interface-Methodref_info	tag	u1	值为 11
	index	u2	指向声明方法的接口描述符 CONSTANT_Class_info 的索引项
	index	u2	指向名称及类型描述符 CONSTANT_NameAndType 的索引项

Class文件详解-常量池

常　量	项　目	类　型	描　述
CONSTANT_Name-AndType_info	tag	u1	值为 12
	index	u2	指向该字段或方法名称常量项的索引
	index	u2	指向该字段或方法描述符常量项的索引
CONSTANT_Method-Handle_info	tag	u1	值为 15
	reference_kind	u1	值必须在 1 ～ 9 之间（包括 1 和 9），它决定了方法句柄的类型。方法句柄类型的值表示方法句柄的字节码行为
	reference_index	u2	值必须是对常量池的有效索引
CONSTANT_Method-Type_info	tag	u1	值为 16
	descriptor_index	u2	值必须是对常量池的有效索引，常量池在该索引处的项必须是 CONSTANT_Utf8_info 结构，表示方法的描述符
CONSTANT_Invoke-Dynamic_info	tag	u1	值为 18
	bootstrap_method_attr_index	u2	值必须是对当前 Class 文件中引导方法表的 bootstrap_methods[] 数组的有效索引
	name_and_type_index	u2	值必须是对当前常量池的有效索引，常量池在该索引处的项必须是 CONSTANT_NameAndType_info 结构，表示方法名和方法描述符

Class文件详解-常量池

Address	0	1	2	3	4	5	6	7	8	9	a	b	c	d	e	f	Dump
00000000	ca	fe	ba	be	00	00	00	32	00	25	07	00	02	01	00	22	漱壕...2.%....."[
00000010	63	6f	6d	2f	6d	69	6d	61	78	75	65	79	75	61	6e	2f	com/mimaxueyuan/
00000020	6a	76	6d	2f	61	65	6e	74	69	74	79	2f	50	65	72	73	jvm/aentity/Pers
00000030	6f	6e	07	00	04	01	00	10	6a	61	76	61	2f	6c	61	6e	on......java/lan
00000040	67	2f	4f	62	6a	65	63	74	01	00	03	61	67	65	01	00	g/Object...age..
00000050	01	49	01	00	04	6e	61	6d	65	01	00	12	4c	6a	61	76	.I...name...Ljav
00000060	61	2f	6c	61	6e	67	2f	53	74	72	69	6e	67	3b	01	00	a/lang/String;..

- **红色区域**
- 0x07=7=CONSTANT_Class_info
- 0x0002=2 代表指向#2常量项的索引
- **蓝色区域**
- 0x01=1=CONSTANT_Utf8_info
- 0x0022=34=长度length
- 后续区域=com/mimaxueyuan/jvm/aentity/Person

Class文件详解-常量池

```
Constant pool:
   #1 = Class              #2             // com/mimaxueyuan/jvm/aentity/Person
   #2 = Utf8               com/mimaxueyuan/jvm/aentity/Person
   #3 = Class              #4             // java/lang/Object
   #4 = Utf8               java/lang/Object
   #5 = Utf8               age
   #6 = Utf8               I
   #7 = Utf8               name
   #8 = Utf8               Ljava/lang/String;
   #9 = Utf8               address
  #10 = Utf8               <init>
  #11 = Utf8               ()V
  #12 = Utf8               Code
  #13 = Methodref          #3.#14         // java/lang/Object."<init>":()V
  #14 = NameAndType        #10:#11        // "<init>":()V
  #15 = Utf8               LineNumberTable
  #16 = Utf8               LocalVariableTable
  #17 = Utf8               this
  #18 = Utf8               Lcom/mimaxueyuan/jvm/aentity/Person;
  #19 = Utf8               getAge
  #20 = Utf8               ()I
  #21 = Fieldref           #1.#22         // com/mimaxueyuan/jvm/aentity/Person.age:I
  #22 = NameAndType        #5:#6          // age:I
  #23 = Utf8               setAge
  #24 = Utf8               (I)V
  #25 = Utf8               getName
  #26 = Utf8               ()Ljava/lang/String;
  #27 = Fieldref           #1.#28         // com/mimaxueyuan/jvm/aentity/Person.name:Ljava/lang/String;
  #28 = NameAndType        #7:#8          // name:Ljava/lang/String;
  #29 = Utf8               setName
  #30 = Utf8               (Ljava/lang/String;)V
  #31 = Utf8               getAddress
  #32 = Fieldref           #1.#33         // com/mimaxueyuan/jvm/aentity/Person.address:Ljava/lang/String;
  #33 = NameAndType        #9:#8          // address:Ljava/lang/String;
  #34 = Utf8               setAddress
  #35 = Utf8               SourceFile
  #36 = Utf8               Person.java
```

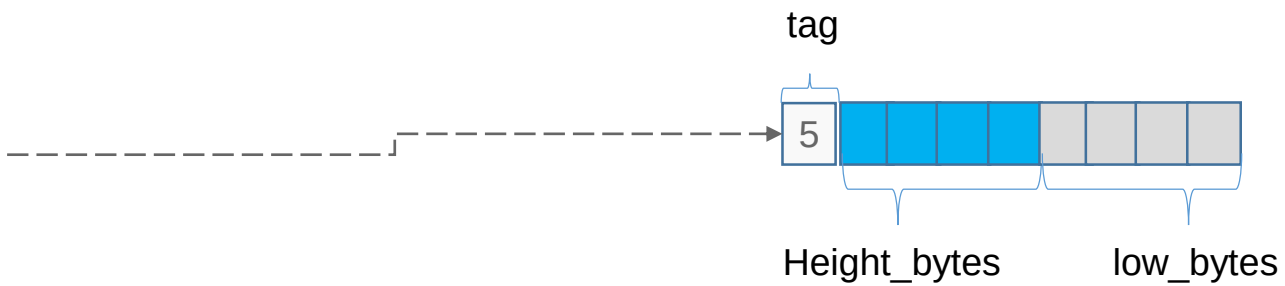
Class文件详解-常量池-基本数据类型

CONSTANT_Integer_info{
    u1 tag=3
    u4 bytes
}

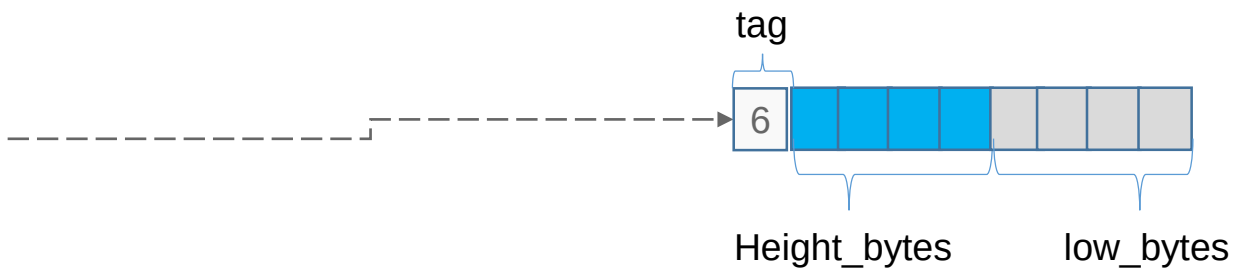
CONSTANT_Float_info{
    u1 tag=4
    u4 bytes
}

Class文件详解-常量池-基本数据类型

CONSTANT_Long_info{
    u1 tag=5
    u4 height_bytes
    u4 low_bytes
}

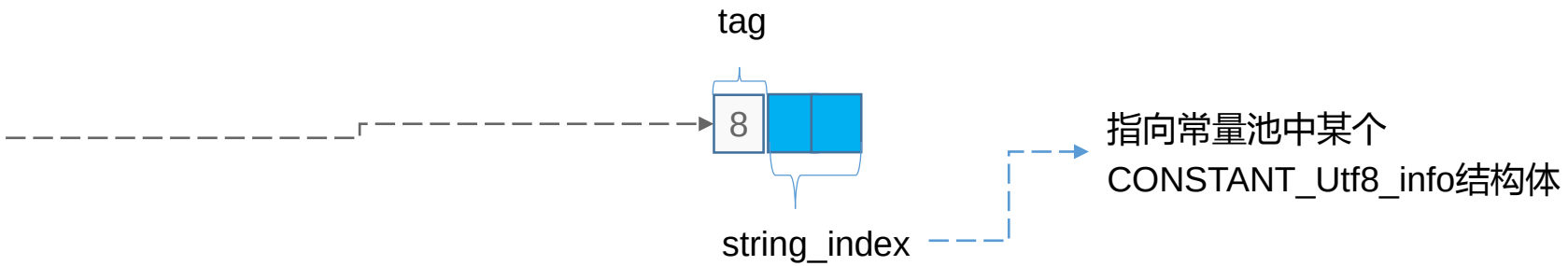


CONSTANT_Double_info{
    u1 tag=6
    u4 height_bytes
    u4 low_bytes
}

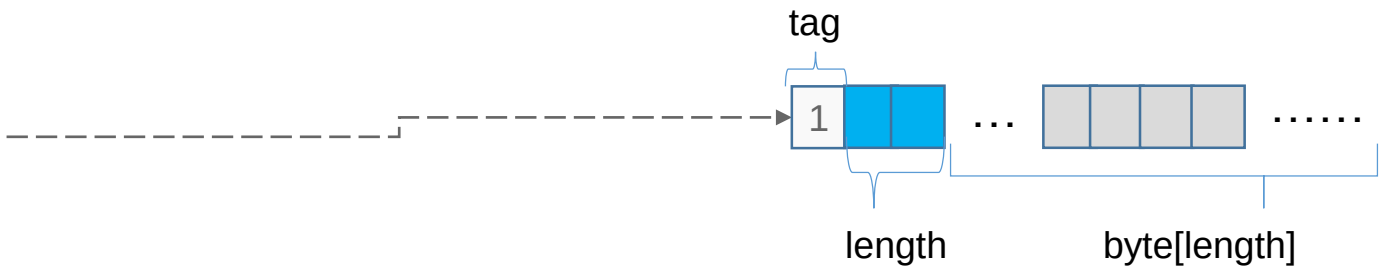


Class文件详解-常量池-字符

CONSTANT_String_info{
    u1 tag=8
    u2 string_index
}

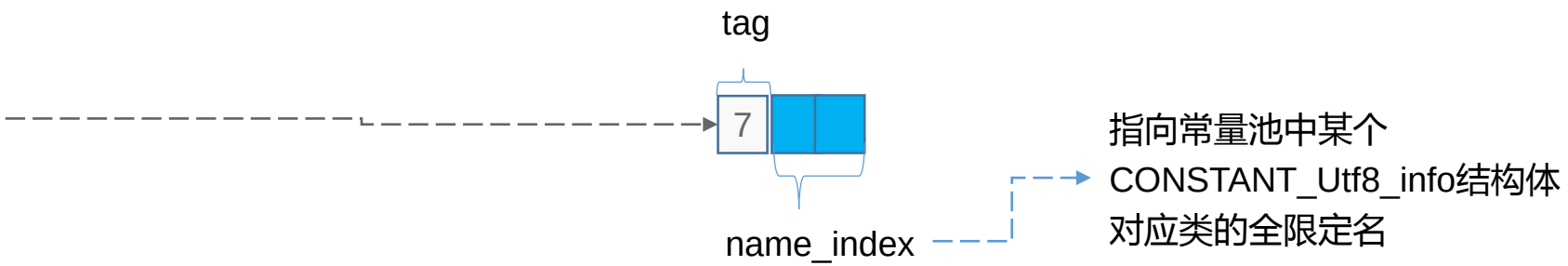


CONSTANT_Utf8_info{
    u1 tag=1
    u2 length
    u1 byte[length]
}



Class文件详解-常量池-类

```
CONSTANT_Class_info{
    u1 tag=7
    u2 name_index
}
```

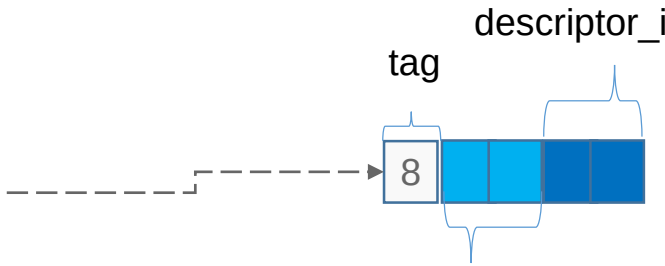


Javap demo

- *com.mimaxueyuan.jvm.bytecode.ConstantPool1*
- *com.mimaxueyuan.jvm.bytecode.ConstantPool2*

Class文件详解-常量池-字段

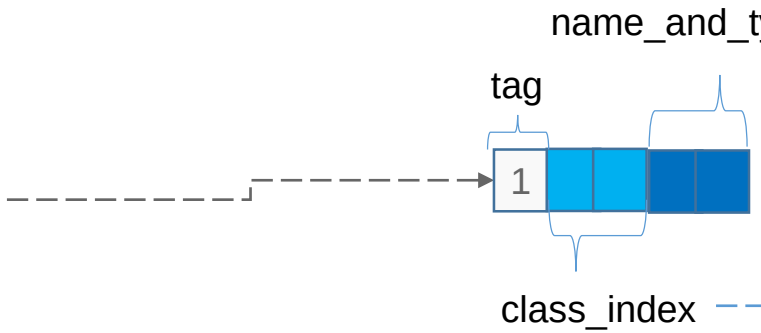
CONSTANT_NameAndType_info{
    u1 tag=12
    u2 name_index
    u2 descriptor_index
}



指向常量池中某个
CONSTANT_Utf8_info结构体
表示字段类型或方法返回类型

指向常量池中某个
CONSTANT_Utf8_info结构体
表示字段或方法的名称

CONSTANT_Fiedref_info{
    u1 tag=9
    u2 class_index
    u1 name_and_type_index
}



指向常量池中某个
CONSTANT_NammeAndType_info
结构体, 表示字段名和类型

指向常量池中某个
CONSTANT_Class_info
结构体, 表示字段所属的类

Class文件详解-常量池-字段

```
Constant pool:
   #1 = Class              #2             // com/mimaxueyuan/jvm/aentity/Person
   #2 = Utf8               com/mimaxueyuan/jvm/aentity/Person
   #3 = Class              #4             // java/lang/Object
   #4 = Utf8               java/lang/Object
   #5 = Utf8               age
   #6 = Utf8               I
   #7 = Utf8               name
   #8 = Utf8               Ljava/lang/String;
   #9 = Utf8               address
  #10 = Utf8               <init>
  #11 = Utf8               ()V
  #12 = Utf8               Code
  #13 = Methodref          #3.#14         // java/lang/Object."<init>":()V
  #14 = NameAndType        #10:#11        // "<init>":()V
  #15 = Utf8               LineNumberTable
  #16 = Utf8               LocalVariableTable
  #17 = Utf8               this
  #18 = Utf8               Lcom/mimaxueyuan/jvm/aentity/Person;
  #19 = Utf8               getAge
  #20 = Utf8               ()I
  #21 = Fieldref           #1.#22         // com/mimaxueyuan/jvm/aentity/Person.age:I
  #22 = NameAndType        #5:#6          // age:I
  #23 = Utf8               setAge
  #24 = Utf8               (I)V
  #25 = Utf8               getName
  #26 = Utf8               ()Ljava/lang/String;
  #27 = Fieldref           #1.#28         // com/mimaxueyuan/jvm/aentity/Person.name:Ljava/lang/String;
  #28 = NameAndType        #7:#8          // name:Ljava/lang/String;
  #29 = Utf8               setName
  #30 = Utf8               (Ljava/lang/String;)V
  #31 = Utf8               getAddress
  #32 = Fieldref           #1.#33         // com/mimaxueyuan/jvm/aentity/Person.address:Ljava/lang/String;
  #33 = NameAndType        #9:#8          // address:Ljava/lang/String;
  #34 = Utf8               setAddress
  #35 = Utf8               SourceFile
  #36 = Utf8               Person.java
```

1

Class文件详解-常量池-字段类型

1、基本数据类型

- B 表示byte，有符号的字节类型
- C 表示char，Unicode字符，UTF-16编码
- D 表示double，双精度浮点数
- F 表示float，单精度浮点数
- I 表示int，整型数
- J 表示long，长整型数
- S 表示short，有符号短整数
- Z 表示boolean，布尔

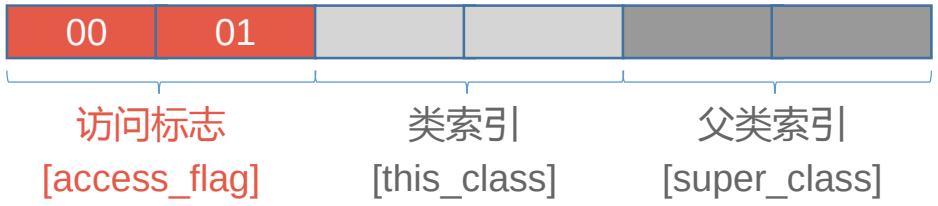
2、对象引用类型

- LClassname； 表示一个Classname的实例，例如Ljava/lang/String 代表一个String对象

3、数组引用类型

- [Type 表示一个类型为Type的数组，例如[Ljava/lang/String 代表一个String数组对象，Type=Ljava/lang/String，每一维将使用一个前置的“[”字符来描述，如："int[]"将被记录为"[I","String[][]"将被记录为"[[Ljava/lang/String;"
- *Javap Demo: com.mimaxueyuan.jvm.bytecode.ConstantPool3*

Class文件详解-access_flag



访问标志，*access_flags* ，用于表示某个类或者接口的访问权限及基础属性

标志名	标志值	标志含义	针对的对像
ACC_PUBLIC	0x0001	public类型	所有类型
ACC_FINAL	0x0010	final类型	类
ACC_SUPER	0x0020	使用新的invokespecial语义	类和接口
ACC_INTERFACE	0x0200	接口类型	接口
ACC_ABSTRACT	0x0400	抽象类型	类和接口
ACC_SYNTHETIC	0x1000	该类不由用户代码生成	所有类型
ACC_ANNOTATION	0x2000	注解类型	注解
ACC_ENUM	0x4000	枚举类型	枚举

- javap demo：*com.mimaxueyuan.jvm.bytecode.AccessFlagDemo0*

invokespecial是一个字节码指令， 用于调用一个方法，一般情况下， 调用构造方法或者使用super关键字显示调用父类的方法时， 会使用这条字节码指令。 这正是ACC_SUPER这个名字的由来。 在java 1.2之前， invokespecial对方法的调用都是静态绑定的， 而ACC_SUPER这个标志位在java 1.2的时候加入到class文件中， 它为invokespecial这条指令增加了动态绑定的功能。

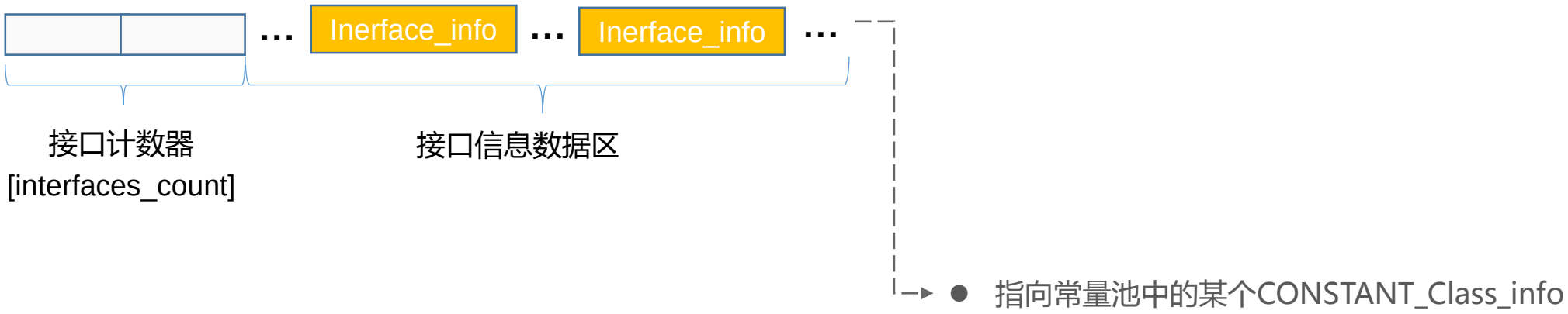
Class文件详解-this_class、super_class

- 常量池中的一个CONSTANT_Class_info，代表当前类的父类
- 常量池中的一个CONSTANT_Class_info，代表当前类

Class文件详解-interface

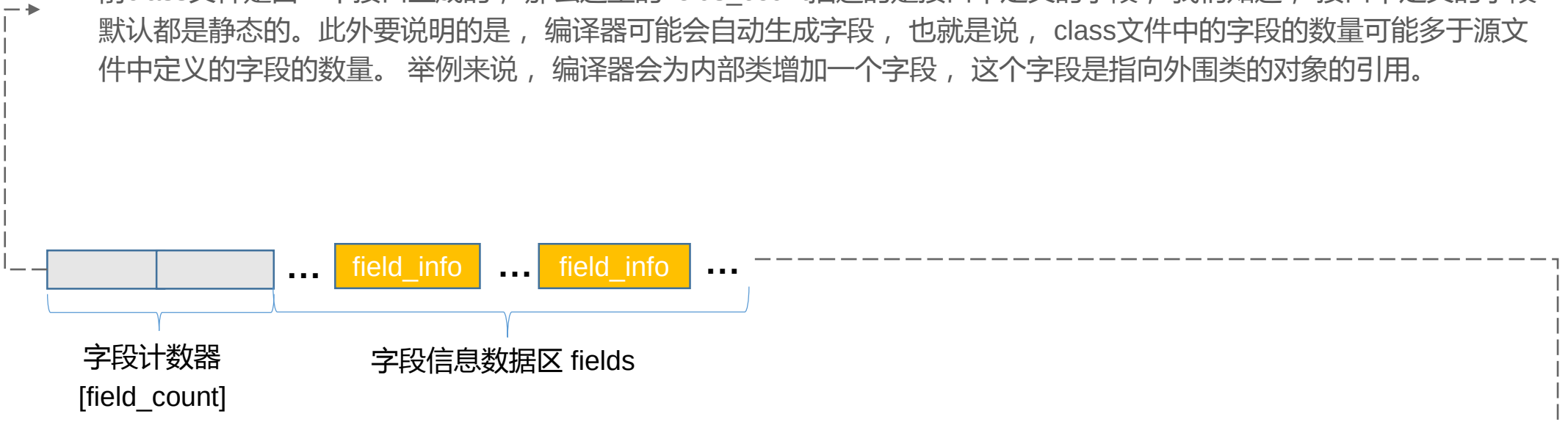
- interface_count表示当前类所实现的接口的数量或者当前接口所继承的超接口的数量。 注意， 只有当前类直接实现的接口才会被统计， 如果当前类继承了另一个类， 而另一个类又实现了一个接口， 那么这个接口不会统计在当前类的

- interfaces， 他可以看做是一个数组， 其中的每个数组项是一个索引， 指向常量池中的一个CONSTANT_Class_info



Class文件详解-field

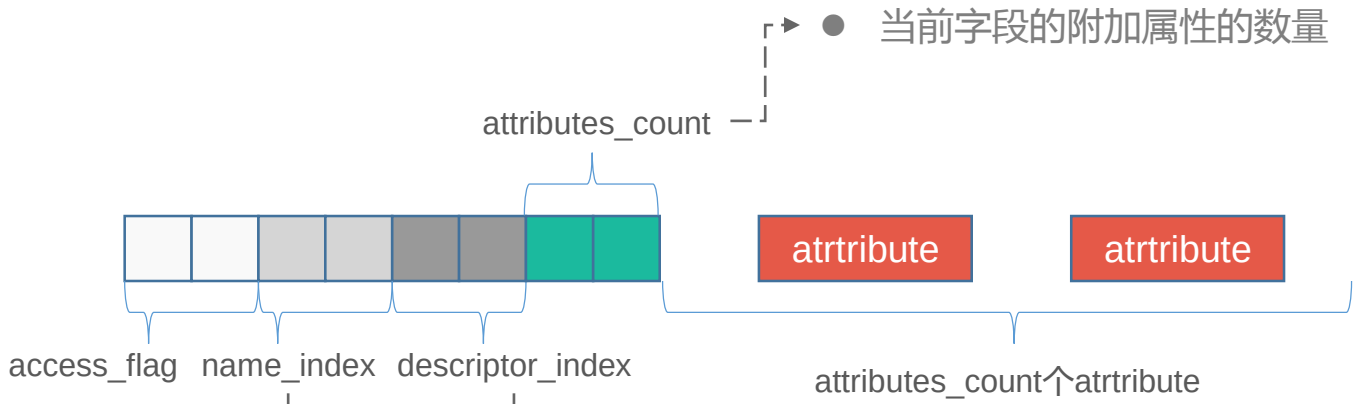
- fields_count描述的是当前的类中定义的字段的个数， 注意， 这里包括静态字段， 但不包括从父类继承的字段。 如果当前class文件是由一个接口生成的， 那么这里的fields_count描述的是接口中定义的字段， 我们知道， 接口中定义的字段默认都是静态的。此外要说明的是， 编译器可能会自动生成字段， 也就是说， class文件中的字段的数量可能多于源文件中定义的字段的数量。 举例来说， 编译器会为内部类增加一个字段， 这个字段是指向外围类的对象的引用。



- fields， 可以把它看做一个数组， 数组中的每一项是一个field_info 。这个数组中一共有fields_count个field_info ， 每个field_info都是对一个字段的描述。 下面我们详细讲解field_info的结构。 每个field_info的结构如下：

Class文件详解-field_info

```
field_info {
  u2 access_flags;
  u2 name_index;
  u2 descriptor_index;
  u2 attributes_count;
  attribute_info attributes[attributes_count];
}
```

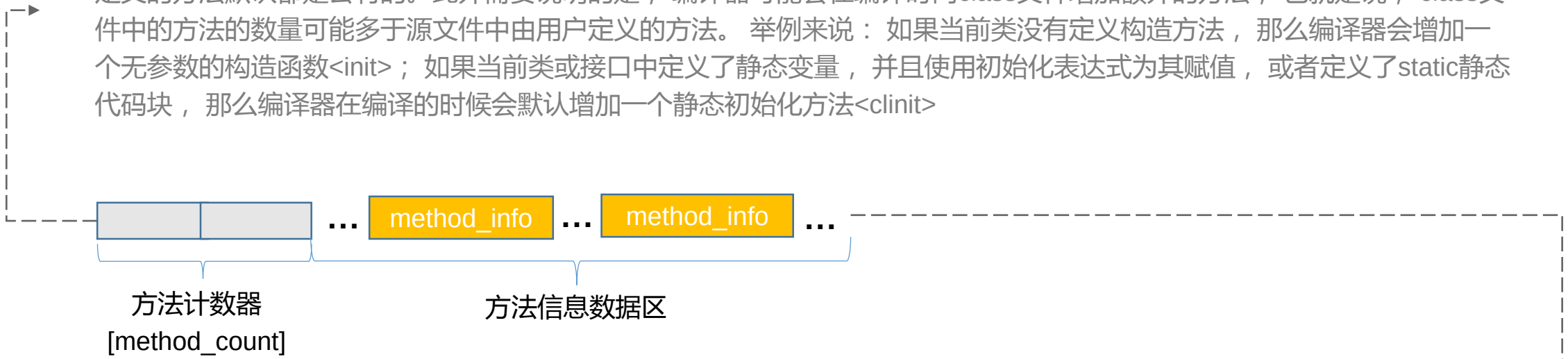


- 必须是对常量池的一个有效索引。常量池在该索引处的项必须是CONSTANT_Utf8_info结构，表示一个有效的字段的非全限定名

- 必须是对常量池的一个有效索引。常量池在该索引处的项必须是CONSTANT_Utf8_info结构，表示一个有效的字段的描述符。

Class文件详解-method

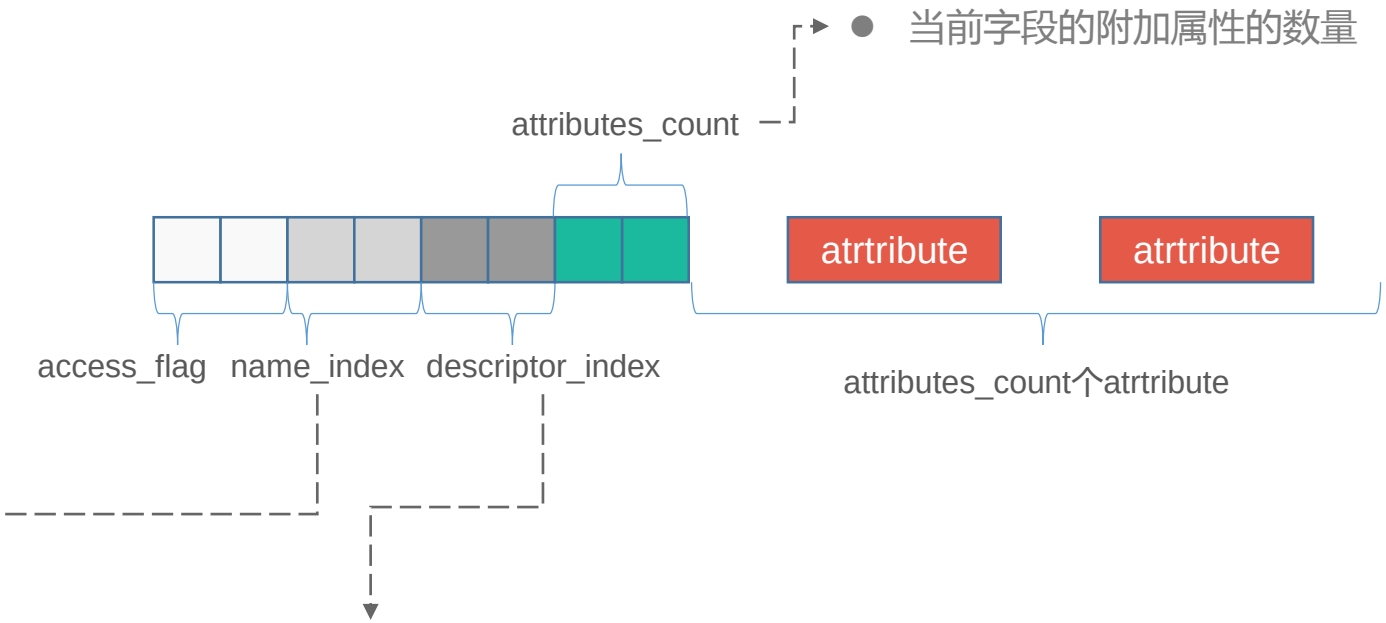
- methods_count描述的是当前的类中定义的方法的个数， 注意， 这里包括静态方法， 但不包括从父类继承的方法。 如果当前class文件是由一个接口生成的， 那么这里的methods_count描述的是接口中定义的抽象方法的数量， 我们知道， 接口中定义的方法默认都是公有的。此外需要说明的是， 编译器可能会在编译时向class文件增加额外的方法， 也就是说， class文件中的方法的数量可能多于源文件中由用户定义的方法。 举例来说： 如果当前类没有定义构造方法， 那么编译器会增加一个无参数的构造函数<init>； 如果当前类或接口中定义了静态变量， 并且使用初始化表达式为其赋值， 或者定义了static静态代码块， 那么编译器在编译的时候会默认增加一个静态初始化方法<clinit>



- 数组中的每一项是一个method_info 。这个数组中一共有methods_count个method_info ， 每个method_info 都是对一个方法的描述

Class文件详解-method_info

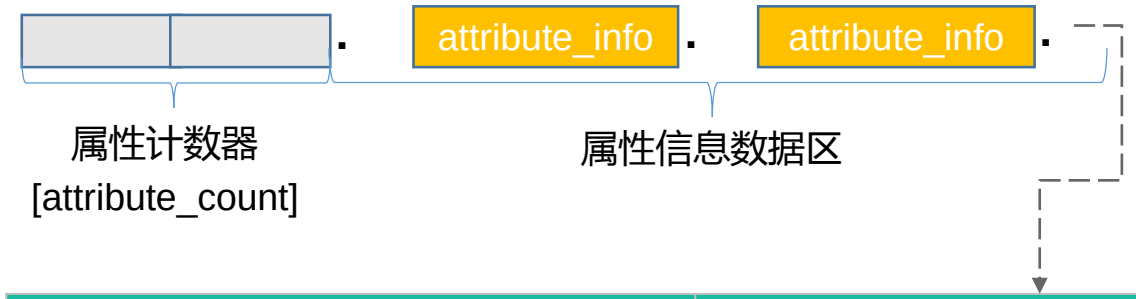
```
method_info {
    u2 access_flags;
    u2 name_index;
    u2 descriptor_index;
    u2 attributes_count;
    attribute_info attributes[attributes_count];
}
```



- 必须是对常量池的一个有效索引。常量池在该索引处的项必须是CONSTANT_Utf8_info结构，表示一个有效的字段的非全限定名

- 必须是对常量池的一个有效索引。常量池在该索引处的项必须是CONSTANT_Utf8_info结构，表示一个有效的字段的描述符。

Class文件详解-attribute



属性名称	使用位置	含义
Code	方法表	Java代码编译成的字节码指令
ConstantValue	字段表	final关键字定义的常量值
Deprecated	类文件、字段表、方法表	被声明为deprecated的方法和字段
Exceptions	方法表	方法抛出的异常
InnerClasses	类文件	内部类列表
LineNumberTale	Code属性	Java源码的行号与字节码指令的对应关系
LocalVariableTable	Code属性	方法的局部变量描述
SourceFile	类文件	源文件名称
Synthetic	类文件、方法表、字段表	标识方法或字段是由编译器自动生成的

Class文件详解-attribute-Code

```
Code {
    u2 attribute_name_index;
    u4 attribute_length;
    u2 max_stack;
    u2 max_locals;
    u4 code_length;
    u1 code[code_lenght];
    u2 exception_table_length;
    exception_info  exception_table[exception_table_length];
    u2 attributes_count;
    attribute_info attributes[attributes_count];
}
```

- max_stack：操作数栈深度最大值，在方法执行的任何时刻，操作数栈深度都不会超过这个值。虚拟机运行时根据这个值来分配栈帧的操作数栈深度
- max_locals：局部变量表所需存储空间，单位为Slot。并不是所有局部变量占用的Slot之和，当一个局部变量的生命周期结束后，其所占用的Slot将分配给其它依然存活的局部变量使用，按此方式计算出方法运行时局部变量表所需的存储空间
  Slot，虚拟机为局部变量分配内存所使用的最小单位，长度不超过32位的数据类型占用1个Slot，64位的数据类型（long和double）占用2个Slot
- code_length和code：用来存放Java源程序编译后生成的字节码指令。code_length代表字节码长度，code是用于存储字节码指令的一系列字节流。
- 每一个指令是一个u1类型的单字节，当虚拟机读到code中的一个字节码（一个字节能表示256种指令，Java虚拟机规范定义了其中约200个编码对应的指令），就可以判断出该字节码代表的指令，指令后面是否带有参数，参数该如何解释，虽然code_length占4个字节，但是Java虚拟机规范中限制一个方法不能超过65535条字节码指令，如果超过，Javac将拒绝编译

Javap demo: com.mimaxueyuan.jvm.bytecode.AttributeDemo0

Class文件详解-attribute-Exceptions

```
Exceptions{
    u2 attribute_name_index;
    u4 attribute_length;
    u2 number_of_exceptions;
    u2 exception_index_table[number_of_exceptions]
}
```

- 列举出方法中可能抛出的受查异常（即方法描述时throws关键字后列出的异常），与Code属性平级
- number_of_exceptions表示可能抛出number_of_exceptions种受查异常
- exception_index_table为异常索引集合，一组u2类型exception_index的集合，每一个exception_index为一个指向常量池中一CONSTANT_Class_info型常量的索引，代表该受查异常的类型
- Javap demo: com.mimaxueyuan.jvm.bytecode.AttributeDemo0.getAge()

Class文件详解-attribute-Deprecated和Synthetic

```
Deprecated{
    u2 attribute_name_index;
    u4 attribute_length;
}
```

```
Synthetic{
    u2 attribute_name_index;
    u4 attribute_length;
}
```

- 这两个属性都属于标志类型的布尔属性，只存在有和没有的区别，没有属性值的概念
- Deprecated属性表示某个类、字段或方法已经被程序作者定为不再推荐使用，可在代码中使用@Deprecated注解进行设置
- Synthetic属性表示该字段或方法不是由Java源码直接产生的，而是由编译器自行添加的

- **Javap demo: @Deprecated**
- com.mimaxueyuan.jvm.bytecode.AttributeDemo0.getAge()

Class文件详解-attribute-ConstantValue

```
ConstantValue{
    u2 attribute_name_index;
    u4 attribute_length;
    u2 constantvalue_index;
}c
```

- ConstantValue属性是一个定长属性，其中attribute_length的值固定为0x00000002，constantvalue_index为一常量池字面量类型常量索引（Class文件格式的常量类型中只有与基本类型和字符串类型相对应的字面量常量，所以ConstantValue属性只支持基本类型和字符串类型）
- 对非static类型变量（实例变量，如：int a = 123;）的赋值是在实例构造器<init>方法中进行的
- 对类变量（如：static int a = 123;）的赋值有2种选择，在类构造器<clinit>方法中或使用ConstantValue属性。当前Javac编译器的选择是：如果变量同时被static和final修饰（虚拟机规范只要求有ConstantValue属性的字段必须设置ACC_STATIC标志，对final关键字的要求是Javac编译器自己加入的要求），并且该变量的数据类型为基本类型或字符串类型，就生成ConstantValue属性进行初始化；否则在类构造器<clinit>方法中进行初始化

- Javap demo:
- com.mimaxueyuan.jvm.bytecode.AttributeDemo0.sex
- com.mimaxueyuan.jvm.bytecode.AttributeDemo0.price
- com.mimaxueyuan.jvm.bytecode.AttributeDemo0.salary

Class文件详解-attribute-InnerClasses

```
InnerClasses{
    u2 attribute_name_index;
    u4 attribute_length;
    u2 number_of_classes;
    inner_classes inner_classes_info [number_of_classes]
}
```

该属性用于记录内部类和宿主类之间的关系。如果一个类中定义了内部类，编译器将会为这个类与这个类包含的内部类生成InnerClasses属性

```
inner_classes_info{
    u2 inner_class_info_index;
    u4 outer_class_info_index;
    u2 inner_name_index;
    u2 inner_name_access_flags;
}
```

inner_class_info_index和outer_class_info_index指向常量池中CONSTANT_Class_info类型常量索引

inner_name_index指向常量池中CONSTANT_Utf8_info类型常量的索引，为内部类名称，如果为匿名内部类，则该值为0

inner_name_access_flags类似于access_flags，是内部类的访问标志

- Javap demo: com.mimaxueyuan.jvm.bytecode.AttributeDemo0.Email

Class文件详解-attribute-SourceFile

```
SourceFile{
    u2 attribute_name_index;
    u4 attribute_length;
    u2 sourcefile_index;
}
```

- sourcefile_index是指向常量池中一CONSTANT_Utf8_info类型常量的索引，常量的值为源码文件的文件名
- 对大多数文件，类名和文件名是一致的，少数特殊类除外（如：内部类），此时如果不生成这项属性，当抛出异常时，堆栈中将不会显示出错误代码所属的文件名

- **Javap demo: 内部类**
- javap -v -private AttributeDemo0$Email.class

Class文件详解-attribute- LineNumberTale

```
LineNumberTale{
    u2 attribute_name_index;
    u4 attribute_length;
    u2 line_number_table_length;
    line_number_table line_number_info[line_number_table_length];
}
```

用于描述Java源码的行号与字节码行号之间的对应关系，非运行时必需属性，会默认生成至Class文件中，可以使用Javac的-g:none关闭或要求生成该项属性信息

```
line_number_info{
    u2 start_pc;
    u2 line_number;
}
```

- 字节码行号
- JAVA源码行号

- 不生成该属性的最大影响是：
- 1，抛出异常时，堆栈将不会显示出错的行号；
- 2，调试程序时无法按照源码设置断点

- 不生成LineNumberTable属性demo
  - javac -g:none AttributeDemo0.java
  - javap -v -private AttributeDemo0.class

Class文件详解-attribute-LocalVariableTable

```
LocalVariableTable{
    u2 attribute_name_index;
    u4 attribute_length;
    u2 local_variable_table_length;
    local_variable_table local_variable_info[local_variable_table_length];
}
```

用于描述栈帧中局部变量表中的变量与Java源码中定义的变量之间的关系，非运行时必需属性，默认不会生成至Class文件中，可以使用Javac的-g:none关闭或要求生成该项属性信息

```
local_variable_info{
    u2 start_pc;
    u2 length;
    u2 name_index;
    u2 descriptor_index;
    u2 index;
}
```

- 局部变量的生命周期开始的字节码偏移量
- 局部变量作用范围覆盖的长度, start_pc + length即为该局部变量在字节码中的作用域范围
- 指向常量池中CONSTANT_Utf8_info类型常量的索引，局部变量名称
- 指向常量池中CONSTANT_Utf8_info类型常量的索引，局部变量描述符
- 局部变量在栈帧局部变量表中Slot的位置，如果这个变量的数据类型为64位类型（long或double），它占用的Slot为index和index+1这2个位置

Class文件详解-attribute-LocalVariableTable

不生成该属性的最大影响是：

- 1，当其他人引用这个方法时，所有的参数名称都将丢失，IDE可能会使用诸如arg0、arg1之类的占位符代替原有的参数名称，对代码运行无影响，会给代码的编写带来不便；
- 2，调试时调试器无法根据参数名称从运行上下文中获取参数值

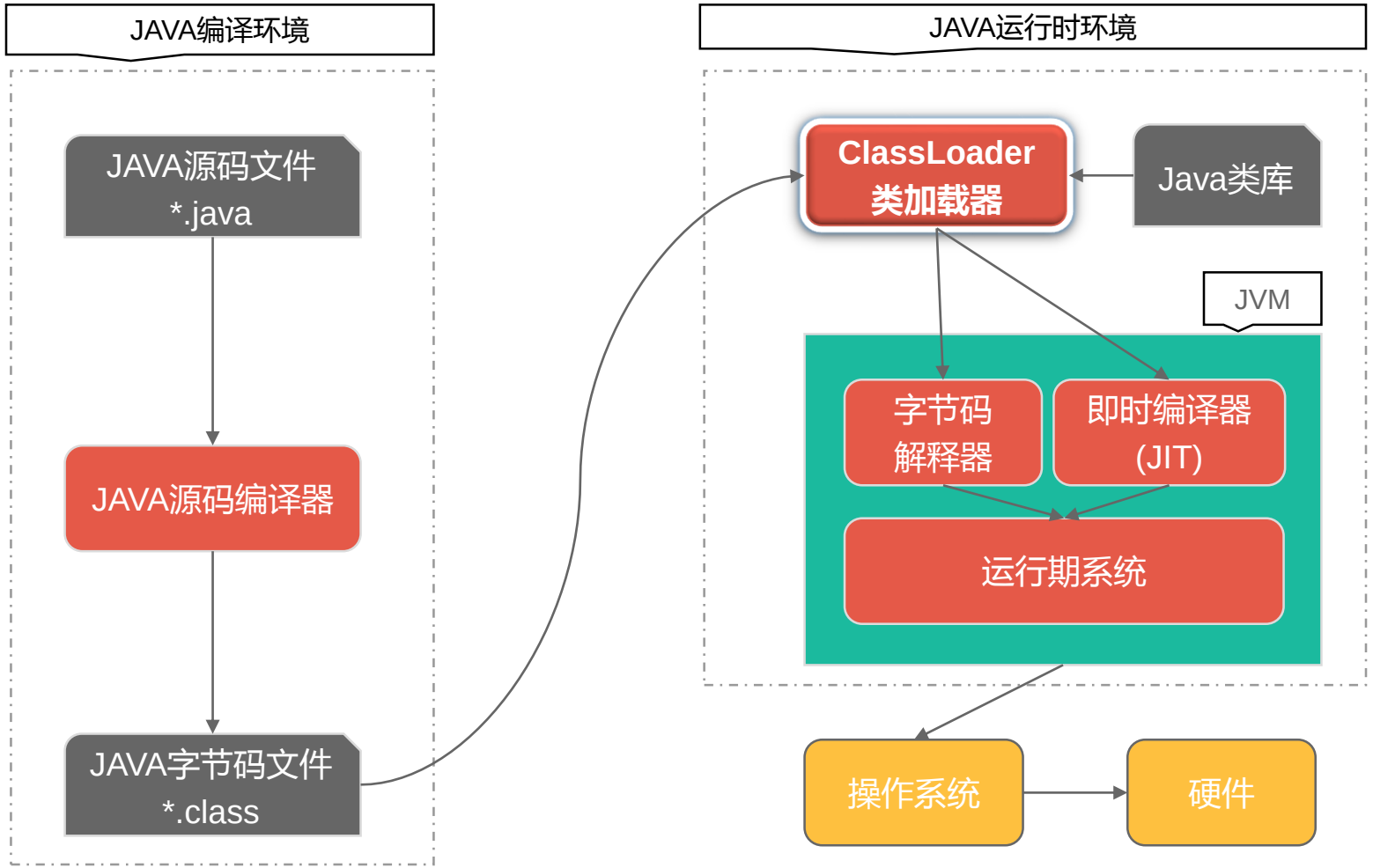
- **不生成LocalVariableTable属性demo**
- javac -g:none AttributeDemo0.java
- javap -v -private AttributeDemo0.class

没有本地变量表

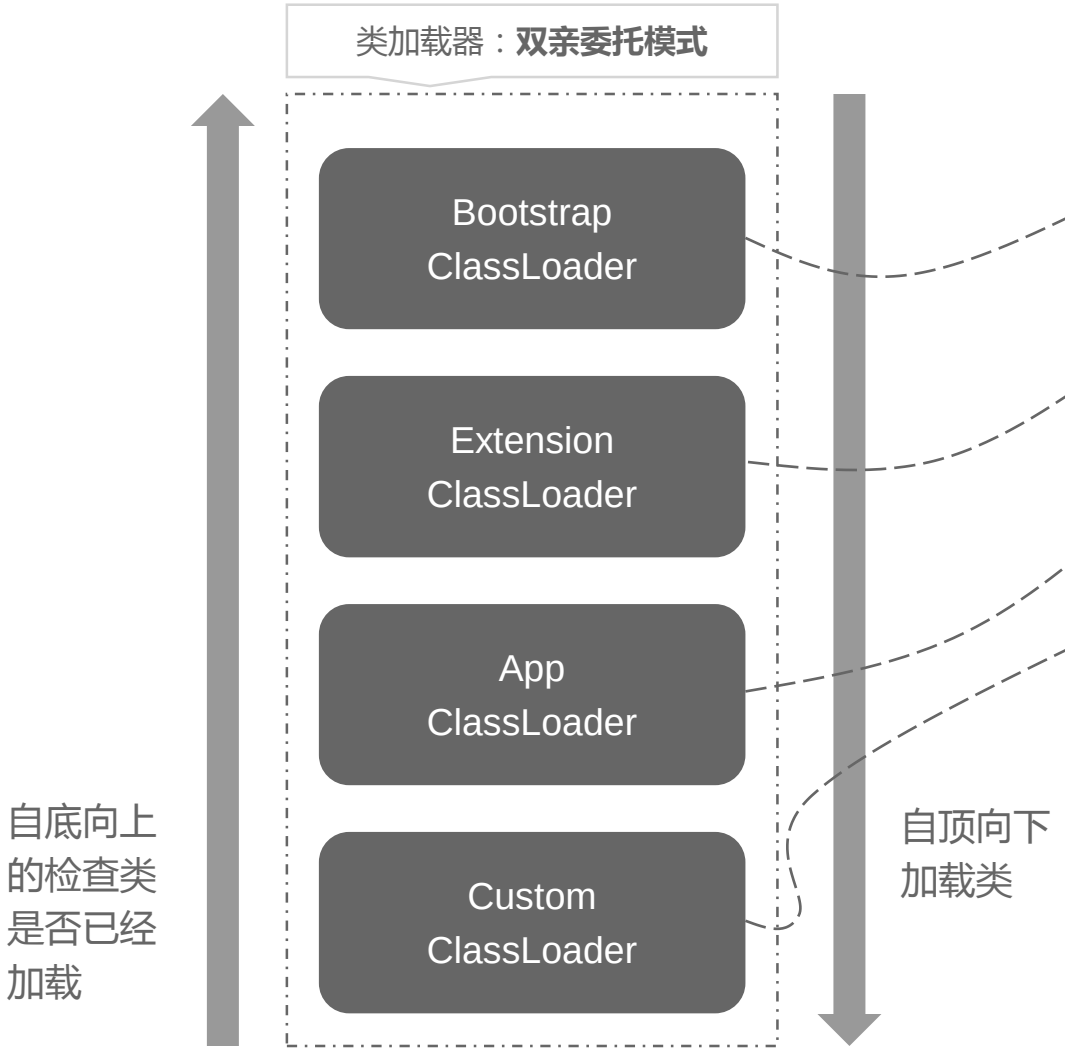
```
public void setAge(int);
  descriptor: (I)V
  flags: ACC_PUBLIC
  Code:
    stack=2, locals=2, args_size=2
       0: aload_0
       1: iload_1
       2: putfield      #5                  // Field age:I
       5: return

public static java.lang.String getName();
  descriptor: ()Ljava/lang/String;
  flags: ACC_PUBLIC, ACC_STATIC
  Code:
    stack=1, locals=0, args_size=0
       0: getstatic     #9                  // Field name:Ljava/lang/String;
       3: areturn
```

类加载器



类加载器



- 负责加载$JAVA_HOME中jre/lib/rt.jar里所有的class，由C++实现，不是ClassLoader子类
- 负责加载java平台中扩展功能的一些jar包，包括$JAVA_HOME中jre/lib/*.jar或-Djava.ext.dirs指定目录下的jar包
- 负责加载classpath中指定的jar包及目录中class
- 应用程序根据自身需要自定义的ClassLoader，如tomcat、jboss都会根据j2ee规范自行实现ClassLoader

- jvm启动的时候，并不会一次性加载所有的class文件，而是根据需要去动态加载

JVM启动类

JVM启动类源码：sun.misc.Launcher

- Jdk提供的JAVA_HOME\src.zip源码是不完整的，缺少OpenJDK的部分源码，所以无法查看
- 怎么样下载OpenJDK的源码, 过程很复杂~~
  - http://www.mimaxueyuan.com/2019/03/07/howto-download-openjdk/
  - jdk-687fd7c7986d.zip\jdk-687fd7c7986d\src\share\classes\sun\misc\Launcher.java

Launcher启动器源码

BootstrapClassLoader加载的类库路径

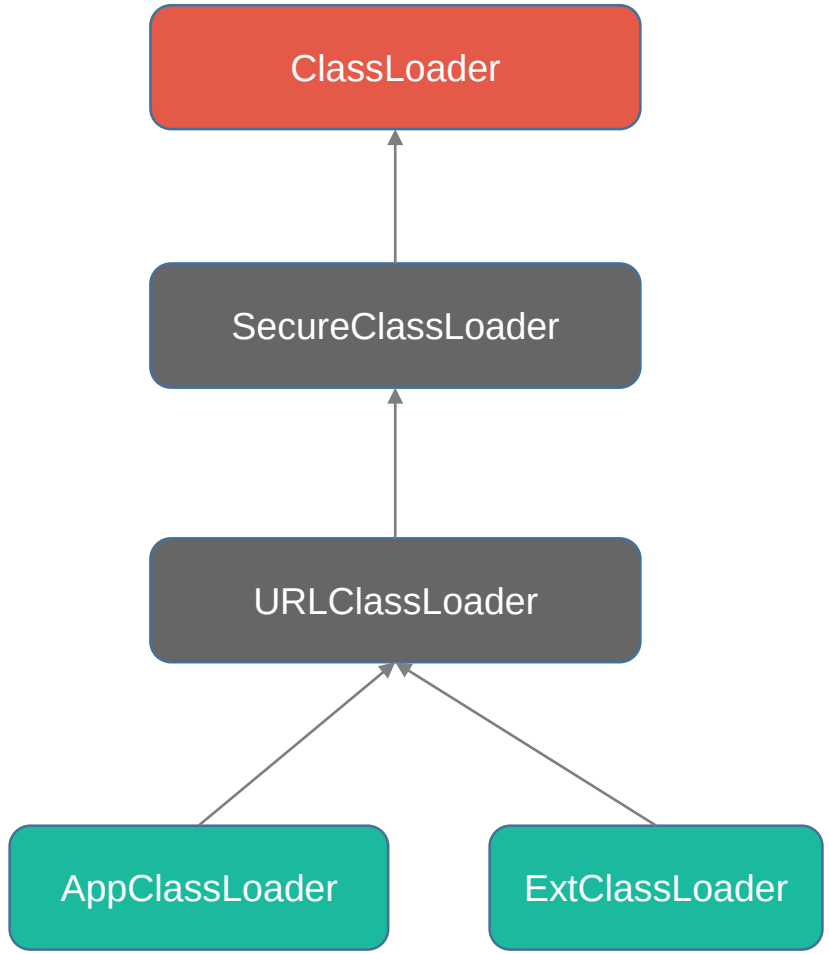
```
16 public class Launcher {
17     private static URLStreamHandlerFactory factory = new Factory((1)null);
18     private static Launcher launcher = new Launcher();
19     private static String bootClassPath = System.getProperty("sun.boot.class.path");
20     private ClassLoader loader;
21     private static URLStreamHandler fileHandler;
```

sun.misc.Launcher 虚拟机入口应用中加载了ExtClassLoader、AppClassLoader

```
27     public Launcher() {
28         ExtClassLoader var1;
29         try {
30             var1 = ExtClassLoader.getExtClassLoader();
31         } catch (IOException var10) {
32             throw new InternalError("Could not create extension class loader", var10);
33         }
34 
35         try {
36             this.loader = AppClassLoader.getAppClassLoader(var1);
37         } catch (IOException var9) {
38             throw new InternalError("Could not create application class loader", var9);
39         }
```

类加载器的关系

- EventInfoClassLoader - oracle.jrockit.jfr.events.EventHandlerCreator
- MethodUtil - sun.reflect.misc
- NashornLoader - jdk.nashorn.internal.runtime
  - ScriptLoader - jdk.nashorn.internal.runtime
  - StructureLoader - jdk.nashorn.internal.runtime
- URLClassLoader - java.net
  - AppClassLoader - sun.misc.Launcher
  - AppletClassLoader - sun.applet
  - ChildClassLoader - org.springframework.expression.spel.standard.SpelComp
  - ChildClassLoader - org.springframework.expression.spel.standard.SpelComp
  - DeploymentClassLoader - org.apache.axis2.deployment
  - ExtClassLoader - sun.misc.Launcher
  - ExtendedURLClassLoader - jodd.util.cl
  - ExtensibleURLClassLoader - org.aspectj.weaver.bcel
    - WeavingURLClassLoader - org.aspectj.weaver.loadtime
  - ExtensibleURLClassLoader - org.aspectj.weaver.bcel
    - WeavingURLClassLoader - org.aspectj.weaver.loadtime
  - FactoryURLClassLoader - java.net
  - Loader - sun.rmi.server.LoaderHandler
  - MLet - javax.management.loading
    - PrivateMLet - javax.management.loading
  - MultiParentClassLoader - org.apache.axis2.classloader
    - JarFileClassLoader - org.apache.axis2.classloader
  - StandardClassLoader - org.apache.catalina.loader
  - UtilClassLoader - org.aspectj.util
  - UtilClassLoader - org.aspectj.util
  - WebappClassLoaderBase - org.apache.catalina.loader
    - ParallelWebappClassLoader - org.apache.catalina.loader
    - WebappClassLoader - org.apache.catalina.loader
      - TomcatEmbeddedWebappClassLoader - org.springframework.boot



三种ClassLoader的加载范围查看

sun.boot.class.path

- BootstrapClassLoader加载的类路径

java.ext.dirs

- ExtensionClassLoader加载的类路径

java.class.path

- AppClassLoader加载的类路径

DEMO演示

- com.mimaxueyuan.jvm.classloader.AppClassLoaderDemo0
- com.mimaxueyuan.jvm.classloader.BootstrapClassLoaderDemo0
- com.mimaxueyuan.jvm.classloader.ExtensionClassLoaderDemo0

父加载器

每一个加载器都有一个父加载器

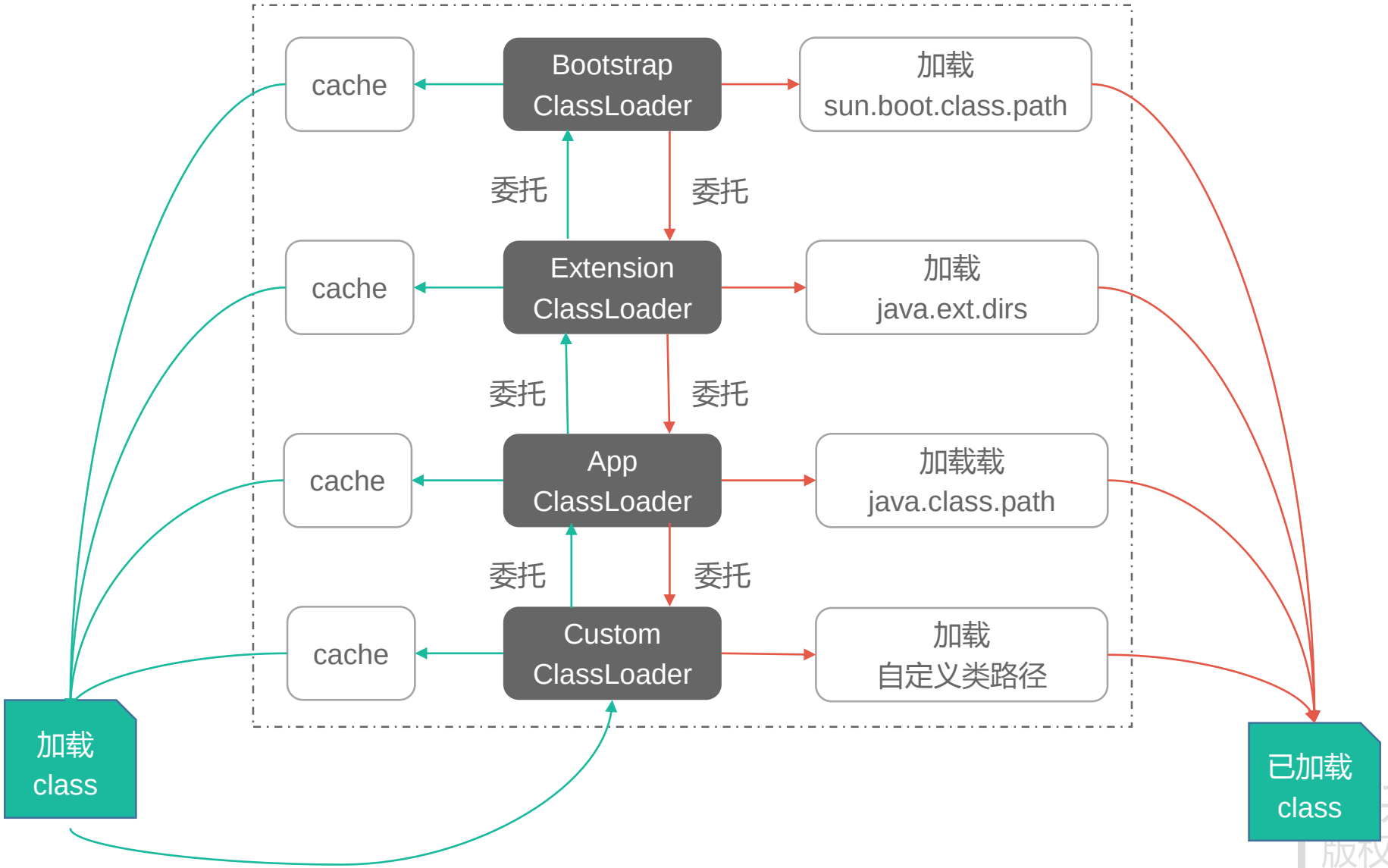
- 父加载器并不是父类
- ExtClassLoader的父加载器是BootstrapClassLoader，由C/C++编写的，它本身是虚拟机的一部分，所以它并不是一个JAVA类，也就是无法在java代码中获取它的引用，JVM启动时通过Bootstrap类加载器加载rt.jar等核心jar包中的class文件，之前的int.class,String.class都是由它加载。

- **扩展：** -verbose:class 输出所有的类加载日志

DEMO演示

- 父加载器演示：com.mimaxueyuan.jvm.classloader.ClassLoaderDemo0

类加载器-双亲委托模型



类加载器-双亲委托模型

类加载器重要源码

- java.lang.ClassLoader.loadClass(String) //加载类
- java.lang.ClassLoader.loadClass(String, boolean) //加载类
- java.lang.ClassLoader.findClass(String) //查找类
- java.lang.ClassLoader.findLoadedClass(String) //查找被加载过的类
- java.lang.ClassLoader.defineClass(String, byte[], int, int) //字节码二进制流转Class对象

类加载器-双亲委托模型&自定义类加载器

双亲委托的作用

- 每个加载器都只加载自己负责范围内的路径下的class文件，所有的类必须先递归到父加载器进行加载，避免了同一个class被重复加载，保证类一个类的字节码只被加载一次。
  - *例如在项目中编写一个java.lang.String类*

- 保证了安全性，防止核心类被篡改
  - *例如自己编写一个类加载器，加载D:\lib下的类，来覆盖系统类*
  - *Demo: com.mimaxueyuan.jvm.classloader.KevinClassLoader0*

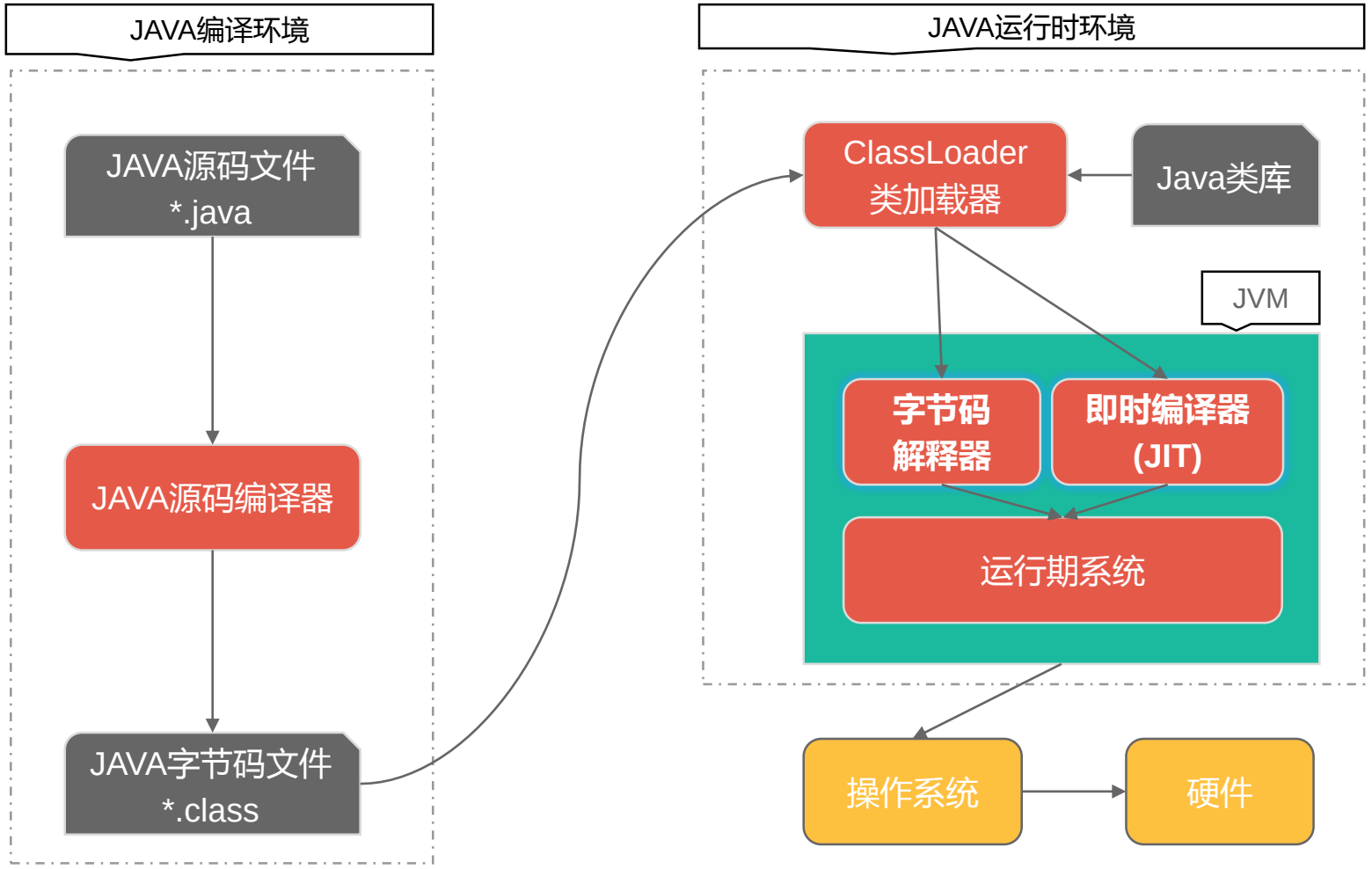
Class文件加解密-防反编译、防侵权

实现方案

- 将class文件加密为classx文件（加密和解密方式可以自由选择）
- 使用自定义ClassLoader先解密再加载

- Demo：com.mimaxueyuan.jvm.classloader.KevinClassLoader1

JAVA代码是如何执行的



JVM执行引擎

字节码解释器、即时编译器、混合模式

字节码解释器

- 一条一条地读取，解释并且执行字节码指令。因为它一条一条地解释和执行指令，所以它可以很快地解释字节码，但是执行起来会比较慢，没有JIT的配合下效率不高。

即时编译器(JIT)

- 即时编译器把整段字节码不加筛选的编译成机器码不论其执行频率是否有编译价值，在程序响应时间的限制下，没有达到最大的优化。

混合模式

- 在解释执行的模式下引入编译执行，刚好可以弥补相互的缺点，达到更优的效果。
- 程序刚开始启动的时候，因为解释器可以很快的解释字节码，所以首先发挥作用，解释执行Class字节码，在合适的时候，即时编译器把整段字节码编译成本地代码，然后，执行引擎就没有必要再去解释执行方法了，它可以直接通过本地代码去执行它。执行本地代码比一条一条进行解释执行的速度快很多，主要原因是本地代码是保存在缓存里的。

热点代码 Hot Spot Code

热点代码技术原理

- 在刚开始的时候，JVM不知道某段代码是否会频繁的调用方法，因此刚开始的时候并不会编译代码，而是用解释器来执行。
- 编译器自动去判断哪些方法会被频繁调用，如果一个方法的执行频率超过一个特定的值的话，那么这个方法就会被JIT编译成本地代码。这些运行频繁的方法或代码块，就会被标记为“热点代码”（Hot Spot Code）。
- 当 JVM 执行某一方法或遍历循环的次数越多，就会更加了解代码结构，那么 JVM 在编译代码的时候就做出相应的优化。

热点代码分类

- 多次被调用的方法
- 多次被执行的循环体

热点代码检测

- 方法计数器：记录方法调用的次数
- 回边计数器：记录代码块循环次数
- 当计数器数值大于默认阈值或指定阈值时，方法或代码块会被编译成本地代码。

热点代码检测方法

方法计数器

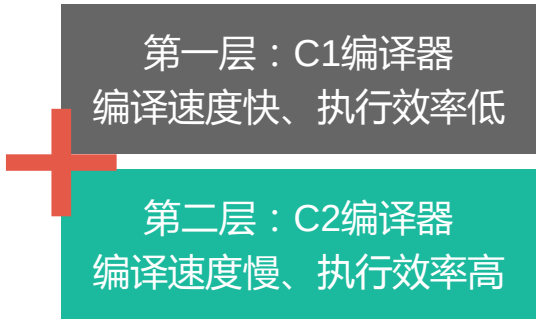
- 采用这种方法的虚拟机会为每个方法建立计数器，统计方法的执行次数，如果执行次数超过一定的阈值就认为它是热点方法。方法调用计数器统计的并不是方法被调用的绝对次数，而是一个相对的执行频率，即一段时间内方法被调用的次数，当超过一定的时间限度，如果方法的调用次数仍然不足以让它提交给即时编译器编译，那这个方法的调用计数器就会被减少一半，这个过程称为方法调用计数器的热度的衰减，而这段时间就成为此方法统计的半衰周期，进行热度衰减的动作在虚拟机进行垃圾收集时顺便进行了

回边计数器

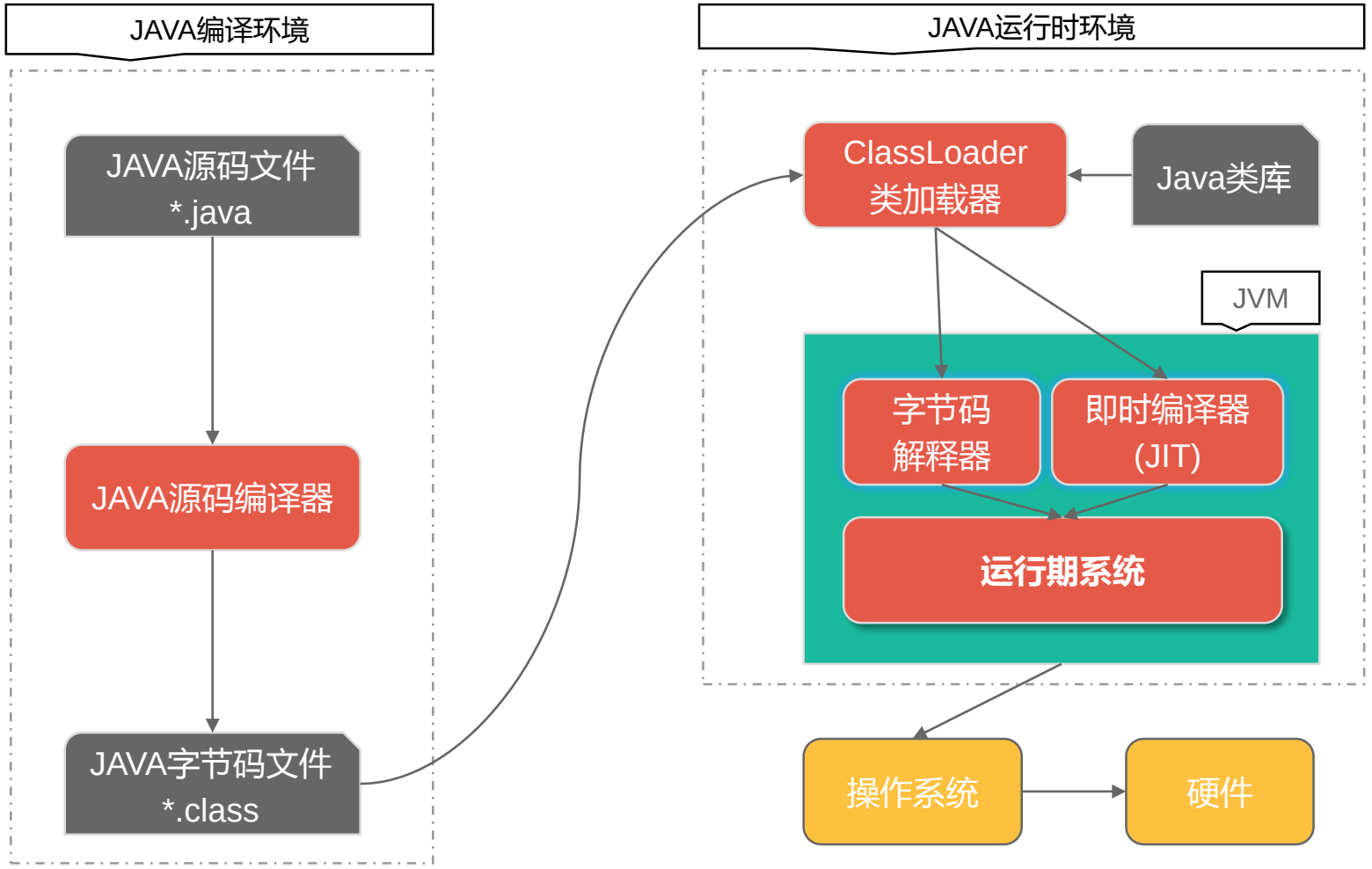
- 它的作用是统计一个方法体重循环体代码执行的次数，在字节码中遇到控制流向后跳转的指令称为回边，显然，建立回边计数器统计的目的就是为了触发OSR编译。没有计数热度衰减的过程，因此这个计数器统计的就是该方法执行循环的绝对次数，当计数器溢出的时候，它还会把方法计数器的值也调整到溢出的状态，这样下次在再进入该方法的时候就会执行标准编译过程。

JVM的分层编译

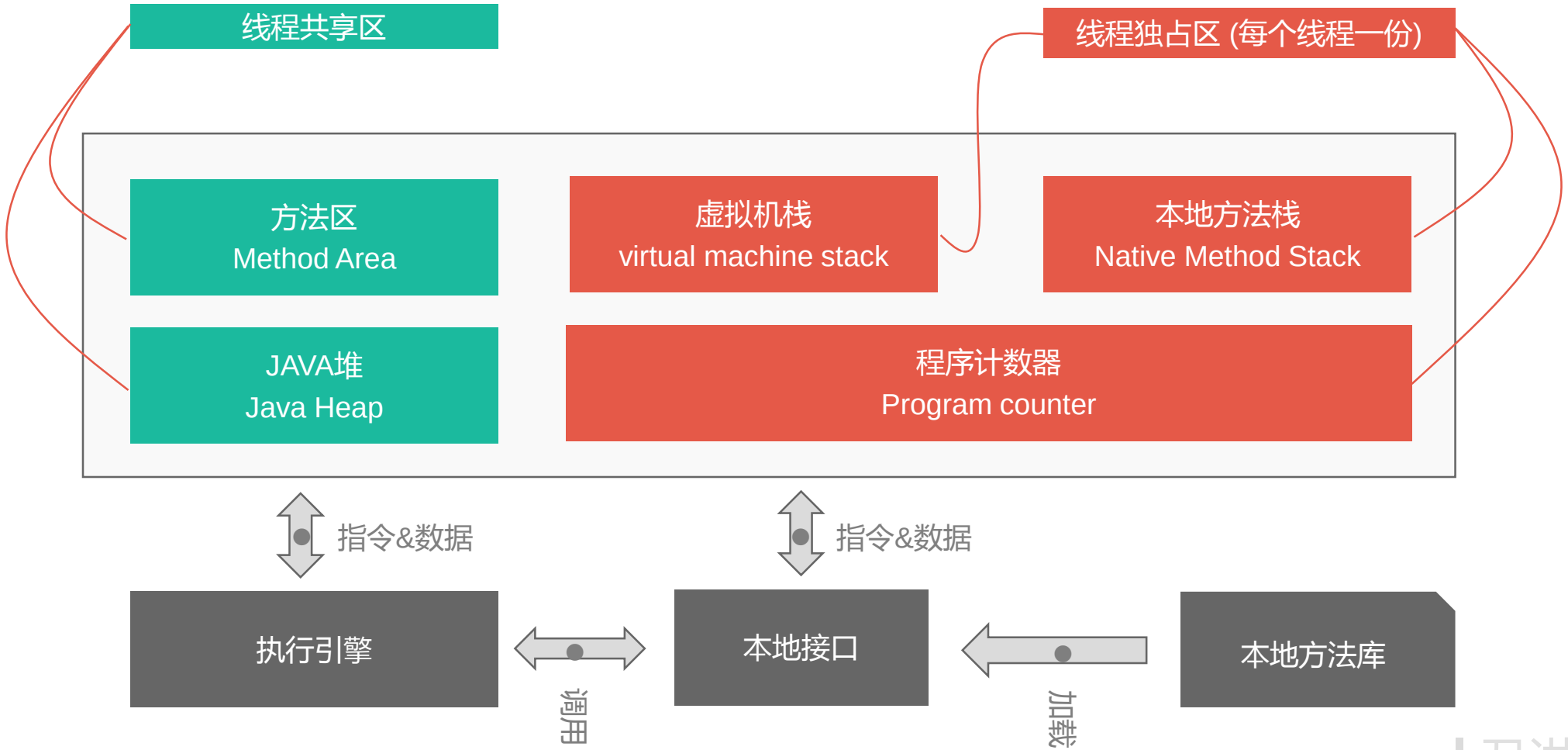
- HotSpot 内置两种编译器，分别是client启动时的c1编译器和server启动时的c2编译器
- c2在将代码编译成机器代码的时候需要搜集大量的统计信息以便在编译的时候进行优化，因此编译出来的代码执行效率比较高，代价是程序启动时间比较长，而且需要执行比较长的时间，才能达到最高性能；
- 与之相反， c1的目标是使程序尽快进入编译执行的阶段，所以在编译前需要搜集的信息比c2要少，编译速度因此提高很多，但是付出的代价是编译之后的代码执行效率比较低，但尽管如此，c1编译出来的代码在性能上比解释执行的性能已经有很大的提升，所以所谓的分层编译，就是一种折中方式，在系统执行初期，执行频率比较高的代码先被c1编译器编译，以便尽快进入编译执行，然后随着时间的推移，执行频率较高的代码再被c2编译器编译，以达到最高的性能



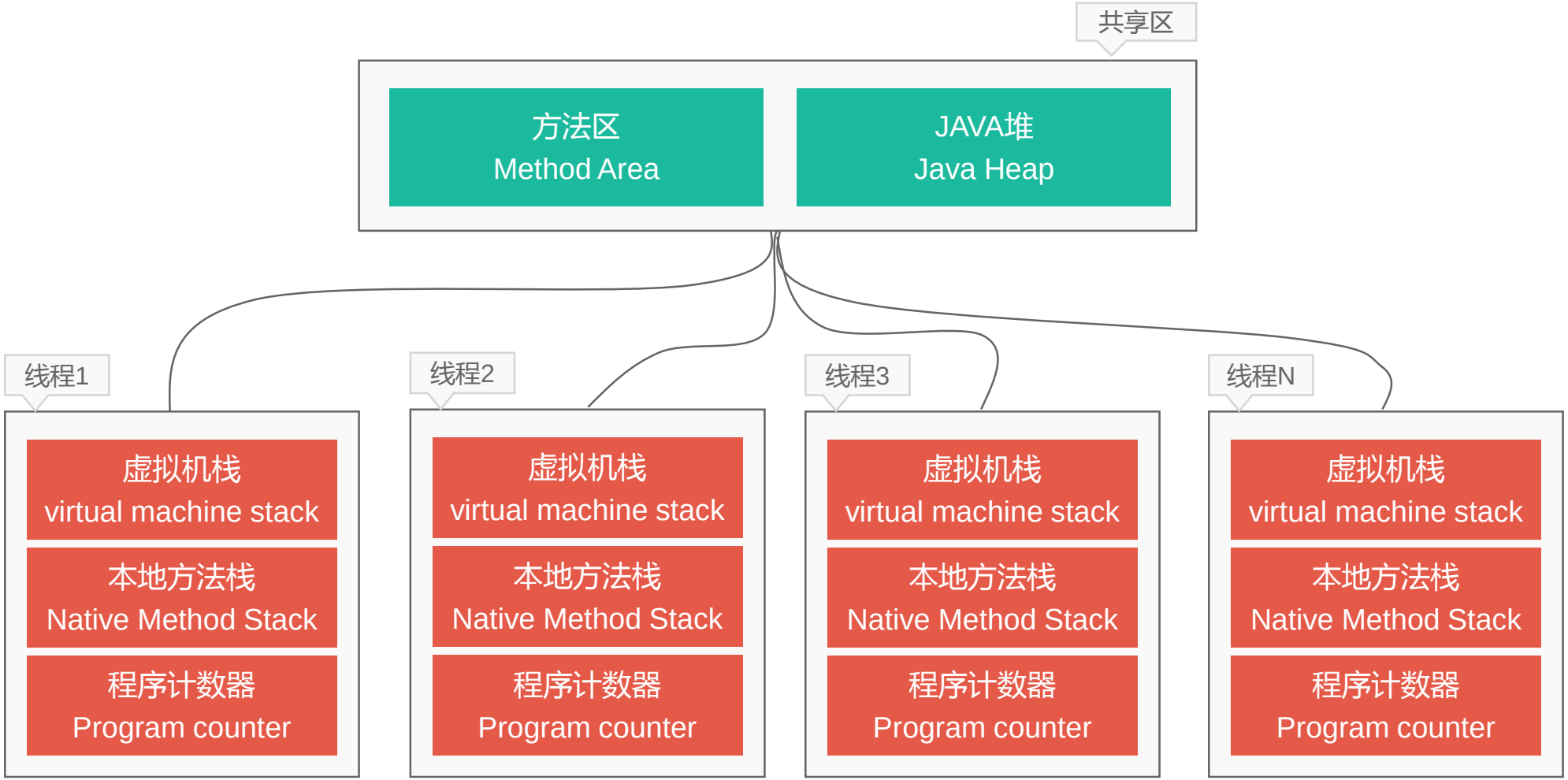
JAVA代码是如何执行的



JVM内存模型



线程共享区与独享区

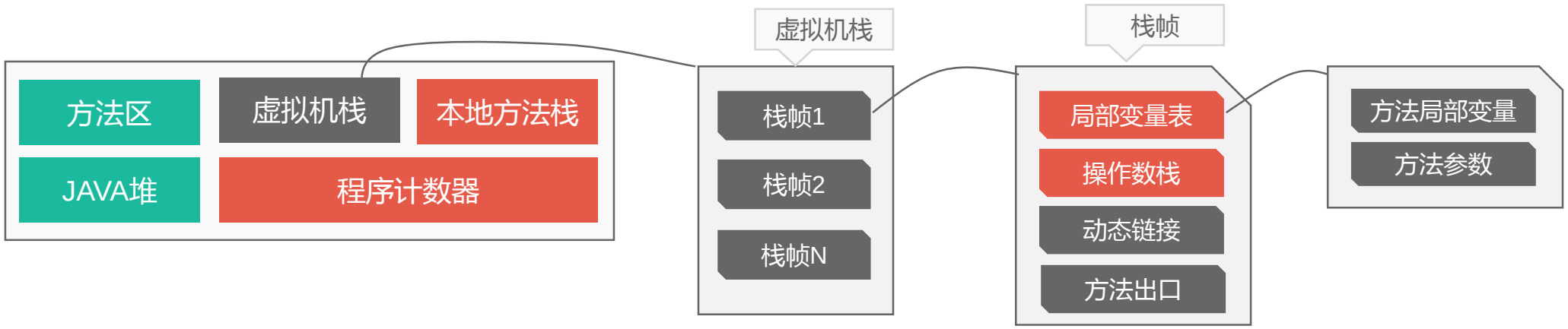


程序计数器 Program counter

说明

- 程序计数器所占用的内存空间很小
- 每个线程都有一个程序计数器
- 内部存放的是一个指示器，用来指定当前线程要执行的字节码指令的地址
- 如果当前执行的是native方法,则程序计数器的值为undefined
- 此区域是唯一一个JVM虚拟机规范当中没有规定任何OOM(OutOfMemeoryError)的区域

虚拟机栈 virtual machine stack



虚拟机栈的结构

- 内部存放的是栈帧(Stack Frame)
- 每个方法执行的时候都会创建一个栈帧，用于存储局部变量表，操作数栈，动态链接，方法出口等信息。
- 每一个方法的调用过程就是一个栈帧再虚拟机栈中入栈到出栈的过程

栈帧的结构

- 编译期可以预知的各种基本数据类型(boolean、byte、chare、short、int、float、long、double）；其中64位长度的long、double类型的数据会占用2个局部变量空间（Slot),其余的数据类型只占用1个。
- 对象引用(reference类型，可能是一个对象起始地址的引用指针、也可能指向一个代表对象的句柄或与此相关的位置）
- returnAddress类型（指向了一条字节码指令的地址）

虚拟机栈 virtual machine stack

- 当有一个方法被调用时，代表这个方法的栈帧入栈；当这个方法返回时，其栈帧出栈。因此，虚拟机栈中栈帧的入栈顺序就是方法调用顺序
- 局部变量表中的变量不可直接使用，如需使用必须通过相关指令将其加载至操作数栈中作为操作数使用

```
int a = 1;
int b = 2;
int c = a + b;
return c;
```

字节码

```
public int operandStackTest1();
  descriptor: ()I
  flags: ACC_PUBLIC
  Code:
    stack=2, locals=4, args_size=1
       0: iconst_1
       1: istore_1
       2: iconst_2
       3: istore_2
       4: iload_1
       5: iload_2
       6: iadd
       7: istore_3
       8: iload_3
       9: ireturn
    LineNumberTable:
      line 30: 0
      line 31: 2
      line 32: 4
      line 33: 8
    LocalVariableTable:
      Start  Length  Slot  Name   Signature
          0      10     0  this   Lcom/mimaxueyuan/jvm/vmstack/OperandStackDemo0;
          2       8     1     a   I
          4       6     2     b   I
          8       2     3     c   I
```

- Demo：/mima-jvm/src/main/java/com/mimaxueyuan/jvm/vmstack/OperandStackDemo0.java

虚拟机栈 virtual machine stack

- Demo： javap com.mimaxueyuan.jvm.vmstack.OperandStackDemo0.class
- 局部变量表中的变量不可直接使用，如需使用必须通过相关指令将其加载至操作数栈中作为操作数使用

字节码指令	说明
0: iconst_1	int类型1压栈
1: istore_1	栈顶出栈，把int=1存入局部变量表中索引为1的slot中,也就是变量a的存储空间
2: iconst_2	int类型2压栈
3: istore_2	栈顶出栈，把int=2存入局部变量表中索引为2的slot中,也就是变量b的存储空间
4: iload_1	把局部变量表中索引为1的sloat中存储的数据压栈，也就是变量a的值
5: iload_2	把局部变量表中索引为2的sloat中存储的数据压栈，也就是变量b的值
6: iadd	将栈顶的两个元出栈后求和，再压栈
7: istore_3	将栈顶元素（求和之后的值）放入局部变量表中索引为3的sloat中，也就是变量c
8: iload_3	把局部变量表中索引为3的sloat中存储的数据压栈，也就是变量c的值
9: ireturn	栈顶元素出栈，返回结果

字节码指令查询方法

- Java语言与虚拟机规范(官方文档)：https://docs.oracle.com/javase/specs/index.html

Java Language and Virtual Machine Specifications

Java SE 12

Released March 2019 as JSR 386

The Java Language Specification, Java SE 12 Edition
- HTML | PDF

The Java Virtual Machine Specification, Java SE 12 Edition
- HTML | PDF

Java SE 11

Released September 2018 as JSR 384

The Java Language Specification, Java SE 11 Edition
- HTML | PDF

The Java Virtual Machine Specification, Java SE 11 Edition
- HTML | PDF

虚拟机栈 virtual machine stack

参数说明

-Xss 设置栈的大小，栈的大小直接决定函数调用的可达深度

-Xsssize
设置线程堆栈大小（以字节为单位）。附加字母k或K表示KB，m或M表示MB，g或G表示GB。默认值取决于虚拟内存。
以下示例以不同的单位将线程堆栈大小设置为1024 KB：
-Xss1m
-Xss1024k
-Xss1048576
此选项相当于-XX:ThreadStackSize。

异常说明

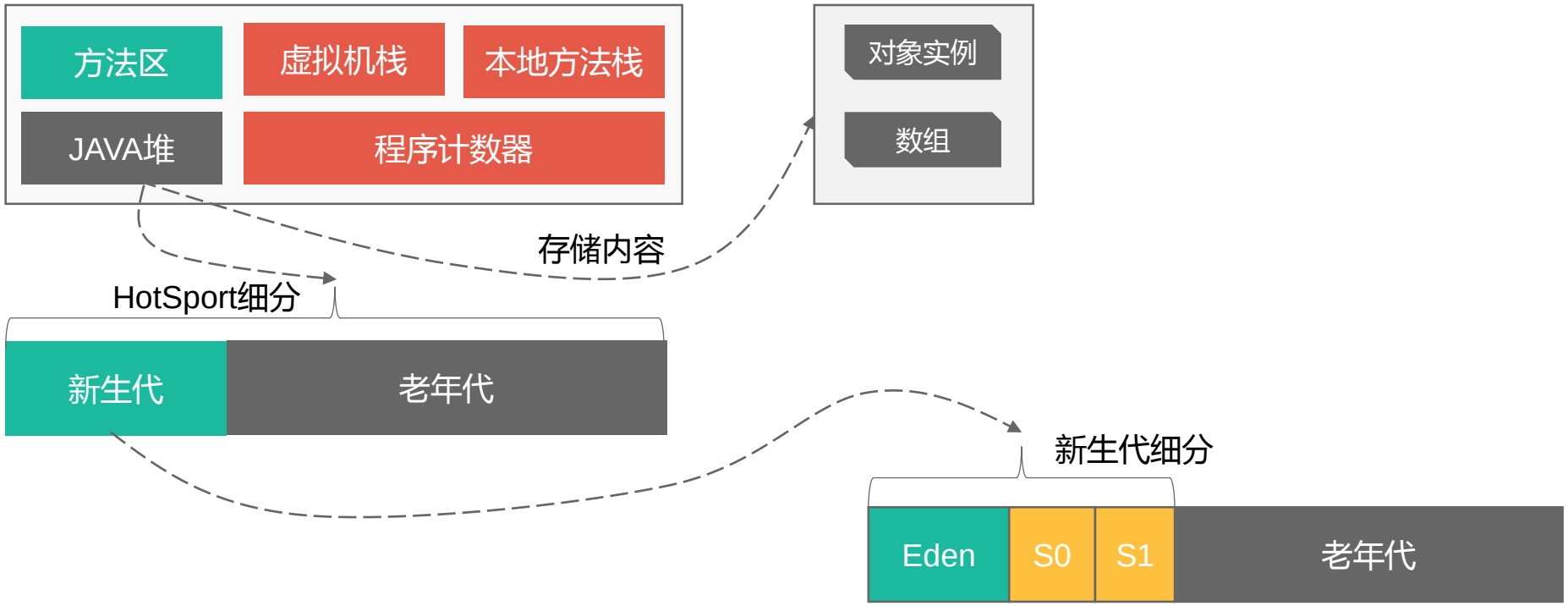
JAVA虚拟机规范,对于虚拟机栈规定了两种异常；
1、如果线程请求的栈的深度大于虚拟机所允许的最大值，则跑出StackOverflowException（虚拟机栈的深度允许动态扩展、也允许固定）
2、当虚拟机栈扩展的时候无法申请到足够的内存则跑出OutOfMemoryException

本地方法栈 Native Method Stack

说明

- 内部存放的是栈帧(Stack Frame)
- 每个Native方法执行的时候都会创建一个栈帧
- 虚拟机规范没有强制此部分实现使用的语言和数据结构，可以自由实现
- 虚拟机规范没有规定大小，可以固定也可以动态计算，如果固定则在每个线程创建的时候都可以单独设置，如果动态计算则要提供参数设置最大最小值。
- Sun HotSpot虚拟机直接把虚拟机栈和本地方法栈合二为一了

JAVA堆 Java Heap



- JVM管理的最大的一块内存区域，不要求物理连续，可以固定大小，也可以动态扩展（当前主流虚拟机实现）
- 存储绝大多数的对象实例和数组
- 是GC的主要区域，因此也叫做GC堆
- 在HotSpotVM中将堆细分为新生代、老年代， 新生代又分为Eden、S0、S1区，他们都和GC相关
- S0又叫From Survivor、S1又叫To Survivor
- 无论怎么划分内部存储的内容不变，都是对象实例和数组
- 当堆内存无法扩展时抛出OOM异常

Java堆 Java Heap参数

-Xmssize

设置堆的初始大小（以字节为单位）。该值必须是1024的倍数且大于1 MB。附加字母k或K表示千字节，m或M指示兆字节，g或G指示千兆字节。以下示例显示如何使用各种单位将分配的内存大小设置为6 MB：

-Xms6291456

-Xms6144k

-Xms6m

如果未设置此选项，则初始大小将设置为老年代和年轻代分配的大小的总和。可以使用-Xmn选项或-XX:NewSize选项设置年轻代的堆的初始大小。

-Xmxsize

指定内存分配池的最大大小（以字节为单位），以字节为单位。该值必须是1024的倍数且大于2 MB。附加字母k或K表示千字节，m或M指示兆字节，g或G指示千兆字节。根据系统配置在运行时选择默认值。对于服务器部署，-Xms并-Xmx经常设置为相同的值。以下示例显示如何使用各种单位将分配的内存的最大允许大小设置为80 MB：

-Xmx83886080

-Xmx81920k

-Xmx80m

该-Xmx选项相当于-XX:MaxHeapSize。

- 以上只是一部分参数，对于Eden、S0、S1区的参数设置，后续GC部分讲解

Java堆 Java Heap参数

-XX:+HeapDumpOnOutOfMemoryError

在java.lang.OutOfMemoryError抛出异常时，通过使用堆分析器（HPROF）将Java堆转储到当前目录中的文件。您可以使用该-XX:HeapDumpPath选项显式设置堆转储文件路径和名称。默认情况下，禁用此选项，并在OutOfMemoryError抛出异常时不转储堆。

-XX:HeapDumpPath=path

设置-XX:+HeapDumpOnOutOfMemoryError选项设置时，设置用于写入堆分析器（HPROF）提供的堆转储的路径和文件名。默认情况下，该文件在当前工作目录中创建，并且名为java_pidpid.hprof，其中pid是导致错误的进程的标识符。以下示例显示如何显式设置默认文件（%p表示当前进程标识符）：

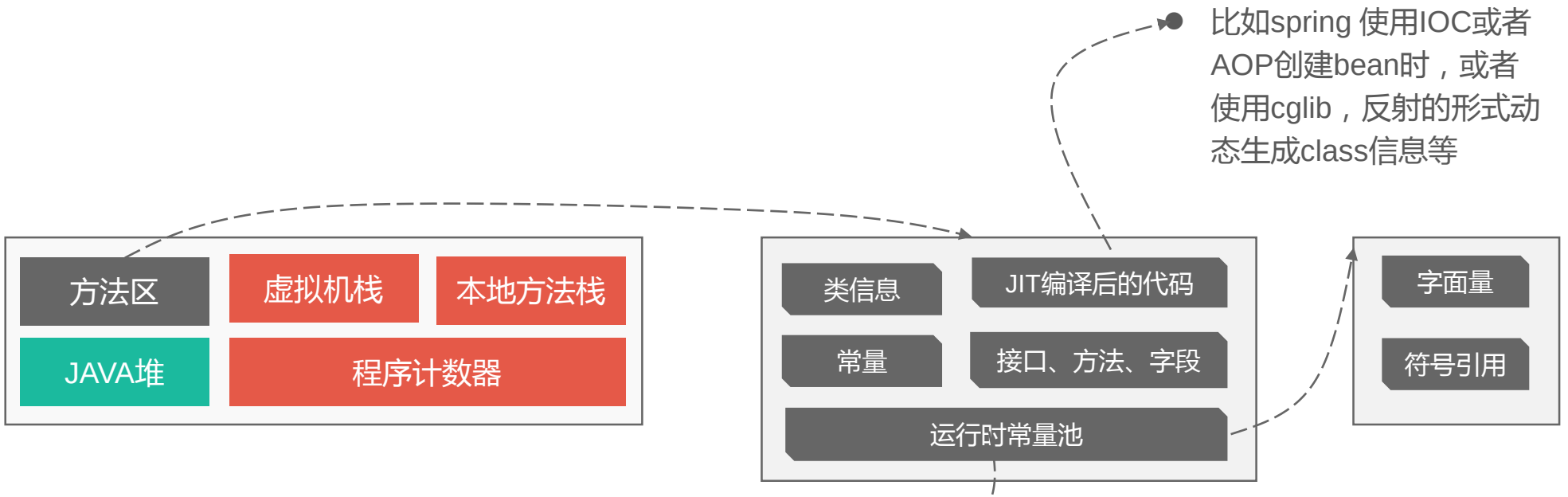
-XX：HeapDumpPath = / java_pid％p.hprof

以下示例显示如何将堆转储文件设置为C:/log/java/java_heapdump.log：

-XX：HeapDumpPath=C:/log/java/java_heapdump.log

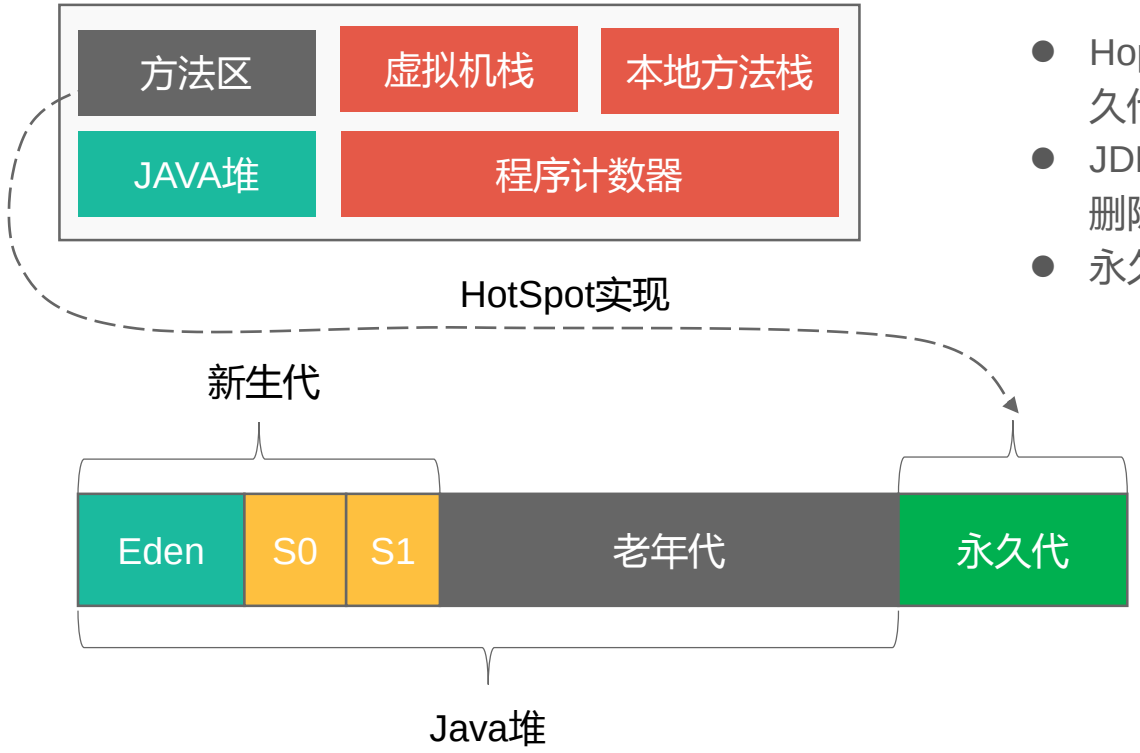
- 推荐设置以上两个参数，并且HeapDumpPath要指定一块有存储空间的区域，否则可能导致存储失败。
- **Demo:** com.mimaxueyuan.jvm.heap.HeapDemo0

方法区 Method Area



方法区 Method Area 与 永久代 Perm

- 方法区被所有线程共享，在虚拟机启动时创建，AVA堆的一个逻辑部分，但是为了和堆做区分，我们也叫它Non-Heap（非堆）
- JVM规范描述方法区可以在内存上连续也可以不连续,可以固定大小也可以动态扩展，也可以不进行垃圾回收。

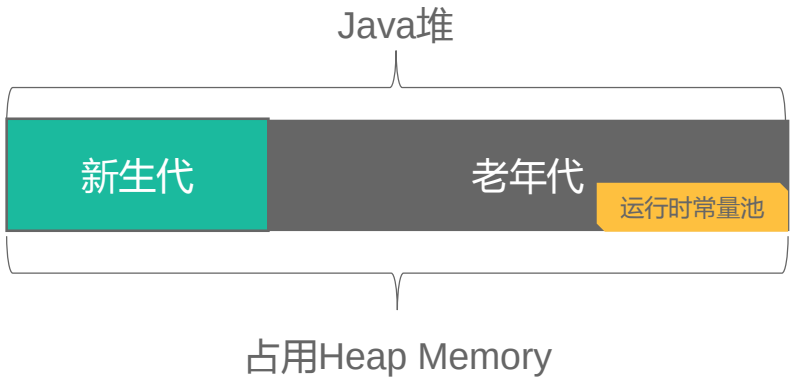


- HoptSpot为了让这个区域也纳入到GC的范围，所以采用了永久代的方式实现了方法区
- JDK8以前（不含）存在永久代（Perm），JDK8以后（含）删除了永久代（Perm）
- 永久代是HotSpot专有的，而在JRokit、J9虚拟机里没有

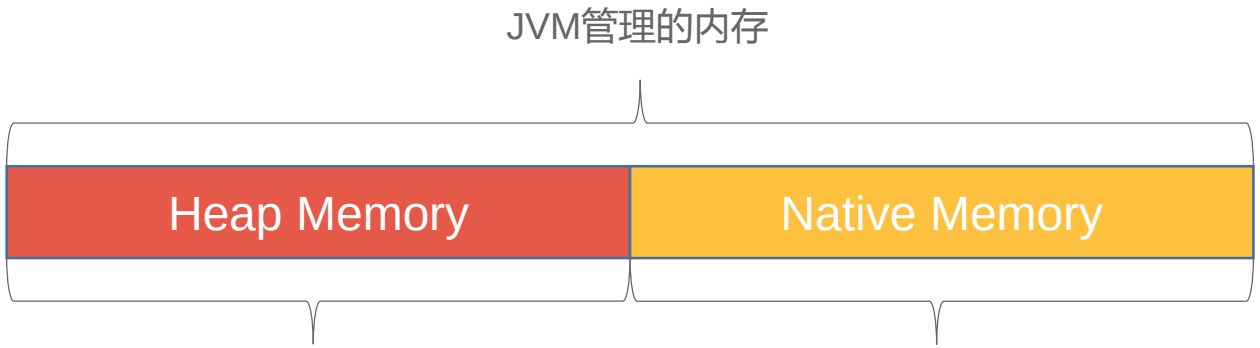
HotSpot 方法区变化

永久代与元数据区的区别

永久代难以预估大小,经常OOM,GC效率极低



Heap Memory 与 Native Memory



JAVA程序使用
使用-Xms、-Xmx限制
超出会引发OOM异常

JVM进程使用
无参数限制,受操作系统和物理内存限制
超出也会引发OOM异常：
1、Native memory allocation (mmap) failed to map XXX bytes for committing reserved memory
2、Allocated 1953546760 bytes of native memory before running out

Perm 参数 与 Metaspace参数

-XX:PermSize=size
设置分配给永久代的空间（以字节为单位），如果超出则会触发垃圾回收。此选项在JDK 8中已弃用，并被该-XX:MetaspaceSize选项取代。

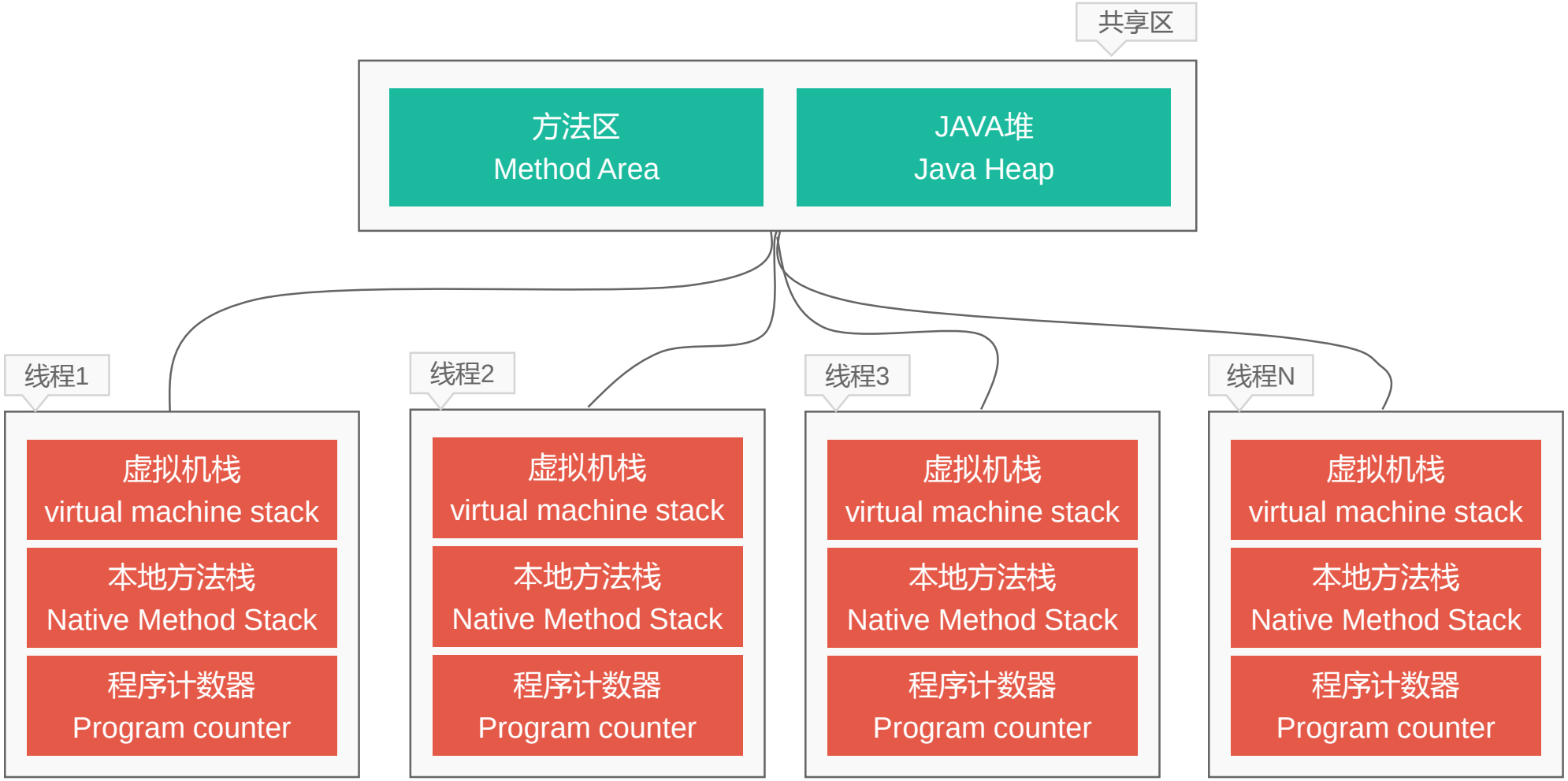
-XX:MaxPermSize=size
设置最大永久代空间大小（以字节为单位）。此选项在JDK 8中已弃用，并由该-XX:MaxMetaspaceSize选项取代。

-XX:MetaspaceSize=size
设置分配的类元数据空间的大小，该空间将在第一次超出时触发垃圾回收。根据使用的元数据量，增加或减少垃圾收集的阈值。默认大小取决于平台。

-XX:MaxMetaspaceSize=size
设置可以为类元数据分配的最大本机内存量。默认情况下，大小不受限制。应用程序的元数据量取决于应用程序本身，其他正在运行的应用程序以及系统上可用的内存量。
以下示例显示如何将最大类元数据大小设置为256 MB：
-XX：MaxMetaspaceSize =256m

- **JDK7 Demo: com.mimaxueyuan.jvm.methodarea.PermGenDemo0**
- **JDK8 Demo: com.mimaxueyuan.jvm.methodarea.MetadataSpaceDemo0**

内存模型与线程安全



对象创建(new操作)过程

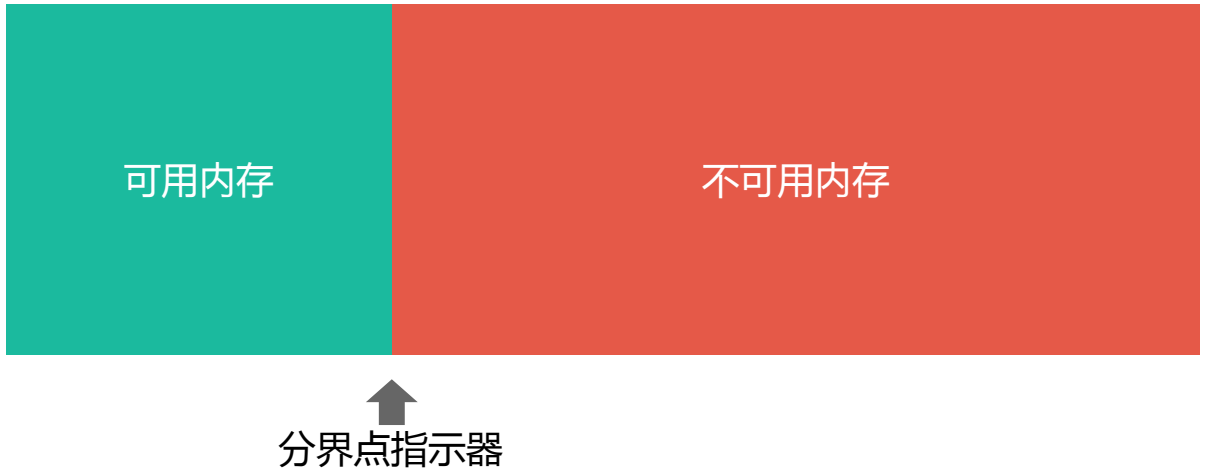
1	2	3		4	5
对象创建	方法区类加载	JAVA堆分配		初始化	完成
Obj a = new Obj()	是否能够在方法区的常量池中定位到一个类的符号引用	两种内存分配方法	空闲列表法	执行<init>方法 初始化对象的数据	
			指针碰撞法		
	类是否已经加载、解析、初始化	内存分配十分频繁,为了防止内存分配错乱需要解决并发问题，解决并发问题有两种方法	同步处理方式		
	如果没有加载需要先进行类加载流程		TLAB方式		
	类加载完毕后 则对象的大小已经确认	内存分配完毕之后进行内存初始化，保证对象的实例字段在java代码中不赋初始化的时候就可以访问			

虚拟机栈	方法区	堆	虚拟机栈	虚拟机栈

堆内存分为-指针碰撞法

指针碰撞法

带有压缩整理功能的垃圾收集器Serial、ParNew等带有Compact过程的收集器，内存规整，使用过的内存与未使用的内存分开，中间放着一个指针作为分界点指示器。
分配内存只是移动指针，向未使用的内存一侧移动一段与对象大小相等的空间即可。这种方式叫“指针碰撞（Bump the Pointer）

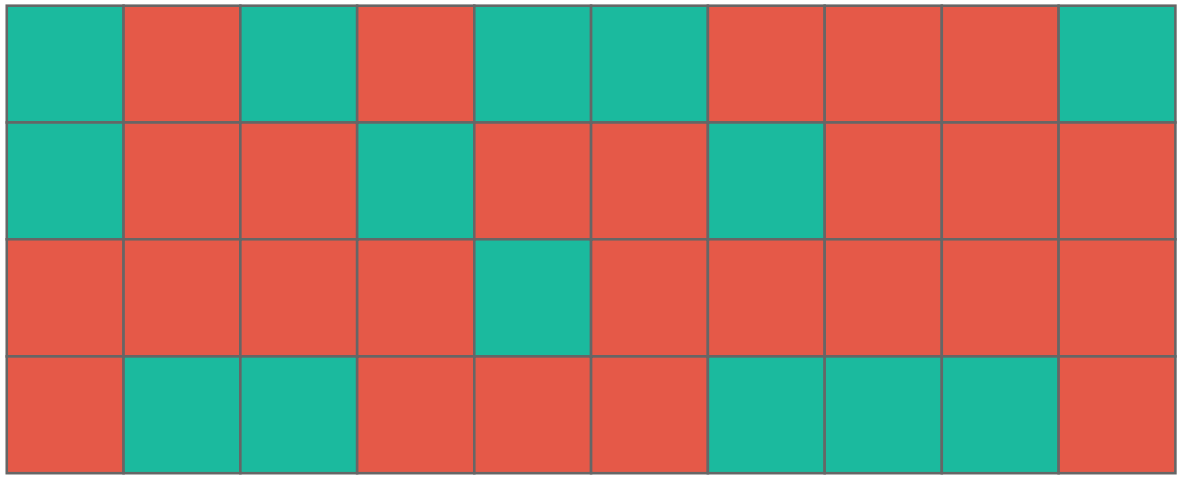


堆内存分为-空闲列表法

空闲列表法

CMS这种基于Mark-Sweep算法的收集器,内存都是不规整的,已使用内存与未使用内存交错存放，虚拟机维护一个记录表，用于记录那些内存是可用的，内存分配的时候需要找一块足够大的空间分配给对象，并更新记录表，这种方式叫”空闲列表“（Free List），通常使用这种方式

空闲列表



堆内存分配-并发处理

内存分配十分频繁,为了防止内存分配错乱需要解决并发问题，解决并发问题有两种方法

同步处理方式

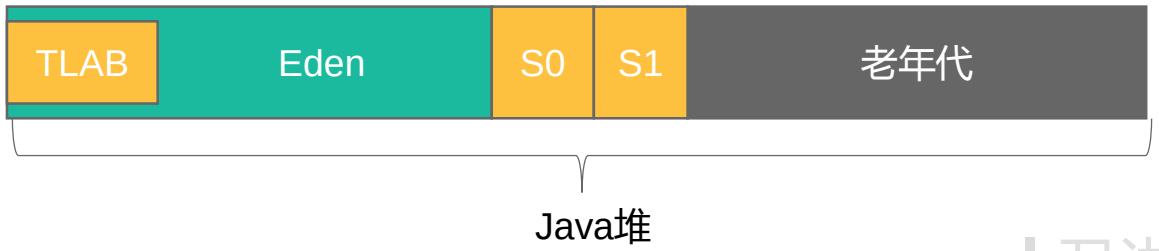
内存分配的动作采用同步处理,JVM为了增加效率采用的是CAS方式

TLAB方式

每个线程在JAVA堆中预先分配一小块内存，叫本地线程分配缓冲区，TLAB(Thread Local Allocation Buffer)

哪个线程要分配内存就在各自的TLAB中先分配

只有在线程的TLAB用完，才会在堆中分配，这时候需要分配新的TLAB的时候才需要同步控制



TLAB相关参数

-XX:+UseTLAB
允许在年轻代空间中使用线程局部分配块（TLAB）。默认情况下启用此选项。要禁用TLAB，请指定-XX:-UseTLAB。

-XX:TLABSize=size
设置线程局部分配缓冲区（TLAB）的初始大小（以字节为单位）。附加字母k或K表示千字节，m或M指示兆字节，g或G指示千兆字节。如果此选项设置为0，则JVM会自动选择初始大小。
以下示例显示如何将初始TLAB大小设置为512 KB：

```
-XXTLABSize=512K
```

-XX:TLABRefillWasteFraction
表示，设置进入TLAB空间，单个对象大小;是一个比例值，默认为64;如果，对象小于整个空间的1/64，则放在TLAB区。如果，对象大于整个空间的1/64，则放在堆区

-XX:+PrintTLAB
表示，查看TLAB信息

-XX:ResizeTLAB
表示，自动调整TLABRefillWasteFraction阈值

- **JDK8 com.mimaxueyuan.jvm.heap.TLABDemo0**

JVM参数-打印XX参数

打印XX参数

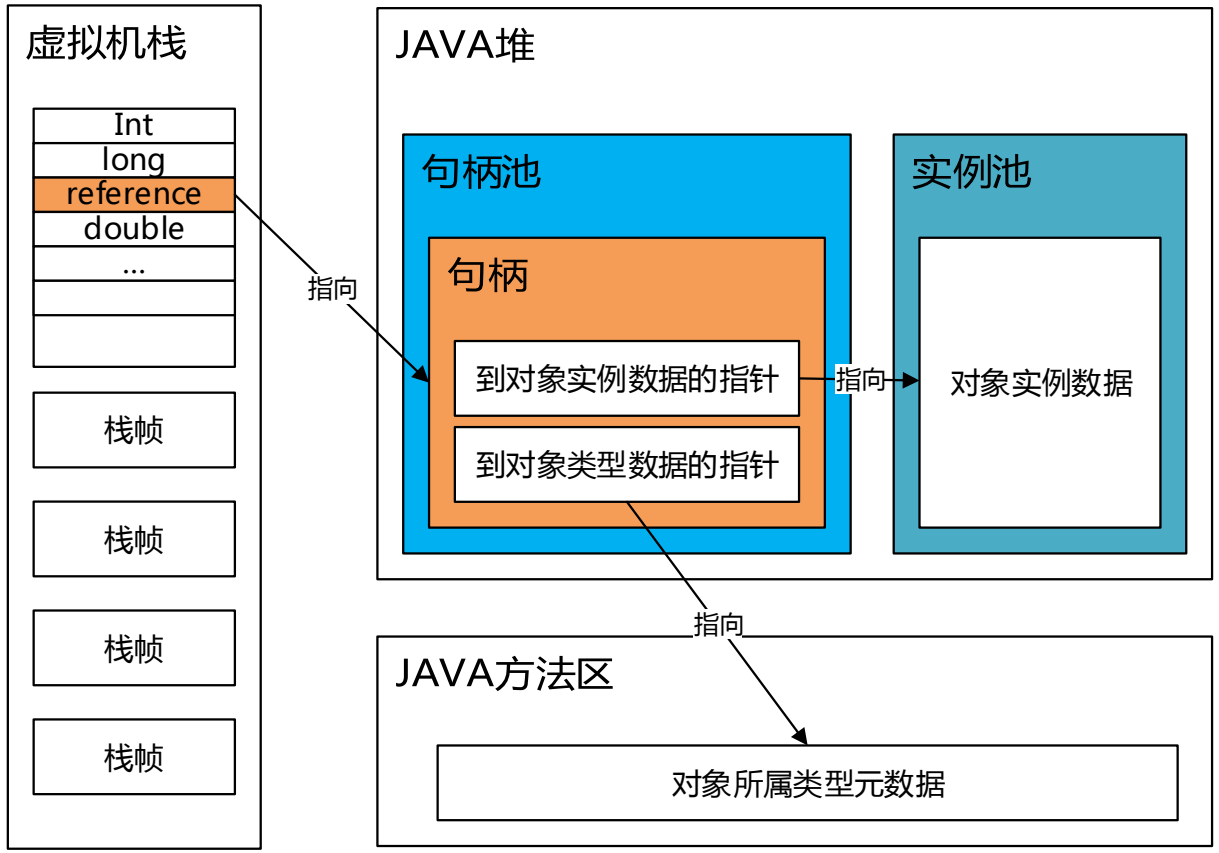
工具	说明
-XX:+PrintCommandLineFlags	打印出用户手动设置或者JVM自动设置的XX选项(如堆空间大小和所选垃圾收集器)
-XX:+PrintFlagsInitial	打印出所有XX选项的默认值
-XX:+PrintFlagsFinal	打印出XX选项在运行程序时生效的值

示例：java -XX:+PrintCommandLineFlags -version

```
-XX:InitialHeapSize=109726400 -XX:MaxHeapSize=1755622400 -XX:+PrintCommandLineFlags -XX:+UseCompress
edClassPointers -XX:+UseCompressedOops -XX:-UseLargePagesIndividualAllocation -XX:+UseParallelGC
java version "1.8.0_181"
Java(TM) SE Runtime Environment (build 1.8.0_181-b13)
Java HotSpot(TM) 64-Bit Server VM (build 25.181-b13, mixed mode)
```

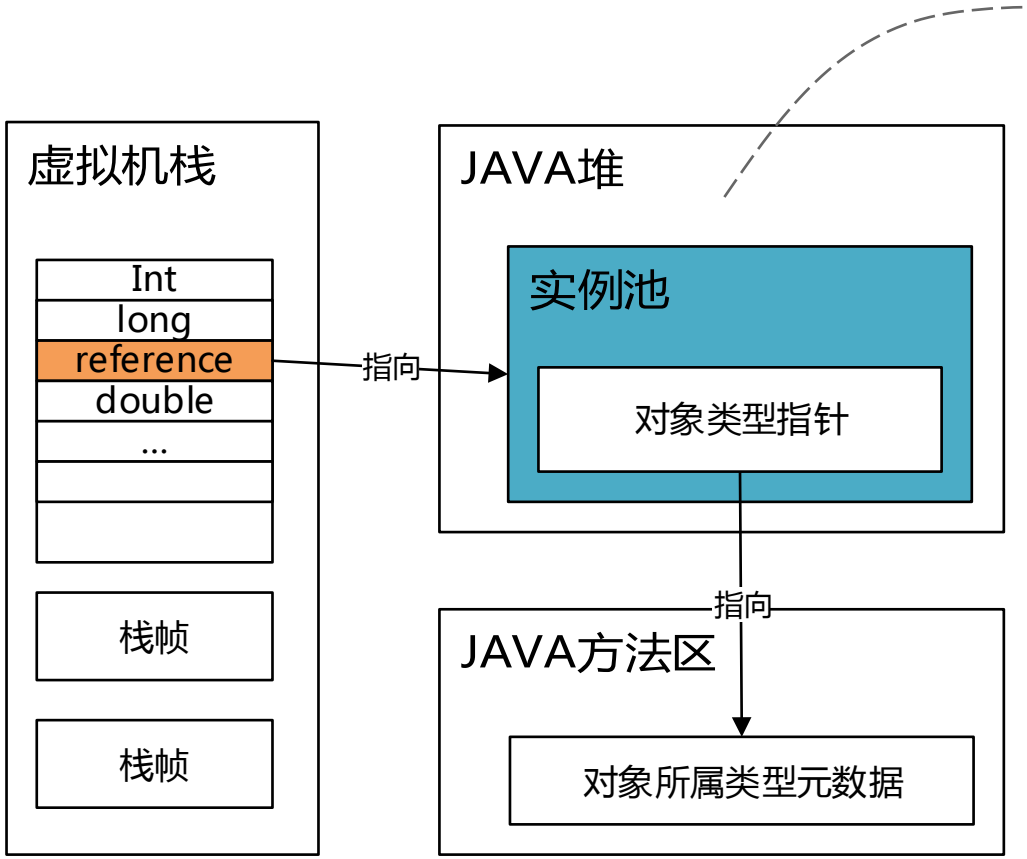
Demo: com.mimaxueyuan.jvm.xxparam.PrintXXParamDemo

对象的访问定位-句柄访问



- **句柄访问方式的优点：**在垃圾回收的时候对象要经常移动，这时候只需要改变句柄中指向对象实例数据的指针即可。而不用修改reference

对象的访问定位-直接访问



- 采用句柄的方式访问对象，需要在Java堆中分配出一部分空间，用于存放对象的句柄。这部分空间可称为句柄池

- **直接访问方式的优点：**减少了一次指针定位的时间开销，JAVA对象的访问十分频繁，效率的提升积少成多，十分可观。Sun HotSpot虚拟机采用直接访问的方式来进行对象定位

对象的内存布局

Mark Word

存储对象自身运行时数据
哈希码(hashCode)、GC分段年龄、锁状态标志、线程持有的锁、偏向线程ID、偏向时间戳等

在32位和64位虚拟机中，分别用32bit和64bit表示，官方成为Mark Word

类型指针

类型指针，对象指向类元数据的指针，虚拟机通过它来找到对象是哪个类的实例

数组长度

如果对象是一个数组，还需要有一块记录数组长度的数据
普通对象可以通过它的元数据(类信息)确定其大小，但是数据组从数据的元数据却无法确认其大小

对齐填充

非必须存在，无特殊含义
HotSportVM要求对象的起始地址必须是8字节的整数倍
因此对象的大小必须大于8字节

当对象实例部分数据没有对齐的时候，就通过对齐补充来补全

实例数据

对象本身的各类型的字段内容，包含从父类继承的内容
顺序受到虚拟机分配策略和字段在源码中的顺序影响

HotSpot的默认分配策略为longs/doubles,ints,shorts/chars,bytes/booleans,oops(Ordinary Object Pointer,一般对象指针)，相同宽度的字段分配到一起

认识GC

为什么要研究GC

排查各种内存溢出、内存泄漏问题
垃圾回收会成为系统高并发的瓶颈

GC主要针对的区域

由于JAVA虚拟机栈、本地方法栈、程序计数器都是线程私有的，随着线程的创建而创建，随着线程的消亡而消亡，所以不需要过多关注

JAVA堆是GC的重点区域

GC日志输出

-XX:+PrintGC 输出GC日志
-XX:+PrintGCDetails 输出GC的详细日志
-XX:+PrintGCTimeStamps 输出GC的时间戳（以基准时间的形式）
-XX:+PrintGCDateStamps 输出GC的时间戳（以日期的形式，如 2013-05-04T21:53:59.234+0800）
-XX:+PrintHeapAtGC 在进行GC的前后打印出堆的信息
-Xloggc:../logs/gc.log 日志文件的输出路径

- **com.mimaxueyuan.jvm.gc.GCLogDemo0**

怎样判断对象已死、可以回收

判断对象是否已经死了

- 引用计数算法
- 可达性分析算法

引用计数法(Reference Counting)

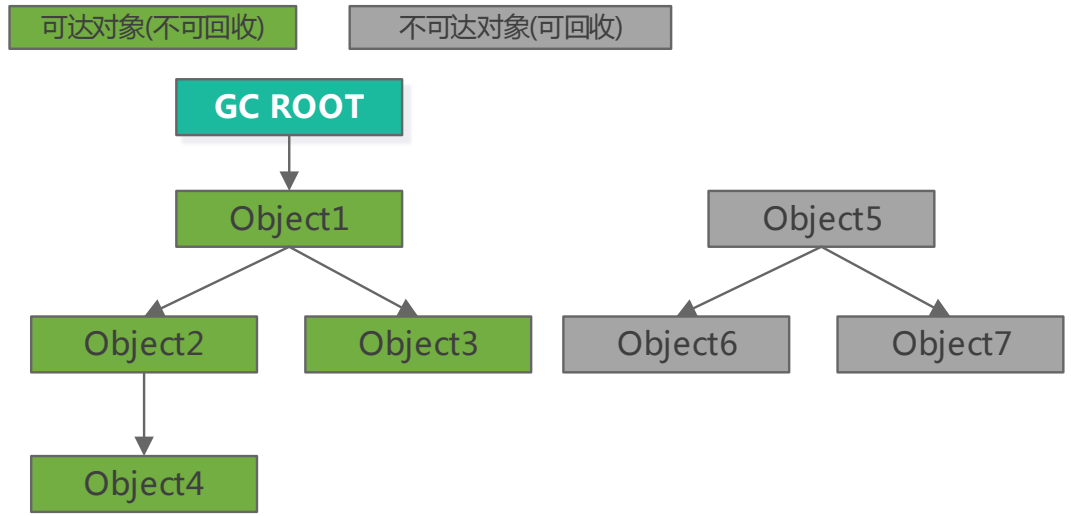
给对象添加一个引用计数器，有地方引用则加1，引用失效就减1，当引用计数器为0时则随时可被回收

实现简单、判断效率高；微软的COM（Component Object Model）、使用ActionScript3的FlashPlayer、Python、游戏领域的Squirel都使用此种算法

无法解决对象之间循环引用的问题
DEMO

可达性分析(Reachabliity Analysis)

从GC Roots作为起点开始搜索，那么整个连通图中的对象便都是活对象，对于GC Roots无法到达的对象便成了垃圾回收的对象，随时可被GC回收。



可以作为GC root的对象

1、栈针中的本地变量中的引用对象
2、方法区中的静态属性引用的对象
3、方法区中常量引用的对象
4、本地方法栈中JNI引用的对象

可达性分析-SafePoint、SafeRagion

- 从GCRoot查找对象的可达路径不应该是逐个的进行枚举检查，否则量太大了，效率存在问题
- 在整个可达性分析期间要保证对象之间的引用关系不再变化，否则准确性就无法保证。这就导致在GC进行时必须停顿所有的JAVA执行线程（Sun将其称为Stop The World）
- 在HotSpot实现中,是使用一组称之为OopMap的数据结构来记录哪些地方存放着对象的引用。 就不需要用穷举的方式去确定引用了。大大的提升了效率。

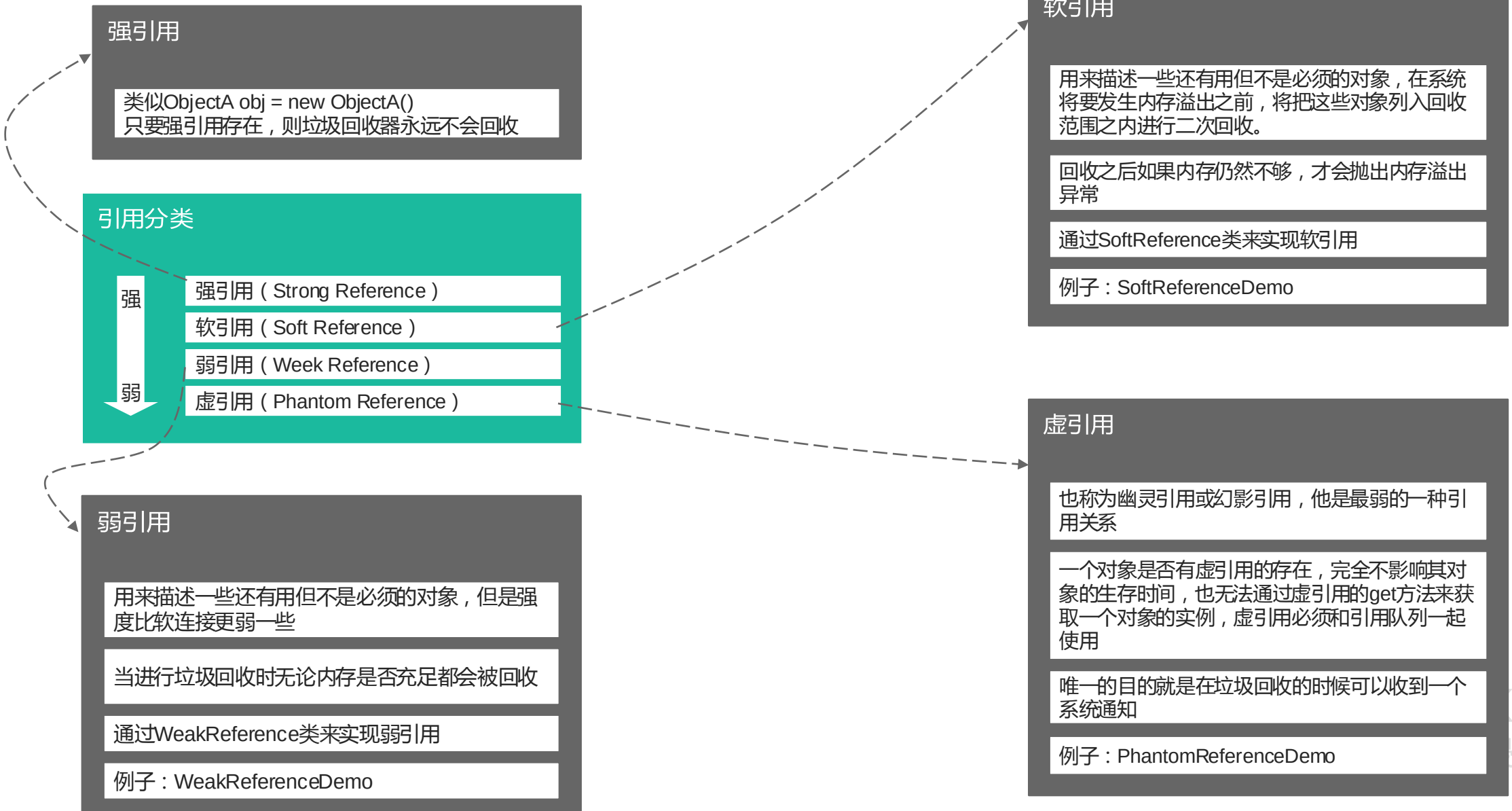
- safePoint是程序中的某些位置，线程执行到这些位置时，线程中的某些状态是确定的，在safePoint可以记录OopMap信息，线程在safePoint停顿，虚拟机进行GC。

- 线程停顿方式有两种，抢先式中断和主动式中断：
- 抢先式中断：虚拟机需要GC时，中断所有线程，让没有到达safePoint的线程继续执行至saftPoint并中断
- 主动式中断：虚拟机不直接中断线程，而是在内存中设置标志位，线程检查到标志位被设置，运行至saftPoint时主动中断

- safePoint一般出现在以下位置：循环体的结尾、方法返回前、调用方法的call之后、抛出异常的位置
- 这些位置保证线程不会长时间运行而无法到达safePoint，避免其他线程都停顿等待本线程。

- safePoint无法解决线程未达到safePoint并处于休眠或等待状态的情况，因此引入safeRegion的概念。
- safeRegion是代码中的一块区域或线程的状态，在safeRegion中，线程执行与否不会影响对象引用的状态。线程进入safeRegion会给自己加标记，告诉虚拟机可以进行GC；线程准备离开safeRegion前会询问虚拟机GC是否完成。

四种对象引用



两次标记与finalize

两次标记

确认一个对象的死亡要经过两次标记
1、确认GCRoot到此对象已经没有任何可达路径

2、判断此对象是否有必须要执行finalize方法，当对象没有覆盖finalize()方法或者finalize()已经被虚拟机调用过, 则虚拟机都不会再去执行finalize

finalize执行

如果一个对象确认需要执行finalize方法,那么会将此对象放入一个F-Queue的队列中，并且由一个虚拟机自动创建的、低优先级的Finalizer线程去执行

虚拟机执行finalize方法，可能不会等到它运行结束；因为如果某一个对象的finalize执行缓慢、死循环等都会影响后续的finalize方法执行；还会导致垃圾回收系统混乱。

finalize是对象拯救自己不被回收的最后一次机会，但是及其不推荐

例子：FinalizeDemo

方法区回收

回收方法区

回收方法区的性价比远远低于回收JAVA堆，常规的一次GC可以回收Heap的70%~95%的空间，而对于方法区则非常有限

方法区回收的内容主要废弃的常量和无用的类

回收废弃的常量

常量池里的对象没有任何引用，也没有任何地方引用这个字面量

回收无用的类

该类的所有实例都已经被回收，不存在任何这个类的实例

加载这个类的ClassLoader已经被回收

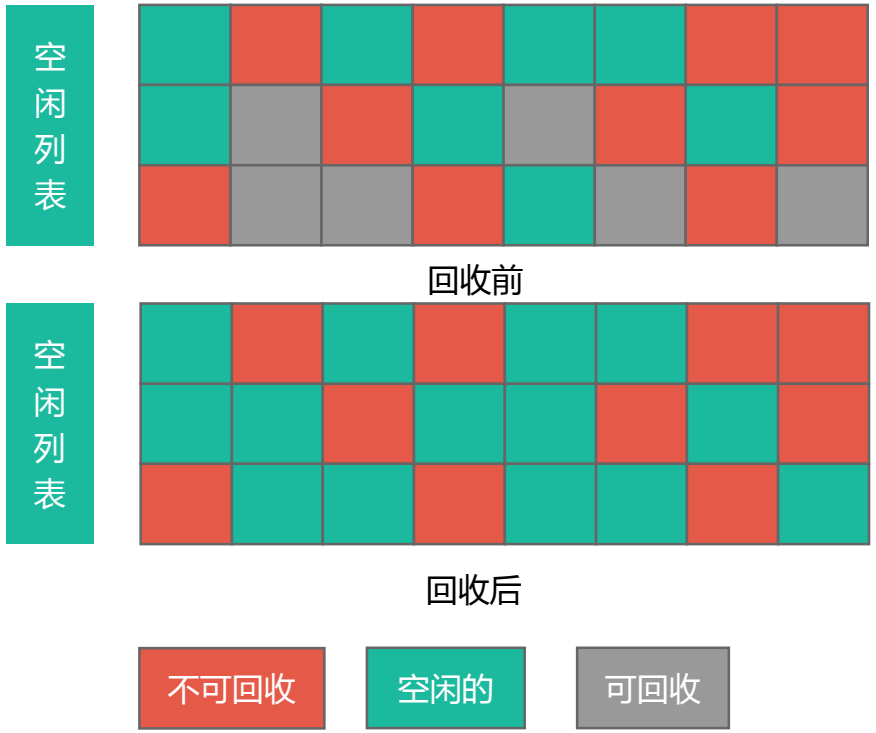
该类对应的java.lang.Class对象没有任何地方引用，无法在任何地方通过反射访问该类

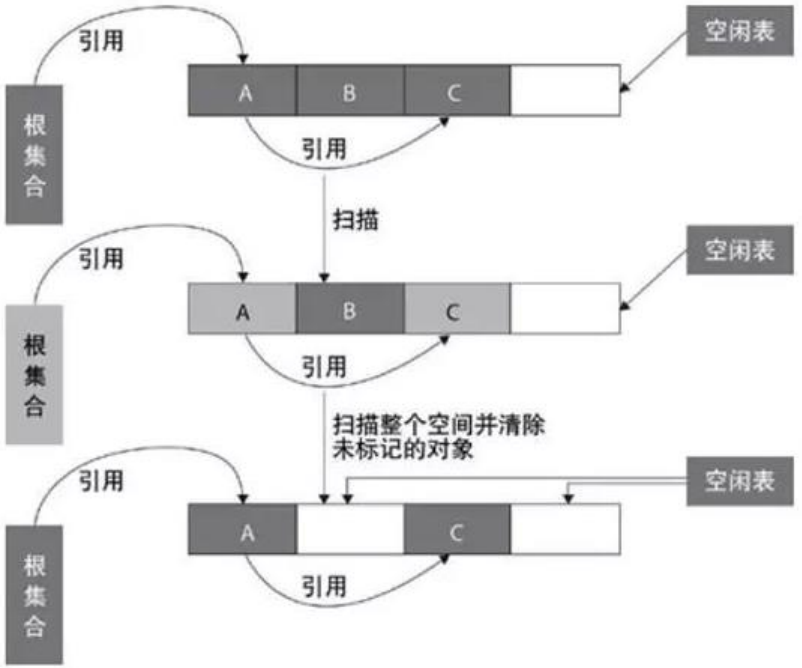
GC四种算法

- 标记清除算法 Mark-Sweep
- 复制算法 Copy
- 标记整理算法 Mark-Compact
- 分代收集策略

GC算法-标记清除 Mark-Sweep

- 最基础的垃圾回收算法，分为标记、清除两个阶段
- 首先标记出所有要回收的对象，在标记完成后统一清除被标记对象
- 存活对象较多的情况下比较高效
- 适用于年老代（即旧生代）

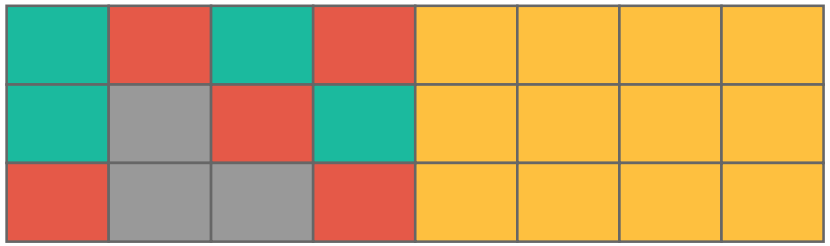




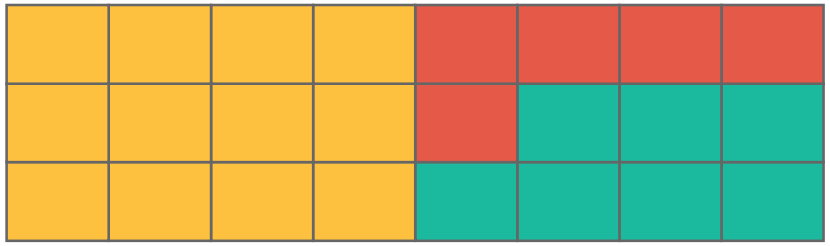
- 扫描了整个空间两次（第一次：标记存活对象；第二次：清除没有标记的对象），效率比较低
- 容易产生内存碎片，再来一个比较大的对象时（典型情况：该对象的大小大于空闲表中的每一块儿大小但是小于其中两块儿的和），会提前触发垃圾回收

GC算法-复制算法 Copy

- 将内存按照大小划分为容量相等的两块，每次只使用其中一块。当这一块用完了，就将还存活的对象复制到另一块上去。然后再把已使用过的内存空间一次清除
- 可以解决效率问题

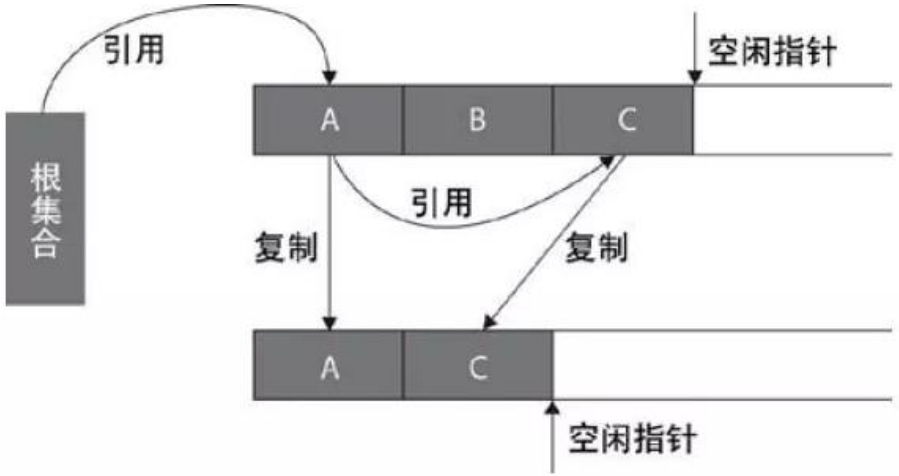


回收前



回收后

不可回收　空闲的　可回收　保留区

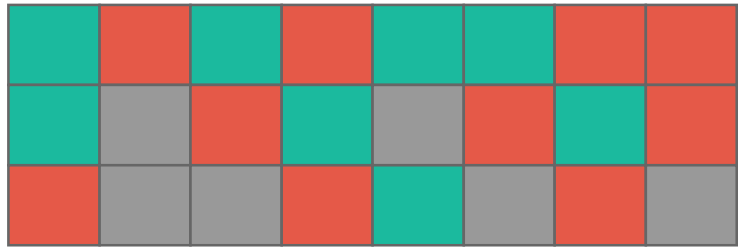


- 存活对象较少的情况下比较高效,
- 适用于年轻代：基本上98%的对象是"朝生夕死"的，存活下来的会很少
- 每次都是对整个半内存块进行清理，效率提升，只扫描了整个空间一次（标记存活对象并复制移动）
- 也不需要考虑内存碎片问题，只需要移动堆顶端指针，按顺序分配内存即可

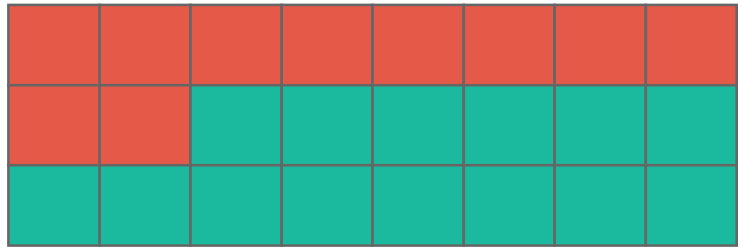
- 可用内存被缩减成了一半，代价较大
- 需要复制移动对象

GC算法-标记整理算法 Mark-Compact

- 分为两步，先标记，再整理
- 先标记出可清理的对象，然后将可清理的对象向一边移动， 让不可清理的对象向另外一段移动。 然后再把可清理对象清除
- 解决复制算法导致的内存区减半问题，解决内存碎片不连续问题

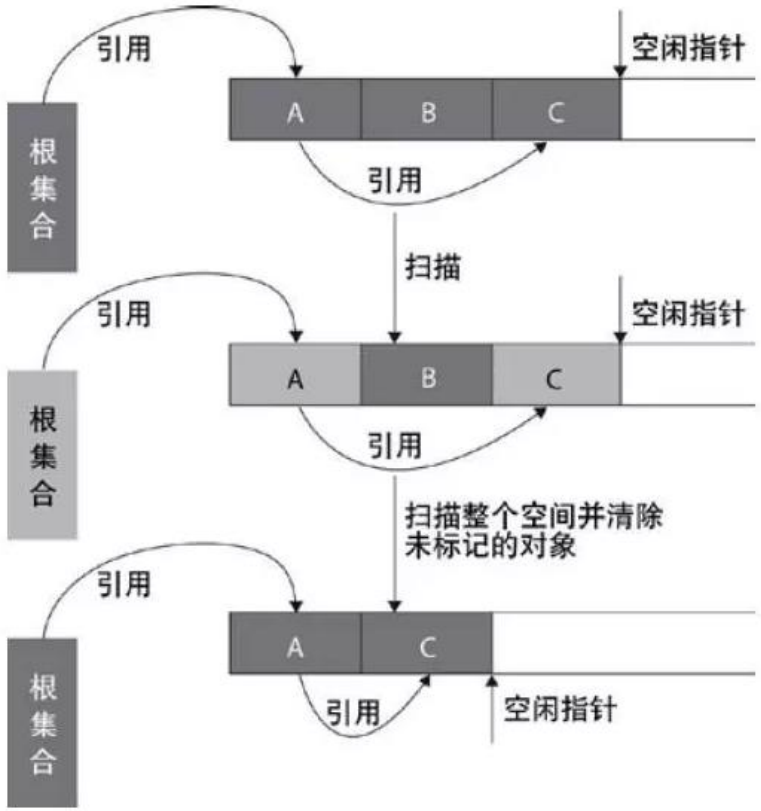


回收前



回收后

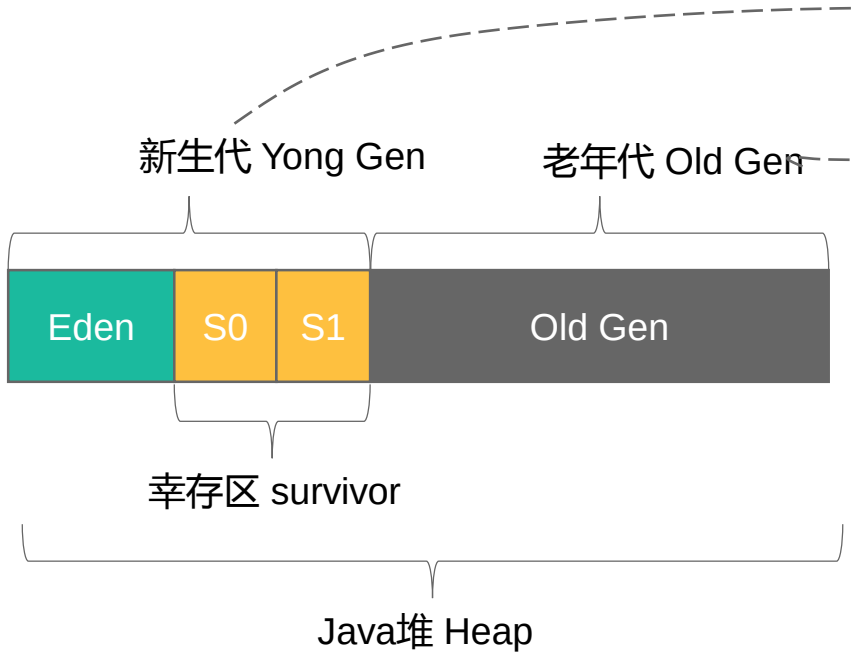
不可回收　空闲的　可回收



- 需要扫描两次
- 需要移动对象

GC算法-分代收集策略

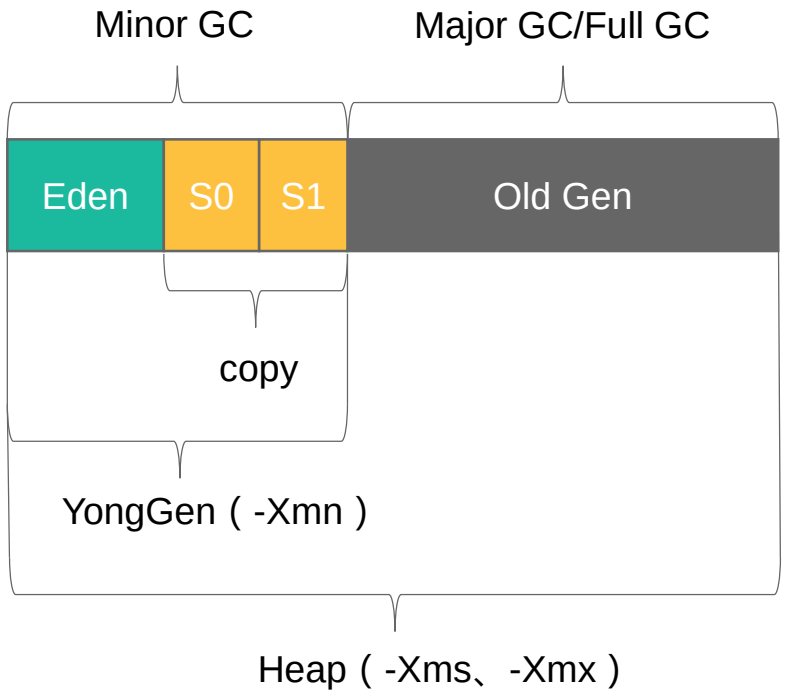
- 根据对象的存活周期不同将内存划分为几块
- 将JAVA 堆分为新生代和老年代，根据不同年代的特点采用适合的算法



- 新生代中每次垃圾回收都发现有大量的对象死去，少量存活，因此采用复制算法，只要完成少量对象的复制就可以完成收集
- 老年代中对象的存活率高、没有额外的空间对他进行分配担保，因此使用[标记-清除]或[标记-整理]算法

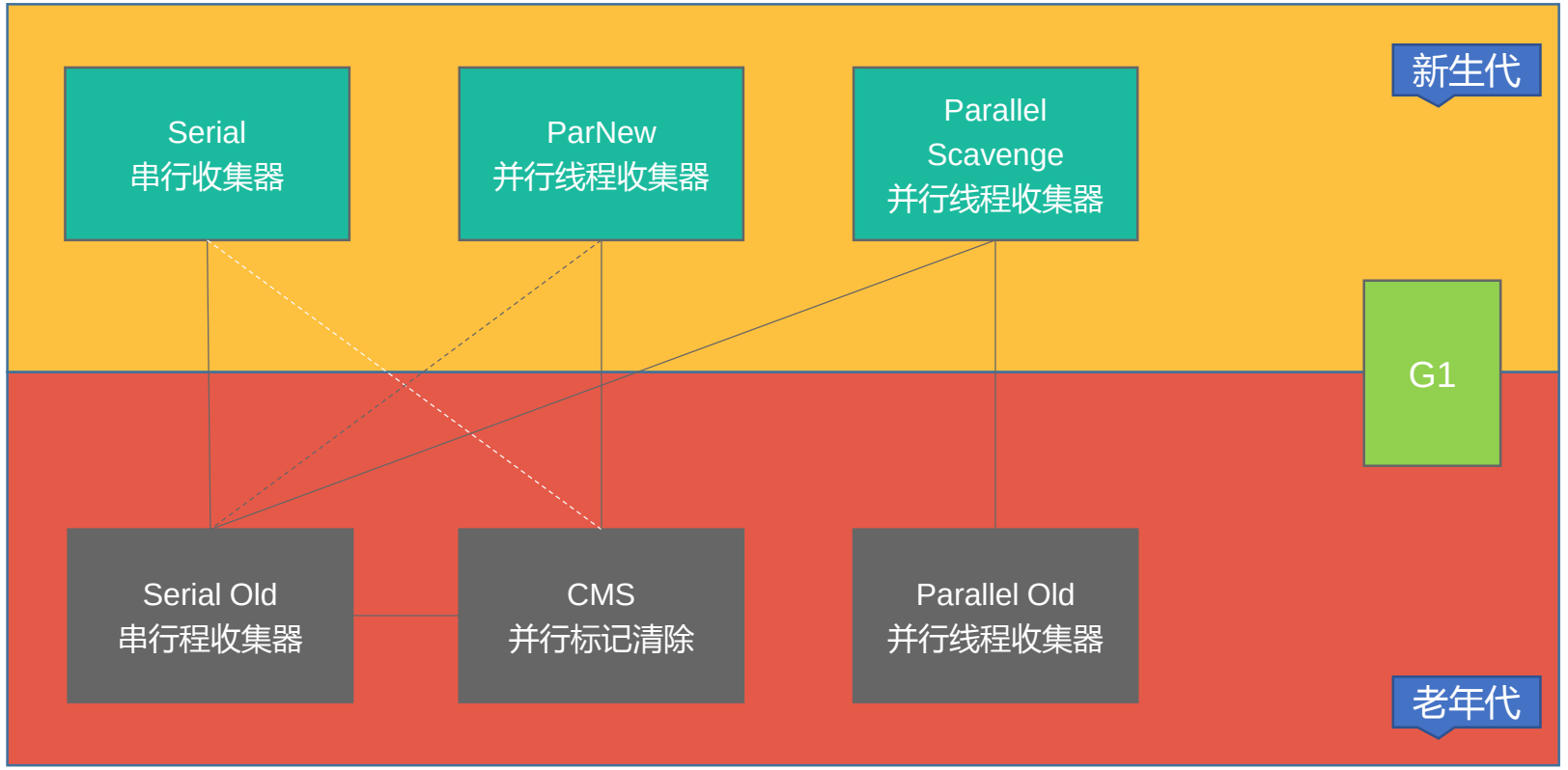
- 年轻代分为Eden区和survivor区（两块儿：from和to），且Eden:from:to=8:1:1

Minor GC、Major GC、Full GC



- 所有新生成的对象首先都是放在年轻代。年轻代的目标就是尽可能快速的收集掉那些生命周期短的对象。当年轻代内存空间被用完时，就会触发垃圾回收。这个垃圾回收叫做Minor GC
- 新生代内存按照8:1:1的比例分为一个eden区和两个survivor(s0,s1)区。一个Eden区，两个 Survivor区。大部分对象在Eden区中生成。回收时先将eden区存活对象复制到一个S0区，然后清空eden区，当这个S0区也存放满了时，则将eden区和S0区存活对象复制到另一个survivor1区，然后清空eden和这个S0区，此时S0区是空的，然后将S0区和S1区交换，即保持S1区为空， 如此往复。
- 如上这样，会有很多对象会被复制很多次（每复制一次，对象的年龄就+1），默认情况下，当对象被复制了15次（这个次数可以通过：-XX:MaxTenuringThreshold来配置），就会进入年老代了。
- 当S1区不足以存放 eden和S0的存活对象时，就将存活对象直接存放到老年代。若是老年代也满了就会触发一次Full GC(Major GC)，也就是新生代、老年代都进行回收。

7种垃圾回收器及组合方式



- 不同厂商,不同版本的垃圾收集器有很大的区别，虚拟机规范中并没有明确规定垃圾收集器应该如何实现。
- 没有最好的垃圾收集器，只有最适合的（根据场景不同）
- 两个收集器之间存在连线，则说明他们可以搭配使用

新生代收集器-Serial

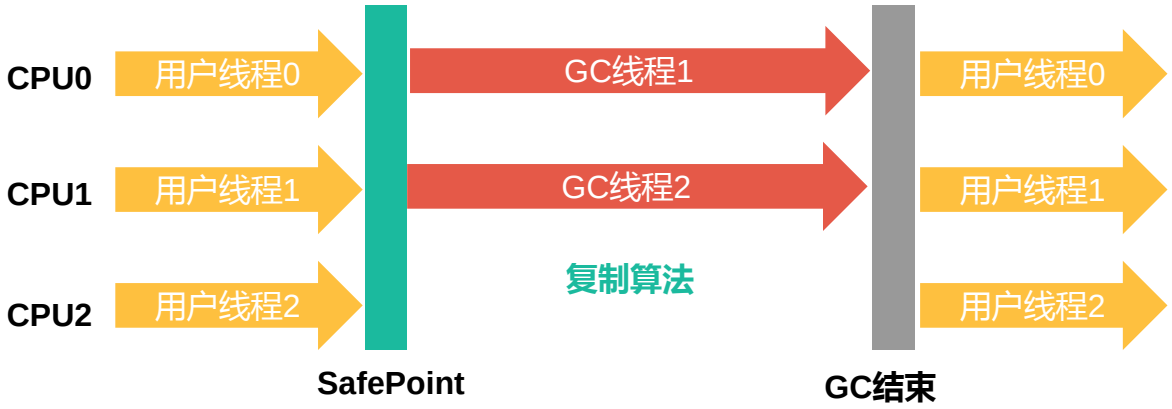
- Serial是单线程收集器，只会用一个CPU或一个线程去完成垃圾回收工作；并且在进行垃圾回收时，必须暂停所有其他的线程，直到收集结束为止。
- 它是虚拟机运行在Client模式（桌面）下的默认新生代收集器；简单高效；对于限定按单个CPU的环境来说，Serial收集器由于没有线程交互的开销，因而可以获得最高的单线程收集效率。

老年代收集器-Serial Old

- Serial Old是Serial收集器的老年代版本，同样是一个单线程收集器，使用标记整理算法，收集的过程中需要暂停所有用户线程

- 和Serial收集器一样，主要也用于Client模式下；如果在Server模式下有两大用途：
- 在JDK1.5之前的版本中与Parallel Scavenge收集器搭配使用
- 作为CMS收集器的的后背方案，在并发收集发生Concurrent Mode Failure时使用

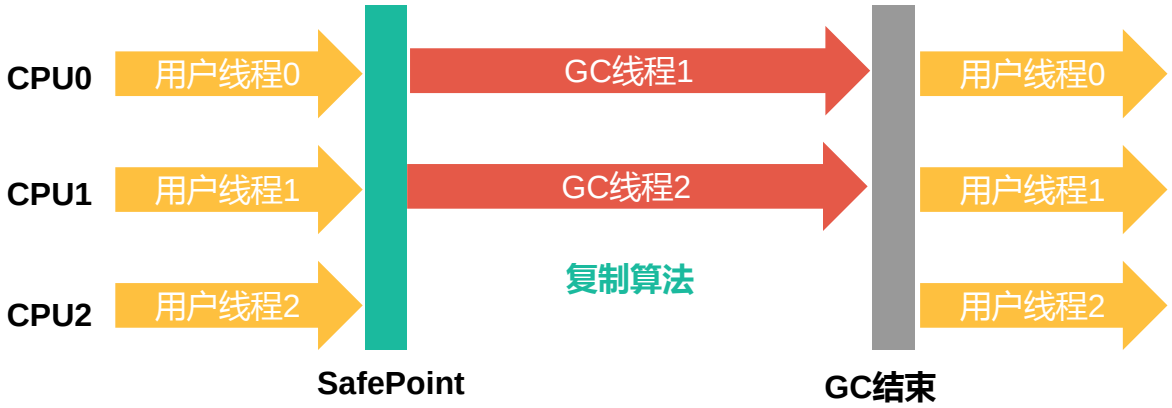
新生代收集器-ParNew



- ParNew（ Parallel New ）收集器是Serial的多线程版本；也是新生代收集器；除了使用多个线程进行垃圾回收之外， 其余行为包括Serial收集器可用的所有控制参数、收集算法、Stop The Word、对象分配规则、回收策略等都和Serial收集器完全一样。

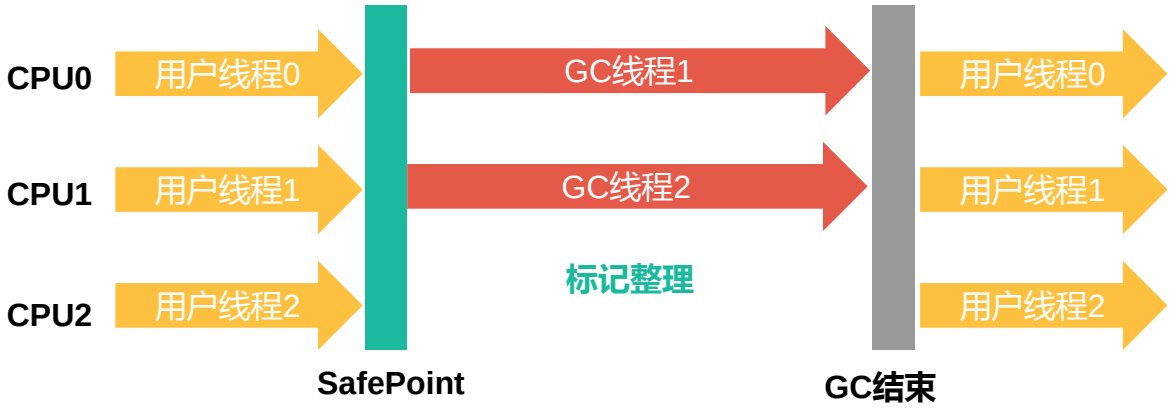
- ParNew收集器是许多运行在Server模式下的虚拟机首选的新生代收集器，因为目前只有它可以与CMS收集器配合工作。
- ParNew收集器在单CPU环境下性能不会比Serial更好
- ParNew开启的收集线程数与CPU的数量相同，在多CPU的环境下它可以有效的利用资源

新生代收集器-Parallel Scavenge, 并行扫描



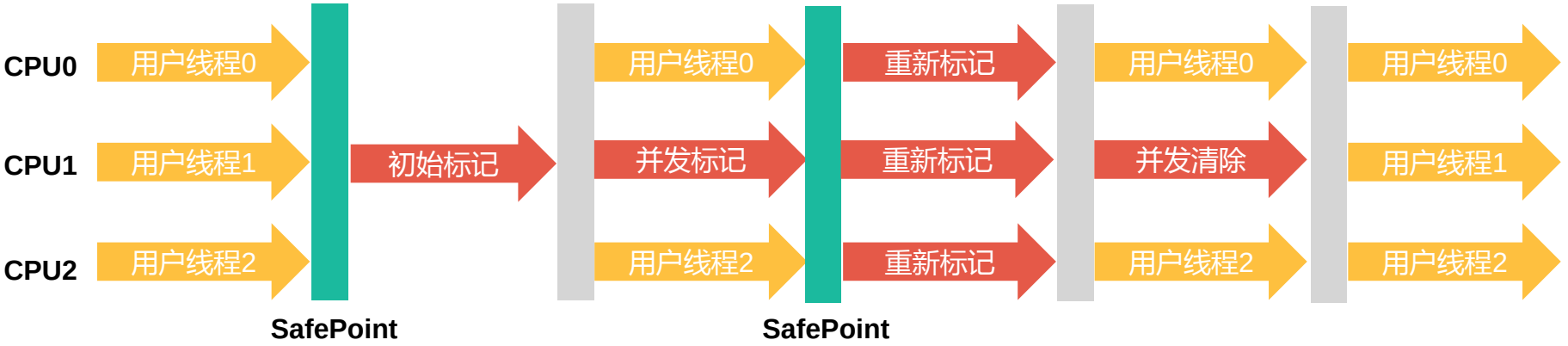
- Parallel Scavenge收集器是并行的多线程新生代垃圾收集器，也是采用复制算法
- Parallel Scavenge收集器与其他收集器最大的特点是关注点不同，CMS等收集器的目的是尽可能的缩短垃圾回收时用户线程的停顿时间；而Parallel Scavenge收集器目的是达到一个可控制的吞吐量（Throughput）。吞吐量=运行用户代码时间/(运行用户代码时间+垃圾收集时间)
- 停顿时间越短则响应速度越快，用户体验越好，所以更适合交互性强的服务程序。而吞吐量高可以高效利用CPU资源，尽快完成程序的运算，主要适合后台运算多交互少的任务。
- Parallel Scavenge收集器也叫吞吐量优先收集器

老年代收集器-Parallel Old



- Parallel Old收集器是Parallel Scavenge收集器的老年代版本，使用多线程和标记-整理算法；这个收集器是在JDK1.6才开始提供的
- 再JDK1.6之前新生代的Parallel Scavenge收集器一直处于比较尴尬的境地，它只能和Serial Old收集器搭配使用，但是由于Serial Old在服务端的性能较差， 所以根本无法获得吞吐量最大化的效果。
- 直到Parallel Old收集器出现后，才真正形成了吞吐量优先的组合；在注重吞吐量和CPU资源敏感的场合，都可以优先考虑Parallel Scavenge新生代加Parallel Old老年代收集器的策略。

老年代收集器-CMS、Concurrent Mark Sweep 多线程标记清除

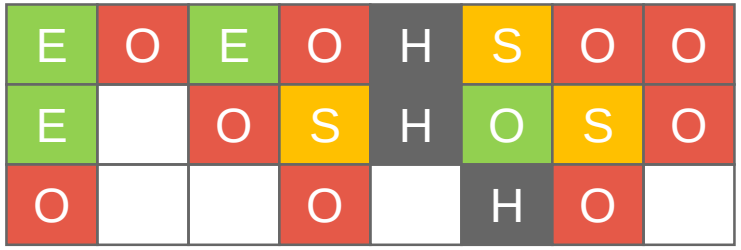


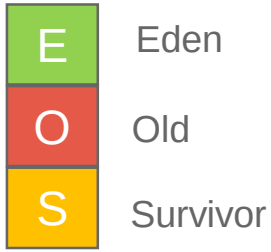
- CMS收集器的目的是把回收停顿时间降低到最短；对于B/S架构的互联网系统，要求服务器的响应速度较高，则适合使用CMS收集器。
- 它的运行过程分为四步：1、初始标记（CMS initial mark）2、并发标记（CMS concurrent mark）3、重新标记（CMS remark）4、并发清除（CMS concurrent sweep）
- 初始标记和重新标记两个过程依然要停止用户线程（Stop The Word）。
  - 初始标记：单线程、STW、仅仅只是标记一下GCRoot能直接关联到的对象，速度很快
  - 并发标记：进行GCRoot Tracing的过程
  - 重新标记：多线程、STW、为了修正并发标记期间因用户线程继续运行而导致标记发生变动的那一部分对象的标记记录，这个阶段的停顿时间一般比初始标记稍长，但是远比并发标记时间要短。

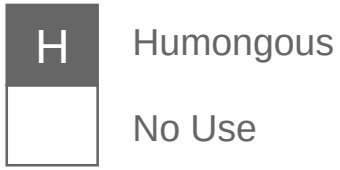
新老年代收集器-G1

- **介绍**
- 2012年才在jdk1.7u4种可以使用，Oracle在jdk1.9中将G1变为默认的垃圾收集器，来代替CMS

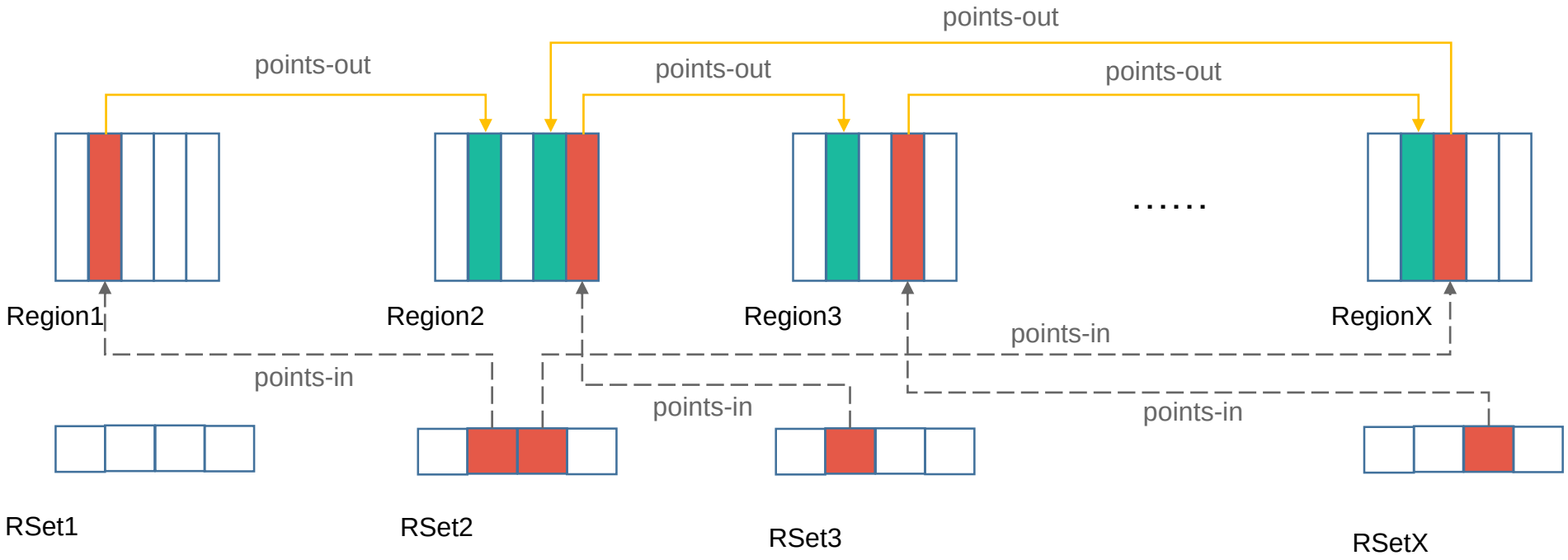
- **优点**
- 并发与并行：G1能充分利用多CPU、多核环境的硬件优势，缩短STW(stop the word)停顿时间
- 分代收集：不需要其他收集器配合就能管理整个GC堆，同时管理新生代和老年代
- 空间整理：整体上基于标记-整理算法，局部基于复制算法，G1运行期间不会产生内存空间碎片，收集后能提供规整的可用内存。
- 可预测的停顿：G1除了降低了STW时间外，还能建立可预测的停顿时间模型，比如指定在M毫秒内，消耗在GC上时间不得超过N毫秒。
- 将新生代、老年代的物理空间划分取消了，再也不用单独设置每个代的大小了，也不用担心它们的内存是否足够。
- G1也采用分代垃圾回收策略







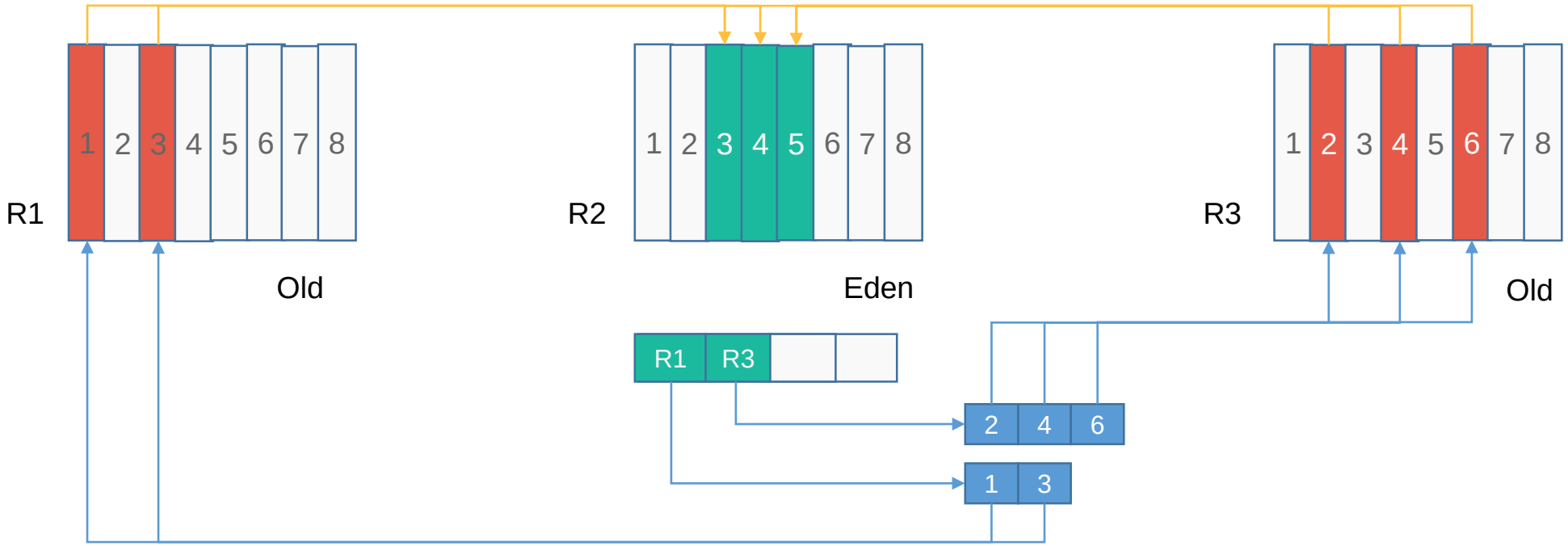
G1垃圾收集器核心-Card和RSet



Card 和 RSet

- 每一个Regin被分为大小相等的Card，每个Card大小为512bytes
- 红色代表引用其他对象、蓝色代表被其他对象引用
- 逻辑上每一个Region都有一个Rset，全称是Remember Set，是一种GC辅助结构，空间换时间策略。
- Rest使用points-in的方式记录points-out引用， Rest中只记录其他Region到本Region的引用关系，本Region内部的引用关系不记录

G1垃圾收集器核心-RSet



- HashTable结构，Key值为引用对象所在Region的起始地址、Value为引用对象所在的Card索引
- 主要记录young->old、old->old之间的引用，如果不记录，就要扫描整个old区去确认对象是否存活，性能低下。
- Young GC的时候，只要扫描所有的年轻代Region的Rset，就可以确认所有老年代到年轻代的引用，不用扫描整个老年代。
- Mixed GC的时候，只要扫描所有老年代Region的Rset，就可以确认所有老年代之间的引用、以及新生代到老年代的引用，这样不用扫描整个年轻代和老年代。
- 总结：付出了空间、维护的代价、带来了GC回收效率的提升

G1垃圾收集器核心-收集类别

单线程Serial Old

Young GC

- Young GC收集年轻代里的Region，主要是对Eden区进行GC，它在Eden空间耗尽时会被触发。在这种情况下，Eden空间的数据移动到Survivor空间中，如果Survivor空间不够，Eden空间的部分数据会直接晋升到年老代空间。Survivor区的数据移动到新的Survivor区中，也有部分数据晋升到老年代空间中。最终Eden空间的数据为空。

阶段1：根扫描：静态和本地对象被扫描
阶段2：更新RS：处理dirty card更新RS
阶段3：处理RS：检测从年轻代指向年老代的对象 -> 记录Cset (Collection Set)
阶段4：对象拷贝：拷贝存活的对象到survivor/old区域
阶段5：处理引用队列：软引用，弱引用，虚引用处理

G1垃圾收集器核心-Mixed GC

Mixed GC

- 回收年轻代的所有Region、以及全局并发标记阶段选出的老年代Region。
- 当old region的对象占总Heap的比例超过阈值（默认45%）之后，就会开始并发标记（Concurent Marking），完成并发标记后，G1会从Young GC切换到Mixed GC, 在Mixed GC中，G1可以增加若干个Old区域的Region到CSet中。
- -XX:InitiatingHeapOccupancyPercent=45，开始一个标记周期的堆占用比例阈值，默认45%，注意这里是整个堆，不同于CMS中的Old堆比例。

阶段1：全局并发标记（global concurrent marking）

S1.1初始标记（initial mark，STW）：暂停所有线程，标记出所有可以直接从GC roots可以到达的对象，这是在Young GC的暂停收集阶段顺带进行的

S1.2根区域扫描（root region scan）：找出所有的GC Roots的Region, 然后从这些Region开始标记可到达的对象。

S1.2并发标记（Concurrent Marking）：在整个堆中查找存活的对象。该阶段与应用程序同时运行。

S1.3最终标记（Remark，STW）：处理SATB(Snaphot-At-The-Beginning标记算法) 缓冲区，跟踪未被访问的存活对象。

S1.4清除垃圾（Cleanup，STW）：执行统计和 RSet 净化的 STW 操作。在统计期间，G1 GC 会识别完全空闲的区域和可供进行混合垃圾回收的区域。

阶段2：拷贝存活对象（evacuation）

G1垃圾收集器核心-全局并发标记

阶段1：全局并发标记（global concurrent marking）：标记存活的对象、并且计算各个Region的活跃度

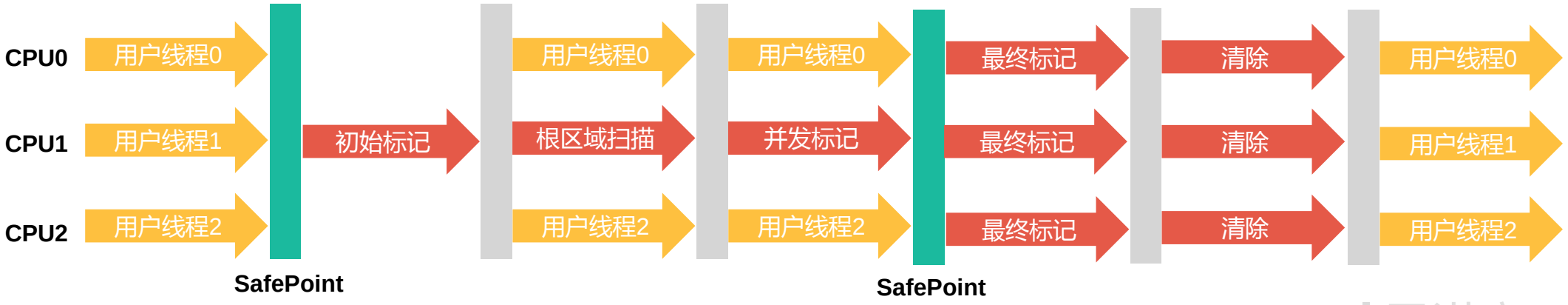
S1.1初始标记（initial mark，STW）：暂停所有线程，标记出所有可以直接从GC roots可以到达的对象，这是在Young GC的暂停收集阶段顺带进行的

S1.2根区域扫描（root region scan）：找出所有的GC Roots的Region, 然后从这些Region开始标记可到达的对象。

S1.3并发标记（Concurrent Marking）：在整个堆中查找存活的对象。该阶段与应用程序同时运行，可以被 STW 年轻代垃圾回收中断

S1.4最终标记（Remark，STW）：清空 SATB 缓冲区，跟踪未被访问的存活对象，并执行引用处理。

S1.5清除垃圾（Cleanup，STW）：执行统计和 RSet 净化的 STW 操作。在统计期间，G1 GC 会识别完全空闲的区域和可供进行混合垃圾回收的区域。

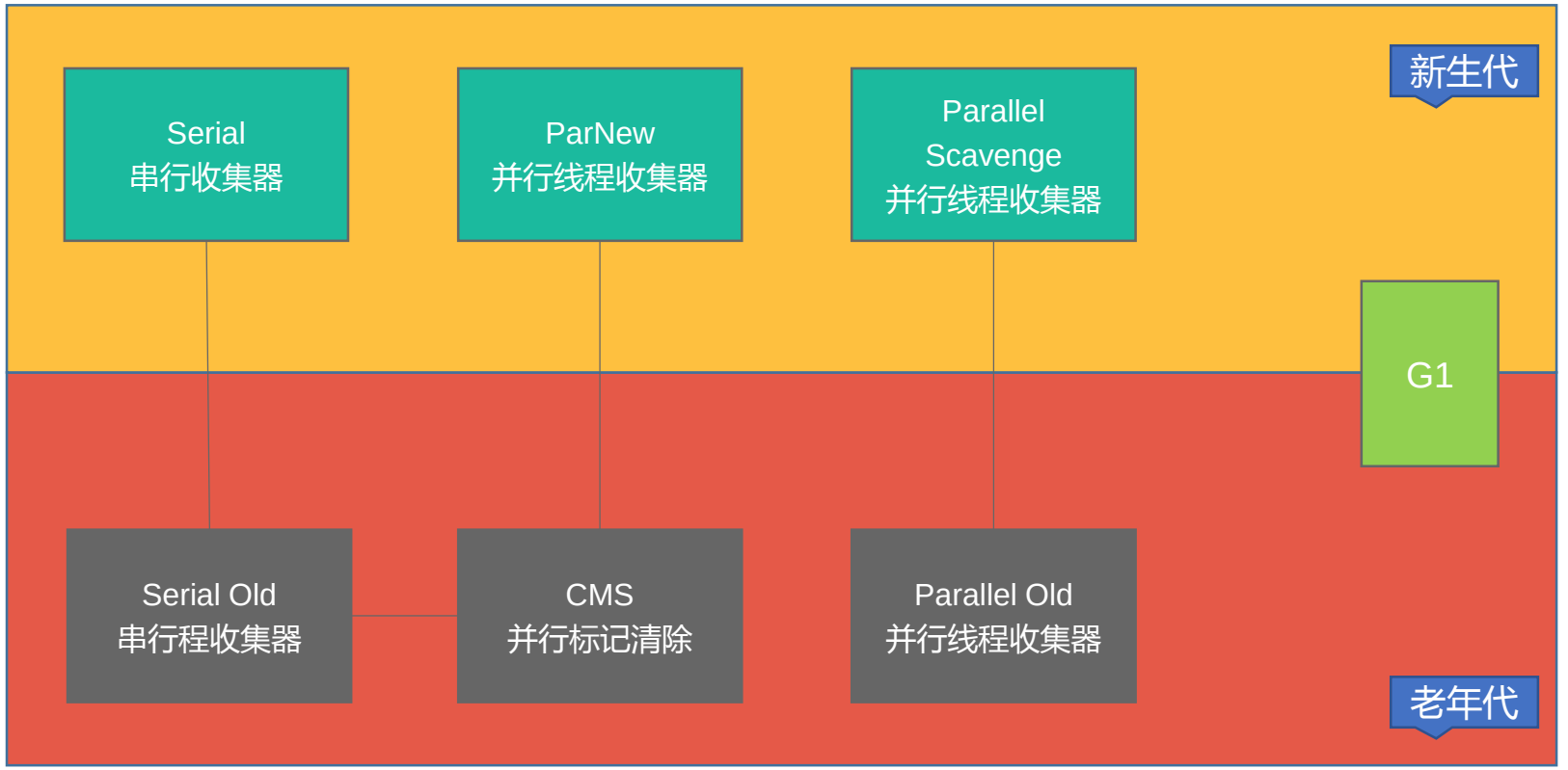


G1垃圾收集器-深入核心

Full GC

- G1的一些收集过程是和应用程序并发执行的，所以可能还没有回收完成，是由于申请内存的速度比回收速度快，新的对象就占满了所有空间，在CMS中叫做Concurrent Mode Failure, 在G1中称为Allocation Failure，这时候就会触发Full GC
- G1是不提供FullGC的，因此会触发Serial Old，使用单线程进行FullGC，一旦FullGC对性能影响十分严重

4种垃圾回收器组合方式



垃圾回收器组合及参数设置

新生代/年轻代	老年代	JVM参数
Serial(DefNew)	Serial Old	-XX:+UseSerialGC
ParNew	CMS + Serial Old	-XX:+UseParNewGC -XX:+UseConcMarkSweepGC jdk1.8种可以不使用第二个参数
Parallel Scavenge	Parallel Old	-XX:+UseParallelGC
Parallel Scavenge	Parallel Old	-XX:+UseParallelOldGC
G1	G1	-XX:+UseG1GC
ParNew	Serial Old	已经废弃，直接使用ParNew+CMS+Serial Old组合，不支持-XX:+UseParNewGC -XX:-UseConcMarkSweepGC参数组合

java -XX:+PrintCommandLineFlags -XX:+UseSerialGC -version
java -XX:+PrintCommandLineFlags -XX:+UseParNewGC -XX:+UseConcMarkSweepGC -version
java -XX:+PrintCommandLineFlags -XX:+UseParallelGC -version
java -XX:+PrintCommandLineFlags -XX:+UseParallelOldGC -version
java -XX:+PrintCommandLineFlags -XX:+UseG1GC -version
java -XX:+PrintCommandLineFlags -XX:+UseSerialGC -XX:+UseConcMarkSweepGC –version //非法
java -XX:+PrintCommandLineFlags -XX:+UseParNewGC -XX:-UseConcMarkSweepGC –version //废弃

垃圾回收器组合及参数设置

-XX:+UseSerialGC
允许使用串行垃圾收集器。对于不需要垃圾收集的任何特殊功能的小型和简单应用程序，这通常是最佳选择。默认情况下，禁用此选项，并根据计算机的配置和JVM的类型自动选择收集器

-XX:+UseParNewGC
允许在年轻代中使用并行线程进行收集。默认情况下，禁用此选项。设置-XX:+UseConcMarkSweepGC选项时会自动启用它。使用-XX:+UseParNewGC不带选项-XX:+UseConcMarkSweepGC的选择是在JDK 8弃用。

-XX:+UseConcMarkSweepGC
允许为老年代使用CMS垃圾收集器。Oracle建议您在并行清除（-XX:+UseParallelGC）垃圾收集器无法满足应用程序延迟要求时使用CMS垃圾收集器。G1垃圾收集器（-XX:+UseG1GC）是另一种选择。
默认情况下，禁用此选项，并根据计算机的配置和JVM的类型自动选择收集器。启用此选项后，将-XX:+UseParNewGC自动设置该选项，您不应禁用该选项，因为JDK 8中已弃用以下选项组合：-XX:+UseConcMarkSweepGC -XX:-UseParNewGC。

-XX:+UseParallelGC
允许使用并行清除垃圾收集器（也称为吞吐量收集器），通过利用多个处理器来提高应用程序的性能。
默认情况下，禁用此选项，并根据计算机的配置和JVM的类型自动选择收集器。如果已启用，则会-XX:+UseParallelOldGC自动启用该选项，除非您明确禁用它。

-XX:+UseParallelOldGC
允许将并行垃圾收集器用于完整的GC。默认情况下，禁用此选项。启用它会自动启用该-XX:+UseParallelGC选项。

-XX:+UseG1GC
允许使用垃圾优先（G1）垃圾收集器。它是一个服务器式垃圾收集器，针对具有大量RAM的多处理器计算机。它以高概率满足GC暂停时间目标，同时保持良好的吞吐量。G1收集器推荐用于需要大堆（大小约为6 GB或更大）且GC延迟要求有限的应用（稳定且可预测的暂停时间低于0.5秒）。
默认情况下，禁用此选项，并根据计算机的配置和JVM的类型自动选择收集器。

GC日志详解-参数详解

-verbose:gc
显示有关每个垃圾回收（GC）事件的信息。

-XX:+PrintGC
允许在每个GC上打印消息。默认情况下，禁用此选项。

-XX:+PrintGCDateStamps
允许在每个GC上打印日期戳。默认情况下，禁用此选项。

-XX:+PrintGCDetails
允许在每个GC上打印详细消息。默认情况下，禁用此选项。

-XX:+PrintGCTaskTimeStamps
允许为每个GC工作线程任务打印时间戳。默认情况下，禁用此选项。

-XX:+PrintGCTimeStamps
允许在每个GC上打印时间戳。默认情况下，禁用此选项。

-XX:+PrintHeapAtGC
在GC发生之前与之后打印堆的详细信息

Demo com.mimaxueyuan.jvm.gc.SerialGCDemo

GC日志解析-Minor GC日志 (详细模式)

参数： -Xmn2m -Xmx6M -XX:+PrintGCDetails -XX:+PrintGCDateStamps -XX:+PrintHeapAtGC -XX:+UseSerialGC -XX:+PrintCommandLineFlags

```
{Heap before GC invocations=0 (full 0): //GC调用之前输出Heap详细信息、GC的第0次调用、第0次full gc
 def new generation   total 1856K, used 723K [0x00000000ffa00000, 0x00000000ffc00000, 0x00000000ffc00000) //新生代、总空间、已使用空间
  eden space 1664K,  43% used [0x00000000ffa00000, 0x00000000ffab4e88, 0x00000000ffba0000)  //eden区、总空间、已使用百分比
  from space 192K,   0% used [0x00000000ffba0000, 0x00000000ffba0000, 0x00000000ffbd0000)   //from区、总空间、已使用百分比
  to   space 192K,   0% used [0x00000000ffbd0000, 0x00000000ffbd0000, 0x00000000ffc00000)   //to区、总空间、已使用百分比
 tenured generation   total 4096K, used 0K [0x00000000ffc00000, 0x0000000100000000, 0x0000000100000000) //老年代、总空间、已使用空间
   the space 4096K,   0% used [0x00000000ffc00000, 0x00000000ffc00000, 0x00000000ffc00200, 0x0000000100000000)  //总空间、已使用空间
 Metaspace       used 2558K, capacity 4486K, committed 4864K, reserved 1056768K //元空间、已使用空间、容量、已提交、预留
  class space    used 279K, capacity 386K, committed 512K, reserved 1048576K
2019-05-03T22:17:41.336+0800: /*日期时间戳*/ 0.171: /*时间戳*/ [GC /*GC类型 Minor GC*/ (Allocation Failure /*GC发生的原因、新生代内存分配失败*/) 2019-05-03T22
:17:41.336+0800:/*日期时间戳*/ 0.171:/*时间戳*/ [DefNew:/*发生位置,新生代*/ 723K/*回收前新生代使用大小*/->191K/*回收后新生代使用大小*/(1856K/*新生代总大小*/),
 0.0017056 secs /*耗时毫秒*/] 723K/*回收前使用的堆大小*/->518K/*回收后堆使用大小*/(5952K/*堆总大小*/), 0.0017719 secs] [Times: user=0.00 sys=0.00, real=0.00
secs]
Heap after GC invocations=1 (full 0):   //GC调用之后输出Heap详细信息、GC的第1次调用、第0次full gc
 def new generation   total 1856K, used 191K [0x00000000ffa00000, 0x00000000ffc00000, 0x00000000ffc00000)
  eden space 1664K,   0% used [0x00000000ffa00000, 0x00000000ffa00000, 0x00000000ffba0000)
  from space 192K,  99% used [0x00000000ffbd0000, 0x00000000ffbffff8, 0x00000000ffc00000)
  to   space 192K,   0% used [0x00000000ffba0000, 0x00000000ffba0000, 0x00000000ffbd0000)
 tenured generation   total 4096K, used 326K [0x00000000ffc00000, 0x0000000100000000, 0x0000000100000000)
   the space 4096K,   7% used [0x00000000ffc00000, 0x00000000ffc51910, 0x00000000ffc51a00, 0x0000000100000000)
 Metaspace       used 2558K, capacity 4486K, committed 4864K, reserved 1056768K
  class space    used 279K, capacity 386K, committed 512K, reserved 1048576K
}
```

Serial收集器下的Minor GC日志标注

GC日志解析-Major/Full GC日志 (详细模式)

参数： -Xmn2m -Xmx6M -XX:+PrintGCDetails -XX:+PrintGCDateStamps -XX:+PrintHeapAtGC -XX:+UseSerialGC -XX:+PrintCommandLineFlags

```
{Heap before GC invocations=2 (full 0):
 def new generation   total 1856K, used 1056K [0x00000000ffa00000, 0x00000000ffc00000, 0x00000000ffc00000)
  eden space 1664K,  63% used [0x00000000ffa00000, 0x00000000ffb083d8, 0x00000000ffba0000)
  from space 192K,   0% used [0x00000000ffba0000, 0x00000000ffba0000, 0x00000000ffbd0000)
  to   space 192K,   0% used [0x00000000ffbd0000, 0x00000000ffbd0000, 0x00000000ffc00000)
 tenured generation   total 4096K, used 3590K [0x00000000ffc00000, 0x0000000100000000, 0x0000000100000000)
   the space 4096K,  87% used [0x00000000ffc00000, 0x00000000fff81900, 0x00000000fff81a00, 0x0000000100000000)
 Metaspace       used 2559K, capacity 4486K, committed 4864K, reserved 1056768K
  class space    used 279K, capacity 386K, committed 512K, reserved 1048576K
2019-05-03T22:17:41.345+0800: 0.180: [GC (Allocation Failure) 2019-05-03T22:17:41.345+0800: 0.180: [DefNew:/*发生位置:新生代*/ 1056K->1056K(1856K), 0.0000308
secs]2019-05-03T22:17:41.345+0800: 0.180: [Tenured: /*发生位置:老年代*/ 3590K/*回收前老年代使用大小*/->3590K/*回收后老年代使用大小*/(4096K/*老年代总大小*/),
0.0018125 secs /*老年代回收耗时*/] 4647K/*回收前堆占用*/->4614K/*回收后堆占用*/(5952K/*堆总大小*/), [Metaspace: /*发生位置:元空间*/2559K
/*回收前占用元空间大小*/->2559K/*回收后占用元空间大小*/(1056768K/*元空间总大小*/)], 0.0018955 secs/*元空间回收耗时*/] [Times:/*耗时*/ user=0.00 /*用户耗时*/
sys=0.00 /*系统耗时*/, real=0.00 secs /*真是耗时*/]
Heap after GC invocations=3 (full 1):   //GC调用之前输出Heap详细信息、GC的第3次调用、第1次full gc
 def new generation   total 1856K, used 1024K [0x00000000ffa00000, 0x00000000ffc00000, 0x00000000ffc00000)
  eden space 1664K,  61% used [0x00000000ffa00000, 0x00000000ffb00010, 0x00000000ffba0000)
  from space 192K,   0% used [0x00000000ffba0000, 0x00000000ffba0000, 0x00000000ffbd0000)
  to   space 192K,   0% used [0x00000000ffbd0000, 0x00000000ffbd0000, 0x00000000ffc00000)
 tenured generation   total 4096K, used 3590K [0x00000000ffc00000, 0x0000000100000000, 0x0000000100000000)
   the space 4096K,  87% used [0x00000000ffc00000, 0x00000000fff81900, 0x00000000fff81a00, 0x0000000100000000)
 Metaspace       used 2559K, capacity 4486K, committed 4864K, reserved 1056768K
  class space    used 279K, capacity 386K, committed 512K, reserved 1048576K
}
```

Serial收集器下的Major GC/Full GC日志标注

GC日志解析-Minor GC和Full GC日志 (简化模式)

参数： -Xmn2m -Xmx6M -XX:+PrintGCDetails -XX:+PrintGCDateStamps -XX:+UseSerialGC -XX:+PrintCommandLineFlags

```
2019-05-03T23:05:28.921+0800: [GC /*GC类型 Minor GC*/  (Allocation Failure) 2019-05-03T23:05:28.921+0800: [DefNew: 1056K->1056K(1856K), 0.0000331 secs]2019-05-03T23:05:28.921+0800: [Tenured: 3590K->3590K(4096K), 0.0040770 secs] 4647K->4614K(5952K), [Metaspace: 2559K->2559K(1056768K)], 0.0042253 secs] [Times: user=0.00 sys=0.00, real=0.00 secs]
2019-05-03T23:05:28.926+0800: [Full GC /*GC类型 Full GC*/ (Allocation Failure) 2019-05-03T23:05:28.926+0800: [Tenured: 3590K->3576K(4096K), 0.0026480 secs] 4614K->4600K(5952K), [Metaspace: 2559K->2559K(1056768K)], 0.0026900 secs] [Times: user=0.00 sys=0.00, real=0.00 secs]
```

Serial收集器下的Major GC/Full GC日志标注

GC日志解析-ParNew+CMS+Serial Old

参数： -Xmn2m -Xmx6M -XX:+PrintGCDetails -XX:+PrintGCDateStamps -XX:+PrintHeapAtGC -XX:+UseParNewGC -XX:+UseConcMarkSweepGC -XX:+PrintCommandLineFlags

```
{Heap before GC invocations=0 (full 0):
 par new generation /*ParNew收集器下的新生代*/  total 1856K, used 725K [0x00000000ffa00000, 0x00000000ffc00000, 0x00000000ffc00000)
  eden space 1664K,  43% used [0x00000000ffa00000, 0x00000000ffab5620, 0x00000000ffba0000)
  from space 192K,   0% used [0x00000000ffba0000, 0x00000000ffba0000, 0x00000000ffbd0000)
  to   space 192K,   0% used [0x00000000ffbd0000, 0x00000000ffbd0000, 0x00000000ffc00000)
 concurrent mark-sweep generation /*CMS收集器下的老年代*/ total 4096K, used 0K [0x00000000ffc00000, 0x0000000100000000, 0x0000000100000000)
 Metaspace       used 2653K, capacity 4486K, committed 4864K, reserved 1056768K
  class space    used 285K, capacity 386K, committed 512K, reserved 1048576K
2019-05-05T13:19:41.076+0800: [GC (Allocation Failure) 2019-05-05T13:19:41.076+0800: [ParNew: /*ParNew收集器下的新生代*/ 725K->192K(1856K), 0.0092494 secs]
725K->548K(5952K), 0.0094504 secs] [Times: user=0.00 sys=0.00, real=0.01 secs]
Heap after GC invocations=1 (full 0):
 par new generation total 1856K, used 192K [0x00000000ffa00000, 0x00000000ffc00000, 0x00000000ffc00000)
  eden space 1664K,   0% used [0x00000000ffa00000, 0x00000000ffa00000, 0x00000000ffba0000)
  from space 192K, 100% used [0x00000000ffbd0000, 0x00000000ffc00000, 0x00000000ffc00000)
  to   space 192K,   0% used [0x00000000ffba0000, 0x00000000ffba0000, 0x00000000ffbd0000)
 concurrent mark-sweep generation total 4096K, used 356K [0x00000000ffc00000, 0x0000000100000000, 0x0000000100000000)
 Metaspace       used 2653K, capacity 4486K, committed 4864K, reserved 1056768K
  class space    used 285K, capacity 386K, committed 512K, reserved 1048576K
}
```

ParNew+CMS+Serial Old收集器下的Minor GC日志标注

GC日志解析-ParNew+CMS+Serial Old

参数： -Xmn2m -Xmx6M -XX:+PrintGCDetails -XX:+PrintGCDateStamps -XX:+PrintHeapAtGC -XX:+UseParNewGC -XX:+UseConcMarkSweepGC -XX:+PrintCommandLineFlags

```
2019-05-05T13:19:41.097+0800: [GC (Allocation Failure) 2019-05-05T13:19:41.097+0800: [ParNew:/*ParNew收集器下的新生代*/ 1057K->1057K(1856K), 0.0000499 secs]
2019-05-05T13:19:41.097+0800: [CMS: /*CMS收集器下的老年代*/ 3619K->3610K(4096K), 0.0071276 secs] 4676K->4634K(5952K), [Metaspace: 2653K->2653K(1056768K)],
0.0072904 secs] [Times: user=0.00 sys=0.00, real=0.01 secs]
Heap after GC invocations=3 (full 1):
 par new generation   total 1856K, used 1024K [0x00000000ffa00000, 0x00000000ffc00000, 0x00000000ffc00000)
  eden space 1664K,  61% used [0x00000000ffa00000, 0x00000000ffb00010, 0x00000000ffba0000)
  from space 192K,   0% used [0x00000000ffba0000, 0x00000000ffba0000, 0x00000000ffbd0000)
  to   space 192K,   0% used [0x00000000ffbd0000, 0x00000000ffbd0000, 0x00000000ffc00000)
 concurrent mark-sweep generation total 4096K, used 3610K [0x00000000ffc00000, 0x0000000100000000, 0x0000000100000000)
 Metaspace       used 2653K, capacity 4486K, committed 4864K, reserved 1056768K
  class space    used 285K, capacity 386K, committed 512K, reserved 1048576K
}
{Heap before GC invocations=3 (full 1):
 par new generation   total 1856K, used 1024K [0x00000000ffa00000, 0x00000000ffc00000, 0x00000000ffc00000)
  eden space 1664K,  61% used [0x00000000ffa00000, 0x00000000ffb00010, 0x00000000ffba0000)
  from space 192K,   0% used [0x00000000ffba0000, 0x00000000ffba0000, 0x00000000ffbd0000)
  to   space 192K,   0% used [0x00000000ffbd0000, 0x00000000ffbd0000, 0x00000000ffc00000)
 concurrent mark-sweep generation total 4096K, used 3610K [0x00000000ffc00000, 0x0000000100000000, 0x0000000100000000)
 Metaspace       used 2653K, capacity 4486K, committed 4864K, reserved 1056768K
  class space    used 285K, capacity 386K, committed 512K, reserved 1048576K
2019-05-05T13:19:41.105+0800: [Full GC (Allocation Failure) 2019-05-05T13:19:41.105+0800: [CMS: /*CMS收集器下的老年代*/ 3610K->3597K(4096K), 0.0062776 secs]
4634K->4621K(5952K), [Metaspace: 2653K->2653K(1056768K)], 0.0063639 secs] [Times: user=0.00 sys=0.00, real=0.01 secs]
```

ParNew+CMS+Serial Old收集器下的Full GC日志标注

GC日志解析-Parallel+Parallel Old

参数： -Xmn2m -Xmx6M -XX:+PrintGCDetails -XX:+PrintGCDateStamps -XX:+PrintHeapAtGC -XX:+UseParallelGC -XX:+PrintCommandLineFlags

```
{Heap before GC invocations=1 (full 0):
 PSYoungGen /*Parallel收集器下的新生代*/      total 1536K, used 737K [0x00000000ffe00000, 0x0000000100000000, 0x0000000100000000)
  eden space 1024K, 72% used [0x00000000ffe00000,0x00000000ffeb86d8,0x00000000fff00000)
  from space 512K, 0% used [0x00000000fff80000,0x00000000fff80000,0x0000000100000000)
  to   space 512K, 0% used [0x00000000fff00000,0x00000000fff00000,0x00000000fff80000)
 ParOldGen /*Parallel Old收集器下的新生代*/       total 4096K, used 2048K [0x00000000ffa00000, 0x00000000ffe00000, 0x00000000ffe00000)
  object space 4096K, 50% used [0x00000000ffa00000,0x00000000ffc00020,0x00000000ffe00000)
 Metaspace       used 2653K, capacity 4486K, committed 4864K, reserved 1056768K
  class space    used 285K, capacity 386K, committed 512K, reserved 1048576K
2019-05-05T13:50:23.961+0800: [GC (Allocation Failure) [PSYoungGen: /*Parallel收集器下的新生代*/ 737K->504K(1536K)] 2785K->2664K(5632K), 0.0847886 secs] [
Times: user=0.14 sys=0.00, real=0.09 secs]
Heap after GC invocations=1 (full 0):
 PSYoungGen      total 1536K, used 504K [0x00000000ffe00000, 0x0000000100000000, 0x0000000100000000)
  eden space 1024K, 0% used [0x00000000ffe00000,0x00000000ffe00000,0x00000000fff00000)
  from space 512K, 98% used [0x00000000fff00000,0x00000000fff7e010,0x00000000fff80000)
  to   space 512K, 0% used [0x00000000fff80000,0x00000000fff80000,0x0000000100000000)
 ParOldGen       total 4096K, used 2160K [0x00000000ffa00000, 0x00000000ffe00000, 0x00000000ffe00000)
  object space 4096K, 52% used [0x00000000ffa00000,0x00000000ffc1c020,0x00000000ffe00000)
 Metaspace       used 2653K, capacity 4486K, committed 4864K, reserved 1056768K
  class space    used 285K, capacity 386K, committed 512K, reserved 1048576K
}
```

Parallel+Parallel Old收集器下的Minor GC日志标注

GC日志解析-Parallel+Parallel Old

参数： -Xmn2m -Xmx6M -XX:+PrintGCDetails -XX:+PrintGCDateStamps -XX:+PrintHeapAtGC -XX:+UseParallelGC -XX:+PrintCommandLineFlags

```
{Heap before GC invocations=3 (full 1):
 PSYoungGen      total 1536K, used 504K [0x00000000ffe00000, 0x0000000100000000, 0x0000000100000000)
  eden space 1024K, 0% used [0x00000000ffe00000,0x00000000ffe00000,0x00000000fff00000)
  from space 512K, 98% used [0x00000000fff80000,0x00000000ffffe030,0x0000000100000000)
  to   space 512K, 0% used [0x00000000fff00000,0x00000000fff00000,0x00000000fff80000)
 ParOldGen       total 4096K, used 2160K [0x00000000ffa00000, 0x00000000ffe00000, 0x00000000ffe00000)
  object space 4096K, 52% used [0x00000000ffa00000,0x00000000ffc1c020,0x00000000ffe00000)
 Metaspace       used 2653K, capacity 4486K, committed 4864K, reserved 1056768K
  class space    used 285K, capacity 386K, committed 512K, reserved 1048576K
2019-05-05T13:50:24.050+0800: [Full GC (Allocation Failure) [PSYoungGen: /*Parallel收集器下的新生代*/ 504K->0K(1536K)] [ParOldGen: /*Parallel
Old收集器下的老年代*/ 2160K->2582K(4096K)] 2664K->2582K(5632K), [Metaspace: 2653K->2653K(1056768K)], 0.0862106 secs] [Times: user=0.16 sys=0.00, real=0.09
secs]
Heap after GC invocations=3 (full 1):
 PSYoungGen      total 1536K, used 0K [0x00000000ffe00000, 0x0000000100000000, 0x0000000100000000)
  eden space 1024K, 0% used [0x00000000ffe00000,0x00000000ffe00000,0x00000000fff00000)
  from space 512K, 0% used [0x00000000fff80000,0x00000000fff80000,0x0000000100000000)
  to   space 512K, 0% used [0x00000000fff00000,0x00000000fff00000,0x00000000fff80000)
 ParOldGen       total 4096K, used 2582K [0x00000000ffa00000, 0x00000000ffe00000, 0x00000000ffe00000)
  object space 4096K, 63% used [0x00000000ffa00000,0x00000000ffc85818,0x00000000ffe00000)
 Metaspace       used 2653K, capacity 4486K, committed 4864K, reserved 1056768K
  class space    used 285K, capacity 386K, committed 512K, reserved 1048576K
}
```

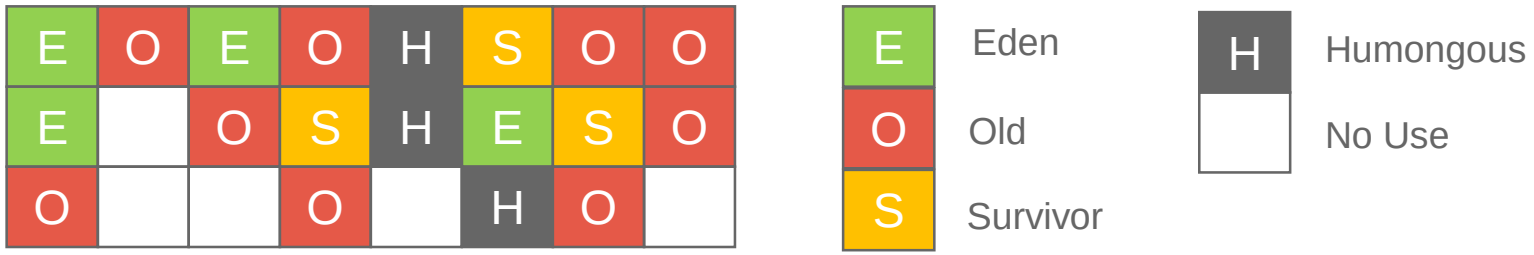
Parallel+Parallel Old收集器下的Full GC日志标注

GC日志解析-差异汇总

收集器组合	新生代	老年代
Serial+Serial Old	DefNew、def new	tenured
ParNew+CMS	ParNew、par new	CMS、concurrent mark-sweep generation
Parallel+Parallel Old	PSYoungGen	ParOldGen

G1垃圾收集器核心-Region

- G1-Garbage First ,专注于垃圾最多的分区，花费较少的时间就能回收较多的垃圾。由于这种特性因此称为G1



Region

- G1收集器将堆内存划分为一系列大小相等的Region区域，Region大小在1MB到32MB在启动时由JVM自己确定-XX:G1HeapRegionSize=n，可以使用这个参数设置每个Region的大小，这里需要是1MB到32MB的2的指数的大小。
- 分为Eden, Survivior, Old，Humongous区，逻辑连续但是无上不连续；一个Old Region收集完成后可能变成一个Eden Region区域。

GC日志解析-G1

参数： -Xmx6M -XX:+PrintGCDetails -XX:+PrintGCDateStamps -XX:+PrintHeapAtGC -XX:+UseG1GC -XX:+PrintCommandLineFlags

```
{Heap before GC invocations=0 (full 0):
 garbage-first heap /*G1堆大小*/  total 6144K, used 1024K [0x00000000ffa00000, 0x00000000ffb00030, 0x0000000100000000)
  region size 1024K /*单个region大小*/, 1 young (1024K) /*1个young region,总大小1024*/, 0 survivors (0K) /*0个survivors region,总大小0*/
 Metaspace       used 2653K, capacity 4486K, committed 4864K, reserved 1056768K
  class space    used 285K, capacity 386K, committed 512K, reserved 1048576K
2019-05-05T14:30:47.729+0800: [GC pause /*GC停顿*/ (G1 Humongous Allocation /*原因是大对象分配*/) (young /*涉及到整理的是yong区*/) (initial-mark /*初始标记*/
), 0.0048533 secs]
   [Parallel Time: 4.1 ms, GC Workers: 4] /*GC并行时长、GC工作线程数量*/
      [GC Worker Start (ms): Min: 436.2, Avg: 436.5, Max: 437.1, Diff: 0.9] /*每一个工作者线程的以毫秒为单位的启动时间,436.2代表第436.2毫秒启动*/
      [Ext Root Scanning (ms): Min: 0.0, Avg: 1.8, Max: 3.3, Diff: 3.2, Sum: 7.3] /*每一个扫描根的工作线程花费的时间*/
      [Update RS (ms): Min: 0.0, Avg: 0.0, Max: 0.0, Diff: 0.0, Sum: 0.0] /*每一个线程更新Remembered Sets花费的时间。Remembered
      Sets是保存到堆中的Region的跟踪引用。我们保存这些改变的跟踪信息到叫作Update Buffers的更新缓存中。Update RS 子任务不能并发的处理更新缓存*/
         [Processed Buffers: Min: 0, Avg: 0.0, Max: 0, Diff: 0, Sum: 0] /*每一个线程处理的Update Buffers(上面提到的)的数量*/
      [Scan RS (ms): Min: 0.0, Avg: 0.0, Max: 0.0, Diff: 0.0, Sum: 0.0] /*每一个工作线程扫描Remembered
      Sets花费的时间，扫描CSet中Region的RSet，避免了扫描整个老年代*/
      [Code Root Scanning (ms): Min: 0.0, Avg: 0.0, Max: 0.0, Diff: 0.0, Sum: 0.0] /*扫描code root耗时。Code Root是JIT编译后的代码里引用了heap中的对象。*/
      [Object Copy (ms): Min: 0.4, Avg: 1.8, Max: 3.0, Diff: 2.6, Sum: 7.0] /*拷贝存活对象到新的Region*/
      [Termination (ms): Min: 0.0, Avg: 0.1, Max: 0.1, Diff: 0.1, Sum: 0.4]
      /*当GC线程完成任务之后尝试结束到真正结束耗时。因为在结束前他会检查其他线程是否有未完成的任务，帮助完成之后再结束。*/
         [Termination Attempts: Min: 1, Avg: 1.0, Max: 1, Diff: 0, Sum: 4]
      [GC Worker Other (ms): Min: 0.1, Avg: 0.1, Max: 0.1, Diff: 0.1, Sum: 0.4] /*花费在其他工作上的时间*/
      [GC Worker Total (ms): Min: 3.2, Avg: 3.8, Max: 4.0, Diff: 0.9, Sum: 15.2] /*花费总时长*/
      [GC Worker End (ms): Min: 440.2, Avg: 440.2, Max: 440.2, Diff: 0.0]
      /*每个线程的结束时间。最小|最大时间戳表示第一个线程和最后一个线程的结束时间。理想情况下同时结束。*/
   [Code Root Fixup: 0.0 ms] /*修复GC期间code root指针改变的耗时*/
   [Code Root Purge: 0.0 ms] /*清除code root耗时*/
   [Clear CT: 0.1 ms] /*清除card tables 中的dirty card的耗时*/
   [Other: 0.6 ms]
      [Choose CSet: 0.0 ms] /*选择进行垃圾回收的Region集合时消耗的时间. 通常很小,在扫描old区时时间稍长*/
      [Ref Proc: 0.4 ms] /*处理soft, weak等引用的耗时*/
      [Ref Enq: 0.0 ms] /*soft, weak等引用放入待处理列表花费的时长*/
      [Redirty Cards: 0.1 ms] /*重新标记dirty card*/
      [Humongous Register: 0.0 ms] /*Humongous注册*/
      [Humongous Reclaim: 0.0 ms] /*Humongou回收*/
      [Free CSet: 0.0 ms] /*释放刚被垃圾收集的heap区所消耗的时间,包括对应的RS*/
   [Eden: /*Eden区回收前已使用大小/总大小*/1024.0K(2048.0K)->0.0B(1024.0K) /*Eden区回收后已使用大小/总大小*/ Survivors: /*幸存区回收前大小*/0.0B->4.0K
   /*幸存区回收后大小*/ Heap: /*堆回收前已使用大小/总大小*/1758.2K(6144.0K)->1672.1K(6144.0K)/*堆回收后已使用大小/总大小*/]
```

G1日志.t