云析学院 2018

JAVA虚拟机及调优

讲师：擎天

目录

Contents

01 基础篇

Java类文件结构

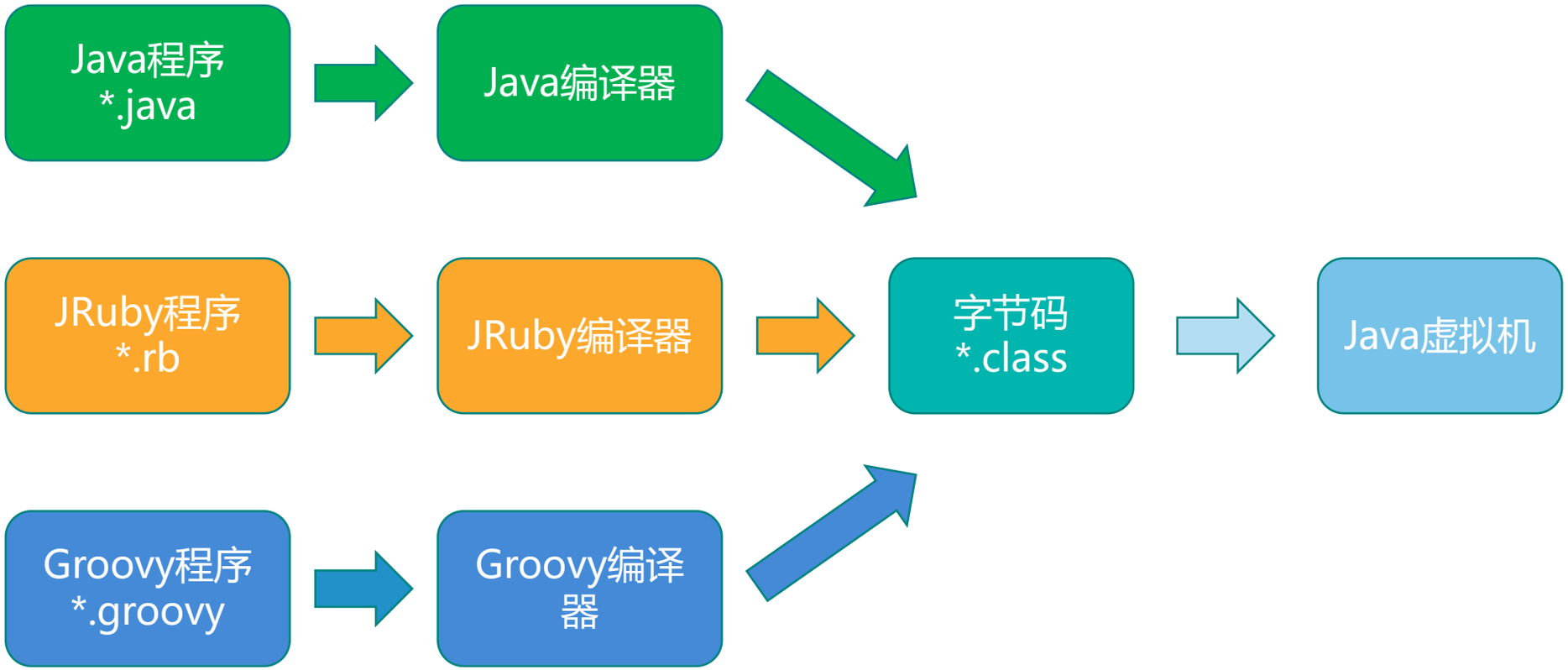
Java虚拟机类加载机制

Java虚拟机运行时数据区

Java垃圾回收策略

其他语言与Java虚拟机

问题：其他语言是否可以使用Java虚拟机作为产品交付媒介



Class文件组成内容

Class文件采用类似于C语言结构体的伪结构体来存储数据

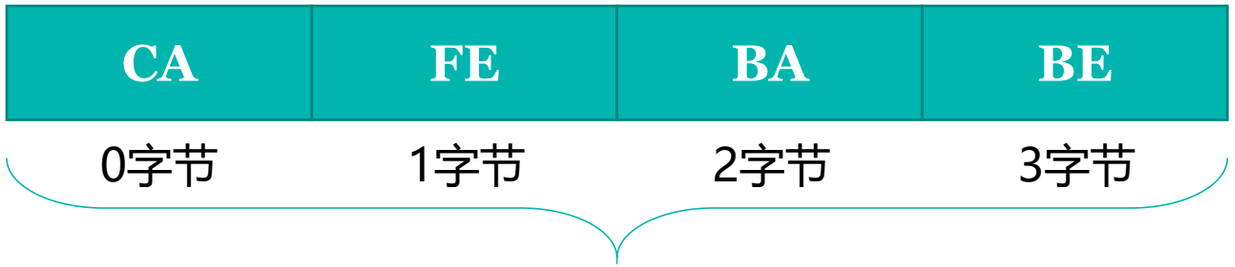
Class文件格式

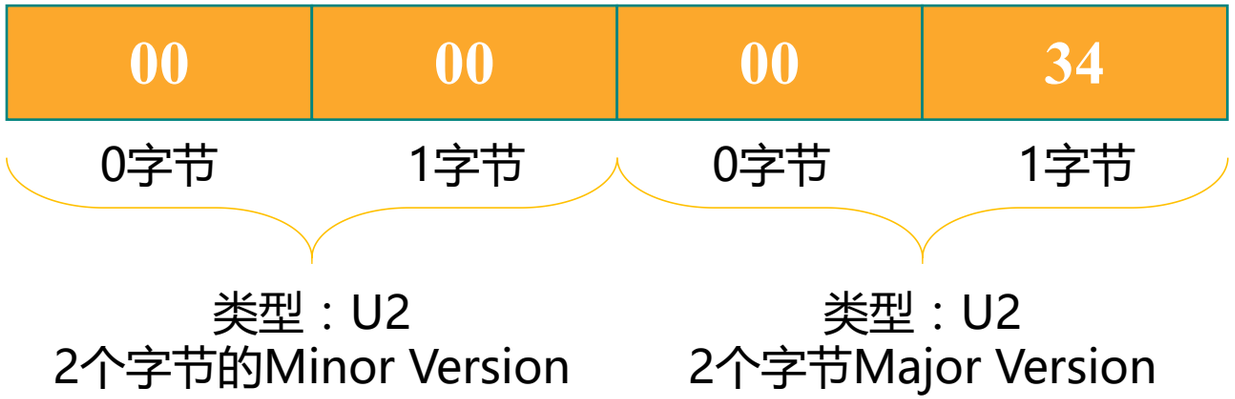
u4	u2	u2	u2	cp_info	u2	u2	u2	u2	u2	u2	field_info	u2	method_info	u2	attribute_info

魔数	u4	magic	1
次版本号	u2	minor_version	1
主版本号	u2	major_version	1
常量池计数器	u2	constant_pool_count	1
常量池	cp_info	constant_pool	constant_pool_count-1
访问标志	u2	access_flags	1
类索引	u2	this_class	1
父类索引	u2	super_class	1
接口计数器	u2	interfaces_class	1
接口索引集合	u2	interfaces	interfaces_count
字段计数器	u2	fields_count	1
字段表集合	field_info	fields	fields_count
方法计数器	u2	methods_count	1
方法表集合	method_info	methods	methods_count
属性计数器	u2	attributes_count	1
属性表集合	attribute_info	attributes	attributes_count

Class魔数和版本

每个Class文件的头4个字节成为魔数（Magic Number）,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。值为：0xCAFEBABE（咖啡宝贝）





紧接魔数的4个字节是Class文件的版本号：

第5-6字节是次版本号（Minor Version）

第7-8字节是主版本号（Major Version）

Class文件结构-常量池

Class文件常量池

常量池代表Class文件中的仓库资源
紧接着主次版本号之后就是常量池入口

常量池主要存放两大类常量

■字面量　■符号引用

字面量接近Java语言层面的常量概念，如
文本字符串、声明为final的常量值等

符号引用包含三类常量：

类和接口的全限定名

org.springframework.....Bean

字段的名称和描述符

private/public/protected

方法的名称和描述符

private/public/protected

Class文件结构-常量池

类型	标志	描述
CONSTANT_Utf8_info	1	UTF-8编码字符串
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	标识方法句柄
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

常量池结构表-1

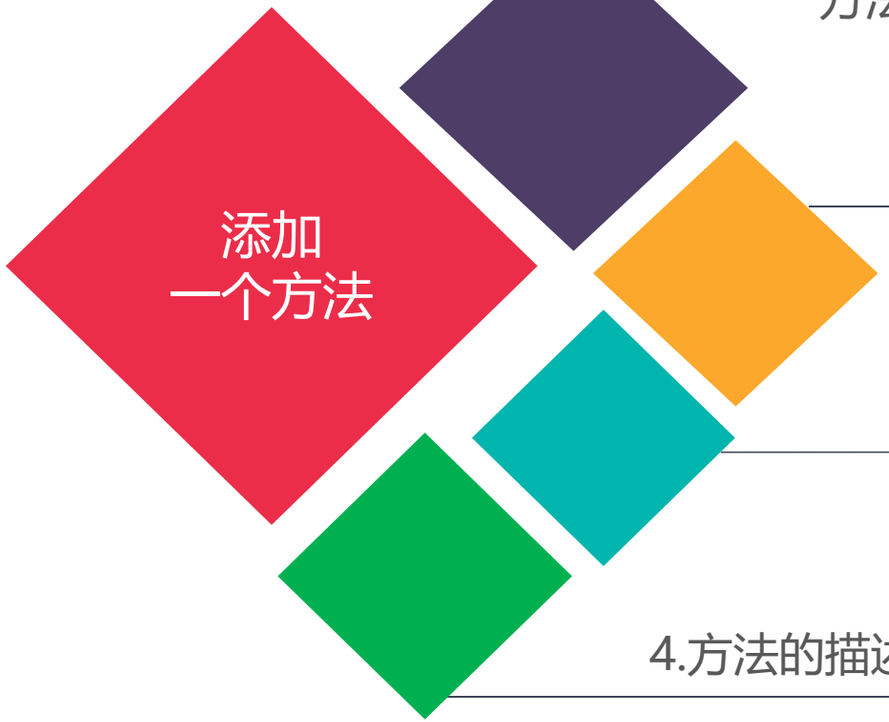
常量	项目	字段类型	常量描述
CONSTANT_utf8_info	tag	u1	值：1
	length	u2	UTF-8编码的字符串占用字节数
	bytes	u1	长度为length的UTF-8编码的字节数
CONSTANT_Integer_info	tag	u1	值：3
	bytes	u4	按照高位在前存储的int值
CONSTANT_Float_info	tag	u1	值：4
	bytes	u4	按照高位在前存储的float值
CONSTANT_Long_info	tag	u1	值：5
	bytes	u8	按照高位在前存储的long值
CONSTANT_Double_info	tag	u1	值：6
	bytes	u8	按照高位在前存储的double值
CONSTANT_Class_info	tag	u1	值：7
	index	u2	指向全限定名常量项的索引
CONSTANT_String_info	tag	u1	值：8
	index	u2	指向字符串字面量的索引
CONSTANT_Fieldref_info	tag	u1	值：9
	index	u2	指向声明字段的类或接口描述符CONSTANT_Class_info的索引项
	index	u2	指向字段描述符CONSTANT_NameAndType的索引项

常量池结构表-2

常量	项目	字段类型	常量描述
CONSTANT_Methodref_info	tag	u1	值：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：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
CONSTANT_NameAndType_info	tag	u1	值：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：15
	reference_kind	u2	值必须是1~9，决定方法类型。该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：16
	descriptor_index	u2	值必须是对常量池的有效索引，常量池在该索引的项必须是CONSTANT_Utf8_info结构，表示方法的描述符
CONSTANT_Invoke-Dynamic_info	tag	u1	值：18
	bootstrap_method_attr_index	u2	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	当前常量池的有效索引，常量池在该索引的项必须是CONSTANT_NameAndType_info结构，表示方法名和方法描述符

常量池结构表-说明

添加一个方法时，常量池中会增加4个常量；同理，添加字段也是如此



1.CONSTANT_Methodref_info

方法的符号引用

2.方法符号引用指向的CONSTANT_NameAndType_info

方法的部分符号引用

3.方法的名称

4.方法的描述符

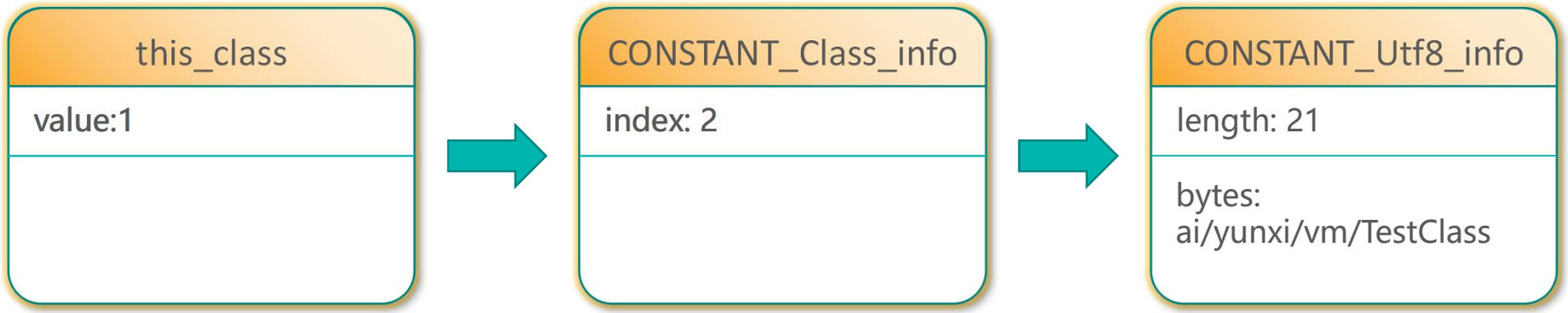
访问标志

紧接着常量池之后的两个字节代表访问标志（access_flags），用于识别一些类或者接口层次的访问信息，包括：这个Class是类还是接口、是否为public类型、是否为abstract类型、类是否声明为final等。标志位及其含义如下表：

标志名称	值	说明
ACC_PUBLIC	0X0001	public类型
ACC_FINAL	0X0010	声明为final，只有类可以设置
ACC_SUPER	0X0020	使用invokespecial字节码指令的新语意，invokespecial指令的语意在JDK1.0.2发生过改变，为了区别这条指令使用哪种语意，JDK1.0.2之后编译出来的类都为真
ACC_INTERFACE	0X0200	接口
ACC_ABSTRACT	0X0400	abstract类型，对于接口或者抽象类来说，此标志值为真，其他类为假
ACC_SYNTHETIC	0X1000	这个类并非由用户代码产生
ACC_ANNOTATION	0X2000	注解
ACC_ENUM	0X4000	枚举

类索引、父类索引与接口索引集合

访问标志之后顺序排列类索引(this)、父类索引(super)、接口索引集合(interfaces)。
Class文件由这三项来确定这个类的集成关系。



类索引和父类索引都是u2类型的数据。
接口索引集合入口第一项是u2类型的接口计数器（interfaces_count）表示索引表的容量（即实现了几个接口）。如果该类没用实现任何接口，则计数器值为0，后面的接口索引表不再占用任何字节。

字段表集合

接口索引集合后边的是字段计数器：用于标识有多少个字段，接着就是字段表集合。

字段表（field_info）用于描述接口或者类中声明的变量。

字段包括类级变量以及实例级变量。可以包括的信息有：

- 字段的作用域（public、private、protected修饰符）
- 实例变量还是类变量（static修饰符）
- 可变性（final）
- 并发可见性（volatile）
- 可否被序列化（transient）
- 字段数据类型（基本类型，对象，数组）
- 字段名称

字段表结构和访问标志

字段表结构

类型	名称	数量
u2	access_flags	1
u2	name_index	1
u2	descriptor_index	1
u2	attribute_count	1
attribute_info	attributes	attribute_count

各个修饰符可以使用标志位表示

但字段叫什么名字、字段被定义成什么类型，都是无法固定的，所以只能引用常量池中的常量来描述

字段访问标志

标志名称	标志值	含义
ACC_PUBLIC	0X0001	字段是否public
ACC_PRIVATE	0X0002	字段是否private
ACC_PROTECTED	0X0004	字段是否protected
ACC_STATIC	0X0008	字段是否static
ACC_FINAL	0X0010	字段是否final
ACC_VOLATILE	0X0040	字段是否volatile
ACC_TRANSIENT	0X0080	字段是否transient
ACC_SYNTHETIC	0X0100	字段是否由编译器自动产生的
ACC_ENUM	0X0400	字段是否enum

全限定名、简单名称及描述符

- **全限定名**

ai/yunxi/vm/TestClasss类的全限定名

仅仅是把类中的“.”替换成了“/”

- **简单名称**

没有类型和参数修饰的方法或者字段名称

如：inc()和int m简单名称就是：inc、m

- **描述符**

用来描述字段的数据类型、方法的参数列表（数量、类型及顺序）和返回值

描述符标识字符含义

标识字符	含义m
B	基本类型byte
C	基本类型char
D	基本类型double
F	基本类型float
I	基本类型int
J	基本类型long
S	基本类型short
Z	基本类型boolean
V	特殊类型void
L	对象类型，如Ljava/lang/Object

方法表集合

通过访问标志、名称索引、描述符索引可清楚的表达方法的定义

属性表是Class文件格式中最具扩展性的一种数据项目

标志名称	标志值	含义
ACC_PUBLIC	0X0001	方法是否public
ACC_PRIVATE	0X0002	方法是否private
ACC_PROTECTED	0X0004	方法是否protected
ACC_STATIC	0X0008	方法是否static
ACC_FINAL	0X0010	方法是否final
ACC_SYNCHRONIZED	0X0020	方法是否synchronized
ACC_BRIDGE	0X0040	方法是否由编译器产生的桥接方法
ACC_VARARGS	0X0080	方法是否接受不定参数
ACC_NATIVE	0X0100	方法是否为native
ACC_ABSTRACT	0X0400	方法是否为abstract
ACC_STRICTFP	0X0800	方法是否为strictfp
ACC_SYNTHETIC	0X1000	防范是否由编译器自动产生

方法表的数据结构

Class文件存储格式中对方法的描述与对字段的描述几乎采用完全一致的方式。

重载（Overload）一个方法，需要

- 要与原方法具有相同的简单名称
- 要与原方法有不同的特征签名

属性表结构和虚拟机规范预定义的属性

Class文件、字段表、方法表、属性表都可以携带自己的属性表集合，用于描述某些场景专有的信息。属性表集合的限制稍微宽松，不再要求各个属性表具有严格顺序，只要不与已有属性名重复，任何人实现的编译器都可以向属性表中写入自己定义的属性信息

类型	名称	数量
u2	attribute_name_index	1
u4	attribute_length	1
u1	info	attribute_length

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、方法表、字段表	被声明为deprecated的方法和字段
Exceptions	方法表	方法抛出的异常
EnclosingMethod	类文件	仅当一个类为局部类或者匿名类时才能拥有这个属性，这个属性用于标识这个类所在的外围方法
InnerClasses	类文件	内部类列表
LineNumberTable	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
StackMapTable	Code属性	JDK1.6中新增的属性，供新的类型检查验证器（Type Checker）检查和处理目标方法的局部变量和操作数栈所需要的类型是否匹配
SourceFile	类文件	记录源文件名称

虚拟机规范预定义的属性

属性名称	使用位置	含义
Signature	类、方法表、字段表	JDK1.5中新增的属性，这个属性用于支持泛型情况下的方法签名，在Java语言中，任何类、接口、初始化方法或成员的泛型签名如果包含了类型变量（Type Variables）或参数化类型（Parameterized Types），则Signature属性会为它记录泛型签名信息。由于Java的泛型采用擦除法实现，在为了避免类型信息被擦除后导致签名混乱，需要这个属性记录泛型中的相关信息
SourceDebugExtension	类文件	JDK1.6中新增的属性，SourceDebugExtension属性用于存储额外的调试信息。譬如在进行JSP文件调试时，无法通过Java堆栈来定位JSP文件的行号，JSR-45规范为这些非Java语言编写，却需要编译成字节码并运行在Java虚拟机中的程序提供了一个进行调试的标准机制，使用SourceDebugExtension属性就可以用于存储这个标准所新加入的调试信息
Synthetic	类、方法表、字段表	标识方法或字段为编译器自动生成的
LocalVariableTypeTable	类	JDK1.5中新增的属性，它使用特征签名代替描述符，是为了引入泛型语法之后能描述泛型参数化类型而添加
RuntimeVisibleAnnotations	类、方法表、字段表	JDK1.5新增的属性，为动态注解提供支持。RuntimeVisibleAnnotations属性用于注明哪些注解是运行时（实际上运行时就是进行反射调用）可见的
RuntimeInvisibleAnnotations	类、方法表、字段表	JDK1.5新增的属性，与RuntimeVisibleAnnotations属性作用刚好相反，用于指明哪些注解是运行时不可见的
RuntimeVisibleParameterAnnotations	方法表	JDK1.5新增的属性，作用与RuntimeVisibleAnnotations属性类似，只不过作用对象为方法参数
RuntimeInvisibleParameterAnnotations	方法表	JDK1.5新增的属性，作用与RuntimeInvisibleAnnotations属性类似，只不过作用对象为方法参数
AnnotationDefault	方法表	JDK1.5新增的属性，用于记录注解类元素的默认值
BootstrapMethods	类文件	JDK1.7中新增的属性，用于保存invokedynamic指令引用的引导方法限定符

属性表集合之Code属性

Java程序方法体中的代码经过Javac编译处理后，最终变为字节码指令存储在Code属性中，Code属性出现在方法表的属性集合之中。但并非所有方法表都有Code属性，例如抽象类或接口。

attribute_name_index指向CONSTANT_Utf8_info类型常量的值固定为“Code”

attribute_length标识属性值的总长度

max_stack代表了操作数栈（Operand Stacks）深度的最大值

max_locals代表了局部变量所表示的存储空间单位：Slot

code_length和code是用来存储Java源程序编译后产生的字节码指令

Code属性表的结构

类型	名称	数量
u2	attribute_name_index	1
u4	attribute_length	1
u2	max_stack	1
u2	max_locals	1
u4	code_length	1
u1	code	code_length
u2	exception_table_length	1
exception_info	exception_table	exception_table_length
u2	attribute_count	
attribute_info	attributes	attribute_count

属性表集合之异常表的结构

```
Code:
  stack=1, locals=5, args_size=1
     0: iconst_1
     1: istore_1
     2: iload_1
     3: istore_2
     4: iconst_3
     5: istore_1
     6: iload_2
     7: ireturn
     8: astore_2
     9: iconst_2
    10: istore_1
    11: iload_1
    12: istore_3
    13: iconst_3
    14: istore_1
    15: iload_3
    16: ireturn
    17: astore        4
    19: iconst_3
    20: istore_1
    21: aload         4
    23: athrow
  Exception table:
     from    to  target type
         0     4     8   Class java/lang/Exception
         0     4    17   any
         8    13    17   any
        17    19    17   any
```

- 字节码0-4行所做的操作数就是将整数1赋值给变量x
- 如果这时没有出现异常，则会继续走到第5-7行，将变量x赋值为3
- 如果出现了异常，PC寄存器指针转到第8行，第9-16行所做的事情是将2赋值给变量x，然后异常抛出，方法结束。
- 如果0-4行出现任何异常，则跳转17行
- 如果8-13行出现任何异常，则跳转17行
- 如果17-19行出现任何异常，则跳转17行

名称	类型	数量
start_pc	u2	1
end_pc	u2	1
handle_pc	u2	1
catch_type	u2	1

属性表集合之Exceptions、LineNumberTable

Exceptions属性是在方法表中与Code属性平级的一项属性。Exceptions属性的作用是列举出方法中可能抛出的受查异常（Checked Exceptions），也就是方法描述时在throws关键字后面列举的异常

名称	类型	数量
attribute_name_index	u2	1
attribute_length	u4	1
number_of_exceptions	u2	1
exception_index_table	u2	number_of_exceptions

LineNumberTable属性用于描述Java源码行号与字节码行号（字节码的偏移量）之间的对应关系。可以在编译的时候分别使用-g:none和-g:lines选项来取消或者要求生成这项信息。

名称	类型	数量
attribute_name_index	u2	1
attribute_length	u4	1
line_number_table_length	u2	1
line_number_table	line_number_info	line_number_table_length

属性表集合之LocalVariableTable

LocalVariableTable属性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。

名称	类型	数量
attribute_name_index	u2	1
attribute_length	u4	1
local_varible_table_length	u2	1
local_variable_table	local_variable_info	local_varible_table_length

start_pc和length属性分别代表了这个局部变量的生命周期开始的字节码偏移量及其作用范围覆盖的长度，两者结合起来就是这个局部变量在字节码之中的作用域范围。
name_index和descriptor_index都是指向常量池中CONSTANT_Utf8_info型常量的索引，分别代表了局部变量的名称及这个局部变量的描述符。
index是这个局部变量在栈帧局部变量表中Slot的位置。当这个变量数据类型是64位类型时（double和long），它占用的Slot为index和index+1两个

名称	类型	数量
start_pc	u2	1
length	u2	1
name_index	u2	1
descriptor_index	u2	1
index	u2	1

属性表集合之LocalVariableTable

LocalVariableTable属性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。

名称	类型	数量
attribute_name_index	u2	1
attribute_length	u4	1
local_varible_table_length	u2	1
local_variable_table	local_variable_info	local_varible_table_length

start_pc和length属性分别代表了这个局部变量的生命周期开始的字节码偏移量及其作用范围覆盖的长度，两者结合起来就是这个局部变量在字节码之中的作用域范围。
name_index和descriptor_index都是指向常量池中CONSTANT_Utf8_info型常量的索引，分别代表了局部变量的名称及这个局部变量的描述符。
index是这个局部变量在栈帧局部变量表中Slot的位置。当这个变量数据类型是64位类型时（double和long），它占用的Slot为index和index+1两个

名称	类型	数量
start_pc	u2	1
length	u2	1
name_index	u2	1
descriptor_index	u2	1
index	u2	1

属性表集合之SourceFile、ConstantValue

SourceFile属性用于记录生成这个Class文件的源码文件名称。sourcefile_index数据项是指向常量池中CONSTANT_Utf8_info型常量的索引，常量值是源码文件的文件名。

名称	类型	数量
attribute_name_index	u2	1
attribute_length	u4	1
sourcefile_index	u2	1

ConstantValue属性的作用是通知虚拟机自动为静态变量赋值。只有被static关键字修饰的常量（类变量）才可以使用这项属性。
目前Sun Javac编译器的选择是：如果同时使用final和static来修饰一个变量，并且这个变量的数据类型是基本类型或者java.lang.String的话，就生成ConstantValue属性来进行初始化，如果这个变量没有被final修饰，或者并非基本类型及字符串，则将会选择在<clinit>方法中进行初始化。

名称	类型	数量
attribute_name_index	u2	1
attribute_length	u4	1
constantvalue_index	u2	1

属性表集合之InnerClasses属性

InnerClasses属性用于记录内部类与宿主类之间的关联

名称	类型	数量
attribute_name_index	u2	1
attribute_length	u4	1
number_of_classes	u2	1
inner_classes	inner_classes_info	number_of_classes

number_of_classes代表需要记录多少个内部类信息。

inner_classes_info表的结构

名称	类型	数量
inner_class_info_index	u2	1
outer_class_info_index	u2	1
inner_name_index	u2	1
inner_class_access_flags	u2	1

标志名称	标志值	含义
ACC_PUBLIC	0X0001	内部类是否public
ACC_PRIVATE	0X0002	内部类是否private
ACC_PROTECTED	0X0004	内部类是否protected
ACC_STATIC	0X0008	内部类是否static
ACC_FINAL	0X0010	内部类是否final
ACC_INTERFACE	0X0040	内部类是否interface
ACC_ABSTRACT	0X0080	内部类是否abstract
ACC_SYNTHETIC	0X0100	内部类是否非用户代码产生
ACC_ANNOTATION	0X0400	内部类是否是一个注解
ACC_ENUM	0X0800	内部类是否是一个枚举

- inner_class_info_index和outer_class_info_index都是指向常量池中CONSTANT_Class_info型常量的索引，分别代表了内部类和宿主类的符号引用。
- inner_name_index代表内部类的名称
- inner_class_access_flags是内部类的访问标志

属性表集合之Deprecated和Synthetic、StackMapTable

Deprecated和Synthetic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代表已经不再推荐使用。Synthetic代表字段或者方法并不是有Java源码直接产生的，而是由编译器自行添加的。

名称	类型	数量
attribute_name_index	u2	1
attribute_length	u2	1

StackMapTable属性在JDK1.6发布后增加到了Class文件规范中，它是一个复杂的变长属性，位于Code属性的属性表中。会在虚拟机类加载的字节码验证阶段被新类型检查验证器（Type Checker）使用，目的在于代替以前比较消耗性能的基于数据流分析的类型推导验证器。一个方法的Code属性最多只能有一个StackMapTable属性。

名称	类型	数量
attribute_name_index	u2	1
attribute_length	u4	1
number_of_entries	u2	1
stack_map_frame	stack_map_frame entries	1

属性表集合之Singature、BoostrapMethod

InnerClasses属性用于记录内部类与宿主类之间的关联

名称	类型	数量
attribute_name_index	u2	1
attribute_length	u4	1
number_of_classes	u2	1
inner_classes	inner_classes_info	number_of_classes

number_of_classes代表需要记录多少个内部类信息。

inner_classes_info表的结构

名称	类型	数量
inner_class_info_index	u2	1
outer_class_info_index	u2	1
inner_name_index	u2	1
inner_class_access_flags	u2	1

标志名称	标志值	含义
ACC_PUBLIC	0X0001	内部类是否public
ACC_PRIVATE	0X0002	内部类是否private
ACC_PROTECTED	0X0004	内部类是否protected
ACC_STATIC	0X0008	内部类是否static
ACC_FINAL	0X0010	内部类是否final
ACC_INTERFACE	0X0040	内部类是否interface
ACC_ABSTRACT	0X0080	内部类是否abstract
ACC_SYNTHETIC	0X0100	内部类是否非用户代码产生
ACC_ANNOTATION	0X0400	内部类是否是一个注解
ACC_ENUM	0X0800	内部类是否是一个枚举

- inner_class_info_index和outer_class_info_index都是指向常量池中CONSTANT_Class_info型常量的索引，分别代表了内部类和宿主类的符号引用。
- inner_name_index代表内部类的名称
- inner_class_access_flags是内部类的访问标志

01 基础篇

Java类文件结构

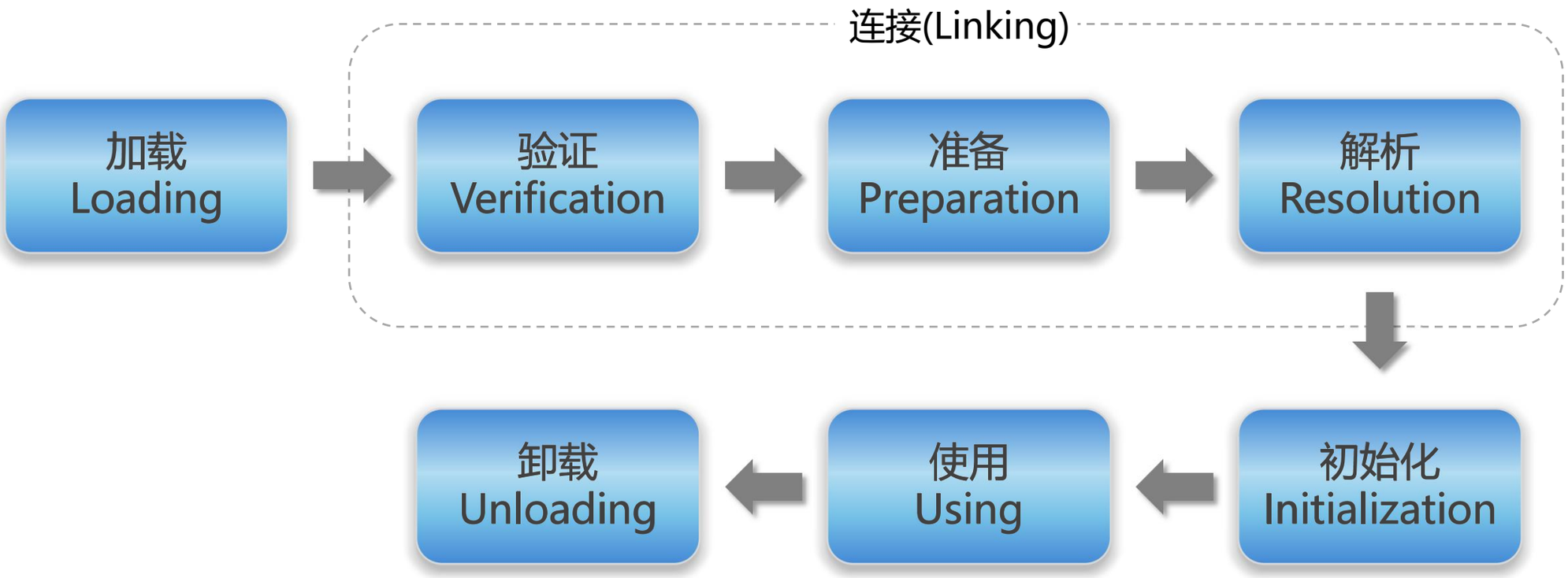
Java虚拟机类加载机制

Java虚拟机运行时数据区

Java垃圾回收策略

类加载机制

类从被加载到虚拟内存中开始，到卸载内存为止，它的整个生命周期包括：



类的加载过程必须按照加载、验证、准备、初始化和卸载的顺序开始，而解析阶段则不一定，它在某些情况下可以在初始化阶段之后开始

类的生命周期

Unsupported major.minor version 52.0
Unsupported major.minor version 51.0
Unsupported major.minor version 50.0
Unsupported major.minor version 49.0

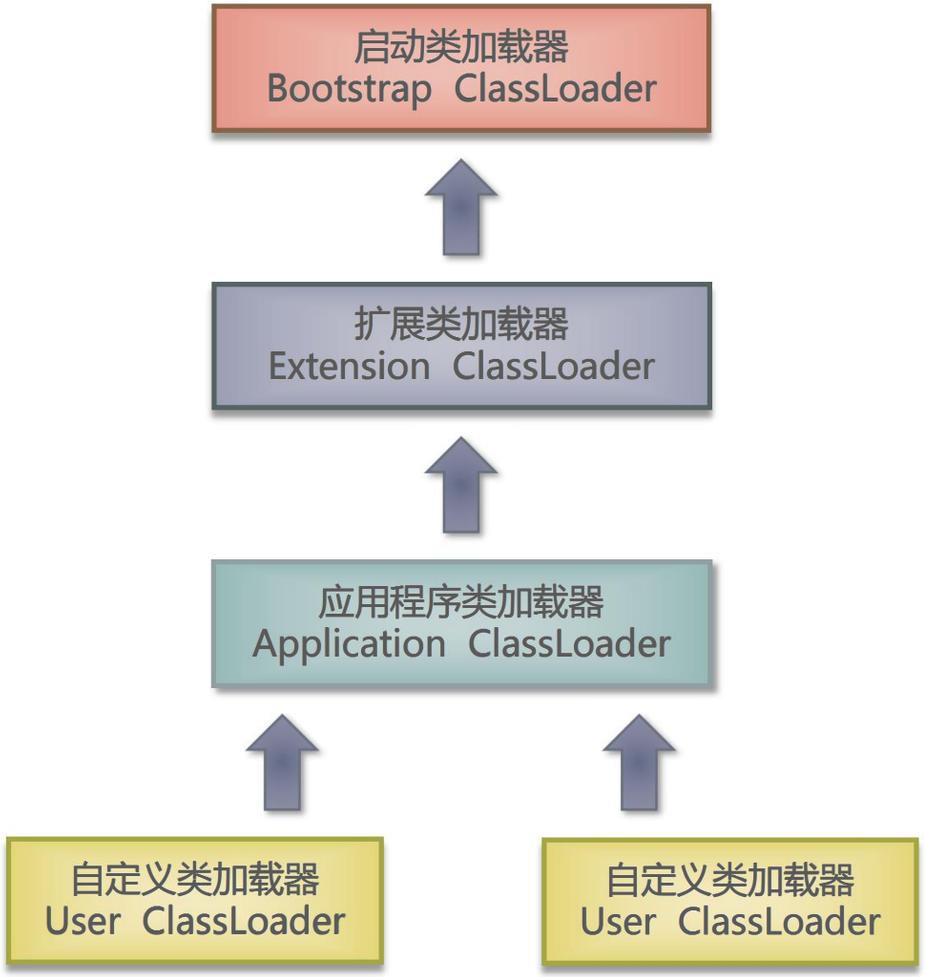
加载=>链接=>初始化=>使用=>卸载

JVM根据Java语言和JVM的语义要求检查这个二进制形式

准备要执行的指定的类，准备阶段为变量分配内存并设置静态变量的初始化

检查指定的类是否引用了其他的类/接口，是否能找到和加载其他的类/接口

类加载机制



启动（Bootstrap）类加载器：负责将JAVA_HOME/lib下面的类库加载到内存中（比如rt.jar）。由于引导类加载器涉及到虚拟机本地实现细节，开发者无法直接获取到启动类加载器的引用，所以不允许直接通过引用进行操作。

标准扩展（Extension）类加载器：负责将JAVA_HOME/jre/lib/ext或者由系统变量 java.ext.dirs指定位置中的类库加载到内存中

应用程序（Application）类加载器：它负责将系统类路径（CLASSPATH）中指定的类库加载到内存中。由于这个类加载器是ClassLoader中的getSystemClassLoader()方法的返回值，因此一般称为系统（System）加载器

检查顺序：自底向上
加载顺序：自顶向下

双亲委任

一种安全机制：双亲委任

程序安全是JDK的事
文件安全是系统的事

父类能加载的
就不给子类加载

程序猿写了不安全的程序
JDK有责任不让它运行

为什么需要双亲委任

黑客自定义一个java.util.List类，该List类具有系统的List类一样的功能，只是在某个函数稍作修改。

这个函数经常使用，假如在这这个函数中植入一些“病毒代码”。并且通过自定义类加载器加入到JVM中。

01 基础篇

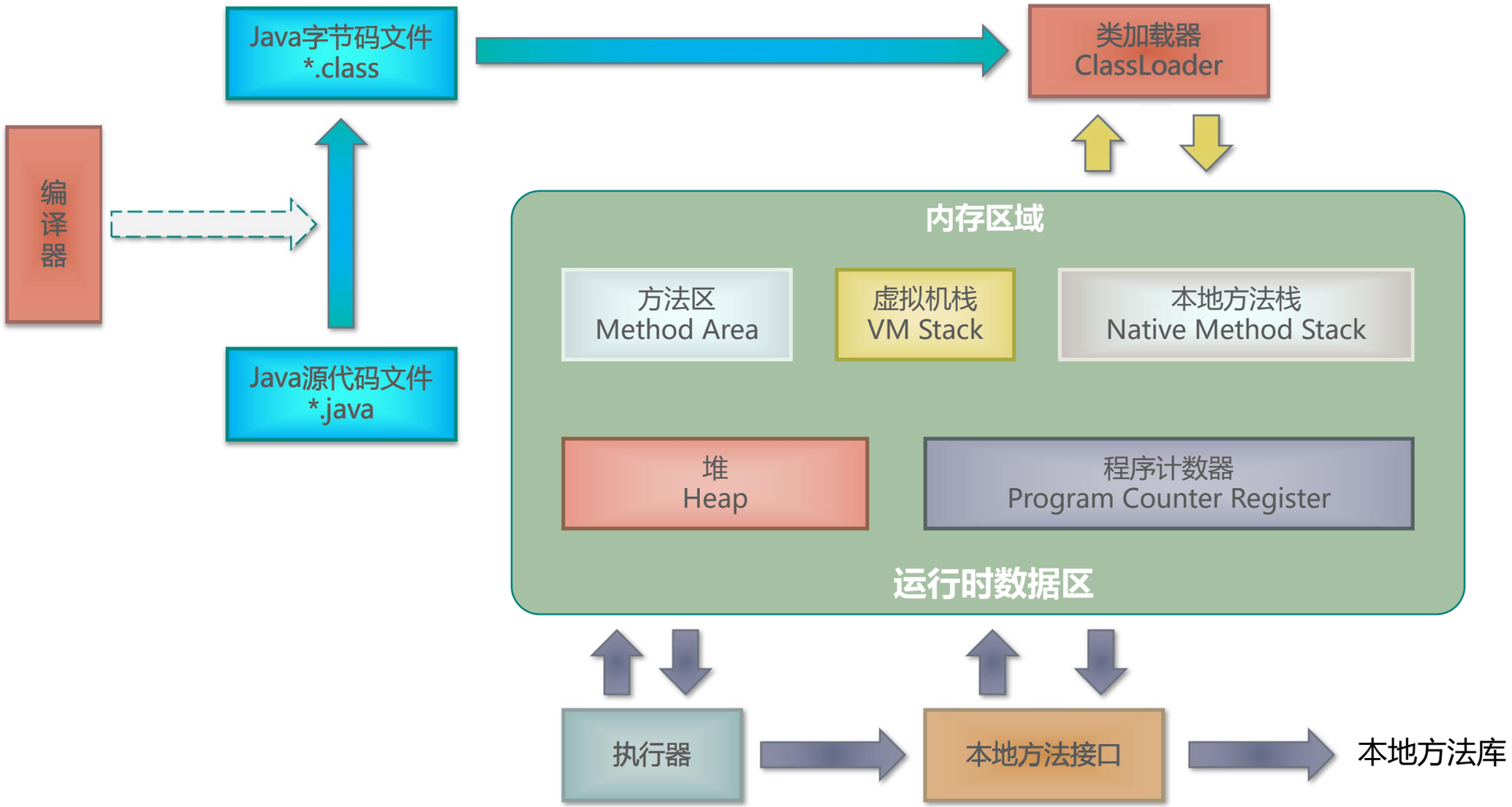
Java类文件结构

Java虚拟机类加载机制

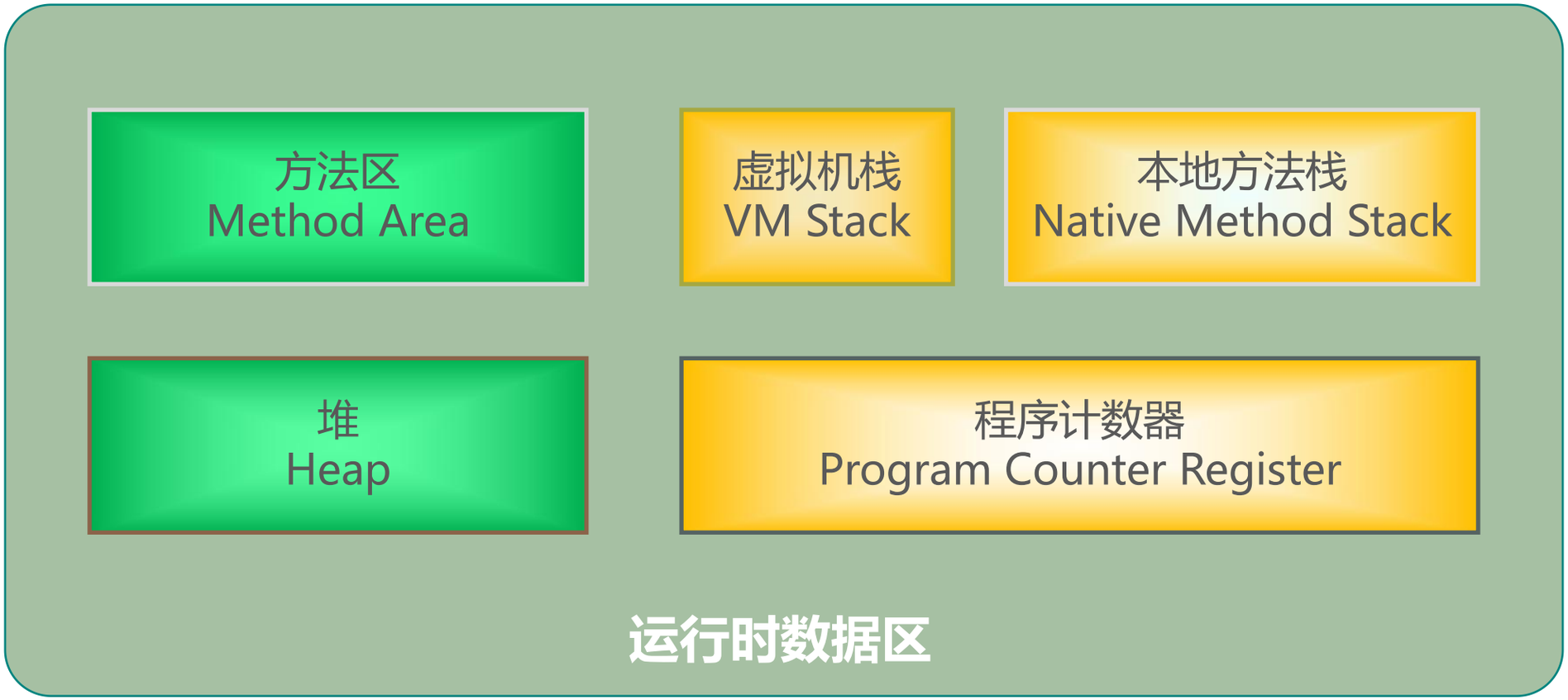
Java虚拟机运行时数据区

Java垃圾回收策略

JVM内存结构规范


Java字节码文件
*.class
类加载器
ClassLoader
编译器
Java源代码文件
*.java
内存区域
方法区
Method Area
虚拟机栈
VM Stack
本地方法栈
Native Method Stack
堆
Heap
程序计数器
Program Counter Register
运行时数据区
执行器
本地方法接口
本地方法库

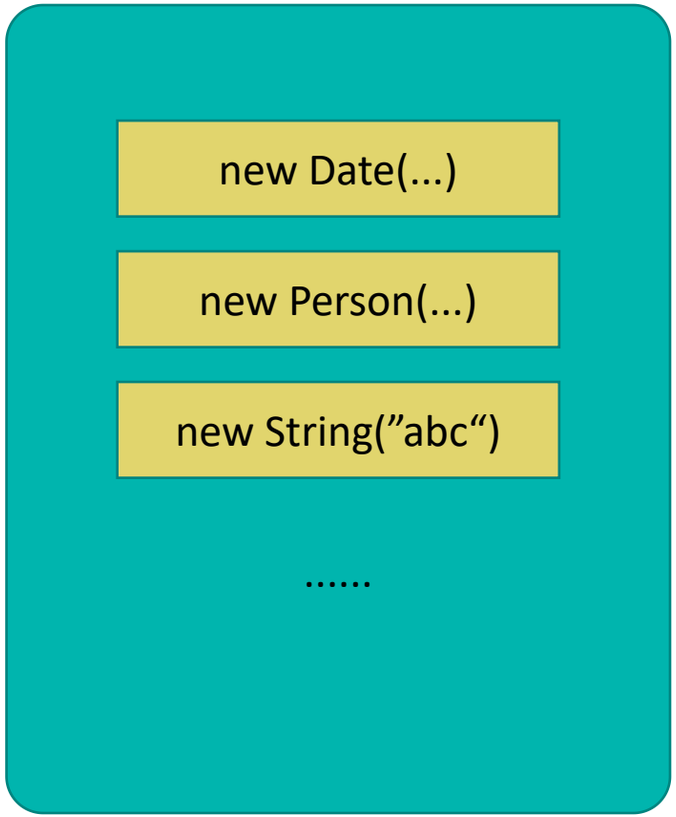
运行时数据区



运行时数据区在所有线程间共享

运行时数据区线程私有

Java堆（Heap）



对于大多数应用来说，Java堆（Heap）是Java虚拟机所管理的内存中最大的一块

- Java堆是被所有线程共享的一块内存区域，在虚拟机启动时创建
- 此内存区域的唯一目的就是存放对象实例，几乎所有的对象实例都在这里分配内存
- OutOfMemoryError异常。如果在堆中没有内存完成实例分配，并且堆也无法再扩展时

堆主要用来存放对象实例

方法区（Method Area）

方法区（Method Area）与Java堆一样，是各个线程共享的内存区域。用于存储：

1. 已被虚拟机加载的类信息
2. 常量
3. 静态变量
4. 即时编译器编译后的代码

当方法区无法满足内存分配需求时
抛出OutOfMemoryError异常

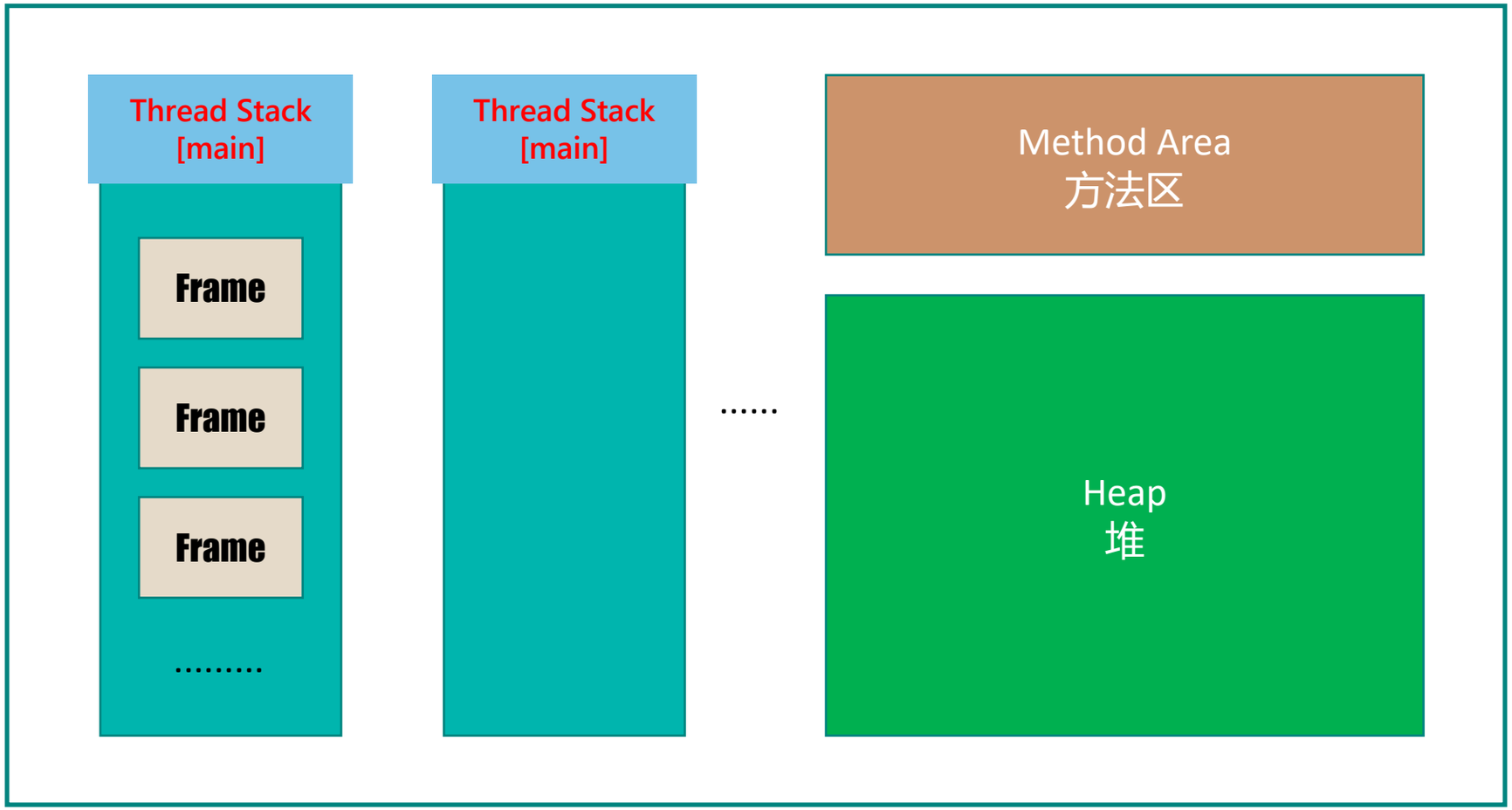
程序计数器（Program Counter Register）

```
public void test(int, double);
  Code:
     0: aload_0
     1: dup
     2: getfield      #18
     5: invokevirtual #26
     8: lconst_1
     9: ladd
    10: invokestatic  #12
    13: putfield      #18
    16: getstatic     #30
    19: new           #36
    22: dup
    23: ldc           #38
    25: invokespecial #40
    28: aload_0
    29: getfield      #18
    32: invokevirtual #43
```

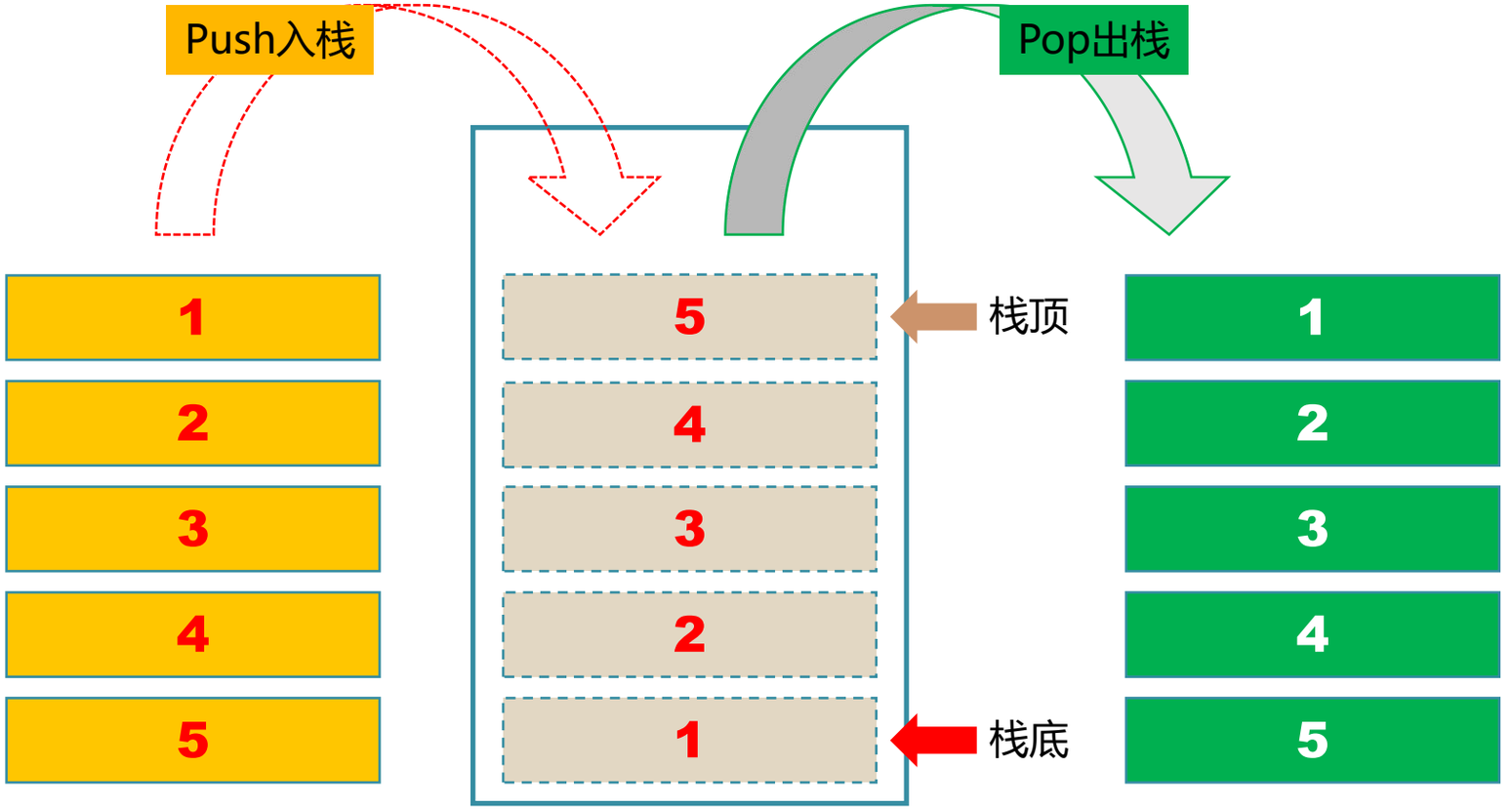
一块较小的内存空间，它的作用是当前线程所执行的字节码行号指示器

唯一一个在JVM规范中没有规定任何OutOfMemoryError的区域

JVM内存结构规范

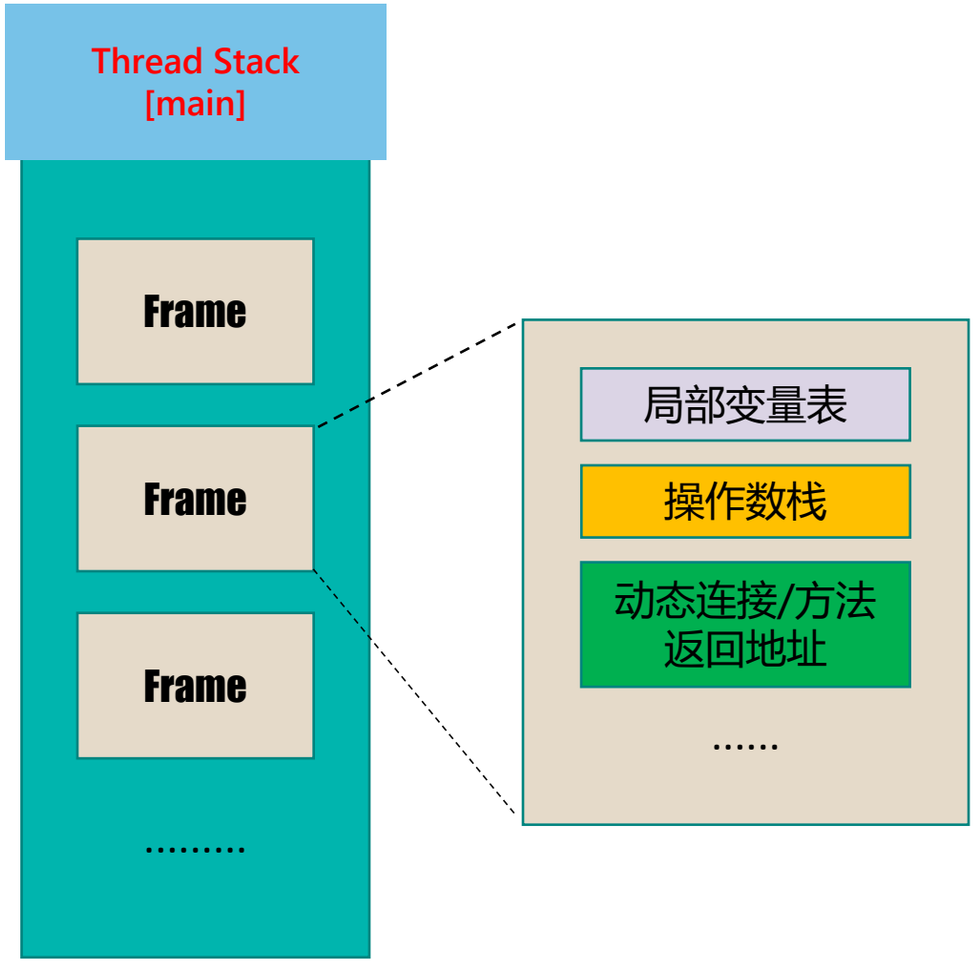

Thread Stack
[main]
Frame
Frame
Frame
.....
Thread Stack
[main]
.....
Method Area
方法区
Heap
堆

JVM内存结构规范——栈



每当启动一个新线程的时候。Java虚拟机都会为它分配一个Java栈。Java以栈帧为单位保存线程的运行状态。虚拟机只会对Java栈执行两种操作：以栈帧为单位的入栈或者出栈。

Java虚拟机栈和栈帧



保存什么

每个线程包含一个栈区，栈中只保存基础数据类型的对象和自定义对象的引用（不是对象）

私有性

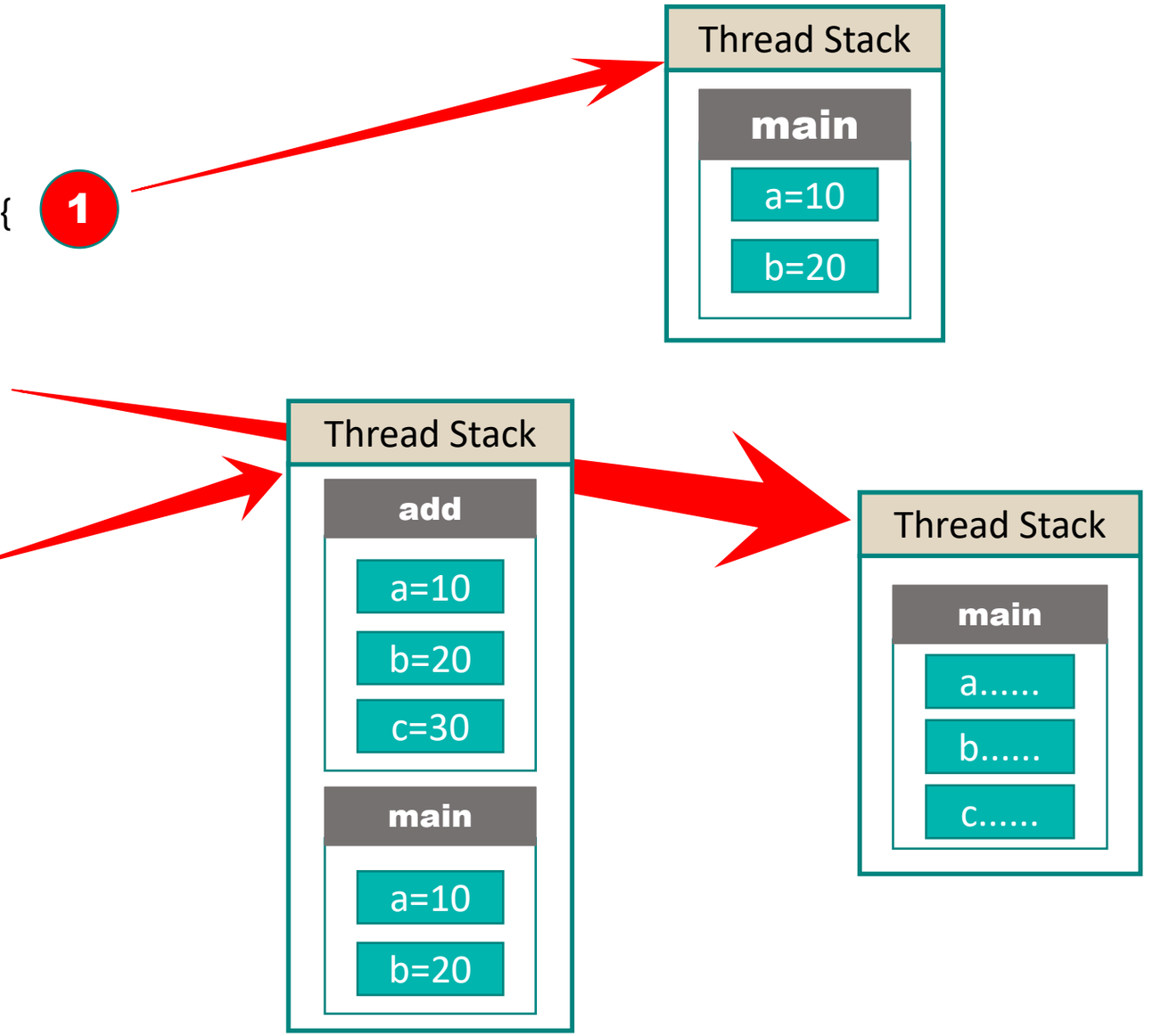
每个栈中的数据(原始类型和对象引用)都是私有的，其他栈不能访问

组成部分

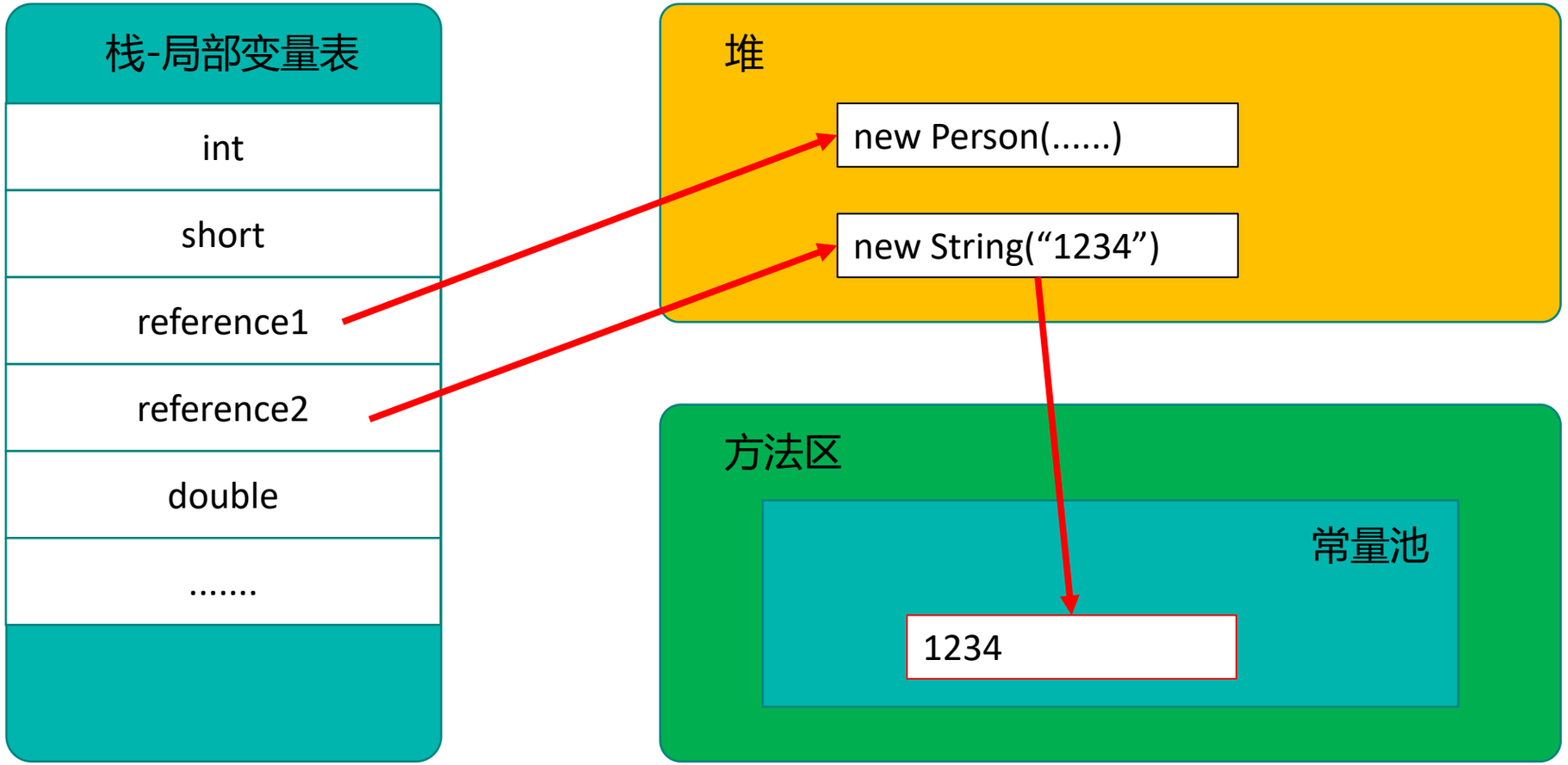
栈分为3个部分：基本类型变量区、执行环境上下文、操作指令区（存放操作指令）

JVM栈之局部变量表

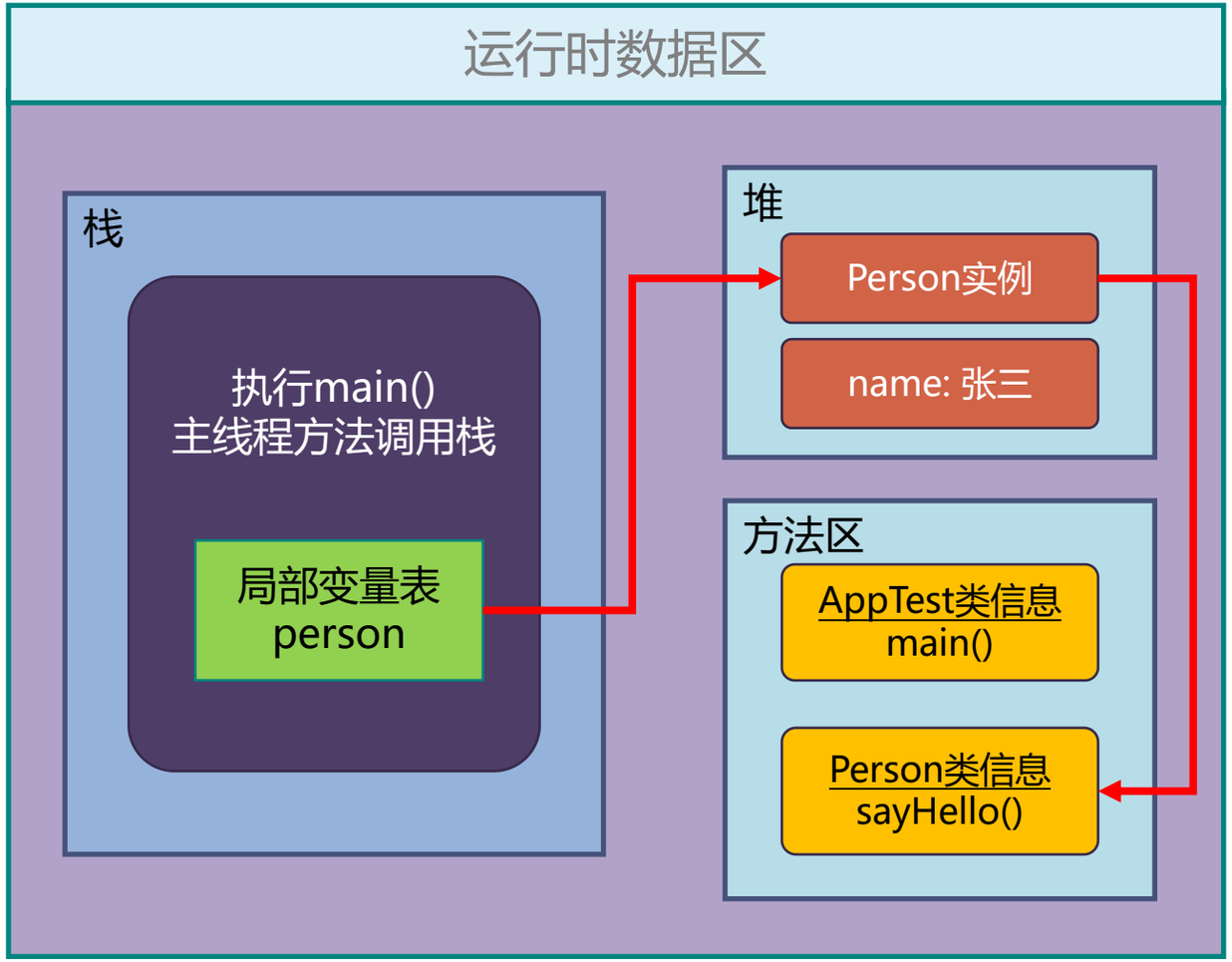
```
public static main(String[] args) {
    int a = 10;
    int b = 20;
    int c = add(a, b);
    System.out.println(c);
}

public static add(int a, int b) {
    int c = a + b;
    return c;
}
```



JVM堆、栈和方法区


栈-局部变量表
int
short
reference1
reference2
double
.....
堆
new Person(.....)
new String(“1234”)
方法区
常量池
1234

JVM执行流程



1. JVM去方法区寻找Person类信息
2. 如果找不到，Classloader加载Person类信息进入内存方法区
3. 在堆内存中创建Person对象，并持有方法区中Person类的类型信息的引用
4. 把person添加到执行main()方法的主线程java调用栈中，指向堆空间中的内存对象
5. 执行person.sayHello()时，JVM根据person定位到堆空间的Person实例
6. 根据Person实例在方法区持有的引用，定位到方法区Person类型信息，获得sayHello() 字节码，执行此方法执行，打印出结果。

01 基础篇

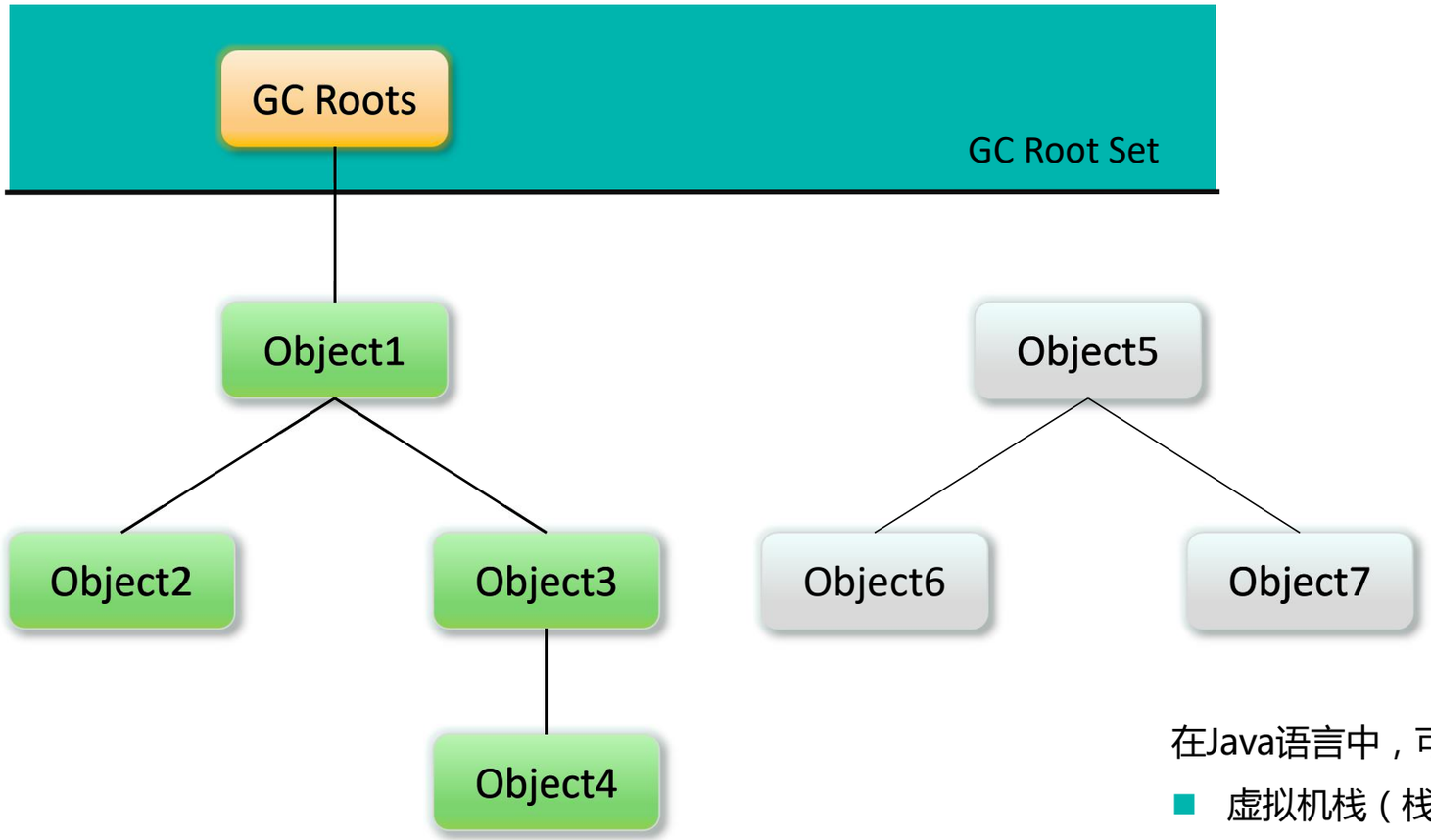
Java类文件结构

Java虚拟机类加载机制

Java虚拟机运行时数据区

Java垃圾回收策略

可达性分析算法



在Java语言中，可作为GC Roots的对象包括下面几种：

- 虚拟机栈（栈帧中的本地变量表）中引用的对象。
- 方法区中类静态属性引用的对象。
- 方法区中常量引用的对象。
- 本地方法栈中JNI（即一般说的Native方法）引用的对象

Java的引用类型

强引用（Strong Reference）

Object obj = new Object(); 这样的常规引用，只要引用还在，就永远不会回收对象。

软引用（Soft Reference）

在发生内存溢出之前，进行回收，如果这次回收之后还没有足够的内存，则报OOM

弱引用（Weak Reference）

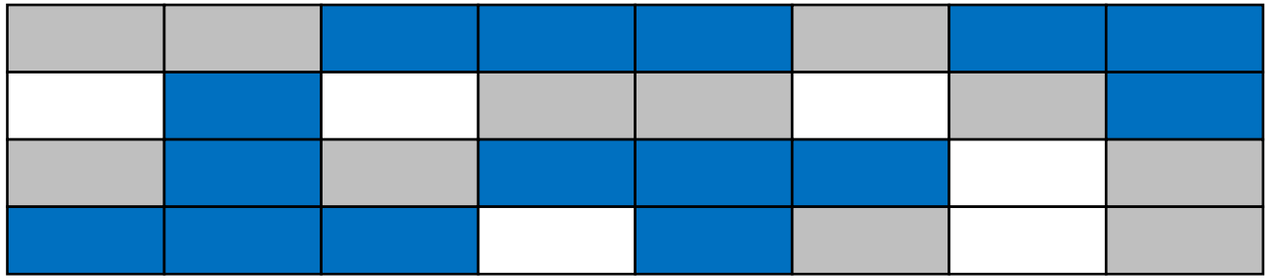
生存到下一次垃圾回收之前，无论当前内存是否够用，都回收掉被弱引用关联的对象

虚引用（Phantom Reference）

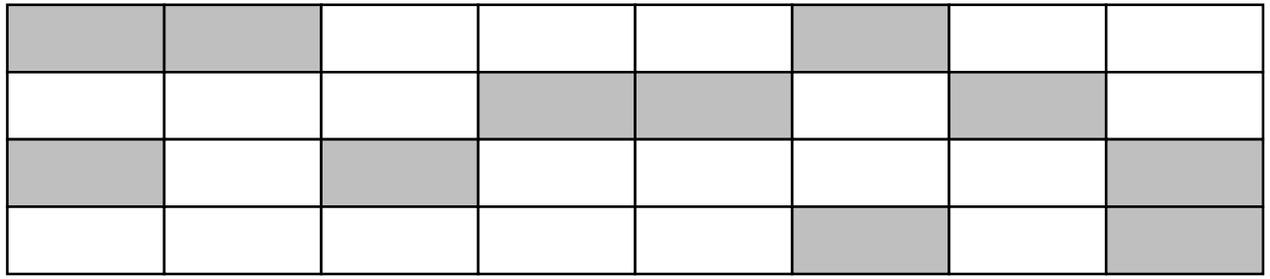
不会对对象的生命周期有任何影响，也无法通过它得到对象的实例，唯一的作用也就是在对象被垃圾回收前收到一个系统通知

标记-清除算法

回收前：



回收后：



该算法分为“标记”和“清除”两个阶段：首先标记出所有需要回收的对象，在标记完成后统一回收所有被标记的对象。

两个主要不足：

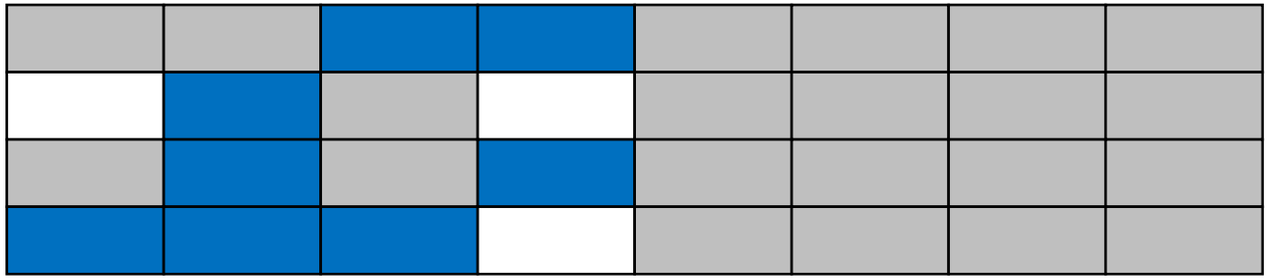
- 一个是效率问题，标记和清除的效率都不高；
- 另一个是空间问题，标记清除之后会产生大量不连续的内存碎片。

复制算法

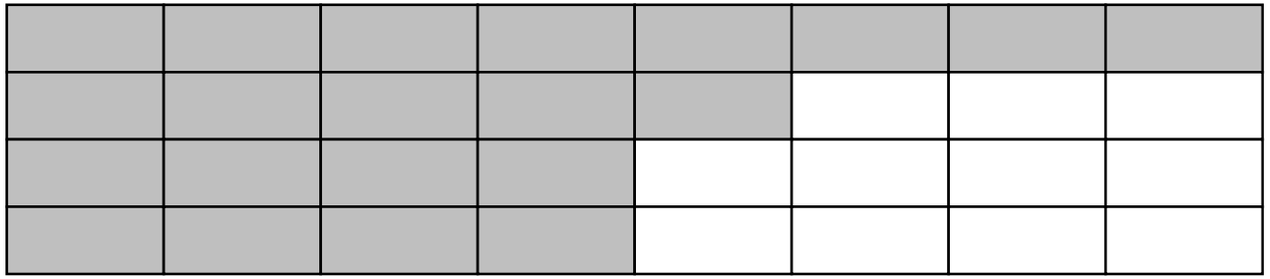
将可用内存按容量划分为大小相等的两块，每次只使用其中的一块。当这一块的内存用完了，就将还存活着的对象复制到另外一块上面，然后再把已使用过的内存空间一次清理掉

缺点：内存缩小为了原来的一半，内存利用率太低

回收前：



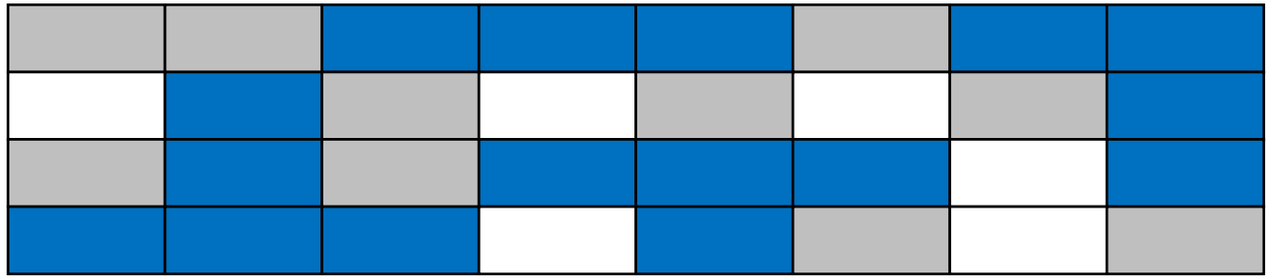
回收后：



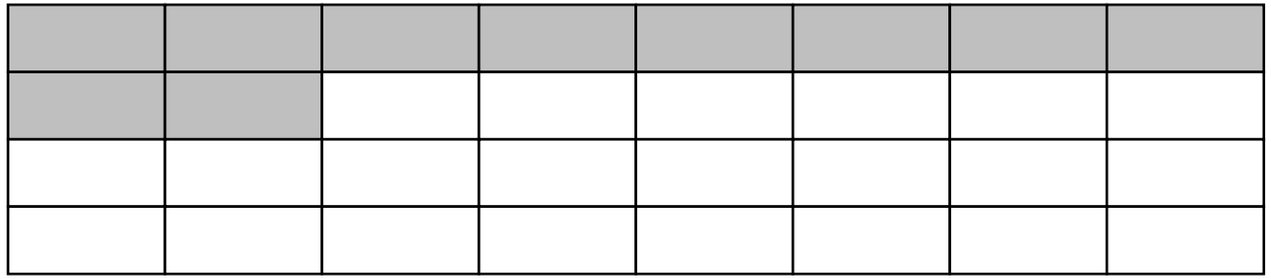
存活对象　可回收对象　未使用　保留区域

标记整理算法

回收前：



回收后：

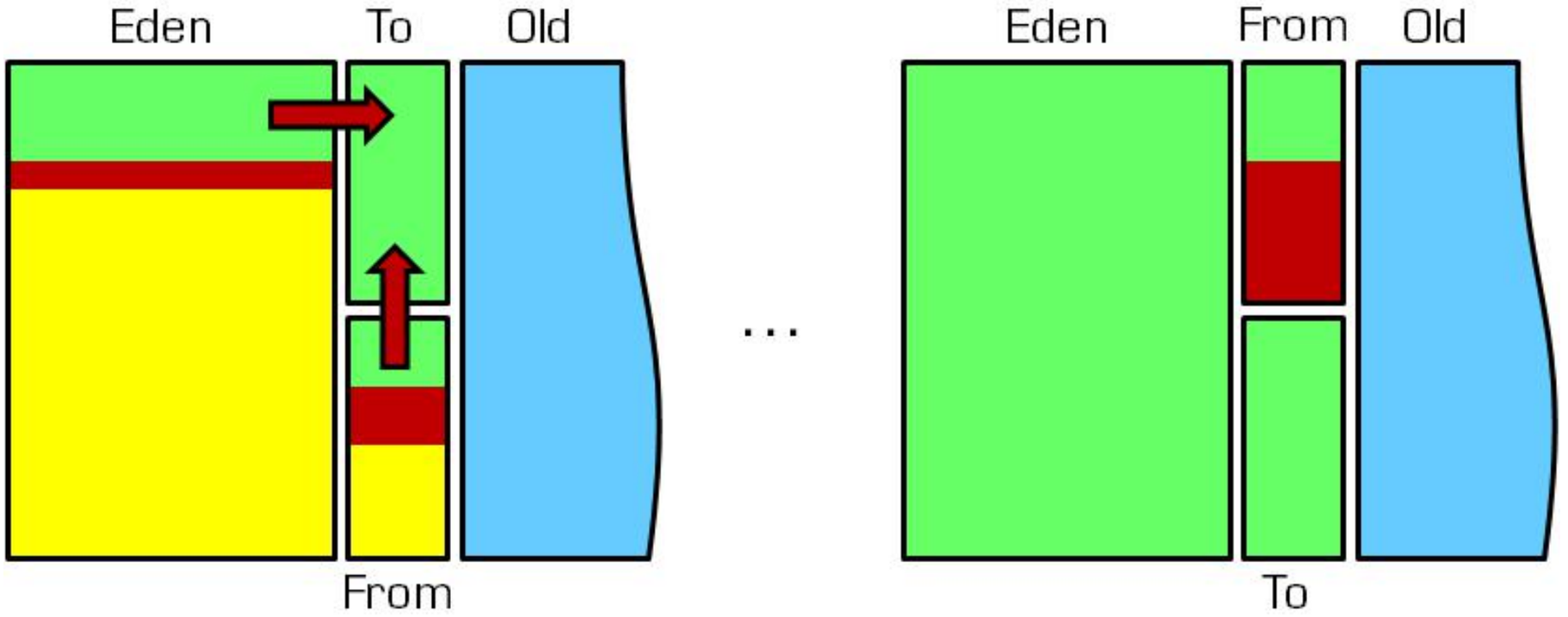


存活对象　可回收对象　未使用

标记-整理（Mark-Compact）算法，标记过程仍然与“标记-清除”算法一样，但后续步骤不是直接对可回收对象进行清理，而是让所有存活的对象都向一端移动，然后直接清理掉端边界以外的内存

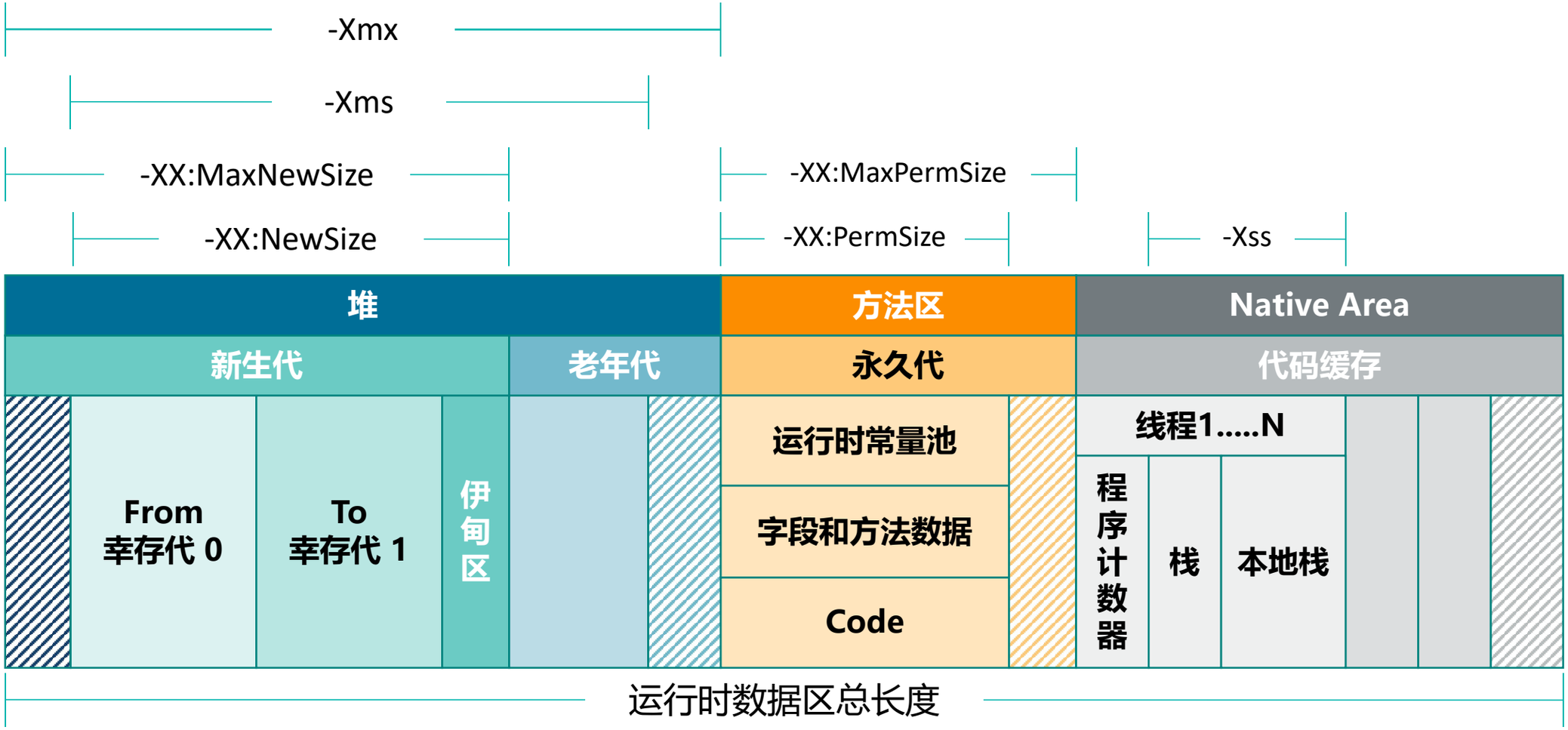
分代收集算法

目前商业虚拟机的垃圾收集都采用“分代收集”（Generational Collection）算法，这种算法是根据对象存活周期的不同将内存划分为几块。

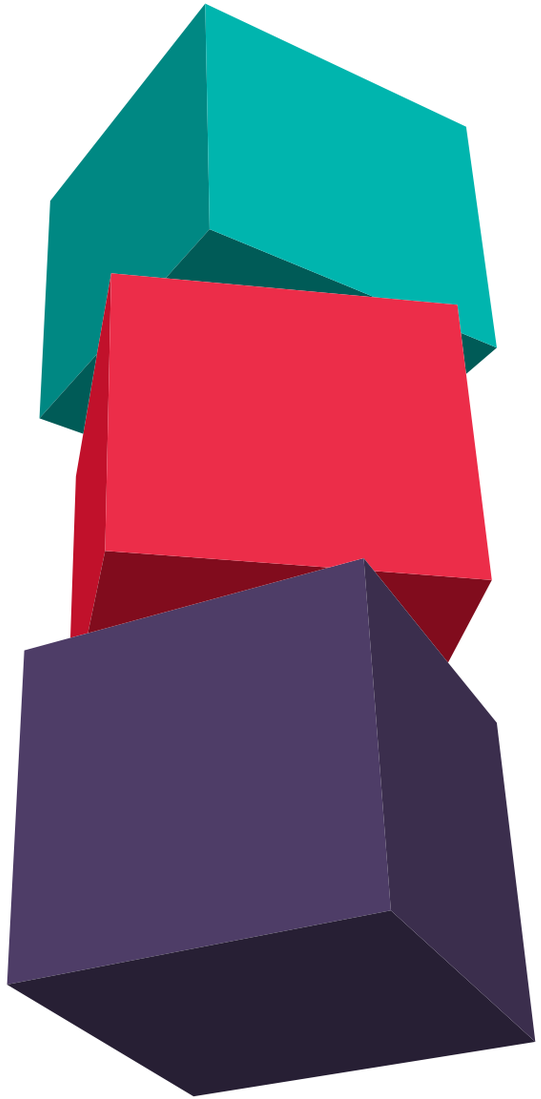


一般是把Java堆分为新生代和老年代，这样就可以根据各个年代的特点采用最适当的收集算法。在新生代中，每次垃圾收集时都发现有大批对象“朝生夕死”，只有少量存活，那就选用复制算法，只需要付出少量存活对象的复制成本就可以完成收集。 而老年代中因为对象存活率高、 没有额外空间对它进行分配担保，就必须使用“标记—清理”或者“标记—整理”算法来进行回收。

JVM内存区域示意图


-Xmx
-Xms
-XX:MaxNewSize
-XX:NewSize
-XX:MaxPermSize
-XX:PermSize
-Xss
堆
方法区
Native Area
新生代
老年代
永久代
代码缓存
From
幸存代 0
To
幸存代 1
伊甸区
运行时常量池
字段和方法数据
Code
线程1.....N
程序计数器
栈
本地栈
运行时数据区总长度

JVM部分内存参数



-Xms	设置堆的最小空间大小
-Xmx	设置堆的最大空间大小
-XX:NewSize	设置新生代最小空间大小
-XX:MaxNewSize	设置新生代最大空间大小
-XX:PermSize	设置永久代最小空间大小
-XX:MaxPermSize	设置永久代最大空间大小
-Xss	设置每个线程的堆栈大小

没有直接设置老年代的参数，但是可以设置堆空间大小和新生代空间大小两个参数来间接控制。

老年代空间大小=堆空间大小-年轻代大空间大小

安全点Safepoint

在OopMap的协助下，HotSpot可以快速且准确地完成GC Roots枚举，但如果为每一条指令都生成对应的OopMap那将会需要大量的额外空间，这样GC的空间成本将会变得很高。

抢先式中断

在GC发生时，首先把所有线程全部中断，如果发现有线程中断的地方不在安全点上，就恢复线程，让它“跑”到安全点上。现在几乎没有虚拟机实现采用抢先式中断来暂停线程从而响应GC事件

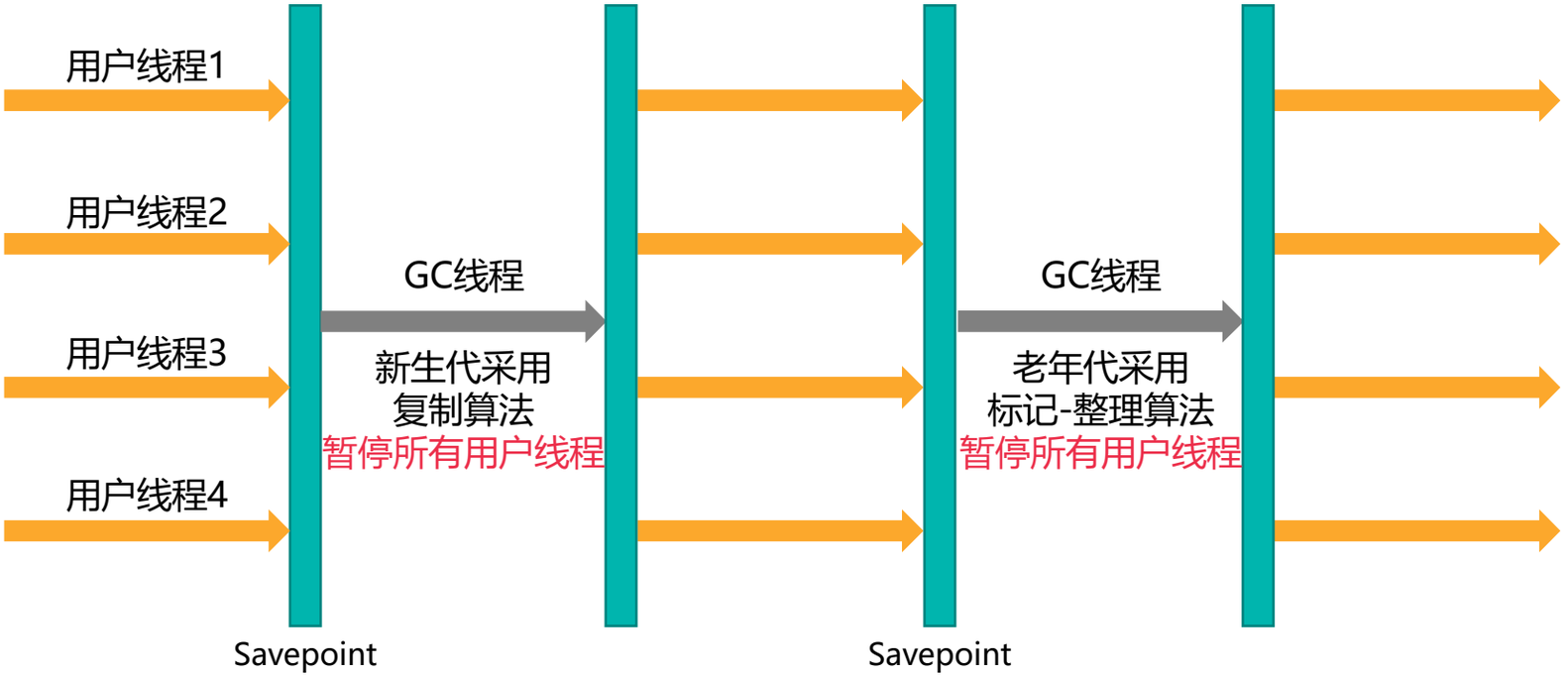
主动式中断

当GC需要中断线程的时候，不直接对线程操作，仅仅简单地设置一个标志，各个线程执行时主动去轮询这个标志，发现中断标志为真时就自己中断挂起

Serial收集器

开启参数：-XX:+UseSerialGC

适用场景：用户的桌面应用场景

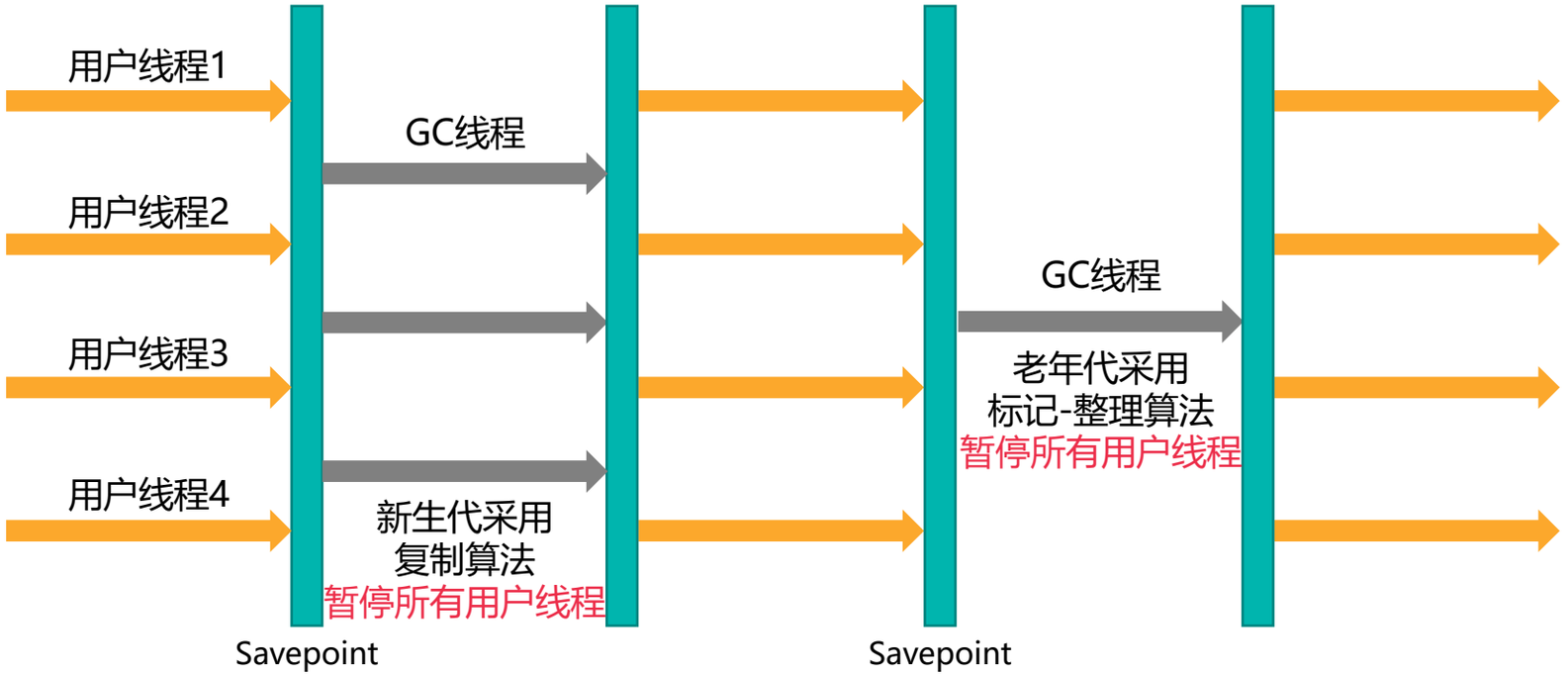


单线程的收集器，它的“单线程”的意义并不仅仅说明它只会使用一个CPU或一条收集线程去完成垃圾收集工作，最重要的是在它进行垃圾收集时，必须暂停其他所有的工作线程，直到它收集结束

ParNew收集器

开启参数：-XX:+UseParNewGC

适用场景：Server首选的新生代收集器



Serial收集器的多线程版本，除了使用多条线程进行垃圾收集之外，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 World、 对象分配规则、 回收策略等都与Serial收集器完全一样

Parallel Scavenge收集器

开启参数：-XX:+UseParallelGC

适用场景：后台计算不需要太多交互

Parallel Scavenge关注点：可控的吞吐量

吞吐量计算公式：运行用户代码时间/（运行用户代码时间+垃圾收集时间）

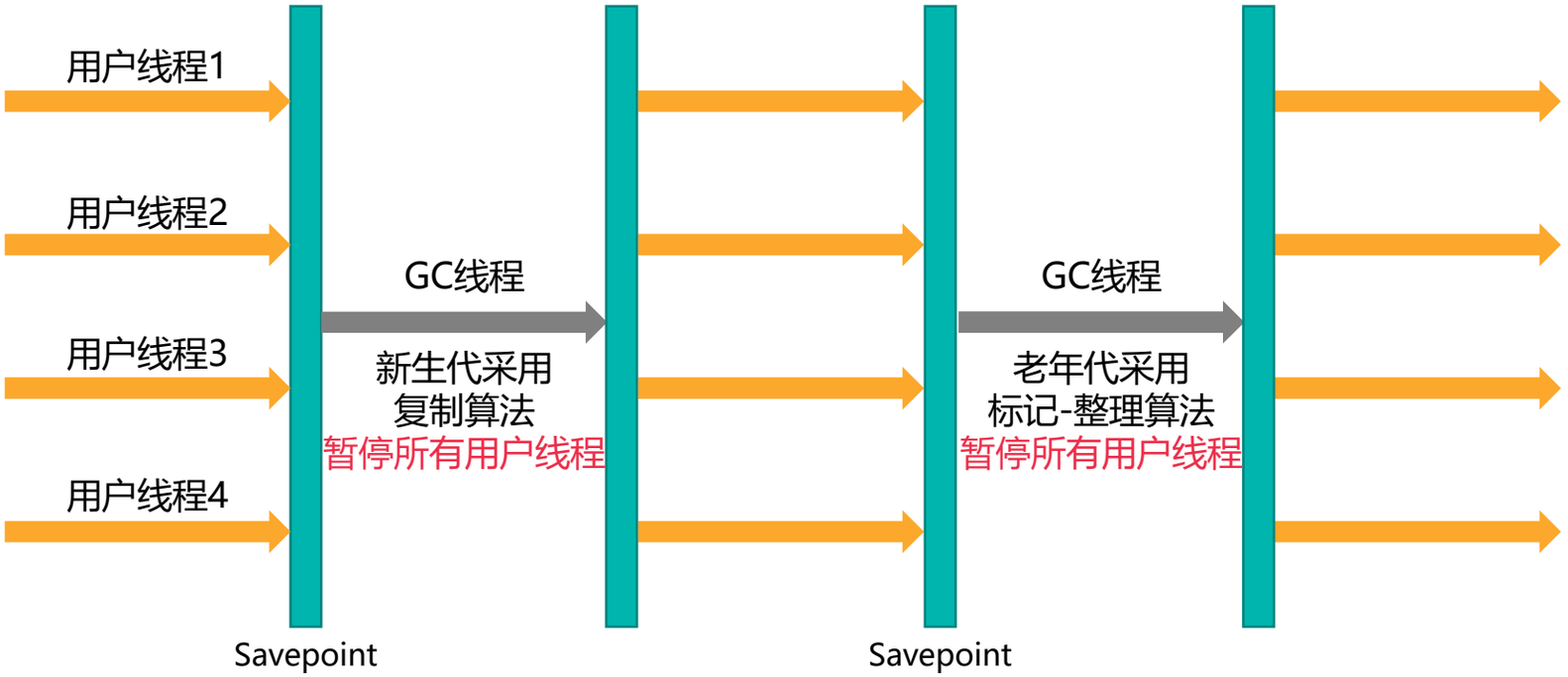
停顿时间越短就越适合需要与用户交互的程序，良好的响应速度能提升用户体验，而高吞吐量则可以高效率地利用CPU时间，尽快完成程序的运算任务。

Parallel Scavenge提供了两个参数用于精确控制吞吐量

- -XX：MaxGCPauseMillis　　// 最大垃圾收集停顿时间（大于0毫秒数）
- -XX：GCTimeRatio　　// 吞吐量大小（大于0且小于100的整数，吞吐量百分比）
- -XX：+UseAdaptiveSizePolicy　　// 内存调优委托给虚拟机管理

Serial Old收集器

适用场景：用户的桌面应用场景

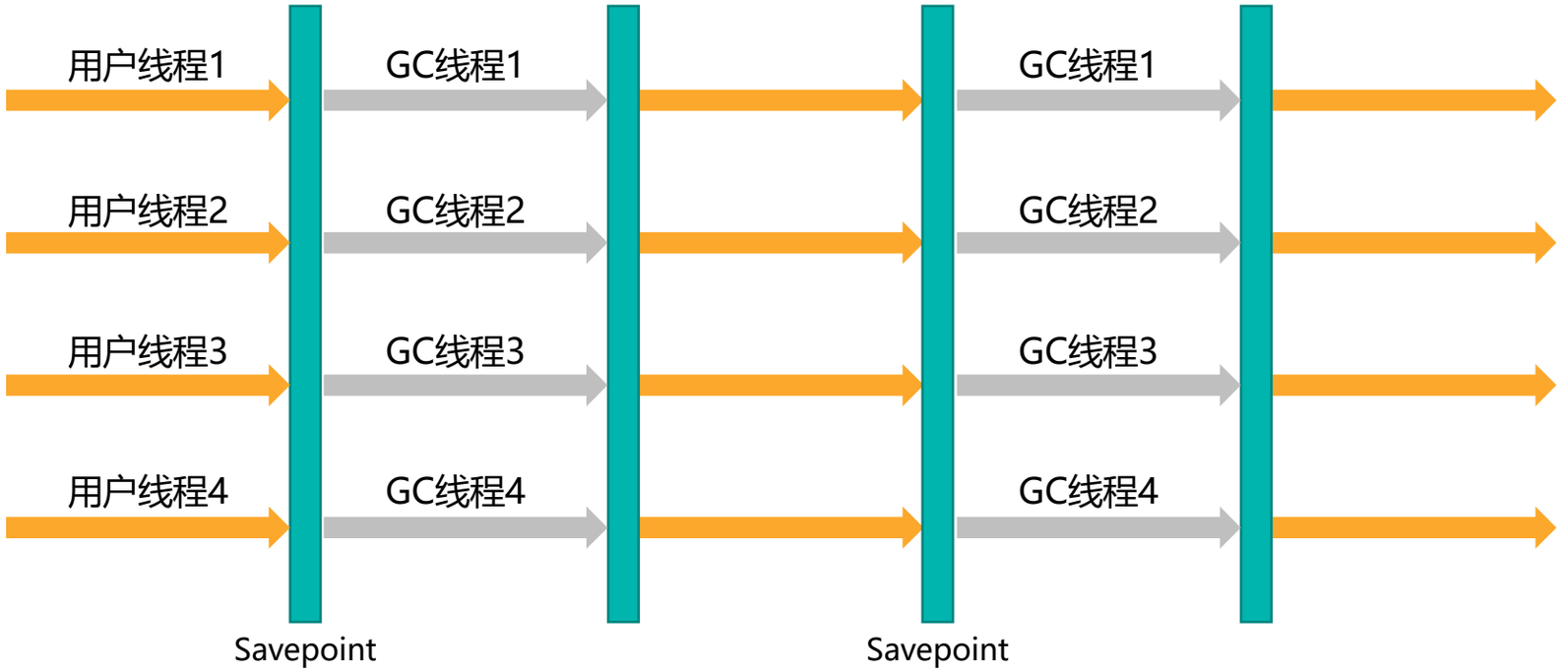


Serial Old是Serial收集器的老年代版本，它同样是一个单线程收集器，使用“标记-整理”算法。 这个收集器的主要意义也是在于给Client模式下的虚拟机使用。

Parallel Old收集器

开启参数：-XX:+UseParallelOldGC

适用场景：用户的桌面应用场景

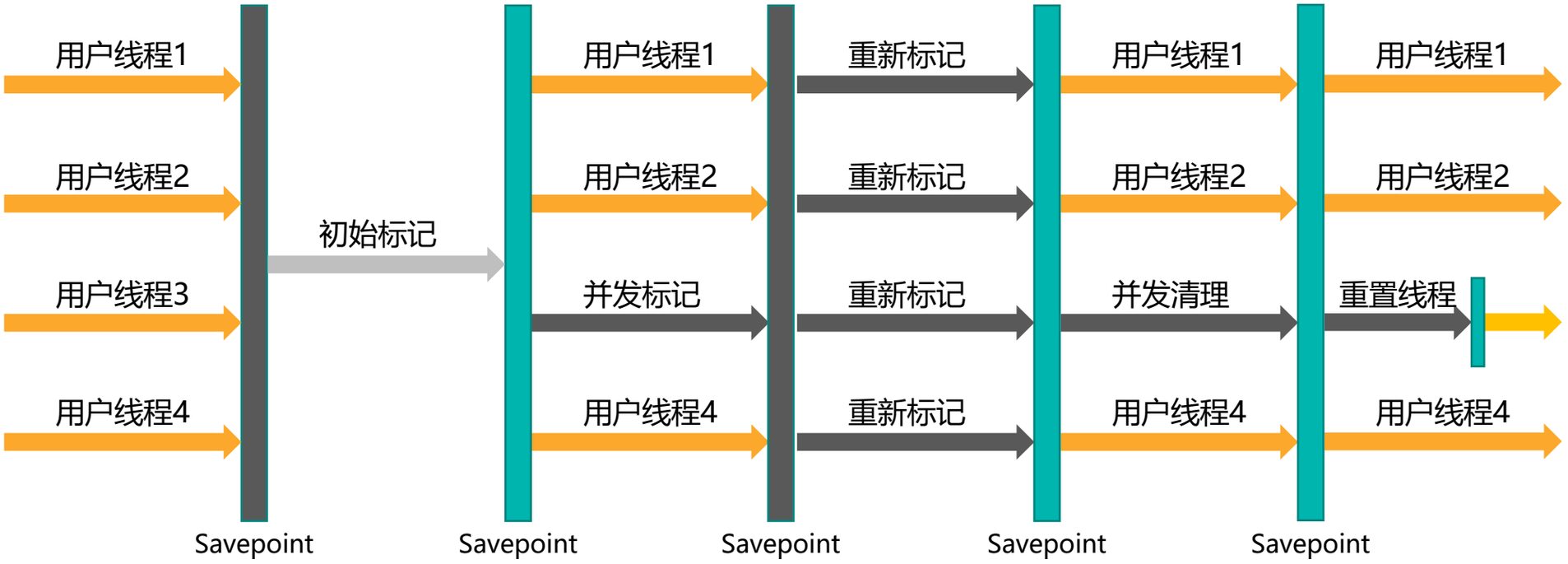


Parallel Old是Parallel Scavenge收集器的老年代版本，使用多线程和“标记-整理”算法。这个收集器从JDK 1.6中才开始提供的。

Concurrent Mark Sweep收集器

开启参数：-XX:+UseConcMarkSweepGC

适用场景：互联网站或者WEB服务端



CMS（Concurrent Mark Sweep）收集器是一种以获取最短回收停顿时间为目标的收集器。CMS收集器是基于“标记—清除”算法实现的。整个过程分为4个步骤，包括：初始标记（CMS initial mark）、并发标记（CMS concurrent mark）、重新标记（CMS remark）、并发清除（CMS concurrent sweep）

G1收集器

适用场景：服务端应用

开启参数：-XX:+UseG1GC

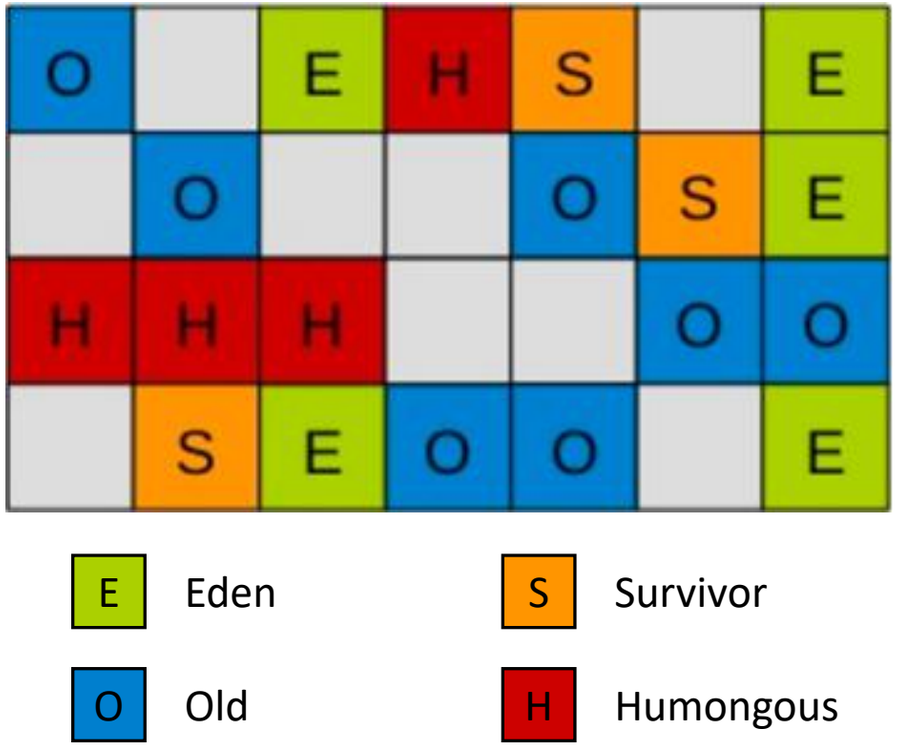
G1算法将堆划分为若干个区域（Region），但它仍然属于分代收集器。

不过，这些区域的一部分包含新生代，新生代的垃圾收集依然采用暂停所有应用线程的方式，将存活对象拷贝到老年代或者Survivor空间。

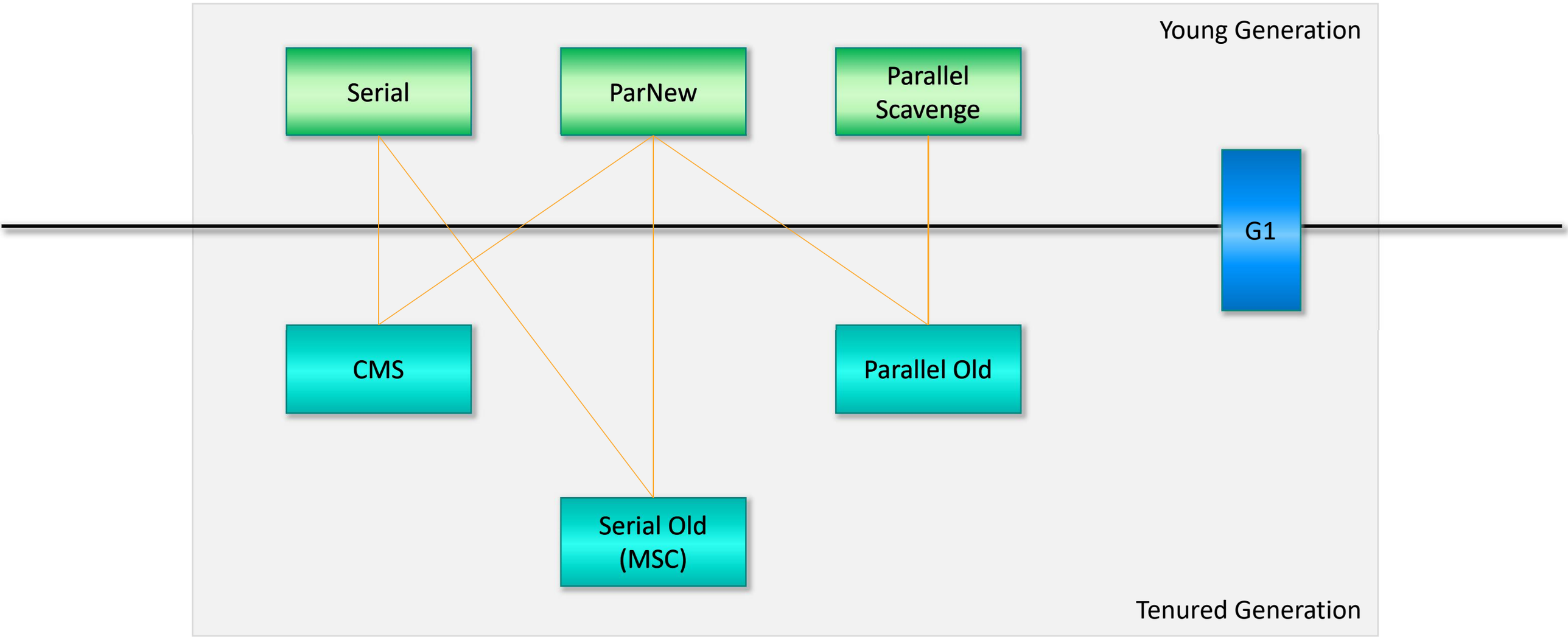
老年代也分成很多区域，G1收集器通过将对象从一个区域复制到另外一个区域，完成了清理工作。这就意味着，在正常的处理过程中，G1完成了堆的压缩（至少是部分堆的压缩），这样也就不会有CMS内存碎片问题的存在了。

G1提供了两种GC模式（两种都是Stop The World的）

- Young GC
- Mixed GC



垃圾回收器


Young Generation
Serial
ParNew
Parallel Scavenge
G1
CMS
Parallel Old
Serial Old (MSC)
Tenured Generation

JVM内存分配与回收策略

大对象直接进入老年代

大对象是指，需要大量连续内存空间的Java对象，典型的大对象就是很长的字符串或者大数组。

-XX:PretenureSizeThreshold可以令大于这个设置值的对象直接在老年代分配。这样可以避免在Eden区及两个Survivor区之间发生大量的内存复制

长期存活的对象将进入老年代

对象在Survivor区中每“熬过”一次Minor GC，年龄就增加1岁，当它的年龄增加到一定程度（默认为15岁），就将会被晋升到老年代中

动态对象年龄判定

如果在Survivor空间中相同年龄所有对象大小的总和大于Survivor空间的一半，年龄大于或等于该年龄的对象就可以直接进入老年代，无须等到MaxTenuringThreshold中要求的年龄

空间分配担保

HandlePromotionFailure，检查老年代最大可用的连续空间是否大于历次晋升到老年代对象的平均大小，如果大于，将尝试着进行一次Minor GC；如果小于，或者设置不允许冒险，那这时也要改为进行一次Full GC。

02高级篇

字节码执行引擎

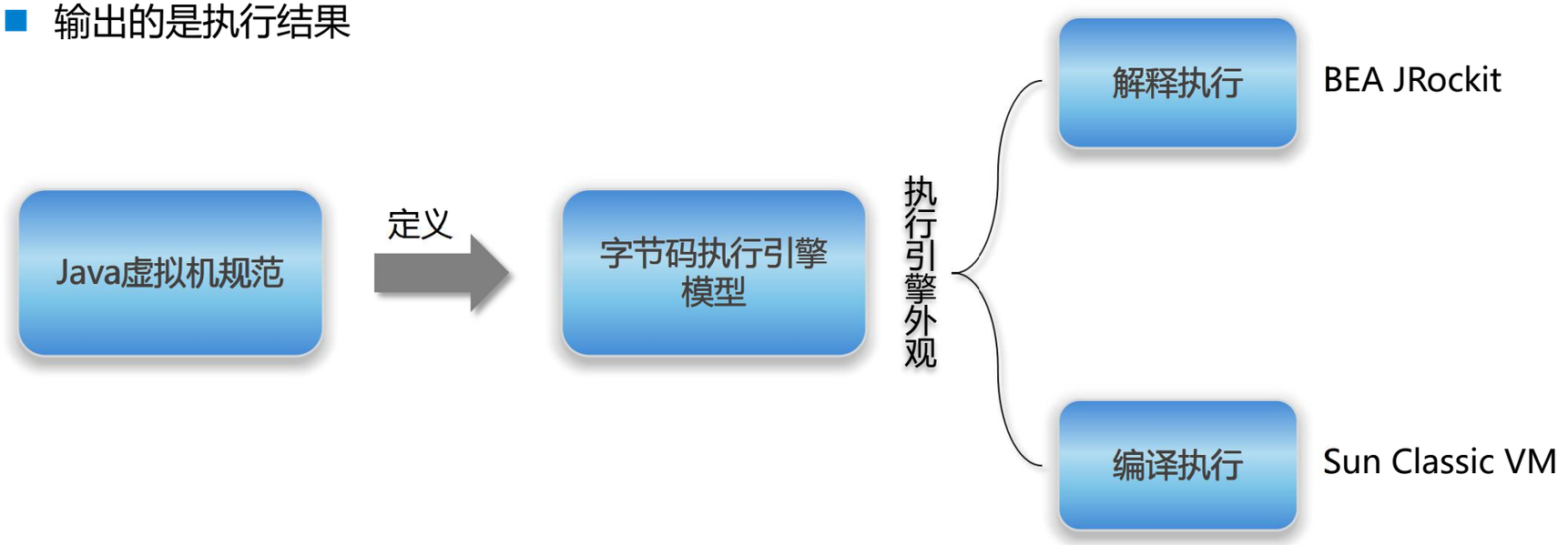
JDK性能监控与故障处理工具

字节码简介

字节码执行引擎

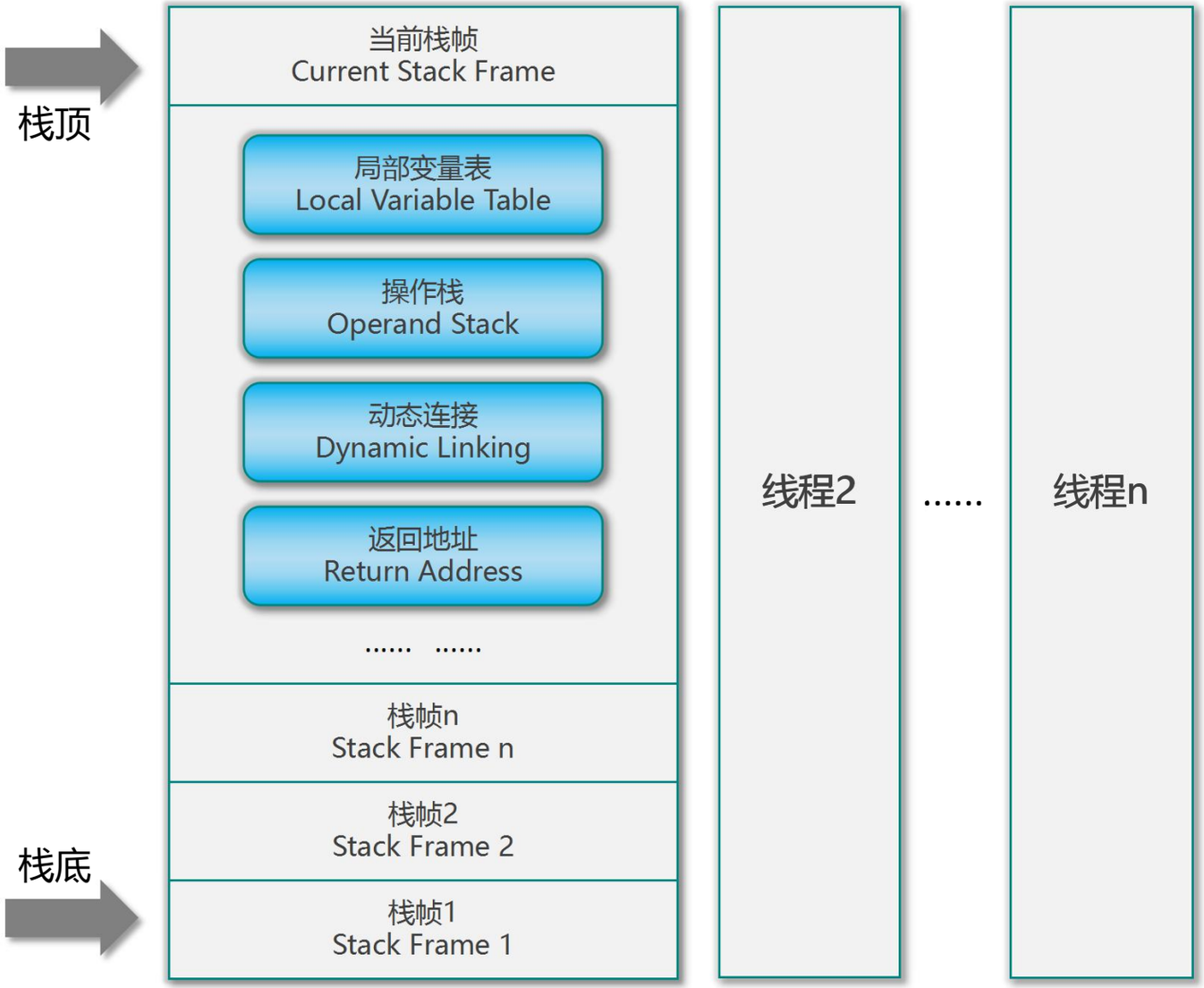
所有的Java虚拟机的执行引擎都是

- 输入的是字节码文件
- 处理过程是字节码解析的过程
- 输出的是执行结果



虚拟机的执行引擎则是由自己实现的，因此可以自行制定指令集与执行引擎的结构体系，并且能够执行那些不被硬件直接支持的指令集格式。

运行时栈帧结构



每一个方法从调用开始至执行完成的过程，都对应着一个栈帧在虚拟机栈里面从入栈到出栈的过程

栈帧存储了方法的

- 局部变量表
- 操作数栈
- 动态连接和
- 方法返回地址
- 额外的附加信息

一个栈帧需要分配多少内存，在编译的时候已经确定，不会受到程序运行期变量数据的影响，而仅仅取决于具体的虚拟机实现

局部变量表

局部变量表（Local Variable Table）是一组变量值存储空间，用于存放方法参数和方法内部定义的局部变量。在Java程序编译为Class文件时，就在方法的Code属性的max_locals数据项中确定了该方法所需要分配的局部变量表的最大容量。

- boolean
- byte
- char
- short
- int
- float
- reference
- returnAddress

可以使用32位或更小的物理内存来存放

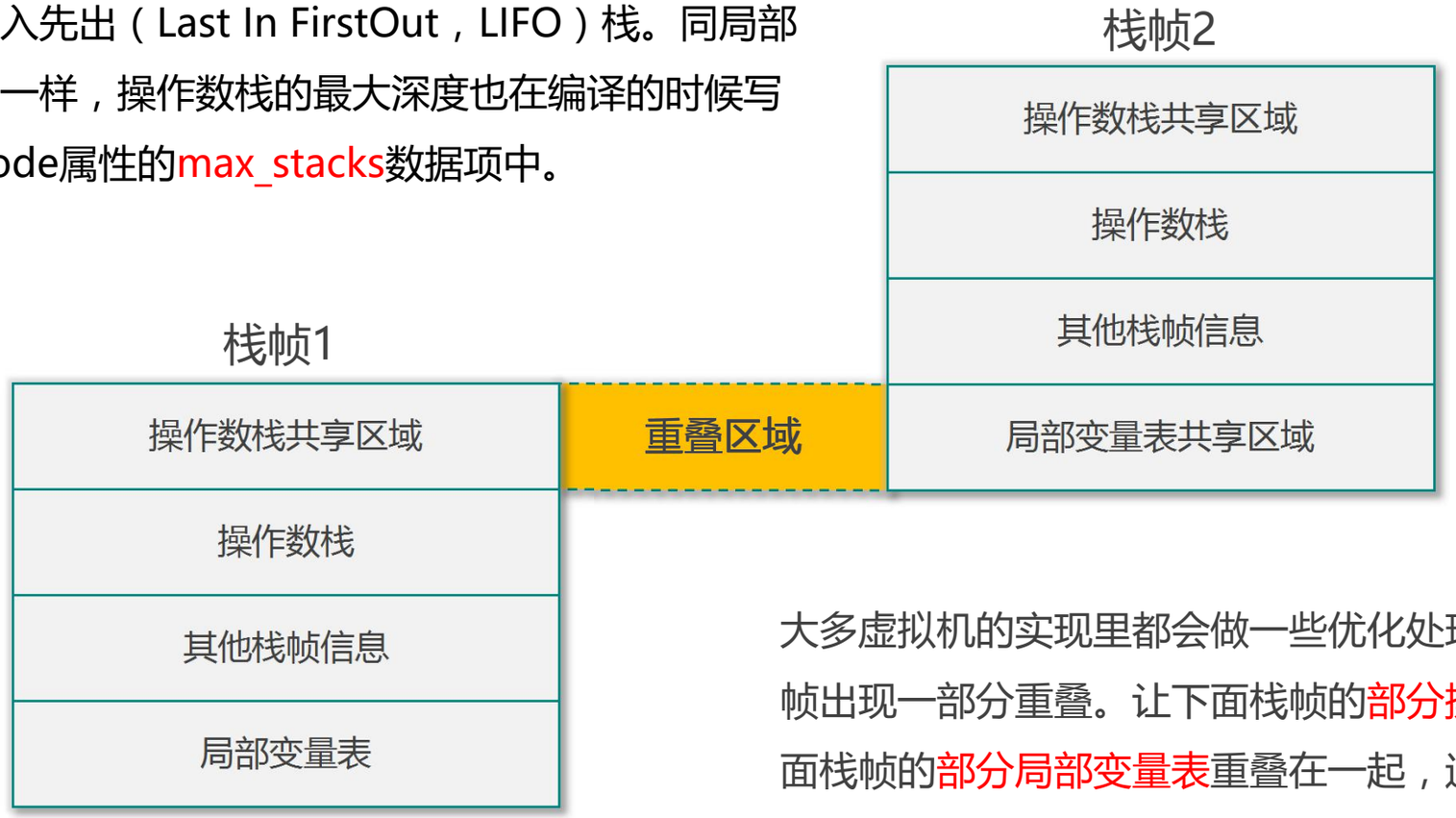
局部变量表的最小容量单位：变量槽（Variable Slot，简称Slot）

一个Slot可以存放一个32位以内的数据类型

对于64位的数据类型，虚拟机会以高位对齐的方式为其分配两个连续的Slot空间

操作数栈

操作数栈（Operand Stack）也常称为操作栈，它是一个后入先出（Last In FirstOut，LIFO）栈。同局部变量表一样，操作数栈的最大深度也在编译的时候写入到Code属性的max_stacks数据项中。

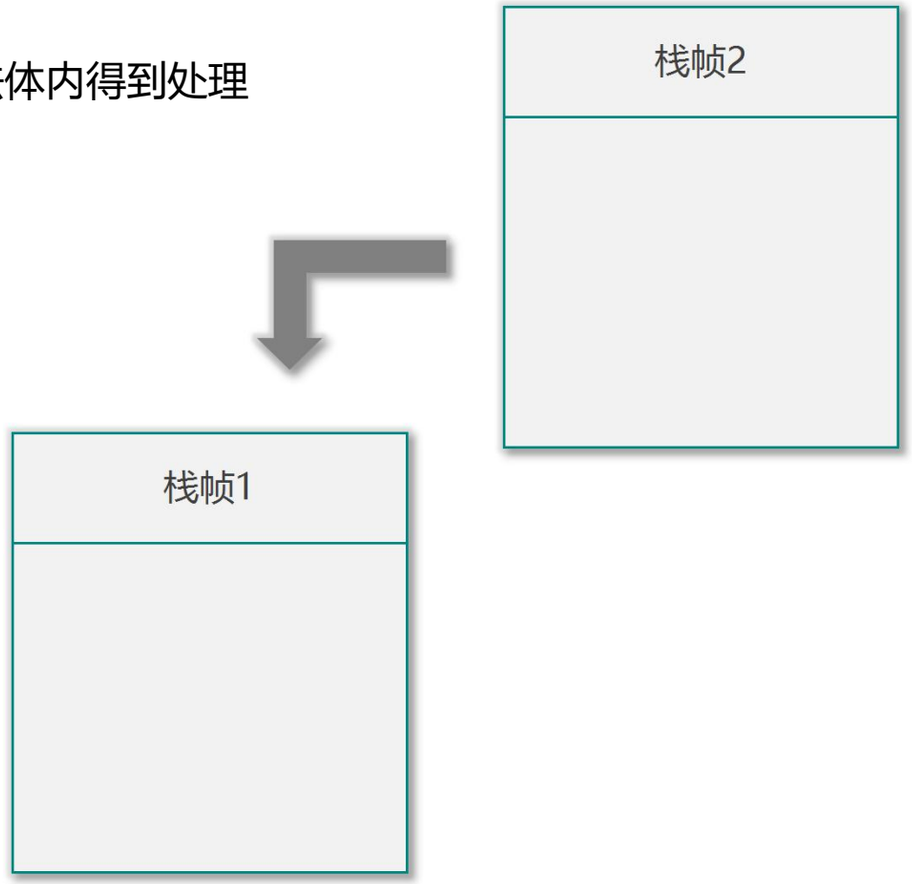


大多虚拟机的实现里都会做一些优化处理，令两个栈帧出现一部分重叠。让下面栈帧的部分操作数栈与上面栈帧的部分局部变量表重叠在一起，这样在进行方法调用时就可以共用一部分数据，无须进行额外的参数复制传递

方法返回地址

当一个方法开始执行后，只有两种方式可以退出这个方法。

- 第一种，执行引擎遇到任意一个方法返回的字节码指令
- 第二种，在方法执行过程中遇到了异常，并且这个异常没有在方法体内得到处理

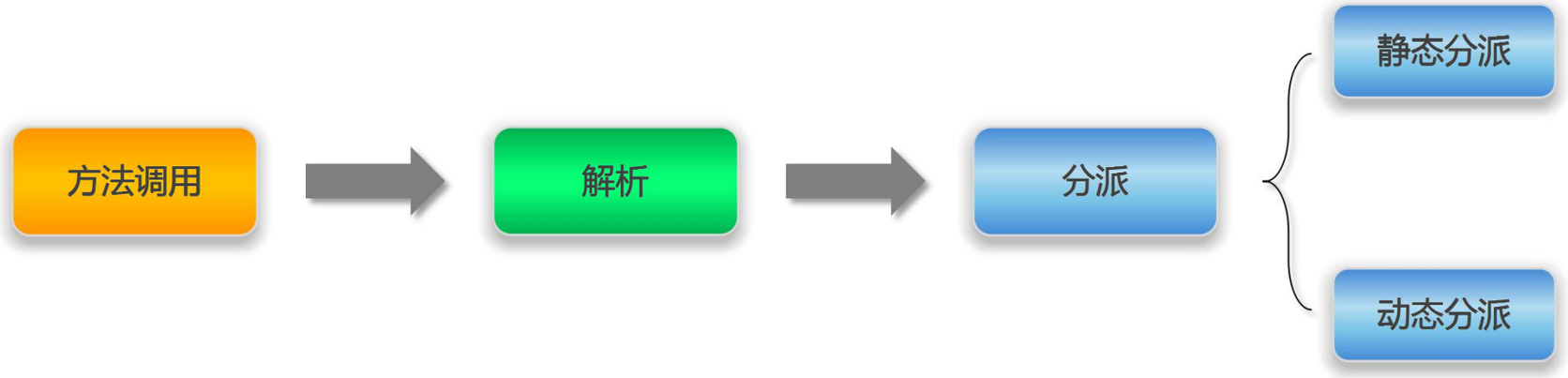


方法退出的过程实际上就等同于把当前栈帧出栈

- 恢复上层方法的局部变量表和操作数栈
- 把返回值（如果有的话）压入调用者栈帧的操作数栈中
- 调整PC计数器的值以指向方法调用指令后面的一条指令等

方法调用

方法调用并不等同于方法执行，该阶段唯一的任务就是确定调用哪一个方法。方法在实际运行时内存布局中的入口地址需要在类加载期间，甚至到运行期间才能确定。



- **非虚方法**

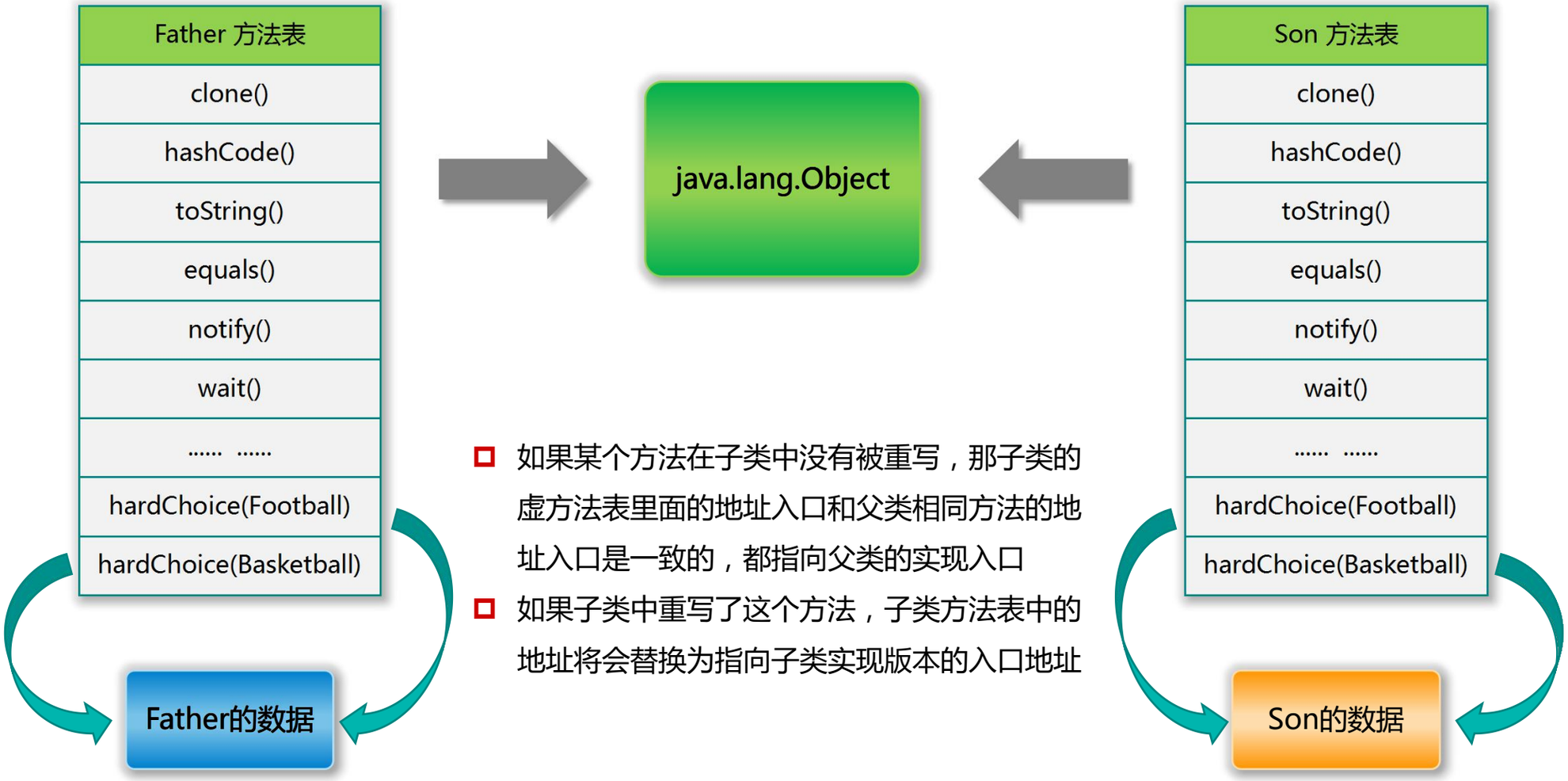
  在类加载的时候就会把符号引用解析为该方法的直接引用，在解析阶段中确定唯一的调用版本，有4类是非虚方法：静态方法、 私有方法、实例构造器、 父类方法

- **虚方法**

  除去final方法和虚方法，其他方法称为虚方法

虚拟机动态分派机制

虚方法表（Vritual Method Table，也称为vtable，与此对应的，在invokeinterface执行时也会用到接口方法表itable）使用虚方法表索引来代替元数据查找以提高性能。虚方法表中存放着各个方法的实际入口地址。



- 如果某个方法在子类中没有被重写，那子类的虚方法表里面的地址入口和父类相同方法的地址入口是一致的，都指向父类的实现入口
- 如果子类中重写了这个方法，子类方法表中的地址将会替换为指向子类实现版本的入口地址

02高级篇

字节码执行引擎

JDK性能监控与故障处理工具

字节码简介

方法调用

JDK的bin目录中有“java.exe”、 “javac.exe”这两个命令行工具，但bin目录之中还有很多其他命令行程序。这些工具被Sun公司作为“礼物”附赠给JDK使用者，这些工具非常强大可以用于监视虚拟机和故障处理。

工具名称	作用描述
jps.exe	JVM进程状态工具(JVM Process Status Tool)，用于显示目标系统上JVM的Java进程信息。
jstat.exe	JVM统计监测工具(JVM Statistics Monitoring Tool)，主要用于监测并显示JVM的性能统计信息。
jinfo.exe	Java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.exe	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.exe	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。
jstack.exe	Java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。
jmc.exe	Java任务控制工具(Java Mission Control)，主要用于JVM的生产时间监测、分析、诊断。
jvisualvm.exe	JVM监测、故障排除、分析工具，主要以图形化界面的方式提供运行于指定虚拟机的Java应用程序的详细信息。
jconsole.exe	图形化用户界面的监测工具，主要用于监测并显示运行于Java平台上的应用程序的性能和资源占用等信息。

虚拟机进程状况工具—jps

功能：显示正在运行的虚拟机进程

参数： -q　只显示LVMID，省略主类信息（LVMID：本地虚拟机进程唯一编号）

-l　显示虚拟机启动进程时传递给main()的参数

-m　显示类全面，如果是jar包显示jar路径

-v　显示虚拟机启动时候的JVM参数

用法：

```
D:\>jps -1
15268
17892 org.jetbrains.jps.cmdline.Launcher
6984 sun.tools.jps.Jps
6844 org.jetbrains.idea.maven.server.RemoteMavenServer
```

运行状态信息—jstat

功能：显示本地或者远程虚拟机进程中的类装载、 内存、 垃圾收集、 JIT编译等运行数据。是定位虚拟机性能问题的首选工具。

用法： jstat [-命令选项] [vmid] [间隔时间/毫秒] [查询次数]

命令选项	描述
-class	类加载、卸载数量、总空间及类装载所耗费的时间
-compiler	显示JIT编译器编译过的方法、耗时等信息
-gc	统计Java堆，包括Eden、Survivor、老年代、永久代的容量，已用空间、GC时间等信息
-gccapacity	显示Java堆各个区域使用到的最大、最小空间
-gcutil	显示已使用空间占总空间的百分比
-gccause	垃圾收集统计概述（同-gcutil），附加最近两次垃圾回收事件的原因
-gcnew	新生代行为统计
-gcnewcapacity	新生代使用到的最大、最小空间统计
-gcold	统计老年代GC状况
-gcoldcapacity	年老代行为统计（同-gcoldcapacity），主要关注使用到的最大、最小空间
-gcpermcapacity	显示永久代使用到的最大、最小空间（-gcmetacapacity）
-printcompilation	显示已经被JIT编译的方法

实时查看调整参数—jinfo

功能：可以用来查看正在运行的Java应用程序的扩展参数，甚至支持在运行时，修改部分参数

参数： -flag <name> pid：打印指定JVM的参数值

-flag [+|-]<name> pid：设置指定JVM参数的布尔值

-flag <name>=<value> pid：设置指定JVM参数的值

用法：
```
jinfo -flag +PrintGC pid
jinfo -flag +PrintGCDetails pid
jinfo -flag +PrintGCTimestamp pid

jinfo -flag -PrintGC pid
jinfo -flag -PrintGCDetails pid
jinfo -flag -PrintGCTimestamp pid
```

内存Dump工具—jmap

功能：用于生成heap dump文件，如果不使用这个命令，还可以使用-XX:+HeapDumpOnOutOfMemoryError参数来让虚拟机出现OOM的时候自动生成dump文件。 jmap不仅能生成dump文件，还可以查询finalize执行队列、Java堆和永久代的详细信息，如当前使用率、当前使用的是哪种收集器等

用法：jmap [-命令选项] [vmid]

命令选项	描述
-dump	生成Java堆快照。格式：-dump:[live,]format=b, file=<filename>，live为是否只生成存活的对象
-histo	显示堆中对象的统计信息，包括类、有多少个实例，合计容量等
-permstat	显示永久代内存状态。只在Linux/Solaris平台下有效
-heap	显示堆详细信息，如使用哪种回收器、参数配置、分代状况等。只在Linux/Solaris平台下有效
-finalizerinfo	显示在F-Queue中等待Finalizer线程执行finalize方法的对象。只在Linux/Solaris平台下有效
-F	当虚拟机进程多-dump没有响应时，可以使用这个选项强制生成dump快照。只在Linux/Solaris平台下有效

堆栈跟踪工具—jstack

功能：用于生成虚拟机当前时刻的线程快照，以便可以进一步定位线程出现长时间停顿的原因，如线程间死锁、死循环、 请求外部资源导致的长时间等待等。

用法：jstack [-命令选项] [vmid]

命令选项	描述
-F	当输出请求不被响应时，强制输出线程堆栈信息
-l	除堆栈信息外，附加显示关于锁的信息
-m	如果涉及本地方法调用，则显示C/C++的堆栈

02高级篇

字节码执行引擎

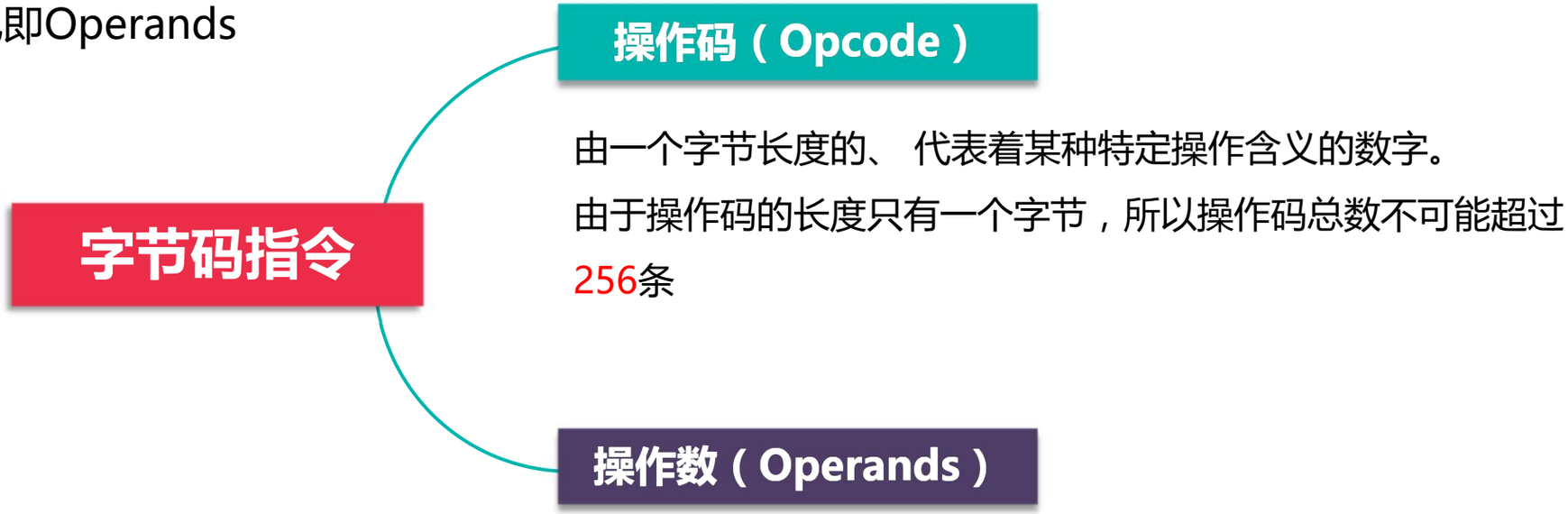
JDK性能监控与故障处理工具

字节码简介

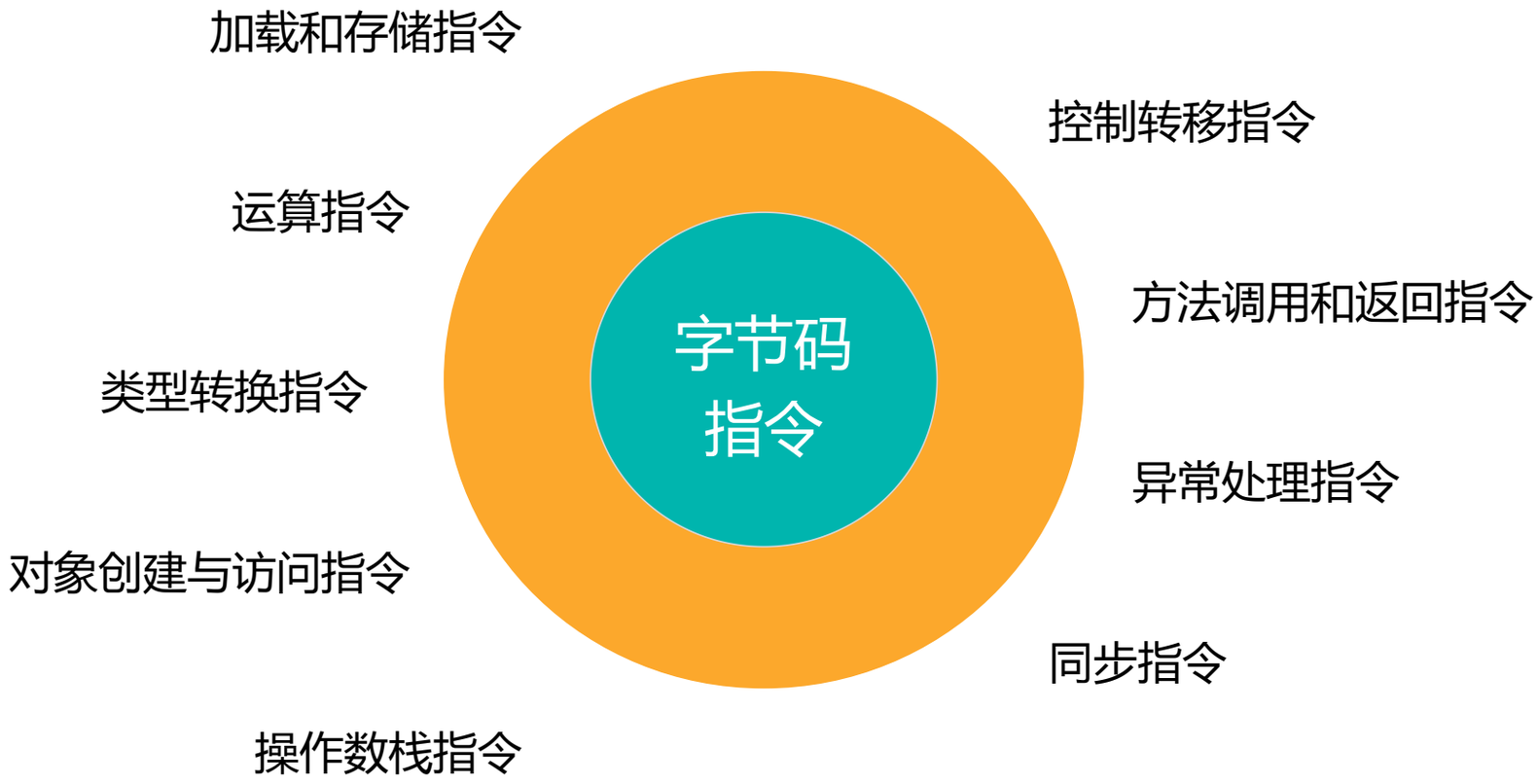
字节码指令

Java虚拟机的字节码指令由两部分组成

- 操作码，也即Opcode（最大256条）
- 操作数，也即Operands



字节码指令分类


加载和存储指令
控制转移指令
运算指令
方法调用和返回指令
字节码
指令
类型转换指令
异常处理指令
对象创建与访问指令
同步指令
操作数栈指令

字节码指令

将一个局部变量加载到操作栈	将一个数值从操作数栈存储到局部变量表	将一个常量加载到操作数栈
iload iload_<n> lload lload_<n> fload fload_<n> dload dload_<n> aload aload_<n>	istore istore_<n> lstore lstore_<n> fstore fstore_<n> dstore dstore_<n> astore astore_<n>	bipush sipush ldc ldc_w ldc2_w aconst_null iconst_m1 iconst_<i> lconst_<l> fconst_<f> dconst_<d>

加法指令：iadd、 ladd、 fadd、 dadd
减法指令：isub、 lsub、 fsub、 dsub
乘法指令：imul、 lmul、 fmul、 dmul
除法指令：idiv、 ldiv、 fdiv、 ddiv
求余指令：irem、 lrem、 frem、 drem
取反指令：ineg、 lneg、 fneg、 dneg

位移指令：ishl、 ishr、 iushr、 lshl、 lshr、 lushr
按位或指令：ior、 lor
按位与指令：iand、 land
按位异或指令：ixor、 lxor
自增指令：iinc
比较指令：dcmpg、 dcmpl、 fcmpg、 fcmpl、 lcmp


云析学院
2018
感谢聆听 批评指导