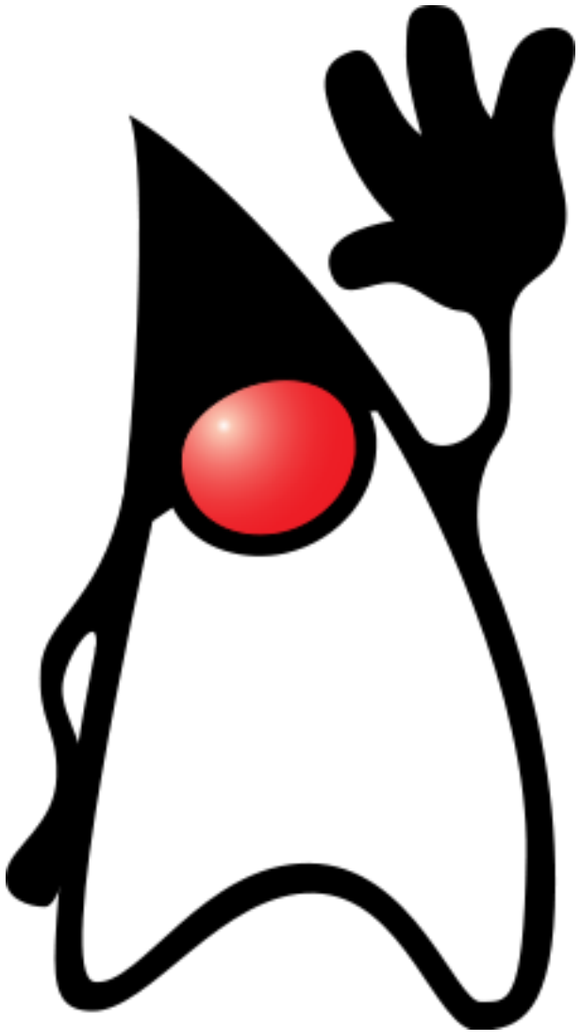


Java虚拟机基础

温绍锦

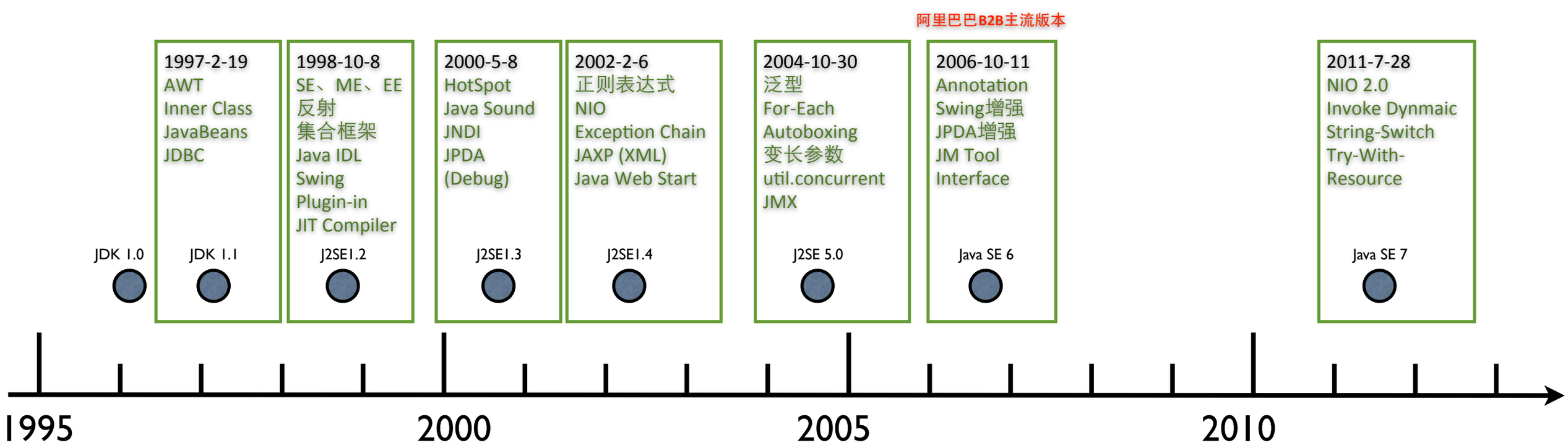
提纲

- HotSpot
- ClassFile
- ClassLoader
- 内存模型、锁、同步
- JVM内存管理和垃圾收集

HotSpot介绍

- Java发展历程
- JVM列表
- OpenJDK
- 编译执行过程
- 解析执行和JIT编译

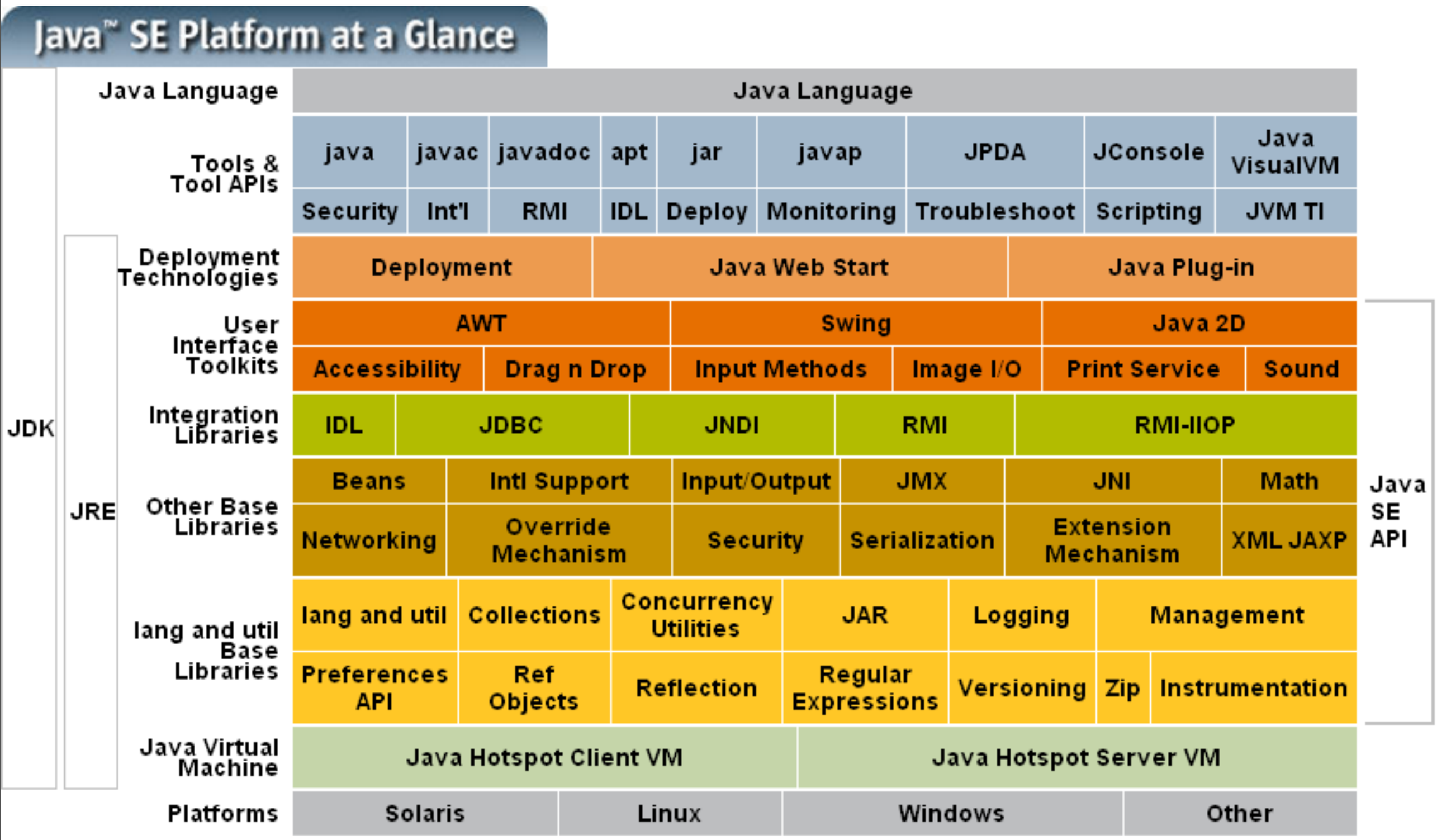
Java发展历程



1995年，Sun发布Java 1.0，承诺：

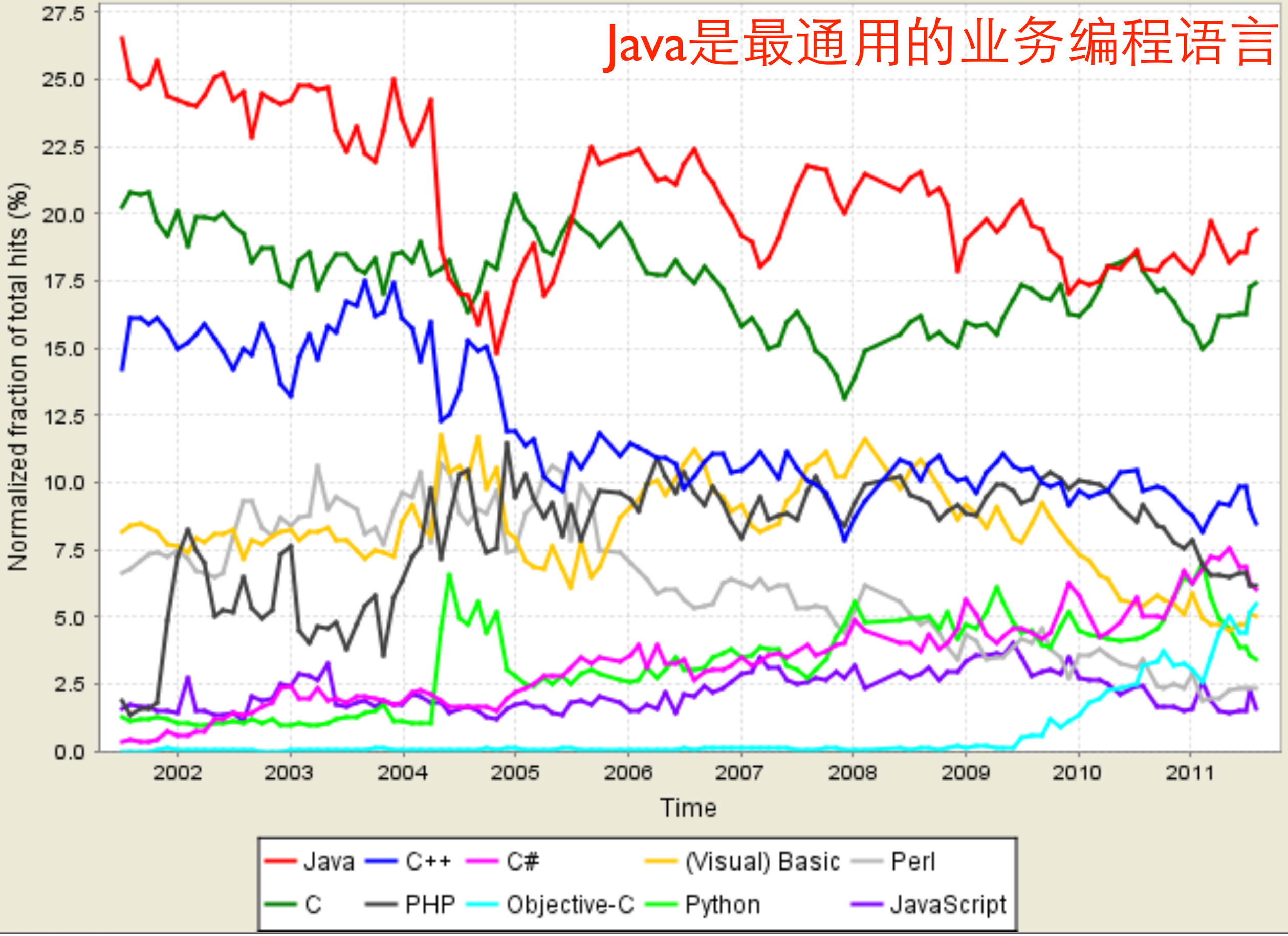
Write Once，Run Anywhere

Java平台


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

Tiobe Programming Community Index

Java是最通用的业务编程语言



Java虚拟机列表

虚拟机	描述
Oracle HotSpot	原来属于SUN，SUN被Oracle收购之后属于Oracle，是目前最流行的JVM
Oracle JRockit	原来属于BEA，BEA被Oracle收购之后属于Oracle，拥有一些优秀特性，将会和HotSpot合并
IBM J9	IBM的JDK
Apple Mac OS Runtime for Java	Apple公司开发的虚拟机，运行在Mac OS X系统上
Apache Harmony	Apache组织开发的虚拟机，基于Apache License 2.0
Dalvik	Google实现的JVM，Android的虚拟机。
Maxine	Oracle的一个用Java编写的Java虚拟机，用于研究目的
其它	有很多虚拟机的实现 http://en.wikipedia.org/wiki/List_of_Java_virtual_machines

合并

HotSpot

- Oracle(SUN)的JVM实现
- 主要用C++实现
- 解析器和编译器混合执行模式
- 默认解析执行，对执行频率高（热点）的代码做动态编译
- 2006年开源

OpenJDK

- Sun在2006年11月13日把**HotSpot**及编译器通过GPL协议开源，称为OpenJDK
- 这是自由软件社区的重要里程碑
- 加入OpenJDK的厂商包括：Oracle、IBM、Apple、SAP
- 支持的操作系统包括：Windows、Linux、Solaris、BSD、MacOS、Haiku。
- 支持的硬件体系架构包括：x86、adm64、sparc、PowerPC、mips、IA64、ARM

HotSpot包括：

- 一个ByteCode Interpreter
- 两个 JIT Compiler：
  - C1 (client编译器)
    - 轻量级
    - 编译时间更短，占用内存少，适合GUI
  - C2 (server编译器)
    - 重量级
    - 执行效率更高，大量编译优化，适合服务器

HotSpot三种执行模式

```
$ java -version
java version "1.6.0_26"
Java(TM) SE Runtime Environment (build 1.6.0_26-b03-383-11A511)
Java HotSpot(TM) 64-Bit Server VM (build 20.1-b02-383, mixed mode)
```

解析和编译混合模式

```
java -Xint -version
java version "1.6.0_26"
Java(TM) SE Runtime Environment (build 1.6.0_26-b03-383-11A511)
Java HotSpot(TM) 64-Bit Server VM (build 20.1-b02-383, interpreted mode)
```

纯解析模式

```
java -Xcomp -version
java version "1.6.0_26"
Java(TM) SE Runtime Environment (build 1.6.0_26-b03-383-11A511)
Java HotSpot(TM) 64-Bit Server VM (build 20.1-b02-383, compiled mode)
```

纯编译模式

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

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

一些关于JIT的参数

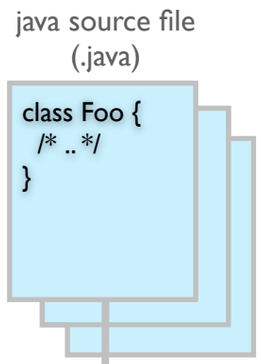
Option	Feature
-Xint	jvm选择纯解析模式，缺省是混合模式
-Xcomp	jvm选择纯编译模式，缺省是混合模式
-XX:+AggressiveOpts	采用激进的优化办法
-XX:+CompileThreshold=1000	判断是否热点进行编译的调用次数
-XX:+CITime	输出JIT编译所耗费的时间
-XX:+PrintCompilation	当一个方法被编译时，打印信息。这个参数对于调优基础组件有用。
-XX:InlineSmallCode -XX:MaxInlineSize=35 -XX:FreqInlineSize=	代码内联的判断依据，调整编译后代码的字节大小是判断条件
-XX:LoopUnrollLimit=	编译优化时将循环展开的上限值

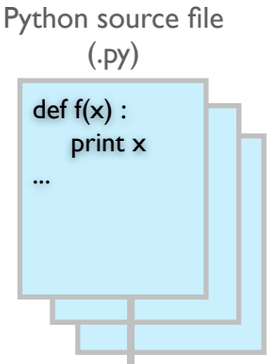
Class文件格式

- Java编译执行流程
- ClassFile的格式介绍
- ClassFile中FieldInfo和MethodInfo介绍
- 类型描述Descriptor介绍
- ClassFile中的Attribute介绍
- JVM指令介绍

编译执行流程

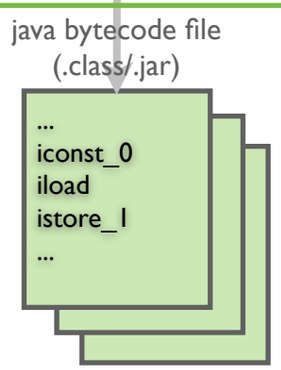
源码

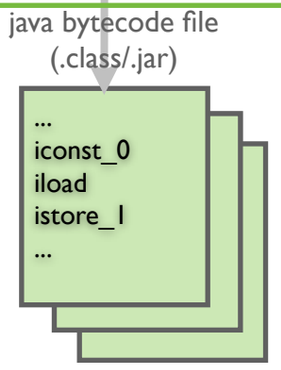




javac

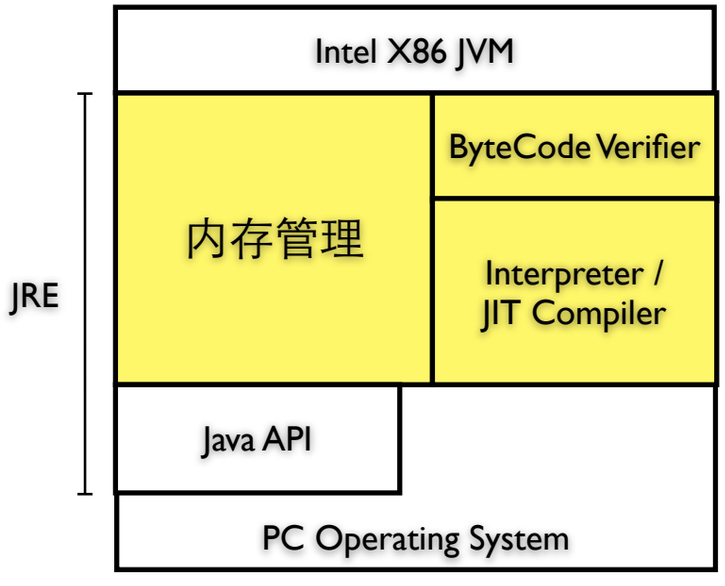
jython

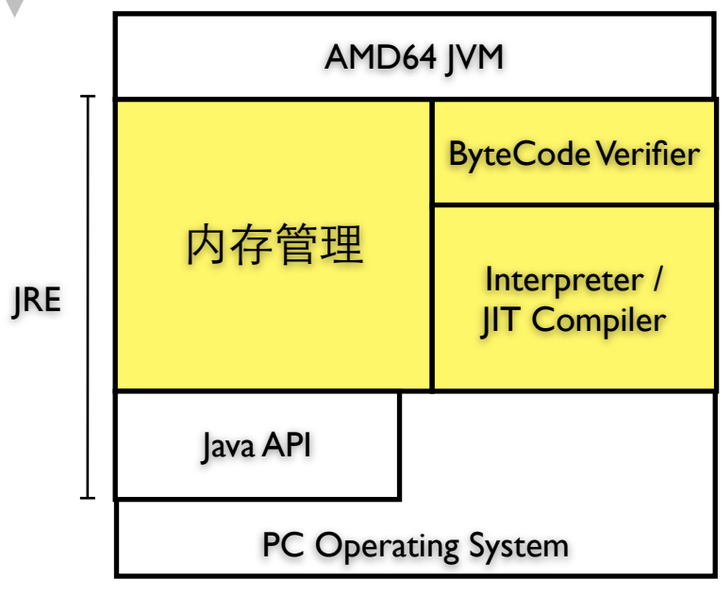




字节码是实现跨平台的基础

执行





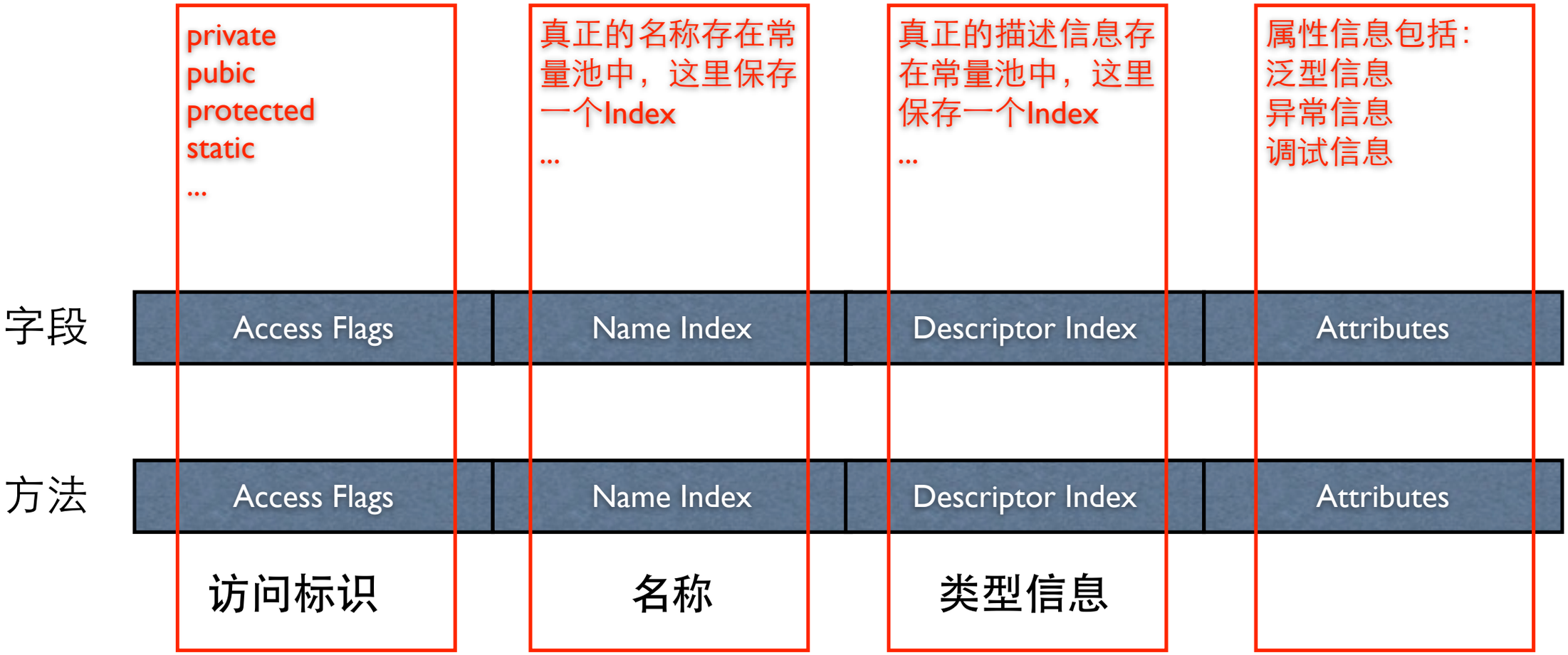
ClassFileFormat

魔数，值为0xCAFEBABE，Java创始人James Gosling制定

包括minor_version和major_version，minor_version：1.1(45), 1.2(46), 1.3(47), 1.4(48), 1.5(49), 1.6 (50), 1.7(51)。指令集多年不变，但是版本号每次发布都变化。

包括字符串常量、数值常量等

字段和方法


private
pubic
protected
static
...
真正的名称存在常量池中，这里保存一个Index
...
真正的描述信息存在常量池中，这里保存一个Index
...
属性信息包括：
泛型信息
异常信息
调试信息
字段
Access Flags
Name Index
Descriptor Index
Attributes
方法
Access Flags
Name Index
Descriptor Index
Attributes
访问标识
名称
类型信息

Descriptor

	Type	
B	byte	带符号的byte
C	char	Unicode字符
D	double	双精度浮点数
F	float	单精度浮点数
I	int	32位整数
J	long	64位整数
S	short	16位整数
Z	boolean	true or false
LClassname;	reference	引用类型， Ljava/lang/String;表示String Ljava/lang/Integer;表示Integer
[	reference	数组类型，例如： [I表示int[]， [Ljava/lang/Object;表示Object[]， [[[D表示double[][][]

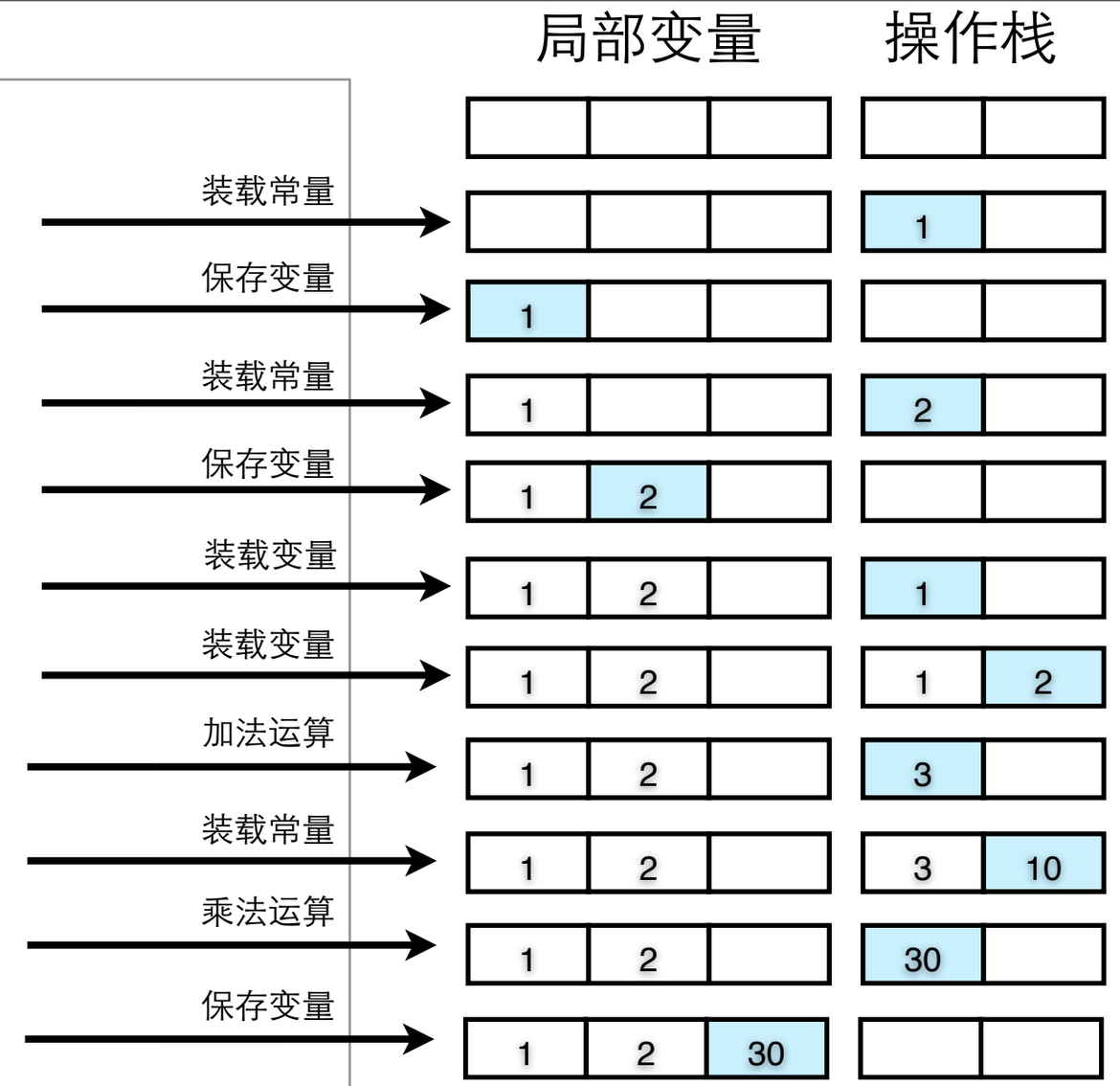
Attribute类型	使用范围	
ConstantValue	FieldInfo	定义字段的初始值
Code	MethodInfo	方法的代码
StackMapTable	CodeAttribute	调试信息
Exception	MethodInfo	异常信息
InnerClass	ClassFile	内嵌类
EnclosingMethod	ClassFile	匿名类
Synthetic	ClassFile/MethodInfo/FieldInfo	缺省构造函数等
Signature	ClassFile/MethodInfo/FieldInfo	泛型信息
SourceFile	ClassFile	源码信息
SourceDebugExtension	ClassFile	调试信息
LineNumberTable	CodeAttribute	调试信息
LocalVariableTable	CodeAttribute	调试信息
LocalVariableTypeTable	CodeAttribute	调试信息
Deprecated	ClassFile/MethodInfo/FieldInfo	
RuntimeVisibleAnnotations	ClassFile/MethodInfo/FieldInfo	Annotation信息
RuntimeInvisibleAnnotations	ClassFile/MethodInfo/FieldInfo	Annotation信息
RuntimeVisibleParameterAnnotations	MethodInfo	Annotation信息
RuntimeInvisibleParameterAnnotations	MethodInfo	Annotation信息
AnnotationDefault	MethodInfo	Annotation信息

```
public static int f() {
    int a = 1;
    int b = 2;
    int c = (a + b) * 10;
    return c;
}
```

```
public static f()I
    ICONST_1
    ISTORE 0
    ICONST_2
    ISTORE 1
    ILOAD 0
    ILOAD 1
    IADD
    BIPUSH 10
    IMUL
    ISTORE 2
    ILOAD 2
    IRETURN
MAXSTACK = 2
MAXLOCALS = 3
}
```

操作栈大小 →

局部变量区大小 →



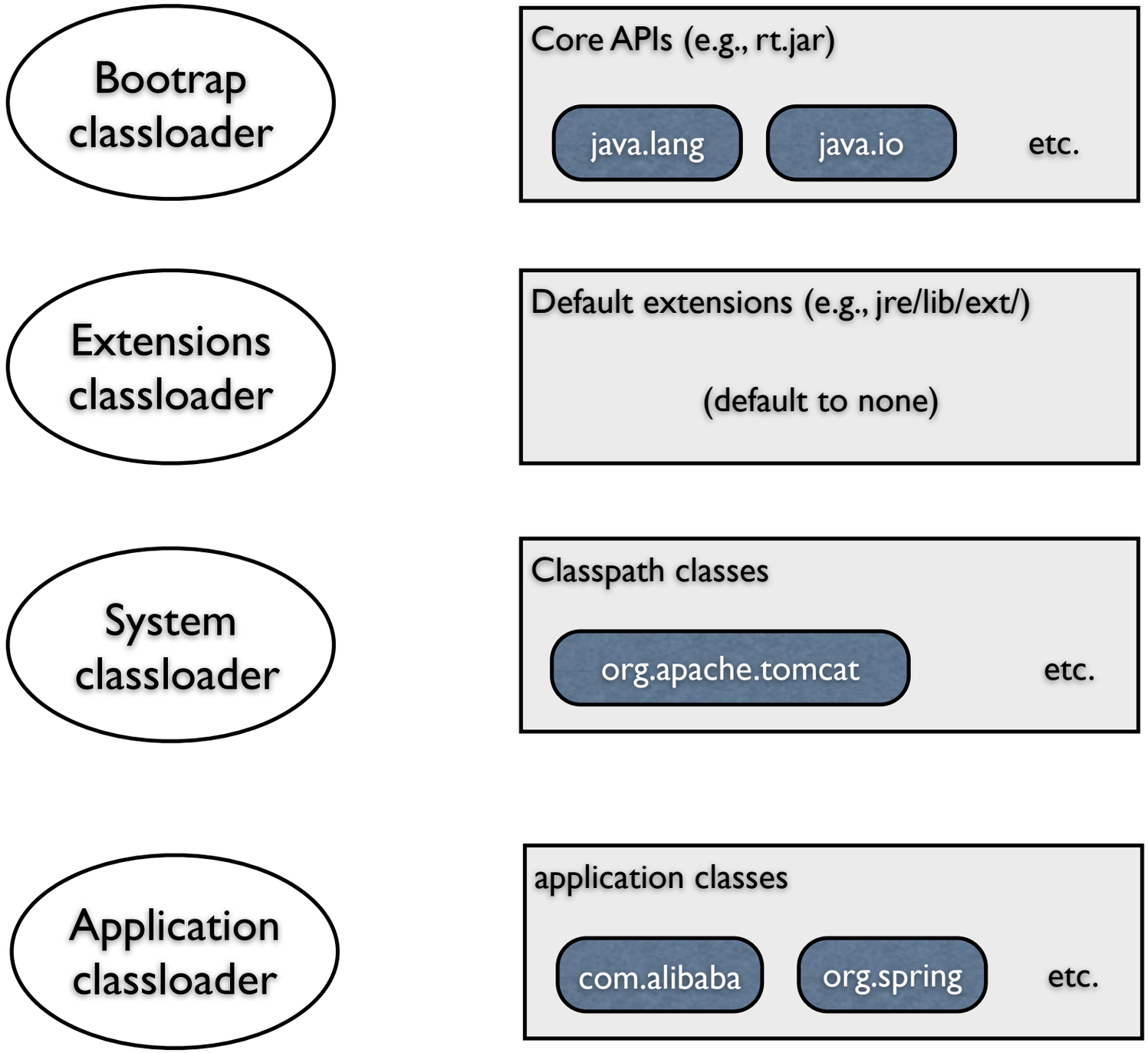
注意：double和long类型会占据两个栈位

装载和存储指令	从局部变量装载到操作栈	iload/lload/fload/dload/aload
	将操作栈保存到局部变量	istroe/lstore/fstore/dstore/astore
	装载常量到操作栈	bipush/sipush/ldc/ldc_w/aconst_null
算术运算指令	加	iadd/ladd/fadd/dadd
	减	isub/lsub/fsub/dsub
	乘	imul/lmul/fmul/dmul
	除	idiv/ldiv/fdiv/ddiv
	求余	irem/lrem/frem/drem
	负数	ineg/lneg/fneg/dneg
	位移	ishl/ishr/iushr/lshl/lshr/lushr
	位操作	ior/lor/iand/land/ixor/lxor
	自增	iinc
	比较	dcmpg/dcmpl/fcmpg/fcmpl/lcmp
类型转换	数值类型转换	i2b/i2c/i2s/l2i/f2i/f2l/d2i/d2l/d2f
对象创建和处理	创建对象	new
	创建数组	newarray/anewarray/mulanewarray
	访问字段	getfield/putfield/getstatic/putstatic
操作栈指令		pop/pop2/dup/dup2/dup_x1/dup_x2/dup2_x1/dup2_x2/swap
跳转指令	条件跳转	ifeg/iflt/ifle/ifne/ifgt/ifge/ifnull/ifnonnull if_icmpeg/if_icmpne/if_icmplt/if_icmpgt/if_icmple/if_acmpeq/if_acmpne
	组合跳转	tableswitch/lookupswitch
	无条件跳转	goto/goto_w/jsr/jsr_w/ret
方法调用		invokevirtual/invokeinterface/invokespecial/invokestatic 方法调用 return/ireturn/lreturn/freturn/dreturn/areturn 返回
同步		monitorenter/monitorexit

ClassLoader

- 各种ClassLoader介绍
- ClassLoader工作机制
- Thread.getContextClassLoader()
- Jar Hell问题以及解决办法

各种ClassLoader介绍



获得ClassLoader的途径

获得当前类的**ClassLoader**

```
clazz.getClassLoader();
```

获得当前线程上下文的**ClassLoader**

```
Thread.currentThread().getContextClassLoader()
```

获得系统的**ClassLoader**

```
ClassLoader.getSystemClassLoader()
```

获得调用者的**ClassLoader**

```
DriverManager.getCallerClassLoader()
```

Jar hell问题以及解决办法

当一个类或者一个资源文件存在多个jar中，就会存在jar hell问题。

可以通过以下代码来诊断问题：

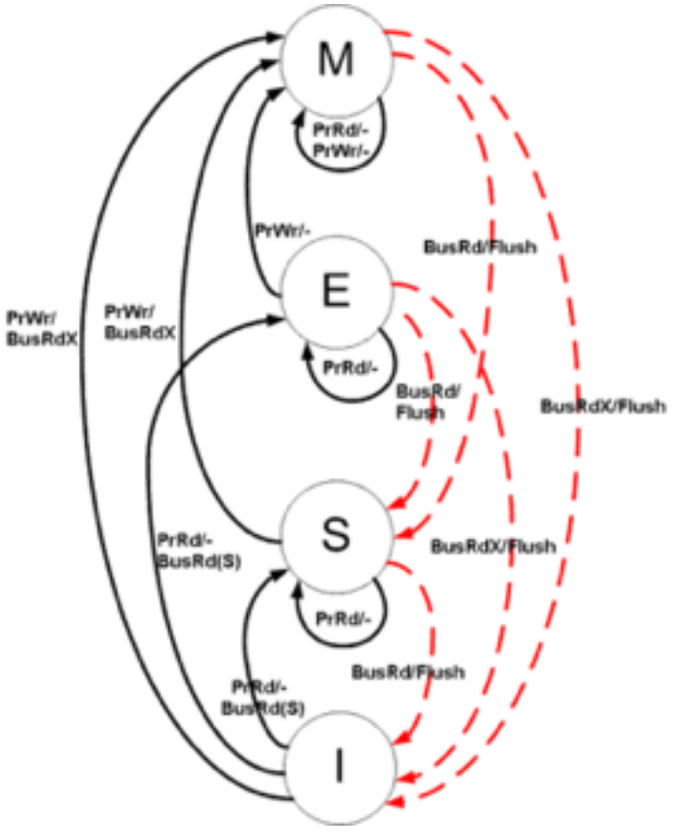
```
ClassLoader classLoader = Thread.currentThread().getContextClassLoader();
String reosurceName = "com/alibaba/simpleEL/dialect/tiny/TinyELEvalService.class";
Enumeration<URL> urls = classLoader.getResources(reosurceName);
while (urls.hasMoreElements()) {
    URL url = urls.nextElement();
    System.out.println(url);
}
```

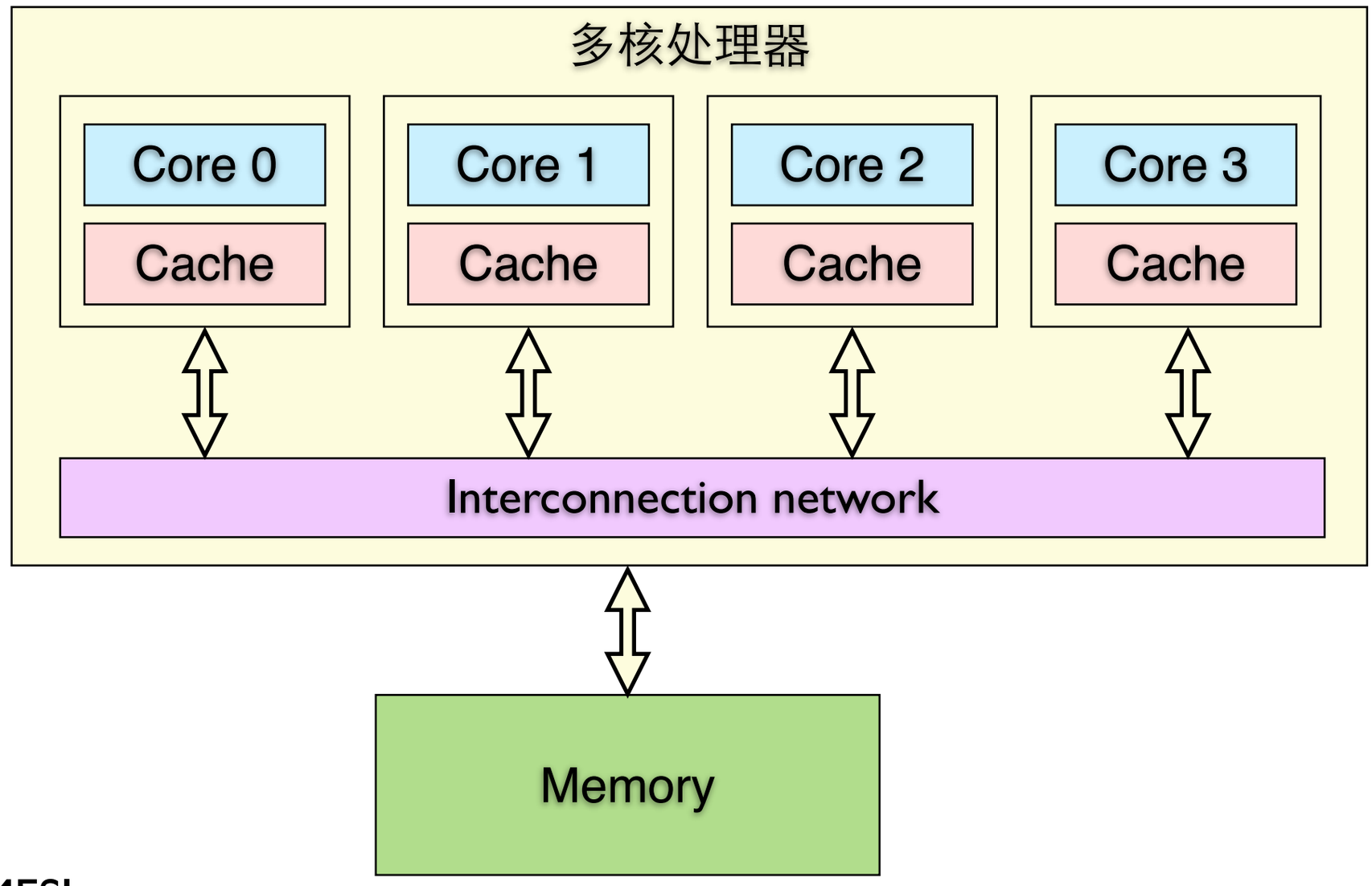
```
jar:file:/Users/admin/.m2/repository/com/alibaba/platform/shared/simpleel/0.1.2/simpleel-0.1.2.jar!/com/alibaba/simpleEL/dialect/tiny/TinyELEvalService.class
```

内存模型、锁和同步

- 多核处理器模型
- volatile
- cas
- synchronized、lock

多核处理器内存模型





多核处理器缓存一致性协议MESI

状态	描述
M(Modified)	这行数据有效，数据被修改了，和内存中不一致，数据只存在于本Cache中
E(Exclusive)	这行数据有效，数据和内存中的一致，数据只存在本Cache中
S(Shared)	这行数据有效，数据和内存中的一致，数据存在多分Cache中
I(Invalid)	这行数据无效

volatile

- 如果不声明volatile，变量装载到本地变量中，或者cpu cache中，多线程下很容易导致状态不一致。
- 声明了volatile，每次访问的都是主存中的数据，一致性能提升，但是还是不可靠的。
- volatile字段的访问效率很低，每次访问都需要十几个nano。大约为lock的1/3时间

CAS（Compare And Swap）

- CAS指令由硬件提供
- 并发程序设计实现的基础
- 486之后并不需要锁总线
- 基于MESI缓存一致性协议

http://blogs.oracle.com/dave/entry/biased_locking_in_hotspot

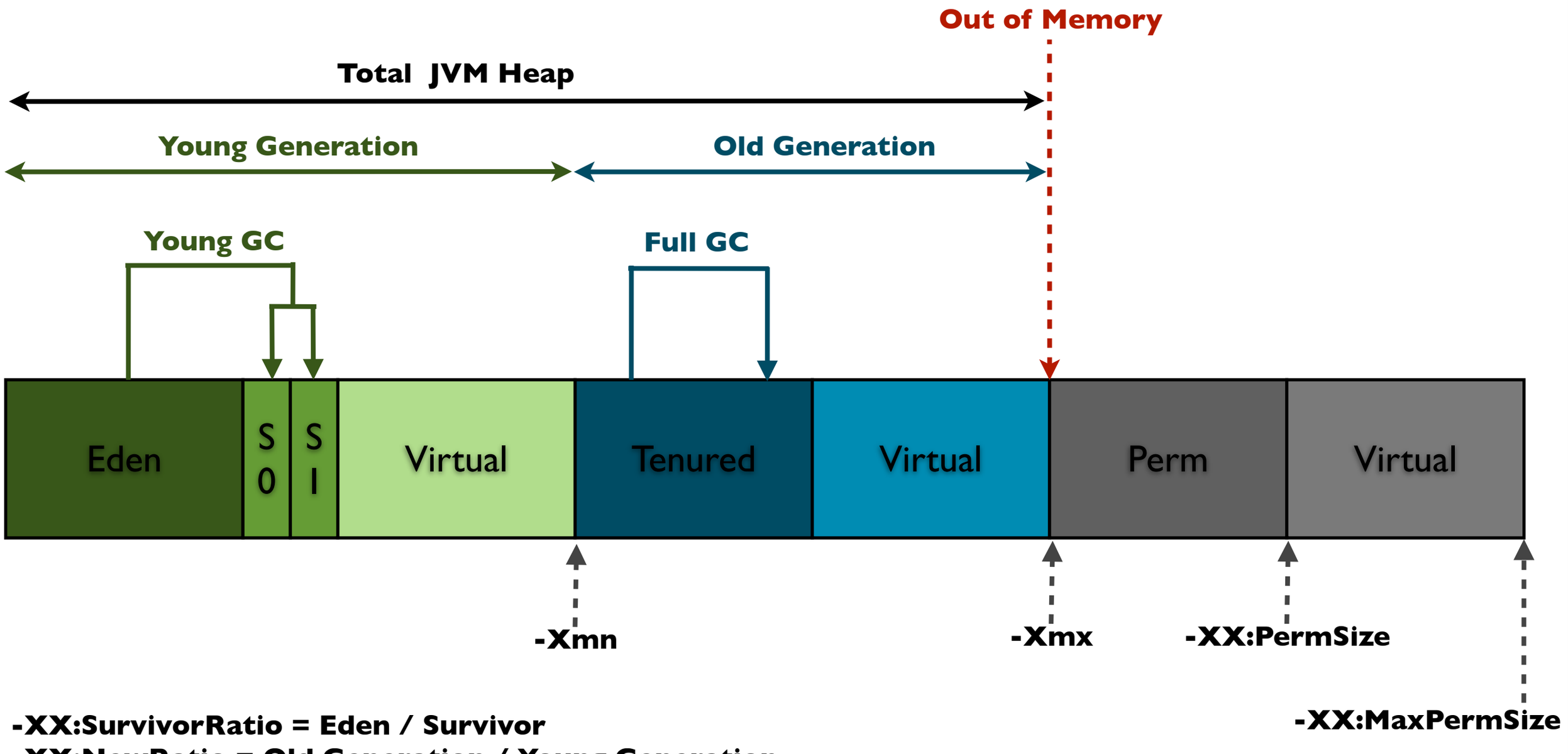
lock和synchronized

- 保证代码块的不可重入
- 底层都是基于CAS实现的
- 通过jstack -l <pid>获得jvm的线程信息和锁信息
-

JVM内存管理和垃圾收集

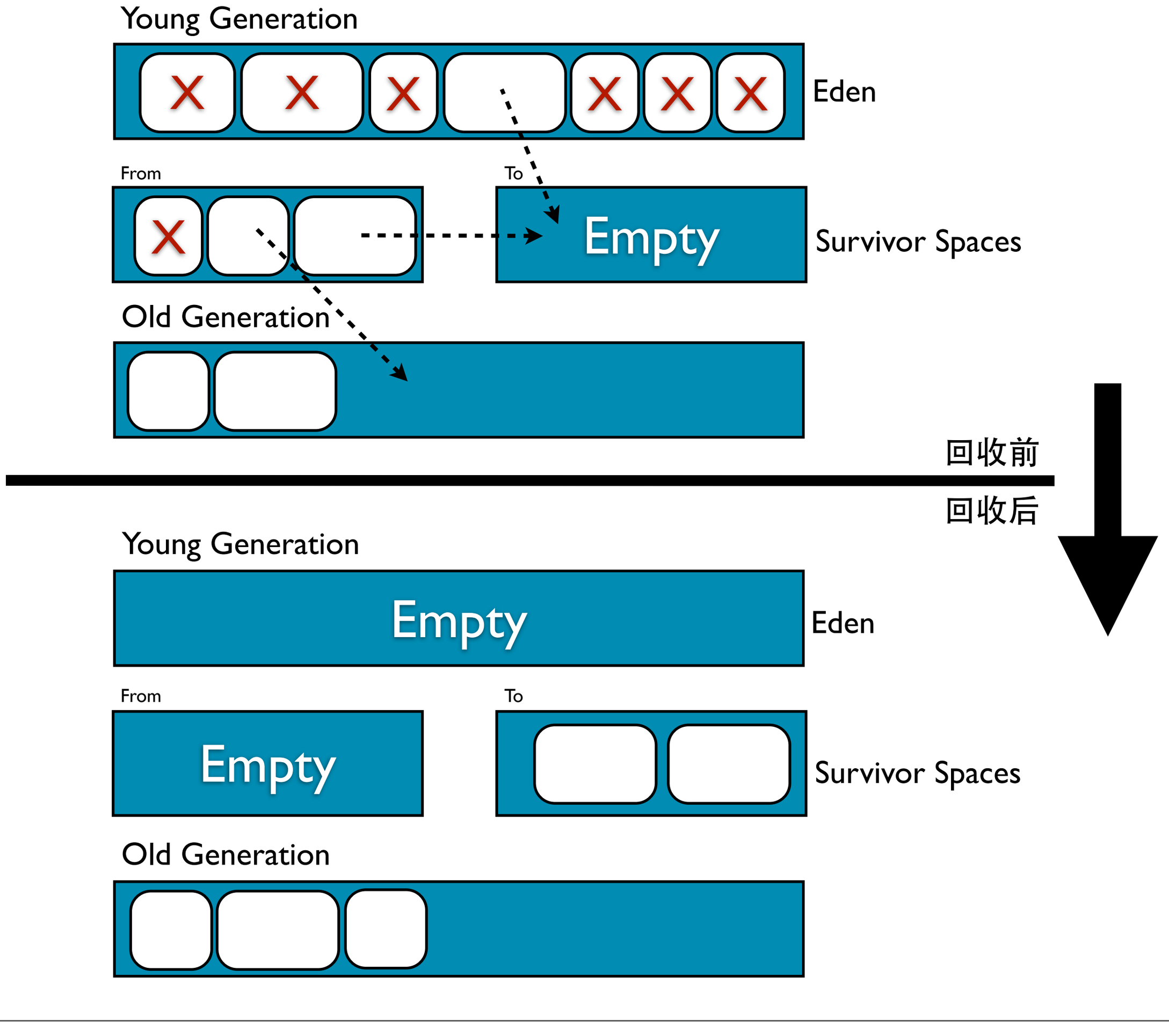
- 内存管理
- 垃圾收集
  - 新生代的垃圾收集
  - 老生代的垃圾收集
  - 不同垃圾收集的行为
  - 垃圾收集器选择

内存管理



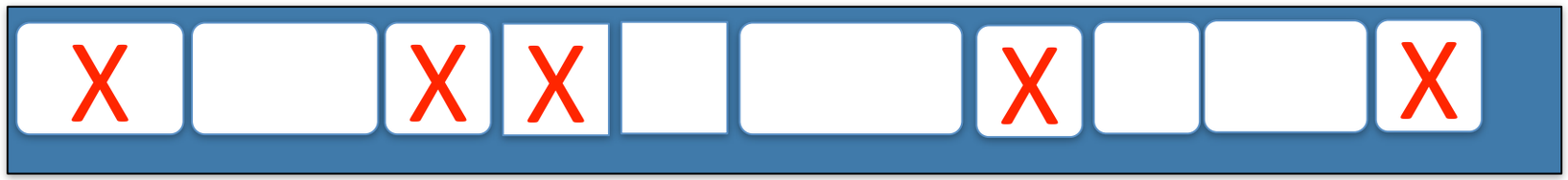
-XX:SurvivorRatio = Eden / Survivor
-XX:NewRatio = Old Generation / Young Generation

垃圾回收


Young Generation
Eden
From
To
Empty
Survivor Spaces
Old Generation
回收前
回收后
Young Generation
Empty
Eden
From
To
Empty
Survivor Spaces
Old Generation

Old Generation Gargage Collector

Start of Compaction



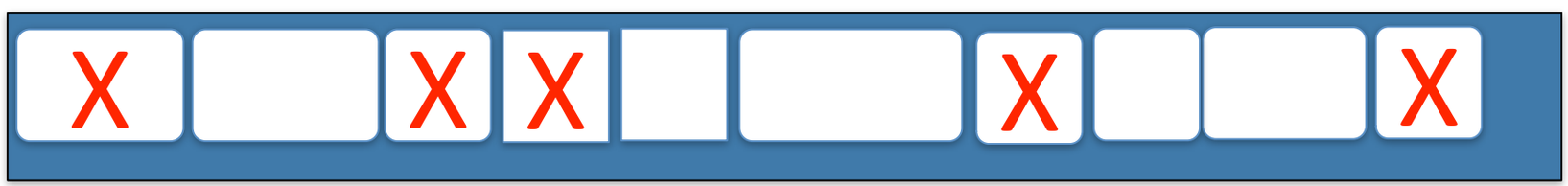
End of Compaction

Compaction of the old generation

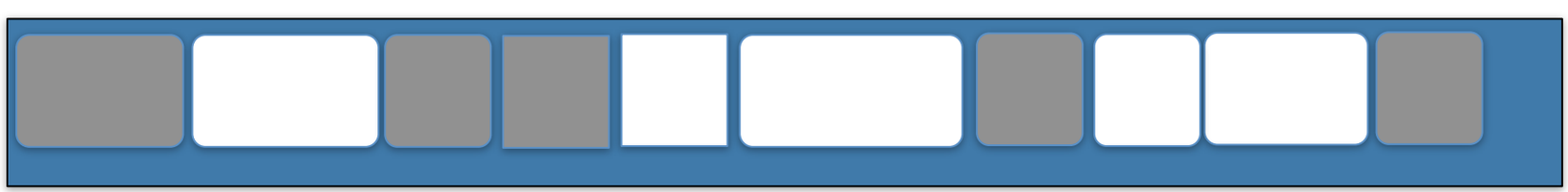
Serial Collector

Start of Sweeping

Concurrent Mark Sweep Collector

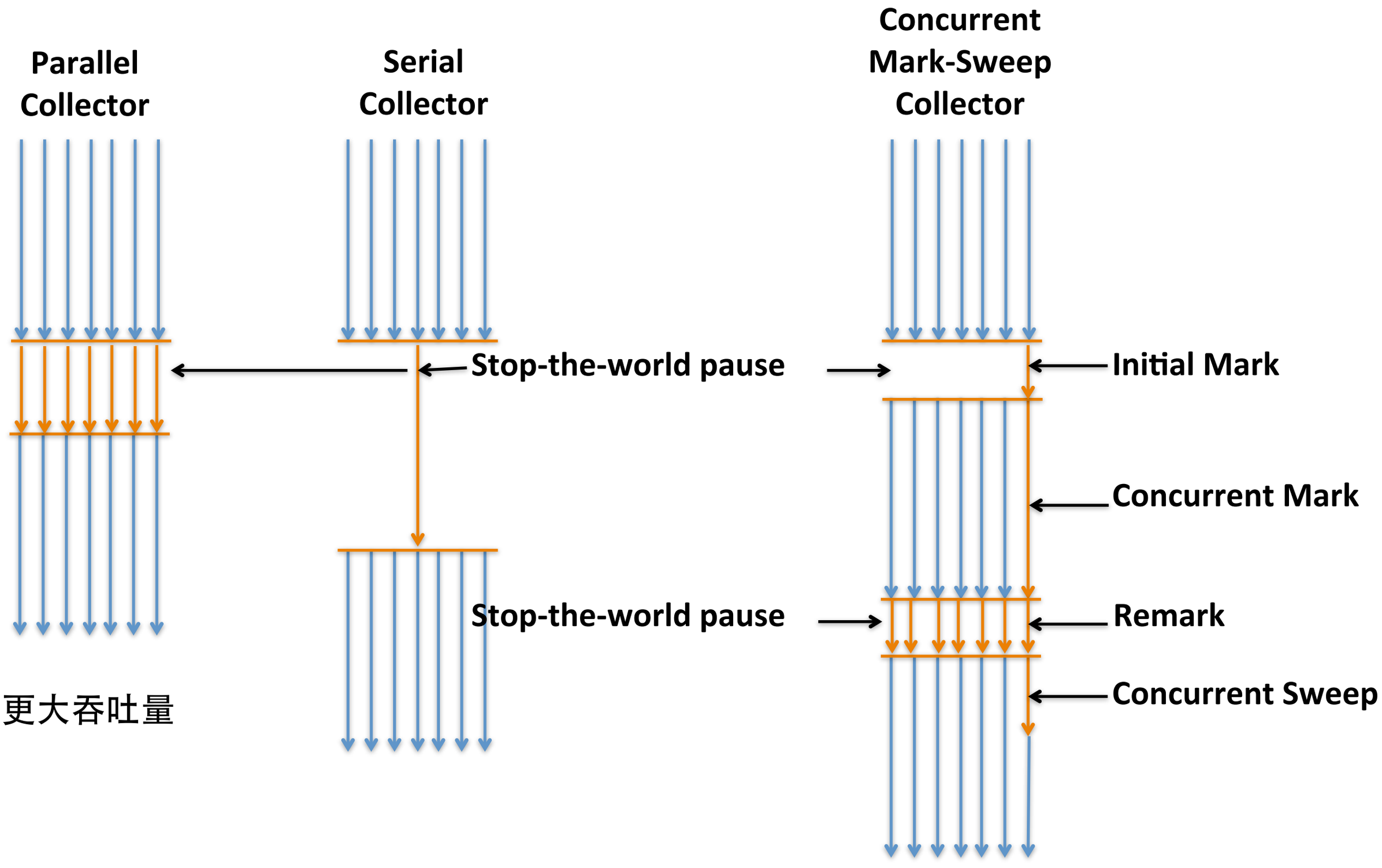


End of Sweeping



CMS sweeping (but not compacting) of old generation

各种GC行为比较


Parallel Collector
Serial Collector
Concurrent Mark-Sweep Collector
Stop-the-world pause
Stop-the-world pause
Initial Mark
Concurrent Mark
Remark
Concurrent Sweep
更大吞吐量
更短的停顿

Garbage Collectior Selection

Option	Garbage Collector Selected	Introducted	Feature
-XX:+UseSerialGC	Serial + Serial Old		
-XX:+UseParNewGC	ParNew + Serial Old		
-XX:+UseParallelGC	Parallel Scavenge + Serial Old	JDK 1.5	多线程
-XX:+UseParallelOldGC	Parallel Scavenge + Parallel Old	JDK 1.6	多线程，更大吞吐量
-XX:+UseConcMarkSweepGC	ParNew + CMS + Serial Old	JDK 1.6	更短停顿
-XX:+UnLockExperimentalVMOption -XX:+UseG1GC	Garbage First	JDK 1.7	大内存， 更短停顿，软实时