Java虚拟机（JVM）

JVM的组成部分

1. **类加载器（Class Loader）：** 类加载器负责将字节码文件加载到内存中，并将其转换为 JVM 内部的数据结构，即类的元数据。类加载器按照一定的层次结构进行工作，常见的有启动类加载器、扩展类加载器和应用类加载器。它们协同工作，保证类的加载是有序和层次化的。
2. **执行引擎（Execution Engine）：** 执行引擎负责执行经过加载的字节码指令。它将字节码翻译成机器码，然后由底层的操作系统和硬件执行。JVM 的执行引擎采用不同的执行模式，如解释执行和即时编译（JIT）执行，以提高执行效率。
3. **运行时数据区域（Runtime Data Areas）：** 运行时数据区域是 JVM 内存的划分，用于存储不同类型的数据和对象。主要包括：
    - **堆（Heap）：** 用于存储对象实例和数组，是 JVM 中最大的内存区域。被所有线程共享，用于存储动态分配的对象。堆分为新生代和老年代，用于垃圾回收的目的。
    - **方法区（Method Area）：** 用于存储类的元数据、静态变量、常量池等信息。方法区也被称为永久代，但在较新的 JDK 版本中，永久代被元空间所取代。
    - **栈（Stack）：** 每个线程在运行时都会创建一个栈，用于存储局部变量、方法参数、方法调用和返回信息。栈中的每个元素称为帧，对应一个方法调用。
    - **本地方法栈（Native Method Stack）：** 与栈类似，用于存储本地方法（由本地语言编写的方法）的调用和返回信息。
    - **程序计数器（Program Counter）：** 每个线程都有一个程序计数器，用于指示下一条将要执行的指令。
4. **本地方法接口（Native Interface）：** 本地方法接口允许 Java 代码调用本地库中的方法，以实现与底层操作系统和硬件的交互。JNI（Java Native Interface）是 Java 提供的一套标准接口。

方法区

1. **存储类的元数据：** 方法区存储了每个类的结构信息，包括类的名称、父类、接口、字段和方法等。这些元数据在类加载时被存储，供运行时使用。
2. **存储常量池：** 常量池包含类、接口、字段和方法的符号引用、字面量和其他常量。方法区会存储每个类的常量池，在运行时可以用于解析符号引用、初始化字段等。
3. **存储静态变量：** 静态变量（类变量）存储在方法区中。这些变量在类加载时被创建，被所有实例共享。
4. **存储方法代码：** 类的字节码指令存储在方法区中，包括方法的字节码、操作数栈、异常处理表等。这些信息用于方法的执行。
5. **存储运行时常量池：** 运行时常量池是在类加载后根据常量池生成的，用于在运行时支持动态方法调用、反射等。

隐式加载和显式加载

隐式加载：指的是程序在使 用 new 等方式创建对象时，会隐式地调用类的加载器把对应的类 加载到 JVM 中。

显式加载：指的是通过直接调用 class.forName() 方法来把所需的类加载到 JVM 中。

类加载器

1. **Bootstrap Class Loader（启动类加载器）**：它是Java虚拟机（JVM）的一部分，负责加载JVM运行所需的核心类，如Java标准库（rt.jar）等。这个类加载器由C++实现，不是Java类加载器，它位于JVM的内部，无法在Java代码中直接访问。
2. **Extension Class Loader（扩展类加载器）**：它负责加载Java扩展类库（如JAR包中的扩展库）的类。扩展类加载器的父类加载器是启动类加载器。
3. **Application Class Loader（应用类加载器）**：它也被称为系统类加载器，负责加载应用程序classpath下的类。这是大多数Java应用程序默认的类加载器，它的父类加载器是扩展类加载器。

垃圾收集器

新生代

Serial收集器：只能进行一个线程的垃圾回收，在收集时所有工作线程都停止工作，使用标记复制算法，串行

ParNew收集器：是Serial收集器的多线程版本，使用标记复制算法

Parallel Scavenge收集器：多线程收集器，关注控制系统运行的吞吐量

老年代

SerialOld收集器：老年代的Serial，负责标记整理

Parallel Old收集器：老年代的Parallel，也是标记整理

CMS收集器：标记整理-并发收集器，初始标记、并发标记、重新标记、并发清除

G1收集器：初始标记，并发标记，复制标记，复制清除

JVM常用工具

jps：显示本地的Java进程，可以查看本地运行着几个Java程序

jinfo：Java运行时的环境参数

jstack：可以观察到JVM当前所有进程的运行状态和线程当前状态

jmap：可以查看JVM当前的物理内存占用状态

jstat <pid>：查看当前JVM各个分区占用情况

Jstat标识符

- **S0C：** Survivor 0区的容量（字节）。
- **S1C：** Survivor 1区的容量（字节）。
- **S0U：** Survivor 0区的使用量（字节）。
- **S1U：** Survivor 1区的使用量（字节）。
- **EC：** Eden区的容量（字节）。
- **EU：** Eden区的使用量（字节）。
- **OC：** Old区（老年代）的容量（字节）。
- **OU：** Old区（老年代）的使用量（字节）。
- **MC：** 元数据区的容量（字节）。
- **MU：** 元数据区的使用量（字节）。
- **CCSC：** 压缩类空间的容量（字节）。
- **CCSU：** 压缩类空间的使用量（字节）。
- **YGC：** 年轻代垃圾回收的次数。
- **YGCT：** 年轻代垃圾回收所花费的总时间（秒）。
- **FGC：** 老年代垃圾回收的次数。
- **FGCT：** 老年代垃圾回收所花费的总时间（秒）。
- **CGC：** 压缩类空间垃圾回收的次数。
- **CGCT：** 压缩类空间垃圾回收所花费的总时间（秒）。
- **GCT：** 所有垃圾回收所花费的总时间（秒）。

JVM内存结构

哪些部分会出现内存溢出

不会出现内存溢出的区域 - 程序计数器

出现 OutOfMemoryError 的情况

① 堆内存耗尽 - 对象越来越多，又一直在使用，不能被垃圾回收

② 方法区内存耗尽 - 加载的类越来越多，很多框架都会在运行期间动态产生新的类

③ 虚拟机栈累积 - 每个线程最多会占用 1 M 内存，线程个数越来越多，而又长时间运行不销毁时

出现 StackOverflowError 的区域

① 虚拟机栈内部 - 方法调用次数过多

方法区、永久代、元空间之间的关系

① 方法区是 JVM 规范中定义的一块内存区域，用来存储类元数据、方法字节码、即时编译器需要的信息等

② 永久代是 Hotspot 虚拟机对 JVM 规范的实现（1.8 之前）

③ 元空间是 Hotspot 虚拟机对 JVM 规范的实现（1.8 以后），使用本地内存作为这些信息的存储空间

JVM内存参数

- -Xmx：表示虚拟机最大容量
- -Xms：表示虚拟机最小容量
- -Xmn：表示虚拟机新生代容量
- -XX:SurvivorRatio：eden源与from区之比（新生代分为eden源、from区和to区）

判断对象是否可被回收

- 引用计数算法：添加计数器，引用计数为0的可以被回收
- 可达性分析算法：没有被GC Roots指着的对象就会被回收

垃圾回收算法

- 标记清除法：标记有指向的，清除没有指向的
- 标记整理法：整理有指向的，比较适合老年代
- 标记复制法：向另一块区域直接复制，比较适合新生代

GC和分代回收算法

- GC目的：在于实现无用对象内存自动释放，减少内存碎片
- GC要点：
  ① 回收区域是**堆内存**，不包括虚拟机栈，在方法调用结束会自动释放方法占用内存
  ② 判断无用对象，使用**可达性分析算法**，**三色标记法**标记存活对象，回收未标记对象
  ③ GC 具体的实现称为**垃圾回收器**
  ④ GC 大都采用了**分代回收思想**，理论依据是大部分对象朝生夕灭，用完立刻就可以回收，另有少部分对象会长时间存活，每次很难回收，根据这两类对象的特性将回收区域分为**新生代**和**老年代**，不同区域应用不同的回收策略
  ⑤ 根据 GC 的规模可以分成 **Minor GC**，**Mixed GC**，**Full GC**

- 分代回收：
  伊甸园 eden，最初对象都分配到这里，与幸存区合称新生代

  幸存区 survivor，当伊甸园内存不足，回收后的幸存对象到这里，分成 from 和 to，采用标记复制算法

  老年代 old，当幸存区对象熬过几次回收（最多15次），晋升到老年代（幸存区内存不足或大对象会导致提前晋升）

- GC规模：
  Minor GC 发生在新生代的垃圾回收，暂停时间短

  Mixed GC 新生代 + 老年代部分区域的垃圾回收，G1 收集器特有

  Full GC 新生代 + 老年代完整垃圾回收，暂停时间长，应尽力避免

三色标记法：黑色已标记（其与下属均被标记）、灰色标记中（下属还未标记）、白色还未标记

并发漏标问题：增量更新（记录被赋值元素）、原始快照（都被记录）

Full GC

1. **老年代空间不足：** 当老年代（通常存储较长时间存活的对象）中的空间不足以容纳新的对象时，会触发Full GC。这可能是因为老年代中的对象无法被回收，导致堆内存不足。
2. **永久代/元空间空间不足：** 在一些早期的JVM版本中，Full GC也可能会涉及到永久代（或元空间），用于回收类加载信息、常量池等。当永久代（或元空间）空间不足时，也会触发Full GC。
3. **显式调用：** 开发者可以通过Java代码显式调用`System.gc()`方法来请求垃圾回收。虽然这并不能立即触发Full GC，但在某些情况下可能会间接导致Full GC的执行。

类加载

加载

通过类名获取二进制字节流

将字节流存储的静态结构（类）转化为运行时的存储结构（对象），即将class文件加载到JVM当中

内存中生成一个Class对象，作为方法区这个类的各种数据的访问入口

验证

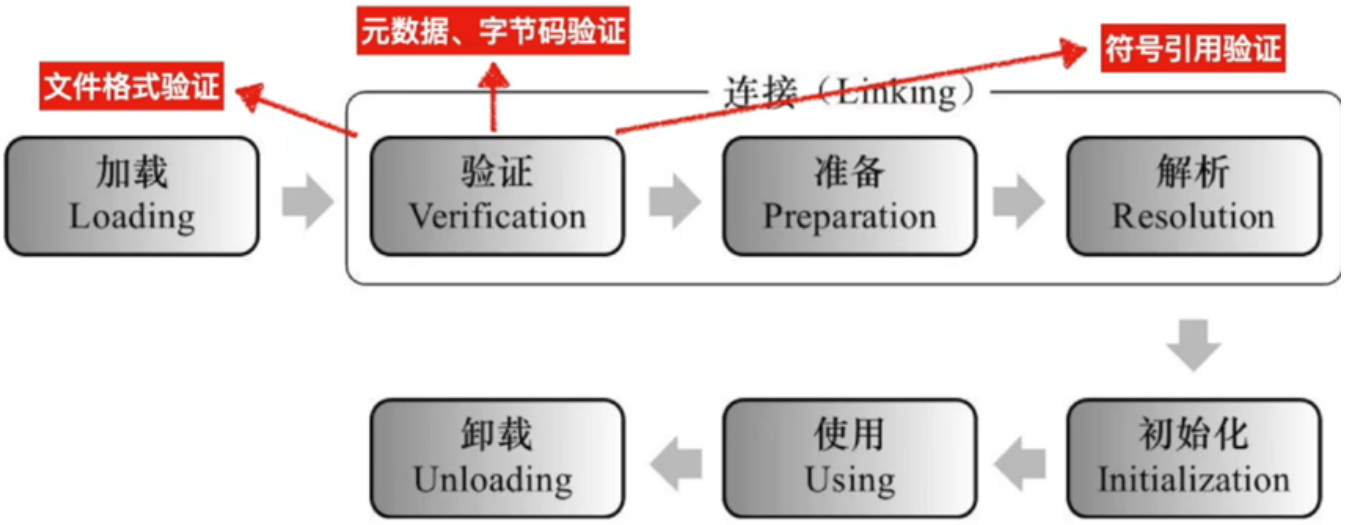
验证字节码、元数据、文件格式、符号引用是否符合规范

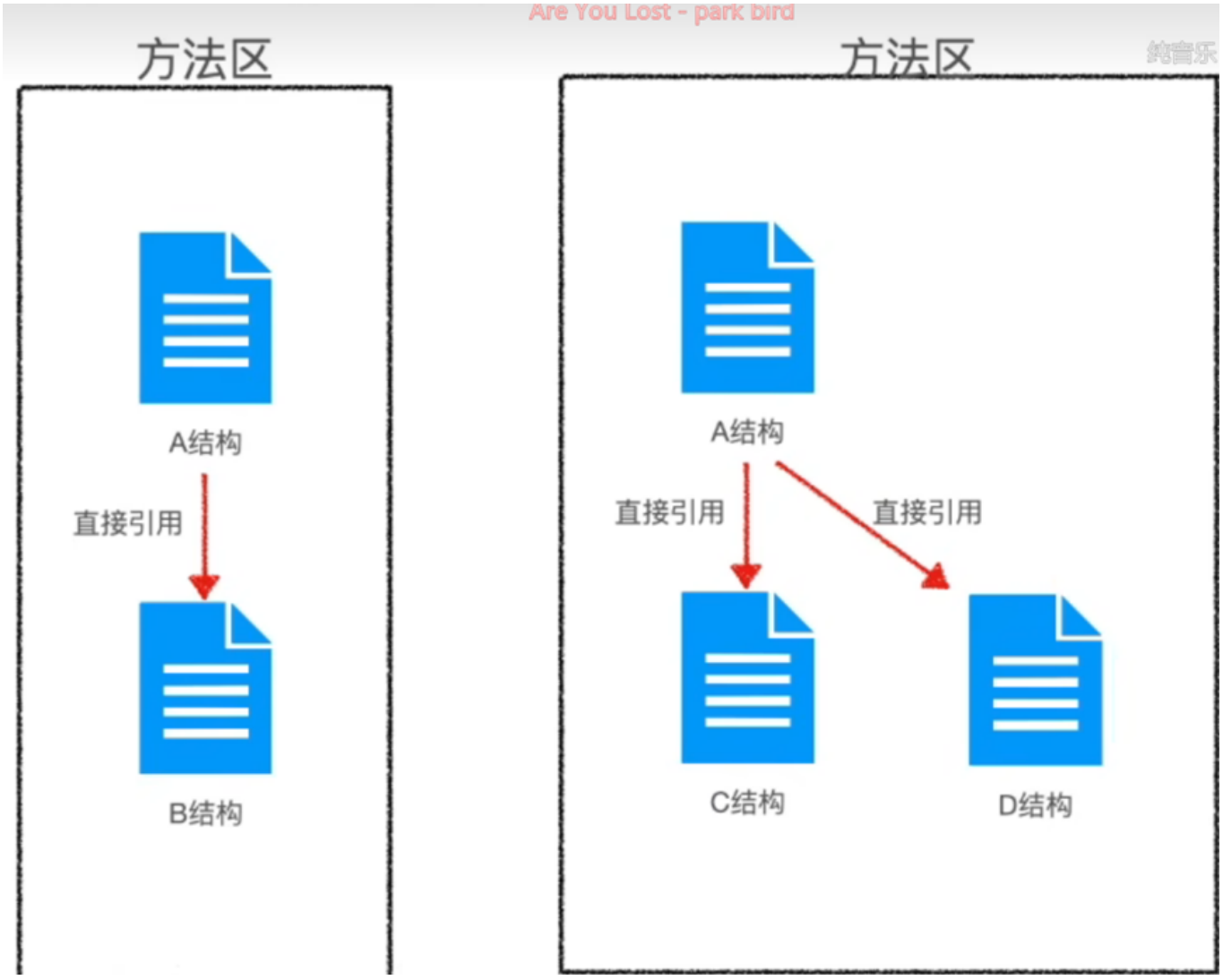
准备

为静态变量分配内存，设置初始值

解析

将符号引用转化为直接引用，符号引用是用来描述目标的符号，直接引用是指向目标的指针、相对偏移量





可达性分析算法：要被Gcroot引用的不可回收，其他可回收

虚拟机栈中的引用的对象
方法区中的类静态属性引用的对象
方法区中的常量引用的对象
本地方法栈中JNI（Native方法）的引用的对象

分代：年轻代、老年代、方法区

G1 GC：多线程、高并发、低暂停

垃圾回收算法

标记清除法：标记后直接删除

标记整理法：标记后将所有可用的放一起，剩下的直接删除

Minor gc和major gc的区别

Minor GC：清理年轻代空间（包括 Eden 和 Survivor 区域），释放在Eden中所有不活跃的对象，释放后若Eden空间仍然不足以放入新对象，则试图将部分Eden中活跃对象放入Survivor区。Survivor区被用来作为Eden及老年代的中间交换区域，当老年代空间足够时，Survivor区的对象会被移到老年代，否则会被保留在Survivor区。

Major GC：清理老年代空间，当老年代空间不够时，JVM会在老年代进行major gc。

Full GC：清理整个堆空间，包括年轻代和老年代空间。

双亲委派机制：把请求交给父类处理

双亲委派机制是 Java 类加载器的一种工作机制，它是一种层次化的加载器结构，由多个类加载器组成。当一个类加载器需要加载一个类时，它会先将这个任务委派给它的父类加载器去完成，如果父类加载器还存在父类加载器，那么这个任务会继续向上委派，直到最顶层的启动类加载器。只有当父类加载器无法完成这个任务时，子类加载器才会尝试自己去加载这个类。

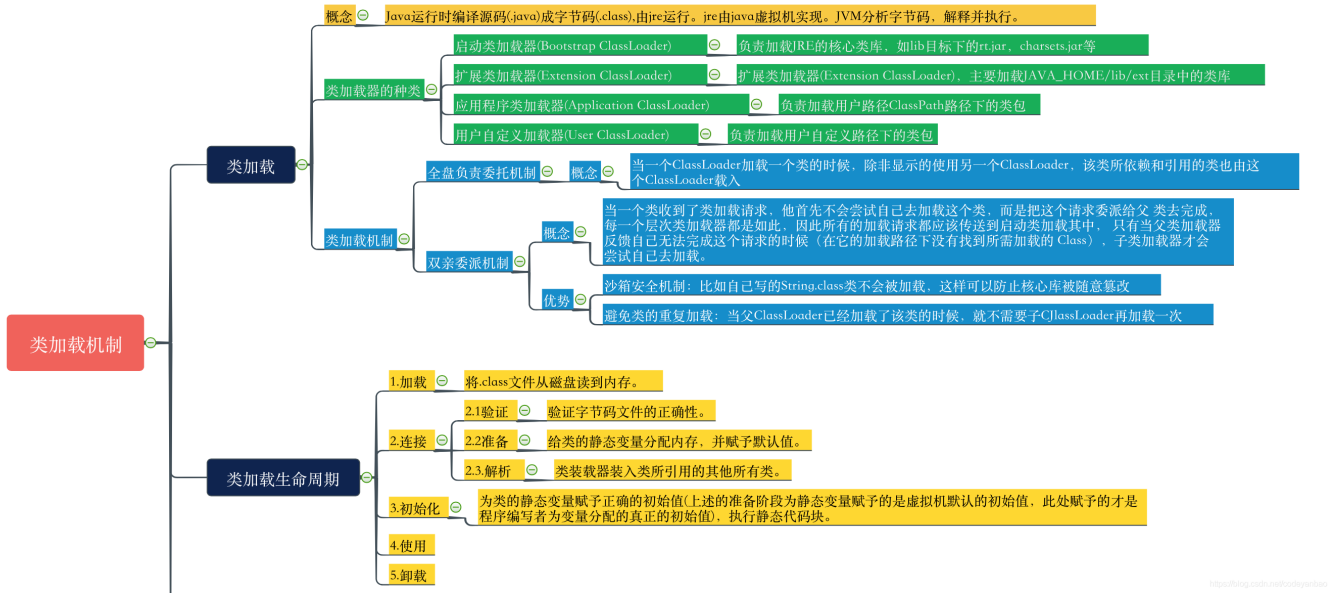
- 如果一个类加载器收到了类加载请求，它并不会自己先加载，而是把这个请求委托给父类的加载器去执行
- 如果父类加载器还存在其父类加载器，则进一步向上委托，依次递归，请求最终将到达顶层的引导类加载器；
- 如果父类加载器可以完成类加载任务，就成功返回，倘若父类加载器无法完成加载任务，子加载器才会尝试自己去加载，这就是双亲委派机制
- 父类加载器一层一层往下分配任务，如果子类加载器能加载，则加载此类，如果将加载任务分配至系统类加载器也无法加载此类，则抛出异常

JVM提供了三层ClassLoader（类加载器）：

- Bootstrap classLoader:主要负责加载核心的类库(java.lang.*等)，构造ExtClassLoader和APPClassLoader。
- ExtClassLoader：主要负责加载jre/lib/ext目录下的一些扩展的jar。
- AppClassLoader：主要负责加载应用程序的主函数类

如果未加载过且不能加载就回到父类加载器

该机制的好处



Tomcat类加载机制

Tomcat 本身也是一个 java 项目，因此其也需要被 JDK 的类加载机制加载，也就必然存在 引导类加载器、扩展类加载器和应用(系统)类加载器。 Common ClassLoader 作为 Catalina ClassLoader 和 Shared ClassLoader 的 parent，而 Shared ClassLoader 又可能存在多个 children 类加载器 WebApp ClassLoader，一个 WebApp ClassLoader 实际上就对应一个 Web 应用，那 Web 应用就有可能存在 Jsp 页 面，这些 Jsp 页面最终会转成 class 类被加载，因此也需要一个 Jsp 的类加载器。 需要注意的是，在代码层面 Catalina ClassLoader、Shared ClassLoader、Common ClassLoader 对应的实体类实际上都是 URLClassLoader 或者 SecureClassLoader，一般 我们只是根据加载内容的不同和加载父子顺序的关系，在逻辑上划分为这三个类加载器； 而 WebApp ClassLoader 和 JasperLoader 都是存在对应的类加载器类的。

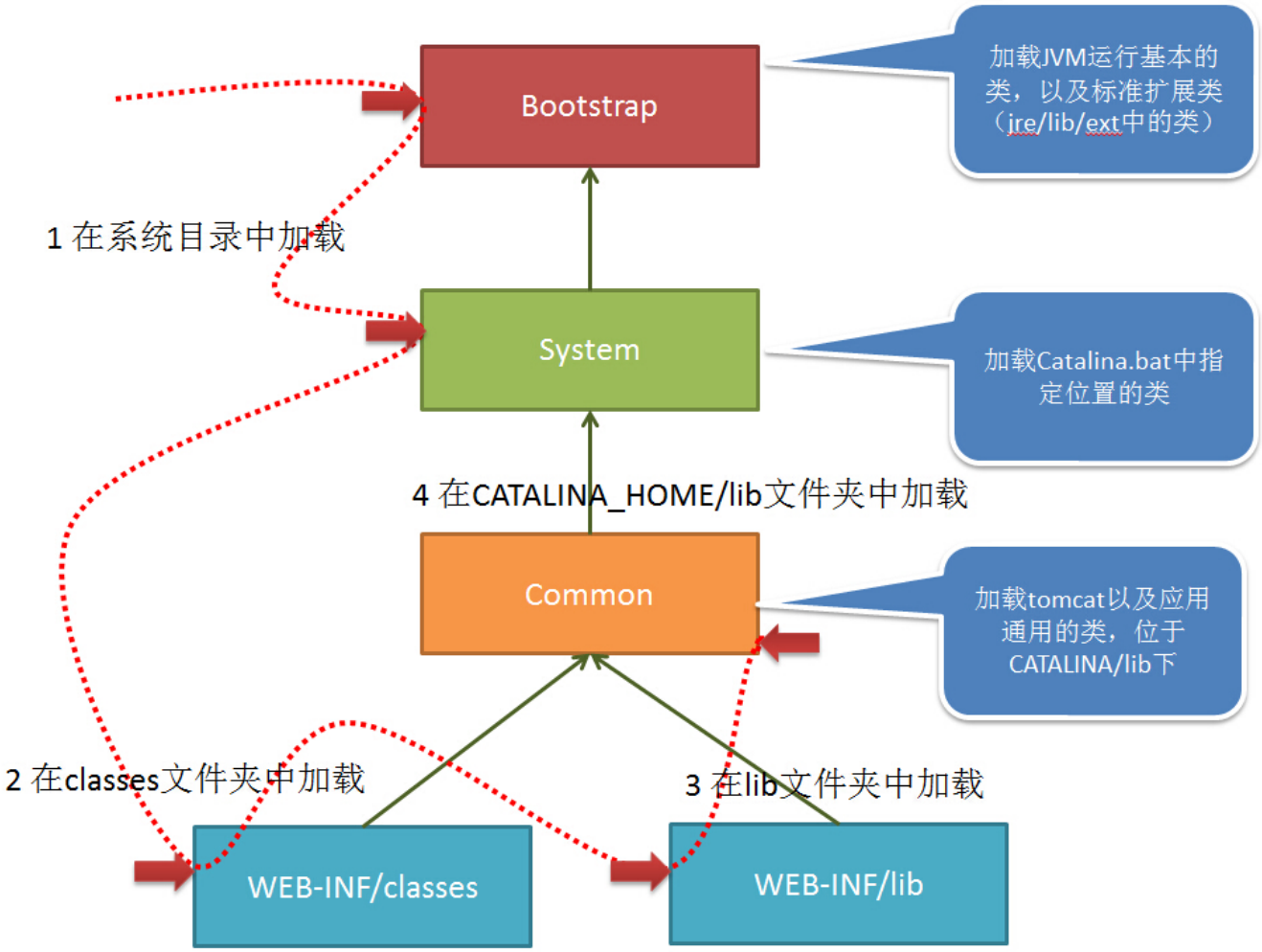
当 tomcat 启动时，会创建几种类加载器：

1 Bootstrap 引导类加载器 加载 JVM 启动所需的类，以及标准扩展类（位于 jre/lib/ext 下）

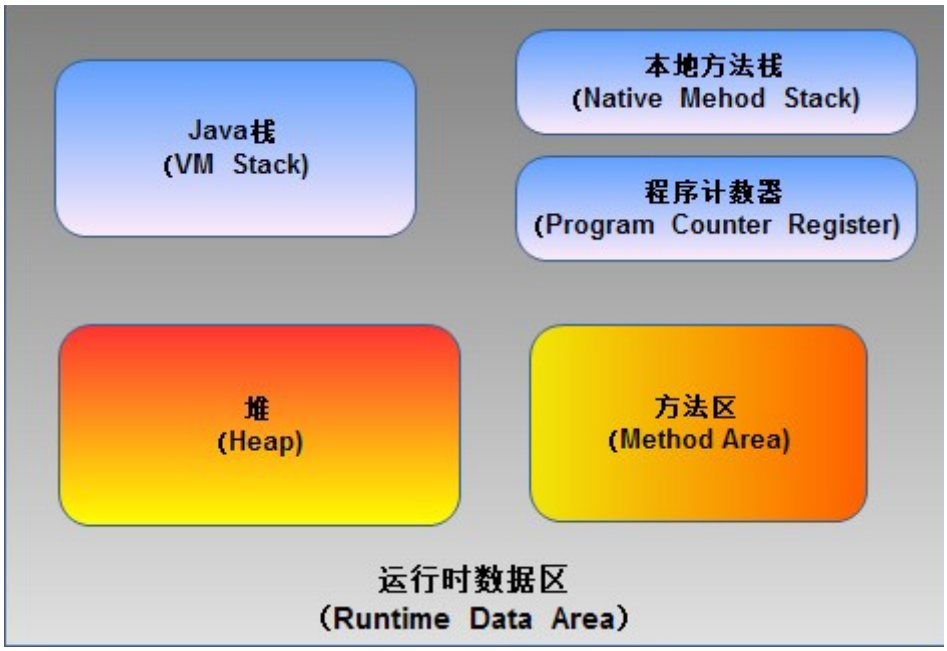
2 System 系统类加载器 加载 tomcat 启动的类，比如 bootstrap.jar，通常在 catalina.bat 或者 catalina.sh 中指定。位于 CATALINA_HOME/bin 下。

3 Common 通用类加载器 加载 tomcat 使用以及应用通用的一些类，位于 CATALINA_HOME/lib 下，比如 servlet-api.jar

4 webapp 应用类加载器每个应用在部署后，都会创建一个唯一的类加载器。该类加载器 会加载位于 WEB-INF/lib 下的 jar 文件中的 class 和 WEB-INF/classes 下的 class 文件。



JVM内存模型



线程共享：

- 方法区：虚拟机加载的类信息、常量、静态变量、JIT（即时编译器）编译后的代码

- 堆区：存储着所有的实例对象（在堆区发生垃圾回收，GC堆）
    - 新生代（Eden、To Survoir、From Survoir）
    - 老年代
    - MetaSpace

线程私有：

- 本地方法栈：本地方法栈和虚拟机栈类似，只不过本地方法栈为虚拟机使用本地方法（native）服务。
    - 本地方法：特指用C、C++等其他语言编写的基于硬件和操作系统的程序
- 虚拟机栈：是栈帧，用于存储局部变量表、操作数栈、动态链接、方法出口等信息，具有局部变量表，操作数栈，常量池引用
- 程序计数器；当前进程以及进程队列

java的引用以及它所指向的数据存储在哪

引用：存储在堆区

数据：方法区

CMS和G1垃圾回收器

CMS（Concurrent Mark-Sweep）垃圾回收器： CMS 是 Java 虚拟机早期引入的一种垃圾回收器，其主要目标是减少垃圾回收造成的停顿时间。传统的垃圾回收器在进行垃圾收集时，会暂停所有应用线程，这就导致在垃圾回收的过程中，应用程序无法响应请求。为了解决这个问题，CMS 引入了并发标记和并发清除的策略。

CMS 垃圾回收器的工作过程分为以下几个阶段：

1. 初始标记（Initial Mark）： 标记直接与根对象直接关联的对象，这个阶段需要暂停应用程序。
2. 并发标记（Concurrent Mark）： 并发地标记所有可达对象。
3. 重新标记（Remark）： 修正并发标记阶段中发生变化的对象，这个阶段需要暂停应用程序。
4. 并发清除（Concurrent Sweep）： 并发地清除未标记的对象。
5. 并发重置（Concurrent Reset）： 重置垃圾回收器的内部状态，准备下一次垃圾回收。

- CMS 回收器适用于对停顿时间要求较高的应用，但由于它在并发标记和并发清除阶段与应用程序并发执行，可能会造成一定的CPU消耗，并且在老年代空间碎片较多时，可能会导致进行碎片整理的Full GC。

G1（Garbage-First）垃圾回收器： G1 是 Java 7 引入的一种垃圾回收器，其目标是解决CMS在碎片整理和一些性能问题上的不足。G1 垃圾回收器采用了分代回收的思想，但不同于其他传统垃圾回收器，它将整个堆内存划分为多个区域，每个区域可能属于年轻代或老年代。

G1 回收器的工作过程如下：

1. 初始标记（Initial Mark）： 标记直接与根对象直接关联的对象，这个阶段需要暂停应用程序。
2. 并发标记（Concurrent Mark）： 并发地标记所有可达对象。
3. 最终标记（Final Remark）： 修正并发标记阶段中发生变化的对象，这个阶段需要暂停应用程序。
4. 复制标记：将可达的对象复制到Survoir里面

5. 筛选回收（G1 Garbage Collection）：根据垃圾回收的需求，在多个区域中选择优先级最高的区域进行垃圾回收，以回收最多的垃圾对象。

- G1 回收器的特点是能够在更短的时间内达到可控制的停顿目标，并且能够避免大部分的内存碎片问题，适用于大内存堆和对停顿时间有严格要求的应用。

永久代和原空间（MetaSpace）

1. **永久代（Permanent Generation）：** 永久代是在 Java 8 及之前的版本中存在的一块内存区域，用于存储类的元数据、静态变量、常量池等信息。它有一个固定的大小，并且在 JVM 启动时就被分配好。由于永久代的大小是有限的，并且不容易进行动态调整，因此在某些情况下可能会导致永久代溢出的问题。
2. **元空间（Metaspace）：** 元空间是在 Java 8 及以后的版本中取代了永久代的一块内存区域。元空间不再受到固定大小的限制，它可以根据需要动态地分配和释放内存。元空间使用的是本地内存（Native Memory），而不是 JVM 进程的堆内存。这使得元空间更加稳定，不容易出现内存溢出的问题。

OOM（OutOfMemory）

1. **堆内存不足：** 堆内存是 Java 程序存储对象实例的地方。如果程序中创建了过多的对象，堆内存可能会耗尽，导致堆内存溢出。常见的情况包括内存泄漏、缓存使用不当等。
2. **方法区/元空间不足：** 方法区（在 Java 8 之前称为永久代）或元空间（在 Java 8 及以后版本中取代了永久代）存储类的元数据、静态变量、常量池等信息。如果加载了大量的类或者常量，方法区/元空间可能会耗尽。
3. **栈溢出：** 每个线程在运行时都有一个栈用于存储方法调用、局部变量等。如果方法调用嵌套层次过深，栈可能会耗尽，导致栈溢出。
4. **本地方法栈溢出：** 本地方法栈用于存储本地方法（由本地语言编写的方法）的调用和返回信息。如果本地方法嵌套层次过深，本地方法栈可能会耗尽，导致本地方法栈溢出。
5. **直接内存溢出：** 直接内存是通过 `ByteBuffer` 等类分配的，通常不受堆大小的限制。但是如果程序过多地分配了直接内存而没有释放，也可能导致直接内存溢出。
6. **递归调用过深：** 如果递归调用没有终止条件或者终止条件不合理，可能会导致栈溢出。
7. **大对象：** 创建过大的对象，超过了堆内存的可用空间，可能导致堆内存溢出。
8. **频繁 Full GC：** 如果程序中产生了大量的垃圾，频繁触发 Full GC（全局垃圾回收），会导致程序的正常运行时间变长，最终可能导致堆内存溢出。
9. **资源泄漏：** 如果程序没有正确地释放资源，如文件句柄、数据库连接等，会导致资源泄漏，耗尽系统资源，从而导致 OOM。

JVM参数调优

1. **堆内存设置：**
    - `-Xmx<size>`：设置最大堆内存大小，例如 `-Xmx1g` 表示最大堆内存为 1GB。
    - `-Xms<size>`：设置初始堆内存大小，避免频繁的堆内存扩展。
2. **新生代与老年代比例：**
    - `-XX:NewRatio=<value>`：设置新生代与老年代的比例，例如 `-XX:NewRatio=2` 表示新生代占整个堆的 1/3。
3. **垃圾回收器选择：**

- `-XX:+UseSerialGC`：使用串行垃圾回收器。
- `-XX:+UseParallelGC`：使用并行垃圾回收器。
- `-XX:+UseConcMarkSweepGC`：使用并发标记-清除垃圾回收器。
- `-XX:+UseG1GC`：使用 G1 垃圾回收器。

4. **垃圾回收相关参数：**
   - `-XX:MaxGCPauseMillis=<value>`：设置最大垃圾回收暂停时间。
   - `-XX:ParallelGCThreads=<value>`：设置并行垃圾回收线程数。
5. **元空间设置：**
   - `-XX:MetaspaceSize=<size>`：设置元空间初始大小。
   - `-XX:MaxMetaspaceSize=<size>`：设置元空间最大大小。
6. **栈内存大小：**
   - `-Xss<size>`：设置线程栈大小，例如 `-Xss256k` 表示线程栈大小为 256KB。
7. **直接内存大小：**
   - `-XX:MaxDirectMemorySize=<size>`：设置直接内存大小。
8. **启用偏向锁和轻量级锁：**
   - `-XX:+UseBiasedLocking`：启用偏向锁优化。
   - `-XX:+UseLightweightLocking`：启用轻量级锁优化。
9. **GC 日志记录：**
   - `-XX:+PrintGC`：打印垃圾回收信息。
   - `-XX:+PrintGCDetails`：打印详细的垃圾回收信息。
10. **性能监控和分析：**
    - `-XX:+PrintCompilation`：打印方法编译信息。
    - `-XX:+HeapDumpOnOutOfMemoryError`：在发生 OOM 时生成堆转储快照。

JIT编译器

JIT（Just-In-Time）编译器就像一个智能的翻译机，它能够在程序运行的时候，把我们用高级语言写的指令翻译成计算机能理解的指令，这样计算机可以更快地执行我们的程序。想象一下，如果我们每次要与计算机交流都需要用一种陌生的语言，会很慢，而 JIT 编译器就像是在我们说话的时候，帮助我们把话翻译成计算机听得懂的语言，这样计算机就能更快地明白我们的意思，加快工作效率。

JIT编译器和Javac编译器

传统的Java编译器（javac）： 这是将Java源代码编译成字节码的编译器。Java源代码经过编译后，会生成字节码文件（以`.class`为扩展名）。这些字节码并不是直接可执行的机器码，而是被设计为在Java虚拟机（JVM）上运行的中间表示。Java虚拟机会将字节码解释执行或者通过JIT编译器编译成机器码，然后执行。

JIT编译器： JIT编译器是Java虚拟机的一部分。当Java应用程序在运行时被执行时，JIT编译器会将字节码动态地编译成机器码，从而实现即时编译（Just-In-Time Compilation）。这些编译后的机器码由处理器直接执行，从而提高了Java应用程序的执行速度。JIT编译器在Java虚拟机内部工作，将热点代码（即频繁执行的代码块）编译成机器码，从而避免了解释执行的性能开销。