Java程序是如何运行起来的

- 编写java代码
- 编译java代码，编译成class字节码文件
- JVM类加载器加载class文件
- JVM字节码执行引擎执行class文件

类加载器

> JVM通过类加载器加载class字节码文件

类加载过程

加载---》验证---》准备---》解析---》初始化---》使用---》卸载

- 加载：

  按需加载，当代码中用到这个类的时候
- 验证阶段：根据JVM规范，验证加载进来的class文件是否符合规范，内容是否完整
- **【核心】准备阶段**：给类分配内存空间，给static变量分配内存空间，设置初始值
- 解析阶段：把符号引用替换为直接引用的过程
- **【核心】初始化**：执行类初始化代码，给变量赋值

  ```
  // 准备阶段给id分配内存空间，设置初始值0
  // 初始化阶段，读取user.id配置，并赋值给id
  public static int id = Configuration.getInt("user.id");
  ```

  什么时候初始化一个类?

  （1）通过`new Object()`来实例化对象时，触发类的加载到初始化的全过程，把这个类准备好，再实例化。在初始化一个类时，如果这个类的父类还没有初始化，必须先去初始化它的父类。

有哪些类加载器?

- 启动类加载器 BootStrap ClassLoader

  负责加载java目录下的核心类，`lib` 目录
- 扩展类加载器 Extension ClassLoader

  加载java目录下 `lib\ext` 的类
- 应用程序类加载器 Application ClassLoader

  负责加载 `classpath` 环境变量所指定的路径中的类，可以理解为去加载自己写好的java代码
- 自定义类加载器

  自定义类加载器，根据自己的需求去加载类

双亲委派机制

类加载是有亲子层级结构的，启动类加载器是最上层的，扩展类加载器在第二层，第三层是应用程序类加载器，最后一层是自定义类加载器

当一个类加载器需要加载一个类时，它首先会委托它的父类加载器去加载，最终传到顶层类加载器。如果父类加载器在自己的加载范围内没有找到这个类，就逐级向下传导

> 先找父加载器去加载，不行的话再由儿子来加载

作用：避免重复加载某些类

Tomcat类加载器

自定义了很多类加载器，用来加载Tomcat自己的核心基础类库

- Common类加载器
- Catalina类加载器
- Shared类加载器

为每个部署在里面的应用都有一个对应的WebApp类加载器，负责加载我们部署的Web应用类

至于JSP类加载器，则是给每个JSP都准备了一个JSP类加载器

> Tomcat 打破了双亲委派机制

每个WebApp负责加载自己对应的那个Web应用的class文件，不会传导给上层类加载器

内存区域

- 程序计数器：记录当前执行的字节码指令位置，线程私有
- 本地方法栈：执行native方法所需的栈空间，线程私有
- 虚拟机栈：执行一个方法，就会对这个方法创建对应的**栈帧**，栈帧里存放这个方法的**局部变量**、操作数栈、动态链接、方法出口。方法调用的过程，就是一个栈帧入栈和出栈的过程。线程私有
- 堆：存放创建的对象
- 元数据空间Metaspace：存放class类信息，常量池，线程共享
- **堆外内存**：通过NIO的allocateDirect这种API，可以在堆外分配内存空间，通过Java虚拟机里的DirectByteBuffer来引用和操作堆外内存空间

垃圾回收

什么是垃圾?

> 内存中创建出来的对象，不再使用

为什么需要回收?

> 由于内存资源有限，需要对一些没用的对象进行回收，否则就会导致内存溢出，程序无法创建新对象

哪些区域需要回收?

> 堆区，虚拟机栈，本地方法栈，元数据区

怎么回收?

> JVM后台有垃圾回收线程，检查对象是否有人引用，标记垃圾对象

元数据区的类能被回收吗？能，满足以下三个条件

> 类回收条件：
>
> - 类的所有实例对象被回收
> - 类的ClassLoader被回收
> - 类的class对象没有被引用

什么情况下JVM里的一个对象会被回收

- 新生代满了会触发Young GC，老年代满了会触发Old GC，永久代满了会触发 Full GC
- 根据可达性分析算法，逐层向上分析是否有谁在引用它，判断一个对象是否有被GC Roots节点引用，如果没有，这个对象就是可回收的
- 类的静态变量，局部变量可以作为GC Roots节点
- 引用分为强引用，软引用，弱引用，虚引用；
  - 强引用的对象不会被回收
  - 软引用的对象一般不会回收，如果GC过后内存空间仍然不够，则会对其进行回收
  - 弱引用：每次执行GC回收的时候会进行回收
  - 虚引用：很少用，可忽略
- 如果一个对象没有被GC Roots引用，到了回收阶段是否一定立马被回收，不一定，可以通过finialize()方法自救
- 类重写执行finialize()方法，在finialize()中让一个GC Roots重新引用这个对象，可以自救

垃圾回收算法

复制算法

把新生代内存空间划分为3块区域，1个Eden区，2个Survivor区，其中Eden区占80%，每一块Survivor区占10%，每次使用其中的一块Survivor区和Eden区

对象刚开始分配在Eden区，如果Eden区满了，就会触发垃圾回收，此时就会把Eden区存活的对象都一次性转移到一块空的Survivor区，接着Eden区被清空

然后再次分配对象到Eden区，如果Eden区又满了，再次触发垃圾回收，此时就会把Eden区存活的对象和Survivor区存活的对象，转移到另一块Survivor区。

标记清理算法

标记存活对象，清除垃圾对象

不足：会造成内存碎片发生

标记清理整理算法

标记存活对象，并将存活的对象挪动到一边，让存活的对象紧凑的挨在一起，避免垃圾回收过后出现过多的内存碎片；然后一次性把垃圾对象回收掉

新生代对象什么时候进入老年代

- 对象经过15次GC后，仍然存活；则会跑到老年代。通过参数`-XX:MaxTenuringThreshold`配置，最大15，由对象头的分代年龄（占4位）决定
- 动态年龄判断，当前放对象的survivor区域里，一批对象的总大小大于了这块Survivor区域的内存大小的50%，那么此时大于等于这批对象年龄的对象，就会直接进入老年代

  例如：1岁+2岁+n岁的对象占用的内存空间超过Survivor区域的50%，那么大于等于n岁的对象进入老年代
- 大对象直接进入老年代，通过参数`-XX:PretenureSizeThreshold`指定多大的对象是大对象，默认是1MB
- Minor GC过后，存活的对象太多，无法放入另一个块Survivor区域，那么这批对象会进入老年代

老年代空间分配担保原则

为了防止Minor GC过后，存活的对象太多，这批对象直接挪到老年代，而老年代空间不够。

在每次只需Minor GC之前，会检查一下老年代的可用内存空间，是否大于新生代所有对象的总和

- 如果老年代可用空间**小于**新生代所有对象，执行FGC
- 如果老年代可用空间**大于**新生代所有对象，大胆Minor GC
- 如果老年代可用空间**大于**历次Minor GC后进入老年代，尝试执行Minor GC
  - Minor GC过后，存活的对象小于Survivor区的大小，存活对象进入Survivor区
  - Minor GC过后，存活的对象大于Survivor区，小于老年代可用内存，存活对象进入老年代
  - Minor GC过后，存活的对象大于Survivor区，大于老年代可用内存，老年代执行FGC，执行FGC过后，剩余空间仍不够，则会导致**OOM内存溢出**

垃圾回收器

Serial和Serial Old

分别用来回收新生代和老年代的垃圾对象，**单线程**，基本不用

ParNew 和 CMS

ParNew是新生代的垃圾回收器，CMS是用老年代的垃圾回收器，**多线程**

ParNew

> 通过参数 -XX:+UseParNewGC 配置新生代使用 ParNew 垃圾回收器
>
> 通过参数 -XX:ParallelGCThreads 配置 ParNew 垃圾回收线程数量（一般不去修改）

（1）停止工作线程，禁止程序继续创建对象

（2）使用多个垃圾回收线程执行垃圾回收

CMS

> 通过参数 -XX:+UseConcMarkSweepGC 配置老年代使用 CMS 垃圾回收器
>
> 标记-清理压缩算法

- **工作原理**

（1）初始标记：标记所有GC Roots引用的对象，停止工作线程，STW

（2）并发标记：工作线程工作，垃圾回收线程，对老年代已有对象进行GC Roots追踪（最耗时）

（3）重新标记：由于并发标记阶段，一边标记一边产生新对象，所以还会有部分对象是第二阶段没有标记出来的，停止工作线程，重新标记，STW

（4）并发清理：工作线程工作，垃圾回收线程清理垃圾对象

- **浮动垃圾**

  由于在并发清理阶段，系统处于运行状态，可能会有对象进入老年代，同时变成垃圾对象，这种垃圾对象就称为"浮动垃圾"，需要等到下一次GC才能被回收

  所以为了给这部分浮动垃圾预留一些空间，当老年代内存占用达到一定比例了，就会执行GC，通过参数

`-XX:CMSInitiatingOccupancyFraction`配置，默认为92%，表示老年代占用比例达到92%触发CMS垃圾回收

- **Concurrent Mode Failure问题**

如果CMS垃圾回收期间，系统要放入老年代的对象大于可用内存空间，这个时候就会触发`Concurrent Mode Failure`，表示并发垃圾回收失败。此时就会自动用Serial Old垃圾回收器替代CMS，STW，重新执行长时间的GC Roots标记，标记出全部垃圾对象，然后一次性回收。

生产环境，CMS垃圾回收的比例需要合理优化，避免`Concurrent Mode Failure`的问题发生

- **内存整理**

  CMS通过配置可以设置在GC过后整理内存，把存活的对象都挪到一边，腾出大块的连续空间，避免内存碎片的发生

  通过参数`-XX:+UseCMSCompactAtFullCollection`表示FGC过后，停止工作线程，进行碎片整理，把存活的对象挪到一边，避免内存碎片

  通过参数`-XX:CMSFullGCsBeforeCompaction`表示多少次FGC过后再执行内存碎片整理，默认0，表示每次FGC都会进行内存整理

G1

> -XX:+UseG1GC 指定G1垃圾回收器
>
> -XX:G1HeapRegionSize 指定Region数量，最多只能分配2048

G1核心概念：G1把内存区域划分成多个大小相等的Region，我们可以设置一个垃圾回收的预停顿的时间，来控制垃圾回收对系统性能的影响。通过追踪每个Region的回收价值（回收对象的大小和回收对象的时间），选择回收时间短回收垃圾多的Region进行回收，在有限的时间内尽量回收尽可能多的垃圾对象

G1 的Region也有新生代和老年代的逻辑区分

新生代

通过 `-XX:G1NewSizePercent` 来设置新生代初始占比，默认占堆内存5%，一般保持默认值

通过 `-XX:G1MaxNewSizePercent` 来设置最多新生代的占比，默认60%。

即 系统运行期间，JVM会给新生代增加更多的Region，最多不会超过60%，一旦Region进行了回收，新生代的Region数量就会减少，这些都是动态的

新生代每个Region仍是按照 Eden 和 Survivor 进行划分的

一旦新生代达到了设定的占据堆内存最大大小的60%，而且Eden区都满了，就会触发新生代GC，采用**复制算法进行回收**，进入 STW 状态

G1也会动态跟踪Region，根据最大停顿时间，标记出一部分Region满足回收条件后，就会触发GC

通过 `-XX:MaxGCPauseMills` 来设定GC停顿时间，默认200ms，G1就会追踪每个Region需要回收多少时间，回收多少对象，保证GC停顿时间控制在指定范围内，尽可能回收掉多一些对象

老年代

按照新生代最多分配60%的Region，老年代则可以占据40%的Region，对象什么时候进入老年代呢?

- 对象在新生代躲过多次垃圾回收，达到了一定年龄，就会进入老年代
- 触发动态年龄规则，某次新生代GC过后，存活对象超过了Survivor的50%，就会将大于等于这批对象最大年龄的对象进入老年代

当老年代占据了堆内存45%的Region的时候，会触发一个新生代+老年代一起回收的**混合回收**阶段

大对象Region

当一个对象超过了Region大小的50%，那么这个对象就会放入到专门的Region，如果一个Region放不下，就会横跨多个Region存储

大对象Region怎么回收，每次新生代和老年代进行回收的时候，会顺带着将大对象Region一起回收

G1混合回收过程

通过参数 `-XX:InitiatingHeapOccupancyPercent` 指定混合回收的触发条件，默认45%

表示当老年代的Region占据了堆内存的Region的45%，会触发混合回收。回收新生代+老年代+大对象Region

- **初始标记**：仅标记GC Roots能直接引用的对象，STW
- **并发标记**：工作线程运行，GC线程进行GC Roots标记，从GC Roots开始追踪所有存活的对象，会记录哪些对象新建了，哪些对象失去了引用
- **最终标记**：根据并发标记阶段记录的那些对象修改，最终标记一下有哪些存活对象，有哪些垃圾对象。
- **混合回收**：计算老年代中每个Region中的存活对象数量，存活对象占比，执行垃圾回收的预期性能和效率，全力回收垃圾，STW，会选择部分Region进行回收，保证回收时间在我们预设的停顿时间范围之内。

  最后一个混合回收阶段会执行多次，通过参数`-XX:G1MixedGCCountTarget` 配置混合回收次数，默认为8。

表示最后一个阶段先停止系统，混合回收部分Region，再恢复系统，再停止系统，混合回收部分Region，反复8次。

进行多次回收就是为了让系统停顿时间不要过长

`-XX:G1HeadWastePercent`，默认5%，混合回收过程中，新生代会不断空出Region出来，一旦空闲出来的Region数量达到了对内存的5%，则立即停止混合回收

`-XX:G1MixedGCLiveThresholdPercent`，默认85%，表示一个Region中存活的对象必须低于85%，这个Region才能进行回收。不然存活对象多，移动成本高

如何保证只做YGC，FGC次数为0?

- 不让对象进入老年代，就不会发生FGC
- 对象进入老年代有以下4种方式：

  1、新生代每发生一次GC，存活对象的年龄就+1，当对象年龄=15时，对象就会跑到老年代

  2、当创建的对象太大，超过配置的大对象阈值，就会跑到老年代

  3、动态年龄判断触发，即在survivor区中有一批对象的年龄超过了survivor区内存的50%，触发动态年龄规则，当大于等于这批对象最大年龄的对象就会跑到老年代

  4、新生代发生Minior GC过后，存活的对象无法放入survivor区，这批对象就会跑到老年代
- 只做YGC的话，就要避免上述4种新生代对象跑到老年代的情况发生

  1、设置分代年龄最大为15

  2、避免大对象的产生，优化代码，减少批处理数据量，控制List，Map等大小

  3、触发动态年龄规则，是由于Survivor区域空间不够，可以调大Survivor区域

  4、Minior GC过后，仍放不下Survivor区域，可以调大Survivor区域

  5、合理分配Eden，Survivor，老年代的内存大小

附录一： 参数配置

- -Xms：堆内存初始大小
- -Xmx：堆内存的最大大小
- -Xmn：堆内存中的新生代大小
- -XX:PermSize：永久代大小（1.7）
- -XX:MaxPermSize：永久代最大大小（1.7）
- -XX:MetaspaceSize：永久代大小（1.8）
- -XX:MaxMetaspaceSize：永久代最大大小（1.8）
- -Xss：每个线程的栈内存大小
- -XX:NewRadio：新生代与老年代占比，默认为2，表示 New：Old = 1 : 2
- -XX:SurvivorRadio：新生代Eden区的占比，默认为8，表示Eden:S0:S1 = 8:1:1
- -XX:MaxTenuringThreshold：对象分代年龄，默认15，最大15
- -XX:PretenureSizeThreshold：大对象大小，默认1MB
- -XX:CMSInitiatingOccupancyFraction：触发CMS垃圾回收的内存占比，默认92%

- -XX:+UseCMSCompactAtFullCollection：老年代开启GC过后进行内存整理
- -XX:CMSFullGCsBeforeCompaction：老年代内存整理频次，几次GC后进行整理，默认0，表示每次GC后都进行整理

附录二、参数调优

新生代

- Survivor空间不够：调整新生代与老年代的比例，分配足够的Survivor空间

  配置Eden区占比`-XX:SurvivorRadio=8`

  配置新生代与老年代占比`-XX:NewRadio=2`，表示新生代占1，老年代占2

  可以避免YGC后对象放不下Survivor区，触发动态年龄规则导致对象跑到老年代去
- 15次YGC后，对象仍然存活；调整 -XX:MaxTenuringThreshold 参数，调小点，让存活时间久的对象，提前进入老年代
- 设置大对象大小`-XX:PretenureSizeThreshold=1M`，一般1MB够用了
- 指定垃圾回收器`-XX:+UseParNew`

老年代

- 配置老年代空间占比超过多少执行FGC，`-XX:CMSInitiatingOccupancyFraction=92`，默认92%
- 指定老年代垃圾回收器，`-XX:+UseCMSConcMarkSweepGC`
- 开启老年代GC后整理内存配置，`-XX:+UseCMSCompactAtFullCollection`
- 配置内存整理的频次，老年代执行多少次GC后进行整理，`-XX:CMSFullGCsBeforeCompaction=0`，默认为0，表示每次FGC，都进行内存整理，避免内存碎片

附录三：命令

- **jps**：查看本机java进程信息

```
E:\software\Java\jdk1.8.0_71\bin>jps
17524 Jps
21988 LoginController
22888 Launcher
6200 Main
5676
```

- **jstack**：打印线程的栈信息

```
E:\software\Java\jdk1.8.0_71\bin>jstack 21988
Full thread dump Java HotSpot(TM) 64-Bit Server VM (25.71-b15 mixed mode):

"Inactive RequestProcessor thread [Was:Default
RequestProcessor/com.sun.tools.visualvm.tools.jmx.CachedMBeanServerConnectionFactory$S
napshotInvocationHandler$2]" #87 daemon prio=1 os_prio=-2 tid=0x00000000154a9000
nid=0xda0 in Object.wait() [0x000000001c55e000]
   java.lang.Thread.State: TIMED_WAITING (on object monitor)
        at java.lang.Object.wait(Native Method)

"pool-8-thread-1" #49 prio=5 os_prio=0 tid=0x00000000154aa800 nid=0x5c84 waiting on
condition [0x0000000000f6e000]
   java.lang.Thread.State: WAITING (parking)
```

- **jmap**：打印堆内存信息

```
E:\software\Java\jdk1.8.0_71\bin>jmap -heap 6200
Attaching to process ID 6200, please wait...
Debugger attached successfully.
Server compiler detected.
JVM version is 25.71-b15

using thread-local object allocation.
Parallel GC with 4 thread(s)

Heap Configuration:
   MinHeapFreeRatio         = 0
   MaxHeapFreeRatio         = 100
   MaxHeapSize              = 268435456 (256.0MB)
   NewSize                  = 8388608 (8.0MB)
   MaxNewSize               = 89128960 (85.0MB)
   OldSize                  = 16777216 (16.0MB)
   NewRatio                 = 2
   SurvivorRatio            = 8
   MetaspaceSize            = 21807104 (20.796875MB)
   CompressedClassSpaceSize = 1073741824 (1024.0MB)
   MaxMetaspaceSize         = 17592186044415 MB
   G1HeapRegionSize         = 0 (0.0MB)

Heap Usage:
PS Young Generation
Eden Space:
   capacity = 38273024 (36.5MB)
   used     = 37867960 (36.11370086669922MB)
   free     = 405064 (0.3862991333007812 5MB)
   98.94164621013485% used
From Space:
   capacity = 6815744 (6.5MB)
   used     = 5056136 (4.821907043457031MB)
   free     = 1759608 (1.6780929565429688MB)
   74.18318528395433% used
To Space:
```

```
   capacity = 7340032 (7.0MB)
   used     = 0 (0.0MB)
   free     = 7340032 (7.0MB)
   0.0% used
PS Old Generation
   capacity = 58195968 (55.5MB)
   used     = 40108296 (38.25025177001953MB)
   free     = 18087672 (17.24974822998047MB)
   68.91937255859375% used
```

- **jstat**：查看gc情况
  - 每秒打印各个区域内存**占用大小**（字节），打印10次

```
jstat -gc pid 1000 10
 S0C    S1C    S0U    S1U      EC       EU        OC         OU       MC     MU
CCSC   CCSU   YGC     YGCT    FGC    FGCT     GCT
128.0  128.0   0.0   128.0   1088.0   439.6     2752.0     472.8    2240.0  95.4   0.0
   0.0        1    0.002   0       0.000    0.002
128.0  128.0   0.0   128.0   1088.0   439.6     2752.0     472.8    2240.0  95.4   0.0
   0.0        1    0.002   0       0.000    0.002
```

  - 每秒打印各个区域内存**占用百分比**，打印10次

```
jstat -gcutil pid 1000 10
S0     S1     E      O      M     CCS    YGC     YGCT    FGC    FGCT     GCT
3.33   0.00  53.24  42.28  93.96  87.50     40    0.327     4    0.392    0.719
3.33   0.00  53.24  42.28  93.96  87.50     40    0.327     4    0.392    0.719
```