JVM

---???---
1. 性能调优
2. 平台稳定性
3. 技术选型
4. 核心功能架构

---栈式架构---
1. 跨平台
2. 指令集小
3. 指令多
4. 执行性能比寄存器差

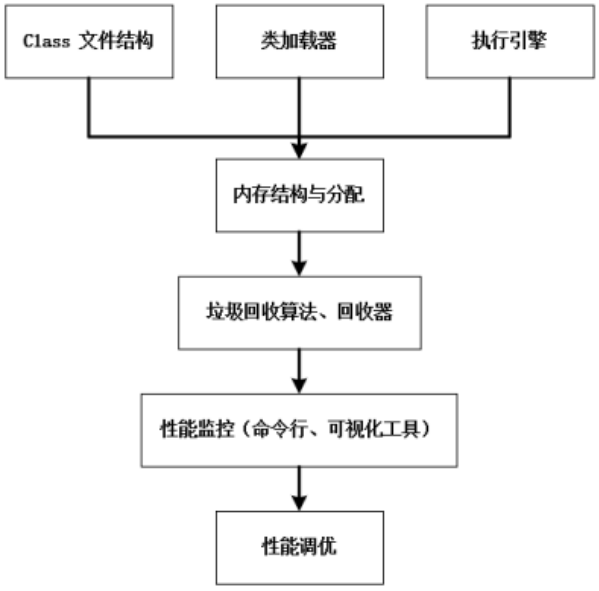
---JVM 特点---
1. 一次编译，到处运行
2. 自动内存管理
3. 自动垃圾回收

---JVM 生命周期---
1. 启动
通过 bootstrap class loader 创建一个 initial class
2. 执行
执行 Java 程序实际就是执行一个 JVM 进程
3. 退出
程序正常退出
程序抛异常或错误而终止
操作系统出现异常而终止
线程调用 Runtime 或 system 的 exit()，且 Java 安全管理器允许此次 exit 或 halt 操作
JNI 规范用 JNI Invocation API 来加载或卸载 Java 虚拟机

---执行流程---
源码 -> 编译（词法分析 -> 语法分析 -> 抽象语法树 -> 语义分析 -> 注解抽象语法树 -> 字节码生成）-> 字节码 -> 类加载 -> 字节码校验 -> 字节码解析执行 -> (JIT编译执行 | 操作系统执行)

---反编译---
javap -v Test.class

Hotspot -> Graal VM



- **Class loader**

1

类加载器（加载 .class 文件至 JVM，成为元数据模版）

- Loading

  加载

  1. 通过类的全限定名获取二进制字节流
  2. 字节流的静态存储结构转成方法区的运行时数据结构
  3. 生成代表这个类的 java.lang.Class 对象，作为方法区这个类的访问入口

  按需加载，从 main() 所在类开始加载

  加载方式

- 1. 本地系统直接加载
  2. 网络获取， Web Applet
  3. 从 zip 压缩包读取，如 jar，war
  4. 运行时计算生成，如动态代理
  5. 从其他文件生成，如 JSP 提取成 .class 文件
  6. 从加密文件读取，如防 .class 文件被反编译的保护措施

- Linking

  链接

  - Verification

    验证（确保 .class 文件中包含的信息符合虚拟机要求，保证 Class 的正确性，不会危害虚拟机本身）

    1. 文件格式验证
    2. 元数据验证
    3. 字节码验证
    4. 符号引用验证

  - Preparation

    准备（为类变量（静态变量）分配内存，并设置其默认值）

    ---注意---
    final static 常量在编译时就会分配，准备阶段会被显式初始化
    类变量分配在方法区，实例变量随对象一起分配到 Java 堆

  - Resolution

    解析（将常量池的符号引用转换为直接引用，主要针对类、接口、字段、类方法、接口方法、方法类型等，对应常量池中的 CONSTANT Class info、CONSTANT Fieldref info、CONSTANT Methodref info 等）

    符号引用 - 一组用来描述引用目标的符号
    直接引用 - 指向目标的指针、偏移量、间接定位目标的句柄

- Initialization

  初始化（执行类构造器 <clinit>()）
  1. <clinit> 方法由静态变量显示赋值代码和静态代码块组成
  2. 类变量显示赋值代码和静态代码块代码从上到下顺序执行
  3. <clinit> 方法只执行一次

  ---注意---
  1. 类的实例化需要先加载并初始化该类，main 所在的类需要先加载和初始化
  2. 子类的初始化需要先初始化父类
  3. 虚拟机保证一个类的 <clinit>() 在多线程下被同步加锁

- 分类

  ---获取方式---

  指定类的类加载器 - clazz.getClassLoader()
  线程上下文的类加载器 - Thread.currentThread().getContextClassLoader()
  系统类加载器 - ClassLoader.getSystemClassLoader()
  调用者的类加载器 - DriverManager.getCallerClassLoader()

  - Bootstrap Class Loader

    启动类加载器（引导类加载器），虚拟机自带，C/C++ 实现

    1. 用于加载核心类库（jre/lib/rt.jar, resource.jar, sun.boot.class.path/...）的 JVM 自身需要的类（包名以 java, javax, sun 等开头）
    2. 用于加载扩展类加载器和系统类加载器，并指定为他们的父类加载器

  - Extension Class Loader

    扩展类加载器（Java 编写，派生自 ClassLoader，父类加载器是启动类加载器）
    用于加载 java.ext.dirs 系统属性指定目录，或扩展目录（jre/lib/ext）的类库，用户的jar 放入此目录也会走扩展类加载器

  - System Class Loader

    系统类加载器（应用类加载器，Java 编写，继承自 ClassLoader，父类加载器为扩展类加载器，程序中默认的类加载器）
    用于加载 classpath 环境变量或 java.class.path 系统变量路径下的类库

    classLoader#getSystemClassLoader()

  - User Defined Class Loader

用户自定义类加载器

---作用---
1. 隔离加载类
2. 修改类加载的方式
3. 扩展加载源
4. 防止源码泄漏

---自定义方法---
1. 继承 java.lang.ClassLoader
2. 重写 loadClass()， JDK1.2 后建议自定义 findclass()
3. 继承 URIClassLoader，避免自定义 findclass()

- 双亲委派机制

  把加载请求交由父类处理的任务委派模式
  把类加载请求委托给父类加载器去执行，直到引导类加载器，如果父类加载器可以完成类加载，则成功返回，否则子类加载器尝试自己加载

  ---优势---
  1. 避免类的重复加载
  2. 保护程序安全，防止核心 API 被随意篡改，触发沙箱安全机制

- 沙箱安全机制

  自定义类的包名与核心源代码相同时，会率先使用引导类加载器，引导类加载器会优先加载 JDK 自带的类，从而触发安全异常，这种保证 Java 核心源代码安全的机制，就是沙箱安全机制

- 类对象是否相同

  1. 类的全限定名一致，包括包名
  2. 加载这个类的类加载器相同

  加载一个使用用户类加载器的类时，类加载器的一个引用会作为该类信息的一部分保存在方法区

- 类的主动使用

  类的主动使用，会导致类的初始化

  1. 创建类的实例
  2. 访问类或接口的静态变量，或给其赋值
  3. 调用类的静态方法
  4. 反射
  5. 初始化类的子类
  6. JVM 启动时被标明为启动类的类
  7. JDK7 开始提供的动态语言支持
  java.lang.invoke.MethodHandle 实例的解析结果 REF getStatic, REF putStatic, REF invokeStatic 句柄对应的类没有初始化，则初始化

  其他使用 Java 类的范式不会导致类的初始化，被看作类的被动使用

- 实例初始化

  实例初始化就是执行 <init> 方法
  1. <init> 方法可能重载多个，有几个构造器就有几个<init> 方法
  2. <init> 方法由非静态实例变量显示赋值代码、非静态代码块、对应构造器代码组成
  3. 非静态实例变量显示赋值和非静态代码块从上到下顺序执行，而对应构造器代码最后执行
  4. 每次创建实例对象，调用对应构造器，执行的就是对应的 <init> 方法
  5. <init> 方法的首行是 super() 或 super(实参列表)，即对应父类的 <init> 方法

- 方法重写
  - 不可以被重写的方法
    - final
    - 静态方法
    - private 等子类中不可见方法
  - 对象的多态性

    1. 子类如果重写了父类的方法，通过子类对象调用的一定是子类重写过的代码
    2. 非静态方法默认的调用对象是 this
    3. this 对象在构造器或者说 <init> 方法中就是正在创建的对象
  - Override vs. Overload
  - Override 的要求
    - 方法名
    - 形参列表
    - 返回值类型
    - 抛出的异常列表
    - 修饰符
  - invokespecial 指令
- 局部变量 vs. 成员变量
  - 声明位置

局部变量：方法体{}中，形参，代码块{}中
成员变量：类中方法外
类变量：有 static 修饰
实例变量：无 static 修饰

- 修饰符

  局部变量：final
  成员变量：public, protected, private, final, static, volatile, transient

- 存储位置

  局部变量：栈
  实例变量：堆
  类变量：方发区

- 作用域

  局部变量：从声明开始，到所属}结束
  实例变量：在当前类的 this.，在其他类的 对象名.
  类变量：在当前类的 类名.，在其他类的 类名. 或 实例名.

- 声明周期

  局部变量：每个线程，每次调用执行都是新的生命周期
  实例变量：随着对象的创建而初始化，随着对象的回收而消亡，每个对象的实例变量是独立的
  类变量：随着类的初始化而初始化，随着类的卸载而消亡，该类的所有对象的类变量是共享的

- **Runtime Area**

  2

  运行时数据区

  线程独有：程序计数器，栈，本地方法栈
  线程共享：堆，堆外内存（永久代或元空间、代码缓存）

  ---JVM 线程---
  与操作系统本地线程直接映射

  ---JVM 系统线程---
  虚拟机线程 - JVM 达到安全点，堆不会变化时出现，stop-the-world 的垃圾收集，线程栈收集，线程挂起，偏向锁撤销
  周期任务线程 - 时间周期事件（如中断）
  GC 线程 - JVM 里各类垃圾收集
  编译线程 - 运行时将字节码编译成本地代码
  信号调度线程 - 接收信号并发送给 JVM

  ---Error 和 GC---
  程序计数器 N N
  虚拟机栈 Y N
  本地方法栈 Y N
  方法区 Y Y
  堆 Y Y

  - Program Counter Register

    程序计数器，物理 PC 寄存器的抽象模拟
    线程私有，生命周期与线程一致

    ---作用---
    1. 指向线程的当前方法（任何时间一个线程只有一个方法在执行）的 JVM 指令地址（native 方法的值是 undefined）；
    2. 字节码解释器通过改变计数器的值选择下一条需要指向的字节码指令，实现程序控制流的指示器，分支、循环、跳转、异常处理、线程恢复等；
    3. 保证 CPU 时间片切换回来后，各个线程互不影响，执行引擎分毫不差的继续执行

    唯一没有定义 OOM 的区域

    CPU 时间片 - CPU 分配给各线程的时间段

  - JVM Stacks

虚拟机栈，每个线程创建时创建一个虚拟机栈，线程的每个方法调用对应虚拟机栈的一个栈帧（Stack Frame）

---生命周期---
与线程一致，随线程结束而销毁

---作用---
负责程序的运行，保存方法的局部变量，参与方法的调用和返回

---特点---
1. 快速有效的分配存储方式，JVM 对 JVM Stacks 的操作只有两个：入栈（执行），出栈（执行结束）
2. 不存在垃圾回收问题（存在栈溢出）

---异常---
StackOverflowError - JVM Stacks 大小固定时，超过最大容量
OutOfMemoryError - JVM Stacks 动态扩展时，超出系统内存容量

---栈帧---
栈的存储单元，对应线程中方法的调用；
如果该方法调用了其他方法，新的栈帧会被创建出来，放在栈顶；
方法返回，栈顶栈帧将执行结果传回前一个栈帧，并弹出栈顶；

---方法返回---
1. return 指令
2. 抛出异常

栈帧大小主要取决于局部变量表和操作数栈

- Local Variables

  局部变量表，一个数字数组，存储方法参数和方法内的局部变量（基本数据类型、对象引用，returnAddress类型）
  只在当前方法调用中有效，虚拟机通过局部变量表完成参数值到参数变量列表的传递

  Code.maximum local variables - 局部变量最大槽数，在编译期确定
  参数和局部变量越多，局部变量表越膨胀，栈帧越大，栈深越小

  局部变量必须显示赋值，否则编译不过

  局部变量表是垃圾回收根节点，也是栈帧中性能调优最需要注意的

  局部变量表线程独有，不存在线程安全问题，但引用对象不一定，对象在栈帧内消亡则安全，返回到外部则不安全

  - Slot

    变量槽，局部变量表的存储单元，存放 8 种基本类型、引用类型、returnAddress类型

    byte/short/char/boolean -> int 占一个 slot
    long/double 占两个 slot

    JVM 通过局部变量表槽位索引访问 slot
    方法被调用时，（this 引用），方法参数，方法内局部变量依次被复制到局部变量表中

    ---重复利用---
    局部变量过了其作用域时，之后声明的局部变量可以复用该局部变量的槽位

- Operand Stack

  操作数栈，LIFO，用来保存计算过程中间结果，变量的临时存储空间，可以存储 Java 的任意数据类型
  Java 虚拟机的解释引擎时基于栈，指的就是操作数栈

  32bit 占一个栈单位深度
  64bit 占两个栈单位深度

  Code.maxstack - 操作数栈最大深度，在编译期确定

  ---栈顶缓存技术---
  将栈顶元素全部缓存在物理 CPU 的寄存器中，提升执行引擎的执行效率（寄存器指令更少，执行速度快）

  - ++i vs. i++

    ++ 操作直接对局部变量表执行，不对操作数栈执行
    ++i 是先对局部变量表的变量自增，再从局部变量表取出该变量压入操作数栈
    i++ 是先从局部变量表取出该变量压入操作数栈，再对局部变量表中的变量自增

- Dynamic Linking

  动态链接，将 ClassFile 常量池中指向变量或方法的符号引用转换成调用运行时常量池中方法的直接引用

  通过对运行时常量池的动态链接，变量和方法可能只需要存储一份，节省了空间
  常量池作用：提供一些符号和常量，便于指令的识别

  - 链接

被调用的方法在编译期可知，且运行期保持不变，此时调用方法的符号引用转直接引用的过程称之为静态链接；
被调用的方法在编译期不可知，只能在程序运行期将符号引用转换成直接应用，这种具有动态性的转换过程称之为动态链接；

- 绑定机制

  绑定是一个字段、方法、类在符号引用转换成直接引用的过程，只会发生一次

  早期绑定是指在编译期可将方法与所属类型进行绑定，明确调用目标，对应静态链接
  晚期绑定是指只能在运行期根据实际类型绑定相关方法，对应动态链接

  用于实现面向对象编程所支持的封装、继承、多态三特性

- 虚方法与非虚方法

  非虚方法：在编译期确定了具体调用版本，运行期不会变，如：静态方法、私有方法、final 方法、实例构造器、父类方法
  虚方法：其他方法，Java 任何普通方法都具备 virtual 特性，可使用 final 抑制 virtual 特性

- 调用指令

  invokestatic - 调用静态方法，【链接.解析】阶段确定唯一方法版本
  invokespecial - 调用 <init>()，私有方法，父类方法，【链接.解析】阶段确认唯一方法
  invokevirtual - 调用所有虚方法
  invokeinterface - 调用接口方法
  ------
  invokedynamic - 动态解析出需要调用的方法，如 Lambda

- 静态语言与动态语言

  1. 动态语言的类型在运行期确定，静态语言的类型在编译期确定
  2. 动态语言判断变量值的类型，静态语言判断变量自身的类型

- 方法重写

  1. 取操作数栈栈顶元素操作对象的实例类型 C
  2. 查找与类型 C 中常量简单名称相符的方法，即可进行权限校验，通过则返回方法的直接引用，否则抛出异常
  java.lang.IllegalAccessError（一般编译期会直接提示异常）
  3. 若类型 C 中无该方法，则按照继承关系从下往上对 C 的各个父类进行查找
  4. 如果始终没有找到方法，则抛出 java.lang.AbstractMethodError

- 虚方法表

  避免面向对象编程中过多的动态分派，JVM 在类的方法区建立一个 virtual method table(非虚方法没有)，替代在类的元数据中查找，提高性能
  在类加载的链接阶段创建并初始化，类变量初始化完成时完成虚方法表的初始化

- Return Address

  方法返回地址，存放调用该方法的程序计数器值，用于方法返回，此时需要恢复到上层方法的局部变量表、操作数栈、将返回值压入调用者栈帧的操作数栈、设置程序计数器值等

  正常执行完成 - 通过程序计数器的值返回
  ireturn - boolean, byte, char, short, int
  lreturn - Long
  freturn - Float
  dreturn - Double
  areturn - 引用类型
  return - void
  出现未处理异常 - 通过异常表来确认返回地址

- 附加信息

  栈帧中还携带了一些 JVM 实现的相关附加信息，如程序调试支持的信息

- Native Method Stack

  本地方法栈，用于管理本地方法的调用，线程私有，登记 native 方法，执行引擎执行时加载本地方法库

  与 JVM Stacks 相同，支持固定大小（StackOverflowError）和动态扩展（OutOfMemoryError）

  ---注意---
  1. 进入本地方法后，不在受 JVM 限制
  2. 本地方法通过本地方法接口反问 JVM 内部运行时数据区
  3. 本地方法可以直接使用本地 CPU 寄存器
  4. 本地方法直接从本地内存的堆分配任意数量的内存
  5. 不是所有 JVM 都支持本地方法，Hotspot VM 支持

- Heap

堆，JVM 所有线程共享的内存空间，也会划分线程私有的缓冲区（TLAB，Thread Local Allocation Buffer)
堆是运行时分配所有对象实例及数组的地方，栈只会存放它们的引用
方法结束后堆中的对象不会被立刻移除，只在 GC 时被移除，堆是 GC 的重点区域

Heap 被划分为 Young Generation Space, Tenure Generation Space, Permanent Space/Metaspace，New:Old - 1:2
新生代，分为 Eden 和 Survivor，Survivor 有 From 和 To 两个，Eden:From:to - 8:1:1
几乎所有对象都是在 Eden 区被 new 出来，80% 朝生夕死，无法在 Eden 区存储的大对象直接进入老年代

-XX:NewRatio=2，设置新生代与老年代占比
-XX:SurvivorRatio=8，设置 Eden 与 Survivor 占比
-Xmn，新生代最大内存

---分代的理由---
优化 GC 性能，80% 的对象朝生夕死

YGC 导致 Survivor 区满后，对象可能直接晋入老年代

-Xms, 等价于 -XX:InitialHeapSize，初始堆内存
-Xmx, 等价于 -XX:MaxHeapSize，最大堆内存
两值通常设置相同，可以避免 GC 后重写计算和分隔堆取大小，提高性能

---查看堆内存分配情况---
jstat -gc pid
-XX:+PrintGCDetails

---常用调优工具---
1. JVM 参数
2. JConsole
3. JVisualVM
4. JProfiler
5. GCViewer
6. GCEasy
7. Java Flight Recorder

- 内存分配过程

  1. new 对象先放在 Eden
  2. Eden 满时，JVM 触发 Minor GC/YGC，销毁 Eden 中不再被 GCRoots 引用的对象，加载新 new 对象，然后将 Eden 区的对象转移到 Survivor(To)
  3. Eden 再满时，Survivor(To) 变为 Survivor(From)，YGC 销毁 Eden 和 From 中不被引用的对象，并将存活对象一致 Survivor(To，谁空谁是 To)
  4. 对象被 YGC 15次（默认）后，晋入老年区
  5. 老年区内存不足时，JVM 触发 Major/Full GC，销毁老年区不被引用的对象
  6. Major/Full GC 后老年区依旧无法存放对象，OOM

  - -XX:MaxTenuringThreshold

    设置 Promotion 年龄

- 内存分配策略

  1. 优先分配到 Eden
  2. 大对象直接进老年代
  3. 长期存活对象进老年代
  4. 动态对象年龄判断（Survivor 中相同年龄的对象超过半数，年年大于等于该年龄的直接进入老年代）
  5. 空间分配担保（-XX:HandlePromotionFailure），Minor GC 时无回收者，无法进入 Survivor 的直接进入老年代

  - -XX:HandlePromotionFailure

    是否启用空间分配担保，JDK6 Update 24 后不再使用
    1. Minor GC 前 JVM 判断老年代最大可用连续空间是否大于新生代所有对象总大小，大于则 Minor GC 安全
    2. 否则，查看是否担保，若担保，继续判断老年代最大连续空间是否大于历次晋升老年代对象的平均大小，大于则 Minor GC（即时有风险），否则 Full GC
    3. 若不担保，直接 Full GC

- 触发 Full GC

  1. 调用 System.gc()，系统建议 FullGC，非必然执行
  2. 老年代空间不足
  3. 方法区空间不足
  4. 晋入老年代的对象大小大于老年代可用内存大小
  5. 通过 Minor GC 晋入老年代的平均大小大于老年代的可用内存大小

- TLAB

  线程私有缓冲区，JVM 为每个线程分配，包含在 Eden，快速分配策略，对象首先会通过 TLAB 分配，失败则进入 Eden，此时为避免多个线程操作统一地址，需要使用加锁等机制
  多线程时可以避免多线程安全问题
  提高内存分配的吞吐量

  -XX:+UseTLAB - 启用TLAB
  -XX:TLABWasteTargetPersent - 设置 TLAB 在 Eden 的大小

- 逃逸分析

一个对象在方法中定义，但被外部所引用，则认为发生了逃逸。如：作为调用参数传递到其他方法

-XX:+DoEscapeAnalysis，显示启用逃逸分析，JKD1.7 后默认启用
-XX:+PrintEscapeAnalysis，查看逃逸分析筛选结果

通过逃逸分析，编译器对代码的优化：
1. 栈上分配 - 如果一个对象没有逃逸，将堆分配转化为栈分配，这样栈帧随着方法结束回收，局部变量对象也被回收，无须进行 GC
2. 同步省略 - 如果一个对象没有逃逸，可以确定这个对象只会有一个线程访问，因此可以不考虑同步（字节码中仍有同步，加载到内存时优化掉）
3. 分离对象或标量替换 - 如果一个对象没有逃逸，JIT 会拆解这个对象成若干标量，避免创建该对象，避免分配堆空间，为栈上分配提供了基础
标量 - 无法再分解的数据，基本数据类型
聚合量 - 可以分解的数据，对象
-XX:+EliminateAllocations，启用标量替换，默认打开

不足：技术不成熟，无法保证逃逸分析的优化多余它的性能消耗
Oracle Hotspot JVM 就未使用栈上分配，但使用了标量替换

intern 和 静态变量从永久代转移到了堆空间

- Metaspace(Method Area)

元空间，或方法区（类信息、运行时常量池、字符串字面量、数字常量），Non-Heap
方法区存放 Class 对象，堆主要存放实例对象
线程共享；JVM 启动时创建，关闭时释放；可以是固定大小，也可以是可扩展的，大小决定了可以保存多少个类；

---OOM 分析---
1. dump 快照分析内存泄漏（Memory Leak） or 内存溢出（Memory Overflow）
2. 内存泄漏可使用工具查看对象的 GC Roots 引用链，定位无法自动回收的代码位置
3. 无内存泄漏的内存溢出则调整堆大小，或优化代码逻辑，缩短对象生命周期

  - 结构

    1. 类型信息（class, interface, enum, annotation)
    类型的全限定名
    直接父类的全限定名
    修饰符（public，abstract，final）
    直接接口列表

    2. 域信息，声明有序
    域名，域类型，域修饰符（public，private，protected，static，final，volatile，transient）

    3. 方法信息，声明有序
    方法名
    返回类型（或void）
    参数的数量和类型（顺序）
    修饰符（public，private，protected，static，final，synchronized，native，abstract）
    方法的字节码（bytecodes），操作数栈，局部变量表，机器大小
    异常表（abstract 和 native 方法除外）

    4. 运行时常量池，加载 ClassFile 中常量池表到内存所得，具备动态性；编译器明确数值字面量，运行期解析的方法字段等引用，这里符号地址换为直接地址
    数据值
    字符串值
    类引用
    字段引用
    方法引用

    5. 静态变量
    non-final 的类变量 - 随类加载而加载
    static final 变量 - 编译器分配，显式赋值
    静态变量的实例也存在堆中

    6. JIT 代码缓存

  - 演进

JDK6: Permanent Generation, 静态变量存放在永久代

JDK7: Permanent Generation, 字符串字面量，静态变量存放在堆
-XX:Permsize 默认 20.75M
-XX:MaxPermsize 32位默认 64M，64位默认 82M

JDK8: Metaspace, 字符串常量，静态变量仍然在堆
-XX:MetaspaceSize 默认 21M
-XX:MaxMetaspaceSize 默认 -1，无限制
一旦触及高水位线，JVM触发 Full GC，卸载没用的类，然后重置高水位线（GC后元空间的两倍？）

OOM（java.lang.OutOfMemoryError: PermGen space/Metaspace）

元空间替代永久代的原因?
1. 为永久代设置空间大小很难确认，元空间不在使用 JVM 内存，而是使用本地内存，大小无限制
2. 永久代很难调优，很难降低 Full GC

StringTable 被放入堆的原因?
开发中大量使用字符串，在元空间被回收效率很低，导致其内存不足，在堆中可以即时回收内存

- GC

主要内容： 常量池（废弃的常量），类型（不再使用）

常量池：字面量、符号引用（类和接口的全限定名，字段名和描述符，方法名和描述符）
废弃常量：没有被任何地方引用的常量
不再使用的类型（同时满足）：
1. 所有实例及其子类实例已被回收
2. 该类的类加载器已被回收
3. 该类的 Class 对象没有被任何地方引用

- **Execution Engine**

3

执行引擎

- 解释器
- 即时编译器
- 垃圾回收器

- **Native Interface**

4

本地方法接口，实现一般有 C/C++ 完成，由 Java 调用；现越来越少，除非与硬件相关，其实也可以通过跨应用的网络调用代替

1. Java 环境交互 - 与操作系统硬件交互
2. 操作系统交互 - 调用底层操作系统本地代码库
3. Sun's Java - 实现 JVM 解释器

- **JVM 参数**

5

- 3 种参数类型
  - 标准参数
  - X 参数
  - XX 参数
    - Boolean 类型
      - -XX:+参数名

        开启参数，eg: -XX:-PrintGCDetails, 关闭 GC 详情输出
      - -XX:-参数名

        关闭参数
    - KeyValue 类型
      - -XX:参数Key=参数Value

        设置参数的值, eg: -XX:MetaspaceSize=21807104, 设置 Java 元空间的值
- 查看方式
  - jps
  - jinfo
    - jinfo -flag 参数名 pid

      查看指定进程的指定参数
    - jinfo -flags pid

      查看指定进程的所有参数
  - -XX:+PrintFlagsInitial

    java -XX:+PrintFlagsInitial, 打印参数默认值

- -XX:+PrintFlagsFinal
  java -XX:+PrintFlagsFinal, 打印参数最终值
  - =, 未修改
  - :=, 已修改
- -XX:+PrintCommandLineFlags
  打印简单初始参数

- 常用参数
  - -Xms
    等价于: -XX:InitialHeapSize, 初始堆内存，默认最大物理内存的1/64
  - -Xmx
    等价于: -XX:MaxHeapSize, 最大堆内存，默认最大物理内存的1/4
  - -Xss
    等价于: -XX:ThreadStackSize, 单个线程栈空间大小，默认值0，取决于平台
    - Linux/x64: 1024KB
    - OS X: 1024KB
    - Oracle Solaris: 1024KB
    - Windows: 取决于虚拟内存大小
  - -Xmn
    年轻代大小
  - -XX:MetaspaceSize
    元空间大小，元空间并不在虚拟机中，而是使用本地内存，默认 20MB+
  - -XX:PrintGCDetails
    打印 GC 收集详情信息
  - -XX:SurvivorRatio
    新生代中 Eden 和 S0/S1 的比例，默认 -XX:SurvivorRatio=8，即Eden:S0:S1 = 8:1:1
  - -XX:NewRatio
    堆结构中老年代:新生代的值，默认 2
  - -XX:MaxTenuringThreshold
    垃圾最大年龄，默认 15，取值0~15，超过 15 次交换就存入老年代

- **OOM**
  6
  - StackOverflowError
    栈过深
  - OutOfMemoryError:java heap space
    new 的对象过多
  - OutOfMemoryError:GC overhead limit exceeded
    超过 98% 的时间在 GC
  - OutOfMemoryError:Direct buffer memory
    OS 本地内存已满
    - -XX:MaxDirectMemorySize，设置最大堆外内存
      - ByteBuffer.allocate(capability)
      - ByteBuffer.allocteDirect(capability)
    - 场景：Netty
  - OutOfMemoryError:unable to create new native thread
    OS Thread 达到极限
  - OutOfMemoryError:Metaspace
    元空间内存不足
    - -XX:MetaspaceSize，默认 20M
    - Metaspace 存放的信息
      - VM 加载的类信息
      - 常量池
      - 静态变量
      - JIT 代码缓存