
第一节：内存结构

为什么要了解虚拟机

JVM 不单单只支持 Java 语言，也支持其他语言（Scala、Kotlin、Groovy 等等）

区块链 2.0--以太坊(比特币是区块链 1.0) 中提供了 EVM 的虚拟机，它的实现和 JVM 类似，基于栈、生成脚本编译成字节码来执行。知识通用。（理论大于实际）

虚拟机历史

了解即可，无需关注

解释执行和编译执行（针对字节码的执行）

解释执行就是边翻译为机器码边执行、即时编译（编译执行）就是先将一个方法中的所有字节码全部编译成机器码之后再执行。

Hotspot 采用的是先解释执行，到了一定时机后热点代码（多次执行、循环等）再翻译成机器码

热点代码探测技术（通过执行计数器找到最有编译价值的代码，如果代码用得非常频繁，就会把这些代码编译成本地代码）。

JRockit 采取的方法是在执行 class 时直接编译为机器码（Java 程序启动速度会比较慢）

J9 和 Hotspot 比较接近，主要是用在 IBM 产品（IBM WebSphere 和 IBM 的 AIX 平台上），华为有的项目用的 J9。

谷歌：Google Android Dalvik VM：使用的寄存器架构，执行 dex（Dalvik Executable）通过 class 转化而来。

未来的 Java 技术

模块化:OSGI (动态化、模块化) , 应用层面就是微服务, 互联网的发展方向

混合语言: 多个语言都可以运行在 JVM 中, google 的 Kotlin 成为了 Android 的官方语言。Scala(Kafka)

多核并行: CPU 从高频次转变为多核心, 多核时代。JDK1.7 引入了 Fork/Join, JDK1.8 提出 lambda 表达式(函数式编程天生适合并行运行)

丰富语法: JDK5 提出自动装箱、泛型(并发编程讲到)、动态注解等语法。JDK7 二进制原生支持。try-catch-finally 至 try-with-resource

64 位: 虽然同样的程序 64 位内存消耗比 32 位要多一点, 但是支持内存大, 所以虚拟机都会完全过渡到 64 位, 32 位的 JVM 有 4G 的堆大小限制 (寻址范围 2 的 32 次方)。

更强的垃圾回收器 (现在主流 CMS、G1) : JDK11 - ZGC (暂停时间不超过 10 毫秒, 且不会随着堆的增加而增加, TB 级别的堆回收)): 有色指针、加载屏障。JDK12 支持并发类卸载, 进一步缩短暂停时间。JDK13(计划于 2019 年 9 月)将最大堆大小从 4TB 增加到 16TB

Java SE 体系架构

JavaSE, Java 平台标准版, 为 Java EE 和 Java ME 提供了基础。

JDK: Java 开发工具包, JDK 是 JRE 的超集, 包含 JRE 中的所有内容, 以及开发程序所需的编译器和调试程序等工具。

JRE: Java SE 运行时环境 , 提供库、Java 虚拟机和其他组件来运行用 Java 编程语言编写的程序。主要类库, 包括: 程序部署发布、用户界面工具类、继承库、其他基础库, 语言和工具基础库

JVM: java 虚拟机, 负责 JavaSE 平台的硬件和操作系统无关性、编译执行代码 (字节码) 和平台安全性

运行时数据区域

这个是抽象概念, 内部实现依赖寄存器、高速缓存、主内存 (具体要分析 JVM 源码 C++语言实现, 没必要看)

计算机的运行=指令+数据, 指令用于执行方法的, 数据用于存放数据和对象的。

虚拟机栈---执行 java 方法、本地方法栈---执行本地方法、程序计数器---程序执行的计数器

Java 中的数据：变量、常量、对象、数组相关。

线程私有

程序计数器

较小的内存空间，当前线程执行的字节码的行号指示器；各线程之间独立存储，互不影响（面试可能问到为什么需要）

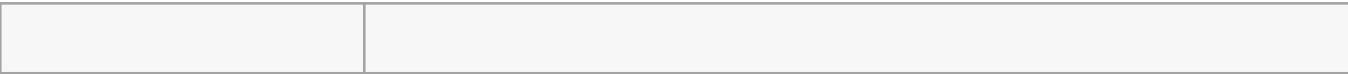
如果线程正在执行的是一个 Java 方法，则指明当前线程执行的字节码行数

如果正在执行的是 Native 方法，这个计数器值则为空（Undefined）

此内存区域是唯一一个不会出现 OutOfMemoryError 情况的区域。

虚拟机栈（JVM 后续的执行子程序有详细的见解）

iload_1	第二个 int 型局部变量进栈
bipush	将一个 byte 型常量值推送至栈顶
isub	栈顶两 int 型数值相减，并且结果进栈
istore_1	将栈顶 int 型数值存入第二个局部变量



栈:

数据结构的特点和 java 中方法中调用方法的特性一致。（为什么 JVM 使用栈 - 演示代码 StackFilo）

虚拟机栈:

异常:

线程请求的栈深度大于虚拟机所允许的深度: `StackOverflowError`

JVM 动态扩展时无法申请到足够的内存时: `OutOfMemoryError`

虚拟机栈:

每个线程私有的，线程在运行时，在执行每个方法的时候都会打包成一个栈帧，存储了局部变量表，操作数栈，动态链接，方法出口等信息，然后放入栈。每个时刻正在执行的当前方法就是虚拟机栈顶的栈帧。方法的执行就对应着栈帧在虚拟机栈中入栈和出栈的过程。

栈的大小缺省为 1M，可用参数 `-Xss` 调整大小，例如`-Xss256k`

在编译程序代码的时候，栈帧中需要多大的局部变量表，多深的操作数栈都已经完全确定了，并且写入到方法表的 `Code` 属性之中，因此一个栈帧需要分配多少内存，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:顾名思义就是局部变量的表，用于存放我们的局部变量的。首先它是一个 32 位的长度，主要存放我们的 Java 的八大基础数据类型，一般 32 位就可以存放下，如果是 64 位的就使用高低位占用两个也可以存放下，如果是局部的一些对象，比如我们的 `Object` 对象，我们只需要存放它的一个引用地址即可。（基本数据类型、对象引用、`returnAddress` 类型）

操作数据栈: 存放我们方法执行的操作数的，它就是一个栈，先进后出的栈结构，操作数栈，就是用来操作的，操作的的元素可以是任意的 java 数据类型，所以我们知道一个方法刚刚开始的时候，这个方法的操作数栈就是空的，操作数栈运行方法是会一直运行入栈/出栈的操作

动态连接:Java 语言特性多态（需要类加载、运行时才能确定具体的方法，后续有详细的讲解）

返回地址:

正常返回：（调用程序计数器中的地址作为返回）

三步曲：

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、
异常的话：（通过异常处理器表<非栈帧中的>来确定）

本地方法栈

各虚拟机自由实现，本地方法栈 native 方法调用 JNI 到了底层的 C/C++(c/c++)可以触发汇编语言，然后驱动硬件)

线程共享的区域

方法区/永久代

用于存储已经被虚拟机加载的类信息，常量("zdy", "123"等)，静态变量(static 变量)等数据，可用以下参数调整：

jdk1.7 及以前： -XX:PermSize； -XX:MaxPermSize；

jdk1.8 以后： -XX:MetaspaceSize； -XX:MaxMetaspaceSize

jdk1.8 以后大小就只受本机总内存的限制

如： -XX:MaxMetaspaceSize=3M

类信息：

类的完整有效名、返回值类型、修饰符（public, private...）、变量名、方法名、方法代码、这个类型直接父类的完整有效名(除非这个类型是 interface 或是 java.lang.Object，两种情况下都没有父类)、类的直接接口的一个有序列表

堆

几乎所有对象都分配在这里，也是垃圾回收发生的主要区域，可用以下参数调整：

- Xms: 堆的最小值；
- Xmx: 堆的最大值；
- Xmn: 新生代的大小；
- XX:NewSize: 新生代最小值；
- XX:MaxNewSize: 新生代最大值；

例如- Xmx256m

运行时常量池

符号引用（一个概念）

一个 java 类（假设为 People 类）被编译成一个 class 文件时，如果 People 类引用了 Tool 类，但是在编译时 People 类并不知道引用类的实际内存地址，因此只能使用符号引用来代替。

而在类装载器装载 People 类时，此时可以通过虚拟机获取 Tool 类的实际内存地址，因此便可以既将符号 org.simple.Tool 替换为 Tool 类的实际内存地址，及直接引用地址。

即在编译时用符号引用来代替引用类，在加载时再通过虚拟机获取该引用类的实际地址。

以一组符号来描述所引用的目标，符号可以是任何形式的字面量，只要使用时能无歧义地定位到目标即可。符号引用与虚拟机实现的内存布局是无关的，引用的目标不一定已经加载到内存中。

字面量

文本字符串 String a = "abc",这个 abc 就是字面量

八种基本类型 int a = 1; 这个 1 就是字面量

声明为 final 的常量

常量池的变化

各版本之间的变化

见课件

直接内存

使用 Native 函数库直接分配堆外内存(NIO)

并不是 JVM 运行时数据区域的一部分，但是会被频繁使用(可以通过-XX:MaxDirectMemorySize 来设置 (不设置的话默认与堆内存最大值一样,也会出现 OOM 异常)

使用直接内存避免了在 Java 堆和 Native 堆中来回复制数据，能够提高效率

测试用例 JavaStack: 设置 JVM 参数-Xmx100m, 运行异常, 因为如果没设置-XX:MaxDirectMemorySize, 则默认与-Xmx 参数值相同为 100M, 分配 128M 直接内存超出限制范围

站在线程角度来看

虚拟机栈、本地方法栈、程序计数器三个区域的生命周期和线程相同。

线程共享区域：设计到生命周期管理和垃圾回收等概念，后续章节有细讲。

深入辨析堆和栈

■ 功能

- 以栈帧的方式存储方法调用的过程，并存储方法调用过程中基本数据类型的变量（int、short、long、byte、float、double、boolean、char 等）以及对象的引用变量（reference），其内存分配在栈上，变量出了作用域就会自动释放；
- 而堆内存用来存储 Java 中的对象。无论是成员变量，局部变量，还是类变量，它们指向的对象都存储在堆内存中；

■ 线程独享还是共享

- 栈内存归属于单个线程，每个线程都会有一个栈内存，其存储的变量只能在其所属线程中可见，即栈内存可以理解成线程的私有内存。

-
- 堆内存中的对象对所有线程可见。堆内存中的对象可以被所有线程访问。

- 空间大小

栈的内存要远远小于堆内存

栈溢出

参数: -Xss256k

`java.lang.StackOverflowError` 一般的方法调用是很难出现的，如果出现了可能会是无限递归。

虚拟机栈带给我们的启示：方法的执行因为要打包成栈桢，所以天生要比实现同样功能的循环慢，所以树的遍历算法中：递归和非递归(循环来实现)都有存在的意义。递归代码简洁，非递归代码复杂但是速度较快。

`OutOfMemoryError`: 不断建立线程。（一般演示不出，演示出来机器也死了）

第二节：JVM 中的对象

虚拟机中的对象

对象的分配

虚拟机遇到一条 new 指令时：根据 new 的参数是否能在常量池中定位到一个类的符号引用，如果没有，说明还未定义该类，抛出 ClassNotFoundException；

1) 检查加载

先执行相应的类加载过程。如果没有，则进行类加载

2) 分配内存

根据方法区的信息确定为该类分配的内存空间大小

指针碰撞 (java 堆内存空间规整的情况下使用)

接下来虚拟机将为新生对象分配内存。为对象分配空间的任务等同于把一块确定大小的内存从 Java 堆中划分出来。

如果 Java 堆中内存是绝对规整的，所有用过的内存都放在一边，空闲的内存放在另一边，中间放着一个指针作为分界点的指示器，那所分配内存就仅仅是把那个指针向空闲空间那边挪动一段与对象大小相等的距离，这种分配方式称为“**指针碰撞**”。

空闲列表 (java 堆空间不规整的情况下使用)

如果 Java 堆中的内存并不是规整的，已使用的内存和空闲的内存相互交错，那就没有办法简单地进行指针碰撞了，虚拟机就必须维护一个列表，记录上哪些内存块是可用的，在分配的时候从列表中找到一块足够大的空间划分给对象实例，并更新列表上的记录，这种分配方式称为“**空闲列表**”。

选择哪种分配方式由 Java 堆是否规整决定，而 Java 堆是否规整又由所采用的垃圾收集器是否带有压缩整理功能决定。

并发安全

除如何划分可用空间之外，还有另外一个需要考虑的问题是对象创建在虚拟机中是非常频繁的行为，即使是仅仅修改一个指针所指向的位置，在并发情况下也并不是线程安全的，可能出现正在给对象 A 分配内存，指针还没来得及修改，对象 B 又同时使用了原来的指针来分配内存的情况。

CAS 机制

解决这个问题有两种方案，一种是对分配内存空间的动作进行同步处理——实际上虚拟机采用 CAS 配上失败重试的方式保证更新操作的原子性；

分配缓冲

另一种是把内存分配的动作按照线程划分在不同的空间之中进行，即每个线程在 Java 堆中预先分配一小块私有内存，也就是本地线程分配缓冲（Thread Local Allocation Buffer,TLAB），如果设置了虚拟机参数 `-XX:+UseTLAB`，在线程初始化时，同时也会申请一块指定大小的内存，只给当前线程使用，这样每个线程都单独拥有一个 Buffer，如果需要分配内存，就在自己的 Buffer 上分配，这样就不存在竞争的情况，可以大大提升分配效率，当 Buffer 容量不够的时候，再重新从 Eden 区域申请一块继续使用。

TLAB 的目的是在为新对象分配内存空间时，让每个 Java 应用线程能在使用自己专属的分配指针来分配空间(Eden 区，默认 Eden 的 1%)，减少同步开销。

TLAB 只是让每个线程有私有的分配指针，但底下存对象的内存空间还是给所有线程访问的（类似于堆），只是其它线程无法在这个区域分配而已。当一个 TLAB 用满（分配指针 `top` 撞上分配极限 `end` 了），就新申请一个 TLAB。

3) 内存空间初始化

（注意不是构造方法）内存分配完成后，虚拟机需要将分配到的内存空间都初始化为零值(如 `int` 值为 0, `boolean` 值为 `false` 等等)。这一步操作保证了对象的实例字段在 Java 代码中可以不赋初始值就直接使用，程序能访问到这些字段的数据类型所对应的零值。

4) 设置

接下来，虚拟机要对对象进行必要的设置，例如这个对象是哪个类的实例、如何才能找到类的元数据信息、对象的哈希码、对象的 GC 分代年龄等信息。这些信息存放在对象的对象头之中。

5) 对象初始化

在上面工作都完成之后，从虚拟机的视角来看，一个新的对象已经产生了，但从 Java 程序的视角来看，对象创建才刚刚开始，所有的字段都还为零值。所以，一般来说，执行 new 指令之后会接着把对象按照程序员的意愿进行初始化，这样一个真正可用的对象才算完全产生出来。

对象的内存布局

在 HotSpot 虚拟机中，对象在内存中存储的布局可以分为 3 块区域：对象头（Header）、实例数据（Instance Data）和对齐填充（Padding）。

对象头包括两部分信息，第一部分用于存储对象自身的运行时数据，如哈希码（HashCode）、GC 标志、对象分代年龄、锁状态标志、线程持有的锁、偏向线程 ID、偏向时间戳等。

对象头的另外一部分是类型指针，即对象指向它的类元数据的指针，虚拟机通过这个指针来确定这个对象是哪个类的实例。

第三部分对齐填充并不是必然存在的，也没有特别的含义，它仅仅起着占位符的作用。由于 HotSpot VM 的自动内存管理系统要求对象的大小必须是 8 字节的整数倍。对象正好是 9 字节的整数，所以当对象其他数据部分（对象实例数据）没有对齐时，就需要通过对齐填充来补全。

对象的访问定位

建立对象是为了使用对象，我们的 Java 程序需要通过栈上的 reference 数据来操作堆上的具体对象。目前主流的访问方式有使用句柄和直接指针两种。

句柄

如果使用句柄访问的话，那么 Java 堆中将会划分出一块内存来作为句柄池，reference 中存储的就是对象的句柄地址，而句柄中包含了对象实例数据与类型数据各自的具体地址信息。

直接指针

如果使用直接指针访问，reference 中存储的直接就是对象地址。

这两种对象访问方式各有优势，使用句柄来访问的最大好处就是 reference 中存储的是稳定的句柄地址，在对象被移动（垃圾收集时移动对象是非常普遍的行为）时只会改变句柄中的实例数据指针，而 reference 本身不需要修改。

使用直接指针访问方式的最大好处就是速度更快，它节省了一次指针定位的时间开销，由于对象的访问在 Java 中非常频繁，因此这类开销积少成多后也是一项非常可观的执行成本。

对 Sun HotSpot 而言，它是使用直接指针访问方式进行对象访问的。

堆内存分配策略

新生代

Eden 区

Survivor(from)区：

设置 Survivor 是为了减少送到老年代的对象

Survivor(to)区：

设置两个 Survivor 区是为了解决碎片化的问题（复制回收算法）

对象优先在 **Eden** 区分配

虚拟机参数:

-Xms20m 堆空间初始 20m

-Xmx20m 堆空间最大 20m

-Xmn10m 新生代空间 10m

-XX:+PrintGCDetails 打印垃圾回收日志，程序退出时输出当前内存的分配情况

注意：新生代初始时就有大小

大多数情况下，对象在新生代 **Eden** 区中分配。当 **Eden** 区没有足够空间分配时，虚拟机将发起一次 **Minor GC**。

大对象直接进入老年代

-Xms20m

-Xmx20m

-Xmn10m

-XX:+PrintGCDetails

-XX:PretenureSizeThreshold=4m 超过多少大小的对象直接进入老年代

-XX:+UseSerialGC

PretenureSizeThreshold 参数只对 **Serial** 和 **ParNew** 两款收集器有效。

最典型的大对象是那种很长的字符串以及数组。这样做的目的：1.避免大量内存复制,2.避免提前进行垃圾回收，明明内存有空间进行分配。

长期存活对象进入老年区

如果对象在 Eden 出生并经过第一次 Minor GC 后仍然存活，并且能被 Survivor 容纳的话，将被移动到 Survivor 空间中，并将对象年龄设为 1，对象在 Survivor 区中每熬过一次 Minor GC，年龄就增加 1，当它的年龄增加到一定程度(默认为 15) 时，就会被晋升到老年代中。

对象年龄动态判定

如果在 Survivor 空间中相同年龄所有对象大小的综合大于 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

空间分配担保

在发生 Minor GC 之前，虚拟机会先检查老年代最大可用的连续空间是否大于新生代所有对象总空间，如果这个条件成立，那么 Minor GC 可以确保是安全的。如果不成立，则虚拟机会查看 HandlePromotionFailure 设置值是否允许担保失败。如果允许，那么会继续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，尽管这次 Minor GC 是有风险的，如果担保失败则会进行一次 Full GC；如果小于，或者 HandlePromotionFailure 设置不允许冒险，那这时也要改为进行一次 Full GC。

HotSpot 默认是开启空间分配担保的。

Java 中的泛型

泛型是什么

泛型，即“参数化类型”。一提到参数，最熟悉的就是定义方法时有形参，然后调用此方法时传递实参。那么参数化类型怎么理解呢？

顾名思义，就是将类型由原来的具体的类型参数化，类似于方法中的变量参数，此时类型也定义成参数形式（可以称之为类型形参），然后在使用/调用时传入具体的类型（类型实参）。

泛型的本质是为了参数化类型（在不创建新的类型的情况下，通过泛型指定的不同类型来控制形参具体限制的类型）。也就是说在泛型使用过程中，操作的数据类型被指定为一个参数，这种参数类型可以用在类、接口和方法中，分别被称为泛型类、泛型接口、泛型方法。

引入一个类型变量 T（其他大写字母都可以，不过常用的就是 T, E, K, V 等等），并且用<>括起来，并放在类名的后面。泛型类是允许有多个类型变量的。

泛型类

```
/*
public class NormalGeneric<T> {
    private T data;

    public NormalGeneric() {
    }

    public NormalGeneric(T data) {
        this();
        this.data = data;
    }
}
```

泛型接口

泛型接口与泛型类的定义基本相同。

```
public interface Generator<T> {  
    public T next();  
}
```

而实现泛型接口的类，有两种实现方法：

1、未传入泛型实参时：

```
public class ImplGenerator<T> implements Generator<T> {  
  
    private T data;
```

在 new 出类的实例时，需要指定具体类型：

```
public static void main(String[] args) {  
    ImplGenerator<String> implGenerator = new
```

2、传入泛型实参

```
public class ImplGenerator2 implements Generator<String> {  
    @Override  
    public String next() {  
        return "King";  
    }
```

在 new 出类的实例时，和普通的类没区别。

泛型方法

```
/*
 * 泛型方法
 * 引入一个类型变量T（其他大写字母都可以，不过常用的就是T, E, K, V等等）
 */
public class GenericMethod {
    //泛型方法
    public <T> T genericMethod(T t) { return t; }
    //普通方法
    public void test(int x, int y){
        System.out.println(x+y);
    }

    public static void main(String[] args) {
        GenericMethod genericMethod = new GenericMethod();
        genericMethod.test( x: 13, y: 7 );
        System.out.println(genericMethod.<~>genericMethod( t: "King" ));
        System.out.println(genericMethod.genericMethod( t: 180));
    }
}
```

泛型方法，是在调用方法的时候指明泛型的具体类型，泛型方法可以在任何地方和任何场景中使用，包括普通类和泛型类。

为什么我们需要泛型？

通过两段代码我们就可以知道为何我们需要泛型

```
public int addInt(int x,int y){  
    return x+y;  
}  
  
public float addFloat(float x,float y){  
    return x+y;  
}
```

实际开发中，经常有数值类型求和的需求，例如实现 int 类型的加法，有时候还需要实现 long 类型的求和，如果还需要 double 类型的求和，需要重新在重载一个输入是 double 类型的 add 方法。

所以泛型的好处就是：

- 适用于多种数据类型执行相同的代码
- 泛型中的类型在使用时指定，不需要强制类型转换

虚拟机是如何实现泛型的？

Java 语言中的泛型，它只在程序源码中存在，在编译后的字节码文件中，就已经替换为原来的原生类型（Raw Type，也称为裸类型）了，并且在相应的地方插入了强制转型代码，因此，对于运行期的 Java 语言来说，`ArrayList<int>`与`ArrayList<String>`就是同一个类，所以泛型技术实际上是 Java 语言的一颗语法糖，Java 语言中的泛型实现方法称为类型擦除，基于这种方法实现的泛型称为伪泛型。

将一段 Java 代码编译成 Class 文件，然后再用字节码反编译工具进行反编译后，将会发现泛型都不见了，程序又变回了 Java 泛型出现之前的写法，泛型类型都变回了原生类型（因为）

```
/**  
 * 泛型擦除  
 */  
  
public class Theory {  
    public static void main(String[] args) {  
        Map<String, String> map = new HashMap<>();  
        map.put("King", "18");  
        System.out.println(map.get("King"));  
    }  
}
```

Theory.class

```
package com.jvm.ch02.gengric;  
  
import java.io.PrintStream;  
import java.util.HashMap;  
import java.util.Map;  
  
public class Theory {  
    public static void main(String[] args) {  
        Map<String, String> map = new HashMap<>();  
        map.put("King", "18");  
        System.out.println((String)map.get("King"));  
    }  
}
```

泛型擦除

使用泛型注意事项（小甜点，了解即可，装 B 专用）

```
public static String method(List<String> stringList){  
    System.out.println("List");  
    return "OK";  
}  
  
public static Integer method(List<Integer> integerList){  
    System.out.println("List");  
    return 0;  
}
```

上面这段代码是不能被编译的，因为参数 `List<Integer>` 和 `List<String>` 编译之后都被擦除了，变成了一样的原生类型 `List<E>`，擦除动作导致这两种方法的特征签名变得一模一样（注意在 IDEA 中是不行的，但是 jdk 的编译器是可以，因为 jdk 是根据方法返回值+方法名+参数）。

JVM 版本兼容性问题：JDK1.5 以前，为了确保泛型的兼容性，JVM 除了擦除，其实还是保留了泛型信息(Signature 是其中最重要的一项属性，它的作用就是存储一个方法在字节码层面的特征签名，这个属性中保存的参数类型并不是原生类型，而是包括了参数化类型的信息)---弱记忆

另外，从 Signature 属性的出现我们还可以得出结论，擦除法所谓的擦除，仅仅是对方法的 Code 属性中的字节码进行擦除，实际上元数据中还是保留了泛型信息，这也是我们能通过反射手段取得参数化类型的根本依据。

第三节：垃圾回收算法与垃圾回收器

学习垃圾回收的意义

Java 与 C++ 等语言最大的技术区别：**自动化的**垃圾回收机制（GC）

为什么要了解 GC 和内存分配策略

- 1、面试需要
- 2、GC 对应用的性能是有影响的；
- 3、写代码有好处

栈：栈中的生命周期是跟随线程，所以一般不需要关注

堆：堆中的对象是垃圾回收的重点

方法区/元空间：这一块也会发生垃圾回收，不过这块的效率比较低，一般不是我们关注的重点

判断对象的存活

引用计数法

给对象添加一个引用计数器，当对象增加一个引用时计数器加 1，引用失效时计数器减 1。引用计数为 0 的对象可被回收。（Python 在用，但主流虚拟机没有使用）

优点：快，方便，实现简单。

缺陷：对象相互引用时（A.instance=B 同时 B.instance=A），很难判断对象是否该回收。

可达性分析（Java 中使用）

（面试时重要的知识点，牢记）

来判定对象是否存活的。这个算法的基本思路就是通过一系列的称为“GC Roots”的对象作为起始点，从这些节点开始向下搜索，搜索所走过的路径称为引用链（Reference Chain），当一个对象到 GC Roots 没有任何引用链相连时，则证明此对象是不可用的。

作为 GC Roots 的对象包括下面几种：

- 1.当前虚拟机栈中局部变量表中的引用的对象
- 2.当前本地方法栈中局部变量表中的引用的对象
- 3.方法区中类静态属性引用的对象
- 4.方法区中的常量引用的对象

请忘记 `finalize`

`finalize` 可以完成对象的拯救，但是 JVM 不保证一定能执行，所以请忘记这个“坑”。

各种引用（Reference）

传统定义：Reference 中存储的数据代表的是另一块内存的起始地址。

强引用

一般的 `Object obj = new Object()`，就属于强引用。

（如果有 GCroots 的强引用）垃圾回收器绝对不会回收它，当内存不足时宁愿抛出 OOM 错误，使得程序异常停止，也不会回收强引用对象。

软引用 SoftReference

垃圾回收器在内存充足的时候不会回收它，而在内存不足时会回收它

软引用非常适合于创建缓存。当系统内存不足的时候，缓存中的内容是可以被释放的。

一些有用但是并非必需，用软引用关联的对象，系统将要发生 OOM 之前，这些对象就会被回收。参见代码：

VM 参数 -Xms10m -Xmx10m -XX:+PrintGC

```
//  
public static void main(String[] args) {  
    User u = new User( id: 1, name: "King"); //new是强引用  
    SoftReference<User> userSoft = new SoftReference<~>(u);  
    u = null;//干掉强引用，确保这个实例只有userSoft的软引用  
    System.out.println(userSoft.get());  
    System.gc(); //进行一次GC垃圾回收  
    System.out.println("After gc");  
    System.out.println(userSoft.get());  
    //往堆中填充数据，导致OOM  
    List<byte[]> list = new LinkedList<>();  
    try {  
        for(int i=0;i<100;i++) {  
            System.out.println("*****"+userSoft.get());  
            list.add(new byte[1024*1024*1]); //1M的对象  
        }  
    } catch (Throwable e) {  
        //抛出了OOM异常时打印软引用对象  
        System.out.println("Exception*****"+userSoft.get());  
    }  
}
```

运行结果

```
User [id=1, name=King]
[GC (System.gc()) 1616K->664K(9728K), 0.0022102 secs]
[Full GC (System.gc()) 664K->594K(9728K), 0.0067387 secs]
After gc
User [id=1, name=King]
*****User [id=1, name=King]
[GC (Allocation Failure) -- 7920K->7928K(9728K), 0.0008810 secs]
[Full GC (Ergonomics) 7928K->7766K(9728K), 0.0071010 secs]
[GC (Allocation Failure) -- 7766K->7766K(9728K), 0.0004155 secs]
[Full GC (Allocation Failure) 7766K->7748K(9728K), 0.0064581 secs]
Exception*****null
```



例如，一个程序用来处理用户提供的图片。如果将所有图片读入内存，这样虽然可以很快的打开图片，但内存空间使用巨大，一些使用较少的图片浪费内存空间，需要手动从内存中移除。如果每次打开图片都从磁盘文件中读取到内存再显示出来，虽然内存占用较少，但一些经常使用的图片每次打开都要访问磁盘，代价巨大。这个时候就可以用软引用构建缓存。

弱引用 WeakReference

垃圾回收器在扫描到该对象时，无论内存充足与否，都会回收该对象的内存。

一些有用（程度比软引用更低）但是并非必需，用弱引用关联的对象，只能生存到下一次垃圾回收之前，GC发生时，不管内存够不够，都会被回收。
参看代码：

```
public static void main(String[] args) {  
    User u = new User(id: 1, name: "King");  
    WeakReference<User> userWeak = new WeakReference<~>(u);  
    u = null; // 干掉强引用，确保这个实例只有 userWeak 的弱引用  
    System.out.println(userWeak.get());  
    System.gc(); // 进行一次 GC 垃圾回收  
    System.out.println("After gc");  
    System.out.println(userWeak.get());  
}
```

```
TestWeakRef x  
"C:\Program Files\Java\jdk1.8.0_191\bin\java.exe" ...  
User [id=1, name=King]  
After gc  
null
```

注意：软引用 SoftReference 和弱引用 WeakReference，可以用在内存资源紧张的情况下以及创建不是很重要的数据缓存。当系统内存不足的时候，缓存中的内容是可以被释放的。

实际运用（WeakHashMap、ThreadLocal）

虚引用 PhantomReference

幽灵引用，最弱，被垃圾回收的时候收到一个通知

如果一个对象只具有虚引用，那么它和没有任何引用一样，任何时候都可能被回收。

虚引用主要用来跟踪对象被垃圾回收器回收的活动

GC (Garbage Collection)

案例 Oom 类

-Xms 堆区内存初始内存分配的大小

-Xmx 堆区内存可被分配的最大上限

-XX:+PrintGCDetails

打印 GC 详情

-XX:+HeapDumpOnOutOfMemoryError

当堆内存空间溢出时输出堆的内存快照

新生代大小配置参数的优先级:

中间 -Xmn 限定大小

-XX:SurvivorRatio

Survivor 区和 Eden 区的比值

8 表示 两个 Eden : Survivor = 8: 2 , 每个 Survivor 占 1/10

可以修改为 2

2 表示 两个 Eden : Survivor = 2: 2 , 各占一半

GC overhead limit exceeded 超过 98% 的时间用来做 GC 并且回收了不到 2% 的堆内存时会抛出此异常

1. 垃圾回收会占据资源

2. 回收效率过低也会有限制

Minor GC

特点: 发生在新生代上, 发生的较频繁, 执行速度较快

触发条件: Eden 区空间不足\空间分配担保

Full GC

特点:主要发生在老年代上（新生代也会回收），较少发生，执行速度较慢

触发条件:

调用 `System.gc()`

老年代区域空间不足

空间分配担保失败

JDK 1.7 及以前的永久代(方法区)空间不足

CMS GC 处理浮动垃圾时，如果新生代空间不足，则采用空间分配担保机制，如果老年代空间不足，则触发 Full GC

垃圾回收算法

复制算法（Copying）

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。这样使得每次都是对整个半区进行内存回收，内存分配时也就不用考虑内存碎片等复杂情况，只要按顺序分配内存即可，实现简单，运行高效。只是这种算法的代价是将内存缩小为了原来的一半。

注意：内存移动是必须实打实的移动（复制），不能使用指针玩。

专门研究表明，新生代中的对象 90% 是“朝生夕死”的，所以一般来说回收占据 10% 的空间够用了，所以并不需要按照 1:1 的比例来划分内存空间，而是将内存分为一块较大的 Eden 空间和两块较小的 Survivor 空间，每次使用 Eden 和其中一块 Survivor[1]。当回收时，将 Eden 和 Survivor 中还活着的对象一次性地复制到另外一块 Survivor 空间上，最后清理掉 Eden 和刚才用过的 Survivor 空间。

HotSpot 虚拟机默认 Eden 和 Survivor 的大小比例是 8:1，也就是每次新生代中可用内存空间为整个新生代容量的 90%（80%+10%），只有 10% 的内存会被“浪费”。

标记-清除算法（Mark-Sweep）

过程:

1. 首先标记所有需要回收的对象
2. 统一回收被标记的对象

缺点:

- 1.效率问题，标记和清除效率都不高
- 2.标记清除之后会产生大量不连续的内存碎片，空间碎片太多可能会导致以后在程序运行过程中需要分配较大对象时，无法找到足够的连续内存而不得不提前触发另一次垃圾收集动作。

标记-整理算法（Mark-Compact）

首先标记出所有需要回收的对象，在标记完成后，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。

垃圾回收器

分代收集

根据各个年代的特点选取不同的垃圾收集算法

新生代使用复制算法

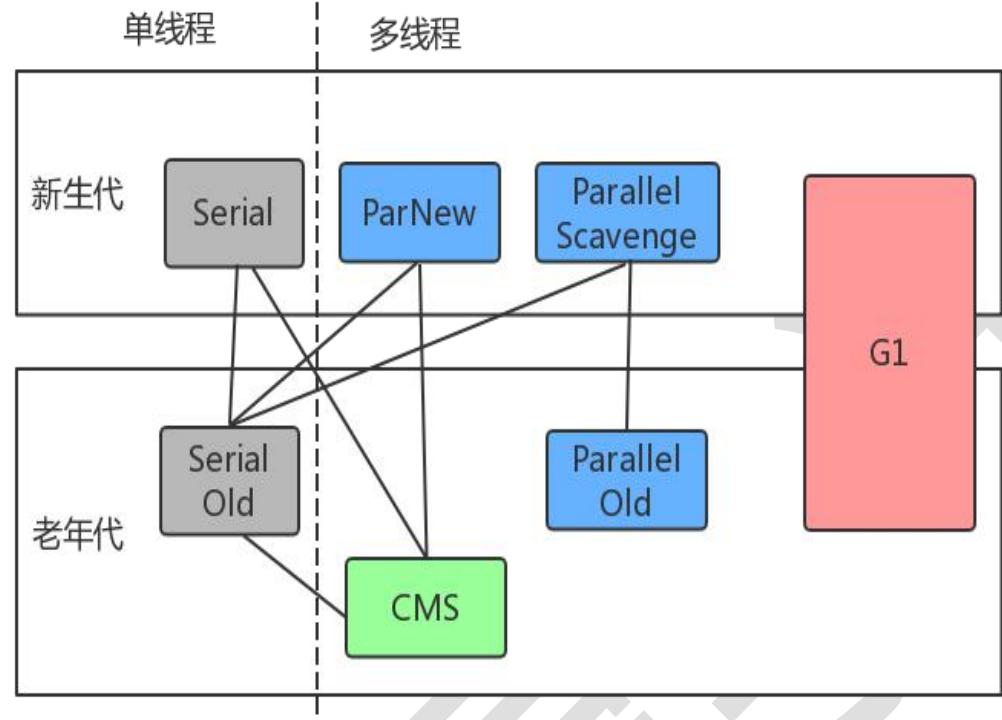
老年代使用标记-整理或者标记-清除算法

jps -v 显示当前使用的垃圾回收器

```
-XX:+UseConcMarkSweepGC  
-Didl: http://www.tunnelin...
```

在新生代中，每次垃圾收集时都发现有大批对象死去，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“标记一清理”或者“标记一整理”算法来进行回收。

请记住下图的垃圾收集器和之间的连线关系。



并行：垃圾收集的多线程的同时进行。

并发：垃圾收集的多线程和应用的多线程同时进行。

注：吞吐量=运行用户代码时间/(运行用户代码时间+ 垃圾收集时间)

垃圾收集时间= 垃圾回收频率 * 单次垃圾回收时间

各种垃圾回收器

Serial/Serial Old

最古老的，单线程，独占式，成熟，适合单 CPU 服务器

-XX:+UseSerialGC 新生代和老年代都用串行收集器

-XX:+UseParNewGC 新生代使用 ParNew，老年代使用 Serial Old

-XX:+UseParallelGC 新生代使用 ParallelGC，老年代使用 Serial Old

ParNew

和 Serial 基本没区别，唯一的区别：多线程，多 CPU 的，停顿时间比 Serial 少

-XX:+UseParNewGC 新生代使用 ParNew，老年代使用 Serial Old

除了性能原因外，主要是因为除了 Serial 收集器，只有它能与 CMS 收集器配合工作。

Parallel Scavenge (ParallelGC) /Parallel Old

关注吞吐量的垃圾收集器，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，主要适合在后台运算而不需要太多交互的任务。

所谓吞吐量就是 CPU 用于运行用户代码的时间与 CPU 总消耗时间的比值，即吞吐量=运行用户代码时间/（运行用户代码时间+垃圾收集时间），虚拟机总共运行了 100 分钟，其中垃圾收集花掉 1 分钟，那有吞吐效率就是 99%。

Concurrent Mark Sweep (CMS)

收集器是一种以获取最短回收停顿时间为为目标的收集器。目前很大一部分的 Java 应用集中在互联网网站或者 B/S 系统的服务端上，这类应用尤其重视服务的响应速度，希望系统停顿时间最短，以给用户带来较好的体验。**CMS** 收集器就非常符合这类应用的需求。

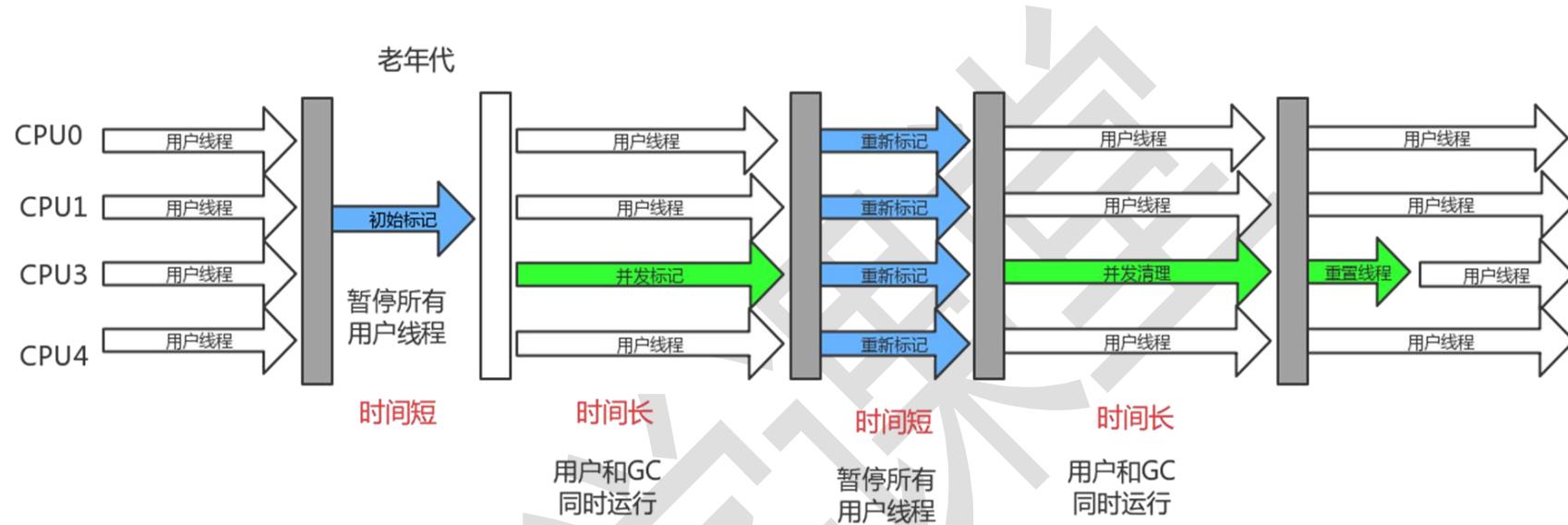
-XX:+UseConcMarkSweepGC，一般新生代使用 ParNew，老年代的用 CMS

从名字（包含“Mark Sweep”）上就可以看出，CMS 收集器是基于“标记—清除”算法实现的，它的运作过程相对于前面几种收集器来说更复杂一些，

垃圾回收过程

整个过程分为 4 个步骤，包括：

- **初始标记：**仅仅只是标记一下 GC Roots 能直接关联到的对象，速度很快，需要停顿（STW -Stop the world）。
- **并发标记：**从 GC Root 开始对堆中对象进行可达性分析，找到存活对象，它在整个回收过程中耗时最长，不需要停顿。
- **重新标记：**为了修正并发标记期间因用户程序继续运作而导致标记产生变动的那一部分对象的标记记录，需要停顿(STW)。这个阶段的停顿时间一般会比初始标记阶段稍长一些，但远比并发标记的时间短。
- **并发清除：**不需要停顿。



优点：

由于整个过程中耗时最长的并发标记和并发清除过程收集器线程都可以与用户线程一起工作，所以，从总体上来说，CMS 收集器的内存回收过程是与用户线程一起并发执行的。

缺点：

CPU 资源敏感：因为并发阶段多线程占据 CPU 资源，如果 CPU 资源不足，效率会明显降低。

浮动垃圾：由于 CMS 并发清理阶段用户线程还在运行着，伴随程序运行自然就还会有新的垃圾不断产生，这一部分垃圾出现在标记过程之后，CMS 无法在当次收集中处理掉它们，只好留待下一次 GC 时再清理掉。这一部分垃圾就称为“浮动垃圾”。

由于浮动垃圾的存在，因此需要预留出一部分内存，意味着 CMS 收集不能像其它收集器那样等待老年代快满的时候再回收。

在 1.6 的版本中老年代空间使用率阈值(92%)

如果预留的内存不够存放浮动垃圾，就会出现 Concurrent Mode Failure，这时虚拟机将临时启用 Serial Old 来替代 CMS。

会产生空间碎片：标记 - 清除算法会导致产生不连续的空间碎片

G1 垃圾回收器

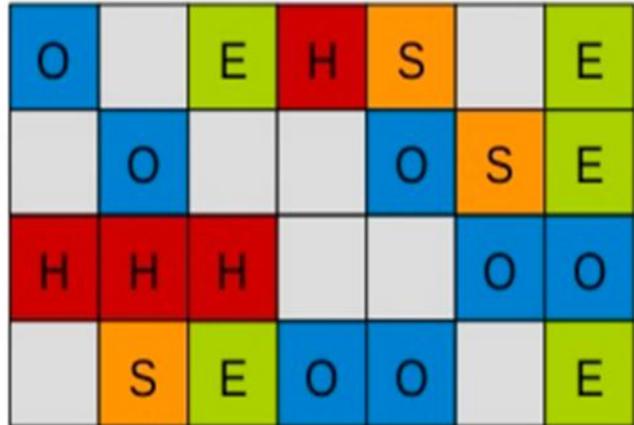
G1 中重要的参数：

-XX:+UseG1GC 使用 G1 垃圾回收器

内部布局改变

G1 把堆划分成多个大小相等的独立区域（Region），新生代和老年代不再物理隔离。

算法：标记一整理（old, humongous）和复制回收算法(survivor)。



GC 模式

Young GC

选定所有新生代里的 Region。通过控制新生代的 region 个数，即新生代内存大小，来控制 young GC 的时间开销。（复制回收算法）

Mixed GC

选定所有新生代里的 Region，外加根据 global concurrent marking 统计得出收集收益高的若干老年代 Region。在用户指定的开销目标范围内尽可能选择收益高的老年代 Region。

Mixed GC 不是 full GC，它只能回收部分老年代的 Region。如果 mixed GC 实在无法跟上程序分配内存的速度，导致老年代填满无法继续进行 Mixed GC，就会使用 serial old GC (full GC) 来收集整个 GC heap。所以我们可以知道，G1 是不提供 full GC 的。

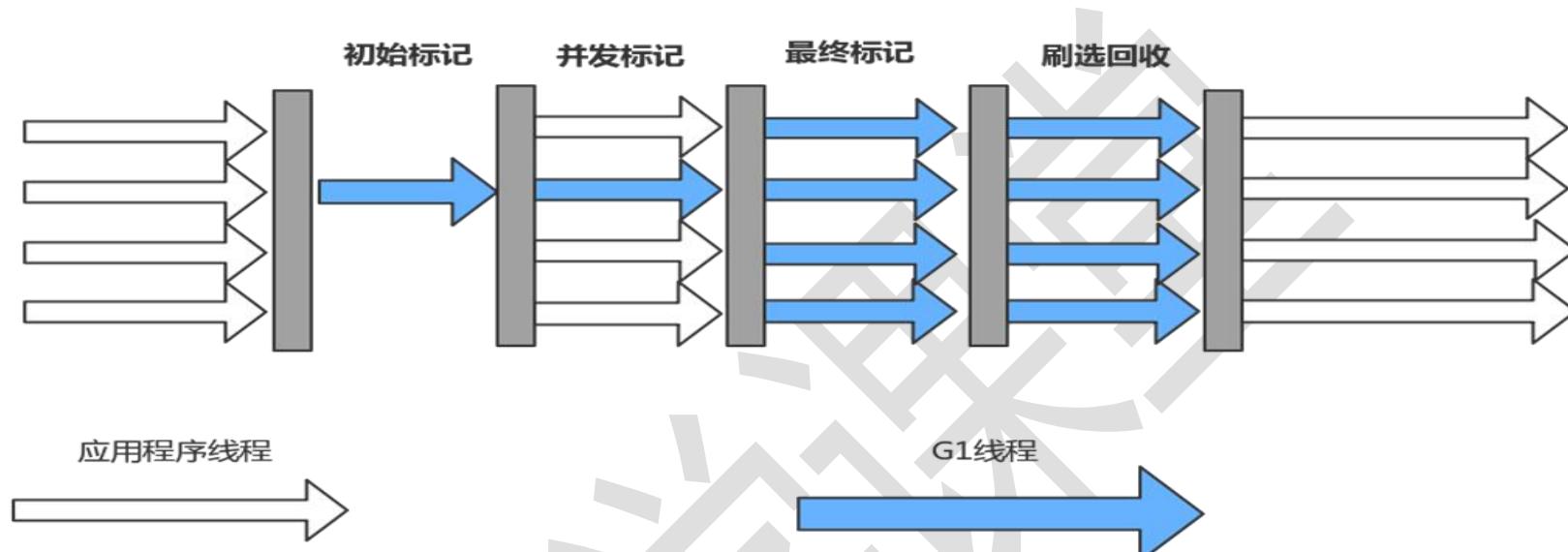
全局并发标记 (global concurrent marking)

初始标记: 仅仅只是标记一下 GC Roots 能直接关联到的对象，并且修改 TAMS (Next Top Mark Start) 的值，让下一阶段用户程序并发运行时，能在正确可以的 Region 中创建对象，此阶段需要停顿线程(STW)，但耗时很短。

并发标记: 从 GC Root 开始对堆中对象进行可达性分析，找到存活对象，此阶段耗时较长，但可与用户程序并发执行。

最终标记: 为了修正在并发标记期间因用户程序继续运作而导致标记产生变动的那一部分标记记录，虚拟机将这段时间对象变化记录在线程的 Remembered Set Logs 里面，最终标记阶段需要把 Remembered Set Logs 的数据合并到 Remembered Set 中。这阶段需要停顿线程(STW)，但是可并行执行。

筛选回收: 首先对各个 Region 中的回收价值和成本进行排序，根据用户所期望的 GC 停顿时间来制定回收计划。此阶段其实也可以做到与用户程序一起并发执行，但是因为只回收一部分 Region，时间是用户可控制的，而且停顿用户线程将大幅度提高收集效率。



特点

空间整合: 不会产生内存碎片

算法: 标记一整理 (humongous) 和复制回收算法(survivor)。

可预测的停顿:

G1 收集器之所以能建立可预测的停顿时间模型，是因为它可以有计划地避免在整个 Java 堆中进行全区域的垃圾收集。G1 跟踪各个 Region 里面的垃圾堆积的价值大小（回收所获得的空间大小以及回收所需时间的经验值），在后台维护一个优先列表，每次根据允许的收集时间，优先回收价值最大的 Region（这也就是 Garbage-First 名称的来由）。这种使用 Region 划分内存空间以及有优先级的区域回收方式，保证了 G1 收集器在有限的时间内可以获取尽可能高的收集效率。

G1 把内存“化整为零”的思路

G1 GC 主要的参数

参数	含义
-XX:G1HeapRegionSize=n	设置 Region 大小，并非最终值
-XX:MaxGCPauseMillis	设置 G1 收集过程目标时间，默认值 200ms，不是硬性条件
-XX:G1NewSizePercent	新生代最小值，默认值 5%
-XX:G1MaxNewSizePercent	新生代最大值，默认值 60%
-XX:ParallelGCThreads	STW 期间，并行 GC 线程数
-XX:ConcGCThreads=n	并发标记阶段，并行执行的线程数
-XX:InitiatingHeapOccupancyPercent	设置触发标记周期的 Java 堆占用率阈值。默认值是 45%。这里的 java 堆占比指的是 non_young_capacity_bytes，包括 old+humongous

垃圾回收器的重要参数（使用-XX:）

参数	描述
UseSerialGC	虚拟机运行在 Client 模式下的默认值，打开此开关后，使用 Serial+Serial Old 的收集器组合进行内存回收
UseParNewGC	打开此开关后，使用 ParNew + Serial Old 的收集器组合进行内存回收
UseConcMarkSweepGC	打开此开关后，使用 ParNew + CMS + Serial Old 的收集器组合进行内存回收。Serial Old 收集器将作为 CMS 收集器出现 Concurrent Mode Failure 失败后的后备收集器使用
UseParallelGC	虚拟机运行在 Server 模式下的默认值，打开此开关后，使用 Parallel Scavenge + Serial Old(PS MarkSweep) 的收集器组合进行内存回收
UseParallelOldGC	打开此开关后，使用 Parallel Scavenge + Parallel Old 的收集器组合进行内存回收
SurvivorRatio	新生代中 Eden 区域与 Survivor 区域的容量比值，默认为 8，代表 Eden : Survivor = 8 : 1
PretenureSizeThreshold	直接晋升到老年代的对象大小，设置这个参数后，大于这个参数的对象将直接在老年代分配
MaxTenuringThreshold	晋升到老年代的对象年龄，每个对象在坚持过一次 Minor GC 之后，年龄就增加 1，当超过这个参数值时就进入老年代
UseAdaptiveSizePolicy	动态调整 Java 堆中各个区域的大小以及进入老年代的年龄
HandlePromotionFailure	是否允许分配担保失败，即老年代的剩余空间不足以应付新生代的整个 Eden 和 Survivor 区的所有对象都存活的极端情况
ParallelGCThreads	设置并行 GC 时进行内存回收的线程数
GCTimeRatio	GC 时间占总时间的比率，默认值为 99，即允许 1% 的 GC 时间，仅在使用 Parallel Scavenge 收集器生效
MaxGCPauseMillis	设置 GC 的最大停顿时间，仅在使用 Parallel Scavenge 收集器时生效
CMSInitiatingOccupancyFraction	设置 CMS 收集器在老年代空间被使用多少后触发垃圾收集，默认值为 68%，仅在使用 CMS 收集器时生效
UseCMSCompactAtFullCollection	设置 CMS 收集器在完成垃圾收集后是否要进行一次内存碎片整理，仅在使用 CMS 收集器时生效
CMSFullGCsBeforeCompaction	设置 CMS 收集器在进行若干次垃圾收集后再启动一次内存碎片整理，仅在使用 CMS 收集器时生效

Stop The World 现象

GC 收集器和我们 GC 调优的目标就是尽可能的减少 STW 的时间和次数。

第四节：JVM 执行子程序

Class 文件结构

计算机只认识 0 和 1，这个称之为本地机器 NativeCode

Jvm 的无关性

与平台无关性是建立在操作系统上，虚拟机厂商提供了许多可以运行在各种不同平台的虚拟机，它们都可以载入和执行字节码，从而实现程序的“一次编写，到处运行” <https://www.oracle.com/technetwork/java/javase/downloads/jdk8-downloads-2133151.html>

各种不同平台的虚拟机与所有平台都统一使用的程序存储格式——字节码（ByteCode）是构成平台无关性的基石，也是语言无关性的基础。Java 虚拟机不和包括 Java 在内的任何语言绑定，它只与“Class 文件”这种特定的二进制文件格式所关联，Class 文件中包含了 Java 虚拟机指令集和符号表以及若干其他辅助信息。

Class 类文件（了解即可）

任何一个 Class 文件都对应着唯一一个类或接口的定义信息，但反过来说，Class 文件实际上它并不一定以磁盘文件的形式存在。

Class 文件是一组以 8 位字节为基础单位的二进制流。

Class 文件格式

各个数据项目严格按照顺序紧凑地排列在 Class 文件之中，中间没有添加任何分隔符，这使得整个 Class 文件中存储的内容几乎全部是程序运行的必要数据，没有空隙存在。

Class 文件格式采用一种类似于 C 语言结构体的伪结构来存储数据，这种伪结构中只有两种数据类型：无符号数和表。

无符号数属于基本的数据类型，以 u1、u2、u4、u8 来分别代表 1 个字节、2 个字节、4 个字节和 8 个字节的无符号数，无符号数可以用来描述数字、索引引用、数量值或者按照 UTF-8 编码构成字符串值。

表是由多个无符号数或者其他表作为数据项构成的复合数据类型，所有表都习惯性地以 “_info” 结尾。表用于描述有层次关系的复合结构的数据，整个 Class 文件本质上就是一张表。

Class 文件格式详解

Class 的结构不像 XML 等描述语言，由于它没有任何分隔符号，所以在其中的数据项，无论是顺序还是数量，都是被严格限定的，哪个字节代表什么含义，长度是多少，先后顺序如何，都不允许改变。

按顺序包括：

魔数与 Class 文件的版本

每个 Class 文件的头 4 个字节称为魔数（Magic Number），它的唯一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。使用魔数而不是扩展名来进行识别主要是基于安全方面的考虑，因为文件扩展名可以随意地改动。文件格式的制定者可以自由地选择魔数值，只要这个魔数值还没有被广泛

采用过同时又不会引起混淆即可。（）

紧接着魔数的 4 个字节存储的是 Class 文件的版本号：第 5 和第 6 个字节是次版本号（MinorVersion），第 7 和第 8 个字节是主版本号（Major Version）。Java 的版本号是从 45 开始的，JDK 1.1 之后的每个 JDK 大版本发布主版本号向上加 1 高版本的 JDK 能向下兼容以前版本的 Class 文件，但不能运行以后版

本的 Class 文件，即使文件格式并未发生任何变化，虚拟机也必须拒绝执行超过其版本号的 Class 文件。

4	5	6	7
00	00	00	34

代表 JDK1.8

常量池

常量池中常量的数量是不固定的，所以在常量池的入口需要放置一项 u2 类型的数据，代表常量池容量计数值（constant_pool_count）。与 Java 中语言习惯不一样的是，这个容量计数是从 1 而不是 0 开始的

常量池中主要存放两大类常量：字面量（Literal）和符号引用（Symbolic References）。

字面量比较接近于 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。

而符号引用则属于编译原理方面的概念，包括了下面三类常量：

类和接口的全限定名（Fully Qualified Name）、字段的名称和描述符（Descriptor）、方法的名称和描述符

访问标志

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；是否定义为 public 类型；是否定义为 abstract 类型；如果是类的话，是否被声明为 final 等

类索引、父类索引与接口索引集合

这三项数据来确定这个类的继承关系。类索引用于确定这个类的全限定名，父类索引用于确定这个类的父类的全限定名。由于 Java 语言不允许多重继承，所以父类索引只有一个，除了 java.lang.Object 之外，所有的 Java 类都有父类，因此除了 java.lang.Object 外，所有 Java 类的父类索引都不为 0。接口索引集合就用来描述这个类实现了哪些接口，这些被实现的接口将按 implements 语句（如果这个类本身是一个接口，则应当是 extends 语句）后的接口顺序从左到右排列在接口索引集合中

字段表集合

描述接口或者类中声明的变量。字段（field）包括类级变量以及实例级变量。

而字段叫什么名字、字段被定义为什么数据类型，这些都是无法固定的，只能引用常量池中的常量来描述。

字段表集合中不会列出从超类或者父接口中继承而来的字段，但有可能列出原本 Java 代码之中不存在的字段，譬如在内部类中为了保持对外部类的访问性，会自动添加指向外部类实例的字段。

方法表集合

描述了方法的定义，但是方法里的 Java 代码，经过编译器编译成字节码指令后，存放在属性表集合中的方法属性表集合中一个名为“Code”的属性里面。与字段表集合相类似的，如果父类方法在子类中没有被重写（Override），方法表集合中就不会出现来自父类的方法信息。但同样的，有可能会出现由编译器自动添加的方法，最典型的便是类构造器“<clinit>”方法和实例构造器“<init>”

属性表集合

存储 Class 文件、字段表、方法表都自己的属性表集合，以用于描述某些场景专有的信息。如方法的代码就存储在 Code 属性表中。

字节码指令

Java 虚拟机的指令由一个字节长度的、代表着某种特定操作含义的数字（称为操作码，Opcode）以及跟随其后的零至多个代表此操作所需参数（称为操作数，Operands）而构成。

由于限制了 Java 虚拟机操作码的长度为一个字节（即 0~255），这意味着指令集的操作码总数不可能超过 256 条。

大多数的指令都包含了其操作所对应的数据类型信息。例如：

`iload` 指令用于从局部变量表中加载 `int` 型的数据到操作数栈中，而 `fload` 指令加载的则是 `float` 型的数据。

大部分的指令都没有支持整数类型 `byte`、`char` 和 `short`，甚至没有任何指令支持 `boolean` 类型。大多数对于 `boolean`、`byte`、`short` 和 `char` 类型数据的操作，实际上都是使用相应的 `int` 型作为运算类型。

阅读字节码作为了解 Java 虚拟机的基础技能，有需要的话可以去掌握常见指令。

加载和存储指令

用于将数据在栈帧中的局部变量表和操作数栈之间来回传输，这类指令包括如下内容。

将一个局部变量加载到操作栈：`iload`、`iload_<n>`、`lload`、`lload_<n>`、`fload`、`fload_<n>`、`dload`、`dload_<n>`、`aload`、`aload_<n>`。

将一个数值从操作数栈存储到局部变量表：`istore`、`istore_<n>`、`lstore`、`lstore_<n>`、`fstore`、`fstore_<n>`、`dstore`、`dstore_<n>`、`astore`、`astore_<n>`。

将一个常量加载到操作数栈：`bipush`、`sipush`、`ldc`、`ldc_w`、`ldc2_w`、`aconst_null`、`iconst_m1`、`iconst_<i>`、`lconst_<I>`、`fconst_<f>`、`dconst_<d>`。

扩充局部变量表的访问索引的指令：`wide`。

运算或算术指令

用于对两个操作数栈上的值进行某种特定运算，并把结果重新存入到操作栈顶。

加法指令：`iadd`、`ladd`、`fadd`、`dadd`。

减法指令：`isub`、`lsub`、`fsub`、`dsub`。

乘法指令：`imul`、`lmul`、`fmul`、`dmul` 等等

类型转换指令

可以将两种不同的数值类型进行相互转换，

Java 虚拟机直接支持以下数值类型的宽化类型转换（即小范围类型向大范围类型的安全转换）：

int 类型到 long、float 或者 double 类型。

long 类型到 float、double 类型。

float 类型到 double 类型。

处理窄化类型转换（Narrowing Numeric Conversions）时，必须显式地使用转换指令来完成，这些转换指令包括：i2b、i2c、i2s、l2i、f2i、f2l、d2i、d2l 和 d2f。

创建类实例的指令

new。

创建数组的指令

newarray、anewarray、multianewarray。

访问字段指令

getfield、putfield、getstatic、putstatic。

数组存取相关指令

把一个数组元素加载到操作数栈的指令：baload、caload、saload、iaload、laload、faload、daload、aaload。

将一个操作数栈的值存储到数组元素中的指令：bastore、castore、sastore、iastore、fastore、dastore、aastore。

取数组长度的指令：arraylength。

检查类实例类型的指令

instanceof、checkcast。

操作数栈管理指令

如同操作一个普通数据结构中的堆栈那样，Java 虚拟机提供了一些用于直接操作操作数栈的指令，包括：将操作数栈的栈顶一个或两个元素出栈：pop、pop2。

复制栈顶一个或两个数值并将复制值或双份的复制值重新压入栈顶：dup、dup2、dup_x1、dup2_x1、dup_x2、dup2_x2。

将栈最顶端的两个数值互换：swap。

控制转移指令

控制转移指令可以让 Java 虚拟机有条件或无条件地从指定的位置指令而不是控制转移指令的下一条指令继续执行程序，从概念模型上理解，可以认为控制转移指令就是在有条件或无条件地修改 PC 寄存器的值。控制转移指令如下。

条件分支：ifeq、iflt、ifle、ifne、ifgt、ifge、ifnull、ifnonnull、if_icmpne、if_icmplt、if_icmpgt、if_icmple、if_icmpge、if_acmpeq 和 if_acmpne。

复合条件分支：tableswitch、lookupswitch。

无条件分支：goto、goto_w、jsr、jsr_w、ret。

方法调用指令

invokevirtual 指令用于调用对象的实例方法，根据对象的实际类型进行分派（虚方法分派），这也是 Java 语言中最常见的方法分派方式。

`invokeinterface` 指令用于调用接口方法，它会在运行时搜索一个实现了这个接口方法的对象，找出适合的方法进行调用。

`invokespecial` 指令用于调用一些需要特殊处理的实例方法，包括实例初始化方法、私有方法和父类方法。

`invokestatic` 指令用于调用类方法（`static` 方法）。

`invokedynamic` 指令用于在运行时动态解析出调用点限定符所引用的方法，并执行该方法，前面 4 条调用指令的分派逻辑都固化在 Java 虚拟机内部，而 `invokedynamic` 指令的分派逻辑是由用户所设定的引导方法决定的。

方法调用指令与数据类型无关。

方法返回指令

是根据返回值的类型区分的，包括 `ireturn`（当返回值是 `boolean`、`byte`、`char`、`short` 和 `int` 类型时使用）、`lreturn`、`freturn`、`dreturn` 和 `areturn`，另外还有一条 `return` 指令供声明为 `void` 的方法、实例初始化方法以及类和接口的类初始化方法使用。

异常处理指令

在 Java 程序中显式抛出异常的操作（`throw` 语句）都由 `athrow` 指令来实现

同步指令

有 `monitorenter` 和 `monitorexit` 两条指令来支持 `synchronized` 关键字的语义

虚拟机栈再认识

整体介绍见 [运行时数据区域](#)，虚拟机栈简单介绍见 [虚拟机栈（JVM 后续的执行子程序有详细的见解）](#)

栈帧中的数据在编译后就已经确定了，写在了字节码文件的 code 属性中（属性表集合）

栈桢详解

当前栈桢有效：一个线程的方法调用链可能会很长，这意味着虚拟机栈会被压入很多栈桢，但在线程执行的某个时间点只有位于栈顶的栈桢才是有效的，该栈桢称为“当前栈桢”，与这个栈桢相关联的方法称为“当前方法”。

局部变量表

局部变量表的容量以变量槽（Variable Slot，下称 Slot）为最小单位，虚拟机规范中导向性地说到每个 Slot 都应该能存放一个 boolean、byte、char、short、int、float、double、long 8 种数据类型和引用 reference，可以使用 32 位或更小的物理内存来存放。

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。Java 语言中明确的（reference 类型则可能是 32 位也可能是 64 位）64 位的数据类型只有 long 和 double 两种。

操作数栈

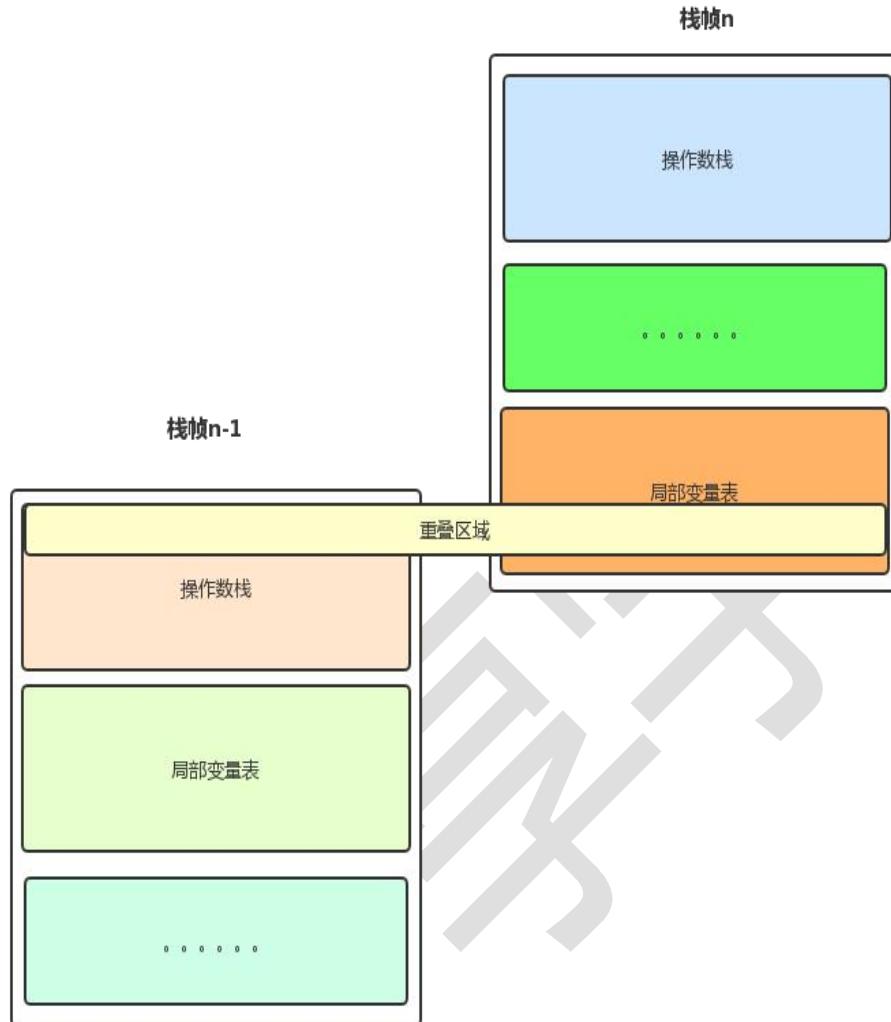
操作数栈（Operand Stack）也常称为操作栈，它是一个先进后出（First In Last Out,FILO）栈。同局部变量表一样，操作数栈的每一个元素可以是任意的 Java 数据类型，包括 float 和 double。32 位数据类型所占的栈容量为 1，64 位数据类型所占的栈容量为 2。

当一个方法刚刚开始执行的时候，这个方法的操作数栈是空的，在方法的执行过程中，会有各种字节码指令往操作数栈中写入和提取内容，也就是出栈/入栈操作。例如，在做算术运算的时候是通过操作数栈来进行的，又或者在“调用其他方法的时候是通过操作数栈来进行参数传递的”。

java 虚拟机的解释执行引擎称为“基于栈的执行引擎”，其中所指的“栈”就是操作数栈。

数据重叠优化

虚拟机概念模型中每两个栈帧都是相互独立的，但在实际应用是我们知道一个方法调用另一个方法时，往往存在参数传递，这种做法在虚拟机实现过程中会做一些优化，具体做法如下：令两个栈帧出现一部分重叠。让下面栈帧的一部分操作数栈与上面栈帧的部分局部变量表重叠在一起，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。



时时就

动态连接

既然是执行方法，那么我们需要知道当前栈帧执行的是哪个方法，栈帧中会持有一个引用（符号引用），该引用指向某个具体方法。

符号引用是一个地址位置的代号，在编译的时候我们是不知道某个方法在运行的时候是放到哪里的，这时我用代号 com/enjoy/pojo/User.Say():V 指代某个类的方法，将来可以把符号引用转换成直接引用进行真实的调用。

用符号引用转化成直接引用的解析时机，把解析分为两大类

静态解析：符号引用在类加载阶段或者第一次使用的时候就直接转换成直接引用。

动态连接：符号引用在每次运行期间转换为直接引用，即每次运行都重新转换。

方法返回地址

方法退出方式有：正常退出与异常退出

理论上，执行完当前栈帧的方法，需要返回到当前方法被调用的位置，所以栈帧需要记录一些信息，用来恢复上层方法的执行状态。正常退出，上层方法的 PC 计数器可以做为当前方法的返回地址，被保存在当前栈帧中。

异常退出时，返回地址是要通过异常处理器表来确定的，栈帧中一般不会保存这部分信息

方法退出时会做的操作：恢复上次方法的局部变量表、操作数栈，把当前方法的返回值，压入调用者栈帧的操作数栈中，使用当前栈帧保存的返回地址调整 PC 计数器的值，当前栈帧出栈，随后，执行 PC 计数器指向的指令。

附加信息

虚拟机规范允许实现虚拟机时增加一些额外信息，例如与调试相关的信息。

一般把 动态连接、方法返回地址、其他额外信息归成一类，称为栈帧信息。

基于栈的字节码解释执行引擎

Java 编译器输出的指令流，基本上是一种基于栈的指令集架构，指令流中的指令大部分都是零地址指令，它们依赖操作数栈进行工作。与基于寄存器的指令集，最典型的就是 x86 的二地址指令集，说得通俗一些，就是现在我们主流 PC 机中直接支持的指令集架构，这些指令依赖寄存器进行工作。

基于栈的指令集

举个最简单的例子，分别使用这两种指令集计算“1+1”的结果，基于栈的指令集会是这样子的：

```
iconst_1  
iconst_1  
iadd  
istore_0
```

两条 `iconst_1` 指令连续把两个常量 1 压入栈后，`iadd` 指令把栈顶的两个值出栈、相加，然后把结果放回栈顶，最后 `istore_0` 把栈顶的值放到局部变量表的第 0 个 Slot 中。

基于寄存器的指令集

如果基于寄存器，那程序可能会是这个样子：

```
mov eax, 1  
add eax, 1  
mov
```

`mov` 指令把 `EAX` 寄存器的值设为 1，然后 `add` 指令再把这个值加 1，结果就保存在 `EAX` 寄存器里面。

基于栈的指令集主要的优点就是可移植，寄存器由硬件直接提供，程序直接依赖这些硬件寄存器则不可避免地要受到硬件的约束。栈架构指令集的主要

缺点是执行速度相对来说会稍慢一些。所有主流物理机的指令集都是寄存器架构也从侧面印证了这一点。

```
LineNumberTable:  
  line 3: 0  
LocalVariableTable:  
  Start  Length  Slot  Name   Signature  
    0      5     0  this  Lcom/xiangxue/ch03>ShowByteCode;  
  
public int calc():  
  descriptor: ()I  
  flags: ACC_PUBLIC  
Code:  
  stack=2, locals=4, args_size=1  
  0: bipush   100  
  2: istore_1  
  3: sipush   200  
  6: istore_2  
  7: sipush   300  
 10: istore_3  
 11: iload_1  
 12: iload_2  
 13: iadd  
 14: iload_3  
 15: imul  
 16: ireturn  
LineNumberTable:  
  line 10: 0  
  line 11: 3  
  line 12: 7  
  line 13: 11  
LocalVariableTable:  
  Start  Length  Slot  Name   Signature  
    0      17    0  this  Lcom/xiangxue/ch03>ShowByteCode;  
    3      14    1    a    I  
    7      10    2    b    I  
   11      6    3    c    I  
}  
SourceFile: "ShowByteCode.java"  
  
D:\XiangXue\vipLesson\JVM\code\src\vip-jvm\bin\com\xiangxue\ch03>
```

方法调用详解

解析

调用目标在程序代码写好、编译器进行编译时就必须确定下来。这类方法的调用称为解析。

在 Java 语言中符合“编译期可知，运行期不可变”这个要求的方法，主要包括静态方法和私有方法两大类，前者与类型直接关联，后者在外部不可被访问，这两种方法各自的特点决定了它们都不可能通过继承或别的方式重写其他版本，因此它们都适合在类加载阶段进行解析。

见实例代码（`dispatch` 包）

静态分派

多见于方法的重载。

```
iles  
11  static abstract class Human{}  
12  static class Man extends Human{ }  
13  static class Woman extends Human{}  
14  
15  public void sayHello(Human guy){  
16      System.out.println("hello,guy!"); //1  
17  }  
18  public void sayHello(Man guy){  
19      System.out.println("hello,gentleman!"); //2  
20  }  
21  public void sayHello(Woman guy){  
22      System.out.println("hello,lady!"); //3  
23  }  
24  
25  public static void main(String[] args){  
26      静态类型  
27      Human h1 = new Man();  
28      Human h2 = new Woman();  
29  
30      StaticDispatch sr = new StaticDispatch();  
31      sr.sayHello(h1);  
32      sr.sayHello(h2);  
33  
34
```

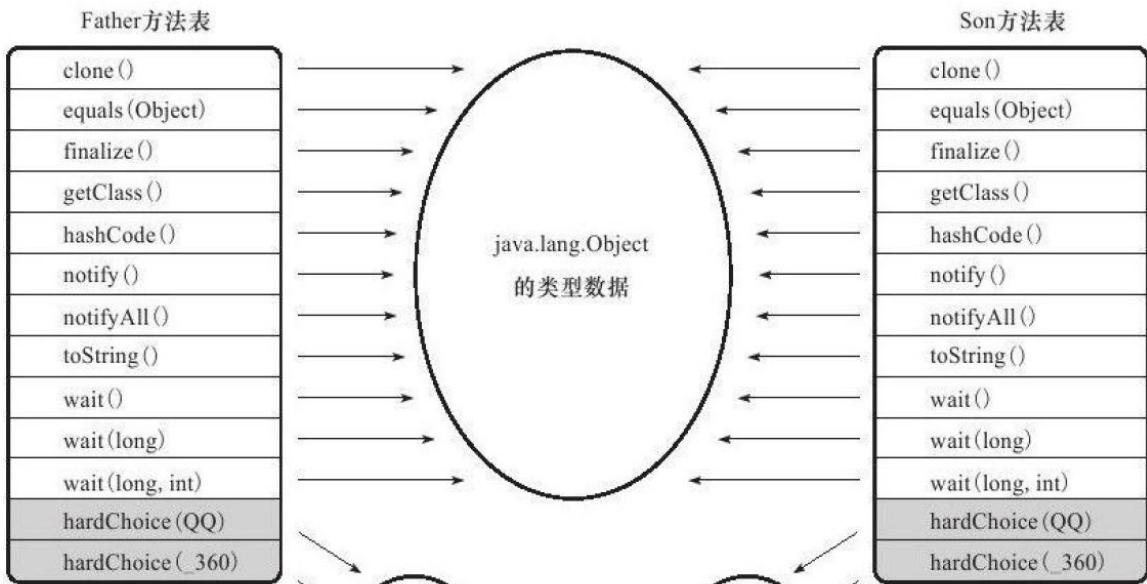
“Human”称为变量的静态类型（Static Type），或者叫做的外观类型（Apparent Type），后面的“Man”则称为变量的实际类型（Actual Type），静态类型和实际类型在程序中都可以发生一些变化，区别是静态类型的变化仅仅在使用时发生，变量本身的静态类型不会被改变，并且最终的静态类型是在编译期可知的；而实际类型变化的结果在运行期才可确定，编译器在编译程序的时候并不知道一个对象的实际类型是什么。

代码中定义了两个静态类型相同但实际类型不同的变量，但虚拟机（准确地说是编译器）在重载时是通过参数的静态类型而不是实际类型作为判定依据的。并且静态类型是编译期可知的，因此，在编译阶段，Javac 编译器会根据参数的静态类型决定使用哪个重载版本，所以选择了 sayHello (Human) 作为调用目标。所有依赖静态类型来定位方法执行版本的分派动作称为静态分派。静态分派的典型应用是方法重载。静态分派发生在编译阶段，因此确定静态分派的动作实际上不是由虚拟机来执行的。

动态分派

静态类型同样都是 Human 的两个变量 man 和 woman 在调用 sayHello () 方法时执行了不同的行为，并且变量 man 在两次调用中执行了不同的方法。导致这个现象的原因很明显，是这两个变量的实际类型不同。

在实现上，最常用的手段就是为类在方法区中建立一个虚方法表。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。PPT 图中，Son 重写了来自 Father 的全部方法，因此 Son 的方法表没有指向 Father 类型数据的箭头。但是 Son 和 Father 都没有重写来自 Object 的方法，所以它们的方法表中所有从 Object 继承来的方法都指向了 Object 的数据类型。



类加载机制

概述

类从被加载到虚拟机内存中开始，到卸载出内存为止，它的整个生命周期包括：加载（Loading）、验证（Verification）、准备（Preparation）、解析（Resolution）、初始化（Initialization）、使用（Using）和卸载（Unloading）7个阶段。其中验证、准备、解析3个部分统称为连接（Linking）

初始化

初始化的 5 种情况

初始化阶段，虚拟机规范则是严格规定了有且只有 5 种情况必须立即对类进行“初始化”（而加载、验证、准备自然需要在此之前开始）：

- 1) 遇到 `new`、`getstatic`、`putstatic` 或 `invokestatic` 这 4 条字节码指令时，如果类没有进行过初始化，则需要先触发其初始化。生成这 4 条指令的最常见的 Java 代码场景是：使用 `new` 关键字实例化对象的时候、读取或设置一个类的静态字段（被 `final` 修饰、已在编译期把结果放入常量池的静态字段除外）的时候，以及调用一个类的静态方法的时候。
- 2) 使用 `java.lang.reflect` 包的方法对类进行反射调用的时候，如果类没有进行过初始化，则需要先触发其初始化。
- 3) 当初始化一个类的时候，如果发现其父类还没有进行过初始化，则需要先触发其父类的初始化。
- 4) 当虚拟机启动时，用户需要指定一个要执行的主类（包含 `main()` 方法的那个类），虚拟机会先初始化这个主类。
- 5) 当使用 JDK 1.7 的动态语言支持时，如果一个 `java.lang.invoke.MethodHandle` 实例最后的解析结果 `REF_getStatic`、`REF_putStatic`、`REF_invokeStatic` 的方法句柄，并且这个方法句柄所对应的类没有进行过初始化，则需要先触发其初始化。

举例 (clazzload 包中例子)

1.对于静态字段，只有直接定义这个字段的类才会被初始化，因此通过其子类来引用父类中定义的静态字段，只会触发父类的初始化而不会触发子类的初始化，如下图：

The screenshot shows a Java project structure and a terminal window. The project tree on the left includes a package named 'clazzload' containing classes 'NotInitialization', 'SubClaszz', and 'SuperClazz'. Inside 'clazzload', there is a folder 'deencrypt' containing a file 'CustomClassLoader'. The terminal window on the right shows the execution of the 'NotInitialization' class. The code for 'NotInitialization' is:

```
5  */  
6  public class NotInitialization {  
7  public static void main(String[] args){  
8      System.out.println(SubClaszz.value);  
9  }
```

The terminal output shows:

```
"C:\Program Files\Java\jdk1.8.0_191\bin\java.exe" ...  
SuperClass init!  
123  
Process finished with exit code 0
```

A red arrow points to the line "SuperClass init!" in the terminal output, and another red arrow points to the line "System.out.println(SubClaszz.value);".

2.数组形式的 new(而不是构造方法)不会触发类初始化

```
5  */  
6 > public class NotInitialization {  
7 >     public static void main(String[] args){  
8 >         SuperClazz[] sca = new SuperClazz[10];  
9 >     }  
10 }  
NotInitialization > main()
```

n: NotInitialization x
"C:\Program Files\Java\jdk1.8.0_191\bin\java.exe" ...
Process finished with exit code 0

3. 直接打印类的常量会不会触发类的初始化：（坑：项目中有可能常量改了，关联使用的类不重新编译就会还是原来的值）

常量 HELLOWORLD，但其实在编译阶段通过常量传播优化，已经将此常量的值“hello world”存储到了 NotInitialization 类的常量池中，以后 NotInitialization 对常量 ConstClass.HELLOWORLD 的引用实际都被转化为 NotInitialization 类对自身常量池的引用了。

也就是说，实际上 NotInitialization 的 Class 文件之中并没有 ConstClass 类的符号引用入口，这两个类在编译成 Class 之后就不存在任何联系了。

```
5  */  
6 > public class NotInitialization {  
7 >     public static void main(String[] args){  
8 >         System.out.println(SuperClazz.HELLOWORLD);  
9 >     }  
10 }  
NotInitialization > main()
```

n: NotInitialization x
"C:\Program Files\Java\jdk1.8.0_191\bin\java.exe" ...
hello world

没有构造方法打印

4. 如果使用常量去引用另外一个常量，这个时候编译阶段无法进行优化，所以才会触发类的初始化。

The screenshot shows a Java IDE interface with the following details:

- Code Editor:** Displays a class definition with static fields and a main method.

```
    }
    public static int value=123;
    public static final String HELLOWORLD="hello world";
    public static final int WHAT = value;
```

```
public static void main(String[]args){
    System.out.println(SuperClazz.WHAT);
```
- Call Stack:** Shows the current stack frame for the main method.
- Registers:** Shows the stack pointer (SP) pointing to the instruction `System.out.println(SuperClazz.WHAT);`.
- Registers:** Shows the stack pointer (SP) pointing to the instruction `System.out.println(SuperClazz.WHAT);`.
- Output:** Shows the command "C:\Program Files\Java\jdk1.8.0_191\bin\java.exe" ... and the output "SuperClass init!" followed by "构造方法".
- Memory Dump:** Shows the value 123 at the address `0x0000000000400000`.

Annotations with red arrows highlight the static field `WHAT` in the code editor, the stack frame for `main()`, the output "构造方法", and the memory dump value "123".

加载阶段

虚拟机需要完成以下 3 件事情：

- 1) 通过一个类的全限定名来获取定义此类的二进制字节流。
- 2) 将这个字节流所代表的静态存储结构转化为方法区的运行时数据结构。
- 3) 在内存中生成一个代表这个类的 `java.lang.Class` 对象，作为方法区这个类的各种数据的访问入口。

验证

是连接阶段的第一步，这一阶段的目的是为了确保 Class 文件的字节流中包含的信息符合当前虚拟机的要求，并且不会危害虚拟机自身的安全。但从整体上看，验证阶段大致上会完成下面 4 个阶段的检验动作：文件格式验证、元数据验证、字节码验证、符号引用验证。

准备阶段

是正式为类变量分配内存并设置类变量初始值的阶段，这些变量所使用的内存都将在方法区中进行分配。这个阶段中有两个容易产生混淆的概念需要强调一下，首先，这时候进行内存分配的仅包括类变量（被 `static` 修饰的变量），而不包括实例变量，实例变量将会在对象实例化时随着对象一起分配在 Java 堆中。其次，这里所说的初始值“通常情况”下是数据类型的零值，假设一个类变量的定义为：

```
public static int value=123;
```

那变量 `value` 在准备阶段过后的初始值为 `0` 而不是 `123`，因为这时候尚未开始执行任何 Java 方法，而把 `value` 赋值为 `123` 的 `putstatic` 指令是程序被编译后，存放于类构造器`<clinit>()`方法之中，所以把 `value` 赋值为 `123` 的动作将在初始化阶段才会执行。假设上面类变量 `value` 的定义变为：`public static final int value=123;`

编译时 Javac 将会为 `value` 生成 `ConstantValue` 属性，在准备阶段虚拟机就会根据 `ConstantValue` 的设置将 `value` 赋值为 `123`。

解析阶段

是虚拟机将常量池内的符号引用替换为直接引用的过程。部分详细内容见[解析](#)

类初始化阶段

是类加载过程的最后一步，前面的类加载过程中，除了在加载阶段用户应用程序可以通过自定义类加载器参与之外，其余动作完全由虚拟机主导和控制。到了初始化阶段，才真正开始执行类中定义的 Java 程序代码在准备阶段，变量已经赋过一次系统要求的初始值，而在初始化阶段，则根据程序员通过程序制定的主观计划去初始化类变量和其他资源，或者可以从另外一个角度来表达：初始化阶段是执行类构造器`<clinit>()`方法的过程。`<clinit>()`方法是由编译器自动收集类中的所有类变量的赋值动作和静态语句块（`static{}块`）中的语句合并产生的，编译器收集的顺序是由语句在源文件中出现的顺序所决定的。

`<clinit>()`方法对于类或接口来说并不是必需的，如果一个类中没有静态语句块，也没有对变量的赋值操作，那么编译器可以不为这个类生成`<clinit>()`方法。

初始化的单例模式（线程安全）

虚拟机会保证一个类的<clinit>（）方法在多线程环境中被正确地加锁、同步，如果多个线程同时去初始化一个类，那么只会有一个线程去执行这个类的<clinit>（）方法，其他线程都需要阻塞等待，直到活动线程执行<clinit>（）方法完毕。如果在一个类的<clinit>（）方法中有耗时很长的操作，就可能造成多个进程阻塞。所以类的初始化是线程安全的，项目中可以利用这点。

类加载器

对于任意一个类，都需要由加载它的类加载器和这个类本身一同确立其在 Java 虚拟机中的唯一性，每一个类加载器，都拥有一个独立的类名称空间。这句话可以表达得更通俗一些：比较两个类是否“相等”，只有在这两个类是由同一个类加载器加载的前提下才有意义，否则，即使这两个类来源于同一个 Class 文件，被同一个虚拟机加载，只要加载它们的类加载器不同，那这两个类就必定不相等。

这里所指的“相等”，包括代表类的 Class 对象的 equals（）方法、isAssignableFrom（）方法、isInstance（）方法的返回结果，也包括使用 instanceof 关键字做对象所属关系判定等情况。

加解密案例(deencrypt 代码)

通过位的二进制异或运算进行加解密（一次就是加密，再运算一次就是解密）

- 1.DemoUser.class 重命名为 DemoUserSrc.class 同时删掉 DemoUser.class，再通过 XorEncrypt 加密生成 DemoUser.class, 使用编辑工具查看下加密前和加密后
- 2.写一个自定义的类加载器，继承 ClassLoader，同时在加载时进行解密。
- 3.写一个 DemoRun 类，使用自定义的类加载器加密，再打印类的对象，看它是哪个类加载器加载的，是否能正常显示。

加解密的项目中运用：可以使用把代码使用私钥加密，在解析阶段使用公钥解密。这样跟用户做项目时提供对应的公钥，自己提供私钥加密后的代码信息。在类加载时使用公钥解密运行。这样可以确保源代码的保密性。

双亲委派模型

对于任意一个类，都需要由加载它的类加载器和这个类本身一同确立其在 Java 虚拟机中的唯一性。

从 Java 虚拟机的角度来讲，只存在两种不同的类加载器：

一种是启动类加载器（Bootstrap ClassLoader），这个类加载器使用 C++ 语言实现，是虚拟机自身的一部分；另一种就是所有其他的类加载器，这些类加载器都由 Java 语言实现，独立于虚拟机外部，并且全都继承自抽象类 `java.lang.ClassLoader`。

启动类加载器（Bootstrap ClassLoader）：这个类将负责将存放在 `<JAVA_HOME>\lib` 目录中的，或者被 `-Xbootclasspath` 参数所指定的路径中的，并且是虚拟机识别的（仅按照文件名识别，如 `rt.jar`，名字不符合的类库即使放在 `lib` 目录中也不会被加载）类库加载到虚拟机内存中。启动类加载器无法被 Java 程序直接引用，用户在编写自定义类加载器时，如果需要把加载请求委派给引导类加载器，那直接使用 `null` 代替即可。

扩展类加载器（Extension ClassLoader）：这个加载器由 `sun.misc.Launcher$ExtClassLoader` 实现，它负责加载 `<JAVA_HOME>\lib\ext` 目录中的，或者被 `java.ext.dirs` 系统变量所指定的路径中的所有类库，开发者可以直接使用扩展类加载器。

应用程序类加载器（Application ClassLoader）：这个类加载器由 `sun.misc.Launcher $App-ClassLoader` 实现。由于这个类加载器是 `ClassLoader` 中的 `getSystemClassLoader()` 方法的返回值，所以一般也称它为系统类加载器。它负责加载用户类路径（`ClassPath`）上所指定的类库，开发者可以直接使用这个类加载器，如果应用程序中没有自定义过自己的类加载器，一般情况下这个就是程序中默认的类加载器。

我们的应用程序都是由这 3 种类加载器互相配合进行加载的，如果有必要，还可以加入自己定义的类加载器。

双亲委派模型要求除了顶层的启动类加载器外，其余的类加载器都应当有自己的父类加载器。这里类加载器之间的父子关系一般不会以继承（Inheritance）的关系来实现，而是都使用组合（Composition）关系来复用父加载器的代码。

使用双亲委派模型来组织类加载器之间的关系，有一个显而易见的好处就是 Java 类随着它的类加载器一起具备了一种带有优先级的层次关系。例如类 `java.lang.Object`，它存放在 `rt.jar` 之中，无论哪一个类加载器要加载这个类，最终都是委派给处于模型最顶端的启动类加载器进行加载，因此 `Object` 类在程序的各种类加载器环境中都是同一个类。相反，如果没有使用双亲委派模型，由各个类加载器自行去加载的话，如果用户自己编写了一个称为 `java.lang.Object` 的类，并放在程序的 `ClassPath` 中，那系统中将会出现多个不同的 `Object` 类，Java 类型体系中最基础的行为也就无法保证，应用程序也将变得一片混乱。

应用程序类加载器

ClassLoader 中的 loadClass 方法中的代码逻辑就是双亲委派模型：

在自定义 ClassLoader 的子类时候，我们常见的会有两种做法，一种是重写 **loadClass** 方法，另一种是重写 **findClass** 方法。其实这两种方法本质上差不多，毕竟 loadClass 也会调用 findClass，但是从逻辑上讲我们最好不要直接修改 loadClass 的内部逻辑。我建议的做法是只在 findClass 里重写自定义类的加载方法。

loadClass 这个方法是实现双亲委托模型逻辑的地方，擅自修改这个方法会导致模型被破坏，容易造成问题。因此我们最好是在双亲委托模型框架内进行小范围的改动，不破坏原有的稳定结构。同时，也避免了自己重写 loadClass 方法的过程中必须写双亲委托的重复代码，从代码的复用性来看，不直接修改这个方法始终是比较好的选择。

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。

Tomcat 类加载源码分析

WebappClassLoader 中 loadClass 方法，源码具体 Tomcat 章节细讲

第五节：JVM 性能优化（上）

内存溢出

内存溢出的原因：程序在申请内存时，没有足够的内存空间

栈溢出

方法死循环递归调用（StackOverflowError）、不断建立线程（OutOfMemoryError）

堆溢出

不断创建对象，分配对象大于最大堆的大小（OutOfMemoryError）

直接内存

JVM 分配的本地直接内存大小大于 JVM 的限制(可以通过-XX:MaxDirectMemorySize 来设置（不设置的话默认与堆内存最大值一样,也会出现

OOM 异常)

方法区溢出

在经常动态生产大量 Class 的应用中, CGLIB 字节码增强, 动态语言, 大量 JSP(JSP 第一次运行需要编译成 Java 类), 基于 OSGi 的应用(同一个类, 被不同的加载器加载也会设为不同的类)

内存泄漏

程序在申请内存后, 无法释放已申请的内存空间。

长生命周期的对象持有短生命周期对象的引用

例如将 ArrayList 设置为静态变量, 则容器中的对象在程序结束之前将不能被释放, 从而造成内存泄漏

连接未关闭

如数据库连接、网络连接和 IO 连接等, 只有连接被关闭后, 垃圾回收器才会回收对应的对象。

变量作用域不合理

例如, 1.一个变量的定义的作用范围大于其使用范围, 2.如果没有及时地把对象设置为 null

内部类持有外部类

Java 的非静态内部类的这种创建方式，会隐式地持有外部类的引用，而且默认情况下这个引用是强引用，因此，如果内部类的生命周期长于外部类的生命周期，程序很容易就产生内存泄漏

如果内部类的生命周期**长于**外部类的生命周期，程序很容易就产生内存泄漏（你认为垃圾回收器会回收掉外部类的实例，但由于内部类持有外部类的引用，导致垃圾回收器不能正常工作）

解决方法：你可以在内部类的内部显示持有一个外部类的软引用(或弱引用)，并通过构造方法的方式传递进来，在内部类的使用过程中，先判断一下外部类是否被回收；

Hash 值改变

在集合中，如果修改了对象中的那些参与计算哈希值的字段，会导致无法从集合中单独删除当前对象，造成内存泄露（有代码案例 Node 类）

内存泄漏和内存溢出辨析

内存溢出：实实在在的内存空间不足导致；

内存泄漏：该释放的对象没有释放，常见于使用容器保存元素的情况下。

如何避免：

内存溢出：检查代码以及设置足够的空间

内存泄漏：一定是代码有问题

往往很多情况下，内存溢出往往是内存泄漏造成的。

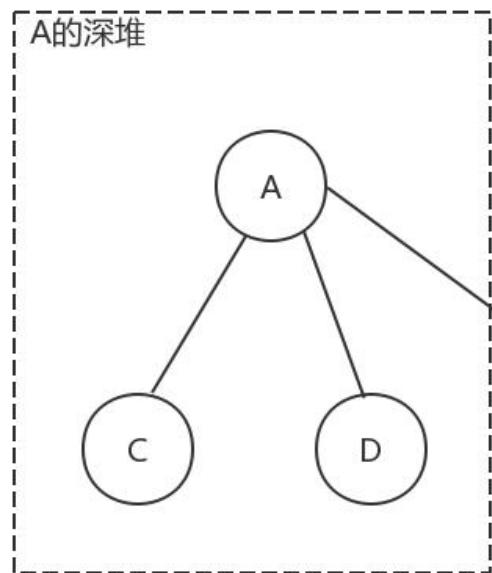
了解 MAT

浅堆和深堆

浅堆：(Shallow Heap) 是指一个对象所消耗的内存。例如，在 32 位系统中，一个对象引用会占据 4 个字节，一个 int 类型会占据 4 个字节， long 型变量会占据 8 个字节，每个对象头需要占用 8 个字节。

深堆：这个对象被 GC 回收后，可以真实释放的内存大小，也就是只能通过对象被直接或间接访问到的所有对象的集合。通俗地说，就是指仅被对象所持有的对象的集合。深堆是指对象的保留集中所有的对象的浅堆大小之和。

举例：对象 A 引用了 C 和 D，对象 B 引用了 E。那么对象 A 的浅堆大小只是 A 本身，而如果 A 被回收，那么 C 和 D 都会被回收(可达性分析算法)，所以 A 的深堆大小为 A+C+D 之和，同时由于对象 E 还可以通过对象 B 访问到，因此不在对象 A 的深堆范围内。



Class Name	Shallow Heap	Retained Heap	Percentage
java.lang.Thread @ 0xff005d38 main	120	28,375,048	98.24%
└─ java.util.LinkedList @ 0xff005c38	32	28,374,472	98.24%
└─ java.util.LinkedList\$Node @ 0xfe2382a8	24	40	0.00%
└─ java.util.LinkedList\$Node @ 0xfe902580	24	40	0.00%
└─ java.util.LinkedList\$Node @ 0xfe581028	24	40	0.00%
└─ java.util.LinkedList\$Node @ 0xfe9025a8	24	40	0.00%
└─ java.util.LinkedList\$Node @ 0xfe1020010	24	40	0.00%

深堆-浅堆=16字节

```
list.add(new Object());
```

↑ 这个对应16个字节

incoming 和 outgoing

JDK 为我们提供的工具

命令行工具

jps

列出当前机器上正在运行的虚拟机进程，JPS 从操作系统的临时目录上去找（所以有一些信息可能显示不全）。

-q :仅仅显示进程，

-m:输出主函数传入的参数。下的 hello 就是在执行程序时从命令行输入的参数

-l: 输出应用程序主类完整 package 名称或 jar 完整名称。

-v: 列出 jvm 参数, -Xms20m -Xmx50m 是启动程序指定的 jvm 参数

jstat

是用于监视虚拟机各种运行状态信息的命令行工具。它可以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，在没有 GUI 图形界面，只提供了纯文本控制台环境的服务器上，它将是运行期定位虚拟机性能问题的首选工具。

假设需要每 250 毫秒查询一次进程 13616 垃圾收集状况，一共查询 10 次，那命令应当是：jstat -gc 13616 250 10

常用参数：

- class (类加载器)
- compiler (JIT)
- gc (GC 堆状态)
- gccapacity (各区大小)
- gccause (最近一次 GC 统计和原因)
- gcnew (新区统计)
- gcnewcapacity (新区大小)
- gcold (老区统计)
- gcoldcapacity (老区大小)
- gcpermcapacity (永久区大小)
- gcutil (GC 统计汇总)
- printcompilation (HotSpot 编译统计)

jinfo

查看和修改虚拟机的参数

jinfo -sysprops 可以查看由 `System.getProperties()` 取得的参数

jinfo -flag 未被显式指定的参数的系统默认值

jinfo -flags (注意 s) 显示虚拟机的参数

jinfo -flag +[参数] 可以增加参数，但是仅限于由 `java -XX:+PrintFlagsFinal -version` 查询出来且

```
bool CMSAbortSemantics          = false           {product}
uintx CMSAbortablePrecleanMinWorkPerIteration = 100            {product}
intx CMSAbortablePrecleanWaitMillis       = 100            {product}
uintx CMSBitMapYieldQuantum           = 10485760        {product}
uintx CMSBootstrapOccupancy          = 50             {product}
```

为 manageable 的参数

jinfo -flag -[参数] pid 可以修改参数

Thread.getAllStackTraces();

案例：JinfoTest 类

1. 程序运行时只打印简单 GC

The screenshot shows the IntelliJ IDEA interface with the following details:

- Project View:** Shows the project structure under "ref-jvm". The "bin" directory is highlighted in yellow. Inside "com.jvm.ch05", the "JinfoTest" class is selected.
- Code Editor:** Displays the content of `JinfoTest.java`. The code creates a byte array of size $1 * 1024 * 1024$ and enters a loop where it repeatedly allocates and nullifies this array, causing frequent GC events.
- Terminal:** Shows the output of running the application with the command `"C:\Program Files\Java\jdk1.8.0_191\bin\java.exe" ...`. The output indicates multiple [GC (Allocation Failure)] events and one [Full GC (Allocation Failure)] event.

```
package com.jvm.ch05;
/*
 * @author 【享学课堂】 King老师
 * VM Args: -Xms20m -Xmx20m -Xmn2m -XX:+PrintGC
 * jinfo演示
 */
public class JinfoTest {
    //填充数据，造成GC
    public static void main(String[] args) {
        while (true){
            byte[] b=null;
            for(int i=0;i<10;i++){
                b=new byte[1*1024*1024];
            }
            try{
                Thread.sleep( millis: 5000 );
            }catch (InterruptedException e){
                e.printStackTrace();
            }
        }
    }
}
```

```
"C:\Program Files\Java\jdk1.8.0_191\bin\java.exe" ...
[GC (Allocation Failure) 1024K->624K(19968K), 0.0011261 secs]
[GC (Allocation Failure) 18672K->18136K(19968K), 0.0018935 secs]
[GC (Allocation Failure) 18136K->18208K(19968K), 0.0016196 secs]
[Full GC (Allocation Failure) 18208K->1648K(19968K), 0.0135614 secs]
```

2.通过 jinfo 修改 参数，打印 GC 详情

```
C:\Users\Administrator>jps  
13248 JinfoTest  
13616  
14656 Jps  
7596 Launcher  
  
C:\Users\Administrator>jinfo -flag PrintGCDetails 13248  
-XX:-PrintGCDetails  
  
C:\Users\Administrator>jinfo -flag +PrintGCDetails 13248  
  
C:\Users\Administrator>jinfo -flag PrintGCDetails 13248  
-XX:+PrintGCDetails
```

```
[GC (Allocation Failure) 18010K->18010K(19968K), 0.0027790 secs]
[GC (Allocation Failure) 18010K->18010K(19968K), 0.0014418 secs]
[Full GC (Allocation Failure) 18010K->1626K(19968K), 0.0075178 secs] 出现GC详情
[GC (Allocation Failure) 18010K->18010K(19968K), 0.0009170 secs]
[GC (Allocation Failure) 18010K->18010K(19968K), 0.0005611 secs]
[Full GC (Allocation Failure) 18010K->1626K(19968K), 0.0027790 secs]
[GC (Allocation Failure) [PSYoungGen: 0K->0K(1536K)] 18010K->18010K(19968K), 0.0010194 secs] [Times: user=0.00 sys=0.00, real=0.00]
[GC (Allocation Failure) [PSYoungGen: 0K->0K(1536K)] 18010K->18010K(19968K), 0.0009202 secs] [Times: user=0.00 sys=0.00, real=0.00]
[Full GC (Allocation Failure) [PSYoungGen: 0K->0K(1536K)] [ParOldGen: 18010K->1626K(18432K)] 18010K->1626K(19968K), [Metaspace: 32000K->32000K(32000K)], 0.0011607 secs] [Times: user=0.00 sys=0.00, real=0.00]
[GC (Allocation Failure) [PSYoungGen: 0K->0K(1536K)] 18010K->18010K(19968K), 0.0009141 secs] [Times: user=0.00 sys=0.00, real=0.00]
[GC (Allocation Failure) [PSYoungGen: 0K->0K(1536K)] [ParOldGen: 18010K->1626K(18432K)] 18010K->1626K(19968K), [Metaspace: 32000K->32000K(32000K)], 0.0009141 secs] [Times: user=0.00 sys=0.00, real=0.00]
```

jmap

用于生成堆转储快照（一般称为 `heapdump` 或 `dump` 文件）。`jmap` 的作用并不仅仅是为了获取 `dump` 文件，它还可以查询 `finalize` 执行队列、Java 堆和永久代的详细信息，如空间使用率、当前用的是哪种收集器等。和 `jinfo` 命令一样，`jmap` 有不少功能在 Windows 平台下都是受限的，除了生成 `dump` 文件的

-dump 选项和用于查看每个类的实例、空间占用统计的-histo 选项在所有操作系统都提供之外，其余选项都只能在 Linux/Solaris 下使用。

jmap -dump:live,format=b,file=heap.bin <pid>

Sun JDK 提供 jhat (JVM Heap Analysis Tool) 命令与 jmap 搭配使用，来分析 jmap 生成的堆转储快照。

```
C:\Users\Administrator>jmap
Usage:
    jmap [option] <pid>
        (to connect to running process)
    jmap [option] <executable <core>
        (to connect to a core file)
    jmap [option] [server_id@]<remote server IP or hostname>
        (to connect to remote debug server)

where <option> is one of:
<none>          to print same info as Solaris pmap
-heap            to print java heap summary
-histo[:live]    to print histogram of java object heap; if the "live"
                  suboption is specified, only count live objects
-clstats         to print class loader statistics
-finalizerinfo   to print information on objects awaiting finalization
-dump:<dump-options> to dump java heap in hprof binary format
                      dump-options:
                          live      dump only live objects; if not specified,
                                    all objects in the heap are dumped.
                          format=b  binary format
                          file=<file> dump heap to <file>
Example: jmap -dump:live,format=b,file=heap.bin <pid>
-F               force. Use with -dump:<dump-options> <pid> or -histo
                  to force a heap dump or histogram when <pid> does not
                  respond. The "live" suboption is not supported
                  in this mode.
-h | -help       to print this help message
-J<flag>        to pass <flag> directly to the runtime system
```

```
C:\Users\Administrator>jmap -dump:live,format=b,file=heap.bin 14532
Dumping heap to C:\Users\Administrator\heap.bin ...
```

jhat

jhat dump 文件名

后屏幕显示“Server is ready.”的提示后，用户在浏览器中键入 <http://localhost: 7000> 就可以访问详情

```
C:\Users\Administrator>jhat C:\Users\Administrator\heap.bin
Reading from C:\Users\Administrator\heap.bin...
Dump file created Tue Jun 04 09:20:19 CST 2019
Snapshot read, resolving...
Resolving 8423 objects...
Chasing references, expect 1 dots.
Eliminating duplicate references.
Snapshot resolved.
Started HTTP server on port 7000
Server is ready.
```

使用 jhat 可以在服务器上生成堆转储文件分析（一般不推荐，毕竟占用服务器的资源，比如一个文件就有 1 个 G 的话就需要大约吃一个 1G 的资源）

jstack

(Stack Trace for Java) 命令用于生成虚拟机当前时刻的线程快照。线程快照就是当前虚拟机内每一条线程正在执行的方法堆栈的集合，生成线程快照的主要目的是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都是导致线程长时间停顿的常见原因。

在代码中可以用 `java.lang.Thread` 类的 `getAllStackTraces()` 方法用于获取虚拟机中所有线程的 `StackTraceElement` 对象。使用这个方法可以通过简单的几行代码就完成 `jstack` 的大部分功能，在实际项目中不妨调用这个方法做个管理员页面，可以随时使用浏览器来查看线程堆栈。（并发编程中的线程安全课程中有具体的案例）

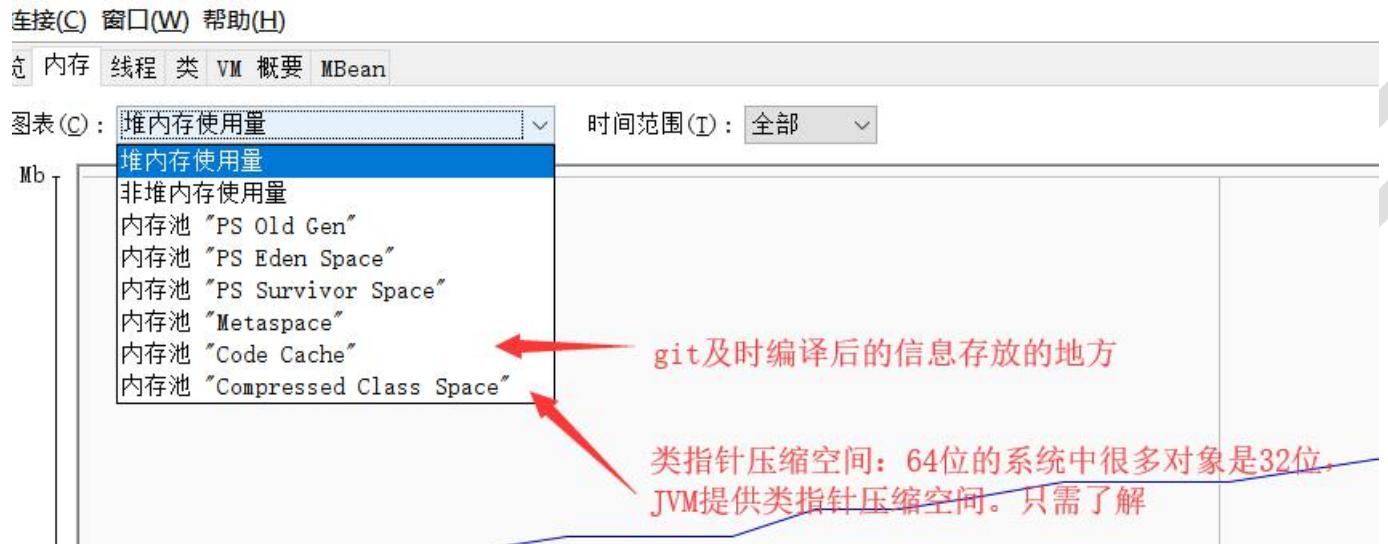
可视化工具

JMX（Java Management Extensions，即 Java 管理扩展）是一个为应用程序、设备、系统等植入管理功能的框架。JMX 可以跨越一系列异构操作系统平台、系统体系结构和网络传输协议，灵活的开发无缝集成的系统、网络和服务管理应用。

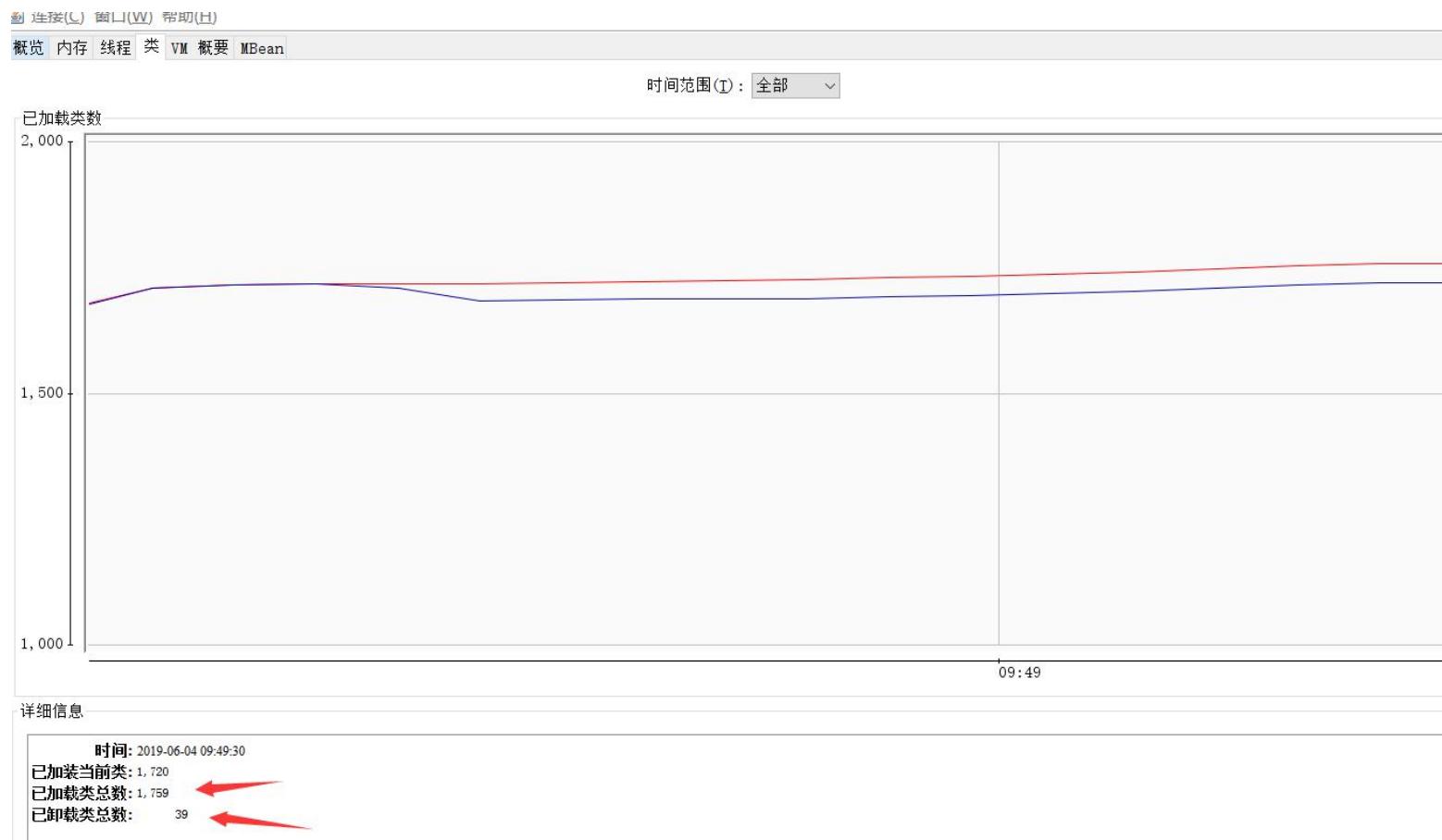
管理远程进程需要在远程程序的启动参数中增加：

- Djava.rmi.server.hostname=.....
- Dcom.sun.management.jmxremote
- Dcom.sun.management.jmxremote.port=8888
- Dcom.sun.management.jmxremote.authenticate=false
- Dcom.sun.management.jmxremote.ssl=false

Jconsole







概览 内存 线程 类 VM 概要 MBean

VM 概要

2019年6月4日 星期二 上午09时50分06秒 CST

连接名称: pid: 14532 com.jvm.ch05.JinfoTest
虚拟机: Java HotSpot(TM) 64-Bit Server VM 版本 25.191-b12
供应商: Oracle Corporation
名称: 14532@90JQ9A4J2G988F

运行时间: 36 分钟
进程 CPU 时间: 11,250 秒
JIT 编译器: HotSpot 64-Bit Tiered Compilers
总编译时间: 1,150 秒

活动线程: 12
峰值: 15
守护程序线程: 11
启动的线程总数: 15

已加载当前类: 1,720
已加载类总数: 1,759
已卸载类总数: 39

当前堆大小: 4,394 KB

提交的内存: 19,968 KB

最大堆大小: 19,968 KB

暂挂最终处理: 0 对象

垃圾收集器: 名称 = 'PS MarkSweep', 收集 = 277, 总花费时间 = 1.560 秒
垃圾收集器: 名称 = 'PS Scavenge', 收集 = 556, 总花费时间 = 0.664 秒

操作系统: Windows 10 10.0
体系结构: amd64
处理程序数: 12
提交的虚拟内存: 84,136 KB

总物理内存: 16,662,820 KB
空闲物理内存: 10,544,532 KB
总交换空间: 17,711,396 KB
空闲交换空间: 8,749,292 KB

VM 参数: -Xms20m -Xmx20m -Xmn2m -XX:+PrintGC -javaagent:C:\Program Files\JetBrains\IntelliJ IDEA 2018.2.5\lib\idea_rt.jar=52263,C:\Program Files\JetBrains\IntelliJ IDEA 2018.2.5\bin -Dfile.encoding=UTF-8
类路径: D:\work\ref-jvm\bin;C:\Program Files\Java\jdk1.8.0_191\jre\lib\charsets.jar;C:\Program Files\Java\jdk1.8.0_191\jre\lib\deploy.jar;C:\Program Files\Java\jdk1.8.0_191\jre\lib\ext\access-bridge-64.jar;C:\Program Files\Java\jdk1.8.0_191\jre\lib\ext\cldrdata.jar;C:\Program Files\Java\jdk1.8.0_191\jre\lib\ext\dnsns.jar;C:\Program Files\Java\jdk1.8.0_191\jre\lib\ext\jaccess.jar;C:\Program Files\Java\jdk1.8.0_191\jre\lib\ext\jfxrt.jar;C:\Program Files\Java\jdk1.8.0_191\jre\lib\ext\localizedata.jar;C:\Program Files\Java\jdk1.8.0_191\jre\lib\ext\nashorn.jar;C:\Program Files\Java\jdk1.8.0_191\jre\lib\ext\sunec.jar;C:\Program Files\Java\jdk1.8.0_191\jre\lib\ext\sunec_provider.jar;C:\Program Files\Java\jdk1.8.0_191\jre\lib\ext\sunmsapi.jar;C:\Program Files\Java\jdk1.8.0_191\jre\lib\ext\sunpkcs11.jar;C:\Program Files\Java\jdk1.8.0_191\jre\lib\ext\zipsf.jar;C:\Program Files\Java\jdk1.8.0_191\jre\lib\javaws.jar;C:\Program Files\Java\jdk1.8.0_191\jre\lib\resources.jar;C:\Program Files\Java\jdk1.8.0_191\jre\lib\rt.jar;C:\Program Files\JetBrains\IntelliJ IDEA 2018.2.5\lib\idea_rt.jar
库路径: C:\Program Files\Java\jdk1.8.0_191\bin;C:\Windows\SunJava\bin;C:\Windows\system32;C:\Windows;C:\Windows\System32;C:\Windows\System32\Wbem;C:\Windows\System32\WindowsPowerShell\v1.0;C:\Program Files\IDM Computer Solutions\UltraEdit;C:\Program Files\Java\jdk1.8.0_191\bin;C:\Program Files\Java\jdk1.8.0_191\jre\bin;C:\Users\Administrator\AppData\Local\Microsoft\WindowsApps;C:\Users\Administrator\AppData\Roaming\npm;C:\Program Files(x86)\SSH Communications Security\SSH Secure Shell;
引导类路径: C:\Program Files\Java\jdk1.8.0_191\jre\lib\resources.jar;C:\Program Files\Java\jdk1.8.0_191\jre\lib\rt.jar;C:\Program Files\Java\jdk1.8.0_191\jre\lib\sunrsrcsign.jar;C:\Program Files\Java\jdk1.8.0_191\jre\lib\jsse.jar;C:\Program Files\Java\jdk1.8.0_191\jre\lib\jce.jar;C:\Program Files\Java\jdk1.8.0_191\jre\lib\charsets.jar;C:\Program Files\Java\jdk1.8.0_191\jre\lib\jfr.jar;C:\Program Files\Java\jdk1.8.0_191\jre\classes

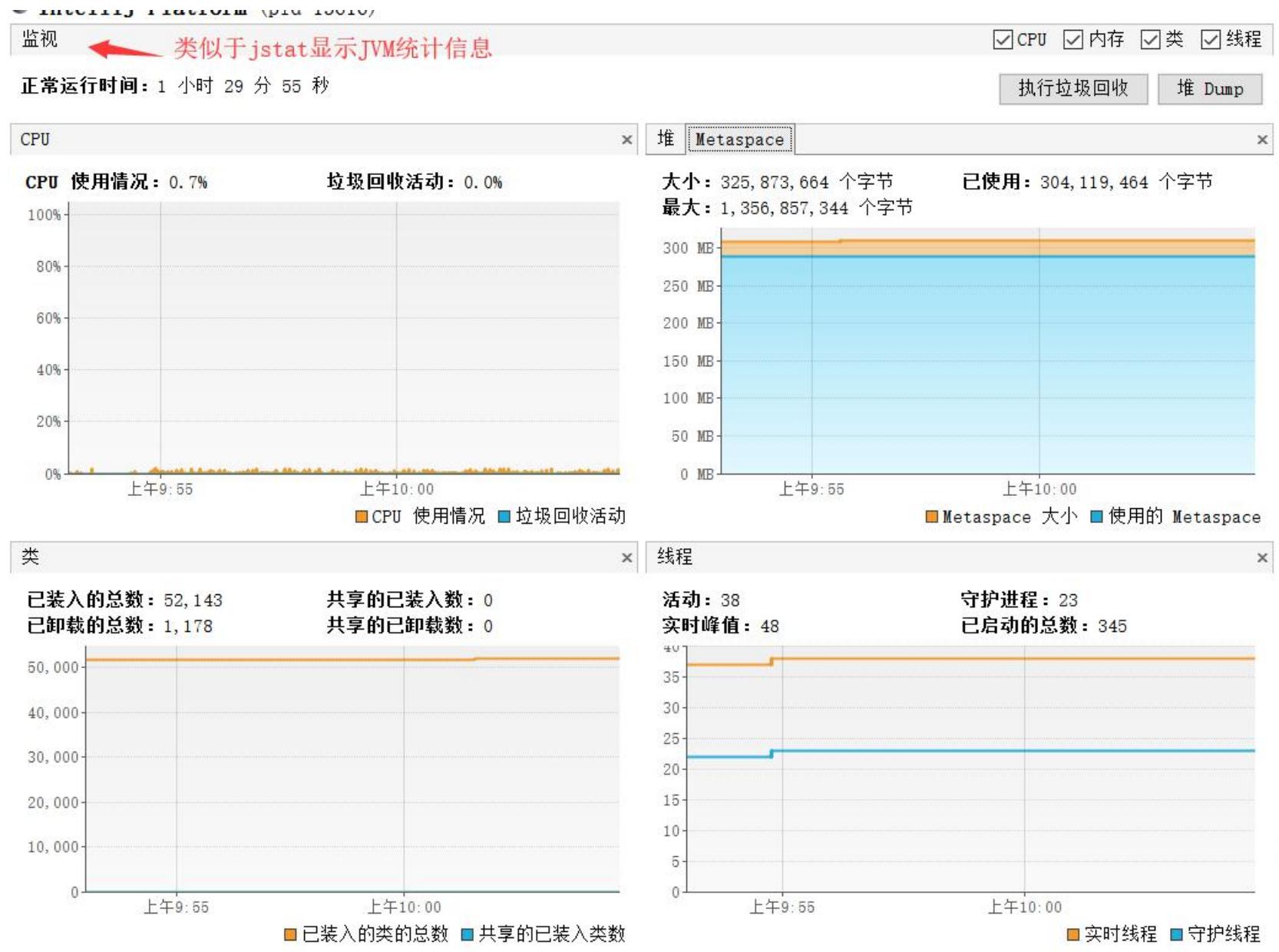
visualvm

插件中心地址

<https://visualvm.github.io>

但是注意版本问题，不同的 JDK 所带的 visualvm 是不一样的，下载插件时需要下对应的版本。

一般来说，这个工具是本机调试用，一般生产上来说，你一般是用不了的（除非启用远程连接）



概述 监视 线程 抽样器 Visual GC

○ IntelliJ Platform (pid 13616)

概述 保存的数据 详细信息

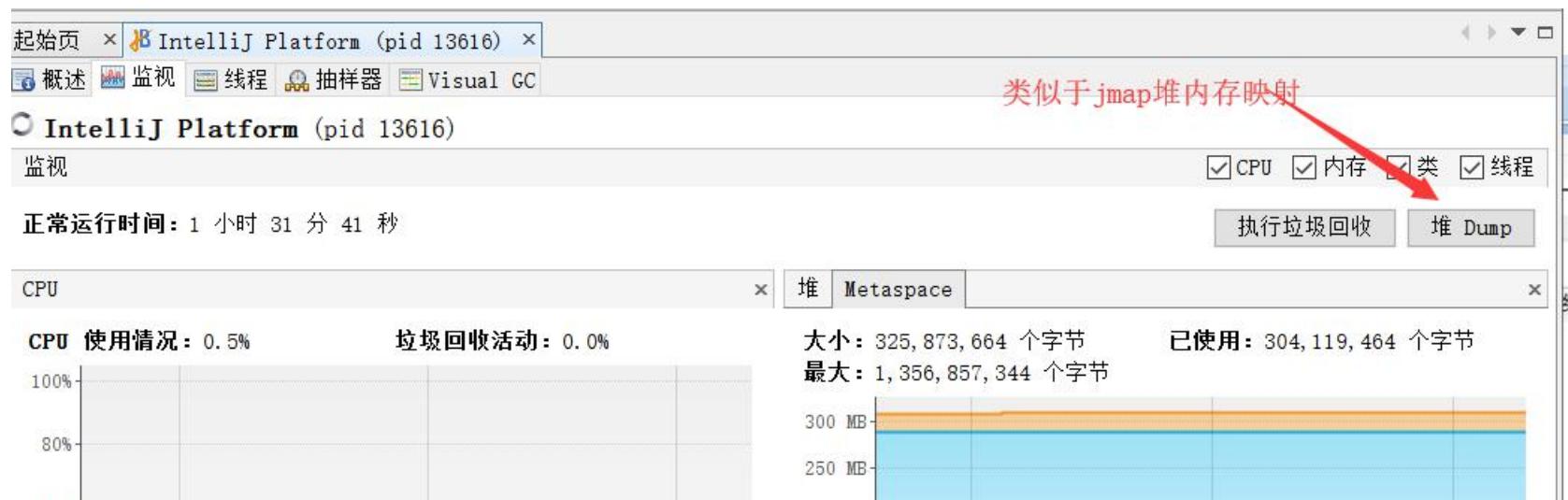
PID: 13616
主机: localhost
主类: <未知>
参数: <无>

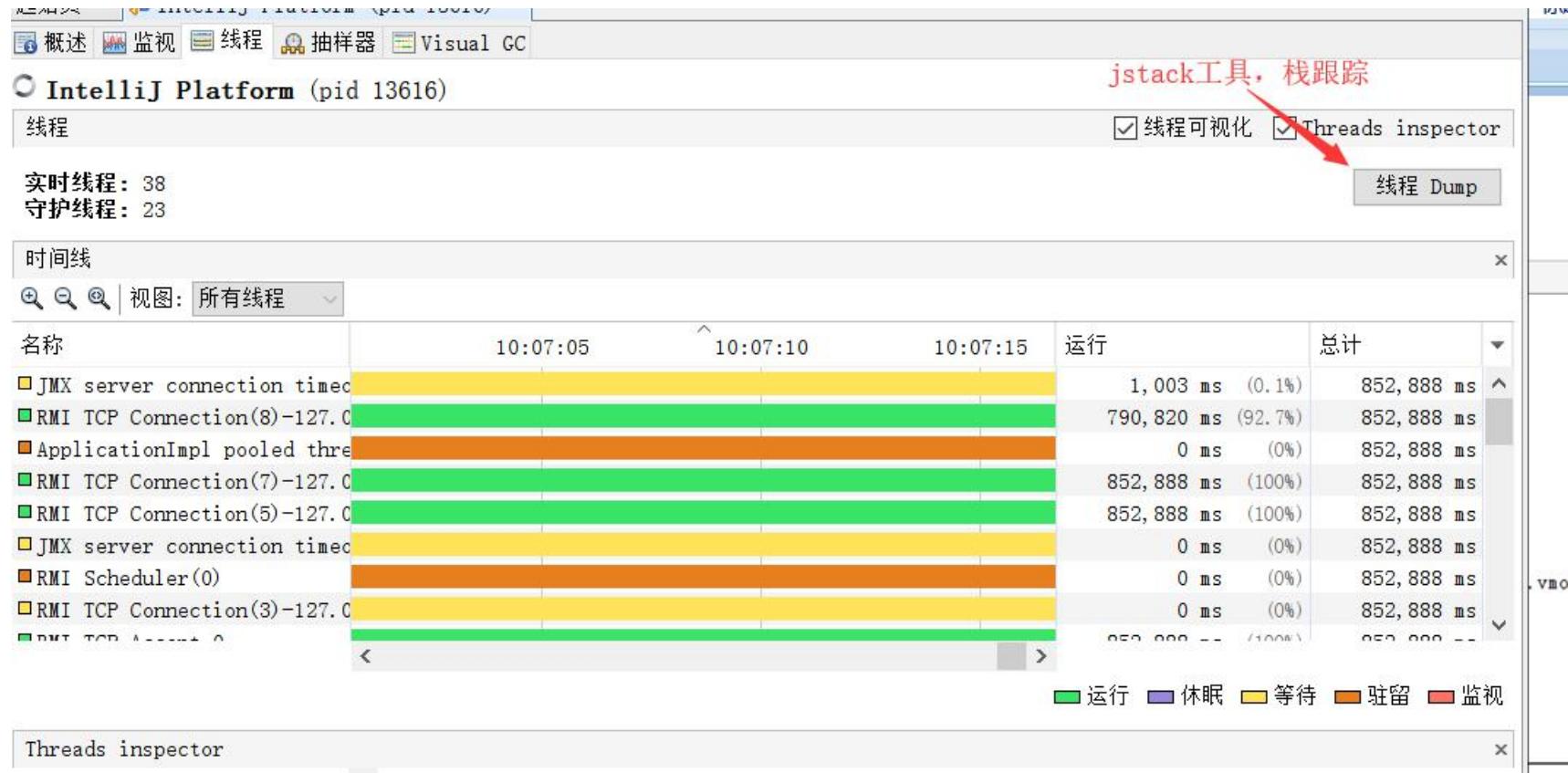
JVM: OpenJDK 64-Bit Server VM (25.152-b19, mixed mode)
Java: 版本 1.8.0_152-release, 供应商 JetBrains s.r.o
Java Home 目录: C:\Program Files\JetBrains\IntelliJ IDEA 2018.2.5\jre64
JVM 标志: <无>

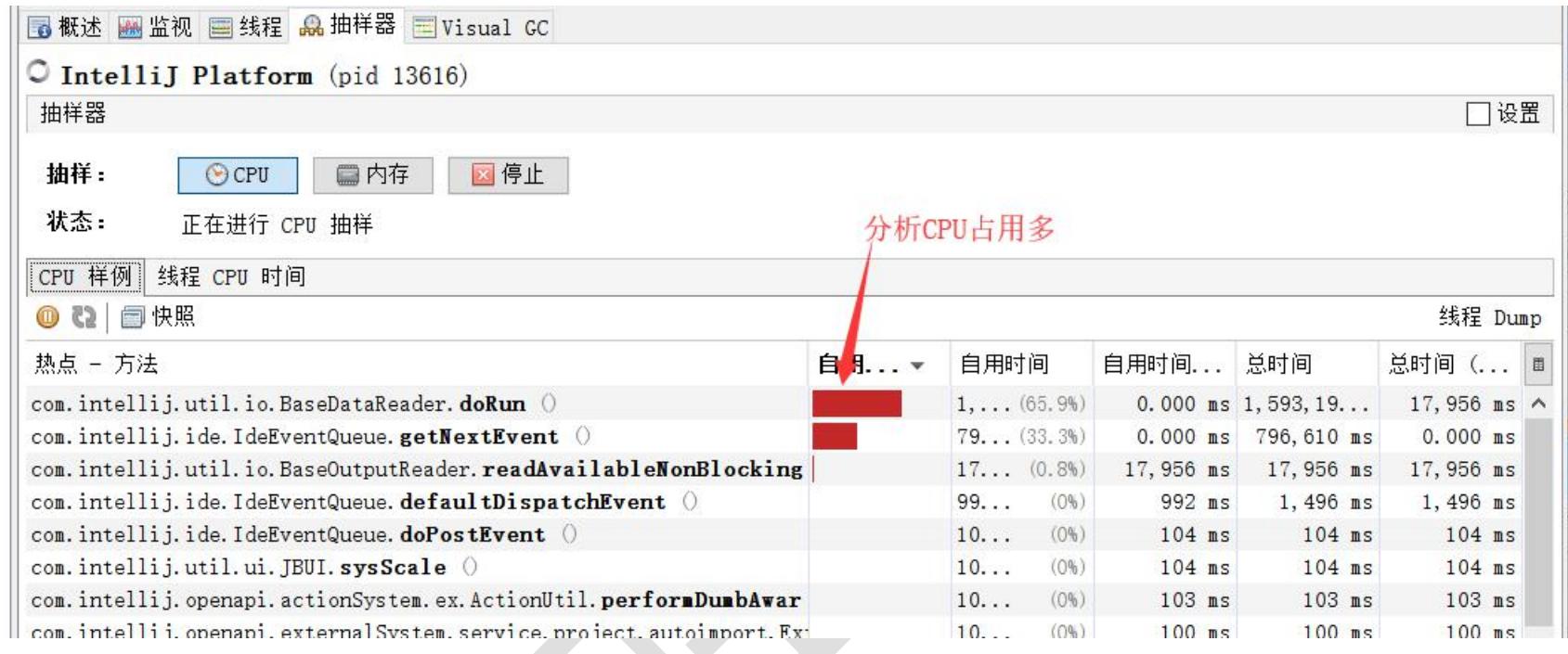
类似于jinfo显示JVM参数配置

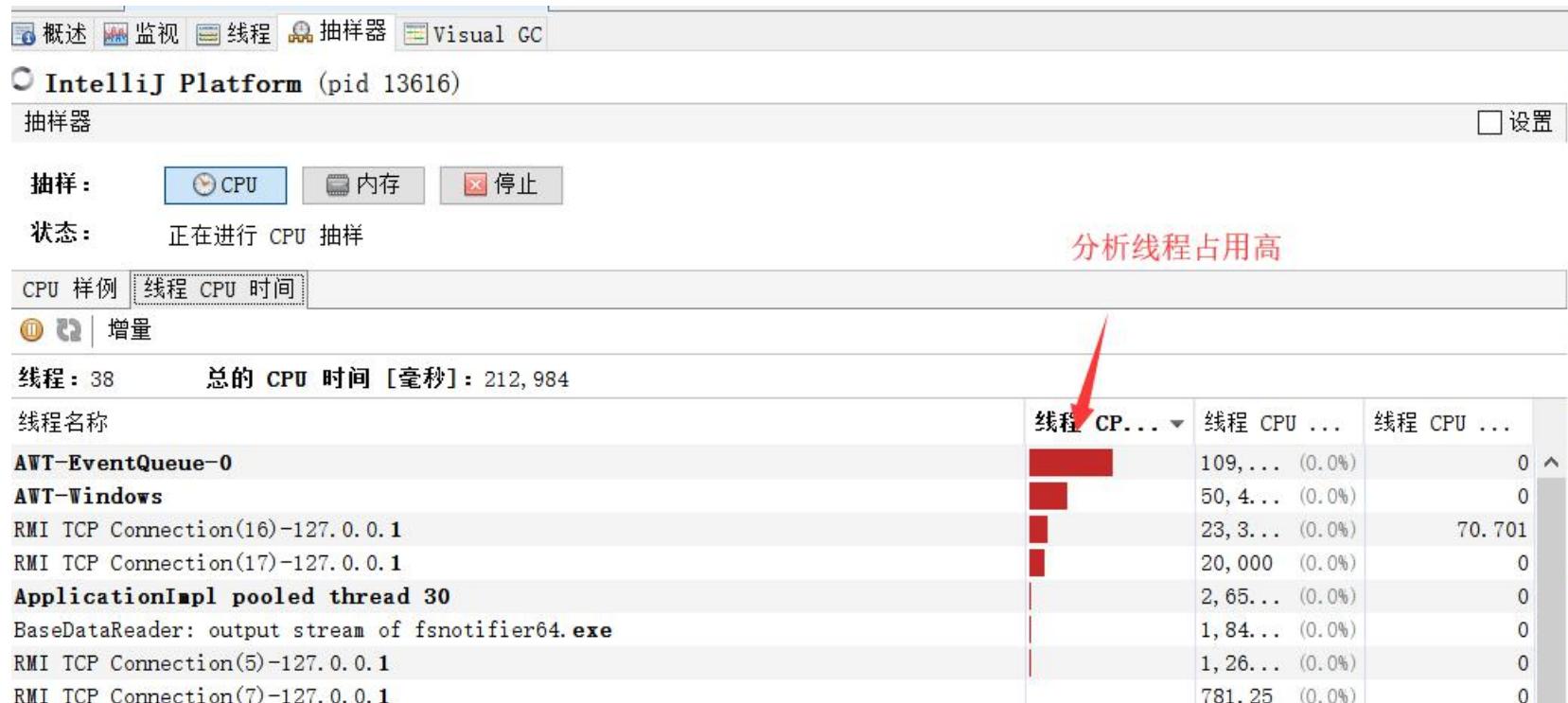
出现 OOME 时生成堆 dump: 启用

保存的数据	JVM 参数	系统属性
	<pre>-Xms128m -Xmx750m -XX:ReservedCodeCacheSize=240m -XX:+UseConcMarkSweepGC -XX:SoftRefLRUPolicyMSPerMB=50 -ea -Dsun.io.useCanonCaches=false -Djava.net.preferIPv4Stack=true -Djdk.http.auth.tunneling.disabledSchemes="" -XX:+HeapDumpOnOutOfMemoryError -XX:-OmitStackTraceInFastThrow -Djb.vmOptionsFile=C:\Program Files\JetBrains\IntelliJ IDEA 2018.2.5\bin\idea64.exe.vmo -Didea.jre.check=true -Didea.native.launcher=true -Didea.paths.selector=IntelliJIdea2018.2 -XX:ErrorFile=C:\Users\Administrator\java_error_in_idea_%p.log -XX:HeapDumpPath=C:\Users\Administrator\java_error_in_idea.hprof</pre>	









GC 的重要参数

生产服务器推荐开启

-XX:-HeapDumpOnOutOfMemoryError 默认关闭，建议开启，在 java.lang.OutOfMemoryError 异常出现时，输出一个 dump.core 文件，记录当时的堆内存快照。

`-XX:HeapDumpPath=./java_pid<pid>.hprof` 用来设置堆内存快照的存储文件路径，默认是 java 进程启动位置，。

调优之前开启、调优之后关闭

`-XX:+PrintGC`

调试跟踪之 打印简单的 GC 信息参数:

`-XX:+PrintGCDetails, +XX:+PrintGCTimeStamps`

打印详细的 GC 信息

`-Xlogger:logpath`

设置 gc 的日志路，如: `-Xlogger:log/gc.log`, 将 gc.log 的路径设置到当前目录的 log 目录下.

应用场景: 将 gc 的日志独立写入日志文件，将 GC 日志与系统业务日志进行了分离，方便开发人员进行追踪分析。

考虑使用

`-XX:+PrintHeapAtGC`, 打印堆信息

参数设置: `-XX: +PrintHeapAtGC`

应用场景: 获取 Heap 在每次垃圾回收前后的使用状况

`-XX:+TraceClassLoading`

参数方法: `-XX:+TraceClassLoading`

应用场景: 在系统控制台信息中看到 class 加载的过程和具体的 class 信息，可用以分析类的加载顺序以及是否可进行精简操作。

`-XX:+DisableExplicitGC` 禁止在运行期显式地调用 `System.gc()`

第六节：JVM 调优和深入了解性能优化

JVM 调优的本质：

并不是显著的提高系统性能，不是说你调了，性能就能提升几倍或者上十倍，JVM 调优，主要调的是稳定。如果你的系统出现了频繁的垃圾回收，这个时候系统是不稳定的，所以需要我们来进行 JVM 调优，调整垃圾回收的频次。

GC 调优原则

调优的原则

- 1、大多数的 java 应用不需要 GC 调优
- 2、大部分需要 GC 调优的，不是参数问题，是代码问题
- 3、在实际使用中，分析 GC 情况优化代码比优化 GC 参数要多得多；
- 4、GC 调优是最后的手段

目的

GC 的时间够小

GC 的次数够少

发生 Full GC 的周期足够的长，时间合理，最好是不发生。

注：如果满足下面的指标，则一般不需要进行 GC：

Minor GC 执行时间不到 50ms；

Minor GC 执行不频繁，约 10 秒一次；

Full GC 执行时间不到 1s;

Full GC 执行频率不算频繁，不低于 10 分钟 1 次；

GC 调优

调优步骤

日志分析

1, 监控 GC 的状态

使用各种 JVM 工具，查看当前日志，分析当前 JVM 参数设置，并且分析当前堆内存快照和 gc 日志，根据实际的各区域内存划分和 GC 执行时间，觉得是否进行优化；

2, 分析结果，判断是否需要优化

如果各项参数设置合理，系统没有超时日志出现，GC 频率不高，GC 耗时不高，那么没有必要进行 GC 优化；如果 GC 时间超过 1-3 秒，或者频繁 GC，则必须优化；

3, 调整 GC 类型和内存分配

如果内存分配过大或过小，或者采用的 GC 收集器比较慢，则应该优先调整这些参数，并且先找 1 台或几台机器进行 beta，然后比较优化过的机器和没有优化的机器的性能对比，并有针对性的做出最后选择；

4, 不断的分析和调整

通过不断的试验和试错，分析并找到最合适的参数

5, 全面应用参数

如果找到了最合适的参数，则将这些参数应用到所有服务器，并进行后续跟踪。

阅读 GC 日志

主要关注 MinorGC 和 FullGC 的回收效率（回收前大小和回收比较）、回收的时间。

-XX:+UseSerialGC

以参数-Xms5m -Xmx5m -XX:+PrintGCDetails -XX:+UseSerialGC 为例：

[DefNew: 1855K->1855K(1856K), 0.0000148 secs][Tenured: 2815K->4095K(4096K), 0.0134819 secs] 4671K

DefNew 指明了收集器类型，而且说明了收集发生在新生代。

1855K->1855K(1856K)表示，回收前 新生代占用 1855K，回收后占用 1855K，新生代大小 1856K。

0.0000148 secs 表明新生代回收耗时。

Tenured 表明收集发生在老年代

2815K->4095K(4096K), 0.0134819 secs: 含义同新生代

最后的 4671K 指明堆的大小。

-XX:+UseParNewGC

收集器参数变为-XX:+UseParNewGC，日志变为：

[ParNew: 1856K->1856K(1856K), 0.0000107 secs][Tenured: 2890K->4095K(4096K), 0.0121148 secs]

收集器参数变为-XX:+ UseParallelGC 或 UseParallelOldGC，日志变为：

[PSYoungGen: 1024K->1022K(1536K)] [ParOldGen: 3783K->3782K(4096K)] 4807K->4804K(5632K),

-XX:+UseConcMarkSweepGC

CMS 收集器和 G1 收集器会有明显的相关字样

-XX:+UseG1GC

GC 调优实战

项目启动 GC 优化

- 1、开启日志分析 `-XX:+PrintGCDetails` 发现有多次 GC 包括 FullGC
- 2、调整 Metadata 空间 `-XX:MetaspaceSize=64m` 减少 FullGC
- 3、减少 Minor gc 次数，增加参数 `-Xms500m` GC 减少至 4 次
- 4、减少 Minor gc 次数，调整参数 `-Xms1000m` GC 减少至 2 次
- 5、增加新生代比重，增加参数 `-Xmn900m` GC 减少至 1 次
- 6、加大新生代，调整参数 `-Xms2000m -Xmn1800m` 还是避免不了 GC，没有必要调整这么大，资源浪费

项目运行 GC 优化

使用 jmeter 同时访问三个接口，index、time、noblemetal

使用 40 个线程，循环 2500 次进行压力测试，观察并发的变化

jmeter 的聚合报告的参数解释：

Label	# Samples	Average	Median	90% Line	95% Line	99% Line	Min	Max	Error %	Throughput	Received KB/sec	Sent KB/sec
nobleMetal	100000	3	2	6	10	29	0	383	0.00%	3904.1/sec	7641.20	488.01
time	100000	3	2	6	10	30	0	384	0.00%	3926.7/sec	3850.69	457.82
index	100000	3	2	6	10	29	0	401	0.00%	3947.3/sec	3196.32	474.13
总体	300000	3	2	6	10	29	0	401	0.00%	11711.9/sec	14630.60	1422.05

错误率（如果压测出现了错误率的话，本次次数就没有参考性）

平均响应时长（毫秒）

50%的请求响应低于这个值

90%的请求时间低于这个值

95%的请求时间低于这个值

最小请求时间

最大请求时间

吞吐量(这里是每秒)

XX:+UseParNewGC

1、使用单线程 GC -XX:+UseSerialGC

JVM压测.jmx (D:\tools\apache-jmeter-3.3\bin\JVM压测.jmx) - Apache JMeter (3.3 r1808647)

文件 编辑 Search 运行 选项 帮助

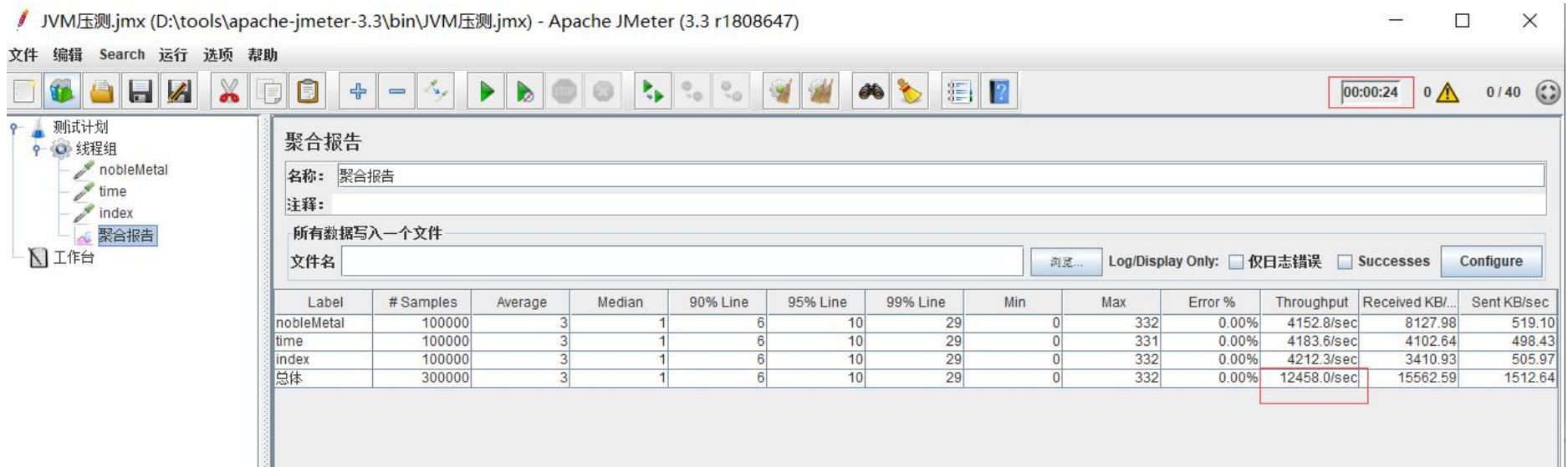
00:00:25 0 / 40

测试计划
线程组
nobleMetal
time
index
聚合报告

聚合报告
名称: 聚合报告
注释:
所有数据写入一个文件
文件名 浏览... Log/Display Only: 仅日志错误 Successes Configure

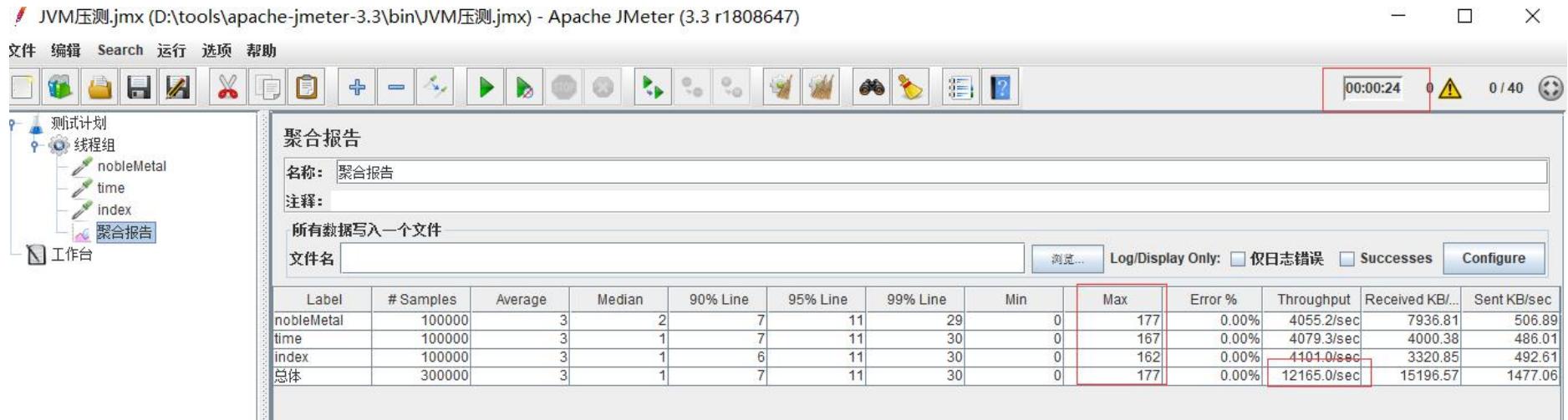
Label	# Samples	Average	Median	90% Line	95% Line	99% Line	Min	Max	Error %	Throughput	Received KB/sec	Sent KB/sec
nobleMetal	100000	3	2	6	10	29	0	383	0.00%	3904.1/sec	7641.20	488.01
time	100000	3	2	6	10	30	0	384	0.00%	3926.7/sec	3850.69	467.82
index	100000	3	2	6	10	29	0	401	0.00%	3947.3/sec	3196.32	474.13
总体	300000	3	2	6	10	29	0	401	0.00%	11711.9/sec	14630.60	1422.05

2、使用多线程 GC -XX:+UseParNewGC



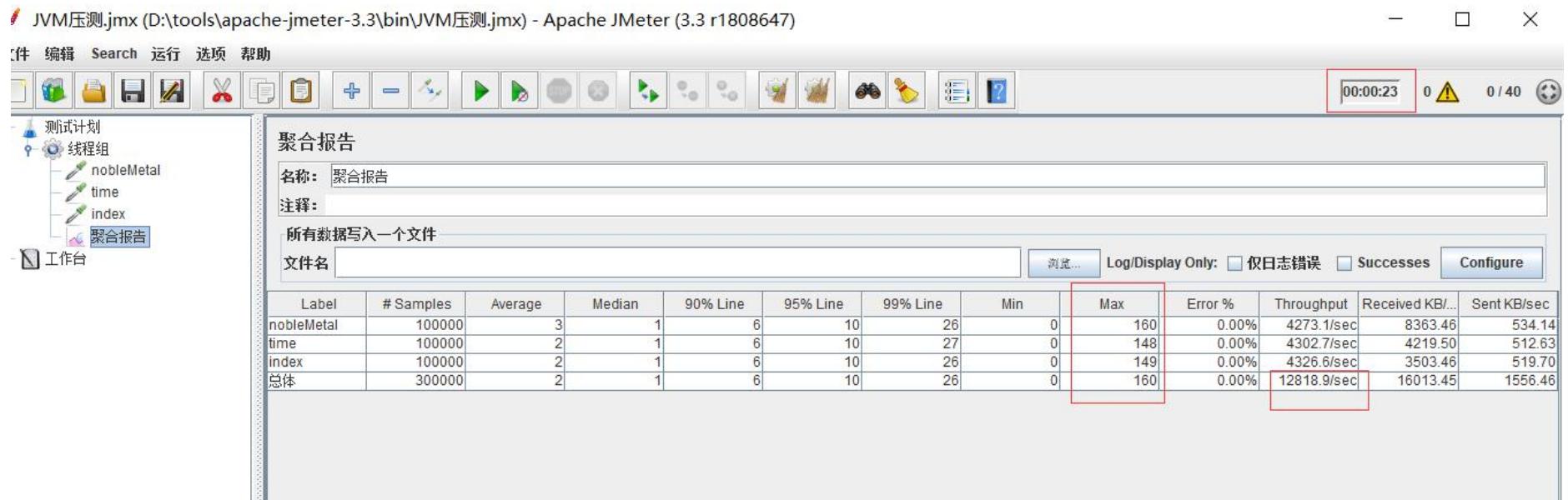
多线程的吞吐量有一定的上升

3、使用 CMS -XX:+UseConcMarkSweepGC



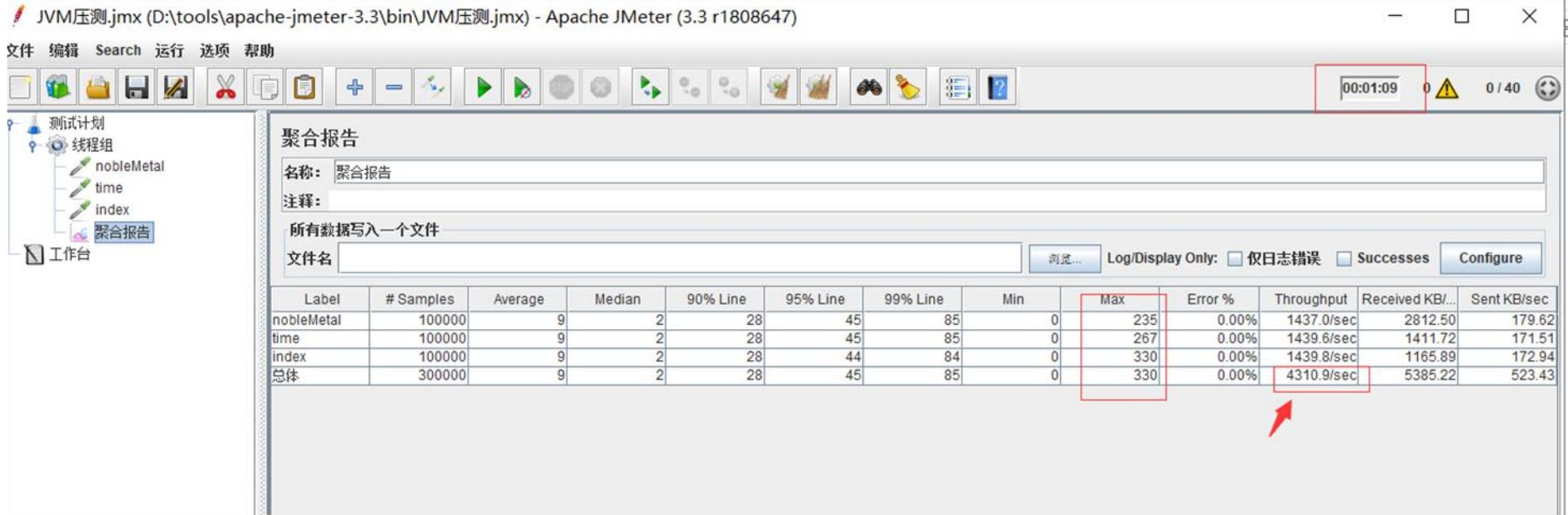
CMS 采用了并发收集，所以 STW 的时间较小，吞吐量较单线程有一定提高，最大请求时间 MAX 有明显的下降。

4、使用 G1 -XX:+UseG1GC



G1 这里的吞吐量是最大的，最大请求时间 MAX 有明显的下降。

一行代码导致频繁 GC，吞吐量下降很快



调优的原则

- 1、大多数的 **java** 应用不需要 **GC** 调优
- 2、大部分需要 **GC** 调优的，不是参数问题，是代码问题
- 3、在实际使用中，分析 **GC** 情况优化代码比优化 **GC** 参数要多得多；
- 4、**GC** 调优是最后的手段

推荐策略

1. 新生代大小选择

-
- 响应时间优先的应用:尽可能设大,直到接近系统的最低响应时间限制(根据实际情况选择).在此种情况下,新生代收集发生的频率也是最小的.同时,减少到达老年代的对象.
 - 吞吐量优先的应用:尽可能的设置大,可能到达 Gbit 的程度.因为对响应时间没有要求,垃圾收集可以并行进行,一般适合 8CPU 以上的应用.
 - 避免设置过小.当新生代设置过小时会导致:1.MinorGC 次数更加频繁 2.可能导致 MinorGC 对象直接进入老年代,如果此时老年代满了,会触发 FullGC.

2. 老年代大小选择

响应时间优先的应用:老年代使用并发收集器,所以其大小需要小心设置,一般要考虑并发会话率和会话持续时间等一些参数.如果堆设置小了,可以会造成内存碎片,高回收频率以及应用暂停而使用传统的标记清除方式;

如果堆大了,则需要较长的收集时间.最优化的方案,一般需要参考以下数据获得:

并发垃圾收集信息、持久代并发收集次数、传统 GC 信息、花在新生代和老年代回收上的时间比例。

吞吐量优先的应用:一般吞吐量优先的应用都有一个很大的新生代和一个较小的老年代.原因是,这样可以尽可能回收掉大部分短期对象,减少中期的对象,而老年代尽存放长期存活对象

GC 调优是个很复杂、很细致的过程,要根据实际情况调整,不同的机器、不同的应用、不同的性能要求调优的手段都是不同的, king 老师也无法告诉大家全部,即使是 jvm 参数也是如此,比如说性能有关的操作系统工具,和操作系统本身相关的所谓大页机制,都需要大家平时去积累,去观察,去实践, king 老师在这个专题上告诉大家的除了各种 java 虚拟机基础知识和内部原理,也告诉大家一个性能优化的一个基本思路和着手的方向。

逃逸分析

是 JVM 所做的最激进的优化,最好不要调整相关的参数。

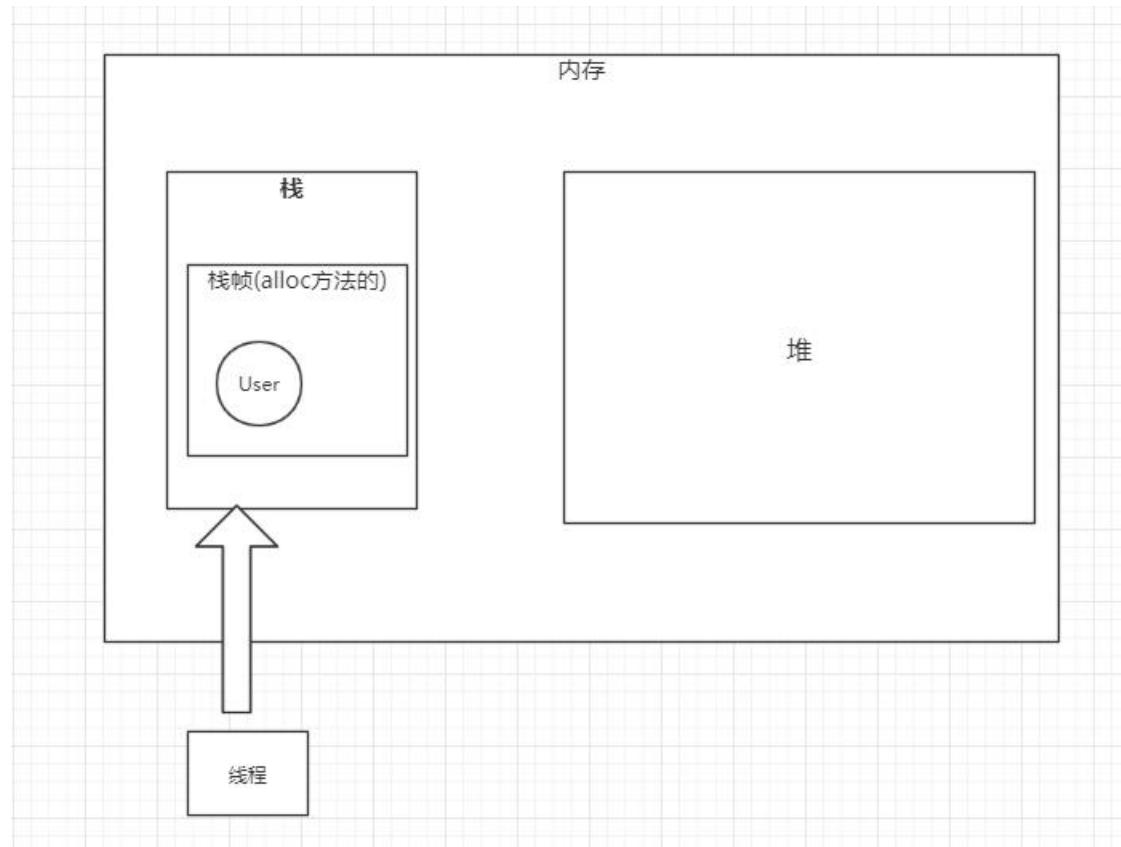
■ 牵涉到的 JVM 参数:

-XX:+DoEscapeAnalysis: 启用逃逸分析(默认打开)
-XX:+EliminateAllocations: 标量替换(默认打开)
-XX:+UseTLAB 本地线程分配缓冲(默认打开)

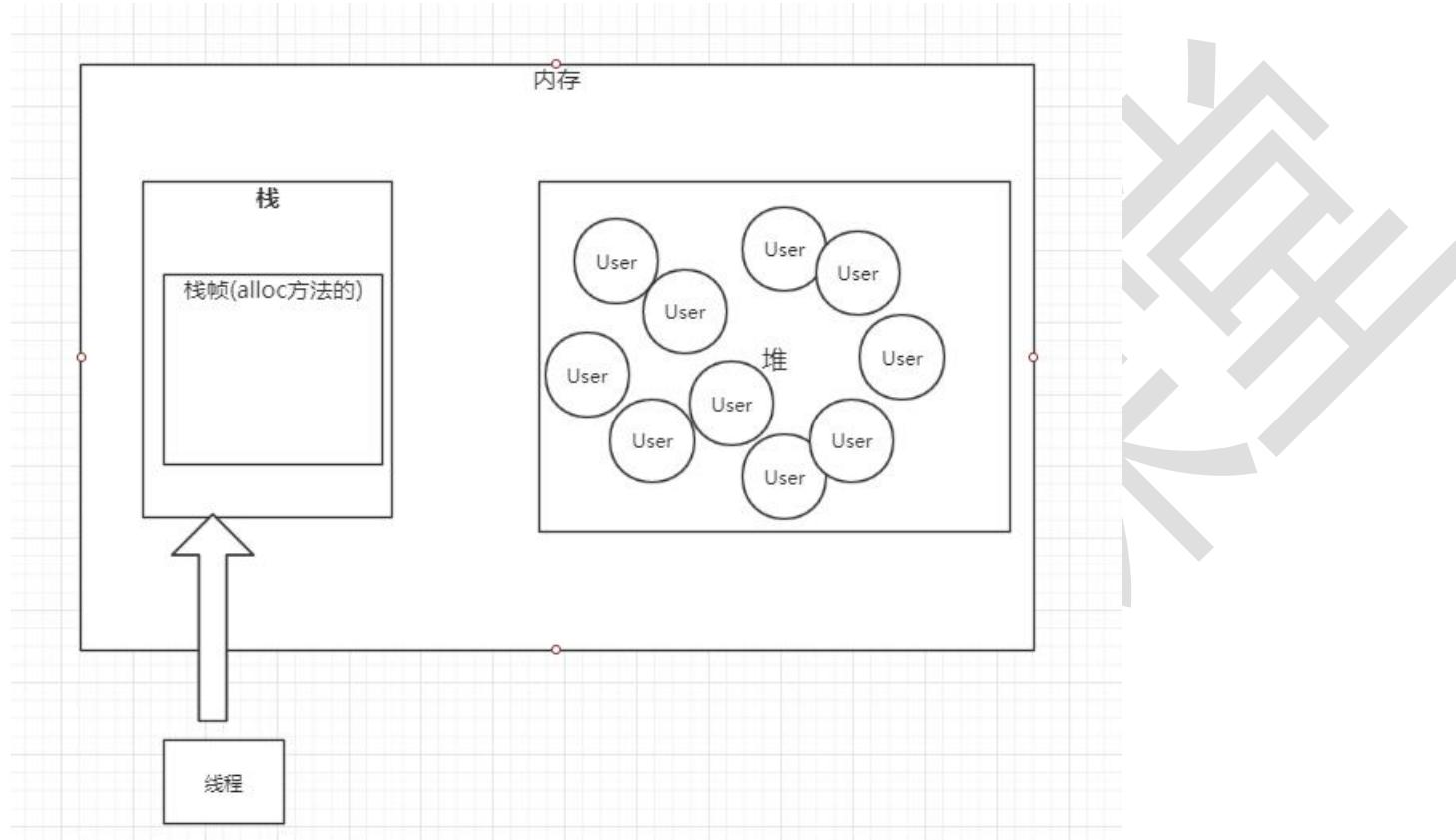
如果是逃逸分析出来的对象可以在栈上分配的话,那么该对象的生命周期就跟随线程了,就不需要垃圾回收,如果是频繁的调用此方法则可以得到很大

的性能提高。

采用了逃逸分析--对象在栈上分配：



没有逃逸分析---对象都在堆上分配（触发频次 GC，加重负担）：



常用的性能评价/测试指标

一个 web 应用不是一个孤立的个体，它是一个系统的部分，系统中的每一部分都会影响整个系统的性能

响应时间

提交请求和返回该请求的响应之间使用的时间，一般比较关注平均响应时间。

常用操作的响应时间列表：

操作	响应时间
打开一个站点	几秒
数据库查询一条记录（有索引）	十几毫秒
机械磁盘一次寻址定位	4 毫秒
从机械磁盘顺序读取 1M 数据	2 毫秒
从 SSD 磁盘顺序读取 1M 数据	0.3 毫秒
从远程分布式换成 Redis 读取一个数据	0.5 毫秒
从内存读取 1M 数据	十几微妙
Java 程序本地方法调用	几微妙
网络传输 2Kb 数据	1 微妙

并发数

同一时刻，对服务器有实际交互的请求数。

和网站在线用户数的关联：1000 个同时在线用户数，可以估计并发数在 5% 到 15% 之间，也就是同时并发数在 50~150 之间。

吞吐量

对单位时间内完成的工作量(请求)的量度

关系

系统吞吐量和系统并发数以及响应时间的关系：

理解为高速公路的通行状况：

吞吐量是每天通过收费站的车辆数目（可以换算成收费站收取的高速费），

并发数是高速公路上的正在行驶的车辆数目，

响应时间是车速。

车辆很少时，车速很快。但是收到的高速费也相应较少；随着高速公路上车辆数目的增多，车速略受影响，但是收到的高速费增加很快；

随着车辆的继续增加，车速变得越来越慢，高速公路越来越堵，收费不增反降；

如果车流量继续增加，超过某个极限后，任务偶然因素都会导致高速全部瘫痪，车走不动，当然后也收不着，而高速公路成了停车场（资源耗尽）。

常用的性能优化手段

避免过早优化

不应该把大量的时间耗费在小的性能改进上，过早考虑优化是所有噩梦的根源。

所以，我们应该编写清晰，直接，易读和易理解的代码，真正的优化应该留到以后，等到性能分析表明优化措施有巨大的收益时再进行。
但是过早优化，不表示我们就可以随便写代码，还是需要注重编写高效优雅的代码。

进行系统性能测试

所有的性能调优，都应该建立在性能测试的基础上，直觉很重要，但是要用数据说话，可以推测，但是要通过测试求证。

寻找系统瓶颈，分而治之，逐步优化

性能测试后，对整个请求经历的各个环节进行分析，排查出现性能瓶颈的地方，定位问题，分析影响性能的主要因素是什么？内存、磁盘 IO、网络、CPU，还是代码问题？架构设计不足？或者确实是系统资源不足？

前端优化常用手段

浏览器/App

减少请求数

合并 CSS, Js, 图片，
生产服务器提供的 all 的 js 文件
http 中的 keep-alive (http1.1 中默认开启) 包括 nginx

使用客户端缓冲

静态资源文件（css、图标等）缓存在浏览器中，有关的属性 Cache-Control（相对时间）和 Expires
如果文件发生了变化，需要更新，则通过改变文件名来解决。

启用压缩

浏览器(zip), 压缩率 80%以上。

减少网络传输量，但会给浏览器和服务器带来性能的压力，需要权衡使用。

资源文件加载顺序

css 放在页面最上面，js 放在最下面。这样页面的体验才会比较好。

浏览器会加载完 CSS 才会对页面进行渲染

JS 只要加载后就会立刻执行。（有些 JS 可能执行时间比较长）

减少 Cookie 传输

cookie 包含在每次的请求和响应中，因此哪些数据写入 cookie 需要慎重考虑（静态资源不需要放入 cookie）

友好的提示（非技术手段）

有时候在前端给用户一个提示，就能收到良好的效果。毕竟用户需要的是不要不理他。

CDN 加速

CDN，又称内容分发网络，本质是一个缓存，而且是将数据缓存在用户最近的地方。无法自行实现 CDN 的时候，可以根据经济实力考虑商用 CDN 服务。

反向代理缓存

将静态资源文件缓存在反向代理服务器上，一般是 Nginx。

WEB 组件分离

将 js, css 和图片文件放在不同的域名下。可以提高浏览器在下载 web 组件的并发数。因为浏览器在下载同一个域名的数据存在并发数限制。

应用服务性能优化

缓存

网站性能优化第一定律：优先考虑使用缓存优化性能

优先原则：缓存离用户越近越好

缓存的基本原理和本质

缓存是将数据存在访问速度较高的介质中。可以减少数据访问的时间，同时避免重复计算。

合理使用缓存的准则

频繁修改的数据，尽量不要缓存，读写比 2:1 以上才有缓存的价值。

缓存一定是热点数据。

应用需要容忍一定时间的数据不一致。

缓存可用性问题，一般通过热备或者集群来解决。

分布式缓存与一致性哈希

以集群的方式提供缓存服务，有两种实现：

1、需要更新同步的分布式缓存，所有的服务器保存相同的缓存数据，带来的问题就是，缓存的数据量受限制，其次，数据要在所有的机器上同步，代价很大。

2、每台机器只缓存一部分数据，然后通过一定的算法选择缓存服务器。常见的余数 hash 算法存在当有服务器上下线的时候，大量缓存数据重建的问题。所以提出了一致性哈希算法。

一致性哈希：

1. 首先求出服务器（节点）的哈希值，并将其配置到 $0 \sim 2$ 的 32 次方的圆（continuum）上。
2. 然后采用同样的方法求出存储数据的键的哈希值，并映射到相同的圆上。
3. 然后从数据映射到的位置开始顺时针查找，将数据保存到找到的第一个服务器上。如果超过 2^{32} 仍然找不到服务器，就会保存到第一台服务器上。

一致性哈希算法对于节点的增减都只需重定位环空间中的一小部分数据，具有较好的容错性和可扩展性。

数据倾斜：

一致性哈希算法在服务节点太少时，容易因为节点分布不均匀而造成数据倾斜问题，此时必然造成大量数据集中到 Node A 上，而只有极少量会定位到 Node B 上。为了解决这种数据倾斜问题，一致性哈希算法引入了虚拟节点机制，即对每一个服务节点计算多个哈希，每个计算结果位置都放置一个此服务节点，称为虚拟节点。具体做法可以在服务器 ip 或主机名的后面增加编号来实现。例如，可以为每台服务器计算三个虚拟节点，于是可以分别计算“Node A#1”、“Node A#2”、“Node A#3”、“Node B#1”、“Node B#2”、“Node B#3”的哈希值，于是形成六个虚拟节点：同时数据定位算法不变，只是多了一步虚拟节点到实际节点的映射，例如定位到“Node A#1”、“Node A#2”、“Node A#3”三个虚拟节点的数据均定位到 Node A 上。这样就解决了服务节点少时数据倾斜的问题。在实际应用中，通常将虚拟节点数设置为 32 甚至更大，因此即使很少的服务节点也能做到相对均匀的数据分布。

集群

可以很好的将用户的请求分配到多个机器处理，对总体性能有很大的提升

异步

同步和异步，阻塞和非阻塞

同步和异步关注的是结果消息的通信机制

同步：同步的意思就是调用方需要主动等待结果的返回

异步：异步的意思就是不需要主动等待结果的返回，而是通过其他手段比如，状态通知，回调函数等。

阻塞和非阻塞主要关注的是等待结果返回调用方的状态

阻塞：是指结果返回之前，当前线程被挂起，不做任何事

非阻塞：是指结果在返回之前，线程可以做一些其他事，不会被挂起。

1. 同步阻塞：同步阻塞基本也是编程中最常见的模型，打个比方你去商店买衣服，你去了之后发现衣服卖完了，那你就一直在店里面一直等，期间不做任何事(包

括看手机), 等着商家进货, 直到有货为止, 这个效率很低。jdk 里的 BIO 就属于 同步阻塞

2.同步非阻塞:同步非阻塞在编程中可以抽象为一个轮询模式, 你去了商店之后, 发现衣服卖完了, 这个时候不需要傻傻的等着, 你可以去其他地方比如奶茶店, 买杯水, 但是你还是需要时不时的去商店问老板新衣服到了吗。jdk 里的 NIO 就属于 同步非阻塞

3.异步阻塞:异步阻塞这个编程里面用的较少, 有点类似你写了个线程池,submit 然后马上 future.get(), 这样线程其实还是挂起的。有点像你去商店买衣服, 这个时候发现衣服没有了, 这个时候你就给老板留个电话, 说衣服到了就给我打电话, 然后你就守着这个电话, 一直等着他响什么事也不做。这样感觉的确有点傻, 所以这个模式用得比较少。

4.异步非阻塞:好比你去商店买衣服, 衣服没了, 你只需要给老板说这是我的电话, 衣服到了就打。然后你就随心所欲的去玩, 也不用操心衣服什么时候到, 衣服一到, 电话一响就可以去买衣服了。jdk 里的 AIO 就属于异步

常见异步的手段

Servlet 异步

servlet3 中才有, 支持的 web 容器在 tomcat7 和 jetty8 以后。

多线程

消息队列

程序

代码级别

一个应用的性能归根结底取决于代码是如何编写的。



选择合适的数据结构

```
public static void main(String[] args) {  
    List<Object> list = new LinkedList<>();  
    int i=0;  
    while(true) {  
        i++;  
        if(i%10000==0) System.out.println("i="+i);  
        list.add(new Object());  
    }  
}
```

选择 ArrayList 和 LinkedList 对我们的程序性能影响很大，为什么？因为 ArrayList 内部是数组实现，存在着不停的扩容和数据复制。

选择更优的算法

举个例子，如何判断一个数是否为 n 的多少次方

*类说明：选择更优的算法

*/

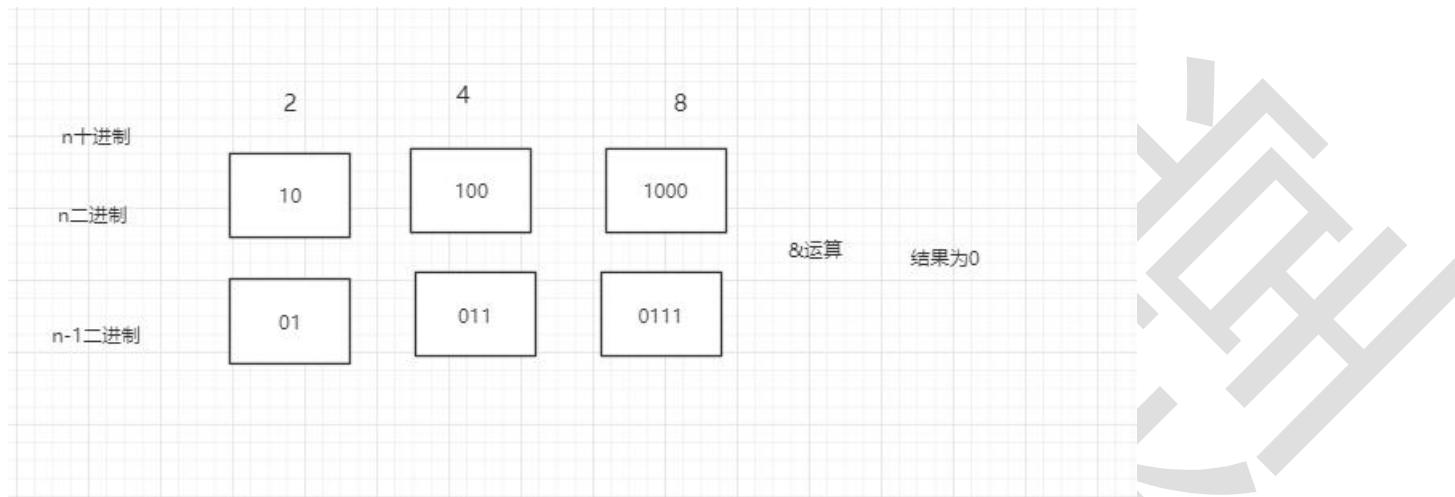
```
public class BetterAlg {  
    //如何判断一个数是否为n的多少次方  
    public static void main(String[] args) throws Exception{  
        int n =2;
```

```
Scanner scanner=new Scanner(System.in);
while(scanner.hasNext){//控制台输入
    int input =scanner.nextInt();

    while (true){
        if(input ==n){
            System.out.println("是("+n+")的次方");
            break;
        }
        if(input%2 !=0){
            System.out.println("不是("+n+")的次方");
            break;
        }else{
            input = input/2;
        }
    }

    //    if((input&(input-1)) ==0){
    //        System.out.println("是("+n+")的次方");
    //    }else{
    //        System.out.println("不是("+n+")的次方");
    //    }
    }
}
```





编写更少的代码

同样正确的程序，小程序比大程序要快，这点无关乎编程语言。

并发编程

资源的复用

目的是减少开销很大的系统资源的创建和销毁，比如数据库连接，网络通信连接，线程资源等等。

单例模式

Spring 中的 bean

池化技术

存储性能优化

尽量使用 SSD

定时清理数据或者按数据的性质分开存放

结果集处理

用 setFetchSize 控制 jdbc 每次从数据库中返回多少数据。

总结：

调优是个很复杂、很细致的过程，要根据实际情况调整，不同的机器、不同的应用、不同的性能要求调优的手段都是不同的。也没有一个放之四海而皆准的配置或者公式。King 老师也无法告诉大家全部与性能相关的知识，即使是 jvm 参数也是如此，再比如说性能有关的操作系统工具，和操作系统本身相关的所谓大页机制，都需要大家平时去积累，去观察，去实践。

King 老师在这个专题上告诉大家的除了各种 java 虚拟机基础知识、内部原理，也告诉大家一个性能优化的一个基本思路和着手的方向。

第七节：编写高效优雅 Java 程序

面向对象

01、构造器参数太多怎么办？

如果参数很多，会导致构造方法非常多，拓展性差，代码难编写，且难以看懂。

用 JavaBeans 模式，

get 和 set

一行构造编程多行代码实现，需要使用额外机制确保一致性和线程安全。

用 builder 模式，

1、5 个或者 5 个以上的成员变量

2、参数不多，但是在未来，参数会增加

Builder 模式：

属于对象的创建模式，一般有

- 1、抽象建造者：一般来说是个接口，包含 1) 建造方法，建造部件的方法（不止一个），2) 返回产品的方法
- 2、具体建造者
- 3、导演者，调用具体的建造者，创建产品对象
- 4、产品，需要建造的复杂对象

对于客户端，创建导演者和具体建造者，并把具体建造者交给导演者，然后由客户端通知导演者操纵建造者进行产品的创建。

在实际的应用过程中，有时会省略抽象建造者和导演者。

优势：如果当大多数参数是可选时，代码易于阅读和编写，比 JavaBean 更加安全。

02、不需要实例化的类应该构造器私有

如，一些工具类提供的都是静态方法，这些类是不应该提供具体的实例的。可以参考 JDK 中的 Arrays。

好处：防止使用者 new 出多个实例。

03、不要创建不必要的对象

1、避免无意中创建的对象，如自动装箱

```
public class Sum {  
    public static void main(String[] args) {  
        long start = System.currentTimeMillis();  
        Long sum = 0L; //对象  
        for(long i=0;i<Integer.MAX_VALUE;i++) {  
            sum = sum+i;  
            //new 20多亿的Long的实例  
        }  
  
        System.out.println("spend time:"+ (System.currentTimeMillis()-start)+"ms");  
    }  
}
```

可以在类的多个实例之间重用的成员变量，尽量使用 static。

性能对比。

但是，要记住，是不要创建**不必要的**对象，而不是不要创建对象。

对象池要谨慎使用，除非创建的对象是非常昂贵的操作，如数据库的连接，巨型对象等等。

04、避免使用终结方法

`finalizer` 方法， jdk 不能保证何时执行，也不能保证一定会执行。如果有确实要释放的资源应该用 `try/finally`。

05、使类和成员的可访问性最小化

模块对外部其他模块来说，隐藏其内部数据和其他实现细节——封装

编写程序和设计架构，最重要的目标之一就是模块之间的解耦。使类和成员的可访问性最小化无疑是有效的途径之一。
类似于微服务，

06、使可变性最小化

尽量使类不可变，不可变的类比可变的类更加易于设计、实现和使用，而且更不容易出错，更安全。

常用的手段：

不提供任何可以修改对象状态的方法；

使所有的域都是 `final` 的。

使所有的域都是私有的。

使用写时复制机制。并发编程中已讲。

07、复合优先于继承

继承容易破坏封装性，而且会使子类的实现依赖于父类。

复合则是在类中增加一个私有域，引用类的一个实例，这样的话就避免了依赖类的具体实现。

只有在子类确实是父类的一个子类型时，才比较适合用继承。
继承需要开发者对父类的结构有一定了解。
实际使用，如果肯定是父类的子类，使用继承，如果不很肯定，使用复合。

08、接口优于抽象类

接口只有方法申明，抽象类可以写方法的实现。
java 是个单继承的（不能继承多个抽象类），但是类允许实现多个接口。
所以当发生业务变化时，新增接口，实现接口只需要新增接口即可。但是抽象类有可能导致不需要变化的类也不得不实现新增的业务方法。
JDK 源码中常用的一种设计方法：定义一个接口，声明一个**抽象的骨架类**实现接口，骨架类实现通用的方法，而实际的业务类可以同时实现接口又继承骨架类，也可以只实现接口。
如 HashSet 实现了 **implements** Set 接口 但是又 **extends** 类 AbstractSet，而 AbstractSet 本身也实现了 Set 接口。其他如 Map，List 都是这样的设计的。

方法

09、可变参数要谨慎使用

可变参数是允许传 0 个参数的
如果是参数个数在 1~多个之间的時候，要做单独的业务控制。
具体代码不优雅。

10、返回零长度的数组或集合，不要返回 null

方法的结果返回 null，会导致调用方的要单独处理为 null 的情况。返回零长度，调用方可以统一处理，如使用 foreach 即可。
JDK 中也为我们提供了 `Collections.EMPTY_LIST` 这样的零长度集合

11、优先使用标准的异常

要尽量追求代码的重用，同时减少类加载的数目，提高类装载的性能。

常用的异常：

`IllegalArgumentException` -- 调用者传递的参数不合适

`IllegalStateException` -- 接收的对象状态不对，

`NullPointerException`

`UnsupportedOperationException` - 不支持的操作

通用程序设计

12、用枚举代替 int 常量

声明的一个枚举本质就是一个类，每个具体的枚举值就是这个枚举类的实例。

1. 使用常量容易在写代码时写错
 2. 使用常量如果要使用描述时比较麻烦
 3. 其他类使用常量时，类编译时会把常量值直接写到字节码中，如果常量值有变化，所有相关的类需要重新编译，否则会不可预料的错误
- 枚举高级：

枚举和行为绑定

所谓枚举的本质就是一个类，而枚举中定义的每一个具体的枚举类型其实就是这个枚举类的一个实例。

策略枚举：

13、将局部变量的作用域最小化

- 1、在第一次使用的地方进行声明
- 2、局部变量都是要自行初始化，初始化条件不满足，就不要声明

最小化的好处，减小局部变量表的大小，提高性能；同时避免局部变量过早声明导致不正确的使用。

14、精确计算，避免使用 float 和 double

float 和 double 在 JVM 存储的时候，有部分要做整数位，有部分要做小数位，所以存在精度上的问题
可以使用 int 或者 long 以及 BigDecimal

15、当心字符串连接的性能

参考代码 com.jvm.ch07.p15.Test。

在存在大量字符串拼接或者大型字符串拼接的时候，尽量使用 StringBuilder 和 StringBuffer

16、控制方法的大小

