全面理解 JVM 虚拟机

-- 楼兰

JVM 虚拟机，这是一个Java 程序员一直以来熟悉但是又陌生的神秘东东。他是夹在 Java 代码与操作系统之间的一层神秘空间。这一次，楼兰就来带大家全面梳理一下这个神秘的 JVM 虚拟机，作为我们后续深入 JVM 细节的一个预热工作。

为什么要学JVM?

首先：面试需要。面试题层出不穷，难道每次面试都靠背几百上千条面试八股?

其次：基础决定上层建筑。自己写的代码都不知道是怎么回事，怎么可能写出靠谱的系统?

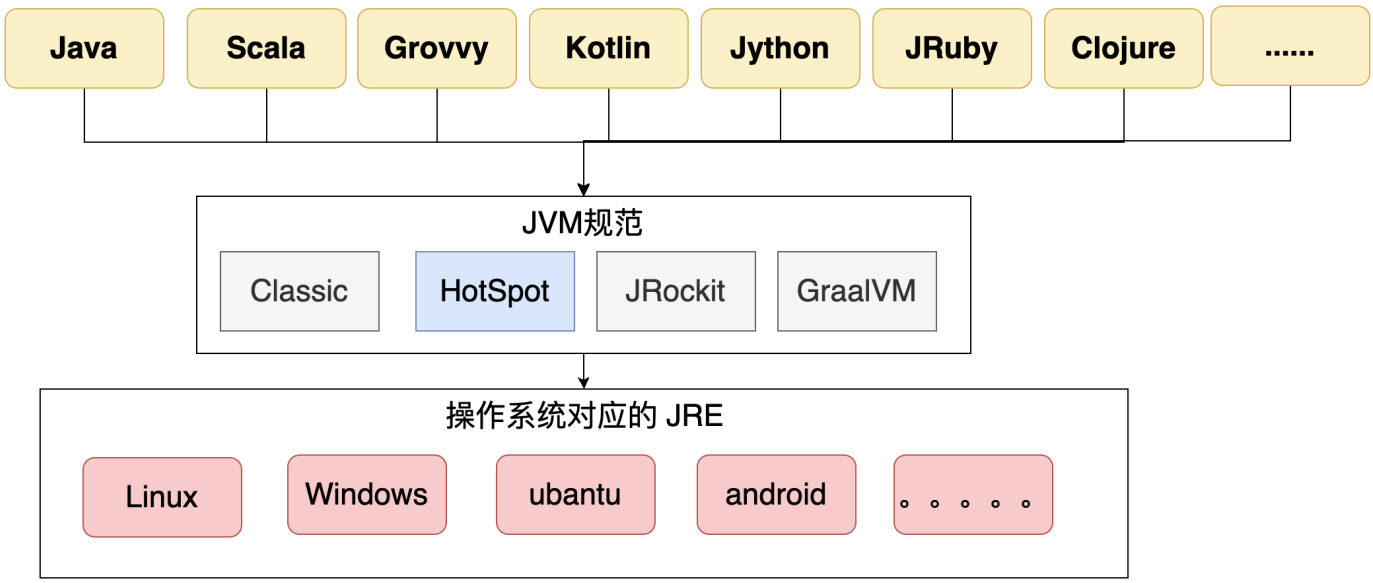
然后：学习JVM也是进行JVM调优的基础。写的代码放到线上要如何运行？要配多少内存？4G够不够？线上环境出问题，服务崩溃了，怎么快速定位？怎么解决问题?

总之，学不学JVM，是能自主解决问题的一流程序员与跟着别人做CRUD的二流程序员的分水岭！二流程序员会觉得学JVM无关紧要，反正开发也用不上。做开发我只要学各种框架就行了。而一流程序员都在尽自己能力把JVM每个底层逻辑整理成自己的知识体系。

一、JVM 后面要学些什么-五分钟

Java发展至今，已经远不是一种语言，而是一个标准。只要能够写出满足JVM规范的class文件，就可以丢到JVM虚拟机执行。通过JVM虚拟机，屏蔽了上层各种开发语言的差距，同时也屏蔽了下层各种操作系统的区别。一次编写，多次执行。

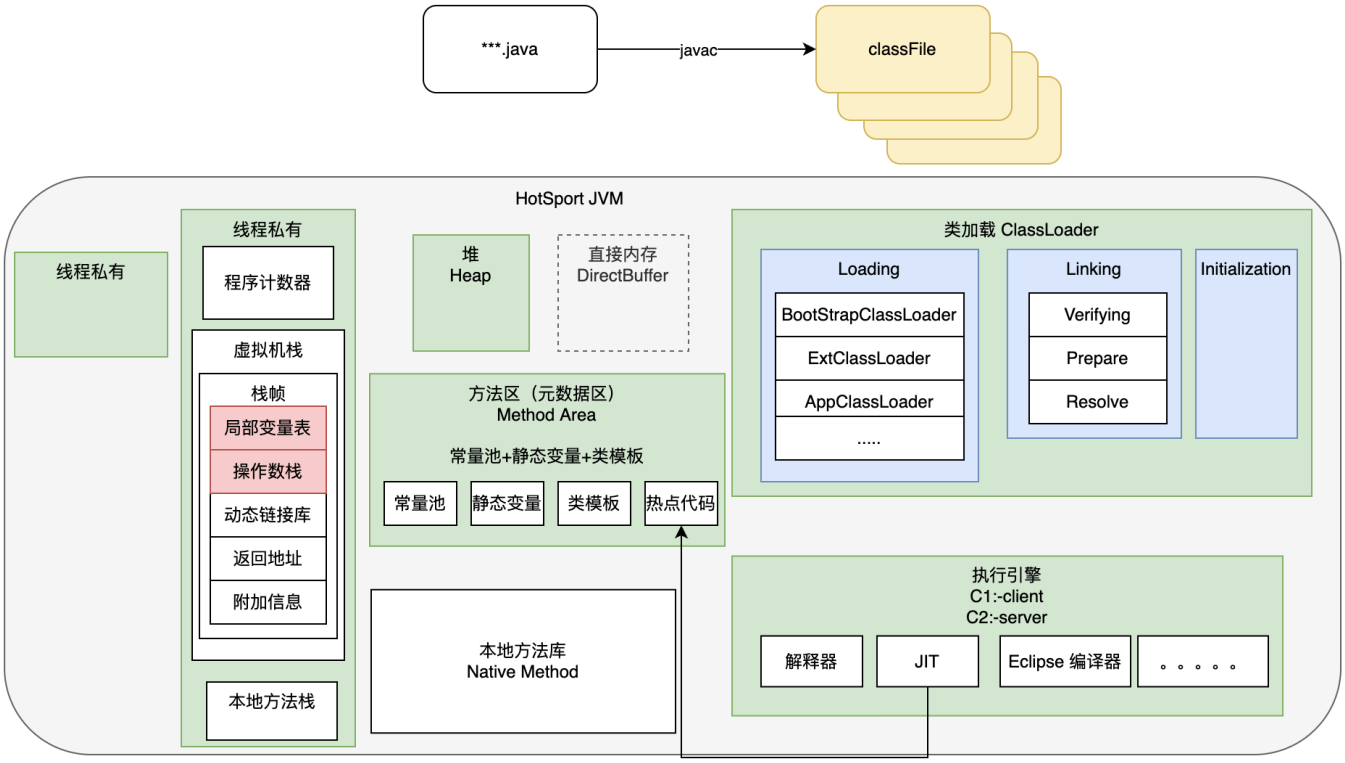
> 你有没有试过在一个项目里同时用Java和Scala进行开发?



JVM也有很多具体的实现版本，现在最主流的是Oracle官方的HotSpot虚拟机。这也是我们课程的重点。

```
# java -version
java version "1.8.0_391"
Java(TM) SE Runtime Environment (build 1.8.0_391-b13)
Java HotSpot(TM) 64-Bit Server VM (build 25.391-b13, mixed mode)
```

一个java文件，整体的执行过程整理如下图：



其中细节非常多，也很容易让人学得枯燥。今天主要是以实战的方式整体理解一下这些核心模块。在后续的课程中，会详细介绍每个部分的细节。

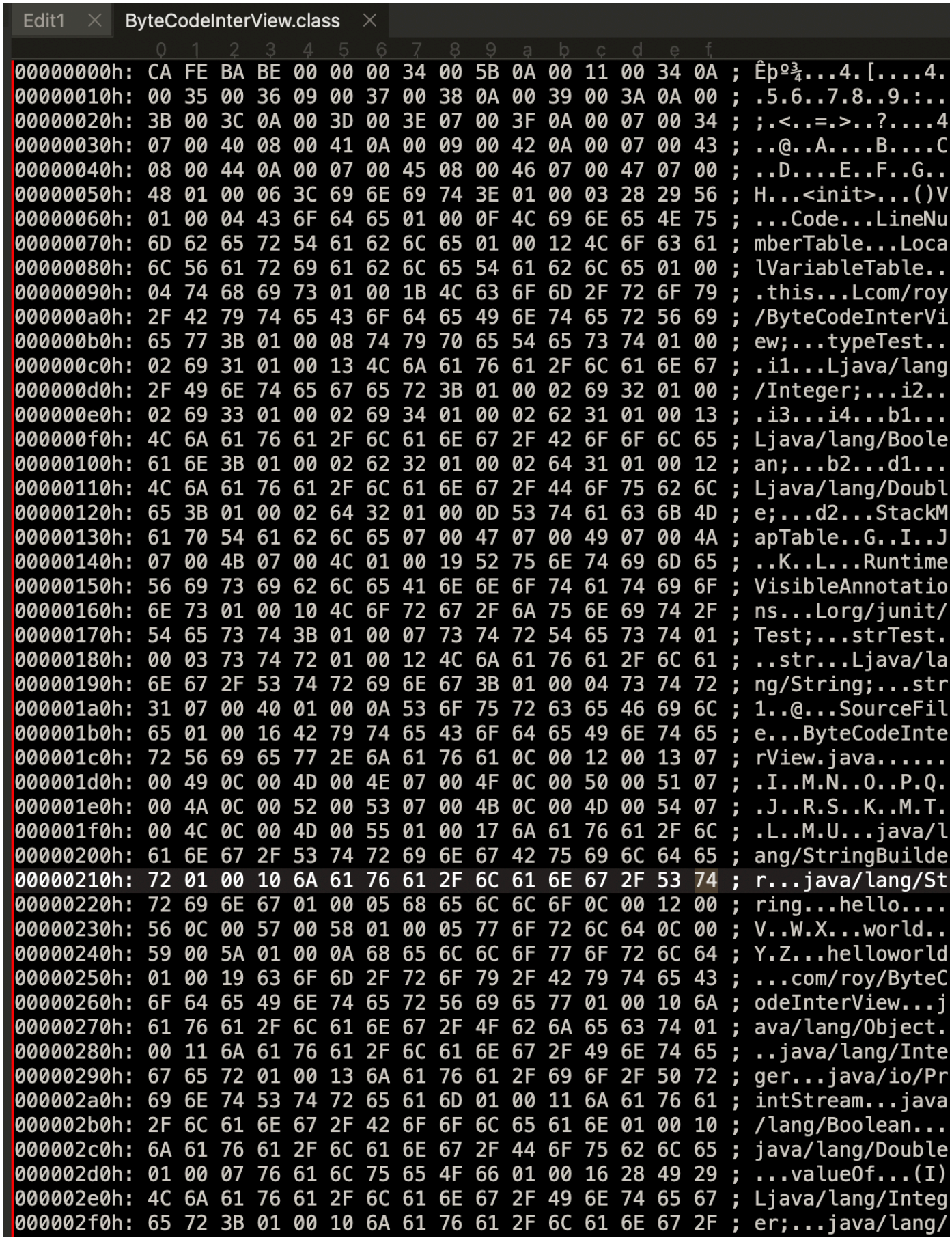
二、Class 文件规范-四十分钟

1、Class文件结构

实际上，我们需要了解的是，Java 官方实际上只定义了JVM的一种执行规范，也就是class文件的组织规范。理论上，只要你能够写出一个符合标准的class文件，就可以丢到 JVM 中执行。至于这个class文件是怎么来的，JVM 虚拟机是不管的。这也是 JVM 支持多语言的基础。

这个规范到底是什么样子呢？当然，你可以直接去看 Oracle 的官方文档。JDK8 的文档地址：https://docs.oracle.com/javase/specs/jvms/se8/html/index.html 。后面也会有课程带你详细分析每一个字节。这里，我们只抽取主体内容。

首先，我们要知道，class文件本质是一个二进制文件，虽然不能直接用文本的方式阅读，但是我们是可以用一些文本工具打开看看的。比如，对于一个ByteCodeInterView.class文件，可以用 UltraEdit 工具打开一个class文件，看到的内容部分是这样的：



中间这一部分就是他的二进制内容。当然这是十六进制的表达。空格隔开的部分代表了 8 个bit，而每一位代表的是 4 个 bit字节，也就是一个十六进制的数字。例如 第一个字母 C 就表示十六进制的 12，二进制是 1100。而所有的class文件，都必须以十六进制的 CAFEBABE 开头，这就是 JVM 规范的一部分。这也解释了 Java 这个词的由来，到底是一种咖啡，还是爪哇岛。

后面的部分就比较复杂了，没法直接看。这时我们就需要用一些工具来看了。这样的工具很多。JDK 自己就提供了一个 javap 指令可以直接来看一些class文件。例如可以用 javap -v ByteCodeInterView.class 查看到这个class文件的详细信息。

当然，这样还是不够直观。我们可以在 IDEA 里添加一个 ByteCodeView 插件来更直观的查看一个 ClassFile 的内容。看到的大概内容是这样的：

> 插件安装以及使用，这里就不多说了，相信对各位都是小菜一碟。

可以看到，一个class文件的大致组成部分。然后再结合官方的文档，或许能够让你开始对class文件有一个大致的感觉。

A class file consists of a single ClassFile structure:

```
ClassFile {
    u4             magic;
    u2             minor_version;
    u2             major_version;
    u2             constant_pool_count;
    cp_info        constant_pool[constant_pool_count-1];
    u2             access_flags;
    u2             this_class;
    u2             super_class;
    u2             interfaces_count;
    u2             interfaces[interfaces_count];
    u2             fields_count;
    field_info     fields[fields_count];
    u2             methods_count;
    method_info    methods[methods_count];
    u2             attributes_count;
    attribute_info attributes[attributes_count];
}
```

例如，前面u4表示四个字节是magic魔数，而这个魔数就是不讲道理的 CAFEBABE 。

而后面的两个u2，表示两个字节的版本号。例如我们用 JDK8 看我们之前的class文件，minor_version就是 00 00，major_version就是 00 34。换成二进制就是 52。52.0 这就是 JVM 给 JDK8 分配的版本号。这两个版本号就表示当前这个class文件是由JDK8编译出来的。后续就只能用8以前版本的JVM执行。这就是JDK版本向前兼容的基础。

例如，如果你尝试用JDK8去引用Spring 6或者SpringBoot 3以后的新版本，就会报错。就是因为Spring 6和SpringBoot 3发布出来的class文件，是用JDK17编译的，版本号是61。JDK8是无法执行的。

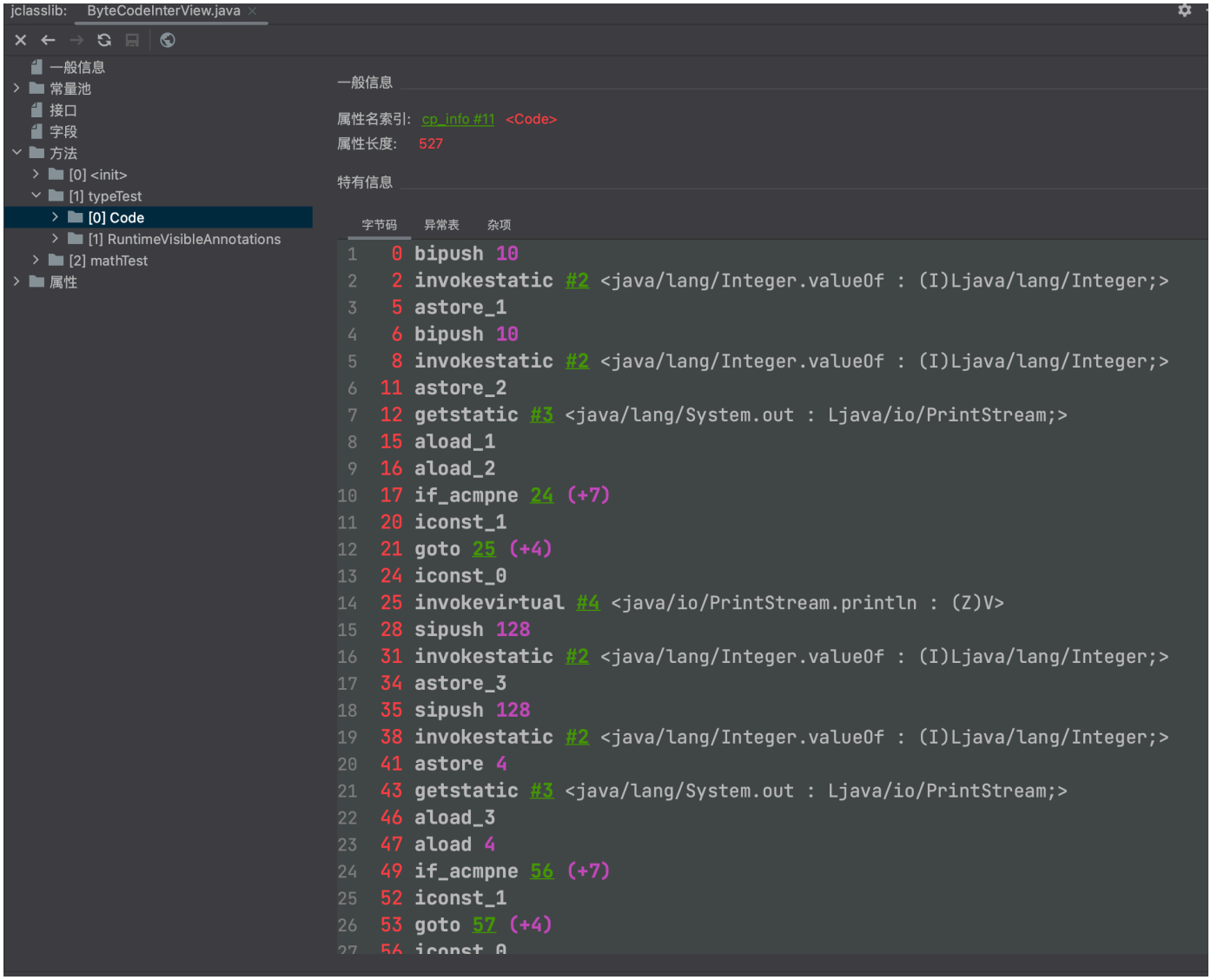
接下来，class文件的整体布局就比较明显了。其中常量池是最复杂的部分，包含了表示这个class文件所需要的几乎所有常量。比如接口名字，方法名字等等。而后面的几个部分，比如方法，接口等都是引用常量池中的各种变量。

> 详细细节，后面VIP课程会带你全程手撕。一个一个字节解读。
>
> 在这里，你可能会注意到一个不太起眼的小细节，常量池中的索引结构是从 1 开始的，而不是像 Java 中其他地方一样，从 0 开始。这样做的目的在于，如果后面某些指向常量池的索引值的数据在特定情况下需要表达“不引用任何一个常量池项目”的含义，就可以把索引值设定为 0 表示。

尽管 Java 发展了很多年，JDK 版本也不断更新，但是 Class 文件的结构、字节码指令的语义和数量几乎没有发生过变动，所有对 Class 文件格式的改进，都集中在方法标志、属性表这些设计上原本就是可扩展的数据结构中添加新内容。

2 、理解字节码指令

而这其中，我们重点关注的是方法，也就是class文件是如何记录我们写的这些关键代码的。例如在ByteCodeInterView中的typeTest这个方法，在class文件中就是这样记录的：



这里每一样就是一个字节码指令。 JVM 虚拟机的字节码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)以及跟随气候的零至多个代表此操作所需要的参数(称为操作数，Operand)构成。其中操作数，可以是一个具体的参数，也可以是一个指向class文件常量池的符号引用，也可以是一个指向运行时常量池中的一个方法。比如第 0 行 bipush 10，操作码就是 bipush，操作数就是 10。这个指令就占据了第 0 行和第 1 行两行。而有些操作码，如 astore_1，就只有一个操作码，没有操作数。

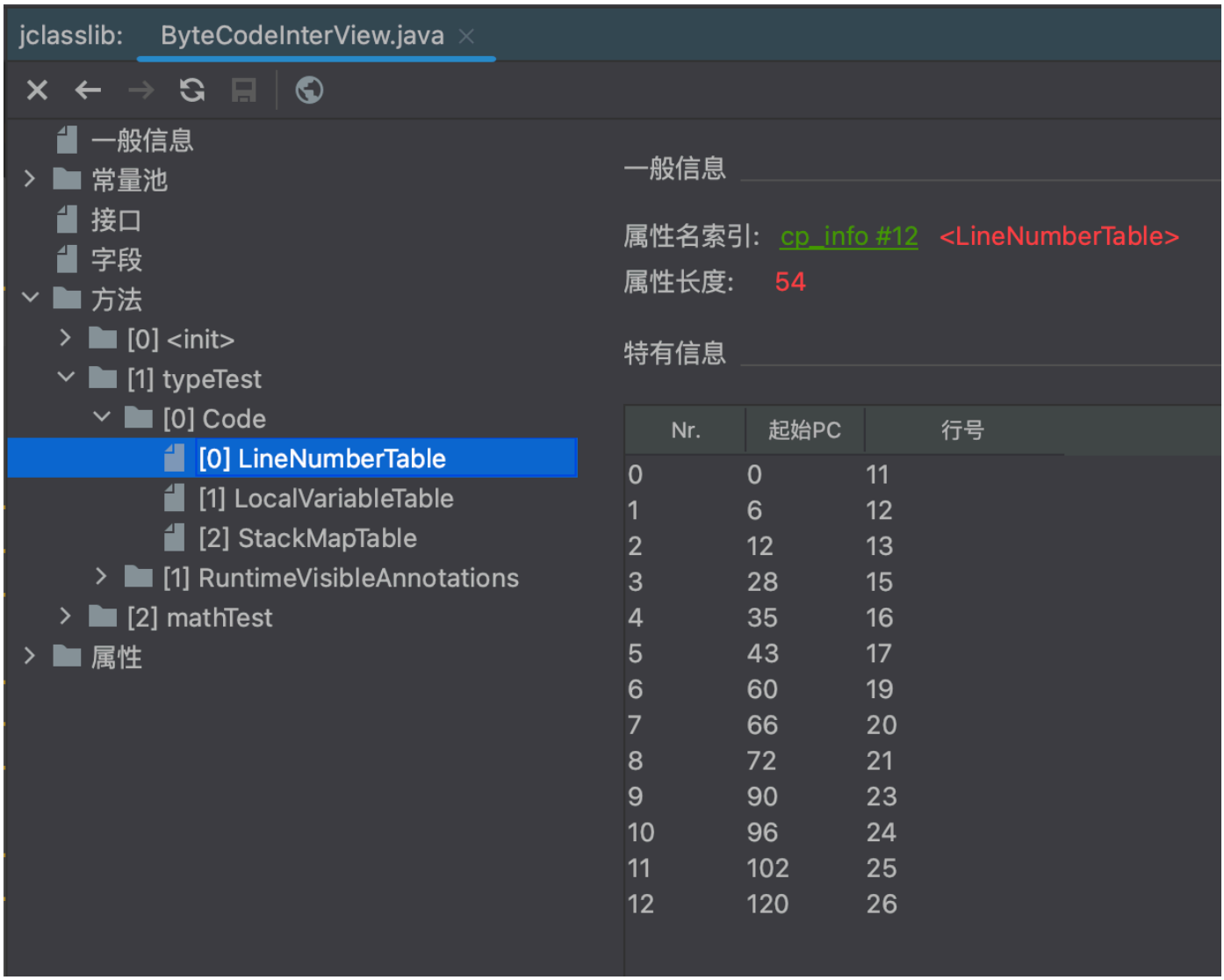
Java 虚拟机中的操作码的长度只有一个字节(能表示的数据是0～255)，这意味着 JVM 指令集的操作码总数不超过 256 条。这些指令相比于庞大的操作系统来说，已经是非常小的了。另外其中还有很多差不多的。 比如 aload_1，aload_2 这些，明显就是同一类的指令。

这些字节码指令，在不同JDK 版本中会稍有不同。具体可以参考 Oracle 官方文档。JDK 文档地址： https://docs.oracle.com/javase/specs/jvms/se8/html/index.html

如果不考虑异常的话，那么 JVM 虚拟机执行代码的逻辑就应该是这样：

```
do{
   从程序计数器中读取 PC 寄存器的值 + 1;
   根据 PC 寄存器指示的位置，从字节码流中读取一个字节的操作码;
   if(字节码存在操作数) 从字节码流中读取对应字节的操作数;
   执行操作码所定义的操作;
}while(字节码流长度>0)
```

这些字节码指令你看不懂？没关系，至少现在，你可以知道你写的代码在 Class 文件当中是怎么记录的了。另外，如果你还想更仔细一点的分辨你的每一样代码都对应哪些指令，那么在这个工具中还提供了一个 LineNumberTable，会告诉你这些指令与代码的对应关系。



> 起始 PC 就是这些指令的字节码指令的行数，行号则对应 Java 代码中的行数。
>
> 实际上，Java 程序在遇到异常时给出的堆栈信息，就是通过这些数据来反馈报错行数的。

4 、字节码指令解读案例

这些字节码指令，我们后面有VIP课带你详细解读。但是，在这之前，你可能会有一个困惑。这些字节码指令是给机器看的，我要去学习这些字节码指令有什么用？接下来我们就来详细分析一个小案例，来看看了解字节码指令的必要性。

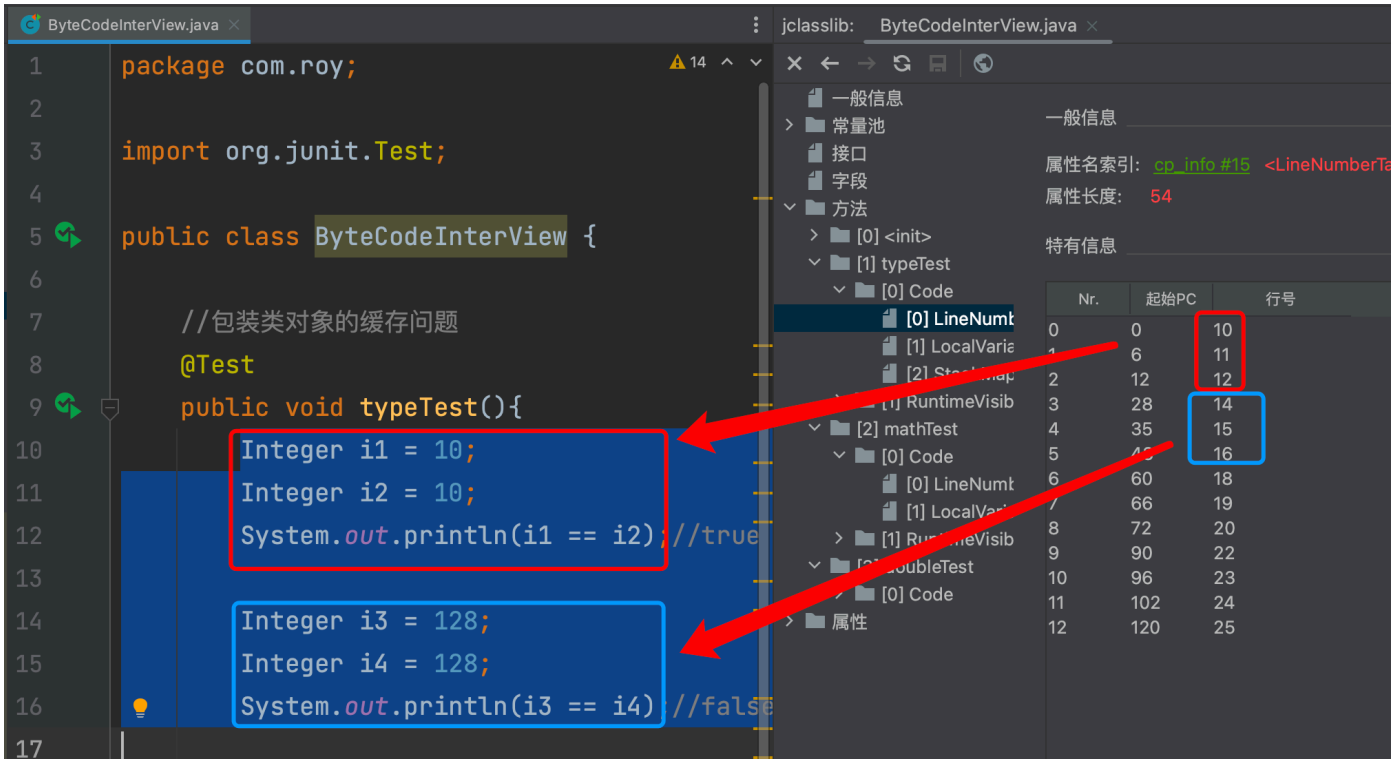
在ByteCodeInterView中，我们写了一个typeTest方法。我们重点来分析其中最容易让人产生困惑的几行代码。

```
        Integer i1 = 10;
        Integer i2 = 10;
        System.out.println(i1 == i2);//true

        Integer i3 = 128;
        Integer i4 = 128;
        System.out.println(i3 == i4);//false
```

执行结果注释在了后面。这些莫名其妙的true和false是怎么蹦出来的？如果你之前恰巧刷到过这样的面试题，或许你会记得这是因为JAVA的基础类型装箱机制引起的小误会。但是如果你没背过呢？或者JAVA中还有很多类似的让人抓狂的面试题，你也一个一个去背吗？那要怎么彻底了解这一类问题呢？你最终还是要学会自己看懂这些字节码指令。

首先，我们可以从LineNumberTable 中获取到这几行代码对应的字节码指令：



以前面三行为例，三行代码对应的 PC 指令就是从 0 到 12 号这几条指令。把指令摘抄下来是这样的：

```
 0 bipush 10
 2 invokestatic #2 <java/lang/Integer.valueOf : (I)Ljava/lang/Integer;>
 5 astore_1
 6 bipush 10
 8 invokestatic #2 <java/lang/Integer.valueOf : (I)Ljava/lang/Integer;>
11 astore_2
12 getstatic #3 <java/lang/System.out : Ljava/io/PrintStream;>
```

可以看到，在执行astore指令往局部变量表中设置值之前，都调用了一次Integer.valueOf方法。

```
    Returns an Integer instance representing the specified int value. If a new Integer instance is not
    required, this method should generally be used in preference to the constructor Integer(int), as
    this method is likely to yield significantly better space and time performance by caching frequently
    requested values. This method will always cache values in the range -128 to 127, inclusive, and may
    cache other values outside of this range.
    Params: i – an int value.
    Returns: an Integer instance representing i.
    Since:   1.5
public static Integer valueOf(int i) {
    if (i >= IntegerCache.low && i <= IntegerCache.high)
        return IntegerCache.cache[i + (-IntegerCache.low)];
    return new Integer(i);
}
```

而在这个方法中，对于[-128,127]范围内常用的数字，实际上是构建了缓存的。每次都从缓存中获取一个相同的值，他们的内存地址当然就是相等的了。这些童年梦魇，是不是在这个过程中找到了终极答案?

> 实际上，你甚至可以使用反射来修改这个内部的 IntegerCache 缓存，从而让 Integer 的值发生紊乱。你有试过这样的骚操作吗?

另外，在这个过程中，我们也看到了在JVM中，是通过一个invokestatic指令调用一个静态方法。实际上JDK中还有以下几个跟方法调用相关的字节码指令:

- 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:用于调用类静态方法(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。前面四条调用指令的分派逻辑都固定在 Java 虚拟机内部，用户无法改变，而invokedynamic指令的分派逻辑是由用户所设定的引导方法决定的。 Java 从诞生到现在，只增加过一条指令，就是invokedynamic。自 JDK7 支持并开始进行改进，这也是为 JDK8 实现Lambda表达式而做的技术储备。

方法调用指令与数据类型无关，而方法返回指令是根据返回值的类型区分的。包括ireturn(返回值是boolean ,byte,char,short,int)，lreturn, freturn, return , 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、类和接口的类初始化方法使用。。

> 面试题：Java 当中的静态方法可以被子类重写吗?
>
> 普通答案：不能吧，因为没见过这么用的。吧啦吧啦吧啦。。。。。我还是做个例子测测吧。
>
> 高手答案：不能。因为在 JVM 中，调用方法提供了几个不同的字节码指令。invokcvirtual 调用对象的虚方法(也就是可被子类重写的这些方法)。invokespecial 根据编译时类型来调用实例方法，比如静态代码块(通常对应字节码层面的cinit 方法)，构造方法(通常对应字节码层面的init方法)。invokestatic 调用类（静态）方法。invokcinterface 调用接口方法。

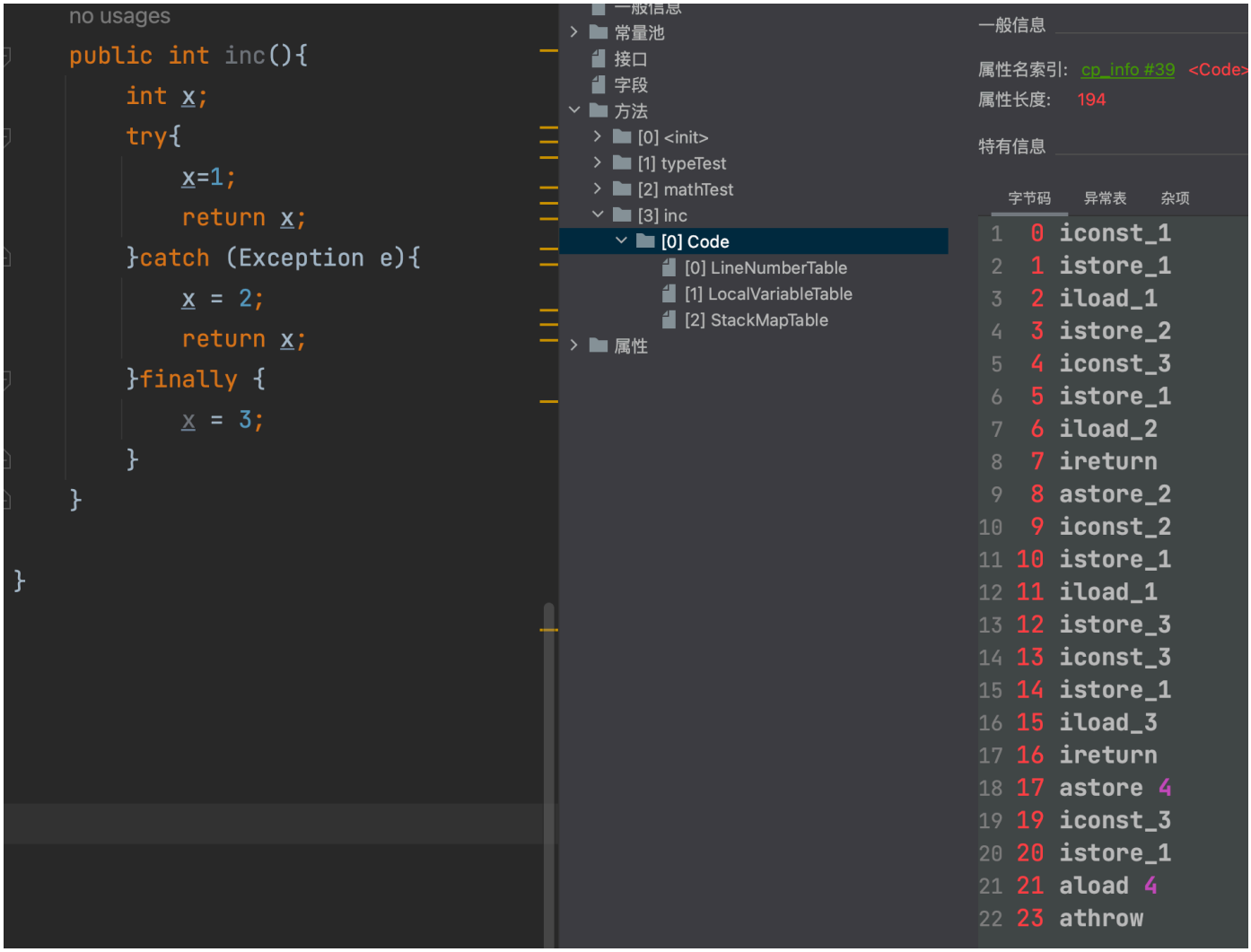
> 静态方法和可重写的方法他们的调用指令都是不一样的，那么肯定是无法重写静态方法的。

5 、深入字节码理解try-cache-finally的执行流程

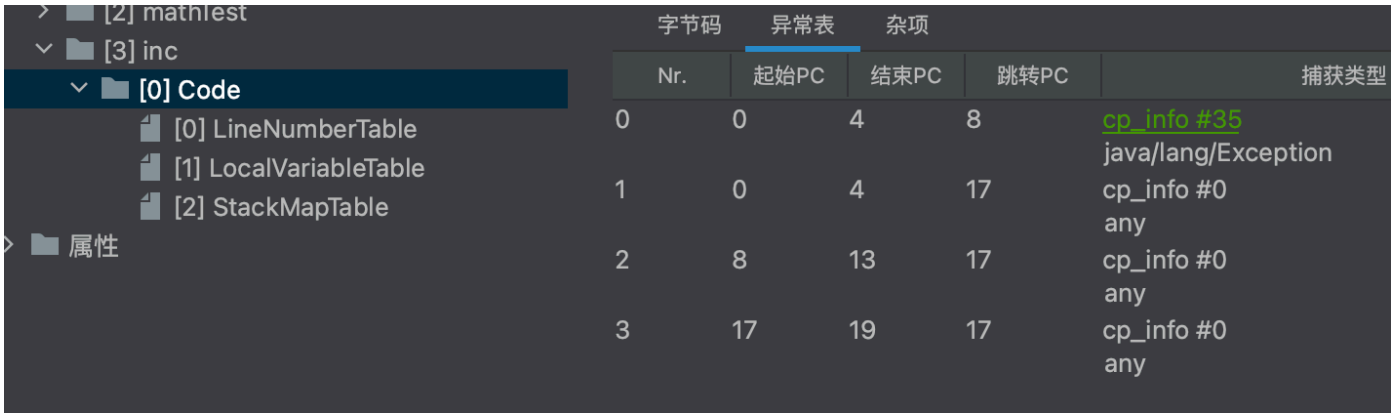
在之前解读字节码时，你们可能会注意到，在字节码指令的上面，有一个异常表的标签。这个异常表就是用来控制抛出异常的情况下的处理流程。我们用下面一个简单代码来做演示：

```
public int inc(){
        int x;
        try{
            x=1;
            return x;
        }catch (Exception e){
            x = 2;
            return x;
        }finally {
            x = 3;
        }
    }
```

这个方法编译出的字节码是这样的：



try-catch-finally的指令提现在哪里呢？这些就在旁边的异常表中。



其实，对照源代码，你大概也应该能够猜到这个异常表的表现形式。异常表中每一行代表一个执行逻辑的分支。表示 当字节码从《起始 PC》到《结束 PC》(不包含结束 PC)之间出现了类型为《捕获异常》或者其子类的异常时，就跳转到《跳转 PC》处进行处理。

可以看到，这里定义了三条明显的执行路径，分别是：

- 如果try语句块中出现了属于 Exception 或者其子类的异常，转到catch语句块处理。
- 如果try语句块中出现了不属于 Exception 或其子类的异常，转到finally语句块处理。
- 如果catch语句块中出现了任何异常，转到finally语句块处理。

补充知识点：字节码指令是如何工作的?

这部分内容后面会有课程，在这里，只作为课外补充，有兴趣可以提前了解一下。

要了解这些指令的作用，就不得不先了解一下 JVM 中两个重要的数据结构：局部变量表和操作数栈。

在 JVM 虚拟机中，会为每个线程构建一个线程私有的内存区域。其中包含的最重要的数据就是程序计数器和虚拟机栈。其中程序计数器主要是记录各个指令的执行进度，用于在 CPU 进行切换时可以还原计算结果。虚拟机栈中则包含了这个线程运行所需要的重要数据。

虚拟机栈是一个先进后出的栈结构，其中会为线程中每一个方法构建一个栈帧。而栈帧先进后出的特性也就对应了我们程序中每个方法的执行顺序。每个栈帧中包含四个部分，局部变量表，操作数栈，动态链接库、返回地址。

- 操作数栈是一个先进后出的栈结构，主要负责存储计算过程中的中间变量。操作数栈中的每一个元素都可以是包括long型和double在内的任意 Java 数据类型。
- 局部变量表可以认为是一个数组结构，主要负责存储计算结果。存放方法参数和方法内部定义的局部变量。以 Slot 为最小单位。
- 动态链接库主要存储一些指向运行时常量池的方法引用。每个栈帧中都会包含一个指向运行时常量池中该栈帧所属方法的应用，持有这个引用是为了支持方法动态调用过程中的动态链接。
- 返回地址存放调用当前方法的指令地址。一个方法有两种退出方式，一种是正常退出，一种是抛异常退出。如果方法正常退出，这个返回地址就记录下一条指令的地址。如果是抛出异常退出，返回地址就会通过异常表来确定。
- 附加信息主要存放一些 HotSpot 虚拟机实现时需要填入的一些补充信息。这部分信息不在 JVM 规范要求之内，由各种虚拟机实现自行决定。

其中最为重要的就是操作数栈和局部变量表了。例如，对于初学者最头疼的++操作，下面的 mathTest 方法

```
    public int mathTest(){
        int k = 1 ;
        k = k++;
        return k;
    }
```

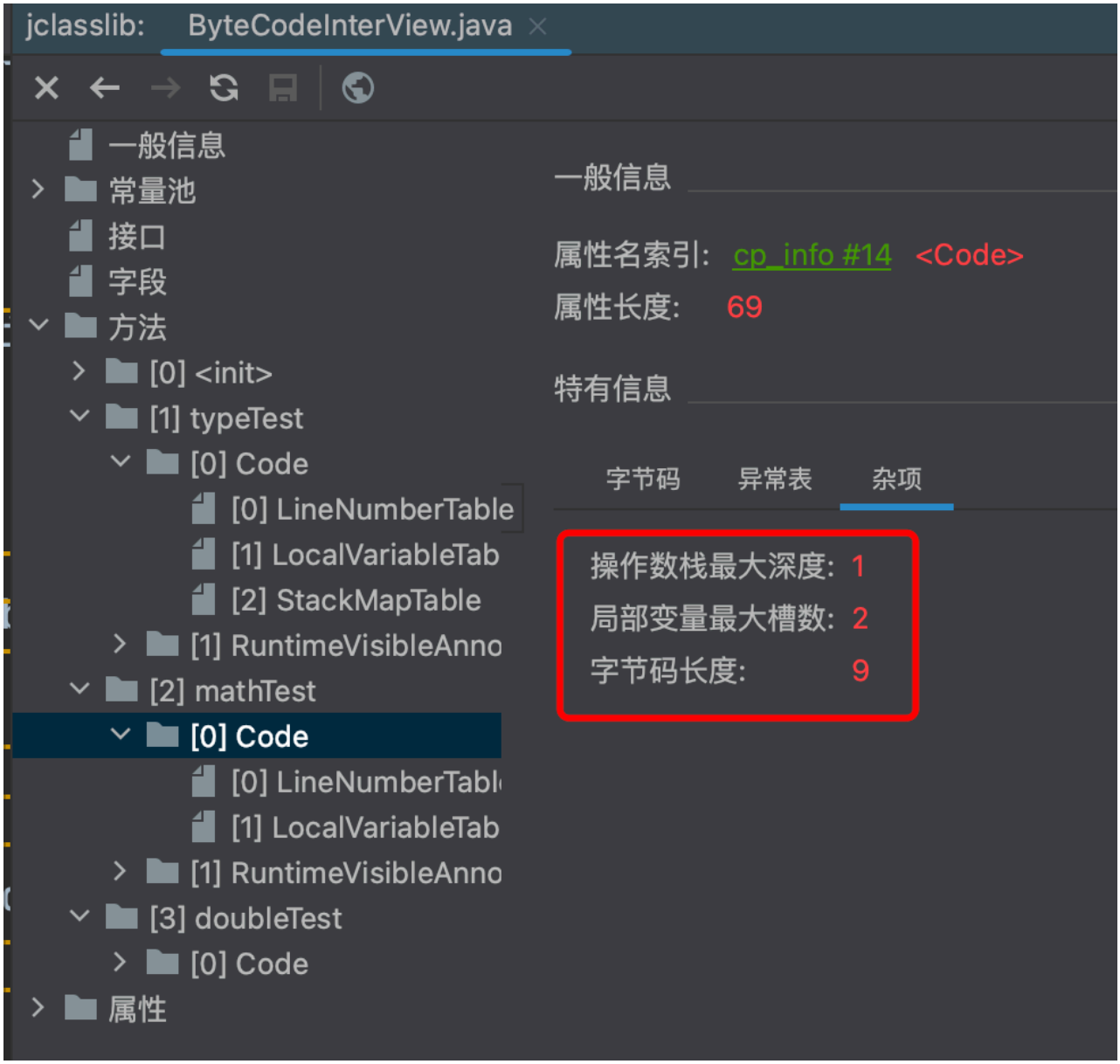
我们都知道k的返回结果是 1，但是++自增操作到底有没有执行呢？就可以按照指令这样进行解释：

```
0 iconst_1   //往操作数栈中压入一个常量1
1 istore_1    // 将 int 类型值从操作数栈中移出到局部变量表1 位置
2 iload_1   // 从局部变量表1 位置装载int 类型的值到操作数栈中
3 iinc 1 by 1  // 将局部变量表 1 位置的数字增加 1
6 istore_1  // 将int类型值从操作数栈中移出到局部变量表1 位置
7 iload_1  // 从局部变量表1 位置装载int 类型的值到操作数栈中
8 ireturn  // 从操作数栈顶，返回 int 类型的值
```

> 这个过程中，k++是在局部变量表中对数字进行了自增，此时栈中还是 1。接下来执行=操作，就对应一个istore指令，从栈中将数字装载到局部变量表中。局部变量表中的k的值(对应索引 1 位置)，就还是还原成了 1。

那么接下来，你是不是可以自行理解一下 k=++k，是怎么执行的呢？

> 另外，这里补充一个在互联网大厂的高级职位面试过程中被问到过的细节问题：
>
> 如何确定一个方法需要多大的操作数栈和局部变量？
>
> 实际上，每个方法在执行前都需要申请对应的资源，主要是内存。如果内存空间不够，就要在执行前直接抛出异常，而不能等到执行过程中才发现要用的内存空间申请不下来。
>
> 有些面试时，是会给你一个具体的方法，让你自己一下计算过程中需要几个操作数栈和几个局部变量。这是对算法的基础要求。但是在工作中，其实class文件当中就记录了所需要的操作数栈深度和局部变量表的槽位数。例如对于 mathTest方法，所需的资源在工具中的纪录是这样的：



这里会有一个小问题，如果你自己推演过刚才的计算过程，可以看到，局部变量表中，明明只用到了索引为 1 的一个位置而已，为什么局部变量表的最大槽数是 2 呢?

这是因为对于非静态方法，JVM 默认都会在局部变量表的 0 号索引位置放入this变量，指向对象自身。所以我们可以在代码中用this访问自己的属性。

一个槽可以存放 Java 虚拟机的基本数据类型，对象引用类型和returnAddress类型

三、类加载-二十分钟

Class文件中已经定义好了一个Java程序执行的全部过程，接下来就是要扔到JVM中执行。既然要执行，就少不了类加载的模块。而好玩的是，类加载模块是少数几个可以在Java代码中扩展的JVM底层功能。

类加载模块在JDK8之后，发生了非常重大的变化。后续以JDK8为主。JDK8以后的类加载功能，会在后续课程中介绍。

1 、JDK8的类加载体系

有了 Class 文件之后，接下来就需要通过类加载模块将这些 Class 文件加载到 JVM 内存当中，这样才能执行。而关于类加载模块，以 JDK8 为例，最为重要的内容我总结为三点：

- 每个类加载器对加载过的类保持一个缓存。
- 双亲委派机制，即向上委托查找，向下委托加载。
- 沙箱保护机制。

2、双亲委派机制

JDK8中的类加载器都继承于一个统一的抽象类ClassLoader，类加载的核心也在这个父类中。其中，加载类的核心方法如下：

```
//类加载器的核心方法
protected Class<?> loadClass(String name, boolean resolve)
        throws ClassNotFoundException
    {
        synchronized (getClassLoadingLock(name)) {
            // 每个类加载起对他加载过的类都有一个缓存，先去缓存中查看有没有加载过
            Class<?> c = findLoadedClass(name);
            if (c == null) {
                //没有加载过，就走双亲委派，找父类加载器进行加载。
                long t0 = System.nanoTime();
                try {
                    if (parent != null) {
                        c = parent.loadClass(name, false);
                    } else {
                        c = findBootstrapClassOrNull(name);
                    }
                } catch (ClassNotFoundException e) {
                }

                if (c == null) {
                    long t1 = System.nanoTime();
                    // 父类加载起没有加载过，就自行解析class文件加载。
                    c = findClass(name);

                    sun.misc.PerfCounter.getParentDelegationTime().addTime(t1 - t0);
                    sun.misc.PerfCounter.getFindClassTime().addElapsedTimeFrom(t1);
                    sun.misc.PerfCounter.getFindClasses().increment();
                }
            }
            //这一段就是加载过程中的链接Linking部分，分为验证、准备，解析三个部分。
            // 运行时加载类，默认是无法进行链接步骤的。
            if (resolve) {
                resolveClass(c);
            }
            return c;
        }
    }
```

这个方法里，就是最为核心的双亲委派机制。

并且，这个方法是protected声明的，意味着，是可以被子类覆盖的，所以，双亲委派机制也是可以被打破的。

> 为什么要打破双亲委派呢？想想Tomcat要如何加载webapps目录下的多个不同的应用？
>
> 而关于类加载机制的所有有趣的玩法，也都在这个核心方法里。比如class文件加密加载，热加载等。

3、沙箱保护机制

双亲委派机制有一个最大的作用就是要保护JDK内部的核心类不会被应用覆盖。而为了保护JDK内部的核心类，JAVA在双亲委派的基础上，还加了一层保险。就是ClassLoader中的下面这个方法。

```
private ProtectionDomain preDefineClass(String name,
                                        ProtectionDomain pd)
    {
        if (!checkName(name))
            throw new NoClassDefFoundError("IllegalName: " + name);
        // 不允许加载核心类
        if ((name != null) && name.startsWith("java.")) {
            throw new SecurityException
                ("Prohibited package name: " +
                 name.substring(0, name.lastIndexOf('.')));
        }
        if (pd == null) {
            pd = defaultDomain;
        }
        if (name != null) checkCerts(name, pd.getCodeSource());
        return pd;
    }
```

这个方法会用在JAVA在内部定义一个类之前。这种简单粗暴的处理方式，当然是有很多时代的因素。也因此在JDK中，你可以看到很多javax开头的包。这个奇怪的包名也是跟这个沙箱保护机制有关系的。

4、类和对象有什么关系

通过类加载模块，我们写的class文件就可以加载到JVM当中。但是类加载模块针对的都是类，而我们写的java程序都是基于对象来执行。类只是创建对象的模板。那么类和对象倒是什么关系呢?

首先：类 Class 在 JVM 中的作用其实就是一个创建对象的模板。也就是说他的作用更多的体现在创建对象的过程当中。而在程序具体执行的过程中，主要是围绕对象在进行，这时候类的作用就不大了。因此，在 JVM 中，类并不直接保存在最宝贵最核心的堆内存当中，而是挪到了堆内存以外的一部分内存中。这部分内存，在 JDK8 以前被成为永久带PermSpace，而在 JDK8 之后被改为了元空间 MetaSpace。

> 堆空间可以理解为JVM的客厅，所有重要的事情都在客厅处理。元空间可以理解为JVM的库房，东西扔进去基本上就很少管了。
>
> 这个元空间逻辑上可以认为是堆空间的一部分，但是他跟堆空间有不同的配置参数，不同的管理方式。因此也可以看成是单独的一块内存。这一块内存就相当于家里的工具间或者地下室，都是放一些用得比较少的东西。最主要就是类的一些相关信息，比如类的元数据、版本信息、注解信息、依赖关系等等。

> 元空间可以通过-XX:MetaspaceSize 和 -XX:MaxMetaspaceSize参数设置大小。但是大部分情况下，你是不需要管理元空间大小的，JVM 会动态进行分配。
>
> 另外，这个元空间也是会进行 GC 垃圾回收的。如果一个类不再使用了，JVM 就会将这个类信息从元空间中删除。但是，显然，对类的回收效率是很低的。只有一些自定义类加载器自行加载的一些类有被回收的可能，大部分情况下，类是不会被回收的。所以堆元空间的垃圾回收基本上是很少有效果的。大部分情况下，我们是不需要管元空间的。除非你的JVM 内存确实非常紧张，这时可以设定 -XX:MaxMetaspaceSize参数，严格控制元空间大小。

然后：在我们创建的每一个对象中，JVM也会保存对应的类信息。

在堆中，每一个对象的头部，还会保存这个对象的类指针(classpoint)，指向元空间中的类。这样我们就可以通过一个对象的getClass方法获取到对象所属的类了。这个类指针，我们也是可以通过一个小工具观察到的。

例如，下面这个 Maven依赖就可以帮我们分析一个对象在堆中保存的信息。

```
<dependency>
      <groupId>org.openjdk.jol</groupId>
      <artifactId>jol-core</artifactId>
      <version>0.17</version>
    </dependency>
```

然后可以用以下方法简单查看一下对象的内存信息。

```
public class JOLDemo {
    private String id;
    private String name;
    public static void main(String[] args) {
        JOLDemo o = new JOLDemo();
        System.out.println(ClassLayout.parseInstance(o).toPrintable());

        synchronized (o){
            System.out.println(ClassLayout.parseInstance(o).toPrintable());
        }
    }
}
```

看到的结果大概是这样：

```
com.roy.JOLDemo object internals:
OFF  SZ               TYPE DESCRIPTION               VALUE
  0   8                    (object header: mark)     0x0000000000000001 (non-biasable; age: 0)
  8   4                    (object header: class)    0x00060828
 12   4   java.lang.String JOLDemo.id                null
 16   4   java.lang.String JOLDemo.name              null
 20   4                    (object alignment gap)
Instance size: 24 bytes
Space losses: 0 bytes internal + 4 bytes external = 4 bytes total

com.roy.JOLDemo object internals:
OFF  SZ               TYPE DESCRIPTION               VALUE
  0   8                    (object header: mark)     0x000000016bb8aa20 (thin lock: 0x000000016bb8aa20)
  8   4                    (object header: class)    0x00060828
 12   4   java.lang.String JOLDemo.id                null
 16   4   java.lang.String JOLDemo.name              null
 20   4                    (object alignment gap)
Instance size: 24 bytes
Space losses: 0 bytes internal + 4 bytes external = 4 bytes total
```


这里ClassPoint 实际上就是一个指向元空间对应类的一个指针。当然，具体结果是被压缩过的。

另外Markdown标志位就是对象的一些状态信息。包括对象的 HashCode，锁状态，GC分代年龄等等。

> 这里面锁机制是面试最喜欢问的地方。无锁、偏向锁(新版本 JDK 中已经废除)、轻量级锁、重量级锁这些东西，都是在Markdown中记录的。

四、执行引擎-五分钟

之前已经看到过，在 Class 文件当中，已经明确的定义清楚了程序的完整执行逻辑。而执行引擎就是将这些字节指令转为机器指令去执行了。这一块更多的是跟操作系统打交道，对开发工作其实帮助就不是很大了。所以，如果不是专门研究语言，执行引擎这一块就没有必要研究太深了。

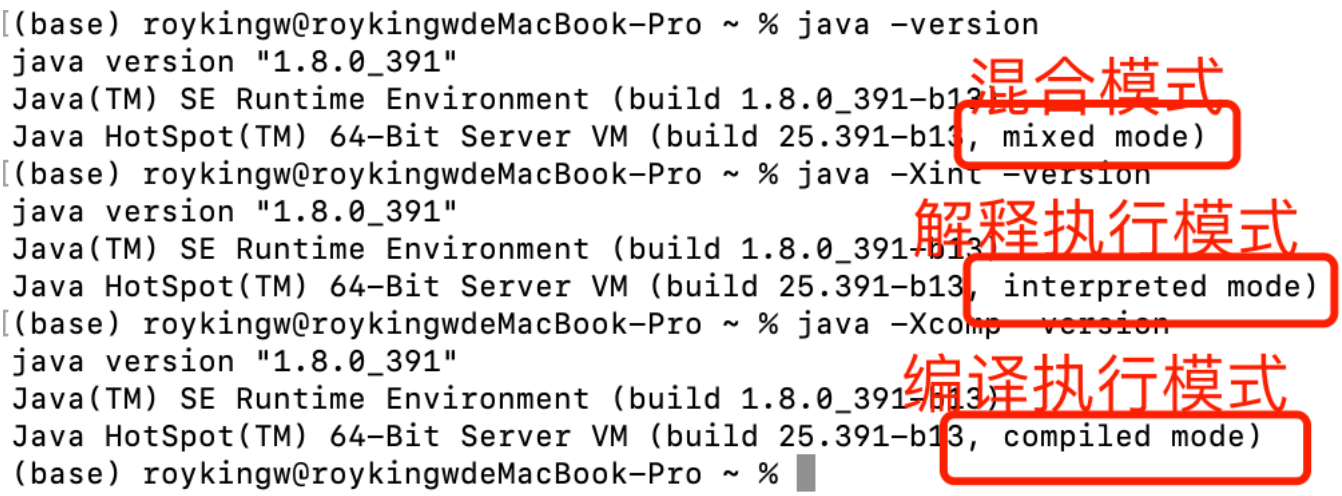
1、解释执行与编译执行

JVM 中有两种执行的方式：

- 解释执行就相当于是同声传译。JVM 接收一条指令，就将这条指令翻译成机器指令执行。
- 编译执行就相当于是提前翻译。好比领导发言前就将讲话稿提前翻译成对应的文本，上台讲话时就可以照着念了。编译执行也就是传说中的 JIT 。

大部分情况下，使用编译执行的方式显然比解释执行更快，减少了翻译机器指令的性能消耗。而我们常用的 HotSpot 虚拟机，最为核心的实现机制就是这个 HotSpot 热点。他会搜集用户代码中执行最频繁的热点代码，形成CodeCache，放到元空间中，后续再执行就不用编译，直接执行就可以了。

但是编译执行起始也有一个问题，那就是程序预热会比较慢。毕竟作为虚拟机，你不可能提前预知到程序员要写一些什么稀奇古怪的代码，也就不可能把所有代码都提前编译成模板。而将执行频率并不高的代码也编译保存下来，也是得不偿失的。所以，现在JDK 默认采用的就是一种混合执行的方式。他会自己检测采用那种方式执行更快。虽然你可以干预 JDK 的执行方式，但是在绝大部分情况下，都是不需要进行干预的。



> 另外，现在也有一种提前编译模式，AOT 。可以直接将Java 程序编译成机器码。比如GraalVM，可以直接将 Java 程序编译成可执行文件，这样就不需要 JVM 虚拟机也能直接在操作系统上执行。
>
> 关于 AOT 是不是会一统天下，也是现在面试中比较喜欢问的问题。虽然在 SpringBoot3 等框架中已经有了落地，但是从目前来看，AOT还远没有成为主流，离一统天下还有点距离。
>
> 少了 JVM 这个中间商之后，虽然大部分情况下是可以提升程序执行性能的，但是，也并不是就完美无缺了。毕竟很显然，这种方式其实是以丧失一定的跨平台特性作为代价的。
>
> 要注意，目前 AOT 这种方式还是不太安全的。毕竟 JVM 打了这么多年的怪，什么牛鬼蛇神都见多了。现在 AOT 要绕开 JVM，那么这些怪就都要自己去打了。中间有个什么疏忽，那是难免的。

2、编译执行时的代码优化

热点代码会触发 JIT 实时编译，而JIT 编译运用了一些经典的编译优化技术来实现代码的优化，可以智能地编译出运行时的最优性能代码。

HotSpot虚拟机中内置了两个（或三个）即时编译器，其中有两个编译器存在已久，分别被称为“客户端编译器”（Client Compiler）和“服务端编译器”（Server Compiler），或者简称为C1编译器和C2编译器（部分资料和JDK源码中C2也叫Opto编译器），第三个是在JDK 10时才出现的、长期目标是代替C2的Graal编译器。Graal编译器采用 Java 语言编写，因此生态的活力更强。并由此衍生出了 GraalVM 这样的支持实时编译的产品。也就是绕过 Class 文件，直接将 Java 代码编译成可在操作系统本地执行的应用程序。这也就是 AOT 技术Ahead Of Time。

C1 会对字节码进行简单和可靠的优化，耗时短，以达到更快的编译速度。启动快，占用内存小，执行效率没有server快。默认情况下不进行动态编译，适用于桌面应用程序。

C2 进行耗时较长的优化，以及激进优化，但优化的代码执行效率更高。启动慢，占用内存多，执行效率高，适用于服务器端应用。 默认情况下就是使用的 C2 编译器。并且，绝大部分情况下也不建议特意去使用 C1。

由于即时编译器编译本地代码需要占用程序运行时间，通常要编译出优化程度越高的代码，所花费的时间便会越长；而且想要编译出优化程度更高的代码，解释器可能还要替编译器收集性能监控信息，这对解释执行阶段的速度也有所影响。为了在程序启动响应速度与运行效率之间达到最佳平衡，HotSpot虚拟机在编译子系统中加入了分层编译的功能，分层编译根据编译器编译、优化的规模与耗时，划分出不同的编译层次，其中包括：

- 第0层。程序纯解释执行，并且解释器不开启性能监控功能（Profiling）。
- 第1层。使用C1编译器将字节码编译为本地代码来运行，进行简单可靠的稳定优化，不开启性能监控功能。
- 第2层。仍然使用C1编译器执行，仅开启方法及回边次数统计等有限的性能监控功能。

- 第3层。仍然使用C1编译器执行，开启全部性能监控，除了第2层的统计信息外，还会收集如分支跳转、虚方法调用版本等全部的统计信息。
- 第4层。使用C2编译器将字节码编译为本地代码，相比起C1编译器，C2编译器会启用更多编译耗时更长的优化，还会根据性能监控信息进行一些不可靠的激进优化。

JDK8 中提供了参数 -XX:TieredStopAtLevel=1 可以指定使用哪一层编译模型。但是，除非你是JVM 的开发者，否则不建议干预 JVM 的编译过程。

3、静态执行与动态执行

静态执行指在 Class 文件编译过程中就已经确定了执行方法。 动态执行指需要在运行期间才能确定调用哪个方法。比如多个重载的方法，需要根据传入类型确定调用哪个方法。

动态执行更多的是关联到invokedynamic指令。在JVM的语言体系中，以Scala为代表的函数式的编程方式会越来越重要，到时候动态执行也会随之变得更为重要。

五、GC 垃圾回收-三十分钟

执行引擎会将class文件扔到JVM的内存当中运行。在运行过程中，需要不断的在内存当中创建并销毁对象。在传统C/C++语言中，这些销毁的对象需要手动进行内存回收，防止内存泄漏。而在Java当中，实现了影响深远的GC垃圾回收机制。

GC 垃圾自动回收，这个可以说是 JVM 最为标志性的功能。不管是做性能调优，还是工作面试，GC 都是 JVM 部分的重中之重。而对于 JVM 本身，GC 也是不断进行设计以及优化的核心。几乎 Java 提出的每个版本都对 GC 有或大或小的改动。这里，我就用目前还是用得做多的 JDK8，带大家快速梳理一下 GC 部分的主线。

1 、垃圾回收器是干什么的

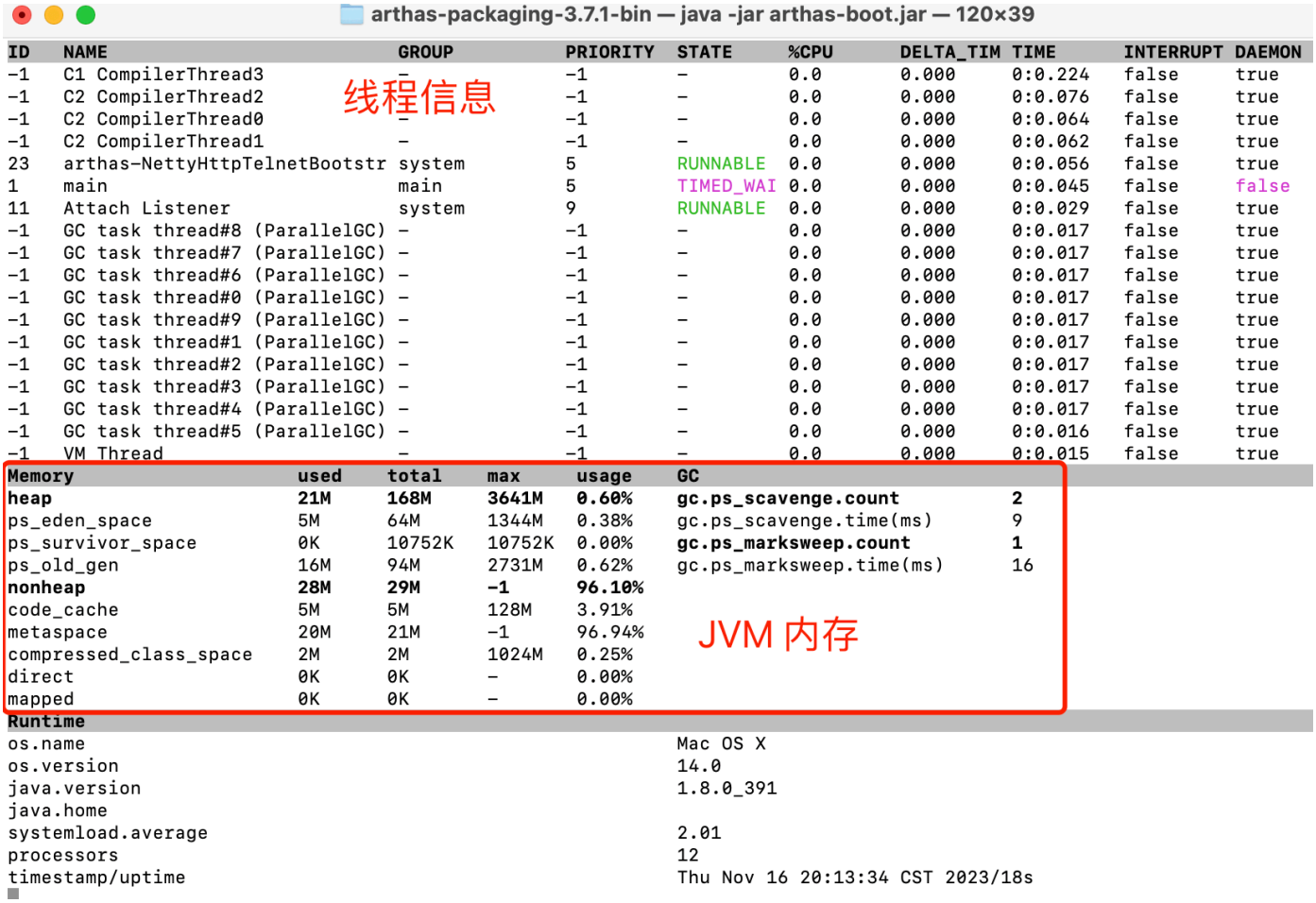
在了解 JVM之前，给大家推荐一个工具，阿里开源的 Arthas 。官网地址：https://arthas.aliyun.com/ 。 这个工具功能非常强大，是对 Java进程进行性能调优的一个非常重要的工具，对于了解 JVM 底层帮助也非常大。

> 具体使用方式参照官方文档。

我们先运行一个简单的 Java 程序：

```
public class GCTest {
    public static void main(String[] args) throws InterruptedException {
        List l = new ArrayList<>();
        for(int i = 0 ; i < 100_0000 ; i ++){
            l.add(new String("ddddddddddd"));
            Thread.sleep(100);
        }
    }
}
```

运行后，使用Arthas 的dashboard指令，可以查看到这个 Java 程序的运行情况。

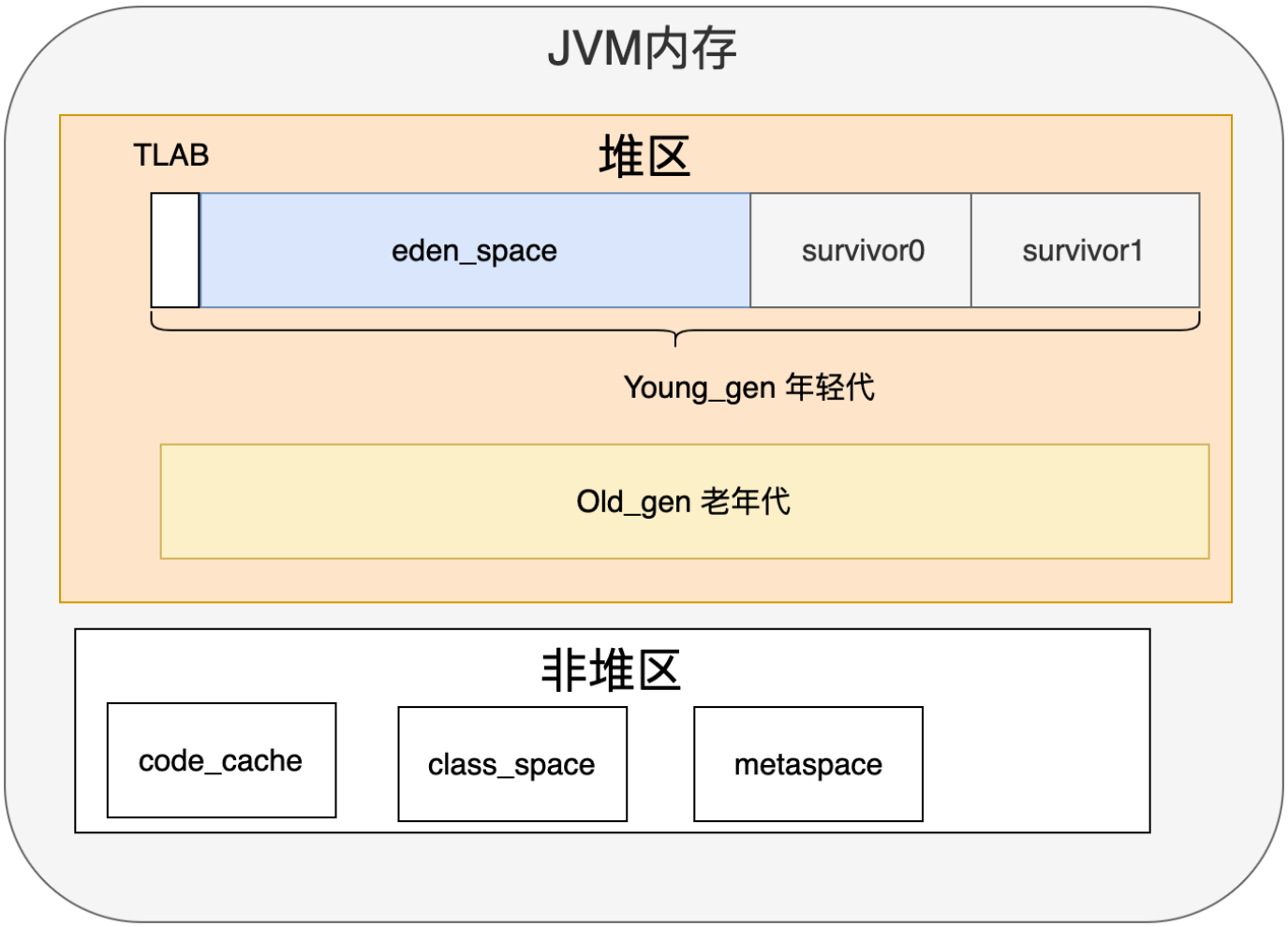



```
ID   NAME                              GROUP           PRIORITY   STATE     %CPU     DELTA_TIM TIME     INTERRUPT DAEMON
-1   C1 CompilerThread3                -               -1         -         0.0      0.000     0:0.224  false     true
-1   C2 CompilerThread2                -               -1         -         0.0      0.000     0:0.076  false     true
-1   C2 CompilerThread0                -               -1         -         0.0      0.000     0:0.064  false     true
-1   C2 CompilerThread1                -               -1         -         0.0      0.000     0:0.062  false     true
23   arthas-NettyHttpTelnetBootstr     system          5          RUNNABLE  0.0      0.000     0:0.056  false     true
1    main                              main            5          TIMED_WAI 0.0      0.000     0:0.045  false     false
11   Attach Listener                   system          9          RUNNABLE  0.0      0.000     0:0.029  false     true
-1   GC task thread#8 (ParallelGC)     -               -1         -         0.0      0.000     0:0.017  false     true
-1   GC task thread#7 (ParallelGC)     -               -1         -         0.0      0.000     0:0.017  false     true
-1   GC task thread#6 (ParallelGC)     -               -1         -         0.0      0.000     0:0.017  false     true
-1   GC task thread#0 (ParallelGC)     -               -1         -         0.0      0.000     0:0.017  false     true
-1   GC task thread#9 (ParallelGC)     -               -1         -         0.0      0.000     0:0.017  false     true
-1   GC task thread#1 (ParallelGC)     -               -1         -         0.0      0.000     0:0.017  false     true
-1   GC task thread#2 (ParallelGC)     -               -1         -         0.0      0.000     0:0.017  false     true
-1   GC task thread#3 (ParallelGC)     -               -1         -         0.0      0.000     0:0.017  false     true
-1   GC task thread#4 (ParallelGC)     -               -1         -         0.0      0.000     0:0.017  false     true
-1   GC task thread#5 (ParallelGC)     -               -1         -         0.0      0.000     0:0.016  false     true
-1   VM Thread                         -               -1         -         0.0      0.000     0:0.015  false     true
Memory                     used      total     max       usage     GC
heap                       21M       168M      3641M     0.60%     gc.ps_scavenge.count          2
ps_eden_space              5M        64M       1344M     0.38%     gc.ps_scavenge.time(ms)       9
ps_survivor_space          0K        10752K    10752K    0.00%     gc.ps_marksweep.count         1
ps_old_gen                 16M       94M       2731M     0.62%     gc.ps_marksweep.time(ms)      16
nonheap                    28M       29M       -1        96.10%
code_cache                 5M        5M        128M      3.91%
metaspace                  20M       21M       -1        96.94%
compressed_class_space     2M        2M        1024M     0.25%
direct                     0K        0K        -         0.00%
mapped                     0K        0K        -         0.00%
Runtime
os.name                                          Mac OS X
os.version                                       14.0
java.version                                     1.8.0_391
java.home
systemload.average                               2.01
processors                                       12
timestamp/uptime                                 Thu Nov 16 20:13:34 CST 2023/18s
```

重点关注中间的 Memory 部分，这一部分就是记录的 JVM 的内存使用情况。而后面的 GC 部分就是垃圾回收的执行情况。我们就从这些能看到的部分作为入口，来理解一下一个 Java 进程是怎么管理他的内存的。

从 Memory 部分可以看到，一个 Java 进程会将他管理的内存分为heap堆区和nonheap非堆区两个部分。其中非堆区的几个核心部分像code_cache(热点指令缓存)，metaspace(元空间),compressed_class_space(压缩类空间)我们之前都接触到了。这一部分就相当于 Java 进程中的地下室，属于不太活跃的部分。而中间heap堆区就相当于客厅了，属于Java 中最为核心的部分。而这其中，又大体分为了eden_space，survivor_space和old_gen三个大的部分，这就是 JVM 内存的主体。我们之前分析的栈区，这里没有列出。

整体内存布局如下图：



其中堆区是 JVM 核心的存放对象的内存区域。他的大小可以由参数 -Xms(初始堆内存大小)，-Xmx(最大堆内存)参数指令。从这两个参数可以看到，堆内存是可以扩展的。如果初始内存不够，JVM 会扩大堆内存。但是如果内存扩展到了最大堆内存时还不够。这时就无法继续扩展了，而是会抛出 OOM 异常。这两个参数在生产环境中最好设置成一样，减少内存扩展时的性能消耗。

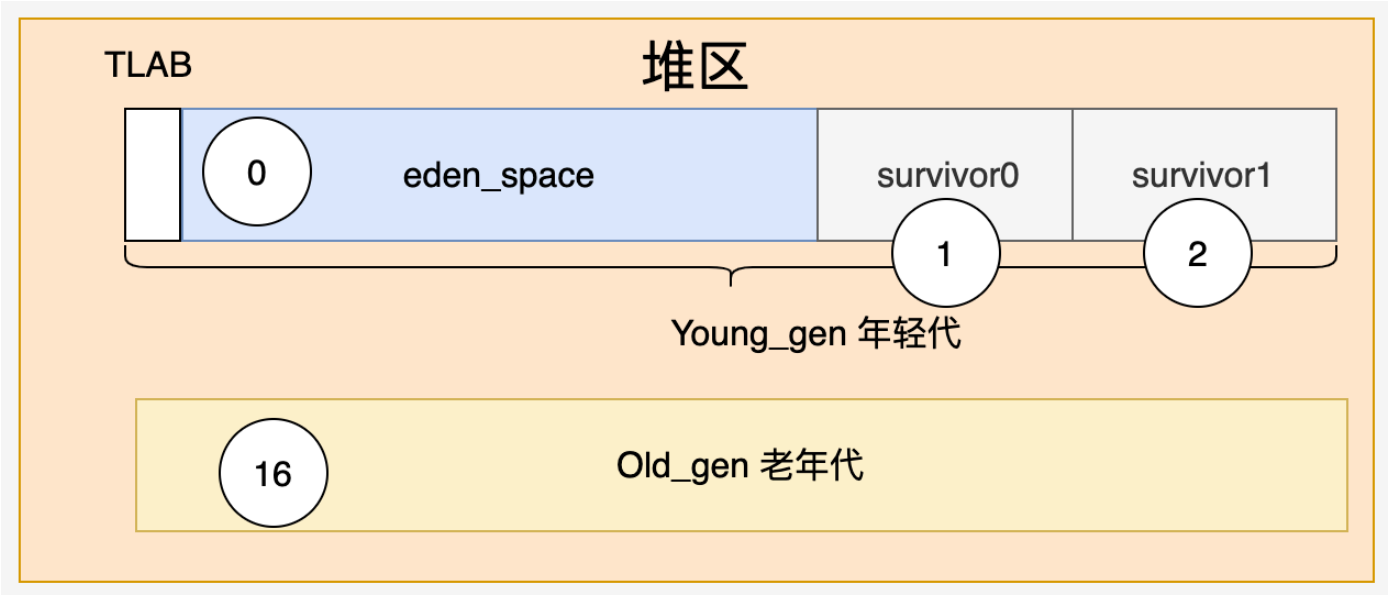
而GC垃圾回收器，就是要对这些内存空间进行及时回收，从而让这些内存可以重复利用。

2、分代收集模型

不同GC，对内存的管理和回收的方式都是不同的。但是这其中面试最喜欢问的，就是关于垃圾分代收集模型。

在Memor部分还可以看到多次出现了 ps_ 这样的字样。这其实就代表JDK8默认的垃圾回收器Parallel Scavenge。

其整体的工作机制如下图：



JAVA做过统计，80%的对象都是“朝生夕死”。这些对象，被集中放在了一块比较小的内存空间当中，快速创建，快速回收，这块内存区域就是年轻代。在年轻代会非常频繁的进行垃圾回收，称为YoungGC。而年轻代又会被进一步划分为一个eden_space和两个survivor。这三个区域的大小比例默认是 8:1:1。

另外少部分需要长期使用的对象，被放到另一块竞争没有那么激烈的对象，则被放到另外一块比较大的内存空间当中，长期保持，这块内存就是老年代。在老年代，垃圾回收的频率则会相对比较低，只有空间不够时才进行，称为OldGC。

年轻代与老年代默认的大小比例是 1:2。

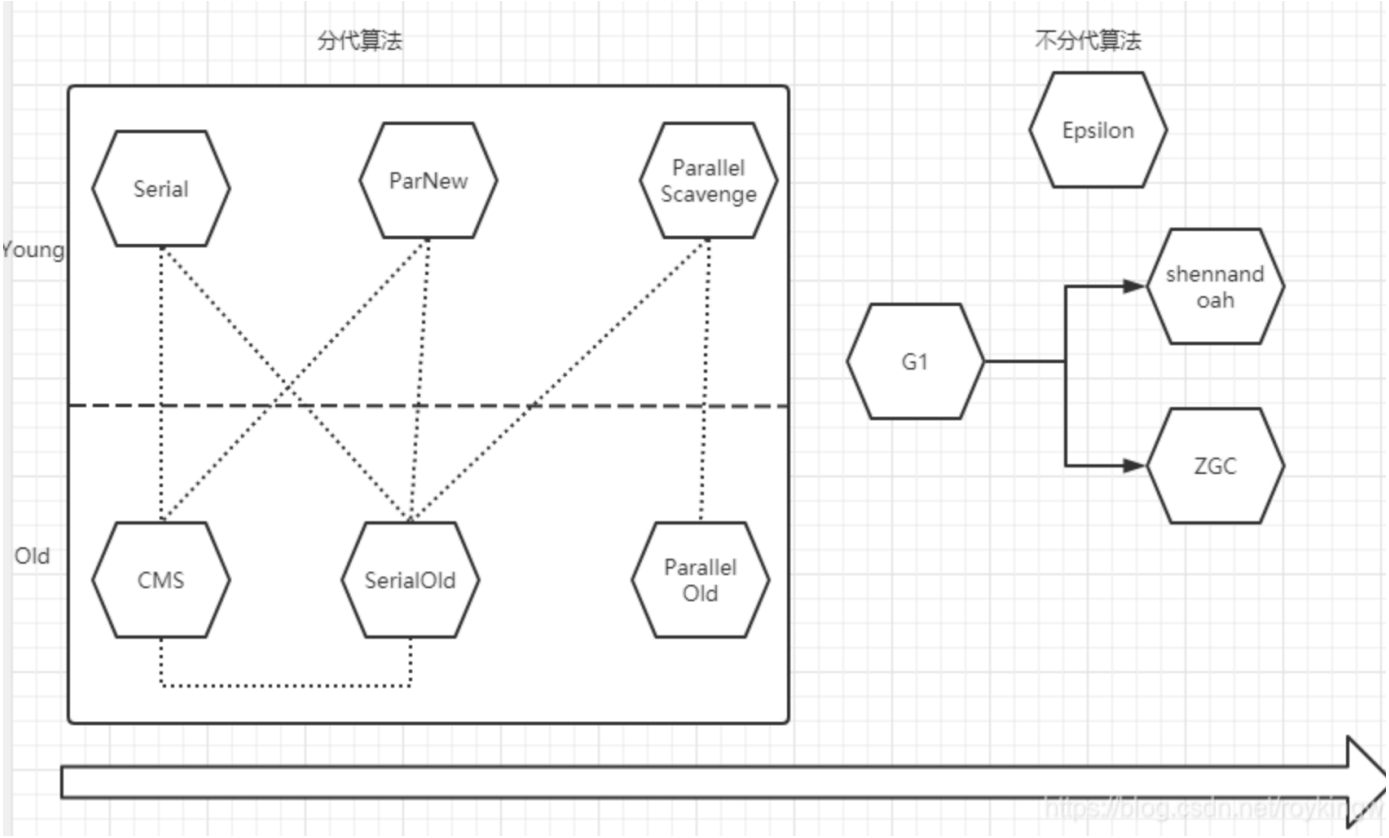
常见的分代收集模型中，对象会优先在eden区创建，经过一次YoungGC后，如果没有被回收，就会被移动到一个survivor区。接下来，下一次YoungGC时，又会被移动到另一块Survivor区。每移动一次，记录一个分代年龄。直到分代年龄太大了(默认是16)，就会被移动到老年代。到老年代后，对象就不再记录分代年龄了，在老年代安安静静的用到退休。

这就是JDK最有代表性的分代年龄收集机制。通过分代收集机制，JVM可以对不同的对象采取不同的回收策略，从而提高垃圾回收的效率。

> 当然，分代收集机制在具体实现过程中，还需要根据具体情况提供更多优化机制。后续会带大家详细了解。
>
> 比如，如果小 O 占用内存非常小，那么在创建小 O 时，JVM 会在Eden_space中单独划分出一小片线程专属的内存空间，称为 TLAB 。小 O 就在 TLAB 中创建。由于 TLAB 空间是线程私有的，所以就可以避免多个线程之间的资源争抢。
>
> 另外，如果小 O 占用的内存非常大，Eden_space都装不下。这时小 O 就会跳过年轻代，直接进入老年代。

3 、JVM中有哪些垃圾回收器?

java 从诞生到现在最新的 JDK21 版本，总共就产生了以下十个垃圾回收器



其中，左边的都是分代算法。也就是将内存划分为年轻代和老年代进行管理。而有虚线的部分表示可以协同进行工作。JDK8默认就是使用的Parallel Scavenge和Parallel Old的组合。也就是在arthas的dashboard中看到的ps。

右侧的是不分代算法。也就是不再将内存严格划分位年轻代和老年代。JDK9 开始默认使用 G1。而 ZGC是目前最先进的垃圾回收器。shennandoah则是OpenJDK 中引入的新一代垃圾回收器，与 ZGC 是竞品关系。Epsilon是一个测试用的垃圾回收器，根本不干活。

> 关于垃圾回收器的细节，后续的VIP课程会详细分析。

六、 GC 情况分析实例-十分钟

关于各个垃圾回收器的细节，后面的课程会做更深入的分享。这里我们只关心GC和开发工作的关系。GC可以说是决定JAVA程序运行效率的关键。因此我们一定要学会定制GC参数，以及分析GC日志。

1、如何定制GC运行参数

在现阶段，各种GC垃圾回收器都只适合一个特定的场景，因此，我们也需要根据业务场景，定制合理的GC运行参数。

另外，JAVA程序在运行过程中要处理的问题是层出不穷的。项目运行期间会面临各种各样稀奇古怪的问题。比如CPU 超高，FullGC 过于频繁，时不时的 OOM 异常等等。这些问题大部分情况下都只能凭经验进行深入分析，才能做出针对性的解决。

如何定制JVM运行参数呢？首先我们要知道有哪些参数可以供我们选择。

关于 JVM 的参数，JVM 提供了三类参数。

一类是标准参数，以-开头，所有 HotSpot 都支持。例如java -version。这类参数可以使用java -help 或者java -? 全部打印出来

二类是非标准参数，以-X 开头，是特定 HotSpot版本支持的指令。例如java -Xms200M -Xmx200M。这类指令可以用java -X 全部打印出来。

最后一类，不稳定参数，这也是 JVM调优的噩梦。以-XX 开头，这些参数是跟特定HotSpot版本对应的，很有可能换个版本就没有了。详细的文档资料也特别少。JDK8 中的以下几个指令可以帮助开发者了解 JDK8 中的这一类不稳定参数。

```
java -XX:+PrintFlagsFinal:所有最终生效的不稳定指令。
java -XX:+PrintFlagsInitial:默认的不稳定指令
java -XX:+PrintCommandLineFlags:当前命令的不稳定指令 --这里可以看到是用的哪种GC。 JDK1.8默认用的ParallelGC
```

2、打印GC日志

有了手段之后，我们最主要的就是要能快速发现问题。

对 JVM 虚拟机来说，绝大多数的问题往往都跟堆内存的 GC 回收有关。因此下面几个跟 GC 相关的日志打印参数是必须了解的。这通常也是进行 JVM 调优的基础。

-XX:+PrintGC: 打印GC信息 类似于-verbose:gc

-XX:+PrintGCDetails: 打印GC详细信息，这里主要是用来观察FGC的频率以及内存清理效率。

-XX:+PrintGCTimeStamps 配合 -XX:+PrintGC使用。在 GC 中打印时间戳。

-XX:PrintHeapAtGC: 打印GC前后的堆栈信息

-Xloggc:filename : GC日志打印文件。

> 不同 JDK 版本会有不同的参数。 比如 JDK9 中，就不用分这么多参数了，可以统一使用-X-log:gc* 通配符打印所有的 GC 日志。

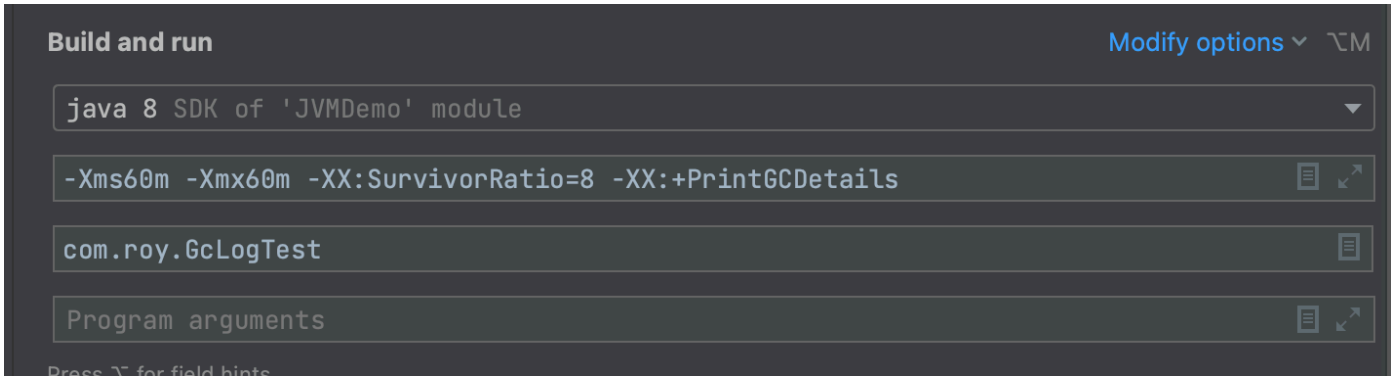
例如下面一个简单的示例代码：

```
public class GcLogTest {
    public static void main(String[] args) {
        ArrayList<byte[]> list = new ArrayList<>();

        for (int i = 0; i < 500; i++) {
            byte[] arr = new byte[1024 * 100];//100KB
            list.add(arr);
            try {
                Thread.sleep(10);
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
        }
    }
}
```

然后在执行这个方法时，添加以下 JVM 参数：

```
-Xms60m -Xmx60m -XX:SurvivorRatio=8 -XX:+PrintGCDetails
```



执行后，可以看到类似这样的输出信息。

```
[GC (Allocation Failure) [PSYoungGen: 17318K->2028K(19456K)] 17318K->15946K(63488K), 0.0059321 secs] [Times: user=0.01 sys=0.01, real=0.01 secs]
[GC (Allocation Failure) [PSYoungGen: 19368K->1996K(19456K)] 33286K->33225K(63488K), 0.0046327 secs] [Times: user=0.00 sys=0.01, real=0.01 secs]
[Full GC (Ergonomics) [PSYoungGen: 1996K->0K(19456K)] [ParOldGen: 31228K->33093K(44032K)] 33225K->33093K(63488K), [Metaspace: 3205K->3205K
 (1056768K)], 0.0099790 secs] [Times: user=0.05 sys=0.01, real=0.01 secs]
[Full GC (Ergonomics) [PSYoungGen: 17390K->6100K(19456K)] [ParOldGen: 33093K->43995K(44032K)] 50484K->50096K(63488K), [Metaspace: 3205K->3205K
 (1056768K)], 0.0079863 secs] [Times: user=0.02 sys=0.01, real=0.01 secs]
Heap
 PSYoungGen      total 19456K, used 6812K [0x00000007beb00000, 0x00000007c0000000, 0x00000007c0000000)
  eden space 17408K, 39% used [0x00000007beb00000,0x00000007bf1a7338,0x00000007bfc00000)
  from space 2048K, 0% used [0x00000007bfe00000,0x00000007bfe00000,0x00000007c0000000)
  to   space 2048K, 0% used [0x00000007bfc00000,0x00000007bfc00000,0x00000007bfe00000)
 ParOldGen       total 44032K, used 43995K [0x00000007bc000000, 0x00000007beb00000, 0x00000007beb00000)
  object space 44032K, 99% used [0x00000007bc000000,0x00000007beaf6de8,0x00000007beb00000)
 Metaspace       used 3213K, capacity 4496K, committed 4864K, reserved 1056768K
  class space    used 353K, capacity 388K, committed 512K, reserved 1048576K
```

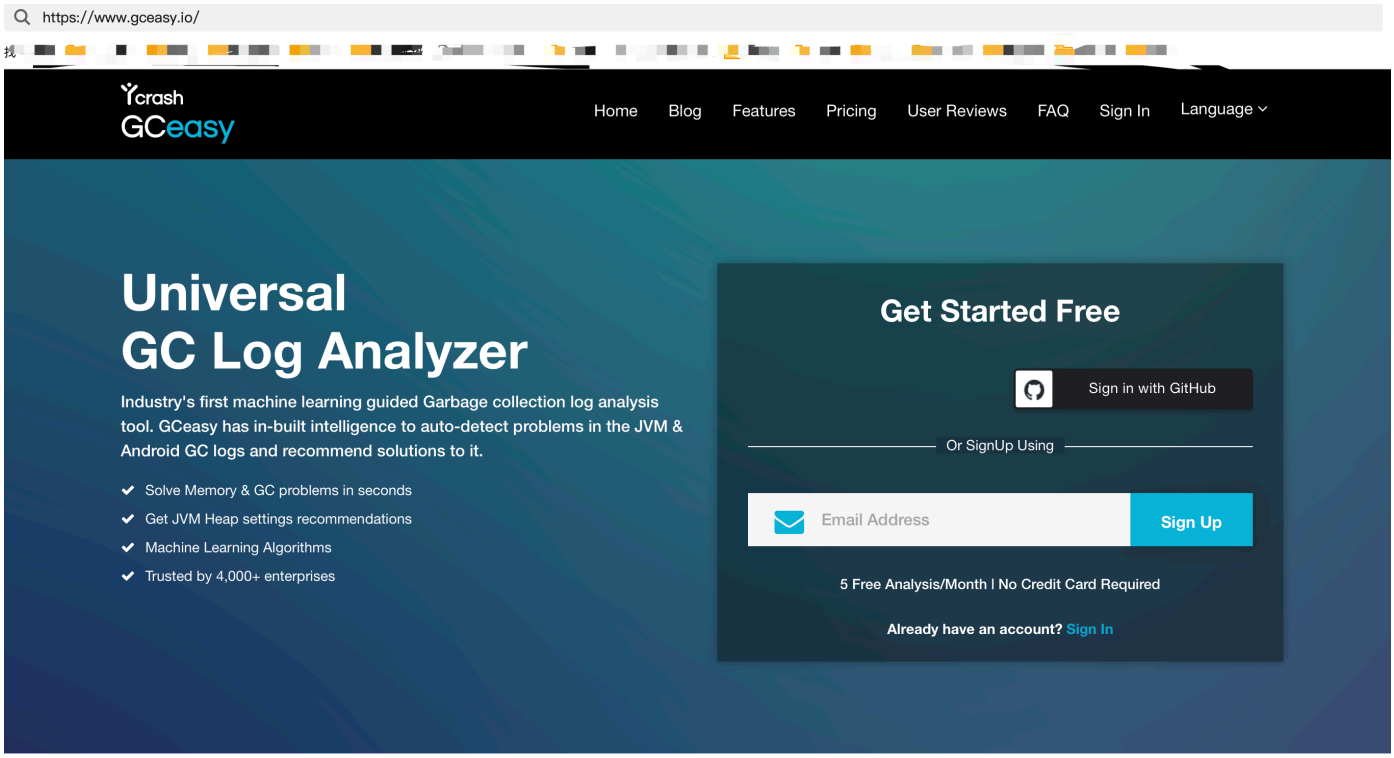
这里面就记录了两次 MinorGC 和两次 FullGC 的执行效果。另外，在程序执行完成后，也会打印出 Heap 堆区的内存使用情况。

当然，目前这些日志信息只是打印在控制台，你只能凭经验自己强行去看。接下来，就可以添加-Xloggc参数，将日志打印到文件里。然后拿日志文件进行整体分析。

3、GC日志分析

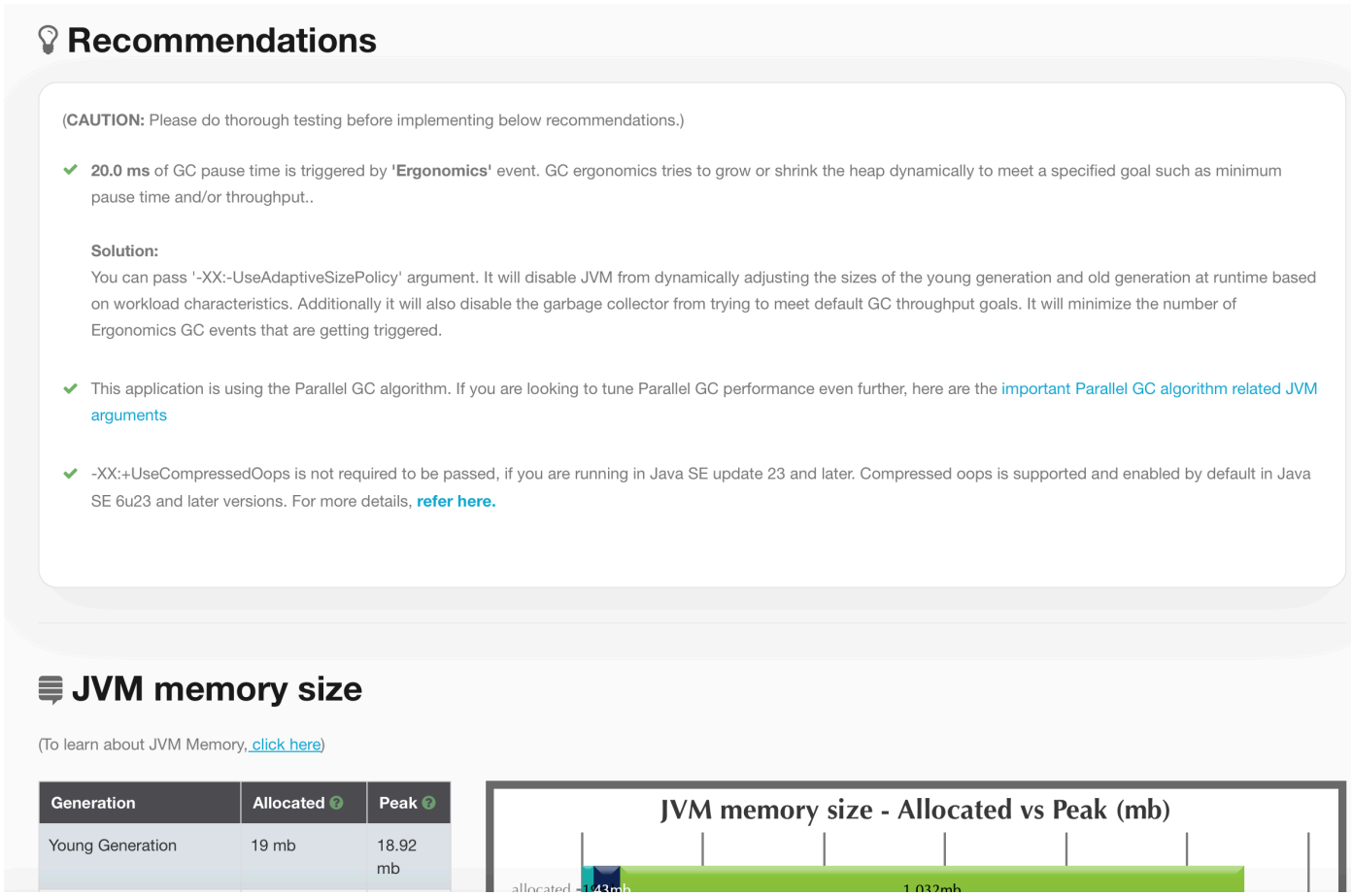
这些GC日志隐藏了项目运行非常多隐蔽的问题，要如何发现其中的这些潜在的问题呢?

这里推荐一个开源网站 https://www.gceasy.io/ 这是国外一个开源的GC 日志分析网站。你可以把 GC 日志文件直接上传到这个网站上，他就会分析出日志文件中的详细情况。



> 这是个收费网站，但是有免费使用的额度。

例如，在我们之前的示例中，添加一个参数 -Xloggc:./gc.log ，就可以将GC日志打印到文件当中。接下来就可以将日志文件直接上传到这个网站上。网站就会帮我们对GC情况进行分析。示例文件得到的报告是这样的：



通过这个报告，可以及时发现项目运行可能出现的一些隐藏问题。并且这个报告也提供了一些具体的修改意见。当然，如果你觉得这些建议还不够满意，那么报告中还提供了非常详细的指标分析，通过这些指标，你可以进一步的分析问题，寻找新的改进方向。

如果是你们自己开发的项目，那么接下来，根据这些建议和数据，做进一步的分析，调整参数，优化配置。到这里，恭喜你，架构师的绝活-JVM调优，你就算是正式入门了。

章节总结

聊到这里，你对于 JVM 是不是开始有一点感觉了？在这个过程当中，你是不是还有很多细节方面的疑问？保持这些疑问，这将成为你后续深入学习那些晦涩枯燥的底层理论的动力。 这不会是一个容易的过程。但是，有挑战才更有价值不是吗?

> 有道云笔记链接：https://note.youdao.com/s/S5Y3wV65