课程说明

- 再次认识Java
- JVM虚拟机内存管理
- 虚拟机性能相关工具
- 实战：内存溢出的定位与分析
- 实战：死锁问题
- VisualVM的应用

1、架构师面对JVM调优，能做什么？

架构师在做系统架构时，除了对于系统架构需要作出考虑外，在程序底层的jvm优化也是必然要考虑的事情，架构师需要考虑到我的系统如何才能更快，更稳定。

如果发现系统出现一些莫名其妙的状况，比如：

- 运行好好的服务突然停止运行了；
- 有的应用突然报内存溢出异常终止了；
- 服务器增加了硬件配置，但是服务的响应速度还是上不去；

作为架构师的你，该如何解决？

在做系统架构时，当我们面对高并发系统时jvm该如何调优？高吞吐的系统又该如何去对jvm做调优？

本套课程就是带领你，深层次的学习java虚拟机，在做系统架构时，轻松的面对上面所提到的问题。

2、再次认识Java

Java 是几乎所有类型的网络应用程序的基础，也是开发和提供嵌入式和移动应用程序、游戏、基于 Web 的内容和企业软件的全球标准。Java 在全球各地有超过 900 万的开发人员，使您能够高效地开发、部署和使用精彩的应用程序和服务。

从笔记本电脑到数据中心，从游戏控制台到科学超级计算机，从手机到互联网，Java 无处不在！

- 97% 的企业桌面运行 Java
- 美国有 89% 的桌面（或计算机）运行 Java
- 全球有 900 万 Java 开发人员
- 开发人员的头号选择
- 排名第一的部署平台
- 有 30 亿部移动电话运行 Java
- 100% 的蓝光盘播放器附带了 Java
- 有 50 亿张 Java 卡在使用
- 1.25 亿台 TV 设备运行 Java
- 前 5 个原始设备制造商均提供了 Java ME

> *说明：后续的内容部分参考自《深入理解VM》一书。*

2.1、Java技术体系

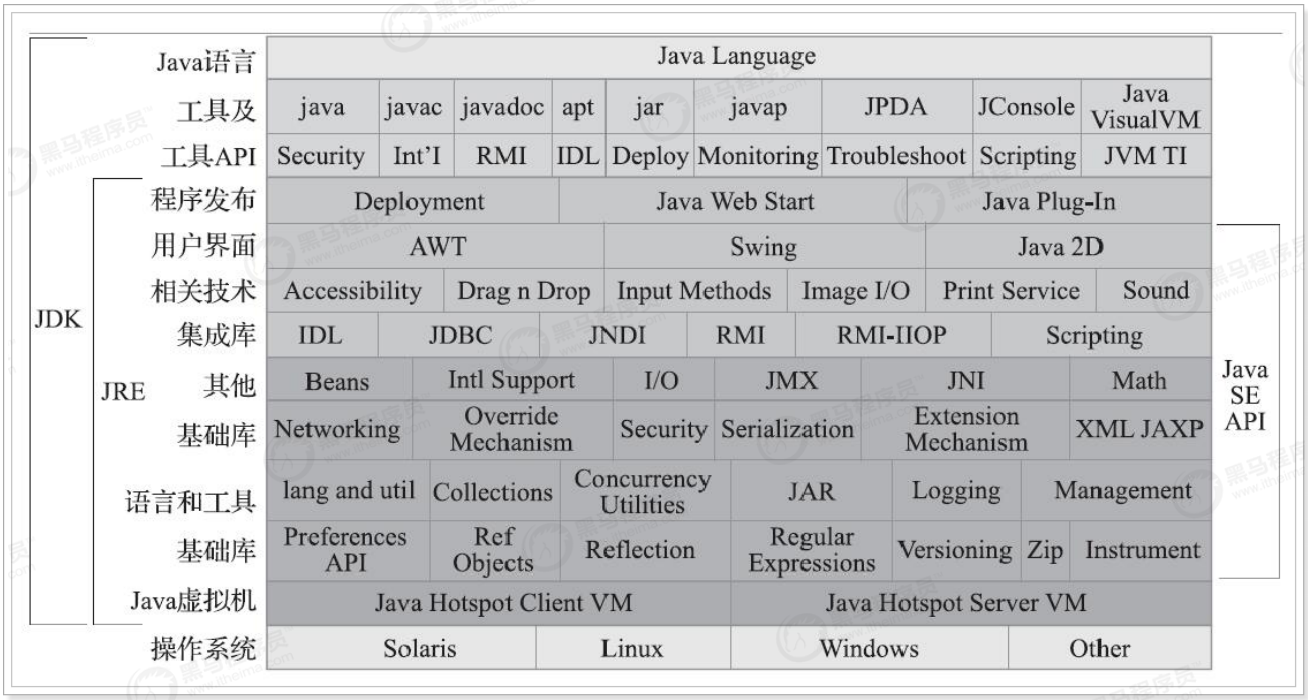
Java技术体系其实已经不仅仅是Java语言的专属，其实也包括可以运行在Java平台的其他语言，比如：Kotlin、Clojure、JRuby、Groovy等语言。对于我们Java程序员而言，我们所指的是JCP官方定义的Java体系。

> JCP：Java Community Process，就是人们常说的“Java社区”，这是一个由业界多家技术巨头组成的社区组织，用于定义和发展Java的技术规范。官网：https://jcp.org/en/home/index

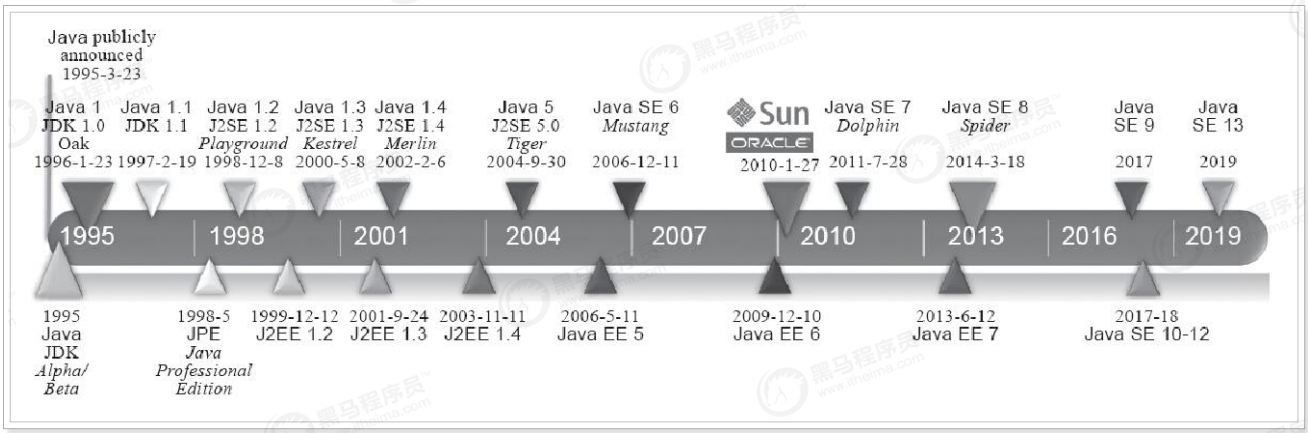
2.1.1、JDK & JRE

Java程序设计语言、Java虚拟机、Java类库这三部分统称为JDK（Java Development Kit），JDK是用于支持Java程序开发的最小环境。

Java类库API中的Java SE API子集和Java虚拟机这两部分统称为JRE（Java Runtime Environment），JRE是支持Java程序运行的标准环境。



2.2、Java发展历史



- 1995年5月23日，Oak语言改名为Java，并且在SunWorld大会上正式发布Java 1.0版本。Java语言第一次提出了“Write Once，Run Anywhere”的口号。
- 1999年4月27日，HotSpot虚拟机诞生。Hot-Spot虚拟机刚发布时是作为JDK 1.2的附加程序提供的，后来它成为JDK 1.3及之后所有JDK版本的默认Java虚拟机。
- 2004年9月30日，JDK 5发布，工程代号为Tiger（老虎），是Java语言的发展史上的又一里程碑事件。JDK的版本不再是“1.x”的命名方式，而采用“JDK x”方式进行命名。

- JDK 8的第一个正式版本于2014年3月18日发布，从JDK 8开始，Oracle启用JEP（JDK Enhancement Proposals）来定义和管理纳入新版JDK发布范围的功能特性。
    - JEP 126：对Lambda表达式的支持，这让Java语言拥有了流畅的函数式表达能力。
    - JEP 104：内置Nashorn JavaScript引擎的支持，成为Java的嵌入式JavaScript引擎。
    - JEP 150：新的时间、日期API。
    - JEP 122：彻底移除HotSpot的永久代。
    - ……
- JDK9于2017年9月21日发布。JDK 9发布后，Oracle随即宣布Java将会以持续交付的形式和更加敏捷的研发节奏向前推进，以后JDK将会在每年的3月和9月各发布一个大版本。每六个JDK大版本中才会被划出一个长期支持（Long Term Support，LTS）版，只有LTS版的JDK能够获得为期三年的支持和更新，普通版的JDK就只有短短六个月的生命周期。JDK 8和JDK 11是LTS版，再下一个就到2021年发布的JDK 17了。
- 2018年3月20日，JDK 10如期发布，这版本的主要研发目标是内部重构，诸如统一源仓库、统一垃圾收集器接口、统一即时编译器接口等，这些都将会是对未来Java发展大有裨益的改进。
- 2018年9月25日，JDK 11发布，这是一个LTS版本的JDK，包含17个JEP，其中有ZGC这样的革命性的垃圾收集器出现，也有把JDK 10中的类型推断加入Lambda语法这种可见的改进。
- 2019年2月，在JDK 12发布前夕，Oracle果然如之前宣布那样在六个月之后就放弃了对上一个版本OpenJDK的维护，RedHat同时从Oracle手上接过OpenJDK 8和OpenJDK 11的管理权利和维护职责。
- 2019年3月20日，JDK 12发布，只包含8个JEP，其中主要有Switch表达式、Java微测试套件（JMH）等新功能，最引人注目的特性无疑是加入了由RedHat领导开发的Shenandoah垃圾收集器。Shenandoah作为首个由非Oracle开发的垃圾收集器，其目标又与Oracle在JDK 11中发布的ZGC几乎完全一致，两者天生就存在竞争。Oracle马上用实际行动抵制了这个新收集器，在JDK 11发布时才说应尽可能保证OracleJDK和OpenJDK的兼容一致，转眼就在OracleJDK 12里把Shenandoah的代码通过条件编译强行剔除掉，使其成为历史上唯一进入了OpenJDK发布清单，但在OracleJDK中无法使用的功能。
- 2020年 3 月 17 日，JDK14 正式 GA(General Available)，新增了Records、switch表达式（JDK 12和JDK 13中的预览特性，现在正式使用）、文本块等新特性。在JVM方面，弃用Parallel Scavenge和SerialOld GC组合、删除CMS垃圾回收器。

2.3、各种JVM虚拟机

Java虚拟机是java运行的基石，不同的虚拟机对于java运行有着非常重要的影响，我们熟知的虚拟机有HotSpot、JRockit还有IBM J9虚拟机，在java发展史中除了这三个知名的虚拟机外，还有一些其他的虚拟机，下面我们一起来了解下。

2.3.1、Classic VM与Exact VM

1996年1月23日，Sun发布JDK 1.0，世界上第一款商用Java虚拟机Classic VM发布。

这款虚拟机只能使用纯解释器方式来执行Java代码，如果要使用即时编译器那就必须进行外挂，但是假如外挂了即时编译器的话，即时编译器就会完全接管虚拟机的执行系统，解释器便不能再工作了。

在JDK 1.2时，曾在Solaris平台上发布过一款名为Exact VM的虚拟机，它的编译执行系统已经具备现代高性能虚拟机雏形，如热点探测、两级即时编译器、编译器与解释器混合工作模式等。

虽然Exact VM的技术相对Classic VM来说先进了许多，但是它的命运显得十分英雄气短，在商业应用上只存在了很短暂的时间就被外部引进的HotSpot VM所取代，甚至还没有来得及发布Windows和Linux平台下的商用版本。

2.3.2、HotSpot VM

HotSpot VM 是Sun/OracleJDK和OpenJDK中的默认Java虚拟机，也是目前使用范围最广的Java虚拟机。在最初并非由Sun公司所开发，而是由一家名为“Longview Technologies”的小公司设计的。HotSpot既继承了Sun之前两款商用虚拟机的优点，也有许多自己新的技术优势，如它名称中的HotSpot指的就是它的热点代码探测技术。

2006年，Sun陆续将SunJDK的各个部分在GPLv2协议下开放了源码，形成了Open-JDK项目，其中当然也包括HotSpot虚拟机。HotSpot从此成为Sun/OracleJDK和OpenJDK两个实现极度接近的JDK项目的共同虚拟机。

Oracle收购Sun以后，建立了HotRockit项目来把原来BEA JRockit中的优秀特性融合到HotSpot之中。到了2014年的JDK 8时期，里面的HotSpot就已是两者融合的结果，HotSpot在这个过程里移除掉永久代，吸收了JRockit的Java Mission Control监控工具等功能。

2.3.3、BEA JRockit JVM

JRockit虚拟机曾经号称是“世界上速度最快的Java虚拟机”，它是BEA在2002年从Appeal Virtual Machines公司收购获得的Java虚拟机。

BEA将其发展为一款专门为服务器硬件和服务端应用场景高度优化的虚拟机，由于专注于服务端应用，它可以不太关注于程序启动速度，因此JRockit内部不包含解释器实现，全部代码都靠即时编译器编译后执行。

除此之外，JRockit的垃圾收集器和Java Mission Control故障处理套件等部分的实现，在当时众多的Java虚拟机中也处于领先水平。

JRockit随着BEA被Oracle收购，现已不再继续发展，永远停留在R28版本，这是JDK 6版JRockit的代号。

2.3.4、IBM J9 VM

IBM J9虚拟机并不是IBM公司唯一的Java虚拟机，不过目前IBM主力发展无疑就是J9。

与BEA JRockit只专注于服务端应用不同，IBM J9虚拟机的市场定位与HotSpot比较接近，它是一款在设计上全面考虑服务端、桌面应用，再到嵌入式的多用途虚拟机。

开发J9的目的是作为IBM公司各种Java产品的执行平台，在和IBM产品（如IBM WebSphere等）搭配以及在IBM AIX和z/OS这些平台上部署Java应用。

从2016年起，IBM逐步将OMR项目和J9虚拟机进行开源，完全开源后便将它们捐献给了Eclipse基金会管理，并重新命名为Eclipse OMR和OpenJ9。

2.3.5、Apache Harmony VM

Apache Harmony是一个Apache软件基金会旗下以Apache License协议开源的实际兼容于JDK 5和JDK 6的Java程序运行平台，它含有自己的虚拟机和Java类库API，用户可以在上面运行Eclipse、Tomcat、Maven等常用的Java程序。但是，它并没有通过TCK（Technology Compatibility Kit）认证。

当Sun公司把自家的JDK开源形成OpenJDK项目之后，Apache Harmony开源的优势被极大地抵消，以至于连Harmony项目的最大参与者IBM公司也宣布辞去Harmony项目管理主席的职位，转而参与OpenJDK的开发。

虽然Harmony没有真正地被大规模商业运用过，但是它的许多代码（主要是Java类库部分的代码）被吸纳进IBM的JDK 7实现以及Google Android SDK之中，尤其是对Android的发展起了很大推动作用。

2.3.6、Microsoft JVM

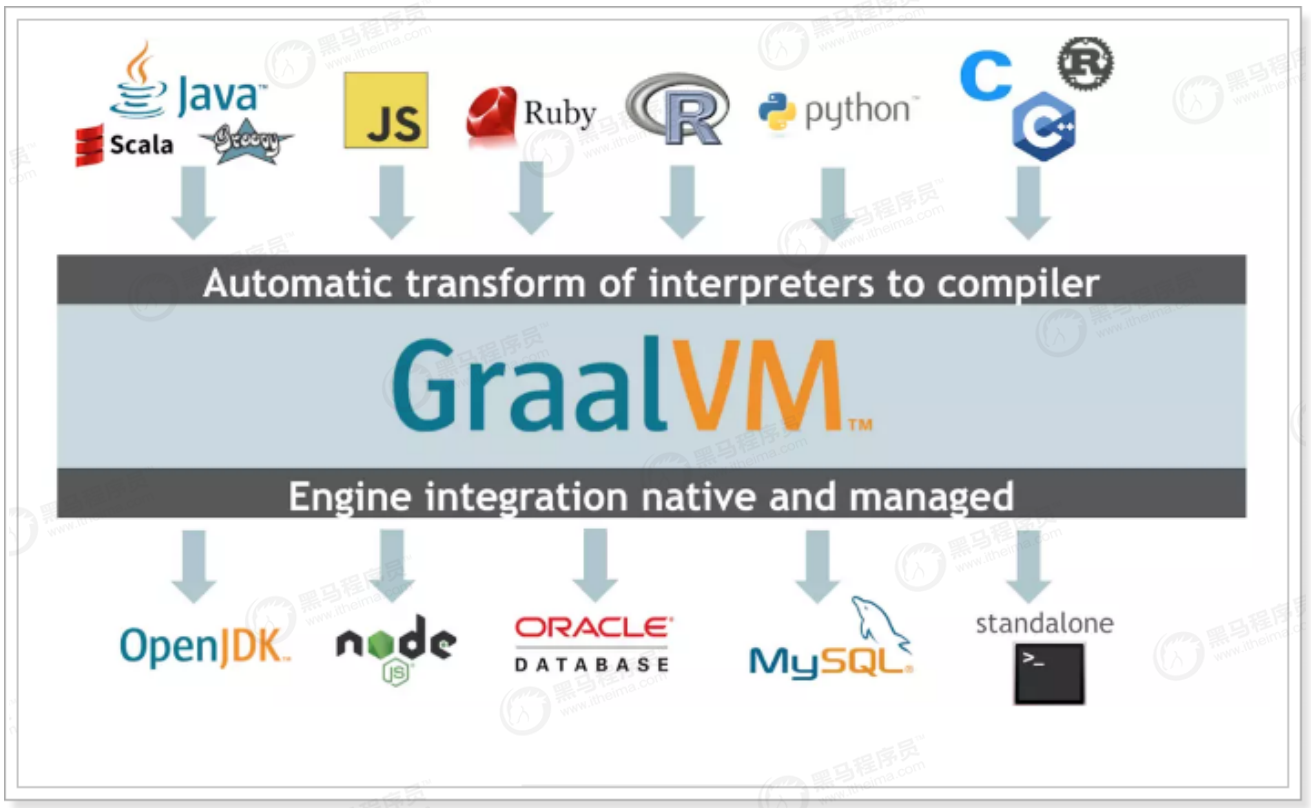
在Java语言诞生的初期，它的主要应用之一是在浏览器中运行Java Applets程序，微软为了在Internet Explorer 3浏览器中支持Java Applets应用而开发了自己的Java虚拟机，虽然这款虚拟机只有Windows平台的版本，“一次编译，到处运行”根本无从谈起，但却是当时Windows系统下性能最好的Java虚拟机。

在1997年10月，Sun公司正式以侵犯商标、不正当竞争等罪名控告微软，在随后对微软公司的垄断调查之中，这款虚拟机也曾作为证据之一被呈送法庭。官司的结果是微软向Sun公司（最终微软因垄断赔偿给Sun公司的总金额高达10亿美元）赔偿2000万美金，承诺终止其Java虚拟机的发展，并逐步在产品中移除Java虚拟机相关功能。

2.3.7、Graal VM

2018年4月，Oracle Labs新公开了一项黑科技：Graal VM，从它的口号“Run Programs Faster Anywhere”就能感觉到一颗蓬勃的野心。

Graal VM 是一个在HotSpot虚拟机基础上增强而成的跨语言全栈虚拟机，可以作为“任何语言”的运行平台使用，这里“任何语言”包括了Java、Scala、Groovy、Kotlin等基于Java虚拟机之上的语言，还包括了C、C++、Rust等基于LLVM的语言，同时支持其他像JavaScript、Ruby、Python和R语言等。

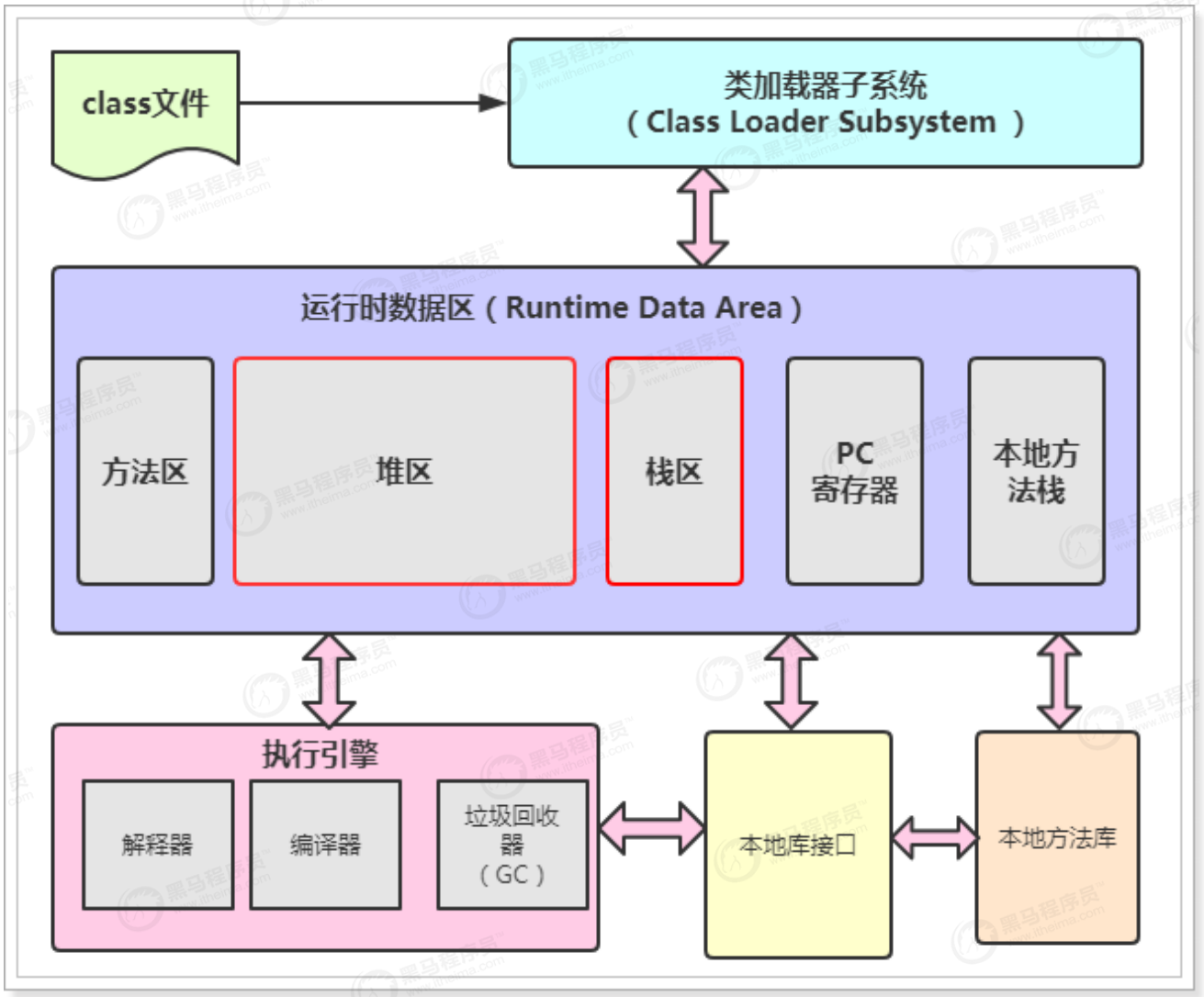


3、JVM虚拟机内存管理

对于Java程序员来说，在虚拟机自动内存管理机制的帮助下，不再需要为每一个new操作去写配对的delete/free代码，不容易出现内存泄漏和内存溢出问题，看起来由虚拟机管理内存一切都很美好。

不过，也正是因为Java程序员把控制内存的权力交给了Java虚拟机，一旦出现内存泄漏和溢出方面的问题，如果不了解虚拟机是怎样使用内存的，那排查错误、修正问题将会成为一项异常艰难的工作。

3.1、JVM整体架构



由上面的图可以看出，JVM虚拟机中主要是由三部分构成，分别是类加载子系统、运行时数据区、执行引擎。

> **类加载子系统**

Java虚拟机把描述类的数据从Class文件加载到内存，并对数据进行校验、转换解析和初始化，最终形成可以被虚拟机直接使用的Java类型。

> **运行时数据区**

Java虚拟机在执行Java程序的过程中会把它所管理的内存划分为若干个不同的数据区域。

这些区域有各自的用途，以及创建和销毁的时间，有的区域随着虚拟机进程的启动而一直存在，有些区域则是依赖用户线程的启动和结束而建立和销毁。

> **执行引擎**

执行引擎用于执行JVM字节码指令，主要有两种方式，分别是解释执行和编译执行，区别在于，解释执行是在执行时翻译成虚拟机指令执行，而编译执行是在执行之前先进行编译再执行。

解释执行启动快，执行效率低。编译执行，启动慢，执行效率高。

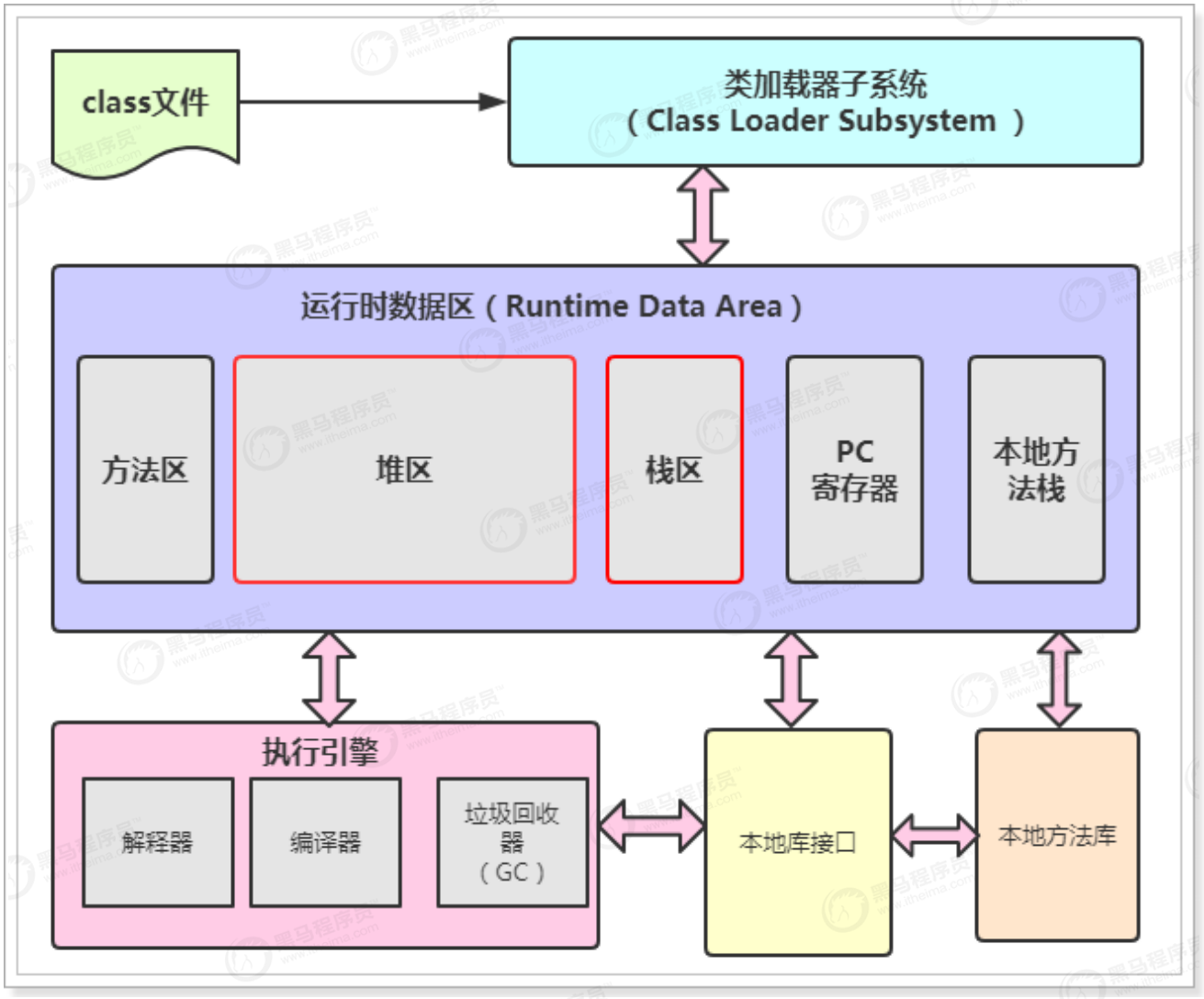
垃圾回收器就是自动管理运行数据区的内存，将无用的内存占用进行清除，释放内存资源。

> 本地方法库、本地库接口

在jdk的底层中，有一些实现是需要调用本地方法完成的（使用c或c++写的方法），就是通过本地库接口调用完成的。比如：System.currentTimeMillis()方法。

3.2、运行时数据区

运行时数据区是jvm中最为重要的部分，也是我们在调优时需要重点关注的区域，下面我们一起了解下这部分区域中的具体内容。



根据《Java虚拟机规范》中的规定，在运行时数据区将内存分为方法区（Method Area）、Java堆区（Java Heap）、Java虚拟机栈（Java Virtual Machine Stack）、程序计数器（Program Counter Register）、本地方法栈（Native Method Stacks）。

3.2.1、程序计数器

程序计数器（Program Counter Register）是一块较小的内存空间，它可以看作是当前线程所执行的字节码的行号指示器。

字节码解释器工作时就是通过改变这个计数器的值来选取下一条需要执行的字节码指令，它是程序控制流的指示器，分支、循环、跳转、异常处理、线程恢复等基础功能都需要依赖这个计数器来完成。

由于Java虚拟机的多线程是通过线程轮流切换、分配处理器执行时间的方式来实现的，在任何一个确定的时刻，一个处理器（对于多核处理器来说是一个内核）都只会执行一条线程中的指令。因此，为了线程切换后能恢复到正确的执行位置，每条线程都需要有一个独立的程序计数器，各条线程之间计数器互不影响，独立存储，我们称这类内存区域为“线程私有”的内存。

3.2.1.1、编写代码

```
package cn.itcast.jvm;

public class Test1 {

    public int add(){
        int a = 1;
        int b = 2;
        int c = a + b;
        return c;
    }

    public static void main(String[] args) {
        Test1 test1 = new Test1();
        int result = test1.add();
        System.out.println(result);
    }
}
```

3.2.1.2、查询class的汇编代码

```
javap -c Test1.class > T.txt
```

```
Compiled from "Test1.java"
public class cn.itcast.jvm.Test1 {
  public cn.itcast.jvm.Test1();
    Code:
       0: aload_0
       1: invokespecial #1                  // Method java/lang/Object."<init>":()V
       4: return

  public int add();
    Code:
       0: iconst_1
       1: istore_1
       2: iconst_2
       3: istore_2
       4: iload_1
       5: iload_2
       6: iadd
       7: istore_3
       8: iload_3
       9: ireturn

  public static void main(java.lang.String[]);
    Code:
```

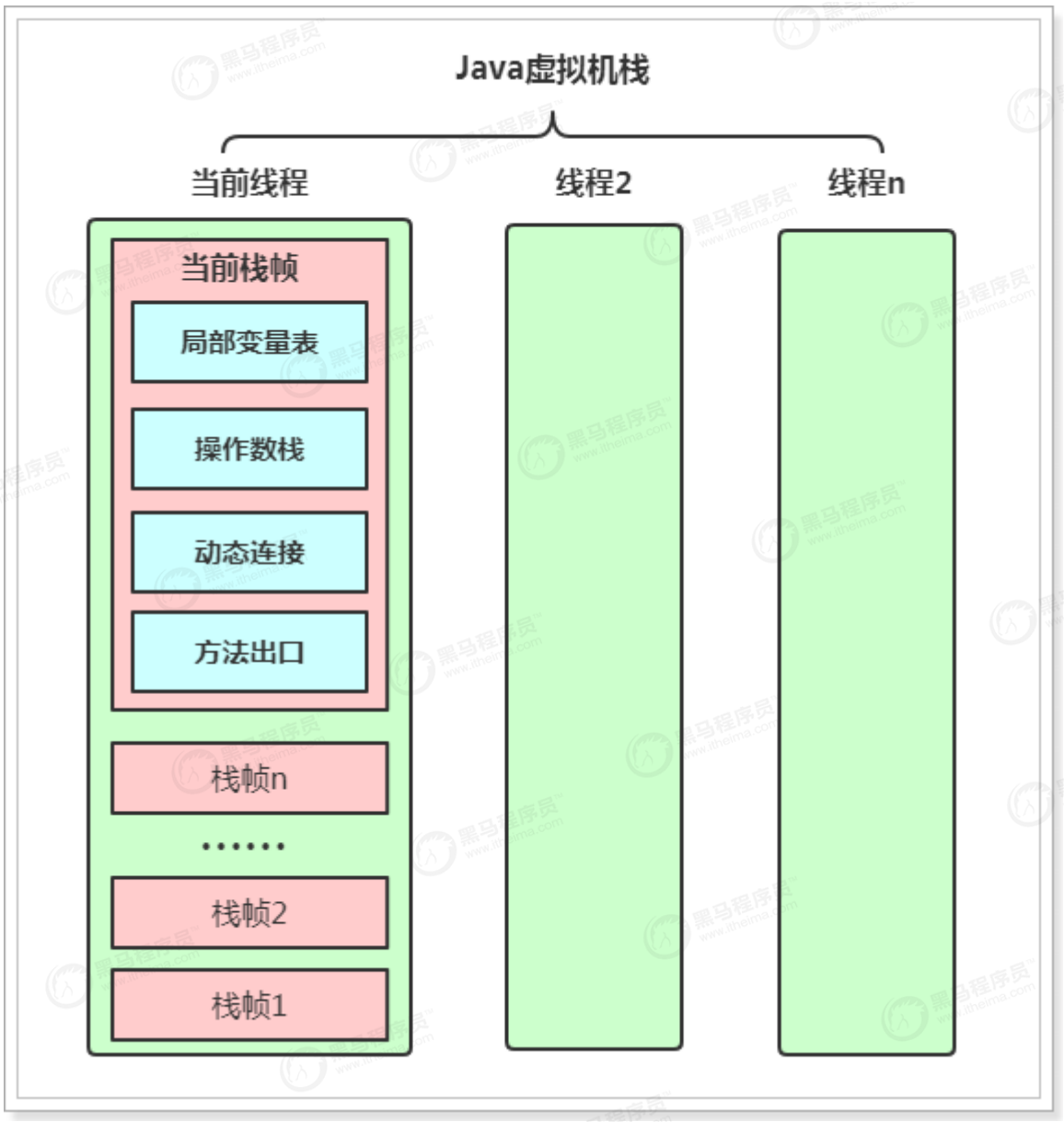
```
         0: new           #2                  // class cn/itcast/jvm/Test1
         3: dup
         4: invokespecial #3                  // Method "<init>":()V
         7: astore_1
         8: aload_1
         9: invokevirtual #4                  // Method add:()I
        12: istore_2
        13: getstatic     #5                  // Field
java/lang/System.out:Ljava/io/PrintStream;
        16: iload_2
        17: invokevirtual #6                  // Method java/io/PrintStream.println:(I)V
        20: return
}
```

可以看到将class文件中字节码进行反汇编，得到上面的代码，其中code所对应的编号就可以理解为计数器中所记录的执行编号。

3.2.2、Java虚拟机栈

与程序计数器一样，Java虚拟机栈也是线程私有的，它的生命周期与线程相同。

Java虚拟机栈描述的是Java方法执行的线程内存模型：每个方法被执行的时候，Java虚拟机都会同步创建一个栈帧，用于存储局部变量表、操作数栈、动态连接、方法出口等信息。每一个方法被调用直至执行完毕的过程，就对应着一个栈帧在虚拟机栈中从入栈到出栈的过程。



3.2.2.1、局部变量表

- 局部变量表是一组变量值的存储空间，用于存放方法参数和方法内部定义的局部变量。
- 在Class文件中，方法的Code属性的max_locals数据项中确定了该方法所需分配的局部变量表的最大容量。
- 该表以变量槽（Variable Slot）为最小单位，一个slot可以存放32位以内的数据，比如：boolean、byte、char、short、int、float等数据，如果存储long、double类型数据，需要占用2个solt。
- 虚拟机通过索引定位的方式使用局部变量表，索引值的范围是从0开始至局部变量表最大的变量槽数量。
- 如果访问的是32位数据类型的变量，索引N就代表了使用第N个变量槽，如果访问的是64位数据类型的变量，则说明会同时使用第N和N+1两个变量槽。
- 局部变量表中第0位索引的变量槽默认是用于传递方法所属对象实例的引用，在方法中可以通过关键字“this”来访问到这个隐含的参数。其余参数则按照参数表顺序排列，占用从1开始的局部变量槽，参数表分配完毕后，再根据方法体内部定义的变量顺序和作用域分配其余的变量槽。

3.2.2.2、操作数栈

- 操作数栈也常被称为操作栈，它是一个先进后出栈。
- 操作数栈的最大深度也在编译的时候被写入到Code属性的max_stacks数据项之中。

- 操作数栈的每一个元素都可以是包括long和double在内的任意Java数据类型。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。
- 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，也就是出栈和入栈操作。
- 操作数栈中元素的数据类型必须与字节码指令的序列严格匹配，例如iadd指令，不能出现一个long和一个float使用iadd命令相加的情况。

3.2.2.3、动态连接

- 每个栈帧都包含一个指向运行时常量池中该栈帧所属方法的引用，持有这个引用是为了支持方法调用过程中的动态连接。
- Class文件的常量池中存有大量的符号引用，字节码中的方法调用指令就以常量池里指向方法的符号引用作为参数。这些符号引用一部分会在类加载阶段或者第一次使用的时候就被转化为直接引用，这种转化被称为静态解析。另外一部分将在每一次运行期间都转化为直接引用，这部分就称为动态连接。

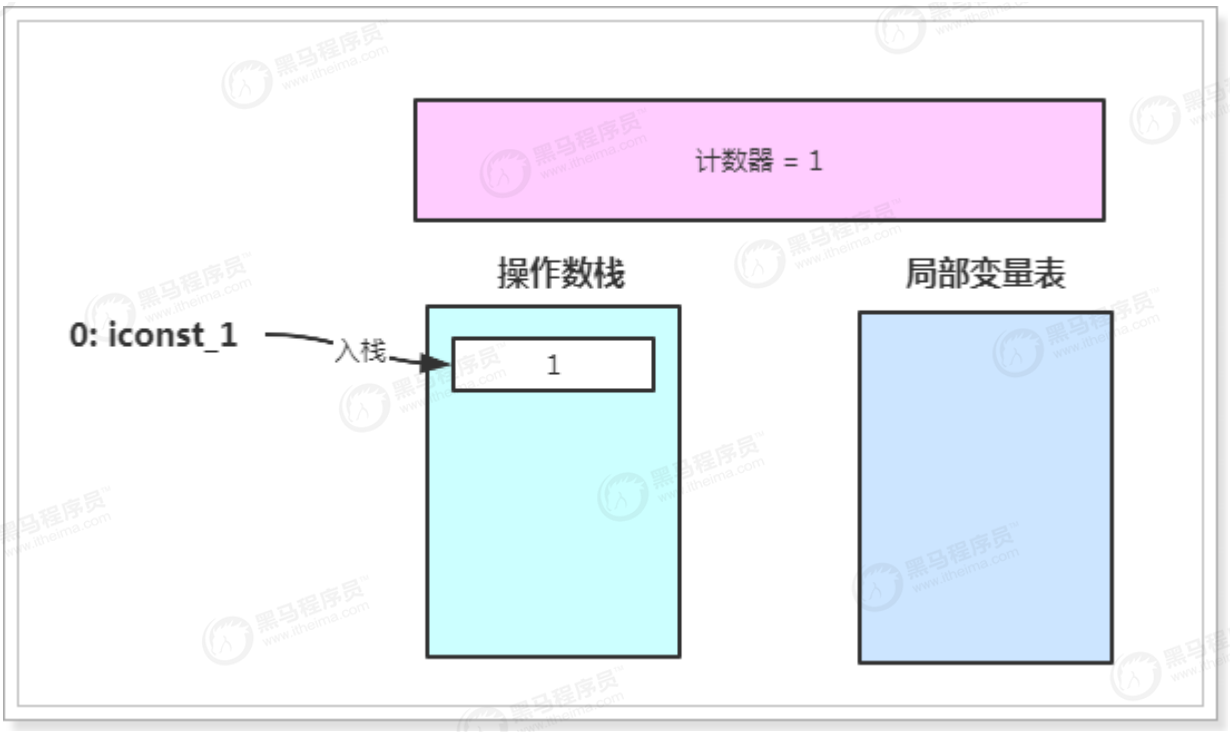
3.2.2.4、方法出口

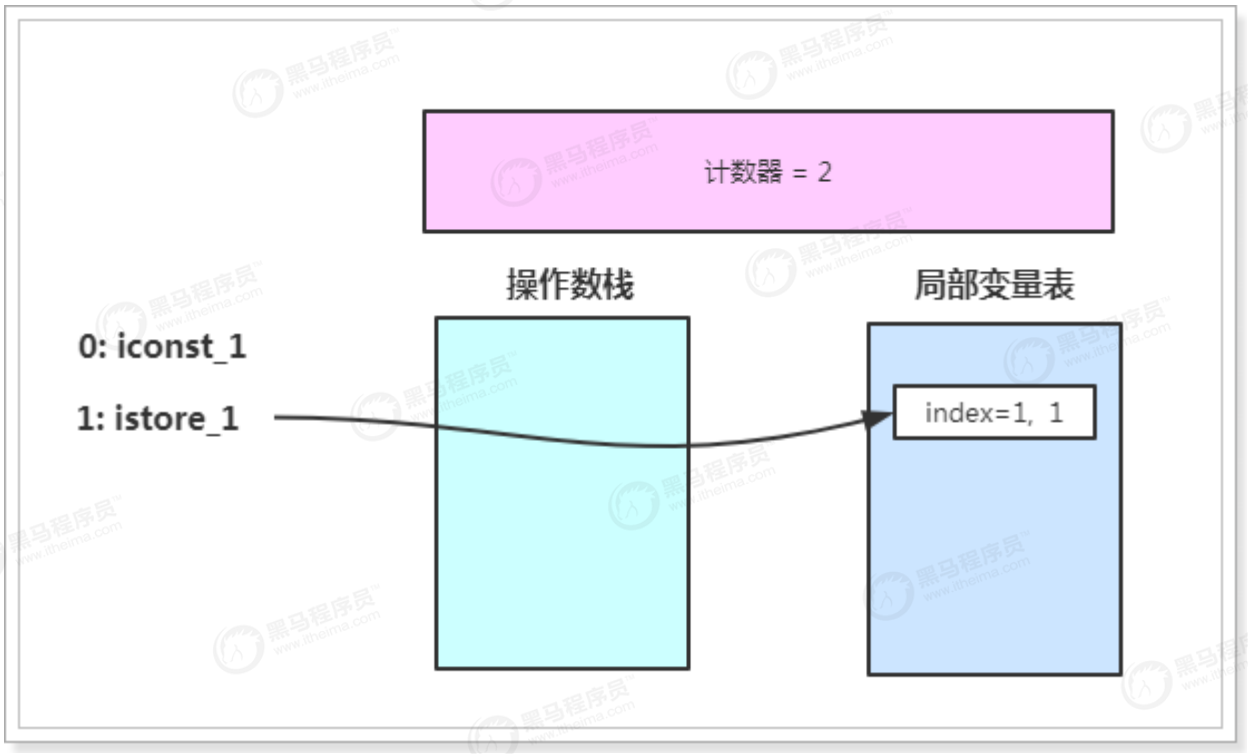
- 当一个方法开始执行后，只有两种方式退出这个方法。
- 第一种方式是执行引擎遇到任意一个方法返回的字节码指令，这时候可能会有返回值传递给上层的方法调用者，方法是否有返回值以及返回值的类型将根据遇到何种方法返回指令来决定，这种退出方法的方式称为“正常调用完成”。
- 另外一种退出方式是在方法执行的过程中遇到了异常，并且这个异常没有在方法体内得到妥善处理。无论是Java虚拟机内部产生的异常，还是代码中使用athrow字节码指令产生的异常，只要在本方法的异常表中没有搜索到匹配的异常处理器，就会导致方法退出，这种退出方法的方式称为“异常调用完成”。这种方法的返回是不会给它的上层调用者提供任何返回值的。
- 无论采用何种退出方式，在方法退出之后，都必须返回到最初方法被调用时的位置，程序才能继续执行，方法返回时可能需要在栈帧中保存一些信息，用来帮助恢复它的上层主调方法的执行状态。
- 方法退出的过程实际上等同于把当前栈帧出栈，因此退出时可能执行的操作有：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等。

3.2.2.5、实例

还是以Test1.class为例，我们看下add()方法执行中，虚拟机栈的执行。

int a = 1; 入栈再到局部变量表的操作：





> 注意：计数指向的是下一个执行行号，以及局部变量表下标从1开始。

3.2.3、本地方法栈

本地方法栈（Native Method Stacks）与虚拟机栈所发挥的作用是非常相似的，其区别只是虚拟机栈为虚拟机执行Java方法（也就是字节码）服务，而本地方法栈则是为虚拟机使用到的本地（Native）方法服务。

3.2.4、Java堆区

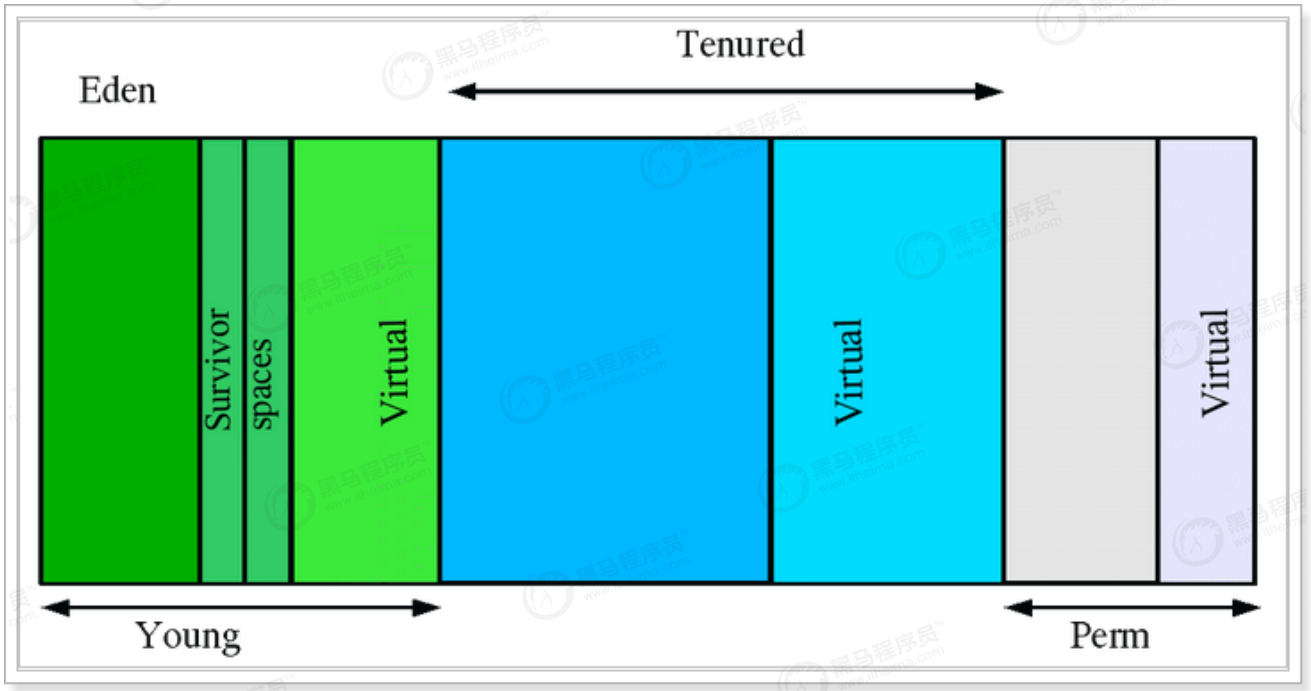
Java堆是被所有线程共享的一块内存区域，在虚拟机启动时创建。此内存区域的唯一目的就是存放对象实例，Java世界里“几乎”所有的对象实例都在这里分配内存。

需要注意的是，《Java虚拟机规范》并没有对堆进行细致的划分，所以对于堆的讲解要基于具体的虚拟机，我们以使用最多的HotSpot虚拟机为例进行讲解。

Java堆是垃圾收集器管理的内存区域，因此它也被称作“GC堆”，这就是我们做JVM调优的重点区域部分。

3.2.4.1、jdk1.7堆内存模型

jvm的内存模型在1.7和1.8有较大的区别，虽然本套课程是以1.8为例进行讲解，但是我们也是需要对1.7的内存模型有所了解，所以接下里，我们将先学习1.7再学习1.8的内存模型。



- Young 年轻区（代）

  Young区被划分为三部分，Eden区和两个大小严格相同的Survivor区，其中，Survivor区间中，某一时刻只有其中一个是被使用的，另外一个留做垃圾收集时复制对象用，在Eden区间变满的时候， GC就会将存活的对象移到空闲的Survivor区间中，根据JVM的策略，在经过几次垃圾收集后，任然存活于Survivor的对象将被移动到Tenured区间。

- Tenured 年老区

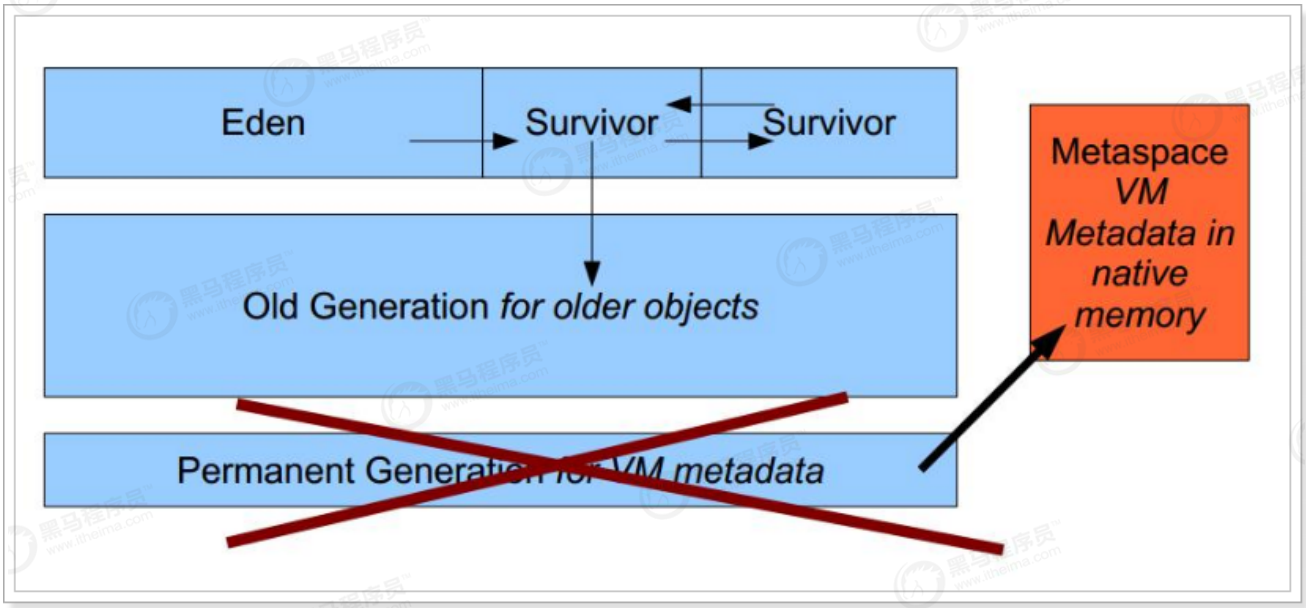
  Tenured区主要保存生命周期长的对象，一般是一些老的对象，当一些对象在Young复制转移一定的次数以后，对象就会被转移到Tenured区，一般如果系统中用了application级别的缓存，缓存中的对象往往会被转移到这一区间。

- Perm 永久区

  Perm代主要保存class,method,filed对象，这部份的空间一般不会溢出，除非一次性加载了很多的类，不过在涉及到热部署的应用服务器的时候，有时候会遇到java.lang.OutOfMemoryError : PermGen space 的错误，造成这个错误的很大原因就有可能是每次都重新部署，但是重新部署后，类的class没有被卸载掉，这样就造成了大量的class对象保存在了perm中，这种情况下，一般重新启动应用服务器可以解决问题。

- Virtual区：
  - 最大内存和初始内存的差值，就是Virtual区。

3.2.4.2、jdk1.8的堆内存模型



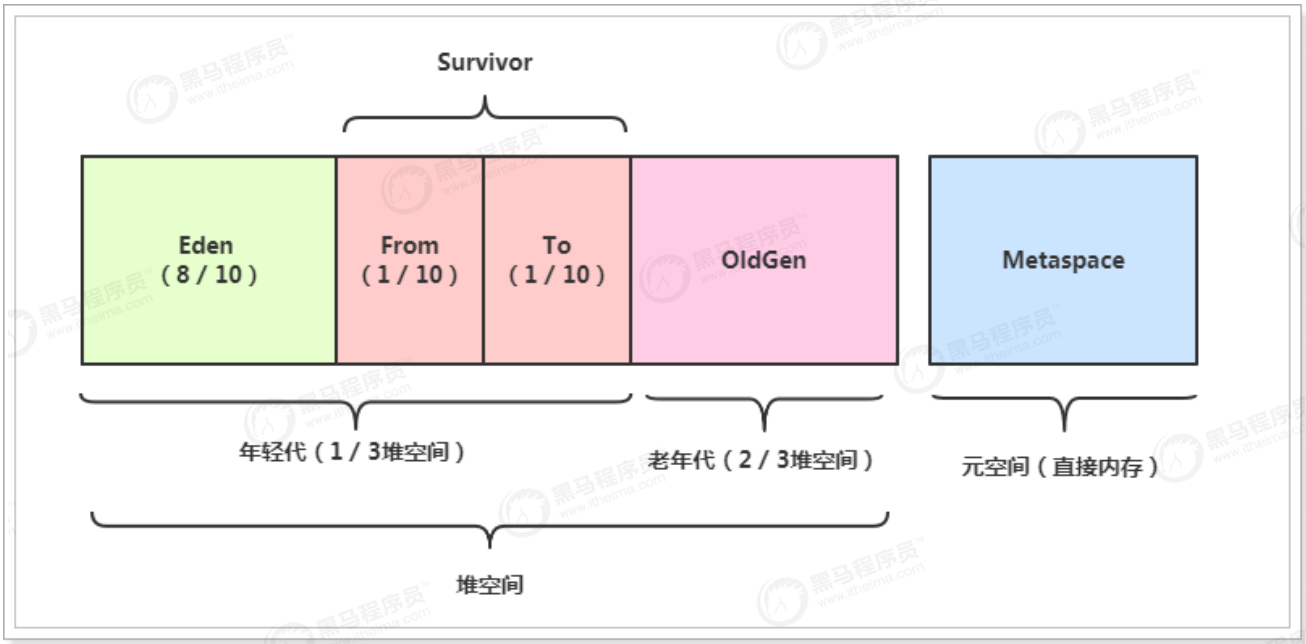
由上图可以看出，jdk1.8的内存模型是由2部分组成，年轻代 + 年老代。

年轻代：Eden + 2*Survivor

年老代：OldGen

在jdk1.8中变化最大的Perm区，用Metaspace（元数据空间）进行了替换。

需要特别说明的是：Metaspace所占用的内存空间不是在虚拟机内部，而是在本地内存空间中，这也是与1.7的永久代最大的区别所在。



上图所显示出的是默认状态下的空间分配情况，如果在没有指定堆内存大小时，默认初始堆内存为物理机内存的1/64，最大堆内存为物理机内存的1/4 或 1G。（JDK8的情况下）

> 官方说明：https://docs.oracle.com/javase/8/docs/technotes/guides/vm/gc-ergonomics.html

> 关于元空间需要注意的是，元空间会自动扩容，默认情况下不受限制，在实际中，经常忽略掉直接内存，使得各个内存区域总和大于物理内存限制，从而导致动态扩展时出现OutOfMemoryError异常。

3.2.4.3、为什么要废弃1.7中的永久区？

官网给出了解释：http://openjdk.java.net/jeps/122

```
This is part of the JRockit and Hotspot convergence effort. JRockit customers do not
need to configure the permanent generation (since JRockit does not have a permanent
generation) and are accustomed to not configuring the permanent generation.

移除永久代是为融合HotSpot JVM与 JRockit VM而做出的努力，因为JRockit没有永久代，不需要配置永久代。
```

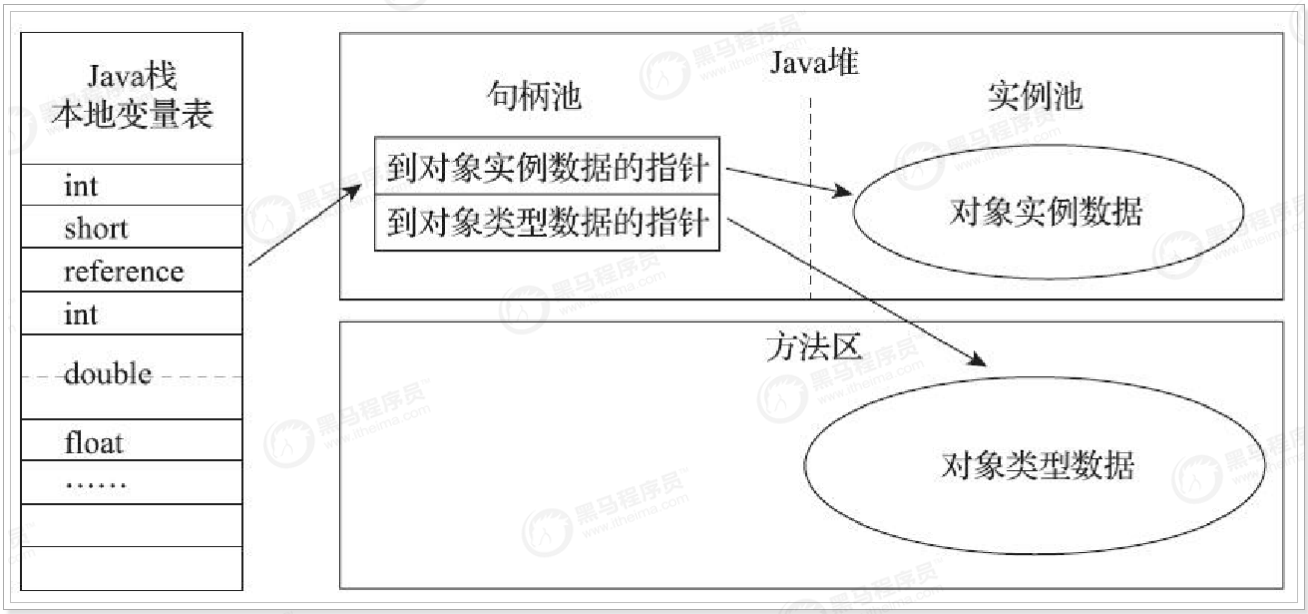
3.2.5、方法区

- 方法区（Method Area）与Java堆一样，是各个线程共享的内存区域，它用于存储已被虚拟机加载的类信息、常量、静态变量、即时编译器编译后的代码缓存等数据。
- 《Java虚拟机规范》中把方法区描述为堆的一个逻辑部分，它却有一个别名叫作“非堆”（Non-Heap），目的是与Java堆区分开来。
- JDK8之前将HotSpot虚拟机把收集器的分代设计扩展至方法区，所以可以将永久代看做是方法区，JDK8之后废弃永久代，用元空间来代替。

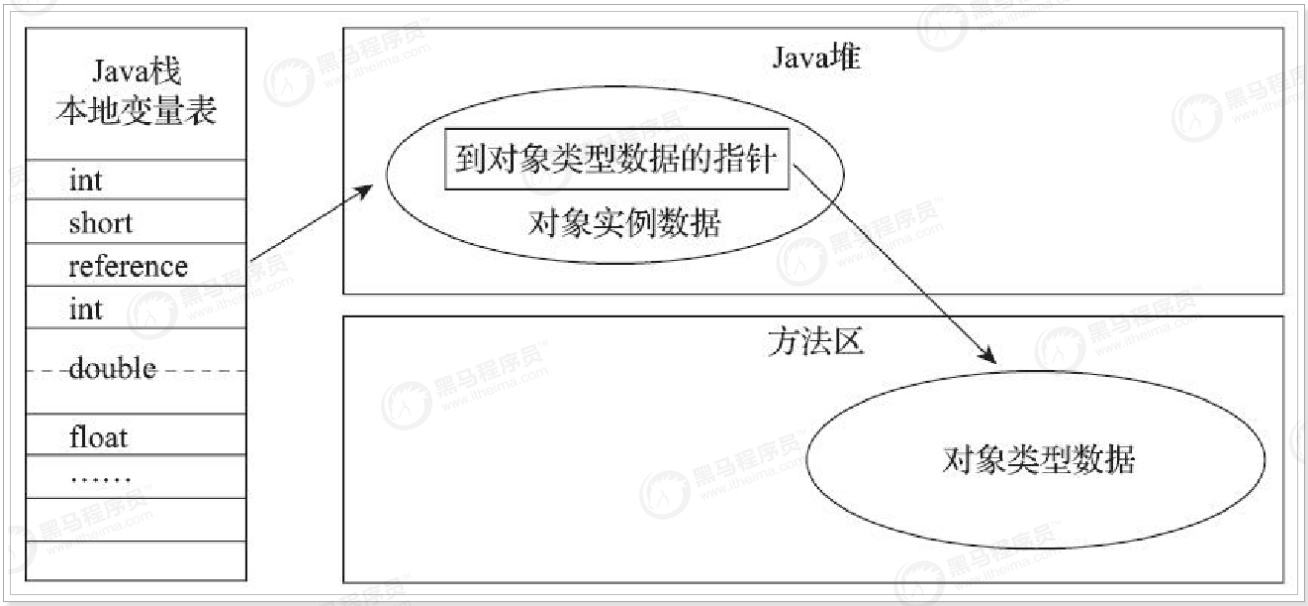
3.2.6、对象的访问

- Java程序会通过栈上的reference数据来操作堆上的具体对象。
- 主流的访问方式主要有使用句柄和直接指针两种：
  - 使用句柄访问
    - Java堆中将可能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具体的地址信息
  - 使用直接指针访问
    - Java堆中对象的内存布局就必须考虑如何放置访问类型数据的相关信息，reference中存储的直接就是对象地址，如果只是访问对象本身的话，就不需要多一次间接访问的开销

句柄访问：



指针访问：



- 使用句柄来访问的最大好处就是reference中存储的是稳定句柄地址，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本身不需要被修改。
- 使用直接指针来访问最大的好处就是速度更快，它节省了一次指针定位的时间开销。
- HotSpot虚拟机采用的是指针访问方式实现。

4、虚拟机性能相关工具

4.1、jvm的运行参数

在jvm中有很多的参数可以进行设置，这样可以让jvm在各种环境中都能够高效的运行。绝大部分的参数保持默认即可。

4.1.1、三种参数类型

jvm的参数类型分为三类，分别是：

- 标准参数
  - -help
  - -version
- -X参数（非标准参数）
  - -Xint
  - -Xcomp
- -XX参数（使用率较高）
  - -XX:newSize
  - -XX:+UseSerialGC

4.1.2、标准参数

jvm的标准参数，一般都是很稳定的，在未来的JVM版本中不会改变，可以使用java -help检索出所有的标准参数。

```
#打印帮助信息
java -help
#查看jvm版本
java -version

java version "1.8.0_141"
Java(TM) SE Runtime Environment (build 1.8.0_141-b15)
Java HotSpot(TM) 64-Bit Server VM (build 25.141-b15, mixed mode)
```

实例：通过-D设置系统属性参数

```
public class TestJVM {

    public static void main(String[] args) {
        String str = System.getProperty("str");
        if (str == null) {
            System.out.println("itcast");
        } else {
            System.out.println(str);
        }
    }
}
```

```
F:\t>javac TestJVM.java

F:\t>java TestJVM
itcast

F:\t>java -Dstr=123 TestJVM
123
```

4.1.3、-server与-client参数

可以通过-server或-client设置jvm的运行参数。

- 它们的区别是Server VM的初始堆空间会大一些，默认使用的是并行垃圾回收器，启动慢运行快。
- Client VM相对来讲会保守一些，初始堆空间会小一些，使用串行的垃圾回收器，它的目标是为了让JVM的启动速度更快，但运行速度会比Server模式慢些。
- JVM在启动的时候会根据硬件和操作系统自动选择使用Server还是Client类型的JVM。
- 32位操作系统
    - 如果是Windows系统，不论硬件配置如何，都默认使用Client类型的JVM。
    - 如果是其他操作系统上，机器配置有2GB以上的内存同时有2个以上CPU的话默认使用server模式，否则使用client模式。
- 64位操作系统
    - 只有server类型，不支持client类型。

```
[root@node01 test]# java -client -showversion TestJVM
java version "1.8.0_141"
Java(TM) SE Runtime Environment (build 1.8.0_141-b15)
Java HotSpot(TM) 64-Bit Server VM (build 25.141-b15, mixed mode)

itcast

[root@node01 test]# java -server -showversion TestJVM
java version "1.8.0_141"
Java(TM) SE Runtime Environment (build 1.8.0_141-b15)
Java HotSpot(TM) 64-Bit Server VM (build 25.141-b15, mixed mode)

itcast

#由于机器是64位系统，所以不支持client模式
```

4.2、-X参数

jvm的-X参数是非标准参数，在不同版本的jvm中，参数可能会有所不同，可以通过java -X查看非标准参数。

```
[root@node01 test]# java -X
    -Xmixed           混合模式执行 (默认)
    -Xint             仅解释模式执行
    -Xbootclasspath:<用 : 分隔的目录和 zip/jar 文件>
                      设置搜索路径以引导类和资源
    -Xbootclasspath/a:<用 : 分隔的目录和 zip/jar 文件>
                      附加在引导类路径末尾
    -Xbootclasspath/p:<用 : 分隔的目录和 zip/jar 文件>
                      置于引导类路径之前
    -Xdiag            显示附加诊断消息
    -Xnoclassgc       禁用类垃圾收集
    -Xincgc           启用增量垃圾收集
    -Xloggc:<file>    将 GC 状态记录在文件中 (带时间戳)
    -Xbatch           禁用后台编译
    -Xms<size>        设置初始 Java 堆大小
    -Xmx<size>        设置最大 Java 堆大小
    -Xss<size>        设置 Java 线程堆栈大小
    -Xprof            输出 cpu 配置文件数据
    -Xfuture          启用最严格的检查, 预期将来的默认值
    -Xrs              减少 Java/VM 对操作系统信号的使用 (请参阅文档)
    -Xcheck:jni       对 JNI 函数执行其他检查
    -Xshare:off       不尝试使用共享类数据
    -Xshare:auto      在可能的情况下使用共享类数据 (默认)
    -Xshare:on        要求使用共享类数据, 否则将失败。
    -XshowSettings    显示所有设置并继续
    -XshowSettings:all
                      显示所有设置并继续
    -XshowSettings:vm 显示所有与 vm 相关的设置并继续
    -XshowSettings:properties
                      显示所有属性设置并继续
    -XshowSettings:locale
                      显示所有与区域设置相关的设置并继续
```

-X 选项是非标准选项，如有更改，恕不另行通知。

4.2.1、-Xint、-Xcomp、-Xmixed

- 在解释模式(interpreted mode)下，-Xint标记会强制JVM执行所有的字节码，当然这会降低运行速度，通常低10倍或更多。
- -Xcomp参数与它（-Xint）正好相反，JVM在第一次使用时会把所有的字节码编译成本地代码，从而带来最大程度的优化。
  - 然而，很多应用在使用-Xcomp也会有一些性能损失，当然这比使用-Xint损失的少，原因是-xcomp没有让JVM启用JIT编译器的全部功能。JIT编译器可以对是否需要编译做判断，如果所有代码都进行编译的话，对于一些只执行一次的代码就没有意义了。
- -Xmixed是混合模式，将解释模式与编译模式进行混合使用，由jvm自己决定，这是jvm默认的模式，也是推荐使用的模式。

示例：强制设置运行模式

```
#强制设置为解释模式
[root@node01 test]# java  -showversion -Xint TestJVM
java version "1.8.0_141"
Java(TM) SE Runtime Environment (build 1.8.0_141-b15)
Java HotSpot(TM) 64-Bit Server VM (build 25.141-b15, interpreted mode)

itcast


#强制设置为编译模式
[root@node01 test]# java  -showversion -Xcomp TestJVM
java version "1.8.0_141"
Java(TM) SE Runtime Environment (build 1.8.0_141-b15)
Java HotSpot(TM) 64-Bit Server VM (build 25.141-b15, compiled mode)

itcast
#注意：编译模式下，第一次执行会比解释模式下执行慢一些，注意观察。


#默认的混合模式
[root@node01 test]# java  -showversion TestJVM
java version "1.8.0_141"
Java(TM) SE Runtime Environment (build 1.8.0_141-b15)
Java HotSpot(TM) 64-Bit Server VM (build 25.141-b15, mixed mode)

itcast
```

4.3、-XX参数

-XX参数也是非标准参数，主要用于jvm的调优和debug操作。

-XX参数的使用有2种方式，一种是boolean类型，一种是非boolean类型：

- boolean类型
  - 格式：-XX:[+-]<name> 表示启用或禁用<name>属性

    - 如：-XX:+DisableExplicitGC 表示禁用手动调用gc操作，也就是说调用System.gc()无效
- 非boolean类型
    - 格式：-XX:<name>=<value> 表示<name>属性的值为<value>
    - 如：-XX:NewRatio=4 表示新生代和老年代的比值为1:4

用法：

```
[root@node01 test]# java -showversion -XX:+DisableExplicitGC TestJVM
java version "1.8.0_141"
Java(TM) SE Runtime Environment (build 1.8.0_141-b15)
Java HotSpot(TM) 64-Bit Server VM (build 25.141-b15, mixed mode)

itcast
```

4.4、-Xms与-Xmx参数

-Xms与-Xmx分别是设置jvm的堆内存的初始大小和最大大小。

-Xmx2048m：等价于-XX:MaxHeapSize，设置JVM最大堆内存为2048M。

-Xms512m：等价于-XX:InitialHeapSize，设置JVM初始堆内存为512M。

适当的调整jvm的内存大小，可以充分利用服务器资源，让程序跑的更快。

示例：

```
[root@node01 test]# java -Xms512m -Xmx2048m TestJVM
itcast
```

4.5、查看正在运行的jvm参数

如果想要查看正在运行的jvm就需要借助于jinfo命令查看。

首选需要启动一个java程序，在资料中提供了itcast-jvm-app-1.0-SNAPSHOT.jar程序，里面是一个spring boot程序，没有做任何实现。

启动这个程序：java -jar itcast-jvm-app-1.0-SNAPSHOT.jar

查找到这个应用的进程编号：

查询看运行参数：

```
F:\t> jinfo -flags 12076
Attaching to process ID 12076, please wait...
Debugger attached successfully.
Server compiler detected.
JVM version is 25.144-b01
Non-default VM flags: -XX:CICompilerCount=2 -XX:InitialHeapSize=201326592 -XX:Ma
xHeapSize=3221225472 -XX:MaxNewSize=1073741824 -XX:MinHeapDeltaBytes=524288 -XX:
NewSize=67108864 -XX:OldSize=134217728 -XX:+UseCompressedClassPointers -XX:+UseC
ompressedOops -XX:+UseFastUnorderedTimeStamps -XX:-UseLargePagesIndividualAlloca
tion -XX:+UseParallelGC
Command line:

#查看某一参数的值，用法：jinfo -flag <参数名> <进程id>
F:\t>jinfo -flag MaxHeapSize 12076
-XX:MaxHeapSize=3221225472
```

4.6、jstat

jstat命令可以查看堆内存各部分的使用量，以及加载类的数量。命令的格式如下：

jstat [-命令选项] [vmid] [间隔时间/毫秒] [查询次数]

4.6.1、查看class加载统计

```
F:\t>jstat -class 12076
Loaded  Bytes  Unloaded  Bytes     Time
  5962 10814.2        0     0.0        3.75
```

说明：

- Loaded：加载class的数量
- Bytes：所占用空间大小
- Unloaded：未加载数量
- Bytes：未加载占用空间
- Time：时间

4.6.2、查看编译统计

```
F:\t>jstat -compiler 12076
Compiled Failed Invalid   Time   FailedType FailedMethod
    3115      0       0     3.43          0
```

说明：

- Compiled：编译数量。
- Failed：失败数量
- Invalid：不可用数量
- Time：时间
- FailedType：失败类型
- FailedMethod：失败的方法

4.6.3、垃圾回收统计

```
F:\t>jstat -gc 12076
 S0C    S1C    S0U    S1U      EC       EU        OC         OU       MC     MU
   CCSC   CCSU   YGC     YGCT    FGC    FGCT     GCT
3584.0 6656.0 3412.1  0.0   180224.0 89915.4   61440.0     5332.1   27904.0 2626
7.3 3840.0 3420.8      6    0.036   1      0.026    0.062

#也可以指定打印的间隔和次数，每1秒中打印一次，共打印5次
F:\t>jstat -gc 12076 1000 5
 S0C    S1C    S0U    S1U      EC       EU        OC         OU       MC     MU
   CCSC   CCSU   YGC     YGCT    FGC    FGCT     GCT
3584.0 6656.0 3412.1  0.0   180224.0 89915.4   61440.0     5332.1   27904.0 2626
7.3 3840.0 3420.8      6    0.036   1      0.026    0.062
3584.0 6656.0 3412.1  0.0   180224.0 89915.4   61440.0     5332.1   27904.0 2626
7.3 3840.0 3420.8      6    0.036   1      0.026    0.062
3584.0 6656.0 3412.1  0.0   180224.0 89915.4   61440.0     5332.1   27904.0 2626
7.3 3840.0 3420.8      6    0.036   1      0.026    0.062
3584.0 6656.0 3412.1  0.0   180224.0 89915.4   61440.0     5332.1   27904.0 2626
7.3 3840.0 3420.8      6    0.036   1      0.026    0.062
3584.0 6656.0 3412.1  0.0   180224.0 89915.4   61440.0     5332.1   27904.0 2626
7.3 3840.0 3420.8      6    0.036   1      0.026    0.062
```

说明：

- S0C：第一个Survivor区的大小（KB）
- S1C：第二个Survivor区的大小（KB）
- S0U：第一个Survivor区的使用大小（KB）
- S1U：第二个Survivor区的使用大小（KB）
- EC：Eden区的大小（KB）
- EU：Eden区的使用大小（KB）
- OC：Old区大小（KB）
- OU：Old使用大小（KB）
- MC：方法区大小（KB）
- MU：方法区使用大小（KB）
- CCSC：压缩类空间大小（KB）
- CCSU：压缩类空间使用大小（KB）
- YGC：年轻代垃圾回收次数
- YGCT：年轻代垃圾回收消耗时间
- FGC：老年代垃圾回收次数
- FGCT：老年代垃圾回收消耗时间
- GCT：垃圾回收消耗总时间

5、实战：内存溢出的定位与分析

内存溢出在实际的生产环境中经常会遇到，比如，不断的将数据写入到一个集合中，出现了死循环，读取超大的文件等等，都可能会造成内存溢出。

如果出现了内存溢出，首先我们需要定位到发生内存溢出的环节，并且进行分析，是正常还是非正常情况，如果是正常的需求，就应该考虑加大内存的设置，如果是非正常需求，那么就要对代码进行修改，修复这个bug。

首先，我们得先学会如何定位问题，然后再进行分析。如何定位问题呢，我们需要借助于MAT工具进行定位分析。

接下来，我们模拟内存溢出的场景。

5.1、内存溢出与内存泄露

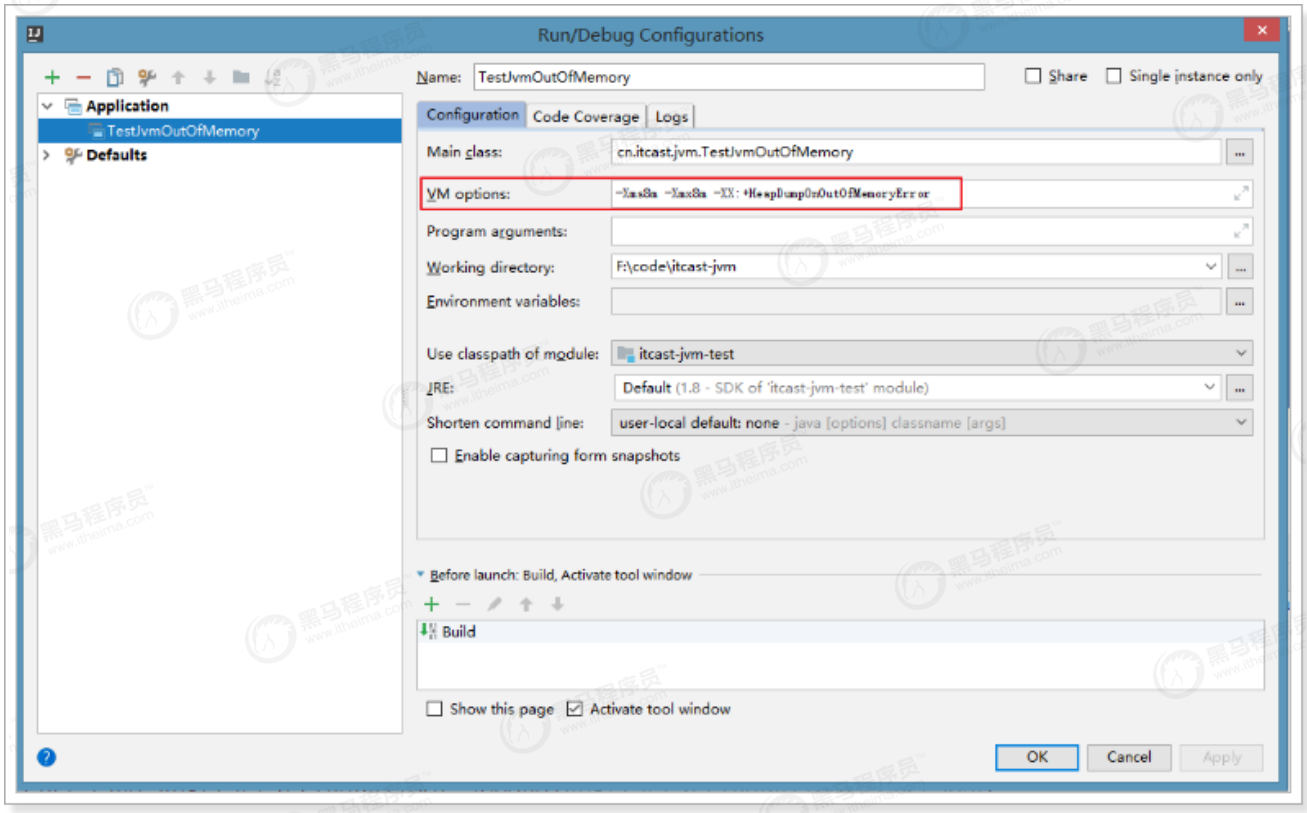
- 内存溢出，是指程序在申请内存时，没有足够的内存空间供其使用，出现out of memory；
- 内存泄露，是指程序在申请内存后，无法释放已申请的内存空间，一次内存泄露危害可以忽略，但内存泄露堆积后果很严重，无论多少内存,迟早会被占光。

5.2、模拟内存溢出

编写代码，向List集合中添加100万个字符串，每个字符串由1000个UUID组成。如果程序能够正常执行，最后打印ok。

```
package cn.itcast.jvm;

import java.util.ArrayList;
import java.util.List;
import java.util.UUID;

public class TestJvmOutOfMemory {

    public static void main(String[] args) {
        List<Object> list = new ArrayList<>();
        for (int i = 0; i < 10000000; i++) {
            String str = "";
            for (int j = 0; j < 1000; j++) {
                str += UUID.randomUUID().toString();
            }
            list.add(str);
        }
        System.out.println("ok");
    }
}
```

为了演示效果，我们将设置执行的参数，这里使用的是Idea编辑器。

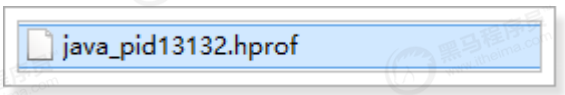



```
#参数如下：
-Xms8m -Xmx8m -XX:+HeapDumpOnOutOfMemoryError
```

5.3、运行测试

```
Dumping heap to java_pid13132.hprof ...
Exception in thread "main" java.lang.OutOfMemoryError: Java heap space
    at java.util.Arrays.copyOf(Arrays.java:3332)
    at
java.lang.AbstractStringBuilder.ensureCapacityInternal(AbstractStringBuilder.java:124)
    at java.lang.AbstractStringBuilder.append(AbstractStringBuilder.java:448)
    at java.lang.StringBuilder.append(StringBuilder.java:136)
    at cn.itcast.jvm.TestJvmOutOfMemory.main(TestJvmOutOfMemory.java:14)
Heap dump file created [8138796 bytes in 0.045 secs]
```

可以看到，当发生内存溢出时，会dump内存到java_pid13132.hprof中，该文件在项目的根目录下。

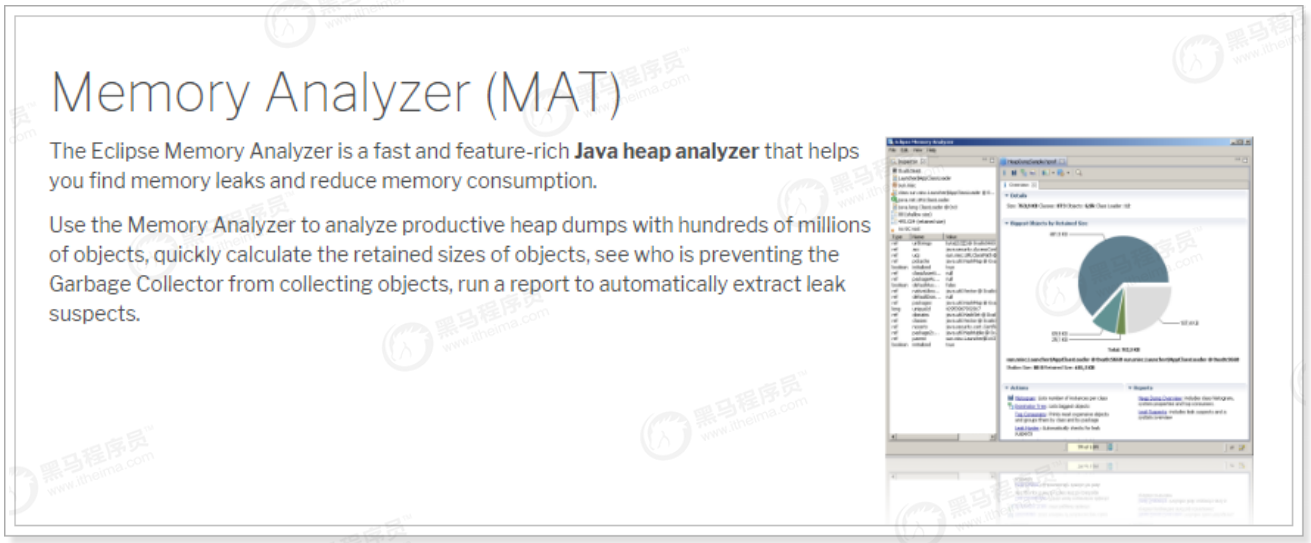


5.4、导入到MAT工具中进行分析

5.4.1、MAT工具介绍

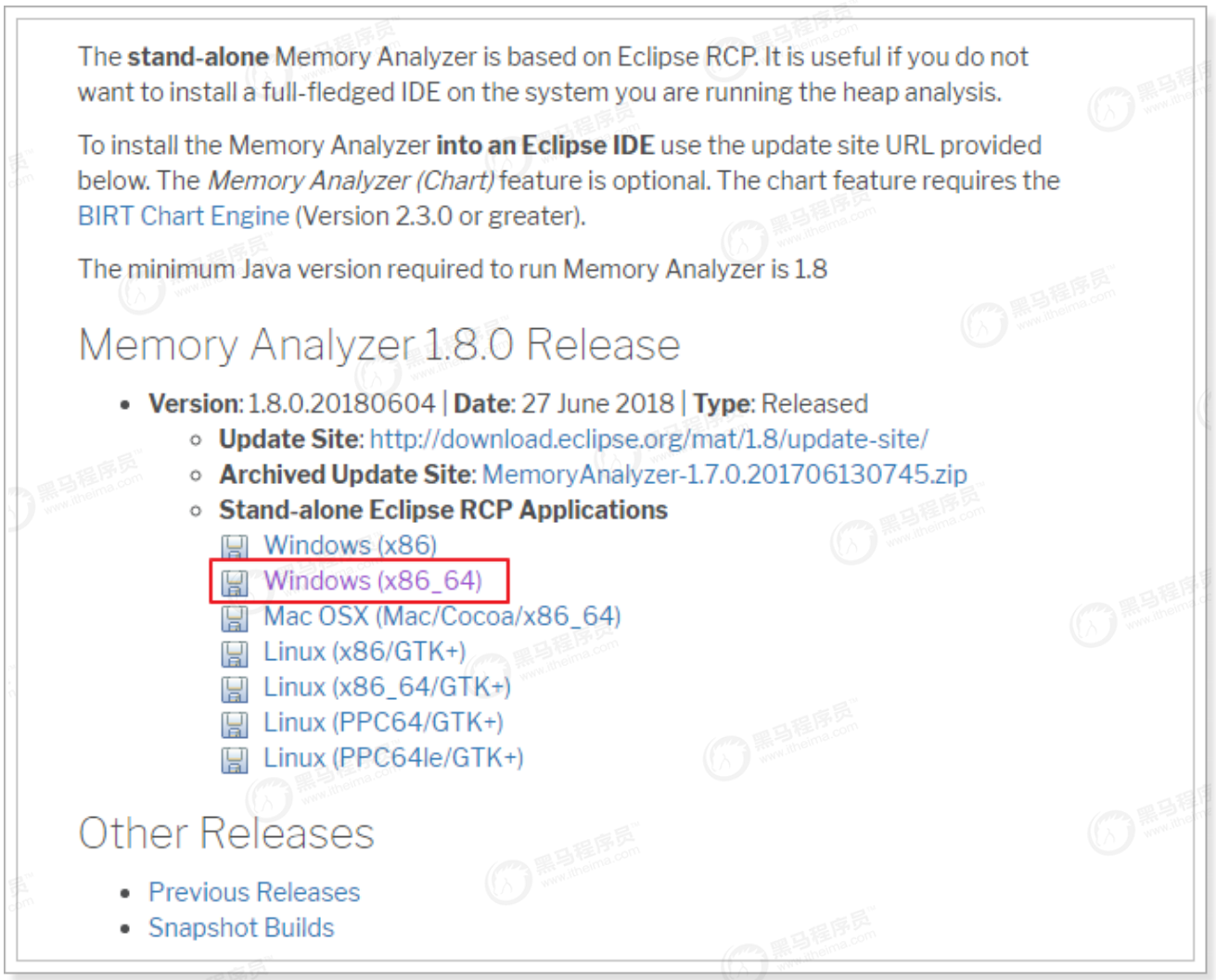
MAT(Memory Analyzer Tool)，一个基于Eclipse的内存分析工具，是一个快速、功能丰富的JAVA heap分析工具，它可以帮助我们查找内存泄漏和减少内存消耗。使用内存分析工具从众多的对象中进行分析，快速的计算出在内存中对象的占用大小，看看是谁阻止了垃圾收集器的回收工作，并可以通过报表直观的查看到可能造成这种结果的对象。

官网地址：https://www.eclipse.org/mat/

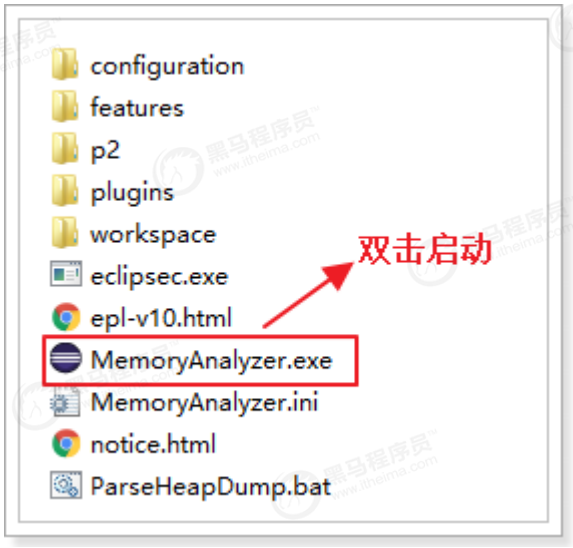


5.4.2、下载安装

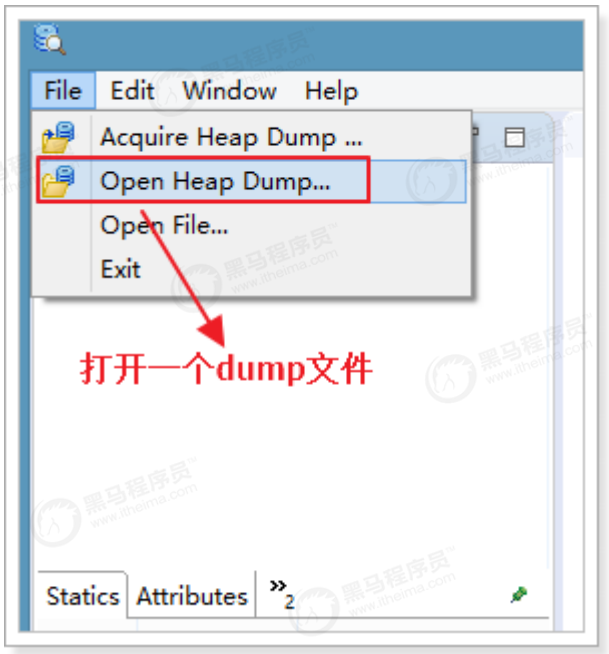
下载地址：https://www.eclipse.org/mat/downloads.php

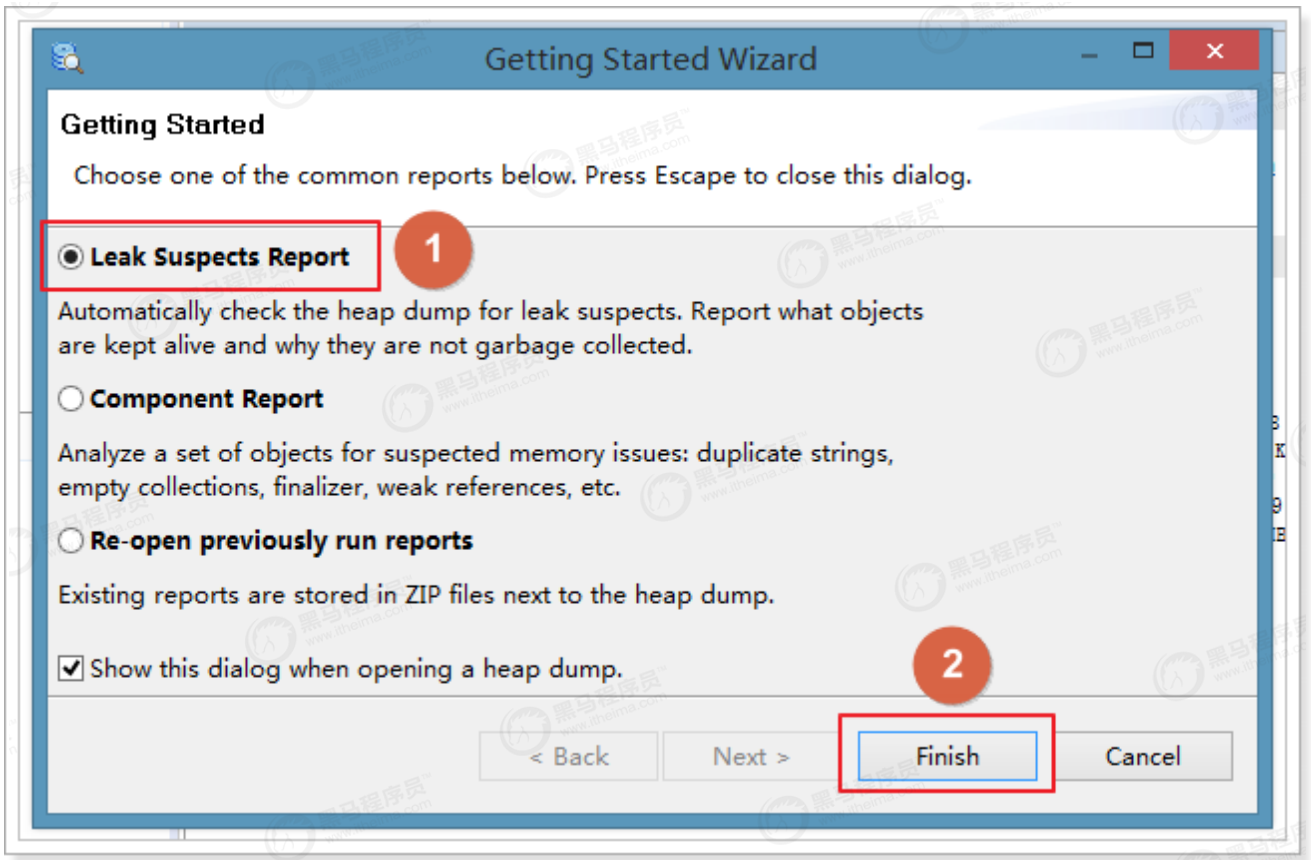


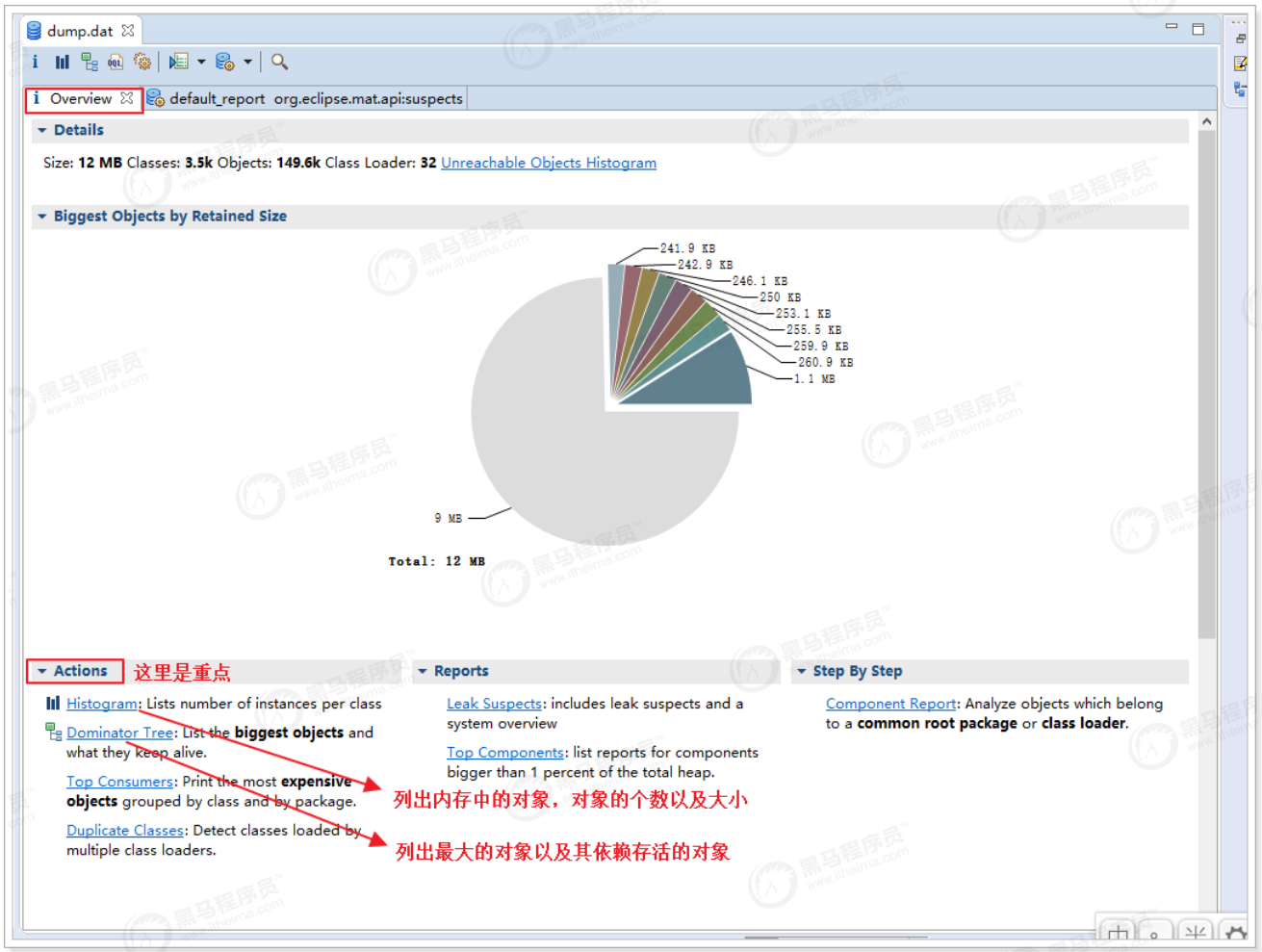
将下载得到的MemoryAnalyzer-1.8.0.20180604-win32.win32.x86_64.zip进行解压：

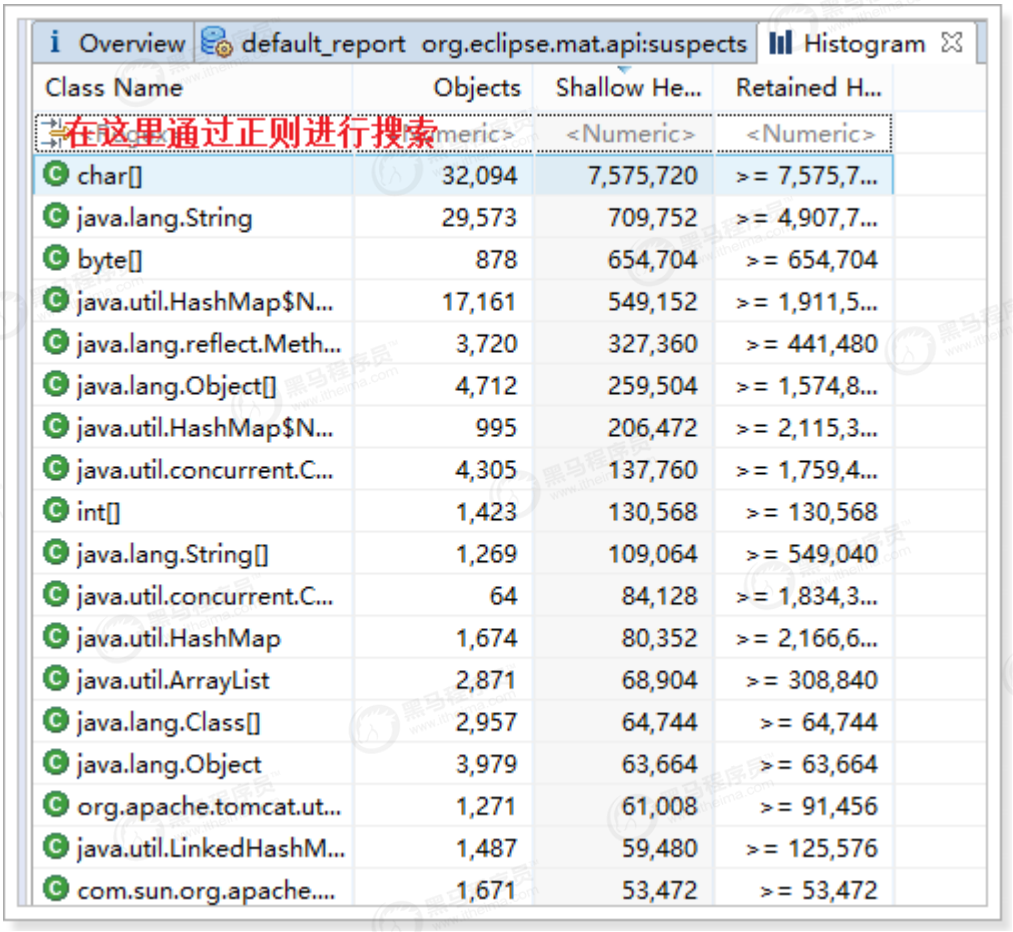

configuration
features
p2
plugins
workspace
eclipsec.exe
epl-v10.html
MemoryAnalyzer.exe
MemoryAnalyzer.ini
notice.html
ParseHeapDump.bat
双击启动

5.4.3、基本使用

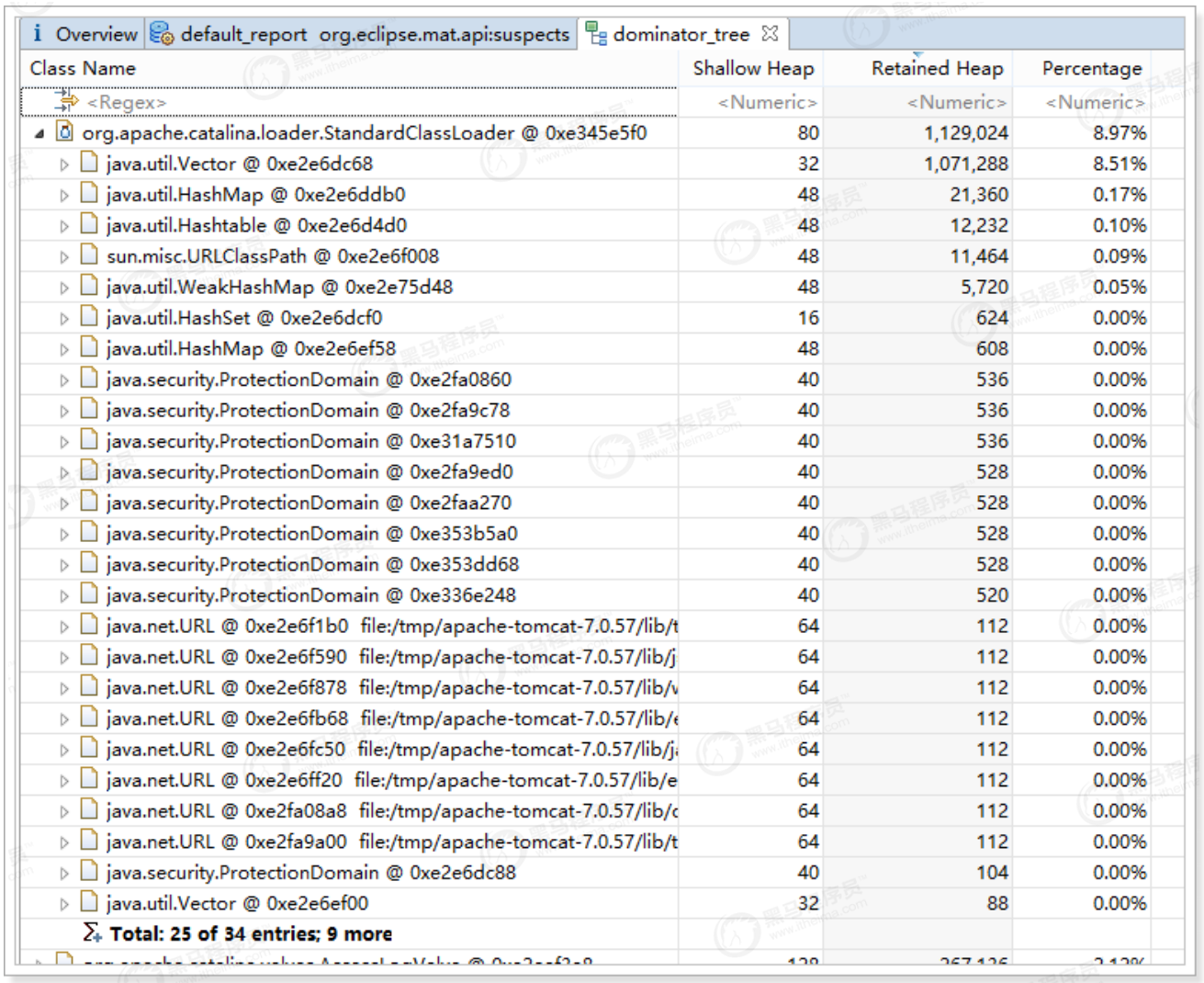

File
Edit
Window
Help
Acquire Heap Dump ...
Open Heap Dump...
Open File...
Exit
打开一个dump文件
Statics
Attributes


Getting Started Wizard
Getting Started
Choose one of the common reports below. Press Escape to close this dialog.
Leak Suspects Report
1
Automatically check the heap dump for leak suspects. Report what objects are kept alive and why they are not garbage collected.
Component Report
Analyze a set of objects for suspected memory issues: duplicate strings, empty collections, finalizer, weak references, etc.
Re-open previously run reports
Existing reports are stored in ZIP files next to the heap dump.
Show this dialog when opening a heap dump.
2
< Back
Next >
Finish
Cancel


dump.dat
Overview
default_report org.eclipse.mat.api:suspects
Details
Size: 12 MB Classes: 3.5k Objects: 149.6k Class Loader: 32 Unreachable Objects Histogram
Biggest Objects by Retained Size
241.9 KB
242.9 KB
246.1 KB
250 KB
253.1 KB
255.5 KB
259.9 KB
260.9 KB
1.1 MB
9 MB
Total: 12 MB
Actions
这里是重点
Reports
Step By Step
Histogram: Lists number of instances per class
Dominator Tree: List the biggest objects and what they keep alive.
Top Consumers: Print the most expensive objects grouped by class and by package.
Duplicate Classes: Detect classes loaded by multiple class loaders.
Leak Suspects: includes leak suspects and a system overview
Top Components: list reports for components bigger than 1 percent of the total heap.
Component Report: Analyze objects which belong to a common root package or class loader.
列出内存中的对象，对象的个数以及大小
列出最大的对象以及其依赖存活的对象


Overview
default_report org.eclipse.mat.api:suspects
Histogram
Class Name
Objects
Shallow He...
Retained H...
在这里通过正则进行搜索
<Numeric>
<Numeric>
<Numeric>
char[] 32,094 7,575,720 >= 7,575,7...
java.lang.String 29,573 709,752 >= 4,907,7...
byte[] 878 654,704 >= 654,704
java.util.HashMap$N... 17,161 549,152 >= 1,911,5...
java.lang.reflect.Meth... 3,720 327,360 >= 441,480
java.lang.Object[] 4,712 259,504 >= 1,574,8...
java.util.HashMap$N... 995 206,472 >= 2,115,3...
java.util.concurrent.C... 4,305 137,760 >= 1,759,4...
int[] 1,423 130,568 >= 130,568
java.lang.String[] 1,269 109,064 >= 549,040
java.util.concurrent.C... 64 84,128 >= 1,834,3...
java.util.HashMap 1,674 80,352 >= 2,166,6...
java.util.ArrayList 2,871 68,904 >= 308,840
java.lang.Class[] 2,957 64,744 >= 64,744
java.lang.Object 3,979 63,664 >= 63,664
org.apache.tomcat.ut... 1,271 61,008 >= 91,456
java.util.LinkedHashM... 1,487 59,480 >= 125,576
com.sun.org.apache.... 1,671 53,472 >= 53,472

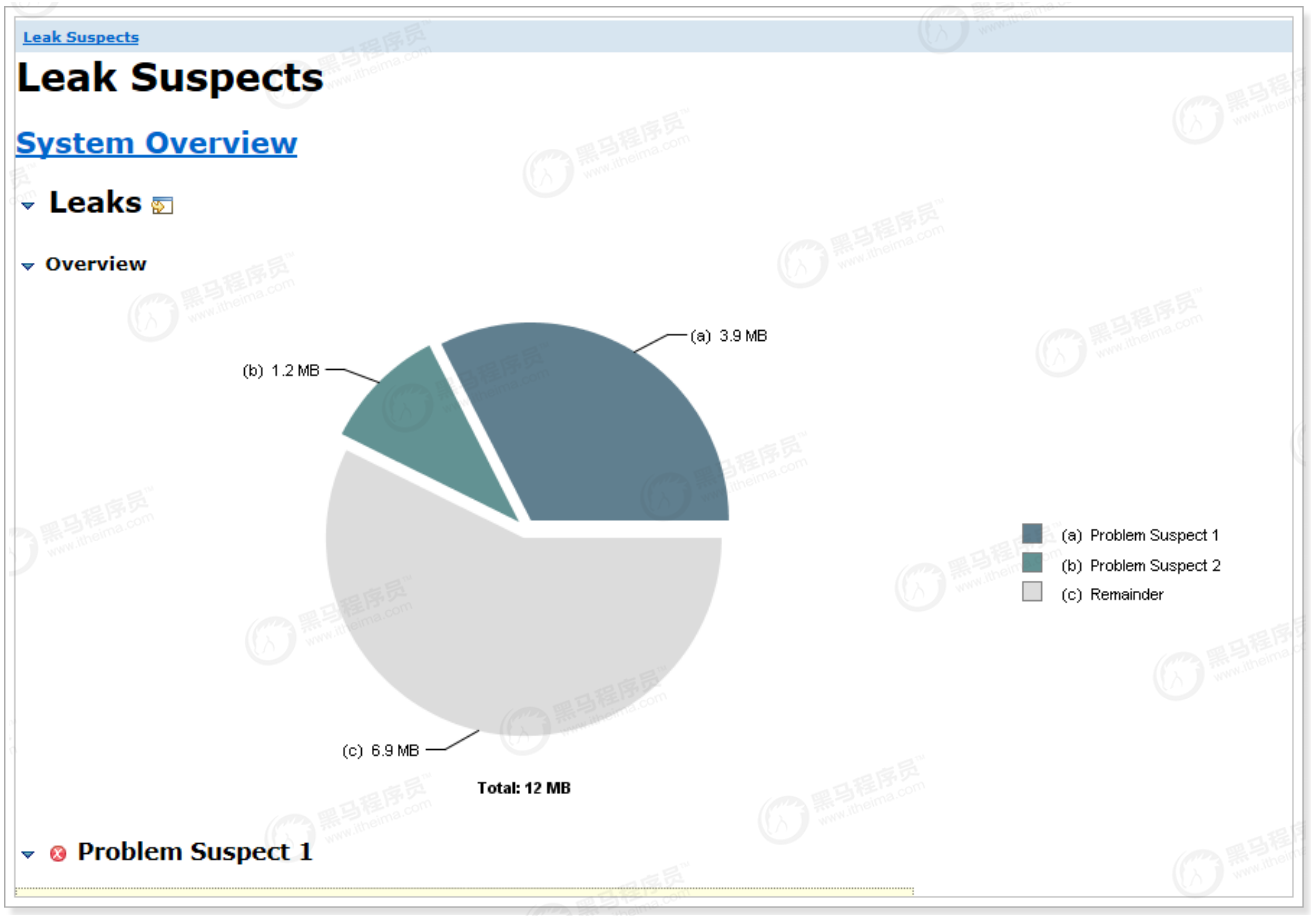
查看对象以及它的依赖：



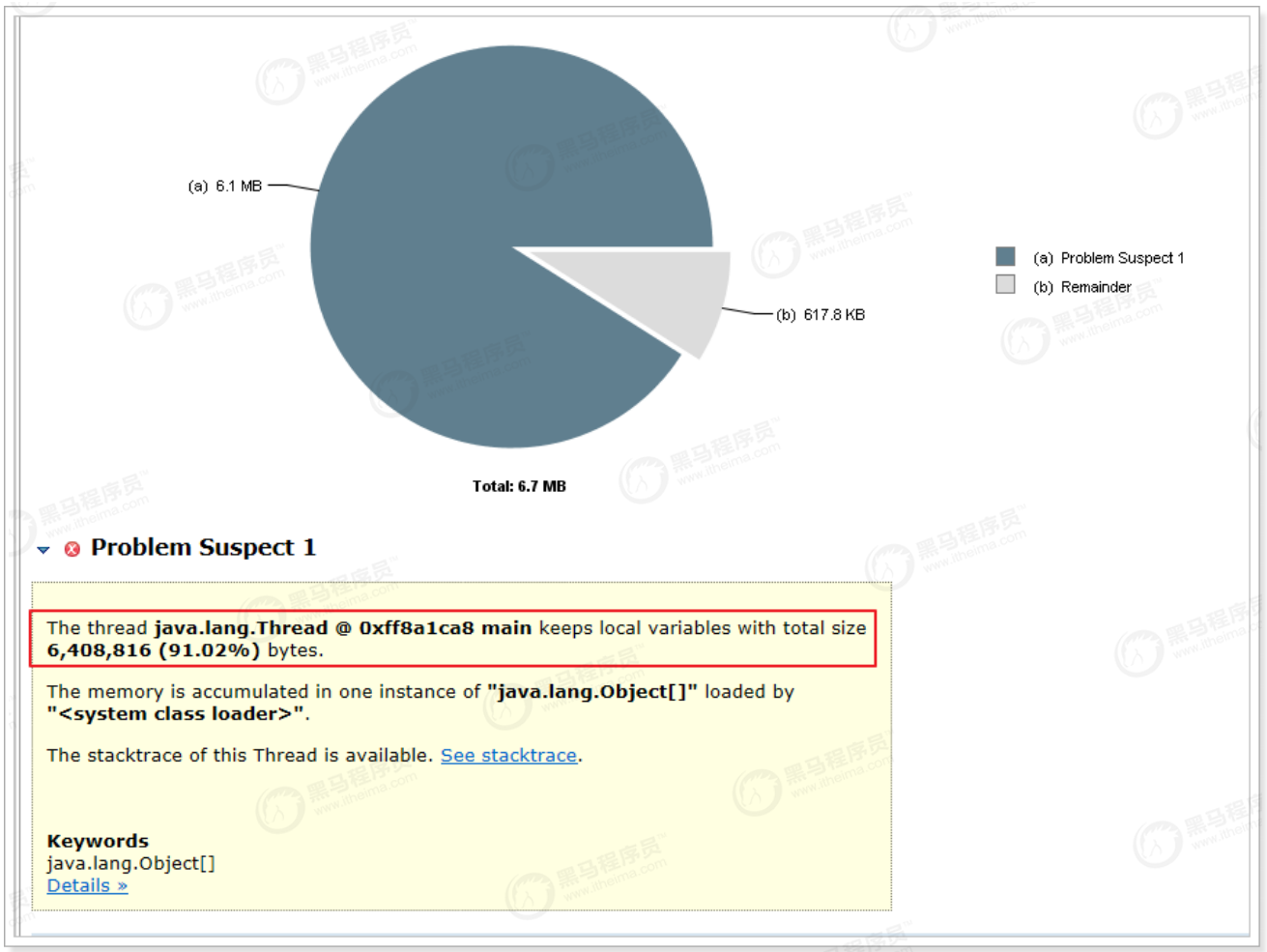
Overview | default_report org.eclipse.mat.api:suspects | dominator_tree

Class Name	Shallow Heap	Retained Heap	Percentage
<Regex>	<Numeric>	<Numeric>	<Numeric>
org.apache.catalina.loader.StandardClassLoader @ 0xe345e5f0	80	1,129,024	8.97%
java.util.Vector @ 0xe2e6dc68	32	1,071,288	8.51%
java.util.HashMap @ 0xe2e6ddb0	48	21,360	0.17%
java.util.Hashtable @ 0xe2e6d4d0	48	12,232	0.10%
sun.misc.URLClassPath @ 0xe2e6f008	48	11,464	0.09%
java.util.WeakHashMap @ 0xe2e75d48	48	5,720	0.05%
java.util.HashSet @ 0xe2e6dcf0	16	624	0.00%
java.util.HashMap @ 0xe2e6ef58	48	608	0.00%
java.security.ProtectionDomain @ 0xe2fa0860	40	536	0.00%
java.security.ProtectionDomain @ 0xe2fa9c78	40	536	0.00%
java.security.ProtectionDomain @ 0xe31a7510	40	536	0.00%
java.security.ProtectionDomain @ 0xe2fa9ed0	40	528	0.00%
java.security.ProtectionDomain @ 0xe2faa270	40	528	0.00%
java.security.ProtectionDomain @ 0xe353b5a0	40	528	0.00%
java.security.ProtectionDomain @ 0xe353dd68	40	528	0.00%
java.security.ProtectionDomain @ 0xe336e248	40	520	0.00%
java.net.URL @ 0xe2e6f1b0 file:/tmp/apache-tomcat-7.0.57/lib/t	64	112	0.00%
java.net.URL @ 0xe2e6f590 file:/tmp/apache-tomcat-7.0.57/lib/j	64	112	0.00%
java.net.URL @ 0xe2e6f878 file:/tmp/apache-tomcat-7.0.57/lib/v	64	112	0.00%
java.net.URL @ 0xe2e6fb68 file:/tmp/apache-tomcat-7.0.57/lib/e	64	112	0.00%
java.net.URL @ 0xe2e6fc50 file:/tmp/apache-tomcat-7.0.57/lib/j	64	112	0.00%
java.net.URL @ 0xe2e6ff20 file:/tmp/apache-tomcat-7.0.57/lib/e	64	112	0.00%
java.net.URL @ 0xe2fa08a8 file:/tmp/apache-tomcat-7.0.57/lib/c	64	112	0.00%
java.net.URL @ 0xe2fa9a00 file:/tmp/apache-tomcat-7.0.57/lib/t	64	112	0.00%
java.security.ProtectionDomain @ 0xe2e6dc88	40	104	0.00%
java.util.Vector @ 0xe2e6ef00	32	88	0.00%
Total: 25 of 34 entries; 9 more			

查看可能存在内存泄露的分析：


Leak Suspects
Leak Suspects
System Overview
Leaks
Overview
(a) 3.9 MB
(b) 1.2 MB
(c) 6.9 MB
Total: 12 MB
(a) Problem Suspect 1
(b) Problem Suspect 2
(c) Remainder
Problem Suspect 1

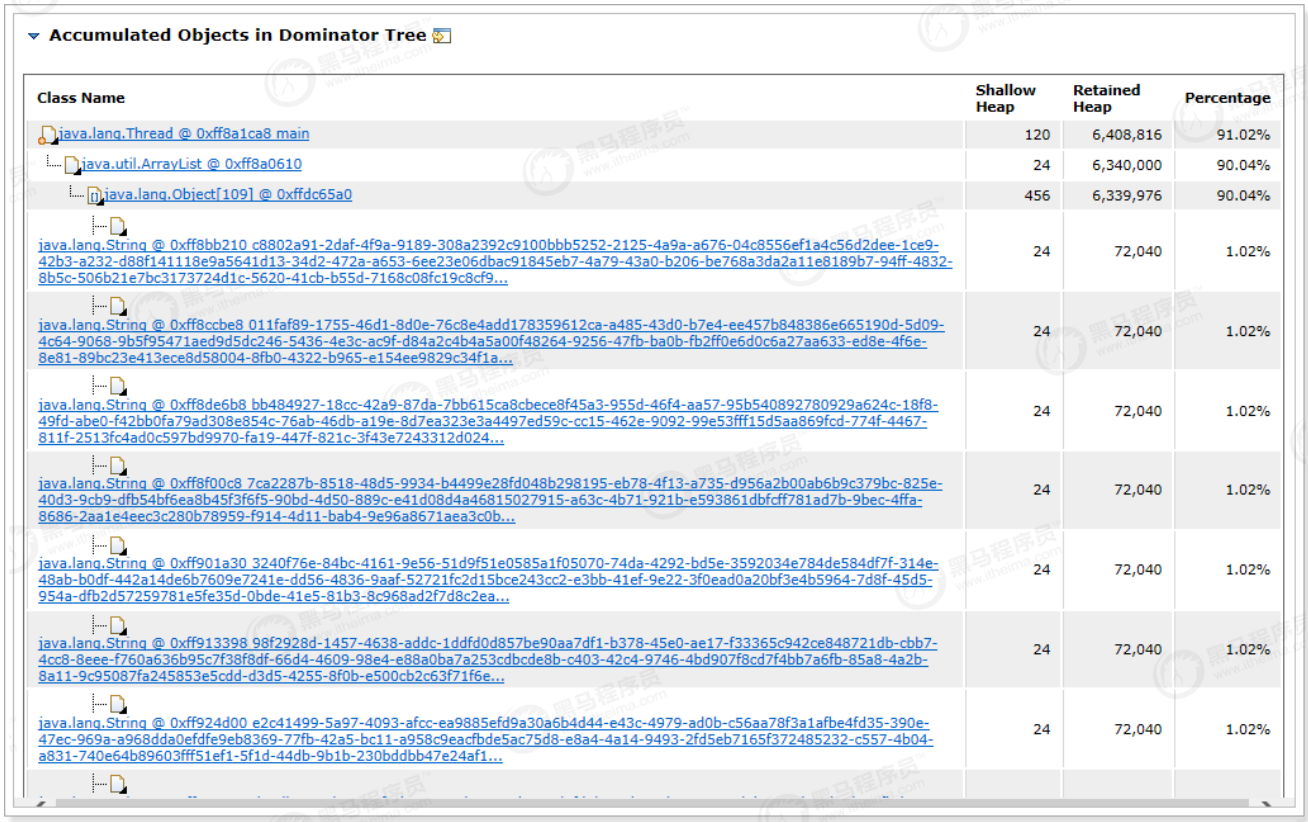
5.4.4、导入分析



可以看到，有91.02%的内存由Object[]数组占有，所以比较可疑。

分析：这个可疑是正确的，因为已经有超过90%的内存都被它占有，这是非常有可能出现内存溢出的。

查看详情：



Accumulated Objects in Dominator Tree

Class Name	Shallow Heap	Retained Heap	Percentage
java.lang.Thread @ 0xff8a1ca8 main	120	6,408,816	91.02%
java.util.ArrayList @ 0xff8a0610	24	6,340,000	90.04%
java.lang.Object[109] @ 0xffdc65a0	456	6,339,976	90.04%
java.lang.String @ 0xff8bb210 c8802a91-2daf-4f9a-9189-308a2392c9100bbb5252-2125-4a9a-a676-04c8556ef1a4c56d2dee-1ce9-42b3-a232-d88f141118e9a5641d13-34d2-472a-a653-6ee23e06dbac91845eb7-4a79-43a0-b206-be768a3da2a11e8189b7-94ff-4832-8b5c-506b21e7bc3173724d1c-5620-41cb-b55d-7168c08fc19c8cf9...	24	72,040	1.02%
java.lang.String @ 0xff8ccbe8 011faf89-1755-46d1-8d0e-76c8e4add178359612ca-a485-43d0-b7e4-ee457b848386e665190d-5d09-4c64-9068-9b5f95471aed9d5dc246-5436-4e3c-ac9f-d84a2c4b4a5a00f48264-9256-47fb-ba0b-fb2ff0e6d0c6a27aa633-ed8e-4f6e-8e81-89bc23e413ece8d58004-8fb0-4322-b965-e154ee9829c34f1a...	24	72,040	1.02%
java.lang.String @ 0xff8de6b8 bb484927-18cc-42a9-87da-7bb615ca8cbece8f45a3-955d-46f4-aa57-95b540892780929a624c-18f8-49fd-abe0-f42bb0fa79ad308e854c-76ab-46db-a19e-8d7ea323e3a4497ed59c-cc15-462e-9092-99e53fff15d5aa869fcd-774f-4467-811f-2513fc4ad0c597bd9970-fa19-447f-821c-3f43e7243312d024...	24	72,040	1.02%
java.lang.String @ 0xff8f00c8 7ca2287b-8518-48d5-9934-b4499e28fd048b298195-eb78-4f13-a735-d956a2b00ab6b9c379bc-825e-40d3-9cb9-dfb54bf6ea8b45f3f6f5-90bd-4d50-889c-e41d08d4a46815027915-a63c-4b71-921b-e593861dbfcff781ad7b-9bec-4ffa-8686-2aa1e4eec3c280b78959-f914-4d11-bab4-9e96a8671aea3c0b...	24	72,040	1.02%
java.lang.String @ 0xff901a30 3240f76e-84bc-4161-9e56-51d9f51e0585a1f05070-74da-4292-bd5e-3592034e784de584df7f-314e-48ab-b0df-442a14de6b7609e7241e-dd56-4836-9aaf-52721fc2d15bce243cc2-e3bb-41ef-9e22-3f0ead0a20bf3e4b5964-7d8f-45d5-954a-dfb2d57259781e5fe35d-0bde-41e5-81b3-8c968ad2f7d8c2ea...	24	72,040	1.02%
java.lang.String @ 0xff913398 98f2928d-1457-4638-addc-1ddfd0d857be90aa7df1-b378-45e0-ae17-f33365c942ce848721db-cbb7-4cc8-8eee-f760a636b95c7f38f8df-66d4-4609-98e4-e88a0ba7a253cdbcde8b-c403-42c4-9746-4bd907f8cd7f4bb7a6fb-85a8-4a2b-8a11-9c95087fa245853e5cdd-d3d5-4255-8f0b-e500cb2c63f71f6e...	24	72,040	1.02%
java.lang.String @ 0xff924d00 e2c41499-5a97-4093-afcc-ea9885efd9a30a6b4d44-e43c-4979-ad0b-c56aa78f3a1afbe4fd35-390e-47ec-969a-a968dda0efdfe9eb8369-77fb-42a5-bc11-a958c9eacfbde5ac75d8-e8a4-4a14-9493-2fd5eb7165f372485232-c557-4b04-a831-740e64b89603fff51ef1-5f1d-44db-9b1b-230bddbb47e24af1...	24	72,040	1.02%

可以看到集合中存储了大量的uuid字符串。

6、实战：死锁问题

有些时候我们需要查看下jvm中的线程执行情况，比如，发现服务器的CPU的负载突然增高了、出现了死锁、死循环等，我们该如何分析呢？

由于程序是正常运行的，没有任何的输出，从日志方面也看不出什么问题，所以就需要看下jvm的内部线程的执行情况，然后再进行分析查找出原因。

这个时候，就需要借助于jstack命令了，jstack的作用是将正在运行的jvm的线程 情况进行快照，并且打印出来：

```
#用法：jstack <pid>
```

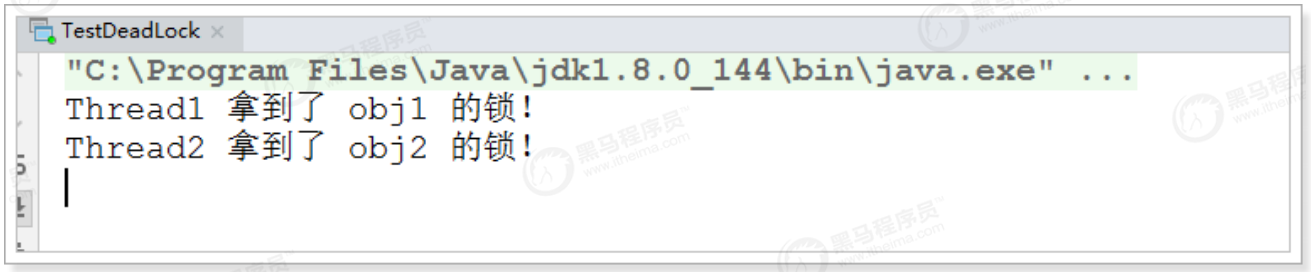
6.1、构造死锁

编写代码，启动2个线程，Thread1拿到了obj1锁，准备去拿obj2锁时，obj2已经被Thread2锁定，所以发送了死锁。

```
package cn.itcast.jvm;

public class TestDeadLock {

    private static Object obj1 = new Object();

    private static Object obj2 = new Object();


    public static void main(String[] args) {
```

```
        new Thread(new Thread1()).start();
        new Thread(new Thread2()).start();
    }

    private static class Thread1 implements Runnable{
        @Override
        public void run() {
            synchronized (obj1){
                System.out.println("Thread1 拿到了 obj1 的锁！");

                try {
                    // 停顿2秒的意义在于，让Thread2线程拿到obj2的锁
                    Thread.sleep(2000);
                } catch (InterruptedException e) {
                    e.printStackTrace();
                }

                synchronized (obj2){
                    System.out.println("Thread1 拿到了 obj2 的锁！");
                }
            }
        }
    }

    private static class Thread2 implements Runnable{
        @Override
        public void run() {
            synchronized (obj2){
                System.out.println("Thread2 拿到了 obj2 的锁！");

                try {
                    // 停顿2秒的意义在于，让Thread1线程拿到obj1的锁
                    Thread.sleep(2000);
                } catch (InterruptedException e) {
                    e.printStackTrace();
                }

                synchronized (obj1){
                    System.out.println("Thread2 拿到了 obj1 的锁！");
                }
            }
        }
    }

}
```

6.2、运行并且查询进程编号



可以看到，程序已经卡在这里了，不在继续往下执行。

```
F:\code\my-jvm>jps -l
7056 org.jetbrains.jps.cmdline.Launcher
10500
6580 jdk.jcmd/sun.tools.jps.Jps
7480
12076 itcast-jvm-app-1.0-SNAPSHOT.jar
12652 cn.itcast.jvm.TestDeadLock  #进程在这里
13964 org.jetbrains.idea.maven.server.RemoteMavenServer
```

6.3、查看线程状态

```
jstack 12652  #查看进程中的线程状态
```

在输出的信息中，已经看到，发现了1个死锁，关键看这个：

```
Found one Java-level deadlock:
=============================
"Thread-1":
  waiting to lock monitor 0x00000000026c3ee8 (object 0x0000000780198bb0, a
java.lang.Object),
  which is held by "Thread-0"
"Thread-0":
  waiting to lock monitor 0x00000000026c40f8 (object 0x0000000780198bc0, a
java.lang.Object),
  which is held by "Thread-1"

Java stack information for the threads listed above:
===================================================
"Thread-1":
        at cn.itcast.jvm.TestDeadLock$Thread2.run(TestDeadLock.java:49)
        - waiting to lock <0x0000000780198bb0> (a java.lang.Object)
        - locked <0x0000000780198bc0> (a java.lang.Object)
        at java.lang.Thread.run(Thread.java:748)
"Thread-0":
        at cn.itcast.jvm.TestDeadLock$Thread1.run(TestDeadLock.java:29)
        - waiting to lock <0x0000000780198bc0> (a java.lang.Object)
        - locked <0x0000000780198bb0> (a java.lang.Object)
        at java.lang.Thread.run(Thread.java:748)

Found 1 deadlock.
```

可以清晰的看到：

- Thread2获取了 <0x0000000780198bc0> 的锁，等待获取 <0x0000000780198bb0> 这个锁
- Thread1获取了 <0x0000000780198bb0> 的锁，等待获取 <0x0000000780198bc0> 这个锁
- 由此可见，发生了死锁。

7、VisualVM

VisualVM，能够监控线程，内存情况，查看方法的CPU时间和内存中的对 象，已被GC的对象，反向查看分配的堆栈(如100个String对象分别由哪几个对象分配出来的)。

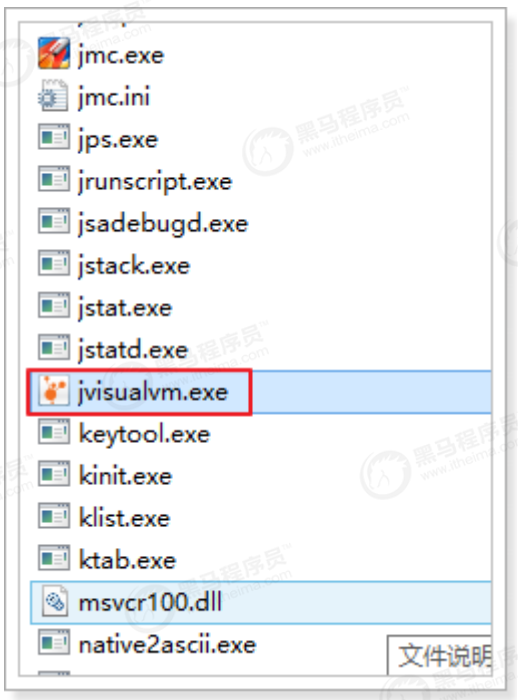
VisualVM使用简单，几乎0配置，功能还是比较丰富的，几乎囊括了其它JDK自带命令的所有功能。

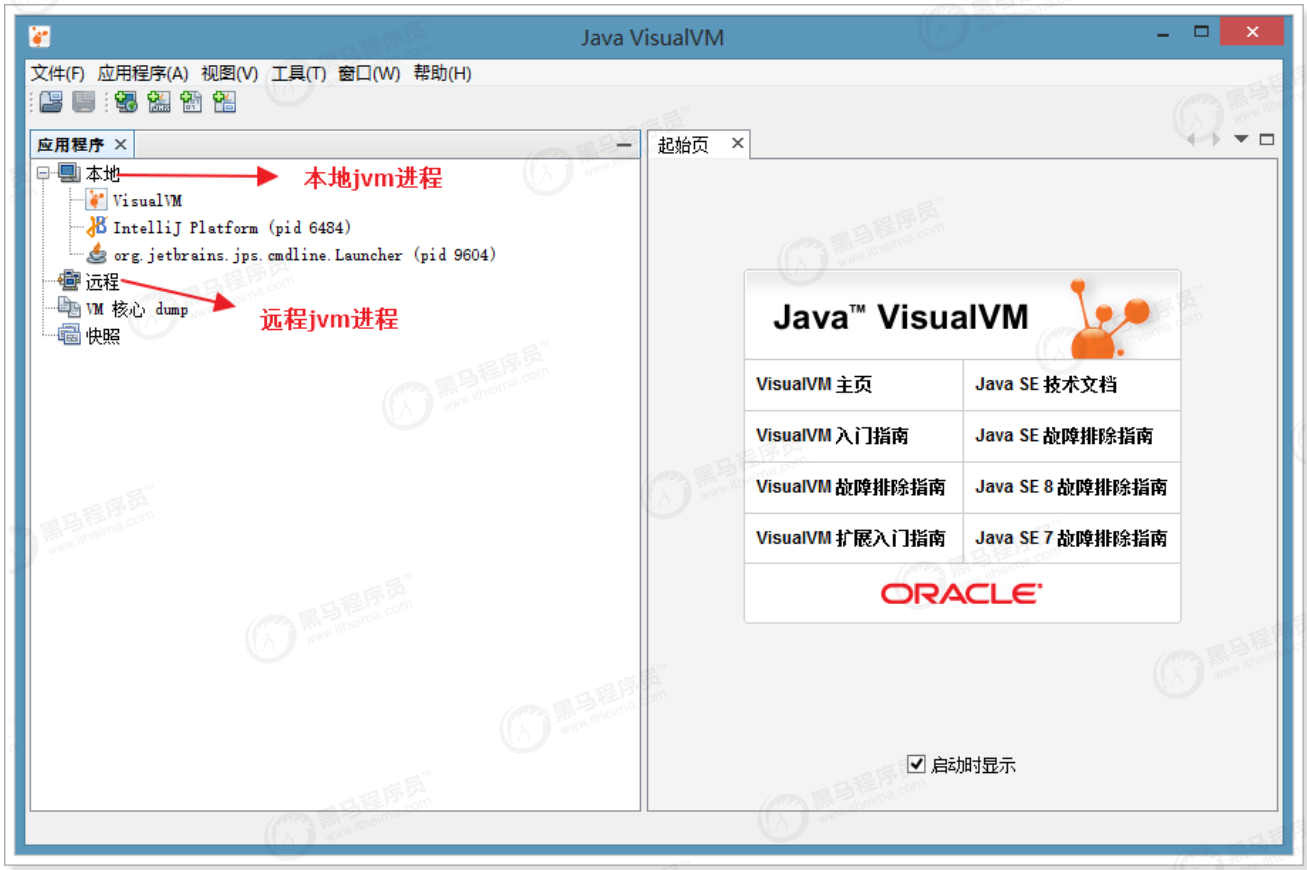
- 内存信息
- 线程信息
- Dump堆（本地进程）
- Dump线程（本地进程）
- 打开堆Dump。堆Dump可以用jmap来生成。
- 打开线程Dump
- 生成应用快照（包含内存信息、线程信息等等）
- 性能分析。CPU分析（各个方法调用时间，检查哪些方法耗时多），内存分析（各类对象占用的内存，检查哪些类占用内存多）
- ……

7.1、基本使用

7.1.1、启动

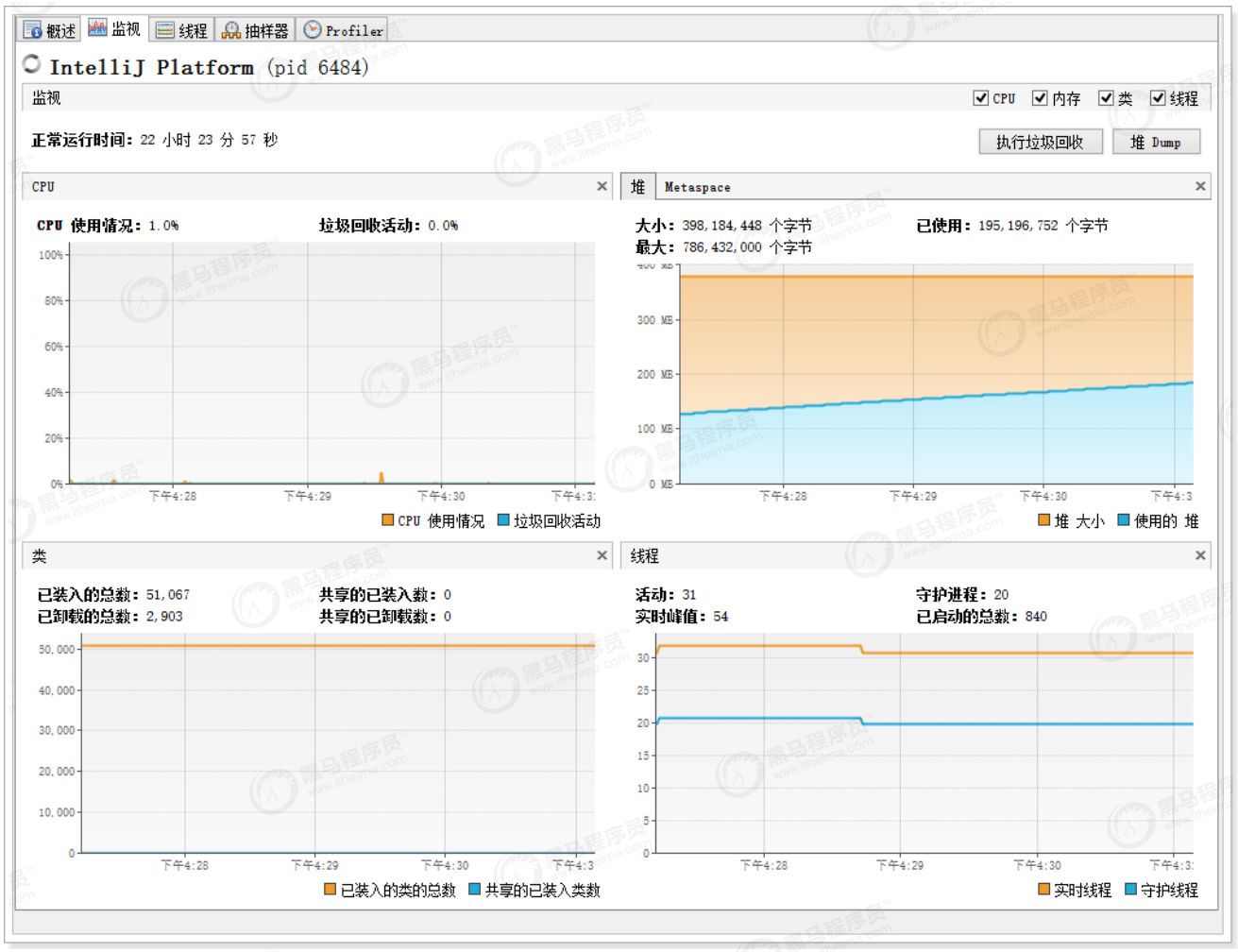
在jdk的安装目录的bin目录下，找到jvisualvm.exe，双击打开即可。



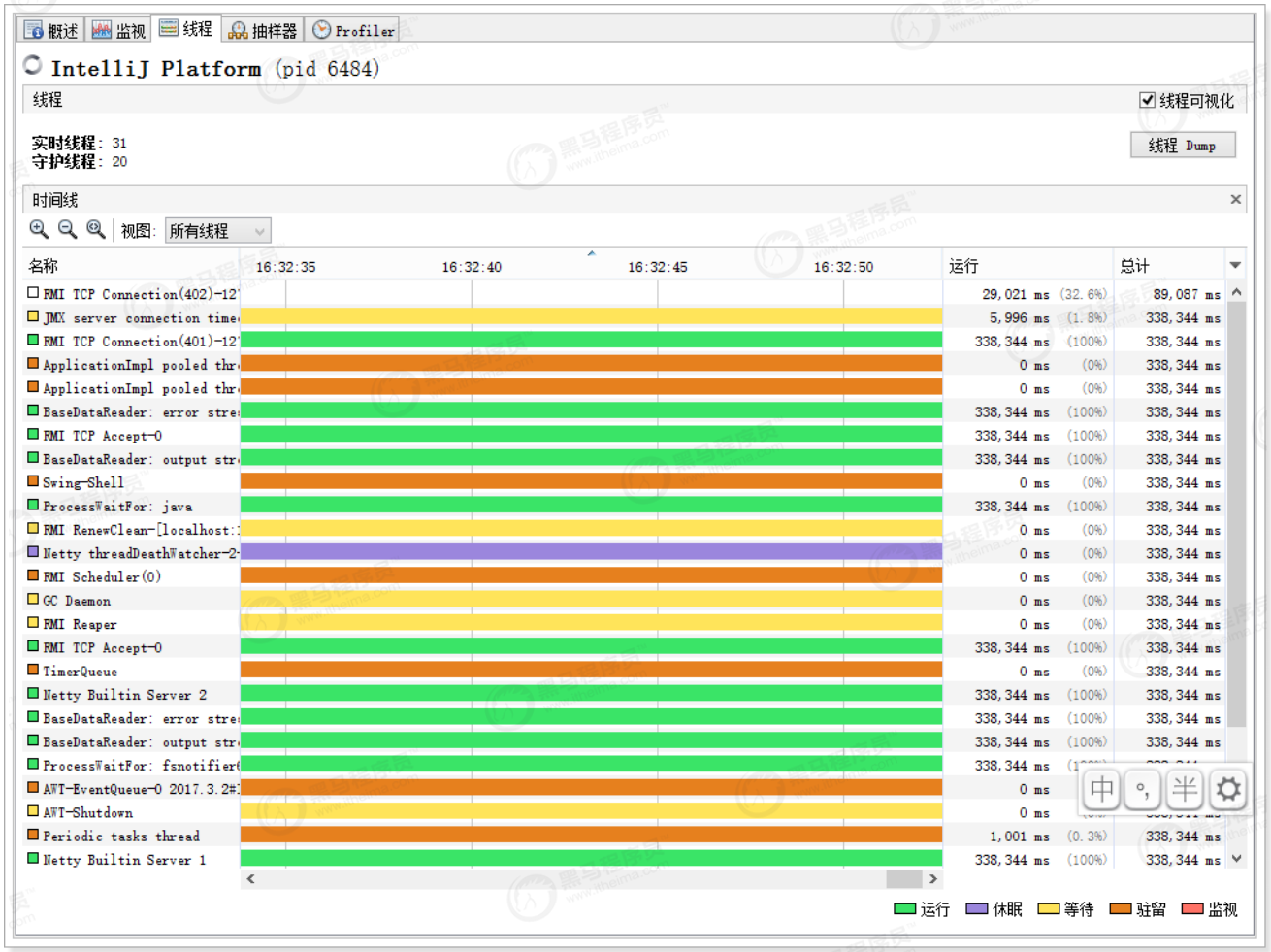


7.1.2、查看本地进程

7.1.3、查看CPU、内存、类、线程运行信息


概述
监视
线程
抽样器
Profiler
IntelliJ Platform (pid 6484)
监视
CPU
内存
类
线程
正常运行时间: 22 小时 23 分 57 秒
执行垃圾回收
堆 Dump
CPU
CPU 使用情况: 1.0%
垃圾回收活动: 0.0%
100%
80%
60%
40%
20%
0%
下午4:28
下午4:29
下午4:30
CPU 使用情况
垃圾回收活动
堆
Metaspace
大小: 398,184,448 个字节
已使用: 195,196,752 个字节
最大: 786,432,000 个字节
300 MB
200 MB
100 MB
0 MB
堆 大小
使用的 堆
类
已装入的总数: 51,067
已卸载的总数: 2,903
共享的已装入数: 0
共享的已卸载数: 0
50,000
40,000
30,000
20,000
10,000
0
已装入的类的总数
共享的已装入类数
线程
活动: 31
实时峰值: 54
守护进程: 20
已启动的总数: 840
30
25
20
15
10
5
0
实时线程
守护线程

7.1.4、查看线程详情



也可以点击右上角Dump按钮，将线程的信息导出，其实就是执行的jstack命令。

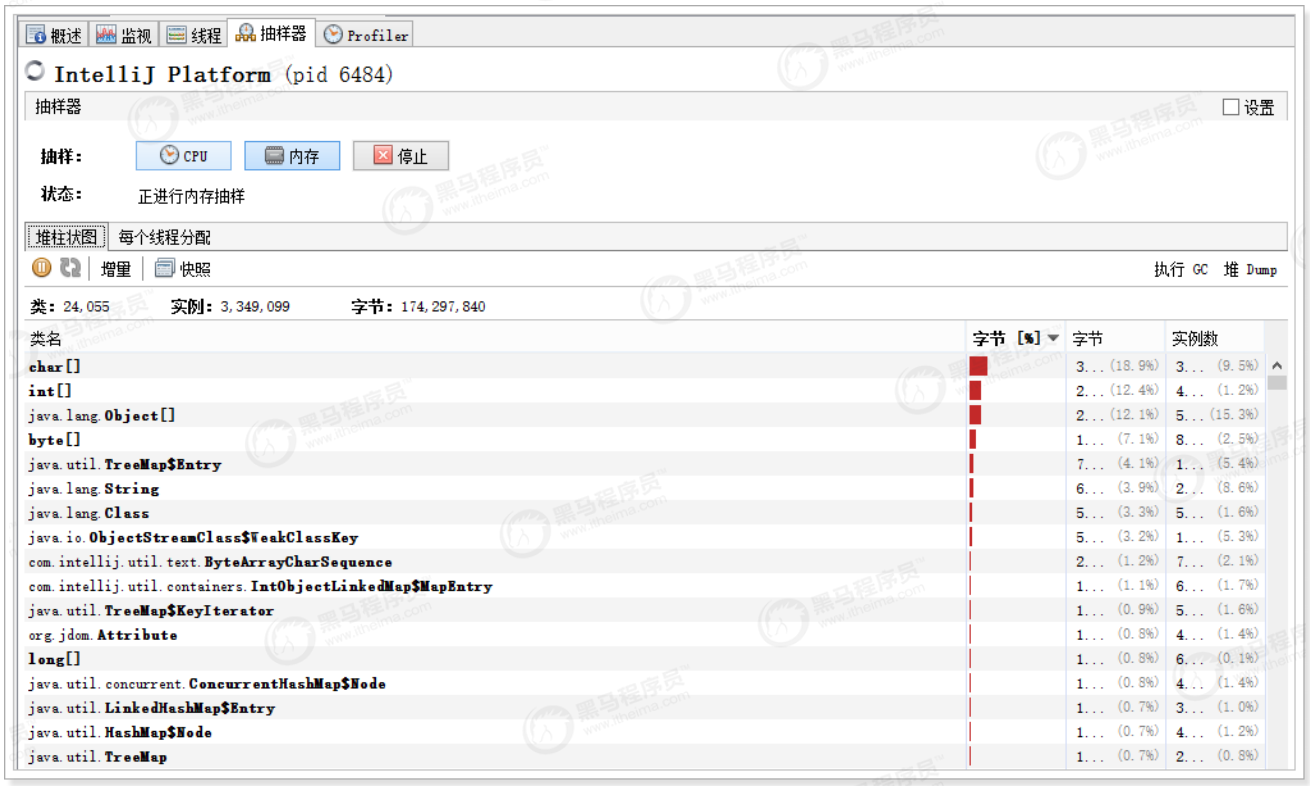
线程 Dump

```
2018-09-11 17:00:54
Full thread dump Java HotSpot(TM) 64-Bit Server VM (25.144-b01 mixed mode):

"RMI TCP Connection(404)-127.0.0.1" #856 daemon prio=5 os_prio=0 tid=0x000000003a064800 nid=0x1db0 runnable [0x00000000241fe000]
   java.lang.Thread.State: RUNNABLE
        at java.net.SocketInputStream.socketRead0(Native Method)
        at java.net.SocketInputStream.socketRead(SocketInputStream.java:116)
        at java.net.SocketInputStream.read(SocketInputStream.java:171)
        at java.net.SocketInputStream.read(SocketInputStream.java:141)
        at java.io.BufferedInputStream.fill(BufferedInputStream.java:246)
        at java.io.BufferedInputStream.read(BufferedInputStream.java:265)
        - locked <0x00000000d5d91db8> (a java.io.BufferedInputStream)
        at java.io.FilterInputStream.read(FilterInputStream.java:83)
        at sun.rmi.transport.tcp.TCPTransport.handleMessages(TCPTransport.java:550)
        at sun.rmi.transport.tcp.TCPTransport$ConnectionHandler.run0(TCPTransport.java:826)
        at sun.rmi.transport.tcp.TCPTransport$ConnectionHandler.lambda$run$0(TCPTransport.java:683)
        at sun.rmi.transport.tcp.TCPTransport$ConnectionHandler$$Lambda$1463/1745933817.run(Unknown Source)
        at java.security.AccessController.doPrivileged(Native Method)
        at sun.rmi.transport.tcp.TCPTransport$ConnectionHandler.run(TCPTransport.java:682)
        at java.util.concurrent.ThreadPoolExecutor.runWorker(ThreadPoolExecutor.java:1149)
        at java.util.concurrent.ThreadPoolExecutor$Worker.run(ThreadPoolExecutor.java:624)
        at java.lang.Thread.run(Thread.java:748)

   Locked ownable synchronizers:
        - <0x00000000d409b0f0> (a java.util.concurrent.ThreadPoolExecutor$Worker)
```

发现，显示的内容是一样的。

7.1.5、抽样器

抽样器可以对CPU、内存在一段时间内进行抽样，以供分析。



7.2、监控远程的jvm

VisualJVM不仅是可以监控本地jvm进程，还可以监控远程的jvm进程，需要借助于JMX技术实现。

7.2.1、什么是JMX？

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。

7.2.2、监控远程的tomcat

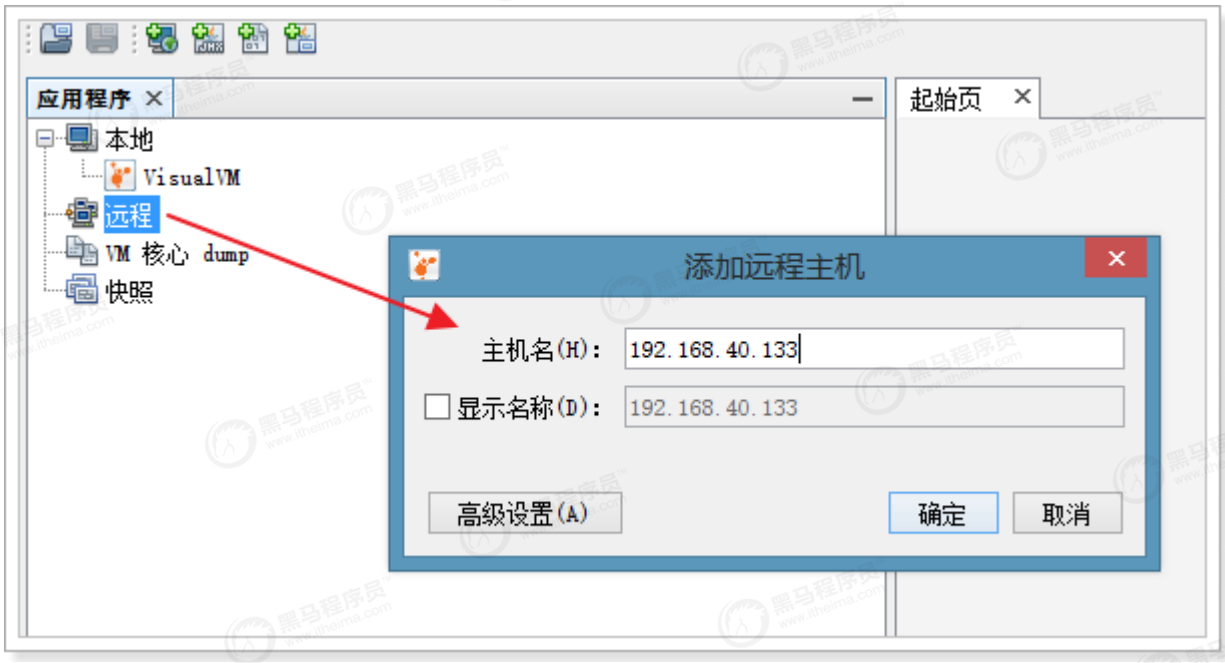
想要监控远程的tomcat，就需要在远程的tomcat进行对JMX配置，方法如下：

```
#在tomcat的bin目录下，修改catalina.sh，添加如下的参数

JAVA_OPTS="-Dcom.sun.management.jmxremote -Dcom.sun.management.jmxremote.port=9999 -
Dcom.sun.management.jmxremote.authenticate=false -
Dcom.sun.management.jmxremote.ssl=false"

#这几个参数的意思是：
#-Dcom.sun.management.jmxremote ：允许使用JMX远程管理
#-Dcom.sun.management.jmxremote.port=9999 ：JMX远程连接端口
#-Dcom.sun.management.jmxremote.authenticate=false ：不进行身份认证，任何用户都可以连接
#-Dcom.sun.management.jmxremote.ssl=false ：不使用ssl
```

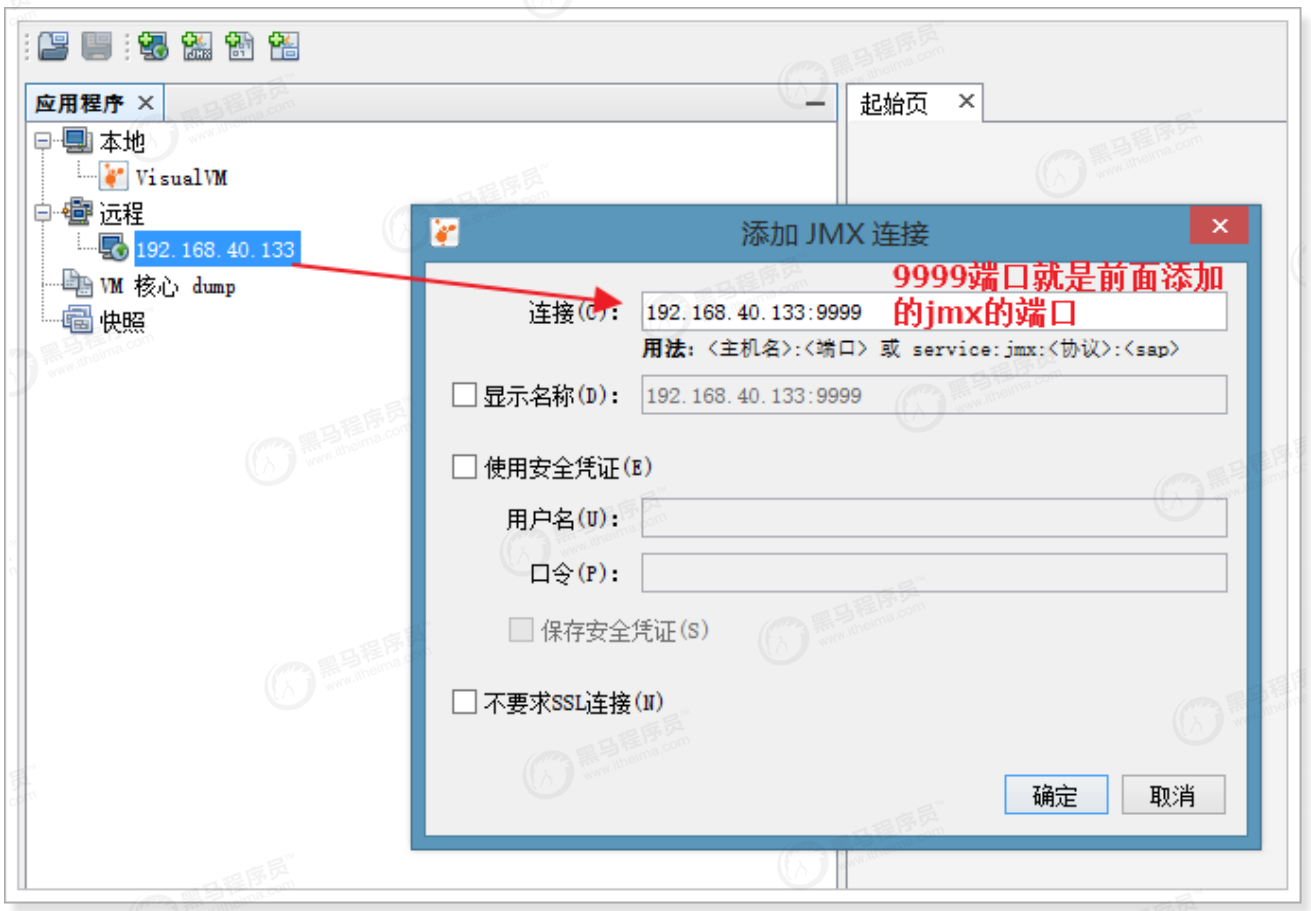
保存退出，重启tomcat。

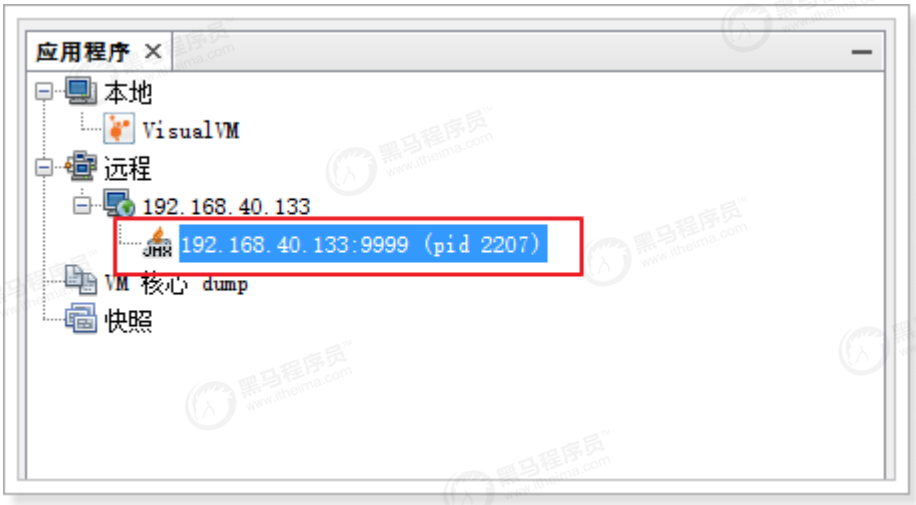
7.2.3、使用VisualJVM连接远程tomcat

添加远程主机：



在一个主机下可能会有很多的jvm需要监控，所以接下来要在该主机上添加需要监控的jvm：





连接成功。使用方法和前面就一样了，就可以和监控本地jvm进程一样，监控远程的tomcat进程。

7.3、检测死锁

7.4、检测堆内存

检测堆内存的具体使用情况，需要安装插件Visual GC进行检测：

7.4.1、编写代码

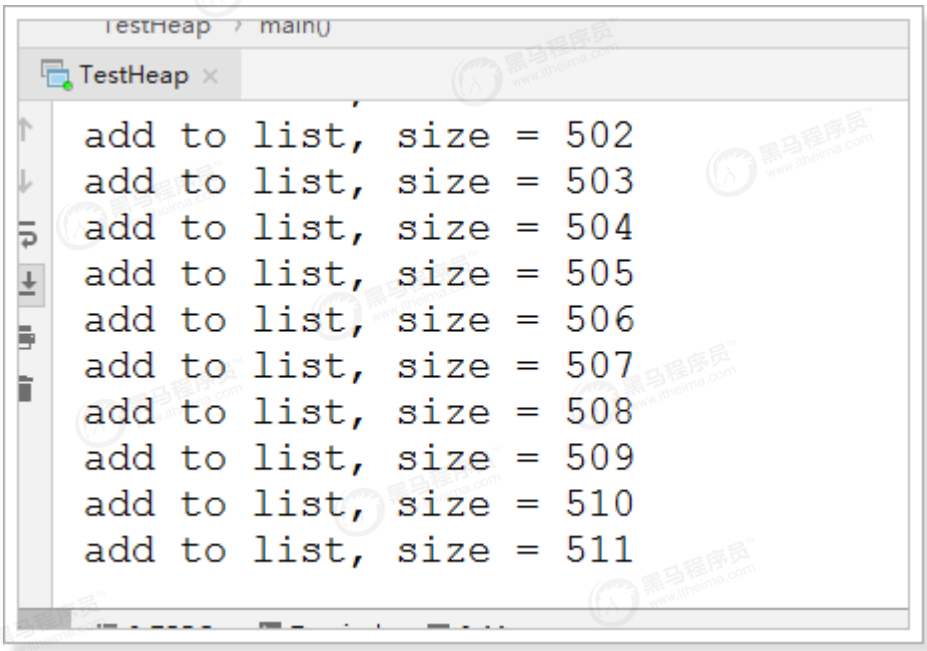
```
package cn.itcast.jvm;

import java.util.ArrayList;
import java.util.List;

public class TestHeap {

    public static void main(String[] args) {
        List<User> userList = new ArrayList<>();
        while (true){
            User user = new User();
            user.setId(1L);
            user.setUsername("user");
            user.setPassword("pass");

            if(System.currentTimeMillis() % 2 ==0 ){
                userList.add(user);//加入到集合，不符合条件的就成了垃圾对象
                System.out.println("add to list, size = " + userList.size());
            }

            try {
                Thread.sleep(1);
            } catch (InterruptedException e) {
                e.printStackTrace();
            }

        }
    }
```

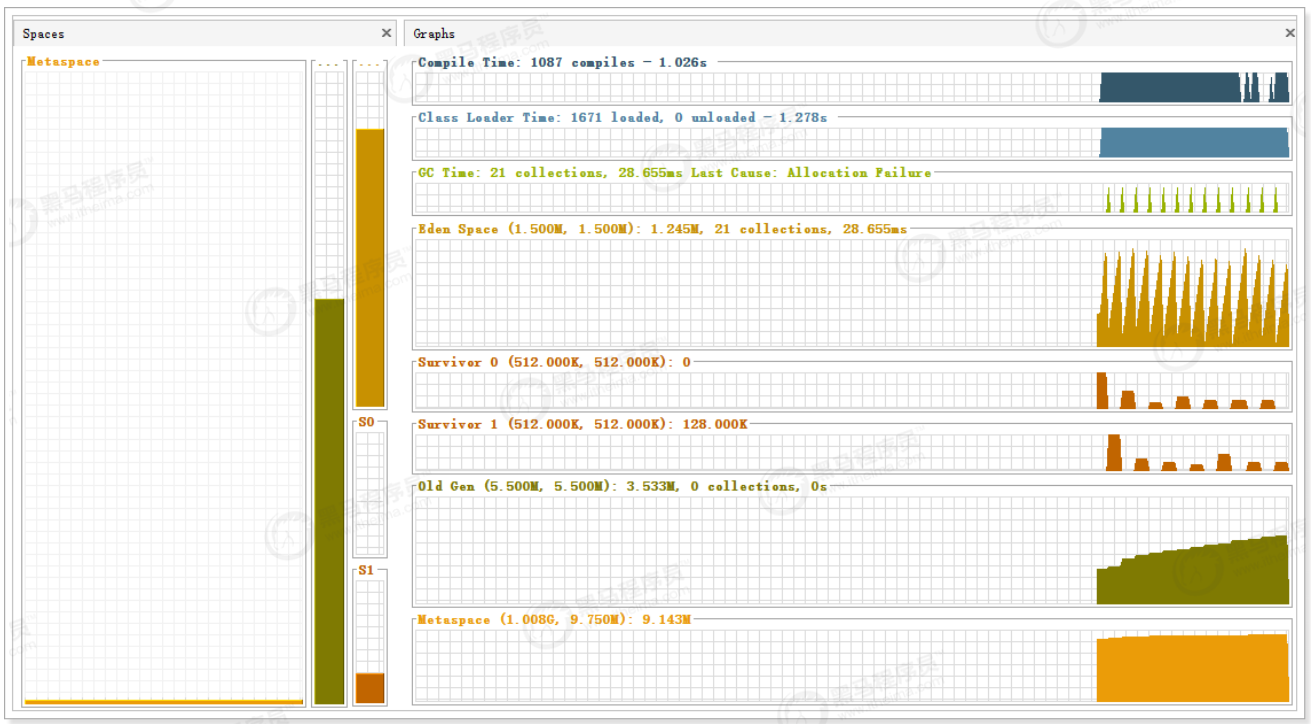
```
}
```

7.4.2、运行测试

运行参数：

```
-Xms8m -Xmx8m -XX:+HeapDumpOnOutOfMemoryError
```





可以看到，年轻代、老年代中的内存使用情况，运行一段时间后观察效果更佳明显。