课程说明

•再次认识Java

* JVM虚拟机内存管理

•虚拟机性制目关工具

-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^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**Wjava**

Java是几乎所有类型的网涪应用程宰的甚础,也是开劉口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9。。万的开发人员，使您能够高效地开发、部署和使 用靖彩的应用程序和服务匸

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

* 9了％ 的企业桌xisiT Java

Java

•美国有89%的席M （'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

•丹巨第一的部署平台

•有3。亿部移动电活运行Java

* 100%的蓝元盘播放器附带了 Java

\*有5。亿张Java卡在使用

* 1.25亿台TV设备运行Java

•前5个原始设笛制造商均提供了 Java ME

*说明：廉的内容部分费自*瀛龄wj> f

2.1、Java技术体系

Java技术体系其实已经不側提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，我们所扌旨的是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虚拟机这两咅。分统称为JRE (Java Runtime Environment) , JRE是支持 Java程序运行的标准环境。



JDK

JavaiM m

工具及 工具API 程序发布 用户界面 相关技术

集成库

javac

Security

InTI

javadoc

RM1

apt

IDL

Java Language

jar

javap

JPDA

JConsole

Deploy

Monitoring

Troubleshoot

Scripting

Java

VisualVM

JVM TI

| AWT | | Swing | | Java 2D | |
| --- | --- | --- | --- | --- | --- |
| Accessibility | Drag n Drop | Input Methods | Image I/O | Print Service | Sound |

Deployment

Java Web Start

Java Plug-In

Scripting

| JDBC | JNDI | RMI | RMI-IIOP |
| --- | --- | --- | --- |

IDL

JRE 其他

基础库

Beans

Networking

Inti Support

Override  
Mechanism

I/O

JMX

Security Serialization

JNT

Extension  
Mechanism

Math

XML JAXP

Java  
SE

API

语言和工具

lang and util

Collections

Concurrency  
Utilities

JAR

Logging

Management

基础库

Ref

Objects

Java虚拟机

操作系统

Preferences

API

Java Hotspot Client VM

Reflection

Regular

Expressions

Versioning Zip Instrument

Java Hotspot Server VM

Solaris

Linux

Windows

Other

2.2、Java发展历史

Java publicly  
aunouiiced

.1995-3-23

J 995

1995 1998-5 1999-12-12 2001-9-24 2003-11-11 2006-5-11

Java JPE J2EE 1.2 J2EE 1.3 J2EE 1.4 Java EE 5

JDK *Java*

*Alpha/ Professional*

*Beta Edition*

2009-12-10

Java EE 6

2013-6-12

Java EE 7

2017-18

Java SE 10-12

|  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| Java 1. | 1 Java 1.2 | Java 1.3 Java 1.4 | | | Java 5 | Java SE 6 | 靈**Sun** | Java SE 7 | | Java SE 8 | Java | Java |
| JDK 1 ： | 1 J2SE 1.2 | J2SE 13 | J2SE 1.4 | | J2SE 5 0 | *Mustang* | *Dolphin* | | *Spider* | SE 9 | SE 13 |
|  | *Playground* | *Kestrel* | *Merlin* | | *Tiger* |  |  |  |  |  |
| 1997-2-19 1998-12-8 | | 2000-5-8 | 2002-2-6 | | 2004-9-30 | 2006-12-11 | 2010-1-27 | 2011-7-28 | | 2014-3-18 | 2017 | 2019 |
| 157. |  | V |  |  | ■ ■ | KWH | ■ , | KT | *r* |  |  | ▼ |
| **u** | 1998  k [[1]](#footnote-2) [[2]](#footnote-3) | 20( | | )1 | :2004 | | 2007 | | 2010 | | *\* | 2013 | 2016 | 2019 |

Java 1

JDK 10 I Oak

1996-1-23

* JDK 8的第一个正式版本于2014年3月18日发布，从JDK8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。

。JEP126 :对Lambda表达式的気寺，这让Java语言拥有了流畅的函数式表达能力。

。JEP 104 :内置NashornJavaScript引擎的支持，成为Java的嵌入式JavaScript引擎。

。JEP150:新的时间、日期API。

。JEP122 :彻底移除HotSpo啲永久代。

O

* JDK9T2017年9月21日发布。JDK9发布后，Oracle随即宣布Java将会以持续交付的形式^更加敏捷的硏发节 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^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。

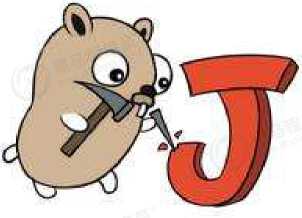
・2018年3月20日，JDK10如期发布，这版本的主要硏发目标是内部重构，诸如统一源仓库、统一垃圾收集器 接口、统一即时编译器接口等，这些都将会是对未来Java发展大有裨益的改进。

* 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JDK10中的类型推断加入Lambda语法这种可见的改进。
* 2019年2月，在JDK12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的代码通过条件编留虽行剔除掉，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，我们鼬口的虚拟机有HotSpot、 JRockit还有RM 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JDK1.0 ,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，但是彳敵口夕匯了即 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、两级即日谛译器、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"襲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啲小公司设计的。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JDKI页目的共同虚拟机。

Oracle收购Sun以后，建立了HotRockit项目来把原来BEAJRockit中的优秀特性融合到HotSpot之中。到了2014年 的JDK8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、 BEAJRockitJVM

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Rockit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。

除此之夕卜,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 也奸领獄平。

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、 IBMJ9VM

旧M J9虚拟机并不是旧M公司唯一的Java虚拟机,不过目前旧M主力发展无疑就是］9。

与BEAJRockit只专注于服务端应用不同，旧MJ9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，在和旧M产品(如旧M WebSphere等)搭配以及在旧M AIX和z/OS这些平台上部署Java应用。

从2016年起,旧M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项目的最大参与者旧M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Harmon弗殳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）被吸纳进旧M的 JDK7实现以及Google Android SD遂中，尤其是对Android的发展起了很大推动作用。

2.3.6、 Microsoft JVM

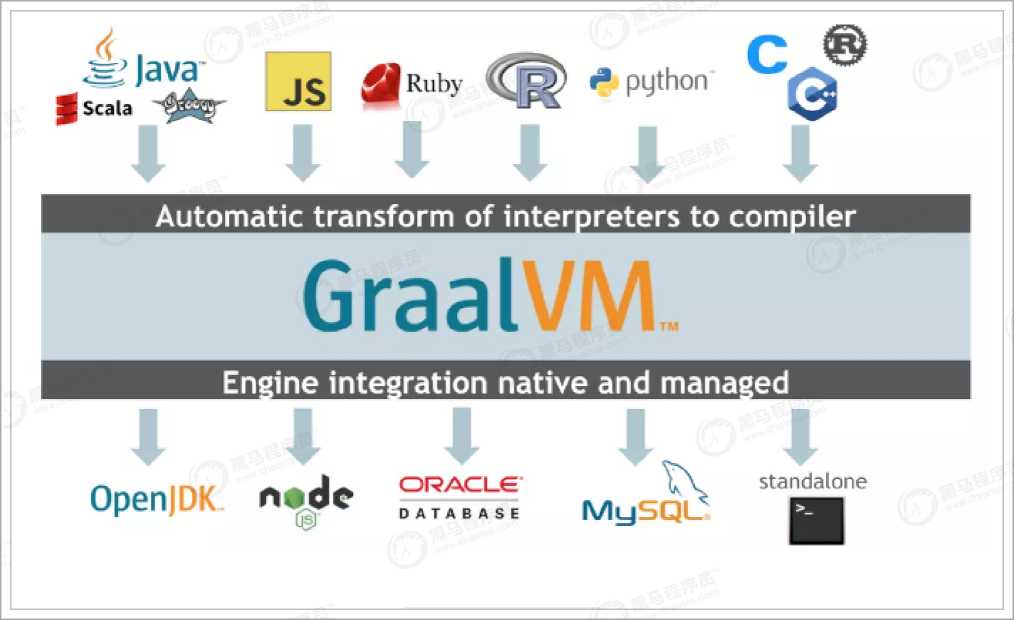
在Java语言诞生的初期，它的主要应用之一是在浏览器中运行Java Applets程序，微软为了在Internet Explorer 3浏 览器中支持Java Applets应用而开发了自己的Java虚拟机，虽然这款虚拟机只有Windows平台的版本，"一次编译， 到处运行”根本无从谈起,但却是当时Windows系统下性能最好的Java虚拟机。

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 虚拟机也曾作为曜之一被呈送法庭。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 觉到一颗蓬勃的野房

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"田可语言啲运行平台使用,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。

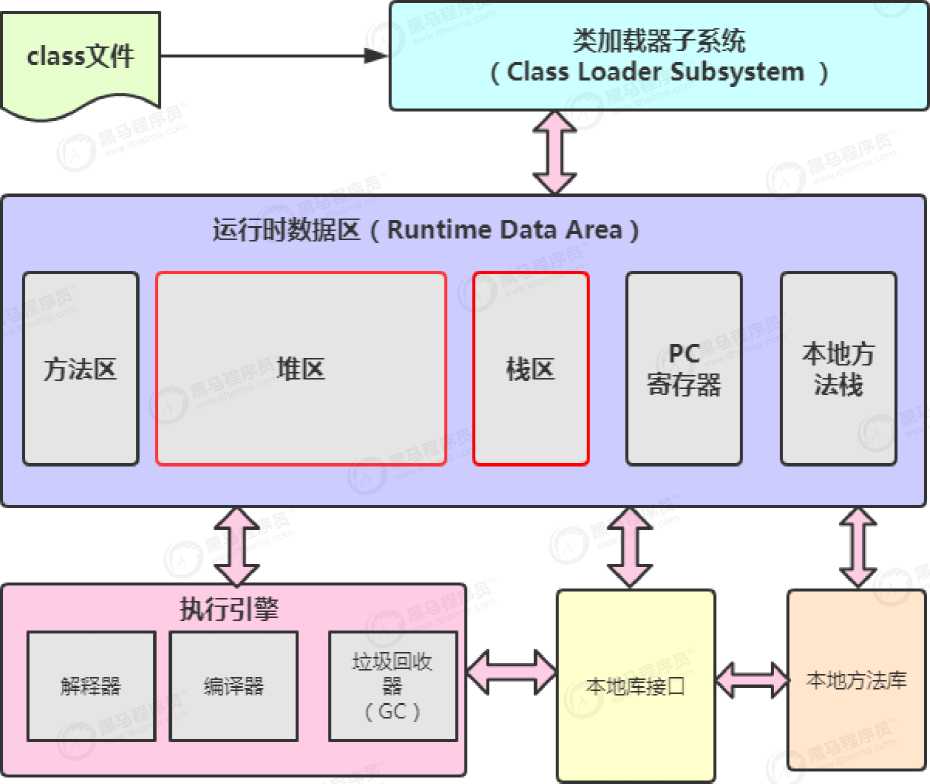


**3**、**JVM**虚拟机内存管理

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，不容易出现内存泄滴口内存溢出问题，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、转换解悟口初始化,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虚拟机前丸行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,以及创衝口销毁的时间，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,分别是解执行和编译执行,区别在于,鱒执行是国丸行 时翻译成虚拟机指令执行，而编译执行是国丸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。

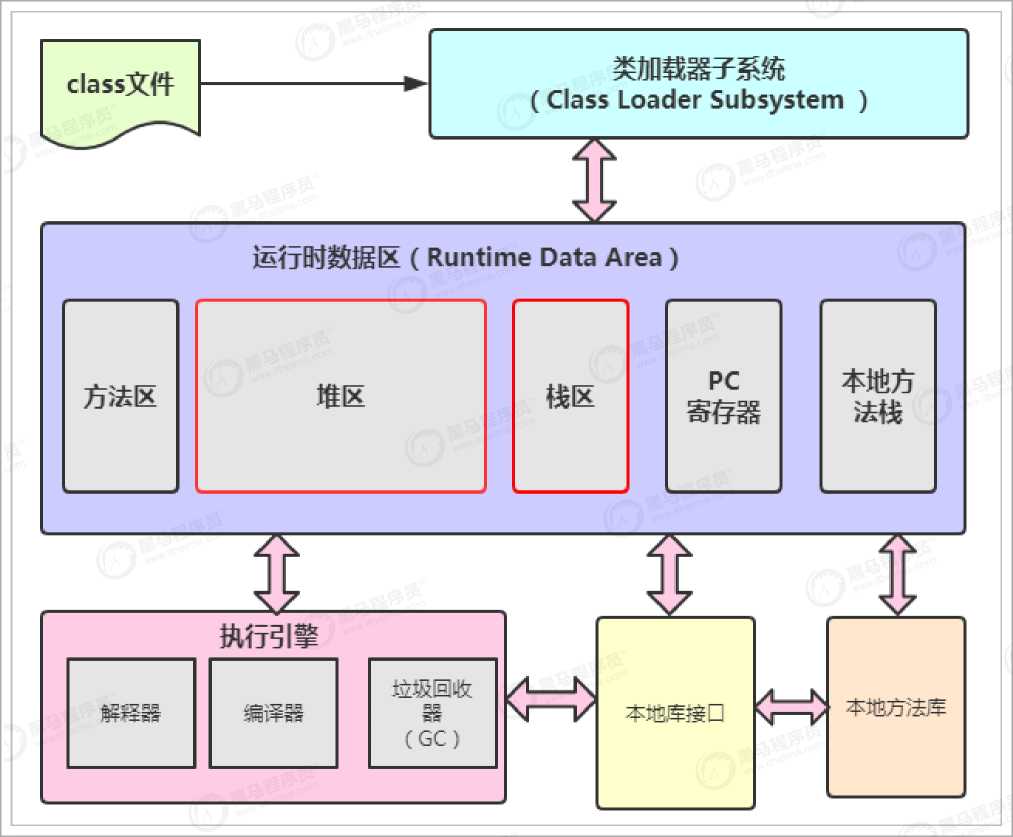
垃圾回收器就是自动管理运行数据区的内存，将无用的内存占用进行清除，释放内存资源。

**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。



根据幻ava虚拟脚范》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，它可以看作是当前绷呈所执行的字节码的行号 指示器。

字节码解器工作时就是鮑改变这个计数器的值来选取下f需劉丸行的字节码指令,它是程序控制流的指示 器，分支、循环、物辟专、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，在田可一个确定的时刻，- 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。

1. 编写代码

package cn.itcast.jvm;

public class Testi (

public int add(){

i nt a = 1;

int b = 2;

int c = a + b;

return c;

}

public static vold main(string[] args) {

Testi testl = new Testl();

int result = testl.add(); System.out.pri ntln(result);

}

}

1. 查询class的汇编代码

javap -c Testl.class > T.txt

compi1ed from "Testl.java"

public class cn.itcast.jvm.Testl (

public cn.itcast.jvm.Testl();

Code:

0: aload\_0

1: i nvokespeci al #1 // Method java/lang/object."<i ni t>":()V

4: return

public int add();

Code:

0: 1const\_l

1: istore\_l

2: iconst\_2

3: istore\_2

4: iload\_l

5: i1oad\_2

6: iadd

7: istore\_3

8: iload\_3

9: i return

public static vold main(java.lang.string[]);

Code:

客醐丽嘲YW由WTO 哄皿氾一販

W，為風間由当身帷罩由躁就黑氾一母。骨写禺口用黑草、翁車翠昭 '驱辎膳f '峯喜至龌昌熟身工由，讷

科一魂聞回朝啪團以叫’細噸帽蟻时母:春翰旦泌翎少絲草& 曾間神驱啪f旳e「

。回帥爵与隨圍粤丰間己'間身雖為歸者甲驱叩福爭以叫，却—器慌K写為与

常剛雪\*「**7**•竺

。旨球爾暗 曰坦由器凛K整聒^也艇旨饗5睥仪坦沖8由茸'成而胆丁便韧'饗二四丄胴国驻丰由饷gsep母住朝也

A(I) : ULiULJd -uipajisiu LJd/OL/FAPf poqq.aiAl //

PL^M //

工()：ppe poqq.aiAi //

A()：u<1LUL>h poq冋i //

T3-S9±/UIaC/1SPD1L/UD SSP|\_D //

ujniaj ：oz

9# Lenu la9>|oau l : zj 厂peo” ：9T

Juipajq-Siu LJd/oL/PAPCn : q.no ■ujaq.sXs/6ue|\_/PAPC

S# d L4.Pisq.a6 : £i

3\_9JO1SL ：2T

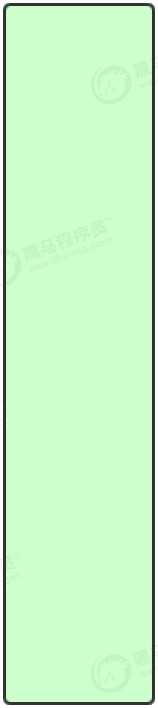
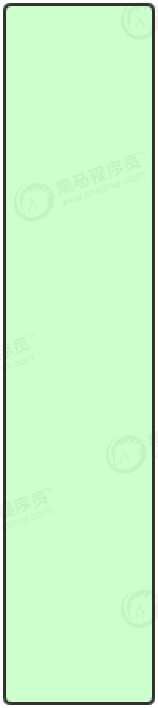
\*# Lenu la9>|oau l ：6

T~ppolp ：8 X\_ajoq.sp :/

£# LELD9dsa>|OAU L :\*

dnp : £

Z# Mau ：0



当前线程

**Java**虚拟机栈

**A**

线程**2**

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」ocals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，索弓|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， 贝U说明会同时使用第N和N+1两个变量槽。

.局部变量表中第。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I赡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SJCode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，也就是出櫛口入栈扌菊作。

•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弓I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頡引用，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，这部分瞒为动态连接。

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彳輔方法返回扌旨令来决定,这种退出方法的方质为"正 常调用完成，

.另夕种退出方式是在方法执行的过程中遇了异常，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th row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田可返回值的。

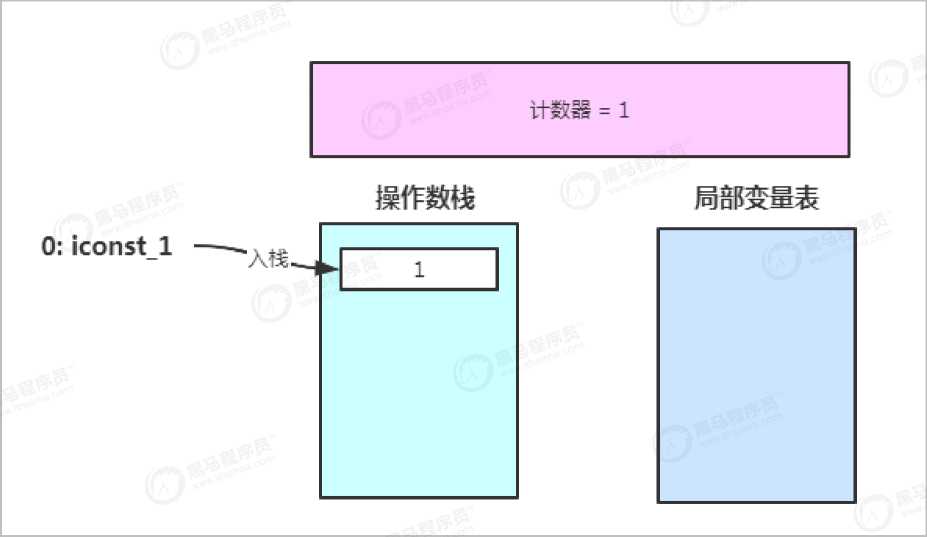
・无论采用彳輔退出方式，在方法退出之后，都必须返回到最初方法被调用时的位置，程序才能继续执行，方 法返回时可能需要在栈帧中保存一些信息，用来帮助恢复它的上层主调方法的执行状态。

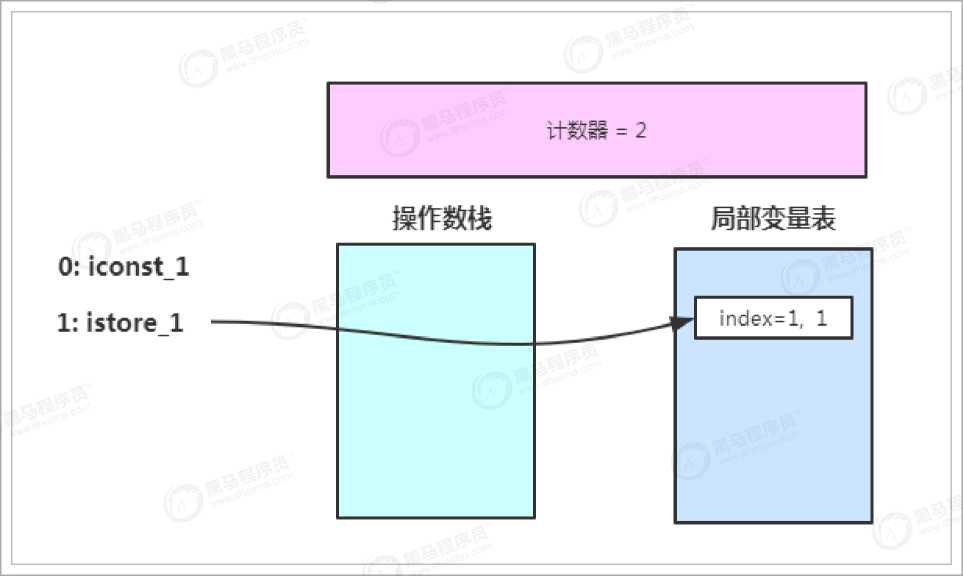
•方法退出的过程实际上等同于把当前栈帧出栈，因此退出时可能执行的操作有：恢复上层方法的局部变量表 和操作数栈，把返回值(如果有的话)压入调用者栈帧的操作数栈中，调整PC计数器的值以J旨向方法调用指 令后面的f指令等。

3.2.2.5、 实例

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

inta = 1;入栈再到局部鮭表的操作：





注意:计数指向的是下一个执行行号，以稠部变量表下标从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槐丸行 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堆是被所有爨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,幻ava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。

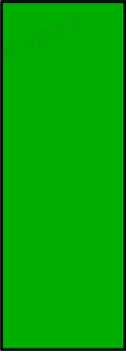
Java堆是垃圾收集器管理的内存区域，因此它也被称作"GC堆”,这就是我们做VM调优的重点区域部分。

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，所限下里,我们将先学习1.7再学习1.8的内存模型。

Tenured

Eden



JO>A.ms

v ►

Young

Perm

* 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SurvivorK ,其中，Survivor区间中，某一时刻只有 其中一个是被使用的，另夕卜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SurvivorK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You ng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§啲缓存，缓存中的对象往往会被转 移到这一区间。

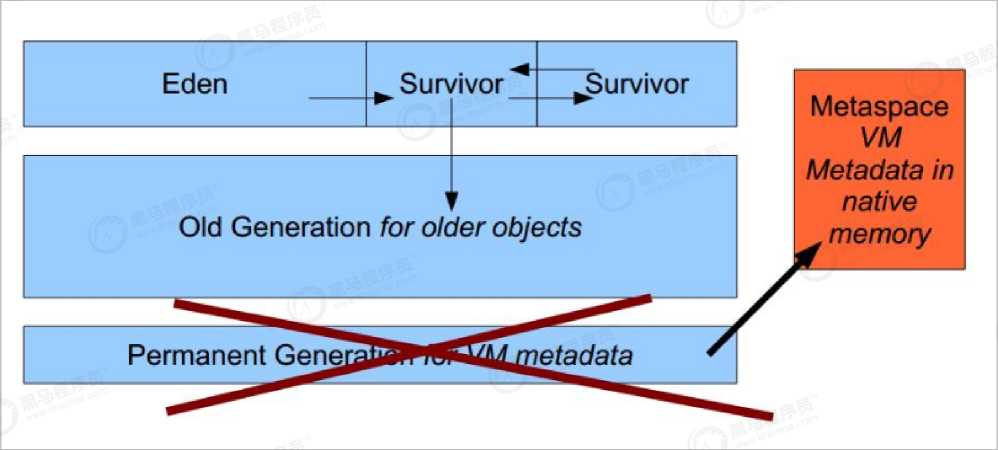
* Perm永久区

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 涉及热部署的应用服务器的时候，有时候会遇SJjava.Iang.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：

o最大内存^初始内存的差值，就是Virtual区。

3.2.4.2、jdk1.8的堆内存模型



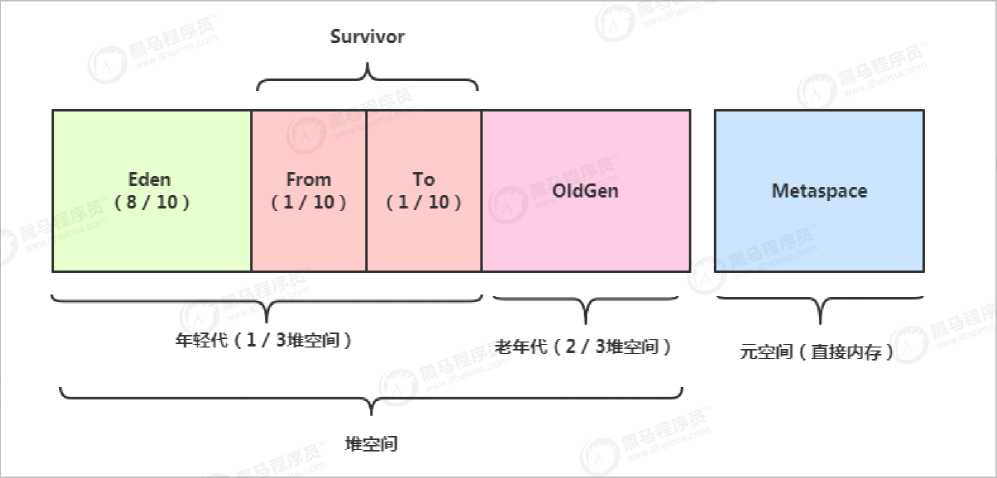
由上图可以看出，jdk1.8的内存模型是由2部分组成，年轻代+年老代。

年芻弋：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1 G。（JDK8的情况下）

官方说明:<https://docs.oracle.eom/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，经常忽畸卓直接内存，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:httD：〃ODenidk.iava.net/ieDs/122

This is part of the JRockit and Hotspot convergence effort. JRockit customers do not need to configure the permanent generation (sinee 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。

・幻ava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:

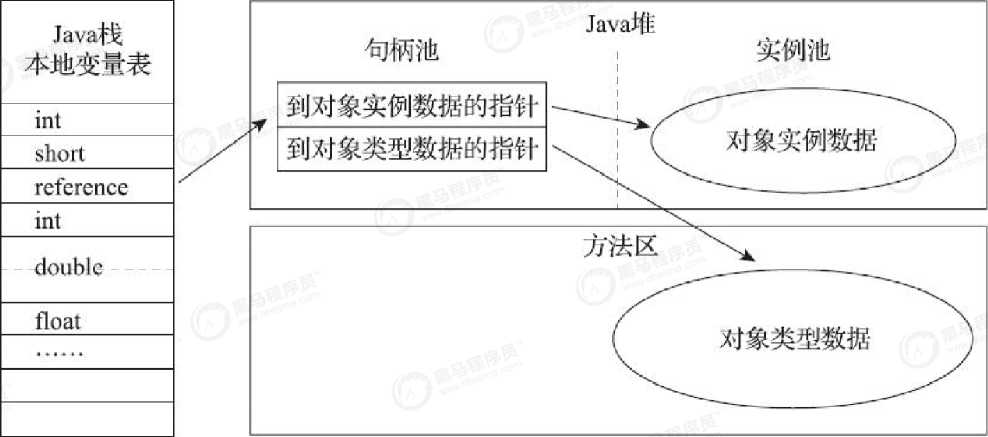
°使用句柄访问

■ 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 中包含了对象实例瓣与类型数据各自具体的地址信息

O使用直密旨针访问

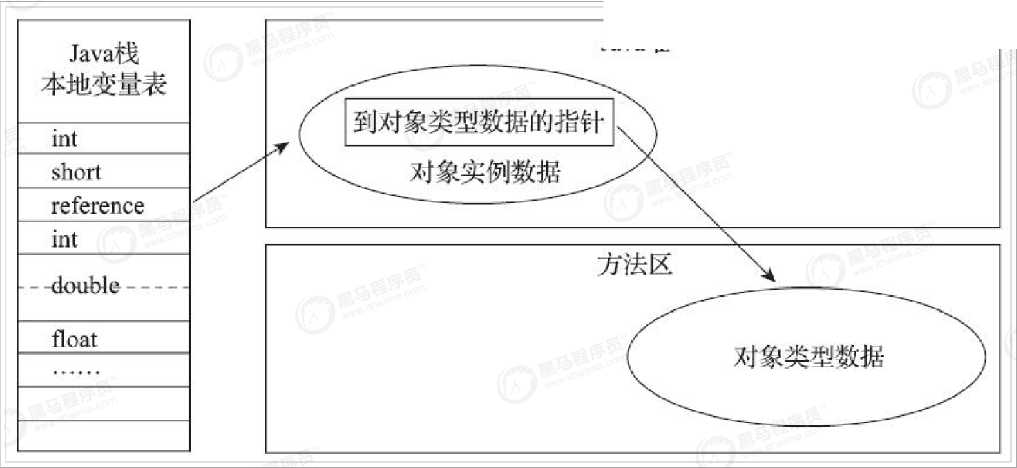
-Java堆中对象的内存布局就必须考虑如何放置访问类型数据的相关信息，reference中存储的頡就 是对象地址,如果只是访问对象本身的话,就不需要多一次间接访问的开销

句柄访问:



指甜方问:

**Java 堆**



•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J旨针定位的时间开销。

* HotSpot虚拟机采用的剧旨针访问方式实现。

**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

o -help

。-version

* -X参数（非信隹参数）

。-Xi nt

o -Xcomp

* -XX参数（使用率较高）

o -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 好看jvm版本 java -version

java version "1.8.0\_141"

Java(TM) SE Runtime Envi ronment (build 1.8.0\_141-bl5)

Java HotSpot(TM) 64-Bit Server VM (build 25.141-bl5, mixed mode)

实例：ifia-D设麋统属性参数

public class TestJVM (

public static vold main(string[] args) { String str = System.getPropertyC'str"); if (str == null) (

System.out.pri ntln("itcast");

} else {

System.out.pri ntln(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-se rve r或-c I i e n t设置j vm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i±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 Envi ronment (build 1.8.0\_141-bl5)

Java Hotspot(TM) 64-Bit Server VM (build 25.141-bl5, mixed mode)

itcast

[root@node01 test]# java -server -showversion TestJVM java version "1.8.0\_141"

Java(TM) SE Runtime Envi ronment (build 1.8.0\_141-bl5)

Java Hotspot(TM) 64-Bit Server VM (build 25.141-bl5, mixed mode) itcast

#由于机器是64位系统，所以不支持c】1 ent模式

4.2、-X参数

jvm的-X参数是日瀚准参数,在不同版本的jvm中,参数可能会有所不同,可以fflidjava -X查看非标准参数。

|  |  |
| --- | --- |
| [root@node01 test]# java -X | |
| -Xmixed | 混合齢执行（默认） |
| -Xi nt | 跳释模式执行 |
| -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 | 禁用类垃圾蛭 |
| -Xi ncgc | 启用增量垃圾幡 |
| -xloggc:<fi le> | 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> | 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 | 不尝试使用共孚类数据 |
| -Xshare:auto | 在可能的情况下使用共孚类数据（默认） |
| -Xshare:on | 要求睇共享类螭，否则将燃 |
| -xshowsetti ngs | 显示所有设置并继续 |
| -xshowsetti ngs: | all  显示所有设置并继续 |
| -xshowsetti ngs: | vm显示所有与vm相关的设置并继续 |
| -xshowsetti ngs: | properti es  显示所有属性设置并继续 |
| -Xshowsetti ngs: | 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J IT编译器的全部功能。J IT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R丸行一次的代码裁有意义了。

・-Xmixed是混合模式，将解模式与编译模式进行混合使用，由jvm自己决定，这是jvm瞅的模式，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 Envi ronment (build 1.8.0\_141-bl5)

Java Hotspot(TM) 64-Bit Server VM (build 25.141-bl5, 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 Envi ronment (build 1.8.0\_141-bl5)

Java Hotspot(TM) 64-Bit Server VM (build 25.141-bl5, compi1ed 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S,注意观察。

#瞅的混合模式

[root@node01 test]# java -showversion TestJVM

java version "1.8.0\_141"

Java(TM) SE Runtime Envi ronment (build 1.8.0\_141-bl5)

Java Hotspot(TM) 64-Bi t Server VM (build 25.141-bl5, mixed mode)

itcast

4.3、-XX参数

-XX参数也是日瀚准参数,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:

* booleang^gy

°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

o 格式:-XX:<name>=<value> 表示<name>属性的值为<value>

。如:-XX:NewRatio=4表示新生代^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 Envi ronment (buiId 1.8.0\_141-bl5)

Java HotSpot(TM) 64-Bi t Server VM (build 25.141-bl5, 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:lnitialHeapSize ,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

Attachi ng to process ID 12076, pl ease wait...

Debugger attached successfully.

Server compi1 er detected.

JVM version is 25.144-bOl

Non-default VM flags: -XX：cicompi1ercount=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 ti on -XX：+useParal1elGC

command line:

#查看某-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 -compi1 er 12076

compi1ed Fai1ed inval1d Time Fai1edType Fai1edMethod

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

SOc sic SOU

S1U

EC

EU

0C

CCSC CCSU YGC

YGCT

FGC

FGCT

GCT

3584.0 6656.0 3412.1

0.0 180224.0

89915.4

61440.0

7.3 3840.0 3420.8

0.036 1

0.026

0.062

#也可以指定打印的间隔和次数，每丄秒中打印一次,共打印5次 F：\t>jstat -gc 12076

sOc sic SOU

1000 5

S1U

EC

EU

oc

CCSC CCSU YGC

YGCT

FGC

FGCT

GCT

3584.0 6656.0 3412.1

0.0

180224.0

89915.4

61440.0

7.3 3840.0 3420.8

0.036 1

0.026

0.062

3584.0 6656.0 3412.1

0.0

180224.0 89915.4 61440.0

7.3 3840.0 3420.8

0.036 1

0.026

0.062

3584.0 6656.0 3412.1

0.0

180224.0 89915.4 61440.0

7.3 3840.0 3420.8

0.036 1

0.026

0.062

3584.0 6656.0 3412.1

0.0

180224.0 89915.4 61440.0

7.3 3840.0 3420.8

0.036 1

0.026

0.062

3584.0 6656.0 3412.1

0.0

180224.0 89915.4 61440.0

| OU | MC | MU |
| --- | --- | --- |
| 5332.1 | 27904.0 | 2626 |
| OU | MC | MU |
| 5332.1 | 27904.0 | 2626 |
| 5332.1 | 27904.0 | 2626 |
| 5332.1 | 27904.0 | 2626 |
| 5332.1 | 27904.0 | 2626 |
| 5332.1 | 27904.0 | 2626 |

7.3 3840.0 3420.8

0.036 1

0.026

0.062

说明：

* SOC :第一个Survivor区的大小（KB ）
* S1C :第二个Survivor区的大小（KB ）

, SOU :第一个SurvivorK的使用大小（KB ）

* S1U :第二个SurvivorK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.1tcast.jvm;

i mport java.uti1.ArrayLi st;

i mport java.uti1.Li st;

import java.uti1.UUID；

public class TestJvmoutofMemory (

public static vold 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().tostri ng();

}

list.add(str);

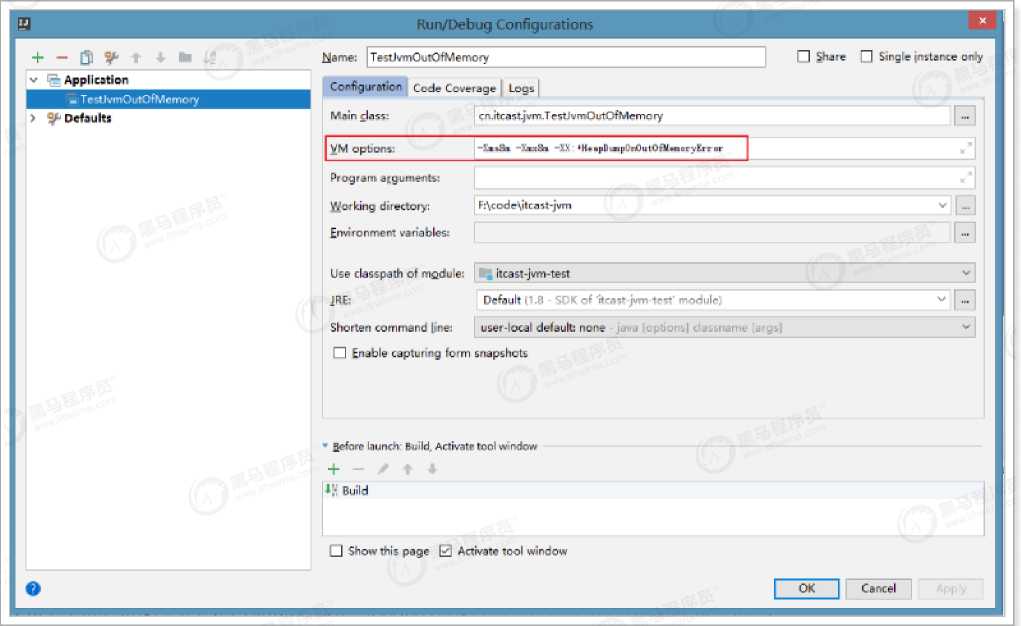
}

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

#指定路径

-Xms8m -Xmx8m -XX：+HeapDumponoutofMemoryError -XX：HeapDumpPath=F：\t\dump.hprof

#手动dump内存

#用法：

jmap -dump:format=b,fi1e=dumpFi1 eName <pid>

#示例

jmap -dump:format=b,fi1e=/tmp/dump.dat 6219

5.3、运行测试

Dumping heap to java\_pidl3132.hprof

Exception in thread "main" java.lang.outofMemoryError: Java heap space

at java.uti1.Arrays.copyof(Arrays.java:3332)

at

java.lang.Abstracts!ri ngBuiIder.ensurecapacitylnternal(AbstractStri ngBuiIder.java:124) at java.lang.AbstractStringBuiIder.append(AbstractStringBuiIder.java:448) at java.lang.StringBuiIder.append(stringBuiIder.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.hpr。仲，该文件在项目的根目录下。

i java pid13132.hprof

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),—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:httDS:〃wwvv.eclii3se.org/mat/

Memory Analyzer (MAT)

The Eclipse Memory Analyzer is a fast and feature-rich Java heap analyzer that helps you find memory leaks and reduce memory consumption.

Use the Memory Analyzer to analyze productive heap dumps with hundreds of millions of objects, quickly calculate the retained sizes of objects, see who is preventing the Garbage Co I lector from collecting objects, run a report to automatically extract leak suspects.

5.4.2、下载安装

下址：httDS：〃wvwv.ecliDse.org/mat/downloads.DhDThe **stand-alone** Memory Analyzer is based on Eclipse RCR It is useful if you do not **want** to install a full-fledged IDE on the system you are running the heap analysis.

To install the Memory Analyzer **into an Eclipse IDE** use the update site **URL** provided below. The *Memory Analyzer (Chart)* feature is optional. The chart feature requires the BIRT Chart Engine (Version 2.3.0 or greater).

The rninimum Java version required to run Memory Analyze『is 1.8

Memory Analyzer 1.8.0 Release

**• Version:** 1.8.0.20180604 | **Date:** 27 June 2018 | **Type:** Released o **Update Site:** htt p://down Ioad.ecI i pse.org/niat/1.8/u pdate-site/ o **Archived Update Site: M emo** ry A n a lyze r-L7.0.201706130745.zi **p** o **Stand-alone Eclipse RCP Applications**

JQ] Windows (x8 6)

鳳 Windows (x86\_64)

[g] MacOSX(Mac/Cocoa/x86\_64)

鳳 Linux(X86/GTK+) I冒I Linux (x86\_64/GTK+) (SI Linux (PPC64/GTKf) 直 Linux (PPC64te/GTK+)

Other Releases

* Previous Releases
* Snapshot Builds

将下载得到的MemoryAnalyzer-1.8.0.20180604-win32.win32.x86\_64.zip进行解压:

configuration

.features.

..P2

i・ plugins

.workspace

l.'I eclipsec.exe

G epl-v10.html

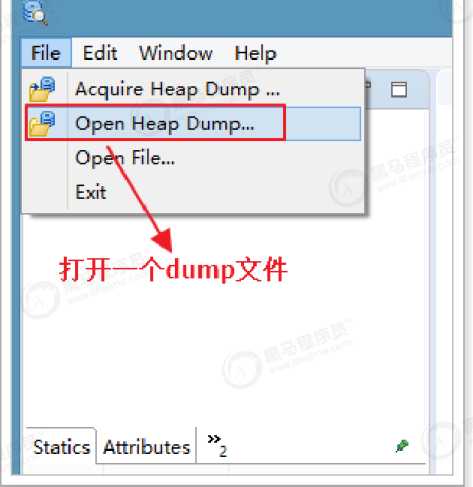
**争** MemoryAnalyzer.exe **警\_!** MemoryAnalyzer.ini **羣** notice.html

**园** ParseHeapDump.bat



5.43.基本使用

Getting Started Wizard



Getting Started

Choose one of the common reports below. Press Escape to close this dialog.

Automatically check the heap dump for leak suspects. Report what objects are kept alive and why they are not garbage collected.

**(•) Leak Suspects Report**

O **Component Report**

Analyze a set of objects for suspected memory issues: duplicate strings, empty collections, finalize^ weak references, etc.

C：l **Re-open previously run reports**

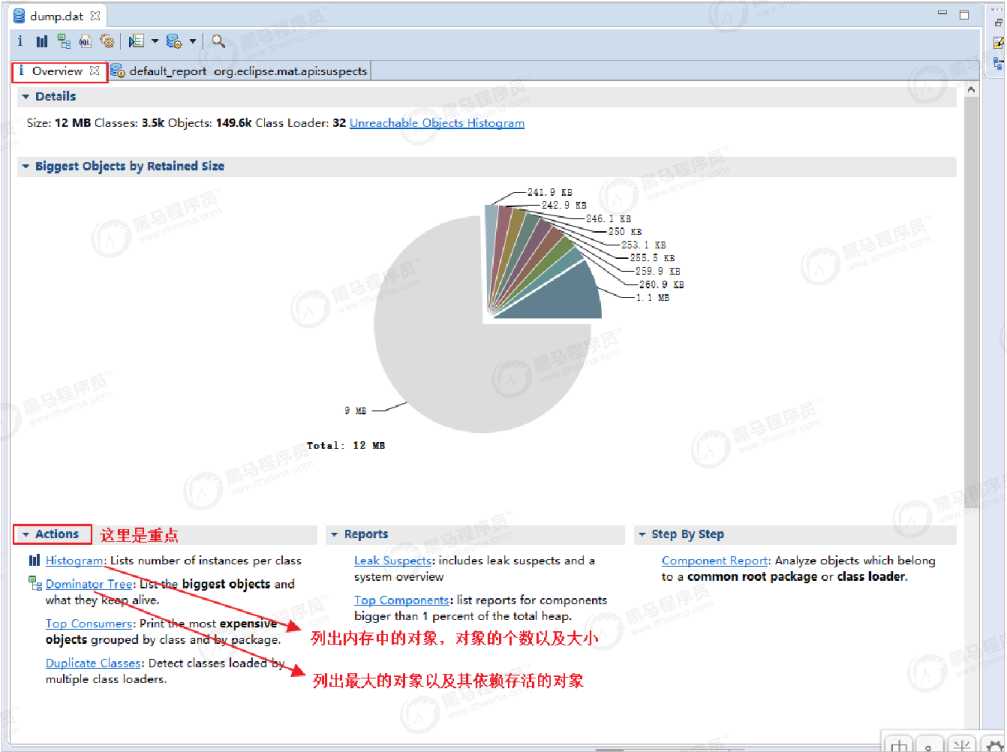
Existing reports are stored in ZIP files next to the heap dump.

0 Show this dialog when opening *a* heap dump.

Finish

Cancel

< Back Next >



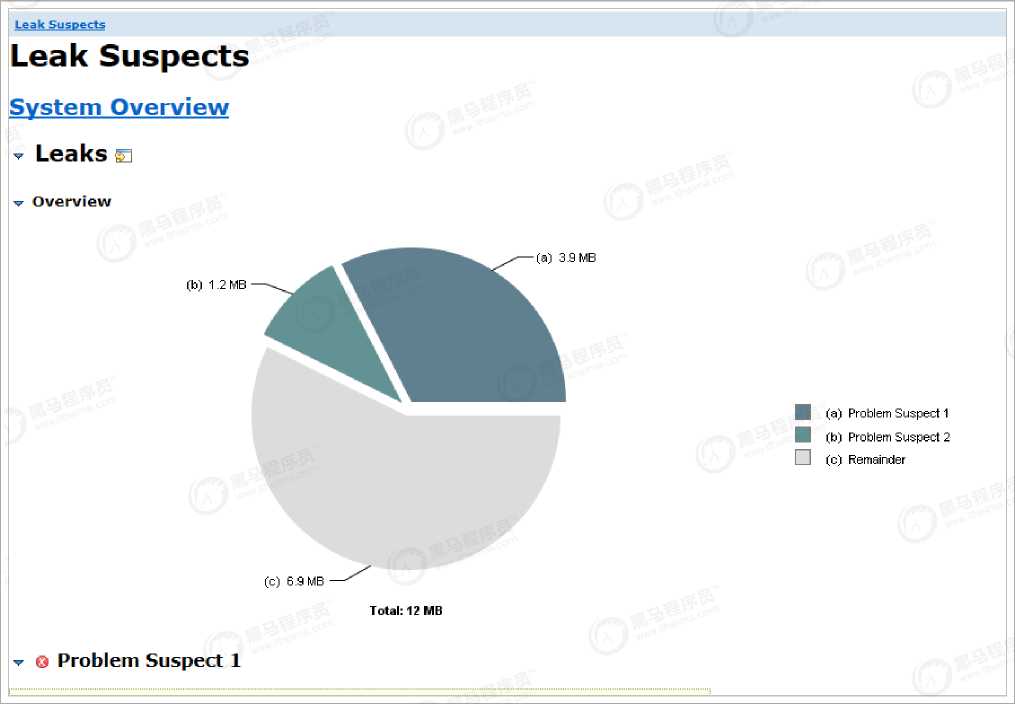
i Overview **瞌** default\_report org.eclipse.mataphsuspects III Histogram

|  |  |  |  |
| --- | --- | --- | --- |
| Class Name | Objects | T  Shallow He... | Retained H... |
| **!幕在这里通过正则进行搜素睥『**ic | | <Numeric> | <Numeric> \ |
| Q charQ | 32,094 | 7,575,720 | >=7,575,7... | |
| *Q* java.lang.String | 29,573 | 709,752 | >=4.907,7... |
| G byteQ | 878 | 654,704 | >=654,704 |
| 0 java.util.HashMapSN... | 17,161 | 549.152 | >=1,911.5... |
| O java.lang.reflect.Meth... | 3,720 | 327,360 | >=441,480 |
| Q java.lang.ObjectO | 4,712 | 259,504 | >=1,574.8... |
| O java.util.HashMapSN... | 995 | 206,472 | >=2.115.3... |
| 0 java. util.concurrent.C... | 4,305 | 137,760 | >=1,759,4... |
| 0 intO | 1,423 | 130,568 | >=130,568 |
| Q java.lang.StringQ | 1,269 | 109,064 | > =549,040 |
| @ java.util.concurrent.C... | 64 | 84,128 | >=1,834.3... |
| Q java.util.HashMap | 1,674 | 80.352 | >=2,166,6... |
| O java. util. Array Li st | 2,871 | 68,904 | >=308,840 |
| G java.Iang.ClassQ | 2,957 | 64,744 | > =54,744 |
| **。**java.lang.Object | 3,979 | 63,654 | > =63,654 |
| @ org.apache.tomcatut.. | 1,271 | 61,008 | >=91,456 |
| O j ava. util. Linked HashM... | 1,487 | 59,480 | >=125.576 |
| ® com.sun.org.apache.... | 1,671 | 53,472 | >=53,472 |

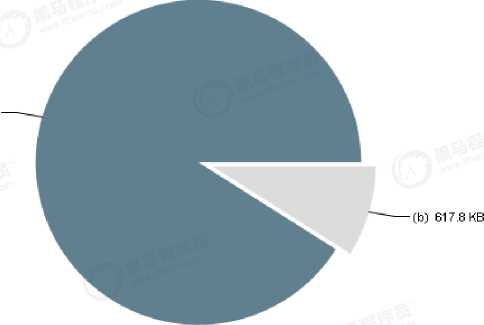
查看对象城它的依赖:

|  |  |  |  |
| --- | --- | --- | --- |
| 1 Overview default\_repo-rt org. ecl ipse. mat. a pi: suspects 阻 d om i n ator\_tree 疚 | | | |
| Class Name | Shallow Heap | Retained Heap | Percentage |
| 券 <Regsx> | <Numeric> | <Numeric> | 〈Numeric〉 |
| 」@1 org.apache.catalina.Ioader.StandardClassLoader @ 0xe345e5f0 | 80 | 1,129,024 | 8.97% |
| > D java.util.Vector @ 0xe2e6dc68 | 32 | L07L2S8 | 8.51% |
| t> D java.util.HashMap @ 0xe2e6ddb0 | 48 | 21,360 | 0.17% |
| > D java.util.Hashtable @ 0xe2e6d4d0 | 48 | 12.232 | 0.10% |
| > D sun.misc.URLClassPath @ Oxe2e6fOO8 | 48 | 11,464 | 0.09% |
| |> D java.util.WeakHashMap @ 0xe2e75d48 | 48 | 5J20 | 0.05% |
| t> D java.util.HashSet @ Oxe^eSdcft) | 16 | 624 | 0.00% |
| > D java.util.HashMap @ 0xe2e6ef58 | 48 | 608 | 0.00% |
| > D j ava. secu rity. Protect! on Dom a i n @ 0xe2fa0860 | 40 | 536 | 0.00% |
| > D j ava. secu rity. Protect! on Dom *a* i n @ OxeZfa^c/S | 40 | 536 | 0.00% |
| A D java.security.Protect!onDomain @ 0xe31a7510 | 40 | 536 | 0.00% |
| t> D j ava. secu rity. Protect! on Dom *a* i n @ Oxe2fa9,edO | 40 | 528 | 0.00% |
| A D java.security.Protect!onDomain @ Oxe2faa27O | 40 | 528 | 0.00% |
| > D java.security.Protect!onDomain @ Oxe353b5aO | 40 | 528 | 0.00% |
| > D java.security.ProtectionDomain @ Oxe353dd68 | 40 | 528 | 0.00% |
| A D java.security.Protect!onDomain @ Oxe336e248 | 40 | 520 | 0.00% |
| > D java. net. URL @ Oxe2e6f1bO fi I e:/tm p/a pa ch e-tom cat- 7.0.5 7/I i b/1 | 64 | 112 | 0.00% |
| t> D java.net.URL @ Oxe2e6f59O fi 1 e:/tm p/a p a ch e-tom cat- 7.0.5 7/1 i b/j | &4 | 112 | 0.00% |
| |> D java.net.URL @ 0xe2e6f878 file:/tmp/apache-tomcat-7.0.57/libA | 64 | 112 | 0.00% |
| t> D java.net.URL ⑥ 0xe2e6fb68 fi 1 e:/tm p/a pa ch e-tom cat- 7.0.5 7/1 i b/t | &4 | 112 | 0.00% |
| A D java.net.URL @ 0xe2&6fc50 fi 1 e:/tm p/a pa ch e-tom cat- 7.0.5 7/1 i b/j | &4 | 112 | 0.00% |
| > D java.net.URL @ Oxe2e6ff2O fi 1 e:/tm p/a pa ch e-to m cat- 7.0.5 7/1 i b/e | 64 | 112 | 0.00% |
| > D java.net.URL ⑥ 0xe2fa08a8 fi 1 e:/tm p/a pa ch e-tom cat- 7.0.5 7/1 i b/c | 64 | 112 | 0.00% |
| t> D java.net.URL @ 0xe2fa9a00 file:/tmp/apache-tomcat-7.0.57/lib/t | 64 | 112 | 0.00% |
| > D java.security.Protect!onDom*a\*n @ 0xe2e6dc88 | 40 | 104 | 0.00% |
| A D java.util.Vector @ Oxe2e6efOO | *32* | S8 | 0.00% |
| **A Total: 25 of 34 entries; 9 more** |  |  |  |
| - 1 ~*、* l j. I ；. | - A ■ If I . joa 。 f-1 [J | • r fi | *\*—t* \* r 广 | r |

查看可能存在内存泄露的分析:



**5.4.4**、导入分析



i：a：i 6.-1 MB

**Total: 6.7 MB**

I (a) Problem Suspect 1 [(h) Remainder

**▼ G Problem Suspect 1**

The thread **java.lang.Thread @ 0xff8alca8 main** keeps local variables with total size **6,408,816 (91.02<Vb)** bytes.

The memory is accumulated in one instance of **''java.Iang.Objectd"** loaded by **"<system class loader**〉".

The stack! race of this Thread is aval labile. See stmcktnacw.

**Keywords**

java. lang.Object[] Details »

可以看到，有91.02%的内存由Object口数组占有，所以比较可舞

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:

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
|  | **v Accumulated Objects in Dominator Tree** |  |  |  |  |
|  | **Class Name** | **Shallow Heap** | **Retained Heap** | **Percentage** |  |
|  | Ian a,Thread 亜 0xff8alca8 main | 120 | 6,408,816 | 91.02% |  |
|  | :niava.utiLArravList 回 0xff8a0610 | 24 | 6,340,000 | 90.04% |  |
|  | :.....可両3.前(**1,0**思由1091 回 Oxffdc$5aO | 456 | 6,339,976 | 90.04% |  |
|  | 1••“□  lava.Iana.Stnna @ 0xff8bb210 c3802a9l-2daf-4f9a-9189-308a2392c9100bbb5252-2125-4a9a-a676-04c8556efla4c56d2dee-1 ce9- | 24 | 72,040 | 1.02% |  |
| 42b3-a232-d 88fl41118e9a5641d 13-34d2-472a-a653-6ee23 e06dbac91845 eb7-4a79-43a0-b20 6-be763a3da2a 1 le8189b7-94ff-4832- |
| 8b5c- 506b21 e7bc3173724d 1c- 5620-4 lcb-b5 5d-7168c08fcl9c8cf9 … |
|  | :•…□  iava.Iana.Strina @ 0xff8ccbe8 011faf89-1755-46dl-8d0e-76c8e4addl78359612ca-a485-43d0-b7e4-ee457b848386e665190d-5d09- | 24 | 72,040 | 1.02% |  |
| 4c64-9068-9b5f95471aed9d5dc246-5436-4e3c-ac9f-d84a2c4b4a5a00f48264-9256-47fb-ba0b-fb2ff0e6d0c6a27aa633-ed8e-4f6e- |
| 8e81-89bc23e413ece8d58004-8fbO-4322-b965-el54ee9829c34fla... |
|  | 1••“0  iava.Iana.Strina @ 0xff3de6b8 bb484927-18cc-42a9-37da-7bb615ca8cbece8f45a3-955d-46f4-aa57-95b540892780929a624c- 18f8- | 24 | 72,040 | 1.02% |  |
| 49fd-abe0-f42bbOfa79ad308e854c-76ab-46db-al9e-8d7ea323e3a4497ed59c-ccl5-462e-9092-99e53fffl5d5aa869fcd-774f-4467- |
| 811f-2513fc4ad0c597bd9970-fal9-447f-821c-3f43e7243312d024... |
|  | 1•…Q  iava.Iana.Strina @ 0xff8f00c8 7ca2287b-8518-4Sd5-9934-b4499e28fd048b298195-eb78-4fl3-a735-d956a2b00ab6b9c379bc-825e- | 24 | 72,040 | 1.02% |  |
| 40d3-9cb9-dfb54bf6ea 8b45f3f6f5-90bd-4d50-889c-e41d08d4a46B15027915-a63c-4b7 l-921b-e593861 dbfcff78 lad7b-9bec-4ffa- |
| 8686-2aa le4eec3c2W0b7W959-f914-4d U-bab4-9e9 6aW671 aea3c0b... |
|  | :D  iava.Iana.Strina @ 0xff901a30 3240f76e-84bc-4161-9e56-51d9f51e0585alf05070-74da-4292-bd5e-3592034e784de584df7f-314e- | 24 | 72,040 | 1.02% |  |
| 48ab-b0df-442al4de6b7609e7241e-dd56-4836-9aaf-52721fc2dl5bce243cc2-e3bb-41ef-9e22-3f0ead0a20bf3e4b5964-7d8f-45d5- |
| *9* 54a-dfb2d 5725978 le5fe35d-0bde-41e5-81 b3-8c968ad 2f7dSc2ea … |
|  | 1•…Q  iava.lana.Strina @ 0xff913398 98f2928d-1457-4638-adde-lddfd0d857be9Oaa7dfl-b378-45e0-ael7-f33365c942ce848721db-cbb7- | 24 | 72,040 | 1.02% |  |
| 4cc8-8eee-f760a636b95c7f38f8df-66d4-4609-98e4-e88aOba7a253cdbcde8b-c403-42c4-9746-4bd907f8cd7f4bb7a6fb-85a8-4a2b- |
| 8all-9c95087fa245853e5cdd-d3d5-4255-8f0b-e500cb2c63f71f6e... |
|  | 1••“0  iava.Iana.Strina @ 0xff924d00 e2c41499-5a97-4093-afcc-ea9885efd9a30a6b4d44-e43c-4979-ad0b-c56aa?af3alafbe4fd35-390e- | 24 | 72,040 | 1.02% |  |
| 47ec-969a-a968dda0efdfe9eb8369-77fb-42a5-bcll-a958c9eacfbde5ac75d8-e8a4-4al4-9493-2fd5eb7165f372485232-c557-4bO4- |
| a831-740e64b89603fff51efl-5fld-44db-9blb-230bddbt>47e24afl... |
|  | HQ .一 鴛一 |  |  |  |  |

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中的线档丸行情况,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，没有缶可的输出，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个爨,Threadl拿到了obj1锁，准备去拿。bj2锁时，obj2已经被Thread2锁定,所以发送了死 锁。

package cn.1tcast.jvm;

public class TestDeadLock

private static object

objl = new

object();

private static object

obj2 = new

object();

public static vold main(string]] args) {

new Thread(new ThreadlO) . start();

new Thread(new Thread2()).start();

}

private static class Threadl implements Runnable(

@overri de

public vold run() ( synchronized (objl)(

System, out. println("Threadl 拿至U了 objl 的锁!");

try {

//停顿2秒的意义在于，让Thread2线程拿到obj2的锁 Thread.sieep(2000);

} catch (interruptedExcept!on e) (

e.pri ntstackTrace();

}

synchronized (obj2)( System, out. pri ntl n ("Threadl 拿到了 obj2 的锁！ ”)； }

}

}

}

private static class Thread? implements Runnable(

@overri de

public vold run() ( synchronized (obj2)(

System, out. println("Thread?拿至U了 obj2 的锁!");

try {

//停顿2秒的意义在于，让Th read丄线程拿到obj丄的锁 Thread.sleep(2000);

} catch (interruptedException e) (

e.pri ntstackTrace();

}

synchronized (objl)( System, out. pri ntl n ("Thread?拿到了 obj 1 的锁！ ”)； }

}

}

}

}

6.2、运行并且查询进程编号

島 **TestDeadLock**

**"C:\Program Files\Java\jdkl.0.0\_144\bin\java.exe" ...**

Threadl 拿至0了 obj 1 的锁！

Thread2 拿至【疗 obj2 而锁！

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>j ps -1

7056 org.jetbrains.jps.cmdl1ne.Launcher

10500

6580 jdk.jcmd/sun.tools.jps.Jps

7480

12076 itcast-jvm-app-1.0-SNAPSHOT.jar

12652 cn. i tcast. 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-1evel deadlock:

"Thread-l":

waiting to lock monitor 0x00000000026c3ee8 (object 0x0000000780198bb0, a java.lang.object),

which is held by "Thread-0"

"Thread-O":

waiting to lock monitor 0x00000000026c40f8 (object 0x0000000780198bc0, a java.lang.object),

which is held by "Thread-1"

Java stack information for the threads 11sted above:

"Thread-l":

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.1ang.Thread.run(Thread.java:748)

"Thread-O":

at cn.itcast.jvm.TestDeadLock$Threadl.run(TestDeadLock.java:29)

* waiting to lock <0x0000000780198bc0> (a java.lang.object)

-locked <0x0000000780198bb0> (a java.lang.object)

at java.1ang.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
* Threadl 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

,蜜呈信息

* Dump堆（本地进程）

, Dump缴呈（本地进程）

•打开堆Dump。堆Dump可以用jmap来生成。

,打开蜜呈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gijjvisualvm.exe ,双击打开即可。

c.exe

O jmc.ini

B jps.exe

H jrunscript.exe **回** jsadebugd.exe

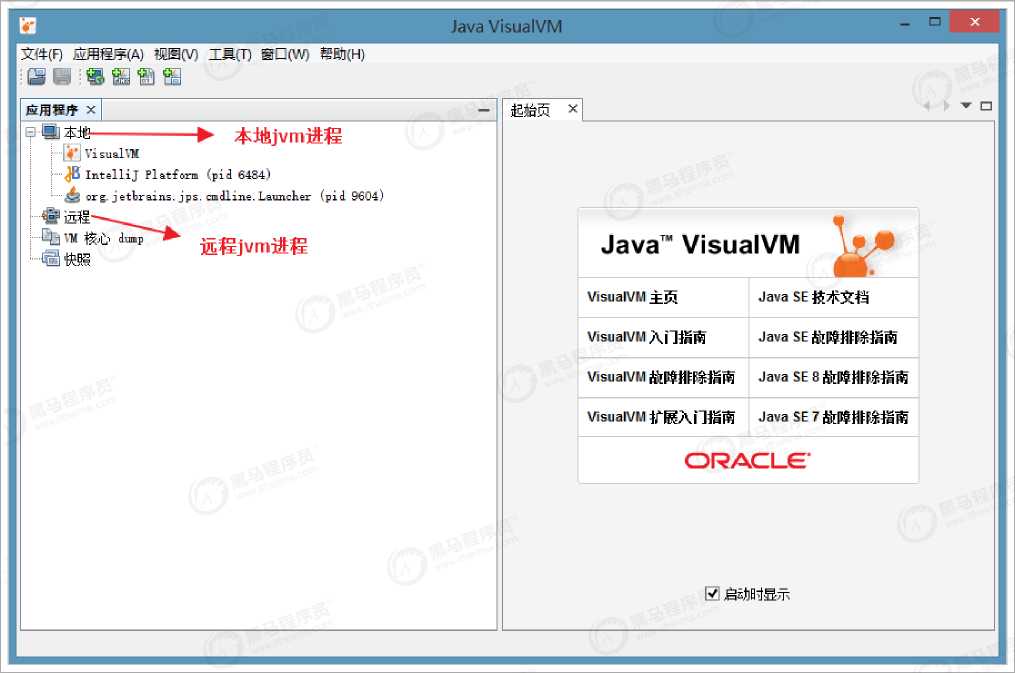
B jstack.exe **叵**3 jstat.exe **回** jstatd.exe jvisual vm.exe I

H keytool.exe **叵**3 kinit.exe H klist.exe

H ktab.exe

y msvcr100.dll

**叵**3 native2ascii.exe *會糾■做兀*



7.1.2、查看本地进程

起始页 *X* | 那 IntelliJ Platform (pid 6484)~可

風摭述I感监视|圄钱程I為抽样器I。

*二* IntelliJ Platform (pid 6484)

概述 回保存的数据回

**PID：** 6484

主机 localhost

主类： ＜未知〉

爹数：〈无〉

**JVM：** Java HotSpot(TM) 64—Bit Server VM (25.144—bOl, mixed mode) Java：版本 1. 8. 0\_144,供应商 Oracle Corp or at ion

—显示**jvm**的一些基本信息

**Java Home** 目录：C:\Frogrsm Files\Java\j dkl. 8. 0\_144\jre

JV标志： <无>

出现00ME时生成堆dup：启用 |卩见参数|系统属性

**-5ms128k**

线程**Bmp：** 0 堆 **Dump：** 0 **Profiler** 快照

**-X«75Q«**

**—XX : ReservedCodeC&ckeSi ze^40m**

**—XI: -1-UseCondarkSveepGC**

**-XI: SoftRefLRVPolicylSPewMBWO**

**—ea**

**vm**的参数信息

**—Dsun. io. useCanonCaclies^alse**

**—Dj ava. net. pr ef er IP v4S t ack=**true

**—XX: +He apDimpOnOiitO fMenoryErr or**

**—XX Omi tSt&ckTracelnFastTliroir**

**—ja.vaagent：C： \Prograa Files\JetBra.ins\IntelliJ IDEA 2017. 3. 2\bin\Jetbra.insCrack~2. 6. 10~release-enc. jar**

**—Djb. va.0pti onsFile^I：** \Frogram F11 e s \ J"e tBr al ns \lnt elli J IDEA 2017. 3. 2\l)Ln\i<leaJ64. exe. vmoptions

**—Xbo4tcla.ssps.tli/s.： C: \Progra> FileE\JetBr&inE\IntelliJ IDEA 2017. 3. 2\lib\boot. j ar**

**—Di dea. jre.** ckeck=true

**—Di de. native. 1 a.uncker=true**

**—Did”, patks. select«r=IntelliJIdea2017. 3**

**—XX : Err or P i 1** : \Users\znijun\j av a\_err or \_i n\_i dea\_%p. log

**—XX : He&pDuapFatli^: \Users\zhi jun\java\_error\_in\_idea. hprof**起始页 X I JB IntelliJ Platform (pid 6484) 田概迷丨函监视I目线程I臨抽样器I。&。理1迎

\* IntelliJ Platform (pid 6484)

概述 回保存的数据0

**PID：** 6484

主机： localhost

主类： ＜未知〉

参数：，:无:〉

**JVM：** Java HotSpot(TM)64一Bit Server VM (25.144-b01? mixed mode) **Java：**版本 1.8. 0\_144j 供应商 Oracle Corp or at ion

显示**jvm**的一些基本信息

**Java Hoae** 目录：C: \Program Files\Java\jd]cl. 8. 0\_144\jre

JV标志：〈无〉

出现00ME时生成雄赢-p：启用

X

线程**Dup：** 0 **ib** D™p： o **Profiler** 快照：

I*刑*参勲|系统属性

**-£ms128b**

**-Xm750«**

**—XX: ReservedCodeC&ckeSi z・W40m**

**—XI: -1-UseCondarkSveepGC**

**-XX: SoftRefLRUP olicyKSP erWI^O**

**—ea.**

**um**的参数信息

**—Dsun. is. useCanon.Caches=£al5e**

**—Dj ava. net. pr e£, er IP v4S t ack=**true

**—XX: +Me apDimpOnOutO fdea oryErr or**

**—XX: —0\*i tStackTracelnPastThroir**

***~j*a.vaagent: C : \Progr aa Files\JetBrains\Intelli J IDEA 2017. 3. 2\bin\Jetbr ainsCrack-2. 6. 10-rel ease-enc. j ar**

**—Djb. n.0pti onsKile^!：** \Frogram Filee\JetBrains\IntelliJ IDEA 2017. 3. 2\l)in\ideaJ64. exe. vmoptions

**—Sbootclasspath/a： C: \Prograa File5\JetBra.ins\IntelliJ IDEA 2017. 3. 2\lib\boot. j *ar***

**—Di dea. jre. ckeck=true**

**—Di de. native. 1 &uneker=true**

**—Didea. patks. select**or=IntellIjldeaSOl7. *3*

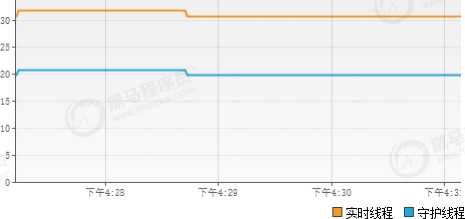
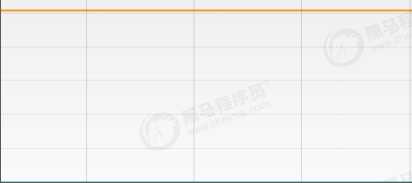
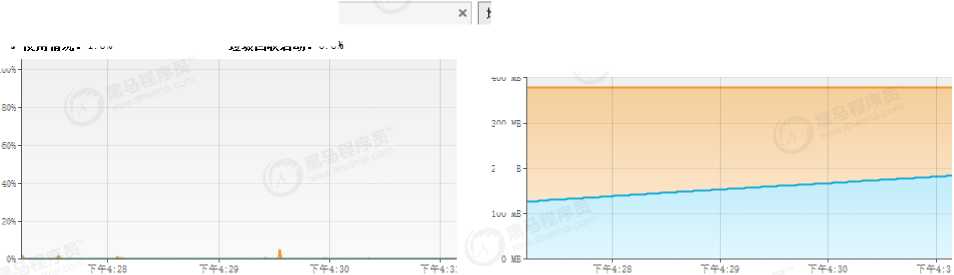
**—XX : Err orFi 1** @弍:\Users\zhijun\j ava\_error\_Ln\_idea\_%p. log

**—XX: HeapDuapF** ath^Z: \UEerE\zhijun\j av a\_err or \_in\_i de a. hprof

7.1.3、查看CPU、内存、类、线程运行信息

田旋 画监视|圏线程I轟抽样器|。旺。珪1成

C IntelliJ Platform (pid 6484)



监视

Hcpu回内存面类叵］线程

|执行垃圾回收［顶峰

正常运行时间：22小时23始57秒

兰

CPU

**CPU**使用情况：**1.0%**

垃圾回收S=h： 0.0%

已使用:195,196, 752个字节

300 ME

100四

■ CPU使用情况■垃圾回收活动

□雄大小口使用的堆

线程

堆 **Metaspace**

大小：398,184, 448个字节 最大：786, 432, 000个字节

10M

200 ME

已装入的总教=51,067 已卸载的总卸=2,903

50.000

40.000

30.000

20.000

10.000

共享的已装入教：。 共享的已卸载教：。

下午4:29 下午4:30 下午4:3

■已装入的类的总数■共享的已装入类数

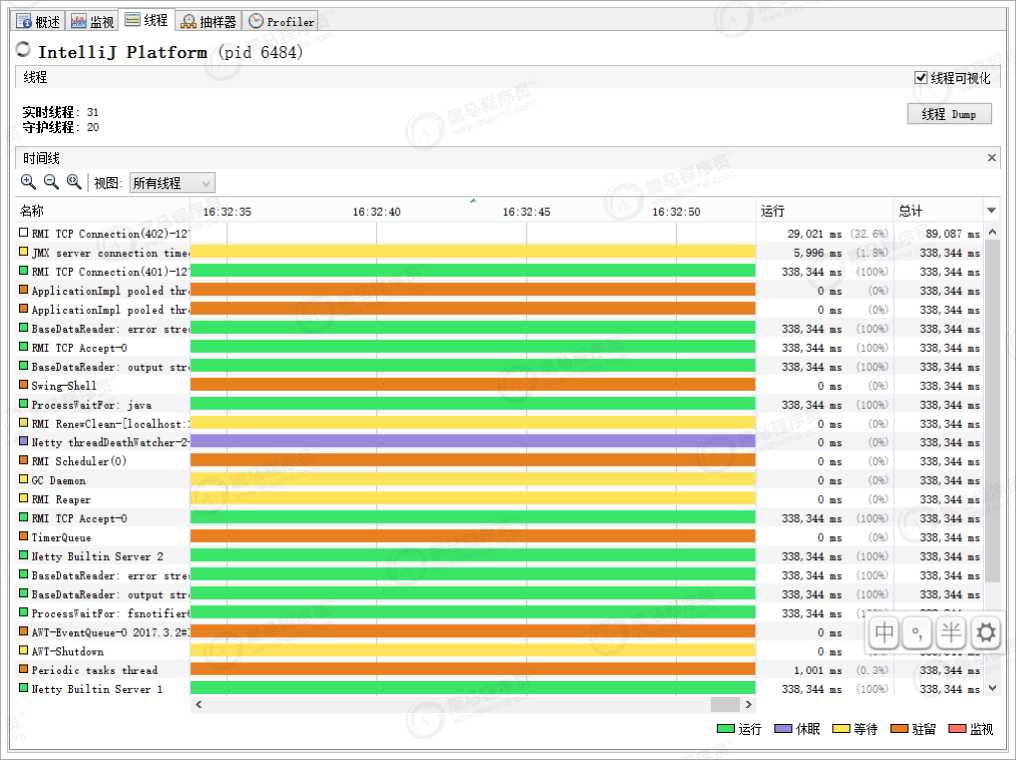
下午4:28

下午4:30

活动:31 实时诚值：54

守护进程：20 已启动的总教：340

7.1.4、查看线程详情



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

2018-09-11 17:00:54

Full thread dump Java HotSpot(TM) 64\_Bit Server VM (25. 144\_b01 mixed mode): 'EMI TCP Connect Ion(404)-12 7. 0. 0.#S56 daemon prio=5 as\_prio=0 t i <1=0x000000003 a064800 nid=0xldh0 runnahle [0x00000000241£e000]

1 ang. Thread. State: RU1HTAELE

at j ava. net. SocketlnputStrearn. socketReadO (Native Method)

at j ava. net. SocketlnputStream. socketRead(SocketlnputStream. java： 116)

at j ava. net. Sock etlnputStr earn, re ad (Sock etlnputStr earn, java： 171) at j ava. net. SocketlnputStream. read(SocketlnputStrearn, j ava： 141) at java. io. BufferedlnputStream. fill(Eu££eredInputStream.. java：246) at j ava. io. Buffers dlnputS tr earn, read (Buffer e dlnputS tr e am. j ava： 265 )

-locked <0x00000000d5d91 Jb8> (a java. io. Buffer®dlnputStresm)

at j ava. io. FilterlnputStream. re&d(FilterInputStream. j ava： 83)

at sun. rmi. tr ansport. t cp. TCPTr ansport. handl eMess age s (TCPTr anspart. j ava： 550)

at sun. mi. transport. tcp. TCFTransportSCoimectionMandler.ruuO(TCFTranspoart. java：826)

at sun. rmi. transport. tcp. TCPTran.Eport$Connecti onHandler. 1 ambda$rtm$0 (TCPTransport. j ava： 683)

at sun. mi. traixspcrt. tcp. TCPTr anspcrt$C onne cti onHandler$$Lsnibda$1463/l 745933817. run (Unknown Source) at j ava. security. AcceEsController. doFrivileged(Uative Method.)

at sun. rmi. transport. tcp. TCPTran.Eport$Conneeti onHandler. rim(TCPTranspoart. j ava： 682)

at j ava. util, concurrent. ThreadPoolExecutor. run^orker (Thre&dPoolExecutor. j ava： 1149)

at j ava. util, concurrent. Thr e adP o olEx e cut or $T ark er. run (Tbr e adP o olEx e cut oar. j ava： 624)

at j ava. lang. Thread. run(Thread. java： 748)

Locked ovnable synchronizers：

一 <0x00000000d409b0f0> (a java. util, concurrent. ThreadPoolExecutor$Toarker)

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。

[昌槪述 圈监视 巨线程 皿抽样器 **0 Profiler**

C Intellij Platform (pid 6484)

抽样器

口设置

抽样： | Qcpo 「口内存 园停止

状态:

正进行內存抽样

执行GC堆Dump

雄柱状囲：每个线程分配

敏僧里（5快照

24, 055

实例J： 3,349,099

字节：174,297, 840

类名

字节[«] ▼

子T>

3

2

2

1

6

5

5

2

1

1

1

1

1

1

1

1

**char []**

**int[]**

**j ava. lang. Obj ect []**

**byte[]**

**java. util. TreeH&p$Entry**

**java. lang. String**

**java. lang. Class**

**java. io. ObjectStreaaClassSTealcClassKey**

**com. intellij. util. text. ByteArr&yCkarSequence**

**com. intellij. util, containers. IntObjectLinkedliap$ll&pEntry java. util. TreeMapSKeylter&tor**

**org. j dom. Attribute**

**long[]**

**java. util, concurrent. ConcurrentHasUap$lode**

**java. util. LinkedH&sU&pSEntry**

**java, util HasUap$Iode**

**java. util. TreeH&p**

7.2、监控远程的jvm

VisualJVM不仅是可以I雌本地jvm进程，还可以I雌远程的jvm进程，需要借助于J MX技术实现。

**7.2.1**、 什么斷**X?**

J MX （Java Management Extensions ,艮卩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。 J MX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、网络和月艮务 管理应用。

7.2.2、 监控远程的tomcat

想要监控逡呈的tomcat ,就需要在远程的tomcat进行对J MX配置，方法如下：

#^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.authenficate=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远接端口

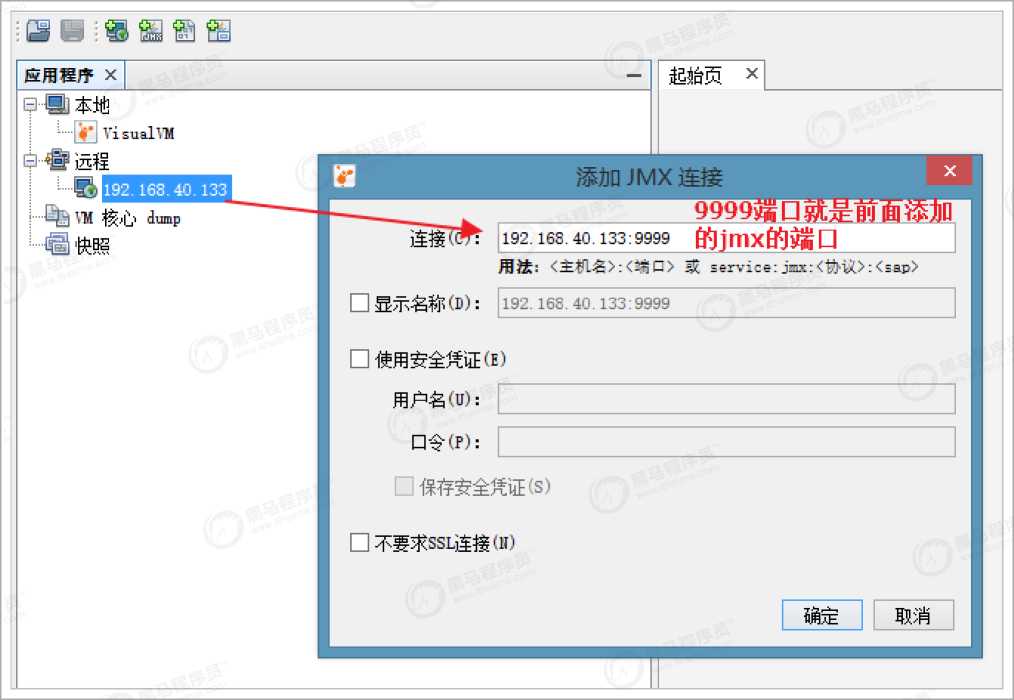
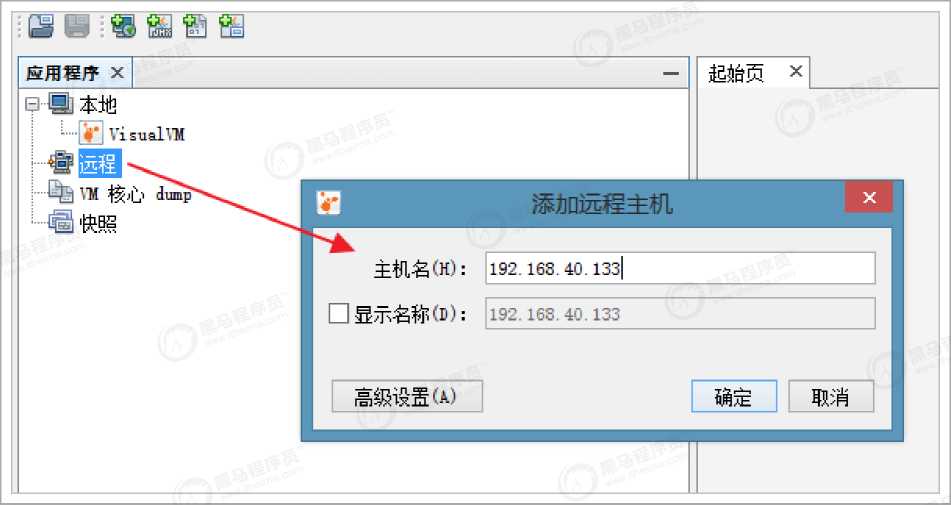
#-Dcom. sun. management .jmxremote. authenti cate=fal se :不进彳亍身份认证，任何用户者阿以连接 #-Dcom. sun.management, jmxremote. ssl=false :不使用ss】

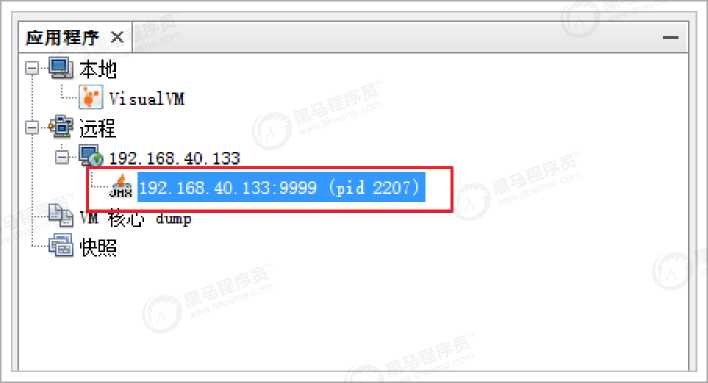
保存退出，重启tomcat。

7.2.3、使用VisualJVM连接远程tomcat

添加远程主机:

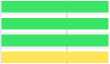
在一个主机下可能会有很多的jvm需要监控，所限下来要在该主机上添加需要监控的jvm :





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



* Monitor Ctrl-Break
* Attach Listener

口 Signal Dispatcher

□ Finalize

C Reference Handler SEMI TCP Connection(2)-192.:

| 1； 998 ms  0 ms | ;(50%) 1  ; (0%> |
| --- | --- |
| 3； 997 mw | ;(100%> |
| 3； 997 mw | ;(100%) |
| 3; 997 ms | i (100%) |
| 0 ms | i (0%> |
| 0 ms | ;(0%) 1 |
| 3 997 ms | :(10 蹄 |
| 3； 997 ms | ;(100ft) |
| 3； 997 ms | I (100%> |
| 0 ms | 1 (0%) |
| 0 ms | i (0%> |
| *2,*997 ms | ;(100%> |

3； 997 ms

3, 997 ms

3j 997 ms

3； 997 ms

3； 997 ms

3； 997 ms

3； 997 ms

3, 997 ms

3j 997 ms

3； 997 ms

3； 997 ms

1. ms
2. ms

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| **O cn. itcast.** jvhl **TestDeadLock** (pid 13220) | | | | | |
| 线程 | | | 线程可视化  线程Dump | | |
| 案时线程：13  寺炉线程：10 | 颇到死靶  生成一个线程口ump以義取更参信息。 | |
| 时働  収■懺|视囹:|所有线程 |  |  |  |  | X |
| 名称 9:57:00 9:57:05 | | 9:57:10 9:57 | :15 | 9:57:20 运行 | 总计 ▼ |

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：



插件

[新可用插件(18)已下载已安装设置

匚检查最新版本(F)]

搜索(S): |

□ □□□

Visual GC

名称

Vi sualM(-Gl assfish Vi sualM!-Extensi ons Startup Profiler BTrace Workbench V~isualVM~Seeuritv

类别

Application Platform Profiling Profiling Security

Tools

w 社区提供的插件

Visual^K-BufferMoni tor Threads Inspector VisualMJ-JConsole

Vi sualVM-MBeans KillApplication

Tracer-J'vmstat Probes Tracer-Monitor Probes

Tracer-Swing ProLes

Tracer-I0 Probes Tracer-Collections Probes

Tracer-J\M Probes OQL Syntax Support

Tools Tools Tools Tools Tools Tracer Tracer Tracer Tracer Tracer Tracer

成本：2.1.2

作者： Jiri Sedlacelc 日鄭：13-11-25

源：Java VisualVM插件中心 主页: https ：//visualvm. Java. net

抽件描迷

Integration of the Visual GC tool into ViEual^H. Visual GC user interface is displayed for each local or remote application with performance counters available via jvmst&t API.

The Visual GC tool attaches to an instrumented MotSpot JTM and collects and griaphically displays garbage collectiaib cl ass loader, and KotSpot compiler performance data.

See the Visual GC home page for information on using the tool: visualgc — Visual Garbage Collection Monitoring Tool.

安装(I) 已迭择1个插件，43M

关闭里 帮助(K)

7.4.1、编写代码

package cn.itcast.jvm;

i mport java.uti1.ArrayLi st;

i mport java.uti1.Li st;

public class TestHeap (

public static vold main(string[] args) (

List<User> userList = new ArrayList<>();

while (true){

User user = new User();

user.setld(lL);

user.setusername("user");

user.setPassword("pass");

if (System. currentTimeMi His() *% 2* ==0 )( use「List.add(use「)；//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.sieep(l);

} catch (interruptedException e) (

e.pri ntstackTrace();

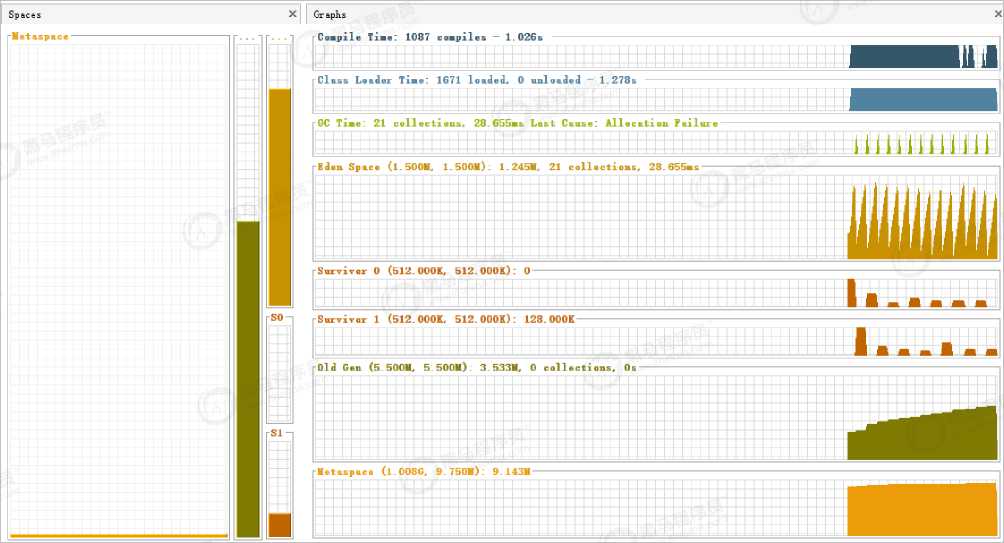
7.4.2、运行测试

运行参数：

-Xms8m -Xmx8m -XX：+HeapDumponoutofMemoryError

|  |  |
| --- | --- |
| 1 esirieap . | mairuj |
| TestHeap |  |

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| add. | **to** | **list,** | size = | 502 |
| add. | **to** | **list,** | size = | 503 |
| add. | **to** | **list,** | size = | 504 |
| add. | **to** | **list,** | size = | 505 |
| add | **to** | **list,** | size = | 506 |
| add | **to** | **list^** | size = | 507 |
| add. | **to** | **list^** | size = | 508 |
| add. | **to** | **list^** | size = | 509 |
| add. | **to** | **list^** | size = | 510 |
| add. | **to** | **list,** | size = | 511 |



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。

1. , 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。HotSpot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。 [↑](#footnote-ref-2)
2. 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啲命名方式,而采用"JDKx”方式进行命名。 [↑](#footnote-ref-3)