

丁应思

dingyingsi

QQ交流群:256924514

博客园 首页 新随笔 联系 管理 订阅 XML

随笔- 333 文章- 0 评论- 46

深入理解JVM—JVM内存模型

我们知道，计算机CPU和内存的交互是最频繁的，内存是我们的高速缓存区，用户磁盘和CPU的交互，而CPU运转速度越来越快，磁盘远远跟不上CPU的读写速度，才设计了内存，用户缓冲用户IO等待导致CPU的等待成本，但是随着CPU的发展，内存的读写速度也远远跟不上CPU的读写速度，因此，为了解决这一纠纷，CPU厂商在每颗CPU上加入了高速缓存，用来缓解这种症状，因此，现在CPU同内存交互就变成了下面的样子。

昵称: [丁应思](#)
园龄: [5年8个月](#)
粉丝: [169](#)
关注: [0](#)
[+加关注](#)

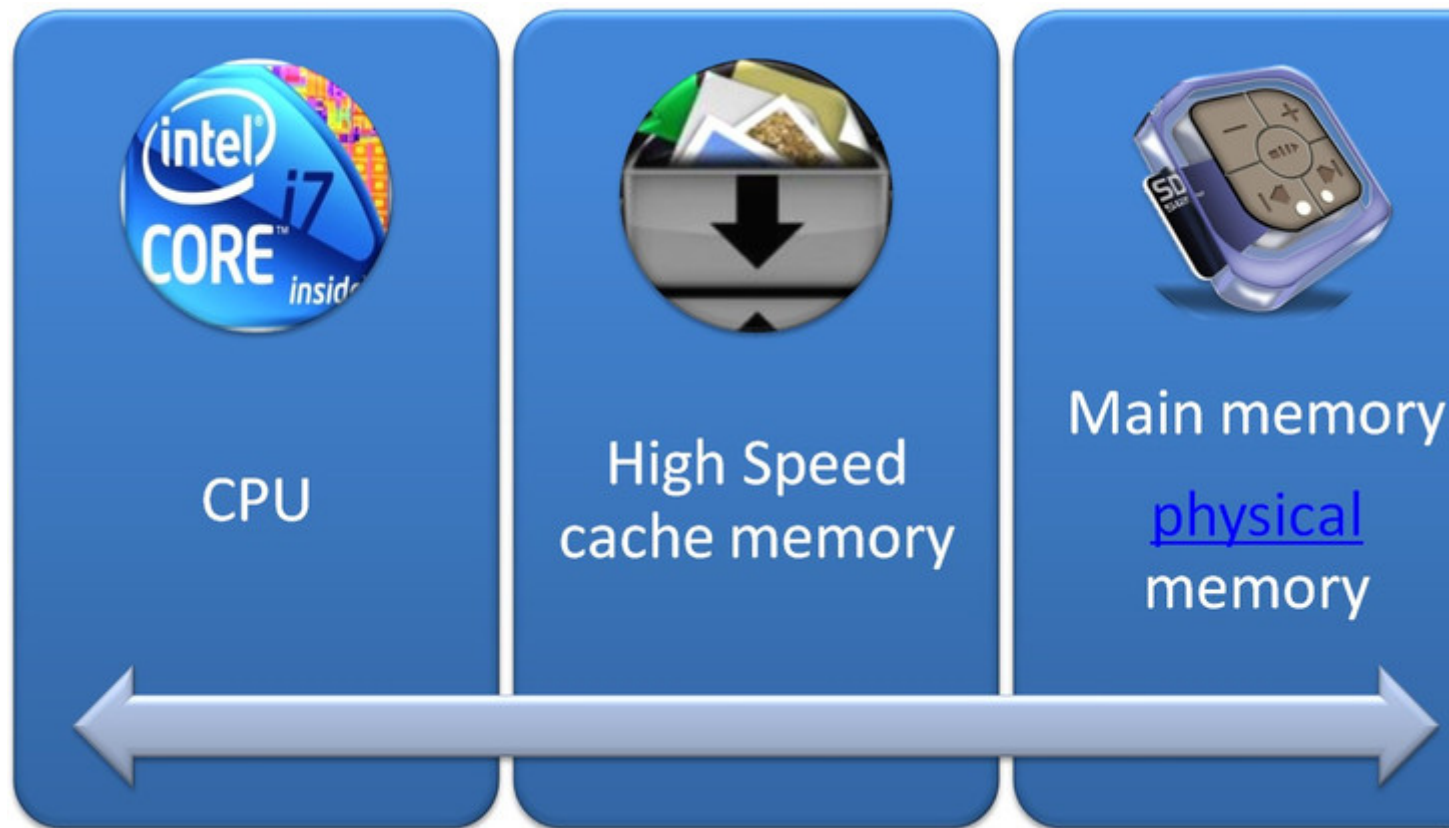
2018年9月						
<	一	二	三	四	五	六
26	27	28	29	30	31	1
2	3	4	5	6	7	8
9	10	11	12	13	14	15
16	17	18	19	20	21	22
23	24	25	26	27	28	29
30	1	2	3	4	5	6

搜索

<input type="text"/>	<input type="button" value="找找看"/>
<input type="text"/>	<input type="button" value="谷歌搜索"/>

常用链接

[我的随笔](#)



同样，根据摩尔定律，我们知道单核

CPU的主频不可能无限制的增长，要想很多的提升新能，需要多个处理器协同工作，[Intel](#)总裁的贝瑞特单膝下跪事件标志着多核时代的到来。

我的评论
我的参与
最新评论
我的标签

随笔分类

AJAX(1)
Android
Ant(1)
Apache(1)
C++(3)
CSS
Docker(1)
Eclipse(2)
Flex(17)
git(8)
Gradle(4)
Hadoop(1)
Hibernate(15)
HTML(4)
HttpClient(1)
JavaScript(12)
JavaSE(34)
jQuery(1)
JSON(1)
JSP(2)
JVM(2)
Linux(46)
Linux C(7)
Lucene(5)
Maven(25)
MongoDB(1)
mybatis(4)
MySQL(13)
nexus(1)
Nginx(2)
Oracle(39)
Pro * c(1)



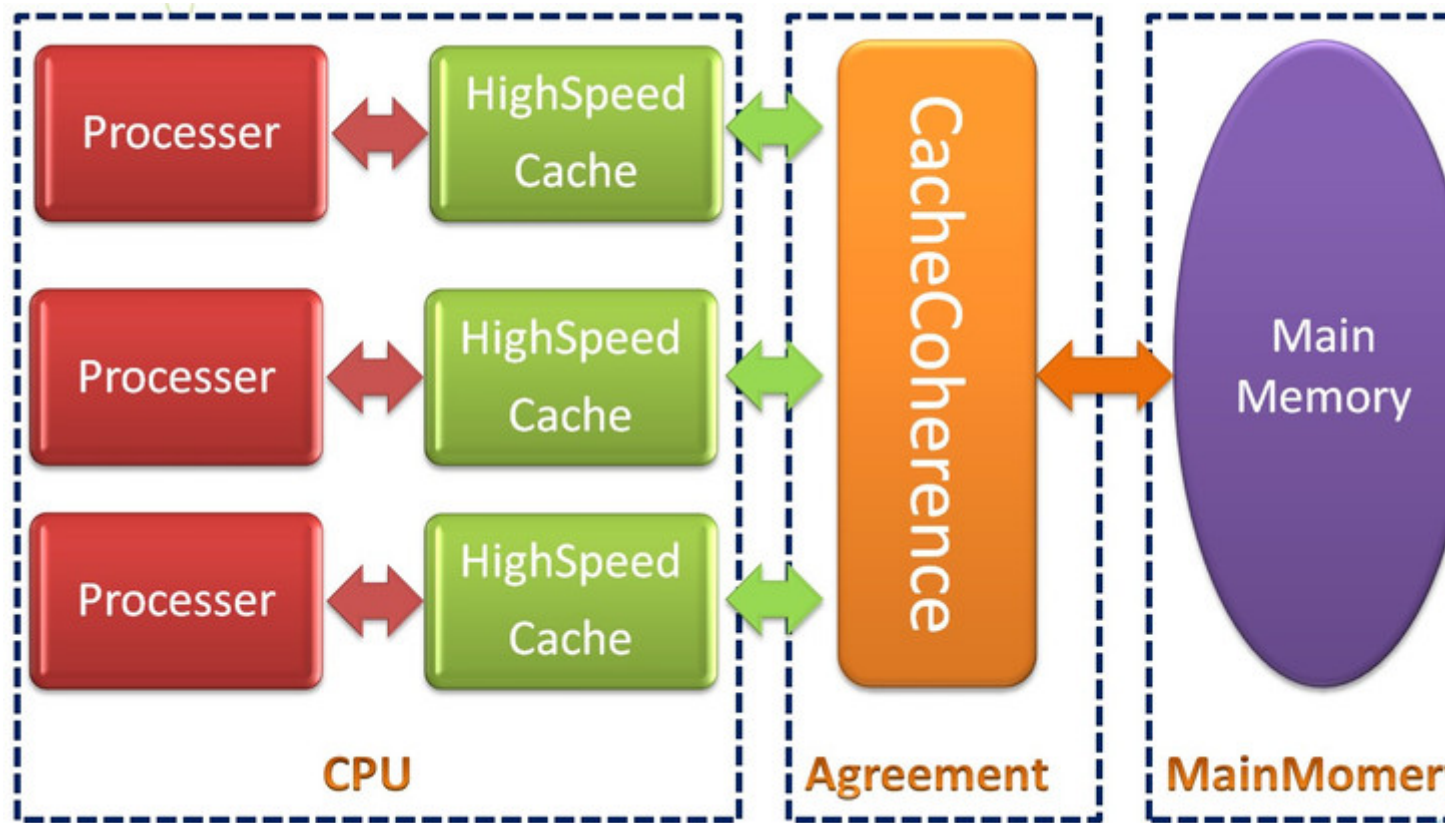
基于高速缓存的存储交互很好的解决了处理器与内存之间的矛盾，也引入了新的问题：缓存一致性问题。在多处理器系统中，每个处理器有自己的高速缓存，而他们又共享同一块内存（下文称主存，

main memory 主要内存），当多个处理器运算都涉及到同一块内存区域的时候，就有可能发生缓存不一致的现象。为了解决这一问题，需要各个处理器运行时都遵循一些协议，在运行时需要将这些协议保证数据的一致性。这类协议包括MSI、MESI、MOSI、Synapse、Firely、DragonProtocol等。如下图所示

quartz(1)
Servlet(5)
Spring(7)
SQL Server(2)
SQLite(1)
Struts(12)
SVN(6)
tomcat(1)
WebLogic(24)
WebService(7)
XML(3)
其他(7)
正则表达式(2)

随笔档案

2018年8月 (4)
2018年7月 (2)
2018年4月 (5)
2018年3月 (3)
2017年11月 (1)
2017年8月 (1)
2017年2月 (3)
2017年1月 (2)
2016年12月 (1)
2016年11月 (3)
2016年10月 (6)
2016年9月 (2)
2016年8月 (2)
2016年7月 (5)
2016年6月 (2)
2016年5月 (4)
2016年3月 (5)
2016年2月 (1)
2016年1月 (1)
2015年12月 (1)
2015年10月 (1)
2015年9月 (4)
2015年8月 (10)



Java

虚拟机内存模型中定义的访问操作与物理计算机处理的基本一致!

2015年7月 (4)
 2015年6月 (4)
 2015年5月 (5)
 2015年4月 (1)
 2015年3月 (1)
 2015年1月 (3)
 2014年12月 (1)
 2014年10月 (3)
 2014年8月 (5)
 2014年7月 (3)
 2014年6月 (10)
 2014年5月 (17)
 2014年4月 (9)
 2014年3月 (10)
 2014年2月 (1)
 2014年1月 (2)
 2013年12月 (3)
 2013年11月 (9)
 2013年10月 (4)
 2013年9月 (9)
 2013年8月 (16)
 2013年7月 (20)
 2013年6月 (19)
 2013年5月 (20)
 2013年4月 (24)
 2013年3月 (12)
 2013年2月 (48)
 2013年1月 (1)

最新评论

1. Re:JVM 内存初学 (堆(heap)、栈(stack)和方法区(method))

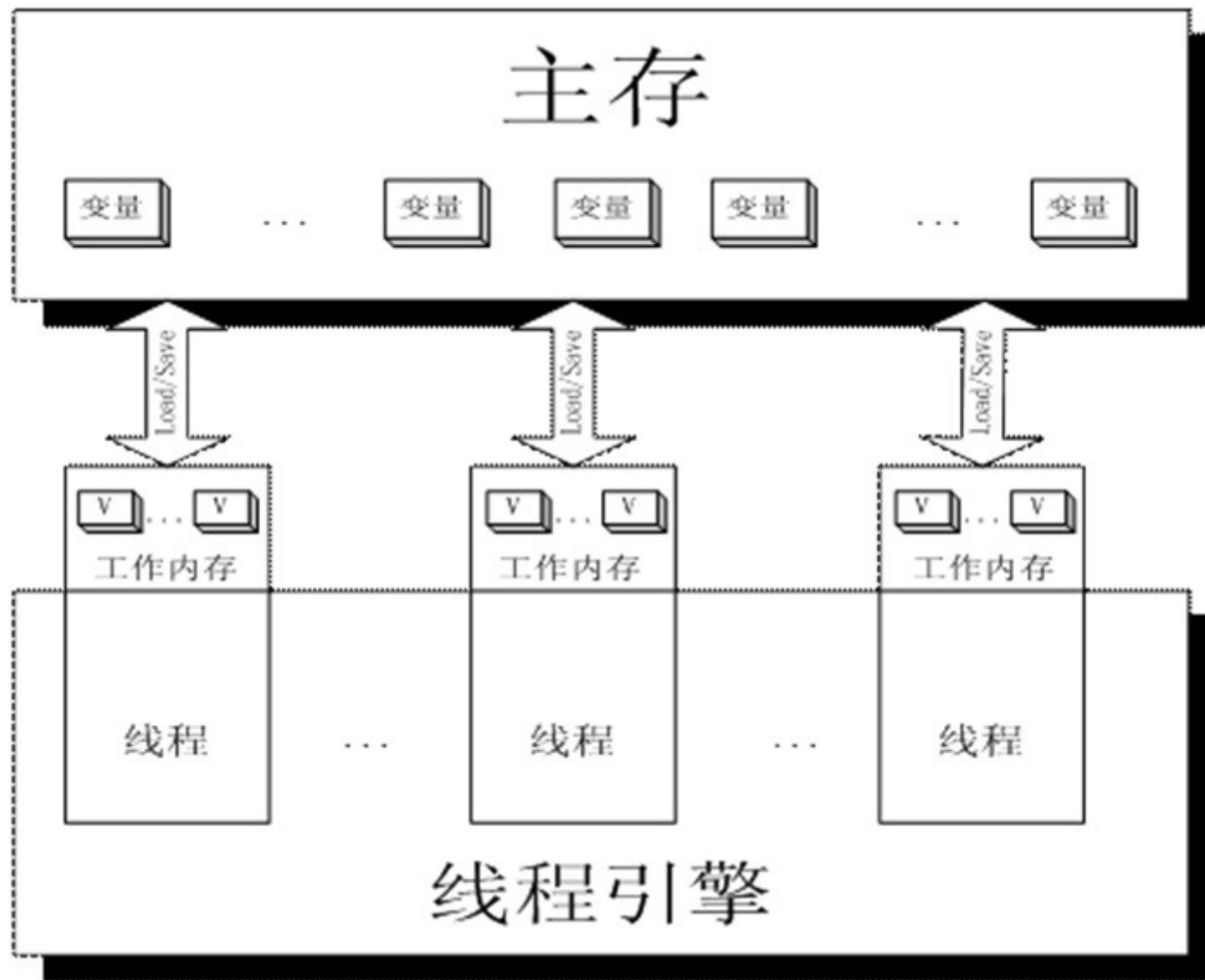
厉害

--xiao朱

2. Re:JVM 内存初学 (堆(heap)、栈(stack)和方法区(method))

厉害厉害, 我转载了哈~

--梦想, 遥不可及



Java

中通过多线程机制使得多个任务同时执行处理，所有的线程共享JVM内存区域main memory，而每个线程又单独的有自己的工作内存，当线程与内存区域进行交互时，数据从主存拷贝到工作内存，进而交由线程处理（操作码+操作数）。更多信息我们会在后面的《深入JVM—JVM类执行机制中详细解说》。

3. Re:JVM 内存初学 (堆(heap)、栈(stack)和方法区(method))

好文,我爱你

--范特斯刹可

4. Re:深入理解JVM—JVM内存模型

顶

--做个有梦想的咸鱼

5. Re:JVM 内存初学 (堆(heap)、栈(stack)和方法区(method))

很棒

--清风扰民、

阅读排行榜

1. 深入理解JVM—JVM内存模型(191620)
2. CentOS修改系统时间(43327)
3. Linux 与 Linux Windows 文件共享(29966)
4. JQuery Div scrollTop ScrollHeight(28663)
5. 如何快速的解决Maven依赖冲突(20716)

评论排行榜

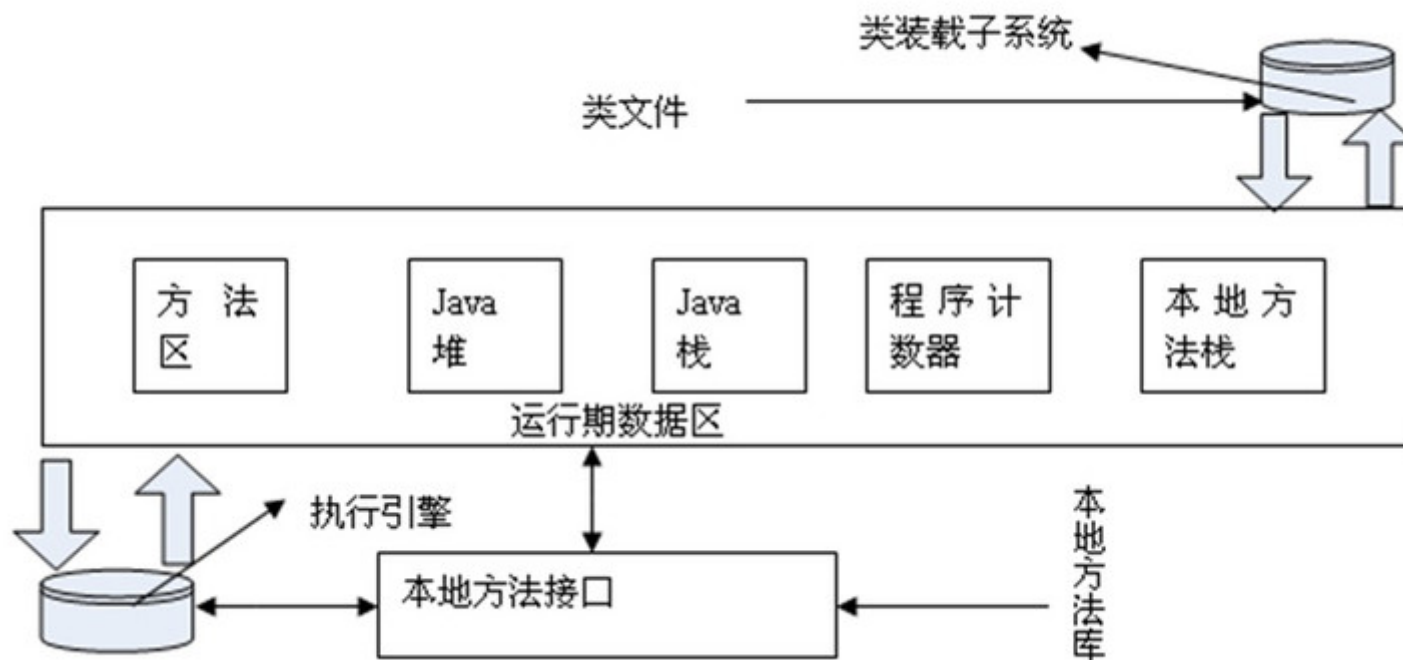
1. 深入理解JVM—JVM内存模型(21)
2. JVM 内存初学 (堆(heap)、栈(stack)和方法区(method))(8)
3. JQuery Div scrollTop ScrollHeight(6)
4. div 移动(2)
5. Maven 映像(2)

推荐排行榜

1. 深入理解JVM—JVM内存模型(19)
2. JVM 内存初学 (堆(heap)、栈(stack)和方法区(method))(10)

在之前，我们也已经提到，JVM的逻辑内存模型如下：

- 3. Linux 与 Linux Windows 文件共享(4)
- 4. JQuery Div scrollTop ScrollHeight (2)
- 5. Spring AOP AspectJ(2)



我们现在来逐个的看下每个到底是做什么的！

1、程序计数器

程序计数器（Program Counter Register）是一块较小的内存空间，它的作用可以看做是当前线程所执行的字节码的行号指示器。在虚拟机的概念模型里（仅是概念模型，各种虚拟机可能会通过一些更高效的方式去实现），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，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并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）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，各条线程之间的计数器互不影响，独立存储，我们称这类内存区域为“线程私有”的内存。

如果线程正在执行的是一个Java 方法，这个计数器记录的是正在执行的虚拟机字节码指令的地址；如果正在执行的是Native 方法，这个计数器值则为空（Undefined）。此内存区域是唯一一个在Java 虚拟机规范中没有规定任何OutOfMemoryError 情况的区域。

2、Java 虚拟机栈

与程序计数器一样，Java 虚拟机栈（Java Virtual Machine Stacks）也是线程私有的，它的生命周期与线程相同。虚拟机栈描述的是Java 方法执行的内存模型：每个方法被执行的时候都会同时创建一个栈帧（Stack Frame ①）用于存储局部变量表、操作栈、动态链接、方法出口等信息。每一个方法被调用直至执行完成的过程，就对应着一个栈帧在虚拟机栈中从入栈到出栈的过程。

经常有人把Java 内存区分为堆内存（Heap）和栈内存（Stack），这种分法比较粗糙，Java 内存区域的划分实际上远比这复杂。这种划分方式的流行只能说明大多数程序员最关注的、与对象内存分配关系最密切的内存区域是这两块。其中所指的“堆”在后面会专门讲述，而所指的“栈”就是现在讲的虚拟机栈，或者说是虚拟机栈中的局部变量表部分。

局部变量表存放了编译期可知的各种基本数据类型（boolean、byte、char、short、int、float、long、double）、对象引用（reference 类型，它不等同于对象本身，根据不同的虚拟机实现，它可能是一个指向对象起始地址的引用指针，也可能指向一个代表对象的句柄或者其他与此对象相关的位置）和returnAddress 类型（指向了一条字节码指令的地址）。

其中64 位长度的long 和double 类型的数据会占用2 个局部变量空间（Slot），其余的数据类型只占用1 个。局部变量表所需的内存空间在编译期间完成分配，当进入一个方法时，这个方法需要在帧中分配多大的局部变量空间是完全确定的，在方法运行期间

不会改变局部变量表的大小。

在Java 虚拟机规范中，对这个区域规定了两种异常状况：如果线程请求的栈深度大于虚拟机所允许的深度，将抛出StackOverflowError 异常；如果虚拟机栈可以动态扩展（当前大部分的Java 虚拟机都可动态扩展，只不过Java 虚拟机规范中也允许固定长度的虚拟机栈），当扩展时无法申请到足够的内存时会抛出OutOfMemoryError 异常。

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，其区别不过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Native 方法服务。虚拟机规范中对本地方法栈中的方法使用的语言、使用方式与数据结构并没有强制规定，因此具体的虚拟机可以自由实现它。甚至有的虚拟机（譬如Sun HotSpot 虚拟机）直接就把本地方法栈和虚拟机栈合二为一。与虚拟机栈一样，本地方法栈区域也会抛出StackOverflowError 和OutOfMemoryError 异常。

4、Java 堆

对于大多数应用来说，Java 堆（Java Heap）是Java 虚拟机所管理的内存中最大的一块。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，几乎所有的对象实例都在这里分配内存。这一点在Java 虚拟机规范中的描述是：所有的对象实例以及数组都要在堆上分配①，但是随着JIT 编译器的发展与逃逸分析技术的逐渐成熟，栈上分配、标量替换②优化技术将会导致一些微妙的变化发生，所有的对象都分配在堆上也渐渐变得不是那么“绝对”了。

Java 堆是垃圾收集器管理的主要区域，因此很多时候也被称做“GC 堆”（Garbage Collected Heap，幸好国内没翻译成“垃圾堆”）。如果从内存回收的角度看，由于现在收集器基本都是采用的分代收集算法，所以Java 堆中还可以细分为：新生代和老年代；再细致一点的有Eden 空间、From Survivor 空间、To Survivor 空间等。如果从内存分配的角度看，线程共享的Java 堆中可能划分出多个线程私有的分配缓冲区（Thread Local

Allocation Buffer, TLAB)。不过, 无论如何划分, 都与存放内容无关, 无论哪个区域, 存储的都仍然是对象实例, 进一步划分的目的是为了更好地回收内存, 或者更快地分配内存。在本章中, 我们仅仅针对内存区域的作用进行讨论, Java 堆中的上述各个区域的分配和回收等细节将会是下一章的主题。

根据Java 虚拟机规范的规定, Java 堆可以处于物理上不连续的内存空间中, 只要逻辑上是连续的即可, 就像我们的磁盘空间一样。在实现时, 既可以实现成固定大小的, 也可以是可扩展的, 不过当前主流的虚拟机都是按照可扩展来实现的(通过-Xmx和-Xms 控制)。如果在堆中没有内存完成实例分配, 并且堆也无法再扩展时, 将会抛出OutOfMemoryError 异常。

4、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等数据。虽然Java 虚拟机规范把方法区描述为堆的一个逻辑部分, 但是它却有一个别名叫做Non-Heap (非堆), 目的应该是与Java 堆区分开来。

对于习惯在HotSpot 虚拟机上开发和部署程序的开发者来说, 很多人愿意把方法区称为“永久代”(Permanent Generation), 本质上两者并不等价, 仅仅是因为HotSpot 虚拟机的设计团队选择把GC 分代收集扩展至方法区, 或者说使用永久代来实现方法区而已。对于其他虚拟机(如BEA JRockit、IBM J9 等)来说是不存在永久代的概念的。即使是HotSpot 虚拟机本身, 根据官方发布的路线图信息, 现在也有放弃永久代并“搬家”至Native Memory 来实现方法区的规划了。

Java 虚拟机规范对这个区域的限制非常宽松, 除了和Java 堆一样不需要连续的内存和可以选择固定大小或者可扩展外, 还可以选择不实现垃圾收集。相对而言, 垃圾收集行为在这个区域是比较少出现的, 但并非数据进入了方法区就如永久代的名称一样“永久”存在了。这个区域的内存回收目标主要是针对常量池的回收和对类型的卸载, 一般来说这个区域的回收“成绩”比较难以令人满意, 尤其是类型的卸载, 条件

相当苛刻，但是这部分区域的回收确实是有必要的。在Sun 公司的BUG 列表中，曾出现过若干个严重的BUG 就是由于低版本的HotSpot 虚拟机对此区域未完全回收而致内存泄漏。

根据Java 虚拟机规范的规定，当方法区无法满足内存分配需求时，将抛出OutOfMemoryError 异常。

5、运行时常量池

运行时常量池（Runtime Constant Pool）是方法区的一部分。Class 文件中除了有类的版本、字段、方法、接口等描述等信息外，还有一项信息是常量池（Constant Pool Table），用于存放编译期生成的各种字面量和符号引用，这部分内容将在类加载后存放方法区的运行时常量池中。

Java 虚拟机对Class 文件的每一部分（自然也包括常量池）的格式都有严格的规定，每一个字节用于存储哪种数据都必须符合规范上的要求，这样才会被虚拟机认可、装载和执行。但对于运行时常量池，Java 虚拟机规范没有做任何细节的要求，不同的提供商实现的虚拟机可以按照自己的需要来实现这个内存区域。不过，一般来说，除了保存Class 文件中描述的符号引用外，还会把翻译出来的直接引用也存储在运行时常量池中①。

运行时常量池相对于Class 文件常量池的另外一个重要特征是具备动态性，Java 语言并不要求常量一定只能在编译期产生，也就是并非预置入Class 文件中常量池的内容才能进入方法区运行时常量池，运行期间也可能将新的常量放入池中，这种特性被开发人员利用得比较多的便是String 类的intern() 方法。

既然运行时常量池是方法区的一部分，自然会受到方法区内存的限制，当常量池无法再申请到内存时会抛出OutOfMemoryError 异常

6、直接内存

直接内存（Direct Memory）并不是虚拟机运行时数据区的一部分，也不是Java 虚拟机规范中定义的内存区域，但是这部分内存也被频繁地使用，而且也可能导致

OutOfMemoryError 异常出现，所以我们放到这里一起讲解。

在JDK 1.4 中新加入了NIO（New Input/Output）类，引入了一种基于通道（Channel）与缓冲区（Buffer）的I/O 方式，它可以使用Native 函数库直接分配堆外内存，然后通过一个存储在Java 堆里面的DirectByteBuffer 对象作为这块内存的引用进行操作。这样能在一些场景中显著提高性能，因为避免了在Java 堆和Native 堆中来回复制数据。

显然，本机直接内存的分配不会受到Java 堆大小的限制，但是，既然是内存，则肯定还是会受到本机总内存（包括RAM 及SWAP 区或者分页文件）的大小及处理器寻址空间的限制。服务器管理员配置虚拟机参数时，一般会根据实际内存设置-Xmx 等参数信息，但经常会忽略掉直接内存，使得各个内存区域的总和大于物理内存限制（包括物理上的和操作系统级的限制），从而导致动态扩展时出现OutOfMemoryError 异常。

逻辑内存模型我们已经看到了，那当我们建立一个对象的时候是怎么进行访问的呢？

在Java 语言中，对象访问是如何进行的？对象访问在Java 语言中无处不在，是最普通的程序行为，但即使是最简单的访问，也会却涉及Java 栈、Java 堆、方法区这三个最重要内存区

域之间的关联关系，如下面的这句代码：

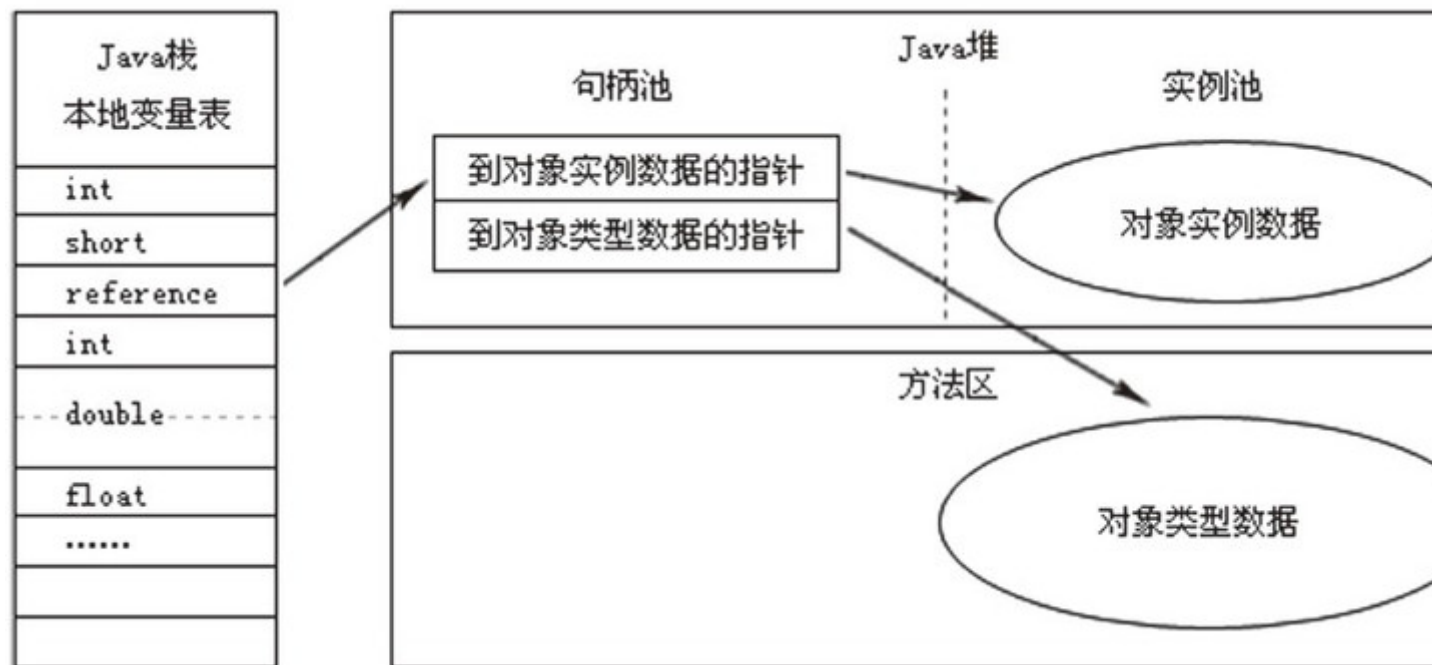
```
Object obj = new Object();
```

假设这句代码出现在方法体中，那“Object obj”这部分的语义将会反映到Java 栈的本地变量表中，作为一个reference 类型数据出现。而“new Object()”这部分的语义将会反映到Java 堆中，形成一块存储了Object 类型所有实例数据值（Instance Data，对象中各个实例字段的数据）的结构化内存，根据具体类型以及虚拟机实现的对象内存布局（Object Memory Layout）的不同，这块内存的长度是不固定的。另外，在Java 堆中还必须包含能查找到此对象类型数据（如对象类型、父类、实现的接口、方法等）的地址信息，这些类型数据则存储在方法区中。

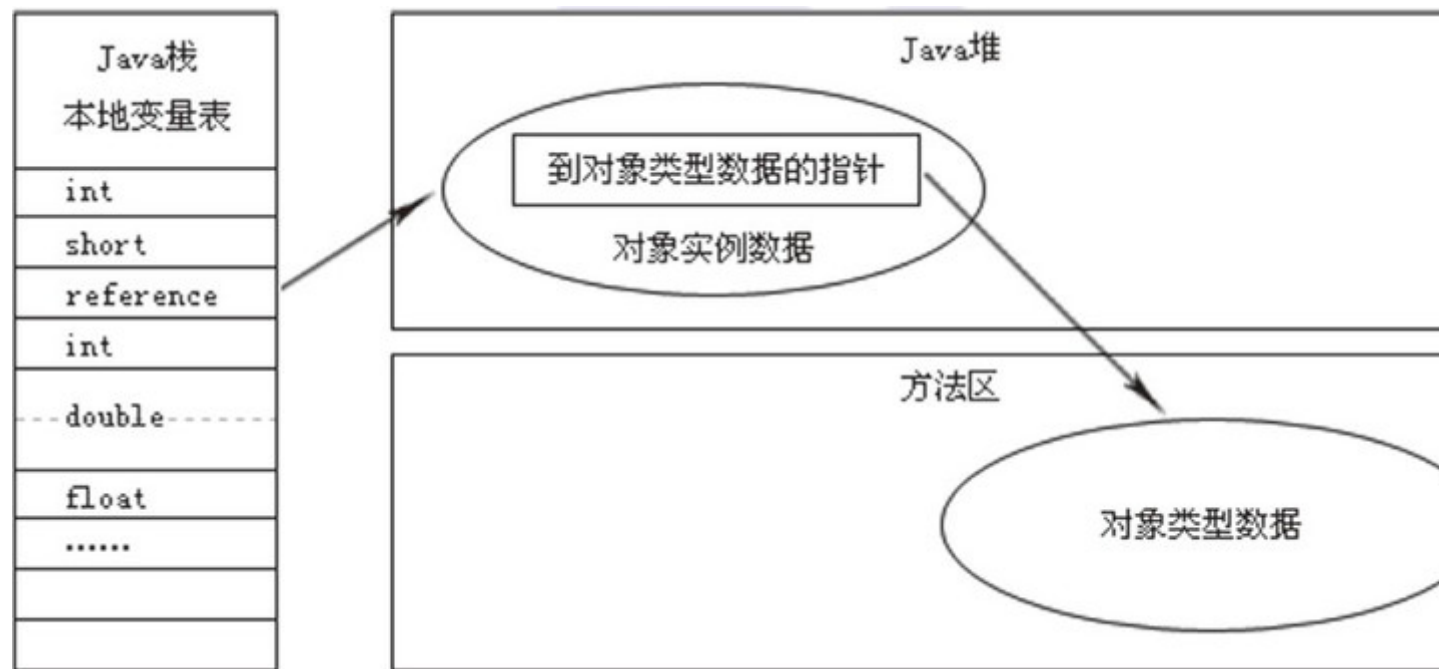
由于reference 类型在Java 虚拟机规范里面只规定了一个指向对象的引用，并没有

定义这个引用应该通过哪种方式去定位，以及访问到Java 堆中的对象的具体位置，因此不同虚拟机实现的对象访问方式会有所不同，主流的访问方式有两种：使用句柄和直接指针。

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



如果使用直接指针访问方式，Java 堆对象的布局中就必须考虑如何放置访问类型数据的相关信息，reference 中直接存储的就是对象地址，如下图所示



这两种对象的访问方式各有优势，使用句柄访问方式的最大好处就是

reference 中存

储的是稳定的句柄地址，在对象被移动（垃圾收集时移动对象是非常普遍的行为）时只

会改变句柄中的实例数据指针，而reference 本身不需要被修改。

使用直接指针访问方式的最大好处就是速度更快，它节省了一次指针定位的时间开

销，由于对象的访问在Java 中非常频繁，因此这类开销积少成多后也是一项非常可观的

执行成本。就本书讨论的主要虚拟机Sun HotSpot 而言，它是使用第二种方式进行对象访问的，但从整个软件开发的范围来看，各种语言和框架使用句柄来访问的情况也十分常见。

下面我们来看几个示例

1、Java 堆溢出

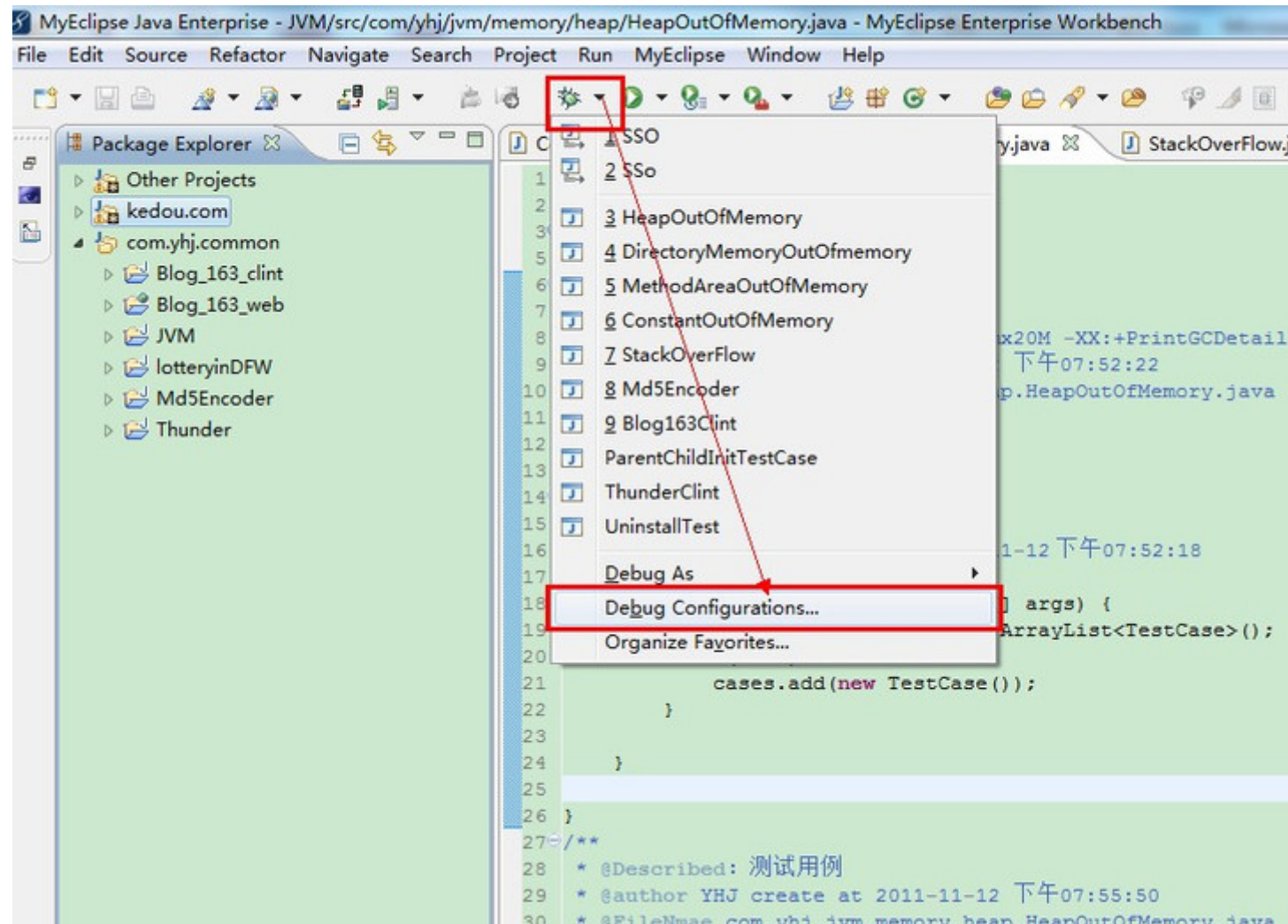
下面的程中我们限制Java 堆的大小为20MB，不可扩展（将堆的最小值-Xms 参

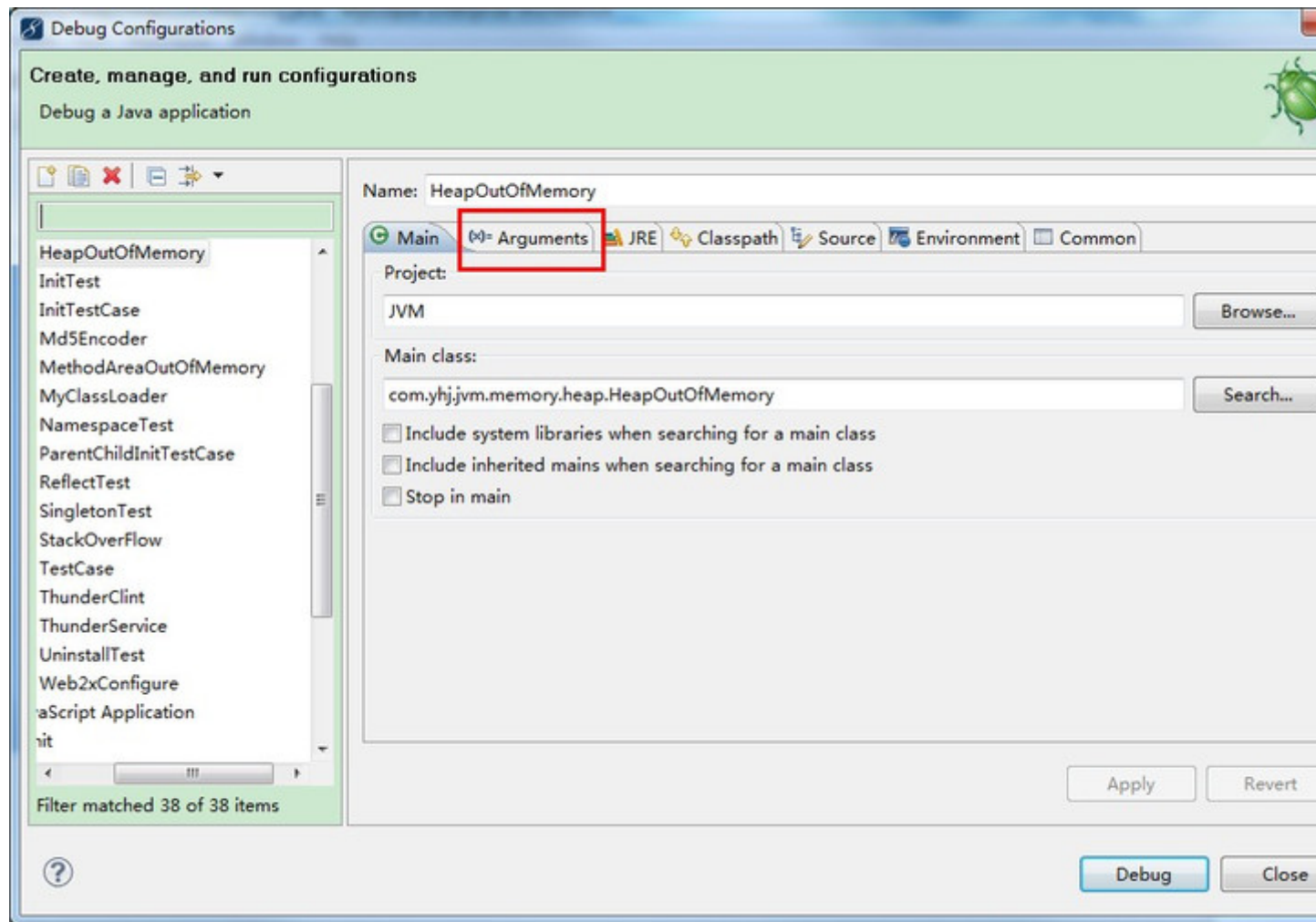
数与最大值-Xmx 参数设置为一样即可避免堆自动扩展），通过参数-XX:+HeapDump

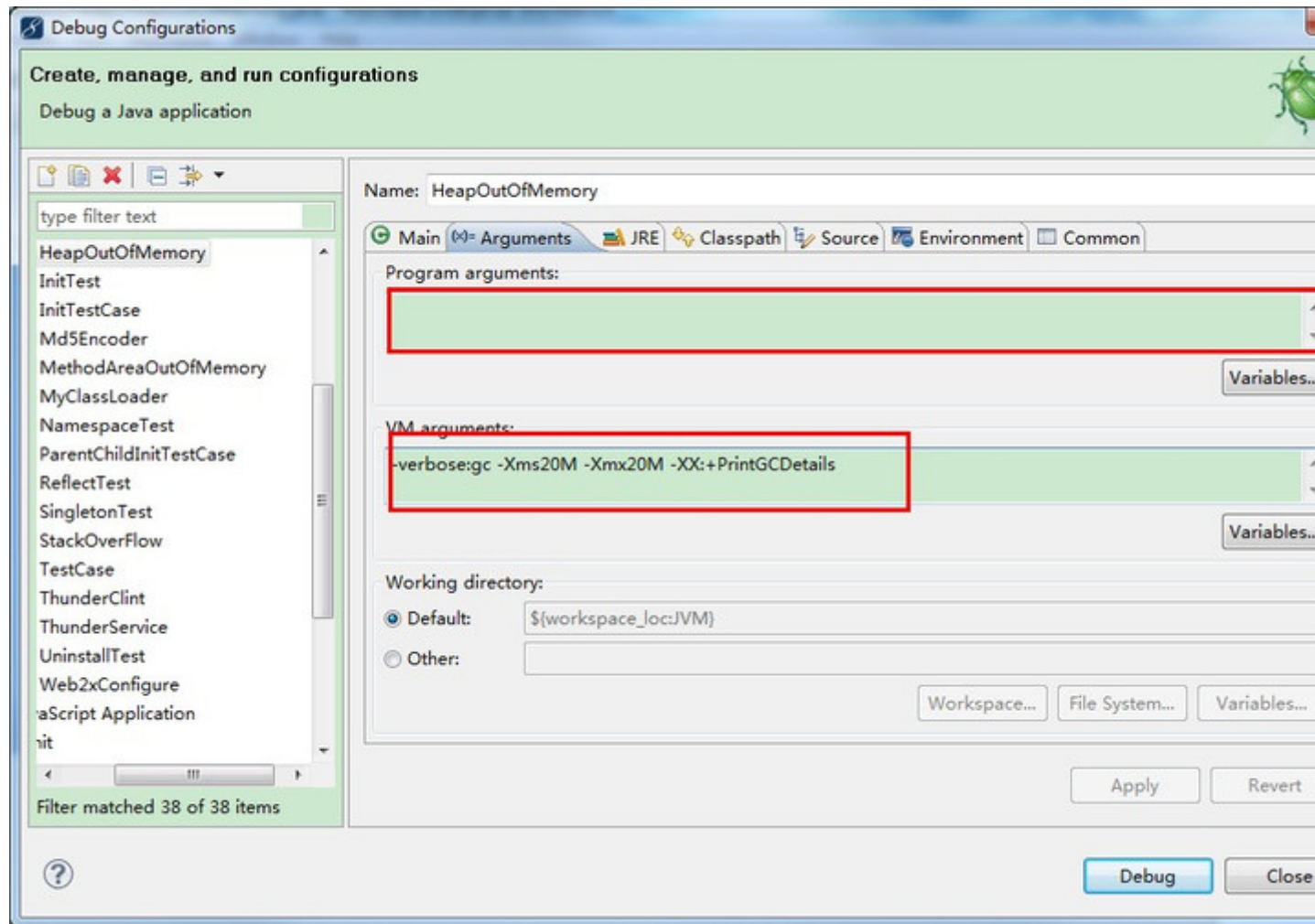
OnOutOfMemoryError 可以让虚拟机在出现内存溢出异常时Dump 出当前的内存堆转储

快照以便事后进行分析。

参数设置如下







```
package com.yhj.jvm.memory.heap;
```

```
import java.util.ArrayList;
```

```
import java.util.List;
```

```
/**
```

```
* @Described: 堆溢出测试
* @VM args:-verbose:gc -Xms20M -Xmx20M -XX:+PrintGCDetails
* @author YHJ create at 2011-11-12 下午07:52:22
* @FileNmae com.yhj.jvm.memory.heap.HeapOutOfMemory.java
*/

public class HeapOutOfMemory {

    /**
     * @param args
     * @Author YHJ create at 2011-11-12 下午07:52:18
     */
    public static void main(String[] args) {
        List<TestCase> cases = new ArrayList<TestCase>();
        while(true){
            cases.add(new TestCase());
        }

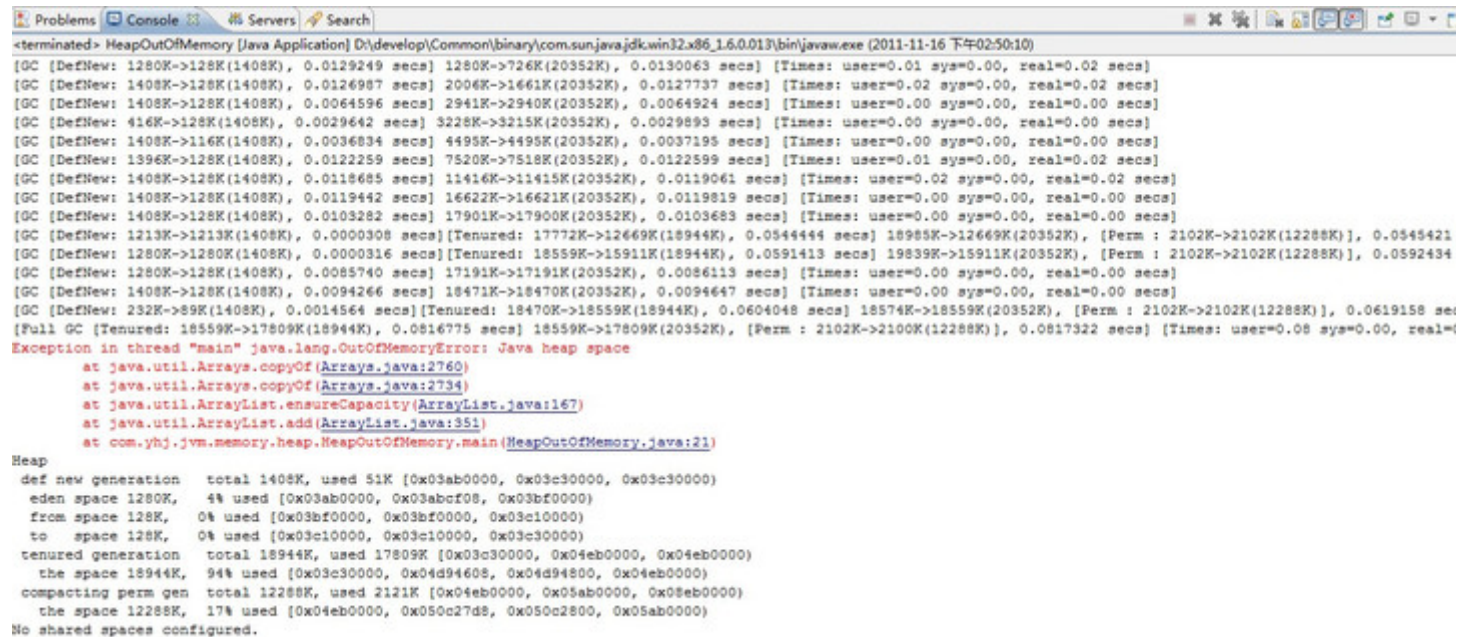
    }

}

/**
* @Described: 测试用例
* @author YHJ create at 2011-11-12 下午07:55:50
* @FileNmae com.yhj.jvm.memory.heap.HeapOutOfMemory.java
*/
```

```
class TestCase{
```

```
}
```



```

<terminated> HeapOutOfMemory [Java Application] D:\develop\Common\binary\com.sun.java.jdk.win32.x86_1.6.0.013\bin\javaw.exe (2011-11-16 下午02:50:10)
[GC [DefNew: 1280K->128K(1408K), 0.0129249 secs] 1280K->726K(20352K), 0.0130063 secs] [Times: user=0.01 sys=0.00, real=0.02 secs]
[GC [DefNew: 1408K->128K(1408K), 0.0126987 secs] 2006K->1661K(20352K), 0.0127737 secs] [Times: user=0.02 sys=0.00, real=0.02 secs]
[GC [DefNew: 1408K->128K(1408K), 0.0064596 secs] 2941K->2940K(20352K), 0.0064924 secs] [Times: user=0.00 sys=0.00, real=0.00 secs]
[GC [DefNew: 416K->128K(1408K), 0.0029642 secs] 3228K->3215K(20352K), 0.0029893 secs] [Times: user=0.00 sys=0.00, real=0.00 secs]
[GC [DefNew: 1408K->116K(1408K), 0.0036834 secs] 4495K->4495K(20352K), 0.0037195 secs] [Times: user=0.00 sys=0.00, real=0.00 secs]
[GC [DefNew: 1396K->128K(1408K), 0.0122259 secs] 7520K->7518K(20352K), 0.0122599 secs] [Times: user=0.01 sys=0.00, real=0.02 secs]
[GC [DefNew: 1408K->128K(1408K), 0.0118685 secs] 11416K->11415K(20352K), 0.0119061 secs] [Times: user=0.02 sys=0.00, real=0.02 secs]
[GC [DefNew: 1408K->128K(1408K), 0.0119442 secs] 16622K->16621K(20352K), 0.0119819 secs] [Times: user=0.00 sys=0.00, real=0.00 secs]
[GC [DefNew: 1408K->128K(1408K), 0.0103282 secs] 17901K->17900K(20352K), 0.0103683 secs] [Times: user=0.00 sys=0.00, real=0.00 secs]
[GC [DefNew: 1213K->1213K(1408K), 0.0000308 secs][Tenured: 17772K->12669K(18944K), 0.0544444 secs] 18985K->12669K(20352K), [Perm : 2102K->2102K(12288K)], 0.0545421
[GC [DefNew: 1280K->1280K(1408K), 0.0000316 secs][Tenured: 18559K->15911K(18944K), 0.0591413 secs] 19839K->15911K(20352K), [Perm : 2102K->2102K(12288K)], 0.0592434
[GC [DefNew: 1280K->128K(1408K), 0.0085740 secs] 17191K->17191K(20352K), 0.0086113 secs] [Times: user=0.00 sys=0.00, real=0.00 secs]
[GC [DefNew: 1408K->128K(1408K), 0.0094266 secs] 18471K->18470K(20352K), 0.0094647 secs] [Times: user=0.00 sys=0.00, real=0.00 secs]
[GC [DefNew: 232K->89K(1408K), 0.0014564 secs][Tenured: 18470K->18559K(18944K), 0.0604048 secs] 18574K->18559K(20352K), [Perm : 2102K->2102K(12288K)], 0.0619158 se
[Full GC [Tenured: 18559K->17809K(18944K), 0.0816775 secs] 18559K->17809K(20352K), [Perm : 2102K->2100K(12288K)], 0.0817322 secs] [Times: user=0.08 sys=0.00, real=
Exception in thread "main" java.lang.OutOfMemoryError: Java heap space
    at java.util.Arrays.copyOf(Arrays.java:2760)
    at java.util.Arrays.copyOf(Arrays.java:2734)
    at java.util.ArrayList.ensureCapacity(ArrayList.java:167)
    at java.util.ArrayList.add(ArrayList.java:351)
    at com.yhj.jvm.memory.heap.HeapOutOfMemory.main(HeapOutOfMemory.java:21)

Heap
def new generation      total 1408K, used 51K [0x03ab0000, 0x03c30000, 0x03c30000)
eden space 1280K,      4% used [0x03ab0000, 0x03abcf08, 0x03bf0000)
from space 128K,       0% used [0x03bf0000, 0x03bf0000, 0x03c10000)
to space 128K,         0% used [0x03c10000, 0x03c10000, 0x03c30000)
tenured generation      total 18944K, used 17809K [0x03c30000, 0x04eb0000, 0x04eb0000)
the space 18944K,       94% used [0x03c30000, 0x04d94608, 0x04d94800, 0x04eb0000)
compacting perm gen      total 12288K, used 2121K [0x04eb0000, 0x05ab0000, 0x05ab0000)
the space 12288K,      17% used [0x04eb0000, 0x050c27d8, 0x050c2800, 0x05ab0000)
No shared spaces configured.

```

Java 堆内存的OutOfMemoryError异常是实际应用中最常见的内存溢出异常情况。出现Java 堆内

存溢出时，异常堆栈信息“java.lang.OutOfMemoryError”会跟着进一步提示“Java heap

space”。

要解决这个区域的异常，一般的手段是首先通过内存映像分析工具（如Eclipse

Memory Analyzer）对dump 出来的堆转储快照进行分析，重点是确认内存中的对象是

否是必要的，也就是要先弄清楚到底是出现了内存泄漏（Memory Leak）还是内存溢

出（Memory Overflow）。图2-5 显示了使用Eclipse Memory Analyzer 打开的堆转储快

照文件。

如果是内存泄漏，可进一步通过工具查看泄漏对象到GC Roots 的引用链。于是就

能找到泄漏对象是通过怎样的路径与GC Roots 相关联并导致垃圾收集器无法自动回收它们的。掌握了泄漏对象的类型信息，以及GC Roots 引用链的信息，就可以比较准确地定位出泄漏代码的位置。

如果不存在泄漏，换句话说就是内存中的对象确实都必须存活，那就应当检查虚拟机的堆参数（-Xmx 与-Xms），与机器物理内存对比看是否还可以调大，从代码上检查是否存在某些对象生命周期过长、持有状态时间过长的情况，尝试减少程序运行期的内存消耗。

以上是处理Java 堆内存问题的简略思路，处理这些问题所需要的知识、工具与经验在后面的几次分享中我会做一些额外的分析。

2、java栈溢出

```
package com.yhj.jvm.memory.stack;

/**
 * @Described: 栈层级不足探究
 * @VM args:-Xss128k
 * @author YHJ create at 2011-11-12 下午08:19:28
 * @FileNmae com.yhj.jvm.memory.stack.StackOverFlow.java
 */

public class StackOverFlow {

    private int i ;

    public void plus() {

        i++;

        plus();
    }
}
```

```
}

/**
 * @param args
 * @Author YHJ create at 2011-11-12 下午08:19:21
 */
public static void main(String[] args) {
    StackOverFlow stackOverFlow = new StackOverFlow();
    try {
        stackOverFlow.plus();
    } catch (Exception e) {
        System.out.println("Exception:stack length:"+stackOverFlow.i);
        e.printStackTrace();
    } catch (Error e) {
        System.out.println("Error:stack length:"+stackOverFlow.i);
        e.printStackTrace();
    }
}

}
```

3、常量池溢出（常量池都有哪些信息，我们在后续的JVM类文件结构中详细描述）

```
package com.yhj.jvm.memory.constant;
```



```
import java.util.ArrayList;

import java.util.List;

/**
 * @Described: 常量池内存溢出探究
 * @VM args : -XX:PermSize=10M -XX:MaxPermSize=10M
 * @author YHJ create at 2011-10-30 下午04:28:30
 * @FileNmae com.yhj.jvm.memory.constant.ConstantOutOfMemory.java
 */
public class ConstantOutOfMemory {

    /**
     * @param args
     * @throws Exception
     * @Author YHJ create at 2011-10-30 下午04:28:25
     */
    public static void main(String[] args) throws Exception {
        try {
            List<String> strings = new ArrayList<String>();
            int i = 0;
            while(true){
                strings.add(String.valueOf(i++).intern());
            }
        } catch (Exception e) {
            e.printStackTrace();
        }
    }
}
```

```
        throw e;
    }

}
```

```
}
```

4、方法去溢出

```
package com.yhj.jvm.memory.methodArea;
```

```
import java.lang.reflect.Method;
```

```
import net.sf.cglib.proxy.Enhancer;
```

```
import net.sf.cglib.proxy.MethodInterceptor;
```

```
import net.sf.cglib.proxy.MethodProxy;
```

```
/**
```

```
 * @Described: 方法区溢出测试
```

```
 * 使用技术 CBlib
```

```
 * @VM args : -XX:PermSize=10M -XX:MaxPermSize=10M
```

```
 * @author YHJ create at 2011-11-12 下午08:47:55
```

```
 * @FileNmae com.yhj.jvm.memory.methodArea.MethodAreaOutOfMemory.java
```

```
 */
```

```
public class MethodAreaOutOfMemory {
```

```
/**
```

```
* @param args
* @Author YHJ create at 2011-11-12 下午08:47:51
*/

public static void main(String[] args) {
    while(true){
        Enhancer enhancer = new Enhancer();
        enhancer.setSuperclass(TestCase.class);
        enhancer.setUseCache(false);
        enhancer.setCallback(new MethodInterceptor() {
            @Override
            public Object intercept(Object arg0, Method arg1, Object[] arg2,
                MethodProxy arg3) throws Throwable {
                return arg3.invokeSuper(arg0, arg2);
            }
        });
        enhancer.create();
    }
}

}

}

/**
* @Described: 测试用例
* @author YHJ create at 2011-11-12 下午08:53:09
* @FileNmae com.yhj.jvm.memory.methodArea.MethodAreaOutOfMemory.java
*/
```

```
class TestCase{
```

```
}
```

5、直接内存溢出

```
package com.yhj.jvm.memory.directoryMemory;
```

```
import java.lang.reflect.Field;
```

```
import sun.misc.Unsafe;
```

```
/**
```

```
 * @Described: 直接内存溢出测试
```

```
 * @VM args: -Xmx20M -XX:MaxDirectMemorySize=10M
```

```
 * @author YHJ create at 2011-11-12 下午09:06:10
```

```
 * @FileNmae com.yhj.jvm.memory.directoryMemory.DirectoryMemoryOutOfmemory.java
```

```
 */
```

```
public class DirectoryMemoryOutOfmemory {
```

```
    private static final int ONE_MB = 1024*1024;
```

```
    private static int count = 1;
```

```
/**
```

```
 * @param args
```

```
 * @Author YHJ create at 2011-11-12 下午09:05:54
```

```
 */
```

```
public static void main(String[] args) {  
    try {  
        Field field = Unsafe.class.getDeclaredField("theUnsafe");  
        field.setAccessible(true);  
        Unsafe unsafe = (Unsafe) field.get(null);  
        while (true) {  
            unsafe.allocateMemory(ONE_MB);  
            count++;  
        }  
    } catch (Exception e) {  
        System.out.println("Exception:instance created "+count);  
        e.printStackTrace();  
    } catch (Error e) {  
        System.out.println("Error:instance created "+count);  
        e.printStackTrace();  
    }  
}
```

分类: [JVM](#)

好文要顶

关注我

收藏该文



丁应思

关注 - 0

粉丝 - 169

[+加关注](#)

19

1

« 上一篇: [Servlet多文件上传方法](#)» 下一篇: [JVM 内存初学 \(堆\(heap\)、栈\(stack\)和方法区\(method\)_\)](#)posted @ 2014-05-30 08:58 丁应思 阅读(191623) 评论(21) [编辑](#) [收藏](#)

评论

#1楼 2015-06-17 22:45 | daryerpang

高手啊

支持(1) 反对(0)

#2楼 2015-08-31 19:05 | He!!

谢谢作者分享，拜读了~~~

支持(1) 反对(0)

#3楼 2015-09-04 10:56 | buxiaoxia

没点自己的东西。。。全是抄袭 《深入理解java虚拟机》

支持(11) 反对(10)

#4楼 2015-10-26 18:10 | 心扉

NIO 是 New Input/Output的意思?

应该是Non-Blocking IO的意思，中文是非阻塞式IO。

[支持\(1\)](#) [反对\(0\)](#)

#5楼 2015-12-24 16:05 | littledreamtown

nio 是New IO 的简称, 在jdk1.4 里提供的新api 。Sun 官方标榜的特性如下: 为所有的原始类型提供(Buffer)缓存支持。字符集编码解码解决方案。Channel : 一个新的原始I/O 抽象。支持锁和内存映射文件的文件访问接口。提供多路(non-bloking) 非阻塞式的高伸缩性网络I/O 。

[支持\(1\)](#) [反对\(0\)](#)

#6楼 2016-04-19 11:22 | zhoujianboy

推荐一篇文章 JVM内存模型和JVM参数的关系: <http://www.haonanj.cn/408.html>

[支持\(1\)](#) [反对\(0\)](#)

#7楼 2016-08-26 12:15 | 在自己的空间里打天地

不错的分享!

[支持\(0\)](#) [反对\(0\)](#)

#8楼 2016-10-10 14:55 | 喝咖啡的考拉

@ buxiaoxia
的确是, 只字不落

[支持\(0\)](#) [反对\(1\)](#)

#9楼 2017-01-09 11:25 | dota小夜曲

@ 喝咖啡的考拉@ buxiaoxia
我觉得楼主做得很好, 通过博文可以直接快速的去获取我想要理解的知识, 楼主写这篇博文应该不是为了给你们看的吧。

[支持\(4\)](#) [反对\(0\)](#)

#10楼 2017-02-25 20:01 | spencerLiu

+1 不管是摘录还是总结, 受益匪浅~谢谢博主

支持(3) 反对(0)

#11楼 2017-04-11 16:45 | AlanCoder

JVM	方法区	运行常量池	堆	栈	程序计数器	本地方法栈
线程共享	是	是	是	否	否	否
作用	类信息、静态方法、常量、JIT代码	方法区一部分, JIT后常量	对象实例	基本类型	行号、内存地址	本地方法、服务
异常	内存溢出	内存溢出	内存溢出	内存溢出、stackoverflow	无	内存溢出、stackoverflow

这样理解对么？

支持(0) 反对(0)

#12楼 2017-04-11 20:50 | 落叶已归根

学习了

支持(0) 反对(0)

#13楼 2017-07-24 11:54 | famary

@ buxiaoxia

确实是《深入理解Java虚拟机》的东西，但是我相信楼主还有有自己的见解的，毕竟不是谁都有时间和机会去看书的。

支持(7) 反对(0)

#14楼 2017-07-30 12:39 | 花开易见落难寻

可以转载吗

支持(0) 反对(0)

#15楼 2017-08-25 09:39 | starskyhu

@ buxiaoxia

自己去记录分享一些好东西 大家共享 支持 厌烦你这种人

支持(2) 反对(1)

#16楼 2017-09-26 15:17 | 诸葛印

@ spencerLiu

你花钱买原作者的版权了么?

支持(0) 反对(1)

#17楼 2017-09-27 11:16 | ThisIsTest

。

支持(0) 反对(0)

#18楼 2017-10-12 12:50 | duoduoCain

@ 心扉

都可以，没什么毛病

支持(0) 反对(0)

#19楼 2017-10-28 11:06 | guagua2222

有收获是最重要的，还是O(∩_∩)O谢谢

支持(0) 反对(0)

#20楼 2017-11-30 15:07 | 怀瑾握瑜XI

栈帧存储的应该是操作数，局部变量……等，并不是操作栈吧。笔者笔误吧？

支持(0) 反对(0)

#21楼 2017-12-11 16:10 | 做个有梦想的咸鱼

顶

支持(0) 反对(0)

[刷新评论](#) [刷新页面](#) [返回顶部](#)

注册用户登录后才能发表评论，请 [登录](#) 或 [注册](#)，[访问网站首页](#)。



最新IT新闻:

- 数据泄漏多是内鬼所为: 高净值信息售价惊人、涌入电信诈骗市场
 - iOS 12正式版来了, 升级后先来试试这13个新功能
 - 极验否认“碰瓷”: 网易云盾方仍没有主动跟我们进行沟通
 - SpaceX揭秘绕月飞行首单旅客: 日本亿万富翁前泽友作
 - 效仿当年“淘宝孵天猫”的旧路, 流量巨兽拼多多谋求转型
- » [更多新闻...](#)



最新知识库文章:

- 为什么说 Java 程序员必须掌握 Spring Boot ?
 - 在学习中, 有一个比掌握知识更重要的能力
 - 如何招到一个靠谱的程序员
 - 一个故事看懂“区块链”
 - 被踢出去的用户
- » [更多知识库文章...](#)