
看笔记前须知

为了区分不同的知识点的重要级别。我会在笔记中使用加粗、颜色或者加()标识的方式强调。

比如：

- 1、**红色字体**，代表需要注意点，特别强调的知识点。
- 2、（非重点），代表该知识只需大致了解即可。
- 3、**加粗的字段**，一般代表总结的语言，需要注意。

JVM 基础知识

JVM 从编译到执行

Java 程序的执行过程

一个 Java 程序，首先经过 `javac` 编译成 `.class` 文件，然后 JVM 将其加载到方法区，执行引擎将会执行这些字节码。执行时，会翻译成操作系统相关的函数。JVM 作为 `.class` 文件的翻译存在，输入字节码，调用操作系统函数。

过程如下：Java 文件->编译器>字节码->JVM->机器码。

JVM 全称 Java Virtual Machine，也就是我们耳熟能详的 Java 虚拟机。它能识别 `.class` 后缀的文件，并且能够解析它的指令，最终调用操作系统上的函数，完成我们想要的操作。

JVM、JRE、JDK 的关系

JVM 只是一个翻译，把 Class 翻译成机器识别的代码，但是需要注意，JVM 不会自己生成代码，需要大家编写代码，同时需要很多依赖类库，这个时候就需要用到 JRE。

JRE 是什么，它除了包含 JVM 之外，提供了很多的类库（就是我们说的 jar 包，它可以提供一些即插即用的功能，比如读取或者操作文件，连接网络，使用 I/O 等等之类的）这些东西就是 JRE 提供的基础类库。JVM 标准加上实现的一大堆基础类库，就组成了 Java 的运行环境，也就是我们常说的 JRE（Java Runtime Environment）。

但对于程序员来说，JRE 还不够。我写完要编译代码，还需要调试代码，还需要打包代码、有时候还需要反编译代码。所以我们会使用 JDK，因为 JDK 还提供了一些非常好用的小工具，比如 javac（编译代码）、java、jar（打包代码）、javap（反编译<反汇编>）等。这个就是 JDK。

具体可以文档可以通过官网去下载：<https://www.oracle.com/java/technologies/javase-jdk8-doc-downloads.html>

JVM 的作用是：从软件层面屏蔽不同操作系统在底层硬件和指令的不同。这个就是我们在宏观方面对 JVM 的一个认识。








同时 JVM 是一个虚拟化的操作系统，类似于 Linux 或者 Windows 的操作系统，只是它架在操作系统上，接收字节码也就是 class，把字节码翻译成操作系统上的机器码且进行执行。

从跨平台到跨语言

跨平台：我们写的一个类，在不同的操作系统上（Linux、Windows、MacOS 等平台）执行，效果是一样，这个就是 JVM 的跨平台性。

为了实现跨平台型，不同操作系统有对应的 JDK 的版本。

<https://www.oracle.com/java/technologies/javase/javase-jdk8-downloads.html>

Product / File Description	File Size	Download
Linux ARM 32 Hard Float ABI	72.87 MB	 jdk-8u251-linux-arm32-vfp-hflt.tar.gz
Linux ARM 64 Hard Float ABI	69.77 MB	 jdk-8u251-linux-arm64-vfp-hflt.tar.gz
Linux x86 RPM Package	171.71 MB	 jdk-8u251-linux-i586.rpm
Linux x86 Compressed Archive	186.6 MB	 jdk-8u251-linux-i586.tar.gz
Linux x64 RPM Package	171.16 MB	 jdk-8u251-linux-x64.rpm
Linux x64 Compressed Archive	186.09 MB	 jdk-8u251-linux-x64.tar.gz
macOS x64	254.78 MB	 jdk-8u251-macosx-x64.dmg

跨语言（语言无关性）：JVM 只识别字节码，所以 JVM 其实跟语言是解耦的，也就是没有直接关联，JVM 运行不是翻译 Java 文件，而是识别 class 文件，这个一般称之为字节码。还有像 Groovy 、Kotlin、Scala 等等语言，它们其实也是编译成字节码，所以它们也可以在 JVM 上面跑，这个就是 JVM 的跨语言特征。**Java 的跨语言性一定程度上奠定了非常强大的 java 语言生态圈。**

JVM 的发展（非重点）

常见的 JVM 实现

Hotspot: 目前使用的最多的 Java 虚拟机。在命令行 `java -version`。它会输出你现在使用的虚拟机的名字、版本等信息、执行模式。

```
C:\Users\Administrator\Desktop>java -version
java version "1.8.0_101"
Java(TM) SE Runtime Environment (build 1.8.0_101-b13)
Java HotSpot(TM) 64-Bit Server VM (build 25.101-b13, mixed mode)
```

Jrocket: 原来属于 BEA 公司，曾号称世界上最快的 JVM，后被 Oracle 公司收购，合并于 Hotspot

J9: IBM 有自己的 java 虚拟机实现，它的名字叫做 J9。主要是用在 IBM 产品（IBM WebSphere 和 IBM 的 AIX 平台上）

TaobaoVM: 只有一定体量、一定规模的厂商才会开发自己的虚拟机，比如淘宝有自己的 VM,它实际上是 Hotspot 的定制版，专门为淘宝准备的，阿里、天猫都是用的这款虚拟机。

LiquidVM: 它是一个针对硬件的虚拟机，它下面是没有操作系统的（不是 Linux 也不是 windows），下面直接就是硬件，运行效率比较高。

zing: 它属于 zual 这家公司，非常牛，是一个商业产品，很贵！它的垃圾回收速度非常快（1 毫秒之内），是业界标杆。它的一个垃圾回收的算法后来被 Hotspot 吸收才有了现在的 ZGC。

JVM 整体知识模块

JVM 能涉及非常庞大的一块知识体系，比如内存结构、垃圾回收、类加载、性能调优、JVM 自身优化技术、执行引擎、类文件结构、监控工具等。但是在所有的知识体系中，都或多或少跟内存结构有一定的关系：
比如垃圾回收回收的就是内存、类加载加载到的地方也是内存、性能优化也涉及到内存优化、执行引擎与内存密不可分、类文件结构与内存的设计有关

系，监控工具也会监控内存。所以内存结构处于 JVM 中核心位置。也是属于我们入门 JVM 学习的最好的选择。

同时 JVM 是一个虚拟化的操作系统，所以除了要虚拟指令之外，最重要的一个事情就是需要虚拟化内存，这个虚拟化内存就是我们马上要讲到的 JVM 的内存区域。

JVM 的内存区域

运行时数据区域

运行时数据区的定义：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

Java 引以为豪的就是它的自动内存管理机制。相比于 C++ 的手动内存管理、复杂难以理解的指针等，Java 程序写起来就方便的多。

所以要深入理解 JVM 必须理解内存虚拟化的概念。

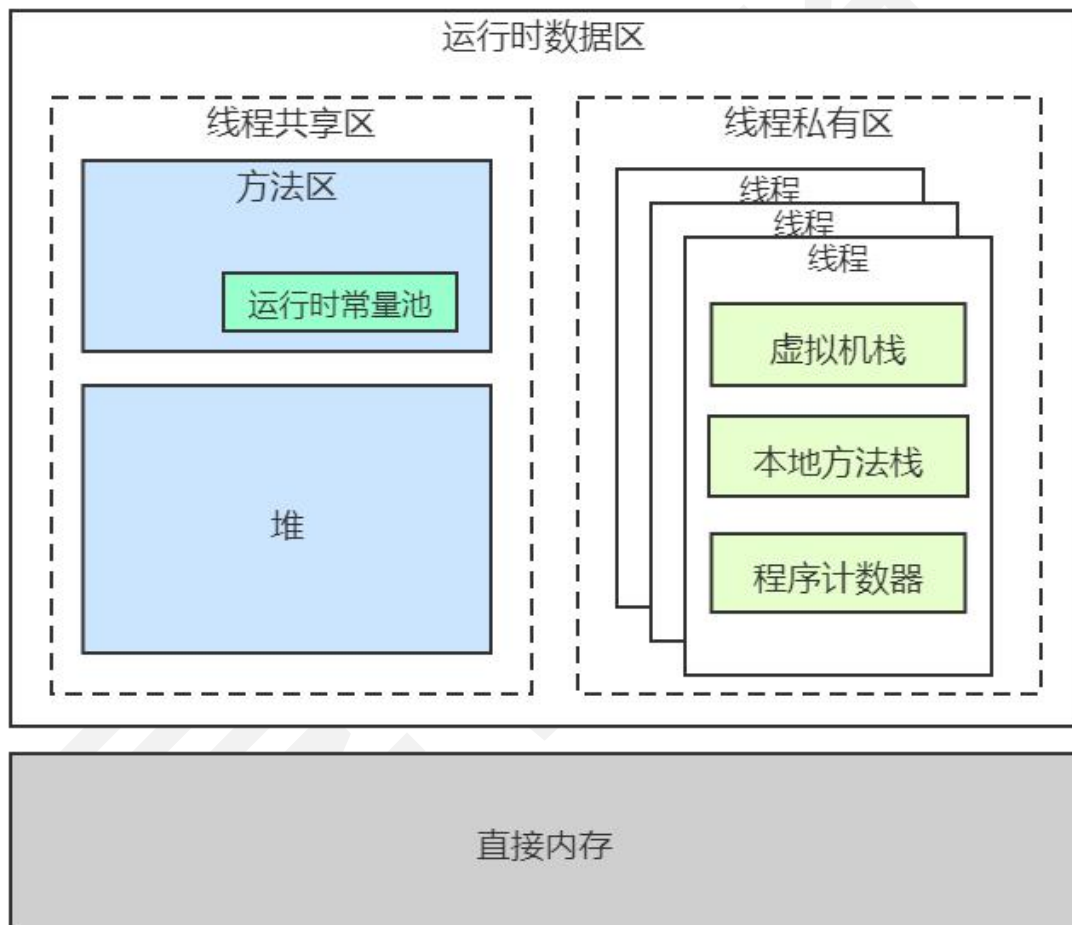
在 JVM 中，JVM 内存主要分为堆、程序计数器、方法区、虚拟机栈和本地方法栈等。

同时按照与线程的关系也可以这么划分区域：

线程私有区域：一个线程拥有单独的一份内存区域。

线程共享区域：被所有线程共享，且只有一份。

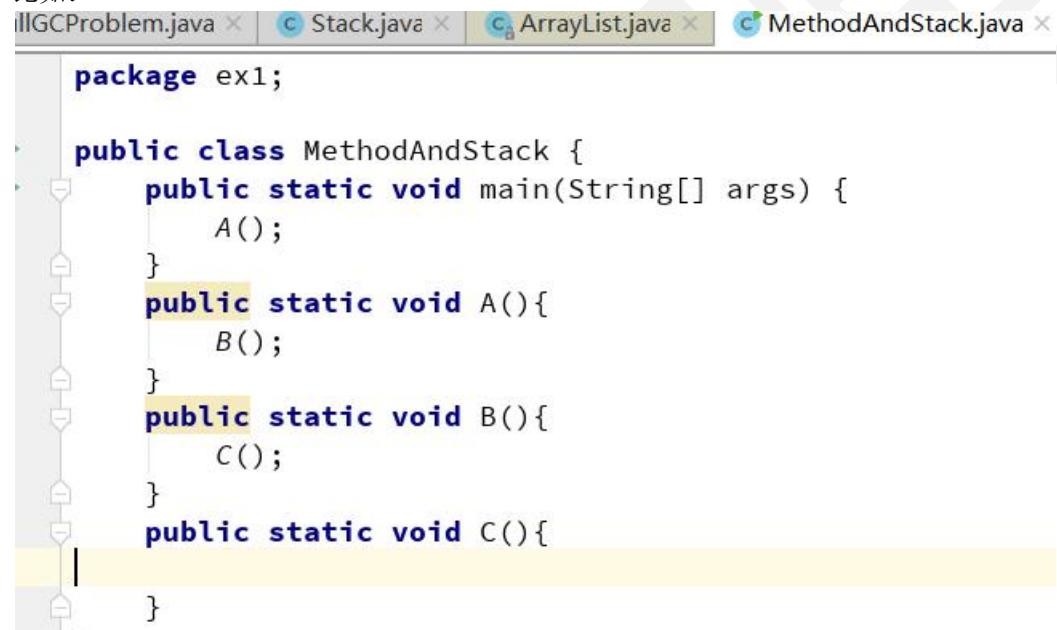
这里还有一个直接内存，这个虽然不是运行时数据区的一部分，但是会被频繁使用。你可以理解成没有被虚拟机化的操作系统上的其他内存（比如操作系统上有 8G 内存，被 JVM 虚拟化了 3G，那么还剩余 5G，JVM 是借助一些工具使用这 5G 内存的，这个内存部分称之为直接内存）



JAVA 方法的运行与虚拟机栈

虚拟机栈是线程运行 java 方法所需的数据，指令、返回地址。其实在我们实际的代码中，一个线程是可以运行多个方法的。

比如：



```
package ex1;

public class MethodAndStack {
    public static void main(String[] args) {
        A();
    }
    public static void A(){
        B();
    }
    public static void B(){
        C();
    }
    public static void C(){
    }
}
```

这段代码很简单，就是起一个 main 方法，在 main 方法运行中调用 A 方法，A 方法中调用 B 方法，B 方法中运行 C 方法。

我们把代码跑起来，线程 1 来运行这段代码，线程 1 跑起来，就会有一个对应 的虚拟机栈，同时执行每个方法的时候都会打包成一个栈帧。

比如 main 开始运行，打包一个栈帧送入到虚拟机栈。



C 方法运行完了，C 方法出栈，接着 B 方法运行完了，B 方法出栈、接着 A 方法运行完了，A 方法出栈，最后 main 方法运行完了，main 方法这个栈帧就出栈了。

这个就是 Java 方法运行对虚拟机栈的一个影响。虚拟机栈就是用来存储线程运行方法中的数据的。而每一个方法对应一个栈帧。

虚拟机栈

栈的数据结构：先进后出(FILO)的数据结构，

虚拟机栈的作用：在 JVM 运行过程中存储当前线程运行方法所需的数据，指令、返回地址。

虚拟机栈是基于线程的：哪怕你只有一个 main() 方法，也是以线程的方式运行的。在线程的生命周期中，参与计算的数据会频繁地入栈和出栈，栈的生命周期是和线程一样的。

虚拟机栈的大小缺省为 1M，可用参数 -Xss 调整大小，例如-Xss256k。

参数官方文档（JDK1.8）：<https://docs.oracle.com/javase/8/docs/technotes/tools/unix/java.html>

-Xss 大小

设置线程堆栈大小（以字节为单位）。附加字母 **k** 或 **K** 表示 KB，**m** 或 **M** 表示 MB，**g** 或 **G** 表示 GB。默认值取决于平台：

- Linux / ARM (32位) : 320 KB
- Linux / i386 (32位) : 320 KB
- Linux / x64 (64位) : 1024 KB
- OS X (64位) : 1024 KB
- Oracle Solaris / i386 (32位) : 320 KB
- Oracle Solaris / x64 (64位) : 1024 KB

下面的示例以不同的单位将线程堆栈大小设置为 1024 KB：

```
-Xss1分钟  
-Xss1024k  
-Xss1048576
```

此选项等效于 `-XX:ThreadStackSize`。

栈帧：在每个 Java 方法被调用的时候，都会创建一个栈帧，并入栈。一旦方法完成相应的调用，则出栈。

栈帧大体都包含四个区域：(局部变量表、操作数栈、动态连接、返回地址)

1、局部变量表：

顾名思义就是局部变量的表，用于存放我们的局部变量的（方法中的变量）。首先它是一个 32 位的长度，主要存放我们的 Java 的八大基础数据类型，一般 32 位就可以存放下，如果是 64 位的就使用高低位占用两个也可以存放下，如果是局部的一些对象，比如我们的 Object 对象，我们只需

要存放它的一个引用地址即可。

2、操作数据栈:

存放 java 方法执行的操作数的，它就是一个栈，先进后出的栈结构，操作数栈，就是用来操作的，操作的元素可以是任意的 java 数据类型，所以我们知道一个方法刚开始的时候，这个方法的操作数栈就是空的。

操作数栈本质上是 JVM 执行引擎的一个工作区，也就是方法在执行，才会对操作数栈进行操作，如果代码不执行，操作数栈其实就是空的。

3、动态连接:

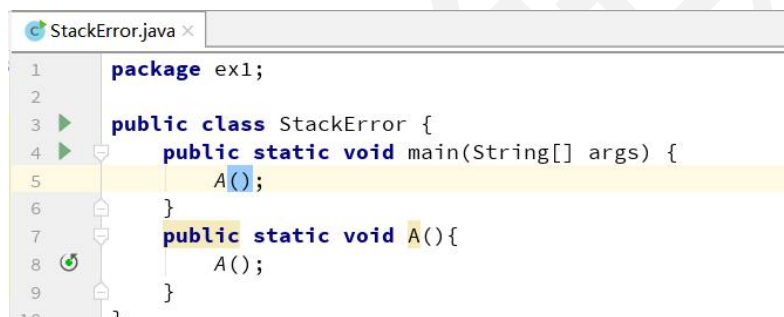
Java 语言特性多态（后续章节细讲，需要结合 class 与执行引擎一起来讲）。

4、返回地址:

正常返回（调用程序计数器中的地址作为返回）、异常的话（通过异常处理器表<非栈帧中的>来确定）

同时，虚拟机栈这个内存也不是无限大，它有大小限制，默认情况下是 1M。

如果我们不断的往虚拟机栈中入栈帧，但是就是不出栈的话，那么这个虚拟机栈就会爆掉。



```
1 package ex1;
2
3 public class StackError {
4     public static void main(String[] args) {
5         A();
6     }
7     public static void A(){
8         A();
9     }
10 }
```

Exception in thread "main" java.lang.StackOverflowError

程序计数器

较小的内存空间，当前线程执行的字节码的行号指示器；各线程之间独立存储，互不影响。

程序计数器是一块很小的内存空间，主要用来记录各个线程执行的字节码的地址，例如，分支、循环、跳转、异常、线程恢复等都依赖于计数器。

由于 Java 是多线程语言，当执行的线程数量超过 CPU 核数时，线程之间会根据时间片轮询争夺 CPU 资源。如果一个线程的时间片用完了，或者是其

它原因导致这个线程的 CPU 资源被提前抢夺，那么这个退出的线程就需要单独的一个程序计数器，来记录下一条运行的指令。

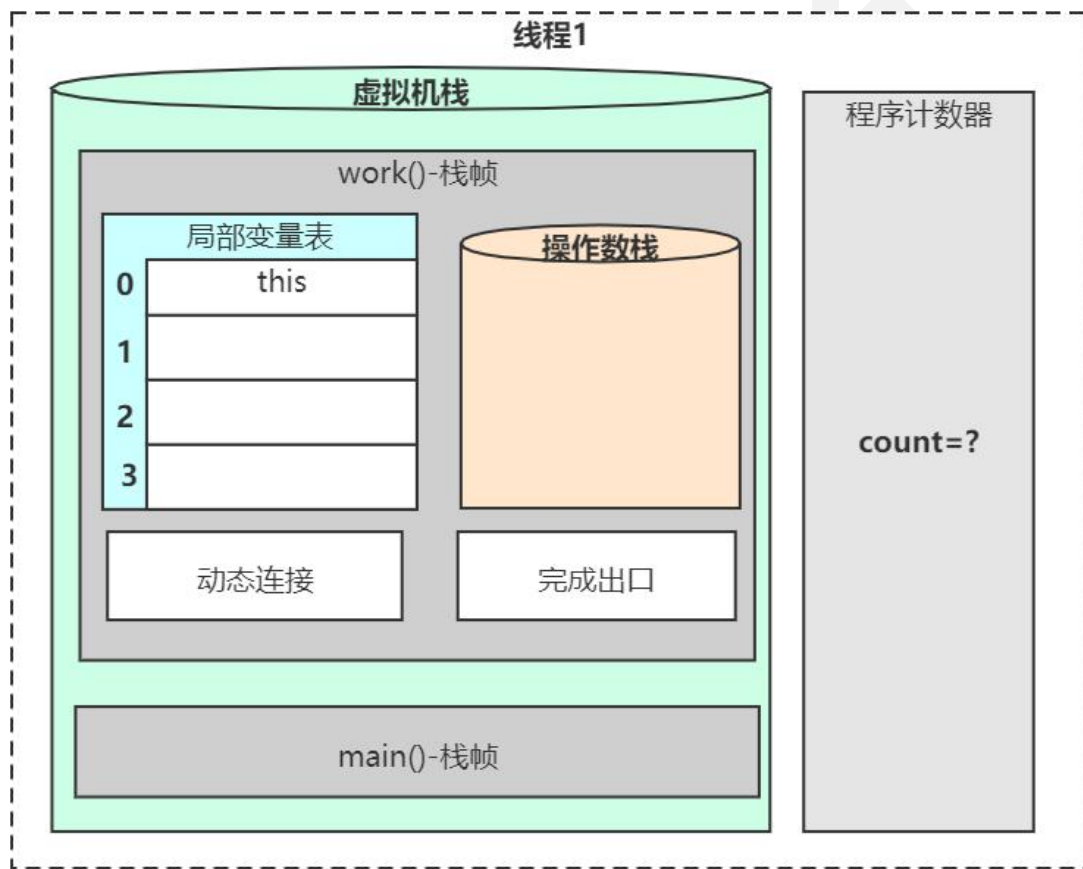
因为 JVM 是虚拟机，内部有完整的指令与执行的一套流程，所以在运行 Java 方法的时候需要使用程序计数器（记录字节码执行的地址或行号），如果是遇到本地方法（native 方法），这个方法不是 JVM 来具体执行，所以程序计数器不需要记录了，这个是因为在操作系统层面也有一个程序计数器，这个会记录本地代码的执行的地址，所以在执行 native 方法时，JVM 中程序计数器的值为空(Undefined)。

另外程序计数器也是 JVM 中唯一不会 OOM(OutOfMemory)的内存区域。

栈帧执行对内存区域的影响

对 class 进行反汇编 `javap -c XXXX.class`

字节码助记码解释地址: <https://cloud.tencent.com/developer/article/1333540>



在 JVM 中，基于解释执行的这种方式是基于栈的引擎，这个说的栈，就是操作数栈。

运行时数据区及 JVM 的整体内存结构

本地方法栈

本地方法栈跟 Java 虚拟机栈的功能类似，Java 虚拟机栈用于管理 Java 函数的调用，而本地方法栈则用于管理本地方法的调用。但本地方法并不是用 Java 实现的，而是由 C 语言实现的(比如 `Object.hashCode` 方法)。

本地方法栈是和虚拟机栈非常相似的一个区域，它服务的对象是 `native` 方法。你甚至可以认为虚拟机栈和本地方法栈是同一个区域。虚拟机规范无强制规定，各版本虚拟机自由实现，HotSpot 直接把本地方法栈和虚拟机栈合二为一。

方法区

方法区 (Method Area) 是可供各条线程共享的运行时内存区域。它存储了每一个类的结构信息，例如运行时常量池 (Runtime Constant Pool) 字段和方法数据、构造函数和普通方法的字节码内容、还包括一些在类、实例、接口初始化时用到的特殊方法

方法区是 JVM 对内存的“逻辑划分”，在 JDK1.7 及之前很多开发者都习惯将方法区称为“永久代”，是因为在 HotSpot 虚拟机中，设计人员使用了永久代来实现了 JVM 规范的方法区。在 JDK1.8 及以后使用了元空间来实现方法区。

元空间

方法区与堆空间类似，也是一个共享内存区，所以方法区是线程共享的。假如两个线程都试图访问方法区中的同一个类信息，而这个类还没有装入 JVM，那么此时就只允许一个线程去加载它，另一个线程必须等待。

在 HotSpot 虚拟机、Java7 版本中已经将永久代的静态变量和运行时常量池转移到了堆中，其余部分则存储在 JVM 的非堆内存中，而 Java8 版本已经将方法区中实现的永久代去掉了，并用元空间 (class metadata) 代替了之前的永久代，并且元空间的存储位置是本地内存。

元空间大小参数：

jdk1.7 及以前（初始和最大值）：-XX:PermSize; -XX:MaxPermSize;
jdk1.8 以后（初始和最大值）：-XX:MetaspaceSize; -XX:MaxMetaspaceSize
jdk1.8 以后大小就只受本机总内存的限制（如果不设置参数的话）

JVM 参数参考：<https://docs.oracle.com/javase/8/docs/technotes/tools/unix/java.html>

Java8 为什么使用元空间替代永久代，这样做有什么好处呢？

官方给出的解释是：

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

永久代内存经常不够用或发生内存溢出，抛出异常 `java.lang.OutOfMemoryError: PermGen`。这是因为在 JDK1.7 版本中，指定的 PermGen 区大小为 8M，由于 PermGen 中类的元数据信息在每次 FullGC 的时候都可能被收集，回收率都偏低，成绩很难令人满意；还有为 PermGen 分配多大的空间很难确定，PermSize 的大小依赖于很多因素，比如，JVM 加载的 class 总数、常量池的大小和方法的大小等。

运行时常量池

运行时常量池（Runtime Constant Pool）是每一个类或接口的常量池（Constant_Pool）的运行时表示形式，它包括了若干种不同的常量：从编译期可知的数值字面量到必须运行期解析后才能获得的方法或字段引用。

运行时常量池是方法区的一部分。运行时常量池相对于 Class 常量池的另外一个重要特征是具备动态性（Class 常量池在类加载章节会具体讲）。

堆

堆是 JVM 上最大的内存区域，我们申请的几乎所有的对象，都是在这里存储的。我们常说的垃圾回收，操作的对象就是堆。

堆空间一般是程序启动时，就申请了，但是并不一定会全部使用。堆一般设置成可伸缩的。

随着对象的频繁创建，堆空间占用的越来越多，就需要不定期的对不再使用的对象进行回收。这个在 Java 中，就叫作 GC（Garbage Collection）。

那一个对象创建的时候，到底是在堆上分配，还是在栈上分配呢？这和两个方面有关：对象的类型和在 Java 类中存在的位置。

Java 的对象可以分为基本数据类型和普通对象。

对于普通对象来说，JVM 会首先在堆上创建对象，然后在其他地方使用的其实是它的引用。比如，把这个引用保存在虚拟机栈的局部变量表中。

对于基本数据类型来说（byte、short、int、long、float、double、char），有两种情况。

当你在方法体内声明了基本数据类型的对象，它就会在栈上直接分配。其他情况，都是在堆上分配。

堆大小参数：

-Xms：堆的最小值；

-Xmx：堆的最大值；

-Xmn：新生代的大小；

-XX:NewSize：新生代最小值；

-XX:MaxNewSize：新生代最大值；

例如- Xmx256m

-Xms 大小

设置堆的初始大小（以字节为单位）。此值必须是1024的倍数且大于1 MB。追加字母k或K表示千字节，m或M表示兆字节，g或G千兆字节。

以下示例说明如何使用各种单位将分配的内存大小设置为6 MB：

```
-Xms6291456  
-Xms6144k  
-Xms6m
```

如果未设置此选项，则初始大小将设置为老一代和年轻一代分配的大小之和。可以使用-Xmn选项或-XX:NewSize选项设置年轻代的堆的初始大小。

-Xmx 大小

指定内存分配池的最大大小（以字节为单位）。此值必须是1024的倍数且大于2 MB。追加字母k或K表示千字节，m或M表示兆字节，g或G千兆字节。默认值是在运行时根据系统配置选择的。对于服务器部署，-Xms并-Xmx经常设置为相同的值。请参阅位于的Java SE HotSpot虚拟机垃圾收集优化指南中的“人体工程学”部分

<http://docs.oracle.com/javase/8/docs/technotes/guides/vm/gctuning/index.html>。

下面的示例演示如何使用各种单位将分配的最大内存大小设置为80 MB：

```
-Xmx83886080  
-Xmx81920k  
-Xmx80m
```

该-Xmx选项等效于-XX:MaxHeapSize。

直接内存（堆外内存）

直接内存有一种更加科学的叫法，堆外内存。

JVM 在运行时，会从操作系统申请大块的堆内存，进行数据的存储；同时还有虚拟机栈、本地方法栈和程序计数器，这块称之为栈区。操作系统剩余的内存也就是堆外内存。

它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；如果使用了 NIO,这块区域会被频繁使用，在 java 堆内可以用 directByteBuffer 对象直接引用并操作；

这块内存不受 java 堆大小限制，但受本机总内存的限制，可以通过-XX:MaxDirectMemorySize 来设置（默认与堆内存最大值一样），所以也会出现 OOM 异

常。

小结:

1、直接内存主要是通过 `DirectByteBuffer` 申请的内存，可以使用参数 “`MaxDirectMemorySize`” 来限制它的大小。

2、其他堆外内存，主要是指使用了 `Unsafe` 或者其他 `JNI` 手段直接申请的内存。

堆外内存的泄漏是非常严重的，它的排查难度高、影响大，甚至会造成主机的死亡。后续章节会详细讲。

同时，要注意 `Oracle` 之前计划在 `Java 9` 中去掉 `sun.misc.Unsafe` API。这里删除 `sun.misc.Unsafe` 的原因之一是使 `Java` 更加安全，并且有替代方案。

目前我们主要针对的 `JDK1.8`，`JDK1.9` 暂时不放入讨论范围中，我们大致知道 `java` 的发展即可。

深入理解 JVM 的内存区域

深入理解运行时数据区

代码示例:

```

public class JVMObject {
    public final static String MAN_TYPE = "man"; // 常量
    public static String WOMAN_TYPE = "woman"; // 静态变量
    public static void main(String[] args) throws Exception {
        Teacher T1 = new Teacher();
        T1.setName("Mark");
        T1.setSexType(MAN_TYPE);
        T1.setAge(36);
        Teacher T2 = new Teacher();
        T2.setName("King");
        T2.setSexType(MAN_TYPE);
        T2.setAge(18);
        Thread.sleep(Integer.MAX_VALUE); // 线程休眠
    }
}

class Teacher{
    String name;
    String sexType;
    int age;

    public String getName() { return name; }
    public void setName(String name) { this.name = name; }

    public String getSexType() { return sexType; }
    public void setSexType(String sexType) { this.sexType = sexType; }
    public int getAge() { return age; }
    public void setAge(int age) { this.age = age; }
}

```

1. JVM 向操作系统申请内存:
JVM 第一步就是通过配置参数或者默认配置参数向操作系统申请内存空间, 根据内存大小找到具体的内存分配表, 然后把内存段的起始地址和终止地址分配给 JVM, 接下来 JVM 就进行内部分配。
2. JVM 获得内存空间后, 会根据配置参数分配堆、栈以及方法区的内存大小

-Xms30m -Xmx30m -Xss1m -XX:MaxMetaspaceSize=30m

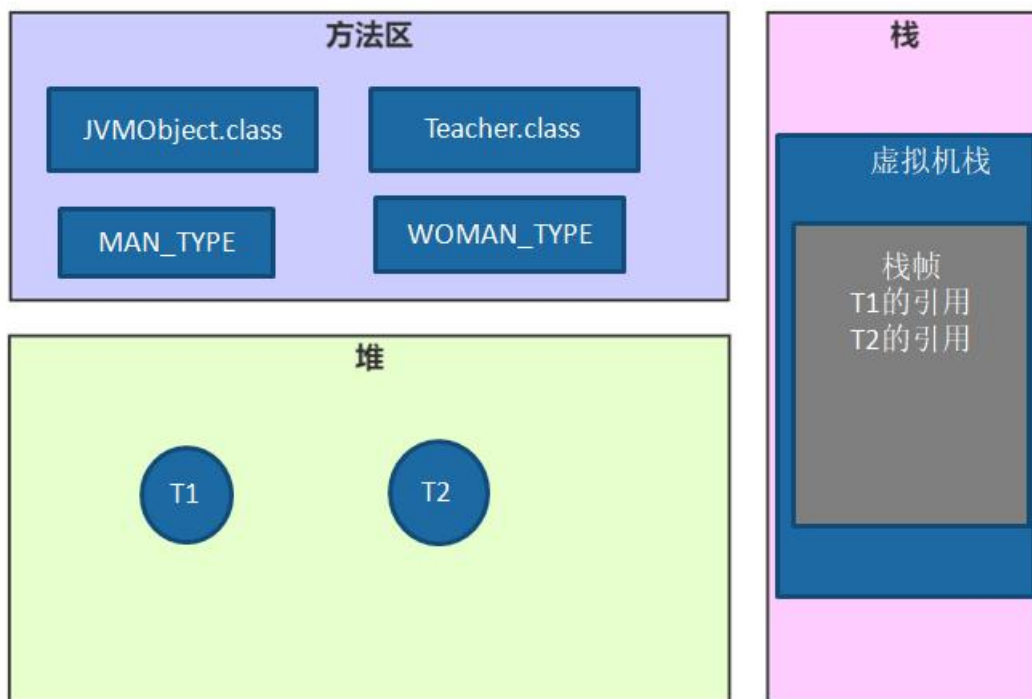
3. 类加载（类加载的细节后续章节会讲）：

这里主要是把 class 放入方法区、还有 class 中的静态变量和常量也要放入方法区

4. 执行方法及创建对象：

启动 main 线程，执行 main 方法，开始执行第一行代码。此时堆内存中会创建一个 student 对象，对象引用 student 就存放在栈中。

后续代码中遇到 new 关键字，会再创建一个 student 对象，对象引用 student 就存放在栈中。



总结一下 JVM 运行内存的整体流程

JVM 在操作系统上启动，申请内存，先进行运行时数据区的初始化，然后把类加载到方法区，最后执行方法。
方法的执行和退出过程在内存上的体现上就是虚拟机栈中栈帧的入栈和出栈。
同时在方法的执行过程中创建的对象一般情况下都是放在堆中，最后堆中的对象也是需要进行垃圾回收清理的。

从底层深入理解运行时数据区

堆空间分代划分

堆被划分为新生代和老年代（Tenured），新生代又被进一步划分为 Eden 和 Survivor 区，最后 Survivor 由 From Survivor 和 To Survivor 组成。
（先需要有概念，后续对象分配和垃圾回收会细讲这块）



GC 概念

GC- Garbage Collection 垃圾回收，在 JVM 中是自动化的垃圾回收机制，我们一般不用去关注，在 JVM 中 GC 的重要区域是堆空间。
我们也可以通过一些额外方式主动发起它，比如 `System.gc()`,主动发起。（项目中切记不要使用）

JHSDB 工具

JHSDB 是一款基于服务性代理实现的进程外调试工具。服务性代理是 HotSpot 虚拟机中一组用于映射 Java 虚拟机运行信息的，主要基于 Java 语言实现的 API 集合。

JDK1.8 的开启方式

开启 HSDB 工具

Jdk1.8 启动 JHSDB 的时候必须将 sawindbg.dll（一般会在 JDK 的目录下）复制到对应目录的 jre 下（注意在 win 上安装了 JDK1.8 后往往同级目录下有一个 jre 的目录）



然后到目录：C:\Program Files\Java\jdk1.8.0_101\lib 进入命令行，执行 `java -cp .\sa-jdi.jar sun.jvm.hotspot.HSDB`

C:\Windows\system32\cmd.exe - java -cp .\sa-jdi.jar sun.jvm.hotspot.HSDB

```
: \Program Files\Java\jdk1.8.0_101\lib> java -cp .\sa-jdi.jar sun.jvm.hotspot.HSDB
```

HSDB - HotSpot Debugger

File Tools Windows

JDK1.9 及以后的开启方式

进入 JDK 的 bin 目录下，我们可以在命令行中使用 `jhsdb hsdh` 来启动它

代码改造

VM 参数加入：

`-XX:+UseConcMarkSweepGC`

-XX: + UseConcMarkSweepGC

启用CMS垃圾收集器用于旧版本。当吞吐量 (-XX:+UseParallelGC) 垃圾收集器无法满足应用程序延迟要求时, Oracle建议您使用CMS垃圾收集器。G1垃圾收集器 (-XX:+UseG1GC) 是另一种选择。

默认情况下, 此选项是禁用的, 并且将根据计算机的配置和JVM的类型自动选择收集器。启用此选项后, 该-XX:+UseParNewGC选项将自动设置, 并且您不应禁用它, 因为JDK 8中已弃用以下选项组合: **-XX:+UseConcMarkSweepGC -XX:-UseParNewGC**。

-XX:-UseCompressedOops

-XX: -UseCompressedOops

禁用压缩指针的使用。默认情况下, 此选项处于启用状态, 并且当Java堆大小小于32 GB时, 将使用压缩指针。启用此选项后, 对象引用将表示为32位偏移量而不是64位指针, 这通常在运行Java堆大小小于32 GB的应用程序时提高性能。此选项仅适用于64位JVM。

当Java堆大小大于32GB时, 也可以使用压缩指针。参见-XX:ObjectAlignmentInBytes选项。

```
package com.jvm.ex2;
```

```
/**
```

```
 * @author King老师
```

```
 *
```

```
 * -Xms30m -Xmx30m -Xss1m -XX:MaxMetaspaceSize=30m
```

```
 */
```

指定垃圾回收器

```
-XX:+UseConcMarkSweepGC -XX:-UseCompressedOops
```

```
public class JVMOBJectByGc {  
    public final static String MAN_TYPE = "man"; // 常量  
    public static String WOMAN_TYPE = "woman"; // 静态变量  
  
    public static void main(String[] args) throws Exception {  
        Teacher T1 = new Teacher();  
        T1.setName("Mark");  
        T1.setSexType(MAN_TYPE);  
        T1.setAge(36);  
        for (int i=0;i<15;i++){//进行15次垃圾回收  
            System.gc();//垃圾回收  
        }  
        Teacher T2 = new Teacher();  
        T2.setName("King");  
        T2.setSexType(MAN_TYPE);  
        T2.setAge(18);  
        Thread.sleep(Integer.MAX_VALUE);//线程休眠很久很久  
    }  
}
```

主动发起GC

JHSDB 中查看对象

实例代码启动



因为 JVM 启动有一个进程，需要借助一个命令 `jps` 查找到对应程序的进程

C:\Windows\system32\cmd.exe

```
G:\VIP课三期\JVM>jps
14144
16016 HSDB
9940 Launcher
1160 JVMObjectBvGc
14540 Jps

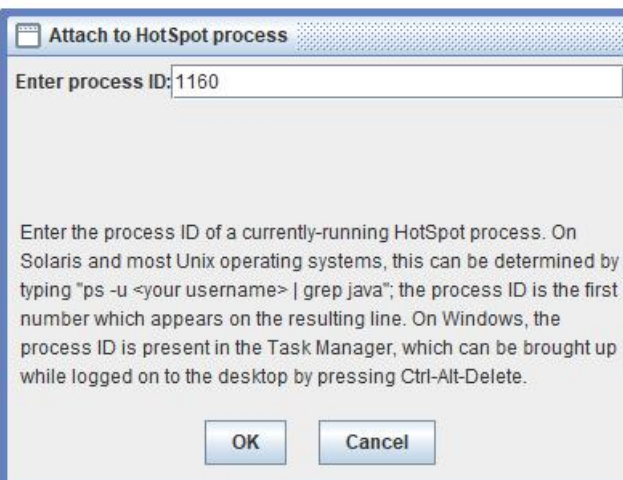
G:\VIP课三期\JVM>
```

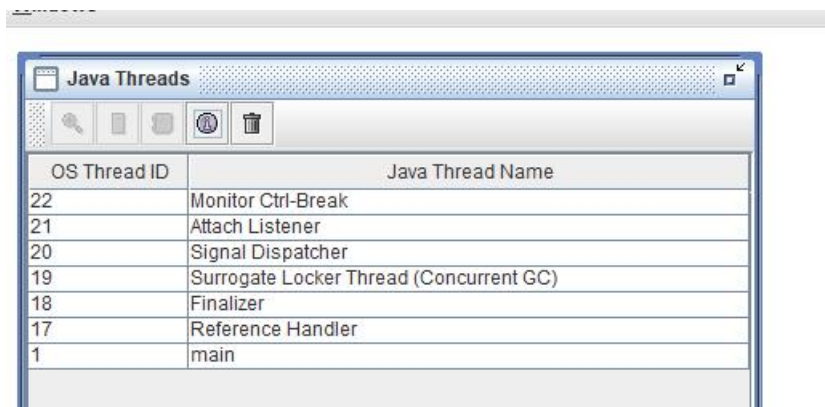
在 JHSDB 工具中 attach 上去

HSDB - HotSpot Debugger

File Tools Windows

Attach to HotSpot process... Alt-A
Open HotSpot core file... Alt-O
Connect to debug server... Alt-S
Detach Alt-D
Exit Alt-X

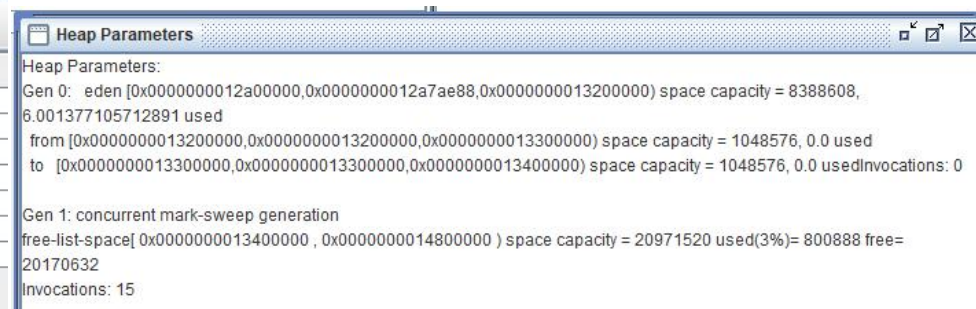
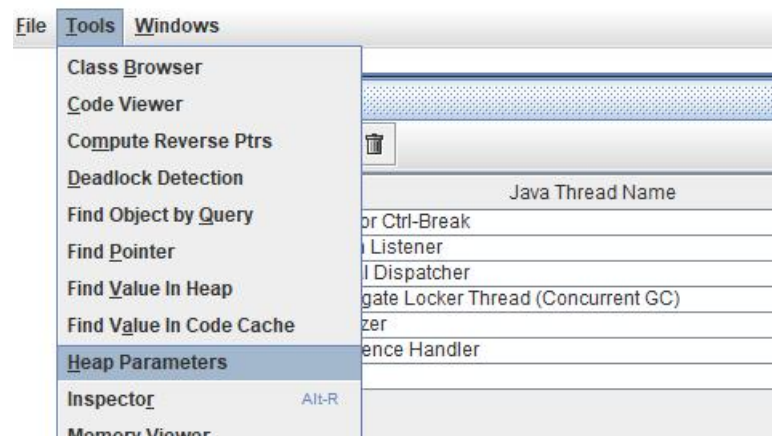




JHSDB 中查看对象

查看堆参数:

HSDB - HotSpot Debugger

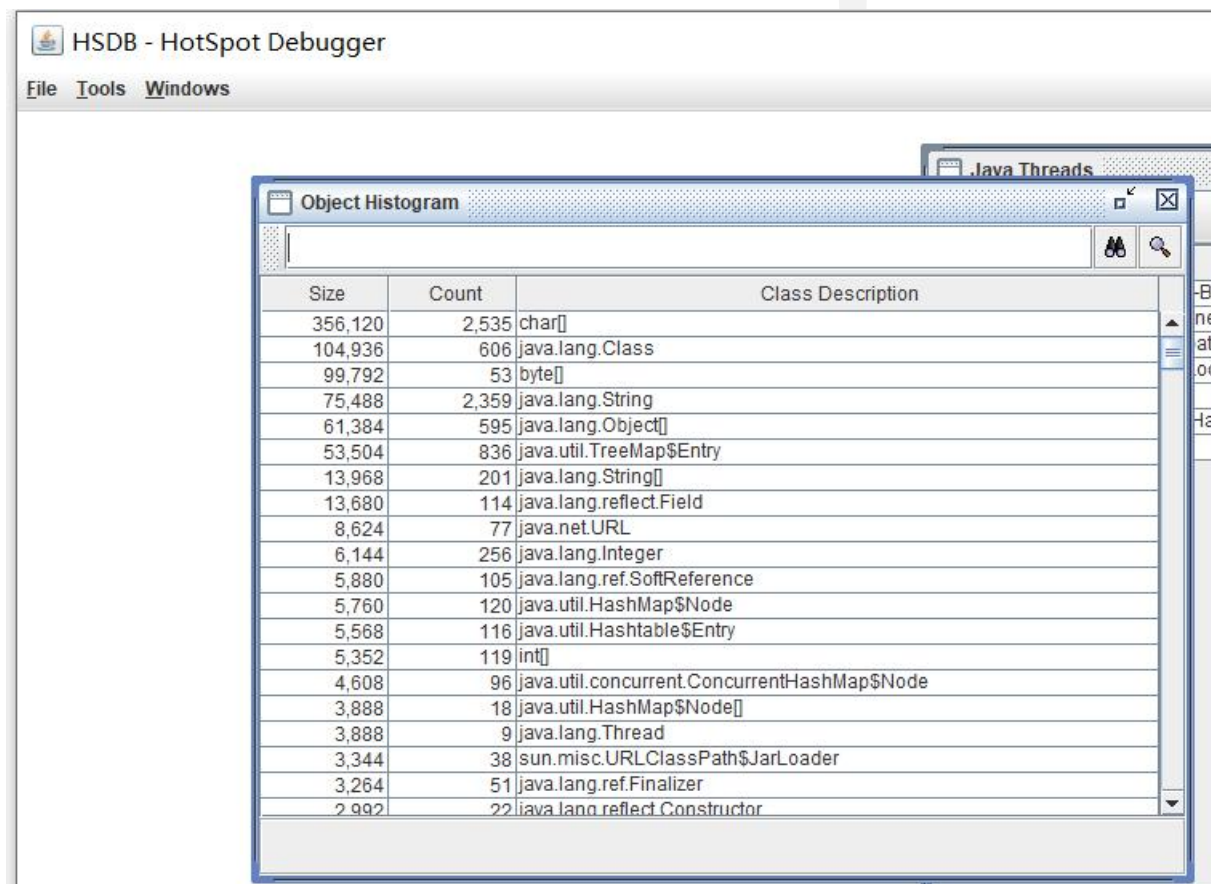


上图中可以看到实际 JVM 启动过程中堆中参数的对照，可以看到，在不启动内存压缩的情况下。堆空间里面的分代划分都是连续的。

再来查看对象：



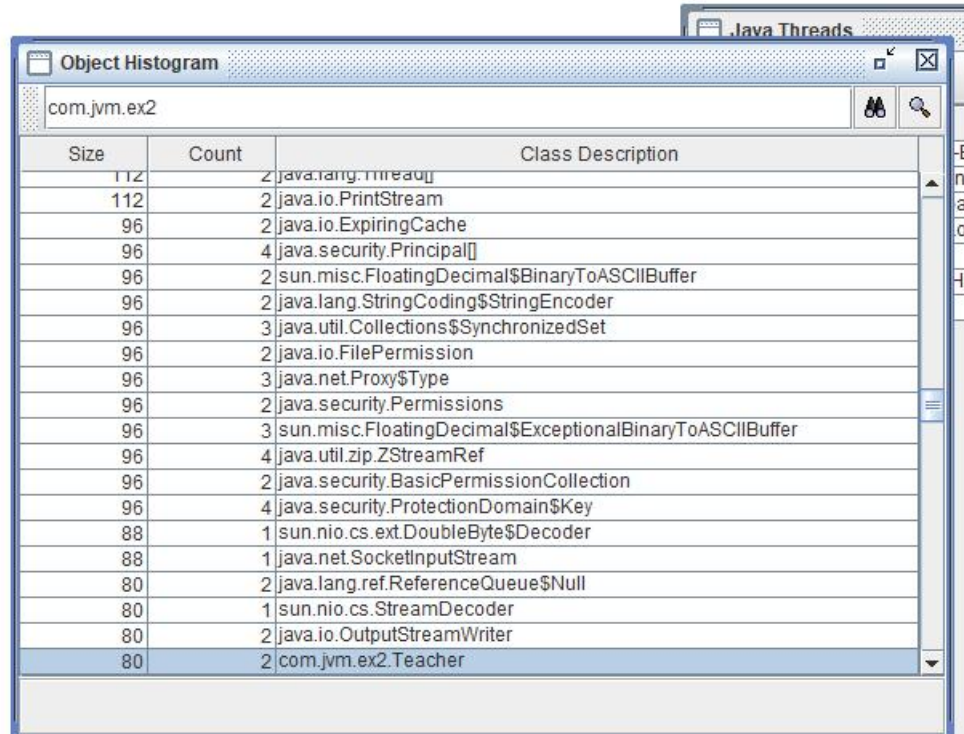
这里可以看到 JVM 中所有的对象，都是基于 class 的对象



全路径名搜索

HotSpot Debugger

Tools Windows



Size	Count	Class Description
112	2	java.lang.Thread
112	2	java.io.PrintStream
96	2	java.io.ExpiringCache
96	4	java.security.Principal
96	2	sun.misc.FloatingDecimal\$BinaryToASCIIBuffer
96	2	java.lang.StringCoding\$StringEncoder
96	3	java.util.Collections\$SynchronizedSet
96	2	java.io.FilePermission
96	3	java.net.Proxy\$Type
96	2	java.security.Permissions
96	3	sun.misc.FloatingDecimal\$ExceptionalBinaryToASCIIBuffer
96	4	java.util.zip.ZStreamRef
96	2	java.security.BasicPermissionCollection
96	4	java.security.ProtectionDomain\$Key
88	1	sun.nio.cs.ext.DoubleByte\$Decoder
88	1	java.net.SocketInputStream
80	2	java.lang.ref.ReferenceQueue\$Null
80	1	sun.nio.cs.StreamDecoder
80	2	java.io.OutputStreamWriter
80	2	com.jvm.ex2.Teacher

双击出现这个 Teacher 类的对象，两个，就是 T1 和 T2 对象。

... HotSpot Debugger

ools Windows

Object Histogram

com.jvm.ex2

Size	Count	Class Description
112	2	ja
112	2	ja
96	2	ja
96	4	ja
96	2	st
96	2	ja
96	3	ja
96	2	ja
96	3	ja
96	2	ja
96	3	st
96	4	ja
96	2	ja
96	4	ja
88	1	st
88	1	ja
80	2	ja
80	1	st
80	2	ja
80	2	cc

Java Threads

Java Thread Name
-Break

Show Objects of Type

Address	Oop	Class Description
0x0000000012a28f70	Oop for com/jvm/ex2/Teacher	InstanceKlass for com/jvm/ex2/T...
0x00000000134c21b8	Oop for com/jvm/ex2/Teacher	InstanceKlass for com/jvm/ex2/T...

Inspect Compute Liveness

Object Histogram

com.jvm.ex2

Size	Count	Class Description
112	2	java.lang.Thread
112	1	java.io.PrintStream

Show Objects of Type

Address	Oop
0x0000000012a28f70	Oop for com/jvm/ex2/Teacher
0x00000000134c21b8	Oop for com/jvm/ex2/Teacher

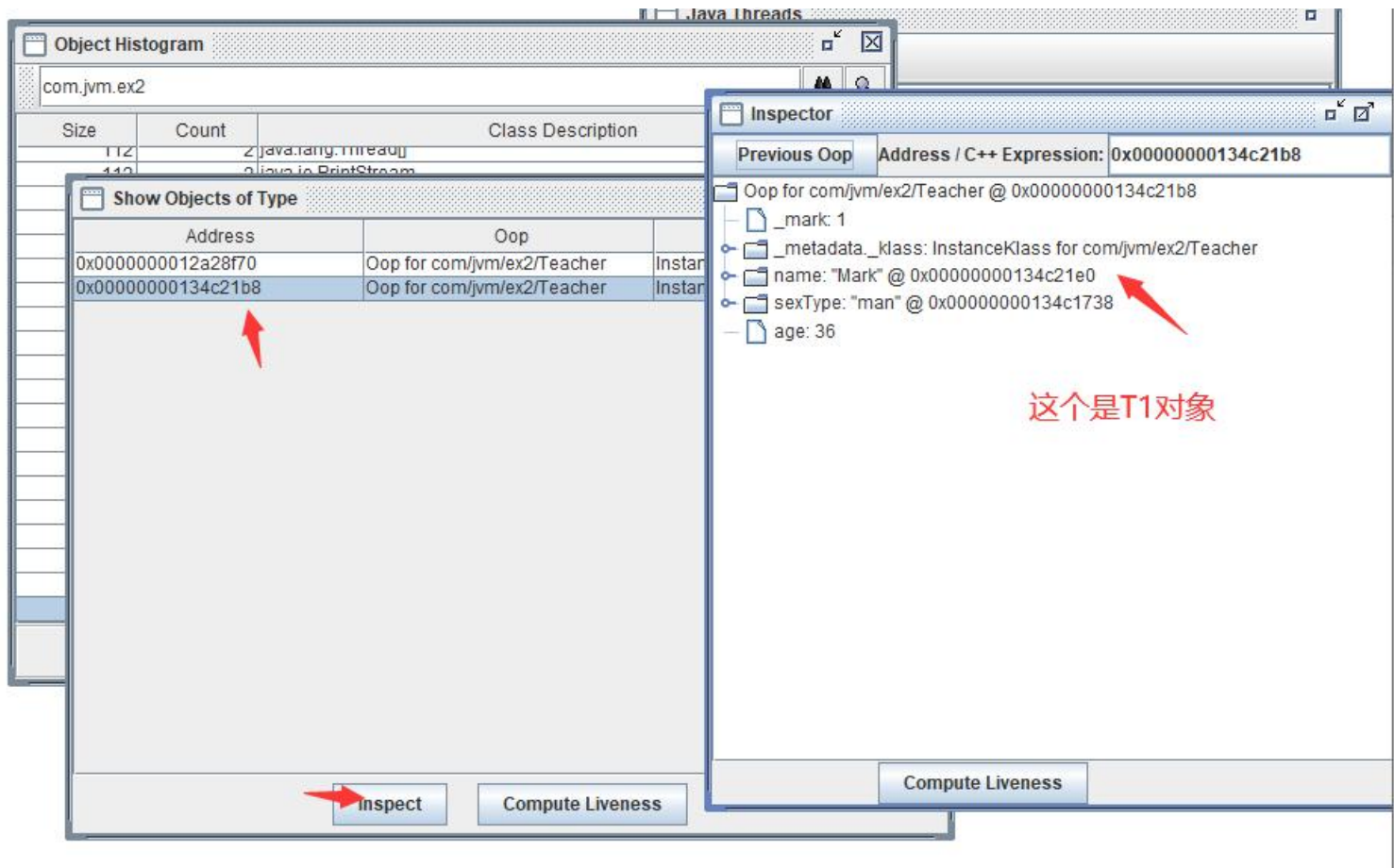
Inspector

Previous Oop Address / C++ Expression: 0x0000000012a28f70

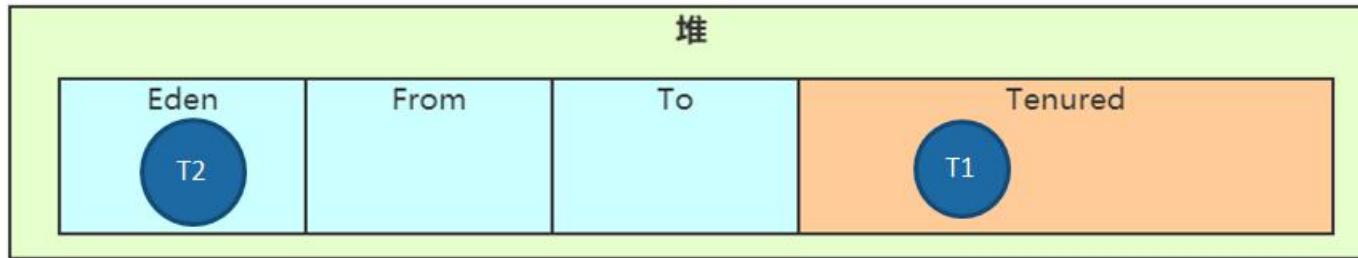
- Oop for com/jvm/ex2/Teacher @ 0x0000000012a28f70
 - _mark: 1
 - _metadata._klass: InstanceKlass for com/jvm/ex2/Teacher
 - name: "King" @ 0x0000000012a28f98
 - sexType: "male" @ 0x00000000134c1738
 - age: 18

Inspect Compute Liveness Compute Liveness

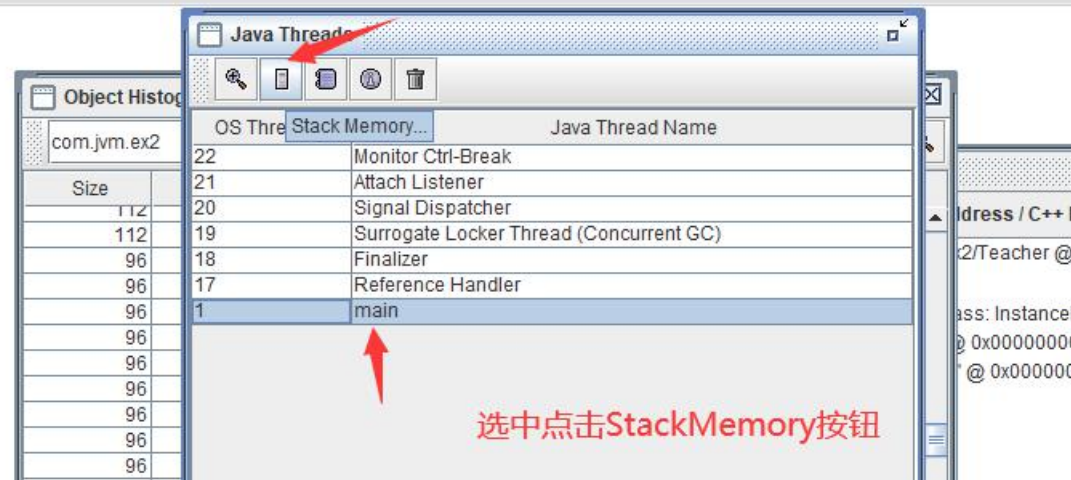
这个是T2对象

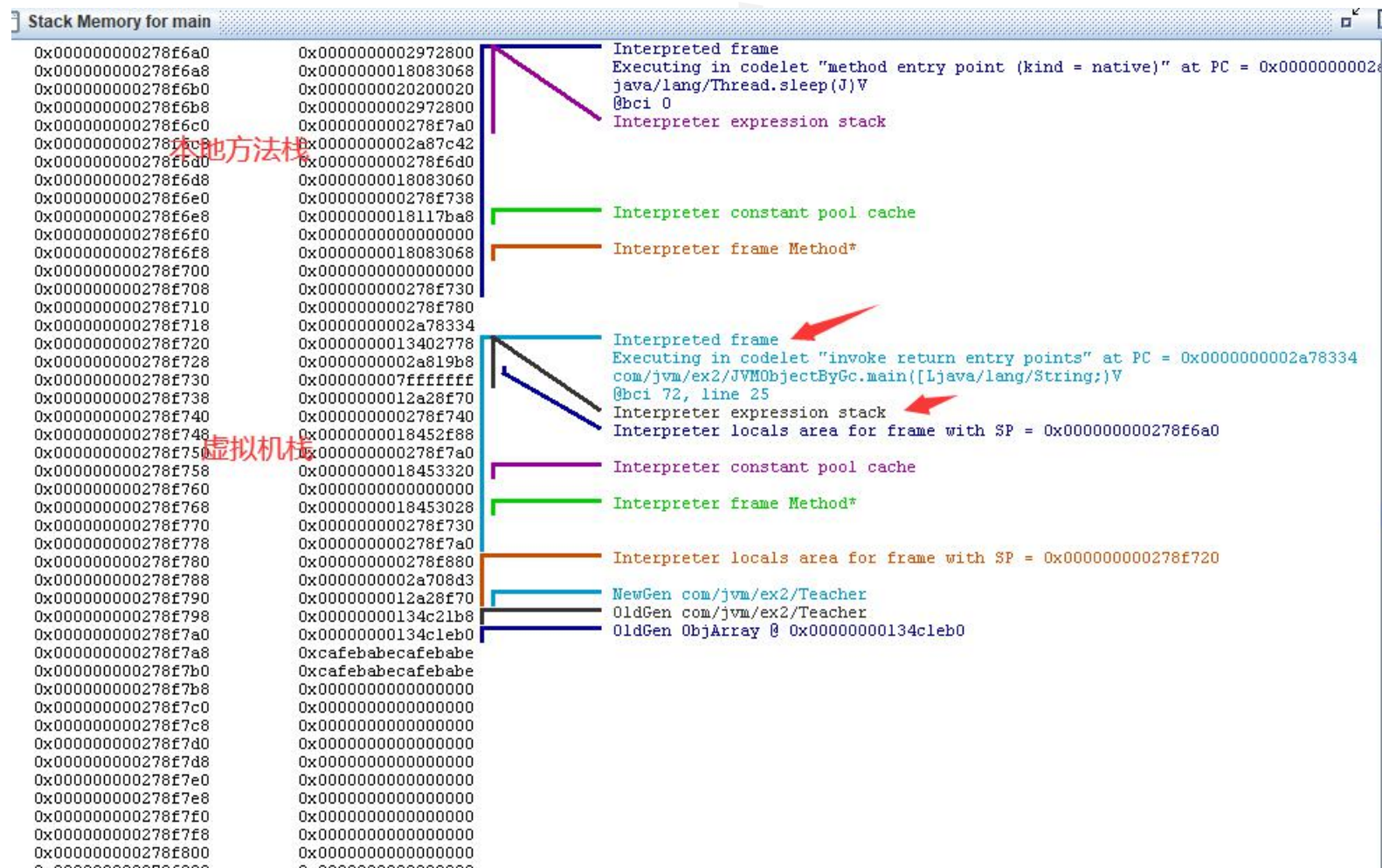


最后再对比一下堆中分代划分可以得出为什么 T1 在 Eden,T2 在老年代



JHSDB 中查看栈





从上图中可以验证栈内存，同时也可以验证到虚拟机栈和本地方法栈在 Hotspot 中是合二为一的实现了。

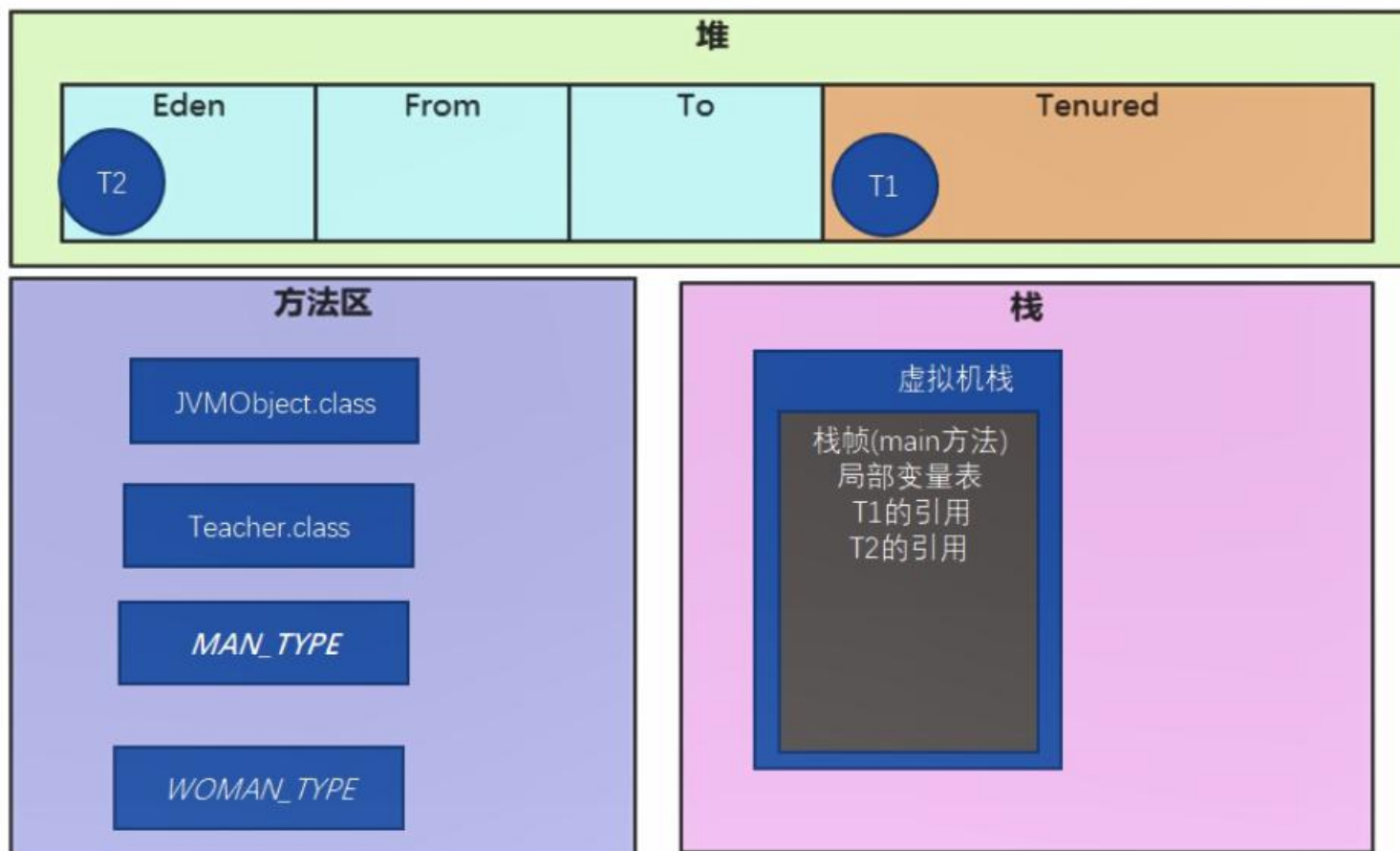
```

JVMObject.java x
1  package com.jvm.ex1;
2
3  /**
4   * @author King老师
5   *
6   * -Xms30m -Xmx30m
7   */
8
9  public class JVMObject {
10     public final static String MAN_TYPE = "man"; // 常量
11     public static String WOMAN_TYPE = "woman"; // 静态变量
12
13     public static void main(String[] args) throws Exception {
14         Teacher T1 = new Teacher(); // the1 方法的执行中，不可回收（存活）
15         T1.setName("Mark");
16         T1.setSexType(MAN_TYPE);
17         T1.setAge(36);
18         for (int i=0;i<15;i++){//进行15次垃圾回收
19             System.gc();
20         }
21         System.out.println("由左至右点");
22     }
23 }

```

当我们通过 Java 运行以上代码时，JVM 的整个处理过程如下：

1. JVM 向操作系统申请内存，JVM 第一步就是通过配置参数或者默认配置参数向操作系统申请内存空间。
2. JVM 获得内存空间后，会根据配置参数分配堆、栈以及方法区的内存大小。
3. 完成上一个步骤后，JVM 首先会执行构造器，编译器会在 .java 文件被编译成 .class 文件时，收集所有类的初始化代码，包括静态变量赋值语句、静态代码块、静态方法，静态变量和常量放入方法区
4. 执行方法。启动 main 线程，执行 main 方法，开始执行第一行代码。此时堆内存中会创建一个 Teacher 对象，对象引用 student 就存放在栈中。执行其他方法时，具体的操作：栈帧执行对内存区域的影响。[栈帧执行对内存区域的影响](#)



从底层深入理解运行时数据区（总结）

深入辨析堆和栈

■ 功能

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

■ 线程独享还是共享

- 栈内存归属于单个线程，每个线程都会有一个栈内存，其存储的变量只能在其所属线程中可见，即栈内存可以理解成线程的私有内存。
- 堆内存中的对象对所有线程可见。堆内存中的对象可以被所有线程访问。

■ 空间大小

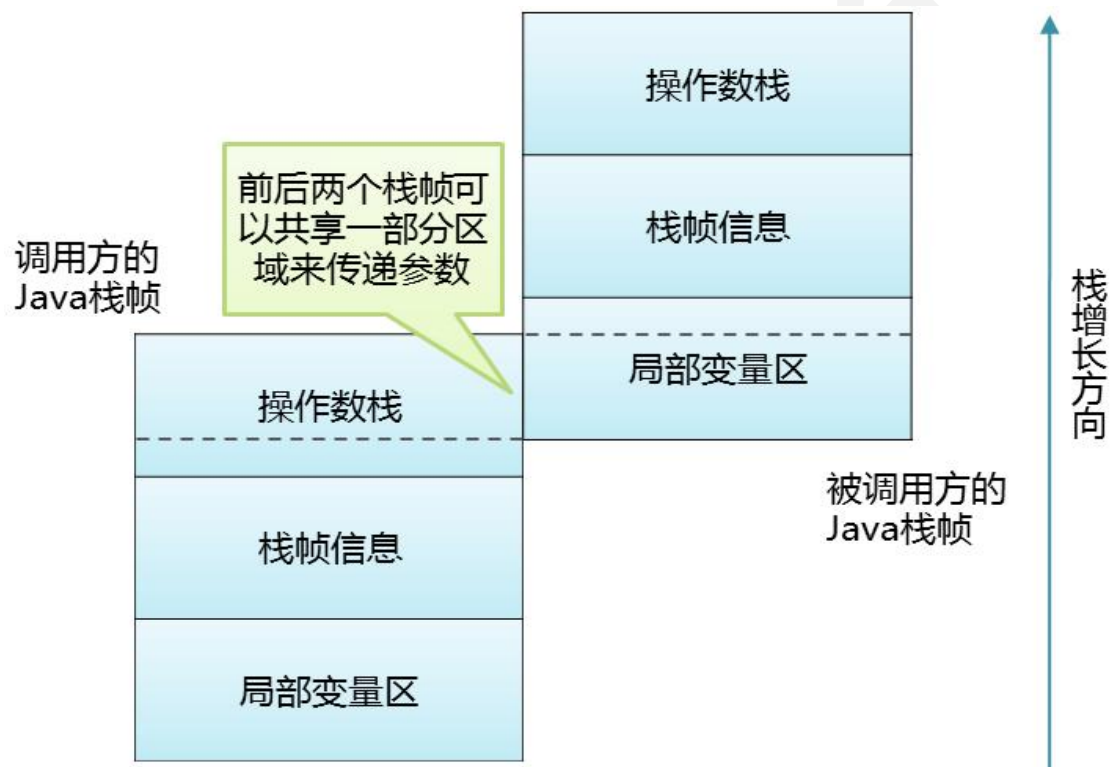
栈的内存要远远小于堆内存

虚拟机内存优化技术

栈的优化技术——栈帧之间数据的共享

在一般的模型中，两个不同的栈帧的内存区域是独立的，但是大部分的 JVM 在实现中会进行一些优化，使得两个栈帧出现一部分重叠。（主要体现在方法中有参数传递的情况），让下面栈帧的操作数栈和上面栈帧的部分局部变量重叠在一起，这样做不但节约了一部分空间，更加重要的是在进行方法调用时

就可以直接公用一部分数据，无需进行额外的参数复制传递了。



使用 JHSDb 工具查看栈空间一样可以看到。

```
* @author King老师
* 栈帧之间数据的共享
*/
public class JVMStack {

    public int work(int x) throws Exception{
        int z =(x+5)*10;//局部变量表有
        Thread.sleep(Integer.MAX_VALUE);
        return z;
    }

    public static void main(String[] args)throws Exception {
        JVMStack jvmStack = new JVMStack();
        jvmStack.work(x: 10);//10 放入main栈帧操作数栈
    }

}
```

内存溢出

栈溢出

参数: -Xss1m, 具体默认值需要查看官网: <https://docs.oracle.com/javase/8/docs/technotes/tools/unix/java.html#BABHDABI>

-Xsssize

Sets the thread stack size (in bytes). Append the letter `k` or `K` to indicate KB, `m` or `M` to indicate MB, `g` or `G` to indicate GB. The default value depends on the platform:

- Linux/ARM (32-bit): 320 KB
- Linux/i386 (32-bit): 320 KB
- Linux/x64 (64-bit): 1024 KB
- OS X (64-bit): 1024 KB
- Oracle Solaris/i386 (32-bit): 320 KB
- Oracle Solaris/x64 (64-bit): 1024 KB

The following examples set the thread stack size to 1024 KB in different units:

```
-Xss1m  
-Xss1024k  
-Xss1048576
```

This option is equivalent to `-XX:ThreadStackSize`.

HotSpot 版本中栈的大小是固定的，是不支持拓展的。

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

虚拟机栈带给我们的启示：方法的执行因为要打包成栈帧，所以天生要比实现同样功能的循环慢，所以树的遍历算法中：递归和非递归(循环来实现)都有

存在的意义。递归代码简洁，非递归代码复杂但是速度较快。

OutOfMemoryError: 不断建立线程，JVM 申请栈内存，机器没有足够的内存。（一般演示不出，演示出来机器也死了）

同时要注意，栈区的空间 JVM 没有办法去限制的，因为 JVM 在运行过程中会有线程不断的运行，没办法限制，所以只限制单个虚拟机栈的大小。

堆溢出

内存溢出：申请内存空间,超出最大堆内存空间。

如果是内存溢出，则通过 调大 `-Xms`，`-Xmx` 参数。

如果不是内存泄漏，就是说内存中的对象却是都是必须存活的，那么久应该检查 JVM 的堆参数设置，与机器的内存对比，看是否还有可以调整的空间，再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、存储结构设计不合理等情况，尽量减少程序运行时的内存消耗。

方法区溢出

(1) 运行时常量池溢出

(2) 方法区中保存的 **Class** 对象没有被及时回收掉或者 **Class** 信息占用的内存超过了我们配置。

注意 **Class** 要被回收，条件比较苛刻（仅仅是可以，不代表必然，因为还有一些参数可以进行控制）：

- 1、该类所有的实例都已经被回收，也就是堆中不存在该类的任何实例。
- 2、加载该类的 **ClassLoader** 已经被回收。
- 3、该类对应的 **java.lang.Class** 对象没有在任何地方被引用，无法在任何地方通过反射访问该类的方法。

-Xnocomclassgc

禁用类的垃圾收集（GC）。这样可以节省一些GC时间，从而缩短了应用程序运行期间的中断时间。

当您 **-Xnocomclassgc** 在启动时指定时，应用程序中的类对象在GC期间将保持不变，并且始终被认为是活动的。这可能会导致更多的内存被永久占用，如果不谨慎使用，将抛出内存不足异常。

代码示例：

cglib 是一个强大的，高性能，高质量的 Code 生成类库，它可以在运行期扩展 Java 类与实现 Java 接口。

CGLIB 包的底层是通过使用一个小而快的字节码处理框架 ASM，来转换字节码并生成新的类。除了 CGLIB 包，脚本语言例如 Groovy 和 BeanShell，也是使用 ASM 来生成 java 的字节码。当然不鼓励直接使用 ASM，因为它要求你必须对 JVM 内部结构包括 class 文件的格式和指令集都很熟悉。

本机直接内存溢出

直接内存的容量可以通过 MaxDirectMemorySize 来设置（默认与堆内存最大值一样），所以也会出现 OOM 异常；

由直接内存导致的内存溢出，一个比较明显的特征是在 HeapDump 文件中不会看见有什么明显的异常情况，如果发生了 OOM，同时 Dump 文件很小，可以考虑重点排查下直接内存方面的原因。

常量池

Class 常量池(静态常量池)

在 class 文件中除了有类的版本、字段、方法和接口等描述信息外，还有一项信息是常量池 (Constant Pool Table)，用于存放编译期间生成的各种字面量和符号引用。

```

major version: 52
flags: ACC_PUBLIC, ACC_SUPER
Constant pool:
 #1 = Methodref      #5.#29      // java/lang/Object."<init>":()V
 #2 = Class           #30         // com/jvm/ch02/Person
 #3 = Methodref      #2.#29      // com/jvm/ch02/Person."<init>":()V
 #4 = Methodref      #2.#31      // com/jvm/ch02/Person.work:()I
 #5 = Class           #32         // java/lang/Object
 #6 = Methodref      #5.#33      // java/lang/Object.hashCode:()I
 #7 = Utf8            <init>
 #8 = Utf8            ()V
 #9 = Utf8            Code
#10 = Utf8           LineNumberTable
#11 = Utf8            LocalVariableTable
#12 = Utf8            this
#13 = Utf8            Lcom/jvm/ch02/Person;
#14 = Utf8            work
#15 = Utf8            ()I
#16 = Utf8            x
#17 = Utf8            I
#18 = Utf8            y
#19 = Utf8            z
#20 = Utf8            main
#21 = Utf8            ([Ljava/lang/String;)V
#22 = Utf8            args
#23 = Utf8            [Ljava/lang/String;
#24 = Utf8            person
#25 = Utf8            object
#26 = Utf8            Ljava/lang/Object;
#27 = Utf8            SourceFile
#28 = Utf8            Person.java
#29 = NameAndType     #7:#8      // "<init>":()V
#30 = Utf8            com/jvm/ch02/Person
#31 = NameAndType     #14:#15     // work:()I
#32 = Utf8            java/lang/Object
#33 = NameAndType     #34:#15     // hashCode:()I
#34 = Utf8            hashCode

```

字面量：给基本类型变量赋值的方式就叫做字面量或者字面值。

比如：`String a="b"`，这里“b”就是字符串字面量，同样类推还有整数字面值、浮点类型字面量、字符字面量。

符号引用：符号引用以一组符号来描述所引用的目标。符号引用可以是任何形式的字面量，**JAVA** 在编译的时候一个每个 **java** 类都会被编译成一个 **class** 文件，但在编译的时候虚拟机并不知道所引用类的地址(实际地址)，就用符号引用来代替，而在类的解析阶段（后续 **JVM** 类加载会具体讲到）就是为了把这个符号引用转化成为真正的地址的阶段。

一个 **java** 类（假设为 **People** 类）被编译成一个 **class** 文件时，如果 **People** 类引用了 **Tool** 类，但是在编译时 **People** 类并不知道引用类的实际内存地址，因此只能使用符号引用（`org.simple.Tool`）来代替。而在类装载机装载 **People** 类时，此时可以通过虚拟机获取 **Tool** 类的实际内存地址，因此便可以既将符号 `org.simple.Tool` 替换为 **Tool** 类的实际内存地址。

运行时常量池

运行时常量池（**Runtime Constant Pool**）是每一个类或接口的常量池（**Constant_Pool**）的运行时表示形式，它包括了若干种不同的常量：从编译期可知的数值字面量到必须运行期解析后才能获得的方法或字段引用。（这个是虚拟机规范中的描述，很生涩）

运行时常量池是在类加载完成之后，将 **Class** 常量池中的符号引用值转存到运行时常量池中，类在解析之后，将符号引用替换成直接引用。运行时常量池在 **JDK1.7** 版本之后，就移到堆内存中了，这里指的是物理空间，而逻辑上还是属于方法区（方法区是逻辑分区）。

在 **JDK1.8** 中，使用元空间代替永久代来实现方法区，但是方法区并没有改变，所谓“*Your father will always be your father*”。变动的只是方法区中内容的物理存放位置，但是运行时常量池和字符串常量池被移动到了堆中。但是不论它们物理上如何存放，逻辑上还是属于方法区的。

字符串常量池

字符串常量池这个概念是最有争议的，**King** 老师翻阅了虚拟机规范等很多正式文档，发现没有这个概念的官方定义，所以与运行时常量池的关系不去抬杠，我们从它的作用和 **JVM** 设计它用于解决什么问题的点来分析它。

以 **JDK1.8** 为例，字符串常量池是存放在堆中，并且与 `java.lang.String` 类有很大关系。设计这块内存区域的原因在于：**String** 对象作为 **Java** 语言中重

要的数据类型，是内存中占据空间最大的一个对象。高效地使用字符串，可以提升系统的整体性能。

所以要彻底弄懂，我们的重心其实在于深入理解 String。

String

String 类分析（JDK1.8）

String 对象是对 char 数组进行了封装实现的对象，主要有 2 个成员变量：char 数组，hash 值。

```
public final class String
    implements java.io.Serializable, Comparable<String>, CharSequence {
    /** The value is used for character storage. */
    private final char value[];

    /** Cache the hash code for the string */
    private int hash; // Default to 0

    /** use serialVersionUID from JDK 1.0.2 for interoperability */
```

String 对象的不可变性

了解了 String 对象的实现后，你有没有发现在实现代码中 String 类被 final 关键字修饰了，而且变量 char 数组也被 final 修饰了。

我们知道类被 final 修饰代表该类不可继承，而 char[] 被 final+private 修饰，代表了 String 对象不可被更改。Java 实现的这个特性叫作 String 对象的不可变性，即 String 对象一旦创建成功，就不能再对它进行改变。

Java 这样做的好处在哪里呢？

第一， 保证 `String` 对象的安全性。假设 `String` 对象是可变的，那么 `String` 对象将可能被恶意修改。

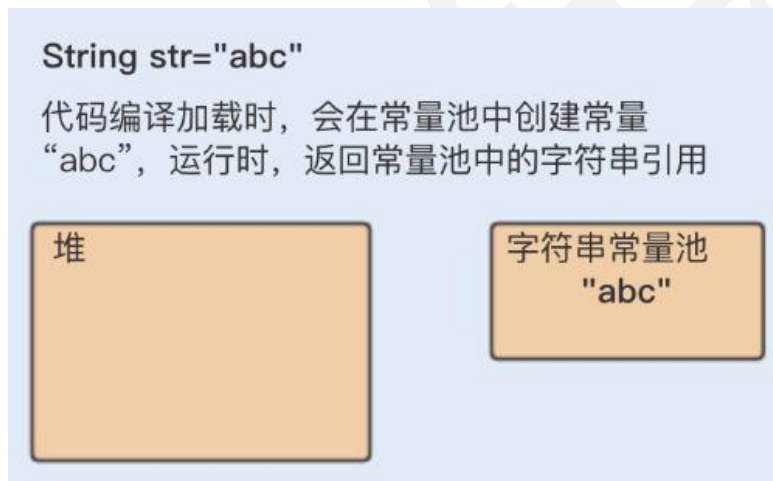
第二， 保证 `hash` 属性值不会频繁变更，确保了唯一性，使得类似 `HashMap` 容器才能实现相应的 `key-value` 缓存功能。

第三， 可以实现字符串常量池。在 `Java` 中，通常有两种创建字符串对象的方式，一种是通过字符串常量的方式创建，如 `String str= "abc"`；另一种是字符串变量通过 `new` 形式的创建，如 `String str = new String("abc")`。

String 的创建方式及内存分配的方式

1、String str= "abc"；

当代码中使用这种方式创建字符串对象时，JVM 首先会检查该对象是否在字符串常量池中，如果在，就返回该对象引用，否则新的字符串将在常量池中被创建。这种方式可以减少同一个值的字符串对象的重复创建，节约内存。（`str` 只是一个引用）

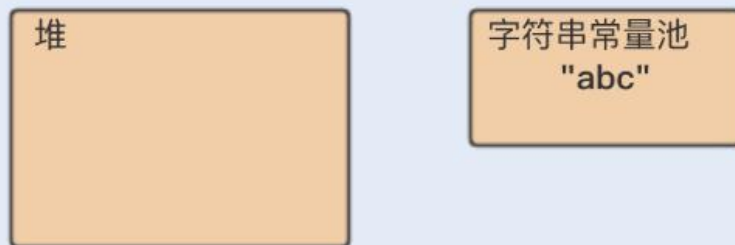


2、String str = new String("abc")

首先在编译类文件时，"abc"常量字符串将会放入到常量结构中，在类加载时，“abc”将会在常量池中创建；其次，在调用 `new` 时，JVM 命令将会调用 `String` 的构造函数，同时引用常量池中的"abc" 字符串，在堆内存中创建一个 `String` 对象；最后，`str` 将引用 `String` 对象。

```
String str = new String("abc")
```

1) 代码编译加载时，会在常量池中创建常量“abc”



2) 在调用new时，会在堆中创建String对象，并引用常量池中的字符串对象char[]数组。并返回String对象引用。



4、使用 new，对象会创建在堆中，同时赋值的话，会在常量池中创建一个字符串对象，复制到堆中。

具体的复制过程是先将常量池中的字符串压入栈中，在使用 String 的构造方法是，会拿到栈中的字符串作为构造方法的参数。

这个构造函数是一个 char 数组的赋值过程，而不是 new 出来的，所以是引用了常量池中的字符串对象。存在引用关系。

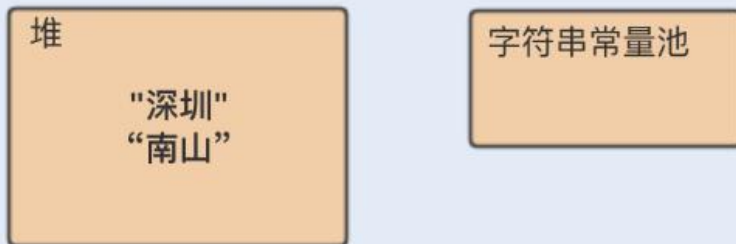
```
public class Location {  
    private String city;
```



```
private String region;  
}  
  
}  
public void mode3(){  
    Location location = new Location();  
    location.setCity("深圳");  
    location.setRegion("南山");  
}
```

```
public class Location {  
    private String city;  
    private String region;  
}
```

在运行时，创建的String对象会在堆中直接创建，不会再常量池中创建



4、String str2= "ab"+ "cd"+ "ef";

编程过程中，字符串的拼接很常见。前面我讲过 String 对象是不可变的，如果我们使用 String 对象相加，拼接我们想要的字符串，是不是就会产生多个

对象呢？例如以下代码：

分析代码可知：首先会生成 `ab` 对象，再生成 `abcd` 对象，最后生成 `abcdef` 对象，从理论上来说，这段代码是低效的。

编译器自动优化了这行代码，编译后的代码，你会发现编译器自动优化了这行代码，如下

`String str= "abcdef";`

5、大循环使用+

```
}  
public void mode5(){  
    String str = "abcdef";  
    for(int i=0; i<1000; i++) {  
        str = str + i;  
    }  
    //编译器同样对这段代码进行了优化。Java 在进行字符串的拼接时，偏向使用 StringBuilder，这样可以提高程序的效率  
    //    String str = "abcdef";  
    //  
    //    for(int i=0; i<1000; i++) {  
    //        str = (new StringBuilder(String.valueOf(str))).append(i).toString();  
    //    }  
}
```

intern

`String` 的 `intern` 方法，如果常量池中有相同值，就会重复使用该对象，返回对象引用。

```
String a = new String( original: "king").intern();
String b = new String( original: "king").intern();
if(a==b) {
    System.out.print("a==b");
}else{
    System.out.print("a!=b");
}
```

- 1、new Sting() 会在堆内存中创建一个 a 的 String 对象，king"将会在常量池中创建
 - 2、在调用 intern 方法之后，会去常量池中查找是否有等于该字符串对象的引用，有就返回引用。
 - 3、调用 new Sting() 会在堆内存中创建一个 b 的 String 对象。
 - 4、在调用 intern 方法之后，会去常量池中查找是否有等于该字符串对象的引用，有就返回引用。
- 所以 a 和 b 引用的是同一个对象。