Table of Contents

深入学习java虚拟机

在学校学了编译原理！所以一直在想Java是如何实现的，之前看过这本书，知道java实现了一种类似汇编语言的Java虚拟机中的jvm编码！

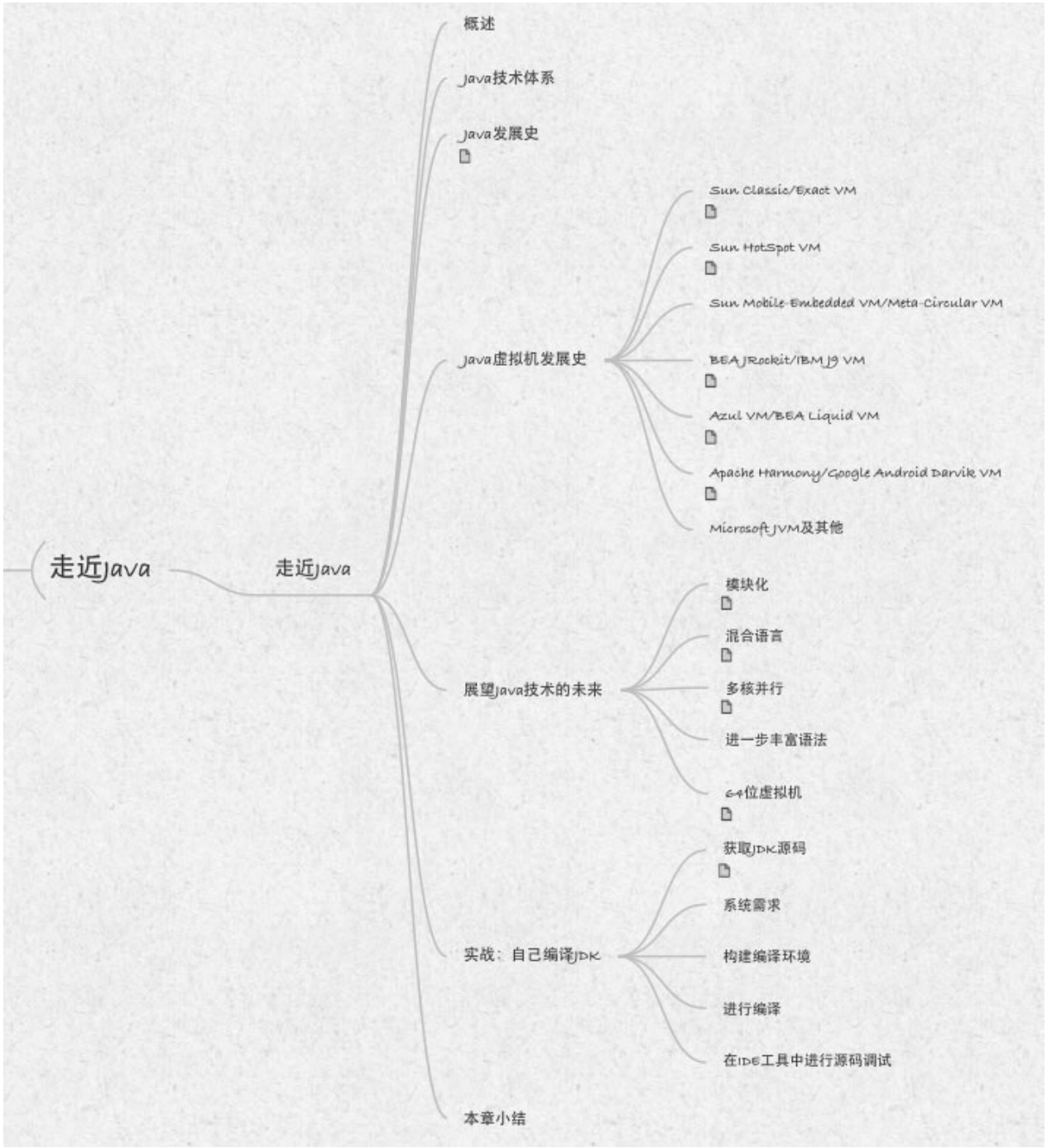
关于我

项目	信息
英文名	Gavin
邮箱	zlcgavin@gmail.com
个人说明	孤独的开发者，现在忙着在互联网公司踩坑

走进Java

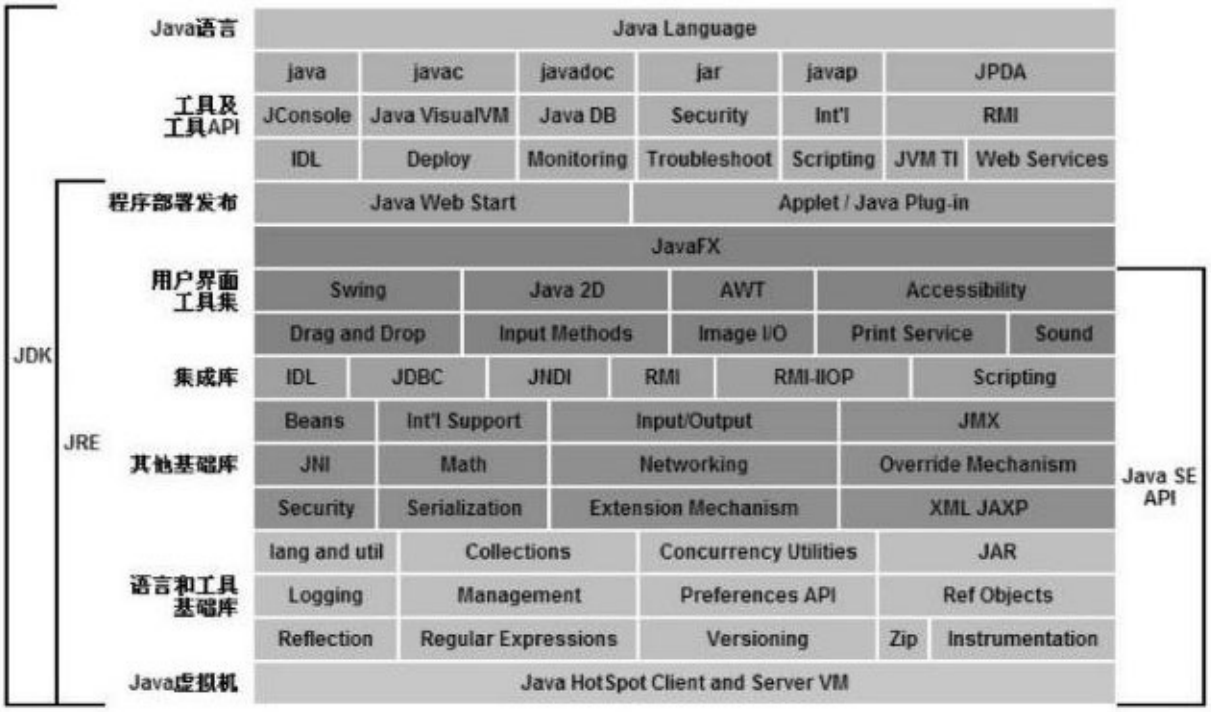
进一步的了解Java的历史和技术发展方向，可以更好地理解技术新特性的作用，从而找到进一步学习的方向感，嗯，也就有了技术安全感。😀

以下是整体的思维导图：



概述

Java技术体系



Java发展史

HotSpot 虚拟机：JDK 1.3及之后所有版本的Sun JDK的默认虚拟机JDK。

1.4同样发布了很多新的技术特性，如正则表达式、异常链、NIO、日志类、XML 解析器和 XSLT 转换器等。

JDK1.5 在Java语法易用性上做出了非常大的改进。例如，自动装箱、泛型、动态注解、枚举、可变长参数、遍历循环（ foreach 循环）等语法特性都是在JDK1.5 中加入的。

JDK 1.6 的改进包括：提供动态语言支持（通过内置 Mozilla JavaScript Rhino 引擎实现）、提供编译 API 和微型 HTTP 服务器 API 等。同时，这个版本对 Java 虚拟机内部做了大量改进，包括锁与同步、垃圾收集、类加载等方面的算法都有相当多的改动。

“ B 计划”把不能按时完成的 Lambda 项目、 Jigsaw 项目和 Coin 项目的部分改进延迟到 JDK 1.8 之中。最终， JDK 1.7 的主要改进包括：提供新的 G1 收集器（ G1 在发布时依然处于 Experimental 状态，直至 2012 年 4 月的 Update 4 中才正式“转正”）、加强对非 Java 语言的调用支持（ JSR-292， 这项特性到目前为止依然没有完全实现定型）、升级类加载架构等。

Java虚拟机发展史

Sun Classic/Exact VM

世界上第一款商用 Java 虚拟机

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 JIT 编译器，就必须进行外挂。但是假如外挂了 JIT 编译器， JIT 编译器就完全接管了虚拟机的执行系统，解释器便不再工作了。

```
java version"1.2.2"
Classic VM（build JDK-1.2.2-001，green threads, sunwjit）
```

其中的"sunwjit" 就是 Sun 提供的外挂编译器，其他类似的外挂编译器还有 Symantec JIT和 shuJIT等。

Exact VM在商业应用上只存在了很短暂的时间就被更为优秀的HotSpot VM所取代

Class VM在JDK 1.2 之前是 Sun JDK 中唯一的虚拟机，在 JDK 1.2 时，它与 HotSpot VM 并存，但默认使用的是 Classic VM。

Sun HotSpot VM

HotSpot VM 既继承了 Sun 之前两款商用虚拟机的优点(准确式内存管理)

HotSpot VM 的热点代码探测能力可以通过执行计数器找出最具有编译价值的代码，然后通知 JIT 编译器以方法为单位进行编译。

在 2008 年和 2009 年， Oracle 公司分别收购了 BEA 公司和 Sun 公司，这样 Oracle 就同时拥有了两款优秀的 Java 虚拟机： JRockit VM 和 HotSpot VM。

Sun Mobile-Embedded VM/Meta-Circular VM

BEA JRockit/IBM J9 VM

由于专注于服务器端应用，它可以不太关注程序启动速度，因此 JRockit 内部不包含解析器实现，全部代码都靠即时编译器编译后执行。除此之外， JRockit 的垃圾收集器和 MissionControl 服务套件等部分的实现，在众多 Java 虚拟机中也一直处于领先水平。

IBM J9 的市场定位与 Sun HotSpot 比较接近，它是一款设计上从服务器端到桌面应用再到嵌入式都全面考虑的多用途虚拟机。

Azul VM/BEA Liquid VM

Azul VM 和 BEA Liquid VM 这类特定硬件平台专有的虚拟机才是“高性能”的武器。

Liquid VM 不需要操作系统的支持，或者说它自己本身实现了一个专用操作系统的必要功能，如文件系统、网络支持等。由虚拟机越过通用操作系统直接控制硬件可以获得很多好处，如在线程调度时，不需要再进行内核态/用户态的切换等，这样可以最大限度-地发挥硬件的能力，提升 Java 程序的执行性能。

Apache Harmony/Google Android Darvik VM

Apache Harmony）被吸纳进 IBM 的 JDK 7 实现及 Google Android SDK 之中，尤其是对 Android 的发展起到了很大的推动作用。

Dalvik VM 是 Android 平台的核心组成部分之一，它的名字来源于冰岛一个名为 Dalvik 的小渔村。 Dalvik VM 并不是一个 Java 虚拟机，它没有遵循 Java 虚拟机规范，不能直接执行 Java 的 Class 文件，使用的是寄存器架构而不是 JVM 中常见的栈架构。

Microsoft JVM及其他

展望Java技术的未来

模块化

模块化是建立各种功能的标准件的前提。最近几年 OSGi 技术的迅速发展、各个厂商在 JCP 中对模块化规范的激烈斗争[1]， 都能充分说明模块化技术的迫切和重要。

OSGi 已经发布到 R5.0 版本，而 Jigsaw 从 Java 7 延迟至 Java 8，在 2012 年 7 月又不得不宣布推迟到 Java 9 中发布。

《深入理解 OSGi: Equinox 原理、应用与最佳实践》

混合语言

各种语言之间的交互不存在任何困难，就像使用自己语言的原生 API 一样方便[1]， 因为它们最终都运行在一个虚拟机之上。

在同一个虚拟机上运行的其他语言与 Java 之间的交互一般都比较容易，但非 Java 语言之间的交互一般都比较烦琐。

多核并行

早在 JDK 1.5 就已经引入 java.util.concurrent 包实现了一个粗粒度的并发框架。而 JDK 1.7 中加入的 java.util.concurrent.forkjoin 包则是对这个框架的一次重要扩充。

通过利用 Fork/Join 模式，我们能够更加顺畅地过渡到多核时代。

函数式编程的一个重要优点就是这样的程序天然地适合并行运行，这对 Java 语言在多核时代继续保持主流语言的地位有很大帮助。

Sumatra 项目就是为 Java 提供使用 GPU（ Graphics Processing Units） 和 APU（ Accelerated Processing Units） 运算能力的工具。

在 JDK 外围，也出现了专为满足并行计算需求的计算框架，如 Apache 的 Hadoop Map/Reduce。

进一步丰富语法

64位虚拟机

由于指针膨胀和各种数据类型对齐补白的原因，运行于 64 位系统上的 Java 应用需要消耗更多的内存。

在 JDK 1.6 Update 14 之后，提供了普通对象指针压缩功能（- XX:+ UseCompressedOops，这个参数不建议显式设置，建议维持默认由虚拟机的 Ergonomics 机制自动开启）

实战：自己编译JDK

获取JDK源码

OpenJDK 7 和 Oracle JDK 7 在程序上是非常接近的，两者共用了大量相同的代码，所以我们编译的 OpenJDK， 基本上可以认为性能、功能和执行逻辑上都和官方的 Oracle JDK 是一致的。

系统需求

构建编译环境

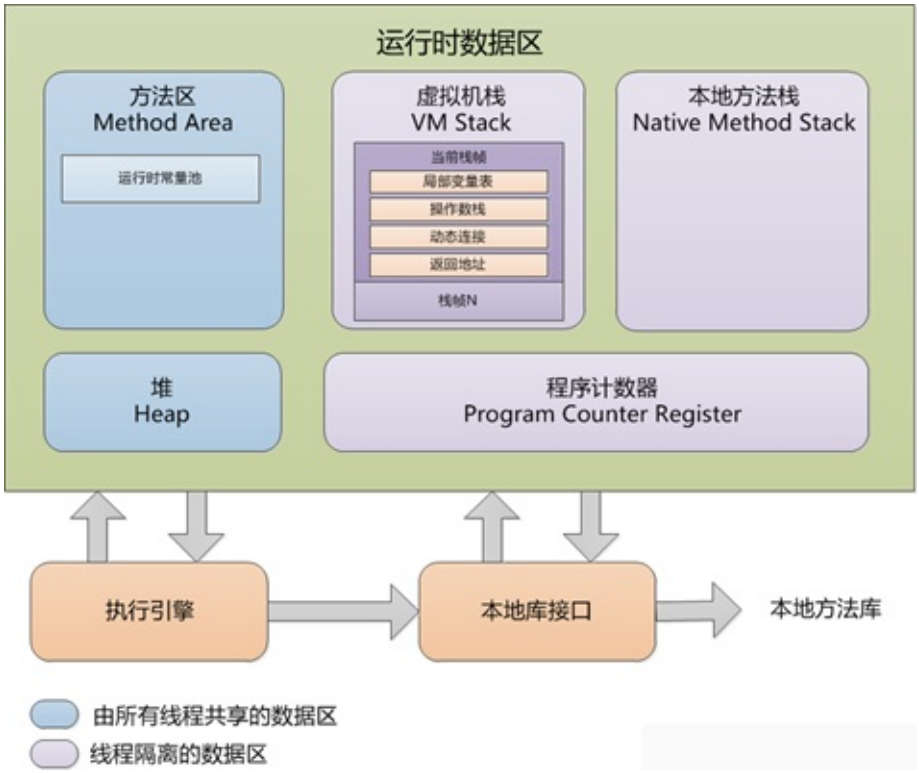
进行编译

在IDE工具中进行源码调试

Java内存区域与内存溢出异常

运行时数据区域

我们看一下Java虚拟机运行时数据区：



程序计数器

程序计数器：是一块较小的内存空间，它可以看作是当前线程所执行的字节码的行号指示器。每个线程都有自己的独立的程序计数器。

如果线程正在执行的是Java方法，那么这个计数器的值就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。

Java虚拟机栈

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：每个方法执行的同时都会创建一个栈帧用于存储局部变量表、操作数栈、动态链接、方法出口等信息。

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），其余的数据类型占1个。局部变量表所需的内存空间在编译期间分配完成，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。

如果线程请求栈的深度大于虚拟机所允许的深度，将抛出StackOverflowError异常；无法申请到内存抛出OutOfMemoryError异常。

本地方法栈

本地方法栈与虚拟机栈所发挥的作用是非常相似的，它们之间的区别不过是虚拟机栈为虚拟机执行java方法，而本地栈则为虚拟机使用到的Native方法服务。

Java堆

Java堆是线程共享的，在虚拟机启动时创建。此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。

Java堆是垃圾收集器管理的主要区域，因此很多时候也被称作“GC堆”。由于现在收集器基本都采用分代收集算法，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。

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）。

方法区（永久代）

线程共享，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。

这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！

运行时常量池

他是方法区的一部分，Class文件中除了有类的版本、字段、方法、接口等描述信息外，还有一项信息就是常量池，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进入方法区的运行时常量池中存放。

直接内存

直接内存不是虚拟机运行时数据区的一部分。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。

在JDK1.4中新加入了NIO类，引入了一种基于通道与缓存区（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堆中的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。

HotSpot虚拟机对象探秘

对象的创建

虚拟机遇到一个new指令时，首先去检查这个指令的参数是否能在常量池中定位到一个类的符号引用，并且检查这个符号引用代表的类是否已经被加载、解析和初始化过。如果没有，那必须先执行相应的类加载过程。接下来JVM将为新生对象分配内存。

如果Java堆是规整的（所有用过的内存在一边，未使用的在另一边，维护麻烦），那么将使用“指针碰撞”的分配方式，否则使用“空闲列表”的分配方式。Java堆是否规整由采用的垃圾收集器是否带有压缩整理功能决定。

但是内存的分配是同步的，如果一个线程刚分配一个对象内存，但是还没有修改指针所指向的位置，那么另一个线程分配对象的时候可能就出错了。解决方法有两个，一是对分配内存空间的动作进行同步处理（**CAS方式**）。另一种是把内存分配的动作按照线程划分在不同的空间进行，每个线程在**java**堆中预分配一小块内存，称为本地线程分配缓冲（**TLAB**）。只有**TLAB**用完并分配新的**TLAB**时，才需要同步。**JVM**是否开启**TLAB**功能，可通过**-XX:+/-UseTLAB**参数来设定。

内存分配完之后，初始化零值（不包括对象头），如果使用TLAB，这一工作过程也可以提前至TLAB分配时进行。

接下来，**JVM**对对象进行必要的设置，例如这个对象是哪个类的实例、如何才能找到类的元数据信息、对象的哈希码、对象的**GC**分代年龄等信息。这些信息存放在对象的对象头中，根据JVM当前运行状态不同，如是否启用偏向锁等，对象头会有不同的设置方式。

执行完new指令后接着执行方法，把对象按照程序员的意愿进行初始化，这样一个对象就初始化完成了。

对象的内存布局

在HotSpot虚拟机中，对象在内存中存储的布局可以分为3块区域：对象头、实例数据和对齐填充。

HotSpot虚拟机的对象头包括两部分信息，第一部分用于存储对象自身的运行时数据（哈希码、GC分代年龄、锁状态标志、线程持有的锁、偏向线程ID、偏向时间戳等，这部分数据的存储官方称为Mark Word），另一部分是类型指针（即对象指向它的类元数据的指针，JVM通过这个指针来确定这个对象是哪个类的实例）。

	HotSpot虚拟机对象头Mark Word	
存储内容	标志位	状态
对象哈希码、对象分代年龄	01	未锁定
指向锁记录的指针	00	轻量级锁定
指向重量级锁的指针	10	膨胀（重量级锁定）
空，不需要记录信息	11	GC标记
偏向线程ID、偏向时间戳、对象分代年龄	01	可偏向

如果对象是一个Java数组，那在对象头中还必须有一块用于记录数组长度的数据。

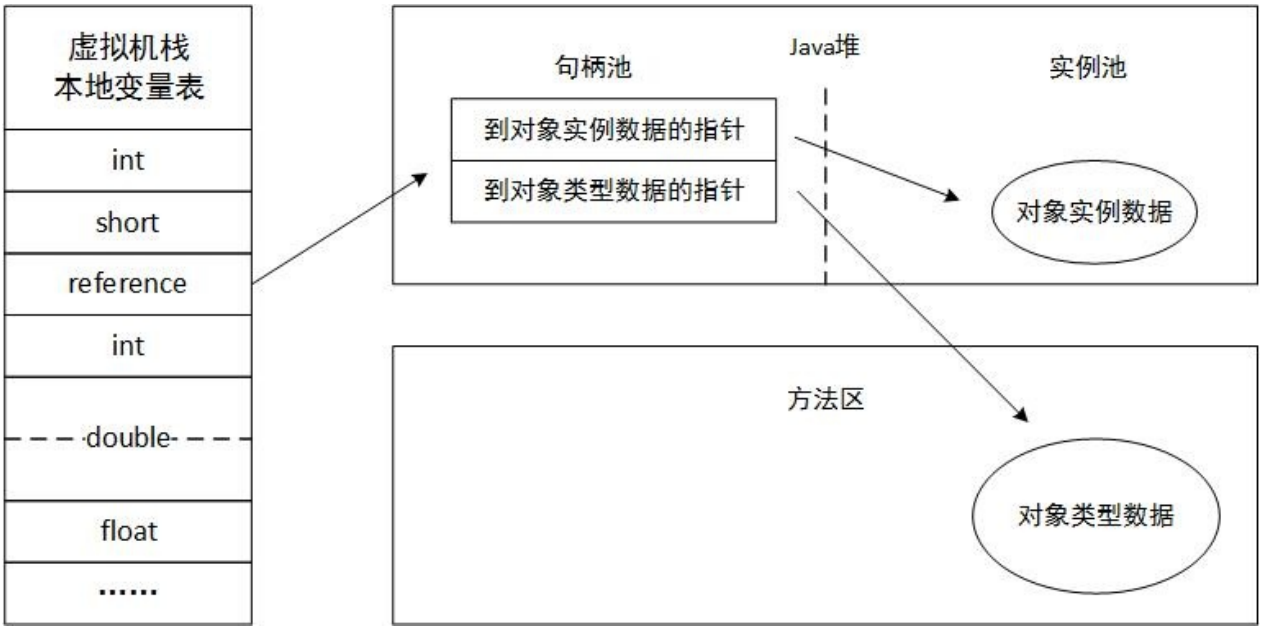
接下来的实例数据是对象真正存储的有效信息，也是在程序代码中所定义的各种类型的字段内容，在父类中定义的变量会出现在子类之前，如果CompactFields参数值为true，那么子类中较窄的变量也可能会插入到父类变量的空隙之中。

第三部分对齐填充并不是必然存在的，也没有特别的含义，它仅仅起着占位符的作用。不满8个字节的时候占位。

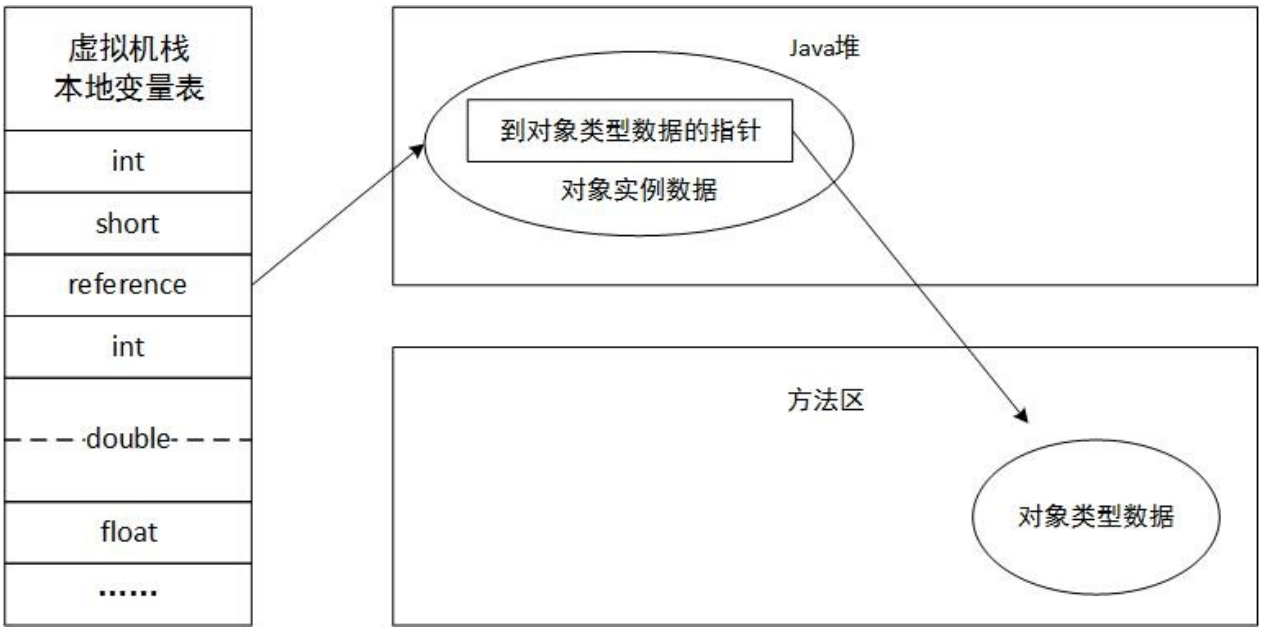
对象的访问定位

我们的Java程序需要通过栈上的Reference数据来操作堆上的具体对象。Reference访问对象的方式目前主流的有两种：句柄和直接指针。

- 如果直接使用句柄访问，java堆中将会划分出一块内存来作为句柄池，reference中存储的是对象的句柄地址，而句柄中包含了对象数据与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，那么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，而reference中存储的直接就是对象地址，如下图所示。



这两种对象访问方式各有优势，使用句柄来访问的最大好处是reference中存储的是稳定的句柄地址，在对象被移动时只会改变句柄中的实例数据指针，而reference本身不需要修改。

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。HotSpot虚拟机使用的是直接指针访问的方式。句柄来访问的情况也十分常见。

OutOfMemoryError异常

主要是为了学习之前学的各种内存区域的内容，还有就是以后遇到内存错误的时候，能够根据异常的信息快速判读是哪个区域的内存溢出，知道是什么样的代码可能会导致这些区域内存溢出，以及出现这些异常后，该如何处置。

Java堆溢出

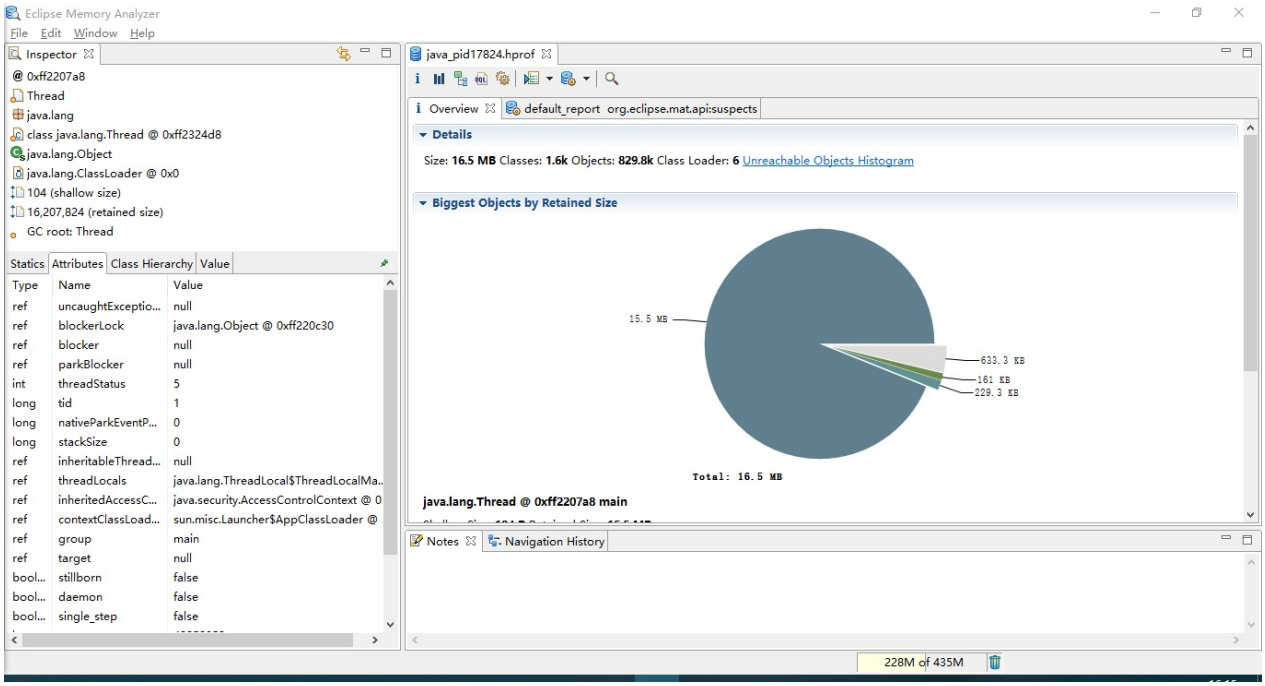
我们在JVM参数中加入：

```
-verbose:gc -Xms20M -Xmx20M -Xmn10M -XX:+PrintGCDetails -XX:SurvivorRatio=8
```

这里堆的大小固定为20M且不可扩展，也可以通过参数-XX:+HeapDumpOnOutOfMemoryError可以让虚拟机在出现内存溢出异常时，Dump当前的内存堆转储快照以便事后进行分析。如果你不想等到发生崩溃性的错误时才获得堆转储文件，也可以通过设置如下 JVM 参数来按需获取堆转储文件。-XX:+HeapDumpOnCtrlBreak

```
List<Object> list = new ArrayList<Object>();
for(int i=0; i< 100000000; i++) {
    HelloBean helloBean1 = new HelloBean();
    list.add(helloBean1);
}
```

这段代码运行会抛出内存溢出的问题，并产生java_pid17824.hprof内存镜像文件，然后通过分析工具打开这个文件查看（比如Eclipse自带的Eclipse Memory Analyzer，这是一个插件需要安装）。重点是确认内存中的对象是否是必要的，也就是要先分清楚到底是出现了内存泄漏还是内存溢出的问题。



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。于是就能找到泄露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。掌握了泄露对象的类型信息及GC Roots引用链的信息，就可以比较准确地定位出泄露代码的位置。

如果不存在泄露，换句话说，就是内存中的对象确实都还必须存活着，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。

https://www.ibm.com/developerworks/cn/opensource/os-cn-ecl-ma/

http://blog.csdn.net/aaa2832/article/details/19419679

虚拟机栈和本地方法栈溢出

由于HotSpot虚拟机中并不区分虚拟机栈和本地方法栈，因此，对于HotSpot来说，虽然-Xoss参数（设置本地方法栈大小）存在，但实际上是没有效果的，栈容量只由-Xss参数设置。关于虚拟机栈和本地方法栈，在Java虚拟机规范中描述了两种异常：

- 如果线程请求的栈深度大于虚拟机所允许的最大深度，将抛出StackOverflowError异常。
- 如果虚拟机在扩展栈时无法申请到足够的内存空间，将抛出OutOfMemoryError异常。

这两种异常其实存在着一些互相重叠的地方。实验结果表明：在单个线程下，无论是由于栈帧太大还是虚拟机栈容量太小，当内存无法分配的时候，虚拟机抛出的都是StackOverflowError异常。如果测试时不限于单线程，通过不断地建立线程的方式倒是可以产生内存溢出异常。

如果是建立过多线程导致内存溢出，在不能减少线程数或者更换64位虚拟机的情况下，就只能通过减少最大堆和减少栈容量来换取更多的线程。

方法区和运行时常量池溢出

由于运行时常量池是方法区的一部分，因此这两个区域的溢出测试就放在一起进行。前面提到JDK1.7开始逐步“去永久代”的事情，在此就以测试代码观察一下这件事对程序的实际影响。

String.intern()是一个Native方法，他的作用是：如果字符串常量池中已经包含一个等于此String常量的字符串，则返回代表池中这个字符串的String对象；否则，将此String对象包含的字符串添加到常量池中，并且返回此String对象的引用。在JDK1.6及之前的版本中，由于常量池分配在永久代内，我们可以通过-XX:PermSize和-XX:MaxPermSize限制方法区大小，从而间接限制其中的常量池的容量。

这意味着重复调用String.intern()在JDK1.6之前的版本中会抛出方法区（PermGen space）OutOfMemoryError,而在JDK1.7中，不会出现。

方法区用于存放Class的相关信息，如类名、访问修饰符、常量池、字段描述、方法描述等。对于这些区域的测试，基本的思路是运行时产生大量的类去填满方法区，直到溢出。虽然直接使用Java SE API也可以动态产生类（如反射时的GeneratedConstructorAccessor和动态代

理等），但在本次实验中操作起来比较麻烦。在下面的代码中，笔者借助CGLib直接操作字节码运行时生成了大量的动态类。

值得特别注意的是，我们在这个例子中模拟的场景并非纯粹是一个实验，这样的应用经常会出现在实际应用中：当前的很多主流框架，如Spring、Hibernate，在对类进行增强时，都会使用到CGLib这类字节码技术，增强的类越多，就需要越大的方法区来保证动态生成的Class可以加载入内存。另外，JVM上的动态语言（如Groovy）通常都会持续创建类来实现动态语言的动态性，随着这类语言的流行，这类溢出也会越来越多。

```
public class JavaMethodAreaOOM {
    public static void main(final String[] args) {
        while(true) {
            Enhancer enhancer = new Enhancer();
            enhancer.setSuperclass(OOMObject.class);
            enhancer.setUseCache(false);
            enhancer.setCallback(new MethodInterceptor() {
                @Override
                public Object intercept(Object o, Method method, Object[] objects, Met
hodProxy methodProxy) throws Throwable {
                    return methodProxy.invokeSuper(o, args);
                }
            });
            enhancer.create();
        }
    }

    static class OOMObject {
    }
}
```

方法区溢出也是一种常见的内存溢出异常，一个雷要被垃圾器回收掉，判定条件是比较严苛的。在经常动态产生大量Class的应用中，需要特别注意类的回收状况。这类场景除了上面提到的程序使用了CGLib字节码增强和动态语言之外，常见的还有：大量JSP或者动态产生JSP文件的应用、基于OSGi的应用等。

本机直接内存溢出

DirectMemory容量可以通过-XX:MaxDirectMemorySize指定，如果不指定，则默认与Java堆最大值（-Xmx指定）一样。代码清单越过了DirectByteBuffer类，直接通过反射获取Unsafe实例进行内存分配（Unsafe类的getUnsafe方法限制了只有引导类加载器才会返回实例，也就是设计者希望只有rt.jar中的类才能使用Unsafe的功能）。因为，虽然使用DirectByteBuffer分配内存也会抛出内存异常，但它抛出异常时并没有真正向操作系统申请内存分配，而是通过计算得知内存无法分配，于是手动抛出异常，真正申请分配内存的方法是unsafe.allocateMemory.

```
/**
 * VM Args: -Xmx20M -XX:MaxDirectMemorySize=10M
 */
public class DirectMemoryOOM {
    private static final int _1MB = 1024*1024;
    public static void main(String[] args) throws IllegalAccessException {
        Field unsafeField = Unsafe.class.getDeclaredFields()[0];
        unsafeField.setAccessible(true);
        Unsafe unsafe = (Unsafe) unsafeField.get(null);
        while(true) {
            unsafe.allocateMemory(_1MB);
        }
    }
}
```

由DirectMemory导致的内存溢出，一个明显的特征是在Heap Dump文件中不会看见明显的异常，如果读者发现OOM之后Dump文件很小，而程序中又直接或者间接使用了NIO，那就可以考虑检查一下是不是这方面的原因。

垃圾收集器与内存分配策略

概述

GC要完成3件事：

1. 哪些内存需要回收？
2. 什么时候回收？
3. 如何回收？

Java内存的程序计数器、虚拟机栈、本地方法栈3个区域随线程而生，随线程而灭；栈中的栈帧随着方法的进入和退出而有条不紊地执行着入栈和出栈操作。每一个栈帧中分配多少内存基本上是在类结构确定下来时就已知的，因此这几个区域的内存分配和回收都具备确定性，在这几个区域内就不需要过多考虑回收的问题，因为方法结束或者线程结束，内存自然就跟随着回收了。

而Java堆和方法区则不一样，一个接口中的多个实现类需要的内存可能不一样，一个方法中的多个实现类需要的内存可能不一样，一个方法中的多个分支需要的内存也可能不一样，只有在程序处于运行期间时才能知道会创建哪些对象，这部分内存的分配和回收是动态的，垃圾收集器所关注的是这部分的内存。

对象已死吗

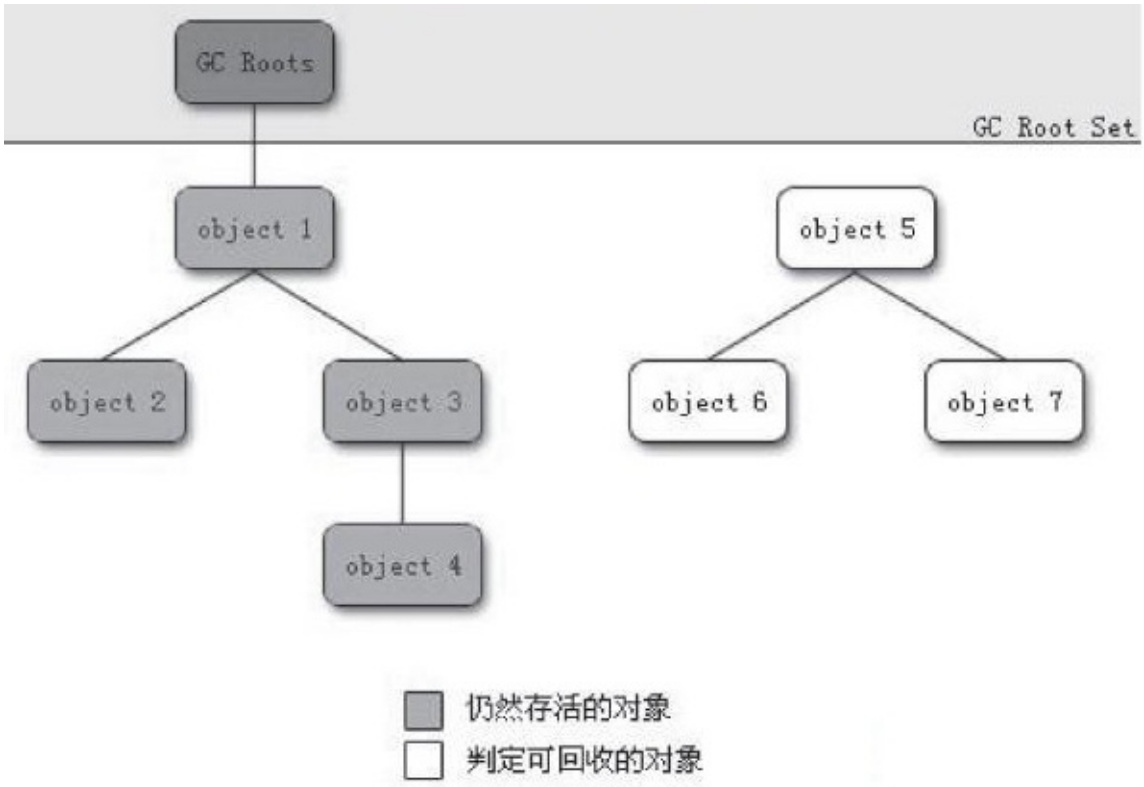
引用计数算法

至少主流的Java虚拟机里面没有选用计数算法来管理内存，其中主要原因是它很难解决对象之间互相循环引用的问题。

可达性分析算法

Java中使用可达性分析（Reachability Analysis）来判定对象是否存活的。

通过一系列的称为“GC Roots”的对象作为起始点，从这些节点开始向下搜索，搜索所走过的路径称为引用链（Reference Chain）,当一个对象到GC Roots没有任何引用链相连时，则证明此对象是不可用的。



在Java语言中，可作为GC Roots的对象包括下面几种：

- 虚拟机栈（栈帧中的本地变量表）中引用的对象。
- 方法区中类静态属性引用的对象。
- 方法区中常量引用的对象。
- 本地方法栈中JNI（即一般说的Native方法）引用的对象。

再谈引用

Java对引用的概念进行了扩充，将引用分为强引用（Strong Reference）、软引用（Soft Reference）、弱引用（Weak Reference）、虚引用（Phantom Reference）4种，这4种引用强度依次逐渐减弱。

- 强引用就是指在程序代码之中普遍存在的，类似“Object obj = new Object()”这类的引用，只要强引用还存在，垃圾收集器永远不会回收掉被引用的对象。

- 软引用是用来描述一些还有用但并非必需的对象。对于软引用关联着的对象，在系统将要发生内存溢出异常之前，将会把这些对象列进回收范围之中进行第二次回收。如果这次回收还没有足够的内存，才会抛出内存溢出异常。在JDK 1.2之后，提供了SoftReference类来实现软引用。

- 弱引用也是用来描述非必需对象的，但是它的强度比软引用更弱一些，被弱引用关联的对象只能生存到下一次垃圾收集发生之前。当垃圾收集器工作时，无论当前内存是否足够，都会回收掉只被弱引用关联的对象。在JDK 1.2之后，提供了WeakReference类来实现弱引用。

- 虚引用也称为幽灵引用或者幻影引用，它是最弱的一种引用关系。一个对象是否有虚引用的存在，完全不会对其生存时间构成影响，也无法通过虚引用来取得一个对象实例。为一个对象设置虚引用关联的唯一目的就是能在这个对象被收集器回收时收到一个系统通知。在JDK 1.2之后，提供了PhantomReference类来实现虚引用。

生存还是死亡

即使在可达性分析算法中不可达的对象，也并非是“非死不可”的，这时候它们暂时处于“缓刑”阶段，要真正宣告一个对象死亡，至少要经历两次标记过程：如果对象在进行可达性分析后发现没有与GC Roots相连接，那么它将会被第一次标记且进行一次刷选，刷选的条件是此对象是否有必要执行finalize方法。当对象没有覆盖finalize方法，或者finalize方法已经被虚拟机调用过，虚拟机将这两种情况都视为“没有必要执行”。

被判定有必要执行finalize方法的对象将被放置与F-Queue的队列中。并在稍后由一个虚拟机自动建立的、低优先级的Finalize线程去执行它。这个执行并不会等待其运行结束，防止阻塞和崩溃。finalize方法是对象逃过死亡命运的最后一次机会，稍后GC将对F-Queue中的对象进行第二次小规模的标记，如果对象要在finalize方法中拯救自己---只要重新与引用链上的任何一个对象建立关联即可。但是一个对象的finalize方法只能被执行一次。

回收方法区

方法区一般可以不回收，回收效率很低。在堆中，新生代的垃圾收集效率70%-90%，而永久代的垃圾回收效率远低于此。

永久代的垃圾回收主要回收两部分内容：废弃常量和无用的类。“废弃常量”判断比较简单，但是“无用的类”的判断复杂一些，需要满足下面3个条件：

- 该类所有的实例都已经被回收，也就是java堆中不存在该类的任何实例。
- 加载该类的ClassLoader已经被回收
- 该类对应的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。

是否对类进行回收，HotSpot虚拟机提供了-Xnoclassgc参数进行控制，还可以使用-verbose:class以及-XX:+TraceClassLoading, -XX:+TraceClassUnLoading查看类架子啊和卸载信息，其中-verbose:class和-XX:+TraceClassLoading可以在Product版的虚拟机中使用，-XX:+TraceClassUnLoading参数需要FastDebug版的虚拟机支持。

在大量使用反射、动态代理、CGLib等ByteCode框架、动态生成JSP以及OSGi这类频繁自定义ClassLoader的场景都需要虚拟机具备类卸载的功能，以保证永久代不会溢出。

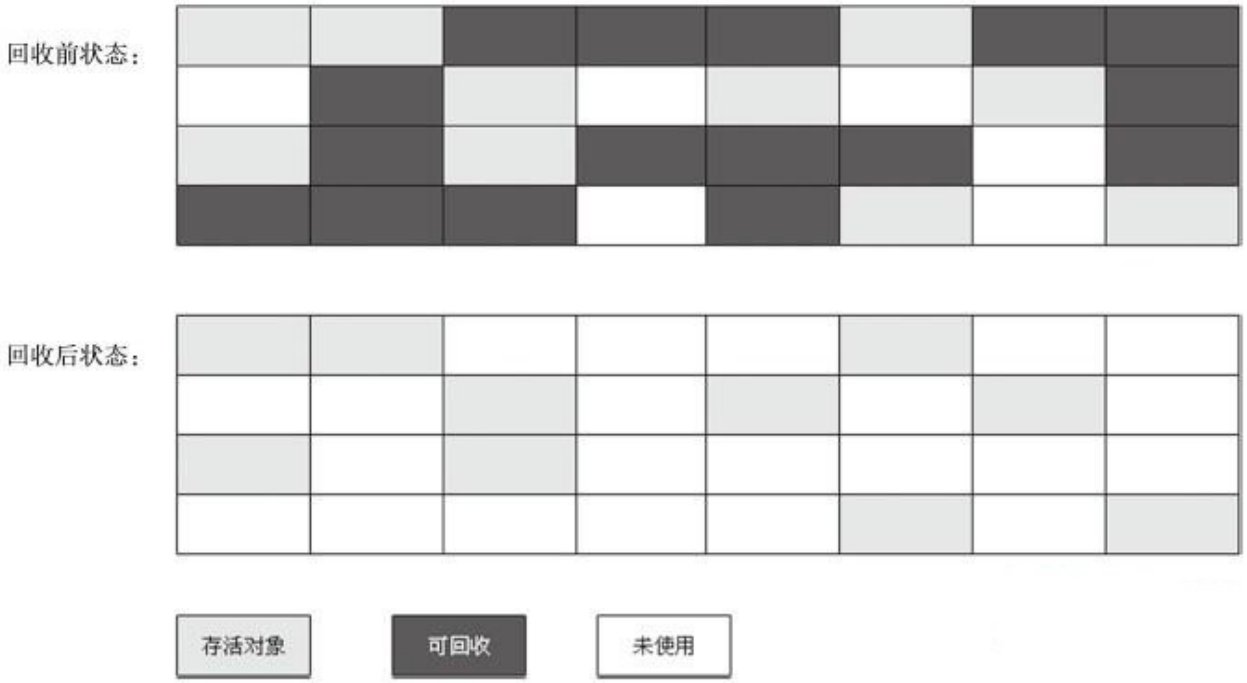
垃圾收集算法

标记-清除算法

算法分为标记和清除两个阶段：首先标记出所有需要回收的对象，在标记完成后统一回收所有被标记的对象，它的标记过程就是使用可达性算法进行标记的。

主要缺点有两个：

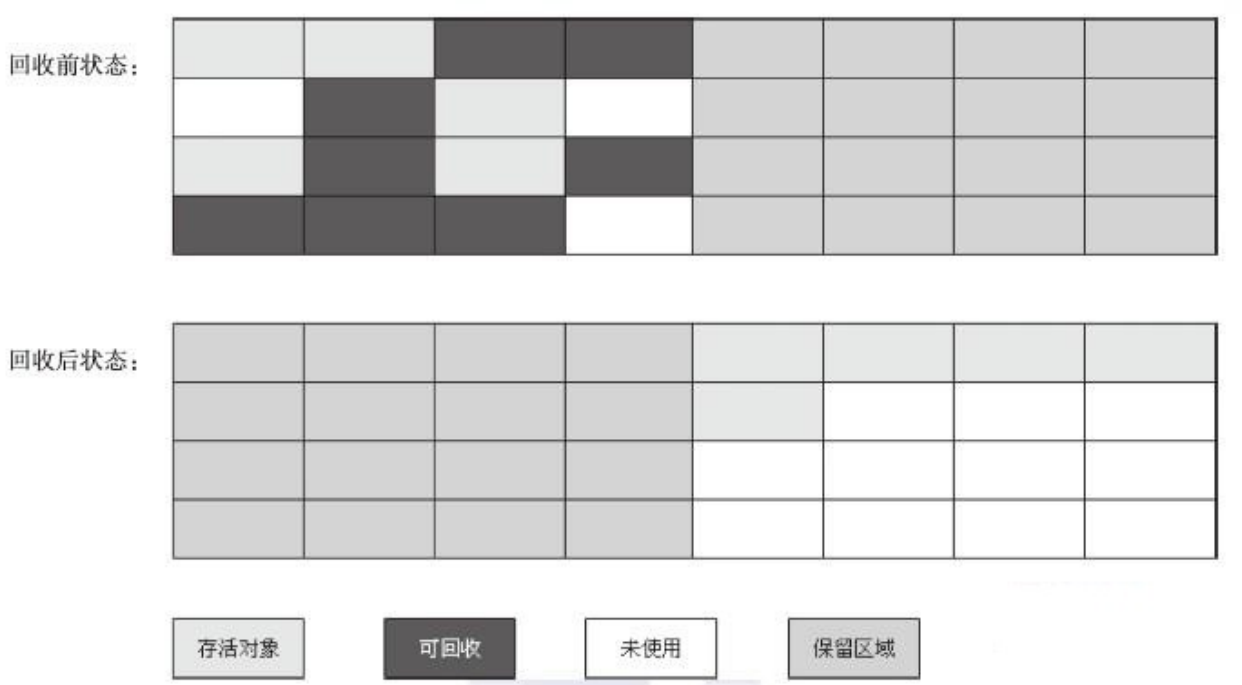
- 效率问题，标记和清除两个过程的效率都不高
- 空间问题，标记清除之后会产生大量不连续的内存碎片



复制算法

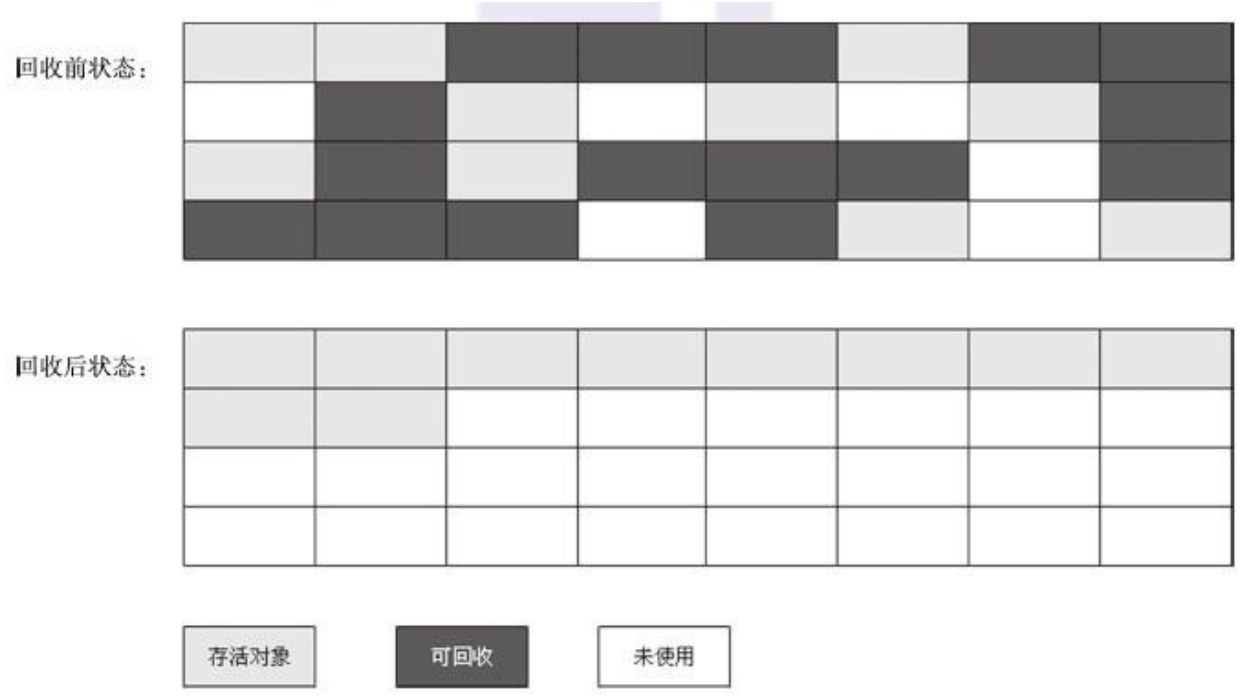
复制算法：将可用内存按照容量分为大小相等的两块，每次只使用其中的一块。当这一块的内存用完了，就将还存活着的对象复制到另一块上面，然后把已使用过的内存空间一次清理掉。

内存分配时不用考虑内存碎片问题，只要一动堆顶指针，按顺序分配内存即可，实现简单，运行高效。代价是将内存缩小为原来的一半。



标记-整理算法

标记整理算法（Mark-Compact），标记过程仍然和“标记-清除”一样，但后续不走不是直接对可回收对象进行清理，而是让所有存活对象向一端移动，然后直接清理掉端边界以外的内存。



分代收集算法

根据对象存活周期的不同将内存分为几块。一般把Java堆分为新生代和老年代，根据各个年代的特点采用最合适的收集算法。在新生代中，每次垃圾收集时有大批对象死去，只有少量存活，可以选用复制算法。而老年代对象存活率高，使用标记清理或者标记整理算法。

HotSpot的算法实现

枚举根节点

GC进行时必须停顿所有Java执行线程（Sun将之称为“Stop The World”）。即使是在号称（几乎）不会发生停顿的CMS收集器中，枚举根节点时也是必须要停顿的。

在HotSpot的实现中，是使用一组称为OopMap的数据结构来达到找到引用对象这个目的的。

安全点

实际上，HotSpot没有为每条指令都生成OopMap，前面已经提到，只是在“特定的位置”记录了这些信息，这些位置称为安全点（Safepoint），即程序执行时并非在所有地方都能停顿下来开始GC，只有在到达安全点时才能暂停。Safepoint的选定既不能太少以至于让GC等待时间太长，也不能过于频繁以致于过分增大运行时的负荷。所以，安全点的选定基本上是以程序“是否具有让程序长时间执行的特征”为标准选定的----因为每条指令执行的时间都非常短暂，程序不太可能因为指令流长度太长这个原因而过长时间运行，“长时间执行”的最明显特征就是指令序列复用，例如方法调用、循环跳转、异常跳转等，所以具有这些功能的指令才会产生Safepoint。

由于GC时，需要所有线程在安全点中断，一种是抢占式中断；另一种是主动式中断，其中抢占式中断就是在GC发生时，首先把所有线程全部中断，如果发现有线程不在安全点，就恢复线程，让它跑到安全点上。现在几乎没有JVM采用这种方式来响应GC事件。 而主动式中断的思想不是直接对线程操作，仅仅是简单设置一个标志，各个线程执行时主动去轮询这个标志，发现中断标志为真时就自己中断挂起。轮询标志的地方和安全点是重合的。

安全区域

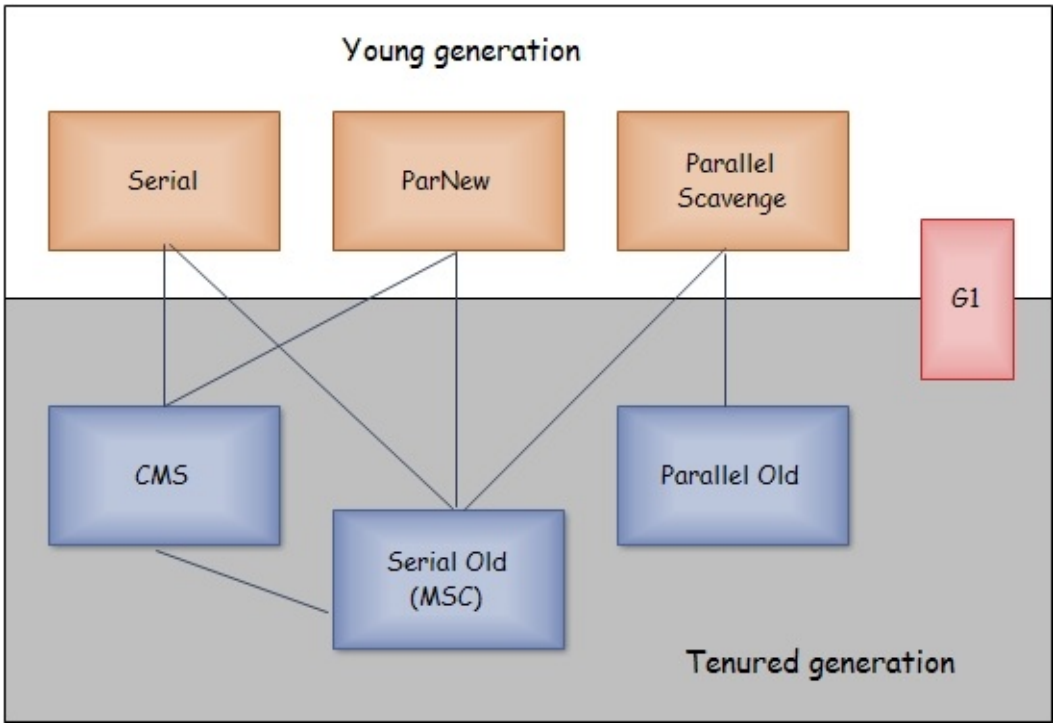
有了安全点之后，也不能完美地解决GC的问题，但实际情况却不一定。当程序没有被分配cpu时间，典型的例子就是线程处于sleep或者blocked状态，这个时候线程无法响应JVM的中断请求，“走”到安全点挂起。对于这种情况，就需要安全区域来解决。

安全区域是指在一段代码片段之中，引用关系不会发生变化。在这个区域中的任意地方开始GC都是安全的，我们也可以把Safe Region看做是被扩展的Safepoint。

关于内存回收，如何进行是由JVM所采用的GC收集器决定的，而通常JVM中往往不止有一种GC收集器。下面看看HotSpot有哪些GC收集器！

垃圾收集器

下面是Sun HotSpot虚拟机1.6版本Update22包含的所有收集器。



Serial Collecor

Serial收集器是单线程收集器，是分代收集器。它进行垃圾收集时，必须暂停其他所有的工作线程，直到它收集结束。

新生代：单线程复制收集算法；老年代：单线程标记整理算法。

Serial一般在单核的机器上使用，是Java 5非服务端JVM的默认收集器，参数-XX:UseSerialGC设置使用。

ParNew收集器

现在大部分的应用都是运行在多核的机器上，显然Serial收集器无法充分利用物理机的CPU资源，因此出现了Parallel收集器。Parallel收集器和Serial收集器的主要区别是新生代的收集，一个是单线程一个是多线程。

老年代的收集和Serial收集器是一样的。

Parallel收集器多在CPU的服务器上，是Java5 服务器端JVM的默认收集器。参数-XX:+UseParallelGC进行设置使用。

Parallel Scavenge收集器

一个新生代收集器，使用复制算法的收集器，又是并行（用户线程阻塞）的多线程收集器。目标是达到一个可控制的吞吐量。

Serial Old 收集器

Serial Old是Serial收集器的老年代版本，它同样是单线程的。使用“标记-整理”算法。

Parallel Old收集器

Parallel old是Parallel Scavenge收集器的老年代版本，使用多线程和“标记-整理”算法。这个收集器是在JDK1.6中才开始提供的。

CMS收集器

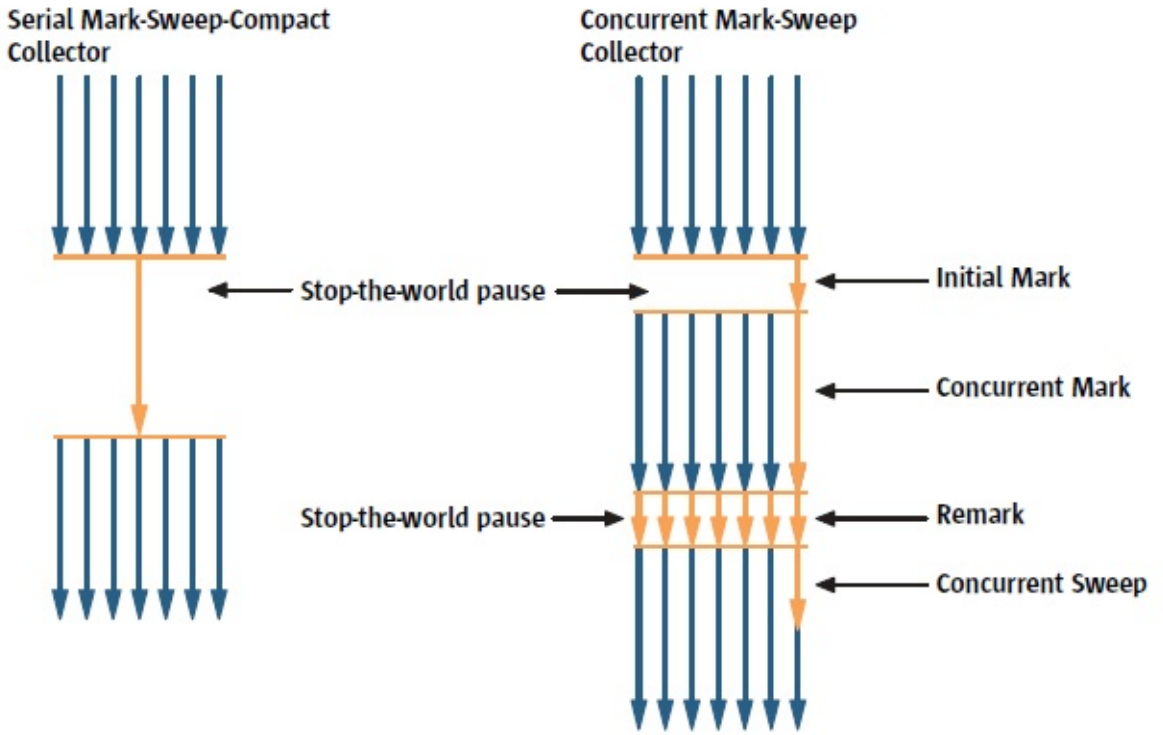
也称“low-latency collector”，为了解决老年代暂停时间过长的问题，并且眞正实现并行收集（程序和GC并行执行）。是一种以获取最短回收停顿时间为目标的收集器。CMS收集器是基于“标记-清除”算法实现的。

新生代：收集和Parallel Collector新生代收集方式一致。

老年代：GC和程序同时进行。

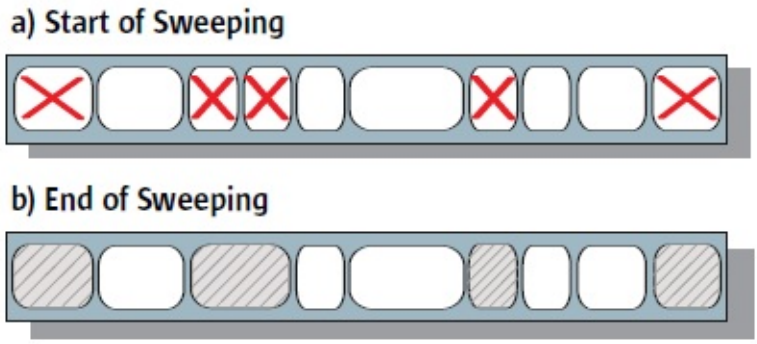
分为四个阶段：

- ①初始标记(initial mark):暂停一会，找出所有活着对象的初始集合。
- ②并行标记(concurrent marking)：根据初始集合，标记出所有的存活对象，由于程序在运行，一部分存活对象无法标出。此过程标记操作和程序同时执行。
- ③重新标记(remark):程序暂停一会，多线程进行重新标记所有在②中没有被标记的存活对象。
- ④并行清理concurrent sweep：回收所有被标记的垃圾区域。和程序同时进行。



由于此收集器在remark阶段重新访问对象，因此开销有所增加。

此收集器的不足是，老年代收集采用标记清除算法，因此会产生很多不连续的内存碎片。



此收集器一般多用于对程序暂停时间要求更短的程序上，多由于web应用（实时性要求高）。参数-XX:+UseConcMarkSweepGC设置使用它。

G1收集器

G1收集器是当今收集器技术发展的最前沿成果之一。G1是一款面向服务端应用的垃圾收集器。HotSpot开发团队赋予它的使命是在未来替换CMS。

它具有以下几个特点：

- 并行与并发：G1能充分利用多CPU、多核环境下的硬件优势。
- 分代收集
- 空间整合：基于“标记-整理”算法实现的收集器。
- 可预测的停顿：这是G1相对于CMS的另一大优势。

关于G1可以具体查看深入理解g1垃圾收集器

理解GC日志

我们先看一段GC日志：

```
[GC [PSYoungGen: 8987K->1016K(9216K)] 9984K->5056K(19456K), 0.0569611 secs] [Times: us
er=0.03 sys=0.02, real=0.06 secs]
[GC [PSYoungGen: 8038K->1000K(9216K)] 12078K->10425K(19456K), 0.0709523 secs] [Times:
user=0.05 sys=0.00, real=0.07 secs]
[Full GC [PSYoungGen: 1000K->0K(9216K)] [ParOldGen: 9425K->8418K(10240K)] 10425K->8418
K(19456K) [PSPermGen: 9678K->9675K(21504K)], 0.3152834 secs] [Times: user=0.39 sys=0.0
0, real=0.32 secs]
[Full GC [PSYoungGen: 8192K->3583K(9216K)] [ParOldGen: 8418K->9508K(10240K)] 16610K->1
3092K(19456K) [PSPermGen: 9675K->9675K(22016K)], 0.1913859 secs] [Times: user=0.34 sys
=0.00, real=0.19 secs]
[Full GC [PSYoungGen: 7716K->7702K(9216K)] [ParOldGen: 9508K->9508K(10240K)] 17224K->1
7210K(19456K) [PSPermGen: 9675K->9675K(21504K)], 0.2769775 secs] [Times: user=0.52 sys
=0.00, real=0.28 secs]
[Full GC [PSYoungGen: 7702K->7702K(9216K)] [ParOldGen: 9508K->9409K(10240K)] 17210K->1
7111K(19456K) [PSPermGen: 9675K->9675K(21504K)], 0.2491993 secs] [Times: user=0.64 sys
=0.00, real=0.25 secs]
```

- “[GC”和“[full DC”说明了这次垃圾回收的停顿类型。如果是调用System.gc()方法所触发的收集，那么这里显示“[Full DC(System)”.
- [DefNew、[Tenured、[Perm 表示GC发生的区域。如果是ParNew收集器，新生代名为“[ParNew”.如果采用Parallel Scavenge收集器，那它配套的新生代名为" [PSYoungGen"。对于老年代和永久代同理。
- [PSYoungGen: 8987K->1016K(9216K)] 9984K->5056K(19456K), 0.0569611 secs]中后面的数字含义是：GC前该内存区域已使用容量->GC后Java堆已使用容量（Java堆总容量）。后面的时间是该区域GC所占用的时间，单位是秒。
- [Times: user=0.03 sys=0.02, real=0.06 secs] 这里的user、sys和real与Linux的time命令所输出的时间含义一，分别代表用户态消耗的CPU时间，内核态消耗的CPU时间和操作从开始到结束所经过的墙钟时间。

垃圾收集器参数总结

参　　数	描　　述
UseSerialGC	虚拟机运行在Client模式下的默认值，打开此开关后，使用Serial + Serial Old的收集器组合进行内存回收
UseParNewGC	打开此开关后，使用ParNew + Serial Old的收集器组合进行内存回收
UseConcMarkSweepGC	打开此开关后，使用ParNew + CMS + Serial Old的收集器组合进行内存回收。Serial Old收集器将作为CMS收集器出现Concurrent Mode Failure失败后的后备收集器使用
UseParallelGC	虚拟机运行在Server模式下的默认值，打开此开关后，使用Parallel Scavenge + Serial Old（PS MarkSweep）的收集器组合进行内存回收
UseParallelOldGC	打开此开关后，使用Parallel Scavenge + Parallel Old的收集器组合进行内存回收
SurvivorRatio	新生代中Eden区域与Survivor区域的容量比值，默认为8，代表Eden:Survivor=8:1
PretenureSizeThreshold	直接晋升到老年代的对象大小，设置这个参数后，大于这个参数的对象将直接在老年代分配
MaxTenuringThreshold	晋升到老年代的对象年龄。每个对象在坚持过一次Minor GC之后，年龄就增加1，当超过这个参数值时就进入老年代
UseAdaptiveSizePolicy	动态调整Java堆中各个区域的大小以及进入老年代的年龄
HandlePromotionFailure	是否允许分配担保失败，即老年代的剩余空间不足以应付新生代的整个Eden和Survivor区的所有对象都存活的极端情况
ParallelGCThreads	设置并行GC时进行内存回收的线程数
GCTimeRatio	GC时间占总时间的比率，默认值为99，即允许1%的GC时间。仅在使用Parallel Scavenge收集器时生效
MaxGCPauseMillis	设置GC的最大停顿时间。仅在使用Parallel Scavenge收集器时生效
CMSInitiatingOccupancyFraction	设置CMS收集器在老年代空间被使用多少后触发垃圾收集。默认值为68%，仅在使用CMS收集器时生效
UseCMSCompactAtFullCollection	设置CMS收集器在完成垃圾收集后是否要进行一次内存碎片整理。仅在使用CMS收集器时生效
CMSFullGCsBeforeCompaction	设置CMS收集器在进行若干次垃圾收集后再启动一次内存碎片整理。仅在使用CMS收集器时生效

内存分配与回收策略

对象的内存分配，往大方向讲，就是在堆上分配（但也可能经过JIT编译后被拆散为标量类型并间接地栈上分配），对象主要分配在新生代的Eden区上，如果启动了本地线程分配缓冲，将按线程优先在TLAB上分配。少数情况下也可能会直接分配在老年代中，分配的规则并不是百分之百固定的，其细节取决于当前使用的是哪一种垃圾收集器组合，还有虚拟机中与内存相关的参数的设置。

接下来我们将会讲解几条最普遍的内存分配规则，并通过代码去验证这些规则。本节下面的代码在测试时使用Client模式虚拟机运行，没有手工指定收集器组合，换句话说，验证的是在使用Serial / Serial Old收集器下（ParNew / Serial Old收集器组合的规则也基本一致）的内存分配和回收的策略。读者不妨根据自己项目中使用的收集器写一些程序去验证一下使用其他几种收集器的内存分配策略。

对象优先在**Eden**分配

大多数情况下，对象在新生代Eden区中分配。当Eden区没有足够空间进行分配时，虚拟机将发起一次Minor GC。

虚拟机提供了-XX:+PrintGCDetails这个收集器日志参数，告诉虚拟机在发生垃圾收集行为时打印内存回收日志，并且在进程退出的时候输出当前的内存各区域分配情况。在实际应用中，内存回收日志一般是打印到文件后通过日志工具进行分析，不过本实验的日志并不多，直接阅读就能看得很清楚。

代码清单3-5的testAllocation()方法中，尝试分配3个2MB大小和1个4MB大小的对象，在运行时通过-Xms20M、 -Xmx20M、 -Xmn10M这3个参数限制了Java堆大小为20MB，不可扩展，其中10MB分配给新生代，剩下的10MB分配给老年代。-XX:SurvivorRatio=8决定了新生代中Eden区与一个Survivor区的空间比例是8:1，从输出的结果也可以清晰地看到“eden space 8192K、from space 1024K、to space 1024K”的信息，新生代总可用空间为9216KB（Eden区+1个Survivor区的总容量）。

执行testAllocation()中分配allocation4对象的语句时会发生一次Minor GC，这次GC的结果是新生代6651KB变为148KB，而总内存占用量则几乎没有减少（因为allocation1、allocation2、allocation3三个对象都是存活的，虚拟机几乎没有找到可回收的对象）。这次GC发生的原因是给allocation4分配内存的时候，发现Eden已经被占用了6MB，剩余空间已不足以分配allocation4所需的4MB内存，因此发生Minor GC。GC期间虚拟机又发现已有的3个2MB大小的对象全部无法放入Survivor空间（Survivor空间只有1MB大小），所以只好通过分配担保机制提前转移到老年代去。

这次GC结束后，4MB的allocation4对象顺利分配在Eden中，因此程序执行完的结果是Eden占用4MB（被allocation4占用），Survivor空闲，老年代被占用6MB（被allocation1、allocation2、allocation3占用）。通过GC日志可以证实这一点。

注意：作者多次提到的**Minor GC**和**Full GC**有什么不一样吗？

- 新生代GC（Minor GC）：指发生在新生代的垃圾收集动作，因为Java对象大多都具备朝生夕灭的特性，所以Minor GC非常频繁，一般回收速度也比较快。
- 老年代GC（Major GC / Full GC）：指发生在老年代的GC，出现了Major GC，经常会伴随至少一次的Minor GC（但非绝对的，在Parallel Scavenge收集器的收集策略里就有直接进行Major GC的策略选择过程）。Major GC的速度一般会比Minor GC慢10倍以上。

```
private static final int _1MB = 1024 * 1024;

/**
 * VM参数：-verbose:gc -Xms20M -Xmx20M -Xmn10M -XX:+PrintGCDetails -XX:SurvivorRatio=8
  */
public static void testAllocation() {
   byte[] allocation1, allocation2, allocation3, allocation4;
   allocation1 = new byte[2 * _1MB];
   allocation2 = new byte[2 * _1MB];
   allocation3 = new byte[2 * _1MB];
   allocation4 = new byte[4 * _1MB];  // 出现一次Minor GC
 }
```

运行结果：

```
[GC [DefNew: 6651K->148K(9216K), 0.0070106 secs] 6651K->6292K(19456K), 0.0070426 secs]
 [Times: user=0.00 sys=0.00, real=0.00 secs]
Heap
 def new generation   total 9216K, used 4326K [0x029d0000, 0x033d0000, 0x033d0000)
  eden space 8192K,  51% used [0x029d0000, 0x02de4828, 0x031d0000)
  from space 1024K,  14% used [0x032d0000, 0x032f5370, 0x033d0000)
  to   space 1024K,   0% used [0x031d0000, 0x031d0000, 0x032d0000)
 tenured generation   total 10240K, used 6144K [0x033d0000, 0x03dd0000, 0x03dd0000)
   the space 10240K,  60% used [0x033d0000, 0x039d0030, 0x039d0200, 0x03dd0000)
 compacting perm gen  total 12288K, used 2114K [0x03dd0000, 0x049d0000, 0x07dd0000)
   the space 12288K,  17% used [0x03dd0000, 0x03fe0998, 0x03fe0a00, 0x049d0000)
No shared spaces configured.
```

大对象直接进入老年代

所谓的大对象是指，需要大量连续内存空间的Java对象，最典型的大对象就是那种很长的字符串以及数组（笔者列出的例子中的byte[]数组就是典型的大对象）。大对象对虚拟机的内存分配来说就是一个坏消息（替Java虚拟机抱怨一句，比遇到一个大对象更加坏的消息就是遇到一群“朝生夕灭”的“短命大对象”，写程序的时候应当避免），经常出现大对象容易导致内存还有不少空间时就提前触发垃圾收集以获取足够的连续空间来“安置”它们。

虚拟机提供了一个-XX:PretenureSizeThreshold参数，令大于这个设置值的对象直接在老年代分配。这样做的目的是避免在Eden区及两个Survivor区之间发生大量的内存复制（复习一下：新生代采用复制算法收集内存）。

执行代码清单3-6中的testPretenureSizeThreshold()方法后，我们看到Eden空间几乎没有被使用，而老年代的10MB空间被使用了40%，也就是4MB的allocation对象直接就分配在老年代中，这是因为PretenureSizeThreshold被设置为3MB（就是3145728，这个参数不能像-Xmx之类的参数一样直接写3MB），因此超过3MB的对象都会直接在老年代进行分配。

注意　PretenureSizeThreshold参数只对Serial和ParNew两款收集器有效，Parallel Scavenge收集器不认识这个参数，Parallel Scavenge收集器一般并不需要设置。如果遇到必须使用此参数的场合，可以考虑ParNew加CMS的收集器组合。

```
private static final int _1MB = 1024 * 1024;

/**
 * VM参数：-verbose:gc -Xms20M -Xmx20M -Xmn10M -XX:+PrintGCDetails -XX:SurvivorRatio=8
 * -XX:PretenureSizeThreshold=3145728
 */
public static void testPretenureSizeThreshold() {
  byte[] allocation;
  allocation = new byte[4 * _1MB];  //直接分配在老年代中
}
```

运行结果：

```
Heap
 def new generation   total 9216K, used 671K [0x029d0000, 0x033d0000, 0x033d0000)
  eden space 8192K,   8% used [0x029d0000, 0x02a77e98, 0x031d0000)
  from space 1024K,   0% used [0x031d0000, 0x031d0000, 0x032d0000)
  to   space 1024K,   0% used [0x032d0000, 0x032d0000, 0x033d0000)
 tenured generation   total 10240K, used 4096K [0x033d0000, 0x03dd0000, 0x03dd0000)
   the space 10240K,  40% used [0x033d0000, 0x037d0010, 0x037d0200, 0x03dd0000)
 compacting perm gen  total 12288K, used 2107K [0x03dd0000, 0x049d0000, 0x07dd0000)
   the space 12288K,  17% used [0x03dd0000, 0x03fdefd0, 0x03fdf000, 0x049d0000)
No shared spaces configured.
```

长期存活的对象将进入老年代

既然虚拟机采用了分代收集的思想来管理内存，那么内存回收时就必须能识别哪些对象应放在新生代，哪些对象应放在老年代中。为了做到这点，虚拟机给每个对象定义了一个对象年龄（Age）计数器。如果对象在Eden出生并经过第一次Minor GC后仍然存活，并且能被Survivor容纳的话，将被移动到Survivor空间中，并且对象年龄设为1。对象在Survivor区中

每“熬过”一次Minor GC，年龄就增加1岁，当它的年龄增加到一定程度（默认为15岁），就将会被晋升到老年代中。对象晋升老年代的年龄阈值，可以通过参数-XX:MaxTenuringThreshold设置。

读者可以试试分别以-XX:MaxTenuringThreshold=1和-XX:MaxTenuringThreshold=15两种设置来执行代码清单3-7中的testTenuringThreshold()方法，此方法中的allocation1对象需要256KB内存，Survivor空间可以容纳。当MaxTenuringThreshold=1时，allocation1对象在第二次GC发生时进入老年代，新生代已使用的内存GC后非常干净地变成0KB。而MaxTenuringThreshold=15时，第二次GC发生后，allocation1对象则还留在新生代Survivor空间，这时新生代仍然有404KB被占用。

```
private static final int _1MB = 1024 * 1024;

/**
 * VM参数：-verbose:gc -Xms20M -Xmx20M -Xmn10M -XX:+PrintGCDetails -XX:SurvivorRatio=8
-XX:MaxTenuringThreshold=1
 * -XX:+PrintTenuringDistribution
 */
@SuppressWarnings("unused")
public static void testTenuringThreshold() {
  byte[] allocation1, allocation2, allocation3;
  allocation1 = new byte[_1MB / 4];
   // 什么时候进入老年代取决于XX:MaxTenuringThreshold设置
  allocation2 = new byte[4 * _1MB];
  allocation3 = new byte[4 * _1MB];
  allocation3 = null;
  allocation3 = new byte[4 * _1MB];
}
```

以MaxTenuringThreshold=1参数来运行的结果：

```
[GC [DefNew
Desired Survivor size 524288 bytes, new threshold 1 (max 1)
- age   1:     414664 bytes,     414664 total
: 4859K->404K(9216K), 0.0065012 secs] 4859K->4500K(19456K), 0.0065283 secs] [Times: us
er=0.02 sys=0.00, real=0.02 secs]
[GC [DefNew
Desired Survivor size 524288 bytes, new threshold 1 (max 1)
: 4500K->0K(9216K), 0.0009253 secs] 8596K->4500K(19456K), 0.0009458 secs] [Times: user
=0.00 sys=0.00, real=0.00 secs]
Heap
 def new generation   total 9216K, used 4178K [0x029d0000, 0x033d0000, 0x033d0000)
  eden space 8192K,  51% used [0x029d0000, 0x02de4828, 0x031d0000)
  from space 1024K,   0% used [0x031d0000, 0x031d0000, 0x032d0000)
  to   space 1024K,   0% used [0x032d0000, 0x032d0000, 0x033d0000)
 tenured generation   total 10240K, used 4500K [0x033d0000, 0x03dd0000, 0x03dd0000)
   the space 10240K,  43% used [0x033d0000, 0x03835348, 0x03835400, 0x03dd0000)
 compacting perm gen  total 12288K, used 2114K [0x03dd0000, 0x049d0000, 0x07dd0000)
   the space 12288K,  17% used [0x03dd0000, 0x03fe0998, 0x03fe0a00, 0x049d0000)
No shared spaces configured.
```

以MaxTenuringThreshold=15参数来运行的结果：

```
[GC [DefNew
Desired Survivor size 524288 bytes, new threshold 15 (max 15)
- age   1:     414664 bytes,     414664 total
: 4859K->404K(9216K), 0.0049637 secs] 4859K->4500K(19456K), 0.0049932 secs] [Times: us
er=0.00 sys=0.00, real=0.00 secs]
[GC [DefNew
Desired Survivor size 524288 bytes, new threshold 15 (max 15)
- age   2:     414520 bytes,     414520 total
: 4500K->404K(9216K), 0.0008091 secs] 8596K->4500K(19456K), 0.0008305 secs] [Times: us
er=0.00 sys=0.00, real=0.00 secs]
Heap
 def new generation   total 9216K, used 4582K [0x029d0000, 0x033d0000, 0x033d0000)
  eden space 8192K,  51% used [0x029d0000, 0x02de4828, 0x031d0000)
  from space 1024K,  39% used [0x031d0000, 0x03235338, 0x032d0000)
  to   space 1024K,   0% used [0x032d0000, 0x032d0000, 0x033d0000)
 tenured generation   total 10240K, used 4096K [0x033d0000, 0x03dd0000, 0x03dd0000)
   the space 10240K,  40% used [0x033d0000, 0x037d0010, 0x037d0200, 0x03dd0000)
 compacting perm gen  total 12288K, used 2114K [0x03dd0000, 0x049d0000, 0x07dd0000)
   the space 12288K,  17% used [0x03dd0000, 0x03fe0998, 0x03fe0a00, 0x049d0000)
No shared spaces configured.
```

动态对象年龄判定

为了能更好地适应不同程序的内存状况，虚拟机并不是永远地要求对象的年龄必须达到了MaxTenuringThreshold才能晋升老年代，如果在Survivor空间中相同年龄所有对象大小的总和大于Survivor空间的一半，年龄大于或等于该年龄的对象就可以直接进入老年代，无须等到

MaxTenuringThreshold中要求的年龄。

执行代码清单3-8中的testTenuringThreshold2()方法，并设置-XX:MaxTenuringThreshold=15，会发现运行结果中Survivor的空间占用仍然为0%，而老年代比预期增加了6%，也就是说，allocation1、allocation2对象都直接进入了老年代，而没有等到15岁的临界年龄。因为这两个对象加起来已经到达了512KB，并且它们是同年的，满足同年对象达到Survivor空间的一半规则。我们只要注释掉其中一个对象new操作，就会发现另外一个就不会晋升到老年代中去了。

```
private static final int _1MB = 1024 * 1024;

/**
 * VM参数：-verbose:gc -Xms20M -Xmx20M -Xmn10M -XX:+PrintGCDetails -XX:SurvivorRatio=8
-XX:MaxTenuringThreshold=15
 * -XX:+PrintTenuringDistribution
 */
@SuppressWarnings("unused")
public static void testTenuringThreshold2() {
  byte[] allocation1, allocation2, allocation3, allocation4;
  allocation1 = new byte[_1MB / 4];
    // allocation1+allocation2大于survivo空间一半
  allocation2 = new byte[_1MB / 4];
  allocation3 = new byte[4 * _1MB];
  allocation4 = new byte[4 * _1MB];
  allocation4 = null;
  allocation4 = new byte[4 * _1MB];
}
```

运行结果：

```
[GC [DefNew
Desired Survivor size 524288 bytes, new threshold 1 (max 15)
- age   1:     676824 bytes,     676824 total
: 5115K->660K(9216K), 0.0050136 secs] 5115K->4756K(19456K), 0.0050443 secs] [Times: us
er=0.00 sys=0.01, real=0.01 secs]
[GC [DefNew
Desired Survivor size 524288 bytes, new threshold 15 (max 15)
: 4756K->0K(9216K), 0.0010571 secs] 8852K->4756K(19456K), 0.0011009 secs] [Times: user
=0.00 sys=0.00, real=0.00 secs]
Heap
 def new generation   total 9216K, used 4178K [0x029d0000, 0x033d0000, 0x033d0000)
  eden space 8192K,  51% used [0x029d0000, 0x02de4828, 0x031d0000)
  from space 1024K,   0% used [0x031d0000, 0x031d0000, 0x032d0000)
  to   space 1024K,   0% used [0x032d0000, 0x032d0000, 0x033d0000)
 tenured generation   total 10240K, used 4756K [0x033d0000, 0x03dd0000, 0x03dd0000)
   the space 10240K,  46% used [0x033d0000, 0x038753e8, 0x03875400, 0x03dd0000)
 compacting perm gen  total 12288K, used 2114K [0x03dd0000, 0x049d0000, 0x07dd0000)
   the space 12288K,  17% used [0x03dd0000, 0x03fe09a0, 0x03fe0a00, 0x049d0000)
No shared spaces configured.
```

空间分配担保

在发生Minor GC之前，虚拟机会先检查老年代最大可用的连续空间是否大于新生代所有对象总空间，如果这个条件成立，那么Minor GC可以确保是安全的。如果不成立，则虚拟机会查看HandlePromotionFailure设置值是否允许担保失败。如果允许，那么会继续检查老年代最大可用的连续空间是否大于历次晋升到老年代对象的平均大小，如果大于，将尝试着进行一次Minor GC，尽管这次Minor GC是有风险的；如果小于，或者HandlePromotionFailure设置不允许冒险，那这时也要改为进行一次Full GC。

下面解释一下“冒险”是冒了什么风险，前面提到过，新生代使用复制收集算法，但为了内存利用率，只使用其中一个Survivor空间来作为轮换备份，因此当出现大量对象在Minor GC后仍然存活的情况（最极端的情况就是内存回收后新生代中所有对象都存活），就需要老年代进行分配担保，把Survivor无法容纳的对象直接进入老年代。与生活中的贷款担保类似，老年代要进行这样的担保，前提是老年代本身还有容纳这些对象的剩余空间，一共有多少对象会活下来在实际完成内存回收之前是无法明确知道的，所以只好取之前每一次回收晋升到老年代对象容量的平均大小值作为经验值，与老年代的剩余空间进行比较，决定是否进行Full GC来让老年代腾出更多空间。

取平均值进行比较其实仍然是一种动态概率的手段，也就是说，如果某次Minor GC存活后的对象突增，远远高于平均值的话，依然会导致担保失败（Handle Promotion Failure）。如果出现了HandlePromotionFailure失败，那就只好在失败后重新发起一次Full GC。虽然担保失败时绕的圈子是最大的，但大部分情况下都还是会将HandlePromotionFailure开关打开，避免Full GC过于频繁，参见代码清单3-9，请读者在JDK 6 Update 24之前的版本中运行测试。

代码和结果，点这里

参考

《深入理解Java虚拟机》读书笔记

最简单例子图解JVM内存分配和回收

虚拟机性能监控与故障处理工具

概述

给一个系统定位问题的时候，知识、经验是关键基础，数据是依据。工具是运用知识处理数据的手段。这里说的数据包括：运行日志、异常堆栈、GC日志、线程快照文件（threaddump/javacore文件）、堆转储快照（heapdump/hprof文件）等。

JDK的命令行工具

JDK的命令行工具大多数是对jdk/lib/tools.jar类库的一层薄包装而已，它们的主要功能代码是在tools类库中实现的。Linux下的这些工具有的甚至是用shell脚本编写的。

SUN JDK监控和故障处理工具：

名称	主要作用
jps	jvm process status tool,显示指定系统内所有的hotspot虚拟机进程
jstat	jvm statistics monitoring tool,用于收集hotspot虚拟机各方面的运行数据
jinfo	configuration info for java，显示虚拟机配置信息
jmap	memory map for java,生成虚拟机的内存转储快照（heapdump文件）
jhat	jvm heap dump browser，用于分析heapmap文件，它会建立一个http/html服务器让用户可以在浏览器上查看分析结果
jstack	stack trace for java ,显示虚拟机的线程快照

jps：虚拟机进程状况工具

以列出正在运行的虚拟机进程，并显示虚拟机执行主类名称以及这些进程的本地虚拟机唯一ID。

jps命令格式 jps [options] [hostid]

jps可以通过RMI协议开启了RMI服务的远程虚拟机进程状态，hostid为RMI注册表中注册的主机名。

jps常用的选项	
属性	作用
-p	只输出LVMID，省略主类的名称
-m	输出虚拟机进程启动时传递给主类main（）函数的参数
-l	输出主类的全名，如果进程执行的是jar包，输出jar路径
-v	输出虚拟机进程启动时jvm参数

jstat：虚拟机统计信息监视工具

jstat是用于监视虚拟机各种运行状态信息的命令行工具。它可以显示本地或者远程虚拟机进程中的类装载、内存、垃圾回收、JIT编译等运行数据，在没有GUI图形界面，只是提供了纯文本控制台环境的服务器上，它将是运行期定位虚拟机性能问题的首选工具。

jstat的命令格式：`jstat [option vmid [interval [s|ms] [count]] ]`

对于命令格式中的VMID和LVMID，如过是本地虚拟机进程，VMID和LVMID是一致的，如果是远程虚拟机，那VMID的格式应当是：`[protocol:] [//] lvmid[@hostname[:port]/servername]`.

参数interval 和count分别表示查询的间隔和次数，如果省略这两个参数，说明只查询一次。

选项	作用
-class	监视装载类、卸载类、总空间以及类装载所耗费的时间
-gc	监视java堆状况，包括eden区、两个survivor区、老年代、永久代等的容量、已用空间、GC时间合计信息
-gccapacity	监视内容与-gc基本相同，但输出主要关注java堆各个区域使用到最大、最小空间
-gcutil	监视内容与-gc基本相同，但输出主要关注已使用控件占总空间的百分比
-gccause	与-gcutil功能一样，但是会额外输出导致上一次gc产生的原因
-gcnew	监视新生代GC情况
-gcnewcapacity	监视内容与-gcnew基本相同，输出主要关注使用到的最大、最小空间
-gcold	监视老年代GC情况
-gcoldcapacity	监视内容与-gcold基本相同，输出主要关注使用到的最大、最小空间
-gcpermcapacity	输出永久代使用到的最大、最小空间
-compiler	输出JIT编译过的方法、耗时等信息
-printcompilation	输出已经被JIT编译过的方法

jinfo：java配置信息工具

jinfo的作用是实时的查看和调整虚拟机各项参数。使用jps命令的-v参数可以查看虚拟机启动时显示指定的参数列表，但如果想知道未被显式指定的参数的系统默认值，除了去找资料以外，就得使用jinfo的-flag选项

jinfo格式 jinfo [option] pid

jinfo在windows 平台仍有很大的限制

jmap：java内存映像工具

jmap命令用于生成堆转储快照。jmap的作用并不仅仅为了获取dump文件，它还可以查询finalize执行队列、java堆和永久代的详细信息。如空间使用率、当前用的是哪种收集器等。

和jinfo命令一样，jmap在windows下也受到比较大的限制。除了生成dump文件的-dump选项和用于查看每个类的实例、控件占用统计的-histo选项在所有操作系用都提供之外，其余选项只能在linux/solaris下使用。

jmap格式 jmap [option] vmid

选项	作用
-dump	生成java堆转储快照。格式为： -dump:[live,]format=b,file=,其中live子参数说明是否只dump出存活的对象
-finalizerinfo	显示在F-Queue中等待Finalizer线程执行finalize方法的对象。只在Linux/Solaris平台下有效
-heap	显示java堆详细信息，如使用哪种收集器、参数配置、分代情况等，在Linux/Solaris平台下有效
-jisto	显示堆中对象统计信息，包含类、实例对象、合集容量
-permstat	以ClassLoader为统计口径显示永久代内存状态。只在Linux/Solaris平台下有效
-F	当虚拟机进程对-dump选项没有相应时。可使用这个选项强制生成dump快照。只在Linux/Solaris平台下有效

jhat：虚拟机堆转储快照分析工具

Sun JDK提供jhat与jmap搭配使用，来分析dump生成的堆快照。jhat内置了一个微型的HTTP/HTML服务器，生成dump文件的分析结果后，可以在浏览器中查看。

用法举例：`jhat test1.bin`

test1.bin为生成的dump文件。

屏幕显示“Server is ready.”的提示后，用户在浏览器中键入http://localhost:7000就可以看到分析的结果了。

分析结果默认是以包围单位进行分组显示，分析内存泄漏问题主要会使用到其中的“Heap Histogram”与OQL标签的功能。前者可以找到内存中总容量最大的对象。后者是标准的对象查询语言，使用类似SQL的语法对内存中的对象进行查询统计。

jstack：java堆栈跟踪工具

jstack命令用于生成虚拟机当前时刻的线程快照（一般称为threaddump或者javacore文件）。线程快照就是当前虚拟机内每一条线程正在执行的方法堆栈集合，生成线程快照的主要目的是定位线程出现长时间停顿的原因，如线程死锁、死循环、请求外部资源导致长时间等待等。

jstack 格式 jstack [option] vmid

option选项的合法值和具体含义

选项	作用
-F	当正常输出的请求不被响应时，强制输出线程堆栈
-l	除堆栈外，显示关于锁的附加信息
-m	如果调用到本地方法的话，可以显示c/c++的堆栈

Tread类新增了一个getAllStackTraces（）方法用于获取虚拟机中所有的线程的StackTraceElement对象。

HSDIS：JIT生成代码反汇编

在Java虚拟机规范中，详细描述了虚拟机指令集中每条指令的执行过程、执行前后对操作数栈、局部变量表的影响等细节。这些细节描述与Sun的早期虚拟机（Sun Classic VM）高度吻合，但随着技术的发展，高性能虚拟机眞正的细节实现方式已经渐渐与虚拟机规范所描述的内容产生了越来越大的差距，虚拟机规范中的描述逐渐成了虚拟机实现的“概念模型”——即实现只能保证规范描述等效。基于这个原因，我们分析程序的执行语义问题（虚拟机做了什么）时，在字节码层面上分析完全可行，但分析程序的执行行为问题（虚拟机是怎样做的、性能如何）时，在字节码层面上分析就没有什么意义了，需要通过其他方式解决。

分析程序如何执行，通过软件调试工具（GDB、Windbg等）来断点调试是最常见的手段，但是这样的调试方式在Java虚拟机中会遇到很大困难，因为大量执行代码是通过JIT编译器动态生成到CodeBuffer中的，没有很简单的手段来处理这种混合模式的调试（不过相信虚拟机开发团队内部肯定是有内部工具的）。因此，不得不通过一些特别的手段来解决问题，基于这种背景，本节的主角——HSDIS插件就正式登场了。

HSDIS是一个Sun官方推荐的HotSpot虚拟机JIT编译代码的反汇编插件，它包含在HotSpot虚拟机的源码之中，但没有提供编译后的程序。在Project Kenai的网站也可以下载到单独的源码。它的作用是让HotSpot的-XX：+PrintAssembly指令调用它来把动态生成的本地代码还原为汇编代码输出，同时还生成了大量非常有价值的注释，这样我们就可以通过输出的代码来分析问题。读者可以根据自己的操作系统和CPU类型从Project Kenai的网站上下载编译好的插件，直接放到JDK_HOME/jre/bin/client和JDK_HOME/jre/bin/server目录中即可。如果没有找到所需操作系统（譬如Windows的就没有）的成品，那就得自己使用源码编译一下。

还需要注意的是，如果读者使用的是Debug或者FastDebug版的HotSpot，那可以直接通过-XX：+PrintAssembly指令使用插件；如果使用的是Product版的HotSpot，那还要额外加入一个-XX：+UnlockDiagnosticVMOptions参数。笔者以代码清单4-6中的简单测试代码为例演示一下这个插件的使用。

关于分析的例子查看书上的例子即可！

利用hsdis和JITWatch查看分析HotSpot JIT compiler生成的汇编代码

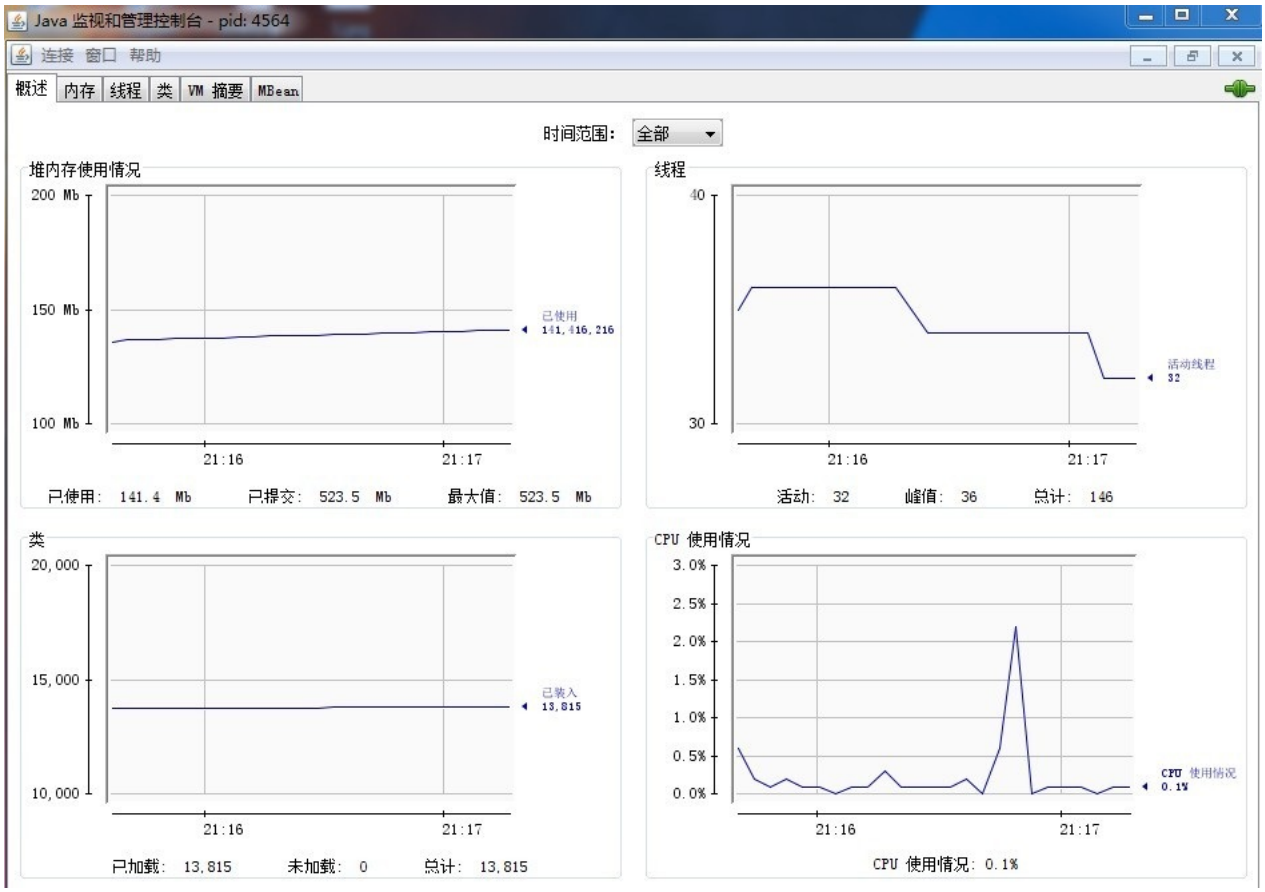
JDK的可视化工具

JDK中除了提供大量的命令行工具外，还有两个功能强大的可视化工具：JConsole和VisualVM。

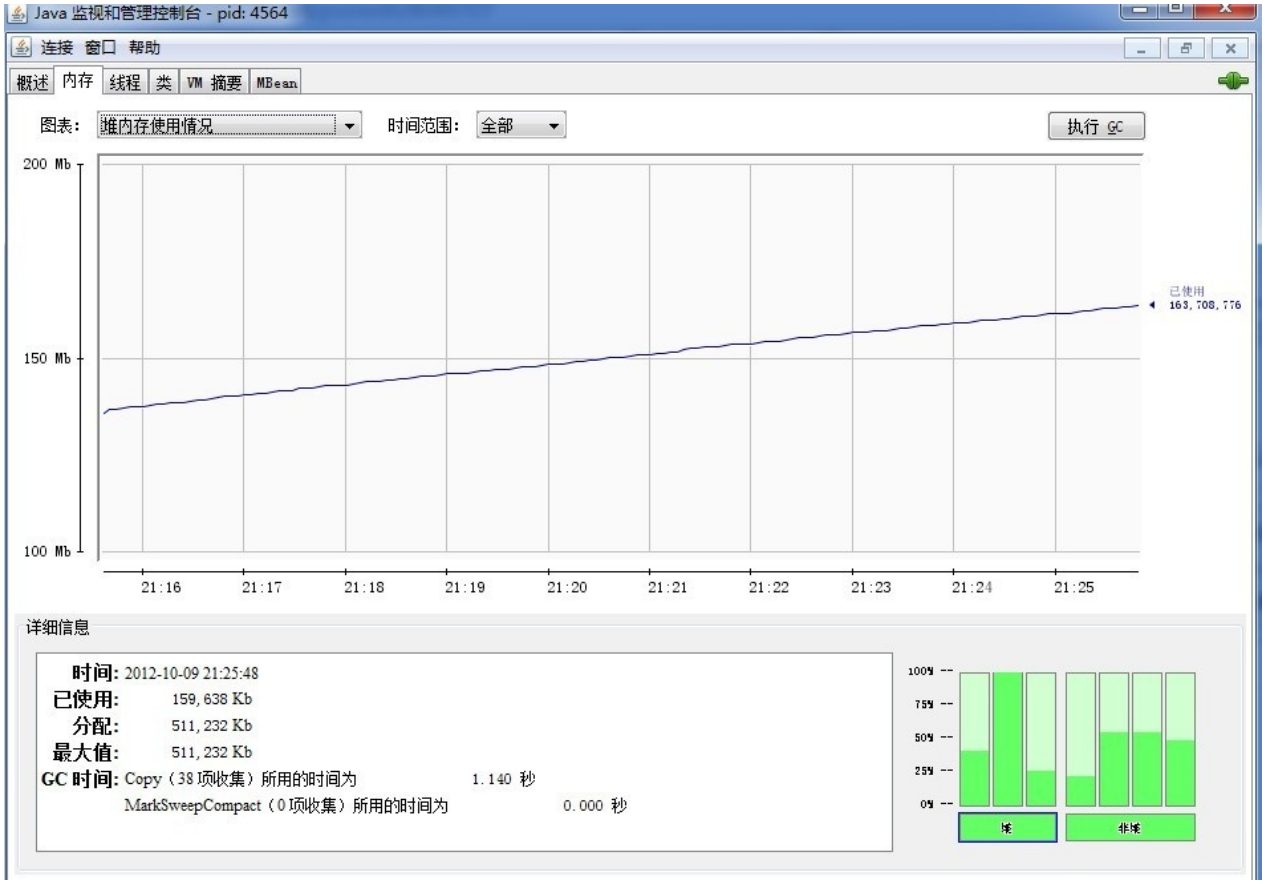
JConsole

JConsole工具在JDK/bin目录下，启动JConsole后，将自动搜索本机运行的jvm进程，不需要jps命令来查询指定。双击其中一个jvm进程即可开始监控，也可使用“远程进程”来连接远程服务器。

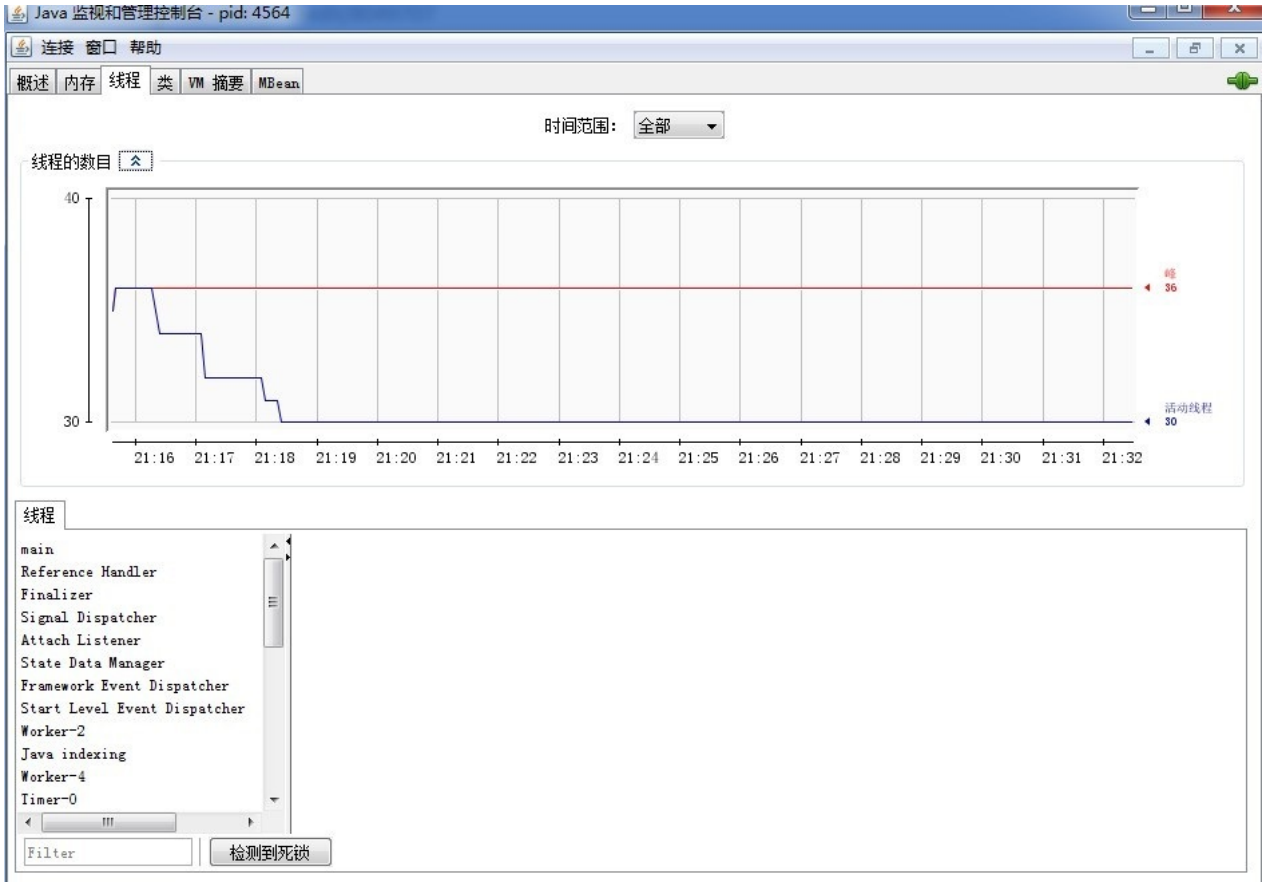
进入JConsole主界面，有“概述”、“内存”、“线程”、“类”、“VM摘要”和"Mbean"六个页签：



内存页签相当于jstat命令，用于监视收集器管理的虚拟机内存(Java堆和永久代)变化趋势，还可在详细信息栏观察全部GC执行的时间及次数。



线程页签：线程长时间停顿的主要原因有：等待外部资源（数据库连接、网络资源、设备资源等）、死循环、锁等待（活锁和死锁）。



最后一个常用页签，VM页签，可清楚的了解显示指定的JVM参数及堆信息。

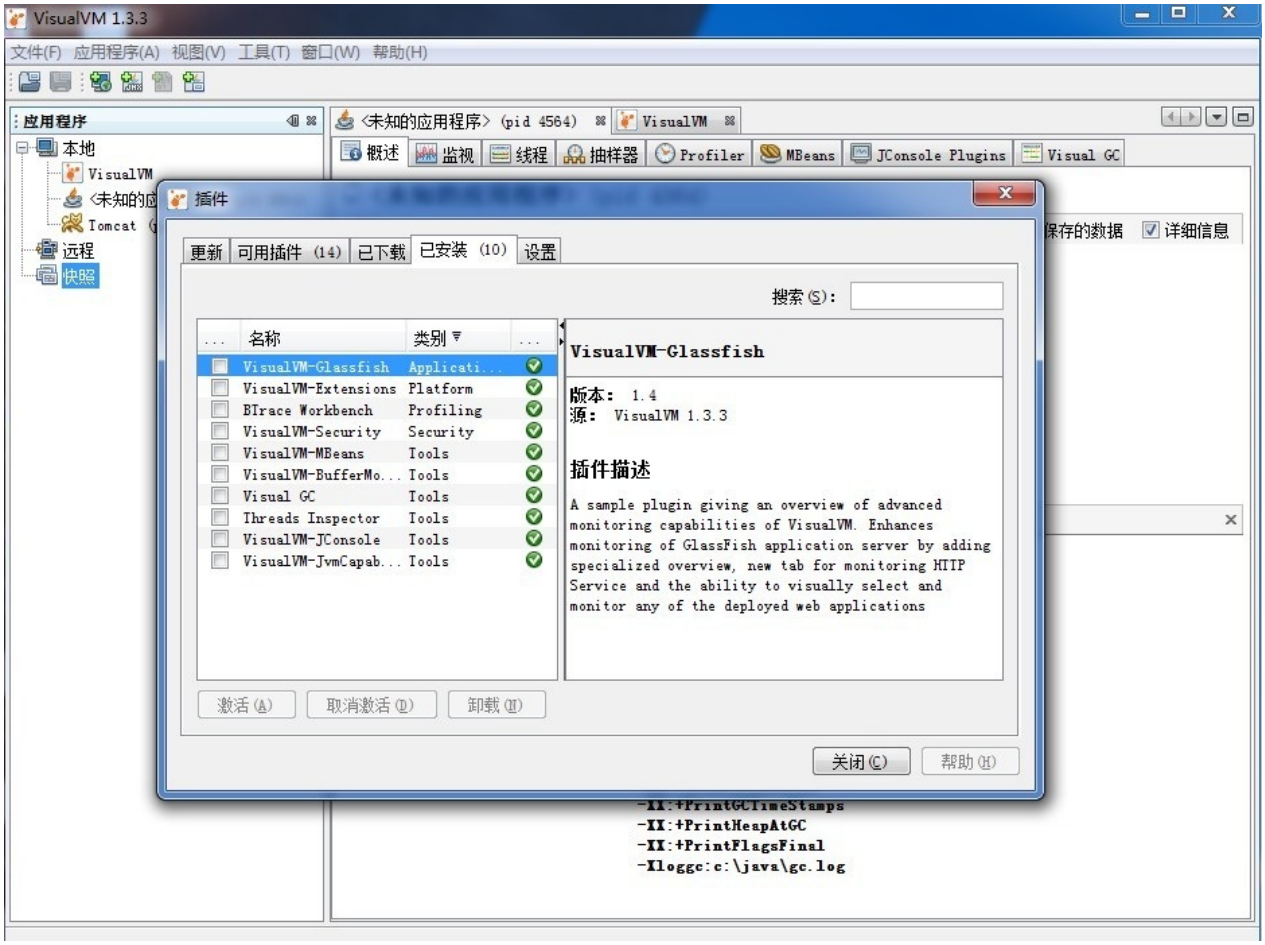
VisualVM：多合一故障处理工具

VisualVM是一个集成多个JDK命令行工具的可视化工具。VisualVM基于NetBeans平台开发，它具备了插件扩展功能的特性，通过插件的扩展，可用于显示虚拟机进程及进程的配置和环境信息(jps，jinfo)，监视应用程序的CPU、GC、堆、方法区及线程的信息(jstat、jstack)等。VisualVM在JDK/bin目录下。

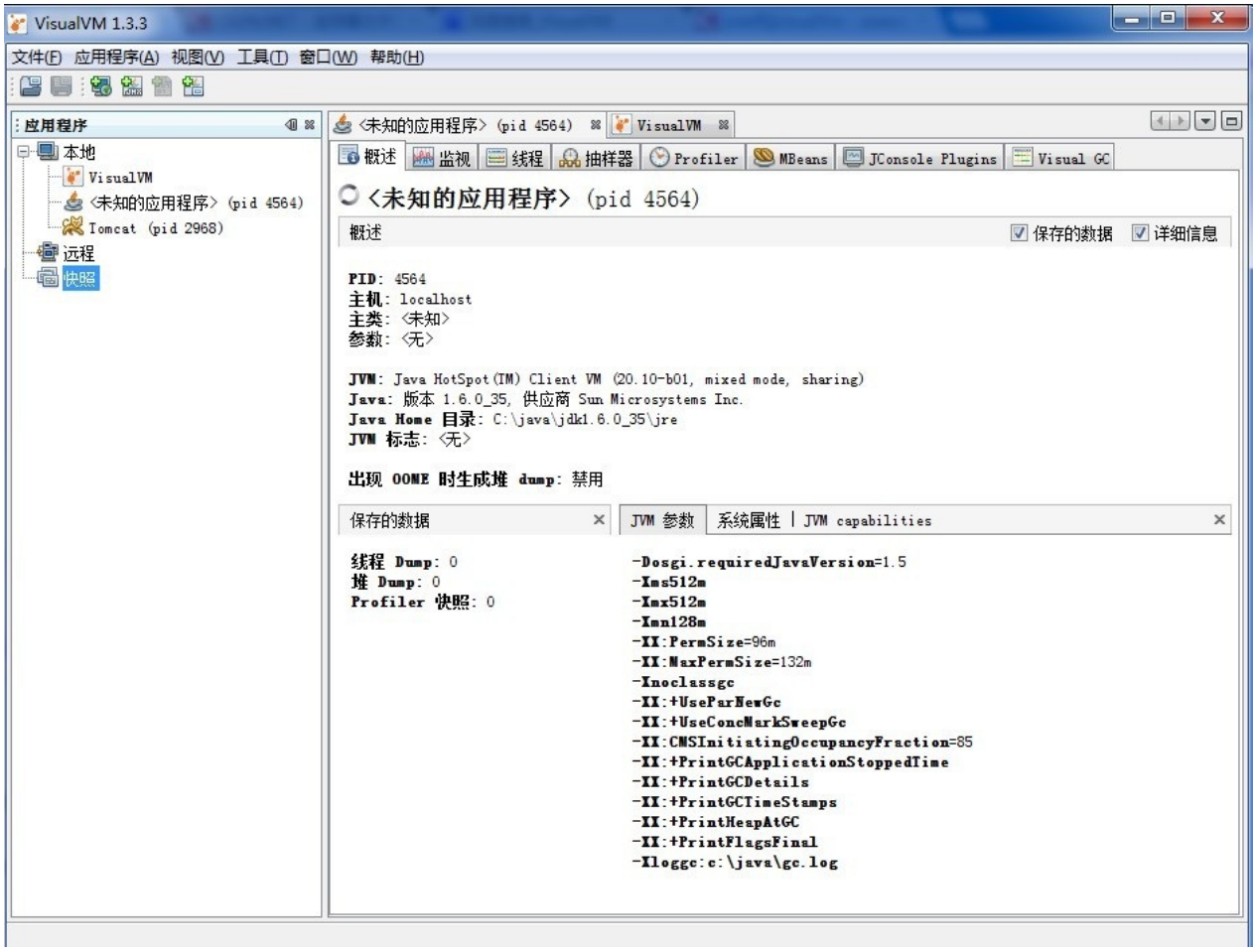
VisualVM的性能分析功能甚至比起JProfiler、YourKit等专业且收费的Profiling工具都不会逊色多少，而且VisualVM还有一个很大的优点：不需要被监视的程序基于特殊Agent运行，因此它对应用程序的实际性能的影响很小，使得它可以直接应用在生产环境中。这个优点是JProfiler、YourKit等工具无法与之媲美的。

插件安装

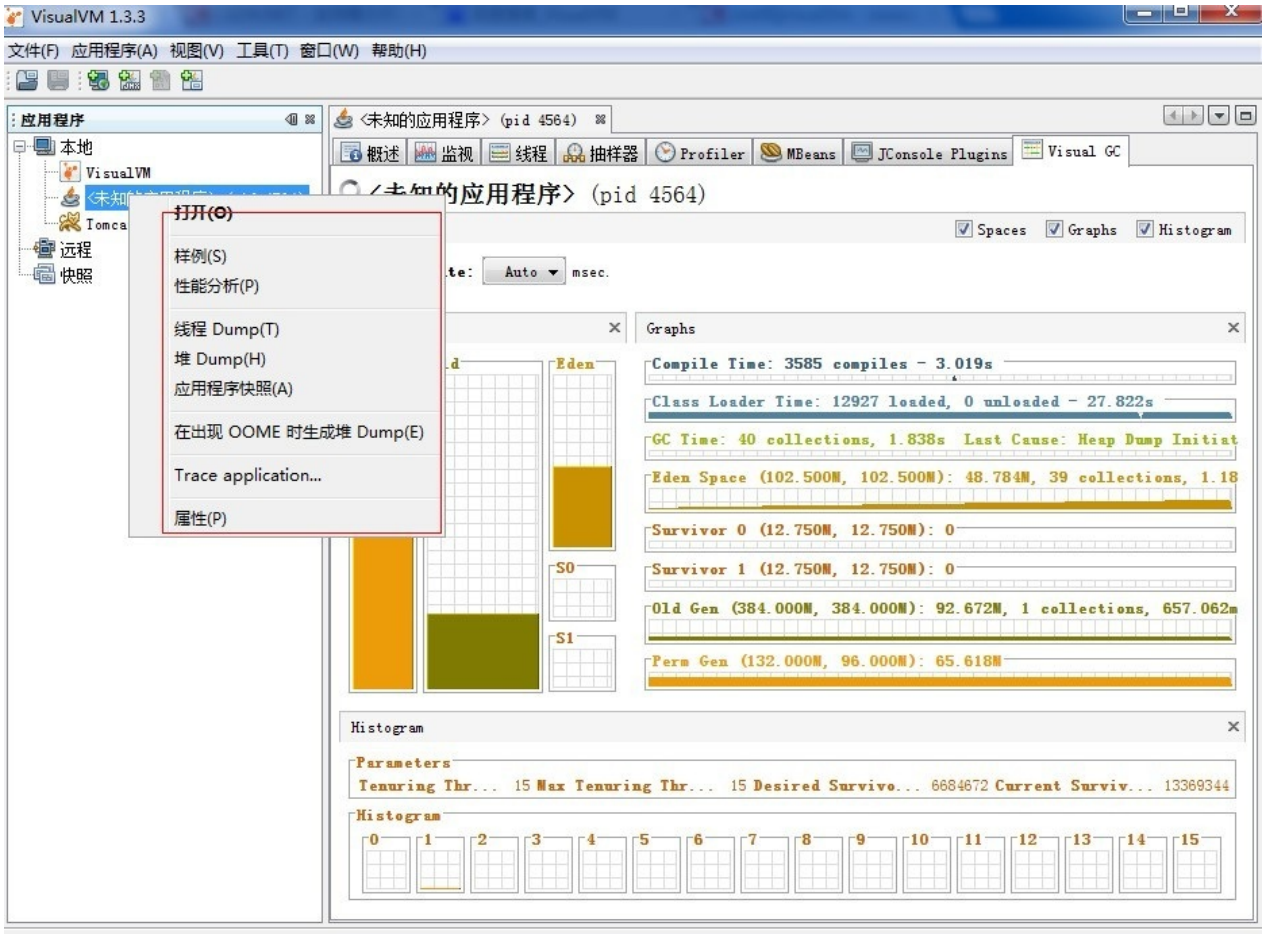
安装插件： 工具- 插件



VisualVM主界面



在VisualVM中生成dump文件：



JProfiler的使用

详见 jprofiler安装图解

调优案例分析与实战

案例分析

高性能硬件上的程序部署策略

1. 通过64位JDK来使用大内存
2. 使用若干个32位虚拟机建立逻辑集群来利用硬件资源。

如果读者计划使用64位JDK来管理大内存，还需要考虑下面可能面临的问题：

- 内存回收导致的长时间停顿。
- 现阶段，64位JDK的性能测试结果普遍低于32位JDK。
- 需要保证程序足够稳定，因为这种应用要是产生堆溢出几乎就无法产生堆转储快照，哪怕产生了转储快照也几乎无法进行分析。
- 相同程序在64位JDK消耗的内存一般比32位JDK大，这是由于指针膨胀，以及数据类型对齐补白等因素导致的。

这些问题听起来吓人，所以现阶段不少管理员选择第二种方式：使用若干个32位虚拟机建立逻辑集群来利用硬件资源。具体做法是在一台物理机器上启动多个应用服务器进程，分配不同端口，然后在前端搭建一个负载均衡器，以及反向代理的方式来分配访问请求。

当然做集群部署，可能会遇到那面的一些问题：

1. 尽量避免节点竞争全局的资源，最典型的就是磁盘竞争。
2. 很难最高效地利用某些资源池，譬如连接池，一般都是在各个节点建立自己独立的连接池，这样有可能导致一些节点池满了而另外一些节点任有较多空余，尽管可以使用JNDI，但是这个有一定的复杂性，并带来额外的性能开销。
3. 各个节点任然不可避免地受到32位的内存限制，在32位Windows平台中，每个进程只能使用2GB的内存，考虑到内存开销，堆一般给1.5G.在Linux或Unix系统中，可以提升到3G乃至接近4G内存。
4. 大量使用本地缓存（如大量使用HashMap作为K/V缓存）的应用，在逻辑集群中会造成较大的内存浪费，因为每个逻辑节点上都有一份缓存，这时候可以考虑把本地缓存改为集中式缓存。

集群间同步导致的内存溢出

在服务使用过程中，往往一个页面会产生数次乃至数十次的请求，因此这个过滤器导致集群各个节点之间网络交互非常频繁。当网络情况不能满足传输要求时，重发数据在内存中不断堆积，很快就产生了内存溢出。

堆外内存导致的溢出错误

例如一个学校的小型项目：基于B/S的电子考试系统，为了实现客户端能实时地从服务器端接收考试数据，系统使用了逆向AJAX技术（也称为Comet或者Server Side Push），选用CometD1.1.1作为服务器端推送框架，服务器是Jetty7.1.4，硬件为一台普通PC机，Core i5CPU， 4GB内存，运行32位Windows操作系统。

没有考虑到堆外内存占用比较高，32位windows平台对每个进程内存限制2G，程序中使用到了CometD 1.1.1框架，有大量的NIO操作需要用到Direct Memory内存，Direct Memory分配不足导致的内存溢出。

从实践经验的角度出发，除了 Java 堆和永久代之外，我们注意到下面这些区域还会占用较多的内存：

- Direct Memory: 可通过- XX: MaxDirectMemorySize 调整大小(HotSpot VM无此参数：http://rednaxelafx.iteye.com/blog/1098791，http://www.dongliu.net/post/504141)，内存不足时抛出 OutOfMemoryError 或者 OutOfMemoryError: Direct buffer memory。
- 线程堆栈：可通过-Xss 调整大小，内存不足时抛出 StackOverflowError（ 纵向无法分配，即无法分配新的栈帧）或者 OutOfMemoryError: unable to create >>li:new native thread（ 横向无法分配，即无法建立新的线程）。
- Socket 缓存区：每个 Socket 连接都 Receive 和 Send 两个缓存区，分别占大约 37KB 和 25KB 内存，连接多的话这块内存占用也比较可观。如果无法分配，则可能会抛出 IOException: Too many open files 异常。
- JNI 代码：如果代码中使用 JNI 调用本地库，那本地库使用的内存也不在堆中。
- 虚拟机和 GC： 虚拟机、 GC 的代码执行也要消耗一定的内存。

外部命令导致系统缓慢

大并发的时候，通过mpstat工具发现CPU使用率很高。

通过Solaris 10的Dtrace脚本可以查看当前情况下那些系统调用话费最多的CPU资源。

结果是fork，用来产生新进程的，Java中不应该有新的进程的产生。

最终找到了答案：每个用户请求的处理都需要执行一个外部 shell 脚本来获得系统的一些信息。执行这个 shell 脚本是通过 Java 的 Runtime. getRuntime(). exec() 方法来调用的。

Java 虚拟机执行这个命令的过程是：首先克隆一个和当前虚拟机拥有一样环境变量的进程，再用这个新的进程去执行外部命令，最后再 退出这个进程。如果频繁执行这个操作，系统的消耗会很大，不仅是 CPU， 内存负担也很重。

服务器JVM进程崩溃

跨系统集成的时候，使用到异步方式调用Web服务，由于两边服务速度不读等，导致很多Web服务没有调用完成，在等待的线程和Socket连接越来越多，超过JVM的承受范围后JVM进程就崩溃了。

可以将异步调用改为生产者/消费者模式的消息队列实现。

测试工具：SoapUI

不恰当数据结构导致内存占用过大

垃圾收集器： ParNew + CMS

在内存中存入了100万个HashMap，就会GC造成停顿。

ParNew 收集器使用的是复制算法，这个算法的高效是建立在大部分对象都“朝生夕灭”的特性上的，如果存活对象过多，把这些对象复制到 Survivor 并维持这些对象引用的正确就成为一个沉重的负担，因此导致 GC 暂停时间明显变长。

如果不修改程序，仅从 GC 调优的角度去解决这个问题，可以考虑将 Survivor 空间去掉（加入参数- XX: SurvivorRatio= 65536、- XX: MaxTenuringThreshold= 0 或者- XX:+ AlwaysTenure），让新生代中存活的对象在第一次 Minor GC 后立即进入老年代，等到 Major GC 的时候再清理它们。这种措施可以治标，但也有很大副作用，治本的方案需要修改程序，因为这里的问题产生的根本原因是用 HashMap ＜ Long, Long ＞结构来存储数据文件空间效率太低。

HashMap分别具有 8B 的 MarkWord、 8B 的 Klass 指针，在加 8B 存储数据的 long 值。在这两个 Long 对象组成 Map. Entry 之后，又多了 16B 的对象头，然后一个 8B 的 next 字段和 4B 的 int 型的 hash 字段，为了对齐，还必须添加 4B 的空白填充，最后还有HashMap 中对这个 Entry 的 8B 的引用，这样增加两个长整型数字，实际耗费的内存为（ Long（ 24B）× 2）+ Entry（ 32B）+ HashMap Ref（ 8B）= 88B， 空间效率为 16B/ 88B= 18%， 实在太低了。

由**Windows**虚拟内存导致的长时间停顿

程序在最小化时它的工作内存被自动交换到磁盘的页面文件之中了，这样发生 GC 时就有可能因为恢复页面文件的操作而导致不正常的 GC 停顿。

在 Java 的 GUI 程序中要避免这种现象，可以加入参数"- Dsun. awt.keepWorkingSetOnMinimize= true" 来解决。

实战：**Eclipse**运行速度调优

这部分知识暂时不看，有时间再来回顾

参考

JVM笔记 – 自动内存管理机制（调优案例分析与实战）

笔记：深入理解JVM 第5章 调优案例分析与实战

类文件结构

无关性的基石

JVM 设计者通过 JSR- 292 基本兑现了对 Java 虚拟机进行适当的扩展，以便更好地支持其他语言运行于 JVM 之上这个承诺。《Java语言规范》和《Java虚拟机规范》。

Java 虚拟机不和包括 Java 在内的任何语言绑定，它只与“ Class 文件”这种特定的二进制文件格式所关联。可以将其他的语言编译成**class**文件，这样就可以实现**JVM**的多语言支持

Class类文件的结构

任何一个 Class 文件都对应着唯一一个类或接口的定义信息，但反过来说，类或接口并不一定都得定义在文件里（譬如类或接口也可以通过类加载器直接生成）。

Class 文件是一组以 8 位字节为基础单位的二进制流。

根据 Java 虚拟机规范的规定， Class 文件格式采用一种类似于 C 语言结构体的伪结构来存储数据，这种伪结构中只有两种数据类型：无符号数和表，后面的解析都要以这两种数据类型为基础。

这里注意：任何一个Class文件都对应着唯一一个雷或者接口的定义信息，但反过来说，类或接口并不一定都得定义在文件里（譬如类或接口也可以通过类加载器直接生成）。

字节码指令简介

这个有点像Java自定义的汇编指令，具体不阐述了。

共有设计和私有设计

只要优化后 Class 文件依然可以被正确读取，并且包含在其中的语义能得到完整的保持，那实现者就可以选择任何方式去实现这些语义，虚拟机后台如何处理 Class 文件完全是实现者自己的事情，只要它在外部接口上看起来与规范描述的一致即可。

Class文件结构的发展

Class文件几乎没有变过，Class文件格式所具备的平台中立（不依赖于特定硬件及操作系统）、紧凑、稳定和可扩展的特点，是Java技术体系实现平台无关、语言无关两项特性的重要支柱。

虚拟机类加载机制

概述

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

Java 里天生可以动态扩展的语言特性就是依赖运行期动态加载和动态连接这个特点实现的。（OSGi技术）

Class文件就是一串二进制的字节流，无论以何种形式存在都可以。

类加载的时机

加载（ Loading）、验证（ Verification）、准备（ Preparation）、解析（ Resolution）、初始化（ Initialization）、使用（ Using）和卸载（ Unloading）7 个阶段。

其中验证、准备、解析 3 个部分统称为连接（ Linking）。

加载、验证、准备、初始化和卸载这 5 个阶段的顺序是确定的，类的加载过程必须按照这种顺序按部就班地开始，而解析阶段则不一定：它在某些情况下可以在初始化阶段之后再开始，这是为了支持 Java 语言的运行时绑定（也称为动态绑定或晚期绑定）。

虚拟机规范则是严格规定了有且只有 **5** 种情况必须立即对类进行**“初始化”**（而加载、验证、准备自然需要在此之前开始）：

- 1） 遇到 new、 getstatic、 putstatic 或 invokestatic 这 4 条字节码指令时，如果类没有进行过初始化，则需要先触发其初始化。
- 2） 使用 java.lang.reflect 包的方法对类进行反射调用的时候，如果类没有进行过初始化，则需要先触发其初始化。
- 3） 当初始化一个类的时候，如果发现其父类还没有进行过初始化，则需要先触发其父类的初始化。
- 4） 当虚拟机启动时，用户需要指定一个要执行的主类（包含 main() 方法的那个类），虚拟机会先初始化这个主类。
- 5） 当使用 JDK1.7 的动态语言支持时，如果一个 java.lang.invoke.MethodHandle 实例最后的解析结果 REF_getStatic、 REF_putStatic、 REF_invokeStatic 的方法句柄，并且这个方法句柄所对应的类没有进行过初始化，则需要先触发其初始化。

通过 子类引用父类的静态字段，不会导致子类初始化。是否要触发子类的加载和验证，在虚拟机规范中并未明确规定，这点取决于虚拟机的具体实现。

通过数组定义来引用类，不会触发此类的初始化。比如`SupperClass[] sca = new SupperClass[10];`这里不会初始化**SupperClass**，但是触发了另外一个名为**“[com.gavin.SuperClass”**的类的初始化阶段，它是一个由虚拟机自动生成的，直接继承于**Object**的子类，创建动作由字节码指令**newarray**触发。

常量在编译阶段会存入调用类的常量池中，本质上并没有直接引用到定义常量的类，因此不会触发定义常量类的初始化。比如**“private static final CONST="123"”,**不会引发此类的初始化。

当一个类在初始化的时候，要求其父类全部都已经初始化过了，但是一个接口在初始化时，并不要求其父类接口全部完成初始化，只有在真正使用到父接口的时候（如引用接口中定义的常量）才会初始化。

类加载过程

加载

在加载阶段，虚拟机需要完成以下 3 件事情：

- 1） 通过一个类的全限定名来获取定义此类的二进制字节流。
- 2） 将这个字节流所代表的静态存储结构转化为方法区的运行时数据结构。
- 3） 在内存中生成一个代表这个类的 java. lang. Class 对象，作为方法区这个类的各种数据的访问入口。

Class文件的获取方式：

- 从 ZIP 包中读取，这很常见，最终成为日后 JAR、 EAR、 WAR 格式的基础。
- 从网络中获取，这种场景最典型的应用就是 Applet。
- 运行时计算生成，这种场景使用得最多的就是动态代理技术，在 java.lang.reflect.Proxy 中，就是用了 ProxyGenerator.generateProxyClass 来为特定接口生成形式为`*$Proxy`的代理类的二进制字节流。
- 由其他文件生成，典型场景是 JSP 应用，即由 JSP 文件生成对应的 Class 类。
- 从数据库中读取。

一个非数组类的加载阶段（准确地说，是加载阶段中获取类的二进制字节流的动作）是开发人员可控性最强的。开发人员可以通过自定义的类加载器去控制字节流的获取方式（即重写一个类加载器的**loadClass**方法），这里以后可以找一些比较重要的类，分析里面所有的函数，从而知道这些东西怎么用。

对于数组类而言，情况就有所不同，数组类本身不通过类加载器创建，它是由 Java 虚拟机直接创建的。但数组类与类加载器仍然有很密切的关系，因为数组类的元素类型（ Element Type， 指的是数组去掉所有维度的类型）最终是要靠类加载器去创建，一个数组类（下面简称为 C） 创建过程就遵循以下规则：

1. 如果数组的组件类型（ Component Type， 指的是数组去掉一个维度的类型）是引用类型，那就递归采用本节中定义的加载过程去加载这个组件类型，数组 C 将在加载该组件类型的类加载器的类名称空间上被标识（这点很重要，在 7. 4 节会介绍到，一个类必须与类加载器一起确定唯一性）。
2. 如果数组的组件类型不是引用类型（例如 int[] 数组）， Java 虚拟机将会把数组 C 标记为与引导类加载器关联。
3. 数组类的可见性与它的组件可见性一致，如果组件类型不是引用类型，那数组类的可见性将默认设置为public

加载阶段完成后，虚拟机外部的二进制字节流就按照虚拟机所需的格式存储在方法区之中，方法区怎么存储由JVM自己定义。然后在内存中实例化一个java.lang.Class类的对象（并没有明确规定是在**Java**堆中，对于**HotSpot**虚拟机而言，**Class**对象比较特殊，它虽然是对象，但是存放在方法区里面）。

加载阶段和连接阶段的部分内容（如一部分字节码文件格式验证动作）是交叉进行的，加载阶段尚未完成，连接阶段可能已经开始。

验证

验证是连接阶段的第一步，这一阶段的目的是为了确保 Class 文件的字节流中包含的信息符合当前虚拟机的要求，并且不会危害虚拟机自身的安全。

在字节码语言层面上，上述 Java 代码无法做到的事情都是可以实现的，至少语义上是可以表达出来的。虚拟机如果不检查输入的字节流，对其完全信任的话，很可能会因为载入了有害的字节流而导致系统崩溃，所以验证是虚拟机对自身保护的一项重要工作。

验证阶段大致上会完成下面 4 个阶段的检验动作：

文件格式验证

第一阶段要验证字节流是否符合 Class 文件格式的规范。

1. 是否以魔数0xCAFEBABE开头
2. 主、次版本号是否在当前虚拟机处理范围之内。
3. 常量池中的常量中是否有不被支持的常量类型（检查常量tag标志）。

.....

元数据验证

第二阶段是对字节码描述的信息进行语义分析，以保证其描述的信息符合 Java 语言规范的要求。

字节码验证

主要目的是通过数据流和控制流分析，确定程序语义是合法的、符合逻辑的。

由于数据流验证的高复杂性，虚拟机设计团队为了避免过多的时间消耗在字节码验证阶段，在 JDK 1.6 之后的 Javac 编译器和 Java 虚拟机中进行了一项优化，给方法体的 Code 属性的属性表中增加了一项名为" StackMapTable" 的属性, 只需要检查StackMapTable属性中的记录是否合法皆可以了。

理论上StackMapTable属性也存在错误或者被篡改的可能，所以是否有可能在恶意篡改了Code属性的同时，也生成相应的StackMapTable属性来骗过虚拟机的类型校验则是虚拟机设计者值得思考的问题。

虚拟机中提供了-XX:-UseSplitVerifier选项来关闭这项优化，或者使用参数-XX:+FailOverToOldVerifier要求在类型校验失败的时候退回到旧的类型推导方式进行校验。

符号引用验证

最后一个阶段的校验发生在虚拟机将符号引用转化为直接引用的时候，这个转化动作将在连接的第三阶段——-——解析阶段中发生。

如果所运行的全部代码（包括自己编写的及第三方包中的代码）都已经被反复使用和验证过，那么在实施阶段就可以考虑使用-Xverify:none 参数来关闭大部分的类验证措施，以缩短虚拟机类加载的时间。

准备

这个阶段进行内存分配的仅包括类变量（被 static 修饰的变量），而不包括实例变量，实例变量将会在对象实例化时随着对象一起分配在 Java 堆中。其次，这里所说的初始值“通常情况”下是数据类型的零值。

```
public static int value=123;
```

value加载类的时候初始化为0，把 value 赋值为 123 的动作将在初始化阶段(方法中)才会执行。

特殊情况：如果类字段的字段属性表中存在 ConstantValue 属性，那在准备阶段变量 value 就会被初始化为 ConstantValue 属性所指定的值，假设上面类变量 value 的定义变为：

```
public static final int value= 123；
```

编译时 Javac 将会为 value 生成 ConstantValue 属性，在准备阶段虚拟机就会根据 ConstantValue 的设置将 value 赋值为 123。

解析

解析阶段是虚拟机将常量池内的符号引用替换为直接引用的过程。

除 invokedynamic(虚拟机指令) 指令以外，虚拟机实现可以对第一次解析的结果进行缓存。

解析动作主要针对类或接口、字段解析、类方法解析、接口方法解析、方法类型解析、方法句柄解析和调用点限定符 7 类符号引用进行。

初始化

初始化阶段是执行类构造器 `<clinit>` 方法的过程。稍后介绍它是怎么生成的，这里我们先看一下 `<clinit>` 方法执行过程中一些可能会影响程序运行行为的特点和细节，这部分相对更贴近于普通的程序开发人员。

1. `<clinit>` 方法是由编译器自动收集类中的所有类变量的赋值动作和静态语句块（static 块）中的语句合并产生的，编译器收集的顺序是由语句在源文件中出现的顺序决定的,静态语句块只能访问到定义在静态语句块之前的变量，定义在它之后的变量，在前面的静态语句块可以赋值，但是不能访问。
2. 虚拟机保证子类的 `<clinit>` 方法执行之前，父类的 `<clinit>` 方法已经执行完毕。
3. 对于接口，不能使用static块，但是可以有静态变量的赋值操作。子类接口的 `<clinit>` 方法调用并不保证父接口的 `<clinit>` 方法被先调用，只有用到父接口的静态变量的时候，父接口 `<clinit>` 方法才会被调用。接口的实现类在初始化时也一样不会执行接口的 `<clinit>` 方法。
4. 虚拟机会保证一个类的 `<clinit>` 方法在多线程环境中被正确地加锁、同步。如果一个线程的 `<clinit>` 方法调用时间过长，就可能造成多个线程阻塞。

类加载器

让应用程序自己决定如何去获取所需要的类。实现这个动作的代码模块称为“类加载器”。

类加载器却在类层次划分、 OSGi、 热部署、代码加密等领域大放异彩。

类与类加载器

即使这两个类来源于同一个 **Class** 文件，被同一个虚拟机加载，只要加载它们的类加载器不同，那这两个类就必定不相等。每一个类加载器，都拥有一个独立的类空间。

不同的类加载器对 instanceof 关键字运算的结果的影响。除此之外，Class对象的equals方法、isAssignableFrom方法、isInstance方法的返回结果也会受影响。

虚拟机中存在了两个 ClassLoaderTest 类，一个是由系统应用程序类加载器加载的，另外一个是由我们自定义的类加载器加载的

双亲委派模型

从 Java 虚拟机的角度来讲，只存在两种不同的类加载器：一种是启动类加载器（ Bootstrap ClassLoader），这个类加载器使用 C++ 语言实现[1]，是虚拟机自身的一部分；另一种就是所有其他的类加载器，这些类加载器都由 Java 语言实现，独立于虚拟机外部，并且全都继承自抽象类 java.lang.ClassLoader。

绝大部分 Java 程序都会使用到以下 3 种系统提供的类加载器：

- 启动类加载器（Bootstrap ClassLoader）：这个类将器负责将存放在 `JAVA_HOME\lib` 目录中的，或者被-Xbootclasspath 参数所指定的路径中的，并且是虚拟机识别的（仅按照文件名识别，如 rt.jar，名字不符合的类库即使放在 lib 目录中也不会被加载）类库加载到虚拟机内存中。用户在编写自定义类加载器时，如果需要把加载请求委派给引导类加载器，那直接使用 null 代替即可。
- 扩展类加载器（Extension ClassLoader）：它负责加载 `JAVA_ HOME\lib\ext` 目录中的，或者被 java.ext.dirs 系统变量所指定的路径中的所有类库。
- 应用程序类加载器（Application ClassLoader）：由于这个类加载器是 ClassLoader 中的 getSystemClassLoader() 方法的返回值，所以一般也称它为系统类加载器。它负责加载用户类路径（ ClassPath）上所指定的类库。如果应用程序中没有自定义过自己的类加载器，一般情况下这个就是程序中默认的类加载器。

双亲委派模型要求除了顶层的启动类加载器外，其余的类加载器都应当有自己的父类加载器。这里类加载器之间的父子关系一般不会以继承（ Inheritance）的关系来实现，而是都使用组合（ Composition）关系来复用父加载器的代码。

如果一个类加载器收到了类加载的请求，它首先不会自己去尝试加载这个类，而是把这个请求委派给父类加载器去完成。

类 java.lang.Object，它存放在 rt.jar 之中，无论哪一个类加载器要加载这个类，最终都是委派给处于模型最顶端的启动类加载器进行加载，因此 Object 类在程序的各种类加载器环境中都是同一个类。

可以尝试去编写一个与 rt.jar 类库中已有类重名的 Java 类，将会发现可以正常编译，但永远无法被加载运行。

破坏双亲委派模型

双亲委派模型的第一次“被破坏”其实发生在双亲委派模型出现之前————即 JDK 1.2 发布之前。

JDK 1.2 之后已不提倡用户再去覆盖 loadClass() 方法，而应当把自己的类加载逻辑写到 findClass() 方法中，在 loadClass() 方法的逻辑里如果父类加载失败，则会调用自己的 findClass() 方法来完成加载，这样就可以保证新写出来的类加载器是符合双亲委派规则的。

双亲委派模型的第二次“被破坏”是由这个模型自身的缺陷所导致的。如果基础类又要调用回用户的代码，那该怎么办？

为了解决这个问题，**Java**设计团队只好引入了一个不太优雅的设计：线程上下文类加载器（**Thread Context ClassLoader**）。这个类加载器可以通过**java.lang.Thread**类的**setContextClassLoader**方法进行设置，如果创建线程时还未设置，它将会从父线程中继承一个，如果在应用程序的全局范围内都没有设置过的话，那这个类加载器默认是应用类加载器。

双亲委派模型的第三次“被破坏”是由于用户对程序动态性的追求而导致的。

OSGi 实现模块化热部署的关键则是它自定义的类加载器机制的实现。每一个程序模块（OSGi 中称为 Bundle）都有一个自己的类加载器，当需要更换一个 Bundle 时，就把 Bundle 连同类加载器一起换掉以实现代码的热替换。在 OSGi 环境下，类加载器不再是双亲委派模型中的树状结构，而是进一步发展为更加复杂的网状结构。

在 Java 程序员中基本有一个共识： OSGi 中对类加载器的使用是很值得学习的，弄懂了 OSGi 的实现，就可以算是掌握了类加载器的精髓。

虚拟机字节码执行引擎

概述

物理机的执行引擎是直接建立在处理器、硬件、指令集和操作系统层面上的，而虚拟机的执行引擎则是由自己实现的，因此可以自行制定指令集与执行引擎的结构体系，并且能够执行那些不被硬件直接支持的指令集格式。

运行时栈帧结构

栈帧存储了方法的局部变量表、操作数栈、动态连接和方法返回地址等信息。每一个方法从调用开始至执行完成的过程，都对应着一个栈帧在虚拟机栈里面从入栈到出栈的过程。

每一个栈帧都包括了局部变量表、操作数栈、动态连接、方法返回地址和一些额外的附加信息。

对于执行引擎来说，在活动线程中，只有位于栈顶的栈帧才是有效的，称为当前栈帧。

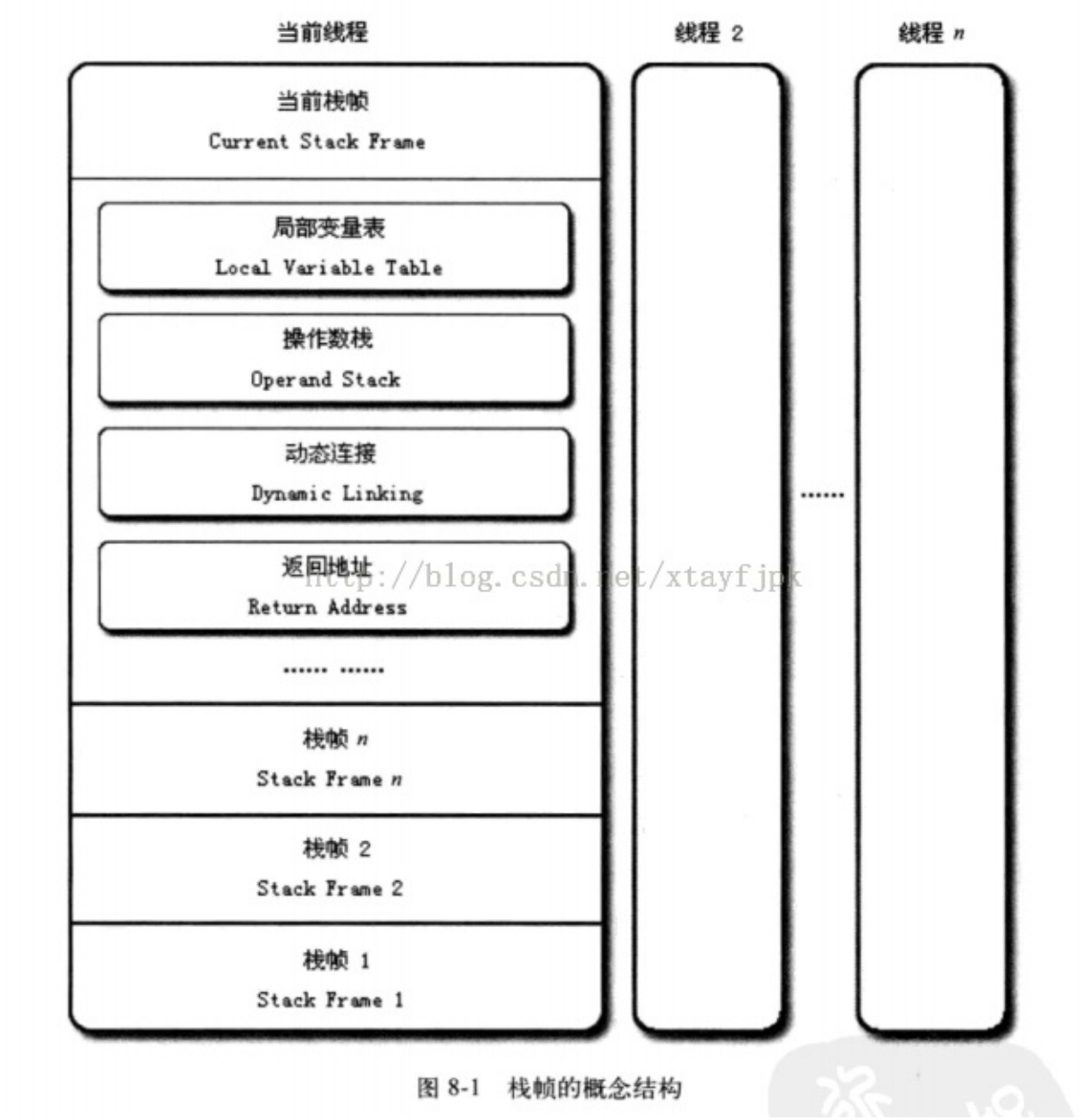


图 8-1　栈帧的概念结构

局部变量表

编译的时候，就在方法的Code属性的max_locals数据项中确定了该方法所需要分配的局部变量表的最大容量。

Java 语言中明确的（ reference 类型则可能是 32 位也可能是 64 位）64 位的数据类型只有 long 和 double 两种。

在方法执行时，虚拟机是使用局部变量表完成参数变量列表的传递过程，如果是实例方法，那么局部变量表中的每0位索引的Slot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”来访问这个隐含的参数，其余参数则按照参数列表的顺序来排列，占用从1开始的局部变量Slot，参数表分配完毕后，再根据方法体内部定义的变量顺序和作用域来分配

其余的Slot。局部变量表中的Slot是可重用的，方法体中定义的变量，其作用域并不一定会覆盖整个方法，如果当前字节码PC计算器的值已经超出了某个变量的作用域，那么这个变量对应的Slot就可以交给其它变量使用。

为了尽可能节省栈帧空间，局部变量表中的 Slot（32位） 是可以重用的，方法体中定义的变量，其作用域并不一定会覆盖整个方法体。

局部变量不像前面介绍的类变量那样存在“准备阶段”。类变量有两次赋初始值的过程，一次在准备阶段，赋予系统初始值；另外一次在初始化阶段，赋予程序员定义的值。因此即使在初始化阶段程序员没有为类变量赋值也没有关系，类变量仍然具有一个确定的初始值。但局部变量就不一样了，如果一个局部变量定义了但没有赋初始值是不能使用的。

原书这里有个有趣的例子，局部变量表复用对垃圾收集的影响

```
public static void main(String[] args){
    byte[] placeHolder = new byte[64 * 1024 * 1024];
    System.gc();
}
```

placeholder 原本所占用的 Slot 还没有被其他变量所复用，所以作为 GC Roots 一部分的局部变量表仍然保持着对它的关联。

书籍《 Practical Java》 中把“不使用的对象应手动赋值为 null” 作为一条推荐的编码规则，赋 null 值的操作在经过 JIT 编译优化后就会被消除掉，这时候将变量设置为 **null** 就是没有意义的。

以下代码片段1在经过 JIT 编译后， System.gc() 执行时就可以正确地回收掉内存，无须写成代码清单2的样子。

代码片段1

```
public static void main(String[] args){
    {
        byte[] placeHolder = new byte[64 * 1024 * 1024];
    }
    System.gc();
}
```

代码片段2

```
public static void main(String[] args){
    {
        byte[] placeHolder = new byte[64 * 1024 * 1024];
        // 这样直接设置为null也是可以被及时回收的
        // placeHolder = null;
    }
    // 离开了placeHolder的作用域之后，执行对局部变量表的读写，让垃圾回收器能够回收到placeHolder占
用的内存
    int a = 0;
    System.gc();
}
```

局部变量不像前面介绍的类变量那样存在“准备阶段”。如果一个局部变量定义了但没有赋初始值是不能使用的。

操作数栈

操作数栈也常被称为操作栈，它是一个后入先出栈。同局部变量表一样，操作数栈的最大深度也是编译的时候被写入到方法表的Code属性的max_stacks数据项中。操作数栈的每一个元素可以是任意Java数据类型，包括long和double。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。栈容量的单位为“字宽”，对于32位虚拟机来说，一个”字宽“占4个字节，对于64位虚拟机来说，一个”字宽“占8个字节。

当一个方法刚刚执行的时候，这个方法的操作数栈是空的，在方法执行的过程中，会有各种字节码指向操作数栈中写入和提取值，也就是入栈与出栈操作。例如，在做算术运算的时候就是通过操作数栈来进行的，又或者调用其它方法的时候是通过操作数栈来行参数传递的。

另外，在概念模型中，两个栈帧作为虚拟机栈的元素，相互之间是完全独立的，但是大多数虚拟机的实现里都会作一些优化处理，令两个栈帧出现一部分重叠。让下栈帧的部分操作数栈与上面栈帧的部分局部变量表重叠在一起，这样在进行方法调用返回时就可以共用一部分数据，而无须进行额外的参数复制传递了，重叠过程如下图：

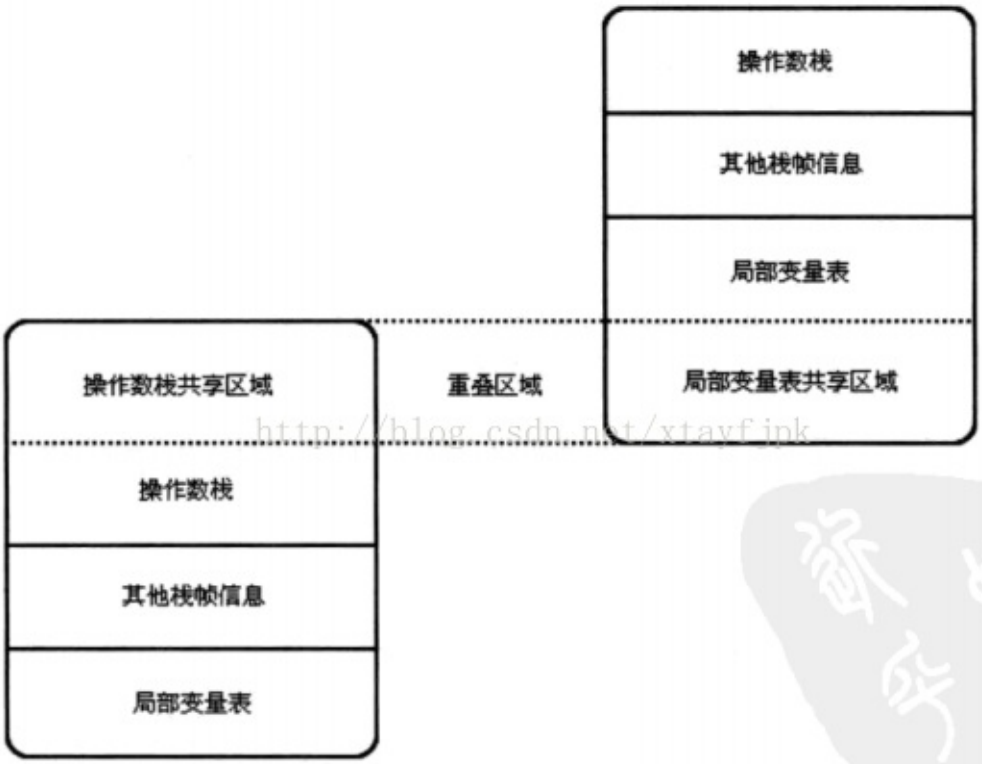


图 8-2　两个栈帧之间的数据共享

动态链接

每个栈帧都包含一个指向运行时常量池中该栈帧所属性方法的引用，持有这个引用是为了支持方法调用过程中的动态连接。在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中指向方法的符号引用为参数。这些符号引用一部分会在类加载阶段或第一次使用的时候转化为直接引用，这种转化称为静态解析。另外一部分将在每一次的运行期期间转化为直接引用，这部分称为动态连接。

方法返回地址

当一个方法被执行后，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(调用当前方法的的方法称为调用者)，是否有返回值和返回值的类型将根据遇到何种方法返回指令来决定，这种退出方法方式称为正常完成出口(Normal Method Invocation Completion)。

另外一种退出方式是，在方法执行过程中遇到了异常，并且这个异常没有在方法体内得到处理，无论是Java虚拟机内部产生的异常，还是代码中使用athrow字节码指令产生的异常，只要在本方法的异常表中没有搜索到匹配的异常处理器，就会导致方法退出，这种退出方式称为异常完成出口(Abrupt Method Invocation Completion)。一个方法使用异常完成出口的方式退出，是不会给它的调用都产生任何返回值的。

无论采用何种方式退出，在方法退出之前，都需要返回到方法被调用的位置，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方法的执行状态。一般来说，方法正常退出时，调用者PC计数器的值就可以作为返回地址，栈帧中很可能会保存这个计数器值。而方法异常退出时，返回地址是要通过异常处理器来确定的，栈帧中一般不会保存这部分信息。

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)压入调用都栈帧的操作数栈中，调用PC计数器的值以指向方法调用指令后面的一条指令等。

附加信息

虚拟机规范允许具体的虚拟机实现增加一些规范里没有描述的信息到栈帧中，例如与高度相关的信息，这部分信息完全取决于具体的虚拟机实现。在实际开发中，一般会把动态连接，方法返回地址与其它附加信息全部归为一类，称为栈帧信息。

方法调用

方法调用并不等同于方法执行，方法调用阶段唯一的任务就是确定调用方法的版本(即调用哪一个方法)，暂时还不涉及方法内部的具体运行过程。在程序运行时，进行方法调用是最普遍、最频繁的操作。在Class文件的编译过程中不包含传统编译中的连接步骤，一切方法调用在Class文件里存储的都只是符号引用，而不是方法在实际运行时内存布局中的入口地址(相当于直接引用)。这个特性给Java带来了更强大的动态扩展能力，但也使得Java方法的调用过程变得相对复杂，需要在类加载期间甚至到运行期间才能确定目标方法的直接引用。

解析

所有方法调用中的目标方法在Class文件里面都是一个常量池中的符号引用，在类加载的解析阶段，会将其中一部分符号引用转化为直接引用，这种解析能成立的前提是：方法在程序真正运行之前就有一可确定的调用版本，并且这个方法的调用版本是运行期是不可改变的。换句话说，调用目标在程序代码写好、编译器进行编译时就必须确定下来。这类方法的调用称为解析(Resolution)。

在Java语言中，符合“编译期可知，运行期不可变”这个要求的方法有静态方法和私有方法两大类，前者与类型直接相关联，后者在外部不可被访问，这两种方法都不可能通过继承或者别的方式重写出其它版本，因此它们都适合在类加载阶段进行静态解析。

与之相对应，在Java虚拟机里提供了5条方法调用字节码指令，分别是：

- invokestatic:调用静态方法
- invokespecial:调用实例构造器方法，私有方法和父类方法。
- invokevirtual:调用虚方法。

- invokeinterface:调用接口方法，会在运行时再确定一个实现此接口的对象。
- invokedynamic:先在运行时动态解析出调用点限定符所引用的方法，然后再执行该方法，在此之前的4条调用指令，分派逻辑是固化在Java虚拟机内部的，而invokedynamic指令的分派逻辑是由用户所设定的引导方法决定的。

只要能被invokestatic与invokespecial指令调用的方法，都可以在解析阶段确定唯一的调用版本，符合这个条件的有静态方法，私有方法，实例构造器和父类方法四类，它们在类加载的时候就会把符号引用解析为该方法的直接引用。这些方法可以统称为非虚方法，与之相反，其它方法就称为虚方法(除去final方法)。

Java中的非虚方法除了使用invokestatic与invokespecial指令调用的方法之后还有一种，就是被final修饰的方法。虽然final方法是使用invokevirtual指令来调用的，但是由于它无法被覆盖，没有其它版本，所以也无须对方法接收都进行多态选择，又或者说多态选择的结果是唯一的。在Java语言规范中明确说明了final方法是一种非虚方法。

解析调用一定是个静态过程，在编译期间就完全确定，在类装载的解析阶段就会把涉及的符号引用全部转变为可确定的直接引用，不会延迟到运行期再去完成。而分派(Dispatch)调用则可能是静态的也可能是动态的，根据分派依据的宗量数可分为单分派与多分派。这两类分派方式两两组件就构成了静态单分派，静态多分派，动态单分派与动态多分派情况。

分派

静态分派

下面是一段程序代码：

```
package com.xtayfjpk.jvm.chapter8;

public class StaticDispatch {

    static abstract class Human {

    }
    static class Man extends Human {

    }
    static class Woman extends Human {

    }

    public void sayHello(Human guy) {
        System.out.println("hello guy...");
    }
    public void sayHello(Man man) {
        System.out.println("hello man...");
    }
    public void sayHello(Woman woman) {
        System.out.println("hello woman...");
    }

    public static void main(String[] args) {
        Human man = new Man();
        Human woman = new Woman();
        StaticDispatch sd = new StaticDispatch();
        sd.sayHello((Man)man);
        sd.sayHello(woman);
    }
}
```

执行结果为：

```
hello man...
hello guy...
```

但为什么会选择执行参数为Human的重载呢？在这之前，先按如下代码定义两个重要的概念：Human man = new Man();

上面代码中的“Human”称为变量的静态类型(Static Type)或者外观类型(Apparent Type)，后面的“Man”则称为变量的实际类型(Actual Type)，静态类型和实际类型在程序中都可以发生一些变化，区别是静态类型的变化仅仅在使用时发生，变量本身的静态类型不会被改变，并且最终的静态类型是编译期可知的；而实际类型变化的结果在运行期才可确定，编译期在编译程序的时候并不知道一个对象的实际类型是什么？如下面的代码：

```
//实际类型变化
Human man = new Man();
man = new Woman();
//静态类型变化
sd.sayHello((Man)man);
sd.sayHello((Woman)man);
```

解释了这两个概念，再回到上术代码中。main()里面的两次sayHello()方法调用，在方法接收者已经确定是对象“sr”的前提下，使用哪个重载版本，就完全取决于传入参数和数据类型。代码中刻意定义了两个静态类型相同，实际类型不同的变量，但虚拟机(准确地说是编译器)在重载时是通过参数的静态类型而不是实际类型作为判定依据的。并且静态类型在编译期是可知的，所以在编译阶段，Javac编译器就根据参数的静态类型决定使用哪个重载版本，所以选择了sayHello(Human)作为调用目标，并把这个方法的符号引用写到main()方法的两条invokevirual指令的参数中。

所有依赖静态类型来定位方法执行版本的分派动作，都称为静态分派。静态分派的最典型应用就是方法重载。静态分派发生在编译阶段，因此确定静态分派的动力实际上不是由虚拟机来执行的。另外，编译器虽然能确定出方法的重载版本，但是很多情况下，这个重载版本并不是“唯一的”，往往只能确定一个“更适合的”版本。这种模糊的结论在0和1构成的计算机世界中算是个比较“稀罕”的事件，产生这种模糊结论的主要原因是字面量不需要定义，所以字面量没有显式的静态类型，它的静态类型只能通过语言上的规则去理解和推断。

动态分派

动态分派与重写(Override)有着很密切的关联。如下代码：

```
package com.xtayfjpk.jvm.chapter8;

public class DynamicDispatch {
    static abstract class Human {
        protected abstract void sayHello();
    }
    static class Man extends Human {
        @Override
        protected void sayHello() {
            System.out.println("man say hello");
        }
    }
    static class Woman extends Human {
        @Override
        protected void sayHello() {
            System.out.println("woman say hello");
        }
    }

    public static void main(String[] args) {
        Human man = new Man();
        Human woman = new Woman();
        man.sayHello();
        woman.sayHello();
        man = new Woman();
        man.sayHello();
    }
}
```

这里显示不可能是根据静态类型来决定的，因为静态类型都是Human的两个变量man和woman在调用sayHello()方法时执行了不同的行为，并且变量man在两次调用中执行了不同的方法。导致这个现象的原是是这两个变量的实际类型不同。那么Java虚拟机是如何根据实际类型来分派方法执行版本的呢，我们使用javap命令输出这段代码的字节码，结果如下：

```
public static void main(java.lang.String[]);
  flags: ACC_PUBLIC, ACC_STATIC
  Code:
    stack=2, locals=3, args_size=1
       0: new           #16                 // class com/xtayfjpk/jvm/chapter8/Dynamic
Dispatch$Man
       3: dup
       4: invokespecial #18                 // Method com/xtayfjpk/jvm/chapter8/Dynami
cDispatch$Man."<init>":()V
       7: astore_1
       8: new           #19                 // class com/xtayfjpk/jvm/chapter8/Dynamic
Dispatch$Woman
      11: dup
      12: invokespecial #21                 // Method com/xtayfjpk/jvm/chapter8/Dynami
cDispatch$Woman."<init>":()V
      15: astore_2
      16: aload_1
      17: invokevirtual #22                 // Method com/xtayfjpk/jvm/chapter8/Dynami
cDispatch$Human.sayHello:()V
      20: aload_2
      21: invokevirtual #22                 // Method com/xtayfjpk/jvm/chapter8/Dynami
cDispatch$Human.sayHello:()V
      24: new           #19                 // class com/xtayfjpk/jvm/chapter8/Dynamic
Dispatch$Woman
      27: dup
      28: invokespecial #21                 // Method com/xtayfjpk/jvm/chapter8/Dynami
cDispatch$Woman."<init>":()V
      31: astore_1
      32: aload_1
      33: invokevirtual #22                 // Method com/xtayfjpk/jvm/chapter8/Dynami
cDispatch$Human.sayHello:()V
      36: return
```

0-15行的字节码是准备动作，作用是建立man和woman的内存空间，调用Man和Woman类的实例构造器，将这两个实例的引用存放在第1和第2个局部变量表Slot之中，这个动作对应了代码中这两句：

```
Human man = new Man();
Human woman = new Woman();
```

接下来的第16-21行是关键部分，第16和第20两行分别把刚刚创建的两个对象的引用压到栈顶，这两个对象是将执行的sayHello()方法的所有者，称为接收者(Receiver)，第17和第21两行是方法调用指令，单从字节码的角度来看，这两条调用指令无论是指令(都是invokevirtual)还是参数(都是常量池中Human.sayHello()的符号引用)都完全一样，但是这两条指令最终执行的目标方法并不相同，其原因需要从invokevirutal指令的多态查找过程开始说起，invokevirtual指令的运行时解析过程大致分为以下步骤：

- a.找到操作数栈顶的第一个元素所指向的对象实际类型，记作C。

- b.如果在类型C中找到与常量中描述符和简单名称都相同的方法，则进行访问权限校验，如果通过则返回这个方法的直接引用，查找结束；不通过则返回java.lang.IllegalAccessError错误。
- c.否则，按照继承关系从下往上依次对C的各个父类进行第2步的搜索与校验过程。
- d.如果始终没有找到合适的方法，则抛出java.lang.AbstractMethodError错误。

  由于invokevirtual指令执行的第一步就是在运行期确定接收者的实际类型，所以两次调用中的invokevirtual指令把常量池中的类方法符号引用解析到了不同的直接引用上，这个过程就是Java语言中方法重写的本质。我们把这种在运行期根据实际类型确定方法执行版本的分派过程称为动态分派。

单分派与多分派

方法的接收者与方法的参数统称为方法的宗量。根据分派基于多少种宗量，可以将分派划分为单分派与多分派两种。单分派是根据一个宗量来对目标方法进行选择，多分派则是根据多于一个宗量对目标方法进行选择。

在编译期的静态分派过程选择目标方法的依据有两点：一是静态类型；二是方法参数，所以Java语言的静态分派属于多分派类型。在运行阶段虚拟机的动态分派过程只能接收者的实际类型一个宗量作为目标方法选择依据，所以Java语言的动态分派属于单分派类型。所在Java语言是一门静态多分派，动态单分派语言。

这部分的实现和例子，需要去看原书。P255.

虚拟机动态分派的实现

由于动态分派是非常频繁的动作，而且动态分派的方法版本选择过程需要在运行时在类的方法元数据中搜索合适的目标方法，因此在虚拟机的实际实现中基于性能的考虑，大部分实现都不会眞的进行如此频繁的搜索。面对这种情况，最常用的优化手段就是在类的方法区中建立一个虚方法表**(Virtual Method Table**，也称**vtable**，与此对应，在**invokeinterface**执行时也会用到接口方法表，**Interface Method Table**，也称**itable)**，使用虚方法表索引来代替元数据据查找以提高性能。

虚方法表中存放着各个方法的实际入口地址。如果某个方法在子类中没有被重写，那么子类的虚方法表里面的地址入口和父类方法的地址入口是一致的，都指向父类的实现入口。如果子类中重写了这个方法，子类方法表中的地址将会被替换为指向子类实现版本的地址入口。

动态类型语言支持

随着 JDK 7 的发布，字节码指令集终于迎来了第一位新成员—— `invokedynamic` 指令。这条新增加的指令是 JDK 7 实现“动态类型语言”（ Dynamically Typed Language） 支持而进行的改进之一。

动态类型语言

动态类型语言的关键特征是它的类型检查的主体过程是在运行期而不是编译期。

“变量无类型而变量值才有类型”这个特点也是动态类型语言的一个重要特征。

JDK 1.7 与动态类型

java.lang.invoke 包

JDK 1.7 实现了 JSR-292， 新加入的 java.lang.invoke 包就是 JSR-292 的一个重要组成部分。

```
public class MethodHandleTest {
  static class ClassA{
      public void println(String s){
          System.out.println(s);
      }
  }

  public static void main(String[] args) throws Throwable {
      Object obj = System.currentTimeMillis() % 2 == 0 ? System.out:new ClassA();
      /* 无论obj最终是那个实现类，下面这句都能正确调用到println方法 */
      getPrintlnMH(obj).invokeExact("icyfenix");
      /* output:
       * icyfenix
       */
  }

  private static MethodHandle getPrintlnMH(Object receiver) throws Throwable{
      /* MethodType: 代表“方法类型”，包含了方法的返回值（methodType()的第一个参数）
       * 和具体参数（methodType()第二个及以后的参数） */
      MethodType mt = MethodType.methodType(void.class, String.class);
      /* lookup()方法来自于MethodHandles.lookup，
       * 这句的作用是在指定类中查找符合给定的方法名称、方法类型，并且符合调用权限的方法句柄。
       * 因为这里调用的是一个虚方法，按照Java语言的规则，方法第一个参数是隐式的，代表该方法的接收者，
       * 也即是this指向的对象，这个参数以前是放在参数列表中进行传递的，而现在提供给了bindTo()方法来
完成这件事情 */
      return MethodHandles.lookup().findVirtual(receiver.getClass(), "println", mt)
              .bindTo(receiver);
  }
}
```

MethodHandle 的使用方法和效果与 Reflection 有众多相似之处，不过，它们还是有以下这些区别：

- Reflection 是在模拟 Java 代码层次的方法调用，而 MethodHandle 是在模拟字节码层次的方法调用。

- **Reflection** 是重量级，而 **MethodHandle** 是轻量级。

Reflection API 的设计目标是只为 Java 语言服务的，而 MethodHandle 则设计成可服务于所有 Java 虚拟机之上的语言，其中也包括 Java 语言。

invokedynamic 指令

在某种程度上， invokedynamic 指令与 MethodHandle 机制的作用是一样的，都是为了解决原有 4 条" invoke*" 指令方法分派规则固化在虚拟机之中的问题，把如何查找目标方法的决定权从虚拟机转嫁到具体用户代码之中。

掌控方法分派规则

invokedynamic 指令与前面 4 条" invoke*" 指令的最大差别就是它的分派逻辑不是由虚拟机决定的，而是由程序员决定。

可以通过" super" 关键字很方便地调用到父类中的方法，但如果要访问祖类的方法呢？

使用 MethodHandle 来解决相关问题.

基于栈的字节码解析执行引擎

解析执行

Java 语言中， Javac 编译器完成了程序代码经过词法分析、语法分析到抽象语法树，再遍历语法树生成线性的字节码指令流的过程。因为这一部分动作是在 Java 虚拟机之外进行的，而解释器在虚拟机的内部，所以 Java 程序的编译就是半独立的实现。

基于栈的指令集与基于寄存器的指令集

Java 编译器输出的指令流，基本上是一种基于栈的指令集架构。

基于栈的指令集主要的优点就是可移植，寄存器由硬件直接提供，程序直接依赖这些硬件寄存器则不可避免地要受到硬件的约束。

栈架构指令集的主要缺点是执行速度相对来说会稍慢一些。

虽然栈架构指令集的代码非常紧凑，但是完成相同功能所需的指令数量一般会比寄存器架构多，因为出栈、入栈操作本身就产生了相当多的指令数量。更重要的是，栈实现在内存之中，频繁的栈访问也就意味着频繁的内存访问，相对于处理器来说，内存始终是执行速度的瓶颈。尽管虚拟机可以采取栈顶缓存的手段，把最常用的操作映射到寄存器中避免直接内存访问，但这也只能是优化措施而不是解决本质问题的方法。由于指令数量和内存访问的原因，所以导致了栈架构指令集的执行速度会相对较慢。

基于栈的解析执行过程

在 HotSpot 虚拟机中，有很多以" fast_" 开头的非标准字节码指令用于合并、替换输入的字节码以提升解释执行性能，而即时编译器的优化手段更加花样繁多。

参考

深入理解Java虚拟机笔记---运行时栈帧结构

深入理解Java虚拟机笔记---方法调用

JVM笔记 – 虚拟机执行子系统（虚拟机字节码执行引擎）

类加载及执行子系统的案例与实战

概述

能通过程序进行操作的，主要是字节码生成与类加载器这两部分的功能。

案例分析

Tomcat：正统的类加载器架构

主流的 Java Web 服务器，如 Tomcat、 Jetty、 WebLogic、 WebSphere 或其他笔者没有列举的服务器，都实现了自己定义的类加载器（一般都不止一个）。因为一个功能健全的 Web 服务器，要解决如下几个问题：

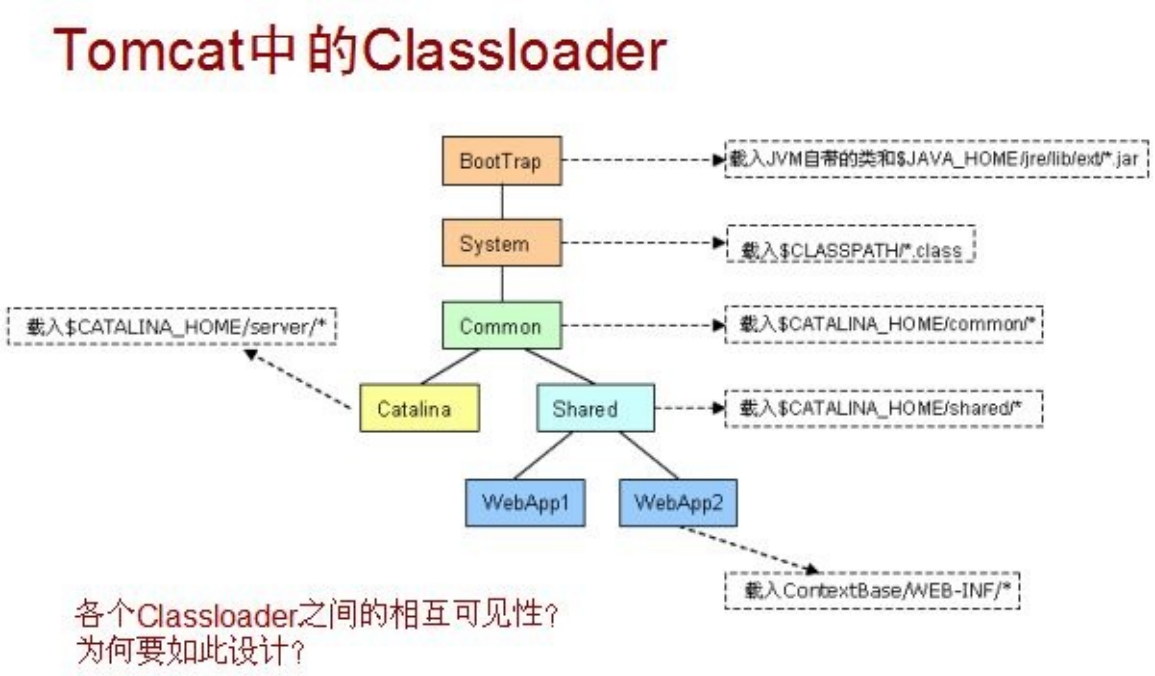
- 部署在同一个服务器上的两个 Web 应用程序所使用的 Java 类库可以实现相互隔离。
- 部署在同一个服务器上的两个 Web 应用程序所使用的 Java 类库可以互相共享。
- 服务器需要尽可能地保证自身的安全不受部署的 Web 应用程序影响。
- 支持 JSP 应用的 Web 服务器，大多数都需要支持 HotSwap 功能。

在部署 Web 应用时，单独的一个 ClassPath 就无法满足需求了，所以各种 Web 服务器都“不约而同”地提供了好几个 ClassPath 路径供用户存放第三方类库。

在 Tomcat 目录结构中，有 3 组目录（"`/common/*`"、"`/server/*`"和"`/shared/*`"）可以存放 Java 类库，另外还可以加上 Web 应用程序自身的目录"`/WEB-INF/*`"，一共 4 组。

- `/common/*`：放在这个下面的类库，可以被Tomcat和多有的Web应用程序共同使用。
- `/server/*`：类库可以被Tomcat使用，对所有的Web应用程序都不可见。
- `/shared/*`：类库可被所有web应用程序共同使用，但对Tomcat自己不可见
- `/WEB-INF/*`：类库仅仅可以被此Web应用程序使用，对Tomcat和其他web程序都不可见。

`CommonClassLoader`、`CatalinaClassLoader`、`SharedClassLoader`和`WebappClassLoader`则是 Tomcat 自己定义的类加载器，



Tomcat热部署原理：

JasperLoader的加载范围仅仅是这个 JSP 文件所编译出来的那一个 Class， 它出现的目的就是为了被丢弃：当服务器检测到 JSP 文件被修改时，会替换掉目前的 JasperLoader 的实例，并通过再建立一个新的 Jsp 类加载器来实现 JSP 文件的 HotSwap 功能。

对于 Tomcat 的 6.x 版本，只有指定了 tomcat/conf/catalina.properties配置文件的 server.loader 和 share.loader 项后才会眞正建立 CatalinaClassLoader 和 SharedClassLoader 的实例，否则会用到这两个类加载器的地方都会用 CommonClassLoader 的实例代替，而默认的配置文件中没有设置这两个 loader 项，所以 Tomcat 6.x 顺理成章地把/common、/server 和/shared 三个目录默认合并到一起变成一个/lib 目录。这个目录里的类库相当于以前/common目录中类库的作用。这是Tomcat设计团队为了简化部署所做的一项改进，如果默认设置不能满足需要，用户可以通过修改配置文件指定 server.loader和share.loader的方式重新启动5.X的加载器架构。

OSGi：灵活的类加载器架构

Java社区有这样一句话：“学习JEE规范，去看JBoss源码；学习类加载器，就去看OSGi源码”。

OSGi在Java程序员中最著名的应用案例就是Eclipse IDE。另外还有许多大型的软件平台和中间件服务器都基于或声明将会基于OSGi规范来实现。

OSGi中的每个模块（称为Bundle）与普通的Java类库区别并不大，两者一般都以JAR格式进行封装，并且内部存储的都是Java Package和Class。但是一个Bundle可以声明它所依赖的 Java Package（通过Import-Package描述），也可以声明它允许导出发布的Java Package（通过Export-Package描述）。在OSGi里面，Bundle之间的依赖关系从传统的上层

模块依赖底层模块转变为平级模块之间的依赖，而且类库的可见性能得到非常精确的控制，一个模块里只有被Export过的Package才可能由外界访问，其他的Package和Class将会隐藏起来。除了更精确的模块划分和可见性控制外，引入OSGi的另外一个理由是，基于 OSGi 的程序很可能（只是很可能，并不是一定会）可以实现模块级的热插拔功能，当程序升级更新或调试除错时，可以只停用、重新安装然后启用程序的其中一部分。

OSGi的这些功能，得益于它灵活的类加载器架构。OSGi的Bundle类加载器之间只有规则，没有固定的委派关系。例如某个Bundle声明了一个它依赖的Package，如果有其他Bundle声明发布了这个Package，那么所有对这个Package的类加载动作都会委派给发布它的Bundle类加载器去完成。不涉及某个具体的Package时，各个Bundle加载器都是平级关系，只有具体使用某个Package和Class的时候，才会根据Package导入导出定义来构造Bundle间的委托和依赖。

如果一个类存在于 Bundle 的类库中但是没有被 Export， 那么这个 Bundle 的类加载器能找到这个类，但不会提供给其他 Bundle 使用，而且OSGi平台也不会把其他Bundle的类加载器请求分配给这个Bundle来处理。

并非所有的应用都适合采用 OSGi 作为基础架构， OSGi 在提供强大功能的同时，也引入了额外的复杂度，带来了线程死锁和内存泄漏的风险。

字节码生成技术与动态代理的实现

javac 也是一个由 Java 语言写成的程序，它的代码存放在 OpenJDK 的 langtools/src/share/classes/com/sun/tools/javac 目录中。 要深入了解字节码生成，阅读 javac 的源码是个很好的途径。

动态代理实现了可以在原始类和接口还未知的时候，就确定代理类的代理行为，当代理类与原始类脱离直接联系后，就可以很灵活地重用于不同的应用场景之中。调用了 sun.misc.ProxyGenerator.generateProxyClass() 方法来完成生成字节码的动作，这个方法可以在运行时产生一个描述代理类的字节码 byte[] 数组。

这里只是列出简单的例子，有时间可以去原书看看！P282.

Retrotranslator：跨越JDK版本

一种名为“ Java 逆向移植”的工具（ Java Backporting Tools）应运而生， Retrotranslator[1] 是这类工具中较出色的一个。

编译器在程序中使用到 `包装对象` 的地方自动插入了很多 Integer.valueOf()、 Float.valueOf() 之类的代码； `变长参数` 在编译之后就自动转化成了一个数组来完成参数传递； `泛型` 的信息则在编译阶段就已经擦除掉了（但是在元数据中还保留着），相应的地方被编译器自动插入了类型转换代码[2]。

从字节码的角度来看，`枚举` 仅仅是一个继承于 java. lang. Enum、自动生成了 values() 和 valueOf() 方法的普通 Java 类而已。

具体看原书！

实战：自己动手实现远程执行功能

看完原书并实践之后，再来补

参考

JVM笔记 – 虚拟机执行子系统（类加载及执行子系统的案例与实战）

早期（编译期）优化

概述

Java语言的“编译期”是一段不确定的操作过程，可能是：

- 前端编译器（编译器的前端）把Java文件转换为class文件；Sun 的 Javac、 Eclipse JDT 中的增量式编译器（ ECJ）。
- 后端编译器（JIT编译期 Just in time compiler）把字节码变成机器码；JIT 编译器： HotSpot VM 的 C1、 C2 编译器。
- 静态编译器（AOT编译器 ahead of time compiler）直接把Java编译成本地机器代码；
- AOT 编译器： GNU Compiler for the Java（GCJ）、Excelsior JET。

本章讨论第一类编译过程。

Javac 这类编译器对代码的运行效率几乎没有任何优化措施（在 JDK 1. 3 之后， Javac 的- O 优化参数就不再有意义）。虚拟机设计团队把对性能的优化集中到了后端的即时编译器中，这样可以让那些不是由javac产生的Class文件（如JRuby、Groovy等语言的Class文件）也同样能享受到编译器优化所带来的好处。但是Javac做了很多针对Java语言编码过程的优化措施来改善程序员的编码风格和提高编码效率。相当多新生的Java语法特性，都是靠编译器的“语法糖”来实现，而不是依赖虚拟机的低层改进来支持，可以说，Java中即时编译器在运行期的优化过程对于程序运行来说更重要，而前端编译器在编译期的优化过程对于程序编码来说关系更加密切。

Javac编译器

它本身就是一个由 Java 语言编写的程序，这为纯 Java 的程序员了解它的编译过程带来了很大的便利。

Javac的源码与调试

Javac的源码放在 `JDK_SRC_HOME/langtools/src/share/classes/com/sun/tools/javac` 中，除了JDK自身的API外，就只引用了 `JDK_SRC_HOME/langtools/src/share/classes/com/sun/*` 里面的代码，调试环境建立起来简单方便，因为基本上不需要处理依赖关系。

以Eclipse IDE环境为例，先建立一个名为“Compile_javac”的java工程，然后把 `JDK_SRC_HOME/langtools/src/share/classes/com/sun/*` 目录下的源文件全部复制到工程的源码目录中。

导入代码期间，源码文件“AnnotationProxyMaker.java”可能会提示“Access Restriction”，被Eclipse拒绝编译。这是由于Eclipse的JRE System Library中默认包含了一系列的代码访问规则（Access Rules），如果代码中引用了这些访问规则所禁止引用的类，就会提示这个错误。可以通过添加一条允许访问Jar包中所有类的访问规则来解决这个问题。

导入了Javac的源码后，就可以运行com.sun.tools.javac.Main的main方法来执行编译了，与命令行中使用的Javac的命令没有什么区别，编译的文件与参数在Eclipse的Debug Configurations面板中的Arguments页签中指定。

从Sun Javac的代码来看，编译过程大致可以分为3个过程，分别是：

1. 解析与填充符号表过程。
2. 插入式注解处理器的注解处理过程。
3. 分析与字节码生成过程。

Javac编译动作的入口是com.sun.tools.javac.main.JavaCompiler类，上述3个过程的代码逻辑集中在这个类的compile和Compile2方法中。

解析与填充符号表

内容太多以后再来补

注解处理器

提供了一组插入式注解处理器的标准 API 在编译期间对注解进行处理。

有了编译器注解处理的标准 API 后，我们的代码才有可能干涉编译器的行为，由于语法树中的任意元素，甚至包括代码注释都可以在插件之中访问到，所以通过插入式注解处理器实现的插件在功能上有很大的发挥空间。

在 Javac 源码中，插入式注解处理器的初始化过程是在 initPorcessAnnotations() 方法中完成的，而它的执行过程则是在processAnnotations() 方法中完成的。

语义分析与字节码生成

编译器获得了程序代码的抽象语法树表示，语法树能表示一个结构正确的源程序的抽象，但无法保证源程序是符合逻辑的。而语义分析的主要任务是对结构上正确的源程序进行上下文有关性质的审查，如进行类型审查。

是否合乎语义逻辑必须限定在具体的语言与具体的上下文环境之中才有意义。

标注检查 **Javac** 的编译过程中，语义分析过程分为标注检查以及数据及控制流分析两个步骤。

标注检查步骤检查的内容包括诸如变量使用前是否已被声明、变量与赋值之间的数据类型是否能够匹配等。在标注检查步骤中，还有一个重要的动作称为常量折叠。

由于编译期间进行了常量折叠，所以在代码里面定义" a= 1+ 2" 比起直接定义" a= 3"， 并不会增加程序运行期哪怕仅仅一个 CPU 指令的运算量。

数据及控制流分析

数据及控制流分析是对程序上下文逻辑更进一步的验证，它可以检查出诸如程序局部变量在使用前是否有赋值、方法的每条路径是否都有返回值、是否所有的受查异常都被正确处理了等问题。

将局部变量声明为 final， 对运行期是没有影响的，变量的不变性仅仅由编译器在编译期间保障。局部变量与字段（实例变量、类变量）是有区别的，它在常量池中没有 CONSTANT_Fieldref_info 的符号引用，自然就没有访问标志（ Access_Flags） 的信息。

解语法糖

在编译阶段还原回简单的基础语法结构，这个过程称为解语法糖。

解语法糖的过程由 desugar() 方法触发。

字节码生成

字节码生成是 Javac 编译过程的最后一个阶段

把前面各个步骤所生成的信息（语法树、符号表）转化成字节码写到磁盘中，编译器还进行了少量的代码添加和转换工作。例如，前面章节中多次提到的实例构造器 `< init >` ()方法和类构造器 `< clinit >` ()方法就是在这个阶段添加到语法树之中的

还有其他的一些代码替换工作用于优化程序的实现逻辑，如把字符串的加操作替换为 StringBuffer 或 StringBuilder。

完成了对语法树的遍历和调整之后，就会把填充了所有所需信息的符号表交给 com. sun. tools. javac. jvm. ClassWriter 类，由这个类的 writeClass() 方法输出字节码，生成最终的 Class 文件，到此为止整个编译过程宣告结束。

Java语法糖的味道

泛型与类型擦除

Java 语言中的泛型则不一样，它只在程序源码中存在，在编译后的字节码文件中，就已经替换为原来的原生类型（Raw Type， 也称为裸类型）了，并且在相应的地方插入了强制转型代码。

泛型擦除成相同的原生类型只是无法重载的其中一部分原因。

方法重载要求方法具备不同的特征签名，返回值并不包含在方法的特征签名之中，所以返回值不参与重载选择，但是在 Class 文件格式之中，只要描述符不是完全一致的两个方法就可以共存。也就是说，两个方法如果有相同的名称和特征签名，但返回值不同，那它们也是可以合法地共存于一个 Class 文件中的。

Signature、 LocalVariableTypeTable 等新的属性用于解决伴随泛型而来的参数类型的识别问题， Signature 是其中最重要的一项属性，它的作用就是存储一个方法在字节码层面的特征签名， 这个属性中保存的参数类型并不是原生类型，而是包括了参数化类型的信息。

由于 List ＜ String ＞和 List ＜ Integer ＞擦除后是同一个类型，我们只能添加两个并不需要实际使用到的返回值才能完成重载。

擦除法所谓的擦除，仅仅是对方法的 Code 属性中的字节码进行擦除，实际上元数据中还是保留了泛型信息，这也是我们能通过反射手段取得参数化类型的根本依据。

自动装箱、拆箱与遍历循环

遍历循环则把代码还原成了迭代器的实现，这也是为何遍历循环需要被遍历的类实现 Iterable 接口的原因。

包装类的“==”运算在不遇到算术运算的情况下不会自动拆箱，以及它们 equals() 方法不处理数据转型的关系，笔者建议在实际编码中尽量避免这样使用自动装箱与拆箱。

条件编译

Java 语言当然也可以进行条件编译，方法就是使用条件为常量的 if 语句。

实战：插入式注解处理器

此部分的只是日后补上！

本章小结

之所以把 Javac 这类将 Java 代码转变为字节码的编译器称做“前端编译器”，是因为它只完成了从程序到抽象语法树或中间字节码的生成，而在此之后，还有一组内置于虚拟机内部的“后端编译器”完成了从字节码生成本地机器码的过程，即前面多次提到的即时编译器或 JIT 编译器，这个编译器的编译速度及编译结果的优劣，是衡量虚拟机性能一个很重要的指标。

参考

JVM笔记 – 程序编译与代码优化（早期（编译期）优化

晚期（运行期）优化

概述

即时编译器并不是虚拟机必需的部分。

本章提及的编译器、即时编译器都是指 HotSpot 虚拟机内的即时编译器，虚拟机也是特指 HotSpot 虚拟机。

HotSpot虚拟机内的即时编译器

解释器与编译器

HotSpot 虚拟机中内置了两个即时编译器，分别称为 Client Compiler 和 Server Compiler。

HotSpot 虚拟机会根据自身版本与宿主机器的硬件性能自动选择运行模式，用户也可以使用"-client" 或"-server" 参数去强制指定虚拟机运行在 Client 模式或 Server 模式。

参数"-Xint" 强制虚拟机运行于“解释模式”（Interpreted Mode）。

参数"-Xcomp" 强制虚拟机运行于“编译模式”（Compiled Mode），这时将优先采用编译方式执行程序，但是解释器仍然要在编译无法进行的情况下介入执行过程。

为了在程序启动响应速度与运行效率之间达到最佳平衡， HotSpot 虚拟机还会逐渐启用分层编译（ Tiered Compilation） 的策略。

实施分层编译后， Client Compiler 和 Server Compiler 将会同时工作，许多代码都可能会被多次编译，用 Client Compiler 获取更高的编译速度，用 Server Compiler 来获取更好的编译质量，在解释执行的时候也无须再承担收集性能监控信息的任务。

编译对象与触发条件

“热点代码”有两类，即：被多次调用的方法。被多次执行的循环体。

这种编译方式因为编译发生在方法执行过程之中，因此形象地称之为栈上替换（ On Stack Replacement， 简称为 OSR 编译，即方法栈帧还在栈上，方法就被替换了）。

判断一段代码是不是热点代码，是不是需要触发即时编译，这样的行为称为热点探测。

目前主要的热点探测判定方式有两种：基于采样的热点探测，基于计数器的热点探测。

在 HotSpot 虚拟机中使用的是第二种——基于计数器的热点探测方法，因此它为每个方法准备了两类计数器：方法调用计数器（ Invocation Counter） 和回边计数器（ Back Edge Counter）。

当计数器超过阈值溢出了，就会触发 JIT 编译。

当编译工作完成之后，这个方法的调用入口地址就会被系统自动改写成新的。

使用虚拟机参数-XX:-UseCounterDecay 来关闭热度衰减，让方法计数器统计方法调用的绝对次数。

使用-XX:CounterHalfLifeTime 参数设置半衰周期的时间，单位是秒。

回边计数器，它的作用是统计一个方法中循环体代码执行的次数。

建立回边计数器统计的目的就是为了触发 OSR 编译。

参数-XX:OnStackReplacePercentage 来间接调整回边计数器的阈值。

编译过程

在默认设置下，无论是方法调用产生的即时编译请求，还是 OSR 编译请求，虚拟机在代码编译器还未完成之前，都仍然将按照解释方式继续执行，而编译动作则在后台的编译线程中进行。

用户可以通过参数-XX:-BackgroundCompilation 来禁止后台编译。

对于 Client Compiler 来说，它是一个简单快速的三段式编译器，主要的关注点在于局部性的优化，而放弃了许多耗时较长的全局优化手段。

而 Server Compiler 则是专门面向服务端的典型应用并为服务端的性能配置特别调整过的编译器,也是一个充分优化过的高级编译器,几乎能达到 GNU C++编译器使用-O2参数时的优化强度。

查看及分析即时编译结果

以后增加

编译优化技术

优化技术概览

这些代码优化变换是建立在代码的某种中间表示或机器码之上，绝不是建立在Java源码之上的。

公共子表达式消除

数组边界检查消除

除了如数组边界检查优化这种尽可能把运行期检查提到编译器完成的思路之外，另外还有一种避免思路：隐式异常处理。

当 foo 不为空的时候,对 value 的访问是不会额外消耗一次对 foo 判空的开销的。代价就是当 foo 真的为空时,必须转入到异常处理器中恢复并抛出 NullPointException异常，这个过程必须从用户态转动内核态中处理，结束后再回到用户态，速度远比一次判空检查慢。

方法内联

只有使用invokespecial指令调用的私有方法、实例构造器、父类方法以及使用invokestatic指令进行调用的静态方法才是在编译期进行解析的。

逃逸分析

逃逸分析的基本行为就是分析对象动态作用域。

如果确定一个方法不会逃逸出方法之外，那让整个对象在栈上分配内存将会是一个很不错的主意，对象所占用的内存空间就可以随栈帧而销毁。在一般应用中，不会逃逸的局部对象所占用的比例很大，如果能使用栈上分配，那大量的对象就会随着方法结束而自动销毁了，垃圾手机系统的压力将会小很多。

同步消除

标量替换

Java与C/C++的编译器对比

除了它们自身的API库实现得好坏之外，其余的比较就成了一场“拼编译器”和“拼输出代码质量”的游戏。

Java虚拟机的即时编译器与C/C++的静态优化编译器相比，可能会由于下列这些原因导致输出的本地代码有一些劣势：

即时编译器运行占用的是用户程序的运行时间

Java语言是动态的类型安全语言

Java语言中虽然没有virtual关键字，但是使用虚方法的频率却远远大于C/C++语言

Java语言是可以动态扩展的语言

Java语言中对象的内存分配都是在堆上进行的，只有方法中的局部变量才能在堆上分配

参考

JVM笔记 – 程序编译与代码优化（晚期（运行期）优化）

Java内存模型与线程

概述

在许多情况下，让计算机同时去做几件事，不仅是因为计算机的运算能力强大了，还有一个很重要的原因是计算机的运算速度与它的存储和通信子系统速度的差距太大，大量的时间花费在磁盘I/O、网络通信或者数据库访问上。

衡量一个服务性能的高低好坏，每秒事务处理数（Transactions Per Second， TPS）是最重要的指标之一，它代表着一秒内服务器端平均能响应的请求总数，而TPS值与程序的并发能力又有非常密切的关系。对于计算量相同的任务，程序线程并发协调得有条不紊，效率自然就会越高；反之，线程间频繁阻塞甚至死锁，将会大大降低程序的并发能力。

Java在高并发上做了很多事，但是了解其原理对于程序员是必要的。

硬件的效率与一致性

基于告诉缓存的存储交互很好地解决了处理器与内存的速度矛盾，但是也为计算机系统带来了更高的复杂度，因为它引入了一个新的问题：缓存一致性。

处理器可能会对输入代码进行乱序执行优化，处理器会在计算之后将乱序执行的结果重组，保证该结果与顺序执行的结果是一致的。

Java内存模型

Java虚拟机规范中试图定义一种Java内存模型来屏蔽掉各种硬件和操作系统的内存访问差异，以实现让Java程序在各种平台下都能达到一致性的内存访问效果。

线程的工作内存中保存了被该线程使用到的变量的主内存副本拷贝，线程对变量的所有操作（读取、赋值等）都必须在工作内存中进行。

主内存与工作内存

Java内存模型的主要目标是定义程序中各个变量的访问规则，即在虚拟机中将变量存储到内存和从内存中取出变量值这样的底层细节。此处的变量(Variable)与Java编译中所说的变量略有区别，它包括了实例字段，静态字段和构成数组对象的元素，但是不包括局部变量与方法

参数，因为后者是线程私有的，不会被共享，自然就不存在竞争的问题。了为获得比较好的执行效率，Java内存模型并没有限制执行引擎使用处理器的特定寄存器或缓存来和主内存进行交互，也没有限制即时编译器调整代码执行顺序这类权限。

Java内存模型规定了所有的变量都存储在主内存(Main Memory)中。每条线程还有自己的工作内存(Working Memory)，线程的工作内存中保存了被该线程使用到的变量的主内存副本拷贝，线程对变量的所有操作(读取，赋值等)都必须是工作内存中进行，而不能直接读写主内存中的变量。不同的线程之间也无法直接访问对方工作内存中的变量，线程间变量值的传递均需要通过主内存来完成，线程、主内存、工作内存三者的交互关系如下图：

图 12-2　线程、主内存、工作内存三者的交互关系（请与图 12-1 对比）

这里大家可能认为工作内存不是主内存的一部分么？其实不然虚拟机会优先将主内存存放在寄存器或者高速缓存块之中。

内存间交互操作

一个变量如何从主内存拷贝到工作内存，如何从工作内存同步回主内存之类的实现细节，Java内存模型中定义了以下8种操作来完成。这8种操作都是原子性的、不可再分的（对double和Long类型除外）。

1. lock(锁定)：作用于主内存变量，它把一个变量标识为一条线程独占的状态。
2. unlock(解锁)：作用于主内存变量，它把一个处理锁定的状态的变量释放出来,释放后的变量才可以被其它线程锁定，unlock之前必须将变量值同步回主内存。
3. read(读取)：作用于主内存变量，它把一个变量的值从主内存传输到线程的工作内存中，以便随后的load动作使用。
4. load(载入)：作用于工作内存变量，它把read操作从主内存中得到的值放入工作内存的变量副本中。
5. use(使用)：作用于工作内存中的变量，它把工作内存中一个变量的值传递给执行引擎，每当虚拟机遇到一个需要使用到变量的字节码指令时将会执行这个操作。
6. assign(赋值)：作用于工作内存变量，它把一个从执行引擎接到的值赋值给工作内存的变量，每当虚拟机遇到一个给变量赋值的字节码指令时执行这个操作。
7. store(存储)：作用于工作内存的变量，它把工作内存中一个变量的值传送到主内存中，以

便随后的write操作使用。

8. write(写入)：作用于主内存的变量，它把store操作从工作内存中得到的值放入主内存的变量中。

如果要把一个变量从主内存复制到工作内存，那就要顺序地执行read和load操作，如果要把变量从工作内存同步回主内存，就要顺序地执行store和write操作。Java内存模型只是要求上述两个操作必须按顺序执行，而没有保证必须是连续执行。也就是说read与load之间、store与write之间是可以插入其它指令的，如果对主在内中的变量a,b进行访问时，一种可能出现的顺序是read a、readb、loadb、load a。除此之外，Java内存模型还规定了执行上述八种基础操作时必须满足如下规则：

1. 不允许read和load、store和write操作之一单独出现，即不允许一个变量从主内存读取了但工作内存不接受，或者从工作内存发起回写但主内存不接受的情况出现。
2. 不允许一个线程丢弃它的最近的assign操作，即变量在工作内存中改变(为工作内存变量赋值)了之后必须把该变化同步回主内存。
3. 一个新变量只能在主内存中“诞生”，不允许在工作内存中直接使用一个未被初始化(load和assign)的变量，换话说就是一个变量在实施use和store操作之前，必须先执行过了assign和load操作。
4. 如果一个变量事先没有被load操作锁定，则不允许对它执行unlock操作：也不允许去unlock一个被其它线程锁定的变量。
5. 对一个变量执行unloack之前，必须把此变量同步回主内存中(执行store和write操作)

对于**volatile**型变量的特殊规则

关键字volatile可以说是Java虚拟机提供的最轻量级的同步机制，但是它并不容易完全被正确、完整地理解，以至于许多程序员都习惯不使用它，遇到竞争问题的时候一律使用synchronized来进行同步。

当一个变量定义成volatile之后，它将具备两种特性：第一是保证此变量对所有线程的可见性，这里的“可见性”是指当一条线程修改了这个变量的值，新值对于其它线程是可以立即得知的，变量值在线程间传递均需要通过主内存来完成，如：线程A修改一个普通变量的值，然后向主内存进行回写，另外一条线程B在线程A回写完成了之后再从主内存进行读取操作，新变量的值才会对线程B可见。

关于volatile变量的可见性，很多人误以为以下描述成立：“volatile对所有线程是立即可见的，对volatile变量所有的写操作都能立即返回到其它线程之中，换句话说，volatile变量在各个线程中是一致的，所以基于volatile变量的运算在并发下是安全的”。这句话的论据部分并没有错，但是其论据并不能得出“基于基于volatile变量的运算在并发下是安全的”这个结论。volatile变量在各个线程的工作内存中不存在一致性问题(在各个线程的工作内存中volatile变量也可以存在不一致的情况，但由于每次使用之前都要先刷新，执行引擎看不到不致的情况，因此可以认为不存在一致性问题)，但是Java里的运算并非原子操作，导致volatile变量的运算在并发下一样是不安全的。

由于volatile变量只能保证可见性，在不符合以下条件规则的去处场景中，仍然需要通过加锁来保证原子性。

1. 运算结果不依赖变量的当前值，或者能确保只有单一的线程改变变量的值。
2. 变量不需要与其它的状态变量共同参与不变约束。

使用volatile变量的第二个语义是禁止指令重排序优化，普通的变量仅仅会保证在该方法的执行过程中所有依赖赋值结果的地方能获取到正确的结果，而不能保证变量的赋值操作的顺序与程序代码中的执行顺序一致。因为在一个线程的方法执行过程中无法感知到这一点，这也就是Java内存模型中描述的所谓的”线程内表现为串行的语义“(Within-Thread As-If-Serial Sematics)。

奇怪的并发现象探究——JMM的指令重排、内存级指令重排

```
Map configOptions;
char[] configText;
//此变量必须定义为volatile
volatile boolean initialized = false;
//假设以下代码在线程A中执行
//模拟读取配置信息，当读取完成后
//将initialized设置为true来通知其它线程配置可用
configOptions = new HashMap();
configText = readConfigFile(fileName);
processConfigOptions(configText, configOptions);
initialized = true;

//假设以下代码在线程B中执行
//等线程A待initialized为true，代表线程A已经把配置信息初始化完成
while(!initialized) {
    sleep();
}
//使用线程A中初始化好的配置信息
doSomethingWithConfig();
```

上面为一段伪代码，其中描述的场景十分常见，只是我们在处理配置文件时一般不会出现并发而已。如果定义initialized变量时没有使用volatile修饰，就可能会由于指令重排序的优化，导致位于线程A中最后一句的代码”initialized = true“被提前执行，这样在线程B中使用配置信息的代码就可能出现错误，而volatile关键字则可以避免此类情况的发生。

volatile变量读取操作的性能消耗与普通变量几乎没有什么差别，但是写操作则可能会慢一些，因为它需要再笨的代码中插入许多内存屏障指令来保证处理器不发生乱序执行。不过即便如此，大多数场景下volatile的总开销仍然要比锁第，我们在volatile与锁之中选择的唯一依据仅仅是volatile的语义能否满足使用场景的需求。

如一个变量的修改不依赖与原值，则这个时候可以使用**volatile**关键字实现先行发生关系。

对于Long和double型变量的特殊规则

对于32位平台，64位的操作需要分两步来进行，与主存的同步。所以可能出现“半个变量”的状态。

在实际开发中，目前各种平台下的商用虚拟机几乎都选择把64位数据的读写操作作为原子操作来对待，因此我们在编码时一般不需要把用到的long和double变量专门声明为volatile。

原子性、可见性与有序性

Java内存模型是围绕着并发过程中如何处理原子性、可见性、有序性这三个特征来建立的，下面是这三个特性的实现原理：

1.原子性(Atomicity)

由Java内存模型来直接保证的原子性变量操作包括read、load、use、assign、store和write六个，大致可以认为基础数据类型的访问和读写是具备原子性的。如果应用场景需要一个更大范围的原子性保证，Java内存模型还提供了lock和unlock操作来满足这种需求，尽管虚拟机未把lock与unlock操作直接开放给用户使用，但是却提供了更高层次的字节码指令monitorenter和monitorexit来隐匿地使用这两个操作，这两个字节码指令反映到Java代码中就是同步块---synchronized关键字，因此在synchronized块之间的操作也具备原子性。

2.可见性(Visibility)

可见性就是指当一个线程修改了线程共享变量的值，其它线程能够立即得知这个修改。Java内存模型是通过在变量修改后将新值同步回主内存，在变量读取前从主内存刷新变量值这种依赖主内存作为传递媒介的方法来实现可见性的，无论是普通变量还是volatile变量都是如此，普通变量与volatile变量的区别是volatile的特殊规则保证了新值能立即同步到主内存，以及每使用前立即从内存刷新。因为我们可以说volatile保证了线程操作时变量的可见性，而普通变量则不能保证这一点。

除了volatile之外，Java还有两个关键字能实现可见性，它们是synchronized。同步块的可见性是由“对一个变量执行unlock操作之前，必须先把此变量同步回主内存中(执行store和write操作)”这条规则获得的，而final关键字的可见性是指：被final修饰的字段是构造器一旦初始化完成，并且构造器没有把“this”引用传递出去，那么在其它线程中就能看见final字段的值。

3.有序性(Ordering)

Java内存模型中的程序天然有序性可以总结为一句话：如果在本线程内观察，所有操作都是有序的；如果在一个线程中观察另一个线程，所有操作都是无序的。前半句是指“线程内表现为串行语义”，后半句是指“指令重排序”现象和“工作内存主主内存同步延迟”现象。

Java语言提供了volatile和synchronized两个关键字来保证线程之间操作的有序性，volatile关键字本身就包含了禁止指令重排序的语义，而synchronized则是由“一个变量在同一时刻只允许一条线程对其进行lock操作”这条规则来获得的，这个规则决定了持有同一个锁的两个同步块只能串行地进入。

先行发生原则

如果Java内存模型中所有的有序性都只靠volatile和synchronized来完成，那么有一些操作将会变得很啰嗦，但是我们在编写Java并发代码的时候并没有感觉到这一点，这是因为Java语言中有一个“先行发生”(Happen-Before)的原则。这个原则非常重要，它是判断数据是否存在竞争，线程是否安全的主要依赖。

先行发生原则是指Java内存模型中定义的两项操作之间的依序关系，如果说操作A先行发生于操作B，其实就是说发生操作B之前，操作A产生的影响能被操作B观察到，“影响”包含了修改了内存中共享变量的值、发送了消息、调用了方法等。它意味着什么呢？如下例：

```
//线程A中执行
i = 1;

//线程B中执行
j = i;

//线程C中执行
i = 2;
```

下面是Java内存模型下一些”天然的“先行发生关系，这些先行发生关系无须任何同步器协助就已经存在，可以在编码中直接使用。如果两个操作之间的关系不在此列，并且无法从下列规则推导出来的话，它们就没有顺序性保障，虚拟机可以对它们进行随意地重排序。

1. 程序次序规则(Pragram Order Rule)：在一个线程内，按照程序代码顺序，书写在前面的操作先行发生于书写在后面的操作。准确地说应该是控制流顺序而不是程序代码顺序，因为要考虑分支、循环结构。
2. 管程锁定规则(Monitor Lock Rule)：一个unlock操作先行发生于后面对同一个锁的lock操作。这里必须强调的是同一个锁，而”后面“是指时间上的先后顺序。
3. volatile变量规则(Volatile Variable Rule)：对一个volatile变量的写操作先行发生于后面对这个变量的读取操作，这里的”后面“同样指时间上的先后顺序。
4. 线程启动规则(Thread Start Rule)：Thread对象的start()方法先行发生于此线程的每一个动作。
5. 线程终止规则(Thread Termination Rule)：线程中的所有操作都先行发生于对此线程的终止检测，我们可以通过Thread.join()方法结束，Thread.isAlive()的返回值等作段检测到线程已经终止执行。
6. 线程中断规则(Thread Interruption Rule)：对线程interrupt()方法的调用先行发生于被中断线程的代码检测到中断事件的发生，可以通过Thread.interrupted()方法检测是否有中断发

生。

7. 对象终结规则(Finalizer Rule)：一个对象初始化完成(构造方法执行完成)先行发生于它的finalize()方法的开始。
8. 传递性(Transitivity)：如果操作A先行发生于操作B，操作B先行发生于操作C，那就可以得出操作A先行发生于操作C的结论。

一个操作"时间上的先发生"不代表这个操作会是"先行发生"，那如果一个操作"先行发生"是否就能推导出这个操作必定是"时间上的先发生"呢？也是不成立的，一个典型的例子就是指令重排序。所以时间上的先后顺序与先生发生原则之间基本没有什么关系，所以衡量并发安全问题一切必须以先行发生原则为准。

Java与线程

并发不一定要依赖多线程（如PHP中很常见的多进程并发），但在Java中，谈论并发，大多数都与线程脱不开关系。

线程的实现

线程时CPU的最小调度单位。

Thread类与大部分的Java API有显著的差别，它的所有关键方法都是声明为Native的。一个Native方法往往意味着这个方法没有使用或无法使用平台无关的手段来实现。

实现线程主要有3种方式：使用内核线程实现、使用用户线程实现和使用用户线程加轻量级进程混合实现。

使用内核线程实现

由于是基于内核线程实现的，所以各种线程操作，如创建、析构及同步，都需要进行系统调用。而系统调用的代价相对较高，需要在用户态和内核态中来回切换。其次，每个轻量级进程都需要有一个内核线程的支持，因此轻量级进程要消耗一定的内核资源（如内核线程的栈空间），因此一个系统支持轻量级进程的数量是有限的。

使用用户线程实现

已被java、ruby等语言放弃

使用用户线程加轻量级进程混合实现

Java线程的实现

虚拟机自己选择，有的加了虚拟机参数实现可选择。

Java线程调度

分为协同式线程调度和抢占式线程调度。Java使用的线程调度方式就是抢占式调度。

线程优先级并不是太靠谱，原因是Java的线程是通过映射到系统的原生线程上来实现的，所以线程调度最终还是取决于操作系统。

状态转换

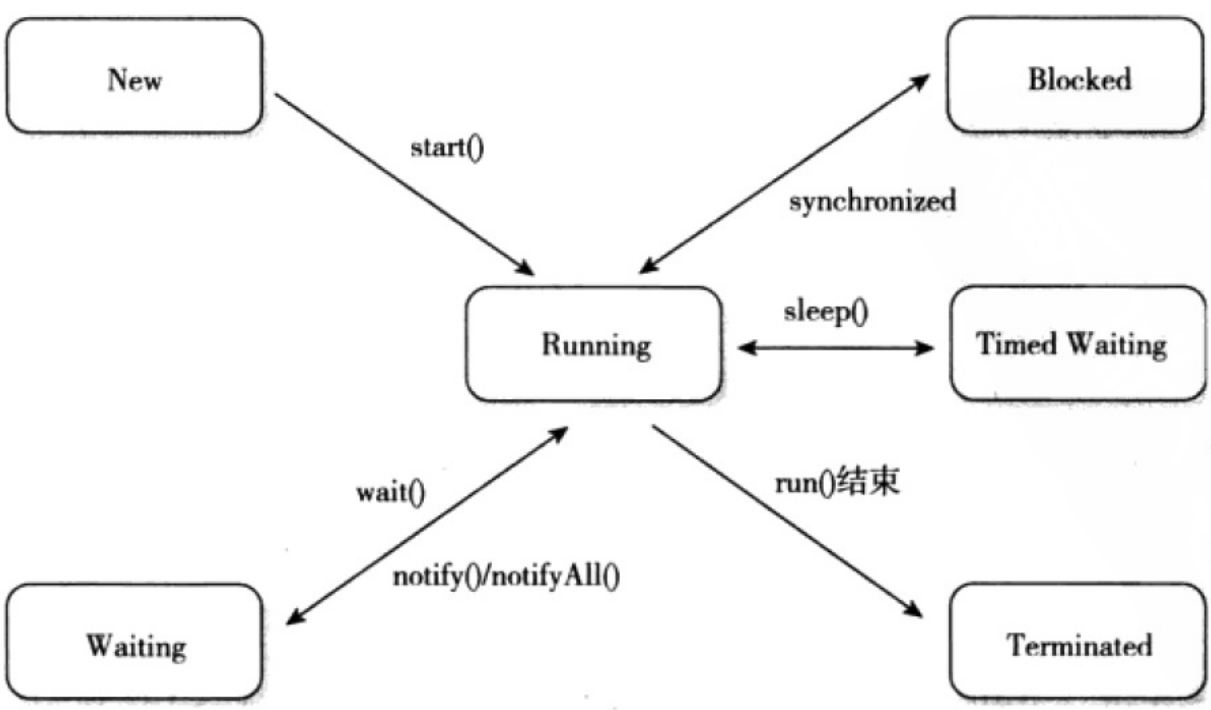


图 12-6　线程状态转换关系

以下方法会让线程陷入无限期的等待状态：

- 没有设置Timeout参数的Object.wait()方法
- 没有设置Timeout参数的Thread.join()方法
- LockSupport.park()方法

以下方法会让线程进入限期等待状体：

- Thread.sleep()
- 设置了Timeout参数的Object.wait()方法
- 设置了Timeout参数的Thread.join()方法
- LockSupport.parkNanos()方法
- LockSupport.parkUntil()方法

关于线程，可以查看多线程的实现方法

线程安全与锁优化

线程安全

当多个线程访问一个对象时，如果不用考虑这些线程在运行环境下的调度和交替执行，也不需要进行额外的同步，或者在调用方进行任何其他的协调操作，调用这个对象的行为都可以获得正确的结果，那么这个对象时线程安全的。

Java语言中的线程安全

按照线程安全的“安全程度”由强至弱来排序，我们可以将Java语言中各种操作共享的数据分为以下5类：不可变、绝对线程安全、相对线程安全、线程建荣和线程对立。

不可变

不可变的对象一定是线程安全的。

保证对象行为不影响自己状态的途径有很多种，其中最简单的就是把对象中带有状态的变量都声明为final。

每次对对象的操作都会返回一个新的对象，而不会去改变旧的对象，这样可以保证对象的线程安全。

Java API中符合不可变要求的类型：String，java.lang.Number的部分子类（如Long和Double的数值包装类，BigInteger和BigDecimal等大数据类型但 `AtomicInteger` 和 `AtomicLong` 则并非不可变的）

绝对线程安全

Java API中标注自己是线程安全的类，大多数都不是绝对线程安全的。

相对线程安全

相对线程安全就是我们通常意义上所讲的线程安全，它需要保证对这个对象单独的操作是线程安全的，我们在调用的时候不需要做额外的保障措施，但是对于一些特定顺序的连续调用，就可能需要在调用端使用额外的同步手段来保证调用的正确性。

Java语言中，大部分的线程安全都属于这种类型，例如Vector，HashTable，Collections的synchronizedCollection()方法包装的集合等。

线程兼容

指对象本身并不是线程安全的，但是通过使用同步手段来保证对象在并发环境中可以安全的使用。Java API中大部分的类都是属于线程兼容的，如ArrayList和HashMap。

线程对立

指无论调用端是否采取了同步措施，都无法在多线程环境中并发使用的代码。

一个线程对立的例子就是Thread类的suspend()和resumn()方法（已被JDK声明废弃了）。

常见的线程对立操作还有System.setIn(), System.setOut(), System.runFinalizersOnExit()等等。

线程安全的实现方法

虚拟机提供的同步和锁机制起到了非常重要的作用。

互斥同步

互斥是实现同步的一种手段，临界区、互斥量和信号量都是主要的互斥实现方式。因此在这4个字里面，互斥是因，同步是果：互斥是方法，同步是目的。

Java中，最基本的互斥同步手段就是synchronized关键字。通过锁计数器+-1，实现对锁的加锁和释放。

synchronized是一个重量级的操作，因为：Java的线程是映射到操作系统的原生线程之上的，如果要阻塞或唤醒一个线程，都需要操作系统来帮忙完成，这就需要从用户态转换到核心态中，因此状态转换需要消耗很多的处理器时间。对于代码简单的同步块（如synchronized修饰的getter()和setter方法），状态转换消耗的时间有可能比用户代码执行的时间还要长。而虚拟机本身也会进行一些优化，比如在通知操作系统阻塞线程之前加入一段自旋等待过程，避免频繁地切入到核心态之中。

还可以使用java.util.concurrent包中的 `ReentrantLock(重入锁)` 来实现同步：JDK1.5多线程环境下synchronized的吞吐量下降的很严重，而ReentrantLock则基本保持在同一个比较稳定的水平上。JDK 1.6之后两者性能基本持平。

虚拟机在未来的性能改进中还会更偏向于原生的synchronize的，所以还是提倡在synchronized能实现需求的情况下，优先考虑使用synchronized来进行同步。

非阻塞同步

非阻塞同步：从处理问题的方式上说，互斥同步属于一种悲观的并发策略。随着硬件指令集的发展，我们可以采用基于冲突检查的乐观并发策略，通俗地说，就是先行操作，如果没有其他线程争用共享数据，那操作就成功了；如果共享数据有争用，产生了冲突，那就再采取其他的补偿措施（最常见的补偿措施就是不断地重试，直到成功为止），这种乐观的并发策略的许多实现偶读不需要把线程挂起，因此这话总同步操作称为非阻塞同步。

非阻塞同步算法实战（一）

非阻塞同步算法实战（二）—BoundlessCyclicBarrier

非阻塞同步算法实战（三）—LatestResultsProvider

无同步方案

如果一个方法本来就不设计共享数据，那它自然就无须任何同步措施去保证正确性，因此会有一些代码天生就是线程安全的。这类代码包括：可重入代码和线程本地存储。

可重入代码(Reentry code)也叫纯代码(Pure code)是一种允许多个进程同时访问的代码。为了使各进程所执行的代码完全相同，故不允许任何进程对其进行修改。程序在运行过程中可以被打断，并由开始处再次执行，并且在合理的范围内（多次重入，而不造成堆栈溢出等其他问题），程序可以在被打断处继续执行，且执行结果不受影响。

可重入代码有一些共同的特征。例如不依赖存储在堆上的数据和公用的系统资源、用到的状态量都由参数中传入、不调用非可重入的方法等。我们可以通过一个简单的原则来判断代码是否具备可重入性：如果一个方法，它的返回结果是可以预测的，只要输入了相同的数据，就都能返回相同的结果，那它就满足可重入性的要求，当然也就是线程安全的。

线程本地存储：TLS--线程局部存储 这个可以用ThreadLocal来设置。

锁优化

为了进一步改进高效并发，HotSpot虚拟机开发团队在JDK1.6版本上花费了大量精力实现各种锁优化。如适应性自旋、锁消除、锁粗化、轻量级锁和偏向锁等。

Java锁的种类以及辨析

自旋锁与自适应自旋

为了让线程等待，我们只需要让线程执行一个忙循环（自旋），这项技术就是所谓的自旋锁。引入自旋锁的原因是互斥同步对性能最大的影响是阻塞的实现，管钱线程和恢复线程的操作都需要转入内核态中完成，给并发带来很大压力。自旋锁让物理机器有一个以上的处理

器的时候，能让两个或以上的线程同时并行执行。我们就可以让后面请求锁的那个线程“稍等一下”，但不放弃处理器的执行时间，看看持有锁的线程是否很快就会释放锁。为了让线程等待，我们只需让线程执行一个忙循环（自旋），这项技术就是所谓的自旋锁。

自旋锁的自旋次数默认值是10次，用户可以使用参数-XX:PreBlockSpin来更改。

JDK1.6引入了自适应的自旋锁。自适应意味着自旋的时间不再固定了，而是由前一次在同一个锁上的自旋时间及锁的拥有者的状态来决定。（这个应该属于试探性的算法）。

java锁的种类以及辨析（一）：自旋锁

锁消除

锁消除是指虚拟机即时编译器在运行时，对一些代码上要求同步，但是被检测到不可能存在共享数据竞争的锁进行清除。锁清除的主要判定依据来源于逃逸分析的数据支持，如果判断在一段代码中，堆上的所有数据都不会逃逸出去从而被其他线程访问到，那就可以把它们当做栈上数据对待，认为它们是线程私有的，同步枷锁自然就无需进行。

java 多线程的锁消除

锁粗化

另一种线程优化方式是锁粗化（或合并，merging）。当多个彼此靠近的同步块可以合并到一起，形成一个同步块的时候，就会进行锁粗化。该方法还有一种变体，可以把多个同步方法合并为一个方法。如果所有方法都用一个锁对象，就可以尝试这种方法。

轻量级锁

Java 轻量级锁原理详解(Lightweight Locking)

偏向锁

大多数锁，在它们的生命周期中，从来不会被多于一个线程所访问。即使在极少数情况下，多个线程真的共享数据了，锁也不会发生竞争。为了理解偏向锁的优势，我们首先需要回顾一下如何获取锁（监视器）。

获取锁的过程分为两部分。首先，你需要获得一份契约.一旦你获得了这份契约，就可以自由地拿到锁了。为了获得这份契约，线程必须执行一个代价昂贵的原子指令。释放锁同时就要释放契约。根据我们的观察，我们似乎需要对一些锁的访问进行优化，比如线程执行的同步块代码在一个循环体中。优化的方法之一就是将锁粗化，以包含整个循环。这样，线程只访问一次锁，而不必每次进入循环时都进行访问了。但是，这并非一个很好的解决方案，因为它可能会妨碍其他线程合法的访问。还有一个更合理的方案，即将锁偏向给执行循环的线程。

将锁偏向于一个线程，意味着该线程不需要释放锁的契约。因此，随后获取锁的时候可以不那么昂贵。如果另一个线程在尝试获取锁，那么循环线程只需要释放契约就可以了。Java 6的HotSpot/JIT默认情况下实现了偏向锁的优化。

JVM内部细节之二：偏向锁（Biased Locking）

HotSpot虚拟机主要参数表

本参数表以JDK 1.6为基础编写，JDK 1.6的HotSpot虚拟机有很多非稳定参数（Unstable Options，即以-XX：开头的参数，JDK 1.6的虚拟机中大概有660多个），使用-XX：+PrintFlagsFinal参数可以输出所有参数的名称及默认值（默认不包括Diagnostic和Experimental的参数，如果需要，可以配合-XX：+UnlockDiagnosticVMOptions/-XX：+UnlockExperimentalVMOptions一起使用），下面的各个表格只包含了其中最常用的部分。参数使用的方式有如下3种：

- -XX：+＜option＞开启option参数。
- -XX：-＜option＞关闭option参数。
- -XX：＜option＞=＜value＞将option参数的值设置为value。

内存管理参数

参数	默认值	使用介绍
DisableExplicitGC	默认关闭	忽略来自 System.gc() 方法触发的垃圾收集
ExplicitGCInvokes Concurrent	默认关闭	当收到 System.gc() 方法提交的垃圾收集申请时，使用 CMS 收集器进行收集
UseSerialGC	Client 模式的虚拟机默认开启，其他模式关闭	虚拟机运行在 Client 模式下的默认值，打开此开关后，使用 Serial+Serial Old 的收集器组合进行内存回收
UseParNewGC	默认关闭	打开此开关后，使用 ParNew+Serial Old 的收集器组合进行内存回收
UseConcMarkSweepGC	默认关闭	打开此开关后，使用 ParNew+CMS+Serial Old 的收集器组合进行内存回收。如果 CMS 收集器出现 Concurrent Mode Failure，则 Serial Old 收集器将作为后备收集器
UseParallelGC	Server 模式的虚拟机默认开启，其他模式关闭	虚拟机运行在 Server 模式下的默认值，打开此开关后，使用 Parallel Scavenge+Serial Old 的收集器组合进行内存回收

（续）

参数	默认值	使用介绍
UseParallelOldGC	默认关闭	打开此开关后，使用 Parallel Scavenge + Parallel Old 的收集器组合进行内存回收
SurvivorRatio	默认为 8	新生代中 Eden 区域与 Survivor 区域的容量比值
PretenureSizeThreshold	无默认值	直接晋升到老年代的对象大小，设置这个参数后，大于这个参数的对象将直接在老年代分配
MaxTenuringThreshold	默认值为 15	晋升到老年代的对象年龄。每个对象在坚持过一次 Minor GC 之后，年龄就增加 1，当超过这个参数值时就进入老年代
UseAdaptiveSizePolicy	默认开启	动态调整 Java 堆中各个区域的大小及进入老年代的年龄
HandlePromotionFailure	JDK 1.5 及以前版本默认关闭，JDK 1.6 默认开启	是否允许分配担保失败，即老年代的剩余空间不足以应付新生代的整个 Eden 和 Survivor 区的所有对象都存活的极端情况
ParallelGCThreads	少于或等于 8 个 CPU 时默认值为 CPU 数量值，多于 8 个时比 CPU 数量值小	设置并行 GC 时进行内存回收的线程数
GCTimeRatio	默认值为 99	GC 时间占总时间的比率，默认值为 99，即允许 1% 的 GC 时间。仅在使用 Parallel Scavenge 收集器时生效
MaxGCPauseMillis	无默认值	设置 GC 的最大停顿时间。仅在使用 Parallel Scavenge 收集器时生效
CMSInitiatingOccupancy Fraction	默认值为 68	设置 CMS 收集器在老年代空间被使用多少后触发垃圾收集，仅在使用 CMS 收集器时生效
UseCMSCompactAtFull Collection	默认开启	设置 CMS 收集器在完成垃圾收集后是否要进行一次内存碎片整理。仅在使用 CMS 收集器时生效
CMSFullGCsBefore Compaction	无默认值	设置 CMS 收集器在进行若干次垃圾收集后再启动一次内存碎片整理。仅在使用 CMS 收集器时生效
ScavengeBeforeFullGC	默认开启	在 Full GC 发生之前触发一次 Minor GC
UseGCOverheadLimit	默认开启	禁止 GC 过程无限制地执行，如果过于频繁，就直接发生 OutOfMemory 异常
UseTLAB	Server 模式默认开启	优先在本地线程缓冲区中分配对象，避免分配内存时的锁定过程

MaxHeapFreeRatio	默认值为 70	当 Xmx 值比 Xms 值大时，堆可以动态收缩和扩展，这个参数控制当堆空闲大于指定比率时自动收缩
MinHeapFreeRatio	默认值为 40	当 Xmx 值比 Xms 值大时，堆可以动态收缩和扩展，这个参数控制当堆空闲小于指定比率时自动扩展
MaxPermSize	大部分情况下默认值是 64MB	永久代的最大值

即时编译参数

参数	默认值	使用介绍
CompileThreshold	Client 模式下默认值是 1500，Server 模式下是 10000	触发方法即时编译的阈值
OnStackReplacePercentage	Client 模式下默认值是 933，Server 模式下是 140	OSR 比率，它是 OSR 即时编译阈值计算公式的一个参数，用于代替 BackEdgeThreshold 参数控制回边计数器的实际溢出阈值
ReservedCodeCacheSize	大部分情况下默认值是 32MB	即时编译器编译的代码缓存的最大值

类型加载参数

参数	默认值	使用介绍
UseSplitVerifier	默认开启	使用依赖 StackMapTable 信息的类型检查代替数据流分析，以加快字节码校验速度
FailOverToOldVerifier	默认开启	当类型校验失败时，是否允许回到老的类型推导校验方式进行校验，如果开启则允许
RelaxAccessControlCheck	默认关闭	在校验阶段放松对类型访问性的限制

多线程相关参数

参数	默认值	使用介绍
UseSpinning	JDK 1.6 默认开启，JDK 1.5 默认关闭	开启自旋锁以避免线程频繁挂起和唤醒
PreBlockSpin	默认值为 10	使用自旋锁时默认的自旋次数
UseThreadPriorities	默认开启	使用本地线程优先级
UseBiasedLocking	默认开启	是否使用偏向锁，如果开启则使用
UseFastAccessorMethods	默认开启	当频繁反射执行某个方法时，生成字节码来加快反射的执行速度

性能参数

参数	默认值	使用介绍
AggressiveOpts	JDK 1.6 默认开启，JDK 1.5 默认关闭	使用激进的优化特性，这些特性一般是具备正面和负面双重影响的，需要根据具体应用特点分析才能判定是否对性能有好处
UseLargePages	默认开启	如果可能，使用大内存分页，这项特性需要操作系统的支持

（续）

参数	默认值	使用介绍
LargePageSizeInBytes	默认为 4MB	使用指定大小的内存分页，这项特性需要操作系统的支持
StringCache	默认开启	是否使用字符串缓存，开启则使用

调试参数

参数	默认值	使用介绍
HeapDumpOnOutOf MemoryError	默认关闭	在发生内存溢出异常时是否生成堆转储快照，关闭则不生成
OnOutOfMemoryError	无默认值	当虚拟机抛出内存溢出异常时，执行指定的命令
OnError	无默认值	当虚拟机抛出 ERROR 异常时，执行指定的命令
PrintClassHistogram	默认关闭	使用 [ctrl]-[break] 快捷键输出类统计状态，相当于 jmap-histo 的功能
PrintConcurrentLocks	默认关闭	打印 J.U.C 中锁的状态
PrintCommandLineFlags	默认关闭	打印启动虚拟机时输入的非稳定参数
PrintCompilation	默认关闭	打印方法即时编译信息
PrintGC	默认关闭	打印 GC 信息
PrintGCDetails	默认关闭	打印 GC 的详细信息
PrintGCTimeStamps	默认关闭	打印 GC 停顿耗时
PrintTenuringDistribution	默认关闭	打印 GC 后新生代各个年龄对象的大小
TraceClassLoading	默认关闭	打印类加载信息
TraceClassUnloading	默认关闭	打印类卸载信息
PrintInlining	默认关闭	打印方法的内联信息
PrintCFGToFile	默认关闭	将 CFG 图信息输出到文件，只有 DEBUG 版虚拟机才支持此参数
PrintIdealGraphFile	默认关闭	将 Ideal 图信息输出到文件，只有 DEBUG 版虚拟机才支持此参数
UnlockDiagnosticVM Options	默认关闭	让虚拟机进入诊断模式，一些参数（如 PrintAssembly）需要在诊断模式中才能使用
PrintAssembly	默认关闭	打印即时编译后的二进制信息

jvm HotSpot虚拟机主要参数表

虚拟机学习系列 - 附 - OQL（对象查询语言）