[

Java千百问_07JVM架构（001）_java内存模型是什么样的

,

点击进入_更多_Java千百问

1、什么是内存模型

Java平台自动集成了线程以及多处理器技术，这种集成程度比Java以前诞生的计算机语言要厉害很多。Java针对多种异构平台的独立性，使得多线程技术也具有了开拓性的一面。

了解线程和进程看这里：线程和进程有什么区别

我们有时候在Java开发中，对于同步和线程安全要求很严格的程序时，往往容易混淆的一个概念就是内存模型。那究竟什么是内存模型呢？

内存模型描述了程序中各个变量（实例域、静态域和数组元素）之间的关系，以及在实际计算机系统中将变量存储到内存、从内存中取出变量这样的底层细节。

Java对象最终是存储在内存里面的，这点没错，但是编译器、运行库、处理器或者系统缓存有权指定内存位置来存储或者取出变量的值。

2、内存模型有哪些规则

内存模型需要具有以下规则：原子性（Atomicity）、可见性（Visibility）、可排序性（Ordering）

1. 原子性（Atomicity）

原子性指的是原子级别的操作，比如最小的一块内存的读写操作，可以理解为Java语言编译过后最接近内存的最底层的操作单元。这种读写操作的数据单元不是变量的值，而是本机码。

原子性规则约定了：访问存储单元内任何类型字段的值以及对其更新操作时，除了long类型和double类型，其他类型的字段是必须要保证其原子性的，这些字段也包括为对象服务的引用。即，如果你获得或者初始化某一些值时（这些值是由其他线程写入的，而且不是从两个或者多个线程在同一时间戳混合写入的），该值的原子性在JVM内部是必须得到保证的。

此外，原子性扩展规则可以延伸到基于long和double的另外两种类型：volatile long和volatile double（volatile为java关键字），没有被volatile声明的long类型以及double类型的字段值虽然不保证其JMM中的原子性，但是是被允许的。

只要在不违反该规则的情况下，JVM并不关心数据的值来自什么线程，正这样使得Java语言在并行运算的设计中，针对多线程的原子性设计变得极其简单。即使开发人员没有考虑到，最终的程序也没有太大的影响。

2. 可见性（Visibility）

可见性指的是一个线程修改的状态对另一个线程是可见的。也就是说一个线程修改的结果，另一个线程马上就能看到。比如：用volatile修饰的变量，就会具有可见性。volatile修饰的变量不允许线程内部缓存和重排序，即直接修改内存，所以对其他线程是可见的。但是这里需要注意一个问题，volatile只能让被他修饰内容具有可见性，但不能保证它具有原子性。比如 volatile int a = 0；之后有一个操作a++；这个变量a具有可见性，但是a++依然是一个非原子操作，也就这这个操作同样存在线程安全问题。

在可见性规则的约束下，定义了一个线程在哪种情况下可以访问或者影响另外一个线程，以及从另外一个线程的可见区域读取相关数据、将数据写入到另外一个线程内。

3. 可排序性（Ordering）

可排序性是指为了提高性能，编译器和处理器可能会对指令做重排序，包括：

- 编译器优化的重排序
  编译器在不改变单线程程序语义的前提下，可以重新安排语句的执行顺序。

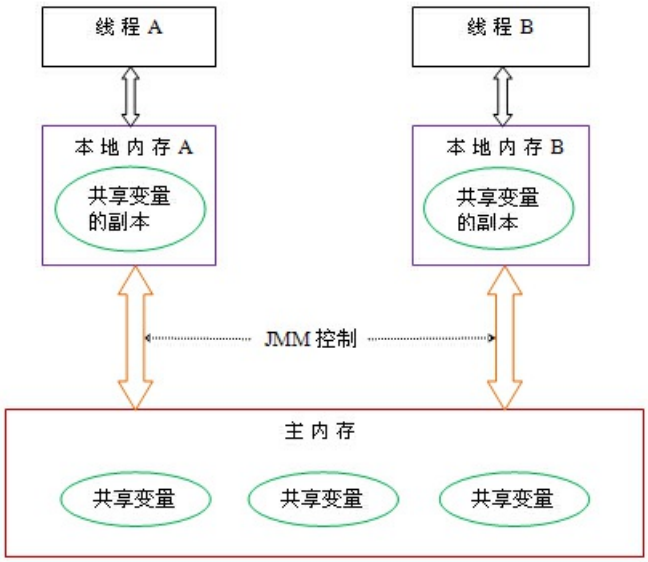
- 指令级并行的重排序
  现代处理器采用了指令级并行技术（Instruction-LevelParallelism,ILP）来将多条指令重叠执行。如果不存在数据依赖性，处理器可以改变语句对应机器指令的执行顺序。

- 内存系统的重排序
  由于处理器使用缓存和读/写缓冲区，这使得加载和存储操作看上去可能是在乱序执行。

volatile修饰的变量不允许线程内部缓存和重排序。
可排序性规则将会约束任何一个违背了规则调用的线程在操作过程中的一些顺序，排序问题主要围绕了读取、写入和赋值语句有关的序列。

3、Java内存模型是什么

JMM（Java内存模型，Java Memory Model的缩写）是控制Java线程之间、线程和主存之间通信的协议。
JMM定义了线程和主内存之间的抽象关系：线程之间的共享变量存储在主内存（main memory）中，每个线程都有一个私有的本地内存（local memory），本地内存中存储了该线程以读/写共享变量的副本。本地内存是JMM的一个抽象概念，并不真实存在。它涵盖了缓存，写缓冲区，寄存器以及其他的硬件和编译器优化。
JMM结构如下：



]

[

Java千百问_07JVM架构（002）_jvm实例的结构是什么样的

,

点击进入_更多_Java千百问

1、jvm实例的结构是什么样的

在Java虚拟机规范中，一个虚拟机实例的行为主要组成部分为：子系统、内存区域、数据类型和指令。
这些组件描述了JVM内部的一个抽象结构。与其说这些组成部分的目的是进行JVM内部结构的一种支配，不如说是提供一种对外部行为的严格定义，该规范定义了这些抽象组成部分的相互作用，以及Java虚拟机执行所需要的行为。

了解jvm内存管理看这里：java内存模型是什么样的
下图描述了JVM实例的一个内部结构，其中主要包括主要的子系统、内存区域。

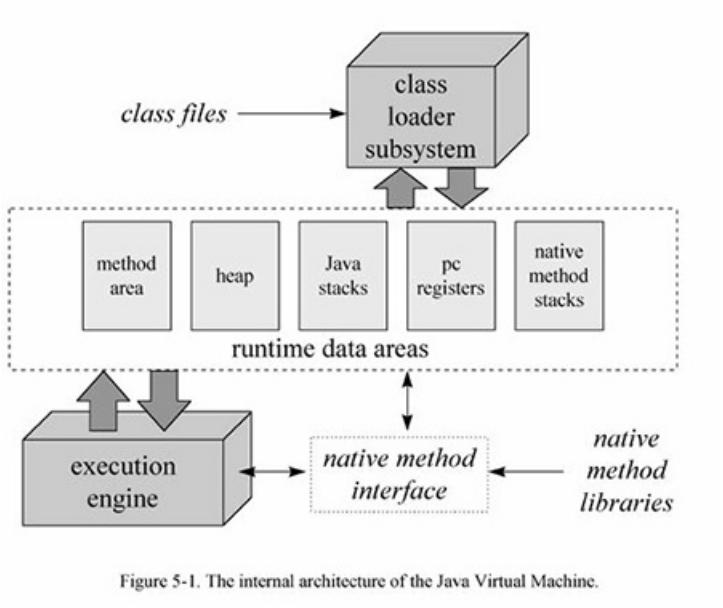


Figure 5-1. The internal architecture of the Java Virtual Machine.

我们来解读一下这个JVM实例的结构图：

1. Java虚拟机有一个类加载器（class loader subsystem）作为JVM的子系统，类加载器针对Class文件进行检测来加载对应的类接口。
2. JVM内部有一个执行引擎（exection engine），用来负责代码的解释和执行。
3. 当JVM运行程序的时候，它的内存用来存储包括字节码、从类文件中提取出来的一些附加信息、以及程序中实例化的对象、方法参数、返回值、局部变量以及计算的中间结果等很多内容。
4. JVM的内存管理分为两部分：每个JVM实例有一个方法区和一个内存堆，这两个区域储存了素有线程的共享数据；每个新的线程启动后，都会被分配自己的PC寄存器（PC registers，程序计数器器）、本地方法栈空间（native method stacks）、栈空间（Java stacks），来储存线程内部的数据。

了解内存分配策略看这里：内存分配有哪些策略
了解jvm是如何管理内存的看这里：jvm是如何管理内存的

]

[

Java千百问_07JVM架构（003）_内存分配有哪些策略

,

点击进入_更多_Java千百问

1、内存分配有哪些策略

我们从编译原理讲起，不同的开发环境、开发语言都会有不同的策略。一般来说，程序运行时有三种内存分配策略：静态的、栈式的、堆式的

- 静态存储
  是指在编译时就能够确定每个数据目标在运行时的存储空间需求，因而在编译时就可以给它们分配固定的内存空间。
  这种分配策略要求程序代码中不允许有可变数据结构的存在，也不允许有嵌套或者递归的结构出现，因为它们都会导致编译程序无法计算准确的存储空间。

- 栈式存储
  栈式存储分配是动态存储分配，是由一个类似于堆栈的运行栈来实现的，和静态存储的分配方式相反。
  在栈式存储方案中，程序对数据区的需求在编译时是完全未知的，只有到了运行的时候才能知道，但是规定在运行中进入一个程序模块的时候，必须知道该程序模块所需要的数据区的大小才能分配其内存。和我们在数据结构中所熟知的栈一样，栈式存储分配按照先进后出的原则进行分配。

- 堆式存储
  堆式存储分配专门负责在编译时或运行时，无法确定存储要求的数据结构的内存分配。
  比如可变长度串和对象实例，堆由大片的可利用块或空闲块组成，堆中的内存可以按照任意顺序分配和释放。

2、java内存分配策略

java的内存分配主要是以堆栈为主，具体如下：

了解java内存模型看这里：java内存模型是什么样的
了解jvm实例结构看这里：jvm实例的结构是什么样的
了解jvm如何管理内存看这里：jvm是如何管理内存的

]

[

Java千百问_07JVM架构（004）_jvm是如何管理内存的

,

点击进入_更多_Java千百问

1、JVM是如何管理内存的

Java中，内存管理是JVM自动进行的，无需人为干涉。

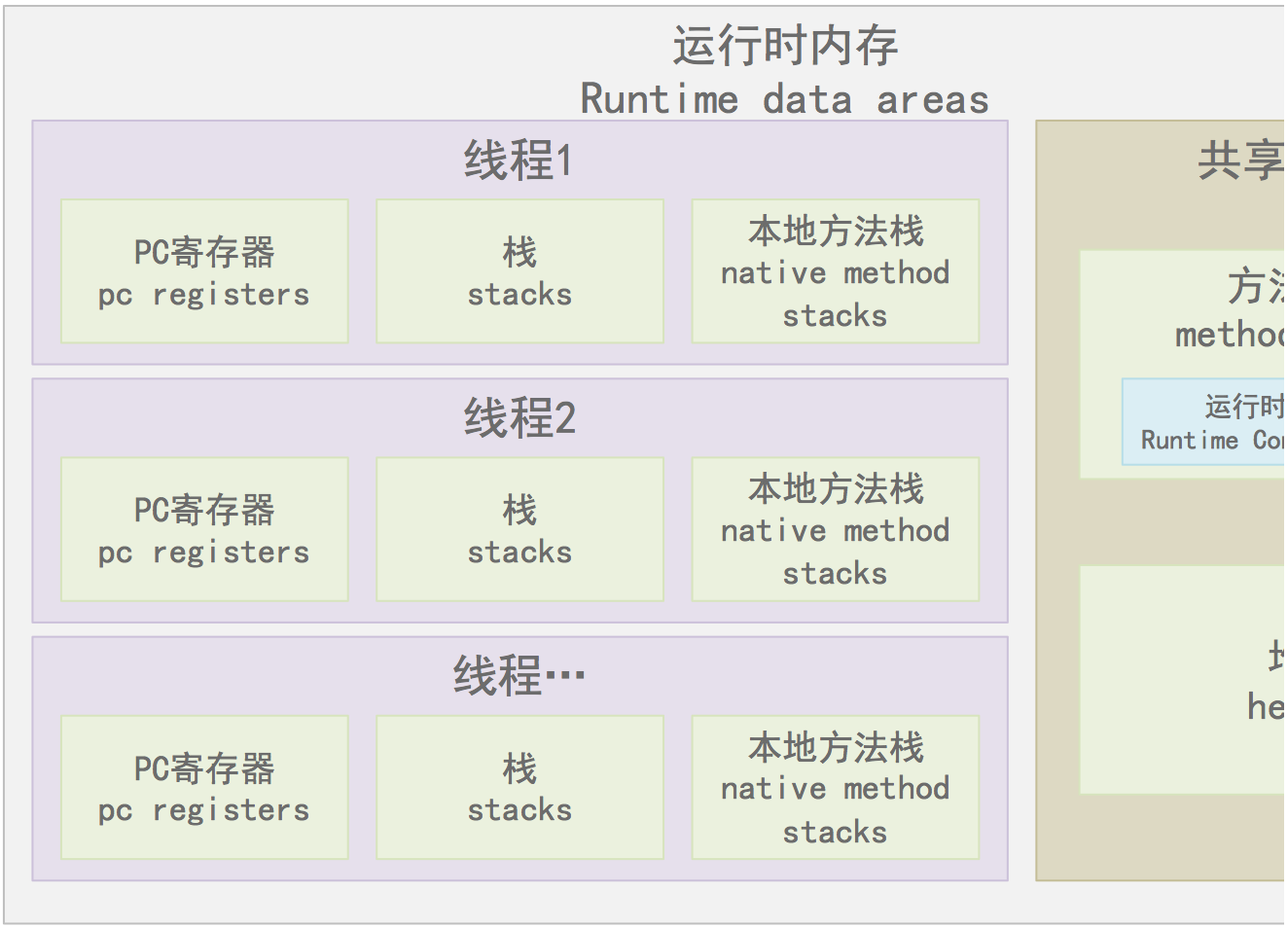
了解java内存模型看这里：java内存模型是什么样的
了解jvm实例结构看这里：jvm实例的结构是什么样的
创建对象或者变量时， JVM会自动分配内存（当然这个分配是遵循严格规则的）。当JVM发现某些对象不再需要的时候，就会对该对象占用的内存进行重分配（释放）操作，而且使得分配出来的内存能够提供给所需要的对象。

在这个方面，其他一些编程语言里面，内存管理是程序员的职责，程序员是需要手动管理内存。这一点C++的程序员很清楚，最终大部分开发时间都花在了调试这种内存管理程序以及修复相关错误上。

了解显示内存管理的弊端看这里：显示内存管理有什么弊端

2、JVM内存组成结构

了解内存分配策略看这里：内存分配有哪些策略
JVM的内存组织需要在不同的运行时数据区进行操作。包括PC寄存器（计数器 pc registers）、方法区（method area）、本地方法栈（native method stacks）、栈（stacks）、堆（heap）。如图：



PC寄存器（计数器 pc registers）
每个新的线程启动后，它就会被JVM在内部分配自己的PC寄存器（计数器 pc registers），通过计数器来指示下一条指令执行。

方法区（method area）
方法区是用来储存类的装载信息的。当JVM加载一个类的时候，它定位到对应路径里去查找对应的Class文件，类加载器读取类文件（线性二进制数据），然后将该文件传递给JVM，JVM从二进制数据中提取信息并且将这些信息存储在方法区。
在JVM内部，所有的线程共享相同的方法区。需要注意的是：类中的静态变量（类变量）就是储存在方法区中的（了解静态变量看这里：局部变量、类变量、实例变量有什么区别）。

方法区中除了有类文件信息外，还包含一个运行时常量池（Runtime Constant Pool），用来存放基本类型包装类（包装类不管理浮点型，整形只会管理-128到127）和String（通过String.intern()方法可以强制将String放入常量池）。之所以称之为动态，是因为不单能在编译期产生常量，运行期间也可以，当然运行时常量池同样是所有线程共享。（Java虚拟机对Class文件的每一部分的格式都有严格的规定，每一个字节用于存储哪种数据都必须符合规范，这样才会被虚拟机装载和执行。但对于运行时常量池，Java虚拟机规范没有做任何细节的要求，不同的提供商实现的虚拟机可以按照自己的需要来实现这个内存区域。）

由于运行时常量池是方法区的一部分，所以会受到方法区内存的限制，当常量池无法再申请到内存时会抛出OutOfMemoryError: PermGen space异常（Java 8以后没有方法区，由本地元空间代替，溢出会抛出OutOfMemoryError: Metaspace异常）。

本地方法栈（native method stacks）
若某线程正在执行一个本地Java方法，该线程的本地方法内存栈中，保存了本地Java方法调用状态，其状态包括局部变量、被调用的参数、它的返回值、以及中间计算结果。
在这种情况下，使得这些本地方法和其他内存数据区的内容尽可能独立，而且这些本地方法执行的字节码，有可能根据操作系统环境的不同，使得其编译出来的本地字节码的结构也有一定的差异。

栈（stacks）

1. 对于线程内存栈分配：当一个新线程启动的时候，JVM会为Java线程创建每个线程的独立内存栈。内存栈是由栈帧构成，在JVM里面，栈帧的操作只有两种：出栈和入栈。正在被线程执行的方法称为当前线程方法，而该方法的栈帧就称为当前帧，而在该方法内定义的类称为当前类。
2. 对于方法：当一个线程调用某个Java方法时，Jvm创建并将一个新帧压入到内存栈中，这个帧成为当前栈帧，当该方法执行的时候，JVM使用内存栈来存储参数引用、局部引用变量、基本类型数值、中间计算结果以及其他相关数据。
   方法在执行过程有可能因为两种方式而结束：如果一个方法返回，属于正常结束；如果在这个过程抛出异常而结束，为异常结束。不论是正常结束还是异常结束，JVM都会弹出或者丢弃该栈帧，则上一帧的方法就成为了当前帧。
3. 对于线程：在JVM中，Java线程的栈数据是某个线程独有的，其他的线程不能修改或访问该线程的栈帧。当一个线程调用某个方法的时候，方法的局部变量是在线程独有的栈帧存储，只有当前线程可以访问该局部变量， 所以不用担心多线程同步访问Java的局部变量。
4. 对于容量：编程过程，允许指定Java栈的初始大小以及最大、最小容量。

堆（heap）

1. 对于对象内存堆分配：当一个Java程序创建一个对象或者一个数组时，JVM实例会针对该对象和数组分配一个新的内存堆空间。在JVM实例内部，只存在一个内存堆的实例，所有的依赖该JVM的Java程序都共享该实例。
2. 对于进程内存堆分配：在Java程序启动的时候，会得到JVM分配的属于自己的堆空间，而且针对每一个Java应用程序，这些运行Java程序的堆空间都是相互独立的。
3. 对于内存堆共享：上述两种分配并不冲突，JVM在初始化运行的时候整体堆空间只有一个，这个是Java语言平台直接从操作系统上能够拿到的整体堆空间（不会超过物理内存最大值），所以的依赖该JVM的程序都可以得到这些内存空间。但是针对每一个独立的Java程序而言，这些堆空间是相互独立的，每一个Java应用程序在运行最初都是依靠JVM来进行堆空间的分配的。
4. 对于进程：两个相同的Java程序，在运行时处于不同的进程中（一般为java.exe），它们各自分配的堆空间都是独立的，不能相互访问。只是两个Java进程初始化拿到的堆空间都是来自JVM的分配（从最初的内存堆实例里面分配出来的）。
5. 对于线程：在同一个Java进程中，不同的线程是可以共享每一个Java程序拿到的内存堆空间的。这也是为什么在开发多线程程序的时候，针对同一个Java程序必须考虑线程安全问题，因为在一个Java进程里，所有的线程是可以共享这个Java进程堆空间中的数据的。了解进程和线程看这里：线程和进程有什么区别
6. 堆内存释放：JVM拥有针对新的对象分配内存的指令，但是却不包含释放该内存空间的指令。当然开发过程可以在Java源代码中显示释放内存或者说在JVM字节码中进行显示的内存释放，但是JVM仅仅只是检测堆空间中是否有引用不可达（不可以引用）的对象，然后将接下来的操作交给垃圾回收器（GC）来处理。

了解java堆和栈的区别看这里：java堆和栈有什么区别
了解堆的结构看这里：java堆内存是什么样的
了解java垃圾处理看这里：java垃圾回收机制是什么

]

[

Java千百问_07JVM架构（005）_显示内存管理有什么弊端

,

点击进入_更多_Java千百问

1、显示内存管理有什么弊端

手动内存管理一般被称为显示内存管理，显示内存管理经常发生两种情况：

1. 引用悬挂
   当一个被某个引用变量正在使用的内存空间，在重新分配过程中被释放掉了，释放后，该引用变量就处于悬挂状态（所引用的对象已经不存在了）。
   如果这个被悬挂引用变量，试图操作原来对象的时候，由于该对象本身的内存空间已经被手动释放掉了（已经不存在了），所以这个执行结果是不可预知的。

2. 内存泄漏
   当某些引用变量不再引用该内存对象的时候，而该对象原本占用的内存并没有被释放，这种情况就是内存泄漏。
   例如，对某个链表进行了内存分配，因为手动分配内存不当，仅仅让引用变量指向了某个元素所处的内存空间，就使得这些元素所处的内存空间对程序来说处于不可达状态，而且这些对象所占有的内存也不能够被再使用，这个时候就发生了内存泄漏。
   而这种情况一旦在程序中发生，就会一直消耗系统的可用内存直到可用的内存耗尽。对计算机而言，内存泄漏是非常严重的，会使得本来正常运行的程序直接因为内存不足而中断。与Java中的Exception并不是一个级别的错误。

对于显示内存管理来说，手动管理内存成本太高，风险很大。所以大多数纯面向对象语言而言，都提供了语言本身具有的内存特性：自动化内存管理。比如Java，提供了一个垃圾回收器（Garbage Collector，GC），自动内存管理提供了更加可靠的代码使得内存能够在程序里面进行合理的分配。

2、java的自动内存管理有什么好处

java的自动内存管理（隐式内存管理）可以解决显示内存管理的问题：引用悬挂和内存泄漏。

1. 对于引用悬挂
   JVM隐藏了对内存的管理，所以不会对正在使用的内存空间进行释放，也就不会出现引用悬挂。

2. 对于内存泄漏
   一旦有对象没有被任何引用变量引用时，也就是说这些对象在JVM的内存池里面成为了不可引用对象，垃圾回收器会直接回收掉这些对象的内存（当然需要满足一些规则，具体看这里：java垃圾回收机制是什么）。

]

[

Java千百问_07JVM架构（006）_java堆和栈有什么区别

,

点击进入_更多_Java千百问

1、java堆和栈有什么区别

了解jvm内存管理看这里：jvm是如何管理内存的
在《jvm是如何管理内存的》这篇文章中，已经对PC寄存器（计数器 pc registers）、方法区（method area）、本地方法栈（native method stacks）、栈（stacks）、堆（heap）内存区域做了介绍，其中栈（stacks）、堆（heap）是java内存管理中非常重要的两个部分，具体区别如下：

	内存栈	内存堆
设计模式	代表处理逻辑	代表数据
分配方式	为每个线程独立分配	进程统一分配，进程内所有线程共享
生命周期	线程启动时分配，线程销毁时回收	进程启动时分配，进程结束时回收
访问权限	不能跨线程访问/修改	所有进程内的线程均可访问
应用分配	执行方法时，存储参数引用、局部引用变量、基本类型数值、中间计算结果等	实例化对象或数组时，划分一部分堆空间用来储存对象或数组
回收方式	即用即回收，方法执行后立即回收方法内的资源，基本数据类型的值	垃圾回收机制回收，对象或数组没有被引用时会在垃圾回收机制控制下分批回收
容量	由Xss来调节，一般不会太大	由-Xmx和-Xms来调节
存取速度	存取速度比堆要快，仅次于寄存器	由于要在运行时动态分配内存，存取速度较慢
进出机制	出栈和入栈，先进后出，不能跳栈	可以指定某个内存空间

2、基本数据类型储存在栈中吗

基本数据类型由于长度固定，且需要空间比较少，所以直接存储在栈中（String是一个特殊的类型，它的值存储在堆中，但是通过池达到和栈类似的存取速度，具体看这里：String在内存中如何存放）。而对象比较大，所以栈中只存储一个4btye的引用地址（逻辑地址）。

实例:

```
int a = 3 ;
int b = 3 ;
```

分析如下：

1. 编译器先处理int a = 3；首先它会在栈中创建一个变量为a的引用，然后查找栈中是否有3这个值，如果没找到，就将3存放进来，然后将a指向3。
2. 接着处理int b = 3；在创建完b的引用变量后，因为在栈中已经有3这个值，便将b直接指向3。这样，就出现了a与b同时均指向3的情况。
3. 这时，如果再令a=4；那么编译器会重新搜索栈中是否有4值，如果没有，则将4存放进来，并令a指向4；如果已经有了，则直接将a指向它。因此a值的改变不会影响到b的值。

]

[

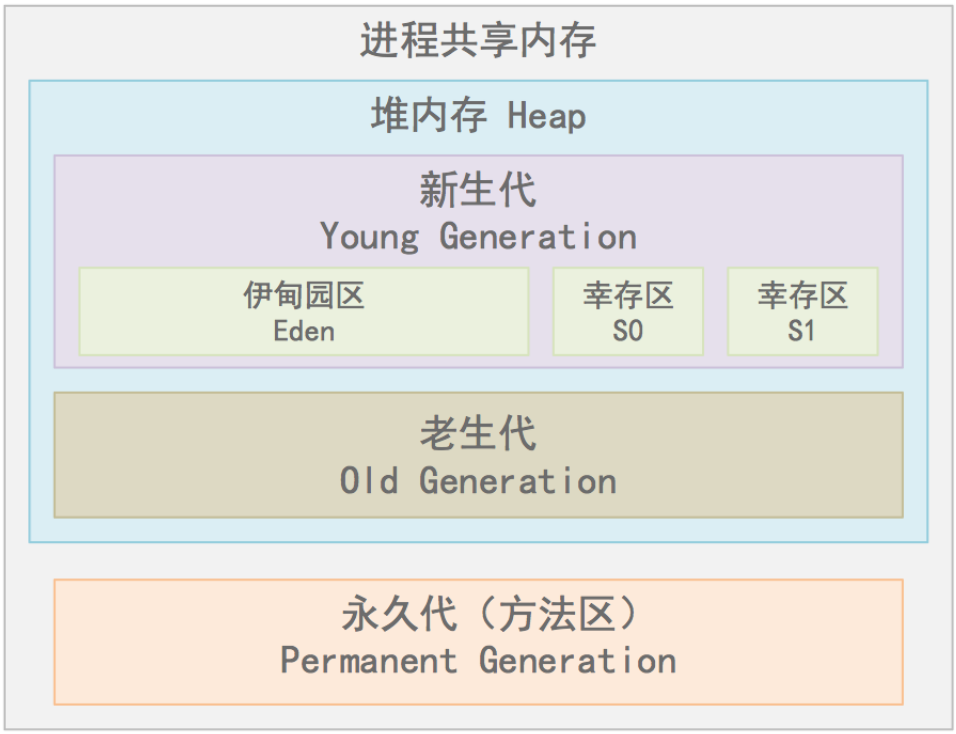
Java千百问_07JVM架构（007）_java堆内存是什么样的

,

点击进入_更多_Java千百问

1.堆内存是什么样的

了解jvm实例模型看这里：jvm实例的结构是什么样的
了解java内存框架看这里：jvm是如何管理内存的
了解堆栈的区别看这里：java堆和栈有什么区别
每一个jvm实例都会被分配一个被所有线程共享的堆内存空间，用来存放对象和数组，作为jvm的数据集中管理区，存取效率、空间释放就成为了重中之重，jvm通过多区架构来完成这两个目标的达成。其结构如下：



2、什么是新生代

新生代（Young Generation）主要是用来存放新生的对象。

新生代又被进一步划分为Eden区（伊甸园区）和Survivor区（幸存区，包含空间相等的S0、S1区，或者说From、To区，没有先后顺序，是Copying算法的需要）。

大多数情况下，java中新建的对象都是在新生代上分配的，通过Copying算法来进行分配内存和垃圾回收。

了解堆的Copying算法看这里：什么是新生代的Copying算法

有两种情况下java新创建的对象会直接到旧生代：

1. 占用空间大的对象/数组，且对象中无外部引用的对象。
2. 通过启动参数上面进行设置-XX：PretenureSizeThreshold=1024(单位是字节），如果对象超过此大小，就直接分配到旧生代。此外，并行垃圾回收器可以在运行期决定那些对象可以直接创建在旧生代。

了解垃圾回收机制看这里：java垃圾回收机制是什么
通过-Xmn设置新生代的大小，通过-XX：SurvivorRatio设置Eden区和Survivor区的比值，有些垃圾回收器会对S0或者S1进行动态的调整。

新生对象根据Copying算法在Eden区/S0区或者Eden区/S1区中分配，Eden区的对象量达到阈值后，发生一次新生代GC。

3、什么是老生代

老生代（Old Generation）主要存放应用程序中生命周期长的内存对象。

在新生代中经过多次垃圾回收仍然存活的对象，会被存放到老生代中。老生代通过标记/整理算法来清理无用内存。

多次回收之后仍然存活的对象，大小是-Xms减去-Xmn。

老生代通过-XX:MaxTenuringThreshold设置最大年龄阈值，每个对象有“对象年龄计数器”，对象由新生代Eden区（伊甸园区）收集到Survivor区（幸存区）后，年龄+1。新生代垃圾清理（GC）后，年龄+1。依次，当年龄>=阈值后进入老生代。

对于年龄阈值有两中特殊情况：

1. 如果在Survivor区（幸存区）中所有相同年龄对象占用了空间的一半以上，大于等于上述年龄的对象直接进入老生代。
2. 占用空间大于-XX:PretenureSizeThreshold设定阈值的大对象（比如大的数组），会直接进入老生代。

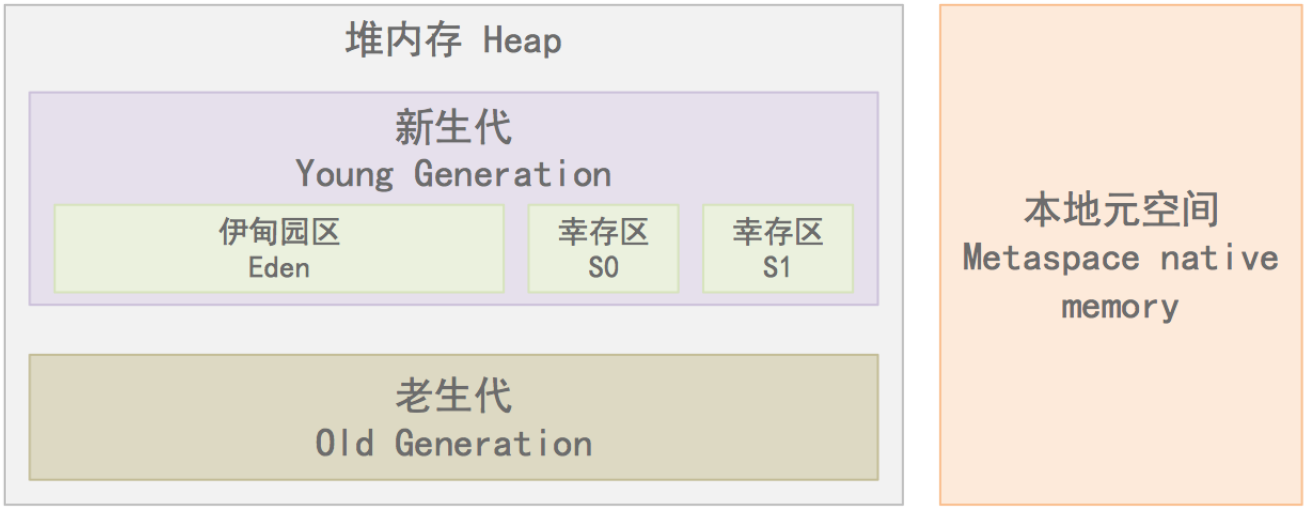
4、什么是永久代

永久代（Permanent Generation）即方法区，主要存放Class和Meta等永久保存的信息（如类、方法、字符串等）。

Class在被加载的时候被放入PermGen space区域。它和存放对象实例的堆内存不同，垃圾收集（GC）不会在主程序运行期对PermGen space进行清理，所以如果你的程序会加载了很多Class的话，就很可能出现PermGen space错误。

这里要说明的是，以上的堆内存机构是Java 8之前的结构，在新版本的Java中有如下变化：

- 在Java 7版本时把驻留字符串（intentd string）放到了老生代区。
- 在Java 8中，永久代在堆中被移除，取而代之的是本地元空间，这与Oracle JRockit和IBM JVM类似，JVM也开始使用本地化的内存，来存放类的元数据。Java 8的堆内存结构如下：



]

[

Java千百问_07JVM架构（008）_java垃圾回收机制是什么

,

点击进入_更多_Java千百问

1、如何判断垃圾对象

垃圾收集的第一步就是先需要算法来标记哪些是垃圾，然后再对垃圾进行处理。通常的编程语言都会用以下算法之一进行判断：

1. 引用计数（ReferenceCounting）算法
   这种方法比较简单直观，核心思路是，给每个对象添加一个被引用计数器，被引用时+1，引用失效-1，等于0时就表示该对象没有被引用，可以被回收。

FlashPlayer/Python使用该算法，简单高效。但是，Java/C#并不采用该算法，因为该算法没有解决对象相互引用的问题，即：当两个对象相互引用且不被其它对象引用时，各自的引用计数为1，虽不为0，但仍然是可被回收的垃圾对象。

1. 根搜索（GC Roots Tracing）算法

基本原理是：GCRoot对象作为起始点（根）。如果从根到某个对象是可达的，则该对象称为可达对象（存活对象，不可回收对象）。否则就是不可达对象，可以被回收。

2、新生代如何清理垃圾

垃圾清理算法

新生代的垃圾收集器通常会假设大部分的对象的存活时间都非常短，只有少数对象的存活时间比较长。根据这个假设，新生代清理垃圾的算法主要是复制算法（Copying）。通过复制算法，可以将没有被引用的对象清理掉，并且可以将经过若干次（可配置）清理仍然存活的对象放入老生代。

了解堆内存看这里：java堆内存是什么样的
了解堆的Copying算法看这里：什么是新生代的Copying算法

垃圾清理触发方式

新生代采用“空闲指针”的方式来控制GC触发，指针保持最后一个在新生代分配的对象位置，当有新的对象要分配内存时，用于检查空间是否足够，不够就触发GC。新生代的GC通常叫做young GC，有时候也叫minor GC。

在连续分配对象过程中，对象会按照复制算法逐渐从Eden区到Survivor区，最后到老生代。

常用配置

- 新生代的大小：-Xmn
- Eden区和Survivor区的比值：-XX：SurvivorRatio

3、老生代如何清理垃圾

垃圾清理算法

老生代与新生代不同，对象存活的时间比较长，比较稳定，因此采用标记/整理（也叫标记-紧凑，Mark-Compact）算法。

了解堆的 Mark算法看这里：什么是老生代的Mark算法

垃圾清理触发方式

老生代的GC，通常叫做full GC，也叫major GC。老生代有多情况会触发GC，不过一般来说发生频率不高：

1. 旧生代空间不足
   调优时尽量让对象在新生代GC时被回收、让对象在新生代多存活一段时间和不要创建过大的对象及数组避免直接在老生代创建对象。

2. Pemanet Generation空间不足
   增大Perm Gen空间，避免太多静态对象。

3. GC后晋升到老生代的平均大小大于老生代剩余空间
   控制好新生代和旧生代的比例。

4. 手动调用System.gc()
   垃圾回收不要手动触发，尽量依靠JVM自身的机制。

常用配置

- 堆的初始空间：-Xms，可以推算出老生代的大小为-Xms减去-Xmn
- 堆的最大空间：-Xmx
- 最大年龄阈值：-XX:MaxTenuringThreshold，即新生代转入老生代的存活次数
- 老生代和新生代的比值：-XX:NewRatio，例如该值为3，则表示新生代与老生代比值为1:3

4、垃圾回收方式有哪些

以上是java垃圾回收机制的基础，JVM为我们提供了若干可供选择的回收方式，即我们俗称的垃圾回收器。主要有4种：

1. 串行垃圾回收器（Serial Garbage Collector）
2. 并行垃圾回收器（Parallel Garbage Collector）
3. 并发标记扫描垃圾回收器（CMS Garbage Collector）
4. G1垃圾回收器（G1 Garbage Collector）

这里我们将 详细介绍说明各类垃圾回收器：java垃圾回收都有哪些方式

]

[

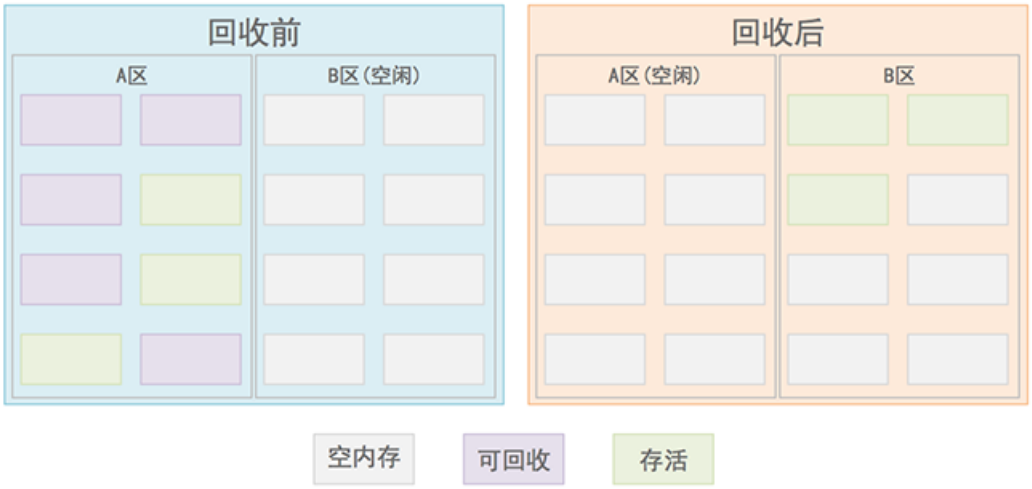
Java千百问_07JVM架构（009）_什么是新生代的复制算法

,

点击进入_更多_Java千百问

1、什么是新生代的复制算法

了解堆内存看这里：java堆内存是什么样的
了解java垃圾回收看这里：java垃圾回收机制是什么
所谓复制算法（Copying），即将内存平均分成A区、B区两块，进行复制+清除垃圾的操作，算法图解如下：



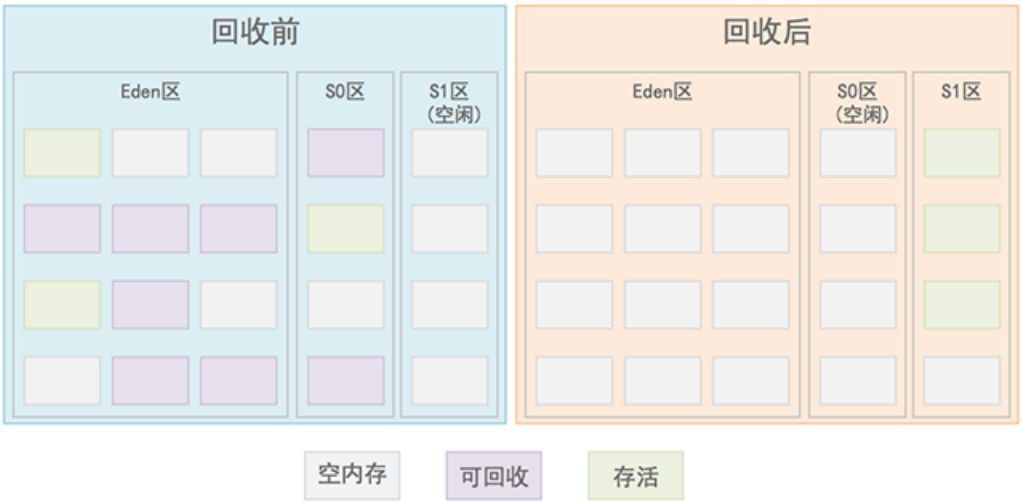
算法过程：

1. 新生对象被分配到A块中未使用的内存当中。当A块的内存用完了， 把A块的存活对象复制到B块。
2. 清理A块所有对象。
3. 新生对象被分配到B块中未使用的内存当中。当B块的内存用完了， 把B块的存活对象复制到A块。
4. 清理B块所有对象。
5. 循环1。

这种算法简单高效，但是内存代价极高，有效内存只为总内存的一半，会浪费掉50%的空间。所以这种算法只是纸面算法，不具备可用性，一般来说都会使用优化的复制算法。

2、什么是优化的复制算法

所谓优化的复制算法，即在复制算法的基础上，使用三个分区（Eden/S0/S1）进行处理，算法图解如下：



Eden/S0/S1默认空间比例Eden:S0:S1为8:1:1，有效内存（即可分配新生对象的内存）是总内存的90%。

算法过程：

1. Eden+S0可分配新生对象；
2. 对Eden+S0进行垃圾收集，存活对象复制到S1。清理Eden+S0。一次新生代GC结束。
3. Eden+S1可分配新生对象；
4. 对Eden+S1进行垃圾收集，存活对象复制到S0。清理Eden+S1。二次新生代GC结束。
5. 循环1。

我们可以看出，如果Eden/S0/S1三个空间的比例为8:1:1，则可能会出现Eden+S0中存活对象超过了总空间的10%（S1、S0的空间都是总空间的10%），在这种情况下，新生代GC会将存活周期长的对象直接放入老生代，而无需达到我们设置的阈值（转入老生代的存活次数，-XX:MaxTenuringThreshold）。

当然，这种情况在正常情况下不会出现（除非特殊场景，或者程序设计问题）。IBM的专门研究表明，新生代中的对象98%是朝生夕死的，所以8:1:1的比例是十分合理的。（每次新生代中可用内存空间为整个新生代容量的90%（80%+10%），只有10%的内存是会被浪费的）。

]

[

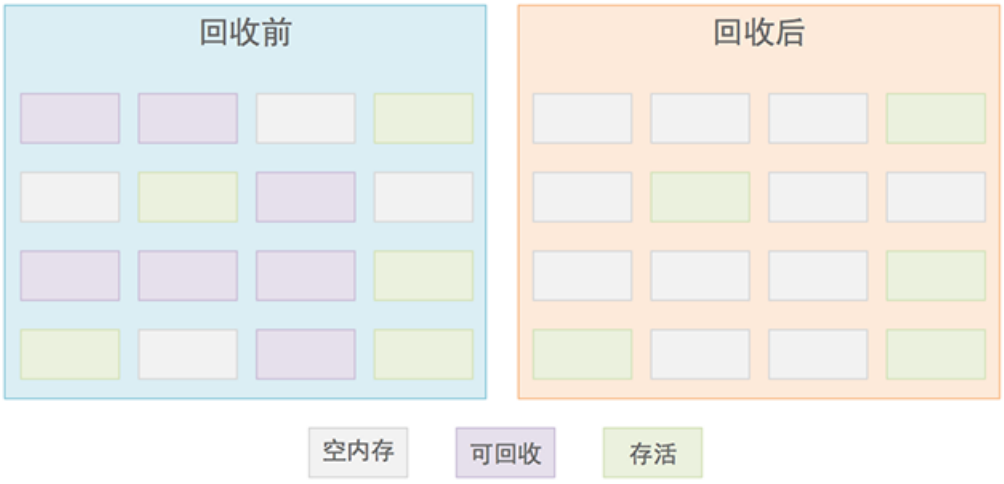
Java千百问_07JVM架构（010）_什么是老生代的标记算法

,

点击进入_更多_Java千百问

1、什么是老生代的标记算法

了解堆内存看这里：java堆内存是什么样的
了解java垃圾回收看这里：java垃圾回收机制是什么
所谓标记算法（Mark），分为多种，最简单直观的即标记-清除算法（Mark-Sweep）。即将认定为可回收的内存做一个标记，然后统一将被标记的清理，算法图解如下：

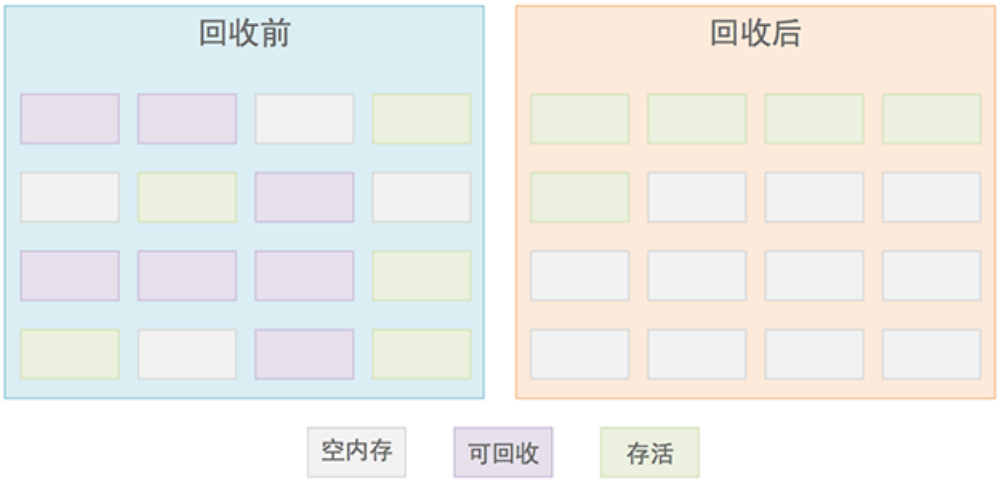


算法过程：
1. 先判定对象是否可回收，对其标记。
2. 统一回收（简单地删除对垃圾对象的内存引用）。

标记-清除算法十分简单直观，且容易实现和理解。但是有一个很严重的问题，内存空间碎片化，这显然是不能接收的，所以Java对老生代的垃圾处理采用的是标记-紧凑算法。

2、什么是老生代的标记-紧凑算法

所谓标记-紧凑算法（Mark-Compact），即在标记-清除算法的基础上，增加了碎片整理这一步，算法图解如下：



算法过程：

1. 标记：标记可回收对象（垃圾对象）和存活对象。
2. 紧凑（也称“整理”）：将所有存活对象向内存开始部位移动，称为内存紧凑（相当于碎片整理）。
3. 清理剩余内存空间。

可以看出，内存的碎片整理虽然会损失一定的效率，但是大大减少了内存的碎片化程度，更有利于内存的使用和分配。

]

[

Java千百问_07JVM架构（011）_java垃圾回收都有哪些方式

,

点击进入_更多_Java千百问

1、java垃圾回收都有哪些方式

所谓垃圾回收方式，是指JVM提供的几种不同的垃圾回收器，不同的垃圾回收器进行垃圾回收时采用不同的方式。当然，总体原则遵循java垃圾回收机制。

了解java内存模型看这里：java内存模型是什么样的
了解堆内存看这里：java堆内存是什么样的
了解java垃圾回收机制看这里：java垃圾回收机制是什么

每种方式都有自己的优势与劣势。我们编程的时候可以通过向JVM传递参数来选择垃圾回收器。不同的垃圾回收期有大的不同，可以为我们提供完全不同的应用程序性能，所以理解每种垃圾回收器，并且根据不同的应用选择进行正确的选择是非常重要的。

垃圾回收器主要有4种：串行垃圾回收器（Serial Garbage Collector）、并行垃圾回收器（Parallel Garbage Collector）、并发标记扫描垃圾回收器（CMS Garbage Collector）、G1垃圾回收器（G1 Garbage Collector），具体如下：

串行垃圾回收器

串行垃圾回收器通过持有应用程序所有的线程进行工作。它为单线程环境设计，只使用一个单独的线程进行垃圾回收，通过冻结所有应用程序线程进行工作，所以可能不适合服务器环境。它最适合的是简单的命令行程序。
通过JVM参数-XX:+UseSerialGC可以使用串行垃圾回收器。

并行垃圾回收器

并行垃圾回收器是JVM的默认垃圾回收器。与串行垃圾回收器不同，它使用多线程进行垃圾回收。相似的是，当执行垃圾回收的时候它也会冻结所有的应用程序线程。

并发标记扫描垃圾回收器

并发标记垃圾回收使用多线程扫描堆内存，标记需要清理的实例并且清理被标记过的实例。并发标记垃圾回收器只会在下面两种情况持有应用程序所有线程：

1. 当标记的引用对象在老生代；
2. 在进行垃圾回收的时候，堆内存的数据被并发的改变。

相比并行垃圾回收器，并发标记扫描垃圾回收器使用更多的CPU来确保程序的吞吐量。如果我们能够为了更好的程序性能分配更多的CPU，那么相比并发垃圾回收器，并发标记扫描垃圾回收器是更好的选择。
通过JVM参数XX:+USeParNewGC打开并发标记扫描垃圾回收器。

G1垃圾回收器

G1垃圾回收器将堆内存分割成不同的区域，并且并发的对其进行进行垃圾回收。G1也可以在回收内存之后对剩余的堆内存空间进行压缩。
G1垃圾回收会优先选择第一块垃圾最多的区域，它适用于堆内存很大的情况。
通过JVM参数–XX:+UseG1GC使用G1垃圾回收器
在Java 8中，G1可以通过JVM参数-XX:+UseStringDeduplication删除重复的字符串，只保留一个char[]来优化堆内存。

2、如何选择不同的垃圾回收器

我们需要根据应用场景、硬件性能和吞吐量需求来决定使用哪一种垃圾回收器，通过JVM参数的配置来选择垃圾回收器：

-XX:+UseSerialGC：串行垃圾回收器
-XX:+UseParallelGC：并行垃圾回收器

-XX:+UseConcMarkSweepGC：并发标记扫描垃圾回收器
-XX:ParallelCMSThreads：并发标记扫描垃圾回收器 =为使用的线程数量
-XX:+UseG1GC：G1垃圾回收器

其他内存常见配置：

-Xms：初始化堆内存大小
-Xmx：堆内存最大值
-Xmn：新生代大小
-XX:PermSize：初始化永久代大小
-XX:MaxPermSize：永久代最大容量

配置JVM GC参数的实例：

```
java -Xmx1024m -Xms512m -Xmn256m -XX:PermSize=64m -XX:MaxPermSize=128m -XX:+UseSerialGC -jar java-application.jar
```

上述配置的含义是：
堆内存512-1024M，新生代256M，永久代64-128M，采用串行垃圾回收器执行java-application.jar。

]

[

Java千百问_07JVM架构（012）_fullGC、minorGC、magorGC有什么区别

,

点击进入_更多_Java千百问

1、fullGC、minorGC、majorGC有什么区别

fullGC、minorGC、majorGC还有youngGC是Java垃圾处理机制（GC）的名词，区分这几个概念非常简单：

1. 老生代进行一次垃圾清理，被称为fullGC或者majorGC。
2. 新生代进行一次垃圾清理，被称为youngGC或者minorGC。

了解java垃圾回收机制看这里：java垃圾回收机制是什么

解释完毕，不过要提一下的是，我们在JVM优化过程中的一个原则就是：

降低youngGC的频率、减少fullGC的次数。

了解JVM优化看这里：[java内存如何优化][3]
[3]:

]

[

Java千百问_07JVM架构（013）_java什么情况会内存溢出

,

点击进入_更多_Java千百问

1、Java堆什么情况会溢出

所有对象的实例都在Java堆上分配内存，堆大小由-Xmx和-Xms来调节，如果程序使用的内存超过了堆最大内存（-Xmx），则会溢出Java heap space。

了解java内存模型看这里：java内存模型是什么样的
了解堆内存看这里：java堆内存是什么样的
了解java垃圾回收机制看这里：java垃圾回收机制是什么

实例：

```
public class HeapOOM {

    static class OOMObject{}

    /**
     * @param args
     */
    public static void main(String[] args) {
        List<OOMObject> list = new ArrayList<OOMObject>();

        while(true){
            list.add(new OOMObject());
        }
    }

}
```

加上JVM参数运行：

```
-verbose:gc -Xms10M -Xmx10M -XX:+PrintGCDetails -XX:SurvivorRatio=8 -XX:+HeapDumpOnOutOfMemoryError
```

其中-verbose:gc是为了打印GC过程的详细情况，上述程序就能很快报出内存溢出（OOM）：
Exception in thread “main” java.lang.OutOfMemoryError: Java heap space

并且能自动生成Dump文件，Dump记录了进程内存的完整信息。

了解Dump文件看这里：什么是Dump文件

2、Java方法区什么情况会溢出

方法区（永久代）是存放虚拟机加载类的相关信息（如类、静态变量和常量），大小由-XX:PermSize和-XX:MaxPermSize来调节，类太多有可能使永久代溢出PermGen space。Java 8 以后移除了方法区，取而代之的是本地元空间Metaspace，大小由-XX:MetaspaceSize和-XX:MaxMetaspaceSize调节，移除的错误也变为java.lang.OutOfMemoryError: Metaspace。

实例：

```
public class MethodAreaOOM {

    static class OOMOjbect{}
```

```
    /**
     * @param args
     */
    public static void main(String[] args) {
        // TODO Auto-generated method stub
        while(true){
            Enhancer eh = new Enhancer();
            eh.setSuperclass(OOMOjbect.class);
            eh.setUseCache(false);
            eh.setCallback(new MethodInterceptor(){

                @Override
                public Object intercept(Object arg0, Method arg1,
                        Object[] arg2, MethodProxy arg3) throws Throwable {
                    // TODO Auto-generated method stub
                    return arg3.invokeSuper(arg0, arg2);
                }

            });
            eh.create();
        }
    }

}
```

加上永久代的JVM参数后：

```
-XX:PermSize=10M -XX:MaxPermSize=10M
```

运行后会报如下异常：
Exception in thread "main" java.lang.OutOfMemoryError: PermGen space

静态变量或常量也会有可能撑爆方法区：

```
public class ConstantOOM {

    /**
     * @param args
     */
    public static void main(String[] args) {
        // TODO Auto-generated method stub
        List<String> list = new ArrayList<String>();
        int i=0;
        while(true){
            list.add(String.valueOf(i++).intern());
        }
    }

}
```

同样加上JVM参数：

```
-XX:PermSize=10M -XX:MaxPermSize=10M
```

运行后报如下异常：
Exception in thread "main" java.lang.OutOfMemoryError: PermGen space

3、Java栈和本地方法栈什么情况会溢出

栈是存放线程调用方法时存储局部变量、操作、方法出口等与方法执行相关的信息，栈大小由-Xss来调节，方法调用层次太多会使栈溢出StackOverflowError。

实例：

```
package com.cutesource;

public class StackOOM {
```

```
    /**
     * @param args
     */

    private int stackLength = 1;

    public void stackLeak(){
        stackLength++;
        stackLeak();
    }

    public static void main(String[] args) throws Throwable{
        // TODO Auto-generated method stub
        StackOOM oom = new StackOOM();
        try{
            oom.stackLeak();
        }catch(Throwable err){
            System.out.println("Stack length:" + oom.stackLength);
            throw err;
        }

    }

}
```

设置JVM参数：

```
-Xss128k
```

运行报出异常：
Exception in thread "main" java.lang.StackOverflowError

打印出Stack length:1007，这里可以看出，128k的栈容量能承载深度为1007的方法调用。当然这个错误很少见，一般只会出现无限循环的递归中，另外，线程太多也会占满栈区域（线程内分配了自己的栈，但是进程中所有线程可使用的栈总大小是一定的）：

```
package com.cutesource;

public class StackOOM {

    /**
     * @param args
     */

    private int stackLength = 1;

    private void dontStop(){
        while(true){
            try{Thread.sleep(1000);}catch(Exception err){}
        }
    }

    public void stackLeakByThread(){
        while(true){
            Thread t = new Thread(new Runnable(){

                @Override
                public void run() {
                    // TODO Auto-generated method stub
                    dontStop();
                }

            });
            t.start();
            stackLength++;
        }
    }
```

```
    public static void main(String[] args) throws Throwable{
        // TODO Auto-generated method stub
        StackOOM oom = new StackOOM();
        try{
            oom.stackLeakByThread();
        }catch(Throwable err){
            System.out.println("Stack length:" + oom.stackLength);
            throw err;
        }

    }

}
```

运行报出异常：
Exception in thread “main” java.lang.OutOfMemoryError:unable to create new native thread

在windows上运行这个例子要很小心，可能会出现系统假死的情况，需要重启机器。

]

[

Java千百问_07JVM架构（014）_什么是Dump文件

,

点击进入_更多_Java千百问

1、什么是Dump文件

Dump文件是进程的内存镜像。可以把程序的执行状态通过调试器保存到dump文件中。

Dump文件是用来给驱动程序编写人员调试驱动程序用的，这种文件必须用专用工具软件打开。

当我们的程序发布出去之后，在客户机上是无法跟踪代码的，所以Dump（扩展名是 .dmp）文件对于我们来说特别重要。我们可以通过.dmp文件把出现问题的情况再现，然后根据再现的状况（包括堆栈调用等情况），可以找到出现问题对应的行号。

2、如何生成Dump文件

生成Dump文件方法多样，最常用的即通过WinDBG软件。步骤如下：

1. 下载安装WinDBG。
2. 打开WinDBG，打开File->Attach to a process，然后再列表中显示需要监视的进程（.exe）。
3. 当程序崩溃之后执行DUMP命令产生.dmp文件，常用命令有（可以查询WinDBG命令手册了解命令含义）：

```
.dump /m C:/dumps/myapp.dmp
.dump /ma C:/dumps/myapp.dmp
.dump /mFhutwd C:/dumps/myapp.dmp
```

执行以上就产生了dmp文件。

3、如何分析Dump文件

通过WinDBG也可以分析Dump文件，步骤如下：

1. 打开WinDBG， 打开File->Sybolm File Path，指定发布exe的时候.pdb生成的路径。
2. 打开File->Open Crash Dump，打开.dmp文件。
3. 使用WinDBG的命令来分析Bug， 如（可以查询WinDBG命令手册了解命令含义）：!analyze -v
4. 可以使用快捷栏的功能，查看一些变量的基本信息

]

[

Java千百问_07JVM架构（015）_XmnXmsXmxXss有什么区别

,

点击进入_更多_Java千百问

1、XmnXmsXmxXss有什么区别

首先，Xmn、Xms、Xmx、Xss都是JVM对内存的配置参数，我们可以根据不同需要区修改这些参数，以达到运行程序的最好效果。

了解jvm内存管理看这里：jvm是如何管理内存的

Xms、Xmx

-Xms、-Xmx分配用来设置进程堆内存的最小大小和最大大小。

了解堆内存看这里：java堆内存是什么样的
了解java垃圾回收机制看这里：java垃圾回收机制是什么

Xmn

-Xmn用来设置堆内新生代的大小。通过这个值我们也可以得到老生代的大小：-Xmx减去-Xmn

了解堆新生代垃圾处理算法看这里：什么是新生代的复制算法

Xss

-Xss设置每个线程可使用的内存大小。

在相同物理内存下，减小这个值能生成更多的线程。当然操作系统对一个进程内的线程数还是有限制的，不能无限生成，经验值在3000~5000左右。

除了这些配置，JVM还有非常多的配置，常用的如下：

1. -XX:PermSize、-XX:MaxPermSize
   分配用来设置永久代的最小大小和最大大小。Java 8以后移除了方法区，取而代之的是本地元空间Metaspace，大小由-XX:MetaspaceSize和-XX:MaxMetaspaceSize调节。

2. -XX:MaxTenuringThreshold
   设置转入老生代的存活次数。如果是0，则直接跳过新生代进入老生代。

3. -XX:NewRatio
   设置老生代和新生代的比值，例如该值为3，则表示新生代与老生代比值为1:3。

]

[

[置顶] Java千百问_07JVM架构（016）_java内存如何优化

,

点击进入_更多_Java千百问

1、java内存如何优化

了解jvm内存管理看这里：jvm是如何管理内存的
了解堆内存看这里：java堆内存是什么样的
java内存的优化主要是通过合理的控制GC来实现，主要原则：
1. 不能只看操作系统级别Java进程所占用的内存，这个数值不能准确的反应堆内存的真实占用情况（因为GC过后这个值是不会变化的）。
2. 使用JDK提供的内存查看工具，比如JConsole和Java VisualVM。
3. 优化内存主要的目的是降低youngGC的频率、减少fullGC的次数，过多的youngGC和fullGC是会占用很多的系统资源（主要是CPU），影响系统的吞吐量。

这里，影响内存效率最关键的就是：fullGC，因为它会对整个堆内存进行整理，占用大量的资源。

2、FullGC何时发生

fullGC作为java内存管理的关键，它的触发时机我们一定要掌握，有4种情况会发生fullGC：

1. 旧生代空间不足
   调优时尽量让对象在新生代GC时被回收、让对象在新生代多存活一段时间和不要创建过大的对象及数组避免直接在旧生代创建对象。

2. Pemanet Generation空间不足
   增大Perm Gen空间，避免太多静态对象。了解内存溢出看这里：java什么情况会内存溢出

3. GC后晋升到老生代的平均大小大于老生代剩余空间
   控制好新生代和旧生代的比例。

4. System.gc()被显示调用
   垃圾回收不要手动触发，尽量依靠JVM自身的机制。

3、GC有哪些优化手段

GC调优手段主要是通过控制堆内存的各个部分比例，以及根据不同场景采用不同的GC策略来实现。下面分别来看：

通过控制堆内存比例优化GC

1. 新生代设置过小
   一是新生代容易占满，导致新生代GC（youngGC）次数非常频繁，增大系统消耗；
   二是大对象直接进入老生代，占据了老生代剩余空间，诱发fullGC。
   了解youngGC和FullGC看这里：fullGC、minorGC、magorGC有什么区别

2. 新生代设置过大

一是新生代设置过大会导致旧生代过小（堆总量一定），从而诱发fullGC。
二是新生代GC耗时大幅度增加。
一般说来新生代占整个堆1/3比较合适。

3. Survivor设置过小
   新生代中根据优化复制算法，如果Survivor区（S0，S1）太小，会导致对象从eden区直接到达旧生代，降低了在新生代的存活时间。一般来说Eden:S0:S1为8:1:1是比较合理的。
   了解新生代复制算法看这里：什么是新生代的复制算法

4. Survivor设置过大
   若Survivor区过大，则eden区会过小，eden区容易达到阈值导致新生代GC频率增加。

5. 新生代对象存活时间过短
   通过-XX:MaxTenuringThreshold=n来控制新生代存活时间，如果存活时间过短，则对象会很快进入老生代，导致fullGC。所以要尽量让不常用对象在新生代被回收。

通过GC策略优化GC

新生代和旧生代都有很多种GC策略和组合搭配，选择这些策略对于我们这些开发人员是个难题，默认情况下JVM会自动调整新生代与老生代的比例、Eden区与Suvivor区的比例来达到性能目标，JVM提供两种较为简单的GC策略的设置方式：

1. 吞吐量优先
   JVM以吞吐量为指标，自行选择相应的GC策略及 控制堆内存的大小比例，来达到吞吐量指标。一次fullGC时间占总可用时间的比例，如果GC时间过长，会相应调整空间的大小（花费在GC上的时间比例不超过1 / (1 + n)）。
   通过-XX:GCTimeRatio=n来设置。

2. 暂停时间优先
   JVM以暂停时间为指标，自行选择相应的GC策略及控制堆内存的大小比例，尽量保证每次GC造成的应用停止时间都在指定的数值范围内完成。如果时间过长，会相应调整空间的大小（单位是毫秒）。
   通过-XX:MaxGCPauseMillis=n来设置。

]

[

Java千百问_07JVM架构（017）_jvm常见配置都有哪些

,

点击进入_更多_Java千百问

1、jvm常见配置都有哪些

了解jvm内存模型看这里：java内存模型是什么样的
了解jvm内存管理看这里：jvm是如何管理内存的
了解jvm垃圾回收机制看这里：java垃圾回收机制是什么

jvm配置非常多，按照不同类型划分，常常用来优化jvm内存（了解jvm内存优化看这里：java内存如何优化），常见配置如下：

了解垃圾回收器看这里：java垃圾回收都有哪些方式

1. 堆设置
   -Xms=n
   初始堆大小。
   -Xmx=n
   最大堆大小。
   -Xmn=n
   新生代大小，该配置优先于-XX:NewRatio，即如果配置了-Xmn，-XX:NewRatio会失效。
   -XX:NewRatio=n
   老生代和新生代的比值，例如该值为3，则表示新生代与老生代比值为1:3。
   -XX:SurvivorRatio=n
   新生代中 Eden区和Survivor区的比值。Survivor区分为等价的两个区S0，S1。例如-XX:SurvivorRatio=3，则表示Eden:S0:S1=3:1:1。
   -XX:PermSize=n
   设置永久代（方法区）大小，Java 8之后被移除。
   -XX:MaxPermSize=n
   设置永久代（方法区）最大大小，Java 8之后被移除。
   -XX:MetaspaceSize=n
   设置本地元空间Metaspace大小，Java 8以后移除了方法区，取而代之的是本地元空间Metaspace。
   -XX:MaxMetaspaceSize=n
   设置本地元空间Metaspace最大大小，Java 8以后移除了方法区，取而代之的是本地元空间Metaspace。
   -XX:MaxGCPauseMillis=n
   设置垃圾收集最大暂停时间。
   -XX:GCTimeRatio=n
   设置一次垃圾回收时间占程序运行时间的百分比，花费在GC上的时间比例不超过1 / (1 + n)。

2. 垃圾收集器设置
   -XX:+UseSerialGC
   设置串行收集器。
   -XX:+UseParallelGC
   设置并行收集器。
   -XX:+UseParallelOldGC
   设置并行年老代收集器。
   -XX:+UseConcMarkSweepGC
   设置并发收集器。
   -XX:+UseG1GC
   G1垃圾回收器。

3. 并行收集器设置
   -XX:ParallelGCThreads=n
   设置并发收集器年轻代收集方式为并行收集时的线程数。

4. 并发收集器设置
   -XX:+CMSIncrementalMode
   设置为增量模式。在增量模式下，并发收集器在并发阶段，不会独占整个周期，而会周期性的暂停，唤醒应用线程。收集器把并发阶段工作划分为片段，安排在次级(minor) 回收运行。适用于需要低延迟，CPU少的服务器。一般单CPU会开启增量模式。
   -XX:ParallelCMSThreads=n
   并发垃圾回收器标记扫描所使用的线程数量

5. 垃圾回收统计信息
   -XX:+PrintGC
   输出GC日志，简要日志。
   -XX:+PrintGCDetails
   打印GC日志，详细展示每个区的日志。
   -XX:+PrintGCTimeStamps
   打印GC日志，详细展示GC运行时间的日志。输出GC的时间戳以基准时间的形式展示。
   -XX:+PrintGCDateStamps
   打印GC日志，详细展示GC运行时间的日志。输出GC的时间戳以以日期的形式展示。
   -XX:+PrintHeapAtGC

打印在进行GC的前后打印出堆的信息。
-Xloggc:../logs/gc.log
日志文件的输出路径。

汇总如下：

]

[

Java千百问_07JVM架构（018）_如何监控jvm的运行情况

,

点击进入_更多_Java千百问

1、如何监控jvm的运行情况

了解jvm内存模型看这里：java内存模型是什么样的
了解jvm内存管理看这里：jvm是如何管理内存的
了解jvm垃圾回收机制看这里：java垃圾回收机制是什么
了解jvm内存优化看这里：java内存如何优化

我们通常使用Jdk工具来监控jvm的运行情况，当然目前有很多第三方产品是通过jdk提供的api来组织数据进行监控的。具体来说有如下监控软件：

- Jconsole
  jdk自带，功能简单，但是可以在系统有一定负荷的情况下使用。对垃圾回收算法有很详细的跟踪。
- JavaVisualVM
  JDK自带，功能强大，与JProfiler类似。推荐使用，通过jdk/bin/jvisualvm即可启动。
- JProfiler
  商业软件，需要付费。功能强大。

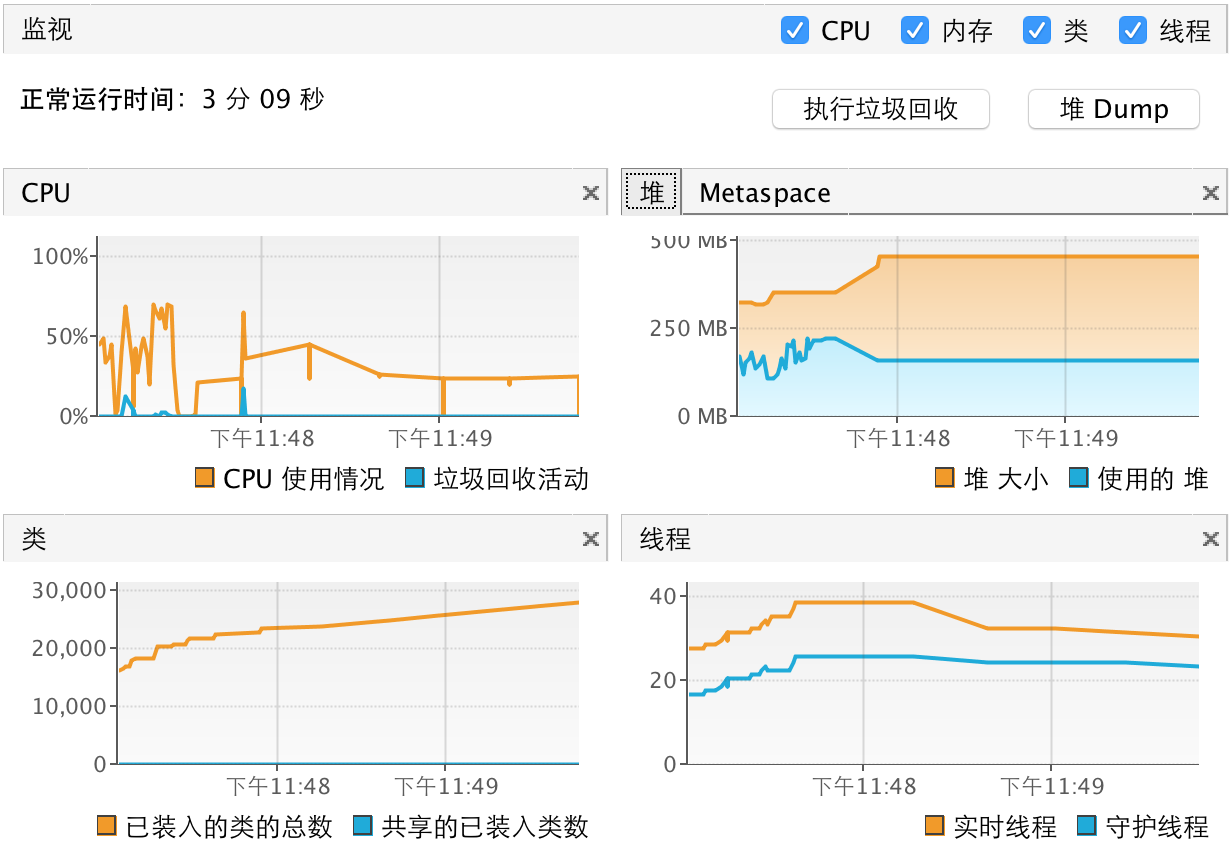
2、监控软件都能监控什么

上面这些调优工具都提供了强大的功能，但是总的来说一般都类似，能够监控CPU、内存、线程等信息。

例如JavaVisualVM具有以下几类功能：CPU、堆、类、线程关键信息，手动进行垃圾回收GC，线程详细信息，CPU、内存抽样信息，生成堆Dump、线程Dump

了解Dump文件看这里：什么是Dump文件

CPU、堆、类、线程关键信息
图形化CPMU、堆、类、线程关键信息供我们查看，并且可以通过"执行垃圾回收"可以手动进行垃圾回收GC。如图：



线程详细信息
应用中每个线程的详细信息，包括运行时、休眠、等待、驻留、监视等状态的线程。如图：

线程 线程可

实时线程: 31
守护线程: 24

线程 Dump

时间线

视图: 所有线程

名称	23:50:10	运行	总计
JMX server connection timeout 48		0 ms (0%)	185,538 r
RMI Scheduler(0)		0 ms (0%)	185,538 r
RMI TCP Connection(1)-192.168.1.		121,903 ms (65.7%)	185,538 r
RMI TCP Accept-0		185,538 ms (100%)	185,538 r
Keep-Alive-Timer		0 ms (0%)	5,363 r
Attach Listener		185,538 ms (100%)	185,538 r
Worker-7		13,044 ms (13.9%)	94,048 r
JavaScript indexing		0 ms (0%)	185,538 r
Worker-6		0 ms (0%)	94,048 r
Worker-5		5,013 ms (5.3%)	94,048 r
Worker-4		0 ms (0%)	138,308 r

运行 休眠 等待 驻留

CPU、内存抽样信息
通过收集一段时间的CPU、内存运行数据，展示内存和线程的详细信息。如图：

抽样器

抽样: CPU 内存 停止

状态: 正进行内存抽样

堆柱状图 每个线程分配

增量 快照 执行 GC 堆 Du

类: 8,514 实例: 3,592,045 字节: 175,703,688

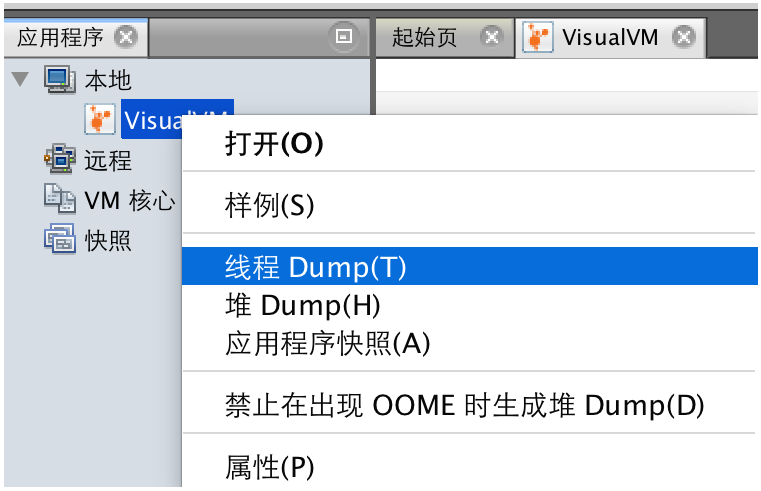
类名	字节 [%]	字节	实例数
char[]		60,163,080 (34.2%)	781,593 (21.7
java.lang.**String**		18,246,168 (10.3%)	760,257 (21.1
java.util.**HashMap$Node**		12,099,488 (6.8%)	378,109 (10.5
byte[]		11,744,720 (6.6%)	17,220 (0.4
java.lang.**Object[]**		6,678,784 (3.8%)	137,490 (3.8
java.util.**HashMap$Node[]**		5,136,824 (2.9%)	75,052 (2.0
java.util.**HashMap**		4,752,384 (2.7%)	99,008 (2.7
int[]		4,560,688 (2.5%)	34,748 (0.9
java.lang.**Class**		4,027,720 (2.2%)	33,874 (0.9

类名过滤器 (包含)

生成堆Dump、线程Dump

通过生成某个应用的堆、线程Dump来分析具体的Dump文件。
通过堆Dump可以监控：堆详细概要信息，堆内类对象分析，堆内对象实例详细信息，OQL堆信息自定义查询
通过线程Dump可以监控：线程状态变化，线程堆栈信息

如图：

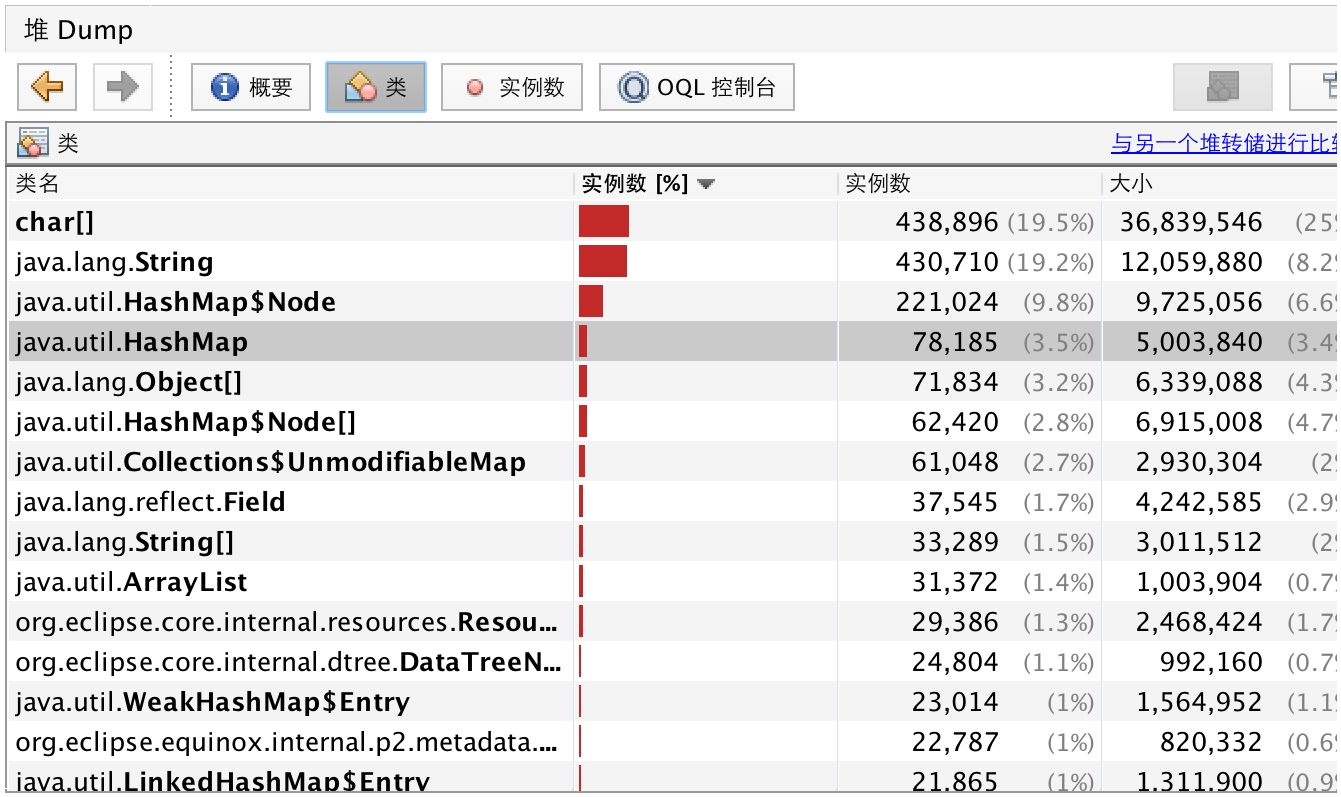


堆详细概要信息

展示堆Dump的概要信息，包括堆基本信息、环境、系统属性、堆转储上的线程。如图：

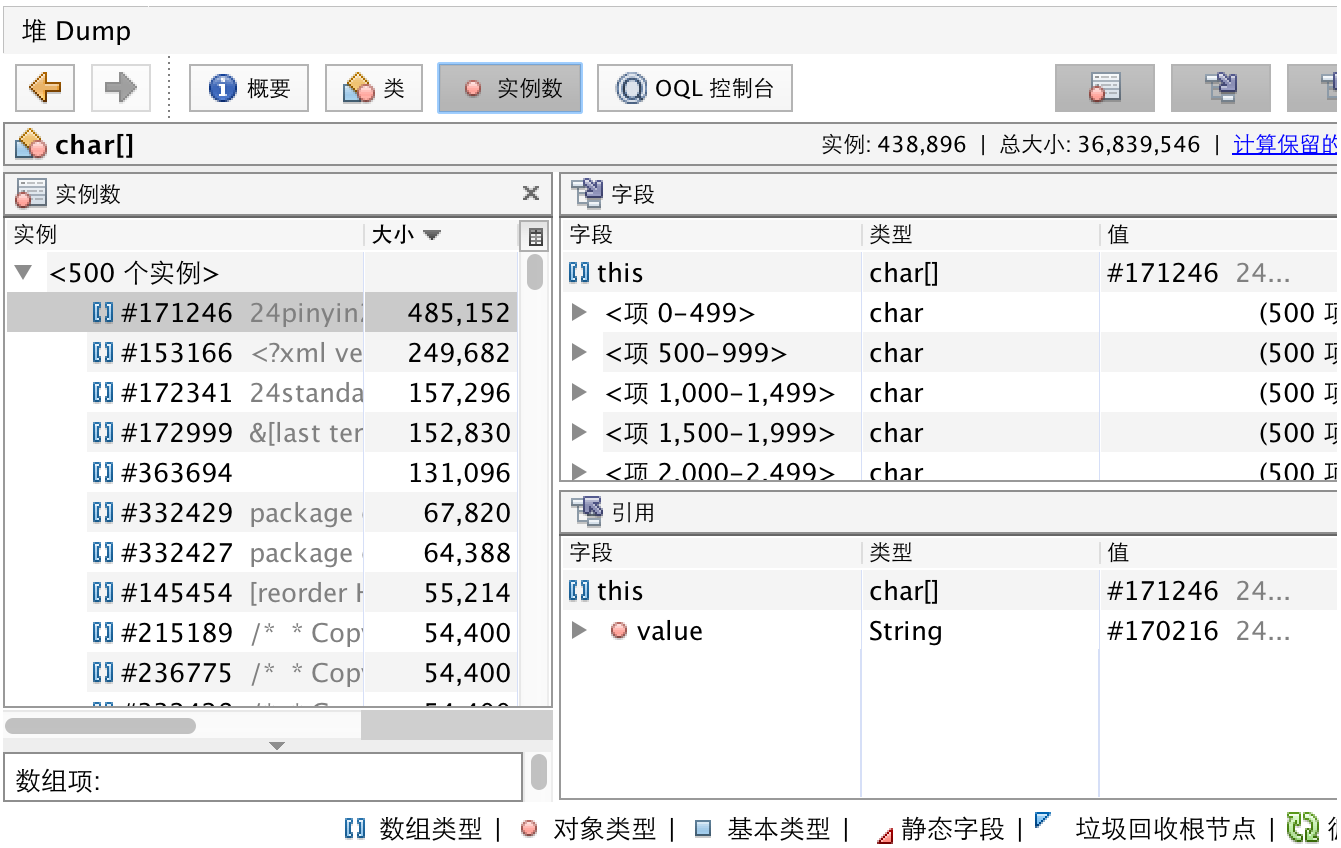
堆内类对象分析

展示堆内使用到的所有类，以及它们的实例数和占用大小。默认按照实例数倒序。如图：



堆内对象实例详细信息

通过类可以进入指定类的实例详细信息，包括每一个实例的详细信息。如图：

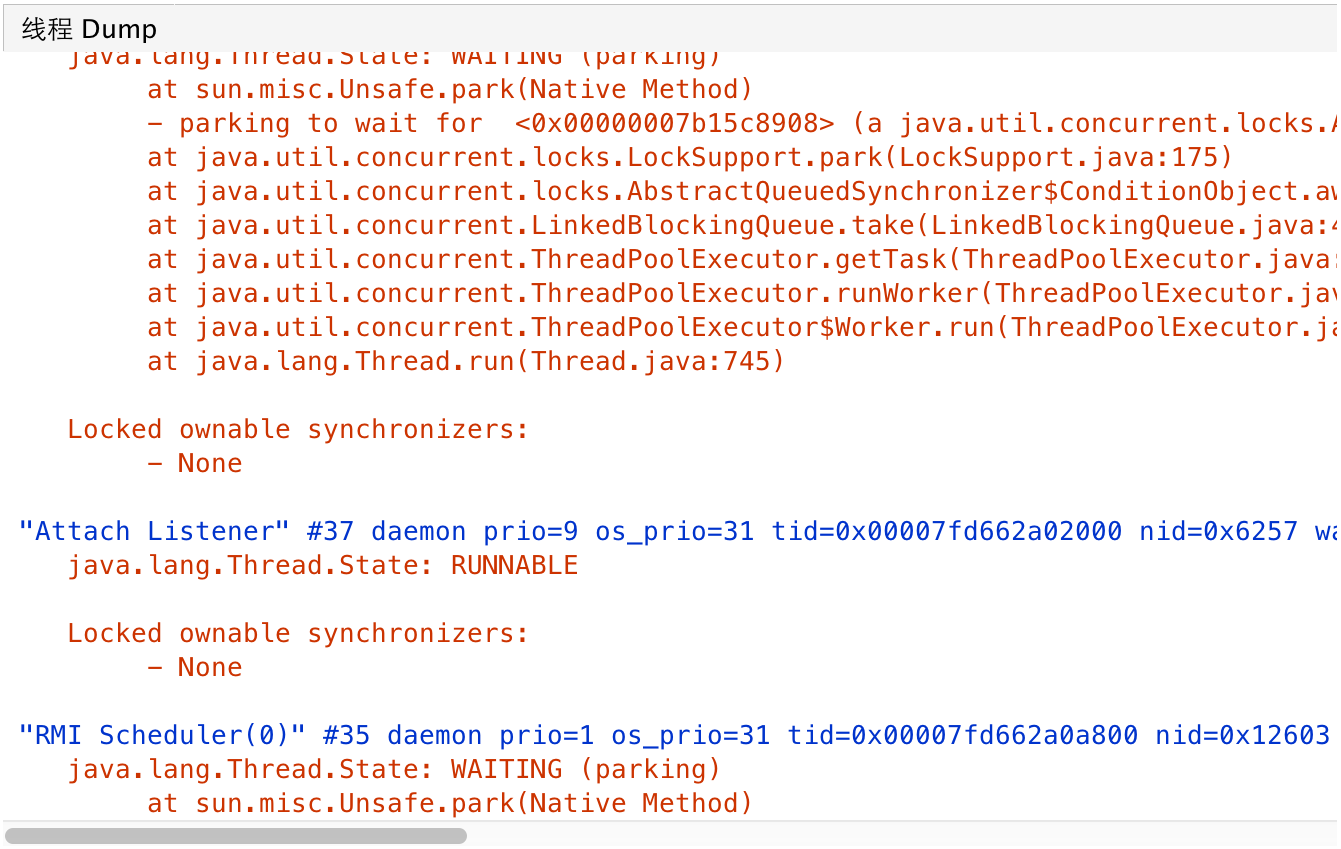


OQL堆信息自定义查询

通过OQL语言查询我们想得到的结果。如图：

线程Dump信息

通过打印线程堆栈，展示线程状态变化，以及运行信息。如图：



我们能够根据我们监控的到信息，能够让我们发现代码的问题，优化的办法。

]

[

Java千百问_07JVM架构（019）_运行时常量池是什么

,

点击进入_更多_Java千百问

1、运行时常量池是什么

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

了解java内存管理看这里：jvm是如何管理内存的
如图：

运行时常量是相对于常量来说的，它具备一个重要特征是：动态性。当然，值相同的动态常量与我们通常说的常量只是来源不同，但是都是储存在池内同一块内存区域。

Java语言并不要求常量一定只能在编译期产生，运行期间也可能产生新的常量，这些常量被放在运行时常量池中。这里所说的常量包括：基本类型包装类（包装类不管理浮点型，整形只会管理-128到127）和String（也可以通过String.intern()方法可以强制将String放入常量池）。

了解基本类型包装器看这里：什么是基本类型包装器
例子：

```
public class TestConst {

    public static String CONST_A = "the const b";// 编译时放入常量池

    public String const_b;

    public Integer const_b_i;

    public Integer const_b_ii;

    public Float const_b_f;

    public static void main(String[] args) {
        TestConst testConst = new TestConst();
        testConst.const_b = "the const b";// 运行时放入常量池
        testConst.const_b_i = 12;// 运行时放入常量池

        testConst.const_b_ii = 128;// 超过127，所以不会放入常量池
        testConst.const_b_f = 2.0f;// 浮点包装器不放入常量池
```

```
        String const_c = "the const b";// 运行时放入常量池
        Integer const_c_i = 12;// 运行时放入常量池

        Integer const_c_ii = 128;// 超过127，所以不会放入常量池
        Float const_c_f = 2.0f;// 浮点包装器不放入常量池

        System.out.println(CONST_A == const_c);
        System.out.println(CONST_A == testConst.const_b);
        System.out.println(testConst.const_b == const_c);
        System.out.println(testConst.const_b_i == const_c_i);
        System.out.println(testConst.const_b_ii == const_c_ii);
        System.out.println(testConst.const_b_f == const_c_f);
    }
}
```

运行结果：

true
true
true
true
false
false

Java虚拟机对Class文件的每一部分的格式都有严格的规定，每一个字节用于存储哪种数据都必须符合规范，这样才会被虚拟机装载和执行。但对于运行时常量池，Java虚拟机规范没有做任何细节的要求，不同的提供商实现的虚拟机可以按照自己的需要来实现这个内存区域。

由于运行时常量池是方法区的一部分，所以会受到方法区内存的限制，当常量池无法再申请到内存时会抛出OutOfMemoryError: PermGen space异常。

在Java 8以后移除了方法区，由本地元空间代替，运行时常量池也放在了本地元空间中，如果这个区内存溢出，则会抛出OutOfMemoryError: Metaspace错误。

]