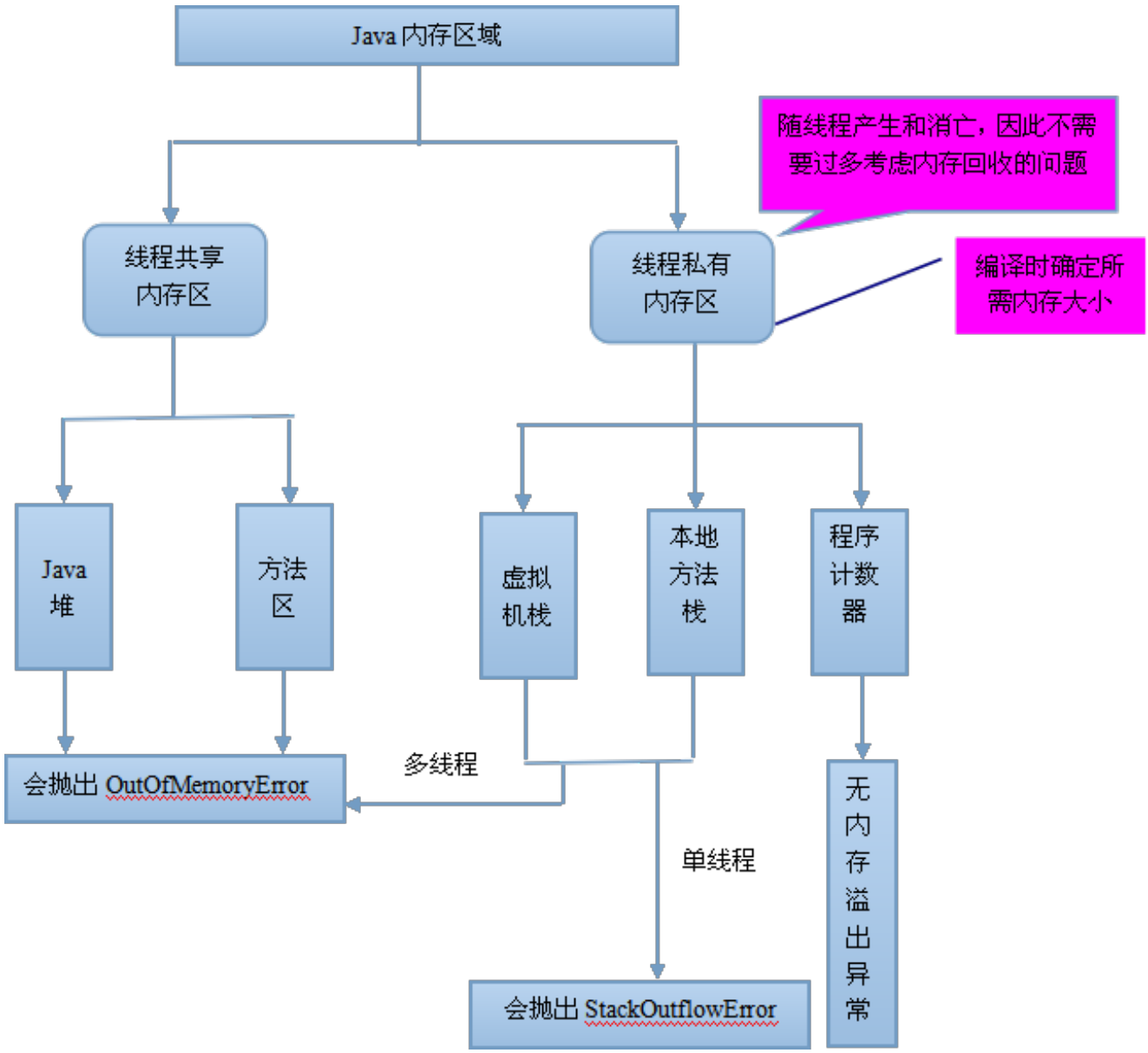
Java 内存區域與內存溢出

内存區域

Java 虛擬機在執行 Java 程序的過程中會把他所管理的內存劃分為若干個不同的數據區域。Java 虛擬機規範將 JVM 所管理的內存分為以下幾個運行時數據區：程序計數器、Java 虛擬機棧、本地方法棧、Java 堆、方法區。下面詳細闡述各數據區所存儲的數據類型。



程序計數器

一塊較小的內存空間，它是當前線程所執行的字節碼的行號指示器，字節碼解釋器工作時通過改變該計數器的值來選擇下一條需要執行的字節碼指令，分支、跳轉、循環等基礎功能都要依賴它來實現。每條線程都有一個獨立的的程序計數器，各線程間的計數器互不影響，因此該區域是線程私有的。

當線程在執行一個 Java 方法時，該計數器記錄的是正在執行的虛擬機字節碼指令的地址，當線程在執行的是 Native 方法（調用本地操作系統方法）時，該計數器的值為空。另外，該內存區域是唯一一個在 Java 虛擬機規範中麼有規

定任何 OOM（內存溢出：OutOfMemoryError）情況的區域。

Java 虛擬機棧

該區域也是線程私有的，它的生命週期也與線程相同。虛擬機棧描述的是 Java 方法執行的內存模型：每個方法被執行的時候都會同時創建一個棧幀，棧它是用於支持續虛擬機進行方法調用和方法執行的數據結構。對於執行引擎來講，活動線程中，只有棧頂的棧幀是有效的，稱為當前棧幀，這個棧幀所關聯的方法稱為當前方法，執行引擎所運行的所有字節碼指令都只針對當前棧幀進行操作。棧幀用於存儲局部變量表、操作數棧、動態鏈接、方法返回地址和一些額外的附加信息。在編譯程序代碼時，棧幀中需要多大的局部變量表、多深的操作數棧都已經完全確定了，並且寫入了方法表的 Code 屬性之中。因此，一個棧幀需要分配多少內存，不會受到程序運行期變量數據的影響，而僅僅取決於具體的虛擬機實現。

在 Java 虛擬機規範中，對這個區域規定了兩種異常情況：

- 如果線程請求的棧深度大於虛擬機所允許的深度，將拋出StackOverflowError異常。
- 如果虛擬機在動態擴展棧時無法申請到足夠的內存空間，則拋出OutOfMemoryError異常。

這兩種情況存在著一些互相重疊的地方：當棧空間無法繼續分配時，到底是內存太小，還是已使用的棧空間太大，其本質上只是對同一件事情的兩種描述而已。在單線程的操作中，無論是由於棧幀太大，還是虛擬機棧空間太小，當棧空間無法分配時，虛擬機拋出的都是 StackOverflowError 異常，而不會得到 OutOfMemoryError 異常。而在多線程環境下，則會拋出 OutOfMemoryError 異常。

下面詳細說明棧幀中所存放的各部分信息的作用和數據結構。

1、局部變量表

局部變量表是一組變量值存儲空間，用於存放方法參數和方法內部定義的局部變量，其中存放的數據的類型是編譯期可知的各種基本數據類型、對象引用（reference）和 returnAddress 類型（它指向了一條字節碼指令的地址）。局部變量表所需的內存空間在編譯期間完成分配，即在 Java 程序被編譯成 Class 文件時，就確定了所需分配的最大局部變量表的容量。當進入一個方法時，這個方法需要在棧中分配多大的局部變量空間是完全確定的，在方法運行期間不會改變局部變量表的大小。

局部變量表的容量以變量槽（Slot）為最小單位。在虛擬機規範中並沒有明確指明一個 Slot 應佔用的內存空間大小（允許其隨著處理器、操作系統或虛擬機的不同而發生變化），一個 Slot 可以存放一個32位以內的數據類型：boolean、byte、char、short、int、float、reference 和 returnAddresss。reference 是對象的引用類型，returnAddress 是為字節指令服務的，它執行了一條字節碼指令的地址。對於 64 位的數據類型（long和double），虛擬機會以高位在前的方式為其分配兩個連續的 Slot 空間。

虛擬機通過索引定位的方式使用局部變量表，索引值的範圍是從 0 開始到局部變量表最大的 Slot 數量，對於 32 位數據類型的變量，索引 n 代表第 n 個 Slot，對於 64 位的，索引 n 代表第 n 和第 n+1 兩個 Slot。

在方法執行時，虛擬機是使用局部變量表來完成參數值到參數變量列表的傳遞過程的，如果是實例方法（非static），則局部變量表中的第 0 位索引的 Slot 默認是用於傳遞方法所屬對象實例的引用，在方法中可以通過關鍵字「this」來訪問這個隱含的參數。其餘參數則按照參數表的順序來排列，佔用從1開始的局部變量 Slot，參數表分配完畢後，再根據方法體內部定義的變量順序和作用域分配其餘的 Slot。

局部變量表中的 Slot 是可重用的，方法體中定義的變量，作用域並不一定會覆蓋整個方法體，如果當前字節碼PC計數器的值已經超過了某個變量的作用域，那麼這個變量對應的 Slot 就可以交給其他變量使用。這樣的設計不僅僅是為了節省空間，在某些情況下 Slot 的復用會直接影響到系統的而垃圾收集行為。

2、操作數棧

操作數棧又常被稱為操作棧，操作數棧的最大深度也是在編譯的時候就確定了。32 位數據類型所佔的棧容量為 1,64

位數據類型所佔的棧容量為 2。當一個方法開始執行時，它的操作棧是空的，在方法的執行過程中，會有各種字節碼指令（比如：加操作、賦值元算等）向操作棧中寫入和提取內容，也就是入棧和出棧操作。

Java 虛擬機的解釋執行引擎稱為「基於棧的執行引擎」，其中所指的「棧」就是操作數棧。因此我們也稱 Java 虛擬機是基於棧的，這點不同於 Android 虛擬機，Android 虛擬機是基於寄存器的。

基於棧的指令集最主要的優點是可移植性強，主要的缺點是執行速度相對會慢些；而由於寄存器由硬件直接提供，所以基於寄存器指令集最主要的優點是執行速度快，主要的缺點是可移植性差。

3、動態連接

每個棧幀都包含一個指向運行時常量池（在方法區中，後面介紹）中該棧幀所屬方法的引用，持有這個引用是為了支持方法調用過程中的動態連接。Class 文件的常量池中存在有大量的符號引用，字節碼中的方法調用指令就以常量池中指向方法的符號引用為參數。這些符號引用，一部分會在類加載階段或第一次使用的時候轉化為直接引用（如 final、static 域等），稱為靜態解析，另一部分將在每一次的運行期間轉化為直接引用，這部分稱為動態連接。

4、方法返回地址

當一個方法被執行後，有兩種方式退出該方法：執行引擎遇到了任意一個方法返回的字節碼指令或遇到了異常，並且該異常沒有在方法體內得到處理。無論採用何種退出方式，在方法退出之後，都需要返回到方法被調用的位置，程序才能繼續執行。方法返回時可能需要在棧幀中保存一些信息，用來幫助恢復它的上層方法的執行狀態。一般來說，方法正常退出時，調用者的 PC 計數器的值就可以作為返回地址，棧幀中很可能保存了這個計數器值，而方法異常退出時，返回地址是要通過異常處理器來確定的，棧幀中一般不會保存這部分信息。

方法退出的過程實際上等同於把當前棧幀出站，因此退出時可能執行的操作有：恢復上層方法的局部變量表和操作數棧，如果有返回值，則把它壓入調用者棧幀的操作數棧中，調整 PC 計數器的值以指向方法調用指令後面的一條指令。

本地方法棧

該區域與虛擬機棧所發揮的作用非常相似，只是虛擬機棧為虛擬機執行 Java 方法服務，而本地方法棧則為使用到的本地操作系統（Native）方法服務。

Java 堆

Java Heap 是 Java 虛擬機所管理的內存中最大的一塊，它是所有線程共享的一塊內存區域。幾乎所有的對象實例和數組都在這類分配內存。Java Heap 是垃圾收集器管理的主要區域，因此很多時候也被稱為「GC堆」。

根據 Java 虛擬機規範的規定，Java 堆可以處在物理上不連續的內存空間中，只要邏輯上是連續的即可。如果在堆中沒有內存可分配時，並且堆也無法擴展時，將會拋出 OutOfMemoryError 異常。

方法區

方法區也是各個線程共享的內存區域，它用於存儲已經被虛擬機加載的類信息、常量、靜態變量、即時編譯器編譯後的代碼等數據。方法區域又被稱為「永久代」，但這僅僅對於 Sun HotSpot 來講，JRockit 和 IBM J9 虛擬機中並不存在永久代的概念。Java 虛擬機規範把方法區描述為 Java 堆的一個邏輯部分，而且它和 Java Heap 一樣不需要連續的內存，可以選擇固定大小或可擴展，另外，虛擬機規範允許該區域可以選擇不實現垃圾回收。相對而言，垃圾收集行為在這個區域比較少出現。該區域的內存回收目標主要針是對廢棄常量的和無用類的回收。運行時常量池是方法區的一部分，Class 文件中除了有類的版本、字段、方法、接口等描述信息外，還有一項信息是常量池（Class文件常量池），用於存放編譯器生成的各種字面量和符號引用，這部分內容將在類加載後存放到方法區的運行時常量池中。運行時常量池相對於 Class 文件常量池的另一個重要特徵是具備動態性，Java 語言並不要求常量一定只能在編譯期產生，也就是並非預置入 Class 文件中的常量池的內容才能進入方法區的運行時常量池，運行期間也可能將新的常量放入池中，這種特性被開發人員利用比較多的是 String 類的 intern（）方法。

根據 Java 虛擬機規範的規定，當方法區無法滿足內存分配需求時，將拋出 OutOfMemoryError 異常。

直接內存

直接內存並不是虛擬機運行時數據區的一部分，也不是 Java 虛擬機規範中定義的內存區域，它直接從操作系統中分配，因此不受 Java 堆大小的限制，但是會受到本機總內存的大小及處理器尋址空間的限制，因此它也可能導致 OutOfMemoryError 異常出現。在 JDK1.4 中新引入了 NIO 機制，它是一種基於通道與緩衝區的新 I/O 方式，可以直接從操作系統中分配直接內存，即在堆外分配內存，這樣能在一些場景中提高性能，因為避免了在 Java 堆和 Native 堆中來回覆制數據。

內存溢出

下面給出個內存區域內存溢出的簡單測試方法。

<table>
<tr><th>内存区域</th><th colspan="2">内存溢出的测试方法</th></tr>
<tr><td>Java 堆</td><td colspan="2">无限循环地 new 对象出来，在 List 中保存引用，以不被垃圾收集器回收。另外，该区域也有可能会发生内存泄露（Memory Leak），
出现问题时，要注意区别。</td></tr>
<tr><td>方法区</td><td colspan="2">生成大量的动态类，或无线循环调用 String 的 intern（）方法产生不同的 String 对象实例，并在 List 中保存其引用，以不被垃圾收集器回收。后者测试常量池，前者测试方法区的非常量池部分。</td></tr>
<tr><td rowspan="2">虚拟机栈和
本地方法栈</td><td>单线程</td><td>多线程</td></tr>
<tr><td>递归调用一个简单的方法：
如不断累积的方法。
会抛出 StackOverflowError</td><td>无线循环地创建线程，并未每个线程无限循环地增加内存。
会抛出 OutOfMemoryError</td></tr>
</table>

這裡有一點要重點說明，在多線程情況下，給每個線程的棧分配的內存越大，反而越容易產生內存溢出異常。操作系統為每個進程分配的內存是有限制的，虛擬機提供了參數來控制 Java 堆和方法區這兩部分內存的最大值，忽略掉程序計數器消耗的內存（很小），以及進程本身消耗的內存，剩下的內存便給了虛擬機棧和本地方法棧，每個線程分配到的棧容量越大，可以建立的線程數量自然就越少。因此，如果是建立過多的線程導致的內存溢出，在不能減少線程數的情況下，就只能通過減少最大堆和每個線程的棧容量來換取更多的線程。

另外，由於 Java 堆內也可能發生內存洩露（Memory Leak），這裡簡要說明一下內存洩露和內存溢出的區別：

內存洩露是指分配出去的內存沒有被回收回來，由於失去了對該內存區域的控制，因而造成了資源的浪費。Java 中一般不會產生內存洩露，因為有垃圾回收器自動回收垃圾，但這也不絕對，當我們 new 了對象，並保存了其引用，但是後面一直沒用它，而垃圾回收器又不會去回收它，這邊會造成內存洩露，

內存溢出是指程序所需要的內存超出了系統所能分配的內存（包括動態擴展）的上限。

對象實例化分析

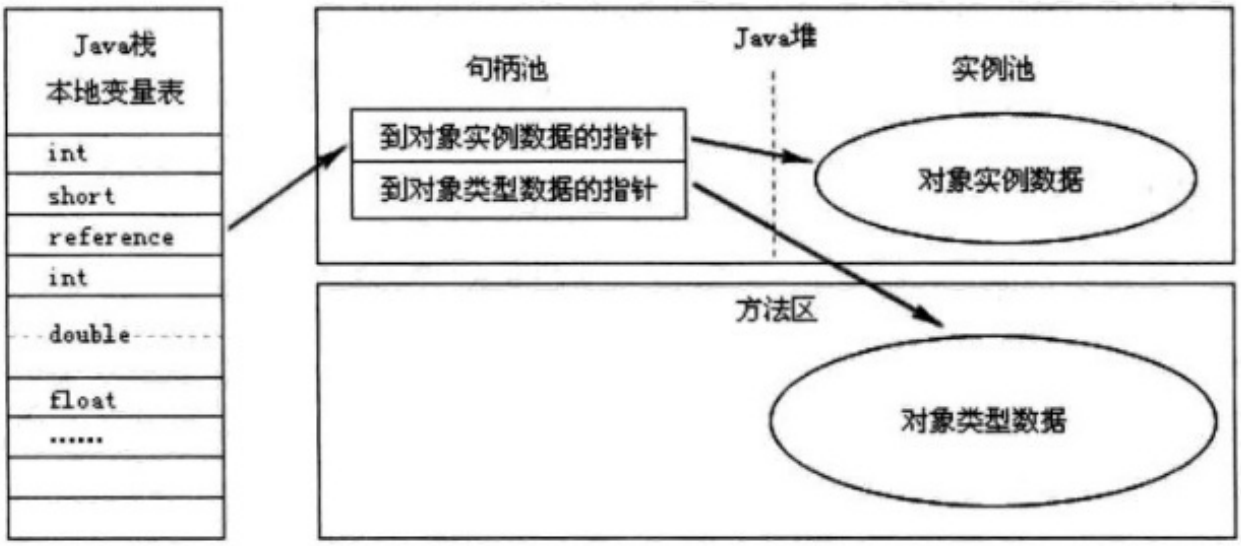
對內存分配情況分析最常見的示例便是對象實例化:

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

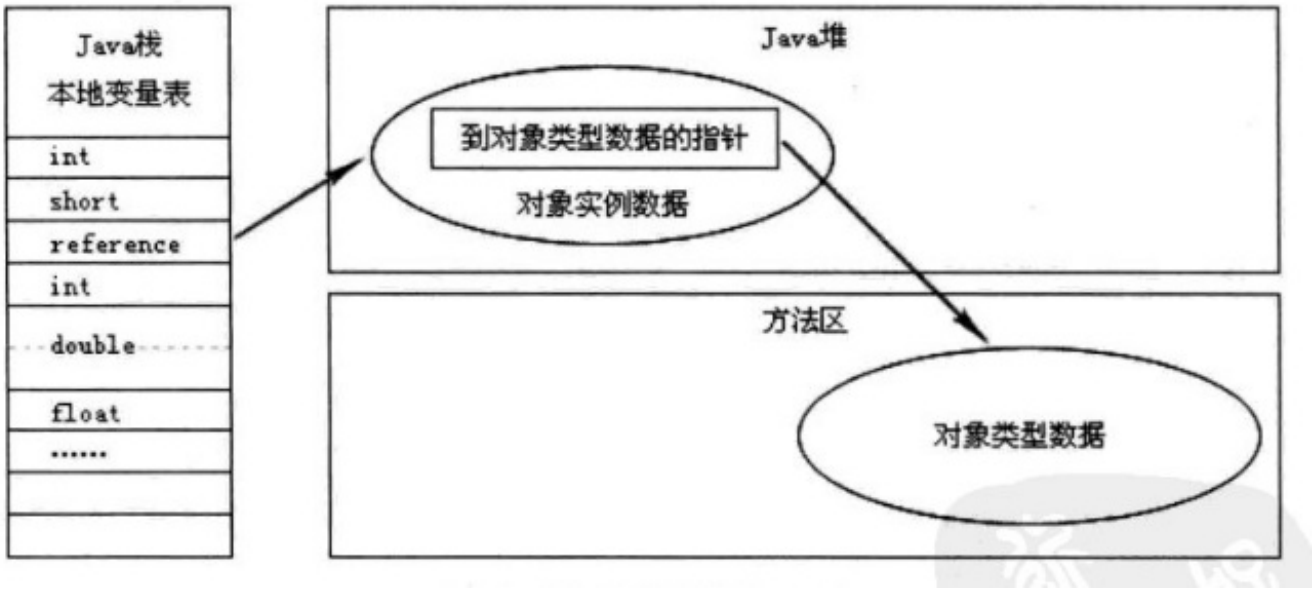
這段代碼的執行會涉及 Java 棧、Java 堆、方法區三個最重要的內存區域。假設該語句出現在方法體中，及時對 JVM 虛擬機不瞭解的 Java 使用這，應該也知道 obj 會作為引用類型（reference）的數據保存在 Java 棧的本地變量表中，而會在 Java 堆中保存該引用的實例化對象，但可能並不知道，Java 堆中還必須包含能查找到此對象類型數據的地址信息（如對象類型、父類、實現的接口、方法等），這些類型數據則保存在方法區中。

另外，由於 reference 類型在 Java 虛擬機規範裡面只規定了一個指向對象的引用，並沒有定義這個引用應該通過哪種方式去定位，以及訪問到 Java 堆中的對象的具體位置，因此不同虛擬機實現的對象訪問方式會有所不同，主流的訪問方式有兩種：使用句柄池和直接使用指針。

通過句柄池訪問的方式如下：



通過直接指針訪問的方式如下：



這兩種對象的訪問方式各有優勢，使用句柄訪問方式的最大好處就是 reference 中存放的是穩定的句柄地址，在對象被移動（垃圾收集時移動對象是非常普遍的行為）時只會改變句柄中的實例數據指針，而 reference 本身不需要修改。使用直接指針訪問方式的最大好處是速度快，它節省了一次指針定位的時間開銷。目前 Java 默認使用的 HotSpot 虛擬機採用的便是是第二種方式進行對象訪問的。