

java內存分配和String類型的深度解析

 importnew.com/15671.html

原文出處：[OSChina-肖慧](#)

一、引題

在java語言的所有數據類型中，String類型是比較特殊的一種類型，同時也是面試的時候經常被問到的一個知識點，本文結合java內存分配深度分析關於String的許多令人迷惑的問題。下面是本文將要涉及到的一些問題，如果讀者對這些問題都瞭如指掌，則可忽略此文。

1、java內存具體指哪塊內存？這塊內存區域為什麼要進行劃分？是如何劃分的？劃分之後每塊區域的作用是什麼？如何設置各個區域的大小？

2、String類型在執行連接操作時，效率為什麼會比StringBuffer或者StringBuilder低？StringBuffer和StringBuilder有什麼聯繫和區別？

3、java中常量是指什麼？String s = 「s」 和 String s = new String(「s」) 有什麼不一樣？

本文經多方資料的收集整理和歸納，最終撰寫成文，如果有錯誤之處，請多多指教！

二、java內存分配

1、JVM簡介

Java虛擬機（Java Virtual Machine 簡稱JVM）是運行所有Java程序的抽象計算機，是Java語言的運行環境，它是Java 最具吸引力的特性之一。Java虛擬機有自己完善的硬體架構，如處理器、堆棧、寄存器等，還具有相應的指令系統。JVM屏蔽了與具體操作系統平台相關的信息，使得Java程序只需生成在Java虛擬機上運行的目標代碼（字節碼），就可以在多種平台上不加修改地運行。

一個運行時的Java虛擬機實例的天職是：負責運行一個java程序。當啟動一個Java程序時，一個虛擬機實例也就誕生了。當該程序關閉退出，這個虛擬機實例也就隨之消亡。如果同一台計算機上同時運行三個Java程序，將得到三個Java虛擬機實例。每個Java程序都運行於它自己的Java虛擬機實例中。

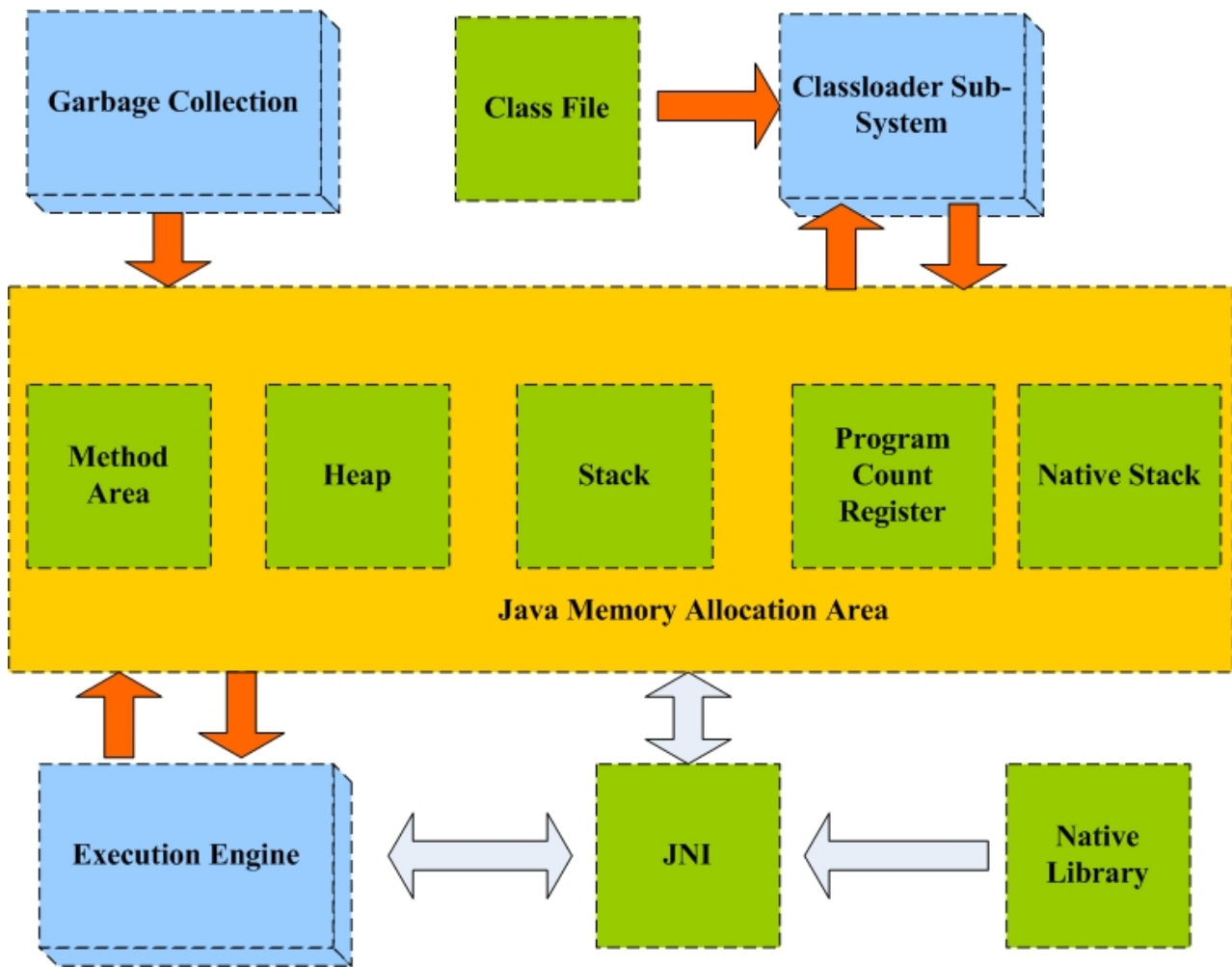
如下圖所示，JVM的體系結構包含幾個主要的子系統和內存區：

垃圾回收器（Garbage Collection）：負責回收堆內存（Heap）中沒有被使用的對象，即這些對象已經沒有被引用了。

類裝載子系統（Classloader Sub-System）：除了要定位和導入二進制class文件外，還必須負責驗證被導入類的正確性，為類變量分配並初始化內存，以及幫助解析符號引用。

執行引擎（Execution Engine）：負責執行那些包含在被裝載類的方法中的指令。

運行時數據區（Java Memory Allocation Area）：又叫虛擬機內存或者Java內存，虛擬機運行時需要從整個計算機內存劃分一塊內存區域存儲許多東西。例如：字節碼、從已裝載的class文件中得到的其他信息、程序創建的對象、傳遞給方法的參數，返回值、局部變量等等。



2、java內存分區

從上節知道，運行時數據區即是java內存，而且數據區要存儲的東西比較多，如果不對這塊內存區域進行劃分管理，會顯得比較雜亂無章。程序喜歡有規律的東西，最討厭雜亂無章的東西。根據存儲數據的不同，java內存通常被劃分為5個區域：程序計數器（Program Count Register）、本地方法棧（Native Stack）、方法區（Methon Area）、棧（Stack）、堆（Heap）。

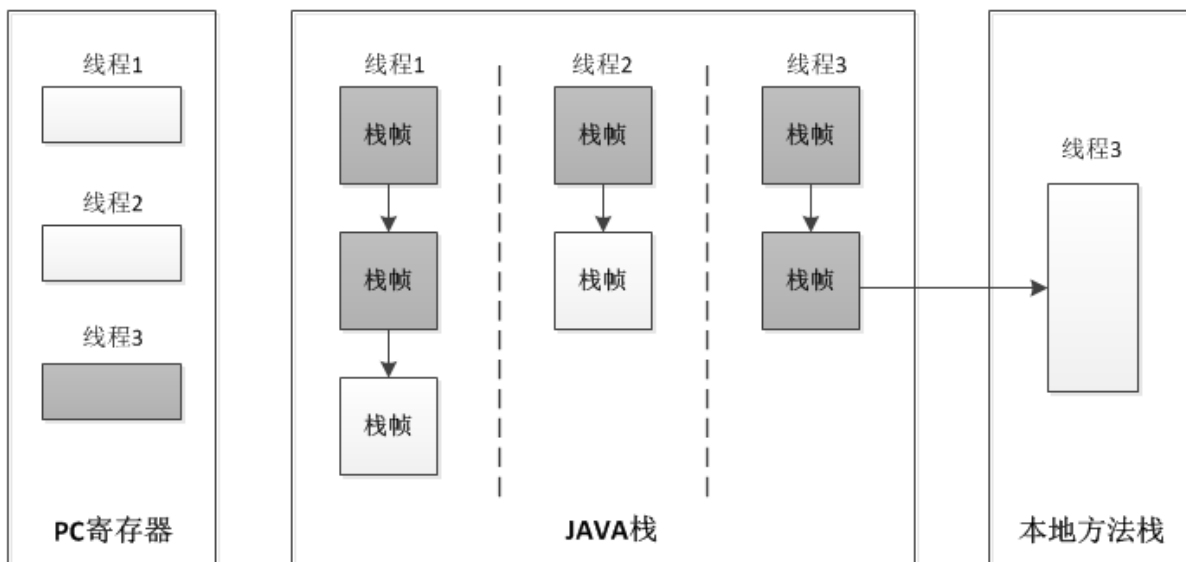
程序計數器（Program Count Register）：又叫程序寄存器。JVM支持多個線程同時運行，當每一個新線程被創建時，它都將得到它自己的PC寄存器（程序計數器）。如果線程正在執行的是一個Java方法（非native），那麼PC寄存器的值將總是指向下一條將被執行的指令，如果方法是 native的，程序計數器寄存器的值不會被定義。JVM的程序計數器寄存器的寬度足夠保證可以持有一個返回地址或者native的指針。

棧（Stack）：又叫堆棧。JVM為每個新創建的線程都分配一個棧。也就是說，對於一個Java程序來說，它的運行就是通過對棧的操作來完成的。棧以幀為單位保存線程的狀態。JVM對棧只進行兩種操作：以幀為單位的壓棧和出棧操作。我們知道,某個線程正在執行的方法稱為此線程的當前方法。我們可能不知道，當前方法使用的幀稱為當前幀。當線程激活一個Java方法，JVM就會在線程的 Java堆棧裡新壓入一個幀，這個幀自然成為了當前幀。在此方法執行期間，這個幀將用來保存參數、局部變量、中間計算過程和其他數據。從Java的這種分配機制來看,堆棧又可以這樣理解：棧(Stack)是操作系統在建立某個進程時或者線程(在支持多線程的操作系統中是線程)為這個線程建立的存儲區域，該區域具有先進後出的特性。其相關設置參數：

- -Xss –設置方法棧的最大值

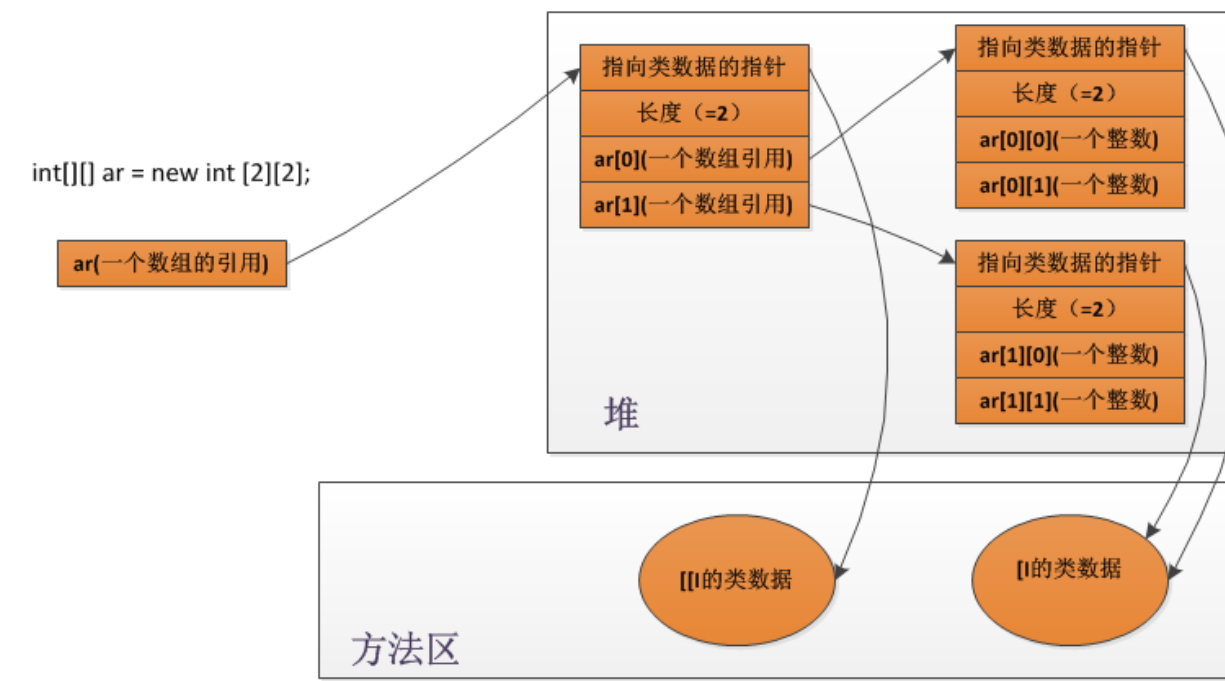
本地方法棧（Native Stack）：存儲本地方法的調用狀態。

线程专有的运行时数据区

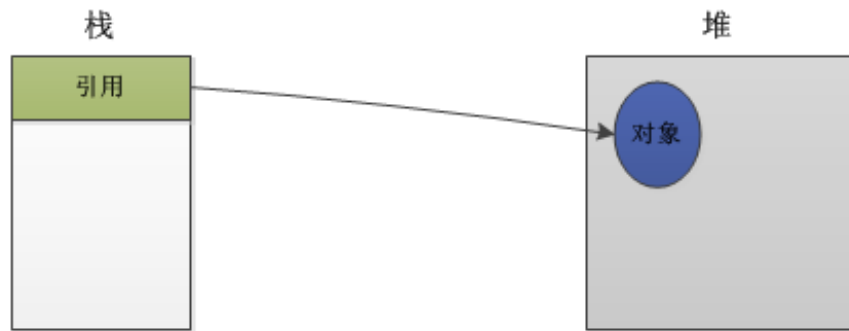


方法区 (Method Area) : 當虛擬機裝載一個class文件時，它會從這個class文件包含的二進制數據中解析類型信息，然後把這些類型信息（包括類信息、常量、靜態變量等）放到方法區中，該內存區域被所有線程共享，如下圖所示。本地方法區存在一塊特殊的內存區域，叫常量池（Constant Pool），這塊內存將與String類型的分析密切相關。

用堆表示数组的一种可能



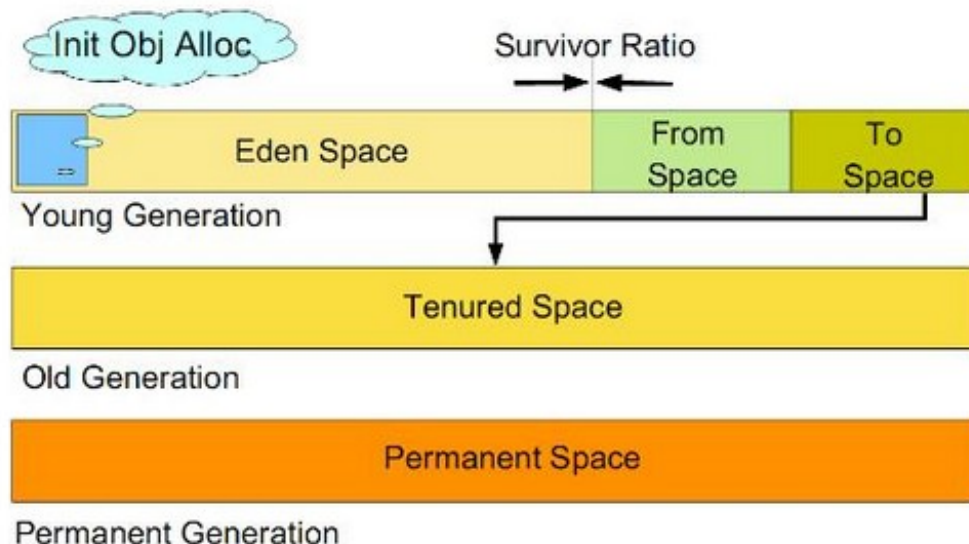
堆 (Heap) : Java堆 (Java Heap) 是Java虛擬機所管理的內存中最大的一塊。Java堆是被所有線程共享的一塊內存區域。在此區域的唯一目的就是存放對象實例，幾乎所有的對象實例都是在此處分配內存，但是這個對象的引用卻是在棧 (Stack) 中分配。因此，執行 `String s = new String(「s」)` 時，需要從兩個地方分配內存：在堆中為String對象分配內存，在棧中為引用（這個堆對象的內存地址，即指針）分配內存，如下圖所示。



JAVA虛擬機有一條在堆中分配新對象的指令，卻沒有釋放內存的指令，正如你無法用Java代碼區明確釋放一個對象一樣。虛擬機自己負責決定如何以及何時釋放不再被運行的程序引用的對象所佔據的內存，通常，虛擬機把這個任務交給垃圾收集器（Garbage Collection）。其相關設置參數：

- -Xms — 設置堆內存初始大小
- -Xmx — 設置堆內存最大值
- -XX:MaxTenuringThreshold — 設置對象在新生代中存活的次數
- -XX:PretenureSizeThreshold — 設置超過指定大小的大對象直接分配在舊世代中

Java堆是垃圾收集器管理的主要區域，因此又稱為「GC 堆」（Garbage Collectioned Heap）。現在的垃圾收集器基本都是採用的分代收集算法，所以Java堆還可以細分為：新生代（Young Generation）和老年代（Old Generation），如下圖所示。分代收集算法的思想：第一種說法，用較高的頻率對年輕的對象(young generation)進行掃描和回收，這種叫做minor collection，而對老對象(old generation)的檢查回收頻率要低很多，稱為major collection。這樣就不需要每次GC都將內存中所有對象都檢查一遍，以便讓出更多的系統資源供應用系統使用；另一種說法，在分配對象遇到內存不足時，先對新生代進行GC（Young GC）；當新生代GC之後仍無法滿足內存空間分配需求時，才會對整個堆空間以及方法區進行GC（Full GC）。



在這裡可能會有讀者表示疑問：記得還有一個什麼永久代（**Permanent Generation**）的啊，難道它不屬於Java堆？親，你答對了！其實傳說中的永久代就是上面所說的方法區，存放的都是jvm初始化時加載器加載的一些類型信息（包括類信息、常量、靜態變量等），這些信息的生存週期比較長，GC不會在主程序運行期對PermGen Space進行清理，所以如果你的應用中有很多CLASS的話,就很可能出現PermGen Space錯誤。其相關設置參數：

- -XX:PermSize —設置Perm區的初始大小

- -XX:MaxPermSize –設置Perm區的最大值

新生代 (Young Generation) 又分為：Eden區和Survivor區，Survivor區有分為From Space和To Space。Eden區是對象最初分配到的地方；默認情況下，From Space和To Space的區域大小相等。JVM進行Minor GC時，將Eden中還存活的對象拷貝到Survivor區中，還會將Survivor區中還存活的對象拷貝到Tenured區中。在這種GC模式下，JVM為了提升GC效率，將Survivor區分為From Space和To Space，這樣就可以將對象回收和對象晉陞分離開來。新生代的大小設置有2個相關參數：

- -Xmn — 設置新生代內存大小。
- -XX:SurvivorRatio — 設置Eden與Survivor空間的大小比例

老年代 (Old Generation)：當 OLD 區空間不夠時，JVM 會在 OLD 區進行 major collection；完全垃圾收集後，若Survivor及OLD區仍然無法存放從Eden複製過來的部分對象，導致JVM無法在Eden區為新對象創建內存區域，則出現「Out of memory錯誤」。

三、String類型的深度解析

讓我們從Java數據類型開始說起吧！Java數據類型通常（分類方法多種多樣）從整體上可以分為兩大類：基礎類型和引用類型，基礎類型的變量持有原始值，引用類型的變量通常表示的是對實際對象的引用，其值通常為對象的內存地址。對於基礎類型和引用類型的細分，直接上圖吧，大家看了一目瞭然。當然，下圖也僅僅只是其中的一種分類方式。

（原文圖丟失）

針對上面的圖，有3點需要說明：

- char類型可以單獨出來形成一類，很多基本類型的分類為：數值類型、字符型（char）和bool型。
- returnAddress類型是一個Java虛擬機在內部使用的類型，被用來實現Java程序中的finally語句。
- String類型在上圖的什麼位置？yes，屬於引用類型下面的類類型。下面開始對String類型的挖掘！

1、String的本質

打開String的源碼，類註釋中有這麼一段話「Strings are constant; their values cannot be changed after they are created. String buffers support mutable strings.Because String objects are immutable they can be shared.」。這句話總結歸納了String的一個最重要的特點：String是值不可變(immutable)的常量，是線程安全的(can be shared)。

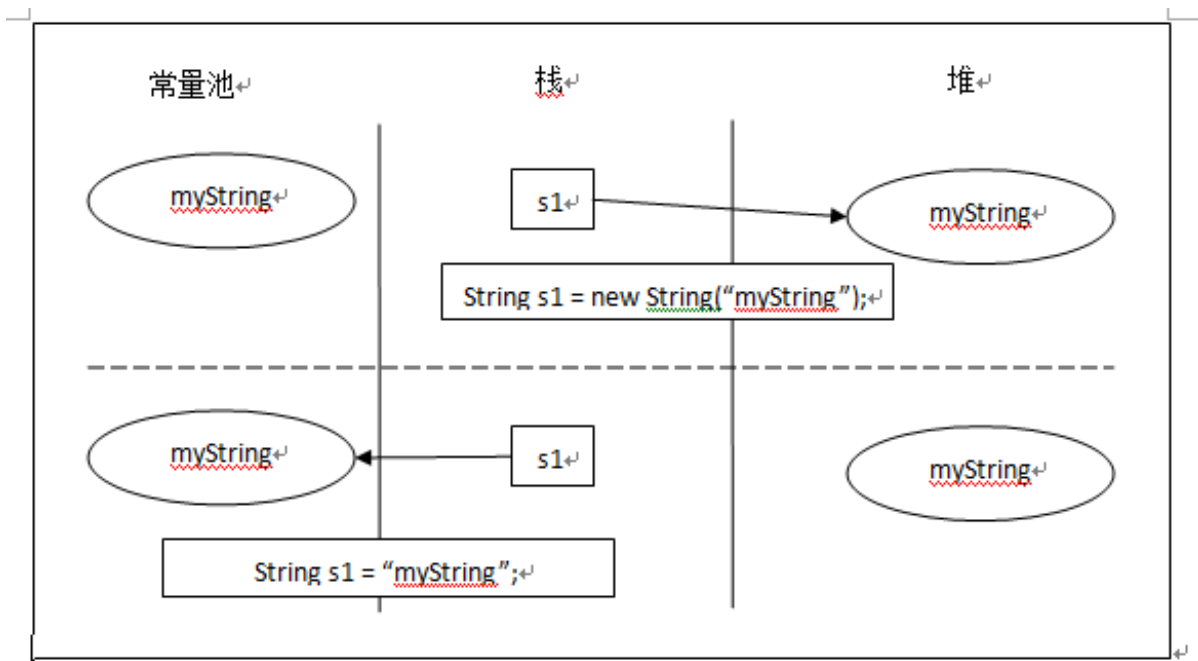
接下來，String類使用了final修飾符，表明了String類的第二個特點：String類是不可繼承的。

下面是String類的成員變量定義，從類的實現上闡明了String值是不可變的(immutable)。

```
private final char value[];  
private final int count;
```

因此，我們看String類的concat方法。實現該方法第一步要做的肯定是擴大成員變量value的容量，擴容的方法重新定義一個大容量的字符數組buf。第二步就是把原來value中的字符copy到buf中來，再把需要concat的字符串值也copy到buf中來，這樣子，buf中就包含了concat之後的字符串值。下面就是問題的關鍵了，如果value不是final的，直接讓value指向buf，然後返回this，則大功告成，沒有必要返回一個新的String對象。但是。。。可惜。。。由於value是final型的，所以無法指向新定義的大容量數組buf，那怎麼辦呢？「return new String(0, count + otherLen, buf);」，這是String類concat實現方法的最後一條語句，重新new一個String對象返回。這下真相大白了吧！

總結：String實質是字符數組，兩個特點：1、該類不可被繼承；2、不可變性(immutable)。



2、String的定義方法

在討論String的定義方法之前，先瞭解一下常量池的概念，前面在介紹方法區的時候已經提到過了。下面稍微正式的給一個定義吧。

常量池(constant pool)指的是在編譯期被確定，並被保存在已編譯的.class文件中的一些數據。它包括了關於類、方法、接口等中的常量，也包括字符串常量。常量池還具備動態性，運行期間可以將新的常量放入池中，String類的intern()方法是這一特性的典型應用。不懂嗎？後面會介紹intern方法的。虛擬機為每個被裝載的類型維護一個常量池，池中為該類型所用常量的一個有序集合，包括直接常量(string、integer和float常量)和對其他類型、字段和方法的符號引用（與對象引用的區別？讀者可以自己去瞭解）。

String的定義方法歸納起來總共為三種方式：

- 使用關鍵字new，如：String s1 = new String(「myString」);
- 直接定義，如：String s1 = 「myString」;
- 串聯生成，如：String s1 = 「my」 + 「String」;這種方式比較複雜，這裡就不贅述了，請參見[java-String常量池問題的幾個例子](#)。

第一種方式通過關鍵字new定義過程：在程序編譯期，編譯程序先去字符串常量池檢查，是否存在「myString」,如果不存在，則在常量池中開闢一個內存空間存放「myString」；如果存在的話，則不用重新開闢空間，保證常量池中只有一個「myString」常量，節省內存空間。然後在內存堆中開闢一塊空間存放new出來的String實例，在棧中開闢一塊空間，命名為「s1」，存放的值為堆中String實例的內存地址，這個過程就是將引用s1指向new出來的String實例。各位，最模糊的地方到了！堆中新出來的實例和常量池中的「myString」是什麼關係呢？等我們分析完了第二種定義方式之後再回頭分析這個問題。

第二種方式直接定義過程：在程序編譯期，編譯程序先去字符串常量池檢查，是否存在「myString」，如果不存在，則在常量池中開闢一個內存空間存放「myString」；如果存在的話，則不用重新開闢空間。然後在棧中開闢一塊空間，命名為「s1」，存放的值為常量池中「myString」的內存地址。常量池中的字符串常量與堆中的String對象有什麼區別呢？為什麼直接定義的字符串同樣可以調用String對象的各種方法呢？

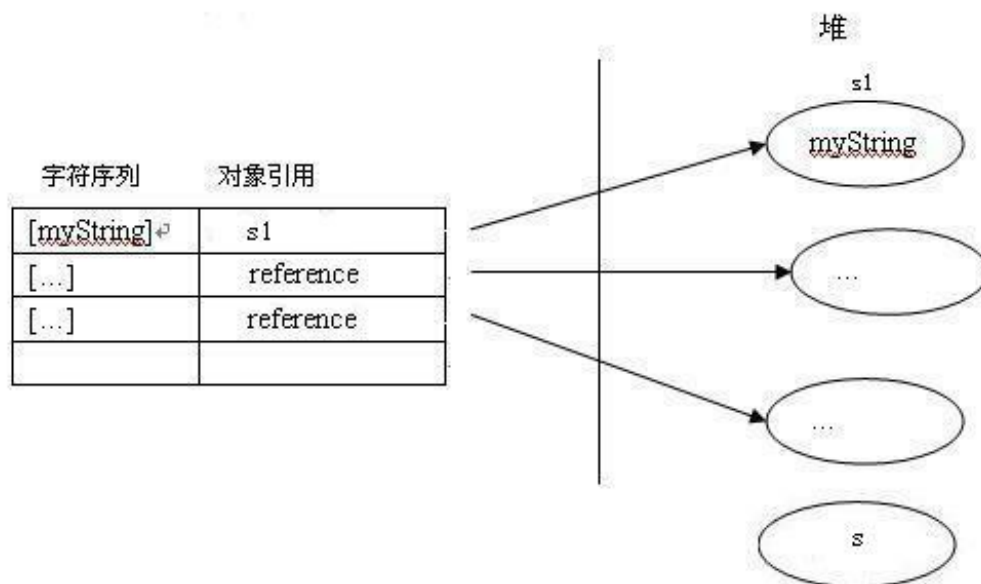
帶著諸多疑問，我和大家一起探討一下堆中String對象和常量池中String常量的關係，請大家記住，僅僅是探討，因為本人對這塊也比較模糊。

第一種猜想：因為直接定義的字符串也可以調用String對象的各種方法，那麼可以認為其實在常量池中創建的也是一個String實例（對象）。String s1 = new String(「myString」);先在編譯期的時候在常量池創建了一個String實例，然後clone了一個String實例存儲在堆中，引用s1指向堆中的這個實例。此時，池中的實例沒有被引用。當接著執行String s1 = 「myString」;時，因為池中已經存在「myString」的實例對象，則s1直接指向池中的實例對象；否則，在池中先創建一個實例對象，s1再指向它。如下圖所示：

這種猜想認為：常量池中的字符串常量實質上是一個String實例，與堆中的String實例是克隆關係。

第二種猜想也是目前網上闡述的最多的，但是思路都不清晰，有些問題解釋不通。下面引用《[JAVA String對象和字符串常量的關係解析](#)》一段內容。

在解析階段，虛擬機發現字符串常量「myString」，它會在一個內部字符串常量列表中查找，如果沒有找到，那麼會在堆裡面創建一個包含字符序列[myString]的String對象s1，然後把這個字符序列和對應的String對象作為名值對([myString], s1)保存到內部字符串常量列表中。如下圖所示：



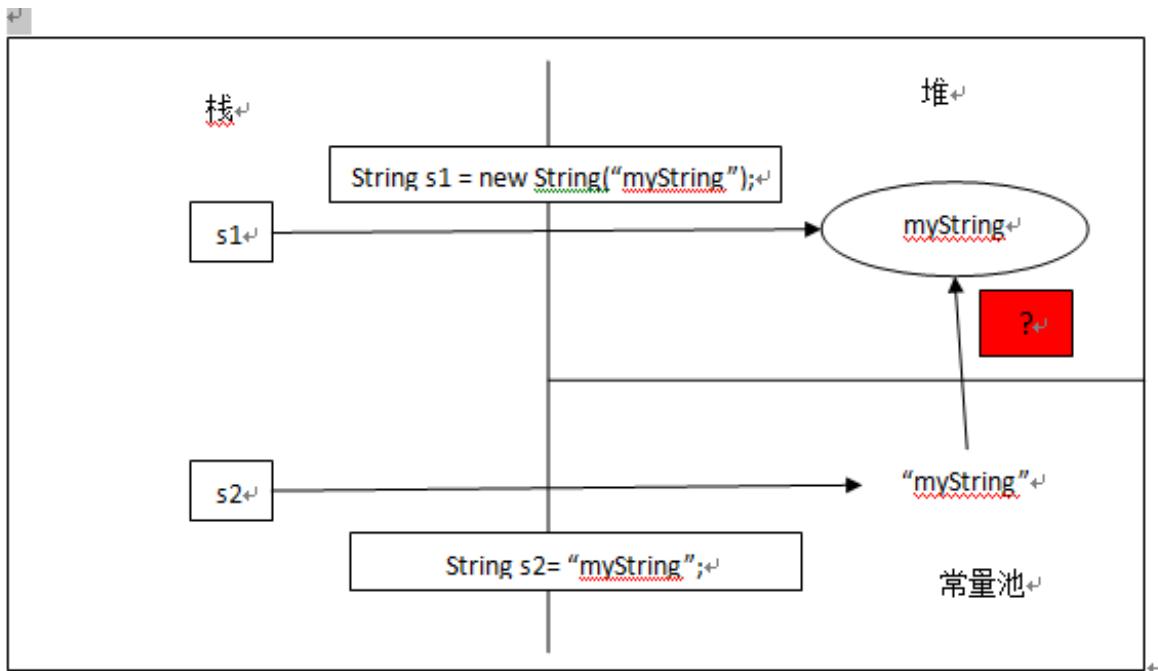
如果虛擬機後面又發現了一個相同的字符串常量myString，它會在這個內部字符串常量列表內找到相同的字符序列，然後返回對應的String對象的引用。維護這個內部列表的關鍵是任何特定的字符序列在這個列表上只出現一次。

例如，String s2 = 「myString」，運行時s2會從內部字符串常量列表內得到s1的返回值，所以s2和s1都指向同一個String對象。

這個猜想有一個比較明顯的問題，紅色字體標示的地方就是問題的所在。證明方式很簡單，下面這段代碼的執行結果，javaer都應該知道。

```
String s1 = new String(「myString」);  
String s2 = 「myString」;
```

System.out.println(s1 == s2); //按照上面的推測邏輯，那麼打印的結果為true；而實際上真實的結果是false，因為s1指向的是堆中String對象，而s2指向的是常量池中的String常量。



雖然這段內容不那麼有說服力，但是文章提到了一個東西——字符串常量列表，它可能是解釋這個問題的關鍵。

文中提到的三個問題，本文僅僅給出了猜想，請知道真正內幕的高手幫忙分析分析，謝謝！

- 堆中new出來的實例和常量池中的「myString」是什麼關係呢？
- 常量池中的字符串常量與堆中的String對象有什麼區別呢？
- 為什麼直接定義的字符串同樣可以調用String對象的各種方法呢？

3、String、StringBuffer、StringBuilder的聯繫與區別

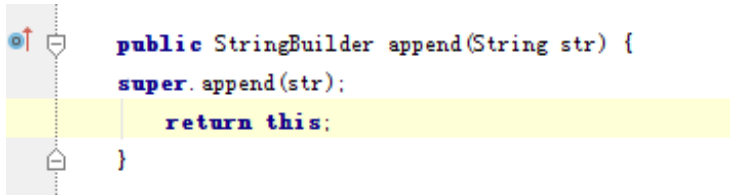
上面已經分析了String的本質了，下面簡單說說StringBuffer和StringBuilder。

StringBuffer和StringBuilder都繼承了抽象類AbstractStringBuilder，這個抽象類和String一樣也定義了char[] value和int count，但是與String類不同的是，它們沒有final修飾符。因此得出結論：**String、StringBuffer和StringBuilder在本質上都是字符數組**，不同的是，在進行連接操作時，**String**每次返回一個新的String實例，而**StringBuffer**和**StringBuilder**的append方法直接返回this，所以這就是為什麼在進行大量字符串連接運算時，不推薦使用String，而推薦StringBuffer和StringBuilder。那麼，哪種情況使用StringBuffer？哪種情況使用StringBuilder呢？

關於StringBuffer和StringBuilder的區別，翻開它們的源碼，下面貼出append()方法的實現。

```
public synchronized StringBuffer append(String str) {
    super.append(str);
    return this;
}
```


面第一張圖是StringBuffer中append()方法的實現，第二張圖為StringBuilder對append()的實現。區別應該一目瞭然，StringBuffer在方法前加了一個synchronized修飾，起到同步的作用，可以在多線程環境使用。為此付出的代價就是降低了執行效率。因此，如果在多線程環境可以使用StringBuffer進行字符串連接操作，單線程環境使用StringBuilder，它的效率更高。



```
public StringBuilder append(String str) {  
    super.append(str);  
    return this;  
}
```

四、參考文獻

[Java虛擬機體系結構](#)

[Java內存管理基礎篇-Java內存分配](#)

[Java堆內存設置優化](#)

[Java內存管理和垃圾回收](#)

[Java堆內存的轉換和回收](#)

[Java虛擬機的JVM垃圾回收機制](#)

[淺談設置JVM內存分配的幾個妙招](#)

[深入Java字符串](#)

[Java性能優化之String篇](#)

[java字符串常量池知識](#)

[Java內存分配及String類型詳解](#)

[Java String的內存機制](#)

[Java之內存分析和String對象](#)

[String類學習總結](#)