|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Primitive type | Size | Minimum | Maximum | Wrapper type |
| boolean | — | — | — | Boolean |
| char | 16 bits | Unicode 0 | Unicode 216- 1 | Character |
| byte | 8 bits | -128 | +127 | Byte |
| short | 16 bits | -215 | +215-1 | Short |
| int | 32 bits | -231 | +231-1 | Integer |
| long | 64 bits | -263 | +263-1 | Long |
| float | 32 bits | IEEE754 | IEEE754 | Float |
| double | 64 bits | IEEE754 | IEEE754 | Double |
| void | — | — | — | Void |

{

int x = 12;

// Only x available

{

int q = 96;

// Both x & q available

}

// Only x available

// q is "out of scope"

}

# ArrayList是JAVA當中的一個類別

與Array差在於因為他是一個寫好的類別,有很多可以直接用的程式碼,所以很方便

使用方法:

1.建構

ArrayList<String> myList = new ArrayList<String>();  //指定是String的型態

ArrayList myList = new ArrayList(); //可以不指定

2.加入元素

String s = new String();

myList.add(s);

 3.查詢list大小

int theSize = myList.size();

 4.查詢特定元素

boolean isIn = myList.contains(s);  //若用上面的例子 因為有s字串 所以回傳true

 5.查詢特定元素位置

int idx = myList.indexOf(s); //會回傳0 表第0個位置

 6.判斷List是否為空

boolean empty = myList.isEmpty(); //因為有一個元素 會回傳false

 7.刪除特定元素

myList.remove(s);

# JavaDoc，在Java的注釋上做文章

2011/7/31

對于Java注釋我們主要了解兩種：

// 注釋一行

/\* ...... \*/ 注釋若干行

但還有第三種，文檔注釋：

/\*\* ...... \*/ 注釋若干行，并寫入 javadoc 文檔

通常這種注釋的多行寫法如下：

/\*\*

\* .........

\* .........

\*/

很多人多忽視了這第三種注釋，那么這第三種注釋有什么用？javadoc 又是什么東西？下面我們就談談。

**一. Java 文檔和 Javadoc**

Java 程序員都應該知道使用 JDK 開發，最好的幫助信息就來自 SUN 發布的 Java 文檔。它分包、分類詳細的提供了各方法、屬性的幫助信息，具有詳細的類樹信息、索引信息等，并提供了許多相關類之間的關系，如繼承、實現接口、引用等。

Java 文檔全是由一些 html 文件組織起來的，在 SUM 的站點上可以下載它們的壓縮包。但是你肯定想不到，這些文檔我們可以自己生成。——就此打住，再吊一次胃口。

安裝了 JDK 之后，安裝目錄下有一個 src.jar 文件或者 src.zip 文件，它們都是以 ZIP 格式壓縮的，可以使用 WinZip 解壓。解壓之后，我們就可以看到分目錄放的全是 .java 文件。是了，這些就是 Java 運行類的源碼了，非常完整，連注釋都寫得一清二楚……不過，怎么看這些注釋都有點似曾相識的感覺？

這就不奇怪了，我們的迷底也快要揭開了。如果你仔細對比一下 .java 源文件中的文檔注釋 (/\*\* ... \*/) 和 Java 文檔的內容，你會發現它們就是一樣的。Java 文檔只是還在格式和排版上下了些功夫。再仔細一點，你會發現 .java 源文件中的注釋還帶有 HTML 標識，如 ＜B＞、＜BR＞、＜Code＞ 等，在 Java 文檔中，該出現這些標識的地方，已經按標識的的定義進行了排版。

終于真像大白了，原來 Java 文檔是來自這些注釋。難怪這些注釋叫做文檔注釋呢！不過，是什么工具把這些注釋變成文檔的呢？

是該請出 javadoc 的時候了。在 JDK 的 bin 目錄下你可以找到 javadoc，如果是 Windows 下的 JDK，它的文件名為 javadoc.exe。使用 javdoc 編譯 .java 源文件時，它會讀出 .java 源文件中的文檔注釋，并按照一定的規則與 Java 源程序一起進行編譯，生成文檔。

介紹 javadoc 的編譯命令之前，還是先了解一下文檔注釋的格式吧。不過為了能夠編譯下面提到的若干例子，這里先介紹一條 javadoc 命令：

javadoc -d 文檔存放目錄 -author -version 源文件名.java

這條命令編譯一個名為 “源文件名.java”的 java 源文件，并將生成的文檔存放在“文檔存放目錄”指定的目錄下，生成的文檔中 index.html 就是文檔的首頁。-author 和 -version 兩上選項可以省略。

**二. 文檔注釋的格式**

文檔注釋可以用于對類、屬性、方法等進行說明。寫文檔注釋時除了需要使用 /\*\* .... \*/ 限定之外，還需要注意注釋內部的一些細節問題。

1. 文檔和文檔注釋的格式化

生成的文檔是 HTML 格式，而這些 HTML 格式的標識符并不是 javadoc 加的，而是我們在寫注釋的時候寫上去的。比如，需要換行時，不是敲入一個回車符，而是寫入 ＜br＞，如果要分段，就應該在段前寫入 ＜p＞。

因此，格式化文檔，就是在文檔注釋中添加相應的 HTML 標識。

文檔注釋的正文并不是直接復制到輸出文件 (文檔的 HTML 文件)，而是讀取每一行后，刪掉前導的 \* 號及 \* 號以前的空格，再輸入到文檔的。如

/\*\*

\* This is first line. ＜br＞

\*\*\*\*\* This is second line. ＜br＞

This is third line.

\*/

編譯輸出后的 HTML 源碼則是

This is first line. ＜br＞

This is second line. ＜br＞

This is third line.

前導的 \* 號允許連續使用多個，其效果和使用一個 \* 號一樣，但多個 \* 號前不能有其它字符分隔，否則分隔符及后面的 \* 號都將作為文檔的內容。\* 號在這里是作為左邊界使用，如上例的第一行和第二行；如果沒有前導的 \* 號，則邊界從第一個有效字符開始，而不包括前面的空格，如上例第三行。

還有一點需要說明，文檔注釋只說明緊接其后的類、屬性或者方法。如下例：

/\*\* comment for class \*/

public class Test {

/\*\* comment for a attribute \*/

int number;

/\*\* comment for a method \*/

public void myMethod() { ...... }

......

}

上例中的三處注釋就是分別對類、屬性和方法的文檔注釋。它們生成的文檔分別是說明緊接其后的類、屬性、方法的。“緊接”二字尤其重要，如果忽略了這一點，就很可能造成生成的文檔錯誤。如

import java.lang.\*;

/\*\* commnet for class \*/

public class Test { ...... }

// 此例為正確的例子

這個文檔注釋將生成正確的文檔。但只需要改變其中兩行的位置，變成下例，就會出錯：

/\*\* commnet for class \*/

import java.lang.\*;

public class Test { ...... }

// 此例為錯誤的例子

這個例子只把上例的 import 語句和文檔注釋部分交換了位置，結果卻大不相同——生成的文檔中根本就找不到上述注釋的內容了。原因何在？

“/\*\* commnet for class \*/”是對 class Test 的說明，把它放在“public class Test { ...... }”之前時，其后緊接著 class Test，符合規則，所以生成的文檔正確。但是把它和“import java.lang.\*;”調換了位置后，其后緊接的就是不 class Test 了，而是一個 import 語句。由于文檔注釋只能說明類、屬性和方法，import 語句不在此列，所以這個文檔注釋就被當作錯誤說明省略掉了。

2. 文檔注釋的三部分

根據在文檔中顯示的效果，文檔注釋分為三部分。先舉例如下，以便說明。

/\*\*

\* show 方法的簡述.

\* ＜p＞show 方法的詳細說明第一行＜br＞

\* show 方法的詳細說明第二行

\* @param b true 表示顯示，false 表示隱藏

\* @return 沒有返回值

\*/

public void show(boolean b) {

frame.show(b);

}

第一部分是簡述。文檔中，對于屬性和方法都是先有一個列表，然后才在后面一個一個的詳細的說明。列表中屬性名或者方法名后面那段說明就是簡述。如下圖中被紅框框選的部分：

簡述部分寫在一段文檔注釋的最前面，第一個點號 (.) 之前 (包括點號)。換句話說，就是用第一個點號分隔文檔注釋，之前是簡述，之后是第二部分和第三部分。如上例中的 “\* show 方法的簡述.”。

有時，即使正確地以一個點號作為分隔，javadoc 仍然會出錯，把點號后面的部分也做為了第一部分。為了解決這個問題，我們可以使用一個 ＜p＞ 標志將第二分部分開為下一段，如上例的“\* ＜p＞show 方法的詳細說明第一行 ....”。除此之外，我們也可以使用 ＜br＞ 來分隔。

第二部分是詳細說明部分。該部分對屬性或者方法進行詳細的說明，在格式上沒有什么特殊的要求，可以包含若干個點號。它在文檔中的位置如下圖所示：

這部分文檔在上例中相應的代碼是：

\* show 方法的簡述.

\* ＜p＞show 方法的詳細說明第一行＜br＞

\* show 方法的詳細說明第二行

發現什么了？對了，簡述也在其中。這一點要記住了，不要畫蛇添足——在詳細說明部分中再寫一次簡述哦！

第三部分是特殊說明部分。這部分包括版本說明、參數說明、返回值說明等。它在文檔中的位置：

第三部分在上例中相應的代碼是

\* @param b true 表示顯示，false 表示隱藏

\* @return 沒有返回值

除了 @param 和 @return 之外，還有其它的一些特殊標記，分別用于對類、屬性和方法的說明……不要推我，我馬上就說。

**三. 使用 javadoc 標記**

javadoc 標記是插入文檔注釋中的特殊標記，它們用于標識代碼中的特殊引用。javadoc 標記由“@”及其后所跟的標記類型和專用注釋引用組成。記住了，三個部分——@、標記類型、專用注釋引用。不過我寧愿把它分成兩部分：@ 和標記類型、專用注釋引用。雖然 @ 和 標記類型之間有時可以用空格符分隔，但是我寧愿始終將它們緊挨著寫，以減少出錯機會。

javadoc 標記有如下一些：

標記

用于

作用

@author

對類的說明

標明開發該類模塊的作者

@version

對類的說明

標明該類模塊的版本

@see

對類、屬性、方法的說明

參考轉向，也就是相關主題

@param

對方法的說明

對方法中某參數的說明

@return

對方法的說明

對方法返回值的說明

@exception

對方法的說明

對方法可能拋出的異常進行說明

下面詳細說明各標記。

1. @see 的使用

@see 的句法有三種：

@see 類名

@see #方法名或屬性名

@see 類名#方法名或屬性名

類名，可以根據需要只寫出類名 (如 String) 或者寫出類全名 (如 java.lang.String)。那么什么時候只需要寫出類名，什么時候需要寫出類全名呢？

如果 java 源文件中的 import 語句包含了的類，可以只寫出類名，如果沒有包含，則需要寫出類全名。java.lang 也已經默認被包含了。這和 javac 編譯 java 源文件時的規定一樣，所以可以簡單的用 javac 編譯來判斷，源程序中 javac 能找到的類，javadoc 也一定能找到；javac 找不到的類，javadoc 也找不到，這就需要使用類全名了。

方法名或者屬性名，如果是屬性名，則只需要寫出屬性名即可；如果是方法名，則需要寫出方法名以及參數類型，沒有參數的方法，需要寫出一對括號。如

成員類型

成員名稱及參數

@see 句法

屬性

number

@see number

屬性

count

@see count

方法

count()

@see count()

方法

show(boolean b)

@see show(boolean)

方法

main(String[] args)

@see main(String[])

有時也可以偷懶：假如上例中，沒有 count 這一屬性，那么參考方法 count() 就可以簡寫成 @see count。不過，為了安全起見，還是寫全 @see count() 比較好。

@see 的第二個句法和第三個句法都是轉向方法或者屬性的參考，它們有什么區別呢？

第二個句法中沒有指出類名，則默認為當前類。所以它定義的參考，都轉向本類中的屬性或者方法。而第三個句法中指出了類名，則還可以轉向其它類的屬性或者方法。

關于 @see 標記，我們舉個例說明。由于 @see 在對類說明、對屬性說明、對方法說明時用法都一樣，所以這里只以對類說明為例。

/\*\*

\* @see String

\* @see java.lang.StringBuffer

\* @see #str

\* @see #str()

\* @see #main(String[])

\* @see Object#toString()

\*/

public class TestJavaDoc {

}

生成的文檔的相關部分如下圖：

String 和 StringBuffer 都是在 java.lang 包中，由于這個包是默認導入了的，所以這兩個類可以直接寫類名，也可以寫類全名。str、str() 為同名屬性和方法，所以方法名需要用 () 區分。main 是帶參數的方法，所以在 () 中指明了參數類型。toString() 雖然在本類中也有 (從 Object 繼承的)，但我們是想參考 Object 類的 toString() 方法，所以使用了 Object#toString()。

奇怪的是，為什么其中只有 str、str() 和 main(String[]) 變成了鏈接呢？那是因為編譯時沒有把 java.lang 包或者 Stirng、StringBuffer、Object 三個類的源文件一起加入編譯，所以，生成的文檔沒有關于那三個類的信息，也就不可以建立鏈接了。后面講解 javadoc 編譯命令的時候還會詳細說明。

上例中如果去把類中的 str 屬性去掉，那么生成的文檔又會有什么變化呢？你會發現，原來是 str, str()，而現在變成了 str(), str()，因為 str 屬性已經沒有了，所以 str 也表示方法 str()。

2. 使用 @author、@version 說明類

這兩個標記分別用于指明類的作者和版本。缺省情況下 javadoc 將其忽略，但命令行開關 -author 和 -version 可以修改這個功能，使其包含的信息被輸出。這兩個標記的句法如下：

@author 作者名

@version 版本號

其中，@author 可以多次使用，以指明多個作者，生成的文檔中每個作者之間使用逗號 (,) 隔開。@version 也可以使用多次，只有第一次有效，生成的文檔中只會顯示第一次使用 @version 指明的版本號。如下例

/\*\*

\* @author Fancy

\* @author Bird

\* @version Version 1.00

\* @version Version 2.00

\*/

public class TestJavaDoc {

}

生成文檔的相關部分如圖：

從生成文檔的圖示中可以看出，兩個 @author 語句都被編譯，在文檔中生成了作者列表。而兩個 @version 語句中只有第一句被編譯了，只生成了一個版本號。

從圖上看，作者列表是以逗號分隔的，如果我想分行顯示怎么辦？另外，如果我想顯示兩個以上的版本號又該怎么辦？

——我們可以將上述兩條 @author 語句合為一句，把兩個 @version 語句也合為一句：

@author Fancy＜br＞Bird

@version Version 1.00＜br＞Version 2.00

結果如圖：

我們這樣做即達到了目的，又沒有破壞規則。@author 之后的作者名和 @version 之后的版本號都可以是用戶自己定義的任何 HTML 格式，所以我們可以使用 ＜br＞ 標記將其分行顯示。同時，在一個 @version 中指明兩個用 ＜br＞ 分隔的版本號，也沒有破壞只顯示第一個 @version 內容的規則。

3. 使用 @param、@return 和 @exception 說明方法

這三個標記都是只用于方法的。@param 描述方法的參數，@return 描述方法的返回值，@exception 描述方法可能拋出的異常。它們的句法如下：

@param 參數名 參數說明

@return 返回值說明

@exception 異常類名 說明

每一個 @param 只能描述方法的一個參數，所以，如果方法需要多個參數，就需要多次使用 @param 來描述。

一個方法中只能用一個 @return，如果文檔說明中列了多個 @return，則 javadoc 編譯時會發出警告，且只有第一個 @return 在生成的文檔中有效。

方法可能拋出的異常應當用 @exception 描述。由于一個方法可能拋出多個異常，所以可以有多個 @exception。每個 @exception 后面應有簡述的異常類名，說明中應指出拋出異常的原因。需要注意的是，異常類名應該根據源文件的 import 語句確定是寫出類名還是類全名。 示例如下：

public class TestJavaDoc {

/\*\*

\* @param n a switch

\* @param b excrescent parameter

\* @return true or false

\* @return excrescent return

\* @exception java.lang.Exception throw when switch is 1

\* @exception NullPointerException throw when parameter n is null

\*/

public boolean fun(Integer n) throws Exception {

switch (n.intValue()) {

case 0:

break;

case 1:

throw new Exception("Test Only");

default:

return false;

}

return true;

}

}

使用 javadoc 編譯生成的文檔相關部分如下圖：

可以看到，上例中 @param b excrescent parameter 一句是多余的，因為參數只是一個 n，并沒有一個 b但是 javadoc 編譯時并沒有檢查。因此，寫文檔注釋時一定要正確匹配參數表與方法中正式參數表的項目。如果方法參數表中的參數是 a，文檔中卻給出對參數 x 的解釋，或者再多出一個參數 i，就會讓人摸不著頭腦了。@exceptin 也是一樣。

上例程序中并沒有拋出一個 NullPointerException，但是文檔注釋中為什么要寫上這么一句呢，難道又是為了演示？這不是為了演示描述多余的異常也能通過編譯，而是為了說明寫異常說明時應考運行時 (RunTime) 異常的可能性。上例程序中，如果參數 n 是給的一個空值 (null)，那么程序會在運行的時候拋出一個 NullPointerException，因此，在文檔注釋中添加了對 NullPointerException 的說明。

上例中的 @return 語句有兩個，但是根據規則，同一個方法中，只有第一個 @return 有效，其余的會被 javadoc 忽略。所以生成的文檔中沒有出現第二個 @return 的描述。

講到這里，該怎么寫文檔注釋你應該已經清楚了，下面就開始講解 javadoc 的常用命令。

**四. javadoc 命令**

運行 javadoc -help 可以看到 javadoc 的用法，這里列舉常用參數如下：

用法：

javadoc [options] [packagenames] [sourcefiles]

選項：

-public 僅顯示 public 類和成員

-protected 顯示 protected/public 類和成員 (缺省)

-package 顯示 package/protected/public 類和成員

-private 顯示所有類和成員

-d ＜directory＞ 輸出文件的目標目錄

-version 包含 @version 段

-author 包含 @author 段

-splitindex 將索引分為每個字母對應一個文件

-windowtitle ＜text＞ 文檔的瀏覽器窗口標題

javadoc 編譯文檔時可以給定包列表，也可以給出源程序文件列表。例如在 CLASSPATH 下有兩個包若干類如下：

fancy.Editor

fancy.Test

fancy.editor.ECommand

fancy.editor.EDocument

fancy.editor.EView

這里有兩個包 (fancy 和 fancy.editor) 和 5 個類。那么編譯時 (Windows 環境) 可以使用如下 javadoc 命令：

javadoc fancyTest.java fancyEditor.java fancyeditorECommand.java fancyeditorEDocument.java fancyeditorEView.java

這是給出 java 源文件作為編譯參數的方法，注意命令中指出的是文件路徑，應該根據實際情況改變。也可以是給出包名作為編譯參數，如：

javadoc fancy fancy.editor

用瀏覽器打開生成文檔的 index.html 文件即可發現兩種方式編譯結果的不同，如下圖：

用第二條命令生成的文檔被框架分成了三部分：包列表、類列表和類說明。在包列表中選擇了某個包之后，類列表中就會列出該包中的所有類；在類列表中選擇了某個類之后，類說明部分就會顯示出該類的詳細文檔。而用第一條命令生成的文檔只有兩部分，類列表和類說明，沒有包列表。這就是兩種方式生成文檔的最大區別了。

兩種方式編譯還有一點不同，

下面再來細說選項。

-public、-protected、-package、-private 四個選項，只需要任選其一即可。它們指定的顯示類成員的程度。它們顯示的成員多少是一個包含的關系，如下表：

-private (顯示所有類和成員)

-package (顯示 package/protected/public 類和成員)

-protected (顯示 protected/public 類和成員)

-public (僅顯示 public 類和成員)

-d 選項允許你定義輸出目錄。如果不用 -d 定義輸出目錄，生成的文檔文件會放在當前目錄下。-d 選項的用法是

-d 目錄名

目錄名為必填項，也就是說，如果你使用了 -d 參數，就一定要為它指定一個目錄。這個目錄必須已經存在了，如果還不存在，請在運行 javadoc 之前創建該目錄。

-version 和 -author 用于控制生成文檔時是否生成 @version 和 @author 指定的內容。不加這兩個參數的情況下，生成的文檔中不包含版本和作者信息。

-splitindex 選項將索引分為每個字母對應一個文件。默認情況下，索引文件只有一個，且該文件中包含所有索引內容。當然生成文檔內容不多的時候，這樣做非常合適，但是，如果文檔內容非常多的時候，這個索引文件將包含非常多的內容，顯得過于龐大。使用 -splitindex 會把索引文件按各索引項的第一個字母進行分類，每個字母對應一個文件。這樣，就減輕了一個索引文件的負擔。

-windowtitle 選項為文檔指定一個標題，該標題會顯示在窗口的標題欄上。如果不指定該標題，而默認的文檔標題為“生成的文檔（無標題）”。該選項的用法是：

-windowtitle 標題

標題是一串沒有包含空格的文本，因為空格符是用于分隔各參數的，所以不能包含空格。同 -d 類似，如果指定了 -windowtitle 選項，則必須指定標題文本。

虛擬器規範中的7個記憶體區域分別是三個執行緒私有的和四個執行緒共享的記憶體區,執行緒私有的記憶體區域與執行緒具有相同的生命周期，它們分別是：**程序計數器**、**虛擬機堆疊暫存器**和**本地方法堆疊暫存器**，四個共享區是所有執行緒共享的，在JVM啟動時就會分配,分別是：**方法區、 常量池、直接記憶體區和堆積暫存器**（即我們通常所說的JVM的記憶體分為堆積暫存器和堆疊暫存器中的堆積暫存器，後者就是前面的執行緒堆疊暫存器）。接下來我們逐一了解這幾個記憶體區域。

## 程序計數器（Program counter register）

java的多執行緒是通過JVM切換時間片執行的，因此每個執行緒在某個時刻可能在執行也可能被掛起，那麼當執行緒掛起之後，JVM再次調度它時怎麼知道該執行緒要執行那條字節碼指令呢？這就需要一個與該執行緒相關的記憶體區域記錄該執行緒下一條指令，而指令計數器就是實現這種功能的記憶體區域。有多少執行緒在編譯時是不確定的，因此該區域也沒有辦法在編譯時分配，只能在創建執行緒時分配，所以說該區域是執行緒私有的，該區域只是指令的計數，占用的空間非常少，所以虛擬機規範中沒有為該區域規定OutofMemoryError。

## 虛擬機堆疊暫存器(VM stack)

先讓我看以下一段程式碼：Java程式碼

class Test{

public static void main(String args) {

Thread th = new Thread;

th.start;

}

}

在執行以上程式碼時，JVM將分配一塊堆疊暫存器空間給執行緒th，用於保存方法內的局部變量，方法的入口和出口等，這些局部變量包括基本類型和物件引用類型，可能有人會問，java的物件引用不是分配在堆積暫存器上嗎？有這樣疑惑的人，可能是沒有理解java中引用和物件之間的區別，當我們寫出以下程式碼時：Java程式碼

public Object test{

Object obj = new Object;

return obj;

}

其中的Object obj就是我們所說的引用類型，這樣的聲明本身是要占用4個字節，而這4個字節在這裡就是在堆疊暫存器空間裡分配的，準確的說是在執行緒堆疊暫存器中為test方法分配的堆疊暫存器框架中分配的，當方法退出時，將會隨堆疊暫存器框架的彈出而自動銷毀，而new Object則是在堆積暫存器中分配的，由GC在適當的時間收回其占用的空間。每個堆疊暫存器空間的預設大小為0.5M，在1.7裡調整為1M，每調用一次方法就會壓入一個堆疊暫存器框架，如果壓入的堆疊暫存器框架深度過大，即方法調用層次過深,就會拋出StackOverFlow(SOF)，SOF最常見的場景就是遞歸中，當遞歸沒辦法退出時，就會拋出此異常，Hotspot提供了參數設置改區域的大小，使用-Xss：xxK，就可以修改預設大小。

## 本地方法堆疊暫存器(Native method stack)

顧名思義，該區域主要是給調用本地方法的執行緒分配的，該區域和執行緒堆疊暫存器的最大區別就是，在該執行緒的申請的記憶體不受GC管理，需要調用者自己管理，JDK中的Math類的大部分方法都是本地方法，一個值得注意的問題是,在執行本地方法時，並不是執行字節碼，所以之前所說的指令計數器是沒法記錄下一條字節碼指令的，當執行本地方法時，指令計數器置為undefined。

接下來是四個執行緒共享區。

## 方法區

這塊區域是用來存放JVM裝載的class的類信息，包括：類的方法、靜態變量、類型信息(接口/父類)，我們使用反射技術時，所需的信息就是從這裡獲取的。

## 常量池

當我們編寫如下的程式碼時：Java程式碼

class Test1{

private final int size=50;

}

這個程序中size因為用final修飾，不能再修改它的值，所以就成為常量，而這常量將會存放在常量區，這些常量在編譯時就知道占用空間的大小，但並不是說明該區域編譯就固定了，執行期也可以修改常量池的大小，典型的場景是在使用String時，你可以調用String的 intern，JVM會判斷當前所創建的String物件是否在常量池中，若有，則從常量區取，否則把該字符放入常量池並返回,這時就會修改常量池的大小，比如JDK中java.io.ObjectStreamField的一段程式碼：Java程式碼

ObjectStreamField(Field field, boolean unshared, booleanshowType){

this.field = field;

this.unshared = unshared;

name = field.getName;

Class ftype = field.getType;

type = (showType || ftype.isPrimitive) ? ftype : Object.class;

signature = ObjectStreamClass.getClassSignature(ftype).intern;

}

這段程式碼將獲取的類的簽名放入常量池。HotSpot中並沒有單獨為該區域分配，而是合併到方法區中。

## 直接記憶體區

直接記憶體區並不是JVM可管理的記憶體區。在JDK1.4中提供的NIO中，實現了高效的R/W操作，這種高效的R/W操作就是通過管道機制實現的，而管道機制實際上使用了本地記憶體，這樣就避免了從本地源文件複製JVM記憶體，再從JVM複製到目標文件的過程，直接從源文件複製到目標文件，JVM通過DirectByteBuffer操作直接記憶體。

## 堆積暫存器

主角總是最後出場，堆積暫存器絕對是JVM中的一等公民，絕對的主角，我們通常所說的GC主要就是在這塊區域中進行的，所有的java物件都在這裡分配，這也是JVM中最大的記憶體區域，被所有執行緒共享，成千上萬的物件在這裡創建，也在這裡被銷毀。

java記憶體分配到這就算是一個完結了，接下來我們將討論java記憶體的回收機制， 記憶體回收主要包含以下幾個方面理解：

第一，局部變量占用記憶體的回收，所謂局部變量，就是指在方法內創建的變量，其中變量又分為基本類型和引用類型。如下程式碼：Java程式碼

public void test{

int x=1;

char y='a';

long z=10L;

}

變量x y z即為局部變量，占用的空間將在test所在的執行緒堆疊暫存器中分配，test執行完了後會自動從堆疊暫存器中彈出，釋放其占用的記憶體，再來看一段程式碼：Java程式碼

public void test2{

Date d = new Date;

System.out.println("Now is "+d);

}

我們都知道上述程式碼會創建兩個物件，一個是Date d另一個是new Date。Date d叫做聲明了一個date類型的引用，引用就是一種類型，和int x一樣，它表明了這種類型要占用多少空間，在java中引用類型和int類型一樣占用4字節的空間，如果只聲明引用而不賦值，這4個字節將指向JVM中地址為0的空間，表示未初始化，對它的任何操作都會引發空指針異常。 如果進行賦值如d = new Date那麼這個d就保存了new Date這個物件的地址，通過之前的記憶體分配策略，我知道new Date是在jvm的heap中分配的，其占用的空間的回收我們將在後面著重分析，這裡我們要知道的是這個Date d所占用的空間是在test2所在的執行緒堆疊暫存器分配的，方法執行完後同樣會被彈出堆疊暫存器，釋放其占用的空間。

第二.非局部變量的記憶體回收,在上面的程式碼中new Date就和C++裡的new創建的物件一樣，是在heap中分配，其占用的空間不會隨著方法的結束而自動釋放需要一定的機制去刪除，在C++中必須由程式設計師在適當時候delete掉，在java中這部分記憶體是由GC自動回收的，但是要進行記憶體回收必須解決兩問題：那些物件需要回收、怎麼回收。判定那些物件需要回收，我們熟知的有以下方法： 一，引用計數法，這應是絕大數的的java 程式設計師聽說的方法了，也是很多書上甚至很多老師講的方法，該方法是這樣描述的，為每個物件維護一個引用計數器，當有引用時就加1，引用解除時就減1，那些長時間引用為0的物件就判定為回收物件，理論上這樣的判定是最準確的，判定的效率也高，但是卻有一個致命的缺陷，請看以下程式碼：Java程式碼

package tmp;

import java.util.ArrayList;

import java.util.List;

public class Test {

private byte buffer;

private List ls;

public Test {

this.buffer = new byte[4 \* 1024 \* 1024];

this.ls = new ArrayList;

}

private List getList {

return ls;

}

public static void main(String args) {

Test t1 = new Test;

Test t2 = new Test;

t1.getList.add(t2);

t2.getList.add(t1);

t1 = t2 = null;

Test t3 = new Test;

System.out.println(t3);

}

}

我們用以下參數執行：-Xmx10M -Xms10M M 將jvm的大小設置為10M，不允許擴展，按引用計數法，t1和t2相互引用，他們的引用計數都不可能為0，那麼他們將永遠不會回收，在我們的環境中JVM共10M，t1 t2占用8m，那麼剩下的2M，是不足以創建t3的，理論上應該拋出OOM。但是，程序正常執行了，(我測試後出現異常:java.lang.OutOfMemoryError: Java heap space,而且是創建t2的時候出現的，註銷了List後執行正常，JVM也GC了兩次,應該是成員變量也會占用一定的空間)

這說明JVM應該是回收了t1和t2的我們加上-XX:+PrintGCDetails執行，將列印GC的回收日記:

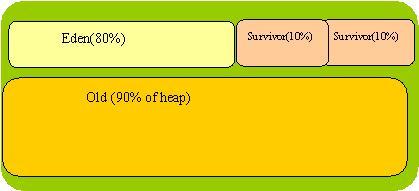
[GC [DefNew: 252K->64K(960K), 0.0030166 secs][Tenured: 8265K->137K(9216K), 0.0109869 secs] 8444K->137K(10176K), [Perm : 2051K->2051K(12288K)], 0.0140892 secs] [Times: user=0.01 sys=0.00, real=0.02 secs]

[GC [DefNew: 252K->64K(960K), 0.0030166 secs][Tenured: 8265K->137K(9216K), 0.0109869 secs] 8444K->137K(10176K), [Perm : 2051K->2051K(12288K)], 0.0140892 secs] [Times: user=0.01 sys=0.00, real=0.02 secs] com.mail.czp.Test@2ce908 Heap def new generation total 960K, used 27K [0x029e0000, 0x02ae0000, 0x02ae0000) eden space 896K, 3% used [0x029e0000, 0x029e6c40, 0x02ac0000) from space 64K, 0% used [0x02ad0000, 0x02ad0000, 0x02ae0000) to space 64K, 0% used [0x02ac0000, 0x02ac0000, 0x02ad0000) tenured generation total 9216K, used 4233K [0x02ae0000, 0x033e0000, 0x033e0000) the space 9216K, 45% used [0x02ae0000, 0x02f02500, 0x02f02600, 0x033e0000) compacting perm gen total 12288K, used 2077K [0x033e0000, 0x03fe0000, 0x073e0000) the space 12288K, 16% used [0x033e0000, 0x035e74d8, 0x035e7600, 0x03fe0000) No shared spaces configured.

從列印的日誌我們可以看出，GC照常回收了t1 t2,這就從側面證明jvm不是採用這種策略判定物件是否可以回收的。

二，根搜索算法，這是當前的大部分虛擬機採用的判定策略，GC執行緒執行時，它會以一些特定的引用作為起點稱為GCRoot，從這些起點開始搜索，把所用與這些起點相關聯的物件標記，形成幾條鏈路，掃描完時，那些沒有與任何鏈路想連接的物件就會判定為可回收物件。具體那些引用作為起點呢，一種是類級別的引用：靜態變量引用、常量引用，另一種是方法內的引用，如之前的test方法中的Date d對new Date的引用，在我們的測試程式碼中，在創建t3時，jvm發現當前的空間不足以創建物件，會出發一次GC，雖然t1和t2相互引用，但是執行t1=t2=null後，他們不和上面的3個根引用中的任何一個相連接，所以GC會判定他們是可回收物件，並在隨後將其回收，從而為t3的創建創造空間，當進行回收後發現空間還是不夠時，就會拋出OOM。

接下來我們就該討論GC 是怎麼回收的了，目前版本的Hotspot虛擬機採用分代回收算法，它把heap分為新生代和老年代兩塊區域,如下圖:



預設的配置中老年代占90% 新生代占10%，其中新生代又被分為一個eden區和兩個survivor區，每次使用eden和其中的一個survivor區，一般物件都在eden和其中的一個survivor區分配，但是那些占用空間較大的物件，就會直接在老年代分配，比如我們在進行文件操作時設置的緩衝區，如byte buffer = new byte[1024\*1024],這樣的物件如果在新生代分配將會導致新生代的記憶體不足而頻繁的gc，GC執行時首先會進行會在新生代進行，會把那些標記還在引用的物件複製到另一塊survivor空間中，然後把整個eden區和另一個survivor區裡所有的物件進行清除，但也並不是立即清除，如果這些物件重寫了finalize方法，那麼GC會把這些物件先複製到一個隊列裡，以一個低級別的執行緒去觸發finalize方法，然後回收該物件，而那些沒有覆寫finalize方法的物件，將會直接被回收。在複製存活物件到另一個survivor空間的過程中可能會出現空間不足的情況，在這種情況下GC回直接把這些存活物件複製到老年代中，如果老年代的空間也不夠時，將會觸發一次Full GC,Full gc會回收老年代中那些沒有和任何GC Root相連的物件，如果Full GC後發現記憶體還是不足，將會出現OutofMemoryError。

Hotspot虛擬機下java物件記憶體的分配和回收到此就算完結了。