除了 **方法区** 概念外，其他的内存结构（概念模型），似乎都存在抽象的（概念）、具体的（内存）两种形态；

StringTable是

```
package sun.misc;
import java.io.PrintStream;

public class Version {


    private static final String launcher_name =
        "java";
    // ...
}
```

RednaxelaFX(作者) ▸ **一叶知秋乎**

HotSpot VM的StringTable的本体在native memory里。它持有String对象的引用而不是String对象的本体。被引用的String还是在Java heap里。一直到JDK6，这些被intern的String在permgen里，JDK7开始改为放在普通Java heap里。

jdk<=6时，hotspot vm（听话宝宝） **永久带**完全包含**方法区**，所以**运行时常量池**也在**永久带**中；
字符串本体：
{
    堆中；（具体的）
    永久带中；（**Copy**的）
}
jdk>=7时，hotspot vm
字符串的本体：
{
    堆中；（具体的）

}

正如**永久带**是**虚拟机厂商**用来实现**方法区**《**JVM规范**》一样；
StringTable是**虚拟机厂商**用来实现**运行时常量池**《**JVM规范**》的方式，顺手还造了一个**字符串常量池的**（《**JVM规范**》**中不存**）**在的概念**；

方法区，运行时常量池 都只是一个概念而已；
永久带、StringTable 都是 不在虚拟机规范中没被提到的；

StringTable是**HotSpot VM**里用来实现字符串驻留功能的**全局数据结构**。如果用Java语法来说，这个**StringTable**其实就是个**HashSet**<String>:
{
StringTable存放于**native memory**中，默认大小为65535；
它并**不保存**驻留String对象**本身**，而是存储这些被驻留的String对象的引用。
VM层面触发的**字符串驻留**:
{
①例如把Class文件里的CONSTANT_String类型常量转换为**运行时对象**;（**运行时常量池**）
②以及Java代码主动触发的字符串驻留（java.lang.String.**intern**()）；
}
①②以上两种请求都由**StringTable**来处理；
}

在**驻留**的过程中，**StringTable::lookup()** 函数是**必经之路**，是用来**探测**（probe）看某个字符串是否已经驻留在StringTable里了。所以在这里下断点的话，某个字符串第一次经过这个地方的时候看调用者是谁就可以看出是什么地方在触发某个字符串的驻留。

在 **StringTable::lookup()** 函数入口下断点，并且配置它为一个条件断点，仅当传入的字符串内容为"java";时才停下来；在碰到上述断点时，查看调用栈（backtrace）。