**[提高 Java 代码性能的各种技巧](http://www.codeceo.com/article/java-coding-skills.html" \o "提高 Java 代码性能的各种技巧 - 码农网)**

2015-05-06    分类：[JAVA开发](http://www.codeceo.com/article/category/develop/java)、[编程开发](http://www.codeceo.com/article/category/develop)、[首页精华](http://www.codeceo.com/article/category/pick)[暂无人评论](http://www.codeceo.com/article/java-coding-skills.html#comments)     来源：[字节技术](http://www.4byte.cn/learning/84930/java-xing-neng-you-hua-shou-ce-ti-gao-java-dai-ma-xing-neng-de-ge-zhong-ji-qiao.html)

分享到：更多[4](http://www.codeceo.com/article/java-coding-skills.html)

**欢迎分享开发问题到**[**码农社区**](http://bbs.codeceo.com/)**！将问题帖子私信**[**@程序员俱乐部**](http://weibo.com/u/2832482174)**，将有近60000人为您解答！**

**Java 6,7,8 中的 String.intern – 字符串池**

这篇文章将要讨论 Java 6 中是如何实现 String.intern 方法的，以及这个方法在 Java 7 以及 Java 8 中做了哪些调整。

**字符串池**

字符串池（有名字符串标准化）是通过使用唯一的共享 String 对象来使用相同的值不同的地址表示字符串的过程。你可以使用自己定义的 Map<String, String> （根据需要使用 weak 引用或者 soft 引用）并使用 map 中的值作为标准值来实现这个目标，或者你也可以使用 JDK 提供的 String.intern()。

很多标准禁止在 Java 6 中使用 String.intern() 因为如果频繁使用池会市区控制，有很大的几率触发 OutOfMemoryException。Oracle Java 7 对字符串池做了很多改进，你可以通过以下地址进行了解 <http://bugs.sun.com/view_bug.do?bug_id=6962931>以及 <http://bugs.sun.com/view_bug.do?bug_id=6962930>

**Java 6 中的 String.intern()**

在美好的过去所有共享的 String 对象都存储在 PermGen 中 — 堆中固定大小的部分主要用于存储加载的类对象和字符串池。除了明确的共享字符串，PermGen 字符串池还包含所有程序中使用过的字符串（这里要注意是使用过的字符串，如果类或者方法从未加载或者被条用，在其中定义的任何常量都不会被加载）

Java 6 中字符串池的最大问题是它的位置 — PermGen。PermGen 的大小是固定的并且在运行时是无法扩展的。你可以使用 -XX:MaxPermSize=N 配置来调整它的大小。据我了解，对于不同的平台默认的 PermGen 大小在 32M 到 96M 之间。你可以扩展它的大小，不过大小使用都是固定的。这个限制需要你在使用 String.intern 时需要非常小心 — 你最好不要使用这个方法 intern 任何无法控制的用户输入。这是为什么在 JAVA6 中大部分使用手动管理 Map 来实现字符串池

**Java 7 中的 String.intern()**

Java 7 中 Oracle 的工程师对字符串池的逻辑做了很大的改变 — 字符串池的位置被调整到 heap 中了。这意味着你再也不会被固定的内存空间限制了。所有的字符串都保存在堆（heap）中同其他普通对象一样，这使得你在调优应用时仅需要调整堆大小。这 个改动使得我们有足够的理由让我们重新考虑在 Java 7 中使用 String.intern()。

**字符串池中的数据会被垃圾收集**

没错，在 JVM 字符串池中的所有字符串会被垃圾收集，如果这些值在应用中没有任何引用。这是用于所有版本的 Java，这意味着如果 interned 的字符串在作用域外并且没有任何引用 — 它将会从 JVM 的字符串池中被垃圾收集掉。

因为被重新定位到堆中以及会被垃圾收集，JVM 的字符串池看上去是存放字符串的合适位置，是吗？理论上是 — 违背使用的字符串会从池中收集掉，当外部输入一个字符传且池中存在时可以节省内存。看起来是一个完美的节省内存的策略？在你回答这个之前，可以肯定的是你 需要知道字符串池是如何实现的。

**在 Java 6，7，8 中 JVM 字符串池的实现**

字符串池是使用一个拥有固定容量的 HashMap 每个元素包含具有相同 hash 值的字符串列表。一些实现的细节可以从 Java bug 报告中获得 <http://bugs.sun.com/view_bug.do?bug_id=6962930>

默认的池大小是 1009 (出现在上面提及的 bug 报告的源码中，在 Java7u40 中增加了)。在 JAVA 6 早期版本中是一个常量，在随后的 java6u30 至 java6u41 中调整为可配置的。而在java 7中一开始就是可以配置的（至少在java7u02中是可以配置的）。你需要指定参数 -XX:StringTableSize=N,  N 是字符串池 Map 的大小。确保它是为性能调优而预先准备的大小。

在 Java 6 中这个参数没有太多帮助，因为你仍任被限制在固定的 PermGen 内存大小中。后续的讨论将直接忽略 Java 6

**Java 7 （直至 Java7u40）**

在 Java7 中，换句话说，你被限制在一个更大的堆内存中。这意味着你可以预先设置好 String 池的大小（这个值取决于你的应用程序需求）。通常说来，一旦程序开始内存消耗，内存都是成百兆的增长，在这种情况下，给一个拥有 100 万字符串对象的字符串池分配 8-16M 的内存看起来是比较适合的（不要使用1,000,000 作为 -XX:StringTaleSize 的值 – 它不是质数；使用 1,000,003代替）

你可能期待关于 String 在 Map 中的分配 — 可以阅读我之前关于 HashCode 方法调优的经验。

你必须设置一个更大的 -XX:StringTalbeSize 值(相比较默认的 1009 ),如果你希望更多的使用 String.intern() — 否则这个方法将很快递减到 0 （池大小）。

我没有注意到在 intern 小于 100 字符的字符串时的依赖情况（我认为在一个包含 50 个重复字符的字符串与现实数据并不相似，因此 100 个字符看上去是一个很好的测试限制）

下面是默认池大小的应用程序日志：第一列是已经 intern 的字符串数量，第二列 intern 10,000 个字符串所有的时间（秒）

0; time = 0.0 sec

50000; time = 0.03 sec

100000; time = 0.073 sec

150000; time = 0.13 sec

200000; time = 0.196 sec

250000; time = 0.279 sec

300000; time = 0.376 sec

350000; time = 0.471 sec

400000; time = 0.574 sec

450000; time = 0.666 sec

500000; time = 0.755 sec

550000; time = 0.854 sec

600000; time = 0.916 sec

650000; time = 1.006 sec

700000; time = 1.095 sec

750000; time = 1.273 sec

800000; time = 1.248 sec

850000; time = 1.446 sec

900000; time = 1.585 sec

950000; time = 1.635 sec

1000000; time = 1.913 sec

测试是在 Core i5-3317U@1.7Ghz CPU 设备上进行的。你可以看到，它成线性增长，并且在 JVM 字符串池包含一百万个字符串时，我仍然可以近似每秒 intern 5000 个字符串，这对于在内存中处理大量数据的应用程序来说太慢了。

现在，调整 -XX:StringTableSize=100003 参数来重新运行测试：

50000; time = 0.017 sec

100000; time = 0.009 sec

150000; time = 0.01 sec

200000; time = 0.009 sec

250000; time = 0.007 sec

300000; time = 0.008 sec

350000; time = 0.009 sec

400000; time = 0.009 sec

450000; time = 0.01 sec

500000; time = 0.013 sec

550000; time = 0.011 sec

600000; time = 0.012 sec

650000; time = 0.015 sec

700000; time = 0.015 sec

750000; time = 0.01 sec

800000; time = 0.01 sec

850000; time = 0.011 sec

900000; time = 0.011 sec

950000; time = 0.012 sec

1000000; time = 0.012 sec

可以看到，这时插入字符串的时间近似于常量（在 Map 的字符串列表中平均字符串个数不超过 10 个），下面是相同设置的结果，不过这次我们将向池中插入 1000 万个字符串（这意味着 Map 中的字符串列表平均包含 100 个字符串）

2000000; time = 0.024 sec

3000000; time = 0.028 sec

4000000; time = 0.053 sec

5000000; time = 0.051 sec

6000000; time = 0.034 sec

7000000; time = 0.041 sec

8000000; time = 0.089 sec

9000000; time = 0.111 sec

10000000; time = 0.123 sec

现在让我们将吃的大小增加到 100 万（精确的说是 1,000,003）

1000000; time = 0.005 sec

2000000; time = 0.005 sec

3000000; time = 0.005 sec

4000000; time = 0.004 sec

5000000; time = 0.004 sec

6000000; time = 0.009 sec

7000000; time = 0.01 sec

8000000; time = 0.009 sec

9000000; time = 0.009 sec

10000000; time = 0.009 sec

如你所看到的,时间非常平均，并且与 “0 到 100万” 的表没有太大差别。甚至在池大小足够大的情况下，我的笔记本也能每秒添加1,000,000个字符对象。

**我们还需要手工管理字符串池吗？**

现在我们需要对比 JVM 字符串池和 WeakHashMap<String, WeakReference<String>> 它可以用来模拟 JVM 字符串池。下面的方法用来替换 String.intern：

private static final WeakHashMap<String, WeakReference<String>> s\_manualCache =

new WeakHashMap<String, WeakReference<String>>( 100000 );

private static String manualIntern( final String str )

{

final WeakReference<String> cached = s\_manualCache.get( str );

if ( cached != null )

{

final String value = cached.get();

if ( value != null )

return value;

}

s\_manualCache.put( str, new WeakReference<String>( str ) );

return str;

}

下面针对手工池的相同测试：

0; manual time = 0.001 sec

50000; manual time = 0.03 sec

100000; manual time = 0.034 sec

150000; manual time = 0.008 sec

200000; manual time = 0.019 sec

250000; manual time = 0.011 sec

300000; manual time = 0.011 sec

350000; manual time = 0.008 sec

400000; manual time = 0.027 sec

450000; manual time = 0.008 sec

500000; manual time = 0.009 sec

550000; manual time = 0.008 sec

600000; manual time = 0.008 sec

650000; manual time = 0.008 sec

700000; manual time = 0.008 sec

750000; manual time = 0.011 sec

800000; manual time = 0.007 sec

850000; manual time = 0.008 sec

900000; manual time = 0.008 sec

950000; manual time = 0.008 sec

1000000; manual time = 0.008 sec

当 JVM 有足够内存时，手工编写的池提供了良好的性能。不过不幸的是，我的测试（保留 String.valueOf(0 < N < 1,000,000,000)）保留非常短的字符串，在使用 -Xmx1280M 参数时它允许我保留月为 2.5M 的这类字符串。JVM 字符串池 (size=1,000,003）从另一方面讲在 JVM 内存足够时提供了相同的性能特性，知道 JVM 字符串池包含 12.72M 的字符串并消耗掉所有内存（5倍多）。我认为，这非常值得你在你的应用中去掉所有手工字符串池。

**在 Java 7u40+ 以及 Java 8 中的 String.intern()**

Java7u40 版本扩展了字符串池的大小（这是组要的性能更新）到 60013.这个值允许你在池中包含大约 30000 个独立的字符串。通常来说，这对于需要保存的数据来说已经足够了，你可以通过 -XX:+PrintFlagsFinal JVM 参数获得这个值。

我尝试在原始发布的 Java 8 中运行相同的测试，Java 8 仍然支持 -XX:StringTableSize 参数来兼容 Java 7 特性。主要的区别在于 Java 8 中默认的池大小增加到 60013：

50000; time = 0.019 sec

100000; time = 0.009 sec

150000; time = 0.009 sec

200000; time = 0.009 sec

250000; time = 0.009 sec

300000; time = 0.009 sec

350000; time = 0.011 sec

400000; time = 0.012 sec

450000; time = 0.01 sec

500000; time = 0.013 sec

550000; time = 0.013 sec

600000; time = 0.014 sec

650000; time = 0.018 sec

700000; time = 0.015 sec

750000; time = 0.029 sec

800000; time = 0.018 sec

850000; time = 0.02 sec

900000; time = 0.017 sec

950000; time = 0.018 sec

1000000; time = 0.021 sec

**测试代码**

这篇文章的测试代码很简单，一个方法中循环创建并保留新字符串。你可以测量它保留 10000 个字符串所需要的时间。最好配合 -verbose:gc JVM 参数来运行这个测试，这样可以查看垃圾收集是何时以及如何发生的。另外最好使用 -Xmx 参数来执行堆的最大值。

这里有两个测试：testStringPoolGarbageCollection 将显示 JVM 字符串池被垃圾收集 — 检查垃圾收集日志消息。在 Java 6 的默认 PermGen 大小配置上，这个测试会失败，因此最好增加这个值，或者更新测试方法，或者使用 Java 7.

第二个测试显示内存中保留了多少字符串。在 Java 6 中执行需要两个不同的内存配置 比如： -Xmx128M 以及 -Xmx1280M （10 倍以上）。你可能发现这个值不会影响放入池中字符串的数量。另一方面，在 Java 7 中你能够在堆中填满你的字符串。

/\*\*

- Testing String.intern.

\*

- Run this class at least with -verbose:gc JVM parameter.

\*/

public class InternTest {

public static void main( String[] args ) {

testStringPoolGarbageCollection();

testLongLoop();

}

/\*\*

- Use this method to see where interned strings are stored

- and how many of them can you fit for the given heap size.

\*/

private static void testLongLoop()

{

test( 1000 \* 1000 \* 1000 );

//uncomment the following line to see the hand-written cache performance

//testManual( 1000 \* 1000 \* 1000 );

}

/\*\*

- Use this method to check that not used interned strings are garbage collected.

\*/

private static void testStringPoolGarbageCollection()

{

//first method call - use it as a reference

test( 1000 \* 1000 );

//we are going to clean the cache here.

System.gc();

//check the memory consumption and how long does it take to intern strings

//in the second method call.

test( 1000 \* 1000 );

}

private static void test( final int cnt )

{

final List<String> lst = new ArrayList<String>( 100 );

long start = System.currentTimeMillis();

for ( int i = 0; i < cnt; ++i )

{

final String str = "Very long test string, which tells you about something " +

"very-very important, definitely deserving to be interned #" + i;

//uncomment the following line to test dependency from string length

// final String str = Integer.toString( i );

lst.add( str.intern() );

if ( i % 10000 == 0 )

{

System.out.println( i + "; time = " + ( System.currentTimeMillis() - start ) / 1000.0 + " sec" );

start = System.currentTimeMillis();

}

}

System.out.println( "Total length = " + lst.size() );

}

private static final WeakHashMap<String, WeakReference<String>> s\_manualCache =

new WeakHashMap<String, WeakReference<String>>( 100000 );

private static String manualIntern( final String str )

{

final WeakReference<String> cached = s\_manualCache.get( str );

if ( cached != null )

{

final String value = cached.get();

if ( value != null )

return value;

}

s\_manualCache.put( str, new WeakReference<String>( str ) );

return str;

}

private static void testManual( final int cnt )

{

final List<String> lst = new ArrayList<String>( 100 );

long start = System.currentTimeMillis();

for ( int i = 0; i < cnt; ++i )

{

final String str = "Very long test string, which tells you about something " +

"very-very important, definitely deserving to be interned #" + i;

lst.add( manualIntern( str ) );

if ( i % 10000 == 0 )

{

System.out.println( i + "; manual time = " + ( System.currentTimeMillis() - start ) / 1000.0 + " sec" );

start = System.currentTimeMillis();

}

}

System.out.println( "Total length = " + lst.size() );

}

}

**总结**

* 由于 Java 6 中使用固定的内存大小（PermGen）因此不要使用 String.intern() 方法
* Java7 和 8 在堆内存中实现字符串池。这以为这字符串池的内存限制等于应用程序的内存限制。
* 在 Java 7 和 8 中使用 -XX:StringTableSize 来设置字符串池 Map 的大小。它是固定的，因为它使用 HashMap 实现。近似于你应用单独的字符串个数（你希望保留的）并且设置池的大小为最接近的质数并乘以 2 （减少碰撞的可能性）。它是的 String.intern 可以使用相同（固定）的时间并且在每次插入时消耗更小的内存(同样的任务，使用java WeakHashMap将消耗4-5倍的内存)。
* 在 Java 6 和 7（Java7u40以前） 中 -XX:StringTableSize 参数的值是 1009。Java7u40 以后这个值调整为 60013 （Java 8 中使用相同的值）
* 如果你不确定字符串池的用量，参考：-XX:+PrintStringTableStatistics JVM 参数，当你的应用挂掉时它告诉你字符串池的使用量信息。