**[JAVA代码效率优化](http://blog.csdn.net/zhshulin/article/details/42390299)**

分类： [JAVA](http://blog.csdn.net/u012909091/article/category/1835109)2015-01-04 13:54 250人阅读 [评论](http://blog.csdn.net/zhshulin/article/details/42390299#comments)(0) [收藏](javascript:void(0);) [举报](http://blog.csdn.net/zhshulin/article/details/42390299#report)

[java](http://www.csdn.net/tag/java)[代码效率](http://www.csdn.net/tag/%e4%bb%a3%e7%a0%81%e6%95%88%e7%8e%87)[性能优化](http://www.csdn.net/tag/%e6%80%a7%e8%83%bd%e4%bc%98%e5%8c%96)

       最近在想自己编程时是否注意过代码的效率问题，得出的答案是：没有。代码只是实现了功能，至于效率高不高没怎么关注，这应该是JAVA程序员进阶的时候需要考虑的问题，不再是单纯的实现功能，也不是完全依赖GC而不关注内存中发生了什么，而要考虑到代码的性能。下面是网上找的一篇关于JAVA代码优化的文章，觉得不错，就转载了。这里面设计到了JAVA基础和J2EE方面的优化建议，有时间会整理一下，现在先转载。

       原文地址是：http://blog.163.com/user\_zhaopeng/blog/static/16602270820122105731329/

1、 尽量指定类的final修饰符 带有final修饰符的类是不可派生的。

       如果指定一个类为final，则该类所有的方法都是final。Java编译器会寻找机会内联（inline）所有的 final方法（这和具体的编译器实现有关）。此举能够使性能平均提高50% 。  
  
  
2、 尽量重用对象。   
       **特别是String 对象的使用中，出现字符串连接情况时应用StringBuffer 代替**。由于系统不仅要花时间生成对象，以后可能还需花时间对这些对象进行垃圾回收和处理。因此，生成过多的对象将会给程序的性能带来很大的影响。  
  
  
3、 **尽量使用局部变量，**调用方法时传递的参数以及在调用中创建的临时变量都保存在栈（Stack）中，速度较快。   
       其他变量，如静态变量、实例变量等，都在堆（Heap）中创建，速度较慢。另外，依赖于具体的编译器/JVM，局部变量还可能得到进一步优化。   
  
  
4、 不要重复初始化变量   
        默认情况下，调用类的构造函数时， Java会把变量初始化成确定的值：所有的对象被设置成null，整数变量（byte、short、int、long）设置成0，float和 double变量设置成0.0，逻辑值设置成false。当一个类从另一个类派生时，这一点尤其应该注意，因为用new关键词创建一个对象时，构造函数链 中的所有构造函数都会被自动调用。  
  
  
5、 在JAVA + ORACLE 的应用系统开发中**，java中内嵌的SQL语句尽量使用大写的形式，以减轻ORACLE解析器的解析负担。**

6、 I/O操作中需要及时释放资源

       Java 编程过程中，进行数据库连接、I/O流操作时务必小心，在使用完毕后，即使关闭以释放资源。

因为对这些大对象的操作会造成系统大的开销，稍有不慎，会导致严重的后果。

7、 保证过期对象的及时回收

由于JVM的有其自身的GC机制，不需要程序开发者的过多考虑，从一定程度上减轻了开发者负担，但同时也遗漏了隐患，过分的创建对象会消耗系统的大量内 存，严重时会导致内存泄露，因此，保证过期对象的及时回收具有重要意义。

JVM回收垃圾的条件是：对象不在被引用；然而，JVM的GC并非十分的机智，即使对象满足了垃圾回收的条件也不一定会被立即回收。所以，建议我们在**对象使用完毕，应手动置成null。**  
  
8、 在使用同步机制时，应尽量使用**方法同步代替代码块同步**。  
  
  
9、 **尽量减少对变量的重复计算**  
  
10、尽量采用**lazy loading 的策略**，即在需要的时候才开始创建。  
  
  
11、慎用异常   
        异常对性能不利。抛出异常首先要创建一个新的对象。Throwable接口的构造函数调用名为fillInStackTrace()的本地 （Native）方法，fillInStackTrace()方法检查堆栈，收集调用跟踪信息。只要有异常被抛出，VM就必须调整调用堆栈，因为在处理过 程中创建了一个新的对象。 **异常只能用于错误处理，不应该用来控制程序流程。**  
  
12、不要在循环中使用： **Try { } catch() { } 应把其放置在最外层。**  
  
  
13、StringBuffer 的使用：   
StringBuffer表示了可变的、可写的字符串。   
有三个构造方法 :   
StringBuffer (); //默认分配16个字符的空间   
StringBuffer (int size); //分配size个字符的空间   
StringBuffer (String str); //分配16个字符+str.length()个字符空间   
你可以通过StringBuffer的构造函数来设定它的初始化容量，这样可以明显地提升性能。  
  
      这里提到的构造函数是StringBuffer(int length)，length参数表示当前的StringBuffer能保持的字符数量。你也可以使用ensureCapacity(int minimumcapacity)方法在StringBuffer对象创建之后设置它的容量。首先我们看看StringBuffer的缺省行为，然后再找 出一条更好的提升性能的途径。   
StringBuffer在内部维护一个字符数组，当你使用缺省的构造函数来创建StringBuffer对象的时候，因为没有设置初始化字符长 度，StringBuffer的容量被初始化为16个字符，也就是说缺省容量就是16个字符。当StringBuffer达到最大容量的时候，它会将自身 容量增加到当前的**2倍再加2，**也就是（2\*旧值+2）。如果你使用缺省值，初始化之后接着往里面追加字符，在你追加到第16个字符的时候它会将容量增加到 34（2\*16+2），当追加到34个字符的时候就会将容量增加到70（2\*34+2）。无论何事只要StringBuffer到达它的最大容量它就不得 不创建一个新的字符数组然后重新将旧字符和新字符都拷贝一遍――这也太昂贵了点。所以总是给StringBuffer设置一个合理的初始化容量值是错不了 的，这样会带来立竿见影的性能增益。StringBuffer初始化过程的调整的作用由此可见一斑。所以，使用一个合适的容量值来初始化 StringBuffer永远都是一个最佳的建议。  
  
  
14、合理的使用Java类 java.util.Vector。   
       简单地说，一个Vector就是一个java.lang.Object实例的数组。Vector与数组相似，它的元素可以通过整数形式的索引访问。但是，Vector类型的对象在创建之后，对象的大小能够根据元素的增加或者删除而扩展、缩小。请考虑下面这个向Vector加入元素的例子：   
Object bj = new Object();   
Vector v = new Vector(100000);   
for(int I=0;   
I<100000; I++) { v.add(0,obj); }  
  
       除非有绝对充足的理由要求每次都把新元素插入到Vector的前面，否则上面的代码对性能不利。在默认构造函数中，Vector的初始存储能力 是10个元素，如果新元素加入时存储能力不足，则以后存储能力每次加倍。Vector类就对象StringBuffer类一样，每次扩展存储能力时，所有 现有的元素都要复制到新的存储空间之中。下面的代码片段要比前面的例子快几个数量级：   
Object bj = new Object();   
Vector v = new Vector(100000);   
for(int I=0; I<100000; I++) { v.add(obj); }  
  
       同样的规则也适用于Vector类的remove()方法。由于Vector中各个元素之间不能含有“空隙”，删除除最后一个元素之外的任意其 他元素都导致被删除元素之后的元素向前移动。也就是说，从Vector删除最后一个元素要比删除第一个元素“开销”低好几倍。  
  
       假设要从前面的Vector删除所有元素，我们可以使用这种代码：   
for(int I=0; I<100000; I++)   
{   
v.remove(0);   
}  
  
      但是，与下面的代码相比，前面的代码要慢几个数量级：   
for(int I=0; I<100000; I++)   
{   
v.remove(v.size()-1);   
}  
  
      从Vector类型的对象v删除所有元素的最好方法是：   
v.removeAllElements();  
  
      假设Vector类型的对象v包含字符串“Hello”。考虑下面的代码，它要从这个Vector中删除“Hello”字符串：   
String s = “Hello”;   
int i = v.indexOf(s);   
if(I != -1) v.remove(s);  
  
      这些代码看起来没什么错误，但它同样对性能不利。在这段代码中，indexOf()方法对v进行顺序搜索寻找字符串 “Hello”，remove(s)方法也要进行同样的顺序搜索。改进之后的版本是：   
String s = “Hello”;   
int i = v.indexOf(s);   
if(I != -1) v.remove(i);  
  
       这个版本中我们直接在remove()方法中给出待删除元素的精确索引位置，从而避免了第二次搜索。一个更好的版本是：   
String s = “Hello”; v.remove(s);  
  
       最后，我们再来看一个有关Vector类的代码片段：   
for(int I=0; I++;I < v.length)  
  
       如果v包含100,000个元素，这个代码片段将调用v.size()方法100,000次。虽然size方法是一个简单的方法，但它仍旧需要 一次方法调用的开销，至少JVM需要为它配置以及清除堆栈环境。在这里，for循环内部的代码不会以任何方式修改Vector类型对象v的大小，因此上面 的代码最好改写成下面这种形式：   
int size = v.size(); for(int I=0; I++;I<size)  
  
      虽然这是一个简单的改动，但它仍旧赢得了性能。毕竟，每一个CPU周期都是宝贵的。  
  
  
15、**当复制大量数据时，使用System.arraycopy()命令。**  
  
  
16、**代码重构：增强代码的可读性。**  
  
17、**不用new关键词创建类的实例**       用new关键词创建类的实例时，构造函数链中的所有构造函数都会被自动调用。但如果一个对象实现了Cloneable接口，我们可以调用它的 clone()方法。clone()方法不会调用任何类构造函数。   
在使用设计模式（Design Pattern）的场合，如果用Factory模式创建对象，则改用clone()方法创建新的对象实例非常简单。例如，下面是Factory模式的一个 典型实现：   
public static Credit getNewCredit() {   
return new Credit();   
}   
      改进后的代码使用clone()方法，如下所示：   
private static Credit BaseCredit = new Credit();   
public static Credit getNewCredit() {   
return (Credit) BaseCredit.clone();   
}   
上面的思路对于数组处理同样很有用。  
  
  
18、乘法和除法**,用移位操作替代乘法操作**可以极大地提高性能。  
  
  
19、在JSP页面中关闭无用的会话。   
一个常见的误解是以为session在有客户端访问时就被创建，然而事实是直到某server端程序调用 HttpServletRequest.getSession(true)这样的语句时才被创建，注意如果JSP没有显示的使用 <%@pagesession=”false”%> 关闭session，则JSP文件在编译成Servlet时将会自动加上这样一条语句HttpSession session = HttpServletRequest.getSession(true);这也是JSP中隐含的session对象的来历。由于session会消耗内 存资源，因此，如果不打算使用session，应该在所有的JSP中关闭它。   
对于那些无需跟踪会话状态的页面，关闭自动创建的会话可以节省一些资源。使用如下page指令：<%@ page session=”false”%>  
  
  
20、JDBC与I/O   
如果应用程序需要访问一个规模很大的数据集，则应当考虑使用块提取方式。默认情况下，JDBC每次提取32行数据。举例来说，假设我们要遍历一个5000 行的记录集，JDBC必须调用数据库157次才能提取到全部数据。如果把块大小改成512，则调用数据库的次数将减少到10次。  
  
  
21、Servlet与内存使用   
许多开发者随意地把大量信息保存到用户会话之中。一些时候，保存在会话中的对象没有及时地被垃圾回收机制回收。从性能上看，典型的症状是用户感到系统周期 性地变慢，却又不能把原因归于任何一个具体的组件。如果监视JVM的堆空间，它的表现是内存占用不正常地大起大落。   
解决这类内存问题主要有二种办法。第一种办法是，在所有**作用范围为会话的Bean中实现HttpSessionBindingListener接口**。这 样，只要实现valueUnbound()方法，就可以显式地释放Bean使用的资源。  
  
另外一种办法就是尽快地把会话作废。大多数应用服务器都有设置会话作废间隔时间的选项。另外，也可以用编程的方式调用会话的 setMaxInactiveInterval()方法，该方法用来设定在作废会话之前，Servlet容器允许的客户请求的最大间隔时间，以秒计。  
  
22、使用**缓冲标记**   
一些应用服务器加入了面向JSP的缓冲标记功能。例如，BEA的WebLogic Server从6.0版本开始支持这个功能，Open Symphony工程也同样支持这个功能。**JSP缓冲标记既能够缓冲页面片断，也能够缓冲整个页面。**当JSP页面执行时，如果目标片断已经在缓冲之中，则 生成该片断的代码就不用再执行。页面级缓冲捕获对指定URL的请求，并缓冲整个结果页面。对于购物篮、目录以及门户网站的主页来说，这个功能极其有用。对 于这类应用，页面级缓冲能够保存页面执行的结果，供后继请求使用。  
  
  
23、选择合适的引用机制   
在典型的JSP应用系统中，页头、页脚部分往往被抽取出来，然后根据需要引入页头、页脚。当前，在JSP页面中引入外部资源的方法主要有两 种：include指令，以及include动作。   
include指令：例如<%@ include file=”copyright.html” %>。该指令在编译时引入指定的资源。在编译之前，带有include指令的页面和指定的资源被合并成一个文件。被引用的外部资源在编译时就确定， 比运行时才确定资源更高效。   
include动作：例如<jsp:include page=”copyright.jsp” />。该动作引入指定页面执行后生成的结果。由于它在运行时完成，因此对输出结果的控制更加灵活。但时，只有当被引用的内容频繁地改变时，或者在对 主页面的请求没有出现之前，被引用的页面无法确定时，使用include动作才合算。  
  
  
24、及时清除不再需要的会话   
为了清除不再活动的会话，许多应用服务器都有默认的会话超时时间，一般为30分钟。当应用服务器需要保存更多会话时，如果内存容量不足，操作系统会把部分 内存数据转移到磁盘，应用服务器也可能根据“最近最频繁使用”（Most Recently Used）算法把部分不活跃的会话转储到磁盘，甚至可能抛出“内存不足”异常。在大规模系统中，串行化会话的代价是很昂贵的。当会话不再需要时，应当及时 调用HttpSession.invalidate()方法清除会话。HttpSession.invalidate()方法通常可以在应用的退出页面调 用。  
  
  
25、不要将数组声明为：public static final 。  
  
  
26、HashMap的遍历效率讨论   
经常遇到对HashMap中的key和value值对的遍历操作，有如下两种方法：Map<String, String[]> paraMap = new HashMap<String, String[]>();   
…………….//第一个循环   
Set<String> appFieldDefIds = paraMap.keySet();   
for (String appFieldDefId : appFieldDefIds) {   
String[] values = paraMap.get(appFieldDefId);   
……   
}  
  
  
//第二个循环   
for(Entry<String, String[]> entry : paraMap.entrySet()){   
String appFieldDefId = entry.getKey();   
String[] values = entry.getValue();   
…….   
}  
  
  
第一种实现明显的效率不如第二种实现。   
分析如下 Set<String> appFieldDefIds = paraMap.keySet(); 是先从HashMap中取得keySet  
  
  
代码如下：   
public Set<K> keySet() {   
Set<K> ks = keySet;   
return (ks != null ? ks : (keySet = new KeySet()));   
}  
  
  
private class KeySet extends AbstractSet<K> {   
public Iterator<K> iterator() {   
return newKeyIterator();   
}   
public int size() {   
return size;   
}   
public boolean contains(Object o) {   
return containsKey(o);   
}   
public boolean remove(Object o) {   
return HashMap.this.removeEntryForKey(o) != null;   
}   
public void clear() {   
HashMap.this.clear();   
}   
}   
其实就是返回一个私有类KeySet, 它是从AbstractSet继承而来，实现了Set接口。  
  
  
再来看看for/in循环的语法   
for(declaration : expression)   
statement  
  
  
在执行阶段被翻译成如下各式   
for(Iterator<E> #i = (expression).iterator(); #i.hashNext();){   
declaration = #i.next();   
statement   
}  
  
  
因此在第一个for语句for (String appFieldDefId : appFieldDefIds) 中调用了HashMap.keySet().iterator()  
  
  
而这个方法调用了newKeyIterator()  
  
  
Iterator<K> newKeyIterator() {   
return new KeyIterator();   
}   
private class KeyIterator extends HashIterator<K> {   
public K next() {   
return nextEntry().getKey();   
}   
}  
  
  
所以在for中还是调用了  
  
  
在第二个循环for(Entry<String, String[]> entry : paraMap.entrySet())中使用的Iterator是如下的一个内部类  
  
  
private class EntryIterator extends HashIterator<Map.Entry<K,V>> {   
public Map.Entry<K,V> next() {   
return nextEntry();   
}   
}  
  
  
此时第一个循环得到key，第二个循环得到HashMap的Entry效率就是从循环里面体现出来的第二个循环此致可以直接取key和value值 而第一个循环还是得再利用HashMap的get(Object key)来取value值现在看看HashMap的get(Object key)方法   
public V get(Object key) {   
Object k = maskNull(key);   
int hash = hash(k);   
int i = indexFor(hash, table.length); //Entry[] table   
Entry<K,V> e = table;   
while (true) {   
if (e == null)   
return null;   
if (e.hash == hash && eq(k, e.key))   
return e.value;   
e = e.next;   
}   
}   
其实就是再次利用Hash值取出相应的Entry做比较得到结果，所以使用第一中循环相当于两次进入HashMap的Entry  
  
  
中而第二个循环取得Entry的值之后直接取key和value，效率比第一个循环高。其实按照Map的概念来看也应该是用第二个循环好一点，它本 来就是key和value的值对，将key和value分开操作在这里不是个好选择。  
  
  
27、array(数组) 和 ArryList的使用   
array（[]）：最高效；但是其容量固定且无法动态改变；   
ArrayList：容量可动态增长；但牺牲效率；   
**基于效率和类型检验，应尽可能使用array，无法确定数组大小时才使用ArrayList！**   
ArrayList是Array的复杂版本   
ArrayList内部封装了一个Object类型的数组，从一般的意义来说，它和数组没有本质的差别，甚至于ArrayList的许多方法，如 Index、IndexOf、Contains、Sort等都是在内部数组的基础上直接调用Array的对应方法。   
ArrayList存入对象时，抛弃类型信息，所有对象屏蔽为Object，编译时不检查类型，但是运行时会报错。   
注：jdk5中加入了对泛型的支持，已经可以在使用ArrayList时进行类型检查。   
从这一点上看来，ArrayList与数组的区别主要就是由于动态增容的效率问题了  
  
 **28、尽量使用HashMap 和ArrayList ,除非必要，否则不推荐使用HashTable和Vector ，后者由于使用同步机制，而导致了性能的开销。**