如何优化代码运行速度，Java性能调优技巧

作者：有故事的人 来源：范文网 www.wtabcd.cn/fanwen/

本文原地址：https://www.wtabcd.cn/fanwen/zuowen/a1c1d60b6404972947b19dd43ce8c3e5.html

可供程序利用的资源（内存、CPU时间、网络带宽等）是有限的，优化的目的就是让程序用尽可能少的资源完成预定的任务。优化通常包含两方面的内容：减小代码的体积，提高代码的运行效率。本文讨论的主要是如何提高代码的效率。

在**Java**
程序中，性能问题的大部分原因并不在于Java语言，而是在于程序本身。养成好的代码编写习惯非常重要，比如正确地、巧妙地运用java
.lang.String类和java.util.Vector类，它能够显著地提高程序的性能。下面我们就来具体地分析一下这方面的问题。

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。在Java核心API中，有许多应用final的例子，例如java.lang.String。为String类指定final防止了人们覆盖length()方法。另外，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。请参见《尽可能使用堆栈变量》。

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、 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、 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并工程造价简历非十分的机智，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

例如：for(int i = 0;i < list.size; i ++) {

…

}

应替换为：

for(int i = 0,int len = list.size();i < len; i ++) {

…

}

10、尽量采用lazy loading 的策略，即在需要的时候才开始创建。

例如： String str = “ aaa ”;

if(i == 1) {

list.add(str);

}

应替换为：

if(i == 1) {

String str = “ aaa ”;

list.add(str);

}

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 obj = 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 obj = 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。

例如：

```
public class ShopCart {
 private List carts ;
 …
 public void add (Object item) {
 if(carts == null) {
 carts = new ArrayList();
}
crts.add(item);
}
public void remove(Object item) {
 if(carts. contains(item)) {
 carts.remove(item);
}
}
public List getCarts() {
 //返回只读列表
 return Collections.unmodifiableList(carts);
}
//不推荐这种方式
//this.getCarts().add(item);
 }
```

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

考虑下面的代码：

```
for (val = 0; val < 100000; val +=5) {
alterX = val * 8; myResult = val * 2;
}
```

用移位操作替代乘法操作可以极大地提高性能。下面是修改后的代码：

```
for (val = 0; val < 100000; val += 5) {
alterX = val << 3; myResult = val << 1;
}
```

修改后的代码不再做乘以8的操作，而是改用等价的左移3位操作，每左移1位相当于乘以2。相应地，右移1位操作相当于除以2。值得一提的是，虽然移位操作速度快，但可能使代码比较难于理解，所以最好加上一些注释。

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religion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 Abstra浴霸什么牌子好ctSet<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_r)

statement

在执行阶段被翻译成如下各式

for(Iterator<E> #i = (expression_r).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();

}

}
```

此时第一个粗心大意的人循环得到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增加记忆力的食物题了

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。

29、StringBuffer 和StringBuilder的区别：

java.lang.StringBuffer线程安全的可变字符序列。一个类似于 String
的字符串缓冲区，但不能修改。StringBuilder。与该类相比，通常应该优先使用 java.lang.StringBuilder类，因为它支持所有相同的操作，但由于它不执行同步，所以速度更快。为了获得更好的性能，在构造 StirngBuffer 或 StirngBuilder
时应尽可能指定它的容量。当然，如果你操作的字符串长度不超过 16 个字符就不用了。
相同情况下使用 StirngBuilder 相比使用 StringBuffer 仅能获得 10%-15% 左右的性能提升，但却要冒多线程不安全的风险。而在现实的模块化编程中，负责某一模块的程序员不一定能清晰地判断该模块是否会放入多线程的环境中运行，因此：除非你能确定你的系统的瓶颈是在 StringBuffer
上，并且确定你的模块不会运行在多线程模式下，否则还是用 StringBuffer 吧。

文章生成doc功能，由范文网开发