Java性能优化的50个细节（珍藏版）

Java后端开发 王磊的博客 4天前

> 来源：http://t.cn/EMze6kc

在JAVA程序中，性能问题的大部分原因并不在于JAVA语言，而是程序本身。养成良好的编码习惯非常重要，能够显著地提升程序性能。

1. 尽量在合适的场合使用单例

使用单例可以减轻加载的负担，缩短加载的时间，提高加载的效率，但并不是所有地方都适用于单例，简单来说，单例主要适用于以下三个方面：

第一，控制资源的使用，通过线程同步来控制资源的并发访问；
第二，控制实例的产生，以达到节约资源的目的；
第三，控制数据共享，在不建立直接关联的条件下，让多个不相关的进程或线程之间实现通信。

2. 尽量避免随意使用静态变量

当某个对象被定义为static变量所引用，那么GC通常是不会回收这个对象所占有的内存，如

```
public class A{
    private static B b = new B();
}
```

此时静态变量b的生命周期与A类同步，如果A类不会卸载，那么b对象会常驻内存，直到程序终止。

3. 尽量避免过多过常地创建Java对象

尽量避免在经常调用的方法，循环中new对象，由于系统不仅要花费时间来创建对象，而且还要花时间对这些对象进行垃圾回收和处理，在我们可以控制的范围内，最大限度地重用对象，最好能用基本的数据类型或数组来替代对象。

4. 尽量使用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防止了使用者覆盖length()方法。另外，如果一个类是final的，则该类所有方法都是final的。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%。

如：让访问实例内变量的getter/setter方法变成”final：

简单的getter/setter方法应该被置成final，这会告诉编译器，这个方法不会被重载，所以，可以变成”inlined”,例子：

```
class MAF {
    public void setSize (int size) {
        _size = size;
    }
    private int _size;
}

更正
class DAF_fixed {
    final public void setSize (int size) {
        _size = size;
    }
    private int _size;
}
```

5. 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。

6. 尽量处理好包装类型和基本类型两者的使用场所

虽然包装类型和基本类型在使用过程中是可以相互转换，但它们两者所产生的内存区域是完全不同的，基本类型数据产生和处理都在栈中处理，包装类型是对象，是在堆中产生实例。在集合类对象，有对象方面需要的处理适用包装类型，其他的处理提倡使用基本类型。

7. 慎用synchronized，尽量减小synchronize的方法

都知道，实现同步是要很大的系统开销作为代价的，甚至可能造成死锁，所以尽量避免无谓的同步控制。synchronize方法被调用时，直接会把当前对象锁了，在方法执行完之前其他线程无法调用当前对象的其他方法。所以，synchronize的方法尽量减小，并且应尽量使用方法同步代替代码块同步。

9. 尽量不要使用finalize方法

实际上，将资源清理放在finalize方法中完成是非常不好的选择，由于GC的工作量很大，尤其是回收Young代内存时，大都会引起应用程序暂停，所以再选择使用finalize方法进行资源清理，会导致GC负担更大，程序运行效率更差。

10. 尽量使用基本数据类型代替对象

```
String str = "hello";
```

上面这种方式会创建一个“hello”字符串，而且JVM的字符缓存池还会缓存这个字符串；

```
String str = new String("hello");
```

此时程序除创建字符串外，str所引用的String对象底层还包含一个char[]数组，这个char[]数组依次存放了h,e,l,l,o

11. 多线程在未发生线程安全前提下应尽量使用HashMap、ArrayList

HashTable、Vector等使用了同步机制，降低了性能。

12. 尽量合理的创建HashMap

当你要创建一个比较大的hashMap时，充分利用这个构造函数

```
public HashMap(int initialCapacity, float loadFactor);
```

避免HashMap多次进行了hash重构,扩容是一件很耗费性能的事，在默认中initialCapacity只有16，而loadFactor是 0.75，需要多大的容量，你最好能准确的估计你所需要的最佳大小，同样的Hashtable，Vectors也是一样的道理。

13. 尽量减少对变量的重复计算

如：

```
for(int i=0;i<list.size();i++)
```

应该改为：

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

并且在循环中应该避免使用复杂的表达式，在循环中，循环条件会被反复计算，如果不使用复杂表达式，而使循环条件值不变的话，程序将会运行的更快。

14. 尽量避免不必要的创建

如：

```
A a = new A();
```

```
if(i==1){
    list.add(a);
}
```

应该改为：

```
if(i==1){
A a = new A();
    list.add(a);
}
```

15. 尽量在finally块中释放资源

程序中使用到的资源应当被释放，以避免资源泄漏，这最好在finally块中去做。不管程序执行的结果如何，finally块总是会执行的，以确保资源的正确关闭。

16. 尽量使用移位来代替'a/b'的操作

"/"是一个代价很高的操作，使用移位的操作将会更快和更有效

如：

```
int num = a / 4;
int num = a / 8;
```

应该改为：

```
int num = a >> 2;
int num = a >> 3;
```

但注意的是使用移位应添加注释，因为移位操作不直观，比较难理解。

17.尽量使用移位来代替'a*b'的操作

同样的，对于'*'操作，使用移位的操作将会更快和更有效

如：

```
int num = a * 4;
int num = a * 8;
```

应该改为：

```
int num = a << 2;
int num = a << 3;
```

18. 尽量确定StringBuffer的容量

StringBuffer 的构造器会创建一个默认大小（通常是16）的字符数组。在使用中，如果超出这个大小，就会重新分配内存，创建一个更大的数组，并将原先的数组复制过来，再丢弃旧的数组。在大多数情况下，你可以在创建 StringBuffer的时候指定大小，这样就避免了在容量不够的时候自动增长，以提高性能。

如：

```
StringBuffer buffer = new StringBuffer(1000);
```

19. 尽量早释放无用对象的引用

大部分时，方法局部引用变量所引用的对象会随着方法结束而变成垃圾，因此，大部分时候程序无需将局部，引用变量显式设为null。

例如：

Java代码

```
Public void test(){
    Object obj = new Object();
    ……
   Obj=null;
}
```

上面这个就没必要了，随着方法test()的执行完成，程序中obj引用变量的作用域就结束了。但是如果是改成下面：

Java代码

```
Public void test(){
    Object obj = new Object();
    ……
    Obj=null;
    //执行耗时，耗内存操作；或调用耗时，耗内存的方法
    ……
}
```

这时候就有必要将obj赋值为null，可以尽早的释放对Object对象的引用。

20. 尽量避免使用二维数组

二维数据占用的内存空间比一维数组多得多，大概10倍以上。

21. 尽量避免使用split

除非是必须的，否则应该避免使用split，split由于支持正则表达式，所以效率比较低，如果是频繁的几十，几百万的调用将会耗费大量资源，如果确实需要频繁的调用split，可以考虑使用apache的StringUtils.split(string,char)，频繁split的可以缓存结果。

22. ArrayList & LinkedList

一个是线性表，一个是链表，一句话，随机查询尽量使用ArrayList，ArrayList优于LinkedList，LinkedList还要移动指针，添加删除的操作LinkedList优于ArrayList，ArrayList还要移动数据，不过这是理论性分析，事实未必如此，重要的是理解好2者得数据结构，对症下药。

23. 尽量使用System.arraycopy ()代替通过来循环复制数组

System.arraycopy() 要比通过循环来复制数组快的多。

24. 尽量缓存经常使用的对象

尽可能将经常使用的对象进行缓存，可以使用数组，或HashMap的容器来进行缓存，但这种方式可能导致系统占用过多的缓存，性能下降，推荐可以使用一些第三方的开源工具，如EhCache，Oscache进行缓存，他们基本都实现了FIFO/FLU等缓存算法。

25. 尽量避免非常大的内存分配

有时候问题不是由当时的堆状态造成的，而是因为分配失败造成的。分配的内存块都必须是连续的，而随着堆越来越满，找到较大的连续块越来越困难。

26. 慎用异常

当创建一个异常时，需要收集一个栈跟踪(stack track)，这个栈跟踪用于描述异常是在何处创建的。构建这些栈跟踪时需要为运行时栈做一份快照，正是这一部分开销很大。当需要创建一个 Exception 时，JVM 不得不说：先别动，我想就您现在的样子存一份快照，所以暂时停止入栈和出栈操作。栈跟踪不只包含运行时栈中的一两个元素，而是包含这个栈中的每一个元素。

如果您创建一个 Exception ，就得付出代价，好在捕获异常开销不大，因此可以使用 try-catch 将核心内容包起来。从技术上讲，你甚至可以随意地抛出异常，而不用花费很大的代价。招致性能损失的并不是 throw 操作——尽管在没有预先创建异常的情况下就抛出异常是有点不寻常。真正要花代价的是创建异常，幸运的是，好的编程习惯已教会我们，不应该不管三七二十一就抛出异常。异常是为异常的情况而设计的，使用时也应该牢记这一原则。

27. 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使用StringBuffer代替，由于系统不仅要花时

间生成对象，以后可能还需要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。

28. 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关键字创建一个对象时，构造函数链中的所有构造函数都会被自动调用。

这里有个注意，给成员变量设置初始值但需要调用其他方法的时候，最好放在一个方法。比如initXXX()中，因为直接调用某方法赋值可能会因为类尚未初始化而抛空指针异常，如：public int state = this.getState()。

29. 在java+Oracle的应用系统开发中，java中内嵌的SQL语言应尽量使用大写形式，以减少Oracle解析器的解析负担。

30. 在java编程过程中，进行数据库连接，I/O流操作，在使用完毕后，及时关闭以释放资源。因为对这些大对象的操作会造成系统大的开销。

31. 过分的创建对象会消耗系统的大量内存，严重时，会导致内存泄漏，因此，保证过期的对象的及时回收具有重要意义。JVM的GC并非十分智能，因此建议在对象使用完毕后，手动设置成null。

32. 在使用同步机制时，应尽量使用方法同步代替代码块同步。

33. 不要在循环中使用Try/Catch语句，应把Try/Catch放在循环最外层

Error是获取系统错误的类，或者说是虚拟机错误的类。不是所有的错误Exception都能获取到的，虚拟机报错Exception就获取不到，必须用Error获取。

34. 通过StringBuffer的构造函数来设定它的初始化容量，可以明显提升性能

StringBuffer的默认容量为16，当StringBuffer的容量达到最大容量时，它会将自身容量增加到当前的2倍+2，也就是2*n+2。无论何时，只要StringBuffer到达它的最大容量，它就不得不创建一个新的对象数组，然后复制旧的对象数组，这会浪费很多时间。所以给StringBuffer设置一个合理的初始化容量值，是很有必要的！

35. 合理使用java.util.Vector

Vector与StringBuffer类似，每次扩展容量时，所有现有元素都要赋值到新的存储空间中。Vector

的默认存储能力为10个元素，扩容加倍。

vector.add(index,obj) 这个方法可以将元素obj插入到index位置，但index以及之后的元素依次都要向下移动一个位置（将其索引加 1）。 除非必要，否则对性能不利。同样规则适用于remove(int index)方法，移除此向量中指定位置的元素。将所有后续元素左移（将其索引减 1）。返回此向量中移除的元素。所以删除vector最后一个元素要比删除第1个元素开销低很多。删除所有元素最好用removeAllElements()方法。

如果要删除vector里的一个元素可以使用 vector.remove(obj)；而不必自己检索元素位置，再删除，如int index = indexOf（obj）;vector.remove(index)。

38. 不用new关键字创建对象的实例

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。

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：

```
private static Credit BaseCredit = new Credit();
public static Credit getNewCredit()
{
    return (Credit)BaseCredit.clone();
}
```

39. 不要将数组声明为：public static final

40. HaspMap的遍历：

```
Map<String, String[]> paraMap = new HashMap<String, String[]>();
for( Entry<String, String[]> entry : paraMap.entrySet() )
{
   String appFieldDefId = entry.getKey();
   String[] values = entry.getValue();
}
```

利用散列值取出相应的Entry做比较得到结果，取得entry的值之后直接取key和value。

41. array(数组)和ArrayList的使用

array 数组效率最高，但容量固定，无法动态改变，ArrayList容量可以动态增长，但牺牲了效率。

42. 单线程应尽量使用 HashMap, ArrayList,除非必要，否则不推荐使用HashTable,Vector，它们使用了同步机制，而降低了性能。

43. StringBuffer,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StringBuffer或StringBuilder时应尽量指定她的容量。当然如果不超过16个字符时就不用了。 相同情况下，使用StringBuilder比使用StringBuffer仅能获得10%~15%的性能提升，但却要冒多线程不安全的风险。综合考虑还是建议使用StringBuffer。

44. 尽量使用基本数据类型代替对象。

45. 使用具体类比使用接口效率高，但结构弹性降低了，但现代IDE都可以解决这个问题。

46. 考虑使用静态方法，如果你没有必要去访问对象的外部，那么就使你的方法成为静态方法。它会被更快地调用，因为它不需要一个虚拟函数导向表。这同时也是一个很好的实践，因为它告诉你如何区分方法的性质，调用这个方法不会改变对象的状态。

47. 应尽可能避免使用内在的GET,SET方法。

48.避免枚举，浮点数的使用。

以下举几个实用优化的例子

一、避免在循环条件中使用复杂表达式

在不做编译优化的情况下，在循环中，循环条件会被反复计算，如果不使用复杂表达式，而使循环条件值不变的话，程序将会运行的更快。例子：

```
import java.util.Vector;
class CEL {
    void method (Vector vector) {
        for (int i = 0; i < vector.size (); i++) // Violation
            ; // ...
    }
}
```

更正：

```
class CEL_fixed {
    void method (Vector vector) {
        int size = vector.size ()
        for (int i = 0; i < size; i++)
            ; // ...
    }
}
```

二、为'Vectors' 和 'Hashtables'定义初始大小

JVM为Vector扩充大小的时候需要重新创建一个更大的数组，将原原先数组中的内容复制过来，最后，原先的数组再被回收。可见Vector容量的扩大是一个颇费时间的事。

通常，默认的10个元素大小是不够的。你最好能准确的估计你所需要的最佳大小。例子：

```
import java.util.Vector;
public class DIC {
    public void addObjects (Object[] o) {
        // if length > 10, Vector needs to expand
        for (int i = 0; i< o.length;i++) {
            v.add(o); // capacity before it can add more elements.
        }
    }
    public Vector v = new Vector(); // no initialCapacity.
}
```

更正：

自己设定初始大小。

```
public Vector v = new Vector(20);
public Hashtable hash = new Hashtable(10);
```

三、在finally块中关闭Stream

程序中使用到的资源应当被释放，以避免资源泄漏。这最好在finally块中去做。不管程序执行的结果如何，finally块总是会执行的，以确保资源的正确关闭。

四、使用'System.arraycopy ()'代替通过来循环复制数组

例子：

```
public class IRB
{
    void method () {
        int[] array1 = new int [100];
        for (int i = 0; i < array1.length; i++) {
```

```
            array1 [i] = i;
        }
        int[] array2 = new int [100];
        for (int i = 0; i < array2.length; i++) {
            array2 [i] = array1 [i]; // Violation
        }
    }
}
```

更正：

```
public class IRB
{
    void method () {
        int[] array1 = new int [100];
        for (int i = 0; i < array1.length; i++) {
            array1 [i] = i;
        }
        int[] array2 = new int [100];
        System.arraycopy(array1, 0, array2, 0, 100);
    }
}
```

五、让访问实例内变量的getter/setter方法变成”final”

简单的getter/setter方法应该被置成final，这会告诉编译器，这个方法不会被重载，所以，可以变成”inlined”,例子：

```
class MAF {
    public void setSize (int size) {
        _size = size;
    }
    private int _size;
}
```

更正：

```
class DAF_fixed {
    final public void setSize (int size) {
        _size = size;
    }
    private int _size;
}
```

六、对于常量字符串，用'String' 代替 'StringBuffer'

常量字符串并不需要动态改变长度。

例子：

```
public class USC {
    String method () {
        StringBuffer s = new StringBuffer ("Hello");
        String t = s + "World!";
        return t;
    }
}
```

更正：把StringBuffer换成String，如果确定这个String不会再变的话，这将会减少运行开销提高性能。

七、在字符串相加的时候，使用 ' ' 代替 " "，如果该字符串只有一个字符的话

例子：

```
public class STR {
    public void method(String s) {
        String string = s + "d" // violation.
        string = "abc" + "d" // violation.
    }
}
```

更正：

将一个字符的字符串替换成' '

```
public class STR {
    public void method(String s) {
        String string = s + 'd'
        string = "abc" + 'd'
    }
}
```

以上仅是Java方面编程时的性能优化，性能优化大部分都是在时间、效率、代码结构层次等方面的权衡，各有利弊，不要把上面内容当成教条，或许有些对我们实际工作适用，有些不适用，还望根据实际工作场景进行取舍，活学活用，变通为宜。