技术大牛：如何提升Java效率的简单方法

你是否正打算优化hashCode()方法？是否想要绕开正则表达式？下面我们来介绍10种简单的Java性能优化方法。

**1、使用StringBuilder**

StingBuilder 应该是在我们的Java代码中默认使用的，应该避免使用 + 操作符。或许你会对 StringBuilder 的语法糖（syntax sugar）持有不同意见，比如：

String x = "a" + args.length + "b";

将会被编译为：

0 new java.lang.StringBuilder [16] 3 dup 4 ldc <String "a"> [18] 6 invokespecial java.lang.StringBuilder(java.lang.String) [20] 9 aload\_0 [args]10 arraylength11 invokevirtual java.lang.StringBuilder.append(int) : java.lang.StringBuilder [23]14 ldc <String "b"> [27]16 invokevirtual java.lang.StringBuilder.append(java.lang.String) : java.lang.StringBuilder [29]19 invokevirtual java.lang.StringBuilder.toString() : java.lang.String [32]22 astore\_1 [x]

但究竟发生了什么？接下来是否需要用下面的部分来对 String 进行改善呢？

String x = "a" + args.length + "b";if (args.length == 1) x = x + args[0];

现在使用到了第二个 StringBuilder，而且这个 StringBuilder 不会消耗堆中额外的内存，但却给 GC 带来了压力。

StringBuilder x = new StringBuilder("a");x.append(args.length);x.append("b");if (args.length == 1); x.append(args[0]);

**2、避免使用正则表达式**

正则表达式给人的印象是快捷简便。但是在 N.O.P.E 分支中使用正则表达式将是最糟糕的决定。如果万不得已非要在计算密集型代码中使用正则表达式的话，至少要将 Pattern 缓存下来，避免反复编译Pattern。

static final Pattern HEAVY\_REGEX = Pattern.compile("(((X)\*Y)\*Z)\*");

如果仅使用到了如下这样简单的正则表达式的话：

String[] parts = ipAddress.split("//.");

这是最好还是用普通的 char[] 数组或者是基于索引的操作。比如下面这段可读性比较差的代码其实起到了相同的作用。

int length = ipAddress.length();int offset = 0;int part = 0;for (int i = 0; i < length; i++) { if (i == length - 1 || ipAddress.charAt(i + 1) == '.') { parts[part] = ipAddress.substring(offset, i + 1); part++; offset = i + 2; }}

上面的代码同时表明了过早的优化是没有意义的。虽然与 split() 方法相比较，这段代码的可维护性比较差。

**3、不要使用iterator()方法**

这条建议不适用于一般的场合，仅适用于在 N.O.P.E 分支深处的场景。尽管如此也应该有所了解。Java 5格式的循环写法非常的方便，以至于我们可以忘记内部的循环方法，比如：

for (String value : strings) { // Do something useful here}

当每次代码运行到这个循环时，如果 strings 变量是一个 Iterable 的话，代码将会自动创建一个Iterator 的实例。如果使用的是 ArrayList 的话，虚拟机会自动在堆上为对象分配3个整数类型大小的内存。

private class Itr implements Iterator<E> { int cursor; int lastRet = -1; int expectedModCount = modCount; // ...

也可以用下面等价的循环方式来替代上面的 for 循环，仅仅是在栈上“浪费”了区区一个整形，相当划算。

int size = strings.size();for (int i = 0; i < size; i++) { String value : strings.get(i); // Do something useful here}

如果循环中字符串的值是不怎么变化，也可用数组来实现循环。

for (String value : stringArray) { // Do something useful here}

**4、不要调用高开销方法**

有些方法的开销很大。以 N.O.P.E 分支为例，我们没有提到叶子的相关方法，不过这个可以有。假设我们的JDBC驱动需要排除万难去计算 ResultSet.wasNull() 方法的返回值。我们自己实现的SQL框架可能像下面这样：

if (type == Integer.class) { result = (T) wasNull(rs, Integer.valueOf(rs.getInt(index)));}// And then...static final <T> T wasNull(ResultSet rs, T value)throws SQLException { return rs.wasNull() ? null : value;}

在上面的逻辑中，每次从结果集中取得 int 值时都要调用 ResultSet.wasNull() 方法，但是 getInt() 的方法定义为：

返回类型：变量值；如果SQL查询结果为NULL，则返回0。

所以一个简单有效的改善方法如下：

static final <T extends Number> T wasNull( ResultSet rs, T value)throws SQLException { return (value == null || (value.intValue() == 0 && rs.wasNull())) ? null : value;}

这是轻而易举的事情。

**5、使用原始类型和栈**

上面介绍了来自 jOOQ的例子中使用了大量的泛型，导致的结果是使用了 byte、 short、 int 和 long 的包装类。但至少泛型在Java 10或者Valhalla项目中被专门化之前，不应该成为代码的限制。因为可以通过下面的方法来进行替换：

//存储在堆上Integer i = 817598;

……如果这样写的话：

// 存储在栈上int i = 817598;

在使用数组时情况可能会变得更加糟糕：

//在堆上生成了三个对象Integer[] i = { 1337, 424242 };

……如果这样写的话：

// 仅在堆上生成了一个对象int[] i = { 1337, 424242 };

**6、避免递归**

现在，类似Scala这样的函数式编程语言都鼓励使用递归。因为递归通常意味着能分解到单独个体优化的尾递归（tail-recursing）。如果你使用的编程语言能够支持那是再好不过。不过即使如此，也要注意对算法的细微调整将会使尾递归变为普通递归。

希望编译器能自动探测到这一点，否则本来我们将为只需使用几个本地变量就能搞定的事情而白白浪费大量的堆栈框架（stack frames）。

**7、使用entrySet()**

当我们想遍历一个用键值对形式保存的 Map 时，必须要为下面的代码找到一个很好的理由：

for (K key : map.keySet()) { V value : map.get(key);}

更不用说下面的写法：

for (Entry<K, V> entry : map.entrySet()) { K key = entry.getKey(); V value = entry.getValue();}

在我们使用 N.O.P.E. 分支应该慎用map。因为很多看似时间复杂度为 O(1) 的访问操作其实是由一系列的操作组成的。而且访问本身也不是免费的。至少，如果不得不使用map的话，那么要用 entrySet() 方法去迭代！这样的话，我们要访问的就仅仅是Map.Entry的实例。

**8、使用EnumSet或EnumMap**

在某些情况下，比如在使用配置map时，我们可能会预先知道保存在map中键值。如果这个键值非常小，我们就应该考虑使用 EnumSet 或 EnumMap，而并非使用我们常用的 HashSet 或 HashMap。下面的代码给出了很清楚的解释：

private transient Object[] vals;public V put(K key, V value) { // ... int index = key.ordinal(); vals[index] = maskNull(value); // ...}

上段代码的关键实现在于，我们用数组代替了哈希表。尤其是向map中插入新值时，所要做的仅仅是获得一个由编译器为每个枚举类型生成的常量序列号。如果有一个全局的map配置（例如只有一个实例），在增加访问速度的压力下，EnumMap 会获得比 HashMap 更加杰出的表现。原因在于 EnumMap 使用的堆内存比 HashMap 要少 一位（bit），而且 HashMap 要在每个键值上都要调用 hashCode() 方法和 equals() 方法。

**9、优化自定义hasCode()方法和equals()方法**

在不能使用EnumMap的情况下，至少也要优化 hashCode() 和 equals() 方法。一个好的 hashCode() 方法是很有必要的，因为它能防止对高开销 equals() 方法多余的调用。

在每个类的继承结构中，需要容易接受的简单对象。让我们看一下jOOQ的 org.jooq.Table 是如何实现的？

最简单、快速的 hashCode() 实现方法如下：

// AbstractTable一个通用Table的基础实现：@Overridepublic int hashCode() { // [#1938] 与标准的QueryParts相比，这是一个更加高效的hashCode()实现 return name.hashCode();}

name即为表名。我们甚至不需要考虑schema或者其它表属性，因为表名在数据库中通常是唯一的。并且变量 name 是一个字符串，它本身早就已经缓存了一个 hashCode() 值。

这段代码中注释十分重要，因继承自 AbstractQueryPart 的 AbstractTable 是任意抽象语法树元素的基本实现。普通抽象语法树元素并没有任何属性，所以不能对优化 hashCode() 方法实现抱有任何幻想。覆盖后的 hashCode() 方法如下：

// AbstractQueryPart一个通用抽象语法树基础实现：@Overridepublic int hashCode() { // 这是一个可工作的默认实现。 // 具体实现的子类应当覆盖此方法以提高性能。 return create().renderInlined(this).hashCode();}

换句话说，要触发整个SQL渲染工作流程（rendering workflow）来计算一个普通抽象语法树元素的hash代码。

equals() 方法则更加有趣：

// AbstractTable通用表的基础实现：@Overridepublic boolean equals(Object that) { if (this == that) { return true; } // [#2144] 在调用高开销的AbstractQueryPart.equals()方法前， // 可以及早知道对象是否不相等。 if (that instanceof AbstractTable) { if (StringUtils.equals(name, (((AbstractTable<?>) that).name))) { return super.equals(that); } return false; } return false;}

**10、考虑使用set而并非单个元素**

最后，还有一种情况可以适用于所有语言而并非仅仅同Java有关。除此以外，我们以前研究的 N.O.P.E. 分支也会对了解从 O(N3) 到 O(n log n)有所帮助。

不幸的是，很多程序员的用简单的、本地算法来考虑问题。他们习惯按部就班地解决问题。这是命令式（imperative）的“是/或”形式的函数式编程风格。这种编程风格在由纯粹命令式编程向面对象式编程向函数式编程转换时，很容易将“更大的场景（bigger picture）”模型化，但是这些风格都缺少了只有在SQL和R语言中存在的：

声明式编程。

在SQL中，我们可以在不考虑算法影响下声明要求数据库得到的效果。数据库可以根据数据类型，比如约束（constraints）、键（key）、索引（indexes）等不同来采取最佳的算法。

在理论上，我们最初在SQL和关系演算（relational calculus）后就有了基本的想法。在实践中，SQL的供应商们在过去的几十年中已经实现了基于开销的高效优化器CBOs (Cost-Based Optimisers) 。然后到了2010版，我们才终于将SQL的所有潜力全部挖掘出来。

但是我们还不需要用set方式来实现SQL。所有的语言和库都支持Sets、collections、bags、lists。使用set的主要好处是能使我们的代码变的简洁明了。

最后，祝每位学习Java的你能成为技术大牛。