**简介：** final 关键字常常被误用 － 声明类和方法时使用过度，而声明实例字段时却使用不足。本月，Java 实践者 Brian Goetz 探究了一些有关有效使用 final 的准则。

如同它的“表亲”－ C 中的 const 关键字一样，根据上下文， final 表示不同的东西。 final 关键字可应用于类、方法或字段。应用于类时，意味着该类不能再生成子类。应用于方法时，意味着该方法不能被子类覆盖。应用于字段时，意味着该字段的值在每个构造器内必须 *只能*赋值一次而且此后该值永远不变。

大多数 Java 文本都适当地描述了使用 final 关键字的用法和后果，但是很少以准则的方式提供有关何时使用 final 及使用频率的内容。根据我的经验， final 非常过度地用于类和方法（通常是因为开发人员错误地相信这会提高性能），而在其用武之地 － 声明类实例变量 － 却使用不足。

**为什么这个类是 final？**

对于开发人员来说，将类声明为 final ，却不给出为何作出这一决定的说明，这样的做法很普遍，在开放源码项目中尤其如此。一段时间之后，特别是如果原来的开发人员不再参与代码的维护，其它开发人员将总是发问“为何类 X 被声明成final ？”。通常没人知道，当有人确实知道或喜欢猜测时，答案几乎总是“因为这能使它运行得更快”。普遍的理解是：将类或方法声明成 final 会使编译器更容易地内联方法调用，但是这种理解是不正确的（或者至少说是大大地言过其实了）。

final 类和方法在编程时可能是非常大的麻烦 － 它们限制您选择重用已有的代码和扩展已有类的功能。有时有很好的理由将类声明成 final （如强制不变性），此时使用 final 的益处将大于其不便之处。性能提高几乎总是成为破坏良好的面向对象设计原则的坏理由，而当性能提高很小或者根本没有提高时，则它真正是个很差的权衡方法。

**过早优化**

出于性能的考虑，在项目的早期阶段将方法或类声明成 final 是个坏主意，这有多个原因。首先，早期阶段设计不是考虑循环计算性能优化的时候，尤其当此类决定可能约束您使用 final 进行设计。其次，通过将方法或类声明成 final 而获得的性能优势通常为零。而且，将复杂的有状态的类声明成 final 不利于面向对象的设计，并导致体积庞大且面面俱到的类，因为它们不能轻松地重构成更小更紧凑的类。

和许多有关 Java 性能的神话一样，将类或方法声明成 final 会带来更佳的性能，这一错误观念被广泛接受但极少进行检验。其论点是：将方法或类声明成 final 意味着编译器可以更加积极地内联方法调用，因为它知道在运行时这正是要调用的方法的版本。但这显然是不正确的。仅仅因为类 X 编译成 final 类 Y，并不意味着同样版本的类 Y 将在运行时被装入。因此编译器不能安全地内联这样的跨类方法调用，不管是不是 final 。只有当方法是 private 时，编译器才能自由地内联它，在这种情况下， final 关键字是多余的。

另一方面，运行时环境和 JIT 编译器拥有更多有关真正装入什么类的信息，可以比编译者作出好得多的优化决定。如果运行时环境知道没有装入继承 Y 的类，那么它可以安全地内联对 Y 方法的调用，不管 Y 是不是 final （只要它能在随后装入 Y 子类时使这种 JIT 编译的代码无效）。因此事实是，尽管 final 对于不执行任何全局相关性分析的“哑”运行时优化器可能是有用的，但它的使用实际上不支持太多的编译时优化，而且智能的 JIT 执行运行时优化时不需要它。

[**回页首**](http://www.ibm.com/developerworks/cn/java/j-jtp1029/index.html#ibm-pcon)

**似曾相识 － 重新回忆 register 关键字**

final 用于优化决定时和 C 中不赞成使用的 register 关键字非常相似。让程序员帮助优化器这一愿望促成了 register 关键字，但事实上，发现这并不是很有用。正如我们在其它方面愿意相信的那样，在作出代码优化决定方面编译器通常比人做得出色，在现在的 RISC 处理器上更是如此。事实上，大多数 C 编译器完全忽略了 register 关键字。早先的 C 编译器忽略它是因为这些编译器根本就不起优化作用；现今的编译器忽略它是因为编译器不用它就能作更好的优化决定。任何一种情况下， register 关键字都没有添加什么性能优势，和应用于 Java 类或方法的 final 关键字很相似。如果您想优化您的代码，请坚持使用那些可以大幅度提高性能的优化，比如使用有效的算法且不执行冗余的计算 － 将循环计算优化留给编译器和 JVM 去做。

[**回页首**](http://www.ibm.com/developerworks/cn/java/j-jtp1029/index.html#ibm-pcon)

**使用 final 保持不变性**

虽然性能并不是将类或方法声明为 final 的好理由，然而有时仍有充足的理由编写 final 类。最常见的是 final 保证那些旨在不发生变化的类 *保持*不变。不变类对于简化面向对象程序的设计非常有用 － 不变的对象只需要较少的防御性编码，并且不要求严格的同步。您不会在您的代码中构建这一设想：类是不变的，然后让某些人用使其可变的方式来继承它。将不变的类声明成 final 保证了这类错误不会偷偷溜进您的程序中。

final 用于类或方法的另一个原因是为了防止方法间的链接被破坏。例如，假定类 X 的某个方法的实现假设了方法 M 将以某种方式工作。将 X 或 M 声明成 final 将防止派生类以这种方式重新定义 M，从而导致 X 的工作不正常。尽管不用这些内部相关性来实现 X 可能会更好，但这不总是可行的，而且使用 final 可以防止今后这类不兼容的更改。

**如果您必须使用 final 类或方法，请记录下为什么这么做**

无论何种情况，当您确实选择了将方法或类声明成 final 时，请记录下为什么这样做的原因。否则，今后的维护人员将可能疑惑这样做是否有好的原因（因为经常没有）；而且会被您的决定所约束，但同时还不知道您这样做的动机是为了得到什么好处。在许多情况下，将类或方法声明成 final 的决定一直推迟到开发过程后期是有意义的，那时您已经拥有关于类是如何交互及可能如何被继承的更好信息了。您可能发现您根本不需要将类声明为 final ，或者您可以重构类以便将 final 应用于更小更简单的类。

[**回页首**](http://www.ibm.com/developerworks/cn/java/j-jtp1029/index.html#ibm-pcon)

**final 字段**

final 字段和 final 类或方法有很大的不同，以至于我觉得让它们共享相同的关键字是不公平的。 final 字段是只读字段，要保证它的值在构建时（或者，对于 static final 字段，是在类初始化时）只设置一次。正如较早讨论的那样，对于 final 类和方法，您将总是问自己是否真的 *需要*使用 final 。对于 final 字段，您将问自己相反的问题 － 这个字段真的 *需要*是可变的吗？您可能会很惊讶，这个答案为何常常是“不需要”。

**文档说明价值**

final 字段有几个好处。对于那些想使用或继承您的类的开发人员来说，将字段声明成 final 有重要的文档说明好处 － 这不仅帮助解释了该类是如何工作的，还获得了编译器在加强您的设计决定方面的帮助。和 final 方法不同，声明 final字段有助于优化器作出更好的优化决定，因为如果编译器知道字段的值不会更改，那么它能安全地在寄存器中高速缓存该值。 final 字段还通过让编译器强制该字段为只读来提供额外的安全级别。

极端情况下，一个类，其字段都是 final 原语或对不变对象的 final 引用，那么该类本身就变成是不变的 － 事实上这是一个非常方便的情况。即使该类不是完全不变的，使其某部分状态不变可以大大简化开发 － 您不必为了保证您正在查看final 字段的当前值或者确保没有其他人在更改对象状态的这部分而保持同步。

那么为什么 final 字段使用得如此不足呢？一个原因是因为要正确使用它们有点麻烦，对于其构造器能抛出异常的对象引用来说尤其如此。因为 final 字段在每个构造器中必须只初始化一次，如果 final 对象引用的构造器可能抛出异常，编译器可能会报错，说该字段没有被初始化。编译器一般比较智能化，足以发现在两个互斥代码分支（比如， if...else块）的每个分支中的初始化恰好只进行了一次，但是它对 try...catch 块通常不会如此“宽容”。例如，大多数 Java 编译器不会接受清单 1 中的代码：

**清单 1. final 引用字段的无效初始化**

|  |
| --- |
| public class Foo {  private final Thingie thingie;  public Foo() {  try {  thingie = new Thingie();  }  catch (ThingieConstructionException e) {  thingie = Thingie.getDefaultThingie();  }  }  } |

但是它们会接受清单 2 中的代码，它相当于：

**清单 2. final 引用字段的有效初始化**

|  |
| --- |
| public class Foo {  private final Thingie thingie;  public Foo() {  Thingie tempThingie;  try {  tempThingie = new Thingie();  }  catch (ThingieConstructionException e) {  tempThingie = Thingie.getDefaultThingie();  }  thingie = tempThingie;  }  } |

[**回页首**](http://www.ibm.com/developerworks/cn/java/j-jtp1029/index.html#ibm-pcon)

**final 字段的局限性**

final 字段仍然有一些严重的限制。尽管数组引用能被声明成 final ，但是该数组的元素却不能。这意味着暴露 public final 数组字段的或者通过它们的方法将引用返回给这些字段的类（例如，清单 3 中所示的 DangerousStates 类）都不是不可改变的。同样，尽管对象引用可以被声明成 final 字段，而它所引用的对象仍可能是可变的。如果您想要使用 final 字段创建不变的对象，您必须防止对数组或可变对象的引用“逃离”您的类。要不用重复克隆该数组做到这一点，一个简单的方法是将数组转变成 List ，例如清单 3 中所示的 SafeStates 类：

**清单 3. 暴露数组引用使类成为可变的**

|  |
| --- |
| // Not immutable -- the states array could be modified by a malicious  caller  public class DangerousStates {  private final String[] states = new String[] { "Alabama", "Alaska", ... };  public String[] getStates() {  return states;  }  }  // Immutable -- returns an unmodifiable List instead  public class SafeStates {  private final String[] states = new String[] { "Alabama", "Alaska", ... };  private final List statesAsList  = new AbstractList() {  public Object get(int n) {  return states[n];  }  public int size() {  return states.length;  }  };    public List getStates() {  return statesAsList;  }  } |

为什么不继承 final 以应用于数组和引用的对象，类似于 C 和 C++ 中 const 的使用那样呢？C++ 中 const 的语义和使用相当混淆，根据它在表达式中所出现的位置表示不同的东西。Java 架构设计师设法把我们从这种混淆中“解救”出来，但遗憾的是他们在这个过程中产生出了一些新的混淆。

[**回页首**](http://www.ibm.com/developerworks/cn/java/j-jtp1029/index.html#ibm-pcon)

**结束语**

要对类、方法和字段有效使用 final ，有一些基本的准则可以遵循。特别要注意的是，不要尝试将 final 用作性能管理工具；要提高您的程序的性能，有更好且约束更少的方法。在反映您程序的基本语义处使用 final ：用来指示这些类将是不可改变的或那些字段将是只读的。如果您选择创建 final 类或方法，请确保您清楚地记录您为何这么做 － 您的同事会感激您的。