定量语言设计

**简介：** 对于任何给定编程语言来说，都不缺乏新功能。语言设计者不仅要执行决定许多可能（而且经常不兼容）语言功能接收优先级这一困难任务，而且他们还必须考虑新语言功能与现有功能令人惊讶的交互。语言的进化通常需要在启用新编码模式的优势与破坏现有 “怪异” 代码潜在成本之间进行权衡。在这种情况下，可以使用实际数据量化 “怪异” 代码不寻常的程度，它可以为决策制定方式提供有价值的线索。

在我的第一次参加 JavaOne 大会时，我出席了一个由 Java™ 架构师 Graham Hamilton 主持的对话节目，他观察到 “ Java 社区的中的每个开发人员都有一个或两个有关 Java 语言的特性创意”。果然，我发现我也有一些。在此新的系列中，即*语言设计者的笔记本*，我将探索评估哪些新的创意可以进入 Java 语言的过程，以及在广泛使用的语言中为新功能腾出空间的挑战。在第一部分中，我将谈论实际数据如何用来告知并影响语言进化决策。

**发展现有语言**

在我们职业生涯中，我们可能都受到过设计新编程语言的诱惑 — 可能是出于无奈。我们描述摆在我们面前的具体问题解决方案的方式，以及我们表达此解决方案的工具，往往并不完全一致。我们用于描述结果代码的词汇唤起了我们的回忆：*笨拙*、*臃肿*、*刺鼻*。有时，问题归咎于我们自己的缺点 — 我们的解决方案根本就不足够精致或灵敏 — 但有时，用真正的编程语言表达时，甚至最精致的解决方案都感觉不那么精致了。

新语言的语言设计者花费了大量的时间来考虑功能；现有语言的语言设计者花费了大量的时间来考虑兼容性。

虽然显而易见的解决方案 — 设计您梦寐以求的语言 — 可能会很有趣，但是这是一场艰苦的战斗。即使您是建立在现有丰富的平台上（如 JVM） — 其提供如垃圾收集、并发控制、安全性、调试器支持以及丰富的运行时库等功能 — 大量工作仍然存在下来：如设计语言、实现编译器、实现任何不被基础平台支持的运行时功能、设计并实现核心库、开发 IDE 集成，等等。但这一切仍是容易的部分！困难的部分在后面：即让用户采用新的语言，以及处理不可避免的投诉和改进建议。然后才是真正困难的部分：面对您梦寐以求的语言其实并不完美的这样一个事实、它需要改进，而您需要在妥协改进计划和破坏用户现有代码的不兼容更改之间作出选择。您有 10 个用户时，比您有 1000 万个用户时更容易调整不兼容性的变更。新语言的语言设计者花费大量时间考虑功能；现有语言的设计者花费大量时间考虑兼容性。

[**回页首**](http://www.ibm.com/developerworks/cn/java/j-ldn1/index.html#ibm-pcon)

**简单示例：更精确的重新抛出（rethrow）分析**

Java SE 7 中的新语言功能之一（作为 Project Coin 的一部分引入（请参考 [参考资料](http://www.ibm.com/developerworks/cn/java/j-ldn1/index.html#resources)））是更精确的异常重新抛出分析。在 Java SE 6 中，语言只使用已声明（静态）类型的异常参数来计算可能从块中抛出的异常类型，这意味着不能编译清单 1 中的程序：

**清单 1. 不能在 Java SE 6 下编译的代码，因为 foo() 没有声明抛出 Throwable**

|  |
| --- |
| void doIo() throws IOException { ... }  void foo() throws IOException {  try {  doIo();  }  catch (final Throwable t) {  log("Exception in foo!", t);  throw t;  }  } |

[清单 1](http://www.ibm.com/developerworks/cn/java/j-ldn1/index.html#listing1) 中的代码要做什么显而易见。通过 catch 子句捕获的惟一内容是 IOException 和未选中的异常（RuntimeException 和Error），所有这些都可从 foo() 抛出。但是异常参数 t 被声明为 Throwable，且如果没有修改 throws 子句，则 Throwable不能从 foo() 中抛出。但这似乎有点苛刻。编译器可以非常清楚地找出哪种类型的异常 t 可以保留并证实 [清单 1](http://www.ibm.com/developerworks/cn/java/j-ldn1/index.html#listing1) 不会违反 foo() 的合同 — 而无需程序员赴汤蹈火。更精确的重新抛出功能准确地启用这种排序分析：编译器从可由块（以及由之前的 catch 子句）抛出的异常集确定异常参数可能具有的异常类型集。

事实证明这个更精确的重新抛出分析（显然是非常有用的功能）与 catch 块的现有可访问性交互（其中编译器将拒绝捕获没有从相应 try 块抛出异常的 catch 块或者已经由以前的 catch 块捕获的异常）。这可能会破坏一些现有的代码，如清单 2 中的代码：

**清单 2. 在 Java SE 6 下编译的代码，但在建议的更精确的重新抛出分析规则下却遭到拒绝**

|  |
| --- |
| class Foo extends Exception {}  class SonOfFoo extends Foo {}  class DaughterOfFoo extends Foo {}  class Test {  void test() {  try {  throw new DaughterOfFoo();  } catch (Foo exception) {  try {  throw exception; // \*\*\*  } catch (SonOfFoo anotherException) {  // Question: is this block reachable?  }  }  }  } |

在旧的分析下，[清单 2](http://www.ibm.com/developerworks/cn/java/j-ldn1/index.html#listing2) 中的程序是合法的，因为 \*\*\* 处的 throw 语句被假定抛出 Foo，因此嵌套 catch 子句可以捕获SonOfFoo。但是更精确的分析可显示在 \*\*\* 处唯一被捕获的是 DaughterOfFoo（或未选中的异常），嵌套 catch 子句无法被访问，因为知道 SonOfFoo 不能在此处被捕获，因此应视为一个错误而被拒绝。如果我们有严格地立场，“从不破坏现有代码”，那么我们将被迫拒绝这项有用的新功能。但是 [清单 2](http://www.ibm.com/developerworks/cn/java/j-ldn1/index.html#listing2) 中的代码看起来很矫造。如果只是因为我们不破坏这种看起来矫造的代码而失去这个不错的新功能，我们将感到羞愧。因此我们应该怎么做呢？

在语言演变中这是一个经典的权衡：即以明显的方式改进语言的优势与破坏一些（可能很奇怪的）程序的未知风险之间进行权衡。虽然很容易一厢情愿地相信没有一个像这样的代码，但是对一些编码支持的网站研究足以证明作出如此归纳的愚蠢。

在异常参数 t 上，更精确重新抛出分析的初始设计还需要 final 修饰符。这种限制的意图是双重的。首先，这是为了防止人为分配，就是所说的针对 t 的 SQLException（这将是一种合法的指定，但是随后将不再是重新抛出的恰当选择）。其次，这是为了减少破坏现有代码的机会，因为相对少的异常参数已经是 final；因此，即使像 [清单 2](http://www.ibm.com/developerworks/cn/java/j-ldn1/index.html#listing2) 那样的示例在自然环境下存在，只有异常参数为 final 时才打开更精确的重新抛出分析，这将大大减少破坏现有代码的机会。然而，final限制是令人烦躁的 — 开发人员常抱怨在编译器弄清楚需要了解的一切时，他们必须声明某些 final。以往这类决策都是根据直觉而定的。但是有一个更好的办法：收集一些数据。

[**回页首**](http://www.ibm.com/developerworks/cn/java/j-ldn1/index.html#ibm-pcon)

**语言库（Corpus ）分析**

如果发展广泛使用语言的挑战之一是那里有大量我们不想破坏的代码，那么一个优势就是有大量代码，我们可用于分析某些习惯用语出现的频率。通过抓取大量代码（*语言库*）并运行分析工具（通常是编译器的仪表化版本），您可以了解很多关于某些习惯用语在实现中出现的频率，以及它们是否普遍存在可以影响我们改进语言。

作为评估使用语言库制定语言改进决策效果的起点，我们采用了几个大型的且积极维护的开源代码库：JDK 库、Apache Tomcat 和 NetBeans。然后我们运行分析工具来检测任一 catch 块实例是否在旧的分析下可以编译而在新的规则下却不能编译。然而我们什么也没有发现。

这是非常令人鼓舞的，因为我们可以从主观的 “我不认为很多人会这样编码” 转到更具体的 “数百万行代码分析显示这种模式并不常发生”。虽然这并不意味着这种编码模式永远不会发生，但是它为其罕见性提供了真实的证据，而不是凭感觉。

在 JDK 代码库中，我们的分析显示：

* catch 块的数量：19,254
* 有效 final 异常参数的数量（不是 final 的异常参数，但是表现仿佛它们是）：19,197 (99.7%)
* 明确的最终异常参数的数量：16 (0.085%)
* 无效的 final、非 final 异常参数的数量：41 (0.21%)
* 编译在精确分析下失败的案例数量：0

有了这些数据，我们愿意放弃将 catch 规范为 final 以便获得更精确分析的要求 — 这使得很多开发人员都很高兴。

我们对于这种有限语言库的经验是如此令人鼓舞，即我们现在使语言库分析成为语言进化过程的关键部分。除了我在早期命名的代码库以外，现在我引入 Qualitas 语言库，这是一个有 100 多个流行开源包的策划集合（请参考 [参考资料](http://www.ibm.com/developerworks/cn/java/j-ldn1/index.html#resources)）。（在此语言库上的分析不会出现任何更精确的分析破坏现有代码的情况。）将来，为了进一步提高语言库分析的预测效率，我们还计划为封闭源代码的 Oracle 产品引入源数据库。

[**回页首**](http://www.ibm.com/developerworks/cn/java/j-ldn1/index.html#ibm-pcon)

**另一个示例：通用类型推断**

Java SE 7 中引入的另一个语言功能（还作为 Project Coin 的一部分）是 *diamond* 或者通用类型参数的类型推断。该功能旨在减少不必要的冗余 — 对于开发人员来说是很常见问题源头 — 在此示例的代码中：

|  |
| --- |
| public List<String> list = new ArrayList<String>(); |

虽然类型参数需要在任务双方进行重复，但是大多数时候重复是多余的。相反，使用类型推断，编译器可以自己处理。有了类型推断，编译器可以通过强制解决计算出变量类型，而不是使程序员明确声明它们。一些语言（如 ML）大量利用类型推断来支持通用方法，但是类型推断只发生在 Java SE 5 中的 Java 语言。虽然类型推断并不完美而且有时以令人困惑的方式失败（这就是为什么我们有时必须使用 foo.<T>bar() 语法明确声明通用方法的类型参数），但是有限的类型推断常常使我们从静输入中受益，那么，输入。

Diamond 功能使用类型推断计算出赋值右侧的类型参数，并允许将前面的示例重新编写为：

|  |
| --- |
| public List<String> list = new ArrayList<>(); |

我们应用 diamond 功能时这个具体的示例只有六个字符的缩短，但是其他示例有很显著的缩短，因为类型参数可以具有更长的名称，且一些类型具有多个类型参数（例如，Map<UserName, List<UserAttribute>>）。

虽然类型推断是一项强大的技术，但是它具有局限性，尤其是在像 Java 那样具有参数多态性（通用）和包含多态性（继承）的语言环境中。从根本上讲，类型推断是约束解决技术，往往对使用哪些约束开始具有多个合理的选择。

作为具体示例，请考虑清单 3，声明了通用类型 Box<T> 并调用了 Box 结构函数：

**清单 3. 来自 Java SE 7 的 diamond 功能的典型使用**

|  |
| --- |
| public class Box<T> {  private T value;  public Box(T value) {  this.value = value;  }  T getValue() {  return value;  }  }  ... = new Box<>(42); |

编译器应该为底部赋值中的 T 推断什么类型？肯定可以为 T 推断很多类型，我们知道：Integer、Number、Object、Comparable<?>、Serializable 以及一些看上去非常奇怪的类型（如 Object & Comparable<? extends Number> & ...）。

虽然设计了此功能，但是社区提出了两个相互竞争的推断架构：

* 使用正在被赋值的变量的类型（称为*简单架构*）。
* 使用赋值上下文，如简单架构中那样，还使用构造函数参数的类型（称为*复杂结构*）。

这两种架构的支持者都声称他们的算法 “更好”，且每一个都引用其他架构的失败为证。当然，两者工作的都很好，其中左边具有类似 Box<Integer> 的简单类型，但是在左边类似 Box<? extends Number> 时，事情开始变得棘手。在这种情况下，简单算法产生 Box<Number>（因为其只使用来自左边的类型），而复杂算法产生更精确的推断 Box<Integer>（因为其考虑到参数 42 的类型）。您可能会认为，“那很好，我喜欢复杂算法因为其给出更准确的答案。” 但是这只是一种情况。在这种情况下左边是 Box<Number>，复杂算法产生一个导致编译错误的推断：复杂算法推断 Box<Integer>，其不会分配给Box<Number>（回想一下，通用型（不同于阵列）不是协变式的，因此分配 Box<Integer> 到 Box<Number> 会产生类型错误），而简单算法产生 Box<Number>，在这种情况下它是更好的选择。更复杂的算法通常不会有完美的答案。

在 diamond 表达式出现在方法调用上下文时会有轻微的不同。请考作为 m() 方法的参数出现的 Box<>：

|  |
| --- |
| void m(Box<Integer> box){...}  ...  m(new Box<>(42)); |

因为 Java 语言类型表达式是自下而上 — 以便在执行过载解析以前方法参数的类型可用 — 简单算法在这里什么也做不了。因此必将得出参数的类型是 Box<Object>，其将导致在方法解析以后出现编译错误，因为 Box<Object> 不会分配给Box<Integer>。另一方面，复杂算法产生 Box<Integer>，因为这是使用构造函数参数可以推断的最具体类型。

因此，很容易找到简单算法产生更好答案的示例，而复杂算法产生更好的答案的另一些示例。我们应该如何选择？从历史上看，这些有关哪种情况更流行或重要，或者哪种错误更令人吃惊的决策都是基于感觉而制定的。但是由于 diamond 是旨在简化现有习惯用语的功能，所有我们可以使用通用方法调用的许多实例来指导我们。因此我们创建了专业版的编译器 — “diamond 查找器” — 其识别在没有导致编译失败的情况下何时可以删除明确的类型参数，且我们列出了两种算法的结果。

结果很有趣。在大约 200,000 个通用构造函数调用的实例上，两种算法大约 90% 的时候都可以预测到结果。这意味着两种架构都有效，且任何一个都不比另一个更有效。因此最起码我们了解到该功能是有用的且足以满足自身的需要 — 其在大多数时间的预测都是正确的，且对于开发人员来说事情变得更容易了。我们还了解到每一个算法在 90% 的时候所预测的都稍有不同 — 且彼此都不是对方的子集。因为任何一个都不比另一个具有更严格的预测，所以我们现在不能挑出一种算法并将其升级为另一种算法。但是由于两种算法都具有大概同等的效果，所以这意味着他们都是合理的选择，因此我们可以在次要考虑的基础上作出决策。最终我们选择了复杂算法，因为其更类似于语言中其他地方的类型推断行为。该决策使语言更加一致、使实现更加易于维护以及提供更多机会以便充分利用将来在类型推断中的改进。

[**回页首**](http://www.ibm.com/developerworks/cn/java/j-ldn1/index.html#ibm-pcon)

**结束语**

在设计新语言功能时，很难预测开发人员将用它们做什么、它们将如何变更编码模式或者哪些方法会使开发人员在它们的缺点上受挫。但是，在细化现有语言功能时，我们可以现有代码库使用静态分析来回答有关开发人员实际上用这些功能来做什么的问题 — 并将那些对人们实际编码产生更明显影响的功能优先。此信息对于制定语言进化决策来说是无价的。在 Java SE 7 中，我们使用语言库分析来完善几个关键的语言设计决策，且这有助于指导我们作出更好的决策。语言库分析将是今后我们工具箱中的一个重要组成部分。

鸣谢：感谢 Oracle 的 Joe Darcy 和 Maurizio Cimadamore 对本文的大力支持。

**简介：** 尽管一些建议的语言功能可以解决遇到的某个问题，但其中大部分功能的存在都有现实环境中的根源，在这些环境中，现有功能无法使程序员轻松、清晰、简洁或安全地表达他们想要的概念。尽管头脑中有一个用例，“此功能使我能够编写我希望能够编写的代码”，但语言设计师还需要评估语言功能可能带来的糟糕代码。

当从头设计语言时，我们有机会分组评估语言功能，调整它们以实现协同交互或避免消极交互。而且我们有机会通过选择语言功能来挑选我们想要的编程语言风格和构思模型。在考虑*现有* 语言的新语言功能时，我们的选择较少。我们常常无法（至少不容易）调整其他功能来适应新功能，而且某些编程语言风格已存在于该语言的方言中。在这些情况下，我们常常只能围绕它们进行设计。

尽管一些建议的功能是由行不通的思路启发得到的，但大部分功能在具体的用例中拥有它们的根源。它们的诞生常常离不开语言中目前表达特定语言风格的非常繁杂、冗长或零散的代码所带来的挫败感，以及 “如果我可以仅编写……该多好” 的想法。但从*启用这段很酷的代码* 到*良好的功能* 是一个漫长的过程。显然，一种语言功能要值得拥有，它必须使我们能够表达某些在之前无法表达的 “良好的” 新程序，但新语言功能也可以使我们能够表达一些 “糟糕” 的程序。而且即使新功能可以避免新的 “糟糕” 程序，它也可能破坏现有的固定语言、用户期望或性能模型特征。改进现有的语言需要权衡更强的表达能力的优势与更低的安全性、功能交互或用户混淆的危害。

**一个简单示例：在对象上使用 switch**

Java SE 7 中引入的一种语言功能是允许 switch 语句操作 String 类型的变量以及原语类型和枚举。扩展 switch 语句的应用范围，不仅扩展到字符串，也扩展到其他类型，这已成为多年来反复的增强请求的主题。（例如，RFE 5012262 请求不仅可在字符串上使用 switch，也可在任何对象上使用 switch，通过其 equals() 方法进行比较（参见 [参考资料](http://www.ibm.com/developerworks/cn/java/j-ldn2/index.html#resources)）。另一个类似的频繁请求是允许非常量表达式显示为 switch 语句的 case 标签。

乍看起来，switch 语句似乎就是等效的 if ... else 语句嵌套在句法上的改进。实际上，开发人员在 switch 和 if...else 语句嵌套之间的选择常常主要基于哪个在代码中更美观。但它们并不相同。switch 语句的内在限制（case 标签必须是常量值，switch 操作符仅限于行为类似于常量的类型）的存在具有性能和安全两方面的原因。常量标签的限制使分支计算成为了一种 O(1) 运算，而不是 O(*n*) 运算。在 if...else 语句嵌套中，到达 else 块需要执行所有比较，因为 if...else 语句的语义需要顺序执行。case 标签被限制为类似常量的值（原语、枚举和字符串），这确保了比较运算没有副作用（因此可以实现在其他情况下无法实现的优化）。如果我们允许将任意对象用作 case 标签，那么调用 equals() 方法可能具有意外的副作用。

如果我们从头设计语言，我们可能有更大的自由来决定程序员便捷性是否比这里的可预测性更重要，并相应地定义switch 语句的语义（和限制）。但对于 Java 语言，时机已过。将 switch 扩展到类似常量的值以外，可能破坏 Java 开发人员多年建立起来的性能模型，所以 switch 中允许任意对象的更高表达能力无法抵消成本。因为 String 类是不变的，并且是高度具体化和受控的，所以将它放入 switch 中很实用，但最好不要止步于此。

[**回页首**](http://www.ibm.com/developerworks/cn/java/j-ldn2/index.html#ibm-pcon)

**一个具有更大争议的示例**

Java SE 8 中最重要的新语言功能是 *lambda 表达式*（或*闭包*）。闭包是一种函数字面量，包含可视为一个值并在以后调用的延迟计算的表达式。而且它们具有*词法作用域*，这意味着闭包内的符号的含义应该与它们的外部含义相同（在闭包内对局部变量求模，这种求模可从词法作用域投影变量）。自 Java SE 1.1 开始，Java 语言就拥有闭包的一种简单形式，即内部类，但它们的限制和繁杂的语法阻碍了真正探索这类代码即数据机制所允许的抽象能力的 API 开发。

在语言中拥有闭包，使 API 能够表达更具协作性的（进而更丰富的）计算，允许客户端提供部分计算。Collections API 支持这种行为的一种有限形式，比如将一个 Comparator 传递到 Collections.sort()，但仅用于相对重量级的运算，比如排序。对于像 “创建一个大小大于 10 的元素列表”，我们会强制客户端手动公开该运算，如以下示例所示：

|  |
| --- |
| Collection<Element> bigOnes = new ArrayList<Element>();  for (Element e : collection)  if (e.size() > 10)  bigOnes.add(e); |

尽管此代码非常紧凑且具可读性，但 Collections API 对我们没有太大帮助，我们被迫进入了一种基本的顺序执行模式（因为 for 循环的语义是顺序的，所以这是我们迭代 Collections 元素的惟一方式）。此运算从 Collections 提取想要的元素子集 ，是一种常见的运算。如果可以将所有控制逻辑（串行或并行的）转移到一个库例程中，仅使用关于我们想要哪些元素的谓词来进行参数化。那么代码可精减为以下形式：

|  |
| --- |
| Collection<Element> bigOnes  = collection.filter(#{ Element e -> e.size() > 10 }); |

我们可以使用内部类实现此目的，但它们的使用非常繁杂，以至于有时似乎解决办法比问题更严重。内部类在开发出集合框架时就已诞生，但内部类的句法开销使支持使用它们的 Collections API 的创建令人不太满意。（这里的 lambda 表达式的语法，以及集合 API 的改进，是暂时的，而且仅能作为可使用 Java SE 8 编写哪些代码的建议。）

上一个例子中的 lambda 表达式是一种具有特别良好的行为的 lambda 表达式，是一种不从其词法作用域捕获任何值的表达式。但表达一种相对于作用域内已有的其他值的计算常常很有用，比如以下方法中对局部变量 *n* 的捕获：

|  |
| --- |
| public static<T> Collection<Element> biggerThan(Collection<Element> coll, int n) {  return coll.filter(#{ Element e -> e.size() > n });  } |

内部类（以及 lambda 表达式）的一个限制是，它们仅能从其词法作用域引用 *final* 的局部变量。Java SE 8 中的 lambda 表达式使这一限制更受欢迎，它们还允许捕获*有效的 final* 变量，即那些没有声明为 final、但在最初赋值后不会修改的变量。（如果实例上下文中存在内部类表达式，那么内部类可引用易变的*实例字段*，但这不是一回事。可以将此情况视为，内部类中对来自闭包类的字段 x 的引用实际上是 Outer.this.x 的简写，其中 Outer.this 是一个隐式声明的 final 局部变量。）在此限制的多种动机中，最大的动机是，将局部变量捕获限制到 final 字段就会允许闭包复制引用，进而保留这样一种行为：局部变量的生命周期就是声明它的代码块的生命周期。

毫无疑问，仅从词法上下文捕获不变的状态这一限制，令程序员非常不满意。他们可能不满意的是，似乎尽管 Java 语言*final* 会获得闭包，但闭包的这一方面似乎没有用武之地。

希望捕获易变的局部变量的典型代码示例可能类似于清单 1：

**清单 1. 通过闭包捕获易变的局部变量（在 Java SE 8 中无效）**

|  |
| --- |
| int sum = 0;  collection.forEach(#{ Element e -> sum += e.size() });  System.out.printf("The sum is %d%n", sum); |

这看起来是一种想要做的合理（甚至明显）的事情，这无疑也是其他一些支持闭包的语言中的一种常见语言风格。为什么我们不想在 Java 中支持此代码？

首先，它看起来与最初并不相同，它是对局部变量语义的一项重要更改。局部变量的生命周期被限制到它的声明所在的代码块的生命周期。但是，lambda 表达式被视为值，因此可存储在变量中并在将捕获的变量声明为超出范围的代码块执行之后执行。如果允许捕获易变局部变量，该平台将需要将局部变量的生命周期延长到任何捕获它的 lambda 表达式的动态生命周期。这是对程序员关于局部变量的预期的重大变更，具体来讲，缺少了任何将此变量声明为奇怪的、新的耐久局部变量的特殊声明。

当您认为 forEach() 方法可能希望从其他线程调用 lambda、从而易变函数可以并行应用到集合的不同元素时，问题会变得更糟。现在，我们在局部变量 sum 上创造了一场数据竞争，因为多个线程可能同时希望更新它。局部变量上的数据竞争将是一种新的危险，因为我们目前始终期望局部变量访问没有数据竞争。没有直观的途径来使 [清单 1](http://www.ibm.com/developerworks/cn/java/j-ldn2/index.html#listing2) 中的代码是线程安全的，这使这种语言风格成为了一种等待时机发生的事故。

在这一点上，明智的方法是规避它。在并发 Java 程序中避免数据争用比我们想象的困难得多。一个远离此危险的安全情形是局部变量不受数据竞争影响，因为它们只能从单一线程访问。通过允许 lambda 表达式捕获易变的局部变量，会使它们的行为类似于字段，而不是不可见的局部变量，进而将它们暴露在数据竞争的危险之中。在 2011 年对语言进行让并发和并行运算更加危险的更改是很愚蠢的。

这种语言风格有可能得到补救，比如，通过在可捕获的易变局部变量上定义一个修饰符（进而明确区分它们与普通局部变量），该修饰符将捕获这些变量的 lambda 的语义限制到定义该变量的线程和词法作用域内。这样一种功能有利有弊，它增加了语言保留特定编程语言风格（以及一种在本质上串行的过时语言风格）的复杂性。

**一种更好的解决方案**

此刻不增加额外的复杂性来支持此语言风格的原因在于，有一种更好的方法来获得相同结果。此语言风格是*映射（mapping）* 运算与*减（reduction）* 或*折叠（folding）* 运算相结合的一个示例，其中将一个联接运算符（比如 sum 或max）成对应用到了一个值序列。得益于联接性，这种减运算支持并行化。我们可以直接在集合上公开一个 mapReduce()方法，如下所示：

|  |
| --- |
| int sum = collection.mapReduce(0, #{ Element e -> e.size() },  #{ int left, int right -> left + right }); |

这里，第一个 lambda 表达式是映射器（将每个元素映射到它的大小），第二个 lambda 表达式是一个减法器，它获取两个大小并相加。此代码计算的结果与 [清单 1](http://www.ibm.com/developerworks/cn/java/j-ldn2/index.html#listing1) 中的示例相同，但采用并行友好的方式。（并行性不是没有代价的，库必须提供并行化，但至少在使用此方式表达语言风格时，库*可以* 并行实现该运算。不仅映射和减法支持并行化，映射和减法运算也可结合到单个并行循环中，这样效率更高。（而且这无需在客户端代码中包含易变状态即可完成。）

事实上，对于映射器和为整数求和而预定义的减法器，我们可以使用 size() 方法的方法引用，更紧凑地表达此过程：

|  |
| --- |
| int sum = collection.mapReduce(0, #Element.size, Reducers.INT\_SUM); |

一旦熟悉了以这种方式指定计算的理念，此代码看起来就像一个问题语句：将整数求和应用到集合中每个元素的 size()方法的结果上。

**不要与它抗争**

大部分开发人员可能无需太多时间即可确定易变局部变量的捕获限制有一种 “解决办法”：将局部变量替换为对一个一元素数组的最终引用，如清单 2 所示：

**清单 2. 使用对一元素数组的最终引用欺骗编译器。不要这么做！**

|  |
| --- |
| int[] sumH = new int[1];  collection.forEach(#{ Element e -> sumH[0] += e.size() });  System.out.printf("The sum is %d%n", sumH[0]); |

这段代码通过编译器，进而可能提供 “在系统上成功完成一项任务” 的短暂满足感。但它重新带来了数据竞争的可能性。这不是个好主意，而且您不应该尝试。就像去除了桌上型锯床的保护套，它将增加事故风险。但与桌上型锯床不同的是，任何受伤的手指更可能是其他人的，而不是您自己的。如果存在一种针对此情形的更安全（且可能更快）的语言风格（映射-减法），则没有借口编写这样的不安全代码，即使它 “在此情形下” 看起来是安全的。

[**回页首**](http://www.ibm.com/developerworks/cn/java/j-ldn2/index.html#ibm-pcon)

**结束语**

对于一项新语言功能，很容易仅看到它会带来的优秀代码。我们应该不停寻找更好的办事方式，但新语言功能也可能导致发生一些确实很糟糕的事情。因为引入糟糕语言功能的风险如此之高，所以语言设计师在进行关于优势是否多于劣势的成本-收益分析时需要持保守态度。如果新功能值得怀疑，我们应该谨记格言 *Primum non nocere*：首先，不要造成伤害。

**简介：** 向语言添加某个重要的新功能时，往往会出现以下情况：新功能是必需的，或者至少鼓励添加的，而附加的其他新功能则有好有坏。在这一期的 [语言设计人员笔记](http://www.ibm.com/developerworks/cn/views/java/libraryview.jsp?view_by=search&sort_by=Date&sort_order=desc&view_by=Search&search_by=%E8%AF%AD%E8%A8%80%E8%AE%BE%E8%AE%A1%E8%80%85%E7%9A%84%E7%AC%94%E8%AE%B0%E6%9C%AC&dwsearch.x=11&dwsearch.y=17)中

一些语言功能，比如 Java SE 7 中引入的新功能 “二进制整数文字量 (binary integer literals)”，均可独立运行。但是，由于需要额外添加一些功能才能正常运行或与现有功能进行交互，许多重要的语言功能遭到遗弃。这是一个潜在的问题，因为添加重要语言功能能总是具有很大风险，在此基础上添加其他功能只会带来更大的风险。

**增加了使用便利功能的压力**

一个语言功能携带另一个功能所带来的潜在风险是：新的功能增加了添加无关 “便利功能” 的压力。我们来看一下 Java SE 5 中添加的自动装箱 (autoboxing) 功能。Java 从一开始就拥有针对这些类型的原始类型（比如 int）和对象 "box" 包装类（比如 Integer），还拥有在这些类型之间转换的方法。自动装箱这一新功能是原始类型与其相应包装类之间的隐式转换，是从一开始就已添加的一个功能，在发布 Java SE 5 之前，一些用户就曾调用过它。该功能是泛型的附带品，也是集合的伴生物，这最终为添加便利功能带来了巨大的压力。在以前，您需要在原始类型与其包装器之间进行转换，但泛型集合使这一情景变得更为普遍，因为现在可以很方便地创建密钥或值都被装箱为原始类型的集合。添加自动装箱功能原本只是有点小麻烦，现在变成了大麻烦，添加自动装箱功能的压力在增加。

另一个类似的示例是 enum 与静态导入（也是 Java SE 5 中添加的新功能）之间的关系。静态导入是将要引入的另一个便利功能；有必要提一下的是，不只是 PI，Math.PI 也总是令人感到讨厌。不过，正是因为 Java SE 5 中添加了 enum 功能，才会产生证明静态导入的压力。Enum 可以轻松创建指定结构化常数，比如 Color.RED、Color.BLUE 等等，当您可以更轻松地创建结构化常数时，就可以节省更多的时间和精力。以前只有几个静态系统定义常数可用，enum 为用户提供了创建静态常数的功能，这使得每次使用时都需要限定名称这一麻烦的事情（而非只是说明是 RED 还是 BLUE）变得更加麻烦。所以，虽然静态导入是一个在添加 enum 之前就已经添加的可独立运行的功能，但是，enum 为验证添加静态导入是否合法带来了压力。

虽然添加这些便利功能不是什么问题，但是它们可能具有副作用。比如，自动装箱功能与三元条件运算符在特定情况下无法进行交互，这会导致因为根本没有用来处理对象引用的代码而抛出 NullPointerException。

[**回页首**](http://www.ibm.com/developerworks/cn/java/j-ldn3/index.html#ibm-pcon)

**LinQ：用一个功能的价格购买六个功能**

只需添加一个语言功能便可获得它附带的一大堆相关功能，.NET 3.0 中添加的核心增强工具 LinQ (Language-Integrated Query) 可能是这方面的一个最佳示例。LinQ 使开发人员能够将 “对象-值” 查询直接嵌入其代码中。不仅可以对数据库使用这些查询，还可以对数据提供程序（比如 XML 文档或内存集合）使用这些查询。将查询语言嵌入通用编程语言的想法似乎很容易实现，但是一旦深入了解后，您就会发现需要添加其他许多功能才能实现它。

下面的 C# 代码展示了一个针对集合的典型 LinQ 查询。它采用了一个 Person 对象集合，并选择和打印出了年龄小于 18 周岁的人的姓名：

|  |
| --- |
| var results =  from p in people  where p.Age < 18  select new {p.FirstName, p.LastName};  foreach (var r in results) {  Console.WriteLine(r.FirstName + " " + r.LastName);  } |

要完成此项工作必须组合使用一些查询。此项查询的结果类型究竟是什么呢？它不是 Person 对象的集合，因为该查询只询问姓名属性。相反，它是只包含姓名属性的类的集合；编译器基于查询的选定字段生成该类。为此，.NET 引进了*匿名类 (anonymous class)*；否则，开发人员必须为每一个查询的结果创建一个新的指定类类型。

LinQ 还要求支持 lambda 表达式（闭包），虽然这在之前的查询示例中并不明显。LinQ 的实现策略涉及到将查询重写至针对某个提供程序的 API 调用中。（此提供程序 API 提供了将查询用于不同数据源的方式）。编译器将查询重写为如下所示：

|  |
| --- |
| var results =  people.Where(p => p.Age < 18)  .Select(p => new {p.FirstName, p.LastName}); |

Where() 方法用了一个谓语来确定是否选择给定元素，并且生成一些可通过过滤器的元素。然后，对于每个选定元素，Select() 方法都会将该元素映像到只包含姓名属性的一个新的匿名类实例。

但是我们的工作还没结束。如果数据提供程序是一个 SQL 数据库，则必须将 WHERE 子句应用于每条记录。实现此操作的一个方法是将数据库中的所有记录都拖至应用程序中，然后测试每条记录的 Age 属性。但是此操作的效率可能很低，我们宁可采用 WHERE 子句评估相近的数据。明智的做法是将 WHERE 子句发送至数据库，但是这意味着我们必须将谓语 p.Age < 18 转换成 SQL，并将其发送给数据库。

使用 LinQ 解决此问题的解决方法是使用*表达树*，这是一种类似反射的机制，您不仅可以用它反射类的成员，还可以用它反射某个方法的代码。这充许 SQL LinQ 提供程序分析传递给 Where() 方法的闭包，并将它转换成 SQL。

将查询转换成对 API 的调用也需要添加*扩展方法*。Where() 方法是在集合对象上调用的，但它不是该集合框架的成员。相反，它是一个由 LinQ 子系统定义的静态方法，并被注入 IEnumerable（Java Iterable 的 .NET 对等物）。否则，用户无法轻松表达针对集合的 LinQ 查询。

最后，我们需要使用*隐式类型化变量*，这样就可以为变量分配查询，不必显式声明其类型。因为查询的结果是某个匿名类型的 IEnumerable，编译器可以识别这种结果的类型，但无法用 C# 表达出来（此外，对于某些查询，结果类型是可表达的，但是叙述冗长，不便写下来）。这就是 C# 支持使用 var 来声明变量的原因，这样做可以让编译器识别出变量类型，无需将它拼写出来，这样做不是鼓励程序员偷懒，而是因为有时无法写下变量类型。

我们还有很长的路要走！最初的时候，似乎目标很容易实现，只需将查询嵌入通用语言即可，结果，需要使用匿名类、隐式类型化变量、闭包、扩展方法以及表达式的反射机制。对于 LinQ 的关键目标而言，这些功能中的每个功能都是至关重要的。

作为用户，您可能会作出这样的结论：用一个功能的价格即可购买六大功能，这是一个不错的想法。但是向现有语言添加一项新的语言功能总是需要付出一定的成本。当您添加语言功能 B 来支持语言功能 A 之后，并不要求只能将功能 B 用于功能 A。功能 B 本身可能并不是很理想，或者无法与其他语言功能进行很好的交互。明显，添加功能 A 是有一个特定目标的，比如使语言更安全、表达更清楚。但要评估功能 B 成功与否的话，不是相对于功能 A 来评估功能 B，二是需要相对于涉及的新语言（包括与功能 A 相关的所有功能）进行评估。如果所涉及的语言不是您最终想要的语言，那么您可能需要重新考虑最初要添加的功能。

[**回页首**](http://www.ibm.com/developerworks/cn/java/j-ldn3/index.html#ibm-pcon)

**Java 中的 Lambda 表达式**

Java SE 8 中的核心语言增强功能是 *lambda 表达式*，或称为*闭包*。但是，正如 .NET 中的 LinQ 一样，lambda 表达式会拖入一大堆其他功能，其中包括 SAM 转换、增强的类型推断、方法引用和扩展方法，以便为用户提供 lambda 的全部效益。

因为 lambda 表达式（表达式表示函数）是 Java 中一种新值，我们需要一种方法来写下其类型。对于 Java 中的 lambda 表达式，早期的建议是要求将*函数类型*添加至类型系统，如 “function from int to int"。函数类型确实是表现 lambda 表达式类型的一种自然方法，但遗憾的是它们无法与现有语言功能 “擦除 (erasure)” 很好地进行交换。因为在底层字节码中，表现函数类型的自然方法是使用泛型，并且应该将函数签名中的原始类型进行装箱，一种类型不能让采用函数类型的多个方法过载，即使它们的参数完全不同。函数类型可能是表达 lambda 类型的一种自然方法，但不是表达已擦除函数类型的方法。

因此，Java SE 8 中的 lambda 表达式会带来另一种不同的相关功能 *SAM 转换*，而不是函数类型。SAM（单一抽象方法）类型一直是我们在 Java 语言中表达函数的一种方式，它与某个方法有关系，比如 Runnable、Comparator 或 ActionListener方法。如果我们构建 SAM 类型（大多数类型已经存在于库中）的 API，编译器可以在 lambda 表达式（类似于函数字面量）与 SAM 类型（其参数类型、返回类型和异常类型均与 lambda 表达式相符）之间进行转换。例如，下列的代码声明了一个 Comparator<String>，该表达式对字符串的长度进行比较，并使用一个 lambda 表达式来定义 Comparator:。

|  |
| --- |
| Comparator<String> c  = (String a, String b) -> a.length() — b.length(); |

因为 lambda 表达式拥有正确的参数和返回类型，编译器将证实可以把这些类型转换成 Comparator<String>，并生成完成此操作的相应代码。这个转换就叫做 *SAM 转换*。

lambda 表达式的基本原理是提供了一种将代码表达为数据的方式，从而可以将代码字面量传递至代码库中，以便在方便的时候随时调用。另一个动机是减少内部类的冗余性，目前 lambda 表达式是达到此效果的最简便的方法。当您踏上了消除冗余句法结构的道路之后，通常会顺着这条道越走越远。所以，lambda 表达式带来了另一个相关功能，即通过*目标类型化（target typing）* 实现的扩展类型推断。因为之前的 lambda 表达式将分配给 Comparator<String>，所以关于 a 和 b 的类型均为 String 的显式声明就显得有些多余，因为编译器通常会为我们指明这一点。通过使用赋值上下文的类型来推断a 和 b 的类型，我们可以将示例代码精简为：

|  |
| --- |
| Comparator<String> c  = (a, b) -> a.length() — b.length(); |

如果我们拥有一个 Person 对象集合，并且想根据 Person 的姓氏对列表进行分类，那么我们立即将代码改写如下：

|  |
| --- |
| Collections.sort(people, new Comparator<Person>() {  Public int compare(Person a, Person b) {  return a.getLastName().compareTo(b.getLastName());  }  } |

在使用 lambda 表达式时，可以使代码更加简洁：

|  |
| --- |
| Collections.sort(people, (a, b) ->  a.getLastName().compareTo(b.getLastName()); |

这在减少冗余性的路上已经跨进了一大步，但仍然不是很抽象，因为它还是会要求用户以命令方式来计算比较函数。因为在库中做了一些小小的更改，所以我们能够更好地利用 lambda 表达式来分离分类的核心部分，即分类关键字的选择。因为 String 是 Comparable，所以分类方法早就应该知道如何在提取分类关键字后执行比较操作：

|  |
| --- |
| Collections.sortBy(people, p -> p.getLastName()); |

这个代码的确好了很多，它开始读取更多像 “按姓氏对人们进行分类” 之类的问题语句。但是，由于删除了样板文件，我们意识到，用来提取分类关键字的词语（在这里是姓氏）本身有点让人晦涩难懂。上面的 lambda 表达式什么都没做，只是提取其参数（在本例中为 none），将该参数传递给现有方法 getLastName()，并使用将在其上调用该方法的对象充当第一个参数。虽然在本例中这样做看起来不是太糟糕，因为没有其他必须采用给定名称的参数（以及重复两次的名称），但是直接为方法命名看起来似乎更好一些。相关功能*方法引用* 允许我们这样做，它通过名称引用某个方法，并将它作为类似 lambda 表达式的 “函数-值” 数据对待：

|  |
| --- |
| Collections.sortBy(people, Person::getLastName); |

最后，由于已经删除了样板文件，sortBy() 方法实际上不应该是某实用类中的静态方法，而应该是集合上的实例方法，这一事实表现得比以往更为明显。但是，接口的不良属性之一是：在我们指定该属性之后，就无法在不破坏现有实现的情况下添加新方法。将要引入 lambda 表达式的最终功能是*虚拟扩展方法*，它充许我们以适当的方式向接口添加新方法，实现此操作的方法是随方法声明提供一个（可重写的）默认实现。这样我们就可以添加 lambda 友好（和潜在并行友好）方法，比如 List 的 forEach()。通过向针对 sortBy() 的 List 添加一个扩展方法，我们的示例现在如下所示：

|  |
| --- |
| people.sortBy(Person::getLastName); |

奇怪的是，我们最终版本根本没有使用 lambdas！但是它体现了向语言添加 lambda 表达式的核心目标，即捕抓部分计算并像传递数据一样传递它们的功能，这使我们能够以更丰富的方式对库的功能（比如分类）进行参数化。在这个特殊的示例中，与 lambda 表达式相比，方法引用更清楚地表达出了我们想要表达的东西，但表达的意思是相同的。

在不使用 SAM 转换、类型推断、方法引用和扩展方法的情况下，将 lambda 表达式添加至 Java，这是完全有可能的。然而，缺少这些功能也有可能最终成为痛点，甚至我们可能都没有意识到痛苦的真正来源在哪儿。

[**回页首**](http://www.ibm.com/developerworks/cn/java/j-ldn3/index.html#ibm-pcon)

**有时恰恰相反**

不向 Java 添加函数类型的理由可能是不想要（或支付不起）不太实用的额外语言功能。虽然函数类型会是表示 lambda 表达式类型的最自然的一种表达方法，并且会减少对大量名义上的类型（比如 Predicate 和 Mapper）的需要，但它与擦除功能之间的互动表现得不太令人满意。人们对此的直接响应是：通过引入另一个相关功能，即具体化，来消除糟糕的互动功能。向 Java 语言添加具体化泛型有利也有弊，但事实上，在添加 lambda 表达式的同时，添加大型的、意义深远（影响语言、编译器和库）的功能，比如具体化功能，这样做有些不太切合实际。因此，鉴于我们无法承受与函数类型相关的附带功能，所以我们只能跟函数类型说再见（至少现在是这样）。

[**回页首**](http://www.ibm.com/developerworks/cn/java/j-ldn3/index.html#ibm-pcon)

**结束语**

大多数大型的语言功能都无法完全独立运行，它们通常需要其他相关功能的支持，然后才能实现它们期望为我们提供的全部利益。在考虑添加具有其他附带功能的功能时，必须认真考虑是否真的想要这些不太实用的相关功能，因为我们可能会被这些功能绊住。如果我们无法接受某项功能的其他相关功能，则有可能必须放弃该功能。