第 12 章

Java

Java 语言源于一个运行在小型设备上的开发项目。随着网页浏览器和 applet（译注1）的普及，它获得了越来越多的应用。通过自动内存管理、虚拟机、一组绑定库和某种程度的“仅须编写一次，不同平台运行”，它日益成为一种通用的编程语言。尽管 Sun Microsystems 已经以自由软件的形式发布了其源代码，但该公司仍通过 Java 社区保留对语言的演变和库管理的一定程度上的控制。

译注1：applet 是指小型应用程序，它仅需要很小的内存资源，通常可以在操作系统间移动的一种应用程序。

278

12.1 功能或者简单性
Power or Simplicity

您说过简单性和功能是讨厌的孪生兄弟。您能解释一下吗？

James Gosling：通常系统想更强大就往往变得很复杂。比如说 Java EE，它拥有事务和持久性等非常强大的功能。但在 Java 的早期，这些是根本不存在的。

该系统确实非常简单，人们也很容易理解。如果只要求自己学习 Java 语言和基本的 API，它仍然是非常简单的。但是，当你开始使用像 Swing 和 Java EE，以及其余一些更强大的子系统时，你会感觉到眼花缭乱、应接不暇。看看 OpenGL 库，里面有好多功能真的很强大。不过，我的老天爷，它可不是那么简单的。

尤其是使用 OpenGL，你需要拥有理想化图形硬件的知识。

James：对。我记得爱因斯坦曾经说过，实际的系统应尽可能地更加简单，而不是更简单。

简单性和复杂性在系统中是一成不变的吗？Larry Wall 谈到了一种关于复杂性的水床理论，如果你在语言的某一部分降低复杂性，它就会重现在其他地方。虽然 Java 核心语言本身很简单，但复杂性是否会出现在例如库等其他地方呢？

James：这句话我喜欢用“打鼹鼠”来解释。通常人们说，“哦，我解决了这个问题”。但如果你看一下周围，你会发现他们并没有真正解决问题，只是将问题转移了。

其中一个问题在于语言是供大家使用的。如果你的语言支持专门的事务，使用该事务的人可能会感觉十分便捷，但不使用人的就会感到十分繁琐。

这要付出无谓的概念性开销。

James：噢，他们肯定要付出一些开销，这取决于具体怎样做，他们可能还要付出一些别的开销。

Java 是一个已经广泛使用了十几年的成熟平台，它还有可能被重新设计为一个简单的系统吗？

James：我不知道，我想或许有可能吧。

事实上有一大堆代码使用了这些东西。人们通常想甩掉那些复杂性，可问题是，如果你仔细调查一下周围的应用程序，你会发现他们还是倾向于大量使用复杂性。

279

你曾想用 Swing 代替 AWT，不过 AWT 至今仍然存在。

James：是的，仍有很多人在使用 AWT，这很让人惊讶。部分原因在于手机上还在用它。

但是还有部分原因在于，尽管和新闻报道相反，Java 令人难以置信地在桌面应用程序中有大量应用，主要是用于构建企业级应用程序。有成千上万的应用程序在使用 beta 库，并且它们一直运转得很好，所以几乎没有什么理由放弃它们。

答案是肯定的，原因在于，在一定程度上，如果你有一个带有合理抽象的面向对象系统，那么在世界改变时，你就可以找到更好的方式，而且能使用那种方式来做事，并且这种新方式不会和旧的发生冲突。

你使用了命名空间、抽象，还有封装的概念。

James：是的。把 JavaEE 变为 JavaEE5，我们对 JavaEE 进行了重大的革命性简化。如果你看今天的 JavaEE，仅仅是翻翻 JavaEE5 手册，它在复杂性方面的确做得不错。但是如果你翻翻旧版的 Java 手册，再尝试把这两版都搞清楚，那就会很烦人了。我们很合理地推进了 JavaEE 的发展，除非你不得不脚踏两只船。生活并非都很愉悦。

向下兼容总是很难。是否允许使用者在最新的 JVM 中运行老的 Java1.1 版本程序，这是 Sun 公司工程师们慎重讨论的问题吗？

James：很有趣的一点是，JVM 自身实际上从来没有被讨论过，因为 JVM 是很稳定的。所有的痛苦和烦恼都来自库。在核心虚拟机中，库却是非常容易管理的。它们被设计成模块化，可以自由进库、出库。其实你可以构建一个类加载器，用它来分开命名空间，从而使你可以同时拥有两个版本的 Java AWT。已经有很多类似的工具存在了。

一旦你准备着手处理虚拟机本身，你的麻烦就来了。但是虚拟机里也没有想象中那么多的问题。

嗯，有人说，在 Java 中实际上有两个编译器。一个是 Java 字节码的编译器，一个是 JIT（运行时编译），JIT 基本上对所有东西都专门进行再次编译。而所有那些令人望而生畏的优化工作都在 JIT 中进行。

James：没错。这些年我们击败了不少确实很好的 C 和 C + +编译器。当你用动态编译器时，你就会体会到两个好处。一个就是你能确切地知道在哪种芯片上运行。很多时候人们在编译一段 C 代码时，他们不得不把代码编译成能够运行在通用 x86 架构上。而你几乎找不到什么库能特别适合它们。你可以下载 Mozilla 的最新版本，而且它能在几乎所有的英特尔的 CPU 架构上良好运行。还有一个相当好的 Linux 库。它非常通用，使用 GCC 来编译，但 GCC 并不是一个很好的 C 编译器。

当 HotSpot 运行时，它确切地知道是自己在什么芯片组上运行，也确切地知道缓存如何工作，知道内存的层次结构如何工作，知道所有的管道互锁如何在 CPU 中工作，它知道该芯片应得到什么指令集扩展，而且会根据你的机器进行精确的优化。另一方面，它可以在应用程序运行时对其有一个清楚地了解。它还能获得统计数据，从而知道哪些东西是重要的。它能够进行任何 C 编译器都做不到的内联。Java 的内联功能是相当惊人的。另外你还可以加上使用现代垃圾收集器的存储管理方式等优点。使用现代垃圾收集器，存储分配会变得非常快。

您是在说移动分配指针。

James：这只是字面上的移动分配指针。在 Java 中 new 的开销约等于 C 中的 `malloc()`。有些时候移动分配指针的基准是比 `malloc()`好 10 倍。涉及大量的小对象时，`malloc()`的表现通常会相当糟糕。

说到 C，您如何设计一个系统编程语言？当一名设计师准备构建一种系统语言时，他需要考虑些什么问题呢？

James：我倾向于不过多考虑语言和功能。在我设计语言的那个年代，非常可悲的是，语言基本上都是为了解决特定的问题。运行的背景是什么？人们用它来做什么？它的领域有何不同？使用 Java 而有不同的就是网络。无处不在的网络会使你在思考的时候有些不同，因为它有很多派生的信息。甚至有可能是你祖母起居室的计算机上正在进行着的某项计算。

或者是看起来不像计算机的一种手持设备？

James：在长长的不断变化的事物清单上，你永远不希望看到蓝屏死机。你不想有复杂的安装特性。因此，Java 最终实现了真正强大的故障隔离机制。大多数人并不这样认为，但这种隔离机制确实存在。像内存指针、垃圾收集、异常处理等都与故障隔离有关。其最终目的是确保系统在有小错误的时候也可以正常运行。就比如你的汽车沿着马路行驶，而门把手松了，你的车仍然可以继续行驶。大多数 C 程序的问题在于，你似乎在做一些完全无害的事情，结果变成了一个讨厌的空指针引用，而且还很有欺骗性。它就像榴霰弹一样到处都是。

281

您无法预言什么时候会出现系统损坏，以及在哪里或何时会崩溃。

James：是的。我屡遭打击，对此我完全不可接受。在 C 出现的时候，在 SUN 的早期，性能高于一切。数组边界检查之类的行为或者问一下“这到底是什么”都是完全不可接受的。当 Java 规范出现时，就成了“数组下标检查无法关闭”。没有“无下标检查之类的事”。一方面，它在一定程度上与 C 大相径庭，因为 C 根本就没有下标检查。另一方面，它也是语言规范的内在部分。

如果您要这样考虑的话，它几乎就没有数组的概念。

James：是的。它增加了一些东西，还有有关的语法。现代编译器的法宝之一，是它们能“定理化地证明”一种方式潜在地所有下标检查。虽然你可能也这么看（你知道，禁止关闭指针检查是一件非常糟糕的事），但事实上，这并不是一件坏事。它没有任何负面的性能影响。在循环外部你也许还要做一些检查，但在循环内部，你不得不为它的出色表现而惊叹。

因为 C 语言的字符串是以空字符结尾的（null-terminated），从而无法计算出它的长度，如果仅仅修复了这个问题，我们可能 40 年前就会有更快的 C 语言。而这恰恰是我最近使用 C 时遇到的最糟糕的问题。

James：是的。我喜欢 C。我已经当了很多很多年的专业 C 程序员了。在很多程序员开始用 C 前，我就已经转到 C 上很多年了。第一个 C 编译器运行在 32K 的 RAM 上，而且对于运行在 32K 的 RAM 上的程序来说，最初的 C 编译器是令人惊奇的，但现在你已很难找到一只只有 32K RAM 的手表。

12.2 品味的问题
A Matter of Taste

无处不在的互联网连通性是如何改变编程语言的概念的呢？

James：关于网络会如何影响编程语言设计，这是一个非常大的问题。只要你连上了网络，你就不得不处理多样性，你就不得不处理交互性，你就不得不考虑故障会对其他部分造成怎样的影响，你不得不去更多地考虑系统可靠性。

尤其是，你不得不担心如何构建一个非常健壮的系统，这个系统可以在面临部分失效的时候仍然继续运转，因为人们所构建的大多数系统总是关心损坏。

282

对于软件的传统观念就是要么全部成功要么全部失败；它会正常工作或根本无法工作。它对 Java 异常机制、强类型系统、垃圾收集器及虚拟机等都非常关注。我的意思是说，网络在 Java 设计、语言、虚拟机上确实都有着深远的影响。

在您设计和编程时，什么对您影响最大？我在看您设计或开发的一个系统时，您指着某个东西并说这就是出自 Gosling 之手，您有什么方法可以做到这些？

James：好的。我希望现实就是那么简单。

我遇到过一些架构师，他们说“我有一个商标设计”，反映了他们趋向于这样对待事物。我更加趋向于没有商标。如果我有一个商标，当你跟已经看过我写的代码的那些人交谈时，我会让他们发狂，他们不得不维护我写的代码，因为我对性能的要求非常之高。

我不会使用积极内联机制，当简单的工作确实难以奏效的时候，还是倾向于使用复杂的算法。我会竭尽全力，会比大多数人使用更多的缓存操作。我会出于条件反射中抛出缓存，因为如果没有缓存，我会很紧张。

那使我想起有人问我“为什么使用数组线性搜索？快速排序（译注2）速度更快。”我的回答是：“可能最多只有 7 个元素需要排序，写一个快速排序代价太高了”。

James：我肯定不会为了 7 个元素而构造一个复杂的数据结构。

很多程序员从来不考虑这些事情的实际后果。

James：让我发狂的一件事是，人们会为他们的开发工作而构建系统，有时做一个简单的线性排序查找就够了。他们知道当这个系统部署的时候，有可能会遇到 10 万个元素，而我只测试过 10 个元素的情况，所以我可能也做一个线性搜索，而且他们总是说“我会考虑性能改进的。”而这就像 Robert Frost 说的一样“但我知道路径延绵无尽头，恐怕我难以再回返。”许多代码就是这样。我有一种感觉：在构建系统以后再返回来修改并不常见，所以这个世界上到处都是慢得难以想象的系统。

是因为时间的限制还是懒惰呢，抑或是为非程序员而使编程变得越来越简单造成的？

James：跟它们每一个都稍微有点关系。这也归功于现代机器越来越快，每台机器里都有三台千兆赫时钟，现在你可以在有限的时间内完成一个无限循环。☺

你们为什么选择让 Java 运行在虚拟机上？

James：它对可移植性帮助极大。这样有助于便携性、可靠性，而且说来也怪，它还非常有助于提高性能。使用适时编译的时候，非常容易获得高性能。让 Java 运行在虚拟机上有许多好处。它对调试有极大的帮助。

除了虚拟机设计之外，你打算做一些别的事情吗，或者说你对 JVM 的工作满意吗？

James：实际上，我对它相当满意。JVM 的设计可能是整个系统架构中最稳定的一部分。如果我要离开时却发现某个问题需要解决，我是不会指责它的，因为它的状态非常好，它没有任何事情令你烦恼。还有一大帮非常聪明的人已经在虚拟机方面工作十几年了，这些人基本都是博士编译专家。

译注2：快速排序（Quicksort）由 C. A. R. Hoare 在 1962 年提出，它是对冒泡排序的一种改进。它的基本思想是：通过一趟排序将要排序的数据分割成独立的两部分，其中一部分的所有数据比另外一部分的所有数据都要小，然后再按此方法对这两部分数据分别进行快速排序，整个排序过程可以递归进行，以此让整个数据变成有序序列。

您会参照其他流行语言来修改 JVM 吗？

James：噢，可能会在某些方面有一些小的调整。现在，我们正在围绕一系列的语言设计问题进行仔细研究。有些东西对其他语言确实有帮助，因此你也想在虚拟机中做一些这样的修改，而事实上你要想找到这样的东西是极为困难的。

我们在虚拟机上遇到的重大问题是那些有点哲学色彩的难题。例如，因为没有裸指针，我们很难在虚拟机上实现类似于 C 或 C++这样的语言。

允许裸指针必然会带来很大的可靠性问题，所以我们决定绝不引入裸指针。我是说这会既涉及可靠性又涉及安全性，这实在是太可怕了。因此，要我把 C 和 C++引到虚拟机上来吗？肯定不会。

你举过一个例子，如果你开车时车载收音机坏了，汽车还是会走的，那我们是否需要为软件提供新的构件块，以避免当一部分出问题时整个系统都停止运转这个基本问题？

James：很多原因在于在这些语言系统当中建立的低级管道系统。C 语言最大的问题之一是“一切都会尖叫着停止”，因为它们使用的是指针方式。

一旦你因为使用指针而出现了任何内存错误，系统核心就会崩溃，你就完了。但是在 Java 中，一方面你不太可能会遇到指针 bug，另一方面出现故障时，你实际上还能控制它。

异常系统确实很好地避免了更多的意外带来的破坏，就好比当你的车载收音机坏了时，你只要简单地断开收音机就可以了。

实际上，用 Java 来构建大型企业系统的人会花一定的时间来保障系统各个部分之间相互地合理隔离，如此一来，出错的部分可以保持相对独立。

如果你看一看 Java 企业标准之类的东西时，你会发现很多带有容错机制的框架运行良好。

这样的话还能保证面向对象方法的完整性和正确性吗？

James：现在，面向对象的程序运行得很好。人们对于优势产品的讨论五花八门，他们对所有语言理论的批评性言论极有兴趣，这些批评往往是针对语言中的旁枝末节的，事实上面向对象编程的基本理念却是极其成功的，几乎没有显现出任何大的缺陷。

有时开发者使用对象技术好像是倍加困难。首先，他们必须设计一个可复用的组件，如果之后要做一些改变，他们就要写一些东西来精确地弥合旧系统留下来的缺口。基本上，我发现在“使用对象实现优秀的设计”和“对象会让事情变得更复杂之间”只有一线之隔。

James：噢，面向对象设计肯定需要某种较高的品味。人们会在这个领域中迷失方向、失去控制，这样的例子比比皆是，事情变得有些疯狂，但是实际上情况还好。接口和对象的使用非常成功，它迫使你去仔细思考每一个子系统之间如何相互关联，而这种理性的应用是非常重要的。

面向对象可以使你把一个系统分解开来，系统的每一部分都可以被拆分，这会对革新、调试、容错及其他一大堆的事情极有帮助。

是的，正确地使用面向对象需要某种较高的品味，但它实际上也不难。而且面向对象是非常、非常有价值的，

比某些人热衷的面条式代码（译注3）有价值多了：面条式代码互相之间的直接联系非常紧密，如果你试图更改一个部分，其余所有部分都会随之改变。这一点非常可怕。

如果你不使用面向对象的方法，那将会面临一个非常非常糟糕的世界。当你的系统开始变得很大，当你有一个很大的团队，或者当你的系统随着时间的变化不断演进时，面向对象就开始展现出它的威力了。

你的模块化工作做得越好，越需要将不同程序员的任务进行隔离。当其他地方修改的时候，它们之间的边界也必须修改，这是非常有用的。

12.3 并发性
Concurrency

人们时常谈论摩尔定律的结局：系统一定会变得越来越大，但并不一定会越来越快。你是否同意这个观点？

James：是的，我同意。摩尔定律是关于门数的。很容易就能看出，遵从摩尔定律，数年以来门数获得了极大的发展。但是如何转换为时钟频率？长久以来，它被转换为时钟频率的机会可以说是微乎其微。

为了获得良好的并发性，是否需要修改现有的设计？

James：噢，是的，它在很大程度上改变了设计，尽管从一个问题域转到另一个问题域会存在很多可变性。

所以像大多数数学软件一样，为了让一些数学软件适用于多线程领域，实际上你必须大幅度地修改算法。虽然在许多企业级软件中，软件通常在框架之内运行。

就像在 Java 中，有应用程序服务器框架 Java EE，而且在 EE 框架中，程序几乎无须意识到它们运行在多线程中；实际上是容器（应用程序服务器）来全权处理多线程问题，同时也是它来处理集群和多进程等所有问题。

这些问题已经被抽象出来，你也不必担心它们，而这些工作对于企业级软件来说是非常好的。企业级软件的特点是由许多彼此互不相关的小事务组成的，它们只是各自独立出现！

数学软件中有很多事务是相关的，它们共享公共数据，等等，而且事情会变得更为复杂。而且也没有什么明确的答案。

我们是否需要通过新的语言工具或库来描述它？是否需要所有程序员都用一种完全不同的思路来理解？

James：哦。这得视情况而定。我认为它和特定领域极为相关。对于大多数基于事务的企业及软件来说，你可以选择 EE 框架，这种框架完全可以直接处理多线程。而如果你使用的是一台 128 核的 Sun 机器，实际上你也很难能找到这样的机器，开发者根本不用去管它，它会充分调动自己的多核去完美地运行，一切都非常轻松。

译注3：面条式代码（Spaghetti code）是软件工程中反面模式的一种，是指一个代码的控制结构复杂、混乱而难以理解，尤其是用了很多 GOTO、异常、线程或其他无组织的分歧架构。得名于程序的流向像面条一样扭曲纠结。

这对程序员的技能要求不高，只要有一个适当的程序员，即使他对多线程并不熟悉，其他的交给 Java 就可以了。

James：没错。这些框架可以帮助人们把多线程问题抽象出来。你要知道，如果你在做仿真之类的事情，当你陷入更多的数值问题时，事情会变得非常困难。要把每一种图形算法和数值计算分解到多条线程上，这个事情从本质上来说就是很复杂的。

286

其中一些原因是因为你要进行数据结构访问和锁定等。通常，从数据结构和算法正确性方面来说，它在本质上非常困难。旅行商（或称货郎担）问题是一个特别棘手的问题。也有些问题比较容易，比如光线跟踪图像渲染，但它是一个领域特定的观测问题，也就是你能够处理单个像素，而且它们是完全独立的。

如果你有那样的硬件，可以在低至像素级的水平上来实现并行化。

James：说得对。实际上，它们表现得非常不错。大多数优良的光线跟踪图像渲染都是这么做的。如计算流体力学，它是天气预报及计算出飞机是否会坠落等这些事情的基础算法，因为流体各部分之间的交流非常多，所以那些问题变得更为复杂。分解一个共享地址空间系统非常容易，但在一个没有共享地址空间的集群之内分解确实是相当困难的。你知道，对于计算流体力学算法，它们很快就会因集群节点之间的通信成本而停止运转。他们在多核状态下表现更好，但往往深受算法特定之苦。

我之所以喜欢 Scala 这样的函数式语言，原因之一是：如果你用 Scala 编写了一个数值算法，那么编译器就可以进一步推理出算法的目的。它可以更好地实现从算法到多线程、多核分布式系统的映射。

这是因为 Scala 是一种纯函数式语言吗？

James：它并不是纯函数式语言。原因之一就是，对于多数人来说它具备双重性质。你也可以把它当成 Java，也可以把它当成纯函数式语言来编程。

有没有这样的问题域，其中共享内存多线程的性能要比函数式更好？

James：实际上，对于企业应用程序来说，这种基于框架的方式在多核分布系统上运行确实非常非常好。我认为 Scala 之类的系统并没有很大的优势。如果你是在编写旅行商之类的算法确实就更加有趣了。

看来来自 Green Project 和 Oak Project 的一个深思熟虑的决策好像是，在无处不在的多线程网络世界中，设计适用于这种网络的编程语言意味着需要用于同步的基本类型，而核心库也必须是线程安全的。

James：我们有大量的线程安全机制。实际上，这是一种非常可怕的情况，因为这通常意味着你的系统上只有一个 CPU 在运行，你肯定要付出一些代价。不过在这种特殊的情况下，我的意思是说，抽象化肯定会于事有补，因为有更多的抽象 API 和接口存在，Java 虚拟机之类的东西就会比真实的机器具有更好的抽象性。底层机制可以做很多改进。

287

在多线程情况下，原来的 HotSpot 虚拟机（译注4）会了解到这与单核的机器是不同的。

译注4：HotSpot 虚拟机是 Java SE 平台的核心组件。它实现了 Java 虚拟机规范，在 Java 运行环境中作为一个共享库提供。

使用 JIT（译注5）技术时，你可以说：“我确实不用担心这部分的同步问题，因为我知道从来没有死锁；我们从未为这个特定的内存段进行线程竞争”。

James：是的，而且它的出现既不可思议，又显而易见。没人意识到这点。这是与 64 位指针类似的问题。

使用 C 语言的人们深受困扰，因为他们必须让应用程序运行在 64 位系统上，而在 Java 应用程序中，绝对没有你必须要做的事。

12.4 设计一种语言
Designing a Language

要不是为了 Java，您还会选择 Scala 语言吗？

James：是的，很可能会。

所有这些新语言不仅仅是研究项目，而是真打实干要在 JVM 上构建一个真正强大的语言，您对此怎么看？

James：我认为它们相当酷。

您难道没感觉到威胁吗，好像它们汲取了你的全部设计精华并避开了你？

James：不，Java 的所有重要之处都在 VM 里。那是互操作性工作的基础。从某种意义来讲，Java 和 ASCII 的句法设计应该能让 C 和 C++程序员感觉舒服些。它确实做得很棒。大多数 C 和 C++程序员都能看一下 Java 代码，并且可以说，“噢，我能看懂。”

即使我不知道 API 的细节，从战术上讲我也明白它在做什么。

James：这确实就是重要的设计目标之一。在某些抽象世界中“什么有可能是世界上最好的编程语言”，这实际上并不是一个目标。我个人认为 Scala 是相当有趣的。Scala 的问题是，它是一种函数式编程语言，而要想用好它，大多数人都得颇费一番周折。

如果让我设计一种仅供我自己使用的编程语言，那很可能大多数人都得为之抓狂。

会不会是 Lisp 呢？

James：它不可能是 Lisp，不过在很多细节上可能与 Lisp 相似。

Bill Joy 曾经说您的目标让 C++程序员放弃并反对 Common Lisp。

James：如果你看一看什么是 JVM，那么从某种意义来说那就是事实。

译注5：JIT（just in time）即时编译技术。它允许实时地将 Java 解释型程序自动编译成本机机器语言，以使程序执行的速度更快。有些 JVM 包含 JIT 编译器。

288

对于程序员来说，VM 不应当很慢，或者无处不在的垃圾收集这种想法是非常高效的。

James：人们确实没有意识到一件事就是，垃圾收集对保证可靠性和安全性非常有益。来看一看 bug 频发的典型大系统：始终都是内存管理 bug。我对一段时期以来发现的所有 bug 进行了统计，特别是查找它们花费的时间。内存破坏 bug 的问题之一就是要花费过多的追查时间。我发誓不想再浪费一个小时再查一次。

我编写了 JVM 的前两个垃圾收集器。垃圾收集器的调试过程非常讨厌，不过一旦完成调试以后，它们就会正常工作。现在我们已经有了一些非常“火箭科学（Rocket Science）”（译注6）。

您最初编写的垃圾收集器是哪一种？

James：我需要一个能在很小的空间中工作的收集器。它在概念上是一个基本标记（mark），带有压缩（compaction）的清扫（sweep），而且还具有一定的异步运行能力。它没有更多的空间来实现更现代化的设计了。如果有人使用句柄，那么空间的使用会进一步压缩。

一个附加的间接指针可以帮助你完成复制吗？

James：因为我尝试使用 C 语言库，但是最初的版本是 semi-exact。在堆中的所有指针都是 exact 类型，但是在栈中就是 inexact 类型，因为它实际上会先扫描 C 语言栈来寻找是否存在看上去类似于指针的东西。

这是我见过的能处理好 C 语言栈的唯一方法，除非你根本不想使用 C 语言栈，而 C 语言栈既有优势也有劣势。

James：对。实际上，这就是 JVM 现在正在做的。它没有使用 C 堆栈。我的意思是它有自己的堆栈机制。对 C 代码，必须用一个独立的堆栈。关于 JNI 的一件棘手的事情就是在各领域之间的转换。

考虑到 C#中很多的灵感来自 Java，您认为 Java 还有哪些特性可供其他编程语言借鉴？

James：噢，我的意思是，C#基本上吸取了 Java 所有的特点，尽管它们非常奇怪地决定添加那些不安全的指针，从而使得安全性和可靠性大打折扣，我感觉这样做是非常傻的，不过人们还是几乎利用了 Java 的大部分特性。

您写了垃圾收集器，这样就不用在调试内存管理 bug 上浪费时间了。您如何评价 C++的指针和 Java 的引用？

James：C++中的指针是一场灾难。它们就是错误之源。它不仅只是直接地实现了指针，而且必须人工地管理回收，最重要的是你必须在指针和整数之间进行类型转换（cast）；而且你还必须建立许多 API。

289

您在 Java 中设计引用的目的是为了解决所有这些问题，同时它依然可提供 C++指针的优点？

James：没错，你可以完成在 C++中使用指针能够完成的所有重要任务。

有些问题是可以通过在语言里实现一个通用解决方案而避免的，您经常遇到过这样的问题么？

James：这种问题在 Java 语言中随处可见。异常机制就是一个例子。在 C++中，人们完全无视他们获得的种

译注6：火箭科学（rocket science），意思是很难做、很难懂的事情。

种错误代码。而 Java 可以使您轻松地处理出现的错误。

适当的默认选择能否帮助程序员不借助外部库或附加软件来编写出更好的代码？

James：多年以来，大多数语言都已经解决了大部分的这类问题。比如，C++中多年存在的多线程问题。Java 代码中多线程是被非常严格设计的，结果就是，Java 可以与多核机器合作得非常、非常好。

语言设计和用它这种语言编写的软件设计之间有何联系？

James：哦，这其中联系非常微妙，而且无处不在。我的意思是，一个面向对象的语言确实鼓励你构建模块化系统。你知道，当你拥有了一个强大的异常处理系统时，它可以鼓励你建立强健的（robust）系统。基本上，你能想到的每一个语言特性，都会对软件设计有着微妙的推动作用。

您希望在标准语言中还具有什么其他特性？自动代码检查怎么样？

James：噢，我们在编程工具里有很多类似的实现。你要知道，如果你看看 makeme(?)的功能，它基本上就像一个实时 LINT（译注7）等所有的那些高级工具。现在，你无法仅单独考虑一种语言，你应当把它们和工具放在一起来看待。

在设计一种语言时，您会考虑人们是如何调试它的吗？

James：要想实现一款高可靠性的软件，需要考虑很多重要的事情，但这不仅仅是调试本身这么简单。在调试方面我们也制定了大量相关的标准，例如系统如何与调试器进行交互等。

实际上，你在语言中做了很多工作，避免陷入你必须调试的困境；这就是内存管理器、强类型系统、线程模型遍布 Java 的原因，这些东西甚至会在你调试之前已经产生了积极的影响。

290

如果一种语言是平台无关的，会不会影响调试？

James：噢，你知道，从开发者的观点来看，调试是完全无缝的，它可以运行在 Mac 上，调试 Linux 服务器上的某些东西。它干得非常棒。

您如何调试 Java 代码？

James：我只使用 NetBeans。

您对 Java 程序员有什么建议吗？

James：使用 NetBeans 非常小心地构建任务，还有很受欢迎的 JUnit，把它们组合在一起效果会更好。

计算机科学系的大学生缺乏什么？

James：多数大学注重事情的技术层面，许多软件工程只是：“好的，给你一段代码，找出一个 bug 来”。而常见的大学作业可能是“写一段可以工作的软件”，这样你在开始时拥有的只是一张空表，因而你可以做任何事情。

译注7：LINT 是一个历史悠久，功能异常强大的静态代码检测工具。

而且，许多软件工程都是团队合作的社会动力学成果，但大学里根本就没教过这个。

您如何看待软件文档？

James：当然是越多越好。

对于 Java API，还有名叫 Javadoc 的工具，它可以从源代码中提取文档。人们往往发现软件文档与实际的 API 经常不匹配。因此，通过自动提取的很多样板文件，同步问题有了很大改进。实际上，Javadoc 生成的 API 文档是可信的。

因此，对于代码文档化来说，最重要的一件事就是使用 Javadoc 工具，然后是在你的代码中添加合适的注释，这样就可以人人受益了。

您认为在构建项目之前需要形式化的或完全的详细说明吗？

James：我对形式规范的看法非常复杂。我认为它们在理论上非常重要；但在实践中，它们看起来并不好用。对于一些小问题来说，它们通常表现尚可，不过问题越大，形式规范的用处就越小，只是因为它们不能很好地扩展。

更重要的是，形式规范通常并不会去解决实际问题，只是把问题转移了。它把你软件中的问题转化成了在你的规格说明中发现的 bug。然而，在规格说明中去发现 bug 往往很难。

即使你不做形式规范，也会做一些需求分析——许多组织都遵循这种瀑布式开发流程，在这种流程中，会有一个小组提出需求文档，然后把它分发给实际构建系统的人。

需求文档通常问题重重，但是只有确实存在很紧的反馈循环时，我们才会发现需求中的这些 bug；你不会在规格说明中发现 bug。因此，虽然我通常对需求和规格说明之类的文档乐此不疲，但我也不会太把它们当回事。我肯定不会指望这些文档能解决大问题。

我也曾经跟使用 UML 和其他语言的人聊过，我感觉听起来很有趣的一种想法是，我们使用高级设计语言来构建软件背后的逻辑。他们提到有可能在编码之前就把模型中的逻辑错误找出来。

James：是的，在 Java 领域中，已经有了许多基于模型的高级工具。你会发现，在 UI 框架、UML 框架的 Web 应用程序框架中，许多高级建模工具的功能是非常强大的。如果你有 NetBeans 软件，你会发现这是相当复杂的 UML 建模系统。你可以在一开始使用 UML 模型来详细描述一款软件，然后让它自动生成软件，或者你可以把 UML 建模工具用作一种“考古”工具来探究一款软件，它们会有帮助，不过仍然有一些问题。

12.5 反馈循环
Feedback Loop

关于 Java 语言自身而不是实现，您获得了多少反馈？

James：小伙子，我们获得了关于语言的很多反馈。

您是怎么处理这些反馈的呢？

James：如果某些特性只有一个或两个人想要，我们就倾向于忽略它。

因为在语言中，你必须非常清楚你改变的东西。在 API 中修改会稍微容易一些，但是通常来讲，要是没有特别强烈的需求，我们就不会做任何改动。

所以如果很多人都有同样的需求，我们就可能会说“噢，好的，那可能值得一做”；而对于百万分之一的开发者要求的功能，我们则可能会说：“把它加进去可能会弊大于利”。

向 Java 开放源代码后您获得了什么经验？

James：哦，我们有很多良性互动。我的意思是，我们在 1995 年开始提供 Java 源代码，人们已经大量地下载和使用，从学位论文项目到安全审计，现在，它变成了一种非常强大的语言。

它会使语言的实现和合作更加精益求精吗？

James：开源之后，会有很多人为其做贡献，这当然是件好事。它最后会成为一场对话。

您能以一种民主的方式来设计语言吗？

James：这是非常非常好的一条界线，如果过于民主，你得到的只会是一堆垃圾；而过于中央集权的话，就可能没人会使用，因为它只是一个人的观点。

所以说，和大众交流，还要有一个适度严密的决策过程，这确实是非常重要的两件事。

您认为开发一种新语言是有益的吗？或者如果有新目标，并且您可以做点什么来实现这个目标，您会去写一种新语言吗？

James：我想两者可能都有一些吧。我想培育一种语言不是问题，不过我认为门槛要求还是相当高的。当被证实有其价值时，培育语言是很不错。如果价值确实非常大，这种随意修改句法之类的事情没什么意义。如果你某天早上起来判定括号很讨厌，我们可能会对此无动于衷。但是对于某些可以真正影响人们构建软件能力的事情来说，我们就会去改变，几年前，我们做了大量工作将范式加入到 Java 中，这就是一件相当好的工作。

设计 API 时您使用什么准则？

James：我的第一个原则是尽可能让它小。而且，我想我的第二个原则是基于用例来进行设计。人们在设计一个 API 时总是有这样的失败模式：他们就像在真空当中设计 API 一样，坐在那里说：“噢，有人可能想要这么做，”或“有人可能想要那么做”，这只会使 API 无限制扩展，变得比实际需要复杂得多。

但是当你实际做这些的时候，你会问“人们想用这个来干什么呢”？去看看那些人们试图去拖曳菜单或是连接网络的系统。你知道其他地方运行的是什么吗？与 API 所写的功能相比，人们实际上又在做什么别的事情呢？

只要对其他语言编写的软件进行统计分析，你就会在 API 设计和语言设计中做很多有趣的事情。

在您设计 Java 的过程中，有什么经验和教训要告诉别人吗？

James：设计语言并不是很难。设计一种语言时，最重要的不是设计语言，而是要搞清楚为什么设计语言。

它的背景是什么，人们会用它来做什么？

真正令 Java 与众不同的是它在网络中的应用。网络会对编程语言设计有什么潜在影响？结果表明，它具有深远的影响，在某种意义上，一旦你考虑到网络的潜在影响，语言设计的选择上就会非常简单。

Java 是否影响了人们对平台无关性的看法？

James：我认为普通用户不会关心平台无关性。我想这一点很有趣，我从一名工程师的角度出发构建出了这些大规模的系统，但是公众根本没有意识这一点。

当你在 ATM 机或收银机上使用 Visa 信用卡时，后面肯定有一大堆 Java 代码正运行在 SUN、IBM、Del 或 HP 机器上，也可能在 x86 架构或是 PowerPC 架构等上，而你在读卡机上刷卡时根本没有留意这些。

而且，这些机制都隐藏在屏幕之后。你乘坐地铁的时候，如果使用类似的卡，比如说在伦敦地铁公司（London Underground）使用 Oyster 卡的时候，那肯定就是一个基于 Java 的系统。

你确实使用了与各种平台无关的特性，而且，如果要强迫消费者意识到构建系统所用的编程语言，那么这个系统就真的是非常失败。我们的目标之一就是做到完全透明，让人们从中解脱出来。当然，这让销售人员心有不悦。他们宁可让你每次乘坐伦敦地铁时都有一个 Java 标识在你面前晃动，不过，这也太荒唐了。