

## C语言家族

— Dennis Ritchie, Bjarne Stroustrup, James Gosling 访谈录

Herb Sutter 访 蒋贤哲 译

///感谢蒋贤哲先生提供译稿。对于其中可能存在的任何瑕疵，请反馈至[xianzhe.jiang@gmail.com](mailto:xianzhe.jiang@gmail.com)，谢谢。

C语言家族「C、C++、Java」主导商业化编程逾 30 多年之久，目前这三种语言都正处在各自关键的转折点。

- C 的第二版 ISO/ANSI 标准已经颁布「C99 已于 1999 年 12 月份正式发布」；C 仍将是世界上最具影响力的语言之一，尤其是在嵌入式系统编程方面。
- 针对 C++的 ISO/ANSI 标准的第一次正式修订将于 2000 年 12 月份完成。以其对面向对象编程和泛型编程无与伦比的支持，C++成为目前世界上应用最为广泛的商业编程语言之一，而且应用的范围仍将稳步增长。
- 目前，在从客户端编程到服务端编程的各个领域，Java 的流行势头正劲。SUN 最近更为坚信 Java 最终将成为一个事实上的标准，而不是一个正式的 ISO/ANSI 或 ECMA 标准，而且已经放弃了在正式标准化方面的努力。

是什么让 C 语言家族拥有如此强大的影响力呢？在不久的将来，在这些语言以及相关的语言领域内，我们将会看到什么呢？本月出版的 *C++ Report* 和 *Java Report* 同时刊登了针对三位传奇式的人物所进行的具有里程碑意义的联合采访，他们三位根本无需介绍，他们就是 C、C++、Java 的缔造者：**Dennis Ritchie**、**Bjarne Stroustrup**、**James Gosling**。

再回首：我们来自何方？

### 1. 什么使 C 语言家族如此成功？应用如此广泛？

#### Dennis Ritchie:

对我而言，理解各种各样的细节经常会有那么一点神秘感。显然，由于被用作 Unix 的混合语言「lingua franca」，C 语言<sup>1</sup>早期「70 年代以及 80 年代的大部分时期」的使用由此得到了相当大的激励。值此期间 Unix 正在研究和科研团体中兴起，直至 80 年代，Unix 成为工作站行业的软件基础。一方面，在某种程度上与 C、Unix 的非政治属性有关「在 1984 年之前，没有与计算机硬件领域的任一寡头有关联」；另一方面，也存在技术方面和准技术方面的因素：该语言被证明是既能够站在一个足够高的角度来表述事情以便使跨越硬件的移植性成为可能；又因为其需求条件很低，以致实施起来简单易行。

#### Bjarne Stroustrup:

C 和 C++之所以如此流行，乃是因为与其他语言相比，更富有弹性、更简单、更高效。C 语言最初的流行在很大程度上受惠于 Unix 的盛行，而 C++最初的流行则是由于它与 C 的高度兼容性。

对于 C 和 C++而言，AT&T 并没有试图去垄断它们，而是允许其研究人员去实现其他方案，这一点对于这两种语言的成功是至关重要的。另外，AT&T 自 C 和 C++的 ANSI 与 ISO 标准化工作伊始，就给予了全力支持。在成为稳健的语言和众多厂商开始竞争之前，C 和 C++并不具有很大的市场。C 和 C++这种非商业性的传播方式强烈地吸引了大批编程人员。

Java 的设计明显异于以上的两种语言，似乎体现出一种很是不同的理念。它最初的流行与在宣传一个语言方面而所开展过的最为密集市场策略有关。自其首次商业亮相开始，Java 的市场定位就不同于其他语言，而且似乎要优越于其他语言。有意思的是，Java 的市场定位于各种组织机构的人员，而不仅仅是程序员。

我怀疑 C/C++ 与 Java 之间诸多不同的根因在于 AT&T 从根本上来讲是计算机、编程语言、工具的使用者，而 Sun 从根本上来讲则是这些东西的提供者。

在这里提醒大家一下：C 和 C++ 都是在位于默里山下贝尔实验室的计算科学研究中心开发出来的，而且是在贝尔实验室和 AT&T 内部开始被正式使用的。当时贝尔实验室是 AT&T 研发中心。现在，贝尔实验室的部分已经成为朗讯的研发部门，而其余部分仍隶属于 AT&T，现在被称为 AT&T 实验室。

所有这三种语言都不是从根本上不同于同时代的其他语言，也不是多么地优越于当时的其他语言。但是它们确实不错，而且是机遇和当时如 Unix、廉价、市场策略「只针对 Java」等社会因素的受益者。

在技术因素方面，C 和 C++ 得益于和机器紧密相连，以及没有对开发人员所想表达的内容进行过多的人为干预。这样就允许底层的系统任务由这些语言来完成，又能够充分利用机器的性能将其交付给用户使用。Java 则得益于它运行在虚拟机上，以及大量的库，从而缩短了程序员在高产之前所需的时间。Unix 对 C 起到了同样的推动作用。但 C++ 却因为它那容量很大但内容却支离破碎的库而倍受折磨，其中许多东西都是由许多相互竞争的厂商所提供的专属产品。

### James Gosling:

我认为首要的原因就是这个家族的语言都是非常务实的语言。大体上，它们都不是语言设计的试验品，而是被那些希望真正做事情的人们拿来作为工具使用。由于被用来编写 Unix 操作系统及其构件，C 曾受到强大的促进作用，同样 C 中很多的东西都直接来源于要用它来实现一个高效的操作系统，以及被用在只有 32K 内存的机器上。

## 2. C、C++、JAVA 最初的主要设计目标是什么？是为了解决哪些问题？哪些问题当时并没有打算去解决呢？

### Ritchie:

C 的出发点「不同于其直接前驱 B」是考虑要设计一种面向字的机器语言，而且适用于其他可用的新型硬件，特别是 PDP-11。没有多久就认识到早期所引入的各类东西使得它适用于更多不同的机器。

### Stroustrup:

我当初设计 C++ 的目标是既能够写出像 Simula 程序一样优雅又像 C 程序那样高效的程序。

我当时所考虑的项目与在一个局域网中跨越大量计算机的分布式操作系统的功能和应用有关。我不得不同时考虑整个系统的复杂度以及低层功能的效率。来自 Simula 的面向对象机制帮助我解决了前者，而 C 中关于系统编程的机制解决了后者。

### Gosling:

与 C 一样，Java 的设计目标也是能够于打造范围相对比较明确的软件，我们一直致力于打造充分支持分布式、可靠、人机交互型的软件，在这些软件中，可靠性和安全性至关重要，从而宁愿牺牲一点性能。

例如，Java 与 C 一个区别就是在 Java 中拥有边界检查功能的真正数组，而事实证明这对于可靠性和安全性都是至关重要的。如果你回顾一下 Internet 安全漏洞的历史，就会发现一些重大缺陷都是由缓冲溢出引起的，即超越了静

态分配的数组的尾端。你看一下以前解决问题的日志，就会发现内存一致性问题经常是各种错误的源头。

然而，如果你想做一些类似 Java 所做的事情，其中对内存模式又有比较严格的要求，那么性能上将会有一定的损失。事实证明利用性能优异的编译器，这些性能上的损失几乎可以忽略不计，但是要达到这一点需要高水准的编译器，当然今天编译器的优化性能是 20 年前的编译器所不能及的。Java 因为追求安全性和可靠性从而与 C 产生了一些差异性，在 Java 中性能方面的问题可以由高水准的编译器进行弥补。而 C 最初的设计方案中，大都是由“不借助性能优异的编译器也可获得高质量代码”的目标推动的。

### 3. 随着岁月的流逝 C、C++、Java 的目标是否发生变化，是否从根本上有所变化呢，以及为什么呢？

#### Ritchie:

在过去的许多年里，我个人关于 C 的目标并没有太大的变化，在 1989 年和 1999 年的标准化工作中，我一直都不是一个中心人物。1989 的 ANSI 和 ISO 标准所制定的内容要远比我们最初的文档出色；其中最重要的引入部分是我早就应当要做的事情「函数原型」。另外，Bjarne 早期在 C++ 方面所做的工作可能具有最为重要的直接影响。

#### Stroustrup:

我认为自己关于 C++ 的整体目标并没有太大的变化。我仍旧希望写出既优雅又高效的程序，我仍然希望正规的系统编程「既包括在机器层面又包括在资源受限的系统上的任务」用 C++ 时可以实现的。

重点与风格上的主要变动来自后来采用了模板和异常机制。从第一次开始写下 C++ 的雏形「即带类的 C」时起，我就知道对容器进行访问需要一种高效的静态检查方法。模板提供了这一切，而且 STL 对如何实际运用这些容器提供了一套技巧。

作为一个更为全面的回答，我把自己在 C++ 目标及设计方面的所思所虑融入在 *The Design and Evolution of C++<sup>2</sup>* 一书中。在这里你可以找到 C++ 为什么是现在这个样子而没有采取其他可行措施之类问题的答案。

#### Gosling:

我猜测，以可控性和可靠性为目标的事情已经越来越流行，因为事实证明它们已经弥补了类似开发时间和人力资源之类的不足。在解释语言「其中一切都具有弹性且是动态的」与类似 C 的静态语言「其中一切静态预编译的」之间采用或是试着去创建另外一种语言，都似乎处在一个尴尬的位置。而 Java 则正处在这两者之间的地狱之中，世界上的联系方式比以往任何时候都变得更有网络性质，更关注安全问题，和人类联系更加密切，Java 的基本设计目标大概上就体现在那里。如果说有什么变化的话，就是它们变得愈发健壮。

### 4. 在你们设计 C、C++、Java 时，哪些特性实现起来最为困难？哪些特性最难以一种最初用户可以接受的方式进行设计呢？

#### Ritchie:

大体上讲，这里有一个取舍的问题。C 与其家族之外的其他语言相比，一个奇怪的方面就是声明语法，其中「一种来自 FORTRAN 的方法」一个类型一旦被提及，那么该变量就得以一种能够反映它在表达式中用法的方法进行修饰。有许多人不喜欢这样，所以我这样做是否正确是一个可以讨论的问题。

一个与此紧密相关的方面就是如何对待数组和指针。考虑到与此相关问题的长期性以及 C++ 为了能够从 C 中脱离开这个问题所经历的曲折来看，它可以被算作一个错误。

**Stroustrup:**

和 C 保持适度的兼容性一直是一个有争议性的问题。为了我对于 C++ 的期望，我需要比 C 提供的更严格静态类型检查。然而，每一个方面的不兼容对于有些人来说就意味着麻烦，特别是对于那些视旧代码「无论写地怎么差劲」胜于新机制「无论怎么样有用」的人。在旧的代码可能被分拆时，程序员和市场人员的谴责在措辞方面将会异常的冷酷，即使这些旧代码是用另外一种语言编写或是以一种非正规的方言编写的。

但是我认为在该方面的争论还是值得的，因为 C、C++ 团体享受到了真正意义上的兼容性。另外，我在 C++ 非抽象领域的许多工作已经被反馈到标准 C 中。例如函数声明「原型」、常量「Dennis 也参与该方面的设计」、在任何地方进行声明，以及「//」类型的注释。

模板在取得正确、完善方面花费了很长的时间。在某种程度上，它们源自于我在「带类的 C」上面进行的宏试验「尽管大部分都失败了」。设计一个满足容器静态类型安全这样的基本需求的参数化类型的机制是比较容易的，但是为了实现弹性和效率则是非常困难的，这需要试验以及来自真正使用时的反馈。目前在标准 C++ 实现的范型编程超出了大部分人的预料之外「尤其是在一个极为强调效率的语言之中」。我猜想这是一个在未来 5 年将会取得惊人进展的领域，在该领域内语言特性可能需要支持新的技术。

**Gosling:**

确实存在很多非常差的设计方案，而且充斥各处。例如，在人们真正关心的事情当中，包括一个在 Java 中被称为接口的东西。接口中的很多设计直接借鉴于标准 C，而且大家对它是褒贬不一，真的是一个很难平衡的方案。

5. 您们是否添加过一些用户并不像你们一样欣赏的特性，而后又不得不去反对或是去除它们？你们从这些经历中学到了什么呢？

**Ritchie:**

在一些用户的压力下，我增加了一些我自身并不看好的特性。枚举类型有一点多余，位域更是如此。关键字「static」非常令人奇怪，因为它既表达了一个存贮生存周期又表达了标准中所谓的连接性「外部可视性」。确实存在许多冗余的地方。

除了一些诸如关键字「entry」之类无关紧要的东西外，我不记得我所添加的特性中有哪些是不得不去除掉的。

**Stroustrup:**

在我设计带类的 C「C with Classes」与 C++ 时，我非常热衷于这样一个想法「我现在还是」，即一个类应当定义其成员函数的运行环境。这就引出了能够创建该种环境以及获取所需资源的构造函数的概念。而析构函数则是相反的过程「负责释放那些资源」。这些想法是设计异常和资源管理方面的基石。可以参考 *The C++ Programming Language*<sup>3</sup> 的新增附录 E *Standard-Library Exception Safety*。该附录可由我的主页<sup>4</sup>进行下载。

基于这一连串的想法，带类的 C「C with Classes」可以让你定义一个在进入成员函数之前被调用的调用函数「call() function」，以及一个在退出成员函数之后被调用的返回函数「return() function」。这种调用/返回「call()/return()」机制是为了让程序员可以管理调用一个单独成员函数所需的资源。这允许我对关键任务库的进行监控。一个任务的调用函数获取一个锁，而其对应的返回函数则释放该锁。该想法部分来源于 Lisp 的「之前」和「之后」方法。但是这种概念并不完善，例如，你不能够通过调用函数「call()」访问一个成员函数的参数。更糟糕的是，在说服人们相信该概念的有用性方面我是彻底失败的，所以在设计 C++ 时，我去掉了调用函数「call()」和返回函数「return()」。

我最近重温了这个问题，而且利用标准 C++ 写了一个模板，针对对象的调用采用任意的后缀和后缀进行封装。与

此有关的一篇文章 *Wrapping C++ Member Function Calls* 发表于上个月的 *C++ Report*<sup>5</sup>。

我试着做一些有关声明语法方面的事情，我考虑使用后缀 `->` 来替代 `*`；而且试验着使变量在声明序列里面的位置可以进行互换。

例如：

```
int (*p)[10];    // 指向整型数组的指针
int p->[10];     // 另外一种替代方式
int->[10] p;     // 另外一种替代方式
->[10]int p;     // 另外一种替代方式
p: ->[10]int;    // 另外一种替代方式
```

遗憾的是，在我对这组想法进行摸索之后，什么结果也没有。我对基本 C 语法所进行的局域性改善，唯一的結果就是又使我花了几天功夫将它们还原回去。

### Gosling:

我没有此类问题。我个人有这样一个原则，那就是基本上不会仅仅因为我个人认为某些东西很酷就将其加入。因为我自始至终拥有一个用户团体，在加入任何新特性之前，我会一直等到几个人一起反对我为止。如果有人说“哦，难道这不酷”，通常我将不予理睬，直到有两三个人跑来对我说“Java 应当做这个”，此时我将着手行动，人们将来也许会实地用到它。

当你从一个公寓搬到另外一个公寓时，有这样一个原则。一个有趣的实验是将你自己公寓中的东西进行打包，将所有的东西都放在盒子里面，然后搬到另外一个公寓中，并在你需要这些东西之前不要打开包裹。那么在你做第一顿饭的时候，开始从包裹中取出东西。经过一个月左右的时间，你就可以利用这件事来准确勾勒出你生活中哪些东西是确实需要的，然后不管你对剩余的东西是多么地喜欢也不管它们是怎么酷，通通将其丢弃。这可以令人惊讶地简化你的生活，你也可以利用这条原则来处理各种各样的设计问题：不要做那些仅仅是比较酷或是令人感兴趣的事情。

### 6. 如果岁月能够重来，而且你们获悉现在的一切，那么在设计 C/C++/Java 时，你们将会采取什么不一样的方法？

### Ritchie:

显然不存在异常严重的错误。当然存在一些我至少会重新考虑的方面「上面提到的一些细节」。

### Stroustrup:

你不可能回到过去，而且重复做相同的事情是没有意义的。

像我经常所提到的一样，我认为没有能够打造一个比较大的标准库是我最大的失误。另外，我非常希望能够在关于 C 声明语法方面作一些事情。如果能够早点让模板融入到语言之中将会帮助许多人能够更好地使用 C++，但是我不知道如何来足够好地来处理这些事情。因为不愿莽撞行事而拯救了我和其他的用户们，这通常使我不想在这些方面再重新设想年轻时的我将会如何考虑。

目前，我很高兴许多编译器都和标准靠得越来越近。那么我就可以使用标准 C++ 来写代码，这就意味着能够在不同的编译器之间和机器之间编写、移植优雅的代码。编程的乐趣以及现在我所写的代码的质量能够防止我过多地去想以前所犯的错误。

### Gosling:

岁月重来，许多事情我都将会以不同的方式来处理。有许多事情我不是完全满意但是又不清楚正确答案是什么。

我对于类与接口之间的差别不是很满意，从许多方面看来正确的方法将不会特别困难。

有许多事情，例如多个返回值是这些天我一直希望添加上的特性之一。这是我真正喜欢的特性，而且大部分人也都在期待着它，“哈？”另外一个众望所归也是我一直致力于去做的机制就是类似 Eiffel 方式的预编译「preconditions」、后编译「postconditions」以及断定「assertions」，而实际上那些当时使用该机制的用户的普遍回答则是“哼？为什么我会需要这样的一种功能？”我认为现在的一般开发者也会这样说。但是也有相当数量的人信仰契约式设计「Design By Contract」，而且它是众人所喜欢的事情之一，“要是世界上剩余的人受到足够的教育以便能够理解它是什么，他们将会渐向佳境”。我非常赞同这一观点，而问题是世上的大部分人很少关心该类事情。

对有些事情我确实有点感到痛心，比如运算符重载。我以纯粹的个人选择剔除掉运算符重载是因为已经目睹了它在 C++ 中被许多人误用。在过去的 5 年到 6 年中，我已经花费了大量的时间来调查人们如何运用运算符重载，结果则是令人感到兴奋不已，因为可以将使用人群划分为三部分：大概有 20% 到 30% 的人认为运算符重载是错误的源头；一些人使用运算符重载做了一些使其备受煎熬的工作，因为他们使用“+”来表示链表插入，这使工作变得混淆不清。众多问题都起源于这样的事实：只有半打的运算符可以有意义地进行重载，然而有大量的运算符都被进行重载，所以你不得不在那些经常和你直觉相违背的选项中进行抉择。占总数约有 10% 的人则实际上对运算符重载运用得当而且真正地对它关注，对于他们而言该项功能是至关重要；这些人几乎都是从事数字运算工作，其中概念对于吸引人们的直觉很重要，因为“+”的意思和他们的直觉相一致，在 a、b 为复数、矩阵或是其他一些有意义的东西时，就有能力来直接讲“a+b”。当遇到类似乘法的事情时你将会变得摇摆不定，因为存在多种多样的乘法运算符，有从根本上不同的矢量积、点积，然而只有仅仅一个运算符，你将如何操作？而且没有平方根的运算符。这两个团体是两个极端，另外夹在中间占 60% 的摇摆团体是那些对以上任一种方法都漠不关心的人。那些认为运算符重载是个坏主义的人所组成的阵营，仅仅从我个人的统计来看，已经明显超过那些专门从事数字运算的人，当然他们的呼声也就高过另一方。所以，当今语言中那些特征都是由团体进行投票加以选择，这不是那些小的标准团体，而是大规模的，那么就很难使运算符重载这一特征得以加入。这样运算符重载这样一个重要的特性就被排除在外了。这也是众多问题的衡量策略。

## 运用语言

### 1. 以您们的经验来看，开发者在使用 C、C++、Java 时所存在的最普遍的错误是什么？

#### Ritchie:

我真的不知道该如何回答这个问题。我认为低级的错误与类型错误，特别是与数组下标和指针引用有关。尽管规则已经很清楚地告诉什么是你能做的，但是在实施的时候，习惯上很少做检查。

我也想冒昧地指出真正严重的问题与没有构思出整个系统的宏观规模结构有关：什么东西在什么地方可视、对事物进行命名、谁可以改动什么。在这方面 C 本身对你的帮助非常有限；你必须自己设计出一个方案来。

#### Stroustrup:

如果像使用 C 或是 Smalltalk 一样使用 C++，那么就会导致出一个性能上严重不理想的 C++。为了能够很好地使用 C++，你得使用一些本质上的 C++ 风格。这些风格是由于使用微而具体的类以及模板才得以体现出来的。一个深受 C 影响的编程风格倾向与使用大量的数组、智能指针操作、类型转换和宏，而不去使用标准库的机制「例如 vector、string、map」。一个深受 Smalltalk 影响的编程风格试图将所有的类都塞到一个继承结构中，而不得不大量运用类型转换「也经常使用宏」。在上述情形中，关键的抽象将会消逝在大量的具体实施细节中，人们使自己「本来不需要」身陷在内存分配、类型转换和宏所构成的剪不断理还乱的乱网之中。

一个很普通但又特别使我感到苦恼的难题就是复杂基类的使用，它将大量的数据成员作为接口的一部分提供给用户。目前抽象基类已经提供了更好的接口以供使用。当时我花费了 10 余年的时间来尝试着根除此类错误：由于没有明确的语言特征支持致使该想法无法实现之后，我在 1989 年增建了抽象类。

这个想法很简单：

```
class interface { // 用户可见
    // 纯虚拟函数
};
class my_implementation : public interface {
    // 数据
    // 函数
    // 覆盖函数
};
```

当用户代码只能看到接口时，就可以预防对“my implementation”的改动。

如果编译时需要普通的数据结构或是操作，它们可以添加在一个仅对编译器可见的基类里面：

```
class common { // 对编译器可见
    // 数据
    // 函数
};
class my_implementation : public interface, protected common {
    // 数据
    // 函数
    // 覆盖函数
};
class your_implementation : public interface, protected common {
    // 数据
    // 函数
    // 覆盖函数
};
```

这是多重继承最简单和最基本的应用之一。

对于许多人而言，希望从 C++ 获得更大的收获所需要的不再是新特征，而仅仅是一个更为适宜的设计和编程风格。关于这一点，我已经写了几篇文章「参考我主页上面的论文部分」。特别是在 *Learning Standard C++ as a New Language*<sup>6</sup> 中对 C 风格和 C++ 风格的编程进行了简单对比。

### Gosling:

也许一个最为普遍的错误就是没有把面向对象编程作为一种需要有鉴赏力的工作。有许多人根本不理解面向对象编程，而仅仅是写一些基于过程的代码，然后将他们的程序写成一个塞满了大量方法的巨型类；他们试图以 C 编程风格来处理面向对象编程。所以经常有人说“面向对象的程序设计，对象！它们酷呆了，咱们大量使用它们吧！”。实际

上, 这就像教育系统上的一个缺陷一样, 如果你看到大量的大学毕业生所被教导仅仅成“对象很不错, 大量使用它们!”, 那么你看到的程序中肯定充满了大量的微型适配器、一到两行的方法, 你不得不对它们进行整理, 此时就发现这些程序的所有时间都是花费在了函数调用方面。那么唯一一种能够很好应付各个方面的优化编译器, 就是那些花费几乎所有时间来消除函数调用和实施大量函数内联的编译器。

2. 依您们的经验来看, 从一个新手成长为一个对 C、C++、Java 相当精通、能够写出非凡产品代码的开发者, 需要多长时间? 对于一个在另外一种或是更多其他语言方面有经验的程序员来说, 又需要多长时间呢? 怎样才能使这个时间缩短呢?

**Ritchie:**

我对此问题也无法回答。有一个与此类似的老套笑话, “嗨, 我从来没有必要来学习 C……”

**Stroustrup:**

这在很大程度上依赖于初学者的背景、第一个试着用 C++ 实现的任务的复杂程度、教与学的方法。

对于一个编程初学者来讲, 一年半的时间似乎比较合适。一个对 C++ 及其所支持的技巧感到陌生的有经验的程序员, 半年的时间更为可能。显然, 我所说的必需时间是指能够在一个重大应用中真正应用语言的各种机制。学写类似“Hello World”的程序可能几分钟就可以了。

我认为一个设计良好的库在提供一个平滑学习曲线以及正确梳理学习 C++ 概念方面是至关重要的。例如, 如果能够利用 C++ 标准库, 那么在学习基本类型、作用域、控制结构概念方面就不用同时兼顾到数组、指针、内存管理等方面。这些基本的底层概念最好能够在稍后一点再进行学习。

在依赖于库的教学中当然存在一定的危险。它们在能够提供一种能力的现象背后隐藏着相当的无知。编程教学的一个目的必须使所教的内容能够被理解, 而不是具有什么神秘感。对许多编程者而言, 系统的行为甚至基本库都满是神秘感, 这是非常危险的。当标准库在其中通常仅仅只作为面向所有用户的便利机制时, C++「C」就成为一种功能强大的语言。

**Gosling:**

我知道对于一些极具 C++ 天赋的程序员而言, 一个下午的时间已经足以完成许多事情。你可能花费了大量的时间来浏览这些库的手册。语言本身可能很容易学习, 但是库中所包含的大量资料占用了大部分时间, 最好的学习方法就是马上写代码、马是开始使用它, 当你需要什么东西的时候, 再去查。

至于那些以前从未写过代码的人来讲, 我就不清楚了。有这样一种比较有趣的现象: 作为一种惯例, 人们过去在大学里面所学的第一门语言乃是 Pascal。原因大多是因为 Pascal 比较简单和简练, 另外, 能够出现在 Pascal 中的错误在别的地方将会以某种更为明显的方式出现。在编程初学者所犯的错误中仅仅指针检查和数组边界检查就占据了绝大部分, 所以第一学年的课程在很大程度上都是和 Pascal 有关。直到因为在 C 中做那些繁杂的工作变得如此简单, C 才得以出现; 拥有一个数组变得如此的简单, 当写下“for(i = 1; i <= length; ...)”时, 你不得不解释, “不, 它不小于或是等于数组的长度而且它不是 1 而是 0, 而且是小于。”但是, 你的循环运行得相当正常, 只是有时你可能会为那些紧紧跟在数组后面的宏的头文件备受折磨, 而且没有人会告诉那是怎么会事。系统当然不会朝你作怪样, 然而程序却会以各种不同的方式崩溃, 因为摧毁其他人的数据结构是非常简单的, 而且一段程序在这里看起来运行得非常良好, 但是在另外一个地方就好像要崩溃一样, 因为该段程序是运行在被第一段程序所摧毁的数据结构上。这种现象在 C、C++ 或是其他任何没有真正内存模式的语言中是普遍存在的。在最近的几年里, 人们实际上已经在把 Java 作为第一门编程语言进行教学方面做了很多努力, 因为 Java 拥有很多安全特性以致于它可以很方便地作为一门启蒙课程,

另外与 Pascal 不同，它是确实与职业具有商业性的联系。

## 展望未来：我们将走向何方？

### 1. C 语言家族之中是否遗漏过一些重要的特性？对 C/C++/Java 各自而言情况又如何呢？

#### Stroustrup:

从哪种程度来考虑 C、C++、Java 是否构成一个语言家族呢？C 和 C++ 具有很多相同的子集，而且在扭转这两种语言之间不可逆转的分离方面，已经做了多年不懈的努力，之所以出现这种趋势乃是因为它们分别是由不同的标准机构控制的。但是 Java 并没有提供真正的一致性，它和 C、C++ 尽管拥有相似的语法结构但是却拥有不同的语义。显然，Java 虽然表面上借鉴自 C 和 C++，但是实质上却更多借鉴于 Modula-3。

不容置疑的是，经过添加一些特性，所有这三种语言都能够得到显著的改进。但是，我很难相信存在一个能够同时使这三种语言都能够得到改进的重要特性。

#### Gosling:

当然所存在的事实足以证明没有任何实质性的特征被遗漏，因为人们能够使其工作得以顺利进展。在这方面关注更多的是语言特性的改变而不是你实际上能够做什么。一些更为明显的特性将会显身于新的语言之中。C 中最令我郁闷的一件事情就是弱内存模式、不能将一个指向字符的指针转换为一个指向整数的指针的情形，如果你这样操作，那么导致的结果将是一些匪夷所思的事情。但是当你能够改变这一切的时候它已经不再是 C，而是 Java 了。

如果你看过 Java 的 PFE 日志就会发现，仅仅有两种特征是人们频繁要求添加的：一个就是断言「assertions」，实际上有一个团队正在为此努力，不久就可以把这一特征加进 Java 之中。另外一个就是类型的多态，譬如模板一类的东西，事实上 Java 目前就具有一个蹩脚的模板系统，体现所有对象的根对象，即 Object。目前也有一个团体正在为 Java 中的类型多态努力。但是事实证明这是一个异常困难的问题。尽管我认为类型多态是一个很好的想法，但在 Java 中仍然舍弃的原因之一就是在学术圈内在于其正确实现方法方面存在相当多的争议。你会发现大部分人的态度都异常强硬，以至于很难找到一致的意见。与此相反，另外一些观点则是非常鲜明：垃圾回收机制一个好主义；而跳转机制则是一个缺陷。

我认为在一般应用性方面语言本身已经相当完美，而使人感兴趣则是在那些更加注重专业性的领域，主要有两个分支：一个注重打造更多的库，如今正在打造的 Java 库超出了任何人的预料；另一个则注重于一些为了特殊用户而创建的语言特性。我的确认为运算符重载对于那些做算术运算的用户来说是至关重要的。在商业领域的人们拥有相当令人信服的理由来要求那些有点类似 COBOL，例如与数据库查询有关的特征。如果你使用 JDBC API 来进行数据库查询，就会有些笨拙，用同样的方法创建数学表达式也有些笨拙。为了在一定程度上使其不再笨拙，系统需要在更深的层次上对数据库和数学的概念进行理解。

### 2. 所有的语言都会随着岁月的变更而改变，拥有一个强大的用户基础既能够支持也能够限制这些变化。C/C++/Java 在发布不同版本时是如何处理这些变化的？在已经拥有了大量的编程者的基础之后，在提升改进方面首要的考虑因素以及技术是什么？

#### Ritchie:

我现在已经彻底明白这是一件异常困难的事情，人们一旦习惯一种成功的语言，不管他们的真实意愿如何，都会变得十分保守。问题是一旦拥有了固定的基础，即使“向上兼容”的扩展对大部分人来讲也会变为一个难以解决的症

结。那些仍然没有升级的机制对使用了这些扩展的程序则无计可施。如果他们比较幸运的话，则至少会有新的语法指出这些问题，因为编译器无法接受它。那么当运气较差时，他们则会遇到各种各样的稀奇古怪问题，因为新语言恰好定义了一些以前没有定义的内容，而新程序若要正常运行则依赖于这些行为的详细说明。

#### Stroustrup:

我认为在语言演化方面的理想策略是在做出决定之前先进行广泛的实验，应当在不损坏旧代码的前提下定义新的特征，且应当使它们在各个地方都适用。然后，经过 1 到 2 年的时间去检查「禁止、反对」旧特征，再经过 1 到 2 年时间就把这些旧特性移除。遗憾的是，这种策略需要不同提供商之间的协调平衡。

较早的编译器由于不利于大家使用近期新增机制而成为 C++ 编程风格上的一大缺陷「在此，“近期”有时意味 10 年时间之内的新增机制」。似乎由于标准的颁布以及大多数用户希望利用标准库刺激各种编译器去遵循标准，同时也刺激了用户去升级到最新的编译器。

因为不同的人对于不曾修改的旧代码的重点持不同意见，所以优化一种编程语言及其编译器在本质上讲是一件困难的事情。过去我曾认为链接方面的兼容性要远远重要于源码的兼容，但是我现在对此已不是十分肯定。许多机构在组合旧代码方面并能够没有什么举措「尽管其中代码经常是支离破碎」。而这就需对编译器进行优化以便对各种变化进行统筹。

我并不热衷于利用编译器的优化来对兼容性和警告之间的变化进行钳制，但是在有些方面，它们又是必需的。不管怎样，所有 C++ 编译的缺省设置都应该完全符合 ISO C++ 标准，偏离该标准的编译器大半有点烦人。令人遗憾的是，对于那些关心交货期限胜于关心代码质量和真正移植性的人而言，则更喜欢使用一种可以通过变化多端的配置选项来与主要提供商的 bug 兼容性相一致的语言。

#### Gosling:

总体来讲，对于语言自身我们过去一直相当的保守。要求对语言进行变革的人数相当少；在 PFE 的记录中和语言变化有关的记录数量是异常的少。语言变革所产生的效用以及正在发生的变革都仅仅局限在库中，因为在创建我们自己所想要的语言方面、为了定义新类而制定的整个类系统方面、以及如何很好地封装它们方面，存在一个既定的规范，而人们所希望的各种改变也大都隶属于此种类型。

有时也存在一些不兼容的变化，其中一些更改了现存代码的含意。如果这些变动对库中的一些内容的含意进来变动，那么大量调用该内容而获得的结果都得一律进行改动。最初的 I/O 库是针对字节进行读写，这主要是因为我们没有足够的时间对其进行完善。实际上，它们现在本来应当针对字符进行读写，而不是字节；在一个字符和一个字节之间存在着相当大的差别，它们不得不过来面对国际化程序、文件封装、ISO、ANSI、UTF-7、UTF-8 等各种标准之间的转换，而所有的这一切使它变得如此复杂，其实并不须如此。所以，我们并不是改动输入流和输出流的语义，实际上是创建被称为“readers 和 writers”的新类，以便于那些愿意继续使用旧类的人们继续使用前者，而为那些愿意改动它们的人们提供了新类。

在库的协助下，大多数情况下你可以避免如此。但是在有些地方确实十分棘手。在线程方面可以充分体验到各种情况，“停止一个线程意味着什么？”“杀死一个线程又意味着什么呢？”有大量的博士论文对该课题进行论证，“thread.kill() 又意味着什么呢？”因为这一点触及到了一大堆纠缠交错而又令人棘手的问题的核心。许多系统，包括 Java 在内，对杀死一个线程的定义在很大程度上都是权宜之计且忽略了相当一部分的内容。当线程中出现 bug 的原因真正被弄清楚时，将会发现杀死一个线程的概念并不是一个十分妥当的概念。

3. 在未来的 2—5 年内，你们希望什么内容可以成为 C/C++/Java 的一部分？未来 10 年呢？又分别为什么呢？

**Ritchie:**

对 C 而言，新的 1999 年标准虽然已经颁布，但并没有真正使用，而且和前一标准相比存在着一些并非具有革命性质的变动。我认为这需要一段时间的酝酿。

**Stroustrup:**

我认为 C++ 的演化重点是在库的创建方面。我希望下一次新的标准会重点考虑如何为标准函数库提供更多的支持机制，语言的变化将主要由对这些机制的需求方面引起。

关于将新增什么库、以及如何保持它们一致连贯将会非常困难。理想的境况就是大家提交给委员会的库具有足够的通用性、相当优雅、相当高效、相当具有创新性，以便给予委员会一个良好的工作基础。当年对 STL 的需求就是这样。委员会不是很擅长于对各种事情进行统一规划。我想这也是由其性质决定的，因为它对于如何使语言更优雅和完善关注要强于对实际设计和试验的关注。由此我担心会出现一系列只关注于「微不足道的」单独问题的无关痛痒的提议。

很明显，许多人都对并发性以及不同系统「例如 GUI、数据库、操作系统、其他语言、组建模型」之间的接口感到担忧。这是最具有挑战性和创新性的领域，其中保持一致性将是主要的挑战。例如，对于来自 C++ 的文本提供一个统一的观点将是至关重要的。对于一些接口使用 C 风格的字符串，另外的使用标准库中的字符串，其他的使用那些来自不同系统的字符串含意，这种方法对于解决混乱不失为一个良方。我建议将标准库中的字符串作为其他系统链接 C++ 系统的基础。总体而言，我们对于标准库需要使用一种统一的资源管理方法。

**4. 在 C 家族语言中，是否还存在另外一种新语言的发展空间/市场需求？****Ritchie:**

很难想象出另外一种和 C 很类似而又能够完全代替它的语言。这要取决于“C 家族”的具体含义是什么。“它是否意味着使用{}？”“在声明起始的地方就定义一个类型？”

**Stroustrup:**

我认为用户社群所能得到的最好服务就是用一种能够对系统编程提供底层支持的语言。我认为 C++ 在该方面性能良好，而且也不存在要把 C 和 C++ 融合为一体的技术因素，反而倒是存在许多政治因素。提供商/承办商喜欢将各种语言以及非标准语言混杂在一起，以便宣称自己在适应性方面的优势所在，以便更好地锁定用户。另外各个组织也都在努力使自己不断更新，所以我不认为任何一种语言、非标准语言、变化会无声无息地消失。

由于没有对系统编程提供底层支持，Java 必需借助于其他语言「比如 C、C++」。相反，要想使 C、C++ 能够安全地下载到另外的机器上，我们要么借助硬件的支持「我认为这是对所有语言的选择方案」、C++ 虚拟机「是的，一些这样的机器已经出现」，要么在程序认证方面做出突破。

那么，在 C/C++/Java 以外是否有必要存在另外一种语言呢？当然更多语言都有其发展空间。问题是这些语言是否应当隶属于一个家族。另外，只有当出现了现存语言无法解决的重大问题时，才有必要去创建一种新语言。那些仅仅对用户提供了一些次要方便之处的语言或是方言，其存在仅仅是分裂了用户社群，且使那些本来为公共利益服务的资源发生了偏移。

**Gosling:**

哦，当然了。我不是很清楚你们对“隶属于这个家族的语言”的定义如何。C 家族中的所有语言都使用括号吗？

AWK 隶属于 C 家族吗？你们可能会在 C 家族语言中调用 AWK。我很乐观地认为将会出现更多的语言；与它们是否隶属于 C 家族并无关联。

在 Java 出现之前的岁月里，一个真正的灾难就是几乎全世界有关编程语言的研究都停下来了。于是 Java 出现了，当然 Java 确实很成功。我在许多大学里面所得到的最常见的评论之一就是，“噢，你将整个编程语言的研究真正地变得有章可循。”我想如果 Java 成为最后一种一闪即逝的语言，你们的评论将会是该大学的悲剧性评论。我很愿意看到一些既新颖又比较有趣的经典出现，而无论我是否参与其中。最好不要让世人等待 20 年之久。也许间隔 5 年会有利于保持一定的稳定性，因为变动一种编程语言就相当于变动一个组织的底层基础结构。在编程技巧方面它也许能够影响到你能够到达的深度，所以实际上很难进行变动。

## 5. 什么样的应用适合采用 C/C++/Java？又有哪些不适用呢？

### Stroustrup:

如果没有库的支持，采用 C++ 的许多大型应用将会变得很困难。在库的支持下，大部分会变得相当容易。对于系统组件编程方面的应用，C++ 具有内在的优势，特别是对于那些对资源有所限定的应用「例如运行时间和内存」。从该点来看，C++ 在组织比较容易维护和演化的大型程序方面提供了显著的支持。

在库的适当支持下，C++ 能够最为一门优秀的教学语言。从教学的角度讲，让学生从最新 C++ 开始而不是沉陷与那些旧风格之中是至关重要的。参考“将 C++ 当作一种新语言来学习”「该篇文章可以从我的主页上进行下载」。在教学方面，C++ 最大的优势可能就是它允许学生以那些与现实问题相一致的方法去掌握大量的编程技巧。

### Gosling:

在涉及网络的任何方面及安全性被重点关注的领域，Java 都是游刃有余。尽管确实有些目标不是设计 Java 的初衷，但是人们也以开始利用 Java 去实现这些功能。有许多人在为能够使 Java 以直接的方式来编写驱动程序方面做了许多工作，这似乎有点令人惊异，但是现在利用 Java 来编写驱动程序还是令人有几分尴尬。另外，有人正在利用 JAVA 处理运算量很大的工作……Java 的设计初衷并不是以此为关键，关于这些方面的事情确实有点复杂，然而如今有很多人都在利用 Java 做运算编程，而且发现所付出的折中也是物有所值。

## 6. 在学习以及使用方面，语言是正在变得愈来愈容易还是愈来愈难呢？随着时间的推移，在 C, C++ 及 Java 中开始显露出「而且/或是/增加」一些特性，你认为在这些语言中是否存在一些具有激进性质的用法？

### Stroustrup:

语言正在变得容易使用。但是因为我们同时正在力图取得更为艰难的目标，所以可能见不得如此。例如，带有标准库机制「例如字符串、矢量、算法」的 C++ 在解决同样问题时要远比使用 C 风格字符串、数组、C 标准库函数的 C++ 容易。

从 C 到 C++，我认为取得了一定的易用性和较强的表达能力。毕竟除了一些不是很重要的例外之处，C++ 乃是 C 的一个超集。但是从 C++ 到 Java，我看不出有这样的一种趋势，Java 试图将编程人员限制到一个单一风格的编程语言中。这就使 Java 在该领域内变得比较容易使用，但是当达到这种风格的边缘时就会导致一个拙劣的编程背景。使用 Java 容器时需要转型就是一个明显的例子。

在我看来，就风格和语言演化方面而论 C++ 和 Java 似乎有点南辕北辙。你可能希望通过对库的充分利用来消除这种分歧，但是如今的 C++ 库注重于模板的高级应用，而如今的 Java 库则注重于内部类，这似乎不是我们所希望的情景。

通常对“易用性”、“学习曲线”、“复杂度”的讨论可能会使人感到迷惑不解，因为同一个概念可以使用一种语言、一个标准库、不属于同一个标准的库，或是项目编程人员所编写的代码进行解释。作为惯例，我倾向于在语言之中或是标准库里面解决复杂的议题。把“困难”放在该处，大部分程序员可以得益于专家们所做出的坚实基础工作。而且我们也不必“重新去发明车轮”。然而，一个语言如果提供这些支持就会被认为是复杂的，而如果将这些问题遗留给单独的项目创建者则被认为是简单易用的。

#### Gosling:

在人们所关心的可用性的形式方面，我的确看到了一种趋势，而它们所获得的易用性究竟达到什么程度则是另外一回事。实际上，我并不认为易用性是一个关键的话题，因为它所包含的只不过是一些无关紧要的内涵，在一定程度上，即为一些并不必需的事情，它是一些你为了使自己更加舒适的东西，就像给椅子添加一个枕头一样。

在我看来，如今所出现的重要问题乃是我们创建的系统的复杂度正在日益增强。当然由我们所创建的系统所依赖的硬件正在按照摩尔定律的某种方式或是推论发展。而我们所创建的软件系统并没有遵循任何类似的曲线，但它们肯定是正在变得更为复杂，而且发展速度非常迅速。你如何创建大规模的复杂系统呢？存在许多我们并不真正了解的事情，而且我认为其中的大部分并不局限于平常人们对编程语言的理解。

最近我最为喜欢的一个例子就是画 Bresenham 曲线的算法。你翻开任何一年级的图形课本就会发现对如何利用 Bresenham 算法画一条直线的讲解，这大概只有 5 个句子的长度，就可以完成所以功能，非常简洁。你去看一下 Bresenham 算法的工业实现方法，就会发现即使不是上万行也是上千行的代码长度。它为什么如此庞大？为什么对于一个实际上非常简单的问题会出现这么对的复杂性？所有这些复杂性很大程度上与各自不同的实际情形有关，得去更好的理解机器底层正在运行的事情，例如高速缓存管道的长度；这个机器的高速缓存的管道是 16 个字节宽？8 个字节宽？32 个字节宽？最后你会以对不同的内层循环分别进行优化而告终。你是正在讨论 1 位像素？18 位像素？16 位像素？15 位像素？正在描绘像素吗？你正在对像素进行异或运算吗？是阿拉伯混合式「alpha-blending」像素吗？很多这样的事情可以使用运算法则的转换形式进行表述，我在 10 年前做了一个项目，其中包含一个在某种程度上可以说是 C 宏处理器的系统。你能够写出算术表达式，而且能够真正在 5 行代码内写出 Bresenham 算法以及表达转换式，且你让它与具有工业实现长度的代码相对应起来。这么做的一大优点就是你能够更好地把握其正确与否、整个测试工作以及能够将问题简单化许多。然而，这是一个编程语言问题还是一个编程环境问题呢？在某种程度上，就你能够做什么方面，你并没有改变语言的语义。你所改变的只是你如何去表达它，对语义表示方法的改变要强于对程序代码的表示。诸如此类的事情涉及众多领域。尽管我所创建的系统是被应用于大量的 Sun 图形算法，但是存在一个重大问题，那就是我不能使别人理解它所做的一切。它非常的诡秘，你可以使用它写出非常酷的代码，但是要让别人理解它作为一个新算法的创建部分和证明定理之间的关系则是异常的困难。对编程语言的大部分设计思路并不太关注“什么东西是很酷的特征？怎么样能够符合一些高调的学术标准？”它真正所关注的是“怎么样才能符合开发人员的需求？”

#### 个人喜好

##### 1. 我们每个人为什么选择软件行业都有不同的缘由，什么样的缘由使您们步入软件行业？什么使其变得如此精彩？

#### Ritchie:

我起始是对物理感兴趣，直到现在我还保持着关注物理前沿领域所发生的事情的业余爱好。在大学的一段时间以及研究生的前期，在计算机理论科学方面「图灵机、复杂度理论」花费了大量的时间。与此同时，我对真正的计算机更加着迷，同时我认为计算机的直接性也起了作用：当你写了一个程序，你可以马上看到它在做什么。所有这些事情都以有趣的方式互相交织。置身于此类的事件当中促使我迈入了软件行业。不知为何我一直不认为自己正在介入软件

业，尽管我在 1968 年已经进入，我想实际上已是事实。

**Stroustrup:**

我不是很清楚到底是什么事情将我和计算机联系起来。我认为计算机是培养科学兴趣方面一个实际有用的工具。我其实很喜欢搭建东西。在你搭建的时候，你从工具、程序/系统、用户那里获得反馈。这就等于使你的想象超越了浮于表面的现象，而且超越了个人、某个学术领域、某个组织的偏见和教条。我比较喜欢 Kristen Nygaard 把编程当作理解事情的一种方式的概念。

**Gosling:**

我自己都不清楚我的内在特质，但是我很喜欢搭积木。在某种程度上，无论是编写软件还是建造一把椅子或是调理一顿晚宴「即烹饪」，我都只是把它们仅仅当作搭积木一样。软件在该方面所具有的优势就是你能够创建复杂得令人惊讶的事情，而且可以相当快地实现它。如果我是一个钟表匠的话，我想我肯定会被创建一个充满齿轮和滑轮的东西所具有的难度挫败；然而，只要瞧瞧一块好表，你取下后盖就发现它是如此精巧。我不清楚它为什么如此精妙，但对我而言是确实确实的精妙。在软件方面，你可以放置任意多的齿轮及凸轮，且可以放置在你想放的任何地方，但是所创建的东西却在不停的变得复杂起来。在一定意义上，我最为喜欢软件的原因就是我终于花费了自己大部分时间来挑战那些复杂的东西。

## 2. 您们所用过的第一种编程语言是何语言？

**Ritchie:**

我现在不是很清晰记得当时的时间。大概在 1962 年前后，我参加了一次关于 COBOL 非授课式的研讨「由 Jean Sammet 主持」，选修了一门和编程有关的课程，该课程我通过接线板进行模拟计算机编程，并编写 Univac I 机器语言程序「因为没有汇编程序」。大概这个时候，我参观了 IBM 在剑桥的办公室，他们给了我一本关于 FORTRAN 的手册，我如获至宝的对其进行阅读。

**Stroustrup:**

我所用的第一种编程语言是 GIER 机器上的 ALGOL60。GIER 是一种丹麦产的带有 1K 内存「我想，再加上一些为了进行硬件测试的额外位」字长为 42 位的计算机。

**Gosling:**

我所用到的第一种编程语言是 FOCAL5。Focal 代表“公式积分”。它是 PDP 的一种脚本语言。我早期所写的大部分程序都是采用 Focal5 编写的。这是一种其整个编译器和运行系统都能够在 24 小时以内实现的系统。它确实是一种精巧的小型语言，其方法比 Basic 更为简单。

## 3. 什么语言，或是语言特征给您们带来了灵感？

**Stroustrup:**

除了 Simula67 以外，当时我最为喜欢的语言就是 Algol68。我认为“Algol68 with Classes”会是一种比“C with Classes”更好的语言。但是，它早已胎死腹中。

**Gosling:**

它们的作用都是显而易见。在使用 Lisp 时，对我影响最大的事情就是垃圾回收机制所带来的重大效用。使用 Simula 以及作为一个 Simula 编译器的局部维护人员才真正将我领进了对象的概念，且使我开始对对象进行思考。使用类似 Pascal 的语言使我真正开始考虑封装的概念。类似 Modula-3 的语言真正促使对类似异常机制的考虑。我曾经

使用了很多语言，其中的大部分都是曾非常有影响力的。在 Java 中你可以遇到所有这一切，而且会说，“这个来自那里，这个来自那里”。

#### 4. 您们所编写的第一个程序使什么？是在什么硬件上编写的？

**Ritchie:**

如上所言，在真正计算机上所运行的第一个的程序为 Univac I 程序。「在学习了大概一年时间之后，这个练习实际上是实现一些 APL 的运算符—Iverson 当时就在旁边」。

**Stroustrup:**

我回忆不起来了，但它肯定是在 GIER 机器上用 Algol60 编写的计算机课程 101 练习。我所编写的第一个真正的给别人使用的程序是用汇编语言为 Burroughs 办公室的计算机编写的商业程序。我以该方法赚得了我在丹麦攻读硕士学位时的大部分费用。

#### 5. 有没有一种能够很好适用于所有「几乎所有」项目开发的编程语言？如果有，是哪一种语言？如果没有，需要采取什么措施来开发该种语言？

**Ritchie:**

不存在，这是一种愚蠢的想法。

**Stroustrup:**

不存在。人们为此分歧很大而且他们的项目区别也很大，关于存在一种完美和极趋完美的语言的概念乃是那些不成熟的程序员和市场人员的痴心梦想。很自然的，每一个语言设计者都在力图强化他的语言以便更好地服务于其核心社群和扩大其影响力，但是适合于每一个人的完美语言是一个没有道理的主意，倒是存在真正的设计取舍和必须做出的折中。

**Gosling:**

我认为覆盖范围最广的语言之一就是 Java，但我是一个极带偏见的例子。如果你正在致力于那些与字符串机器模式有关的工作，Perl 可能是更好的选择。我想这些确实是我近来使用最多的语言，较老语言之中的大部分是完全包容的；使用其中的一些语言完全是历史原因而别无其他。

#### 6. 程序员们经常谈论“一种简单语言”编程的优点和缺点。这种说法对您们而言意味着什么，在您们看来，C/C++/Java 是否是一种简单语言？

**Ritchie:**

C「其他类似的语言」在某种程度上是简单的，尽管它们是非常微妙的；另外，一些类似 Pascal 语言却更是不言自明的简单。日趋明显的是，那些本不是语言核心部分、诸如库之类的方面正在变得越来越复杂。1999 年的 C 标准在库方面的扩容远远大于在语言方面的扩容；C++ STL 和其他内容正在变得庞大；AWT 以及其他和 Java 有关的事情也一样。

**Stroustrup:**

我认为关于“简单”存在三种明显的概念：容易去学，容易表达想法，和一些数学表达式有一一对应的形式。以此种方式来看，这三种语言没有一个是简单的。但是，一旦掌握，就可以使用 C 和 C++ 来表达非常复杂和高深的想法——尤其是当这些想法在受到一些现实世界资源限制的情况下进行表达时。

**Gosling:**

我作为一个语言设计者而言，近来我并没有为此而斤斤计较，真正的“简单”最终意味着我能够使 Java 任意开发者能够在自己的头脑中把握该规范。例如，该规范表示 Java 不是没人真正理解的东西，尽管大量的此类语言最终都带有大量的例外情况。如果对 C 开发人员就 UNSIGNED 进行一个测试，你很快就会发现几乎没有 C 开发者真正理解对 UNSIGNED 所进行的操作，以及 UNSIGNED 算法究竟是什么。诸如此类的事情使 C 变得复杂。我认为 Java 的语言部分很简单，至于库你们得等待一下了。

**7. 您认为从大学的计算机学科到整个业界的工程应用程序，是否遗漏了一些有助于改善软件质量的主题？****Stroustrup:**

在一些很有声望的计算机科学系，你不用编写任何代码就可以毕业。这本来是不应该成为可能的。在没有完成一个重要的编程项目的情况下，不应当有人可以获得一个计算机科学或是计算机工程学位而毕业。代码是计算的基础，对代码没有“感觉”的人将会对技巧、工具、以及创建良好系统的时间产生严重的错觉。

许多蹩脚的设计和编程都是因为对良好的代码应当由什么构成带有根深蒂固的曲解。我认为读写代码应当成为培训每一个计算机专业人员的重要基础部分，即使在对那些不用为了编程而维系生计的决策者而言也是如此。无论我们如何大谈特谈“计算科学”和“软件工程”，创建系统仍然和读、写、维护代码具有很大的实际联系，而且在可预见的未来这种关系将维持不变。

当然我不是在强调仅仅应该教育如何编码。我希望能够在理论和实际技巧之间到达平衡。教育体系的其他部分存在相反的问题。它们仅仅是在培训人们的实际技巧而不是在对它们进行教育。

**Gosling:**

第一件遗漏的事情就是忽视了大量的现实问题。人们在技术方面取得了良好的教育；他们可能缺少的是一个对在实际工程中如何运作的最佳视角：你怎么样处理人与人之间的动态关系？如果你拿到了一些大型项目，如何将其打包成块分配给不同的人，人们之间如何进行联系？当大部分程序员所做的工作大部分不是编写新代码，通常所做的大部分工作就是在已经存在的代码里面进行查找 bug 时，你会如何进行查找 bug？你如何应对一位性情古怪的经理？然而大量的教学课程实际上都是仅仅关于如何编写软件。

**8. 在 C 家族语言之外，您们最喜欢什么语言？什么因素使您们对其非常感兴趣？****Ritchie:**

我不得不佩服类似 Lisp 的语言。但是，正如在前一个关于“简单性”的问题中一样，一个异常简单的语言在实际应用中会变成一些令人担忧的东西。

**Stroustrup:**

Algol68, CLOS 和 ML 涌上心头。在每一种情况中，所吸引我的是其灵活性和一些非常优雅的代码例子。

**9. 你们最为喜欢的软件类书籍是什么？****Ritchie:**

这是个人的喜好问题，但是 Kernighan 和 Pike 「单独或是合作的」的作品都不错<sup>78</sup>。如果剔除讨论其他地方所发布的程序方面的大量笑话，以及表明上它们是如何拙劣之外，在某些方面我最喜欢较老的 Kernighan 和 Plauger

编程风格<sup>9</sup>。软件书籍我最为喜欢的是Ross Schneider所著的*Travels in Computerland*<sup>10</sup>，也是很旧，现在已经很难找到了。

**Stroustrup:**

Brooks所著的*The Mythical Man Month*<sup>11</sup>及*Object-Oriented Design*<sup>12</sup>，以及有Gamma等所著的*Design Patterns*<sup>13</sup>。

**Gosling:**

从很早开始我最为喜欢的作品就是Bill Wulf等在设计一个优化编译器方面的作品。我想这类著作可能已经绝版20年了，但是它仍不失为一部伟大的作品，如果你曾希望确切知道如何编写一个编译器，那么它是我所能说出的最好的作品。另外有Jon Bentley的*Programming Pearls*<sup>14</sup>，这也是一部精妙绝伦的图书，我还希望他能够写的更多一些。我也很喜欢Sedgewick有关算法的书籍<sup>15</sup>。

10. 如果让您们推荐包括自己的作品在内的有关C/C++/Java方面两本书，这两本数将是什么书以及为什么？

**Stroustrup:**

我将再一次推荐*Design Patterns*，可以使人们关注灵活性。作为补充，我想推荐一部注重效率的书，但是没有一本能够让我完全满意的。对许多人而言，Koenig and Moo所著的*Ruminations on C++*<sup>16</sup>在解除人们对C++是什么以及能做什么的偏见方面具有不菲价值。

**Gosling:**

我是那些时时争论Java书籍最差劲的人员之一，因为我基本上不看任何的Java书籍。通常我所读的是网上的API文档。

## 编程语言标准

1. 为编程语言制定一个正式标准「例如ISO/ANSI」是否重要呢？优点是？缺点呢？

**Ritchie:**

从某些方面来看，一个正式的标准还是必需的，特别是对于那些非正式演化而来的语言「当然有C，另外还有C++、Java」。该进程确实会将一些事情以某种方式确定下来，而且会使一些设计者原先没有预见到的一些变化和扩展拿到桌面上来进行商讨。一个书面的标准另外也给各种组织部门一定的保证，以确保其中那些对一种语言或是其他一些软件具有无比狂热的使用社群不去追逐由少数人所引领的潮流。这对于该进程增加了一定的严肃性。

同时，也存在一些缺点。因为不管当初最初设计者的设计意图是多么的明朗「或是模糊」，最终却以在该进程中参与者所提出的各种奇怪意见弄得满地涂鸦而告终。复杂性不可避免要出现妥协的结果。

**Stroustrup:**

在如今浓厚的商业氛围下，制定一个正式标准是至关重要的。如果没有一个正式标准，就无法应对那些贪婪的提供商。从根本上来讲，ISO并不是一个完备的防御体系，也不可能对所有的语言、库及工具进行标准化。然而，如果缺乏一个标准，用户将完全听凭于供货商的摆布。

拥有一个标准的最大益处在于编程语言就可以不被那些为了满足一个公司或是一小撮公司的商业利益操纵。ISO

标准来源于一个民主的进程，它代表着来自个人、公司、国家的各种声音。

而相反，其主要缺点在于真正的民主需要耗费时间，而且由于标准委员会的资源有限，所以对于一些具有创新性的东西很难做出民主裁决。但是，也不是不可能：总记得 STL 吧。

**Gosling:**

我认为制定一个正式标准将是一件美妙的事情。但是我认为大部分人对于一个正式标准是什么这个问题持有的则是非常幼稚的观点。基本上那些制定正式标准的机构就犹如一个对堆在其身上的档案进行储存的场所。他们根本没有一致性测试方面的概念。对于一个 ANSI 标准的 C 编译器而言，你所需做的仅仅就是声称你自己拥有一个 ANSI 标准 C 编译器，根本不存在测试方面的事情。我在标准化方面所遇到的艰难处境就是正式的标准化进程具有极高的政治性因素，在某种程度上这是不为大部分人所知道的。为了使 Java 符合一个正式的标准，我们已经做了大量的调整，但是政治性因素是如此的荒谬以至于在两种情况下都变得不可能了。

**Stroustrup:**

标准化则广泛的适用于 C 和从 C++。

2. 对于一种编程语言而言，拥有一个事实上的标准是否至关重要呢？其优点和缺点分别是？

**Stroustrup:**

一个事实上的标准对于其所有者及其盟友是极具益处的。诸如教授、学生和小公司的第三方社群也可以受益与此。只要拥有事实标准的公司正处在抗衡竞争对手的过程当中，那么工具和服务都会比较低廉，与此同时市场因素也会影响到科技和商业环境。该阶段过后，价格就会回升。

**Gosling:**

我认为在很大程度上事实上的标准才是真正重要的标准。那些写在标准文档中内容很难起到作用，起作用倒是那些大家真正实施的标准。事实标准让我困惑的是那些来源于一个事实标准的各种实现差异很大。在 Unix 方面就有很多称自己为 Unix，但相互之间的差异又不是稍微变动。我是一个 Linux 迷，而问题是：有这么多种的 Linux，且其差异很大，这样日子就难过了。

3. 从对 C/C++/Java 进行标准化的进程中，您们受益如何呢？

**Stroustrup:**

制定民主的标准一方面单调乏味另一方面又是必需的。那些真正想去做“正确事情”的人们及机构已经是标准化成功了一半。但是政治性问题则不可能得到解决。解决一个政治性问题的方法是在其根源找出存在的技术问题然后解决掉它。到那时，政治性问题将会自动消失。

**Gosling:**

我所接受的教育几乎都是带有政治性因素，我宁愿从来都没有接受过它。谢谢

## 21 世纪的软件工业

尽管预言不可能 100%准确，但是要做出预言还是很困难的，预言还是极具价值的。以您们看来，是什么力量在决定我们在将来要写的哪一类软件？我们编写该类软件的方法是什么呢？

**Ritchie:**

近来所发生的最为明显的变化就是解释方式的“脚本”语言的增加。基本的应用诸如 HTML、Perl、Tcl/Tk、Javascript，再如为制作 WWW 页面或是进行页面滚动的描述—图形包。

传统意义上从事计算机工作的人数在增加，但是做出来的东西是为让别人来看的工作人员的数量却是爆增。

**Stroustrup:**

未来？如今连现在都几乎很难表述。

我们已经目睹了更多强调正确性、质量、安全性的问题。我们的文明进展极大地依赖于软件，但是我们在计算机领域拥有的却是极低的专业精神。

**Gosling:**

我认为两大推动力量分别为复杂性，以及变化很大的软件运行环境。人们编写只运行于一台主机上的软件的时光一去不返。多年来，事情发展的已经渗透到各种高深领域，程序运行环境就如在一个智能卡内或是蜂窝电话，而且这些事情正在变得越来越重要。在一个开发者能仅仅关注于网络中的一台机器的情况下，我们所创建的系统就会显露出大量的点对点特征，而开发者就不得不去关注整个系统；而系统的环境则遍布许多地方，所以就会出现大量的复杂性，现在这种和运行环境有关的复杂性和多样性正在爆发。

至于我们写什么样的软件的问题，去考虑一下你正要试图解决的问题，它则是异常有趣的。当你看到一段软件，去看一下在解决一个问题时它的贡献具体到什么程度。它正在向几乎不能再小的界限发展，因为软件的其余部分都是关于：你如何处理用户接口？你如何和网络通讯？你如何应对从错误处恢复？你如何应对可靠性？你如何处理那些你实际要解决的问题的外围问题？一个有趣的应用例子是 Quicken、checkbook balancer。维持你的 checkbook 的物件非常简单。然而你看一下其周围的附带东西，它怎么样使它们都具有可靠性、便于使用、和环境进行集成等等，正是这些所谓的外围物件在向整个环节渗透。

**Stroustrup:**

通常我们仅仅是把大量的问题抛给人们。我认为从长远的观点来看，我们仅仅需要对系统的发展、部署、维持提供一个更为系统化的方法。专业精神的提高是必需的，而且这也有助于我们使用更为先进的工具和技术。现在存在一种降低语言和工具门槛的趋势，以便于所谓的“程序员”经过几个星期的“培训”及“历练”就可以开始使用它们了。我想这种趋势得务必扭转。我们需要专业的工具。

我不认为语言将变得既是解释式又是编译式的。编译与解释之间的取舍应当由具体的程序或是部分程序的实际需求来决定。

在编程方面，我想我们将会拥有另外一些新的事情，就犹如我们今天所知道的一样。将会出现一些专业的编程人员来处理这些代码而且将会使用工具来读、分析、写这些代码。然而我认为大量的软件将由那些其他领域的专家来编写或是简单重新组织个人的环境。这些非专业人士使用什么我不得而知，但是将大部分依赖于由专业认识所提供的支持性能非常好的基础功能，而且它不会被其使用者认为是任何和代码有关的东西。我怀疑它将是具有高度声明规范性及严格遵循某种规则的。

我真的希望到 21 世纪 50 年代，软件和系统能够高级得让我今天无法想象，就像 50 年代的专家很少有机会能够对当今的系统做出具有任何准确和详尽的预见。

**Gosling:**

所发现的让我对编程技术有几分沮丧的事情，就是所看到的各种各样的开发环境，它们被定位于解决各种具体问题，比如创建用户接口。然而针对那些高端开发者，比如那些实际实现各种算法的开发者，如果使用其中的一种高端集成环境，通常这些 IDE 根本对你没有什么帮助，因为它们使你深陷于非常简单的文本编辑器。当今对于高端开发者而言首选的软件开发环境本质上还是 EMACS。在这些高端用户内心，虽然这些工具已达 20 年之久，也不存在什么大的变化。人们在图形编程环境方面已经备受挫折，而且也由于这样那样的原因而逐步走向失败。

我经常感到在这些稀奇古怪的开发环境理念里面存在一些不错的想法，但是它为什么不能运作呢？人们为什么仍在使用 ASCII 文本编程呢？在 ASCII 文本的一行行编程风格以外应该还存在众多的编程风格，在 ASCII 里面，大概只有 80 字符的宽度，大部分是 7 位的 ASCII 码，以及你甚至能够在打字机上打出来的东西。但这就是现如今实际的编程情景，而且事实证明想获得突破将是一件非常困难的事情。

## 注释

- 
1. B. Kernighan and D. Ritchie. The C Programming Language, 2nd edition (Prentice Hall, 1998) ISBN 0131103709.
  2. B. Stroustrup. The Design and Evolution of C++ (Addison-Wesley, 1994) ISBN 0201543303.
  3. B. Stroustrup. The C++ Programming Language, Special Edition (Addison-Wesley, 2000) ISBN 0201700735.
  4. <http://www.research.att.com/~bs>
  5. B. Stroustrup. "Wrapping C++ Member Function Calls" (C++ Report, 12(6), June 2000).
  6. B. Stroustrup. "Learning Standard C++ as a New Language" (C/C++ Users Journal, May 1999; also in CVu 12(1) January 2000).
  7. B. Kernighan and R. Pike. The Practice of Programming (Addison-Wesley, 1999) ISBN 020161586X.
  8. B. Kernighan and R. Pike. The UNIX Programming Environment (Prentice Hall, 1984) ISBN 013937681X
  9. B. Kernighan and P.J. Plauger. The Elements of Programming Style (McGraw-Hill, 1988) ISBN 0070342075.
  10. B. Schneider. Travels in Computerland: Or, Incompatibilities and Interfaces: A Full and True Account of the Implementation of the London Stage Information Bank, ASIN 0201067374.
  11. F. Brooks. The Mythical Man-Month: Essays on Software Engineering (Addison-Wesley, 1995) ISBN 0201835959.
  12. G. Booch. Object-Oriented Analysis and Design With Applications (Addison-Wesley, 1994) ISBN 0805353402.
  13. E. Gamma, R. Helm, R. Johnson, and J. Vlissides. Design Patterns : Elements of Reusable Object-Oriented Software (Addison-Wesley, 1995) ISBN 0201633612.
  14. J. Bentley. Programming Pearls, Second Edition (Addison-Wesley, 1999) ISBN 0201657880.
  15. R. Sedgewick. Algorithms in C, 3rd edition (Addison-Wesley, 2000) ISBN 0201849372. (There are at least ten "Algorithms" books by Sedgewick, covering different scopes or specialized for different programming languages. This is one of the most recently written/updated.)
  16. A. Koenig and B. Moo. Ruminations on C++ (Addison-Wesley, 1996) ISBN 0201423391.