

掀起 C++0x 的盖头来： C++之父 Bjarne Stroustrup 访谈

译者：陈良乔

Blog: <http://imcc.blogbus.com/>

反馈: chenlq@live.com

原文: <http://www.codeguru.com/article.php/c18357>

C++0x 作为 C++的下一个国际标准，已经在业界热炒多年。但是，尽管业界对这个新标准千呼万唤，她总是不愿意过早地来到我们面前。在最近一次 CodeGuru 对 C++之父 Bjarne Stroustrup 博士的采访中，C++之父终于给我们带来了好消息——C++0x 的标准化工作已经接近尾声，C++0x 呼之欲出。下面是整个采访过程的节选，我们可以通过这个访谈，掀起 C++0x 的盖头来，了解 C++0x 的最新进展，新的特性以及未来的计划。

Danny Kalev: C++0x 的标准化过程进展如何？我们现在有多么接近这个新的 C++标准？

Bjarne Stroustrup: 我们计划在 2011 年 3 月 26 日进行最终的技术投票。虽然其后还会进行正式的国家投票以及 ISO 官僚主义的拖延，但是我十分相信，我们会在 2011 年使这个官方的标准来到我们面前。

Danny Kalev: 在所有关于 C++0x 的核心特性以及库的改进中，C++0x 为一个典型的 C++程序员带来了哪些好东西？C++0x 的哪些方面使您特别骄傲？最后，鉴于目前市面上还缺少关于 C++0x 的相关教材和资料，对于程序员们学习和使用 C++0x 的新特性您有什么建议？

Bjarne Stroustrup: C++0x 对于 C++的改进是以许多小的语言特性的改进以及部分新特性的增加的形势出现的，并不是对 C++革命性的更新。我猜想，许多改进对于大多数人而言并不是十分重要的，但是会让 C++变成一门更好的程序设计语言。但是这一点也并不会影响我的核心观点：C++0x 的改进弥散地分布在 C++语言的各个部分；它们在许多地方以多种形式改善我们的 C++代码；并不像人们通常理解的改进一样，被隔离成某一个独立的新增加的组件。更形象地说，我认为 C++0x 的改进就像我们获得了很多新的种类的砖块，这样我们可以构建很多以前无法轻松构建的建筑，并且更加容易和灵活优雅。实际上，使用 C++0x，我总是能够写出比使用 C++98 更加简单更加优雅的程序，并且，通常也会有更高的性能。

当我们首先来看几个可以让 C++ 程序员的生活更加轻松惬意的 C++0x 改进。考虑下面这段代码：

```
void f(vector<pair<string,int>>& vp)
{
    struct is_key {
        string s;
        bool operator()(const pair<string,int>&p)
        {
            return p.first == s;
        }
    };

    auto p = find_if(vp.begin(), vp.end(),
        is_key{"simple"});
// ...
}
```

这段代码看起来并没有什么激动人心的新改进，但是我要指出其中有四个小特性是 C++98 所不具备的：

1. 在 “vector<pair<string,int>>” 中，第一个 “>” 和第二个 “>” 之间并没有空格，我认为这是 C++0x 中最小的改进，但是却可以省去程序员们添加空格的繁琐。
2. 我定义了一个变量 p 但是并没有明确地指出它的数据类型，作为替代，我使用了 auto 作为其数据类型，这就意味着 “使用初始器 (initializer) 的数据类型”，所以 p 的数据类型就是 vector<pair<string,int>>::iterator。这将节省程序员编码以及调试可能出现的 Bug 的时间。这是 C++0x 最 “老” 的新特性，我在 1983 年就实现了这个特性，但是因为一些兼容性问题，这一特性一直没有被纳入 C++ 标准。
3. 局部结构体 is_key 被用作模板参数类型，也许你并没有注意到这一点，但是这样的用法在 C++98 中是非法的。
4. 最后，我使用初始器 {"simple"} 创建了一个键。在 C++98 中，我们只能够以这样的方式初始化一个变量而不能初始化一个函数参数。C++0x 通过 “{...}” 操作符，提供了一致的初始化方式。

我们还可以利用 Lambda 表达式进一步简化这个例子：

```
void f(vector<pair<string,int>>& vp)
{
    auto p = find_if(vp.begin(), vp.end(),
        [](const pair<string,int>&p)
        {
            return p.first=="simple";
        });
    // ...
}
```

Lambda 表达式是对函数对象的定义和使用的一种简化。这里，我们使用 Lambda 表达式简单地表示了 `find_if()` 算法的谓词使用 `pair` 作为参数并将其第一个元素与 “simple” 进行比较。

是的，这些主要的新特性都很好，但是 C++ 程序员们所关心的那些很重要的问题呢？

- 传统的 “threads-and-locks” 风格的系统级并行计算的类型安全得到了支持。和一个可以用于无锁（lock-free）编程的新的内存模型一起，它们将共同为 C++ 程序员们编写更加高效，更具备可移植性的并行计算程序提供强有力的支持。
- C++0x 提供了一个更高抽象层次的并行计算模型。这个并行计算模型基于异步地执行多个任务，而这些任务之间又是通过所谓的消息缓冲（message buffer）进行通信的。
- 一个新的正则表达式标准库组件
- 哈希容器
- 移动语义以及移动语义在标准库中的应用。特别地，现在我们可以以传值的方式从函数返回一个体积比较大的对象。例如：

```
vector<int> make_vec(int n)
{
    vector<int> res;
    for (int i=0; i<n; ++i)
        res[i] = rand_int(0,100000);

    return res;
}
```

标准库中的 `vector` 有一个移动构造函数（move constructor），它可以接受一个右值并简单地直接将其转换为目标对象，而不是复制容器中的所有元素来完成对象的创建。这就表示函数的返回可以通过简单的少数几次赋值完成，而不再是通过更多的，比如一百万次，逐个复制元素来完成函数的返回。这样，我们无需再使用繁琐而危险的指针，引用，内存的申请和释放等。移动语义为我们传递大体积的对象提供了一个全新的完整的解决方案。特别地，它的实现也非常简单，并且使得对于数据的操作更加富有效率，比如两个矩阵的乘法操作：

```
Matrix operator*(const Matrix&, const Matrix&);
```

我可以一直继续下去，但是那将是一个长长的列表，但是这也将我们引向了更有趣的第二个问题：人们应该如何学习和使用 C++0x 中的这些新特性？我正在写第四版的《C++ 编程语言》（4th edition of The C++ Programming Language），但是那还有很多工作要做，那将会花费超过一年的时间。我想一定有其他的技术作者正在写或者正打算写关于 C++0x 的书，但是专家们或者 C++ 的初学者想要找到比较好的书以及技术参考资料，恐怕还要等一段时间。（译注：我正在写一本全面覆盖 C++0x 新特性的 C++ 参考书《我的第一本 C++ 书》，即将由华中科技大学出版社出版，敬请期待）幸运的是，现在已经有一些关于 C++0x 的早期技术资料了，比如我的 C++0x FAQ，它提供了

很多简短的例子，以展示 C++0x 的核心特性，标准库的改建以及其它现在可以使用特性。但是，我们还需要更多的 FAQ 以及在线文档。我们还需要一些成系统地解释如何使用 C++0x 的新特性从而更好地支持 C++ 的开发的资料。基于这样的考虑，我们需要的应该是一本书。

当我在写程序的时候：使用 C++ 的原则和实践经验，并假设我正在使用的编程语言是 C++0x，但是，如果不使用 C++0x 的新特性那将是一件非常痛苦的事情。我可以十分肯定地预言，对于 C++ 的培训者和学习者，C++0x 将是上天的恩赐。C++0x 对于对于一些好的编程技术和风格提供了大量的更有力的支持。比如，现在有一个普遍一致的初始化机制，现在我们都统一使用 “{...}” 操作符完成变量的初始化工作，并且不管我们在什么地方使用 “{v}” 初始化一个变量 x，我们都会得到相同的结果。对于 C++98 中不一致的初始化形式，“=v”、“={v}”和“(v)”，这是一个非常大的改进。

```
vector<double> v = { 1,2,3,4}; // a user-defined type
double a[] = { 1,2,3,4};      // an aggregate
int f(const vector<double>&);
int x = f({1,2,3,4});
auto p = new vector<double>{1,2,3,4};
struct S { double a, b; };
S s1{1,2};                    // has no constructor
complex<double> z { 1,2,};    // has constructor
```

在我们从 C++0x 的简化中获得好处之前，也许我们会经历这样一个黑暗的时期——很多人会通过列举 C++0x 的新的语法规则或者是孔乙己式地深究 C++0x 的语法细节来展示自己的“聪明才智”。实际上，这样做是有害的。

我们不能指望人们仅仅通过阅读就能对 C++0x 编程有一个很好的理解。人们必须在开发实践中真正地使用这些新特性。幸运的是，C++0x 的很多新特性已经在很多编译器（例如，GCC 和 Microsoft Visual C++）中实现了。C++0x 不是象牙塔中的科学研究，而是真实地来到了我们身边！

Danny Kalev: 总体而言，你认为将右值引用添加到 C++0x 是值得的吗？除了性能的提升之外，一个典型的 C++ 程序员还能从右值引用中获得什么其他的好处呢？比如更简洁的设计，更简单的算法等等？

Bjarne Stroustrup: 我觉得将右值引用加入 C++0x，不仅仅是值得，而是非常值得。移动语义可以作为一个长期存在的问题——如何从函数中返回一个体积较大的数据结构——的解决方案。对于这个问题，移动语义给了我们一个显而易见的，简单而高效的答案：直接将结果从函数中移动到目标位置；不需要复制结果；不需要在内存管理上玩什么技巧；不需要使用混乱的特殊用途的内存管理方案；不需要函数的调用者预先申请内存；不需要通过额外的参数进行值的传递；不需要任何形式的垃圾回收机制。我认为这是右值引用的两个应用中的最重要的一个。它将影响我们使用 C++ 进行开发的每一个人，并且会让我们的生活变得更好。开玩笑地说，以前很多人都说“聪明的

程序员才能使用 C++”，现在，有了移动语义，不那么“聪明”的程序员也可以使用 C++了。我们可以省掉我们的聪明了。

值得注意的是，写一个有关右值引用移动的操作通常是一件非常简单的事情，它不像送火箭上天那么困难啦。

```
class Matrix {
    double* elem;    // 指向成员变量的指针
    int dim1, dim2;
public:
    Matrix(Matrix&& a)
        :dim1(a.dim1), dim2(a.dim2), elem(a.elem) // 移动数据
        { a.dim1=0; a.dim2=0; a.elem=nullptr; } // 将原来的数据清空
    // ....
};
```

这就是整个移动构造函数的完整过程：移动数据并将原来的数据清空。有了它的帮助，我们甚至可以简单而高效地返回一个 10000*10000 的矩阵。

当然，对于程序库的开发者而言这也将是非常重要的那一天：在程序库中，有很多地方可以使用移动语义以简化程序库的实现，并且在更多的地方，转发（右值引用的另外一个重要应用）将有助于程序库的设计与实现。并且，移动语义和完美转发并不是只有专家才能掌握的高深技术，每一个 C++程序员都可以使用它们来简化我们的程序，提高程序的性能。

Danny Kalev: 在一些 C++0x 新特性，诸如右值引用，Lambda 表达式，的设计中有很多困难。一些批评者也声称，C++太老了并且不够灵活。这些抱怨是否有一定的道理？会不会在将来的某一天，你决定不再扩展和改进 C++，转而使用一种新的编程语言代替？

Bjarne Stroustrup: 我更经常听到抱怨是 C++太灵活并且太大。新的语言往往是比较简单的，因为它还没有形成一个庞大的社区。所有语言都会随着时间的流逝而增长。当然，修改一门大型的，有悠久历史并已经被广泛使用的编程语言要比推倒一切重来困难得多。但是，很多新语言都会夭折。并且，对于真实世界的应用来说，这些新语言显得太过简单了。向 C++中添加新的内容是非常苦难的，对于一个新特性的建议者来说，要让这个新特性获得接受的过程通常也是漫长和痛苦的。但是，一旦这个新特性获得接受，它将会对很多人产生十分重大的影响。如果我不想影响整个世界，我完全可以通过填字游戏，写小说或者是设计一门好玩的编程语言来让自己的才智得到发挥。

当然，我也曾经梦想过设计一门比 C++更新的、更小的、更好的编程语言，但是，每当我看到这门新语言可以解决的问题，以及这门新语言可能产生的影响，我就觉得大多数通过一门新的编程语言可以解决的问题同样都可以通过改进 C++及其标准库来获得解决。就对编程世界产生积极影响——至少对于我来说——而言，看起来比较繁琐的对 C++的不断改进要比设计、实现和推广一门新的编程语言要好得多。

Danny Kalev: 关于对 Unicode 的支持, C++0x 提供了 `char_16` 和 `char_32`, 以及 `u16string` 和 `u32string` 来支持 UTF16 和 UTF32 编码的字符串。但是, 它们在标准库中的输入输出流中并没有得到支持。例如, 标准库中没有所谓的 `u16cout` 或 `u32cout`。我想知道的是, 我们该如何使用 `char16_t` 字符串并将它们输出?

Bjarne Stroustrup: 显然, 我们应该有支持 `unicode` 的输入输出流以及在标准库中的其他的扩展对 `unicode` 进行支持。标准委员会知道这样的需要, 但是没人有足够的能力和时间来实现它。因此, 不幸的是, 这是 C++ 中一个你不得不寻求第三方支持的地方。实际上有很多现有的程序库都可以很好地支持 `unicode`, 例如, 可以用于构建网络应用以及互联网应用程序的 Poco 库 (<http://pocoproject.org/index.html>)。另外, Microsoft Visual C++ 对 Unicode 也有很好的支持。

不幸的是, 我们并没有从标准库的层次上为 `unicode` 提供完整的支持, 并且我们应该记住, 大多数程序库并应该也不能够被包含在标准库中。我的 C++ 页面上有很多关于程序库, 程序库收集以及程序库列表的链接, 大约估计有超过 10000 个 C++ 程序库 (包括商业的和开源的)。但是问题的关键是, 你必须找到合适的程序库并评估它们。

Danny Kalev: 最后, 兔年已经到了, 能不能和我们分享一下你的新年愿望是什么呢?

Bjarne Stroustrup:

- 让 C++0x 成为一个正式的 ISO 标准
- 完成《C++ 编程语言》第四版的初稿
- 和我的外孙共度更多的美好时光
- 提出至少一个感兴趣的新的技术观点