

C++0x概览

Bjarne Stroustrup, 荣耀/李建忠 译

C++0x的工作已经进入了一个决定性的阶段。ISO C++委员会对C++0x的目标是使其成为“C++09”。这意味着我们要在2008年完成这个标准以便被ISO成员国批准。最后提交的标准设施将选自目前正被讨论的那些提案。为了按时完成此项工作，委员会已经停止审查新的提案并将精力集中于目前已经被讨论的那些提案上。

本文简要描述了C++0x标准化工作的指导原则，展示了一些可能的语言扩展的例子，并列出了一些被提议的新标准库设施。

说明：本文根据Bjarne Stroustrup在2005年11月19-21日在上海举行的“Modern C++ Design & Programming”技术大会上的主题发言“Direction for C++0x”整理而得。本文英文版《A Brief Look at C++0x》又于2006年1月2日发表在Artima.com网站。

指导原则

C++是一门偏向于系统编程的通用编程语言。它

- ◇ 是一个更好的C
- ◇ 支持数据抽象
- ◇ 支持面向对象编程
- ◇ 支持泛型编程

当我说“系统编程”时，我是指传统上与操作系统以及基础工具有关的那一类编程任务。包括操作系统核心、设备驱动程序、系统工具、网络应用、字处理工具、编译器、某些图形和GUI应用、数据库系统、游戏引擎、CAD/CAM、电信系统，等等。这类工作在当前的C++用户中占有主导地位。例子参见我的个人主页“Applications”单元 (<http://www.research.att.com/~bs/applications.html>)。

C++0x的目标是使以上的说法仍然成立。它并不是要消除这些编程风格（styles）（或“paradigms”，范型）之一（比方说，使C++不那么兼容于C），或者添加一种全新的“范型”。最有效的编程风格是联合使用这些技术，这也就是我们常说的“多范型编程（multi-paradigm programming）”。因此，我们可以说我们希望改进C++使其成为一门更好的多范型编程语言。

C++0x的高级目标是

- ◇ 使C++成为一门更好的系统编程语言和构建库的语言。
 - 而不是为特定子社群提供专用设施（例如数值计算或Windows风格的应用程序开发）。
- ◇ 使C++更易于教和学。
 - 通过增强的一致性、更强的保证以及针对新手的设施支持。

换句话说，在 C++98 已经很强的领域（以及一些更多的、C++98 支持的较为自然的、一般化的领域），C++0x 应该比 C++98 做得更好。对于一些专有的应用程序领域来说，例如数值计算、Windows 风格的应用程序开发、嵌入式系统编程，C++0x 应该依赖于程序库。C++ 在基本语言特性（如基于栈的对象和指针）方面所具有的效率，和在抽象机制（如类和模板）方面所具有的通用性和灵活性，使得程序库在非常广泛的应用领域都能保持它的吸引力，也因此降低了 C++ 对各种新的语言特性的需求。

我们不能为了降低 C++ 在教与学方面的难度，而去移除某些语言特性。保持 C++ 稳定性与兼容性是我们主要的考虑。因此，不管是以什么方式来移除其中任何重要的特性都是行不通的（而移除其中不重要的特性对于解决问题又没有实质性的帮助）。那么留给我们的选择恐怕只有“将规则一般化”和“添加更易于使用的特性”。两者都是我们的目标，但是后者更容易一些。例如，更好的程序库（容器与算法）可以帮助用户避免一些底层设施（例如数组与指针）带来的问题。那些能够“简化程序库的定义和应用”的语言设施（例如“concepts”与“通用初始化器列表”，下面将会谈到它们）也将有助于改善 C++0x 的易用性。

一些人可能对此持有反对意见，“不要为了新手而将 C++ 降格，适合新手的语言已经有很多了！”或者“最好的办法还是将新手变成专家！”这些人的观点并非毫无道理，但是现实是新手总比专家要多。而且许多 C++ 用户完全不必、也没有意愿成为 C++ 专家——他们是各自领域的专家（比如物理学家、图形学专家、硬件工程师），只不过他们需要使用 C++。在我个人来看，C++ 已经太过“专家友好”了，我们完全可以在花费很少的情况下为“新手们”提供更好的支持。事实上，这种支持不会损及任何 C++ 代码的性能（零成本原则依旧适用）、灵活性（我们不打算禁止任何东西）、与简洁度。相反，我们的目标是简化这些理念的表达。最后，值得指出的是，C++ 是如此之大，而且应用如此广泛，各种设计技巧可谓汗牛充栋，以至于我们很多时候也都是“新手”。

C++0x 的改进应该以这样的方式进行：结果所得语言应该更易于学和用。以下是委员会考虑的一些规则：

- ◇ 提供稳定性和兼容性（针对 C++98 而言，可能的话还有 C）
- ◇ 优先考虑库设施，其次才是语言扩展
- ◇ 只进行可以改变人们思考方式的修改
- ◇ 优先考虑一般性而非专用性
- ◇ 同时为专家和新手提供支持
- ◇ 增强类型安全性（通过为当前不安全的设施提供安全的替代品）
- ◇ 改进直接处理硬件的性能和能力
- ◇ 适应现实世界

当然，对这些思想和规则的应用与其说是一门科学不如说是一门艺术，人们对于什么才是 C++ 的自然发展以及什么才是一种新的范型有着不同的意见。C++0x 将极有可能支持可选的垃圾收集机制，并将以一个机器模型外加支持线程的标准库设施（可能还有别的）来支持并发编程。一些人也许认为这过于激进，但我并不这么认为：人们已经在 C++ 中（在垃圾收集有意义的领域）使用垃圾收集很多年了，而且几乎每个人都曾使用过线程。在这些情况下，我们需要做的不过是将现行的实践加以标准化而已。

我们之所以专注于“只进行可以改变人们思考方式的修改”，是因为这种方式可以使我们的努力获得最大的回报。每一项改变都有代价，不管是在实现方面、还是在学习等其他方面。而且，每项改变的代价并不总是直接和其带来的回报正相关。语言上的主要进步/收益并非体现在如何改进程序员编写的某一行代码上，而是体现在如何改进程序员解决问题和组织程序的方式上。面向对象程序设计和泛型程序设计改变了很多人的思考方式——这也是 C++ 语言设施支持这些风格的目的。因此，作为语言和程序库的设计

者来说，最好的做法就是把我们的时间投入到那些能够帮助人们改变思考方式的设施和技巧上。

请注意最后一条规则“适应现实世界”。一如既往，C++的目标不是创建一门“最美丽”的语言（尽管只要有可能我们都希望“美丽”），而是提供最有用的语言。这就意味着兼容性、性能、易于学习，以及与其他系统和语言的互操作性，才是应该严肃考虑的问题。

语言特性

让我们来看看使用C++0x新特性的代码的可能模样：

```
template<class T> using Vec = vector<T, My_alloc<T>>;
Vec<double> v = { 2.3, 1.2, 6.7, 4.5 };
sort(v);
for(auto p = v.begin(); p!=v.end(); ++p)
    cout << *p << endl;
```

在C++98中，除了最后一行代码外其余每一行都是不合法的，而且在C++98中我们不得不编写更多（易犯错误）的代码来完成工作。我希望无需我的解释你就可以猜测到这段代码的含义，不过我们还是逐行看一看。

```
template<class T> using Vec = vector<T, My_alloc<T>>;
```

在这里，我们定义Vec<T>作为vector<T, My_alloc<T>>的别名。换句话说，我们定义一个名为Vec的标准vector，其工作方式正如我们常用的vector那样，除了它使用我自己定义的配置器（My_alloc）而不是默认的配置器之外。C++中缺乏定义这种别名以及绑定（bind）部分而非全部模板参数的能力。按照传统，这被称为“template typedefs”，因为我们一般采用typedef来定义别名，但出于技术上的原因，我们偏向于使用using。这种语法的优势之一是，它将被定义的名字展示于易被人们发现的显著位置。还要注意另一个细节，我没有像下面这样写：

```
template<class T> using Vec = vector< T, My_alloc<T> >;
```

我们将不再需要在表示结束符的两个“>”之间添加空格。原则上这两个扩展已经被接受了。

接下来我们定义和初始化一个Vec：

```
Vec<double> v = { 2.3, 1.2, 6.7, 4.5 };
```

采用一个初始化列表来初始化用户自定义容器（vector<double, My_allocator<double>>）是一种新方式。在C++98中，我们只能将这种初始化列表语法用于聚合体（包括数组和传统的struct）。至于究竟如何实现这种语言扩展仍然在讨论中。最可能的解决方案是引入一种新型构造函数：“序列构造函数”。允许上面的例子可以运作将意味着C++更好地满足其基础设计准则之一：对用户自定义类型和内建类型的支持一样好。在C++98中，数组比vector具有记号上的优势。在C++0x中，情况将不再如此。

接下来，我们对该vector进行排序：

```
sort(v);
```

为了在STL的框架内做这件事，我们必须针对容器和迭代器对sort进行重载。例如：

```
template<Container C> // 使用 < 对容器排序
void sort(C& c);

template<Container C, Predicate Cmp> // 使用 Cmp 对容器排序
where Can_call_with<Cmp, typename C::value_type>
void sort(C& c, Cmp less);

template<Random_access_iterator Ran> // 使用 < 对序列排序
void sort(Ran first, Ran last);

template<Random_access_iterator Ran, Predicate Cmp> // 使用 Cmp 对序列排序
where Can_call_with<Cmp, typename Ran::value_type>
void sort(Ran first, Ran last, Cmp less);
```

这里演示了C++0x目前提议中最具意义的扩展部分（也是有可能被接受的部分）：**concepts**。基本上，一个concept就是一个type的type，它指定了一个type所要求的属性。在这个例子中，concept Container用于指定前两个版本的sort需要一个满足标准库容器要求的实参，where子句用于指定模板实参之间所要求的关系：即判断式（predicate）可以被应用在容器的元素类型上。有了concepts，我们就可以提供比目前好得多的错误消息，并区分带有相同数目实参的模板，例如：

```
sort(v, Case_insensitive_less()); // 容器与判断式
```

和

```
sort(v.begin(), v.end()); // 两个随机访问迭代器
```

在concepts的设计中存在的最大的困难是维持模板的灵活性，因此我们不要求模板实参适合于类层次结构或要求所有操作都能够通过虚函数进行访问（就象Java和C#的泛型所做的那样）。在这些语言的泛型中，实参的类型必须是派生自泛型定义中指定的接口（在C++中类似于接口的是抽象类）。这意味着所有的泛型实参都必须适合于某个类层次结构。这将要求部分开发人员在设计的时候就做一些不合理的预设，从而为他们强加一些不必要的约束。例如，如果你编写了一个泛型类，而我又定义了一个类，只有在我知道你指定的接口、并将我的类从该接口派生的情况下，人们才可以将我的类用作这个泛型类的实参。这种限制太过严格。

当然对于这种问题总有解决办法，但那会使代码变得复杂化。另一个问题是我们不能直接在泛型中使用内建类型。因为内建类型（例如int）并不是类，也就没有泛型中指定接口所要求的函数——这时候你必须为这些内建类型做一个包装器类，然后通过指针来间接地访问它们。另外，在泛型上的典型操作会被实现为一个虚函数调用。那样的代价可能相当高（相对于仅仅使用简单的内建操作来说，比如+或者<）。以这种方式来实现的泛型，只不过是抽象类的“语法糖”。

有了concepts之后，模板将保持它们的灵活性和性能。在委员会可以接受一个具体的concept设计之前，仍然有很多工作要做。然而，由于承诺显著更好的类型检查、更好的错误信息和更好的表达力，concepts将成为一个极有可能的扩展。它将使得我们从目前的标准容器、迭代器、和算法开始就能设计出更好的程序库接口。

最后，考虑最后一行用于输出我们的vector元素的代码：

```
for (auto p = v.begin(); p!=v.end(); ++p)
    cout << *p << endl;
```

这儿与C++98的区别在于我们不需要提及迭代器的类型：auto的含义是“从初始化器（initializer）中推导出所声明的变量的类型”。这种对auto的使用方式可以大大消除当前替代方式所导致的冗长和易出错的代码，例如：

```
for (vector< double, My_alloc<double> >::const_iterator p = v.begin(); p!=v.end(); ++p)
    cout << *p << endl;
```

这儿提到的新的语言特性的目标都在于简化泛型编程，原因在于泛型编程已经是如此流行，“使得现有语言设施受到了很大的压力”。许多“modern”的泛型编程技术接近于“write only”技术，并有孤立于其用户的危险。为了使得泛型编程成为主流（就象面向对象编程成为主流那样），我们必须使模板代码更易于阅读、编写、和使用。许多目前的用法只管编写时候的好处。但真正好的代码应该简洁（相对于它要做的事情来说）、易于检查、和易于优化（也就是高效）。这就意味着许多简单的思想可以在C++0x中简单地进行表达，并且结果代码坚定不移得高效。在C++98中前者的情况可不是这样，至少对于非常大的范围的依赖于模板的技术的情况不是如此。借助于更好的类型检查和类型信息更广泛的使用，C++代码将会变得更简短、清晰、易于维护，也更容易获得正确性。

库设施

从理想上说，我们应该尽量不修改C++语言，而集中于扩充标准库。然而，那些具有足够大的通用性的能够进入标准的库设计起来并不容易，而且一如既往，标准委员会缺乏足够的资源。我们由相对少的一组志愿者构成，并且都有“日常工作”。这就给我们能对新库进行的冒险添加了不幸的限制。另一方面，委员会很早就开始库的工作了，一个关于库的技术报告（Library TR）也在最近被投票通过了，它提供了一些对程序员来说具有直接的用处的设施：

- ◇ 哈希表（Hash Tables）
- ◇ 正则表达式（Regular Expressions）
- ◇ 通用智能指针（General Purpose Smart Pointers）
- ◇ 可扩展的随机数字设施（Extensible Random Number Facility）
- ◇ 数学专用函数（Mathematical Special Functions）

我尤其赏识能够有标准版本的正则表达式和哈希表（名为unordered_map）。此外，Library TR还为基于STL构建泛型库的人们提供了广泛的设施：

- ✧ 多态函数对象包装器 (Polymorphic Function Object Wrapper)
- ✧ Tuple类型
- ✧ Type Traits
- ✧ 增强的成员指针适配器 (Enhanced Member Pointer Adaptor)
- ✧ 引用包装器 (Reference Wrapper)
- ✧ 用于计算函数对象返回类型的统一方法 (Uniform Method for Computing Function Object Return Types)
- ✧ 增强的绑定器 (Enhanced Binder)

这儿不是详述这些库的细节或者深入讨论委员会希望提供的更多的设施的场合。如果你对此感兴趣，我建议你看看WG21站点（参见后面的“信息资源”）上的提案、库“期望列表 (wish list)”（在我的主页上），以及BOOST库（www.boost.org）。我个人希望看到更多的对应用程序构建者有着直接好处的库，例如Beman Dawes的用于操纵文件和目录的库（当前是一个BOOST库）以及一个socket库。

目前的提案列表仍然相当的保守，并不是各个地方都如我所期望的那样进取。不过，还有更多来自于委员会海量的建议中的提案正被考虑，将有更多的库或者成为C++0x标准的一部分、或者成为将来委员会的技术报告。不幸的是，资源的缺乏（时间、财力、技能、人力等）仍将继续限制我们在这个方向上的进展。悲哀的是，我无法给大家太希望得到的一个新标准库——一个标准GUI库——带来希望。GUI库对于C++标准委员会的志愿者来说是一个太大的任务，而且是一个太困难的任务，因为已经有很多（非标准、大型、有用、但受支持的）GUI库的存在。请注意，纵然它们是非标准的，主要的C++ GUI库还是有比大多数编程语言更多的用户，并且通常有更好的支持。

除了这些通用的库之外，委员会还在“Performance TR”中提呈了一个到最基层的硬件的库接口。该TR的首要目标是帮助嵌入式系统程序员，同时还驳斥了有关C++代码性能低下以及C++正变得不适合低层任务的流言蜚语。

总结

“将一个数组中所有的形状绘制出来”是面向对象编程中一个经典的例子（回想早期Simula的日子）。使用泛型编程，我们可以将其泛化，从而支持绘制任意容器（存储着Shape指针）中的每个元素。

```
template<Container C>
void draw_all(C& c)
where Usable_as<typename C::value_type,Shape*>
{
    for_each(c, mem_fun(&Shape::draw));
}
```

在C++0x中，我们希望将Container作为一个标准concept，将Usable_as作为一个标准判断式。其中for_each算法已经在C++98中有了，但是接受容器（而非一对迭代器）作为参数的版本要依赖于C++0x中的concept。其中的where子句用于支持算法来表达其对于实参的要求。就这里来说，draw_all()函数（明确）要求容器中的元素必须可以被作为（即可以隐式转换为）Shape*使用。这里的where子句通过简单要求一个Shape*容器，为我们提供了某种程度的灵活性和通用性。除了元素为Shape*的任何容器外，我们还可以使用那

些元素可以被用作Shape*的任何容器，例如list<shared_ptr<Shape*>>（其中shared_ptr将有可能成为C++0x标准库中的一个类）、或者元素类型继承自Shape*的容器，例如deque<Circle*>。

假设我们有p1、p2、p3三个点，我们可以编写如下代码来测试draw_all():

```
vector<Shape*> v = {
    new Circle(p1,20),
    new Triangle(p1,p2,p3),
    new Rectangle(p3,30,20)
};

draw_all(v);

list<shared_ptr<Shape*>> v2 = {
    new Circle(p1,20),
    new Triangle(p1,p2,p3),
    new Rectangle(p3,30,20)
};

draw_all(v2);
```

“绘制所有形状”的例子很重要，因为如果你可以很好地实现它，那么你就掌握了大多数面向对象编程中关键的东西。通过融合泛型编程（concepts与模板）、常规编程（例如独立标准库函数mem_fun()）、和简单数据抽象(mem_fun()函数返回的函数对象)，上面的代码演示了多范型编程的力量。这个简单的示例为我们开启了一扇通往许多优雅和高效的编程技巧的大门。

我希望在看完上面的例子之后，你的反应是“如此简单！”，而不是“如此聪明！如此高级！”在我看来，许多人都在聪明和高级的道路上太过投入。但设计与编程的真正目的是使用最简单的方案来完成工作，并用尽可能清晰的方式来表达。C++0x设计的目标便是更好地支持这样的简单方案。

信息资源

我的主页 (<http://www.research.att.com/~bs>) 包含有大量的有用的信息。那儿有我自己的作品（书籍、文章、访谈、FAQ等）的信息，以及我发现很有帮助作用的资源的链接（一个有意思的C++应用程序列表，一个C++编译器列表，以及到有用的库的链接（例如BOOST），等等）。关于C++0x，你可以发现：

- ◇ 语言特性和库设施的“期望列表 (wish lists)”
- ◇ 标准：IOC/IEC 14882—International Standard for Information Systems—Programming Language C++
- ◇ Performance TR: ISO/IEC PDTR 18015—Technical Report on C++ Performance
- ◇ Library TR: JTC1.22.19768 ISO/IEC TR 19768—C++ Library Extensions
- ◇ 一个到WG21 (ISO C++ 标准委员会) 站点的链接，在那儿你可以看到所有正被讨论的提案
- ◇ 一个包含我给委员会的提案（包括concept）的页面。（请记住并非所有的提案都会被接受，而且基

本上所有被接受的提案，在被接受之前都会有一些主要的变化和改进。)

感谢

感谢荣耀鼓励我对许多地方进行了澄清。同时感谢Nicholas Stroustrup, Bjorn Karlsson, 以及来自我的689班级的有益的反馈。

关于作者

Bjarne Stroustrup是C++语言的设计者和第一位实现者。现任美国德州农工大学 (Texas A&M University) 计算机科学系首席教授，在此之前他一直担任AT&T实验室的大规模程序设计研究部门（自其2002年底创建以来）的主管。另外，您可以通过访问Bjarne Stroustrup先生的个人主页<http://public.research.att.com/~bs/>来了解更多信息。