

C++ View

创刊词

创刊号 第1期

顶天立地的神话

精彩推荐

走近 C++ 之父 Bjarne Stroustrup

Traits：实用的模板新技术

深入 BCB 理解 VCL 的消息机制

目录（创刊号）

星闻

[走近 C++之父 Bjarne Stroustrup \(p5 \)](#)
C++之父评论 C++，不可不看。

抢先快报

[Qt 最新消息 \(p10 \)](#)

Trolltech 公司终于允许大家在 WINDOWS 平台下使用 Qt 进行非商业开发了。

管中窥豹

[C++的不足之处讨论系列（一）：虚拟函数 \(p11 \)](#)
同时知道长处和短处，我们才能用好 C++。

[钻穿牛角尖：求两数和 \(p17 \)](#)

求两数和，很简单啊。呵呵，试试看再说。

泛型编程与 STL

[Traits：实用的模板新技术 \(p19 \)](#)

往往听见别人说得热闹，可惜自己不知道这是什么东东。想知道的朋友，不妨看看。

设计样式与原则

[C++常用小技巧 \(p25 \)](#)

有很多小技巧，用着用着大家都这么用了。既然大家都会了，您也该会吧？

[Smart Pointer 访谈录 \(p27 \)](#)

Smart pointer 是 C++世界里的名人。不认识？天，还不快看看它的访谈录。

面向对象与应用架构

[深入 BCB 理解 VCL 的消息机制（一）\(p30 \)](#)

大名鼎鼎的 VCL，其消息机制更是以简洁出名，想钻到内部看看吗？

视点

创刊词

顶天立地的神话

以 C++ 为主要内容的刊物，难找。C++ Report 和 C/C++ Users Journal，算是北斗泰山。惜北斗已逝，C++ 世界中，又少一道风景。至于国内中文 C++ 的资料，可以说不少了，但大都集中在 VC 怎么用 Wizard 帮助写代码，BCB 又怎么拖动鼠标做出一个按钮。C++ 基础，以 C++ 为载体的世界，长期以来，却被忽略了。

C++ View 就这么创刊了，也许为了弥补这个缺憾，也许仅仅是一时的冲动，我没有考虑这份刊物会办多久，甚至连出版的时间亦没有定数。然而随心所欲，不正是 C++ 的内涵么？不管 C++ View 能否带走一片云彩，能给大家带来哪怕是一点点的帮助，我就心满意足了。

C++ View 也希望大家尊重版权。本刊鼓励大家多发表原创文章，而对于翻译外文的作品，请一定要征得原文作者的同意，切记，切记。

当您看了这第一期杂志，有任何赞扬和批评，都请您一定给我们发 email，告诉您的感想、意见，以及您的相关情况。同时也希望您多多来稿，让我们共同把 C++ View 办得更好。

编辑 王 曦

设计 / 排版 虫 虫

反馈 / 投稿 cppview@china.com

主页 <http://cppview.yeah.net>

顶天立地的神话

虫 虫

李敖说，神话有三种。神话。反攻大陆。台独。

虫虫说，神话有三次。Visual C++。Java。C#。

神话是不会变成现实的。

李敖可以发泄他的不满，可以《上山·上山·爱》。

虫虫不行，虫虫看到自己高考语文不及格的时候，只能瞪大眼睛。

当然包括看到神话的时候。神话是不会变成现实的，虫虫只在网上看到神话。

神话一。时间：Visual C++出来以后。地点：某 BBS。

Q：请问？学完 C 以后学 VC 好还是学 C++？

A：当然是学 C++ 呀！C++ 是基础呢！学好了再想学 VC 也不难了。C C++ VC。

神话二。时间：Java 出来以后。地点：某 BBS。

Q：现在都说 OO，到底是该学 C++ 还是 Java 啊？

A：Java 比 C++ 更 OO，更先进，应 C C++ VC Java。

神话三。时间：C#出来以后。地点：某 BBS。

Q：现在要学的太多了，到底学什么啊？

A：进化过程是 C C++ VC COM Java C#。

虫虫差点想写一部现代版的《二十年目睹之怪现状》，只可惜虫虫年龄不够，加上在娘胎里的日子都不够。

有人学“完”了 C，然后就应该学 C++，然后再学 VC？还要不要再学 BC？恕虫虫孤陋寡闻，只听说过 Microsoft 有个集成开发工具叫 VC，还有哪门子语言也叫 VC？

“Java 比 C++ 更 OO，更先进。”这句话不知道是 Sun 说的还是 Moon 说的。那么，请给我个理由先。

- OO 的含义是什么？
- 更 OO 的评定标准是什么？
- 越 OO 就越先进的依据是什么？

如果没有，虫虫很遗憾，Java 始终只是一门编程语言，不要指望鸡窝里飞出金凤凰。

Microsoft 不愧为 Copy&Paste 大家，坚信没有最 OO，只有更 OO，立马来个 C#，居然&果然说，这比 Java 更 OO。

于是，进化论又出新版本了，更新速度直逼北京地图的出版速度。

天，善变，不是说“天有不测风云”么？地，沉稳，泰山不是还稳稳立在那里享受五岳独尊么？人，行走天地间，不是就图个问心无愧，顶天立地么？

可惜人的天性是往上爬，所谓“人往高处走，水往低处流”。一心顶天的人不少，立志立地的人寥寥。甚至女权主义也要女性顶住半边天，或者多半边天，而不是抢占一亩三分地。如果不是担心无性生殖的成功率，可能全部天女人都占了。可见李闯王不占地的失败教训依然没能深入人心。

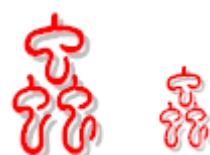
环顾四周，难道竟没有一个人要立地的吗？有，虫虫要立地，因为虫虫不会飞。李昌镐要立地，他下围棋向来喜欢取实地……

说了这么多废话，虫虫的意思就是：注意立地！当然别急着成佛，先得成人。台湾著名资讯作家侯捷亦反复强调，勿在浮砂筑高台。这也是 C++ View 创刊的初衷之一，希望更多的人关心基础。须知 IT 业日新月异，而基础的东西才是长期基本不变且有持久生命力的。虫虫亦不会否认把握最近前沿知识的重要性。顶天立地，才是好男儿&好女儿。

C++ View 关注的焦点自然是 C++，然而 C/C++ 俨然已是工业标准，应用面之广，是其他编程语言难以望其项背的。因此 C++ View 虽会理论与实践并重，但将重点放在基础上。诸如 MFC 多了一个组件，不在我们关注的范围之内。而多出的组件蕴涵了什么新技术、新想法，与原来的框架如何联系以保持框架的正常运作，这才是我们的话题。希望 C++ View 的出现，能刮起一阵“立地”风。夯实基础才是正途。速成法有是有，记得《葵花宝典》吗？记得上面的第一句话吗？欲练神功，挥刀自宫。想不想试试？

末了，虫虫还想废话两句。顶天立地是一个梦想，是一个神话。不周山不就是顶天立地的一根支柱么？然而水神共工被火神祝融打败后，一气之下就把它撞断了，还连累女娲来炼石补天。莫非即使成为中流砥柱，也要由于成为某些“神”的发泄对象而夭折？

虫虫不知道，虫虫但愿这只是一个神话，而神话是不会成为现实的。



走近 C++ 之父 Bjarne Stroustrup

myan



编者按：很荣幸，本文中翻译的内容及照片均得到了这位 C++ 之父 Bjarne Stroustrup 博士的允许，也感谢这位前辈的对本刊祝福“Good Luck with your magazine”。博士的主页地址是 <http://www.research.att.com/~bs/>。

Bjarne Stroustrup 博士，1950 年出生于丹麦，先后毕业于丹麦阿鲁斯大学和英国剑桥大学，AT&T 大规模程序设计研究部门负责人，AT&T、贝尔实验室和 ACM 成员。

(编者注：Stroustrup 博士的简历可参考 <http://www.research.att.com/~bs/bio.html>)

1979 年，B. S 开始开发一种语言，当时称为“C with Class”，后来演化为 C++。1998 年，ANSI/ISO C++ 标准建立，同年，B. S 推出了其经典著作 The C++ Programming Language 的第三版。C++ 的标准化标志着 B. S 博士倾 20 年心血的伟大构想终于实现。但计算机技术的发展一日千里，就在几年前人们还猜想 C++ 最终将一统天下，然而随着 Internet 的爆炸性增长，类似 Java、C# 等新的、现代感十足的语言咄咄逼人，各种 Script 语言更是如雨后春笋纷纷涌现。在这种情况下，人们不禁有些惶恐不安。C++ 是不是已经过时了呢？其前景如何？标准 C++ 有怎样的意义？应该如何学习？我们不妨看看 B. S 对这些问题的思考。以下文字是译者从 Stroustrup 主页上发表的 FAQ 中精选出来的。(原文可参考 http://www.research.att.com/~bs/bs_faq.html 及 http://www.research.att.com/~bs/bs_faq2.html)

请谈谈 C++ 书。

没有，也不可能有一本书对于所有人来说都是最好的。不过对于那些真正的程序员来说，如果他喜欢从“经典风格”的书中间学习一些新的概念和技术，我推荐我的 The C++ Programming Language, 1998 年的第三版和特别版。那本书讲的是纯而又纯的 C++，完全独立于平台和库（当然得讲到标准库）。该书面向那些有一定经验的程序员，帮助他们掌握 C++，但不适合毫无经验的初学者入门，也不适合那些临时程序员品尝 C++ 快餐。所以这本书的重点在于概念和技术，而且在完整性和精确性上下了不少功夫。如果你想知道为什么 C++ 会变成今天的模样，我的另一本书 The Design and Evolution of C++ 能给你满意的答案。理解设计的原则和限制能帮助你写出更好的程序。

<http://www.accu.com/> 是最好的书评网站之一，很多有经验的程序员在此仗义执言，不妨去看看。

(编者注：The C++ Programming Language 的特别版比第三版多了两篇附录：本地化 Locales 和标准库的异常安全 Standard-Library Exception Safety。这两篇附录可在 Stroustrup 博士的主页上下载。)

学习 C++要花多长时间？

这要看你说的“学习”是什么意思了。如果你是一个 Pascal 程序员，你应该能很快地使你的 C++ 水平达到与 Pascal 相近的程度；而如果你是一个 C 程序员，一天之内你就能学会使用 C++ 进行更出色的 C 风格编程。另一方面，如果你想完全掌握 C++ 的主要机制，例如数据抽象，面向对象编程，通用编程，面向对象设计等等，而此前又对这些东西不很熟悉的话，花上个一两年是不足为奇的。那么是不是说这就是学习 C++ 所需要的时间呢？也许再翻一番，我想打算成为更出色的设计师和程序员最起码也要这么长的时间。如果学习一种新的语言不能使我们的工作和思想方式发生深刻变革，那又何苦来哉？跟成为一个钢琴家或者熟练掌握一门外语相比，学习一种新的、不同的语言和编程风格还算是简单的。

了解 C 是学习 C++ 的先决条件吗？

否！C++ 中与 C 相近的子集其实比 C 语言本身要好学，类型方面的错误会少一些，也不像 C 那样绕圈子，还有更好的支持库。所以应该从这个子集开始学习 C++。

要想成为真正的 OO 程序员，我是不是得先学习 Smalltalk？

否。如果你想学 Smalltalk，尽管去学。这种语言很有趣，而且学习新东西总是一个好主意。但是 Smalltalk 不是 C++，而且把 Smalltalk 的编程风格用在 C++ 里不会有好结果。如果你想成为一个出色的 C++ 程序员，而且也没有几个月的时间百无聊赖，请你集中力量学好 C++ 以及其背后的思想。

我如何开始学习 C++？

这取决于你的基础和学习动机。如果你是个初学者，我想你最好找个有经验的程序员来帮助你，要不然你在学习和实践中不可避免的犯下的种种错误会大大地打击你的积极性。另外，即使你的编译器配备了充足的文档资料，一本 C++ 书籍也永远是必不可少的，毕竟文档资料不是学习编程思想的好教材。

选择书籍时，务必注意该书是不是从一开始就讲授标准 C++，并且矢志不渝地使用标准库机制。例如，从输入中读取一个字符串应该是这样的：

```
string s; // Standard C++ style  
cin >> s;
```

而不是这样的：

```
char s[MAX]; /* Standard C style */  
scanf ("%s", s);
```

去看看那些扎实的 C++ 程序员们推荐的书吧。记住，没有哪本书对所有人来说都是最好的。

另外，要写地道的 C++ 程序，而避免用 C++ 的语法写传统风格的程序，新瓶装旧酒没多大意义。（遗憾的是，目前在市面上的中文 C++ 教材中，符合 B. S 的这个标准的可以说一本都没有，大家只好到网上找一些英文的资料来学习了。——译者）

怎样改进我的 C++ 程序？

不说。这取决于你是怎么使用该语言的。大多数人低估了抽象类和模板的价值，反过来却肆无忌惮地使用造型机制(cast)和宏。这方面可以看看我的文章和书。抽象类和和模板的作用当然是提供一种方便的手段建构单根的类层次或者重用函数，但更重要的是，它们作为接口提供了简洁的、逻辑性的服务表示机制。

语言的选择是不是很重要？

是，但也别指望奇迹。很多人似乎相信某一种语言能够解决他们在系统开发中遇到的几乎所有问题，他们不断地去寻找完美的编程语言，然后一次次的失败，一次次的沮丧。另外一些人则将编程语言贬为无关紧要的细节，把大把大把的银子放在开发流程和设计方法上，他们永远都在用着 COBOL, C 和一些专有语言。一种优秀的语言，例如 C++，能帮助设计者和程序员做很多事情，而其能力和缺陷又能够被清楚地了解和对待。

ANSI/ISO 标准委员会是不是糟蹋了 C++？

当然不是！他们（我们）的工作很出色。你可以在一些细节上找些歪理来挑刺，但我个人对于这种语言以及新的标准库可是欣欣然。ISO C++较之 C++的以前版本更出色更有条理。相对于标准化过程刚刚开始之初，你今天可以写出更优雅、更易于维护的 C++程序。新的标准库也是一份真正的大礼。由于标准库提供了 strings, lists, vectors, maps 以及作用于其上的基本算法，使用 C++的方式已经发生了巨大的变化。

你现在有没有想删除一些 C++ 特性？

没有，真的。问这些问题的人大概是希望我回答下面特性中的一个：多继承、异常、模板和 RTTI。但是没有它们，C++就是不完整的。在过去的 N 年中，我已经反复考虑过它们的设计，并且与标准委员会一起改进了其细节，但是没有一个能被去掉又不引起大地震。

从语言设计的角度讲，我最不喜欢的部分是与 C 兼容的那个子集，但又不能把它去掉，因为那样对于在现实世界里工作的程序员们来说伤害太大了。C++与 C 兼容，这是一项关键的设计决策，绝对不是一个叫卖的噱头。兼容性的实现和维护是十分困难的，但确实使程序员们至今受益良多。但是现在，C++已经有了新的特性，程序员们可以从麻烦多多的 C 风格中解脱出来。例如，使用标准库里的容器类，象 vector, list, map, string 等等，可以避免与底层的指针操作技巧混战不休。

如果不和 C 兼容，你所创造的语言是不是就会是 Java？

不是，差得远。如果人们非要拿 C++ 和 Java 来作比较，我建议他们去阅读 The Design and Evolution of C++，看看 C++ 为什么是今天这个样子，用我在设计 C++ 时遵从的原则来检验这两种语言。这些原则与 SUN 的 Java 开发小组所持的理念显然是不同的。除了表面语法的相似性之外，C++ 与 Java

是截然不同的语言。在很多方面，Java 更像 Smalltalk（译者按：我学习 Java 时用的是 Sun 的培训教材，里面清楚地写道：Java 在设计上采用了与 C++ 相似的语法，与 Smalltalk 相似的语义。所以可以说 Java 与 C++ 是貌合神离，与 Smalltalk 才是心有灵犀）。Java 语言相对简单，这部分是一种错觉，部分是因为这种语言还不完整。随着时间的推移，Java 在体积和复杂程度上都会大大增长。在体积上它会增长两到三倍，而且会出现一些实现相关的扩展或者库。这是一条每个成功的商业语言都必须走过的发展之路。随便分析一种你认为在很大范围内取得了成功的语言，我知道肯定是有例外者，而且实际上这非常有道理。

上边这段话是在 Java 1.1 推出之前写的。我确信 Java 需要类似模板的机制，并且需要增强对于固有类型的支持。简单地说，就是为了基本的完整性也应该做这些工作。另外还需要做很多小的改动，大部分是扩展。1998 年秋，我从 James Gosling（Java 语言的创始人——译者）那里得到一份建议书，说是要在 Java 中增加固有类型、操作符重载以及数学计算支持。还有一篇论文，是数学分析领域的世界级大师，伯克利大学的 W. Kahan 教授所写的 How Java's Floating-Point Hurts Everyone Everywhere（“且看 Java 的浮点运算如何危害了普天下的芸芸众生”——译者），揭露了 Java 的一些秘密。

我发现在电视和出版物中关于 Java 的鼓吹是不准确的，而且气势汹汹，让人讨厌。大肆叫嚣凡是 Java 的代码都是垃圾，这是对程序员的侮辱；建议把所有的保留代码都用 Java 重写，这是丧心病狂，既不现实也不负责任。Sun 和他的追随者似乎觉得为了对付微软罪恶的“帝国时代”，就必须如此自吹自擂。但是侮辱和欺诈只会把那些喜欢使用不同编程语言的程序员逼到微软阵营里去。

Java 并非平台无关，它本身就是平台。跟 Windows 一样，它也是一个专有的商业平台。也就是说，你可以为 Windows/Intel 编写代码，也可以为 Java/JVM 编写代码，在任何一种情况下，你都是在为一个属于某个公司的平台写代码，这些代码都是与该公司的商业利益扯在一起的。当然你可以使用任何一种语言，结合操作系统的机制来编写可供 JVM 执行的程序，但是 JVM 之类的东西是强烈地偏向于 Java 语言的。它一点也不像是通用的、公平的、语言中立的 VM/OS。

私下里，我会坚持使用可移植的 C++ 作大部分工作，用不同的语言作余下的工作。

（“Java is not platform-independent, it is the platform”，B. S 的这句评语对于 C++ 用户有着很大的影响，译者在国外的几个新闻组里看到，有些 C++ 高手甚至把这句话作为自己的签名档，以表明对 Java 的态度和誓死捍卫 C++ 的决心。实际上有很多程序员不光是把自己喜爱的语言当成一种工具，更当成一种信仰。——译者）

您怎么看待 C# 语言？

就 C# 语言本身我没什么好说的。想让我相信这个世界还需要另外一个专有的语言可不是一件容易的事，而且这个语言还是专门针对某一个专有操作系统的，这就更让我难以接受。直截了当地说，我不是一个专有语言的痴迷者，而是一个开放的正式标准的拥护者。

在做重大项目时，您是不是真的推荐 Ada，而不是 C++？

当然不是。我不知道这是谁传出来的谣言，肯定是一个 Ada 信徒，要么是过分狂热，要么是不怀好意。

你愿不愿意将 C++ 与别的语言比较？

抱歉，我不愿意。你可以在 The Design and Evolution of C++ 的介绍性文字里找到原因。

有不少书评家邀请我把 C++ 与其它的语言相比，我已经决定不做此类事情。在此我想重申一个我很久以来一直强调的观点：语言之间的比较没什么意义，更不公平。主流语言之间的合理比较要耗费很大的精力，多数人不会愿意付出这么大的代价。另外还需要在广泛的应用领域有充分经验，保持一种不偏不倚、客观独立的立场，有着公正无私的信念。我没时间，而且作为 C++ 的创造者，在公正无私这一点上我永远不会获得完全的信任。

人们试图把各种语言拿来比较长短，有些现象我已经一次又一次地注意到，坦率地说我感到担忧。作者们尽力表现的公正无私，但是最终都是无可救药地偏向于某一种特定的应用程序，某一种特定的编程风格，或者某一种特定的程序员文化。更糟的是，当某一种语言明显地比另一种语言更出名时，一些不易察觉的偷梁换柱就开始了：比较有名的语言中的缺陷被有意淡化，而且被拐弯抹角地加以掩饰；而同样的缺陷在不那么出名的语言里就被描述为致命硬伤。类似的，有关比较出名的语言的技术资料经常更新，而不太出名的语言的技术资料往往是几年以前的，试问这种比较有何公正性和意义可言？所以我对于 C++ 之外的语言的评论严格限制在一般性的特别特定的范畴里。

换言之，我认为 C++ 是大多数人开发大部分应用程序时的最佳选择。

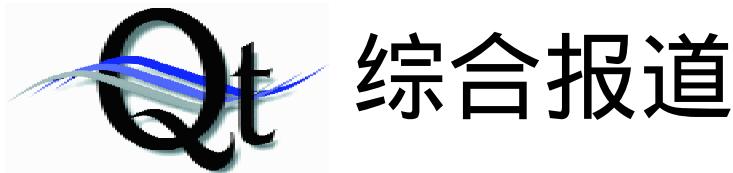
别人可是经常拿他们的语言与 C++ 比来比去，这让你感到不自在了吗？

当这些比较不完整或者出于商业目的时，我确实感觉不爽。那些散布最广的比较性评论大多是由某种语言，比方说 Z 语言的拥护者发表的，其目的是为了证明 Z 比其它的语言好。由于 C++ 被广泛地使用，所以 C++ 通常成了黑名单上的头一个名字。通常，这类文章被夹在 Z 语言的供货商提供的产品之中，成了其市场竞争的一个手段。令人震惊的是，相当多的此类评论引用那些在开发 Z 语言的公司中工作的雇员的文章，而这些经不起考验文章无非是想证明 Z 是最好的。特别是在这些比较中确实有一些零零散散的事实，（所以更具欺骗性——译者）毕竟没有一种语言在任何情况下都是最好的。C++ 当然不完美，不过请注意，特意选择出来的事实虽然好像正确，但有时是完全的误导。

以后再看到语言比较方面的文章时，请留心是谁写的，他的表述是不是以事实为依据，以公正为准绳，特别是评判的标准是不是对于所引述的每一种语言来说都公平合理。这可不容易做到。

在做小项目时，C 优于 C++ 吗？

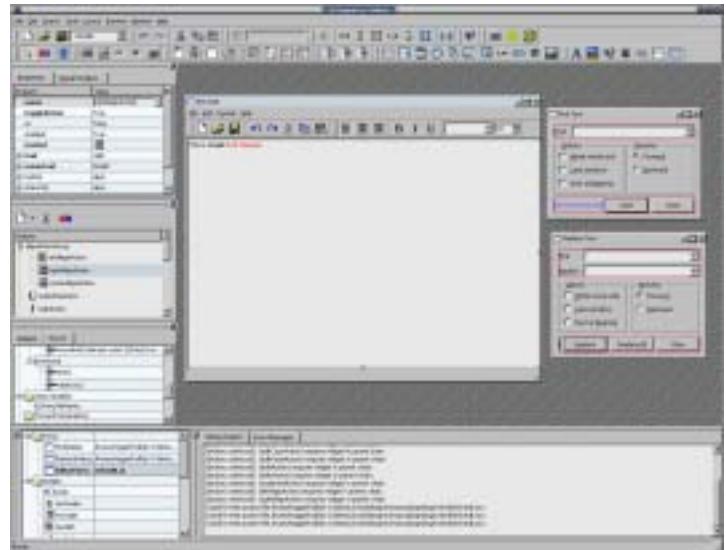
我认为非也。除了由于缺乏好的 C++ 编译器而导致的问题之外，我从没有看到哪个项目用 C 会比用 C++ 更合适。（不过现在 C++ 编译器导致的问题还是不可忽略的，当你看到同样功能的 C++ 程序可执行代码体积比 C 大一倍而且速度慢得多时，会对此有所感触的。——译者）



虫 虫

Qt , 这个 Open Source 世界著名的 C++跨平台类库 , 伴随着 Linux 和 KDE 成长。

- 1992 年 , Qt 的设计团队成立 ;**
- 1994 年 , Trolltech 公司在挪威成立 ;**
- 1995 年 , Qt 出现第一个商业化版本 ;**
- Qt 的功能亦日渐强大。
- 2001-5-18 , Qt 3.0 beta 1 发布 , 其特色是**
 - 提供数据库访问
 - Qt Designer 功能大大提高
 - Qt Linguist 首次正式亮相
 - 国际字符显示功能增强
 - 新的组件模型
 - 支持多个监视器
 - 更多激动人心功能



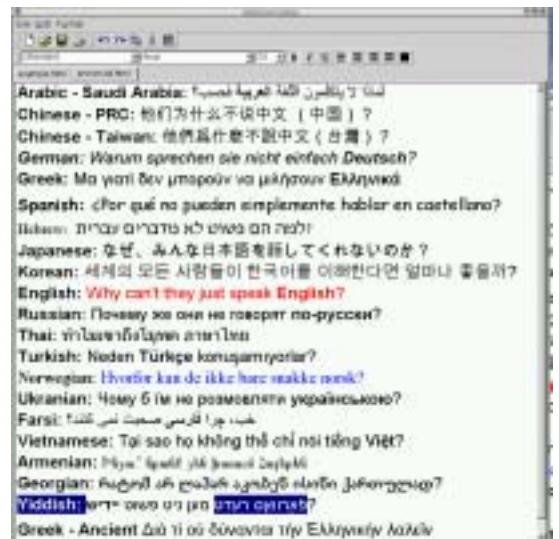
商业化的模式在 Open Source 世界绝对是背叛 , 迫于压力 , 也因为 GNOME/Gtk 的竞争 , Qt 先是以 QPL 方式开放 , 再到 Open Source for Unix/Linux 。不过长久以来 , Qt/Windows 却至多可下载 30 天评估版 , 这对广大 Windows 开发者来说 , 不能不说是一个遗憾。

2001-6-26 , Trolltech 终于开放了 Qt/Windows 非商业许可 (Non-Commercial License) 的版本。

所谓非商业许可 , 简单地说 , 就是

- 可用于评估 ;
- 可用于个人编写自由软件 , 但源代码必须公开 ;
- 如需编写商业程序或机构用途 , 必须购买商业许可。

Qt 组作为 Trolltech 公司的旗舰产品 , 提供可高度重用的面向对象组件 , 功能众多的 API 支持 , 高性能的 GUI 构件 , 提供国际化支持 , 并支持 OpenGL 。更令人激动的是 , Qt 的移植性非常好 , 在 Microsoft Windows 95/98/NT/2000 、 MacOS X 、 Linux 、 Solaris 、 HP-UX 、 Tru64 (Digital UNIX) 、 Irix 、 FreeBSD 、 BSD/OS 、 SCO 和 AIX 诸多平台上表现出色。



欲了解更多详细的信息 , 请访问 Trolltech 公司的主页 www.trolltech.com 。

C++的不足之处讨论系列（一）：虚拟函数

Ian Joyner
cber 译

以下文章翻译自 Ian Joyner 所著的

C++?? A Critique of C++ and Programming and Language Trends of the 1990s 3/E 【 Ian Joyner 1996】

原著版权属于 Ian Joyner ,

征得 Ian Joyner 本人的同意 , 我得以将该文翻译成中文。因此 , 本文的中文版权应该属于我;-)

该文章的英文及中文版本都用于非商业用途 , 你可以随意地复制和转贴它。不过最好在转贴它时加上我的这段声明。

如有人或机构想要出版该文 , 请最好联系原著版权所有人及我。

该篇文章已经包含在 Ian Joyner 所写的 Objects Unencapsulated 一书中 (目前已经有了日文的翻译版本) , 该书的介绍可参见于 :

http://www.prenhall.com/allbooks/ptra_0130142697.html

<http://efsa.sourceforge.net/cgi-bin/view/Main/ObjectsUnencapsulated>

<http://www.accu.org/bookreviews/public/reviews/o/o002284.htm>

Ian Joyner 的联系方式 : i.joyner@acm.org

我的联系方式 : cber@email.com.cn

前言 :【译者所写的】

要想彻底的掌握一种语言 , 不但需要知道它的长处有哪些 , 而且需要知道它的不足之处又有哪些。这样我们才能用好这门语言 , 也才能说我们自己掌握了这门语言。

在所有对 C++ 的批评中 , 虚拟函数这一部分是最复杂的。这主要是由于 C++ 中复杂的机制所引起的。虽然本篇文章认为多态 (polymorphism) 是实现面向对象编程 (OOP) 的关键特性 , 但还是请你不要对此观点 (即虚拟函数机制是 C++ 中的一大败笔) 感到有什么不安 , 继续看下去 , 如果你仅仅想知道一个大概的话 , 那么你也可以跳过此节。【译者注 : 建议大家还是看看这节会比较好】

在 C++ 中 , 当子类改写 / 重定义 (override/redefine) 了在父类中定义了的函数时 , 关键字 virtual 使得该函数具有了多态性 , 但是 virtual 关键字也并不是必不可少的 (只要在父类中被定义一次就行了) 。编译器通过产生动态分配 (dynamic dispatch) 的方式来实现真正的多态函数调用。

这样 , 在 C++ 中 , 问题就产生了 : 如果设计父类的人员不能预见到子类可能会改写哪个函数 , 那么子类就不能使得这个函数具有多态性。这对于 C++ 来说是一个很严重的缺陷 , 因为它减少了软件组件 (software components) 的弹性 (flexibility) , 从而使得写出可重用及可扩展的函数库也变得困难起来。

C++ 同时也允许函数的重载 (overload) , 在这种情况下 , 编译器通过传入的参数来进行正确的函数调用。在函数调用时所引用的实参类型必须吻合被重载的函数组 (overloaded functions) 中某一个函数的形参类型。重载函数与重写函数 (具有多态性的函数) 的不同之处在于 : 重载函

数的调用是在编译期间就被决定了，而重写函数的调用则是在运行期间被决定的。

当一个父类被设计出来时，程序员只能猜测子类可能会重载/重写哪个函数。子类可以随时重载任何一个函数，但这种机制并不是多态。为了实现多态，设计父类的程序员必须指定一个函数为 `virtual`，这样会告诉编译器在类的跳转表（class jump table）【译者窃以为是 vtable，即虚拟函数入口表】中建立一个分发入口。于是，对于决定什么事情是由编译器自动完成，或是由其他语言的编译器自动完成这个重任就放到了程序员的肩上。这些都是从最初的 C++ 的实现中继承下来的，而和一些特定的编译器及联结器无关。

对于重写，我们有着三种不同的选择，分别对应于：“千万别”，“可以”及“一定要”重写：

- 1、重写一个函数是被禁止的。子类必须使用已有的函数；
- 2、函数可以被重写。子类可以使用已有的函数，也可以使用自己写的函数，前提是这个函数必须遵循最初界面的定义，而且实现的功能尽可能的少及完善；
- 3、函数是一个抽象的函数。对于该函数没有提供任何的实现，每个子类都必须提供其各自的实现。

父类的设计者必须要决定 1 和 3 中的函数，而子类的设计者只需要考虑 2 就行了。对于这些选择，程序语言必须要提供直接的语法支持。

选项 1

C++ 并不能禁止在子类中重写一个函数。即使是被声明为 `private virtual` 的函数也可以被重写。【Sakkinen92】中指出了即使在通过其他方法都不能访问到 `private virtual` 函数，子类也可以对其进行重写。【译者注：Sakkinen92 我也没看过，但经我简单的测试，确实可以在子类中重写父类中的 `private virtual` 函数】

实现这种选择的唯一方法就是不要使用虚拟函数，但是这样的话，函数就等于整个被替换掉了。首先，函数可能会在无意中被子类的函数给替换掉。在同一个 scope 中重新宣告一个函数将会导致名字冲突（name clash）；编译器将会就此报告出一个“duplicate declaration”的语法错误。允许两个拥有同名的实体存在于同一个 scope 中将会导致语义的二义性（ambiguity）及其他问题（可参见于 name overloading 这节）。

下面的例子阐明了第二个问题：

```
class A
{
public:
    void nonvirt();
    virtual void virt();
};

class B : public A
{
```

```

public:
void nonvirt();
void virt();
};

A a;
B b;
A *ap = &b;
B *bp = &b;

bp->nonvirt();           //calls B::nonvirt as you would expect
ap->nonvirt();           //calls A::nonvirt even though this object is of type B
ap->virt();               //calls B::virt, the correct version of the routine for B objects

```

在这个例子里，`B` 扩展或替换掉了 `A` 中的函数。`B::nonvirt` 是应该被 `B` 的对象调用的函数。在此处我们必须指出，C++ 给客户端程序员（即使用我们这套继承体系架构的程序员）足够的弹性来调用 `A::nonvirt` 或是 `B::nonvirt`，但我们也提供一种更简单，更直接的方式：提供给 `A::nonvirt` 和 `B::nonvirt` 不同的名字。这可以使得程序员能够正确地，显式地调用想要调用的函数，而不是陷入了上面的那种晦涩的，容易导致错误的陷阱中去。具体方法如下：

```

class B: public A
{
public:
void b_nonvirt();
void virt();
}
B b;
B *bp = &b;
bp->nonvirt();           //calls A::nonvirt
bp->b_nonvirt();         //calls B::b_nonvirt

```

现在，`B` 的设计者就可以直接的操纵 `B` 的接口了。程序要求 `B` 的客户端（即调用 `B` 的代码）能够同时调用 `A::nonvirt` 和 `B::nonvirt`，这点我们也做到了。就 Object-Oriented Design(OOD) 来说，这是一个不错的做法，因为它提供了健壮的接口定义（strongly defined interface）【译者认为：即不会引起调用歧义的接口】。C++ 允许客户端程序员在类的接口处卖弄他们的技巧，借以对类进行扩展。在上例中所出现的就是设计 `B` 的程序员不能阻止其他程序员调用 `A::nonvirt`。类 `B` 的对象拥有它们自己的 `nonvirt`，但是即便如此，`B` 的设计者也不能保证通过 `B` 的接口就一定能调用到正确版本的 `nonvirt`。

C++同样不能阻止系统中对其他处的改动不会影响到 B。假设我们需要写一个类 C，在 C 中我们要求 nonvirt 是一个虚拟的函数。于是我们就必须回到 A 中将 nonvirt 改为虚拟的。但这又将使得我们对于 B::nonvirt 所玩弄的技巧又失去了作用（想想看，为什么:D）。对于 C 需要一个 virtual 的需求（将已有的 nonvirtual 改为 virtual）使得我们改变了父类，这又使得所有从父类继承下来的子类也相应地有了改变。这已经违背了 OOP 拥有低耦合的类的理由，新的需求，改动应该只产生局部的影响，而不是改变系统中其他地方，从而潜在地破坏了系统的已有部分。

另一个问题是，同样的一条语句必须一直保持着同样的语义。例如：对于诸如 a->f() 这样的多态性语句的解释，系统调用的是由最符合 a 所真正指向类型的那个 f()，而不管对象的类型到底是 A，还是 A 的子类。然而，对于 C++ 的程序员来说，他们必须要清楚地了解当 f() 被定义成 virtual 或是 non-virtual 时，a->f() 的真正涵义。所以，语句 a->f() 不能独立于其实现，而且隐藏的实现原理也不是一成不变的。对于 f() 的宣告的一次改变将会相应地改变调用它时的语义。与实现独立意味着对于实现的改变不会改变语句的语义，或是执行的语义。

如果在宣告中的改变导致相应的语义的改变，编译器应该能检测到错误的产生。程序员应该在宣告被改变的情况下保持语义的不变。这反映了软件开发中的动态特性，在其中你将能发现程序文本的永久改变。

其他另一个与 a->f() 相应的，语义不能被保持不变的例子是 构造函数（可参考于 C++ ARM, section 10.9c, p 232），而 Eiffel 和 Java 则不存在这样的问题。它们中所采用的机制简单而又清晰，不会导致 C++ 中所产生的那些令人吃惊的现象。在 Java 中，所有的一起都是虚拟的，为了让一个方法【译者注：对应于 C++ 的函数】不能被重写，我们可以用 final 修饰符来修饰这个方法。

Eiffel 允许程序员指定一个函数为 frozen，在这种情况下，这个函数就不能在子类中被重写。

选项 2

是使用现有的函数还是重写一个，这应该是由撰写子类的程序员所决定的。在 C++ 中，要想拥有这种能力则必须在父类中指定为 virtual。对于 OOD 来说，你所决定不想作的与你所决定想作的同样重要，你的决定应该是越迟下越好。这种策略可以避免错误在系统前期就被包含进去。你作决定越早，你就越有可能被以后所证明是错误的假设所包围；或是你所作的假设在一种情况下是正确的，然而在另一种情况下却会出错，从而使得你所写出来的软件比较脆弱，不具有重用性（ reusable ）【译者注：软件的可重用性对于软件来说是一个很重要的特性，具体可以参考《Object-Oriented Software Construct》中对于软件的外部特性的叙述，P7, Reusability, Chapter 1.2 A REVIEW OF EXTERNAL FACTORS】。

C++ 要求我们在父类中就要指定可能的多态性（这可以通过 virtual 来指定），当然我们也可以在继承链中的中间的类导入 virtual 机制，从而预先判断某个函数是否可以在子类中被重定义。这种做法将导致问题的出现：如那些并非真正多态的函数（not actually polymorphic）也必须通过效率较低的 table 技术来被调用，而不像直接调用那个函数来的高效【译者注：在文章的上下文中并没有出现 not actually polymorphic 特性的确切定义，根据我的理解，应该是声明为 polymorphic，

而实际上的动作并没能体现 polymorphic 这样的一种特性】。虽然这样做并不会引起大量的花费 (overhead), 但我们也知道, 在 OO 程序中经常会出现使用大量的、短小的、目标单一明确的函数, 如果将所有这些都累计下来, 也会导致一个相当可观的花费。C++中的政策是这样的: 需要被重定义的函数必须被声明为 virtual。糟糕的是, C++同时也说了, non-virtual 函数不能被重定义, 这使得设计使用子类的程序员就无法对于这些函数拥有自己的控制权。【译者注: 原作中此句显得有待推敲, 原文是这样写的: it says that non-virtual routines cannot be redefined, 我猜测作者想表达的意思应该是: If you have defined a non-virtual routine in base, then it cannot be virtual in the base whether you redefined it as virtual in descendant.】

Rumbaugh 等人对于 C++中的虚拟机制的批评如下: C++拥有了简单实现继承及动态方法调用的特性, 但一个 C++的数据结构并不能自动成为面向对象的。方法调用决议(method resolution)以及在子类中重写一个函数操作的前提必须是这个函数/方法已经在父类中被声明为 virtual。也就是说, 必须在最初的类中我们就能预见到一个函数是否需要被重写。不幸的是, 类的撰写者可能不会预期到需要定义一个特殊的子类, 也可能不知道那些操作将要在子类中被重写。这意味着当子类被定义时, 我们经常需要回过头去修改我们的父类, 并且使得对于通过创建子类来重用已有的库的限制极为严格, 尤其是当这个库的源代码不能被获得是更是如此。(当然, 你也可以将所有的操作都定义为 virtual, 并愿意为此付出一些小小的内存花费用于函数调用)【RBPEL91】

然而, 让程序员来处理 virtual 是一个错误的机制。编译器应该能够检测到多态, 并为此产生所必须的、潜在的实现 virtual 的代码。让程序员来决定 virtual 与否对于程序员来说是增加了一个簿记工作的负担。这也就是为什么 C++只能算是一种弱的面向对象语言 (weak object-oriented language): 因为程序员必须时刻注意着一些底层的细节 (low level details), 而这些本来可以由编译器自动处理的。

在 C++中的另一个问题是错误的重写 (mistaken overriding), 父类中的函数可以在毫不知情的情况下被重写。编译器应该对于同一个名字空间中的重定义报错, 除非编写子类的程序员指出他是有意这么做的 (即对于虚函数的重写)。我们可以使用同一个名字, 但是程序员必须清楚自己在干什么, 并且显式地声明它, 尤其是在将自己的程序与已经存在的程序组件组装成新的系统的情况下更要如此。除非程序员显式地重写已有的虚函数, 否则编译器必须要给我们报告出现了名字被声明多处(duplicate declaration)的错误。然而, C++却采用了 Simula 最初的做法, 而这种方法到现在已经得到了改良。其他的一些程序语言通过采用了更好的、更加显式的方法, 避免了错误重定义的出现。

解决方法就是 virtual 不应该在父类中就被指定好。当我们需要运行时的动态绑定时, 我们就在子类中指定需要对某个函数进行重写。这样做的好处在于: 对于具有多态性的函数, 编译器可以检测其函数签名(function signature)的一致性; 而对于重载的函数, 其函数签名在某些方面本来就不一样。第二个好处表现在, 在程序的维护阶段, 能够清楚地表达程序的最初意愿。而实际上后来的程序员却经常要猜测先前的程序员是不是犯了什么错误, 选择一个相同的名字, 还是他本来就想重载这个函数。

在 Java 中, 没有 virtual 这个关键字, 所有的方法在底层都是多态的。当方法被定义为 static, private 或是 final 时, Java 直接调用它们而不是通过动态的查表的方式。这意味着在需要被动态调用时, 它们却是非多态性的函数, Java 的这种动态特性使得编译器难以进行进一步的优化。

Eiffel 和 Object Pascal 迎合了这个选项。在它们中，编写子类的程序员必须指定他们所想进行的重定义动作。我们可以从这种做法中得到巨大的好处：对于以后将要阅读这些程序的人及程序的将来维护者来说，可以很容易地找出来被重写的函数。因而选项 2 最好是在子类中被实现。

Eiffel 和 Object Pascal 都优化了函数调用的方式：因为他们只需要产生那些真正多态的函数的调用分配表的入口项。对于怎样做，我们将会在 global analysis 这节中讨论。

选项 3

纯虚函数这样的做法迎合了让一个函数成为抽象的，从而子类在实例化时必须为其提供一个实现这样的一个条件。没有重写这些函数的任何子类同样也是抽象类。这个概念没有错，但是请你看一看 pure virtual functions 这一节，我们将在那节中对于这种术语及语法进行批判讨论。

Java 也拥有纯虚方法(同样 Eiffel 也有)，实现方法是为该方法加上 deferred 标注。

结论：

virtual 的主要问题在于，它强迫编写父类的程序员必须要猜测函数在子类中是否有多态性。如果这个需求没有被预见到，或是为了优化、避免动态调用而没有被包含进去的话，那么导致的可能性就是极大的封闭，胜过了开放。在 C++的实现中，virtual 提高了重写的耦合性，导致了一种容易产生错误的联合。

Virtual 是一种难以掌握的语法，相关的诸如多态、动态绑定、重定义以及重写等概念由于面向于问题域本身，掌握起来就相对容易多了。虚拟函数的这种实现机制要求编译器为其在 class 中建立起 virtual table 入口，而 global analysis 并不是由编译器完成的，所以一切的重担都压在了程序员的肩上了。多态是目的，虚拟机制就是手段。Smalltalk， Objective-C， Java 和 Eiffel 都是使用其他的一种不同的方法来实现多态的。

Virtual 是一个例子，展示了 C++在 OOP 的概念上的混沌不清。程序员必须了解一些底层的概念，甚至要超过了解那些高层次的面向对象的概念。Virtual 把优化留给了程序员；其他的方法则是由编译器来优化函数的动态调用，这样做可以将那些不需要被动态调用的分配（即不需要在动态调用表中存在入口）100%地消除掉。对于底层机制，感兴趣的应该是那些理论家及编译器实现者，一般的从业者则没有必要去理解它们，或是通过使用它们来搞清楚高层的概念。在实践中不得不使用它们是一件单调乏味的事情，并且还容易导致出错，这阻止了软件在底层技术及运行机制下（参见并发程序）的更好适应，降低了软件的弹性及可重用性。

钻穿牛角尖：求两数和

bugn

编者按：在用 C++ 编写程序的过程中，我们都会遇到一些不大不小的问题。说穿了不名一钱，可往往令很多高手阴沟里翻船。相信大家都有各自宝贵的经验吧？您不妨把这些经验与故事告诉我们，让大家共同分享。

问题：

怎么编写求两数和的函数？

解决：

简单吧？既然没有给定这两个数的类型，当然是用模板了。

```
template <typename P1, typename P2>
P1 add(const P1& p1, const P2& p2)
{
    return p1+p2;
}
```

OK 了，是吧？呵呵，您怎么知道返回值一定是 P1 呢？如果我们要这个返回值是 P2 呢？哦，那再加一段

```
template <typename P1, typename P2>
P2 add(const P1& p1, const P2& p2)
{
    return p1+p2;
}
```

又 OK 了是吧？咦，编译不能通过啊。两个函数只有返回类型不一样，当然不能同时存在。怎么办？保留 P1？保留 P2？都要？麻烦了。

其实，应该可以想到的是返回一个新类来当作类型。

```
template <typename P1, typename P2>
AddT<P1, P2> add(P1 x, P2 y)
{
    return AddT<P1, P2>(x,y);
}
```

这是在 STL 里大量使用的办法。注意这里 AddT 模版构造了一个新的类 AddT<P1, P2>。我们接着定义 AddT 如下：

```

template <typename P1, typename P2>
class AddT
{
public:
    AddT(const P1& p1, const P2& p2)
        : m_p1(p1), m_p2(p2) {}
    operator P1()
    {return m_p1 + (P1)m_p2;}
    operator P2()
    {return (P2)m_p1 + m_p2;}
private:
    const P1& m_p1;
    const P2& m_p2;
};

```

在使用的时候我们还要指定返回之类型。例如 `(int)add(3.0, 2)`
又 OK 了是吧？呵呵，再看看，如果是

```

float a=1.0;
float b=2.0;
(int)add(a,b);

```

这时编译又出问题了。由于这个时候类型 P1、P2 是一样的，那么那个类型转换函数就出现重复定义，怎么办？

模板部分特化 (template partial specilization) 就是救命稻草。不过 VC6 不支持 (VC6 仅支持一种很特殊的情况)，GCC2.95 和 BCC5.5.1 可以。

加入以下代码：

```

template <typename P1>
class AddT<P1, P1>
{
public:
    AddT(P1 p1, P1 p2)
        : m_p1(p1), m_p2(p2) {}
    operator P1()
    {return m_p1 + (P1)m_p2;}
private:
    const P1 &m_p1, &m_p2;
};

```

这下应该差不多了吧？看看，一个这么简单的问题，居然也折腾了这么久。

Traits：实用的模板新技术

Nathan C.Myers

虫虫译

译注：Nathan C.Myers 在夏威夷长大，从 1993 年起，他就是 ISO/ANSI C++ 标准委员会的成员。C++ 标准化最困难的部分之一——本地化，以及关键字 explicit，都有他莫大的功劳。他在 C++ Report 和 Dr. Dobb's Journal 发表过多篇文章。本文于 1995 年 6 月发表在 C++ Report 上，虽然已经过了好几年了，但对 traits 的叙述非常清晰、有见地，值得一看。本文也提到了业界一句著名的“谚语”：extra level of indirection（额外的中间层）。本文翻译经过 Myers 先生授权，原文在 <http://www.cantrip.org/traits.html>，有兴趣的朋友不妨参考一下。Translated by Chong-chong, with original author's permission. Copyright 2001 by Chong-chong & Nathan Myers.

C++ 标准库的一个重要内容就是国际化，使其能适用于不同的语言，流类和字符串类理所当然在其中扮演着重要的角色。然而实际操作起来却困难重重，不过由此也促成了一项非常有用的新技术——Traits。

问题的产生

众所周知，在输入输出流 Iostream 库中，streambuf 的实现依赖于一个非常特殊的值：文件结束符 EOF。在传统的流库中，EOF 的类型是 int，当然相应的读取函数也返回 int 类型：

```
class streambuf {  
    ...  
    int sgetc(); // 返回下一个字符或者是 EOF  
    int sgetn(char*, int N); // 获取 N 个字符  
};
```

注意这里有两种类型：字符类型 char 和 EOF 的类型 int。

由于国际化的需要，我们并不仅仅局限于处理单字节的 char，比如双字节的 wchar_t，于是我们将 streambuf 模板化。

```
template <class charT, class intT>  
class basic_streambuf {  
    ...  
    intT sgetc();  
    int sgetn(charT*, int N);  
};
```

同样，我们需要两个类型：字符类型 charT 和 EOF 类型 intT。问题就出在这个 intT 上。

对于使用者而言，结束符是哪个，是什么类型，根本不需要也不应该关心。更糟糕的是，`sgetc` 函数到底该返回什么？另一个模板参数？大麻烦出现了。这时候，救星降临了。

Traits 技术

针对 `charT`，我们先写一个比较奇怪的东西。

```
template <class charT>
struct ios_char_traits { };
```

您可能纳闷了，这是什么东西啊，空空如也。呵呵，没错，对于一个陌生的 `charT` 类型，您又知道些什么呢？不过对于实在的类型，我们就要写一些实在的东西了。

```
struct ios_char_traits<char> {
    typedef char char_type;
    typedef int int_type;
    static inline int_type eof()
    { return EOF; }
};
```

奇怪了，这个 `ios_char_traits<char>` 也没有任何数据成员，只有些 `typedef` 定义和一个静态成员函数。呵呵，别急，好戏登场了。

```
template <class charT>
class basic_streambuf {
public:
    typedef ios_char_traits<charT> traits_type;
    typedef traits_type::int_type int_type;
    int_type eof()
    { return traits_type::eof(); }

    ...
    int_type sgetc();
    int sgetn(charT*, int N);
};
```

除了这些奇奇怪怪的 `typedef`，倒跟前面差不多。嘿嘿，可得注意了，这次可只有一个模板参数，也就是使用者唯一需要关心的参数——字符类型，仔细体会体会吧。编译器通过我们的 `traits` 类就得到了足够的信息。

如果我们要在流类中使用新的字符类型，比如宽字符类型 `wchar_t`，只需要用新类型把 `ios_char_traits` 特化一下就可以了：

```
struct ios_char_traits<wchar_t> {
    typedef wchar_t char_type;
    typedef wint_t int_type;
    static inline int_type eof()
        { return WEOF; }
};
```

字符串当然可以用同样的方法处理。

在模板参数为原生类型或者某些不能添加成员的类的时候，traits 技术就大有用武之地。

另一个例子

我们再看看 Traits 的用法。下面这个例子来自于 ANSI/ISO C++ 标准草案。

假如我们要写一个数值分析库，那么这个库至少应该能处理 float、double 和 long double 类型。对每种类型而言，都有最大指数值、epsilon 和尾数长度等等信息。这些信息都定义在头文件 <float.h> 中，不过模板参数化的时候可不知道具体的类型应该对应于 FLT_MAX_EXP 还是 DBL_MAX_EXP。而 Traits 技术就能简洁明了的解决这一问题。

```
template <class numT>
struct float_traits { };
struct float_traits<float> {
    typedef float float_type;
    enum { max_exponent = FLT_MAX_EXP };
    static inline float_type epsilon()
        { return FLT_EPSILON; }
    ...
};

struct float_traits<double> {
    typedef double float_type;
    enum { max_exponent = DBL_MAX_EXP };
    static inline float_type epsilon()
        { return DBL_EPSILON; }
    ...
};
```

这下我们就可以放心使用 max_exponent，而不必管它到底是 float 还是 double，或者甚至是您自己定义的类。

```

template <class numT>
class matrix {
public:
    typedef numT num_type;
    typedef float_traits<num_type> traits_type;
    inline num_type epsilon()
    {return traits_type::epsilon();}
    ...
};

```

注意在我们举过的例子中，每个模板都提供了一些 public 的 `typedef`，用以说明自身及相关的信息。这些信息就是 Traits 技术的关键，记住了，始终利用 `typedef` 提供相关的信息。

默认模板参数

上面的例子在 1993 年的编译器上就应该可以实现。在那年 11 月的一次会议上，C++作出了一个小小的改动，那就是默认模板参数，由此我们可以写出更好的程序。

下面这个例子来自 Stroustrup 的 **Design and Evolution of C++** 的 359 页。首先，我们假设有这么一个有点儿像 traits 的模板类 CMP：

```

template <class T> class CMP {
    static bool eq(T a, T b)
    { return a == b; }
    static bool lt(T a, T b)
    { return a < b; }
};

```

以及一个我们很熟悉的字符串模板：

```
template <class charT> class basic_string;
```

于是我们可以定义如下一个操作字符串的函数 compare：

```
template <class charT, class C = CMP<charT>>
int compare(const basic_string<charT>&, const basic_string<charT>&);
```

忽略掉那些实现细节，我们把目光放在 `compare` 的参数上。首先注意第二个参数 `C`，它的默认类型不仅仅是一个类，而且还是一个具体化的模板类；其次，注意这个模板类的参数居然是前一个参数 `charT`！这在普通的函数声明中是绝对不可以的！不过对于模板参数又另当别论了。

由于有了 `eq` 和 `lt` 的默认定义，我们可以借此用 `compare` 函数来比较两个字符串，如果需

要的话（比如进行大小写不敏感的比较），甚至还可以换成我们自己的定义。同样，我们可以把 `streambuf` 模板写成这样：

```
template <class charT, class traits = ios_char_traits<charT> >
class basic_streambuf {
public:
    typedef traits traits_type;
    typedef traits_type::int_type int_type;
    int_type eof() { return traits_type::eof(); }
    ...
    int_type sgetc();
    int sgetn(charT*, int N);
};
```

这样，对于某种特殊的字符类型，即使其所属字符集的文件结束符比较特别，我们都可以用相应的 `traits` 替换即可。

运行时间变量 (Runtime-variable) Traits

我们甚至还能走得更远。`basic_streambuf` 的构造函数我们可以这样写：

```
template <class charT, class traits = ios_char_traits<charT> >
class basic_streambuf {
    traits traits_; // member data
    ...
public:
    basic_streambuf (const traits& b = traits())
        : traits_(b) { ... }
    int_type eof() { return traits_.eof(); }
};
```

加上一个默认的构造函数参数，我们不仅在编译时间，甚至在运行时间也可以使用 `traits` 模板参数。这时，`traits_.eof()` 可能调用 `traits` 的某个静态或者正常的成员函数。一个非静态的成员函数就能使用从构造函数传递进来并保存了的值，STL 中的 `allocator` 对于标准容器就是这个道理。

总结

`Traits` 技术非常使用，几乎可以在所有支持模板的编译器编译通过。它提供了一种非常漂亮

的手段来联系相关的类型、值以及模板函数。而语言上一个简单的扩充，而使得这项技术更加富有弹性而不损及效率，甚至在运行时间照样能够发挥作用。

有人曾经希望我给 Traits 类下一个定义，我想可以是：

一种聚合了有用的类型和常数、用以替代模板参数的类，成功地实践了“extra level of indirection(额外的中间层)”这句可以解决所有软件难题的名言。(A class used in place of template parameters. As a class, it aggregates useful types and constants; as a template, it provides an avenue for that "extra level of indirection" that solves all software problems.)

也有人曾问我这项技术以 traits 为名的来龙去脉。其实以前是叫做 *baggage*，我一直都偏好这个名字。但是没办法，有人不喜欢这个词出现在标准里，莫名其妙地提出一些无聊的理由。早期对 *baggage template* 的定义提到它包含了 traits，于是某人(我忘了是谁)说不如就叫 *traits template*，于是就有了现在这个名。

需要说明一点。您也许有时候会发现，本文讲的方法在有些编译器上不能编译通过。解决的方法是用关键字 `typename` 来提及模板类的成员类型：

```
template <class charT, class traits = ios_char_traits<charT> >
class basic_streambuf {
public:
    typedef traits traits_type;
    typedef typename traits_type::int_type int_type;
    int_type eof() { return traits_type::eof(); }
    ...
    int_type sgetc();
    int sgetn(charT*, int N);
};
```

只要编译器支持 `typename`，就万事大吉了。

参考文献

- Stroustrup, B. **Design and Evolution of C++**, Addison-Wesley, Reading, MA, 1993.
- Barton, J.J., and L.R. Nackman, **Scientific and Engineering C++**, Addison-Wesley, Reading, MA, 1994.
- Veldhuizen, T. "[Expression Templates](#)", **C++ Report**, June 1995, SIGS Publications, New York.

C++中常用的几个小技巧

plpliuly

也许只是一些技巧而已，不过值得反复重用就变成语言的 Idioms 了。说到 Idioms，POSA 认为 Pattern 可以分为三个层次（或者粒度），Architecture patterns、Design patterns、Programming language Idioms。也就是说 Idioms 很多是跟语言相关的。

1、实现抽象类（不允许实例化）

这个可能大家都知道，把构造函数声明为 protected。（当然如果类中包含纯虚函数自然就不能实例化，但是有的抽象类需要为虚函数提供缺省实现，因此可能不包含纯虚函数。）

2、实现 final 类（不允许被派生）

```
namespace Private{
    class NonDerivableHelper
    {
        NonDerivableHelper() {}
        friend class NonDerivable;
    };
}

#ifndef NDEBUG
#define FINAL_CLASS
#else
#define FINAL_CLASS : private virtual Private::NonDerivableHelper
#endif

class NonDerivable FINAL_CLASS
{
    ...
};
```

这里 namespace 并非必须，只是一个比较好的习惯而已。

3、实现禁止拷贝构造和赋值操作符

```
class MyClass
{
private:
    MyClass(const MyClass& );
    const MyClass& operator = ( const MyClass& );
};
```

为了防止成员函数调用操作，这两个函数应该只有声明没有实现。

C++中的 Idioms 很多很多，比如 TypeTraits、SmartPointer 这些都可以视为经典的 Idioms。

注：关于第二个实现 final 类的方式，只要是 virtual 继承即可，至于是 private、protected 还是 public 关系不大。在 comp.lang.c++.moderated (2001/5/15) 上，Andrei Alexandrescu 有他的见解，他不赞成上面文章中使用的 friend-based 方式，直接去掉 friend 那一行，这时必须采用 private virtual 继承方式。private virtual 继承确实使得没有任何类能只继承 NonDerivable。

```
class D: public NonDerivable
{
    D() {} //编译错误
}
```

不过多重虚拟继承又绕了过去

```
class D FINAL_CLASS, public NonDerivable
{
    D() {} //一切正常
}
```

Andrei Alexandrescu 又提出了一个非常巧妙的办法：把 Private::NonDerivableHelper 的构造函数加一个参数，改为 NonDerivableHelper(int) {}，然后把 NonDerivable 的构造函数写成 NonDerivable(): NonDerivableHelper(0) {}

```
#ifdef NDEBUG
#define FINAL_CLASS
#else
#define FINAL_CLASS : private virtual Private::NonDerivableHelper
#endif

namespace Private
{
    class NonDerivableHelper
    {
        protected:
            NonDerivableHelper(int) {} //ADDED CTOR ARGUMENT
    };
}

class NonDerivable FINAL_CLASS
{
public:
    NonDerivable() : NonDerivableHelper(0) {} //PASS A DUMMY VALUE
};
```

Smart Pointer 访谈录

虫 虫

您好，感谢您接受我们的采访。能简单地介绍一下您的名字吗？

我的英文名叫 smart pointer，中文名我自己都不清楚，有人叫我“聪明指针”，有人又叫我“灵巧指针”，还有“智能指针”，都不太爽。不过就连 smart pointer 也不是名副其实，因为我并不是一个指针（pointer）。

您不是一个指针？那您是？

其实我是某种形式的类（Class），不过我的特长就是模仿 C/C++ 中的指针，当然大家也就稀里糊涂地叫我 pointer 了。所以希望大家一定要记住两点：我是一个类而非指针，但我的特长是模仿指针。

那您怎么做到像指针的呢？

易容术知道吗？C++ 的模板技术和运算符重载给了我很大的发挥空间。为了装得像，首先我必须是高度类型化的（strongly typed），模板给了我这个功能；其次我需要模仿指针主要的两个运算符 -> 和 *，那就需要进行运算符重载。

这样说似乎很抽象，您能详细说明一下吗？

其实很容易勾画出中国人的基本特点：黄皮肤、黑头发（大部分老人除外）能吃苦、能存钱、爱打麻将。当然不排除例外，我只是觉得我接触的大部分中国人都这样。同样，对于 smart pointer，也可以给出一个大概的轮廓。

```
template<class T> class SmartPtr {  
public:  
    SmartPtr(T* p = 0);  
    SmartPtr(const SmartPtr& p);  
    ~SmartPtr();  
    SmartPtr& operator=(SmartPtr& p);  
  
    T& operator*() const {return *the_p;}  
    T* operator->() const {return the_p;}  
private:  
    T *the_p;  
}
```

正如我所说的，这只是一个大概的印象，很多东西是可以更改的。比如可以去掉或加上一些 const，这都需要根据具体的应用环境而定。注意重载运算符 * 和 ->，正是它们使我看起来跟普通的指针很相像。而由于我是一个类，在构造函数、析构函数中您都可以通过恰当的编程达到一些不错的效果。

那您能给大家举一个例子吗？

当然可以。比如 C++ 标准库里的 `std::auto_ptr` 就是应用很广的一个例子。它的实现在不同版本的 STL 中虽有不同，但原理都是一样，大概是下面这个样子：

```
template<class X> class auto_ptr
{
public:
    typedef X element_type;

    explicit auto_ptr(X* p = 0) throw()
        : the_p(p) {}

    auto_ptr(auto_ptr<X>& a) throw()
        : the_p(a.release()) {}

    auto_ptr<X>& operator =(auto_ptr<X>& rhs) throw()
    {
        reset(rhs.release());
        return *this;
    }

    ~auto_ptr() throw() {delete the_p;}

    X& operator* () const throw() {return *the_p;}
    X* operator-> () const throw() {return the_p;}

    X* get () const throw() {return the_p;}
    X* release() throw()
    {
        X* tmp = the_p;
        the_p = 0;
        return tmp;
    }
    void reset(X* p = 0) throw()
    {
        if (the_p!=p)
        {
            delete the_p;
            the_p = p;
        }
    }
private:
    X* the_p;
};
```

关于 `auto_ptr` 的使用我不想多说，这不是我们今天的主要话题。它的主要优点是不用 `delete`，可以自动回收已经被分配的空间，由此可以避免资源泄露的问题。很多 Java 的拥护者经常不分黑白的污蔑 C++ 没有垃圾回收机制，其实不过是贻笑大方而已。抛开在网上许许多多的商业化和非商业化的 C++ 垃圾回收库不提，`auto_ptr` 就足以有效地解决这一问题。并且即使在产生异常的情况下，`auto_ptr` 也能正确地回收资源。这对于写出异常安全（exception-safe）的代码具有重要的意义。

那在使用 smart pointer 的过程中，是否有什么值得注意的问题呢？

这个问题就太泛泛了，针对不同的 smart pointer，有不同的注意事项。比如 `auto_ptr`，您就不能把它用在标准容器里，因为它只在内存中保留一份实例。不过我相信把握我前面说的两个原则：smart pointer 是类而不是指针，是模仿指针，那么一切问题都好办。比如，smart pointer 作为一个类，那么以下的做法就可能有问题。

```
SmartPtr<int> p;
if(p==0)
if(!p)
if(p)
```

很显然，`p` 不是一个真正的指针，这么做可能出错。而 `SmartPtr` 的设计也是很重要的因素。您可以加上一个 `bool SmartPtr::null() const` 来进行判断。如果坚持要用上面的形式，那也未尝不可。我们就加上 `operator void*()` 试试：

```
template<class T> class SmartPtr {
public:
...
operator void*() const {return the_p;}
...
private:
T* the_p;
};
```

这招在 `basic_ios` 中就使用过了。这里也可以更灵活地处理，比如类本身需要 `operator void*()` 这样地操作，那么上面这招就不灵了。那我们还有重载 `operator !=()` 等等方法。不怕做不到，只怕想不到。

您能总结一下 smart pointer 的实质吗？

smart pointer 的实质就是一个外壳，一层包装。正是多了这层包装，我们可以做出许多普通指针无法完成的事，比如前面资源自动回收，或者自动进行引用记数，比如 ATL 中 `CComPtr` 和 `CComQIPtr` 这两个 COM 接口指针类。然而事事都是一把双刃剑，正由于多了这些功能，又会使 smart pointer 丧失一些功能。一定切记，画虎画皮难画骨，smart pointer 毕竟和真正的指针是大大不同的。

非常感谢您的接受采访。祝您在中国玩得愉快！

谢谢。

深入 BCB 理解 VCL 的消息机制（一）

CKER

引子

本文所谈及的技术内容都来自于 Internet 的公开信息。由笔者在闲暇之际整理后，贴出来以饴网友，姑且妄称原创。每次在国外网站上找到精彩文章的时候，心中都会暗自叹息，为什么在中文网站难以觅得这类文章呢？其实原因大家都明白。

时至今日，学习 Windows 编程的兄弟们都知道消息机制的重要性。所以理解消息机制也成了不可或缺的功课。

大家都知道，Borland 的 C++ Builder 以及 Delphi 的核心是 VCL。作为 Win32 平台上的开发工具，封装 Windows 的消息机制当然也是必不可少的。

那么，在 C++ Builder 中处理消息的方法有哪些呢？它们之间的区别又在哪里？如果您很清楚这些，呵呵，对不起啦，请关掉这个窗口。

如果不清楚那就和我一起深入 VCL 的源码看个究竟吧。

方法 1：使用消息映射(Message Map)重载 TObject 的 Dispatch 虚成员函数

这个方法大家用的很多。形式如下

```
BEGIN_MESSAGE_MAP  
VCL_MESSAGE_HANDLER( ... ...)  
END_MESSAGE_MAP( ... )
```

但这几句话实在太突兀，C++ 标准中没有这样的定义。不用讲，这显然又是宏定义。它们到底怎么来的呢？CKER 第一次见到它们的时候，百思不得其解。嘿嘿，不深入 VCL，怎么可能理解？

在\Borland\CBUILDER5\Include\Vcl 找到 sysmac.h，其中有如下的预编译宏定义：

```
#define BEGIN_MESSAGE_MAP virtual void __fastcall Dispatch(void *Message) \  
{ \  
    switch (((PMessag)Message)->Msg) \  
    { \  
  
#define VCL_MESSAGE_HANDLER(msg,type,meth) \  
    case msg: \  
        meth(*((type *)Message)); \  
        break; \  
  
// NOTE: ATL defines a MESSAGE_HANDLER macro which conflicts with VCL's macro. The  
// VCL macro has been renamed to VCL_MESSAGE_HANDLER. If you are not using ATL,  
// MESSAGE_HANDLER is defined as in previous versions of BCB.  
file://  
#if !defined(USING_ATL) && !defined(USING_ATLVCL) && !defined(INC_ATL_HEADERS)  
#define MESSAGE_HANDLER VCL_MESSAGE_HANDLER  
#endif // ATL_COMPAT
```

```
#define END_MESSAGE_MAP(base)
    default: \
        base::Dispatch(Message); \
    break; \
} \
}
```

这样对如下的例子：

```
BEGIN_MESSAGE_MAP
VCL_MESSAGE_HANDLER(WM_PAINT, TMessage, OnPaint)
END_MESSAGE_MAP(TForm1)
```

在预编译时，就被展开成如下的代码

```
virtual void __fastcall Dispatch(void *Message)
{
    switch (((PMessage)Message)->Msg)
    {
        case WM_PAINT:
            OnPaint((TMessage *)Message));
            //消息响应句柄，也就是响应消息的成员函数，在 TForm1 中定义
            break;
        default:
            TForm1::Dispatch(Message);
            break;
    }
}
```

这样就很顺眼了，对吧。对这种方法有两点要解释一下：

1. `virtual void __fastcall Dispatch(void *Message)` 这个虚方法的定义最早可以在 `TObject` 的定义中找到。打开 BCB 的帮助，查找 `TForm` 的方法 (Method)，你会发现这里很清楚的写着 `Dispatch` 方法继承自 `TObject`。如果您关心 VCL 的继承机制的话，您会发现 `TObject` 是所有 VCL 对象的基类。`TObject` 的抽象凝聚了 Borland 的工程师们的心血。如果有兴趣。您应该好好查看一下 `TObject` 的定义。

很显然，所有 `TObject` 的子类都可以重载基类的 `Dispatch` 方法，来实现自己的消息调用。如果 `Dispatch` 方法找不到此消息的定义，会将此消息交由 `TObject::DefaultHandler` 方法来处理。抽象基类 `TObject` 的 `DefaultHandler` 方法实际上是空的。同样要由继承子类重载实现它们自己的消息处理过程。

2. 很多时候，我见到的第二行是这样写的：

```
MESSAGE_HANDLER(WM_PAINT, TMessage, OnPaint)
```

在这里，您可以很清楚地看到几行注解，意思是 ATL 中同样包含了一个 `MESSAGE_HANDLER` 的宏定义，这与 VCL 发生了冲突。为了解决这个问题，Borland 改用 `VCL_MESSAGE_HANDLER`。当您没有使用 ATL 的时候，`MESSAGE_HANDLER` 将转换成 `VCL_MESSAGE_HANDLER`。但如果您使用了 ATL 的话，就会有问题。所以我建议您始终使用 `VCL_MESSAGE_HANDLER` 的写法，以免出现问题。（未完待续）

C++View

视点：

庖丁解羊

精彩推荐：

- STL有序容器武道会
- Traits：类型的 else-if-then 机制
- 开放 - 封闭原则 OCP

C++View

第 2 期

第 2 期 目录 2001 年 8 月

视点

庖丁解羊	03
闲谈编程思想	

回音壁	53
-----	----

抢鲜快报

GCC 3.0 印象记	06
-------------	----

管中窥豹

钻穿牛角尖	08
using 声明 vs. 虚函数	
C++的不足之处讨论系列（二）	10
全局分析	

泛型编程与标准模板库

Generic<Programming>	12
Traits：类型的 else-if-then 机制	

STL 有序容器武道会	23
-------------	----

设计样式与原则

设计笔记 Engineering Notebook	34
开放 - 封闭原则 OCP	

面向对象与应用架构

深入 BCB 理解 VCL 的消息机制（二）	47
------------------------	----

导读

首先，小编宣布一件值得所有炎黄子孙高兴的事：
北京申奥成功！（台下扔来臭鸡蛋无数，还有声音在骂：“萨马兰奇都宣布快一个月了，还要你说？！”小编当然有备而来，撑开雨伞挡住枪林“蛋”雨，继续）那天正好是 C++ View 创刊的日子，得了一个好彩头。以后我们可以真切感受到北京 7 年内发生的变化了。比如环境越来越好，房价越来越高，想买房的朋友的 money 啊……

要说这申奥成功的原因嘛，我看……（台下又传来不满的声音：“今天是介绍 C++ View 还是申奥啊？”诸位不急，不急）主要是中华文化的吸引力。华夏文明的一个核心内容就是“相生相克”，二进制就是一个不错的例子。这期我们推出 Robert C.Martin 的经典文章《开放 - 封闭原则 OCP》。又开放又封闭，是否矛盾呢？从“相生相克”的原理去想想吧。

“相生相克”的思想由老子整理为道家的核心思想之一，其下当然是庄子。今天我们就推出现代版的《庖丁解牛》，哦，不对，是《庖丁解羊》。牛呢？呵呵，牛角尖都穿了，早跑了。这次的《钻穿牛角尖》我们将讨论一个虚函数的实现问题，不少编译器在这里都翻了船哦。（台下有人问：“最新的 GCC 3.0 没问题吧？”）呵呵，不急，这期专门为大家奉上《GCC 3.0 印象记》。

申奥其实是各个城市的比武大会，当然 STL 里的容器也有这么一说，请看《STL 有序容器武道会》。同时我们还提供了一道大餐：Andrei Alexandrescu 的《Traits：类型的 else-if-then 机制》，一定要好好体会体会。

这期的《C++的不足之处讨论系列（二）》我们将讨论全局分析，同时继续《深入 BCB 理解 VCL 的消息机制（二）》。

请所有关心本刊的朋友都看看本期的《回音壁》。

庖丁解羊

闲谈编程思想

作者：holyfire

经过多年的编程，吾发现编程的思想对一个程序员是至关重要的，以前拿到一个问题，马上就开始考虑如何来一点一点把他用代码来实现，于是啃啊啃啊于愁眉笼罩千辛万苦下每完成了一点就欣喜若狂。编着编着发现这里不足，那里没考虑到于是东补补西凑凑，异常蹩脚的完成了一个小程序，当时的感觉就像我完成了全世界，或许那个时代是我最开心的。

完成了四五个程序的我，开始发现我做了很多重复的劳动，于是乎开始做我的程序库，编写功能型模块。想到了这些模块只要写一遍就可以受用无穷不禁得意洋洋，或许那个时代我是聪明。

当我写了五六千行代码的时候，我发现我写的功能模块已经记不清什么是干什么的了，而我已经习惯使用别人写的比我好的模块时，我开始迷茫，我做的一切究竟是为了什么。这时我开始研究别人的代码，开始懂得什么是优化，什么是数据结构，什么是数学建模，我开始研究应用数学和软件工程。这时我开始使用 C++ 用类来写程序，或许那个年代我是明智的。

当我写了四五十个*.h 和 *.cpp 后，开始发现我写的类一点都不通用，无法再利用。我开始深入学习类以及面向对象，明白了继承和封装，这时的我才开始深深思考什么是重要的，于是我开始分析问题而不是一上手就开始编代码。或许以前的我是个傻瓜^_^。

我足足花了三年多的时间来浪费我的青春做了别人仅需一年的事情（当然着三年中我积累的经验不可能是一年能做到的，但是对于编程上的修养，我浪费的时间已太多），如果你们没有意识到这点，或许也会浪费很多时间。

分析能力对一个程序员来说是一个重要的属性，它直接关系到你对事物的理解能力和组织能力，程序员做的工作就是将一个事物分解成数学元素然后重新组织在计算机里重现，所以如何处理事物是个要点。

我们了解一个事物并进行描述的时候，往往是先处理它的一部分，接着处理另一部分，然后等所有的部分都处理完的时候，具体的事物就重现了。比如给一个不知道山羊的人介绍山羊，如果你对他说山羊就是山羊，他是无论如何都不会明白的。如果说山羊是一种有四只脚的脑袋上有两只角，还有短短的尾巴，那他模模糊糊会有些形象在脑海里了，如果加上有绒绒的毛，可笑的胡子，咩咩的叫的食草动物，至少他不会和牛混淆起来，如果你能描述的更详细更好，那他有九成知道山羊是什么样子了，当他看见山羊的时候就会立刻明白。对于什么都不懂得计算机你不用说明什么，但是要是别人或者几年以后的你自己能看明白你的文档，详细的说明是必须的。而从上面的过程看来，将事物分成小块来处理是个好办法，我们把这个过程叫做划分。当然划分是可以继续下去，将划分的小块继续划分直到不能划分为止。当然要对一只山羊划分需要一个对山羊有着全面了解的人，所以划分你得问题的时候要对问题有全面的了解。看上去这对一个还没有了解这个问题的人是一个矛盾，划分一个事物有助于了解这个事物而要很好的划分却需要对事物充分的了解。于是我们要有所改变，要改变的是我们的做法。我们先粗略的了解问题，然后粗略的划分一下，将划分的模块再粗略的了解一下，然后将它粗略的划分，等到都成为不可划分的小块时，我们再来组织它。而且这

那个时候，我们对问题已经有全面的了解了，这时候我们将小块的重复部分扔掉（一般一定会有重复的部分），然后将这些小块重新将问题组合起来，想一下将一只大卸八块的山羊组合起来，哈哈，一定是活不过来了，不过有羊肉吃也不错。如果你很顺利的组合起来了，表示你很好的划分了这个问题而且了解的很透彻，以后的事将会一帆风顺，你这时脑袋里一定充满了解决这个问题的方案，已经跃跃欲试了。不要急，聪明的人会仔细的将划分的过程多看几遍，重组的方法多试几种，这不但对以后和编程有好处，也是更好解决方案的起点。

现在我们来解剖一只山羊，如果你愿意还可以细分，动物爱好者请回避。

山羊

- > 脑袋
- > 躯体
- > 四肢
- > 尾部

脑袋	躯体	四肢	尾部
五官	胸	两只前肢	尾巴
胡子	腹	两只后肢	绒毛
绒毛	背		骨架
骨架	内脏		
	绒毛		
	骨架		
五官	内脏	前肢	
耳朵	心	脚	
眼睛	脾	绒毛	
鼻子	肝	骨架	
嘴巴	肺	后肢	
	胃	脚	
	肠	绒毛	
	肾	骨架	

最后我们得到的是：

山羊 脑袋 躯体 四肢 尾部 脑袋 五官 胡子 绒毛 骨架 躯体 胸 腹 背 内脏 绒毛 骨架 四肢 两只前肢 两只后肢 尾部 尾巴 绒毛 骨架 五官 耳朵 眼睛 鼻子 嘴巴 内脏 心 脾 肝 肺 肾 胃 肠 前肢 脚 绒毛 骨架 后肢 脚 绒毛 骨架

去掉重复的部分：

山羊 脑袋 躯体 四肢 尾部 五官 胡子 绒毛 骨架 胸 腹 背 内脏 两只前肢 两只后肢 尾巴 耳朵 眼睛 鼻子 嘴巴 心 脾 肝 肺 肾 胃 肠 脚 脚

最终的不可划分的子模块：

胡子 绒毛 骨架 胸 腹 背 尾巴 耳朵 眼睛 鼻子 嘴巴 心 脾 肝 肺 胃 肠 脚

现在我们将它重新组合一下，是不是很简单的就完成了。而且我们需要处理的东西并不多。现在我们来用数学元素来描述一下

胡子： 数量不多的多种形状的轮廓，有各种颜色
绒毛： 非常多的多种形状的轮廓，有各种颜色
骨架： 多种形状的轮廓，有一种颜色
胸： 某形状的轮廓，有一种颜色
腹： 某形状的轮廓，有各种颜色
背： 某形状的轮廓，有各种颜色
尾巴： 某形状的轮廓，有各种颜色
耳朵： 最多两个的某形状的轮廓，有各种颜色
眼睛： 最多两个的某形状的轮廓，有各种颜色
鼻子： 最多一个的某形状的轮廓，有各种颜色
嘴巴： 某形状的轮廓，有各种颜色
心： 某形状的轮廓，有一种颜色
脾： 某形状的轮廓，有一种颜色
肝： 某形状的轮廓，有一种颜色
肺： 某形状的轮廓，有一种颜色
胃： 某形状的轮廓，有一种颜色
肠： 多种形状的轮廓，有一种颜色
脚： 多种形状的轮廓，有各种颜色

现在我们又得出了数量、轮廓和颜色这个三个元素，他们是所有元素都有的共性，我们找到了组合山羊的最小划分，虽然在科学上这不是正确答案，但粗略地表达一只山羊这已足够。好，虽然历时不多，我们在脑海里已经解决了这个问题。





GCC 3.0 印象记

编者按：GCC 3.0 正式发布了，小编我委托虫虫和 beyond_ml 测试一下。由虫虫列出测试项目，编写测试程序；beyond_ml 在 Redhat 6.0 上分别用 GCC 2.9x 和 GCC 3.0 测试。如果您有什么新发现，不妨发 email 告诉我们。

2001-6-18 GCC 3.0 正式发布 (<http://gcc.gnu.org>)

从 1999 年 4 月开始，GCC 的含义从 GNU C Compiler 变成了 GNU Compiler Collection，支持 C、C++、Objective C、Chill、Fortan 和 Java 等语言。其中 C++ 编译器 G++ 当然是我们关注的焦点。从 GCC 网站的介绍中可以看出，G++ 作了不少改进，更好地支持 C++ 标准。我们将选出相关的内容逐一评论。我们对 GCC 3.0 的总体印象是，有不少改进，但“革命尚未成功”，还得加把劲儿。

支持使用 `using` 从基类中引入成员函数。

这是 GCC 老版本的一个大 bug 了。例如，下面这段程序虽然符合 C++ 标准，但在以前版本的 G++ 中不能被编译。

```
struct A
{
    void f() {}
};

struct B: A
{
    using A::f;
    void f(int) {}
};

void main()
{
    B b;
    b.f();
}
```

在 GCC 3.0 中，这段程序就能够顺利通过了，可以算是一个进步。（编者注：GCC 3.0 处理虚拟函数仍然失当，bug 依旧，《钻穿牛角尖：`using` 声明 vs. 虚函数》一文中有详细的分析。）

加强对嵌套类型的存取控制。

看看这个程序。

```
class X
{
private:
    typedef int Y;
```

```

};

void main()
{
    X::Y a;
}

```

按标准，这样的程序显然是错误的，`X::Y` 不可访问。然而在 GCC 2.9x 上却能编译通过。GCC 3.0 加强了对嵌套类型的管理，这个问题已经不复存在。

下面这段程序来自 C++ 标准 ISO/IEC 14882 第 184 页。

```

class C
{
    class A{};
    A *p;
    class B : A
    {
        A      *q;
        C::A   *r;      //error
        B      *s;
        C::B   *t;      //error
    };
};

```

标记为 error 的两行按标准应该出错，然而在 GCC 2.9x 和 3.0 上均编译通过，可见 GCC 对嵌套类的访问控制还不尽如人意。

改进了名字粉碎机制。

这是编译器编译、连接的具体实现，我们就不做测试了。

某些以前可以编译通过的无效转换现在都将被拒绝。

在帮助文件里特别提到了函数指针的转换，GCC 3.0 比以前更为严格。可惜我们没有发现好的例子。不过以前函数指针转换的一个大 bug，现在依然没有修复。

```

#include <vector>
template <class C> void test(C f) {}
void main()
{
    test(&std::vector<int>::push_back);
}

```

上面 `class C` 代表的是一个模板类的成员函数指针。在 GCC 2.9x 和 GCC 3.0 中，我们都会得到 'no matching function for call to `test ({unknown type})' 的编译错误，而这是一个正确的程序。

钻穿牛角尖：using 声明 vs. 虚函数 虫 虫

编者按：在用 C++ 编写程序的过程中，我们都会遇到一些不大不小的问题。说穿了不名一钱，可往往令很多高手阴沟里翻船。相信大家都有各自宝贵的经验吧？您不妨把这些经验与故事告诉我们，让大家共同分享。

问题：

下列程序输出结果是什么？

```
#include <iostream>
using namespace std;
struct A
{
    virtual void f() {cout<<"A::f"<<endl;}
};
struct B : A
{
    virtual void f() {cout<<"B::f"<<endl;}
};
struct C : B
{
    using A::f;
};
void main()
{
    C c;
    c.f();
    c.C::f();
}
```

解答：

当然，运行一下不就知道了？不过您在不同的编译器运行，结果可能千差万别。gcc 2.9x 上输出两个 B::f，gcc 3.0 上又输出两个 A::f。怎么回事呢？

c.f() 与 c.C::f() 有什么区别？这看起来像是个很简单的问题，很多人都会以为它们是等价的，然而这道题的结果恐怕又让您大跌眼镜，正确结果应该是 B::f 和 A::f。（注：这个问题改编自 C++ 标准 ISO/IEC 14882 第 169 页，有兴趣的朋友不妨看看。）在 Borland C++ Compiler 5.5.1 和 Intel C/C++ Compiler 5.0 上的结果均正确。

c.C::f() 输出 A::f 我相信大家不会感到意外，因为在类 C 中明确地有 using A::f 这一行，

`C::f` 当然就等价于 `A::f` 了，可是 `c.f()` 怎么又会调用 `B::f` 呢？查查标准吧，在 168 页有这么一句：The rules for member lookup are used to determine the final overrider for a virtual function in the scope of a derived class but ignoring names introduced by using-declarations. 大意是说，`using` 声明（`using-declaration`）和虚函数两不相干。既然不相干，`c.f()` 得到这样的结果也就不足为奇。

但是别忙。`c.f()` 是动态调用吗？我们知道，动态调用一般是以基类指针或引用的形式，但是这里的 `c` 既不是指针或引用，也不是基类，怎么会是动态调用呢？

以 Borland C++ Compiler 5.5.1 为例，我们来看看 `c.f()` 和 `c.C::f()` 的汇编代码。

```
c.f();
lea eax,[ebp-0x04]
push eax
mov edx,[ebp-0x04]
call dword ptr [edx]
pop ecx

c.C::f();
lea ecx,[ebp-0x04]
push ecx
call A::f()
pop ecx
```

不懂汇编的朋友也不用着急，只看加黑的两行即可。`c.f()` 中有 `call dword ptr [edx]` 这一句，这是读取虚拟函数表 VFT (Virtual Function Table)，进行动态调用；而在 `c.C::f()` 中直接 `call a::f()`，是静态调用，在编译阶段就已经固定下来。

`c.f()` 果然是动态调用的，这也是 Borland 和 Intel 两个编译器结果正确的关键，而 GCC 2.9x 和 3.0 就栽在这里。按我们的经验，`c.f()` 一般是静态调用，但这里却是动态，why？

注意一个关键所在，`C::f` 是继承自它的祖先类，类 `C` 本身是没有函数 `f()`。在这种情况下，`c.f()` 就必须进行动态调用，因为我们无法在编译的时候确定 `C::f`。所以如果我们把类 `C` 改为

```
struct C : B
{
    using A::f;
    virtual void f() {cout<<"C::f"<<endl;}
};
```

一旦我们重写了函数 `f`，`c.f()` 和 `c.C::f()` 就是完全等价的，编译时就静态绑定，都会输出 `C::f`。注意，这时候别被那个 `using A::f;` 迷惑了，`C` 本身有了 `f`，就不会再用祖先类的 `f` 了。

C++的不足之处讨论系列(二)：全局分析

Ian Joyner
cber 译

以下文章翻译自 Ian Joyner 所著的 *C++?? A Critique of C++ and Programming and Language Trends of the 1990s* 3/E【Ian Joyner 1996】

原著版权属于 Ian Joyner，征得 Ian Joyner 本人的同意，我得以将该文翻译成中文。因此，本文的中文版权应该属于我;-)

该文章的英文及中文版本都用于非商业用途，你可以随意地复制和转贴它。不过最好请在转贴时加上前面的这段声明。

如果有人或机构想要出版该文，请最好联系原著版权所有人及我。

另外，该篇文章已经包含在 Ian Joyner 所写的《Objects Unencapsulated》一书中（目前已经有了日文的翻译版本），该书的介绍可参见于：

http://www.prenhall.com/allbooks/ptr_0130142697.html

<http://efsa.sourceforge.net/cgi-bin/view/Main/ObjectsUnencapsulated>

<http://www.accu.org/bookreviews/public/reviews/o/o002284.htm>

Ian Joyner 的联系方式：i.joyner@acm.org
我的联系方式：cber@email.com.cn

译者前言：

要想彻底的掌握一种语言，不但需要知道它的长处有哪些，而且需要知道它的不足之处又有哪些。这样我们才能用好这门语言，避免踏入语言中的一些陷阱，更好地利用这门语言来为我们的工作所服务。

Ian Joyner 的这篇文章以及他所著的 *Objects Unencapsulated* 一书中，向我们充分的展示了 C++ 的一些不足之处，我们应该充分借鉴于他已经完成的伟大工作，更好的了解 C++，从而写出更加安全的 C++ 代码来。

【P&S 94】中提到对于类型安全的检测来说有两种假设。一种是封闭式环境下的假设，此时程序中的各个部分在编译期间就能被确定，然后我们可以对于整个程序来进行类型检测。另一种是开放式环境下的假设，此时对于类型的检测是在单独的模块中进行的。对于实际开发和建立原型来说，第二种假设显得十分有效。然而，【P&S 94】中又提到，“当一种已经完成的软件产品到达了成熟期时，采用封闭式环境下的假设就可以被考虑了，因为这样可以使得一些比较高级的编译技术得以有了用武之处。只有在整个程序都被了解的情况下，我们才可能在其上面执行诸如全局寄存器分配、程序流程分析及无效代码检测等动作。”(附：【P&S 94】Jens Palsberg and Michael I. Schwartzbach, Object-Oriented Type Systems, Wiley 1994)

C++中的一个主要问题就是：对于程序的分析过程被编译器（工作于开放式环境下的假设）和链接器（依赖于十分有限的封闭式环境下的分析）给划分开了。封闭式环境下的或是全局的分析被采用的实质原因有两个方面：首先，它可以保证汇编系统的一致性；其次，它通过提供自动优化，减轻了程序员的负担。

程序员能够被减轻的主要负担是：设计父类的程序员不再需要（不得不）通过利用虚拟函数的修饰成份（virtual），来协助编译器建立起 vtable。正如我们在“虚拟函数”中所说，这样做将会影响到软件的弹性。Vtable 不应该在一个单独的类被编译时就被建立起来，最好是在整个系统被装配在一起时一并被建立。在系统被装配（链接）时期，编译器和链接器协同起来，就可以完全决定一个函数是否需要在 vtable 中占有一席之地。除上述之外，程序员还可以自由地使用在其他模块中定义的一些在本地不可见的信息；并且程序员不再需要维护头文件的存在了。

在 Eiffel 和 Object Pascal 中，全局分析被应用于整个系统中，决定真正的多态性的函数调用，并且构造所需的 vtable。在 Eiffel 中，这些是由编译器完成的。在 Object Pascal 中，Apple 扩展了链接器的功能，使之具有全局分析的能力。这样的全局分析在 C/Unix 环境下很难被实现，所以在 C++ 中，它也没有被包含进去，使得负担被留给了程序员。

为了将这个负担从程序员身上移除，我们应该将全局分析的功能内置于链接器中。然而，由于 C++一开始的版本是作为一个 Cfront 预处理器实现的，对于链接器所做的任何必要的改动不能得到保证。C++的最初实现版本看起来就像一个拼凑起来的东西，到处充满着漏洞。【译者认为：这也太过分了吧：】C++的设计严格地受限于其实现技术，而不是其他（例如没有采用好的程序语言设计原理等），因为那样就需要新的编译器和链接器了。也就是说，现在的 C++发展严格地受限于其最初的试验性质的产品。

我现在确信这种技术上的依赖关系（即 C++依赖于早先的 C）严重地损害了 C++，使之不是一个完整意义上的面向对象的高级语言。一个高级语言可以将簿记工作从程序员身上接手过去，交给编译器去完成，这也是高级语言的主要目的。缺乏全局（或是封闭式环境下的）分析是 C++的一个主要不足，这使得 C++ 在和 Eiffel 之类语言相比时显得十分地不足。由于 Eiffel 坚持系统层次上的有效性及全局分析，这意味着 Eiffel 要比 C++ 显得有雄心多了，但这也是 Eiffel 产品为什么出现地这么缓慢的主要原因。

Java 只有在需要时才动态地载入软件的部分，并将它们链接起来成为一个可以运行的系统。也因而使得静态的编译期间的全局分析变成不可能的了（因为 Java 被设计成为一个动态的语言）。然而，Java 假设所有的方法都是 virtual 的，这也就是为什么 Java 和 Eiffel 是完全不同的工具的一个原因。关于 Eiffel，可以参见于 Dynamic Linking in Eiffel(DLE)。

Traits：类型的 else-if-then 机制

Andrei Alexandrescu

myan 译

编者按：经Andrei Alexandrescu先生授权，从本期开始，我们推出Generic<Programming>系列。这一系列文章部分被收入Andrei Alexandrescu先生的大作*Modern C++ Design*一书中。这篇文章非常出色，然而难度比较大，小编在myan的翻译的基础上，请cber、plpliuly和虫虫帮忙修改润色，并且在【】加入了小编的注解，希望能帮助大家更好的理解本文。由于水平有限，理解可能有失误之处，大家最好能看看原文，与译文对照比较学习，如果有什么心得，记着来email讨论。本文最初发表在C++ Report 2000年第6期上，原文见http://www.joopmag.com/html/from_pages/crarticle.asp?ID=402。

什么是traits？为什么人们老爱提起它，并认为它是C++泛型编程中的重要技术呢？

简单来说，traits的重要性就在于能在编译时间（compile-time）通过类型（type）确定函数的调用，尽管我们往往习惯于在运行时间（run-time）通过值（value）来确定【traits是类型驱动（type-driven），依照具体的类型产生相应的模板类或模板函数，而比如多态则是在运行时间通过值来确定调用函数的，会付出性能上的代价】。更妙的是，traits能让您根据其产生环境（context）作出类型判定，使得代码更清晰可读，更易于维护，这正应了那句曾解决了软件工程界无数难题的名言“extra level of indirection（额外的中间层）”。如果正确使用traits，我们在享受上述好处的同时，亦不必付出性能、安全及耦合性等方面的代价。

例子

Traits不仅仅是纯粹的泛型编程的工具，对一些特定问题的解决亦有很大的帮助。不信？先看下面这个例子吧。

假设我们正在编写一个关系数据库应用程序。开始，我们可能会使用数据库供应商所提供的本地API库来访问数据，当然过不了多久就会发现，为了提高灵活性，使其更好地适于手中需要解决的问题，我们有必要写一些函数来包装这些原始的API。生活的色彩也因此而来，不是吗？【使用供应商提供的原始API进行编程，程序可读性往往极差，还记得WINDOWS SDK编程中那个庞大的switch吗？于是各种进行高层包装的应用框架才大行其道。】

一般这些API能提供一些基本的功能，用以把原始数据（比如行集合或者查询结果），从游标（Cursor）处传送到内存中。OK，现在我们要在不暴露底层细节的前提下，写一个更高层次的函数，用于从一列中取出某个值。其形式大概是下面这个样子（我们假设这些API都是以DB_和db_开头）：

```
// Example 1: Wrapping a raw cursor int fetch
// operation.
// Fetch an integer from the
//     cursor "cr"
```

```

//      at column "col"
//      in the value "val"
void FetchIntField(db_cursor& cr, unsigned int col, int& val)
{
    // Verify type match
    if (cr.column_type[col] != DB_INTEGER)
        throw std::runtime_error("Column type mismatch");
    // Do the fetch
    if (!db_access_column(&cr, col))
        throw std::runtime_error("Cannot transfer data");
    db_integer temp;
    memcpy(&temp, cr.column_data[col], sizeof(temp));
    // Required by the DB API for cleanup
    db_release_column(&cr, col);
    // Convert from the database native type to int
    val = static_cast<int>(temp);
}

```

这种接口函数都是我们曾经写过的：凌乱不堪、极度缺乏灵活性（highly imperative）和大量低层细节纠缠不清……而这还仅仅是一个简单的例子。我们所期望的 FetchIntField 函数，应该具有更高层次的抽象性 不用考虑过多细节就能从某列的游标处析取出一个整数。

对于这种非常有用的函数，我们当然希望尽可能地重用（reuse）。那怎么做呢？泛型化重要的一步就是让它不仅仅能处理整数类型 int【不仅仅局限于具体的类型，这是重用性，也是抽象性的体现】。为此，我们得理解针对 int 类型的这部分代码。但是首先我们注意一下，DB_INTEGER 和 db_interger 是什么意思，从哪儿掉下来的呢？一般来说，关系数据库的供应商们会在 API 中提供类型映射的辅助措施：为其支持的每种类型定义一个符号常数及一些 typedef 或简单的结构，把数据库类型对应到 C/C++ 的类型上。下面是假想的数据库 API 头文件：

```

#define DB_INTEGER 1
#define DB_STRING 2
#define DB_CURRENCY 3
...
typedef long int db_integer;
typedef char     db_string[255];
typedef struct {
    int integral_part;
    unsigned char fractionary_part;
} db_currency;
...

```

我们来写一个从游标处析取 double 值的函数 FetchDoubleField，作为我们泛型化的第一步。数据库提供的类型是 db_currency，不过我们需要以 double 的形式来操作。

FetchDoubleField 跟 FetchIntField 几乎就是一对双胞胎，非常相似。在下面的例子里，我们将以加粗的形式标记出这对“双胞胎”不同的部分。

```
// Example 2: Wrapping a raw cursor double fetch operation.
void FetchDoubleField(db_cursor& cr, unsigned int col, double& val)
{
    if (cr.column_type[col] != DB_CURRENCY)
        throw std::runtime_error("Column type mismatch");
    if (!db_access_column(&cr, col))
        throw std::runtime_error("Cannot transfer data");
db_currency temp;
memcpy(&temp, cr.column_data[col], sizeof(temp));
db_release_column(&cr, col);
val = temp.integral_part + temp.fractionary_part / 100.;
}
```

这对“双胞胎”相似吧？不过我们可没兴趣为数据库支持的每种类型都写一段相似的代码。如果这些 FetchIntegerField、FetchDoubleField 以及其他 Fetch... 可以合起来只写一次，那多好啊。

那么，让我们来列举一下这对“双胞胎”的不同之处。

- 输入类型：double/int
- 内部使用类型：db_currency/db_integer
- 常数：DB_CURRENCY/DB_INTEGER
- 算法：表达式/静态转换 static_cast

输入类型（double/int）跟其他几点之间没有明显的对应规则，这完全取决于数据库供应商所提供的约定（convention）和定义（definition）。模板机制本身则爱莫能助，它还不具备如此高级的类型推理能力【由输入类型来选择常数，这的确非模板本身能力所及】。因为我们处理的是原始类型，继承机制亦不能把不同的类型联系起来。由于受 API 的限制以及问题本身的底层特性，乍一看，泛型方法已经“疑无路”了；其实，我们还可以柳暗花明。【怎样处理尽可能多的类型？很多人的想法是写一个基类，然后其他类均从此基类继承，于是我们编写的函数均针对基类即可，这是很多类库采用单根结构的原因之一。然而这种方法耦合性较高，而且对于 int 等原生类型无效（也不是绝对没有办法，比如 Java 是写一个原生类型的包装类，当然效率及可读性都不及本文的方法）。】

初窥 TRAITS 门径

Traits 正是解决此问题的灵丹妙药。它能把依赖某种类型（比如上面的 double/int）根据不同的结构及行为作出相应调整的代码段联结起来。为此，Traits 依赖于 C++ 的语言特性：显式模板特化（explicit template specialization），这种特性能让我们为每种特定的类型提供特定的模板类实现。

```
// Example 3: A traits example
//
```

```

template <class T>
class SomeTemplate
{
    // generic implementation (1)
    ...
};

template <>
class SomeTemplate<char>
{
    // implementation tuned for char (2)
    ...
};

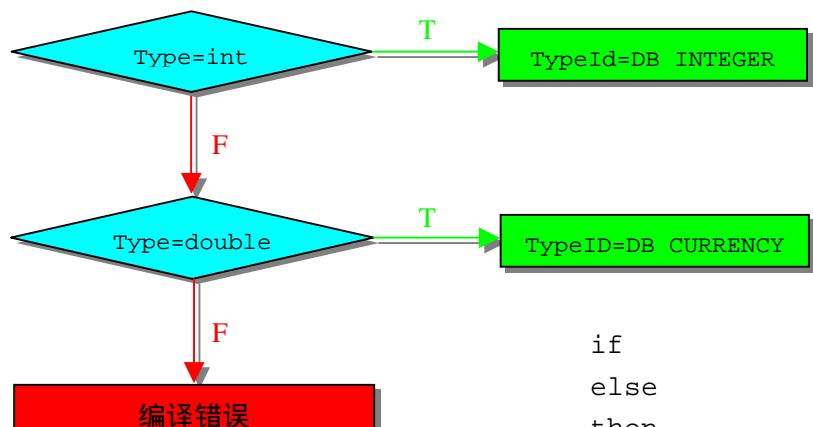
SomeTemplate<int> a;           // will use (1)
SomeTemplate<char*> b;         // will use (1)
SomeTemplate<char> c;          // will use (2)

```

如果用 `char` 实例化类模板 `SomeTemplate`，编译器就会采用显式特化的方案 (1)。当然对于其他类型，编译器则会实例化通用模板 (2)。这看起来很像 `if` 语句，只不过是由类型驱动罢了。通常最通用的模板（相当于 `else` 部分）最先定义，`if` 语句靠后一点。我们甚至可以根本不提供通用模板，只提供特化部分，使其他的实例化都导致编译错误。【这里把编译器的选择过程类比为 `if-then-else` 语句：如果(`if`)存在特化，就(`then`)选择特化的版本；否则(`else`)选择通用的版本。但由于通用的部分需要先声明，所以看起来 `else` 部分反而在前面，于是就成了今天的主题：类型的 `else-if-then` 机制。】

现在我们把这个语言特性与手头的问题联系起来，实现一个用读取的类型参数化的模板函数 `FetchField`。在此函数中，我们要能作如下推断：假定一个整数常量 `TypeId`，如果要获取的类型是 `int`，那它的值是 `DB_INTEGER`；如果要获取的类型是 `double`，那么，它的值是 `DB_CURRENCY`；否则，将会发生编译错误。类似地，

我们也可以根据获取的类型的不
同，来操作不同的数据类型 (`db_integer/db_currency`) 和不同的转换算法。【这是假想的
编译器推理过程，不妨参考一下编者补充的图。】



让我们用显式模板特化来解决这个问题，利用模板类来掩盖前面提到的那些不同之处，并针对 `int` 和 `double` 来显式特化这个模板类。每个特化都为这些不同之处提供相同的名字【比如 `db_integer` 和 `db_currency` 都被统一成 `DbNativeType`，不同的转换算法都被统一成 `Convert`】。

```

// Example 4: Defining DbTraits
//
// Most general case not implemented
template <typename T> struct DbTraits;
// Specialization for int
template <>
struct DbTraits<int>
{
    enum { TypeId = DB_INTEGER };
    typedef db_integer DbNativeType;
    static void Convert(DbNativeType from, int& to)
    {
        to = static_cast<int>(from);
    }
};

// Specialization for double
template <>
struct DbTraits<double>
{
    enum { TypeId = DB_CURRENCY };
    typedef db_currency DbNativeType;
    static void Convert(const DbNativeType& from, double& to)
    {
        to = from.integral_part + from.fractionary_part / 100.;
    }
};

```

现在写 `DbTraits<int>::TypeId` 得到 `DB_INTEGER` ,写 `DbTraits<double>::TypeId` 就得到 `DB_CURRENCY` ,写 `DbTraits<anything else>::TypeId` 呢 ? 得到编译错误 ,因为模板类本身只声明而尚未定义。【注意这里 traits 的写法 :让通用的部分为空 ,在细节部分根据每个类型写出特化的模板类 ,而在抽象层面提供统一的接口。】

有没有悟到点什么 ? 现在看看我们怎么利用 `DbTraits` 来实现一个通用的 `FetchField` 函数 ,把所有的不同之处 枚举类型、数据库原生类型、转换算法 通通掩盖在 `DbTraits` 的保护伞下。这样 ,我们的函数只包括 `FetchIntegerField` 和 `FetchDoubleField` 相同的部分。

```

// Example 5: A generic, extensible FetchField using DbTraits
//
template <class T>
void FetchField(db_cursor& cr, unsigned int col, T& val)
{
    // Define the traits type

```

```

typedef DbTraits<T> Traits;
if (cr.column_type[col] != Traits::TypeId)
    throw std::runtime_error("Column type mismatch");
if (!db_access_column(&cr, col))
    throw std::runtime_error("Cannot transfer data");
typename Traits::DbNativeType temp;
memcpy(&temp, cr.column_data[col], sizeof(temp));
Traits::Convert(temp, val);
db_release_column(&cr, col);
}

```

OK，搞定！我们所实现的正是一个 traits 类模板。

Traits 依赖于显式模板特化，把代码中类型相关的不同之处封装在统一的接口中，而这种接口跟一个普通的 C++ 类没有什么两样，可以包含嵌套类型、成员函数、成员变量，模板化的用户代码可以通过 traits 模板类公开的接口间接访问。

这样的 traits 接口通常是隐式的——这是 traits 类模板和使用它的代码之间的约定。隐式接口访问比函数表征【函数表征就是其参数类型、个数及返回值类型】更为宽松，比如尽管 `DbTraits<int>::Convert` 和 `DbTraits<double>::Convert` 这两个函数的表征不同，但由于遵从调用代码与它们之间的约定，因此都能正常运行。【这里我们可以充分看出 traits 跟传统多态的不同之处。多态的实现方式是针对基类提供函数，而 traits 是透过中间层，访问事先约定好的类型和函数，比如 `DbTraits<T>::Convert` 就是约定好的函数名。】

Traits 模板类在一组高层次上有意义的设计选择的基础上建立起统一的接口，但在实现细节上（类型、值、算法）又有所区别。由于 traits 记录了一个概念（concept），一组相互关联的决策，因此可能重用在类似的环境（context）中。在此例中，我们就可以在其他的数据库资料操作（比如把数据写回游标）中重用 `DbTraits`。

定义：Traits 模板是一种可以为一组相互关联而类型不同的设计选择提供统一的符号化接口（可能显式特化）的模板类。

Definition: A traits template is a template class, possibly explicitly specialized, that provides a uniform symbolic interface over a coherent set of design choices that vary from one type to another.

TRAITS AS ADAPTERS (做适配子的 Traits)

前面我们已经把数据库说得够多了，现在我们换个流行的话题——smart pointer。

假设我们在开发一个模板类 `SmartPtr`。对于 smart pointer 来说，最棒的是它看起来跟普通的指针没什么两样，却能使内存管理自动化；而不太好对付的是实现它们的那些不同寻常的代码（与 smart pointers 相关的技术跟稀奇古怪的巫术没什么两样）。这一残酷的事实告诉我们一个重要的实践经验：我们最好尽可能一劳永逸，写出一个出色的、具有工业强度的 smart pointer 来满足我们绝大

部分的要求。此外，常常我们不能靠修改类来配合 smart pointer，于是我们的 SmartPtr 必须很有弹性。

很多类层次（class hierarchy）都使用引用计数（reference counting），提供相关函数管理对象生存期。然而由于引用计数的实现并没有统一的标准，各个 C++ 库供应商的实现在语法及语义上可能都有所不同。比如，在我们的程序中，可能有下列两种接口：

- 我们的大部分类都实现 RefCounted 接口：

```
class RefCounted
{
public:
    void IncRef() = 0;
    bool DecRef() = 0; // if you DecRef() to zero
                      // references, the object is destroyed
                      // automatically and DecRef() returns true
    virtual ~RefCounted() {}
};
```

- 由第三方提供的 Widget 类使用了一个略为不同的接口：

```
class Widget
{
public:
    void AddReference();
    int RemoveReference(); // returns the remaining
                          // number of references; it's the client's
                          // responsibility to destroy the object
    ...
};
```

我们当然不想维护两个 smart pointer 类。我们希望在我们自己的和基于 Widget 的继承层次之间共享一个统一的 SmartPtr 后端。一个基于 traits 的解决方案就能提供统一的语法及语义接口，来包装这两个略为不同的接口，也就是先建立通用模板支持 RefCounted 接口，然后为 Widget 建立特化的版本：

```
// Example 6: Reference counting traits
//
template <class T>
class RefCountingTraits
{
    static void Refer(T* p)
    {
```

```

        p->IncRef(); // assume RefCounted interface
    }
    static void Unrefer(T* p)
    {
        p->DecRef(); // assume RefCounted interface
    }
};

template <>
class RefCountingTraits<Widget>
{
    static void Refer(Widget* p)
    {
        p->AddReference(); // use Widget interface
    }
    static void Unrefer(Widget* p)
    {
        // use Widget interface
        if (p->RemoveReference() == 0)
            delete p;
    }
};

```

在 SmartPtr 里，我们这样使用 RefCountingTraits：

```

template <class T>
class SmartPtr
{
private:
    typedef RefCountingTraits<T> RCTraits;
    T* pointee_;
public:
    ...
    ~SmartPtr()
    {
        RCTraits::Unrefer(pointee_);
    }
};

```

您可能会提出，在上例中为什么不直接特化针对 Widget 的 SmartPtr 的构造及析构函数呢？为什么要模板特化 traits 而不是 SmartPtr 本身呢？这样还可以少掉一个多余的类呢。对此例而言，您说的都对，但是却有些我们不得不注意的缺点：

- 缺乏扩展性。如果 SmartPtr 需要再加一个参数，那就没辙了，我们不能针对 Widget 和

一个任意类型 U 部分特化 SmartPtr<T, U>的成员函数。顺便提一句，对于很多模板参数来说，smart pointers 可是个不错的考虑（见 Van Horn 所举的丰富的例子¹）；【作者所说做不到的情形如下：

```
template <class T, class U>
class Demo {
public:
    void dostuff() {...} //Generic 版本 dostuff()
};

template<> void Demo<char, int>::dostuff() {...}
//OK，针对成员函数进行特化
template <class U> void Demo<Widget, U>::dostuff() {...}
//error，由于不存在 Demo<Widget, U>的部分特化定义，所以编译失败。
```

不过不知道是不是理解的问题，这么说未免也过于绝对。比如前面这个编译出错的地方，我们只需要在前面加入特化版本类的声明就行了，也就是加入：

```
template <class U> Demo<Widget, U>
{
public:
    void dostuff();
}
```

当然这个办法很白痴，但似乎也可以达到目的。】

- 代码不够清晰。trait 有个名字，能很好地组织相关的东西，因此使代码更易懂。相形之下，直接特化 SmartPtr 的成员函数难免留下斧凿的痕迹，生硬而不自然；
- 对同一类型不能使用多种 traits。

用继承机制的解决方案，也存在上述缺陷²，更不用说继承本身有很多值得注意的问题。解决这样的变体问题，使用继承实在太笨重了。此外，通常用以取代继承方案的另一种经典机制 containment，用在这里也显得繁琐不堪。相反，traits 方案简洁明了，物合其用。

Traits 扮演的一个重要角色就是接口胶合剂（interface glue）——可适应性极强的通用适配子。如果不同的类对一个给定概念的实现略微不同，traits 可以把它们统一在一个公共接口下。

对一个给定类型提供多种 TRAITS

现在，我们假设所有的人都很喜欢这个 SmartPtr 模板类，直到有一天，在多线程应用程序里开始出现神秘的 bug，美梦破灭了。后来发现罪魁祸首是 Widget，它的引用计数函数并不是线程安全的。现在我们不得不亲自实现 Widget::AddReference 和 Widget::RemoveReference，最合理的位置应该是在 RefCountingTraits 中，打上个补丁吧：

```
// Example 7: Patching Widget's traits for thread safety
//
```

```

template <>
class RefCountingTraits<Widget>
{
    static void Refer(Widget* p)
    {
        Sentry s(lock_); // serialize access
        p->AddReference();
    }
    static void Unrefer(Widget* p)
    {
        Sentry s(lock_); // serialize access
        if (p->RemoveReference() == 0)
            delete p;
    }
private:
    static Lock lock_;
};

```

不幸的是，虽然重新编译、测试之后运行正确了，但是程序慢得像蜗牛。仔细分析之后发现，刚才的所作所为往程序里塞了一个糟糕的瓶颈。实际上只有少数几个 Widget 是需要能够被好几个线程访问的，余下的绝大多数 Widget 都是只被一个线程访问的。

我们要做的就是告诉编译器，按我们的需求分别使用多线程 traits 和单线程 traits 这两个不同版本。你的代码主要使用单线程 traits。

如何告诉编译器使用那个 traits？这么干：把 traits 作为附加模板参数传给 SmartPtr。缺省情况下传递以前那个 traits 模板，而用特定的类型实例化特定的模板。

```

template <class T, class RCTraits = RefCountingTraits<T> >
class SmartPtr
{
    ...
};

```

对单线程版的 RefCountingTraits<Widget> 不做改动，而把多线程版放在一个单独的类中：

```

class MtRefCountingTraits
{
    static void Refer(Widget* p)
    {
        Sentry s(lock_); // serialize access
        p->AddReference();
    }
}

```

```

    static void Unrefer(Widget* p)
    {
        Sentry s(lock_); // serialize access
        if (p->RemoveReference() == 0)
            delete p;
    }
private:
    static Lock lock_;
} ;

```

现在可将 `SmartPtr<Widget>` 和 `SmartPtr<Widget, MtRefCountingTraits>` 分别用于单线程和多线程。OK 了！就跟 Scott Meyers 可能会说的那样，“你要是没体会过快乐，就不知道怎么找乐。”

如果一种类型只要一个 trait 由可以应付，那么只要用显式模板特殊化就够了。现在即使一个类型需要多个 trait 来应付我们也搞得定。所以，traits 必须能够从外界塞进来，而不是在内部“算出来”。一个应当谨记的惯用法是提供一个 traits 类作为最后一个模板参数。缺省的 traits 通过模板参数缺省值给定。

定义：一个 traits 类（与 traits 模板类相对）或者是一个 traits 模板类的实例，或者是一个与 traits 模板类展现出相同接口的单独的类。

Definition: A traits class (as opposed to a traits template) is either an instantiation of a traits template, or a separate class that exposes the same interface as a traits template instantiation.

问题

我们对 traits 的讨论刚刚开始。今后的专栏中，我将论述各种情形下的 traits，多用途 (general-purpose) traits，以及广层次 (hierarchy-wide) traits 能让我们不仅仅为某一个类，而且能一次性为整个层次或整个子层次定义 traits。

但先提个问题。在我们多线程的代码中仍然有缺乏效率地方，能找出来吗？如何解决这个问题？

致谢

感谢 Scott Meyers 提出最初想法和专栏的名字，感谢 Sorin Jianu 和 Herb Sutter 仔细审阅了全文。

参考文献

1. K. S. Kevin S. Van Horn's Smart Pointers, http://www.xmission.com/~ksvhsoft/code/smart_ptrs.html.
2. Sutter, H. "Uses and Abuses of Inheritance, Part 2," C++ Report, 11(1): 58–61, Jan. 1999.

STL 有序容器武道会

beyond_ml

编者按：尺有所短，寸有所长，对于 STL 中的容器也不例外。那么，怎样使用它们才能扬长避短呢？通过本文的评测，相信大家都可以得到很多启发。小编以为，本文最重要的是结论和分析，程序并非重点，有兴趣的朋友，完全可以按照自己的习惯编写一个。值得注意的是，同样的测试程序，在 SGI、RogueWave 和 Plauger 分别实现的三种 STL 版本下（分别分发在 GCC、Borland C++ Compiler 和 Microsoft Visual C++ 中），其结论可能大相径庭。本文采用的是 SGI-STL。更多的宝藏，就有待大家进一步发掘了。

Vector：矢量容器，C++容器模板中的大哥大，属于一个加强版的队列。之所以这样说，是因为它不但有队列形式的索引，还能动态的添加扩充，使用尤其广泛，并在许多经典教材中被作为重点介绍。

他的特点是：把被包含的对象以数组的形式存储，支持索引形式的访问（这种访问速度奇快无比）。但由此也产生了一个问题，由于数据存储形式的固定化，势必导致内存的连续分布，在这种情况下，你如果想在他中间部位 insert 对象的话，会让你吃尽苦头。

Deque：英文原意“double-ended-queue”。这是 C++有序容器中闻名遐迩的双向队列。他在设计之初，就为从两端添加和删除元素做了特殊的优化。同样也支持随机访问，也有类似于 vector 的[]操作符，请大家不要因此就把他和 vector 混为一谈。

从本质上讲，他使用了 MAP 的结构和方法来存储对象。存储空间被化整为零，许多小的连续空间增强了它的灵活性，因此，从 deque 两端添加、删除元素是十分方便的。最重要的一点：如果在不知道内存具体需求的时候，使用 deque 绝对是比 vector 好的，具体怎么好，我们可以从下面的测试结果中看出来。另外插一句，deque 和 vector 有点像双胞胎，大多数情况下，deque 和 vector 是可以互换使用的。

List：容器模板中的双向链表。设计他的目的可能就是为了在容器中间作添加、删除操作吧，他添加、删除的速度快，正所谓有得必有失，他元素访问的速度可不敢恭维，而且没有[]操作。即便你只是从前到后的 travels，也不见得就能快多少。

他的最大特点是结构灵活，机动性强。本身是以链表的形式储存对象数据，当随机添加、删除元素时，在速度上占有明显的优势。并且，由于内存中分配的空间不连续，他对资源的要求也十分的低。尤其使用大对象的时候，这可是一个不错的选择。

“本次测试使用的测试程序由 GCC3.0 在 Redhat6.1 上编译通过，使用 CPU 时钟频率为基本计时单位。好，比武正式开始：”

➤ 测试 1：内存管理：

测试方法：人为造成这些模板存储空间的数据溢出，比较他们在这种不利情况下的系统消耗。系统消耗在这里指：对象构造函数、拷贝函数、析构函数的调用次数。测试程序如下：

```
#include <vector>
#include <deque>
#include <list>
#include <iostream>
#include <ctime>
using namespace std;

#define TEST(type, f) test(type<Count>(), &type<Count>::f)

class Count
{
private:
    static int cons, des, copy, ass;
public:
    static const int max = 100000;

    Count() {++cons;}
    ~Count() {++des;}
    Count(const Count&) {++copy;}
    void operator =(const Count&) {++ass;}

    static void clear()
        {cons = des = copy = ass = 0;}
    friend class Report;
};

class Report
{
    static Report nr;
    Report() {} // Private constructor
public:
    ~Report()
    {
        cout << "\n-----\n"
        << "Creations: " << Count::cons
        << "\nCopy-Constructs: " << Count::copy
```

```

        << "\nAssignments: " << Count::ass
        << endl;
    }
};

int Count::cons=0, Count::des=0, Count::copy=0, Count::ass=0;
Report Report::nr;

template <class T, class C>
void test(T container, C (T::*f)(const typename T::value_type &))
{
    Count c;
    clock_t t = clock();
    for(int i = 0; i < Count::max; ++i) (container.*f)(c);
    cout << "Total time: " << clock() - t << endl;
}

int main()
{
    TEST(vector, push_back);           //测试 vector
//    TEST(deque, push_back);          //测试 deque
//    TEST(list, push_back);           //测试 list
}

```

目标：插入 100000 个相同的 Noisy 对象。（之所以要插入这么多，就是为了引起其内部存储空间的溢出。）正常情况下，我们期望一次构造函数，100000 次 copy 函数，和 100001 次的析构函数。

Vector 测试结果：

Total time: 140000

Creations: 1 (构造函数调用)

Copy-Constructs: 231071 (拷贝函数调用)

结果分析：我只是插十万个对象而已，怎么会调用 231071 次拷贝函数？原来，每当容器 Vector 建立的时候，他会为数据预先开辟一个保留空间，这个空间通常不是很大，在使用过程中，如果实际使用空间超过了这个限定，vector 会再次申请内存，然后将原有的数据和新的数据一起转移到新的空间里去，这样周而复始，所以，vector 搬家的动静是很大的。（但大家如果有兴趣的话，可以试试让 Vector 在测试之前，先 reserve 足够的空间，那结果可就大不一样了哦）另外，从这个结果，我们还可以分析一下它 push_back 的速度，有了上面的分析，我想大家不难做出这样的判断：真正影响 vector 的 push_back 操作速度的主要因素就是上面提到的：存储空间溢出问题。否则，在插入数据的速度

上，vector 决不会输给其它的有序容器。

Deque 测试结果：

Total time: 30000

Creations: 1

Copy-Constructions: 100195

结果分析：嗯，结果还是不错。不过那多出来的 195 个也不太好么。多出来的一点原因如下：deque 在内存中虽然不是大群居，但也是小聚居。并且使用 map 的结构来索引每一个对象以及他们的顺序。这也会导致在添加数据的时候会有额外的操作。push_back 的速度十分令人满意。

List 测试结果：

Total time: 140000

Creations: 1

Copy-Constructions: 100000

结果分析：list 在内存中完全是离散的，没有必要为分配多的内存而进行任何额外的负担，而这本身更是它的长处所在，可以从结果中看出没有任何多余的操作。但 list 的 push_back 速度实在一般，这次不是因为额外操作，而是因为在分配离散内存空间时所浪费的时间。

（编者注：上面的测试结果显示 vector<deque<list，而在实际使用中，还有其他因素。比如 vector 虽然“搬家”次数多，然而由于连续分配空间，在元素对象的构造函数不是十分复杂的情况下，速度往往还更快。如果再考虑到析构，vector 释放连续空间的速度远远大于 list 释放离散空间的速度。另外在 RogueWave STL 中，deque 的结果跟 list 是一样的。）

有了上面的一些基本概念，是不是对 Vector、deque、list 的基本运作已经有了一个大致的了解？是否明白了影响他们速度的主要原因？的确，如果单从访问、插入的角度去看问题，他们的速度优势应该是这样排列的：vector>deque>list，但在处理空间溢出的问题时，他们的表现却应该是倒过来的。也许这就是 C++ 版的鱼和熊掌的故事吧，呵呵，谈到这里，我们想下面的比较已经不重要了，也许，你已经有了一个自己的结果。那就让我们再看看下面这个程序，如果不介意的话，我会把所有的结果同时列出来，这样可以方便我的比较。

```
// SequencePerformance.cpp
```

```
#include <vector>
```

```
#include <queue>
#include <list>
#include <iostream>
#include <string>
#include <typeinfo>
#include <ctime>
#include <cstdlib>
using namespace std;
class FixedSize
{
    int x[20];
} fs;

template<class Cont>
struct InsertBack
{
    void operator()(Cont& c, long count)
    {
        for(long i = 0; i < count; i++)
            c.push_back(fs);
    }
    char* testName() { return "InsertBack"; }
};

template<class Cont>
struct InsertFront
{
    void operator()(Cont& c, long count)
    {
        long cnt = count * 10;
        for(long i = 0; i < cnt; i++)
            c.push_front(fs);
    }
    char* testName() { return "InsertFront"; }
};

template<class Cont>
struct InsertMiddle
{
    void operator()(Cont& c, long count)
    {
        typename Cont::iterator it;
        long cnt = count / 10;
```

```
        for(long i = 0; i < cnt; i++)
        {
            it = c.begin();
            it++;
            c.insert(it, fs);
        }
    }
    char* testName() { return "InsertMiddle"; }
};

template<class Cont>// Not for list
struct RandomAccess
{
    void operator()(Cont& c, long count)
    {
        int sz = c.size();
        long cnt = count * 100;
        for(long i = 0; i < cnt; i++)
            c[rand() % sz];
    }
    char* testName() { return "RandomAccess"; }
};

template<class Cont>
struct Traversal
{
    void operator()(Cont& c, long count)
    {
        long cnt = count / 100;
        for(long i = 0; i < cnt; i++)
        {
            typename Cont::iterator it = c.begin(),
            end = c.end();
            while(it != end) it++;
        }
    }
    char* testName() { return "Traversal"; }
};

template<class Cont>
struct Swap
{
    void operator()(Cont& c, long count)
```

```
{  
    int middle = c.size() / 2;  
    typename Cont::iterator it = c.begin(),  
    mid = c.begin();  
    it++; // Put it in the middle  
    for(int x = 0; x < middle + 1; x++)  
        mid++;  
    long cnt = count * 10;  
    for(long i = 0; i < cnt; i++)  
        swap(*it, *mid);  
}  
char* testName() { return "Swap"; }  
};  
  
template<class Cont>  
struct RemoveMiddle  
{  
    void operator()(Cont& c, long count)  
    {  
        long cnt = count / 10;  
        if(cnt > c.size())  
        {  
            cout << "RemoveMiddle: not enough elements"  
            << endl;  
            return;  
        }  
        for(long i = 0; i < cnt; i++)  
        {  
            typename Cont::iterator it = c.begin();  
            it++;  
            c.erase(it);  
        }  
    }  
    char* testName() { return "RemoveMiddle"; }  
};  
  
template<class Cont>  
struct RemoveBack  
{  
    void operator()(Cont& c, long count)  
    {  
        long cnt = count * 10;  
        if(cnt > c.size())
```

```

    {
        cout << "RemoveBack: not enough elements"
        << endl;
        return;
    }
    for(long i = 0; i < cnt; i++)
        c.pop_back();
}
char* testName() { return "RemoveBack"; }

};

template<class Op, class Container>
void measureTime(Op f, Container& c, long count)
{
    string id(typeid(f).name());
    bool Deque = id.find("deque") != string::npos;
    bool List = id.find("list") != string::npos;
    bool Vector = id.find("vector") != string::npos;
    string cont = Deque ? "deque" : List ? "list"
        : Vector ? "vector" : "unknown";
    cout << f.testName() << " for " << cont << ":" ;
    // Standard C library CPU ticks:
    clock_t ticks = clock();
    f(c, count); // Run the test
    ticks = clock() - ticks;
    cout << ticks << endl;
}

typedef deque<FixedSize> DF;
typedef list<FixedSize> LF;
typedef vector<FixedSize> VF;
int main()
{
    srand(time(0));
    long count = 10000;
    DF deq;
    LF lst;
    VF vec, vecres;
    vecres.reserve(count); // Preallocate storage
    measureTime(InsertBack<VF>(), vec, count);
    measureTime(InsertBack<VF>(), vecres, count);
    measureTime(InsertBack<DF>(), deq, count);
    measureTime(InsertBack<LF>(), lst, count);
}

```

```
// Can't push_front() with a vector:  
//! measureTime(InsertFront<VF>(), vec, count);  
measureTime(InsertFront<DF>(), deq, count);  
measureTime(InsertFront<LF>(), lst, count);  
measureTime(InsertMiddle<VF>(), vec, count);  
measureTime(InsertMiddle<DF>(), deq, count);  
measureTime(InsertMiddle<LF>(), lst, count);  
measureTime(RandomAccess<VF>(), vec, count);  
measureTime(RandomAccess<DF>(), deq, count);  
// Can't operator[] with a list:  
//! measureTime(RandomAccess<LF>(), lst, count);  
measureTime(Traversal<VF>(), vec, count);  
measureTime(Traversal<DF>(), deq, count);  
measureTime(Traversal<LF>(), lst, count);  
measureTime(Swap<VF>(), vec, count);  
measureTime(Swap<DF>(), deq, count);  
measureTime(Swap<LF>(), lst, count);  
measureTime(RemoveMiddle<VF>(), vec, count);  
measureTime(RemoveMiddle<DF>(), deq, count);  
measureTime(RemoveMiddle<LF>(), lst, count);  
vec.resize(vec.size() * 10); // Make it bigger  
measureTime(RemoveBack<VF>(), vec, count);  
measureTime(RemoveBack<DF>(), deq, count);  
measureTime(RemoveBack<LF>(), lst, count);  
} ///:~
```

使用 SGI STL , g++3.0 , redhat 6.1 编译通过结果如下：

```
InsertBack for vector: 50000  
InsertBack for deque: 20000  
InsertBack for list: 30000  
InsertFront for deque: 200000  
InsertFront for list: 270000  
InsertMiddle for vector: 11970000  
InsertMiddle for deque: 10000  
InsertMiddle for list: 0  
RandomAccess for vector: 740000  
RandomAccess for deque: 1870000  
Traversal for vector: 320000  
Traversal for deque: 4710000  
Traversal for list: 7130000  
Swap for vector: 90000
```

```
Swap for deque: 100000
Swap for list: 170000
RemoveMiddle for vector: 14190000
RemoveMiddle for deque: 10000
RemoveMiddle for list: 50000
RemoveBack for vector: 10000
RemoveBack for deque: 40000
RemoveBack for list: 180000
```

面对这个结果，心里有数的您，想来也不会在插入、删除的操作消耗上有什么疑问，但这里有几个比较的结果我们还是要一起来分析一下的。

➤ iterator 访问速度

使用 iterator 访问 1000000 次的测试结果：

```
Traversal for vector: 330000
Traversal for deque: 4710000
Traversal for list: 7130000
```

从读数据的速度来看，list 的表现十分让人迷惑不解。对此，我还想不到什么好的解释，也许和程序运行时主机的内存状态有关吧。从总体上说，Vector 和 list 的表现可以说是不分伯仲，但我个人的观点是 vector 肯定要好一些，因为他的内存是连续的。

➤ 交换数据的速度

这里是十万次交换的结果：

```
Swap for vector: 80000
Swap for deque: 100000
Swap for list: 170000
```

vector 的集群优势非常适合作内存交换，到底是搬家的行家。与之相比另外两位的表现就只能算是一般了。对 list 来说，他在内存中的离散分布，天生就不合适做此类的工作。

（编者注：对于 swap，其实各个容器都提供了专有的 swap 成员函数，有的只需要交换一下头指针，速度快得惊人，有兴趣的朋友不妨一试。这里说明了容器专有成员函数比一般的算法来得快，也就是 Scott Meyers 的新作 *Effective STL* 第 44 款 Prefer member functions to algorithms with the same names 的内容，诸位不妨参考一下。）

➤ erase 的速度

这里是一千次删除的结果：

```
RemoveMiddle for vector: 15320000
```

RemoveMiddle for deque: 0
RemoveMiddle for list: 60000

在任何情况下，不适合中间插入，同样意味着不适合从中间删除。所以，vector 的狼狈可见一斑。

但我们也不应该就此把问题绝对化，仅仅根据这里的测试结果，片面的认为一个容器的好与坏是不对的，就目前三大 STL 来说，就有很多不同的情况和因素影响着这些容器的效率。所以在其他的平台的不同编译器下，你完全有理由接受不同的测试结果，所谓具体问题具体分析也就是这个道理。

Result Table :

比赛项目\参赛选手	Vector	Deque	List
内存管理	Normal	Good	perfect
使用[]和 at() 操作访问数据	Good	Normal	N/A
Iterator 的访问速度	Good	good	Good
Push_back 操作（后插入）	Good	Good	Good
Push_front 操作（前插入）	N/A	good	Good
Insert（中间插入）	Normal	Perfect	Perfect
Erase（中间删除）	Normal	Perfect	Perfect
Pop_back（后部删除）	Perfect	Perfect	Normal
Swap（交换数据）	Perfect	good	Good
遍历	Perfect	Good	Normal

是到了宣布本场比赛冠军的时候了，可看看这张比分表，好像……哦，好像大家各有所长啊。

其实比赛的果只说明了一件事：在不同的情况下，使用不同有序容器模板，可以使我们的程序有最高的效率。不然也就辜负了 C++ 大师们给我们设计出这么多优秀的模板的期望。除此之外，我们在使用这些模板时还有许多窍门可以用，例如在使用 vector 的时候，如果我们能预先 reserve 足够的空间，使用效率将成倍提高！另外，千万别因为这场小小的较量就轻视他们中的任何一个，如果能让它们充分发挥，你的程序想来也将上一个档次。让我们共同努力吧。

开放 - 封闭原则 OCP

(The Open-Closed Principle)

Robert C. Martin
plpliuly&虫虫译

编者按：经 Robert Martin 先生的同意，从本期开始，我们将陆续翻译他于 1996 年在 C++ Report 上发表的关于面向对象设计原则的系列文章。虽然今天看来这些原则都非常朴素，很多话都似曾相识，但是，把它们作为原则系统地提出来，却有利于我们更好的理解这些原则并据此不断地反思和改善我们的设计活动。虽然这些文章发表于 5 年前，但我们相信这些文章仍然对读者朋友有帮助和启发作用。

译者的话：Java 和 C#出来以后，谁更 OO，谁最 OO 的无聊争论更是如火如荼，C++当然是众矢之的。经常见到用 Object Pascal、Java 的人说，C++的 RTTI 功能太弱，所以 C++落后。喜欢 C++的人，甚至坚信 C++是一种信仰的人，心里很不是滋味，但又说不出什么。译者写过些小程序，等到要升级的时候，却发现要改的代码不计其数，升级后 bug 不断，只好从头再来。为什么会这样？直到知道了开放 - 封闭原则 OCP (The Open-Closed Principle)，才慢慢有所领悟。希望大家在看完后也有所领悟，能来 email 与我们探讨。本文原文见 <http://www.objectmentor.com/publications/ocp.pdf>。

本文是我在 C++ Report 杂志上开设的“设计笔记”(Engineering NoteBook)专栏系列文章中的第一篇。这个专栏中文章将主要讨论 C++的应用和面向对象设计 (OOD)，也会涉及一些软件工程的问题。我将尽力使这些文章更具务实性并能够对从事一线开发的软件工程师们有直接的帮助。这些文章均使用 Booch 表示法作为设计图的图示方法。

关于面向对象设计，有许多启发式规则。比如，“所有的成员变量都应该是私有的”，“应该避免使用全局变量”，和“使用运行时刻类型标识 (RTTI) 是危险的”等等。追根溯源，这些启发式规则是如何得来的呢？道理何在呢？是否在任何情况均正确呢？本文就是试图探索这些启发式规则背后的设计原则——开放 - 封闭原则 (The Open-Closed Principle)。

Ivar Jacobson 曾经说过：“任何系统在其生命周期中都会发生变化。如果我们希望开发出的系统不会在第一版本后就被抛弃，那么我们就必须牢牢记住这一点。”¹那么怎样的设计才能面对需求的改变却可以保持相对稳定，从而使得系统可以在第一个版本以后不断推出新版本呢？Bertrand Meyer²在 1988 年提出的著名的开放 - 封闭 (open-closed) 原则给我们提供了一个很好的指引。他的大意是：

¹ *Object Oriented Software Engineering a Use Case Driven Approach*, Ivar Jacobson, Addison Wesley, 1992, p 21.

² *Object Oriented Software Construction*, Bertrand Meyer, Prentice Hall, 1988, p 23

软件组成实体（类，模块，函数，等等）应该是可扩展的，但是不可修改的。*[SOFTWARE ENTITIES (CLASSES, MODULES, FUNCTIONS, ETC.) SHOULD BE OPEN FOR EXTENSION, BUT CLOSED FOR MODIFICATION.]*

若程序中一处变动就产生连锁反应，导致一系列相关模块的变动，那么这个程序设计上不合理的地方已经暴露无遗：脆弱而缺乏灵活性，行为不可预先知，亦不可重用。而开放 - 封闭原则可以简洁明快地处理这一问题。按此原则，我们应该设计永不作改动的模块。如果需求变化，需要扩展模块的行为功能，我们只需要添加新的代码即可，不必触及已经正常运行的部分。

描述

遵循开放 - 封闭 (open-closed) 原则设计出的模块具有两个主要的表征。

1 . 可扩展，即“对扩展是开放的”(Open For Extension)

这意味着模块的行为功能可以被扩展，在应用需求改变或需要满足新的应用需求时，我们可以让模块以不同的方式工作。

2 . 不可更改，即“对更改是封闭的”(Closed for Modification)

这些模块的源代码是不可改动的。任何人都不许修改模块的源代码。

这两个表征好像是互相矛盾的：扩展模块的行为功能的通常方式就是修改此模块，不允许更改的模块通常都被认为是具有固定行为功能的。这二者之间的矛盾怎么调和呢？

关键是抽象

在 C++ 中遵循面向对象设计的原则，可以构造出固定却能够描述一组任意个可能行为的抽象体。这个抽象体就是抽象基类，而这一组任意个可能行为则表现为所有可能的派生类。模块可以操作一个抽象体。由于模块依赖于一个固定的抽象体，因此它可以是不允许修改 (closed for modification) 的；同时，通过从这个抽象体派生，也可扩展此模块的行为功能。

图 1 展示了一个简单的没有遵循开放 - 封闭 (open-closed) 原则的设计。Client 类和 Server 类都是具体类。Server 类的成员方法并不一定是 virtual 的。Client 类使用 Server 类。如果我们希望 Client 对象使用另一个服务对象，那么必须更改 Client 类，在使用 Server 类的地方改用新的服务类。



图 1 Closed Client

图 2 表示的是一个对应图一中问题但遵循开放 - 封闭 (open-closed) 原则的设计。在这个设计中，`AbstractServer` 类是一个拥有纯虚函数 (pure-virtual member function) 的抽象类。`Client` 类使用这个抽象类。而 `Client` 类的对象则使用 `Server` 类的派生类的对象。如果我们希望 `Client` 对象使用一个不同的服务类，那么只需从 `AbstractServer` 类派生一个新的类。`Client` 类不需作任何改动。

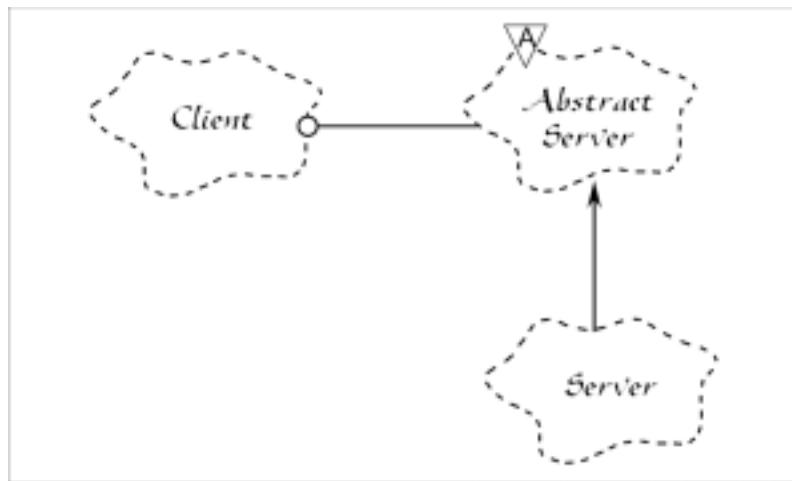


图 2 Open Client

Shape 抽象

考虑如下例子：一个需要在标准的图形用户界面 (GUI) 上画圆和矩形的应用程序。而且圆和矩形必须按特定的顺序画出。因此，我们需要以一定顺序创建的圆和矩形形成一个链表，程序遍历该链表，依次画出每一个圆和矩形。

如果采用 C 语言来解决这个问题，我们使用的是不遵循开放 - 封闭 (open-closed) 原则的面向过程的方法，有可能得到如 Listing 1 的解决办法。我们可以看到有一系列数据结构，它们的第一个成员都相同，但是其余的成员都不同。每个结构中的第一个成员是用来标识该结构体是代表圆抑或矩形的类型码。`DrawAllShapes` 函数遍历一个指针数组——数组中的元素是指向这些数据结构的指针——先检查类型码，然后调用对应的函数 (`DrawCircle` 或者 `DrawSquare`)。

Listing 1

```
/*Procedural Solution to the Square/Circle Problem*/
```

```
enum ShapeType {circle, square};

struct Shape
{
    ShapeType itsType;
};

struct Circle
{
    ShapeType itsType;
    double itsRadius;
    Point itsCenter;
};

struct Square
{
    ShapeType itsType;
    double itsSide;
    Point itsTopLeft;
};

// 下面两个函数的实现定义在别处
// 

void DrawSquare(struct Square*)
void DrawCircle(struct Circle*);

typedef struct Shape *ShapePointer;
void DrawAllShapes(ShapePointer list[], int n)
{
    int i;
    for (i=0; i<n; i++)
    {
        struct Shape* s = list[i];
        switch (s->itsType)
        {
            case square:
                DrawSquare((struct Square*)s);
                break;
            case circle:
                DrawCircle((struct Circle*)s);
                break;
        }
    }
}
```

函数 DrawAllShapes 不符合开放 封闭 (open-closed) 原则，因为对于新的 shape 类型不能做到自身封闭。如果我希望这个函数能够画出包括“三角形”在内的一系列图形，我就必须得更改

这个函数本身。实际上，每增加一个新的 shape 类型，我都必须更改这个函数。

当然这只是一个非常简单的例子。在实际的程序中，类似 DrawAllShapes 函数中的 switch 语句极有可能在应用程序中的各个函数中重复不断的出现；每一个 switch 语句负责完成的工作差别甚微。每当在这样的应用程序中增加一个新的 shape，就意味着需要找出所有包含上述 switch 语句（或者链式的 if/else 语句）的函数，并在每一处添加对新的 shape 类型的判断。更糟的是，并不是所有的 switch 语句和 if/else 语句都和 DrawAllShapes 中的那样有比较好的结构。更有可能的情形是，if 语句中的判断条件由逻辑操作符组合而成，或者是处理方式相同的 case 语句被成组处理。这样，要发现所有的需要增加对新的 shape 类型进行判断的地方，恐怕就是一件大耗心神的工作了。

Listing 2 展示了在开放 - 封闭原则下解决矩形 / 圆形问题的代码。其中创建了一个抽象类 Shape。这个抽象类有一个唯一的纯虚函数 Draw。Circle 类和 Square 类都是 Shape 类的派生类。

Listing 2

```
/*OOD solution to Square/Circle problem.*/
class Shape
{
public:
    virtual void Draw() const = 0;
};

class Square : public Shape
{
public:
    virtual void Draw() const;
};

class Circle : public Shape
{
public:
    virtual void Draw() const;
};

void DrawAllShapes(Set<Shape*>& list)
{
    for (Iterator<Shape*>i(list); i; i++)
        (*i)->Draw();
}
```

可以看到，如果我们想要扩展 DrawAllShapes 函数的行为使之能够画一个新的 shape 类型，我们所做的就只是增加一个新的 Shape 类的派生类。DrawAllShapes 函数并不需要改变。这样 DrawAllShapes 就遵循了 OCP 原则。勿需改动自身代码，它的行为就可以被扩展。

在实际开发的程序中 Shape 类可能有更多的方法。但加入一个新的 shape 类型到应用程序中

依然是简单的，因为所需做的工作就只是创建一个 Shape 类的新的派生类并实现它的所有函数。我们再也不需要为了找出应用程序中所有需要更改的地方而焦头烂额。

符合 OCP 原则的程序只通过增加代码来变化而不是通过更改现有代码来变化，因此这样的程序就不会引起象非开放 封闭（open-closed）的程序那样的连锁反应的变化。

选择性的封闭（Strategic Closure）

有一点需要清楚的是，没有任何一个大的程序能够做到 100% 的封闭。比如，假设我们在前面的例子中增加一个条件，要求所有的圆必须在矩形之前画出，那么 Listing 2 中的 DrawAllShapes 函数会怎样呢。显然，DrawAllShapes 函数无法对这种变化做到封闭。一般来讲，无论模块是多么的“封闭”，都会存在一些无法对之封闭的变化。

既然不可能完全封闭，因此就必须选择性地对待这个问题。也就是说，设计者必须对于他（她）设计的模块应该对何种变化封闭做出选择。当然这需要设计者具备一些从经验中积累的预测能力。有经验的设计者对用户和应用领域很熟悉，能够以此来判断各种变化的可能性。于是他（她）可以确保让设计对于最有可能发生的变化是遵循 OCP 原则的。

使用抽象获得显式封闭（Explicit Closure）

怎样才能使得 DrawAllShapes 函数对于画出顺序的变化是封闭的呢？请记住封闭是建立在抽象基础之上的。于是，为了让 DrawAllShapes 对于画出顺序变化是封闭的，我们需要一个“顺序抽象体”。一个具体的顺序定义了一种 shape 类型必须在另一种 shape 类型之前画出。

一个顺序规则意味着，给定两个对象，可以推导出应该先画哪一个。因此，我们可以定义一个 Shape 类的成员方法叫做 Precedes，该方法以另一个 Shape 作为参数并返回 bool 型结果。如果接受消息者（译者：即被调用方法者）应该先于以参数传入的对象画出，那么函数返回 true。

在 C++ 中这个函数可以通过重载 operator< 来代替。Listing 3 中是加入决定画出顺序的方法后的 Shape 类。

Listing 3

```
/*Shape with ordering methods.*/
class Shape
{
public:
    virtual void Draw() const = 0;
```

```

    virtual bool Precedes(const Shape&) const = 0;
    bool operator<(const Shape& s) {return Precedes(s);}
} ;

```

现在我们已经有了决定两个 Shape 对象的画出顺序的方法，我们可以对链表中的 shape 对象进行排序后依序画出。Listing 4 中是 C++ 的实现代码。代码中用了 Set, OrderedSet 和 Iterator 类，这些类的定义在我的书³中讲述的 Components 包的代码中（如果你想得到 Components 包的免费源码，请发邮件至 rmartin@oma.com）。

Listing 4

```

/*DrawAllShapes with Ordering*/
void DrawAllShapes(Set<Shape*>& list)
{
    // copy elements into OrderedSet and then sort.
    OrderedSet<Shape*> orderedList = list;
    orderedList.Sort();
    for (Iterator<Shape*> i(orderedList); i; i++)
        (*i)->Draw();
}

```

这给我们提供了一种对 Shape 对象排序的方法，也使得各个 Shape 对象可以按照一定顺序输出到图形用户界面。但是我们至此还没有一个很好的对于顺序的抽象体。按照我们的设计，Shape 对象应该重定义 Precedes 方法来指定顺序。但这究竟是如何工作的呢？我们应该在 Circle::Precedes 成员函数中写些什么代码来保证圆形一定会被先于矩形画出呢？请看 Listing 5。

Listing 5

```

Ordering a Circle
bool Circle::Precedes(const Shape& s) const
{
    if (dynamic_cast<Square*>(s))
        return true;
    else
        return false;
}

```

显然这个函数并不符合开放 封闭(open-closed)原则。因为没有办法使得这个函数对于 Shape 类的派生类封闭。每次创建一个新的 shape 派生类，这个函数就得改写。

³ Designing Object Oriented C++ Applications using the Booch Method, Robert C. Martin, Prentice Hall, 1995.

使用“数据驱动”的方法获取封闭性

Shape 派生类的封闭性可以通过使用表驱动的方法来获得，这种方法不需要强制改动每一个派生类。Listing 6 给出了一种可能实现。

通过这种方法我们基本上达到了使得 DrawShapes 函数对于顺序问题的封闭，也使得每一个 Shape 派生类对于新的 Shape 派生类或者是顺序规则的改变（比如，改变顺序为矩形必须先于圆画出）是封闭的。

Listing 6

```
/*Table driven type ordering mechanism*/
#include <typeinfo.h>
#include <string.h>
enum {false, true};
typedef int bool;
class Shape
{
public:
    virtual void Draw() const = 0;
    virtual bool Precedes(const Shape&) const;
    bool operator<(const Shape& s) const
        {return Precedes(s);}
private:
    static char* typeOrderTable[];
};

/*
译者注：由于typeinfo.name没有标准，因此最好直接用typeinfo作为表中的元素类型，而不是用类名字符串。
*/
char* Shape::typeOrderTable[] =
{
    "Circle",
    "Square",
    0
};
// This function searches a table for the class names.
// The table defines the order in which the
// shapes are to be drawn. Shapes that are not
// found always precede shapes that are found.
//
bool Shape::Precedes(const Shape& s) const
{
```

```

const char* thisType = typeid(*this).name();
const char* argType = typeid(s).name();
bool done = false;
int thisOrd = -1;
int argOrd = -1;
for (int i=0; !done; i++)
{
    const char* tableEntry = typeOrderTable[i];
    if (tableEntry != 0)
    {
        if (strcmp(tableEntry, thisType) == 0)
            thisOrd = i;
        if (strcmp(tableEntry, argType) == 0)
            argOrd = i;
        if ((argOrd > 0) && (thisOrd > 0))
            done = true;
    }
    else // table entry == 0
        done = true;
}
return thisOrd < argOrd;
}

```

对于不同的图形的画出顺序变化不封闭的唯一部分就是表本身。因为表可以从别的模块中分离出来，放在单独一个模块中，因此这些变化不会影响其他模块。

进一步扩展封闭性

故事还没有结束。我们已经设法使得 Shape 类层次和 DrawShapes 函数对于依赖于图形类型的画出顺序是封闭的。然而，如果画出顺序与图形类型无关，那么 Shape 派生类并不对这种顺序的变化封闭。我们似乎需要根据一个更加高层次的结构来决定画出各个 shape 的顺序。关于这个问题的深入彻底探讨已经超过了本文的范围；然而有兴趣的读者可能会考虑定义一个 OrderedObject 的抽象类，并从 Shape 类和 OrderedObject 类派生一个新的抽象类 OrderedShape。

启发式规则和习惯用法

在本文开头我们曾经提到过，开放 封闭 (open-closed) 原则是这些年关于 OOD 的很多启发式

规则和惯用法的根本动机。

所有成员变量都应该是私有的

这是关于 OOD 的最为知名的经验规则之一。成员变量只应该被同一个类中的成员方法可见。成员变量不应该被其他任何类知道，包括派生类。因此它们应该声明为 `private`，而不是 `public` 或者 `protected`。

根据开放 封闭 (open-closed) 原则，这个经验规则的道理是显而易见的。当成员变量改变以后，所有依赖于这个变量的方法都需改变。因此，没有任何依赖于这个变量的函数能够对之封闭。

在 OOD 中，我们期望类成员方法并不对成员变量的改变封闭。然而我们期望其他的类，包括派生类都对这些变量的变化是封闭的。我们对这种期望的特性有一个名字，称之为：封装。

如果你确信一个成员变量永远不会改变，那又是怎样的呢？必须将它声明为 `private` 吗？例如，Listing 7 中的 `Device` 类中又一个成员变量 `bool status`。这个成员变量包含了上次操作的状态。如果操作成功，`status` 的值为 `true`，否则为 `false`。

Listing 7

```
non-const public variable
class Device
{
public:
    bool status;
};
```

我们知道这个成员变量的类型和含义永远都不会改变。因此为什么不将其声明为 `public` 并直接访问它呢？如果这个变量真的不会改变，又如果所有的使用者都循规蹈矩，只是查询 `status` 的内容，那么事实上将这个变量声明为 `public` 没有什么害处。然而，请考虑一下哪怕是只有一个使用者对 `status` 进行写操作的情形。这样一来，这个使用者就可以影响所有其它的 `Device` 的使用者。这意味着任何 `Device` 的使用者都无法对这个“行为不端”的模块的变化封闭。这样做的风险实在太大。

另一方面，假入我们有一个如 Listing 8 所示的 `Time` 类。这个类中的 `public` 成员变量会带来不良的副作用吗？当然，这些变量是不大可能变化的。即使使用者改变了这些变量也无妨，因为这些变量本来就应该由使用者来改变。而且派生类也不大可能需要重定义对这些成员变量的设置操作。

Listing 8

```
class Time
{
public:
    int hours, minutes, seconds;
```

```

Time& operator-=(int seconds);
Time& operator+=(int seconds);
bool operator< (const Time&);
bool operator> (const Time&);
bool operator==(const Time&);
bool operator!=(const Time&);
};

```

对于 Listing 8 中的代码我唯一能够指出的缺陷就是对于时间的设置是非原子性的。也就是说，使用者可以改变 `minutes` 而不改变 `hours` 变量。这也许会导致 `Time` 对象的值不一致。我倾向于用一个带三个参数的成员方法来设置时间，这样便使得时间的设置具有原子性。但这个理由并不足以充分到必须将这些变量设为 `private`。

不难举出其他的情况说明将成员变量设为 `public` 将会产生一些问题。然而，最终来讲，并没有压倒性的理由要求成员变量必须声明为 `private`。我仍然认为将成员变量声明为 `public` 只是一种不好的风格，但并不一定是一个有问题的设计。我之所以认为这是一个不好的风格，是因为创建一个对应的 `inline` 成员函数的代价是非常小的；而这种很小的代价换来的是能够抵御可能破坏封闭性的风险。

因此，在少有的完全遵循开放 封闭（open-closed）原则的例子中，完全杜绝 `public` 和 `protected` 的成员变量与其说是因为一种设计的本质上的要求不如说是一种风格上的选择。

永远不要用全局变量

反对使用全局变量的原因和反对使用 `public` 的成员变量的原因很相似。与一个全局变量关联的模块不可能对于可能写这个全局变量的其他模块做到封闭。任何一个模块以非常方式访问这个全局变量都可能破坏其他使用这个全局变量的模块。让多个模块因为一个模块的怪异行为而遭受牵连，显然是非常危险的。

另一方面，如果与全局变量关联的模块不是很多，或者全局变量不允许破坏其一致性的访问，那么这样的全局变量并不会带来很大弊处。因此，设计者必须权衡使用一个全局变量所牺牲的封闭性和使用该全局变量所带来的方便性，并决定使用这样一个全局变量是否值得。

此处又有一个关于风格的问题。代替全局变量的方法一般都不需很高的代价。在这种前提下即使是牺牲很小的封闭性来使用全局变量也被认为是不好的风格。然而，有些情况下全局变量的方便性是很重要的。全局变量 `cout` 和 `cin` 就是例子。在这种情况下，如果没有破坏开放 封闭（open-closed）原则，那么牺牲风格来获得这种方便性是值得的。

RTTI 是危险的

另一个著名的戒条就是反对使用 `dynamic_cast`。`dynamic_cast` 和运行时刻类型标识（RTTI）的任何形式都被认为本质上是危险而应该避免使用的。常举的例子就是类似 Listing 9 中的

情形，这段代码显然违背了开放 封闭（open-closed）原则。然而 Listing 10 中使用 `dynamic_cast` 的相似例子却并没有破坏开放 封闭（open-closed）原则。

Listing 9

```
/*RTTI violating the open-closed principle.*/
class Shape {};
class Square : public Shape
{
private:
    Point itsTopLeft;
    double itsSide;
    friend DrawSquare(Square*);
};

class Circle : public Shape
{
private:
    Point itsCenter;
    double itsRadius;
    friend DrawCircle(Circle*);
};

void DrawAllShapes(Set<Shape*>& ss)
{
    for (Iterator<Shape*>i(ss); i; i++)
    {
        Circle* c = dynamic_cast<Circle*>(*i);
        Square* s = dynamic_cast<Square*>(*i);
        if (c)
            DrawCircle(c);
        else if (s)
            DrawSquare(s);
    }
}
```

Listing 10

```
/*RTTI that does not violate the open-closed Principle.*/
class Shape
{
public:
    virtual void Draw() const = 0;
};

class Square : public Shape
{
// as expected.
```

```
};

void DrawSquaresOnly(Set<Shape*>& ss)
{
    for (Iterator<Shape*>i(ss); i; i++)
    {
        Square* s = dynamic_cast<Square*>(*i);
        if (s)
            s->Draw();
    }
}
```

二者中的区别是：首先，每当一个增加一个新的 Shape 类的派生类，Listing 9 中的代码都必须更改。（不用说这显然是一种笨拙的方法）然而，Listing 10 中的代码却无需改动。因此，Listing 10 中的代码没有破坏开放 封闭（open-closed）原则。

根据一般的经验，如果使用 RTTI 不会破坏开放 封闭（open-closed）原则，那么就是安全的。

结论

关于开放 封闭（open-closed）原则可以说的东西远不止这些。从多种意义上来说，这个原则是面向对象设计的核心。遵循这个原则带来的好处就是面向对象技术所声称的优点：可重用性和可维护性。然而，并不是说仅仅使用一种面向对象编程语言就是遵循这个原则。而是依赖于设计者对程序中他认为可能发生变化的部分做出合理的设计上的抽象。

这篇文章可以说是我的新书 *The Principles and Patterns of OOD*（注：《面向对象设计的原则和模式》）中一章的压缩版本，这本书将由 Prentice Hall 出版。在接下来的文章中我们将一起探讨面向对象设计的另外一些原则。我们也将讨论各种设计模式以及结合 C++ 的实现时的优缺点。我们也将讨论 Booch 的关于类的类别划分在 C++ 中扮演的角色以及作为 C++ 的 namespace 的可应用性。我们也将定义面向对象设计中“内聚”（cohesion）和“耦合”（coupling）的含义，同时会讨论衡量面向对象设计质量的准则。还有其他一些很有意思的话题。

深入 VCL 理解 BCB 的消息机制（二）

CKER

方法 2：重载 TControl 的 WndProc 方法

还是先谈谈 VCL 的继承策略。VCL 中的继承链的顶部是 `TObject` 基类。一切的 VCL 组件和对象都继承自 `TObject`。

打开 BCB 帮助查看 `TControl` 的继承关系：

`TObject->TPersistent->TComponent->TControl`

原来 `TControl` 是从 `TPersistent` 类的子类 `TComponent` 类继承而来的。`TPersistent` 抽象基类具有使用流 stream 来存取类的属性的能力。

`TComponent` 类则是所有 VCL 组件的父类。

这就是所有的 VCL 组件包括您的自定义组件可以使用 dfm 文件存取属性的原因（当然要是 `TPersistent` 的子类，我想您很少需要直接从 `TObject` 类来派生您的自定义组件吧）。

`TControl` 类的重要性并不亚于它的父类们。在 BCB 的继承关系中，`TControl` 类的是所有 VCL 可视化组件的父类。实际上就是控件的意思吧。所谓可视化是指您可以在运行期间看到和操纵的控件。这类控件所具有的一些基本属性和方法都在 `TControl` 类中进行定义。

`TControl` 的实现位于 `\Borland\CBUILDER5\Source\Vcl\control.pas` 中可以找到。（可能会有朋友问你怎么知道在那里？使用 BCB 提供的 Search -> Find in files 很容易找到。或者使用第三方插件的 grep 功能。）

好了，进入 VCL 的源码吧。说到这里免不了要抱怨一下 Borland。哎，为什么要用 pascal 实现这一切……:-)

`TControl` 继承但并没有重写 `TObject` 的 `Dispatch` 方法。反而提供了一个新的方法 `WndProc`。一起来看看 Borland 的工程师们是怎么写的吧。

```
procedure TControl.WndProc(var Message: TMessage);
var
  Form: TCustomForm;
begin
  //由拥有control的窗体来处理设计期间的消息
  if (csDesigning in ComponentState) then
    begin
      Form := GetParentForm(Self);
```

```
if (Form <> nil) and (Form.Designer <> nil) and
  Form.Designer.IsDesignMsg(Self, Message) then Exit;
end
//如果需要，键盘消息交由拥有control的窗体来处理
else if (Message.Msg >= WM_KEYFIRST) and (Message.Msg <= WM_KEYLAST)
then
begin
  Form := GetParentForm(Self);
  if (Form <> nil) and Form.WantChildKey(Self, Message) then Exit;
end
//处理鼠标消息
else if (Message.Msg >= WM_MOUSEFIRST) and (Message.Msg <= WM_MOUSELAST)
then
begin
  if not (csDoubleClicks in ControlStyle) then
    case Message.Msg of
      WM_LBUTTONDOWNDBLCLK, WM_RBUTTONDOWNDBLCLK, WM_MBUTTONDOWNDBLCLK:
        Dec(Message.Msg, WM_LBUTTONDOWNDBLCLK - WM_LBUTTONDOWN);
    end;
  case Message.Msg of
    WM_MOUSEMOVE: Application.HintMouseMessage(Self, Message);
    WM_LBUTTONDOWN, WM_LBUTTONDOWNDBLCLK:
      begin
        if FDragMode = dmAutomatic then
          begin
            BeginAutoDrag;
            Exit;
          end;
        Include(FControlState, csLButtonDown);
      end;
    WM_LBUTTONUP:
      Exclude(FControlState, csLButtonDown);
  end;
end
// 下面一行有点特别。如果您仔细的话会看到这个消息是CM_VISIBLECHANGED。
// 而不是我们熟悉的WM_开头的标准Windows消息。
// 尽管Borland没有在它的帮助中提到有这一类的CM消息存在。但很显然这是BCB的
// 自定义消息。呵呵，如果您对此有兴趣可以在VCL源码中查找相关的内容。一定会有不小的收获。
else if Message.Msg = CM_VISIBLECHANGED then
  with Message do
    SendDockNotification(Msg, WParam, LParam);
// 最后调用dispatch方法。
```

```
    Dispatch(Message);
end;
```

看完这段代码，你会发现 TControl 类实际上只处理了鼠标消息，没有处理的消息最后都转入 Dispatch() 来处理。

但这里需要强调指出的是 TControl 自己并没有获得焦点 Focus 的能力。TControl 的子类 TWinControl 才具有这样的能力。我凭什么这样讲？呵呵，还是打开 BCB 的帮助。很多朋友抱怨 BCB 的帮助实在不如 VC 的 MSDN。毋庸讳言，的确差远了。而且这个帮助还经常有问题。但有总比没有好啊。

言归正传，在帮助的 The TWinControl Branch 分支下，您可以看到关于 TWinControl 类的简介。指出 TWinControl 类是所有窗体类控件的基类。所谓窗体类控件指的是这样一类控件：

1. 可以在程序运行时取得焦点的控件。
2. 其他的控件可以显示数据，但只有窗体类控件才能和用户发生键盘交互。
3. 窗体类控件能够包含其他控件(容器)。
4. 包含其他控件的控件又称做父控件。只有窗体类控件才能够作为其他控件的父控件。
5. 窗体类控件拥有句柄。

除了能够接受焦点之外，TWinControl 的一切都跟 TControl 没什么分别。这一点意味着 TWinControl 可以对许多的标准事件作出响应，Windows 也必须为它分配一个句柄。并且与这个主题相关的最重要的是，这里提到是由 BCB 负责来对控件进行重画以及消息处理。这就是说，TWinControl 封装了这一切。

似乎扯的太远了。但我要提出来的问题是 **TControl 类的 WndProc 方法中处理了鼠标消息。但这个消息只有它的子类 TWinControl 才能够得到啊！？**

这怎么可以呢……Borland 是如何实现这一切的呢？这个问题实在很奥妙。为了看个究竟，再次深入 VCL 吧。

还是在 control.pas 中，TWinControl 继承了 TControl 的 WndProc 方法。源码如下：

```
procedure TWinControl.WndProc(var Message: TMessage);
var
  Form: TCustomForm;
  KeyState: TKeyboardState;
  WheelMsg: TCMMouseWheel;
begin
  case Message.Msg of
```

```
WM_SETFOCUS:
begin
    Form := GetParentForm(Self);
    if (Form <> nil) and not Form.SetFocusedControl(Self) then Exit;
end;

WM_KILLFOCUS:
if csFocusing in ControlState then Exit;

WM_NCHITTEST:
begin
    inherited WndProc(Message);
    if (Message.Result = HTTRANSPARENT) and
    (ControlAtPos(ScreenToClient(
        SmallPointToPoint(TWMNCHitTest(Message).Pos)), False) <> nil)
then
    Message.Result := HTCLIENT;
    Exit;
end;

WM_MOUSEFIRST..WM_MOUSELAST:
// 下面这一句话指出，鼠标消息实际上转入 IsControlMouseMsg 方法来处理了。
if IsControlMouseMsg(TWMMouse(Message)) then
begin
    if Message.Result = 0 then
        DefWindowProc(Handle, Message.Msg, Message.wParam,
Message.lParam);
    Exit;
end;

WM_KEYFIRST..WM_KEYLAST:
if Dragging then Exit;

WM_CANCELMODE:
if (GetCapture = Handle) and (CaptureControl <> nil) and
(CaptureControl.Parent = Self) then
    CaptureControl.Perform(WM_CANCELMODE, 0, 0);

else
with Mouse do
    if WheelPresent and (RegWheelMessage <> 0) and
    (Message.Msg = RegWheelMessage) then
begin
    GetKeyboardState(KeyState);
    with WheelMsg do
begin
    Msg := Message.Msg;
    ShiftState := KeyboardStateToShiftState(KeyState);
    WheelDelta := Message.WParam;
```

```

    Pos := TSmallPoint(Message.LParam);
  end;
  MouseWheelHandler(TMessage(WheelMsg));
  Exit;
end;
end;
inherited WndProc(Message);
end;

```

鼠标消息是由 IsControlMouseMsg 方法来处理的。只有再跟到 IsControlMouseMsg 去看啦。源码如下：

```

function TWinControl.IsControlMouseMsg(var Message: TWMMouse): Boolean;
var
  //TControl 出现啦
  Control: TControl;
  P: TPoint;
begin
  if GetCapture = Handle then
  begin
    Control := nil;
    if (CaptureControl <> nil) and (CaptureControl.Parent = Self) then
      Control := CaptureControl;
    end else
      Control := ControlAtPos(SmallPointToPoint(Message.Pos), False);
  Result := False;
  if Control <> nil then
  begin
    P.X := Message.XPos - Control.Left;
    P.Y := Message.YPos - Control.Top;
    file://TControl 的 Perform 方法将消息交由WndProc 处理。
    Message.Result := Control.Perform(Message.Msg, Message.Keys,
    Longint(PointToSmallPoint(P)));
    Result := True;
  end;
end;

```

原来如此，TWinControl 最后还是将鼠标消息交给 TControl 的 WndProc 来处理了。这里出现的 Perform 方法在 BCB 的帮助里可以查到，是 TControl 类中开始出现的方法。它的作用就

是将指定的消息传递给 TControl 的 WndProc 过程。

结论就是 TControl 类的 WndProc 方法的消息是由 TWinControl 类在其重载的 WndProc 方法中调用 IsControlMouseMsg 方法后使用 Perform 方法传递得到的。

由于这个原因，BCB 和 Delphi 中的 TControl 类及其所有的派生类都有一个先天的而且是必须的限制。那就是所有的 TControl 类及其派生类的 Owner 必须是 TWinControl 类或者 TWinControl 的派生类。Owner 属性最早可以在 TComponent 中找到，一个组件或者控件是由它的 Owner 拥有并负责释放其内存的。这就是说，当 Owner 从内存中释放的时候，它所拥有的所有控件占用的内存也都被释放了。Owner 最好的例子就是 Form。Owner 同时也负责消息的分派，当 Owner 接收到消息的时候，它负责将应该传递给其所拥有的控件的消息传递给它们。这样这些控件就能够取得处理消息的能力。TImage 就是个例子：你可以发现 Borland 并没有让 TImage 重载 TControl 的 WndProc 方法，所以 TImage 也只有处理鼠标消息的能力，而这种能力正是来自 TControl 的。

唧唧噥噥的说了一大堆。终于可以说处理消息的第二种方法就是重载 TControl 的 WndProc 方法了。例程如下：

```
void __fastcall TForm1::WndProc(TMessage &Message)
{
    switch (Message.Msg)
    {
        case WM_CLOSE:
            OnCLOSE(Message); // 处理WM_CLOSE 消息的方法
            break;
    }
    TForm::WndProc(Message);
}
```

乍看起来，这和上次讲的重载 Dispatch 方法好象差不多。但实际上还是有差别的。差别就在先后次序上，从前面 TControl 的 WndProc 可以看到，消息是先交给 WndProc 来处理，最后才调用 Dispatch 方法的啦。

这样，重载 WndProc 方法可以比重载 Dispatch 方法更早一点点得到消息并处理消息。

好了，这次就说到这里。在您的应用程序里还有没有比这更早得到消息的办法呢？有，下次再说。（未完待续）

H_{otline}

回音壁

大家好，我是 C++ View 的主编。C++ View 第一期推出后，不少朋友发来 email，给予赞许和鼓励，同时也提出了许多宝贵的意见和建议，小编在此一并谢过，并且说明几件事。

定位

C++ View 定位于相对较高层次的 C++ 理论与实践，要求读者和作者具有一定的 C++ 基础。本刊初步确定的方向是：泛型编程和标准模板库（Generic Programming & STL）面向对象与应用架构（Object-Oriented & Application Framework）以及设计样式与原则（Design Patterns & Principles）。

运作模式

现在 C++ View 的运作模式是基于免费原则，读者可以免费阅读，作者及编辑亦没有任何报酬。但是希望读者积极反馈意见和建议。

名称及标志

C++ View 只有英文名，对于一个中文杂志来说，希望有一个响亮的中文名。同时 C++ View 亦需要一个刊徽。希望各位朋友能为我们出谋划策。

编辑人员

现在的编辑人员少得可怜，希望在 C++ View 的几个方向上有一定造诣并且时间较为充裕的朋友能够加盟，让我们更好地为大家服务。另外 C++ View 也需要美工人员帮忙设计。由于现在 C++ View 并没有商业化，因此加盟的朋友在商业化之前没有任何报酬。

栏目设置

除前面提到的三个方向外，C++ View 还有其他的栏目。所有栏目均欢迎大家投稿，栏目也会根据大家的要求增加或减少。“视点”欢迎大家灌水，在符合法律法规的前提下发表任何看法；“星闻”则是介绍人物，可以是大人物，也可以是我们自己；“抢鲜快报”是报道一些新鲜的事情或者评测新软件；“管中窥豹”取以一斑窥全貌之意，讨论一些虽然小却能蕴涵深刻道理的问题。

版权

本刊鼓励原创，同时也会翻译国外优秀的经典文章。原创文章如果由作者和编辑修改共同完成，则修改后的文章归本刊和作者共有，修改前的文章仍归作者所有。翻译文章如果是本刊取得授权，则作者不得另作用途。所有文章转载前必须经过其全部所有者同意。本刊在免费阶段可以自由传播，但必须以整体方式，不得转载单独的文章。

为了方便大家，各位可以发 email 订阅 C++ View，我们将为您奉上 C++ View 最新的消息。在 email 中，请尽可能详细地告诉我们您的情况，以便我们根据读者的情况及时作出相应的调整。

C++ VIEW

第3期

```
#include <iostream.h>

class CObject
{
public:
    virtual void Serialize() { cout << "CObject::Serialize()\n\n"; }

class CDocument : public CObject
{
public:
    int m_data1;
    void func() { cout << "CDocument::func()\n" << endl;
        Serialize();
    }
};
```

焦点

给虫虫的第一封信

专访 Bjarne Stroustrup

```
class CMyDoc : public CDocument
```

精彩推荐

◆ Traits on Steroids
◆ Liskov 替换原则
◆ 委托模式



第3期目录 2001年9月

焦点 Focus

给虫虫的第一封信	03
专访 Bjarne Stroustrup	05
回音壁	83

特稿 Feature

C++空成员类优化	16
深入 BCB 理解 VCL 的消息机制（三）	22

专栏 Column

Generic<Programming>	
Traits on Steroids	28
C++的不足之处讨论系列	
保证类型安全的连接属性	37
设计笔记	
Liskov 替代原则 LSP	39
Pattern Hatching	
从 GoF 谈起	49
P = A + D	
vector 容器的性能分析	55
模式罗汉拳	
委托模型	62
天方夜谭 VCL	
开门	68

导读

日本首相例行的参拜换来例行的抗议，如此而已。反倒是一个日本狂徒的反问让我傻了眼：你们中国人的“靖国神社”呢？你们可曾为战争中的英雄表达过任何的敬意？……是啊，8月15日那天我们是否想起为抗日战争中为国捐躯的民族英雄呢？我们又是否用某种方式表达了敬意呢？

小编把栏目作了相应的调整，大家觉得如何？这次小编非常荣幸地采访了 C++之父，在此极力推荐《专访 Bjarne Stroustrup》，仔细品位大师的深意吧。相信大家都知道《设计模式》这本书，但是具体的应用呢？从本期开始，我们推出 John Vlissides 博士的“Pattern Hatching”系列，第一篇是《从 GoF 谈起》。Vlissides 博士可是《设计模式》的作者，Gang of Four 之一哦，绝对权威。

庆祝三个原创专栏开张喽！starfish 的“P=A+D”，意思是“Program=Algorithm+Data structure”，首先将会进行 STL《容器 vector 的性能分析》。透明的“模式罗汉拳”专栏，这次为大家讲述《委托模式》。《倚天屠龙记》里说，罗汉拳是少林拳法中的入门功夫，“看来平平无奇，但练到精深之处，实是威力无穷”，知道厉害了吧？还有虫虫的“天方夜谭 VCL”。这小子小说看多了，文章起些名字也怪怪的：《开门》，好像有一点点卫斯理的风格。

另外《Traits on Steroids》和《Liskov 替代原则 LSP》都是相当经典的文章，不可错过！《C++空成员类优化》讲述了 C++ 中一个非常重要的特性，《深入 BCB 理解 VCL 的消息机制》刊出最后一部分，同时还有透明《给虫虫的第一封信》。所有关心 C++ View 的朋友，请锁定《回音壁》。

主 编：王 曦 主页：<http://cppview.yeah.net>
 封面设计：俞 蕤 <http://cppview.y365.com>
 排 版：虫 虫 电邮：cppview@china.com
 本期编辑：王 曦 cppview@sohu.com

给虫虫的第一封信

透明

这是我给虫虫的第一个 mail，原文写于 8 月 12 日凌晨两点左右，神情恍惚，思维紊乱，文笔也毫无条理。现在整理它，我不想改动原来的文字，并将后来的想法加在括号中用斜体字标注，于是更显凌乱。希望各位能从我散乱的文字中读出我的心声。

这是透明给虫虫的第一封信，信中从虫虫在第 1 期上写的创刊词谈起，对虫虫，也对 C++ View 提出了一些意见。信中斜体是透明的注释，楷体则是小编的注释，其余为正文。

想了一百条理由今天不熬夜：明天要出门、连续熬夜三天……然后用一条理由将它们全部推翻：我忍不住 C++ View 的诱惑。

JAVA 的确比 C++ 更加 OO，这是其语言本身决定的，不必去争。我们要的是一个能优雅有效的解决问题的语言，至于它是 OO 的还是 structure 的，这不是我们关心的（至少不是我们最关心的）。

关于这个问题，虫虫提出了三个问题：

- OO 的含义是什么？
- 更 OO 的评定标准是什么？
- 越 OO 就越先进的依据是什么？

拙见如下：

- OO 的含义是“面向对象”。The C++ Programming Language 中这样写：“The (Object-Oriented) programming paradigm is: Decide which classes you want; provide a full set of operations for each class; make commonality explicit by using inheritance.”
- “更 OO”的意思是，JAVA 中所有的东西都是类、对象，所有的函数都是虚拟，其面向对象的程度更深更彻底；相比之下，C++ 保留了与 C 兼容的子集，支持面向过程、数据抽象、泛型等 paradigm，OO 程度并不彻底。
另外虫虫说 C#“更 OO”，可能是因为对其底层机制并不了解吧？C# 是建立在 COM 基础之上的，而 COM……那里可有一大堆的全局函数呢。
- 谬论。JAVA 最大的优势是可移植性，在 VM 的基础上可以开发具有更高通用性的软件。但是其高度面向对象的特性和 VM 机制严重影响了效率。即使不考虑 VM 的影响，JAVA 所有的函数都是虚拟函数，查表就会耗去更多的时间。只能说，JAVA 与 C++ 各有特点，各有应用范围。

虫虫看了这段，大呼冤枉，差点“八月飞雪”。我问他怎么了，他只是一个劲儿地嘀咕“我没有说 C# 更 OO 啊”。

更重要的是：我们喜欢 C++。甚至我们喜欢在特别的情况下可以破坏封装的办法，尽管我们尽量不这样做。

（以上是我看完开篇词及 Stroustrup 访谈之后的想法）

不要学着侯捷说话的语气。尽管他很牛，但我仍然不喜欢他言语中的狂傲。请保持谦逊的态度，你和你的杂志都应当。

在下有幸看到侯先生给另一位老师的赠书，扉页上侯先生提字曰：“海若无涯天做岸，山到穷顶我为峰”。愚见认为，作为一个科技工作者，无论如何不应当有这样的张狂；而我等没有绝顶功力的

学习者，更是不应当先学到这种张狂。

这也算“狂”吗？我想文学就是这样，想不想头发就是“三千丈”，或者“历史的瞬间”，几个世纪就过去了。

做“立地”的人，很好，也很难。你知道吗？风雨袭来的时候，被击倒的一定是“立地”的人。为什么？杨花柳絮般漂浮的“高手”们早已随波逐流了。因此，我对你表示三分钦佩。不管你的勇气是否只是因为年少的轻狂。一个真正的人，顶天立地的人，是不会倒的。

文章略显单薄。希望你不会倒下。如果你在半年之内倒下，白白浪费了自己的时间；如果你能坚持，也许你真会成为开天辟地者。

（看完第一期之后的想法）

虫虫给我的回信验证了我的担忧。大家都有现实的忙碌，有多少人能坚定自己的梦想？XPROGRAMMER 的主编 think 有稳定的工作，维持杂志也相当艰难，毕竟现在用杂志赚钱还比较困难。但是如果 C++ VIEW 在半年内倒下，只有 CSDN 这些人知道，岂非闭门造车？我希望所有喜欢 C++ VIEW 的同仁能一起支持它，不要让它倒下。

小编做事可没有这么执着。在第 1 期的导读中，小编说过，不管 C++ View 能否带走一片云彩，能带给大家哪怕是一点点帮助，我就心满意足了。

话很烫，因为心很真，不愿和志同道合者做表面文章。在下只是 C++ 的初学者，曾经走过很多弯路，现在也不敢说初窥门径。完全是出于一种热诚才对你有那么多的评论。

对于 C++ 的技术细节，我无法与众多高手论剑；对于设计模式，在下却敢与众人煮酒而谈。不知 C++ View 对于这方面内容可有兴趣涉及？若能借贵刊一管让广大程序员得以窥豹，岂非美事一件？

我很贪心。我的意思就是想要一个专栏，因为我有很多东西想与大家分享。后来虫虫就真的答应我一个专栏，并给我很多赞誉。对于这个专栏，我感谢虫虫及所有人的信任；对于这些赞誉，愧不敢当。我只是想有一个空间与所有同好讨论喜欢的东西，绝不敢有半分非分之想。

你以为小编我真想给啊？虫虫那家伙无时无刻无处都缠着我，我……

希望能结交你这位小朋友，热切盼复。

现在我想已经算和这位小朋友结交了。对这样一位热血青年，不必我再来褒奖了。如果中国的大学生都能象这样，何愁四化不成祖国不兴？谨劝戒某些“爱国志士”：有时间发表过激言论、有时间攻击人家网站，不如来做些实事。也寄言虫虫：我不便对中国的大学教育做过多评论，但希望你不要被大学磨平了棱角。

我问虫虫有何感想，虫虫反问我，南京的雨花石很漂亮吧？为什么呢？那是因为石子儿从长江上游到下游，磨平了棱角，却没有失去个性，反而更显出了色彩。

我 闭上眼睛

天空 变得透明

专访 Bjarne Stroustrup

翻译：ALNG

编者按：在 C++ View 第 1 期中我们介绍了这位 C++ 之父，相信大家都很熟悉了。这次，小编对这位大人物进行了专访，非常感谢 Stroustrup 博士的精彩回答和耐心的解释。感谢 cber 添补的问题，和对小编中国式英语的纠正。同时还要感谢 ALNG 精彩的中文翻译，这能方便大家更好地理解 Stroustrup 博士的深邃思想。关于翻译的一些细节问题，小编与 ALNG 争吵了整整一天，大体上达成了一致。有些需要说明的地方，都在文中以楷体注明。不过小编还得声明，Stroustrup 博士回答的内容均以英文为准。

C++ 的 ANSI/ISO 标准化标志着 C++ 的成熟。能告诉我们在标准化过程中，您感到最难忘、最快乐以及最遗憾的事分别是什么吗？

The ANSI/ISO standardization of C++ indicates that the C++ language has matured. Would you please tell us the most unforgettable, the happiest and the most regrettable things you felt in the course of standardization?

标准化是一个极具价值的重要活动，它在很大程度上被低估，困难重重。通过标准化，C++ 变得更好了，还获得了有着惊人表达力的标准库。编译器提供商总是想锁定用户，正式的标准化则是用户拥有的为数不多的防御手段之一。

Standardization is an extremely valuable, most important, largely underestimated, and most frustrating activity. C++ became a better language through standardization and acquired a standard library of surprising expressive power. Formal standardization is one of the few defenses that a user has against the interests of compiler suppliers, who always try to lock in their users.

很难挑出特定的事。委员会的大多数工作形式上都是发现、提炼、建立信任的过程，要花时间。不过最重要的事一定是 1990 年对以 *The C++ Programming Language* 第二版（其中引入了模板和异常机制）作为参考手册来标准化 C++ 的初次投票或 1998 年批准 ISO 标准的最终表决，两者之一。

It's hard to pick out specific events. Most of the work in the committee has the form of a process of discovery, refinement, and building of trust. Such things take time. However, the single most important event must be either the initial 1990 vote to standardize C++ based on the reference manual of the 2nd edition of "The C++ Programming Language" (that is, with templates and exception handling) or the final 1998 vote ratifying the ISO standard. In between those events, the vote to accept the STL as part of the standard library standard stands out as a most happy event.

没有任何负面或遗憾的事可以与这些积极的投票相提并论。所谓“遗憾”的事，要么是细微的技术细节，要么是（暂时）分化了委员会而使进一步的进展更加困难的讨论。我本来是反对 C 风格的 cast，也不想引入仅允许整型的静态常量成员在类中初始化的机制。不过这些只是无关大局的细节。【注：cast 翻译成什么好呢？其本意是铸造、打型，诸位以为，“映射”、“转换”、“型铸”哪一个翻译更贴切呢？或者有其他更好的翻译？】

There is no negative/regrettable event of a magnitude to match these positive votes. All "regrettable things"

are either very minor technical details or discussions that (temporarily) polarized the committee so as to make further progress harder. I would have liked to deprecate C-style casts and not to introduce in-class initialization of static const members of integral types (only), but these are minor details.

我期待着另外一次重要表决。明年某个时间委员会将决定 ISO C++的未来方向，这可是头等大事啊！
I am looking ahead to yet another key vote. Sometime over the next year, the committee will decide on the future directions for ISO C++. That will be an event of the first magnitude.

C++的标准化为何会困难重重？另外可以再谈谈委员会里的工作进程吗？

Why is the standardization of C++ frustrating? And would you please tell us more about the process of the work in the committee?

标准化是个缓慢的进程，常常聚焦在琐细的技术细节上。你要让几十个来自不同国家、受过不同技术教育的人达成一致，并需要代表着各种组织（或仅仅本人）的委员富有成果地合作。C++委员会不是一个满足于 60% 对 40% 的差距“获胜”的组织。我们认为这样的表决结果就是失败。我们的目标是一致同意，意思是“基本人人赞同”。我们为达成一致不懈努力。很艰难，差不多每个人起码是有时候都希望有一个快捷一点的方式。当然，我们的成果是一个公认能很好地满足一个大得难以置信的群体的需求的语言，而不是对某一用途或某个人来说的完美语言。最终我们达成了标准的一致通过（ANSI 中 43-0，ISO 中 22-0）。有人告诉我，对编程语言标准而言，这一赞成度前所未有。

Standardization is a slow process, often focussed on minute technical details, and you need to get dozens of people from many countries and from very diverse technical cultures to agree. Also people representing very different organizations (or just themselves) need to collaborate productively. The C++ committee is not an organization that is happy with a vote being "won" by a 60% to 40% margin. Such a vote would be considered a failure. We aim for consensus, meaning "almost everybody agrees" and work until we reach that. That's hard, and everybody - at least sometimes - wish for a faster way. However, the result is a language that is acknowledged to be good enough for an incredibly large community, rather than being just perfect for any one use or any one individual. In the end, we managed to get unanimous votes (43-0 in ANSI and 22-0 in ISO) for the standard. I have been told that this degree of agreement has never before been achieved for a programming language standard.

首先委员会要弄清真正的问题和可行的技术解决方案。我称之为“发现”。接着我们把解决方案提炼成标准文本中精确描述的东西。参加标准过程的个人总是必须学会相互协作以及相信他人的善意和专业才能。我称之为“建立信任”——这非常可能是标准进程中最重要的一部分，没有互信我们将一事无成。

First the committee has to figure out what the real problems are and what kind of technical solution is feasible. This, I referred to as "discovery". Next we have to refine that solution into something precisely described in standards text. And always the individuals taking part in the standards process must learn to work with each other and to trust the good intent and the professional abilities of others. That, I referred to as "building of trust" - it is quite possibly the most important single part of the standard process; without mutual trust nothing can be achieved.

Alexander Stepanov 说有一次他曾和你争论。因为他认为 C++的模版函数应该象 Ada 通用类一样显式

实例化，而你坚持认为函数应使用重载机制隐式实例化。由于你的坚持，这一技术后来在 STL 中发挥了重要作用。能和我们讲讲这个故事吗？

Alexander Stepanov said that once he had argued with you because he thought C++ template functions should be explicitly instantiated like Ada generics, while you insisted that functions be instantiated implicitly using an overloading mechanism. Thanks to your insistence, this particular technique later plays an important part in STL. Could you tell us more about this story?

我没有多少可补充的。在模版成为 C++ 的一部分之前，Alex 和我花时间讨论过一些语言特性。在我看来，那时 Ada 上的经验给了他过分的影响，而他有着我很大程度上缺乏的宝贵的泛型编程的实践经验。他加强了我对不牺牲效率和内联的偏爱。我们都讨厌宏而喜欢类型安全。他本来想要更强的模板参数的静态类型检验。我也这么想，不过找不到可以不限制表达能力或牺牲效率的实现方法。尤其是，我过去是，现在还是，对把模板参数限制在继承层次的努力持怀疑态度。

I can't add much. Alex and I spent some time discussing language features before templates became part of C++. He was - in my opinion – at the time overly influenced by his experience with Ada, but he also had valuable practical experience with generic programming that I largely lacked. He reinforced my bias in favor of uncompromising efficiency and inlining. We both shared a dislike of macros and a liking for type safety. He would have liked stronger static type checking of template arguments. So would I, but didn't see a way of getting that without limiting what could be expressed or compromising efficiency. In particular, I was - and am – very suspicious of attempts to limit template arguments to inheritance hierarchies.

后来，Alex 创造性地使用了我设计的模板特性，这导致了 STL 的产生，使得目前人们开始重视泛型（generic）及生成（generative）编程。和 Alex 争论很有意思！关于我对他风格的印象，参看 <http://www.stlport.org/resources/StepanovUSA.html>【注：这是一篇 STL 之父 Alexander Stepanov 的访谈录，内容相当激进，心脏不好的人请做好一切必要准备^_^。Alex 在 GP 上有极深的造诣，这篇访谈颠覆性不小，甚至可以看到他对 OO 的批判！也许彻底抛弃 OO 很难，但 Alex 的话极富启发性，值得一看】。

Later, Alex used the template features I designed in innovative ways that led to the STL, and to the current emphasis on generic and generative programming. Alex is always fun to argue with! For an impression of his style, see <http://www.stlport.org/resources/StepanovUSA.html>.

我试验过在不使用语言扩展的情况下约束模板参数的多种方式。我早期的想法在 *The Design and Evolution of C++* 一书中有叙述，其后期的变体成了现在普遍使用的约束和概念检查的一部分。这些系统在表现力和弹性上比常见于其他语言的内建设施要强太多。如果要举例的话，可以参看我的 *C++ Style and Technique FAQ* (http://www.research.att.com/~bs/bs_faq2.html#constraints)。

I experimented with ways of constraining template arguments without using language extensions. My early ideas are described in "The Design and Evolution of C++" and later variations are part of the now common uses of constraints and concept checking. These systems are far more expressive and flexible than built-in facilities found in other languages. For an example, see my "C++ Style and Technique FAQ" (http://www.research.att.com/~bs/bs_faq2.html#constraints).

Jerry Schwarz 在 *Standard C++ IOStream and Locales* 一书中的前言中回顾了 IOStream 的历史。我想在从经典流到标准 IOStream 的转变过程中一定有很多趣事，您能不能给我们讲一些呢？【注：此书由德国 Angelika Langer 和 Klaus Kreft 夫妇编著，是迄今为止该领域最权威和最完整的著作，中文版

【《标准 C++ 输入输出流与本地化》由人民邮电出版社出版。】

Jerry Schwarz reviewed the history of IOStream in the preface of the book *Standard C++ IOStream and Locales*. I guess that there must be many interesting stories in the process of transitioning from classic stream into the standard IOStream. Can you tell us some?

我不想再给 Jerry 对从我设计的流到目前的 IO 流转变的叙述添砖加瓦。然而，我想强调原来的流库简单而高效，我花了两个月的时间来设计和建构。

I do not want to try to add to Jerry's description of the transition from my streams to the current iostreams. Instead, I'd like to emphasize that the original streams library was a very simple and very efficient library. I designed and built it in a couple of months.

关键的决定在于格式与缓冲的分离，并使用类型安全的表达式语法(依赖于<<和>>运算符)。与 AT&T 贝尔实验室的同事 Doug McIlroy 探讨后，我做出了以上决定。实验表明，诸如<、>、逗号和=都不适合，后来我选择了<<和>>。类型安全允许一些原本在 C 风格库中需要在运行时决定的事，可在编译时决定，因而提供了非凡的性能。我刚开始使用流后不久，Dave Presotto 把我的实作的缓冲部分透明地替换成更好的，不过后来直到他告诉我，我才注意到这点。【注：请注意“透明(transparently)”这个词，也许这个翻译不是特别好，但是说明一点，Stroustrup 设计的这个流库相当出色，结构相当漂亮，甚至于库的一部分被换掉了，其功能丝毫不受影响，居然连其作者也没有察觉！】

The key decisions were to separate formatting from buffering, and to use the type-safe expression syntax (relying on operators << and >>). I made these decisions after discussions with my colleague Doug McIlroy at AT&T Bell Labs. I chose << and >> after experiments showed alternatives, such as < and >, comma, and = not to work well. The type safety allowed compile-time resolution of some things that C-style libraries resolve at run-time, thus giving excellent performance. Very soon after I started to use streams, Dave Presotto transparently replaced the whole buffering part of my implementation with a better one. I didn't even notice he'd done that until he later told me!

目前的 IO 流肯定小不了，不过我相信，在许多通常没有使用 IO 流全部通用性的情形下，借助于强力的优化，我们可以重获原来的效率。注意，IO 流那样的复杂度是为了应付我原来的经典流没有考虑的需求。例如，带本地化的标准 IO 流就可以处理经典流力不能及的汉字和汉字串。

The current iostreams library will never be small, but I believe that aggressive optimization techniques will allow us to regain the efficiency of the original in the many common cases where the full generality of iostreams is not used. Note that much of the complexity in iostreams exists to serve needs that my original iostreams didn't address. For example, standard iostreams with locales can handle Chinese characters and strings in ways that are beyond the scope of my original streams.

有人说 Java 是纯粹面向对象的，而 C#更胜一筹。而还有很多人说它们纯粹是面向金钱的。以您之见呢？

It's said that Java is purely object-oriented, while C# is even more. And many people say they are purely money-oriented. What's your opinion?

我喜欢“面向金钱”这个词 :-) 还有 Andrew Koenig 的说法“面向大话”我也喜欢。C++可不面向这两个东东。

I like the term "money-oriented" :-) I also like Andrew Koenig's phrase "buzzword-oriented". C++ is

neither.

对这点我还想指出，我认为纯粹性并非什么优点。C++显著地强项恰恰在于其支持多种有效的编程风格（多种思维模型吧，如果你一定要这么说）及其组合。最优雅最有效也最容易维护的解决方案常常涉及到一种以上的风格（编程模型）。如果一定要用吸引人的字眼，C++是一种多思维模型的语言。在软件开发的庞大领域，需求千变万化，起码需要一种支持多种编程的设计风格的通用语言，而且很可能需要一种以上呢。再说，世界之大，总容得下好几种编程语言吧？那种认为一种语言对所有应用和每个程序员都是最好的看法，根本就是荒谬的。【注：paradigm 的中文翻译似乎没有约定。ALNG 偏好“典范”或者“范式”，小编则喜欢侯捷先生使用的“思维模式”或者“思维模型”。总之，paradigm 的大概意思是 an example or pattern，大家理解就好。】

More to the point, I don't think "purity" is a virtue. The signal strength of C++ is exactly that it supports several effective styles of programming (several paradigms, if you must), and combinations of these styles. Often, the most elegant, most efficient, and the most maintainable solution involves more than one style (paradigm). If you must use fancy words, C++ is a multi-paradigm programming language. Given the wide variety of demands in the huge area of software development, there is a need for at least one general-purpose language supporting a range of programming and design styles, and probably for more than one such language. Also, there is room for many programming languages in the world. The idea that a single language is best for every application and every programmer is absurd.

Java 和 C#的主要强项是从其所有者那里得到的支持。这意味着低价（为取得市场份额免费发放实作和库），强力到无耻的营销（欺骗宣传），以及由于缺乏替代提供商产生的标准表象。当然，就 Java 的情形而言，当其他供应商和修改版出现后，版本、兼容性和移植问题也会像其他语言一样重新冒出来。

Java and C#'s main strengths are the support they receive from their owners. This implies a low price (implementations and libraries given away for free to gain market share), intensive and unscrupulous marketing (hype), and an appearance of a standard due to lack of alternative suppliers. However, when, as in the case with Java, other suppliers and revised versions eventually appear, versioning, compatibility, and portability problems re-emerge, as with other languages.

不被语言所有者操纵的开放进程所产生的正式标准是无可替代的。如果用户不想看到这种语言因为其所有者或者所谓“一般用户”的利益，不顾经济上无足轻重的“少数派”的反对而改来改去，像 ISO 这样的正式的标准进程，则是唯一的希望。

There is no substitute for formal standards, generated by an open process that is not manipulated by a language owner. A formal standards process, such as ISO's, is a user's only hope for a language that isn't jerked around for the benefit of its owner or for the benefit of "average users" over the objections of "minorities" deemed economically unimportant.

C++本可以简单点或容易使用点（更纯粹，如果你一定要这么说），不过这样就无法支持那些有着“不同寻常”的需求的用户了。我个人很关注这么一些人，他们要构建可靠性、运行效率以及可维护性远高于行业平均水准的系统。我的猜测是在 10 年的跨度中大多数程序员都将面临“不同寻常”的技术要求，他们可以从 C++的多思维模型结构受益，而 Java 和 C#之类“简化”语言则爱莫能助。

C++ could be simpler and easier to use (purer, if you must), but not while still supporting users with "unusual" demands. I am personally very concerned to support people building systems with demands for

reliability, run-time performance, and maintainability that are far greater than the industry average. My conjecture is that over the span of a decade most programmers will face "unusual" technical requirements that will benefit from C++'s multi-paradigm structure while not being well served by "simplified" languages such as Java and C#.

我认为模板和泛型编程是现代 C++ 的核心，是无损效率、类型安全代码的关键。然而它们并不适合经典的面向对象编程思维模型。

I consider templates and generic programming central to modern C++. They are the keys to uncompromisingly efficient, type-safe code. However, they don't fit the classical object-oriented paradigm.

Ian Joyner 在 *C++??: A Critique of C++ and Programming and Language Trends of the 1990s* 一书中比较了 C++ 和 Java 并批评了 C++ 的许多机制。你赞成他的观点吗？尤其是多数新语言都有垃圾收集机制，C++ 中会加入吗？

In the book *C++??: A Critique of C++ and Programming and Language Trends of the 1990s*, Ian Joyner compared C++ to Java and Eiffel and criticized many mechanisms of C++. Do you agree with him? Especially, most new languages have a garbage collection mechanism. Will it be added to C++? 【注：Ian Joyner 这本书的中文翻译就是本刊连载的“C++的不足之处讨论系列”。】

Ian Joyner 对 C++ 的观点，我不敢苟同。撇开这点，垃圾收集可能算是有价值的技术，不过并不是万能丹，它也会带来问题。对 C++ 而言，自动垃圾收集是一个有效的实作技术，有许多为 C++ 设计的不错的垃圾收集器（商业支持和免费的都有），而且也被广泛地使用（参看我的 C++ 页面上的链接）。然而 C++ 中垃圾收集机制应该是可选的，这样在不适合垃圾收集的地方，如严格的实时应用程序，可以免受其累。关于垃圾收集，我的 *The C++ Programming Language* 一书和我的主页上都用评注，可以参看。

No. I don't agree with Ian Joyner about C++. Independently of that, garbage collection can be a valuable technique, but it is not a panacea and it can also cause problems. Automatic garbage collection is a valid implementation technique for C++. Good garbage collectors exist for C++ (both commercially supported and free) and are widely used (see links on my C++ page). However, garbage collection is optional in C++ so that applications for which GC is unsuitable, such as hard real time applications, aren't burdened by it. See my comments about GC in "The C++ Programming Language (3rd Edition)" and on my home pages.

我期望下一个 C++ 标准中能大体上对我上面和其他地方说的内容做出明确的声明。

I expect that the next C++ standard will explicitly state roughly what I just said above and elsewhere.

就此而论，C++ 可以优雅地处理一般的资源，而不仅仅局限于内存。尤其是“resource acquisition is initialization”（资源获得就是初始化）技术（参看 D&E、TC++PL3 和我的技术 FAQ）支持对任意资源进行简单并且符合异常安全（exception-safe）要求的管理。没有析构函数的 Java 不可能支持这一技术。

In this context, it is worth noting that C++ has support for elegant techniques for handling resources in general, and not just memory. In particular, the "resource acquisition is initialization" technique (see D&E, TC++PL3, and my technical FAQ) supports simple, exception-safe management of arbitrary resources. Since Java lacks destructors it cannot support that technique.

STL 是一个超凡脱俗的跨平台架构。有没有考虑在其他方面，比如 GUI（图形用户接口），设计这样的标准架构？

STL is an excellent cross-platform framework. Have you considered designing such standard frameworks on other aspects, GUI for example?

很自然地，很多人会想如何在其他领域借鉴 STL 的成功。比如在数值运算和图论方面都有了许多有趣的工作。相关链接可以参看我的网页。

Naturally. Many have wondered how to replicate STL's success in other areas. For example, interesting work has been done in numerics and for graphs. See my C++ page for links.

标准 GUI 价值极大，不过我怀疑其政治上的可行性。太多有钱的大公司在维持其专有 GUI 上有着重大的商业利益，而且要求用户放弃现在所使用的 GUI 库也殊非易事。【注：有朋友可能奇怪，一个 GUI 库怎么扯出“政治 (politically)”来了？西方人口中的“政治”，在中文里并没有真正对应的词语。这里的意思是 of concerning public affairs，跟中文里的“政治”无关。下一段就是对这个所谓“政治上的可行性”的详细解释。】

A standard GUI would be of immense value, but I doubt that it is politically feasible. Too many rich companies have serious commercial interests in maintaining their proprietary GUIs. Also, users cannot easily change from what they are currently using.

这里我所说的可行性是就商业和技术而言。现在有好几种广泛使用的 GUI，即使标准委员会提供一个替代方案，它们也不会就此退出。其所有者和用户常常有充分理由会只是忽略新标准。更糟的情况：某些公司或群体会积极反对这样的标准，因为他们认为标准不如他们已有的库，或者因为差异太大而使得转换到新 GUI 不可行。必须理解，如果标准不能充分服务于其目标用户，用户会视而不见。许多 ISO 标准因为无人理会而变得无关紧要。C++标准可不想成为其中一员把现有实践拉近到一起，标准就功德无量了。我们不希望将来 ISO C++标准被人忽略。

What I refer to is what is commercially and technically feasible. There are several very widely used GUIs. They won't just go away if a standards committee decided on an alternative. Their owners and their users would - often for good reasons - simply ignore a new standard. Worse: some company or group of people might actively oppose such a standard because they considered it inferior to what they already had or simply too different for a switch to the new GUI to be feasible. It is important to understand that if a standard doesn't adequately serve its intended user, then those users will simply ignore them. Many ISO standards are irrelevant because nobody follows them. The C++ standard is not one of those - it is doing immeasurable good by pulling the implementations closer together - and we don't want the ISO C++ standard to be an ignored standard in the future.

注意 STL 成功的一个主要原因在于它是一个技术突破。它可不单是“又一个容器库”，因此它不需要和许多现有的容器库（其中几个品质卓著）直接竞争。我猜想 C++ 要有一个标准 GUI，我们需要技术突破，加上好运多多。

Note that one major reason that the STL succeeded was that it was a technical breakthrough. It wasn't simply "yet another container library", so it didn't have to compete directly against the many existing container libraries (several of which were of excellent quality). My guess is that for C++ to get a standard GUI, we need a technical breakthrough plus a lot of luck.

不过我还是怀疑委员会有由必需的专业技术和资源来构建一个可以成为真实世界中真正标准的 GUI. However, I still doubt that the committee has the technical expertise and the resources necessary to produce a GUI that could become a real standard in the real world.

我对标准库的想法倾向于修补现有库的遗漏（如 hash_map 和正则表达式），以及通过更广泛的运行时间类型信息和并发库来支持分布运算（可选）。

My thoughts for the standard library goes more towards filling in gaps in the current library (e.g. hash_map and regular expressions) and support for distributed computing through more extensive (optional) run-time type information and concurrency libraries.

有时大家忘了，库不是非得成为标准的一部分才有用。有成千上万有用的 C++ 库。例如，参 C++ 库 FAQ（我的 C++ 网页有链接）

People sometime forget that a library doesn't have to be part of the standard to be useful. There are thousands of useful C++ libraries. For example, see the C++ libraries FAQ (link on my C++ page).

泛型编程是 C++ 特殊的编程思维模型。你是怎样看 GP(泛型编程) 和 OO(面向对象) 的？将来 C++ 会提供更强大的机制来支持 GP 吗？有没有考虑引入其他思维模型，比如面向模式？

Generic programming is a special paradigm in C++. What do you thinking of GP and OO? Will C++ provide more powerful mechanisms to support GP in the future? And have you considering importing other paradigms, pattern-oriented for example?

我认为，在 C++ 中面向对象和泛型编程相互补充得极好，我所写的许多最优美的代码都依赖于两者的结合。也就是说，认为 OOP 和 GP 水火不容的观点，是错误的。它们是应该组合使用的技巧，语言应该支持这样的组合 C++ 正是如此。

I think that object-oriented and generic programming complements each other nicely in C++, and many of my most elegant pieces of code relies on combinations of the two. That is, it is wrong to think of OOP and GP as completely distinct paradigms. They are techniques that should be used in combination, and a language should support such combinations - as C++ does.

我觉得 C++ 相当好地支持了泛型编程，所以只需要细微的增加。模板化的 typedef 是个显而易见的例子。我们要谨慎地扩展语言，仅当扩展对要表述的内容提供重大的便利时，我们才这样做。比如我不支持对模板参数约束检查提供直接语言支持的想法。通过约束 / 概念检查模板，我们已经可以比用为 C++ 和相似的语言提议的各种各样的语言扩展做得更多。

I think that C++ supports generic programming rather well, so that it needs only minor additions. An obvious example is templated typedefs. We have to be careful to extend the language only where extensions provide major advantages in what can be expressed. For example, I don't support ideas of direct language support for template argument constraints checking. We can already do more with constraints/concept checking templates than could be done with the various language extensions proposed for C++ and similar languages.

谈起“思维模型”和“新的思维模型”让我很为难，只有很少的想法配得上这样美妙的字眼。我也担心对新观念过于直接的支持，可能会限制和跟不上我们的观念和技术的进一步演化。理想的情况是，语言设施应有效地支持非常通用的观念，这样大家可以使用这些设施用各种风格来编写代码。

至于 C++ 能优雅地支持哪些模式概念，能和不能通过与已有风格的组合，还有待观察。我认为，只需要很少新的特定语言概念来支持模式。

I'm very reluctant to talk about "paradigms" and "new paradigms" - very few ideas deserve such fancy terms. I also worry that too direct support of new ideas can be limiting and failing to cater for further evolution of our ideas and our techniques. Ideally, language facilities should support very general ideas efficiently so that people can use those facilities to write code in a variety of styles. I think that it remains to be seen what patterns ideas can and cannot be supported elegantly through a combination of styles already supported in C++. I suspect that very few new language concepts are needed specifically to support patterns.

今后 C++ 会支持分布开发吗？对 RTTI 和多线程的进一步支持呢？

Will C++ support the disturbed development later? And what about further support for RTTI and multi-thread?

对。如果事情进展能如我所愿，C++ 标准的下一次修订会通过提供扩展的类型信息和并发支持库来支持分布计算。我觉得这不需要特别的语言扩展。不过在存在并发的情况下现有语言设施实作需要做出额外的保证。

Yes. If things progress as I hope they will, the next revision of the C++ standard will support distributed computing through the provision of extended type information and concurrency-support libraries. I do not think this will require specific language extensions. Making additional guarantees about the implementation of existing language facilities in the presence of concurrency will be needed, though.

我没有太多可说，因为围绕下一标准应该和不该包含哪些的讨论才刚刚开始。我的看法是 C++ 需要一个无缝地支持线程（在同一地址空间内）进程（在不同地址空间）及远端进程（可能有重大的通讯延时而且网络可能暂时分离）的标准库。支持这点会需要超越简单的 Unix 或 Windows 线程的设施。但是我并不认为这需要设计新的语言元件。

There is not much that I can add to that now, because the discussions about what should and should not be in the next standard have just started. My view is that C++ need a standard library that seemlessly support threads (within a single address space), processes (with separate address spaces), and remote processes (where communication delays can be significant and where a network may become separated for a while). Supporting this will require facilities beyond simple Unix or Windows threads. However, I don't think it need involve new language primitives.

最近一个叫做 YASLI 的项目启动了。YASLI 代表“又一个标准库实作”，其目的是成为新一代的 C++ 标准库实作。您对此有何感想？【注：这个项目最初的想法来自于今年 5 月份 Andrei Alexandrescu 在 news://comp.lang.c++.moderated 上的讨论，详细信息可以在 <http://www.stlport.org> 上查找，也可参考

<http://www.stlport.com/cgi-bin/forum/dcboard.cgi?az=list&forum=DCForumID10&conf=DCConfID2>。】

Recently a new project called YASLI which stands for "Yet Another Standard Library Implementation" has been started, that intents to be the new generation of C++ Standard Library implementation. What do you think about it?

知之甚少，无从说起。

I don't know enough about that project to have an opinion.

据说大人物年轻时就会表现出与常人的差异，请问您在大学就读时表现过什么与众不同的地方？

It's believed that great men would show their differences against others when they are young. So what differences did you show when studying in the universities?

我不清楚是否有人认为我显著得与众不同。我猜想，我可能比多数人天真和理想主义那么一点点。另外比之多数人，我在解决现实问题的时间会多一点吧 我要挣钱以免陷入债务。我可不能债台高筑，因为我家不算富有，我一直被要求努力工作。另一方面，我倾向于学习我感兴趣的多种东西（包括哲学和历史），而不仅只是那些直接有助于我取得学位和提高成绩的东西。

I'm not sure anyone considered me as significantly different from others. I suspect that I was a bit more naive and idealistic than most. I also spent more time working on practical problems than most - to earn money to avoid getting into debt. Not building up debt was important for me because I don't come from a rich family. I have been told that I worked hard. On the other hand, I tended to work on a variety of things that interested me (including philosophy and history) rather than just on things that directly helped me finish my degree or improve my grades.

喜欢安徒生童话吗？在《夜莺》里他写到了中国。您对中国、中华文化和中国人的印象如何？以前去过中国吗？2008年来中国看奥运会可能是个不错的主意。

Do you like reading Andersen's fairy stories? He wrote something about China in the story of The Nightingale. So what's your impression about China, the Chinese culture and the Chinese people? Have you ever been to China before? Maybe visiting China for the Olympics in 2008 would be a good idea.

作为丹麦人，我自然知道安徒生童话。我刚好也很喜欢它们。《夜莺》中描绘的中国纯是虚构，与当时的中国可能有也可能没有任何关系。安徒生创造了那个“中国”来泛指多个国家及其统治者。

As a Dane, I naturally know Hans Christian Andersen's tales. I also happen to like them. The China described in "The Nightingale" is a fiction that may or may not have anything to do with the China that then existed. Andersen created that "China" to be able to make universal points about countries and their rulers.

很难对象中国这么巨大的概念有一个印象。我遇到的中国人大都是程序员或工程师，因此我对中国人民的视野可能过于狭窄，有误导之嫌。哪怕是象我的本国丹麦这样的小国和文化体，其复杂程度也不是某个人可以完全理解的 只有 500 万丹麦人。我对历史很感兴趣，因此也看了几本中国历史和文化题材的书。不过这可能意味着我头脑里的中国古多于今。我在台湾讲过一个星期的学，喜欢呆在那里，不过还没有机会访问大陆。

It is hard to have *one* impression about something as huge as China. The Chinese that I have met are mostly programmers or engineers, so I probably have a misleadingly narrow view of Chinese people. Even a small country and culture as my native Denmark is too complex for any individual to fully comprehend - and there are only 5 million Danes. I have a strong interest in history, so I have read several books on Chinese history and culture. However, that implies that ancient China may loom larger in my mind than it should compared to modern China. I lectured in Taiwan for a week and enjoyed my stay there, but I have not yet had the opportunity to visit the mainland.

关于中国历史和文化的书我看过不少。因为中国历史悠久、幅员辽阔，主要的焦点集中于早期的事件、人和传统，几乎没有描绘近 10 或 20 年的中国。从新闻和中国朋友那里获知发生了巨大的变化，我对今日中国的无知是巨大的（尽管可能不象多数人对远方国度那么无知）。比如我对当今中国的文学和音乐一无所知。因而，想到中国时我想起的很多东西可能都严重过时 尽管我极其小心地想避免此类错误。顺便说一下，我对主要从书本上获知的世界其他地区也有类似的问题。【注：Stroustrup 博士对中国历史有相当的了解，在谈论姓“王”的中国人时，我提到世界第二大软件公司 CA 的总裁王嘉廉（Charles Wang），他则提及王安石。Stroustrup 博士说“对当今中国的文化和音乐一无所知”，小编就暂时推荐了两首流行音乐。由于 Stroustrup 博士喜欢安徒生童话，小编就首先推荐了熊天平的《火柴天堂》，以《卖火柴的小女孩》为背景。另外一首《长城》，来自取得华语流行音乐最高成就的 Beyond 乐队（不过随着家驹的离去，这已是永远的回忆了）。新近的歌嘛，孙燕姿的《风筝》蛮好听的，下次再说吧，呵呵。】

I have read many books about Chinese history and culture. Because of the length of Chinese history and the size of China, most focus on events, people, and traditions of earlier times, and hardly any describe China as it has been for the last ten or 20 years. I know from the news and from Chinese friends that major changes have happened and that my ignorance of current-day China is immense (though probably not as immense as most people's ignorance about far-away countries). For example, I have no idea of current Chinese literature or music. Thus, when I think of China, some of what I think may be seriously out of date - despite the care I obviously take to avoid such mistakes. By the way, I have similar problems for other regions of the world that I also know of primarily from books.

我对大型人群和有组织的群体事件不太热心，因此我会远离 2008 年奥运会，就象我远离本可以参加的其它各届奥运会一样。希望能找其他某个时间访问中国。

I'm not keen on huge crowds and organized mass events, so I'll stay far away from the 2008 Olympics, just as I stayed away from every other Olympic games that I might have attended. I hope to find some other time to visit China.

编后：Stroustrup 博士的经典名著 *The C++ Programming Language*，最新是第 3 版。同时，大家还可以看到所谓的“特别版”。特别版与第 3 版的不同之处在于，封面不同，并且多了两篇附录：Locales（本地化）和 Standard-Library Exception Safety（标准库的异常安全），可以在 Stroustrup 博士的主页上下载。第 3 版的简体中文版由南京东南大学计算机科学与工程系的徐宝文教授翻译，机械工业出版社出版。据徐教授透露，本书争取在今年年底或明年初出版。价钱嘛，呵呵，还得出版社来定。至于特别版比第 3 版多出的两篇附录是否加入中文版的问题，仍在与出版社协商中。Stroustrup 博士专门为中文版作了序，限于篇幅，此处仅列出最后一段。

I am pleased to write a foreword to Professor Xu's Chinese translation of 'The C++ Programming Language'. I am aware that my books have been available to Chinese computer professionals for many years, but for most students it is a great help to be able to have a textbook available in their native language. I am therefore very happy that this authorized translation makes C++ much more accessible to Chinese students, programmers, and other computer professionals.

C++空成员类优化

Nathan C. Myers

翻译：张岩

编者注：空成员类大小是 0 吗？往往都不是。然而在 STL 中 traits 等模板技术的大量使用，空成员类并不“空”的问题却大大限制了性能的发挥。那么，什么时候空成员类该“空”或不“空”呢？为什么？本文将对此作详细的介绍。原文曾经刊登在 97 年 8 月 Dr. Dobb Journal 的 C++ Issue 专栏上，翻译经 Nathan Myers 先生同意，参考 <http://www.cantrip.org/emptyopt.html>。

C++标准库（草案）中包含着各种有用的模板，其中包括获奖的 STL（见 DDJ Mar95）中的一些扩展版本。这些模板提供了非常好的弹性，现在，它们在实际应用中被最优化来获取最高的性能。和它们在程序中的用处一样，它们作为表达有效设计的实例，或者作为使你自己的组件富有弹性并同时具有效率的灵感的源泉，也是非常有益的。

提供弹性的一种方式就是使用“空”类——没有数据成员的类。理想情况下，这些类不应该耗费任何储存空间。它们通常是定义了一些 `typedef` 或者是一些成员函数，然后，你用自己的类（不能确定是否为空类）来替换它们从而满足某种特殊需求。默认的空成员类是到现在为止最经常用到的，所以，这一情况必须被最优化，这样我们才可以不都需要为不经常遇到的特殊需求付出代价。

由于一个不幸的语言定义的细节（稍后作详细的解释），空成员类的实例往往会占据空间。在其它的类的成员中，这种空间的开销会变成其它方式——小的对象变的足够大以致无法在某些环境中应用。如果这种开销不能在构造标准库的时候避免的话，由库的弹性而引入的效率开销会让许多用户望而却步。在标准库中应用的优化技术也同样适合你自己的代码。

空成员膨胀

这里有一个例子，在标准 C++ 库中，每一个 STL 容器类的构造函数都包含并拷贝一个 `allocator` 参数。当容器需要空间的时候，它就向它的 `allocator` 成员索取。这样，有对内存分配有特殊要求的用户（比如共享区间）可以为它定义一个 `allocator`，并且把它传入一个容器的构造函数，这样容器元素就可以被放置于其中。在普通条件下，标准的默认 `allocator` 被使用，它将申请工作委托给全局的 `operator new`。它是一个空对象。

这里的简短的代码是一个可能的实现。在 Listing 1 中，标准的 `allocator allocator<>`，只有一个成员函数。

Listing 1: The Standard Default Allocator

```
template <class T>
```

```

class allocator { // an empty class
    . .
    static T* allocate(size_t n)
    { return (T*) ::operator new(n * sizeof T); }
    . .
};


```

在 Listing 2 中，广义表的容器模板中维护了一个私有的 allocator 成员，这是从它的构造函数参数中拷贝得到的。

Listing 2: The Standard Container list<>

```

template <class T, class Alloc = allocator<T> >
class list {
    . .
    Alloc alloc_;
    struct Node { . . . };
    Node* head_;

public:
    explicit list(Alloc const& a = Alloc())
        : alloc_(a) { . . . }
    . .
};


```

注：

1. list<>的构造函数被声明为“`explicit`”，这样编译器不会将它用作为一个自动转型。
(这是一个新的语言特性。)
2. list<>是如何从一个能为 `T` 提供空间的allocator中，为一个Node对象分配空间的，将是
下一篇文章的主题。

成员 `list<>::alloc_` 在对象中通常占据四个 byte 的空间，即使是在 `Alloc` 是一个空成员类 `allocator<T>` 的默认情况下。list 对象的一些额外空间看起来并不是很糟糕，直到你想到一个包含许多 list 对象的 vector 时，或者是在一个 hash 表中，你才会考虑到严重性。每一个 list 中的多余的垃圾都作了被乘数，并且降低了虚拟内存页面中的 list 头的数目。浪费的空间使程序变慢。

空对象

这种开销如何才能避免呢？同样，这需要理解开销产生的原因。标准 C++ 语言定义如是说：

一个有着空的成员序列和基类对象的类称为空类。一个空类的完整的对象和成员子对象是非零长度。

为什么没有成员数据的对象要占据空间呢？考虑下面的代码：

```
struct Bar { };
struct Foo {
    struct Bar a[2];
    struct Bar b;
};
Foo f;
```

`f.b` 和 `f.a[]` 的元素的地址是什么呢？如果 `sizeof(Bar)` 是零的话，它们可能就会有相同的地址。如果是借助地址来跟踪不同的对象的话，那么 `f.b` 和 `f.a[0]` 会显现得像是同一个对象。标准委员会选择了通过禁止零长度可寻址对象来解决这些问题。

尽管如此，为什么一个空成员会占据如此多的空间（4个byte，在我们的例子中）？在所有我知道的编译器中，`sizeof(Bar)` 是1。然而，在大多数体系下，对象需要根据它们的类型进行对齐。例如，如果声明：

```
struct Baz {
    Bar b;
    int* p;
};
```

当今大多数硬件体系下，`sizeof(Baz)` 是8，这是因为编译器补齐（padding）使成员 `Baz::p` 不跨越一个字（word）的边界。（见图1a）

图1a：

```
struct Baz
```



图1b：

```
struct Baz2
```



如何避免这一开销？草案在注脚处暗示：

一个空类的基类的子对象 (base class subobject) 可以是零长度的。

换句话说，如果像这样声明 Baz2，

```
struct Baz2:Bar {  
    int* p;  
};
```

这钟情况，编译器就允许为空基类保存 0 byte；因此，`sizeof(Baz2)` 在大多数机器上只为 4。（见图 1b）

编译器提供商不是必须来完成这个优化，并且到目前，许多厂商没有这样做。然而，你可以期待最遵循标准的编译器会这样做，因为标准库中的许多组件（不仅仅是容器）的效率依赖这一优化。

消除膨胀

我们已经找到了消除空间开销所需的原理。怎样执行呢？让我们先考虑如何来修正在我们例子中的实现，模板的 `list<>`。可以直接从 allocator 中继承，如 Listing 3 中那样，

Listing 3: A Native Way to Eliminate Bloat

```
template <class T, class Alloc = allocator<T> >  
class list : private Alloc {  
    . . .  
    struct Node { . . . };  
    Node* head_;  
  
public:  
    explicit list(Alloc const& a = Alloc())  
        : Alloc(a) { . . . }  
    . . .  
};
```

并且，这通常会起作用。`list<>` 中成员函数的代码会用 `this->allocate()` 代替原来的 `alloc_.allocate()` 来获得空间。然而，用户提供的 `Alloc` 类型允许含有 `virtual` 成员，这将会与派生的 `list<>` 成员冲突。（可以设想一个私有成员 `void list<>::init()`，和一个 `virtual` 成员 `bool Alloc::init()`。）

一个好得多的解决方法是用 `list<>` 的数据成员将 `allocator` 封装，例如指向第一个节点的指针（如 Listing 4 所示），这样 `allocator` 接口将不会外露。

Listing 4: A Better Way to Eliminate Bloat

```
template <class T, class Alloc = allocator<T> >
class list {
    . . .
    struct Node { . . . };
    struct P : public Alloc {
        P(Alloc const& a) : Alloc(a), p(0) { }
        Node* p;
    };
    P head_;

public:
    explicit list(Alloc const& a = Alloc())
        : head_(a) { . . . }
    . . .
};
```

现在，`list<>`的成员通过“`head_.allocate()`”来获得储存空间。并且通过“`head_.p`”来获得第一个元素。这非常完美了，没有任何不必要的开销，`list<>`的用户感觉不到有任何不同。和其他任何出色的优化一样，它使实现变的有一点不整洁。但是这不会影响到接口。

一个封装的解决方案

到现在，还有可以改进的余地。和往常一样，改进引入了模板。在 Listing 5 中我们将这种技术封装让它更容易，更简洁地被应用。

Listing 5: Packaging the Technique

```
template <class Base, class Member>
struct BaseOpt : Base {
    Member m;
    BaseOpt(Base const& b, Member const& mem)
        : Base(b), m(mem) { }
};
```

使用这个模板，我们的 `list<>` 声明将像 Listing 6 种所示，

Listing 6: The Best Way to Eliminate Bloat

```
template <class T, class Alloc = allocator<T> >
class list {
    . . .
```

```
struct Node { . . . };
BaseOpt<Alloc,Node*> head_;
```

public:

```
explicit list(Alloc const& a = Alloc())
: head_(a,0) { . . . }
```

. . .

```
} ;
```

这一 `list<>` 的声明并没有比我们的最初的未优化版本更大、更杂乱。任何其他的库组件（标准的或非标准的）可以简单地使用 `BestOpt<>`。成员代码只有一点杂乱；同时已开始很难看清发生了什么，`BestOpt<>` 的声明提供了一个方法来文档化这个技术和原因。

在 `BestOpt<>` 中加入成员是一件十分诱人的事情，但是这不会改善它：她们会如 Listing 3 所展现的那样，出现派生类和基类参数的冲突。

最后

这种技术在任何很好支持模板技术的编译器下都可以被应用。并不是所有的 C++ 编译器都可以支持空成员基类优化，虽然 Sun, HP, Microsoft 的编译器都已经这样做了。但这这种技术即使是在编译器不支持的情况下也不会产生额外的开销。当你手头有一个能很好支持标准的编译器的时候，它很有可能会做出这种优化，如果你的代码利用了这种技术，它们会自动变得更有效率。

Fergus Henderson 在这项技术细化的方面做出了重要的贡献。

后记：最新的 Borland 编译器可以通过一个模式切换来支持这项优化；在这个版本中，它的默认选项是关闭的。Watcom 和 Symantec 的编译器可以支持这种优化。Metaware 的编译器可以在 OS/2 系统上完成此优化。Egcs 编译器在-fnew-abi 选项打开（这将会变成缺省值）的时候支持此优化。Apple 的 MrCpp 支持。Metrowerks 4.0 编译器支持，但是（像 IBM 的一样）它有一些过于大胆，竟然将两个相同类型的子对象（subobject）放到同一地址上。据一位 Watcom 的工程师反映，在实现空基类优化后，STL 的基准速度提高了 30%。

深入 VCL 理解 BCB 的消息机制（三）

CKER

方法 3 来自 TApplication 的方法

不用我多废话，大家都知道 TApplication 在 BCB 中的重要性。在 BCB 的帮助中指出：TApplication、TScreen 和 TForm 构成了所有 BCB 风格的 Win32 GUI 程序的脊梁，他们控制着您程序的行为。TApplication 类提供的属性和方法封装了标准 Windows 程序的行为。TApplication 表现了在 Windows 操作系统中创建、运行、支持和销毁应用程序的基本原理。因此，TApplication 大大简化了开发者和 Windows 环境之间的接口。这正是 BCB 的 RAD 特性。

TApplication 封装的标准 Windows 行为大致包括如下几部分：

- 1> Windows 消息处理
- 2> 上下文关联的在线帮助
- 3> 菜单的快捷键和键盘事件处理
- 4> 异常处理
- 5> 管理由操作系统定义的程序基础部分，如：MainWindow 主窗口、WindowClass 窗口类等。

一般情况下，BCB 会为每个程序自动生成一个 TApplication 类的实例。这部分源码可以在 yourproject.cpp 文件中见到(这里假定您的工程名称就叫 yourproject.bpr)。

当然 TApplication 是不可见的，他总是在您的 Form 背后默默的控制着您的程序的行为。但也不是找不到蛛丝马迹。如果您新建一个程序(New Application)，然后不作任何改动，编译运行的话，你会发现程序窗体的 Caption 是 Form1，但在 Windows 的状态条上的 Caption 确写着 project1 的字样。这就是 TApplication 存在的证据。当然，这只是一个臆测，实战的方法应该打开 BCB 附带的 WinSight 来查看系统的进程。您可以清楚的看到 TApplication 类的存在，他的大小是 0(隐藏的嘛)，然后才是 TForm1 类。

好了，既然 TApplication 封装了消息处理的内容。我们就研究一下 TApplication 的实际动作吧。实际上消息到达 BCB 程序时，最先得到它们的就是 TApplication 对象。经由 TApplication 之后，才传递给 Form 的。以前的方法都是重载 TForm 的方法，显然要比本文所提到的方法要晚一些收到消息。对您来说，是不是希望在第一时间收到消息并处理它们呢？

要清楚的知道 TApplication 的处理机制还是深入 VCL 源码。首先看一看最最普通的一段代

码吧。

```
#include <vcl.h>
#pragma hdrstop
USERES("Project1.res");
USEFORM("Unit1.cpp", Form1);
//-----
WINAPI WinMain(HINSTANCE, HINSTANCE, LPSTR, int)
{
    try
    {
        // 初始化 Application
        Application->Initialize();
        // 创建主窗口，并显示
        Application->CreateForm(__classid(TForm1), &Form1);
        // 进入消息循环，直到程序退出
        Application->Run();
    }
    catch (Exception &exception)
    {
        Application->ShowException(&exception);
    }
    return 0;
}
```

短短的几行代码就可以让您的 BCB 程序自如运行。因为一切都已经被 VCL 在后台封装好了。Application->Run()方法进入程序的消息循环，直到程序退出。一起跟进 VCL 源码看个究竟吧。

TApplication 的定义在 forms.pas 中。

```
procedure TApplication.Run;
begin
    FRunning := True;
    try
        AddExitProc(DoneApplication);
        if FMainForm <> nil then
            begin
                // 设置主窗口的显示属性
                case CmdShow of
                    SW_SHOWMINNOACTIVE: FMainForm.FWindowState := wsMinimized;
                    SW_SHOWMAXIMIZED: MainForm.WindowState := wsMaximized;
                end;
                if FShow MainForm then
```

```

if FMainForm.FWindowState = wsMinimized then
    Minimize else
    FMainForm.Visible := True;
// 看见了吧，这里有个循环，直到Terminated 属性为真退出。Terminated 什么意思，就是取消，结束
repeat
    HandleMessage
until Terminated;
end;
finally
    FRunning := False;
end;
end;

```

消息处理的具体实现不在 Run 方法中，很显然关键在 HandleMessage 方法，看看这函数名字 - 消息处理。只有跟进 HandleMessage 瞧瞧喽。

```

procedure TApplication.HandleMessage;
var
    Msg: TMsg;
begin
    if not ProcessMessage(Msg) then Idle(Msg);
end;

```

咳，这里也不是案发现场。程序先将消息交给 ProcessMessage 方法处理。如果没什么要处理的，就转入 Application.Idle 方法“程序在空闲时调用的方法”。

呼呼，再跟进 ProcessMessage 方法吧。

```

function TApplication.ProcessMessage(var Msg: TMsg): Boolean;
var
    Handled: Boolean;
begin
    Result := False;
    if PeekMessage(Msg, 0, 0, 0, PM_REMOVE) then
begin
    Result := True;
    if Msg.Message <> WM_QUIT then
begin
        Handled := False;
        if Assigned(FOnMessage) then FOnMessage(Msg, Handled);
        if not IsHintMsg(Msg) and not Handled and not IsMDIMsg(Msg) and
not IsKeyMsg(Msg) and not IsDlgMsg(Msg) then

```

```
begin
    TranslateMessage(Msg);
    DispatchMessage(Msg);
end;
else
    FTerminate := True;
end;
end;
```

哎呀呀，终于有眉目了。ProcessMessage 采用了一套标准的 Windows API 函数 PeekMessage TranslateMessage;DispatchMessage。

有人说：Application->OnMessage = MyOnMessage; //不能响应 SendMessage 的消息，但是可以响应 PostMessage 发送的消息，也就是消息队列里的消息

SendMessage 和 PostMessage 最主要的区别在于发送的消息有没有通过消息队列。

原因就在这里。ProcessMessage 使用了 PeekMessage(Msg, 0, 0, 0, PM_REMOVE) 从消息队列中提取消息。然后先检查是不是退出消息。不是的话，检查是否存在 OnMessage 方法。如果存在就转入 OnMessage 处理消息。最后才将消息分发出去。

这样重载 Application 的 OnMessage 方法要比前两种方法更早得到消息，可以说是最快捷的方法了吧。举个例子：

```
void __fastcall TForm1::MyOnMessage(tagMSG &Msg, bool &Handled)
{
    TMessage Message;
    switch (Msg.message)
    {
        case WM_KEYDOWN:
            Message.Msg = Msg.message;
            Message.WParam = Msg.wParam;
            Message.LParam = Msg.lParam;
            MessageDlg("You Pressed Key!", mtWarning, TMsgDlgButtons() << mbOK, 0);
            Handled = true;
            break;
    }
}

void __fastcall TForm1::FormCreate(TObject *Sender)
```

```
{  
    Application->OnMessage = MyOnMessage;  
}
```

现在可以简短的总结一下 VCL 的消息机制了。

标准的 BCB 程序使用 Application->Run() 进入消息循环，在 Application 的 ProcessMessage 方法中，使用 PeekMessage 方法从消息队列中提取消息，并将此消息从消息队列中移除。然后 ProcessMessage 方法检查是否存在 Application->OnMessage 方法。存在则转入此方法处理消息。之后再将处理过的消息分发给程序中的各个对象。至此，WndProc 方法收到消息，并进行处理。如果有无法处理的交给重载的 Dispatch 方法来处理。要是还不能处理的话，再交给父类的 Dispatch 方法处理。最后 Dispatch 方法实际上将消息转入 DefaultHandler 方法来处理。

“ 嘿嘿，实际上，你一样可以重载 DefaultHandler 方法来处理消息。但是太晚了一点。我想没有人愿意最后一个处理消息吧...:-) ”

写到这里似乎可以结束了。但如果您看过上一篇的话，一定会注意到 Application->HookMainWindow 方法。这又是怎么一回事呢？

如果您打算使用 Application->OnMessage 来捕获所有发送至您的应用程序的消息的话，您大概要失望了。原因已经讲过，它无法捕获使用 SendMessage 直接发送给窗口的消息，因为这不通过消息队列。您也许会说我可以直接重载 TApplication 的 WndProc 方法。呵呵，不可以。因为 TApplication 的 WndProc 方法被 Borland 申明为静态的，从而无法重载。显而易见，这么做的原因很可能是 Borland 担心其所带来的副作用。那该如何是好呢？

查看 TApplication 的 WndProc 的 pascal 源码可以看到：

```
procedure TApplication.WndProc(var Message: TMessage);  
... // 节约篇幅，此处与主题无关代码略去  
begin  
try  
    Message.Result := 0;  
    for I := 0 to FWindowHooks.Count - 1 do  
        if TWindowHook(FWindowHooks[I]^)(Message) then Exit;  
    ... // 节约篇幅，此处与主题无关代码略去
```

WndProc 方法一开始先调用 HookMainWindow 挂钩的自定义消息处理方法，然后再调用缺省过程处理消息。这样使用 HookMainWindow 就可以在 WndProc 中间接加入自己的消息处理方法。使用这个方法响应 SendMessage 发送来的消息很管用。最后提醒一下，使用 HookMainWindow 挂钩之后一定要对应的调用 UnhookMainWindow 卸载钩子程序。给个例子：

```
void __fastcall TForm1::FormCreate(TObject *Sender)
{
    Application->HookMainWindow(AppHookFunc);
}

bool __fastcall TForm1::AppHookFunc(TMessage &Message)
{
    bool Handled ;
    switch (Message.Msg)
    {
        case WM_CLOSE:
            mrYes==MessageDlg( "Really Close??", mtWarning, TMsgDlgButtons() << mbYes
<<mbNo, 0)? Handled = false : Handled = true ;
            break;
    }
    return Handled;
}

void __fastcall TForm1::FormDestroy(TObject *Sender)
{
    Application->UnhookMainWindow(AppHookFunc);
}

void __fastcall TForm1::Button1Click(TObject *Sender)
{
    SendMessage(Application->Handle,WM_CLOSE,0,0);
}
```

这样，将本文中的两种方法相结合，您就可以自如的处理到达您的应用程序的各种消息了。（全文完）

编后：《深入 VCL 理解 BCB 的消息机制》全文都已刊出，非常感谢 CKER 为大家所作的精彩介绍，让我们得以深入 VCL 内部，系统地了解整个消息机制，并且知道了许多处理消息的方法及其背后的原理。从这一期开始，C++ View 将推出从另一角度剖析 VCL 的系列文章《天方夜谭 VCL》，希望得到大家一如既往的关注和支持。同时，也欢迎大家赐稿，把您对自己喜欢的应用框架的感悟，比如 MFC、Qt、Gtk+等等，发 email 告诉我们，与大家共同分享。

Generic<Programming>: Traits on Steroids

Andrei Alexandrescu

yefeng 译

编者按：以前我们介绍了 traits 的一些基本用法，大家感觉如何？这次我们接着介绍 traits 的一些更为有趣的用途，让大家兴奋一下（Steroid 是类固醇，常见的兴奋剂）。Generic<Programming>系列的部分资料已经收录在大师 Andrei Alexandrescu 的大作 *Modern C++ Design: Generic Programming and Design Patterns Applied* 一书里，中文版由侯捷和孟岩两位大虾翻译，估计在明年上半年就可与读者见面。感谢 Andrei Alexandrescu 先生的授权。英文原文发表在 C++ Report 2000 年第 6 期上，可参考 http://www.joopmag.com/html/from_pages/crarticle.asp?ID=773。

在上期的 Generic<Programming>^[1] 中，我们讨论了 traits 模板和 traits 类。这篇文章进一步讨论 traits 对象和全层次（hierarchy-wide）traits。

Traits 技术很有用，但是什么时候你需要这种非凡的灵活性呢？如果你用了 traits，你怎么才能避免手工向现有类层次中的大量的类添加 traits 的苦差事呢？这篇文章以上一次的 SmartPtr 为例，解答这些问题。特别是介绍了全层次（hierarchy-wide）traits，一项非常 cool 的 C++新技术，可以让你一下子为整个类层次而不仅是单个类定义 traits。

[回到 SMARTPTR](#)

上一次的专栏里介绍了一个 smart pointer，它可以根据客户对模板实例化，的方式不同而用于单线程或多线程的代码中。再来看一下 SmartPtr 的定义：

```
template <class T, class RCTraits = RefCountingTraits<T> >
class SmartPtr
{
    ...
};
```

用 RefCountingTraits 可以对 SmartPtr 进行定制，以适应不同类型 T 所使用的引用计数的语法和语义。如果你要在单线程代码中用 SmartPtr，RefCountingTraits 就可以了。否则，你必须另外提供一个 traits 类（MtRefCountingTraits）作为第二个模板参数。MtRefCountingTraits 保证在多线程情况下，引用计数是安全的。

```
class MtRefCountingTraits
{
    static void Refer(Widget* p)
    {
        // serialize access
        Sentry s(lock_);
        p->AddReference();
    }
};
```

```

    }
    static void Unrefer(Widget* p)
    {
        // serialize access
        Sentry s(lock_);
        if (p->RemoveReference() == 0)
            delete p;
    }
private:
    static Lock lock_;
} ;

```

对于单线程的 widget，客户代码可以用 `SmartPtr<Widget>`，对于多线程的 widget，可以用 `SmartPtr<Widget, MtRefCountingTraits>`。如果没有上一篇文章最后留下的那个问题，事情就这样简单。那个问题是：在多线程版的 `SmartPtr` 里，哪一部分还是低效的？

正如很多读者指出的那样，问题在于 `MtRefCountingTraits` 用了类级别的加锁操作。Herb Sutter 形象地说：`Static lock, bad juju!`【juju，符咒】当你进行串行化的操作时，比如 `MtRefCountingTraits::Refer()`，类级别的锁会把 `MtRefCountingTraits` 类的所有对象都锁住，因为 `lock_` 是所有 `MtRefCountingTraits` 实例共享的静态变量。

如果你有很多线程频繁的操作 `widget` 的 smart pointer，这可能会成为程序低效的一个根源。原本可能完全无关的线程在复制 `SmartPtr<Widget, MtRefCountingTraits>` 对象时，也必须排队等待。

在对象级别上加锁是解决这个问题的一个方法。要使用对象级别的锁，只需把 `MtRefCountingTraits` 的成员 `lock_` 改成非静态的普通成员变量，就可以对每一个对象单独加锁。但是这个方法的缺点是每个对象都因为增加了一个 `lock_` 变量而变大了。让我们来实现对象级别加锁的 smart pointer。

TRAITS 对象

当我们把对象级别加锁的方法运用到 `SmartPtr` 上时，我们遇到了一个问题。让我们再来看一下 `SmartPtr` 的析构函数的定义：

```

template <class T, class RCTraits = RefCountingTraits<T> >
class SmartPtr
{
private:
    T* pointee_;
public:
    ...
~SmartPtr()

```

```

    {
        RCTraits::Unrefer(pointee_);
    }
};

```

正如你所看到的那样，根本没有 RCTraits 的对象。SharedPtr 的析构函数用静态函数的语法调用 RCTraits::Unrefer()。SharedPtr 把 RCTraits 作为只有两个静态函数的包装来使用。现在 traits 类需要保存一些状态，所以我们开始讨论如何保存 traits 对象。显然，保存 traits 对象的地方就在 SharedPtr 对象里，因此我们可以这样修改代码：

```

template <class T, class RCTraits = RefCountingTraits<T> >
class SharedPtr
{
private:
    T* pointee_;
    RCTraits rcTraits_;
public:
    ...
~SharedPtr()
{
    rcTraits_.Unrefer(pointee_);
}
};

```

现在，SharedPtr 拥有了一个 Lock 对象，并使用这个对象完成对象级别的加锁操作，这正是我们想要的。属于不同线程的 SharedPtr 对象不再共享任何数据，因此不会有任何同步的问题。问题解决了。

然而，SharedPtr 变得更大了。“这是显然的，”我听到你说，“我们首先要保证的就是多线程的 SharedPtr 拥有一个 Lock 对象。”但是，不仅仅是多线程 SharedPtr 变大了，单线程的 SharedPtr 也变大了，尽管没有任何附加的数据（回忆一下，RefCountingTraits 没有任何数据成员）。这是为什么呢？因为 C++ 中空对象的大小也不是 0。这条规则在 C++ 语言的很多地方都是合理的。（比如，如果有大小为 0 的对象的话，你怎样才能建立这样的对象的数组？）

不管这条规则是否明智，至少在现在这种情况下，它对我们是不利的。SharedPtr<Something, RefCountingTraits<Something> > 要比一个单纯的指向 T 的指针大，这是不应该的。现在单线程的 SharedPtr 的大小至少是 sizeof(T*)+1，但通常由于字对齐和字节填充的原因，最终 SharedPtr 对象大小可能会在 2*sizeof(T*) 左右。如果你有很多单线程的 SharedPtr，尺寸增加的代价会变得很显著，更不用说通过传值方式传递 SharedPtr 的附加消耗了。

幸运的是，C++ 标准中有另一条关于对象大小的规则，可以帮助我们解决这个问题。这就是空基类优化（empty base optimization）。如果类 D 的基类 B 是空的（没有非静态数据成员），那么 D 对象中的 B 子对象的有效大小可以是 0。这并没有违反前面那条规则，因为 B 子对象被包含于 D 对象

中；当然，如果你抽出一个单独的 B 对象时，它还是有非 0 的大小。你是否可以使用空基类优化取决于你的编译器，因为这条规则的实现是可选，而不是必需的。Metrowerks 的 Code Warrior 5.x 和 Microsoft Visual C++ 6.0 都实现了空基类优化。还有其他地方也需要使用这个优化。^{*}【编者注：请参考《C++空成员优化》一文。】

把空基类优化应用到前面的 SmartPtr 代码中，我们可以让 SmartPtr 继承 RCTraits，而不是用聚合。通过这种方式，如果 RCTraits 是空的，编译器会通过优化去掉多余的空间；如果 RCTraits 不是空的，那么结果和聚合的情况一样。

我们应该用那种继承呢？private，protected，还是 public？不要忘了这只是一个实现上的优化，而不是概念的变化。不管怎么说，SmartPtr 不是一个 RCTraits。因此，最好的选择是私有继承。

```
template <class T, class RCTraits = RefCountingTraits<T> >
class SmartPtr : private RCTraits
{
private:
    T* pointee_;
public:
    ...
~SmartPtr()
{
    RCTraits::Unrefer(pointee_);
}
};
```

这只是利用继承来优化对象大小的一个技巧。有趣的是，我们又回到了用两个冒号的写法，因为现在 RCTraits 是 SmartPtr 的基类。

当 traits 需要保持状态时，就需要用 traits 对象了。Traits 对象可以是其他对象的一部分，也可以作为参数传递。当 traits 对象可能为空时，也许以可以考虑用继承的技巧来优化对象的内存布局，当然你的编译器要支持空基类优化。

定义：traits 对象是 traits 类的一个实例。

Definition: A traits object is an instance of a traits class.

插曲

Traits 模板，traits 类，traits 对象……当我们的讨论从纯粹的静态代码生成方式转变到具有状态的实体时，我们的表达方式也从最静态的方式（模板）发展到具有更多动态特性的方式（完整的 traits

^{*}这几个编译器里所包含的标准 C++ 库中，在容器类的实现时利用了空基类优化。每一个标准的容器都聚合了一个 allocator 对象，缺省的 allocator 通常是一个空类。

对象）。Traits 模板完全是一种编译时的机制；它们在编译结束前就已经消失了。在另一个极端，traits 对象是具有状态和行为的动态实体。

更进一步的动态化是使用多态 traits 和 traits 对象的指针或者引用。但那已经超出 traits 的范畴了。确切地说，多态 traits 是一个策略（Strategy）设计模式。[\[2\]](#)

使用能满足要求的任何一种 traits 机制，并且尽可能选择静态的方案。相对于运行时的解决方案，我们一般更倾向于选择编译时的解决方案。编译时的解决方案意味着：编译器会对代码进行更好的检查，并且往往生成的代码有更好的效率。当然，另一方面，动态【注：dynamism，这里一语双关，也可解释为活力，有生气】为生活带来情趣。

全层次 TRAITS

Traits 往往不是只用于单个类型，而是用于整个类层次。例如，引用计数的方法通常对于整个类层次都是一样的。如果不对于每个类都手工添加 traits，而能够定义一个 traits 可以用于整个类层次，那就好了。但是，traits 技术的基础模板对于继承是一无所知的。这怎么办呢？

也许一个好设计的首要准则是要有灵活性，不要局限于一种策略。解决设计问题就像攻打坚固的城堡：如果一个策略不行，最好就换另外一个。一个坏的策略可能也能解决问题，但是比其他方法代价更高。

根据这个想法，我们重新理一下思路。我们需要找一种在类型层次中保持不变的东西，用它来建立一个类模板。你猜那是什么？嵌套类（在类中定义的类）！除非你重新定义，嵌套类在继承过程中是不变的。嵌套类可以像其他符号一样被继承。这看上去是个值得一试的方法。为了能自动加上嵌入的类型定义，我们先做一个简单的模板：

```
template <class T>
struct HierarchyRoot
{
    // HierarchyId is a nested class
    struct HierarchyId {};
};
```

比如说我们有一个以 Shape 为根的类层次（图 1）。为了表示 Shape 是根，你可以让它继承 Hierarchy<Shape>，如下所示。其他类不变。

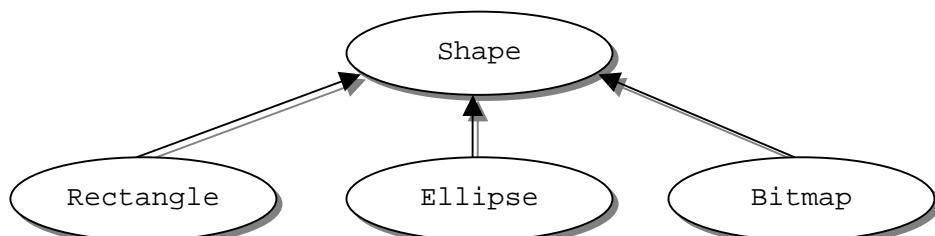


图 1 Shape 层次

```

class Shape : public HierarchyRoot<Shape>
{
    ...
};

class Rectangle : public Shape
{
    ...
};

```

如果你想防止从 Shape 到 HierarchyRoot<Shape>的隐式类型转换（通常也是不希望的），你可以这样定义 Shape：

```

class Shape : private HierarchyRoot<Shape>
{
    ...
public:
    using HierarchyRoot<Shape>::HierarchyId;
};

```

我们得到一个关键的结果：Rectangle::HierarchyId 和 Shape::HierarchyId 是同样的类型。不论你直接或者间接地从 Shape 派生新类，只要你不重新定义符号 HierarchyId，这个符号代表的类型就在整个继承体系中保持不变。

要设计一个使用全层次 traits 的 SmartPtr 和设计使用普通 traits 的 SmartPtr 一样简单。你只要用 T::HierarchyId 代替 T 就行了，象这样：

```

template <class T, class RCTraits =
    RefCountingTraits<typename T::HierarchyId> >
class SmartPtr
{
    ...
};

```

现在，假设在你的应用程序中有两个类层次关系：一个以 Shape 为根，另一个以 Widget 为根。象 Shape 一样，Widget 从 HierarchyRoot<Widget>继承。现在你可以这样为两个类层次特化 RefCountingTraits：

```

template <>
class RefCountingTraits<Shape::HierarchyId>
{
    ...
};

template <>

```

```
class RefCountingTraits<Widget::HierarchyId>
{
    ...
};
```

就是这样，上面的 traits 可以正确的应用于在两个类继承体系中的类，甚至对还没有定义的类也没有问题。下面两节中将指出，全层次 traits 是相当灵活的。

定制全层次 TRAITS

简单的 traits 可以为每个类型提供特化；全层次 traits 为每个类层次提供特化。有时候你可能遇到介于两者之间的情况：你为整个继承体系提供了 traits 模板，同时也要对体系中单独的一个或两个类型进行特化。

你可以这样定义 traits 模板来达到目的：

```
template <class HierarchyId>
class HierarchyTraits
{
    ... most general traits here ...
};

template <class T>
class Traits
: public HierarchyTraits<T::HierarchyId>
{
    // empty body - inherits all symbols from base class
};
```

这个 traits 模板怎样工作呢？客户代码可以这样使用：`Traits<Shape>`，`Traits<Circle>` 等。若想特化整个 Shape 层次的 traits，我们可特化 `HierarchyTraits<Shape::HierarchyId>`。缺省情况下，因为 `Traits<T>` 继承 `HierarchyTraits<T::HierarchyId>`，所有 Shape 的派生类会使用 `HierarchyTraits<T::HierarchyId>` 里定义的 traits。（我敢打赌，如果你跟踪所有这些符号的来源，你会得到很多乐趣。其实这很简单，`HierarchyTraits` 对应于整个层次，`Traits` 对应于每个类型。）

如果你想特化一个特定的类的 traits，比如说 `Ellipse`，你可以直接特化 Traits 模版：

```
template <>
class Traits<Ellipse>
{
    ... specialized stuff for Ellipse ...
};
```

你可以选择是否继承 `HierarchyTraits<Ellipse>`。如果你只想重写一两个符号，你可以选择继承；如果你想完全重写 `Ellipse` 的 traits，你可以选择不继承。这完全取决于你。

对于刚才提到的继承的使用还有一点说明：从动态多形的观点来看，事实上 `Traits<T>` 继承 `HierarchyTraits<T::HierarchyId>` 是不恰当的，因为 `HierarchyTraits` 不是一个具有多态性的基类。我们在这里用继承是因为另一个理由：把继承作为符号传递的工具，目的是让 `Traits<T>` 具有 `HierarchyTraits<T::HierarchyId>` 里定义的所有符号。继承不仅可以用来实现动态的多态性，也可以用来在编译时操作类型。

用这一节里介绍的 Traits-HierarchyTraits 方法，可以对类层次 traits 根据每个类型进行特化。在上面讨论的例子中，`Traits<Rectangle>` 和 `Traits<Circle>` 用的是共同的 `HierarchyTraits<Shape::HierarchyId>`，而 `Traits<Ellipse>` 用的是特化了的 `Traits<Ellipse>`。实现这一切不需要做太多的手脚。

子层次的 TRAITS

假设你要对于 `Shape` 的类层次的一个子层次重新定义 traits，比如图中以 `Bitmap` 为根的子树（图 2）。因此你需要特化 `Bitmap` 和它的直接或间接子类的 traits。

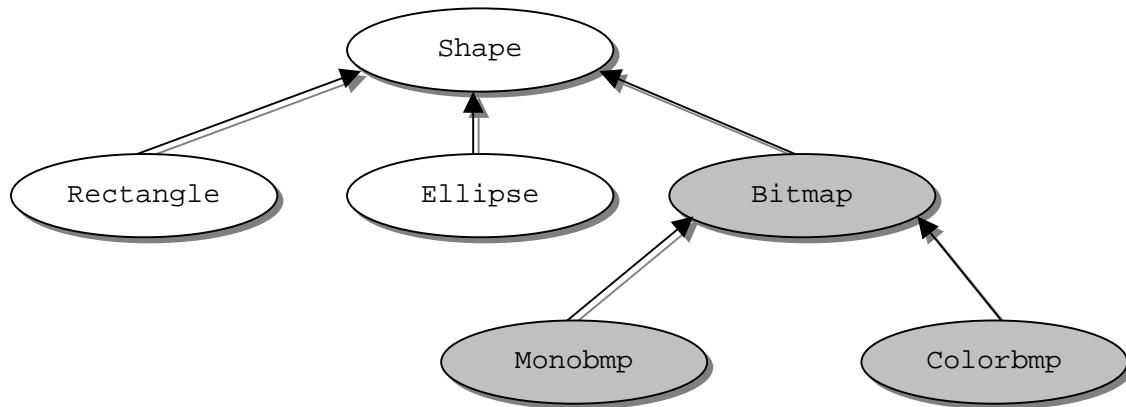


图 2 Shape 层次的 Bitmap 子层次

为了达到这个目的，你可以让 `Bitmap` 同时继承 `Shape` 和 `HierarchyRoot<Bitmap>`。然后你必须通过 `using` 语句来消除符号 `HierarchyId` 的二义性，像这样：

```

class Bitmap : public Shape,
public HierarchyRoot<Bitmap>
{
    ...
public:
    using HierarchyRoot<Bitmap>::HierarchyId;
};
  
```

通过 `using` 语句，`Bitmap` 类会优先使用 `HierarchyRoot<Bitmap>::HierarchyId`，而不是 `Shape::HierarchyId`。这样你可以用 `Bitmap::HierarchyId` 来特化 traits，并且 `Bitmap` 的子类也将用这个特化，除非你为某个子层次又定义了不同的 traits。

注意事项

全层次 traits 的最大的缺点是需要修改类层次中的基类，而你有时候不能做到这点。还好，你会得到一个编译错误("Class Widget does not define a type HierarchyId")，而不是运行时的错误。

你可以对于无意义类型（如 `void`）特化全层次 traits 模板，这样可以在一定程度上解决这个问题。对于你不能修改的类层次，你可以用 `HierarchyTraits<void>`。虽然不是很灵活，但在开发受阻时，也不失为一个可行的方法。

还有其它不用介入类层次来实现全层次 traits 的方法，但那些方法往往更脆弱，并且暴露出各种错误。我始终欢迎读者提出各种建议。

结论

当 trait 必须保持某些状态的时候，就需要用到 traits 对象。如果 trait 类的状态是可选的（有些 traits 有状态，有些没有），那么最好是通过继承的技巧，利用空基类优化（如果可能的话）。只需要一点点手脚，你就可以定义全层次 traits。通过这种方式，你只需为一个类层次写一次 traits。全层次 traits 可以提供很大的灵活性，你可以对类层次中一个特定的类定义特别的 traits，也可以对一个子层次定义 traits。

全层次 traits 使用继承的方法比较怪异。继承不仅仅是实现运行时多态性的工具，也是编译时操纵类型的工具。C++ 把继承的两种特性混合在一起，有时候会引起误解。

然而，最好的消息是，全层次 traits 只用了模板的基本功能。就是说，即使你现在使用的编译器不那么符合标准，你还是可以用这个技术。

感谢

非常感谢 Herb Sutter，他花时间审阅了这篇文章并提出深刻的见解。

参考

1. Alexandrescu, A. "Traits: The else-if-then of Types", *C++ Report*, 12(4): 22–25, 31, Apr. 2000.
2. Gamma, E., et al. *Design Patterns*, Addison-Wesley, Reading, MA, p. 315, 1995.

本文包含 *Design with C++* (暂题，Andrei Alexandrescu, © 2001 Addison Wesley Longman (in press)) 一书中的文字和示例。【已经出版了，书名叫 *Modern C++ Design: Generic Programming and Design Patterns Applied*】

C++的不足之处讨论系列：保证类型安全的连接属性

Ian Joyner
cber 译

以下文章翻译自 Ian Joyner 所著的 *C++?? A Critique of C++ and Programming and Language Trends of the 1990s* 3/E 【Ian Joyner 1996】。

原著版权属于 Ian Joyner，征得 Ian Joyner 本人的同意，我得以将该文翻译成中文。因此，本文的中文版权应该属于我;-)

该文章的英文及中文版本都用于非商业用途，你可以随意地复制和转贴它。不过最好请在转贴时加上前面的这段声明。

如果有人或机构想要出版该文，请最好联系原著版权所有人及我。

另外，该篇文章已经包含在 Ian Joyner 所写的 *Objects Unencapsulated* 一书中（目前已经有了日文的翻译版本），该书的介绍可参见于：

http://www.prenhall.com/allbooks/ptra_0130142697.html

<http://efsa.sourceforge.net/cgi-bin/view/Main/ObjectsUnencapsulated>

<http://www.accu.org/bookreviews/public/reviews/o/o002284.htm>

Ian Joyner 的联系方式：i.joyner@acm.org
我的联系方式：cber@email.com.cn

译者前言：

要想彻底的掌握一种语言，不但需要知道它的长处有哪些，而且需要知道它的不足之处又有哪些。这样我们才能用好这门语言，避免踏入语言中的一些陷阱，更好地利用这门语言来为我们的工作所服务。

Ian Joyner 的这篇文章以及他所著的 *Objects Unencapsulated* 一书中，向我们充分地展示了 C++ 的一些不足之处，我们应该充分借鉴于他已经完成的伟大工作，更好地了解 C++，从而写出更加安全的 C++ 代码来。

C++ARM 中解释说 type-safe linkage 并不能 100% 的保证类型安全。既然它不那 100% 的保证类型安全，那么它就肯定是不安全的。统计分析显示：即便在很苛刻的情况下，C++ 出现单独的 O-ring 错误的可能性也只有 0.3%。但我们一旦将 6 种这样的可能导致出错的情况联合起来放在一起，出错的几率就变得大为可观了。在软件中，我们经常能够看到一些错误的起因就是其怪异的联合。OO 的一个主要目的就是要减少这种奇怪的联合出现。

大多数问题的起因都是一些难以察觉的错误，而不是那些简单明了的错误导致问题的产生。而且在通常的情况下，不到真正的临界时期，这样的错误一般都很难被检测到，但我们不能由此就低估了这种情况的严肃性。有许多的计划都依赖于其操作的正确性，如太空计划、财政结算等。在这些计划中采用不安全的解决方案是一种不负责任的做法，我们应该严厉禁止类似情况的出现。

C++在 type-safe linkage 上相对于 C 来说有了巨大的进步。在 C 中，链接器可以将一个带有参数的诸如 `f(p1, ...)` 这样的函数链接到任意的函数 `f()` 上面，而这个 `f()` 甚至可以没有参数或是带有不同的参数都行。这将会导致程序在运行时出错。由于 C++ 的 type-safe linkage 机制是一种在链接器上实做的技巧，对于这样的不一致性，C++ 将统统拒绝。

C++ARM 将这样的情况概括如下 “ 处理所有的不一致性-> 这将使得 C++ 得以 100% 的保证类型安全 -> 这将要求对链接器的支持或是机制（环境）能够允许编译器访问在其他编译单元里面的信息 ”。

那么为什么市面上的 C++ 编译器（至少 AT&T 的是如此）不提供访问其他编译单元中的信息的能力呢？为什么到现在也没有一种特殊的专门为 C++ 设计的链接器出现，可以 100% 的保证类型安全呢？答案是 C++ 缺乏一种全局分析的能力（在上一节中我们讨论过）。另外，在已有的程序组件外构造我们的系统已经是一种通用的 Unix 软件开发方式，这实现了一定的重用，然而它并不能为面向对象方式的重用提供真正的弹性及一致性。

在将来，Unix 可能会被面向对象的操作系统给替代，这样的操作系统足够的“开放”并且能够被合适地裁剪用以符合我们的需求。通过使用管道（pipe）及标志（flag），Unix 下的软件组件可以被重复利用以提供所需的近似功能。这种方法在一定的情况下行之有效，并且颇负效率（如小型的内部应用，或是用以进行快速原型研究），但对于大规模、昂贵的、或是对于安全性要求很高的应用来说，采取这样的开发方法就不再适合了。在过去的十年中，集成的软件（即不采用外部组件开发的软件）的优点已经得到了认同。传统的 Unix 系统不能提供这样的优点。相比而言，集成的系统更加的复杂，对于开发它们的开发人员有着更多的要求，但是最终用户（end user）要求的就是这样的软件。将所有的东西拙劣的放置于一起构成的系统是不可接受的。现在，软件开发的重心已经转到组件式软件开发上面来了，如公共领域的 OpenDoc 或是 Microsoft 的 OLE。

对于链接来说，更进一步的问题出现在：不同的编译单元和链接系统可能会使用不同的名字编码方式。这个问题和 type-safe linkage 有关，不过我们将会在“重用性及兼容性”这节讲述之。

Java 使用了一种不同的动态链接机制，这种机制被设计的很好，没有使用到 Unix 的链接器。Eiffel 则不依赖于 Unix 或是其他平台上的链接器来检测这些问题，一切都由编译器完成。

Eiffel 定义了一种系统层上的有效性（system-level validity）。一个 Eiffel 编译器也就因此需要进行封闭环境下的分析，而不是依赖于链接器上的技巧。你也可以就此认为 Eiffel 程序能够保证 100% 的类型安全。对于 Eiffel 来说有一个缺点就是，编译器需要干的事情太多了。（通常我们会说的是它太“慢”了，但这不够精确）目前我们可以通过对于 Eiffel 提供一定的扩展来解决这个问题，如融冰技术（melting-ice technology），它可以使我们对于系统的改动和测试可以在不需要每次都进行重新编译的情况下进行。

现在让我们来概括一下前两个小节 - 有两个原因使我们需要进行全局（或封闭环境下的）分析：一致性检测及优化。这样做可以减掉程序员身上大量的负担，而缺乏它是 C++ 中的一个很大的不足。

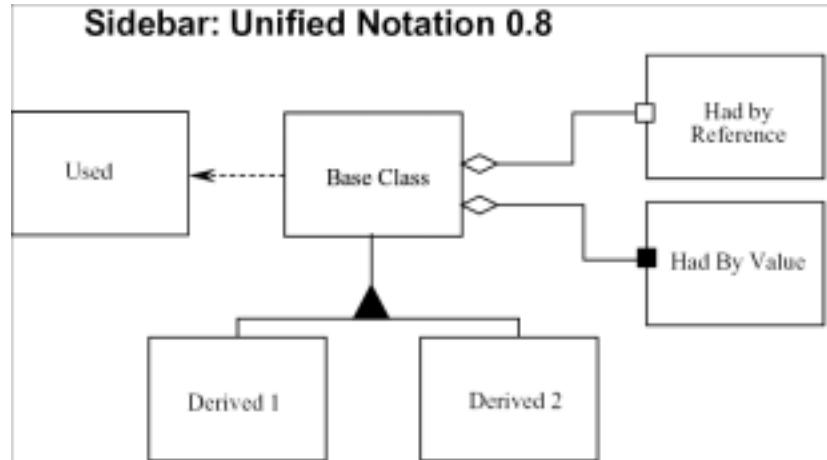
Liskov 替换原则 LSP

Robert C.Martin

虫虫&pliuly 译

译者的话：滥用继承是一个相当普遍的现象。我们认为会用继承，会用多态，便以为自己很OO，如果再会用Java或者C#，那就更OO了。其实，我们往往是在滥用继承，在亵渎OO的精髓。有朋友知道ISA关系，但是对于ISA真正的含义，依然是一无所知。其实早在1987年的OOPSLA大会上，麻省理工学院（MIT）计算机科学实验室的Liskov女士就发表了经典文章*Data Abstraction and Hierarchy*，其中提出了以她名字命名的Liskov替换原则（The Liskov Substitution Principle），简称LSP。该原则说明了什么时候该使用继承，什么时候不该使用以及为什么。一年后此文发表在ACM的SIGPLAN Notices杂志上，更是影响深远。涉及该领域的文章，引用[Liskov87]或[Liskov88]的不计其数，此文的经典价值和不朽地位可见一斑。原文的理论性非常强，不易理解。本文是著名技术作家Robert Martin先生在1996年为C++ Report所写的文章，所有例子均采用C++书写，通俗易懂，处处透出真知灼见。本文翻译经 Robert Martin 先生同意，不胜感激。Robert Martin 先生的英文原文可参考<http://www.objectmentor.com/publications/lsp.pdf>。

这是我为设计笔记（Engineering Notebook）专栏所写的第二篇文章。在下面，我将采用Booch和Rumbaugh新的统一表示法UN 0.8作为设计图的图示法，图例如下所示。【译者注：原文中间还有一段对此专栏的介绍，同上期相同，此处略去。不知道大家是否注意到，上期的文章采用的还是Booch表示法，这期就变成一个所谓“统一表示法”。而这些文章写于1996年，正是软件工程领域发生深刻变革的时候。以后大家还会看到，图示法又变成UML。在这里，译者简单介绍一下相关的背景。面向对象建模语言出现于70年代，90年代初期出现发展



的黄金时代，出现了百家争鸣的局面，语言数多达50多种，虽然空前繁荣，却也不利于人们互相的交流。这几十种中，最出名的就是Booch提出的Booch93，Rumbaugh提出的OMT以及Jacobson提出的OOSE这三种。从94年开始，Booch和Rumbaugh开始合作，一年以后，他们统一了Booch93和OMT-2，提出了一种新的方法，也就是本文所采用的UN 0.8，实现了统一化。同时，Jacobson也加入进来。他们于96年发表了UML 0.9和0.91，并在97年正式发表UML 1.0和1.1，实现了标准化。UML 1.1被OMG正式采纳为面向对象标准建模语言，实现了工业化，为业界所广泛接受。更详细的情况，可以参考北京航空航天大学的张莉和周伯生两位教授所写的《标准建模语言UML的概念》一文。】

介绍

上一期我们讨论了开放 - 封闭 (Open-Closed) 原则OCP。此原则是写出可维护且可重用代码的基础，讲述了一段设计良好的代码可以在不作修改的情况下进行扩充。同样，在一个设计良好的程序中，添加新功能只需添加新代码即可，不用修改已经正常运行的部分。

OCP原则背后的主要机制是抽象 (abstraction) 和多态 (polymorphism)。在静态类型语言中，比如C++，支持抽象和多态的关键机制是继承 (inheritance)。正是使用了继承，我们才可以构建出派生类并使之遵循抽象基类中纯虚函数所定义的抽象多形接口的。

是什么设计规则在支配着这种特殊的继承用法呢？最佳的继承层次的特征又是什么呢？怎样的情况又让我们容易掉进不符合OCP原则的陷阱中去呢？这些就是本文要解答的问题。

Liskov替换原则

使用指向基类的指针或引用的函数，必须能够在不知道具体派生类对象类型的情况下使用它们。

FUNCTIONS THAT USE POINTERS OR REFERENCES TO BASE CLASSES MUST BE ABLE TO USE OBJECTS OF DERIVED CLASSES WITHOUT KNOWING IT.

这就是Liskov替换原则LSP (The Liskov Substitution Principle) 的解释。大概在8年前【译者注：对今天而言，已经是十几年前了】，Barbara Liskov首先写到¹：

这里需要如下的替换性质：若对于每一个类型S的对象 o_1 ，都存在一个类型T的对象 o_2 ，使得在所有针对T编写的程序P中，用 o_1 替换 o_2 后，程序P的行为功能不变，则S是T的子类型。

What is wanted here is something like the following substitution property: If for each object o_1 of type S there is an object o_2 of type T such that for all programs P defined in terms of T, the behavior of P is unchanged when o_1 is substituted for o_2 then S is a subtype of T.

【译者注：这段话就是LSP的精髓，不要怕拗口，请仔细反复体会。Barbara Liskov的原文是OO的经典之作，有兴趣的朋友不妨找来读一读。原文中这段话的前面还有一部分：A type hierarchy is composed of subtypes and supertypes. The intuitive idea of a subtype is one whose objects provide all the behavior of objects of another type (*the supertype*) plus something extra.意思是说，类型层次由子类型和超类型（记得Java里“超类”就是“父类”吧？）组成，直觉告诉我们，子类型的含义就是该类型的对象提供了另外一个类型（超类型）的对象的所有行为功能，并有所扩充。】

想想违背该原则的后果，LSP原则的重要性就不言而喻了。如果某个函数使用了指向基类的指针或引用，却违背LSP原则，那么这个函数必须了解该基类的所有派生类。这个函数显然违背开放 - 封闭原则OCP，因为一旦新构建该基类的子类，此函数就需要修改。

一个违背LSP原则的简单例子

在一个违背LSP原则的事例中，最容易见到的就是根据对象的类型，运用C++的运行时类型辨别RTTI来选择函数。比如

¹ Barbara Liskov, "Data Abstraction and Hierarchy," SIGPLAN Notices, 23,5 (May, 1988).

```

void DrawShape(const Shape& s)
{
    if (typeid(s) == typeid(Square))
        DrawSquare(static_cast<Square&>(s));
    else if (typeid(s) == typeid(Circle))
        DrawCircle(static_cast<Circle&>(s));
}

```

很显然，DrawShape函数的设计非常不合理，它必须知道Shape类所有的派生类，而每构建一个从Shape类派生的新类它都必须作更改，甚至很多人认为这种函数结构简直是OOD的诅咒。

正方形和矩形，更微妙的违规

有一个违反了LSP原则但更为微妙的事例。考虑某应用程序使用了如下形式的Rectangle类：

```

class Rectangle
{
public:
    void SetWidth(double w) {itsWidth=w;}
    void SetHeight(double h) {itsHeight=h;}
    double GetHeight() const {return itsHeight;}
    double GetWidth() const {return itsWidth;}
private:
    double itsWidth;
    double itsHeight;
};

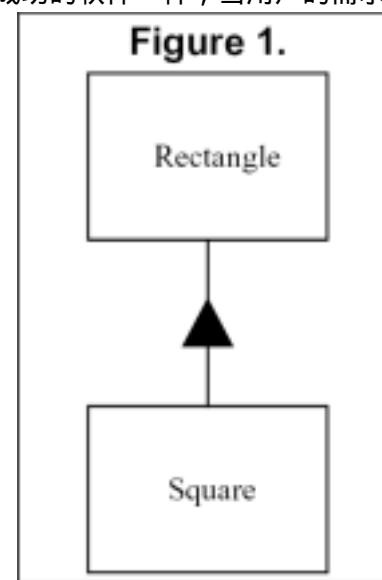
```

假设这个应用程序运行得很好，安装在很多地方。和任何一个成功的软件一样，当用户的需求发生变化时，就需要添加新的功能。假设某一天，用户不满足于仅仅操作矩形，要求添加操作正方形的功能。

在C++中我们经常说继承是ISA关系，也就是说，如果某种新类型的对象被认为与已有类的对象之间是ISA的关系，那么这个新类应从这个已有的类派生。

显然，从一般的意义上讲，一个正方形就是一个（ISA）矩形。既然存在ISA关系，那么从Rectangle类派生建立Square类的模型，便是合乎逻辑的。（见Figure 1）

许多人以为，ISA的这种用法是面向对象分析OOA（Object Oriented Analysis）的基本技术之一。一个正方形也是一个矩形，于是Square类就应派生自Rectangle类。不过，这种想法会带来一



些微妙但极为值得重视的问题。一般如果没有实际地在程序中使用这些代码的话，问题是不容易发现的。

首先引起我们注意的是Square并不同时需要成员变量itsHeight和itsWidth，但还是会继承它们。显然这是浪费，更进一步，如果我们创建成百成千个Square对象（比如在CAD/CAE中复杂的电路的每个元件的引线或管脚都是一个正方形），浪费的程度就惊人了。

然而，就算我们不关心内存效率，还有没有其他问题呢？当然有！Square会继承SetWidth和SetHeight函数，而这两者对正方形来说完全不适合，因为正方形的长和宽是相等的。这是找出设计问题的一个重要线索。不过这个问题是可以避免的，我们可以按下面的方式改写SetWidth和SetHeight：

```
void Square::SetWidth(double w)
{
    Rectangle::SetWidth(w);
    Rectangle::SetHeight(w);
}
void Square::SetHeight(double h)
{
    Rectangle::SetHeight(h);
    Rectangle::SetWidth(h);
}
```

现在只要长和宽中其中一个发生变化，另一个也会相应地改变。这样，Square仍然保持不变，仍是严格数学意义上的正方形。

```
Square s;
s.setWidth(1); // Fortunately sets the height to 1 too.
s.setHeight(2); // sets width and height to 2, good thing.
```

不过考虑下面这个函数：

```
void f(Rectangle& r)
{
    r.setWidth(32); // calls Rectangle::SetWidth
}
```

如果我们向这个函数传递一个指向某Square对象的引用，这个Square对象就被搞乱了，因为它的长并不会改变。这显然违反了LSP原则。以Rectangle的派生类作参数传入时，函数f不能正确运行。错误的原因是在Rectangle中没有把setWidth和setHeight声明为虚函数。

这个错漏还是很容易修补的。但是新派生类的创建还是使我们改变了基类，这就意味着设计是有问题的，的确违背了OCP原则。也许有人会反驳说，忘记把setWidth和setHeight作为虚函数

才是真正的缺陷。然而，这很难让人信服，毕竟设定一个矩形的长和宽是非常原始的操作，如果不是预见到Square的存在，我们凭什么非要让进行这两项操作的函数是虚函数呢？

不管怎么说，假设我们先接受这个理由并以如下修改后的代码结束这段争论：

```
class Rectangle
{
public:
    virtual void SetWidth(double w) { itsWidth=w; }
    virtual void SetHeight(double h) { itsHeight=h; }
    double GetHeight() const { return itsHeight; }
    double GetWidth() const { return itsWidth; }
private:
    double itsHeight;
    double itsWidth;
};

class Square : public Rectangle
{
public:
    virtual void SetWidth(double w);
    virtual void SetHeight(double h);
};

void Square::SetWidth(double w)
{
    Rectangle::SetWidth(w);
    Rectangle::SetHeight(w);
}

void Square::SetHeight(double h)
{
    Rectangle::SetHeight(h);
    Rectangle::SetWidth(h);
}
```

真正的问题

现在我们有Square和Rectangle这两个类，看起来也都能工作。不管对Square对象进行什么操作，它都与数学意义的正方形保持一致。而且，我们还可以向接受指向Rectangle的指针或引用的函数传递Square，而Square依然保持正方形的特性，也与该函数相容。

看来这个Square模型似乎是自相容的，是正确的。可惜这个结论是错的。一个自相容的模型不

见得就一定与所有的用户程序相容。考虑下面的函数g：

```
void g(Rectangle& r)
{
    r.setWidth(5);
    r.setHeight(4);
    assert(r.getWidth() * r.getHeight() == 20);
}
```

这个函数认为所传递进来的一定是Rectangle，调用了其成员函数GetWidth和SetHeight。对Rectangle而言，此函数运行正确，但是如果传递Square对象就会发生“断言错误【译者注：assertion error，函数assert引起的错误】”。这才是真正的问题所在：写这个函数的程序员假设改变Rectangle的宽度不会改变其长度，这是否正确？

显然写函数g的程序员所作的假设非常合理。但如果传入Square对象，这个假设就有问题了。因此，存在使用指向Rectangle对象的指针或引用的函数，不能正确操作Square对象。这些函数把违背LSP原则的事实暴露无遗。派生自Rectangle的Square也破坏了这些函数的不可修改的特性（closed for modification），因而也违背了OCP原则。

有效性不是内部性质

这让我们作出一个重要的结论：一个模型，如果隔离来看，并不能真正意义上判定其有效性。模型的有效性只能通过客户程序表现出来。比如，如果隔离来看，最后那一版本的Rectangle和Square是自相容且有效的，但如果从只是对基类作出了一些合理假设的程序员的角度来看，这个模型显然有问题。

因此，考虑某一特定设计方案是否恰当的时候，我们不能仅仅简单地孤立地来看这个方案，而应该从使用该设计方案的人可能作出的合理假设的观点来审视。

错在哪（What Went Wrong? W³）

那么到底怎么搞的？Square和Rectangle这个显然很合理的模型为什么有问题？并且，Square不是一个Rectangle吗？这里不存在ISA关系吗？

的确不存在！一个正方形倒是一个矩形，但是一个Square对象并非一个Rectangle对象。为什么？因为Square对象的行为功能（behavior）与Rectangle对象的行为功能（behavior）并不相容。从行为功能的角度来看（behaviorally），一个Square就不是一个Rectangle！行为功能（behavior），才是软件所关注的问题。

LSP原则清楚地指出，OOD中ISA关系是就行为功能而言。行为功能（behavior）不是内在的、私有的，而是外在、公开的，是客户程序所依赖的。例如，上面函数g的设计者依赖于矩形的长和宽独立变化【译者注：也是说，一个改变并不影响另一个】这个事实。这两个变量的独立性就是其他程序员可能依赖的外在的、公开的行为功能。

按照LSP和OCP原则，所有派生类的行为功能必须和客户程序对其基类所期望的保持一致。

DBC (Design by Contract)

LSP与Bertrand Meyer所说的DBC (译者注：Design by Contract，也就是所谓“契约式设计”、“合同式设计”或“约定式设计”²) 有很紧密的联系。按这种方式，类的方法声明为先决条件(precondition) 和后续条件 (postcondition)。为了让方法得以执行，先决条件必须为真。完成后，方法保证后续条件为真。

Rectangle::SetWidth(double w)的先决条件可看作是

```
assert((itsWidth == w) && (itsHeight == old.itsHeight));
```

按Meyer所述³，派生类的先决条件和后续条件规则是：

当重新定义派生类中的例行程序时，我们只能用更弱的先决条件和更强的后续条件替换之。

...when redefining a routine [in a derivative], you may only replace its precondition by a weaker one, and its postcondition by a stronger one.

也就是说，当通过其基类接口使用某对象时，用户只知道基类的先决条件和后续条件。因此，派生对象不能奢望用户遵从比基类更强的先决条件。同时，派生类必须与基类所有的后续条件一致。也就是说，它们的行为功能和输出不能和基类任何已经确立的限制相抵触。基类的用户不应被派生类的输出搞糊涂。

显然Square::SetWidth(double w)后续条件比Rectangle::SetWidth(double w)弱，因为它不符合基类中“(itsHeight == old.itsHeight)”这一条，因而Square::SetWidth(double w)违背了基类所订下的“合同”或“契约”。

某些语言，比如Eiffel，直接支持先决条件和后续条件，可以让我们直接声明，并提供运行时间系统校验。C++没有此项特性，但即使在C++中我们也可以自己考虑每个方法的先决条件和后续条件，保证没有违背Meyer所说的规则。如果为每个方法注明先决条件和后续条件，也是很有建设性的事。

【译者注：DBC是个很复杂的话题。Eiffel之父Bertrand Meyer在其经典名著*Object Oriented Software Construction*第2版第11章中详细论述了这个问题，有兴趣的朋友可以参考一下。什么是DBC呢？Meyer的定义是：把类和其客户之间的关系看作是一个正式的协议，明确各方的权利和义务。(viewing the relationship between a class and its clients as a formal agreement, expressing each party's rights and obligations.) 在关于条件的强 (strong) 与弱 (weak) 的论述中，Meyer举了一个很有意思的例子，转述如下：您的一位朋友是个大懒虫，要找份工作，但老想做简单容易的事。他在广告上看到了一些薪水和补助都差不多的工作，来向您讨意见：应该选择先决条件弱还是强的工作呢？对于后续条

² *Object Oriented Software Construction*, Bertrand Meyer, Prentice Hall, 1988

³ 同上，p256

件，也有同样的问题。先决条件是指开始工作的条件。先决条件越强，需要处理的事情就越有限，当然对您朋友来说就越好。如果工作的先决条件是“太阳从西边出来”这种始终为假的条件，那么您朋友太爽了，根本不用做事。对于后续条件来说，正好倒了过来。后续条件越弱，您朋友越拣便宜。如果后续条件始终为真，那是再好不过的事情了。】

一个实际的例子

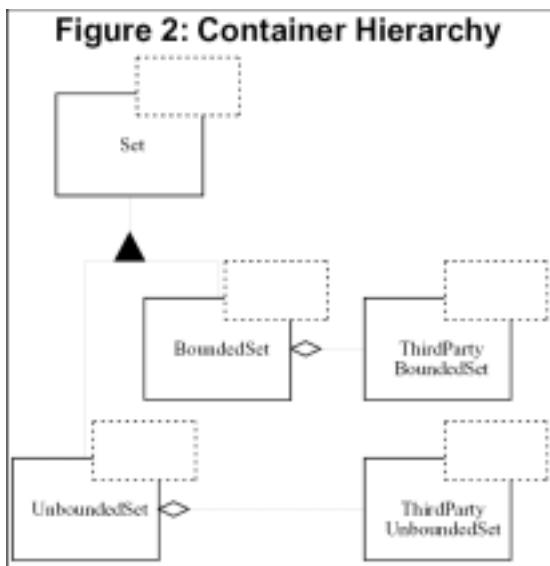
说够了这些正方形和矩形了！LSP在实际的软件开发中是否能发挥作用呢？我们来看看一个用例研究，它来自我以前做的一个项目。

动机

我不喜欢第三方提供的一些容器类的接口，因为我不希望自己的程序严重依赖于这些容器类，也许以后我会换更好的。因此我把它们包装在我自己的抽象接口下。（见Figure 2）

我写了个类 Set，提供了 Add、Delete 和 IsMember 这几个纯虚函数。

```
template <class T>
class Set
{
public:
    virtual void Add(const T&) = 0;
    virtual void Delete(const T&) = 0;
    virtual bool IsMember(const T&) const = 0;
};
```



这个结构统一了第三方提供的有限集和无限集这两个类，让我们可以通过一个公共的接口访问它们。这样，使用 Set<T>& 作参数的客户程序就不必关心这个 Set 是有限集还是无限集了。如

```
template <class T>
void PrintSet(const Set<T>& s)
{
    for (Iterator<T>i(s); i; i++)
        cout << (*i) << endl;
}
```

不用关心 Set 的具体类型，这是一个大大的优点，意味着程序员可以在具体的事例中再选择需要哪种 Set，而客户函数均不会受影响。在内存紧张而时间要求不严格的情况下，程序员可以选择 BoundedSet，反之则可以选择 UnboundedSet，客户程序则通过基类 Set 的公共接口操作这些对

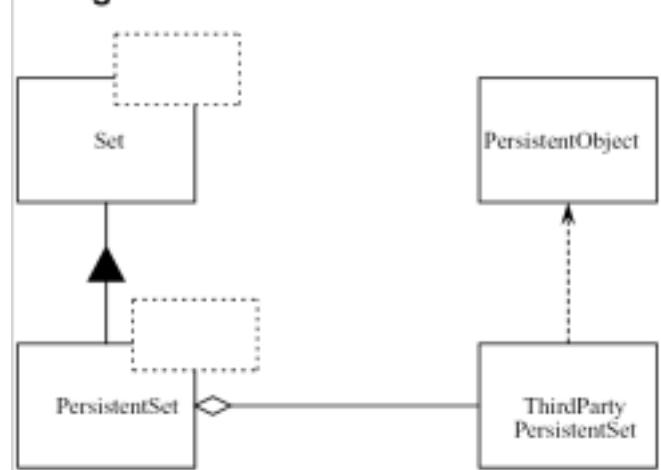
象，而不必关心到底使用的是何种类型。

问题

我想在该层次中加入PersistentSet。所谓持久性（persistent）集合是指可以把信息写入流，稍后可以由该程序或另外的程序读入刚才的信息。遗憾的是，我能够访问的具有持久化功能的第三方容器类不是一个模板类，它只接受虚基类PersistentObject的派生对象作为其元素，而只接受虚基类PersistentObject的派生对象（见Figure 3）。

表面看来没什么，其实隐藏着一个很别扭的设计问题。当客户程序向基类Set加入元素时，如果这个Set正好是PersistentSet，如何保证客户只加入PersistentObject的派生对象？

Figure 3. Persistent Set



考虑下面PersistentSet::Add的代码：

```

template <class T>
void PersistentSet::Add(const T& t)
{
    PersistentObject& p =
        dynamic_cast<PersistentObject&>(t); // throw bad_cast
    itsThirdPartyPersistentSet.Add(p);
}
  
```

显然如果任何客户程序企图向这个PersistentSet添加不是从PersistentObject派生的对象，将会发生运行时间错误。然而，没有一个使用抽象基类的客户程序会预计到调用Add时会抛出异常。由于像Add这样的函数对于Set的派生类表现出与基类不一致的行为功能，所以，这样的类层次的变化违背了LSP原则。

这是个问题吗？当然是。以前传递Set的派生对象给这些函数根本没有问题，现在传递PersistentSet却会引发运行时间错误。调试这种问题很困难，因为这个运行时刻错误发生之处距离产生这个错误的逻辑有误的代码很远。逻辑错误可能使因为把PersistentSet传递给一个有错的函数，也可能向PersistentSet加入的对象不是派生自PersistentObject。不管哪种，具体发生逻辑错误的地方可能离调用Add方法的地方还有十万八千里呢。找到问题很难，解决更难。

不符合LSP的解决方案

怎么办呢？若干年前，我是按定义约定的方式而没有在源代码中解决。我约定不让

PersistentObject和PersistentSet暴露给整个程序，而只让某个模块清楚。该模块负责读写所有容器。当要写某个容器时，把容器的所有内容复制为PersistentObject，再加入PersistentSet，然后写入流中。读入时，先把信息从流读到PersistentSet中，再把PersistentObject从PersistentSet中删掉并复制为普通的对象并加入一般的Set中。

这种解决方法限制性很强，但也是我当时想到的唯一方法，可以不让PersistentSet对象出现在要加入非持久性对象的函数中。此外这也解除了程序其余部分对整个持久化概念的相关性。

这个解决方案是否奏效呢？没有。有些没有理解到这个约定重要性的工程师，在程序的多处地方违背了这个约定。这就是使用约定方式的问题，我不得不不断地跟每位工程师解释。如果某位不同意，那么这个约定就被违背，由此也宣告了整个结构的失败。

符合LSP的解决方案

那又怎么办？其实可确认Persistent跟Set之间并不存在ISA关系，它不应派生自Set。因此我们将分离但不会完全分离这个层次。Set和Persistent有不少相同的特性，事实上，仅仅是Add方法在LSP原则下出了问题。因此我们可以创建如Figure 4的一个层次，使Set和PersistentSet成为兄弟关系，统一在一个包含测试成员关系以及迭代等操作的抽象接口下，且不会向PersistentSet加入不是派生自PersistentObject的对象。

```
template <class T>
void PersistentSet::Add(const T& t)
{
    itsThirdPartyPersistentSet.Add(t);
    // This will generate a compiler error if t is
    // not derived from PersistentObject.
}
```

如上所示，任何试图向PersistentSet加入非派生自PersistentObject的对象都将导致编译错误（第三方的持久性集合的接口需要的是PersistentObject&）。

结论

OCP原则是OOD中很多说法的核心，它可以让应用程序更易维护、更易重用及更健壮。LSP原则（也即DBC）是符合OCP原则应用程序的一项重要特性。仅当派生类能完全替换基类时，我们才能放心地重用那些使用基类的函数和修改派生类型。

Pattern Hatching

从 GoF 谈起

作者 : John Vlissides

翻译 : HolyFire

1995.3~1995.4 发表于 C++ Report

首先，值得一提的是，在“ The Column Without a Name ”专栏中，Jim Coplien 对各式各样模式的讨论打下了基础。在这个专栏里我将会提出我对这门新兴学科的另一方面的看法，它反映了我还在 GoF (注：Gang of Four，鼎鼎大名的《设计模式》一书的四位作者) 时的经验。呵呵，这里提及的可不是什么不良团伙，它指的是 Erich Gamma, Richard Helm, Ralph Johnson 和我自己。当时，我们合作写下了 *Design Patterns: Elements of Reusable Object-Oriented Software* (注：中文版《设计模式：可复用面向对象软件的基础》，机械工业出版社出版) 一书，介绍了从众多的面向对象软件和系统设计中精炼出来的 23 个模式。

在《设计模式》中，我们尝试着阐述面向对象设计中可复用的部分，它们赋予了一个好的软件中那些难以掌握的特性：优雅、弹性、延展性和可复用程度。我们从中提取了这部分，虽然形式上和 Alexander 的不太一样，但是它们还是信守了模式的理念。我们模式的形式会在后面详细介绍。

书中的模式来源于许多应用领域：包括用户界面、编译器、编程环境、操作系统、分布系统、金融模型以及计算机辅助设计。我们谨慎地只包含那些在多个领域中一次次得到证明的设计模式。

将我们的模式称为“设计模式”主要是由于两个原因。我们的工作起源于 Erich Gamma 的博士论文，文中他发明了这个术语。他想强调他是在追求设计技术而不是其他领域，诸如分析或实现等软件开发技巧。另一个缘故是因为单独对于“模式 (pattern)”这个词，不同的人有着不同的理解，特别是一些模式迷。前面加上“设计”提供了必要的限定。不过因为我在本栏中主要谈设计模式，很多的地方我可以省掉这个前缀。

我最初取了“ Pattern Hatching ”作这个专栏的标题，因为它在计算机学科中很常见（另外，其他的标题都被用了）。不过现在回过头来，我感到它很好地体现了本栏目的意图。“ Hatching ”意味着我们并不是在创造什么东西，而是意味着从现存的基础发展。多么恰当的比喻：《设计模式》是我们的孵化器，因为许多新生命很有希望从这里脱颖而出。（我相信将很少有机会进一步使用这一比喻）

“ Pattern Hatching ”不仅仅是那本书的翻版。我的目标是在这本书的基础上，利用其中的概念，从中学习并改进他们。

设计模式 vs. Alexander 模式

设计模式与 Alexander 模式的结构完全不同。首先，Alexander 模式从问题的描述开始，接着是用一个例子来阐明这个问题，并解说和分析其中关键的方方面面，最后是解决方案的简洁描述。除了一些打印修饰以外，这种风格就像平常的讲说，适合于从头到尾通读。不足处是结构相当粗糙，再细微一层可说是没有结构，只是在讲故事。如果你需要某个模式的某个重点的详细信息，你必须查阅很多篇幅。

相比之下，设计模式很有结构性。必须这样做，因为设计模式（平均 10 页一个）包含了比 Alexander 模式（平均 4 页一个）更多的内容。设计模式也详细描述了如何去实现一个模式，包括例程和其实现的讨论。Alexander 很少用分类的方式来处理文档的细节，我们也可以用类似 Alexander 的结构，但我们要重要的是能快速的查到设计模式以及实现。因为我们并没有列出如何应用模式的步骤（用 Alexander 的话说，一个真正的模式语言应给出它），帮助你找到适合的模式的指导就会比较少。即使你知道你想要那个模式，你也很难在它的篇幅里找到你感兴趣的部分。我们必须把它做得能让设计者方便的找到适合他们问题的模式。这促使我们用了更有条理的方式

设计模式结构

每个设计模式包括以下 13 个部分：

1. 名称 (Name)
2. 意图 (Intent)
3. 别名 (Also Known As)
4. 动机 (Motivation)
5. 适用性 (Applicability)
6. 结构 (Structure)
7. 参与者 (Participants)
8. 协作 (Collaborations)
9. 效果 (Consequences)
10. 实现 (Implementation)
11. 代码示例 (Sample Code)
12. 已知应用 (Known Uses)
13. 相关模式 (Related Patterns)

前 3 个部分标识了一个模式。第 4 部分提示了 Alexander 式模式的内容：描述问题的具体例子及相应的背景和解决方案。第 5 到第 9 部分给出了一个模式的抽象定义。大多数人容易接受先用具体例子说明再抽象说明的东西。所以每个模式先用一个问题和解决方案的来引出，再上升到抽象的形式。第 10 部分又回到具体的了，而第 11 部分是所有部分中最具体的。第 12 部分起了目录的作用，第 13 部分提供了相关连接。

从组合模式开始

以组合 (Composite) 模式来说。它的体现了两方面：设计一个对象 s 以树结构的方式来处理

部分-整体的结构，提供给用户一个统一的接口来处理这些对象 s 而且不用管这些对象是内部节点还是叶节点。为了说明这个模式，我们先设计一个层次式的文件系统。现在我只把重点放在两个特别重要的设计问题上，我将以这个例子为基础在下面的篇幅中向你展示模式内容的安排。从用户的角度看，文件系统应处理任意大小和复杂度的文件结构，它不应在文件结构能达到的广度和深度上做任何限定。从实现者的观点看，文件结构的表示应容易处理和扩展。

假设你正在实现一个列出目录中文件的命令。你所写的取目录名的代码和取文件名的代码不应有什么不一样。也就是说，你应能一致的对待目录和文件而不管他们叫什么名字。这样得出的代码会比较容易写和维护。你也会想处理新类型的文件（比如 symbolic links），而不用把半个系统从头实现一遍。

很明显，文件和目录是我们的问题域中的关键元素，我们需要一种方法来引进这些元素的特殊版本，即使在我们完成了设计之后。一个显然的方法是把这些元素表示成对象。如图 1 所示。

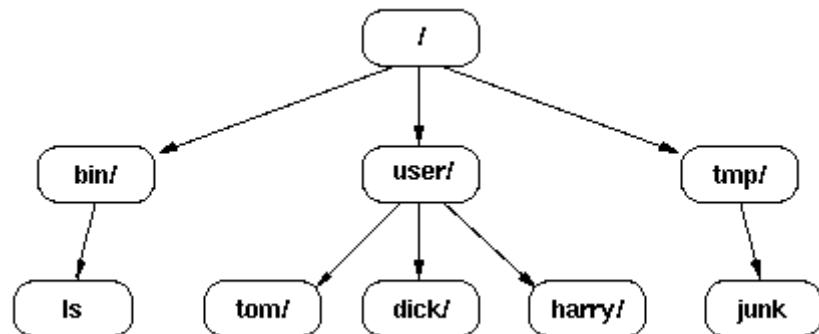


图 1

你怎样实现这样的结构呢？情况是我们有两种类型的对象（两个类），一种是文件，一种是目录。我们想统一的对待文件和目录，这意味着他们必须有一个公共接口。那也意味着这两个类必须从一个公共（抽象）基类（取名为“Node”）继承。我们也知道文件包含在目录中。总之，这些限制从根本上定义了类层次：

```

class Node {
public:
    // declare common interface here
protected:
    Node();
};

class File : public Node {
public:
    File();

    // redeclare common interface here
}
  
```

```

};

class Directory : public Node {
public:
    Directory();

    // redeclare common interface here
private:
    List _nodes;
};

```

下一个问题是关于公共接口的组成。哪些是文件及目录共有的操作呢？还有相同的属性 `:name`, `size`, `protection` 等等，每个属性要有读取和修改其值的操作。像这种文件及目录明显共有的操作统一处理很容易。但当操作并不是明显的适用于两个类时技巧性的东西就表现出来了。

比如，用户用的最多的是列出目录中文件，那意味着目录需要一个接口来遍历它的孩子。这里是一个返回第 `n` 个孩子的简单版：

```
virtual Node* GetChild(int n);
```

`GetChild` 必须返回一个 `Node *`，因为目录可以包括 `File` 对象或 `Directory` 对象。返回值的类型带来一个重要的后果：它强迫我们不仅要在类 `Directory` 中定义 `GetChild`，还要在类 `Node` 中定义 `GetChild`。为什么呢？因为我们需要列出子目录的孩子。实际上，用户会经常想要在文件系统的结构自上向下遍历。除非我们能在 `GetChild` 返回的对象中调用 `GetChild`，否则就不能满足要求。所以，如同对属性的操作一样，`GetChild` 也是我们所要的统一的操作。

`GetChild` 也是让我们递归定义 `Directory` 操作的关键。例如，假设类 `Node` 声明了一个 `Size` 操作返回该目录树共占用的字节，类 `Directory` 可以定义他自己的操作版本为孩子们的 `Size` 操作返回值的累计：

```

long Directory::Size () {
    long total = 0;
    Node* child;

    for (int i = 0; child = GetChild(i); ++i) {
        total += child->Size();
    }

    return total;
}

```

目录和文件的例子说明了 Composite pattern 的各个重要方面：他生成了任意复杂度的树结构，揭示了如何统一的对待那些对象。适用性 (Applicability) 部分叙述了这些方面。这部分的内容告诉

我们在下列时候使用组合 (Composite) :

- 你希望表示部分 - 整体的对象结构。
- 你希望用户能不用关心对象组合与单个对象的区别。用户可以同一的对待组合 (Composite) 结构中的所有对象。

组合 (Composite) 模式的结构 (Structure) 部分是 Composite 类结构正统的 OMT 图。说正统是因为它代表了我们 (四人帮) 所注意到的最通用的类的布局。它并不是类及其关系绝对的表示，因为对不同的特定设计及实现来说，接口是变化的。(图示的模式当然也可以实现那些)

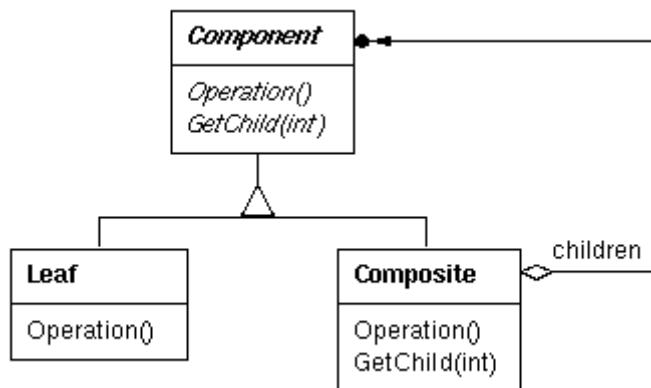


图 2

图 2 表示了该模式中的成员类及其静态关系。Component 是我们前述 Node 的对应。还有成员类：子类 Leaf (对应 File) 和子类 Composite (对应 Directory) 。箭头线从 Composite 指向 Component 表示 Composite 包含 Component 的实例，箭头顶端的圆表示可以有多个实例，如果没有圆，表示只有一个实例。箭头线尾部的菱形表示 Composite 拥有它的孩子，也就是说，删除 Composite 也会删除它的孩子，这也保证了 Component 的对象是不共享的，确保了树结构。

该模式的参与者 (Participant) 和协作 (Collaboration) 部分分别描述了这些成员间的静态和动态关系。

组合 (Composite) 的效果 (Consequences) 部分总结了该模式的好处及可靠性。另外，该模式支持任意复杂度的树结构。这个特性的好处是节点 (node) 的复杂性向客户隐藏了：客户不需要判断他们操作的 Component 是一个 leaf 还是一个 Composite，因为他们无需知道。这使客户代码更独立于 Component 的代码。客户代码也更简单，因为它能统一的对待 Leaf 和 Composite。客户不需要去考虑怎样根据 Component 类型的不同来执行不同的代码。最好的是，你可以添加新的 Component 类型而不必触动现有的代码。

然而，组合 (Composite) 的负面影响是他会导致这样一个系统，其中的每个类看起来都好像其他的类。期间明显的区别只在运行时才体现出来。这使得代码很难懂，即使是你自己做的类实现。更甚，如果该模式使用在一个较低的层次上或过小的点上，对象的数量会是不可容忍的。

正如你猜到的，有很多关于组合(Composite)模式的实现的讨论。我们将关注的一些讨论包括：

- 何时何地使用缓存储存信息以提升性能；
- 与 Component 类申请存储空间是否有关系；
- 用什么数据结构来存储孩子；
- 是否需要在 Component 类中声明增加及删除孩子的操作；

由于篇幅有限，我会在今后详细地讨论这些以及其他实现的问题。

结束语

人们倾向于以下两者之一的方式来理解设计模式，我会在下面比较说明这两种方式。

设想一个电子爱好者：没受过什么正式的训练，却也在数年中设计制作了一些新奇的小玩艺，比如收音机、盖革(Geiger)计数器、安全警报器等等。一天这个爱好者认为应该回学校得到一个电子的学位，为自己的才能获取一些官方的认可。当翻开教科书时，他惊讶于书中的内容是这么眼熟，不是指术语和符号，而是那些潜在的理念。爱好者继续着看见许多名词和他已经潜在使用了多年的理所当然的东西。在他看来，那些只是一个接一个的顿悟罢了。

现在来看看上同样课程，学习同样材料的一年级学生。这个学生在别的方面很杰出，但是电子方面还是刚刚开始。课程中的材料对他来说很痛苦，并不因为他笨，但是课本的内容对他来说是全新的。这个学生花了好多时间来理解那些知识。在辛苦的努力之后，他终于做到了。

如果你像那个电子爱好者，那就更努力吧。反之，如果你像那个一年级学生，那就用点心，你在学习好的模式时付出的努力会在你每次将之运用于设计时回报你。你无需怀疑！

也许电子及电子爱好者、学生的比喻并不对每个人都是最好的类比。如果你乐意，想想 Alfred North Whitehead 在 1943 年说的（即使在现在也是显然的），这番话可能会和你有一种更加亲切的共鸣：艺术就是从经验中提取出模式，以及在此过程中获得的优雅的享受。

参考资料

1. Gamma, E., R. Helm, R. Johnson, J. Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*, Addison-Wesley, Reading, MA, 1995.
2. Alexander, C., et al. *A Pattern Language*, Oxford University Press, New York, 1977.
3. Gamma, E. *Object-Oriented Software Development Based on ET++: Design Patterns, Class Library, Tools*, (in German), PhD thesis, University of Zurich Institut fur Informatik, 1991.

P = A + D

vector 容器的性能分析

starfish

C++ View 第二期的一篇文章《STL 有序容器武道会》通过实验的方式比较了 STL 中的三种容器 vector、dqueue 和 list 的性能。该文中对 vector 进行了 100000 次 push_back 操作，但却发现容器中对象的拷贝构造函数被调用了 231071 次。根据该文的解释，“当容器 vector 建立的时候，他会为数据预先开辟一个保留空间，这个空间通常不是很大，在使用过程中，如果实际使用空间超过了这个限定，vector 会再次申请内存，然后将原有的数据和新的数据一起转移到新的空间里去，这样周而复始，所以，vector 搬家的动静是很大的。”

这个解释很难令人满意：

1. 当 vector 再次申请内存的时候，他会申请多大的内存？
2. 为什么进行 100000 次 push_back，对象的拷贝构造函数恰好被调用 231071 次？
3. 通过进一步的实验我们可以发现，无论 push_back 多少次，对象的拷贝构造函数被调用的次数总是介于 push_back 的次数的二至三倍之间，为什么？

要想回答上述问题，就必须了解 vector 的实现细节，并从理论上分析 vector 的性能。

动态表

STL 中的 vector 实际上是一种称为**动态表**的数据结构。所谓动态表，是指一种能够自动扩张和收缩的表。在开始的时候动态表预先分配一块连续的存储空间，将表中的元素存储在这块连续的空间内；如果不间断地向表中插入元素，最终原来的那块存储空间可能会不够用，这时候动态表就会自动分配一块更大的存储空间，并将表中原来的元素全部复制到这块较大的空间内，这个过程叫做**表的扩张**；类似地，如果有许多对象被从表中删去了，就应该给原表分配一个更小的空间，并且将所有的元素复制到新的存储区内，这个过程叫做**表的收缩**。在后文中，我们要具体分析表的动态扩张和收缩的性能。

我们首先定义非空表 T 的装载因子 $\alpha(T)$ 为表中存储的对象项数和表的大小(表中最多可容纳的元素的个数)的比值。对一个没有元素的空表，定义其大小为 0，其装载因子为 1。如果某一动态表的装载因子以一个常数为上界，则表中未使用的空间就始终不会超过整个空间的某一常数部分。

表的扩张

这里我们通过 vector 的 push_back 操作来分析动态表的扩张的性能。当执行 push_back 操作的时候，如果 vector 原来的存储空间还有空余，则直接将元素插入表尾即可；如果存储空间已满，则要重新分配一块存储空间，将原来表中的所有元素全部复制到该空间内，然后再将新元素插入到表尾。

这里有一个问题：新申请的空间应该以多大为宜？vector 所采取的算法策略是：分配一个比原表大一倍的新表。这样如果只对 vector 做 push_back 操作，表的装载因子总是至少为 $1/2$ ，浪费掉的空间就始终不会超过表的总空间的一半。

在下面的伪代码中，我们假设对象 T 表示一个表，域 table[T] 包含了一个指向表的存储块的指针，域 num[T] 包含了表中的项数，域 size[T] 为表中总的槽数。开始时，表是空的：num[T]=size[T]=0。

```

01 push_back(T, x)
02 if size[T]=0
03     then 给 table[T] 分配一个槽的空间
04         size[T]←1
05 if num[T]=size[T]
06     then 分配一个有 2*size[T] 个槽的空间的新表
07         将 table[T] 中所有的项复制到新表中
08         释放 table[T]
09         table[T] 指向新表的存储块地址
10         size[T]←2*size[T]
11     将 x 插入 table[T] 的尾部
12     num[T]←num[T]+1

```

上述的伪代码描述了 push_back 的大致过程，对于 vector 的 insert 操作，其伪代码与此类似。上述伪代码中第 11 行是一次基本插入，只需要一个单位的时间；第 6~10 行是一次表的扩张，其所需的时间和表中的元素成线性关系。

现在我们来分析一下作用于一个初始为空的表上的 n 次 push_back 操作的序列。设第 i 次操作的代价 c_i ，如果在当前的表中还有空间（或该操作是第一个操作），则 $c_i = 1$ ，因为这时我们只需在第 11 行中执行一次基本插入操作即可。如果当前的表是满的，则发生一次扩张，这时 $c_i = i$ ，即第 11 行中基本插入操作的代价 1 再加上第 7 行中将原表中的项复制到新表中的代价 $i-1$ 。如果执行了 n 次操作，则一次操作的最坏情况代价为 $O(n)$ ，由此可得 n 次操作的总的运行时间的上界 $O(n^2)$ 。

但这个界不很精确，因为在执行 n 次 push_back 操作的过程中并不常常包括扩张表的代价。特别地，仅当 $i-1$ 为 2 的整数幂时第 i 次操作才会引起一次表的扩张。实际上，每一次操作的平摊代价为 $O(1)$ ，这一点我们可以用 **聚集方法**加以证明。第 i 次操作的代价为

$$c_i = \begin{cases} i, & \text{如果 } i-1 \text{ 是 } 2 \text{ 的幂} \\ 1, & \text{否则} \end{cases}$$

由此，n 次 push_back 操作的总代价为

$$\sum_{i=1}^n c_i = n + \sum_{j=1}^{\lfloor \log_2 n \rfloor} 2^j < 3n \quad (1)$$

可见，连续的 n 次 push_back 操作的代价总是以 $3n$ 为上界的，平均每次 push_back 的代价就是

3。

我们将 $n=100000$ 代入 (1) 式，恰好得到

$$\sum_{i=1}^{100000} c_i = 100000 + \sum_{j=1}^{\lfloor \log_2 100000 \rfloor} 2^j = 231071$$

这就解释了为什么对 vector 进行 100000 次 push_back 操作却调用了 231071 次拷贝构造函数。

通过采用**会计方法**，我们可以对为什么一次 push_back 操作的平摊代价会是 3 有一些认识。从直觉上看，每一项要支付三次基本 push_back 操作：将其自身插入现行表中，当表扩张时对其自身的移动，以及对另一个在扩张表时已经移动过的另一项的移动。例如，假设刚刚完成扩张后某一表的大小为 m ，那么表中共有 $m/2$ 项，且没有“存款”。对每一次插入操作要收费 3 元。立即发生的基本插入的代价为 1 元，另有 1 元放在刚插入的元素上作为存款，余下的 1 元放在已在表中原来的 $m/2$ 个项的某一项上作为存款。填满该表另需要 $m/2$ 次插入，这样，到该表包含了 m 个项时，该表已满，每一项上都有 1 元钱以支付在表扩张期间的插入。

也可以用**势能方法**来分析一系列 n 个 push_back 操作，我们还将在后文中用此方法来设计一个平摊代价为 $O(1)$ 的 pop_back 的操作。开始时我们先定义一个势函数 Φ ，在完成扩张时它为 0，当表满时它也达到表的大小，这样下一次扩张的代价就可由存储的势来支付了。函数：

$$\Phi(T) = 2 \cdot \text{num}[T] - \text{size}[T] \quad (2)$$

是一种可能的选择。在刚刚完成一次扩张后，我们有 $\text{num}[T]=\text{size}[T]/2$ ，于是有 $\Phi(T)=0$ ，这正是所希望的。在就要做一次扩张前，有 $\text{num}[T]=\text{size}[T]$ ，于是 $\Phi(T)=\text{num}[T]$ ，这也正是我们希望的。势的初值为 0，又因为表总是至少为半满， $\text{num}[T] \geq \text{size}[T]/2$ ，这就意味着 $\Phi(T)$ 总是非负的。所以， n 次 push_back 操作的总的平摊代价就是总的实际代价的一个上界。

为了分析第 i 次 push_back 操作的平摊代价，我们用 num_i 来表示在第 i 次操作后表中所存放的项数，用 size_i 表示在第 i 次操作之后表的大小， Φ_i 表示第 i 次操作之后的势。开始时， $\text{num}_0=0$ ， $\text{size}_0=0$ 和 $\Phi_0=0$ 。

如果第 i 次 push_back 操作没有能触发一次表的扩张，则 $\text{num}_i=\text{num}_{i-1}+1$ ， $\text{size}_i=\text{size}_{i-1}$ ，且该操作的平摊代价为

$$\begin{aligned}\hat{c}_i &= c_i + \Phi_i - \Phi_{i-1} \\ &= 1 + (2\text{num}_i - \text{size}_i) - (2\text{num}_{i-1} - \text{size}_{i-1}) \\ &= 1 + (2\text{num}_i - \text{size}_i) - (2(\text{num}_i - 1) - \text{size}_i) \\ &= 3\end{aligned}$$

如果第 i 次操作确实触发了一次扩张，则 $\text{num}_i=\text{num}_{i-1}+1$ ， $\text{size}_i=2\text{size}_{i-1}=2\text{num}_{i-1}=2(\text{num}_i-1)$ ，且该操作的平摊代价为

$$\begin{aligned}
\hat{c}_i &= c_i + \Phi_i - \Phi_{i-1} \\
&= num_i + (2num_i - size_i) - (2num_{i-1} - size_{i-1}) \\
&= num_i + (2num_i - 2(num_i - 1)) - (2(num_i - 1) - (num_i - 1)) \\
&= num_i + 2 - num_i + 1 \\
&= 3
\end{aligned}$$

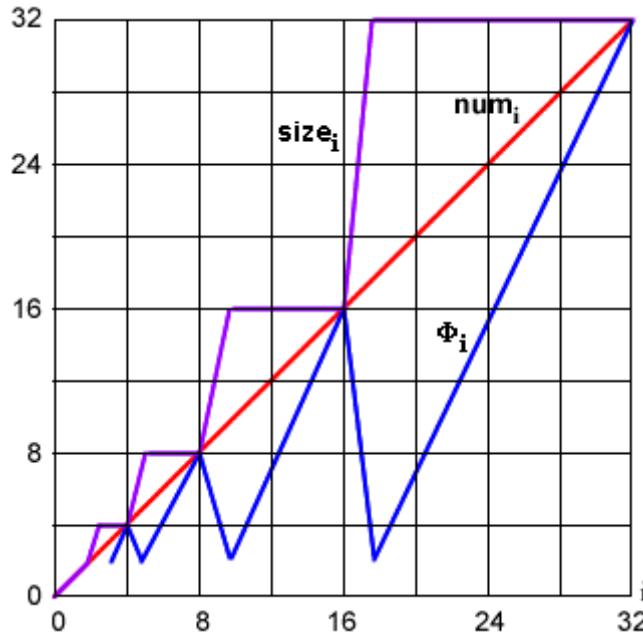


图 1 对表中项目数 num_i , 表中的空位数 $size_i$, 以及势 Φ_i 作用 n 次 `push_back` 操作的效果

图 1 画出了 num_i , $size_i$ 和 Φ_i 的各个值。在第 i 次操作后对这些量中的每一个都要加以计算。图中红线表示 num_i , 紫线表示 $size_i$, 蓝线表示 Φ_i 。注意在每一次扩张前 , 势已增长到等于表中的项目数 , 因而可以支付将所有元素移到新表中去的代价。此后 , 势降为 0 , 但一旦引起扩张的项目被插入时其值就立即增加 2。

表扩张和收缩

为了实现 `pop_back` 操作 , 只要将表尾的项从表中去掉即可。但是 , 当某一表的装载因子过小时 , 我们就希望对表进行收缩 , 使得浪费的空间不致太大。表收缩与表扩张是类似的 : 当表中的项数降得过低时 , 我们就要分配一个新的、更小的表 , 而后将旧表的各项复制到新表中。旧表所占用的存储空间则可被释放 , 归还到存储管理系统中去。在理想情况下 , 我们希望下面两个性质成立 :

- 动态表的装载因子由一常数作为下界 ;
- 各表操作的平摊代价由一常数作为下界。

另外 , 我们假设用基本插入和删除操作来测度代价。

关于表收缩和扩张的一个自然的策略是当向表中插入一个项时将表的规模扩大一倍，而当从表中删除一项就导致表的状态小于半满时，则将表缩小一半。这个策略保证了表的装载因子始终不会低于 $1/2$ ，但不幸的是，这样又会导致各表操作具有较大的平摊代价。请考虑一下下面这种情况：我们对某一表 T 执行 n 次操作，此处 n 为 2 的整数幂。前 $n/2$ 个操作是插入，由前面的分析可知其总代价为 $O(n)$ 。在这一系列插入操作的结束处， $\text{num}[T]=\text{size}[T]=n/2$ 。对后面的 $n/2$ 个操作，我们执行下面这样一个序列： I,D,D,I,I,D,D,I,I, \dots ，其中 I 表示插入， D 表示删除。第一次插入导致表扩张至规模 n 。紧接的两次删除又将表的大小收缩至 $n/2$ ；紧接的两次插入又导致表的另一次扩张，等等。每次扩张和收缩的代价为 $\Theta(n)$ ，共有 $\Theta(n)$ 次扩张或收缩。这样， n 次操作的总代价为 $\Theta(n^2)$ ，而每一次操作的平摊代价为 $\Theta(n)$ 。

这种策略的困难性是显而易见的：在一次扩张之后，我们没有做足够的删除来支付一次收缩的代价。类似地，在一次收缩后，我们也没有做足够的插入以支付一次扩张的代价。

我们可以对这个策略加以改进，即允许装载因子低于 $1/2$ 。具体来说，当向满的表中插入一项时，还是将表扩大一倍，但当删除一项而引起表不足 $1/4$ 满时，我们就将表缩小为原来的一半。这样，表的装载因子就以常数 $1/4$ 为下限界。这种做法的基本思想是使扩张以后表的装载因子为 $1/2$ 。因而，在发生一次收缩前要删除表中一半的项，因为只有当装载因子低于 $1/4$ 时方会发生收缩。同理，在收缩之后，表的装载因子也是 $1/2$ 。这样，在发生扩张前要通过扩张将表中的项数增加一倍，因为只有当表的装载因子超过 1 时方能发生扩张。

我们略去了 pop_back 的代码，因为它与 push_back 的代码是类似的。为了方便分析，我们假定如果表中的项数降至 0，就释放该表所占存储空间。亦即，如果 $\text{num}[T]=0$ ，则 $\text{size}[T]=0$ 。

现在我们用势能方法来分析由 n 个 push_back 和 pop_back 操作构成的序列的代价。先定义一个势函数 F ，它在刚完成一次扩张或收缩时值为 0，并随着装载因子增至 1 或降至 $1/4$ 而变化。我们用 $\alpha(T)=\text{num}[T]/\text{size}[T]$ 来表示一个非空表 T 的装载因子。对一个空表，因为有 $\text{num}[T]=\text{size}[T]=0$ ，且 $\alpha(T)=1$ ，故总有 $\text{num}[T]=\alpha(T)*\text{size}[T]$ ，无论该表是否为空。我们采用的势函数为

$$\Phi(T) = \begin{cases} 2 \cdot \text{num}[T] - \text{size}[T] & \text{若 } \alpha(T) \geq 1/2 \\ \text{size}[T]/2 - \text{num}[T] & \text{若 } \alpha(T) < 1/2 \end{cases} \quad (3)$$

请注意，空表的势为 0；势总是非负的。这样，以 Φ 表示的一列操作的总平摊代价即为其实际代价的一个上界。

在进行详细分析之前，我们先来看看势函数的某些性质。当装载因子为 $1/2$ 时，势为 0。当它为 1 时，有 $\text{size}[T]=\text{num}[T]$ ，这就意味着 $\Phi(T)=\text{num}[T]$ ，这样当因插入一项而引起一次扩张时，就可用势来支付其代价。当装载因子为 $1/4$ 时，我们有 $\text{size}[T]=4*\text{num}[T]$ ，它意味着 $\Phi(T)=\text{num}[T]$ ，因而当删除某个项引起一次收缩时就可用势来支付其代价。图 2 说明了对一系列操作势是如何变化的。

图 2 中红线表示 num_i ，紫线表示 $size_i$ ，蓝线表示 Φ_i 。注意在每一次扩张前，势已增长到等于表中的项目数，因而可以支付将所有元素移到新表中去的代价。类似地，在一次收缩之前，势也增加到等于表中的项目数。

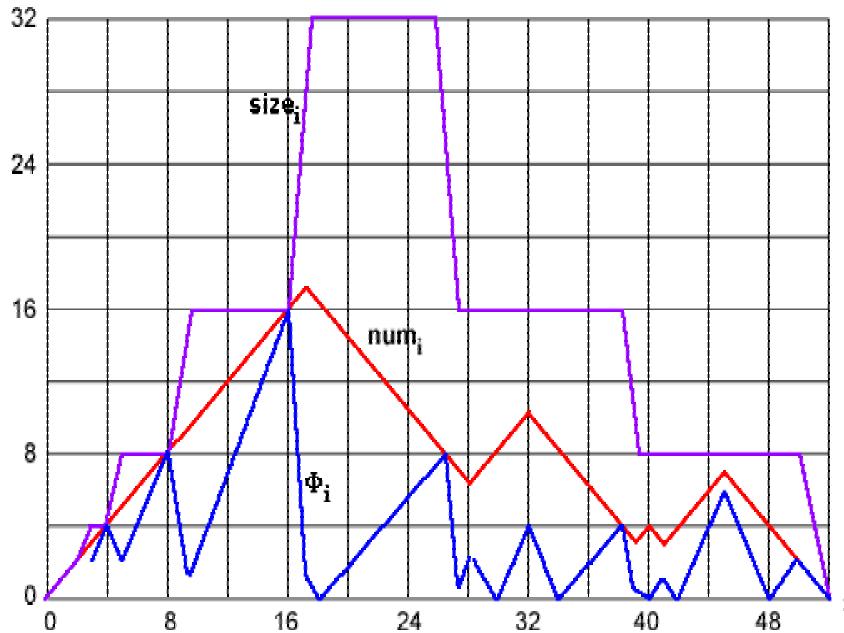


图 2 对表中的项目数 num_i 表中的空位数 $size_i$ 及势 Φ_i
作用 n 个 `push_back` 和 `pop_back` 操作的效果

为分析 n 个 `push_back` 和 `pop_back` 的操作序列，我们用 \hat{c}_i 来表示第 i 次操作的实际代价， c_i 表示其参照 F 的平摊代价， num_i 表示在第 i 次操作之后表中存储的项数， $size_i$ 表示第 i 次操作后表的大小， α_i 表示第 i 次操作后表的装载因子， F_i 表示第 i 次操作后的势。开始时， $num_0=0$, $size_0=0$, $a_0=1$, $F_0=0$ 。

我们从第 i 次操作是 `push_back` 的情况开始分析。如果 $\alpha_{i-1} \geq 1/2$ ，则所要做的分析就与对表扩张的分析完全一样。无论表是否进行了扩张，该操作的平摊代价 c_i 都至多是 3。如果 $\alpha_{i-1} < 1/2$ ，则表不会因该操作而扩张，因为仅当 $\alpha_{i-1}=1$ 时才发生扩张。如果还有 $\alpha_i < 1/2$ ，则第 i 个操作的平摊代价为

$$\begin{aligned}\hat{c}_i &= c_i + \Phi_i - \Phi_{i-1} \\ &= 1 + (size_i/2 - num_i) - (size_{i-1}/2 - num_{i-1}) \\ &= 1 + (size_i/2 - num_i) - (size_i/2 - num_i - 1) \\ &= 0\end{aligned}$$

如果 $\alpha_{i-1} < 1/2$ ，但 $\alpha_i \geq 1/2$ ，那么

$$\begin{aligned}
\hat{c}_i &= c_i + \Phi_i - \Phi_{i-1} \\
&= 1 + (2num_i - size_i) - (size_{i-1}/2 - num_{i-1}) \\
&= 1 + [2(num_{i-1} + 1) - size_{i-1}] - (size_{i-1}/2 - num_{i-1}) \\
&= 3num_{i-1} - \frac{3}{2}size_{i-1} + 3 \\
&= 3\alpha_{i-1}size_{i-1} - \frac{3}{2}size_{i-1} + 3 \\
&< \frac{3}{2}size_{i-1} - \frac{3}{2}size_{i-1} + 3 \\
&= 3
\end{aligned}$$

因此，一次 push_back 操作的平摊代价至多为 3。

现在我们再来分析一下第 i 个操作是 pop_back 的情形。这时， $num_i = num_{i-1} - 1$ 。如果 $\alpha_{i-1} < 1/2$ ，我们就要考虑该操作是否会引起一次收缩。如果没有，则 $size_i = size_{i-1}$ ，而该操作的平摊代价则为

$$\begin{aligned}
\hat{c}_i &= c_i + \Phi_i - \Phi_{i-1} \\
&= 1 + (size_i/2 - num_i) - (size_{i-1}/2 - num_{i-1}) \\
&= 1 + (size_i/2 - num_i) - [size_i/2 - (num_i + 1)] \\
&= 2
\end{aligned}$$

如果 $\alpha_{i-1} < 1/2$ 且第 i 个操作触发一次收缩，则该操作的实际代价为 $c_i = num_i + 1$ ，因为我们删除了一项，移动了 num_i 项。这时， $size_i/2 = size_{i-1}/4 = num_i + 1$ ，而该操作的平摊代价为

$$\begin{aligned}
\hat{c}_i &= c_i + \Phi_i - \Phi_{i-1} \\
&= (num_i + 1) + (size_i/2 - num_i) - (size_{i-1}/2 - num_{i-1}) \\
&= (num_i + 1) + [(num_i + 1) - num_i] - [(2num_i + 2) - (num_i + 1)] \\
&= 1
\end{aligned}$$

当第 i 次操作为 pop_back 且 $\alpha_{i-1} < 1/2$ 时，其平摊代价仍有一常数上界。具体的分析从略。

总之，因为每个操作的平摊代价都有一常数上界，所以作用于一动态表上的 n 个操作的实际时间为 $O(n)$ ，每个操作的平摊时间就是 $O(1)$ 。

至此我们解决了在本文开头提出的三个问题。我们可以看到，vector 是一种可以自动扩张和搜索的动态表，他的扩张和收缩的代价平摊到每个插入或删除操作上只有 $O(1)$ 的复杂度，在表尾进行一次插入或删除操作的平均代价不超过 3；但是如果要在表中间进行插入或删除，除了支付表的扩张和收缩的代价，还要支付移动表中元素的代价，因此在表中间进行插入或删除平均代价是 $n/2+3$ ，其中 n 是表的长度。具体分析方法与对数组平均性能的分析类似，这里从略。

模式罗汉拳

委托模式

[透明](#)

梗概

委托是对一个类的功能进行扩展和复用的方法。它的做法是：写一个附加的类提供附加的功能，并使用原来的类的实例提供原有的功能。

场景

扩展和复用一个类的功能常用的一种方法是继承，而另一种更普遍的方法则是委托。在很多情况下委托很适用，而继承则并不适用。另外在[MEYERS98]中也讲到，公有继承表现的设计思想是“is-a-kind-of”⁴，私有继承表现的设计思想则是“is-implemented-in-terms-of”，这些关系都是静态的、不能在运行时改变的。而在一些情况下我们需要表现的设计思想是“is-a-role-played-by”的关系，在这些情况下不应该用继承的方法。

下面用一个例子来帮助说明。假设我们为一个航空公司设计软件系统，于是我们必须用一些类来表示各种各样的“人”，包括机组人员、售票员、旅客等等。一种思路是这样：因为这些都是抽象的“人”，因此设计一个 Person 抽象类，并从这个抽象类衍生出我们需要的各种类。由此我们得到下面的类图：

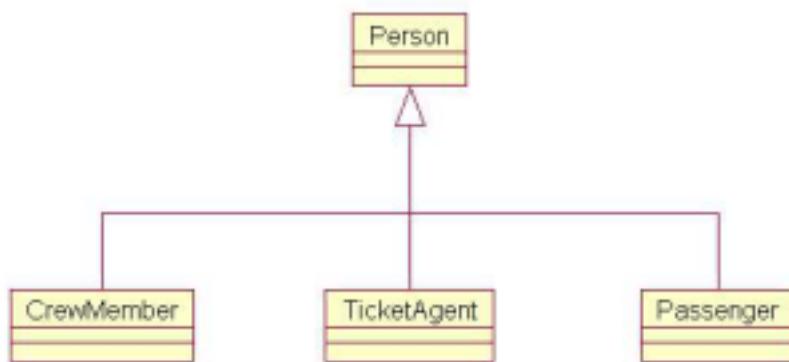


图 1：用继承的方法建模

这个设计方案的问题是很明显的：一个机组人员在休假的时候可能乘坐飞机而成为一个旅客；航空公司也可能把机组人员调去做售票员……是的，一个人可能成为三种角色中的任何一种。如果我们一定要坚持继承的思路，那么我们可能得到下面这个图：(见下页图 2)

⁴ [MEYERS98]的 ITEM35 指出，public 继承表示“is-a”的关系，也就是这里所说的“is-a-kind-of”。

很明显我们遇到了“类爆炸”的问题。我们这里只有三个角色，就需要用七个衍生类来表现所有的情况。如果我们有六个角色呢？我们将需要63个衍生类。（我画图2用了15分钟的时间，如果要画63个衍生类……呵呵，呵呵）

而且即使使用了这么多衍生类，我们仍然有困难。因为继承所表现的“is-a-kind-of”关系是静态的，在编译时就固定了。而一个“人”可能在不同的时间扮演不同的角色，于是我们可能需要用多个对象来表现同一个“人”的不同角色，这也是一件相当麻烦的事情。

而另一方面，如果我们用委托的方式来表现这个问题，我们可以得到一个相当优雅的解决方案，上面提到的问题都自然的解决了。这样的解决方案如图3所示：

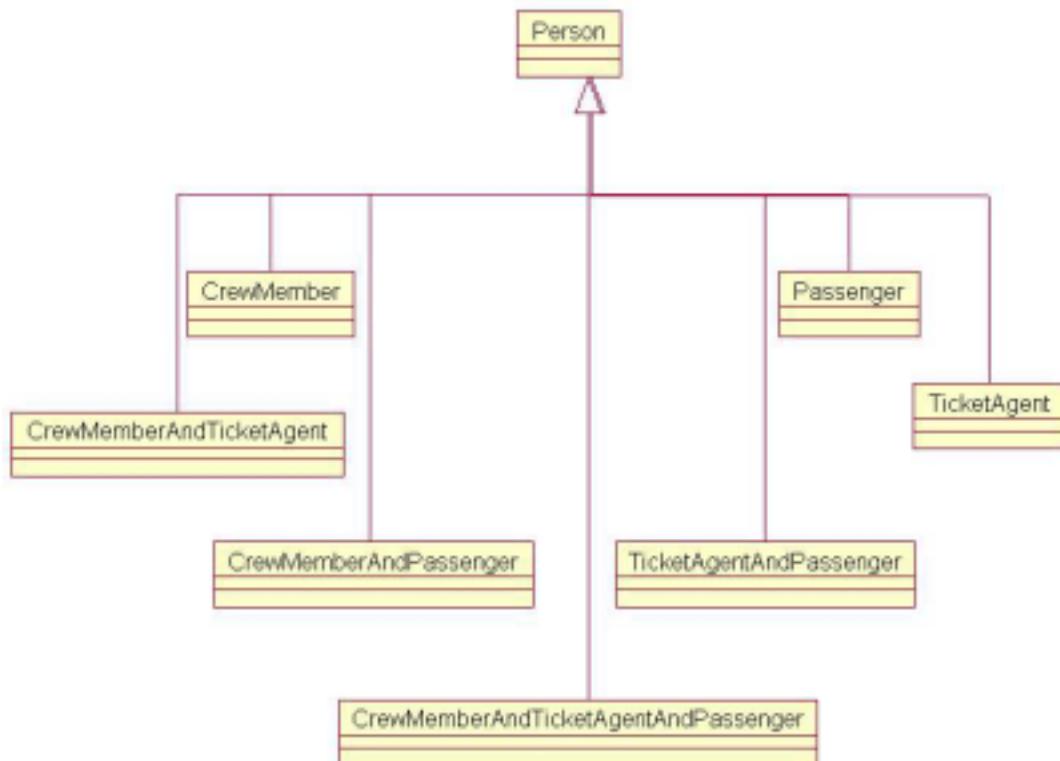


图2：继承解决方案的发展（类爆炸的实例）

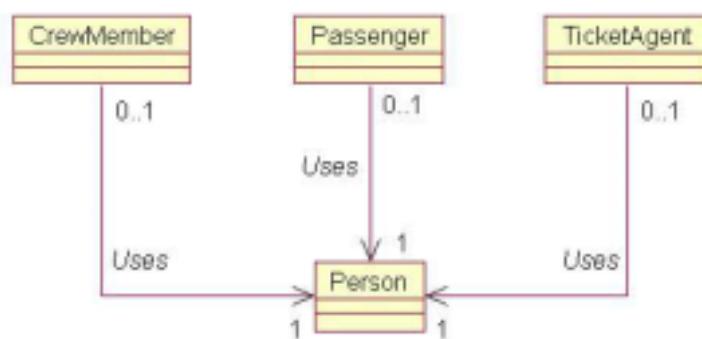


图3：使用委托的建模

约束

如果你发现一个对象需要在不同的时间“成为”不同的衍生类，那么首先这个对象根本不应该是“是”一个衍生类。因为一个对象一旦作为衍生类被创建出来，它就只能是这个衍生类的实例而不能扮演其他角色了。另一方面，一个对象可以在不同的时间把不同的行为委托给不同的对象。

如果你发现一个衍生类在试图隐藏其超类的方法或变量，这说明这个类根本不应当从这个超类衍生得到，因为根本没有什么合理的理由来隐藏超类的方法或变量。但另一方面，如果使用委托的设计方法，你就可以随意选择需要的方法或变量。

把一个类设计成现有的具体类的衍生类也不是一件值得推荐的事情。（这个话题的 C++ 版本在 [MEYERS96] 中有非常详细的介绍，因此我就不再这里赘言了。）

“不适当的继承”在实际中被如此广泛的应用，以至于可以把它们归纳成一种“反模式（antipattern）”了。正如上面所说的，继承一个具体类可能导致各种无法预料的问题。实际上，可能绝大多数对类的功能的扩展和复用都不应该使用继承。

解决方案

委托是对类的行为进行复用和扩展的一条途径。它的工作方式是：包含原有类的实例引用，实现原有类的接口，将对原有类方法的调用转发给内部的实例引用。图 4 展示了本模式的一般形式：



图 4：使用委托模式对类的行为进行复用和扩展

委托的用途比继承更加广泛。用继承能实现的对类的任何形式的扩展都可以用委托的方式完成。因此在[GoF]中也建议尽量用委托代替继承。

参与者

- Delegator（委托者）
 - 保存 Delegate 的实例引用。
 - 实现 Delegate 的接口。
 - 将对 Delegate 接口方法的调用转发给 Delegate。

- Delegate（受委托者）
 - 接受 Delegator 的调用，帮助 Delegator 实现其接口。

效果

使用委托模式可以避免继承方法遇到的问题。另外，使用委托模式可以很容易的在运行时对其进行组合。

委托模式的主要缺点是类之间的联系、类体系结构不如继承那样清楚明显。不过也有一些方法可以改善这些联系的清楚程度。

- 使用一致并且清楚的名称，让程序的读者可以直观的知道 Delegator 和 Delegate 之间的联系。比如说，如果用一个类来代理一些 Widget 衍生类的创建，那么把这个类命名为 WidgetFactory 就是不错的方法。
- 在程序中写上适当的注释，告诉读者：这里使用委托模式。
- 遵循 Law of Demeter 模式[GRAND99]，即：如果两个类之间只有间接联系，采用间接委托；如果有直接联系，采用直接委托。这可以减少类之间的联系数量。
- 使用大家都知道的设计和编码模式。因为委托模式是很多其他模式的基础，如果在其他模式中使用委托模式，读者更容易理解。

实现

委托模式的实现是非常直接的。Delegator 保存 Delegate 的实例引用，并转发相应的方法调用。

C++应用

委托模式可以说是无处不在的，C++中的例子俯拾皆是。一个例子就是标准库中的 `auto_ptr`。`auto_ptr` 用一个私有变量保存原始指针，并将对 `operator*` 和 `operator&` 的调用转发给原始指针。（在[MEYERS96]中有对 `auto_ptr` 的详细描述。）

代码示例

作为实际的例子，我们仍然考虑“场景”一节中讲的航空公司软件系统。这一次我们要在“飞行段”⁵中随时检查飞机上行李的数量。我们用 `FlightSegment` 类表示飞行段，用 `LuggageCompartment` 表示行李舱，Client 将使用 `checkLuggage()` 函数向 `FlightSegment` 查询行李情况。一个飞行段可能会使用不同的飞机，当然也就有不同的行李舱了。我们使用委托模式实现对行李的检查，得到如图 5 所示的类图：

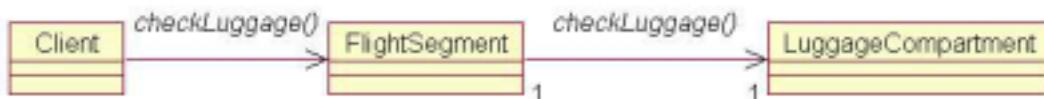


图 5：“行李检查”的例子中使用的类

⁵

“飞行段”是指旅行中的一部分。一个飞行段中，飞机将从一个机场起飞、在另一个机场停留、然后再次起飞，旅客不必换飞机。

如图所示，任意时刻一个 FlightSegment 都保存一个 LuggageCompartment 的指针（当飞行进行的时候，这个指针不为 NULL；没有飞行的时候，这个指针则为 NULL），它把 checkLuggage() 函数调用转发给 LuggageCompartment，并将返回值传回给 Client。

首先是 FlightSegment 的代码：

```
class LuggageCompartment;           //前置声明，以保证成员指针正确声明。
class FlightSegment
{
private:
    LuggageCompartment * m_pLC;      //成员指针
public:
    void SetLuggageCompartment(LuggageCompartment * pLC)
    {   m_pLC = pLC; }   //设置成员指针
    FlightSegment()           //构造函数将成员指针初始化为 null
    {   m_pLC = 0; }
    int checkLuggage()
    {
        if(m_pLC==0)
            return -1;
        return m_pLC->checkLuggage(); //将函数调用委托给成员指针
    }
};
```

然后是 LuggageCompartment 的代码：

```
class LuggageCompartment
{
private:
    int m_iLuggage;                //私有变量，保存现在的行李总数
public:
    LuggageCompartment();          //构造函数
    {   m_iLuggage = 0; }
    int TakeoutLuggage();          //取出一件行李
    {
        if(m_iLuggage!=0)
            m_iLuggage--;
        return m_iLuggage;
    }
    int InsertLuggage();           //放入一件行李
    {   return (++m_iLuggage); }
    int checkLuggage();            //检查行李总数
    {   return m_iLuggage; }
```

```
};
```

最后是 client 可能写出这样的代码：

```
...
#include <iostream.h>
...
FlightSegment segment;
LuggageCompartment lc1, lc2;
...
for(int i=0; i<10; i++)
    lc1.InsertLuggage();
...
segment.SetLuggageCompartment(&lc1);
cout<<"Now we have "<<segment.checkLuggage()<<" Luggages."<<endl;
...
segment.SetLuggageCompartment(&lc2);
cout<<"Now we have "<<segment.checkLuggage()<<" Luggages."<<endl;
...
```

(以上程序在 MICROSOFT VC++6.0 及 BORLAND C++ BUILDER 5.0 编译通过。)

相关模式

几乎所有其他模式都使用了本模式。一些模式——例如 Decorator 模式和 Proxy 模式[GRAND98]对 Delegation 模式的依赖是最为明显的。

参考书目

- [GoF] Gamma etc., *Design Patterns: Elements of Reusable Object-Oriented Software*. [Addison-Wesley](#) 1995. 中文版：《设计模式：可复用面向对象软件的基础》，李英军等译，机械工业出版社 2000 年。
- [GRAND98] [Mark Grand](#), *Patterns in Java (volume 1)*, [Wiley computer publishing](#) 1998.
- [GRAND99] [Mark Grand](#), *Patterns in Java (volume 2)*, [Wiley computer publishing](#) 1999.
- [MEYERS96] Scott Meyers, *More Effective C++*, [Addison-Wesley](#) 1996. 繁体中文版：《More Effective C++国际中文版》，[侯捷](#)译，[培生教育出版集团](#) 2000 年。
- [MEYERS98] Scott Meyers, *Effective C++ (2nd Edition)*, [Addison-Wesley](#) 1998. 繁体中文版：《Effective C++ (2nd Edition) 国际中文版》，[侯捷](#)译，[培生教育出版集团](#) 2000 年。

天方夜谭 VCL

作者：虫虫

前言

如果你爱他，让他学 VCL，因为那是天堂。
如果你恨他，让他学 VCL，因为那是地狱。
《天方夜谭 VCL》

传说很久很久以前，中国和印度之间有个岛。那里的国王每天娶一个女子，过夜后就杀，闹得鸡犬不宁，最后宰相的女儿自愿嫁入宫。第一晚，她讲了一个非常有意思的故事，国王听入了迷，第二天没有杀她。此后她每晚讲一个奇特的故事，一直讲到第一千零一夜，国王终于幡然悔悟。这就是著名的《一千零一夜》，也就是《天方夜谭》。印度和中国陆地接壤，那么相信传说中所指的岛，必然是在南中国海 - 马六甲海峡 - 印度洋某个地方。现在我也算是在这其间的一个海岛上，正值夜晚，也就借借“天方夜谭”的大名吧。

初中我最喜欢的编程环境是 Turbo C 2.0，高一开始用 Visual Basic。后来用了没多久就发现，如果想做一个稍微复杂的东西，就需要不停地查资料来调用 API，得在最前面作一个长得可怕的 API 函数声明。于是我开始怀念简洁的 C 语言。有位喜欢用 Delphi 的师哥，知道我极为愤恨 Pascal，把我引向 C++ Builder。即使对于 C++ 中的继承、多态这些简单概念都还是一知半解，我居然也开始用 VCL 编一些莫名其妙的小程序（VCL 上手倒真容易），开始熟悉 VCL 的结构，同时也了解了 MFC 和 SDK，补习 C++ 的基础知识。后来我才觉得，VCL 易学易用根本是个谎言。其实 VCL 相当难学，甚至比 MFC 更麻烦。

不知道为什么，C++ Builder 的资料出奇地少，也许正是这个原因，C++ Builder 论坛上的人情味也特别浓。不管是初学 VCL 时常问些莫名其妙白痴问题的天极论坛，还是现在我经常驻足的 CSDN，C++ Builder 论坛给人的感觉总是很温馨。每次 C++ Builder 都比同等版本 Delphi 晚出，每次用 C++ 还不得不看 Object Pascal 的脸色，我想这是很多人心里的感受。CLX 已经出现在 Delphi6 中，C++ Builder6 的发布似乎还遥遥无期。CLX 会代替 VCL 吗？看来似乎不会，后面还会提到。我也看过不少号召把 VCL 用 C++ 改写的帖子，往往雷声大雨点小。看看别人老外，说干就干，一个 FreeCLX 项目就这么启动了。

用 MFC 的人比用 VCL 的运气好，他们有 Microsoft 的支持，有 *Inside Visual C++*、*Programming Windows 95 with MFC*、*MFC Internals* 这些天王巨星的英文名著和中文翻译，也有诸如侯捷先生的《深入浅出 MFC》（即 *Dissecting MFC*）这些出色的中文原创作品。使用 Delphi 的人也远比使用 C++ Builder 的命好，关于 Delphi 的精彩资料远远比 C++ Builder 多，很无奈，真的很无奈。

C++ View 杂志的主编向我约稿，我很为难，因为时间和技术水平都成问题。借用侯捷先生一句话，要拒绝和你住在同一个大脑同一个躯壳的人日日夜夜旦旦夕夕的请求，是很困难的。于是我下决心，写一系列分析 VCL 内部原理的文章。所谓“天方夜谭”，当然对初学者不会有立杆见影的帮助，甚至于会让您觉得“无聊”。这些文章面向的朋友应该比较熟悉 VCL，有一定 C++ 的基础（当然会 Object Pascal 和汇编更好），比如希望知道 VCL 底层运作机制的朋友，和希望自己开发应用框架或者想用 C++ 重写 VCL 的朋友。同时我更希望大家交流一下解剖应用框架的经验，让我们不局限于 VCL 或者 MFC，能站在更高的角度看问题，共同提高自己的能力。

在深入探讨 VCL 之前，先得把 VCL 主要的性质说一下。

- 同 SmallTalk 和 Java 所带的框架一样，VCL 是 Object Pascal 的一部分，也就是说语言和框架之间没有明确的界限。比如 Java 带有 JDK，任写一个类都是 `java.lang.Object` 的子类。VCL 和 Object Pascal 是同样的道理。当然，Object Pascal 为了兼容以前的 Pascal，依然允许某个类没有任何父类，但本系列文章将不再考虑这种情形。
- 同大多数框架一样，VCL 采取的是单根结构。也就是说，VCL 的结构是以一棵 `TObject` 为根的继承树，除 `TObject` 外的所有 VCL 类都是 `TObject` 直接或间接的子类。
- 由于 Object Pascal 的语言特性，整个结构中只使用单继承。

所以，VCL 的本质是一个 Object Pascal 类库，提供了 Object Pascal 和 C++ 两个接口。在剖析的过程中，请时刻牢记这一点。

文章的组织结构是就事论事，一次一个话题。由于 VCL 并不像 MFC 是一个独立的框架，它与 Object Pascal、IDE、编译器结合非常紧密，所以在剖析过程中不免会提到汇编。当然不会汇编的朋友也不用怕，我会把汇编代码都解释清楚，并尽量用 C++ 改写。

文中有很多图是表示类的内存结构，如图所示。其中方框表示一个变量，两端伸出表示还有若干个变量，椭圆标注是说明虚线圆圈中的整个对象（在后面虚线圆圈不会画出）。

文中的程序，如非特别说明，均可以在 Console Application 模式下（如果使用了 VCL 类则需要复选“Use VCL”）编译通过。

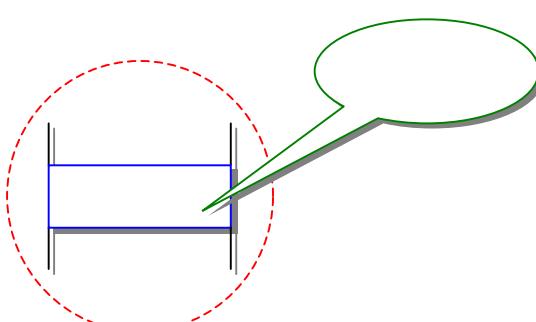


图 1 图例

开门

倒霉者如愚公，开门就见太行、王屋山。在一怒之下他开始移山，最后幸亏天神帮忙搬走了。中国人不喜欢开门见山的性格可能就是愚公传下来的，说话做事老爱绕弯。当然我也不能免俗，前面废话了一大堆，现在接着来。

提起 RTTI (runtime type identification，运行时间类型辨别)，相信大家都很熟悉。C++的 RTTI 功能相当有限，主要由 typeid 和 dynamic_cast 提供¹。至于这两者的实现方式²，不是我们今天的话题，我们所关注的，乃是 VCL 所提供的“高级”RTTI 的底层机制。

熟悉框架的朋友都知道，框架往往会提供“高级”的 RTTI 功能。我曾看过一个论调，说 Java 和 Object Pascal 比 C++ 好，原因是因为它们的 RTTI 更“高级”。且不论滥用 RTTI 极为有害，事实上，C++ 用宏 (macro) 亦可以模拟出相同功能的 RTTI³。

不过对于 VCL 类来说，您清楚其 RTTI 机制的运作情况吗？对于如下

```
class A: public TObject
{
    ...
}

...
A* p = new A;
```

为什么 p->ClassName()；就能返回类 A 的名字“ A ”呢？

为什么 A::ClassName(p->ClassParent()) 就可以返回 A 的基类名“ TObject ”呢？

为什么……？

其实这都是编译器暗箱操作的结果。说白了，编译器先在某个地方把类名写好，到时候去取出来就行。关键在于，如何去取出来呢？显然有指针指向这些数据，那么这些指针放在什么地方呢？

记得《阿里巴巴和四十大盗》的故事吧？宝藏是早就存在的，如果知道口诀“芝麻，开门吧”，就可以拿到宝藏。同样，类的相关信息是编译器帮我们写好了的，我们所关心的，就是如何获取这些信息的“口诀”。

不过这一切，要从虚函数开始，我们得先复习一下 C/C++ 的对象模型。

虚拟函数表 VFT

C 语言提供了基于对象 (Object-Based) 的思维模型，其对象模型非常清晰。比如

```
struct A
{
    int i;
    char c;
};
```

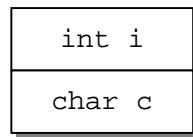


图 2 结构的内存布局

在 32 位系统上，变量 `i` 占用 4 个字节，变量 `c` 占用 1 个字节。编译器可能还会在后面添加 3 个字节补齐。那么，`sizeof(A)` 就是 8。

C++ 提供了面向对象 (Object-Oriented) 的思维模型，其对象模型建立在 C 的基础上。对于没有虚函数的类，其模型与 C 中的结构 (struct) 完全一样。但如果存在虚函数，一般在类实体的某个部分会存在一个指针 `vptr`，指向虚拟函数表 VFT (Virtual Function Table) 的入口。显然，对于同一个类的所有对象，这个 `vptr` 都是相同的。例如

```
class A
{
private:
    int i;
    char c;
public:
    virtual void f1();
    virtual void f2();
};

class B: public A
{
public:
    virtual void f1();
    virtual void f2();
};
```

当我们作如下调用的时候

```
A* p;
...
p->f2();
```

程序本身并不知道它会调用 `A::f` 还是 `B::f` 或是其它函数，只是通过类实体中的 `vptr`，查到 VFT 的入口，再在入口中查询函数地址，进行调用。由于 Borland C++ 编译器把 `vptr` 放在类实体的头部，因此下面均有此假设。

为了更充分地说明问题，我们从汇编级来分析一下。假设我们采用的是 Borland C++ 编译器。

```
p->f2();
```

这句的汇编代码是

```
mov eax,[ebp-0x04]
push eax
mov edx,[eax]
call dword ptr [edx+0x04]
pop ecx
```

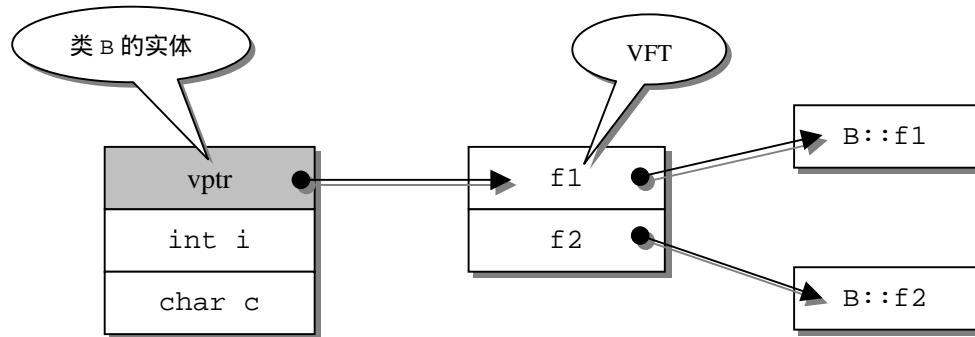


图 3 C++类实体的内存布局

第一句 `ebp-0x04` 是指针变量 `p` 的地址，第一句是把 `p` 所指向的对象的地址传送到 `eax`；
 第二句不用管它；
 第三句是把对象头部的指针 `vptr` 传到 `edx`，即已取得 `VFT` 的入口；
 第四句是关键，`edx` 再加 4（32 位系统上一个指针占 4 个字节），也就是调用了从 `VFT` 入口算起的第二个函数指针，即 `B::f2`；
 第五句不用管它。

相信大家对 `VFT` 和 C++的对像模型有一个更深刻的认识吧？对于 `VFT` 的实现，各个编译器是不一样的。有兴趣的朋友不妨可以自行探索一下 Microsoft Visual C++和 GCC 的实现方法，比较一下它们的异同。

知道了 `VFT` 的结构，那么想想下面这个程序的结果是什么。

```
#include <iostream>
using namespace std;

class A
{
    int c;
    virtual void f();
}
```

```

public:
    A(int v = 0) { c = v; }
};

void main()
{
    A a, b(20);
    cout<<*(void**)&a<<endl;
    cout<<*(void**)&b<<endl;
}

```

我想您应该能理解其中`*(void**)&a` 吧？这是取得 vptr 的值，也就是 `a` 所在内存空间的前 4 个字节，一个指针。下面我们还会使用类似的语句。

无庸质疑，结果是输出两个完全相同的值。前面我们已经说过，对于同一个类的所有对象，其 vptr 值都是相同的。

那么这个 VFT 到底有什么作用呢？现在看来，似乎就是储存虚函数的地址。

虚拟方法表 VMT

如何通过类的实体来找到类的相关 RTTI 信息呢？显然，VFT 是同一个类的所有实体共享的数据，而 RTTI 正好也是。那么，把 RTTI 放在 VFT 里，就是个不错的选择。

往哪儿放呢？VFT 从入口开始往后是各个虚函数的指针，那么 RTTI 只能放在两个地方：入口以前或者所有虚函数指针之后。显然，放在入口以前更好，至少我们不用关心虚函数的多少，RTTI 的位置也可以相对确定。

VCL 就采用了这个办法来放置 RTTI，不过把 VFT 换了名字，叫虚拟方法表 VMT（Virtual Method Table）。VMT 的结构是怎样的呢？Borland 所提供的帮助文件里没有任何相关资料，不过我们在 `Include\Vcl\system.hpp` 中就能找到如下蛛丝马迹。

```

static const Shortint vmtSelfPtr = 0xffffffffb4;
static const Shortint vmtIntfTable = 0xffffffffb8;
static const Shortint vmtAutoTable = 0xffffffffbc;
static const Shortint vmtInitTable = 0xffffffffc0;
static const Shortint vmtTypeInfo = 0xffffffffc4;
static const Shortint vmtFieldTable = 0xffffffffc8;
static const Shortint vmtMethodTable = 0xffffffffcc;
static const Shortint vmtDynamicTable = 0xffffffffd0;
static const Shortint vmtClassName = 0xffffffffd4;
static const Shortint vmtInstanceState = 0xffffffffd8;
static const Shortint vmtParent = 0xffffffffdc;

```

```

static const Shortint vmtSafeCallException = 0xfffffffffe0;
static const Shortint vmtAfterConstruction = 0xfffffffffe4;
static const Shortint vmtBeforeDestruction = 0xfffffffffe8;
static const Shortint vmtDispatch = 0xfffffffffec;
static const Shortint vmtDefaultHandler = 0xffffffffff0;
static const Shortint vmtNewInstance = 0xfffffffff4;
static const Shortint vmtFreeInstance = 0xfffffffff8;
static const Shortint vmtDestroy = 0xfffffffffc;

```

注意这些常数值中的负数采用的是补码表示法。求一个负数的补码，先写出相应正数的补码表示，再按位求反，最后(在最低位)加1即可。对于求32位负数的补码，也可以用它本身减去0xffffffff再减1即可。以0xffffffffc为例， $0xffffffffc - 0xffffffff - 1 = -0x04$ ，这就是结果。我们还可以从Borland提供的原始码Source\Vcl\system.pas获得，其中就是用负数表示。

看着这份表格，从这些变量名中，我们已经猜到了其大概的分布情况。这些数字之间的间隔都是4，可以猜想这些都是指针：函数指针或者数据指针。从这些常数的名字我们就可以知道它们的作用，比如vmtClassName自然就是储存类名的指针。入口0以前，就是VCL对象的关键数据。无疑，它们蕴涵了TObject乃至VCL对象关键的秘密，也就是VMT的分布结构。

这以上只是我们的推测，我们还应该验证一下。我们知道的事实是，每一个对象必然都包含了其所属类的相关信息。比如任何一个C++类的实体，都包含一个指向虚拟函数表VFT的指针。VCL类的实体必然也包含一个指向虚拟方法表VMT的指针。

```

#include <vcl.h>
#include <iostream>
using namespace std;

class A: public TObject
{
    int x;
    virtual void f1() {}
    virtual void f2() {}

public:
    A(int v = 0): x(v) {}

};

void main()
{
    A* p = new A;, * q = new A(100);
    void* a = *(void** )p, * b = *(void** )q;
    void* c = p->ClassType(), * d = q->ClassType();
    cout<<a<< ' '<<b<<endl;
    cout<<c<< ' '<<d<<endl;
}

```

```

cout<<__classid(A)<<endl;
delete p;
delete q;
}

```

结果很有意思，输出的五个指针地址完全一样！`a` 和 `b` 相同，从前面的例子我们就可以知道。然而 `TObject` 的 `ClassType` 方法和 `__classid` 操作符的返回值也跟这两者相同，这就有点意思了。查查帮助就可以知道，`__classid` 是 C++ Builder 中新增的扩展关键字，返回类的 VMT 的入口地址；而 `TObject` 的 `ClassType` 方法则是返回对象的类信息，返回类型是 `TClass`（也就是 `TMetaClass*`）。这说明，每个 VCL 类实体的头部包含的指针，就是指向 VMT 的入口地址。而这个位置，也就是 `TObject` 的成员函数 `ClassType` 的返回值，亦即运算符 `__classid` 返回的类 A 的信息，只不过这个返回值是以 `TClass`（即 `TMetaClass*`）的形式存在。

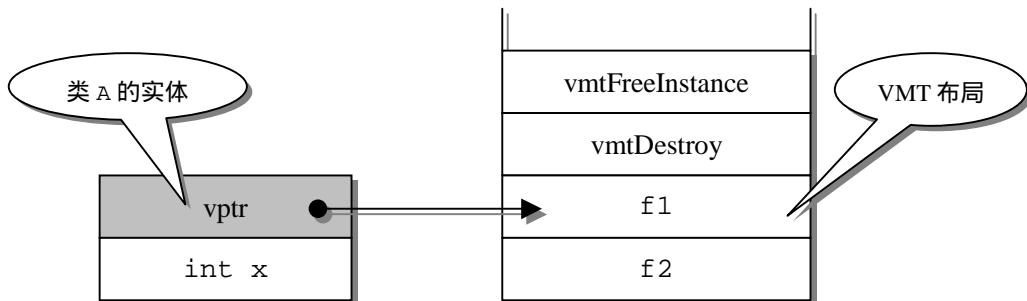


图 4 VCL 类的 VMT 入口

我们已经知道了 VMT 的结构，现在又找到了其入口，此时的兴奋不亚于阿里巴巴知道“芝麻，开门吧”这句咒语时的感受。既然知道了开门的咒语，还不赶快进去拿宝藏？

牛刀小试

乘着东风，我们来模拟一下 VCL 简单的 RTTI 功能。为方便起见，我们仿造 `TObject`，写一个类 `FObject`（呵呵，如果把 `TObject` 看成 True Object，我们的 `FObject` 就是 False Object）。要问下面这段代码从哪里来？大部分都 Copy&Paste 自 `Include\Vcl\systobj.h` 文件。

```

class FObject
{
public:
    FObject(); /* Body provided by VCL {} */
    Free();
    TClass     ClassType();
    void      CleanupInstance();
    void *    FieldAddress(const ShortString &Name);

/* class method */

```

```
static TObject * InitInstance(TClass cls, void *instance);
static ShortString ClassName(TClass cls);
static bool ClassNameIs(TClass cls, const AnsiString string);
static TClass ClassParent(TClass cls);
static void * ClassInfo(TClass cls);
static long InstanceSize(TClass cls);
static bool InheritsFrom(TClass cls, TClass aClass);
static void * MethodAddress(TClass cls, const ShortString &Name);
static ShortString MethodName(TClass cls, void *Address);

/* Hack: GetInterface is an untyped out object parameter and
 * so is mangled as a void*. In practice, however, it is
 * really a void**. Be sure when using this method to provide
 * two levels of indirection and cast away one of them.
 */

bool GetInterface(const TGUID &IID, /* out */ void *Obj);

/* class method */
static PInterfaceEntry GetInterfaceEntry(const TGUID IID);
static PInterfaceTable * GetInterfaceTable(void);

ShortString ClassName()
{
    return ClassName(ClassType());
}

bool ClassNameIs(const AnsiString string)
{
    return ClassNameIs(ClassType(), string);
}

TClass ClassParent()
{
    return ClassParent(ClassType());
}

void * ClassInfo()
{
    return ClassInfo(ClassType());
}

long InstanceSize()
```

```
{  
    return InstanceSize(ClassType());  
}  
  
bool InheritsFrom(TClass aClass)  
{  
    return InheritsFrom(ClassType(), aClass);  
}  
  
void * MethodAddress(const ShortString &Name)  
{  
    return MethodAddress(ClassType(), Name);  
}  
  
ShortString MethodName(void *Address)  
{  
    return MethodName(ClassType(), Address);  
}  
  
virtual HResult SafeCallException(TObject *, void *);  
virtual void AfterConstruction();  
virtual void BeforeDestruction();  
virtual void Dispatch(void *Message);  
virtual void DefaultHandler(void* Message);  
  
private:  
    virtual TObject* NewInstance(TClass cls);  
  
public:  
    virtual void FreeInstance();  
    virtual ~TObject(); /* Body provided by VCL {} */  
};
```

当然 FObject::ClassType 我们已经会写了，那就是

```
TClass FObject::ClassType()  
{  
    return *(TClass*)this;  
}
```

我们会在后面陆续把这些成员函数填充完整。先举个例，拿类名 (ClassName) 开刀吧。

查查 VMT 表，`vmtClassName = 0xfffffd4`，我们就从这里下手。主要的步骤是：

- 找到 VMT 的入口；
- 通过 `vmtClassName` 找到储存类名的地址；
- 获取类名。

`0xfffffd4` 也就相当于 `-44`，也就是 VMT 入口指向的地址开始，倒数第 44 字节到倒数第 41 字节这 4 个字节所代表的指针，指向类名。假设入口指向的地址是 `cls`，那么 `vmtClassName` 所代表的地址就是 `(char*)cls - 44`，亦即 `(char*)cls + vmtClassName`。

注意一个字符串格式的问题，VCL 既然是用 Object Pascal 写的，其中储存类名的字符串的格式必然是 Pascal 传统方式，也就是第 1 个字节为字符串的长度，紧接着为字符串的实际内容。在 C++ Builder 中，与之对应的类型是 `ShortString`。

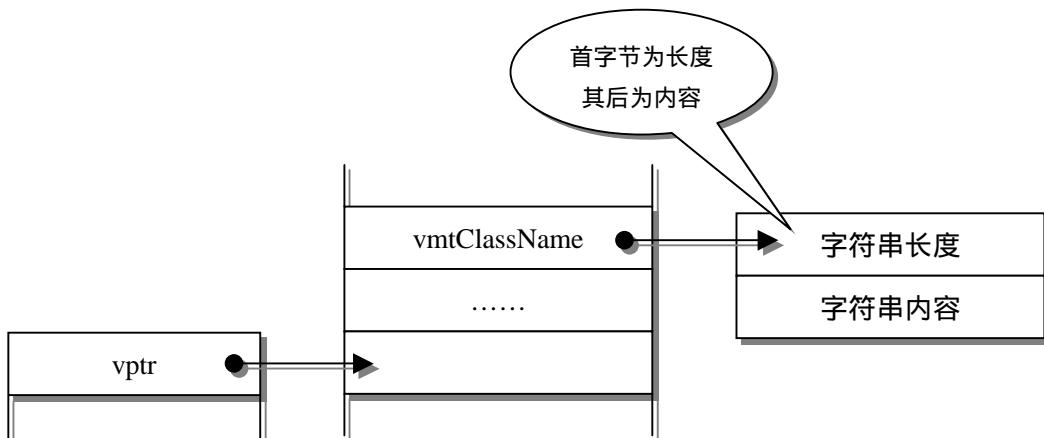


图 5 `TObject::ClassName` 的运作方式

代码如下：

```
ShortString FObject::ClassName(TClass cls)
{
    ShortString* r = *(ShortString**)((char*)cls + vmtClassName);
    return *r;
}
```

我们不妨测试一下。

```
#include <vcl.h>
#include <memory>
#include <iostream>
using namespace std;
... 插入 FObject 相应的代码...
void main()
```

```

{
    auto_ptr<TList> list(new TList);
    FObject* p = (FObject*)list.get();
    cout<<AnsiString(p->ClassName()).c_str()<<endl;
    cout<<AnsiString(list->ClassName()).c_str()<<endl;
}

```

输出结果在我们意料之内，都是“TList”。

对于函数 `ClassNameIs`，我们就可以轻而易举地完成了。

```

bool FObject::ClassNameIs(TClass cls, const AnsiString string)
{
    return string==ClassName(cls);
}

```

有朋友可能奇怪，你怎么知道 `TObject::ClassName` 是这样的呢？

三种办法：

- 猜，用经验推测；
- 看 Borland 提供的原始码；
- 看编译以后的汇编码。

在 Borland 提供的原始码中，我们可以看到 `TObject::ClassName` 的实现如下：

```

class function TObject.ClassName: ShortString;
asm
    { -> EAX VMT }
    { EDX Pointer to result string }
    PUSH ESI
    PUSH EDI
    MOV EDI, EDX
    MOV ESI, [EAX].vmtClassName
    XOR ECX, ECX
    MOV CL, [ESI]
    INC ECX
    REP MOVSB
    POP EDI
    POP ESI
end;

```

熟悉汇编的朋友就可以由此写出相应的 C/C++代码来。对于不会的朋友，根据我们的讲解，相信也可以轻而易举地完成吧。

希望您在看这段的时候，不妨先用第1种办法，然后结合2、3看看，一定收获不小。

势如破竹

接下来就太简单了，我们不再举例，把相应的成员函数补充完整即可。您不妨先自己试着写写，探索一下，再与汇编代码和文中的代码作比较，一定乐趣无穷。

`TObject::ClassInfo` 是做什么的？问我啊？我也不知道。VCL 的帮助里说，用 `ClassInfo` 可以访问包含对象类型、祖先类和所有 `published` 属性信息的 RTTI 表。这个表只是内部使用，`TObject` 提供了其它方法来访问 RTTI 信息。我们先写出它的实现。

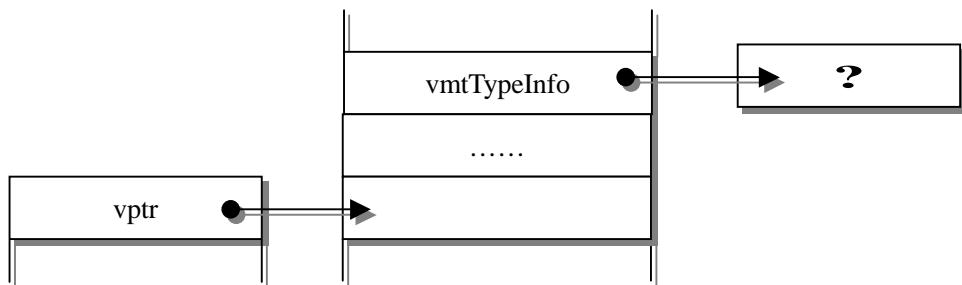


图 6 `TObject::ClassInfo` 的运作方式

```
void * FObject::ClassInfo(TClass cls)
{
    return *(void**)((char*)cls + vmtTypeInfo);
}
```

Borland 的说法可信吗？这个函数返回值的类型是 `void *`，明摆着不愿意透露更多的信息。您不妨按上面 `ClassName` 的方法测试一下，对于 `TList`，`ClassInfo` 输出的结果居然是 0！也就是一个空指针！什么东东？别急，后面我们会掀开这个 `void *` 的面纱，现在姑且卖个关子。

VCL 框架中只存在单继承，这是由 Object Pascal 语言的特性决定的。这样，每一个类只有唯一一个父类，函数 `TObject::ClassParent` 就能帮您把父类找出来。

```
TClass FObject::ClassParent(TClass cls)
{
    TClass* r = *(TClass**)((char*)cls + vmtParent);
    return (r)? (*r) : 0;
}
```

由此，我们也能很轻松地模拟 `TObject::InheritsForm` 的实现。

```
bool FObject::InheritsFrom(TClass cls, TClass aClass)
{
    while(aClass)
    {
        if(aClass==cls) return true;
        cls = ClassParent(cls);
    }
    return false;
}
```

要知道一个对象所占的字节数，`TObject::InstanceSize` 就可以达到目的。

```
long FObject::InstanceSize(TClass cls)
{
    return *(long*)((char*)cls + vmtInstanceSize);
}
```

有朋友可能说，C++不是有 `sizeof` 操作符吗？为什么不用呢？在 VCL 中，`sizeof` 有两个缺陷。首先 `sizeof` 是完全静态的，也就是说，如果您写 `sizeof(...)`，编译以后，这会被替换为一个常数，没有任何的求值过程，因此不能动态求值；其次，VCL 类必须与指针或引用的形式存在。所以对于

```
TObject *a;  
...
```

`sizeof(*a)` 这样的表达式是错误的。而且即使 `TObject` 不是 VCL 类，使用 `sizeof(*a)` 还是相当于 `sizeof(TObject)`，没有实际价值。

结束语

现在我们已经打开了通向 VCL 类秘密的大门。回头一看，VMT 跟 VFT 有什么区别与联系呢？其实 VMT 可以算是 VFT 具体化的一个结果，也就是说，VMT 是在 VFT 基础上发展出来的一种具有“规范”性质的结构，所有的 VCL 类都遵循这个“规范”。这很像 COM 与 C++纯虚基类的关系。

通过 VMT，VCL 放置了一些重要的信息，由此来实现 RTTI。所以“高级”RTTI 功能其实是相当低级和简单的一项技术。就其实现方式而言，大致有三种。MFC 用宏（macro）模拟算是一类，完全符合 C++ 标准，不需要对语言进行扩充，也不依赖于特定的编译器，不过给人臃肿的感觉；VCL 则是完全由编译器实现，同时扩充了 C++ 的语言特性，必须在 Borland 的编译器上编译，但是很简洁；另外建构 KDE 基础的跨平台框架 Qt⁴，则采用了折中的方式，扩充了 C++ 的关键字，书写很简洁，在编译之前必须用 Qt 提供的程序 MOC 进行预处理，把扩充部分的代码改写为符合 C++ 标准的代码，然后才可以在任何符合 C++ 标准的编译器上编译。

代表作	实现方式	不依赖特定编译器	简洁程度	编译次数
MFC	宏插入	是	一般	1
VCL	编译器生成	否	好	1
Qt	预编译程序生成	是	较好	2 (包括 MOC)

注：如果长期仅在 Windows 平台下进行开发的朋友，可能没有听说过 Qt 的大名。事实上在 Linux 世界里，这可是个响当当的名头。Qt 是一套完善的 C++ 框架，横跨 Unix/Linux、Windows、Mac OS 诸多平台，内部机制相当有趣。Borland 最新的 Kylix 和 Delphi6 所采用的跨平台框架 CLX（分为 BaseCLX、VisualCLX、DataCLX、NetCLX 四个部分，BaseCLX 与 VCL 顶部几个类相同），其可视化部分 VisualCLX 就建构在 Qt 上，这多少让我感到失望和不满。Qt 本身就跨平台，VisualCLX 建构在 Qt 上，自然也跨平台，但是 CLX 是用 Object Pascal 包装了一个 C++ 框架 我不敢想象，C++ Builder6 中的 CLX 是否又用 C++ 再来包装这个包装了 C++ 框架的 Object Pascal 框架呢？如果真是如此，其效率和调试难度……

对于“高级”RTTI 的实现形式，VCL 用了 TMetaClass（其中 TClass 就是 TMetaClass*）来配合存储类的信息，也就是所谓“类的类”，这非常普遍。MFC 中的 CObject 与 CRuntimeClass，JDK 中的 java.lang.Object 和 java.lang.Class 都是如此。比如对于一个 TObject *p，如何获取其父类名呢？我们必须借助 TMetaClass：可以先用 p->ClassType 返回父类的信息（是一个 TMetaClass* 类型），再以此为参数传入 TObject::ClassName 就可以获得结果，也就是 TObject::ClassName(p->ClassType()) 即可。

同时我们也应该拆穿所谓“拥有更高级 RTTI 的语言本身也更高级”的谎言。至少从我那少得可怜的经验来看，对于一套框架，除非需要和 IDE 配合，否则在绝大部分情况下，RTTI 是完全没有必要的，甚至是有害的⁵。希望使用和设计框架的朋友三思。

致谢

非常感谢孟岩和孙春阳对本文所提出的宝贵意见。

参考

1. Bjarne Stroustrup. *The C++ Programming Language*, 3e. Addison-Wesley, Reading, MA. 1997.
2. Stanley Lippman. *Inside the C++ Object Model*. Addison-Wesley, Reading, MA. 1996
侯捷 .《深度探索 C++ 物件模型》. 番峰资讯股份有限公司 . 1998 .
3. 侯捷 .《深度探索 C++ 对象模型》. 华中科技大学出版社 . 2001 .
4. 虫虫 .《Qt 最新消息》. C++ View . 2001 , 7 .
5. Robert C.Martin. “The Open-Closed Principle”. C++ Report. 1996, 1.
plliuly , 虫虫 .《开放封闭原则 OCP》. C++ View . 2001 , 8 .

H_{otline}

回音壁

大家好，我是 C++ View 的主编。C++ View 第 2 期发布后，收到不少朋友的 email，小编都一一回复了。其中一些有价值的意见和建议，特挑选出来放在这里，希望大家能发告诉小编自己的想法。

我觉得你们在人物介绍一栏里可以介绍一些优秀程序员的成长经历。比如他们是怎样走上程序员道路的，又是如何循序渐进的学习的，他们为什么选择了当前的研究方向。

怎么说呢，很成功的人，比如 Bill Gates，小编是高攀不上了。而且，软件的设计、包装、策划……都是大学问，我想不一定非得介绍程序员吧？《星闻》栏目会介绍各种各样的人物，比如 C++ 之父这样的大师，或者说不定哪天小编来个自吹自擂呢:-)

建议每期放上一篇适合初学者的文章，毕竟还有很大部分人刚刚接触 C++。可以向别的杂志学习，增加书评栏目。

C++ View 的定位相对高阶，而且国内适合初学者的资料恐怕非常多，我想 C++ View 适用于基础比较牢的“初学者”吧，呵呵。另外书评的栏目，本来一直有此计划，但是效果不尽如人意，我们做的也不会比别人更好，毕竟读万卷书是很难的事情，何必赶这趟浑水呢？

不过小编正在准备中，也许不久，我们就会推出一个名著剖析的专栏。

杂志尚待推广，毕竟有很多不象我那么幸运且渴望得到贵刊。

这个嘛……怎么推广呢？小编实在没有 money 去打广告啊，这实在需要各位读者朋友的大力支持啊。

你们的杂志做的好棒，没有废话，没有广告，免费，专业，纯技术性。我非常喜欢，但由于我刚开始学 C++，所以上面有很多内容对我来说还是有点难了。我很希望杂志上可以讲解一些基本的，常用的编程算法之类的东西，和其他一些适合初学者的东西。哈~不知道可不可以？还有就是我希望这本杂志永远都不要钱 !!! 哈哈哈~ !!! 不过我现在已经决定就是要钱我也要买了（不过还是不希望你们要钱~~）。

我终于知道评价杂志“棒”的“三无”原则了：无废话、无广告、无需付钱:-)

收费？难啊……这份杂志还是可以值几块钱吧？

算法的建议，这期的 C++ View 已经开设了相关的专栏，不过算法很适合初学者吗？*_*

为了方便大家，各位可以发 email 订阅 C++ View，我们将为您奉上 C++ View 最新的消息。在 email 中，请尽可能详细地告诉我们您的情况，以便我们根据读者的情况及时作出相应的调整。我们的 email 是：cppview@sohu.com。有任何消息，订阅了 C++ View 的朋友会在第一时间得到通知。

C++View

焦点：

Andrei Alexandrescu介绍

精彩推荐：

- 天方夜谭VCL：多态
- 模式罗汉拳：Interface(接口)
- 虚拟函数是否应该仅被声明为private/protected?

C++View

第4期

return

第4期目录 2001年10月

焦点 Focus

Andrei Alexandrescu 介绍	03
回音壁	59

特稿 Feature

虚函数是否应该仅被声明为 private/protected	04
--------------------------------	----

专栏 Column

Generic<Programming>	
类型和数值间的映射	09

设计笔记	
依赖倒置原则 DIP	23

C++批评系列	
函数重载	20

Pattern Hatching	
弃儿、养子和替身	34

模式罗汉拳	
Interface (接口) 模式	42

天方夜谭 VCL	
多态	47

导读

我想我要是再不把 C++ View 第 4 期交出来，虫虫真要跟我急了。想他原本出于对朋友的信任，委托我在他开学无暇之际接手第 4 期的编务工作，每次他向我询问杂志进展，我都一次比一次多一些愧疚地说“就快差不多了”，没料到“差不多”的一点让他等的竟是如此漫长，连热心的读者朋友都等得不耐烦了，何况他这个主编呢。说来的确惭愧，虫虫将文章都已经约好了，封面也准备了，前几期的版面模板都给我了，我几乎什么都不需要做就可以过一把编辑瘾。尽管我也努力了，但工作上的事情几乎占用了我所有的时间，加上我做事效率低下，最终使得我的这份“答卷”确实有些让人失望。

说这些当然不是想把我向虫虫的“忏悔书”贴在这儿作导读。一来我是向大家解释第 4 期 C++ View 为何姗姗来迟，并向关心我们杂志的朋友真诚地道歉；二来，我想以我的经历顺便告诉读者朋友们，要做好一期这样的杂志的确不容易，它涉及更多的不是技术上的工作，而是一些诸如翻译的遣词用句、字体格式、排版效果的细节小事。尽管这是一份免费杂志，但虫虫做事的认真和要求的严格令我不敢怠慢(尽管结果并不令人满意)。这份杂志虽然是虫虫一手创办的，但是我觉得它是我们所有喜欢 C++ 和程序设计的人所组成的整个社区的，因此，我希望大家能够共同出力来做好这份杂志，毕竟一两个人的精力和时间是有限的。我真心地期待有能力的朋友毛遂自荐，为这份杂志作一些具体的事情。

文章结束之前不能全是废话，我得向大家介绍完这期的文章才算最后完成虫虫交给我的任务：这期文章不多，但不乏精彩之篇。虫虫亲自捉笔的“天方夜谭 VCL”第二篇给我们讲述 VCL 中的多态的实现，文笔很好；另外一篇原创专栏“模式罗汉拳”的主人透明这次为我们讲解“接口”这个看似简单实则“玄机重重”的模式；yefeng 继续翻译 Andrei 的 Generic<Programming> 系列，模板发烧友不可错过；本期翻译的 John Vlissides 的“Pattern Hatching”第二篇用的名字颇为吸引人，内容当然也是可圈可点；还有译自 Robert Martin 的“设计笔记”系列的 DIP 原则的介绍也许会让你有所感悟。

主 编：王 曦 主页：<http://cppview.yeah.net>
 封面设计：棒棒虎 <http://cppview.y365.com>
 本期编辑：plpliuly 电邮：cppview@sohu.com

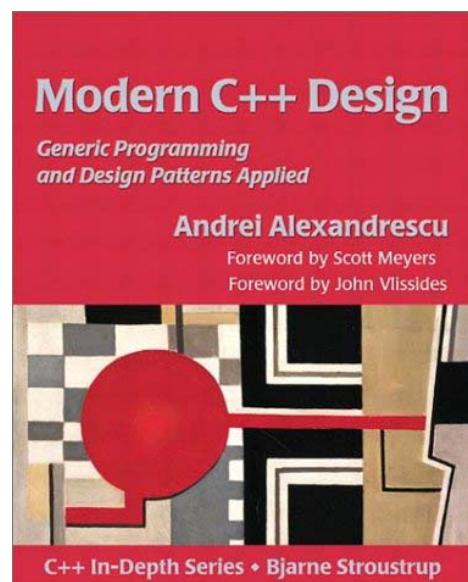
Andrei Alexandrescu 介绍



Andrei Alexandrescu 博士毕业于座落于华盛顿州西雅图的华盛顿大学，并在那儿获得了 Ph.D. 学位。他是泛型编程方面的重量级作品 *Modern C++ Design: Generic Programming and Design Patterns Applied* 一书的作者。该书在 2001 年 3 月由美国 Addison-Wesley 出版，中文繁体版和简体版都在翻译过程之中（繁体版由台湾著名技术作家侯捷先生和大陆的孟岩先生合译，简体版正由清华出版社委托北京大学潘爱民先生翻译）。

Andrei Alexandrescu 是 C/C++ User Journal 杂志的专栏作家。他经常在世界各地的学术会议上发表论文演讲，比如 2001 年春天在英国举行的 ACCU (the Association of C&C++ Users) 大会，在美国举行的 OOPSLA (ACM Conference on Object Oriented Programming, System, Language, and Application) 2000，在意大利举行的 Extreme Programming 2000 会议。Andrei 也是 C++ 研讨会 (The C++ Seminar) 的主要授课人之一，其他的主要授课人包括 Scott Meyers、Herb Sutter、Dan Saks、Steve Ehrhart 等。

关于 *Modern C++ Design: Generic Programming and Design Patterns Applied* 一书，*Design Patterns* (《设计模式》) 的作者之一 John Vissides 在他为此书作的序中写到：该书所描述的内容融会了泛型程序设计、template metaprogramming、OOP、设计模式等诸多程序设计技术。它为 C++ 的程序设计展现了一道崭新的风景线。无论是对编程，还是对软件设计本身，甚至是软件分析和架构，都带来了崭新的理念。



虚拟函数是否应该仅被声明为 private/protected ?

Cber

问题导入

我想对于大家来说，虚拟函数并不能算是个陌生的概念吧。至于怎么样使用它，大部分人都会告诉我：通过在子类中重写(override)基类中的虚拟函数，就可以达到 OO 中的一个重要特性——多态(polymorphism)。不错，虚拟函数的作用也正是如此。但如果我要你说一说虚拟函数被声明为 public 和被声明为 private/protected 之间的区别的话，你又是否还能象先前一样肯定地告诉我答案呢？

其实在一开始，我和大家一样，都喜欢把虚拟函数声明为 public（我并没有做太多的调查就说了这些，因为我身边的程序员们几乎都是这样做的）。这样做的好处很明显：我们可以轻而易举地在客户端（client，相对于 server，server 指的是我们所使用的继承体系架构，client 指的就是调用该体系中方法/函数的外部代码）调用它，而不是通过利用那些烦人的 using 声明，或是强加给类的 friend 关系来满足编译器的 access 需求。OK，这是一个很不错的做法，简单、并且还能达到我们的要求。

但根据 OO 三大特性中的另一个特性——封装(encapsulation)来说（另一个就是继承），需要我们将界面(interface)与实作(implementation)分开，即向外表现为一个通用的界面，而把实作上的细节封装在模块内不让 client 端知晓。界面与实作的分离，使得我们得以设计出耦合度更低、扩展性更好的系统来，并且还可以从这样的系统中提取出更多的可重用(reusable)的设计。

对于 OO 来说，封装是它的头等大事，享有最高的权利，其他的设计如果和它有着冲突，则以符合它的设计为准。这样，问题就出来了，万一我们所希望出现的多态正好是具体的实作细节并且我们不希望把它暴露给 client 端的话，那我们应该怎么样改动我们的设计以使得它能够适应封装的需求呢？

可行的解决办法

幸好，C++中不但支持 public 的虚拟函数，也有着 private/protected 虚拟函数（在此我不想对于 public 和 private/protected 之间的区别多说）。前者是我们常用的形式，我也不多说，我们在此主要关心的是 private/protected 的虚拟函数。

你可能会有疑惑，既然虚拟函数被声明为 private (protected 不算，因为子类可以直接访问基类的 protected 成员)，那子类中怎么还能对它进行重写呢？在此，你的疑虑是多余的，C++标准（也称 ISO 14882）告诉我们，虚拟函数的重写与它的具体存储权限没有任何关系，即便是声明为 private

的虚拟函数，在子类中我们也同样可以重写它。因此，碰到上面所说的问题，我们就可以得到如下的设计：

```
class Base {  
public:  
    void do_something()  
    {  
        //.....  
        really_do_something();  
        //.....  
    }  
private:  
    virtual void really_do_something()  
    {  
        //do the polymorphism code here  
    }  
};  
  
class Derived: public Base {  
private:  
    void really_do_something()  
    {  
        //do the polymorphism code here  
    }  
};
```

如果我们需要从上面的设计中得到实际上的多态行为，只要象下面一样调用 `do_something` 就可以了：

```
//client code  
Base& b;           //or Base* pb;  
b.do_something();  //or pb->do_something();
```

这样我们就得以解决了在开始处提出的那个问题。

问题引申

那就这样完了吗？没有。相反，至此我们才开始进行我们今天的讨论。首先让我们来看看多态的实现：

```
void Base::do_something()
```

```
{  
    //.....  
    really_do_something();  
    //.....  
}
```

我们可以发现，在调用真正对多态有贡献的 `really_do_something()` 之前及调用后，我们还可以在其中添加我们自身的代码（如一些控制代码等），这样我们“好像”就可以轻而易举地实现了 Bertrand Meyers 所提出的“Design By Contract” (DBC)¹了：

```
void Base::do_something()  
{  
    //our precondition code here  
    really_do_something();  
    //our postcondition code here  
}
```

然后，让我们去看看 Template Method 这个 Pattern²，发现所谓的 Template Method 也主要就是通过这种方式来进行的。于是，我们是否可以这么想呢：将所有的虚拟函数都声明为 `private/protected`，然后再使用一个 `public` 的非虚拟函数调用它，这样，我们不就得到了上面所列出的所有好处吗？

详细分析

简单看来，好像那么做真的是好处大大的，既不会造成效率上的损失（通过将该 `public` 的非虚拟函数 `inline` 化，简单的函数转调用的开销就可以被消除掉），又能够获得上述所有的好处。何乐而不为呢？

实际上来看，有不少程序员也正是这么做的（Herb Sutter 所调查的结果表明，这里面甚至还包括那些实作标准函数库的程序员们，当然，他们所考虑到的使用这种技巧的理由不会仅仅是我下面所给出的其他人的理由^_^）。有的人甚至还认为，虚拟函数就应该被声明为 `private/protected`（当然，虚拟的析构函数不能够算在其中之列，否则就会有大乱子了）。

但让我们再仔细地考虑一下，想想一些比较极端的例子。假设我们有一个类，它拥有的虚拟函数的个数非常之多（就算它 10000 个吧），那即使大多数情况下只是简单的函数转调用动作，我们是否还应该为它的每一个虚拟函数都提供一个公开的非虚拟的界面呢？这时，为你的程序提供一个接口类(即没有任何成员变量，所有的方法都是纯虚函数的类)是一个不错的解决方案。

¹ *Object-Oriented Software Construction*. Chapter 11: Design by Contract: building reliable software, 影印版国内有售。

² *Design Patterns: Elements Of Reusable Object-Oriented Software*, 中文翻译版国内有售。

还有，因为这样做的结果将会是：基类中的那个 `public` 的非虚拟界面函数必须能够适合所有的子类的情况，这样，我们将所有的责任都推倒基类上去了，这不能算是一个好的设计方法。假设我们有了一个继承体系极深的架构，在对基类进行了多次继承后，我们突然发现，新的子类已经无法适应原有的那个界面了。于是，为了继续执行我们的虚拟函数 `private` 化，我们就将不得不把基类的代码给翻出来并改正它。幸运点的是，基类的代码是我们可以得到的，这样我们最起码还是有机会改正的（虽然有的时候，我们已经无法看懂基类中的代码了）；糟糕的是，我们的基类是通过我们使用的一个函数库中得到的，而该函数库的代码我们无法获得，这个时候我们该怎么办呢？由此可见，如果在设计可能会被进行深度继承的类继承体系架构时，要想继续使用 `private` 的虚拟函数的话，对于设计基类的要求就将会变的非常之高（因为在以后，基类的任何小小改动造成的后果传递到了继承的低端时都将被显著的放大），而让设计人员去猜测以后所有的可能使用情况是件不现实的事情，这样也就容易产生脆弱的、需要被频繁改动的设计。请记住一点：FBC(Fragile Base Class)是一件可怕的事情，在我们的程序中应当避免出现这种情况。

另外，在你决定把你程序中的虚拟函数改为 `private/protected` 前，你有没有一个很好的理由呢？如果你只是说：“哦，我不知道，不过这样做可能会在以后的某天产生作用”。不错，时刻让自己的程序保持可扩展性是很好的一件事情，但那都是基于你可以预见未来的扩展之上的(这种预见主要来自于你对于该领域的深刻认识或是你平时的经验)。在没有任何理由的情况下，仅仅靠着一句“它以后可能会有用”就往自己的程序中添加进去某种特性听起来好像很炫，但实际上它可能对你的程序有百害而无一利。在我们现有的各种 Framework 中，有着很多类似的“以后可能会有用”的特性，结果最终都被证明为没有被使用到，这不能不说这是对于开发工作的一种浪费。因此，还是让我们记住在 XP³ 中所说的 YNGNI(You Never Going to Need It)，对于现阶段没有用到的特性，还是不要提供为好。不过，如果你能够预见到以后的扩展的话，还是请你为它留下一个可扩展的便利。

此外，基于编译器的角度来看，当你一旦改动了基类，那么所需要重新编译的就不仅仅是基类本身了，所有从该基类继承下来的派生类也都将被重新编译。这样，我们就不得不又浪费掉大量的编译时间了。尤其是当我们决定大量使用 `inline` 的方式来转调用时，所需的时间就更加多了（因为 `inline` 函数在编译时会被扩展成实际的调用代码）。这也可以说是一种语法上的 FBC 问题。此外，当你决定向你的继承体系中增加一个函数，并改变了基类接口的行为，你就有可能破坏了整个继承体系，并使得外部的 client 端代码也受到了冲击。这种情况可以说是一种语义上的 FBC 问题。请记住：稳定的代码永远不要建立在不稳定的代码基础之上。

现在，再让我们回到Template Method上面来看。什么时候该使用TM呢？从Design Patterns中得到它的意图为：定义一个操作中的算法的骨架，而将一些步骤延迟到子类中。Template Method使得子类可以不改变一个算法的结构即可重定义该算法的某些特定步骤。这和我们所谈论的虚拟函数是不是应该为`private/protected`完全是不相干的，虽说在实现TM时我们会用到`private/protected`的虚拟函数，但并不是所有的`private/protected virtual`都为TM。

最后，完全使用 `private/protected virtual` 还有一个问题就是：OO 中所提倡的弹性。我们知道，OO 中的弹性通常都是由继承中的多态提供的，但有时我们也会使用组合中的委托。实际上已经有很多的 Patterns 都是这么做的了，如：Proxy, Adapter, Bridge, Decorator 等。如果一味地追求

³ Extreme Programming，一种轻量级的软件开发方式，注重开发中的灵活性，测试及其他……可以从下面网站上得到有关它的更多信息：www.extremeprogramming.org。

private/protected virtual, 势必使得我们只能在程序中使用继承了, 为了一棵树而放弃一片森林的事情, 我想大家也都不愿意做吧。

结论

说了半天, 我也该收工了:-)现在开始进行我观点的归纳:

一般说来, 把虚拟函数声明为 private/protected 是一个很不错的办法⁴, 但如果一旦把它作为一个唯一的 Silver Bullet 来使用的话, 就会产生许许多多的问题。在这篇文章中我也只是大概的谈了其中的部分, 还有其他的一部分内容由于现今还没有完全整理好, 也就不多说了。希望能够在下次再把它完善掉。

参考资料

1. *Object-Oriented Software Construction*, Second Edition, Bertrand Meyer, 清华大学出版社出版 (影印版)
2. 设计模式可复用面向对象软件的基础, GoF, 李英军等译, 机械工业出版社出版
3. Conversations: Virtually Yours, Herb Sutter & Jim Hyslop, CUJ 以及网络上相关的资料

后记

写该文的最初冲动来源于 newsgroup: comp.lang.c++.moderated 上面的一个讨论: Virtual methods should only be private or protected? 在观看了 Kevlin Henney、Herb Sutter 以及 James Kanze 等几位大师的精彩言论后, 总想把自己的感受写下来。一开始, 我倒是写了很多, 但没有完全写完。近来由于比较忙的情况, 因此也就慢慢地把此事差点给忘记了。不是虫虫催着我要稿的话, 我想也不知道要到什么时候我才能把它给写完:(, 即便是现在, 由于很久没有复习这些资料, 很多的东西也没能写进去, 如果大家觉得意犹未尽的话, 可以直接到 newsgroup 中找到该 thread, 里面有着完整的讨论内容。

⁴ 可以参见于 Herb Sutter 和 Jim Hyslop 发表的 Conversations: Virtually Yours 一文, 在 CUJ 站点上可以找到这篇文章。

Generic<Programming>: 类型和数值间的映射

Andrei Alexandrescu

ye_feng 译

编者按：本文是Andrei Alexandrescu的Generic<Programming>系列文章之一，原文可以在<http://www.cuj.com/experts/1810/alexandr.htm?topic=experts>找到

在C++里，“转换（conversion）”一词描述了从一种类型的值取得另一种类型的值的过程。然而，有时候你会需要另一种转换：你可能需要从一种类型里取得一个数值，或者相反。这样的转换在C++里不太自然，因为类型和数值之间有一条难以逾越的鸿沟。但是在一些特定的情况下，我们需要突破这两者之间的界限，这篇文章将讨论怎么做。

从整数映射到类型

对很多泛型编程惯用法来说，有一个很有用的模板，它却出人意料地简单：

```
template <int v>
struct Int2Type
{
    enum { value = v };
};
```

对于每一个不同的常整数参数，`Int2Type`会“生成”一个不同的类型。因为不同的模板实例是不同的类型，所以`Int2Type<0>`和`Int2Type<1>`等是不同的。而且，产生这个类型的值被“保存”在枚举成员`value`里。

我们可以方便地使用`Int2Type`来“类型化”某个常整数。比如说，我们设计了一个类模板`NiftyContainer`：

```
template <typename T>
class NiftyContainer
{
    ...
};
```

`NiftyContainer`保存了一个指向T的指针。在`NiftyContainer`的一些成员函数里，我们需要复制T类型的对象。如果T是非多态的类型，我们可以这样：

```
T* pSomeObj = ...;
```

```
T* pNewObj = new T(*pSomeObj);
```

对于多态的类型，那就要复杂一些。我们约定，所有用于NiftyContainer的多态类型都要定义虚函数clone，然后就可以这样复制对象：

```
T* pNewObj = pSomeObj->Clone();
```

因为容器要能够接受两种类型，所以我们必须实现上面两种复制算法，并且在编译时能够选择适当的一个。我们可以为NiftyContainer增加一个布尔类型的模板参数来表示T是否为一个多态类，靠程序员来传递正确的标志。

```
template <typename T, bool isPolymorphicWithClone>
class NiftyContainer
{
    ...
};

NiftyContainer<Widget, true> widgetBag;
NiftyContainer<double, false> numberBag;
```

如果NiftyContainer里的类型不是多态的，那么它的很多成员函数都可以优化，因为可以期望对象的大小是固定的，而且对象有确定的语义。在这些成员函数里，我们要根据模板参数isPolymorphic在两个算法中择其一。

乍一看，只需要用一个if就可以做到。

```
template <typename T, bool isPolymorphic>
class NiftyContainer
{
    ...
    void DoSomething(T* pObj)
    {
        if (isPolymorphic)
        {
            ... polymorphic algorithm ...
        }
        else
        {
            ... non-polymorphic algorithm ...
        }
    }
};
```

问题是，这样的代码不能编译。举例来说，如果多态的算法用到了`pObj->Clone`，那么对于没有定义成员函数`Clone`的类型，`NiftyContainer::DoSomething`都不能编译。编译时`if`的哪个分支会执行是很明显的，但是编译器可不管这些，它总会编译两个分支，尽管最后优化的时候会删除不会执行的死代码。当你调用`NiftyContainer<int, false>`的`DoSomething`时，编译器会停在调用`pObj->Clone`的地方，并且说：“嘿！”

还不止这些呢。即使`T`是多态类型，这段代码可能还是不能编译。假设`T`的拷贝构造函数被禁用了（通过把拷贝构造函数声明为`private`或者`protected`，一个设计良好的多态类应该这样做），那么针对非多态类的那个分支里的`new T(*pObj)`调用将编译失败。

如果编译器可以不去编译死代码就好了，但现实不是如此。那么什么是令人满意的解决方案呢？

结果是，有若干解决方案，用`Int2Type`可以提供一个特别干净利落的方法。`Int2Type`可以把布尔变量`isPolymorphic`的`true`和`false`两个值转换成两个不同的类型，然后我们就可以借助函数重载（overloading）使用`Int2Type<isPolymorphic>`了。瞧！

下面请惯用法“整数类型化”现身说法。

```
template <typename T, bool isPolymorphic>
class NiftyContainer
{
private:
    void DoSomething(T* pObj, Int2Type<true>)
    {
        ... polymorphic algorithm ...
    }
    void DoSomething(T* pObj, Int2Type<false>)
    {
        ... non-polymorphic algorithm ...
    }
public:
    void DoSomething(T* pObj)
    {
        DoSomething(pObj, Int2Type<isPolymorphic>());
    }
};
```

代码很简单，也达到目的了。`DoSomething`调用一个私有的重载函数。根据`isPolymorphic`的值，两个私有重载函数之一会被调用，完成分派。`Int2Type<isPolymorphic>`类型的临时变量根本没使用，它只是用来传递类型信息。

Not So Fast, Skywalker!

看了上面的例子，你可能会想，通过模板特化的种种技巧，也许还有更聪明的办法。为什么需要那个临时的哑变量呢？确实还有更好的办法。然而，令人惊奇的是，其他方法在简单性、一般性、有效性上，很难胜过Int2Type。

一种尝试是对于任何类型T和isPolymorphic的两个可能值特化NiftyContainer::DoSomething。这是模板部分特化（partial template specialization）的拿手好戏，对吧？

```
template <typename T>
void NiftyContainer<T, true>::DoSomething(T* pObj)
{
    ... polymorphic algorithm ...
}

template <typename T>
void NiftyContainer<T, false>::DoSomething(T* pObj)
{
    ... non-polymorphic algorithm ...
}
```

尽管上面的代码看上去非常漂亮，可这是不合法的，没有“部分特化类模板的一个成员函数”这么一回事。我们只能对NiftyContainer整体进行部分特化。

```
template <typename T>
class NiftyContainer<T, false>
{
    ... non-polymorphic NiftyContainer ...
};
```

你也可以整体特化DoSomething：

```
template <>
void NiftyContainer<int, false>::DoSomething(int* pObj)
{
    ... non-polymorphic algorithm ...
}
```

但是，很奇怪，你不能做介于两者之间的事。[\[1\]](#)

另一个解决办法是使用traits[\[2\]](#)，在NiftyContainer外面（traits类中）实现DoSomething。这可能是个很笨拙的方法，因为它把DoSomething的部分实现隔离开来了。

第三个方法还是用traits，通过在NiftyContainer内部定义私有的traits类，以求把所有东西都放在一起。长话短说，这个方法是可行的，但是当你最终实现它的时候，你就会认识到Int2Type方法有多好了。Int2Type方法最大的优点是，你可以把小巧的Int2Type模板放在你的类库里，并且在文档中注明它的用法。

类型到类型的映射

考虑下面的函数：

```
template <class T, class U>
T* Create(const U& arg)
{
    return new T(arg);
}
```

Create把参数传递给构造函数，以创建一个新的对象。

假设在你的应用程序里有一条规则：Widget类型是以前写的，其对象在构造的时候需要两个参数，第二个参数需要传固定的值-1。所有Widget的派生类却不需要这样做。

怎样对Create进行特化，使它特殊照顾Widget类型呢？你不能对部分特化模板函数，就是说不能这样干：

```
// Illegal code - don't try this at home
template <class U>
Widget* Create<Widget, U>(const U& arg)
{
    return new Widget(arg, -1);
}
```

因为C++没有函数部分特化功能，我们唯一可用的手段还是重载。我们可以传递一个T类型的哑对象，靠重载来实现。

```
// An implementation of Create relying on overloading
template <class T, class U>
T* Create(const U& arg, T)
{
    return new T(arg);
}
```

```

template <class U>
Widget* Create(const U& arg, Widget)
{
    return new Widget(arg, -1);
}
// Use Create()
String* pStr = Create("Hello", String());
Widget* pW = Create(100, Widget());

```

Create的第二个参数只是为了选择适当的重载函数。这也是这个方法的一个麻烦：你在创建一个没有用的复杂对象上浪费了时间。即使优化可以帮点儿忙，如果Widget不提供或不允许调用缺省构造函数，那就没辙了。

著名的谚语“额外的中间层（extra level of indirection）”在这里也可以得到运用。一个想法是用T*而不是T作为模板参数。运行时，你总是传递空指针，空指针的创建代价是相当低的。

```

template <class T, class U>
T* Create(const U& arg, T*)
{
    return new T(arg);
}

template <class U>
Widget* Create(const U& arg, Widget*)
{
    return new Widget(arg, -1);
}
// Use Create()
String* pStr = Create("Hello", (String*)0);
Widget* pW = Create(100, (Widget*)0);

```

这个方法最多会让用Create的人感到费解。为了把这个方法作为一个惯用法固定下来，我们可以用一个和Int2Type类似的简单模板。

```

template <typename T>
struct Type2Type
{
    typedef T OriginalType;
};

```

现在可以写：

```

template <class T, class U>
T* Create(const U& arg, Type2Type<T>)

```

```
{  
    return new T(arg);  
}  
  
template <class U>  
Widget* Create(const U& arg, Type2Type<Widget>)  
{  
    return new Widget(arg, -1);  
}  
  
// Use Create()  
String* pStr = Create("Hello", Type2Type<String>());  
Widget* pW = Create(100, Type2Type<Widget>());
```

这样比原来的那个方法更容易说清楚。和Int2Type一样，你可以把Type2Type加到你的类库里去，并注明它的用法。

检测可转换性和继承关系

在实现模板函数和模板类时，经常会遇到一个问题：任意给定两个类型B和D，怎么才能知道D是不是从B继承下来的？

在编译时发现这样的关系，是在泛型库中实现高级优化的关键。在一个泛型函数中，如果这个类实现了某个特定的接口，我们就可以利用相应的优化算法，而不必对其进行dynamic_cast。

检测继承关系依赖于一个更一般的机制：检测可转换性（convertibility）。我们要一起解决的更一般性的问题是：如何检测一个任意类型T能不能自动转换为另一个任意类型U。

有一个用sizeof的解决方案（你原来是不是认为sizeof只在给memset传参数时才会用到？）。sizeof有令人吃惊的威力，因为任一个不管多复杂的表达式，sizeof都能得出其大小，并且竟然不用在运行时计算。这意味着sizeof可以识别重载、模板实例化、类型转换规则——在C++表达式里可能出现的任何东西。事实上，sizeof是一个完整的推导表达式类型的机制，最终sizeof丢掉表达式而只返回其结果的大小[3]。

进行类型检查的一个设想是用sizeof和重载函数。你提供一个函数的两个重载版本：一个接受类型U的参数，这是转换目的类型；另一个接受其他任何类型的参数。你用你希望测试可转换性的类型T去调用这个函数，如果接受U的那个函数被调用了，你就可以知道T可以转换成U；反之，如果另一个函数被调用，那么T不能转换成U。

为了检测哪个函数被调用，你要使两个重载版本返回大小不同的两个类型，然后用sizeof来区

分它们。返回什么类型无所谓，只要它们有不同的大小。

让我们先来创建两个大小不同的类型（显然，char和long double确实有不同的大小，但是这不是C++标准保证的）。一个最简单的方案如下：

```
typedef char Small;
struct Big { char dummy[2]; };
```

根据定义，`sizeof(Small)`为1。`Big`的大小还是不知道，但是可以确定的是它肯定比1大，我们只要有这个保证就行了。

接下来，我们需要两个重载函数。一个接受U，返回上面的两个类型之一，比如说`Small`。

```
Small Test(U);
```

怎么写接受其他任何类型参数的函数呢？我们不能用模板，因为模板总是会实例化为最匹配那个版本，这样就避免了类型转换。我们需要的是更糟糕的参数匹配，而不是自动转换。快速浏览一下函数调用的匹配规则，我们发现省略参数方式是最倒霉的，它是最后被选择的方式。我们需要的就是这个。

```
Big Test(...);
```

（用一个C++对象调用省略参数的函数，将产生未定义的结果，但是不用管它，没有人会真的去调用它，这个函数甚至根本不用实现。）

然后我们传入一个T类型的实体，调用`Test`，再用`sizeof`测试结果：

```
const bool convExists =
    sizeof(Test(T())) == sizeof(Small);
```

就是这样！`Test`的参数是一个缺省构造的T类型的对象，`sizeof`取出表达式的结果的大小。结果一定是`sizeof(Small)`和`sizeof(Big)`中的一个，就看编译器能不能作类型转换。

有一个小问题，如果T的缺省构造函数是私有的呢？这样的话，表达式T以及我们做的一切都不能编译。幸好，有一个简单的解决办法——用一个稻草人似的函数来返回一个T对象。然后，编译器就会满意了，我们也是。

```
T MakeT();
const bool convExists =
    sizeof(Test(MakeT())) == sizeof(Small);
```

（像`MakeT`和`Test`这样的函数，不仅什么也不干，甚至根本就不存在。但是，你却可以用它们做这么多事，是不是很漂亮？）

现在我们让它真正工作起来。让我们把所有东西包装在一个类模板里，把类型推导的细节隐藏起来，只把结果暴露给用户。

```
template <class T, class U>
class Conversion
{
    typedef char Small;
    struct Big { char dummy[2]; };
    static Small Test(U);
    static Big Test(...);
    T MakeT();
public:
    enum { exists =
        sizeof(Test(MakeT())) == sizeof(Small) };
};
```

我们可以这样测试一下Conversion模板类：

```
int main()
{
    using namespace std;
    cout
        << Conversion<double, int>::exists << ' '
        << Conversion<char, char*>::exists << ' '
        << Conversion<size_t, vector<int> >::exists << ' ';
}
```

这个小程序打印“1 0 0”。注意，尽管std::vector实现了一个以size_t为参数的构造函数，但是转换测试还是返回0，因为那个构造函数被声明为explicit。

我们可以在Conversion里实现另外两个常数：

exists2Way: 表示T和U之间是否存在双向转换。例如，int和double就是这样，可以互相转换。各种用户自定义的类型也可以实现这样的双向转换。

sameType: 如果T和U是同一类型，则为真。

```
template <class T, class U>
class Conversion
{
    ... as above ...
    enum { exists2Way = exists &&
        Conversion<U, T>::exists };
};
```

```
enum { sameType = false };
};
```

我们部分特化Conversion来实现sameType。

```
template <class T>
class Conversion<T, T>
{
public:
    enum { exists = 1, exists2Way = 1, sameType = 1 };
};
```

嗨，那么怎么做继承检测呢？最好的事情就是，当你有了检测类型转换的方法后，检测继承就很简单了。

```
#define SUPERSUBCLASS(B, D) \
(Conversion<const D*, const B*>::exists && \
!Conversion<const B*, const void*>::sameType)
```

很显然是吗？可能还有一点模糊。如果D是从B共有继承下来，或者B和D是同一种类型，SUPERSUBCLASS(B, D)为真。SUPERSUBCLASS是通过检测能不能把const D*转换成const B*来实现的。只有在三种情况下可以把const D*隐式地转换为const B*：

- B和D是同一种类型；
- B是D的无二义性的共有基类；
- B是void。

最后一种情况通过第二个判断排除。实际应用中接受第一种情况（B和D是同一类型）是有用的，因为实用中可能你经常需要认为一个类是它自己的基类。如果你想要严格的测试，你可以这样写：

```
#define SUPERSUBCLASS_STRICT(B, D) \
(SUPERSUBCLASS(B, D) && \
!Conversion<const B, const D>::sameType)
```

为什么代码中要加上那些const修饰符？原因是你会希望转换测试因为const的关系而失败。因此，上面代码中在所有地方都用了const；如果模板代码用了两次const（对一个已经是const的类型再加const修饰），第二个const会被忽略。总之，在SUPERSUBCLASS里用const，我们总是在安全的一边。

为什么要叫SUPERSUBCLASS，而不是更漂亮的BASE_OF或者INHERITS呢？这是因为一个很实际的理由：如果用INHERITS(B, D)的话，我老是要忘记测试的方向——是测试B继承于D还是反过来？SUPERSUBCLASS(B, D)就可以把哪个是第一个哪个是第二个分清楚（至少对我来说是这样）。

结论

关于这里介绍的三个惯用法，最重要的一点是，它们是可重用的。你可以把它们写到类库里去，让别的程序员使用，而不需要要求他们掌握复杂的内部的工作机制。

重要技术的重用性是很重要的。如果要求人们记住一个复杂的技术来做一件事，而他们可以用相对简单但是有点冗长的方法做到同样的事，那么他们不会用复杂的方法。给他们一个简单的黑盒子，可以帮他们做一些奇妙而有用的事，他们会喜欢它并使用它，因为它是free的。

`Int2Type`, `Type2Type`和`Conversion`属于一个通用的工具类库。通过做一些重要的编译时类型推导，它们使程序员在编译时能做更多的事。

致谢

如果Clint Eastwood问我：“你觉得幸运吗？”，我一定会说是的。这是我在这一系列中的第三篇文章，这得益于Herb Sutter本人的直接关注，以及非常好的建议。感谢日本的Tomio Hoshida指出一个bug，并且提出富有洞察力的建议。

本文包括Andrei Alexandrescu所著*Modern C++ Design* (Addison-Wesley, 2001) 一书的部分内容。

注：

- [1] 现在概念上没有要求C++支持函数的部分特化，我个人认为，这是一个值得期望的特性。
- [2] Andrei Alexandrescu. "Traits: The else-if-then of Types", C++ Report (April 2000).
孟岩. 《Traits：类型的else-if-then机制》. C++ View (第2期) .
- [3] 有个提案要求C++增加`typeof`操作符；就是返回一个表达式类型的操作符。如果有`typeof`的话，模板写起来会更容易，理解起来也更容易。GNU C++在它的扩展功能里，已经实现了`typeof`。显然，`typeof`和`sizeof`背后实现是类似的，因为`sizeof`总是要计算类型的。[参考Bill Gibbons将在以后的CUJ里发表的文章，他在目前的C++标准下提供了一个漂亮的，而且几乎是自然的`typeof`实现。]

C++批评系列：函数重载

Ian Joyner
cber 译

译者前言：要想彻底的掌握一种语言，不但需要知道它的长处有哪些，而且需要知道它的不足之处又有哪些。这样我们才能用好这门语言，避免踏入语言中的一些陷阱，更好地利用这门语言来为我们的工作所服务。Ian Joyner 的这篇文章以及他所著的 *Objects Inencapsulated* 一书中，向我们充分的展示了 C++的一些不足之处，我们应该充分借鉴于他已经完成的伟大工作，更好的了解 C++，从而写出更加安全的 C++代码来。

C++允许在参数类型不同的前提下重载函数。重载的函数与具有多态性的函数（即虚函数）不同处在于：调用正确的被重载函数实体是在编译期间就被决定了的；而对于具有多态性的函数来说，是通过运行期间的动态绑定来调用我们想调用的那个函数实体。多态性是通过重定义（或重写）这种方式达成的。请不要被重载(overloading)和重写(overriding)所迷惑。重载是发生在两个或者是更多的函数具有相同的名字的情况下。区分它们的办法是通过检测它们的参数个数或者类型来实现的。重载与 CLOS 中的多重分发 (multiple dispatching) 不同，对于参数的多重分发是在运行期间多态完成的。

【Reade 89】中指出了重载与多态之间的不同。重载意味着在相同的上下文中使用相同的名字代替出不同的函数实体（它们之间具有完全不同的定义和参数类型）。多态则只具有一个定义体，并且所有的类型都是由一种最基本的类型派生出的子类型。C. Strachey 指出，多态是一种参数化的多态，而重载则是一种特殊的多态。用以判断不同的重载函数的机制就是函数标示 (function signature)。

重载在下面的例子中显得很有用：

```
max( int, int )
max( real, real )
```

这将确保相对于类型 `int` 和 `real` 的最佳的 `max` 函数实体被调用。但是，面向对象的程序设计为该函数提供了一个变量，对象本身被当作一个隐藏的参数传递给了函数（在 C++中，我们把它称为 `this`）。由于这样，在面向对象的概念中又隐式地包含了一种对等的但却更有更多限制的形式。对于上述讨论的一个简单例子如下：

```
int i, j;
real r, s;
i.max(j);
r.max(s);
```

但如果我们这样写：`i.max(r)`，或是 `r.max(j)`，编译器将会告诉我们在这其中存在着类型不匹配的错误。当然，通过重载运算符的操作，这样的行为是可以被更好地表达如下：

```
i max j    或者
r max s
```

但是，`min` 和 `max` 都是特殊的函数，它们可以接受两个或者更多的同一类型的参数，并且还可以作用在任意长度的数组上。因此，在 Eiffel 中，对于这种情况最常见的代码形式看起来就像这样：

```
il:COMPARABLE_LIST[INTEGER]
rl:COMPARABLE_LIST[REAL]

i := il.max
r := rl.max
```

上面的例子显示，面向对象的编程典范（paradigm），特别是和泛型化（genericity）结合在一起时，也可以达到函数重载的效果而不需要 C++ 中的函数重载那样的声明形式。然而 C++ 使得这种概念更加一般化。C++ 这样作的好处在于，我们可以通过不止一个的参数来达到重载的目的，而不是仅使用一个隐藏的当前对象作为参数这样的形式。

另外一个我们需要考虑的因素是，决定（resolved）哪个重载函数被调用是在编译阶段完成的事情，但对于重写来说则推后到了运行期间。这样看起来好像重载能够使我们获得更多性能上的好处。然而，在全局分析的过程中编译器可以检测函数 `min` 和 `max` 是否处在继承的最末端，然后就可以直接的调用它们（如果是的话）。这也就是说，编译器检查到了对象 `i` 和 `r`，然后分析对应于它们的 `max` 函数，发现在这种情况下没有任何多态性被包含在内，于是就为上面的语句产生了直接调用 `max` 的目标代码。与此相反的是，如果对象 `n` 被定义为一个 `NUMBER`，`NUMBER` 又提供一个抽象的 `max` 函数声明（我们所用的 `REAL.max` 和 `INTERGER.max` 都是从它继承来的），那么编译器将会为此产生动态绑定的代码。这是因为 `n` 既可能是 `INTEGER`，也有可能是 `REAL`。

现在你是不是觉得 C++ 的这种方法（即通过提供不同的参数来实现函数的重载）很有用？不过你还必须明白，面向对象的程序设计对此有着种种的限制，存在着许多的规则。C++ 是通过指定参数必须与基类相符合的方式实现它的。传入函数中的参数只能是基类，或是基类的派生类。例如：

```
A.f( B someB )    {...}
class B ...
class D : public B ...
A a;
D d;
a.f( d );
```

其中 `d` 必须与类 '`B`' 相符，编译器会检测这些。

通过不同的函数签名（signature）来实现函数重载的另一种可行的方法是，给不同的函数以不同的名字，以此来使得它们的签名不同。我们应该使用名字来作为区分不同实体（entities）的基础。编译器可以交叉检测我们提供的实参是否符合于指定的函数需要的形参。这同时也导致了软件更好的自记录（self-document）。从相似的名字选择出一个给指定的实体通常都不会很容易，但它的好处确实值得我们这样做。

[Wiener95]中提供了一个例子用以展示重载虚拟函数可能出现的问题：

```
class Parent
{
public:
    virtual int doIt( int v )
    {
        return v * v;
    }
};

class Child: public Parent
{
public:
    int doIt( int v, int av = 20 )
    {
        return v * av;
    }
};

int main()
{
    int i;
    Parent *p = new Child();
    i = p->doIt(3);
    return 0;
}
```

当程序执行完后 `i` 会等于多少呢？有人可能会认为是 60，然而结果却是 9。这是因为在 `Child` 中 `doIt` 的签名与在 `Parent` 中的不一致，它并没有重写 `Parent` 中的 `doIt`，而仅仅是重载了它，在这种情况下，缺省值没有任何作用。

Java 也提供了方法重载，不同的方法可以拥有同样的名字及不同的签名。

在 Eiffel 中没有引入新的技术，而是使用范型化、继承及重定义等。Eiffel 提供了协变式的签名方式，这意味着在子类的函数中不需要完全符合父类中的签名，但是通过 Eiffel 的强类型检测技术可以使得它们彼此相匹配。

依赖倒置原则 DIP

Robert Martin

plpliuly 译

编者按：本文是 Robert Martin 关于 00 设计原则的著名文章系列“设计笔记”(Engineering Notebook) 的第 3 篇。原文可以在 <http://www.objectmentor.com> 网站上找到。

我在上一期的文章中向大家介绍了 Liskov 替换原则 LSP。将这个原则应用于 C++ 开发中，对怎样使用 public 继承有很好的指导作用。该原则指出：如果一个函数可以对一个基类的指针或引用进行操作，那么它应该同样可以操作该基类的派生类并且不需要知道派生类的具体类型。这意味着派生类中定义的虚函数不应该比基类的对应虚函数有更严格的前提条件（译者：虚函数执行前的输入条件，当然不仅仅是虚函数的输入参数）；也不应该减弱调用后的后续条件（译者：虚函数执行后的结果）。这也意味着基类中定义的虚函数必须在派生类中出现；而且应该的确是完成某个有意义的工作。当这个原则被破坏，操作基类指针或引用的函数就不得不判断具体对象的类型以确保这些函数可以正确地操作该对象。需要进行类型判断就意味着违反了我们最开始讨论的开放-封闭 OCP 原则（Open-Closed Principle）。

在本期的专栏中，我们将讨论 OCP 和 LSP 原则在程序结构上的引申。通过应用这些原则所设计出的程序结构自身也可以总结出一个原则。我称之为“依赖倒置原则 DIP”(the Dependency Inversion Principle)。

软件出了什么问题？

我们中的大多数人都有过与“设计糟糕”的软件打交道的不快经历。有些人甚至还有更不愉快的经历：发现我们自己就是这些“设计糟糕”的软件的作者。那么到底是什么东西使得设计如此糟糕呢？

大多数软件工程师并不是一开始就进行着“糟糕的设计”。然而大多数软件最终还是沦落为我们称之为缺乏良好设计的东西。为什么会这样呢？是因为设计本来就很糟糕，还是因为设计真的象臭肉一样经历了腐化变质过程？这个问题的核心是我们没有对于“糟糕的设计”的准确定义。

“糟糕的设计”的定义

你是否有这样的经历：当你非常得意地把你自认为很好的设计交给同事，你的同事却用嘲笑的语气向你抱怨：“你为什么要这样设计呢？”我有这样的经历，我也看到同样的事情发生在很多别的工程师身上。显然，对同一个设计持相反意见的工程师们使用了不同的标准来界定什么是“糟糕的设计”。我所见到的最为普遍使用的标准就是 TNTWIWHDI，也就是“这不是我所使用的方法(That's not the way I would have done it.)。”

但我认为存在一系列所有的工程师都会认同的标准。一个软件模块在实现需求的同时如果还表现出以下三个特性，那么它就是一个设计糟糕的软件。

1. 很难改动，因为每一处改动就会影响系统中过多的模块。（缺乏灵活性）
2. 当你做了一处改动，却导致系统的另一个模块发生了问题。（脆弱性）

3. 很难在别的应用程序中重用这个模块，因为不能将它从现有的应用程序中独立的提取出来。
(不可重用性)

而且，很难举出例子来说明一个不具有上述任何一点特性的软件模块——也就是说这个软件模块具有很好的灵活性，鲁棒性，可重用性，而且实现了所有的需求——却是一个设计很糟糕的东西。因此，我们可以使用这三个特性来确切地评判一个设计是好还是坏。

导致“糟糕设计”的原因

是什么导致一个设计缺乏灵活性，很脆弱，不可重用呢？是设计中各个模块之间的互相依赖关系。如果一个设计很难改动那么就意味着这个设计缺乏灵活性。导致缺乏灵活性的原因就是因为软件的各个部分有着紧密地相互依赖关系，任何一处改动就牵发其相关模块的一系列连锁反应的改动。当设计者或维护者难以预测改动所引起的连锁反应，那么这种改动所产生的影响就是无法事先预估的。面对这种不可预测性，管理者变得极不情愿批准对已有设计做出改动。于是设计就确确实实变得很僵化而缺乏灵活性了。

脆弱性是指，当有一处改动，程序的很多地方就可能冒出问题。常常是，出现新问题的地方与改动的地方并没有概念上的关联。这种脆弱性大大降低了设计和维护组织的可信性。用户和管理者无法预测产品的质量。应用程序中某个部分的一处简单改动就导致毫不相关的其他部分出问题。要修正这种问题就又会牵出更多的问题，从而使得维护过程变成了象一只狗在不停的追逐自己的尾巴一样。

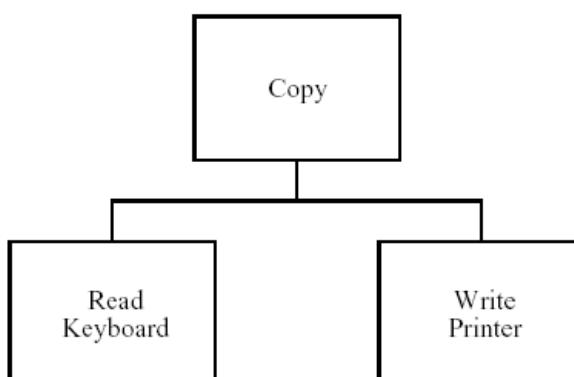
当一个设计中的值得重用的部分过度依赖于其他不需要重用的细节，这样的设计就缺乏重用性。如果一个设计者正在考察已有设计以判断其是否可以在别的应用程序中重用时，那么他的脑海里思考的是将这个设计应用到新的程序中将会怎样。然而，假设这个设计中的各个部分是高度耦合的，要想重用其中某个值得重用的部分，你就得首先花费大量的工夫将这个部分从与其他部分的千丝万缕的关系中分离出来。大多数情况下，这种设计是不会被重用的，因为分离出可重用的模块所花费的代价往往高于重新设计所需的代价。

例子：“copy”程序

一个简单的例子或许可以帮助更好的说明这个问题。设想一个简单的程序，其任务就是实现将键盘输入的字符拷贝到打印机上。进一步假设实现平台中的操作系统并不支持设备无关性。我们可以为这个程序构造出如图 1 的结构：

图 1 是一个“结构图 (structure chart)”¹。

Figure 1. Copy Program.



¹ 参考：《结构化系统设计实用指南》(The Practical Guide Press, 1988)

图中显示了在这个应用程序中总共有三个模块，或称子程序。“Copy”模块调用其他两个模块。我们可以轻易的想象出在“Copy”模块中有一个循环(参考 Listing 1)。循环体中调用“Read Keyboard”模块获取键盘输入的字符，然后将字符送给“Write Printer”模块将字符打印。

```
//Listing 1 The Copy Program
void Copy ()
{
    int c;
    while ( (c = ReadKeyboard ()) != EOF)
        WritePrinter (c);
}
```

两个底层模块具有很好的可重用性。它们可以被用在许多需要访问键盘和打印机的程序中。这种重用就如同我们重用子程序库中的子程序一样。

然而“Copy”模块在一个不涉及键盘和打印机的程序中就无法重用。这个模块乃实现整个系统功能的关键所在，正是“Copy”模块封装了我们想重用的一种有意思的策略。然而这样一个模块却不能重用，委实不应该。

例如，设想一个新的程序用来实现拷贝键盘输入的字符到磁盘文件。我们自然就想到重用“Copy”模块，因为“Copy”模块封装了我们所需要的高层的策略，也就是它知道如何从一个字符源拷贝字符到接收器中。不幸的是，“Copy”模块依赖“Write Printer”模块，因此并不能在新的应用中重用这个模块。

```
//Listing 2. The "Enhanced" Copy Program
enum OutputDevice {printer, disk};
void Copy (outputDevice dev)
{
    int c;
    while ( (c = ReadKeyboard ()) != EOF)
        if (dev == printer)
            WritePrinter (c);
        else
            WriteDisk (c);
}
```

我们当然可以对“Copy”加以改动以使之具备新的功能。(参看 Listing 2) 我们可以在拷贝策略中加上“if”语句以通过判断某个标志变量来选择“Write Printer”模块或“Write Disk”模块。然而这就在系统中增加了新的相互依赖关系。随着时间的推移，更多的设备就会加入拷贝程序中，“Copy”模块就会充满 if/else 的语句，也意味着依赖很多低层的模块。这最终将使“copy”模块变得僵化而脆弱。

依赖倒置

上述问题的一个显著特点就是包含高层策略的模块，也就是 Copy () 模块，依赖于它所控制的低层模块（也就是 WritePrinter () 和 ReadKeyboard ()）。如果我们能够找到一个方法使得 Copy () 模块独立于它所控制的细节，那么我们就可以非常容易的重用这个模块。我们可以在别的程序中使用这个模块从任何输入设备拷贝字符到任何输出设备。OOD 给我们提供了一个执行这种“依赖倒置”的机制。

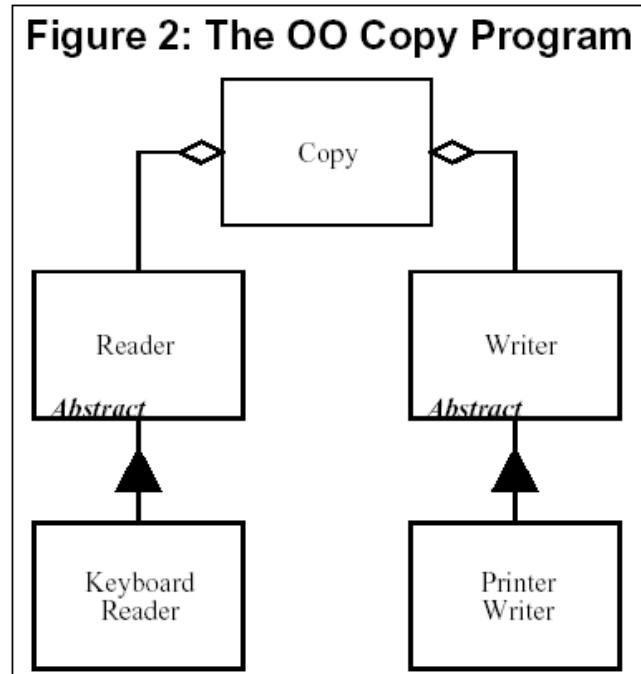
请思考图 2 中的简单类图。在此图中我们有一个“Copy”类，此类包含有一个抽象的“Reader”类和一个抽象的“Writer”类。我们很容易就能想象出“Copy”类中的一个循环，此循环从“Reader”中读取字符，然后送到“Writer”（参考 Listing 3）。然而此处的“copy”类既不依赖于“Keyboard Reader”也完全不依赖于“Printer Writer”。这样，依存关系就被倒置了；“Copy”类依赖于抽象类，具体的 reader 和 writer 也依赖于同样的抽象类。

```
//Listing 3: The OO Copy Program
class Reader
{
public:
    virtual int Read () = 0;
};

class Writer
{
public:
    virtual void Write (char) = 0;
};

void Copy (Reader& r, Writer& w)
{
    int c;
    while ( (c=r.Read ()) != EOF)
        w.Write (c);
}
```

现在我们就可以重用“Copy”类，而不再需要依赖于“Keyboard Reader”和“Printer Writer”了。我们可以设计新的“Reader”和“Writer”的派生类给“Copy”类使用。而且，无论创建了多少种“Readers”和“Writers”，“Copy”类不会依赖于它们中任何一个。导致程序缺乏灵活性或者很脆弱的相互依赖关联已不复存在。Copy () 自身可以被用在许多不同的具体场合。因而它是易重用的。



设备无关性

到此为止，一些读者可能要说，通过使用 stdio.h 中提供的设备无关的函数，也就是 getchar 和 putchar 函数（参看 Listing 4），用 C 语言写一个 Copy() 就可以轻易得到上面所描述的各种好处。如果你仔细考虑 Listing 2 和 Listing 4 中的代码，你会发现二者在逻辑上是等价的。图 3 中的抽象类被 Listing 4 中一种不同的抽象代替。Listing 4 中的代码的确没有用到类和虚函数，但也使用了抽象和多态。而且，它同样应用了依赖倒置。Listing 4 中的 Copy 程序不依赖于它所控制的任何细节，而是依赖于在 stdio.h 中声明的抽象体。而且，最终被触发的输入输出驱动程序也只是依赖于这些在 stdio.h 中声明的抽象体。所以，我们可以说 stdio.h 库中的设备无关特性是关联倒置的另一个例子。

```

//Listing 4: Copy using stdio.h
#include <stdio.h>
void Copy ()
{
    int c;
    while ( (c = getchar ()) != EOF)
        putchar (c);
}
  
```

到目前为止，我们已经讨论过了DIP的好几个例子，不妨总结一下DIP的通用描述。

关联倒置原则 DIP

A. 高层模块不应该依赖于低层模块。二者都应该依赖于抽象。*(HIGH LEVEL MODULES SHOULD NOT DEPEND UPON LOW LEVEL MODULES. BOTH SHOULD DEPEND UPON ABSTRACTIONS.)*

B. 抽象不应该依赖于细节。细节应该依赖于抽象。*(ABSTRACTRIONS SHOULD NOT DEPEND UPON DETAILS. DETAILS SHOULD DEPEND UPON ABSTRACTIONS.)*

有人可能会问我为什么使用“倒置（inversion）”这个词。坦白地讲，这是由于很多传统的软件开发方法，比如结构化分析和设计，总是倾向于创建高层模块依赖于低层模块、抽象则依赖于细节的软件结构。实际上这些方法的目的只是想以此定义子程序层次结构，而子程序层次结构则描述了高层模块是怎样调用低层模块的。图1就是这样一个层次结构的典型例子。这样，一个设计良好的面向对象的程序的依赖关系结构相对于传统过程式方法设计的通常的结构而言就是被“倒置”了。

请考虑一下当高层模块依赖于低层模块时意味着什么。高层模块包含了一个应用程序中的重要策略选择和业务模型。也正是高层模块使得其所在应用程序区别于其他。然而，如果这些高层模块依赖于低层模块，那么对低层模块的改动就会直接影响它们；从而迫使它们也做出改动。

这种情形其实是非常荒谬的。应该是高层模块的改动迫使低层模块改动。高层模块应该优先于低层模块。无论如何高层模块都不应该依赖于低层模块。

而且，我们更希望能够重用的是高层模块。因为，我们仅通过使用子程序库的方式就可以很好的重用低层模块。当高层模块依赖于低层模块，在不同的场合重用高层模块就变得非常困难了。然而，当高层模块独立于低层模块时，高层模块就能够非常容易地被重用。这是框架设计的一个核心原则。

层次化

Booch曾经说过：“所有的结构良好的面向对象架构都具有非常清晰的层次，每一个层次通过一个被很好地定义和控制的接口向外提供了一系列相互内聚的服务。”²对于这个陈述的简单理解可能会致使一个设计者设计出类似图3的结构。在图中高层的策略类调用了低层的机制层；而机制层又调用更具体的工具类。这看起来似乎没什么不妥，然而它存在一个隐伏的特性，那就是：Policy Layer对于其下层次一直到Utility Layer的改动都是非常敏感的。这意味着关联是可传递的。Policy Layer依赖于某些依赖于Utility Layer的层次，因此Policy Layer传递性地依赖于Utility Layer。这是非常不幸的。

² Object Solutions, Grady Booch, Addison Wesley, 1996, p54

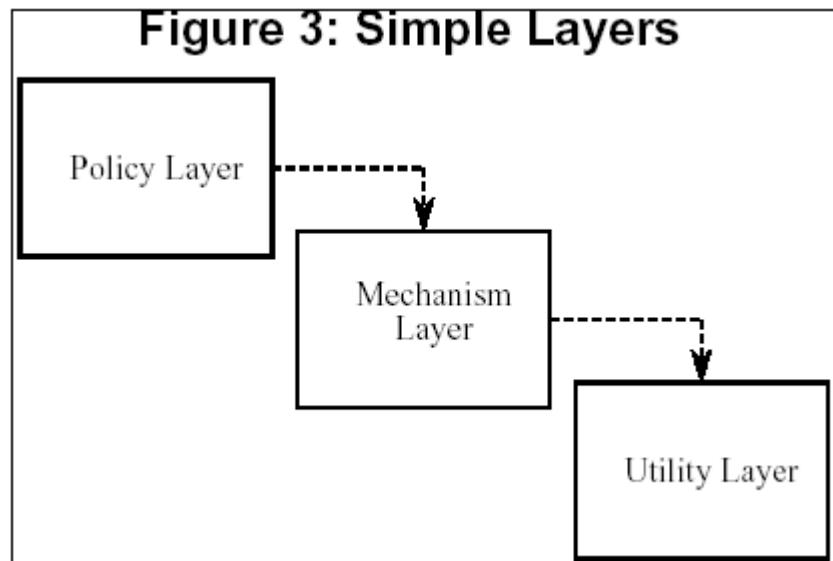
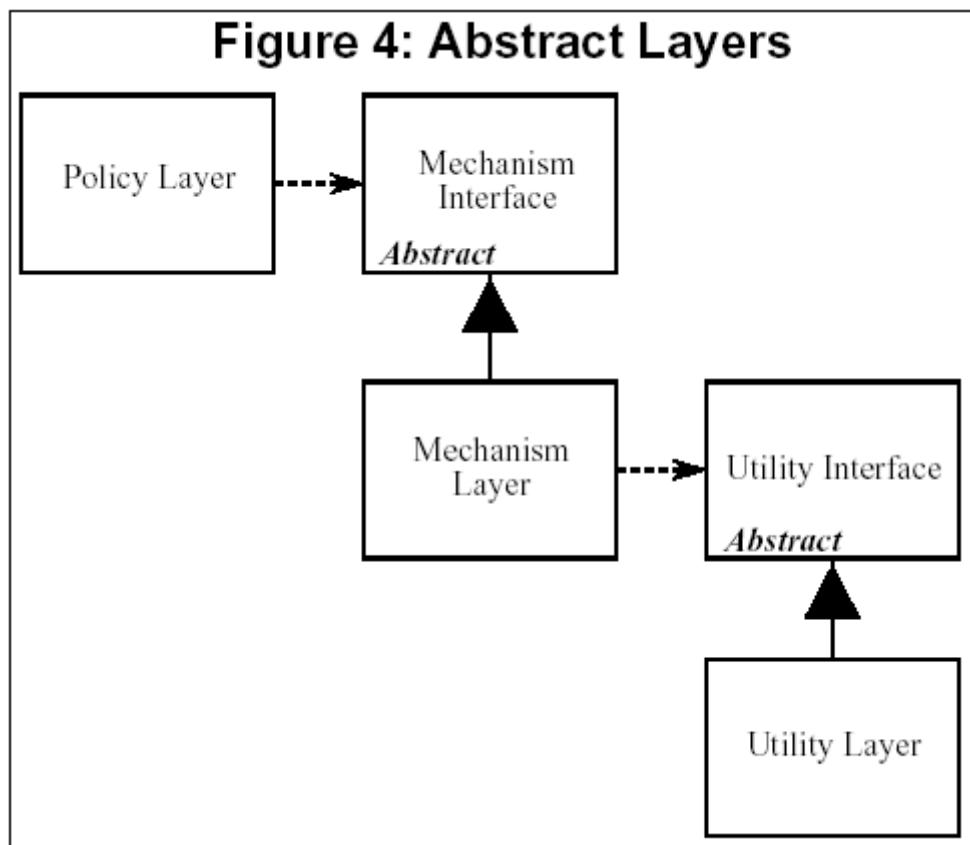


图4显示了一个更为合适的模型。每一个较低层次都用一个抽象类来表示。实际的层次从这些抽象类进行派生。每一个高层的类都通过抽象接口使用下一层。这样，任何一层都不依赖于其它层，而是依赖于抽象类。不仅仅是Policy Layer对于Utility Layer的传递性依赖关系被解除了，Policy Layer对于Mechanism Layer的依赖都不复存在了。



通过使用这个模型，Policy Layer不会受Mechanism Layer或Utility Layer的任何改动的影响。而且，

任何别的场合，只要定义了遵循Mechanism Layer接口的低层模块，都可以重用Policy Layer。

在 C++ 中分离接口与实现

有人可能要抱怨图3中的结构并没有显示依赖关系，也没有展现我所声称的可传递的依赖问题。毕竟，Policy Layer只是依赖于Mechanism Layer的接口。为什么对于Mechanism Layer的实现的改动就会影响到Policy Layer呢？

在某些面向对象语言中，的确是这样的。在这些语言中，接口是自动与实现分离的。然而在C++中却没有这样的接口与实现的分离。在C++中，分离的是类定义和成员函数定义。

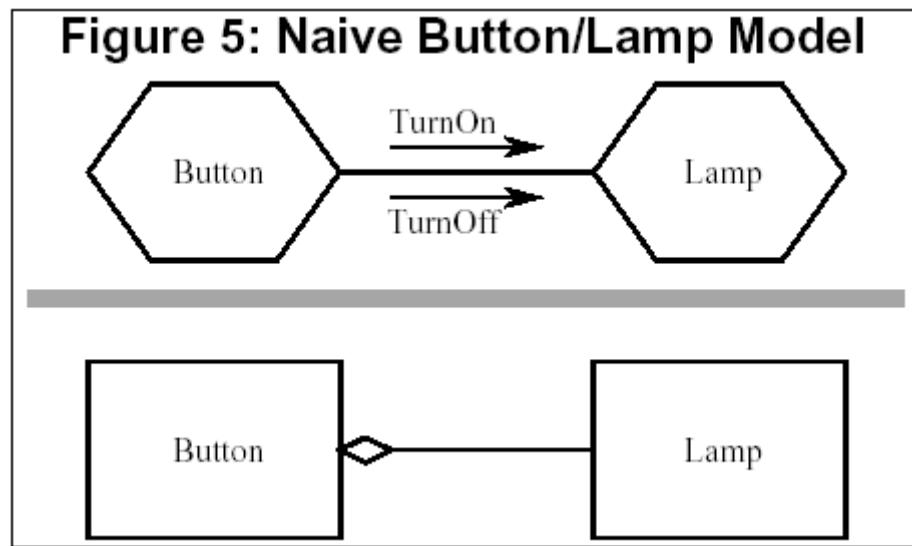
在C++中，我们将一个类分离成两个模块：一个.h模块和一个.cc模块。.h模块包含类的定义，而.cc模块包含类的成员函数的定义。.h模块中类的定义包含了所有成员函数及成员变量的声明。这些信息已经超出了简单的接口。所有的辅助工具和私有变量都被声明在.h模块中。这些工具和私有变量是属于类的实现的一部分，然而却出现在所有的类使用者都必须依赖的模块中。由此看出，在C++中，实现并没有自动地和接口分离。

我们可以通过使用纯抽象类来解决C++中不能自动完成接口和实现分离的问题。一个纯抽象类是一个只包含纯虚函数的类。这样一个类就是一个纯接口；其.h模块不包含任何实现细节。图4显示的就是这样一个结构。图4中的抽象类就是指纯抽象类，因此每一层仅仅依赖于下层的接口。

一个简单的例子

关联倒置可以应用于任何存在一个类向另一个类发送消息的地方。比如，Button对象和Lamp对象之间的情形。

Button对象感知外界环境的变化。它可以判断是否被用户“按下”。但它不在乎是通过什么样的机制感知外界的。可能是图形用户界面中的一个按钮，也可能是一个真正的能够用手指按下的按钮，甚至是一个家庭安全系统中的一个运动监测器。Button可以检测到用户激活或关闭它。指示灯对象影响外部环境。在接收到TurnOn消息后，它显示某种灯光。在接收到TurnOff消息后它将灯光熄灭。物理机制无关紧要。它可以是一个计算机控制台的LED，也可以是一个泊车处的水银灯，甚至是激光打印机的激光。



该怎样设计一个用Button对象控制Lamp对象的系统呢？图5显示了一个不太成熟的模型。Button对象简单地发送TurnOn和TurnOff消息给Lamp对象。为方便起见，Button类使用了“包容”关系持有一个Lamp类的实例。

Listing 5中是对应这个模型的代码。请注意Button类是直接依赖于Lamp类的。事实上，button.cc模块#include了Lamp.h模块。这个依赖关系意味着当Lamp类改动时，按钮类必须改动，至少需要重新编译。而且，如果想重用Button类来控制另一个对象，比如Motor对象，是不大可能的。

```

// Listing 5: Naive Button/Lamp Code
-----lamp.h-----
class Lamp
{
public:
    void TurnOn ();
    void TurnOff ();
};

-----button.h-----
class Lamp;
class Button
{
public:
    Button (Lamp& l) : itsLamp (&l) {}
    void Detect ();
private:
    Lamp* itsLamp;
};

-----button.cc-----
#include "button.h"

```

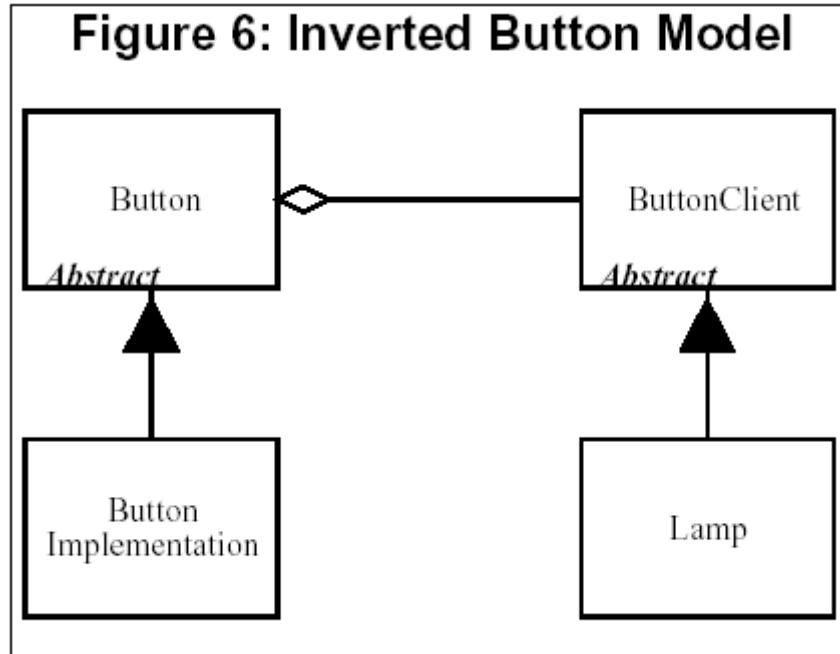
```
#include "lamp.h"
void Button::Detect ()
{
    bool buttonOn = GetPhysicalState () ;
    if (buttonOn)
        itsLamp->TurnOn () ;
    else
        itsLamp->TurnOff () ;
}
```

图5和Listing 5违反了关联倒置原则。应用程序的高层策略没有与低层模块分离；抽象没有与具体细节分离。没有这种分离，高层策略就自动地依赖于低层模块，抽象自动地依赖于具体细节。

找出潜在的抽象

什么东西是高层策略？是应用程序背后的抽象，那些不随具体细节改变而改变的真理。在 Button/Lamp例子中，背后的抽象就是检测用户开/关指令并将指令传给目标对象。是用什么机制检测用户的指令呢？不重要。什么是目标对象？同样无关紧要。这些都是不会影响到抽象的具体细节。

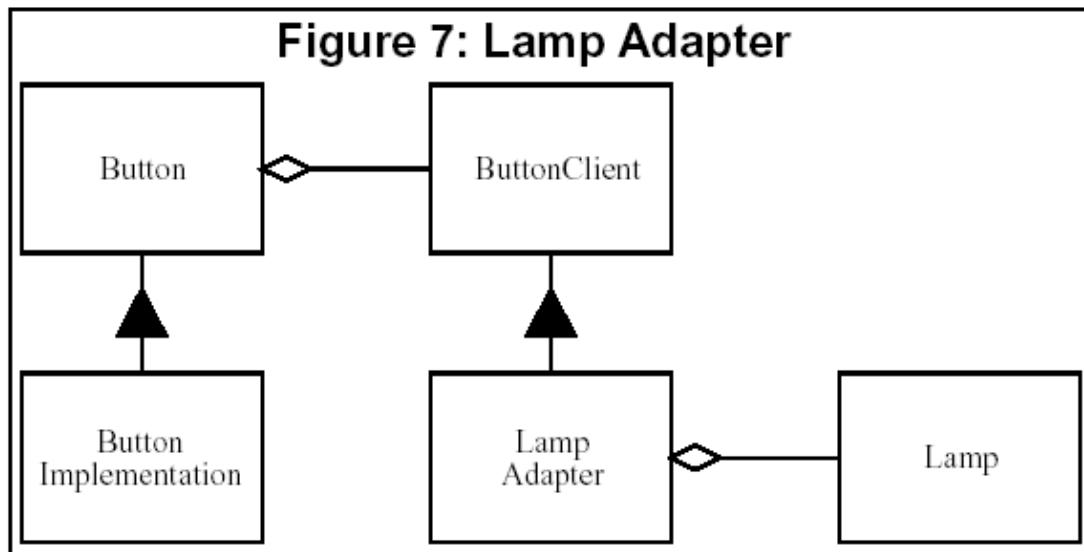
要做到遵循依赖倒置原则，我们必须从问题的具体细节中分离出抽象。然后调整设计的依赖关系让细节依赖于抽象。图6展现的是这样一个设计。



在图6中，我们将Button类的抽象从具体的实现细节中分离出来。Listing 6中是对应的代码。请

注意高层策略完全包括在抽象的按钮类中³。Button类对于检测用户指令的机制完全不了解；它也不知道关于灯的任何信息。这些细节已经被分离到具体的派生类中：ButtonImplementation和Lamp。

Listing 6中的高层策略对于任何类型的按钮和任何类型的待控制的设备都是可以重用的。而且，它不会受到低层机制改动的影响。因此，它对于改动是鲁棒的，灵活的，而且是可重用的。



进一步抽象

有人可能会对图6和Listing 6中的设计提出合乎情理的抱怨：那就是被按钮控制的设备对应的类必须从ButtonClient派生，如果Lamp类是来自第三方提供的库，那么我们就无法更改其源码。

图7显示了怎样应用适配器（Adaptor）模式将一个第三方提供的Lamp对象连接到模型中来。LampAdaptor类简单的将从ButtonClient中继承的TurnOn和TurnOff消息转换成Lamp类需要的消息。

结论

关联倒置原则是实现面向对象所声称的诸多优点的一个重要原则。这个原则的正确应用对于创建可重用框架是必需的。它对于构建具有高弹性的代码同样是至关重要的，因为，当抽象和具体细节被分离以后，代码的维护工作就变得容易多了。

³ 热爱设计模式的读者可能已经识别出在Button层次结构中应用了模板方法（Template Method）模式。成员函数Button::Detect()是使用了纯虚函数Button::GetState()的模板。参看：Design Patterns, Gamma,et.al.Addison Wesley, 1995

Pattern Hatching

——弃儿、养子和替身

作者: John Vlissides

翻译: 虫虫

原文出处: C++ Report, June 1995

Pattern Hatching 专栏的第一篇文章介绍了 COMPOSITE (组合) 模式, 说明了如何在文件系统的设计中使用该模式。这次我们将由此更深入地进行探讨, 关注 Node 类接口设计中一个重要的权衡问题, 并尝试为这个初具规模的设计增加新的功能。

再谈 COMPOSITE 模式

COMPOSITE 模式构成了这个应用程序的骨架, 向我们展示了如何用面向对象的方式来描述一个分级文件系统的基本特征。该模式通过继承和组合, 把几个关键的协作者——Component、Composite 和 Leaf 类——联系起来, 以支持任意大小和复杂度的文件系统结构, 同时也能让客户以统一的方式处理文件和目录等。

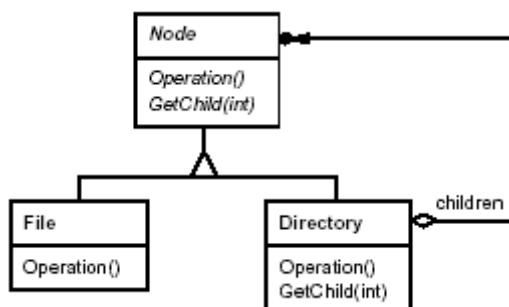
统一性的关键依赖于文件系统中各个对象的公共接口。迄今为止, 我们的设计中有三类对象: Node、File 和 Directory., 其关系如图 3 所示。

上次我解释了对于文件和目录有不同意义的操作需要在基类中声明的原因, 取得和设定节点名字【译注: 即 GetName 和 SetName】和保护状态【译注: 即 GetProtection 和 SetProtection】的操作均在此列。我也解释了为什么要把访问子节点的操作 (GetChild) 放在公共接口中, 虽然乍看之下这并不适合 File 对象。这次我们将考虑一些并那么明显的公共操作。

ADOPT/ORPHAN: NODES AND FILES NEED NOT APPLY

我们希望一个 Directory 对象能枚举出其孩子, 但是首先必须以某种方式获取它们。从哪里获取呢? 目录本身可不能承担创建它可能包含的所有孩子的责任——这是文件系统使用者的权力。合理的方法是, 让文件系统的客户来创建文件和目录, 并把它们放在所希望的地方。这就意味着 Directory 对象会收养子节点而非创建它们。因此 Directory 需要如下的一个接口来收养孩子:

```
virtual void Adopt(Node* child)
```



Node, File, Directory 之间的关系

当客户对某个目录调用 Adopt 时，客户显式地把对给定孩子（可能是任意类型的 Node）责任移交给了目录。责任意味着所有权：当目录被删除时，子类也是同样的下场。这就是 Directory 和 Node 间聚合关系（如图中菱形所示）的本质。

现在客户能让某个目录承担对某个孩子的责任，那么也应该有一个放弃责任的调用函数：

```
virtual void Orphan(Node* child)
```

这里 Orphan 并不意味着父目录完蛋了，或被删除了，而仅仅说明该目录不再是这个孩子的父目录。孩子依然存在，也许过会儿就被另一个节点收养，也许被删除。

那么这跟统一性有什么关系？为什么我们只为定义这些操作，而不能用在别的什么地方呢？

OK，假设我们这么做。现在考虑一下客户如何实现这些改变文件系统结构的操作。一个创建新目录的用户级命令就是客户的一个实例。命令的用户界面无关紧要，假设就是 UNIX 下的 mkdir 这样一个命令行界面。mkdir 把要创建的目录名作为一个参数：

```
mkdir newsubdir
```

事实上，用户可以在目录名前面加上任何合法的路径：

```
mkdir subdirA/subdirB/newsubdir
```

只要 subdirA 和 subdirB 存在且都为目录而非文件，就没有问题。更一般地，subdirA 和 subdirB 应该是 Node 子类中拥有孩子的类的实体。否则，用户应该得到出错信息。

我们如何实现 mkdir 呢？我们先假设 mkdir 能找到当前目录，也就是说它能获取一个 Directory 对象的引用，符合用户对当前目录的选择¹。在当前目录上添加新目录仅仅需要创建一个 Directory 实体，再以新目录为参数调用当前目录的 Adopt 方法即可。

```
Directory* current;  
//...  
current->Adopt(new Directory("newsubdir"));
```

简单。但对于一般的情况呢？mkdir 如何满足一般的路径名？

这下复杂了。mkdir 必须要

1. 找到 subdirA 对象（不存在则报错）；
2. 找到 subdirB 对象（不存在则报错）；
3. 让 subdirB 收养 newsubdir 对象。

第 1 和第 2 条需要遍历当前目录，以及遍历 subdirA（如果存在的话）以找寻代表 subdirB 的节点。mkdir 实现的核心部分可以是一个以路径名为参数的递归函数。

¹ 客户可以通过一个众所周知的区域获知当前的 Directory 对象，比如 Node 类的一个静态操作。访问公共区域是 SINGLETON（单件）模式的拿手好戏，这是后话，暂且不提。

```
void Client::Mkdir (Directory* current, const char* path) {
    const char* subpath = Subpath(path);

    if (subpath == 0) {
        current->Adopt(new Directory(path));
    } else {
        const char* name = Head(path);
        Node* child = Find(name, current);
        if (child) {
            Mkdir(child, subpath);
        } else {
            cerr << name << " nonexistent." << endl;
        }
    }
}
```

Head 和 Subpath 是字符串处理程序。Head 返回路径中的第一个目录或文件名，Subpath 返回剩余部分，Find 操作在一个目录中搜索给定名字的孩子。

```
Node* Client::Find (const char* name, Directory* current)
{
    Node* child = 0;

    for (int I = 0; child = current->GetChild(); ++i) {
        if (strcmp(name, child->GetName()) == 0) {
            return child;
        }
    }
    return 0;
}
```

注意 Find 必须返回 Node*，因为这是 GetChild 所返回的类型。这并非不合理，因为一个孩子可能是 Directory 类型，也可能是 File 类型。但如果注意，这个小小的细节就会让 Client::Mkdir 翻船——无法编译。

再看看这个递归调用 Mkdir。它传入参数的类型是 Node*，而非所需要的 Directory*。问题是处于在层次下层时，我们无法分辨一个孩子到底是一个 File 还是 Directory。一般情况下，只要客户不在乎这点差别，这倒是一件好事。但在这里看起来我们的确要在乎这点差别，因为只有 Directory 定义了收养和遗弃孩子的接口。

不过我们真正在乎的究竟是什么呢？或者更扼要地说，客户（mkdir 命令）是否需要在乎？否。客户需要做的就是建立一个新目录或者向用户报错。

AN EQUAL OPPORTUNITY INTERFACE

如果我们透过 Node 类统一处理 Adopt 和 Orphan 如何呢？“天啦！”有人反驳，“这些操作对叶部件（如 File）毫无意义”。这个假设很有远见么？如果以后定义了一种新的叶部件，比如不论接受到什么都通通将其干掉的垃圾箱（更准确地说，回收站），又如何呢？如果叶子收养东西意味着产生出错信息呢？很难保证 Adopt 对叶子永远都没有意义，对 Orphan 也是同样的道理。

另一方面，争论一开始是否需要区分 File 和 Directory 似乎有点意思——一切都应该是 Directory——但具体实现的确不同。Directory 对象比文件多一个包袱：存储孩子的数据结构，用以缓冲孩子的信息来提升性能，等等。经验表明，在许多应用程序中，叶子比内部节点多得多，这也是 COMPOSITE 模式限定要区分 Leaf 和 Composite 类的原因。

我们来看看，如果在所有 Node 上定义 Adopt 和 Orphan，而不仅仅局限于 Directory 类，会带来什么。这些操作默认动作是产生出错信息。

```

virtual void Node::Adopt (Node*) {
    cerr << GetName() << " is not a directory." << endl;
}

virtual void Node::Orphan (Node* child) {
    cerr << child->GetName() << " is not a directory." << endl;
}

```

【译注：这里 virtual 只是告诉诸位，这些函数应该是虚函数。相信大家都清楚，virtual 只能出现在成员函数的声明中，而不能出现在实现里。】

这些不一定是最适当的出错信息，不过明白意思就行了。这些操作可以抛出异常，也可以什么也不做——有许多可能的做法。无论如何，Client::Mkdir 运行得很出色²！注意现在也不需要改动 File 类。

需要指出：虽然 Adopt 和 Orphan 起来也许并非我们所要统一处理的操作，但至少在这个应用程序中，却有实实在在的好处。最可能的办法是引入某种向下映射，使客户能辨认出节点的类型。

```

void Client::Mkdir (Directory* current, const char* path) {
    const char* subpath = Subpath(path);

    if (subpath == 0) {
        current->Adopt (new Directory(path));
    }
}

```

² 基本上出色。我承认这里我忽略了内存管理问题。特别地，对一个叶子调用 Adopt 时可能出现内存泄露，因为客户向某节点传递所有权时，该节点可能没有成功地接受。这是 Adopt 的一个大问题，因为用它操作 Directory 对象时可能会失败（比如客户权限不够）。若 Node 是引用计数的形式，问题则不复存在，失败时引用计数递减或不增即可。

```
    } else {
        const char* name = Head(path);
        Node* node = Find(name, current);

        if (node) {
            Directory* Child = dynamic_cast<Directory*>(node);

            if (child) {
                Mkdir(chld, subpath);
            } else {
                cerr << name << " is not a directory." << endl;
            }
        } else {
            cerr << name << " nonexistent." << endl;
        }
    }
}
```

注意 `dynamic_cast` 是如何引入了额外的控制路径？当用户在路径指定的名字是文件而非路径时，我们需要处理这种特殊情况，这也说明了不统一如何使得客户更麻烦。

【译注：我个人感觉这个函数的设计不是很好。原文发表后不久，Michael Hittesdorf 指出，该函数的形式应该是 `void Client::Makdir (Node* current, const char* path)`。不过按我的理解，这段代码有两个问题，一个是作者自己已经提到的异常安全问题，一个则是应该尽量避免的 `dynamic_cast`。我觉得代码可以改成如下形式：

```
//如果不想改动下面这两个函数的返回值类型，那么抛出异常也可以。
/*virtual*/ int Node::Adopt(Node*)
{
    ...
    return 0;
}

/*virtual*/ int Directory::Adopt(Node*)
{
    ...
    return 1;
}

void Client::Mkdir(Node* current, const string& path)
{
    if (!current) return;
```

```
string subpath = Subpath(path);
if(subpath.empty()){
    auto_ptr new_dir(new Directory(path));
    if(current->Adopt(new_dir.get()))
        new_dir.release();
} else
    Mkdir(Find(Head(path), current), current);
}
```

这样就避免了我刚才提到的问题，不过 John Vlissides 博士并不欣赏，只好留给大家作参考⑧】

BUT WHERE DO SURROGATES FIT INTO ALL THIS?

问得好，我们正该看看添加一个新东东，也就是，符号连接（symbolic links，即 Mac Finder 中的 aliases 或 OS/2 Workplace Shell 中的 shadows）【译注：对应于 Microsoft Windows 中的快捷方式（shortcut）】，其实也就是对文件系统中另一个节点的引用。它并非那个节点本身，而是其替身。如果删除某个符号连接，它所指代的节点并不受影响。符号连接拥有可能与所指代节点不同的访问权限。

不过在绝大部分情况下，符号连接的行为就像节点本身。如果它指代某个文件，客户可以把它当作那个文件一样处理；也就是说，客户可以通过这个连接编辑甚至存储该文件。如果它指代某个目录，客户可以通过它在该目录中添加或删除节点。

符号连接很爽，可以让我们在不用搬动或复制文件的前提下访问远程文件和目录，对于那些只能存在于某处却需要在外地访问的节点特别有用。如果我们的设计不加入符号连接，那就太疏忽了。

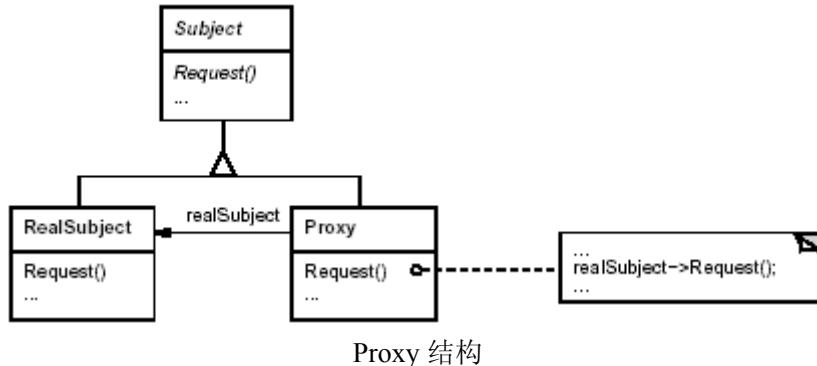
那些为买 *Design Patterns* 一书花了大把银子的朋友首先会问：“有没有哪个模式能帮忙设计并实现符号连接呢？”事实上，这可是一个重大的问题：“如何为手上的任务找到正确的设计模式？”*Design Patterns* 的 1.7 节建议按如下步骤：

- 考虑设计模式如何解决问题（也就是看看 1.6 节，可惜这次我们没时间谈这个）。
- 看看意图（Intent）部分有什么可能正确的东西（有点用蛮力的感觉）。
- 学习模式之间有何关系（现在对我们而言依然很棘手，不过我们在逐渐提高）。
- 看看哪些模式的设计意图（创建型、结构型、行为型）符合我们想要做的东西（比如往文件系统中加入符号连接就意味着其目的应该是结构型）。
- 检查导致重新设计的相关原因（见 *Design Patterns* 第 24 页），应用能帮助我们避免重新设计的模式（这里我们不用真正担心重新设计）。
- 考虑设计中哪些是要变动的。*Design Patterns* 中的表 1.2 列出了每个模式允许改变的方面。

看看表 1.2 中的结构型模式，我们看到

- ADAPTER 能改变对象的接口，
- BRIDGE 能改变对象的实现，
- COMPOSITE 能改变对象的结构和组合方式，

- DECORATOR 能不派生子类而改变职责，
- FAÇADE 能改变子系统的接口，
- FLYWEIGHT 能改变对象的存储开销，
- PROXY 能改变对象的访问方式及其位置，



也许是我偏袒，不过 PROXY 显然就是我们所要的模式。过去看看，该模式的意图是：为其他对象提供一种代理以控制对该对象的访问。

动机（Motivation）部分说到应用该模处理延迟载入图片的问题（跟在网页浏览器中的情况也差不多）。适用性（Applicability）部分告诉我们，PROXY 模式适用于实现更多用途或更精致的引用，包括一种可以控制访问另一个对象的保护代理（protection proxy）——这正是我们需要的。

应用 PROXY

OK，如何在我们的文件系统设计中应用 PROXY 模式呢？看看该模式的结构图（Structure diagram）图 4，我们能看到三个关键类：抽象类 **Subject**，具现子类 **RealSubject** 和 **Proxy**。由此可推断 **Subject**、**RealSubject** 和 **Proxy** 有一致的接口。同时，子类 **Proxy** 也包含了一个对 **RealSubject** 的引用。

该模式的参与者（Participants）部分说，**Proxy** 类提供了与 **Subject** 完全一致的接口，故 **Proxy** 对象能代替真正的主体。更进一步地，**RealSubject** 正是代理所代表的对象。

把这些关系对应回我们文件系统里的那些类，显然我们需要的公共接口就是 **Node** 的那些操作（别忘了，这可是 COMPOSITE 模式教我们的）。这说明 **Node** 类相当于模式中 **Subject** 类。为此我们需要从 **Node** 派生一个子类，以相当于模式中的 **Proxy** 类，命名为 **Link**：

```

class Link : public Node {
public:
    Link(Node* );
    // redeclare common Node interface here
private:
    Node* _subject;
};
  
```

成员 `_subject` 提供了一个对真正主体的引用。注意我们略微违反了该模式的结构图，原本该引用的类型应该是 `RealSubject`，也就是说，这里引用的类型本应该是 `File` 或者 `Directory`，但我们希望对任何 `Node` 类型，符号连接都能正常工作。不过看看模式中对参与者 `Proxy` 的描述，我们能看到这样的话：

[`Proxy`] 保存一个引用使得代理可以访问真正的主体。如果 `RealSubject` 和 `Subject` 的接口相同，`Proxy` 可以引用 `Subject`。

根据以上的讨论，可以看出 `File` 和 `Directory` 确实需要共享 `Node` 接口。因此 `_subject` 是一个指向 `Node` 类型的指针。如果不是公共的接口，定义一个能同时指代文件和目录的符号连接则要困难得多。事实上如果这样，我们就不得不定义两种完全一样的符号连接，一个指代文件而另一个指代目录，仅此而已。

最后一个重要的问题是关于 `Link` 实现 `Node` 的接口。实现方式不过是把每一项操作都委派给 `_subject` 相应的操作。比如，`GetChild` 可能委派以如下形式：

```
Node* Link::GetChild (int n) {
    return _subject->GetChild(n);
}
```

在某些情况下，`Link` 可能表现出与其主体不同的行为，如 `Link` 能实现与其主体不同的保护性操作，这时它可以像 `File` 一样实现操作。

【译注：这里 `Link` 的实现很粗糙，在实际情况中，还有很多需要考虑的因素。比如 `Link` 所指代的主体被删除或替换了，又当如何呢？这些细节问题值得我们注意。本文原文发表后，Laurion Burchall 在给 John Vlissides 的 email 中也谈到这些问题，他认为可以引入 `OBSERVER` 模式来解决主体被删除的情况，当然替换的情况就要注意了。另外他提到，在 UNIX 和 Mac 中，符号连接保存的是它们所指代主体的名字，而不是其他（比如对主体的引用），这样也有助于解决上面的问题。】

STAY TUNED

当改进设计时，一个值得密切注意的倾向是把基类当垃圾一样乱扔：操作随着时间的推移增加，接口越来越复杂。文件系统每添加一个新特性，就要迎来一两个操作函数。今天要支持扩展特性，下星期要进行新的大小统计计算，下个月又要一个 GUI 中返回图标的操作。不久，`Node` 类将会成为一个庞然大物，理解、维护以及由它派生子类都会变得非常困难。

下次我们将讨论这个问题，看看如何在不修改已有类的前提下在设计中加入新操作，别急哦。

参考

1. GoF. *Design Patterns: Elements of Reusable Object-Oriented Software*, Addison-Wesley, Reading, MA, 1995.

模式罗汉学：Interface（接口）模式

透明

梗概

用一个类使用其他类的实例提供的数据和服务。由于“使用服务的类”通过一个接口来访问这些“提供服务的实例”，所以它可以与这些实例所属的类保持互不依赖。

场景

假设你正在为一家公司编写采购管理软件。在你的程序中需要下列实体的信息：生产厂商、货运公司、收货地址、交款地点……这些实体的一个共同就是：它们都有一条“街道地址”信息。这条信息将在用户界面的不同部分出现，所以你希望用一个类来显示、编辑街道地址。这样，每当用户界面中出现街道地址时，你可以复用这个类。我们把这个类称为 AddressPanel 类。

你希望 AddressPanel 对象可以从一个独立的数据对象中接收、设置地址信息。但是，对于这些数据对象，AddressPanel 对象应该把它们当作什么类的实例呢？很明显你应该用不同的类来表示生产厂商、货运公司……等等的实体。如果你用 C++ 这样支持多重继承的语言编程，你可以让“AddressPanel 使用的数据对象所属的类”在继承其他类的同时还继承一个地址类。但是如果你使用的编程语言象 JAVA 这样只支持单继承，你就应该再想想其他的解决方案。

在 JAVA 中你可以用接口（interface）来代替 C++ 中的多重继承。这样 AddressPanel 类就只要求数据对象实现地址接口。你也可以通过这个接口访问或设置数据对象中的地址信息。通过这个接口，AddressPanel 对象可以在不知道对象确切类型的前提下调用数据对象的方法。图 1 的类图表示出了这种关系。

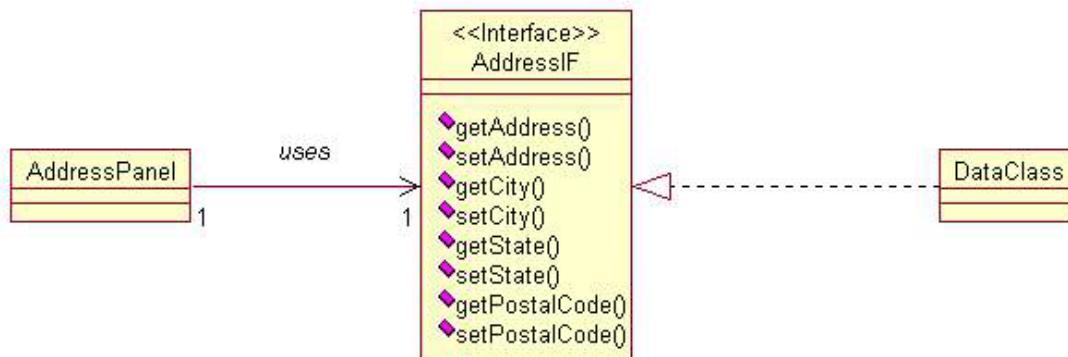


图 1 通过地址接口进行间接访问

约束

- 如果一个类的实例必须使用另一个对象、而这个对象又属于一个特定的类，那么复用性会受到损害。

- 如果“使用”类只需要“被使用”类的某些方法、而不是要求“被使用”类与“使用”类有“is-a”的关系，就可以考虑让“被使用”类实现一个接口、“使用”类通过这个接口来使用需要的方法，从而限制了类之间的依赖。

解决方案

为了避免类之间因为彼此使用而造成的耦合，让它们通过接口间接使用。图 2 表示出这种组织结构。

图 2 中的角色如下：

- Client（客户）——使用实现了 IndirectionIF 的其他类。
- IndirectionIF（间接接口）——提供间接性，保证 Client 与 Service 之间的独立性。
- Service（服务者）——实现 IndirectionIF，为 Client 提供服务。

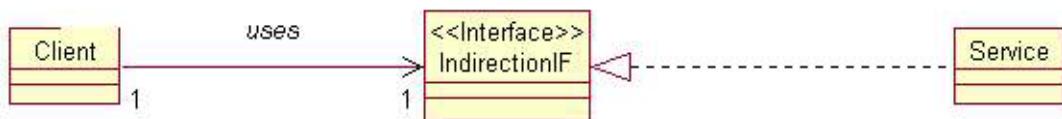


图 2：类之间通过接口解耦

效果

“让调用者通过接口间接使用服务者”，这是面向对象设计的基础——多态性正是从这样的设计中产生的。

使用 Interface 模式，可以保证需要服务的类不与任何提供服务的类发生耦合。

和其他任何间接性一样，Interface 模式会让程序变得更加难以理解。

使用 Interface 模式同样有可能造成对继承的滥用。但是，通过 Interface 模式让你可以从接口的角度考虑设计思想。这是面向对象设计的三大原则之一：“针对接口编程，而不是针对实现编程” [GoF95, P12]。（另外两条原则是：“优先使用对象组合，而不是类继承” [GoF95, P13]¹ 和“发现并封装变化点” [GoF95, P20]）

实现

实现 Interface 非常直接：定义提供服务的接口，让客户类通过该接口访问服务者对象，并让服务者类实现该接口。

Interface 模式在 JAVA 中得到了直接的支持：interface 关键字。超越语言本身，实际上在 C++ 和 JAVA 中，你都可以用同样直接的方式实现 Interface 模式——C++ 尽管不提供 interface 关键字，但我们也通常把纯虚类称为“接口”，并用多重继承来“实现”接口。在用 C++ 实现 COM 规范时，我们也是用纯虚类来作为接口的。

¹ 关于这条原则的详细解释，请看 C++ View 第 3 期上的“委托模式”一文。

同时，Interface 模式还有不那么直接的实现方式。图 2 中的 Client 和 IndirectionIF 之间的“使用”关系可能不那么直接；Client 和 Service 都可能不是一个类而是一组类甚至一个应用程序；IndirectionIF 也可能不是一个接口（或抽象类）而是一个具体类；IndirectionIF 和 Service 之间也可能不是实现（继承）的关系而是使用的关系……重要的是，应该随时想到“针对接口编程”的原则。

C++应用

正如前面所说的，C++（当然也包括其他所有的面向对象语言）是依靠 Interface 模式来实现多态性的。这种比较接近细节的应用，不提也罢，实在太多了！

当我们实现 COM 规范时，我们就使用了 Interface 模式：客户程序只知道接口信息，并用 CoCreateInstance 函数来获取适当的服务器对象。但是，三个角色之间的关系更加复杂。（如果读者对此有兴趣，请参阅[PAN99]）

代码示例

我将就“场景”一节中讨论过的问题给出一个简略的示例代码。单是代码本身不能完整的表示 Interface 模式的精妙，但愿能起到抛砖引玉之效，让读者能理解“针对接口编程”的思想。

首先是代替 AddressPanel 出现的 Client 类：

```
#include "AddressIF.h"
class Client
{
public:
    Client() {
        m_pAddress = NULL;
    }

    string GetAddress() {
        if(m_pAddress!=NULL)
            return m_pAddress->getAddress();
        else
            return string();
    }

    bool SetAddress(string & strAddress) {
        if(m_pAddress!=NULL) {
            m_pAddress->setAddress(strAddress);
            return true;
        }
    }
}
```

```
    else
        return false;
};

void SetAddressIF(AddressIF * pAddress) {
    m_pAddress = pAddress;
};

private:
    AddressIF * m_pAddress;
};
```

然后是本模式的核心：提供间接性的“接口”。我们用一个纯虚类 AddressIF 来实现它：

```
#include <string>
using std::string;
class AddressIF
{
public:
    virtual void setAddress(const string & strAddress) =0;
    virtual string getAddress() =0;

};
```

AddressIF 只是提供需要的方法，由服务者类来实现它。

最后是服务者——DataClass 类的代码。它通过继承来“实现” AddressIF “接口”。

```
#include "AddressIF.h"
// .....
// 其他头文件。
class DataClass :
    // .....继承其他类
    public AddressIF
{
public:
    // .....其他方法
    virtual string getAddress() {
        return m_strAddress;
    };
    virtual void setAddress(const string & strAddress) {
        m_strAddress = strAddress;
    };
};
```

```
private:  
    // .....其他成员数据。  
    string m_strAddress;  
};
```

主控程序可能是这样：

```
#include <iostream.h>  
#include "DataClass.h"  
#include "Client.h"  
  
void main()  
{  
    Client client;  
    DataClass dc1, dc2;  
  
    dc1.setAddress("Beijing");  
    dc2.setAddress("Shanghai");  
  
    client.SetAddressIF(&dc1);  
    cout<<client.GetAddress().c_str()<<endl;  
  
    client.SetAddressIF(&dc2);  
    cout<<client.GetAddress().c_str()<<endl;  
}
```

相关模式

Delegation 模式[GRAND98]——Delegation 模式和 Interface 模式经常在一起使用。²
另外有很多模式使用了 Interface 模式。

参考书目

- [GoF95] Erich Gamma etc., *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley 1995. 中文版：《设计模式：可复用面向对象软件的基础》，李英军等译，机械工业出版社 2000 年 9 月。
- [GRAND98] [Mark Grand](#), *Patterns in Java (volume I)*, Wiley computer publishing 1998.
- [PAN99] [潘爱民](#), 《COM 原理与应用》，清华大学出版社 1999 年 11 月。

² 我在 C++ View 第三期上介绍了 Delegation 模式。

天方夜谭 VCL

作者：虫虫

多态



我们中国人崇拜龙，所谓“龙生九种，九种各别”。哪九种？《西游记》里西海龙王对孙悟空说：“第一个小黄龙，见居淮渎；第二个小骊龙，见住济渎；第三个青背龙，占了江渎；第四个赤鬃龙，镇守河渎；第五个徒劳龙，与佛祖司钟；第六个稳兽龙，与神官镇脊；第七个敬仲龙，与玉帝守擎天华表；第八个蜃龙，在大家兄处砥据太岳。此乃第九个鼍龙，因年幼无甚执事，自旧年才着他居黑水河养性，待成名，别迁调用，谁知他不遵吾旨，冲撞大圣也。”（注：鼍龙是文雅的说法，民间叫法是猪婆龙，也就是扬子鳄。）如果您冲着这九位说一声“Let's go”，那场面可壮观了，有天上飞的，有水里游的，也有地上爬的。同样是“go”，“go”的具体形式却各不相同，这正是多态“一个接口，多种实现”的典型例子。

多态的实现方法很多，其中 C++ 直接支持的方式有：通过关键字 `virtual` 提供虚函数进行迟后联编，以及通过模板（template）实现静态多态性，它们都各有用武之地。我们比较熟悉的是虚函数，这是建构类层次的重要手段，我们也已经分析过虚函数的原理¹。然而在有些情况下，虚函数的性能并不是最优，故 VCL 还提供了一种动态（dynamic）函数，用法和虚函数一模一样，只要把 `virtual` 换成 `DYNAMIC` 就可以了。VCL 的帮助文件里说，动态函数跟虚拟函数相比，空间效率占优，时间效率不行，真的吗？其实现原理又是如何呢？我们又应该如何权衡这两者的使用呢？我们将从一个相当一般的角度来讨论这些问题。

虚函数的苦恼

如下类层次来自一个图形绘制程序的一部分。为了方便管理，界面与具体的图形设计分离。各种图形以动态连接库的方式提供，作为插件的形式。这样可以在不重新编译主程序的情况下，增加或减少各种图形。

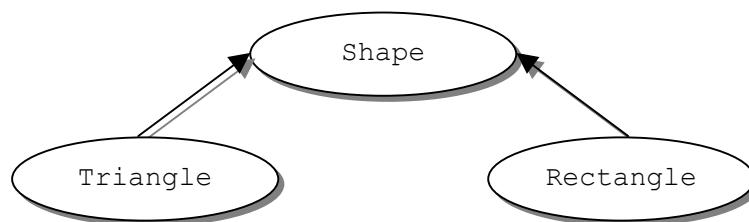


图 1 Shape 类层次

最初 `Shape` 的声明是

```
class Shape {  
private:  
    int x0, y0;  
protected:  
    Shape();  
    virtual ~Shape();  
public:  
    int x() const;  
    int y() const;  
    virtual void draw(void *) = 0;  
    virtual int move(int, int);  
};
```

后来因为功能扩充，添加了两个虚函数。

```
class Shape {  
private:  
    int x0, y0;  
protected:  
    Shape();  
    virtual ~Shape();  
public:  
    int x() const;  
    int y() const;  
    virtual int move(int, int);  
    virtual void draw(void *) = 0;  
    virtual void save(void *) const = 0;  
    virtual void load(void *) = 0;  
};
```

后来又作过一些修改，又添加了若干虚函数。问题就在于，虚函数一但增加，虚拟函数表 VFT 就会发生变化，这时候，主程序就必须重新编译。更糟糕的是，一旦版本升级，派生自不同版本 shape 的图形绝对不可以混用²。所以我们可以看到硬盘里充斥着 mfc20.dll、mfc40.dll、mfc42.dll……却一个也不能删除，这就是 MFC 升级所带来的 DLL 垃圾。怎么办？

初步解决

我在网上问过这样的问题，得到的答复主要有：

- 用 COM；
- 预先多写一些无用的虚函数，留出扩充空间。

其实上面的方法都能很好地解决这个问题。但是推广看来，也有一定局限性。COM 不适合解决类层次过深的情况，预留的空间则是不折不扣的“鸡肋”。

追根究底，这个局限性是因为父类和子类的虚拟函数表 VFT 之间过强的关联性：子类的 VFT 的前面一部分必须与父类相同。而当父类和子类不在同一个 DLL 或 EXE 中的时候，这个要求是很难满足的。父类一旦改变，子类如果不重新编译，就将导致错误。解决的方法，当然就是取消父类和子类 VFT 之间的关联性。我设计了一个很笨的解决办法，但可以取消这个关联性，使虚函数保证始终只有 2 个。

```
#define Dynamic // Dynamic 什么都不是，只是好看一点

struct point
{
    int x, y;
};

class dispatch_error{};

class Shape {
private:
    int x0, y0;
protected:
    Shape();
    virtual ~Shape();
    virtual void dispatch(int id, void* in, void* out);
    // in 和 out 是函数的输入输出参数，id 是每个函数唯一的标记符号，即代号
    // 实际运用中，id 不一定是整数，也可以是 128 位 UUID，或者字符串等等
public:
    int x() const;
    int y() const;
    Dynamic int move(int dx, int dy)
    {
        int r;
        point p = {dx, dy};
        dispatch(-1, &p, &r);
        return r;
    }
    Dynamic void draw(void *hdc)      {dispatch(-2, hdc, 0);}
    Dynamic void save(void* o) const {dispatch(-3, o, 0);}
    Dynamic void load(void* i)       {dispatch(-4, i, 0);}
};

void Shape::dispatch(int id, void* in, void* out)
{
    switch(id)
```

```

    {
        case -1:
        ...
        case -2:
        ...
        ...
    default:
        throw(dispatch_error()); // 若函数不存在则抛出异常
    }
}

```

如果子类 Triangle 要改写 Shape::draw，那么只需要

```

void Triangle::dispatch(int id, void* in, void* out)
{
    switch(id)
    {
        ...
        case -2: // 改写 Shape::draw
        ...
        ...
    default:
        Shape::dispatch(id, in, out); // 函数不存在则向父类找
    }
}

```

这样的“Dynamic 函数”就解决了前面的问题，只有析构和 dispatch 这两个虚函数。父类和子类的 VFT 之间没有关联性，可以自由改动而不会互相影响。

评头论足

我们来对这种解决方案作了评价：的确解决了虚函数的问题，但是也付出了不小的代价：时间效率和可读性，由此也决定了该方案的应用面不广，一般用于

- 虚函数很少或几乎不需改写的情况。这样有助于减少 VFT 的大小。至于运行速度则没有什么提高，毕竟 VFT 的访问速度是常数级³；
- 父类需要经常更新而子类不方便同步更新，对效率要求又不高的情况。一般的应用程序都可以使用。

从模式（Patterns）的角度来看，这种方法是典型的职责链（Chain of Responsibility）模式⁴：调用请求从最低层子类开始一层层往上传递，直到被处理或者最后抛出异常。这种模式运用非常广泛，比如 VCL 消息映射⁵ 和 COM 中 IDispatch 接口⁶，与上述解决方案的形式都非常相似。

这个解决方案还可以作进一步的完善，以更好地适用于单根结构的框架。比如单根结构的类库，如 MFC 和 VCL，通过 RTTI 可以找到唯一的父类，那么可以分离数据（函数代号和指针）和代码（调配部分），以简化结构。解决的方法就是典型的表格驱动，有不少书^{7, 8}都用此来优化 COM 中 IUnknown 接口的 QueryInterface。我们引入类 DMT 来储存函数的代号和指针。

```
#include <cstdarg>
using namespace std;

class DMT {
    char* const ptr;
    const DMT* const parent;
public:
    DMT(const DMT* const, const int, ...);
    ~DMT() {delete []ptr;}
    short size() const {return *(short*)ptr;}
    const void* find(int) const;
};


```

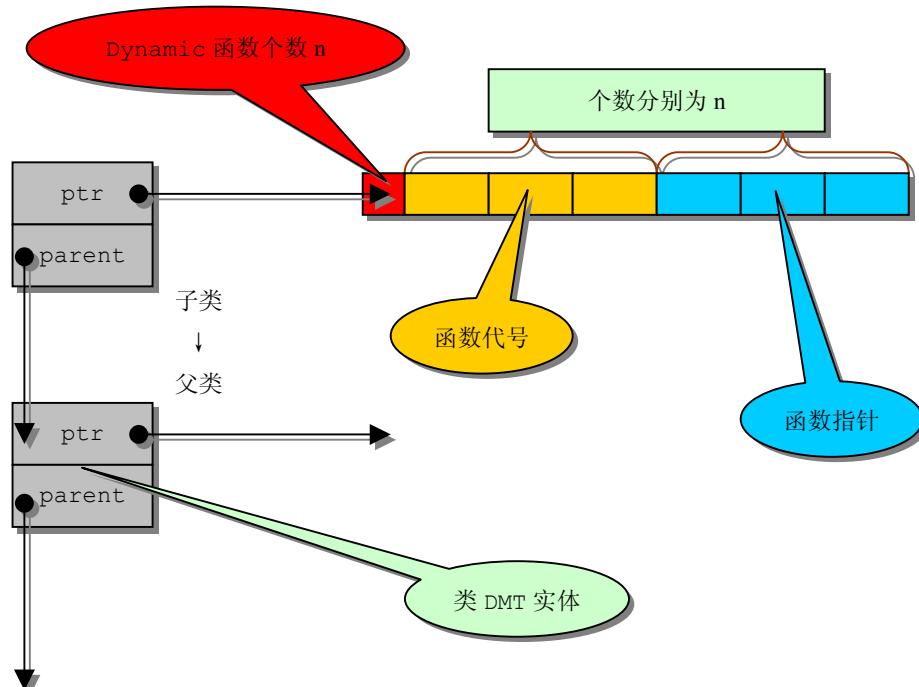


图 2 类 DMT 图解

需要特别注意的是 DMT::ptr 所分配的空间。在 32 位系统上，对于 n 个“Dynamic 函数”，需要 $\text{sizeof}(\text{short})$ 字节储存 n （红色部分）， $\text{sizeof}(\text{void}^*) * n$ 字节储存函数代号（黄色部分），以及 $\text{sizeof}(\text{void}^*) * n$ 字节储存函数指针（蓝色部分），一共是 $\text{sizeof}(\text{short}) + 2 * n * \text{sizeof}(\text{void}^*)$ 字节。子类和父类的 DMT 可以通过链表形式连接起来。下面我们看看

DMT::find 和 DMT::DMT 的实现。

```

const void* DMT::find(int i) const
{
    const int* begin = (int*)(ptr + sizeof(short)), *p;
    for(p = begin; p < begin + size(); ++p)
        if(*(int*)p == i)
            return *(void**)(p+ size());
    // 找到对应的函数代号后，向前跳 DMT::size() 则是相应的函数指针
    return (parent)? parent->find(i): 0;
}

DMT::DMT(const DMT* const p, const int n, ...)
    : parent(p), ptr(new char[sizeof(short)+2*n*sizeof(void*)])
    // ptr 分配的空间大小如前所述
{

    int* i = (int*)(ptr + 2), c;
    *(short*)ptr = n; // 往头 sizeof(short) 字节写入 n (红色部分)

    va_list ap;
    va_start(ap, n);

    for(c = 0; c < n; ++c) // 往黄色部分写入函数代号
        *(i++) = va_arg(ap, int);

    typedef void (DMT::*temp_type)();
    temp_type temp;

    for(c = 0; c < n; ++c) // 往蓝色部分写入函数指针
    {
        temp = va_arg(ap, temp_type);
        *(i++) = *(int*)&temp;
    }

    va_end(ap);
}

```

下面我们在 Shape 类层次中应用 DMT 类。

```

class Shape {
private:
    int x0, y0;

```

```

        void int f_move(void* dx, void* dy) {...}

protected:
    static const DMT dmt_Shape; // Shape 类的 DMT
    const DMT* const dmt; // 指向该类 DMT 的指针
    Shape() : dmt(&dmt_Shape) {...}
    virtual ~Shape();

    void dispatch(int id, void* in, void* out) // 这次不是虚函数!
    {
        void (A::*f)(void*, void*);
        *(const void**)&f = dmt->find(id);
        (this->*f)(in, out);
    }
public:
    int x() const;
    int y() const;
    Dynamic int move(int dx, int dy)
    {
        int r;
        point p = {dx, dy};
        dispatch(-1, &p, &r);
        return r;
    }
    Dynamic void draw(void *hdc)      {dispatch(-2, hdc, 0);}
    Dynamic void save(void* o) const {dispatch(-3, o, 0);}
    Dynamic void load(void* i)       {dispatch(-4, i, 0);}
};

const DMT Shape::dmt_Shape =
DMT(0, 4, -1, -2, -3, -4, &Shape::f_move, 0, 0, 0);

```

背景突出部分就是有改动的地方。如果子类 Triangle 要改写 Shape::draw，那么只需要

```

class Triangle {
private:
    void f_draw(void*);
    ...
protected:
    static const DMT dmt_Triangle;
    ...
public:
    Triangle() {dmt = &dmt_Triangle; ...}
    ...
};

```

```
const DMT Triangle::dmt_Triangle =
    DMT(Shape::dmt_Shape, ..., -2, ..., &Triangle::f_draw...);
```

这就是对“Dynamic 函数”的另一种实现，这样可以分离数据和代码。当然这个示例并不具备实际应用价值，在静态成员初始化、调用约定、可读性等诸多设计上都有不少问题，仅仅起演示作用。

动态（dynamic）函数

Object Pascal 提供了两种函数实现多态：一种是我们熟悉的虚拟（virtual）函数，另外一种则是动态（dynamic）函数，其实就是对前面的“Dynamic 函数”提供的语言级别的支持。

可能有些用 C++ Builder 的朋友说，C++ Builder 里怎么看到啊？在 C++ Builder 里，标识动态函数的宏（macro）是 DYNAMIC，也就是 __declspec(dynamic)，这是 Borland 对 C++ 的扩充。像 TControl::Click、TControl::MouseMove 等等都是动态函数。DYNAMIC 的用法和 virtual 基本一致，我所发现的不同仅仅是，当子类改写父类相应函数时，子类中 virtual 可以省略，而 DYNAMIC 则不行。

那么，每个类的动态函数的入口在哪里呢？上次，我们已经挖出了 VMT 的分布图，里面就有 vmtDynamicTable = 0xfffffd0 这么一句，字面就告诉我们，这是动态方法表 DMT（Dynamic Method Table）的入口。不妨检验一下。

```
#include <vcl.h>
#include <iostream>
struct A: private TObject
{
    DYNAMIC void f1() = 0;
    void f3() {}
    virtual void f4() {}
    DYNAMIC void f2() = 0;
};
struct B: A
{
    DYNAMIC void f1() {}
    DYNAMIC void f2() {}
};
void main()
{
    A* p = new B;
    std::cout<<(void*)p<<std::endl;
    p->f1();
    p->f2();
```

```

    delete p;
}

```

这个程序会输出“0118095C”。当然不同的机器上这个数值可能有所不同，总之先记下了。

其中 `p->f1();` 的汇编代码是

```

push dword ptr [ebp-0x30]
or edx,-0x01           ;这句其实就相当于 mov edx,0xffffffff
mov eax,[ebp-0x30]
call System::FindDynaInst(void *, int)
call eax
pop ecx

```

`p->f2();` 的汇编代码是

```

push dword ptr [ebp-0x30]
mov edx,0xfffffff
mov eax,[ebp-0x30]
call System::FindDynaInst(void *, int)
call eax
pop ecx

```

程序很简单，我们说明一下。

- `or edx,-0x01` 跟 `mov edx,0xffffffff` 的效果是完全一样的，任何数和 `0xffffffff` 进行“或”运算的结果当然都是 `0xffffffff`；
- 这两段唯一的不同就是 `mov edx,0xfffffff`（也就是 `or edx,-0x01`）和 `mov edx,0xfffffff`，我们上次已经说过了补码表示法，这里其实就是分别传递的 `A::f1` 和 `A::f2` 的函数代号-1 和-2；
- 执行过 `mov eax,[ebp-0x30]` 这一句后，我们可以发现 `eax` 的值就是刚才我们记下的数（`0118095C`），这里包含了指向 VMT 入口的指针；
- 向 `System::FindDynaInst` 传入的两个参数就分别是包含指向 VMT 入口的指针，以及相应函数的代号，分别在 `eax` 和 `edx` 里；
- 显然 `System::FindDynaInst` 把对应函数代号的函数指针放在 `eax` 里，`call eax` 就调用相应的函数。

这就是整个大的流程。现在我们关心的是，`System::FindDynaInst(void*, int)` 到底做了些什么。我们可以跟踪进去，再跳一层，我们来到了函数中，源代码就是 `Source\Vcl\system.pas` 中的 `_FindDynaInst`。

```

procedure      _FindDynaInst;
asm
    PUSH    EBX

```

```

MOV    EBX, EDX      ;EBX 储存了函数的代号
MOV    EAX, [EAX]     ;EAX 获得 VMT 入口地址
CALL   GetDynaMethod ;调用 GetDynaMethod
MOV    EAX, EBX
POP    EBX
JNE    @@exit
POP    ECX
JMP    _AbstractError

@@exit:
end;

```

那么我们还得看看 GetDynaMethod 的源代码。

```

procedure      GetDynaMethod;
  {function GetDynaMethod(vmt: TClass; selector: Smallint) : Pointer;}
asm
  { ->  EAX      vmt of class          }
  {      BX       dynamic method index   }
  { <-  EBX      pointer to routine    }
  {      ZF = 0 if found               }
  {      trashes: EAX, ECX            }

  PUSH   EDI
  XCHG   EAX, EBX      ;交换 eax 和 ebx 的值
  JMP    @@haveVMT     ;交换后 ebx 是 VMT 入口地址, eax 是函数代号

@@outerLoop:
  MOV    EBX, [EBX]     ;取地址

@@haveVMT:
  MOV    EDI, [EBX].vmtDynamicTable ;EDI 是 DMT 的入口
  TEST   EDI, EDI           ;测试是否存在 DMT (EDI 是否为 0)
  JE     @@parent          ;若 DMT 不存在, 在父类中继续找
  MOVZX ECX, word ptr [EDI] ;取头两个字节, 即动态函数个数
  PUSH   ECX
  ADD    EDI, 2             ;跳至黄色部分 (见后面的图)
  REPNE SCASW              ;查找 eax
  JE     @@found           ;若找到则跳转
  POP    ECX

@@parent:
  MOV    EBX, [EBX].vmtParent ;在父类中继续
  TEST   EBX, EBX           ;是否有父类
  JNE    @@outerLoop        ;有则继续查找
  JMP    @@exit             ;无则跳转

```

```

@@found:
    POP    EAX
    ADD    EAX, EAX      ; 以下两步是清除 ZF，其中 ECX 值为 0
    SUB    EAX, ECX      { this will always clear the Z-flag ! }
    MOV    EBX, [EDI+EAX*2-4] ; edi-1 是函数代号所在处

@@exit:
    POP    EDI
end;

```

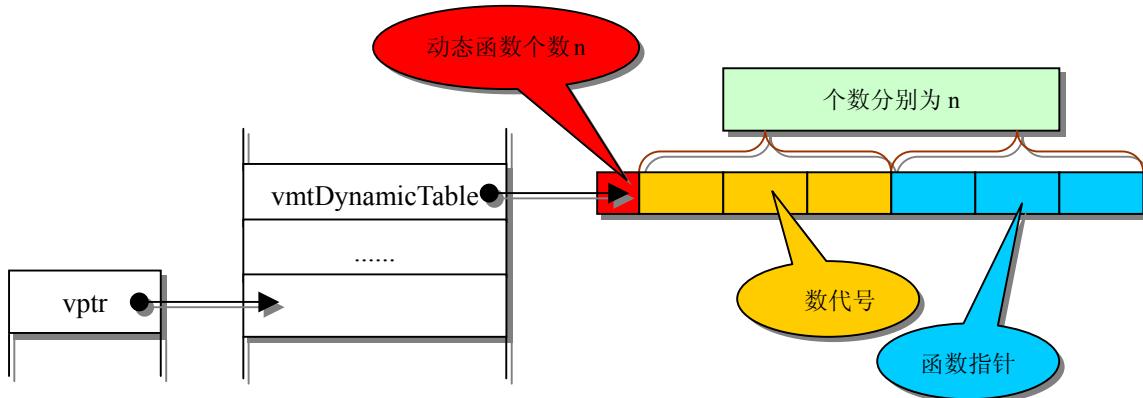


图 3 动态方法表 DMT

看汇编头晕吧？嘿嘿，对着注释看看这个图就清楚了。vmtDynamicTable 所指向的地址，就是一个 DMT，而它的结构，我们前面已经分析过了。唯一需要说明的是

```

ADD    EAX, EAX      ; EAX 值为 n，自加后为 2*n
SUB    EAX, ECX      ; ecx 值已经递减为 0，这句仅仅是清除 ZF 标志位
MOV    EBX, [EDI+EAX*2-4] ;

```

清除 ZF 是因为 `_FindDynaInst` 要由此判断是否找到相应的函数。而 `edi-4` 为函数代号所在的地方，`edi-4+4*n` 即为函数指针所在，也就是 `edi+eax*2-4`。

其实不需要与汇编纠缠，在前面我们已经知道了其原理，大同小异罢了。

结束语

C++的重用性是对源代码级而言，而对二进制级重用性的支持则捉襟见肘。特别是动态连接库 DLL 的广泛运用，更显出解决这个问题的重要性。COM 的口号之一正是 COM as a Better C++⁷。讲 COM 的书中往往指出若干 C++的不足，其实不少是可以解决的。比如

- 问题：不同编译器的名字粉碎机制不同，导致不同编译器编译的模块不能顺利连接。
- 解决：使用 DEF 文件。

- 代价：操作麻烦，增加维护负担，但对程序效率没有任何影响。
- 问题：不同版本的类大小不一样，主要原因是成员变量增加或减少，导致分配空间时出错。
- 解决：隐藏实现，成员变量仅保留一个指针 `void *`，在运行时动态申请空间。
- 代价：可读性和性能均受影响。

添加普通成员函数没有什么大的问题，但是添加虚函数则影响 VFT，可能导致程序错误甚至系统崩溃。解决的办法在前面已经说明，其中良好的设计是必不可少的。建议

- 根类的设计一定要慎重，VCL 从开始至今，`TObject` 类的变化始终很少，否则牵一发而动全身，维护性就大打折扣；
- 类层次应尽可能浅，尽量避免使用继承等耦合性很强的关系，严格遵循 Liskov 替换原则 LSP⁹；
- 如果程序只在 WINDOWS 下运行，可以考虑使用 COM；
- 如果始终使用 Borland 的编译器，并对性能要求不高，可以考虑使用动态（dynamic）函数；
- 多写几个无用的虚函数占位，也是个不错的方法。

动态函数应用在合适的地方，这一点可以参考 VCL 各个类中动态函数的使用情况。另外，动态函数所节约的 VFT 空间微不足道，在有的情况反而 DMT 的空间占得更多。总体来说，动态函数在时间上吃亏，空间上占的便宜也不大。在我看来，解除了父类和子类 VFT 之间的关联性，才是动态函数最大的好处。

不论是辩证唯物主义，还是道家思想，都强调事物的两面性。不论什么方法，都是一把双刃剑，所谓“祸兮福之所倚，福兮祸之所伏”。我们要做的，就是权衡利弊，结合具体的环境，扬长避短。

参考

1. 虫虫.《天方夜谭 VCL：开门》. *C++ View*. 2001, 9.
2. George Shepherd, Brad King. *Inside ATL*. Microsoft Press. 1999.
3. Stanley Lippman. *Inside the C++ Object Model*. Addison-Wesley, Reading, MA. 1996
侯捷.《深度探索 C++ 物件模型》. 暮峰资讯股份有限公司. 1998.
侯捷.《深度探索 C++ 对象模型》. 华中科技大学出版社. 2001.
4. GoF. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, Reading, MA. 1995.
李英军等.《设计模式：可复用面向对象软件的基础》. 机械工业出版社. 2000.
5. CKER.《深入 BCB 理解 VCL 的消息机制》. *C++ View*. 2001, 7.
6. Dale Rogerson. *Inside COM*. Microsoft Press. 1997.
杨秀章.《COM 技术内幕》. 清华大学出版社. 1999.
7. Don Box. *Essential COM*. Addison-Wesley, Reading, MA. 1998.
侯捷.《COM 本质论》. 暮峰资讯股份有限公司. 1999.
潘爱民.《COM 本质论》. 中国电力出版社. 2001.
8. Brent E. Rector and Chris Sells. *ATL Internals*. Addison-Wesley, Reading, MA. 1999.
潘爱民, 新语.《ATL 深入解析》. 中国电力出版社. 2001.
9. Robert C.Martin. “The Liskov Substitution Principle”. *C++ Report*. 1996, 3.
虫虫, plpliuly.《Liskov 替换原则 LSP》. *C++ View*. 2001, 9.

Hotline

回音壁

1. 编辑：

你好！

首先感谢你带给大家这么好的杂志。但是我个人觉得第二和第三期的封面设计不好，当然这点小问题不会对杂志造成大的影响，但作为喜欢该杂志的一名读者，我希望你们能把该杂志做得越来越好。

我是非常喜欢 C++ 的。今后如果有好的 idea，我会写成文章寄来的。

如果有新的一期出来，请及时通知我，谢谢。

回复：封面设计一直是我们头疼的事情，我们非常希望在这方面有特长的朋友鼎力帮忙。

2. 关于“高级” RTTI 的不同声音

C++ View 第3期《天方夜潭 VCL》讨论了 Delphi 的“高级” RTTI。大致有如下特征：

特征 1. `p->ClassName()` 就能返回类 A 的名字“A”

特征 2. `A::ClassName(p->ClassParent())` 就可以返回 A 的基类名“TObject”

事实上，C++ 的 RTTI 机制并非此文所说的那么“低级”。

首先，我们来看看 typeid

```
const type_info& operator typeid(typeid or expression)
```

typeid 返回 type_info 类对象的引用。type_info 声明如下（引自 MSDN）

```
class type_info {  
public:  
    virtual ~type_info();  
    int operator==(const type_info& rhs) const;  
    int operator!=(const type_info& rhs) const;  
    int before(const type_info& rhs) const;  
    const char* name() const;  
    const char* raw_name() const;  
private:  
    ...  
};
```

除了 operator == 和 !=，还有 3 个成员函数

```
int before(const type_info& rhs) const;
```

为 type_info 类型定义全序关系，便于使用关联容器。

注意：在不同程序或编译器间此关系是不同的。

```
const char* name() const;  
返回类名。这就等价于 TObject::ClassName()。
```

```
const char* raw_name() const;  
返回经修饰（decorated）的类名，不适合人阅读。
```

可见特征 1 在 C++ 中已经实现。

再看特征 2。为什么 C++ 没有返回基类的能力？

- a. 多重继承。所以 C++ 不存在 super 关键字。
- b. 真的有必要吗？使用一个类需要了解它的基类吗？这已经违背了 OOP 的初衷。

总之，C++ RTTI 机制相对于 Delphi 或 Java 并不“低级”。新的 Framework 完全可以建立在 C++ 标准的 RTTI 机制上。至于为什么 VCL, MFC, Qt 都有自己的 RTTI 机制，因为 VCL 是用 Object Pascal 写的，而 MFC, Qt 出现都较早，C++ 3.0 才加入 RTTI 支持。

虫虫回复：谢谢您的意见。我完全同意您的看法。事实上我在文章中也是这么说的，也许是我说的不太清楚或者其他什么原因吧。RTTI 一般也就是 Exception-Handling 和 IDE 用。对于动态语言，如 JAVA、OP，当然可以用。C++ 就不必了。

3. 未见 csdn 上的通知，至今日才看到了 C++ View 的第三期。先看了《给虫虫的一封信》，完后，心中澎湃，想与人交流。

恕我简陋，能否介绍一下“透明”。他的一番谆谆教导，倍感亲切。语气甚是恳切，如长者。但他对文学写作态度却不认同。“海若无涯天做岸，山到穷顶我为峰。”没有见到过侯先生写有此诗。好诗，好诗！李白也狂傲，“仰天大笑出门去，我辈岂是蓬蒿人。”但很多人喜欢李白的诗啊。我不知道侯生活中是否如此，他的一篇《深入浅出 MFC》让我看的过瘾，《Inside the C++ Object Model》的翻译也体现了他的治学严谨。个人觉得李傲、侯未达到大师级，算是学者。而学者多狂傲。今未见有大师，只有侯。喜欢侯游山玩水，著书立作。后又有几人能从事自己喜欢的东西。肯定是自己年少轻狂，才说这些话。“待到识尽愁滋味，却道天凉好个秋。”

“大家都有现实的忙碌，有多少人能坚定自己的梦想？”这句话令我如芒在背，喜欢 C++，喜欢知道从哪里来，哪里去。可现实的生活？学业，前途，已令我很少好好的静下心看书。看了 c++View 后，知道自己只不过刚进门而已，知道自己的浅薄。更需要补充。可这条道有前途么？到处充斥着用 vb, jsp, asp 开发数据库的东东，没有人会要 c++ 的，或者很少。自己会坚持么？知道 C++ Object Model 有用么？

“透明”能开设计模式的专栏，非常欣喜。我从网上看到国外有很多学习设计模式的小组。好不容易 download 一个教授“建议”（只是建议）23 个模式的学习顺序，但看的还是一头雾水。后来逐渐看的其中一些模式在一些地方的使用，才算懂得了几个。希望能从这里学到更多。

关于杂志的建议，编者想把杂志办的怎样就怎样吧，一口菜的口味不能调的大家都满意。

至于收费，最好能得到赞助。如果一期收一次费，跑邮局也太累人了。或可以 3,4 期收一次费。
至此，晚安。

回复：非常感谢你的支持。上次侯先生在清华的讲座上为他的那两句话特意作了说明，他当初写这两句话只是为了鼓励别人，并非自我抒怀。

C++ VIEW

第5期
2001年11月号

焦点：

评测 Visual C++ .Net beta2

特稿：

鸟鸣涧

Design Patterns

专栏：

模式罗汉拳——Immutable模式与string类的实现

Object Oriented

阿P正传——创业初期

芝麻开门之Qt——总介

Advanced C++



第 5 期目录 2001 年 11 月

焦点 Focus

回音壁	03	plliuly 做的 C++ View 第 4 期相当不错吧？他现在实在太忙，第 5 期就砸到我头上了。上次由于病毒泛滥，网络实在不行，上传晚了些，实在抱歉，希望大家海涵。
评测 Visual C++. Net beta 2	04	

特稿 Feature

标准 C++本地化	14	这次我们推出一个新栏目“鸟鸣涧”。听说过 Obfuscated C++、Guru of the Week 吗？这就是一个类似的栏目，最近主要提供一些关于 C++标准库的题目，就当作智力体操吧，希望大家能积极参与讨论。本地化是 C++非常重要的一部分，这次《标准 C++本地化》先给大家一个初步的印象。
鸟鸣涧	23	

专栏 Column

Generic<Programming> 简化异常安全代码	24	庆祝三个专栏开张啦！小飞侠带来了专栏“阿 P 正传”，用幽默、实际的例子给我们讲述设计模式。阿 P？阿 P 是谁？当然是阿 Q 他老哥啦 还记得 C++ View 第 1 期中我们介绍的跨平台 C++类库 Qt 吗？这次同步快梭将在“芝麻开门之 Qt”中带我们一起去体会其中的方便与快捷，看看这位构建 KDE 的功臣有多大的能量。另外，OpenGL 在工业界的核心地位，也深深地影响了跨平台 C++ GUI 类库，成为其中必不可少的一环，“芝麻开门之 OpenGL”能给我们关于 OpenGL 基本的印象，领略一下 OpenGL 的强大威力。
C++批评系列 继承的本质	38	
芝麻开门之 OpenGL 设置 OpenGL (一)	40	

模式罗汉拳

Immutable 模式与 string 类的实现	50	.Net 炒得火热，作为 C++迷，当然最关心 VC.Net 啦。这次我们会看到 VC.Net beta 2 对标准支持的评测，让我们能获得一个大概的印象。听说大师 Stanely Lippman 也加盟 VC.Net 小组，前景光明啊:-)
阿 P 正传 创业初期	55	
芝麻开门之 Qt 总介	63	“ Generic<Programming> ” 和 “ C++批评系列 ” 两个专栏依然在各自的领域讨论非常有意思的话题，希望感兴趣的朋友继续关注。

主编：王 曜 主页：<http://cppview.yeah.net>
封面：小飞侠 <http://cppview.v365.com>
编辑：小 Jon 电邮：cppview@sohu.com

导读

plliuly 做的 C++ View 第 4 期相当不错吧？他现在实在太忙，第 5 期就砸到我头上了。上次由于病毒泛滥，网络实在不行，上传晚了些，实在抱歉，希望大家海涵。

这次我们推出一个新栏目“鸟鸣涧”。听说过 Obfuscated C++、Guru of the Week 吗？这就是一个类似的栏目，最近主要提供一些关于 C++标准库的题目，就当作智力体操吧，希望大家能积极参与讨论。本地化是 C++非常重要的一部分，这次《标准 C++本地化》先给大家一个初步的印象。

庆祝三个专栏开张啦！小飞侠带来了专栏“阿 P 正传”，用幽默、实际的例子给我们讲述设计模式。阿 P？阿 P 是谁？当然是阿 Q 他老哥啦 还记得 C++ View 第 1 期中我们介绍的跨平台 C++类库 Qt 吗？这次同步快梭将在“芝麻开门之 Qt”中带我们一起去体会其中的方便与快捷，看看这位构建 KDE 的功臣有多大的能量。另外，OpenGL 在工业界的核心地位，也深深地影响了跨平台 C++ GUI 类库，成为其中必不可少的一环，“芝麻开门之 OpenGL”能给我们关于 OpenGL 基本的印象，领略一下 OpenGL 的强大威力。

.Net 炒得火热，作为 C++迷，当然最关心 VC.Net 啦。这次我们会看到 VC.Net beta 2 对标准支持的评测，让我们能获得一个大概的印象。听说大师 Stanely Lippman 也加盟 VC.Net 小组，前景光明啊:-)

“ Generic<Programming> ” 和 “ C++批评系列 ” 两个专栏依然在各自的领域讨论非常有意思的话题，希望感兴趣的朋友继续关注。

最后，希望所有对 C++ View 感兴趣的朋友都看看《回音壁》，我把它调整到了第一位，希望别扔……

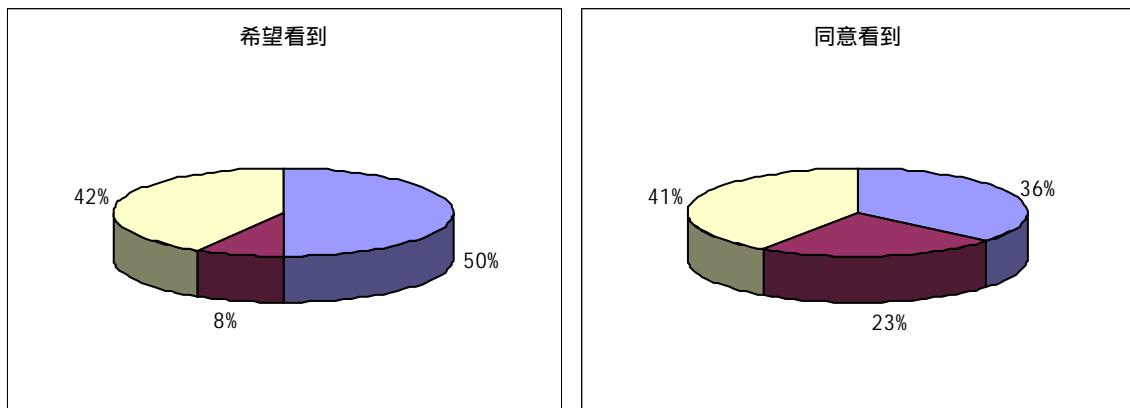
Hotline

回音壁

大家好，我是 Jonathan，叫我 Jon 就行了。

首先要感谢同步快梭为我们提供了一个稳定快速的空间，C++ View 把“家”都搬过去啦。他的软件 AutoSyncFTP (<http://www.joyinternet.net>) 也实在太爽了。想第 4 期 C++ View 传了足足整整一下午，传得虫虫都要从一楼跳到四楼了。这次“搬家”全部文件才不到半分钟，福气啊。顺便花了些时间把主页改了改，希望大家能喜欢。

这次用 email 通知大家第 4 期发布时，同时做了一次调查，特把结果公布于下。



其中红色、黄色、蓝色分别代表 OpenGL/DirectX、COM/ATL/WTL 以及 STL。我们会依此做出适当的安排，谢谢大家的关注。

由于网络原因，发给三位朋友 (liqinhit、KAIHUA.QI、qyice) 的 email 被退回，非常抱歉，希望能提供一个新的信箱。另外如果有朋友发 email 而没有得到回应的，请再发一次。

有些朋友投的文章，排版特别累，所以这次把投稿文章的一些要求给大家说一下，希望大家能够理解。

1. 请尽量使用 Microsoft Word 格式；
2. 中文的字体不限，但不论中英文，尽量使用标准字体，以免别人看不到相应的效果；
3. 英文一定不要使用中文字体，如 this 不好看，最好是 this；
4. 请用不同的字体区分同代码相关和无关的英文，如 Courier New 与 Times New Roman。
5. UML 图请用 SmartDraw 绘制，以 OLE 方式粘贴，以解决图片模糊的问题。

11 月 18 日大家看了狮子座流行雨吗？许了什么愿呢？祝愿，天下，好人，好梦。

评测 Visual C++ .Net beta 2

vcmfc

由于无意中说出自己已经安装了 VC.NET beta 2，加上我也是一名 C++ fan，所以虫虫请我测试一下 VC.NET 对 C++ 标准的支持度。盛情难却之下答应了虫虫，可是却遇到一个问题：“用哪些测试数据较有权威性呢？”头痛了几天后，在虫虫的提醒下，决定采用其建议：蓝本就是侯捷先生网站上的一篇关于各种编译的测试报告《C++ Primer 答客问 (27) 【标准与实作之间】PC 环境上三种编译器对 C++ Standard 的支援》，最后我进行总结（非常粗糙）。最后我要提的是：C++ 大师，C++ Primer、Inside The C++ Object Model 及 Essential C++ 等书的作者 Stanley Lippman 加盟 Visual C++ .Net 开发小组，相信 Visual C++ .Net 的正式版会更激动人心。

言归正传，由于侯捷的蓝本大部分其中在 Standard C++ 语言身上，对于 STL 的部分相当的少，以致于我也没有进行这方面的测试（因为我就懂那么一丁点 STL）。还有一点，那就是测试太码太长了，无法全部在下面进行罗列，请大家参考 <http://www.csdn.net/expert/jihou/qa-cpp-primer-27.txt>（简体中文）或 <http://www.jihou.com/qa-cpp-primer-27.txt>（繁体中文）中详细的代码。以下几个方面进行总结：

一些常见 BUG 的修复：

- C++ Primer p411

主题：string* 的建构（直接指定以另一个 string*）

测试结果：VC6[x] BCB4[o] G++[o] **VC7[]**

```
#include <string>
using namespace std;
int main()
{
    string *pstr_type = new string( "Brontosaurus" );
    string *pstr_type2( pstr_type ); // <== VC6 error.
    delete pstr_type;
    delete pstr_type2;
}
```

- C++ Primer p643 中

主题：直接在类声明中对 const int 成员变量赋初值

测试结果：VC6[x] BCB4[o] G++[o] **VC7[]**

```
class Account {
public:
```

```

    static const int namesize = 16; // <== 类声明中初始化
};

const int Account::namesize;

```

- C++ Primer p904

主题：using declaration 可将基类中的同名成员函数放进子类的范围中

测试结果：VC6[x] BCB4[o] G++[x] VC7[]

但如果将下例的 using declaration 移到 Shy::mumble() 之前，则 VC6[o]

```

#include <iostream>
#include <string>
using namespace std;

class Diffident {
public:
    void mumble (int) { ... };
};

class Shy : public Diffident {
public:
    void mumble(string) { ... };
    using Diffident::mumble;
};

```

函数模板方面：

- 6.p.492, p.499, p.500

主题：以模板（非类型）参数作数组大小

测试结果：VC6[x] BCB4[o] G++[o] VC7[]

```

template <class Type, int size>
Type min( const Type (&r_array)[size] )
    // VC6 error C2057: expected constant expression
{ /* ... */ }

void main()
{
    int ia[5]={40,20,49,17,28};    // G++ 要求需为 const int ia[5]。
    min(ia);

```

```
}
```

- C++ Primer p500

主题：利用转换运算符，以某特定类型具现化（instantiated）模板函数。

测试结果：VC6[x] BCB4[x] G++[x] **VC7[X]**

```
template <typename Type, int size>
Type min( Type (&r_array)[size] ) { /*... */ } // VC6 error C2057

typedef int (&rai)[10];           // rai：“10个int组成的数组”的引用
typedef double (&rad)[20];        // rad：“20个double组成的数组”的引用

// overloaded functions
void func( int (*)(rai) ) { };    // int(*)(rai) 是函数指针类型,
                                    // 该函式的参数类型是 rai。
void func( double (*)(rad) ) { }; // double(*)(rad) 是函数指针类型,
                                    // 该函式的参数类型是 rad.

void main()
{
    func(static_cast<double(*)(rad)>(&min)); // (1) 此行无法编译
    // BCB4 E2335: Overloaded 'min' ambiguous in this context
    // G++: undefined reference to 'func(double (*)(double (&)[19]))'
}
```

解决之道：绕个道，就可以。将上述（1）：

```
func(static_cast<double(*)(rad)>(&min));
```

改为以下即可：

```
double(*fp)(rad) = &min; // instantiate 'min', using specified type.
// VC7 出错信息：error C2440: 'initializing' : cannot convert from
// 'overloaded-function' to 'double (__cdecl *)(rad)'
func(fp);
```

- C++ Primer p507

主题：指定函数模板部份参数的类型，另一部份由编译器推得。

测试结果：VC6[x] BCB4[x] G++[o] **VC7[]**

```
template <class T1, class T2, class T3>
```

```

T1 sum( T2 v2, T3 v3)
{ return T1(v2+v3); }

typedef unsigned int ui_type;

ui_type calc( char ch, ui_type ui )
{
    // 以下明白指定 T1 为 ui_type ,
    // T2 被编译器推导为 char , T3 被推导为 ui_type。
    ui_type (*pf)( char, ui_type ) = &sum< ui_type >;
    ui_type loc = (*pf)(ch, ui);
    return loc;
}

void main()
{
    calc('c', ui_type(1024));
}

```

- C++ Primer p508

主题：指定函数模板参数类型

测试结果：VC6[x] BCB4[x] G++[o] **VC7[X]**

```

template <class T1, class T2, class T3>
T1 sum( T2 op1, T3 op2 ) { /* ... */ return T1(10); }

void manipulate( int (*pf)( int,char ) ) { };
void manipulate( double (*pf)( float,float ) ) { };

void main( )
{
    manipulate( &sum< double, float, float > );
}

//VC 7 出错信息：
//error C2668: 'manipulate' : ambiguous call to overloaded function

```

异常处理:

- C++ Primer p554

主题：function try block

测试结果：VC6[x] BCB4[x] G++[o] VC7[]

```
#include <iostream>
using namespace std;

class popOnEmpty { /* ... */ };
class pushOnFull { /* ... */ };

int main()
try {
    throw popOnEmpty();
    throw pushOnFull();
    return 0;
}
catch ( pushOnFull ) {
    cout << "catch pushOnFull" << endl;
}
catch ( popOnEmpty ) {
    cout << "catch popOnEmpty" << endl; // <-- 执行结果：此行。
}
```

- C++ Primer p564

主题：exception specification

测试结果：BCB4 表现佳，G++ 尚可，VC6 粗糙 VC7[]

```
#include <iostream>
using namespace std;

class popOnEmpty { /* ... */ };
class pushOnFull { /* ... */ };

void func1() throw (pushOnFull)
{
    throw popOnEmpty(); // BCB4 warning: Throw expression violates
                        //      exception specification in function
                        //      func1() throw(pushOnFull)
                        // VC6 : no error, no warning
                        // G++ : no error, no warning

    throw pushOnFull(); // BCB4 Warning : Unreachable code in function
                        //      func1() throw(pushOnFull)
                        // VC6 : no error, no warning
}
```

```

        // G++ : no error, no warning
}

int main( )
{
    try {
        func1();
        return 0;
    }
    catch ( pushOnFull ) {
        cout << "catch pushOnFull" << endl;
    }
    catch ( popOnEmpty ) {
        cout << "catch popOnEmpty" << endl;
    }
}

```

Friend:

- C++ Primer p861

主题：类模板部分特化

测试结果：VC6[x] BCB4[o] G++[o] **VC7[X]**

```

#include <iostream>
using namespace std;

// form 1
template <class T, int hi, int wid>
class Screen {
public:
    void print() { cout << hi << ' ' << wid << " form1" << endl; }
};

// form 2 (template partial specialization)
template <class T, int hi>
class Screen <T, hi, 80> {
public:
    void print() { cout << hi << " form2" << endl; }
}; // VC6 error C2989

```

```

// form 3

template <class T, int hi>
class Screen <T*, hi, 25> {
public:
    void print(){    cout << hi << ' ' << sizeof(T*) << ' '
                    << sizeof(T) << " form3" << endl; }
};

int main()
{
    Screen<int, 100, 40> s1;
    Screen<int, 100, 80> s2;
    Screen<int, 500, 25> s3;
    Screen<char*, 300, 25> s4;
    Screen<double*, 400, 25> s5;

    s1.print(); // output: 100 40 form1
    s2.print(); // output: 100 form2
    s3.print(); // output: 500 25 form1
    s4.print(); // output: 300 4 1 form3
    s5.print(); // output: 400 4 8 form3
    return 0;
}

```

- C++ Primer p1090

主题：friend operator<<

测试结果：VC6[x] BCB4[o] G++[o] VC7[]

注意：如果使用<iostream.h>而非<iostream>，在 VC6 中运用 friend 就没有问题。但如果这么做的话，由于下面用到<string>，一定得 using namespace std; 而这在 VC6 中似乎导致暗中含入<iostream>，於是与<iostream.h>起冲突。VC6 在这主题上表现不佳。

```

#include <iostream>
#include <string>
using namespace std;

class WordCount {
    friend ostream& operator<<(ostream&, const WordCount&);
public:
    WordCount( string& word, int cnt=1 ): _word(word), _occurs(cnt)
    { };
private:
    string _word;
    int _occurs;

```

```

};

ostream& operator <<( ostream& os, const WordCount& wd )
{ // format: <occurs> word
    os << "< " << wd._occurs << " > " << wd._word; // VC error!
    return os;
}

void main()
{
    string s1("Hello");
    string s2("jjhou");
    WordCount w1(s1, 5);
    WordCount w2(s2, 7);
    cout << w1 << endl; // < 5 > Hello
    cout << w2 << endl; // < 7 > jjhou
}

```

STL:

- 20.C++ Primer p1156~p1157

主题：泛型算法 `max()` 和 `min()` **VC7[X]**

测试结果：VC6 所附的 STL 竟未支援 `max` 和 `min` 这两个泛型算法。

```

#include <iostream>
#include <string>
using namespace std;

template <typename T>
T min( T v1, T v2) // <-- note
{
    return (v1 < v2 ? v1 : v2);
}

class Rect {
    friend ostream& operator<<(ostream& os, Rect& r);
public:
    Rect(int i) : _i(i) { }
    bool operator<(Rect& rhs)
        { return (this->_i < rhs._i ? true : false); }
}

```

```

private:
    int _i;
};

ostream& operator<<(ostream& os, Rect& r)
{
    os << r._i;
    return os;
}
void main()
{
    cout << min( 17, 15) << endl;           // 15
    cout << min(13.5, 13.57) << endl;        // 13.5
    cout << min('a', 'e') << endl;          // a
    cout << min("jjhou", "allan") << endl;   // allan

    Rect r1(6), r2(3), r3(9);
    cout << r1 << r2 << r3 << endl;        // 639
    cout << min(r1, r2) << endl;            // 3
}

```

此程序在 VC6 中失败 ,但如果改为 #include<iostream.h>并移除 using namespace std; 则成功 ,这是 VC6 的 bug !(见先前对 friend 的讨论 ,C++ Primer p1090) [VC7:编译不通过 ,采用 VC6 的方法却可以 ,跟 VC6 是一样的。](#)

- C++ Primer p.412 中下

主题 : auto_ptr 的 reset() 操作

测试结果 : VC6[x] BCB4[o] G++[x] [VC7\[\]](#)

```

#include <memory> // for auto_ptr
using namespace std;
int main()
{
    auto_ptr<int> p_auto_int; // <== G++ error. 见前述 p.410 例
    p_auto_int.reset(new int(1024)); // <== VC6 and G++ error
}

```

[一些小细节 :](#)

- C++ Primer p213 下 , p.393 下

主题 : for 循环区域内初始化的对象皆为本地 (local) 对象。

测试结果 : VC6[x] BCB4[o] G++[o] **VC7[]**

```
void main()
{
    for (int ival=0; ival< 10; ++ival);
    for (int ival=0; ival< 10; ++ival);
}
```

- C++ Primer p688

主题 : 本地 (local) 类

测试结果 : VC6[x] VB4[x] G++[x] **VC7[X]**

```
int a, val;
void foo( int val )
{
    static int si;
    enum Loc { a = 1024, b };
    class Bar {
public:
    Loc locVal; // ok;
    int barVal;
    void fooBar( Loc l = a ) { // ok: Loc::a
        barVal = val; // error: local object
        barVal = ::val; // ok: global object
        barVal = si; // ok: static local object
        locVal = b; // ok: enumerator
    }
};

// ...
}

int main()
{
    foo(5);
}
```

从整体的来看 , VC.NET 主要改进集中在模板的各个方面 , 以更好地支持 STL。当然 , 其它的
小细节也进行大量的修复 , 从以上测试也可以看得出来。虽然对模板的支持依然不理想 , 看来它也
是一个相当不错的编译器。

标准 C++本地化

Nathan C. Myers

翻译：张岩

编者按：本地化是标准 C++ 极重要的部分，用以提供国际化支持，这也是 C++ 标准化的重要成果之一。本文作者是本地化的设计者 Nathan Myers，其权威性不言而喻。非常感谢 Nathan Myers 先生对本刊的授权，也感谢张岩的精彩翻译。本文的英文原文可在 <http://www.cantrip.org/locale.html> 找到，有兴趣的朋友不妨参考一下。

当用户所使用的语言和你的代码使用的语言相同时，不用花费许多力气就可以让程序在菜单、日期时间、消息和有序表上以正确的语言显示。然而，如今的软件传播范围愈来愈广，从而可能使得你的代码和用户之间只剩下在机器语言上是相同的。如果你的程序能够供使用各种语言的用户使用，那无疑会带来很好的市场前景。

程序运行时系统中的地区，语言等环境信息以及用户与此相关的设置被称为程序的本地化属性（locale）。分离本地化部分和程序代码从而使它们更容易被修改的工作，称为国际化（internationalization）。C 语言中，locale 只描述了整个国家的共性，但 C++ 的 locale 更为灵活。

挑战

为了支持国际化，C++ 标准库面临许多挑战。因为服务器程序可能会在整个世界上有许多客户端，所以标准库必须在同一个程序中支持一个以上的 locale。因为你的程序可能要使用多线程（或者即将使用），所以标准库一定是可重入的。因为用户的文化和偏好是不可预测的，所以它一定需要可扩展性。因为你也有可能不需要或目前还不需要考虑到 local 的问题，所以它必须是可以忽略的。最后，因为它是标准，所以它必须高效、易用、安全。

在标准库中，为了应对所有这些需求，解决方法使用了所有语言的能力，包括一些未被所有编译器实现的新标准特性。本文将介绍 C++ 的 locale 库的使用方法。也指出了它的可能的实现方法，所以，在你自己的代码中也可以使用同样的技术。

Locale 对象

在 C 中的 locale 标准（不是 C++ 中的）描述了字符的编码（连同排序规则）和少许的几个数值类型的格式：数字、日期和货币。这些描述只不过是真实文化和个人选择中的沧海一粟罢了；其它一些需要经常使用的包括时区、度量系统、纸张尺寸、窗口颜色及字体、公民身份、笺头、e-mail 地址、性别、和鞋号。不可能有标准来规范所有的个人偏好。C++ 只提供了（如例子所示）与传统 C 语言 locale 库中提供了相同的范畴——不同的是，你可以扩展它。

理解标准 C++ locale 库的关键是 facet。facet，指的是一个可以从一个 locale 对象中获得的类接口。例如，C++ 标准库中的 facet num_put<> 构成了数值类型，collate<> 提供了字符串值的顺序。每个 facet 也是 locale 中的一个对象。每一个 locale 对象都会包含着一系列的 facet 对象来提供这些服

务。

C++ locale 是一个十分简单的对象，并且可以像内建类型被（高效地）传递，复制，赋值。接受 locale 作为参数的函数能够声明一个默认值，`locale()`——当前全局 locale 的一份拷贝；这可以允许函数省略这一参数，获得合理的行为。每一个 `iostream` 实例都有一个 `locale`，需要在 `operator <<` 和 `operator >>` 中使用。这些方法使得 `locale` 以一种低姿态形式出现，从而避免它们在不需要这些强大特性的地方出现。

一个 Date 类的例子

这些在代码中会以如何形式出现呢？Example 1 是一个简单的 Date 类。

Example 1: A simple Date class

```
//file: date.h
#include <iostream> // for istream, ostream
#include <ctime> // for struct tm

namespace ChronLib {
    class Date {
        long day; // days since 1752-09-14
    public:
        enum Month { jan, feb, mar, arp, may, jun,
                    jul, aug, sep, oct, nov, dec };
        Date(int year, Month month, int day);
        void asCLibTime(struct std::tm*) const;
    };

    std::ostream&
    operator << (std::ostream& os, Date const& date);

    std::istream&
    operator >> (std::istream& is, Date& date);
}
```

不要为那些 `std::` 符号烦恼；C++ 标准库的所有组件都是 `std` 命名空间中的成员。新的标准头文件 `<iostream>` 和 `<ctime>` 声明了例子中的标准名字，`::` 符号给出对这些成员的访问方法。

还没有任何关于 `locale` 和 `facet` 的东西出现；只有一个构造函数，流操作符和一个成员函数 `asCLibTime()`，用其它库填充标准 C 库中的 `struct tm` 以作通信使用。（查看 `strftime()` 手册来获得更多关于 `struct tm` 的信息）这样，你不需要了解任何关于 `locale` 在 Date 类中的使用的事情。

用户对希望见到的日期格式会在世界地图上变化。如果将格式在 `operator >>` 和 `<<` 中进行编

码的话，就会令许多用户不满。取而代之，对这些操作符进行编码的时候，可以将格式化的工作委托给流的 locale 对象。例 3 中将会对这个进行详细叙述。

一个实例程序

用户怎样控制一个用 locale 技术的 operator <<产生的格式呢？Example 2 是最简单的一种可能情况。

Example 2: A program that uses your preferred locale

```
#include <iostream> /* for cout */
#include <locale> /* for locale */
#include "date.h"

int main()
{
    using namespace std;
    using ChronLib::Date;
    cout.imbue( locale("") );
    cout << Date(1942, Date::dec, 7) << endl;
    return 0;
}
```

新的标准头文件，`<iostream>`和`<locale>`，声明了在这里被使用的 `std::locale`、`std::cout` 和 `std::endl`，“`date.h`”是在例 1 中被声明。以 `using` 为开头的代码允许在函数后面代码省略 `std::`。

构造函数调用 `locale("")` 来创建一个 `locale` 对象来表示用户偏好。标准并没有说明这意味着什么，但是，在许多系统上，库会用任何能找到的环境变量（经常 `LANG` 或者 `LC_ALL`）是来替换空字符串。对于 American locale 为例，通常的名字，是“`en_US`”。（在 POSIX 系统上，可以键入“`locale -a`”来列出所有支持的 `locale`。）

对 `cout.imbue()` 的调用在 `cout` 中安装了新建的 `locale`，以待不同的 `operator <<` 使用。下一行使用了 `Date` 的 `operator <<`（在例 1 中声明），它将工作委托给从 `cout` 中得到的 `locale` 中的 facet。

使用 Facet

使用 `locale` 的 facet，需要调用`<locale>`中的全局模版函数 `use_facet<>()`。Figure 1 给出了它的声明。

Figure 1: The standard template `use_facet<>()`

```

namespace std {
    template <class Facet>
        Facet const& use_facet(locale const& loc);
}

```

例如，对于一个具有成员函数 `shoeSize()` 的 facet 类 `stats`，和一个叫做 `loc` 的 locale 对象，调用是如下形式：

```
use_facet<Stats>(loc).shoeSize();
```

这种调用的语法被称为函数模板，需要为模板显示的提供类型参数而不是编译器通过参数进行推断 (deduce)，目前，还不是所有编译器都可以支持此特性；这叫做“显示模板函数限定”(explicit template function qualification)。这与新的类型转换的语法很相似，比如 `dynamic_cast<>`，并且，事实上 `use_facet<>()` 是一种安全的转换。在上面的例子中，返回的引用立即调用了 `locale` 对象 `loc` 中储存的 `Stats` 实例的 `Stats::shoeSize()` 成员函数。

一个示例的 operator <<

Example 3 是 `Date::operator <<` 的完整实现，这其中使用了标准的 `facet time_put<char>`。

Example 3: Operator<< for class Date

```

// date_insert.C
#include <ctime> /* for struct tm */
#include <ostream> /* for ostream */
#include <locale>
#include "date.h"

namespace ChronLib {
    std::ostream&
    operator<<(std::ostream& os, Date const& date)
    {
        using namespace std;
        ostream::sentry cerberus(os); //1
        if(!bool(Cerberus)) return os; //2
        struct tm tmbuf; date.asCLibTm(&tmbuf) //3
        time_input<char> const& timeFacet=
            use_facet< time_put<char> >(os.getloc()); //4
        if(timeFacet.put(
            os, os.os.fill(), &tmbuf, 'x').failed()) //5
            os.setstate(os.badbit);
        return os; //6
    }
}

```

```
}
```

这有点让人眼花缭乱。标志//1 和//2 两行代码创建并检测一个标准“ ostream::sentry ”对象。（这是标准库中的一个新类；它的构造函数为 ostream 准备输出。在多线程环境中，它可能会锁定流）。行//3 用参数 date 的内容填充了局部变量 tmbuf。

有意思的在后面：在行//4 中， os.getloc() 获得一个 ostream 参数 os 中的 locale 对象，对 use_facet<>() 的调用会得到那个 locale 对象中的标准 facet time_put<char> 的一个引用。在//5 中，对 time_put<char>::put 的调用将字符写出到流 os 并且返回一个值来报告错误。（现在无需对 put 的参数过分计较。）行//6 析构了 locale::sentry 对象（可能对流解锁）并将流 os 返回。

想想这意味着什么。头文件 “ date.h ” 中并没有提及 locale ，但是在 operator >> 中隐藏的代码，在 main() 中的几行代码，就可以在世界任何一个地方正确的格式化 date。（如果 main() 中没有这些，获得的将是默认的 C locale 行为）

自己的 Facet

标准的 facet 被设计成为可以被派生来获得改进的 locale 特性。派生的 facet 会继承 facet 基类的接口，但是可以通过改写（ override ）虚成员函数来改变它的行为。

派生不是扩展 locale 的唯一途径。可以编写自己的 facet，并建立一个 locale 来维护它。 Example 4 就是前面提到的那个简单的 Stats facet。

Example 4: The sample Stats facet

```
// stats.h
#include <locales>
class Stats : public std::locale::facet {
public:
    static std::locale::id id;

    Stats(int ss) : shoeSize_(ss) {}
    int shoeSize() const { return shoeSize_; }
private:
    Stats(Stats&); //not defined
    void operator = (Stats&); //not defined

    int shoeSize_;
};

// stats.C
#include "stats.h"
std::locale::id Stats::id;
```

是什么让 Stats 类成为一个 facet 呢？它从 locale::facet 继承，有一个 locale::id 类型的公有静态成员变量 id，并且它的成员函数被声明为 const。就这么多。它不需要缺省构造函数，拷贝构造函数，或者赋值操作符。（它们在这里被声明的原因是确保调用他们的人会得到一个编译期间的错误。）

一个 facet 类实例只在作为 locale 的一部分时才有效。Example 5 给出了在 locale 中安装一个 facet 实例的方法。

Example 5: Using the Stats facet

```
locale aLocale( locale(), new Stats(48) );
int s = use_facet<Stats>(aLocale).shoeSize();
```

第一条语句创建了一个称为 aLocale 的 locale 对象作为全局 locale 的一份副本，并在其中加入了一个新创建的 facet Stats。（在一个真正的程序中，很可能是在文件中得到 Stats 的构造参数。）它用到了 locale 类的一个模板构造函数（见 Figure 5），它会从指针参数推断出 facet 的类型。（这是由模板构造函数支持的，至于模板成员，是最近加入语言并没有被所有编译器实现的特性。）locale 会接管 facet 对象的所有权，这样就不需要 delete 也不会产生内存泄漏。第二条语句，和前面的例子一样，给出了它的使用方法。

标准的 facet 最为实用，这样就无需准备，或者让用户准备语言的数据文件。任何新建的实用 facet 都可以被发布，并独立于 C++ 标准而标准化；这样，对于每种语言的数据文件就可以收集到一起并在 internet 上共享给需要它们的人。

底层机制

这些到底是如何运作的？它完全可以被普通的 C++ 实现。你完全可以在自己的程序中使用相同的技术。

首先，locale 对象可以被高效拷贝构造和赋值的原因是它只包含了一个指针，如 Figure 2 所示。

Figure 2: Standard C++ locale implementation

```
class locale {
public:
    class facet;
    class id;

    ~locale()
        {if (imp_>refs_-- == 0) delete imp_; }
    locale()
        : imp_(__globle_imp){++imp_>refs_;}
    locale(locale const& other)
```

```

        : imp_(other.imp_)  {++imp_->refs_;}
template < class Facet >
locale( locale const& other, Facet* f);
explicit locale(char const* name);
//other constructors
locale& operator = (locale const& l);

template < class Facet >
friend Facet const& use_facet(locale const& )

private:
    struct imp{
        size_t refs_;
        vector< Facets* > facets_;
        imp(const imp&);
        ~imp();
    };
    imp* imp_;
};


```

(只有在前面例子中用到的成员才在这里列出来。) 唯一的一个 non-const 的成员函数就是赋值操作符，这样，所有的副本都可以共享同一个用成员指针 imp_ 指出的“实现向量”(implementation vector)。拷贝构造函数复制 imp_ 并增加了引用计数；缺省构造函数用同样的方法复制了全局实例的一个指针。(这一定义用到了一个尚未提到的新语言特性：从“char const*”构造的构造函数被声明为“explicit”以确保编译器不会用这个函数做隐式转换。)

Facet 基类，locale::facet，正如下面 Figure 3 所示，也用到了引用计数。

Figure 3: locale::facet base class definition

```

class locale::facet {
    friend class locale;
    friend class locale::imp;
    size_t refs_;
protected:
    explicit facet(int refs = 0);
    virtual ~facet();
};


```

它有一个虚析构函数，这样，当计数器为 0 的时候，locale 可以安全的析构所有继承自它的类实例。(这个定义依赖于另一个新近加入语言的特性：嵌入类能够在他被包含的类的外部定义。)

注意到例子中的一个约定俗成：引用计数为零暗示着只有一个单独的引用。这让任何一个静态实例在任何静态构造函数执行之前拥有一个初始的计数。这一特性十分有用，虽然这在代码中并没有体现出来。

有被体现。

唯一有些花招的地方就是在 Figure 4 中的 `locale::id` 类中。

Figure 4: locale::id class definition

```
class locale::id {
    friend class locale;
    size_t index_;
    static size_t mark_;
};
```

回想到每一个 facet 类型都包含一个静态的 `locale::id` 类型的成员。这样，每一个 facet 类型中都含有一个静态实例。缺省构造函数 `id()` 并没有初始化成员 `index_`。因为，“静态构造函数”的调用时间是随机的，它们可能在实例被使用后才被调用，所以，不依赖构造函数的初始化是相当重要的。每一个静态实例的成员都在程序装载的时候确保初始化为零，直到它们被设置为其它的值。成员 `index_` 是在什么地方被设定的呢？

Figure 5 给出了 Example 5 中使用过的 `locale` 模板构造函数的定义。

Figure 5: locale template constructor

```
template < class Facet >
locale::locale (locale const& other, Facet* f)
{
    imp_ = new imp( *other.imp_);
    imp_->refs = 0; // one reference
    size_t& index = Facet::id.index_;
    if (!index)
        index = ++Facet::id.mark_;
    if (index >= imp_->facets_.size() )
        imp_->facets_.resize(index+1);
    ++f->facet::refs_;
    facet*& fpr = imp_->facets_[index];
    if (fpr) --fpr->refs_;
    fpr = f;
}
```

构造函数开始的时候复制了 `other` 的实现向量 (implement vector)，这增加了所有 facet 的引用计数。然后，它设置 `Facet::id.index_` 来给 facet 一个是否被用过的标识，如果必要的话，增长 vector 来适应。这样，`id::index_` 成员在真正使用之前是零，只有当 `locale` 对象包含的 facet 拥有它的所有权的时候，它才会被认为是被使用过。(这代码并不是线程安全的；线程安全的版本要不容易理解一些，但是也很相似。)

注意到，这个模板构造函数，和 `use_facet<>()`，只在 `Facet` 参数真正符合为一个 facet 的需要时候才可以实例化；否则，会得到一个编译或者连接错误。因此，库增强了它的接口需求。

函数模板 `use_facet<>()`

在 Figure 1 中被声明，并在几个例子中调用的模板 `use_facet`，在 Figure 6 中定义。

Figure 6: The Function Template `usefacet<>()`

```
template < class Facet >
    inline Facet const&
    use_facet(locale const& loc)
{
    size_t index = Facet::id.index_;
    locale::facet* fp;
    if ( index >= loc.imp_->facets_.size() || 
        (fp = loc.imp_->facets_[index]) == 0 )
        throw bad_cast();
    return static_cast< Facet const& >(*fp);
}
```

如果 `facet` 没有被指定其 `id`，或者它的实例（或者派生实例）出现在 `locale` 的参数中，`use_facet<>()` 会抛出一个异常。（这里的检测方法很有技巧性：如果 `index` 比 `vector` 的长度大，或者偏移 `index` 处的指针为 0，这样 `facet` 就不存在；在偏移 0 处的指针，对应一个未初始化的 `facet` 的 `index`，其值总是为 0。）

我省略了赋值操作符和析构函数，因为它们没有什么价值。我也省略了从字符串构建 `locale` 的构造函数（Example 2 中所示的），因为它不适合作为杂志文章的内容。

总结

C++ 标准 `locale` 库提供了比在这儿呈现的更丰富的内容。尽管如此，最有意义的 `facet` 仍待设计。C++ 标准委员会正在对标准的制定进行收尾；这些留给在 POSIX 上和 internet 上的兴趣小组中工作的人们，让阻碍在每个交互程序中拥有“preference”菜单的部件得以标准化。也许，最迫切的需求莫过于一个能够校验 internet 时区数据库版本的标准时区 `facet`。

鸟 鸣 涧

编者按：从这一期开始，我们推出了“鸟鸣涧”栏目，为老鸟和菜鸟们提供一个争鸣的地方。我们将刊出一系列与 C++ 标准库有关的题目。不论您有多少把握，能做出多少题，甚至哪怕只有一点点想法，都请把您宝贵的思想写成 email，发到 cppview@sohu.com 或 cppview@263.net（如果几天内没收到回信，可能是网络原因造成邮件丢失）那里有饥饿的***等着呢。如果您能提供题目，或者有什么好点子，都请您告诉我们，好吗？

小时候常听孔融让梨的故事，还好他让的不是苹果，所以每次吃苹果时，我都能心安理得，挑个最大的。后来上学了，一考试自然就会有成绩，有分数。出分数的时候，忙着冲上去看谁的最高，然后敲诈他或她请客。说不定以后看到一群 MM，也在心里选个最漂亮的，作为……:-)

现在回头仔细想想这些生活中的小事，我们究竟是如何完成的呢？最关键的一步，就是依次比较每一个苹果、分数、MM……那么，这样的事物必须有可比性（comparable）。可比性，就是能在一群东西中选出最棒一个的约束条件之一。

1、下面这个程序中的 T，需要哪些约束条件？

```
template <class T>
T Max(T* first, T* last)
{
    if (first == last)
        return *first;
    T* r = first;
    while (++first != last)
        if (*r < *first)
            r = first;
    return *r;
}
```



人闲桂花落

夜静春山空

月出惊山鸟

时鸣春涧中

2、由上面得出的约束条件，判断原生类型（如 int、char *、bool 等内建类型）中哪些不能作 T；

3、写一个类 MyClass，能满足以上的约束条件；

4、考虑一下如何修改 Max，使得我们刚才写的类 MyClass 不能正常工作，但其余所有满足以上约束条件的类或原生类型都正常工作。

Generic<Programming>：简化异常安全代码

Andrei Alexandrescu & Petru Marginean

翻译：ye_feng

编者按：这是 Generic<Programming>系列中又一篇极好的文章。资源管理一直是编程中挥之不去的阴影，而其中最常见的就是资源泄露问题。以 Java 为代表的语言引入了垃圾收集(Garbage Collection)的机制，然而也只能勉强算解决内存泄露的一个方向，而更广义的资源同步问题，又如何是好？有没有更好的解决方案呢？这，就是本文的话题。感谢 Andrei Alexandrescu 先生授权，本文原文请见 <http://www.cuj.com/experts/1812/alexandr.htm?topic=experts>，有兴趣的朋友不妨参考一下。

尽管有点自卖自夸，我还是要在一开始就告诉你，这篇文章里有精彩内容。因为我说服我的好朋友 Petru Marginean 和我合作写这篇文章。Petru 开发了一个对处理异常很有用的库。我们一起改进其实现，由此我们得到一个精炼的库，在特定的情况下，它可以大大简化异常安全代码的编写。

在有异常的情况下要写出正确的代码不是一件容易的事，让我们一起来面对它。异常建立了一个单独的控制流，它和应用程序的主控制流几乎没有关系。要了解异常的控制流需要一种不同的思维方式，并且需要新的工具。

➤ 写异常安全的代码是困难的：一个例子

比如说你正在开发一个现在时髦的即时消息服务器程序。用户可以登录和注销，并且可以互相发送消息。你有一个服务器端的数据库保存用户信息，并且在内存里记录已登录的用户。每个用户可以有好友列表，这个列表同时在内存里和数据库里保存。

当一个用户增加或者删除一个好友时，你需要做两件事：更新数据库，并且更新内存中那个用户的缓存。就这么简单的一件事。

假设你的模型里每个用户的信息用一个叫 User 的类来表示，用户数据库用 UserDatabase 类表示。增加一个好友的操作看起来就象下面这样：

```
class User
{
    // ...
    string GetName();
    void AddFriend(User& newFriend);
private:
    typedef vector<User*> UserCont;
    UserCont friends_;
```

```

        UserDatabase* pDB_;
    };
void User::AddFriend(User& newFriend)
{
    // Add the new friend to the database
    pDB_->AddFriend(GetName(), newFriend.GetName());
    // Add the new friend to the vector of friends
    friends_.push_back(&newFriend);
}

```

令人吃惊的是，只有两行的 `User::AddFriend` 里隐藏了一个致命的错误。在内存用尽的情况下，`vector::push_back` 会通过抛出异常来表示操作失败。在那种情况下，你最终只把好友加到数据库里去，但是没有加到内存信息里。

现在我们遇到了问题，是吗？在任何情况下，缺少信息一致性是危险的。很可能在你的应用程序里的很多地方都假设数据库里的信息和内存里的是同步的。

一个简单的解决方法是考虑交换两行代码的顺序：

```

void User::AddFriend(User& newFriend)
{
    // Add the new friend to the vector of friends
    // If this throws, the friend is not added to
    //     the vector, nor the database
    friends_.push_back(&newFriend);
    // Add the new friend to the database
    pDB_->AddFriend(GetName(), newFriend.GetName());
}

```

这确实能在 `vector::push_back` 失败的情况下保护数据一致性。不幸的是，当你查看 `UserDatabase::AddFriend` 的文档，你发现它也会抛出异常。现在你会把好友加到 `vector` 里，但没有加到数据库里。

这时候你会质问做数据库的人们：“为什么你们不返回出错代码，而要抛出异常呢？”他们会说：“我们使用的是一个运行在 TZN 网络上的高可靠的集群数据库服务器，极少出错。因此，我们认为应该用异常来表示出错是最好的，因为异常只出现在异常的情况下，不是吗？”

这个理由讲得通，但是你还是要处理出错情况。你不会希望因为数据库出错而导致整个系统一片混乱。这样你修复数据库时，不必关闭整个服务器程序。

本质上，你必须做两个操作，它们中的任何一个都可能失败。当其中一个失败时，你必须撤销全部操作。让我们来看看怎么做这件事。

➤ 方法 1：粗鲁的方法

一个简单的办法是在 `try-catch` 块中抛出异常。

```
void User::AddFriend(User& newFriend)
{
    friends_.push_back(&newFriend);
    try
    {
        pDB_->AddFriend(GetName(), newFriend.GetName());
    }
    catch (...)
    {
        friends_.pop_back();
        throw;
    }
}
```

如果 `vector::push_back` 失败，那没有问题，因为 `UserDatabase::AddFriend` 不会被执行。如果 `UserDatabase::AddFriend` 失败，你捕获这个异常（不管什么类型），然后你调用 `vector::pop_back` 撤销 `push_back` 的操作，然后再次抛出同样类型的异常。

这样的代码可以工作，代价是增加了代码量，并显得臃肿。本来两行的程序变成了六行。想象一下如果你的代码到处是这样的 `try-catch` 语句，那太可怕了，所以这个方法一点也不吸引人。

而且，这个方法也不好扩展。假如你有三个操作要做，用这个方法写出来的代码将臃肿得多。你有两个差不多坏的写法可选：用嵌套的 `try` 语句，或者用使流程更复杂的附加标志。这个方法会导致很多问题，如代码膨胀、效率下降，以及最重要的可理解和可维护性降低。

➤ 方法 2：原则 (politically) 上正确的方法

如果你把上面的代码给任何一个 C++ 专家看，它很可能会告诉你：“啊，那方法不好。你应该使用惯用法 RAII (Resource Acquisition Is Initialization，资源分配即初始化) [1]，在出错的情况下，依靠析构函数来释放资源。”

OK，让我们沿着这条路走下去。对于每一个你需要撤销的操作，都需要有一个对应的类，这个类的构造函数“做”这个操作，而析构函数“撤销”这个操作，除非你调用了一个“提交”函数，那样的话析构函数就什么也不做。

用一些代码可以把这一切说清楚。对于 `push_back` 操作，我们写一个 `VectorInserter` 类，就像这样：

```

class VectorInserter
{
public:
    VectorInserter(std::vector<User*>& v, User& u)
        : container_(v), user_(u), commit_(false)
    {
        container_.push_back(&u);
    }
    void Commit() throw()
    {
        commit_ = true;
    }
    ~VectorInserter()
    {
        if (!commit_) container_.pop_back();
    }
private:
    std::vector<User*>& container_;
    User& user_;
    bool commit_;
};

```

大概上面代码里最重要的东西就是 `Commit` 后面的 `throw()`。它说明了一个事实，就是 `Commit` 调用永远成功（不会抛出异常）。因为你已经完成你的工作——`Commit` 只是告诉 `VectorInserter`：“一切顺利，不用恢复任何东西。”

你可以像这样使用整个机制：

```

void User::AddFriend(User& newFriend)
{
    VectorInserter ins(friends_, &newFriend);
    pDB_->AddFriend(GetName(), newFriend.GetName());
    // Everything went fine, commit the vector insertion
    ins.Commit();
}

```

`AddFriend` 现在有两个不同的部分：动作阶段——完成操作；提交阶段——不会抛出异常，只是停止所有的撤销工作。

`AddFriend` 的工作方式很简单：如果任何一个操作失败，那么就不能到达提交点，全部操作都会被取消。`VectorInserter` 会把加入的数据 `pop_back` 掉，所以程序会保持在调用 `AddFriend` 以前的状态。

这个方法在所有情况下都工作得很好。比如，当 `vector` 插入失败时，`ins` 的析构函数不会被调用，因为 `ins` 没有被成功构造出来。

这个方法很好，但是在现实世界里，做不到那么简洁。你必须编写一大堆很小的类来支持这个方法。额外的类意味着要写额外的代码，多费脑筋，以及在你的 class browser 里会有更多的条目。而且，你会有更多的地方需要处理异常安全问题。仅仅为了在析构函数里撤销一个操作而不断增加新的类，从生产率来看，这不是最聪明的办法。

哦，还有，`VectorInserter` 里有一个 bug，你注意到了吗？编译器为你隐式生成的拷贝构造函数会导致错误：如果被复制的对象还没有提交过，那么在以后的析构函数里，就可能做过多的 `pop_back` 操作。定义一个类是很困难的，这是我们要避免写很多类的另一个理由。

➤ 方法 3：现实中的方法

在现实世界里，当程序员坐下来编写 `AddFriend` 时，要么他看过上面的几个选择，要么他没有时间去关心这些。一天过去后，你知道真实的结果通常是什么吗？你当然知道：

```
void User::AddFriend(User& newFriend)
{
    friends_.push_back(&newFriend);
    pDB_->AddFriend(GetName(), newFriend.GetName());
}
```

这是个基于“not-too-scientific”论调的解决方法：

译者注：我觉得作者的观点是，现实中的程序员会排斥为了“不太”会出现的错误而采用“太”复杂难用的技术或者编程规范，从而在改善程序的安全性和保证开发进度的权衡中选择进度，并认为这是“科学”的做法。其实这两者并不是“鱼与熊掌，不可得兼”的东西，这是作者在这一系列文章中介绍 GP 可重用类模板技术的目的。

“谁说过内存会用完？这机器有半 G 内存呢！”

“程序会因为没有内存而崩溃？要那样的话，内存交换早就让系统慢得像蜗牛了。”

“做数据库的那帮家伙说 `AddFriend` 几乎不可能失败。他们用的是 XYZ 和 TZN！”

“这太麻烦了，不值得。以后 review 的时候再考虑它吧。”

一个方法如果需要很多约束规则和受到抱怨的工作，那么它就不具有吸引力。在进度的压力下，一个好的但是笨拙的方法会变得不实用。虽然每个人都知道应该按照书本上说的那样去做，但他们始终都喜欢走捷径。唯一的方法是提供一个可重用的解决方案，正确而容易使用。

当你走捷径时，因为你知道你的代码不完美，你会怀着某种不愉快的心情 check in (译者注：我觉得这里是代码管理的 check in，在一个项目的 coding 阶段，一个模块完成的标志就是把代码 check in 到全体代码中去。我觉得原文的意思是，虽然 coding 做完了，也可以通过测试，符合 check in 的标准，迫于进度，你可以也必须 check in 代码，但是因为你知道还有隐患，所以有不完美的感觉) 你的代码。但是这种心情会逐渐消失，因为所有的测试都可以通过。但是随着时间推移，那些“理论上”会引起问题的地方，还是会开始从现实中冒出来。

你知道你遇到了问题，而且是个大问题：你放弃了对应用程序正确性的控制。现在，当服务器程序崩溃时，你没有很多线索去找错：硬件故障？真正的 bug？还是因异常而引起的混乱状态？你遇到的不是无心的 bug，而是你自己故意引入的。

即使在一段时间内程序可以工作，但是事情总是会变化的。用户数目会增加，导致内存使用到达极限。你的网络管理员可能为了保证性能而禁止内存分页系统。你的数据库可能不是那么可靠。你对这一切都没有准备。

➤ 方法 4：Petru 的方法

用 ScopeGuard——我们稍后详细介绍——你很容易就可以写出简洁、正确而高效的代码：

```
void User::AddFriend(User& newFriend)
{
    friends_.push_back(&newFriend);
    ScopeGuard guard = MakeObjGuard(
        friends_, &UserCont::pop_back);
    pDB_->AddFriend(GetName(), newFriend.GetName());
    guard.Dismiss();
}
```

上面代码里，guard 对象唯一的任务就是在它离开作用域时，调用 friends_.pop_back，除非你调用了 Dismiss。如果你调用了，那么 guard 就什么也不做。

ScopeGuard 在它的析构函数里实现自动调用某个全局函数或者成员函数。在有异常的情况下，你会想要实现自动撤销原子操作的功能，这时候 ScopeGuard 会很有用。

你可以这样使用 ScopeGuard：如果你希望几个操作按照“要么全做，要么全不做”的方式工作，你可以在紧接着每个操作后面放一个 ScopeGuard，这个 ScopeGuard 可以取消前面的操作：

```
friends_.push_back(&newFriend);
ScopeGuard guard = MakeObjGuard(friends_, &UserCont::pop_back);
```

ScopeGuard 也可用于普通函数：

```

void* buffer = std::malloc(1024);
ScopeGuard freeIt = MakeGuard(std::free, buffer);
FILE* topSecret = std::fopen("cia.txt");
ScopeGuard closeIt = MakeGuard(std::fclose, topSecret);

```

当整个原子操作成功时，你 Dismiss 所有 guard 对象。否则每个 ScopeGuard 对象会忠实的调用你构造它时所传的那个函数。

有了 ScopeGuard，你可以简单的安置各种撤销操作，而不再需要写特别的类来做诸如删除 vector 的最后一个元素、释放内存、关闭文件这样的事情。这使 ScopeGuard 成为编写异常安全代码的一个极其有用、并且可重用的解决方案，它使一切变得很简单。

➤ 实现 ScopeGuard

ScopeGuard 是对 C++ 惯用法 RAII (资源分配即初始化) 典型实现的一个推广。它们的区别在于 ScopeGuard 只关注资源清理的那部分——资源分配由你自己做，而 ScopeGuard 处理资源的释放 (事实上，可以论证清理工作是这个谚语里最重要的部分)。

释放资源有很多种形式，比如调用一个函数、调用一个 functor、或者调用一个对象的成员函数，而每种方式都可能有零个、一个或者更多的参数。

自然，我们通过一个类层次关系来对这些变体建模。层次中各个类的对象的析构函数完成实际工作。层次中的根为 ScopeGuardImplBase 类，如下：

```

class ScopeGuardImplBase
{
public:
    void Dismiss() const throw()
    {   dismissed_ = true;   }

protected:
    ScopeGuardImplBase() : dismissed_(false)
    {}
    ScopeGuardImplBase(const ScopeGuardImplBase& other)
        : dismissed_(other.dismissed_)
    {   other.Dismiss();   }
    ~ScopeGuardImplBase() {} // nonvirtual (see below why)
    mutable bool dismissed_;

private:
    // Disable assignment
    ScopeGuardImplBase& operator=
    const ScopeGuardImplBase& );

```

```
};
```

ScopeGuardImplBase 集中了对 `dismissed_` 标志的管理，这个标志控制派生类是否要执行清理工作。如果 `dismissed_` 为真，则派生类在他们的析构函数里什么也不做。

现在我们来看看 ScopeGuardImplBase 析构函数定义里缺少的 `virtual`。如果析构函数不是 `virtual` 的，你怎么可以期望析构函数有正确的多态行为呢？好，把你的好奇心再保持一会儿，我们手里还有张王牌，我们可以通过它得到多态的析构行为，而不必付出虚函数的代价。

现在我们先来看看怎么实现这样一个对象，它在析构函数里调用一个带一个参数的函数或者 functor。然而当你调用了 `Dismiss`，那么这个函数或者 functor 就不会被调用。

```
template <typename Fun, typename Parm>
class ScopeGuardImpl1 : public ScopeGuardImplBase
{
public:
    ScopeGuardImpl1(const Fun& fun, const Parm& parm)
        : fun_(fun), parm_(parm)
    {}
    ~ScopeGuardImpl1()
    {
        if (!dismissed_) fun_(parm_);
    }
private:
    Fun fun_;
    const Parm parm_;
};
```

为了方便使用 `ScopeGuardImpl1`，我们写一个辅助函数。

```
template <typename Fun, typename Parm>
ScopeGuardImpl1<Fun, Parm>
MakeGuard(const Fun& fun, const Parm& parm)
{
    return ScopeGuardImpl1<Fun, Parm>(fun, parm);
}
```

`MakeGuard` 依靠编译器推导出模板函数中的模板参数，这样你就不用自己指定 `ScopeGuardImpl1` 的模板参数了——事实上你不需要显式创建 `ScopeGuardImpl1` 对象。这个技巧也被一些标准库中的函数所使用，如 `make_pair` 和 `bind1st`。

你还对不使用虚构造函数而得到多态性析构行为的方法感到好奇吗？下面是 `ScopeGuard` 的定义，会让你大吃一惊的是，它仅仅是一个 `typedef`：

```
typedef const ScopeGuardImplBase& ScopeGuard;
```

好了现在让我们来揭开全部神秘机制。根据 C++ 标准，如果 `const` 的引用被初始化为对一个临时变量的引用，那么它会使这个临时变量的生命期变得和它自己一样。让我们举个例子来解释这件事。如果你写：

```
FILE* topSecret = std::fopen("cia.txt");
ScopeGuard closeIt = MakeGuard(std::fclose, topSecret);
```

那么 `MakeGuard` 创建了一个临时变量，它的类型为（看以前做一下深呼吸）：

```
ScopeGuardImpl1<int (&)(FILE*), FILE*>
```

这是因为 `std::fclose` 是接受 `FILE*` 类型参数返回 `int` 的函数。具有上面那个类型的临时变量被指派给了 `const` 引用 `closeIt`。根据上面提到的 C++ 语言规则，这个临时变量会和它的引用 `closeIt` 有同样长的生存期——当这个临时变量被析构时，会调用正确的析构函数。接着，析构函数关闭文件。

`ScopeGuardImpl1` 支持有带参数的函数（或 `functor`），很容易就可以写出不带参数、带两个参数或带更多参数的类（`ScopeGuardImpl0`、`ScopeGuardImpl2`……）。当你有了这些类，你就可以重载 `MakeGuard`，从而得到一个优美、统一的语法：

```
template <typename Fun>
ScopeGuardImpl0<Fun>
MakeGuard(const Fun& fun)
{
    return ScopeGuardImpl0<Fun>(fun);
}
...
```

到现在为止，我们已经有了一个强大的工具来表达调用一组函数的原子操作。`MakeGuard` 是一个优秀的工具，特别是它同样可以用于 C 语言的 API，而不需要写很多包装类。

更好的是，它不损失效率，因为它不涉及到虚函数调用。

➤ 针对对象和成员函数的 `ScopeGuard`

到现在为止，一切都很好，但是怎么调用对象的成员函数呢？其实这一点也不难。让我们来实现 `ObjScopeGuardImpl0`，一个可以调用对象的无参数成员函数的类模板。

```
template <class Obj, typename MemFun>
```

```

class ObjScopeGuardImpl0 : public ScopeGuardImplBase
{
public:
    ObjScopeGuardImpl0(Obj& obj, MemFun memFun)
        : obj_(obj), memFun_(memFun)
    {}
    ~ObjScopeGuardImpl0()
    {
        if (!dismissed_) (obj_.*fun_)();
    }
private:
    Obj& obj_;
    MemFun memFun_;
} ;

```

ObjScopeGuardImpl0 有一点特别，因为它用了不太为人所知的语法：指向成员函数的指针和 operator.*()。为了理解它是如何工作的，让我们来看看 MakeObjGuard 的实现（我们在本节开始已经利用过 MakeObjGuard 了）。

```

template <class Obj, typename MemFun>
ObjScopeGuardImpl0<Obj, MemFun, Parm>
MakeObjGuard(Obj& obj, Fun fun)
{
    return ObjScopeGuardImpl0<Obj, MemFun>(obj, fun);
}

```

现在，如果你调用：

```
ScopeGuard guard = MakeObjGuard(friends_, &UserCont::pop_back);
```

会创建一个如下类型的对象：

```
ObjScopeGuardImpl0<UserCont, void (UserCont::*())()>
```

幸好，MakeObjGuard 让你免于写那些跟字符型图标一样单调的类型。工作机制还是一样——当 guard 离开作用域，临时对象的析构函数会被调用。析构函数通过指向成员的指针调用成员函数。这里我们用到了.*操作符。

➤ 错误处理

如果你读过 Herb Sutter 关于异常的著作[2]，你就会知道一条基本原则：析构函数不应该抛出异常。一个会抛出异常的析构函数会让你无法写出正确的代码，并且会再没有任何警告的情况下终止

你的应用程序。在 C++ 里，当一个异常被抛出，在堆栈展开 (unwinding) 的时候某个析构函数又抛出另一个异常，应用程序会被马上终止。

ScopeGuardImplX 和 ObjScopeGuardImplX 分别调用了一个未知的函数或成员函数，那个函数可能会抛出异常。这会终止程序，因为我们设计 guard 的析构函数的目的就是：当有异常被抛出，在展开 (unwinding) 堆栈时，调用这个未知函数！理论上，你不应该把可能抛出异常的函数传给 MakeGuard 或者 MakeObjGuard。在实用中（你可以从供下载的代码中看到），析构函数对异常采取了防御措施。

```
template <class Obj, typename MemFun>
class ObjScopeGuardImpl0 : public ScopeGuardImplBase
{
    ...
public:
    ~ScopeGuardImpl1()
    {
        if (!dismissed_)
            try { (obj_.*fun_)(); }
            catch(...) {}
    }
}
```

是的，catch(...) 块什么事也不做。这可不是随手写的，这在异常处理的领域中是很基本的：如果你的“撤销/恢复”操作也失败了，那么你几乎没有事情可以做了。你尝试恢复，但是不管恢复操作是否成功，你都应该继续下去。

以我们的即时消息为例，一个可能动作顺序是：你向数据库里加入了一个好友数据，但当把它插入 friends_ vector 时失败了，当然你会尝试把它从数据库里再删掉。虽然可能性很小，但是从数据库里删除数据时，不知道为什么也失败了，这种情况就很讨厌了。

一般来说，你应该在那些保证可以成功撤销的操作上使用 guard。

➤ 支持传引用的参数

在 Petru 和我很高兴地使用 ScopeGuard 一段时间以后，我们遇到一个问题。考虑下面的代码：

```
void Decrement(int& x) { --x; }
void UseResource(int refCount)
{
    ++refCount;
    ScopeGuard guard = MakeGuard(Decrement, refCount);
    ...
}
```

```
}
```

上面代码中的 guard 对象确保 refCount 的值在 UseResource 函数退出时保持不变。(这个惯用法在一些共享资源的情况下很有用。)

尽管有用，但上面的代码不能工作。问题在于，ScopeGuard 保存了 refCount 的一个拷贝(看一下 ScopeGuardImpl1 的定义，在成员变量 parm_ 里)而不是对它的引用。然而在这个例子里，我们需要的是保存 refCount 的一个引用，这样才能让 Decrement 对它进行操作。

一个解决办法是再实现一些类，例如 ScopeGuardImplRef，以及 MakeGuardRef。这会有许多重复劳动，并且在实现处理多参数的类时，这个办法就很难应付了。

我们采取的办法是：使用一个辅助类，它把引用转变为一个值。

```
template <class T>
class RefHolder
{
    T& ref_;
public:
    RefHolder(T& ref) : ref_(ref) {}
    operator T& () const
    {
        return ref_;
    }
};

template <class T>
inline RefHolder<T> ByRef(T& t)
{
    return RefHolder<T>(t);
}
```

RefHolder 以及和它配套的辅助函数 ByRef 可以无缝地使引用适合于值的语义，并且使 ScopeGuardImpl1 不需要任何改变就可以使用引用。你要做的只是把引用形式的参数用 ByRef 包装一下，就象这样：

```
void Decrement(int& x) { --x; }
void UseResource(int refCount)
{
    ++refCount;
    ScopeGuard guard = MakeGuard(Decrement, ByRef(refCount));
    ...
}
```

我们发现这个方法很有说明性，它提醒你正在用引用方式传递参数。

这个支持引用的办法最好的一点是在 `ScopeGuardImpl1` 中的 `const` 修饰。这里是相关的代码摘要：

```
template <typename Fun, typename Parm>
class ScopeGuardImpl1 : public ScopeGuardImplBase
{
    ...
private:
    Fun fun_;
    const Parm parm_;
};
```

这个小小的 `const` 非常重要。它防止使用非 `const` 引用的代码通过编译和不正确地运行。换句话说，如果你忘记使用 `ByRef`，编译器不会让这样的错误代码通过。

➤ 再等一下，还有一点

到现在为止，你有了一个好工具可以帮助你写出正确的代码，而不用发愁。然而有时候你会想要 `ScopeGuard` 在退出一个代码块时始终执行。这种情况下，定义一个 `ScopeGuard` 类型的哑变量很麻烦——你只需要一个临时变量，而不需要给它命名。

宏 `ON_BLOCK_EXIT` 可以做到这样，你可以这样写出表达力更好的代码：

```
{  
    FILE* topSecret = fopen("cia.txt");  
    ON_BLOCK_EXIT(std::fclose, topSecret);  
    ... use topSecret ...  
} // topSecret automagically closed
```

`ON_BLOCK_EXIT` 表示：“我希望在当前代码块退出时做这个动作。”类似的，`ON_BLOCK_EXIT_OBJ` 对于成员函数调用实现相同的功能。

这些宏使用了不太正统的（虽然合法的）花招，这里就不公开了。如果你好奇，你可以到代码里去查看这些宏（因为编译器的 bug，用 Microsoft VC++的朋友必须关闭“Program Database for Edit and Continue”设定，否则 `ON_BLOCK_EXIT` 会有问题）。

➤ 现实中的 `ScopeGuard`

我们喜欢 `ScopeGuard` 是因为它易于使用和概念简单。这篇文章详细讲了整个实现，但是要解

释 ScopeGuard 的用法只要几分钟。ScopeGuard 在我们的同事中间象野火一样迅速传播开来，每个人都认为它是一个很有价值的工具，很多情况下有助于防止因为异常而过早返回。有了 ScopeGuard，你可以轻松地编写异常安全的代码，而且理解和维护也同样简单。

每个工具都有推荐的使用方法，ScopeGuard 也不例外。你应该象 ScopeGuard 期望的那样使用它——作为函数中的自动变量。你不应该把 ScopeGuard 对象用作成员变量，或者在堆上分配它们。为此，供下载的代码中包含了一个 Janitor 类，它和 ScopeGuard 做的事情一样，但是采取了更通用的做法——代价是损失了一些效率。因为编译器的 bug，Borland C++ 5.5 的用户需要使用 Janitor 而不是 ScopeGuard。

➤ 结论

我们讨论了一些在编写异常安全代码中出现的一些情况。在比较了几个在这些情况下获得异常安全性的方法以后，我们介绍了一个方法，适用于有防错性（并且不会再 throw）撤销操作可用的情况。ScopeGuard 使用了若干泛型编程的技术，让你指定在 ScopeGuard 退出代码块时调用的函数和成员函数。作为可选项，你也可以解除 ScopeGuard 对象的动作。

当你需要实行资源的自动释放，并且可以依靠防错的撤销操作，ScopeGuard 在这种情况下对你很有帮助。当你把几个可能会失败，但是也可以撤销的操作组成一个原子操作，这个惯用法就变得很重要了。当然这个方法也有不适用的情况。

➤ 致谢

Herb Sutter 对本文进行了特别的技术审查。作者也感谢 Mihai Antonescu 和 Dan Pravat 对本文所做的修正以及所提的建议。

➤ 参考资料

- [1] Bjarne Stroustrup. *The C++ Programming Language*, 3rd Edition (Addison-Wesley, 1997), page 366.
- [2] Herb Sutter. *Exceptional C++: 47 Engineering Puzzles, Programming Problems, and Solutions* (Addison-Wesley, 2000).

C++批评系列

继承的本质

Ian Joyner

cber 译

以下文章翻译自 Ian Joyner 所著的 *C++?? A Critique of C++ and Programming and Language Trends of the 1990s* 3/E 【Ian Joyner 1996】。

原著版权属于 Ian Joyner，征得 Ian Joyner 本人的同意，我得以将该文翻译成中文。因此，本文的中文版权应该属于我;-)

该文章的英文及中文版本都用于非商业用途，你可以随意地复制和转贴它。不过最好请在转贴时加上前面的这段声明。

如果有人或机构想要出版该文，请最好联系原著版权所有人及我。

另外，该篇文章已经包含在 Ian Joyner 所写的 *Objects Unencapsulated* 一书中（目前已经有了日文的翻译版本），该书的介绍可参见于：

http://www.prenhall.com/allbooks/ptr_0130142697.html

<http://efsa.sourceforge.net/cgi-bin/view/Main/ObjectsUnencapsulated>

<http://www.accu.org/bookreviews/public/reviews/o/o002284.htm>

Ian Joyner 的联系方式：i.joyner@acm.org

我的联系方式：cber@email.com.cn

译者前言：

要想彻底的掌握一种语言，不但需要知道它的长处有哪些，而且需要知道它的不足之处又有哪些。这样我们才能用好这门语言，避免踏入语言中的一些陷阱，更好地利用这门语言来为我们的工作所服务。

Ian Joyner 的这篇文章以及他所著的 *Objects Unencapsulated* 一书中，向我们充分地展示了 C++ 的一些不足之处，我们应该充分借鉴于他已经完成的伟大工作，更好地了解 C++，从而写出更加安全的 C++ 代码来。

继承关系是一种耦合度很高的关系，它与组合及一般化 (genericity) 一样，提供了 OO 中的一种基本方法，用以将不同的软件组件组合起来。一个类的实例同时也是那个类的所有祖先的实例。为了保证面向对象设计的有效性，我们应该保存下这种关系的一致性。在子类中的每一次重新定义都应该与其祖先类中的最初定义进行一致性检查。子类中应该保存下其祖先类的需求。如果存在着不能被保存的需求，就说明了系统的设计有错误，或者是在系统中此处使用继承是不恰当的。由

于继承是面向对象设计的基础，所以才会要求有一致性检测。C++中对于非虚拟函数重载的实现，意味着编译器将不会对其进行一致性检测。C++并没有提供面向对象设计的这方面的保证。

继承被分成“语法”继承和“语义”继承两部分。Saake 等人将其描述如下：“语法继承表示为结构或方法定义的继承，并且因此与代码的重复使用(以及重写被继承方法的代码)联系起来。语义继承表示为对对象语义(即对象自己)的继承。这种继承形式可以从语义的数据模型中得知，在此它被用于代表在一个应用程序的若干个角色中出现的一个对象。”[SJE 91]。Saake 等人集中研究了继承的语义形式。通过是行为还是语义的继承方式的判断，表示了对象在系统中所扮的角色。

然而，Wegner 相信代码继承更具有实际的价值。他将语法与语义继承之间的区别表示为代码和行为上的区别[Weg 91] (p43)。他认为这样的划分不会引起一方与另一方的兼容，并且还经常与另一方不一致。Wegner 同样也提出这样的问题：“应该怎样抑制对继承属性的修改？”代码继承为模块化(modularisation)提供一个基础。行为继承则依赖于“is-a”关系。这两种继承方式在合适处都十分有用。它们都要求进行一致性的检测，这与实际上的有意义的继承密不可分。

看起来在语义保持关系中那些限制最多的形式中，继承似乎是其中最强的形式；子类应该保存祖先类中的所有假设。

Meyer [Meyer 96a and 96b]也对继承技术进行了分类。在他的分类法中，他指出了继承的 12 种用法。这些分析也给我们怎么使用继承提供了一个很好的判断标准，如：什么时候应该使用继承，什么时候不应该它。

软件组件就象七巧板一样。当我们组装七巧板时，每一块板的形状必须要合适，但更重要地是，最终拼出的图像必须要有意义，能够被说得通。而将软件组件组合起来就更困难了。七巧板只是需要将原本是完整的一幅图像重新组合起来。而对软件组件的组合会得到什么样的结果，是我们不可能预见到的。更糟的是，七巧板的每一块通常是由不同的程序员产生的，这样当整个的系统被组合起来时，对于它们的吻合程度的要求就更高了。

C++中的继承像是一块七巧板，所有的板块都能够组合在一起，但是编译器却没有办法检测最终的结果是否有意义。换句话说，C++仅为类和继承提供了语法，而非语义。可重用的 C++ 函数库的缓慢出现，暗示了 C++ 可能会尽可能地不支持可重用性。相反的是，Java、Eiffel 和 Object Pascal 都与函数库包装在一起出现。Object Pascal 与 MacApp 应用软件框架联系非常紧密。Java 也从与 Java API 的耦合中解脱出来，取而代之的是一个包容广泛的函数库。Eiffel 也同样是与一个极其全面的函数库集成在一起，该函数库甚至比 Java 的还要大。事实上函数库的概念已经成为一个优先于 Eiffel 语言本身的工程，用以对所有在计算机科学中通用的结构进行重新分类，得到一个常用的分类法。[Meyer 94]。

芝麻开门之 OpenGL

设置 OpenGL (一)

作者 : Jeff Molofee

翻译 : CKER 虫虫

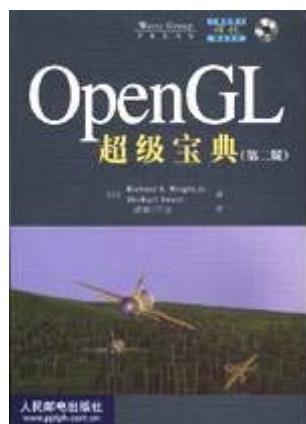
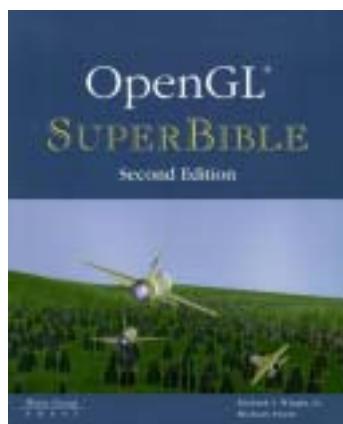
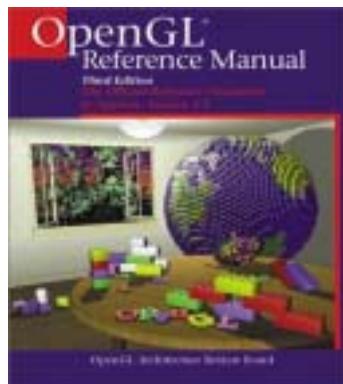
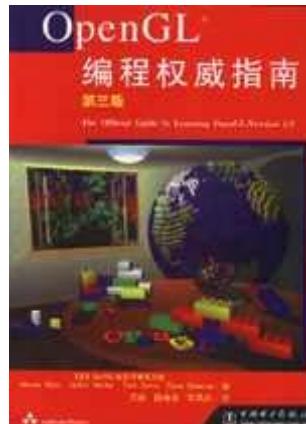
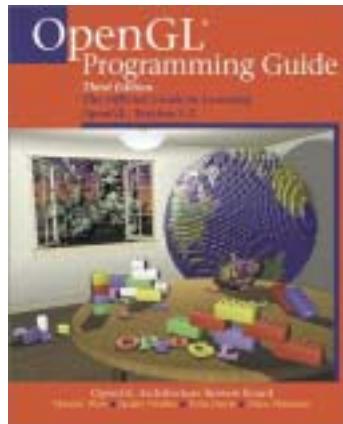
编者按 : OpenGL , 最早由 SGI 开发的跨平台 3D 图形软件接口 , 已成为工业标准 , 《侏罗纪公园》中的恐龙就是 OpenGL 所立下的汗马功劳。另外 ,[许多著名的 C++ 跨平台 GUI 类库 , 就是以 OpenGL 为基础而建构的](#) , 可见 OpenGL 大有用武之地。 Jeff 的 OpenGL 教程 , 内容翔实 , 并且不断更新 , 是我所见过最好的 OpenGL 入门教程。感谢 Jeff 的授权 , 让我们能开始这激动人心的 OpenGL 之旅。

译者的话 Jeff 的例程所提供的源代码几乎涵盖了各个平台各种语言的编译器版本 , 包括 Visual C++ 、 Borland C++ 、 Visual Basic 、 MacOS X/ GLUT 、 Linux/ GLX 、 Code Warrior 、 Delphi 、 C++ Builder 、 MASM 、 ASM 、 MingW32 & Allegro 以及 Python 等等的不同平台下的多种编译器。这在国内市场上的百元大书中似乎也未曾见到 , 我们实在应该向他学习。关于 OpenGL 的专用术语的翻译难免有错误和不妥之处 , 请多加指正。另外 , 要想流畅的运行例程 , 您的爱机应该够劲 , 内存应该够大 , 还要支持 3D 硬件加速的显卡 , TNT 总该有吧:-)

这篇译文包括了原文的前言、第一课以及各个平台下 OpenGL 的设置问题。由于本系列教程主要在 Windows 平台下讲述 , 所以使用其他操作系统的朋友一定要注意本文中的讲述。

前言摘要 : 这系列教程可能会有疏漏 , 解释也可能不够清楚 , 也许不算是学习 OpenGL 最好的材料 , 我仅仅是希望让初学 OpenGL 的人能轻松点儿。如果您真的下定决心 , 很想好好得学学 OpenGL , 那您应该把银子撒向经典书籍 : OpenGL 红皮书 (ISBN 0-201-46138-2) 和蓝皮书 (ISBN 0-201-46140-4)(我手头有这两本书的第二版), 不过对初学者而言 , 这两本书的确难了点 , 但到目前为止 , 这可是 OpenGL 领域最好的书。我还推荐一本颇具争议的 *OpenGL Super Bible* 。

读读我的代码 , 看看书 , 如果需要也可以问我问题。一旦您的水平已经超越了教程 ,



请去一些专业站点充充电，如 <http://www.opengl.org>，及我主页上的 OpenGL 链接，链接的站点都有丰富的宝藏，往往由某位大虾制作，他们写的代码可就比我的好多了。(译注：我们所附加的图就是作者提到的三本书的中英文版，分别是红皮书、蓝皮书和超级宝典。OpenGL 虽然是跨平台的工业标准，但是各个平台上的基本配置方法却不同。由于本教程是在 Windows 平台下讲述，作者还特别说明了在其他平台下的问题，请参考。)

MacOS

首先，最重要的是要一个编译器，现在 Macintosh 上最好最流行的是 [Metrowerks Codewarrior](#)。如果您是一位学生，可以选择教育版，它除了更便宜外跟专业版没有任何区别。其次需要安装 Apple 的 [OpenGL SDK](#)。然后就行了！

用 GLUT 开始，下面开头是必要的头文件：

```
#include <GL/gl.h>
#include <GL/glu.h>
#include <GL/glut.h>
#include "tk.h"
```

第一个是 OpenGL 标准用法，另三个是我们程序中要用的附加调用。然后定义一些常量，用于定义窗口的高和宽，然后是函数原型：

```
#define kWindowWidth 400
#define kWindowHeight 300

GLvoid InitGL(GLvoid);
GLvoid DrawGLScene(GLvoid);
GLvoid ReSizeGLScene(int Width, int Height);
```

以及函数 main()：

```
int main(int argc, char** argv)
{
    glutInit(&argc, argv);
    glutInitDisplayMode (GLUT_DOUBLE | GLUT_RGB | GLUT_DEPTH);
    glutInitWindowSize (kWindowWidth, kWindowHeight);
    glutInitWindowPosition (100, 100);
    glutCreateWindow (argv[0]);
    InitGL();
    glutDisplayFunc(DrawGLScene);
    glutReshapeFunc(ReSizeGLScene);
    glutMainLoop();
    return 0;
```

```
}
```

glutInit(), glutInitDisplayMode(), glutInitWindowSize(), glutInitWindowPosition() 以及 glutCreateWindow() 均用于设置我们的 OpenGL 程序。InitGL() 在 Mac 和 Windows 程序中作用相同。glutDisplayFunc(DrawGLScene) 告诉 GLUT，我们画场景时需要使用 DrawGLScene 函数。glutReshapeFunc(ReSizeGLScene) 则告诉 GLUT，当窗口大小改变时使用 ReSizeGLScene 函数。

以后，我们还会使用 glutKeyboardFunc() 告诉 GLUT，当某个键被按下时我们要使用的回调函数，以及 glutIdleFunc() 来告诉 GLUT，在没有收到消息时反复调用的全局空闲回调函数（经常用来处理连续的动画场景，比如在 OpenGL 空间中旋转物体时）。

最后，由 glutMainLoop() 启动程序。从此，除非程序退出，否则不再回到 main() 函数。

这样就差不多了。

- Tony Parker , asp@usc.edu

Solaris

下面简单讲一下如何在 Sun 工作站 Solaris 7 上安装 OpenGL 和 GLUT 库。

首先确认安装了 Solaris DEVELOPER，这样才能保证所有必需的头文件已经存在，最简单的办法是安装 Solaris 时选为开发版（development version），用普通的 Solaris 安装 CD ROM 就行。然后注意您的/usr/include 和/usr/openwin/include 目录下有必要的文件即可。

Sun 没随 Solaris 提供 C/C++ 编译器，不过运气还好，我们不用花钱:-)

<http://www.sunfreeware.com>

在这里可以找到 gcc (GNU Compiler Collection) for Solaris，安装很简单。

```
> pkgadd -d gcc-xxxversion
```

这将把 gcc 装在/usr/local 下。当然也可以用 admintool 做：

```
> admintool
```

```
Browse->Software  
Edit->Add
```

然后选择安装包所在的目录即可。如果必要，我建议同时下载并安装 libstdc++ 库。现在的 Solaris 中都应该有。检查一下：

```
> cd /usr/openwin/lib  
> ls libGL*
```

应该显示：

```
libGL.so@ libGLU.so@ libGLw.so@  
libGL.so.1* libGLU.so.1* libGLw.so.1*
```

这表示运行库已经安装了。不过头文件也在吗？

```
> cd /usr/openwin/include/GL  
> ls
```

这应该输出：

```
gl.h glu.h glxmd.h glxtokens.h  
glmacros.h glx.h glxproto.h
```

看看下面的链接，这是个 FAQ。

<http://www.sun.com/software/graphics/OpenGL/Developer/FAQ-1.1.2.html>

有问题就看看吧。您没有 OpenGL？版本太低？去下个新的吧：

<http://www.sun.com/solaris/opengl>

注意下载 OS 相应版本的补丁，这得用 root 权限做。现在有 OpenGL 了，但是 GLUT 呢？在这里可以找到：<http://www.sun.com/software/graphics/OpenGL/Demos/index.html>

我把 GLUT 装在/usr/local，这是装这种东西的好地方。装了以后，我运行 progs/下的示例时，却说找不到 libglut.a。为了能让操作系统找到运行库，还得把 GLUT 的路径加到变量 LD_LIBRARY_PATH。

如果在用/bin/sh，这样做：

```
>  
LD_LIBRARY_PATH=/lib:/usr/lib:/usr/openwin/lib:/usr/dt/lib:/usr/local/lib:  
ib:/usr/local/sparc_solaris/glut-3.7/lib/glut  
  
> export LD_LIBRARY_PATH
```

如果在用 csh，这样做：

```
> setenv LD_LIBRARY_PATH  
/lib:/usr/lib:/usr/openwin/lib:/usr/dt/lib:/usr/local/lib:/usr/local/sp  
arc_solaris/glut-3.7/lib/glut
```

检验一切是否正确：

```
> echo $LD_LIBRARY_PATH  
/lib:/usr/lib:/usr/openwin/lib:/usr/dt/lib:/usr/local/lib:/usr/local/sp  
arc_solaris/glut-3.7/lib/glut
```

搞定。

- [Lakmal Gunasekara](#) 1999

MacOS X public beta

首先需要一个编译器。两种可用的编译器是 Apple 的 Project Builder 和 Metrowerks CodeWarrior。从 2000 年 10 月中开始，Project Builder 成为自由软件，所以我们主要谈谈如何在 Project Builder 中创建 GLUT 工程。

从“File->New Project”开始，选择 Cocoa Application，选择工程名，IDE 就出现了。然后从“Project->Add Framework...”加入 GLUT.framework

程序开头是头文件，我们仅仅需要一个，而不是三个：

```
#include <GLUT/glut.h>
```

然后跟 MacOS 中的做法一样，略去。好好享受 OpenGL 吧。

- R.Goff (unreality@mac.com)

欢迎来到我的 OpenGL 教程。我是个对 OpenGL 充满激情的普通男孩！我第一次听说 OpenGL 是 3Dfx 发布 Voodoo1 卡的 OpenGL 硬件加速驱动的时候。我立刻意识到 OpenGL 是那种必须学习的东西。不幸的是当时很难从书本或网络上找到关于 OpenGL 的讯息。我花了 N 个小时来调试自己书写的代码，甚至在 IRC 和 EMail 上花更多的时间来恳求别人帮忙。但我发现那些懂得 OpenGL 高手们保留了他们的精华，对共享知识也不感兴趣。实在让人灰心！

我的目的是帮助那些对 OpenGL 有兴趣却又需要帮助的人。在每个教程中，我都会尽可能详细的来解释每一行代码的作用。我会努力让我的代码更简单（您无需学习 MFC）！就算您是个 VC、OpenGL 的超级新手也可以读通代码。

教程的这一节将会教您如何设置一个 OpenGL 窗口。它可以只是一个窗口或是全屏幕的、可以任意大小、任意色彩深度。此处的代码很稳定且很强大，可以在您所有的 OpenGL 项目中使用。我所有的教程都将基于此节的代码！代码也很容易阅读和修改，应该没有内存泄漏。感谢 Fredric Echols！

现在就让我们直接从代码开始吧。假定使用 VC，先创建一个新工程（低版本的 VC 需要将 bool 改成 BOOL，true 改成 TRUE，false 改成 FALSE，VC4 和 VC5 没有问题），创建一个新的 Win32

程序（不是 Console 控制台程序）后，还需链接 OpenGL 库文件。操作如下：Project > Settings，然后单击 LINK 标签。在“Object/Library Modules”选项中的开始处(kernel32.lib 前)增加 OpenGL32.lib GLu32.lib 和 GLaux.lib 后单击 OK。现在可以开始写 OpenGL 程序了。

代码的前 4 行包括了我们使用的每个库文件的头文件。如下所示：

```
#include <windows.h>                                // Windows 的头文件
#include <gl\gl.h>                                    // OpenGL32 库的头文件
#include <gl\glu.h>                                    // GLu32 库的头文件
#include <gl\glaux.h>                                  // GLaux 库的头文件
```

接下来需要设置在程序中计划使用的所有变量。本节中的例程将创建一个空的 OpenGL 窗口，因此我们暂时还无需设置大堆的变量。余下需要设置的变量不多，但十分重要。您将会在以后所写的每一个 OpenGL 程序中用到它们。

第一行设置的变量是 Rendering Context（着色描述表），每一个 OpenGL 都被连接到一个着色描述表上。着色描述表将所有的 OpenGL 调用命令连接到 Device Context（设备描述表）上。我将 OpenGL 的着色描述表定义为 hRC。要让您的程序能够绘制窗口的话，还需要创建一个设备描述表，也就是第二行的内容。Windows 的设备描述表被定义为 hDC。DC 将窗口连接到 GDI，而 RC 将 OpenGL 连接到 DC。第三行的变量 hWnd 将保存由 Windows 给我们的窗口指派的句柄。最后，第四行为我们的程序创建了一个 Instance（实例）。

```
HGLRC hRC=NULL;                                     // 永久着色描述表
HDC hDC=NULL;                                       // 私有 GDI 设备描述表
HWND hWnd=NULL;                                      // 保存我们的窗口句柄
HINSTANCE hInstance;                                 // 保存程序的实例
```

下面的第一行设置一个用来监控键盘动作的数组。有许多方法可以监控键盘的动作，但这里的方法很可靠，并且可以处理多个键同时按下的情况。

变量 active 用来告知程序窗口是否处于最小化的状态。如果窗口已经最小化，我们可以做从暂停代码执行到退出程序的任何事情。我喜欢暂停程序，这样可以使得程序不用在后台保持运行。

fullscreen 变量的作用相当明显：若程序在全屏状态下运行，fullscreen 的值为 TRUE，否则为 FALSE。这个全局变量的设置十分重要，它让每个过程都知道程序是否运行在全屏状态下。

```
bool keys[256];                                     // 用于键盘例程的数组
bool active=TRUE;                                    // 窗口的活动标志，缺省为 TRUE
bool fullscreen=TRUE;                                // 全屏标志缺省设定成全屏模式
```

现在我们需要定义 WndProc()。

```
LRESULT CALLBACK WndProc(HWND, UINT, WPARAM, LPARAM);
```

下面的代码的作用是重新设置 OpenGL 场景的大小，而不管窗口的大小是否已经改变（假定没有使用全屏模式）。甚至您无法改变窗口的大小时（例如您在全屏模式下），它仍至少运行一次在程序开始时设置我们的透视图。OpenGL 场景的尺寸将被设置成它显示时所在窗口的大小。

```
GLvoid ReSizeGLScene(GLsizei width, GLsizei height)
{
    if(height==0)                                // 重置并初始化 GL 窗口大小
        height=1;                                // 防止被零除
    glViewport(0, 0, width, height);           // 将 Height 设为 1
                                                // 重置当前的视口 (Viewport)
```

下面几行为透视图设置屏幕。越远的东西看起来越小，这样就创建了一个现实外观的场景。此处透视按照基于窗口宽度和高度的 45 度视角来计算。0.1f, 100.0f 是我们在场景中所能绘制深度的起点和终点。

glMatrixMode(GL_PROJECTION) 指明接下来的两行代码将影响 projection matrix (投影矩阵)。投影矩阵负责为我们的场景增加透视。glLoadIdentity() 近似于重置。它将所选的矩阵状态恢复成其原始状态。调用 glLoadIdentity() 之后我们为场景设置透视图。

glMatrixMode(GL_MODELVIEW) 指明任何新的变换将会影响 modelview matrix (模型观察矩阵)。模型观察矩阵中存放了我们的物体讯息。最后我们重置模型观察矩阵。如果不能理解这些术语，请别着急。在以后的教程里，我会向大家解释。只要知道如果您想获得一个精彩的透视场景的话，必须这么做。

```
glMatrixMode(GL_PROJECTION);          // 选择投影矩阵
glLoadIdentity();                     // 重置投影矩阵

// 计算窗口的外观比例
gluPerspective(45.0f,(GLfloat)width/(GLfloat)height,0.1f,100.0f);
glMatrixMode(GL_MODELVIEW);           // 选择模型观察矩阵
glLoadIdentity();                     // 重置模型观察矩阵
}
```

接下的代码段中，我们将对 OpenGL 进行全面的设置：设置清除屏幕所用的颜色，打开深度缓存，启用 smooth shading (阴影平滑)，等等。这个例程直到 OpenGL 窗口创建之后才会被调用。此过程将有返回值。但我们此处的初始化没那么复杂，现在还用不着担心这个返回值。

```
int InitGL(GLvoid)                  // 开始对 OpenGL 进行所有设置
{
```

下一行启用 smooth shading (阴影平滑)。阴影平滑通过多边形精细的混合色彩，并对外部光进行平滑。我将在另一个教程中更详细的解释阴影平滑。

```
glShadeModel(GL_SMOOTH);           // 启用阴影平滑
```

下一行设置清除屏幕时所用的颜色。如果您对色彩的工作原理不清楚的话，我快速解释一下。色彩值的范围从 0.0f 到 1.0f。0.0f 代表最黑的情况，1.0f 就是最亮的情况。glClearColor 后的第一个参数是 Red Intensity（红色分量），第二个是绿色，第三个是蓝色。最大值也是 1.0f，代表特定颜色分量的最亮情况。最后一个参数是 Alpha 值。当它用来清除屏幕的时候，我们不用关心第四个数字。现在让它为 0.0f。我会用另一个教程来解释这个参数。

混合红绿蓝三原色可得到不同的色彩，因此使用 glClearColor(0.0f, 0.0f, 1.0f, 0.0f)，将用亮蓝色来清除屏幕。如果用 glClearColor(0.5f, 0.0f, 0.0f, 0.0f) 的话，将使用中红色来清除屏幕。不是最亮 (1.0f)，也不是最暗 (0.0f)。要得到白色背景，您应该将所有的颜色设成最亮 (1.0f)。要黑色背景的话，您该将所有的颜色设为最暗 (0.0f)。

```
glClearColor(0.0f, 0.0f, 0.0f, 0.0f); // 黑色背景
```

接下来的三行必须做的是关于 depth buffer（深度缓存）。将深度缓存设想为屏幕后面的层。深度缓存不断地跟踪物体进入屏幕内部有多深。我们本节的程序其实没有真正使用深度缓存，但几乎所有在屏幕上显示 3D 场景 OpenGL 程序都使用深度缓存。它的排序决定那个物体先画。这样您就不会将一个圆形后面的正方形画到圆形上来。深度缓存是 OpenGL 十分重要的部分。

```
glClearDepth(1.0f); // 设置深度缓存  
glEnable(GL_DEPTH_TEST); // 启用深度测试  
lDepthFunc(GL_LESS); // 所作深度测试的类型
```

接着告诉 OpenGL，我们希望进行最好的透视修正。这会十分轻微地影响性能，但使得透视图看起来好一点。

```
glHint(GL_PERSPECTIVE_CORRECTION_HINT, GL_NICEST);  
// 真正精细的透视修正
```

最后 我们返回 TRUE。如果我们希望检查初始化是否 OK，我们可以查看返回的 TRUE 或 FALSE 的值。如果有错误发生的话，您可以加上您自己的代码返回 FALSE。目前，我们不管它。

```
return TRUE; // 初始化 OK  
}
```

下一段包括了所有的绘图代码。任何您所想在屏幕上显示的东东都将在此段代码中出现。以后的每个教程中我都会在例程的此处增加新的代码。如果您对 OpenGL 已经有所了解的话，您可以在 glLoadIdentity() 调用之后，返回 TRUE 值之前，试着添加一些 OpenGL 代码来创建基本的形。如果您是 OpenGL 新手，等着我的下个教程。目前我们所作的全部就是将屏幕清除成我们前面所决定的颜色，清除深度缓存并且重置场景。我们仍没有绘制任何东东。

返回 TRUE 值告知我们的程序没有出现问题。如果您希望程序因为某些原因而中止运行，在返回 TRUE 值之前增加返回 FALSE 的代码告知我们的程序绘图代码出错。程序即将退出。

```

int DrawGLScene(GLvoid)           // 从这里开始进行所有的绘制
{
    glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);
                           // 清除屏幕和深度缓存
    glLoadIdentity();           // 重置当前的模型观察矩阵
    return TRUE;               // 一切 OK
}

```

下一段代码只在程序退出之前调用。`KillGLWindow()` 的作用是依次释放着色描述表，设备描述表和窗口句柄。我已经加入了许多错误检查，如果程序无法销毁窗口的任意部分，都会弹出带相应错误消息的讯息窗口，告诉您什么出错了，这样在代码中查错变得容易多了。

```

GLvoid KillGLWindow(GLvoid)       // 正常销毁窗口
{

```

我们在 `KillGLWindow()` 中所作的第一件事是检查我们是否处于全屏模式。如果是，我们要切换回桌面。我们本应在禁用全屏模式前先销毁窗口，但在某些显卡上这么做可能会使得桌面崩溃。所以我们还是先禁用全屏模式。这将防止桌面出现崩溃，并在 Nvidia 和 3dfx 显卡上都工作得很好！

```

if (fullscreen)                  // 我们处于全屏模式吗？
{

```

我们使用 `ChangeDisplaySettings(NULL, 0)` 回到原始桌面。将 `NULL` 作为第一个参数，`0` 作为第二个参数传递强制 Windows 使用当前存放在注册表中的值（缺省的分辨率、色彩深度、刷新频率，等等）来有效的恢复我们的原始桌面。切换回桌面后，我们还要使得鼠标指针重新可见。

```

ChangeDisplaySettings(NULL, 0);   // 是的话，切换回桌面
ShowCursor(TRUE);              // 显示鼠标指针
}

```

接下来查看是否拥有着色描述表 (`hRC`)。没有则跳转至后面查看是否拥有设备描述表。

```

if (hRC)                        // 我们拥有着色描述表吗？
{

```

如果存在着色描述表的话，下面的代码将查看我们能否释放它（将 `hRC` 从 `hDC` 分开）。这里请注意我使用的的查错方法：基本上我只是让程序通过调用 `wglGetCurrent(NULL, NULL)` 尝试释放着色描述表，然后我再查看释放是否成功，巧妙地将数行代码结合到一行。

```

if (!wglGetCurrent(NULL, NULL))   // 我们能否释放 DC 和 RC 描述表？
{
    MessageBox(NULL, "Release Of DC And RC Failed.",

```

```
        "SHUTDOWN ERROR" , MB_OK | MB_ICONINFORMATION );
}
```

下一步我们试着删除着色描述表。如果不成功的话弹出错误消息，然后把 hRC 设为 NULL。

```
if (!wglDeleteContext(hRC))           // 我们能否删除 RC?
{
    MessageBox( NULL, "Release Rendering Context Failed.",
               "SHUTDOWN ERROR" , MB_OK | MB_ICONINFORMATION );
}
hRC=NULL;                           // 将 RC 设为 NULL
}
```

现在查看是否存在设备描述表，有则尝试释放，不能则弹出错误消息，然后把 hDC 设为 NULL。

```
if(hDC&&!ReleaseDC(hWnd,hDC))      // 能否释放 DC
{
    MessageBox( NULL, "Release Device Context Failed.",
               "SHUTDOWN ERROR" , MB_OK | MB_ICONINFORMATION );
    hDC=NULL;                         // 将 DC 设为 NULL
}
```

现在查看是否存在窗口句柄，调用 DestroyWindow(hWnd) 来尝试销毁窗口。如果不能，则弹出错误窗口，然后把 hWnd 设为 NULL。

```
if(hWnd&&!DestroyWindow(hWnd))      // 能否销毁窗口
{
    MessageBox( NULL, "Could Not Release hWnd.",
               "SHUTDOWN ERROR" , MB_OK | MB_ICONINFORMATION );
    hWnd=NULL;                        // 将 hWnd 设为 NULL
}
```

最后要做的是注销我们的窗口类。这允许我们正常销毁窗口，接着在打开其他窗口时，不会收到诸如“Windows Class already registered ((窗口类已注册)”的错误消息。

```
if(!UnregisterClass("OpenGL",hInstance))// 能否注销类
{
    MessageBox( NULL, "Could Not Unregister Class.",
               "SHUTDOWN ERROR" , MB_OK | MB_ICONINFORMATION );
    hInstance = NULL;                  // 将 hInstance 设为 NULL
}
}
```

(未完待续)

模式罗汉拳

Immutable 模式与 string 类的实现

作者：[透明](#)

前言：在 C++ 中要实现引用计数、共享内存的 string 类，对象的生存期管理是一个大问题。但是，使用 Immutable 模式之后，情况将得到大大改善。

梗概

禁止改变对象的状态，从而增加共享对象的坚固性、减少对象访问的错误，同时还避免了在多线程共享时进行同步的需要。

实现方法：在对象构造完成以后就完全禁止改变任何状态信息。如果需要改变状态，则生成一个状态与原对象不同的新对象。

场景

假设你正在为一家游戏公司开发一个和外太空、宇宙飞船有关的游戏，当然你有必要用某种方式来表示一艘宇宙飞船（不管它是属于地球人的还是属于外星人的）所处的位置。很自然的，你决定编写一个Position类。从一个Position对象应该可以查询到当前位置的x坐标和y坐标（我们的游戏比较简单，二维地图，呵呵），还应该可以根据输入的偏移量得到新的位置。很正确的设计，不是吗？（见例1）

例1：Position的设计

```
class Position{  
private:  
    int x, y;      //简单点，用整型数来表示坐标  
public:  
    Position(int x, int y){ //ctor需要两个参数。  
        this->x = x;  
        this->y = y;  
    }  
    int getX( ){ return x; }  
    int getY( ){ return y; }  
    void Offset(int offX, int offY){ //根据偏移量得到新的位置  
        x+=offX;  
        y+=offY;  
    }  
}
```

但是，如果我们的Position需要在多线程环境下使用，它能保证线程安全吗？答案是很明显的No！如果两条线程同时调用同一个Position对象的Offset函数，你就无法保证得到的结果是什么了。所以，为了保证线程安全，也许你还会想给Offset函数加上同步机制——麻烦了！

换个角度想想怎么样？假如我们根本不让Offset函数修改Position的内容？假如我们让Offset函数生成一个新的Position对象？如果是这样，Position对象就已经是线程安全的了——它没有任何“写”操作，而没有写操作的类是不需要同步的。于是我们这样做了，并且很轻松的得到了一个线程安全的Position类。（见例2）

例2：线程安全的Position类（这里只展示Offset函数）

```
Position Position::Offset(int offX, int offY){ //根据偏移量得到新的位置
    return Position(x+offX, y+offY);
}
```

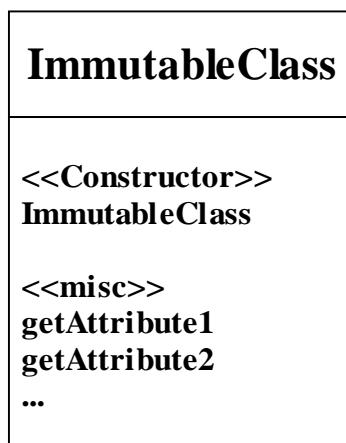
约束

- 你有一个天性被动的类。这个类的实例不需要改变自己的状态。同时这个类的实例还被其他多个对象共享。
- 正确协调被共享的对象的状态改变非常困难。当一个对象的状态发生改变时，所有使用它的对象都应该得到通知。这造成了对象之间的紧耦合。
- 在多线程共享时，还需要使用同步机制来保证线程安全性。

解决方案

为了避免状态改变带来的诸多麻烦，不允许对实例的状态做任何修改。具体的做法就是：不在类的公开接口中出现任何可以修改对象状态的方法，只出现状态读取方法。如果client需要不同的状态，就生成一个新的对象。（见图1）

图1：Immutable 模式的类图



效果

- 不再需要协调状态修改的代码，也不再需要协调任何同步代码。
- 生成了更多的对象。增加了对象生成和销毁的开销。

实现

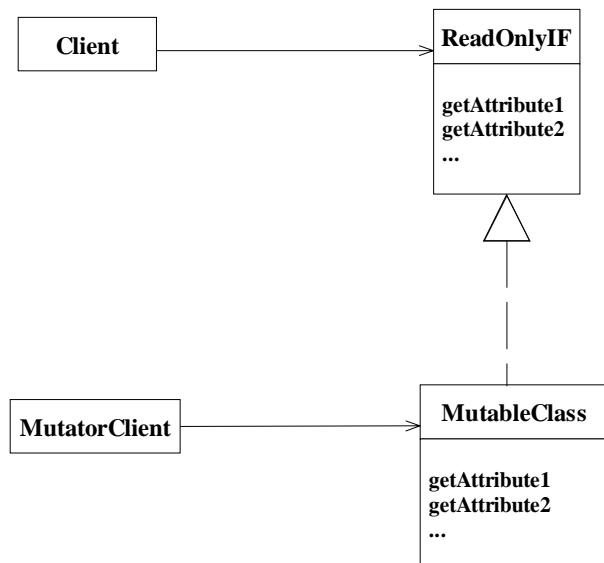
Immutable 模式的实现主要有以下两个要点：

1. 除了构造函数之外，不应该有其它任何函数（至少是任何 public 函数）修改任何成员变量。
2. 任何使成员变量获得新值的函数都应该将新的值保存在新的对象中，而保持原来的对象不被修改。

在“效果”中我已经讲到：Immutable 模式会大大提高对象生成和销毁的频率。因此，在 C++ 中实现 Immutable 模式时，还必须特别注意对象的生存周期。你可以尝试用智能指针[Meyers96, Item28]来帮助你处理对象的销毁问题，但是无论如何你都必须仔细检查以确保没有内存泄漏——如果每艘飞船的每次移动都会造成内存泄漏，你的游戏该是多么糟糕！

此外，Immutable 模式还有一种变体：Read Only Object 模式。它的做法是：当一个类的对象对于某些 client 可写、某些 client 不可写时，让这个类实现一个 ReadOnly 接口。然后让可写的 client 直接访问对象，而让不可写的 client 通过 ReadOnly 接口访问该对象，从而实现了不同的读写权限控制。（如图 2 所示）

图 2 : Read Only Object 模式



Immutable 模式与 string 类的实现策略

如果你也读过[Meyers96]，我想你一定对那个应用在 String 类上的 COW (Copy-On-Write) 策略[Meyers96, Item29]印象深刻。COW 策略是“lazy evaluation”的发展形式。如果对 String 类的写操作数量很少，那么 COW 策略将大大提高整个 String 类的效率，并大大降低空间开销。

可是你知道吗？在 STL 中的 string 类并没有采用 COW 策略，从例 3 就可以看出这一点。为什么？为什么这么好的策略没有得到采用？相信你从[Meyers96]中便可发现：实际在 String 类上实现 COW 策略是如此复杂。更何况我们还必须考虑线程安全的问题。我完全有理由认为：正是因为考虑到这些复杂的情况，STL 的实现者们才最终决定用一个比较低效但是安全的实现方案。

例 3：STL 中的 string::operator= 和 string::operator[]

```
//下面代码出自 SGI STL 2000 年 6 月 8 日版本
//为了帮助读者理解，我做了些微改动，并在关键位置加上注释

//如果使用 COW 策略，operator= 应该不做内容复制，而是进行引用计数
string& string::operator=(const string& s) {
    if (&s != this)
        assign(s.begin(), s.end()); // 这里的 operator= 只是简单的内容复制而已
    return *this;
}

//如果使用 COW 策略，const 的 operator[] 和非 const 的 operator[] 应该不同
//但是这里两个 operator[] 完全相同
const char & string::operator[](int n) const
{
    return *(_M_start + n); } // _M_start 是字符数组的起始位置
char & string::operator[](int n)
{
    return *(_M_start + n); }
```

看到这些，我不能不开始猜想：为什么 STL 的设计者们一定要保留这些给他们造成麻烦的“修改函数”（即可以修改 string 内容的函数）？我想，这是因为他们希望让 string 的行为方式尽量接近于 C 语言的 char *型字符串。不然，我真的想不出其他任何保留 operator[] 的理由。

那么，如果不非要让 string 类的行为方式接近 char *型字符串，如果 string 类的读操作应用频率远远大于写操作（在实际应用中这是很常见的），你会考虑如何实现一个 string 类？啊，也许你已经想到了：Immutable 模式。你可以很舒服的使用[Meyers96]教你的引用计数方法来节约存储空间，你不必再担心写操作的同步问题或别的什么，因为已经没有写操作。任何改变字符串内容的操作都将得到一个新的 string 对象。而对象生存期管理和存储空间管理这两个大问题也因为 Immutable 模式的引入而大大简化，你完全可以参照[Meyers96]第 183 页到第 189 页的内容自己来解决它们。

代码示例

我用了一天的时间，做了一个简单的 `ImmutableString` 实现。其中实现细节用了 `Proxy` 类 [Meyers96]，并参考了 COM 的引用计数规则 [Pan99]。在这个例子中，读者可以感觉到：`Immutable` 模式大大简化了共享空间的字符串类型的实现，并为其中的一些方法（比如 `subString`）的实现提供了非常大的便利。本来我想把代码放在文章里面，但是时间和空间受限，最后决定放弃。如果读者有兴趣，可以到下面 URL 去下载一份。

<http://gigix1980.home.sohu.com/ImmutableString.zip>

在该代码中，我做了一个简单的效率测试：反复进行字符串对象的赋值 (`operator=`) 操作。结果表明：`ImmutableString` 的效率比 `std::string` 高出了一倍左右。假如你的业务就是不断的读取数据库、不断的赋值、不断的输出，而不对字符串进行修改，那么 `ImmutableString` 的效率提升是非常可观的。

该示例代码在 VC .NET 下编译通过。

相关模式

经常会使用 Abstract Factory 模式[GOF95]来创建新的对象。

大量的对象经常通过 Flyweight 模式[GOF95]被共享。

参考书目

[Meyers96] Scott Meyers, *More Effective C++*, Addison-Wesley, 1996.

[GOF95] Erich Gamma etc., *Design Patterns: Elements of Reusable Object-Oriented Software*, Addison-Wesley, 1995.

中译本：《设计模式：可复用面向对象软件的基础》，李英军等译，机械工业出版社，2000 年 9 月。

[PAN99] 潘爱民，《COM 原理与应用》，清华大学出版社，1999 年 11 月。

阿 P 正传

创业初期

——创建型模式

小飞侠

作者的话：想学习 Design Patterns 吗？《设计模式》（英文 *Design Patterns*）一书无疑是本上乘之作，而国内出版的设计模式书籍也就这么一本（据我所知到现在为止是如此）。John Vlissides 建议学习设计模式的三本书依次是：1. *Design Patterns Explained*；2. *Design Patterns*；3. *Patterns Hatching*。设计模式的初学者直接跳到 2 学习可能会感到难度颇大，无奈没有选择只好硬着头皮看（引用孟岩先生的话：“但那本书有一个缺点，不好懂。从风格上讲，该书与其说是为学习者而写作的教程范本，还不如说是给学术界人士看的学术报告。这一点包括该书作者和像 Bjarne Stroustrup 这样的大师都从不讳言：）“阿 P 正传”专栏就是针对这个问题，用非常好懂的故事（算是吧）来讲述各种模式。翻出一句老话：“授人与渔，而非授人与鱼。”渔，OO 思想是也；鱼，模式是也。模式不是 Bible，在《设计模式》一书中的 23 个模式都是根据已有经验总结出来并分类而得，所以说模式的真谛在于 OO 思想而非模式本身。但注意了，正所谓鱼与熊掌不可兼得，文章不会涉及任何应用问题，关于细节也不会过多涉及，所以您最好有《设计模式》这本书。

文中所有的图均是针对本文的例子而画，如要看“纯”的图请参考《设计模式》；为了能使图更好懂，我会使用《设计模式》中没有的调用箭头来代替函数的伪代码注解或指向构造函数的实例化箭头，这些箭头用彩色。其他符号和《设计模式》书中一致。

阿 P，咱们的主人公，是台式 PC 制造厂的 CEO，所有台式 PC 的配件诸如：CPU，RAM……都由公司自己制造（也就是它们都是类，需要的时候就 new）。

```
class CPU{    //abstract class
public:
    virtual void setcpu() = 0;
};

class CPU_750M : public CPU{
public:
    void setcpu(){
        //implementation
    }
    CPU_750M(){
        //implementation
    }
private:
```

```
//CPU DATA  
};
```

在创业初期阿 P 决定制造编程用的电脑（众人：连 CPU 都能自己造还说创业初期：P）这种情况下 productPC() 函数就能解决问题：

```
void productPC(){  
    CPU* handler_CPU = new CPU_750M;  
    handler_CPU->setcpu();  
    //RAM,DISK...  
}
```

之所以决定生产编程专用电脑是因为阿 P 想起了编程的日子——那些破烂机器根本不适合用来编程！想想有些 C++ IDE 带来的无聊和而可悲的等待吧！

编程专用电脑推出不久之后阿 P 收到了不少 E-MAIL，都是感谢，赞美之类的言辞，说阿 P 拯救了处于水深火热之中的程序员！编程专用电脑实在是太棒了！

“嘿……”阿 P 得意极了。公司知名度提高了不少，阿 P 的钱包厚度也增加了 N 倍。CEO 就是 CEO，阿 P 没有被胜利冲昏头，决定使产品多样化，生产 GAMEPC 和 CG 工作站。这就要求对 productPC() 作更改，生产 GAMEPC 时把 productPC() 的实现改为针对 GAMEPC 的，生产 CG 工作站时……不过你马上打消了这个愚蠢的念头。要么分别针对要生产的每一种 PC 专门写一个 productPC_???() 函数，但你发现这不利于以后的发展，如果有两种几乎一样的 PC 要生产，在你写出第一个 productPC_???() 后，不得不又重头写一个几乎一样的 productPC_???()。

“哦，效率太差了，太浪费资金了！”阿 P 叫到。没办法，只好请了一位专家来改进制造方式。“虽然得花很多银子，不过希望值得，咳咳……”

几天后，改进方案摆到了办公桌上，阿 P 看完后很满意：这个方案把 productPC() 封装到一个类 PRPC 中，productPC() 现在不参与 new 操作，它只负责设置个配件 (set???())，而相应的 new 操作（也就是需要变动的部分）由类的虚函数来完成。想制造 GAMEPC 只需继承 PRPC 并改写 PRPC 的虚函数即可。

```
/*FACTORY METHOD 模式*/  
class PRPC{//PRoduct PC  
public:  
    void productPC();//为什么不用构造函数？绝大部分 productPC() 都一样，  
                    //如果用 constructor 还要为每个一样的实现重写一遍  
//or protected:  
    virtual CPU* MakeCPU() = 0;  
    virtual RAM* MakeRAM() = 0;  
    //...
```

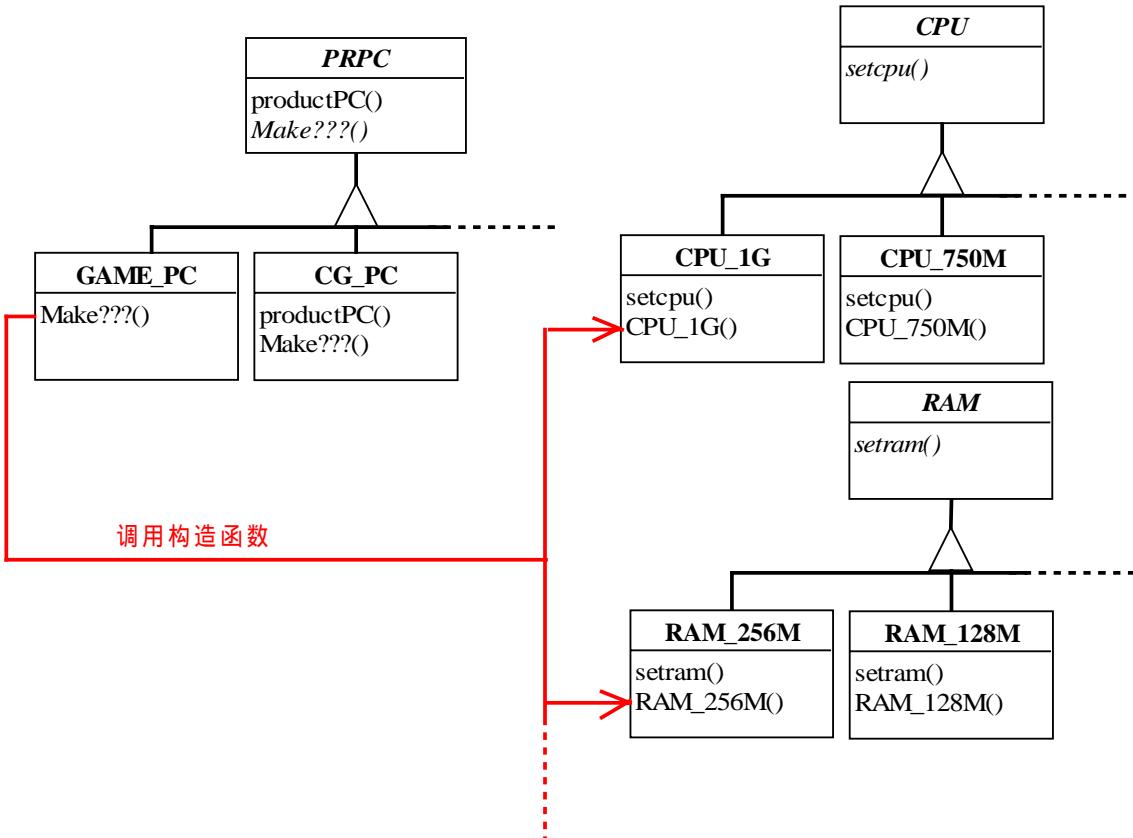
```

};

class GAME_PC : public PRPC{
public:
    CPU* MakeCPU(){
        return new CPU_1G;
    }
    RAM* MakeRAM(){
        return new RAM_256M;
    }
    //...
};

void PRPC::productPC(){
    CPU* handler_CPU = MakeCPU();
    handler_CPU->setcpu();
    //RAM,DISK...
}

```



调用 `GAME_PC::productPC(); PRO_PC::productPC(); CG_PC::productPC();` 即可！
不过在 `CG_PC` 的 `productPC()` 中有一个问题：CG 工作站的组装调试比较复杂，在基类定义的

`productPC()`不能满足需要，不过没关系，在 `CG_PC` 中对 `productPC()` 的实现重写就是了（见 *Effective C++ ITEM 37 & Design Patterns p73* 的实现）

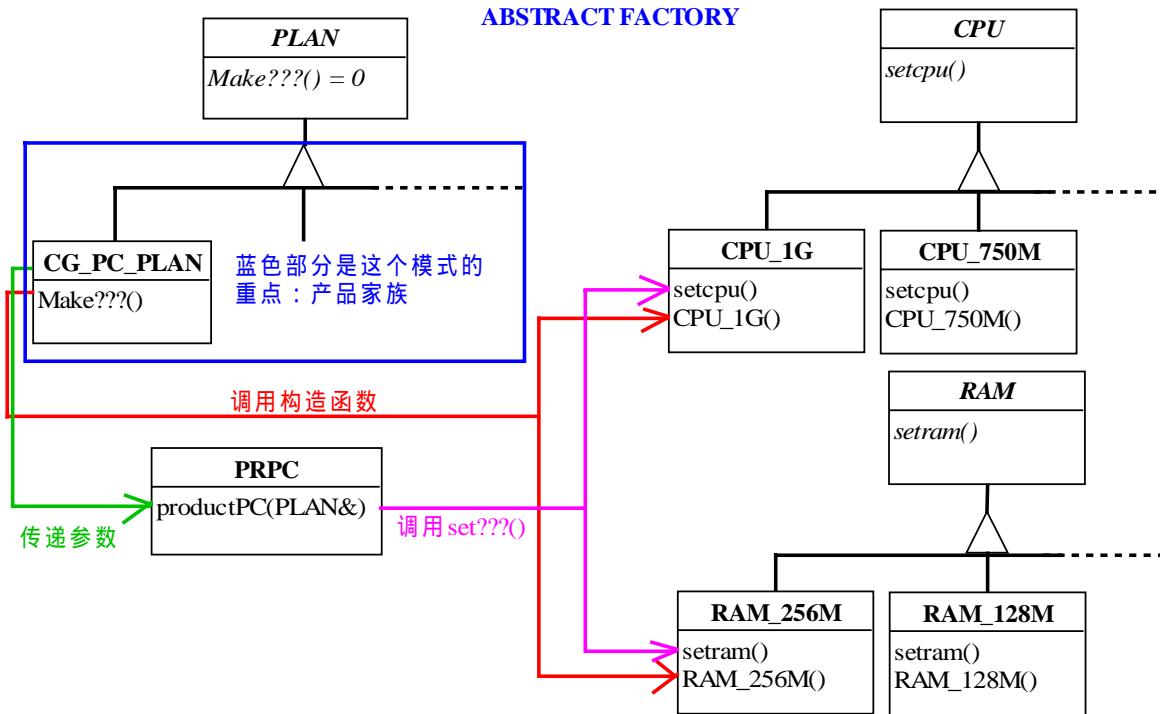
经过阿 P 的领导，公司的市场占有率急速上升。阿 P 每天做梦都在笑……可惜好日子总是太短，`GAMEPC` 的业绩严重下滑，这使阿 P 大为恼怒。毕竟身为 CEO，阿 P 并没有失去理智，开始对市场进行调查。调查发现市场对 `GAMEPC` 的品种需求越来越丰富，早已不满足一种 `GAMEPC` 了。阿 P 决心要好好教训竞争对手，是他们让玩家的口味越来越挑剔。是生产新系列 `GAMEPC` 的时候了，这一个新系列的 `GAMEPC` 继承自 `GAME_PC` 类。可阿 P 发现这非常不划算：要生产的新系列 `GAMEPC` 之间差异比较小，可不得不分别为它们创建厂房，这个大问题使阿 P 开始失去理智，在会议上指责 `FACTORY METHOD` 的设计者是个白痴。

这一次，阿 P 聘请了一位知名专家来帮助公司改进生产方式。在聘请他之前阿 P 调查了好久，这次再也不能把宝贵的银子花在白痴身上了。这个设计就是 `ABSTRACT FACTORY`。为了能让 `productPC()` 适应性更强，方案把虚函数的部分迁到另一个类中，让 `productPC()` 接受那个类类型的参数。这就好比那个类是生产计划，生产什么样的产品全在计划里，`productPC()` 接受计划生产产品，而不同的生产计划只需改写那个新分离出的类即可。

```
/*ABSTRACT FACTORY 模式*/
class PRPC{
public:
    void productPC(PLAN& C){ //C is Concrete
        CPU* handler_CPU = C.MakeCPU();
        handler_CPU->setcpu();
        //RAM,DISK...
    }
};

class PLAN{
public:
    virtual CPU* MakeCPU() = 0;
    virtual RAM* MakeRAM() = 0;
    // MakeDISK... (一系列的 Make???函数)

    //看出 ABSTRACT FACTORY 的弱点了吗？接口规范是已经定好了的
    //，想要生产音乐工作站有 MIDI 键盘这种东西怎么办？增加新的虚
    //函数？这样的话就要写另外一个 productPC 函数，其参数类型为
    // MUSIC_PLAN& 导致继承体系的复杂度增加。ABSTRACT FACTORY
    //的目的在于创建家族 (family) 对象
};
```



在改造 CG_PC 时新的麻烦又来了，CG_PC::productPC() 的实现不同于其他，不过知名专家还是有两把刷子的，他把组装设置 (set???() 操作) 也移到了 PLAN 中：

```
/*BUILDER 模式*/ (实际是 ABSTRACT FACTORY 与 BUILDER 的综合)
class PRPC{
public:
    void productPC(APLAN& C){
        C.buildCPU();
        C.buildRAM();
        //...
    }
};

class APLAN{//Advanced PLAN
public:
    virtual void buildCPU() = 0;
    //...
};

class CG_PC_APLAN : public APLAN{
public:
    void buildCPU(){
        CPU* handler1_CPU = new CPU_1G;
        RAM* handler2_RAM = new RAM_256M;
        //...
    }
};
```

```

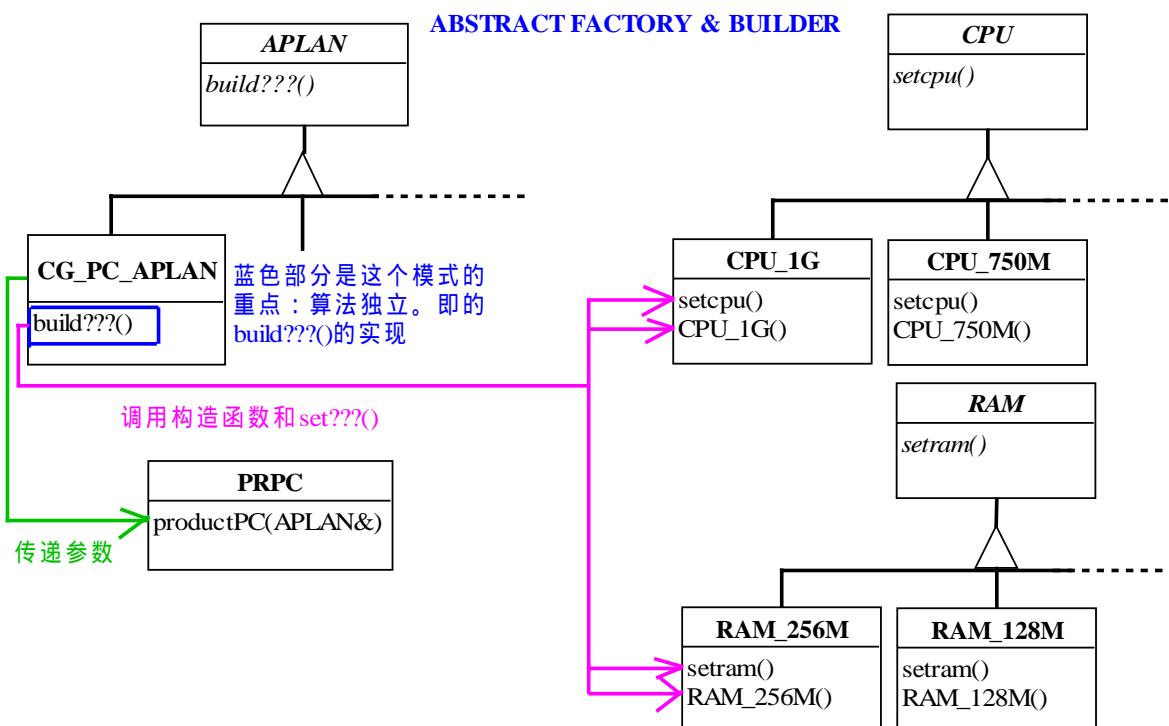
CPU* handler2_CPU = new CPU_1G;
handler1_CPU->setcpu();
handler2_CPU->setcpu();
//RAM,DISK...的new, set操作
    //这就是BUILDER模式的算法独立性，算法（这里
    //是组装调试的方法，即set???函数的调用）
    //与Client无关。ABSTRACT FACTORY与BUILDER //模
    式的区别在于前者返回零件，后者返回成品（组装//调试好的）

}

//buildRAM,DISK...
};

}

```



```

//生产：
PRPC master;
CG_PC_APLAN CPlan;
master.productPC(CPlan);

```

“知名专家就是不一样，太好了……” 经过这些折磨，身为 CEO 的阿 P 也有些底气不足了。还好改进很成功，公司的市值几乎翻了一番，阿 P 又重温了梦里大笑的感觉。不过阿 P 并没有满足于现有的成绩，决定打进笔记本电脑市场，但是笔记本的 CPU、RAM 等与台式机都不一样，为了尽快打进市场，阿 P 可不想研制自己的???ForNote 类，而是购买外公司的产品。好，现在继承 A_PLAN

产生一个新类 NOTE_A_PLAN，在它的虚函数中 new ???ForNote;这个“计划书”让工厂去做 new ???ForNote。

但是，问题来了。由于并没有???ForNote 类，这个在具体制造过程中的 new ???ForNote; 是个错误！也就是说“实现级”的代码并不知道如何生产需要的东西或是不想与此有关联，这就需要把 new 操作转移到客户端的代码中，这样具体的创建过程就与我们无关了（既然我们 new 出了产品怎么还说与创建过程无关呢？请见后记）。供货厂商在产品类中提供了一个接口：Clone()，它的作用是返回自己，这个函数看起来像这样（该函数是拷贝构造函数，关于深浅拷贝问题请看 *Design Patterns*）：

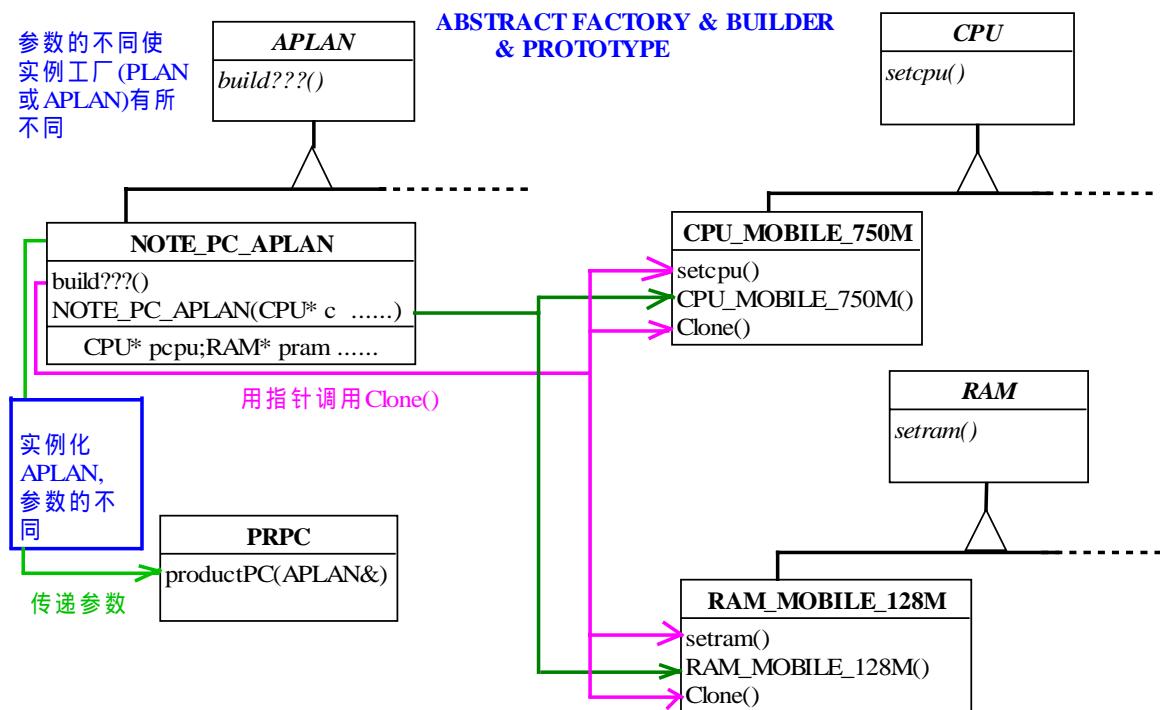
```
???ForNote* Clone(){return ???ForNote(*this);}

/*PROTOTYPE 模式*/ (ABSTRACT FACTORY, BUILDER 和 PROTOTYPE 的综合)
class NOTE_APLAN : public APLAN{
public:
    NOTE_A_PLAN(CPU* ,RAM* ...);
    void buildCPU(){ //返回类型 CPU* ，感谢标准化！
        CPU* handler_CPU = pcpu->Clone();
        handler_CPU->setcpu();
    }
    //...
protected:    //会有 NOTEPC 的新系列
    CPU* pcpu;
    RAM* pram;
    //...
};

NOTE_A_PLAN::NOTE_A_PLAN(CPU* c, RAM* r, ...){
    pcpu = c;
    pram = r;
    //...
}
```

相应的 concrete plan:

```
NOTE_A_PLAN(new CPU_MOBILE_750M,new RAM_MOBILE_128M, ...) CPlan;
//只在调用级做 new 动作,与工厂里的实现无关
```



没过多久，笔记本电脑销售排行榜上就有了公司的产品。阿 P 没有闲着，开始考虑公司下一步拓展计划.....

(未完待续)

后记：乍一看，FACTORY METHOD 是个“完美”的模式，ABSTRACT FACTORY 和 BUILDER 模式能做到的它也可以，而且还可以解决产品体系无法构成家族的问题。但是，如果家族产品类层次很庞大，那么相对应的 Creator(concrete factory, in story is GAME_PC, CG_PC...) 的继承继承.....会引起继承爆炸，并且爆炸发生在客户端（你得自己动手继承所以直接影响你）。滥用继承是件蠢事，不是吗？相应的情况发生在 Abstract Factory 或 BUILDER 模式，那么继承爆炸发生在“实现级”(concrete factory)，Client 不必理会 concrete factory 的继承，但是要和众多的 concrete factory 打交道，用它们做参数。PROTOTYPE 模式可以避免这个问题。它的 concrete factory 中 new 出的产品不在实现级而在客户端，也就是说 concrete factory 生产产品的方式不是定死的，想更改它吗？不必继承，只需在 concrete factory 实例化时的参数上做手脚。值得注意的是：客户端的 new 引发的构造函数应该尽可能简单，这个 new 的目的只是为了让指针取用 clone()。真正的构造应该发生在 clone()里面的拷贝构造中。你看出了 PROTOTYPE 模式的弱点了吗？它不能对 Client 隐藏产品类，你得和众多的产品打交道（这个总数当然比 PLAN 还要多的多），而且有些产品类中没有 clone()接口，你又无法更改它们，这样就不能使用 PROTOTYPE。所以在使用它们时要找一个平衡点。

作者：同步快梭

提起 Qt，可能很多人都不太熟悉或者没听说过，它是什么？它能为我们干什么？

如果把 Qt 比喻成 VC 中的 MFC，你可能就可以大概了解它的功能与定位，尽管这样的比喻并不精确。

Qt 是挪威 Trolltech 公司开发的面向对象、高度封装的跨平台 C++ 图形用户界面应用软件框架，它为用户提供了一个开发商业级别、多平台应用软件的开发环境。目前支持以下三个平台：

- **Windows** 平台，包括 Windows 95/98/ME, NT4, 2000 和 XP。
- **X11** 平台，包括 Linux, Solaris, HP-UX, Irix, AIX 以及其他一些 Unix 的变种。
- **Mac** 平台。

此外 Qt 还为嵌入式应用程序的开发提供了一个嵌入式版本。

用过 linux 的用户都知道 linux 发行商一般提供两套图形管理程序，一个是 kde，另一个是 gnome，其他还有一些，但以这两个为主。Kde 就是以 Qt 为基础设计的一套管理器。

截止到本稿的写作日，Qt 的最新版本为 3.0 正式版，由于 Trolltech 是一个商业公司，因此 Qt 并不完全按 GPL 的准则发行。最初发行 Qt 时源代码是不公开的，这导致了部分 linux 开发者的反感，他们决定重新设计一个图形管理器与 kde 抗衡，这就是 gnome 的由来。

现在 Qt 通过三种许可协议发放：

- 1、商业开发，使用对象是开发商用软件的程序员；
- 2、教学专用，使用对象是用来教学 Qt 的各个学院、大学；
- 3、GPL，使用对象是自由软件的开发者。

用收费差异来衡量的话，可分为企业版、专业版与免费版，其中 X11 平台下的各个版本都提供源代码。Windows 平台下只有企业版与专业版提供源代码且版本并不同步，如当前的最新版本为 3.0 版，但 windows 免费版本是 2.3 版。

Qt 的跨平台特征在代码级实现而非运行级，这与 JAVA 有本质区别。Qt 通过各个类向开发人员提供统一的成员函数与变量，底层则使用操作系统相关的代码进行实现，简而概之，就是一次编写，到处编译。因此用 Qt 开发的应用程序其执行效率要比同等的 Java 程序高许多。可用于工控等实时性要求很高的程序。

Qt 的特征包括以下几点：

数据库编程

Qt 3.0 内建了一组独立于各平台和数据库的 API，专门用来调用 SQL 数据库，这组 API

为 Oracle、PostgreSQL 以及 MySQL 提供 ODBC 以及特殊数据库驱动程序支持。Qt 3.0 内置 GUI 和底层数据库同步的数据检测支持功能，而 Qt Designer 完全支持这些新的控制功能，为数据库提供应用软件快速开发工具(RAD)解决方案。

Qt Designer

Qt Designer 是一个全功能的图形用户界面开发工具，与 Delphi 的界面较为相似。它支持包括菜单和工具栏的应用软件主窗口的交互式设计，以及完全支持可定制模式的窗口小部件。此外，Qt Designer 还内置了 C++ 编辑器，允许用户在 RAD 环境中直接编辑源代码。

Qt Linguist

Qt Linguist 是一个本地化工具，能够让用户把基于 Qt 开发的程序从一种语言简单、智能地转变成另外一个语言，适合于开发国际版软件。它能够把程序中所有可见的文本转换成任何支持统一字符编码标准(Unicode)和指定平台的语言，它最主要的特征是一个适应特殊目的编辑工具和多语言术语智能数据库。一旦完成新的翻译，数据库将保存这些术语，以便以后再次使用。此外，Qt Linguist 还完全支持 Unicode 3。

Qt Assistant

Qt Assistant 是 Qt 3.0 提供的一个独立应用软件，它能够浏览 Qt 的类文档，Qt Designer 和 Qt Linguist 手册。此外，它还提供了目录检索，内容纵览，书签，历史记录以及在页面内搜索等功能。

国际化文本显示

Qt 3.0 支持多内码混合的文本，设置是在系统没有安装 Unicode 字体的情况下，以及完全支持 right-to-left 和 bi-directional 型的语言，像阿拉伯语 (Arabic) 和希伯来语 (Hebrew)。

支持 HTTP 网络协议

Qt 3.0 的网络编程模块提供一个通过 HTTP 协议交换数据的 API (以前版本已经实现 FTP 协议)。

支持多显示器

Qt 3.0 允许应用软件支持多个显示器。在 Unix 平台上，Qt 3.0 支持 Xinerama 和传统的多显示器技术，而 Windows 平台上则是 Windows 98 和 2000 支持的虚拟桌面技术。Qt 3.0 提供一个独立于系统平台的 API 以实现上述技术。

新的组件模式

这个特征类似于 Windows 下的 COM(虽然 COM 也号称平台无关，但目前似乎仅在 Windows 平台上应用)，Qt 3.0 提供一个独立于系统平台的 API，以现实共享库加载等功能。

美观的 GUI 界面实现

Qt 3.0 支持浮动窗口，扩展了风格引擎，支持大量的标准窗口部件，包括进度显示条 (progressbar)，旋转框(spinbox)以及表格标题 (table header) 等。此外，它还为交互式文本编辑增加了图形界面控制。

可接近性支持

Qt 可控制与提供有关可接近性体系结构的信息，通过 Qt 提供的标准工具可开发视觉或肢体残疾用户使用的应用软件（例如 Windows Magnifier 和 Narrator）。

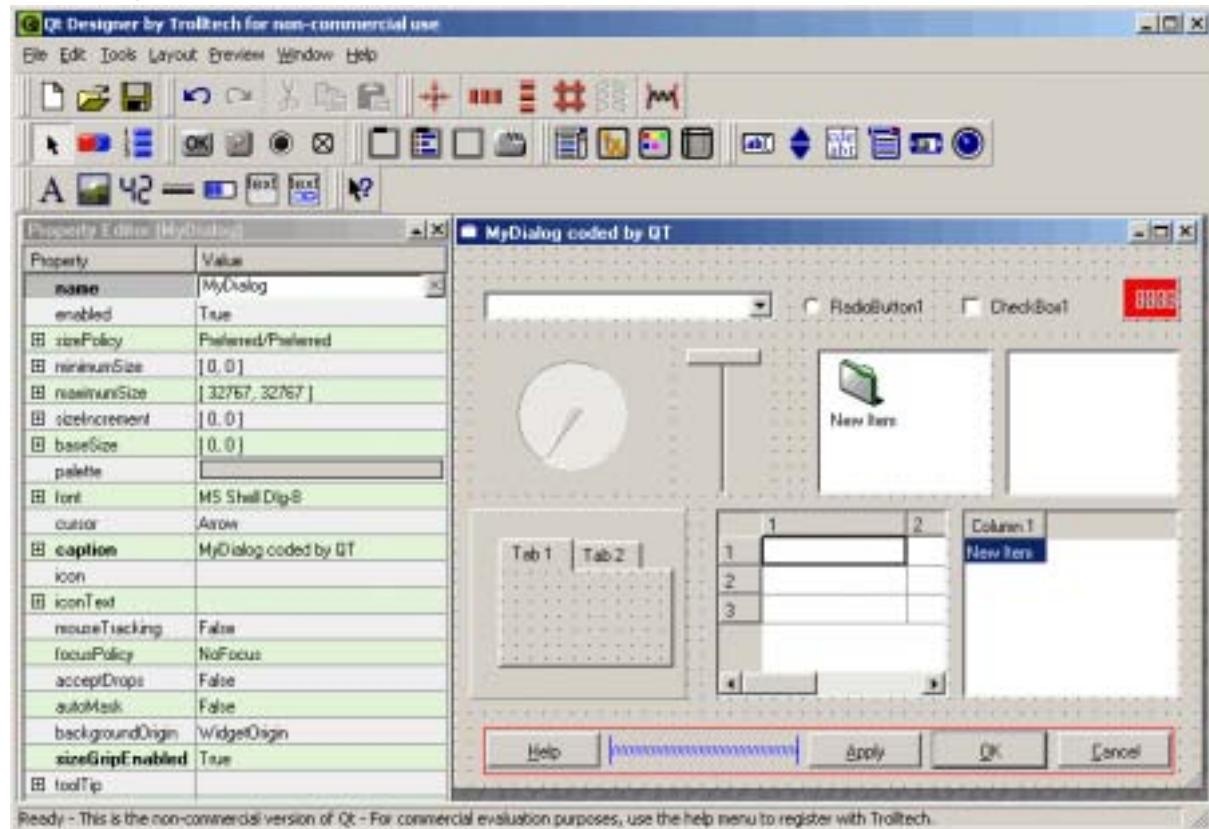
64 位安全

Qt 3.0 支持即将推出的最新一代 64 位硬件。

Qt 是我所接触的一个很好的类库（可能用平台更合适），虽然我用得最多的还是 MFC。这个很好并不单单是它的类可以为我们的程序开发减少工作量，当然这也是一个原因。更重要的是我们可以从 Qt 本身学到很多体系结构的设计。虽然 Linux 本身就是一个很好的体系，但如果一个开发人员要从 Linux 的原码开始学起全盘掌握的话可能要花费相当多的时间。Qt 整套架构不算小，这使得它有着丰富的功能，诸如图形、网络、数据库、各种控件、IO 操作以及与 COM 类似的接口，但是也并不是非常的庞大以至于大多数人搞上两三年还搞不清楚其细节。Qt 是面向对象设计的，各种功能封装在类中，整套架构清晰明了，和 VC 相比 VC 似乎隐藏了更多的细节。同时 Qt 是开放源码的，又有商业公司对它精心呵护，加上世界各地开发人员的踊跃参与使它兼具实用性与理论性。用心地体会 Qt，我们可以在较短的时间内提升自己的总体设计水平，而不仅仅是编程。

备注：Qt 官方站点为 <http://www.trolltech.com>，相关软件可在其上下载。Qt 的中文资料极少，我和一些网友翻译了一部分，在 <http://www.joyinternet.net/cn/qt>（编者：也可从 C++ View 网站 Resources 中找到链接），如有兴趣翻译文档的朋友请与我联系：feedback@joyinternet.net 或 qq: 14504123。

附 Qt Designer 的界面快照。



C++ VIEW



WTL 之父 Nenad Stefanovic 访谈录 02

WTL 简介 14

鸟鸣涧：第 5 期问题解答 17

使用 Qt 与 MySQL C API 开发 MySQL 查询器 26

Generic<Programming>：volatile——编写多线程程序的好帮手 35

模式罗汉拳：Composite 模式与自动化测试框架的实现 44

天方夜谭 VCL：生死 50

6

WTL 之父 Nenad Stefanovic 访谈录

采访、整理：myan

翻译：cber

myan 引介

作为现代 C++最重要的特色技术，template 正在各个传统领域攻城略地。从基本算法与数据结构，到正则表达式与 XML 解析，从高性能数学计算，到资源的分配与管理，从网络分布式计算环境，到组件模型创建，从静态多态性的维度扩展，到设计模式的自动生成，神奇的 template 显示出其令人叹为观止的强劲实力，如果不是有一个隐隐的痛处，template 爱好者简直可以去狂欢了。

这个隐隐的痛处，就是在 GUI 编程领域。

现有的大部分成熟 GUI 框架和工具库，其定型时间都在 90 年代早期，不管是为什么原因，总之我们根本看不到 template 技术在这些环境中的任何重要运用。无论是专有 MFC 和 OWL，还是开源的 wxWindow 和 Mozilla，以至于是专有还是开源都说不清楚的 Qt，它们在其他方面有着诸多不同，偏偏倒是在对待模板技术上空前一致：严格限制在底层的数据结构领域内，抵制模板技术流入 GUI 主体结构。最过分的 wxWindow 和 Mozilla，在代码编写规范里严厉禁止使用 1990 年之后发展出来的任何 C++特性，模板、异常、多继承、STL 等等，均在黑名单上。诸位有兴趣，不妨去看看，那与其说是一份 C++代码编写规范，倒不如说是对 C++ 现代特性在 GUI 领域应用的一份不公正的判决书。

难道模板技术真的在 GUI 领域无用武之地吗？

WTL 给出了一个响亮的回答。

WTL 是微软 ATL 开发组成员 Nenad Stefanovic 先生在 ATL Windowing 机制上发展起来的一整套 GUI 框架，运用 template 技术组织和创建 GUI 对象，构筑了精致的面向对象框架（没错，在这里 object oriented 与 template 达成了精致的融合）。虽然没有获得微软的官方支持，虽然其使用者人数很少，但是确实是“用过的都说好”，有位微软 MVP 人士甚至说，这是微软有史以来推出的一个最优秀的一个 framework。真是一个有趣的讽刺，最好的东西居然不被官方支持。有关于 WTL 的流言不少，比如这东西原本是微软内部专用，只是因为不小心才被泄漏出来等等，这更加剧它的神秘色彩。相信大家对它一定有不少问题。我们特别邀请到了 WTL 之父 Nenad Stefanovic 先生，进行了一次网上的访谈，希望能帮助大家了解 WTL 的真面目。

【C++ View】: I think most of our readers are not very familiar with you, so would you please tell us your story here? We are very fond of that. What do you think about China and Chinese people?

【C++ View】: 我想，可能我们的读者中有很多人对您还不是很熟悉，您能不能在此给我们简单介绍一下您自己呢？我们将非常乐意听到您的自述。还有，您能不能也对我们讲述一下您对于中国以及中国人民的基本看法呢？

【Nenad】: I am a software developer at Microsoft. Your readers will probably know me as a creator of Windows Template Library, WTL. I am from former Yugoslavia, where I finished school and started working on software development. I've been living in US for 10 years now.

I am intrigued and impressed by the Chinese culture and tradition. I think that China is now in a great position of progress as a country and a nation. I discovered that, being from former Yugoslavia, I already know many things about China, and getting to know people from China gave me a bit of the "everyday" life perspective as well. I'd like to learn more, so I hope to visit China one day.

【Nenad】: 好的。我现在在 Microsoft 工作，是它里面的一个软件开发人员。你们杂志的读者中可能有人知道，我就是 Windows Template Library (WTL) 的创作者。我来自于前南斯拉夫，在那我完成了我的学业并开始了我作为软件开发人员的工作生涯。现在，我在美国居住的时间已经超过了 10 年了。

中国的文化以及传统给我留下了极为深刻的印象，我对此十分感兴趣。我想，作为一个国家以及民族，中国已经处于一个伟大并且不断在成长中的位置上。作为一个从前南斯拉夫来的人，我早就了解到关于中国的很多事情。在与来自中国的人民的接触过程中，我还了解到了你们日常生活的一些状况。我还想了解更多（有关中国的事情），希望有一天我可以到中国来游览。

【C++ View】: When and why did you first thought about WTL? What's its original purpose? How do you see its future?

【C++ View】: 您是什么时候开始想起要开发 WTL 呢？为什么？您在开发它时的最初目的是什么？您又是如何地看待它未来的发展呢？

【Nenad】: WTL was born while I was working on ATL (Active Template Library). We were extending ATL to support ActiveX controls, and I was working on the windowing support. I started thinking that the same techniques can be applied to much broader windowing support, for the much richer UI for controls, components, and also applications. So, WTL was created as a part of ATL that would extend ATL to support any kind of UI related component or application. It did not ship with ATL in Visual Studio, however, so I decided to ship it as a standalone library that extends ATL.

I think that WTL will continue to be a great option for developers writing Windows applications and components. I don't see big changes or additions to WTL, because one of the design principles for WTL was to follow the Win32 UI API and design. It will continue to do so.

【Nenad】：WTL 是我在从事 ATL (Active Template Library) 开发工作时的产物。那时我们正在扩展 ATL，使之得以支持 ActiveX control，而我负责的就是其中对于窗口机制部分的支持。这时，我就开始想，是不是可以把同样的技术应用到更为广泛的窗口机制中，以获得更丰富的 UI 控制、组件、以至于应用程序呢？于是，作为 ATL 的一部分，WTL 被开发出来了。它将 ATL 进行了扩展，以使得它可以支持各种类型的与 UI 相关的组件或者应用程序。然而，它并没有随着 ATL 一同集成在 Visual Studio 中被发布，于是我就决定将它作为一个单独的 ATL 扩展库发布出去。

我认为 WTL 将一直是那些在 Windows 下开发应用程序以及组件的开发者的一个很好的选择。我并不认为在以后，我们会对 WTL 有一个大的改动（或者增添），因为 WTL 的一个设计宗旨就是“遵循 Win32 UI 的 API 及其设计”。现在如此，将来还是会如此下去。

【C++ View】：I first heard about WTL in July, 2000. At that time, I thought: "No official support, no documentation, no commercial hype, it will die in 6 months." Now, fifteen months passed, it spreads wider and be more vigorous. Lots of C++ programmers, esp. the ones we regard them as "gurus and masters" involved in WTL. I know it's surely because WTL is a wonderful library, but it must be more than a wonderful library to gain such attentions without official force. What do you think WTL's relative success? What's the reason?

【C++ View】：我第一次接触 WTL 是在 2000 年 7 月。在那时，我就想：“没有官方的支持，没有文档，也没有商业吹捧，它最多只能存活 6 个月。”但现在 15 个月过去了，它反而流传得更为广泛，更加的生机勃勃。许多 C++ 程序员，尤其是一些我们所认知的“专家”以及“大师”，都在使用 WTL。我当然知道这主要是因为 WTL 的出色，但我想，能够在没有官方的力量牵涉的情况下吸引如此多的注意，WTL 一定还有更出色的东西，请问您是如何看待 WTL 的成功呢？它成功的原因又是什么？

【Nenad】：I think that the main reason of WTL's success is that it did fit the need of developers at the right time. More and more developers started using ATL, and it was natural for them to start using WTL when they needed more UI support.

It seems that WTL was perceived as a more open project than others, judging by the support provided by other parties in the development community. Many people did a wonderful job of creating samples, documentation and support for WTL. The support from the programming community is very important part of the acceptance and success of WTL.

【Nenad】：我认为 WTL 成功的最主要原因就是，它确实而且及时地满足了开发者的需求。越来越多的开发人员开始使用 ATL，当他们需要更多的 UI 支持时，他们很自然的就会开始使用

WTL。

从其他的开发团队所提供支持来看，WTL 看起来似乎要比其他的项目更加开放。许许多多人为 WTL 做了大量工作，如：创建示例代码，撰写文档等。WTL 之所以能够被广为接受并获得如此大的成功，来自于这些开发团队的支持绝对是一个重要的因素。

【C++ View】 : What do you think about MFC? Do you like it? If you don't, why? And the most confusing thing is Managed C++, is it C++? Do the leaders of MC++ really think some C++ users will go to learn it? Do you believe?

【C++ View】 : 请问您对于 MFC 是怎么看的？您喜欢它吗？如果不，为什么呢？还有，最让人迷惑不解的就是 Managed C++ 了，它是不是 C++ 呢？MC++ 的提倡者是不是真的认为会有一些 C++ 的用户转而去学习它呢？您的看法又是如何呢？

【Nenad】 : I think that MFC is a great framework library. Don't forget that MFC was designed at the time that C++ compiler was rather limited, and the main platform was 16-bit Windows. Unfortunately, because MFC was designed as a framework, it was really hard to evolve it to use better C++ support in newer compilers, and to add support for new features added to Windows in the meantime. What I don't like about MFC is the DLL approach, which causes many compatibility problems, and framework design, which dictates too many things about app design.

Managed C++ is an extension to C++ which allows C++ programs to use managed code. It is very important to understand that you can compile your existing C++ code using MC++ without any changes. MC++ allows developers to use both familiar non-managed C++ and managed code in the same module. That provides an excellent way to extend existing code to interact with managed code, as well to create new projects that can use both managed and traditional C++.

【Nenad】 : 我认为 MFC 是一个了不起的框架库。请不要忘了，在 MFC 被设计出来初期，那时的 C++ 编译器还具有很多的限制，并且那时主要的平台还只是 16 位的 Windows。不幸的是，由于 MFC 被设计成为一个框架，使得我们很难利用新编译器中那些更好的 C++ 特性来改进它，也很难将 Windows 中的很多新特性添加到 MFC 中。我不喜欢 MFC 的地方是它高度依赖 DLL 的特性——因为它将导致许多兼容性方面的问题；还有就是 MFC 的整个框架设计——它在应用程序的设计中限定了太多东西。

Managed C++ 是 C++ 的一个扩展，它允许 C++ 程序得以使用受管（managed）代码。我们需要了解的一个很重要的特性就是，我们可以使用 MC++ 来编译已有的 C++ 代码而无需对它们进行任何改动。MC++ 允许开发者同时使用他们所熟悉的非受管代码以及受管代码来开发同一个模块。这就提供了一个非常好的途径，使已有的代码与新的受管代码相互作用，并也可使得我们创建一个项目，同时使用受管的和传统的 C++ 代码。

【C++ View】: In the past 15 years or more, C and C++ is the base of almost all Microsoft's technologies(OS, COM, etc.). We C++ user paid a lot of hard work to catch them, because we felt what we paid was worthy(?). Now, it seems the climate changed. .NET is coming, the world is going to be full of CLRs and/or JVMs. There has been a decampment from C++. So what do you think about the future of C++ (not MC++) in Microsoft technologies? Will it go away? Will it become a marginal language?

【C++ View】: 在过去的 15 年中 (甚至更长的一段时间内), C 以及 C++ 构成了几乎所有 Microsoft 技术的基础 (如: OS, COM 等)。我们这些 C++ 用户花费了大量的时间来熟悉并掌握它们 (C 以及 C++)，因为我们相信我们所付出的一定会有回报 (?)。但现在的风向好像有了很大的改变。.NET 出现了，世界似乎就要充斥 CLR (Common Language Runtime, 公共语言运行库) 以及 / 或 JVM (Java Virtual Machine, Java 虚拟机)。现在 C++ 已经出现了退潮的迹象。那么，请问您对于 C++ (不是 MC++) 在 Microsoft 技术中的前景如何看待？它是否会由此消亡？还是就此沦落为一门边缘语言？

【Nenad】: Well, the world is changing too. The new type of development for Web services and connected applications is on the horizon. I think that new languages, like Java, C#, and VB.NET, were developed to address two main issues - to simplify software development and to provide better support for Internet development. Simplifying software development allows more developers to write good applications and cuts down on time needed to finish a project. Supporting Internet development is obviously very important in this time when Internet is used more and more in every part of everyday life.

I think that C++ will continue to be an important language, especially for ISV's and for system development. On the other hand, I believe that .NET will be very important platform soon. .NET has the potential to be the main programming platform for the future, but it is reasonable to expect that the transition will take some time.

【Nenad】: 是的，世界也已经发生了变化。对于网络服务以及连接这样的新型应用程序的开发已经浮上了水面。我认为那些新的编程语言 (如 Java, C#, 以及 VB.NET) 都是针对以下两个主要的问题而开发出来的——简化软件的开发过程以及对于 Internet 应用程序开发提供更好的的支持。简化软件的开发过程使得更多的开发者可以写出更多更好的应用程序并减少完成开发项目所需要的时间。而支持 Internet 的开发，对于这个 Internet 越来越深入到我们的日常生活中的时代来说，毫无疑问是一件非常重要的事情。

我认为 C++ 会继续作为一门重要的编程语言发挥作用，尤其是对那些独立软件开发商和那些系统级开发来说更是如此。从另一方面来说，我相信 .NET 将会在不久以后成为另外一个非常重要的开发平台。对于未来来说，.NET 拥有成为主流编程平台的潜力，但我们必须认识到，这样的过渡阶段肯定要持续一段时间。

【C++ View】: There are lots of beginners in our readers, after they learn (standard) C++, they want to seek a path to master enough Microsoft technologies to be practical and proficient programmers. Could you recommend such a path? Should they learn Win32 API programming? Is it worthy of studying MFC? Is WTL/ATL/STL a reliable solution? Or goto C# directly? Many many people will thank you if you give them frank advice.

【C++ View】: 我们的读者中有很多是初学者，在他们学习完（标准）C++后，他们希望能够找到一条道路，掌握到足够多的 Microsoft 的技术使自己成为经验丰富的、熟练的程序员。您能不能给我们指出这样的一条道路来呢？我们是不是应该学习 Win32 API 编程？学习 MFC 是否是值得的？WTL/ATL/STL 算得上是一个可靠的解决方案吗？又或是我们应该直接学习 C#？如果您能够给我们一些建议，相信会有很多的人为此而感激您的。

【Nenad】: I think that depends on their plans and ambitions. The more of those things you do, the better you are off in the long run. But, you also have to balance that with the practical issues. So, I think that people who see their future in the Internet development can go directly to C# or VB.NET, and study .NET platform. Those who would like to have more knowledge of the Windows platform and services it provides should certainly learn more about Win32 API and libraries that support Windows programming.

【Nenad】: 我认为这主要取决于他们的计划以及雄心。你所做的越多，在长时间竞争中你就越占据优势。不过你也要注意保持与实际问题的平衡。我建议那些决心以后只做 Internet 相关开发的人可以直接去学习 C# 或者 VB.NET，同时学习 .NET 平台。而那些更多地了解 Windows 平台以及它所提供的服务方面知识的人，当然就必须需要更多地了解有关 Win32 API 以及那些支持 Windows 编程的库相关的知识。

【C++ View】: Soon after I began to learn WTL, a warm-hearted man posted me an email. He wrote: "You won't be a good WTL programmer if not a good ATL programmer, you won't be a good ATL programmer if not a COM programmer. And once you decide to learn COM, you are beginning your travel to hell." Is COM so difficult to learn? How to study WTL? We must learn API, COM, ATL and WTL in sequence, do we? And what about COM, will it remain to be the core technology of Microsoft, or just be substituted by .NET and dismissed?

【C++ View】: 在我刚开始学习 WTL 后不久，有一位热心人给我发了份邮件。他写道：“如果你不是一个好的 ATL 程序员的话，你就不可能成为一个好的 WTL 程序员；如果你不会 COM 编程的话，你就不可能成为一个好的 ATL 程序员；但一旦你决定开始学习 COM，你就迈出了踏向地狱的第一步。”COM 是不是真的那么难学？我们应该如何地来学习 WTL 呢？我们是不是应该按照这样的顺序学下来呢，API->COM->ATL->WTL？还有，COM 将会变得如何？他是不是还能够保持 Microsoft 的核心技术这一头衔，抑或是被 .NET 给替换掉然后就此消失？

【Nenad】: I don't think it is necessary to master COM to use and understand WTL. Win32 UI knowledge is more important than COM to understand WTL. But it is true that knowledge of ATL is

required, and ATL mainly supports COM. So, COM knowledge is desirable, but not required.

I don't think that COM is hell, but it sure does require a lot to learn to be an expert. Keep in mind that many people don't have to be COM experts to use COM, or to use WTL. Just understanding basic principles of COM is enough to use it, and then people can learn more when needed.

【Nenad】: 我不认为使用和理解 WTL 就一定要掌握 COM。相比于 COM 来说，Win32 UI 的知识对于理解 WTL 显得更为重要。但毫无疑问的是，ATL 的相关知识是必不可少的。由于 ATL 主要任务就是支持 COM，所以，有 COM 的知识只是会更好一些而已，但它们并不是必需的。

我也不认为 COM 是一个噩梦，但毫无疑问的是，想要成为一个 COM 专家，要学的东西实在是太多了。但请记住一件事情，很多使用 COM 或者 WTL 的人并不都是 COM 方面的专家。要想使用它们，人们所需了解的只是一些 COM 的基本原理就够了，其他的相关的知识则可以在需要时再去学习。

【C++ View】: What do you think about Generic Programming? Is it a whole different paradigm from OOP, or just OOP's supplemental facility? Can we combine GP and OOP? In the hard work of design and implement WTL, you must had got an insight about the relationship between OOP and GP, what's it?

【C++ View】: 请问您对于泛型程序设计是如何看待的？它到底是 OOP 的一个补充呢，还是完全不同于 OOP 的另外一个程序设计范型呢？我们是否可以将 GP 以及 OOP 一同联合使用？我想，在设计和实作出 WTL 的艰苦过程中，对于 OOP 以及 GP 之间的关系，您一定有了自己的看法，您能不能给我们说一下呢？

【Nenad】:Generic Programming and OOP are very different, mostly because Generic Programming doesn't explicitly express relationships between design elements. They can, however, be used together very efficiently.

WTL uses a combination of Generic Programming and OOP design. Templates are mostly used to implement traditional OOP classes. I'd like to point out that WTL doesn't use any "pure" design, and it doesn't strictly follow any design guidelines or styles. I do think, though, that WTL does use one of the main strengths of the C++ language - it uses appropriate paradigm that is the most suitable for a particular problem.

【Nenad】:GP 和 OOP 非常不同 ,这主要是由于 GP 从不显式地表达出设计元素之间的关联来。然而，它们也可以被非常高效地组合运用。

WTL 中使用了一种 GP 连同 OOP 的设计。我在其中大量使用了模板来实作出传统的 OOP 中

的类。我很乐意指出的是，WTL 中并没有使用一种“纯”设计，它也没有遵循任何的设计指导方针或者设计规格。可是，我还是认为 WTL 使用到了 C++ 语言中的最主要的精髓处——对于一个特定的问题使用一种最适合它的适当典范。

【C++ View】: Recently, the famous C++ pioneer Stanley Lippman joined Microsoft and became a member of Visual C++.NET group. How do you think about this? What message do you think your company like to pass to public? Does this mean Microsoft want to make VC.NET a full-standardized C++ compiler and hold C++ as your core system language?

【C++ View】: 最近，著名的 C++ 元老级大师 Stanley Lippman 加入了 Microsoft 并成为其 VC.NET 开发小组中的一员。请问您对于此事是如何看待的？您认为 Microsoft 试图向公众传播一种什么样的信息呢？这是否也意味着 Microsoft 希望 VC.NET 成为一个完全标准化的 C++ 编译器，并继续保持 C++ 的核心系统语言地位呢？

【Nenad】: I think that shows that Microsoft is committed to advance the C++ compiler and language, and ready to get the best people to help. I am sure that VC++.NET will continue to be powerful tool for developing applications, and that it will also include additional capabilities for the .NET development. Compliance with the C++ Standard is an ongoing work, and we will see further improvements there, too.

【Nenad】: 我认为这显示了 Microsoft 对于促进 C++ 编译器以及语言继续发展的决心，并为此找到了最佳人选来获取帮助。我确信 VC.NET 将会继续是开发应用程序的强有力工具，并且它同时还将包含有 .NET 开发能力。目前我们正在进行兼容 C++ 标准方面的工作，不久我们就会看到成效。

【C++ View】: I'm learning WTL and ATL, since you are the author of WTL and a member of ATL group, can you give me some advice?

【C++ View】: 我现在正在学习 WTL 以及 ATL，既然您是 WTL 的作者，同时又是 ATL 开发小组中的一员，您能不能给我一些建议呢？

【Nenad】: There are several areas of programming that are very important for WTL and ATL: knowledge of the C++ language in general, understanding of templates, COM for ATL, and Windows UI programming for WTL. Solid knowledge in these areas is very beneficial for WTL and ATL developers, and it also helps to understand the source code for both libraries.

I would also like to encourage everybody to write programs. That is the best way to learn how to use any library, or a programming language or operating system. Writing programs often brings problems that must be solved that are not addressed in books. Reading about something is very

useful to start learning, writing programs is the best next step.

【Nenad】：对于 WTL 和 ATL 来说，有好几个编程方面的领域是十分重要的：大体上的 C++ 语言知识，了解模板，COM（对 ATL 而言），以及 Windows UI 编程（对 WTL 而言）。在这些领域有着坚实的基础对于 WTL 以及 ATL 开发人员来说有着很大的好处，同时对于理解这两个的源代码也能够起到帮助作用。

我同样也很乐意去鼓励大家多写程序。这也是学习如何使用一个程序库，或者一门编程语言，或者一个操作系统的一个最好的方法。在写程序的过程中经常会出现一些书本上没有提及但又必须被解决的问题。在开始学习时读一些东西是很有用的，而写程序则是向纵深发展的最佳方式。

【C++ View】：They say we are in the gate of Post-PC times, it will be a embedded world, and there will be embedded smart device everywhere, and the embedded industry will build a far large market compare to PC's. Do you believe it? Do you think WTL and other C++ template libraries are available and appropriate for embedded development? Are there available for Internet development?

【C++ View】：有人说，我们现在已经处于后 PC 时代的门口，未来将会是一个嵌入系统的世界，嵌入式的智能设备将会无所不在，并且对比 PC 来说，嵌入系统的产业将会是一个更大的市场。您是否相信这些呢？您是否认为 WTL 以及其他的一些 C++ 模板库对于嵌入式开发也适用呢？它们是否适合 Internet 开发？

【Nenad】：Yes, I think that large number of various devices that we use everyday will become small, specialized computers. That doesn't mean that the number and importance of PCs will go down, just that there are many other devices that will be enhanced to be programmable and connected. Those new devices will provide a great opportunity for software developers, because they will all have software and somebody has to write it.

Many of the C++ libraries are quite appropriate for embedded development, and WTL would also be in cases where Windows based user interface is important (for example, Pocket PC platform). Great flexibility and small footprint are always very important features for embedded development, so template libraries are in the very good position there.

【Nenad】：是的，我认为我们现今所使用的各种设备中的大部分在以后都将会演变成一些小的，具有专门用途的计算机。但这并不意味着 PC 的数目以及重要性将会由此降低，只不过表明还有着许多其他的设备需要被加强以使得我们可以对其进行编程并且连接。由于必须需要有软件的支持，而软件又需要有人来写，这些新的设备将会给软件开发人员带来极大的机遇。

有许多的 C++ 函数库都可用于嵌入系统的开发，WTL 也将会在那些 Windows 用户界面较为重要的开发中（例如，在 Pocket PC 平台上面开发）占有一席之地。对于嵌入式开发来说，良好

的弹性，微小的内存耗用永远都会是很重要的特性，而模板库在这方面占据了一个非常好的地位。

【C++ View】：In the past 7 years or more, COM is Microsoft's core technology. And we now can see that in the next decade, the core role may be .NET. My question is, what's wrong with COM? Where will COM be? Will it disappear? Will it be substituted by something else? What's the relationship between COM and .NET? Is .NET based on COM? Is it worthy of learning COM now?

【C++ View】：在过去的 7 年（甚至更长的一段时间）内，COM 都是 Microsoft 中的核心技术。现在我们可以预见到，在下一个十年间，这个核心将会变为.NET。我的问题是，COM 有什么过错？COM 将何去何从？它是否会逐渐消失呢，还是会被其他的一些技术给替代？COM 和.NET 之间的关系是什么样的情况？.NET 是否是基于 COM 之上呢？现在学习 COM 是不是还值得？

【Nenad】：Maybe you shouldn't ask what is wrong with COM, but just think of .NET as the evolution of COM. .NET extends what was started with COM- creating reusable binary components - and brings additional important features: rich metadata, great run-time, built-in security, versioning, etc. All of these new features are important for development today, and it is really great that .NET has extensive support for them. Interoperability between .NET and COM is also provided, so that the previously developed COM components can still be used in the .NET environment.

I still think that it is a good idea to learn COM - it is a great first step even for people who want to learn .NET, and it also provides better understanding of the design and implementation of .NET itself.

【Nenad】：或许你不应该问 COM 有什么过错，而是应该把.NET 看成 COM 的进化。.NET 扩展了 COM 最初的目的——创建可重用的二进制程序组件——并向其中添加了很多重要的特性：丰富的元数据，了不起的运行库，内建的安全机制，版本机制等。对于现今的软件开发来说，所有的这些新的特性都非常重要，所以.NET 能够广泛地支持它们，是一件很伟大的产品。Microsoft 同时也提供了在.NET 和 COM 之间进行互操作的能力，这使得以前所开发出来的 COM 组件在.NET 环境中同样也能得到使用。

我仍然认为学习 COM 是一个很好的主意——它甚至对于那些希望学习.NET 的人们来说也是一个很好的开端，学习 COM 同时也有助于我们更深入地理解 .NET 本身的设计和实现。

【C++ View】：I know you must be a C++ fan. Now the language is facing lots of challenges. To counterattack, Dr. Stroustrup suggest to develop many useful libraries, and teach the programmers to

use C++ as a high level language. Now there are several wonderful modern C++ libraries. Aside of ATL, WTL and STL, there are still Boost library, MTL, ACE/TAO, DTL, etc. It seems the C++ community is preparing a movement. Do you think the movement will success? Why? What are your colleagues'(in Microsoft VC.NET group) attitude towards such a movement?

【C++ View】：我猜想您肯定是一个 C++ 爱好者。现在这门语言面对着许多的挑战。作为反击措施，Stroustrup 博士提议开发许多有用的库，并引导 C++ 程序员把 C++ 当作一门高级语言来使用。现在我们已经可以得到好几个极好的现代的 C++ 库，除去 ATL、WTL 以及 STL 之外，还有 Boost 库、MTL、ACE/TAO、DTL 等。一切都显示着 C++ 社区正在酝酿着一次变革。请问您觉得这场变革能否成功？为什么？您的那些 Microsoft 中的 VC.NET 开发小组中的同事对于此态度是怎样的？

【Nenad】：C++ is a great language and its importance remains high, even with the new challenges. Libraries are an excellent addition to the language itself, as they provide very useful reusable code for developers. The existence of many great C++ libraries shows the size and strength of the C++ community. I think that is already a success, and that it will continue. You can be sure that the VC++.NET group is aware of the existing libraries, and I expect them to continue to enhance the support for them.

【Nenad】：C++ 是一门伟大的语言，即便遇到了新的挑战，它仍然将是非常重要的。程序库对于语言本身是极好的补充，它们为开发者提供了一些十分有用的可重用代码。存在如此众多的、了不起 C++ 程序库，这件事情本身就表明了 C++ 社区的庞大和强大。我认为这场变革已经成功了，并且会一直成功下去。你们可以放心，VC.NET 开发组是不会对这些已有的程序库熟视无睹的，我预期他们会不断地加强对于这些库的支持。

【C++ View】：The last question. Since lots of people don 't acquaint themselves with WTL, now we have a chance for you, the father of WTL, to introduce WTL in a short speech here. What would you like to speak?

【C++ View】：最后一个问題。既然许多人并不了解 WTL，作为 WTL 之父，您现在有机会在这里做一个演讲，请简短地介绍一下 WTL。

【Nenad】：WTL is a template based library for user interface development. It extends ATL to provide classes for implementing user interface for applications, components, and controls. It provides classes for various user interface elements: top-level windows, MDI, standard and common controls, common dialogs, property sheets and pages, GDI objects, UI updating, scrollable windows, splitter windows, command bars, etc.

WTL is implemented using the same template architecture as ATL, so it is a natural fit for ATL developers. It also doesn't alter or hide Windows specific constructs, thus allowing Windows programmers to use WTL without surprises. The important design goal of WTL was to avoid

inter-dependencies - classes themselves do not reference other WTL classes. That means that your program will contain just the code that you actually use, and nothing else. Coupled with the use of templates, this allows creation of very small programs without run-time dependencies.

WTL delivers the object oriented way to program for the Windows user interface, while keeping the code size small. It also provides a great foundation that developers can extend with their own classes.

And finally - WTL was written with a hope that developers will enjoy using it. I hope you will use it and enjoy it, too.

WTL 是一个基于模板的、专为开发用户界面的程序库。它扩展了 ATL，并提供了一些类用来实现应用程序的用户界面、组件和控件。它提供了各种类来支持各种各样的用户界面元素：顶级窗口、MDI、标准控件和通用控件、通用的对话框、属性表以及属性页、GDI 对象、UI 更新、可滚动的窗口、分割窗口、命令条等等……

WTL 的实现使用了和 ATL 一样的模板架构，所以对于 ATL 开发者显得很自然。同时它并没有改变或者是隐藏那些 Windows 相关结构，那些 Windows 程序员在使用 WTL 时也不会感到很吃惊。WTL 的一个主要设计原则就是避免在没有引用到其他 WTL 类时，出现不必要的内部依赖。这意味着我们的程序将只包含有我们实际上所使用的代码，除此之外再无其他的东西。加上了模板的使用后，这样做得到的结果就是一些非常小的，不依赖于运行库的程序。

WTL 专注于用面向对象的方法来编写 Windows 的用户界面程序，同时保持代码的尺寸很小。同时，它也为开发者提供了一个很好的基础，可以写新的类来扩展 WTL。

最后，我在编写 WTL 时就希望开发者能够喜欢在开发中使用它。我同样也希望您能够使用它并喜欢上它。

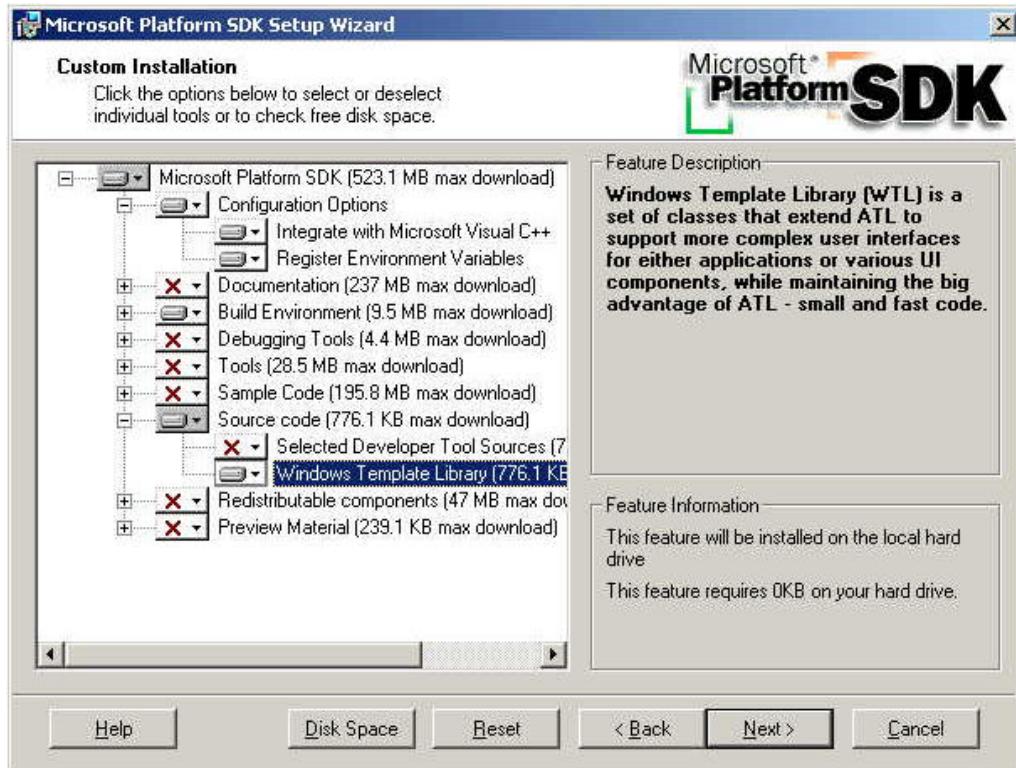
WTL 简介

作者 : vcmfc

在 ATL 出现的时候 , 一些部分 COM 的编程人员开始觉得开发 COM 运用是一种快乐 , 因为使用它很方便地开发小规模的 COM 组件 , 但好景不长 , 现实的 COM 组件是包罗相当广泛的 , 特别当准备包装窗口控件时 , 发现 ATL 提供了的支持相当缺乏。因此 Microsoft 推出了半成品与没有技术支持的 WTL , 这也是 WTL 诞生的原因。

很多初次接触 WTL 的朋友都问 “ WTL 这三个字母代表什么呢 ? ” : WTL 全称为 Windows Template Library, 构架于 ATL 之上 , 采用 C++ 模板技术来包装大部窗口控制 , 并给出一个与 MFC 相似的应用框架。

他们紧跟着问 “ 那我如何得到它呢 ? ” : 由于 WTL 是 Microsoft 推出的 , 在 Microsoft 的 Platform SDK 中就有 , 以下是部分画面 :



或者能过以下链接下载 : <http://msdn.microsoft.com/msdn-files/027/001/586/wtl31.exe>

跟着问题又来了 , “ 我该如何使用它们呢 ? ” : 在你安装完了 WTL SDK 之后 , 在安装目录中有一个 AtlApp60.Awx 的向导文件 , 将它拷贝到你安装 Visual C++ 的目录 : Microsoft Visual Studio\common\mesdev98\bin\ide\ 目录下 (实在不行使用 Windows 的搜索文件查找.awx) , 这是在 VC 的应用程序向导里就有跟 MFC 相似的 WTL 应用程序向导。

如果你是 MFC 的使用者 , 你可能会再问 “ WTL 与 MFC 在包装窗口控制有哪些不同呢 ? ” : 我只能用以下表格回答你 :

Feature	MFC	WTL
Stand-alone library	Yes	No (built on ATL)
AppWizard support	Yes	Yes
ClassWizard support	Yes	No
Officially supported by Microsoft	Yes	No (Supported by volunteers inside MS)
Support for OLE Documents	Yes	No
Support for Views	Yes	Yes
Support for Documents	Yes	No
Basic Win32 & Common Control Wrappers	Yes	Yes
Advanced Common Control Wrappers (Flat scrollbar, IP Address, Pager Control, etc.)	No	Yes
Command Bar support (including bitmapped context menus)	No (MFC does provide dialog bars)	Yes
CString	Yes	Yes
GDI wrappers	Yes	Yes
Helper classes (CRect, Cpoint, etc.)	Yes	Yes
Property Sheets/Wizards	Yes	Yes
SDI, MDI support	Yes	Yes
Multi-SDI support	No	Yes
MRU Support	Yes	Yes
Docking Windows/Bars	Yes	No

Splitters	Yes	Yes
DDX	Yes	Yes (not as extensive as MFC)
Printing/Print Preview	Yes	Yes
Scrollable Views	Yes	Yes
Custom Draw/Owner Draw Wrapper	No	Yes
Message/Command Routing	Yes	Yes
Common Dialogs	Yes	Yes
HTML Views	Yes	Yes
Single Instance Applications	No	No
UI Updating	Yes	Yes
Template-based	No	Yes
Size of a statically linked do-nothing SDI application with toolbar, status bar, and menu	228KB +MSVCRT.DLL (288KB)	24k (with /OPT:NOWIN98) (+ MSVCRT.DLL if you use CString)
Size of a dynamically linked do-nothing SDI application with toolbar, status bar, and menu	24KB+MFC42.DLL (972KB) +MSVCRT.DLL (288KB)	N/A
Runtime Dependencies	CRT (+ MFC42.DLL, if dynamically linked)	None (CRT if you use CString)

最后再说两句。由于 WTL 不是 Microsoft 的正式产品，因此得不到 Microsoft 的技术支持，虽然有不少民间技术团体的支持，但这还不够；关于 WTL 的技术文章相当的少，而且 WTL 使用 C++ 的 Template 技术，这是一种相对较新的技术，无法与 MFC 混合使用，使用它需要重新学习它，以致于相当少的人使用它。

相关参考：

- 1.田光照《WTL 简介》http://www.vchelp.net/article/submit/wtl_intro.htm
- 2.Ben Burnett “ Introduction to WTL - Part 1 ” <http://www.codeproject.com/wtl/index.asp#Beginners>
- 3.Dharma Shukla, Chris Sells, and Nenad Stefanovic “ Makes UI Programming a joy ”

鸟 鸣 涧

虫虫评

(问题请见 C++ View 第 5 期)

我们先从第 3 个问题开始，这个问题大家都做对了。一种比较简单的方式是

```
class MyClass
{
public:
    bool operator <(const MyClass&) {return true;}
    // 这是开玩笑的写法，只能通过编译，没有实际价值
};
```

第 1 和第 2 是相辅相成的，一个错了另一个也对不了。先看看第 1 题：

```
template <class T>
T Max(T* first, T* last)
{
    if (first == last)
        return *first;
    T* r = first;
    while (++first != last)
        if (*r < *first)
            r = first;
    return *r;
}
```

T 显然必须具备可比性 (Comparable)，更准确地说，是要重载运算符 < (LessThanComparable)，这个大家都能看出来。另一个有点隐蔽约束条件就漏网了，那就是可拷贝构造 (CopyConstructive)，上面程序中着重指出的就是需要约束条件的地方。

看看下面这个类：

```
class MyClass_error
{
public:
    bool operator <(const MyClass_error&) {return true;}
    MyClass_error() {}
private:
    MyClass_error(const MyClass_error&) {}
};
```

由于拷贝构造函数为私有 (private) , 下面这段调用就无法通过编译了。

```
int main()
{
    MyClass_error a[2];
    Max(a, a + 2);
}
```

因此第 1 题的答案是两个 : LessThanComparable 和 CopyConstructive , 很遗憾 , 只有很少的朋友看到了后者。好在 C++ 编译器会为我们的类自动添上默认的拷贝构造函数 , 第 3 题才容易做对。

那么第 2 题也迎刃而解了。所有的原生类型都可以比大小 , 但是只有 void 不可拷贝构造。

第 4 题收到了很多不同的答案 , 都是对的。下面把几种典型答案列在下面 , 大家一同欣赏。

一种方式自然是在 Max 中加入动态 RTTI 判断 , 有朋友采用 MFC 给出一种方法 :

```
class CMyClass: public CObject
{
public:
    DECLARE_DYNAMIC(CMyClass)
    bool operator < (const CMyClass&) {return true;}
    CMyClass() {}
    CMyClass(const CMyClass&) {}
};

IMPLEMENT_DYNAMIC(CMyClass, CObject)

template <class T>
T Max(T* first, T* last)
{
    if (DYNAMIC_DOWNCASE(CMyClass, first))
        ASSERT(FALSE);
    // 以下与上面的程序相同
}
```

用 C++ 本身的 dynamic_cast 也可以给出完全类似的解决方法。

```
template <class T>
T Max(T* first, T* last)
{
```

```
assert(dynamic_cast<MyClass*>(first) == 0);
//...
}
```

这里有点问题：原生类型（如 int）也都不能通过编译，总不至于再给它们写包装类吧？因此这个方法不太好，并且也不是很符合题意：题目要求仅仅 MyClass 不行，而这里成了 MyClass 及其子类通通不行。好在用 typeid 可以补救。

```
template <class T>
T Max(T* first, T* last)
{
    assert(typeid(*first) != typeid(MyClass));
    //...
}
```

比较好的方式是利用函数重载或模板特化，不用改动 Max 函数本身，只需要再加上：

```
MyClass Max(MyClass*, MyClass*)
// 或者是 template <> MyClass Max(MyClass*, MyClass*)
{
    assert(false);
}
```

这是一些朋友的方式，漂亮，可惜画蛇添足。其实我们可以只写出函数的声明，不写其实现：

```
MyClass Max(MyClass*, MyClass*);
```

如果调用了 MyClass 版本的 Max，连接器就会报错，因为这个函数没有具体的实现代码！

接下来是稍微深入的背景介绍，如果您看了一点就觉得很糊涂，就不用看下去，忘掉它！

大家知道，面向对象（object oriented）中三个重要的概念是：
object（对象）– class（类）– inheritance（继承），

以 class 为核心，class 的实体是 object，class 之间的基本二元关系是 inheritance。

其实泛型编程（generic programming）也有类似的三套马车，叫作：
model（模型）– concept（概念）– refinement（细化）

以 concept 为核心(比如题目中的 T 可代表的类型),concept 的实体是 model(比如苹果、MM),concept 由一系列 requirements 组成,若 concept B 包含了 concept A 的所有 requirement(s),那 B 就是 A 的 refinement。比如 LessThanComparable 和 CopyConstructible 就是两个 requirements,它们共同构成了 T 所代表的 concept,其中 T 既可以是类,也可以是原生类型,因为泛型编程中并没有“类”这个概念。

从集合论的角度看,inheritance 和 refinement 这两种关系都是偏序关系,都具有

- ✓ 自反性(每个 concept 当然都是自己的 refinement);
- ✓ 反对称性(若 A、B 两个 concept 互为 refinement,那它们必相等);
- ✓ 传递性(若 B 是 A 的 refinement,C 是 B 的 refinement,那 C 必是 A 的 refinement。
类似地,若类 B 是类 A 的子类,类 C 是 B 的子类,那 C 必为 A 的子类)。

我们可以把某个 A (class/concept) 看作是其所有的实体 (object/model) 所组成的集合,那么上述的 inheritance/refinement 关系也可以看作是集合间的包含关系,其基本的定义是:

$$A \subseteq B \Leftrightarrow (\forall x)(x \in A \rightarrow x \in B).$$

比如,若 A、B 两个 concept,B 是 A 的 refinement,b 是 B 的 model,那 b 必是 A 的 model。

这就是泛型编程的基本理论。更详细的介绍,请参考 Matthew Austern 所著 *Generic Programming the STL* 一书。运用泛型的关键,就在于抽象出所谓的 concept,而很好的经验和例子并不多,STL 和 boost 库 (<http://www.boost.org>) 是不可多得的材料。

concept 的检验 (check) 是一个很重要的问题。比如对 Max 应用前面那个拷贝构造函数为私有的 MyClass_error 类,编译时光标会停在 Max 函数的两个 return 语句处,说 MyClass_error 的拷贝构造函数不可访问。其实真正的错误是 MyClass_error 不是一个符合 Max 函数的参数所代表的 concept 的 model,而这出错信息往往让人摸不着头脑。

或者又比如

```
#include <list>
#include <algorithm>
using namespace std;

int main()
{
    list<int> a;
    //...
    sort(a.begin(), a.end());
}
```

编译出错时，光标往往停在一个莫名其妙的头文件里，错误信息是：“operator - 没有实现”等类似的东西，让人一头雾水。其实错误真正的原因在于，sort 要求参数的 concept 至少应该是 RandomAccessIterator，而 list::iterator 却是 BidirectionalIterator，不满足条件。

相信这是诸位在使用 STL 经常遇到的问题。这就要求我们的代码能对参数的 concept 进行检查，以方便应用。

C++大师们早就注意到这个问题。C++之父 Stroustrup 博士在其所著的 *Design and Evolution of C++* (简称 D&E) 一书中，明确地提出了 constraints for template parameters 的问题，并给出了一种非常通用的解决方案。我向 Stroustrup 博士询问有关学习模板技术的问题时，他首先推荐的便是这种技巧。原文请见 http://www.research.att.com/~bs/bs_faq2.html#constraints，征得 Stroustrup 博士同意，特在此做一介绍，同时也对 Stroustrup 博士深表谢意。

考虑如下代码：

```
template< class Container>
void draw_all(Container& c)
{
    for_each(c.begin(), c.end(), mem_fun(&Shape::draw));
}
```

如果传入的类型错了，编译错误会在 for_each() 中产生。比如容器的类型是 int，for_each() 中就会产生错误，因为 Shape::draw() 并不能接受 int 参数。

为了提前捕捉到错误，可以这么写：

```
template< class Container>
void draw_all(Container& c)
{
    Shape* p = c.front();
    // 仅当容器元素是 Shape* 时才成立
    for_each(c.begin(), c.end(), mem_fun(&Shape::draw));
}
```

如果传入类型不对，对于绝大部分编译器，初始化 p 时就会引发编译错误。这样的技巧很常见，我们可以将其推广、整理一下：

```
template< class Container>
void draw_all(Container& c)
{
    typedef typename Container::value_type T;
    Can_copy<T, Shape*>(); // 只接受元素类型为 Shape* 的容器
```

```

    for_each(c.begin(), c.end(), mem_fun(&Shape::draw));
}

```

Can_copy 可以这么定义：

```

template< class T1, class T2> struct Can_copy {
    static void constraints(T1 a, T2 b) { T2 c = a; b = a; }
    Can_copy() { void(*p)(T1,T2) = constraints; }
};

```

编译时，Can_copy 检验 T1 是否可以被赋给 T2。Can_copy< T,Shape*>检验 T 是否为 Shape*，或 Shape 派生类指针，或某个（由用户定义）可转换为 Shape* 的类型。注意模板定义很精当，“增之一分则嫌多，减之一分则太少”：

- ✧ 一行写出静态函数 constraints，参数类型就是需要检验的类型；
- ✧ 一行列出需要检验的约束条件（在 constraints 函数中）；
- ✧ 一行引发编译器检查（在构造函数中）

看看这个方法的好处

- ✧ 不需要声明或复制变量，因为写 constraint 的人不需要去设想一个类型是如何初始化，是否能被复制、销毁，等等（当然，除非就是要测试这些属性）；
- ✧ 编译器并不会产生多余的代码；
- ✧ 不需要宏；
- ✧ 编译器能对错误的约束条件给出一个令人满意的出错信息，包括单词“constraints”（提示阅读者）constraints 的名字，以及导致编译失败的错误（比如“cannot initialize Shape* by double*”）。

那为什么不把 Can_copy() 和一些类似的更爽的好东东放到语言中呢？D&E 中分析了在 C++ 中表达通用的 constraints 所需要解决的一些棘手的问题。从此以后，出现了许多方法，使得写出 constraints 更容易，而仍然能产生易懂的出错信息。

Stroustrup 博士相信，刚才在 Can_copy() 所用的方法是 Alex Stepanov 和 Jeremy Siek 发明的，不过 Can_copy() 并不应在现在加入标准，这需要更多的实践运用。C++ 社群使用着各种不同类型的 constraints，对于何种类型的 constraints 更为高效使用，大家还没有形成广泛的共识。

再看看下面：

```

template< class T, class B> struct Derived_from {
    static void constraints(T* p) { B* pb = p; }
    Derived_from() { void(*p)(T*) = constraints; }
};

```

```

template< class T1, class T2> struct Can_copy {
    static void constraints(T1 a, T2 b) { T2 c = a; b = a; }
    Can_copy() { void(*p)(T1,T2) = constraints; }
};

template< class T1, class T2 = T1> struct Can_compare {
    static void constraints(T1 a, T2 b) { a==b; a!=b; a< b; }
    Can_compare() { void(*p)(T1,T2) = constraints; }
};

template< class T1, class T2, class T3 = T1> struct Can_multiply {
    static void constraints(T1 a, T2 b, T3 c) { c = a*b; }
    Can_multiply() { void(*p)(T1,T2,T3) = constraints; }
};

struct B {};
struct D : B {};
struct DD : D {};
struct X {};

int main()
{
    Derived_from<D, B>();
    Derived_from<DD, B>();
    Derived_from<X, B>();
    Derived_from<int, B>();
    Derived_from<X, int>();

    Can_compare<int, float>();
    Can_compare<X, B>();
    Can_multiply<int, float>();
    Can_multiply<int, float, double>();
    Can_multiply<B, X>();

    Can_copy<D*, B*>();
    Can_copy<D, B*>();
    Can_copy<int, B*>();
}

```

对于经典的 constraints：元素必须是派生自 Mybase 的指针，可以这么写

```

template< class T> class Container : Derived_from<T, Mybase> {
    // ...

```

```
};
```

事实上，`Derived_from` 并没有检查派生性，而是转换性。给 `constraints` 找个好听的名字可真费脑子。

沿着这条路走下去，boost 库发展出一个专门解决这类问题的库 BCCL (Boost Concept Check Library)，可以参考参考。猜猜这个库的作者是谁？就是前面提到的 Jeremy Siek。

回到我们的题目，再谈谈最后一题。第 4 题的解答主要来自两种思路，一是模板特化，一是类似于刚才检验 concept 的方式。但是我们收到的解答都是利用了 RTTI 这种低效易出错的方式，其实可以用模板给出一种类似但很简洁、很高效的方法：

```
template <class T, class U> class EqualType {};
template <class T> class EqualType<T, T>

class MyClass;

template <class T>
T Max(T* first, T* last)
{
    EqualType<T, MyClass>();
    //...
}
```

如果使用了 `MyClass` 版本的 `Max`，编译就不会通过。这个方法有通用性和普适性吗？当然！在 Andrei Alexandrescu 所著的 *Modern C++ Design: Generic Programming and Design Patterns Applied* 一书中附有一个 Loki 库（可在 <http://www.moderncppdesign.com> 找到其源代码）。下面是利用 Loki 库的方法：

```
#include <Loki/TypeManip.h>
#include <Loki/static_check.h>
using namespace Loki;

//...

template <class T>
T Max(T* first, T* last)
{
    STATIC_CHECK(!Conversion<T, MyClass>::sameType, NotMyClass);
    //NotMyClass 没有意义，只是方便阅读
    //...
}
```

我们可以走得更远。比如题目改为 MyClass 及其子类均不可通过，那么代码是

```
template <class T>
T Max(T* first, T* last)
{
    STATIC_CHECK(SUPERSUBCLASS(MyClass, T), NotMyClassOrDerived);
    //NotMyClassOrDerived 没有意义，只是方便阅读
    //...
}
```

还可以把题目改为 MyClass 可以但子类不行，甚至更为复杂的条件，等等，都可以利用 Loki 库轻松解决。至于这其中用到的技术，似乎过于 tricky，与这几个问题关系也不大，有兴趣的朋友看看源代码，或者参考 *Modern C++ Design* 一书吧。

不过需要提一下 STATIC_CHECK，也就是所谓的静态断言 (assert)，一般的实现形式是：

```
template <int i> class assertT;
template <> class assertT<true> {};
```

使用方法是

```
assertT<表达式>();
```

如果表达式值为假，由于没有相应的类的实现，编译不能通过。

Loki 库中的宏 STATIC_CHECK 就是对类似的模板类的包装。Scott Meyers 在 *Modern C++ Design* 一书的序言中对这种方法颇为感慨，这种方法也越来越流行。比如在 C++ “准” 标准库 boost 中，也有一个类似的宏 BOOST_STATIC_ASSERT。

小结一下。我们简单介绍了泛型编程的一些基本概念，由此讨论了 concept 检验的一些问题。同时，我们讨论了另一类检验问题（第 4 题）。一种值得记住的技巧是利用模板特化：对于需要引发错误的那个特化版本，只写声明，不写实现。

使用 Qt 与 MySQL C API 开发 MySQL 查询器

作者 : SoftMusic

Email:mysql_gu@263.net

MySQL 是个好东东，关于它的优点互联网上有好多文章，这里不再多说。但令人不是十分满意的是它的查询界面不是很友好，不过没有关系，我们完全可以自己做一个。在这里向大家简单介绍一下如何使用 Qt 库和 MySQL 的 C API 作 MySQL 查询器。

首先介绍一个 MySQL 的 C API 的开发流程。

- 1、初始化一个 MYSQL 结构，使用函数 `mysql_init()`。
- 2、连接到一个 MySQL 服务器，使用函数 `mysql_real_connect()`。
- 3、向服务器发出查询请求，使用函数 `mysql_real_query()`。
- 4、取回并储存结果集合，使用函数 `mysql_store_result()`。
- 5、进行你需要的处理过程。
- 6、释放结果集合并关闭连接，分别使用函数 `mysql_free_result()` 和 `mysql_close()`。

这里不再对这些函数进行更详细的解释，下面会有其具体的例子，更多的内容可以在 MySQL 的官方网站上查询到 (<http://www.mysql.com>)。

关于 Qt 这里也不再进行更多的介绍，下面将直接进入本项目的分析与实现。

本项目要实现的功能分两个部分，一部分是方便用户编辑 SQL 脚本，与 SQL 脚本文件有关新建、打开、保存、关闭，以及对文本的剪切、复制、粘贴、查找、替换等功能，另一部分是与 MySQL 有关的连接、查询、显示、断开等功能。

前一部分的功能主要是与用户编辑 SQL 脚本有关，我们可以单独为此设计一个类，这个类要从 `QTextEdit` 中继承而来。并且此类需要实现我们需要的槽。这个类的定义如下：

```
class QGBKEdit : public QTextEdit
{
    Q_OBJECT
public:
    QGBKEdit(QWidget *parent=0, const char *name=0);
    virtual ~QGBKEdit();
public slots:
    void newFile();
    void openFile();
    void saveFile();
    void saveasFile();
    void closeFile();
    void replaceEdit();
    void findEdit();
```

```

private:
    QString m_fileName;           //用来保存当前的文件名。
    bool m_buserCancel;
};

```

下面详细解释一下类 QGBKEdit 的实现。构造函数的实现如下：

```

QGBKEdit::QGBKEdit(QWidget *parent, const char *name)
    : QTextEdit( parent, name )
{
    m_buserCancel=FALSE;
}

```

注意一定要有参数的初始化，否则此类不能作为其它部件的子部件使用。变量 m_buserCancel 的作用在下面会看到在这里不再说明。

newFile 槽的实现：

```

void QGBKEdit::newFile()
{
    closeFile();
}

```

新建文件实际是清空文件框和当前文件名，给用户重新开始新文件的操作而已，故这里直接调用关闭文件的函数。

openFile 槽的实现：

```

void QGBKEdit::openFile()
{
    if (isModified()){
        switch (QMessageBox::warning(this,"MyQuery",
            QString("The text is modify.Do you want save it?"),
            QMessageBox::Yes,
            QMessageBox::No,QMessageBox::Cancel|QMessageBox::Default)){
            case QMessageBox::Yes:
                saveFile();
                if (m_buserCancel){
                    m_buserCancel=FALSE;
                    return;
                }
                break;
    }
}

```

```
case QMessageBox::No:
    clear();
    //m_fileName="";
    break;
case QMessageBox::Cancel:
    return;
    break;
default:
    break;
}
}
if (length(>0)
    clear();
m_fileName= QFileDialog::getOpenFileName( QString::null, "SQL
script file(*.sql)",
                                         this, "Open File" );
QFile* file=new QFile(m_fileName);
QString s;
if ( file->open(IO_ReadOnly) ) {
    QTextStream t(file);
    s = t.read();
    insert(s,TRUE);
    file->close();
}
setModified(FALSE);
}
```

说明：如果用户对文本框中的文本进行了改动，首先要询问用户是否保存改变，这里使用 QMessageBox 的静态函数 warning() 提供用户三种选择：保存（Yes）不保存（No）取消（Cancel），并根据用户的选择作出相应的动作。如果用户没有选择取消，则进行下面真正的打开文件的操作。

首先取得要打开的文件名，这里使用类 QFileDialog。此类能实现所有与文件打开保存有关的对话框显示，并返回用户的选择，但它不真正实现文件打开，保存等操作，需要用户自己具体实现。这里我们使用其它静态函数 getOpenFileName() 函数显示打开对话框，下面的文件保存中使用 getSaveFileName() 函数显示保存对话框。

取回文件名后，建立一个文件对象 QFile* file=new QFile(m_fileName);

打开此文件，建立一个文本流对象 QTextStream t(file);

从流中读出所有数据对 QString 对象 s 中，文本框中写入此字符串。

如果是要保存文件则将文本框中的字符流向文本流对象(t << text)，在 saveFile() 函数中将看到相关的代码。

然后关闭文件。

最后把文本框的文本改变标志设为 FALSE。

有关文件的其它操作（保存、另存为、关闭）这里不再说明，相信读者看到源代码后能够理解。

下面介绍一下查找、替换操作的实现。这里需要两个对话框，为此我们将建立两个新类，MyFindDlg 和 MyReplaceDlg。其中 MyFindDlg 类从 QDialog 继承而来，因为 MyReplaceDlg 类要除了实现 MyFindDlg 类的全部功能外，还要添加替换操作的功能，所以 MyReplaceDlg 类从 MyFindDlg 类继承而来。如果用 VB 实现这个两个对话框，你会怎么办，对不起，你只能画两个相似的 Form——笔者有一段时间就是在做类似的重复工作——闲言少叙述，回到正题。文本框类与查找/替换类的相互联系是这样：文本框类将 this 指针提交给查找/替换类，然后显示查找/替换类，所有的查找/替换操作在查找/替换类内实现，用户完成操作后按 Cancel 按钮返回。这个在 QGBKEdit 类的 findEdit() 函数的代码就成了这个样子：

```
MyFindDlg dlg;
dlg.setEdit(this);
dlg.exec();
```

而 replaceEdit() 函数为：

```
MyReplaceDlg dlg;
dlg.setEdit(this);
dlg.exec();
```

由于 QTextEdit 本身已经实现查找字符串的功能，所以这里我们可以很容易的实现我们的要求。具体如下：

MyFindDlg 类中的查找的实现：

```
QString sTmp=m_fwedit->text();
//首先取要查找的字符串。
bool bFind=FALSE;
while
( !(bFind=m_edit->find(sTmp,m_chkcase->isChecked(),FALSE,TRUE,&m_iparaAt,&m_iCharAt))
 && m_iparaAt < m_edit->paragraphs()){
 m_iparaAt++;
}
//看上面的循环语句，如果在本段中找不到要找的字符串，则在下一段中查找，直到找到或全文搜索完毕。
if (!bFind){
 QMessageBox::information(this,"No Found",
 "Don't found the string what you want!");
 return FALSE;
```

```

//如果没有找到，则显示没有找到对话框，并返回 FALSE。
}
m_icharAt+=sTmp.length();
//如果找到则将查找起始位置后置，以便下一次查找。
MyReplaceDlg 类中的替换的实现：
if (!m_edit->hasSelectedText()){
    if (!findNextString())
        return FALSE;
//首先找到要替换的字符串
m_editRep->selectAll();
m_editRep->copy(); //选择、复制目标字符串到剪贴板。
m_edit->paste();
//然后粘贴一下即可，简单吧。

```

说了这么多了，还没有到正题，前面这些只不过作了一个最简单的文本编辑器而已。下面我们将要说本程序的“核心”了，如果可以这么说的话。

所有与 MySQL 有关的操作都有主窗口类 MyView 中实现，直接与 MySQL 服务器打交道的函数这里定义了三个槽连接 (connect())，执行 (Exec())，断开 (Disconnect())，显示返回结果的函数有两个：一个是显示成功执行结果的 showRes()，另一个是显示错误码执行结果的 showError()。

下面按执行顺序进行说明。

Connect() 函数：同 MySQL 数据库的连接主要是通过一个对话框类 (MyConnDlg) 实现，如果连接成功则返回 Accepted，否则返回 Rejected。

```

void MyConnDlg::OnOk()
{
    assert(m_connsql);
    QString user,host,pwd;
    host=m_editserver->text();
    user=m_edituser->text();
    pwd=m_editpwd->text();
    if (!mysql_real_connect(m_connsql,host,user,pwd,NULL,0,NULL,0)){
        QMessageBox::warning(this, "Connect
Error", (QString)mysql_error(m_connsql));
    }else{
        accept();
    }
}

```

首先取用户输入的要连接的主机地址、用户名、密码，然后执行 mysql_real_connect()

函数，如果连接成功则返回 0，对话框类退出并返回 accept()。否则发出连接错误警告，并显示服务器返回的错误信息。这个函数作为一个槽并连接到 OK 按钮的 clicked() 信号。Cancel 按钮的 clicked() 信号被连接到 OnCancel() 槽。OnCancel() 槽只有一句 reject();

执行 SQL 脚本函数：

```
bool MyView::Exec()
{
    if (!m_bConnected){
        QMessageBox::warning(this, "NO Connected", "Now you don't connect
to any MySQL server.please try to connect!");
        return false;
    }
    //首先取用户输入的脚本
    QString sql;
    if (m_edit->hasSelectedText())
        sql=m_edit->selectedText();
    else
        sql=m_edit->text();
    //如果为空则返回 false
    if (sql.isEmpty())
        return false;

    if (mysql_real_query(mysql,sql,(int)sql.length())==0){      //执行，
        //如果返回 0
        MYSQL_RES* result=mysql_store_result(mysql);           //取回结果集
        showRes(result);          //显示结果集
        mysql_free_result(result);      //释放结果集
    }else{
        showError();      //如果执行发生错误则显示错误
    }
    return true;
}
```

显示结果集函数：

```
void MyView::showRes(MYSQL_RES* res)
{
    initTable();
    //如果执行的为类似 Create 或 insert into 语句的动作查询并为结果集返回，则只
```

```
打印一个操作成功的提示即可。
if (res==NULL){
    m_table->setNumCols(1);
    m_table->setNumRows(1);
    m_table->setText(0,0,"operation is successful.");
    return;
}
//分别取返回结果集的行数与列数
int fieldNum=mysql_num_fields(res);
int rowNum=mysql_num_rows(res);
QTextCodec *codec = QTextCodec::codecForName("GBK"); //生成一个 GBK 字符集
//调整 QTable 对象的行数与列数使其它有足够的大小显示结果集。
m_table->setNumCols(fieldNum);
m_table->setNumRows(rowNum);
MYSQL_FIELD* fields=mysql_fetch_fields(res); //取列信息
QHeader* header=m_table->horizontalHeader(); //取表头
//逐列打印表头
for (int j=0;j<fieldNum;j++){
    header->setLabel(j,codec->toUnicode(fields[j].name)); //注意这里，将字符串转换成 Unicode 以便能显示简体中文。
}
MYSQL_ROW row;
int rowID=0;
while (row=mysql_fetch_row(res)){ //逐行取值
    for (int i=0;i<fieldNum;i++){
        QString value=row[i];
        m_table->setText(rowID,i,codec->toUnicode(value) ? codec->toUnicode(value) : "NULL"); //逐列显示。
    }
    rowID++;
}
}
```

showError()函数如下：

```
void MyView::showError()
{
    initTable();
    m_table->setNumCols(1);
    m_table->setNumRows(1);
    m_table->setText(0,0,(QString)mysql_error(mysql));
```

```
}
```

这样本程序的全部功能实现完毕，加入些其它修饰，然后编译，连接即可。这里说明一下，由于 MySQL C API 实现需要 socket 连接所以在 Windows 下你需要包含头文件 #include <Winsock2.h> 连接时需要库 wsock32.lib，当然你一定要包含 mysql.h 头文件和连接 libmysql.lib 库（废话）。

最后要说一下的是生成 Visual C++ 的 dsp 文件，如果你是高手的话可以自己写一个，但我不懂，即使我能我也不想那么做。Qt 为我们提供了一个工具 qmake 使用它我们可以很容易的生成相应的 dsp 文件。为此我们还需要自己作一些工作，就是写一个工程文件这个文件的扩展名一般为 .pro。本程序的工程文件为 MyQuery.pro，内容如下：

```
TEMPLATE = app
HEADERS = MyReplaceDlg.h \
           GBKEdit.h \
           MyFindDlg.h \
           MyConnDlg.h \
           MyView.h
SOURCES = MyReplaceDlg.cpp \
           GBKEdit.cpp \
           MyFindDlg.cpp \
           MyConnDlg.cpp \
           MyView.cpp \
           main.cpp
CONFIG += qt warn_on release
TARGET = MyQuery
```

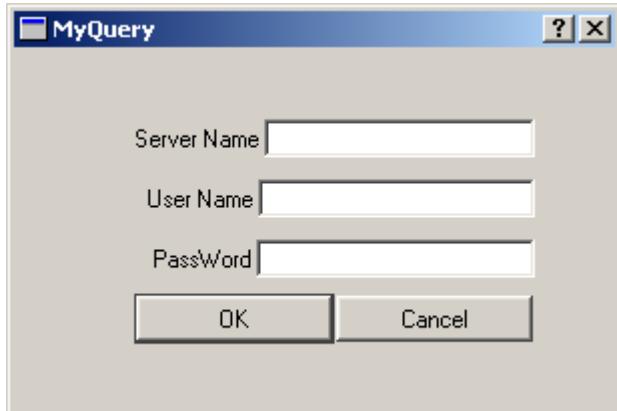
解释一下，TEMPLATE 指明此程的类型，本例我们是一生成一个应用，所以为 app。HEADERS 指明工程包含的定义文件，同样 SOURCES 指明工程的实现文件，CONFIG 指明工程的配置信息，本例中的意思是说我们要生成一个带有 warn 信息的 qt 发布（release）版程序。鉴于篇幅的限制这里不作详细说明，有关的资料可以查阅 qt 参考文档。

本程序的操作也很简单，程序起动后先连接到一个 MySQL 服务器，然后在文本框中键入你要执行的 SQL 脚本，再按执行按钮，完成所有的操作后断开连接，最后退出程序即可。

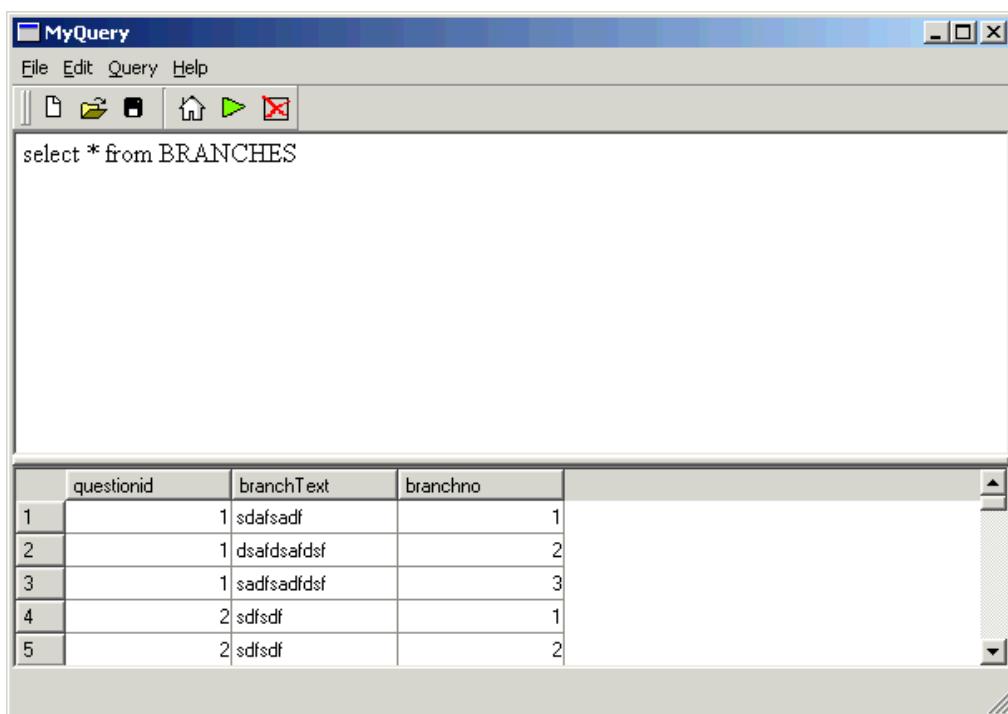
文章所附工程文件可在 http://flagon.net9.org/cppview/code/006_qt.zip。

下面是一个本程序运行时的截图，供参考。

登录时图：



运行时图：



Generic<Programming> : volatile——编写多线程程序的好帮手 让编译器帮你检测竞争条件

Andrei Alexandrescu
ye_feng 译

并不是我故意想弄糟你的心情，但是在这期专栏里，我们将讨论多线程编程这一话题。正如上一期 Generic<Programming> 里所说的，编写异常安全 (exception-safe) 的程序是非常困难的，但是和编写多线程程序比起来，那简直就是儿戏。

多线程的程序是出了名的难编写、难验证、难调试、难维护，这通常是件苦差事。不正确的多线程程序可能可以运行很多年也不出一点错，直到满足某些临界的条件时，才出现意想不到的奇怪错误。

不用说，编写多线程程序的程序员需要使用可能得到的所有帮助。这期专栏将专注于讨论竞争条件 (race conditions) —— 这通常是多线程程序中各种麻烦的根源——深入了解它并提供一些工具来防止竞争。令人惊异的是，我们将让编译器尽其所能来帮助你做这些事。

仅仅一个不起眼的关键字

尽管 C 和 C++ 标准对于线程都明显的“保持沉默”，但它们以 `volatile` 关键字的形式，确实为多线程保留了一点特权。

就象大家更熟悉的 `const` 一样，`volatile` 是一个类型修饰符 (type modifier)。它是被设计用来修饰被不同线程访问和修改的变量。如果没有 `volatile`，基本上会导致这样的结果：要么无法编写多线程程序，要么编译器失去大量优化的机会。下面我们就一个个说明。

考虑下面的代码：

```
class Gadget
{
public:
    void Wait()
    {
        while (!flag_)
        {
            Sleep(1000); // sleeps for 1000 milliseconds
        }
    }
    void Wakeup()
    {
```

```

        flag_ = true;
    }
    ...
private:
    bool flag_;
} ;

```

上面代码中 Gadget::Wait 的目的是每过一秒钟去检查一下 flag_ 成员变量，当 flag_ 被另一个线程设为 true 时，该函数才会返回。至少这是程序作者的意图，然而，这个 Wait 函数是错误的。

假设编译器发现 Sleep(1000) 是调用一个外部的库函数，它不会改变成员变量 flag_，那么编译器就可以断定它可以把 flag_ 缓存在寄存器中，以后可以访问该寄存器来代替访问较慢的主板上的内存。这对于单线程代码来说是一个很好的优化，但是在现在这种情况下，它破坏了程序的正确性：当你调用了某个 Gadget 的 Wait 函数后，即使另一个线程调用了 Wakeup，Wait 还是会一直循环下去。这是因为 flag_ 的改变没有反映到缓存它的寄存器中去。编译器的优化未免有点太……乐观了。

在大多数情况下，把变量缓存在寄存器中是一个非常有价值的优化方法，如果不用的话很可惜。C 和 C++ 给你提供了显式禁用这种缓存优化的机会。如果你声明变量是使用了 volatile 修饰符，编译器就不会把这个变量缓存在寄存器里——每次访问都将去存取变量在内存中的实际位置。这样你要对 Gadget 的 Wait/Wakeup 做的修改就是给 flag_ 加上正确的修饰：

```

class Gadget
{
public:
    ... as above ...
private:
    volatile bool flag_;
} ;

```

大多数关于 volatile 的原理和用法的解释就到此为止，并且建议你用 volatile 修饰在多个线程中使用的原生类型变量。然而，你可以用 volatile 做更多的事，因为它是神奇的 C++ 类型系统的一部分。

把 volatile 用于自定义类型

volatile 修饰不仅可以用于原生类型，也可以用于自定义类型。这时候，volatile 修饰方式类似于 const（你也可以对一个类型同时使用 const 和 volatile）。

与 const 不同，volatile 的作用对于原生类型和自定义类型是有区别的。就是说，原

生类型有 volatile 修饰时，仍然支持它们的各种操作（加、乘、赋值等等），然而对于 class 来说，就不是这样。举例来说，你可以把一个非 volatile 的 int 的值赋给一个 volatile 的 int，但是你不能把一个非 volatile 的对象赋给一个 volatile 对象。

让我们举个例子来说明自定义类型的 volatile 是怎么工作的。

```
class Gadget
{
public:
    void Foo() volatile;
    void Bar();
    ...
private:
    String name_;
    int state_;
};

Gadget regularGadget;
volatile Gadget volatileGadget;
```

如果你认为 volatile 对于对象来说没有什么作用的话，那你可要大吃一惊了。

```
volatileGadget.Foo(); // ok, volatile fun called for
                      // volatile object
regularGadget.Foo(); // ok, volatile fun called for
                     // non-volatile object
volatileGadget.Bar(); // error! Non-volatile function called for
                      // volatile object!
```

从没有 volatile 修饰的类型到相应的 volatile 类型的转换是很平常的。但是，就想 const 一样，你不能反过来把 volatile 类型转换为非 volatile 类型。你必须用类型转换运算符：

```
Gadget& ref = const_cast<Gadget&>(volatileGadget);
ref.Bar(); // ok
```

一个有 volatile 修饰的类只允许访问其接口的一个子集，这个子集由类的实现者来控制。用户只有用 const_cast 才可以访问这个类型的全部接口。而且，象 const 一样，类的 volatile 属性会传递给它的成员（例如，volatileGadget.name_ 和 volatileGadget.state_ 也是 volatile 变量）。

volatile，临界区和竞争条件

多线程程序中最简单也是最常用的同步机制要算是 mutex (互斥对象) 了。一个 mutex 只提供两个基本操作：Acquire 和 Release。一旦某个线程调用了 Acquire，其他线程再调用 Acquire 时就会被阻塞。当这个线程调用 Release 后，刚才阻塞在 Acquire 里的线程中，会有一个且仅有一个被唤醒。换句话说，对于一个给定的 mutex，只有一个线程可以在 Acquire 和 Release 调用之间获取处理器时间。在 Acquire 和 Release 调用之间执行的代码叫做临界区(critical section)。Windows 的用语可能会引起一点混乱，因为 Windows 把 mutex 本身叫做临界区，而 Windows 的 mutex 实际上指进程间的 mutex。如果把它们分别叫作线程 mutex 和进程 mutex 可能会好些。)

Mutex 是用来避免数据出现竞争条件。根据定义，所谓竞争条件就是这样一种情况：多个线程对数据产生的作用要依赖于线程的调度顺序的。当两个线程竞相访问同一数据时，就会发生竞争条件。因为一个线程可以在任意一个时刻打断其他线程，数据可能会被破坏或者被错误地解释。因此，对数据的修改操作，以及有些情况下的访问操作，必须用临界区保护起来。在面向对象的编程中，这通常意味着你在一个类的成员变量中保存一个 mutex，然后在你访问这个类的状态时使用这个 mutex。

多线程编程高手看了上面两个段落，可能已经在打哈欠了，但是它们的目的只是提供一个准备练习，我们现在要和 volatile 联系起来了。我们将把 C++ 的类型和线程的语义作一个对比。

在一个临界区以外，任意线程会在任何时间打断别的线程；这是不受控制的，所以被多个线程访问的变量容易被改得面目全非。这和 volatile 的初衷是一致的——防止编译器无意地缓存这样的变量。

在由一个 mutex 限定的临界区里，只有一个线程可以进入。因此，在临界区中执行的代码有和单线程程序有相同的语义。被控制的变量不会再被乱改一气——你可以去掉 volatile 修饰。

简而言之，线程间共享的数据在临界区之外会被乱改，而在临界区之内则不会。

你通过对一个 mutex 加锁来进入一个临界区，然后你用 const_cast 去掉某个类型的 volatile 修饰，如果我们能成功地把这两个操作放到一起，那么我们就在 C++ 类型系统和应用程序的线程语义建立起联系。这样我们可以让编译器来帮我们检测竞争条件。

LockingPtr

我们需要有一个工具来做 mutex 的获取和 const_cast 两个操作。让我们来设计一个 LockingPtr 类，你需要用一个 volatile 的对象 obj 和一个 mutex 对象 mtx 来初始化它。在 LockingPtr 对象的生命期中，它会保证 mtx 处于被获取状态，而且也提供对去掉 volatile 修饰的 obj 的访问。对 obj 的访问类似于 smart pointer，是通过 operator-> 和 operator* 来进行的。const_cast 是在 LockingPtr 内部进行。这个转化在语义上是正

确的，因为 LockingPtr 在其生存期中始终拥有 mutex。

首先，我们来定义和 LockingPtr 一起工作的 Mutex 类的框架：

```
class Mutex
{
public:
    void Acquire();
    void Release();
    ...
};
```

为了使用 LockingPtr，你需要用操作系统提供的数据结构和底层函数来实现 Mutex。

LockingPtr 是一个模板，用被控制变量的类型作为模板参数。例如，如果你希望控制一个 Widget，你就要这样写 LockingPtr <Widget>。

LockingPtr 的定义很简单，它只是实现了一个单纯的 smart pointer。它关注的焦点只是在于把 const_cast 和临界区操作放在一起。

```
template <typename T>
class LockingPtr {
public:
    // Constructors/destructors
    LockingPtr(volatile T& obj, Mutex& mtx)
        : pObj_(const_cast<T*>(&obj)),
          pMtx_(&mtx)
    {   mtx.Lock();   }
    ~LockingPtr()
    {   pMtx_->Unlock();   }
    // Pointer behavior
    T& operator*()
    {   return *pObj_;   }
    T* operator->()
    {   return pObj_;   }

private:
    T* pObj_;
    Mutex* pMtx_;
    LockingPtr(const LockingPtr&);
    LockingPtr& operator=(const LockingPtr&);
};
```

尽管很简单，LockingPtr 对于编写正确的多线程代码非常有用。你应该把线程间共享

的对象声明为 `volatile`，但是永远不要对它们使用 `const_cast`——你应该始终是用 `LockingPtr` 的自动对象（automatic objects）。让我们举例来说明。

比如说你有两个线程需要共享一个 `vector<char>` 对象：

```
class SyncBuf {
public:
    void Thread1();
    void Thread2();
private:
    typedef vector<char> BufT;
    volatile BufT buffer_;
    Mutex mtx_; // controls access to buffer_
};
```

在一个线程的函数里，你只需要简单地使用一个 `LockingPtr<BufT>` 对象来获取对 `buffer_` 成员变量的受控访问：

```
void SyncBuf::Thread1() {
    LockingPtr<BufT> lpBuf(buffer_, mtx_);
    BufT::iterator i = lpBuf->begin();
    for (; i != lpBuf->end(); ++i) {
        ... use *i ...
    }
}
```

这样的代码很容易编写，也很容易理解——每当你需要使用 `buffer_` 时，你必须创建一个 `LockingPtr<BufT>` 来指向它。当你这样做了以后，你就可以访问 `vector` 的全部接口。

这个方法的好处是，如果你犯了错误，编译器会指出它：

```
void SyncBuf::Thread2() {
    // Error! Cannot access 'begin' for a volatile object
    BufT::iterator i = buffer_.begin();
    // Error! Cannot access 'end' for a volatile object
    for (; i != lpBuf->end(); ++i) {
        ... use *i ...
    }
}
```

你不能访问 `buffer_` 的任何函数，除非你进行了 `const_cast` 或者用 `LockingPtr`。这两者的区别是 `LockingPtr` 提供了一个有规则的方法来对一个 `volatile` 变量进行 `const_cast`。

`LockingPtr` 有非常好的表达力。如果你只需要调用一个函数，你可以创建一个无名的临时 `LockingPtr` 对象，然后直接使用它：

```
unsigned int SyncBuf::Size() {
    return LockingPtr<BufT>(buffer_, mtx_)->size();
}
```

回到原生类型

我们已经看到了 `volatile` 对于保护对象免于不受控的访问是多么出色，并且看到了 `LockingPtr` 是怎么提供了一个简单有效的办法来编写线程安全的代码。现在让我们回到原生类型，`volatile` 对它们的作用方式是不同的。

让我们来考虑一个多个线程共享一个 `int` 变量的例子。

```
class Counter
{
public:
    ...
    void Increment() { ++ctr_; }
    void Decrement() { --ctr_; }
private:
    int ctr_;
};
```

如果 `Increment` 和 `Decrement` 是在不同的线程里被调用的，上面的代码片断里就有 bug。首先，`ctr_` 必须是 `volatile` 的。其次，即使是一个看上去是原子的操作，比如 `++ctr_`，实际上也分为三个阶段。内存本身是没有运算功能的，当对一个变量进行增量操作时，处理器会：

- 把变量读入寄存器
- 对寄存器里的值加 1
- 把结果写回内存

这个三步操作称为 RMW (Read-Modify-Write)。在一个 RMW 操作的 Modify 阶段，大多数处理器都会释放内存总线，以使其他处理器能够访问内存。

如果在这个时候另一个处理器对同一个变量也进行 RMW 操作，我们就遇到了一个竞争条件：第二次写入会覆盖掉第一次的值。

为了防止这样的事发生，你又要用到 `LockingPtr`：

```
class Counter
{
public:
    ...
    void Increment() { ++*LockingPtr<int>(ctr_, mtx_); }
    void Decrement() { -*LockingPtr<int>(ctr_, mtx_); }

private:
    volatile int ctr_;
    Mutex mtx_;
};
```

现在这段代码正确了，但是和 SyncBuf 相比，这段代码的质量要差一些。为什么？因为对于 Counter，编译器不会在你错误地直接访问 ctr_（没有对它加锁）时产生警告。虽然 ctr_ 是 volatile 的，但是编译器还是可以编译++ctr_，尽管产生的代码绝对是不正确的。编译器不再是你的盟友了，你只有自己留意竞争条件。

那么你该怎么做呢？很简单，你可以用一个高层的结构来包装原生类型的数据，然后对那个结构使用 volatile。这有点自相矛盾，直接用 volatile 修饰原生类型是一个不好的用法，尽管这是 volatile 最初期望的用法！

volatile 成员函数

到现在为止，我们讨论了具有 volatile 数据成员的类；现在让我们来考虑设计这样的类，它会作为更大的对象的一部分并且在线程间共享。这里，volatile 的成员函数可以帮助很大的忙。

在设计类的时候，你只对那些线程安全的成员函数加 volatile 修饰。你必须假定外面的代码会在任何地方任何时间调用 volatile 成员函数。不要忘记：volatile 相当于自由的多线程代码，并且没有临界区；非 volatile 相当于单线程的环境或者在临界区内。

比如说，你定义了一个 Widget 类，它用两个方法实现了同一个操作——一个线程安全的方法和一个快速的不受保护的方法。

```
class Widget
{
public:
    void Operation() volatile;
    void Operation();

    ...
private:
    Mutex mtx_;
};
```

注意这里的重载（overloading）用法。现在 `Widget` 的用户可以用一致的语法调用 `Operation`，对于 `volatile` 对象可以得到线程安全性，对于普通对象可以得到速度。用户必须注意把共享的 `Widget` 对象定义为 `volatile`。

在实现 `volatile` 成员函数时，第一个操作通常是用 `LockingPtr` 对 `this` 进行加锁，然后其余工作可以交给非 `volatile` 的同名函数做：

```
void Widget::Operation() volatile
{
    LockingPtr<Widget> lpThis(*this, mtx_);
    lpThis->Operation(); // invokes the non-volatile function
}
```

小结

在编写对线程程序的时候，使用 `volatile` 将对你十分有益。你必须坚持下面的规则：

- 把所有共享对象声明为 `volatile`
- 不要对原生类型直接使用 `volatile`
- 定义共享类时，用 `volatile` 成员函数来表示它的线程安全性。

如果你这么做了，而且用了简单的泛型组件 `LockingPtr`，你就可以写出线程安全的代码，并且大大减少对竞争条件的担心，因为编译器会替你操心，并且勤勤恳恳地为你指出哪里错了。

在我参与的几个项目中，使用 `volatile` 和 `LockingPtr` 产生了很大效果。代码十分整洁，也容易理解。我记得遇到过一些死锁的情况，但是相对于竞争条件，我宁愿对付死锁的情况，因为它们调试起来容易多了。那些项目实际上根本没有碰到过有关竞争条件的问题。

致谢

非常感谢 James Kanze 和 Sorin Jianu 提供了很有洞察力的意见。

后续话题：

滥用 `volatile` 的实质？

在专栏“Generic<Programming>: volatile — Multithreaded Programmer's Best Friend”发表以后，我收到很多反馈意见。就像是注定的一样，大部分称赞都是私人信件，而抱怨都发到

USENET 新闻组 comp.lang.c++.moderated 和 comp.programming.threads 里去了。随后引起了很长很激烈的讨论，如果你对这个主题有兴趣，你可以去看看这个讨论，它的标题是“ volatile, was: memory visibility between threads. ”。

我知道我从这个讨论中学到了很多东西。比如说，文章开头的 Widget 的例子不太切题。长话短说，在很多系统（比如 POSIX 兼容的系统）中，volatile 修饰是不需要的，而在另一些系统中，即使加了 volatile 也没有用，程序还是不正确。

关于 volatile correctness，最重要的一个问题就是它依赖于类似 POSIX 的 mutex，如果在多处理器系统上，光靠 mutex 就不够了——你必须用 memory barriers。

另一个更理性的问题是：严格来说通过类型转换把变量的 volatile 属性去掉是不合法的，即使 volatile 属性是你自己为了 volatile correctness 而加上去的。正如 Anthony Williams 指出的，可以想象一个系统可能把 volatile 数据放在一个不同于非 volatile 数据的存储区中，在这种情况下，进行地址变换会有不确定的行为。

另一个批评是 volatile correctness 虽然可以在一个较低层次上解决竞争条件，但是不能正确的检测出高层的、逻辑的竞争条件。例如，你有一个 mt_vector 模版类，用来模拟 std::vector，成员函数经过正确的线程同步修正。考虑这段代码：

```
volatile mt_vector<int> vec;  
...  
if (!vec.empty()) {  
    vec.pop_back();  
}
```

这段代码的目的是删除 vector 里的最后一个元素，如果它存在的话。在单线程环境里，他工作地很好。然而如果你把它用在多线程程序里，这段代码还是有可能抛出异常，尽管 empty 和 pop_back 都有正确的线程同步行为。虽然底层的数据（vec）的一致性有保证，但是高层操作的结果还是不确定的。

无论如何，经过辩论之后，我还是保持我的建议，在有类 POSIX 的 mutex 的系统上，volatile correctness 还是检测竞争条件的一个有价值的工具。但是如果你在一个支持内存访问重新排序的多处理器系统上，你首先需要仔细阅读你的编译器的文档。你知道你在做什么。

最后，Kenneth Chiu 提到了一篇非常有趣的文章 <http://theory.stanford.edu/~freunds/race.ps>，猜猜题目是什么，“ Type-Based Race Detection for Java ”。这篇文章讲了怎么对 Java 的类型系统作一点小小的补充，从而让编译器和程序员一起在编译时检测竞争条件。

Eppur si muove.

【这是伽利略在被迫放弃他一直信仰的哥白尼的地动说时所说过的一句辩解的话，意思是“它（地球）仍在运动”。】

模式罗汉拳

Composite 模式与自动化测试框架的实现

作者：[透明](#)

梗概

Composite 模式将相似的对象以树型结构的方式组合在一起，使开发者可以创建复杂的对象。另外，Composite 模式要求树中的对象都有共同的超类（或者说：接口），因此可以用同样的方式来处理树中的对象。

场景

讲到 Composite 模式，总会涉及“文档格式化”这个例子，我也继续用这个例子。假设你正在设计一个文档格式化程序，这个程序的作用就是把字 (character) 格式化为行 (line of text)，很多行就组织成栏 (column)，栏再组织成页 (page)。一篇文档 (document) 还可能包括其他的元素，例如图片 (image) 之类的。栏和页中还可以有框 (frame)，框中可以再容纳栏。栏、框和行都可以包含图片。从上面的描述，你可以得到这样的一个设计：

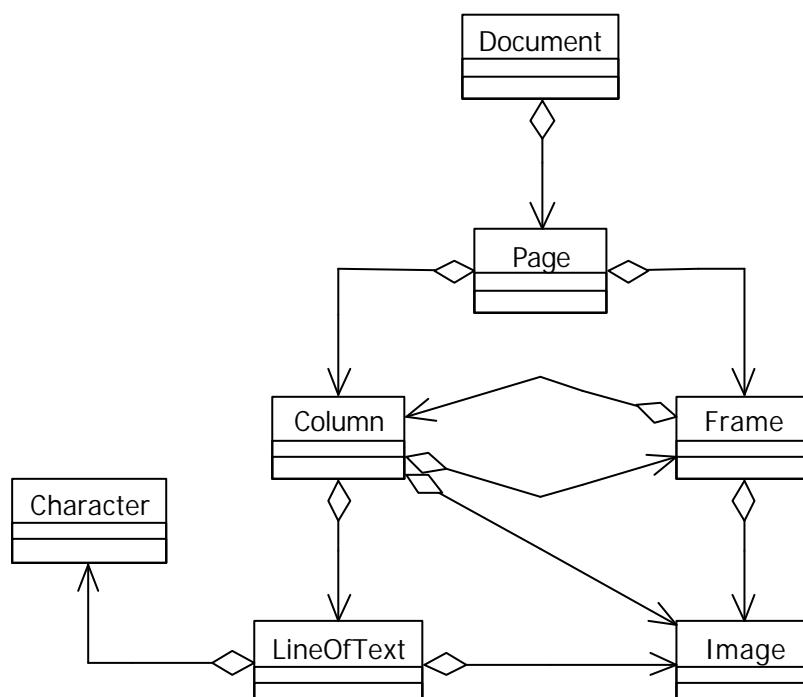


图 1：文档结构

这些不同的关联使得系统的复杂度变得巨大。你可以想象，维护这样的一个系统会有多困难！如果使用 Composite 模式，系统的复杂度会大大降低（如图 2 所示）。

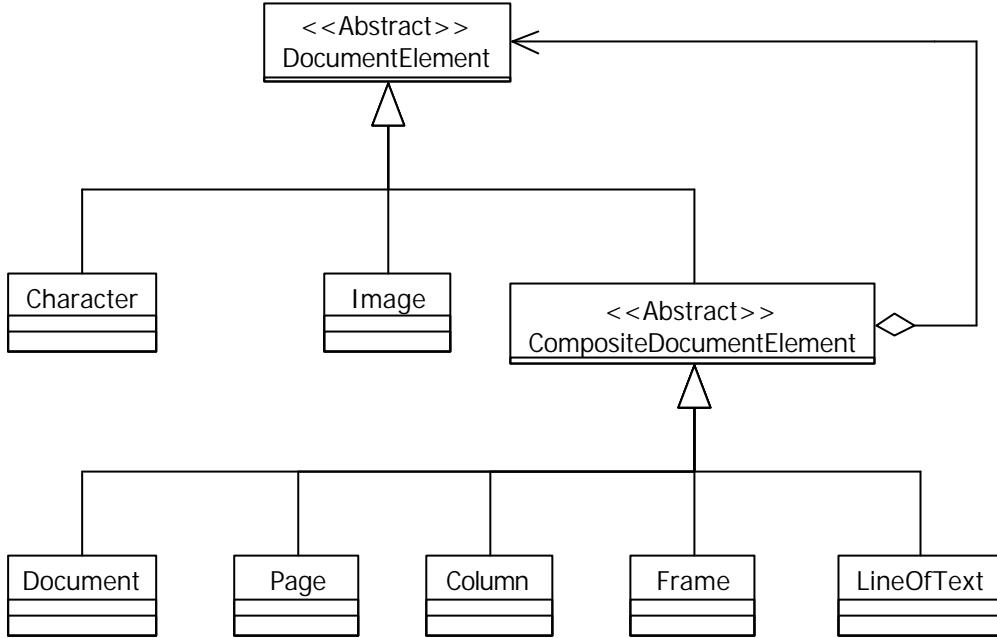


图 2：使用 Composite 模式，系统变得清晰

约束

- 你手上有一个复杂的对象，希望将它分解成一个“由部分组成整体”的类体系。
- 你希望尽量减少这个类体系的复杂度。为了达到这个目的，应该让继承树中的每个子对象都尽量少去了解其他子对象。

解决方案

使用 Composite 模式。所有类都派生自共同的基类（在上面的例子中就是 DocumentElement）；然后，可以包容其他元素的类派生自 CompositeDocumentElement 类（这个类也派生自 DocumentElement 类）。Composite 模式的一般结构如图 3 所示，具体的实现细节我们将在下面讲到。

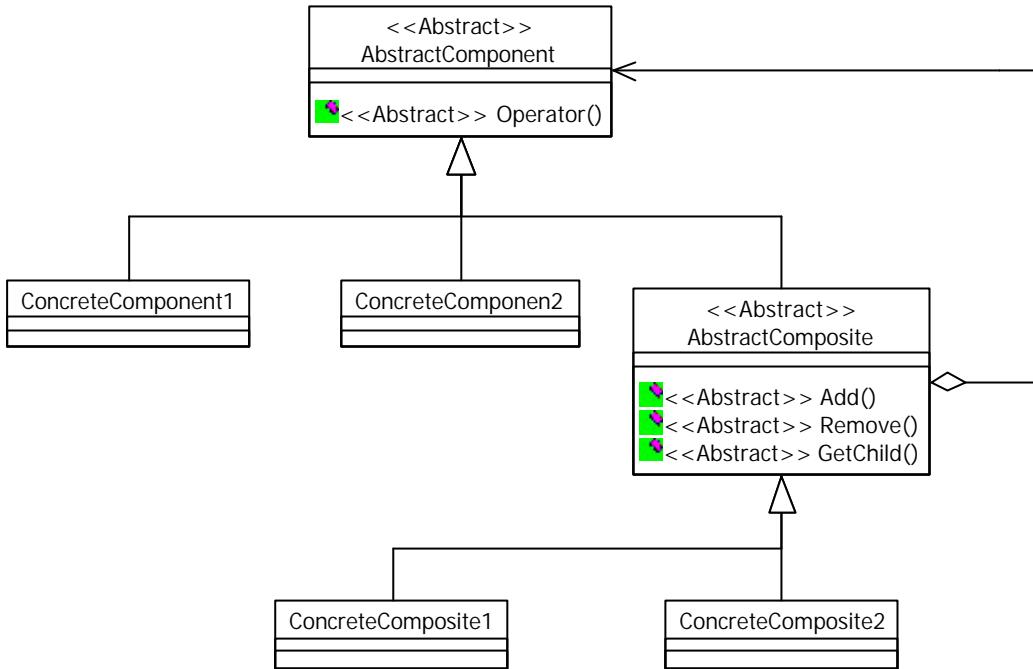


图 3：Composite 模式的结构

效果

- 树型结构的组合对象可以把其中包含的所有对象都当作 `AbstractComponent` 类的实例来处理，不管这些对象实际上是简单对象还是组合对象。
- 客户可以把组合对象也当作 `AbstractComponent` 类的实例来处理，而不必再去了解子类的实现细节。
- 如果客户在 `AbstractComposite` 派生类的对象上调用了 `AbstractComponent` 类的方法，该对象就会把调用转发给其中包含的 `AbstractComponent` 对象。
- 如果客户调用方法的对象是 `AbstractComponent` 的派生对象、而不是 `AbstractComposite` 的派生对象，并且这个方法还需要与场景相关的信息，那么 `AbstractComponent` 对象就会把请求转发给自己的父对象（也就是包容自己的对象）。
- `Composite` 模式允许任何 `AbstractComponent` 派生对象成为任何 `AbstractComposite` 派生对象的子对象（也就是被容纳的对象）。如果你需要更多的限制，就必须为 `AbstractComposite` 及其子类加上类型判别代码，这就会使 `Composite` 模式的价值打折扣。
- 被容纳的对象可能会有特有的方法。你可以在 `AbstractComponent` 类中声明这个方法，并给它一个空的实现，这样就可以在组合对象中使用这个方法，而且不需要引入类型判别代码。

实现

- 如果被包容的对象需要向包容对象转发请求，那么你可以让被包容对象保存一个指向包容对象的指针，这样转发会更简单。
- 如果要让被包容对象保存包容对象的指针，那么就必须有某种机制来保证两者关联的一致性。最好是在 Add() 方法（或其他功能相似的方法）中添加相关的设置。
- 出于效率的考虑，对象可以把父对象转发过来的方法调用的结果暂存（cache）起来。
- 如果子对象暂存了方法调用的结果，当结果不再正确的时候，父对象就必须提醒子对象。

实现一个自动化测试框架

第一个问题：什么是“自动化测试框架”？顾名思义，自动化测试框架（automated testing framework）就是可以自动对代码进行单元测试的框架。在传统的软件开发流程中，计划、设计、编码和测试都有各自独立的阶段，阶段之间不回溯，所以测试是不是自动化并不重要——反正有的是时间来慢慢测试。但是，在新的软件开发流程中，迭代周期变短，要求对代码进行频繁地重构。而这就要求单元测试必须能够自动、简便、高速地运行，否则重构就是不现实的。^{注1}

OK，我假设你已经明白了测试框架的作用，现在我们来看看它的需求。别忘了，这可是“自动化”的测试框架，它应该简单到开发者按一个按钮就能完成所有测试的程度。所以，我们必须以某种方式将测试用例（test case）组织成一个测试套件（suite），然后才能很方便地自动运行它；此外，还必须能很简单地向套件中添加新的测试用例，添加多少都可以，而且还不影响套件的正常运行；而且，测试套件还应该可以随意组合，也就是说：一个套件应该可以包含其他的套件。

看看这些需求，想到了什么？很明显，这就是一个 Composite 模式。简单的结构如图 4：

^{注1} 关于重构和自动化测试框架的概念，参见 Martin Fowler 的《Refactoring》。此外，《程序员》杂志 2001 年第 12 期技术专题“代码重构”也有关于重构的知识。

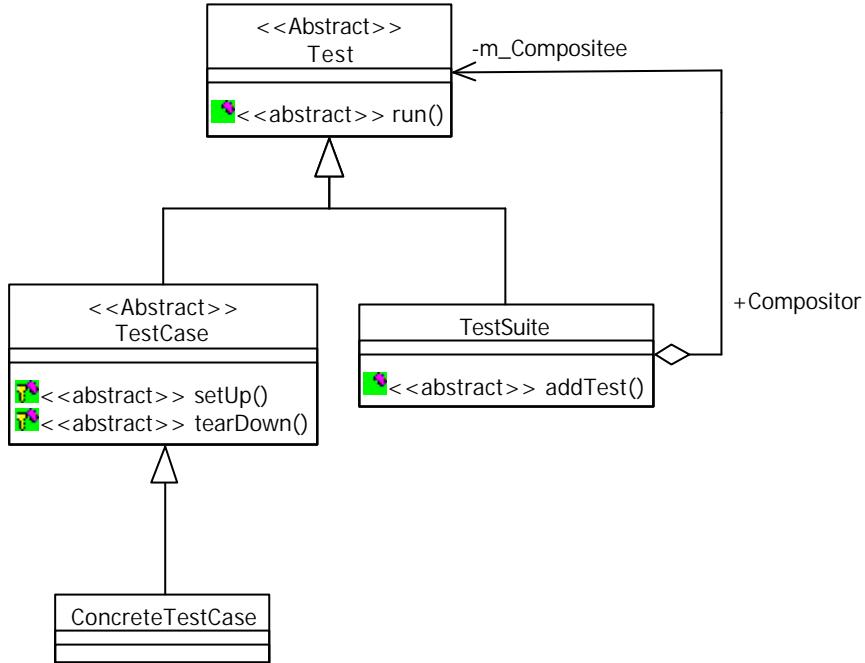


图 4：CUnit 的核心框架

在《Refactoring》中，Martin Fowler 介绍了 Java 的自动化测试框架 JUnit。参考 JUnit 的结构，我用 C++写了一个测试框架 CUnit，图 4 就是 CUnit 的结构。这是一个典型的 Composite 模式：TestSuite 可以容纳任何派生自 Test 的对象；当调用 TestSuite 对象的 run()方法时，它会遍历自己容纳的对象，逐个调用它们的 run()方法；客户无须关心自己拿到的究竟是 TestCase 还是 TestSuite，他（它）只管调用对象的 run()方法，然后分析 run()方法返回的结果就行。^{注2}

代码示例

下面，我们来看看 CUnit 的一些关键代码。首先是 Test 类，它定义了一个公用的接口：

```

class Test
{
public:
    virtual void run() = 0;
};
```

然后，TestCase 类继承了 Test 类，加入了两个新方法 setUp()和 tearDown()（关于这两个方法的用途，请参见《Refactoring》一书的相关章节）。TestCase 不实现 run()方法，所以它也是一个抽象类。用户需要从 TestCase 类派生出自己的测试用例类，并根据自己的需要来实现 run()方法。另外，用户可能想自己实现 setUp()和 tearDown()这两个方法，也有可能不做任何

^{注2} 实际上我的 CUnit 与 JUnit 还有一定的差异，因为我的主要目的是为了阐述 Composite 模式的应用，而非设计真正实用的测试框架。在 Source Forge 上有一个叫 CppUnit 的项目，其结构与 JUnit 几乎毫无二致，而且也更加完善。如果读者希望使用 C++ 的测试框架，可以到 <http://gigix.topcool.net/download/CppUnit.zip> 下载 CppUnit。

实现，所以这两个方法应该是虚方法，但不能是纯虚方法。

```
class TestCase : public Test
{
protected:
    virtual void setUp(){};
    virtual void tearDown(){};
};
```

TestSuite 类也继承了 Test 类。由于 TestSuite 是一个 Composite 类，所以它能够容纳其他 Test 类型的对象（用 addTest()方法添加）；而 TestSuite::run()则遍历这些被包容的对象，逐个调用它们的 run()方法。

```
class TestSuite : public Test
{
public:
    void addTest(Test * test){
        m_Compositee.push_back(test);
    };
    virtual void run(){
        for(int i=0; i<m_Compositee.size(); i++)
            m_Compositee[i]->run();
    };
private:
    vector<Test*> m_Compositee;
};
```

以上就是 CUnit 的主要代码。

当然，要实现自动化的单元测试，仅靠这个类体系是远远不够的，还需要其他很多的技巧。我把 CUnit 的全部代码上传到了 <http://gigix.topcool.net/download/03.zip>，欢迎有兴趣的读者与我一起讨论。

相关模式

- **Chain of Responsibility 模式**

添加相应的父对象连接，就可以把 Chain of Responsibility 模式和 Composite 模式组合起来。这样，子对象就可以从某个祖先那里得到信息，而不必知道究竟从哪个祖先得到信息。

- **Visitor 模式**

Visitor 模式可以把 Composite 模式中散布在多个类中的操作封装在一个类中。

天方夜谭 VCL

作者：虫虫

生死

生命是什么？科学和宗教都给出了不同的诠释。有句话也许说得更有意思：生命是这样一种东西，如果你把它当作一个开场或结局，那么它总是一样的；而当你把它当作一个过程，它总是不同的。其实，万事万物又何尝不是分别以生和死作为开场和结局呢？对象也不例外，不过生成以及销毁对象都需要健全的机制作保证。否则不仅对象本身遭殃，甚至会导致程序乃至整个系统崩溃。

传说中，东方的天、人、阿修罗、畜生、饿鬼、地狱六道轮回（以及由此演变出的丰都鬼城），和西方的地狱、炼狱、天堂，都有一套非常完整、严密、健全的机制，管理着时空中各种生命体。同样，一套框架也需要这么一套机制来管理记忆体中的对象，以保证正常运作。VCL自然也不例外。但虫虫这次并不准备详细分析涉及VCL对象生死的代码，相信大家对剖析涉及底层汇编都有了一定的经验。所以前面虫虫会对这方面提几句，把重点放在设计的结构和模式上，并解决一个在BBS上看到的问题。

对象生成

对象生成的方式几乎都是一样，一般流程如右图所示（VCL类的初始化是指初始化VMT和接口指针）。对象一般生存在两个地方，栈（stack）或自由存储区（free store）¹中。由于Object Pascal只支持第二种方式，所以VCL类都在自由存储区中，表现在C++中就是必须使用new和delete分配、回收空间，速度自然会比存在于栈中的普通C++类要慢一些。

控制VCL对象生成过程的代码主要在TObject::InstanceSize、TObject::NewInstance、TObject::InitInstance几个成员函数中。有兴趣的朋友可对照右图分析一下，源代码在Source\Vcl\system.pas里，没有什么特别之处。我们将把重点主要放在分析动态生成（Dynamic Creation）机制上。

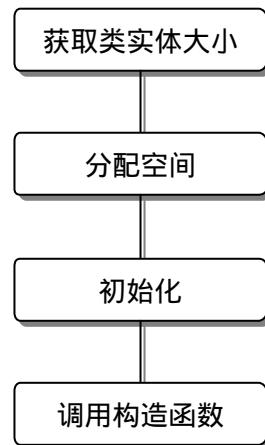


图1 对象生成流程

动态生成是一个相当实用的技术。比如上次我们提到的一个绘制图形的程序²，具体的图形以插件的方式提供，主程序对相应的图形类一无所知，但是仍然需要“动态”地生成这些对象。又比如Delphi/C++Builder的IDE对象设计器，也是一个很好的例子：鼠标双击，一个对象就动态生成在设计面板，可以供我们设计之用了。C++语言本身并没有也不可能提供对动态生成的支持，不过MFC中用宏（macro）模拟就可以取得令人满意的效果³。

仔细想来，动态生成是个很好笑的技术，它需要程序生成一个对其性质并不清楚的对象。您能造一个您不知道的东西吗？不可能。但是如果告诉您制造的原料和方法呢？那当然就很简单了。所以动态生成的关键是：留好事先约定的接口。MFC 的宏模拟就是一种方式，Object Pascal 则是使用了另一种方式。

“高级” RTTI 方式往往会引入一个所谓“类的类”，即“元类（metaclass）”的概念⁴。在 Object Pascal 中，每个类都有一个代表其相应信息的类。代表 TObject 类信息的类是 TClass，代表 TPersistent 类信息的类就是 TPersistentClass.....

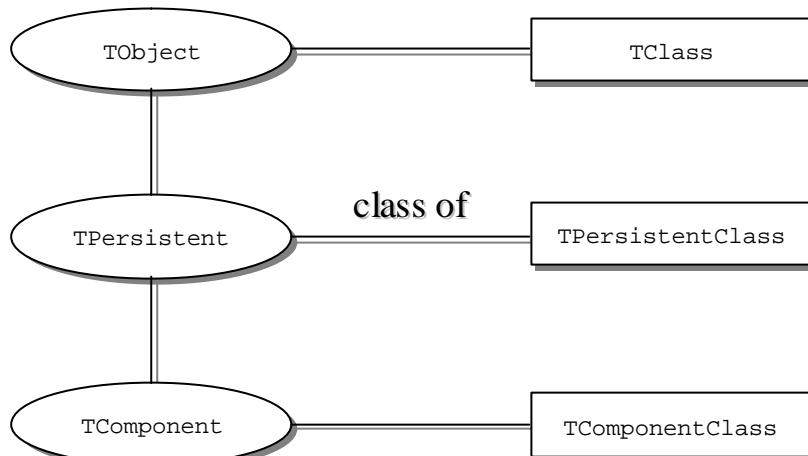


图 2 实现 RTTI 的 metaclass

Object Pascal 可以借此来实现对 TComponent 派生类的“动态生成（Dynamic Creation）”机制。

```

function CreateComponent(AOwner: TComponent; AClass: TComponentClass)
  :TComponent;
var
  Instance: TComponent;
begin
  Instance := TComponent(AClass.NewInstance);
  {try}
  Instance.Create(Owner);
  {except
  raise;
  end;}
  CreateComponent := Instance;
end;
  
```

但是在 C++ Builder 中，这一切就无效了。对于所谓的“元类”，C++ Builder 的 VCL 中只提供 TMetaClass 类，并且功能很少，甚至根本没有提供 NewInstance 等方法，巧妇难为无米之炊啊。Borland 怎么这么偏心眼呢？又怎么办呢？国内各大 BBS 上，我问过，也看别人问过这样的问题，可以总是没有答案。在国外的 BBS 上这样的问题也不少，而常见的解决方案是，用 Object Pascal 写单元，再让 C++ 调用，这也未免……

动态生成的难点

TMetaClass 究竟是什么？我们已经知道，就是指向 VMT 入口的指针。假如给我们一个 TMetaClass，我们能做到动态生成吗？对照对象生成的流程图，我们来分析

- 获取类实体大小：通过 TMetaClass::InstanceSize 可以做到；
- 分配空间：这个当然可以；
- 初始化：如果无特殊需要，直接调用 TObject::NewInstance 就行了。老弟，不是开玩笑吧？TObject::NewInstance 是 private！呵呵，想个办法绕过去，通过 VMT 表“开门”不就 OK？）
- 调用构造函数：倒！这个怎么办？TMetaClass 里可没记录过某个类的构造函数，再说，一个类的构造函数好象也不只一个吧？

唯一的问题，就出在构造函数上：我们需要一个形式固定的“虚”构造函数，也就是我们刚才说的：留好实现约定的接口。这一招的要点，跟 MFC 是一致的。

虚构造函数？C++ 的构造函数可以是虚（virtual）的吗？当然不可以。VCL 类从 TComponent 类开始，构造函数就是虚的（难怪刚才我们只生成 TComponent 的派生类），Object Pascal 所谓的虚构造是如何做到的呢？Scott Meyers 在 *Virtualizing constructors and non-member functions* 一文⁵ 中详细说明了所谓虚构造的实现方式：事先约定一个普通的虚函数，其功能是构造函数而已。也就是说，Object Pascal 所谓的虚构造函数（名字是 Create），不过是一个事先约定好功能的普通虚成员函数罢了。MFC 是这么做的，事实上，VCL 也一定差不多。在设计模式中，这叫做 FACTORY METHOD 模式，正好又名 VIRTUAL CONSTRUCTOR 模式⁶。

那么，这个虚函数一定可以在 VMT 中找出来！问题不就解决了吗？在 TComponent 类的 VMT 入口开始的若干个指针地址中找出 TComponent::TComponent（也就是 Object Pascal 中的 TComponent.Create），相信是很容易的事情吧？

```
#include <vcl.h>
#include <iostream>
using namespace std;
```

```

void main()
{
    void** p = (void**)__classid(TComponent);
    for(int i = 0; i < 15; ++i)
        cout<<*(p++)<<' \t';
}

```

这几行程序能输出 TComponent 的 VMT 中前 15 个函数地址。在我的机器上输出结果是（也许在您那里有所不同）：

40026BD4	40030A54	40026AF0	40030B2C	400309F8
40030B38	40030C30	40030F88	40030B48	40030B40
40030F90	400306CC	0000000E	00010000	45840000

我们再查查 TComponent::TComponent 的地址。通过 IDE 菜单中的 View->Debug Windows->Modules，选择 vcl50.bpl，找到 TComponent::TComponent（如下图）。看到了吧？它的地址跟上面输出的第 12 个地址完全相同，这，就是我们的目标！（其实我们还有更偷懒的方法：看看前面那段 Object Pascal 程序的汇编代码就行啦！）

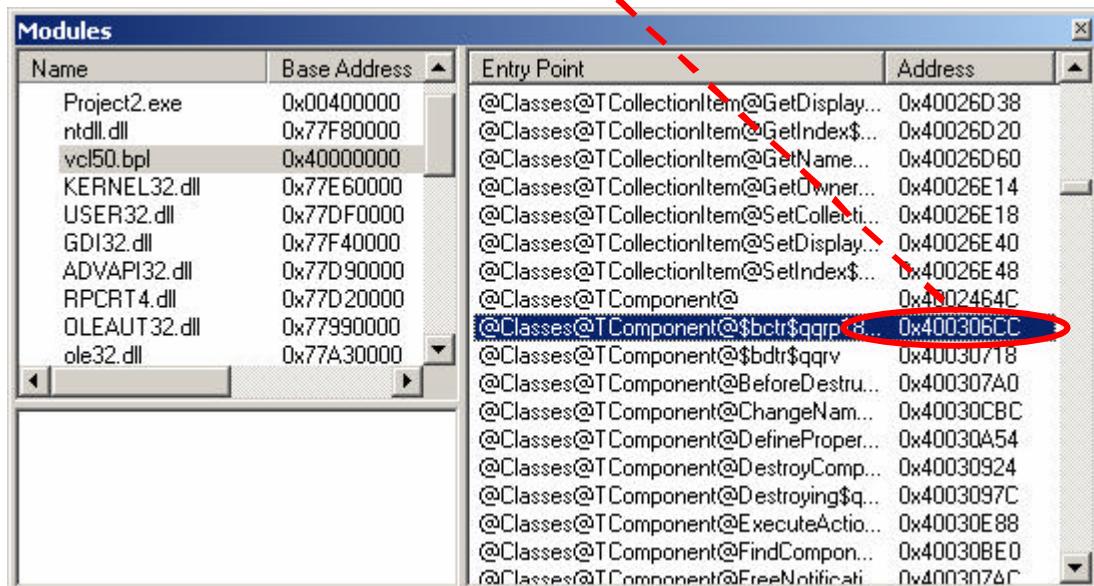


图 3 寻找 TComponent::TComponent 的地址

现在写个我们自己的 CreateComponent 很容易了吧？从 VMT 入口算，第 12 个指针是构造函数的指针。而 VCL 类是“虚”构造的，所有 TComponent 的派生类的构造函数地址也会放在那里。所以，从 VMT 第一个指针往前数 11 个，就是我们需要的第 12 个指针了。

```

TComponent* CreateComponent(TComponent* AOwner, TClass cls)
{

```

```

TComponent* r;
const void* const fn
    = *(void**)((char*)cls + vmtNewInstance),
* const fc = *((void**)cls + 11);
//fn 是 NewInstance 的地址 ,
//fc 则是构造函数的地址 , 从第一个往后数 11 个 , 即第 12 个

//下面为了偷懒 , 用几句汇编
asm{
    mov eax, cls
    call fn           //调用 NewInstance , 执行图 1 流程前 3 步
    mov r, eax
    mov edx, AOwner
    call fc           //调用构造函数
}
return r;
}

```

我们不妨测试一下。

```

#pragma inline
#include <vcl.h>
#include <iostream>
using namespace std;

class TTestButton: public TButton
{
public:
    __fastcall TTestButton(TComponent* Owner): TButton(Owner)
    {
        cout<<"Hello!"<<endl;
    }
    __fastcall ~TTestButton()
    {
        cout<<"Goodbye!"<<endl;
    }
};

TComponent* CreateComponent(TComponent* AOwner, TClass cls)
{
    //...
}

```

```

void main()
{
    TComponent* p = CreateComponent(0, __classid(TTestButton));
    cout<<AnsiString(p->ClassName()).c_str()<<endl;
    delete p;
}

```

输出：

```

Hello!
TTestButton
Goodbye!

```

结果令人满意！

属主机制

这里简单提一下 VCL 类的组织形式：属主（Owner）机制。

从 `TComponent` 开始，VCL 类的构造函数就带有一个 `TComponent*` 类型（Object Pascal 中表现为 `TComponent`）的参数 `AOwner`。每个从 `TComponent` 继承的类的实体都拥有唯一一个所有者（Owner，亦即属主），这就决定了这些类之间是树形关系。例如

```

TForm* Form1 = new TForm(Application);
TForm* Form2 = new TForm(Application);
TButton* Button1 = new TButton(Form1);

```

这样就形成了如下以 `Application` 为根（Root）的树形结构：

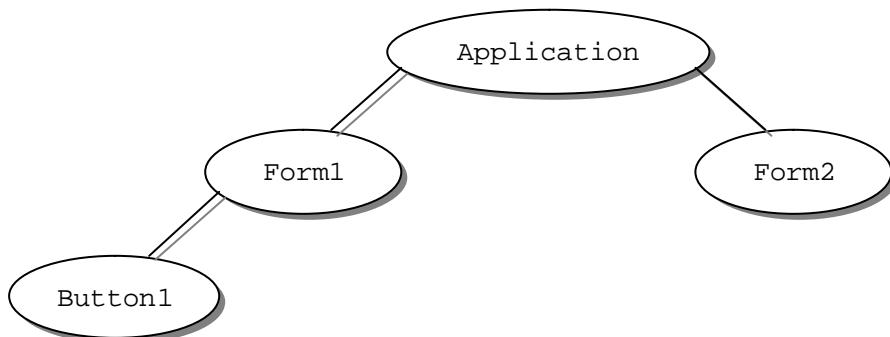


图 4 树形结构

上图中的箭头表示从属关系，也就是下图的意思

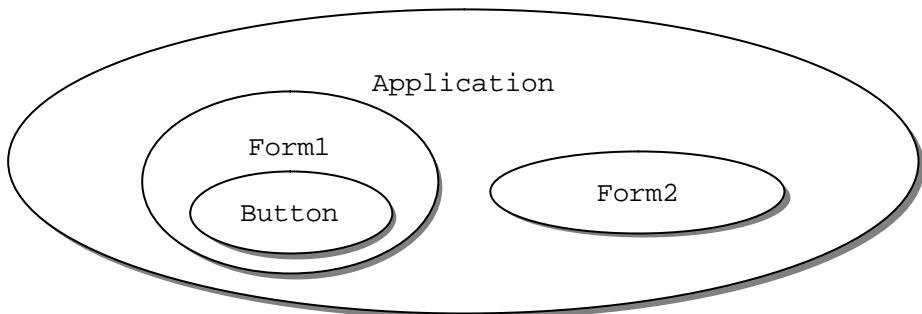


图 5 从属关系

于是当 Application 析构的时候，它会通知自己所拥有的对象 Form1 和 Form2 析构，Form1 和 Form2 再通知自身所拥有的对象析构，如此递归下去，正如同推倒了多米诺骨牌（Domino）：Form1 和 Form2 以及它们所拥有的对象，不用显式调用析构函数，资源就自动回收了。

这种 Owner 机制就是设计模式中典型的结构型模式 COMPOSITE（组合）⁶，即某个对象拥有一系列类似的对象，而这一系列对象中的每一个又拥有一系列的对象，如此递归下去，管理起来很方便。由 Application 向其所拥有的对象发送析构的消息的方式，是典型的行为模式 Observer（观察者），这也是 VCL 消息处理的基本模型，我们以后会详细讨论这一问题。

小结

本来开始想对对于 VCL 类实体具体的生成和销毁做汇编级的分析，后来发现没有多少新的东西，就留给有兴趣的朋友钻研吧。我不得不承认，这篇文章的“黑客”气太重了点儿，扩充性并不好。下次我们将逐步进入 VCL 消息处理机制，看看它的精华部分。

参考

1. Herb Sutter. *Exceptional C++*. Addison-Wesley, Reading, MA. 1999.
侯捷 .《Exceptional C++中文版》. 培生教育出版集团 . 2000 .
2. 虫虫 .《天方夜谭 VCL：多态》. *C++ View* . 2001 , 10 .
3. 侯捷 .《深入浅出 MFC》,2/e .松岗电脑图资料股份有限公司 / 华中科技大学出版社 .1997 / 2001 .
4. 虫虫 .《天方夜谭 VCL：开门》. *C++ View* . 2001 , 9 .
5. Scott Meyers. *More Effective C++*. Addison-Wesley, Reading, MA. 1996.
侯捷 .《More Effective C++中文版》. 培生教育出版集团 . 2000 .
6. GoF. *Design Patterns: Elements of Resuable Object-Oriented Software*. Addison-Wesley, Reading, MA. 1995.
李英军等 .《设计模式：可复用面向对象软件软件的基础》. 机械工业出版社 . 2000 .

C++ 标准委员会投票记事之 Valley Forge

《C++ STL（中文版）》试读

量子计算简介

Generic Programming：再谈 Min 和 Max

大师与信徒

SGI STL 源码剖析之 Iterator

芝麻开门之 OpenGL：设置 OpenGL (二)

C++ 批判系列之六：多继承

深入浅出OOD (一)

第 7 期目录 2002 年 5 月

闲话乱弹

大师与信徒 P3

潮起潮落

C++ 标准委员会投票记事之 Valley Forge P4

鸟鸣涧

鸟鸣涧 P7

源码分析经验谈

SGI STL 结构剖析之一：Iterators P8

专栏

C++ 批评系列之六：多继承 P17

Generic<Programming>：再谈 Min 和 Max P19

P = A + D：量子计算简介（一） P26

模式罗汉拳：Interlude 之深入浅出 OOD（一） P40

芝麻开门之 OpenGL：设置 OpenGL（二） P44

书斋

《C++ STL 中文版》试读 P55

主编： 王曦

封面： 熊剑青

本期编辑： 王昕

主页：<http://www.c-view.org>

电邮：cppview@sohu.com

大 师 与 信 徒

痴人

大师者，在某领域有着卓越声誉之人也。因此对于大师的观点，我等尚在苦苦寻觅之徒不可不听也。

信徒者，唯大师马首是瞻也。但凡大师所言，皆深以为然，奉为圭臬。但凡遇事，必先云：某大师于某某处曰某某某。

然而大师和信徒都是一般的人。也就是说，大师与凡人之间的观点也就会不尽相同。由于大师在领域中的卓越地位，他们的观点通常比之他人的观点要更具权威性，所以，引用大师的观点来辅助自己的论点并没有什么错。

不过，不知道大家有没有想到这样的一个问题。那就是：由于大师关注的角度与一般人不太可能一样，因此有时大师的观点并不足以在一些非常基本的场所起大作用。举例来说，在现代物理中的相对论，如果我们把它拿来对付经典物理中的问题，那会是如何一般景象呢？虽说这样做并没有错，但总让人觉得有“高射炮打蚊子”之嫌吧。

说了上面那么多的废话，其实我想表达的意思是：对于大师的观点，我们不可不借鉴一番，但也要考虑到自身的实际情况，适当挑选对自己帮助最大的部分实施，其他的则可以稍微地进行一些修改或裁剪。如果放到自身的学习过程中那就是：对于如何学习，那是自己的事情，别人的观点只是起着一些辅助作用。如果一切都人云亦云的话，即便你重复的是大师的言论，也只能证明你是一个不错的“传声筒”而已。如果你能够从别人的言论中获得对自己有用的信息，并使得自身的提高超过最初的基础的话。我想这两种情况之间的差距，不啻于画家与画匠吧。

在读书时总听到一句话：“师父领进门，修行靠个人”。总觉得此话不错，首先它肯定了师父（我们也可以把大师等同于另一个师父）的作用，但其实它突出的却是第二句，也就是突出了个人的作用，就是说徒弟不一定要比师父差（其实应该说是：不超过师父的徒弟不是好徒弟）。从这句话中体现了先辈们盼望“长江后浪推前浪”的愿望。试想，如果我们的社会如果没有这样的进步，那我们现在说不定还在哪棵树上啃香蕉呢。

子曰：“己所不欲，勿施于人”。我想在此对此话稍做修改，让其变成：“己所欲，亦勿施于人”。对于他人，还是给他一片自由的天地吧。

投票记事之 Valley Forge

Sean A. Corfield (ALNG 翻译)

宾夕法尼亚州的弗吉谷 (Valley Forge) 是数次著名战役的战场。最近的一次 X3J16/WG21 会议于 1994 年 11 月在它那位于几条高速公路之间的会议中心召开。大约是为了体现此地的精神，C++ 委员会中的几个派别间干了几场小仗。很不幸，这些阵仗只是为了几个语法上的小问题而已，其中掺杂了不少政治意味在内。

历史这样被创造出来

然而，当委员会通过投票决定从标准库草案中删除掉一些部分内容时，历史就这样被创造出来了。但又有谁听说过这样的事呢？我们都应该知道标准委员会只会增加东西，不是吗？在委员会的这次表决中（即删除草案中部分内容）包括了对容器类以及流类中的一些“巴洛克式”部分的早期尝试。

随着 STL 在 1994 年 7 月的引入，我们现在拥有了一个可以用来支撑整个库的一致性框架，并正致力于把库的其他部分改造成为 STL 风格：将字符串及流模板化，增加迭代器并使类接口整体上更一致。STL 中提供了容器 `vector`，这使得位串 (`bit_string`)，动态数组 (`dyn_array`) 以及动态指针数组 (`ptr_dyn_array`) 变成多余（因为他们可以分别为 `vector<bool>`，`vector<T>` 和 `vector<T*>` 所替代而无损于功能）。

委员会对流库也作了一些小手术，去除掉了 `stdiostream` 而改用 `fstream`。`stdiostream` 的本意是用来辅助程序员混合使用 `stdio/stream` 进行编程，但是 `fstream` 的语义已逐步改变，现在这样的帮助已全无必要。流库成长过程中的另一个受害者是 `strstream`（流世界的 `sscanf / sprintf`），它现在已不被赞成使用（即：被标注为在标准的未来版本可以考虑被删除掉的潜在目标），因为 `stringstream` 可以提供一种更安全使用类 `string` 的替代方法。

关于异常的斗争

英国人提出了反对意见，他们说异常差不多破坏了每个程序。如下的代码片段很好地展示了异常是如何把事情搞乱的

```
void f()
{
    T*      p = new T;
    // 一些处理
    delete p;
}
```

如果处理过程引发异常抛出，那么语句 “`delete p;`” 便不会被执行，这就导致了内存泄漏。为了让上面的代码具有“异常安全性 (exception-safe)”，我们就得采用与资源管理相关的“获得即初始化 (initialisation is acquisition)” 惯用法：

```
class T_handle
{
public:
    T_handle(T* pp)
```

```

        : p(pp) { }
~T_Handle()
{ delete p; }
operator T*()
{ return p; }
private:
    T*      p;
};

void f()
{
    T_Handle p(new T);
    // 一些处理
    // 当离开作用区域或
    // 异常被抛出时 ,
    // P 会被自动删除掉
}

```

英国人将这种做法视为一张基本的惯用法。这使得我们在草案登记投票中投赞成票前需要找到一个解决方法（投票正在进行，其详情可参见 CVu7.1 中的 "The Casting Vote"）。Greg Colvin 提出了几个提议，其中涵盖了“智能指针”和垃圾收集。为了使英国人以及其他 ISO 成员满意，委员会最终采纳了其中的一个提议。这个用法与上例相似的类模版被称为 auto_ptr。Greg Colvin 同时还建议使用 counted_ptr，它可以用作引用计数对象的基础。但是库工作组的主席在向委员会其他成员介绍这个类时的描述不太让人信服，并且他还不赞成所谓的“ISO 敲诈”（即对其他特性或函数库成员投赞成票的结果取决于委员会是否接受另外的某个语言特性或库类）。正如我以前说过的那样，ANSI 委员会的几个成员也持相同的观点，这样要想取得国际共识就更难了。最终的结果是 ANSI 对 counted_ptr 投了反对票，而 ISO 赞成接受 counted_ptr。经过一些增进共识的政治操作之后，一些国家改变了他们的投票方向。最终该提议被撤回，但同时也造成了在 1995 年 3 月德克萨斯 Austin 的再次表决。委员会内部的团体之争还在继续，但至少其政治意味减轻了不少。

异常合约

异常规格 (exception specifications) 能有什么用？答案可能是：“不大有用”。我们虽然可以在函数声明中指明该函数会抛出什么样的异常，但我们又可以在多大程度上信任这样的声明呢？在 Valley Forge[本次投票地点，译注]之前，答案是：“很少”。这是因为当编译器看到了一份异常规格时，它就得产生必要的代码来捕获其他的异常，并调用 unexpected()。而函数可能抛出它所设想的任何异常，这样也就破坏了函数异常规格所制订出的合约。这也是英国人所指出的另一个 bug。对于这种情况的可能的解决办法包括：从语言中去掉异常规格；或使异常规格可以在编译时被检查，不过这两者都没有得到委员会的普遍支持。最后语言特性扩展工作组提出了一个容易为大家所接受的折衷方案：在 unexpected() 中不能抛出任何违反合约的东西。这意味着编译器可以对异常规格进行合理的检查，并在规定不可调用 unexpected() 处做一些优化处理。这样就使得使用异常规格变得有吸引力得多，似乎同时就满足了几个不同的团体。

模板——他们应该放在哪？

在书写可移植的 C++ 程序中最困难的地方可能就是如何组织模板代码以使得其能在多个平台上编译。（对 Microsoft C++ 程序员来说，只要不是在 NT 上运行 VC++2.0 的话，那就用不着担心这些）。对 Borland 程序员来说，这都是预先确定好了的：你只需包含所有你的定义，编译器帮你厘清。对 CFront 程序员来说，它也有良好的定义：如果声明部分在 x.h 中的话，那么定义部分就在 x.c 中。所有的一切都显得很好，没有任何问题，但一旦你试

着在上述两个环境之间移植代码时，你就麻烦大了。所以对多平台工具提供商来说，这是一个真实而昂贵的问题。

模板编译是英国人所强烈关注的另一个问题，扩展工作组也一直在致力为此寻找一个可能的解决方案。确切地说，是 Bjarne Stroustrup 本人对此孜孜以求。在此此前的最佳结果是一个新的指令，它告诉编译器可以到哪去找模板的定义。这也是一种折中方案，它既满足了一些编译器厂商所偏爱的预处理指令，也给以其他人（和大多数用户）一个他们所喜欢的完全自动化或者说魔术般的系统。最终采用的方案意味着模板的行为和语言其他部分大致相似。声明被放在头文件中而定义则被放在源文件中。因为其行为方式与非模板如出一辙，如果需要其为静态[static]或内联[inline]的话，我们也可以将定义放入头文件中。这样用户使用固然容易，但厂家实现起来就要大费周章了。希望我们大家都能够从这一（重大）变更中得益。

explicit

正如你所预料那样，我们对另一关键字进行了表决。在叫嚷之前，先让我回过头给点背景资料。你是否经常碰到单参构造函数（a constructor with one argument）？在需要时，隐式类型转换真是妙不可言，不过它也常常在预料不到的地方冒了出来。考虑如下的代码片段：

```
void f( string );
f( 5 );
```

吃惊吗？这个调用完全合法，因为存在从 5 到 string 的隐式转换。那它都干了些什么勾当呢？是创建一个 5 个元素的 string？还是一个值为 5 的单字符 string？要想得到正确的答案，就必须清楚地了解这个类，而这种情况可能很常见。类设计者如何才能避免这些隐式转换呢？目前常用的技巧是加一个需要用户指定的哑元参数（dummy argument）或者使用中间转换类型。这两个办法都很“聪明”，但也都很“丑陋”，它们都具有我们所不想要的语义。Nathan Myers 提议增加 constructor 关键字来指明非转换构造函数（为了对称，也提议了 destructor 关键字）。这个提议总体上很受欢迎。实际上德国早就将非转换构造函数问题作为其反对票问题之一。不幸的是提议在全体委员中付诸讨论时有几个人对其语法提出了异议。我们一遍遍考虑了所有有趣的建议，最后委员会决定接受单个关键字 explicit。我们可以将一个构造函数声明为 explicit，这样它再也不能用于隐式类型转换。

哇！没有涉及内核？

当然，语言核心工作组很忙！这次他们主要处理了对象形态获取[object shape acquisition]以及对象的常数性这两个问题。他们还重审了构造期间的多态行为。如果你对此主题很感兴趣，请通过 e-mail 和我探讨。但我可不想公然跳入这样一个蛇坑！

下集预告

下次投票记事专栏将讲述 1995 年 3 月在德州 Austin 发生的事。我们将了解 CDR 投票的结果，然后我应该就可以报道是否开始 CD 投票了。

Sean A. Corfield

鸟 鸣 涧

编者按：“鸟鸣涧”栏目为老鸟和菜鸟们提供一个争鸣的地方。我们正刊出一系列与 C++ 标准库有关的题目。不论您有多少把握，能做出多少题，甚至哪怕只有一点点想法，都请您宝贵的思想写成 email，发到 cppview@sohu.com，那里有饥饿的***等着呢。如果您能提供题目，或者有什么好点子，都请您告诉我们，好吗？

本期的题目为《C++ STL 中文版》第一章后所有习题（感谢电力出版社提供该书的第一章、第五章以及附录 A 的电子版）。

习题1-1

写出下面操作所需的功能最少的迭代器种类：

- 提供无限个0
- 向文件中写入一个值序列
- 实现一个栈（后进先出队列）

习题1-2

下面列出的几种迭代器中，哪一种是可以替换的：

- 输出迭代器
- 只读前向迭代器
- 随机存取迭代器

习题1-3

迭代器同样也可以基于Fortran语言格式的Do循环：

```
for (p = first; p <= last; ++p)
    <process>(*p);
```

试比较这种格式与STL中所选择的那种格式（见本章“输入迭代器”一节）。

习题1-4

解释为什么在所写的算法中使用其他种类的迭代器，而不是随机存取迭代器？

习题1-5

解释为什么宁愿定义一个仅能通过迭代器来存取的数据结构，而不是让它可以被随机存取呢？

习题1-6

[较难] 写出这样的一个模板类bidir<FwdIt>，当我们用一个前向迭代器类型来特化FwdIt时，它的表现就和双向迭代器一样。你会采用何种方法来使它和预期中的双向迭代器行为一致？

习题1-7

[特难] 写出这样的一个模板类ran_read<InIt>，当我们用一个输入迭代器类型来特化InIt时，它的表现就和一个只读的随机存取迭代器一样。

SGI STL 结构剖析之一 Iterators

[zhouzicn](mailto:zhouzicn@sohu.com)

前言：

接触 GP&STL 大约是半年之前，学习过程中得到 CSDN 的许多网友的帮助，特别是侯捷老师的许多文章。最近，在《GP&STL》及侯捷老师的《STL 源码剖析》的影响下，我尝试着去理解 SGI STL 的代码，记录了一些个人的心得与大家交流。我的信箱是：zhouzicn@sohu.com，欢迎来信。

Iterators 是 STL 中的关键组件，SGI STL 中头文件 iterators 包含五个文件¹，它们是：stl_config.h, stl_relop.h, stddef.h, iostream.h 和 stl_iterator.h。而 stl_iterator.h 是 SGI STL 中 iterators 这个组件的核心部分，也是我们讨论的主体部分。

1. Iterators 的引入

先让我们来看一个算法：

figure 1:

```
template <class InputIterator, class T>
InputIterator find(InputIterator first, InputIterator last, const T& value)
{
    while (first != last && *first != value) ++first;
    return first;
}
```

大凡使用过 C++ 模板 (template) 的对上面的这个 find 算法不会陌生。这是一个模板函数，用于查找 first 到 last 之间，第一个值为 value 的位置，若不存在，则返回 last。那么下面我们来看看这个算法如何用于不同的数据结构。

首先看看，它对数组的支持情况。

figure 2:

```
void main(void)
{
    int A[5]={0,1,2,3,4};

    if (find(A,A+5,3) != (A+5))      { cout<<"found\n"; }
    else                                { cout<<"not found!\n"; }
}
```

¹ stl_config.h：出于移植性的考虑，该文件通过条件编译针对不同的编译器作了部分常数设定；
 stl_relop.h：该文件对 !=、>、<=、>= 四个运算符，结合模板进行重载，使之可以用于任何类型；
 stddef.h：该文件是 C++ 的库函数，通过宏定义一些常数（如 ptrdiff_t、size_t 等）；
 iostream.h：该文件 C++ 的标准输入/输出库函数。
 stl_iterator.h：该文件是 SGI STL 中 iterators 组件的核心部分。

显然，通用算法 `find` 可以用于数组，而且可以看出算法使用数组对应的指针作为参数。自然我们希望该算法也能够用于其它数据结构，那么我们看看对链表是否支持，考虑 `figure 3` 算法。

`figure 3 :`

```
struct Item
{
    int val;
    item* next;
};

void main(void)
{
    Item* listHead = new(Item);
    listHead->val = 0;
    listHead->next = null;
    Item* listTail = listHead;

    for( int j = 1; j < 5; j++)
    {
        Item *tmp=new(Item);
        tmp->val = j;
        tmp->next = null;

        listTail->next = tmp;
        listTail = listTail->next;
    }

    if (find(listHead,NULL,3)!= NULL)
    {
        //error !
        cout<<"found!\n ";
    }
    else
    {
        cout<<"not found!\n";
    }
}
```

`figure 3` 中的算法类似 `figure 2` 调用了 `find` 函数，但在编译时出错，这是什么原因呢？我们再看看算法 `find` 中的语句 `while (first != last && *first != value) ++first;` 容易看出，出错的原因在于调用 `listHead` 时，我们没有定义 `listHead!=NULL`、`*listHead`、`*listHead!=3` 和 `++listHead` 这几个操作符。好，下面我们定义这几个运算符，再看看结果。

`figure 4`

```
struct Item
{
    int val;
    item* next;
};

template <class Item>
struct Myiter
{
    Item *ptr; //保持与 listHead 的联系
    Myiter(Item*p=0):ptr(p){}
    Item&operator*() const
    {return *ptr ;} //对应 *first 运算要求
    Myiter&operator++()
```

```

    {ptr=ptr->next; return*this;}           //对应 ++first
    bool operator!= (const Myiter&i) const
    {return ptr!=i.ptr;}                   //对应 first != last
}
bool operator!= (const Item&item, int n)
{return item.val!=n;}                  //对应 *first != value

void main(void)
{
    Item* listHead = new(Item);
    listHead->val = 0;
    listHead->next = null;
    Item* listTail = listHead;

    for( int j = 1; j < 5; j++)
    {
        Item *tmp=new(Item);
        tmp->val = j;
        tmp->next = null;

        listTail->next = tmp;
        listTail = listTail->next;
    }

    if(find(Myiter<Item>(listHead),Myiter<Item>(),3) != Myiter<Item>())
        cout<<"found!\n ";
    else
        cout<<"not found!\n";
}
}

```

好，通过 Myiter 的中间作用，算法 find 终于可以作用于链表之上了。在 STL 中正是使用这种方法使通用算法不必直接作用于每个数据结构。

学过软件工程的朋友肯定还记得，我们设计软件时总是把相关的部分放到一处。再考虑 Myiter 会发现它需要了解 list 中的许多细节，所以 STL 中也就把 iterator 的这部分放在对应的数据结构（容器）中，作为它们专属的 iterators。于是就又出现了两个问题，一是对应于不同的算法和容器，它们的 iterators 是否相同；二是各个容器又自己的 iterators，它们是否有相同点。

2. 概念 (concept) 与 iterators 的分类

这一节将就不同的算法和容器 iterators 的不同点展开讨论，在正式讨论之前，我们有必要先引进几个术语。

2.1. 概念(concept)、模型 (model) 和细化 (refinement)

概念 C 是指描述某种抽象类型的一组需求 (requirements)，例如，上面定义 myiter 之前我们提出的要求 (first != last ; *first != value ; ++first) 就可以作为一个概念看待。在 STL 中的概念很多，常见的基本概念有：assignable、default constructible、equality comparable、less than comparable、range 等。

类型 T 能够满足概念 C 所有的需求，则称 T 为 C 的一个模型（model）。

概念 C1 的需求 req1 包含概念 C2 的需求 req2，则 C1 对 C2 作了细化。

对于概念的理解可以从三个角度：（1）描述某种抽象类型的一组需求；（2）某类抽象类型的组合；（3）一组正确的代码。第三点比较难理解，但很有用，因为概念都是在对算法的定义及形式化模板参数的研究取得的。（对如何通过 Tecton 语言的描述产生新概念，有必要进一步学习）。

这些术语与 iterators 有什么关系？有，太有了。每个 iterator 的出现都是在为满足算法和容器的需求的情况下定义的，这也就是我们所说的各个 iterator 的概念²。

2.2. Input Iterators

前面我们也提到了一些基本的概念（assignable、default constructible、equality comparable、lessthan comparable、range 等），在定义完备的类时，通常需要满足能进行构造或缺省构造、能被复制或赋值³，所有的 iterators 都是以类的形式定义的，自然都具备这些能力（后面的讨论不再说明）。

你肯定还记得前面为了再链表上使用算法 find 而定义 Myiter 时，需要的几个需求⁴。事实上，Input iterators 的概念就是研究 find 算法的过程中总结出来的，可以概括为：

- 能进行构造或缺省构造；
- 能被复制或赋值；
- 能进行相等性比较（对应满足 first != last 的需求）；
- 能进行逐步前向移动（对应满足 ++first 的需求）；
- 能进行读取值（x = *p，但不能改写，对应满足 *first 的需求）。

2.3. Output Iterators

讨论 Output iterators 的概念之前，我们先看看算法 copy：

figure 5

```
Template <class InputIterator, class OutputIterator>
OutputIterator copy (InputIterator first, InputIterator last,
OutputIterator result)
{
    for( ; first != last; ++result, ++first)
        *result = * first;
    return result;
}
```

如果要为这个算法设计 iterators，你会发现需要满足 first != last、++result、++first、*result = * first 这几个运算。在此基础上进行完备，就得到 Output Iterator 的需求：

- 能进行构造或缺省构造；
- 能被复制或赋值；
- 能进行相等性比较（对应满足 first != last 的需求）；
- 能进行逐步前向移动（++）（对应满足 ++result, ++first 的需求）；
- 能进行写入值（*p = x，但不能读出，对应满足 *result = * first 的需求）；

² 容器，算法等 STL 的其它组件也有相应的概念。

³ 可以参考一些 C++ 资料，如 The C++ Programming Language 等。

⁴ 它们是 first != last、*first != value、*first、++first

2.4. Forward Iterators

使用 Input Iterators 和 Output Iterators 可以基本满足算法和容器的要求，但还是有一些算法需要同时具备两者的功能，例如 replace 算法就是一例。Forward iterators 就是为满足这些需求而定义的。同样先看看 replace 算法（用新值代替旧值）的需求：

figure 6

```
template <class ForwardIterator, class T>
void replace ( ForwardIterator first, ForwardIterator last,
const T&old_value, const T& new_value)
{
    for ( ; first != last; ++first)
        if (*first == old_value)
            *first = new_value;
}
```

这里的 first、last 就是 Forward Iterators 的模型，其需求包括：first != last、++first、*first == old_value、*first = new_value。这样，Forward Iterators 的需求概括为：

- 能进行构造或缺省构造
- 能被复制或赋值
- 能进行相等性比较
- 能进行逐步前向移动（++）
- 能进行读取、写入值（ $x = *p$ ，且可 $*p = x$ ）

2.5. Bidirectional Iterators

Bidirectional Iterators 的概念是在 Forward iterators 的基础上加入逐步后向移动（--）这个要求，从而满足两个方向的运动。

2.6. Random Access Iterators

Random Access Iterators 的条件（需求）更多，它覆盖前四种 iterators 的所有条件，并且要满足下面条件（对数组的操作要求）：

- Operator + (int) //随机访问 iterators 可以随机移动，而不是逐步的。
- Operator += (int)
- Operator - (int)
- Operator -= (int)
- Operator - (random access iterator)
- Operator [] (int) //随机取某的位置容器的值
- Operator < (random access iterator) //随机访问 iterators 之间的比较
- Operator > (random access iterator)
- Operator <= (random access iterator)
- Operator >= (random access iterator)

3. Iterators 相关（associated）类型和 Traits 编程技巧

综观五种 iterators，我们发现从前到后需求越来越多⁵，也就是所谓的细化。这样在一切情况下都可以使用需求最细的 Random Access Iterators，确实可以，但不好，因为过多的需求自然会降低它的效率，实际编程时应该选择正好合适的 iterators 以期得到最高的效率，那么如何确定该选择的 iterators 呢？还有，我们如何知道 iterators 指向容器单元的类型以及其它相关类型呢？

⁵ Input iterators 与 Output iterators 除外。

为了解决这些问题，STL 采用了一种称为 traits 编程技巧来完成（可能是因为 GP STL 与 C++ 发展是同步的，C++ 只是 GP 思想的一种工具，而不是为 GP 独身制作的缘故，GP 中许多想法不能直接用 C++ 表达）。这些内容就组成了 iterators 的另外部分，并独立出来，放在 stl_iterator.h 这个文件中。

3.1. value type

value type 是为了表示 iterators 所指向容器单元的类型而引入的，你也许会问这有什么作用（最初我也这样想过），当然有用！特别在算法中常需要它。下面是 SGI STL 中的一个简单的算法：

figure 7

```
template < class ForwardIterator1,
          class ForwardIterator2,
          class *T >
inline void __iter_swap ( ForwardIterator1 a,
                         ForwardIterator2 b, T * )
{
    T tmp = *a; *a = *b; *b = tmp;
```

figure 8

```
template <class ForwardIterator1, class ForwardIterator2 >
inline void iter_swap (ForwardIterator1 a, ForwardIterator2 b)
{
    __iter_swap ( a, b, value_type(a));
}
```

figure 8 算法在调用 **figure 7** 法时，就使用了一个函数 `value_type()` 取得 iterator 所指向容器单元的类型，再用这个类型定义临时变量 `tmp` 用于交换两个单元中的内容。下面就看看 `value_type()` 这个函数是如何定义的。

figure 9

```
template <class Iterator>
inline typename iterator_traits<Iterator>::value_type* value_type (const
Iterator&)
{
    return static_cast6 <typename
iterator_traits<Iterator>::value_type*> (0);
```

在这个算法中又调用了模板类 `iterator_traits`，它的定义如下：

figure 10

```
template <class Iterator> struct iterator_traits
{
    typedef typename Iterator::value_type value_type ;
```

当然，在各个 iterator 中也会定义各自的 `value_type`，如图 **figure 10**：

figure 11

```
template <class T, class Distance> struct forward_iterator {
    typedef T value_type ;
```

通过 **figure 9、10、11** 的类型定义，**figure 7、8** 的算法就可以方便的得到 `forward_iterator1 a` 所指向容器单元的类型。

⁶ 类型转换，参见《More Effective C++》

当然，指针本身可以看作一种 iterator，但上述的方法在 C++ 中对指针不适用，如何解决指针这种情况？C++ 中的局部实例化（stl_config.h 中提到的 partial specialization 技术，对 template 中的部分参数使用实际的类型或数值）的方法可以提供答案。我们来看下面的代码：

figure 12

```
template <class T> struct iterator_traits<T*>
{
    // 针对指针
    typedef T value_type;
}
template <class T> struct iterator_traits<const T*>
{
    // 针对 const 指针
    typedef T value_type;
}
```

通过 figure 12 中的代码，使用指针带来的解决了。

如果编译器不支持这种 C++ 功能怎么办？SGI STL 也考虑到了这个情况，所以同时定义了另外一套算法，再利用条件编译把两种情况分开。有兴趣的朋友可以去看看 SGI STL 的原代码。

3.2. difference type、reference type 和 pointer type

采用类似定义 value_type 的方法，SGI STL 定义了 difference type、reference type 和 pointer type 这三种相关类型。它们分别表示两个 iterators 之间的距离、iterators 指向单元类型的引用和指针类型。

3.3. iterator tags

算法需要作用于合适的 iterator，使功能与效率俱佳，那么如何确定呢？这就是 iterator tags 的作用。STL 中，根据 5 种 iterators 的细化关系，定义了相应的一个继承关系，如下：

figure 13

```
struct input_iterator_tag {};
struct output_iterator_tag {};
struct forward_iterator_tag : public input_iterator_tag {};
struct bidirectional_iterator_tag : public forward_iterator_tag {};
struct random_access_iterator_tag : public
bidirectional_iterator_tag {};
```

然后，对一个算法针对不同的 iterators 作重载，定义不同的函数，再用一个函数确定调用那些函数。下面，以 advance(Iterator p, Distance n) 为例（函数 advance 将 p 前进 n 距离），进行具体说明（列举两种 iterators）。

figure 14

```
template <class InputIterator, class Distance>
inline void __advance( InputIterator& i, Distance n,
                      input_iterator_tag)
{
    while (n--) ++i;
}

template <class RandomAccessIterator, class Distance>
inline void __advance( RandomAccessIterator& i, Distance n,
                      random_access_iterator_tag)
{
    i += n;
}
```

figure 15

```
template <class InputIterator, class Distance>
inline void advance(InputIterator& i, Distance n)
{
    __advance(i, n, iterator_category(i));
}
```

figure 15 中的函数 `advance` 调用 **figure 14** 中重载过的函数 `__advance()`，至于到底调用哪个，决定于 `iterator_category(i)` 等于哪个 `__advance()` 种的第三个参数是 `input_iterator_tag`，还是 `random_access_iterator_tag`。SGI STL 中又定义了这个模板函数：

figure 16

```
template <class I>
inline typename iterator_traits<I>::iterator_category
iterator_category(const I&)
{
    typedef typename iterator_traits<I>::iterator_category
        category;
    return category();
}
```

当然，这里的 `iterator_category` 需要与其它相关类型一样在 `iterator_traits` 中增加如下内容：

figure 17

```
template <class Iterator>
struct iterator_traits
{
    tpyedef typename Iterator::iterator_category
        iterator_category;
}

template <class T> struct iterator_traits<T*>
{
    tpyedef typename random_access_iterator_tag
        iterator_category; //指针是一个随机访问 iterator
}

template <class T>
struct iterator_traits<const T*>
{
    tpyedef typename random_access_iterator_tag
        iterator_category;
}
```

再在 `inputIterator` 和 `RandomAccessIterator` 本身的结构中定义不同的 `iterator_category` 值：

figure 18

```
template <class T, class Distance>
struct input_iterator
{
    tpyedef input_iterator_tag iterator_category;
};

template <class T, class Distance>
struct random_access_iterator
{
    tpyedef random_access_iterator_tag iterator_category;
};
```

类似 `advance` 定义的方法，`stl_iterator.h` 中还定义了一个函数 `distance(first,last,n)` 用于为 `n` 增加上 `first` 到 `last` 的距离。

3.4. 小结

前面我们提到，在容器内定义了使用该容器的 iterators，所以不同的容器有不同的 iterators，而且用户可以自己定义 iterators，但是算法不能分清它使用的 iterators 是哪一种，SGI STL 中使用 Traits 技巧解决了这个难题。同时，从各种 iterators 抽取共性，形成相关类型，方便算法调用。总之，SGI STL 通过使用 Traits 编程技巧，把问题分成多个层次，增加了灵活性，提高了通用性，同时没有降低效率。

4. Stl_iterator.h 中的其它内容

stl.iterator.h 中还包含了一些对 iterators 进行扩展的适配器，它们是：

- **back_insert_iterator** 一种 iterator 的适配器，从容器的尾部插入。
- **front_insert_iterator** 一种 iterator 的适配器，从容器的头部插入。
- **insert_iterator** 一种 iterator 的适配器，在容器的指定位置插入。
- **reverse_iterator** 一种 iterator 的适配器，使某 iterator 逆向。
- **istreamt_iterator** 一种 iterator 的适配器，用于输入流。
- **ostreamt_iterator** 一种 iterator 的适配器，用于输出流。

参考资料：

- [1] Generic Programming and STL. M. H. Austern [Austern99]
- [2] STL 源码剖析 侯捷 [jjhou 02a]
- [3] SGI STL 原代码
- [4]The C++ Programming Language SE [Stroustrup99]

C++批评系列之六

多继承

Ian Joyner

cber 译

Eiffel和C++都提供了多继承的机制。但Java却没有，因为它认为多继承会导致许多问题的出现。不过Java提供了接口（interface）作为一种替换机制，它类似于Objective C中的协议（protocol）。Sun宣称接口可以提供多继承所能提供的所有特性。

Sun所宣称的“多继承会带来许多的问题”这个观点是对的，尤其是在C++中用以实现多继承的方法更能说明这一点。那些看起来似乎使用多继承会比单继承更简单的理由，现在都已被证明是毫无意义。例如，如何制订对于从两个类之上继承得到的具有相同名字的数据项之间的策略？它们之间是否兼容？如果是的话，那他们是否应该被合并成为一个实体？如果不兼容，那应该如何区分它们？……这样的列表可以列出很长很长。

Java的接口机制也可以用以实现多继承，但它也有一个很重要的不同之处（与C++相比）：继承中的接口必须是抽象的。由于使用接口并没有任何的实作，这就消除了需要从不同实作之间选择的可能。Java允许在接口中声明具有常数字段。当需要多继承时，他们就合并成为一个实体，这样也就不会导致歧义的产生。但是，当这些常数具有不同的值时，又有什么会发生呢？

由于Java不支持多继承，我们就不能像在C++和Eiffel中那样使用混合（mixin）了。混合是一种特性，它可以把从不同的类中得到的不同的非抽象的函数放到一起形成一个新的复杂的类。例如，我们可能希望从不同的源代码中导入一些utility函数。然而，我们也可以通过使用组合而不是继承来达到同样的效果，因此，这也就不会对Java构成一个重要的攻击了。

Eiffel在解决多继承问题时并没有导入一个单独的接口机制。

有些人可能认为，相对于多继承来说，单继承更优雅一些。这是一个很特别的观点。

BETA [Madsen 93]就属于认为“多继承不优雅”的那一种：“Beta中没有多继承，这主要是因为（对于多继承）缺乏一个深刻的理论上的理解，并且当前的（对于多继承的）建议在技术上看来也非常复杂”。他们引用了Flavors（一种可以将类混合在一起的语言）为证据。与Madsen相比，Flavors中的多继承与其顺序有关，也就是说，继承自（A，B）和继承自（B，A）是不一样的。

Ada95是另一种不支持多继承的语言。Ada95支持单继承，并把它叫做标记类型扩展（tagged type extension）。

另外一些人认为，对于某些特殊模型下的问题，多继承可以提供优雅的解法，因此为之付出的努力也是值得的。虽然上面所列出的关于多继承的问题列表并不完善，它仍然显示：

与多继承相关的问题是可以被系统地辨识出来的，而一旦问题被确认，它们也就可以被优雅地解决。当[Sakkinen 92]对于多继承研究到达一个很深的程度后，它就得出了上述定义。

Eiffel中采用的方法是，多继承会引发一些有趣的且有挑战性的问题，然后再优雅地解决它们。程序员所需做的所有决定都被限制在类的继承子句中。它包括使用renaming来保证众多从继承中得来的同名特性最终成为具有不同名字的特性，对于继承而来的特性所施展的新的export策略：redefining和undefining，以及用来消除歧义的select。在所有的情况下，编译器都会为我们做好这一切，为了使得语义清晰而不管是选择使用fork或是join，程序员都具有完全的控制权。

C++中相对Eiffel来说有着另外一种不同的用于消除歧义的机制。在Eiffel中，在renames子句中，特性间必须有着不同的名字。在C++中，可以使用域解析操作符`::`来区分成员。Eiffel的做法好处在于，歧义在声明中就被消除掉了。Eiffel的继承子句相对C++的来说要复杂不少，但它的代码也显得更简单，更稳固，并更具弹性。这也就是声明方法与操作符方法相比的好处所在。在C++中，每次当我们碰到在多个成员间具有歧义时，我们必须在代码中使用域解析操作符。这使得代码变得混乱不堪，影响其延展性，如果有其他地方的改变会影响歧义时，我们可能就需要在歧义可能出现的每个地方改变已有的代码。

依照[Stroustrup 94]中12.8节所说，ANSI委员会考虑过使用renaming，但是这个提议被委员会中的一个成员所阻塞掉了，他坚持让委员会中的其他成员用两周时间来好好地考虑这个问题。在12.8节中给出的例子显示了在没有显示的renaming的前提下，如何做可以得到同样的效果。问题在于，如果这都需要那些专家们使用两周来考虑如何实现，那留给我们的空间又有多少呢？

域解析操作符并不只是被用来消除多继承所带来的歧义。由于设计良好的语言可以避免歧义的出现，因此域解析操作符也是一个丑陋的，加深复杂性的实作手法。

在C++中，“如何来声明多继承中的父类们”是一个很复杂的问题。它影响到了建构函数被调用的次序，当程序员确实想从子类转到父类时也会导致问题的出现。然而，我们也可以把这个称为不好的程序设计风格。

C++和Eiffel的另一个不同之处在于直接的重复继承，Eiffel中允许：

```
class B inherit A, A end  
但  
class B : public A, public A { };  
却不被C++认可。
```

Generic<Programming>: 再谈 Min 和 Max

Andrei Alexandrescu
ye_feng 译

在 1995 年 1 月，Scott Meyers 在他发表于 C++ Report 里的一篇名为《min, max and more》[1]的文章里，对 C++ 社群提出了一个挑战。在仔细地分析了基于宏和代表当时模板技术水平的实现后，他得出结论：

那么究竟什么才是最好的方法——正确的方法——来实现 `max`？借用 Tevye 的不朽名言来说：“我告诉你：我不知道。”面对以上的分析，我越来越觉得我是在告诉人们使用宏的方法可能是最好的，但我讨厌宏。如果你知道 `max` 的一种优雅实现，请告诉我。

据我所知，这个问题跟六年前一样富有挑战性。本文将尝试这个挑战。

Min 和 Max

好，我们先来回顾一下 Scott 的问题。基于宏的 `min` 一般看起来是这样的：

```
#define min(a, b) ((a) < (b) ? (a) : (b))
```

这个宏工作得很好，因为它非常通用（能支持所有表达式，只要 `operator<` 和 `operator?:` 有意义）。不幸的是 `min` 总是要将每个参数计算两次，这可能会引起很多混乱。它的用法看上去太象一个函数了，但它的行为却不象。（关于 `min` 和一般的宏所引起的各种问题，如果你想了解更广泛的讨论，请参考 Herb 的文章[3]。）

在 C++ 标准库里，提供了一个基于模板的解决方案，简单有效：

```
template <class T>
const T& min(const T& lhs, const T& rhs)
{
    return lhs < rhs ? lhs : rhs;
}
```

你可以看到，这个方法在所有地方（参数和返回值）都用了 `const`，这是它的问题之一。假设你想做这样一件事：把两个浮点数 `a` 和 `b` 中的较小的一个的值加上 2。那么你会想这么写：

```
double a, b;
...
min(a, b) += 2;
```

如果用基于宏的 `min`，这样一点也没有问题，但是基于模板的那个就不行了，因为你不能修改 `const` 对象。Scott 指出，引入第二个版本：

```
template <class T>
T& min(T& lhs, T& rhs)
{
    return lhs < rhs ? lhs : rhs;
}
```

同样不能令人满意，因为编译器无法对付混合情况——两个参数一个是 `const` 而另一个不是。

而且，模板不能很好地处理自动类型转换和提升（promotion），就是说下面的代码不能编译：

```

int a;
short int b;
...
int smallest = min(a, b); // error: can't figure out T
                           // in template instantiation

```

不用说，基于宏的 `min` 在需要类型转换的情况下还是可以很好工作。而用基于模板的方法，你就必须写：

```
int smallest = min(a, int(b)); // aha, T is int
```

提供一个好的 `min/max` 实现的要点就是：它的行为要类似于宏，而又没有宏的各种问题。

一个（几乎是）好的开始

下面这个聪明的方法是打破成规思维方式的一个好例子：

```

template <class L, class R>
class MinResult {
    L& lhs_;
    R& rhs_;
public:
    operator L&() { return lhs_ < rhs_ ? lhs_ : rhs_; }
    operator R&() { return lhs_ < rhs_ ? lhs_ : rhs_; }
    MinResult(L& lhs, R& rhs) : lhs_(lhs), rhs_(rhs) {}
};

template <class LR>
class MinResult<LR, LR> {
    LR& lhs_;
    LR& rhs_;
public:
    operator LR() { return lhs_ < rhs_ ? lhs_ : rhs_; }
    MinResult(LR& lhs, LR& rhs) : lhs_(lhs), rhs_(rhs) {}
};

template <class L, class R>
MinResult min(L lhs, R rhs)
{
    return MinResult(lhs, rhs);
}

```

我们需要用部分特化的 `MinResult<LR, LR>` 来把两个运算符合成一个，否则 `operator L&` 和 `operator R&` 就会形成重复定义。

基于 `MinResult` 的方法把计算推迟到必须的时候，只有在取得结果的时候才会去做计算。比如：

```

int a, b;
...
min(a, b);

```

实际上什么也不做。另一方面，如果你写：

```
int c = min(a, b);
```

编译器会调用 `min` 返回的 `MinResult<int, int>` 临时对象的 `operator int&`，这个函数会进行计算并且返回正确的结果。

尽管你能够很好地修正 `const` 引起的问题（上面没有提到），基于 `MinResult` 的方法

还是不能令人满意，因为有二义性的问题。考虑下面的代码：

```
int a;
short int b;
extern Fun(int);
extern Fun(short int);
...
Fun(min(a, b)); // error! Don't know which overload to invoke!
```

`MaxResult<int, short int>` 支持两种转换：转成 `int&` 和转成 `short int&`。结果编译器无法在 `Fun` 的两个重载版本中决定选择哪一个。而基于宏的方法再一次通过了测试，如你所期望的那样，最终会调用 `Fun(int)`。

寻找一个类型

真正能够解决问题的应该是这样一个工具：对于两个类型 `L` 和 `R`，能够得到 `min(L, R)` 的恰当类型。例如，如果 `L` 是 `char`，`R` 是 `int`，那么结果应该是 `int`。假设我们已经有了这么一个工具（让我们叫它 `MINTYPE`），我们就可以得到最终的解决方案，即下面四个函数：

```
template <class L, class R>
MINTYPE(L, R)
Min(L& lhs, R& rhs)
{ return lhs < rhs ? lhs : rhs; }

template <class L, class R>
MINTYPE(const L, R)
Min(const L& lhs, R& rhs)
{ return lhs < rhs ? lhs : rhs; }

template <class L, class R>
MINTYPE(L, const R)
Min(L& lhs, const R& rhs)
{ return lhs < rhs ? lhs : rhs; }

template <class L, class R>
MINTYPE(const L, const R)
Min(const L& lhs, const R& rhs)
{ return lhs < rhs ? lhs : rhs; }
```

这四个重载的 `Min` 对应于 `const` 和非 `const` 参数的四种可能的组合。

到现在为止，一切顺利。但是怎么定义 `MINTYPE` 呢？好，能进行类型计算的神圣的技术之一就是 traits[4]。当然，我们可以这样来得到 `Min` 的类型：

```
#define MINTYPE(L, R) typename MinTraits<L, R>::Result

template <class L, class R> struct MinTraits;

// Specialization for the L == R case
template <class LR> struct MinTraits<LR, LR> {
    typedef LR& Result;
};

// Specialization for bool and char
template <> struct MinTraits<bool, char> {
    typedef char Result;
};
```

...

这虽然可行，但是你要写非常非常多的代码。C++里一共有 14 种算术类型，你必须对于它们间的每一种组合都写一个特化的 `MinTraits`。然后你还必须加入 `const` 变种。有一些技巧可以帮你简化这件工作，比如说宏，但是这还不是一流的解决方案。

即使那样，这个方案还不完整。你必须考虑指针和用户定义的类。还有，如果对于基类和派生类调用 `Min` 怎么办？比如说你定义了一个 `Shape` 类，而且定义了 `operator<` 来根据面积对 `Shape` 对象排序。

```
class Shape {
    ...
    unsigned int Area() = 0;
};

bool operator<(const Shape& lhs, const Shape& rhs) {
    return lhs.Area() < rhs.Area();
}

class Rectangle : public Shape { ... };

void Hatch(Shape& shape)
{
    Rectangle frame;
    ...
    Shape& smallest = Min(shape, frame);
    ... use smallest ...
}
```

如果上面调用的那个 `Min` 可以推算出 `Rectangle` 派生自 `Shape`，并且返回 `Shape` 的一个引用，那不是很好吗？因为 `Rectangle` 的引用可以自动转换为 `Shape` 的引用，所以这是很合理的。

但是，在你开始想做这件事的时候，你的想法已经超出 `min` 宏所能做的了。对于上面的例子，`min` 宏不能正确工作，因为表达式：

```
shape < frame ? shape : frame
```

将把两部分都转换成同样的类型，所以它相当于：

```
shape < frame ? shape : Shape(frame)
```

这不是我们想要的。事实上，它做了一件很糟糕的事——对象切片（slicing）。

这篇文章将讲述一个 `Min` 的实现，它能让你得到基于宏的版本的所有优点，并且更多。更好的是，这个实现大小也很合理——总共只有 80 行左右代码（还包括 `Max`）。感兴趣吗？把你的咖啡放到微波炉里再热一下，我们慢慢谈。

Loki

Okey，刚才我撒谎了。代码只有 80 行，但这没有把使用的类库计算在内。更确切地说，我用了 Loki，在我写的书[5]里介绍的一个通用库。在众多功能中，Loki 提供了高级的类型操纵手段。`Min` 实现里用到的 Loki 工具有：

1. `Typelists`。除了保存的是类型而不是值外，`Typelists`[6]和通常的列表一样。

例如下面语句：

```
typedef TYPELIST_3(float, double, long double)
FloatingPointTypes;
```

创建了一个包含三个类型的 typelist，保存在 `FloatingPointTypes` 里。给出一个 typelist（例如 `FloatingPointTypes`）和任意一个类型 `T`，你可以通过一个编译时的算法 `Loki::TL::IndexOf` 得到 `T` 在 typelist 里的位置，例如：

```
Loki::TL::IndexOf<FloatingPointTypes, double>::value
```

结果等于 1。如果类型不在 typelist 里，结果是 -1。

2. 我们要用到的第二个工具是 `Select` 类模板，它在[7]里有详细描述。简单来说，`Select` 允许你根据一个编译时的布尔常量在两个类型中选择一个。例如：

```
typedef Loki::Select<sizeof(wchar_t) <
    sizeof(short int), wchar_t, short int>::Result
SmallInt;
```

把 `SmallInt` 定义为 `wchar_t` 和 `short int` 中最小的整数类型。

3. `TypeTraits` 该类模板可以进行各种关于类型的推演，例如“这个类型是指针吗？它指向什么类型？”等等。我们只用到 `TypeTraits` 中的 `NonConstType` 类型定义。`TypeTraits<T>::NonConstType` 是一个 `typedef`，用来去掉 `T` 的 `const` 修饰，如果说有的话。
4. 最后，我们会用到 `Conversion` 类[7]，它可以检测任意一个类型是否可以被隐式地转换为另一个类型。`Conversion` 是实现上面提到的关于 `Shape` 和 `Rectangle` 的魔术的基础。

MinMaxTraits 类模板

为了简化类型计算，我建立了一个简单的线性层次，列出各种算术类型。基本上我是按照特定顺序列出所有算术类型的：对于任意两个算术类型，`Min` 的结果都是排在列表后面的那个。下面就是这个列表（现在先不考虑 `const`）：

```
namespace Private
{
    typedef TYPELIST_14(
        const bool,
        const char,
        const signed char,
        const unsigned char,
        const wchar_t,
        const short int,
        const unsigned short int,
        const int,
        const unsigned int,
        const long int,
        const unsigned long int,
        const float,
        const double,
        const long double)
    ArithTypes;
}
```

基本上，无符号类型在有符号类型的后面，尺寸大的类型在尺寸小的后面，浮点类型在整数类型后面。举例来说，如果你把一个 `long int` 和一个 `double` 传给 `Min`，那么结果将是 `double` 类型，因为在 `ArithTypes` 中 `double` 排在 `long int` 后面。

现在来看求 `Min` 结果类型的算法。给出两个非引用的类型 `L` 和 `R`，那么步骤是这样的：

- 先假设 **Result** 为 **R**。
- 如果 **R** 可以被隐式地转换为 **L**，那么把 **Result** 改成 **L**。
- 如果 **L** 和 **R** 是算术类型，并且在上面的 **Private::ArithTypes** 里 **R** 排在 **L** 后面，那么把 **Result** 改为 **R**。这一步用来处理所有算术转换。
- 如果 **L&** 可以被自动转换为 **R&**，而不需要引入临时变量，那么把 **Result** 改为 **R&**。这一步确保 **Min(frame, shape)** 之类的调用返回 **Shape&**。
- 如果 **R&** 可以被自动转换为 **L&**，而不需要引入临时变量，那么把 **Result** 改为 **L&**。这一步确保 **Min(shape, frame)** 之类的调用返回 **Shape&**。

你可以在下载的代码里看到 **MinMaxTraits** 的实现。上面算法里最难的部分就是判断“不牵涉临时变量的转换”。本质上，当一个非 **const** 的 **T** 的引用可以转换为非 **const** 的 **U** 的引用时，**T** 就可以转换 **U** 而不引入临时变量。

Min 和 Max 重载函数

Min 和 **Max** 各有四个重载函数，对应于 **const** 和非 **const** 参数的四种组合。为了防止上面 **Shape/Rectangle** 例子里所讨论的对象切片（slicing）问题，**Min** 的实现和经典的 **a < b ? a : b** 略有不同：

```
template <class L, class R>
typename MinMaxTraits<L, R>::Result
Min(L& lhs, R& rhs)
{ if (lhs < rhs) return lhs; return rhs; }

template <class L, class R>
typename MinMaxTraits<const L, R>::Result
Min(const L& lhs, R& rhs)
{ if (lhs < rhs) return lhs; return rhs; }

... two more overloads ...
... similar Max implementation ...
```

两个 **return** 语句保证了正确的类型转换，而不会产生对象切片。四个重载函数覆盖了所有混合情况，例如：**Min(a + b, c + d)** 或 **Min(a + b, 5)**。

分析

让我们来看看这个新开发的 **Min** 是怎么满足 Scott Meyers 的要求的。他认为一个好的 **Min/Max** 实现应该能做到下面四件事：

1. 提供函数调用的语义（包括类型检查），而不是宏的语义。**Min** 显然可以做到。
2. 支持 **const** 和非 **const** 参数（包括在同一个调用里混用）。由于那四个重载函数，**Min** 支持 **const** 和非 **const** 参数的任意组合。
3. 支持两个不同类型的参数（当然指有意义的情况）。**Min** 的确支持不同类型的参数，而且还有很多宏和简单模板方法所达不到的智能：**Min** 会分辨各种算术类型，并进行合理的转换。类型转换的选择过程（基于 **Private::ArithTypes**）可以由库的作者控制。
4. 不需要显式实例化。**Min** 不需要显式实例化。

Min 对于指针也可以正确工作（甚至指向不同但有关联的类型的指针，如 **Shape*** 和 **Rectangle***）。这是由于算法中的第一步。

一个值得注意的功能是：**Min** 用了一个可以由你配置的算法来推导结果类型，而不限

于类型系统里预先定义的规则。如果你对这里的算法不满意，你通过对它进行调整，可以做到相当多你想做的事情，包括根据语义进行类型选择。例如，`unsigned char` 和 `unsigned int` 的最小值始终选 `unsigned char` 类型，因为 `unsigned char` 的值域包含在 `unsigned int` 里。通过改变类型推导算法，你可以做到这样“聪明”的类型选择。

到目前为止，一切都非常好，但是还有一点细节需要说明。很遗憾，我可以得到的所有编译器都不能编译 Min。公正地说，每个编译器都在不同的代码段中翻了船。我确信代码是正确的，因为如果把那些编译器组合起来就可以编译了。但到现在为止，我还没有见到过一个可以运行的例子。所以，如果你可以得到一个最新的编译器，并且能够试试这段代码的话，请告诉我。

Look Ahead in Anger

这些天我在读 Ray Kurzweil 的《The Age of Spiritual Machines》[8]。Kurzweil 认为——而且相当有说服力——在 2020 年代你将可以用 1000 美元买到有和人脑一样能力的机器。

当我想让人们——也许就是我自己，希望只是有一点点老，但聪明得多——在 20 年后再读这篇文章时的情形，我就忍不住想笑。“哎呀，在 2001 年，这帮家伙用当时最流行的编程语言，连实现 `min` 和 `max` 这种不需要大脑的东西也会有麻烦。哈，这个人还写了一整篇文章，用了这么多深奥的技术，才把 `min` 和 `max` 搞定。”

是不是 `min` 和 `max` 不重要呢？我不这么认为。最小和最大是出现在数学和现实生活中的两个简单概念。如果一个编程语言无法表达数学和生活中的简单概念，那么这个语言一定有很严重的问题。“妈妈，我不在乎单词和文法，我只想写诗！”如果你必须加入一些临时变量，然后写“`a < b ? a : b`”，而这时候你脑子里实际上想的是“`min(expr1, expr2)`”，这说明你碰到一个严重问题：你是在用一个只会计算任意两个表达式的最小值的机器工作，而它不能表达最小值的概念。

这里有些不对劲，不是吗？C++ 不是唯一要指责的。Java 和 C#——两个被认为更高级的新语言——同样完全有能力表达 `min` 和 `max`。因为，你知道，`min` 和 `max` 不是对象。

也许将来这个时期会被称为“对象疯狂”。谁知道呢……我不禁懊恼地问：“程序员，你究竟去向何方？”

致谢

感谢所有在 Usenet 上参加关于 `volatile` 讨论的人们，特别是 Dave Butenhof，Kenneth Chiu，James Kanze，Kaz Kylheku，Tom Payne 和 David Schwartz。

参考书目

- [1] <http://www.aristeia.com/Papers/C++ReportColumns/jan95.pdf>
- [2] <http://www.cuj.com/experts/1902/alexandr.htm>
- [3] <http://www.gotw.ca/gotw/077.htm>
- [4] A. Alexandrescu. "Traits: the else-if-then of Types," C++ Report, April 2000.
- [5] A. Alexandrescu. Modern C++ Design (Addison-Wesley Longman, 2001).
- [6] J. Vlissides and A. Alexandrescu. "To Code or Not to Code," C++ Report, March 2000.
- [7] A. Alexandrescu. "Generic<Programming>: Mappings between Types and Values," C/C++ Users Journal Experts Forum, September 2000, <http://www.cuj.com/experts/1810/alexandr.htm>.
- [8] R. Kurzweil. The Age of Spiritual Machines: When Computers Exceed Human Intelligence (Penguin USA, 2000).

量子计算简介（一）

Starfish.H

传统计算机的极限

近30多年来，制造技术的革命已经大大提高了传统硅芯片的集成度。1971年英特尔公司生产的第一块芯片只含有2300个晶体管，而目前的奔腾4 芯片则集成了4200万个晶体管。该公司的奠基人之一摩尔在20世纪70年代发现，集成在一块芯片上的晶体管数量大约每两年增加一倍。这就是人们所说的摩尔定律。按此计算，到2010年，一块芯片上的晶体管数目将超过10亿个。然而，面对如此高密度的集成度，传统的物理规律将不再起作用，耗能和散热问题也无法克服。当存储器达到1024兆位时，集成电路的线宽将到0.1微米，这是理论上的微电子线路集成度的极限。假如超过这个极限，线路中电子的波粒二象性将表现得十分明显，其量子效应将不可忽略。根据测不准原理，自由电子在将不再是以线性规律沿线路运动。它下一个时刻在何处出现将是难以确定的，甚至会在线路间“漂移”，而不是按规定的线路运动。可想而知，用这种集成度制成的处理器是不可用的。可是目前的CPU线宽已经达到0.13微米，已经相当接近0.1微米这个理论极限了，这是否意味着电子计算机的发展已到达终点了呢？

从另一方面看，自从Alan Turing在1936年发明图灵机以来，图灵机一直是现代电子计算机的基本计算模型。虽然计算机技术疯狂地向前发展，但是现在最先进的电子计算机和1946年发明的第一台电子计算机ENIAC在本质上没有任何区别，甚至和几千年前中国人发明的算盘在本质上也是相同的。对于一些复杂的问题，例如大整数的因子分解，无向图中寻找哈密尔顿回路等，在目前的电子计算机上还没有找到有效的算法。用现在最快的计算机来分解一个300位的大整数的质因子，即使计算到整个宇宙毁灭也算不出结果。现有的计算机对这些问题并没有高效的算法，并不是因为计算速度不够快，而是问题本身的特殊性质和经典图灵机计算模型的内在缺陷造成的。要解决这些困扰人类的难题，唯一的途径就是打破经典的图灵机计算模型。

量子计算机

传统计算机在人类探索宇宙的雄心壮志前显得力不从心，于是人们不断地提出各种新型结构的计算机。量子计算机在这场“角逐”中逐渐脱颖而出。

1996年，量子计算机的先驱之一，IBM公司Thomas. J. Watson研究实验室的Charles. H. Bennett在英国的《自然》杂志新闻与评论栏声称，量子计算机将进入工程时代。同年，美国《科学》周刊科技新闻中报道，量子计算机引起了计算机领域的革命。

量子计算机的概念起源于对可逆计算机的研究。在计算机的发展中，小型化和高度集成化是一个重要的目标，但是随着芯片体积的缩小和集成度的提高，计算机的能耗对芯片的影响越来越大。能耗制约着集成度，也限制着计算机的运行速度。

20世纪60年代，IBM公司Thomas. J. Watson研究实验室的Rolf Landauer考察了能耗问题。他指出：能耗产生于计算过程中的不可逆操作。例如，对两位进行异或操作，输出的结果只有一位，这个过程损失了一个自由度，因此是不可逆的。按照热力学定律，必然会产生一定的热量。但是不可逆操作并非不可避免。例如对异或门的操作加以改进，保留一个无

用的位，这个操作就变成了可逆的。因此从物理原理上讲，只要把所有的不可逆操作改造为可逆操作，就可以实现无能耗的计算。

后来，Bennett 在严格地考虑了这个问题后证明：所有经典不可逆计算机都可以改造成可逆计算机，而不影响计算能力。在量子力学中，可逆操作可以便用一个幺正矩阵⁷来表示。

Argonne 国家实验室的 Paul Benioff 最早使用量子力学来描述可逆计算机。在量子可逆计算机中，使用一个二能级的量子体系来表示一位，这个量子体系处在量子态 0 和 1 上⁸。

量子可逆计算机是使用量子力学语言表述的经典计算机，它没有利用到量子力学的本质特性。美国著名物理学家 Richard P. Feynman 指出，这些量子特性可能在未来的量子计算机中起到本质的作用。牛津大学的理论物理学家 David Deutsch 找到了一类问题。对该类问题，量子计算机存在多项式算法，而经典计算机却需要指数算法。1994 年 Shor 给出了关于大数质因子分解的量子算法，这个算法可以在量子计算机上用多项式时间解决该问题。此算法立刻轰动了世界，掀起了研究量子计算机的热潮。

目前，世界各个先进国家都在努力发展量子技术。美国的 Los Alamos 国家实验室、麻省理工学院、加州大学等已经开展了使用核磁共振技术构造量子计算机的实验研究；英国国防部研究部最早使用光纤实现了量子密钥分发的实验；欧盟委员会与欧洲量子信息物理协会签订了为期 4 年的项目研究合同。目前我国也有不少学者研究量子信息技术，但是和国际水平还有一定差距，有待进一步加强。

量子计算的物理学背景

量子计算是指利用量子力学原理进行的计算，这与传统的计算有较大的区别。为了便于读者理解后文的内容，我们先对量子计算的量子物理学背景做一些简单的介绍。

确定性的丧失

20 世纪以前的物理学认为自然界存在两种物质：一种是粒子，它的运动状态和运动规律可以用牛顿力学来描述；另一种物质是场，它的运动规律遵循 Maxwell 方程组。但无论是哪一种，他们的运动方程都由 Laplace 方程决定。给出系统的初始状态，通过求解运动方程，就可以唯一地确定系统在任意时刻的运动状态。

按照经典物理的理论，整个世界是确定的，世界上没有真正的随机。所谓的随机只是因为我们对所需的参数认识不够而造成的。以掷硬币为例，我们如果知道了硬币的一切参数（包括质量、密度等）和外界的一切参数（包括重力，抛掷角度，抛掷力大小等），那么掷硬币的结果是完全可以通过求解运动方程计算出来的。

将这种思想推而广之，在我们的宇宙诞生之初，所有的粒子和场的初始状态都是确定的，如果宇宙的运动发展存在着一致的规律，那么整个宇宙的发展过程在宇宙诞生之初就由一系列运动方程完全确定了。宇宙中的任何事物，太阳系、地球、人类等等，一切的运动变化结果，都在宇宙诞生之初就完全确定了。

1819 年，Laplace 出版了《关于概率的哲学论文》[*Essai philosophique sur*

⁷ 幺正矩阵，Unitary Matrix，又称为酉矩阵。一个矩阵 A 是幺正矩阵当且仅当 $A \cdot A^* = I$ ，这里 A^* 是 A 的共轭转置矩阵， I 是单位矩阵。

⁸ 量子位(qubit) 和经典的比特位(bit) 最大的区别在于，一个 qubit 可以同时处于 0 和 1 两种状态！这点将在后文详细说明。

I esprobabi li tes]。Lapl ace写道：

我们应当把宇宙的现状看作它的先前状况的结果，看作随后状况的原因。假定一位神明能够知晓使得自然生机勃勃的所有力，和构成自然的所有物体在一瞬间的状况：对于这个神明来说，没有任何事物会是不确定的；未来会和过去一样在它眼前出现。

在1927 年前大多数物理学家都同意上述见解。这种Lapl ace决定论断言，如果给出宇宙在某个瞬间的状况、情境，宇宙在无论未来还是过去的任何瞬间的状况就是完全被决定的。这种观点在西方哲学界被称为“科学决定论”。

然而，量子理论的诞生彻底打破了这种确定性！

量子理论断言：我们的宇宙中存在着根本意义上的随机，这种随机不是因为参数不够无法计算造成的，而是因为时间、空间和物质之间一种未知的纠缠关系产生的。

按照量子理论，人的主观意识甚至会对外界的客观实在产生影响！这与经典的唯物论观点是相互冲突的。后文我们将会看到，对于量子而言，如果人不去观测它，它处于不确定的状态；而一旦观测它，它就会陷入一个确定的状态。量子所处的状态和人的观测方式有关。从这个意义上来说，人主观上所选择的观测方式将会直接影响到客观世界的量子的状态！

微观粒子的波粒二相性

量子理论的诞生首先从Einstein发现光的波粒二相性开始。

光的波动学说源于Huygens，波的重要特征就是存在干涉、衍射以及偏振现象。19世纪初Yong 等人所做的实验使得光的波动性得到了大量的实验证实。尤其是19 世纪70年代Maxwell 创立的电磁理论，更是揭示了光是一定频率的电磁波。

但是按照经典物理的电磁辐射理论，人们却无法解释黑体辐射⁹能量密度按照频率分布的实验结果。于是Planck不得不假设光是以离散的形式辐射出来，每一份辐射是一个光量子，它的能量为 $E = h \cdot \nu$ ，其中 ν 是辐射频率， h 是Planck常数。

当光照射到金属上的时候，就会有电子从金属中溢出，这就是光电效应。按照光的电磁理论，光的能量只决定于光的强度，与频率是无关的。但是光电效应只有在光的频率大于一定的值的时候才能激发电子溢出，而与照射光的强度无关。

Einstein意识到：电磁辐射不仅发射和吸收是以量子形式进行的，而且传播也是以量子形式进行的。Einstein认为辐射场本身就是由光量子组成。每个光子的能量就是 $E = h \cdot \nu$ 。当光照射到金属阴极上的时候，能量为 $h \cdot \nu$ 的光子被电子吸收，电子用这份能量用来克服金属表面的吸引力，剩余的能量作为自己的动能。因此，光不仅具有波动性，还具有粒子性。光具有波粒二相性。

⁹黑体辐射：普通物体发射的热辐射（红外 - 可见 - 紫外）依赖于物体温度 T 、形状、表面性质、材料性质等等。热辐射也可以约束在一个空腔内，这时系统若处于平衡态，能量密度 μ 就只是温度的函数。对于一定的频率 ν 或波长 λ ，有

$$\mu = \mu(\nu, T) \text{ 或 } \mu = \mu(\lambda, T)$$

也就是说空腔的形状等因素对能量密度没有影响，否则的话永动机就将成为可能。这种平衡辐射就叫做空腔辐射或黑体辐射。后一名称的由来是因为当辐射从小孔进入空腔以后，一般很难再射出来，即使空腔内壁是抛光的，辐射透出小孔的可能性也很小，所以小孔是黑的——只吸收不反射。远处房屋的窗户看起来是黑的也是同样的道理。

1924年，de Broglie在光的波粒二相性的启发下，提出了像电子这样的实物粒子也具有波粒二相性。任何物体都伴随着波，而且物体的运动和波的传播是不可分割的。实物粒子的能量和动量与光子的相同：

$$E = h \cdot v, \quad p = \frac{h}{2\pi} \cdot k$$

其中 k 是波矢量。这种波被称为德布罗意波。

1927年，Davisson和Germer用实验证实了电子也具有和X射线类似的衍射现象，从而证明了电子也具有波动性。不久，人们又陆续通过实验证实了中子、原子也具有波动性。人们逐渐认识到：一切的微观粒子都具有波粒二相性。

Bohr在1926年给出了物质波的正确解释。Bohr认为物质波并不代表实际物理量的波动，而只是刻画粒子空间位置分布的概率波。

图1是电子双缝干涉的实验装置。电子枪S发出的电子经过两个距离很近的平行窄缝1和2到达屏幕上。当窄缝很小时，屏幕上会出现电子形成的干涉图样。

1. 关闭2，只打开1，电子在屏幕上出现的分布如曲线A所示；
2. 关闭1，只打开2，电子在屏幕上出现的分布如曲线B所示；
3. 同时打开1和2，电子在屏幕上出现的分布如曲线C所示。

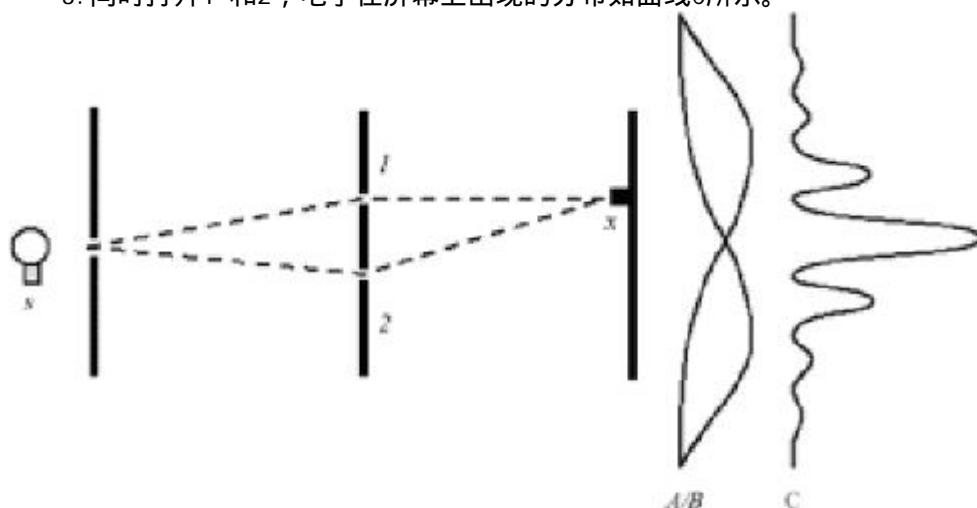


图1. 电子双缝干涉实验

曲线C并不是曲线A和曲线B的简单叠加，而是和Young的双缝衍射实验一样出现明暗相间的条纹分布。这种分布表明电子表现出和光类似的波动性。

按照Bohr的解释：电子通过双缝后，并不能到达屏幕上的任意位置，而是以一定的概率打在不同的位置上。在电子出现概率大的地方，电子打在上面的可能性就大，多个电子一次入射或者一个电子长时间入射出现在这里的电子数目就多，形成干涉极大，使电子在屏上的分布出现干涉图样。物质波是粒子分布的概率波，物质波的强度分布描述了物质粒子的空间分布。这样就把物质粒子的波动性和粒子性统一起来了。

原子里的幽灵

随着研究的深入，人们发现用经典的概率论无法完美地解释量子世界里的种种匪夷所思的现象。下面的三个实验将展现出量子世界的奇特现象。

实验1 电子双缝干涉实验

在图1的电子双缝干涉实验中，添加一个可以测量电子运动轨迹的装置。该装置如果测量到电子将会从第一个缝通过，则自动堵住第二个缝；如果测量到电子将会从第二个缝通过，则自动堵住第一个缝。这样的话，每个电子仍然有两条路径可以选择。但是屏幕上却不会再出现干涉条纹！

这就好像电子中存在着一个幽灵，可以知道实验者在欺骗它！

实验2 光的偏振实验

图2是光的偏振实验。

第一个实验结果似乎说明了偏振片A过滤掉了那些不是水平偏振方向的光子，而让那些偏振方向是水平方向的光子通过。但是按照经典概率论来解释，由于偏振片A的入射光的偏振方向是随机的，所以入射光子的偏振方向平均分布在0到360度之间，偏振方向恰好是180度（水平方向）的光子数目应该极少。如果偏振片A起到过滤作用，那么出射光应该非常弱，而不会是入射光强的一半。

第二个实验结果似乎可用偏振片的“过滤”作用来解释，因为通过A以后的光子都是水平偏振方向的，所以不可能通过垂直偏振片C；

第三个实验结果就更加匪夷所思了。如果通过A以后的光子都是水平偏振方向，那么应该没有光子能通过B，更不可能有光子通过C。

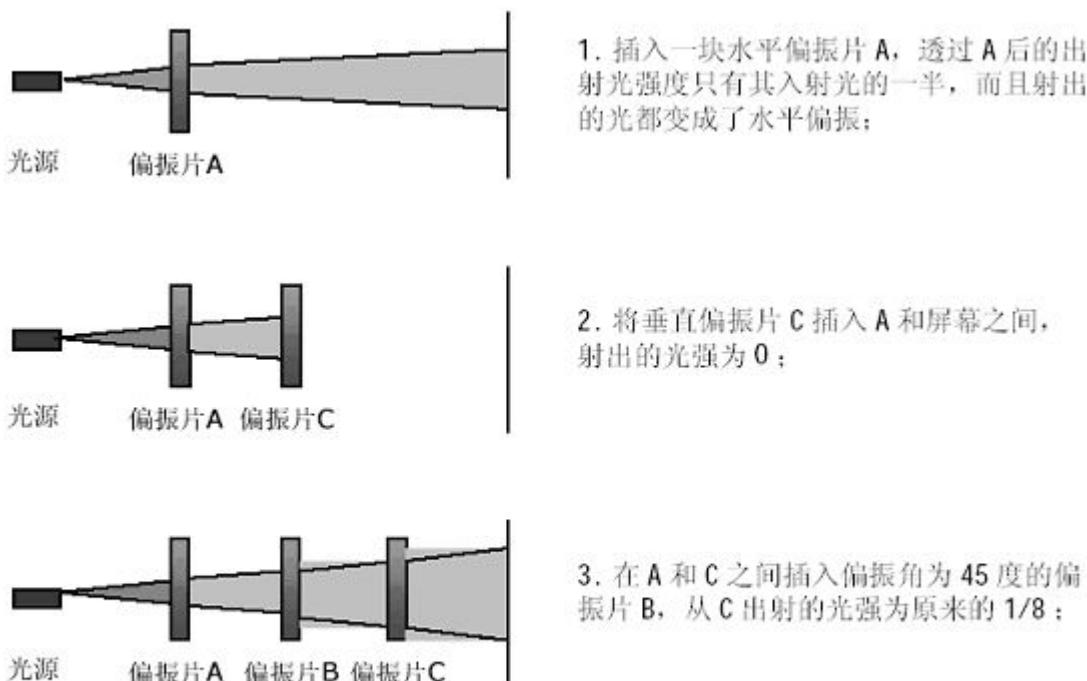
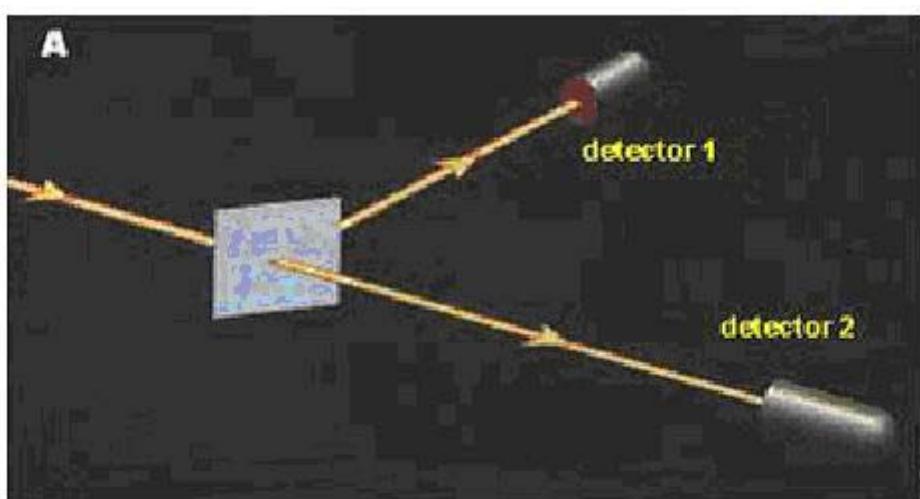
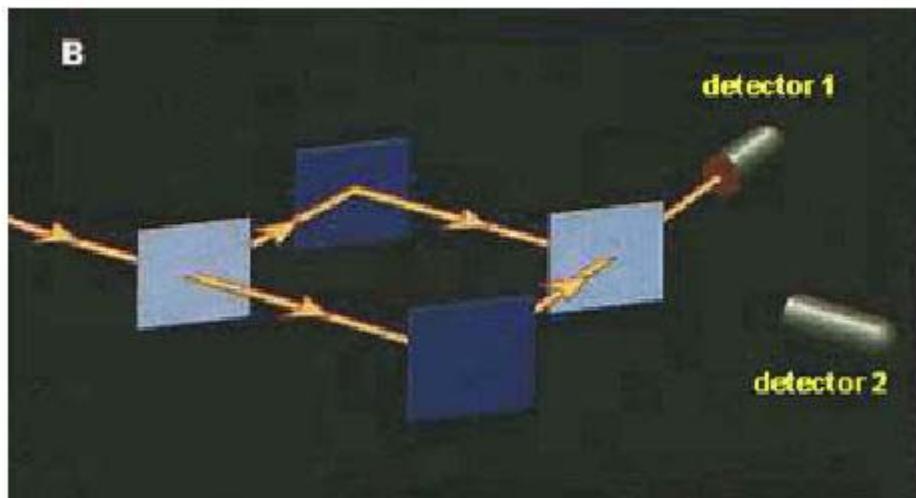


图 2. 光的偏振实验

实验3 光的反射和折射实验



A图中入射光通过一个半透镜，有一半的光被反射，另外一半被折射；如果减少入射光的强度，使得在一段时间内平均只发射一颗光子。则该光子有50%的概率通过半透镜，有50%的概率被反射。



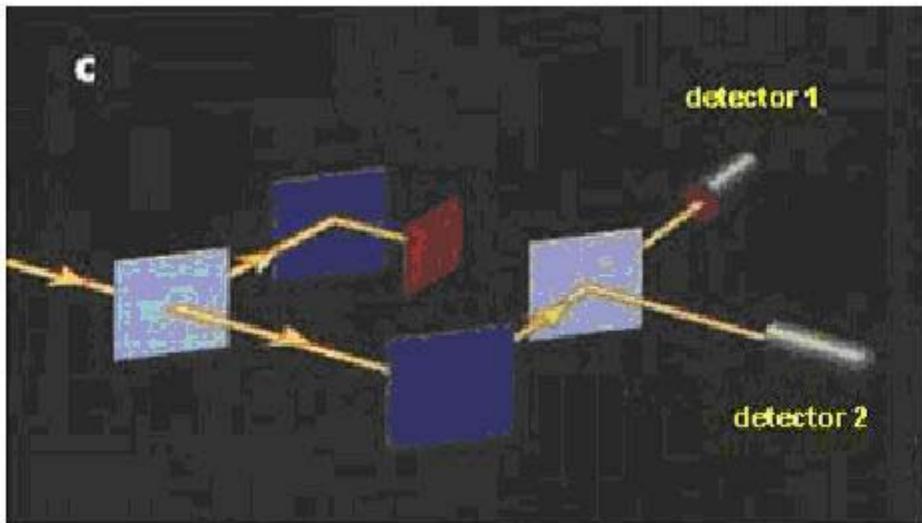
在B图所示的装置中，适当地调节上方的反光镜，总会找到一个位置，使得只有探测器1测量到光线，而探测器2测量不到任何光线。探测器1测量到的光强等于初始入射光强。

按照光的波动理论，这种现象被解释为光的干涉。入射光线通过第一个半透镜以后分为两条光线，到达第二个半透镜的时候因为上下两条光线存在光程差，所以发生干涉：上面的光线的透射光波和下面光线的反射光波恰好反相，所以互相抵消；而上面的光线的反射光波和下面光线的透射光波恰好同相，所以互相叠加。因而造成了只有探测器1探测到光线，且光强等于入射光强；而探测器2探测不到任何光线；

但奇怪的是，即使每次只发射一个光子，仍然只有探测器1探测到光子。我们用RR表示光子在两个半透镜都被发射； RT表示光子在第一个半透镜被反射，第二个被透射； TR表示光子在第一个半透镜被透射，第二个被反射； TT表示光子在两个半透镜都被透射。

按照经典的概率论，一颗光子通过半透镜被发射和被透射的概率各为50%。通过第一个半透镜后，有50%的光子被反射，这些光子可以100%地到达第二个半透镜；然后又有50%的光子被反射，50%的光子被透射。所以RR 和RT 的概率都是 $50\% * 50\% = 25\%$ ；同理，TR和TT的概率也都是25%。所有RR 和TT 的光子都可以被探测器1 探测到，所有RT和TR的光子都可以被探测器2探测到。所以应该有50%的光子被1 探测到，50%的光子被2探测到。

然而实验结果却告诉我们：调节上方的反光镜，可以使得只有探测器1测量到光子，而探测器2测量不到任何光子。用经典的概率论无法解释实验结果。



仍然是B图中的装置。这时用一块挡板挡住了上面的光线路径，如C图所示。结果探测器1和2都能探测到光线，且光强都是入射光强的1/4；

即使每次只发射一个光子，结果还是一样，各有1/4的光子被1和2探测到；

问题在于，在经过第一个探测器时被透射的光子走的是下面的路径，它怎么会知道上面的路径有没有被挡住呢？如果上面的路径没有被挡住，则那个光子在到达第二个探测器的时候绝对不会被反射；但如果上面的路径被挡住了，那个光子就好像知道这点一样，到达第二个半透镜的时候有50%的概率会被反射。

几率幅

Feynman 用“几率幅”(Probability Amplitudes) 对上述种种匪夷所思的现象给出了合理的解释。

Feynman 将一个事件发生的几率幅用一个复数 c 表示， $|c|^2$ 就是该事件发生的概率。

经典概率论认为：一个事件如果可以通过两种不可区分的途径发生，而这两种途径的概率分别是 P_1 和 P_2 ，则该事件发生的概率是 $P_1 + P_2$ ；一个事件如果需要通过两个步骤发生，这两个步骤发生的概率分别为 P_1 和 P_2 ，则该事件发生的概率是 $P_1 * P_2$ 。

Feynman 则提出了不同的概率计算规则。

Feynman 规则1：一个事件如果可以通过两种不可区分的途径发生，而这两种途径的几率幅分别是 C_1 和 C_2 ，则该事件发生的几率幅是 $C_1 + C_2$ ；这里 C_1 和 C_2 都是复数，该事件发生的概率是 $|C_1 + C_2|^2$ 。

Feynman 规则2：一个事件如果需要通过两个步骤发生，这两个步骤发生的几率幅分别为 C_1 和 C_2 ，则该事件发生的几率幅是 $C_1 * C_2$ ；这里 C_1 和 C_2 都是复数，该事件发生的概率是 $|C_1 * C_2|^2 = |C_1|^2 * |C_2|^2$ ，恰好等于 $P_1 * P_2$ ，这和经典概率论的结论一致。

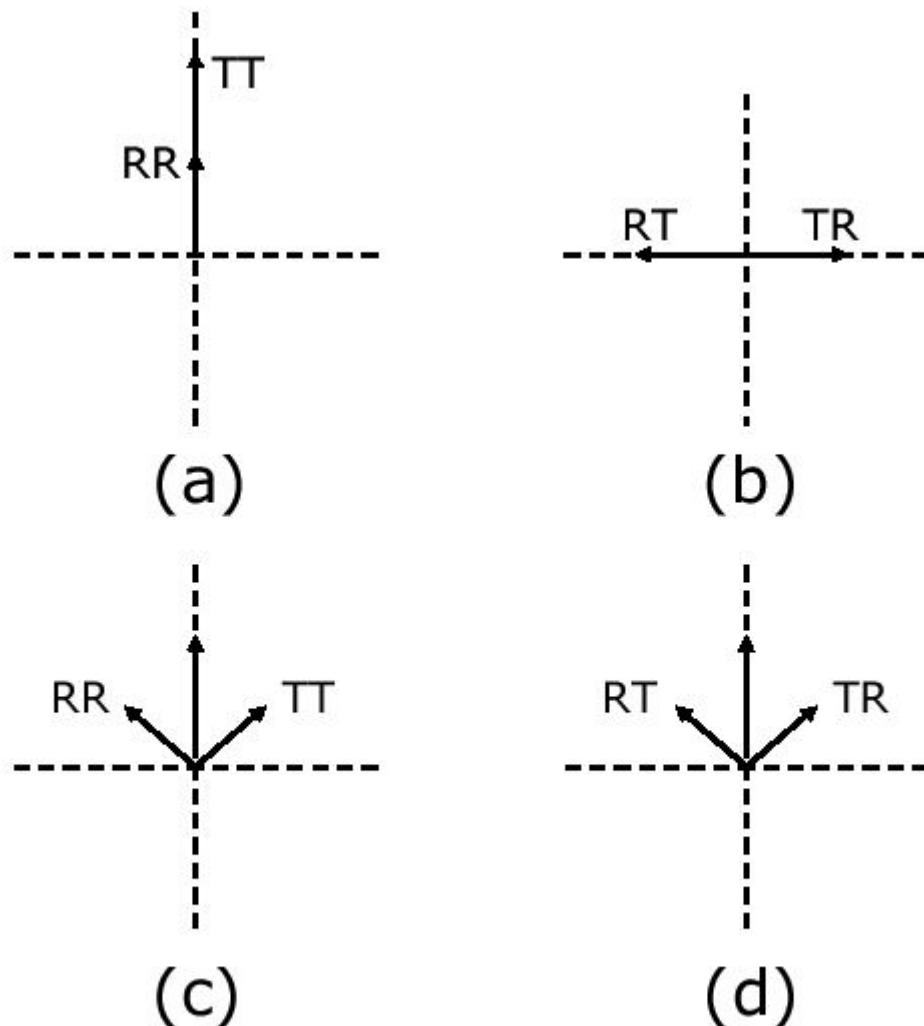


图 3. 用几率幅解释实验 3 的结果

用几率幅可以前述三个实验的结果。

在实验3的B图中，光子通过RR和TT两种途径可以被探测器1探测到，根据Feynman 规则，光子被探测器1探测到的几率幅是RR和TT的几率幅的相加，而这两个几率幅正好同向，且他们的模的平方都是0.25，所以相加之后的几率幅如图3(a) 所示；

同理，光子被探测器2 探测到的几率幅是RT和TR的几率幅的相加，而这两个几率幅正好反向，且他们的模的平方都是0.25，相加后几率幅如图3 (b)所示；

倘若实验3的B图中上下两个反光镜完全对称，光线的两条路经所经过的长度相同，那么光子的几率幅叠加如图3(c)和图3(d)所示；最后的叠加结果使得探测器1和2各有50%的概率探测到光子。

改变反光镜到半透镜的距离并不改变光子所走的路径的概率(几率幅的长度不变)，只是改变几率幅的角度。

几率幅的最重要之处在于：两个事件如果都有P的发生概率，则两个事件同时发生的概率可能为0！因为两个长度相同的复矢量相加可能因方向相反而相互抵消。

对于实验3的C图，则更容易解释了。在没有用板挡住挡住上面一条路径的时候，RR, RT和TR, TT是相互独立的事件；而用板挡住上面一条路径以后，RR, RT和TR, TT已经不是独立事件了，所以计算几率幅的方式需要改变。计算方式还是和经典概率论中的方式类似，只不过把概率相加和相乘换为几率幅的相加和相乘，最后计算结果和实验完全符合。

几率幅同样可以解释实验1的电子双缝干涉实验。在实验实验1的图1中，A、B、C曲线乃是概率的分布，而概率是几率幅的模的平方；所以C曲线并不是A和B的简单叠加，而是A和B所代表的几率幅相加以后模的平方。

我们同样可以用几率幅对实验2给出合理的解释。

我们首先回忆一下实验2的操作过程。先在光源和屏幕之间插入一块水平偏振片A，结果透过A后的出射光强度只有其入射光的一半，而且射出的光都变成了水平偏振；然后将垂直偏振片C插入A和屏幕之间，结果射出的光强为0；最后在A和C之间插入偏振角为45度的偏振片B，结果从C出射的光强为原来的1/8。整个实验过程如图2 所示。

根据第一个实验，我们知道有一半的光子可以通过水平偏振片。

我们用基向量和分别表示光子的垂直偏振方向和水平偏振方向，这两个基向量称为被测力学量的本征态。在光子到达偏振片A之前，每一颗光子的状态都可以用这两个基向量的线性组合表示为：

$$a| + b|$$

其中a, b都是复数，且 $a^2 + b^2 = 1$ 。该式表示当光子到达水平偏振片A以后，被A挡住的几率幅为a，透过A的几率幅为b。

对量子态的测量要求把该量子态分别投影到其对应的正交基上，如图4所示。在对该状态进行测量之前，观测到状态| 的几率幅为a，观测到状态| 的几率幅为b。对光子进行一次测量的以后，光子的状态就由原来的不确定的 $a| + b|$ 塌陷到一个确定的状态，或者是| ，或者是| 。

除非被测量的量子态是被测力学量的一个本征态，否则任何测量都会改变量子态，而且不能由改变后的量子态推知原来的量子态。这是量子态的一个重要特性。

用上述原理可以解释前面的偏振实验。插入偏振片可以看作是对光子的量子态进行一次测量。在光子到达偏振片A 的时候，没通过的光子塌陷到了 | 态，通过的光子塌陷到了 | 状态。

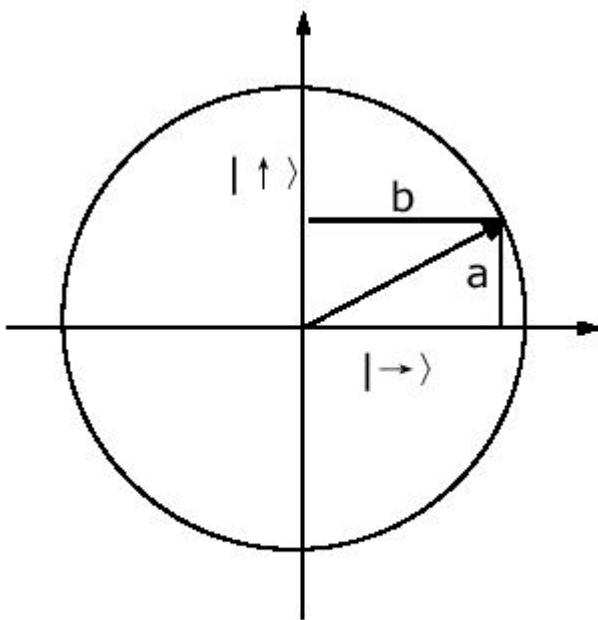


图 4. 对量子态进行测量

如果在A 后面放上垂直偏振片C , 因为通过A的光子都是态 , 所以不可能有光子通过C。在A 和C之间插入偏振片B时 , 由于偏振片B的正交基可以表示为 :

$$\left\{ \frac{1}{\sqrt{2}}(|\uparrow\rangle + |\rightarrow\rangle), \frac{1}{\sqrt{2}}(|\uparrow\rangle - |\rightarrow\rangle) \right\}$$

我们将它们分别记作 E_α 和 E_β , 量子态为 E_β 的光子将通过偏振片B。

而通过A 的光子的量子态为 :

$$|\rightarrow\rangle = \frac{1}{\sqrt{2}}\left(\frac{1}{\sqrt{2}}(|\uparrow\rangle + |\rightarrow\rangle)\right) - \frac{1}{\sqrt{2}}\left(\frac{1}{\sqrt{2}}(|\uparrow\rangle - |\rightarrow\rangle)\right) = \frac{1}{\sqrt{2}}E_\alpha - \frac{1}{\sqrt{2}}E_\beta$$

于是通过A的光子再通过B的几率幅为 $\frac{1}{\sqrt{2}}$, 也就是仍有1/2的光子可以通过B , 因而通过B以后的光强度是光源发出的光强的 $1/2 * 1/2 = 1/4$ 。

通过B 以后光子的状态塌陷为 E_β , 而 $E_\beta = \frac{1}{\sqrt{2}}(|\uparrow\rangle - |\rightarrow\rangle)$, 所以这些光子通过C的几率幅为 $\frac{1}{\sqrt{2}}$, 也就是说其中仍有1/2的光子可以通过C。因此最后通过C的光强度是原来光源发出的光强的 $1/2 * 1/2 * 1/2 = 1/8$ 。

至此 , 我们用几率幅完美地解释了前文三个实验的结果。

量子位和量子寄存器

量子位

前文我们用基向量的线性组合 $a|0\rangle + b|1\rangle$ 来表示一个光子的偏振状态。如果我们把 0 当作0， 1 当作1，则一个量子的状态可以表示为 $a|0\rangle + b|1\rangle$ ；这表示该光子处于0的几率幅是a，处于1的几率幅是b。

这样的一个光量子所包含的信息称为一个量子比特(qubit) 或量子位。

一个量子位可以同时处于1和0两种状态，但是一旦测量这个量子，它将立刻塌陷到某一个确定的状态，这点从前文的实验3中可以看出来。

量子寄存器

n个量子可以组成一个n位的量子寄存器。因为每个量子都可以同时处于0和1两种状态，所以n位的量子寄存器可以同时处于 2^n 种状态。对n位的量子寄存器进行一次运算，则可以同时产生 2^n 个结果，这就相当于一种超大规模的并行计算。

但可惜的是，我们最后需要要测量该量子寄存器中量子的状态，但因为对于一个量子只能取出0和1中的一个状态，所以虽然有 2^n 个结果，最终我们只能以一定的概率测量到其中的一个结果。

解决的方法是：在运算的过程中我们不做任何测量，只在最后进行一次测量，得到我们需要的结果。在后文的量子算法中我们将对这点作详细地说明。

量子计算的数学基础

在研究量子计算的时候如果总是需要考虑每一步计算的量子力学意义，将是很不方便的。这就好像当年电子计算机刚刚出现的时候，每一个逻辑运算门都有明确的物理学含义，但后来人们利用数字逻辑这个数学工具，将计算从物理过程抽象为数学过程，给计算机的发展带来了极大的方便。在量子计算中，也有类似的数学工具，这就是Hilbert空间。

Hilbert 空间

一个系统的量子状态由各种粒子的位置、动量、偏振、自旋等组成，并且随时间的演化过程遵循着方程，而它的状态空间可以用波函数的Hilbert空间来描述。

向量空间就是一组元素 $\{u, v, w, \dots\}$ 的集合L，并满足：

1. L对加法运算是封闭的；
2. 域F的任意一个数与L的任意一个元素相乘的结果仍然是L中的元素；
3. 对于 $u, v \in L, a, b \in F$, 满足

$$a(u+v) = au+av \in L$$

$$(a+b)u = au+bu \in L$$

$$a(bu) = (ab)u$$

则称L为域F上的向量空间。当F为复数域时，相应的向量空间就是复向量空间。

我们称定义了内积的向量空间为内积空间。内积的定义如下：对于每一对元素 $u, v \in L$ ，都有域 F 中的一个数与之对应，记为 (u, v) ，称为 u 与 v 的内积。内积具有以下性质：

$$(u, v) \geq 0$$

$$(u, v) = (v, u)^*$$

$$(w, (au + bv)) = a(w, u) + b(w, v)$$

其中 * 表示复共轭。 $(u, u) = |u|^2$ 的非负平方根定义为 u 的模。

一个完备的内积空间就是 Hilbert 空间。在 Hilbert 空间中取 m 个向量 u_1, u_2, \dots, u_m ，同时取域 F 中的 m 个数 a_1, a_2, \dots, a_m ，如果

$$a_1 u_1 + a_2 u_2 + \dots + a_m u_m = 0$$

当且仅当所有的 $a_i = 0, i = 1, 2, \dots, m$ 时才成立，那么就称向量 u_1, u_2, \dots, u_m 是线性无关的。在一个 Hilbert 空间中，如果最多有 N 个线性无关向量，则称该向量空间是 N 维的。

Dirac 表示法

量子力学系统可以由 Hilbert 空间中的向量来表示，表示量子态的向量称为状态向量。

通常量子状态空间和作用在上面的变换可用向量、矩阵来描述。但物理学家 Dirac 发明了一套更简洁的符号来表示状态向量。

用 $|\varphi\rangle$ （称为右矢）表示量子系统的量子态，用 $\langle\varphi|$ （称为左矢）表示的共轭转置。

例如，一个二维向量空间的正交基 $\{[1, 0]^T, [0, 1]^T\}$ 可以表示为 $\{|0\rangle, |1\rangle\}$ 。任意向量 $[a, b]^T$ 都可以表示为 $|0\rangle$ 和 $|1\rangle$ 的线性组合 $a|0\rangle + b|1\rangle$ ，这里 a, b 都是复数。

向量 $|\varphi\rangle$ 的共轭转置记作 $\langle\varphi|$ ，例如：

$$|\varphi\rangle = a|0\rangle + b|1\rangle = \begin{bmatrix} a \\ b \end{bmatrix}, \text{ 则 } \langle\varphi| = \begin{bmatrix} a^* & b^* \end{bmatrix}$$

这里 a^*, b^* 分别表示 a, b 的共轭复数。

两个向量 $|\varphi_1\rangle$ 和 $|\varphi_2\rangle$ 的内积记作 $\langle\varphi_1|\varphi_2\rangle$ ，例如：

$$|\varphi_1\rangle = \begin{bmatrix} a_1 \\ b_1 \end{bmatrix}, \quad |\varphi_2\rangle = \begin{bmatrix} a_2 \\ b_2 \end{bmatrix}$$

$$\langle\varphi_1|\varphi_2\rangle = \begin{bmatrix} a_1^* & b_1^* \end{bmatrix} \begin{bmatrix} a_2 \\ b_2 \end{bmatrix} = a_1^* \cdot a_2 + b_1^* \cdot b_2$$

两个向量 $|\varphi_1\rangle$ 和 $|\varphi_2\rangle$ 的外积记作 $|\varphi_1\rangle\langle\varphi_2|$ ，例如：

$$|\varphi_1\rangle = \begin{bmatrix} a_1 \\ b_1 \end{bmatrix}, \quad |\varphi_2\rangle = \begin{bmatrix} a_2 \\ b_2 \end{bmatrix}$$

$$|\varphi_1\rangle\langle\varphi_2| = \begin{bmatrix} a_1 \\ b_1 \end{bmatrix} \begin{bmatrix} a_2^* & b_2^* \end{bmatrix} = \begin{bmatrix} a_1 a_2^* & a_1 b_2^* \\ b_1 a_2^* & b_1 b_2^* \end{bmatrix}$$

算符

向量空间算符就相当于向量空间的函数，它规定了一个具体的对应关系：

$$|\varphi\rangle = A|\psi\rangle$$

我们说算符A作用于 $|\psi\rangle$ 得到了 $|\varphi\rangle$ 。注意这里等式两端的向量的维数未必相同。

满足以下条件的算符称为线性算符：

$$A(|\alpha\rangle + |\beta\rangle) = A|\alpha\rangle + A|\beta\rangle$$

$$A(k|\alpha\rangle) = k(A|\alpha\rangle)$$

这里k是任意复数。

线性算符具有以下性质：

1. 线性算符的值域也是一个右矢空间；
2. 若定义域是有限维空间，则值域空间的维数小于等于定义域空间的维数；
3. 在定义域中，那些受A作用得到零矢量的右矢全体，也构成一个右矢空间；

量子力学中出现的算符大部分都是线性算符。

复数对右矢的数乘，也可以看作算符对右矢的作用，每一个复数都可以看作一个算符。

算符的加法和乘法定义为：

$$(A + B)|\alpha\rangle = A|\alpha\rangle + B|\alpha\rangle$$

$$BA|\alpha\rangle = B(A|\alpha\rangle)$$

如果两个算符A, B 满足

$$AB = BA$$

则这两个算符是可对易的。

量子计算中通常讨论的n维向量空间的线性算符就是一个的矩阵。算符A作用到向量上就相当于进行矩阵乘法，结果还是一个n维Hilbert空间的向量；

(未完待续……)

模式罗汉拳：Interlude

深入浅出 OOD（一）

撰文/透明

有物昆成，先天地生。萧呵！谬呵！独立而不改，可以为天地母。吾未知其名，字之曰道。吾强为之名曰大，大曰逝，逝曰远，远曰反。道大，天大，地大，王亦大。

——《道德经》，第二十五章

软件不软

从 60 年代的软件危机，到今天传统软件工程方法处处碰壁的处境，都说明一个问题：软件不软（Software is Hard）[Martin, 95]。说实话，软件是一块硬骨头，真正开发过软件的人都会有此感觉。一个应用总是包含无数错综复杂的细节，而软件开发者则要把所有这些细节都组织起来，使之形成一个可以正常运转的程序，这实在不是一件简单的事。

为什么会这样？举个例子来说：用户可以很轻松地说“我要一个字处理软件”，他觉得“字处理软件”这样一个概念是再清楚不过的了，根本不需要更多的描述；而真正开发一个字处理软件却是一件困难无比的事情——我曾经亲眼看到开发字处理软件的人们是怎样受尽折磨的。为什么表述一个概念很容易，而实现一个概念很困难？因为人们太擅长抽象、太擅长剥离细节问题了。

软件不软，症结就在这里：用户很容易抽象地表达自己的需求，而这种抽象却很难转化为程序代码；软件不软，因为很简单的设想也需要大量的时间去实现；软件不软，因为满足用户的需求和期望实在太困难；软件不软，因为软件太容易让用户产生幻想。

而另一方面，计算机硬件技术却在飞速发展。从几十年前神秘的庞然大物，到现在随身携带的移动芯片；从每秒数千次运算到每秒上百亿次运算。当软件开发者们还在寻找能让软件开发生产力提高一个数量级的“银弹”[Brooks, 95]时，硬件开发的生产力早已提升了百倍千倍。这是为什么？

硬件工程师们能够如此高效，是因为他们都很懒惰。他们永远恪守“不要去重新发明轮子”的古训，他们尽量利用别人的成果。你看到有硬件工程师自己设计拨码开关的吗？你看到有硬件工程师自己设计低通滤波电路的吗？你看到有硬件工程师自己设计计时器的吗？他们有一套非常好的封装技术，他们可以把电路封装在一个接插件里面，只露出接口。别人要用的时候，只管按照接口去用，完全不必操心接插件内部的实现。

而软件工程师们呢？在 STL 被 C++ 标准委员会采纳之前（甚至之后），每个 C++ 程序员都写过自己的排序算法和链表，并都认为自己比别人写得更好...真是令人伤心。

OOD 可以让软件稍微“软”一点

软件不软，不过 OOD 可以帮助它稍微“软”一点。OOD 为我们提供了封装某一层面上的功能和复杂性的工具。使用 OOD，我们可以创建黑箱软件模块，将一大堆复杂的东西藏到一个简单的接口背后。然后，软件工程师们就可以使用标准的软件技术把这些黑箱组合起来，形成他们想要的应用程序。

Grady Booch 把这些黑箱称为类属（class category），现在我们则通常把它们称为“组件（component）”。类属是由被称为类（class）的实体组成的，类与类之间通过关联（relationship）结合在一起。一个类可以把大量的细节隐藏起来，只露出一个简单的接口，这正好符合人们喜欢抽象的心理。所以，这是一个非常伟大的概念，因为它给我们提供了封装和复用的基础，让我们可以从问题的角度来看问题，而不是从机器的角度来看问题。

软件的复用最初是从函数库和类库开始的，这两种复用形式实际上都是白箱复用。到 90 年代，开始有人开发并出售真正的黑箱软件模块：框架（framework）和控件（control）。框架和控件往往还受平台和语言的限制，现在软件技术的新潮流是用 SOAP 作为传输介质的 Web Service，它可以使软件模块脱离平台和语言的束缚，实现更高程度的复用。但是想一想，其实 Web Service 也是面向对象，只不过是把类与类之间的关联用 XML 来描述而已 [Li, 02]。在过去的十多年里，面向对象技术对软件行业起到了极大的推动作用。在可以预测的将来，它仍将是软件设计的主要技术——至少我看不到有什么技术可以取代它的。

上面，我向读者介绍了一些背景知识，也稍微介绍了 OOD 的好处。下面，我将回答几个常见的问题，希望能借这几个问题让读者看清 OOD 的轮廓。

什么是 OOD？

面向对象设计（Object-Oriented Design，OOD）是一种软件设计方法，是一种工程化规范。这是毫无疑问的。按照 Bjarne Stroustrup 的说法，面向对象的编程范式（paradigm）是 [Stroustrup, 97]：

- 决定你要的类；
- 给每个类提供完整的一组操作；
- 明确地使用继承来表现共同点。

由这个定义，我们可以看出：OOD 就是“根据需求决定所需的类、类的操作以及类之间关联的过程”。

OOD 的目标是管理程序内部各部分的相互依赖。为了达到这个目标，OOD 要求将程序分成块，每个块的规模应该小到可以管理的程度，然后分别将各个块隐藏在接口（interface）的后面，让它们只通过接口相互交流。比如说，如果用 OOD 的方法来设计一个服务器-客户端（client-server）应用，那么服务器和客户端之间不应该有直接的依赖，而是应该让服务器的接口和客户端的接口相互依赖。

这种依赖关系的转换使得系统的各部分具有了可复用性。还是拿上面那个例子来说，客户端就不必依赖于特定的服务器，所以就可以复用到其他的环境下。如果要复用某一个程序块，只要实现必须的接口就行了。

OOD 是一种解决软件问题的设计范式（paradigm），一种抽象的范式。使用 OOD 这种设计范式，我们可以用对象（object）来表现问题领域（problem domain）的实体，每个对象都有相应的状态和行为。我们刚才说到：OOD 是一种抽象的范式。抽象可以分成很多层次，从非常概括的到非常特殊的都有，而对象可能处于任何一个抽象层次上。另外，彼此不同但又互有关联的对象可以共同构成抽象：只要这些对象之间有相似性，就可以把它们当成同一类的对象来处理。

OOD 到底从哪儿来？

有很多人都认为：OOD 是对结构化设计（Structured Design，SD）的扩展，其实这是不对的。OOD 的软件设计观念和 SD 完全不同。SD 注重的是数据结构和处理数据结构的过程。而在 OOD 中，过程和数据结构都被对象隐藏起来，两者几乎是互不相关的。不过，追根溯源，OOD 和 SD 有着非常深的渊源。

1967 年前后，OOD 和 SD 的概念几乎同时诞生，它们分别以不同的方式来表现数据结构和算法。当时，围绕着这两个概念，很多科学家写了大量的论文。其中，由 Dijkstra 和 Hoare 两人所写的一些论文讲到了“恰当的程序控制结构”这个话题，声称 goto 语句是有害的，应该用顺序、循环、分支这三种控制结构来构成整个程序流程。这些概念发展构成了**结构化程序设计方法**；而由 Ole-Johan Dahl 所写的另一些论文则主要讨论编程语言中的单位划分，其中的一种程序单位就是**类**，它已经拥有了面向对象程序设计的主要特征。

这两种概念立刻就分道扬镳了。在结构化这边的历史大家都很熟悉：NATO 会议采纳了 Dijkstra 的思想，整个软件产业都同意 goto 语句的确是有害的，结构化方法、瀑布模型从 70 年代开始大行其道。同时，无数的科学家和软件工程师也帮助结构化方法不断发展完善，其中有很多今天足以使我们振奋的名字，例如 Constantine、Yourdon、DeMarco 和 Dijkstra。有很长一段时间，整个世界都相信：结构化方法就是拯救软件工业的“银弹”。当然，时间最后证明了一切。

而此时，面向对象则在研究和教育领域缓慢发展。结构化程序设计几乎可以应用于任何编程语言之上，而面向对象程序设计则需要语言的支持¹⁰，这也妨碍了面向对象技术的发展。实际上，在 60 年代后期，支持面向对象特性的语言只有 Simula-67 这一种。到 70 年代，施乐帕洛阿尔托研究中心（PARC）的 Alan Kay 等人又发明了另一种基于面向对象方法的语言，那就是大名鼎鼎的 Smalltalk。但是，直到 80 年代中期，Smalltalk 和另外几种面向对象语言仍然只停留在实验室里。

到 90 年代，OOD 突然就风靡了整个软件行业，这绝对是软件开发史上的一次革命。不过，登高才能望远，新事物总是站在旧事物的基础之上的。70 年代和 80 年代的设计方法揭示出许多有价值的概念，谁都不能也不敢忽视它们，OOD 也一样。

OOD 和传统方法有什么区别？

还记得结构化设计方法吗？程序被划分成许多个模块，这些模块被组织成一个树型结构。这棵树的根就是主模块，叶子就是工具模块和最低级的功能模块。同时，这棵树也表示调用结构：每个模块都调用自己的直接下级模块，并被自己的直接上级模块调用。

那么，哪个模块负责收集应用程序最重要的那些策略？当然是最顶端的那些。在底下的那些模块只管实现最小的细节，最顶端的模块关心规模最大的问题。所以，在这个体系结构中越靠上，概念的抽象层次就越高，也越接近问题领域；体系结构中位置越低，概念就越接近细节，与问题领域的关系就越少，而与解决方案领域的关系就越多。

但是，由于上方的模块需要调用下方的模块，所以这些上方的模块就**依赖于**下方的细节。换句话说，**与问题领域相关的抽象要依赖于与问题领域无关的细节！**这也就是说，当实现细节发生变化时，抽象也会受到影响。而且，如果我们想复用某一个抽象的话，就必须把它依赖的细节都一起拖过去。

¹⁰ 我听见有人说“我可以用 C 语言实现面向对象”，不过我不希望你也会说“我喜欢这样做”。很明显，如果没有语言特性的支持，面向对象方法将寸步难行。其实结构化设计也是一样，不过几乎所有的编程语言都提供了对三种基本流程结构（顺序，条件，以及循环）的支持，所以基本不会遇到这个问题。

而在 OOD 中，我们希望倒转这种依赖关系：我们创建的抽象不依赖于任何细节，而细节则高度依赖于上面的抽象。这种依赖关系的倒转正是 OOD 和传统技术之间根本的差异，也正是 OOD 思想的精华所在。

OOD 能给我带来什么？

问这个问题的人，脑子里通常是在想“OOD 能解决所有的设计问题吗？”没有银弹。OOD 也不是解决一切设计问题、避免软件危机、捍卫世界和平……的银弹。OOD 只是一种技术。但是，它是一种优秀的技术，它可以很好地解决目前的大多数软件设计问题——当然，这要求设计者有足够的能力。

OOD 可能会让你头疼，因为要学会它、掌握它是很困难的；OOD 甚至会让你失望，因为它也并不成熟、并不完美。OOD 也会给你带来欣喜，它让你可以专注于设计，而不必操心那些细枝末节；OOD 也会使你成为一个更好的设计师，它能提供给你很好的工具，让你能开发出更坚固、更可维护、更可复用的软件。

C++ 是一种“真正的”面向对象编程语言吗？

最后这个问题和我们今天的主题关系不大，不过这是一个由来已久的问题，而且以后也不一定会有合适的时机说明这个问题，所以今天一起回答了。

很多人都觉得 C++ 缺少“真正的”面向对象语言所必须的一些特性，例如垃圾收集（garbage collection）、多分配（multiple-dispatch）之类的。但是，缺少这些特性并不影响开发者们用 C++ 实现面向对象的设计思路。

在我看来，任何语言，只要它直接支持面向对象设计的实现，那它就是“真正的”面向对象语言。用这个标准来评价，C++ 是完全符合的：它直接支持继承、多态、封装和抽象，而这些才是面向对象最重要的。而 VB 和 C 这样的语言不能直接支持这些（尽管可以用特殊的技巧来实现），所以不是“真正的”面向对象语言。

参考书目

- [Brooks, 95] Frederick Brooks, *The Mythical Man-Month*, Addison-Wesley, 1995.
- [Li, 02] 李维，《Delphi 6/Kylix 2 SOAP/Web Service 程序设计篇》，机械工业出版社，2002 年。
- [Martin, 95] Robert Martin, *Designing Object-Oriented C++ Applications Using the Booch Method*, Prentice-Hall, 1995.
- [Stroustrup, 97] Bjarne Stroustrup, *The C++ Programming Language (Special Edition)*, Addison-Wesley, 1997.

芝麻开门之 OpenGL

设置 OpenGL (二)

作者 : Jeff Molofee

翻译 : CKER

接下来的代码段创建我们的 OpenGL 窗口。我花了很多的时间来决定是否是创建一个全屏幕窗口（这样可以少写很多额外的代码），或者是创建一个易于定制且界面友好的窗口（这样需要写更多的代码）。最后我还是觉得用较多的代码来创建一个界面友好的窗口会是一个最佳的选择。因为我总是在 EMail 中收到诸如此类的问题：怎样只创建窗口而不是使用全屏幕？怎样改变窗口的标题栏？怎样改变窗口的分辨率或 pixel format（象素格式）？下面的代码已经实现了这一切！但您最好还要学点关于 material（材质）的知识，这样您在书写自己的 OpenGL 程序时，会感觉容易的多！

如您所见，下面这个函数的返回值是布尔变量（TRUE 或 FALSE）。它还带有 5 个参数，其意义分别是窗口的标题栏 title、宽度 width、高度 height，色彩深度 bits（16/24/32）和全屏标志 fullscreenflag（TRUE 全屏模式，FALSE 窗口模式）。返回的布尔值告诉我们窗口是否成功创建。

```
BOOL CreateGLWindow(char* title, int width, int height,
                     int bits, bool fullscreenflag)
{
```

当我们要求 Windows 为我们查找一个相匹配的象素格式时，查找结束后的结果将被保存在变量 PixelFormat 中。

```
GLuint PixelFormat; // 保存查找匹配
的结果
```

wc 被用来保存 Window Class 的结构。Window Class 结构中保存着窗体信息。通过改变该结构中不同的成员，我们就可以改变窗口的外观以及行为。每个窗口都属于一个 Window Class。在创建窗口之前，我们必须先为该窗口注册一个 Class。

```
WNDCLASS wc; // 窗体类结构
```

dwStyle 和 dwExStyle 被用来存储窗口的正常以及扩展情况下的风格信息。由于使用了变量来存储这些风格，我们就可以根据需要被创建窗口的类型（是全屏幕下的弹出窗体还是窗口模式下的带边框的普通窗体）来改变窗口的风格。

```
DWORD dwExStyle; // 扩展窗体风格
DWORD dwStyle; // 窗体风格
```

下面的 5 行代码可以用来取得矩形的左上角和右下角的坐标值。我们可以使用这些值来调整窗体使得其绘图区的大小恰好对应分辨率数值。通常在创建一个 640x480 的窗体时，窗体的边框会占掉一些分辨率数值。

```
RECT WindowRect; // 取得矩形左上角和右下角的坐标值
WindowRect.left=(long)0; // 将窗口左上角座标设置为 (0, 0)
WindowRect.right=(long)width;// 将窗口右下角座标设置为
                           // (width, height)
WindowRect.top=(long)0;
WindowRect.bottom=(long)height;
```

在下一行代码中，我们让全局变量 fullscreen 等于 fullscreenflag。如果我们希望让窗口全屏显示的话，变量 fullscreenflag 必须为 TRUE。如果我们没有让 fullscreen 等于 fullscreenflag 的话，变量 fullscreen 将仍保留其原来的值——FALSE。如果我们希望在全屏显示模式下销毁窗口时，由于变量 fullscreen 为 FALSE 而不是 TRUE，那么我们将无法切换回到桌面，因为计算机将误以为它已经处于桌面模式了。**我希望这能够引起大家的注意。**简单说来，fullscreen 必须和 fullscreenflag 相等，否则就会有问题出现。

```
fullscreen = fullscreenflag; // 设置全局全屏标志
```

在下一部分代码中，我们将取得窗体的实例句柄，然后定义 Window Class。

CS_HREDRAW 和 CS_VREDRAW 的意思是：无论何时，只要窗体发生变化时就强制其重画。CS_OWNDC 为窗体创建一个私有的 DC，意味着 DC 不能在程序间共享。WndProc 是程序的消息处理函数。由于没有使用额外的 Window 数据，后两个字段设为零。然后设置实例。因为不想给窗体来个图标，我们接着将 hIcon 设为了 NULL。鼠标指针被设为标准的箭头。背景色无所谓(在 GL 中设置)。此外，由于我们也不想在窗体中拥有一个菜单，所以也将其设为 NULL。类的名字可以您想要的任何名字。出于简单，我将使用“OpenGL”。

```
hInstance = GetModuleHandle(NULL); // 取得窗体实例
// 在窗体移动时重画之并为其增加一个私有的 DC
wc.style = CS_HREDRAW | CS_VREDRAW | CS_OWNDC;
wc.lpfnWndProc = (WNDPROC) WndProc; // 消息处理函数
wc.cbClsExtra = 0; // 没有额外的 Window 数据
wc.cbWndExtra = 0; // 没有额外的 Window 数据
wc.hInstance = hInstance; // 设置窗体实例
wc.hIcon = LoadIcon(NULL, IDI_WINLOGO); // 使用缺省的图标
wc.hCursor = LoadCursor(NULL, IDC_ARROW); // 使用箭头型鼠标
wc.hbrBackground = NULL; // 由于 GL 缘故，不设置背景色
wc.lpszMenuName = NULL; // 不需要菜单
wc.lpszClassName = "OpenGL"; // 设置 Class 的名字
```

现在我们要注册 Class 了。如果在注册过程中出错的话，将会弹出一个显示错误消息的对话框。点击 OK 按钮就可以退出当前运行的程序。

```
if (!RegisterClass(&wc)) // 尝试注册 Window Class
{
    MessageBox(NULL, "Failed To Register The Window
        Class.", "ERROR", MB_OK|MB_ICONEXCLAMATION);
    return FALSE; // 退出并返回 FALSE
}
```

现在我们开始检查程序是否应该运行在全屏模式下，还是在窗口模式下运行。如果应该是全屏模式的话，我们将尝试设置全屏模式。

```
if (fullscreen) // 要尝试全屏模式吗?
{
```

下一部分的代码关系到很多人都可能会关心的问题：如何切换到全屏模式。在切换到全屏模式时，有几件十分重要的事您必须牢记。必须确保您在全屏模式下所用的宽度和高度等同于窗口模式下的宽度和高度。最最重要的是要在创建窗口之前设置好全屏模式。这里的代码中，您无需再担心宽度和高度，它们已被设置成与显示模式所对应的大小。

```

DEVMODE dmScreenSettings; // 设备模式
memset(&dmScreenSettings, 0, sizeof(dmScreenSettings));
dmScreenSettings.dmSize = sizeof(dmScreenSettings);
dmScreenSettings.dmPelsWidth = width; // 所选屏幕宽度
dmScreenSettings.dmPelsHeight = height; // 所选屏幕高度
dmScreenSettings.dmBitsPerPel = bits; // 每象素所选的色彩深度
dmScreenSettings.dmFields =
    DM_BITSPERPEL | DM_PELSWIDTH | DM_PELSHEIGHT;

```

上面的代码分配了用于存储视频设置的空间，并为我们所需切换到的屏幕设定了的宽、高、以及色彩深度。在下面的代码中，我们将尝试设置全屏模式。dmScreenSettings 中保存了所有的宽、高、以及色彩深度讯息。在下一行中，ChangeDisplaySettings 被用来尝试将屏幕切换成与 dmScreenSettings 所匹配的模式。当模式被切换时，我们使用了参数 CDS_FULLSCREEN，因为这样做不仅移去了屏幕底部的状态条，而且它在来回切换时，也没有移动或改变您在桌面上的窗口。

```

// 尝试设置显示模式并返回结果。注：CDS_FULLSCREEN 移去了状态条。
if
(ChangeDisplaySettings(&dmScreenSettings, CDS_FULLSCREEN) != DISP_CHANGE_SUCCESSFUL)
{

```

如果不能设置模式的话，以下的代码将会被运行。如果不存在可以匹配的全屏模式，程序将弹出一个消息框，它提供了两个选项：在窗体模式下运行程序或退出程序。

```

// 若模式失败，提供两个选项：退出或在窗口内运行。
if ( MessageBox (NULL, "The Requested Fullscreen Mode Is
Not Supported By\nYour Video Card. Use Windowed Mode
Instead?", "NeHe GL", MB_YESNO | MB_ICONEXCLAMATION ) == IDYES )
{

```

如果用户选择使用窗体模式，变量 fullscreen 的值变为 FALSE，程序继续运行。

```

fullscreen=FALSE;
}
else
{ // 选择窗体模式(Fullscreen=FALSE)

```

如果用户选择退出，程序将弹出一个消息框用于告知用户程序将结束。并返回 FALSE 告诉程序窗体未能成功创建。然后程序将退出。

```

MessageBox( NULL, "Program Will Now Close.", "ERROR",
            MB_OK | MB_ICONSTOP );
return FALSE; // 退出并返回 FALSE
}
}
}

```

由于设置全屏模式可能失败，用户也可能决定在窗体下运行，所以在设置屏幕/窗口之前，我们需要再次检查 fullscreen 的值是否为 TRUE 或 FALSE。

```

if (fullscreen)
{ // 仍处于全屏模式吗？

```

如果程序仍处于全屏模式，我们就设置扩展窗体风格为 WS_EX_APPWINDOW，这将强制窗体在可见时可以处于最前面。再将窗体的风格设为 WS_POPUP。这个类型的窗体没有边框，使我们的全屏模式得以完美显示。

最后，我们在程序中禁用鼠标指针。当程序不需要交互时，在全屏模式下禁用鼠标指针通常是个好主意。当然，这都取决于你的想法。

```
dwExStyle=WS_EX_APPWINDOW;           // 扩展窗体风格
dwStyle=WS_POPUP;                   // 窗体风格
ShowCursor(FALSE);                 // 隐藏鼠标指针
}
else
{
```

如果我们使用窗口而不是全屏模式，我们就在扩展窗体风格中增加了 WS_EX_WINDOWEDGE，这将增强窗体的 3D 感观。窗体风格改用了 WS_OVERLAPPEDWINDOW 而不是 WS_POPUP，这将创建一个带标题栏、可变大小的边框、菜单和最大化/最小化按钮的窗体。

```
dwExStyle=WS_EX_APPWINDOW | WS_EX_WINDOWEDGE; // 扩展窗体风格
dwStyle=WS_OVERLAPPEDWINDOW;                  // 窗体风格
}
```

下一行代码将根据创建的窗体类型调整窗口。调整的目的是使得窗体大小正好等于所要求的分辨率。通常边框会占用窗体的一部分。使用 AdjustWindowRectEx 后，OpenGL 场景就不会再被边框盖住。实际上窗口变得更大以便绘制边框。在全屏模式下，此命令无效。

```
// 调整窗口达到真正要求的大小
AdjustWindowRectEx(&WindowRect, dwStyle, FALSE, dwExStyle);
```

在下一段代码中，我们开始创建窗口并检查窗口是否成功创建。我们将向 CreateWindowEx() 传递其所需的所有参数。如扩展风格、类名字(与您在注册窗体类时所用的名字相同)、窗体标题、窗体风格、窗体的左上角坐标(0,0 是个安全的选择)、窗体的宽和高。我们不想要一个父窗口，也不想要菜单，所以这些参数可以统统被设为 NULL。另外我们还传递了窗口的实例。最后，我们将最后一个参数设为 NULL。

注意我们在窗体风格中包括了 WS_CLIPSIBLINGS 和 WS_CLIPCHILDREN。要让 OpenGL 正常运行，这两个属性是必须的。它们阻止别的窗体在我们的窗体内/上绘图。

```
if ( !(hWnd=CreateWindowEx(
    dwExStyle,                      // 扩展窗体风格
    "OpenGL",                        // 类名字
    title,                           // 窗口标题
    WS_CLIPSIBLINGS |               // 必须的窗体风格属性
    WS_CLIPCHILDREN |              // 必须的窗体风格属性
    dwStyle,                          // 选择的窗体属性
    0, 0,                            // 窗口位置
    // 计算调整好的窗口宽度
    WindowRect.right-WindowRect.left,
    // 计算调整好的窗口高度
    WindowRect.bottom-WindowRect.top,
    NULL,                            // 无父窗口
```

```

NULL, // 无菜单
hInstance, // 实例
NULL)); // 不向 WM_CREATE 传递任何东东

```

下面我们来查看窗体是否正常创建。如果创建成功，hWnd 中将保存窗口的句柄。如果创建失败，程序将弹出出错信息并退出。

```

{
KillGLWindow();
// 重置显示区
MessageBox( NULL, "Window Creation Error.", "ERROR",
            MB_OK | MB_ICONEXCLAMATION);
Return FALSE; // 返回 FALSE
}

```

下面的代码描述像素格式。我们选择通过 RGBA (红、绿、蓝、alpha 通道) 来支持 OpenGL 和双缓存的格式。并试图找到匹配我们所选定的色彩深度 (16 位、 24 位、 32 位) 的像素格式。最后，我们设置了 16 位的 Z- 缓存。其余的参数要么未使用，要么就是不重要 (模板缓存 (stencil buffer) 和聚集缓存 (accumulation buffer) 除外) 。

```

static PIXELFORMATDESCRIPTOR pfd= //pfd 告诉窗口我们所希望的
东东
{
sizeof(PIXELFORMATDESCRIPTOR), //上诉格式描述符的大小
1, // 版本号
PFD_DRAW_TO_WINDOW | // 格式必须支持窗口
PFD_SUPPORT_OPENGL | // 格式必须支持 OpenGL
PFD_DOUBLEBUFFER, // 必须支持双缓冲
PFD_TYPE_RGBA, // 申请 RGBA 格式
bits, // 选定色彩深度
0, 0, 0, 0, 0, // 忽略的色彩位
0, // 无 Alpha 缓存
0, // 忽略 Shift Bit
0, // 无聚集缓存
0, 0, 0, 0, // 忽略聚集位
16, // 16 位 z- 缓存 ( 深度缓存 )
0, // 无模板缓存
0, // 无辅助缓存
PFD_MAIN_PLANE, // 主绘图层
0, // 保留
0, 0, 0 // 忽略层遮罩
};

```

如果前面在创建窗体时没有错误发生，我们将接着尝试取得 OpenGL 设备描述表。若我们无法取得 DC，将会弹出一个错误消息，然后程序退出(返回 FALSE)。

```

if (!(hDC=GetDC(hWnd))) //取得设备描述表了么？
{
KillGLWindow();
// 重置显示区
MessageBox( NULL, "Can't Create A GL Device
Context.", "ERROR",
            MB_OK | MB_ICONEXCLAMATION );

```

```

    return FALSE;           // 返回 FALSE
}

```

设法为 OpenGL 窗口取得设备描述表后，我们将尝试找到对应于此前我们所选定的像素格式的像素格式。如果 Windows 不能找到一个匹配的像素格式的话，程序将弹出一个错误消息，并退出(返回 FALSE)。

```

if (!(PixelFormat=ChoosePixelFormat(hDC,&pf))) // Windows 找到相应的像素格式了吗?
{
    KillGLWindow();           // 重置显示区
    MessageBox( NULL, "Can't Find A Suitable
PixelFormat.", "ERROR",
            MB_OK | MB_ICONEXCLAMATION );
    return FALSE;             // 返回 FALSE
}

```

如果 Windows 找到相应的像素格式，我们就将尝试设置像素格式。如果无法设置，程序将弹出一个错误消息，并退出(返回 FALSE)。

```

// 能够设置像素格式么?
if (!SetPixelFormat(hDC,PixelFormat,&pf))
{
    KillGLWindow();           // 重置显示区
    MessageBox( NULL, "Can't Set The PixelFormat.", "ERROR",
            MB_OK | MB_ICONEXCLAMATION );
    return FALSE;             // 返回 FALSE
}

```

正常设置像素格式后，我们将尝试取得着色描述表。如果不能取得着色描述表的话，程序将弹出一个错误消息，并退出(返回 FALSE)。

```

if (!(hRC=wglCreateContext(hDC))) // 能否取得着色描述表?
{
    KillGLWindow();           // 重置显示区
    MessageBox( NULL, "Can't Create A GL Rendering
Context.", "ERROR",
            MB_OK | MB_ICONEXCLAMATION );
    return FALSE;             // 返回 FALSE
}

```

如果到现在仍未出现错误的话，那么我们已经设法取得了设备描述表和着色描述表。接着要做的是激活着色描述表。如果无法激活，程序将弹出一个错误消息，并退出(返回 FALSE)。

```

if (!wglGetCurrent(hDC,hRC)) // 尝试激活着色描述表
{
    KillGLWindow();           // 重置显示区
    MessageBox( NULL, "Can't Activate The GL Rendering
Context.", "ERROR", MB_OK | MB_ICONEXCLAMATION );
    return FALSE;             // 返回 FALSE
}

```

一切顺利的话，我们的 OpenGL 窗口已经创建完成，接着就可以显示它啦。将它设为前端窗口(给它更高的优先级)，并将焦点移至此窗口。然后调用 ReSizeGLScene 将屏幕的宽度和高度设置给透视 OpenGL 屏幕。

```
ShowWindow(hWnd, SW_SHOW);           // 显示窗口
SetForegroundWindow(hWnd);           // 略略提高优先级
SetFocus(hWnd);                     // 设置键盘的焦点至此窗口
ReSizeGLScene(width, height);       // 设置透视 GL 屏幕
```

最后就到了 InitGL()，在这里我们可以设置光照、纹理、等等任何需要设置的东东。您可以在 InitGL() 内部自行定义错误检查，并返回 TRUE(一切正常)或 FALSE(有什么不对)。例如，如果您在 InitGL() 内装载纹理并出现错误，您可能希望程序停止。如果您返回 FALSE 的话，下面的代码会弹出错误消息，并退出程序。

```
if (!InitGL())                      // 初始化新建的 GL 窗口
{
    KillGLWindow();                 // 重置显示区
    MessageBox(NULL, "Initialization
Failed.", "ERROR", MB_OK | MB_ICONEXCLAMATION);
    Return FALSE;                  // 返回 FALSE
}
```

到这里，我们就可以安全的推定窗口已经被成功创建了。我们向 WinMain() 返回 TRUE，告知它程序没有错误，以防止程序退出。

```
return TRUE;                         // 成功
}
```

下面的代码中将处理所有的窗口消息。当我们注册好窗口类之后，我们就告知程序跳转到这部分代码中来处理窗口消息。

```
LRESULT CALLBACK WndProc( HWND hWnd,   // 窗口的句柄
    UINT uMsg,                      // 窗口的消息
    WPARAM wParam,                 // 附加的消息内容
    LPARAM lParam)                 // 附加的消息内容

{
```

下来的代码比对 uMsg 的值，然后转入 case 处理，uMsg 中保存了我们要处理的消息名字。

```
switch (uMsg)
{
    // 检查 Windows 消息
```

如果 uMsg 等于 WM_ACTIVE，我们将检查窗口是否仍然处于激活状态。如果窗口已被最小化，将变量 active 设为 FALSE。如果窗口已被激活，将变量 active 的值设为 TRUE。

```
case WM_ACTIVATE:                // 监视窗口激活消息
{
    if (!HIWORD(wParam))         // 检查最小化状态
    {
        active=TRUE;             // 程序处于激活状态
    }
}
```

```

    }
else
{
active=FALSE;           // 程序不再激活

}
return 0;               // 返回消息循环

}

```

如果消息是 WM_SYSCOMMAND(系统命令) , 再次比对 wParam。如果 wParam 是 SC_SCREENSAVE 或 SC_MONITORPOWER 的话 , 不是有屏幕保护程序要运行 , 就是显示器想进入节电模式。通过返回 0 , 我们就可以阻止这两件事发生。

```

case WM_SYSCOMMAND:          // 中断系统命令
{
switch (wParam)              // 检查系统调用 Check System
Calls
{
case SC_SCREENSAVE:          // 屏保要运行?
case SC_MONITORPOWER:        // 显示器要进入节电模式?
return 0;                     // 阻止发生
}
break;                        // 退出
}

```

如果 uMsg 是 WM_CLOSE , 窗体将被关闭。我们发出退出消息 , 主循环将被中断。变量 done 将被设为 TRUE , WinMain()的主循环将会中止 , 然后程序结束运行。

```

case WM_CLOSE:                // 收到 Close 消息?
{
PostQuitMessage(0);           // 发出退出消息
return 0;
}

```

如果键盘有键按下 , 通过读取 wParam 的信息我们就可以找出键值。然后 , 我就将键盘数组 keys[]中相应的数组成员的值设为 TRUE。这样以后就可以查找 key[]来得知什么键被按下过。这样我们就允许多个键被同时按下的情况出现。

```

case WM_KEYDOWN:             // 有键按下么?
{
keys[wParam] = TRUE;          // 如果是 , 设为 TRUE

return 0;                      // 返回
}

```

同样 , 如果键盘有键释放 , 通过读取 wParam 的信息也可以找出该键的键值。然后将键盘数组 keys[]在相应的数组组成员的值设为 FALSE。这样就可以通过查找 key[]来得知什么键被按下 , 什么键被释放了。键盘上的每个键都可以用 0-255 之间的一个数来代表。举例来说 , 当我们按下 40 所代表的键时 , keys[40]的值将被设为 TRUE。放开的话 , 它就被设为 FALSE。这也就是我们如何使用 key 数组来记录按键操作的步骤。

```

case WM_KEYUP:                // 有键放开么?
{

```

```

keys[wParam] = FALSE;           // 如果是，设为 FALSE

return 0;                      // 返回

}

```

当调整窗口大小时，uMsg 最终将等于消息 WM_SIZE。通过读取 lParam 的 LOWORD 和 HIWORD，我们就可以得到窗口新的宽度和高度。将它们传递给 ReSizeGLScene()，OpenGL 场景将调整为新的宽度和高度。

```

case WM_SIZE: // 调整 OpenGL 窗口大小
{
    // LoWord=Width, HiWord=Height
    ReSizeGLScene(LOWORD(lParam),HIWORD(lParam));

    return 0; // 返回

}
}

```

其余的我们所不关心的消息将被传递给 DefWindowProc 由 Windows 自行处理。

```

// 向 DefWindowProc 传递所有未处理的消息。
return DefWindowProc(hWnd,uMsg,wParam,lParam);
}

```

下面是我们的 Windows 程序的入口。我们在此处调用我们的窗口创建例程，处理窗口消息，并监视人机交互。

```

int WINAPI WinMain( HINSTANCE hInstance,      // 实例
                     HINSTANCE hPrevInstance, // 前一个实例
                     LPSTR lpCmdLine,       // 命令行参数
                     int nCmdShow )         // 窗口显示状态
{

```

我们设置两个变量。msg 用来检查是否有消息等待处理。done 的初始值设为 FALSE。这意味着我们的程序仍未完成运行。只要 done 为 FALSE，程序就将继续运行。一旦 done 的值改变为 TRUE，程序就将退出。

```

MSG msg;                         // Windowsx 消息结构
BOOL done=FALSE;                // 用来退出循环的 Bool 变量

```

这段代码完全可选。程序弹出一个消息框，询问用户是否希望在全屏模式下运行。如果用户点击了 NO 按钮，fullscreen 变量将从缺省的 TRUE 改变为 FALSE，程序也改而在窗口模式下运行。

```

// 提示用户选择运行模式
if (MessageBox(NULL,"Would You Like To Run In Fullscreen
Mode?", "Start
FullScreen?", MB_YESNO | MB_ICONQUESTION)==IDNO)
{
    fullscreen=FALSE; // 窗口模式
}

```

接着创建 OpenGL 窗口。CreateGLWindow 函数的参数依次为标题、宽度、高度、色彩深度，以及全屏标志（TRUE 代表全屏模式，FALSE 代表窗口模式）。就这么简单！我很欣赏这段代码的简洁。如果窗口未能被成功创建，函数返回 FALSE。程序立即退出（并返回 0）。

```
// 创建 OpenGL 窗口
if (!CreateGLWindow( "NeHe's OpenGL Framework",
                     640, 480, 16, fullscreen))
{
    return 0; // 失败退出
}
```

下面是循环的开始。只要 done 保持为 FALSE，循环就将一直进行。

```
while (!done) // 保持循环直到 done=TRUE
{
```

我们要做的第一件事是检查是否有消息在等待。使用 PeekMessage()可以在不阻塞我们程序运行的前提下对消息进行检查。许多程序使用 GetMessage()，这也很好。但使用 GetMessage()，程序在收到 paint 消息或其他别的什么窗口消息之前不会做任何事。

```
if (PeekMessage(&msg, NULL, 0, 0, PM_REMOVE))
{ // 有消息在等待吗？
```

在下面的代码中，我们查看程序中是否出现了退出消息。如果当前的消息是由 PostQuitMessage(0) 引起的 WM_QUIT，done 变量被设为 TRUE，程序将退出。

```
if (msg.message==WM_QUIT) // 收到退出消息？
{
    done=TRUE; // 是，则 done=TRUE
}
else // 不是，处理窗口消息
{
```

如果消息不是退出消息，我们将解释该消息，然后将它分发出去，以使得 WndProc() 或 Windows 能够处理它。

```
TranslateMessage(&msg); // 翻译消息
DispatchMessage(&msg); // 发送消息
}
}
else // 如果没有消息
{
```

如果没有消息，我们将绘制我们的 OpenGL 场景。下面代码中的第一行将查看窗体是否处于激活状态。接着场景被着色，DrawGLScene() 的返回值也将被检查。如果 DrawGLScene() 返回 0 或者是按下 ESC 键，done 变量被设为 TRUE，程序将会退出。

```
// 绘制场景。监视 ESC 键和来自 DrawGLScene() 的退出消息
if (active && !DrawGLScene()) // 仅在活动窗体中着色
{
    if (keys[VK_ESCAPE]) // ESC 按下了么？
{
```

```

        done=TRUE;           // ESC 发出退出信号
    }
    else                  // 不是退出的时候，刷新屏幕
    {

```

如果所有的一切都运行正常的话，我们就交换缓存（通过使用双缓存可以实现无闪烁的动画）。通过使用双缓存，我们实际上是在另一个看不见的“屏幕”上绘图。当我们交换缓存后，我们当前的屏幕被隐藏，现在看到的是刚才看不到的屏幕。这也是我们看不到场景绘制过程的原因。场景只是即时显示。

```

        SwapBuffers(hDC);      // 交换缓存（双缓存）
    }
}

```

下面的一点代码是最近新加的(05-01-00)。它允许用户按下 F1 键在全屏模式和窗口模式间切换。

```

if (keys[VK_F1])          // F1 键按下了么？
{
    keys[VK_F1]=FALSE;     // 若是，使对应的 Key 数组中的值为
    FALSE

    KillGLWindow();        // 销毁当前的窗体

    fullscreen=!fullscreen; // 切换 全屏 / 窗体 模式

    // 重建 OpenGL 窗口
    if (!CreateGLWindow("NeHe's OpenGL Framework",
                        640,480,16,fullscreen))
    {
        return 0; // 如果窗体未能创建，程序退出
    }
}
}
}
}

```

如果 done 变量不再是 FALSE，程序退出。正常销毁 OpenGL 窗体，将所有的内存释放，退出程序。

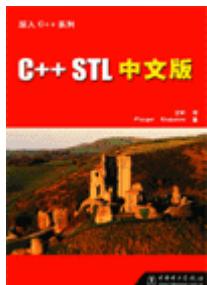
```

// 关闭程序
KillGLWindow();          // 销毁窗口
return (msg.wParam);     // 退出程序
}

```

在这一课中，我已试着尽量详细解释一切。每一步都与设置有关，并创建了一个全屏 OpenGL 程序。当您按下 ESC 键程序就会退出，并监视窗口是否激活。我花了大概 2 周时间来写代码，一周时间来改正 BUG 以及和其他编程高手讨论，2 天（大概 22 小时）来写这个 HTML 文件。如果您有什么意见或建议请给我 EMAIL。如果您认为有什么不对或者代码中还存在什么地方可以改进，请告诉我。我期待能够做出最好的 OpenGL 教程，因此您的反馈对我来说很重要。

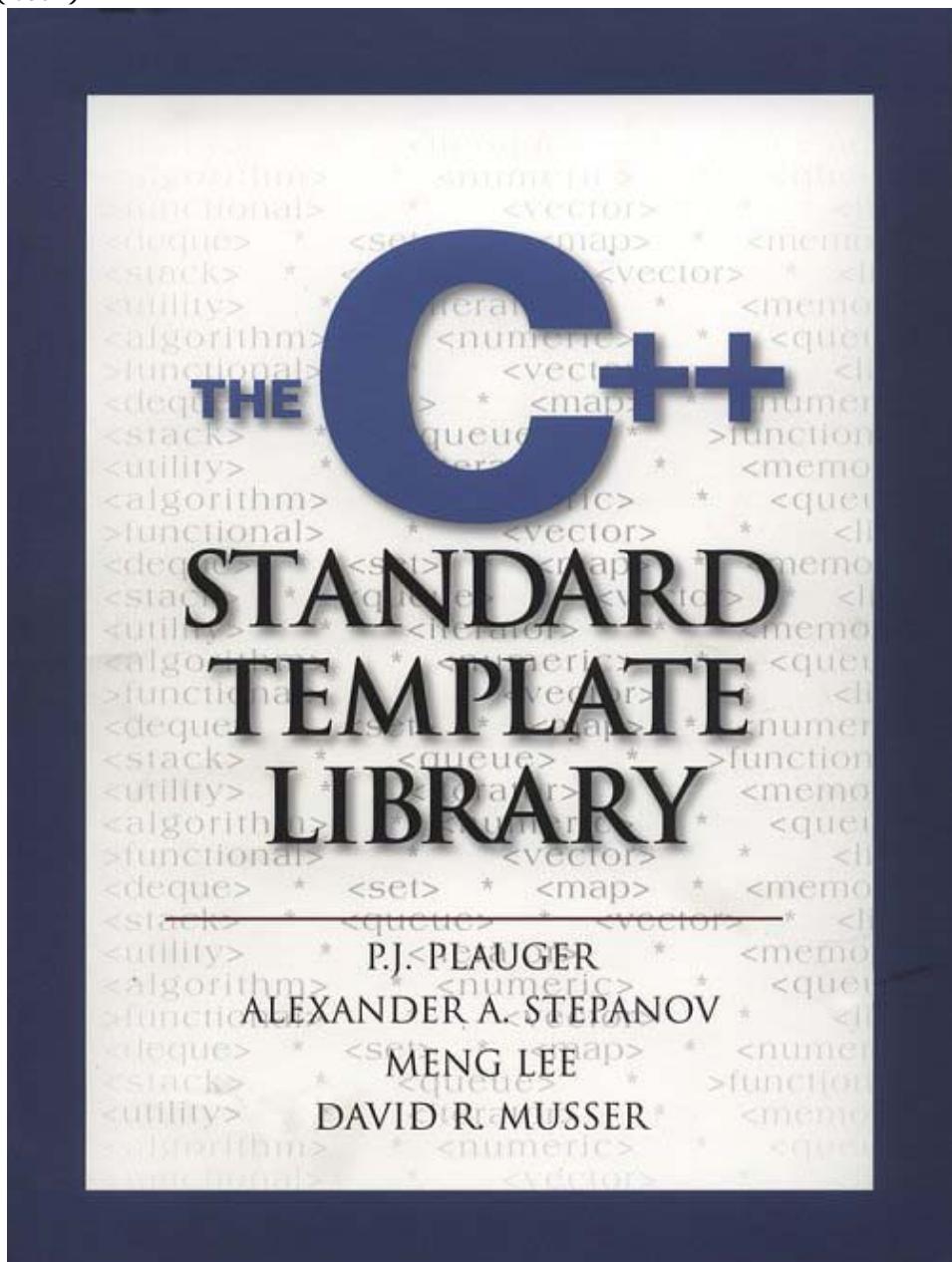
《C++ STL（中文版）》书稿试读



感谢电力出版社提供该书的第一章、第五章以及附录 A 的书稿于本刊发表。希望它能够使大家对于 STL 的学习起到帮助。

作为译者，我想在此透露一个小秘密：那就是本书的习题十分之具有价值，如果您能够把它们全部做出来的话，那您就可以称得上是“精通”STL 了（最起码可以在我面前这么说）。在本期的“鸟鸣涧”栏目中，我们给大家安排的就是第一章的习题，希望大家能够踊跃地向 cppview@sohu.com 发表您对于这些题目的看法，希望能够从大家的想法中获得一些启发及问题的解决之道，最终使得大家共同进步。

王昕 (cber)



第1章 迭代器

背景知识

迭代器在STL中起着粘合剂的作用，用来将STL的各个部分结合在一起。从本质上来说，STL提供的所有算法都是模板，我们可以通过使用自己指定的迭代器来对这些模板进行特化。我们也建议大家按照这种格式写出自己的算法模板。同样，在标准库中所提供的所有容器也都提供了这样的迭代器，用以存取它们所管理的数据序列。同样也建议大家在自己定义的容器中提供与之相配的迭代器。由于对象指针可以被当作任意种类的迭代器使用，所以这不会给你带来许多限制。

<utility> 在此，我们主要以<utility>、<iterator>和<memory>这三个头文件为线索展开讨论。这三个头文件中的第一个能提供的信息远比迭代器技术要多。
<iterator> 但是，对这三个头文件的使用一直贯穿整个STL，它们也用来提供实现
<memory> STL所必需的机制。在接下来的三章中，我们将分别对它们进行更详细的讲解。在此先提醒大家一下，对于这些细节方面的讨论有时十分单调乏味，而且在开始讨论算法和容器前，我们几乎很难察觉到这些细节的好处。从现在起，我们仅仅关心迭代器所共有的一些属性，像如何将它们分类以及不同种类的特点是什么等。

功能描述

C++中的迭代器相对于C中的对象指针来说更加一般化。指针本身就可以作为定义好的迭代器来使用。这种一般化行为主要体现在可以在C++中声明新类，然后对于这些类中的大部分操作符进行重载（overload），赋予它们新的意义。甚至还可以让迭代器指向一个奇异值（singular value），该迭代器的行为与一个定义有问题的指针十分相似。

输出迭代器 可使用迭代器来存取有序序列中的元素。在此，“存取（access）”是一个一般化的术语，我们既用它来表示将值存储到对象中，也用它来表示从一个对象中获得它所保存的值。如果需要创建一个新的值序列，并且以有序的方式来产生值，可以写一个如下的循环语句：

```
for ( ; <not done>; ++next)
    *next = <whatever>;
56
```

在此，`next`表示一个迭代器类型为 X 的对象，`<not done>`是一个用来检测循环是否应该终止的谓词（predicate），而`<whatever>`则是一个表达式，类型为序列中元素的类型 T，最起码也是那种很容易转化为 T 的类型。

表 1-1：
输出迭代器
的属性

表达式	结果的类型	含 义	注 释
<code>X(a)</code>	<code>X</code>	产生一个 a 的拷贝	析构函数是可见的 <code>*X(a) = t</code> 与 <code>*a = t</code> 的作用相同
<code>X u(a) X u = a</code>	<code>X&</code>	<code>u</code> 是 a 的拷贝	
<code>r = a</code>	<code>X&</code>	将 a 赋值给 r	结果 <code>*r = t</code> 和 <code>*a = t</code> 作用相同
<code>*a = t</code>	<code>void</code>	在序列中存储新元素	
<code>++r</code>	<code>X&</code>	指向下一个元素	<code>&r == &++r</code>
<code>r++</code>	可以转换为 <code>const X&</code>	{ <code>X tmp = r;</code> <code>++r;</code> <code>return tmp;</code> }	
<code>*r++ = t</code>	<code>void</code>		
注释：X 是迭代器类型；a 的类型为 X&；T 是元素类型；t 的类型为 T			

具有这样的类型 X，可以这样使用的迭代器，称为输出迭代器（output iterator）。从上面的讨论来看，一个输出迭代器至少还需要定义下面这些操作：

- `*next = <whatever>` 将`<whatever>`的值赋给序列中将要产生的下一个元素。

在 C 中经常使用一种稍有不同的方法来完成同样的操作：

```
while (<not done>)
    *next++ = <whatever>;
```

为了支持这种写法，表达式`*next++ = <whatever>`必须将上面所描述的两种操作结合起来，使之具有和 C 中的指针一样的意义。

实际上，一个输出迭代器所能保证的一点也不比上面列出来的属性多（甚至不能判断一个输出迭代器是否前进得太多）。围绕着输出迭代器的一系列操作包括一个复制构造函数、一个析构函数、一个赋值操作符，它们都具有通常所需的属性。与这些可怜的属性相对应的是，输出迭代器允许以各种各样的方式来实现它们。甚至可以将一个合适的输出迭代器装扮成每次存储一个输出记录的形式，使之输出到一个文件中。

表1-1以接近数学中常用符号的形式形象地描述了输出迭代器的各种属性。对于其他种类的迭代器，本书也将提供同样形式的表格。我们发现，使用这样的符号记录方式对于示例代码及上面的注释来说能起很大的帮助，但也没有必要使用它来完全替换掉那些代码及注释。在某些对于输出迭代器来说非常重要的限制条件中，这样的表格并不能完全清楚地表达它们。如：

- 必须保证输出迭代器在存储每一个元素后得以增加。
- 必须保证输出迭代器在两次存储的间隔内增加的次数不会超过一次。

如果输出迭代器实际上就是指针的话，上面的限制就不明显了。但当你看到STL使用一些具有特殊目的的输出迭代器的技巧时，就会知道对于这些限制的需求了。你可能会更欣赏表1-1中一些相对深奥的条目，如有关r++的返回值类型等。但从现在起，只需要记住，输出迭代器只是在类似上面给出的循环语句中使用。

输入迭代器

输入迭代器（input iterator）用来产生新的序列。为了顺序存取已有的值，或只是需要对已有的序列进行遍历，可以这样做（和前面的方法略有不同）：

```
for (p = first; p != last; ++p)
    <process>(*p);
```

在这里，p、first和last都是迭代器类型X的对象，<process>是一个函数，它能够接受元素类型为T的参数。对于上面出现的由元素组成的序列，我们用一个半封闭区间[first, last)来表示，其中，first和last各代表一个迭代器。

注意，last其实并不表示序列中的任何一个元素。实际上，它表示的是“end-of-sequence”标记，如在实际的序列末端紧接着的第一个元素。空序列是满足first == last的序列。C标准中也经常使用这种方法来操作C中的数组——可以将指向数组末端后的第一个元素的地址存储在指针中，但不能得到这个指针所指向的元素的值。迭代器一般化了这种概念，使其更加完美。

具有这样的类型X，可以这样使用的迭代器称为输入迭代器（input iterator）。从上面的讨论来看，输入迭代器至少还需要定义下面这些操作：

- 当两个类型为X的迭代器p和q没有同时指向一个元素时，p != q就为真（为方便起见，p == q通常都被定义成与p != q在逻辑上相反）。
- *p是类型T的一个右值（rvalue）。（右值是一个拥有值的表达式如-37，但它并不一定必须用来引用一个对象。）表达式p == last没有定义。
- ++p将改变p的值，将它指向序列中目前所指向元素的下一个元素。表达式p == last没有定义。

在C中经常使用一种稍有不同的方法来完成同样的操作：

```
while (first != last)
    <process>(*first++);

```

表 1-2：
输入迭代器
的属性

表达式	结果的类型	含 义	注 释
X(a)	X	产生a 的一个拷贝	析构函数是可见的， * X (a) 和 * a作用 相同
X u (a) X u = a	X&	u 是a 的一个拷贝	创建完成后 u == a
r=a	X&	将 a 赋值给 r	结果: r = t
a == b	可以转换为bool	相等比较	a和b在同一值域内
a != b	可以转换为bool	!(a == b)	
*a	T	从序列中存取元素	不是 end-of-sequence
a->m	m的类型	(*a).m	T有成员m
++r	X&	指向下一个元素	&r == &++r, r不是 end-of-sequence, r的拷贝无效
(void)r++	void	(void)++r	
*r++	T	{T tmp = *r; ++r; return tmp; }	
注释：X是迭代器类型；a和b 的类型为X&；r 的类型为X&；T是元素类型； t 的类型为T			

为了支持这种写法，表达式*first++必须将上面所描述的两种操作结合起来，使之具有和C中的指针一样的意义。

从这些属性我们可以得知，只有当可以从first到达last时，[first, last)才是一个有限的序列。换另一种说法就是，first的值在经过有限次的增加后必定会得到一个确定的值，它与last的值是相等的。

与其兄弟输出迭代器一样输入迭代器所保证的也不比上面的这些属性多。同样围绕着输入迭代器的一系列操作包括一个复制构造函数、一个析构函数和一个赋值操作符，它们都带有通常所需的一些属性。这其中唯一新增的就是指针指向操作符p->m，只有在类型 T 是一个结构化类型时，它才被定义。同样，输入迭代器也允许有各种各样的实现。

end-of-
sequence
值

你甚至可以将一个合适的输入迭代器装扮成每次存取一个输入记录的形式，从一个文件中获得所需要的输入。为了能使用这种技巧，输入迭代器定义了一个end-of-sequence值，它在大多数情况下是一种end-of-file标记。

这个end-of-sequence值存储在last中。用来指向一个真正记录的迭代器first不会等于last。然而，当first指向的记录是序列中的最后一个元素时，再增加first的值将导致其值发生改变，变为与last相等。也就是说，我们可以以同样的控制方式来完成这两件事情：通过增加指针的值来遍历一个数组与通过增加输入迭代器的值来遍历一个文件。

表1-2以接近数学中常用的符号形式形象地描述了输入迭代器的各种属性。我们再次重申一遍，这样的表并不能将所有有关输入迭代器的重要限制都讲述清楚。例如，它没有说，只有当两个迭代器中至少有一个具有end-of-sequence值时，它们之间的比较才一定有意义。（当使用任意类型的输入迭代器时，对于序列中的两个明显不同的地方，你甚至不能很明确地说出序列是否被越界存取。）建议在处理输入迭代器时遵循我们在输出迭代器处所提出的建议——只在类似上面给出的循环语句中使用它们。

前向 迭代器

从我们先前列举的一些理由来看，输出迭代器和输入迭代器可以说“足智多谋”。它们可以用来处理任意长度的文件。然而，迭代器的一个更加普遍的用法是：存取一个完全存储在内存中的序列。在这种情况下，所需要使用的迭代器就必须具有较少让人觉得惊奇的属性。

举例来说，你仍然只是满足于从头至尾地存取一个序列中的所有元素。但这次你想要对于该序列中的元素同时具有读和写的权利，或者想在先前所存取的任意地方标注出一个位置作为“书签”以方便下次存取。在这种情况下，仍然使用我们在讲解输出迭代器和输入迭代器时所用的那个控制循环，你所需要的很简单，那就是让迭代器看起来更像是一个传统的指针。

运用前向迭代器（forward iterator）就可以达到所有的这些要求。和输入迭代器一样，可以比较两个前向迭代器是否相等，但现在它们可以都为（或都不为）end-of-sequence。当然，和先前所讨论的一样，这两个迭代器的值必须处于同一个值域中。和输入迭代器一样，前向迭代器也可以有end-of-sequence值，它的意义和数组中“off the end”元素的地址差不多，你也可以将它想像成其他任意的end-of-sequence标记。你还可以用一个前向迭代器的多个有效拷贝来指向当前序列中的任意位置。

可以把前向迭代器想像成为一个指向单向链表中元素的指针。你可以明确地指出它是否指向链表的末端（用null来标记end-of-sequence）。如果它指向的元素不是end-of-sequence，就可以通过它来存取该链表的元素，或是把它移到序列中下一个元素的位置。但是不能让它回退，也不能通过它来直接存取链表中的任意元素。

严格地讲，你“可以”通过使用前向迭代器来完成这些操作中的一部

即数组中最后的那个元素后面所紧接着的那个元素。一般来说，它并不具有实际意义，通常我们用它的地址来判断数组是否结束。——译者注 60

分。但这具有一定的欺骗性，不管链表的长度如何，你都不可能在恒定的时间内完成这些操作。对于所有的迭代器来说，都存在着一个隐式要求：我们对于迭代器的所有操作都不能有太大的开销。不能随着迭代器所指向序列的长度改变而改变操作所需花费的时间。

表1-3以接近数学中常用符号的形式形象地描述了前向迭代器的各种属性。

表1-3：
前向迭代器
的属性

表达式	结果的类型	含 义	注 释
X()	X	产生一个默认值	析构函数是可见的 ,值可以是end-of-sequence
X u X u = a	X&	U具有默认值	
X (a)	X	产生a 的一个拷贝	析构函数是可见的 , *x(a)和*a 的作用相同
X u(a) X u = a	X&	u是a 的一个拷贝	创建完成后u == a
r = a	X&	将a 赋值给r	结果: r == a
A == b	可转换为bool	相等比较	a和b在同一值域中
A != b	可转换为bool	!(a == b)	
*a	T&	从序列中存取元素	a不是end-of-sequence, a == b 意味着 *a == *b
*a = t	T&	在元素中存储	a不是end-of-sequence, x 是可变的
a->m	m的类型	(*a).m	T有成员m
++r	X&	指向下一元素	&r == &++r, r 不是 end-of-sequence, r == s 意味着 ++r == ++s
R++	可转换为常量x&	{ x tmp = r; ++r; return tmp; }	
*r++	T&	{ T tmp = *r; ++r; return tmp; }	
注释：x是迭代器类型；a和b的类型为x；r和s的类型为x&；T是元素类型，t 的类型为T			

双向
迭代器

比前向迭代器应用更为广泛的一种迭代器同时支持递增及递减操作。
通过使用这种迭代器的这些特性，许多算法都得以以更加高效的方式实现。

STL所定义的双向迭代器 (bidirectional iterator) 就是这样的一种迭代器，与前向迭代器相比，它多了可以在序列中逆向移动这种特性。

我们可以将双向迭代器想像成指向一个双向链表中的元素的指针。我们可以明确地指出它是否指向该链表的末端（用 null 来标记 end-of-sequence）。如果迭代器指向的不是链表的末端，我们就可以通过它来存取该链表中的元素，或是将它移到序列中下一个元素的位置。如果迭代器指向的不是链表中的第一个元素，我们就可以把它回退到序列中前一个元素的位置。但是我们不可能通过它直接存取链表中任意位置的元素，至少我们不可能通过恒定时间的操作来达到这个目的。

表1-4以类似于常用数学符号的形式描述了双向迭代器的各种附加属性。所有适用于前向迭代器的属性，也同样适用于双向迭代器。

表 1-4：
双向迭代器的
附加属性

表达式	结果的类型	含 义	注 释
--r	X&	指向下一个元素	对于一些 s , ++s = r , &r == &--r, r 不是 end-of-sequence, r == s 意味着 --r == --s
r--	可转换为常量x&	{ x tmp = r; --r; return tmp; }	
*r--	T&	{ T tmp = *r; --r; return tmp; }	
注释：x是迭代器类型；r和s的类型为x&；T是元素类型；所有其他属性和前向迭代器相同			

随机存取 迭代器

我们所讨论的最后一种迭代器具有C语言中对象指针的所有强大功能。除了在双向迭代器中所提到的那些特性外，随机存取迭代器还支持与整型值的加减操作、指针之间的相减、两个迭代器在序列中的顺序比较，以及使用下标方式操作该迭代器等。某些算法只能依靠这种程度的弹性才有可能运作良好。（通过二分法来对序列进行排序和快速查找就是这样的两个例子。）

然而，请记住，随机存取迭代器仍然不是C语言风格的指针。例如，它们之间存在的一个区别是某种类型Dist，它可以是也可以不是一种最基本的整数类型。我们可以将作用于整型值的算法应用于Dist对象，但这并不能够阻止我们将Dist定义为一个类。

表1-5以类似于常用数学符号的形式描述了随机存取迭代器的各种附加属性（相对于双向迭代器来说）。所有适用于双向迭代器的属性同

样也适用于随机存取迭代器。

表 1-5：
随机存取迭代器的附加属性

表达式	返回值类型	含 义	注 释
$a < b$	可转换为bool的任意类型	从a可以到达b	a和b处于同一个值域中
$a > b$	可转换为bool的任意类型	$b < a$	
$a \leq b$	可转换为bool的任意类型	$!(b < a)$	
$a \geq b$	可转换为bool的任意类型	$!(a < b)$	
$r += n$	X&	{ Dist m = n; while (0 < m) --m, ++r; while (m < 0) ++m, --r; return r; }	
$a + n$ $n + a$	X	{ X tmp = a; tmp += n; return tmp; }	
$r -= n$	X&	$r += -n$	
$a - n$	X	$a + (-n)$	
$b - a$	Dist	{ Dist m = 0; while (a < b) ++a, ++m; while (b < a) ++b, --m; return m; }	a和b处于同一个值域中
$a[n]$	可转换为T的任意类型	$*(a + n)$	

注释：x是迭代器类型；a 和 b 的类型为X；r 和 s 的类型为X&；T是序列中元素的类型；Dist是类型X 的差距类型；其他属性和双向迭代器中讨论的一样。

使用迭代器

STL中大量使用了迭代器，它们用于不同的算法和算法所作用的序列之间，起着桥梁的作用。为了本书中其他章节的简洁起见，我们使用迭代器类型的名字（或前缀）来代指迭代器的种类。为了提升其功能，我们把不同的迭代器总结为以下几类：

- OutIt — 假设X为一个输出迭代器，那么它只能通过存储来间接地拥有一个值V。我们在向输出迭代器中存储了一个值之后，必须将其递增。如：`(*X++ = V)`、`(*X = V, ++X)`，或者是`(*X = V, X++)`。
- InIt — 假设X为一个输入迭代器，那么它的值也可以为end-of-sequence。如果它不等于end-of-sequence的话，我们就可以通过间接的方式来存取它所拥有的值V，如：`(V = *X)`。如果想要取得序列中的下一个元素的值（或是end-of-sequence），必须将其递增，如：`++X`、`X++`、或者是`(V = *X++)`。一旦我们对一个输入迭代器进行了递增，它的所有其他拷贝就不保证一定能够完成下面的操作：比较、间接取值、或者是再对其进行递增等。

- FwdIt — 假设X是一个前向迭代器，如果*X是可变的，那么它就可以用来替换输出迭代器（因为这时它本身就是一个输出迭代器）。同样，它也可以用来替换（或者就是）一个输入迭代器。然而，也可以通过一个前向迭代器来读取它所拥有的值（通过使用`V = *X`），而这个值就是你刚刚存储到它之中的那个值（通过使用`*X = V`）。你可以同时拥有一个前向迭代器的多份拷贝，它们中的每一份都可以用来间接取值，或是各自进行递增。

- BidIt — 假设X是一个双向迭代器，那么我们可以用它来替换一个前向迭代器。并且，你还可以对双向迭代器进行递减，如：`--X`、`X--`、或者是`(V = *X--)`。

- RanIt — 假设X和Y都是随机存取迭代器，那么我们就可以用它们来替换一个双向迭代器（因为它们本来就是一个双向迭代器）。我们同样也可以对随机存取迭代器进行许多整数运算（这一点和对象指针一样）。如果N为一个整型对象，那么我们就可以这样写：`X[N]`、`X < Y`、`X - N`、`N + X`等。

注意，对象指针可以用来替换一个随机存取迭代器（或者说它本来就是随机存取迭代器）。

迭代器的种类

迭代器分类的层次可以总结为下面的三种情况。如果只需对序列进行只写（write-only）操作，我们可以选择以下的任意一种迭代器：

- 输出迭代器
 - > 前向迭代器
 - > 双向迭代器
 - > 随机存取迭代器

右向箭头（->）意味着“可以被...替换”。例如，对于那些需要使用输出迭代器的算法，如果我们传给它们一个前向迭代器，它们的执行不应该有任何异常。但反之则不成立。

如果只需对序列进行只读 (read-only) 操作 , 我们可以选择以下的任意一种迭代器 :

输入迭代器

- > 前向迭代器
- > 双向迭代器
- > 随机存取迭代器

在这种情况下 , 输入迭代器是这些种类中功能最少的一种。

最后 , 如果需要对序列进行读写 (read/write) 操作 , 我们可以选择以下的任意一种迭代器 :

前向迭代器

- > 双向迭代器
- > 随机存取迭代器

记住 , 对象指针总是可以当作随机存取迭代器来使用。因此 , 只要它支持对指定的序列进行适当的读写操作 , 它也就可以当作任意种类的迭代器来使用。

迭代器的这种 “ 代数学上 ” 的应用几乎是STL中其他部分的基础。清楚地了解每一种迭代器的适用范围及限制条件 , 对于我们了解迭代器在STL容器及算法中的应用极为重要。

习题

习题1-1

写出下面操作所需的功能最少的迭代器种类 :

- 提供无限个 0
- 向文件中写入一个值序列
- 实现一个栈 (后进先出队列)

习题1-2

下面列出的几种迭代器中 , 哪一种是可以替换的 :

- 输出迭代器
- 只读前向迭代器
- 随机存取迭代器

习题1-3

迭代器同样也可以基于Fortran语言格式的Do循环 :

```
for ( p = first; p <= last; ++p )
    <process>(*p);
```

试比较这种格式与STL中所选择的那种格式 (见本章 “ 输入迭代器 ” 一节)。

- 习题1-4 解释为什么在所写的算法中使用其他种类的迭代器，而不是随机存取迭代器？
- 习题1-5 解释为什么宁愿定义一个仅能通过迭代器来存取的数据结构，而不是让它可以被随机存取呢？
- 习题1-6 [较难] 写出这样的一个模板类bidir<FwdIt>，当我们用一个前向迭代器类型来特化FwdIt时，它的表现就和双向迭代器一样。你会采用何种方法来使它和预期中的双向迭代器行为一致？
- 习题1-7 [特难] 写出这样的一个模板类ran_read<InIt>，当我们用一个输入迭代器类型来特化InIt时，它的表现就和一个只读的随机存取迭代器一样。

第5章 算法

背景知识

算法是STL中的“苦工”。它们表现为一系列的模板函数。也就是说，在STL中最类似于传统函数库的那部分就是算法。我们一直强调，算法的主要不同之处在于：算法是一些模板函数。它们并不是作为预先编译好的对象模块组成的可链接库来提供的。确切地说，它们一般都是完整地定义在STL头文件中。我们可以以众多的方式来特化每一个模板函数，以此极大地提升它作为泛型程序组件的适用性。

迭代器 除了少数的例外情况，这些模板函数都是使用迭代器作为它的参数，以此来在序列上进行各种操作。这也就是我们在本书的一开始花大力气来详细介绍迭代器的原因，尽管这个话题有时候会很枯燥。在了解了迭代器的广泛潜能之后，现在你应该准备好去了解这些算法中的含意了。

这些模板函数相当独立。我们只需添加极少的代码，就可以把它们放置在程序的各个地方，尤其是在那些我们原本使用传统指针作为迭代器的地方。也就是说，我们在下几章中所提出的方法可以很快运用在我们所写的任何C++代码中，并且作用相当明显。

容器 这些算法和我们将要在本书的后面部分描述的STL容器之间合作得非常好。实际上，许多容器模板类的成员函数为了突出其优势而使用了这些算法。但是请注意，对容器的理解并不是使用算法所必需的。算法并不直接使用STL容器。确切地说，算法通过使用这些容器对象的成员函数提供的

迭代器来操作容器对象所管理的序列。这也就是我们把容器放在本书的最后讲述的原因。可以在不知道容器的前提下理解算法，但在一点也不了解算法的前提下想把容器描述清楚就不那么简单了。

```
<algorithm>
<numeric>
<functional>
```

我们已经在前面介绍的几个头文件中描述过一些算法，例如第4章中的 uninitialized_copy、uninitialized_fill 和 uninitialized_fill_n。剩下的算法在两个头文件 <algorithm> 和 <numeric> 中定义。另外还有一个头文件 <functional>，在它里面定义了许多的用以描述函数对象（function object）的模板类，很多算法可以使用这些函数对象以发挥其优势。

很多算法的一个关键部分是它必须能够完成确定的测试。举例来说，为了从一个序列中找到最小的元素，我们可能需要定义一些类似于“smallest”的谓词。更确切地说，算法必须重复性地检测诸如 smaller(x, y) 之类的表达式的布尔结果（在这个表达式中，x 和 y 都是序列中的元素）。在这种情况下，显而易见的解决方法就是使用表达式 $x < y$ ——实际上有很多的算法也是这么做的。例如，我们可以声明模板函数如下：

```
template<class FwdIt>
FwdIt smallest(FwdIt first, FwdIt last);
```

它返回一个迭代器，指向区间 [first, last] 中最小的那个元素（使用 operator< 来判断元素的大小）。

函数对象

但是采取这样的方法来实现该算法的一个重要方面实在让人觉得遗憾，尤其是当模板机制容许以很大的自由度存取序列的时候更是如此。于是，我们引入了函数对象（function object）。函数对象没有必要存储任何数据。和我们在第3章中讲述的迭代器标签一样，函数对象的类型本身就已经传达了足够的信息。下面就是一种声明函数对象的方法：

```
struct smaller_int {
    bool operator()(int x, int y) const
        {return (x < y); }
} f;
```

可以将 f 看作函数名。例如，如果一切无误的话，表达式 f(x, y) 就会调用在 smaller_int 中所定义的那个成员函数。

也就是说，可以声明另一个版本的模板类 smallest，使它看起来如下所示：

```
template<class FwdIt, class Pred>
FwdIt smallest(FwdIt first, FwdIt last, Pred pr);
```

然后就可以在实际调用该函数时通过 smallest 提供你所希望的功能，如下

面所描述的一样：

```
int get_smallest(int *first, int *last)
{return (*smallest(first, last, smaller_int())); }
```

在上面的函数调用中，第三个参数产生了一个无关紧要的对象，它的主要目的就是在模板函数特化时，检测其参数Pred的实际类型。

顺便说一句，请注意：第三个参数也可以为一个指向函数的指针。也就是说，也可以将上面的例子重写成如下模样：

```
bool is_smaller_int(int x, int y)
{return (x < y); }

int get_smallest(int *first, int *last)
{return (smallest(first, last, &is_smaller_int)); }
```

（其中在is_smaller_int前面的&符号是可选的。）C和C++都允许将表达式pfn(x, y)作为更有启迪作用的(*pfn)(x, y)的替代形式。这样，模板特化将产生实际有效的代码。

与指向函数的指针相比，函数对象有如下的几个好处：

- 函数对象可以定义多个重载版本。
- 在函数被调用时，可以使用函数对象所存储的值。
- 在必要时可以用一个函数指针所替换。
- 函数调用很有可能会被内联代码替换。

和上面给出的模板函数smallest差不多的是，算法由通常以两种形式出现在STL中的谓词来表现，其中一种形式固定了最可能使用的谓词形式；另一种形式则以一个附加的函数对象参数来给出该谓词。

这种方法的最终结果就是使实现算法的代码体积几乎大了一倍，至少在一个典型的实现中是这样的。但是我们由此获得的回报就是：对于那些经常使用的谓词，可以得到紧凑且高效的代码，并且不会为此牺牲我们在使用函数对象时所带来的功能。

我们将在后续章节中发现函数对象的功能确实是非常强大的。第8章将提供大量的细节。

功能描述

在对算法的描述中，我们会使用这样几个惯用语：

在区间...中

惯用语“在区间[A, B)中”意味着具有0个或多个离散值的序列从A开始到B结束（但不包括B）。区间只有在B对于A来说是可到达(reachable)

时才算是有效的——A可到达B意味着：把A存储在一个对象N中($N = A$)，然后对这个对象进行0次或多次递增($++N$)，在进行有限次的递增后该对象和B相等($N == B$)。

惯用语“位于区间[A, B)中的每个N”意味着N是从值A开始、经过0次或多次递增后直到等于B时的所有值。其中 $N == B$ 这种情况不在区间范围内。

最小值

惯用语“对于位于区间[A, B)中的每个N，使得X成立的最小值 (lowest value)”意味着从A开始，对于位于区间[A, B)中的每个N，当判断条件X成立时，那个值就是我们所说的“最小值”。

最大值

惯用语“对于位于区间[A, B)中的每个N，使得X成立的最大值 (highest value)”通常意味着对于所有位于区间[A, B)中的N都检测一次X是否成立。该函数在每次X成立时都会把N的一个拷贝存储到K中。如果最后K中存储有某个值的话，我们就会用K的值来替换N的最终值（它本来等于B）。然而，对于双向或者随机存取迭代器来说，它也可能意味着N是从区间末端开始，经过有限次的递减后，使判断条件X成立的那个值。

X - Y

当X和Y可以为不同于随机存取迭代器的迭代器时，诸如X - Y这样的表达式有着和其外表相符的数学意义。如果该函数一定要得到这样的一个值的话，它并不一定要执行operator-。这同样也适用于诸如X + N和X - N（此处N为一个整型值）这样的表达式。

严格弱序

有些算法使用的谓词对序列中的元素对施加严格弱序 (stric weak ordering)。例如，对于谓词pr(X, Y)来说：

- pr(X, X)为false(即在排序好的序列中，X不可能排在它自身前面)。
- 当!pr(X, Y) && !pr(Y, X)为true时，我们就说X和Y有相等次序 (equivalent ordering) (并没有定义 $X == Y$)。
- pr(X, Y) && pr(Y, Z)意味着pr(X, Z) (次序具有传递性)。

这些算法中的部分隐式地使用了谓词 $X < Y$ ，还有部分使用了作为函数对象传递过来的谓词pr (X, Y)。满足严格弱序需求的谓词有：操作于算术类型以及string对象上的 $X < Y$ 和 $X > Y$ 。然而，请注意：操作于同样类型上面的谓词 $X \leq Y$ 和 $X \geq Y$ 并不能够满足这个需求。

按...排序

如果对于区间[0, last - first)中的每个N以及区间(N, last - first)中的每个M，谓词!(*first + M) < *(first + N)都为true，那么我们就说：由区间[first, last)中的迭代器所指定的元素序列是一个“按operator<排序的序列”。（注意元素是以升序排序的。）谓词函数operator< (或者其他替代它的函数) 不应该改变它们的操作数。而且，它必须在所比较的操作数上施加严格弱序。

堆

在下面条件满足时，我们就说由区间[first, last)中的迭代器所指定的

元素序列是“一个按operator<排序的堆（heap）”：

- 对于每个位于区间[1, last - first]中的N来说，谓词!(*first < *(first + N))都为true。（即第一个元素就是堆中最大的那个元素。）
- 可能在对数时间内向堆中插入(push_heap)一个新元素或者从堆中删除(pop_heap)最大的元素并且结果序列也能够保持堆的准则。

对于堆的内部结构，我们仅能通过模板函数make_heap、pop_heap和push_heap来得知一二。（参见第6章。）对于一个有序序列来说，谓词函数operator<或者其他替代函数不应该改变它们的操作数，并且它们必须在所比较的操作数上施加严格弱序。

使用算法

我们可以像使用传统的函数库一样使用STL算法。将定义你想使用的模板函数的头文件包含到程序中。然而，请记住：对于不正常使用它们的程序来说，其出错模式将会很不一样。如果在程序中错误地调用了传统的函数，翻译器不是给出一个诊断信息，就是暗地里将一个或多个参数的类型转换成该函数所期望的类型。但对于后者来说，它将导致运行期间的古怪行为，不过幸好现在的调试程序(debugger)一般都可以帮我们识别这样的错误。

如果我们错误地特化一个模板函数，翻译器的反应可能和上面一样，也可能为模板参数选择一个错误的类型。这种不正确的类型可能会导致难以理解的出错信息产生，尤其是对于那些要特化其他模板的模板更是如此。在经过多次对模板的特化后，即便是运行时的除错工作也会变得异常困难。

为了缓和这个问题，我们惟一能做到的最重要的事情就是：将函数参数表达式中的隐式类型转换的使用最小化。无论如何，这都是一个很棒的主意，尤其是当函数有着多个重载版本时更是如此。参数表达式的类型可以影响到重载版本的选择，当没有一个重载版本的参数类型能够精确地和它匹配时，我们就有可能得到错误的重载调用。如果翻译器能够报告一个歧性错误而不是自己胡乱地猜测，我们就应该感到庆幸了。

由于模板函数可以使用参数类型来决定模板参数，所以它误入歧途的机会。本书中的代码在参数表达式中所使用的类型转换机制都很明智，这极大地降低了出现类型为不正确的可能性。我们建议采用类似的风格。

习题

习题5-1

请写出本章中所描述过的模板函数smallest的两种形式。

- 习题5-2 请对上一题中所写出的模板函数进行修改，使之可以使用输入迭代器（而不是前向迭代器）正常地工作。
- 习题5-3 在下面的序列中，dominates是否构成了一个严格弱序？如果没有，为什么？
- ```
rock dominates scissors
scissors dominates paper
paper dominates rock
```
- 习题5-4      在下面的序列中，哪些可能是堆？
- {1}
  - {1, 1}
  - {1, 2}
  - {8, 7, 6, 5, 4, 3, 2, 1}
- 习题5-5      写出模板类less，使得less<T>()(x, y)在x小于y时返回true(此处x和y的类型都是T)。
- 习题5-6      写出模板类less\_by，使得less\_by<T>(d)(x, y)在x + d小于y时返回true(此处d、x和y的类型都是T)。
- 习题5-7      [较难] 请写出模板函数is\_less\_by，使得指向它的一个特化版本的函数指针可以取代上题中的函数对象less\_by<T>(d)。
- 习题5-8      [特难] 如果有一个序列，它所遵循的排序规则会在排序过程中发生改变（如：任务的优先化列表），你将如何对这个序列进行排序？

## 附录 A 接口

STL的头文件非常独立。由于所有的模板定义都限制在头文件中，所以它们不需要从任何C++源文件获得支持。它们也可以用来写出高移植性的标准C++程序。实际上，STL的头文件只直接依赖于标准C++库中的其他4个头文件：

- |                                                                                                     |                                                                                                                                                                                                                                                             |
|-----------------------------------------------------------------------------------------------------|-------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <b>&lt;cstddef&gt;</b><br><b>&lt;iostream&gt;</b><br><b>&lt;new&gt;</b><br><b>&lt;stdexcept&gt;</b> | <ul style="list-style-type: none"> <li>• &lt;cstddef&gt;中定义了类型ptrdiff_t和size_t。</li> <li>• &lt;iostream&gt;中定义了几个模板的前向引用(forward reference)。</li> <li>• &lt;new&gt;中声明了operator delete和operator new的几个版本。</li> <li>• &lt;stdexcept&gt;中定义了两个异常类。</li> </ul> |
|-----------------------------------------------------------------------------------------------------|-------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
- 当然，这些头文件也会依赖于标准 C++库中的其他头文件，但对 STL 的直接接口已经足够窄了。

**cstdint**

程序清单 A-1 列出了文件 cstdint 中的相关部分。C++ 标准引入了这个头文件，使得原来在标准 C 头文件<stddef.h>中的定义现在包含在名字空间 std 中。STL 中只使用了<cstdint>中的类型定义 ptrdiff\_t 和 size\_t。

```
程序清单 A-1: // cstdint standard header (partial)

cstdint namespace std {
 // TYPE DEFINITIONS
 typedef int ptrdiff_t; // or another signed integer type
 typedef unsigned int size_t; // or another unsigned integer type
} /* namespace std */
```

**iosfwd**

程序清单 A-2 列出了文件 iosfwd 中的相关部分。C++ 标准导入这个头文件是为了给其他头文件中所定义的模板和类提供前向引用(即不完全类型的声明)。STL 中只对如下 4 个模板迭代器指向的模板声明感兴趣：istream\_iterator、 ostream\_iterator、 istreambuf\_iterator 和 ostreambuf\_iterator。

## 程序清单 A-2:

```
// iosfwd standard header (partial)

iosfwd namespace std {
 // FORWARD REFERENCES
 template<class E>
 class char_traits;
 template<class E, class Tr>
 class basic_istream;
 template<class E, class Tr>
 class basic_ostream;
 template<class E, class Tr>
 class basic_streambuf;
} /* namespace std */
```

**char\_traits**

char\_traits 描述了在输入流(或输出流)中元素(或通常意义的字符)的属性。

**basic\_istream**

basic\_istream 描述的对象控制从输入流中提取元素的操作。一个常见的例子就是标准的输入流 cin。

|                        |                                                                                                                                                                                                                                                    |
|------------------------|----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <b>basic_ostream</b>   | basic_ostream 描述的对象控制向输出流中插入元素的操作。一个常见的例子就是标准的输出流 cout。                                                                                                                                                                                            |
| <b>basic_streambuf</b> | basic_streambuf 描述的对象实现了对输入流(或输出流)的实际缓冲。                                                                                                                                                                                                           |
| <b>new</b>             | 程序清单 A-3 列出了文件 new 中的相关部分。C++ 标准引入该头文件作为传统的头文件<new.h>的后续。为了显式调用 operator new 或 operator delete，必须在程序中包含<new>。(相反地，如果只是简单地写一些 new 和 delete 的表达式，就不需要包含这个头文件。) 模板类 allocator 以及模板函数 get_temporary_buffer 会直接调用这些操作符，以使得存储空间的分配及释放从和对象的构造及销毁的耦合中解脱出来。 |

## 程序清单 A-3:

```
// new standard header (partial)

new namespace std {
 // STRUCT nothrow_t
 struct nothrow_t {};
 extern const nothrow_t nothrow; // = nothrow_t()
} /* namespace std */

// OPERATOR new
void *operator new(size_t);
void *operator new(size_t,const std::nothrow_t&) throw();
inline void *operator new(size_t,void *P); // just return P

// OPERATOR delete
void operator delete(void *P);
```

|                  |                                                                                                                                                                                                                                                                    |
|------------------|--------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <b>nothrow_t</b> | 结构体 nothrow_t 是标准化委员会的一个发明。它只用于辨别 operator new 的一个版本，在该版本中，如果不能从系统中分配到足够的存储空间，它将返回一个空指针。相反，传统的 operator new 版本如今已不再返回空指针了；如果不能分配到足够的存储空间，它就会向外抛出一个异常。至于常数 nothrow 只是为了方便而产生的，我们可以用 new(nothrow) X 来产生一个类型为 X 的对象，而不是用 new(nothrow_t()) X。相对来说，后者比较啰嗦且容易产生不必要的附加代码。 |
|------------------|--------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|

- new**           STL 使用了三种形式的 operator new :
- 如果请求的存储分配必须成功或是抛出异常(不成功) , 使用 operator new(N)。
  - 如果请求的存储分配可以失败并且返回一个空指针 , 使用operator new(N, nothrow)。
  - 为了支持定位放置 new 的语法 , 使用operator new(N, P)。
- 定位放置**      上面给出的最后那种new表达式使得我们可以在已经分配的空间上面构造出另外一个对象 , 如 :
- new(P) T()**
- 它将会在先前所分配好的未构造的存储空间上 (以P指明该空间的起始位置)调用类T的默认构造函数 , 以构造出一个类型为T的对象。
- delete**         通过调用operator delete(P)来释放先前由operator new所分配的存储空间。如果在该空间上已经有对象被构造出来 , 请先显式地将它销毁。
- stdexcept**      程序清单A-4列出了文件stdexcept中的相关部分。C++标准导入这个头文件并在它里面定义了几个异常类。STL类只显式地抛出以下两个异常类的对象 :

```
程序清单 A-4: // stdexcept standard header (partial)

stdexcept #include <exception>

namespace std {

 // CLASS logic_error

 class logic_error : public exception {
public:
 explicit logic_error(const string& S);
};

 // CLASS length_error

 class length_error : public logic_error {
public:
 explicit length_error(const string& S);
};
}
```

---

实际上该空间也可以是已经构造过的空间 ,但此时调用定位放置 new 要小心从事 ,否则容易产生内存泄露。  
译者注。

```
// CLASS out_of_range

class out_of_range : public logic_error {

public:
 explicit out_of_range(const string& s);
};

} /* namespace std */
```

- length\_error** • 为了向外报告试图把容器中的被控序列变得太长的企图，STL会抛出类型为length\_error的异常。
  - out\_of\_range** • 为了向外报告试图在随机存取容器中存取以一个有效区间外的元素的企图，STL会抛出类型为out\_of\_range的异常。
  - e**
  - logic\_error** 上面这两个类都是派生自logic\_error。当出现的错误可能是由不正确的程序逻辑引起时，就会向外抛出类型为该基类的异常。
  - exception** 实际上，logic\_error也是由类exception所派生而来的，exception同时也是标准C++库所抛出的一切异常的基类。很显然，exception应该在头文件<exception>中定义。所有的这些异常类都定义了一个带有错误消息的构造函数，该错误消息是一个类型为string的对象。一旦包含头文件<string>，也就获得了模板类basic\_string和char\_traits的定义。
  - string** 在这个头文件中，它还同样定义了string，它是basic\_string<char,char\_traits<char>>的同义词。
- 如此等等。如果试图把STL头文件的所有依赖关系都列出来的话，我们可能最终会把整个标准C++库的大部分都描述一遍。此处的展示出于有限的目的。它只是展示了STL所依靠金字塔部分的一个小小的顶端。