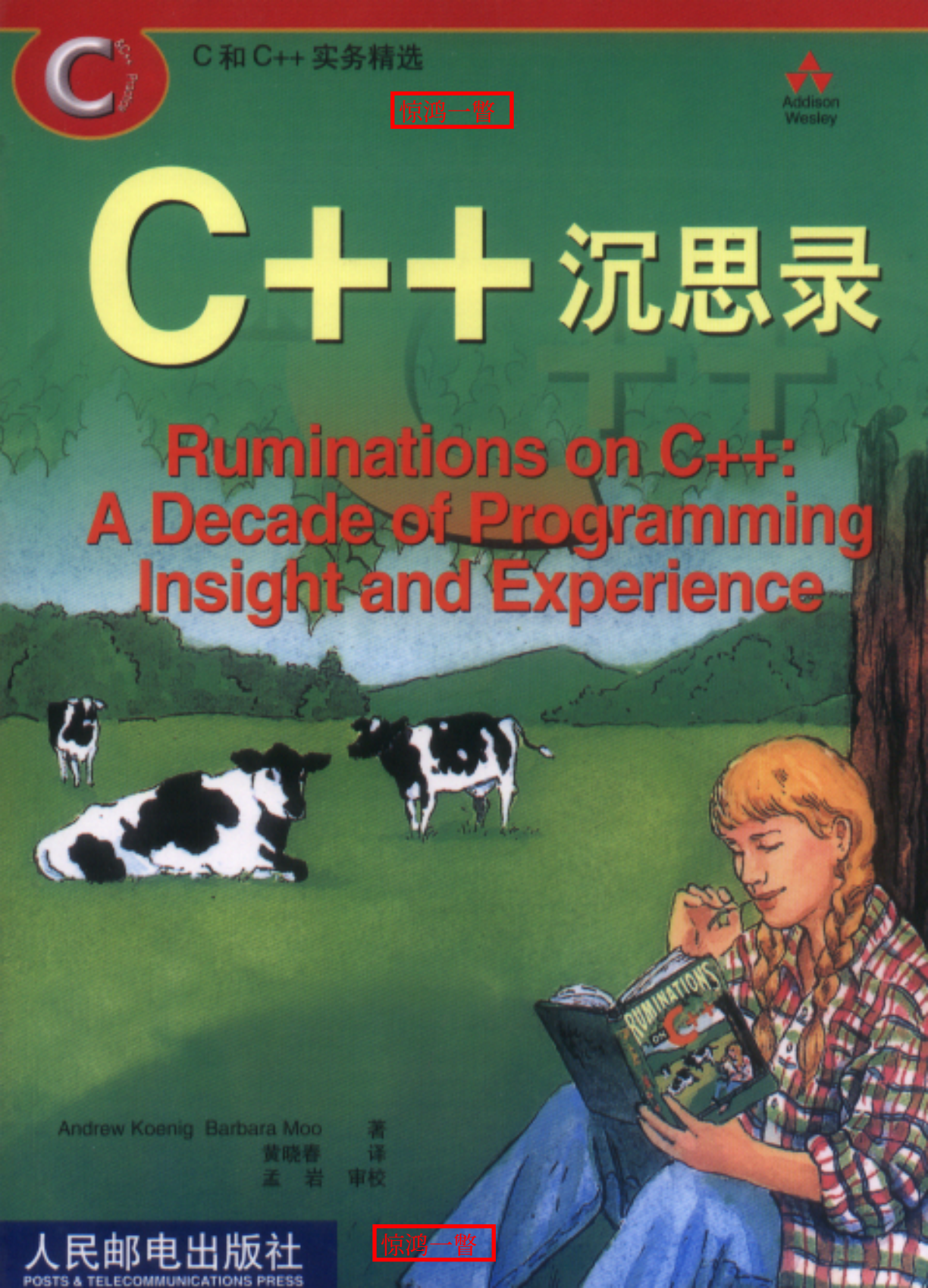
C 和 C++ 实务精选

惊鸿一瞥

Addison Wesley

C++沉思录

Ruminations on C++: A Decade of Programming Insight and Experience



Andrew Koenig Barbara Moo 著
黄晓春 译
孟 岩 审校

人民邮电出版社
POSTS & TELECOMMUNICATIONS PRESS

惊鸿一瞥

内容提要

本书基于作者在知名技术杂志发表的技术文章、世界各地发表的演讲以及斯坦福大学的课程讲义整理、写作而成，融聚了作者 10 多年 C++程序生涯的真知灼见。

全书分为 6 大部分，共 32 章，分别对 C++语言的历史和特点、类和继承、STL 与泛型编程、库的设计等几大技术话题进行了详细而深入的讨论，细微之处几乎涵盖了 C++所有的设计思想和技术细节。全书通过精心挑选的实例，向读者传达先进的程序设计的方法和理念。

本书适合有一定经验的C++程序员阅读学习，可以帮助你加强提高技术能力，成为 C++程序设计的高手。

前言

原由

1988 年初，大概是我刚刚写完 *C Traps and Pitfalls*（本书中文版《陷阱与缺陷》由人民邮电出版社出版）的时候，Bjarne Stroustrup 跑来告诉我，他刚刚被邀请参加了一个新杂志的编委会，那个杂志叫做《面向对象编程月刊》(*Journal of Object-Oriented Programming，JOOP*)。该杂志试图在那些面孔冰冷的学术期刊与满是产品介绍和广告的庸俗杂志之间寻求一个折中。他们在找一个 C++专栏作家，问我是否感兴趣。

那时，C++对于编程界的重要影响才刚刚开始。Usenix 其时才刚刚在新墨西哥圣达菲举办了第一届 C++交流会。他们预期有 50 人参加，结果到场的有 200 人。更多的人希望搭上 C++快车，这意味着 C++社群急需一个准确而理智的声音，去对抗必然汹涌而至的谣言大潮。需要有个人能够在谣言和实质之间明辨是非，在任何混乱之中保持冷静的头脑。无论如何，我顶了上去。

在写下这些话的时候，我正在构思我为 *JOOP* 撰写的第 63 期专栏。这个专栏每期或者每两期就会刊登。其间，我也有过非常想中断的时候，非常幸运的是，Jonathan Shopiro 接替了我。偶尔，我只是写一些当期专栏的介绍，然后到卓越的丹麦计算机科学家 Bjørn Stavstrup[1]那里去求助。此外，Livleen Singh 曾跟我谈起为季刊 *C++ Journal* 撰写稿件的事，那个杂志在发行 6 期之后停刊了。Stan Lippman 也甜言蜜语地哄着我在 *C++ Report* 上开了个专栏，当时这本杂志刚刚从一分简陋的通信时刊正式成为成熟的杂志。加上我在 *C++ Report* 上发表的 29 篇专栏文章，我一共发表了 98 篇文章。

[1] 就是 C++创造者 Bjarne Stroustrup，这里可能是丹麦文。——译者注

在这么多的杂志刊物里，分布着大量的材料。如果这些文章单独看来是有用的，那么集结起来应该会更有用。所以，Barbara[1]和我（主要是 Barbara）重新回顾了所有的专栏，选择出其中最好的，并根据一致性和连续性的原则增补和重写了这些文章。

本书正是世界所需的又一本 C++书籍

既然你已经知道了本书的由来，我就再讲讲为什么要读这本书，而不是其他的 C++书籍。天知道！C++方面的书籍太多了，为什么要选这一本呢？

第一个原因是，我想你们会喜欢它。大部分 C++书籍都没有顾及到这点：它们应该是基于科目教学式的。吸引人最多不过是次要目标。

杂志专栏则不同。我猜想肯定会有一些人站在书店里，手里拿着一本 *JOOP*，扫一眼我 Koenig 的专栏之后，便立刻决定购买整本杂志。但是要是我自认为这种情况很多的话，就未免太狂妄自大了。绝大多数读者都是在买了书之后读我的专栏的，也就是说他们有绝对的自由来决定是否读我的专栏。所以，我得让我的每期专栏都货真价实。

本书不对那些晦涩生僻的细节进行琐碎烦人的长篇大论。初学者不应该指望只读这本书就能学会 C++。具备了一定基础的人，比如已经知道几种编程语言的人，以及已经体会到如何通过阅读代码推断出一门新语言的规则的人，将能够通过本书对 C++有所了解。大部分从头开始学的读者先读 Bjarne Stroustrup 的 *The C++ Programming Language*（Addison-Wesley 1991）或者 Stan Lippman 的 *C++ Primer*（Addison-Wesley 1991），然后再读这本书，效果可能会更好。[2]

这是一本关于思想和技术的书，不是关于细节的。如果你试图了解怎样用虚基类实现向后翻腾两周半，就请到别处去找吧。这里所能找到的是许多等待你去阅读分析的代码。请试一试这些范例。根据我们的课堂经验，想办法使这些程序运行起来，然后加以改进，能够很好地巩固你的理解。至于那些更愿意从分析代码开始学习的人，我们也从本书中挑选了一些范例，放在 ftp.aw.com 的目录 cseng/authors/koenig/ruminations 下，可以匿名登录获取。

如果你已经对 C++有所了解，那么本书不仅能让你过一把瘾，而且能对你有所启示。这也是你应该阅读本书的第二个原因。我的意图并不是教 C++本身，而是想告诉你用 C++编程时怎样进行思考，以及如何思考问题并用 C++表述解决方案。知识可以通过系统学习获取，智慧则不能。

---

[1] 本书合作者 Barbara Moo 是 Andrew Koenig 的夫人，退休前是 Bell 实验室高级项目管理人员，曾负责 Fortran 和 CFront 编译器的项目管理。——译者注

[2] 这两本 C++百科大全类的名著分别于 1997 年和 1998 年推出了各自的第三版，Bjarne Stroustrup 还于 2000 年推出了 *The C++ Programming Language* 特别版。——译者注

组织

就专栏来说，我尽力使每期文章都独立成章，但我相信，对于结集来说，如果能根据概念进行编排，将更易于阅读，也更有趣味。因此，本书划分为6篇。

第一篇是对主题的扩展介绍，这些主题将遍布本书的其余部分中。本部分中没有太多的代码，但是所展现的有关抽象和务实的基本思想贯穿本书，更重要的是，这些思想渗透了C++设计原则和应用策略。

第二篇着眼于继承和面向对象编程，大多数人都认为这些是C++中最重要的思想。你将知道继承的重要性何在，它能做什么。你还会知道为什么将继承对用户隐藏起来是有益的，以及什么时候要避免继承。

第三篇探索模板技术，我认为这才是C++里最重要的思想。我之所以这样认为，是因为这些模板提供了一种特别的强大的抽象机制。它们不仅可以构造对所包含的对象类型一无所知的容器，还可以建立远远超出类型范畴的泛型抽象。

继承和模板之所以重要的另一个原因是，它们能够扩展C++，而不必等待（或者雇佣）人去开发新的语言和编译器。进行扩展的方法之一就是通过类库。第四篇谈到了库——包括库的设计和使用。

对基础有了很好的理解以后，我们可以学习第五篇中的一些特殊编程技术了。在这部分，你可以知道如何把类紧密地组合在一起，或者把它们尽可能地分离开。

最后，在第六篇，我们将返回头来对本书所涉及到的内容做一个回顾。

编译和编辑

这些经年累月写出来的文章有一个缺陷，就是它们通常都没有用到语言的现有特性。这就导致了一个问题：我们是应该在C++标准尚未最终定稿的时候，假装ISO C++已经成熟了，然后重写这些专栏，还是维持古迹，保留老掉牙的过时风格呢？[1]

还有许多这样的问题，我们选择了折中。对那些原来的栏目有错的地方——无论是由于后来语言规则的变化而导致的错误，还是由于我们看待事物的方式改变而导致的错误——我们都做了修正。一个很普遍的例子就是对 const 的使用，自从 const 加入到语言中以来，它的重要性就在我们的意识中日益加强。

另一方面，例如，尽管标准委员会已经接受 bool 作为内建数据类型，这里大量的范例还是使用 int 来表示真或者假的值。这是因为这些专栏文章早在这之前就完成了，使用 int 作为真、假值还将继续有效，而且要使绝大多数编译器支持 bool 还需要一些年头。

---

[1] 本书编写于1996年底，当时C++标准已经发布了草案第二版，非常接近最终标准。次年(1997)，C++标准正式定稿。本书内容是完全符合C++标准的。——译者注

致谢

除了在 *JOOP*、*C++ Report*、*C++ Journal* 中发表我们的观点外，我们还在许多地方通过发表讲演（和听取学生的意见）来对它们进行提炼。尤其值得感谢的是 Usenix Association 和 SIGS Publications 举办的会议，以及 *JOOP* 和 *C++ Report* 的发行人。另外，在 Western Institute in Computer Science 的赞助下，我们俩在斯坦福大学讲授过多次单周课程，在贝尔实验室我们为声学研究实验室和网络服务研究实验室的成员讲过课。还有 Dag Brück 曾为我们在瑞典组织了一系列的课程和讲座。Dag Brück 当时在朗德理工学院自动控制系任教，现在在 Dynasim AB。

我们也非常感谢那些阅读过本书草稿以及那些专栏并对它们发表意见的人：Dag Brück, Jim Coplien, Tony Hansen, Bill Hopkins, Brian Kernighan(他曾笔不离手地认真阅读了两遍), Stan Lippman, Rob Murray, George Otto 和 Bjarne Stroustrup。

如果没有以下人员的帮助，这些专栏永远也成不了书。他们是 Deborah Lafferty, Loren Stevens, Addison-Welsey 的 Tom Stone，以及本书编辑 Lyn Dupre。

我们特别感谢 AT&T 开通的经理们，是他们使得编写这些专栏并编辑成书成为可能。他们是 Dave Belanger, Ron Brachman, Jim Finucane, Sandy Fraser, Wayne Hunt, Brian Kernighan, Rob Murray, Ravi Sethi, Bjarne Stroustrup，以及 Eric Sumner。

Andrew Koenig
Barbara Moo
新泽西州吉列
1996 年 4 月

目录

第 0 章　序幕 1

0.1　第一次尝试 1

0.1.1　改进 2

0.1.2　另一种改进 3

0.2　不用类来实现 4

0.3　为什么用 C++更简单 5

0.4　一个更大的例子 6

0.5　结论 6

第一篇　动　机

第 1 章　为什么我用 C++ 11

1.1　问题 11

1.2　历史背景 12

1.3　自动软件发布 12

1.3.1　可靠性与通用性 13

1.3.2 为什么用 C 14
1.3.3 应付快速增长 15
1.4 进入 C++ 15
1.5 重复利用的软件 20
1.6 后记 21
第 2 章 为什么用 C++工作 23
2.1 小项目的成功 23
2.1.1 开销 24
2.1.2 质疑软件工厂 24
2.2 抽象 25
2.2.1 有些抽象不是语言的一部分 26
2.2.2 抽象和规范 26
2.2.3 抽象和内存管理 27
2.3 机器应该为人服务 28
第 3 章 生活在现实世界中 29

第二篇 类和继承

第 4 章 类设计者的核查表 37
第 5 章 代理类 47
5.1 问题 47
5.2 经典解决方案 48
5.3 虚复制函数 49
5.4 定义代理类 50
5.5 小结 53
第 6 章 句柄：第一部分 55

6.1 问题.....55
6.2 一个简单的类.....56
6.3 绑定到句柄.....58
6.4 获取对象.....58
6.5 简单的实现.....59
6.6 引用计数型句柄.....60
6.7 写时复制.....62
6.8 讨论.....63
第 7 章 句柄：第二部分.....67
7.1 回顾.....68
7.2 分离引用计数.....69
7.3 对引用计数的抽象.....70
7.4 存取函数和写时复制.....73
7.5 讨论.....73
第 8 章 一个面向对象程序范例.....75
8.1 问题描述.....75
8.2 面向对象的解决方案.....76
8.3 句柄类.....79
8.4 扩展 1：新操作.....82
8.5 扩展 2：增加新的节点类型.....85
8.6 反思.....86
第 9 章 一个课堂练习的分析（上）.....89
9.1 问题描述.....89
9.2 接口设计.....91
9.3 补遗.....93
9.4 测试接口.....94

9.5 策略.....95
9.6 方案.....96
9.7 图像的组合.....99
9.8 结论.....102
第 10 章 一个课堂练习的分析（下）.....103
10.1 策略.....103
10.1.1 方案.....104
10.1.2 内存分配.....105
10.1.3 结构构造.....107
10.1.4 显示图像.....110
10.2 体验设计的灵活性.....116
10.3 结论.....119
第 11 章 什么时候不应当使用虚函数.....121
11.1 适用的情况.....121
11.2 不适用的情况.....122
11.2.1 效率.....122
11.2.2 你想要什么样的行为.....125
11.2.3 不是所有的类都是通用的.....127
11.3 析构函数很特殊.....127
11.4 小结.....129

第三篇 模 板

第 12 章 设计容器类.....133
12.1 包含什么.....133
12.2 复制容器意味着什么.....134

12.3 怎样获取容器的元素 137
12.4 怎样区分读和写 138
12.5 怎样处理容器的增长 139
12.6 容器支持哪些操作 141
12.7 怎样设想容器元素的类型 141
12.8 容器和继承 143
12.9 设计一个类似数组的类 144
第 13 章 访问容器中的元素 151
13.1 模拟指针 151
13.2 获取数据 153
13.3 遗留问题 155
13.4 指向 const Array 的 Pointer 159
13.5 有用的增强操作 161
第 14 章 迭代器 167
14.1 完成 Pointer 类 167
14.2 什么是迭代器 170
14.3 删除元素 171
14.4 删除容器 172
14.5 其他设计考虑 173
14.6 讨论 174
第 15 章 序列 175
15.1 技术状况 175
15.2 基本的传统观点 176
15.3 增加一些额外操作 181
15.4 使用范例 184
15.5 再增加一些 188

15.6 请你思考 190
第 16 章 作为接口的模板 191
16.1 问题 191
16.2 第一个例子 192
16.3 分离迭代方式 192
16.4 遍历任意类型 195
16.5 增加其他类型 196
16.6 将存储技术抽象化 196
16.7 实证 199
16.8 小结 200
第 17 章 模板和泛型算法 203
17.1 一个特例 204
17.2 泛型化元素类型 205
17.3 推迟计数 205
17.4 地址独立性 207
17.5 查找非数组 208
17.6 讨论 210
第 18 章 泛型迭代器 213
18.1 一个不同的算法 213
18.2 需求的分类 215
18.3 输入迭代器 216
18.4 输出迭代器 216
18.5 前向迭代器 217
18.6 双向迭代器 218
18.7 随机存取迭代器 218
18.8 是继承吗 220

18.9 性能 220
18.10 小结 221
第 19 章 使用泛型迭代器 223
19.1 迭代器类型 224
19.2 虚拟序列 224
19.3 输出流迭代器 227
19.4 输入流迭代器 229
19.5 讨论 232
第 20 章 迭代器配接器 233
20.1 一个例子 233
20.2 方向不对称性 235
20.3 一致性和不对称性 236
20.4 自动反向 237
20.5 讨论 240
第 21 章 函数对象 241
21.1 一个例子 241
21.2 函数指针 244
21.3 函数对象 246
21.4 函数对象模板 248
21.5 隐藏中间类型 249
21.6 一种类型包罗万象 250
21.7 实现 251
21.8 讨论 253
第 22 章 函数配接器 255
22.1 为什么是函数对象 255

22.2 用于内建操作符的函数对象.....256
22.3 绑定者（Binders）.....257
22.4 更深入地探讨.....258
22.5 接口继承.....259
22.6 使用这些类.....260
22.7 讨论.....261

第四篇 库

第 23 章 日常使用的库.....265
23.1 问题.....265
23.2 理解问题：第 1 部分.....267
23.3 实现：第 1 部分.....267
23.4 理解问题：第 2 部分.....270
23.5 实现：第 2 部分.....270
23.6 讨论.....272
第 24 章 一个库接口设计实例.....275
24.1 复杂问题.....276
24.2 优化接口.....277
24.3 温故知新.....279
24.4 编写代码.....280
24.5 结论.....282
第 25 章 库设计就是语言设计.....283
25.1 字符串.....283
25.2 内存耗尽.....284
25.3 复制.....287

25.4 隐藏实现.....290
25.5 缺省构造函数.....292
25.6 其他操作.....293
25.7 子字符串.....295
25.8 结论.....296
第 26 章 语言设计就是库设计.....297
26.1 抽象数据类型.....297
26.1.1 构造函数与析构函数.....297
26.1.2 成员函数及可见度控制.....299
26.2 库和抽象数据类型.....299
26.2.1 类型安全的链接(linkage).....299
26.2.2 命名空间.....300
26.3 内存分配.....302
26.4 按成员赋值(memberwise assignment)和初始化.....303
26.5 异常处理.....305
26.6 小结.....306

第五篇 技 术

第 27 章 自己跟踪自己的类.....309
27.1 设计一个跟踪类.....309
27.2 创建死代码.....312
27.3 生成对象的审计跟踪.....313
27.4 验证容器行为.....315
27.5 小结.....320
第 28 章 在簇中分配对象.....321

28.1 问题.....321
28.2 设计方案.....321
28.3 实现.....324
28.4 加入继承.....326
28.5 小结.....327
第 29 章 应用器、操纵器和函数对象.....329
29.1 问题.....329
29.2 一种解决方案.....332
29.3 另一种不同的解决方案.....332
29.4 多个参数.....334
29.5 一个例子.....335
29.6 简化.....337
29.7 思考.....338
29.8 历史记录、参考资料和致谢.....339
第 30 章 将应用程序库从输入输出中分离出来.....341
30.1 问题.....341
30.2 解决方案 1：技巧加蛮力.....342
30.3 解决方案 2：抽象输出.....343
30.4 解决方案 3：技巧，但无蛮力.....345
30.5 评论.....348

第六篇 总 结

第 31 章 通过复杂性获取简单性.....351
31.1 世界是复杂的.....351
31.2 复杂性变得隐蔽.....352

31.3 计算机也是一样.....353
31.4 计算机解决实际问题.....354
31.5 类库和语言语义.....355
31.6 很难使事情变得容易.....357
31.7 抽象和接口.....357
31.8 复杂度的守恒.....358
第 32 章 说了 Hello world 后再做什么.....361
32.1 找当地的专家.....361
32.2 选一种工具包并适应它.....362
32.3 C 的某些部分是必需的.....362
32.4 C 的其他部分不是必需的.....364
32.5 给自己设一些问题.....366
32.6 结论.....368
附录 Koenig 和 Moo 夫妇访谈.....371
索引.....377

惊鸿一瞥

第0章 序 幕

有一次，我遇到一个人，他曾经用各种语言写过程序，唯独没用过 C 和 C++。他提了一个问题："你能说服我去学习 C++，而不是 C 吗？"，这个问题还真让我想了一会儿。我给许许多多人讲过 C++，可是突然间我发现他们全都是 C 程序员出身。到底该如何向从没用过 C 的人解释 C++呢？

于是，我首先问他使用过什么与 C 相近的语言。他曾用 Ada[1] 编写过大量程序——但这对我毫无用处，我不了解 Ada。还好他知道 Pascal，我也知道。于是我打算在我们两个之间有限的共通点之上找到一个例子。

下面看看我是如何向他解释什么事情是 C++可以做好而 C 做不好的。

0.1 第一次尝试

C++的核心概念就是类，所以我一开始就定义了一个类。我想写一个完整的类定义，它要尽量小，要足够说明问题，而且要有用。另外，我还想在例子中展示数据隐藏(data hiding)，因此希望它有公有数据(public data)和私有数据（private data)。经过几分钟的思索，我写下这样的代码：

```
# include <stdio.h>

class Trace {
public:
        void print(char* s) { printf("%s", s); }
};
```

我解释了这段代码是如何定义一个名叫 Trace 的新类，以及如何用 Trace 对象来打印输出消息：

[1] Ada 语言是在美国国防部组织下于 20 世纪 70 年代末开发的基于对象的高级语言，特别适合于高可靠性、实时的大型嵌入式系统软件，在 1998 年之前是美国国防部唯一准许的军用软件开发语言，至今仍然是最重要的军用系统软件开发语言。——译者注

```
int main()
{
        Trace t;
        t.print("begin main()\n");
        // main 函数的主体
        t.print("end main()\n");
}
```

到目前为止，我所做的一切都和其他语言很相似。实际上，即使是C++，直接使用 printf 也是很不错的，这种先定义类，然后创建类的对象，再来打印这些消息的方法，简直舍近求远。然而，当我继续解释类 Trace 定义的工作方式时，我意识到，即便是如此简单的例子，也已经触及到某些重要的因素，正是这些因素使得C++如此强大而灵活。

0.1.1 改进

例如，一旦我开始使用 Trace 类，就会发现，如果能够在必要时关闭跟踪输出(trace output)，这将会是个有用的功能。小意思，只要改一下类的定义就行：

```
#include <stdio.h>

class Trace {
public:
        Trace() {noisy = 0; }
        void print(char* s) { if (noisy) printf("%s", s); }
        void on() { noisy = 1; }
        void off() { noisy = 0; }

private:
        int noisy;
};
```

此时类定义包括了两个公有成员函数 on 和 off，它们影响私有成员 noisy 的状态。只有 noisy 为 on（非零）才可以输出。因此，

```
t.off();
```

会关闭 t 的对外输出，直到我们通过下面的语句恢复 t 的输出能力：

```
t.on();
```

我还指出，由于这些成员函数定义在Trace类自身的定义内，C++会内联(inline)扩展它们，所以就使得即使在不进行跟踪的情况下，在程序中保留 Trace 对象也不必付出许多代价。我立刻想到，只要让 print 函数不做任何事情，然后重新编译程序，就可以有效地关闭所有 Trace 对象的输出。

0.1.2 另一种改进

当我问自己“如果用户想要修改这样的类，将会如何？”时，我获得了更深层的理解。

用户总是要求修改程序。通常，这些修改是一般性的，例如“你能让它随时关闭吗？”或者“你能让它打印到标准输出设备以外的东西上吗？”我刚才已经回答了第一个问题。接下来着手解决第二个问题，后来证明这个问题在C++里可以轻而易举地解决，而在C里却得大动干戈。

我当然可以通过继承来创建一种新的Trace类。但是，我还是决定尽量让示例简单，避免介绍新的概念。所以，我修改了Trace类，用一个私有数据来存储输出文件的标识，并提供了构造函数，让用户指定输出文件：

```
#include <stdio.h>

class Trace {
public:
        Trace() { noisy = 0; f = stdout; }
        Trace (FILE* ff) { noisy = 0; f = ff; }
        void print(char* s)
                { if (noisy) fprintf(f, "%s", s); }
        void on() { noisy = 1; }
        void off() { noisy = 0; }

private:
        int noisy;
        FILE* f;
};
```

这样改动，基于一个事实：

```
printf(args);
```

等价于：

```
fprintf(stdout, args);
```

创建一个没有特殊要求的Trace类，则其对象的成员f为stdout。因此，调用fprintf所做的工作与调用前一个版本的printf是一样的。

类Trace有两个构造函数：一个是无参构造函数，跟上例一样输出到stdout；另一个构造函数允许明确指定输出文件。因此，上面那个使用了Trace类的示例程序可以继续工作，但也可以将输出定向到比如说stderr上：

```
int main()
{
        Trace t(stderr);
        t.print("begin main()\n");
        // main 函数的主体
        t.print("end main()\n");
}
```

简而言之，我运用C++类的特殊方式，使得对程序的改进变得轻而易举，而且不会影响使用这些类的代码。

0.2 不用类来实现

此时，我又开始想，对于这个问题，典型的C解决方案会是怎样的。它可能会从一个类似于函数trace()（而不是类）的东西开始：

```
#include <stdio.h>
void trace(char *s)
{
        printf("%s\n", s);
}
```

它还可能允许我以如下形式控制输出：

```
#include <stdio.h>
static int noisy = 1;
void trace(char *s)
{
        if(noisy)
                printf("%s\n", s);
}
void trace_on() { noisy = 1; }
void trace_off() { noisy = 0; }
```

这个方法是有效的，但与C++方法比较起来有3个明显的缺点。

首先，函数trace不是内联的，因此即使当跟踪关闭时，它还保持着函数调用的开销[1]。在很多C的实现中，这个额外负担都是无法避免的。

[1] Dag Brück指出，首先考虑效率问题，是C/C++文化的“商标”。我在写这段文字时，不由自主地首先把效率问题提出来，可见这种文化对我的影响有多深！

第二，C版本引入了3个全局名字：trace、trace_on和trace_off，而C++只引入了1个。

第三，也是最重要的一点，我们很难将这个例子一般化，使之能输出到一个以上的文件中。为什么呢？考虑一下我们会怎样使用这个trace函数：

```
int main()
{
        trace("begin main()\n");
        // main 函数主体
        trace("end main()\n");
}
```

采用C++，可以只在创建Trace对象时一次性指定文件名。而在C版本中，情况相反，没有合适的位置指定文件名。一个显而易见的办法就是给函数trace增加一个参数，但是需要找到所有对trace函数的调用，并插入这个新增的参数。另一种办法是引入名为trace_out的第4个函数，用来将跟踪输出转向到其他文件。这当然也得要求判断和记录跟踪输出是打开还是关闭。考虑一下，譬如，main调用的一个函数恰好利用了trace_out向另一个文件输出，则何时切换输出的开关状态呢？显然，要想使结果正确需要花费相当的精力。

0.3 为什么用C++更简单

为什么在C方案中进行扩展会如此困难呢？难就难在没有一个合适的位置来存储辅助的状态信息——在本例中是文件名和“noisy”标记。在这里，这个问题尤其让人恼火，因为在原来的情况下根本就不需要状态信息，只是到后来才知道需要存储状态。

往原本没有考虑存储状态信息的设计中添加这项能力是很难的。在C中，最常见的做法就是找个地方把它藏起来，就像我这里采用“noisy”标记一样。但是这种技术也只能做到这样；如果同时出现多个输出文件来搅局，就很难有效控制了。C++版本则更简单，因为C++鼓励采用类来表示类似于输出流的事物，而类就提供了一个理想的位置来放置状态信息。

结果是，C倾向于不存储状态信息，除非事先已经规划妥当。因此，C程序员趋向于假设有这样一个“环境”：存在一个位置集合，他们可以在其中找到系统的当前状态。如果只有一个环境和一个系统，这样考虑毫无问题。但是，系统在不断增长的过程中往往需要引入某些独一无二的东西，并且创建更多这类东西。

0.4 一个更大的例子

我的客人认为这个例子很有说服力。他走后，我意识到刚刚所揭示的东西跟我认识的另一个人在一个非常大的项目里得到的经验非常相似。

他们开发交互式事务处理系统：屏幕上显示着纸样表单的电子版本，一群人围坐在跟前。人们填写表单，表单的内容用于更新数据库，等等。在项目接近尾声的时候，客户要求做些改动：划分屏幕以同时显示两个无关的表单。

这样的改动是很恐怖的。这种程序通常充满了各种库函数调用，都假设知道“屏幕”在哪里和如何更新。这种改变通常要求查找出每一条用到了“屏幕”的代码，并要把它们替换为表示“屏幕的当前部分”的代码。

当然，这些概念就是我们在前面的例子中看到的隐藏状态(hidden state)的一种。因此，如果说在C++版本中修改这类应用程序比在C版本中容易，就不足为奇了。所需要做的事就是改变屏幕显示程序本身。相关的状态信息已经包含在类中，这样在类的多个对象中复制它们只是小事一桩。

0.5 结论

是什么使得对系统的改变如此容易？关键在于，一项计算的状态作为对象的一部分应当是显式可用的，而不是某些隐藏在幕后的东西。实际上，将一项计算的状态显式化，这个理念对于整个面向对象编程思想来说，都是一个基础[1]。

小例子里可能还看不出这些考虑的重要性，但在大程序中它们就对程序的可理解性和可修改性产生很大的影响。如果我们看到如下的代码：

```
push(x);
push(y);
add();
z=pop();
```

[1] 关于面向对象程序设计和函数式程序设计(functional programming)之间的区别，下面的这种说法可能算是无伤大雅的：在面向对象程序设计中，某项计算的结果状态将取代先前的状态，而在函数式程序设计中，并非如此。

我们可以理所当然地猜测存在一个被操作的堆栈，并设置z为x和y的和，但是我们还必须知道应该到何处去找这个堆栈。反之，如果我们看到

```
s.push(x);
s.push(y);
s.add();
z=s.pop();
```

猜想堆栈就是s准没错。确实，即使在C中，我们也可能会看到

```
push(s, x);
push(s, y);
add(s);
z=pop(s);
```

但是C程序员对这样的编程风格通常不以为然，以至于在实践中很少采用这种方式——除非他们发现确实需要更多的堆栈。原因就是C++采用类将状态和动作绑在一起，而C则不然。C不赞成上述最后一个例子的风格，因为要使例子运行起来，就要在函数push、add和pop之外单独定义一个s类型。C++提供了单个地方来描述所有这些东西，表明所有东西都是相互关联的。通过把有关系的事物联系起来，我们就能更加清晰地用C++来表达自己的意图。

惊鸿一瞥

第一篇

动　机

抽象是有选择的忽略。比如你要驾驶一辆汽车，但你又必须时时关注每样东西是如何运行的：发动机、传动装置、方向盘和车轮之间的连接等等；那么你要么永远设法开动这辆车，要么一上路就马上发生事故。

与此类似，编程也依赖于一种选择，选择忽略什么和何时忽略。也就是说，编程就是通过建立抽象来忽略那些我们此刻并不重视的因素。C++很有趣，它允许我们进行范围极其宽广的抽象。C++使我们更容易把程序看作抽象的集合，同时也隐藏了那些用户无须关心的抽象工作细节。

C++之所以有趣的第二个原因是，它设计时考虑了特殊用户群的需求。许多语言被设计用于探索特定的理论原理，还有些是面向特定的应用种类。C++不然，它使程序员可以以一种更抽象的风格来编程，与此同时，又保留了 C 中那些有用的和已经深入人心的特色。因此，C++保留了不少 C 的优点，比如偏重执行速度快、可移植性强、与硬件和其他软件系统的接口简单等。

C++是为那些信奉实用主义的用户群准备的。C 和 C++程序员通常都要处理杂乱而现实的问题；他们需要能够解决这些问题的工具。这种实用主义在某种程度上体现了 C++语言及其使用者的灵活性。例如，C++程序员总是为了特定的目的编写不完整的抽象：他们会为了解决特定问题设计一个很小的类，而不在乎这个类是否提供所有用户希望的所有功能。如果这个类够用了，则他们可以对那些不尽如人意的地方视而不见。有的情况下，现在的折衷方案比未来的理想方案好得多。

但是，实用主义和懒惰是有区别的。虽然很可能把 C++程序写得极其难以维护，但是也可以用 C++把问题精心划分为分割良好的模块，使模块与模块之间的信息得到良好的隐藏。

本书坚持以两个思想为核心：实用和抽象。在这一篇中我们开始探讨C++如何支持这些思想，后面几篇将探索C++允许我们使用的各种抽象机制。

惊鸿一瞥

第1章 为什么我用C++

本章介绍一些个人经历：我会谈到那些使我第一次对使用C++产生兴趣的事情以及学习过程中的心得体会。因此，我不会去说哪些东西是C++最重要的部分，相反会讲讲我是如何在特定情况下发现了C++的优点。

这些情形很有意思，因为它们是真实的历史。我的问题不属于类似于图形、交互式用户界面等“典型面向对象的问题”，而是属于一类复杂问题；人们最初用汇编语言来解决这些问题，后来多用C来解决。系统必须能在许多不同的机器上高效地运行，要与一大堆已有的系统软件实现交互，还要足够可靠，以满足用户群的苛刻要求。

1.1 问题

我想做的事情是，使程序员们能更简单地把自己的工作发布到不断增加的机器中。解决方案必须可移植，还要使用一些操作系统提供的机制。当时还没有C++，所以对于那些特定的机器来说，C基本上就是唯一的选择。我的第一个方案效果不错，但实现之困难令人咋舌，主要是因为要在程序中避免武断的限制。

机器的数目迅速增加，终于超过负荷，到了必须对程序进行大幅度修改的时候了。但是程序已经够复杂了，既要保证可靠性，又要保证正确性，如果让我用C语言来扩展这个程序，我真担心搞不定。

于是我决定尝试用C++进行改进工作。结果是成功的：重写后的版本较之老版本在效率上有了极大的提高，同时可靠性丝毫不打折扣。尽管C++程序天生不如相应的C程序快，但是C++使我能在自己的智力所及的范围内使用一些高超的技术，而对我来说，用C来实现这些技术太困难了。

我被 C++吸引住，很大程度上是由于数据抽象，而不是面向对象编程。C++允许我定义数据结构的属性，还允许我在用到这些数据结构时，把它们当作“黑匣子”使用。这些特性用 C 实现起来将困难许多。而且，其他的语言都不能把我所需的效率和可靠性结合起来，同时还允许我对付已有的系统（和用户）。

1.2 历史背景

1980 年，当时我还是 AT&T 贝尔实验室计算科学研究中心的一名成员。早期的局域网原型刚刚作为试验运行，管理方希望能鼓励人们更多地利用这种新技术。为了达到这个目的，我们打算增加 5 台机器，这超过了我们现有机器数目的两倍。此外，根据硬件行情的趋势来看，我们最终还会拥有多得多的机器（实际上，他们承诺使中心的网络拥有 50 台左右的机器）。这样一来，我们将不得不应对由此引发的软件系统维护问题。

维护问题肯定比你想象的还要困难得多。另外，类似于编译器的关键程序总在不断变化。这些程序需要仔细安装；磁盘空间不够或者安装时遇到硬件故障，都可能导致整台机器报废。而且，我们不具备计算中心站的优越条件：所有的机器都由使用的人共同合作负责维护。因此，一个新程序要想运行到另一台机器上，唯一的方法就是有人自愿负责把它放到上面。当然，程序的设计者通常是不愿意做这件事的。所以，我们需要一个全局性的方法来解决维护问题。

Mike Lesk 多年前就意识到了这个问题，并用一个名叫 uucp 的程序“部分地”加以解决，这个程序此后很有名气。我说“部分地”，是因为 Mike 故意忽略了安全性问题。另外，uucp 一次只允许传递一个文件，而且发送者无法确定传输是否成功。

1.3 自动软件发布

我决定扛着 Mike 的大旗继续往下走。我采用 uucp 作为传输工具，通过编写一个名叫 ASD(Automatic Software Distribution, 自动软件发布)的软件包来为程序员提供一个安全的方法，使他们能够把自己的作品移植到其他机器上，我预料这些机器的数量会很快变得非常巨大。我决定采用两种方式来增强 uucp：更新完成后通知发送者，允许同时在不同的位置安装一组文件。

这些功能理论上都不是很困难，但是由于可靠性和通用性这两个需求相互冲突，所以实现起来特别困难。我想让那些与系统管理无关的人用 ASD。为了这个目的，我应该恰当地满足他们的需求，而且没有任何琐碎的限制。因此，我不想对文件名的长度、文件大小、一次运行所能传递的文件数目等问题作任何限制。而且一旦 ASD 里出现了 bug，导致错误的软件版本被发布，那就是 ASD 的末日，我决不会再有第二次机会。

1.3.1　可靠性与通用性

C 没有内建的可变长数组：编译时修改数组大小的唯一方法就是动态分配内存。因此，我想避免任何限制，就不得不导致大量的动态内存分配和由此带来的复杂性，复杂性又让我担心可靠性。例如，下面给出 ASD 中的一个典型的代码段：

```
/* 读取八进制文件 */
param = getfield(tf)
mode = cvlong(param, strlen(param), 8);

/* 读入用户号 */
uid = numuid(getfield(tf));

/* 读入小组号 */
gid = numgid(getfield(tf));

/* 读入文件名（路径） */
path = transname(getfield(tf));

/* 直到行尾 */
geteol(tf);
```

这段代码读入文件中用 tf 标识的一行的连续字段。为了实现这一点，它反复调用了几次 getfield，把结果传递到不同的会话程序中。

代码看上去简单直观，但是外表具有欺骗性：这个例子忽略了一个重要的细节。想知道吗？那就想想 getfield 的返回类型是什么。由于 getfield 的值表示的是输入行的一部分，所以显然应该返回一个字符串。但是 C 没有字符串；最接近的做法是使用字符指针。指针必须指到某个地方；应该什么时候用什么方法回收内存？

C 里有一些解决这类问题的方法，但是都比较困难。一种办法就是让 getfield 每次都返回一个指针，这个指针指向调用它的新分配的内存，调用者负责释放内

存。由于我们的程序先后 4 次调用了 getfield，所以也需要先后 4 次在适当场合调用 free。我可不愿意使用这种解决方法，写这么多的调用真是很讨厌，我肯定会漏掉一两个。

所以，我再一次想，假如我能承受漏写一两个调用的后果，也就能承受漏写所有调用的后果。所以另一种解决方法应该完全无需回收内存，每次调用时，让 getfield 分配内存，然后永远不释放。我也不能接受这种方法，因为它会导致内存的过量消耗，而实际上，通过仔细地设计完全可以避免内存不足的问题。

我选择的方法是让 getfield 所返回内存块的有效期保持到下次调用 getfield 为止。这样，总体来说，我不用老是记着要回收 getfield 传回的内存。作为代价，我必须记住，如果打算把 getfield 传回的结果保留下来，那么每次调用后就必须将结果复制一份（并且记住要回收用于存放复制值的那块内存）。当然，对于上述的程序片断来说，付出这个代价是值得的，事实上，对于整个 ASD 系统来说，也是合适的。但是跟完全无需回收内存的情况相比，使用这种策略显然还是使得编写程序的难度增大。结果，我为了使程序没有这种局限性所付出的努力，大部分都花在进行簿记工作的程序上，而不是解决实际问题的程序上。而且由于在簿记工作方面进行了大量的手工编码，我经常担心这方面的错误会使 ASD 不够可靠。

1.3.2 为什么用 C

此时，你可能会问自己："他为什么要用 C 来做呢？"。毕竟我所描述的簿记工作用其他的语言来写会容易得多，譬如 Smalltalk、Lisp 或者 Snobol，它们都有垃圾收集机制和可扩展的数据结构。

排除掉 Smalltalk 是很容易的：因为它不能在我们的机器上运行！Lisp 和 Snobol 也有这个问题，只不过没那么严重：尽管我写 ASD 那会儿的机器能支持它们，但无法确保在以后的机器上也能用。实际上，在我们的环境中，C 是唯一确定可移植的语言。

退一步，即使有其他的语言可用，我也需要一个高效的操作系统接口。ASD 在文件系统上做了很多工作，而这些工作必须既快又稳定。人们会同时发送成百上千的文件，可能有数百万个字节，他们希望系统尽可能快，而且一次成功。

1.3.3　应付快速增长

我开始开发 ASD 的时候，我们的网络还只是个原型：有时会失效，不能与每台机器都连通。所以我用 uucp 作传输工具——我别无选择。然而，一段时间后，网络第一次变得稳定，然后成为了不可或缺的部分。随着网络的改善，使用 ASD 的机器数目也在增加。到了大概 25 台机器的时候，uucp 已经慢得不能轻松应付这样的负载了。是时候了，我们必须跨过 uucp，开始直接使用网络。

对于使用网络进行软件发布，我有一个好主意：我可以写一个 spooler 来协调数台机器上的发布工作。这个 spooler 需要一个在磁盘上的数据结构来跟踪哪台机器成功地接收和安装了软件包，以便人们在操作失败时可以找到出错的地方。这个机制必须十分强健，可以在无人干预的情况下长时间运行。

然而，我迟疑了好一阵，ASD 最初版本中那些曾经困扰过我的琐碎细节搞得我泄了气。我知道我希望解决的问题，但是想不出来在满足我的限制条件的前提下，应该如何用 C 来解决这些问题。一个成功的 spooler 必须：

- 有与尽量多的操作系统工具的接口。
- 避免没有道理的限制。
- 速度上必须比旧版本有本质的提高。
- 仍然极为可靠。

我可以解决所有这些问题，除了最后一个。写一个 spooler 本身就很难，写一个可靠的 spooler 就更难。一个 spooler 必须能够对付各种可能的奇异失败，而且始终让系统保持可以恢复的状态。

我在排除 uucp 中的 bug 上面花了数年的功夫，然而我仍然认为，对于我新的 spooler 来说，要想成功，就必须立刻做到真正的 bugfree。

1.4　进入 C++

在那种情况下，我决定来看看能否用 C++来解决我的问题。尽管我已经非常熟悉 C++了，但还没有用它做过任何严肃的工作。不过 Bjarne Stroustrup 的办公室离我不远，在 C++演化的过程中，我们曾经在一起讨论。

当时，我想 C++有这么几个特点对我有帮助。

第一个就是抽象数据类型的观念。比如，我知道我需要将向每台计算机发送软件的申请状态存储起来。我得想法把这些状态用一种可读的文件保存起来，然后在必要的时候取出来，在与机器会话时应请求更新状态，并能最终改变标识状

态的信息。所有这一切都要求能够灵活进行内存的分配：我要存储的机器状态信息中，有一部分是在机器上所执行的任何命令的输出，而这输出的长度是没有限定的。

另一个优势是 Jonathan Shopiro 最近写的一个组件包，用于处理字符串和链表。这个组件包使得我能够拥有真正的动态字符串，而不必在簿记操作的细节上战战兢兢。该组件包同时还支持可容纳用户对象的可变长链表。有了它，我一旦定义了一个抽象数据类型，比如说叫 machine_status，就可以马上利用 Shopiro 的组件包定义另一个类型——由 machine_status 对象组成的链表。

为了把设计说得更具体一些，下面列出一些从 C++版的 ASD spooler 中选出来的代码片断。这里变量 m 的类型是 machine_status：[1]

```
struct machine_status {
    String p;                       // 机器名
    List<String> q;                 // 存放可能的输出
    String s;                       // 错误信息，如果成功则为空
}
//...

m.s = domach(m.p, dfile, m.q);  // 发送文件
if (m.s.length() == 0) {        // 工作正常否?
    sendfile = 1;               // 成功——别忘了，我们是在发送一个文件
    if (m.q.length() == 0)      // 是否有输出?
        mli.remove();           // 没有，这台机器的事情已经搞定
    else
        mli.replace(m);         // 有，保存输出
} else {
    keepfile = 1;               // 失败，提起注意，稍后再试
    deadmach += m.p;            // 加到失败机器链表中
    mli.replace(m);             // 将其状态放回链表
}
```

这个代码片断对于我们传送文件的每台目标机器都执行一遍。结构体 m 将发送文件尝试的执行结果保存在自己的 3 个域当中：p 是一个 String，保存机器的名字；q 是一个 String 链表，保存执行时可能的输出；s 是一个 String，尝试成功时为空，失败时标明原因。

函数 domach 试图将数据发送到另一台机器上。它返回两个值：一个是显式的；另一个是隐式的，通过修改第三个参数返回。我们调用 domach 之后，m.s 反映了发送尝试是否成功的信息，而 m.q 则包含了可能的输出。

[1] 细心的读者可能会发现我把数据成员设为 public，并为此惊讶。我是故意这样做的：machine_status 是一个简单的类，其结构就是其接口。对于如此简单的类来说，把成员设为 private 没有任何好处。（从这个小小的脚注可以看到作者的实用主义态度，相对于后来很多人所奉行的教条主义，确实有很大的差别—译者注）。

然后，我们通过将 m.s.length()与 0 比较来检查 m.s 是否为空。如果 m.s 确实为空，那么我们将 sendfile 置 1，表示我们至少成功地把文件发送到了一台机器上，然后我们来看看是否有什么输出。如果没有，那么我们可以把这台机器从需要处理的机器链表中删除。如果有输出，则将状态存储在 List 中。变量 mli 就是一个指向该 List 内部元素的指针(mli 代表“machine list iterator”，机器链表迭代器)。

如果尝试失败，未能有效地与远程机器对话，那么我们将 keepfile 置为 1，提醒我们必须保留该数据文件，以便下次再试，然后将当前状态存到 List 中。

这个程序片断中没什么高深的东西。这里的每一行代码都直接针对其试图解决的问题。跟相应的 C 代码不同，这里没有什么隐藏的簿记工作。这就是问题所在。所有的簿记工作都可以在库里被单独考虑，调试一次，然后彻底忘记。程序的其余部分可以集中精力解决实际问题。

这个解决方案是成功的，ASD 每年要在 50 台机器上进行 4000 次软件更新。典型的例子包括更新编译器的版本，甚至是操作系统内核本身。较之 C，C++使我得以从根本上在程序里更精确地表达我的意图。

我们已经看到了一个 C 代码片断的例子，它展示了一些隐秘的细枝末节。现在，我们来研究一下，为什么 C 必须考虑这些细致末节，再来看一看 C++程序员怎样才可能避免它们。

C 中隐藏的约定

尽管 C 有字符文本量，但它实际上没有真正的字符串概念。字符串常量实际上是未命名的字符数组的简写（由编译器在尾部插入空字符来标识串尾），程序员负责决定如何处理这些字符。因此，比方说，尽管下面的语句是合法的：

```
char hello[] = "hello";
```

但是这样就不对了：

```
char hello[5];
hello = "hello";
```

因为 C 没有复制数组的内建方法。第一个例子中用 6 个元素声明了一个字符数组，元素的初值分别是 ‘h’、‘e’、‘l’、‘l’、‘o’ 和 ‘\0’（一个空字符）。第二个例子是不合法的，因为 C 没有数组的赋值，最接近的方法是：

```
char *hello;
hello = "hello";
```

这里的变量 hello 是一个指针，而不是数组：它指向包含了字符常量“hello”的内存。

假设我们定义并初始化了两个字符“串”：

```
char hello[] = "hello";
char world[] = " world";
```

并且希望把它们连接起来。我们希望库可以提供一个 concatenate 函数，这样我们就可以写成这样：

```
char helloworld[];                              //错误
concatenate(helloworld, hello, world);
```

可惜的是，这样并不奏效，因为我们不知道 helloworld 数组应该占用多大内存。通过写成

```
char helloworld[12];                            //危险
concatenate(helloworld, hello, world);
```

可以将它们连接起来，但是我们连接字符串时并不想去数字符的个数。当然，通过下面的语句，我们可以分配绝对够用的内存：

```
char helloworld[1000];                          //浪费而且仍然危险
concatenate(helloworld, hello, world);
```

但是到底多少才够用？只要我们必须预先指定字符数组的大小为常量，我们就要接受猜错许多次的事实。

避免猜错的唯一办法就是动态决定串的大小。因此，譬如我们希望可以这样写

```
char *helloworld;
helloworld = concatenate(hello, world);     //有陷阱
```

让 concatenate 函数负责判断包含变量 hello 和 world 的连接所需内存的大小、分配这样大小的内存、形成连接以及返回一个指向该内存的指针等所有这些工作。实际上，这正是我在 ASD 的最初的 C 版本中所做的事情：我采用了一个约定，即所有串以及类似串的值的大小都是动态决定的，相应的内存也是动态分配的。然而什么时候释放内存呢？

对于 C 的串库来说无法得知程序员何时不再使用串了。因此，库必须要让程序员负责决定何时释放内存。一旦这样做了，我们就会有很多方法来用 C 实现动态串。

对于 ASD，我采用了 3 个约定。前两个在 C 程序中是很普遍的，第三个则不是：

1．串由一个指向它的首字符的指针来表示。

2．串的结尾用一个空字符标识。

3．生成串的函数不遵循用于这些串的生命期的约定。例如，有些函数返回指向静态缓冲区的指针，这些静态缓冲区要保持到这些函数的下一次调用；而其他函数则返回指向调用者要释放的内存的指针。这些串的使用者需要考虑这些各不相同的生命周期，要在必要的时候使用 free 来释放不再需要的串，还要注意不要释放那些将在别的地方自动释放的串。

类似“hello”的字符串常量的生命周期是没有限制的，因此，写：

```
char *hello;
hello = "hello";
```

后我不必释放变量 hello。前面的 concatenate 函数也返回一个无限存在的值，但是由于这个值保存在自动分配的内存区，所以使用完后应该将它释放。

最后，有些类似 getfield 的函数返回一个生存期经过精心定义的但是有限的值。我甚至不应该释放 getfield 的值，但是如果想要将它返回的值保存一段很长的时间，我就必须记得将它复制到时间稍长的存储区中。

为什么要处理 3 种不同的存储期？我无法选择字符常量：它们的语义是 C 的一部分，我不能改变。但是我可以使所有其他的字符串函数都返回一个指向刚分配的内存的指针。那么我就不必决定要不要释放这样的内存了：使用完后就释放内存通常都是对的。

我不让所有这些字符串函数都在每次调用时分配新内存的主要原因是，这样做会使我的程序十分巨大。例如，我将不得不像下面这样重写 C 程序代码段（见 1.3.1 节）：

```
/* 读取八进制文件 */
param = getfield(tf);
mode = cvlong(param, strlen (param), 8);
free(param);

/* 读入用户号 */
s = getfield(tf);
uid = numuid(s);
free(s);

/* 读入小组号 */
s = getfield(tf);
gid = numgid(s);
free(s);
```

```
/* 读入文件名（路径） */
s = getfield(tf);
path = transname(s);
free(s);

/* 直到行尾*/
geteol(tf);
```

看来我还应该有一些其他的可选工具来减小我所写程序的大小。

使用 C++修改 ASD 与用 C 修改相比较，前者得到的程序更简短，而所依赖的常规更少。作为例子，让我们回顾 C++ ASD 程序。该程序的第一句是为 m.s 赋值：

```
m.s = domach(m.p, dfile, m.q);
```

当然，m.s 是结构体 m 的一个元素，m.s 也可以是更大的结构体的组成部分，等等。如果我必须自己记住要释放 m.s 的位置，就必然对两件事情有充分的心理准备。第一，我不会一次正确得到所有的位置；要清除所有 bug 肯定要经过多次尝试。第二，每次明显地改变某个东西的时候肯定会产生新的 bug。

我发现使用 C++就不必再担心所有这些细节。实际上，我在写 C++ ASD 时，没有找到任何一个与内存分配有关的错误。

1.5 重复利用的软件

尽管 ASD 的 C 版本里有许多用来处理字符串的函数，我却从没有想过要把它们封装成通用的包。向人们解释使用这些函数要遵循哪些规则实在是太麻烦了。而且，根据多年和计算机用户打交道的经验，我知道了一件事，那就是：在使用你的程序时，如果因为不遵守规则而导致工作失败，大部分人不会反躬自省，反而会怪罪到你头上。C 可以做好很多事情，但不能处理灵活多变的字符串。

C++版本的 ASD spooler 也使用字符—字符串函数，已经有人写过这些函数，所以我不用写了。和我当初发布 C 字符串规则比起来，编写这些函数的人更愿意让其他人来使用这些 C++字符串例程，因为他不需要用户记住那些隐匿的规定。同样的，我使用串库作为例程的基础来实现分析文件名所需的指定的模式匹配，而这些例程又可抽取出来用于别的工作。

此后我用 C++编程时，还有过几次类似的经历。我考虑问题的本质是什么，再定义一个类来抓住这个本质，并确保这个类能独立地工作。然后在遇到符合这

个本质的问题时就使用这个类。令人惊讶的是，解决方法通常只用编译一次就能工作了。

我的 C++程序之所以可靠，是因为我在定义 C++类时运用的思想比用 C 做任何事情时都多得多。只要类定义正确，我就只能按照我编写它的初衷那样去用它。因此，我认为 C++有助于直接表达我的思想并实现我的目的。

1.6 后记

这章内容基于一篇专栏文章，从我写那篇文章到现在已经过去很多年了。在这段时间里，我很欣慰地看到一整套 C++类库逐渐形成了。C 库到处都是，但是，可以肯定至少我所见过的 C 库都有一定的问题。而 C++则相反，它能实现真正的针对通用目的的库，编写这些库的程序员甚至根本不必了解他们的库会用于何处。

这正是抽象的优点。

惊鸿一瞥

第2章 为什么用C++工作

在第1章中，我解释了C++吸引我的地方，以及为什么要在编程中使用它。本章将对这一点进行补充说明。过去的10年时间，我都用在了开发C++编程工具，理解怎样使用它们，编写教授C++的资料，以及修改优化C++标准等工作上。C++有何魅力让我如此痴迷呢？本章中，我将做出解答。这些问题的跨度很大，就像开车上班和设计汽车之间的差距。

2.1 小项目的成功

我们很容易就会注意到：很多最成功的、最有名的软件最初是由少数人开发出来的。这些软件后来可能逐渐成长，然而，令人吃惊的是许多真正的赢家都是从小系统做起的。UNIX操作系统就是最好的例子，C编程语言也是。其他的例子还包括：电子表格、Basic和FORTRAN编程语言、MS-DOS和IBM的VM/370操作系统。VM/370尤其有趣，因为它完全是在IBM正规生产线之外发展起来的。尽管IBM多年来一直不提倡客户使用VM/370，但该操作系统仍牢牢占据IBM大型机的主流市场。

同样令人吃惊的是，很多大项目的最终结果却表现平平。我实在不愿意在公共场合指手画脚，但是我想你自己也应该能举出大量的例子来。

到底是什么使得大项目难以成功呢？我认为原因在于软件行业和其他很多行业不一样，软件制造的规模和经济效益不成正比。绝大多数称职的程序员能在一两个小时内写完一个100行的程序，而在大项目中通常每个程序员每天平均只写10行代码。

2.1.1 开销

有些负面的经济效益是由于项目组成员之间相互交流需要大量时间。一旦项目组的成员多到不能同时坐在一张餐桌旁，交流上的开销问题就相当严重了。基于这一点，就必须要有某种正规的机制，保证每个项目成员对于其他人在做什么都了解得足够清楚，这样才能确保所有的部分最终能拼在一起。随着项目的扩大，这种机制将占用每个人更多的时间，同时每个人要了解的东西也会更多。

我们只需要看一下项目组成员是如何利用时间的，就会发现这些开销是多么明显：管理错误报告数据库；阅读、编写和回顾需求报告；参加会议；处理规范以及做除编程外的任何事情。

2.1.2 质疑软件工厂

由于这些开销是有目共睹的，所以很多人正在寻找减少它的途径。起码到目前为止，我还没有见过什么有效的方法。这是个难题，我们可能没有办法解决。当项目达到一定规模时，尽管作了百般努力，所有的一切好像还是老出错；塔科马海峡大桥和“挑战者号”航天飞机灾难至今仍然历历在目。

有些人认为大项目的开销是在所难免的。这种态度的结果就是产生了有着过多管理开销的复杂系统。然而，更常见的情况是，这些所谓的管理最终不过是另一种经过精心组织的开销。开销还在，只是被放进干净的盒子和图表中，因此也更易于理解。有些人沉迷于这种开销。他们心安理得地那么做，就好像它是件“好事”——就好像这种开销真地能促进而不是阻碍高效的软件开发。毕竟，如果一定的管理和组织是有效的，那么更多的管理和组织就应该更有效。我猜想，这个想法给程序项目引进的纪律和组织，与为工厂厂房引进生产流水线一样。

我希望这些人错了。实际上我所接触过的软件工厂给我的感觉很不愉快。每个单独的功能都是一个巨大机器的一部分，“系统”控制一切，人也要遵从它。正是这种强硬的控制导致生产线成为劳资双方众多矛盾的焦点。

所幸的是，我并不认为软件只能朝这个方向发展。软件工厂忽视了编程和生产之间的本质区别。工厂是制造大量相同（或者基本相同）产品的地方。它讲求规模效益，在生产过程中充分利用了分工的优势。最近，它的目标已经变成了要完全消除人力劳动。相反，软件开发主要是要生产数目相对较少的、彼此完全不同的人造产品。这些产品可能在很多方面相似，但是如果太相似，开发工作就变

成了机械的复制过程了，这可能用程序就能完成。因此，软件开发的理想环境应该不像工厂，而更像机械修理厂——在那里，熟练的技术工人可以利用手边所有可用的精密工具来尽可能地提高工作效率。

实际上，只要在能控制的范围内，程序员（当然指称职的）就总是争取让他们的机器代替自己做它们所能完成的机械工作。毕竟，机器擅长干这样的活儿，而人很容易产生厌倦情绪。

随着项目规模越来越大，越来越难以描述，这种把程序员看成是手工艺人的观点也渐渐变得难以支持了。因此，我曾尝试描述应该如何将一个庞大的编程问题当作一系列较小的、相互独立的编程问题看待。为了做到这一点，我们首先必须把大系统中各个小项目之间存在的关系理顺，使得相关人员不必反复互相核查。换言之，我们需要项目之间有接口，这样，每个项目的成员几乎不需要关心接口之外的东西。这些接口应该像那些常用的子程序和数据结构的抽象一样成为程序员开发工具中的重要组成部分。

2.2　抽象

自从 25 年前开始编程以来，我一直痴迷于那些能扩展程序员能力的工具。这些工具可以是编程语言、操作系统，甚至可以是关于某个问题的独特思维方式。我知道有一天我将能够轻松解决问题，这些问题是我在刚开始编程时想都不敢想的——我也知道，我不是独自前行。

我最钟情的工具有一个共性，那就是抽象的概念。当我在处理大问题的时候，这样的工具总是能帮助我将问题分解成独立的子问题，并能确保它们相互独立。也就是说，当我处理问题的某个部分的时候，完全不必担心其他部分。

例如，假设我正在用汇编语言写一个程序，我必须时常考虑机器的状态。我可以支配的工具是寄存器、内存，以及运行于这些寄存器、内存上的指令。要用汇编语言做成任何一件有用的事情，就必须把我的问题用这些特定概念表达出来。

即使是汇编语言也包含了一些有用的抽象。首先是编写的程序在机器执行之前先被解释了。这就是用汇编语言写程序和直接在机器上写程序的区别。更难以察觉的是，“内存”和“寄存器”的概念本身就是对机器设计者这一部分的抽象。如果抛开抽象不用，则程序的运行就要表示成处理器内无数个门电路的状态变换。如果你的想象力够丰富的话，就可以看到除此之外还有更多层次的抽象。

高级语言提供了更复杂的抽象。甚至用表达式替代一连串单独的算术指令的想法，也是非常重大的。这种想法在20世纪50年代首次被提出时显得很不同凡响，以至于后来成了FORTRAN命名的基础：Formula Translation。抽象如此有用，因此程序员们不断发明新的抽象，并且运用到他们的程序中。结果几乎所有重要的程序都给用户提供了一套抽象。

2.2.1 有些抽象不是语言的一部分

考虑一下文件的概念。事实上每种操作系统都以某种方式使文件能为用户所用。每个程序员都知道文件是什么。但是，在大多数情况下，文件根本不是物理存在的！文件只是组织长期存储的数据的一种方式，并由程序和数据结构的集合提供支持来实现这个抽象。

要使用文件做任何一件有意义的事情，程序员必须知道程序是通过什么访问文件的，以及需要什么样的请求队列。对于典型的操作系统来说，必须确保提出不合理请求的程序得到相应的错误提示，而不能造成系统本身崩溃或者文件系统破坏。实际上，现代的操作系统已经就一个目的达成了共识，就是要在文件之间构筑“防火墙”，以便增加程序在无意中修改数据的难度。

2.2.2 抽象和规范

操作系统提供了一定程度的保护措施，而编程语言通常没有。那些编写新的抽象给其他程序员用的程序员，往往不得不依靠用户自己去遵守编程语言技术上的限制。这些用户不仅要遵守语言的规则，还要遵守其他程序员制定的规范。

例如，由malloc函数实现的动态内存的概念就是C库中经常使用的抽象。你可以用一个数字作参数来调用malloc，然后它在内存中分配空间，并给出地址。当你不再需要这块内存时，就用这个地址作参数来调用free函数，这块内存就返回给系统留作它用。

在很多情况下，这个简单的抽象都相当有用。不论规模大小，很难想象一个实际的C程序不使用malloc或者free。但是，要成功地使用抽象，必须遵循一些规范。要成功地使用动态内存，程序员必须：

- 知道要分配多大内存。
- 不使用超出分配的内存范围外的内存。
- 不再需要时释放内存。
- 只有不再需要时，才释放内存。
- 只释放分配的内存。
- 切记检查每个分配请求，以确保成功。

要记住的东西很多，而且一不留神就会出错。那么有多少可以做成自动实现的呢？用 C 的话，没有多少。如果你正在编写一个使用了动态内存的程序，就难免要允许你的用户释放掉任何由他们分配的内存，这些内存的分配是他们对程序调用请求的一部分。

2.2.3　抽象和内存管理

有些语言通过垃圾收集（garbage collection）来解决这个问题，这是一种当内存空间不再需要时自动回收内存的技术。垃圾收集使得编写程序时能更方便地采用灵活的数据结构，但要求系统在运行速度、编译器和运行时系统复杂度方面付出代价。另外，垃圾收集只回收内存，不管理其他资源。C++采用了另外一种更不同寻常的方法：如果某种数据结构需要动态分配资源，则数据结构的设计者可以在构造函数和析构函数中精确定义如何释放该结构所对应的资源。

这种机制不是总像垃圾收集那样灵活，但是在实践中，它与许多应用更接近。另外，与垃圾收集比起来它有一个明显的优势，就是对环境要求低得多：内存一旦不用了就会被释放，而不是等待垃圾收集机制发现之后才释放。

仅仅这些还不够，要想名正言顺地放弃自动垃圾收集，还应该有一些好的理由。但是构造函数和析构函数的概念在其他方面也有很好的意义。用抽象的眼光看待数据结构，它们中的许多都有关于初始化和终止的概念，而不是单纯地只有内存分配。例如，一个代表缓冲输出文件的数据结构必须体现一个思想，就是缓冲区必须在文件关闭前释放。这种约定总是在一些让人意想不到的细节地方出现，而由此产生的 bug 也总是非常隐蔽、难觅其踪。我曾经写过一个程序，整整 3 年后才发现里面隐藏了一个 bug![1] 在 C++中，缓冲输出文件类的定义必须包括一个释放该缓冲区的析构函数。这样就不容易犯错了。垃圾收集对此无能为力。

同理，C++的很多地方也都用到了抽象和接口。其间的关键就是要能够把问题分解为完全独立的小块。这些小块不是通过规则相互联系的，而是通过类定义

[1] 该类问题的例子，请参看作者的另一本著作 *C Traps and Pitfalls*(Addison Wesley 1989)。该书中文版已由人民邮电出版社出版，书名为《C 陷阱与缺陷》。

和对成员函数和友元函数的调用联系起来的。不遵守规则，就会马上收到由编译器而不是由异常征兆的出错程序发出的诊断消息。

2.3 机器应该为人服务

为什么我要关注语言和抽象？因为我认为大项目是无法高效地、顺利地投入使用的，也不可能加以管理。我从没见过，也不能想象，会有一种方法使得一个庞大的项目能够对抗所有这些问题。但是，如果我能找到把大项目化解为众多小问题的方法，就能引入个体优于混乱的整体、人类优于机器的因素。我们必须做工具的主人，而不是其他任何角色。