Linux
程序设计实践

李 林 段翰聪／著

Linux Chengxu Sheji Shijian

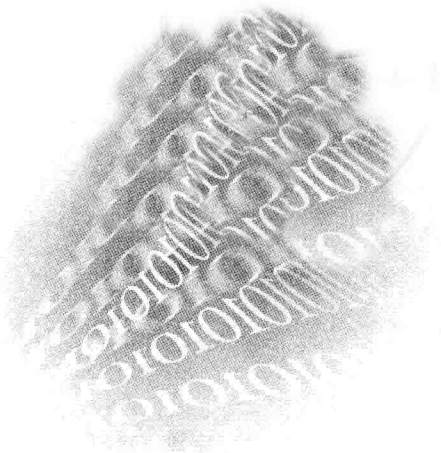
电子科技大学出版社

本书受电子科技大学“十二五”规划研究生教材建设资助出版

Linux 程序设计实践

Linux Chengxu Sheji Shijian

李　林　段翰聪／著



电子科技大学出版社

图书在版编目（CIP）数据

Linux 程序设计实践 / 李林，段瀚聪著. —成都：电子科技大学出版社，2013. 7

ISBN 978-7-5647-1752-0

Ⅰ. ①L… Ⅱ. ①李… ②段… Ⅲ. ①Linux 操作系统－程序设计－研究生－教材 Ⅳ. ①TP316.89

中国版本图书馆 CIP 数据核字（2013）第 180207 号

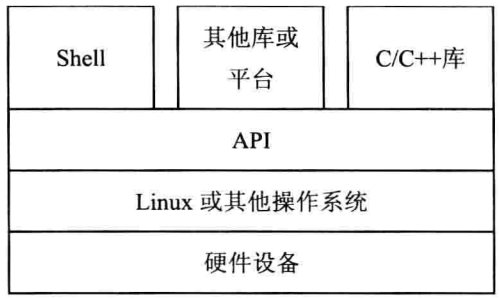
Linux 程序设计实践

李 林 段翰聪 著

出　　版：电子科技大学出版社（成都市一环路东一段 159 号电子信息产业大厦 邮编：610051）
策划编辑：杜 倩
责任编辑：李 毅
主　　页：www.uestcp.com.cn
电子邮箱：uestcp@uestcp.com.cn
发　　行：新华书店经销
印　　刷：成都火炬印务有限公司
成品尺寸：185 mm×260 mm　　印张 22　　字数 538 千字
版　　次：2013 年 7 月第一版
印　　次：2013 年 7 月第一次印刷
书　　号：ISBN 978-7-5647-1752-0
定　　价：46.00 元

序　言

随着 Internet 的不断发展，Linux 已经成为了网络服务器市场首选操作系统之一。这吸引了众多 ICT 从业人员、研究人员以及相关专业学生的注意。越来越多的人，都投入到了 Linux 相关知识的学习中。Linux 知识点所涉及的范围很广，从基本的操作使用、管理维护，到各种程序库的学习使用，再到内核模块的编写，不一而足。而本书的关注点在哪里呢？要回答这个问题，请先看下图。



从程序员视角看计算机系统

上图是站在程序员角度，所看到的计算机系统的层次结构。从图中可知，从下到上该结构一共可分为四个层次，即硬件设备、操作系统、API，以及各类开发库。对于硬件设备层而言，那是硬件工程师的工作；而内核工程师，在该体系结构中，将会看到各种操作系统，在本书中即 Linux 操作系统；第三层 API 层，对应的是基于操作系统 API 进行开发的应用程序工程师；而最上一层，则对应了基于 C/C++等各种库进行开发的应用程序工程师。显然，基于操作系统 API 和各种库的开发工作，均属于应用程序工程师。那么，本书讨论的焦点在哪里呢？本书既不讨论硬件设备的构建，也不讨论内核模块的编写。本书的关注点在于：基于 Linux 操作系统 API 和 C/C++库的应用程序开发。

目前市面上已有不少的书籍或教材，以 Linux API、C/C++库为主题进行讨论。那么，笔者为何还要编写本书呢？笔者长期以来一直在高校从事 Linux 程序设计方面的教育工作，而在此期间，经常收到同学们的诸多反馈意见。在众多的反馈意见中，有很多同学都不约而同地提到了一个共性问题，即教材上介绍的每个 API 都搞明白了，都会使用了，然而一旦被要求写个几百行以上的程序，就不知道该如何下手了。这在初学者中，是一个普遍存在的现象。每个 API 都理解了，单独使用也没有任何问题，可为什么一遇到稍微复杂的题目，需要若干 API 配合使用时便不知所措了呢？原因当然是多方面的，而其中一个非常重要的因素是：目前市场上已有的书籍或教材，大多都只注重 API 原理性的介绍，没有从灵活使用的角度讨论，更没有按照现代程序设计或软件设计的思想，分析各类 API 综合使用的方法。针对这个问题，笔者也才兴起了编写本书的念头。

作为上述问题的解决之道，本书把 Linux 系统常见 API 的介绍，同现代程序设计思想结合起来。具体而言，本书开发了一个 Linux 系统下的、基于 C++的执行体程序库。该程序库

包含了线程和进程的创建模型、同步模型以及通信模型。本书以此程序库为线索，在介绍API的过程中，将结构化的程序设计思想、基于对象的程序设计思想、面向对象的程序设计思想、基于模板的面向对象的程序设计思想、基于方面的程序设计思想、基于接口的程序设计思想等6种编程范式结合起来，分析各类API的综合使用方法。例如，在第4章讨论创建进程的fork函数时，将会按照上述6种编程范式，分别对fork函数进行封装；通过该实例，读者既能掌握fork函数的基本使用方法，又能了解6种编程范式思想的异同点，还能自觉按照现代程序设计思想灵活使用它。相信读者结合执行体程序库阅读本书，一定会在Linux系统API综合使用方面有所进步。

阅读本书，读者需要具备两个前提条件。一是会Linux系统的基本操作，二是有一定C++程序设计的经验。全书分成5个章节，其中5.2节至5.4节为段翰聪著，其余为李林著。

第1章入门知识，将会介绍Linux发展史，以及6种编程范式分别实现加法器封装的方法，以达到初步认识这6种范式的目的。

第2章日志的实现，将会首先介绍Linux系统中文件操作的基本方法，然后在此基础之上引出程序库的出错处理，以及日志类的构建方法。在日志类的构建中，第2章将会讨论写日志的效率问题。

第3章线程的封装，将会首先讨论如何创建线程，然后按照多种编程方式对线程的创建进行封装。接着，第3章将会对常见的线程同步方式进行封装，这主要包括了封装互斥量、条件变量，以及在此基础之上封装出类似Windows系统中的事件对象。第3章还将讨论线程间的消息通信机制，封装出消息循环的框架、消息发送机制以及名字服务。第3章结束时，将会得到执行体程序库的一个基本框架。

第4章进程的封装，同第3章类似，首先会讨论进程的创建方法，然后按照6种编程范式对进程创建进行封装。这是本书的第三次讨论使用6种范式封装同一个事物。在讨论进程创建封装时，还将讨论文件描述符继承等若干细节问题。接下来，第4章将讨论进程同步的封装。在这部分内容中，将会封装记录锁、共享存储、共享互斥量、共享条件变量，以及共享事件对象等。在进程通信的讨论中，将会介绍命名管道的使用细节，并在第3章基础上将基于命名管道的消息通信封装入执行体程序库。

第5章执行体程序库的使用，将会讨论如何把程序库打包成一个静态库；然后再给出一个英文文章词频统计程序的实现，以验证程序库的有效性。

另外，读者可以登陆http://222.197.182.144或http://www.uestcp.com.cn即电子科技大学出版社网站，在“资源下载”列表中获取随书代码。随书代码的运行环境，均为64位ubuntu操作系统。

笔者相信通过本书的阅读，读者不仅能掌握Linux系统API的基本使用方法，而且还能自觉地按照现代程序设计思想灵活运用这些API，使自己从程序员到架构师的道路上迈出坚实的一步。让我们开始Linux程序设计之旅吧。

目 录

第1章 入门知识

现在让我们开始 Linux 系统程序设计之旅吧。不过在这之前，有些必要的东西还是需要事先铺垫下的。比如 Linux 的发展史。学了半天 Linux 编程，居然不知道它的来龙去脉，貌似有点说不过去，还是介绍下吧。序言里面说了 6 种编程范式，很多人可能已经有点手痒了。本章就给大家露个脸吧，算是个初见面。好了闲话休提，上正菜。

1.1 来龙去脉

笔者刚上大学那会儿，国内一些公司已经开始关注 Linux 操作系统了。记得有次家庭聚会，有位工作中偶有接触 Unix 的亲戚问笔者："最近 Linux 很火，给我介绍下它吧。"正好前不久，笔者看了一篇介绍 Linux 发展的文章；于是，就依葫芦画瓢地复述了一遍。亲戚听后直说，"不错，不错，果然是专家。"汗，其实那时还从来没在 Linux 上面写过程序。真没想到，Linux 发展史还有这用处。看来不管怎么样，为了这个"专家"或者"砖家"的名头，Linux 发展史是不得不介绍下了。

从何说起呢？当然还是从头说起。笔者在所在高校承担了一门课程——《Linux 环境高级编程》。该课程选用的教材，就是著名的《Unix 环境高级编程》一书。记得有次上课，一位同学询问笔者："我们课程不是 Linux 吗，怎么用 Unix 的书？"当然这其中的原因有很多，但有一条是不容忽视的。那便是，Unix 实为 Linux 的源头。所谓正本清源，那么下面就从源头说起了。

回到美苏争霸的时代。1957 年，前苏联发射了第一颗人造卫星。这个消息一出来，美国人坐不住了。当时的艾森豪威尔总统，下令成立高等研究计划署（ARPA，Advanced Research Projects Agency），并投入了巨额的科研经费，以推动包括电子计算机在内的各项科学技术的发展。ARPA 貌似有点印象吧，学过计算机网络课程的，都应该知道 ARPANET。在这样一个背景下，20 世纪 60 年代，麻省理工最先实现了一个兼容分时系统（CTSS，Compatible Time-Sharing System）。在这个系统基础之上，1963 年麻省理工采用 IBM 的大型计算机，连接了近 160 台终端机，并可以让 30 位用户同时使用各类资源。但是到了 1965 年，这套系统就不堪重负了，麻省理工决定开发一个更为大型的分时计算机系统，即 MULTICS（MULTiplexed Information and Computing System）。MULTICS 计划连接 1000 部终端机，允许 300 位用户同时使用计算资源。麻省理工联合了通用电子公司、贝尔实验室共同开发这个系统。到了 1969 年，虽然经过四年的奋战，但是 MULTICS 依然没有达到预先设定的目标，这直接导致了贝尔实验室的退出。与此同时，该实验室的两位大神肯·汤普逊（Ken Thompson）和丹尼斯·里奇（Dennis Ritchie）正在使用 Fortran 语言，把原本在 MULTICS 中开发的一款名为太空旅游的游戏，移植到 GECOS System 上——看来游戏事业确实昌盛，60 年代就有了。但是 GECOS System 的 CPU 实在太贵了，两位大神又以 PDP-7 迷你计算机取代 GECOS 系统。在整个移植过程中，两位大神开发了一套包含文件系统、进程子系统、

实用程序的操作系统。而当时贝尔实验室刚好退出 MULTICS 计划，这两位大神玩笑似的便把这套操作系统称为 UNiplexed Information and Computing System，UNICS，简称为 UNIX。[①] 这个小故事再次印证了那一句出自《增广贤文》的古训："有心栽花花不开，无心插柳柳成荫。"麻省理工联合了好几家单位，费尽心机，最终 MULTICS 还是失败了；而我们的两位大神呢？就因为一个玩笑便开创了 Unix，乃至 Linux 几十年的佳话。

到了 20 世纪七八十年代，Unix 得到了巨大的发展。特别是在 1973 年，用 C 语言重写了 Unix 源程序，这个就是 Unix 的首个正式版本 V5。之后，UC Berkeley 在 V6 的基础上，发布了 1 BSD（Berkeley Software Distribution）；之后又发布了 2 BSD、3 BSD、4.1BSD、4.2BSD 等。初期对 Unix 不感冒的 AT&T，也发布了自己的 Unix 版本，System III、System V、SVR 2（System V Release 2）等。当然，Unix 的发行版本还有很多，例如 Solaris，这里不再一一列举。

追本溯源了半天，现在也该回到我们的正题上来了。Unix 的高昂价格，使得很多 PC 小用户望而却步，全世界的计算机爱好者们，都希望开发一个自由的操作系统。这当然包括了 Linux 的创始者林纳斯·本纳第克特·托瓦兹（Linus Benedict Torvalds）。1991 年，时年 21 岁的芬兰大二学生林纳斯，同很多爱好者一样，正在钻研一个名为 Minix 的操作系统。由于该操作系统是为了教学而开发的，因此它各方面的能力都极其有限。即便如此，通过对 Minix 的学习，也激发了我们的主人公和广大的爱好者去开发一个操作系统的热情。然而开发操作系统不是一件简单的事情，还有许多辅助的工作要做，例如构建编译链接的环境、调试的环境等。而这一系列的难题，都被当时的 GNU 计划解决了。GNU（GNU's Not Unix 的递归缩写）计划，是 1984 年创办，旨在开发一个类似 Unix，但是自由软件的完整操作系统，即 GNU 系统。在该计划下，先后开发了 emacs、bash shell、gcc、gdb 等软件，为林纳斯的开发工作提供了有力的支持。在 Linux 的发展道路上，还有一个十分重要的标准对其起到了指导性的作用，那就是 POSIX（Portable Operating System Interface for Computing Systems）标准。这个标准是根据 Unix 的实践情况，对系统的 API 接口原型进行了规范，这也就使得 Linux 与许多 Unix 系统，在应用层的层面上兼容。这也回答了为什么笔者上 Linux 的课，却用 Unix 的教材。Linux 的发展，不是林纳斯一个人的功劳。所谓众人拾柴火焰高，正是由于全球众多计算机骇客通过 Internet 的加入，才使得 Linux 发展到了今天。有点乱了。总结一下，Linux 的诞生和发展主要有五大支柱：Unix、Minix、GNU、POSIX、Internet。经过二十多年的发展，Linux 家族可谓枝繁叶茂，拥有若干发行版本，例如 Ubuntu、SUSE、Fedora、Debian、Red Hat 等。[②]

这里有个小补充，是关于"Linux"发音的。很多人都把"Linux"这个词读作[`linju:ks]，这是不对的；正确的念法是['li:nəks]。你看，当你在别人面前"哩呐克斯"一下，多专业啊！

1.2 范式初见面

相信大家还记得本书要讨论的多种编程范式吧。没错，在序言里面，我们就提到了若干

① 上述内容主要参考了《Unix 简史》一文，特此说明。

② 上述内容主要参考了《浅析：关于 Linux 操作系统的发展史》一文，特此说明。

种范式：结构化的程序设计思想、基于对象的程序设计思想、面向对象的程序设计思想、基于接口的程序设计思想及其模板实现、面向方面的程序设计思想。这些范式各是什么意思呢？别着急，我们以一个加法器的例子来逐一说明。

1.2.1 结构化的程序设计思想

我们现在需要实现一个加法器：在这个加法器中，已经保存了被加数，现在需要传递加数给这个加法器，以让其返回加法计算结果。很简单吧。嗯？忘了什么是被加数？唉，怎么和我一样，打小就分不清加数和被加数。记住了，加号左边的是被加数，右边的是加数。好了，需求分析做完了，现在该来说说怎么实现这个加法器了。如果你是一个习惯于传统 C 语言的结构化程序设计思想的拥趸，我想你的第一反应多半是：简单，用一个结构体来保存被加数，然后再外带一个加法函数就行了嘛。代码清单 1.1 就表达了这种想法。

代码清单 1.1

```
1  struct SLAugend
2  {
3     int iAugend;
4  };
5
6  int Add(struct SLAugend *pSLAugend, int iAddend)
7  {
8     return pSLAugend->iAugend + iAddend;
9  }
```

在代码清单 1.1 中，结构体 SLAugend，顾名思义，即保存了加法器的被加数；具体而言，就是由其字段 iAugend 保存。第 6 行至第 9 行给出了加法函数的定义。该函数接收两个参数，一是 SLAugend 结构体的指针，二是加数 iAddend。实现方法很简单，自己看代码吧。

这就是一个十分典型的结构化思想的实现实例，但故事还没有完。老板来了！老板可能会对你说，这个加法器要修改一下：现在需要给被加数添加一个权重值；但是以前的加法器仍需保留，因为还有一部分代码会使用它。老板真讨厌！不是吗？每当你要完成任务时，总有新的事情分给你，工作永远都做不完啊[①]。没办法，拿人家的手短，吃人家的嘴软，继续当“码农”吧。“码”的思路还是那样：既然还有一部分代码要用老的加法器，那么老加法器我们还是要保留的；这样一来，就可以按照代码清单 1.1 的思路，开发新的加法器了。具体的方法，可参见代码清单 1.2。

① 老板就像 Erlang 中的监控树，你一旦清闲，总能及时发现你。Erlang 不错，天生的分布式高手，推荐给同学们研究研究。

代码清单 1.2

```
1  struct SLWeightingAugend
2  {
3     int iAugend;
4     int iWeight;
5  };
6
7  int WeightingAdd(struct SLWeightingAugend *pSLWeightingAugend, int iAddend)
8  {
9     return pSLWeightingAugend->iWeight * pSLWeightingAugend->iAugend + iAddend;
10 }
```

代码清单 1.2 的思路同代码清单 1.1 是完全一致的，不同的只是结构体和函数的名称。很显然，SLWeightingAugend 保存了被加数和它的权重，而 WeightingAdd 则是带权重的加法函数。看到这里，可能一些读者会觉得这些结构体名称、函数名称、变量名称等，是不是太长了？不不不，长的还在后面。恕笔者卖个关子，先不解释为什么取较长的名称，先来谈谈对待代码注释的态度。笔者还记得大一的时候上第一门计算机专业课 Pascal 语言，授课的老师就说了写程序一定要写注释；到后来与一些公司进行项目合作时，某公司就要求注释的行数要占到代码行数的 20%以上。大有不写注释，就是冒天下之大不韪之势。那么笔者对待注释的态度是什么呢？旗帜鲜明地表示：坚决反对写注释！没错，你没有看错，就是反对写注释！这是在唱反调吗？如果你这样想，那就是了。

为什么不写注释呢？第一个原因是不少人的注释写得很无聊，例如“定义了一个整型变量”。这种情况，在一些对注释行数有要求的公司屡见不鲜，这就叫上有政策下有对策。当然这个原因不是主要的，更重要的是第二个原因：很多时候，代码和注释出现不一致的情况。为什么会这样呢？造成这种情况的原因很多，其中一个十分重要的因素就是赶进度。一开始时间不紧张的时候，通常程序员都能够保持良好的习惯，为代码配上注释。到了项目中后期，进度紧张了，客户又提出了一大堆需求方面的变更。这个时候老板催你，项目经理催你，测试人员催你。此时你的选择只有加班加点把代码改好，谁还有空管注释，觉都睡不够！ICT 行业加班太常见了，特别是一些小公司。很显然，像这种注释和代码表达不同意思的情形，不但无益，反而有害，应将之坚决摒弃。

那么不写注释，别人能看懂你的代码吗？能！首先，笔者基本使用的都是面向对象的语言，如 C++等，它们具有很好的封装性。并且笔者在编程实践时，严格按照职责单一的原则，这就导致了每个类都很小。通常，很多类的代码行数都在两三百行之内。再加上，每个类都对应一个.h 文件和一个.cpp 文件，因此，不会存在一个文件上千行，而导致阅读者无从下手的情况。第二，回到刚才讨论的名称较长的问题。之所以要给类、函数、变量等取长的名称，原因就在于让读者能够顾名思义。比如 WeightingAdd，一看就知道是带权重的加法运算方法；而 SLWeightingAugend，显然就是带权重的被加数的结构体（笔者习惯使用 SL 作为结构体的前缀，而 CL 作为类的前缀）；pSLWeightingAugend，则是指向 SLWeightingAugend 结构体实例的一个指针。像这种一看名称就能理解含义的情况，还需要写注释吗？当然不了！关于这种情况有个专业术语，即代码自注释。很显然即使再赶进度，修改代码的时候保持其

自注释性也不会怎么费事，特别是在有代码自动补全工具的帮助下，更是轻松惬意了。

实际上不写注释只是笔者在这里一个吸引眼球的说法。有些类似于说明文档的东西，还是需要写的。在笔者的编程实践中，主要体现在三个方面。一是重要的类需要一个总体性的说明，二是复杂的算法需要进行说明，三是类之间的关系需要说明。笔者认为，能不能让别人和自己看懂代码，最关键的是第三个方面。这个可以通过类图、类级别的时序图等工具加以辅助。

我们还是回到加法器的例子吧。嗯，这一节有点像散文了，不过笔者还是做到了形散神不散，总算又回到加法器了。代码清单 1.1 和 1.2 给出了两个版本的加法器实现，但是故事还没有完，因为有一部分老代码需要使用带权重的加法运算。显然没有什么好办法，只有修改这些老代码，让它们使用带权重的加法器了。

好了，现在我们分析一下按照结构化程序设计思想实现的加法器有什么缺陷？学过面向对象的同学肯定一口就能说出来，数据和操作这个数据的函数或方法没有封装在一起。不错，书没有白背。确切一点就是，这个加法器没有把被加数、权重以及操作它们的加法运算封装在一起。这只是缺陷之一。之二呢？因为引入带权重的加法器之后，需要对部分老代码进行修改，显然没有做到代码封闭，即没有实现这一变化点的封装。代码封闭性是什么？面壁吧，自己回去看看面向对象的书。结构化思想的讨论到此为止吧，下面来说说基于对象的方法。

1.2.2 基于对象的程序设计思想

在对象的世界中，任何东西都可以被当成对象。那么按照这个说法，我们需要实现的这个加法器，显然也是一种对象了。用过 C++或者类似面向对象语言的同学，第一反应我想是：编写一个加法器的类，用一个数据成员保存被加数，然后再写一个 public 的加法方法就好了。是的，没错，请看代码清单 1.3。

代码清单 1.3

```
1  class CLAdder
2  {
3  public:
4     explicit CLAdder(int iAugend)
5     {
6        m_iAugend = iAugend;
7     }
8
9     int Add(int iAddend)
10    {
11       return m_iAugend + iAddend;
12    }
13
14 private:
15    int m_iAugend;
16 };
```

代码清单 1.3 给出了类 CLAdder 的实现。关于类的名称，笔者通常习惯于使用名词，就

像这里的 Adder 加法器一样；而数据成员，则以 m_开头。在 CLAdder 类中，成员 m_iAugend 保存了被加数，而 Add 方法则执行简单的加法运算。这里有一点需要说明，就是构造函数前面的那个关键字 explicit。加上该关键字的意图，是为了防止隐式类型转换。具体的内容，可参见相关标准说明。

同样的，故事还没有完，需要实现带权重的加法运算，但是仍有老代码会使用这个 CLAdder 类。没办法了，我们只有再实现一个带权重的加法器。依葫芦画瓢，CLWeightingAdder 类出炉了，请见代码清单 1.4。CLWeightingAdder 类使用了两个数据成员，以分别保存被加数和权重，并提供了带权重的加法方法 Add 函数。

代码清单 1.4

```
1  class CLWeightingAdder
2  {
3  public:
4    CLWeightingAdder(int iAugend, int iWeight)
5    {
6      m_iAugend = iAugend;
7      m_iWeight = iWeight;
8    }
9
10   int Add(int iAddend)
11   {
12     return m_iAugend * m_iWeight + iAddend;
13   }
14
15 private:
16   int m_iAugend;
17   int m_iWeight;
18 };
```

现在我们来比较一下基于对象方法和结构化方法的差异。笔者相信，你至少能说出一条。对了，那就是封装，数据和操作这个数据的方法被封装在了一起。那么在这里，便是被加数或权重、加法运算或带权重的加法运算，被封装在了类 CLAdder 或类 CLWeightingAdder 中。只要实例化出一个对象来，你就能使用其加法方法。关于什么是封装、什么是对象，为什么需要封装、为什么需要对象，笔者相信很多人都能背出一大堆概念和理论来。这在笔者的课堂上，已经见证过多次了。同学们的背功，确实厉害。但是当笔者询问封装、对象这些概念或者理论是怎么来的时候，能回答出来的同学却不多。那么这些概念到底是怎么产生的呢？是哪个做理论研究凭空想象出来的吗？还是哪个为了写论文，脑袋一拍臆造出来的？当然，这些都不是。那么到底来自于哪里呢？来自于编程实践。

让我们回头来看看代码清单 1.1 吧。一个实际项目，往往都会有很多的.h 文件和.c 文件，以有效组织源代码。按照这个思路，通常我们会把结构体 SLAugend 的定义放在一个.h 文件中，而 Add 函数的定义和声明，则分别放在一个.c 文件和.h 文件中。这会给加法器的使用者和实现者，带来一点小麻烦。什么麻烦呢？加法器的使用者，需要老是不停地查看 SLAugend 的头文件和 Add 函数的头文件，因为他需要使用结构体的成员，调用 Add 函数；

而实现者，也需要不断地查看 SLAugend 的头文件。如果结构体成员、操作它的函数比较多的话，那这个麻烦就更大了。怎么办呢？很自然的想法是，干脆把 SLAugend 结构体和 Add 函数放在一起好了。而且，这样管理起来也方便。于是就有了代码清单 1.5-a。

代码清单 1.5-a

```
1  struct SLAdder;
2
3  typedef int (*FUNC_ADD)(struct SLAdder *, int);
4
5  struct SLAdder
6  {
7     int iAugend;
8     FUNC_ADD pFuncAdd;
9  };
10
11  int Add(struct SLAdder *pSLAdder, int iAddend)
12  {
13     return pSLAdder->iAugend + iAddend;
14  }
```

在代码清单 1.5-a 中，我们使用结构体 SLAdder 代表加法器。它包含了两个字段。一是被加数 iAugend，另一个则是函数指针，其类型如第 3 行所示。该函数指针，需要指向实际的加法运算函数。显然这样一来，就把代码清单 1.1 中的 SLAugend 结构体和 Add 函数放在了一起，只是结构体的名称变为了 SLAdder 而已。那么如何使用这个版本的加法器呢？很简单，请看代码清单 1.5-b。

代码清单 1.5-b

```
1  struct SLAdder adder;
2  adder.pFuncAdd = Add;
3
4  adder.iAugend = 3;
5
6  cout << adder.pFuncAdd(&adder, 5) << endl;
```

在代码清单 1.5-b 中，第 1 行实例化了 adder 出来。然后，将 Add 函数的地址，赋予了 pFuncAdd 字段，并进行了被加数的初始化操作。而第 6 行，则通过 pFuncAdd 字段调用了加法函数。貌似第 6 行有点像成员函数的调用了。adder 就是对象，pFuncAdd 就是成员函数了，很像吧！随着这种组织结构体和它的操作函数的方式的广泛应用，程序员很快就发现了问题。什么问题呢？太麻烦了！只要有一个函数指针的字段，通常就得定义一个函数指针类型，就得在结构体的每一个实例创建后完成函数指针的赋值操作，即绑定某个具体的函数。确实太麻烦了，还容易弄出错，特别是在程序员个个都是懒人的情况下。那怎么办呢？交给编译器做算了。把函数的定义或声明，都放在结构体里面去，让编译器自己去解析；结构体实例创建后，让编译器自动绑定函数。

另外，再请大家注意 pFuncAdd 调用的第一个参数，addr 的地址。为什么需要这个呢？因为 Add 函数要访问 adder 的被加数字段，所以必须把 adder 的地址传给该函数。而刚才又提到了，这个 adder 可以类比成对象，那么对象的地址是什么呢？成员函数，对象地址，你能想到什么呢？对了，this 指针。这第一个参数，不就是起到了 this 指针的作用吗?按照这种方式组织结构体和操作结构体的函数，很显然，每次函数调用都需要程序员手工地把结构体实例的地址写上去，而这个实例实际上已经出现在了“.”号的前面。程序员往往是懒惰的，就这点小重复也想利用，能省点力气就省点。那怎么办呢？交给编译器做吧，让编译器自动地帮我们补充上对象的地址，无论是在调用中还是在函数体内。

事情貌似变得很美妙了。结构体里面有数据，有函数，函数里头还隐藏了个 this 指针。嗯，这是个新东西，是个创新点，可以发论文了，不过先得取个响亮的名字。不错，类、对象、封装，高深的词汇！是的，对象的思想就这样诞生了。当然，前面讨论了这么多，只是对象产生的一个例子，一个侧面而已。但是从这个例子，也多少可以看出计算机领域中技术和科学的关系来。笔者认为，在计算机领域，往往是技术实践走在前面，而后科学理论跟上解释。也即是说，技术带着理论在走。

东拉西扯这么多，散文也要回到中心思想的，还是继续分析基于对象的方法，同结构化方法的差异。前面已经讨论到了封装：基于对象的方法能够实现封装，而这正是结构化方法所欠缺的。1.2.1 节讨论结构化程序设计思想时分析道：当新增带权重的加法器时，破坏了代码的封闭性，即没能封装这一变化点。那么基于对象思想，也就是代码清单 1.3 和 1.4 所示，能封装这一变化点吗？显然也不行。原本使用 CLAdder 类的老代码，若要改成使用 CLWeightingAdder 类，则必须修改对象创建时的类型、参数传递时的类型等。这显然破坏了代码的封闭性。由此可见，基于对象的方法，仅在结构化方法的基础上，弥补了封装的缺陷，但依然遗留了变化点不能很好封装的问题。

1.2.3 面向对象的程序设计思想

基于对象和面向对象的区别是什么呢？笔者相信很多人都能背出一大堆概念来，但是从技术实现角度来讲，很简单，面向对象多了继承和虚函数。而这又会带来什么好处呢？多的不说，先上代码，我们有代码有真相。

代码清单 1.6-a

```
1  class CLAdder
2  {
3  public:
4      explicit CLAdder(int iAugend)
5      {
6          m_iAugend = iAugend;
7      }
8
9      virtual ~CLAdder()
10     {
11     }
12
```

```
13    virtual int Add(int iAddend)
14    {
15        return m_iAugend + iAddend;
16    }
17
18  protected:
19      int m_iAugend;
20  };
```

代码清单 1.6-a 同代码清单 1.3 相比大致相同，其区别主要是下面几个方面。第一，Add 函数变成了虚函数。这个意图很明显，就是希望派生类重写它。第二，m_iAugend 数据成员访问权限变成了 protected，这也是保证派生类能访问到它。第三，就是增加了一个虚析构函数。为什么要增加这个虚析构呢？没有它，会有什么问题呢？笔者发现很多同学对这个问题都不太了解，其实这应该是教 C++的老师讲的。上初中的时候，初中老师总是抱怨小学老师该讲的没讲；到了高中，高中老师总是抱怨初中老师该讲的没讲；到了大学，大学老师又抱怨中学老师该讲的没讲；看来现在笔者也只有抱怨下 C++老师了，呵呵，这个传统也得以延续。思维发散完了，回来说说这个虚析构。如果 CLAdder 没有虚析构函数，当 delete 一个类型为 CLAdder 的对象指针，但该指针实际指向的是其派生类对象时，则派生类的析构函数将不会被调用。

面向对象版的 CLAdder 写好了，但是事情并没有完，因为老板说了还需要写带权重的加法运算。不过思路清晰多了，让 CLWeightingAdder 从 CLAdder 派生吧。没错，请看代码清单 1.6-b。

代码清单 1.6-b

```
1   class CLWeightingAdder : public CLAdder
2   {
3   public:
4       CLWeightingAdder(int iAugend, int iWeight) : CLAdder(iAugend)
5       {
6           m_iWeight = iWeight;
7       }
8
9       virtual ~CLWeightingAdder()
10      {
11      }
12
13      virtual int Add(int iAddend)
14      {
15          return m_iAugend * m_iWeight + iAddend;
16      }
17
18  protected:
19      int m_iWeight;
20  };
```