Broadview®
www.broadview.com.cn

IBM China Development Laboratories Series
IBM中国开发中心作品系列

C++应用程序性能优化

冯宏华 徐莹 程远 汪磊 等编著

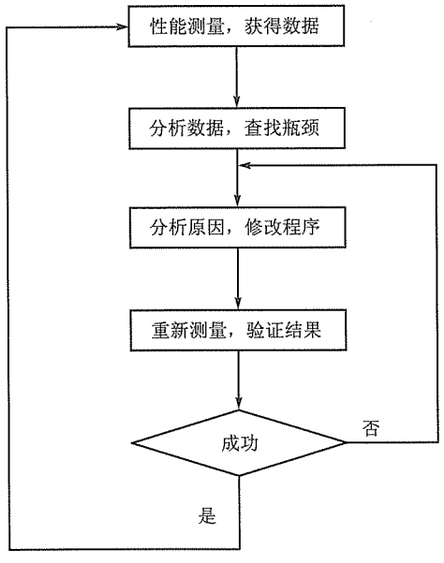
電子工業出版社
PUBLISHING HOUSE OF ELECTRONICS INDUSTRY
HTTP://WWW.PHEI.COM.CN

前　言

在计算机发展的早期阶段，硬件资源相对而言是非常昂贵的。不论是CPU时间，还是内存，都给编程人员设置了很大的限制。因此，当时程序对运行性能和内存空间占用的要求是非常严格的。很多开发人员为了减少1%的CPU运行时间或者减少几十个，甚至几个字节而努力。随着计算机技术的快速发展，硬件资源变得相对便宜。因此有的观点认为在开发软件时，软件的性能优化将不再重要，硬件将解决性能问题。但事实上，这种观点是相对片面的。的确，硬件的发展解决了部分软件的性能问题。但随着硬件计算能力的提高，人们对软件功能的要求也越来越高。当今的软件功能越来越复杂，给用户的界面和操作体验也越来越智能和友好，这些需求带来的软件性能上的要求是硬件不能完全解决的。很多实际的项目证明，如果在开发软件时不重视性能的优化，最终虽然实现了功能上的要求，但软件的运行效率低下，也不能给用户带来很好的效益。因此，软件的性能优化是计算机软件开发过程中需要一直关注的重要因素。

程序性能优化的过程

从开发过程的角度来看，程序的性能优化流程一般如下图所示。



性能优化的第1步是测量，尤其是规模较大，并且比较复杂的软件系统，测量性能数据是进行性能优化的基础。有了真实的数据，才可以进行第2步，即分析数据，从而找到系统真正的瓶颈所在。毫无疑问，优化应该是针对系统的性能瓶颈进行，而找到性能瓶颈应该是建立在真实性能数据的基础上，而不能是主观臆断。现在有很多工具可以辅助进行性能测量和数据分析，本书也会介绍一些这方面工具的使用方法和实践经验。

进行性能优化的核心在于第3步，即分析原因，修改程序，这也将是本书重点介绍的部分。程序的性能包括很多方面，常见的有程序的启动速度，运行速度及运行时占用的内存等。而影响这些性能的因素也很多，大致可以分为如下两类。

- 软件编程设计因素：如算法和数据结构的选择，编程语言的使用等。
- 软件系统结构因素：如动态库/静态库的组织、外部数据的存储及网络环境等。

软件编程设计因素可以看做是程序的内在本质，一般来说，也是对软件性能影响较大的因素。只有对编程语言、算法和数据结构有深入的了解，才能分析出原因，并且找出解决性能问题的方法。本书将针对C++语言，深入介绍C++程序性能优化的方法和实例。

软件系统的结构因素可以看做是程序的外在形式，它们一般和操作系统紧密相关。尤其是现在的软件系统，由于功能复杂，大都采用组件形式，以最大限度地提高可复用性。因此，一般会包含一些动态库/静态库，这些库的组织也会影响到软件系统的性能。本书将针对Windows和Linux介绍动态库和静态库的基本知识及其对软件系统性能的影响。需要指出的是，上面这个优化的过程需要在软件开发的整个过程中不断地迭代进行。而且开始得越早，出现的性能问题越容易解决。

本书的结构

本书主要针对C++程序的性能优化，由4篇组成。第1篇介绍C++语言的对象模型，与性能有关的语言特性及一些数据结构的性能，本篇是C++程序优化的基础。

第2篇主要介绍C++程序内存使用的优化。内存无疑是影响程序性能的重要因素，很多程序由于没有采用正确的方法分配和使用内存，不仅占用内存较多，而且运行效率不高。在本篇中将结合一些操作系统的内存管理机制介绍如何在特定的平台下进行内存优化。此外，还将深入介绍C++语言管理动态内存的机制和方法，并介绍一个常用的内存管理方法，

即内存池的实现。

第 3 篇介绍程序启动性能的优化。程序的启动性能不仅受到软件编程设计因素的影响，也会受到系统结构因素的影响，尤其是动态库的影响。本篇将首先介绍动态库的基本知识，然后介绍一些程序启动性能优化的具体方法。

“工欲善其事，必先利其器”，好的工具会大大提高程序性能优化的效率。第 4 篇将介绍 3 类性能工具，即内存分析工具、性能分析工具和 I/O 检测工具，它们是性能测量和分析的利器。

本书适用于有一定 C++开发经验的开发人员，也可以作为高等院校相关专业师生的参考书。

致　谢

本书是集体创作的结晶，在此感谢大家出色的协作精神。本书的写作也占用了大家大量的休息、娱乐，以及和家人在一起的时间，所以在此感谢作者们家人的理解和支持。同时，在成书的过程中与许多人的关怀、鼓励和支持密不可分，其中包括CDL总裁Josephine，律师Andrew，Director Dennis，资深经理Debbi 和Cindy，研发经理阎小兵和扈晓炜等，在此表示感谢。最后，特别感谢出版社的有关领导、协调人员及编辑，没有他们的支持和参与，本书的出版是不可能的。

由于时间仓促、水平有限，书中难免有许多不妥、甚至错误之处。在此敬请读者不吝指出，我们将愿意与读者共同探讨，并不胜感激。

目　录

第 1 篇　C++程序优化基础

第 2 篇 内存使用优化

第 3 篇 应用程序启动性能优化

第 1 篇　C++程序优化基础

如果说 C++开发人员对于程序性能的追求，丝毫不弱于沙漠中迷途路人对绿洲的渴望，相信很多人会表示赞同（尤其是那些几近偏执的完美主义者）。从使用者的角度来看，软件的可用性始终都是第一位的。即使程序的功能再强大，缓慢的执行速度或是庞大的资源消耗都会让人望而生畏。不幸的是，开发高性能的 C++程序并不是一件简单的工作，起码不会比在沙漠中找水更容易（从事性能提升的 C++工作者也许有相似体会）。它需要开发人员有坚实的 C++编程基础、深厚的数据结构知识，以及宽阔的知识面。当然这有些危言耸听，不过起码说明了本书存在的必要性。

如果你希望开发出高效的 C++程序，需要做的第 1 件事就是熟悉所使用的这门语言及其相关的各种基本数据结构，这正是本篇的目的（当然，如果你已经是一个熟练的 C++开发人员，那么可以直接跳过本篇）。深入了解 C++对象的内存布局、分配及释放方式，以及生命周期，能够帮助你使用正确的编程方法优化生成的代码的大小，并且提高内存使用效率。例如，在什么情况下禁止在堆中产生某种对象，而在什么情况下确保一个对象在堆中产生，以及这样的方法对程序性能提升有何帮助。

本篇详细介绍影响 C++程序性能的主要特性，构造/析构函数关乎对象的资源是否被正确地分配和释放；继承和虚函数是 C++的核心之一，而继承同样也是影响 C++程序性能的一个重要因素；临时对象是比较容易被人忽视的性能影响因素，但其确确实实非常重要；inline 函数是把双刃剑，适当使用能够降低函数的调用开销，滥用则有可能适得其反。

另一方面，本篇还从各种常用操作的角度（遍历、插入、删除、排序及查找），对几种主要数据结构（数组、链表、哈希表及二叉树）的时间及空间复杂度进行详细分析。同时，读者可以通过一个详细的程序实例分析动态数组，即在大型软件系统中很常用的一种数据结构。

总之，通过阅读本篇，读者将熟悉在何种应用环境下使用最佳数据结构，并且深入理解各种 C++语言的对象模型和性能特性。

第 1 章　C++对象模型

对象模型是面向对象程序设计语言的一个重要方面，它会直接影响面向对象语言编写程序的运行机制及对内存的使用机制，因此了解对象模型是进行程序优化的基础。本章将首先介绍一般意义上程序中的数据在内存中的分布，以及程序使用的不同种类的内存等基本概念。然后介绍 C++语言中对象的生命周期，以及 C++对象的内存布局。只有深入了解了 C++对象模型，才能避免程序开发中一些不易察觉的内存错误。从而改善程序的性能，提高程序的质量。

1.1 基本概念

1.1.1 程序使用内存区

一般而言，计算机程序由代码和数据组成，这两个部分也是影响一个程序所需内存的重要因素。代码是程序运行的指令，例如数学运算、比较、跳转，以及函数调用等，其大小通常是由程序的功能及复杂程度决定的。当然，正确地使用程序编写技巧及编程语言的语法特性会优化所生成的代码的大小；数据是代码要处理的对象，也是本章要讨论的重点。

一个程序占用的内存区一般分为如下 5 种。

- 全局/静态数据区。
- 常量数据区。
- 代码区。
- 栈。
- 堆。

显然，程序的代码存储在代码区中。而程序的数据则要根据数据种类的不同，存储在不同的内存区中。在 C++语言中，数据有不同的分类方法，例如常量和变量、全局数据和局部数据，以及静态数据和非静态数据等。此外，还有程序运行过程中动态产生和释放的数据，这些数据存放在不同的数据区中。

- 全局/静态数据区存储全局变量及静态变量（包括全局静态变量和局部静态变量）。
- 常量数据区中存储程序中的常量字符串等。
- 栈中存储自动变量或者局部变量，以及传递的函数参数等，而堆是用户程序控制的存储区，存储动态产生的数据。

下面通过一个例子来说明各种类型的数据在内存中的位置：


```
#include <stdio.h>
#include <stdlib.h>

int nGlobal = 100;

int main(void)
{
  char       **pLocalString1  = "LocalString1";    // 4 个字节对齐
  const char *pLocalString2   = "LocalString2";
  static int  nLocalStatic    = 100;

  int         nLocal          = 1;
  const int   nLocalConst = 20;

  int        *pNew            = new int[5];        // 16 个字节对齐
  char       *pMalloc     = (char *)malloc(1);

  printf("global variable:                    0x%x\n", &nGlobal);
  printf("static variable:                    0x%x\n", &nLocalStatic);
  printf("local expression 1:                 0x%x\n", pLocalString1);
  printf("local expression (const):           0x%x\n", pLocalString2);
  printf("\n");

  printf("new:                                0x%x\n", pNew);
  printf("malloc:                             0x%x\n", pMalloc);
  printf("\n");

  printf("local pointer(pNew):                0x%x\n", &pNew);
  printf("local pointer(pLocalString2):       0x%x\n", &pLocalString2);
  printf("local pointer(pLocalString1):       0x%x\n", &pLocalString1);
  printf("local variable(nLocal):             0x%x\n", &nLocal);
  printf("local pointer(pMalloc):             0x%x\n", &pMalloc);
  printf("local const variable:               0x%x\n", &nLocalConst);

  return 0;
}
```

在上面的程序中，共定义了 8 个变量，其中 1 个全局变量 nGlobal、1 个局部静态变量 nLocalStatic，以及 6 个局部变量。程序中涉及的数据有字符串常量和动态申请的内存区。

在 Windows XP 中用 Visual Studio .Net 2003 编译运行，下面是程序的输出：

```
global variable:                    0x409040
static variable:                    0x409044
local expression 1:                 0x4070d8
local expression (const):           0x4070e8

new:                                0x372aa8
malloc:                             0x372ac8

local pointer(pNew):                0x12fecc
local pointer(pLocalString2):       0x12fed0
local pointer(pLocalString1):       0x12fed4
local variable(nLocal):             0x12fed8
local pointer(pMalloc):             0x12fedc
local const variable:               0x12fee0
```

由输出结果可以看出全局变量（nGlobal）和静态变量，以及局部的静态变量（nLocalStatic）存储在全局/静态数据区中。字符串常量存储在常量数据区，而且是 4 个字节对齐的。pLocalString1 指向的字符串 LocalString1 的长度是 12 个字节，再加上结束符\0，应该是 13 个字节。但在存储 pLocalString2 的字符串 LocalString2 时，却是在 pLocalString1（指向 0x4078d8）的 16 个字节后，从 0x4078e8 开始的。如果 pLocalString1 增加 4 个字符，长度变成 16 个字节。加上结束符\0，共 17 个字节，则 pLocalString2 的地址会从 pLocalString1 的 20 个字节后开始，即 0x4078ec。程序中的其他字符串常量，如 printf()函数中的格式串也存储在常量数据区中。

通过 new 或者 malloc 获得的内存是堆的内存。在上面的程序中，通过 new 申请了 5 个整数需要的内存，通过 malloc 申请了 1 个字节。这里再次遇到了内存边界对齐的问题，即堆上分配内存时按 16 个字节对齐，所以申请 5 个整数共 20 个字节，但占据 32 个字节的内存（0x371ac8~0x372aa8）。再申请 1 个字节时，系统会从 32 个字节后开始分配。

内存对齐方式虽然会浪费一些内存，但由于 CPU 在对齐方式下运行比较快，所以一般对程序性能还是有好处的。除了上面提到的对齐，C/C++程序中的 struct、union 和 class 在编译时也会对成员变量进行对齐处理。开发人员可以通过#pragma pack()或者编译器的编译选项来控制对 struct、union 和 class 的成员变量按多少字节对齐，或者关闭对齐。

图 1-1 所示为上面程序涉及的各种变量、数据及所在的内存区。

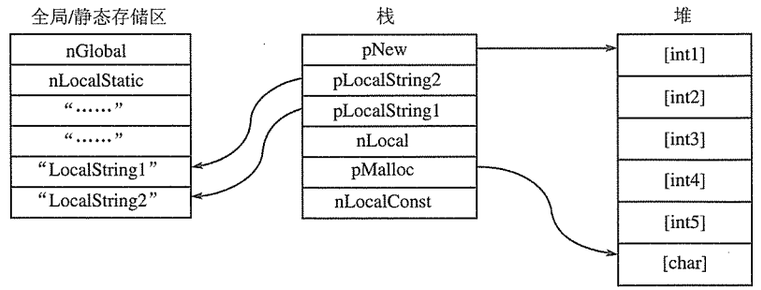


图 1-1 程序涉及的各种变量、数据及所在的内存区

1.1.2 全局/静态存储区及常量数据区

这部分存储区是在程序编译阶段已分配好，在整个程序运行过程中始终存在，用于存储全局变量、静态变量，以及字符串常量等。其中字符串常量存储的区域是不可修改的内存区域，如下面的程序代码会在运行时导致程序退出：

```
char    *pLocalString1  = "LocalString1";    // 指向一个字符串常量
pLocalString[1] = 'a';        // 试图修改不可修改的内存区
```

全局/静态存储区中除了全局变量，还有静态变量，在 C 语言中可以定义静态变量。静态变量在第 1 次进入作用域时被初始化，以后再次进入域时不必初始化。在 C++语言中，可以同样使用静态变量。除此之外，还可以定义类的静态成员变量。众所周知，类和对象是 C++语言中非常重要的概念。定义一个类之后，可以构造这个类的多个对象。如果要在同一个类的多个对象之间共享数据，可以使用全局变量，但这样会破坏类的封装性。因此 C++语言提供了类的静态成员变量，可以用来在类的多个对象之间共享数据。类的静态成员变量存储在全局/静态存储区中，并且只有一份拷贝，由类的所有对象共享。

下面看一个简单的例子，也是静态成员变量常用的一种情形，即计数类的对象：

```
#include <stdio.h>
#include <stdlib.h>

class A
{
public:
            int val;
```

```
        static int nCount;

        A() {nCount++;};
        ~A() {nCount--;};
};

int A::nCount = 0;

int main(void)
{
    A a;
    A b;
    printf("number of A:          %d\n",   A::nCount);
    printf("non-static variable: 0x%x\n",     &a.val);
    printf("non-static variable: 0x%x\n",     &b.val);
    printf("static class member: 0x%x\n",     &a.nCount);
    printf("static class member: 0x%x\n",     &b.nCount);
}
```

在这个例子中，类 A 定义了一个静态成员变量 nCount 用来对 A 的对象计数。为了对比静态成员和非静态成员在内存中的区别，还定义了一个类的成员变量 val。为了简单起见，将类 A 的所有成员的访问权限设置为 public。

在 main()函数中，实例化了 A 的两个对象 a 和 b。下面是程序在 Windows XP 下用 Visual Studio .Net 2003 编译后运行的输出，其中显示了 nCount 中记录了 A 的对象的个数，以及 nCount 和 val 在内存中的位置的不同：

```
number of A:            2
non-static variable:        0x12fee0
non-static variable:        0x12fedc
static class member:        0x4086c0
static class member:        0x4086c0
```

可以看到，a 和 b 对象中 val 变量的内存地址不同，而静态成员变量 nCount 的地址相同。对比 1.2.1 节中的例子，可以知道，类 A 的每一个对象会有自己的 val 存储空间，是在栈中分配的。而类 A 的所有对象共享一个 nCount 的存储空间，是在全局/静态存储区中分配的。

1.1.3 堆和栈

在 C/C++语言中，当开发人员在一个函数内部定义了一个变量，或者向某个函数传递参数时，这些变量或参数存储在栈中。当退出这些变量的作用域时，这些栈上的存储单元会被自动释放。当开发人员通过 malloc 来申请一块内存或者通过 new 来创建一个对象时，申请的内存或者对象所占的内存在堆上分配。开发人员需要记录得到的地址，并且在不需要时负责释放这些内存。

在 1.1.1 节的例子中，通过 new 在堆上申请了 5 个整数所需的内存，将获得的地址记录在栈上的变量 pNew 中。然后用 malloc 在堆上申请了 1 个字节的内存，将获得的地址记录在栈上的变量 pMalloc 中。

```
int     *pNew           = new int[5];           // 16 个字节对齐
char    *pMalloc    = (char *)malloc(1);
```

需要注意的是，在 main()结束时，pNew 和 pMalloc 自身是栈上的内存单元，会被自动释放，而它们指向的内存是堆上的。虽然指向它们的指针已经不存在，但它们不会被自动释放，因此就造成了内存泄漏，这也是在使用 C/C++时需要非常注意的地方。通过 malloc()获得的堆上的内存需要用 free()来释放，而通过 new 获得的堆上的内存，则要用 delete 来释放。

既然栈上的内存不存在泄漏的问题，而堆上的内存容易引起内存泄漏，为什么还要用堆上的内存？这是因为很多应用需要动态地管理数据，一个简单的例子就是链表。当需要为链表新增节点时，就无法使用栈上的对象，需要在堆上申请内存并创建节点。此外栈的大小有限制，占用内存较多的对象或数据只能分配在堆上。

除了上面所说的堆和栈关于使用和释放上的不同，二者还有如下方面的区别需要注意。

（1）大小：一般说来，一个程序使用栈的大小是固定的，由编译器决定。例如，Visual Studio 2003 栈大小的默认值是 1 MB。当然开发人员可以通过编译选项来指定栈的大小，但通常栈都不会太大。例如，当采用 Visual Studio 2003 编译默认值时，下面的程序代码会在运行时出错，原因是栈溢出：

```
int main()
{
    ……;
    char buf[1024*1024]; // 声明一个栈上的 1 MB 数组，栈溢出
```

```
    ……;
    return 0;
}
```

而堆的大小一般只受限于系统有效的虚拟内存的大小，因此可以用来分配创建一些占用内存较大的对象或数据。

（2）效率：栈上的内存是系统自动分配的，压栈和出栈都有相应的指令进行操作。因此效率较高，并且分配的内存空间是连续的，不会产生内存碎片；而堆上的内存是由开发人员来动态分配和回收的。当开发人员通过 new 或者 malloc 申请堆上的内存时，系统需要按一定的算法在堆空间中寻找合适大小的空闲堆，并修改相应的维护堆空闲空间的链表，然后返回地址给程序。因此效率比栈要低，此外还容易产生内存碎片。

图 1-2 所示为一个内存碎片的例子。开始时，程序在堆上申请了 5 个 100 个字节大小的内存块，然后释放了其中不连续的两个。此时当需要申请一个 150 个字节的内存块时，则无法充分利用刚才释放的两个小块内存。如此可见，连续创建和删除占用内存较小的数据或对象时，很容易在堆上造成内存碎片，使得内存的使用效率降低。

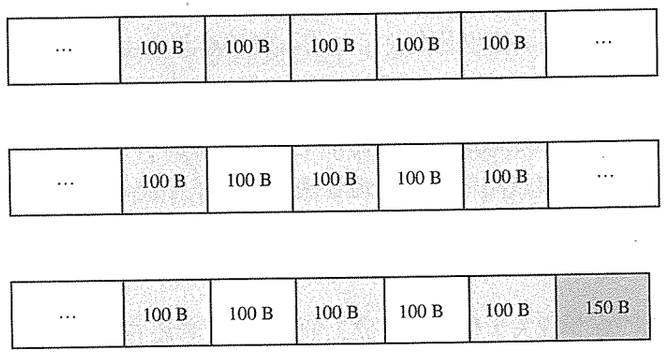


图 1-2　一个内存碎片的例子

1.1.4　C++中的对象

毫无疑问，类和对象是 C++非常重要的概念。从 C++对象模型的角度来说，对象就是内存中的一片区域（C++标准，1.8: An object is a region of storage），因此对象在内存中的位置也符合上面的介绍。根据 C++标准，一个对象可以通过定义变量，或者通过 new 操作符，

或者通过实现来创建（C++标准，1.8: An object is created by a definition (3.1), by a new-expression (5.3.4) or by the implementation (12.2) when needed）。如果一个对象通过定义在某个函数内的变量或者实现需要的临时变量来创建时，它是栈上的一个对象；如果一个对象是定义在全局范围内的变量，则它是存储在全局/静态数据区；如果一个对象是通过 new 操作符来创建时，它是堆上的一个对象。

对面向对象的 C++程序设计而言，程序运行过程中的大部分数据都应该封装在对象中，而程序的行为也由对象的行为决定。因此深入了解对象的内部结构，从而正确地设计和使用对象，对于设计开发高性能的 C++程序非常重要。

1.2　对象的生命周期

对象的生命周期是指对象从创建到被销毁的过程，创建对象时要占用一定的内存，而对象被销毁后要释放对应的内存，因此整个程序占用的内存也随着对象的创建和销毁而动态地发生变化。深入理解对象的生命周期会帮助分析程序对内存的消耗情况，从而找到改进的方法。

前面提到对象的创建有 3 种方式，它们创建的对象的生命周期各有不同。

（1）通过定义变量创建对象：在这种情况下，变量的作用域决定了对象的生命周期。当进入变量的作用域时，对象被创建。而退出变量的作用域时，对象被销毁。值得注意的是静态变量和全局变量，由于全局变量的作用域是整个程序，因此被声明为全局变量的对象在程序调用 main()函数之前被创建。当程序退出 main()函数之后，全局对象才被销毁。静态对象与全局对象类似，虽然静态变量的作用域不是整个程序，但静态变量是存储在全局/静态数据区中，在程序开始时已经分配好。因此声明为静态变量的对象第 1 次进入作用域时被创建，直到程序退出时被销毁。

下面的例子说明了通过定义创建的对象的生命周期：

```
#include <stdio.h>
#include <stdlib.h>

class A {
   public:
```

```
        A() {printf("A created.\n");};
        ~A() {printf("A destroyed.\n");};
};
class B {
   public:
        B() {printf("B created.\n");};
        ~B() {printf("B destroyed.\n");};
};

A globalA;

int foo(void)
{
        printf("\nfoo() ------------->\n");
        A  localA;
        static B localB;
        printf("foo() <-------------\n");
        return 0;
}

int main(void)
{
        printf("main() ----------------------------->\n");
        foo();
        foo();
        printf("\nmain() <-----------------------------\n");
        return 0;
}
```

程序非常简单，用构造函数和析构函数来追踪对象的创建和销毁过程。程序中共定义了 3 个对象，即一个 A 的全局对象 globalA、一个 A 的局部对象 localA 和一个 B 的静态对象 localB，其中 localA 和 localB 的作用域是函数 foo()。通过在 main()函数中调用两次 foo()来看 localA 和 localB 的创建和销毁的不同。

下面是程序的输出（//为注释）：

```
A created.                                  // globalA 被创建
main() ------------------------------>

foo() ------------->
```



```
A created.                                  // localA 被创建
B created.                                  // localB 被创建
foo() <-------------
A destroyed.                                // localA 被销毁

foo() ------------->
A created.                                  // localA 再次被创建
foo() <-------------
A destroyed.                                // localA 再次被销毁

main() <------------------------------
B destroyed.                                // localB 被销毁
A destroyed.                                // globalA 被销毁
```

通过程序运行结果可以看出全局对象在 main()开始前被创建，在 main()退出后被销毁；静态对象在第 1 次进入作用域时被创建，在 main()退出后被销毁；局部对象在进入作用域时被创建，在退出作用域时被销毁。对于静态对象，当程序从来没有进入到其作用域，也不会被创建。

需要提醒的是，C++中的作用域由{和}定义，并不一定是整个函数。在下面的代码中，对象 a 在条件语句成立时才被创建，退出条件语句块后就被销毁，而不是在 foo()调用结束时被销毁：

```
int foo()
{
      int nVal = 0;

      ...

      if (nVal>0)
      {
            A a;
      }
      ...
}
```

（2）通过 new 操作符创建对象：这种情况相对比较简单，但也最容易造成内存泄漏。通过 new 创建的对象会一直存在，直到被 delete 销毁。即使指向该对象的指针已被销毁，但还没有调用 delete，该对象就会一直存在。即占据内存空间，直到程序退出，因此也就造

成了内存泄漏。

下面的程序说明了通过 new 创建的对象的生命周期：

```
#include <stdio.h>
#include <stdlib.h>

class A {
public:
      A() {printf("A created.\n");};
      ~A() {printf("A destroyed.\n");};
};
A* createA(void)
{
      A* p = new A();
      return p;
}

void deleteA(A* p)
{
      delete p;
}

int main(void)
{
      A *pA = createA();        // 第 1 个对象被创建
      pA = createA();           // 第 2 个对象被创建
      deleteA(pA);              // 只销毁了第 2 个对象
      return 0;
}
```

在函数 createA()中用 new 创建了 A 的一个对象，将其地址记录在自动变量 pA 中。然后又调用了 createA()，再次创建一个对象。并将地址记录在 pA 中，此时已经没有指针指向第 1 个被创建的对象。将 pA 传递给 deleteA()时，调用 delete 销毁的是第 2 个被创建的对象。而第 1 个被创建的对象就会一直存在，直到程序退出。而且即使是在程序退出时，这个对象的析构函数不会被调用，即这个对象不是被“优雅”地销毁的。这是内存泄漏中非常常见的一种情况，在后面的章节中还会详细讨论内存泄漏的问题。

（3）通过实现创建对象：这种情况一般是指一些隐藏的中间临时变量的创建和销毁。

它们的生命周期很短，也不易被开发人员察觉。但常常是造成程序性能下降的瓶颈，尤其是对于那些占用内存较多，创建速度较慢的对象。

这些临时对象一般是通过拷贝构造函数创建的，下面首先通过一个简单的例子看看这些临时对象的生命周期：

```
#include <stdio.h>
class A
{
      public:
            A() {printf("A created.\n");};
            A(A& a) { printf("A created with copy\n");}
            ~A() {printf("A destroyed.\n");};
};

A foo(A a)
{
            A b;
            return b;
}

int main(void)
{
            A a;
            a = foo(a);
            return 0;
}
```

在类的定义中增加了拷贝构造函数，用来跟踪对象的创建。简单地从程序上看，似乎只有两个对象被创建，即 main()中的对象 a 和 foo()中的对象 b，但实际上程序的输出如下：

```
A created.
A created with copy
A created
A created with copy
A destroyed.
A destroyed.
A destroyed.
A destroyed.
```

共有 4 个对象被创建，究竟是什么地方多创建了 2 个对象？通过程序的输出不难发现，

多出的两个对象都是拷贝构造的。在上面的程序中不难看出，foo()函数的参数和返回值是通过值传递的。在调用 foo()时，需要把实参复制一份，压入 foo()的栈中。而返回值也要复制一份放在栈中，在 foo()调用结束时，参数出栈就会返回给调用者。因此在调用 foo()时，需要构造一个 a 的副本。由此调用了一次 A 的拷贝构造函数，创建了一个临时对象。同样在返回时，构造一个 b 的副本，即一个临时对象。这两个对象都是在 foo()调用返回后被自动销毁，因为是栈上的对象。

在上面的例子中，只有两个临时对象被创建，似乎不是大问题。但如果 foo()是在一个循环内被调用，那将会有更多的对象被创建和销毁。如果这些对象在构造时需要分配很多资源，如内存等，就会造成资源在短时间内被频繁地分配和释放（假设资源在析构时被正确地释放），甚至有可能造成大量的内存泄漏（假设资源在析构时没有被释放）。

上例中的问题比较好修正，可以通过传递引用的方式来解决，即 A foo(A& a)。这样就不会构造参数的临时对象，但返回时的临时对象仍然存在。为此需要根据实际情况决定如何修改，如可以返回指针或引用，也可以增加一个指针类型参数作为函数的输出。

在实际的 C++软件开发中，除了上面的例子，还会有大量其他类型的隐性临时对象存在，如重载+及++等操作符。对对象进行算术运算时，也会有临时对象，因为这些重载的操作符实际上也是函数。对于这些情况，都要尽量避免不必要的临时对象的出现。

当一个派生类实例化一个对象时，会先构造一个父类对象。同样在销毁一个派生类的对象时，要首先销毁其父类的对象。这个父类的对象是一个隐含的对象，其生命周期和派生类对象绑定在一起，所以不单独讨论。但是要注意的是，如果构造父类对象开销很大，会造成所有子类的构造都有很大的开销。

1.3 C++对象的内存布局

前面两节介绍了 C++对象在整个程序内存空间中的位置及对象的生命周期，本节将介绍单个 C++对象内部的存储结构及 C++一些语法特性的实现方式。了解这些结构及内部实现，对于正确地设计类，选用合适的实现方法是必不可少的。需要注意的是，C++对象的内部结构及实现和编译器紧密相关，不同的编译器可能会有不同的实现方式。本节介绍的主要是 Windows 下 Visual C++及 GNU 的 gcc 编译器。

1.3.1 简单对象

本节从最简单的 C++对象开始。在一个 C++对象中包含成员数据和成员函数，成员数据可以分为静态成员数据和非静态成员数据；成员函数可以分为静态成员函数、非静态成员函数和虚函数。下面是一个简单的 C++类，以此为例说明 C++如何存储和实现这个类的对象：

```
class simpleClass
{
      public:
        static int nCount;
        int nValue;
        char c;

      simpleClass();
      virtual ~simpleClass();

      int getValue(void);
      virtual void foo(void);
      static void addCount();
};
```

首先来看 simpleClass 的对象的大小，通过 sizeof()得到 simpleClass 的对象是 12 个字节。

由 1.1 节可以知道，静态数据成员 static int nCount 存储在全局/静态存储区中，并不作为对象占据的内存的一部分，sizeof()返回的大小不包括 nCount 所占据的内存的大小。而非静态数据成员 int nValue 和 char c 存储在对象占据的内存中，不论是在全局/静态存储区，还是在堆上或者栈上。nValue 是整型，大小为 4 个字节(32 位计算机)。c 是字符型，大小应该是 1 个字节。但在 32 位计算机上，为了提高效率会按 4 个字节对齐，因此也占据了 4 个字节。

这样，simpleClass 中的数据成员共占用了 8 个字节。而 simpleClass 中还有 1 个静态成员函数，两个虚函数（包括 1 个析构函数）和两个非静态成员函数（包括 1 个构造函数）。事实上，C++编译器还会为 simpleClass 自动添加一些成员函数，如拷贝构造函数等，显然这些函数不会只占用 4 个字节。

如果修改 simpleClass，即去掉两个虚函数，此时通过 sizeof()得到的 simpleClass 对象的大小就变成了 8 个字节，可见剩下的 4 个字节是和虚函数有关的。虚函数是 C++中的一个重要特性，用来实现面向对象中的多态性。即只有在程序运行时，才能决定一个父类对象的指针调用的函数是父类还是子类中的实现。为了实现这一特性，C++编译器在碰到含有虚函数的类时，会分配一个指针指向一个函数地址表，叫做“虚函数表”，这 4 个字节就是虚函数表指针所占据的 4 个字节。下面通过一个简单的例子来说明虚函数表指针在 C++对象的内存中的位置：

```
// simpleClass 的定义同上
int main()
{
simpleClass aSimple;

printf("Object start address:    %x\n", &aSimple);
printf("nValue address:          %x\n", &aSimple.nValue);
printf("c address:               %x\n", &aSimple.c);
printf("size: %d\n", sizeof(simpleClass));

return 0;
}
```

在上面的例子中，输出了 simpleClass 的一个实例 aSimple 在内存中的地址，以及两个成员数据 nValue 和 c 的地址。下面是在 Windows 下用 Visual C++编译运行的结果：

```
Object start address:    12fed8
nValue address:          12fedc
c address:               12fee0
size: 12
```

不难看出，虚函数表指针占据的应该是一个对象开始的 4 个字节，这一点在后面介绍继承和虚函数调用的具体实现时会进一步说明。

对于静态成员函数和非静态成员函数，C++编译器采用与普通 C 函数类似的方式进行编译，只不过对函数名进行了名称修饰(name mangling)，用来支持重载。并且在参数列表中增加了一个 this 指针，用来表明是哪一个对象调用的该函数。因此，静态成员函数和非静态成员函数的多少对对象的大小没有影响。

由上面的分析，可以有以下结论。

（1）非静态数据成员是影响对象占据内存大小的主要因素，随着对象数目的增加，非静态数据成员占据的内存会相应增加。

（2）所有的对象共享一份静态数据成员，所以静态数据成员占据的内存的数量不会随着对象数目的增加而增加。

（3）静态成员函数和非静态成员函数不会影响对象内存的大小，虽然其实现会占据相应的内存空间，同样也不会随着对象数目的增加而增加。

（4）如果对象中包含虚函数，会增加 4 个字节的空间，不论有多少个虚函数。

图 1-3 所示为一个简单 C++对象的内存布局。

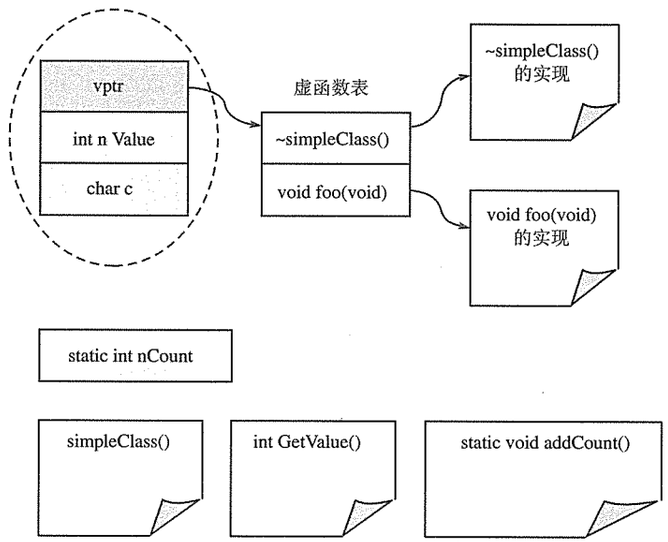


图 1-3 一个简单 C++对象的内存布局

在图中，一个 C++对象只有虚线部分中的内容才是随着对象数目的增加而需要增加内存的部分。即使部分的内容是随着程序的运行动态变化的，也是进行内存优化时需要特别关注的部分。其他部分一般是静态的，在程序编译阶段就决定了它们的大小，当然这个部分内容决定了在运行时会有多少数目的对象。当分析一个程序运行状态下内存的动态情况而改进时，需要修改的更多的应该是虚线外的部分。

在虚函数表 vtable 中不必完全是指向虚函数实现的指针。当指定编译器打开 RTTI 开关

时，vtable 中的第 1 个指针指向的是一个 typeinfo 的结构，每个类只产生一个 typeinfo 结构的实例。当程序调用 typeid()来获取类的信息时，实际上就是通过 vtable 中的第 1 个指针获得了 typeinfo。

1.3.2 单继承

毫无疑问，继承是 C++语言中非常重要的概念，也是所有面向对象语言中的重要概念。通过继承，在设计时可以清晰地表示对象之间的关系，在开发时可以方便地进行重用。任何一个实际的 C++应用程序中，都避免不了使用继承。本节将按照继承的种类介绍派生对象的内存布局。

众所周知，在 C++中继承可以分为单继承和多继承。首先来看最简单的单继承的情况，还是以 1.3.1 节中定义的类为例。在其基础上派生一个子类 derivedClass，并增加一个成员数据，如图 1-4 所示。

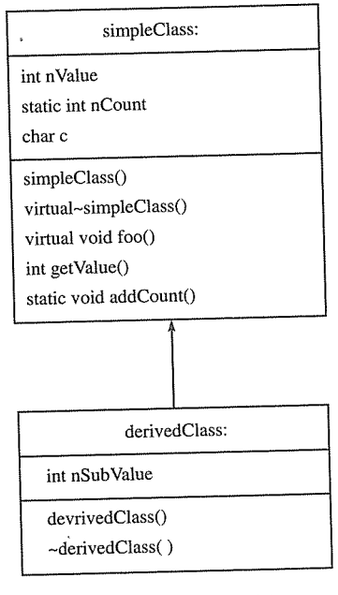


图 1-4 单继承示例

本节仍采取上节中的方法，从类的实例的大小入手，通过其中成员变量在内存中的位

置来验证派生类对象的内存布局。

由前面的分析可以知道，在 C++对象模型类实例的内存布局中只包含成员数据。而在成员函数中，只用虚函数会对内存布局有影响（虚函数表）。派生类在实例化时，需要首先构造一个父类的实例。为了验证它们在内存布局中的关系，用下面的程序输出图 1-4 中两个类的实例的大小及成员数据的内存地址：

```
#include <stdio.h>

class simpleClass
{
    public:
        static int nCount;
        int nValue;
        char c;

        simpleClass() { printf("create base class.\n"); };
        virtual ~simpleClass() {printf("destroy base class.\n");};

        int getValue(void);
        virtual void foo(void) {};
        static void addCount();
};

class derivedClass : public simpleClass
{
    public:
        int nSubValue;
        derivedClass() {printf("create derived class.\n\n");};
        ~derivedClass() {printf("destroy derived class.\n");};
        virtual void foo(void) {printf("foo in derivedClass\n");};
};

int simpleClass::nCount = 0;

int main()
{
    derivedClass aSimple;
```

```
        printf("Object start address:     %x\n", &aSimple);
        printf("nValue address:           %x\n", &aSimple.nValue);
        printf("c address:                %x\n", &aSimple.c);
        printf("nSubValue address:        %x\n", &aSimple.nSubValue);
       printf("baseClass size: %d, derivedClass size: %d\n\n", sizeof (simpleClass),
sizeof(derivedClass));

        return 0;
    }
```

在基类和派生类的构造和析构函数中分别添加了输出，用来验证其构造和析构的顺序。下面是程序的输出：

```
create base class.
create derived class.

Object start address:       13fed4
nValue address:             13fed8
c address:                  13fedc
nSubValue address:          13fee0
simpleClass size: 12, derivedClass size: 16

destroy derived class.
destroy base class.
```

根据基类和派生类的构造与析构函数的输出，可以知道，在构造一个派生类的实例时要首先构造一个基类的实例，而这个基类的实例在派生类的实例销毁之后被销毁。当构造基类的实例时，其内存布局与 1.3.1 节中的简单对象的内存布局是一样的。

其次可以看到，derivedClass 的实例的大小是 16 个字节，基类 simpleClass 的大小是 12 个字节。而 derivedClass 中增加了一个整型成员变量 nSubValue，需要占据 4 个字节。显然 derivedClass 也需要一个虚函数表，因此可以得出，派生类 derivedClass 与其基类 simpleClass 使用的是同一个虚函数表。或者说，派生类在构造时，不会再创建一个新的虚函数表，而应该在基类的虚函数表中增加或修改（关于虚函数的创建和修改，在 1.4 节中会详细介绍）。

derivedClass 实例 aSimple 的地址是 0x13fed4，其第 1 个数据成员的地址是 0x13fed8。前面已经介绍过，在简单对象模型中，虚函数表是在一个类实例的开始部分。因此可以得出，在派生类的实例中，虚函数表的位置没有发生变化。

由以上分析，可以得出单继承对象的内存布局示意，如图 1-5 所示。由于静态成员数据、静态成员函数和非静态成员函数的内存布局没有变化，所示在图中没有画出。

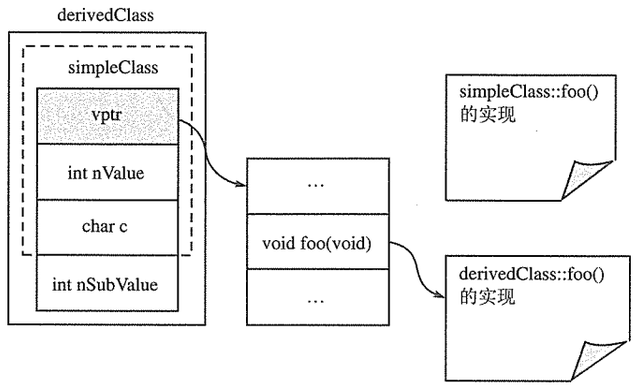


图 1-5 单继承对象的内存布局示意

该图表示了单继承情况下对象内存布局的情况，在派生类实例的头部存在一个基类的实例，派生类的实例使用的是在创建基类实例时建立的虚函数表。但需要注意的是，虚函数表的内容在建立派生类的实例时发生了变化。虚函数表中 virtual void foo(void)的位置存储的是指向 derivedClass::foo()的实现，而并非其基类 simpleClass 中 foo()的实现。

1.3.3 多继承

C++中另外一种继承是多继承，单继承中的派生类有且只有一个基类。而在多继承中，派生类可以有一个以上的基类。多继承是 C++语言中颇受争议的一个语法特性，它就像一把双刃剑，在提供了许多便利及强大的功能的同时，也带来了一些让人容易产生错误的不便。许多后来的面向对象程序设计语言中取消了多继承，而是提供了更清晰的接口的概念。但在 C++语言中，仍然是通过多继承来实现接口。在许多面向接口编程的模型，如 COM 中，都是用多继承来实现的。例如，假设现在需要开发一个文本处理软件，要求有些文本既可以打印，也可以存储，而有些文本只能打印或存储。考虑到可扩展性，一个比较好的设计是将打印和存储分别定义为两个接口，在其中定义相应的方法。当一个类实现了这两个接口时，其对象既可以存储，也可以打印。如果只实现了其中的一个接口，则只具备相应的功能。当将来系统需要扩展其他功能时，只要定义新的接口即可。图 1-6 所示为这个例子。

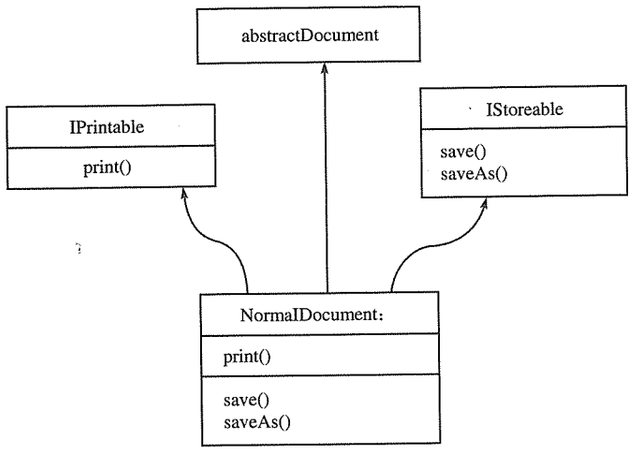


图 1-6　多继承示例

因为 C++中没有接口这样一个语法特性，因此在实现时，normalDocument 从 3 个基类中派生出来并实现对应的方法。

为了讨论多重继承的内存布局，如下仍通过一个简单的程序及相应的输出来研究：

```
#include <stdio.h>

class simpleClass1
{
public:
    int nValue1;
    char c;

    simpleClass1() { printf("create simpleClass1.\n"); };
    virtual ~simpleClass1() {printf("destroy simpleClass1.\n");};

    int getValue(void) {};
    virtual void foo1(void) {};
};

class simpleClass2
{
public:
    int nValue2;
```



```
    simpleClass2() { printf("create simpleClass2.\n"); };
    virtual ~simpleClass2() {printf("destroy simpleClass2.\n");};

    int getValue(void) {};
    virtual void foo2(void) {};
};

class derivedClass : public simpleClass1, public simpleClass2
{
    public:
        int nSubValue;
        derivedClass() {printf("create derived class.\n\n");};
        ~derivedClass() {printf("destroy derived class.\n");};
        virtual void foo2(void) {};
};

int main()
{
    derivedClass aSimple;

    printf("Object start address:        %x\n", &aSimple);
    printf("nValue1 address:             %x\n", &aSimple.nValue1);
    printf("c address:                   %x\n", &aSimple.c);
    printf("nValue2 address:             %x\n", &aSimple.nValue2);
    printf("nSubValue address:           %x\n", &aSimple.nSubValue);
    printf("simpleClass1 size: %d, simpleClass2 size: %d, derivedClass size:
%d\n\n", sizeof(simpleClass1), sizeof(simpleClass2), sizeof(derivedClass));

    return 0;
}
```

这里继续使用 simpleClass 的例子，将其改名为“simpleClass1”。并新添加了一个类 simpleClass2，derivedClass 由这两个类派生出来。simpleClass1 中有两个数据成员，一个整型 nValue1，一个字符型 c。而 simpleClass2 和 derivedClass 中都只有一个整型数据成员 nValue2 和 nSubValue，simpleClass1 和 simpleClass2 中都含有虚函数。

通过构造函数和析构函数的输出，可以验证对象创建和销毁的顺序。在主程序中，输出对象及其数据成员的内存地址来验证对象的内存布局。下面是程序的输出：

```
create simpleClass1.
create simpleClass2.
create derived class.

Object start address:          12fecc
nValue1 address:               12fed0
c address:                     12fed4
nValue2 address:               12fedc
nSubValue address:             12fee0
simpleClass1 size: 12, simpleClass2 size: 8, derivedClass size: 24

destroy derived class.
destroy simpleClass2.
destroy simpleClass1.
```

可以得出以下结论。

（1）与单继承相同，创建派生类的对象时，要首先创建基类的对象。由于多继承中一个派生类有多个基类，因此创建基类的对象时要遵循一定的顺序，这个顺序是由派生类声明时决定的。如果将 derivedClass 的声明改为：

```
class derivedClass : public simpleClass2, public simpleClass1
```

在输出中就会看到 simpleClass2 会首先被创建，基类对象销毁的顺序与创建的顺序相反。

（2）simpleClass1 的一个对象的大小是 12 个字节，simpleClass2 的一个对象的大小是 8 个字节（因为含有虚函数，因此各包含了一个 4 个字节的虚函数表指针）。而派生类 derivedClass 增加了一个 4 个字节的整型成员数据，大小是 24 个字节。可见同单继承一样，虚函数表的指针还是在基类的开始部分。如果将 simpleClass2 中的所有虚函数改为非虚函数，则 simpleClass2 的对象将不需要虚函数表指针。大小将变为 4 个字节，相应的派生类 derivedClass 的大小将变为 20 个字节。

（3）在多继承中还需要注意的就是避免二义性。在上面的例子中，simpleClass1 和 simpleClass2 都定义了 getValue()函数。如果在 main()函数中调用 aSimple.getValue()，编译器将无法知道要调用的是哪一个基类中的 getValue()，因此会报二义性错误。同样，成员数据也是如此。如果将 simpleClass1 中的 nValue1 和 simpleClass2 中的 nValue2 改成相同名称“nValue”，则在访问 aSimple.nValue 同样会出现二义性错误。

图 1-7 所示为多继承的最简单内存布局。每一个派生类的实例中包含所有基类的实例，每个基类的实例有自己的虚函数表。在上面的例子中，派生类 derivedClass 没有重新实现 simpleClass1 的虚函数 foo1()。因此在基类 simpleClass1 的虚函数表中，foo1()的指针仍然是指向 simpleClass1 的 foo1()实现。而 derivedClass 中重新实现了虚函数 foo2()，因此 simpleClass2 的虚函数表中，foo2()的指针被重新指向了 derivedClass 实现的 foo2()。

多继承会引入许多复杂的情况，“菱形继承”是非常典型的一种。图 1-8 说明了什么是菱形继承。

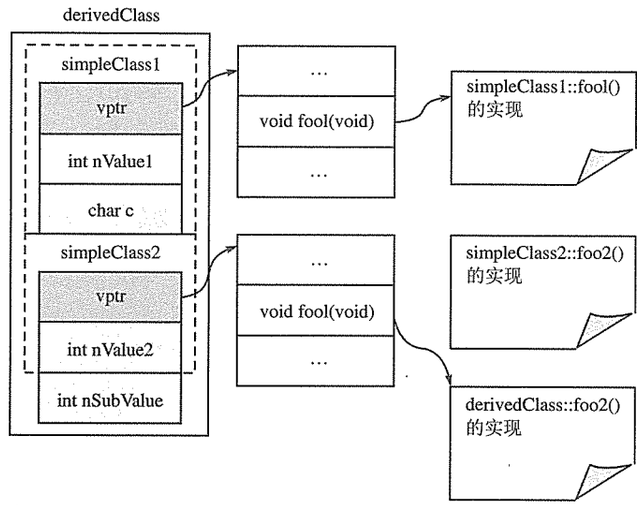


图 1-7 多继承的最简单内存布局

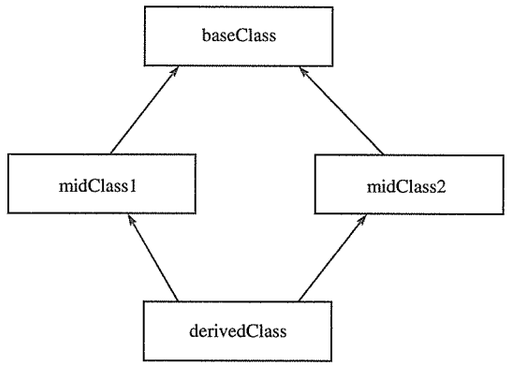


图 1-8 菱形继承

derivedClass 由 midClass1 和 midClass2 派生而来，而 midClass1 和 midClass2 都是由 baseClass 派生出来，这样 4 者之间的继承关系形成一个菱形，因此称之为“菱形继承”。现在还是通过如下示例程序来研究菱形继承会带来哪些有趣的情况：

```
#include <stdio.h>

class baseClass
{
    public:
        int nValue1;
        char c;

        baseClass() { nValue1=0; printf("create baseClass.\n"); };
        virtual ~baseClass() {printf("destroy baseClass.\n");};

        virtual void foo1(void) {};
        void getValue(void) {printf("nValue1 = %d\n", nValue1);};
};

class midClass1 : public baseClass
{
    public:
            int nMidValue1;

            midClass1() { printf("create midClass1.\n"); };
            virtual ~midClass1() {printf("destroy midClass1.\n");};

            int getValue(void) {};
            virtual void setValue(void) {nValue1 = 10;};
};

class midClass2  : public baseClass
{
    public:
            int nMidValue2;

            midClass2() { printf("create midClass2.\n"); };
            virtual ~midClass2() {printf("destroy midClass2.\n");};
```



```
            int getValue(void) {};
            virtual void setValue(void) {nValue1 = 20;};
};

class derivedClass : public midClass1, public midClass2
{
     public:
            int nSubValue;
            derivedClass() {printf("create derived class.\n\n");};
            ~derivedClass() {printf("destroy derived class.\n");};
            virtual void foo2(void) {};
};

int main()
{
    derivedClass aSimple;

    printf("Object start address:    %x\n", &aSimple);
    printf("nValue1 address:         %x\n", &aSimple.midClass1::nValue1);
    printf("c address:               %x\n", &aSimple.midClass1::c);
    printf("midValue1 address:       %x\n", &aSimple.nMidValue1);
    printf("nValue1 address:         %x\n", &aSimple.midClass2::nValue1);
    printf("c address:               %x\n", &aSimple.midClass2::c);
    printf("midValue2 address:       %x\n", &aSimple.nMidValue2);
    printf("nSubValue address:       %x\n", &aSimple.nSubValue);
    printf("baseClass size: %d, midClass1 size: %d, midClass2 size: %d,
derivedClass size: %d\n\n",  sizeof(baseClass), sizeof(midClass1),
sizeof(midClass2), sizeof(derivedClass));

    aSimple.midClass2::setValue();
    aSimple.baseClass::getValue();

    aSimple.midClass1::setValue();
    aSimple.baseClass::getValue();
    return 0;
}
```

这个程序实现了图 1-8 中给出的菱形继承的关系，用构造和析构函数的输出来跟踪对象的生命周期并输出各个成员数据的内存地址。下面是该程序的输出：

```
create baseClass.
create midClass1.
create baseClass.
create midClass2.
create derived class.

Object start address:        13fec0
nValue1 address:             13fec4
c address:                   13fec8
midValue1 address:           13fecc
nValue1 address:             13fed4
c address:                   13fed8
midValue2 address:           13fedc
nSubValue address:           13fee0
baseClass size: 12, midClass1 size: 16, midClass2 size: 16, derivedClass size:
36

nValue1 = 0
nValue1 = 10
destroy derived class.
destroy midClass2.
destroy baseClass.
destroy midClass1.
destroy baseClass.
```

首先会看到创建了两次 baseClass，这是因为在构造 derivedClass 时，要首先构造其基类 midClass1 和 midClass2。而构造这两个类时，又要分别构造其基类 baseClass，因此就创建了两个 baseClass 的实例。图 1-9 简单地说明了这种情况。

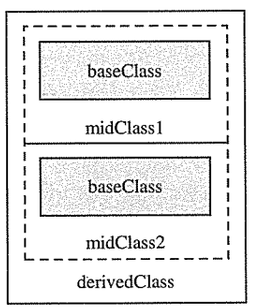


图 1-9　两个 baseClass 的实例

在 derivedClass 的实例中，存在两个 baseClass 的实例，每一个都有自己的内存空间。这不仅浪费了内存空间，也带来了很多二义性的问题。例如在输出祖父基类 baseClass 的成员数据时，代码如下：

```
printf("nValue1 address:     %x\n", &aSimple.midClass1::nValue1);
printf("nValue1 address:     %x\n", &aSimple.midClass2::nValue1);
printf("c address:           %x\n", &aSimple.midClass1::c);
printf("c address:           %x\n", &aSimple.midClass2::c);
```

如果采用 aSimple.nValue1 或者 aSimple.c，会在编译时引发二义性错误。而其输出的内存地址也表明在 derivedClass 的实例中存在两份 nValue1 和 c 的拷贝，分别位于不同的 baseClass 的实例中。同样在调用 setValue()和 getValue()函数时，也必须明确指明调用的是哪一个基类的方法；否则也会引发二义性编译错误。

```
aSimple.midClass2::setValue();
aSimple.baseClass::getValue();

aSimple.midClass1::setValue();
aSimple.baseClass::getValue();
```

但在调用 baseClass::getValue()还是会存在问题。程序中首先调用 midClass2::setValue()设置 nValue1 为 20，但通过 baseClass::getValue()输出 nValue1 时，仍然是初始值 0。

```
nValue1 = 4214651
nValue1 = 10
```

这是因为编译器默认 baseClass::getValue()调用的是第 1 个基类 midClass1 中的 baseClass 实例的方法，即 midClass1::baseClass::getValue()，因此输出并不是通过 midClass2::setValue()设置的值。由此可见，菱形继承会带来很多不易察觉的隐藏的错误。

造成这些问题的原因就是存在两个 baseClass 的实例，为了避免这种情况，C++语言提供了虚拟继承（virtual）。当使用虚拟继承时，公共的基类只存在一个实例。现在修改上例中的 midClass1 和 midClass2 的定义，使其由虚拟继承 baseClass：

```
class midClass1 : virtual public baseClass
......
class midClass2 : virtual public baseClass
......
```

程序的其他部分保持不变，在 Windows 上通过 Visual C++编译输出的结果如下：

```
create baseClass.
create midClass1.
create midClass2.
create derived class.

Object start address:        13febc
nValue1 address:             13fedc
c address:                   13fee0
midValue1 address:           13fec4
nValue1 address:             13fedc
c address:                   13fee0
midValue2 address:           13fed0
nSubValue address:           13fed4
baseClass size: 12, midClass1 size: 24, midClass2 size: 24, derivedClass size:
40

nValue1 = 20
nValue1 = 20
nValue1 = 10
destroy derived class.
destroy midClass2.
destroy midClass1.
destroy baseClass.
```

可以得出如下结论。

（1）baseClass 只创建了一个实例，其数据成员的地址相同。

（2）baseClass 的实例放在 derivedClass 实例的内存空间中的最后部分。

（3）没有了二义性问题，setValue()和 getValue()的数据是一致的。

（4）不使用虚拟继承时，midClass1 和 midClass2 的大小是 16 个字节，derivedClass 的大小是 36 个字节。而使用了虚拟继承后，midClass1 和 midClass2 的大小变为 24 个字节，而 derivedClass 的大小变为 40 个字节。为了支持虚拟继承，不同的编译器的做法会有所不同。在 Visual C++中通过添加一个虚基类表（virtual base table）的指针来实现（类似于虚函数表指针），因此造成了空间变大。对具有虚拟继承基类的对象进行类型转换时，也会有比较大的开销，这些都是在使用虚拟继承时需要考虑的地方。

图 1-10 所示为上述虚拟继承的例子中 derivedClass 对象的内存布局。

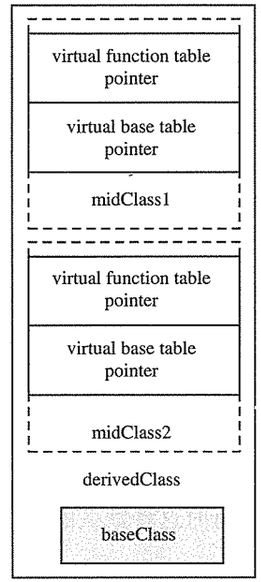


图 1-10 derivedClass 对象的内存布局

1.4 构造与析构

前面讨论了 C++对象的内存布局，本节将讨论内存布局是如何形成的。即对象是如何创建的，此外还将讨论对象的销毁过程。

在 1.2 节“对象的生命周期”中介绍了 C++对象的 3 种创建方式，不论哪种方式，创建一个对象时，都需要获得所需的内存空间，并且调用类的构造函数。

根据 1.1 节可以知道，内存空间要根据对象是哪种类型的变量来决定，如是栈上的空间，或者是通过 new 获得的堆上的空间等。

对于构造函数，C++标准中规定每个类都必须有构造函数。如果开发人员没有定义，则编译器会提供一个默认的构造函数。这个默认的构造函数不带任何参数，也不会对成员数据进行初始化。如果类中定义了任何一种形式的构造函数，则不会产生默认的构造函数。例如定义下面的类：

```
class simpleClass
{
    public:
        int nValue;
        simpleClass(int n) {nValue = n;};
};

simpleClass aSimple;      // 编译错误，没有合适的构造函数
```

其中定义了 simpleClass 的一个带参数构造函数，因此没有默认的构造函数。但在声明 aSimple 时，没有给出参数，就会出现编译错误。可以修改为 simpleClass aSimple(10)来实例化一个 simpleClass 的对象。

除了默认的构造函数，每个类中还必须定义拷贝构造函数。同样，如果没有定义成员，编译器会自动产生一个默认的拷贝构造函数。这个拷贝构造函数执行的是位拷贝，即按找对象的内存空间逐个字节进行复制。有时这种默认的拷贝构造函数会带来隐含的内存问题。下面是一个典型的例子：

```
#include <stdio.h>

class simpleClass
{
    private:
    //  simpleClass(simpleClass &);
        char *buffer;
    public:
        int nValue;
        simpleClass(int n) {
            nValue = n;
            buffer = new char[n];
        };

        ~simpleClass() {
            if (buffer)  {
                printf("buffer %x deleted.\n", buffer);
                delete buffer;
            }
        };
};
```



```
int main()
{
    simpleClass a(10);
    simpleClass b = a;
    printf("%d\n", a.nValue);

    return 0;
}
```

在这个程序中，simpleClass 在构造时分配了 n 个字节的缓冲区，在析构时会释放这个缓冲区。但是由于没有定义拷贝构造函数，因此 simpleClass b=a 会通过位拷贝构造一个 simpleClass 对象 b，其中的 buffer 指向和 a 对象 buffer 相同的内存区。在 a 和 b 对象析构时，这块内存区被释放两次，因此会造成程序崩溃，这是非常典型的默认拷贝构造函数带来的内存错误。如果不想通过赋值或者拷贝构造对象，可以将拷贝构造函数定义在 private 区域，这样 simpleClass b=a 就会在编译时出错。

1.5 本章小结

本章介绍了程序数据在内存中分布的基本概念，然后重点介绍了 C++语言的对象模型、内存布局，以及对象的生命周期。这些基本概念是本书后面几章，包括 C++性能相关的语言特性、动态内存管理，以及内存池的设计与实现等的基础。后面几章将在这个基础上介绍一些 C++内存管理和性能优化的一些高级技巧。

第 2 章 C++语言特性的性能分析

大多数开发人员通常都有这个观点，即汇编语言和C语言适合用来编写对性能要求非常高的程序。而 C++语言的主要应用范围是编写复杂度非常高，但是对性能要求不是那么严格的程序。因为在大多数人看来，C++语言相对前面两种语言来说，设计时因为考虑到要支持多种编程模式，比如面向对象编程和范型编程，另外还有其他诸如例外处理等，从而引入了太多新的语言特性。而这些特性往往使得编译器在编译程序时，插入了很多额外的代码。这样不仅仅使得最终生成的二进制代码体积膨胀，而且使得执行速度下降。举个例子说，比如开发人员只写了“Object obj;”这样一个简单的语句。但在实际编译时，如果 Object 是一个十分复杂的类，可能会引起很多操作，尤其是当构造还需要从堆中动态分配内存来放置 Object 中的成员变量对象时更是如此。

但是事实往往并非如此，很多时候，一个程序的速度在框架设计完成时大致已经确定了，而并非是因为采用了 C++语言才使其速度没有达到预期的目标。因此当一个程序的性能需要提高时，首先需要做的是用性能检测工具对其运行的时间分布进行一个准确的测量，找出关键路径和真正的瓶颈所在，然后针对瓶颈进行分析和优化，而不是一味盲目地将性能低劣归咎于所采用的语言。事实上，如果框架设计不做修改，即使用 C 语言或者汇编语言重新改写，也并不能保证提高总体性能。

因此当遇到性能问题时，首先检查和反思程序的总体框架。然后用性能检测工具对其实际运行做准确地测量，再针对瓶颈进行分析和优化，这才是正确的思路。

但不可否认的是，确实有一些操作或者 C++的一些语言特性比其他因素更容易成为程序的瓶颈，一般公认的有如下因素。

（1）缺页：如第四章中所述，缺页往往意味着需要访问外部存储。因为外部存储访问相对于访问内存或者代码执行，有数量级的差别。因此只要有可能，应该尽量想办法减少缺页。

（2）从堆中动态申请和释放内存：如 C 语言中的 malloc/free 和 C++语言中的 new/delete 操作非常耗时，因此要尽可能优先考虑从线程栈中获得内存。优先考虑栈而减少从动态堆中申请内存，不仅仅是因为在堆中开辟内存比在栈中要慢很多，而且还与“尽量减少缺页”这一宗旨有关。当执行程序时，当前栈帧空间所在的内存页肯定在物理内存中，因此程序代码对其中变量的存取不会引起缺页；相反，从堆中生成的对象，只有指向它的指针在栈上，对象本身却是在堆中。堆一般来说不可能都在物理内存中，而且因为堆分配内存的特性，即使两个相邻生成的对象，也很有可能在堆内存位置上相隔很远。因此当访问这两个对象时，虽然分别指向它们指针都在栈上，但是通过这两个指针引用它们时，很有可能会引起两次“缺页”。

（3）复杂对象的创建和销毁：这往往是一个层次相当深的递归调用，因为一个对象的创建往往只需要一条语句，看似很简单。另外，编译器生成的临时对象因为在程序的源代码中看不到，更是不容易察觉，因此尤其值得警惕和关注。本章中专门有两节分别讲解对象的构造和析构，以及临时对象。

（4）函数调用：因为函数调用有固定的额外开销，因此当函数体的代码量相对较少，且该函数被非常频繁地调用时，函数调用时的固定额外开销容易成为不必要的开销。C 语言的宏和 C++语言的内联函数都是为了在保持函数调用的模块化特征基础上消除函数调用的固定额外开销而引入的，因为宏在提供性能优势的同时也给开发和调试带来了不便。在 C++中更多提倡的是使用内联函数，本章会有一节专门讲解内联函数。

2.1 构造函数与析构函数

构造函数和析构函数的特点是当创建对象时，自动执行构造函数；当销毁对象时，析构函数自动被执行。这两个函数分别是一个对象最先和最后被执行的函数，构造函数在创建对象时调用，用来初始化该对象的初始状态和取得该对象被使用前需要的一些资源，比如文件/网络连接等；析构函数执行与构造函数相反的操作，主要是释放对象拥有的资源，而且在此对象的生命周期这两个函数都只被执行一次。

创建一个对象一般有两种方式，一种是从线程运行栈中创建，也称为“局部对象”，一般语句为：

```
{
        ……
        Object obj;                ①
        ……
}                                  ②
```

销毁这种对象并不需要程序显式地调用析构函数，而是当程序运行出该对象所属的作用域时自动调用。比如上述程序中在①处创建的对象 obj 在②处会自动调用该对象的析构函数。在这种方式中，对象 obj 的内存在程序进入该作用域时，编译器生成的代码已经为其分配（一般都是通过移动栈指针），①句只需要调用对象的构造函数即可。②处编译器生成的代码会调用该作用域内所有局部的用户自定义类型对象的析构函数，对象 obj 属于其中之一，然后通过一个退栈语句一次性将空间返回给线程栈。

另一种创建对象的方式为从全局堆中动态创建，一般语句为：

```
{
        ……
        Object* obj = new Object;      ①
        ……
        delete obj;                    ②
        ……
}                                      ③
```

当执行①句时，指针 obj 所指向对象的内存从全局堆中取得，并将地址值赋给 obj。但指针 obj 本身却是一个局部对象，需要从线程栈中分配，它所指向的对象从全局堆中分配内存存放。从全局堆中创建的对象需要显式调用 delete 销毁，delete 会调用该指针指向的对象的析构函数，并将该对象所占的全局堆内存空间返回给全局堆，如②句。执行②句后，指针 obj 所指向的对象确实已被销毁。但是指针 obj 却还存在于栈中，直到程序退出其所在的作用域。即执行到③处时，指针 obj 才会消失。需要注意的是，指针 obj 的值在②处至③处之间，仍然指向刚才被销毁的对象的位置，这时使用这个指针是危险的。在 Win32 平台中，访问刚才被销毁对象，可能出现 3 种情况。第 1 种情况是该处位置所在的“内存页”没有任何对象，堆管理器已经将其进一步返回给系统，此时通过指针 obj 访问该处内存会引起“访问违例”，即访问了不合法的内存，这种错误会导致进程崩溃；第 2 种情况是该处位置所在的“内存页”还有其他对象，且该处位置被回收后，尚未被分配出去，这时通过指针 obj 访问该处内存，取得的值是无意义的，虽然不会立刻引起进程崩溃，但是针对该指针的后续操作的行为是不可预测的；第 3 种情况是该处位置所在的“内存页”还有其他对象，且该处位置被回收后，已被其他对象申请，这时通过指针 obj 访问该处内存，取得的值其实是程序其他处生成的对象。虽然对指针 obj 的操作不会立刻引起进程崩溃，但是极有可能会引起该对象状态的改变。从而使得在创建该对象处看来，该对象的状态会莫名其妙地变化。第 2 种和第 3 种情况都是很难发现和排查的 bug，需要小心地避免。

创建一个对象分成两个步骤，即首先取得对象所需的内存（无论是从线程栈还是从全局堆中），然后在该块内存上执行构造函数。在构造函数构建该对象时，构造函数也分成两个步骤。即第 1 步执行初始化（通过初始化列表），第 2 步执行构造函数的函数体，如下：

```
class  Derived  :  public Base
{
public :
        Derived() : i(10),  string("unnamed")          ①
        {
            ...                                        ②
        }
        ...

private :
        int  i;
        string  name;
        ...
};
```

①步中的 “: i(10)， string("unnamed")” 即所谓的“初始化列表”，以“:”开始，后面为初始化单元。每个单元都是“变量名（初始值）”这样的模式，各单元之间以逗号隔开。构造函数首先根据初始化列表执行初始化，然后执行构造函数的函数体，即②处语句。对初始化操作，有下面几点需要注意。

（1）构造函数其实是一个递归操作，在每层递归内部的操作遵循严格的次序。递归模式为首先执行父类的构造函数（父类的构造函数操作也相应的包括执行初始化和执行构造函数体两个部分），父类构造函数返回后构造该类自己的成员变量。构造该类自己的成员变量时，一是严格按照成员变量在类中的声明顺序进行，而与其在初始化列表中出现的顺序完全无关；二是当有些成员变量或父类对象没有在初始化列表中出现时，它们仍然在初始化操作这一步骤中被初始化。内建类型成员变量被赋给一个初值。父类对象和类成员变量对象被调用其默认构造函数初始化，然后父类的构造函数和子成员变量对象在构造函数执行过程中也遵循上述递归操作。一直到此类的继承体系中所有父类和父类所含的成员变量都被构造完成后，此类的初始化操作才告结束。

（2）父类对象和一些成员变量没有出现在初始化列表中时，这些对象仍然被执行构造函数，这时执行的是“默认构造函数”。因此这些对象所属的类必须提供可以调用的默认构造函数，为此要求这些类要么自己“显式”地提供默认构造函数，要么不能阻止编译器“隐式”地为其生成一个默认构造函数，定义除默认构造函数之外的其他类型的构造函数就会阻止编译器生成默认构造函数。如果编译器在编译时，发现没有可供调用的默认构造函数，并且编译器也无法生成，则编译无法通过。

（3）对两类成员变量，需要强调指出即“常量”（const）型和“引用”（reference）型。因为已经指出，所有成员变量在执行函数体之前已经被构造，即已经拥有初始值。根据这个特点，很容易推断出“常量”型和“引用”型变量必须在初始化列表中正确初始化，而不能将其初始化放在构造函数体内。因为这两类变量一旦被赋值，其整个生命周期都不能修改其初始值。所以必须在第一次即“初始化”操作中被正确赋值。

（4）可以看到，即使初始化列表可能没有完全列出其子成员或父类对象成员，或者顺序与其在类中声明的顺序不符，这些成员仍然保证会被“全部”且“严格地按照顺序”被

构建。这意味着在程序进入构造函数体之前，类的父类对象和所有子成员变量对象都已经被生成和构造。如果在构造函数体内为其执行赋初值操作，显然属于浪费。如果在构造函数时已经知道如何为类的子成员变量初始化，那么应该将这些初始化信息通过构造函数的初始化列表赋予子成员变量，而不是在构造函数体中进行这些初始化。因为进入构造函数体时，这些子成员变量已经初始化一次。

下面这个例子演示了构造函数的这些重要特性：

```
#include <iostream>
using namespace std;

class A
{
    public:
        A() {    cout << "A::A()" << endl;     }
};

class B : public A
{
    public:
        B() : j(0)   {    cout << "B::B()" << endl;     }

private:
        int j;
};

class C1
{
public:
        C1(int i)  :  a(i)  {     cout << "C1::C1()" << endl;  }

private:
        int a;
};

class C2
{
public:
        C2(double val)  :  d(val)  { cout << "C2::C2()" << endl;     }
```



```
private:
        double d;
};

class C3
{
public:
        C3(int v = 0)  :  j(v)  {  cout << "C3::C3()" << endl;  }

private:
        int j;
};

class D : public B
{
public:
        D(double v2, int v1):c2(v2), c1(v1){cout<< "D::D()"<<endl;}    ②

private:
        C1 c1;
        C2 c2;
        C3 c3;
};

int main()
{
        D  d(1.0,   3);                                                ①
        return 0;
}
```

在这段代码中，类 D 继承自类 B，类 B 继承自类 A。然后类 D 中含有 3 个成员变量对象 c1、c2 和 c3，分别为类型 C1、C2 和 C3。

此段程序的输出为：

```
A::A()                ③
B::B()                ④
C1::C1()              ⑤
C2::C2()              ⑥
C3::C3()              ⑦
```

```
D::D()                                                    ⑧
```

可以看到，①处调用 D::D(double，int)构造函数构造对象 d，此构造函数从②处开始引起了一连串的递归构造。从输出可以验证递归操作的如下规律。

（1）递归从父类对象开始，D 的构造函数首先通过“初始化”操作构造其直接父类 B 的构造函数。然后 B 的构造函数先执行“初始化”部分，该“初始化”操作构造 B 的直接父类 A，类 A 没有自己的成员需要初始化，所以其“初始化”不执行任何操作。初始化后，开始执行类 A 的构造函数，即③的输出。

（2）构造类 A 的对象后，B 的“初始化”操作执行初始化类表中的 j(0)对 j 进行初始化。然后进入 B 的构造函数的函数体，即④处输出的来源。至此类 B 的对象构造完毕，注意这里看到初始化列表中并没有“显式”地列出其父类的构造函数。但是子类在构造时总是在其构造函数的“初始化”操作的最开始构造其父类对象，而忽略其父类构造函数是否显式地列在初始化列表中。

（3）构造类 B 的对象后，类 D 的“初始化”操作接着初始化其成员变量对象，这里是 c1，c2 和 c3。因为它们在类 D 中的声明顺序就是 c1 -> c2 -> c3，所以看到它们也是按照这个顺序构造的，如⑤，⑥，⑦ 3 处输出所示。注意这里故意在初始化列表中将 c2 的顺序放在了 c1 的前面，c3 甚至都没有列在初始化列表中。但是输出显示了成员变量的初始化严格按照它们在类中的声明顺序进行，而忽略其是否显式地列在初始化列表中，或者显示在初始化列表中的顺序如何。应该尽量将成员变量初始化列表中出现的顺序与其在类中声明的顺序保持一致，因为如果使用一个变量的值来初始化另外一个变量时，程序的行为可能不是开发人员预想的那样，比如：

```
class Object
{
public:
        Object() :  v2(5),   v1(v2 * 3) { … }
private:
       int  v1,  v2;
}
```

这段程序的本意应该是首先将 v2 初始化为 5，然后用 v2 的值来初始化 v1，从而 v1=15。然而通过验证，初始化后的 v2 确实为 5，但 v1 则是一个非常奇怪的值（在笔者的电脑上输出是 12737697）。这是因为实际初始化时首先初始化 v1，这时 v2 还尚未正确初始化，根

据 v2 计算出来的 v1 也就不是一个合理的值了。当然除了将成员变量在初始化列表中的顺序与其在类中声明的顺序保持一致之外，最好还是避免在初始化列表中用某个成员变量的值初始化另外一个成员变量的值。

（4）随着 c1、c2 和 c3 这 3 个成员变量对象构造完毕，类 D 的构造函数的“初始化”操作部分结束，程序开始进入其构造函数的第 2 部分。即执行构造函数的函数体，这就是⑧处输出的来源。

析构函数的调用与构造函数的调用一样，也是类似的递归操作。但有两点不同，一是析构函数没有与构造函数相对应的初始化操作部分，这样析构函数的主要工作就是执行析构函数的函数体；二是析构函数执行的递归与构造函数刚好相反，而且在每一层的递归中，成员变量对象的析构顺序也与构造时刚好相反。

正是因为在执行析构函数时，没有与构造函数的初始化列表相对应的列表，所以析构函数只能选择成员变量在类中声明的顺序作为析构的顺序参考。因为构造函数选择了自然的正序，而析构函数的工作又刚好与其相反，所以析构函数选择逆序。因为析构函数只能用成员变量在类中的声明顺序作为析构顺序（要么正序，要么逆序），这样使得构造函数也只能选择将这个顺序作为构造的顺序依据，而不能采用初始化列表中的作为顺序依据。

与构造函数类似，如果操作的对象属于一个复杂继承体系中的末端节点，那么其析构函数也是十分耗时的操作。

因为构造函数/析构函数的这些特性，所以在考虑或者调整程序的性能时，也必须考虑构造函数/析构函数的成本，在那些会大量构造拥有复杂继承体系对象的大型程序中尤其如此。下面两点是构造函数/析构函数相关的性能考虑。

（5）在 C++程序中，创建/销毁对象是影响性能的一个非常突出的操作。首先，如果是从全局堆中生成对象，则需要首先进行动态内存分配操作。众所周知，动态内存分配/回收在 C/C++程序中一直都是非常费时的。因为牵涉到寻找匹配大小的内存块，找到后可能还需要截断处理，然后还需要修改维护全局堆内存使用情况信息的链表等。因为意识到频繁的内存操作会严重影响性能的下降，所以已经发展出很多技术用来缓解和降低这种影响，比如后续章节中将说明的内存池技术。其中一个主要目标就是为了减少从动态堆中申请内存的次数，从而提高程序的总体性能。当取得内存后，如果需要生成的目标对象属于一个复杂继承体系中末端的类，那么该构造函数的调用就会引起一长串的递归构造操作。在大

型复杂系统中，大量此类对象的创建很快就会成为消耗 CPU 操作的主要部分。因为注意和意识到对象的创建/销毁会降低程序的性能，所以开发人员往往对那些会创建对象的代码非常敏感。在尽量减少自己所写代码生成的对象同时，开发人员也开始留意编译器在编译时“悄悄”生成的一些临时对象。开发人员有责任尽量避免编译器为其程序生成临时对象，下面会有一节专门讨论这个问题。语义保持完全一致

（6）已经看到，如果在实现构造函数时，没有注意到执行构造函数体前的初始化操作已经将所有父类对象和成员变量对象构造完毕。而在构造函数体中进行第 2 次的赋值操作，那么也会浪费很多的宝贵 CPU 时间用来重复计算。这虽然是小疏忽，但在大型复杂系统中积少成多，也会造成程序性能的显著下降。

减少对象创建/销毁的一个很简单且常见的方法就是在函数声明中将所有的值传递改为常量引用传递，比如下面的函数声明：

```
int foo( Object  a);
```

应该相应改为：

```
int foo( const Object&  a );
```

因为 C/C++语言的函数调用都是“值传递”，因此当通过下面方式调用 foo 函数时：

```
Object a;                    ①
...
int i = foo(a);              ②
```

②处函数 foo 内部引用的变量 a 虽然名字与①中创建的 a 相同，但并不是相同的对象，两个对象“相同”的含义指其生命周期的每个时间点所指的是内存中相同的一块区域。这里①处的 a 和②处的 a 并不是相同的对象，当程序执行到②句时，编译器会生成一个局部对象。这个局部对象利用①处的 a 拷贝构造，然后执行 foo 函数。在函数体内部，通过名字 a 引用的都是通过①处 a 拷贝构造的复制品。函数体内所有对 a 的修改，实质上也只是对此复制品的修改，而不会影响到①处的原变量。当 foo 函数体执行完毕退出函数时，此复制品会被销毁，这也意味着对此复制品的修改在函数结束后都被丢失。

通过下面这段程序来验证值传递的行为特征：

```
#include <iostream>
using namespace std;
```



```
class Object
{
public:
        Object(int i = 1)        { n = i;cout << "Object::Object()" << endl; }
        Object(const Object& a)
        {
            n = a.n;
            cout << "Object::Object(const Object&)" << endl;
        }
        ~Object()                { cout << "Object::~Object()" << endl; }

        void inc()               { ++n; }
        int val() const          { return n; }

private:
        int n;
};

void foo(Object a)
{
        cout << "enter foo, before inc(): inner a = " << a.val() << endl;
        a.inc();
        cout << "enter foo, after inc(): inner a = " << a.val() << endl;
}

int main()
{
        Object a;                                                    ①

        cout << "before call foo : outer a = " << a.val() << endl;
        foo(a);                                                      ②
        cout << "after call foo : outer a = " << a.val() << endl;    ③

        return 0;
}
```

输出为：

```
Object::Object()                                    ④
before call foo : outer a = 1
Object::Object(const Object&)                       ⑤
```

```
enter foo, before inc(): inner a = 1                    ⑥
enter foo, after inc(): inner a = 2                     ⑦
Object::~Object()                                       ⑧
after call foo : outer a = 1                            ⑨
Object::~Object()
```

可以看到，④处的输出为①处对象 a 的构造，而⑤处的输出则是②处 foo(a)。调用开始时通过构造函数生成对象 a 的复制品，紧跟着在函数体内检查复制品的值。输出与外部原对象的值相同（因为是通过拷贝构造函数），然后复制品调用 inc()函数将值加 1。再次打印出⑦处的输出，复制品的值已经变成了 2。foo 函数执行后需要销毁复制品 a，即⑧处的输出。foo 函数执行后程序又回到 main 函数中继续执行，重新打印原对象 a 的值，发现其值保持不变（⑨处的输出）。

重新审视 foo 函数的设计，既然它在函数体内修改了 a。其原意应该是想修改 main 函数的对象 a，而非复制品。因为对复制品的修改在函数执行后被“丢失”，那么这时不应该传入 Object a，而是传入 Object& a。这样函数体内对 a 的修改，就是对原对象的修改。foo 函数执行后其修改仍然保持而不会丢失，这应该是设计者的初衷。

如果相反，在 foo 函数体内并没有修改 a。即只对 a 执行“读”操作，这时传入 const Object& a 是完全胜任的。而且还不会生成复制品对象，也就不会调用构造函数/析构函数。

综上所述，当函数需要修改传入参数时，如果函数声明中传入参数为对象，那么这种设计达不到预期目的。即是错误的，这时应该用应用传入参数。当函数不会修改传入参数时，如果函数声明中传入参数为对象，则这种设计能够达到程序的目的。但是因为会生成不必要的复制品对象，从而引入了不必要的构造/析构操作。这种设计是不合理和低效的，应该用常量引用传入参数。

下面这个简单的小程序用来验证在构造函数中重复赋值对性能的影响，为了放大绝对值的差距，将循环次数设置为 100 000：

```
#include <iostream>
#include <windows.h>
using namespace std;

class Val
{
```



```
public:
        Val(double v = 1.0)
        {
            for(int i = 0; i < 1000; i++)
                d[i] = v + i;
        }

        void Init(double v = 1.0)
        {
            for(int i = 0; i < 1000; i++)
                d[i] = v + i;
        }

private:
        double d[1000];
};

class Object
{
public:
        Object(double d) : v(d) {}                      ①
        /*Object(double d)                              ②
        {
             v.Init(d);
        }*/

private:
        Val v;
};

int main()
{
        unsigned long  i,    nCount;

        nCount = GetTickCount();

        for(i = 0;  i < 100000;  i++)
        {
            Object obj(5.0);
        }
```

```
        nCount = GetTickCount() - nCount;
        cout << "time used : " << nCount << "ms" << endl;

        return 0;
}
```

类 Object 中包含一个成员变量，即类 Val 的对象。类 Val 中含一个 double 数组，数组长度为 1 000。Object 在调用构造函数时就知道应为 v 赋的值，但有两种方式，一种方式是如①处那样通过初始化列表对 v 成员进行初始化；另一种方式是如②处那样在构造函数体内为 v 赋值。两种方式的性能差别到底有多大呢？测试机器（VC6 release 版本，Windows XP sp2，CPU 为 Intel 1.6 GHz 内存为 1GB）中测试结果是前者（①）耗时 406 毫秒，而后者（②）却耗时 735 毫秒，如图 2-1 所示。即如果改为前者，可以将性能提高 44.76%。

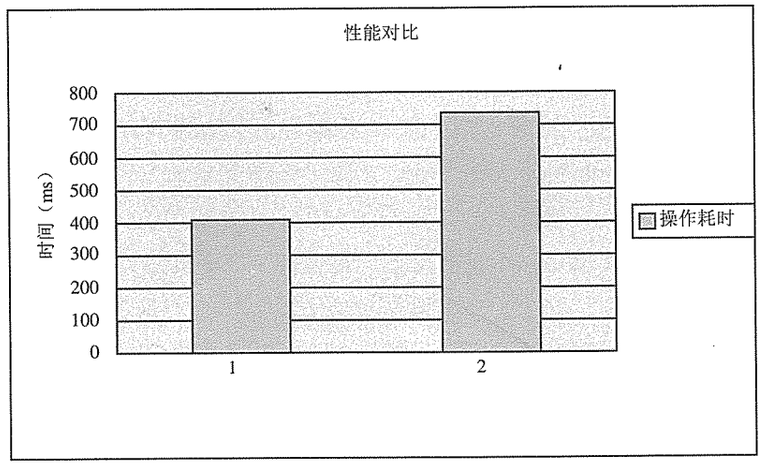


图 2-1　两种方式的性能对比

从图中可以直观地感受到将变量在初始化列表中正确初始化，而不是放置在构造函数的函数体内。从而对性能的影响相当大，因此在写构造函数时应该引起足够的警觉和关注。

2.2　继承与虚拟函数

虚拟函数是 C++语言引入的一个很重要的特性，它提供了“动态绑定”机制，正是这一机制使得继承的语义变得相对明晰。

（1）基类抽象了通用的数据及操作，就数据而言，如果该数据成员在各派生类中都需要用到，那么就需要将其声明在基类中；就操作而言，如果该操作对各派生类都有意义，无论其语义是否会被修改或扩展，那么就需要将其声明在基类中。

（2）有些操作，如果对于各个派生类而言，语义保持完全一致，而无需修改或扩展，那么这些操作声明为基类的非虚拟成员函数。各派生类在声明为基类的派生类时，默认继承了这些非虚拟成员函数的声明/实现，如同默认继承基类的数据成员一样，而不必另外做任何声明，这就是继承带来的代码重用的优点。

（3）另外还有一些操作，虽然对于各派生类而言都有意义，但是其语义并不相同。这时，这些操作应该声明为基类的虚拟成员函数。各派生类虽然也默认继承了这些虚拟成员函数的声明/实现，但是语义上它们应该对这些虚拟成员函数的实现进行修改或者扩展。另外在实现这些修改或扩展过程中，需要用到额外的该派生类独有的数据时，将这些数据声明为此派生类自己的数据成员。

再考虑更大背景下的继承体系，当更高层次的程序框架（继承体系的使用者）使用此继承体系时，它处理的是一个抽象层次的对象集合（即基类）。虽然这个对象集合的成员实质上可能是各种派生类对象，但在处理这个对象集合中的对象时，它用的是抽象层次的操作。并不区分在这些操作中，哪些操作对各派生类来说是保持不变的，而哪些操作对各派生类来说有所不同。这是因为，当运行时实际执行到各操作时，运行时系统能够识别哪些操作需要用到“动态绑定”，从而找到对应此派生类的修改或扩展的该操作版本。

也就是说，对继承体系的使用者而言，此继承体系内部的多样性是“透明的”。它不必关心其继承细节，处理的就是一组对它而言整体行为一致的“对象”。即只需关心它自己问题域的业务逻辑，只要保证正确，其任务就算完成了。即使继承体系内部增加了某种派生类，或者删除了某种派生类，或者某某派生类的某个虚拟函数的实现发生了改变，它的代码不必任何修改。这也意味着，程序的模块化程度得到了极大的提高。而模块化的提高也

就意味着可扩展性、可维护性，以及代码的可读性的提高，这也是“面向对象”编程的一个很大的优点。

下面通过一个简单的实例来展示这一优点。

假设有一个绘图程序允许用户在一个画布上绘制各种图形，如三角形、矩形和圆等，很自然地抽象图形的继承体系，如图 2-2 所示。

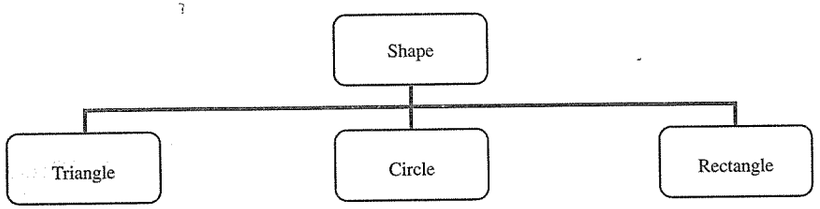


图 2-2　图形的继承体系

这个图形继承体系的设计大致如下：

```
class Shape
{
public:
        Shape();
        virtual ~Shape();

        virtual void Draw();
        virtual void Rotate();

private:
        ...
};

class Triangle : class Shape
{
public:
        Triangle();
        ~Triangle();

        void Draw();
        void Rotate(int angle);
```



```
        ...
};

class Circle : class Shape
{
public:
        Circle();
        ~ Circle();

        void Draw();
        void Rotate(int angle);
    ...
};

class Rectangle : class Shape
{
public:
        Rectangle();
        ~ Rectangle();

        void Draw();
        void Rotate(int angle);
        ...
};
```

为简单起见，让每个 Shape 对象都支持“绘制”和“旋转”操作，每个 Shape 的派生类对这两个操作都有自己的实现：

```
void Triangle::Draw()
{
        ...

}

void Circle::Draw()
{
        ...

}
```

```
void Rectangle::Draw()
{
        ...

}

void Triangle::Rotate(int angle)
{
        ...

}

void Circle::Rotate(int angle)
{
        ...

}

void Rectangle::Rotate(int angle)
{
        ...

}
```

再来考虑这个图形继承体系的使用，这里很自然的一个使用者是画布，设计其类名为“Canvas”：

```
public Canvas
{
public:
        Canvas();
        ~Canvas();

        void Paint();

        void RotateSelected(int angle);

        ...

private:
        ShapeList shapes;
};

...

void Canvas::Paint()
{
        while(shapes.GetNext())
        {
            Shape* sh = shapes.GetNext();
            sh->Draw();                                    ①
            shapes.Next();
        }
        ...
}

void RotateSelected(int angle)
{
        Shape* select_shape = GetCurrentSelected();
        if(select_shape)
            select_shape->Rotate(angle);                   ②
        ...
}
```

Canvas 类中维护一个包含所有图形的 shapes，Canvas 类在处理自己的业务逻辑时并不关心 shapes 实际上都是哪些具体的图形；相反，如①处和②处所示，它只将这些图形作为一个抽象，即 Shape。在处理每个 Shape 时，调用每个 Shape 的某个操作即可。

这样做的一个好处是当图形继承体系发生变化时，作为图形继承体系的使用者 Canvas 而言，它的改变几乎没有，或者很小。

比如说，在程序的演变过程中发现需要支持多边型（Polygon）和贝塞尔曲线（Bezier）类型，只需要在图形继承体系中增加这两个新类型即可：

```
class Polygon : class Shape
{
public:
        Polygon();
        ~Polygon();
```

```
        void Draw();
        void Rotate(int angle);
        ...
};

void Polygon::Draw()
{
        ...

}

void Polygon::Rotate(int angle)
{
        ...

}


class Bezier : class Shape
{
public:
        Bezier();
        ~Bezier();

        void Draw();
        void Rotate(int angle);
        ...
};

void Bezier::Draw()
{
        ...
}

void Bezier::Rotate(int angle)
{
        ...

}
```


而不必修改 Canvas 的任何代码，程序即可像以前那样正常运行。同理，如果以后发现不再支持某种类型，也只需要将其从图形继承体系中删除，而不必修改 Canvas 的任何代码。可以看到，从对象继承体系的使用者（Canvas）的角度来看，它只看到 Shape 对象，而不必关心到底是哪一种特定的 Shape，这是面向对象设计的一个重要特点和优点。

虚拟函数的“动态绑定”特性虽然很好，但也有其内在的空间以及时间开销，每个支持虚拟函数的类（基类或派生类）都会有一个包含其所有支持的虚拟函数指针的“虚拟函数表”（virtual table）。另外每个该类生成的对象都会隐含一个“虚拟函数指针”（virtual pointer），此指针指向其所属类的“虚拟函数表”。当通过基类的指针或者引用调用某个虚拟函数时，系统需要首先定位这个指针或引用真正对应的“对象”所隐含的虚拟函数指针。“虚拟函数指针”，然后根据这个虚拟函数的名称，对这个虚拟函数指针所指向的虚拟函数表进行一个偏移定位，再调用这个偏移定位处的函数指针对应的虚拟函数，这就是“动态绑定”的解析过程（当然 C++规范只需要编译器能够保证动态绑定的语义即可，但是目前绝大多数的 C++编译器都是用这种方式实现虚拟函数的），通过分析，不难发现虚拟函数的开销：

- 空间：每个支持虚拟函数的类，都有一个虚拟函数表，这个虚拟函数表的大小跟该类拥有的虚拟函数的多少成正比，此虚拟函数表对一个类来说，整个程序只有一个，而无论该类生成的对象在程序运行时会生成多少个。
- 空间：通过支持虚拟函数的类生成的每个对象都有一个指向该类对应的虚拟函数表的虚拟函数指针，无论该类的虚拟函数有多少个，都只有一个函数指针，但是因为与对象绑定，因此程序运行时因为虚拟函数指针引起空间开销跟生成的对象个数成正比。
- 时间：通过支持虚拟函数的类生成的每个对象，当其生成时，在构造函数中会调用编译器在构造函数内部插入的初始化代码，来初始化其虚拟函数指针，使其指向正确的虚拟函数表。
- 时间：当通过指针或者引用调用虚拟函数时，跟普通函数调用相比，会多一个根据虚拟函数指针找到虚拟函数表的操作。

内联函数：因为内联函数常常可以提高代码执行的速度，因此很多普通函数会根据情况进行内联化，但是虚拟函数无法利用内联化的优势，这是因为内联函数是在“编译期”

编译器将调用内联函数的地方用内联函数体的代码代替（内联展开），但是虚拟函数本质上是“运行期”行为，本质上在“编译期”编译器无法知道某处的虚拟函数调用在真正执行的时候会调用到那个具体的实现（即在“编译期”无法确定其绑定），因此在“编译期”编译器不会对通过指针或者引用调用的虚拟函数进行内联化。也就是说，如果想利用虚拟函数的“动态绑定”带来的设计优势，那么必须放弃“内联函数”带来的速度优势。

根据上面的分析，似乎在采用虚拟函数时带来和很多的负面影响，但是这些负面影响是否一定是虚拟函数所必须带来的？或者说，如果不采用虚拟函数，是否一定能避免这些缺陷？

还是分析以上图形继承体系的例子，假设不采用虚拟函数，但同时还要实现与上面一样的功能（维持程序的设计语义不变），那么对于基类 Shape 必须增加一个类型标识成员变量用来在运行时识别到底是哪一个具体的派生类对象：

```
class Shape
{
        public:
        Shape();
        virtual ~Shape();

        int GetType() { return type; }                    ①

        void Draw();                                      ③
        void Rotate();                                    ④

private:
        int type;                                         ②
        ...
};
```

如①处和②处所示，增加 type 用来标识派生类对象的具体类型。另外注意这时③处和④处此时已经不再使用 virtual 声明。

其各派生类在构造时，必须设置具体类型，以 Circle 派生类为例：

```
class Circle : class Shape
{
public:
        Circle() : type(CIRCLE) {...}                     ①
```



```
        ~Circle();

        void Draw();
        void Rotate(int angle);
        ...
};
```

对图形继承体系的使用者（这里是 Canvas）而言，其 Paint 和 RotateSelected 也需要修改：

```
void Canvas::Paint()
{
        while(shapes.GetNext())
        {
            Shape* sh = shapes.GetNext();
            //sh->Draw();
            switch(sh->GetType())
            {
                case(TRIANGLE)
                    ((Triangle*)sh)->Draw();
                case(CIRCLE)
                    ((Circle*)sh)->Draw();
                case(RECTANGLE)
                    ((Rectangle*)sh)->Draw();
                ...
            }
            shapes.Next();
        }
        ...
}

void RotateSelected(int angle)
{
        Shape* select_shape = GetCurrentSelected();
        if(select_shape)
        {
            //select_shape->Rotate(angle);
            switch(select_shape->GetType())
            {
                case(TRIANGLE)
```

```
                ((Triangle*)select_shape)->Rotate(angle);
            case(CIRCLE)
                ((Circle*)select_shape)->Rotate(angle);
            case(RECTANGLE)
                ((Rectangle*)select_shape)->Rotate(angle);
            ...
        }
    }
    ...
}
```

因为要实现相同的程序功能（语义），已经看到，每个对象虽然没有编译器生成的虚拟函数指针（析构函数往往被设计为 virtual，如果如此，仍然免不了会隐含增加一个虚拟函数指针，这里假设不是这样），但是还是需要另外增加一个 type 变量用来标识派生类的类型。构造对象时，虽然不必初始化虚拟函数指针，但是仍然需要初始化 type。另外，图形继承体系的使用者调用函数时虽然不再需要一次间接的根据虚拟函数表找寻虚拟函数指针的操作，但是再调用之前，仍然需要一个 switch 语句对其类型进行识别。

综上所述，这里列举的 5 条虚拟函数带来的缺陷只剩下两条，即虚拟函数表的空间开销及无法利用“内联函数”的速度优势。再考虑虚拟函数表，每一个含有虚拟函数的类在整个程序中只会有一个虚拟函数表。可以想像到虚拟函数表引起的空间开销实际上是非常小的，几乎可以忽略不计。

这样可以得出结论，即虚拟函数引入的性能缺陷只是无法利用内联函数。

可以进一步设想，非虚拟函数的常规设计假如需要增加一种新的图形类型，或者删除一种不再支持的图形类型，都必须修改该图形系统所有使用者的所有与类型相关的函数调用的代码。这里使用者只有 Canvas 一个，与类型相关的函数调用代码也只有 Paint 和 RotateSelected 两处。但是在一个复杂的程序中，其使用者很多。并且类型相关的函数调用很多时，每次对图形系统的修改都会波及到这些使用者。可以看出不使用虚拟函数的常规设计增加了代码的耦合度，模块化不强，因此带来的可扩展性、可维护性，以及代码的可读性方面都极大降低。面向对象编程的一个重要目的就是增加程序的可扩展性和可维护性，即当程序的业务逻辑发生变化时，对原有程序的修改非常方便。而不至于对原有代码大动干戈，从而降低因为业务逻辑的改变而增加出错的可能性。根据这点分析，虚拟函数可以大大提升程序的可扩展性及可维护性。

因此在性能和其他方面特性的选择方面，需要开发人员根据实际情况进行权衡和取舍。当然在权衡之前，需要通过性能检测确认性能的瓶颈是由于虚拟函数没有利用到内联函数的优势这一缺陷引起；否则可以不必考虑虚拟函数的影响。

2.3 临时对象

从 2.1 节“构造函数和析构函数”中已经知道，对象的创建与销毁对程序的性能影响很大。尤其当该对象的类处于一个复杂继承体系的末端，或者该对象包含很多成员变量对象（包括其所有父类对象，即直接或者间接父类的所有成员变量对象）时，对程序性能影响尤其显著。因此作为一个对性能敏感的开发人员，应该尽量避免创建不必要的对象，以及随后的销毁。这里“避免创建不必要的对象”，不仅仅意味着在编程时，主要减少显式出现在源码中的对象创建。还有在编译过程中，编译器在某些特殊情况下生成的开发人员看不见的隐式的对象。这些对象的创建并不出现在源码级别，而是由编译器在编译过程中“悄悄”创建（往往为了某些特殊操作），并在适当时销毁，这些就是所谓的“临时对象”。需要注意的是，临时对象与通常意义上的临时变量是完全不同的两个概念，比如下面的代码：

```
void  swap(int  *px,  int  *py)
{
    int temp;                                                    ①

   temp = *px;
   *px = *py;
   *py = temp;
}
```

习惯称①句中的 temp 为临时变量，其目的是为了暂时存放指针 px 指向的 int 型值。但是它并不是这里要考察的“临时对象”，不仅仅是因为一般开发人员不习惯称一个内建类型的变量为“对象”（所以不算临时“对象”）。而且因为 temp 出现在了源码中，这里考察的临时对象并不会出现在源码中。

到底什么才是临时对象？它们在什么时候产生？其生命周期有什么特征？在回答这些问题之前，首先来看下面这段代码：

```
#include <iostream>
#include <cstring>

using namespace std;

class Matrix
{
public:
        Matrix(double d = 1.0)
        {
            cout << "Matrix::Matrix()" << endl;

            for(int i = 0; i < 10; i++)
                for(int j = 0; j < 10; j++)
                    m[i][j] = d;
        }

        Matrix(const Matrix& mt)
        {
            cout << "Matrix::Matrix(const Matrix&)" << endl;
            memcpy(this, &mt, sizeof(Matrix));
        }

        Matrix& operator=(const Matrix& mt)
        {
            if(this == &mt)
                return *this;

            cout << "Matrix::operator=(const Matrix&)" << endl;
            memcpy(this, &mt, sizeof(Matrix));

            return *this;
        }

        friend const Matrix operator+(const Matrix&, const Matrix&);
        //...

private:
        double m[10][10];
};
```



```
const Matrix operator+(const Matrix& arg1, const Matrix& arg2)
{
        Matrix sum;                                                    ①
        for(int i = 0; i < 10; i++)
            for(int j = 0; j < 10; j++)
                sum.m[i][j] = arg1.m[i][j] + arg2.m[i][j];

        return sum;                                                    ②
}

int main()
{
        Matrix a(2.0), b(3.0), c;                                      ③
        c = a + b;                                                     ④

        return 0;
}
```

分析代码，③处生成 3 个 Matrix 对象 a，b，c，调用 3 次 Matrix 构造函数。④处调用 operator+(const Matrix&, const Matrix&)执行到①处时生成临时变量（注意此处的 sum 并不是“临时对象”），调用一次 Matrix 构造函数。④处 c = a + b 最后将 a + b 的结果赋值给 c，调用的是赋值操作，而不会生成新的 Matrix 对象，因此从源码分析，此段代码共生成 4 个 Matrix 对象。

但是输出结果：

```
Matrix::Matrix()                                                       ①
Matrix::Matrix()                                                       ②
Matrix::Matrix()                                                       ③
Matrix::Matrix()                                                       ④
Matrix::Matrix(const Matrix&)                                          ⑤
Matrix::operator=(const Matrix&)                                       ⑥
```

①、②、③3 处输出分别对应对象 a、b 和 c 的构造，④处输出对应的是 operator+(const Matrix&, const Matrix&)中 sum 的构造，⑥处输出对应的是 c = a + b 句中最后用 a + b 的结果向 c 赋值，那么⑤处输出对应哪个对象？

答案是在这段代码中，编译器生成了一个“临时对象”。

a + b 实际上是执行 operator+(const Matrix& arg1, const Matrix& arg2)，重载的操作符本质上是一个函数，这里 a 和 b 就是此函数的两个变量。此函数返回一个 Matrix 变量，然后进一步将此变量通过 Matrix::operator=(const Matrix& mt)对 c 进行赋值。因为 a + b 返回时，其中的 sum 已经结束了其生命周期。即在 operator+(const Matrix& arg1, const Matrix& arg2)结束时被销毁，那么其返回的 Matrix 对象需要在调用 a + b 函数（这里是 main()函数）的栈中开辟空间用来存放此返回值。这个临时的 Matrix 对象是在 a + b 返回时通过 Matrix 拷贝构造函数构造，即⑤处的输出。

既然如上所述，创建和销毁对象经常会成为一个程序的性能瓶颈所在，那么有必要对临时对象产生的原因进行深入探究，并在不损害程序功能的前提下尽可能地规避它。

临时对象在 C++语言中的特征是未出现在源代码中，从堆栈中产生的未命名对象。这里需要特别注意的是，临时对象并不出现在源代码中。即开发人员并没有声明要使用它们，没有为其声明变量。它们由编译器根据情况产生，而且开发人员往往都不会意识到它们的产生。

产生临时对象一般来说有如下两种场合。

（1）当实际调用函数时传入的参数与函数定义中声明的变量类型不匹配。

（2）当函数返回一个对象时（这种情形下也有例外，下面会讲到）。

另外，也有很多开发人员认为当函数传入参数为对象，并且实际调用时因为函数体内的该对象实际上并不是传入的对象，而是该传入对象的一份拷贝，所以认为这时函数体内的那个拷贝的对象也应该是一个临时对象。但是严格说来，这个拷贝对象并不符合“未出现在源代码中”这一特征。当然只要能知道并意识到对象参数的工作原理及背后隐含的性能特征，并能在编写代码时尽量规避之，那么也就没有必要在字面上较真了，毕竟最终目的是写出正确和高效的程序。

因为类型不匹配而生成临时对象的情况，可以通过下面这段程序来认识：

```
class Rational
{
public:
        Rational (int a = 0, int b = 1 ) : m(a), n(b) {}                    ①

private:
        int m;
        int n;
};

...
void foo()
{
    Rational r;
    r = 100;                                                                ②
    ...
}
```

当执行②处代码时，因为 Rational 类并没有重载 operator=(int i)，所以此处编译器会合成一个 operator=(const Rational& r)。并且执行逐位拷贝（bitwise copy）形式的赋值操作，但是右边的一个整型常量 100 并不是一个 Rational 对象，初看此处无法通过编译。但是，需要注意的一点是 C++编译器在判定这种语句不能成功编译前，总是尽可能地查找合适的转换路径，以满足编译的需要。这里，编译器发现 Rational 类有一个如①处所示的 Rational(int a=0, int b=1)型的构造函数。因为此构造函数可以接受 0、1 或 2 个整数作为参数，这时编译器会“贴心”地首先将②式右边的 100 通过调用 Rational::Rational(100, 1)生成一个临时对象，然后用编译器合成的逐位拷贝形式的赋值符对 r 对象进行赋值。②处语句执行后，r 对象内部的 m 为 100，n 为 1。

从上面例子中，可以看到 C++编译器为了成功编译某些语句，往往会在私底下“悄悄”地生成很多从源代码中不易察觉的辅助函数，甚至对象。比如上段代码中，编译器生成的赋值操作符、类型转换，以及类型转换的中间结果，即一个临时对象。

很多时候，这种编译器提供的自动类型转换确实提高了程序的可读性，也在一定程度上简化了程序的编写，从而提高了开发速度。但是类型转换意味着临时对象的产生，对象的创建和销毁意味着性能的下降，类型转换还意味着编译器还需要生成额外的代码等。因此在设计阶段，预计到不需要编译器提供这种自动类型转换的便利时，可以明确阻止这种自动类型转换的发生，即阻止因此而引起临时对象的产生。这种明确阻止就是通过对类的构造函数增加“explicit”声明，如上例中的代码，可以通过如下声明来阻止：

```
class Rational
{
public:
```

```
        explicit Rational (int a = 0, int b = 1 ) : m(a), n(b) {}     ①

private:
        int m;
        int n;
};

...
void foo()
{
   Rational r;
   r = 100;                                                            ②
   ...
}
```

此段代码编译时在②处报一个错误，即“binary '=' : no operator defined which takes a right-hand operand of type 'const int' (or there is no acceptable conversion)”，这个错误说明编译器无法将 100 转换为一个 Rational 对象。编译器合成的赋值运算符只接受 Rational 对象，而不能接受整型。编译器要想能成功编译②处语句，要么提供一个重载的“=”运算符，该运算符接受整型作为参数；要么能够将整型转换为一个 Rational 对象，然后进一步利用编译器合成的赋值运算符。要想将整型转换为一个 Rational 对象，一个办法就是提供能只传递一个整型作为参数的 Rational 构造函数（不一定非要求该构造函数只有一个整型参数，因为考虑到默认值的原因。如上面的例子，Rational 的构造函数接受两个整型参数。但是因为都有默认值，因此调用该构造函数可以有 3 种方式，即无参、一个参数和两个参数），这样编译器就可以用该整型数作为参数调用该构造函数生成一个 Rational 对象（临时对象）。

但是上面没有重载以整型为参数的“=”操作符，虽然提供了一个能只传入一个整型作为参数的构造函数，但是用“explicit”限制了此构造函数。因为 explicit 的含义是开发人员只能显式地根据这个构造函数的定义调用，而不允许编译器利用其来进行隐式的类型转换。这样编译器无办法利用它来将 100 转换为一个临时的 Rational 对象，②处语句也无法编译。

上面提到，可以通过重载以整型为参数的“=”操作符使②处成功编译的目的，看这种方法：


```
class Rational
{
public:
    explicit Rational (int a = 0, int b = 1 ) : m(a), n(b) {}          ①
    Rational& operator=(int a) {m=a; n=1; return *this; }              ③

private:
    int m;
    int n;
};

...
void foo()
{
   Rational r;
   r = 100;                                                            ②
   ...
}
```

如③处所示，重载了“=”操作符。这样当编译②处时，编译器发现右边是一个整型数，它首先寻找是否有与之匹配的重载的“=”操作符。找到③处的声明，及定义。这样它利用③处来调用展开②处为 r.Rational::operator=(100)，顺利通过编译。

需要指出的是，重载“=”操作符后达到了程序想要的效果，即程序的可读性及代码编写的方便性。同时还有一个更重要的效果（对性能敏感的程序而言），即成功避免了一个临时对象的产生。因为“=”操作符的实现，仅仅是修改了被调用对象的内部成员对象，整个过程中都不需要产生临时对象。但是重载“=”操作符也增加了设计类 Rational 的成本，如果一个类可能会支持多种其他类型对它的转换，则需要进行多次重载，这无疑会使得这个类变得十分臃肿。同样，如果一个大型程序有很多这样的类，那么因为代码臃肿引起的维护难度也相应会增加。

因此在设计阶段，在兼顾程序的可读性、代码编写时的方便性、性能，以及程序大小和可维护性时，需要仔细分析和斟酌。尤其要对每个类在该应用程序实际运行时的调用次数及是否在性能关键路径上等情况进行预估和试验，然后做到合理的折衷和权衡。

如前所述，还有一种情形往往导致临时对象的产生，即当一个函数返回的是某个非内建类型的对象时。这时因为返回结果（一个对象）必须要有一个地方存放。所以编译器会

从调用该函数的函数栈桢中开辟空间，并用返回值作为参数调用该对象所属类型的拷贝构造函数在此空间中生成该对象。在被调用函数结束并返回后，可以继续利用此对象（返回值），如：

```
#include <iostream>

using namespace std;

class Rational
{

        friend const Rational operator+(const Rational& a, const Rational& b);

public:
        Rational (int a = 0, int b = 1 ) : m(a), n(b)
        {
            cout << "Rational::Rational(int,int)" << endl;
        }
        Rational (const Rational& r) : m(r.m), n(r.n)
        {
            cout << "Rational::Rational(const Rational& r)" << endl;
        }
        Rational& operator=(const Rational& r)
        {
            if(this == &r)
                return(*this);

            m=r.m;
            n=r.n;

            cout << "Rational::operator=(const Rational& r)" << endl;

            return *this;
        }

private:
        int m;
        int n;
};
```



```
const Rational operator+(const Rational& a, const Rational& b)
{
        cout << "operator+() begin" << endl;
        Rational temp;
        temp.m = a.m + b.m;
        temp.n = a.n + b.n;
        cout << "operator+() end" << endl;
        return temp;                                                    ②
}

int main()
{
        Rational r, a(10,10), b(5,8);
        r = a + b;                                                      ①

        return 0;
}
```

执行①的处语句时，相当于在 main 函数中调用 operator+(const Rational& a, const Rational& b)函数。在 main 函数栈中会开辟一块 Rational 对象大小的空间。在 operator+(const Rational& a, const Rational& b)函数的②处，函数返回被销毁的 temp 对象为参数调用拷贝构造函数在 main 函数栈中开辟的空间中生成一个 Rational 对象，然后在 r=a+b 的“=”部分执行赋值运算符操作，输出如下：

```
Rational::Rational(int,int)
Rational::Rational(int,int)
Rational::Rational(int,int)
operator+() begin
Rational::Rational(int,int)
operator+() end
Rational::Rational(const Rational& r)
Rational::operator=(const Rational& r)
```

但 r 在之前的默认构造后并没有用到，此时可以将其生成延迟，如下所示：

```
#include <iostream>

using namespace std;
```

```
class Rational
{

        friend const Rational operator+(const Rational& a, const Rational& b);

public:
        Rational (int a = 0, int b = 1 ) : m(a), n(b)
        {
            cout << "Rational::Rational(int,int)" << endl;
        }
        Rational (const Rational& r) : m(r.m), n(r.n)
        {
            cout << "Rational::Rational(const Rational& r)" << endl;
        }
        Rational& operator=(const Rational& r)
        {
            if(this == &r)
                return(*this);

            m=r.m;
            n=r.n;

            cout << "Rational::operator=(const Rational& r)" << endl;

            return *this;
        }

private:
        int m;
        int n;
};

const Rational operator+(const Rational& a, const Rational& b)
{
        cout << "operator+() begin" << endl;
        Rational temp;
        temp.m = a.m + b.m;
        temp.n = a.n + b.n;
        cout << "operator+() end" << endl;
        return temp;                                                    ②
```



```
}

int main()
{
        Rational a(10,10), b(5,8);
        Rational r = a + b;                                             ①

        return 0;
}
```

这时输出为：

```
Rational::Rational(int,int)
Rational::Rational(int,int)
operator+() begin
Rational::Rational(int,int)
operator+() end
Rational::Rational(const Rational& r)
```

已经发现，经过简单改写，这段程序竟然减少了一次构造函数和一次赋值操作。为什么？原来改写后，在执行①处时的行为发生了很大的变化。编译器对“=”的解释不再是赋值运算符，而是对象 r 的初始化。在取得 a+b 的结果值时，也不再需要在 main 函数栈桢中另外开辟空间。而是直接使用为 r 对象预留的空间，即编译器在执行②处时直接使用 temp 作为参数调用了 Rational 的拷贝构造函数对 r 对象进行初始化。这样，也消除了临时对象的生成，以及原本发生在①处的赋值运算。

通过这个简单的优化，已经消除了一个临时对象的生成，也减少了一次函数调用（赋值操作符本质上也是一个函数）。这里已经得到一个启示，即对非内建类型的对象，尽量将对象延迟到已经确切知道其有效状态时。这样可以减少临时对象的生成，如上面所示，应写为：

```
Rational r = a + b。
```

而不是：

```
Rational r;
…
r = a + b;
```

当然这里有一个前提，即在 r = a + b 调用之前未用到 r，因此不必生成。

再进一步，已经看到在 operator+(const Rational& a, const Rational& b)实现中用到了一个局部对象 temp，改写如下：

```
#include <iostream>

using namespace std;

class Rational
{

        friend const Rational operator+(const Rational& a, const Rational& b);

public:
        Rational (int a = 0, int b = 1 ) : m(a), n(b)
        {
            cout << "Rational::Rational(int,int)" << endl;
        }
        Rational (const Rational& r) : m(r.m), n(r.n)
        {
            cout << "Rational::Rational(const Rational& r)" << endl;
        }
        Rational& operator=(const Rational& r)
        {
            if(this == &r)
                return(*this);

            m=r.m;
            n=r.n;

            cout << "Rational::operator=(const Rational& r)" << endl;

            return *this;
        }

private:
        int m;
        int n;
};

const Rational operator+(const Rational& a, const Rational& b)
```



```
{
        cout << "operator+() begin" << endl;
        return Rational(a.m + b.m, a.n + b.n);                    ②
}

int main()
{
        Rational a(10,10), b(5,8);
        Rational r = a + b;                                       ①

        return 0;
}
```

这时输出如下：

```
Rational::Rational(int,int)
Rational::Rational(int,int)
operator+() begin
Rational::Rational(int,int)
```

如上，确实消除了 temp。这时编译器在进入 operator+(const Rational& a, const Rational& b)时看到①处是一个初始化，而不是赋值。所以编译器传入参数时，也传入了在 main 函数栈桢中为对象 r 预留的空间地址。当执行到②处时，实际上这个构造函数就是在 r 对象所处的空间内进行的，即构造了 r 对象，这样省去了用来临时计算和存放结果的 temp 对象。

需要注意的是，这个做法需要与前一个优化配合才有效。即 a+b 的结果用来初始化一个对象，而不是对一个已经存在的对象进行赋值操作，如果①处是：

```
r = a + b;
```

那么 operator+(const Rational& a, const Rational& b)的实现中虽然没有用到 temp 对象，但是仍然会在调用函数（这里是 main 函数）的栈桢中生成一个临时对象用来存放计算结果，然后利用这个临时对象对 r 对象进行赋值操作。

对于 operator+(const Rational& a, const Rational& b)函数，常常看到有如下调用习惯：

```
    Rational a, b;
…
    a = a + b;
```

这种写法也经常会用下面这种写法代替：

```
    Rational a, b;
…
    a += b;
```

这两种写法除了个人习惯之外，在性能方面有无区别？回答是有区别。而且有时还会很大，视对象大小而定。因此设计某类时，如果需要重载 operator+，最好也重载 operator+=，并且考虑到维护性，operator+用 operator+=来实现。这样如果这个操作符的语义有所改变需要修改时，只需要修改一处即可。

对 Rational 类来说，一般 operator+=的实现如下：

```
Rational& operator+=(const Rational& rhs)
{
    m += rhs.m;
    n += rhs.n;
    return (*this);
}
```

这里可以看到，与 operator+不同，operator+=并没有产生临时变量，operator+则只有在返回值被用来初始化一个对象，而不是对一个已经生成的对象进行赋值时才不产生临时对象。而且往往返回值被用来赋值的情况并不少见，甚至比初始化的情况还要多。因此使用 operator+=不产生临时对象，性能会比 operator+要好，为此尽量使用语句：

```
a += b;
```

而避免使用：

```
a = a + b;
```

相应地，也应考虑到程序的代码可维护性（易于修改，因为不小心的修改会导致不一致等）。即尽量利用 operator+=来实现 operator+，如下：

```
const Rational operator+(const Rational& a, const Rational& b)
{
        return Rational(a) += b;
}
```

同理，这个规律可以扩展到-=、*=和/=等。

操作符中还有两个比较特殊的，即++和--。它们都可以前置或者后置，比如 i++和++i。

二者的语义是有区别的，前者先将其值返回，然后其值增 1；后者则是先将值增 1，再返回其值。但当不需要用到其值，即单独使用时，比如：

```
i++;
++i;
```

二者的语义则是一样的，都是将原值增 1。但是对于一个非内建类型，在重载这两个操作符后，单独使用在性能方面是否有差别？来考察它们的实现。仍以 Rational 类作为例子，假设++的语义为对分子（即 m）增 1，分母不变（暂且不考虑这种语义是否符合实际情况），那么两个实现如下：

```
const Rational& operator++()    //prefix
{
    ++m;
    return (*this);
}

const Rational operator++(int)  //postfix
{
    Rational tmp(*this);                                    ①
    ++(*this);
    return tmp;
}
```

可以看到，因为考虑到后置++的语义，所以在实现中必须首先保留其原来的值。为此需要一个局部变量，如①处所示。然后值增 1 后，将保存其原值的局部变量作为返回值返回。相比较而言，前置++的实现不会需要这样一个局部变量。而且不仅如此，前置的++只需要将自身返回即可，因此只需返回一个引用；后置++需要返回一个对象。已经知道，函数返回值为一个对象时，往往意味着需要生成一个临时对象用来存放返回值。因此如果调用后置++，意味着需要多生成两个对象，分别是函数内部的局部变量和存放返回值的临时变量。

有鉴于此，对于非内建类型，在保证程序语义正确的前提下应该多用：

```
++i;
```

而避免使用：

```
i++;
```

同样的规律也适用于前置--和后置--（与=/+=相同的理由，考虑到维护性，尽量用前置++来实现后置++）。

至此，已经考察了临时对象的含义、产生临时对象的各种场合，以及一些避免临时对象产生的方法。最后来查看临时对象的生命周期。在 C++规范中定义一个临时对象的生命周期为从创建时开始，到包含创建它的最长语句执行完毕，比如：

```
string a, b;
const char* str;
…
if( strlen( str = (a + b).c_str() ) > 5)                    ①
{
        printf("%s\n", str);                                ②
        …
}
```

在①处，首先创建一个临时对象存放 a+b 的值。然后从这个临时 string 对象中通过 c_str() 函数得到其字符串内容，赋给 str。如果 str 的长度大于 5，就会进入 if 内部，执行②处语句。问题是，这时的 str 还合法否？

答案是否定的，因为存放 a+b 值的临时对象的生命在包含其创建的最长语句结束后也相应结束了，这里是①处语句。当执行到②处时，该临时对象已经不存在，指向它内部字符串内容的 str 指向的是一段已经被回收的内存。这时的结果是无法预测的，但肯定不是所期望的。

但这条规范也有一个特例，当用一个临时对象来初始化一个常量引用时，该临时对象的生命会持续到与绑定到其上的常量引用销毁时，如：

```
string a, b;
…
if( …)
{
        const string& c = a + b                             ①
        cout << c << endl;                                  ②
        …
}
```

这时 c 这个常量 string 引用在①处绑定在存放 a+b 结果的临时对象后，可以继续在其使用域（scope）内正常使用，如在②处语句中那样。这是因为 c 是一个常量引用，因为被

它绑定。所以存放 a+b 的临时对象并不会在①处语句执行后销毁，而是保持与 c 一样的生命周期。

2.4 内联函数

在 C++语言的设计中，内联函数的引入可以说完全是为了性能的考虑。因此在编写对性能要求比较高的 C++程序时，非常有必要仔细考量内联函数的使用。

所谓“内联”，即将被调用函数的函数体代码直接地整个插入到该函数被调用处，而不是通过 call 语句进行。当然，编译器在真正进行“内联”时，因为考虑到被内联函数的传入参数、自己的局部变量，以及返回值的因素，不仅仅只是进行简单的代码拷贝，还需要做很多细致的工作，但大致思路如此。

开发人员可以有两种方式告诉编译器需要内联哪些类成员函数，一种是在类的定义体外；一种是在类的定义体内。

（1）当在类的定义体外时，需要在该成员函数的定义前面加“inline”关键字，显式地告诉编译器该函数在调用时需要“内联”处理，如：

```
class Student
{
public:
        String   GetName();
        int      GetAge();
        void     SetAge(int ag);
        ……
private:
        String   name;
        int      age;
        ……
};

inline String GetName()
{
        return name;
}
```

```
inline int GetAge()
{
        return age;
}

inline void SetAge(int ag)
{
        age = ag;
}
```

（2）当在类的定义体内且声明该成员函数时，同时提供该成员函数的实现体。此时，“inline”关键字并不是必需的，如：

```
class Student
{
public:
        String  GetName()       { return name; }
        int     GetAge()        { return age; }
        void    SetAge(int ag)  { age = ag; }
        ……
private:
        String  name;
        int     age;
        ……
};
```

当普通函数（非类成员函数）需要被内联时，则只需要在函数的定义时前面加上“inline”关键字，如：

```
    inline int DoSomeMagic(int a, int b)
{
        return a * 13 + b % 4 + 3;
}
```

因为 C++是以“编译单元”为单位编译的，而一个编译单元往往大致等于一个“.cpp”文件。在实际编译前，预处理器会将“#include”的各头文件的内容（可能会有递归头文件展开）完整地拷贝到 cpp 文件对应位置处（另外还会进行宏展开等操作）。预处理器处理后，编译真正开始。一旦 C++编译器开始编译，它不会意识到其他 cpp 文件的存在。因此并不会参考其他 cpp 文件的内容信息。联想到内联的工作是由编译器完成的，且内联的意思是

将被调用内联函数的函数体代码直接代替对该内联函数的调用。这也就意味着，在编译某个编译单元时，如果该编译单元会调用到某个内联函数，那么该内联函数的函数定义（即函数体）必须也包含在该编译单元内。因为编译器使用内联函数体代码替代内联函数调用时，必须知道该内联函数的函数体代码，而且不能通过参考其他编译单元信息来获得这一信息。

如果有多个编译单元会调用到某同一个内联函数，C++规范要求在这多个编译单元中该内联函数的定义必须是完全一致的，这就是“ODR”（one-definition rule）原则。考虑到代码的可维护性，最好将内联函数的定义放在一个头文件中，用到该内联函数的各个编译单元只需#include 该头文件即可。进一步考虑，如果该内联函数是一个类的成员函数，这个头文件正好可以是该成员函数所属类的声明所在的头文件。这样看来，类成员内联函数的两种声明可以看成是几乎一样的，虽然一个是在类外，一个在类内。但是两个都在同一个头文件中，编译器都能在#include 该头文件后直接取得内联函数的函数体代码。讨论完如何声明一个内联函数，来查看编译器如何内联的。继续上面的例子，假设有个 foo 函数：

```
#include "student.h"
...

void foo()
{
        ...
        Student abc;
        abc.SetAge(12);
        cout << abc.GetAge();
        ...
}
```

foo 函数进入 foo 函数时，从其栈帧中开辟了放置 abc 对象的空间。进入函数体后，首先对该处空间执行 Student 的默认构造函数构造 abc 对象。然后将常数 12 压栈，调用 abc 的 SetAge 函数（开辟 SetAge 函数自己的栈帧，返回时回退销毁此栈帧）。紧跟着执行 abc 的 GetAge 函数，并将返回值压栈。最后调用 cout 的<<操作符操作压栈的结果，即输出。

内联后大致如下：

```
#include "student.h"
...
```

```
void foo()
{
        ...
        Student abc;
        {
            abc.age = 12;
        }
        int tmp = abc.age;
        cout << tmp;
        ...
}
```

这时，函数调用时的参数压栈、栈帧开辟与销毁等操作不再需要，而且在结合这些代码后，编译器能进一步优化为如下结果：

```
#include "student.h"
...

void foo()
{
        ...
        cout << 12;
        ...
}
```

这显然是最好的优化结果；相反，考虑原始版本。如果 SetAge/GetAge 没有被内联，因为非内联函数一般不会在头文件中定义，这两个函数可能在这个编译单元之外的其他编译单元中定义。即 foo 函数所在编译单元看不到 SetAge/GetAge，不知道函数体代码信息，那么编译器传入 12 给 SetAge，然后用 GetAge 输出。在这一过程中，编译器不能确信最后 GetAge 的输出。因为编译这个编译单元时，不知道这两个函数的函数体代码，因而也就不能做出最终版本的优化。

从上述分析中，可以看到使用内联函数至少有如下两个优点。

（1）减少因为函数调用引起开销，主要是参数压栈、栈帧开辟与回收，以及寄存器保存与恢复等。

（2）内联后编译器在处理调用内联函数的函数（如上例中的 foo()函数）时，因为可供分析的代码更多，因此它能做的优化更深入彻底。前一条优点对于开发人员来说往往更显

而易见一些，但往往这条优点对最终代码的优化可能贡献更大。

这时，有必要简单介绍函数调用时都需要执行哪些操作，这样可以帮助分析一些函数调用相关的问题。假设下面代码：

```
void foo()
{
        ...
        i = func(a, b, c);                    ①
        ...                                   ②
}
```

调用者（这里是 foo）在调用前需要执行如下操作。

（1）参数压栈：这里是 a、b 和 c。压栈时一般都是按照逆序，因此是 c->b->c。如果 a、b 和 c 有对象，则需要先进行拷贝构造（前面章节已经讨论）。

（2）保存返回地址：即函数调用结束返回后接着执行的语句的地址，这里是②处语句的地址。

（3）保存维护 foo 函数栈帧信息的寄存器内容：如 SP（堆栈指针）和 FP（栈帧指针）等。到底保存哪些寄存器与平台相关，但是每个平台肯定都会有对应的寄存器。

（4）保存一些通用寄存器的内容：因为有些通用寄存器会被所有函数用到，所以在 foo 调用 func 之前，这些寄存器可能已经放置了对 foo 有用的信息。这些寄存器在进入 func 函数体内执行时可能会被 func 用到，从而被覆写。因此 foo 在调用 func 前保存一份这些通用寄存器的内容，这样在 func 返回后可以恢复它们。

接着调用 func 函数，它首先通过移动栈指针来分配所有在其内部声明的局部变量所需的空间，然后执行其函数体内的代码等。

最后当 func 执行完毕，函数返回时，foo 函数还需要执行如下善后处理。

（1）恢复通用寄存器的值。

（2）恢复保存 foo 函数栈帧信息的那些寄存器的值。

（3）通过移动栈指针，销毁 func 函数的栈帧，

（4）将保存的返回地址出栈，并赋给 IP 寄存器。

（5）通过移动栈指针，回收传给 func 函数的参数所占用的空间。

在前面章节中已经讨论，如果传入参数和返回值为对象时，还会涉及对象的构造与析构，函数调用的开销就会更大。尤其是当传入对象和返回对象是复杂的大对象时，更是如此。

因为函数调用的准备与善后工作最终都是由机器指令完成的，假设一个函数之前的准备工作与之后的善后工作的指令所需的空间为 SS，执行这些代码所需的时间为 TS，现在可以更细致地从空间与时间两个方面来分析内联的效果。

（1）在空间上，一般印象是不采用内联，被调用函数的代码只有一份，调用它的地方使用 call 语句引用即可。而采用内联后，该函数的代码在所有调用其处都有一份拷贝，因此最后总的代码大小比采用内联前要大。但事实不总是这样的，如果一个函数 a 的体代码大小为 AS，假设 a 函数在整个程序中被调用了 n 次，不采用内联时，对 a 的调用只有准备工作与善后工作两处会增加最后的代码量开销，即 a 函数相关的代码大小为：n * SS + AS。采用内联后，在各处调用点都需要将其函数体代码展开，即 a 函数相关的代码大小为 n * AS。这样比较二者的大小，即比较(n * SS + AS)与(n*AS)的大小。考虑到 n 一般次数很多时，可以简化成比较 SS 与 AS 的大小。这样可以得出大致结论，如果被内联函数自己的函数体代码量比因为函数调用的准备与善后工作引入的代码量大，内联后程序的代码量会变大；相反，当被内联函数的函数体代码量比因为函数调用的准备与善后工作引入的代码量小，内联后程序的代码量会变小。这里还没有考虑内联的后续情况，即编译器可能因为获得的信息更多，从而对调用函数的优化做得更深入和彻底，致使最终的代码量变得更小。

（2）在时间上，一般而言，每处调用都不再需要做函数调用的准备与善后工作。另外内联后，编译器在做优化时，看到的是调用函数与被调用函数连成的一大块代码。即获得的代码信息更多，此时它对调用函数的优化可以做得更好。最后还有一个很重要的因素，即内联后调用函数体内需要执行的代码是相邻的，其执行的代码都在同一个页面或连续的页面中。如果没有内联，执行到被调用函数时，需要跳到包含被调用函数的内存页面中执行，而被调用函数所属的页面极有可能当时不在物理内存中。这意味着，内联后可以降低“缺页”的几率，知道减少“缺页”次数的效果远比减少一些代码量执行的效果。另外即使被调用函数所在页面可能也在内存中，但是因为与调用函数在空间上相隔甚远，所以可能会引起“cache miss”，从而降低执行速度。因此总的来说，内联后程序的执行时间会比没有内联要少。即程序的速度更快，这也是因为内联后代码的空间“locality”特性提高了。

但正如上面分析空间影响时提到的，当 AS 远大于 SS，且 n 非常大时，最终程序的大小会比没有内联时要大很多。代码量大意味着用来存放代码的内存页也会更多，这样因为执行代码而引起的“缺页”也会相应增多。如果这样，最终程序的执行时间可能会因为大量的“缺页”而变得更多，即程序的速度变慢。这也是为什么很多编译器对于函数体代码很多的函数，会拒绝对其进行内联的请求。即忽略“inline”关键字，而对如同普通函数那样编译。

综合上面的分析，在采用内联时需要内联函数的特征。比如该函数自己的函数体代码量，以及程序执行时可能被调用的次数等。当然，判断内联效果的最终和最有效的方法还是对程序的大小和执行时间进行实际测量，然后根据测量结果来决定是否应该采用内联，以及对哪些函数进行内联。

如下根据内联的本质来讨论与其相关的一些其他特点。

如前所述，因为调用内联函数的编译单元必须有内联函数的函数体代码信息。又因为 ODR 规则和考虑到代码的可维护性，所以一般将内联函数的定义放在一个头文件中，然后在每个调用该内联函数的编译单元中#include 该头文件。现在考虑这种情况，即在一个大型程序中，某个内联函数因为非常通用，而被大多数编译单元用到对该内联函数的一个修改，就会引起所有用到它的编译单元的重新编译。对于一个真正的大型程序，重新编译大部分编译单元往往意味着大量的编译时间。因此内联最好在开发的后期引入，以避免可能不必要的大量编译时间的浪费。

再考虑这种情况，如果某开发小组在开发中用到了第三方提供的程序库，而这些程序库中包含一些内联函数。因为该开发小组的代码中在用到第三方提供的内联函数处，都是将该内联函数的函数体代码拷贝到调用处，即该开发小组的代码中包含了第三方提供代码的“实现”。假设这个第三方单位在下一个版本中修改了某些内联函数的定义，那么虽然这个第三方单位并没有修改任何函数的对外接口，而只是修改了实现，该开发小组要想利用这个新的版本，仍然需要重新编译。考虑到可能该开发小组的程序已经发布，那么这种重新编译的成本会相当高；相反，如果没有内联，并且仍然只是修改实现，那么该开发小组不必重新编译即可利用新的版本。

因为内联的本质就是用函数体代码代替对该函数的调用，所以考虑递归函数，如：

```
[inline] int foo(int n)
{
```

```
        ...
        return foo(n-1);
}
```

如果编译器编译某个调用此函数的编译单元，如：

```
void func()
{
        ...
        int m = foo(n);
        ...
}
```

考虑如下两种情况。

（1）如果在编译该编译单元且调用 foo 时，提供的参数 n 不能知道其实际值，则编译器无法知道对 foo 函数体进行多少次代替。在这种情况下，编译器会拒绝对 foo 函数进行内联。

（2）如果在编译该编译单元且调用 foo 时，提供的参数 n 能够知道其实际值，则编译器可能会视 n 值的大小来决定是否对 foo 函数进行内联。因为如果 n 很大，内联展开可能会使最终程序的大小变得很大。

如前所述，因为内联函数是编译期行为，而虚拟函数是执行期行为，因此编译器一般会拒绝对虚拟函数进行内联的请求。但是事情总有例外，内联函数的本质是编译器编译调用某函数时，将其函数体代码代替 call 调用，即内联的条件是编译器能够知道该处函数调用的函数体。而虚拟函数不能够被内联，也是因为在编译时一般来说编译器无法知道该虚拟函数到底是哪一个版本，即无法确定其函数体。但是在两种情况下，编译器是能够知道虚拟函数调用的真实版本的，因此虚拟函数可以被内联。

其一是通过对象，而不是指向对象的指针或者对象的引用调用虚拟函数，这时编译器在编译期就已经知道对象的确切类型。因此会直接调用确定的某虚拟函数实现版本，而不会产生“动态绑定”行为的代码。

其二是虽然是通过对象指针或者对象引用调用虚拟函数，但是编译时编译器能知道该指针或引用对应到的对象的确切类型。比如在产生的新对象时做的指针赋值或引用初始化，发生在于通过该指针或引用调用虚拟函数同一个编译单元并且二者之间该指针没有被改变赋值使其指向到其他不能确切知道类型的对象（因为引用不能修改绑定，因此无此之虞）。

此时编译器也不会产生动态绑定的代码，而是直接调用该确定类型的虚拟函数实现版本。

在这两种情况下，编译器能够将此虚拟函数内联化，如：

```
inline virtual int x::y (char* a)
{
   ...
}

void z (char* b)
{
   x_base* x_pointer = new x(some_arguments_maybe);
   x x_instance(maybe_some_more_arguments);
   x_pointer->y(b);
   x_instance.y(b);
```

当然在实际开发中，通过这两种方式调用虚拟函数时应该非常少，因为虚拟函数的语义是“通过基类指针或引用调用，到真正运行时才决定调用哪个版本”。

从上面的分析中已经看到，编译器并不总是尊重“inline”关键字。即使某个函数用“inline”关键字修饰，并不能够保证该函数在编译时真正被内联处理。因此与 register 关键字性质类似，inline 仅仅是给编译器的一个“建议”，编译器完全可以视实际情况而忽略之。

另外从内联，即用函数体代码替代对该函数的调用这一本质看，它与 C 语言中的函数宏（macro）极其相似，但是它们之间也有本质的区别。即内联是编译期行为，宏是预处理期行为，其替代展开由预处理器来做。也就是说编译器看不到宏，更不可能处理宏。另外宏的参数在其宏体内出现两次或两次以上时经常会产生副作用，尤其是当在宏体内对参数进行++或—操作时，而内联不会。还有，预处理器不会也不能对宏的参数进行类型检查。而内联因为是编译器处理的，因此会对内联函数的参数进行类型检查，这对于写出正确且鲁棒的程序，是一个很大的优势。最后，宏肯定会被展开，而用 inline 关键字修饰的函数不一定会被内联展开。

最后顺带提及，一个程序的惟一入口 main()函数肯定不会被内联化。另外，编译器合成的默认构造函数、拷贝构造函数、析构函数，以及赋值运算符一般都会被内联化。

2.5 本章小结

相对 C 语言而言，C++语言确实引入了很多新的语言特性。而很多开发人员在遇到用C++语言编写的应用程序性能问题时，也往往会倾向于将性能问题归咎于这些新的语言特性，但实际情形往往并不是这样的。对待性能问题，我们应该采取一个客观的态度。在遇到性能问题并做出真正的性能测量之前，不要轻易假定瓶颈所在。往往很多时候，应用程序的性能是因为该程序的功能和复杂度引起的，而非语言特性本身。如果实际的性能测量证明瓶颈确实是因为某些语言特性引起的，这时需要对该语言特性的使用场合进行仔细分析，然后在不损害其带来的设计任务的前提下进行性能改善。本章着重分析了几个可能会对性能引起下降的语言特性，包括构造函数/析构函数、继承与虚拟、临时对象，以及内联函数，对它们的深刻理解常常能够在编码阶段避免很多性能问题。

第 3 章 常用数据结构的性能分析

第 1 章和第 2 章详细介绍了 C++的对象模型和与性能相关的主要特性，本章将结合这些特性从数据结构的角度分析如何提高程序的性能。

我们知道，各种不同的数据结构组合在一起构成了一个程序的基本框架。在同样的需求下使用不同的数据结构，程序的性能表现往往大相径庭。例如，分别对存有大量节点的链表和数组进行排序。根据不同的实际情况选择或创建一种合适的数据结构，并结合相应的算法实现各种操作，是保证程序执行效率的重要前提。

本章首先根据各种实用操作（遍历、插入、删除、排序及查找），结合程序实例对常用数据结构的性能进行分析。然后结合 C++的对象模型，介绍一种大型软件中常用的数据结构——动态数组。

3.1 常用数据结构性能分析

本节将从程序执行性能和内存占用的角度对几种最常用的数据结构及相关操作进行分析，并结合程序实例为读者提供有效使用这些数据结构的方法。

1. 数组

数组是开发人员最熟悉也是最常用的一种线性表，对于静态或能事先确定大小的数据集合，采用数组进行存储是最佳选择。

数组的优点一是查找方便，利用下标即可立即定位到所需的数据节点；二是添加或删除元素时不会产生内存碎片；三是不需要考虑数据节点指针的存储（而指针在链表和树中经常被使用）。然而作为一种静态的数据结构，数组也有内存使用率低及可扩展性差的缺点。无论数组中实际有多少个元素，编译器总是根据事先设定好的容量分配内存。如果超出边界，则需要建立新的数组。

2. 链表

链表是另一种常用的线性表，一个链表其实就是一个由指针连接的数据链（Chain）。每个数据节点由指针域和数据域组成，指针一般指向链表中的下一个节点。如果这个节点是链表中的最后一个元素，指针则为 NULL。在双向链表（Double Linked List）中，指针域还包括一个指向上一节点的指针。而在跳转链表（Skip Linked list）中，指针域包含指向任意某个关联项的指针。

```
template<class E>
class LinkNode
{
      E data;
      LinkNode<E>* pNext; //指向下一个节点的指针
      LinkNode<E>* pPrev; //指向上一个节点的指针
      LinkNode<E>* pConnection; //指向关联节点的指针
public:
      LinkNode(const E& e):pNext(NULL),pPrev(NULL) { data = e; };
      LinkNode<E>* Next() const { return pNext; };
      LinkNode<E>* Prev() const { return pPrev; };
};
```


与事先静态分配好存储空间的数组不同，链表的长度是可变的。只要内存空间足够，程序就能不断地为链表插入新的数据项。另外，在数组中所有的数据项都被存放在一段连续的存储空间中，而链表中的数据项则可能随机地被分配到内存中的某个位置。

3. 哈希表

数组和链表有各自的优缺点，数组能够方便地定位到任何数据项，但是扩展性较差。而链表则恰恰相反，插入和删除任意一个节点非常简单，但是无法提供快捷的数据项定位。当需要处理较大规模的数据集合时，常常希望能将二者的优点结合在一起。哈希表便是这样一种数据结构，通过结合数组和链表的特点，能够达到较好的扩展性和较高的访问效率。虽然每个开发人员都可能有自己构建哈希表的方法，但是它们有共同的基本结构，如图 3-1 所示。

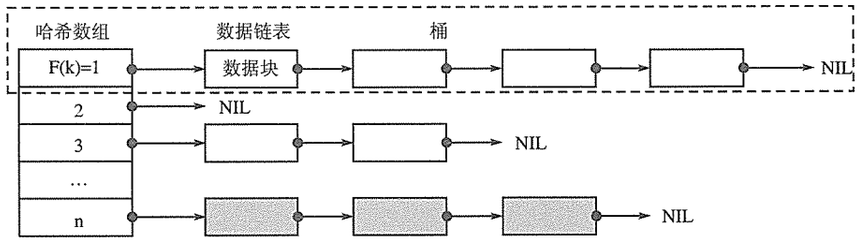


图 3-1 哈希表的基本结构

哈希数组的每一项都有指针指向一个小的链表，与某项相关的所有数据节点都被存储在链表之中。当程序需要访问某个数据节点时，它不需要遍历整个哈希表。而是首先找到数组中的项，然后查询“子链表”找到目标节点。每个“子链表”被称为一个桶（Bucket），例如，图 3-1 中的哈希表共有 n 个桶。如何定位一个存储目标节点的桶，由数据节点的“关键字”域（Key）和哈希函数共同决定。虽然有多种映射方法，但实现哈希函数最常用的方法还是除法映射。除法函数的形式如下：

```
F(k) = k % D
```

其中 k 是数据节点的关键字，D 是一个事先设定的常量，F(k)是桶的序号（等同于哈希数组中每个项的下标）。以下是该哈希表的实现：

```
//数据节点的定义
template<class E,class Key>
```

```
class LinkNode
{
      //数据节点的指针域，这是一个双向链表
      LinkNode<E,Key>* pNext;
      LinkNode<E,Key>* pPrev;
      //节点的数据域，包括数据和关键字
      Key key;
      E  data;
public:
      //构造函数
      LinkNode(const E& e, const Key& k): pNext(NULL), pPrev(NULL)
      {
          data = e;
          key = k;
      };
      //析构函数
      ~LinkNode();
      //设置以及读取指针
      void SetNextNode(LinkNode<E,Key>* next){ pNext = next;}
      LinkNode<E,Key>* Next() const { return pNext;};
      void SetPrevNode(LinkNode<E,Key>* prev){ pPrev = prev;};
      LinkNode<E,Key>* Prev() const { return pPrev;};
      //读取数据
      E&  GetData() const { return data; };
      Key& GetKey()  const { return key; };
};
//哈希表定义
template<class E,class Key>
class HashTable
{
      typedef LinkNode<E,Key>* PtrLinkNode;
      PtrLinkNode* hash_array;//哈希数组
      int size;//哈希数组大小
public:
      //构造函数，初始大小默认设为 100
      HashTable(int sz = 100);
      ~HashTable();
      bool Insert(const E& data);
      bool Delete(const Key& k)。
      bool Search(const Key& k,E& ret) const;
```



```
private:
      PtrLinkNode SearchNode(const Key& k) const;
      //哈希函数，取关键值与数组大小的模
      int HashFun(const Key& k){ return k % size;};
};
//哈希表的构造函数
template<class E,class Key>
HashTable::HashTable( int sz )
{
      size = sz;
      hash_array = new PtrLinkNode[size];//初始化哈希数组大小
      //将每个项设初值为 NULL
      for(int i=0;i<size;i++)
         hash_array[i]= NULL;
}
//哈希表的析构函数
template<class E,class Key>
HashTable::~HashTable()
{
    for(int i=0;i<size;i++)
    {
       if(hash_array[i]!=NULL)
       {
          //释放每个桶的内存
          PtrLinkNode p = hash_array[i];
          while(p)
          {
              PtrLinkNode pDel = p;
              p = p->Next();
              delete pDel;
          }
       }
    }
    delete []hash_array;
}
```

不难发现，哈希函数决定了一个哈希表的效率和性能。例如，如图 3-2 所示。当 $f(k) = k$ 时，哈希表中的每个桶有且仅有一个节点。此时哈希表就像一个一维数组，虽然每个数据节点的指针会造成一定内存空间的浪费，但是查找效率最高（时间复杂度为 $O(1)$）。

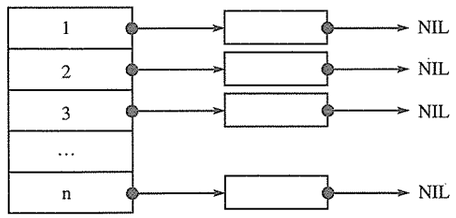


图 3-2　$f(k) = k$ 时的哈希表

另一个极端的例子是当 $f(k)$ = c（c 为常量）时，哈希表中所有的节点都存放在同一个桶中。此时哈希表已退化成链表，同时还加上一个多余、基本上为空的数组。这时查找一个节点的时间复杂度为 $O(n)$，效率最低，如图 3-3 所示。

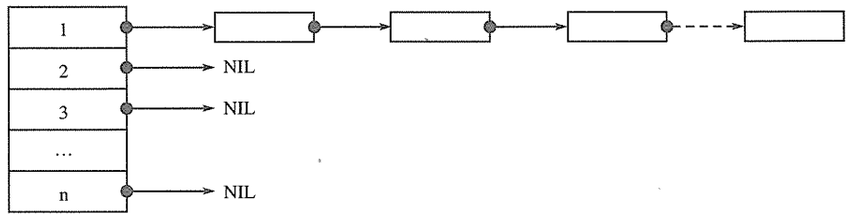


图 3-3　$f(k) = c$ 时的哈希表

由此可见，一方面，构建一个理想的哈希表，需要尽可能地使用让数据节点分配更均匀的哈希函数；另一方面，哈希表本身的结构也是影响其性能的一个重要因素。例如，桶的数量太少会造成巨大的链表，导致查找效率低下；太多的桶则会导致内存的浪费。好的办法是在设计和实现哈希表之前，分析数据节点的关键值，根据其分布来决定到底构造多大的哈希数组并采用什么样的哈希函数。

本节介绍的哈希表实例只是一种比较通用和简单的实现，事实上数据节点的组织方式多种多样，并不局限于链表。一个桶可以是一棵树，甚至可以是另一个哈希表。不管采用什么样的结构，目的只有一个，即使哈希表达到最佳性能以适应程序要求。

4．二叉树

二叉树是一种很常用的数据结构，其中最为开发人员熟知的是二叉查找树。在一棵二叉查找树中，所有节点的左子节点的关键值都小于（或等于）本身，而右子节点的关键值都大于（或等于）本身。由于平衡二叉查找树的搜索算法与有序数组的折半查找算法原理相同，所以其查询效率要远远高于链表（$O(\mathrm{Log}_2 n)$，而链表为 $O(n)$）。但由于树中每个节点都要保存两个指向子节点的指针，空间代价则要高于单向链表和数组。另外，与链表相同，树中每个节点的内存分配也是不连续的，这将导致内存碎片。即便如此，二叉树在插入、删除及查找等操作上的良好表现仍使其成为最常使用的数据结构之一。二叉树的链表实现如下：

```
template<class E>
class TreeNode
{
        E data;
        TreeNode<E>* leftnode;
        TreeNode<E>* rightnode;
public:
        TreeNode(const E& e):leftnode(NULL),rightnode(NULL) { date = e; };
        TreeNode<E>* Left() const { return leftnode; };
        TreeNode<E>* Right() const { return rightnode; };
};
```

下面将根据遍历、插入、删除、排序、查找 5 个基本操作，分别对数组、链表、哈希表和二叉树的性能做进一步分析。

3.1.1　遍历

1．数组

遍历数组的操作很简单，无论是顺序（array[i++]）或是逆序（array[i--]）访问都能遍历整个数组。而且从任意的初始位置 k 开始，也能很方便地遍历整个数组：

```
int idx = k;//k 是初始位置
do
{
    DoSomething(array[idx++]);
    idx = idx % n;
}while(idx!=k);
```

2．链表

跟踪指针便能完成整个链表的遍历：

```
LinkNode<E>* pNode = pFirst;
while(pNode)
{
     pNode = pNode->Next();
     DoSomething(pNode);
}
```

双向链表可以支持顺序和逆序遍历。而对于跳转链表，通过过滤某些无用节点可以快速地实现遍历。

3. 哈希表

如果我们事先知道所有节点的 Key 值，那么可以通过这些值和哈希函数找到每一个非空的桶，然后遍历这个桶的链表；否则只能通过遍历哈希数组的方式访问每个桶：

```
for(int i=0;i<size;i++)
{
     PtrLinkNode pNode = hash_array[i];
     while(pNode!=NULL)
     {
          DoSomething(pNode);
          pNode = pNode->Next();
     }
}
```

4. 二叉树

遍历二叉树主要有 3 种方法，即前序、中序和后序遍历，下面列出这 3 种遍历方法的递归实现：

```
//前序遍历
template<class E>
void PreTraverse(TreeNode<E>* pNode)
{
     if(pNode!=NULL)
     {
          DoSomething(pNode);
          PreTraverse(pNode->Left());
          PreTraverse(pNode->Right());
     }
}
```



```
//中序遍历
template<class E>
void InTraverse(TreeNode<E>* pNode)
{
     if(pNode!=NULL)
     {
        InTraverse(pNode->Left());
        DoSomething(pNode);
        InTraverse(pNode->Right());
     }
}
//后序遍历
template<class E>
void PostTraverse(TreeNode<E>* pNode)
{
     if(pNode!=NULL)
     {
         PostTraverse(pNode->Left());
         PostTraverse(pNode->Right());
         DoSomething(pNode);
     }
}
```

使用递归方法进行遍历的主要问题是：随着树的深度增加，程序对函数栈空间的使用越来越多，由于栈空间大小有限，递归遍历方法可能导致内存耗尽的问题。解决办法主要有两个：（1）使用非递归算法实现前序、中序和后序遍历，即仿照递归算法执行时函数工作栈的变化状况，建立一个栈对当前遍历路径上的节点进行记录，根据栈顶元素是否存在左/右子节点的不同情况，决定下一步操作（将子节点入栈或是将当前节点退栈），从而完成对二叉树的遍历。（2）使用线索二叉树，即根据遍历规则，在每个叶子节点上增加指向后续节点的指针。

3.1.2 插入

1. 数组

由于数组中的所有数据节点都保存在一片连续的存储区中，所以当插入新节点时常常需要移动插入位置之后的所有节点以腾出空间，才能正确地将新节点复制到该位置。如果

恰好数组已满，还需要重新建立一个新的且容量更大的数组，将原数组所有的节点拷贝过来。因此数组的插入操作与其他数据结构相比，时间（甚至空间）复杂度较高。

如果向一个未满的数组插入节点，最好的情况是插入到数组的末尾，时间复杂度是O(1)。而最差的情况是插入到数组头，这样就需要移动数组中的每个节点，时间复杂度是O(n)。在实际的程序开发中，我们应该尽量创造前一种，而避免后一种情况。例如，当有多个数据需要插入数组时，按照有序原则（与数组排列顺序对应）进行插入，可以使直接加入队尾的概率最大化。

如果向一个满数组插入节点，一般的做法是创建一个容量更大的数组。然后将原数组中的每个节点拷贝过来，同时插入新节点。最后删除原数组，这种方法的时间复杂度为 O(n)。而且在删除原数组之前，两个数组必须并存一段时间，空间开销较大。如果这时碰巧发现程序已耗完所有内存，则必须移动部分数据到外部存储空间，当原数组被删除后再将其读回到新数组，导致巨大的时间开销。因此在实际开发中，如果可以预见向满数组中插入节点的数量，我们可以为数组额外开辟一块空缓存以适应这种情况（这种方法还可以避免某些越界错误以增加程序的健壮性，当然我们更提倡在算法中排除越界的可能）。

2. 链表

在链表中插入一个节点非常方便，对于单向链表，只需要修改插入位置之前节点的 pNext 指针使其指向本节点，然后将本节点的 pNext 指针指向下一个节点即可（需要格外注意在链表头和链表尾插入节点的情况）；对于双向链表和跳转链表，还需要修改其他相关节点的指针。不过，这种插入操作与链表的长度无关，时间复杂度为 O(1)。当然，如何定位链表中的插入节点需要花费一些时间，我们会在 3.1.5 节介绍。

3. 哈希表

在哈希表中插入一个节点需要完成两步操作，即定位桶并向链表插入节点，示例如下：

```
template<class E,class Key>
bool HashTable::Insert(const E& data)
{
     Key k = data; //从 data 提取关键字
     PtrLinkNode pNew = new LinkNode<E,Key>(data,k);//创建一个新节点
     int idx = HashFun(k);//定位桶
     PtrLinkNode p = hash_array[idx];
     if( NULL == p )
     {
         //如果是空桶，直接插入
         hash_array[idx] = pNew;
         return true;
     }
      //否则在链表头插入节点
      hash_array[idx] = pNew;
      pNew->SetNextNode(p);
      p->SetPrevNode(pNew);
      return true;
}
```

这个插入操作的时间复杂度是 $O(1)$。如果桶的链表是有序的，还需要花一些时间定位链表中的合适位置。如果链表长度为 M，那么插入操作的时间复杂度为 $O(M)$。

4. 二叉树

二叉树的结构直接影响插入操作的效率，对于一棵如图 3-4 所示的平衡二叉查找树，插入节点的时间复杂度为 $O(\mathrm{Log}_2 n)$。

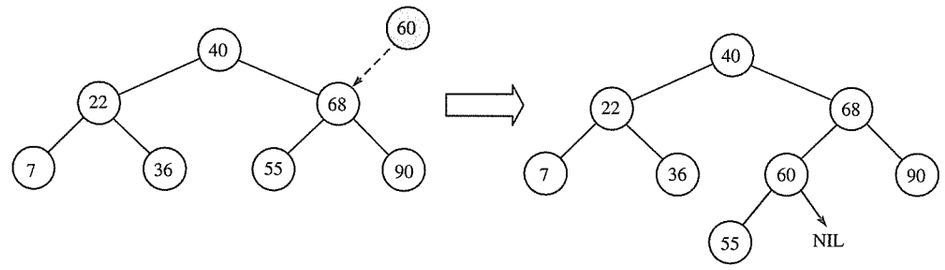


图 3-4　平衡二叉查找树的插入操作

而对于不平衡的二叉树，插入节点的时间复杂度则较高。我们举一个极端的例子，图 3-5 表示一棵完全非平衡树，所有节点的 LeftChild 指针皆为空。这时二叉树已经退化为链表，插入一个新节点的时间复杂度为 $O(n)$。

不难推断，当节点数量很多时，对平衡二叉树中进行插入操作的效率要远高于非平衡二叉树。在实际开发中，我们应该尽量避免非平衡二叉树的出现，或是将非平衡二叉树变换为平衡二叉树。一个比较简单的做法如下。

（1）中序遍历非平衡二叉树，在一个数组中保存所有节点的指针（前提是内存足够）。

（2）由于数组中所有元素都是有序排列的，所以可以使用折半查找法遍历数组，自上而下地逐层构建平衡二叉树。

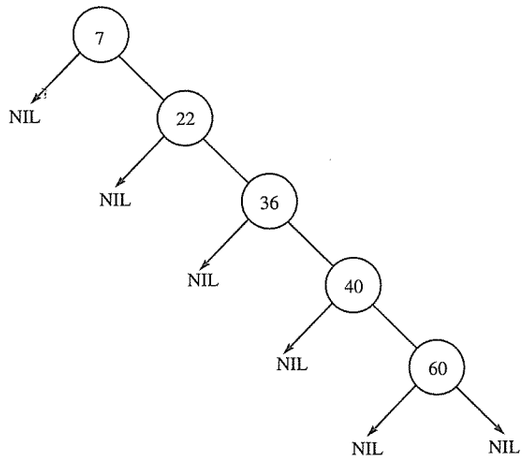


图 3-5　完全非平衡树

3.1.3　删除

1．数组

从数组中删除节点与插入操作类似，如果要求数组中没有“空洞”，则程序需要在节点删除后将其后面的所有节点向前移。在最差情况下（删除数组首节点），时间复杂度为 $O(n)$；而在最好情况下（删除数组末节点），时间复杂度为 $O(1)$。

值得一提的是，在某些场合（如动态数组，将在 3.2 节介绍），当做完删除操作后如果数组中有大量闲置空间，则需要缩小数组。即创建一个容量较小的新数组，将原数组中所有节点复制过来，再将原数组删除。这将导致较大的空间与时间开销，因此我们必须谨慎的设置数组大小，既要尽量避免内存空间的浪费，也要减少“增大（缩小）”数组的操作。

一个比较好的办法是：每当需要删除数组中的某个节点时，并不将其真正删除，而是在这个节点中设置一个标志位“bDelete”，将其设为 true，同时禁止其他程序使用该节点。待数组中需要删除的节点数达到一定阈值时，再将它们统一删除。这样避免了多次移动节

点的操作，从而降低了时间复杂度。

2．链表

在链表中删除节点几乎与插入节点的操作完全相同，删除操作的时间复杂度是 $O(1)$。

3．哈希表

从哈希表中删除一个节点与其插入操作也基本类似：首先通过哈希函数和链表遍历（如果桶由链表实现）找到待删节点，然后删除节点并重新设置前向和后向指针，如果被删节点是这个桶的首节点，则将桶的头指针指向其后续节点。下面是一个程序实例：

```
template<class E,class Key>
bool HashTable::Delete(const Key& k)
{
     PtrLinkNode p = SearchNode(k);//找到关键值匹配的节点
     if( NULL==p )
         return false;
     //修改前向节点和后向节点的指针
     PtrLinkNode pPrev = p->Prev();
     if(pPrev)
     {
         PtrLinkNode pNext = p->Next();
         if(pNext)
            pNext->SetPrevNode(pPrev);
         pPrev->SetNextNode(pNext);
     }
     else
     {
         //如果前向节点为空，说明当前节点是链表的首节点，
         //需要修改哈希数组中该项的指针，使其指向后向节点
         int idx = HashFun(k);
         hash_array[idx] = p->Next();
         if( p->Next()!=NULL )
             p->Next()->SetPrevNode(NULL);
     }
     delete p;
     return true;
}
```

4．二叉树

从二叉树中删除一个节点需要分不同情况讨论。

（1）如果是叶子节点，那么将其直接删除。

（2）如果该节点仅有一个子节点，则将子节点替换该节点。

（3）该节点的左右子节点都在。这是一种比较棘手的情况：由于每个子结点都可能有自己的子树，我们需要找到子树中合适的节点，将其树立为新的根节点，并整合这两棵子树，然后将其重新加入到原二叉树。

我们仍以二叉查找树为例，详细分析第 3 种情况。图 3-6 所示是一棵平衡的二叉查找树，它的节点“40”被删除，留下了两棵子树。

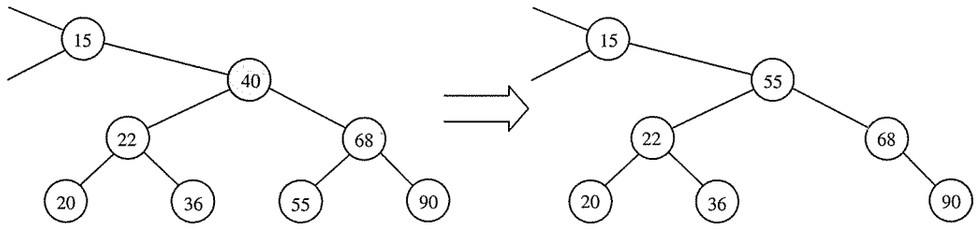


图 3-6　从二叉查找树中删除节点

我们需要将这两棵子树整合成一棵平衡的二叉查找树，然后将其加入原树，步骤如下。

（1）从被删除节点的右子树开始中序遍历，找到第 1 个节点后停止，这个节点便是新子树的根节点。

（2）由于新节点不可能同时拥有两棵子树，我们可以按照上文所述的（1）或（2）方法将其从原位置移除，然后将新节点插入到被删除节点的位置。

这样一个节点的删除操作便完成了，但是删除操作有可能降低新的二叉查找树的平衡性，还需要对树进行一些额外的修改以保持平衡。使用“红/黑”树能有效地解决此问题，本书限于篇幅，不再对其进行介绍，有兴趣的读者可以参考与数据结构的相关书籍。

3.1.4　排序

1．数组

数组的排序方法在很多数据结构的相关书籍中都有介绍，这里不再赘述。

2．链表

虽然链表在插入及删除操作上性能优越，但是排序的复杂度却很高，尤其是单向链表。因为在链表中访问某个节点需要依赖链表中的其他节点，而不像数组可以根据下标直接定位到任意一项。因此，大多数内部排序算法都不适用于链表。

当然，我们可以为链表提供一些特殊的功能使其支持各种排序算法。例如，重载下标操作符[]，赋值操作符=，比较操作符<、>、<=和>=；或为链表对象提供一些函数，如定位函数 GetElement(int idx)，以及节点交换函数 Swap(int first, int second)等。但即使这样，仍无法改变定位操作的时间复杂度为 O(n)的事实，排序效率仍然非常低下。

在实际开发中，可以采用“数组链表”的方法。当需要排序时，构造一个数组，存放链表中每个节点的指针。在排序过程中通过这个数组定位每个节点，并实现节点的交换。下面是其一个示例：

```
template<class E>
class LinkChain
{
        typedef   LinkNode<E>* PtrNode;
        PtrNode*  link_array;//指针数组
        int       size;      //链表大小
        int       count;     //链表节点个数
public:
        LinkChain(int sz);
        ~LinkChain();
        bool Delete(int idx);//删除指定下标的节点
        bool Insert(int idx,const E& element);//在指定下标处插入节点
        bool Swap(int first, int second);//交换两个节点
        E& operator[](int idx)const;//重载下标操作符
};
//构造函数以及析构函数
template<class E>
```

```
LinkChain::LinkChain(int sz):count(0)
{
    size = sz;
    //初始化指针数组，将每个节点赋初值为空
    link_array = new PtrNode[sz];
    for(int i=0;i<size;i++)
        link_array[i] = NULL;
}
template<class E>
LinkChain::~LinkChain()
{
    PtrNode pDel = link_array[0];//逐个删除链表节点
    while(pDel)
    {
        PtrNode pNext = pDel->Next();
        delete  pDel;
        pDel =  pNext;
    }
    delete []link_array;//删除指针数组
}
//插入和删除操作
template<class E>
bool LinkChain::Delete(int idx)
{
    if(idx<0||idx>=count)//判断是否越界
        return false;
    PtrNode pDel = link_array[idx];
    if(pDel!=NULL)
    {
        if( idx>0 )
        {
            //如果不是首节点，将前向节点的 next 指针指向后向节点
            PtrNode pPrev = link_array[idx-1];
            if(NULL==pPrev)//异常处理
              return false;
            pPrev->Next() = pDel->Next();
        }
        delete pDel; //删除节点
    }
    for(int i=idx;i<count-1;i++)//调整指针数组
        link_array[i] = link_array[i+1];
```



```
    count--;
    return true;
}
template<class E>
bool LinkChain::Insert(int idx,const E& element)
{
    //判断是否存在越界，或是错误的插入位置
    if(idx<0||idx>=count||(count+1)>size||!link_array[idx-1])
        return false;
    PtrNode pNew = new LinkNode<E>(element);
    if(idx>0)
    {
        //在非首节点位置插入
        PtrNode pInst = link_array[idx-1];//找到前一个节点
        if( NULL == pInst )//异常处理
        {
            delete pNew;
            return false;
        }
        pNew->Next()=pInst->Next();//更新指针
        pInst->Next() = pNew;
    }
    else//如果在首节点处插入，则将新节点的 next 指针直接指向原链表头
        pNew->Next() = link_array[idx];
    link_array[idx] = pNew;
    count++;
    return true;
}
//节点交换
template<class E>
bool LinkChain::Swap(int first, int second)
{
    if(first<0||first>=count||second<0||second>=count||first==second)
        return false;
    //如果有一个节点为空，交换失败
    if(NULL==link_array[first]||NULL==link_array[second])
        return false;
    //默认 second 是下标较大的节点，当 second＝0 时，执行交换律
    if(0==second)
        return Swap(second,first);
    PtrNode pPrev1,pPrev2,pTemp;
```

```
//更新前向节点的指针
pPrev2 = link_array[second-1];
if(NULL==pPrev2)//异常处理
   return false;
pPrev2->Next() = link_array[first];
if(first!=0)
{
   pPrev1 = link_array[first-1];
   if(NULL==pPrev1)//异常处理
      return false;
   pPrev1->Next() = link_array[second];
 }
 //更新后向节点的指针
 pTemp = link_array[first]->Next();
 link_array[first]->Next() = link_array[second]->Next();
 link_array[second]->Next() = pTemp;
//交换节点本身
 pTemp = link_array[second];
 link_array[second] = link_array[first];
 link_array[first] = pTemp;
 return true;
}
//重载下标操作符
template<class E>
E& LinkChain::operator[](int idx)const
{
      if(idx<0||idx>=count)
         throw OUT_OF_BOUND;
      if(NULL==link_array[idx])
         throw INVALID_ELEMENT;
      return link_array[idx]->GetData();
}
```

这种链表数组为直接访问链表的节点提供了便利，但是牺牲了一些空间以存放指针数组。因此，如果在程序中希望得到一个有序的链表，最好的方法还是在构建链表时将每个节点插入到合适的位置。或采用归并算法，将多个有序链表合并起来。

3. 哈希表

由于采用哈希函数访问每个桶，因此在哈希表中对哈希数组进行排序毫无意义。但是

具体节点的定位需要通过查询每个桶的链表完成（如本书中的例子，当然桶也可以是一棵二叉树或二级哈希表），将这些链表排序将提高节点的查询效率。链表的排序如前所述，这里不再赘述。

4. 二叉树

对于二叉查找树来说，其本身就是有序的，采用中序遍历的方法能够得到它有序的节点输出。正如 3.1.2 节和 3.1.3 节所述，这种有序性来源于其增删节点时所遵循的规则。而对于一棵未排序的二叉树，所有节点被随机的组织在一起，你可能得经历 $O(n)$的时间（最差的情况是需要遍历整棵树）才能找到自己需要的节点。由于无序二叉树不常使用，这里不再对其进行深入讨论。

3.1.5 查找

1. 数组

数组的最大优点就是可以通过下标任意地访问节点，而不需要借助任何指针、索引或采用遍历的方法，时间复杂度为 $O(1)$。不过在实际应用中，我们常常需要在下标未知的情况下查找节点。这时也只能遍历数组，时间复杂度将达到 $O(n)$。如果是有序数组，一个最好的查找算法是二分查找法：

```
template<class E>
E  BinSearch(E array[],const E& value,int start,int end)
{
      //判断起点和终点是否合法
      if(end-start<0)
         return INVALID_INPUT;
      //如果起点或终点有符合要求的，直接将其返回
      if(value == array[start])
         return array[start];
      if(value == array[end])
         return array[end];
      while( end>(start+1) )
      {
         //折半查找
         int temp = (start + end)/2;
         if(value == array[temp])
```

```
            return array[temp];
        if(array[temp]<value)
            start = temp;
        else
            end = temp;
    }
    throw CANNOT_FIND;
}
```

折半查找法的时间复杂度是 $O(\log_2 n)$，与二叉查找树的查询效率相同。

不过对于乱序的数组，只能通过遍历方法查找节点。在实际应用中，由于数组中的数据常常是动态变化的——某些节点可能被清除，也会不时有新节点加入。我们可以设置一个额外的标志变量保存更新节点的下标，在执行查询时，从这个下标开始遍历数组。这对于查询与更新数据相关节点（它们常常保存在更新节点的附近）的情况，执行效率将比从头开始遍历更高。这种方法在某些多媒体系统开发中经常被使用，如视音频数据帧的查找。

2. 链表

在链表中查找一个节点是一件痛苦的事情，如果是单向链表，在最差情况下需要遍历整个链表才能找到需要的节点，时间复杂度为 $O(n)$。不过，在某些情况下，链表的查询效率能够得到提高。

（1）有序链表。

如果链表有序，同时可以事先获取某些节点的数据，那么可以选择与目标数据最接近的一个节点开始查找（最优节点）。不过这种方法的效率取决于已知节点在链表中的分布。对于双向有序链表，这种方法效率更高。例如，正中节点已知，那么查询的时间复杂度为 $O(n/2)$。

（2）跳转链表。

如果事先能够根据链表中节点之间的关系建立指针关联，那么查询的效率将大大提高。

3. 哈希表

如上文所述，在哈希表中查询节点的效率与桶的链表长度有关，时间复杂度为 $O(m)$，其中 m 是桶的链表长度。查找算法的实现如下：

```
template<class E,class Key>
```



```
PtrLinkNode HashTable::SearchNode(const Key& k) const
{
    int idx = HashFun(k);//定位桶
    if( NULL==hash_array[idx] )//如果是空桶，直接返回
        return NULL;
    //遍历桶的链表，如果发现有匹配的节点，将其返回
    PtrLinkNode p = hash_array[idx];
    while(p)
    {
       if( k == p->GetKey() )
          return p;
       p = p->Next();
    }
    return NULL;//未找到，返回 NULL
}
```

4. 二叉树

在二叉查找树中查询一个节点的效率由树的形状决定，对于一棵平衡二叉树（如图 3-4 所示），查询效率为 $O(\mathrm{Log}_2 n)$。而对于一棵完全不平衡二叉树（如图 3-5 所示），查询效率则降至 $O(n)$。因此在实际开发中，开发人员都希望构建尽量平衡的二叉树以提高查询效率。然而如前所述，平衡二叉树受插入/删除操作影响很大。经常是每增/删一个节点，就要调整树的结构。因此如何调整一棵树的平衡度，需要根据实际的开发需求采取合适的策略。一种比较理想的情况是，当树的插入/删除操作很多时，不需要在每次操作后都调整二叉树的平衡度，而是在密集的查询操作之前统一调整一次。

3.2 动态数组的实现及分析

本节将对动态数组的定义，结构进行阐述，并结合实际工程中的一个例子分析其特点。

3.2.1 动态数组简介

在实际程序开发中，数组是最常用的数据结构之一。很多情况下，开发人员为了满足可扩展性（scalability）的需要，会采用多种方式实现数组。其中一个极端的例子是如果在

编译时就知道数组所有的维数，则可以静态地定义这个数组。例如：

```
void StaticArray()
{
      //数组元素个数 num 是编译期常量
      const unsigned short num = 256;
      char str[num];
      //访问数组中的每个元素
      for( unsigned short i=0; i<num; i++ )
           DoSomething(str[i]);
}
```

在定义了静态数组之后，无论程序如何使用这个数组，该数组在内存中所占空间的大小及位置是确定不变的。如果程序员定义的是一个全局数组，编译器将在静态数据区为这个数组分配空间；如果是个局部数组（例如定义在某个函数中），编译器将在栈（stack）上为这个数组分配空间。

一般地，在很多情况下，我们事先无法知道数组中究竟会有多少个元素。例如，从一个TCP 客户端的套接字（socket）接收一个包，数据区的长度只有在解析这个包之后才能获得：

```
void PacketArray(unsigned long length)
{
      //创建一个接收缓存区以存放包的数据
      //注意：如果内存分配失败（可能由于内存耗尽）
      //要捕获这个异常，以免后面发生错误操作
      try
      {
          BYTE* buffer = new BYTE[length];
      }catch(...){
          throw;
      }
      //注意：由于上面的程序采用指针分配内存，
      //我们需要捕获每一个异常，以免漏过 delete 操作
      try
      {
         for( unsigned long i=0; i<length; i++ )
             ReadAt(buffer[i]);
      }catch(...){
         delete []buffer;
         throw;
      }
      //在最后也一定要 delete
      delete []buffer;
}
```


在这种情况下，C++编译器在堆（或自由存储区，使用哪种内存区，取决于你采用new/delete，还是 malloc(relloc)/free 方式）上为数组分配内存。这样，数组的容量在程序运行时决定。与静态数组相比，动态数组能够为程序开发提供更大的灵活性。

动态数组的优点主要如下。

（1）可分配空间较大：一般来说，在 32 位 Windows 系统下，堆内存可以达到最大 4 GB 的空间。而对于栈来说，编译器对其都有限制。例如，在 MS Visual C++ 6.0 下，默认的栈空间大小是 1 MB（当然，开发人员可以设置这个值，但是为了保证程序运行效率，栈空间不宜太大）。

（2）使用灵活：开发人员可以根据实际需要，自行决定数组的大小甚至是维数。

当然，俗语说得好，“天下没有免费的午餐”。动态数组虽然为程序开发带来了更多的灵活性，但并非十全十美。

（1）空间分配效率比静态数组低：如前所述，静态数组一般由栈分配空间，而动态数组则一般由堆分配空间。栈是机器系统提供的数据结构，计算机会在底层为栈提供支持。即分配专门的寄存器存放栈的地址，压栈及出栈都有专门的指令执行，因而决定了栈的效率比较高。而堆由 C++函数库提供，其内存分配机制与栈相比要复杂得多。例如，为了分配一块内存，库函数会按照一定的算法（具体算法可以参考数据结构或操作系统的相关文献，这里不再赘述）在堆内存中搜索可用的足够大小的空间，如果发现空间不够（可能由于虚拟内存碎片太多），它将调用内核方法去增加程序数据段的存储空间。从而程序就有机会分到足够大小的内存。显然，堆的效率比栈要低得多。

（2）容易造成内存泄漏：动态数组需要开发人员手工地分配和释放内存，容易由于开发人员的疏漏而造成内存泄漏。

3.2.2　动态数组实践及分析

由于动态数组有灵活及可分配空间大的特点，所以在实际工程中得到了广泛应用。本节我们将结合一个实时视频系统的例子，对其进行详细分析。

在实时视频系统中，视频服务器承担视频数据的缓存及转发工作。一般地，服务器为每台摄像机开辟一定大小且独立的缓存区。视频帧被写入此缓存区后，服务器在某一时刻再将其读出，并向客户端转发。视频帧的保存是临时的，由于缓存区大小有限而视频数据源源不断，所以一帧在被写入后，过一段时间便会被新来的视频帧所覆盖。视频帧的缓存时间由缓存区和该帧的长度决定。

由于视频帧数据量巨大，而一台服务器通常需要支持数十台，甚至上百台摄像机，如何设计缓存结构是系统成功的重要问题。一方面，如果事先分配固定数量的内存，运行时不再增删，那么服务器只能支持一定数量的摄像机，灵活性很小；另一方面，由于视频服务器程序在启动时就占据了一大块内存，将导致系统整体性能的下降，于是我们考虑采用动态数组来实现视频缓存。

首先，我们在服务器中为每台摄像机分配一个一定大小的缓存块，由类 CamBlock 实现。如图 3-7 所示，每个 CamBlock 对象中有两个动态数组，分别是存放图像数据的_data 和存放视频帧索引信息的_frameindex。每当程序在内存中缓存（读取）一个视频帧时，对应的 CamBlock 对象将根据索引表_frameindex 找到该帧在_data 中的存放位置，然后将数据写入（读出）。_data 是一个循环队列（circular queue），一般根据 FIFO 原则进行数据读取。即如果有新帧进入队列，程序会在_data 中最近写入帧的末尾开始复制。如果超出了数组的长度，则从头覆盖。

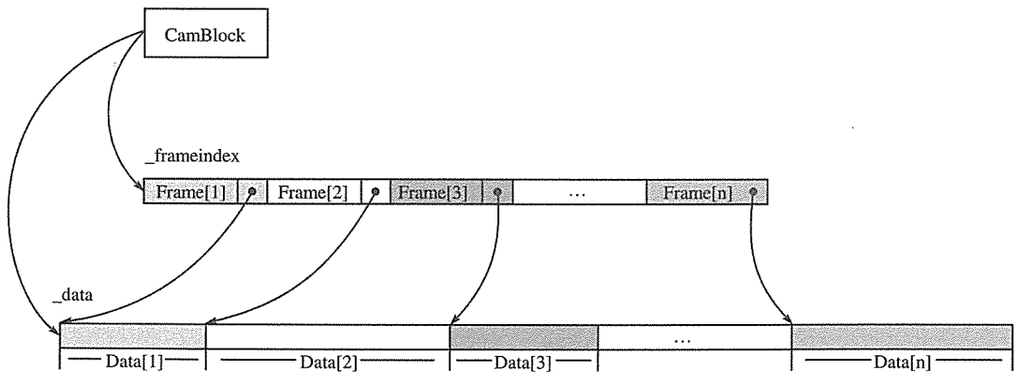


图 3-7　视频帧索引表及数据缓存区

由于篇幅关系，在这里我们不再详细描述视频帧存储与读取的过程，而是将重点放在动态数组的构建与使用上。下面是 CamBlock 的定义：


```
//视频帧的基本数据结构
typedef struct
{
    unsigned short  idcamera;          //摄像机 ID
    unsigned long  length;             //数据长度
    unsigned short  width;             //图像宽度
    unsigned short  height;            //图像高度
    unsigned char*  data;              //图像数据地址
}Frame;
//单台摄像机缓存模块定义
class CamBlock
{
    Frame*          _frameindex;       //帧索引表
    unsigned char*  _data;             //存放图像数据的缓存区
    unsigned long   _length;           //缓存区大小
    int             _idcam;            //对应摄像机的序号
    unsigned short  _numframes;        //可存放帧的数量
    unsigned long   _framepos;         //最后一帧的位置
public:
    CamBlock(int id, unsigned long len, unsigned short numframes):
    _data(NULL), _length(0), _idcam(-1), _numframes(0)
    {
        //确保缓存区的大小未超过阈值
        if(len>MAX_LENGTH||numframes>MAX_FRAMES)
            throw;
        try{
            //为帧索引表分配空间
            _frameindex = new Frame[numframes];
            //为摄像机分配一块指定大小的缓存
            _data   = new unsigned char[len];
        }catch(...){
            //如果分配失败，抛出例外
            throw;
        }
        memset( _data, 0, len ); //将缓存清 0
        _length = len;
        _idcam  = id;
        _numframes = numframes;
    };
    ~CamBlock()
    {
```

```
        //释放空间
        delete []_data;
        delete []_frameindex;
    };
    //存放一帧，根据索引表将视频数据存入缓存
    BOOL SaveFrame( const Frame* frame );
    //读取一帧，根据索引表定位到某一帧，读出并返回
    BOOL ReadFrame( Frame* frame );
}
```

现在我们已经有了每台摄像机缓存块的定义，但如何管理这些相对独立的内存块，使程序能够方便地定位到任意一台摄像机的缓存，以致任意一帧？我们还需要建立另一个索引表 CameraArray 来管理所有 CamBlock 对象。

如图 3-8 所示，CameraArray 利用一个动态数组 camera_bufs 管理所有 CamBlock 对象。这个数组并不直接存放 CamBlock 对象本身，而是存放指向每个 CamBlock 对象的指针。

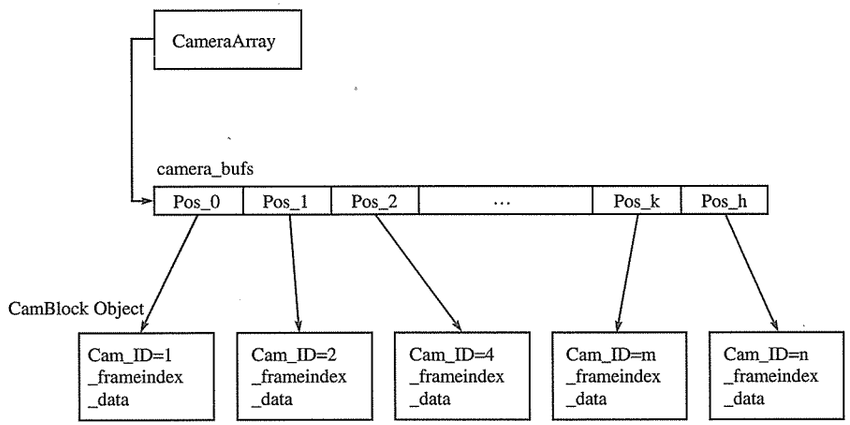


图 3-8　CameraArray 中的动态数组 camera_bufs

在初始化 CameraArray 时，构造函数根据传入参数 max 决定 camera_bufs 数组的初始容量，并为每个 CamBlock*指针分配空间。在运行过程中，若系统需要为某台摄像机开辟新的缓存区，它将利用对应的指针构造一个 CamBlock 对象：

```
//CameraArray 类的声明
class CameraArray
{
```



```
    typedef CamBlock* PtrBlock;
    PtrBlock*         camera_bufs; //摄像机视频缓存
    unsigned short    camera_num;  //当前已连接上的摄像机数目
    unsigned short     max_num;    //camera_bufs 的容量
    unsigned short     inc_num;    //camera_bufs 的增量
public:
    CameraArray(unsigned short max, unsigned short inc);
    ~CameraArray();
    //插入一个新摄像机记录
    CamBlock* InsertBlock( unsigned short idcam, unsigned long size, unsigned
                           short numframes );
    //删除一台摄像机记录
    BOOL      RemoveBlock( unsigned short idcam );
private:
    //根据摄像机 ID 返回它在数组中的位置
    unsigned short GetPosition(unsigned short idcam);
};
//CameraArray 类的构造函数：
CameraArray::CameraArray(unsigned short max, unsigned short inc):
camera_bufs(NULL), max_num(0), inc_num(0), camera_num(0)
{
    //如果输入参数越界，抛出一个例外
    if(max>MAX_CAMERAS||inc>MAX_INCREMENT)
        throw;
    //初始化 camera_bufs 数组
    try{
        camera_bufs = new PtrBlock[max];
    }catch(...){
        throw;
    }
    max_num = max;
    inc_num = inc;
}
//CameraArray 类的析构函数：
CameraArray::~CameraArray()
{
    for(unsigned short i=0;i<camera_num;i++)
        delete camera_bufs[i];
    delete []camera_bufs;
}
```

在实际的视频系统中，我们常常给每个摄像机安排一个整数类型的 ID。在 CameraArray 中，程序按照 ID 递增的顺序排列每个摄像机的 CamBlock 对象以方便查询。当一个新的摄像机接入系统时，程序会根据它的 ID 在 CameraArray 中找到一个合适的位置，然后利用这个指针创建新的 CamBlock 对象（通过 InsertBlock 函数）。同样的，如果某个摄像机断开连接，程序也会根据它的 ID 找到对应的缓存块并将其删除(通过 RemoveBlock 函数)。这两个函数定义如下：

```
CamBlock* CameraArray::InsertBlock( unsigned short idcam, unsigned long size,
unsigned short numframes )
{
    //在数组中找到合适的插入位置
    unsigned short pos = GetPosition(idcam);
    //已达到数组边界，需要扩大数组才能满足要求
    if(camera_num==max_num)
    {
        //定义一个新的数组指针，并指定其维数
        PtrBlock* newbufs = NULL;
        try{
            newbufs = new PtrBlock[ max_num+inc_num ];
        }catch(...) {
            throw;
        }
        //将原数组中的数据拷贝到新数组
        //注意：数组中的每个元素只是指向不同摄像机缓存块的指针，
        memcpy( newbufs, camera_bufs, max_num*sizeof(PtrBlock) );
        //释放原数组的内存
        delete []camera_bufs;
        max_num+=inc_num;
        //更新数组指针
        camera_bufs = newbufs;
    }
    if( pos!=camera_num )
    {
        //在数组中插入一个块，由于是按 camera id 的递增顺序分配块的内存
        //因此需要将该位置以后的所有块指针向后移 1
        memmove( camera_bufs + pos + 1, camera_bufs + pos ,
(camera_num-pos)*sizeof (PtrBlock) );
    }
    ++camera_num;
```



```
    CamBlock* newblock = new CamBlock( idcam, size, numframes );
    camera_bufs[pos] = pNode;
    return camera_bufs[pos];
}
```

值得一提的是，如果接入系统的摄像机数量超过了最初创建 CameraArray 对象时的设计容量，则考虑到系统的可扩展性，只要硬件条件允许，我们需要增加 camera_bufs 的长度。具体做法是定义一个新的数组 newbufs，其长度是 max_num+inc_num（inc_num 是事先设定的一个增量）。将原 camer_bufs 的内容拷贝到新数组（注意：这里拷贝的是每台摄像机缓存块的指针而不是对象本身），然后将 camera_bufs 的指针指向新数组。

删除操作几乎是插入的逆过程，即在 camera_bufs 中找到与摄像机 ID 对应的 CamBlock 对象，将其删除：

```
BOOL CameraArray::RemoveBlock( unsigned short idcam )
{
    //在数组中找到合适的插入位置
    unsigned short pos = GetPosition(idcam);
    if(camera_num<1)
        return FALSE;
    camera_num--;
    //删除 pos 对应的内存块
    PtrBlock delblock = camera_bufs[pos];
    delete delblock;
    if( pos!=camera_num )
    {
        //将 pos 之后的元素向前移一位
        memmove( camera_bufs + pos, camera_bufs + pos + 1,
(camera_num-pos)*sizeof (PtrBlock) );
    }
    //如果数组中有过多空闲的元素空间，将其释放
    if( max_num - camera_num > inc_num )
    {
        //重新计算数组的长度
        unsigned short len = ( (camera_num/inc_num)+1 )*inc_num;
        //定义新的数组指针，并指定其维数
        PtrBlock* newbufs = NULL;
        try{
            newbufs = new PtrBlock[len];
```

```
            }catch(...){
                throw;
            }
            //将原数组中的元素拷贝到新数组中
            memcpy( newbufs, camera_bufs, camera_num*sizeof(PtrBlock) );
            delete[] camera_bufs;
            camera_bufs = newbufs;
            max_num = len;
        }
        return TRUE;
    }
```

与 InsertBlock 函数对应，如果在删除一台摄像机记录时，发现数组中有过多的空闲空间，我们需要释放这些空间。做法与 InsertBlock 相似，为一个新的数组开辟缓存区，从原数组拷贝数据，然后将原数组删除。

至此，视频服务器的基本内存管理机制已经建立。可以看到，无论是视频缓存块的分配，还是摄像机索引的创建，都是基于动态数组实现的。动态数组灵活并可扩展的特性为这种内存需求变化较大的程序提供了很好的借鉴意义。

3.3 本章小结

本章首先根据不同的实用操作，对各种常用数据结构的性能进行了分析，然后结合一个程序实例介绍了动态数组的原理及其应用。总的来说，每种数据结构各有特点。在不同的环境下的性能表现也各不相同，没有一种完美的数据结构能够在各种操作下皆表现最优。在实际的程序开发中，开发人员应该根据主要操作的频率和数据的分布选取合适的数据结构和算法实现，以达到程序整体性能最优。

第 2 篇　内存使用优化

编写高性能的应用程序时，内存是一个无法绕开的话题。不能正确并高效地管理所写应用程序使用的内存，开发高性能的程序的可能性可以说是微乎其微。正是看到这一点，本书特地通过 3 章来详细讲述内存的底层管理机制、语言层的使用方法，以及一些在实际应用中被证明行之有效的高效使用内存的方法。本篇的 3 章分列如下。

第 4 章：操作系统的内存管理。
第 5 章：动态内存管理。
第 6 章：内存池。

第 4 章 操作系统的内存管理

4

长期以来，在计算机系统中，内存都是一种紧缺和宝贵的资源，应用程序必须在载入内存后才能执行。以前，在内存空间不够大时，同时运行的应用程序的数量就会受到很大限制。甚至当某个应用程序在某个运行时所需内存超过物理内存时，该应用程序就会无法运行。现代操作系统（诸如 Windows 和 Linux）的内存管理都能解决这个问题，解决的方法就是虚拟内存的引入。

本质上虚拟内存就是要让一个程序的代码和数据在没有全部载入内存时即可运行。运行过程中，当执行到尚未载入内存的代码，或者要访问还没有载入到内存的数据时，虚拟内存管理器动态地将这部分代码或数据从硬盘载入到内存中。而且在通常情况下，虚拟内存管理器也会相应地先将内存中某些代码或者数据置换到硬盘中，为即将载入的代码或数据腾出空间。

因为内存和硬盘之间的数据传输相对代码执行来说，是非常慢的操作，因此虚拟内存管理器在保证工作正确的前提下，还必须考虑效率因素。比如，它需要优化置换算法，尽量避免就要执行的代码或访问的数据刚被置换出内存，而很久没有访问的代码或数据却一直驻留在内存中。另外它还需要将驻留在内存的各个进程的代码或数据维持在一个合理的数量上，并且根据该进程的性能表现动态调整此数量，等等，使得程序运行时将其涉及的磁盘 I/O 次数降到尽可能低，以提高程序的运行性能。

本章前一部分着重介绍 Windows 的虚拟内存管理机制，后一部分则简要介绍 Linux 的虚拟内存管理机制。

4.1 Windows 内存管理

如果从应用程序的角度来看 Windows 虚拟内存管理系统，可以扼要地归结为一句话。即 Win32 虚拟内存管理器为每一个 Win32 进程提供了进程私有且基于页的 4 GB（32 位）大小的线性虚拟地址空间，这句话可以分解如下：

（1）“进程私有”意味着每个进程都只能访问属于自己的地址空间，而无法访问其他进程的地址空间，也不用担心自己的地址空间会被其他进程看到（父子进程例外，比如调试器利用父子进程关系来访问被调试进程的地址空间，这里不详述）。需要注意的是，进程运行时用到的 dll 并没有属于自己的虚拟地址空间。而是其所属进程的虚拟地址空间，dll 的全局数据，以及通过 dll 函数申请的内存都是从调用其进程的虚拟地址空间中开辟。

（2）“基于页”是指虚拟地址空间被划分为多个称为“页”的单元，页的大小由底层处理器决定，x86 中页的大小为 4 KB。页是 Win32 虚拟内存管理器处理的最小单元，相应的物理内存也被划分为多个页。虚拟内存地址空间的申请和释放，以及内存和磁盘的数据传输或置换都是以页为最小单位进行的。

（3）“4 GB 大小”意味着进程中的地址取值范围可以从 0x00000000 到 0xFFFFFFFF。Win32 将低区的 2 GB 留给进程使用，高区的 2 GB 则留给系统使用。

Win32 中用来辅助实现虚拟内存的硬盘文件称为“调页文件”，可以有 16 个，调页文件用来存放被虚拟内存管理器置换出内存的数据。当这些数据再次被进程访问时，虚拟内存管理器会先将它们从调页文件中置换进内存，这样进程可以正确访问这些数据。用户可以自己配置调页文件。出于空间利用效率和性能的考虑，程序代码（包括 exe 和 dll 文件）不会被修改，所以当它们所在的页被置换出内存时，并不会被写进调页文件中，而是直接抛弃。当再次被需要时，虚拟内存管理器直接从存放它们的 exe 或 dll 文件中找到它们并调入内存。另外对 exe 和 dll 文件中包含的只读数据的处理与此类似，也不会为它们在调页文件中开辟空间。

当进程执行某段代码或者访问某些数据，而这些代码或者数据还没有在内存时，这种情形称为“缺页错误”。缺页错误的原因有很多种，最常见的一种就是已经提到的，即这些代码和数据被虚拟内存管理器置换出了内存，这时虚拟内存管理器在这段代码执行或者这些数据被访问前将它们调入内存。这个操作对开发人员来说是透明的，因此大大简化了开发人员的负担。但是调页错误涉及磁盘 I/O，大量的调页错误会大大降低程序的总体性能。因此需要了解缺页错误的主要原因，以及规避它们的方法。

4.1.1 使用虚拟内存

Win32 中分配内存分为两个步骤：“预留”和“提交”。因此在进程虚拟地址空间中的页有 3 种状态：自由（free）、预留（reserved）和提交（committed）。

（1）自由表示此页尚未被分配，可以用来满足新的内存分配请求。

（2）预留指从虚拟地址空间中划出一块区域（region，页的整数倍数大小），划出之后这个区域中的页不能用来满足新的内存分配请求，而是用来供要求“预留”此段区域的代码以后使用。预留时并没有分配物理存储，只是增加了一个描述进程虚拟地址空间使用状态的数据结构（VAD，虚拟地址描述符），用来记录这段区域已被预留。“预留”操作相对较快，因为没有真正分配物理存储。也正因为没有分配真正的物理存储，所以预留的空间并不能够直接访问，对预留页的访问会引起“内存访问违例”（内存访问违例会导致整个进程立刻退出，而不仅仅是中止引起该违例的线程）。

（3）提交，若想得到真正的物理存储，必须对预留的内存进行提交。提交会从调页文件中开辟空间，并修改 VAD 中的相应项。注意，提交时也并没有立刻从物理内存中分配空间，而只是从磁盘的调页文件中开辟空间。这个空间用做以后置换的备份空间，直到有代码第一次访问这段提交内存中的某些数据时，系统发现并没有真正的物理内存，抛出缺页错误。虚拟内存管理器处理此缺页错误，直到这时才会真正分配物理内存，提交也可以在预留的同时一起进行。需要注意的是，提交操作会从调页文件中开辟磁盘空间，所以比预留操作的时间长。

这也是 Win32 虚拟内存管理中的 demand-paging 策略的一个体现，即不到真正访问时，不会为某虚拟地址分配真正的物理内存。这种策略一是出于性能考虑，将工作分段完成，提高总体性能；二是出于空间效率考虑，不到真正访问时，Win32 总是假定进程不会访问大多数的数据，因而也不必为它们开辟存储空间或将其置换进物理内存，这样可以提高存储空间（磁盘和物理内存）的使用效率。

设想某些程序对内存有很大的需求，但又不是立即需要所有这些内存，那么一次就从物理存储中开辟空间满足这些还只是“潜在”的需求，从执行性能和存储空间效率来说，都是一种浪费。因为只是“潜在”需求，极有可能这些分配的内存中很大一部分最后都没有真正被用到。如果在申请的时候就一次性为它们分配全部物理存储，无疑会极大地降低空间的利用效率。

另一方面，如果完全不用预留及提交机制，只是随需分配内存来满足每次的请求，那么对一个会在不同时间点频繁请求内存的代码来说，因为在它请求内存的不同时间点的间隙极有可能会有其他代码请求内存。这样这段在不同时间点频繁请求内存的代码请求得到的内存因为虚拟地址不连续，无法很好地利用空间 locality 特性，对其整体进行访问（比如遍历操作）时就会增加缺页错误的数量，从而降低程序的性能。

预留和提交在 Win32 中都使用 VirtualAlloc 函数完成，预留传入 MEM_RESERVE 参数，提交传入 MEM_COMMIT 参数。释放虚拟内存使用 VirtualFree 函数，此函数根据不同的传入参数，与 VirtualAlloc 相对应，可以释放与虚拟地址区域相对应的物理存储，但该虚拟地址区域还可处于预留状态，也可以连同虚拟地址区域一起释放，该段区域恢复为自由状态。

线程栈和进程堆的实现都利用了这种预留和提交两步机制，下面仅以线程栈为例来说明 Win32 系统是如何使用这种预留和提交两步机制的。

创建线程栈时，只是一个预留的虚拟地址区域，默认是 1 MB（此大小可在 CreateThread 或在链接时通过链接选项修改），初始时只有前两页是提交的。当线程栈因为函数的嵌套调用需要更多的提交页时，虚拟内存管理器会动态地提交该虚拟地址区域中的后续页以满足其需求，直到到达 1 MB 的上限。当到达此预留区域大小的上限（默认 1 MB）时，虚拟内存管理器不会增加预留区域大小，而是在提交最后一页时抛出一个栈溢出异常，抛出栈溢出异常时该栈还有一页空间可用，程序仍可正常运行。而当程序继续使用栈空间，用完最后一页后，还继续需要存储空间，这时就超过了上限，会直接导致进程退出。

所以为防止线程栈溢出导致整个程序退出，应该注意尽量控制栈的使用大小。比如减少函数的嵌套层数，减少递归函数的使用，尽量不要在函数中使用太大的局部变量（大的对象可以从堆中开辟空间存放，因为堆会动态扩大，而线程栈的可用内存区域在线程创建时就已固定，之后在整个线程生命期间无法扩展）。

另外为了防止因为一个线程栈的溢出导致整个进程退出，可以对可能会产生线程栈溢出的线程体函数加异常处理，捕获在提交最后一页时抛出的溢出异常，并做出相应处理。

4.1.2 访问虚拟内存时的处理流程

对某虚拟内存区域进行了预留并提交之后，就可以对该区域中的数据进行访问了，下图描述了当程序对某段内存访问时的处理流程：

如图 4-1 所示，当该数据已在物理内存中时，虚拟内存管理器只需将指向该数据的虚拟地址映射为物理指针，即可访问到物理内存中的真正数据。这一步不会涉及磁盘 I/O，速度相对较快。

当第一次访问一段刚刚提交的内存中的数据时，因为并没有真正的物理内存分配给它。或者该数据以前已被访问过，但是被虚拟内存管理器置换出了内存。这两种情形都会引发缺页错误，虚拟内存管理器此时会处理这一缺页错误，它先检测此数据是否在调页文件中已有备份空间（exe 和 dll 的代码页和只读数据页情形与此类似，但是其备份空间不在调页文件，而是包含它们的 exe 或 dll 文件）。如果是这两种情况，表明访问的数据在磁盘中有备份，接下来虚拟内存管理器就需要在物理内存中找到合适的页，并将存放在磁盘的备份数据置换进物理内存。

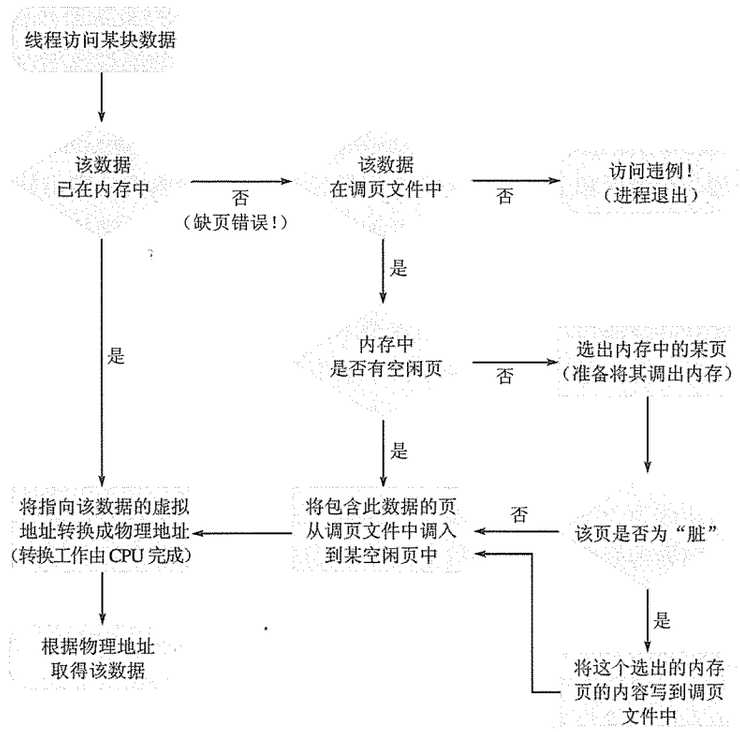


图 4-1　访问虚拟内存的处理流程

虚拟内存管理器首先查询当前物理内存中是否有空闲页，虚拟内存管理器维护一个称为“页帧数据库”（page-frame database）的数据结构，此数据结构是操作系统全局的，当 Windows 启动时被初始化，用来跟踪和记录物理内存中每一个页的状态，它会用一个链表将所有空闲页连接起来，当需要空闲页时，直接查找此空闲页链表，如果有，直接使用某个空闲页；否则根据调页算法首先选出某个页。需要指出的是，虚拟内存管理器调页时并不是只调入一个页，为了利用局部特性，它在调入包含所需数据的页的同时，会将其附近的几个页一起调入内存。这里为了简单和清楚起见，假定只调入目标页。但应该意识到 Win32 调页时的这个特性，因为可以利用它来提高程序效率。这个页将会用来存放即将从磁盘置换进来的页的内容。选出某个内存页后，接着检查此页状态，如果此页自上次调进内存以来尚未被修改过，则直接使用此页（代码页和只读页也可以直接使用）；反之，如果此页已被修改过（“脏”），则需要先将此页的内容“写”到调页文件中与此页相对应的备份

页中，并随即将此页标为空闲页。

现在，有了一个空闲页用来存放即将要访问的数据。此时，虚拟内存管理器会再次检测，此数据是否是刚被申请的内存且是第一次被访问。如果是，则直接将此空闲页清 0 使用即可（不必从磁盘中将其备份页的内容读进，因为该备份页中的内容无意义）；如果不是，则需要将调页文件中该页的备份页读到此空闲页中，并随即将此页的状态从空闲页改为活动页。

此时，此数据已在物理内存页中，通过虚拟地址映射到物理地址，即就可访问此数据了。

上述为访问成功时的情形，但情形并非总是如此。比如当用户定义了一个数组，而此数组刚好在其所在页的下边界，且此页的下一页刚好是自由或者预留的（不是提交的，即没有真正的物理存储）。当程序不小心向下越界访问此数组，则首先引发缺页错误。随即虚拟内存管理器在处理缺页错误时检测到它也不在调页文件中，这就是所谓的“访问违例”（access violation）。访问违例意味着要访问的地址所在的虚拟内存页还没有被提交，即没有实际的物理存储与之对应，访问违例会直接导致整个进程退出（即 crash）。

可以看到，指针越界访问的后果根据运行时实际情况而有所不同。如上所述，当数组并非处于其所在页的边界，越界后还在同一页中，这时只会“误访问”（ 误读或误写，其中误读只会影响到正在执行的代码；误写则会影响到其他处代码的执行）该页中其他数据，而不会导致整个进程的 crash。即使在该数组真的处于其所在页的边界，且越界后指针值落在了其相邻页。但如果此相邻页碰巧也为一个提交页，此时仍然只是“误访问”，也不会导致进程的 crash。这也意味着，同一个应用程序的代码中存在着指针越界访问错误，运行时有时 crash，但有时则不会。

Microsoft 提供了一个监测指针越界访问的工具 pageheap，它的原理就是强制使每次分配的内存都位于页的边界，同时强制该页的相邻页为自由页（即不分配其相邻页给程序使用）。这样每次越界访问都会立即引起 access violation，导致程序 crash。从而使得指针越界访问错误在开发期间一定会被暴露出来，而不会发生某个指针越界访问错误一直隐藏到 Release 版本，直到最终用户使用时才被发现的情形。

4.1.3 虚拟地址到物理地址的映射

如上所述，在确保访问的数据已在物理内存中后，还需要先将虚拟地址转换为物理地址，即“地址映射”，才能够真正访问此数据。本节讲述 Win32 中虚拟内存管理器如何将虚拟地址映射为物理地址。

Win32通过一个两层表结构来实现地址映射，因为4 GB虚拟地址空间为每个进程私有，相应地，每个进程都维护一套自己的层次表结构用来实现其地址映射。第一层表称为“页目录”（page directory），实际上就是一个内存页（4 KB = 4 096 byte）。这一页以四个字节为单元分为 1 024 项，每一项称为一个“页目录项”（Page Directory Entry，PDE）；第二层表称为“页表”（page table），共有 1 024 个页表。页目录中每一个页目录项 PDE 对应这一层中的某一个页表，每一个页表也占了一个内存页。这一页中的 4 KB，即 4 096 个字节也像页目录那样被分成 1 024 项，每项 4 个字节，页表的每一项则称为“页表项”（Page Table Entry，PTE）。每一个页表项 PTE 都指向物理内存中的某一个页帧，如图 4-2 所示。

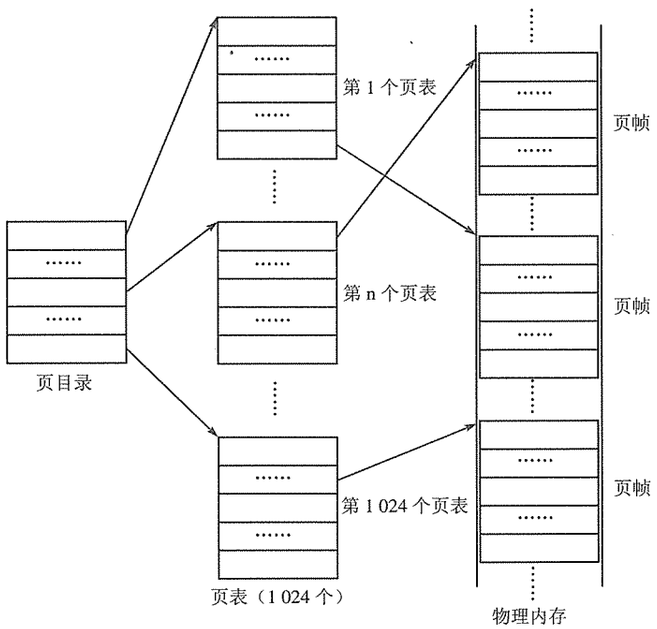


图 4-2 页表

已经知道，Win32 提供了 4 GB（32 位）大小的虚拟地址空间。因此每个虚拟地址都是一个 32 位的整数值，这 32 位由 3 个部分组成，如图 4-3 所示。

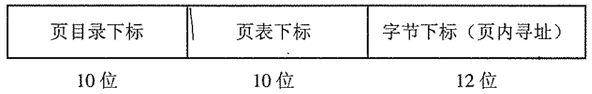


图 4-3 虚拟地址空间

这三个部分中的第一部分，即前 10 位为页目录下标，用其可以定位在页目录的 1 024 项中的某一项。根据定位到的那一项的项值，可以找到第 2 层页表中的某一个页表。虚拟地址的第二部分，即中间的 10 位为页表下标，可用来定位刚刚找到的页表的 1 024 项中的某一项。此项值可以找到物理内存中的某一个页，此页包含此虚拟地址所代表的数据。最后用虚拟地址的第三部分，即最后 12 位可用来定位此物理页中的特定的字节位置，12 位刚好可以定位一个页中的任意位置的字节。

举一个具体的例子，假设在程序中访问一个指针（Win32 中的“指针”意味虚拟地址），此指针值为 0x2A8E317F，图 4-4 所示为虚拟地址到物理地址的映射过程。

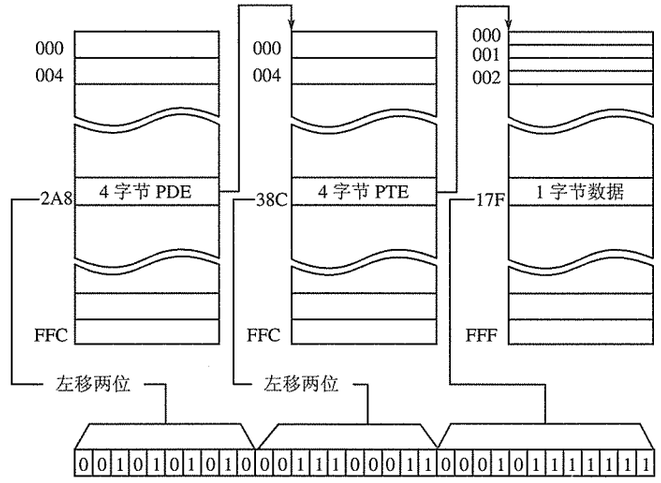


图 4-4 虚拟地址到物理地址的映射过程

0x2A8E317F 的二进制写法为 0010101010,0011100011,000101111111，为了方便起见，将这 32 位分成 10 位、10 位和 12 位。第一个 10 位 00101010 用来定位页目录中的页目录项，因为页目录项为四个字节，定位前将此 10 位左移两位，即 0010101000（0x2A8）。再

用此值作为下标找到对应的页目录项，此页目录项指向一个页表。同样方法再用第二个 10 位 0011100011 定位此页表中的页表项。此页表项指向真正的物理内存，然后用最后 12 位 000101111111 定位页内的数据（此时这 12 位不用再左移，因为物理页内定位时，需要能定位到每一个字节。而不像页目录和页表中，只需要定位每 4 个字节的第 1 个字节），即为此指针指向的数据。

上面假设的是此数据已在物理内存中，其实，“判断访问的数据是否在内存中”这一步骤，也是在这个地址映射过程中完成的，Win32 总是假使数据已在物理内存中，并进行地址映射。页表项中有一位用来标识包含此数据的页是否在物理内存页中，当取得页表项时，检测此位，如果在，就是本节描述的过程，如果不在，则抛出缺页错误，此时此页表项中包含了此数据是否在调页文件中，如果不在，则为访问违例，如果在，此页表项可查出了此数据页在哪个调页文件中，以及此数据页在该调页文件中的起始位置，然后根据这些信息将此数据页从磁盘中调入物理内存中，再继续进行地址映射过程。

已经说过，为了实现虚拟地址空间各进程私有，每个进程都拥有自己的页目录和页表结构，对不同进程而言，页目录中的页目录项值（PDE），以及页表中的页表项值（PTE）都是不同的，因此相同的指针（虚拟地址）被不同的进程映射到的物理地址也是不同的。这也意味着，在不同进程间传递指针是没有意义的。

4.1.4 虚拟内存空间使用状态记录

当通过 VirtualAlloc 申请一块虚拟内存时，虚拟内存管理器是如何知道哪些内存块是自由的，可以用来满足此次内存请求呢？即 Win32 虚拟内存如何维护和记录每一个进程的 4 GB 虚拟内存地址空间的使用状态，如各个区域的状态、大小及起始地址呢？

上一节中，读者也许会认为可以通过遍历页目录和页表中的项值来收集虚拟内存空间的使用状态，但这样做首先有效率问题，因为每次申请内存都需要做一次搜索。但这个方法不仅仅是因为效率有问题，而且还是行不通的，对预留的页来说，虚拟内存管理器并没有为之分配物理存储。所以也就不会为其填写页表项，这时遍历页表无法分辨某块虚拟内存是自由还是预留的。另外即使对提交页来说，遍历页表也无法得到完整的信息，正如 4.1.1 节中提到的 Win32 在虚拟内存管理时用到的主要策略 demand-paging，即 Win32 虚拟内存管理器在程序没有实际访问某块内存前，总是假定这块内存不会被访问到，因此不会为这

块内存做过多处理，包括不会为其分配真正的物理内存空间，甚至页表，即进程中用来完成虚拟地址到物理地址映射的页表的存储空间也是随需分配的。

Win32 虚拟内存管理器使用另外一个数据结构来记录和维护每个进程的 4 GB 虚拟地址空间的使用及状态信息，这就是虚拟地址描述符树（Virtual Address Descriptor，VAD）。每一个进程都有一个自己的 VAD 集合，这个集合中的 VAD 被组织成一个自平衡二叉树，以提高查找的效率。另外只有预留或者提交的内存块才会有 VAD，自由的内存块没有 VAD（因此不在 VAD 树结构中的虚拟地址块就是自由的）。VAD 的组织如图 4-5 所示。

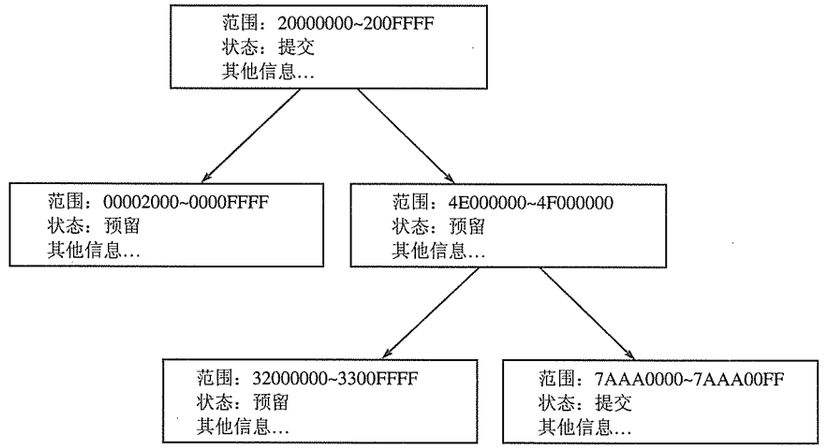


图 4-5 VAD 的组织结构

（1）当程序申请一块新内存时，虚拟内存管理器只需访问 VAD 树。找到两个相邻 VAD，只要小的 VAD 的上限与大的 VAD 的下限之间的差值满足所申请的内存块的大小需求，即可使用二者之间的虚拟内存。

（2）当第一次访问提交的内存时，虚拟内存管理器根据上一节描述的流程。即总是假定该数据页已在物理内存中，并进行虚拟地址到物理地址的转换。当找到相应的页目录项后发现该页目录项并没有指向一个合法的页表，它就会查找该进程的 VAD 树。找到包含该地址的 VAD，并根据 VAD 中的信息，比如该内存块的大小、范围，以及在调页文件中的起始位置等，随需生成相应的页表项，然后从刚才发生缺页错误的地方继续进行地址映射。由此可以看出，一个虚拟内存页被提交时，除了在调页文件中开辟一个备份页之外，不会生成包含指向它的页表项的页表，也不会填充指向它的页表项，更不会为之开辟真正的物

理内存页，而是直到第一次访问这个提交页时，才会“随需地”从 VAD 中取得包含该页的整个区域的信息，生成相应页表，并填充相应页的表项。

（3）当访问预留的内存时，虚拟内存管理器也是根据上一节描述的流程进行虚拟地址到物理地址的映射，找到相应的页目录项后发现该页目录项并没有指向一个合法的页表，它就会查找该进程的 VAD 树，找到包含该地址的 VAD。这时它会发现此段内存块只是预留的，而没有提交，即并没有对应的真正的物理存储，这时直接抛出访问违例，进程退出。

（4）当访问自由的内存时，虚拟内存管理器还是根据上一节描述的流程进行虚拟地址到物理地址的映射。找到相应的页目录项后发现该页目录项并没有指向一个合法的页表，它就会查找该进程的 VAD 树，发现并没有 VAD 包含此虚拟地址，此时可以知道该地址所在的虚拟地址页是自由状态，直接抛出访问违例，进程退出。

4.1.5 进程工作集

因为频繁的调页操作引起的磁盘 I/O 会大大降低程序的运行效率，因此对每一个进程，虚拟内存管理器都会将其一定量的内存页驻留在物理内存中。并跟踪其执行的性能指标，动态调整这个数量。Win32 中驻留在物理内存中的内存页称为进程的“工作集”（working set），进程的工作集可以通过“任务管理器”查看，其中“内存使用”列即为工作集大小。图 4-6 中绿色方框的数字是笔者写作本书时所用 Word 编辑器的工作集大小，即 38740 KB。

工作集是会动态变化的，进程初始时只有很少的代码页和数据页被调入内存。当执行到未被调入内存的代码或者访问到尚未调入内存的数据时，这些代码页或者数据页会被调入物理内存，工作集也随之增长。但工作集不能无限增长，系统为每个进程都定义了一个默认的最小工作集（根据系统物理内存大小，此值可能为 20~50 MB）和最大工作集（根据系统物理内存大小，此值可能为 45~345 MB）。当工作集到达最大工作集，即进程需要再次调入新页到物理内存中时，虚拟内存管理器会将其原来的工作集中的某些页先置换出内存，然后将需要调入的新页调入内存。

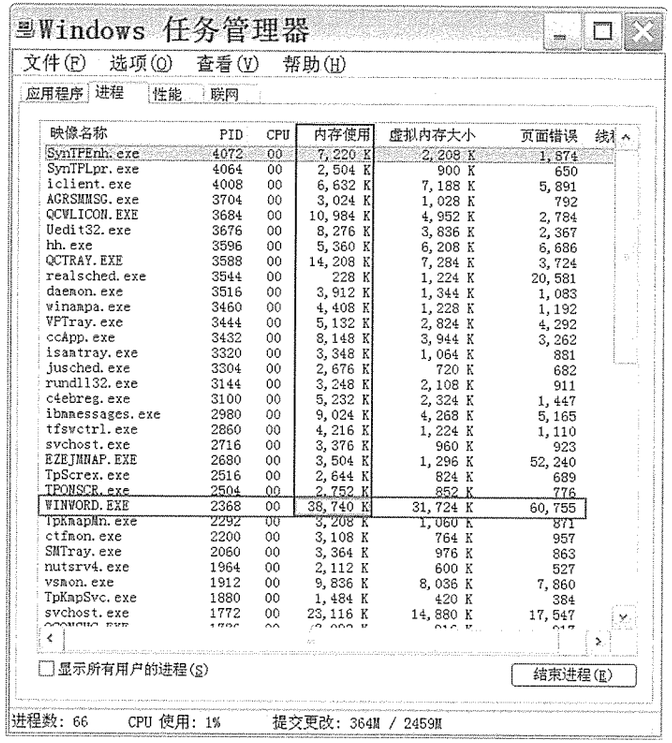


图 4-6 工作集

因为工作集的页驻留在物理内存中，因此对这些页的访问不涉及磁盘 I/O，相对而言非常快；反之，如果执行的代码或者访问的数据不在工作集中，则会引发额外的磁盘 I/O，从而降低程序的运行效率。一个极端的情况就是所谓的颠簸或抖动（thrashing），即程序的大部分的执行时间都花在了调页操作上，而不是代码执行上。

如前所述，虚拟内存管理器在调页时，不仅仅只是调入需要的页，同时还将其附近的页也一起调入内存中。综合这些知识，对开发人员来说，如果想提高程序的运行效率，应该考虑以下两个因素。

（1）对代码来说，尽量编写紧凑代码，这样最理想的情形就是工作集从不会到达最大阀值。在每次调入新页时，也就不需要置换已经载入内存的页。因为根据 locality 特性，以前执行的代码和访问的数据在后面有很大可能会被再次执行或访问。这样程序执行时，发

生的缺页错误数就会大大降低，即减少了磁盘 I/O，在图 4-6 中也可以看到一个程序执行时截至当时共发生的缺页错误次数。即使不能达到这种理想情形，紧凑的代码也往往意味着接下来执行的代码更大可能就在相同的页或相邻页。根据时间 locality 特性，程序 80%的时间花在了 20%的代码上。如果能将这 20%的代码尽量紧凑且排在一起，无疑会大大提高程序的整体运行性能。

（2）对数据来说，尽量将那些会一起访问的数据（比如链表）放在一起。这样当访问这些数据时，因为它们在同一页或相邻页，只需要一次调页操作即可完成；反之，如果这些数据分散在多个页（更糟的情况是这些页还不相邻），那么每次对这些数据的整体访问都会引发大量的缺页错误，从而降低性能。利用 Win32 提供的预留和提交两步机制，可以为这些会一同访问的数据预留出一大块空间。此时并没有分配实际存储空间，然后在后续执行过程中生成这些数据时随需为它们提交内存。这样既不浪费真正的物理存储（包括调页文件的磁盘空间和物理内存空间），又能利用 locality 特性。另外内存池机制也是基于类似的考虑。

4.1.6 Win32 内存相关 API

在 Win32 平台下，开发人员可以通过如下 5 组函数来使用内存（完成申请和释放等操作）。

（1）传统的 CRT 函数（malloc/free 系列）：因为这组函数的平台无关性，如果程序会被移植到其他非 Windows 平台，则这组函数是首选。也正因为这组函数非 Win32 专有，而且介绍这组函数的资料俯拾皆是，这里不作详细介绍。

（2）global heap/local heap 函数（GlobalAlloc/LocalAlloc 系列）：这组函数是为了向后兼容而保留的。在 Windows 3.1 平台下，global heap 为系统中所有进程共有的堆，这些进程包括系统进程和用户进程。它们对此 global heap 内存的申请会交错在一起，从而使得一个用户进程的不小心的内存使用错误会导致整个操作系统的崩溃。local heap 又被称为“private heap”，与 global heap 相对应，local heap 为每个进程私有。进程通过 LocalAlloc 从自己的 local heap 里申请内存，而不会相互干扰。除此之外，进程不能通过另外的用户自定义堆或者其他方式动态地申请内存。到了 Win32 平台，由于考虑到安全因素，global heap 已经废弃，local heap 也改名为“process heap”。为了使得以前针对 Windows 3.1 平台写的

应用程序能够运行在新的 Win32 平台上，GlobalAlloc/ LocalAlloc 系列函数仍然得到沿用，但是这一系列函数最后都是从 process heap 中分配内存。不仅如此，Win32 平台还允许进程除 process heap 之外生成和使用新的用户自定义堆，因此在 Win32 平台下建议不使用 GlobalAlloc/LocalAlloc 系列函数进行内存操作，因此这里不详细介绍这组函数。

（3）虚拟内存函数（VirtualAlloc/VirtualFree 系列）：这组函数直接通过保留（reserve）和提交（commit）虚拟内存地址空间来操作内存，因此它们为开发人员提供最大的自由度，但相应地也为开发人员内存管理工作增加了更多的负担。这组函数适合于为大型连续的数据结构数组开辟空间。

（4）内存映射文件函数（CreateFileMapping/MapViewOfFile 系列）：系统使用内存映射文件函数系列来加载.exe 或者.dll 文件。而对开发人员而言，一方面通过这组函数可以方便地操作硬盘文件，而不用考虑那些繁琐的文件 I/O 操作；另一方面，运行在同一台机器上的多个进程可以通过内存映射文件函数来共享数据（这也是同一台机器上进程间进行数据共享和通信的最有效率和最方便的方法）。

（5）堆内存函数（HeapCreate/HeapAlloc 系列）：Win32 平台中的每个堆都是各进程私有的，每个进程除了默认的进程堆，还可以另外创建用户自定义堆。当程序需要动态创建多个小数据结构时，堆函数系列最为适合。一般来说 CRT 函数（malloc/free）就是基于堆内存函数实现的。

1. 虚拟内存

虚拟内存相关函数共有 4 对，即 VirtualAlloc/VirtualFree、VirtualLock/VirtualUnlock、VirtualQuery/VirtualQueryEx 及 VirtualProtect/VirtualProtectEx。其中最重要的是第一对，本节主要介绍这一对。

```
LPVOID VirtualAlloc(
    LPVOID lpAddress,
    DWORD dwSize,
    DWORD flAllocationType,
    DWORD flProtect
);
```

VirtualAlloc 根据 flAllocationType 的不同，可以保留一段虚拟内存区域（MEM_RESERVE）或者提交一段虚拟内存区域（MEM_COMMIT）。当保留时，除了修改进程的

VAD 之外（准确地说是增加了一项），并没有分配其他资源，如调页文件空间或者实际物理内存，甚至没有创建页表项。因此非常快捷，而且执行速度与保留空间的大小没有关系。因为保留仅仅只是让内存管理器预留一段虚拟地址空间，并没有实在的存储（硬盘上的调页文件空间或者物理内存），因此访问保留地址会引起访问违例，这是一种严重错误，会直接导致进程退出；相反，提交虚拟内存时，内存管理器必须从系统调页文件中开辟实际的存储空间，因此速度会比保留操作慢。但是需要注意的是，此时在物理内存中并没有立刻分配空间用来与这段虚拟内存空间相对应，甚至也没有相应的页表项被创建，但是提交操作会相应修改 VAD 项。只有首次访问这段虚拟地址空间中的某个地址时，由于缺页中断，虚拟内存管理器查找 VAD，接着根据 VAD 的内容，动态创建 PTE，然后根据 PTE 信息，分配物理内存页，并实际访问该内存。由此可见，真正花费时间的操作不是提交内存，而是对提交内存的第一次访问！这种 lazy-evaluation 机制对程序运行性能是十分有益的，因为如果某个程序提交了大段内存，但只是零星地对其中的某些页进行访问，如果没有这种 lazy-evaluation 机制，提交大段内存会极大地降低系统的性能。

与之相对，VirtualFree 释放内存，它提供两种选择：可以将提交的内存释放给系统，但是不释放保留的虚拟内存地址空间；也可以在释放内存的同时将虚拟内存地址空间一并释放，这样这块虚拟内存地址空间的状态变回初始的自由状态。如果内存是提交状态，VirtualFree 因为会释放真正的存储空间而比较慢；如果只是释放保留的虚拟内存地址空间，那么因为只需要修改 VAD，该操作会很快。

除此之外，VirtualLock 保证某块内存在 lock 期间一直在物理内存中，因此对该内存的访问不会引起缺页中断。lock 的内存用 VirtualUnlock 解锁。因为 VirtualLock 会把内存锁定在物理内存中，如果这些内存实际中访问的并不频繁，那么会使得其他经常使用到的内存反而增大了被调页出去的概率，从而降低了系统的整体性能，因此在实际使用中，并不推荐使用 VirtualLock/VirtualUnlock 函数。VirtualQuery 可以获得传入指针所在的虚拟内存块的状态，如包含该指针所在页的虚拟内存区域的基址，以及该区域的状态等。VirtualProtect 可用来修改某段区域的提交内存页的存取保护标志。

2. 内存映射文件

内存映射文件主要有三个用途，Windows 利用它来有效使用 exe 和 dll 文件，开发人员利用它来方便地访问硬盘文件，或者实现不同进程间的内存共享。第一种这里不详细介绍，只介绍后两种用途。首先讨论它提供的方便访问硬盘文件的机制，一旦通过这种机制将一

个硬盘文件（部分或者全部）映射到进程的一段虚拟地址空间中，读写该文件的内容就像通过指针访问变量一样。假设 pViewMem 为文件映射到内存的首址，那么：

```
*pViewMem = 100;                    //写文件的第 1 个字节
char ch = *(pViewMem + 50);         //读文件的第 50 个字节内容
```

下面介绍这种机制的使用步骤。

（1）新建或者打开一个硬盘文件。

此步骤用来获得一个文件对象的句柄，用 CreateFile 函数来新建或者打开一个文件：

```
HANDLE CreateFile(
PCSTR pszFileName,
DWORD dwDesiredAccess,
DWORD dwShareMode,
PSECURITY_ATTRIBUTES psa,
DWORD dwCreationDisposition,
DWORD dwFlagsAndAttributes,
HANDLE hTemplateFile);
```

其中 pszFileName 参数指示该文件的路径名，dwDesiredAccess 参数表示该文件内容将会被如何访问，此参数包括 0 、GENERIC_READ、GENERIC_WRITE，以及 GENERIC_READ | GENERIC_WRITE 共 4 种可能，分别表示“不能读也不能写”（在只为了读取该文件属性时使用）、“只读”、“只写”，以及“既可读也可写”；dwShareMode 参数用来限定对该文件的任何其他访问的权限，也包括上述 4 种类型。剩余的几个参数因为与要讨论的问题关系不大，所以不赘述。

此函数成功时，会返回一个文件对象句柄；否则会返回 INVALID_HANDLE_ VALUE。

（2）创建或者打开一个文件映射内核对象。

还需要有一个文件映射内核对象，正是它真正将文件内容映射到内存中。如果已经存在此内核对象，只需通过 OpenFileMapping 函数将其打开即可，这个函数返回该命名对象的句柄。大多数情况下，需要新建一个文件映射内核对象，此时调用 CreateFileMapping 函数：

```
HANDLE CreateFileMapping(
HANDLE hFile,
PSECURITY_ATTRIBUTES psa,
DWORD fdwProtect,
```

```
DWORD dwMaximumSizeHigh,
DWORD dwMaximumSizeLow,
PCTSTR pszName);
```

hFile 参数是第一个步骤中返回的文件内核对象句柄；psa 参数是指明内核对象安全特性的，不详述；fdwProtect 参数指明了对映射到内存页中的文件内容的存取权限，这个权限必须与第一个步骤中的文件访问权限对应；dwMaximumSizeHigh 和 dwMaximumSizeLow 参数指明映射的最大的空间大小，因为 Windows 支持大小达到 64 位的文件，因此需要两个 32 位的参数；pszName 为内核对象名称。

此步只是创建了一个文件映射内核对象，并没有预留或者提交虚拟地址空间，更没有物理内存页被分配出来存放文件内容。

（3）映射文件的内容到进程虚拟地址空间。

访问文件内容之前，必须将要访问的文件内容映射到内存中，通过 MapViewOfFile 函数完成：

```
PVOID MapViewOfFile(
HANDLE hFileMappingObject, '
DWORD dwDesiredAccess,
DWORD dwFileOffsetHigh,
DWORD dwFileOffsetLow,
SIZE_T dwNumberOfBytesToMap);
```

其中参数分别为：用来映射内存映射内核对象的句柄，映射的文件内容到内存内存页的存取权限，需要映射的文件内容的起始部分在文件中的偏移及大小。映射时并不需要一次将整个文件的内容全部映射到内存中。

这个函数的操作包括从进程虚拟地址空间中预留出所需映射大小的一段区域，然后提交。提交时并不是从系统的调页文件中开辟空间用来作为该段区域的备份存储，而是内存映射内核对象所对应的文件的指明区域。与虚拟内存使用的惟一不同就是该段虚拟地址空间区域的备份存储不同，其他都是一样的。同样，此时并没有真正的物理内存开辟出来，直到通过返回的指针访问已经映射到内存中的文件内容时，因为发生缺页错误，系统才会分配物理内存页，并将对应的文件存储中的内容调页到该物理内存页。

（4）访问文件内容。

现在可以通过 MapViewOfFile 函数返回的指针来访问该段映射到内存中文件内容，就

像本小节演示的那样，通过指针访问硬盘文件内容。

这里需要提醒的是，通过该指针修改文件内容时，修改的结果常常不会立刻反映到文件中，因为实际上是在对调入物理内存页中的数据进行修改。考虑到性能因素，该页并不会每做一次修改就立刻将该修改同步到硬盘文件中。如果需要在某个时候强制将之前所做的修改一次性同步到与之对应的硬盘文件中时，可以通过 FlushViewOfFile 函数达到这个目的：

```
BOOL FlushViewOfFile(PVOID pvAddress, SIZE_T dwNumberOfBytesToFlush);
```

这个函数传入需要将修改同步到硬盘文件中的内存块的起始地址和大小。

（5）取消文件内容到进程虚拟地址空间的映射。

当该段映射到内存中的文件内容访问完毕，不再需要访问时，为了有效地利用系统的资源，应该及时回收该段内存，这时调用 UnmapViewOfFile 函数：

```
BOOL UnmapViewOfFile(PVOID pvBaseAddress);
```

此函数传入 MapViewOfFile 函数返回的指针，系统回收对应的 MapViewOfFile 调用时预留并提交的虚拟内存地址空间区域，这样该段区域可被其他申请使用。另外因为对应的备份存储不是系统的调页文件，所以不存在备份存储回收的问题。

（6）关闭文件映射内核对象和文件内核对象。

最后，在完成任务不再使用该文件时，通过 CloseHandle(hFile)和 CloseHandle (hMapping)来关闭文件并释放内存映射文件的内核对象句柄。

下面接着讨论如何利用内存映射文件内核对象来进行进程间的内存共享。

进程间通过内存映射文件进行内存共享时，该内存映射文件内核对象常常不是基于某一个硬盘文件，而是从系统的调页文件中开辟空间作为临时用做共享的存储空间。因此与单纯地利用内存映射文件来访问硬盘文件内容稍有不同，下面是通过内存映射文件来进行进程间内存共享的步骤。假设有进程 A 和进程 B，进程 A 通过 CreateFileMapping 创建一个基于系统调页文件的名为“SharedMem”的内存映射文件内核对象：

```
HANDLE m_hFileMapA = CreateFileMapping
(INVALID_HANDLE_VALUE, NULL, PAGE_READWRITE, 0,
10 * 1024, TEXT("SharedMem"));
```

需要注意的是，因为现在不再基于普通的硬盘文件，所以不需要调用 CreateFile 来新建或者打开文件这个步骤，注意此时传入的文件句柄参数为 INVALID_HANDLE_VALUE，此参数代表从调页文件中开辟空间作为共享内存。

进程 B 通过 OpenFileMapping 打开刚才进程 A 创建的名为“SharedMem”的内存映射文件内核对象：

```
HANDLE m_hFileMapB = OpenFileMapping(..., TEXT("SharedMem"));
```

进程 A 和进程 B 都可以用此内存映射文件内核对象将从系统调页文件中开辟的那块存储空间的全部或者部分映射到内存中，然后即可使用。

进程 A：

```
...
PVOID pViewA = MapViewOfFile(m_hFileMapA, FILE_MAP_READ | FILE_MAP_WRITE, 0, 0,
0);
...
```

进程 B：

```
...
PVOID pViewB = MapViewOfFile(m_hFileMapB, FILE_MAP_READ | FILE_MAP_WRITE, 0, 0,
0);
...
```

它们各自对该共享内存的修改都能够及时地被对方看到。另外需要注意的是，它们映射到的虚拟内存空间区域并不一定有相同的起始地址，这是因为它们拥有自己的虚拟地址空间。

还有一个需要引起注意，但很难发现的问题是因为创建基于系统调页文件的内存映射文件内核对象是通过传入 hFile 为 INVALID_HANDLE_VALUE 的参数来标记的，而创建或者打开普通硬盘文件失败时的返回值也是 INVALID_HANDLE_VALUE，因此诸如下面这段代码存在的 bug 是很难发现的：

```
...
HANDLE hFile = CreateFile(...);
HANDLE hMap = CreateFileMapping(hFile, ...);
if (hMap == NULL)
return(GetLastError());
...
```


这段代码的本意是首先创建或者打开一个普通的硬盘文件，然后创建一个基于此文件的内存映射文件内核对象，而并不是想创建一个基于系统调页文件的该对象。但是可以看到，当第 1 句 CreateFile 执行失败时，返回 INVALID_HANDLE_VALUE。这个返回值立刻被传入到 CreateFileMapping 函数，结果创建了一个基于系统调页文件的内存映射文件内核对象。这并不是这段代码的原意，而且也会造成问题。因为基于普通硬盘文件的内存映射文件内核对象的操作往往希望将最后的结果保存在该文件中，而基于系统调页文件的内存映射文件内核对象的操作往往只是关注该数据在执行期的结果，操作完毕后并不保存该结果。当 CreateFile 失败且程序运行后，程序运行无误。但是当检查结果文件时，会发现该文件要么没有被创建，要么数据没有改动，因为随后的操作都是基于系统调页文件的！

因此当使用基于普通硬盘文件的内存映射文件内核对象时，一定要在 CreateFile 调用完后检查返回值。

3. 堆

分配多个小块内存一般都会选择使用堆函数，比如链表节点和树节点等，堆函数的最大优点就是开发人员不用考虑页边界之类的琐碎事情；劣势就是堆函数的操作相对虚拟内存和内存映射文件来说速度要慢些，而且无法像虚拟内存或者内存映射文件那样直接提交或者回收物理存储。

进程都有一个默认的堆，其初始区域大小默认是 1 MB，链接时可以通过/HEAP 参数修改此默认值。很多操作的临时存储都使用进程的默认堆，比如绝大多数的 Win32 函数，进程默认堆的句柄可以通过 GetProcessHeap 函数获得。

因为程序大部分的内存需求都是从进程默认堆中分配的，而且在多线程情况下还需要考虑线程安全问题。因此对特定的应用，这种情况会造成程序的性能下降。针对这种需求，Win32 提供了自定义堆机制。

自定义堆的步骤如下。

（1）创建自定义堆。

与进程默认堆（进程创建时系统自动创建）不同，自定义堆需要开发人员首先通过 HeapCreate 函数创建：

```
HANDLE HeapCreate(
```

```
DWORD fdwOptions,
SIZE_T dwInitialSize,
SIZE_T dwMaximumSize);
```

fdwOptions 参数可以指明是否需要串行化访问支持（HEAP_NO_SERIALIZE），以及分配和回收内存出错时是否抛出异常（HEAP_GENERATE_EXCEPTIONS）。当该自定义堆会被多个线程同时访问时，需要加上串行化访问支持，但相应的性能会有所下降。

dwInitialSize 参数指明该自定义堆创建时提交的存储大小（页大小的倍数），dwMaximumSize 参数则指明该自定义堆从进程虚拟地址空间中预留出的区域大小。随着对此自定义堆内存的分配，提交的存储大小随之变大，但此参数限制了增大的极限。另一种情况时是 dwMaximumSize 为 0，此时该自定义堆可以一直增长，直到进程虚拟地址空间用完。

（2）从自定义堆中分配内存。

从自定义堆中分配内存调用函数 HeapAlloc（从进程默认堆中分配内存也调用此函数）：

```
PVOID HeapAlloc(
HANDLE hHeap,
DWORD fdwFlags,
SIZE_T dwBytes);
```

hHeap 参数即上一步骤中返回的堆内核对象句柄，fdwFlags 可以取 HEAP_ZERO_MEMORY、HEAP_GENERATE_EXCEPTIONS 和 HEAP_NO_SERIALIZE 共 3 个值，HEAP_ZERO_MEMORY 指明返回的内存必须全部清 0。HEAP_GENERATE_ EXCEPTIONS 指明此次分配内存如果失败，需要抛出异常。如果该自定义堆创建时指明过此参数，则其上的内存分配不必再指明此参数；如果堆创建时没有指明，则可以在每次申请时指明。HEAP_NO_SERIALIZE 参数指明此次分配不必串行化访问支持。最后的 dwBytes 参数指明此次分配的内存大小，返回值为分配内存的起始位置。

（3）释放内存。

从堆中释放内存调用 HeapFree 函数：

```
BOOL HeapFree(
HANDLE hHeap,
DWORD fdwFlags,
PVOID pvMem);
```


这个函数的参数意义很明显，无须赘述。需要指出的是，这样释放内存并不能保证所有物理存储被回收，一是因为物理存储以页大小为单位判断是否可以回收；二是 Windows 设计堆机制时对效率的考虑。

（4）销毁自定义堆。

当程序不再需要使用某个自定义堆时，调用 HeapDestroy 函数：

```
BOOL HeapDestroy(HANDLE hHeap);
```

对堆的销毁有几点需要说明，一是当堆销毁时，所有从该堆分配的内存全部被回收，而不必对那些内存一一进行释放，同时该堆占用物理存储以及虚拟地址空间区域也会被系统回收；二是如果没有显式销毁自定义堆，这些堆会在程序退出时被系统销毁。需要注意的是，线程创建的自定义堆并不会在线程退出时被销毁，而是当整个进程退出时才会被销毁，从资源利用效率角度出发，应该在自定义堆不再被使用时立即销毁；三是进程默认堆不能通过此函数销毁，更严格地说，进程默认堆在进程退出前是不能被销毁的。

自定义堆的其他函数如下。

（1）获得进程所有堆：

```
DWORD GetProcessHeaps(
DWORD dwNumHeaps,
PHANDLE pHeaps);
```

此函数返回进程目前所有的堆（包括进程默认堆），传入存放所有堆内核对象句柄的数组，以及数组的大小，返回值为堆数目。

（2）修改分配内存的大小：

```
PVOID HeapReAlloc(
HANDLE hHeap,
DWORD fdwFlags,
PVOID pvMem,
SIZE_T dwBytes);
```

这个函数可以修改原来分配的内存块（pvMem）的大小，新的大小由参数 dwBytes 指明。

（3）查询某块分配内存的大小：

```
SIZE_T HeapSize(
```

```
HANDLE hHeap,
DWORD fdwFlags,
LPCVOID pvMem);
```

这个函数可以查询到原来分配的一个内存块的大小。当该内存块指针是外部模块传入时，如果需要知道该块确切大小时，这个函数就可以发挥作用。

（4）堆压缩：

```
UINT HeapCompact(
HANDLE hHeap,
DWORD fdwFlags);
```

此函数将相邻的回收回来的自由块合并在一起，需要注意的是，这个函数并不能移动已经分配的内存块，即它并不能消除内存碎片。

自定义堆有如下优点。

（1）减少碎片，节省内存：由于大多数自定义堆是为某些特定的数据结构创建的，所以这些数据结构大小相同，从而使得上次释放的空间有更大机会刚好可以满足下一次的内存申请，从而减少了碎片的产生。

（2）由于局部性获得的性能提高：由上所述，自定义堆上的内存块大多是某些特定的数据结构，比如链表节点或者树节点。这些数据结构有着很强的时间局部性，即程序往往会在相邻的时间内访问所有这些数据。如果这些数据都放在某一个自定义堆中，这种空间局部性就会极大地减少对这些数据整体访问引起的缺页错误，从而提高了程序的运行性能。

（3）由避免线程同步获得的性能提高：如前所述，因为进程默认堆可能会被多个线程同时访问，因此添加了保证线程安全的串行化访问支持，但串行化访问支持的代价就是性能下降。如果某个自定义堆只允许某单个线程访问，那么此自定义堆不必添加串行化访问支持，从而提高程序的性能。

4.2 Linux 内存管理机制

Linux 的内存管理主要由两个部分组成，一个部分负责物理内存的申请与释放，物理内存的申请与释放最小单位与 Windows 一样，都是以“页”为单位，在 IA32 中页的大小是 4 KB；

另外一个部分负责处理虚拟内存，虚拟内存的操作主要包括虚拟地址空间与实际存储空间的映射，物理内存页与硬盘页之间的置换等，下面各节分别介绍这几个方面的内容。

4.2.1 进程的内存布局

一个 32 位 Linux 进程的地址空间为 4 GB，与 Windows 类似，这个 4 GB 空间并不能全部被一个 Linux 进程的用户空间代码使用，而是有一部分留给了内核。但与 Windows 不同，Linux 中 4 GB 的高位 1 GB 空间留给了内核，低位的 3 GB 由进程用户代码使用，如图 4-7 所示。

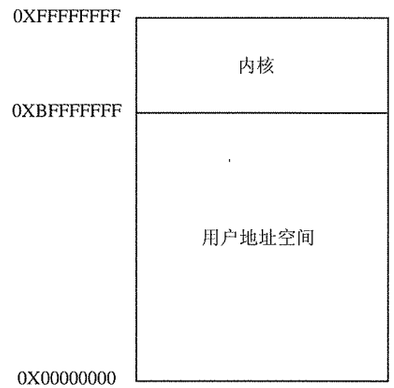


图 4-7 一个 32 位 Linux 进程的地址空间

0X00000000~0XBFFFFFFF 这一 3 GB 区域为用户地址空间，而上面的 0XC0000000 ~ 0XFFFFFFFF 这一 1 GB 区域保留给内核使用。用户地址空间进一步被分成程序代码区、数据区（包括初始化数据区 DATA 和未初始化数据区 BSS）、堆和栈。程序代码区占据最底端，紧接着往上为数据区，先是初始化数据区，然后是未初始化数据区。代码区中存放应用程序的机器代码，运行过程中代码不能被修改，因此代码区内存为只读，且大小固定。数据区中存放了应用程序中的全局数据，静态数据和一些常量字符串等。数据区的大小也是固定的。

如图 4-8 所示，除了代码区，初始化数据区和未初始化数据区之外，还有两个动态增长和缩减的区域即堆和栈。

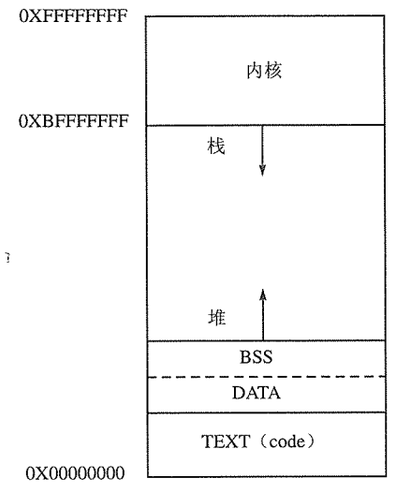


图 4-8　堆和栈

堆从未初始化数据区开始，向上端动态增长，增长过程中虚拟地址值变大；而栈则从高位地址开始，向下端动态增长，虚拟地址值变小。

堆是应用程序在运行过程中动态申请的内存空间，如开发人员通过 malloc/new 需要动态生成对象或者开辟内存空间时，最终会调用系统调用 brk 来动态调整数据区的大小。当这些动态内存区域使用完毕，需要开发人员明确使用相应的 free/delete 释放这些内存空间。free/delete 最终也会调用到 brk 调整数据区的大小。

栈与堆有明显的区别，栈用来存放函数的传入参数、临时变量，以及返回地址等数据。这些数据不需要通过 malloc/new 来为之开辟内存空间，而且其增长与缩减是因为函数的调用与返回，而不必开发人员的额外操作。堆空间的维护不需要开发人员的参与，而且也没有内存泄漏的危险。

初始化数据区和未初始化数据区还有一些值得注意的事情，初始化数据区存放的是那些在编译期就能够知道由程序设定初始值的全局变量及静态变量等。这些初始值必须保存在最终生成的二进制文件中，并且在程序运行时会原封不动地将这个区域映射到进程的初始化数据区域。当程序声明 N 个这样的初始化数据，且其空间占据 M 大小，那么在二进制文件中，就会开辟 M 大小的区域。即依次存放 N 个数据，且每个都设置了相应的初始值。当程序运行时，这块区域就会原封不动地映射到该对应进程的“初始化数据区”。“原封不

动”意味着在二进制文件中这块区域的大小与进程虚拟地址空间中一样大，而且排列和对应地址的值也一样。

但未初始化数据区则不如此，如果一个全局变量或者静态变量在源代码文件中没有被赋予初始值，那么在程序启动后，在第 1 次被赋值前，其初始值为 0。即这些数据本质上还是有初始值的，只不过初始值为 0。但是当最终生成二进制文件时，这些未初始化数据区并不会像初始化数据区那样，占据它们对应变量的总大小的区域，而只是用一个值来记录其总大小。比如，一个程序的代码指令有 100 KB 大小，所有的初始化数据总大小为 100 KB，所有的未初始化数据总大小为 150 KB，那么在最终生成的二进制文件中，代码区会有 100 KB。接着 100 KB 大小的初始化数据区，然后接着一个 4 个字节大小的空间，其值为 150×1024（指示其大小为 150 KB），注意，这里不是一个 150 KB 大小的所有单元值都为 0 的空间。这样可以节省二进制文件的大小，即节省硬盘空间。但是在进程虚拟地址空间中，对应于这一区域的未初始化数据区的大小必须是 150 KB。因为在程序运行时，程序必须能够真正访问到这些变量中的每一个。这也意味着，当程序启动时，当监测到二进制文件中未初始化数据区的那个值为 150×1024 时，系统会开辟出一个 150 kB 大小的区域作为进程的未初始化数据区域并且同时用 0 来初始化这一区域。

4.2.2　物理内存管理

物理内存是用来存放代码指令与供代码指令操作的数据的最终场所，因此物理内存的管理是一个内存管理系统的极其重要的一个任务。与 Windows 系统使用页帧数据库来管理物理内存相对应，Linux 系统使用页分配器（page allocator）来管理物理内存，页分配器负责分配和回收所有的物理内存页（物理内存的分配与回收的最小单元为 4 KB 大小的页）。

页分配器的核心算法称为“兄弟堆算法”（buddy-heap algorithm），这个算法的思想是每个物理内存区域都会有一个与之相邻的所谓“兄弟”区域。当这两个区域被回收后，会被合并成为一个区域。进一步，当这个被合并区域的相邻区域也被回收后，它们会被进一步合并成为一个更大的区域。当有物理内存请求到来时，页分配器会首先检测是否有大小与之一致的区域。如果有，直接用找到的匹配区域满足这一请求；如果没有，则找到一个更大的区域。并继续划分，直到分出的区域能够满足这一请求。

配合这一算法的是必须有链表用来记录自由的物理内存区域，对于每个相同大小的自由区域，会有一个链表用来将它们串联起来。这样每种大小的自由区域，都会有一个链表用来串联它们。这些自由区域的大小都是 2 的某次幂，比如 1，2，4，8……。

如图 4-9 所示，当有一个 8 KB（2 页）的请求到来，当前最小可供分配的区域为 64 KB 大小。这时，会先分成两个 32 KB 大小的区域。继而将低位的 32 KB 大小区域分成两个 16 KB 大小的区域，再将最低位的 16 KB 大小的区域分成两个 8 KB 大小的区域，然后分配高位的 8 KB 以满足申请要求。

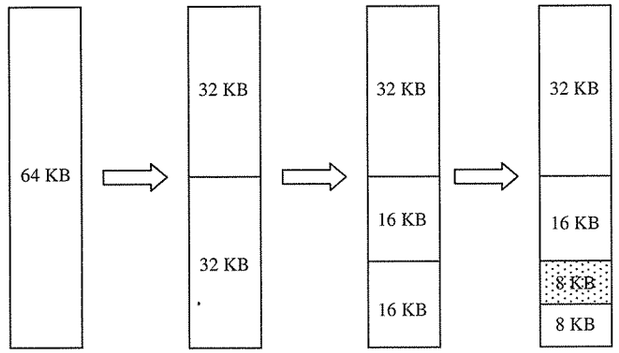


图 4-9　物理内存管理

4.2.3　虚拟内存管理

虚拟内存管理器的主要任务是维护应用程序的虚拟地址空间使用信息，如哪些区域已被使用——映射，这些区域是否有硬盘文件作为备份存储。如果有，该区域对应在硬盘的哪个区域等；另外一个重要的功能就是调页，如在程序访问某些尚未调至物理内存的数据时，虚拟内存管理器负责定位它们，并将其置换进物理内存。如果物理内存这时没有自由页，还需要将物理内存中的某些页先置换出去等。

用来维护应用程序的虚拟地址空间使用信息的数据结构是 vm_area_struct（这个数据结构所维护的信息和作用都类似于 Windows 系统中的 VAD，而且它们组织方式也很类似，都是二叉平衡树），每个 vm_area_struct 结构都描述了一个进程虚拟地址空间中被分配的区域。当 vm_area_struct 的个数不超过 32 时，这些 vm_area_struct 被链成一个链表结构；当超过 32 时，这些 vm_area_struct 则被组织成一个二叉平衡树，采用这种数据结构的目的是为了

提高查询速度。当程序通过某个指针访问某个数据时，系统会查询这个 vm_area_struct 二叉平衡树。如果发现这个指针没有落在任何一个 vm_area_struct 所表示的区域内，则判定这个指针所代表的地址还没有被分配。即这是一个非法的指针访问（类似于 Windows 系统中的 access violation），也是一个很严重的程序错误。

```
struct vm_area_struct
{
    struct mm_struct * vm_mm;
    unsigned long vm_start;
    unsigned long vm_end;
    struct vm_area_struct *vm_next;
    pgprot_t vm_page_prot;
    unsigned short vm_flags;
    short vm_avl_height;
    struct vm_area_struct * vm_avl_left;
    struct vm_area_struct * vm_avl_right;
    struct vm_area_struct *vm_next_share;
    struct vm_area_struct **vm_pprev_share;
    struct vm_operations_struct * vm_ops;
    unsigned long vm_offset;
    struct file * vm_file;
    unsigned long vm_pte;
};
```

4.2.4　虚拟地址映射为物理地址

当通过程序的指针访问某个数据时，因为指针值本质上是一个虚拟地址值。所以这个虚拟地址值必须首先被转换为一个物理地址值，才能真正访问其所指代的数据。

与 Windows 虚拟内存管理系统不同的是，Linux 使用了 3 层映射策略将一个虚拟地址映射为一个物理地址。

可以看到，相比 Windows 系统，Linux 的 3 层映射多了 Middle 这一层。但是对于 IA32 体系来说，Middle 这一层实际上是无用的。因此 Linux 的虚拟地址映射物理地址的方式实际上与 Windows 上是一样的，参考本章的 4.1.3 节。

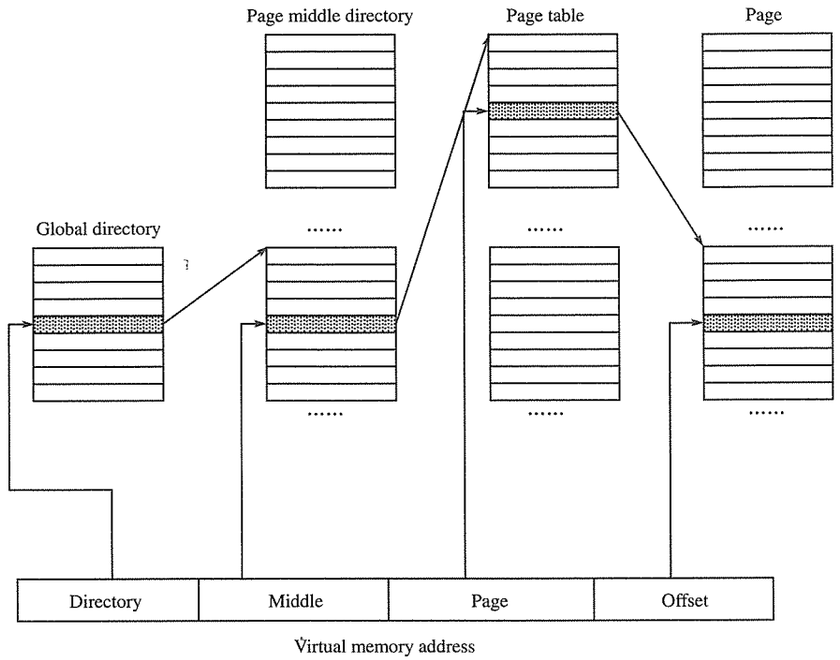


图 4-10　虚拟地址映射为物理地址

4.3　本章小结

无论是 malloc/free，还是 new/delete 都是通过操作系统的内存管理机制实现的，因此理解底层操作系统的内存管理机制对编写正确且高效的应用程序可谓至关重要。本章介绍了当前最流行的两种操作系统——Windows 和 Linux 的内存管理机制，并分析了缺页的原因及其对性能的重要影响，同时提供了一些常用的用来减少缺页的思路与方法。虽然通常编写应用程序时都倾向于直接利用语言提供的 malloc/free（C 语言）或 new/delete（C++语言）来操纵内存（第 5 章），但往往也会在开发一些应用程序时发现这些内存操作成为了最后的瓶颈所在，这时，需要考虑运用内存池（第 6 章）或者直接利用操作系统提供的内存相关的 API 函数来操纵内存以提高性能。

第 5 章　动态内存管理

通过第 1 章可以知道，C++程序中的存储空间可以分为静态/全局存储区，以及栈区和堆区。静态/全局存储区和栈区的大小一般在程序编译阶段决定；而堆区则随着程序的运行而动态变化，每一次程序运行都会有不同的行为。这种动态内存的管理对于一个程序在运行过程中占用的内存大小及程序运行的性能有非常重要的影响。本章将介绍在 C++中如何管理动态内存，以及如何使用 C++的语言特性来提高动态内存的管理效率，减少错误的发生。

5.1 operator new/delete

C++标准中 3.7.3 中规定，C++实现通过全局“allocation functions”new/new []和“deallocation functions”delete/delete[]来提供动态内存的访问和管理。operator new 可以是类的成员函数，或全局函数。一般来说，C++的运行库提供了默认的全局 new/new[]和 delete/delete[]的实现。程序自身也可以用自定义的实现来取代运行库提供的实现，但一个程序最多只能有一种自定义的实现。下面是 C++标准中定义的 new/new[]和 delete/delete[]的声明：

```
namespace std {
class bad_alloc;
struct nothrow_t {};
extern const nothrow_t nothrow;
typedef void (*new_handler)();
new_handler set_new_handler(new_handler new_p) throw();
}
void* operator new(std::size_t size) throw(std::bad_alloc);
void* operator new(std::size_t size, const std::nothrow_t&) throw();
void operator delete(void* ptr) throw();
void operator delete(void* ptr, const std::nothrow_t&) throw();
void* operator new[](std::size_t size) throw(std::bad_alloc);
void* operator new[](std::size_t size, const std::nothrow_t&) throw();
void operator delete[](void* ptr) throw();
void operator delete[](void* ptr, const std::nothrow_t&) throw();
void* operator new (std::size_t size, void* ptr) throw();
void* operator new[](std::size_t size, void* ptr) throw();
void operator delete (void* ptr, void*) throw();
void operator delete[](void* ptr, void*) throw();
```

众所周知，new 试图分配给定大小 size 的内存。如果成功，返回获得内存块的起始地址。这里需要指出的是，C++标准中没有规定是否要对获得的内存进行初始化。这意味着如果开发人员没有显式地为获得的内存赋初始值，内存中的初始值就完全决定于编译器的实现。

在 operator new 的定义中，可以看到如下 3 种形式：


```
void* operator new(std::size_t size) throw(std::bad_alloc);
void* operator new(std::size_t size, const std::nothrow_t&) throw();
void* operator new (std::size_t size, void* ptr) throw();
```

第 3 种形式的定义称之为“placement new”，它返回指定位置的内存作为分配的内存，本章在后面讨论这个问题。

而前两种形式的定义的主要区别在于内存分配失败时的处理，第 1 种抛出 std::bad_alloc()异常，这也是现在 C++标准要求的；第 2 种没有抛出异常，而是返回 0，这是为了兼容以前的 C++代码。不论是哪种情况，应用程序都要处理内存分配失败的情况。很多应用程序都没有处理内存分配失败的情况，但对于一个需要长期稳定运行的系统来说，这种处理是必不可少的。应用程序可以通过捕获 std::bad_alloc 异常或者返回值检查内存分配是否成功，而更好的方法是使用 C++中的 new_handler()函数。

C++标准中对内存分配失败有明确的规定，系统会调用当前安装的 new_handler()函数。这个错误处理函数是通过 set_new_handler()安装到系统上的，C++规定 new_handler 要执行下述操作中的一种。

（1）使 new 有更多的内存可用，然后返回。

（2）抛出一个 bad_alloc 或其派生的异常。

（3）调用 abort()或者 exit()退出。

下面通过一个例子，查看如何使用 new_handler 处理内存分配失败的情况：

```
#include <stdio.h>
#include <new>

using namespace std;

char *gPool = NULL;

void my_new_handler();

int main()
{
    set_new_handler(my_new_handler);
    gPool = new char[100*1024*1024];
    if (gPool!=NULL)
```

```
    {
        printf("Preserve 101MB memory at %x.\n", gPool);
    }

    char *p = NULL;
    for(int i=0; i<20; i++)
    {
        p = new char[100*1024*1024];
        printf("%d * 100M, p = %x\n", i+1, p);
    }

    printf("Done.\n");
    return 0;
}

void my_new_handler()
{
    if (gPool!=NULL)
    {
        printf("try to get more memory ...\n");
        delete[] gPool;
        gPool = NULL;
        return;
    }
    else
    {
        printf("I can not help ...\n");
        throw bad_alloc();
    }
    return;
}
```

在 32 位的 Windows 系统中，一个进程可以访问的内存空间是 4 GB，但可以用来动态分配的最大内存是 2 GB。在上面的例子中，开始首先分配了 100 MB 的空间。记录在 gPool 这个全局变量中，则剩下的可用空间为 19×100 MB。然后使用循环进行 20 次内存分配，应该在执行第 19 次和第 20 次内存分配时，都会造成失败。

程序通过 set_new_handler()设置了一个内存分配失败处理函数 my_new_handler()，当内存分配失败时会调用这个函数。在这个函数中检查 gPool 的值，如果 gPool 指向的空间还

存在，则释放这个空间，即可获得 100 MB 的额外空间，从而满足空间分配的需求；如果 gPool 指向的空间已经释放，则没有任何可用的缓冲区，抛出异常。

下面是程序的输出结果：

```
Preserve 101MB memory at 410020.
1 * 100M, p = 6820020
2 * 100M, p = cc30020
3 * 100M, p = 13040020
4 * 100M, p = 19450020
5 * 100M, p = 1f860020
6 * 100M, p = 25c70020
7 * 100M, p = 2c080020
8 * 100M, p = 32490020
9 * 100M, p = 388a0020
10 * 100M, p = 3ecb0020
11 * 100M, p = 450c0020
12 * 100M, p = 4b4d0020
13 * 100M, p = 518e0020
14 * 100M, p = 57cf0020
15 * 100M, p = 5e100020
16 * 100M, p = 64510020
17 * 100M, p = 6a920020
18 * 100M, p = 70d30020
try to get more memory ...
19 * 100M, p = 410020
I can not help ...

This application has requested the Runtime to terminate it in an unusual way.
Please contact the application's support team for more information.
```

由结果可以看出，当执行第 19 次内存分配时，由于内存不够，调用了 my_new_handler()。从而获得了新的可用内存，完成了分配。而当执行第 20 次内存分配时，gPool 已经被分配，因此再也没有可用内存。于是 my_new_handler()抛出了异常，导致程序退出。

前面提到 operator new 的第 3 种定义是 placement new，它最初的用途用来控制分配内存的位置。例如下面的例子：

```
#include <stdio.h>
#include <new>
```

```
using namespace std;
int main()
{
        char buffer[100];
        char * p = new(buffer) char[20];

        printf("buffer:\t%x\np:\t%x\n", buffer, p);
        return 0;
}
```

程序的输出为：

```
buffer: 12fe74
p:      12fe74
```

可见，p 获得的内存不是堆上的内存，而是事先在栈上分配的内存块 buffer。当然这块内存也就不需要通过 delete 来释放；除非是 placement new 不仅申请了内存，还创建了对象，才需要通过 delete 调用该对象的析构函数来销毁对象。

placement new 对于希望自己控制内存分配的程序来讲非常有用，很多的程序会申请一块很大的内存空间，然后所有的对象的都是创建在这个内存空间中。这样程序会自己控制内存的分配与整理等，从而提高效率，尤其是那些需要大量快速创建和销毁小对象的程序。

随着 C++的发展，在 operator new 中除了申请的内存的大小及位置。也可以加入其他的一些附加参数，但仍称之为“placement new”。应用程序可以通过重载 operator new 和 operator delete，添加一些自己的处理。例如，程序中可以在 debug 版本中定义下面的 operator new：

```
#include <stdio.h>
#include <new>

using namespace std;

void *operator new(std::size_t n, char* file, int line)
{
    printf("size: %d\nnew at %s, %d\n", n, file, line);
    return ::operator new(n);
}

void operator delete(void *p, char *file, int line)
```



```
{
      printf("delete at %s, %d\n", file, line);
      ::operator delete(p);
      return;
}

#define new new(__FILE__, __LINE__)
int main()
{
    char * p = new char[10];

    operator delete(p, __LINE__);
    return 0;
}
```

在调用 new 或者 delete 时，输出哪个文件的哪一行通过 new 申请了多大内存，在哪个文件的哪一行通过 delete 释放了内存。下面是程序的输出：

```
size: 10
new at D:\Temp\test11.cpp, 24
delete at D:\Temp\test11.cpp, 26
```

可以看出，如果不需要改变内存分配的方式，开发人员可以通过重载 replacement new 来增加一些参数。这样可以帮助开发人员分析程序内存的使用情况，从而做出相应的优化。而如果要改变实际内存分配的方式，或者不想使用系统提供的内存分配/释放的库，则需要实现自己定义的 operator new/delete。

5.2 自定义全局 operator new/delete

前面提到 operator new/delete 可以是类的成员函数，也可以是全局函数，本节和下一节中将讨论这两种类型的 operator new/delete 的作用及其实现。

当应用程序需要用统一的机制来控制数据的内存分配情况，并且不想使用系统提供的内存管理机制时，可以通过重写自己的全局 operator new/delete 来实现。一般对于一些对内存要求较高的应用程序可能会采用这种方式。此外，有时为了调试内存分配情况，也会实现全局的 operator new/delete，增加一些特殊处理，如产生 log 等来方便调试和排错。Scott

Meyers 在其名著《Effective C++》中指出，自己重写 operator new/delete 时，很重要的一点就是函数提供的行为要和系统默认的 operator new 一致。书中给出了全局 operator new/delete 的伪代码，以及一个用 malloc/free 来实现的 operator new/delete，如下面的代码所示。在此基础上，增加了一段测试代码，用来说明 C++处理自定义全局 operator new/delete 时的机制：

```
#include <stdio.h>
#include <new>
#include <stdlib.h>

using namespace std;
char *gPool = NULL;
void my_new_handler();

void *operator new(std::size_t size)
{
    void *pReturn = NULL;
    printf("my operator new\n");
    if (size == 0)
        size = 1;              // 至少分配 1 字节

    while (1) {
        pReturn = malloc(size);
        if (pReturn!=NULL)
        {
            printf("Got memeory\n");
            return (pReturn);
        }

        new_handler globalhandler = set_new_handler(0); // 找到新的 new_handler
        set_new_handler(globalhandler);

        if (globalhandler)
        {
            printf("call new_handler...\n");
            (*globalhandler)();      // 如果内存不足了，就调用 new_handler
        }
        else
        {
            printf("no memory, no new_handler\n");
```



```
            throw std::bad_alloc();
        }
    }
}

void operator delete(void *p)
{
    printf("my operator delete\n");
    return free(p);
}

int main()
{
    set_new_handler(my_new_handler);
    gPool = new char[100*1024*1024];
    if (gPool!=NULL)
    {
        printf("Preserve 101MB memory at %x.\n", gPool);
    }

    char *p = NULL;
    for(int i=0; i<20; i++)
    {
        p = new char[100*1024*1024];
        printf("%d * 100M, p = %x\n", i+1, p);
    }

    printf("Done.\n");
    return 0;
}

void my_new_handler()
{
    if (gPool!=NULL)
    {
        printf("try to get more memory ...\n");
        delete[] gPool;
        gPool = NULL;
        return;
    }
```

```
    else
    {
        printf("I can not help ...\n");
        throw bad_alloc();
    }
    return;
}
```

可以看到，在 operator new 中是一个无限循环；除非正常分配了内存或者是 new_handler 按照 C++标准规定执行了 3 种操作中的一种，或者是不存在 new_handler。下面是程序的输出：

```
my operator new
Got memeory
Preserve 101MB memory at 410020.
my operator new
Got memeory
1 * 100M, p = 6820020
my operator new
Got memeory
2 * 100M, p = cc30020
my operator new
Got memeory
3 * 100M, p = 13040020
my operator new
Got memeory
4 * 100M, p = 19450020
my operator new
Got memeory
5 * 100M, p = 1f860020
my operator new
Got memeory
6 * 100M, p = 25c70020
my operator new
Got memeory
7 * 100M, p = 2c080020
my operator new
Got memeory
8 * 100M, p = 32490020
my operator new
Got memeory
```



```
9 * 100M, p = 388a0020
my operator new
Got memeory
10 * 100M, p = 3ecb0020
my operator new
Got memeory
11 * 100M, p = 450c0020
my operator new
Got memeory
12 * 100M, p = 4b4d0020
my operator new
Got memeory
13 * 100M, p = 518e0020
my operator new
Got memeory
14 * 100M, p = 57cf0020
my operator new
Got memeory
15 * 100M, p = 5e100020
my operator new
Got memeory
16 * 100M, p = 64510020
my operator new
Got memeory
17 * 100M, p = 6a920020
my operator new
Got memeory
18 * 100M, p = 70d30020
my operator new
call new_handler...
try to get more memory ...
my operator delete
Got memeory
19 * 100M, p = 410020
my operator new
call new_handler...
I can not help ...

This application has requested the Runtime to terminate it in an unusual way.
Please contact the application's support team for more information.
```

在第 18 次内存分配时，出现了内存不足，这时按照 C++标准调用了已经安装 new_handler。之后，再次进入循环，试图重新分配内存并成功。而第 19 次内存分配时，调用 new_handler 之后没有获得更多内存。抛出了 bad_alloc()异常，从而导致程序退出。

如果在 main()函数中注释掉 set_new_handler(my_new_handler);，operator new 中会检查到没有 new_handler。所以在第 18 次内存分配失败后就输出 no memory, no new handler，直接抛出 bad_alloc()异常。程序的输出如下：

```
18 * 100M, p = 70d30020
my operator new
no memory, no new_handler

This application has requested the Runtime to terminate it in an unusual way.
Please contact the application's support team for more information.
```

5.3 自定义类 operator new/delete

当自定义全局的 operator new/delete 时，程序中的所有内存分配释放将使用统一的方式。然而在某些情况下，程序希望创建不同的类对象时使用不同的内存分配方式，尤其是在一些要求内存分配效率的程序中。为此，可以通过类成员函数形式的 operator new/delete 重载来为一些特定的类实现这个类自己的 operator new/delete，而其他则保持使用系统默认的 operator new/delete。

将前面一节中定义的全局的 operator new/delete 改为一个类的静态成员函数，即可实现自定义类的 operator new/delete。下面是一个简单的例子（为简单起见，例中没有实现自定义的 operator delete。但实际开发中，如果要实现自定义 operator new，最好也要实现自定义的 operator delete）：

```
#include <stdio.h>
#include <new>

using namespace std;

class simpleClass
{
public:
```



```
    int m_nValue;
    static void * operator new(size_t n);
    static void * operator new(std::size_t n, char* file, int line);
};

void *simpleClass::operator new(std::size_t n, char* file, int line)
{
    printf("size: %d\nnew at %s, %d\n", n, file, line);
    return ::operator new(n);
}

void * simpleClass::operator new(size_t n)
{
    printf("size: %d\n", n);
    return ::operator new(n);
}

class derivedClass : public simpleClass
{
public:
    int m_nDerivedValue;
};

int main()
{
    simpleClass *p1 = new simpleClass();
    derivedClass *p2 = new(__FILE__, __LINE__) derivedClass();
    char *buffer = new char[10];
}
```

在上面的例子中，为类 simpleClass 实现了自定义的 operator new 和 placement new。这样当通过 new simpleClass()来创建一个对象时，调用 simpleClass 类自己实现的 operator new，而不是全局的。由于继承的关系，所有由 simpleClass 派生出的派生类 derivedClass 也继承了父类 operator new。所以当通过 operator new 或者 placement new 来创建 derivedClass 的对象时，也会调用 simpleClass 类中实现的 operator new。从下面程序的输出结果中，可以很容易地看到这一点：

```
size: 4
size: 8
new at D:\Temp\test12.cpp, 35
```

当执行 simpleClass *p1 = new simpleClass();时，调用 simpleClass 的 operator new，输出要分配的内存空间的大小。当调用 derivedClass *p2 = new(_FILE_, _LINE_) derivedClass()时，调用的是基类的 placement new。输出不仅包括内存的大小，还有调用语句所在的源程序文件名和行号。当执行 char *buffer = new char[10]时，调用全局 operator new，因此没有任何输出。

在这里要注意的是，派生类会默认继承使用基类的自定义的 operator new。因此，如果基类自定义的 operator new 只是针对基类进行内存分配效率优化，并且不希望派生类使用基类的自定义 operator new，就需要在基类中进行处理。即判断这个分配内存的请求是基类对象的，还是派生类对象的。如果不是基类对象的，则交给全局的默认 operator new 处理即可。一个简单的方法就是判断传递给 operator new 的要申请内存的大小，所以前面例子中的代码可以修改如下：

```
void *simpleClass::operator new(std::size_t n, char* file, int line)
{
    if (n!=sizeof(simpleClass))
        return ::operator new(n);
    printf("size: %d\nnew at %s, %d\n", n, file, line);
    return ::operator new(n);
}

void * simpleClass::operator new(size_t n)
{
    if (n!=sizeof(simpleClass))
        return ::operator new(n);
    printf("size: %d\n", n);
    return ::operator new(n);
}
```

或者在派生类中重新定义派生类的 operator new，让派生类直接调用全局的 operator new，而不必修改基类的 opertator new 方法。这样，只有其他派生类还会使用基类的 operator new，而只有明确重载 operator new 的派生类才能调用全局默认的 operator new。修改后的代码如下：

```
void *simpleClass::operator new(std::size_t n, char* file, int line)
{
    printf("size = %d  \nnew at %s, %d\n", n, file, line);
    return ::operator new(n);
}

void * simpleClass::operator new(size_t n)
{
    printf("size: %d\n", n);
    return ::operator new(n);
}

class derivedClass : public simpleClass
{
public:
    int m_nDerivedValue;
    static void * derivedClass::operator new(size_t n);
};
void * derivedClass::operator new(size_t n)
{
    return ::operator new(n);
}
```


5.4　避免内存泄漏

一个复杂的 C++程序中最容易出现，也是最头疼的一个问题就是内存泄漏。即忘记释放申请的内存，造成程序占用的内存不断上升，系统性能会不断下降，甚至会内存耗尽而导致程序崩溃。以 Java 为代表的很多现代编程语言提供了自动的垃圾回收机制，即程序动态申请的内存不需要开发人员主动释放。如同任何自动变量一样，会由系统在合适时回收内存。虽然现在出现了很多 C++的库来支持自动的垃圾回收，但 C++语言一直没有把自动垃圾回收作为语言的内置机制，这个问题也是长期以来 C++社区争论不休的问题。

其实，虽然没有自动垃圾回收机制，C++语言还是提供了足够强大和灵活的机制，使得开发人员可以有效地避免内存泄漏。众所周知，通过 new 获得的内存或者创建的对象，一定要通过 delete 来释放，这样就不会有内存泄漏。可以想到，将这种分配和释放的过程封装到一个类中。即在构造时分配内存，在析构时释放内存，从而保证没有内存泄漏。下面是一个最简单的例子：

```
#include <stdio.h>
#include <string.h>

class simpleClass
{
private:
    char *m_buf;
    size_t m_nSize;
public:
    simpleClass(size_t n=1)  {
        m_buf = new char[n];
        m_nSize = n;
    };
    ~simpleClass() {
        printf("%d is deleted at %xd\n", m_nSize, m_buf);
        delete[] m_buf;
    };
    char *GetBuf() {
        return m_buf;
    };
};

void foo()
{
    simpleClass a(10);
    char *p = a.GetBuf();

    strcpy(p, "Hello");
    printf("%s\n", p);
}

int main()
{
    foo();
    printf("exit main() ...\n");
}
```

在这个例子中，对 char 类型的内存分配封装在类 simpleClass 中。通过声明一个 simpleClass 的对象，并给出所需内存的大小，即可获得相应的内存。而且这个内存在

simpleClass 对象的作用域退出时会自动释放，不需要开发人员显式调用 delete 来释放，也避免了开发人员忘记释放动态内存的错误。

在 foo()中获得相应的内存后，执行字符串拷贝操作，然后输出字符串的内容。当 foo()调用结束后，退出 simpleClass 对象的作用域。在销毁对象时调用析构函数，释放申请的动态内存。下面是程序的输出：

```
Hello
10 is deleted at 32c50d
exit main() ...
```

这个例子演示了最基本的思路，但是还存在很多的问题，显而易见的一个问题是拷贝构造函数和赋值的问题。将上面例子中的 foo()函数稍微修改，增加一个赋值语句：

```
void foo()
{
    simpleClass a(10);
    simpleClass b = a;
    char *p = a.GetBuf();

    strcpy(p, "Hello");
    printf("%s\n", p);
}
```

在实现 simpleClass 时，并没有实现拷贝构造函数。因此编译器会构造一个默认的拷贝构造函数，执行位拷贝(bit copy)操作，即将对象 a 的内容逐个字节的拷贝到对象 b 中，因此 a 中 m_buf 和拷贝后 b 中 m_buf 指向的是同一个地址的内存。当 a 和 b 都被销毁时，m_buf 指向的内存被销毁两次，从而造成错误。下面是程序的输出：

```
Hello
10 is deleted at 32c50d
10 is deleted at 32c50d
exit main() ...
```

一个简单的解决方法是禁止拷贝构造，在 simpleClass 的声明中将拷贝构造函数声明为私有函数。这样当 simpleClass 的使用者试图进行赋值或者作为参数传递给函数时，都会出现编译错误。但这样会造成很多的限制，有时不能达到程序的要求。这样就需要通过引用计数的方法，既避免了对同一块内存的多次删除，也允许拷贝构造函数。

所谓引用计数，就是对要使用的内存维护一个计数器，记录当前有多少指针指向这块内存。当有指针指向这块内存时，计数器加 1；反之，当指向这块内存的指针被销毁时，计数器减 1。当这块内存的计数器为 0 时，表示没有指针指向这块内存，这块内存才可以被释放。不难看出，类的构造和析构函数是执行计数器加 1 和减 1 操作的最合适之处。将 simpleClass 的实现修改如下（这段代码用来说明引用计数的基本原理，没有对异常等复杂情况进行讨论。）：

```
#include <stdio.h>
#include <string.h>

class simpleClass
{
private:
    char *m_buf;
    size_t m_nSize;
    int *m_count;
public:
    simpleClass(size_t n=1)  {
        m_buf = new char[n];
        m_nSize = n;

        m_count = new int;
        *m_count = 1;
        printf("count is: %d\n", *m_count);
    };
    simpleClass(const simpleClass& s) {
        m_nSize = s.m_nSize;
        m_buf = s.m_buf;
        m_count = s.m_count;

        (*m_count) ++;
        printf("count is: %d\n", *m_count);
    };
    ~simpleClass() {
        (*m_count) --;
        printf("count is: %d\n", *m_count);
        if (*m_count==0)
        {
            printf("%d is deleted at %xd\n", m_nSize, m_buf);
            delete[] m_buf;
            delete m_count;
        }
    };
    char *GetBuf() {
        return m_buf;
    };
};

void foo()
{
    simpleClass a(10);
    char *p = a.GetBuf();

    strcpy(p, "Hello");
    printf("%s\n", p);

    simpleClass b=a;
    printf("b=%s\n", b.GetBuf());
}

int main()
{
    foo();
    printf("exit main() ...\n");
}
```

在这个例子中，首先构造 simpleClass 的对象 a。然后通过拷贝构造创建对象 b，二者指向同一块内存。但由于使用了对象计数，因此在销毁对象 a 和 b 时，这块内存不会被释放两次。程序的输出如下：

```
count is: 1
Hello
count is: 2
b=Hello
count is: 1
count is: 0
10 is deleted at 32c50d
exit main() ...
```

从上面的输出可以看出引用计数器 count 的变化情况。当 a 被构造时，count 初始化为 1。当执行 b=a 时，count 增加为 2。a 和 b 指向的是同一块内存，存储的内容都是“Hello”。当退出 foo()时，b 首先被销毁，count 减 1。但仍大于 0，所以内存没有被释放。当 a 被销毁时，count 减为 0，因此释放对应的内存。

上面的程序还是有一点漏洞，例如将 foo()修改如下：

```
void foo()
{
    simpleClass a(10);
    char *p = a.GetBuf();

    strcpy(p, "Hello");
    printf("%s\n", p);

    simpleClass b=a;
    simpleClass c(20);
    c=a;
    printf("b=%s, c=%s\n", b.GetBuf(), c.GetBuf());
}
```

在通过拷贝构造创建 b 之后，声明了一个 c 对象，申请的内存大小是 20 个字节。然后把 a 赋值给 c，此时 c 应该指向 a 的内存。而 c 原来指向的内存则无指针指向，因此应该被释放。但上面的程序没有处理这种情况，从而造成了内存泄漏，解决的方法是重载 operator =。在类 simpleClass 的声明中添加 operator =如下：

```
simpleClass& operator=(const simpleClass& s)
{
    if (m_buf == s.m_buf)
        return *this;

    (*m_count)--;

    if (*m_count==0)
    {
        printf("%d is deleted at %xd\n", m_nSize, m_buf);
        delete[] m_buf;
        delete m_count;
    }
```



```
    // point to the new buffer.
    m_buf = s.m_buf;
    m_nSize = s.m_nSize;
    m_count = s.m_count;
    (*m_count)++;
} ;
```

在赋值前，首先将当前的计数器减 1。并检查是否可以释放，这样就避免了内存泄漏。程序的输出如下：

```
count is: 1
Hello
count is: 2
count is: 1
20 is deleted at 30f60d
b=Hello, c=Hello
count is: 2
count is: 1
count is: 0
10 is deleted at 32c50d
exit main() ...
```

可以看到，声明 c 时申请的 20 个字节的内存在将 a 赋值给 c 时自动被释放。

5.5 智能指针

上节介绍了通过对申请和释放内存的封装来避免内存泄漏的基本思路。事实上，早已出现了很多 C++的库，提供了称之为“智能指针”的模板类。使得开发人员可以方便地管理动态内存，而不必担心内存泄漏的问题。

智能指针就是存储指向动态内存/对象的指针的类，其使用和工作机制与 C++内置的指针非常相似，而最大的不同是会在适当的时间自动删除指向的内存或对象。此外，一般还提供 reset()方法，可以显式释放内存或销毁对象。它们使用 operator->和 operator*来生成原始指针，这样智能指针看上去就像一个普通指针。但它们要考虑很多的因素，例如所有权、线程安全，以及异常安全问题等。本节首先介绍基本的 std::auto_ptr，然后介绍常见的 Boost 中的几种智能指针。

auto_ptr 是 C++标准中提供的一种智能指针，首先看一个使用它的简单的例子：

```
#include <iostream>
#include <memory>
#include <string>

using namespace std;

class simple
{
private:
    size_t m_nSize;
public:
    simple()  {
        m_nSize = 10;
        cout << "simple created" << endl;
    };
    ~simple() {
        cout << "simple destroyed" << endl;
    };
    void foo() {
        cout << "m_nSize is: " << m_nSize << endl;
        cout << "foo() is called" << endl;
    };
};

int main()
{
    auto_ptr<string> pString(new string("Hello, auto ptr"));
    cout << *pString << endl;

    auto_ptr<simple> pSimple(new simple());
    pSimple->foo();
}
```

在这个例子中，首先通过 auto_ptr<string> pString(new string(“Hello, auto ptr”))声明了一个指向 string 的智能指针，又通过 auto_ptr<simple> pSimple(new simple())声明了一个指向自己类 simple 的对象的智能指针。由上节知道，实际上是创建了两个类的对象，这两个对象中分别存储指向 string 和 simple 的指针。当构造这两个对象时，对指针进行了初始化；

当销毁这个对象时，释放指针所指向的内存。由于这两个对象是一个栈上的对象，所以会自动销毁，因此就避免了内存泄漏的问题。下面是程序的输出，可以看到，动态创建的 simple 的对象被自动销毁（在本节后面的例子中将继续使用 simple 类作为例子，不再重复给出这个类的定义和实现；除非做了修改）：

```
Hello, auto ptr
simple created
m_nSize is: 10
foo() is called
simple destroyed
```

与 5.3 节中的内容有几点不同，一是 auto_ptr 的实现使用了模板。这是为了使其有普适性，可以用于各种数据类型或者用户自定义的类，而不必为每一种数据类型或者类重复实现一遍，这也是 C++中模板的典型用法；二是 auto_ptr 中封装的指针作为构造函数的参数传递进来，而不是在构造函数中申请内存或者创建对象并封装获得的指针。这也是为了适应不同的数据类型或者类，因为不同对象有不同的创建方法，因此很难在 auto_ptr 的构造函数中来创建。而一般是在 auto_ptr 之外创建好，然后将获得的指针作为参数传递给 auto_ptr 的构造函数；三是对 auto_ptr 的使用如同 C++内置指针，可以用 operator *和->等操作符，这是因为在 auto_ptr 的实现中重载了这些操作符。

上面是智能指针的一些基本要求。在实现智能指针时还要考虑很多问题，而所有权问题就是一个重要的方面。所谓所有权问题就是考虑智能指针在发生拷贝构造或者赋值时，需要对所封装的指针执行何种操作。即是完全复制一份，还是完全转移，或者二者共享同一份拷贝。智能指针有不同的语义，就会有不同的操作。对于 std::auto_ptr，将上面的例子简单地做如下修改：

```
int main()
{
    auto_ptr<string> pString(new string("Hello, auto ptr"));
   cout << *pString << endl;   // 正常输出
    auto_ptr<string> pString2(pString);
    cout << *pString << endl;    // 出错!

    auto_ptr<simple> pSimple(new simple());
    pSimple->foo();
}
```

在声明 pString 之后，通过复制 pString 声明 pString2。然后输出 pString 的内容，但运行的结果是程序崩溃。这是因为 std::auto_ptr 在所有权上的语义是完全转移，即发生拷贝构造或者赋值时，它所封装的指针的所有权也被转移到新的变量中。在上面的例子中，pString 将不再指向它原来的内存，所以发生了错误。而对 C++内置的指针进行赋值时，两个指针指向的将是同样的内存地址，所以就不会有这样的问题。

除了所有权问题，std::auto_ptr 还有一些其他限制。比较典型的如不能用于数组 STL 容器等。因此在 C++标准之外，有很多由 C++开发人员自行设计实现的智能指针，其中最著名应该是 Boost 智能指针(www.boost.org)。

Boost 中的智能指针有 3 类，即 scoped_ptr/scoped_arrary、shared_ptr/shared_arrary 和 weak_ptr。

scoped_ptr 是最基本的智能指针实现，只满足最基本的需求。即在 boost::scoped_ptr 析构时，会自动删除所管理的对象或内存，也可以通过显式调用 reset()方法来销毁对象释放内存。但与 std::auto_ptr 不同，boost:scoped_ptr 不会将所有权转移，即不会通过赋值或者拷贝构造将所指向的内存或对象转移到另外的 boost::scoped_ptr 中。通过 boost::scoped_ptr 的实现可以清楚地看到这一点：

```
template<class T> class scoped_ptr // noncopyable
{
private:

   T * ptr;

   scoped_ptr(scoped_ptr const &);
   scoped_ptr & operator=(scoped_ptr const &);

   typedef scoped_ptr<T> this_type;
……
}
```

boost::scoped_ptr 将拷贝构造函数和赋值 operator =()放在 private 中，这样保证了不会发生所有权问题。所以下面的程序中，当试图为一个 boost::scoped_ptr 赋值时，会出现编译错误，即 boost::scoped_ptr 代码中的 noncopyable 的意义：

```
#include <boost/scoped_ptr.hpp>
#include <boost/scoped_array.hpp>
```



```
int main()
{
    boost::scoped_ptr<simple> pSimple(new simple);
    boost::scoped_ptr<simple> pSimple2(pSimple); // 编译错误
    boost::scoped_ptr<simple> pSimple3 = pSimple; // 编译错误
}
```

提示：

使用 boost 库与使用其他非标准 C++一样，需要将头文件所在的目录添加到编译器或者集成开发环境的头文件搜索目录中。在使用命令行或者 makefile 进行编译时，一般通过编译器的-I 选项来指定头文件搜索路径。

与 std::auto_ptr 相同，boost::scoped_ptr 也不能用于数组和 C++标准容器库。当需要用于数组时，要用到 boost::scoped_array，它是 boost:scoped_ptr 的数组扩展形式。看下面的程序：

```
void boostArray()
{
    boost::scoped_array<simple> pSimple(new simple[2]);
}
void autoPtrArray()
{
    std::auto_ptr<simple> pSimple(new simple[2]);
}

int main()
{
    cout << "------------- calling boostArray() --------------" << endl;
    boostArray();
    cout << "------------- boostArray() finished -------------" << endl;

    cout << "------------- calling autoPtrArray() -------------" << endl;
    autoPtrArray();
    cout << "------------- autoPtrArray() finished -------------" << endl;
}
```

程序的输出如下：

```
------------- calling boostArray() --------------
simple created
```

```
simple created
simple destroyed
simple destroyed
------------- boostArray() finished -------------
------------- calling autoPtrArray() -------------
simple created
simple created
simple destroyed
------------- autoPtrArray() finished -------------
```

由程序的输出可以看出，当 std::atuo_ptr 用于数组时，只自动释放一个对象。数组中的其他对象没有自动释放，造成了内存泄漏。而 boost::scoped_array 没有这个问题，数组中的每一个对象都被正确地自动释放。

shared_ptr 则是 boost 提供的适用于“可拷贝”或“可赋值”的智能指针。与 scoped_ptr 不同，它可以执行拷贝构造或者赋值操作，即可以有多个 shared_ptr 指向同一个对象。当最后一个指向这个对象的 shared_ptr 被销毁或显式调用 reset()方法时，才会销毁这个对象，释放内存。shared_ptr 使用“引用计数”方法来记录有多少 shared_ptr 指向所管理的对象，在程序中可以通过调用 use_count()方法来检查当前指向所管理对象的 shared_ptr 的个数。下面是使用 shared_ptr 的简单的例子：

```
#include <iostream>
#include <boost/shared_ptr.hpp>

using namespace std;

class simple
{
private:
    char *m_buf;
    size_t m_nSize;
    int *m_count;
public:
    simple()  {
        m_nSize = 10;
        cout << "simple created" << endl;
    };
    ~simple() {
        cout << "simple destroyed" << endl;
```



```
    };
    void foo() {
        cout << "foo() is called, m_nSize is: " << m_nSize << endl;
        m_nSize++;
    };
};

void sharedPtr()
{
    boost::shared_ptr<simple> pSimple(new simple);
    boost::shared_ptr<simple> pSimple2(pSimple);
    boost::shared_ptr<simple> pSimple3 = pSimple;
    pSimple->foo();
    pSimple2->foo();
    pSimple3->foo();

    cout << "exit sharedPtr()" << endl;
}
int main()
{
    sharedPtr();
    cout << "exit main()" << endl;
}
```

在这个例子中，首先稍加修改前面的 simple 类中 foo()方法的实现，用来验证 boost::shared_pt 对所指向的对象的所有权的语义。在主程序中首先声明了一个指向 simple 对象的 boost::share_ptr，然后通过拷贝构造和赋值语句将另外两个 boost::shared_ptr 指针指向该对象（如果是使用 scoped_ptr，这两个语句不能通过编译）。当有新的 boost::shared_ptr 指向这个对象时，这些指针中的“引用计数”将会相应地变化。图 5-1 所示为这种变化。

由于 3 个指针实际上指向的都是同一个对象，所以调用 foo()方法时，改变的也是同一个 m_nSize 的值。下面是程序的输出：

```
simple created
foo() is called, m_nSize is: 10
foo() is called, m_nSize is: 11
foo() is called, m_nSize is: 12
exit sharedPtr()
simple destroyed
exit main()
```

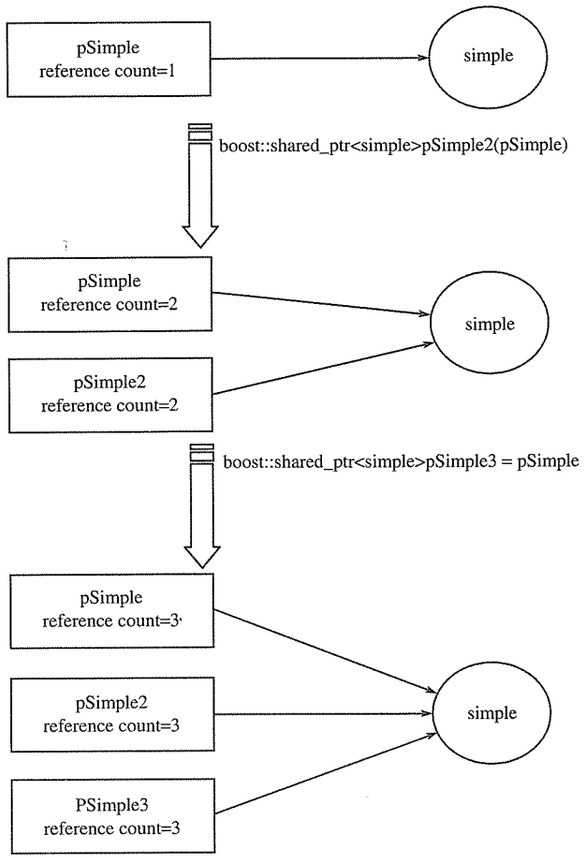


图 5-1　指针中的引用计数的变化

当程序退出 sharedPtr()函数时，3 个指针 pSimple、pSimple2 和 pSimple3 会自动销毁，而创建的 simple 的对象被自动销毁。事实上，simple 对象是在指向它的引用计数减到 0 时才自动销毁的。下面的例子通过 boost::shared_ptr 提供的 use_count()和 reset()方法来说明 simple 销毁的过程，use_count()方法可以输出当前的引用计数；reset()方法可以显式断开 boost::shared_ptr 与所指向的对象之间的联系，相当于销毁了这个指针。将前面的 sharedPtr()方法修改如下：

```
void sharedPtr()
{
    boost::shared_ptr<simple> pSimple(new simple);
    boost::shared_ptr<simple> pSimple2(pSimple);
    boost::shared_ptr<simple> pSimple3 = pSimple;
    pSimple->foo();
    pSimple2->foo();
    pSimple3->foo();

    cout << "use count is:" << pSimple.use_count() << endl;
    pSimple2.reset();
    cout << "use count is:" << pSimple.use_count() << endl;
    pSimple3.reset();
    cout << "use count is:" << pSimple.use_count() << endl;
    pSimple.reset();

    cout << "exit sharedPtr()" << endl;
}
```


在退出 sharedPtr()函数之前，分别对 pSimple2 和 pSimple3 调用 reset()方法，并输出当前引用计数的值。程序的输出如下：

```
simple created
foo() is called, m_nSize is: 10
foo() is called, m_nSize is: 11
foo() is called, m_nSize is: 12
use count is:3
use count is:2
use count is:1
simple destroyed
exit sharedPtr()
exit main()
```

可以看到，当调用一次 reset()方法后，引用计数的值减 1。当引用计数为 1 时，再调用 reset()方法，引用计数变为 0。此时所指向的对象被销毁，这样由程序的输出可以看出 simple 对象在 sharedPtr()退出之前被销毁。而与前面的例子不同，在退出 sharedPtr()之后才被销毁。图 5-2 所示为这个过程。

与 boost::scoped_ptr 相比，由于 boost::shared_ptr 可以拷贝构造并赋值，因此它可以作为 C++标准容器库中的元素。而当用于数组时，也需要用其数组扩展形式 boost::shared_array。这点与 boost::scoped_ptr 类似，本节不再给出具体的例子。

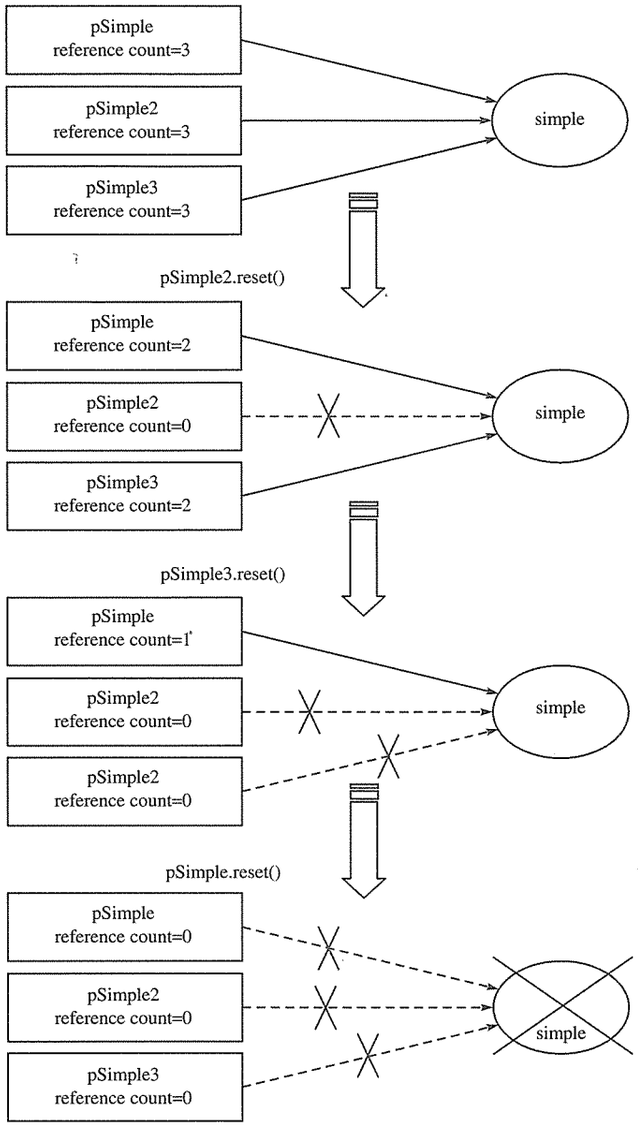


图 5-2　销毁对象的过程

boost::shared_ptr 采用引用计数来实现共享语义，因此不可避免地存在循环引用的问题。而打破循环引用的常规方法就是所谓“弱引用”，即在使用之前要检查是否可靠。boost 提

供的 boost::weak_ptr 用来解决这个问题。boost::weak_ptr 中存储的是指向一个已经由 boost::shared_ptr 所管理的对象的“弱引用”，其中没有重载 operator->。因此当需要访问该对象时，需要将 boost::weak_ptr 转换为 boost::shared_ptr 来进行。这种转换可以通过构造一个新的 boost::shared_ptr 或者调用 boost::weak_ptr 的 lock()方法来进行。如果此时该对象已经不存在，则这两种方法都将失败。lock()将会返回空，而 boost::shared_ptr 的构造函数将会抛出 boost::bad_weak_ptr 异常。下面是一个简单的例子（simple 类的声明和实现与前面的例子相同）：

```
#include <iostream>
#include <boost/smart_ptr.hpp>

using namespace std;

void testPtr()
{
    boost::shared_ptr<simple> pSimple(new simple);
    boost::weak_ptr<simple> pSimple2(pSimple);

    //pSimple.reset();

    if (pSimple2.lock())
        pSimple2.lock()->foo();
    else
        cout << "simple is gone!" << endl;

    try{
        boost::shared_ptr<simple> pSimple3(pSimple2);
        pSimple3->foo();
    } catch (boost::bad_weak_ptr)
    {
        cout << "simple is gone, can not construct shared_ptr!" << endl;
    }

    cout << "exit sharedPtr()" << endl;
}
int main()
{
```

```
    testPtr();
    cout << "exit main()" << endl;
}
```

在 testPtr()函数中，首先声明一个 boost::shared_ptr，然后声明一个指向同一个对象的 boost::weak_ptr。当使用这个 boost::weak_ptr 时，首先需要通过 lock()或者构造一个新的 boost::shared_ptr 来验证该对象是否还存在。当没有通过 pSimple.reset()销毁该对象时，这个 boost::weak_ptr 即可正常访问该对象，程序的输出如下：

```
simple created
foo() is called, m_nSize is: 10
foo() is called, m_nSize is: 11
exit sharedPtr()
simple destroyed
exit main()
```

而当修改 testPtr()函数，调用 pSimple.reset()来销毁 simple 对象。然后使用 boost::weak_ptr 时发生错误，程序的输出如下：

```
simple created
simple destroyed
simple is gone!
simple is gone, can not construct shared_ptr!
exit sharedPtr()
exit main()
```

可以看到，此时通过 boost::weak_ptr 对 simple 对象的访问没有成功。

除了 boost 库之外，C++社区还开发了很多的智能指针库，比较著名的如 Loki。它由 C++社区的大师 Andrei Alexandrescu 设计实现，采用的是基于策略的实现方式。其中包括 7 个方面的策略，即 Ownership Policies、Storage Policies、Conversion Policies、Checking Policies、StrongPtr Ownership Policies、Delete Policies 和 Reset Policies。

这些也是设计一种智能指针时需要考虑的因素，开发人员可以自己实现这 7 个方面的策略，这样就组合成了一种新类型的自定义的智能指针。Loki 本身也提供了几种常用策略的实现，如类似 boost 中的 scoped_ptr 及 shared_ptr 类型的策略等。而其强大在于开发人员可以通过实现自己的策略或者这 4 种策略的组合来不断扩充智能指针的类型，而避免不断地在库中增加智能指针的类型。

5.6 本章小结

本章介绍了 C++语言中动态内存管理的一些基本概念和方法，动态内存管理是 C++程序中的非常重要部分，也是影响程序性能的重要因素。为了有效地进行动态内存管理，将开发人员从内存泄漏的深渊中拯救出来，开发人员希望 C++也有类似其他高级程序设计语言的“垃圾回收”机制。为此，C++社区开发出了各种不同的“智能指针”库。智能指针是 C++社区所关心的热点问题之一，也是开发人员进行动态内存管理的双刃剑。正确地使用智能指针，可以减少开发人员的错误，提高开发效率；如果使用不正确，也将给程序带来灾难。这里的关键是正确地理解每种智能指针的语义，了解其优点和限制。然后结合程序的需求，选择合适的智能指针或者避免使用智能指针。

第6章 内存池

本章主要讨论如何通过自定义内存池（Memory Pool）来提高程序性能和内存利用效率。默认的内存管理函数是为“通用”目的设计的，在分配和释放内存时，为了应付不同的情况，需要做更多的判断和处理，从而增加了额外的开销。而自定义内存池是针对特定应用程序的内存管理方式，内存的分配释放性能会大大提升。

本章首先简单介绍自定义内存池性能优化的原理，然后列举软件开发中常用的内存池的不同类型，并给出具体实现的实例。

6.1 自定义内存池性能优化的原理

如前所述，读者已经了解到“堆”和“栈”的区别。而在编程实践中，不可避免地要大量用到堆上的内存。例如在程序中维护一个链表的数据结构时，每次新增或者删除一个链表的节点，都需要从内存堆上分配或者释放一定的内存；在维护一个动态数组时，如果动态数组的大小不能满足程序需要时，也要在内存堆上分配新的内存空间。

6.1.1 默认内存管理函数的不足

利用默认的内存管理函数 new/delete 或 malloc/free 在堆上分配和释放内存会有一些额外的开销。

系统在接收到分配一定大小内存的请求时，首先查找内部维护的内存空闲块表，并且需要根据一定的算法（例如分配最先找到的不小于申请大小的内存块给请求者，或者分配最适于申请大小的内存块，或者分配最大空闲的内存块等）找到合适大小的空闲内存块。如果该空闲内存块过大，还需要切割成已分配的部分和较小的空闲块。然后系统更新内存空闲块表，完成一次内存分配。类似地，在释放内存时，系统把释放的内存块重新加入到空闲内存块表中。如果有可能的话，可以把相邻的空闲块合并成较大的空闲块。

默认的内存管理函数还考虑到多线程的应用，需要在每次分配和释放内存时加锁，同样增加了开销。

可见，如果应用程序频繁地在堆上分配和释放内存，则会导致性能的损失。并且会使系统中出现大量的内存碎片，降低内存的利用率。

默认的分配和释放内存算法自然也考虑了性能，然而这些内存管理算法的通用版本为了应付更复杂、更广泛的情况，需要做更多的额外工作。而对于某一个具体的应用程序来说，适合自身特定的内存分配释放模式的自定义内存池则可以获得更好的性能。

6.1.2 内存池的定义和分类

自定义内存池的思想通过这个“池”字表露无疑，应用程序可以通过系统的内存分配

调用预先一次性申请适当大小的内存作为一个内存池，之后应用程序自己对内存的分配和释放则可以通过这个内存池来完成。只有当内存池大小需要动态扩展时，才需要再调用系统的内存分配函数，其他时间对内存的一切操作都在应用程序的掌控之中。

应用程序自定义的内存池根据不同的适用场景又有不同的类型。

从线程安全的角度来分，内存池可以分为单线程内存池和多线程内存池。单线程内存池整个生命周期只被一个线程使用，因而不需要考虑互斥访问的问题；多线程内存池有可能被多个线程共享，因此则需要在每次分配和释放内存时加锁。相对而言，单线程内存池性能更高，而多线程内存池适用范围更广。

从内存池可分配内存单元大小来分，可以分为固定内存池和可变内存池。所谓固定内存池是指应用程序每次从内存池中分配出来的内存单元大小事先已经确定，是固定不变的；而可变内存池则每次分配的内存单元大小可以按需变化，应用范围更广，而性能比固定内存池要低。

6.1.3 内存池工作原理示例

下面以固定内存池为例说明内存池的工作原理，如图 6-1 所示。

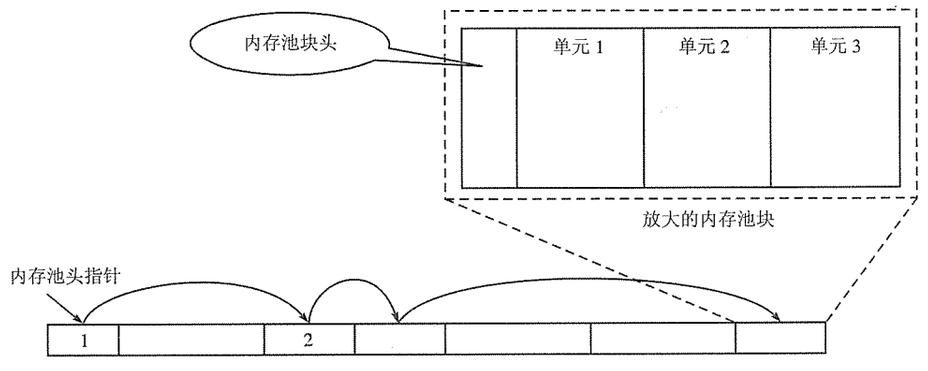


图 6-1 固定内存池

固定内存池由一系列固定大小的内存块组成，每一个内存块又包含了固定数量和大小的内存单元。

如图 6-1 所示，该内存池一共包含 4 个内存块。在内存池初次生成时，只向系统申请了一个内存块，返回的指针作为整个内存池的头指针。之后随着应用程序对内存的不断需求，内存池判断需要动态扩大时，才再次向系统申请新的内存块，并把所有这些内存块通过指针链接起来。对于操作系统来说，它已经为该应用程序分配了 4 个等大小的内存块。由于是大小固定的，所以分配的速度比较快；而对于应用程序来说，其内存池开辟了一定大小，内存池内部却还有剩余的空间。

例如放大来看第 4 个内存块，其中包含一部分内存池块头信息和 3 个大小相等的内存池单元。单元 1 和单元 3 是空闲的，单元 2 已经分配。当应用程序需要通过该内存池分配一个单元大小的内存时，只需要简单遍历所有的内存池块头信息，快速定位到还有空闲单元的那个内存池块。然后根据该块的块头信息直接定位到第 1 个空闲的单元地址，把这个地址返回，并且标记下一个空闲单元即可；当应用程序释放某一个内存池单元时，直接在对应的内存池块头信息中标记该内存单元为空闲单元即可。

可见与系统管理内存相比，内存池的操作非常迅速，它在性能优化方面的优点主要如下。

（1）针对特殊情况，例如需要频繁分配释放固定大小的内存对象时，不需要复杂的分配算法和多线程保护。也不需要维护内存空闲表的额外开销，从而获得较高的性能。

（2）由于开辟一定数量的连续内存空间作为内存池块，因而一定程度上提高了程序局部性，提升了程序性能。

（3）比较容易控制页边界对齐和内存字节对齐，没有内存碎片的问题。

6.2 一个内存池的实现实例

本节分析在某个大型应用程序实际应用到的一个内存池实现，并详细讲解其使用方法与工作原理。这是一个应用于单线程环境且分配单元大小固定的内存池，一般用来为执行时会动态频繁地创建且可能会被多次创建的类对象或者结构体分配内存。

本节首先讲解该内存池的数据结构声明及图示，接着描述其原理及行为特征。然后逐一讲解实现细节，最后介绍如何在实际程序中应用此内存池，并与使用普通内存函数申请内存的程序性能作比较。

6.2.1 内部构造

内存池类 MemoryPool 的声明如下：

```
class MemoryPool
{
private:
    MemoryBlock*   pBlock;
    USHORT         nUnitSize;
    USHORT         nInitSize;
    USHORT         nGrowSize;

public:
                   MemoryPool( USHORT nUnitSize,
                               USHORT nInitSize = 1024,
                               USHORT nGrowSize = 256 );
                   ~MemoryPool();

    void*          Alloc();
    void           Free( void* p );
};
```

MemoryBlock 为内存池中附着在真正用来为内存请求分配内存的内存块头部的结构体，它描述了与之联系的内存块的使用信息：

```
struct MemoryBlock
{
    USHORT         nSize;
    USHORT         nFree;
    USHORT         nFirst;
    USHORT         nDummyAlign1;
    MemoryBlock*   pNext;
    char           aData[1];

     static void* operator new(size_t, USHORT nTypes, USHORT nUnitSize)
     {
         return ::operator new(sizeof(MemoryBlock) + nTypes * nUnitSize);
     }
     static void  operator delete(void *p, size_t)
```

```
    {
        ::operator delete (p);
    }

    MemoryBlock (USHORT nTypes = 1, USHORT nUnitSize = 0);
    ~MemoryBlock() {}
};
```

此内存池的数据结构如图 6-2 所示。

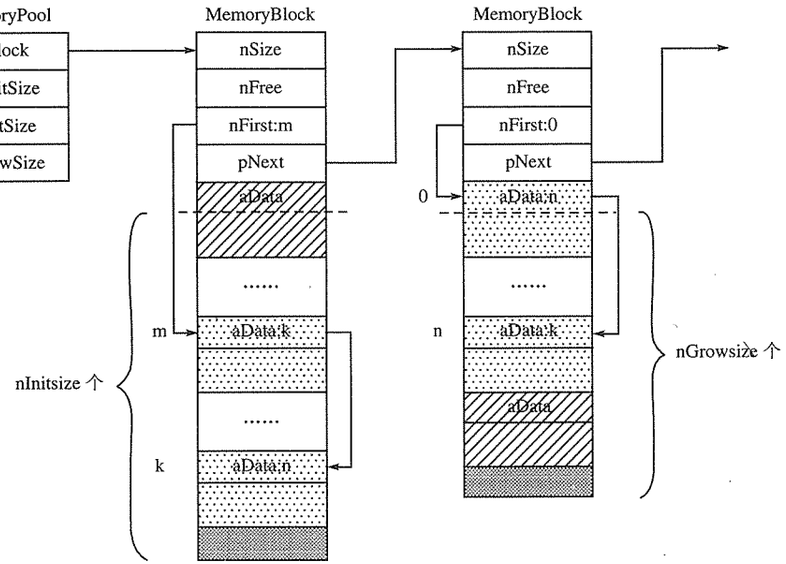


图 6-2　内存池的数据结构

6.2.2　总体机制

此内存池的总体机制如下。

（1）在运行过程中，MemoryPool 内存池可能会有多个用来满足内存申请请求的内存块，这些内存块是从进程堆中开辟的一个较大的连续内存区域，它由一个 MemoryBlock 结构体和多个可供分配的内存单元组成，所有内存块组成了一个内存块链表，MemoryPool 的 pBlock 是这个链表的头。对每个内存块，都可以通过其头部的 MemoryBlock 结构体的

pNext 成员访问紧跟在其后面的那个内存块。

（2）每个内存块由两部分组成，即一个 MemoryBlock 结构体和多个内存分配单元。这些内存分配单元大小固定（由 MemoryPool 的 nUnitSize 表示），MemoryBlock 结构体并不维护那些已经分配的单元的信息；相反，它只维护没有分配的自由分配单元的信息。它有两个成员比较重要：nFree 和 nFirst。nFree 记录这个内存块中还有多少个自由分配单元，而 nFirst 则记录下一个可供分配的单元的编号。每一个自由分配单元的头两个字节（即一个 USHORT 型值）记录了紧跟它之后的下一个自由分配单元的编号，这样，通过利用每个自由分配单元的头两个字节，一个 MemoryBlock 中的所有自由分配单元被链接起来。

（3）当有新的内存请求到来时，MemoryPool 会通过 pBlock 遍历 MemoryBlock 链表，直到找到某个 MemoryBlock 所在的内存块，其中还有自由分配单元（通过检测 MemoryBlock 结构体的 nFree 成员是否大于 0）。如果找到这样的内存块，取得其 MemoryBlock 的 nFirst 值（此为该内存块中第 1 个可供分配的自由单元的编号）。然后根据这个编号定位到该自由分配单元的起始位置（因为所有分配单元大小固定，因此每个分配单元的起始位置都可以通过编号×分配单元大小来偏移定位），这个位置就是用来满足此次内存申请请求的内存的起始地址。但在返回这个地址前，需要首先将该位置开始的头两个字节的值（这两个字节值记录其之后的下一个自由分配单元的编号）赋给本内存块的 MemoryBlock 的 nFirst 成员。这样下一次的请求就会用这个编号对应的内存单元来满足，同时将此内存块的 MemoryBlock 的 nFree 递减 1，然后才将刚才定位到的内存单元的起始位置作为此次内存请求的返回地址返回给调用者。

（4）如果从现有的内存块中找不到一个自由的内存分配单元（当第 1 次请求内存，以及现有的所有内存块中的所有内存分配单元都已经被分配时会发生这种情形），MemoryPool 就会从进程堆中申请一个内存块（这个内存块包括一个 MemoryBlock 结构体，及紧邻其后的多个内存分配单元，假设内存分配单元的个数为 n，n 可以取值 MemoryPool 中的 nInitSize 或者 nGrowSize），申请完后，并不会立刻将其中的一个分配单元分配出去，而是需要首先初始化这个内存块。初始化的操作包括设置 MemoryBlock 的 nSize 为所有内存分配单元的大小（注意，并不包括 MemoryBlock 结构体的大小）、nFree 为 n-1（注意，这里是 n-1 而不是 n，因为此次新内存块就是为了满足一次新的内存请求而申请的，马上就会分配一块自由存储单元出去，如果设为 n-1，分配一个自由存储单元后无须再将 n 递减 1），nFirst 为 1（已经知道 nFirst 为下一个可以分配的自由存储单元的编号。为 1 的原因与 nFree

为 *n*-1 相同，即立即会将编号为 0 的自由分配单元分配出去。现在设为 1，其后不用修改 nFirst 的值），MemoryBlock 的构造需要做更重要的事情，即将编号为 0 的分配单元之后的所有自由分配单元链接起来。如前所述，每个自由分配单元的头两个字节用来存储下一个自由分配单元的编号。另外，因为每个分配单元大小固定，所以可以通过其编号和单元大小（MemoryPool 的 nUnitSize 成员）的乘积作为偏移值进行定位。现在唯一的问题是定位从哪个地址开始？答案是 MemoryBlock 的 aData[1]成员开始。因为 aData[1]实际上是属于 MemoryBlock 结构体的（MemoryBlock 结构体的最后一个字节），所以实质上，MemoryBlock 结构体的最后一个字节也用做被分配出去的分配单元的一部分。因为整个内存块由 MemoryBlock 结构体和整数个分配单元组成，这意味着内存块的最后一个字节会被浪费，这个字节在图 6-2 中用位于两个内存的最后部分的浓黑背景的小块标识。确定了分配单元的起始位置后，将自由分配单元链接起来的工作就很容易了。即从 aData 位置开始，每隔 nUnitSize 大小取其头两个字节，记录其之后的自由分配单元的编号。因为刚开始所有分配单元都是自由的，所以这个编号就是自身编号加 1，即位置上紧跟其后的单元的编号。初始化后，将此内存块的第 1 个分配单元的起始地址返回，已经知道这个地址就是 aData。

（5）当某个被分配的单元因为 delete 需要回收时，该单元并不会返回给进程堆，而是返回给 MemoryPool。返回时，MemoryPool 能够知道该单元的起始地址。这时，MemoryPool 开始遍历其所维护的内存块链表，判断该单元的起始地址是否落在某个内存块的地址范围内。如果不在所有内存地址范围内，则这个被回收的单元不属于这个 MemoryPool；如果在某个内存块的地址范围内，那么它会将这个刚刚回收的分配单元加到这个内存块的 MemoryBlock 所维护的自由分配单元链表的头部，同时将其 nFree 值递增 1。回收后，考虑到资源的有效利用及后续操作的性能，内存池的操作会继续判断：如果此内存块的所有分配单元都是自由的，那么这个内存块就会从 MemoryPool 中被移出并作为一个整体返回给进程堆；如果该内存块中还有非自由分配单元，这时不能将此内存块返回给进程堆。但是因为刚刚有一个分配单元返回给了这个内存块，即这个内存块有自由分配单元可供下次分配，因此它会被移到 MemoryPool 维护的内存块的头部。这样下次的内存请求到来，MemoryPool 遍历其内存块链表以寻找自由分配单元时，第 1 次寻找就会找到这个内存块。因为这个内存块确实有自由分配单元，这样可以减少 MemoryPool 的遍历次数。

综上所述，每个内存池（MemoryPool）维护一个内存块链表（单链表），每个内存块由一个维护该内存块信息的块头结构（MemoryBlock）和多个分配单元组成，块头结构 MemoryBlock 则进一步维护一个该内存块的所有自由分配单元组成的“链表”。这个链表不

是通过“指向下一个自由分配单元的指针”链接起来的，而是通过“下一个自由分配单元的编号”链接起来，这个编号值存储在该自由分配单元的头两个字节中。另外，第 1 个自由分配单元的起始位置并不是 MemoryBlock 结构体“后面的”第 1 个地址位置，而是 MemoryBlock 结构体“内部”的最后一个字节 aData（也可能不是最后一个，因为考虑到字节对齐的问题），即分配单元实际上往前面错了一位。又因为 MemoryBlock 结构体后面的空间刚好是分配单元的整数倍，这样依次错位下去，内存块的最后一个字节实际没有被利用。这么做的一个原因也是考虑到不同平台的移植问题，因为不同平台的对齐方式可能不尽相同。即当申请 MemoryBlock 大小内存时，可能会返回比其所有成员大小总和还要大一些的内存。最后的几个字节是为了“补齐”，而使得 aData 成为第 1 个分配单元的起始位置，这样在对齐方式不同的各种平台上都可以工作。

6.2.3 细节剖析

有了上述的总体印象后，本节来仔细剖析其实现细节。

（1）MemoryPool 的构造如下：

```
MemoryPool::MemoryPool( USHORT _nUnitSize,
                        USHORT _nInitSize, USHORT _nGrowSize )
{
    pBlock     = NULL;                                          ①
    nInitSize  = _nInitSize;                                    ②
    nGrowSize  = _nGrowSize;                                    ③

    if ( _nUnitSize > 4 )
        nUnitSize = (_nUnitSize + (MEMPOOL_ALIGNMENT-1)) & ~(MEMPOOL_ALIGNMENT-1);
                                                                ④
    else if ( _nUnitSize <= 2 )
        nUnitSize = 2;                                          ⑤
    else
        nUnitSize = 4;
}
```

从①处可以看出，MemoryPool 创建时，并没有立刻创建真正用来满足内存申请的内存块，即内存块链表刚开始时为空。

②处和③处分别设置“第 1 次创建的内存块所包含的分配单元的个数”，及“随后创建

的内存块所包含的分配单元的个数”，这两个值在 MemoryPool 创建时通过参数指定，其后在该 MemoryPool 对象生命周期中一直不变。

后面的代码用来设置 nUnitSize，这个值参考传入的_nUnitSize 参数。但是还需要考虑两个因素。如前所述，每个分配单元在自由状态时，其头两个字节用来存放“其下一个自由分配单元的编号”。即每个分配单元“最少”有“两个字节”，这就是⑤处赋值的原因。④处是将大于 4 个字节的大小_nUnitSize 往上“取整到”大于_nUnitSize 的最小的 MEMPOOL_ALIGNMENT 的倍数（前提是 MEMPOOL_ALIGNMENT 为 2 的倍数）。如_nUnitSize 为 11 时，MEMPOOL_ALIGNMENT 为 8，nUnitSize 为 16；MEMPOOL_ALIGNMENT 为 4，nUnitSize 为 12；MEMPOOL_ALIGNMENT 为 2，nUnitSize 为 12，依次类推。

（2）当向 MemoryPool 提出内存请求时：

```
void* MemoryPool::Alloc()
{
    if ( !pBlock )                                          ①
    {
            ……
    }

    MemoryBlock* pMyBlock = pBlock;
    while (pMyBlock && !pMyBlock->nFree )                   ②
        pMyBlock = pMyBlock->pNext;

    if ( pMyBlock )                                         ③
    {
        char* pFree = pMyBlock->aData+(pMyBlock->nFirst*nUnitSize);
        pMyBlock->nFirst = *((USHORT*)pFree);

        pMyBlock->nFree--;
        return (void*)pFree;
    }
    else                                                    ④
    {
        if ( !nGrowSize )
            return NULL;

         pMyBlock = new(nGrowSize, nUnitSize) FixedMemBlock(nGrowSize,
nUnitSize);
```



```
        if ( !pMyBlock )
            return NULL;

        pMyBlock->pNext = pBlock;
        pBlock = pMyBlock;

        return (void*)(pMyBlock->aData);
    }

}
```

MemoryPool 满足内存请求的步骤主要由四步组成。

①处首先判断内存池当前内存块链表是否为空，如果为空，则意味着这是第 1 次内存申请请求。这时，从进程堆中申请一个分配单元个数为 nInitSize 的内存块，并初始化该内存块（主要初始化 MemoryBlock 结构体成员，以及创建初始的自由分配单元链表，下面会详细分析其代码）。如果该内存块申请成功，并初始化完毕，返回第 1 个分配单元给调用函数。第 1 个分配单元以 MemoryBlock 结构体内的最后一个字节为起始地址。

②处的作用是当内存池中已有内存块（即内存块链表不为空）时遍历该内存块链表，寻找还有“自由分配单元”的内存块。

③处检查如果找到还有自由分配单元的内存块，则“定位”到该内存块现在可以用的自由分配单元处。“定位”以 MemoryBlock 结构体内的最后一个字节位置 aData 为起始位置，以 MemoryPool 的 nUnitSize 为步长来进行。找到后，需要修改 MemoryBlock 的 nFree 信息（剩下来的自由分配单元比原来减少了一个），以及修改此内存块的自由存储单元链表的信息。在找到的内存块中，pMyBlock->nFirst 为该内存块中自由存储单元链表的表头，其下一个自由存储单元的编号存放在 pMyBlock->nFirst 指示的自由存储单元（亦即刚才定位到的自由存储单元）的头两个字节。通过刚才定位到的位置，取其头两个字节的值，赋给 pMyBlock->nFirst，这就是此内存块的自由存储单元链表的新的表头，即下一次分配出去的自由分配单元的编号（如果 nFree 大于零的话）。修改维护信息后，就可以将刚才定位到的自由分配单元的地址返回给此次申请的调用函数。注意，因为这个分配单元已经被分配，而内存块无须维护已分配的分配单元，因此该分配单元的头两个字节的信息已经没有用处。换个角度看，这个自由分配单元返回给调用函数后，调用函数如何处置这块内存，内存池无从知晓，也无须知晓。此分配单元在返回给调用函数时，其内容对于调用函数来说是无意义的。因此几乎可以肯定调用函数在用这个单元的内存时会覆盖其原来的内容，

即头两个字节的内容也会被抹去。因此每个存储单元并没有因为需要链接而引入多余的维护信息，而是直接利用单元内的头两个字节，当其分配后，头两个字节也可以被调用函数利用。而在自由状态时，则用来存放维护信息，即下一个自由分配单元的编号，这是一个有效利用内存的好例子。

④处表示在②处遍历时，没有找到还有自由分配单元的内存块，这时，需要重新向进程堆申请一个内存块。因为不是第一次申请内存块，所以申请的内存块包含的分配单元个数为 nGrowSize，而不再是 nInitSize。与①处相同，先做这个新申请内存块的初始化工作，然后将此内存块插入 MemoryPool 的内存块链表的头部，再将此内存块的第 1 个分配单元返回给调用函数。将此新内存块插入内存块链表的头部的原因是该内存块还有很多可供分配的自由分配单元（除非 nGrowSize 等于 1，这应该不太可能。因为内存池的含义就是一次性地从进程堆中申请一大块内存，以供后续的多次申请），放在头部可以使得在下次收到内存申请时，减少②处对内存块的遍历时间。

可以用图 6-2 的 MemoryPool 来展示 MemoryPool::Alloc 的过程。图 6-3 是某个时刻 MemoryPool 的内部状态。

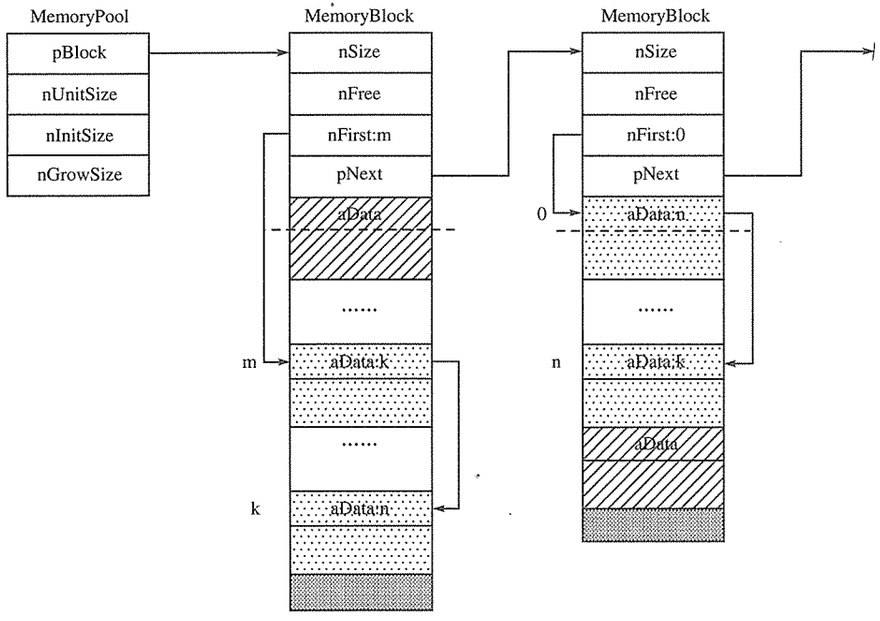


图 6-3　某个时刻 MemoryPool 的内部状态

因为 MemoryPool 的内存块链表不为空，因此会遍历其内存块链表。又因为第 1 个内存块里有自由的分配单元，所以会从第 1 个内存块中分配。检查 nFirst，其值为 *m*，这时 pBlock->aData+(pBlock->nFirst*nUnitSize)定位到编号为 *m* 的自由分配单元的起始位置（用 pFree 表示）。在返回 pFree 之前，需要修改此内存块的维护信息。首先将 nFree 递减 1，然后取得 pFree 处开始的头两个字节的值（需要说明的是，这里 aData 处值为 *k*。其实不是这一个字节。而是以 aData 和紧跟其后的另外一个字节合在一起构成的一个 USHORT 的值，不可误会）。发现为 *k*，这时修改 pBlock 的 nFirst 为 *k*。然后，返回 pFree。此时 MemoryPool 的结构如图 6-4 所示。

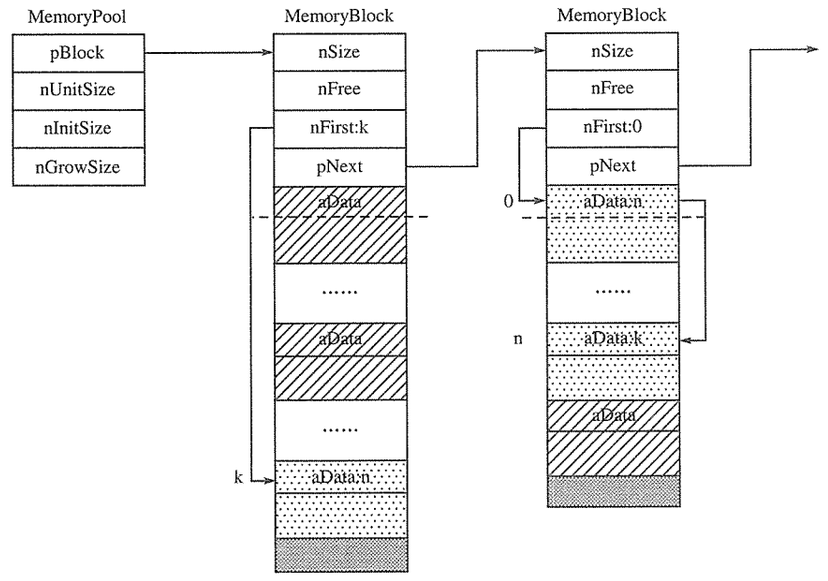


图 6-4　MemoryPool 的结构

可以看到，原来的第 1 个可供分配的单元（m 编号处）已经显示为被分配的状态。而 pBlock 的 nFirst 已经指向原来 *m* 单元下一个自由分配单元的编号，即 *k*。

（3）MemoryPool 回收内存时：

```
void MemoryPool::Free( void* pFree )
{
    ......
```

```
        MemoryBlock* pMyBlock = pBlock;

        while ( ((ULONG)pMyBlock->aData > (ULONG)pFree) ||
            ((ULONG)pFree >= ((ULONG)pMyBlock->aData + pMyBlock->nSize)) )      ①
        {
            ......
        }

        pMyBlock->nFree++;                                                      ②
        *((USHORT*)pFree) = pMyBlock->nFirst;                                   ③
        pMyBlock->nFirst = (USHORT)(((ULONG)pFree-(ULONG)(pBlock->aData)) /
nUnitSize);                                                                     ④

        if (pMyBlock->nFree*nUnitSize == pMyBlock->nSize )                      ⑤
        {
            ......
        }
        else
        {
            ......
        }
    }
```

如前所述，回收分配单元时，可能会将整个内存块返回给进程堆，也可能将被回收分配单元所属的内存块移至内存池的内存块链表的头部。这两个操作都需要修改链表结构。这时需要知道该内存块在链表中前一个位置的内存块。

①处遍历内存池的内存块链表，确定该待回收分配单元（pFree）落在哪一个内存块的指针范围内，通过比较指针值来确定。

运行到②处，pMyBlock 即找到的包含 pFree 所指向的待回收分配单元的内存块（当然，这时应该还需要检查 pMyBlock 为 NULL 时的情形，即 pFree 不属于此内存池的范围，因此不能返回给此内存池，读者可以自行加上）。这时将 pMyBlock 的 nFree 递增 1，表示此内存块的自由分配单元多了一个。

③处用来修改该内存块的自由分配单元链表的信息，它将这个待回收分配单元的头两个字节的值指向该内存块原来的第一个可分配的自由分配单元的编号。

④处将 pMyBlock 的 nFirst 值改变为指向这个待回收分配单元的编号，其编号通过计算

此单元的起始位置相对 pMyBlock 的 aData 位置的差值，然后除以步长（nUnitSize）得到。

实质上，③和④两步的作用就是将此待回收分配单元“真正回收”。值得注意的是，这两步实际上是使得此回收单元成为此内存块的下一个可分配的自由分配单元，即将它放在了自由分配单元链表的头部。注意，其内存地址并没有发生改变。实际上，一个分配单元的内存地址无论是在分配后，还是处于自由状态时，一直都不会变化。变化的只是其状态（已分配/自由），以及当其处于自由状态时在自由分配单元链表中的位置。

⑤处检查当回收完毕后，包含此回收单元的内存块的所有单元是否都处于自由状态，且此内存是否处于内存块链表的头部。如果是，将此内存块整个的返回给进程堆，同时修改内存块链表结构。

注意，这里在判断一个内存块的所有单元是否都处于自由状态时，并没有遍历其所有单元，而是判断 nFree 乘以 nUnitSize 是否等于 nSize。nSize 是内存块中所有分配单元的大小，而不包括头部 MemoryBlock 结构体的大小。这里可以看到其用意，即用来快速检查某个内存块中所有分配单元是否全部处于自由状态。因为只需结合 nFree 和 nUnitSize 来计算得出结论，而无须遍历和计算所有自由状态的分配单元的个数。

另外还需注意的是，这里并不能比较 nFree 与 nInitSize 或 nGrowSize 的大小来判断某个内存块中所有分配单元都为自由状态，这是因为第 1 次分配的内存块（分配单元个数为 nInitSize）可能被移到链表的后面，甚至可能在移到链表后面后，因为某个时间其所有单元都处于自由状态而被整个返回给进程堆。即在回收分配单元时，无法判定某个内存块中的分配单元个数到底是 nInitSize 还是 nGrowSize，也就无法通过比较 nFree 与 nInitSize 或 nGrowSize 的大小来判断一个内存块的所有分配单元是否都为自由状态。

以上面分配后的内存池状态作为例子，假设这时第 2 个内存块中的最后一个单元需要回收（已被分配，假设其编号为 *m*，pFree 指针指向它），如图 6-5 所示。

不难发现，这时 nFirst 的值由原来的 0 变为 *m*。即此内存块下一个被分配的单元是 *m* 编号的单元，而不是 0 编号的单元（最先分配的是最新回收的单元，从这一点看，这个过程与栈的原理类似，即先进后出。只不过这里的“进”意味着“回收”，而“出”则意味着“分配”）。相应地，*m* 的“下一个自由单元”标记为 0，即内存块原来的“下一个将被分配出去的单元”，这也表明最近回收的分配单元被插到了内存块的“自由分配单元链表”的头部。当然，nFree 递增 1。

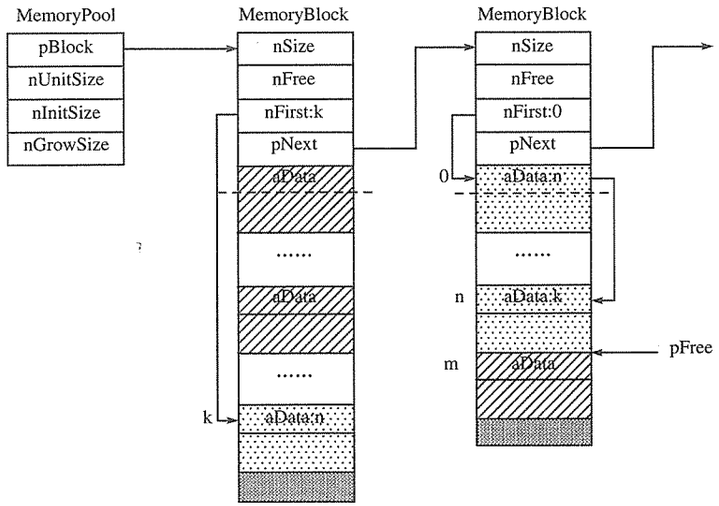


图 6-5　分配后的内存池状态

处理至⑥处之前，其状态如图 6-6 所示。

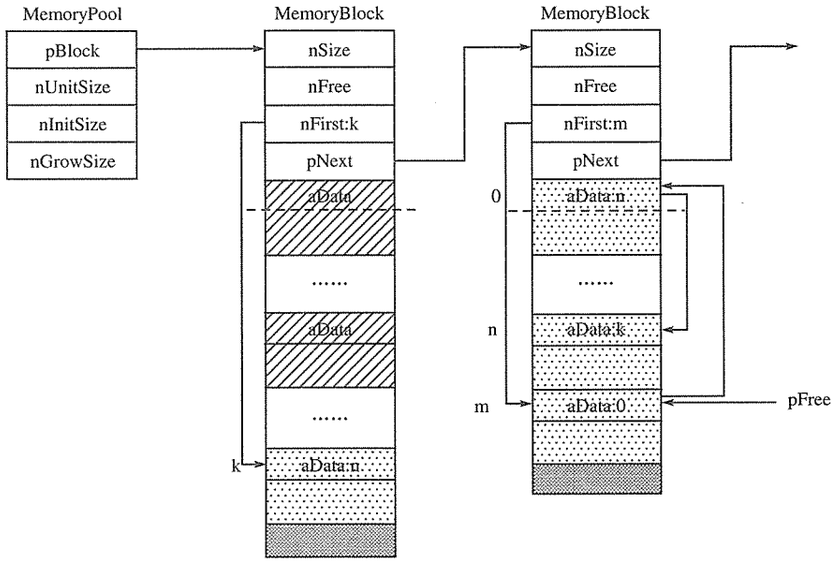


图 6-6　处理至⑥处之前的内存池状态

这里需要注意的是，虽然 pFree 被“回收”，但是 pFree 仍然指向 *m* 编号的单元，这个单元在回收过程中，其头两个字节被覆写，但其他部分的内容并没有改变。而且从整个进程的内存使用角度来看，这个 *m* 编号的单元的状态仍然是“有效的”。因为这里的“回收”只是回收给了内存池，而并没有回收给进程堆，因此程序仍然可以通过 pFree 访问此单元。但是这是一个很危险的操作，因为首先该单元在回收过程中头两个字节已被覆写，并且该单元可能很快就会被内存池重新分配。因此回收后通过 pFree 指针对这个单元的访问都是错误的，读操作会读到错误的数据，写操作则可能会破坏程序中其他地方的数据，因此需要格外小心。

接着，需要判断该内存块的内部使用情况，及其在内存块链表中的位置。如果该内存块中省略号“……”所表示的其他部分中还有被分配的单元，即 nFree 乘以 nUnitSize 不等于 nSize。因为此内存块不在链表头，因此还需要将其移到链表头部，如图 6-7 所示。

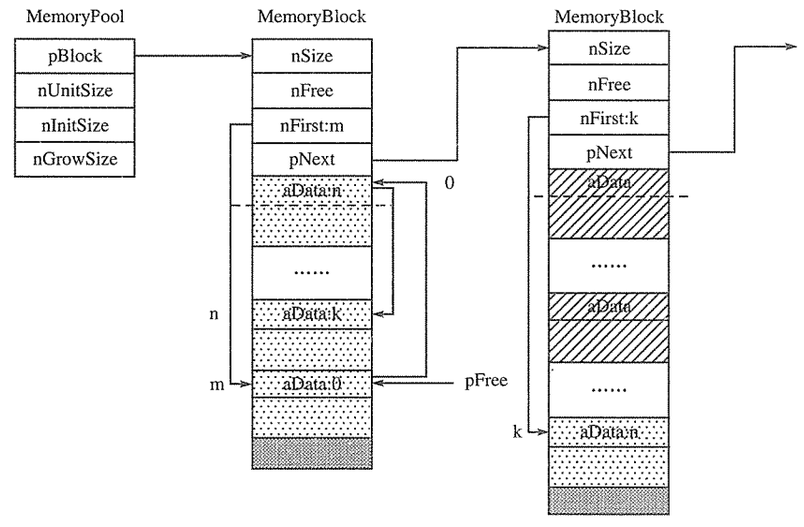


图 6-7　因回收引起的 MemoryBlock 移动

如果该内存块中省略号“……”表示的其他部分中全部都是自由分配单元，即 nFree 乘以 nUnitSize 等于 nSize。因为此内存块不在链表头，所以此时需要将此内存块整个回收给进程堆，回收后内存池的结构如图 6-8 所示。

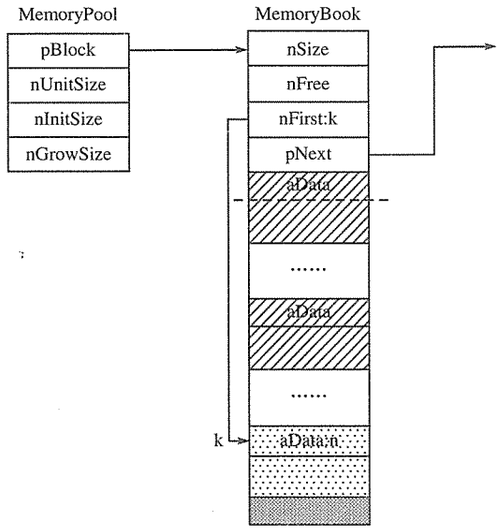


图 6-8　回收后内存池的结构

一个内存块在申请后会初始化，主要是为了建立最初的自由分配单元链表，下面是其详细代码：

```
MemoryBlock::MemoryBlock (USHORT nTypes, USHORT nUnitSize)
    : nSize  (nTypes * nUnitSize),
      nFree  (nTypes - 1),                                    ④
      nFirst (1),                                             ⑤
      pNext  (0)
{
        char * pData = aData;                                 ①
        for (USHORT i = 1; i < nTypes; i++)                   ②
        {
            *reinterpret_cast<USHORT*>(pData) = i;            ③
            pData += nUnitSize;
        }
}
```

这里可以看到，①处 pData 的初值是 aData，即 0 编号单元。但是②处的循环中 i 却是从 1 开始，然后在循环内部的③处将 pData 的头两个字节值置为 i。即 0 号单元的头两个字节值为 1，1 号单元的头两个字节值为 2，一直到（nTypes-2）号单元的头两个字节值为

（nTypes-1）。这意味着内存块初始时，其自由分配单元链表是从 0 号开始。依次串联，一直到倒数第 2 个单元指向最后一个单元。

还需要注意的是，在其初始化列表中，nFree 初始化为 nTypes-1（而不是 nTypes），nFirst 初始化为 1（而不是 0）。这是因为第 1 个单元，即 0 编号单元构造完毕后，立刻会被分配。另外注意到最后一个单元初始并没有设置头两个字节的值，因为该单元初始在本内存块中并没有下一个自由分配单元。但是从上面例子中可以看到，当最后一个单元被分配并回收后，其头两个字节会被设置。

图 6-9 所示为一个内存块初始化后的状态。

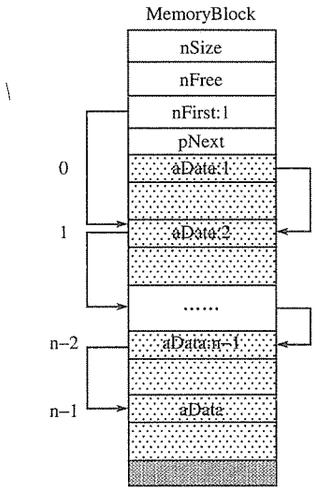


图 6-9　一个内存块初始化后的状态

当内存池析构时，需要将内存池的所有内存块返回给进程堆：

```
MemoryPool::~MemoryPool()
{
    MemoryBlock* pMyBlock = pBlock;
    while ( pMyBlock )
    {
        .....
    }
}
```

6.2.4 使用方法

分析内存池的内部原理后，本节说明如何使用它。从上面的分析可以看到，该内存池主要有两个对外接口函数，即 Alloc 和 Free。Alloc 返回所申请的分配单元（固定大小内存），Free 则回收传入的指针代表的分配单元的内存给内存池。分配的信息则通过 MemoryPool 的构造函数指定，包括分配单元大小、内存池第 1 次申请的内存块中所含分配单元的个数，以及内存池后续申请的内存块所含分配单元的个数等。

综上所述，当需要提高某些关键类对象的申请 / 回收效率时，可以考虑将该类所有生成对象所需的空间都从某个这样的内存池中开辟。在销毁对象时，只需要返回给该内存池。“一个类的所有对象都分配在同一个内存池对象中”这一需求很自然的设计方法就是为这样的类声明一个静态内存池对象，同时为了让其所有对象都从这个内存池中开辟内存，而不是缺省的从进程堆中获得，需要为该类重载一个 new 运算符。因为相应地，回收也是面向内存池，而不是进程的缺省堆，还需要重载一个 delete 运算符。在 new 运算符中用内存池的 Alloc 函数满足所有该类对象的内存请求，而销毁某对象则可以通过在 delete 运算符中调用内存池的 Free 完成。

6.2.5 性能比较

为了测试利用内存池后的效果，通过一个很小的测试程序可以发现采用内存池机制后耗时为 297 ms。而没有采用内存池机制则耗时 625 ms，速度提高了 52.48%。速度提高的原因可以归结为几点，其一，除了偶尔的内存申请和销毁会导致从进程堆中分配和销毁内存块外，绝大多数的内存申请和销毁都由内存池在已经申请到的内存块中进行，而没有直接与进程堆打交道，而直接与进程堆打交道是很耗时的操作；其二，这是单线程环境的内存池，可以看到内存池的 Alloc 和 Free 操作中并没有加线程保护措施。因此如果类 A 用到该内存池，则所有类 A 对象的创建和销毁都必须发生在同一个线程中。但如果类 A 用到内存池，类 B 也用到内存池，那么类 A 的使用线程可以不必与类 B 的使用线程是同一个线程。

另外，在第 1 章中已经讨论过，因为内存池技术使得同类型的对象分布在相邻的内存区域，而程序会经常对同一类型的对象进行遍历操作。因此在程序运行过程中发生的缺页应该会相应少一些，但这个一般只能在真实的复杂应用环境中进行验证。

6.3 本章小结

内存的申请和释放对一个应用程序的整体性能影响极大，甚至在很多时候成为某个应用程序的瓶颈。消除内存申请和释放引起的瓶颈的方法往往是针对内存使用的实际情况提供一个合适的内存池。内存池之所以能够提高性能，主要是因为它能够利用应用程序的实际内存使用场景中的某些“特性”。比如某些内存申请与释放肯定发生在一个线程中，某种类型的对象生成和销毁与应用程序中的其他类型对象要频繁得多，等等。针对这些特性，可以为这些特殊的内存使用场景提供量身定做的内存池。这样能够消除系统提供的缺省内存机制中，对于该实际应用场景中的不必要的操作，从而提升应用程序的整体性能。

第 3 篇　应用程序启动性能优化

应用程序的启动性能是应用程序给最终用户的第一印象，是衡量一个应用程序易用性的重要指标。因而其优化是程序性能优化的重要组成部分。

第 7 章详细介绍应用程序的物理布局，即存在于硬盘上的可执行程序格式和动态链接库格式及其被操作系统加载到内存中的布局。通过该章，读者可以深入地了解应用程序可执行文件和动态链接库的知识，为应用程序的启动优化做好准备。

第 8 章介绍在 Win32 和 Linux 平台上应用程序从编译链接到加载启动的过程。引入了“冷启动”和“热启动”性能概念的定义，并分别探讨了影响冷启动性能和热启动性能的相关因素。通过阅读该章，读者对应用程序的启动过程会有详细的了解，并且从原理上理解不同因素对于程序启动性能的影响。

第 9 章具体介绍程序启动性能优化的一般步骤，然后总结获得精确而稳定的程序启动性能的测试方法，最后以主要篇幅列举了优化程序启动性能的方法。

第 7 章　动态链接与动态库

在计算机发展的早期，一个软件通常就是一个单一的可执行程序，而当前的大型软件一般都是由一个可执行程序和多个动态库组成的。动态库的出现给软件的开发、重用和维护带来很多好处。但是同时也增加了一些复杂性，它使使用者更难理解程序的启动过程，并且动态库的引入使得软件在运行过程中可能会出现一些让使用者感到迷惑的情况。本章的主要目的是通过解析动态链接和动态库的一些细节，让读者对动态库的工作原理有一些较为深刻的认识。具备了这些知识，就可以通过优化动态库来提高应用程序的启动性能。第 9 章详细介绍将如何通过优化动态库来提高应用程序的启动性能。

本章首先介绍了传统的静态链接技术及相应产生的静态链接库，详细描述了链接过程的两个重要任务：符号解析和重定位。接着介绍了动态链接的优势，并详细地介绍了Windows 和 Linux 系统中动态库的两种实现：Windows 系统的动态链接库（DLL）和 Linux 系统的动态共享对象（DSO）。在介绍这两种动态链接方案时，不仅介绍了如何创建和使用这两种动态库，而且详细描述了这两种动态库中与动态链接密切相关的数据结构，以及这些数据结构在动态链接时的作用，同时指出创建与使用这两种动态库时应注意的方面。

7.1 链接技术的发展

链接主要是把分散的数据和代码收集并合成一个单一的可加载并可执行的文件。链接可以发生在代码静态编译、程序被加载时，或者应用程序执行时。在早期的计算机系统中，人们用手工的方法链接。后来，链接由称为“链接器”的程序自动执行。链接技术在 20 世纪 70 年代已经成熟，随着动态库的出现，又出现了动态链接技术。本节首先简要介绍在目前的环境下一个可执行程序的创建及使用过程中所涉及到的编译、链接和加载 3 个阶段，然后详细地介绍链接的过程。

7.1.1 编译、链接和加载

计算机刚诞生时，人们用机器语言来编写计算机程序，创建、修改和维护程序是非常困难的。高级语言的出现使人们可以很容易编写计算机程序，并创造出复杂且功能强大的计算机软件。但是高级语言不能直接被计算机接受，它必须被编译为机器语言才能被计算机使用。编译器是把高级语言编译为指定机器语言的工具，它以高级语言编写的程序源代码为输入，产生一个包含机器代码及相关信息（包括符号表和重定位信息等）的目标文件。

随着计算机应用的发展，人们开发出大量的软件，用来处理各种问题，很多软件都是以库的形式存在的。当开发新的软件时，可以链接这些已存在的库从而获得相应的功能。链接能是把分散的数据和代码收集并合成一个单一的可加载并可执行的文件。链接使分离编译成为可能。如果没有链接，你只能编写一个巨大的单一的源代码。任何修改都会导致对该源代码的重新编译，这不仅非常浪费时间，而且巨大的单一源代码也不易维护。同时，链接实现了可执行代码级的重用。链接通常由链接器来处理，它以目标文件，库文件作为

输入，产生可执行文件及相关信息。

在操作系统出现前，每个程序都拥有计算机全部内存的支配权。所以程序可以以固定的内存地址编译和链接，认为计算机的所有内存地址都可以访问。但是在操作系统出现后，程序必须和操作系统，甚至可能还必须和其他程序一起共享计算机的内存。这意味着在操作系统把程序装载到内存之前，不能确定程序运行的真实地址，因此把程序从文件读入到内存后可能还需要执行一些额外的操作。加载器可以把二进制文件读入到内存的某个绝对地址，并且对程序进行相应的修改。随着虚拟内存技术的广泛使用，每个运行时的应用程序又可以获得整个地址空间的支配权。因此编译生成一个可执行程序时，其默认的加载地址可以在编译链接时确定。但是动态库可以被加载到不同进程中的不同地址的特性，又使得加载器必须在加载动态库时执行一些额外的工作。加载器操作的对象是可执行程序，可能还包括一些动态共享库。

下面以构建并执行一个 Linux 系统下简单的可执行程序来展示上述 3 个过程。用以下两个文件编译成一个可执行的 myExe，并在 Linux 控制台上执行：

```
//source file a.c
#include <stdio.h>
extern int max(int, int)

int main()
{
  int n1 =10;
  int n2 =20;
  int m = max(n1, n2);
  printf("max= %d\n", m);
  return 0;
}

//source file b.c
int max(int a, int b)
{
  if (a>b) return a;
  else
     return b;
}
```

使用 GCC 如下命令可以直接编译为可执行程序 myExe：

```
gcc  -o myExe  a.c b.c
```

在 GCC 编译器内部，首先由 C 预处理器（cpp）将 a.c 翻译为一个 ASCII 中间文件 a.i：

```
cpp [other command line options]  a.c  /tmp/a.i;
```

并由 C 编译器（ccl）将中间文件 a.i 翻译为一个 ASCII 汇编语言文件 a.s：

```
ccl [other command line options]  /tmp/a.i  -o /tmp/a.s
```

然后由汇编器（as）将 a.s 翻译为一个可重定位的目标文件 a.o：

```
as [other command line options]  /tmp/a.s  -o /tmp/a.o
```

GCC 编译器对源文件 b.c 也采用相同的步骤，把 b.o 编译为可重定位的目标文件 b.o：

最后，它调用链接器（ld）将 a.o、b.o，及一些必要的系统目标文件组合起来，创建一个可执行的目标文件 myExe：

```
ld [other command line options]  /tmp/a.o /tmp/b.o  -o myExe
```

得到可执行目标文件 myExe 后，在 shell 命令行上输入其名称：

```
./myExe
```

shell 调用加载器。Linux 上加载器是一个称为“execve”的程序，它把可执行程序的代码和数据拷贝到内存中，然后把控制交给程序。

与链接器和加载器相比，编译器的功能比较容易区分。它翻译源代码文件，产生二进制的目标代码文件。随着动态库的出现，链接器和加载器的功能在某些方面有重合，它们的主要功能包括以下几个方面。

（1）程序装载：把程序从磁盘拷贝到内存中以准备运行。某些情况下装载仅包含把数据从磁盘拷贝到内存中，其他情况则可能包含分配存储空间、设置内存保护位，或者把虚拟地址映射到磁盘页面中等。

（2）重定位：编译器产生目标文件中数据和代码的地址一般都是从 0 开始，但很少有操作系统会把代码加载到 0 地址并且执行。更为重要的是，如果每个目标文件的数据和代码地址都从 0 开始。当一个程序包含多个目标文件时，就会产生地址重叠。重定位就是为输出的目标文件中的每一部分指定一个加载地址，并且修改相应的代码和数据以反映这种变化。

（3）符号解析：由多个子程序构建一个可执行程序时，子程序之间的相互引用通过符号进行。程序也可以通过符号来引用代码库中的功能，符号解析就是将符号引用和符号定义关联起来。

尽管链接器和装载器的功能有许多重叠，但是还是有相当理由把进行程序加载的部分定义为加载器，把完成符号处理的部分定义为链接器。它们中的任何一个都能执行做重定位工作，并且已经有了能够完成所有 3 个功能的链接装载器。

7.1.2　静态链接与静态链接库

1. 静态链接与静态链接库

静态链接一般以可重定位的目标文件，包含可重定位目标文件的静态库文件，以及动态库或动态库的桩文件（stub）作为输入，产生完全链接的可执行文件。该可执行文件能被操作系统加载并执行。图 7-1 所示为一个静态链接过程的示意。

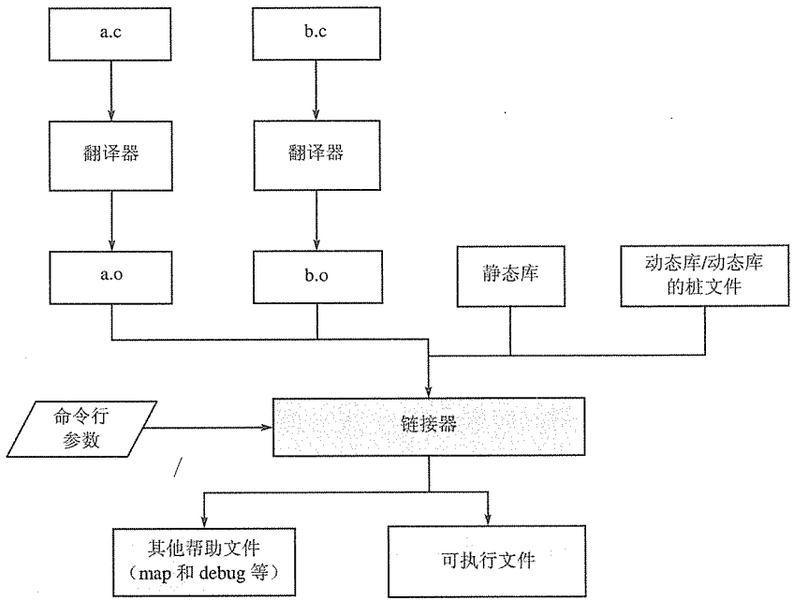


图 7-1　一个静态链接过程的示意

可重定位的目标文件一般由多个节（section）组成，有的节中包含指令，有的节中包含初始过的全局变量，有的节中包含未初始过的全局变量。静态链接把所有输入的可重定位的目标文件及静态库中相关的目标文件链接为一个整体。链接的过程就是把分布在各个可重定位的目标文件中相应的节合并起来，同时完成符号解析和重定位。

静态库是一个文件，其中包含一些相关的目标模块（即可重定位的目标文件）。如果没有静态库，当人们需要重用已有的以目标代码形式存放的软件资产时，就会很困难。以重用标准 C 函数来说，如果每个标准的 C 函数都被保存成一个单独并可重定位的目标文件，那么使用者必须知道需要使用的每个目标文件的位置（该目标文件含有一个标准 C 函数）。即使这些目标文件放置在同一个目录下，使用者也必须很清楚地了解这些标准函数之间的依赖关系，并且把所有相关的目标文件都输入到链接命令中，这样即耗时又容易出错。如果把所有的标准 C 函数编译到一个目标文件中，链接命令将会很简单。但是产生的包含全部标准 C 函数的目标文件将会十分庞大，并且这个巨大的目标文件也会被完全链接到最终的可执行程序中。最终的可执行程序也会因为包含了许多无用的代码而变得很大，以至浪费了存储空间。静态库就是为了解决上述问题而提出的。其实，静态库就是一种打包机制。编译系统（或者相关系统工具）可以把相关的目标模块打包成一个以某种格式存放的文件，静态库可以作为链接器的输入。如果程序需要引用静态库提供的某个函数，链接时只需要在命令行中输入该静态库。链接器将只拷贝被程序引用的目标模块，以及该目标模块所依赖的那些目标模块。这样既方便了使用，又不会把无用的代码复制到最终的可执行程序中。

不同的系统的静态库的格式不全相同，在 UNIX 系统中，静态库以一种称为“存档”（archive）的特殊文件格式存放在磁盘上。存档文件包括一个头部，它描述了每个成员目标文件的大小和位置，以及一组可重定位目标文件。开始时，存档文件不包含符号信息。后来的新版本都包含了符号信息以方便链接器查询，符号信息一般包括符号的名称，以及该符号所在目标文件在存档文件中的偏移量。COFF 和 ELF 静态库中的符号信息以如下格式来描述：

```
int nsymbols; //静态库中包含的符号数量
int member[]; //成员在存档文件中的偏移量
char strings[];//null 结尾的字符串集合
```

符号信息的最初 4 个字节指明了静态库中包含的符号数量，其后是一个“位置”数组，数组中的每个元素是一个目标文件在存档文件中的偏移量。最后是一个以 null 结尾的字符串集合，其中字符串表中第 1 个字符串所代表的符号是在“位置”数组中第 1 个元素所指

定的目标文件中定义。依次类推，每个字符串（符号）与每个“位置”数组元素（目标文件）形成对应关系。这样，链接器可以很方便地定位到引用符号所在的目标文件，并把该目标文件链接到最终的可执行程序中。

2. 符号解析

符号解析指把目标文件中每一个外部符号引用与该符号的唯一定义关联起来。软件一般由多个源代码文件组成的，由于分离编译的实现，每个源代码文件可以被单独地编译为目标文件。并且一个源代码文件可以引用外部的数据和函数，这种引用体现在目标文件中就是符号引用。符号可以分为本地链接符号和全局链接符号，本地链接符号指该符号的可见范围限制在符号定义的模块中；全局链接符号除了能够被定义该符号的模块使用外，还可以被其他模块使用。对于 C 语言来说，C 的源代码扮演着模块的角色。声明带有 static 属性的全局变量或者函数是本地链接符号，不带有 static 属性的全局变量或函数都是全局链接符号。

对于引用本地链接符号，编译器只允许每个本地符号有一个定义，因此在编译时即可确定本地链接符号的引用。对于全局链接符号，又可以分为引用本模块定义的全局链接符号和引用其他模块定义的全局链接符号。当编译器遇到一个没有在本模块中定义的全局链接符号时，会生成一个链接器符号表表项，由链接器来处理。如果链接器不能从输入模块中找到相应的符号定义，就会报告一个链接错误而终止；如果链接器从输入模块中发现相应符号在多处被定义，则报告一个链接错误而终止，或者取某个定义而舍弃其他定义。

不同的链接器会采用不同的规则来处理多次被定义的全局符号。UNIX 链接器根据编译器提供的全局符号的强弱属性来判断，强符号指函数和已初始化的全局变量；弱符号指未初始化的全局变量。如果链接器发现同样的一个符号在多处被定义为强符号，它就会报告一个链接错误而终止；如果发现同样的一个符号在多处被定义为一个强符号和多个弱符号，则就会选择强符号作为该符号的定义；如果发现同一个符号在多处被定义为多个弱符号，则会任选一个弱符号作为该符号的定义。

符号解析从直观上看比较简单，但是由于不同情况的出现，使得它常常使人感到迷惑。

（1）区分与静态库链接和与动态库链接。

程序（一些目标文件）可以与静态库进行静态链接，也可以与动态库或者动态库的桩文件进行静态链接。程序与静态库链接时，链接器将会拷贝静态库中被程序引用的目标模

块，以及该目标模块所依赖的那些目标模块到最终的可执行文件中。如果程序中的某个全局符号与静态库中被拷贝的目标模块中的某个全局符号相同，就会产生冲突，如下面的例子所示：

```
//main.c
#include <stdio.h>
extern int getFromLibA();
void printme(){
  printf("Print in main\n");
}

int main()
{
  int num = getFromLibA();
  printme();
  return 0;
}

//a.c packaged in myLibA.a
void printme(){
  printf("Print in a.c\n");
}

int getFromLibA(){
  return 0;
}
```

main.c 中定义了一个全局的强符号 printme，并引用了一个外部全局符号 getFromLibA。如果该外部全局符号来自于一个静态库 myLibA.a 中的目标模块 a，并且在该目标模块中同样定义了一个全局的强符号 printme，则程序与静态库 myLibA.a 链接时，这两个 printme 会产生冲突，链接器将报告一个链接错误。

但是如果把目标模块 a 放置到一个动态库中，产生的动态库也会包含这个全局的强符号 printme。当程序与动态库进行静态链接时，链接器不会报告全局符号多处定义的链接错误。首先，动态库已经是一个完全链接的二进制文件，不能从中分离出构成动态库的各个目标文件；其次，程序与动态库进行静态链接时，链接器不会把相应的代码拷贝到最终的可执行文件中。它只是告诉可执行程序需要依赖哪些动态库，或者指出可执行程序中的外

部全局符号来自于哪个动态库。本章后面两节将详细介绍动态库与动态链接。

（2）与静态库链接。

静态库中目标模块含有的例程引用同一个库中的其他目标模块的例程，或者引用其他静态库中目标模块的例程的现象是很普遍的。因此，链接一个静态库，对静态库中的符号解析是一个反复的过程。UNIX 链接器在链接过程中会维护 3 个集合，一是最终要被合并成可执行文件的可重定位目标文件的集合 E；二是未解析符号的集合 U；三是在前面输入文件中已定义的符号集合 D。当链接一个静态库时，链接器会顺序地扫描静态库中的符号表。如果发现静态库中的某个目标模块定义了一个符号，并且这个符号可以解析未解析符号集合 U 中的某个符号时，链接器就把该目标模块加入到可重定位目标文件的集合 E 中，同时修改集合 U 和 D 以反映这个目标模块中引用和定义的符号。接着链接器重新扫描静态库的符号表，寻找可以解析的符号，如果找到相应的符号定义，则重复上述过程。然后再次扫描符号表，直到扫描完一遍，没有新的目标模块被加入到可重定位目标文件的集合 E 中，才完成对该静态库的链接。

UNIX 链接器和大多数 Windows 链接器的命令行支持目标文件和静态库混写（即多个目标文件中间可以插入一个静态库，多个静态库中间可以插入一个目标文件）。虽然这给使用者带来了方便，但是必须非常注意库和目标文件的顺序。如果在命令行中定义一个符号的库出现在引用这个符号的目标文件之前，那么引用就不能被解析，链接会失败。同时，库与库之间也有依赖关系，被依赖的库必须放在引用该库符号的库的后面。按照依赖关系，在链接的命令行中，一般先输入目标文件，接着是应用程序相关的特殊库，后面是一些常用功能库（如网络库、输入 / 输出库及图像处理库等），最后是标准的系统库。

如果库之间发生循环依赖，则需要在命令行中重复输入静态库。如 mymain.c 调用 libx.a 中的函数 X，该函数又调用了 liby.a 中的函数 Y。而 liby.a 中的函数 Y 又调用了 libx.a 中的其他函数，那么 libx.a 就必须在命令行中重复出现：

```
gcc mymain.c libx.a liby.a libx.a
```

如果情况更为复杂，甚至会出现：

```
gcc mymain.c lib1.a lib2.a lib3.a lib4.a lib1.a lib2.a lib3.a lib4.a
```

等各种复杂的循环依赖。

3. 重定位

可重定位的目标文件一般由多个节（section）组成，静态链接需要把分布在各个可重定位的目标文件中的相应的节合并起来。对于目标文件中的代码来说，编译器在分别编译每个目标文件时产生的代码都是从 0 地址开始的。当多个代码节合成为一个代码段时，需要根据其在最终代码段的位置做相应的调整。同时，链接器也需要对已经解析的符号分配运行时地址。这些工作就是重定位，它发生在符号解析后。在进行符号解析时，链接器会读入每个可重定位的目标文件，把每个符号引用和符号定义关联起来。当重定位时，链接器已经知道其输入目标模块中的每个节（如代码节和数据节）的确切大小。重定位可以分为如下两个步骤。

（1）重定位节和符号定义：链接器将所有相同类型的节合并成为同一类型的新的聚合节。例如，来自所有输入模块的代码节将合并成最终可执行文件的代码节。然后链接器为新产生的聚合节赋予运行时的地址，为输入模块的每个节赋予运行时的地址，为输入模块中的每个符号赋予运行时的地址。当这一步完成后，程序的每一条指令和每一个全局变量的运行时地址都已经确定。

（2）重定位与节地址相关的指令和重定位节中的符号引用：链接器需要修改指令中使用的绝对或相对地址，并把符号引用替换为该符号的运行时地址。例如，指令中用绝对地址来访问全局数据（或者跳转到绝对的程序地址）。如果该全局数据（跳转的目标指令）和访问它的指令本来就在同一个目标文件中，链接器会根据这个全局数据（目标指令）所在节的运行时地址做相应的修改。修改时，一般是在原来的地址上加上节的运行时地址。这是因为，编译器生成的指令一般以 0 地址开始（即数据 / 指令在相应节中的偏移量）。如果该全局数据和访问它的指令不在同一个目标文件中，就会有一个符号引用，链接器会根据该符号的运行地址来作相应的修改。

重定位与节地址相关的指令和重定位节中的符号引用，这两种重定位之间的界限可以是模糊的。处理与节地址相关的代码重定位的方法之一是为程序的各个节的基地址指定一个符号，然后把与节地址相关的重定位看成对基地址符号引用的重定位。

链接器可以完成重定位的功能，是因为可重定位的目标文件中有一种称为“重定位表项”（relocation entry）的数据结构，这些表项放置在一起构成了可重定位目标文件的重定位节。当汇编器生成目标文件时，它并不知道数据和代码最终的运行时地址，也不知道引

用的外部符号的最终运行地址。一旦遇到最终位置未知的情况时，它就会生成一个重定位表项，告诉链接器在生成可执行程序时如何修改。Linux ELF 文件重定位表项的数据结构如下所示：

```
typedef struct {
    int offset;       //需要修改的引用在节中的偏移
    int symbol:24,    //需要修改的引用所关联的符号
        type:8;       //重定位的类型
} Elf32_Rel;
```

offset 表示需要修改的引用在节中的偏移，symbol 表示需要修改的引用所关联的符号，type 表示重定位的类型。链接器从 offset 中找到需要修改之处，从 symbol 中找到符号的运行时地址（如果重定位与符号无关，则 symbol 为 STN_UNDEF，其值为 0），并根据重定位的类型进行修改。由于 offset 代表的值是节中的偏移，因此不同的节有与之对应的重定位节。在可重定位的 ELF 文件中，与代码节对应的重定位节是.rel.text，与数据节对应的重定位节是.rel.data。ELF 定义了 11 种不同的重定位类型，其中 R_386_PC32 和 R_386_32 是在静态链接时较常见的类型。R_386_PC32 指重定位一个使用 32 位 PC 相关的地址引用，R_386_32 指重定位一个使用 32 位绝对地址的引用。

除了静态链接时需要进行重定位外，加载时也可能需要进行重定位，特别是在程序被加载的地址与其默认的首地址不一致时。随着虚拟存储技术的广泛使用，每个进程都有独立的地址空间，链接产生的可执行程序一定可以被加载到其默认的首地址。因此现代的操作系统在加载可执行程序时，不需要进行代码重定位。但是随着动态库的出现，动态库的加载又需要进行重定位，参见本章的后两节。

总之，静态链接过程是一个两遍扫描处理的过程。它必须首先扫描输入的文件以确定各个节的长度，并收集所有符号的定义和引用之处。在这个过程中，链接器会创建一个节表列出输入文件中定义的所有节和一个符号表。根据第 1 遍输出的数据，链接器可以确定各节的运行时地址。进一步可以得到每条指令和每个全局数据的运行时地址，并且为符号指定运行时地址。第 2 遍使用在第 1 遍中收集的信息来控制真正的链接过程，它读取并重定位目标代码，把符号的引用替代为运行时地址。调整代码和数据中的地址以反映重定位后的节地址，并把重定位后的代码写到输出文件中。最后产生相关辅助信息并输出文件。

7.1.3 动态链接与动态库

静态链接很容易实现和使用，但是它有一些不是之处。

首先，如果构造一个可执行程序需要静态链接一些静态库，当静态库版本升级时必须重新链接这些新的静态库来升级该可执行程序。

其次，通过静态链接产生可执行程序时，该可执行程序所依赖的静态库中的所有数据和代码都会被复制到可执行程序中，使得产生的可执行程序的体积比较庞大。更为浪费的是，某些常用的代码（比如某些 C 的标准函数）几乎会被每个可执行程序所包含。这不仅会浪费磁盘空间，而且当这些可执行程序被加载到内存执行时，会浪费宝贵的内存资源。

动态库（或称为“共享库”）就是为解决上述问题而产生的。它实现了代码共享。所有引用该动态库的可执行目标文件共享一份相同的代码和数据，而自己不必拥有它，即不必像静态库中的代码和数据那样被复制和嵌入到引用它们的可执行文件中。而且在运行时，动态库的代码段在内存中只有一个副本，它可以被正在运行的不同进程共享。对应用程序来说，动态库的升级也是透明的，应用程序不需要重新链接这些新版本的动态库来升级。并且动态库允许程序在运行时装载和卸载例程，进而动态地扩展应用程序的功能。因为以上优势，动态库被广泛使用。动态库也是一种目标模块，在运行时可以被加载到任意的内存地址，并在内存中与可执行程序链接，这个过程就是动态链接。

动态链接同样也要解决符号解析和重定位的问题，并且在程序运行时解决。为了实现符号解析，当静态链接生成可执行程序时，必须链接该可执行程序所依赖的动态库或者动态库的桩文件。此时，如果发现可执行程序的某个符号来自于某个动态库，静态链接器并不把相关的代码和数据复制到可执行程序中而是仅仅把一些信息记录在可执行程序中。Winodws 系统会在可执行程序中记录哪些符号来自于哪个动态库。在运行时，Winodows 系统会寻找及加载它所依赖的所有的动态库，并根据输入符号与动态库的匹配关系，从每个动态库中取得相应符号的运行时地址。而 Linux 系统只会记录可执行程序依赖哪些动态库。运行时，Linux 系统也会寻找及加载它所依赖的所有的动态库。由于缺少输入符号与动态库的匹配关系，它会建立一个搜索路径。按搜索路径指示的动态库顺序，依次寻找输入符号并取得该符号的运行时地址。由于动态库可以加载到进程地址空间的任意位置，所以如果其被加载的地址与动态库的缺省首地址不一样，运行时必须进行重定位。为了实现重

定位，动态库中需要保存一些重定位表项（relocation entry），这些表项针对与加载地址相关的代码。运行时，根据实际的加载地址，在重定位项的指导下修改相应的代码。对于动态链接的具体实现，不同的操作系统的解决方法不尽相同，但是思想差不多。在后面的 Winodws 的动态链接库（DLL，Dynamic Linked Library）和 Linux 的动态共享对象（DSO）中将有详细的介绍。

7.2 Windows DLL，Dynamic Linked Library

Windows 支持动态链接的方案是 DLL，DLL 是 Windows 操作系统的基石。所有 Windows API 的实现都包含在 DLL 中。其中最为重要是的 Kernel32.dll、User32.dll 和 GDI32.dll 这 3 大 DLL。Kernel32.dll 提供了进程、线程和内存管理等功能；User32.dll 提供了诸如窗口创建及消息发送等与用户界面相关的内容；GDI32.dll 则提供了画图和文字显示等功能。本节首先介绍 DLL 的一些基础知识及如何创建和使用 DLL，然后简单地描述 DLL 所采用的 PE（Portable Executable）文件格式，并详细介绍其中与动态链接密切相关的.edata 导出段和.idata 导入段，最后指出在创建和使用 DLL 时应注意的方面。

7.2.1 DLL 基础

1. DLL 与可执行应用及静态库

顾名思义，DLL 是动态链接库，它在程序运行时与可执行程序链接。如果一个可执行程序使用了一个 DLL，当可执行程序运行时，操作系统会把 DLL 加载到内存，并且解析可执行程序对该 DLL 功能的符号引用，使得可执行程序可以调用 DLL 的功能。

（1）可执行应用和 DLL 都是可执行模块，但是有所区别。

从使用者的角度来看，可执行应用可以在 Windows 资源管理器或控制台上直接运行，而 DLL 不可以。

从开发者的角度来看，可执行应用需要有一个 main 函数，并且 Winodws 的可执行应用常常需要包含处理消息循环或创建窗口的代码。DLL 只是包含了一组应用程序可以使用的函数。

从系统的角度来看，运行时的可执行应用（即一个进程）具有拥有权，而 DLL 不会拥有任何东西。当 DLL 的一个功能被线程调用时，DLL 的函数将使用线程的堆栈，并且 DLL 的函数中所创建的任何对象都属于该线程。一个 DLL 被链接到可执行应用，其中的代码和数据看上去就像可执行应用中的代码和数据。

（2）DLL 和静态库都可以被视为一些函数的集合，但是有一些显著的区别。

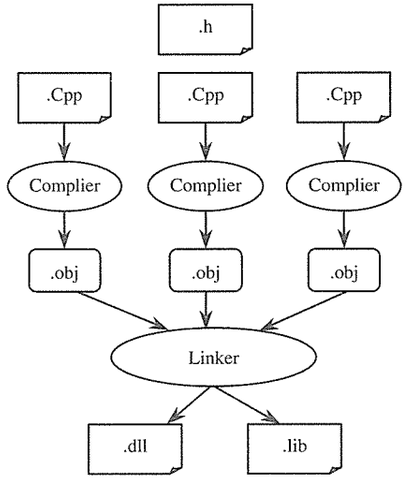
静态链接库一般以 LIB 文件形式存在，它是一个对象文件集合；而 DLL 则存在于一个可执行文件中，它在需要时可以被操作系统加载到内存。

静态库与可执行应用链接时，每个可执行应用都会含有一份静态库中的数据和代码。然而，Windows 支持多个应用程序同时使用相同 DLL 的一个实例。

静态库中只能包含代码和数据；Windows DLL 除了可以包含代码和数据，它还可以包含类似位图、图标及光标等资源。

2. 创建 DLL

创建一个 DLL 与创建一个静态库或一个可执行应用类似。图 7-2 所示为创建一个 DLL 的过程。



①创建包含输出功能的头文件

②创建 C/C++文件

③把 C/C++文件分别编译成目标文件

④⑤链接产生 DLL 及相应的导入库

图 7-2 创建一个 DLL

（1）与创建一个静态库类似，必须确定这个 DLL 对外提供的功能。把这些功能记录在一个或多个头文件中，其中包含该 DLL 需要对外提供的函数原型、类型定义和符号声明。这些头文件不仅在实现 DLL 功能的相应的 C/C++文件中需要使用，并且也会被调用该 DLL 功能的可执行程序或其他 DLL 使用。

（2）创建一些 C/C++文件来实现该 DLL 功能。

（3）把这些 C/C++文件分别编译成目标文件。

（4）通过链接器把产生的目标文件静态链接成一个单一 DLL 目标文件。在链接时需要告诉链接器产生的是一个 DLL 模块。对于 Microsoft 链接器，传给链接器/DLL 选项即可。

（5）如果链接器发现该 DLL 有任何函数或变量的输出，除了会产生 DLL 模块外，同时也会产生一个导入库（import lib）。该导入库列出 DLL 所输出的所有函数和变量的名称，即该 DLL 对外提供的功能。但是导入库不会包含任何实现代码。

从图 7-2 中可以看出 DLL 也是由一些目标模块（object module）组成，并且提供一些功能供可执行应用和别的 DLL 来使用。因此在创建 DLL 时必须指明其对外提供哪些功能，它们可以是变量、函数和 C++类，有多种方法可以指定 DLL 中导出哪些符号。

（6）使用__declspec(dllexport)修饰符。

在需要输出的函数、类和数据的声明前加上*_declspec(dllexport)*的修饰符，表示输出。当 Microsoft 的 C/C++编译器看到变量、函数原型或 C++类之前的这个修饰符时，它将某些附加信息嵌入到产生的.obj 文件中。当链接器把所有的.obj 文件链接成 DLL 时，会对上述附加信息进行分析，并产生导入库，同时产生一个输出符号表嵌入到产生的 DLL 中。

（7）使用模块定义文件（.DEF）。

模块定义文件(.DEF)是由一个或多个用于描述 DLL 属性的模块语句组成的文本文件。用于指明导出符号的 DEF 文件至少必须包含以下模块定义语句。

- 第 1 个语句必须是 LIBRARY 语句，指出 DLL 的名字。
- EXPORTS 语句中列出需要导出符号的名字，将要输出的符号经过编译器修饰过的名称罗列在 EXPORTS 语句之下。
- 可以使用 DESCRIPTION 语句描述 DLL 的用途（此句可选）。
- “;”对一行进行注释（可选）。

在模块定义文件的 EXPORT 部分指定要输出的函数或者变量。语法格式如下：

```
entryname[=internalname] [@ordinal[NONAME]] [DATA] [PRIVATE]
```

说明如下。

- entryname：输出的函数或者数据被引用的名称。
- internalname：同 entryname。
- @ordinal：表示在输出表中的顺序号（index）。
- NONAME：表示仅仅以顺序号输出（不使用 entryname）。
- DATA：表示输出的是数据项，使用 DLL 输出数据的程序必须声明该数据项为 _declspec（DLLimport）。
- PRIVATE：阻止 entryname 被 LINK 放入到导入库中。

上述各项中，只有 entryname 项是必须的，其他可以省略。对于 C 函数来说，entryname 可以等同于函数名；但是对 C++函数（成员函数及非成员函数）来说，entryname 是修饰名。可以从.map 映像文件中得到要输出函数的修饰名。如果要输出一个 C++类，则要把输出的数据和成员的修饰名都写入.def 模块定义文件中。

（8）LINK 时在命令行中指出。

对链接程序 LINK 指定/EXPORT 命令行参数，输出有关符号。命令行参数格式与 DEF 文件中指出输出符号的格式类似。

```
/EXPORT:entryname[=internalname][,@ordinal[,NONAME]][,DATA]
```

3. 使用 DLL

可以通过两种方式来使用 DLL 的功能。

（1）隐式的调用。

隐式调用指可执行应用的源代码通过函数名直接调用 DLL 的输出函数，调用方法和调用程序内部其他的函数是一样的。

如前所述，程序员在建立一个 DLL 文件时，链接程序会自动生成一个与之对应的 LIB 导入库。该库文件包含了 DLL 导出函数的符号名和可选的序列号，但是并不含有实际的代

码。在构建可执行应用时，需要把该导入库文件与与构建可执行应用的目标文件进行静态链接。通过链接该导入库，链接器可以解析可执行应用的所有的外部符号引用，并且把相应的 DLL 文件名（但不是完全的路径名）存储在可执行应用文件内部。

当应用程序运行时，Windows 根据这些信息发现并加载 DLL，然后通过符号名或序列号实现对 DLL 函数的动态链接。所有被应用程序使用的 DLL 文件都会在可执行应用 EXE 文件加载时被加载在到内存中。

（2）显式调用。

在可执行应用运行时，由应用程序显式加载需要的 DLL 并且显式链接到需要的输出符号。换句话说，当应用程序运行时，它能够决定是否需要调用 DLL 中的函数。如果需要，它可以将 DLL 显式的加载到进程的地址空间中，并且获得需要调用函数的虚拟内存地址，然后使用该地址来调用该函数。这种方法的优点是一切操作都发生在应用程序运行时。

应用程序可以调用 LoadLibrary 或 LoadLibraryEx 来显式加载 DLL 模块，这两种方法会设法找到需要的 DLL 文件，并把它映射到调用进程的地址空间。两个函数返回的 HINSTANCE 值用于标识 DLL 文件映像映射到的虚拟地址。如果 DLL 不能被映射到进程的地址空间中，则返回 NULL：

```
HINSTANCE  LoadLibrary(PCTSTR pszDLLPathName);

HINSTANCE  LoadLibrary.Ex(
    PCTSTR  pszDLLPathName,
    HANDLE  hFile,
    DWORD  dwFlags);
```

一旦 DLL 被映射到进程的地址空间，即可通过调用 GetProcAddress 来获取它所需要引用的符号的虚拟地址：

```
FARPROC  GetProcAddress(
    HINSTANCE  hinstDll,
    PCSTR  pszSymbolName);
```

GetProcAddress 函数将符号名或序列号转换为 DLL 内部的地址。参数 hinstDll 是先前 LoadLibary（Ex）返回的 HINSTANCE；pszSymbolName 是一个以 0 结尾的字符串地址，它指定需要得到地址的符号的名称，或者一个需要得到地址的符号的序号。如果 DLL 模块的输出表中不存在需要的符号，GetProcAddress 返回 NULL。

7.2.2 DLL 如何工作

1. PE 文件格式

Windows 的可执行程序 EXE 和动态链接库 DLL 都采用 PE（Portable Executable）文件格式，二者之间的区别是语义上的区别。在文件中只有一个特殊标志指示该 PE 文件是一个 DLL，还是一个 EXE。

PE 文件的一个显著的优点是存在磁盘上的数据结构和内存中的数据结构是一致的，即数据在磁盘上的存放方式与数据被读到内存中的存放方式一致。因此当知道如何从 PE 文件中找到想要的信息，即可采用几乎相同的方式从内存中找到相关信息。由于 PE 文件的这个特性，可以把一个 PE 文件加载到内存中的主要工作就是把 PE 文件映射到内存。但是 PE 文件并不会作为一个单一的文件被整体映射到内存。Windows 加载器会根据 PE 文件中的某些信息，有选择地把 PE 文件中的某些部分映射到内存。

PE 文件格式被组织为一个线性的数据流，它由一个 MS-DOS 头部开始，接着是一个实模式的程序残余以及一个 PE 文件标志，紧接着 PE 文件头和可选头部。这些之后是所有的段头部，段头部之后跟随所有的段实体。文件的结束处是一些其它的区域，其中有一些混杂的信息，包括重定位信息、符号表信息、行号信息以及字串表数据，示例如图 7-3 所示。

图 7-3 PE 文件格式示例

段包含 PE 文件的主要内容，包括代码、数据、资源，以及其他可执行信息，每个段都有一个头部和一个实体（原始数据）。一个 Windows 的应用程序典型地拥有 9 个预定义段，它们是.text、.bss、.rdata、.data、.rsrc、.edata、.idata、.pdata 和.debug，但不是每个应用程序都需要所有的这些段。

.text 是可执行代码段，在静态链接时，Windows NT 默认的做法是将所有 OBJ 文件中的代码段组成一个单独的段。

.bss 段表示应用程序的未初始化的全局数据，包括所有函数或源模块中声明为 static 的变量。

.rdata 段表示只读的数据，比如常量、字符串和调试目录信息等。

所有其他变量（除了出现在栈上的自动变量外）存储在.data 段之中，基本上这些是应用程序或模块的全局变量

2. 动态链接和 Import Address Table（IAT）

可执行程序（包括可执行应用和 DLL）可以使用 DLL 中的输出函数和变量。如果可执行程序通过隐式调用方式使用 DLL 的函数或变量，在加载可执行程序时会自动加载该 DLL。同时，加载器会保证该 DLL 所依赖的其他 DLL 也会被自动加载。加载动态库后，需要完成动态链接。动态链接的任务主要是把可执行程序引用的 DLL 函数（或变量）绑定到该函数（或变量）被加载到地址空间的虚拟地址。

一般来说，编译器将 C++中的函数调用生成汇编指令时有两种形式，一种如 CALL 0x0040100C；另一种如 CALL DWORD PTR [0x0040100C]。第 1 种方式直接把控制权转移到 0x0040100C 地址，第 2 种方式是从 0x0040100C 地址中取出一个双字节的值作为下一条被执行指令的地址。当可执行程序的源代码被编译时，如果代码中没有特别信息，编译器无法判断一个函数调用是调用本模块中的函数，还是调用别的模块中的函数。因此编译器只会产生如下格式的指令：

```
CALL XXXXXXXX
```

其中 XXXXXXXX 是目标函数的地址，该值由链接器在静态链接阶段填入。但是如果这个函数是来自一个外部的 DLL，静态链接时无法得到目标函数的虚拟地址。必须等到可执行程序运行时，DLL 被 Winodws 加载器加载到内存后才能由 Windows 加载器填写 CALL 指令中目标函数的地址。注意到此时修改的 CALL 指令处于代码段。如果可执行程序中有多处调用了这个外部 DLL 的函数，并且调用点分散在各处，就需要 Windows 加载器修改各处 CALL 指令。这不仅会加大加载器的工作量，而且会消耗更多的内存（参考第 4 章的内容）。因此，静态链接时会产生如下的目标代码：

```
CALL 0x0040100C

0x0040100C: JMP DWORD PTR [0x00405030]
```

首先调用语句（CALL）会把控制权转移到一小段辅助代码，然后执行辅助代码。该辅助代码仅仅是一个间接跳转语句，从 0x00405030 地址中取出一个双字节的值作为下一条被

执行指令的地址。0x00405030 地址处于可执行程序（PE 文件）中的一个特殊区间。该区间是一段连续的数据结构，每一项是函数指针，这个函数指针数组就是 import address table（IAT）。每个函数指针的值就是这个外部 DLL 函数被加载到地址空间的虚拟地址，该值由 Windows 加载器填写。因此不论这个外部 DLL 函数被可执行程序调用多少次，也不管调用处如何分布，Windows 加载器只需要修改 IAT 表中的一项即可以完成函数的调用和其地址的绑定。

那么 JMP 指令如何产生？由上一节得知，如果一个可执行程序需要隐式使用外部 DLL 的函数，在构建可执行应用时，必须静态链接该 DLL 的导入库文件。这个 JMP 指令就来自于导入库。打开一个导入库，检查相应的导出函数的代码，你会发现一个类似的 JMP 指令。

如果可执行程序的源代码在调用外部 DLL 函数时能提供一些特殊信息，编译器可以根据这些信息产生的指令如下：

```
CALL DWORD PTR [0x00405030]
```

该指令的意思是从 0x00405030. 地址中取出一个双字节的值作为下一条被执行指令的地址。显然，0x00405030 地址应该处于 IAT 的地址范围中，即是 IAT 中的一项。这种调用方式比第 1 种调用方式更为简洁。它不仅少执行了一条 JMP 语句，而且产生的可执行文件会小一些（没有 JMP 指令）。

只要使用_declspec(dllimport)修饰符来修饰函数，告诉编译器该函数来自于外部的 DLL。编译器就会产生这种指令：

```
CALL DWORD PTR [XXXXXXXX]
```

而不是：

```
CALL XXXXXXXX
```

同时，编译器会产生__imp_functionname 符号作为 CALL 指令函数地址部分的符号引用。例如，如果需要调用 MyFunction 函数，编译器会产生__imp_MyFunction 符号。查看导入库，你会发现除了一些通常的符号名外，还会有一些以__imp__开头的符号名。链接器在静态链接时会把__imp_这种符号名直接绑定到 IAT 中的一项，而把普通的函数符号名绑定到 JMP 辅助代码段。

例如：构成某个 DLL 的 C++的代码中定义了如下输出函数：

```
extern "C" __declspec(dllexport) int fnDLLC(void);
extern "C"__declspec(dllexport) void printFromDLL();
```

使用 dumpbin 工具可以在相应的导入库中可以查到如下的符号：

```
_imp_fnDLLC
_fnDLLC
_imp_printFromDLL
_printFromDLL
```

很明显，每个导出函数都有一个以__imp_开头的对应符号。

3. PE 文件中的导入段和导出段

PE 文件中有两个特殊的段（导入段.idata 和导出段.edata）用来支持 DLL 的动态链接，导入段.idata 中列出了这个可执行程序需要从外部 DLL 引入的符号（包括函数和变量）；而导出段.edata 中记录这个可执行程序对外输出的符号。一般来说，可执行应用 EXE 仅含有一个.idata 段，而 DLL 总是含有一个.edata 段和一个.idata 段（如果该 DLL 需要使用别的 DLL）。

（1）导出段.edata。

DLL 对外输出符号时必须指定对外输出哪些符号，并且让使用者可以获得这些符号在 DLL 中的地址。每个对外输出符号有一个序数号与之关联，使用者可以通过查询序数号来得到相应符号的地址。当然，每个输出符号都有一个 ASCII 字符名与之关联。

通常来说，DLL 的使用者通过符号名，而不是符号相关的序数来定位 DLL 中的一个符号。然而，当通过符号名来定位符号时，系统首先要根据符号名获得所对应的序数，再用序数来取得相应的地址。因此，通过序数来定位输出符号的地址会更加有效。但是这需要 DLL 的创建者保证相同的符号在该 DLL 的不同版本中以同样的序数输出；否则就会带来问题。在实践中，以序数方式输出符号常常被用在那些很少被修改的系统服务 DLL 中。

导出段.edata 由一个称为“输出目录表”（export directory table）的数据结构和定义了输出符号的表构成，示例如图 7-4 所示：

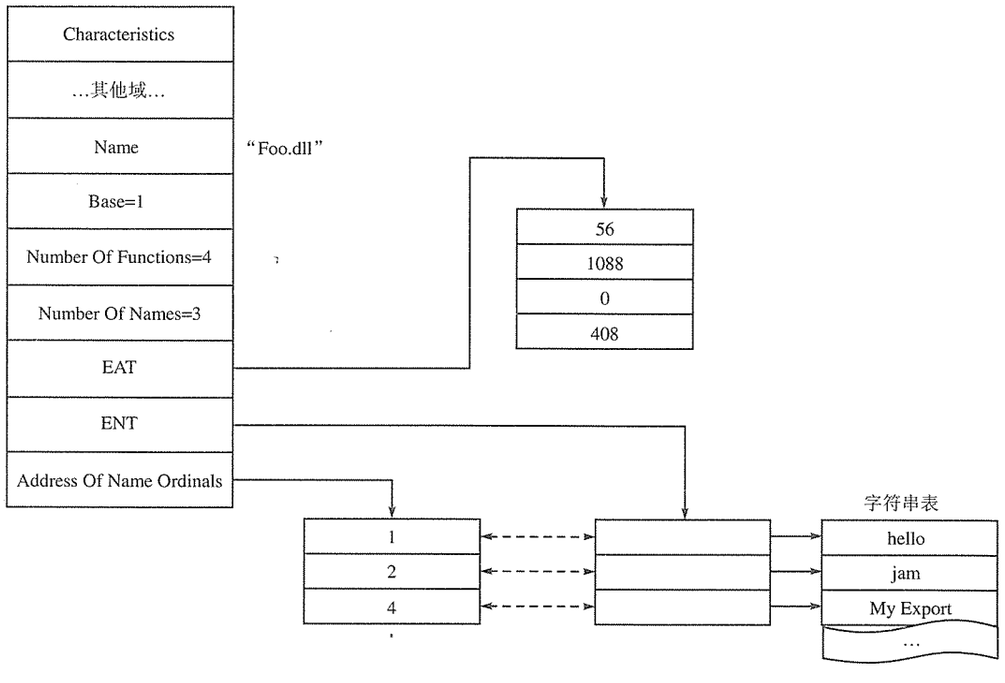


图 7-4　输出目录表示例

通过输出目录表可以定位 3 个数组。输出地址表（Export Address Table EAT）是必须存在的，它是一个指针数组，每一项是一个输出符号的 RVA（Relative Virtual Address，相对虚拟地址，相对于 PE 文件的基地址）。输出序数表和输出名称表（ENT，Export Name Table）是并列的。输出名称表中的每一项也是一个 RVA，其指向字符串表中一个代表其名称的字符串。而输出序数表的每一项则是符号的输出序数。

如图 7-4 所示，如果需要通过函数名的方式来定位 Foo.dll 输出的 MyExport 函数。

- 根据输出目录表可以得到输出名称表的地址。
- 在输出名称表中查询 MyExport 字符串，得到它位于输出名称表的第 3 项。
- 从输出序数表相应的第 3 项中取得 MyExport 的输出序数 4。
- 把取得的输出序数 4 作为输出地址表的索引，从输出地址表的第 4 项中取得 MyExport 的 RVA。

输出序数并不都是从 0 开始计数的，在输出目录表中有一项（Base）记录了输出序数的起始值。因此从输出序数表中取得的输出序数需要减去输出序数的起始值，再用该值来作为输出地址表的索引。输出名称表中的符号名称以字母顺序排列，因此系统可以使用二分查找法来查找输出符号。

（2）导入段.idata

与.edata 段类似，导入段有一个称为“输入目录表”（import directory table）的数据结构。一个可执行程序可以从多个外部 DLL 中导入符号，导入段中的每一个输入目录表代表一个隐式链接的 DLL，这些输入目录表构成一个数组存放在.idata 前部。.idata 段中没有字段指出该数组的项数，但它的最后一个单元被设置为 NULL，可以由此计算出该数组的项数。输入目录表主要包含两个指针，分别指向两个数组，如图 7-5 所示。

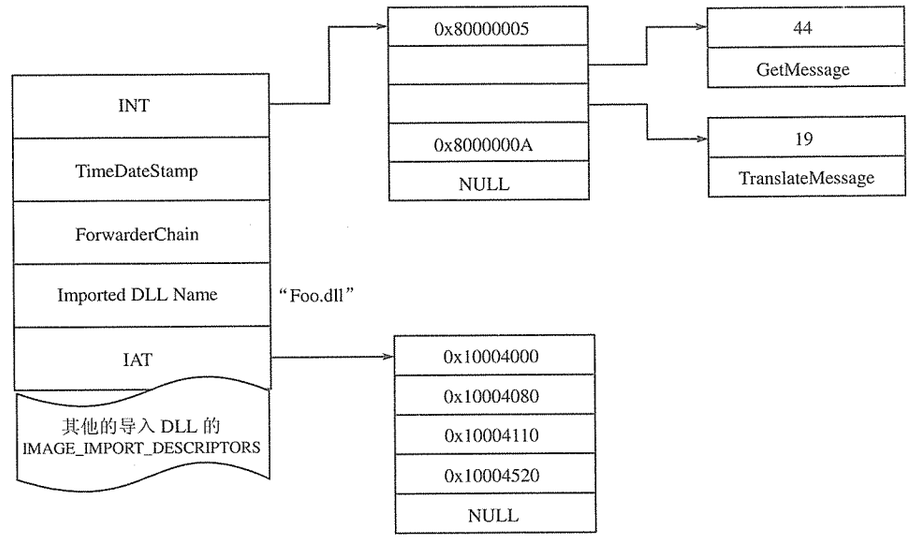


图 7-5 输入目录表

这两个数组是完全相同的，一个数组称为“IAT”（Import Address Table）。另一个数组称为“INT”（Import Name Table），数组中的每一个元素都是一个指针大小的数据结构 IMAGE_THUNK_DATA，每一个 IMAGE_THUNK_DATA 元素与一个导入符号对应。这两个数组都是以设置为 0 的 IMAGE_THUNK_DATA 元素结束。

IMAGE_THUNK_DATA 是一个联合（union），它可以有如下解释：

```
DWORD Function;          // Memory address of the imported function
DWORD Ordinal;           // Ordinal value of imported API
DWORD AddressOfData;     // RVA to an IMAGE_IMPORT_BY_NAME with
                         // the imported API name
DWORD ForwarderString;   // RVA to a forwarder string
```

可执行程序初始时（即没有与其依赖的 DLL 完成动态链接），IMAGE_THUNK_DATA 元素的值或者是导入符号的序数，或者指定导入符号的名称（一个 RVA 指向一个包含符号名称的数据结构，该数据结构称为“IMAGE_IMPORT_BY_NAME”）。如何区分 IMAGE_THUNK_DATA 元素的值是序数还是指针关键看 IMAGE_THUNK_DATA 的最高位，如果最高位被设置，则 IMAGE_THUNK_DATA 元素的值代表序数；如果最高位没有设置，则 IMAGE_THUNK_DATA 元素的值代表指向 IMAGE_IMPORT_BY_NAME 的 RVA。

IMAGE_IMPORT_BY_NAME 的结构如下：

```
typedef struct IMAGE_IMPORT_BY_NAME{
WORD Hint;       // the ordinal of imported API might be
BYTE Name;       // ASCII name of imported API
};
```

Name 含有导入函数的函数名，函数名是一个 ASCII 字符串。注意这里虽然将 Name 的大小定义为字节，其实它是可变尺寸域，只不过没有更好方法来表示结构中的可变尺寸域。

Hint 指示函数在其所驻留 DLL 的导出段中输出名称表（ENT）的索引号，该值是被 PE 加载器用来在 DLL 的输出名称表里快速定位符号。如果被定位的符号名与 Name 域指定的符号名匹配，则使用被快速定为的符号；否则使用二分查找法在整个输出名称表中搜索。该值不是必需的，一些链接器将此值设为 0。

当可执行程序与其依赖的 DLL 完成动态链接后，IMAGE_THUNK_DATA 元素的值可以是导入符号的虚拟地址。当可执行程序没有被 Windows 加载器加载到内存时，IAT 表与 INT 表中的值完全相同。但是当 Windows 加载器把可执行程序加载到内存时，它会重写 IAT 的每一项为导入符号的虚拟地址（这是动态链接的关键一步）。不过，Windows 加载器不会修改 INT 中的内容。为什么 PE 文件中需要保存两个完全相同的 IMAGE_THUNK_ DATA 数组？因为有些工具（如 BIND）可以重写 IAT 表中的值。当 Windows 加载器把可执行程序加载到内存时，如果加载器需要访问原来的信息，它可以从 INT 表中得到。

4. 基地址重定位（base relocation）

可执行程序中的很多指令都与内存地址相关。当构建一个可执行程序时，需要提供给可执行程序一个缺省的基地址，该地址指明可执行程序希望被加载到地址空间的虚拟地址。只有当可执行程序确实被加载到其希望的基地址，可执行程序中使用的虚拟地址才是正确。一般来说，构建一个可执行应用 EXE 时，如果未指定基地址，LINK 产生的缺省基地址是 0x400000。而对于 DLL，LINK 产生的缺省基地址是 0x10000000。

当执行一个可执行程序时，可执行应用 EXE 是第 1 个被加载到内存的可执行模块。因此 EXE 程序都是可以被加载到它所希望的基地址上。但是 DLL 则没有这么幸运，如果在执行可执行程序时需要加载多个 DLL（或者它们有相同基地址，或者其地址空间相互重叠）。除了第 1 个被加载的 DLL 可以被满足外，其他 DLL 都会被加载到另外的地址上。此时需要做基地址重定位以避免可执行程序中地址引用的错误。

基地址重定位信息记录了可执行程序中的一些位置，每个位置上的值都是一个虚拟地址。加载器修改这些位置上的值来完成重定位，这些重定位信息被放置在 PE 文件的.reloc 段。PE 文件中重定位段由多个重定位块构成，每一块用来描述一个内存页中的所有重定位项。每个重定位块以一个 IMAGE_BASE_RELOCATION 结构开头，后面跟着在本页面中使用的所有重定位项，每个重定位项占用 16 位的地址（即一个 word），IMAGE_BASE_RELOCATION 结构的定义如下：

```
typedef struct IMAGE_BASE_RELOCATION{
DWORD VirtualAddress;     // 重定位内存页的起始 RVA
DWORD SizeOfBlock;        // 重定位块的长度，包括 IMAGE_BASE_RELOCATION
};
```

VirtualAddress 字段是当前页面起始地址的 RVA 值，本块中所有重定位项中的 12 位地址加上这个起始地址后就得到了真正的 RVA 值。SizeOfBlock 字段定义的是当前重定位块的大小，从这个字段的值可以算出块中重定位项的数量。IMAGE_BASE_RELOCATION 结构后面跟着重定位项，每个重定位项的 16 位数据位中的低 12 位需要重定位的数据在页面中的地址，剩下的高 4 位被用来描述当前重定位项的种类，见下表。

高 4 位值	在 Windows.inc 中的预定义值	含　义
0	IMAGE_REL_BASED_ABSOLUTE	这个重定位项无意义，仅仅用来作为对齐
1	IMAGE_REL_BASED_HIGH	重定位地址指向的双字中，仅仅高 16 位需要被修正
2	IMAGE_REL_BASED_LOW	重定位地址指向的双字中，仅仅低 16 位需要被修正
3	IMAGE_REL_BASED_HIGHLOW	重定位地址指向的双字的 32 位都需要被修正

7.2.3 关于 DLL 的杂项

1. 导入类和变量

一个 DLL 可以导出函数、变量和 C++类以供其他模块（DLL 及 EXE）使用。在实践中，一般只导出函数。一个 DLL 可以导出 C++类，其他模块如果想要使用这个 C++类，必须确保这个模块是用与编译 DLL 相同的编译器编译出来。基于这个原因，一般 DLL 不导出 C++类；除非确信可执行模块的开发者与 DLL 的开发者使用相同的开发环境。声明导出类的最简单的方式是在该类的声明前加上*_declspec(dllexport)*的修饰符，使用导出类的程序也需要声明该类为_declspec(dllimport)。

DLL 中导出变量可能会破坏模块的封装性，同时也使 DLL 的代码难以维护。根据前面的知识，可执行模块在使用 DLL 导出的全局变量时，从 IAT 表中得到的仅仅是导出变量的地址。如果确实需要使用导入变量，传统的做法需要显式地以指针方式来声明导入变量，并且在使用时显式地解引用。现在由于微软新的编译器提供额外支持，使用 DLL 输出变量的程序必须声明该变量为_declspec(dllimport)，编译器会自动加上额外的指针解引用操作。

2. 延迟加载（delay load）

Visual C++ 6.0 增加 DLL 延迟加载的一种新特性。如果一个可执行程序依赖于某个 DLL，当这个可执行程序被加载到内存时，不需要自动加载那个 DLL。仅当可执行程序第 1 次使用这个 DLL 的功能时，该 DLL 才会被加载。这是因为当可执行程序第 1 次使用这个 DLL 的函数时，有一段辅助代码会调用 LoadLibrary 来加载该 DLL，并调用 GetProcAddress 来找到函数的地址。

需要注意延迟加载的特性完全由编译器和某些运行时系统库共同完成，Windows 系统对此并没有做出特别处理。当构建某个可执行程序时，可以指定延迟加载某个 DLL。此时，链接器不会在.idata 段中产生与该 DLL 相关的输入目录表数据结构，而产生一些类似的数据结构，这个数据结构名为“ImgDelayDescr”。每一个延迟加载的 DLL 都有一个 ImgDelayDescr 与之对应。如下所示：

```
typedef struct ImgDelayDescr{
DWORD grAttrs;        //描述该结构的某些属性。目前只有一个值 dlattrRva（1），表示该结构
```



```
中所涉及的地址是 RVA，而不是一个真正的虚拟地址
    DWORD rvaDLLName;     //一个字符串的 RVA，指明 DLL 的名称
    DWORD rvaHmod;
    DWORD rvaIAT;         //IAT 表的位置
    DWORD rvaINT;         //INT 表的位置
    DWORD rvaBoundIAT;    //目前没有用到
    DWORD rvaUnloadIAT;   //目前没有用到
    DWORD dwTimeStamp;
    };
```

但是 Windows 加载器并不会处理这些数据，它们会被某些运行时系统库来操作。ImgDelayDescr 中也有指针指向与.idata 中类似的 IAT 与 INT 表，此时可执行程序中的代码是通过间接引用 ImgDelayDescr 指向的 IAT（而不是.idata 段中的 IAT）的项来使用延迟加载 DLL 的输出符号。这两个表的格式与正常导入库的 IAT 与 INT 表的格式完全一致，但是 IAT 表中的值并不相同。正常导入的 DLL 的 IAT 表中的值与 INT 表相同。当加载该 DLL 时，IAT 表会被 Windows 加载器修改成导入符号的实际虚拟地址；而延迟加载的 DLL 的 IAT 表中的值则指向一段辅助代码。该辅助代码会调用 LoadLibrary 来加载该 DLL，调用 GetProcAddress 来找到函数的地址，并且把得到的导入函数虚拟地址写入 IAT 表的相应项中。可执行程序下次调用该函数时，不需要再查找该导入函数的地址。

构建可执行程序时，如果想要指定延迟加载某个 DLL，只需要在 LINK 时加上/DELAYLOAD:dllname 参数。该参数可以被重复使用来指定延迟加载多个 DLL，同时在 LINK 时需要链接 Delayimp.lib 库，Delayimp.lib 库是 Visual C++提供的。

7.3 Linux DSO

Unix 世界使用动态共享对象 DSO（Dynamic Shared Object），或称为“动态共享库”（Dynamic Shared Library）来实现动态链接。本节将首先介绍共享对象所采用的 ELF（Executable and Linking Format）文件格式，然后描述可执行程序在运行时与共享对象进行动态链接的过程，以及共享对象中对动态链接过程起重要作用的数据结构。最后介绍如何创建和使用共享对象，并且指出在创建和使用共享对象过程中需要注意的方面。

7.3.1 DSO 与 ELF

1. ELF（Executable and Linking Format）的出现

ELF 文件格式是由 Unix System Laboratories 开发的，很快就被广泛接受，成为 Unix 世界的二进制文件格式的标准。它已经成为 Linux、Solaris 2.x，以及 SVR4 等操作系统的二进制文件的缺省格式。

Linux 最初使用的二进制文件格式是 a.out，但是 a.out 在设计时并没有考虑到动态链接。使用 a.out 作为动态库的主要限制是动态库被加载到进程的地址空间后不能进行重定位，这会要求每个动态库有一个固定的加载地址。因此必须存在一个集中分配动态库地址空间的机制；否则会出现不同的动态库使用相同的地址空间，从而导致灾难性的后果。集中分配动态库地址空间本身就很难实现，并且大量的动态库会把地址空间划分成许多部分，甚至耗尽地址空间的所有地址。同时，固定的地址空间也不利于动态库的维护。当在动态库增加新功能时，产生的新版本的动态库不能突破已分配的地址空间。种种原因使得人们需要一种新的文件格式来展示动态链接的强大功能。ELF 格式的强大功能和灵活性使其被 Linux 采用，并替换 a.out 格式。

ELF 格式适用于可执行程序、可重定位二进制文件和动态库（共享对象），这些文件中包含数据、代码和提供给操作系统、链接器和加载器使用的信息。可执行程序可以被直接拷贝到内存并执行。可重定位目标文件可以在链接时与其他可重定位的目标文件合并，共享对象可以在加载或者运行时，被动态地加载到内存并完成链接。

2. ELF 结构简介

由于 ELF 适用于上述 3 种二进制文件，因此它可以被不同的工具操作。但是不同的工具以不同的视角来对待 ELF 文件，编译器，汇编器和链接器把 ELF 文件看成由节头部表（section header table）描述的一组节（section）组成；而系统加载器则把 ELF 文件看成由程序头表（program header table）描述的一组段（segment）组成，如图 7-6 所示。

如果想得到关于符号表、重定位及动态链接等信息，需要与链接器一样对待 ELF 文件，即把它看成一组节；如果想得到数据段及代码段的位置信息，需要把 ELF 文件看成一组段。这些节或者段是由相应的节头部表和程序头部表描述，不同类型的文件需要不同的头部表。可重定位文件为了提供给链接器相应的信息，需要包含节头部表；可执行程序为了映射到

内存中，需要包含程序头部表；而共享对象既需要映射到内存，又需要提供动态链接的信息，需要包含节头部表和程序头部表。因此对于共享对象，可以有两种角度来解读该对象。虽然视角不同，但是内容相同。一个段常常包含多个节，例如，一个只读的可加载段通常包括代码节、只读数据节及动态链接符号节。

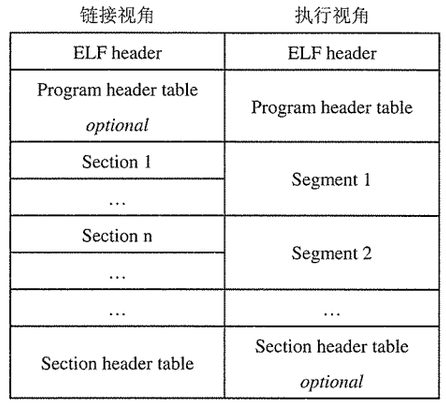


图 7-6 ELF 结构

所有的 ELF 文件都是以 ELF 头部（ELF header）开始的，其描述如下所示，并且 ELF 头部也是文件中唯一的一个有固定位置的数据结构：

```
#define EI_NIDENT    16

typedef struct   {
    unsigned char    e_ident[EI_NIDENT]; //文件的 ID，见下文解释
    ELF32_Half       e_type;             //二进制文件的类型
    ELF32_Half       e_machine;          //目标体系结构
    ELF32_Word       e_version;          //ELF 版本
    ELF32_Addr       e_entry;            //第 1 条可执行指令的地址
    ELF32_Off        e_phoff;            //程序头部表的偏移
    ELF32_Off        e_shoff;            //节头部表的偏移
    ELF32_Word       e_flags;            //处理器相关的标志
    ELF32_Half       e_ehsize            //ELF 头部的大小
    ELF32_Half       e_phentsize;        //程序头部表中每项的大小
    ELF32_Half       e_phnum;            //程序头部表包含表目的个数
    ELF32_Half       e_shentsize;        //节头部表中每项的大小
    ELF32_Half       e_shnum;            //节头部表包含表目的个数
```

```
    ELF32_Half      e_shstrndx;         //节头部表中某个项的索引，该项
                                        //指出包含节名称的字符串节
} ELF32_Ehdr;
```

ELF 头部的前 16 个字节是文件的 ID 信息。最初四个字节（也是文件的最初 4 个字节）是确定该文件是否为 ELF 文件的魔法数字（magic number）。接着 3 个字节描述了 ELF 头部余下部分的格式，它们指明了字的大小，字节序和 ELF 头部的版本信息。一旦程序读了 class 和 byteorder 标志，它就知道该 ELF 文件的字节序（big-endian 或者 little-endian）和字的大小（32 位还是 64 位），这样程序可以做相应的转换。因此 ELF 文件可以适应不同类型的机器，即使该机器的字节序和 ELF 文件的字节序不一样。ELF 头部剩余的部分描述了二进制文件的类型（可重定位、可执行，共享对象，还是内存映像文件），目标体系结构（SPARC、x86 或 68k 等），使用 ELF 格式的版本。如果存在程序头部表或者节头部表，它们的大小和位置也是在 ELF 头部中描述。对于可执行程序，ELF 头部表还要指明第 1 条可执行指令的位置。

3. 链接器视角下的 ELF 文件

一个可重定位的 ELF 文件（包括共享对象）可以看成由一些节（section）组成，一个典型的可重定位的 ELF 文件包括图 7-7 所示的节。

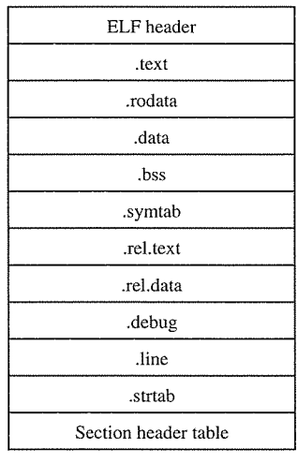


图 7-7　一个典型的可重定位的 ELF 文件

不同节的大小和位置在节头部表中指出，节头部表是一个数组，数组的每一项都是一个用来描述节的数据结构，数组的个数和每一项的大小都由 ELF 头部表的 e_shnum 和

e_shentsize 指出。描述节的数据结构如下：

```
typedef struct  {
    Elf32_Word  sh_name;        //节的名称
    Elf32_Word  sh_type;        //节的类型
    Elf32_Word  sh_flags;       //节的属性
    Elf32_Addr  sh_addr;        //映射到内存的起始地址
    Elf32_Off   sh_offset;      //节在 ELF 文件中的偏移
    Elf32_Word  sh_size;        //节的大小
    Elf32_Word  sh_link;        //与节类型相关的值
    Elf32_Word  sh_info;        //额外的一些信息
    Elf32_Word  sh_addralign;   //地址对齐限制
    Elf32_Word  sh_entsize;     //节中每项的大小
} Elf32_Shdr
```

该数据结构指出节的名称、类型、映射到内存的起始地址（如果该节可以被加载）、位置（该节在 ELF 文件中的偏移），以及节的大小等。其中的节主要分为如下类型。

（1）PROGBITS：该类型的节为程序内容，包括代码、数据和调试信息。

（2）NOBITS：该类型的节在文件中不占空间，但是类似 PROGBITS。该类型用在程序加载时为 BSS 数据分配空间。

（3）SYMTAB 和 DYNSYM：该类型的节为符号表；SYMTAB 为通常的链接器提供符号，当然也有可能在动态链接时被使用。作为一个完整的符号表，它可能包含一些动态链接时不需要的符号。因此一个 ELF 文件也可能包含了一个 DYNSYM 类型的节，它保存一个动态链接时所需最小的符号集合来节省空间。由于动态符号表需要被加载到内存，因此该符号表被缩减到越小越好。

（4）STRTAB：该类型的节为字符串表。ELF 文件可以包含多个字符串表，每个字符串表有各自的用途。例如，有的字符串表包含节的名称，有的字符串表包含普通符号的名称，而有的字符串表包含动态链接符号的名称。

（5）REL and RELA：该类型的节为重定位信息，它们保存具有明确加数的重定位项，即每个重定位项需要加上一个固定的值。REL 的重定位项所指明的代码或者数据中已经包括了基地址，而 RELA 的重定位项本身包含基地址信息。

（6）DYNAMIC：该类型的节包含动态链接信息

（7）HASH：该节保存一个符号的哈希（hash）表，所有参与动态链接的共享对象一定包含一个符号哈希表（hash table）。

节头部表的 sh_flags 成员保存一个标志，用来描述节的属性，其值可以是 WRITE、ALLOC 和 EXECINSTR。WRITE 表示该节包含进程执行过程中可被写的数据，ALLOC 表示 该节在进程执行过程中占据着内存。一些控制节不需要在该目标文件的内存映像中出现，对于这些节，这个属性应该关闭；EXECINSTR 表示该节包含可执行的机器指令。

上面描述的一个典型的可重定位的 ELF 所包含的节的类型和属性见下表：

节（section）名称	类型（sh_type）	属性（attribute）
.text	PROGBITS	ALLOC + EXECINSTR
.rodata	PROGBITS	ALLOC
.data	PROGBITS	ALLOC + WRITE
.bss	NOBITS	ALLOC + WRITE
.symtab	SYMTAB	见下文
.rel.text	REL	见下文
.rel.data	REL	见下文
.debug	PROGBITS	None
.line	PROGBITS	None
.strtab	STRTAB	见下文

.text 节保存程序的代码或者说可执行指令。

.rodata 节保存只读数据，在进程映像中构造不可写的段。

.data 节保存初始化数据，这些数据需要存在于程序内存映像中。

.bss 节保存着未初始化的数据，这些数据需要存在于程序内存映像中。当程序开始运行，系统初始化这些数据为 0。该节不占文件空间，正如其节类型 NOBITS 指示的一样。

.symtab 节保存一个符号表。假如文件有一个可装载的段，并且包含该节，那么节的 ALLOC 属性将被设置；否则不设置。

.rel.text 和.rel.data 节保存代码或者数据的重定位信息。假如文件包含一个可装载的段，并且这个段包含该节，那么该节的 ALLOC 属性将设置；否则不设置。

.debug 节保存调试的信息。

.line 节包含行数信息，它描述源程序与机器代码之间的对应关系。

.strtab 节保存字符串。假如文件有一个可装载的段，并且包括该节，那么节的 ALLOC 属性将被设置；否则不设置。

除了上述节之外，还有一些节。例如，后面将要介绍的为动态链接服务的.got 节和.plt 节，C++程序可能要包括的.init 节、.fini 节、.rel.init 节和.rel.fini 节。

4．程序执行视角下的 ELF 文件

可执行的 ELF 文件的与可重定位的 ELF 文件在总体上格式相同，但是它被安排为更加适合被映射到内存。可执行的 ELF 文件（包括共享对象）可以看成由一些段（segment）组成，一个典型的可执行的 ELF 所包含的段如图 7-8 所示。

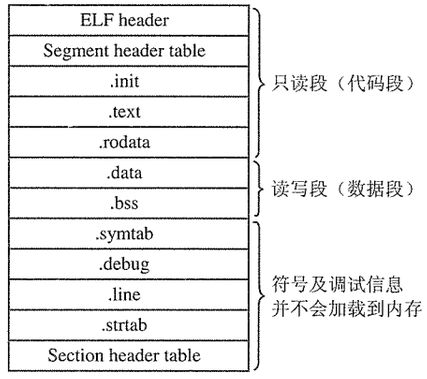


图 7-8 一个典型的可执行的 ELF 所包含的段

这些段被程序头部表（program header table）描述，程序头部表在 ELF 文件中紧接着 ELF 头部（ELF header）。它也是一个数组，数组的每一项都是一个用来描述段的数据结构，数组的个数和每一项的大小都由 ELF 头部表的 e_phnum 和 e_phentsize 指出。程序头部表中描述段的数据结构如下：

```
typedef  struct  {
    Elf32_Word   p_type;
    Elf32_Off    p_offset;
    Elf32_Addr   p_vaddr;
    Elf32_Addr   p_paddr;
    Elf32_Word   p_filesz;
```

```
    Elf32_Word   p_memsz;
    Elf32_Word   p_flags;
    Elf32_Word   p_align;
} Elf32_Phdr;
```

（1）p_type：指出该段的类型，可能的类型值包括可加载（LOAD）、动态链接信息（DYNAMIC）及辅助信息（NOTE）等等。

（2）p_offset：给出该段的驻留位置相对于文件开始处的偏移。

（3）p_vaddr：给出该段在内存中的首字节地址。

（4）p_paddr：在与物理地址定位相关的系统中，是为该段的物理地址而保留。由于 System V 忽略了应用程序的物理地址定位，所以该成员对于可执行文件和共享对象而言是未指定内容的。

（5）p_filesz：给出文件映像中该段的字节数，它可能是 0。

（6）p_memsz：给出内存映像中该段的字节数，它可能是 0。（如果包含 BSS，该值会大于 p_filesz）

（7）p_flags：给出和该段相关的标志，可能的值为可读、可写及可执行。

（8）p_align：可加载的进程段必须有合适的 p_vaddr 和 p_offset 值，该成员给出段在内存和文件中的对齐值。0 和 1 表示不需要对齐；否则 p_align 必须为正的 2 的幂，并且 p_vaddr 应当等于 p_offset 模 p_align。

一个可执行的 ELF，文件通常只有少数几个段，如包含代码和只读数据的只读段，或可读写数据的可读写段，每个段通常包含若干个节。程序执行时，操作系统通过程序头部表提供的信息把段映射到地址空间并执行。现代的操作系统一般都采用虚存技术，需要把段映射到完整的多个页面（page）中。也就是段的头部应该映射到一个页的起始地址，段的尾部如果不能达到一个页的最后，其后的地址也不应该被其他段所占有。

一个简单的方法是在可执行的 ELF 文件的段间填充字节，使得文件中的段的大小已经按页的大小对齐。虽然段映射变得简单，但是会使文件变大，并且包含许多无用的填充字节。ELF 使用的映射方式如图 7-9 所示。

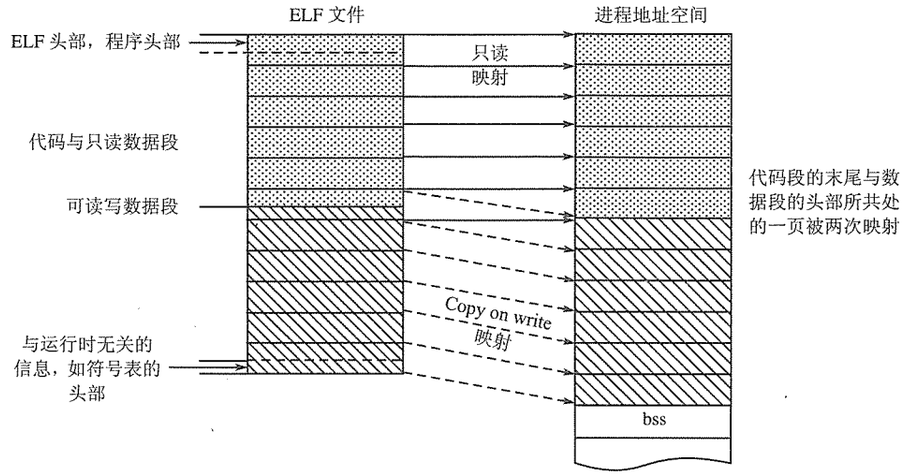


图 7-9 ELF 使用的映射方式

由于没有明确的要求，可以需要代码段（text segment）中的代码必须从 0 地址（或者从系统规定的缺省地址，如 0x8048000）开始，因此可以把 ELF 头部及程序头部表等也映射到地址空间。因为程序不会访问这些地址，把它们映射到地址空间并不会带来任何副作用。同时代码段的末尾和数据段（data segment）的头部可能会被映射两次，这也不会有问题。因为代码中对数据引用的地址已经考虑了这个情况，并不会用文件中的偏移量来计算地址，只需要把多余的代码和多余的数据作为字节填充即可。最后的数据页面也许会包含与正在运行的进程无关的文件信息。

7.3.2 DSO 如何工作

1. GOT（Global Offset Table，全局偏移量表）

动态库对外提供一些函数以供可执行程序使用。由于动态库只在可执行程序运行时被动态地加载到可执行程序的地址空间，并且被加载的地址不确定。因此在静态链接时，可执行程序并不知道需要调用的动态库的函数的实际地址。在 7.2 节中介绍了 Windows DLL 引入了 IAT（Import Address Table）表，代码可以从 IAT 表中取得目标函数的地址，而 IAT 表在动态链接时被更新。通过这种间接的方式来获取目标函数的地址，可以避免产生代码段的重定位。Linux 的共享对象（DSO）也采用类似的方法。不过，在 Linux 的可执行程序

（包括共享对象）代码中，如果有对全局对象的引用，都采取一种间接的方式，而不论该全局对象是否在本模块中被定义（可以通过一些方法来避免这种情况，后文将有介绍）。

和 Windows DLL 中的 IAT 表类似，GOT 表中保存所有全局对象的实际地址。假如程序需要直接访问某个全局符号，那么这个符号将有一个 GOT 表项，编译后产生的代码也是通过间接的方式（通过相关的 GOT 表项）取得该全局符号的实际地址。由于全局对象的实际地址需要动态链接后才能得知，因此相关的 GOT 项都有一个可重定位项与之对应。初始时，GOT 表保存重定位时所需要的信息。当系统加载所需的共享对象后，动态链接器会处理重定位项，那些类型为 R_386_GLOB_DAT 的重定位项就是与 GOT 表相关的重定位项。动态链接器根据相关信息计算出相关符号的绝对地址，并且更新 GOT 相关的表项为正确的值。在交给进程映像代码控制权以前，动态链接器需要处理所有 GOT 相关的重定位。

GOT 表的 0 号表项（即第 1 个表项）是为保存动态结构（dynamic structure）地址保留的，动态结构在后面将介绍。在 32 位 Intel 系统结构中，在 GOT 中的 1 号和 2 号表项也是保留的，具体看后面的过程链接表（PLT Procedure Linkage Table）。

对于如下一段对全局对象进行引用的代码：

```
extern int global_var;
extern int global_func(int);
int call_func (void) {
    return global_func(global_var);
}
```

用-fpic 或者-fPIC 选项编译成 DSO，产生的相应代码如下：

```
movl  global_var@GOT(%ebx), %eax
push  (%eax)
call  global_func@PLT
```

全局变量 global_var 的地址从 GOT 表中得到，它在 GOT 表中表项的地址与当前指令的地址（PC 寄存器，%ebx）的相对值（即 global_var@GOT）在静态链接时已确定，而全局函数 global_func 的地址从 PLT（Procedure Linkage Table）中得到。

2. PLT

一个在 32 位 Intel 系统结构下共享对象的 PLT 如下所示：

```
.PLT0: push1  4(%ebx)
       jmp    *8(%ebx)
       nop; nop
       nop; nop
.PLT1: jmp    *name1@GOT(%ebx)
       push1  $offset
       jmp    .PLT0@PC
.PLT2: jmp    *name2@GOT(%ebx)
       push1  $offset
       jmp    .PLT0@PC
       …
```

这里只展示了 3 个 PLT 项，PLT 中的表项可以任意多。在 32 位 Intel 系统结构下，每个表项的大小相同，为 16 个字节。并且%ebx 寄存器必须存储 GOT 表的首地址，这由函数的调用者负责在函数调用前来设置。第 1 个表项 PLT0 比较特殊，下面会有详细介绍，其余的表项从组成上看非常类似。第 1 条指令都是一个非直接跳转指令，跳转的地址是从 GOT 表中取得，即每个 PLT 项都有一个 GOT 项与之对应。当可执行程序（包括共享对象）被加载到内存时，动态链接器把每个 PLT 项所对应的 GOT 项中的值设置成该 PLT 项中第 2 条指令的地址。即当第 1 次执行 PLT 项的第 1 条非直接跳转指令时，从 GOT 中取得的跳转地址就是该 PLT 项的第 2 条指令。

执行该跳转指令后，就会执行 PLT 项的第 2 条指令。该指令是一个压栈指令，它把一个重定位项的偏移量压到栈中，该重定位项的偏移量是一个 32 位且非负的字节偏移量（从重定位表头算起）。这个重定位项是一个 R_386_JMP_SLOT 类型，它指出 GOT 中的某个项（该项正是这个 PLT 项所对应的 GOT 项）需要重定位，并且该重定位项也包含一个符号表的索引，因此可以告诉动态链接器哪个符号要被引用（这个符号就是与该 PLT 相关的全局符号，比如程序中调用了全局函数 global_func，最终目标代码是通过 PLT 来间接调用。该全局函数在 PLT 中有一个表项，该表项中压栈指令指定的重定位项中所指定的符号就是全局函数符号 global_func）。

在压入一个重定位项的偏移量后，程序跳到.PLT0，这是 PLT 表中的第 1 个 PLT 项。首先，把 GOT 表的第 2 个表项压入栈，这个 GOT 表项存放一个动态链接器需要的鉴别信息。然后程序跳转到 GOT 表的第 3 个表项所指定的地址，该地址是动态链接器的程序入口地址。

接下来，动态链接器根据压入栈中的重定位项所指定的符号确定该全局符号的实际地

址。并根据重定位信息，把这个实际地址存储到相应的 GOT 表项中，并且跳转到全局函数的实际地址执行该全局函数。以后对该 PLT 项的使用就不需要经过动态链接器来查找实际地址，而从对应的 GOT 表项中取得函数的实际地址，直接跳转到函数的实际地址。因此通过 PLT 来调用全局函数仅仅比使用绝对地址来调用函数多出一次非直接跳转指令（第 2 次和以后的调用），但是它消除了对代码段进行重定位。

根据上面的介绍，PLT 表中的代码在动态链接时是不需要修改的，因此 PLT 表一般处于只读的段中。需要在动态链接时被修改的是 GOT 表，GOT 表中存储的基本上都是地址，在 32 位 Intel 系统结构下，每个 GOT 表项的大小为 4 个字节。

3．位置无关的代码

共享对象的一个优点是它可以被不同的进程加载到不同的地址，但是代码中有很多对全局对象和函数的引用。如何确保代码段可以被加载到不同的地址而不需要修改。位置无关的代码（Position Independent Code，PIC）可以实现，PIC 是可以被加载到任意地址并不需要加载器或链接器修改即可执行的代码。

在一个 IA32 系统中。对同一模块中非全局的函数调用不需要特殊处理，因为调用处和被调用函数入口的偏移量是确定的，可以使用 PC（Program Counter，程序计数器）相关的调用，这已经是 PIC 代码了。然而对全局变量的引用，对外部函数的调用如何实现 PIC？可以使用 GOT 来实现对全局对象的引用。GOT 是静态链接时创建的，并且每个可执行程序（包括共享对象）只有一个 GOT 表，每个全局变量在 GOT 表中都有一个表项。ELF 的设计者发现全局变量的引用处与该全局变量在 GOT 表中的表项之间的偏移量是固定的，并且在静态链接时就可以确定，因此可以采用如下的 PIC 代码来实现对全局变量的引用：

```
    call   L1
L1: popl   %ebx;
    addl   $VAROFF, %ebx
    movl   (%ebx), %eax
```

前两条指令的作用是把 PC 值设置到寄存器%ebx 中，第 1 条指令对 L1 的调用会将返回地址（正好是 popl 指令的地址）压入栈；第 2 条指令把刚压入栈的地址弹出到寄存器%ebx 中。因此%ebx 中存储的正是 popl 指令的地址，也就是当前指令的地址（PC 的值）；第 3 条指令为%ebx 增加一个固定偏移量，使得它指向 GOT 中适当的表项。该表项中包含了全局数据项的绝对地址，这时就可以通过 GOT 表项来间接引用全局变量。

对于外部函数的调用，则采取 PLT 来解决。PLT 也是在静态链接时生成的，并且每个外部函数在 PLT 表中都有一项。外部函数调用处与该函数在 PLT 中的表项之间的偏移量也是固定的，因此可以采用 PC 相关的调用来调用 PLT 的相关表项的代码，如下面代码所示：

```
    call   L1
L1: popl   %ebx
    addl   $GOTOFF, %ebx
    call    $FUNCOFF@PLT
```

与通过 GOT 来引用全局变量一样，前两条指令的作用也是把 PC 值设置到寄存器%ebx 中。第 3 条指令中$GOTOFF 代表当前指令与 GOT 表头部地址之间的偏移量，该值是固定的。通过前 3 条指令可以得到 GOT 表的绝对地址，并存储在%ebx 中。这是因为 PLT 表中的代码也是位置无关的代码，在生成 PLT 代码时缺省认为寄存器%ebx 保存 GOT 表的绝对地址。

图 7-10 所示为 PIC 的实现原理。

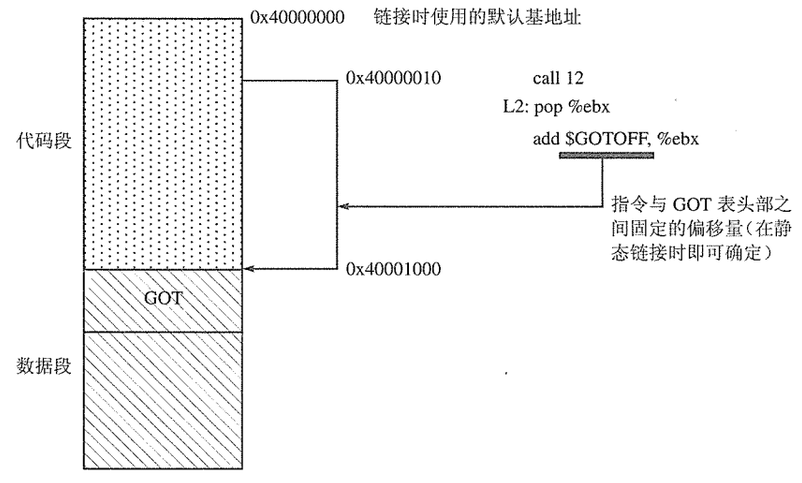


图 7-10　PIC 的实现原理

4．动态链接过程

操作系统执行一个依赖于多个共享库的可执行程序的大致过程如下。

（1）将可执行程序的所有可加载的段映射到进程的地址空间。

（2）将所依赖共享对象的所有可加载的段也映射到进程的地址空间。

（3）为可执行文件和它所依赖的共享对象进行重定位。

（4）向程序传递控制，即跳转到可执行程序的入口。

当操作系统运行程序时，首先将可执行程序的所有可加载的段映射到进程的地址空间。注意可执行文件中有一个 PT_INTERP 段，该段的 p_offset 域指出一个以 NULL 结尾的字符串，这个字符串指定一个解释器的文件名。即动态链接器 ld.so，ld.so 本身就是 ELF 格式的共享库。当把动态链接器映射到一个合适的地址后，就可以启动 ld.so。动态链接器需要知道可执行程序的信息及执行动态链接任务后把控制传递到何处，操作系统会传递一个辅助向量（auxiliary vector）到堆栈。辅助向量（辅助信息）包括如下内容。

（1）AT_PHDR，AT_PHENT 和 AT_PHNUM：可执行程序的程序头部表（program header table）的地址，及其中表项的大小和数目。如果系统还没有把程序映射到内存，那么就会是一个 AT_EXECFD，它包含程序文件打开后的文件描叙符。

（2）AT_ENTRY：程序开始地址，动态链接器在初始化后就会跳到该地址。

（3）AT_BASE：动态链接器被装载的地址。

此时，ld.so 入口部分的自引导代码首先找到自己的 GOT。GOT 的第 1 个表项指向 ld.so 文件中的动态结构（dynamic structure），这对于动态链接器是重要的，因为它必须初始化自身，而不依赖于其他程序来重定位它在内存中的映像。通过这个动态结构，动态链接器可以找到自己的重定位入口，在自己的数据段中重定位地址。并且解析装载其他符号的基本例程的代码的位置，之后动态链接器会把程序符号表和链接器符号表组成为一个符号表链。

动态结构（dynamic structure）是一个数组，数组中每个元素的数据结构如下：

```
typedef struct {
    Elf32_Sword  d_tag;
    Union  {
          Elf32_Word    d_val;
          Elf32_Addr    d_ptr;
    } d_un;
} Elf32_Dyn;

extern Elf32_Dyn DYNAMIC[];
```


每个元素中 d_tag 的值控制了 d_un 的解释，常用的 d_tag 值及 d_un 解释见下表。

d_tag 名称	d_tag 值	d_un 使用	d_un 解释
DT_NEEDED	1	d_val	该元素保存以 NULL 结尾的字符串在字符串表中的偏移量，这个字符串代表这个 ELF 文件所依赖库的名字
DT_SONAME	14	d_val	该元素保存以 NULL 结尾的字符串在字符串表中的偏移量，这个字符串代表这个共享对象的名字
DT_RPATH	15	d_val	该元素保存以 NULL 结尾的字符串在字符串表中的偏移量，这个字符串代表搜索库的搜索路径
DT_SYMTAB	6	d_ptr	该元素保存符号表的地址
DT_HASH	4	d_ptr	该元素保存符号哈希表的地址
DT_RELA	7	d_ptr	该元素保存重定位表的地址，这个重定位表中的表项有明确的加数。假如该元素存在，动态结构必须也要有 DT_RELASZ 和 DT_RELAENT 元素
DT_RELASZ	8	d_val	该元素保存 DT_RELA 指定的重定位表的大小，以字节来衡量
DT_RELAENT	9	d_val	该元素保存 DT_RELA 指定的重定位表中表项的大小，以字节来衡量

除了以上值外，还有与 DT_RELA 类似的 DT_REL、相关的 DT_RELSZ、DT_RELENT，以及其他值。该动态结构以一个 d_tag 值为 DT_NULL(0)的元素结束。

如果一个可执行程序（包括共享对象）需要动态链接其他的共享对象，它必须有一个类型为 PT_DYNAMIC 的段。该段包含一个.dynamic 节（section），其中的内容是上述动态结构。当动态链接器准备好后，它就开始查找程序所需要的共享库的名字。它从可执行程序的程序头部表中找到 PT_DYNAMIC 段，最终从该动态段（即动态结构）中找到 DT_NEEDED 元素，得到所需要的库名。

链接器采用如下方式寻找所需要的库。

（1）查看动态段是否包含一个称为“DT_RPATH”的项（它是一个以冒号分隔的库文件搜索目录表），这个项是在程序被静态链接时由命令行选项或者环境变量添加的，如果存在该项，搜索动态库是否在 DT_RPATH 的项指定的目录中。

（2）查看是否存在环境变量 LD_LIBRARY_PATH（它是一个以冒号分隔的库文件搜索目录列表），搜索动态库是否在 LD_LIBRARY_PATH 的项指定的目录中。

（3）链接器查看/etc/ld.so.conf 文件，它包含库名和路径的一个对应列表。如果库名存在，链接器使用它对应的路径，用这个查找方法能够找到大部分库。

（4）如果上述的查找失败，链接器查找默认路径/usr/lib。如果库文件依旧没有找到，则显示一个错误后退出。

链接器找到共享对象后，可以从中读取 ELF 头以及程序头部表并找到各个段的位置，然后链接器为共享对象的代码段和数据段分配空间并映射到内存。接着通过共享对象的动态段，链接器把这个共享对象的符号表添加到符号表链中。如果该共享对象所依赖的其他共享对象没有被加载到进程的地址空间，则添加共享对象到加载队列中。

链接器依次对加载队列中的每个共享对象进行处理，完成这个过程后，所有库都已经被映射。动态链接器在逻辑上拥有了一个全局的符号表，它是全部程序和被映射库的符号表的联合。此时链接器再次访问每个共享对象，对每个共享对象进行重定位，并填充共享对象的 GOT。如果一个库有.init 节，则其作为库的特定初始化例程被动态链接器调用（C++中的全局对象的构造在放置.init 中）。当上面的工作完成以后，所有的共享对象都已经准备好。链接器调用可执行程序的开始地址（AT_ENTRY），程序开始执行。

7.3.3 构建与使用 DSO

在 7.2.1 节中介绍了创建 Winodws DLL 的过程，构建一个 Linux 系统的动态共享对象（DSO）与创建一个 Windows DLL 的过程类似。首先需要确定共享对象对外提供的功能，即该共享对象对外提供的 API。该 API 一般定义在一个或多个头文件中，包含对外提供的全局函数和变量的声明。接着编写 C/C++文件来实现这些功能，并单独编译每个源代码文件成目标代码文件，最后把这些目标代码文件静态链接成共享对象。为此需要在链接时，把-shared 链接选项传给链接器。如果源代码依赖某些静态库，链接时也需要把这些库输入到链接命令中。下面是产生一个简单的共享对象的例子：

```
gcc -fpic -o a.o a.c
gcc -fpic -o b.o b.c
gcc -shared -o mysh.so a.o b.o
```

首先，单独编译 a.c 与 b.c 为目标文件 a.o 与 b.o。注意，编译时需要传入-fpic 选项，告诉编译器生成位置无关的代码。位置无关的代码可以被加载到地址空间的任意位置而不需要修改，这符合共享对象的特性，因此构成共享对象的源文件需要被编译为位置无关的代码。然后执行链接命令，为链接器传入-shared 选项，把目标文件 a.o 与 b.o 链接为共享对象 mysh.so。与链接生成 DLL 不同，链接器并不会产生一个对应的导入库（import lib）以

供依赖该动态库的其他模块使用。如果可执行程序或者其他共享库的创建过程中需要依赖这个共享库，应如何链接？直接把该共享对象放置到链接命令中即可，如下例所示：

```
gcc -fpic -shared -o my2sh.so m.c n.c mysh.so
```

如果 m.c 中使用到了 mysh.so 中提供的功能，在创建由 m.c 与 n.c 构成的共享对象 my2sh.so 时，只需要把共享库 mysh.so 放置在链接命令中，链接器在链接时会把相关信息（产生的共享对象依赖于 mysh.so 的信息）写入到 my2sh.so 中。

作为共享库的 DSO，主要是为了实现软件的重用。理想情况下，系统只应该存在某个共享对象的一个版本，它被保存在系统的某个目录中。同时，共享对象也会因为修改或功能升级而产生不同的版本，这有可能发生链接产生可执行程序时所用的共享对象的版本与运行可执行程序时动态链接的共享对象的版本并不一致。为了处理这种情况，Linux 对共享对象使用搜索名和实名。每个共享库都有一个特定的搜索名（soname），搜索名的命名约定如下：

```
lib  +  库名  +  .so  +  .  +  version
 |      |       |_____________|
前缀    库名          后缀
```

在 7.3.2 节中介绍了的动态结构，其中有两项 DT_NEEDED，DT_SONAME。它们保存搜索名，DT_SONAME 代表共享对象的搜索名；DT_NEEDED 代表所依赖的共享对象的搜索名。链接器通过搜索名来寻找相关的共享对象。在文件系统中，搜索名是一个指向实名的符号连接。每个共享库也有一个实名，组成如下：

```
搜名  +  .  +  子版本号  +  .  +  发布号
```

最后的句点和发布号是可选项。因此链接产生一个共享对象时，比较正式的方法如下：

```
gcc -shared -WI, -soname, libmysh.so.1 -o libmysh.so.1.0.1 a.c b.c
```

libmysh.so.1 是新产生的共享对象的搜索名，libmysh.so.1.0.1 是实名，即共享对象的文件名。当构建新版本的共享对象时，只要保持库的搜索名不变，链接器就可以顺利匹配。

共享对象的使用与 DLL 的使用类似，也可以分为隐式调用和显示调用。如果通过隐式方式使用共享对象所提供的功能（全局函数或变量），可以在源代码中与使用本模块的函数或变量一样使用外部共享对象定义的全局函数或变量。不过在静态链接时需要在链接命令中加入被依赖的共享库的名称，以完成符号解析并建立依赖关系。上面的例子即通过隐式

方式来使用共享对象。可执行程序通过隐式方式使用共享对象，当其被加载时，动态链接器会自动加载和链接这些共享对象。

除了隐式调用外，Linux 为链接器提供了一个简单的接口，允许应用程序在运行时加载和链接共享对象，而不需要在编译时链接这些共享对象。

（1）dlopen()。

函数原型如下：

```
void * dlopen(const char *filename, int flag);
```

功能描述：dlopen 函数可以打开一个共享对象，然后为后面的使用做准备。如果要加载的库依赖于其他库，必须首先加载依赖库。dlopen 操作失败，返回 NULL 值。如果库已经被加载，则 dlopen 会返回同样的句柄。同一个共享库可以被一个进程多次加载，每加载一次，共享库的句柄使用数加 1。

参数中的文件名 filename 如果以“/”开头，表示使用的是绝对路径。dlopen 直接使用它，而不查找某些环境变量或者系统设置的共享库所在的目录；否则 dlopen 就会按照下面的顺序查找共享库。

- 根据环境变量 LD_LIBRARY_PATH 查找。
- 根据/etc/ld.so.cache 查找。
- 依次在/lib 和/usr/lib 目录查找。

在 dlopen 函数中，参数 flag 的值必须是 RTLD_LAZY 或者 RTLD_NOW。RTLD_LAZY 表示暂时不处理未解析的全局符号，而首先把共享库加载到内存，等使用到未解析的全局符号时解析（这正是 ELF 中 PLT 所提供的特性）；RTLD_NOW 表示需要在 dlopen 返回前解析所有未定义的全局符号，如果未能解析所有的未定义的符号，则 dlopen 以失败告终（为此需要重定位与 PLT 相关的所有 R_386_JMP_SLOT 类型的重定位项）。

（2）dlerror()。

函数原型如下：

```
char *dlerror(void);
```

功能描述：dlerror 可以获得最近一次 dlopen，dlsym 或 dlclose 操作的错误信息，返回

NULL 表示无错误。dlerror 在返回错误信息的同时，也会清除错误信息。

（3）dlsym()。

函数原型如下：

```
void *dlsym(void *handle, const char *symbol);
```

功能描述：在 dlopen 之后，共享库已被加载到内存。参数 handle 是使用 dlopen 得到的共享对象的句柄。dlsym 可以获得指定函数（symbol）在内存中的位置（指针），如果找不到指定函数，则返回 NULL 值。

（4）dlclose()。

函数原型如下：

```
int dlclose(void *);
```

功能描述：将已经装载的共享库的句柄使用数减 1。当句柄使用数减至 0 时，则该库会被卸载。

源代码中使用这些接口需要包含 dlfcn.h 头文件，并且在编译时需要使用-ldl 链接选项，例如：

```
gcc -o myExe -ldl myMain.c
```

在构建与使用 DSO 时应注意如下方面。

（1）Windows 系统与 Linux 系统对全局符号的不同处理。

在全局符号的处理上，Windows 系统与 Linux 系统的观点不一致。Linux 系统的 ELF 文件对每个可执行程序（包括使用的动态库）使用一个名字空间，而 Windows 系统的每个可执行文件（包括动态库）都有一个独立名字空间。Linux 系统在产生 ELF 格式的可执行程序或共享库的过程中，当遇到一个全局符号引用时，不管该全局符号来自于其他模块，还是在本模块中定义，静态链接器并不会把该符号引用与某个具体的符号定义对应起来。而 Windows 系统在产生 PE 格式的可执行程序或动态链接库的过程中，遇到一个全局符号引用。如果该全局符号在构成可执行程序或动态链接库的目标文件中有定义，则用这个符号定义来解析该全局符号引用；如果该全局符号在构成可执行程序或动态链接库的目标文件中没有定义，则从链接命令中相关的动态库的导入库中查询，查找到后建立全局符号与

输入动态库的对应关系。

一个 ELF 程序（包括 DSO）知道其所需要的符号和库，但是并没有记录符号和库的对应关系。当运行时，这会带来很大的灵活性，例如：

```
//main.c
extern int getFromA();
extern int getFromB();
extern void printFromSO();

int main()
{
  int a = getFromA();
  int b = getFromB();
  printFromSO();
  return 0;
}
//a.c
#include <stdio.h>
int getFromA()
{
  return 10;
}

void printFromSO()
{
  printf("print from a.so \n");
}
//b.c
#include <stdio.h>
int getFromB()
{
  return 20;
}

void printFromSO()
{
  printf("print from b.so \n");
}
```


首先，用下列指令分别创建两个共享对象：

```
gcc -shared -fpic -o a.so a.c
gcc -shared -fpic -o b.so b.c
```

a.so 和 b.so 分别对外提供 getFromA、printFromSO 和 getFromB，以及 printFromSO 功能，接着创建一个使用这两个共享对象的可执行程序 myExe：

```
gcc -o myExe main.c a.so b.so
```

产生的可执行程序 myExe 含有 3 个外部的全局符号（getFromA、getFromB 和 printFromSO），但是 myExe 并没有指明每个符号来自于哪个共享库（如 printFromSO 来自于共享库 a.so）。

可执行程序在控制台上运行：

```
./myExe
```

输出结果是：

```
print from a.so
```

这是因为动态链接时，在符号的搜索路径中 a.so 出现在 b.so 之前（产生 myExe 的命令中，a.so 出现在 b.so 之前）。如果按如下所示，首先定义系统变量 LD_PRELOAD，再运行程序，

```
export LD_PRELOAD= b.so
./myEXE
```

则相应的输出结果是：

```
print from b.so
```

这是因为使用系统变量 LD_PRELOAD=b.so，会使符号的搜索路径中 b.so 出现在 a.so 之前。ELF 对全局符号的处理带来灵活性的同时，也会给用户带来一些迷惑。想象一个可执行文件调用一个例程 FUNC_A 和 FUNC_B，FUNC_A 在共享对象 A 中定义，FUNC_B 在共享对象 B 中定义。如果一个新版本的共享对象 A 恰好定义了 FUNC_B，并且符号的搜索路径中共享对象 A 出现在共享对象 B 之前，则 ELF 程序会优先选择新的在 A 中定义 FUNC_B。

另一个方面，Windows 可执行程序如果使用了外部全局符号，该 PE 文件（即可执行

程序）指明了某个符号来自于某个库，这使得 PE 缺乏灵活，但是对不经意的欺骗有更大的抵抗力。

（2）本地定义的全局符号的处理。

全局符号的解析在 Linux 系统中不区分是否在本模块中定义，即使模块引用了同一模块中已定义一个全局符号，在运行时该符号也可能会被解析为其他模块中的定义。例如：

```
//main.c
extern int getFromA();
extern int getFromB();
extern void printFromSO();

int main()
{
  int a = getFromA();
  int b = getFromB();
  printFromSO();
  return 0;
}
//a.c
#include <stdio.h>
int getFromA()
{
  return 10;
}

void preprint()
{
  printf("preprint from a.so \n");
}

void printFromSO()
{
  preprint();
  printf("print from a.so \n");
}
//b.c
#include <stdio.h>
int getFromB()
```



```
{
  return 20;
}

void preprint()
{
  printf("preprint from b.so \n");
}
```

与上例一样，首先编译产生 a.so 与 b.so。注意，这个版本的 b.so 中不包含 printFromSO 函数，而 a.so 与 b.so 都新增加了一个 preprint 全局函数。再利用这两个共享对象，同样编译产生可执行程序 myExe。在控制台上运行产生如下结果：

```
preprint from a.so
print from a.so
```

如果使用系统变量 LD_PRELOAD=b.so，使得符号的搜索路径中 b.so 出现在 a.so 之前，则运行结果如下：

```
preprint from b.so
print from a.so
```

从上面结果可知，即使 a.so 中的定义了 preprint 函数，运行时函数 printFromSO 所调用的 preprint 也可能来自其他共享对象（这里是 b.so）。

在 ELF 中，对全局符号的引用一般都转换为对全局符号地址的引用。全局符号的地址放置在 GOT 表中，这些表项需要在运行时重定位，因此全局符号的地址只有在运行时才能确定。不但一个共享库中的全局符号可以抢先匹配另一个共享库中已定义的全局符号，可执行程序也可以（而且常常）抢先匹配共享库中已定义的全局符号。例如：

```
//main.c
#include <stdio.h>
extern void printFromSO();

int main()
{
  printFromSO();
  return 0;
}
```

```
void preprint()
{
  printf("preprint from main\n");
}
//a.c
#include <stdio.h>
void preprint()
{
  printf("preprint from a.so \n");
}

void printFromSO()
{
  preprint();
  printf("print from a.so \n");
}
```

如下使用 gcc 分别编译产生 a.so 和 myExe：

```
gcc -shared  -fpic -o a.so a.c
gcc -o myExe main.c a.so
```

在命令行上运行，结果如下：

```
preprint from main
print from a.so
```

由于可执行程序是全局符号的搜索路径上的第 1 个模块，因此 a.so 中的全局符号 preprint 被解析为可执行程序中的 preprint 全局符号的定义。

但是把目标文件编译为一个可执行程序而非共享对象时，编译器可能会做一些优化。例如：

```
//main.c
#include <stdio.h>
extern int getFromA();
extern int getFromB();
extern void printFromSO();

int main()
{
  int a = getFromA();
```



```
  int b = getFromB();
  printFromSO();
  return 0;
}

void printFromSO()
{
printf("print from main\n");
}
```

因为编译器知道可执行程序是全局符号的搜索路径上的第 1 个模块，所以如果某个全局符号在该模块中已经有定义，则直接把该符号引用与符号定义关联起来。全局函数 printFromSO 在本模块中已有定义，并且编译指令明确指出最终产生可执行程序，因此编译器可以优化产生代码。对全局符号 printFromSO 的引用可以不通过 GOT 表来间接引用，直接用本地的定义来解析。

如果用户确实想把共享库中的某个全局符号绑定到本模块中相应的符号定义，可以在链接产生共享对象时使用-Bsymbolic 链接选项，这样产生的共享对象的动态结构（dynamic structure）中会增加一个 d_tag 值为 DT_SYMBOLIC 的元素。如果该元素在共享对象中出现，将会改变动态链接器对全局符号的解析算法。对于该共享对象中的全局符号的搜索路径将不把可执行程序放在搜索路径中的第 1 位置，而把这个共享对象放置在搜索路径中的第 1 位置。如果该共享对象中没有相应的全局符号的定义，动态链接器再按照通常情况来搜索可执行程序和其他的共享对象。

（3）控制全局符号的输出。

从前面的介绍可以得知，构造一个共享对象时，如果链接指令和源代码中没有特别的声明，全局符号都会进入共享对象的动态符号表，可以被其他共享对象或可执行程序引用。但是过多的全局符号暴露给外部可执行模块，增加共享对象的维护难度。并且由于动态符号表中全局符号只有在运行时才能被解析，不仅会增加可执行程序的启动时间，而且在动态的解析过程中，即使打算引用同一模块中的全局符号的定义，该全局符号的定义也可能被其他模块中的定义所屏蔽，因此必须要减少共享对象的全局符号的输出。

可以采用如下方法来减少全局符号的输出。

- 使用 static 修饰符来修饰函数或者变量。

定义为 static 的函数或者变量的可见范围是文件范围，因此当编译器把源代码文件编译成目标代码文件时，会把 static 函数和 static 变量的引用与同一文件的该函数和变量的定义关联起来。这样可以减少全局符号的数量，从而减少由于需要解析全局符号而产生的相关措施。比如，对于全局变量可以去掉相关的 GOT 表项，对于全局函数可以去掉相关的 PLT 表项，同时消除了相关的运行时重定位信息。

- 定义符号的全局可见性。

最新的 ELF ABI 中定义了符号的可见性，每个符号都有一个可见性属性。可见性分为 4 个种类，它们是 STV_DEFAULT、STV_INTERNAL、STV_HIDDEN 及 STV_PROTECTED。其中可见性的值为 STV_DEFAULT 表示符号的可见性是正常的，它可以被输出，并且可以被其他模块中的相同符号所屏蔽；值为 STV_PROTECTED 表示该符号可以被输出，但是如果本模块已有定义，则使用本模块的定义；值为 STV_HIDDEN 则指出该符号不可以被输出，即其他模块不可以访问该符号。通过把符号的可见性设置为 STV_HIDDEN 可以控制全局符号的输出。

gcc 4.0 中提供了-fvisibility 命令行选项来设置符号的可见性，例如：

```
gcc -shared  -fpic -fvisibility=hidden -o a.so a.c
```

不过，使用该选项会影响所有符号的可见性。当–fvisibility=hidden 出现在 gcc 的命令行中时，gcc 会把除了特别指明可见性符号外的所有符号的可见性设置成 STV_HIDDEN，即对外不可见。因此使用该选项时，一定要显式指明需要输出哪些符号；否则生成的共享对象不对外输出任何符号，也就不能对外提供任何功能。

如何显式指明符号的可见性？由于 C/C++语言中没有相应的机制，所以只能使用 gcc 中的属性（attribute）机制。例如：

```
int __attribute__((visibility("default"))) some_exported_function(void);
int __attribute__((visibility("hidden"))) some_hidden_variable;
```

除了可以使用属性来说明每个符号的可见性，gcc 4.0 还可以使用 pragma 来控制一段符号声明或定义的可见性。这种方法可以在头文件中使用，使得头文件看起来比较简洁。例如：

```
#pragma GCC visibility push(default)
int exported_function_one(void);
int exported_function_two(void);
```



```
int exported_function_three(void);
#pragma GCC visibility pop

#pragma GCC visibility push(hidden)
int hidden_function_one(void);
int hidden_function_two(void);
int hidden_function_three(void);
#pragma GCC visibility pop
```

在源代码中使用属性（attribute）或 pragma 指明符号的可见性为 hidden，除了可以使编译器在目标代码的符号表中设置相应符号的可见性值为 STV_HIDDEN 外，还可以告诉编译器在本模块中对该符号的引用使用本地定义来解析。编译器可以产生与 static 符号类似的寻址方案，同样可以消除由于解析全局符号而产生的相关操作。

- 使用输出表（export map）来控制全局符号的输出。

输出表是一些传递给静态链接器的信息，它告诉链接器目标文件中的哪些符号可以输出。在输出表中，每个符号只有一种属性，或者可以输出，或者不可以输出。在输出表中可以列出每个符号的名称，也可以使用通配符。输出表如下所示：

```
{
    global:
        exported_function_one;
        exported_function_two;
        my_exported_fun*;
    local:
        *;
}
```

该输出表告知链接器符号 exported_function_one，符号 exported_function_two，以及以 my_exported_fun 开头的全局符号是需要输出的符号，其余符号都是本地符号，不可以对外输出。在创建输出表时，一般建议列出所有需要输出的符号，使用通配符“*”来指出除了前面指明的输出符号外，剩余的全局符号都是本地符号。这样既可以避免遗漏任何全局符号(未指明该符号是输出符号还是本地符号)，也可以防止不小心把某个符号即放置到 global 区又放置到 local 区中。注意，如果输出表中列出的是 C++的符号名，应该列出它们被装饰（mangled）的名称。

使用输出符号表可以方便和统一地管理全局符号的输出，避免了源代码中到处充斥 gcc

的属性声明或 pragma 声明。使用输出符号表来控制符号的输出是发生在链接阶段，链接器不会把声明为本地的全局符号加入到动态符号表中（这样这些符号就不会被输出，同时它们也不会被其他模块的同名符号所屏蔽），同时一些相应的重定位信息也不会放置到最终的共享对象中。但是在链接阶段，一般链接器对目标系统知之甚少。因此链接器不会修改编译阶段产生的代码，相关的优化（针对本地符号的优化）也就不可能发生。比如，引用全局变量需要通过 GOT 表来间接引用（从 GOT 表中获取全局变量的地址，再通过地址来访问全局变量）。如果确定该全局变量的引用会被解析成本地定义，则在编译阶段不需要产生通过 GOT 表来间接引用的指令。

总之，在构造共享对象是需要尽量减少全局符号的输出。在控制符号输出方法的选择上，优先选择前面两种方法，以第 3 种方法作为补充。

7.4 本章小结

动态库可以在运行时被加载到应用程序的任意地址空间，运行时多个进程可以共享一个动态库。并且动态库进行版本升级时不需要重新编译应用程序，这些特性使得动态库成为构造大型软件的基石。本章介绍了动态库和动态链接的原理及相关基础知识，并以 Windows DLL 和 Linux DSO 为例介绍了动态库的文件结构、与动态链接相关的数据结构，以及动态链接的原理。但是一个软件包括过多的动态库也会带来一些副作用，比如降低应用程序的启动性能（第 8 章中详细分析了动态库对应用程序启动性能的影响）。当软件被划分成多个模块开发时，在开发阶段每个模块可以产生一个或多个动态库。但是如果构建的动态库并不会被重用，在软件发布时应该尽量减少动态库的数量，把那些没有重用价值的动态库合并到可执行程序中。

第 8 章 程序启动过程

应用程序的启动是一个比较复杂的过程，其中包含操作系统和应用程序自身要完成的一系列工作，同时涉及 CPU 和 IO 方面的操作。本章通过剖析应用程序的启动过程来探讨影响应用程序启动性能的因素。

本章首先简要介绍在 Win32 和 Linux 平台上应用程序从编译链接到加载启动的过程。在此基础上，引入“冷启动”（cold startup）和“热启动”（warm startup）性能概念的定义。之后分别探讨影响冷启动和热启动性能的相关因素。

通过阅读本章，读者对应用程序的启动过程会有详细的了解，并且从原理上理解不同因素对于程序启动性能的影响。

8.1 Win32 程序启动过程

在现代操作系统中，计算机应用程序以文件形式（包括可执行文件、动态库文件、配置文件和其他文件等）保存在磁盘中。所谓“程序启动过程”，指从用户发出请求执行程序，到该应用程序完全启动（例如程序界面完全显示，准备响应用户的输入）这个过程。从系统分工的角度来看，程序启动过程包括如下两个阶段。

（1）操作系统负责把程序从磁盘读入内存并且建立相应的运行环境。

（2）应用程序自身的初始化过程。

正如第 7 章描述的那样，一个 Win32 的可执行程序（Executable program，即通常所见的.exe 文件）要符合 PE 文件格式，主要包括数据段（.data）和代码段（.text）。在 IA32 架构的计算机中应用程序的代码和数据在内存中，CPU 从程序入口处开始取出每一条指令，按序执行。

在内存中，应用程序的代码表示为一系列有序的指令集合，每一条指令占据内存的一定单元（例如在内存地址为 0040103A 的单元中存储一条“call printf (00401060)”指令，即调用 00401060 处的函数）；应用程序的数据（全局变量和静态变量等）同样占据内存的一定区域，对这些数据的引用则转换为存放该数据内存区域的地址（例如在地址 0040102F 处的指令“mov eax,[gData (00424d8c)]”。要把 gData 的值放到 eax 寄存器中，此处引用全局变量 gData 时使用存放该全局变量的内存地址 00424d8c）。而操作系统的加载器（loader）的任务就是把磁盘中的可执行程序的物理文件读入内存并且转换为程序在内存中的表示。

下图为一个简单的示例程序，它在内存中指令和数据排列如下：

```
00400000   dec        ebp  //程序入口点，可执行程序默认加载地址为 00400000
00400001   pop        edx
...
@ILT+0(_main):
00401005   jmp        main (00401010) //main 函数
0040100A   int        3
0040100B   int        3
0040100C   int        3
0040100D   int        3
```



```
0040100E   int        3
0040100F   int        3
--------- hello.cpp  ---------------
1:    // hello.cpp
2:    //
3:
4:    #include "stdafx.h"
5:
6:    int gData = 24;
7:
8:    int main(int argc, char* argv[])
9:    {
00401010   push       ebp
00401011   mov        ebp,esp
00401013   sub        esp,44h
00401016   push       ebx
00401017   push       esi
00401018   push       edi
00401019   lea        edi,[ebp-44h]
0040101C   mov        ecx,11h
00401021   mov        eax,0CCCCCCCCh
00401026   rep stos    dword ptr [edi]
10:       int lData = 36;
00401028   mov        dword ptr [ebp-4],24h  //局部变量赋值
11:       printf("Hello World! gData is %d \n",gData);
0040102F   mov        eax,[gData (00424d8c)] //引用全局变量
00401034   push       eax
00401035   push       offset string "Hello World! gData is %d \n" (00422fac) //
引用字符串
0040103A   call       printf (00401060) //调用库函数
0040103F   add        esp,8
12:       return 0;
00401042   xor        eax,eax
13:   }
00401044   pop        edi
00401045   pop        esi
00401046   pop        ebx
00401047   add        esp,44h
0040104A   cmp        ebp,esp
0040104C   call       _chkesp (004010e0)
```

```
00401051   mov        esp,ebp
00401053   pop        ebp
00401054   ret
...
00424D8C   0018  0000 //数据段的全局变量 gData
```

为了对可执行程序的格式和启动加载过程有一个更全面的了解，这里简单地回顾从C++的源代码到可执行程序文件的转换对应过程，即编译链接过程。

众所周知，C++的源代码包括一系列头文件（.h,.hxx,.hpp）和源文件（.cxx,.cpp），另外编写程序通常需要引用第三方的库。源代码中一个关键概念是“符号”，变量的名字是“符号”，类名/方法名也是“符号”。简单地说，编译链接的目的就是把开发人员可读的“符号”转换为 CPU 可理解的内存地址。

符号的用法有严格的规则，首先，一个符号必须声明后才能使用，这个规则由编译器（compiler）来检查，否则会出现编译错误；其次，在程序的某处使用（引用）一个符号时，程序必须能找到这个符号的实现（对于变量来说，需要找到存放这个变量的内存地址；对于方法来说，需要解析出这个函数的入口地址），这个规则由链接器（linker）来保证；否则会出现链接错误。

第 1 步：预编译展开一些宏。

第 2 步：为每一个.cxx 源文件编译一个目标文件（.obj,.o），显然目标文件中至少包含二进制的代码段和数据段。在 cxx 源文件中有可能会引用在其他 cxx/hxx 中定义的符号，也可能自己定义的一些符号被 cxx 文件引用，这些作用域超过一个 cxx 文件的符号称为“public 符号”（例如非静态函数）。因此每一个目标文件中也包含一个符号表，用于记录自己引用的符号及自己提供的 public 符号。

第 3 步：编译器合成这些目标文件成一个库文件（.lib），同时解析可以找到的符号引用。此时这个库文件包含了二进制的代码段和数据段，同样也会包含一个符号表，因为有一些符号需要引用其他静态/动态链接库的导出符号。

第 4 步：链接器负责把目标的库文件和所有需要引用的静态/动态链接库进行链接，即需要首先把其他静态库合成到可执行文件中。转换相应的符号引用为地址，然后确保所引用的其他动态链接库的符号存在。最终生成可执行文件，可执行文件的符号表只需要记录导入符号表。

在了解可执行程序的内容格式和生成过程后，下面具体讨论应用程序的启动过程。

如果一个应用程序只包含一个单一的可执行文件，并不依赖于任何其他动态链接库，那么应用程序启动时，操作系统首先创建相应的进程并分配私有的进程空间。然后操作系统的加载器负责把可执行文件的数据段和代码段映射到进程的虚拟内存中（注意只是映射，并不直接全部拷贝可执行文件的数据段和代码段到内存，同时预取有限的代码段进入内存），把 CPU 的 IP 指向程序入口点，即可开始执行。在顺序执行或跳转语句执行的过程中，如果发现需要执行的指令或者需要引用的资源还没有在内存中，则发出缺页中断（参见第 4 章中有关内存管理机制的相应描述），操作系统负责把相应的内容读入到内存中。

在现实情况中，应用程序通常会包含多个部分，如一个可执行文件和若干动态链接库，并且这些可执行文件/库文件通常还需要引用到操作系统的动态链接库或者第三方的动态链接库（可以用微软的 dependencywalker 工具查看这些关联信息，图 8-1 所示为一个最简单的 Hello.exe 示例程序所依赖的动态链接库。它依赖于操作系统提供的 Kernel32.dll，而 Kernel32.dll 又进一步需要 Ntdll.dll。图中的 dependencywalker 列出了 Kernel32.dll 输出的函数和 Hello.exe 需要的函数及其他信息）。

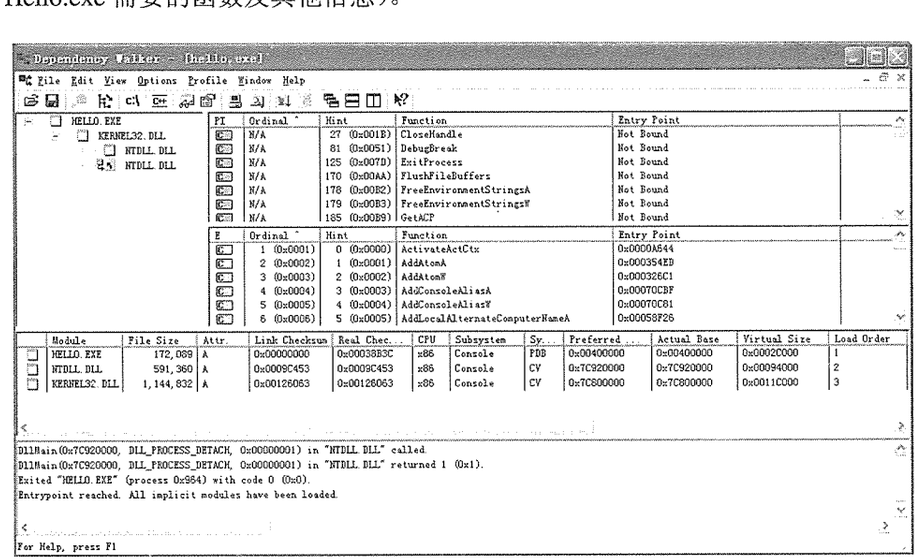


图 8-1　一个最简单的 Hello.exe 示例程序

此时程序启动的过程包含比较复杂的步骤。

第 1 步：操作系统首先创建相应的进程并分配私有的进程空间，然后操作系统的加载器负责把可执行文件的数据段和代码段映射到进程的虚拟内存空间中。

第 2 步：加载器读入可执行程序的导入符号表，根据这些符号表可以查找出该可执行程序所有依赖的动态链接库。

第 3 步：加载器针对该程序依赖的每一个动态链接库调用 LoadLibrary。

（1）查找对应的动态库文件，加载器为该动态链接库确定一个合适的基地址。如果该基地址和动态链接库希望加载的基地址不同，加载器还要为该库做 rebase，然后把整个动态链接库映射到进程的虚拟内存空间中。

（2）加载器读取该动态链接库的导入符号表和导出符号表，比较应用程序要求的导入符号是否匹配该库的导出符号。

（3）针对该库的导入符号表，查找对应依赖的动态链接库，如果有则跳转到第 3 步。

（4）调用该动态链接库的初始化函数。

直到所有应用程序直接/间接依赖的动态链接库处理完毕，全部映射到应用程序进程空间，成为应用程序进程的一部分。

第 4 步：初始化应用程序的全局变量，对于全局对象自动调用构造函数。

第 5 步：进入应用程序入口点函数开始执行。

8.2 Linux 程序启动过程

Linux 操作系统下应用程序的启动过程和 Win32 有类似之处，在 Linux 下应用程序加载和执行的具体步骤如下。

第 1 步：用户在 linux 命令终端中输入运行程序的命令，然后按回车键。

第 2 步：首先接管的是 exec 系统调用，它会为应用程序的运行准备一些环境变量等，并且为运行的命令找到相应的解释器。

第 3 步：通常应用程序的解释器就是 ld（loader/加载器），ld 接管控制权后首先需要读入这个可执行程序的文件的一部分，包括文件头及共享对象（so，对应于 Windows 下的动

态链接库）区等。然后检查这个可执行文件所依赖的共享对象 so（这些信息都在可执行文件中），并且在 LD_LIBRARY_PATH 和系统默认库文件夹的位置查找这些库是否存在。如果不存在，则报告错误并退出执行。

第 4 步：针对每一个依赖的库，ld 需要首先读入这个 so 的一部分文件头和相关信息。然后递归查找该共享对象所依赖的其他共享对象，直到最底层。在这个过程中，ld 会在内部维护一个数据结构用来记录所有这些共享对象之间的相互依赖关系。最终，ld 会确认所有的该可执行程序直接或间接依赖的 so 都存在。

第 5 步：ld 会把所有依赖的 so 映射到该程序的进程空间的虚拟内存中（只是映射，并不是把全部 so 文件的内容读入内存）。显然，每一个共享对象在该进程的虚拟内存空间中占据不同的连续区域。它们的“基地址”各不相同，从而其内部的一些用绝对地址表示的符号需要做出相应的修改。这个过程称为“relocation 过程”。

第 6 步：初始化应用程序的全局变量，对于全局对象自动调用构造函数。

第 7 步：进入 main 函数开始执行。

8.3 影响程序启动性能的因素

程序启动性能定义为程序启动所需要的时间，从用户的角度看，即指从用户启动该程序到用户可以使用该程序的这一段时间。程序启动性能是应用程序易用性的一个重要的指标，是用户对应用程序的第一印象，直接决定用户对应用程序的评价。

在工程实践中，程序启动性能有两个指标需要分别对待，即冷启动性能和热启动性能。冷启动性能指在操作系统重新启动后，应用程序第 1 次启动所需要的时间；热启动性能则指应用程序第 2 次及其以后启动所需要的时间。之所以要区分这两个指标，是因为操作系统的缓存机制在发挥作用。程序第 1 次启动完毕并且退出运行后，操作系统仍然会在内存的硬盘缓存和系统缓存中暂时保留可执行程序、相应的动态链接库，以及程序用到的其他配置文件/资源文件的内容。从而在应用程序的第 2 次启动时，可以减少大量的 IO 操作，导致热启动性能要好于冷启动性能。

程序启动的时间无非包括两个部分，即 IO 操作消耗的时间和程序代码运行占用的 CPU 时间。

从工程实践中的应用程序来看，启动过程中通常不会执行太多的代码，也不会消耗太多的 CPU 时间。换句话说，程序的冷启动性能大部分取决于 IO 操作消耗的时间。相应地，程序的热启动性能则大部分取决于 CPU 时间。因为程序在第 2 次启动时，大部分需要的文件已经被操作系统缓存，IO 操作就占少数了。一个典型示例如图 8-2 所示。

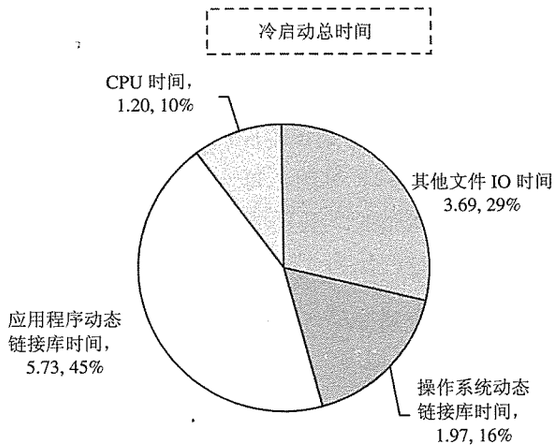


图 8-2　一个冷启动的典型示例

该程序的冷启动时间约为 12 秒，其中有 90%的时间都消耗在 IO 操作上（包括程序自身动态链接库占 45%，程序需要的操作系统的动态链接库 16%，以及启动过程中访问其他文件的 IO 占 29%）；该程序的热启动时间约为 1.5 秒，其中大部分（1.2 秒）为 CPU 时间。

本节讨论影响 IO 和 CPU 的时间，进而影响到应用程序的启动性能的有关因素。

8.3.1　源代码因素

源代码因素主要指必须修改源代码才可以解决或者缓解启动性能问题。

1．全局变量初始化

程序的全局变量在 main 函数执行前就要被自动初始化，初始化过程需要调用该对象的构造函数。如果程序中的全局变量或者这些全局变量的构造函数执行的操作过多，则会占

用较多的 CPU 时间，进而影响到程序的启动性能。判断的方法主要是在 main 函数之前和之后设置时间点，如果确认 main 函数之前的启动时间在总的启动时间中占据了较多部分，则应该是程序启动时执行了过多的全局变量初始化操作。

2．代码自身消耗 CPU

代码自身消耗 CPU 的情况比较容易理解，低效的启动代码直接导致差的启动性能。不过一般情况下，程序启动时不大可能执行过多的纯计算工作，所以代码自身消耗 CPU 过多导致程序启动缓慢的情况并不多见。如果通过其他工具分析为启动性能中的 IO 问题，则可以利用 quantify 等工具定位消耗 CPU 的热点代码，加以优化。

3．无用代码导致可执行程序尺寸膨胀

大型的应用程序，尤其是经过多年开发进化的应用程序经常会有很多遗留下来的无用代码，这些代码编译后必然使得可执行程序或动态库的尺寸膨胀，因此会增加加载应用程序的时间。

8.3.2　动态链接库因素

动态链接库的因素在程序启动性能分析中占据举足轻重的地位，实践表明，大部分程序的启动性能问题和动态链接库有关系。而应用程序对动态链接库的依赖关系体现为两种，即隐式依赖和显式调用。

（1）隐式依赖：应用程序不通过 LoadLibrary 和 GetProcAddress 而直接使用该动态链接库的函数，依赖关系可以通过可执行程序的导入符号表得到，属于静态的依赖关系。也可以通过 dependencywalker 等工具直接看到。

（2）显式调用：应用程序并不静态依赖该动态链接库，而是在需要时显式调用 LoadLibrary 等函数加载动态库，可以通过 dependencywalker 等工具运行应用程序时动态观察到。

具体来说有以下这些情况会影响应用程序的启动性能。

1．动态链接库数量及大小

操作系统每次加载一个动态链接库需要有一些固定的时间消耗，例如，打开动态链接

库文件、读入该动态链接库的导入符号表和导出符号表等。从而加载多个小的动态链接库的时间之和要远大于单独加载一个大的动态链接库的时间。

同时，在动态链接库个数相等的情况下，如果动态链接库比较大，则意味着有可能导致较多的 IO。这是因为操作系统的 IO 最小是以页（4 KB 字节）为单位的，如下所示两个不同的动态库 A 和 B 都包含函数 func1 和 func2：

```
A.dll : page1(func1,func2);page2(…)
B.dll: page1(func1,func8);page2(func2,func3),page3(…)
```

但是由于 B 还有一些无用的代码导致编译的动态库比较大，从而使得 func1 和 func2 跨了两个内存页；而 A 代码尺寸紧凑，func1 和 func2 只跨了 1 个内存页。显然在应用程序需要执行 func1 和 func2 时，A 库只需要 1 次缺页中断引发的 IO 操作；B 库则需要两次。

总而言之，在考虑应用程序启动性能时，不能一概而论地说应用程序需要的动态链接库个数多了就一定要合并，或者个数少就一定要分割。例如，一个程序启动时只用了一个动态链接库，但是这个库文件很大。这时就要考虑把大的动态链接库分割成两个或多个较小的库，使得启动时无用的代码分离到其他库中，在启动以后需要时再加载。又如应用程序启动时需要加载数十个不同的动态链接库，而且这些动态库的代码都是启动必需的，不可精简。这种情况就需要把这些多而小的动态链接库尽量地合并成一个或数个大的库，减小操作系统加载库的工作量。所以正确的做法应该是要考虑到应用程序启动过程中涉及的动态链接库的个数和大小，个数越少越好，尺寸越小越好。

2. 启动时加载的操作系统的动态链接库

应用程序或多或少都需要用到操作系统的动态链接库，虽然操作系统会优化一些常用的动态库（例如在系统启动时预先加载等），但是在应用程序启动过程中过多地使用操作系统的动态库同样会带来性能问题，这一类问题可以通过文件 IO 监测工具（例如 Filemon）来发现。

3. 调用代码时引发缺页导致 IO

调用代码时引发的缺页应该是最经常遇到的启动性能问题，尤其是大型的应用程序（这类程序安装后的可执行文件/库文件可达数百兆字节）更是如此。

默认情况下，编译器首先把源代码编译为目标文件，链接器再把多个目标文件链接成

一个可执行程序或者库文件。经过这个过程后，动态链接库文件中各个函数的物理排列顺序就是开发人员在源代码中编写的函数的顺序。但是在应用程序启动时，CPU 执行调用这些函数的顺序则完全取决于程序自身的逻辑调用关系，与源代码中的函数顺序几乎没有关系。而操作系统调用应用程序和动态链接库中的函数又是按需读入内存的，如果启动过程中需要的函数十分分散地存在于动态链接库中，则会引起很多不必要的 IO 操作，导致较差的启动性能。

例如，在源文件中，这些函数以如下顺序排列：

```
xxx.cpp : func1, func2, func3, func4, func5, func6, func7, func8.
```

那么在编译后的动态链接库文件中，它们会以同样的顺序排列。假设每 4 个函数的大小可以占用一个内存页，那么这个动态链接库的内容如下所示：

```
xxx.dll : page1 ( func1, func2, func3, func4) , page2 ( func5, func6, func7, func8).
```

假设应用程序启动时，只有 func1 和 func7 被调用。而操作系统加载动态链接库的方式是以内存页为单位的，从而操作系统会首先读入 page1（包含 func1），然后读入 page2（包含 func7），共需要两次缺页中断和 8 KB 的 IO 操作。

但是如果已知程序启动时函数调用顺序为 func1->func7，从而主动地重新排列动态链接库中的函数顺序，如下所示：

```
xxx.dll : page1( func1, func7, func2, func3), page2( func4, func5, func6, func8).
```

在这种情况下，操作系统就只需要读入 page1（同时包含 func1 和 func7）。这样就节省了一次缺页中断和 4 KB 的 IO，大大提高了程序的启动性能。

下面以一个具体的程序实例来说明动态链接库代码自身缺页中断引发的 IO 对程序启动性能的影响。

以一个名为“EdrLib.dll”的动态链接库用来模拟实际的动态库的内容，这个动态库只输出了一个函数，即“Dummy()”，这个函数也只调用了一个函数“Dummy1499()”。类似地，Dummy1499()除了做一些运算外也只调用了一个函数“Dummy1498()”。依此类推，直到调用 “Dummy0()”为止。EdrLib.dll 由这 1501 个函数组成，编译后的大小为 9.76 MB，平均每个函数占 6.5 KB。如下所示：

```
void Dummy1499()
{
```

```
        int a = 10, b = 20, c = 30, d = 40, e = 50, f = 60;
        a = 2 * (b << 2) + 5 * c % 3 - 3 * d / 2 + 7 * e % 2 + 4 * (f >> 3) + a *
(5 << 2);
        b = 5 * a % 3 + 3 * c % 5 - 7 * d / 4 - 2 * e % 2 + 6 * f % 2 + b * (4 >>
2);
        c = 10 * (b << 1) - 3 * a % 4 + 7 * d % 3 + 6 * (e << 4) - 7 * (f << 2) +
c * 7 % 2;
        d = a * 100 % 3 - b * 4 % 2 + 3 * c % 2 + 5 * (e << 3) + 9 * f / 4 + d * 11
/ 3;
        e = a * 7 / 3 - b * 4 % 2 + c * (3 << 2) - 5 * d / 3 + 11 * f % 7 + e * 13 %
3;
        f = 83 * (a >> 2) + 7 * b / 3 + 5 * c % 2 - 7 * d % 3 + 99 * e % 8 + f * 16
/ 4;
        ......
        Dummy1498();
    }
```

实验程序 EdrTest.exe 直接调用 EdrLib.dll 的惟一导出函数 Dummy()。由于 Dummy()相继调用了其他 1 500 个 Dummy*函数，从而在该程序被第 1 次执行时，由于操作系统需要依次把 Dummy*的函数体读入内存，因此引发一系列的缺页中断。实验程序的内容如下：

```
    int WINAPI WinMain (HINSTANCE hInstance, HINSTANCE hPrevInstance,
                        PSTR szCmdLine, int iCmdShow)
    {
         LARGE_INTEGER  fq, c1, c2;
         DWORD dwStart, s1;
         LONG  s2;
         wchar_t str[100];
         //初始化性能计数器
         if(QueryPerformanceFrequency(&fq))
         {
            MessageBox (NULL, TEXT ("Current system supports high-resulotion
timing…"), TEXT ("Continue…"), MB_OK) ;

            MessageBox (NULL, TEXT ("Click to begin executing Dummy() for the first
time…"), TEXT ("Dummy"), MB_OK) ;
            //获得开始时间
            QueryPerformanceCounter(&c1);
            Dummy();
            //获得结束时间
```



```
            QueryPerformanceCounter(&c2);
            s1 = c2.u.LowPart - c1.u.LowPart;
            s2 = c2.u.HighPart - c1.u.HighPart;
            //计算 Dummy 的运行时间，精确到毫秒
            dwStart = ((s2 << 32) + s1) * 1000.0 / ((fq.u.HighPart << 32) +
fq.u.LowPart);
            swprintf(str, L"Time used to execute Dummy() for the first time: %d ms",
dwStart);
            MessageBox (NULL, str, TEXT ("Dummy"), MB_OK) ;

            MessageBox (NULL, TEXT ("Click to begin executing Dummy() for the second
time…"), TEXT ("Dummy"), MB_OK) ;
            //第二次运行
            QueryPerformanceCounter(&c1);
            Dummy();
            QueryPerformanceCounter(&c2);
            s1 = c2.u.LowPart - c1.u.LowPart;
            s2 = c2.u.HighPart - c1.u.HighPart;
            dwStart = ((s2 << 32) + s1) * 1000.0 / ((fq.u.HighPart << 32) +
fq.u.LowPart);
            swprintf(str, L"Time used to execute Dummy() for the second time: %d ms",
dwStart);
            MessageBox (NULL, str, TEXT ("Dummy"), MB_OK) ;
         }
         else
         {
            // doesn't support high-resulotion timing, hence use GetTickCount to
profile… omit for space
         }
        return 0 ;
    }
```

实验的过程为重新启动操作系统后，先后两次执行 EdrTest.exe。并且用一个监视缺页中断的工具 pfmon 来记录两次执行引起的缺页中断，结果如下：

（1）第 1 次执行 EdrTest.exe。

- Dummy()第 1 次：用时 919 ms，因为调用 Dummy 系列函数引起的缺页 1567 次。其中硬缺页 223 次，软缺页 1 344 次。

- Dummy()第 2 次：用时 6 ms，因为调用 Dummy 系列函数引起的缺页 84 次。其中硬缺页 3 次，软缺页 81 次。

（2）第 2 次执行 EdrTest.exe。

- Dummy()第 1 次：用时 12 ms，因为调用 Dummy 系列函数引起的缺页 918 次。其中硬缺页 20 次，软缺页 898 次。
- Dummy()第 2 次：用时 6 ms，因为调用 Dummy 系列函数引起的缺页 99 次。其中硬缺页 2 次，软缺页 97 次。

结果中所谓的“硬缺页”指需要从硬盘上读取数据的缺页中断，即引发实实在在的 IO。从结果中可以清楚地看到冷启动和热启动时程序启动性能的巨大差别，这主要是由与程序代码自身需要调入内存引起的缺页中断导致的。

接下来可以把程序稍做调整，以验证动态链接库中的函数顺序对程序启动性能的影响。

这一次 dll 内部的函数依然包含 1 500 个函数，但是需要全部导出，函数之间不必再有调用关系。例如：

```
void Dummy1499()
{
    int a = 10, b = 20, c = 30, d = 40, e = 50, f = 60;
    a = 2 * (b << 2) + 5 * c % 3 - 3 * d / 2 + 7 * e % 2 + 4 * (f >> 3) + a *
(5 << 2);
    b = 5 * a % 3 + 3 * c % 5 - 7 * d / 4 - 2 * e % 2 + 6 * f % 2 + b * (4 >>
2);
    c = 10 * (b << 1) - 3 * a % 4 + 7 * d % 3 + 6 * (e << 4) - 7 * (f << 2) +
c * 7 % 2;
    d = a * 100 % 3 - b * 4 % 2 + 3 * c % 2 + 5 * (e << 3) + 9 * f / 4 + d * 11
/ 3;
    e = a * 7 / 3 - b * 4 % 2 + c * (3 << 2) - 5 * d / 3 + 11 * f % 7 + e * 13 %
3;
    f = 83 * (a >> 2) + 7 * b / 3 + 5 * c % 2 - 7 * d % 3 + 99 * e % 8 + f * 16
/ 4;
    ......
}
void Dummy1498()
{
    int a = 10, b = 20, c = 30, d = 40, e = 50, f = 60;
```



```
    a = 2 * (b << 2) + 5 * c % 3 - 3 * d / 2 + 7 * e % 2 + 4 * (f >> 3) + a *
(5 << 2);
    b = 5 * a % 3 + 3 * c % 5 - 7 * d / 4 - 2 * e % 2 + 6 * f % 2 + b * (4 >>
2);
    c = 10 * (b << 1) - 3 * a % 4 + 7 * d % 3 + 6 * (e << 4) - 7 * (f << 2) +
c * 7 % 2;
    d = a * 100 % 3 - b * 4 % 2 + 3 * c % 2 + 5 * (e << 3) + 9 * f / 4 + d * 11
/ 3;
    e = a * 7 / 3 - b * 4 % 2 + c * (3 << 2) - 5 * d / 3 + 11 * f % 7 + e * 13 %
3;
    f = 83 * (a >> 2) + 7 * b / 3 + 5 * c % 2 - 7 * d % 3 + 99 * e % 8 + f * 16
/ 4;
    ......
}
... ...
void Dummy0()
{
    int a = 10, b = 20, c = 30, d = 40, e = 50, f = 60;
    a = 2 * (b << 2) + 5 * c % 3 - 3 * d / 2 + 7 * e % 2 + 4 * (f >> 3) + a *
(5 << 2);
    b = 5 * a % 3 + 3 * c % 5 - 7 * d / 4 - 2 * e % 2 + 6 * f % 2 + b * (4 >>
2);
    c = 10 * (b << 1) - 3 * a % 4 + 7 * d % 3 + 6 * (e << 4) - 7 * (f << 2) +
c * 7 % 2;
    d = a * 100 % 3 - b * 4 % 2 + 3 * c % 2 + 5 * (e << 3) + 9 * f / 4 + d * 11
/ 3;
    e = a * 7 / 3 - b * 4 % 2 + c * (3 << 2) - 5 * d / 3 + 11 * f % 7 + e * 13 %
3;
    f = 83 * (a >> 2) + 7 * b / 3 + 5 * c % 2 - 7 * d % 3 + 99 * e % 8 + f * 16
/ 4;
    ......
}
```

按照此种方法编写的 dll 编译后内部函数的顺序依然和源代码中的函数顺序保持一致，应该是从 Dummy1499()到 Dummy0()。此时由 Edrtest 程序按照不同的顺序调用 1 500 函数中的 500 个函数，例如第 1 次调用从 Dummy1499()到 Dummy1000()模拟函数排列紧密时的情况。第 2 次调用只选择序号为 3 的倍数，即 Dummy1497()，Dummy1494()，……，Dummy3()，Dummy0()模拟函数排列混乱的情形。比较这两种不同情形的冷启动性能，具体的结果请读者自行验证。

4．动态链接库初始化工作

操作系统的加载器加载动态链接库之后，都会自动调用其初始化函数。如果这个初始化函数需要做的工作过多，占用过多 CPU 事件或者引发大量 IO，则同样会导致应用程序的启动性能下降。

5．动态链接库符号可见性

动态链接库的符号，例如函数名，如果需要被外部（其他库文件）可见/可访问，则需要放入导出符号表中，而这个导出符号表在每次动态库加载时都要被加载器读入内存并且扫描匹配。如果一个动态链接库导出过多的符号，或者导出符号的名字比较复杂，则也会引起加载器消耗较多的 CPU 事件和 IO 以执行字符串操作来处理这个导出符号表，从而影响到程序启动的性能。

6．动态链接库 relocation 问题

动态链接库中的主要内容是 CPU 可执行的代码，包括函数和全局变量等。在动态链接库内部函数和全局变量的表示有两种形式，即符号表示和相对地址表示。符号表示即为函数或者全局变量取一个名字供其他动态库引用，符号表示的符号需要集中放置在导出符号表中；相对地址表示则是该函数的入口地址相对于动态链接库的基地址的偏移量。真正执行时，显然 CPU 需要在虚拟内存中的绝对地址。对于符号表示来说，引用者通过查找被引用者的导出符号表，获得这个符号表示的函数的地址；对于相对地址表示的函数，则在加载器确定该动态链接库的基地址所在的绝对地址后，直接计算得到函数在虚拟内存空间中的绝对地址。这些查表或者计算都需要消耗时间，如果数量过多，也会影响到应用程序的启动性能。

8.3.3 配置文件/资源文件因素

另一种影响启动性能的可能因素是启动部分的代码频繁访问配置文件/资源文件而导致 IO 过多，这种情况通常发生在应用程序有大量小配置文件/资源文件。同时在启动过程中需要在不同的时间多次访问这些小文件，从而造成 IO 次数过多和 IO 操作过于分散（不能有效利用硬盘的连续访问来提高性能），导致启动性能下降。

例如，4 次 IO 操作分别按 1、2、3、4 的顺序访问数据文件，则硬盘的磁头运动轨迹如图 8-3 所示，需要来回地寻道移动。对于 5 400 转的磁盘来说，平均每次寻道需要 12 ms 以上的时间，比顺序访问的 IO 来说性能降低了很多。

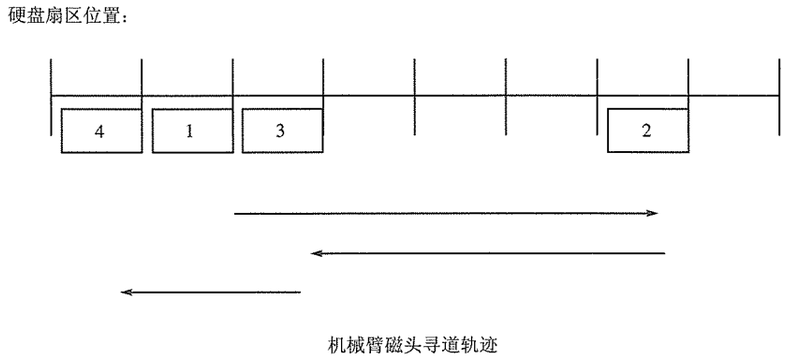


图 8-3　硬盘的磁头运动轨迹

8.3.4 其他因素

1．硬盘文件碎片问题

在测试应用程序启动性能时，会注意到一个显著的现象。如果应用程序安装在空的“干净”的分区上，并且在安装之后执行磁盘碎片整理，则应用程序的启动性能有可能会大幅提升。

在理想状态下，顺序读取一个文件时，应该不需要磁头的寻道操作。但是应用程序看到的“顺序”文件只是一个逻辑上的概念，操作系统的文件系统模块以块为单位管理物理磁盘空间。当文件系统经过长期添加、修改或删除文件之后，有可能不再有足够大的连续块，从而一个逻辑上连续的文件/文件目录在物理上实际由一系列的“碎片”组成。这种情况下应用程序的文件（包括动态链接库文件和其他文件）如果没有执行碎片整理，则有可能因为碎片太多（文件的内容没有连续存放在磁盘的相邻扇区中，在物理位置上相互远离），从而导致执行 IO 时过多的寻道时间，如图 8-4 所示。

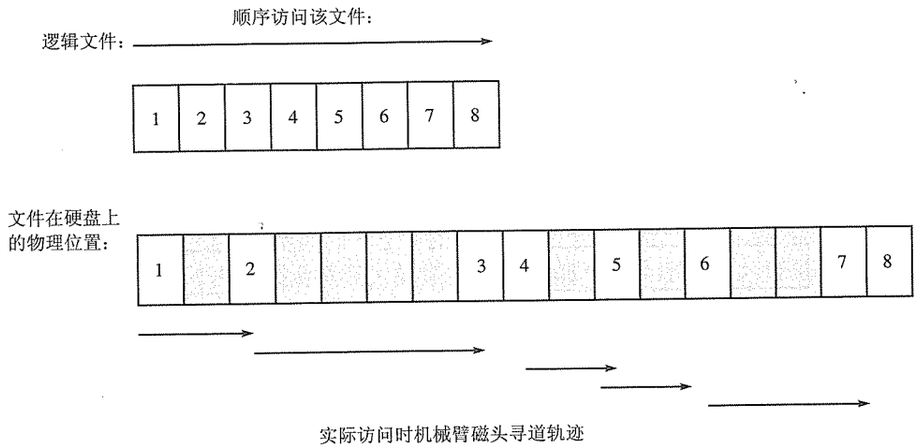


图 8-4　硬盘文件碎片

图中的文件在硬盘上的物理存放位置被分为 6 块碎片（1、2、3、4、5、6、7 和 8），如果没有碎片，顺序访问该文件只需要一次寻道操作。而有了 6 块碎片则至少需要 6 次寻道操作，寻道时间增加了 5 倍。

2．操作系统的预读（Prefetch）功能

WindowsXP 系统为了提高应用程序的启动性能，实现了独特的“预读”功能。应用程序运行所需要的代码和数据被操作系统以“页”为单位（在 x86 体系中一页的大小为　4 KB）从硬盘上调入内存中，从而大部分启动时间会被缺页中断引起的 IO 所占用。而预读功能就是操作系统在应用程序启动之前，预先把该应用程序启动时所需要的代码页和数据页调入内存。从而在程序真正启动时，避免了硬缺页中断引发的 IO 操作，大大提高了程序的启动性能。

为了实现预读的功能，每当应用程序启动发生硬缺页中断时，操作系统会记录该应用程序的 IO 访问的文件系统的文件及其具体的起止位置，该应用程序的所有这些信息会保存在预读区域指示文件中。在\\WINDOWS\Prefetch 中，针对每一个常用的应用程序，都保存了对应的预读区域指示文件。例如 ACRORD32.EXE-13285B88.pf 是 Acrobat reader 的预读指示文件，其中的“13285B88”是应用程序安装路径的一个哈希值。当应用程序启动时，操作系统有机会首先访问对应的预读区域指示文件。操作系统的文件系统模块会为该应用

程序打开所有启动时需要的文件，操作系统的内存管理模块会对照预读区域指示文件指定的起止位置把应用程序需要的数据页和代码页调入内存。一切准备就绪后，应用程序才开始启动。

简单地说，预读功能优化启动性能的原理是对顺序读写硬盘的速度要大于随机读写速度，节约寻道时间。

假设一个应用程序 app.exe 启动时需要 a.dll 和 b.dll 中的代码和数据，这 3 个文件每个在硬盘上占用了 6 个内存页，如下所示：

App.exe(app_page1, app_page2, app_page3, app_page4, app_page5, app_page6)

a.dll(a_page1, a_page2, a_page3, a_page4, a_page5, a_page6)

b.dll(b_page1, b_page2, b_page3, b_page4, b_page5, b_page6)

假设启动时引发的缺页中断顺序为 app_page1,a_page5,b_page4,app_page2,b_page1,a_page2,b_page2,a_page3,a_page4,b_page3。在这种模式下，硬盘的磁头需要不断地从文件 app.exe 跳到 a.dll，再跳到 b.dll。如此往复，至少需要 10 次寻道时间。

而通过操作系统的预读功能，记录了 app.exe-12223333.pf。其中指示启动 app.exe 时，需要打开文件 app.exe，读取该文件的 app_page1,app_page2,app_page3。然后打开文件 a.dll，读取该文件的 a_page2,a_page3,a_page4,a_page5。再打开文件 b.dll，读取该文件的 b_page1,b_page2,b_page3,b_page4。以这种 IO 模式，则硬盘的磁头只需要 3 次寻道时间，其他大部分时间是在顺序读取文件。这样大大减少了 IO 时间，提高了性能。要说明的是，这个例子是理想状态下的情形，在实际中即使顺序读取一个文件，也有可能因为文件的碎片等因素而导致多次的寻道操作。

为了增强操作系统依据预读指示文件预读时带来的好处，每 3 天左右的空闲时间，WindowsXP 会针对预读指示文件所指定的那些应用程序文件执行一个局部的碎片整理。需要做整理的这些文件路径会保存在\Windows\Prefech\Layout.ini 中。

8.4　本章小结

应用程序启动过程同时涉及操作系统的动作和应用程序自身的启动动作，操作系统需

要为应用程序的启动创建环境、加载可执行文件和动态链接库，并且初始化应用程序的部分数据；应用程序则需要执行自身启动时的程序逻辑，其中包括初始化数据，并访问配置文件及资源文件等。了解了这一点，应用程序开发人员在考虑优化程序启动过程时要开阔视野，不仅局限在应用程序自身的启动过程，而是应该在操作系统的层次来考虑问题。

从另一个角度来说，应用程序启动消耗的时间无非包括 CPU 运算占用的时间和 IO 操作占用的时间，而一次 IO 操作消耗的时间相对于 CPU 运算来说是非常可观的。从这一点出发，应用程序在启动时要尽量减少和避免不必要的 IO 操作。

第 9 章会通过实例来详细介绍一些应用程序启动过程的优化方法。

第 9 章　程序启动性能优化

应用程序的启动速度是应用程序可用性的重要指标，同时也是应用程序展现给最终用户的第一印象，用户都希望得到闪电般快速的程序启动体验。尤其对于需要经常反复启动的应用程序（例如浏览器和字处理器等）来说，启动性能更是至关重要，因而每一个软件产品都应该设法使自己的启动性能得到最大的优化。

因为程序启动性能的重要性，现代的操作系统已经为优化应用程序的启动做了一些工作。比如在程序第 1 次启动后的一段时间内，操作系统会在内存中缓存该程序启动时需要的文件部分，从而在下一次启动该程序时减少 IO，提升启动性能。

本章首先介绍程序启动性能优化的一般步骤，然后总结获得精确稳定的程序启动性能的测试方法，最后以主要篇幅列举了优化程序启动性能的方法。

优化应用程序的启动性能的方法可以分为外围针对可执行文件的直接修改和内部针对源代码的优化两大类别。

9.1 优化程序启动性能的步骤

和通常的性能优化方法论类似，优化应用程序启动性能通常遵循如下步骤。

（1）定义启动性能问题，包括定义启动阶段的范围和设定启动性能的可行目标。

（2）在定义启动性能的问题之后，需要通过测试（自动或者人工方式）来获得具体的启动性能数据。

（3）测试得到的稳定精确的数据是一切优化工作的基础，在此基础之上，利用性能分析工具来确定应用程序启动性能的瓶颈或者影响启动性能的因素。

（4）针对特定的性能瓶颈或者影响因素设计具体的优化方案，并且实施优化或者实验优化方案。

（5）针对优化后的应用程序重新回到第 2 步来确定优化后的测试结果，并且和预期目标比较。如果达成目标，则优化完成；否则需要重复（2）~（5）的过程，反复优化。

整体流程如图 9-1 所示。

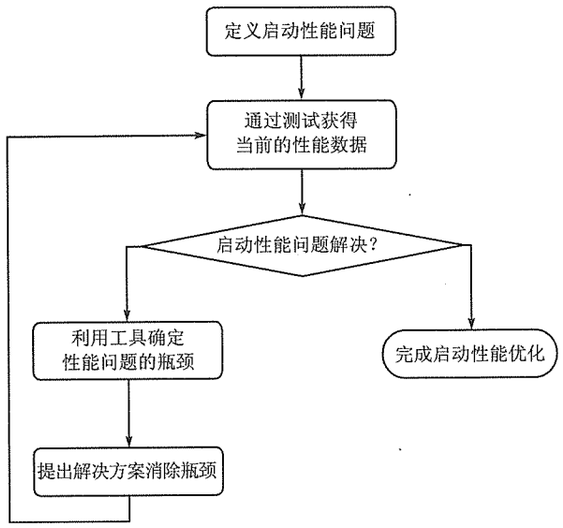


图 9-1 优化程序启动性能的步骤

9.2 测试程序启动性能的方法

在性能优化的工作中，性能测试处于十分重要的地位。一方面，性能问题的存在需要性能测试的结果来证明；另一方面，某一种优化方法的效果也需要性能测试的数据来体现。

具体到程序启动性能测试，因为影响程序启动性能的因素很多，所以一旦测试的数据不够精确和恒定，就会出现很多难以解释的问题，也有可能根据错误的测试结果而丢掉正确的优化方法。

对于同一个应用程序，热启动的时候由于操作系统已经缓存程序启动所需要的大量文件，所以热启动性能一定要好于冷启动性能。同样地，由于热启动有操作系统的参与，从而使得热启动性能有一定的不确定性。即无法确定操作系统为应用程序缓存多少文件，或者何时又从缓存中置换出多少文件。因而本节讨论程序启动性能的测试方法主要以冷启动为主，以获得一个比较精确而相对稳定的测试结果，热启动性能也可以用类似方法获得。

为了获得精确可用的启动性能测试数据，需要遵循如下测试原则。

1. 保持相同的测试环境

这一条原则是性能测试的基本，但是在程序启动性能测试时往往容易忽略一些细节而导致较大的偏差。

根据工程实践经验，在程序启动性能测试中必须注意以下要点。

（1）应用程序必须单独安装在一个空的分区中（如果再次安装测试该程序的不同版本，则需要首先格式化该分区），每次测试安装的分区必须保持不变，包括分区容量、文件系统和分配单元大小等。

这是为了保证磁盘 IO 的影响保持恒定。对于 IO 敏感的应用程序启动过程来说，如果随意安装在不同的分区，或者和其他应用程序混装在一个分区中，导致的启动性能误差可达 20%左右。

如果确实不能满足这一条原则，至少应该在每次启动性能测试之前，对安装测试程序的分区执行碎片整理。

（2）测试电脑的操作系统要尽量地保持“清洁”，不要安装过多的其他程序，尤其是会定期活动或者引起网络连接的程序（例如防病毒软件和防火墙程序等）。

这是为了排除其他程序对测试目标程序的干扰。

（3）需要每次重新启动操作系统，在操作系统启动之后必须等待一定的时间。直到操作系统的 CPU 占用、网络占用和 IO 活动都为 0，内存占用量恒定，然后开始执行测试过程。

（4）测试所用的不同版本的应用程序安装包应该有相同的来源，例如同一个产品的发行版本安装包可能和开发小组使用的开发版本情况不同（前者可能包含一些 help 文件和多语言资源文件等）。

（5）测试电脑的电源管理功能，例如有的 CPU 有自动调整运行频率的功能。为此需要在测试时强制设置为固定频率运行。显卡也有类似需要注意的问题。

（6）在 WindowsXP 中测试时，要考虑操作系统的预读功能对于应用程序启动性能的影响，可以考虑删除\\Windows\Prefetch\下对应的预读指示文件或者暂时禁止操作系统的预读功能。

如果每次冷启动性能的测试都能遵循上述原则，那么得到的性能数据就是稳定可靠的。

2. 尽量利用自动测试

自动测试的优点一是节省了人力，更重要的是能保证测试结果的精确性和稳定性。

下面是一个自动测试应用程序启动性能的实例，它利用一个 java 程序启动目标应用程序并记录第 1 个开始时间点，同时注册一个 listener 到目标应用程序中。当目标应用程序启动完成之后，会引发一个“StartupFinish”的事件。java 程序监听到这个事件后记录第 2 个时间点，这两个时间点之差就是目标应用程序的启动时间。代码如下：

```
class MyEventListener implements XEventListener{
    private long nStartupTime = 0;

    MyEventListener(long nStartup, long nTimes){
       nStartupTime = nStartup;
    }

    //监听到 StartupFinish 事件以后的处理操作
    public void notifyEvent(EventObject evt){
```



```
        if(evt.EventName.equals("StartupFinished"))   {
         System.out.println("load time: \t" + (System.currentTimeMillis() -
nStartupTime));
         System.exit(0);
        }
    }
}

public static void main(String args[]) {
    //记录目标应用程序启动前的时间点
    long nStartupTime = System.currentTimeMillis();
    MyEventListener myListener = new MyEventListener(nStartupTime);
    //启动目标应用程序，并且传入 listener
    startupApplication( myListener );
}
```

在 Windows 平台上，把这样一个 Start.bat 文件放在启动文件夹中即可开始自动冷启动性能测试：

```
cd "C:\Documents and Settings\tester"
sleeptool.exe 200//在 Windows 上没有直接的 sleep 命令，一个简单的 sleep 工具
call javalauncher.bat//调用上文的 java 小程序记录精确的启动时间。
sleeptool.exe 30
shutdown -r -t 10
```

sleeptool 的具体实现如下：

```
#include "stdafx.h"
#include "Windows.h"
#include <stdlib.h>

int main(int argc, char* argv[])
{
    if ( argc != 2 )
    {
        printf( " Usage : sleeptool time ( in seconds, range is <0-3600> ) \n
");
        return 0;
    }

    char *stopstring;
    unsigned long seconds = strtoul(argv[1],&stopstring,10);
```

```
        if (seconds > 3600)
        {
            printf( " Usage : sleeptool time ( in seconds, range is <0-3600> ) \n
");
            return 0;
        }

        Sleep((DWORD)seconds*1000);
        printf("Sleep %u seconds \n",seconds );

        return 0;
    }
```

总之，精确测试程序启动性能是程序启动性能优化的前提，也是关键一步，必须认真对待。

9.3 优化可执行文件和库文件

发现程序的启动性能问题后，就要着手进行优化。本节说明从外围优化的方法（尽量少的修改源代码）。

9.3.1 减少动态链接库的数量

减少动态链接库的数量是提高程序启动性能的一个重要原则，是构造大型应用程序时需要考虑的首要问题。例如微软的 Office 套件程序启动时只需要两个动态链接库。根据实践经验，如果一个应用程序启动时需要加载的动态链接库达到数十个，则把这些动态库的数量减少到 10 个以内，至少可以提高启动性能达 15%。

在性能优化工程实践中，减少应用程序启动时需要加载的动态链接库数量可以通过如下两种方法来实现。

（1）强制性的修改代码，使得不必要的动态链接库依赖尽量减少。即针对一些虽然加载，但是代码量很少的动态链接库，把启动时需要的函数代码分离出来，加入在其他动态

链接库中。

例如 A.exe 启动时需要 B.dll 和 C.dll 两个动态链接库。B.dll 包含 100 个函数，其中 99 个函数都需要在启动时用到；而 C.dll 也包含 100 个函数，其中只有 1 个函数 funcD()需要在启动时用到。为了减少动态链接库的数量，则只需把 C.dll 中的那个函数 funcD()移入 B.dll 即可。

这种方法的缺点是需要修改大量的代码并且适用的范围比较窄。

第 2 种方法是合并动态链接库，其思路就是把启动时需要加载的多个较小的动态链接库合并成一个大的动态库。这种方法操作比较简便，几乎不需要修改代码，适用范围也比较广泛。具体操作步骤如下（以把一个 DLL1 合并到另一个 DLL2 中为例）。

（1）修改生成该 DLL1 的 makefile，使该 DLL1 不被生成。同时需要把原来构成该 DLL1 的那些 obj 文件合并生成一个静态 lib。

如注释掉 makefile 中的 DLLTARGET1，以及相关宏的定义。

```
# DLLTARGET1=$(CM _TARGET)
```

（2）把 DLL1 相关的静态 lib 复制到适当的位置，使得 DLL2 所在模块可以存取到 DLL1.lib。

（3）修改 DLL2 模块的 makefile，使得合并后的 DLL2 会包含 DLL1 中的静态 lib。

（4）把 DLL1 的 export 函数添加到 DLL2 的 export 函数表中。

（5）修改那些原来依赖 DLL1 模块的 makefile。如果在 makefile 中直接指定 import lib 的名字，则用 DLL2 的 import lib 替换。

（6）修改编译脚本，使得 DLL1 先于 DLL2 编译。如果它们没有编译上的依赖，则增加该依赖关系。

（7）如果 DLL1 和 DLL2 中有相同名字的全局函数/全局变量，需要进行相应的修改，可能会涉及较多的代码修改。

（8）如果 DLL1 中的资源与 DLL2 中的资源的 ID 冲突，需要进行相应的修改。

其他的一些特殊情况需要注意，例如如果 DLL1 和 DLL2 都导出了一些类似 factory 的方法，则需要把合并后的 DLL2 的 factory 方法中加入 DLL1 所支持的对象 facotry 的创建方法。

（10）还可能需要修改应用程序的安装程序，确保不会安装合并掉的 DLL1。

总之，减少动态链接库的数量对于提高应用程序的启动性能是一个行之有效的方法，需要优先考虑。

当然如果不仅仅考虑程序启动性能，动态链接库的数量也并非是越少越好。例如适当地划分动态连接库有利于程序针对不同动态链接库模块化的升级。

9.3.2 减小动态链接库尺寸

减少动态链接库的尺寸同样是从减少 IO 的角度考虑的，具体的实现方法有如下两种。

（1）通过编译优化选项。

通过这种方法大概可以缩小动态链接库尺寸 10%左右，但是性能提升的效果因具体程序而异，并不是特别明显。

（2）清除冗余代码。

这种方法通常适用于历史比较长的大型应用程序，有一些遗留代码已经不再有任何用途，这些冗余代码需要清除。如果可以去除的冗余代码占的比例比较大，那么这样的结果即可大大减少应用程序启动时引发的硬缺页的次数，从而提高性能。

9.3.3 优化可执行文件和库文件中的代码布局

如第 8 章所述，代码布局（Code Layout）对程序启动性能的影响也不可小视。优化代码布局就是通过重新把动态链接库中的函数排列得更加紧密从而达到减少 IO，提高性能的目的。

Windows 上优化代码布局的总体步骤分为两步，一是获得函数调用的顺序文件（.PRF，以动态链接库为单位）；二是把这些 PRF 文件传递给链接器，链接器会自动按照 PRF 文件把函数在动态链接库中的位置重新排列。

第 1 步获得 PRF 文件需要有强有力的工具的支持，因为必须知道应用程序启动时调用的所有函数的名字。

这里以 sws 工具（Smooth Working Set Tool，详见参考文献）为例，具体步骤如下。

1. 准备环境

（1）复制 SWSDLL.lib 到目标程序的编译库目录下。

复制 dbghelp.dll、SWS.exe、SWS.pdb、SWSFile.dll、SWSFile.pdb、SWSdll.dll、SWSdll.pdb、BugslayerUtil.dll 和 BugslayerUtil.pdb 到目标程序的安装目录下。

（2）安装 mssdk 准备 rebase 工具。

（3）修改 makefile。

- 为编译选项加上/Gh 开关，为链接选项加上 SWSDLL.lib：

```
CFLAGSSWS= /Gh
LINKFLAGSSWS= SWSDLL.lib
CFLAGS+= $(CFLAGSSWS)
LINKFLAGS+= $(LINKFLAGSSWS)
```

- 为最终的链接选项加上顺序指示文件：

```
REORDER =/ORDER:@ $(TARGET).PRF
$(LINK)  $(LINKFLAGS) $(REORDER) …
```

2. 嵌入_penter 代码

这一步的目的是在应用程序所有的函数前面嵌入一小段程序_penter，通过这段程序可以记录每一个函数被调用的次数和时机，为以后生成顺序文件做准备。

针对每一个需要函数重排的模块执行如下处理。

（1）删除旧的目标文件。

（2）重新编译加上_penter 代码的调试版本。

（3）复制该模块下的 .dll .pdb 到目标程序的安装目录。

（4）重复，直到完成所有需要函数重排的模块。

3. 生成顺序文件.PRF

在目标程序的安装目录下执行如下处理。

（1）rebase 所有的 dll。

（2）运行目标程序数次，sws 将会在目标程序的安装目录下针对每一个需要 reorder 模块的所有 dll 生成.SWS 和.SDW 文件。

（3）针对每一个有对应.SWS 文件的 dll 执行"sws -t <目标 dll 或 exe 名字>"，生成对应的.PRF 文件。

（4）复制所有的.PRF 文件到目标程序的源代码目录下面。

提示：

sws 的其他用法如下。

（1）sws -g <目标 dll>：模块名.SWS 只包含模块中的地址及用于存放执行计数的空间，模块名 .SDW 包含函数地址、函数大小，以及相关联的名称。module name.run number.SWS 数据文件中保存每个模块在一次运行中的执行计数。

（2）sws -d <目标 dll>：查看 dll 内部的函数顺序。

（3）sws -t <目标 dll 或 exe>：生成 prf 文件，.TWS 文件包含合计之后的执行计数（按从高到低的顺序排序）。

第 2 步的执行过程如下。

（1）依据顺序文件生成 dll。

（2）针对每一个需要函数重排的模块执行如下处理。

- 删除旧的目标文件。
- 重新依据顺序文件编译生成新的 dll，这是内部函数排好顺序的 dll。

通过函数重新排列的方式，通常可以获得 10%左右的启动性能提升。

9.4 优化源代码

除了优化可执行程序文件和库文件的方法之外，对于程序内部的源代码自身同样可能有优化的空间，本节说明实践中优化源代码的方法。

9.4.1 优化启动时读取的配置文件及帮助文件

一个大型应用程序除了包含可执行文件、库文件等与程序逻辑直接相关的文件外，通常还会由大量的其他文件组成。例如，包含用户配置信息的配置文件、包含帮助信息的帮助文件，以及包含图标信息及字符串信息等的资源文件，这些文件在启动时也会被访问。如果访问模式不合理，同样也会引起 IO 问题而导致程序启动性能下降。

对于这一类文件（以下统称为“配置文件”）优化的一个总的原则就是尽量合并。根据统计，在一个 CPU 为 2.6 GHz，硬盘为 5 400 r/m 转速的系统中，打开一次文件的操作就需要 40 ms 左右。如果应用程序包含 25 个配置文件，则需要至少 1 秒钟的打开时间。如果把这 25 个配置文件能够合并成一个配置文件，那么节省的时间将近 1 秒钟。

优化配置文件的工作最好在程序设计的初期就考虑到这样的性能问题，即设计一个简单高效的配置文件访问系统，程序启动的阶段存取的配置文件的个数尽量少一些。如果是优化一个已有的系统，则可能需要做较多的改动，甚至重构整个应用程序配置文件的代码。

9.4.2 预读频繁访问的文件

还有一种优化程序启动的方案是预读程序启动时需要频繁访问的文件，这种启动性能问题的特征如下。

假设应用程序在启动时需要频繁访问某一个文件 A，但是每次访问该文件的不同部分，每次访问的大小也没有规律。例如某大型应用程序有一个类似与注册信息数据库的文件，其中存储了数万条信息，程序启动时需要数千次访问该文件，但是每次访问只需要读取几个字节的信息，这种频繁随机的访问模式势必导致把大量的 IO 时间浪费在寻道时间上。

这种性能问题可以通过 filemon 之类的 IO 监测工具来发现。

相反地，如果一次性把需要频繁访问的文件全部预读入内存，那么这种操作由于是顺序访问，从而节省了大量的寻道时间。而后续的访问更只是内存操作，非常快速。

预读文件由如下两种方法实现。

（1）需要构造相应的数据结构，把文件读入内存并且解析后将其内容存放到一个内存

数据结构中，以供后续存取。这种方式是最全面的方法，但是可能需要新设计数据结构并编写大量代码，工作量比较大。

（2）利用操作系统的文件缓存特性，因为当前大部分硬盘自身也带有一定量大小的硬件缓存。程序启动时直接预读一遍待缓存的文件，但并不放在具体的数据结构中。由于操作系统会缓存该文件到内存中，从而也可以达到同样的效果并且代码量比较小。在预读时尽量用内存映射函数，如 Windosw 的 MapViewOfFile，Linux 的 mmap 等可以获得更好的性能。

在 Windows 操作系统中实现这种方法的示例代码如下：

```
int MapAndReadThroughFile(const char * pszName)
{
    //创建文件 handle.
    HANDLE hFile = ::CreateFile (
        pszName,
        GENERIC_READ,
        FILE_SHARE_READ,
        (LPSECURITY_ATTRIBUTES)NULL,
        OPEN_EXISTING,
        (FILE_ATTRIBUTE_NORMAL | FILE_FLAG_RANDOM_ACCESS),
        (HANDLE)NULL);

    // 判断是否创建成功
    if (hFile == INVALID_HANDLE_VALUE)
    {
        hFile = 0;
        printf("**********%s not found! ****************\n", pszName);
        return -1;
    }

    HANDLE hMap = ::CreateFileMapping ( hFile, NULL, SEC_COMMIT | PAGE_READONLY,
0, 0, NULL);
    if (hMap == NULL || hMap == INVALID_HANDLE_VALUE)
    {
        printf("Could not create file mapping object.\n");
       return -1;
    }
```



```
    //映射内存文件
int nSize = ::GetFileSize (hFile, NULL);
char * pBuf = (char*)::MapViewOfFile (hMap, FILE_MAP_READ, 0, 0, nSize);
if (pBuf == NULL)
{
    printf("Could not map view of file");
    return -2;
}
char buf[512];
for (int i =0; i<nSize/512; i++)
{
    //把该映射文件顺序读入内存，操作系统会自动放入缓存中
    CopyMemory(buf, (pBuf+i*512), 512);
}


UnmapViewOfFile(pBuf);

CloseHandle(hMap);

return 0;
}
```

9.4.3　清除产生 exception 的代码

例如在 VC 中可以直接检测到“first chance exception”，如图 9-2 所示。

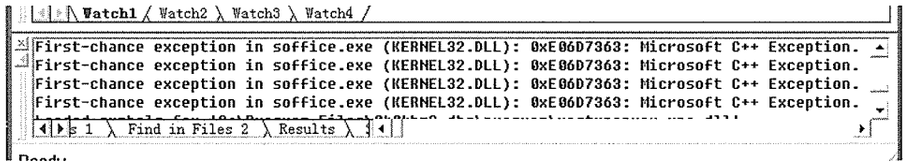


图 9-2　检测到“first chance exception”

一次 exception 会导致程序脱离正常的流程，带来一系列异常处理。而异常处理的代码通常又不在应用程序的工作集中，从而引发缺页中断，导致性能下降。应用程序应该在启动过程中尽量清除这些产生异常的代码，避免性能下降。

9.4.4 PreLoad

可以通过构造一个和应用程序对应的独立 Preload 程序来预先加载应用程序启动时需要的动态链接库和其他数据。这样当应用程序启动时，它所需要的大部分代码页和数据页都已经出现在内存中，这样可以避免相应的缺页中断。

在操作系统启动时，PreLoad 程序作为一个后台的服务启动，使得用户使用应用程序时可以更快速地启动。

因为 PreLoad 程序需要从操作系统启动后一直驻留在内存中，所以在设计 PreLoad 时要注意到内存占用的问题和 CPU 消耗的问题。

另外 PreLoad 在实现时注意也要考虑到应用程序在卸载和升级时候的情况，即卸载时需要实现退出 PreLoad 程序，升级时需要退出后再恢复。

9.4.5 延迟初始化

可以考虑把一些程序启动初期不需要的初始化工作推迟到程序启动之后完成，让用户获得快速启动的体验。

具体延迟的时机可以在程序空闲时处理，例如：

```
    void IdleProc()
    {
MSG aMsg;
//如果检测到消息队列中的鼠标事件，则表明不是在空闲时段
        If (PeekMessageW ( &aMsg, 0, WM_MOUSEFIRST, WM_MOUSELAST,
                           PM_NOREMOVE | PM_NOYIELD ) )
        Return;

//如果检测到键盘事件，同样不是空闲。
If  (PeekMessageW ( &aMsg, 0, WM_KEYDOWN, WM_KEYDOWN,
                           PM_NOREMOVE | PM_NOYIELD ))
        Return;

        DoSomeIdleJobs();
}
```


或者在需要该资源时初始化，使用 singleton 模式，例如：

```
void ShowSomething()
{
    if (NULL==gHelpInstance)
    {
        //此处开始初始化需要的资源：
        gHelpInstance=createHelpInstance();
        if (NULL==gHelpInstance)
            return;
    }
    gHelpInstance.Show();
    return;
}
```

可以延迟初始化的工作分类如下。

（1）启动时加载和初始化不需要的模块。

（2）一些帮助信息的初始化。

（3）启动时不显示的菜单及工具栏相关的资源。

（4）必须由用户操作后才需要的资源。

9.4.6 多线程化启动

如果应用程序具有如下的特点，则适合多线程化启动。

启动时需要加载大量的动态链接库，引发大量的 IO 操作。同时这些动态链接库的初始化函数又需要执行很多计算密集型的操作，长时间占用 CPU 的时间。

这时可以考虑把应用程序的 IO 等待时间和 CPU 占用时间交错并行处理，从而减少总的启动时间。

9.5 本章小结

应用程序启动性能优化的步骤和普通的性能优化步骤基本一致，是一个迭代循环的过

程，即测试->发现问题->优化->验证。为了保证每一次的优化方案可验证，需要一个精确稳定的启动性能的测试环境。

本章详细介绍了应用程序启动性能优化的方法。在了解这些方法的基础上，应用程序开发人员应该从设计阶段就开始关注程序的启动性能，尽量设计出紧凑且体积小的程序代码，减少启动过程中引发的 IO 操作。

第 4 篇　性能工具

软件的规模越来越大，软件的性能问题逐渐突出。软件的性能很大程度上取决于软件的总体设计，但是运行时软件的局部性能问题可以被发现并解决。性能工具可以发现软件局部的性能问题，没有它的帮助，很难想象一个开发团队能写出质量很高的大型软件。性能工具不仅能迅速定位程序中某些隐含的错误，还能帮助开发人员找到运行时程序的某些性能瓶颈，有效地提高软件质量。本篇将为读者介绍 3 类性能工具，分别是内存分析工具、性能分析工具和 IO 监测工具。

第 10 章介绍内存分析工具 IBM Rational Purify，它可以用来查找各种内存错误，如数组越界访问和内存泄漏等。

第 11 章介绍性能分析工具 IBM Rational Quantify，它主要用于提高软件性能。这个工具能提供完备、准确的性能数据，以方便开发人员定位程序瓶颈。

第 12 章介绍 IO 监测工具 FileMon，它可以实时监测与文件相关的各种操作，为分析应用程序的启动性能提供客观数据。

第 10 章　内存分析工具 IBM Rational Purify

Rational Purify 是 IBM 的软件测试工具集合 Rational PurifyPlus 中的一员。Rational PurifyPlus 主要面向.NET、Visual C/C++、Java，以及 Visual Basic 的应用程序，它包括 3 个工具——Rational Purify、Rational PureCoverage 和 Rational Quantify。其中 Rational Purify 主要用于查找各种内存错误。在.NET 和 Java 应用程序中，它能快速地定位包括垃圾回收在内的各种内存管理问题；在 Visual C/C++应用程序中，它能查找各种传统的内存访问和使用错误，如数组越界访问和内存泄漏等。本章将为读者详细介绍这个工具。

与很多主流程序调试及性能检测工具相比，Rational Purify 的使用相对简单，它可以作为插件与某些 IDE 开发环境集成在一起（例如，Microsoft Visual Studio），开发人员在使用 IDE 开发过程中可以快速获得 Rational Purify 的自动调试和源码编辑功能。另外，Rational Purify 带有及时调试功能。当检测到错误时，它将自动停止编程并启动调试器。开发人员也可以通过 Rational Purify 工具栏，将该调试器附加到正在运行的流程中。这将大大增强诊断应用程序问题的能力，从而缩短查找、复审和修正错误所需的时间。通过本章，读者将学会如何使用这种功能强大的性能调试工具，并了解其工作原理。

10.1 Rational Purify 工作原理

在介绍 Rational Purify 的工作原理之前，我们首先回顾经常让 C++开发人员感觉“生不如死”的常见内存使用错误。

（1）内存泄漏。

内存泄漏常常被 C++开发人员认为是产生程序错误的万恶之首。当分配的内存没有被正确释放时，这些看似不起眼的“遗忘”内存块可能导致系统资源耗尽。更让人烦恼的是，这种错误一般不会在短时间内被发现。如果系统结构很复杂，错误定位也是一件很困难的事情。

（2）数组越界读/写。

这种错误的特点是数组的读写可能在大多数情况下表现正常，但是当某个特殊条件产生时，错误便产生了。

（3）访问未初始化的内存。

这也是很多入门级开发人员经常遇到的问题，结果是导致不可预知的程序错误。这是一个观念上的问题，很多人没有养成在使用指针前要初始化这样的习惯，或主观地认为自己申请的内存的默认值为 0。无论如何，养成初始化内存的好习惯是很重要的。

（4）访问已经释放的内存。

当程序逻辑变得越来越复杂时，一个小小的失误可能导致某个函数继续访问一块已经被释放的内存，这种问题是导致大型程序崩溃的主要原因之一。

对于各种内存使用问题，Rational Purify 使用了具有专利技术的目标代码插入（Object Code Insertion）技术，即通过在程序的目标代码中插入特殊指令来检查内存的状态和使用情况。这种技术的优点是不需要修改源代码，只需要重新编译程序即可对其进行分析。

在运行过程中，Rational Purify 监视源程序的每个内存操作，决定其是否合法。它建立一个状态表，跟踪系统内存中每个字节的状态。状态表用两位保存字节的状态信息，第 1 位记录这个字节是否已被分配；第 2 位记录其是否已被初始化。如图 10-1 所示，利用这两位，Rational Purify 能够跟踪系统中的 4 种内存。即未分配且未初始化的内存（红色）、已分配但是未初始化的内存（黄色）、已分配且已初始化的内存（绿色）和未分配但已初始化的内存（蓝色）。

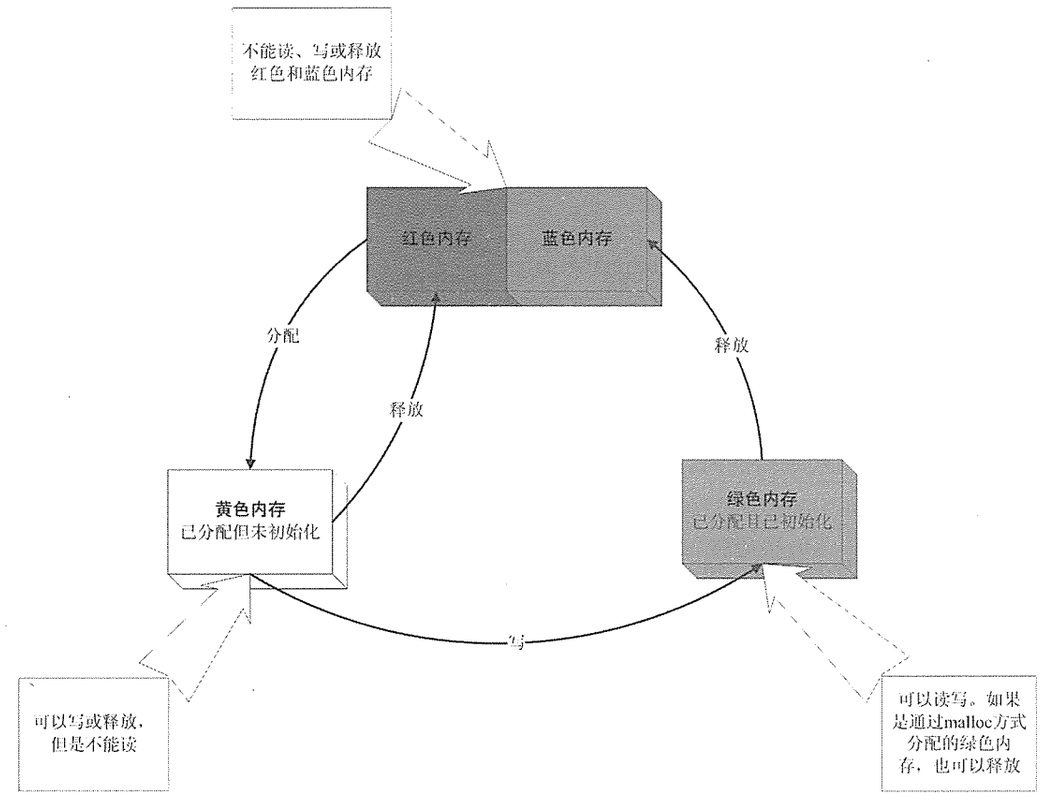


图 10-1 4 种内存状态

对于程序的每一种内存操作，Rational Purify 根据对应的状态信息判别其是否合法。如果非法，Rational Purify 将报错。

（1）红色内存。

开始，Rational Purify 将堆和栈置为红色，表示未分配且未初始化。分配内存后，Rational Purify 在每个分配的动态内存块和静态数据项前后插入“红色”保护区域，用来检查数组边界错误。对这些区域执行读、写或是内存释放操作都是非法的，因为对应的内存不属于被检测程序。需要特别指出的是，访问未初始化内存错误在某些情况下其实是合法的操作，如内存拷贝。

（2）黄色内存。

程序使用 new/malloc 分配的内存被置为黄色，这些内存虽然已被分配，但是仍未初始化。因此只能写或释放它们，而读操作是非法的。例如，在函数入口 Rational Purify 将堆栈的帧置为黄色。

（3）绿色内存。

当向黄色内存写入数据后，Rational Purify 将其置为绿色。这意味着该内存已被程序分配和初始化，此时读、写或是释放绿色内存都是合法的。例如，Rational Purify 初始化数据段和 bss 段的内存。

（4）蓝色内存。

当释放一段已经使用的内存后，Rational Purify 将其置为蓝色。这时该内存的状态是“未分配但已初始化”，对其执行任何访问都是非法的。

除了检查各种动态内存错误之外，Rational Purify 也能探测对各种全局变量和静态变量的非法引用。Rational Purify 将红色区域插入到程序中每个静态数据区的周围，如果程序试图从这些红色区域读或写入数据，Rational Purify 将报告一个数组越界错误。

对于数据段的内存，仅当所有引用都针对已知变量时，Rational Purify 才插入红色区域。因为如果 Rational Purify 找到一个数据引用，关联到从数据段起始位置到某个已知变量之间的一段内存，则其无法决定这个引用究竟包括了哪个变量。在这种情况下，Rational Purify 仅在数据段的起始和结束位置周围插入红色区域，而不是在各个数据变量之间。

10.2 Rational Purify 使用指南

Rational Purify 的使用非常方便，一个开发人员只要稍加学习便能轻松掌握。本节将结合程序实例介绍 Rational Purify 的使用，首先说明各种参数配置。

1. Setting 中的 Default Setting 选项

在 Default Setting 选项集中，用户可以设置错误信息查找、显示或文件缓存等配置项。以后每次运行 Rational Purify，程序都会遵照这些设置错误检查。

（1）Error and Leaks 选项卡，如图 10-2 所示。

配置项	说　明
Show first message only	在相同错误第 1 次出现时显示信息
Show UMC message	显示 UMC 信息
Memory leaks	程序退出时报告内存泄漏情况
Memory in use	退出时报告内存使用情况
Handles in use	退出时报告句柄使用情况
Show maximum call stack detail	显示最大调用堆栈细节
Deferred free queue	设置延迟释放队列长度和阈值，这两个值越大，用来缓存非法指针访问的内存就越多，捕获的 FMR（Free Memory Read）和 FMW（Free Memory Write）错误消息也越多
Red zone length	红色区域长度，该区域越大，捕获的越界读写错误越多

（2）PowerCheck 选项卡，如图 10-3 所示

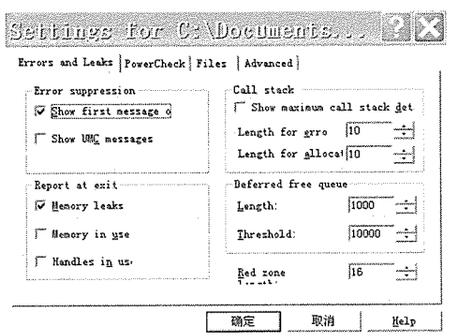


图 10-2　Error and Leaks 选项卡

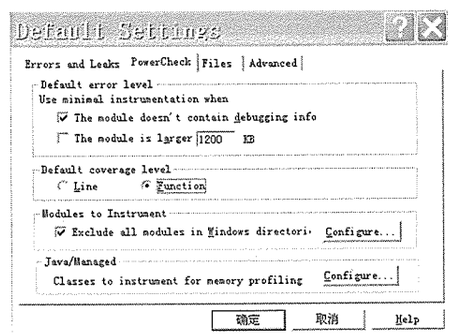


图 10-3　PowerCheck 选项卡

配置项	说　明
Default error level	默认的错误检测级别
The module doesn't contain debugging info	当模块不包含调试信息时（使用最小检测代码集 Minimal Instrumentation，异常信息、栈的内存错误和大部分堆内存错误，除了 Windows API 函数引起之外，都无法检测）
The module is larger…KB	当模块大于……KB 时（使用最小检测代码集）
Default coverage level	默认的覆盖标准
Exclude all modules in Windows directory	排除所有 Windows 目录下的模块
Class to instrument for memory profiling	对于各种托管代码（Managed Code，如 Java 代码和.Net 托管代码），Purify 通过 memory profiling 方式收集性能数据。通过此选项，用户可以选择会参与将被检测的 Java 类

（3）Files 选项卡，如图 10-4 所示。

配置项	说　明
Cache directory	Rational 运行分析工具插入目标代码时，缓存各种待检测文件（包括可执行文件和 DLL 等）的目录
Source file search	搜索源文件

（4）Advanced 选项卡，如图 10-5 所示。

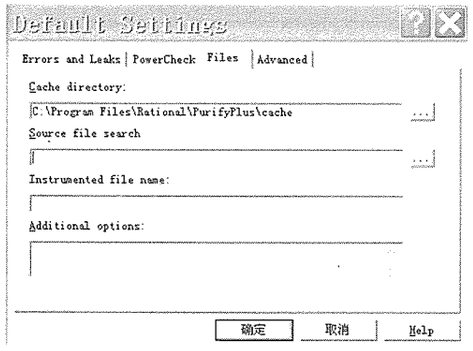


图 10-4　Files 选项卡

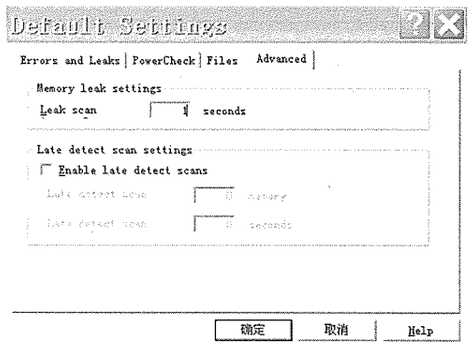


图 10-5　Advanced 选项卡

配置项	说　明
Leak scan…seconds	内存泄漏扫描时间间隔
Late detect scan…memory	检查每个已分配堆栈块和延迟释放队列中堆栈块红色区域，设置检测的内存大小
Late detect scan…seconds	设置 Late detect scan 的间隔时间

2. Setting 中的 Preferences 选项

在 Preferences 选项集合中，用户可以个性化地设置 Rational Purify 的各项功能。

（1）Runs 选项卡，如图 10-6 所示。

配置项	说　明
Show instrumentation progress	是否显示检测进程
Show instrumention warnings	在检测程序时，如果查找到警报信息，是否显示
Show LoadLibrary instrumentation progress	当需要调用库文件时，是否显示检测进程
Confirm run cancellation	当用户选择取消程序运行时，是否弹出证实信息
Create automatic merge	（针对不同测量结果）是否自动合并
Use default filter set	使用默认的过滤器设置
Use case sensitive path name	路径名区分大小写
Break on warning in addition to error	当错误增加时中断调试
Use the following debugger	使用指定的调试工具

（2）Workspace 选项卡，如图 10-7 所示。

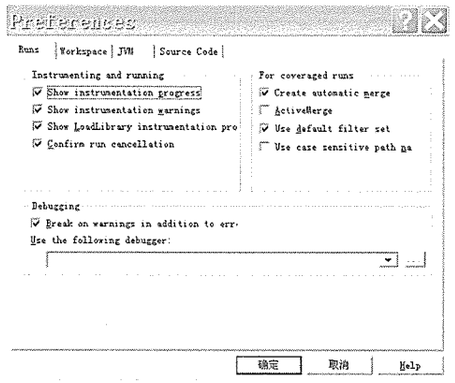


图 10-6　Runs 选项卡

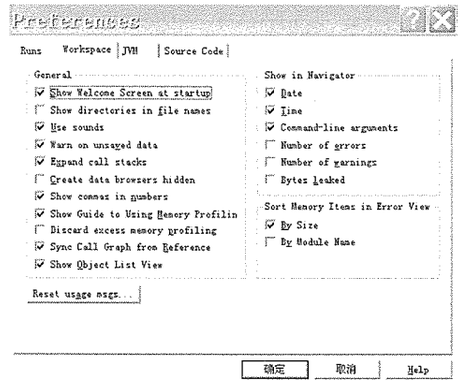


图 10-7　Workspace 选项卡

配置项	说　明
Show welcome screen at startup	当每次启动 Purify 时，是否显示欢迎对话框
Show directories in file names	是否显示完整的文件路径名
Use Sounds	开启/关闭声音
Warn on unsaved data	当关闭一个尚未保存数据的 Purify 程序时，是否提示用户
Expand call stacks	增加堆栈调用的个数

续表

配置项	说　明
Create data browsers hidden	创建隐藏的数据浏览器，数据浏览器包括以下表项：错误视图（Error View）、模块视图列表（Module View Tab）、文件视图列表（File View Tab）及函数队列视图表（Function List View Tab）
Show commas in numbers	在数字中显示逗号分隔
Show Guide to Using Memory profiling	当使用 Profile 方式检测内存错误时，显示向导信息
Discard excess memory profiling	放弃多余的内存 profiling 数据
Sync Call Graph From Reference	根据参考检测结果，同步调用关系图
Show Object List View	显示对象列表视图
Show in Navigator	在导航窗口中是否显示日期、时间及命令行等信息
Sort Memory Items in Error View	选择以何种方式在错误视图中显示内存项（根据大小或模块名）

（3）JVM 选项卡，如图 10-8 所示。

用户可以通过这个选项选择待检测程序运行在哪种 Java 虚拟机上。

（4）Source Code 选项卡，如图 10-9 所示。

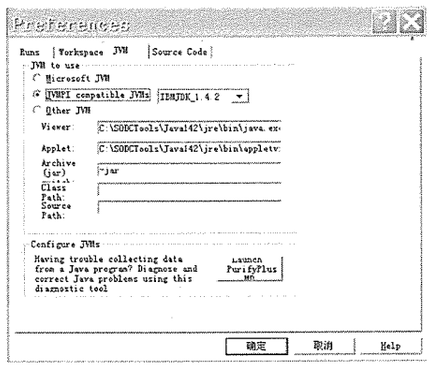


图 10-8　JVM 选项卡

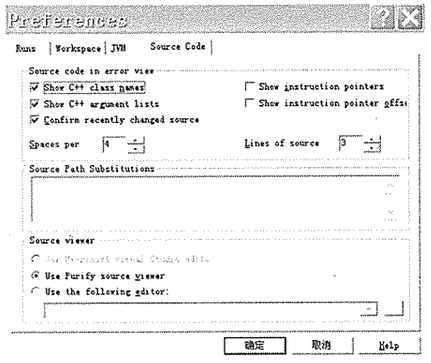


图 10-9　Source Code 选项卡

配置项	说　明
Show C++ class names	显示 C++类名
Show C++ argument lists	显示 C++参数列表
Confirm recently changed source	确认最近修改的源代码
Show instruction pointers	显示指令指针
Show instruction pointers offset	显示指令指针的偏移量
Spaces per	设置空格大小

续表

配置项	说　明
Lines of source	设置显示的源码行数
Source view	选择查看源码所使用的阅读器，可选项有微软 Visual Studio 编辑器、Purify 源码阅读器及用户自选的阅读器

3．各种错误类型

Purify 中定义的各种错误如下表所示。

错误名	说　明
ABR	数组越界读
ABW	数组越界写
ABWL	程序在已分配内存块的头之前或尾之后写入数据
Beyond stack read	栈越界读
Beyond stack write	栈越界写
COM	COM 操作失败，包括 COM API 调用失败和接口调用失败
EXC	连续弹出异常
EXU	未处理的异常
FFM	重复释放内存
FIM	释放不合法的内存（如未分配的）
FMM	释放了不匹配的内存
FMR	自由内存（未分配的）读
FMW	自由内存写
FMWL	向已释放的内存写数据，或从一个尚未分配的堆内存中引用数据
HAN	非法句柄
ILK	COM 接口泄漏
IPR	非法指针读
IPW	非法指针写
MAF	内存分配失败
MIU	内存正在使用
MLK	内存泄漏
MPK	潜在的内存泄漏
NPR	空指针读
NPW	空指针写
PAR	坏参数
UMC	拷贝未做初始化的内存
UMR	读未做初始化的内存

10.3 Rational Purify 实例分析

本节将结合一个“乘法矩阵链”的程序实例详细描述 Rational Purify 的使用。本程序用 Visual C++6.0 编写，其主要功能如下。

（1）自动生成一个乘法矩阵链 N 个矩阵构成的一个链 M1，M2，……，Mn，矩阵 Mi 的维数为 Di-1×Di(1≤i≤n)，并将其存入文件 matrix.log。下次程序启动时，直接从该文件中读取数据。

（2）线性矩阵连乘，即顺序地对矩阵链中的所有矩阵进行连乘，在文件中打印结果。

下面使用 Rational Purify 对其进行检查。

（1）打开 Rational Purify 程序，其初始界面如图 10-10 所示。

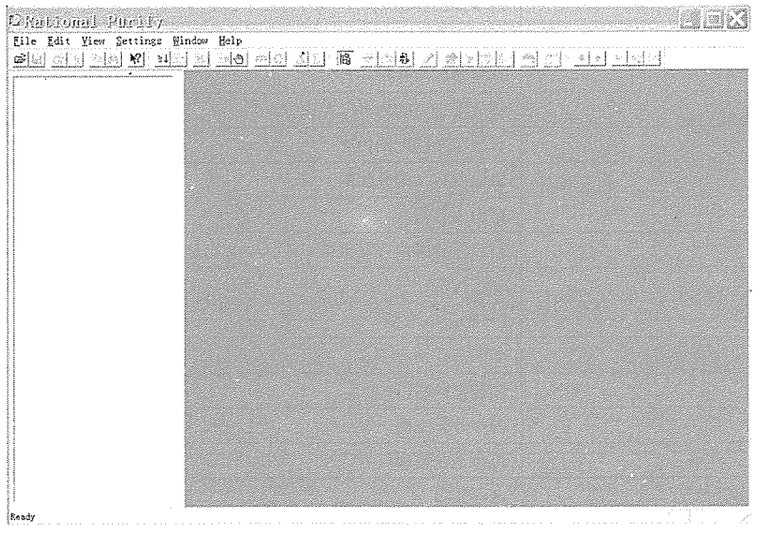


图 10-10 Rational Purify 初始界面

（2）为测试“乘法矩阵链”程序，选择 File->Run 选项，出现如图 10-11 所示的 Run Program 对话框。

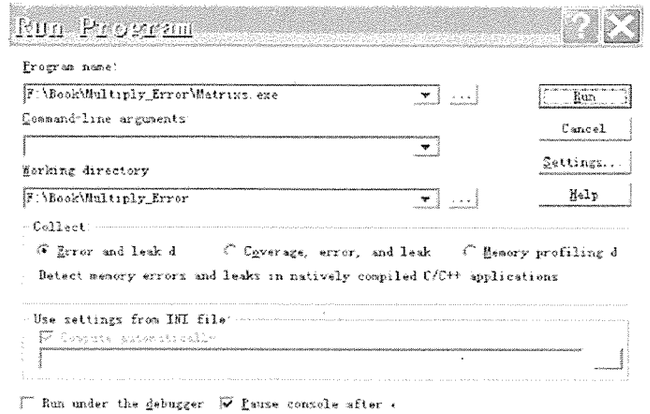


图 10-11 Run Program 对话框

选择被测程序的路径以及输入命令行参数。在Collect选项组中选择要收集的信息类别，对于非托管的 C/C++代码，选择第 1 项。即 Error and leak detect 单选按钮，然后单击 Run 按钮运行程序。

（3）查找错误。程序运行结束后，显示如图 10-12 所示的 Rational Purify 的检查结果。

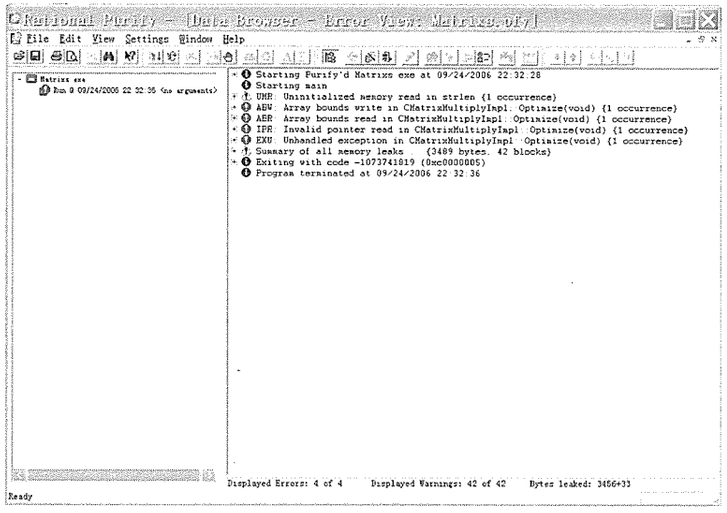


图 10-12 检查结果

从图中发现，UMR（读未初始缓存）、ABW（写数组越界）、ABR（读数组越界）、IPR（读非法指针）、EXU（未处理的异常）及内存泄漏各一处。

（4）排查错误。我们首先排查 UMR 错误，如图 10-13 所示。

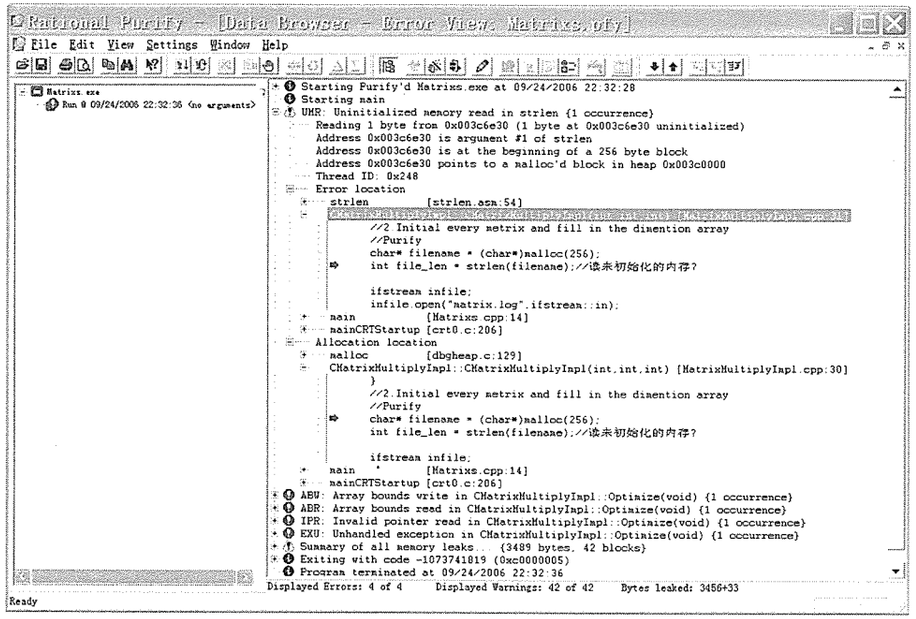


图 10-13　排查 UMR 错误

原来错误原因是为 filename 分配了内存之后，尚未初始化，就使用 strlen 函数对其进行访问。

接下来排查除内存泄漏之外的其他几处错误，如图 10-14 所示。

可以发现，我们在 CMatrixMultiplyImpl 类的构造函数中为 CostTable 和 LookUpTable 设定的长度是 m_numMat，然而在 Optimize 函数中却对其进行了越界访问（正确的 for 循环的上限条件应该为 i<(m_numMat-len+1)）。这也是 ABW、IPR 和 EXU 的错误根源，因为在该 for 循环中也对 CostTable 数组执行写操作。

最后我们排查内存泄漏问题，如图 10-15 所示。

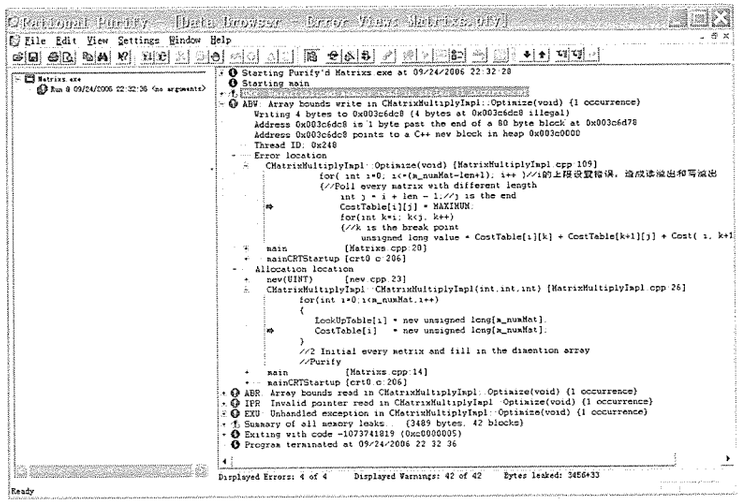


图 10-14　排查除内存泄漏之外的其他几处错误

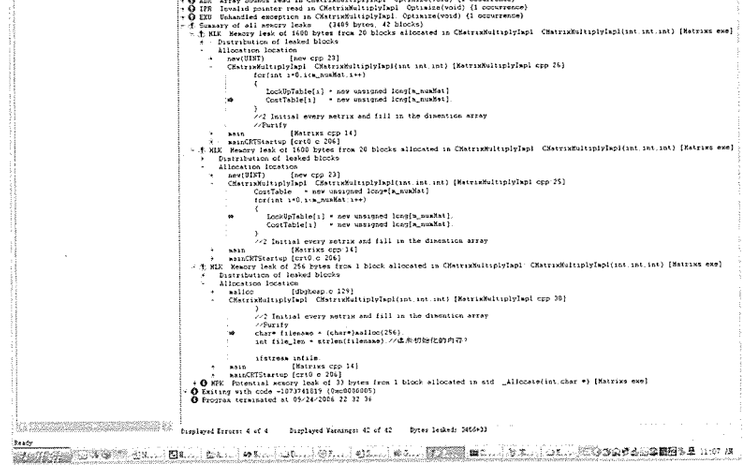


图 10-15　排查内存泄漏问题

在程序中共有 3 处内存泄漏，即未释放在 for 循环中为 LookUpTable[i]分配的内存、未释放在 for 循环中为 CostTable[i]分配的内存及未释放为 filename 分配的内存。

现在，我们已经找到了“乘法矩阵链”程序中所有的错误，将其一一改正后，Ralional Purify 的运行结果如图 10-16 所示。

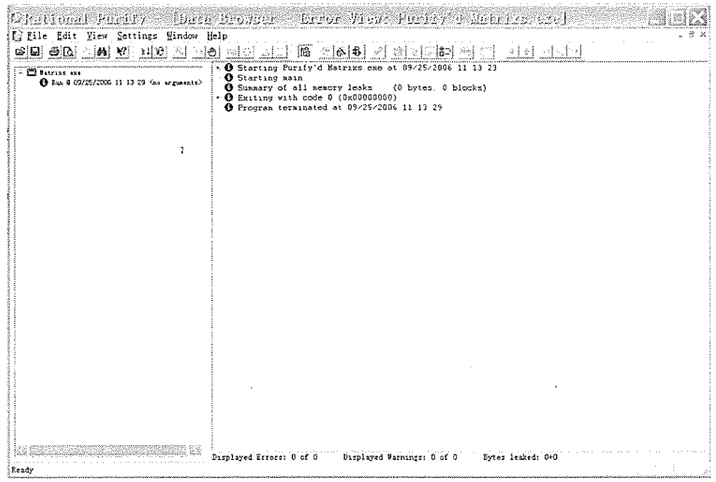


图 10-16 没有内存错误的“乘法矩阵链”程序

10.4 本章小结

本章介绍了 Rational Purify 的主要功能及工作原理，并结合“乘法矩阵链”程序实例为读者详细描述了其使用方法。不难发现，Rational Purify 可以从多个侧面反映应用程序的质量——功能、可靠性和性能。通常，软件测试人员只有在进行功能测试遇到可靠性问题时，才会发现它们。因为与内存相关引起应用程序崩溃的编程错误，并不一定会出现在运行该程序的每台计算机上。这些错误在开发和单元测试时可能看不到，有时只是在最终用户使用此软件时才会显现出来，结果只好发布一个又一个的补丁程序来解决这些始料未及的问题。Rational Purify 通过检测影响可靠性的相关内存错误，可以提高 Java 和 C++软件的质量。它在执行功能测试的同时检测可靠性问题，从而弥补质量测试的不足，为开发人员提供了修正问题所需的所有诊断信息。Rational Purify 还能减少错误相互“遮挡”而导致的“测试—修正”循环所花费的大量时间。Rational Purify 主动搜索并记录与内存相关的编程错误，而不是消极地等待应用程序崩溃，使开发人员可以同时查找多个错误并减少软件发布之前所需的“测试—修正”循环次数。

第 11 章 性能分析工具 IBM Rational Quantify

性能始终是一个程序员开发应用软件时最需要关注的要点之一。如果程序占用资源过多或运行速度过慢，无论其功能如何强大，都难以被用户接受。然而，开发一个高性能的软件无论如何都不是一件简单的事情。代码中纷繁复杂的交互操作，数目众多的 DLL 文件，甚至操作系统、硬件、网络，以及其他进程都可能是引起性能下降的原因。本章将为读者介绍一个强大的性能分析工具 IBM Rational Quantify。作为 Rational 家族中的重要一员，Quantify 能够快速准确的定位程序性能瓶颈，是软件开发人员的得力助手。

80—20 法则告诉我们，一个程序 80%的资源用于 20%的代码上（虽然这只是一条经验法则，但至少反映了大部分我们正在使用程序的状况）。在一个大型的软件中，常常存在某些瓶颈，制约了软件的性能和执行效率。通常，我们在设计时已经获知某些瓶颈的位置。但也常常遇到这种情况，即软件的初期版本的性能与预计的仍有较大差距。

定位这些未知的性能瓶颈是一件枯燥而又困难的事情（相信很多开发人员都曾有过这样的经验）。如果有一种工具能够快速帮助我们找到程序中哪个部分用了最多的时间，无疑是整个软件开发部门的福音。很明显，Rational PurifyPlus 的自动化测试工具 Rational Quantify 能够满足这个要求。Rational Quantify 是一个面向 Visual C++、Visual Basic，以及 Java 开发环境的程序性能瓶颈测试工具，它可以自动检测出影响程序执行速度的性能瓶颈，并提供参数分析，以供改进所需。Rational Quantify 可以在不同粒度（代码行级和函数级）上测定性能，使软件开发人员能逐步定位，直至找到影响程序性能的“罪魁祸首”。使用 Rational Quantify 特有的 PowerTune 功能，开发人员可以更好地控制数据记录的速度和准确性。另外，用户也可以按模块调整 Rational Quantify 收集信息的级别。对于程序中感兴趣的模块可以收集详细信息；而对于不太重要的模块，可以加快数据记录的速度并简化收集过程。

用户还可以实时地控制性能数据的记录。Rational Quantify 可以收集应用程序在整个运行过程中的性能数据，也可以只收集用户最感兴趣的某些阶段的数据。当用户运行 Rational Quantify 时，还可以收集有关应用程序及其使用的每个构件的性能数据集。更重要的是，Rational Quantify 还提供了强大的分析功能，帮助用户充分利用性能数据和时间。例如，Rational Quantify 的 Diff 分析功能可以用图形方式比较两次运行的执行时间，测定所做更改产生的正面或负面影响，以帮助用户核实所做的更改是否正确。Rational Quantify 的 Merge 功能使用户可以总结任意多次运行和任意多个应用程序产生的性能数据，这样可以用开发人员能够理解的语言为其提供所需的信息，以便调整特定构件，达到可执行文件或程序执行的最佳整体性能。

11.1 Rational Quantify 工作原理

IBM Rational 的专利技术——目标代码插入（OCI）是 PurifyPlus 工具集的核心技术。Rational Quantify 使用这项技术对程序执行的指令进行计数，同时它还计算执行每条指令需

要花费多少个机器时钟周期。计算时钟周期意味着每次执行程序后，Rational Quantify 记录的时间总是相等的（当然前提是输入不变，而且程序正确）。这种完全的可重复性使用户可以精确地计算出算法和数据结构的改进对程序性能的提高。

由于 Rational Quantify 统计的是时钟周期，因此能在任意范围内提供准确的性能数据。用户不需要像使用 Profile 那样，重复多次运行程序以求时间平均值的方法来测试性能。只要使用 Rational Quantify 运行一个测试程序并保存结果，这些数据就可以作为今后与改进版本比较的基准。

Rational Quantify 对程序执行时每个系统调用所花费的时间都进行了精确测量（利用机器时钟），并在程序结束后向用户报告该程序到底用了多少时间来完成这些调用。当程序改进后，用户可以立即看到改进的效果，例如 I/O 操作时间和网络延时等。用户也可以选择采用操作系统内核的计时系统来测量这些调用。Rational Quantify 可以精确计算出每个函数被不同调用者调用的时间，这样我们就能找到占用了大部分执行时间的函数。

Rational Quantify 可以在不同的源码级别上记录性能数据。

（1）行。

在此模式下，Rational Quantify 按行级提供性能数据。此时，测量性能的代价最高，即需要相对最多的时间来收集数据。测量结束后，性能数据可以在 Annotated Source Window 中显示。对于 Visual C++的未托管代码（native code）程序，Quantify 会根据设备的型号，计算执行每行源代码的机器周期总数。

（2）函数。

函数级与行级的时间精确度相同，但数据更简略。当用户不需要确切知道每一行代码的执行情况时，函数级的测量往往是最佳选择。对于非托管代码程序，Rational Quantify 根据调试信息标识出每个函数，然后计算每个函数或过程所需的时钟周期总数。

（3）时间。

在此模式下，Rational Quantify 使用 Profile 方式记录程序花费在每个函数上的时间。由于 Rational Quantify 仅在函数退出时计算时间，而不像其他计算时钟周期的方式那样，增量地在每个内部分支或函数调用发生时进行计算，计时方式能够提供更好的运行时性能

（runtime performance）。在这种模式下，Rational Quantify 仅记录被测试程序的每个输出函数的时间数据，而其内部函数的执行时间将作为属性关联到输出函数。使用计时方式测量的结果是精确的，但是它们常常会受到处理器和内存状态的影响，因此是不可重复的。当无法修改模块中的代码时（说明无法使用目标代码插入技术，例如测试系统模块或第三方模块），计时方式是测试程序性能的一个最佳选择。需要说明的是，计时方式仅适用于非托管代码。

Rational Quantify 提供了 4 种计时方法，即用户计时（User Time）、用户＋内核计时（User + Kernel Time）、内核计时（Kernel Time）及共用计时（Elapsed Time）。对于某个函数，Rational Quantify 究竟采用何种计时方法，以及测量类型，取决于函数落在哪个分类之中，如下表所示。

函数分类	函数类型	默认计时方法	测量类型
用户模块	选择使用计时方式，而不是计算机器周期方式的函数	用户计时	Timed
系统模块	除了正在阻塞和等待之外的其他系统函数	用户＋内核计时	System
阻塞或等待的函数	等待 I/O 操作的系统函数，如 WriteFile 或 ReadFile，或是阻塞在同步方式下的各种对象，如 WaitForSingleObject	共用计时	Sys Block/Sys Wait

11.2 Rational Quantify 使用指南

与第 10 章介绍的 Rational Purify 相似，Rational Quantify 的使用非常方便。本章我们仍将结合执行矩阵链运算的程序实例详细介绍 Rational Quantify 的使用方法。

1. Setting 中的 Default Setting 选项

通过设定默认配置，用户在每次运行 Rational Quantify 检查程序（及其修改版本）的性能时，可以保证数据采集方式的一致性。

（1）PowerTune 选项卡，如图 11-1 所示。

配置项	说　明
Default Measurement	默认测量级别，可以选择行级、函数级或时间级
Time all modules in Windows directory	是否对 Windows 目录下的所有模块计时

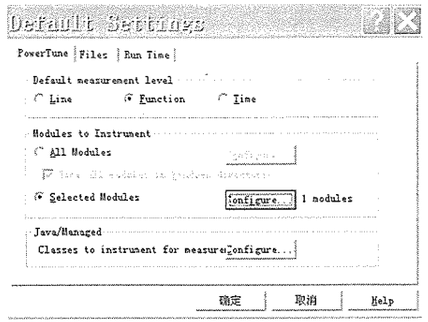


图 11-1　PowerTune 选项卡

（2）Files 选项卡。

用户可以在其中设置默认缓存文件的存放位置，与 Rational Purify 完全相同，此处不再赘述。

（3）RunTime 选项卡，如图 11-2 所示。

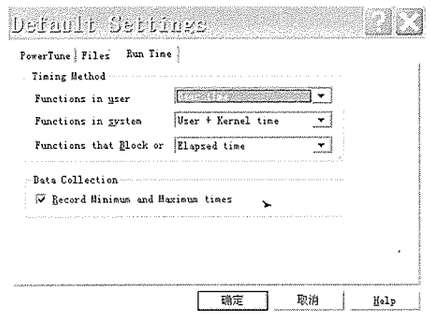


图 11-2　Run Time 选项卡

配置项	说　明
Time Method	计时方法，可以分别选择在执行用户函数、系统函数和阻塞/等待函数时采用的计时方法，包括用户计时、用户计时＋内核计时，内核计时，以及共用计时。另外，也可以选择忽略这类函数
Data Collection	数据收集，选择是否在测试中记录函数运行的最小及最大时间。不记录这些时间可以提高程序运行的速度

2. Setting 中的 Preferences 选项

Preferences 选项允许用户设置个性化测设过程，仅保留并显示需要的数据。

（1）Runs 选项卡，如图 11-3 所示。

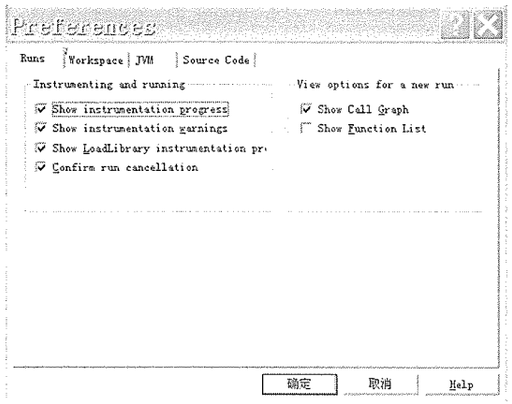


图 11-3　Runs 选项卡

配置项	说　明
Show instrumentation Progress	显示程序测试进展
Show instrumentation warning	显示程序警报信息
Show LoadLibrary instrumentation progress	显示库文件调用进展，如果未选此项，程序结束后源文件列表将显示为空
Confirm run cancellation	确认能够取消程序运行
Show Call Graph	当程序运行结束后，显示函数调用图表
Show Function List	当程序运行结束后，显示函数性能分析表

（2）Workspace 项

Rational Quantify 的各种用户交互配置，与 Rational Purify 相同，不再赘述。

（3）JVM 选项卡

用户可以通过这个选项选择测试程序运行在哪种 Java 虚拟机上，与 Rational Purify 相同，不再赘述。

11.3　Rational Quantify 实例分析

我们回到第 10 章介绍的乘法矩阵链的例子，该程序的主要功能是自动生成 200 个满足连乘条件的矩阵并将其在文件中打印出来，随后执行矩阵链的乘法。现在我们查看 Rational Quantify 如何定位程序的性能瓶颈。

（1）打开 Rational Quantify。如图 11-4 所示，Rational Quantify 的界面与 Rational Purify 非常相似。事实上，PurifyPlus 的 3 个工具虽然功能各不相同，但是界面和使用方法却基本一致。用户只要掌握其中一种工具，即可很快掌握其他两种。

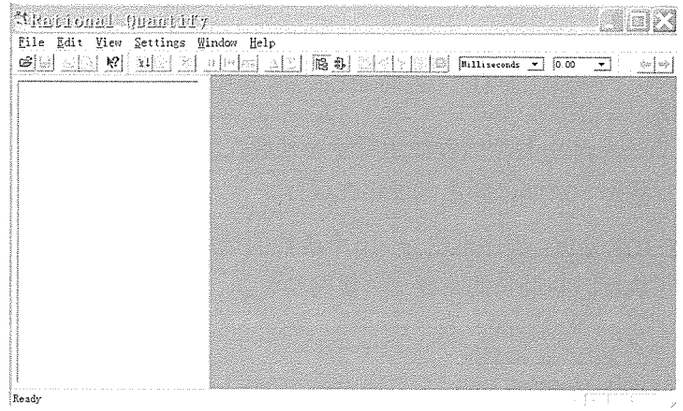


图 11-4　Rational Quantify 的初始界面

与 Rational Purify 的惟一不同是在界面的右上角，用户可以选择时间单位，共有 4 项。即微秒（microseconds）、毫秒（milliseconds）、秒（seconds）及机器时钟周期（machine cycles），用户可以根据实际需要选择本次测量的时间粒度。

（2）运行 matrixs.exe 文件，选择 file->Run 选项，输入需要测试程序的存放路径以及工作目录。如果程序需要输入参数或是读取配置文件，可以在对应的“command line arguments”和“Use setting from INI from”中输入参数的值。然后单击 Run 按钮开始执行程序，如图 11-5 所示。

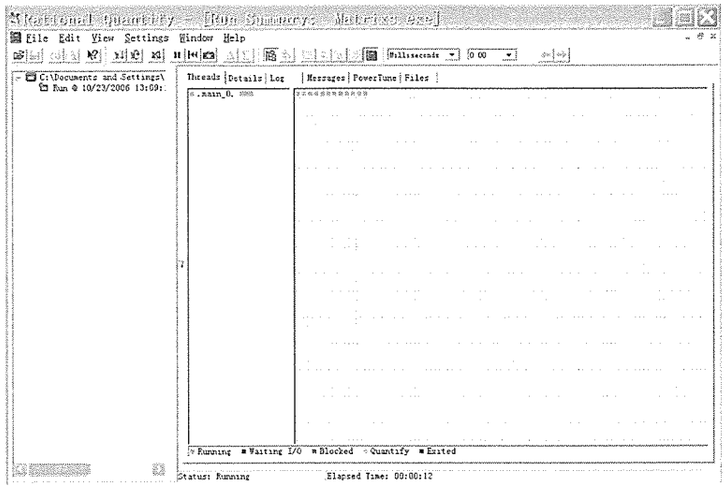


图 11-5　Rational Quantify 运行状态

其中显示 Matrixs.exe 中各个线程的运行状态（现在仅有主线程在运行）。不同的颜色代表了不同的运行状态，例如，绿色代表正在运行。

（3）性能结果数据分析。程序退出后，一个函数的调用关系图会被弹出如图 11-6 所示，其他粗线条代表了执行时间最长的路径（关键路径）。

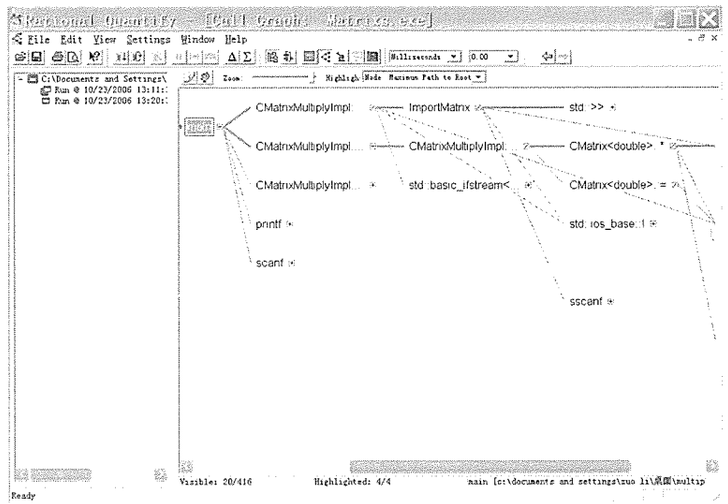


图 11-6　Matrix.exe 的调用关系图

根据调用关系图，我们可以大致确定程序的哪个分支占用了较多的时间。接下来我们进一步细化，选择工具栏中的 Function List 选项，程序执行过程中所有函数（指定模块中的）的详细性能指标显示在窗口中，如图 11-7 所示。

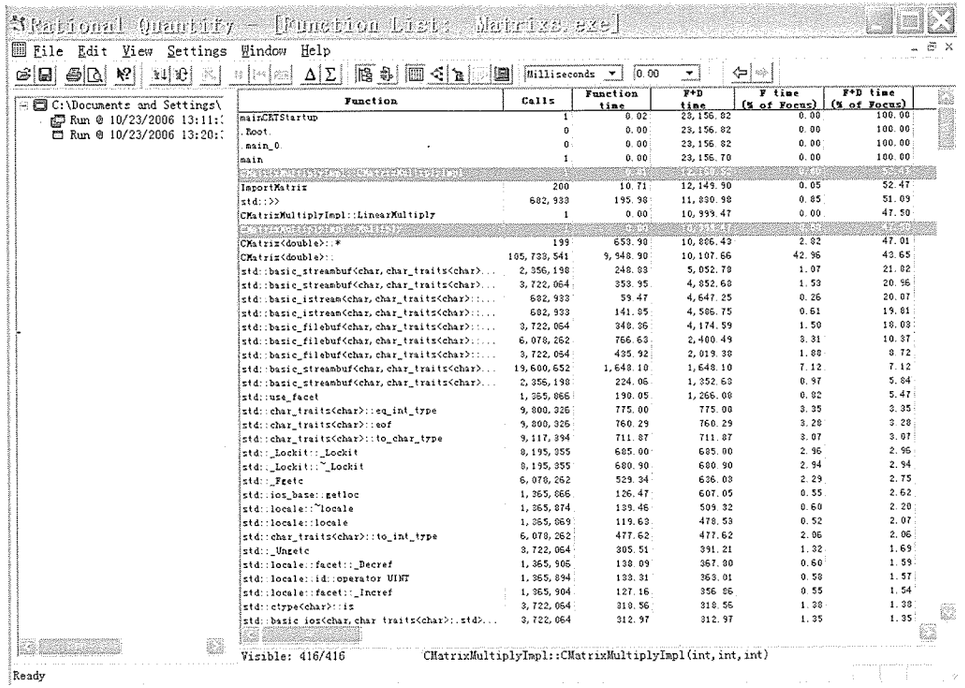


图 11-7　函数详细性能分析指标

在图中，我们发现“乘法矩阵链”程序的初始化函数（即 CMatrixMultiplyImpl 的构造函数）和乘法函数 CMatrixMultiplyImpl::Multiply 占据了程序绝大部分的执行时间。从“F+D time(% of Focus)”（即 Function Time + Descendents Function Time，函数及其子代函数执行时间）可以看出，这两个函数分别占用了总时间的 52.47%和 47.50%。

然而，只得到零散的函数性能数据不足以帮助我们找到真正的瓶颈所在。双击某个函数记录，我们可以看到针对该函数的更为详细的性能分析结果，如图 11-8 所示。

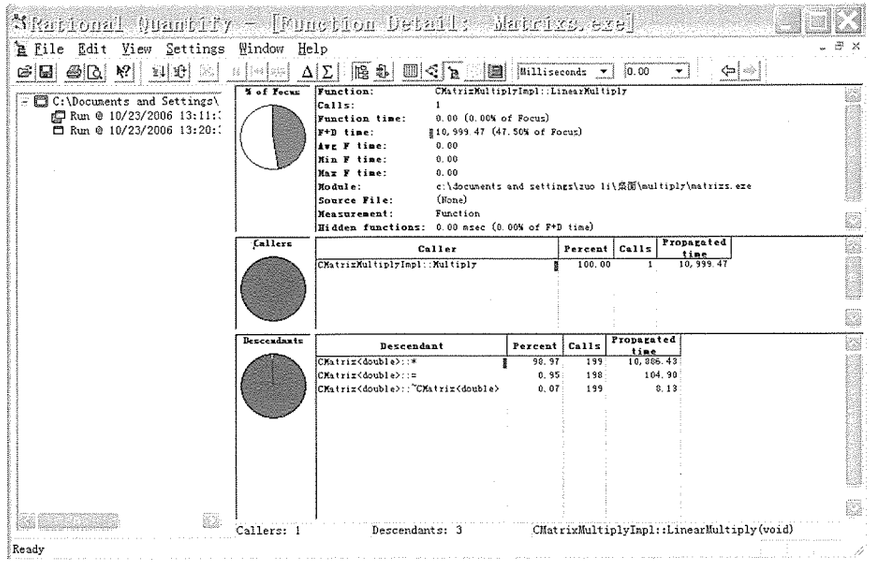


图 11-8　更为详细的性能分析结果

以 CMatrixMultiplyImpl::Multiply 为例，根据图 11-8，我们可以发现该函数全部的执行时间都花费在其调用函数 LinearMultiply 上。LinearMultiply 采用了线性相乘的方法，对整个矩阵链按从头到尾的顺序将每个矩阵相乘。其实，这种算法本身存在巨大的性能问题：矩阵乘法适合结合律，无论对一个矩阵链如何加括号都可以产生相同的结果，但是运算代价有可能相差悬殊，线性矩阵连乘常常可能无法达到性能最优。

这样我们轻松地找到了程序的运算瓶颈，对于另一个重要的性能瓶颈——I/O 操作（即造成矩阵链初始化时间较长的原因），也可以利用同样的方法找到它。限于篇幅，我们不再赘述。

根据分析结果，我们开始对矩阵链的连乘算法进行优化。根据动态程序设计思想（具体算法可以参考《实用算法的分析与程序设计》一书），我们采用“自底向上”的方式找到矩阵链最优的括号设置。随后根据最优化的结果，迭代地实现矩阵链乘法。我们再次利用 Rational Quantify 对优化后的程序进行分析，结果如图 11-9 所示。

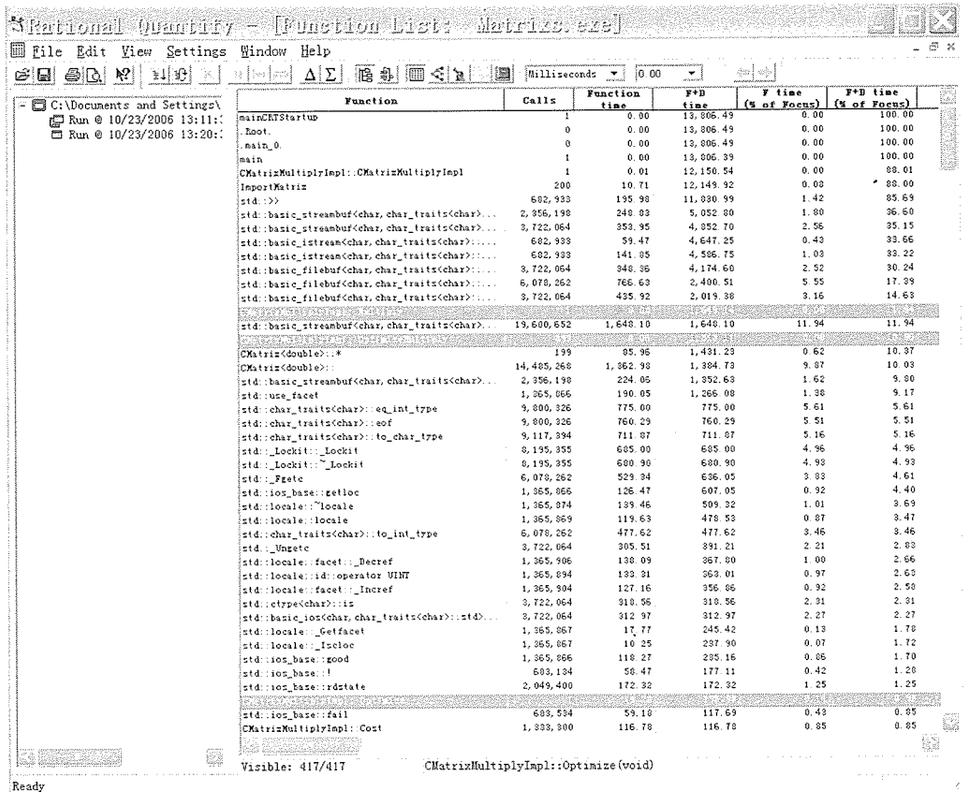


图 11-9　优化后的 Matrixs.exe 测试结果

再次利用 Rational Quantify 测试后，我们发现使用矩阵链优化算法（Optimize 和 OptimizeMultiply）的 CMatrixMultiplyImpl::Multiply 函数执行时间大大减少，在程序中占的比重由 47.5%减为 11.94%（没有对矩阵链初始化函数进行优化，可以利用其时间作为参考，与新的乘法函数进行对比），如图 11-10 所示。

选择工具栏中的 Compare Runs 选项可以对比优化前后的 Rational Quantify 结果。对于同一组数据，Multiply 的执行时间从 10999.47 ms 下降到 1649.14 ms。

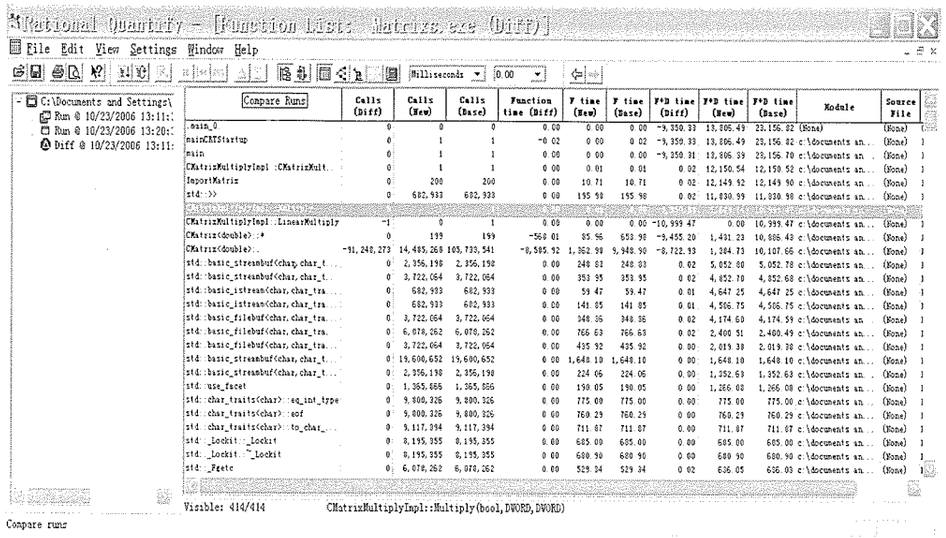


图 11-10　优化前后的测试结果对比

11.4　本章小结

本章介绍了 IBM Rational Quantify 的主要功能和工作原理，并进一步结合“乘法矩阵链”的例子为读者详细描述了如何利用 Rational Quantify 找到程序的性能瓶颈。现在我们已经知道，Rational Quantify 是一个面向 Visual C++、Visual Basic 或者 Java 开发环境的程序性能瓶颈测试工具。它不仅可以自动检测影响程序执行效率的性能瓶颈并提供参数分析，还可以在不同粒度（代码行级和函数级）上测定性能。使软件开发人员能够逐步定位，直至找到影响程序性能的“罪魁祸首”。相信 Rational Quantify 能成为读者攻坚程序性能难题的一把利器。

第 12 章　实时 IO 监测工具 FileMon

FileMon 是一个实时的 Windows 文件系统监视工具，它记录并显示全部所有与文件相关的操作（如读取、修改及出错信息等）。使用者可以存储所看到的结果，以便以后利用。它还提供过滤功能，通过设定条件使其显示的信息更符合用户的需求。FileMon 是一个免费的软件，可以从 www.sysinternals.com/Utilities/Filemon.html，或其他网站自由下载。尽管微软已经收购 Winternals Software 公司，但是该工具仍然可以免费下载（www.sysinternals.com/Blog/）。

12.1 FileMon 的工作原理

FileMon 的实现与操作系统的 I/O 子系统相关。由于 Windows 9X 与 Windows NT 及其后续版本的 I/O 子系统的实现方式不同，因此 FileMon 的 Windows 9X 与 Windows NT 版本的实现原理也不相同。本节以 Windows 2000 的 I/O 子系统为例，说明 FileMon 的工作原理。

Windows 2000 的 I/O 子系统如图 12-1 所示，它由多个可执行模块和设备驱动程序组成，其中 I/O 管理器和设备驱动程序是完成 I/O 操作最为重要的组件。I/O 管理器定义了处理 I/O 请求的处理框架，一个用户的 I/O 请求经过 I/O 管理器的处理被导向 I/O 管理器所管理的设备。一个设备驱动程序通常为某种特定类型的设备提供 I/O 接口，它接受 I/O 管理器发来的命令。由于设备驱动程序提供统一且模块化的接口，使得 I/O 管理器不需要知道驱动程序内部的细节即可调用。

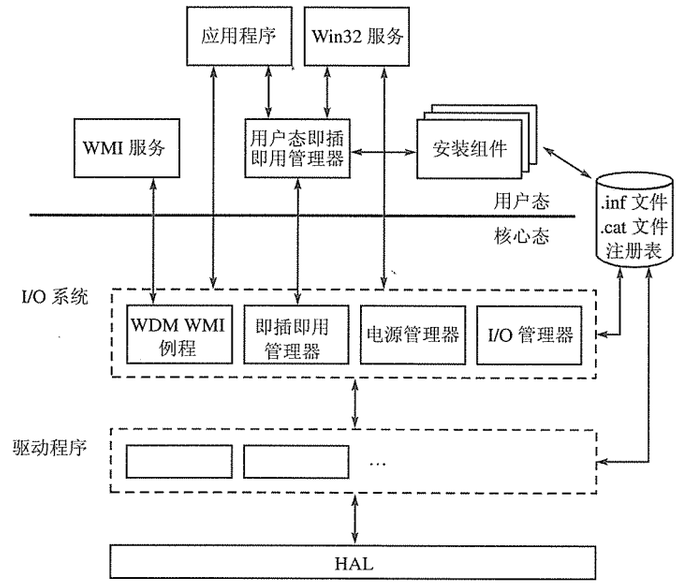


图 12-1 Windows 2000 的 I/O 子系统

I/O 管理器和设备驱动程序之间通过传递 I/O 请求包（IRP）来推动完成 I/O 请求。IRP

是一种数据结构，它描述了一个完整的 I/O 请求信息。I/O 管理器创建 IRP，并把该 IRP 指针传递给正确的驱动程序。驱动程序接收 IRP，执行 IRP 指定的操作，并在操作完成后将 IRP 传回给 I/O 管理器。I/O 管理器接到从驱动程序返回的 IRP，或者结束一个 I/O 操作并释放 IRP，或者把该 IRP 传递给另一个驱动程序做进一步的处理。图 12-2 以访问磁盘中的文件为例说明驱动程序是如何工作的。首先文件系统驱动程序从 I/O 管理器处接受对某个文件某个位置的写数据请求，并将该请求转化为对磁盘某个特定的逻辑位置写一定数量字节的请求，然后它将该请求通过 I/O 管理器传递给磁盘驱动程序。最后，磁盘驱动程序将该请求转换为磁盘上的物理位置（柱面、磁道和扇区）并操作磁头完成写数据操作。

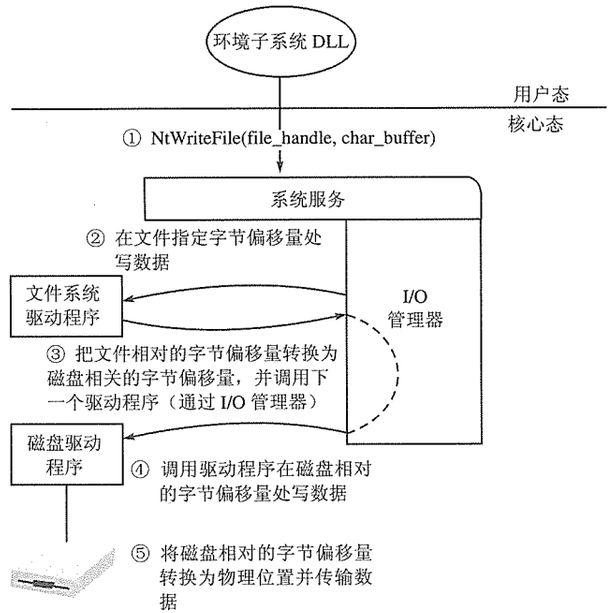


图 12-2 驱动程序的分层工作

这种方式可以实现驱动程序之间任务的划分及分层，同时也很容易将另外的驱动程序插入到驱动程序的层次中。FileMon 通过增加一个驱动程序来监控文件系统的 I/O，如图 12-3 所示。

FileMon 的核心是一个驱动程序，它可以创建过滤器设备对象（filter device object），并将其与目标文件系统设备对象（file system device object）关联，从而使得 FileMon 能够得到所有访问文件系统驱动的 IRP 和快速 I/O（Fast I/O）请求。

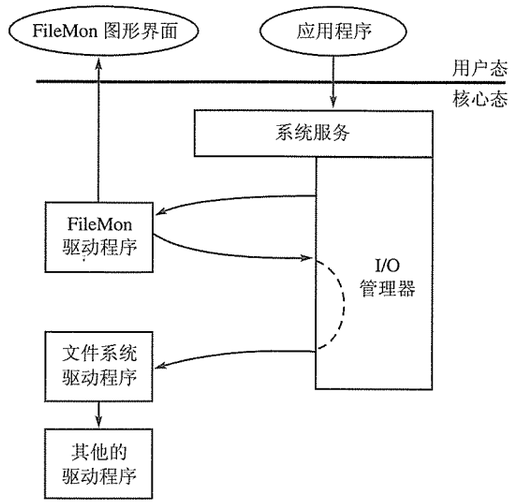


图 12-3　FileMon 的工作原理

12.2　FileMon 使用指南

FileMon 不仅提供强大的文件监视功能，而且使用非常方便。图 12-4 所示为 FileMon 的一个屏幕截图。

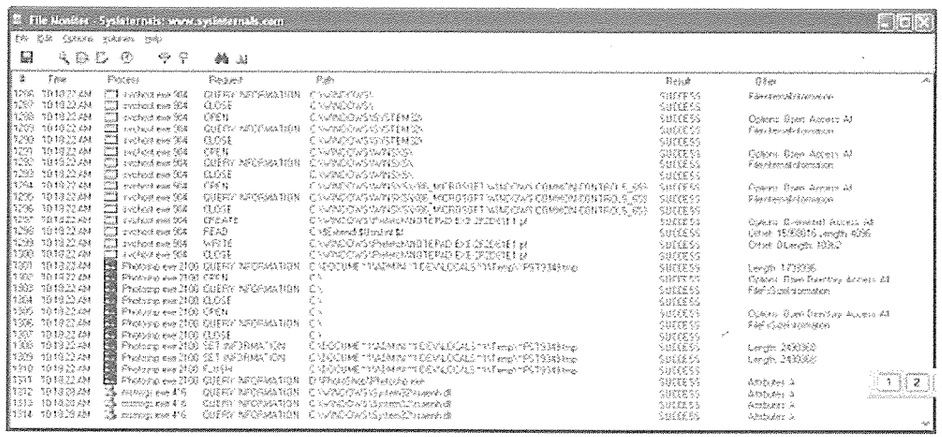


图 12-4　FileMon 的一个屏幕截图

（1）基本功能。

使用 FileMon 来实时监控文件系统的活动，用户必须要有管理员权限。FileMon 启动后就会立即监控所有本地磁盘的活动，可以通过菜单 Volumes 提供的选项来选择所需要监控的磁盘。用户可以使用菜单、工具栏或热键提供的功能来暂停监视、清空窗口显示的内容、禁止自动滚屏，以及存储监视结果等。

（2）设置过滤器。

由于 FileMon 会显示当前文件系统的所有活动，用户可以通过设置过滤器来选择输出感兴趣的内容。通过选择 Options–>Filter–>Highlight 选项或单击工具栏中的 Filter 按钮来打开如图 12-5 所示的 Filemon Filter 对话框。

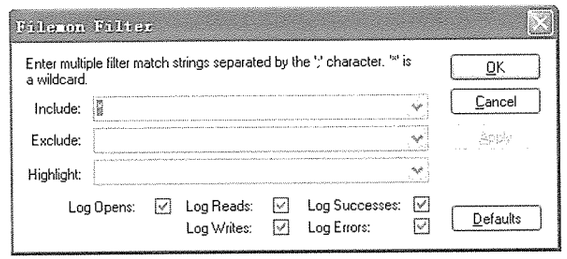


图 12-5　Filemon Filter 对话框

用户可以在Include文本框中输入需要包含的字符串，需要过滤的字符串可以在Exclude文本框中输入。若需要包含多个字符串，则用；隔开（例如，myApp*;*temp*）。这些字符串之间的关系是或的关系。在输入的字符串中可以用*字符来代表任意字符串，并且过滤器对这些字符串是大小写不敏感的。例如，在 Include 文本框中指定“C:\WINDOWS”，在 Exclude 文本框中指定“C:\WINDOWS\system32”，FileMon 将会显示任何访问“C:\WINDOWS”下的文件和目录的活动，但不包括访问“C:\WINDOWS\ system32”下的文件和目录的活动。

如果设定过滤器的条件，以后每次 FileMon 启动时都会弹出 Filemon Filter 对话框要求确认当前设置或重新设置。启动FileMon时可以在命令行中加上/q参数来阻止Filemon Filter对话框在 FileMon 启动时弹出。

提示：

如果设置了过滤器条件，并且启动 FileMon 时并没有重新设置当前的条件，有些情况下发现 FileMon 并没有记录相应的 I/O 活动。此时，只需要首先把过滤器条件设置成默认值，然后再重新设置为需要的配置，FileMon 即可正常工作。

提示：

FileMon 一般能正常运行，但是某些情况下，比如修改了显示的字体，发现 FileMon 不能正常显示所捕捉的 I/O 活动。此时，或者只有几行结果迅速翻滚，或者也不显示任何内容。为此只要删除注册表中相应的项（HKEY_CURRENT_USER\Software \Sysinternals\FileMon），重新启动 FileMon 即可。

（3）高级输出模式。

Filemon 的默认输出模式会自动过滤一些活动，并且对所有的 I/O 操作给出比较简单的类别。FASTIO_CHECK_IF_POSSIBLE 操作被过滤，失败的 FASTIO_READ 操作也被过滤，而成功的 FASTIO_READ 操作会被记录为“READ”。另外，默认输出模式也不会记录系统进程对文件系统的操作，例如内存管理系统因为换页而执行的文件操作。通过选择 Options–>Advanced output 选项把输出模式改为高级模式。

（4）输出内容。

- # 列：表示顺序号。
- Time 列：表示时间，可以是时钟的时间，表示访问该文件的时间。也可以是操作该文件的持续时间，通过工具栏上的 Duration/Clock 按钮来切换。
- Process 列：代表进程名和进程号。
- Request 列：代表文件操作的种类，通过选中选择 Options–>Advanced output 选项可以得到更为详细的 I/O 操作类型

如果输出模式为默认输出模式，Request 列显示的值是一些比较容易理解的值，如 OPEN、CLOSE、READ、WRITE 及 QUERY_INFORMATION 等。

如果输出模式为 Advanced 模式，Request 列显示更具体的一些 I/O 操作，如磁盘驱动程序的主功能代码 IRP_MJ_CREATE、IRP_MJ_CLOSE、IRP_MJ_READ、IRP_MJ_WRITE 及 IRP_MJ_QUERY_INFORMATION 等，或者快速 I/O 功能代码 FASTIO_LOCK、

FASTIO_READ、FASTIO_WRITE 及 FASTIO_QUERY_BASIC_INFO 等，这些功能代码都代表了磁盘驱动程序的某个例程。

- Path 列：代表进程所访问文件的路径。
- Result 列：表示进程所访问文件的结果。
- Other 列：列出一些补充信息。

12.3 使用 FileMon 解决问题

（1）作为问题解决工具。

当应用程序遇到文件访问错误时，一般不能为最终用户提供十分明确的信息。这类问题包括无权访问文件、需要使用的文件不存在，以及需要使用的文件被其他进程使用等。这时，应用程序很可能给出一些无用的信息，然后停止或者退出。

FileMon 可以帮助用户找出问题的原因。首先设置过滤器，使得 FileMon 只记录感兴趣的 I/O 活动。然后从 FileMon 记录的结果中从后往前分析那些不成功的 I/O 活动。当然，如果某个应用程序在一个环境下可以正常运行，在另一个环境下不能正常运行，则需要分别对这两种情况记录 I/O 活动，并对比分析结果，以帮助发现问题。

下面是的例子演示了如何使用 FileMon 来解决与文件访问相关的问题，使用 make 作为 build 工具来 build 某个工程时，碰到如图 12-6 所示的错误：

```
link /COMMENT:"sax_645_____"  /MACHINE:IX86 @C:\DOCUME~1\wlei\LOCALS~1\Temp\m
8
Microsoft (R) Incremental Linker Version 6.00.8447
Copyright (C) Microsoft Corp 1992-1998. All rights reserved.

/MAP /NODEFAULTLIB /OPT:NOREF /RELEASE /DEBUG:notmapped,full /DEBUG:notmapped,
ll /DEBUGTYPE:cv /SUBSYSTEM:CONSOLE /DLL -out:..\wntmsci9.pro\bin\sax.uno.dll
LINK : fatal error LNK1212: error opening program database; file is in use
dmake:  Error code 188, while making '..\wntmsci9.pro\bin\sax.uno.dll'
```

图 12-6 链接时发生的错误

重新执行 build 指令，并同时打开 FileMon 来观察 Link.exe 的 I/O 活动，得到如图 12-7 所示的信息：

LINK.EXE:984	IRP_MJ_CLOSE	\slb\expatwrap.lib	SUCCESS	
LINK.EXE:984	IRP_MJ_CREATE	\bin\sax.uno.pdb	SHARING VIOLATION	Options: Open Access
LINK.EXE:984	IRP_MJ_CREATE	\bin\sax.uno.pdb	SUCCESS	Options: Open Access
LINK.EXE:984	IRP_MJ_QUERY_VOL...	\bin\sax.uno.pdb	BUFFER OVERFLOW	FileFsVolumeInformati...
LINK.EXE:984	IRP_MJ_QUERY_INF...	\bin\sax.uno.pdb	SUCCESS	FileInternalInformation
LINK.EXE:984	FASTIO_QUERY_STA...	\bin\sax.uno.pdb	SUCCESS	Length: 656384
LINK.EXE:984	IRP_MJ_CLEANUP	\bin\sax.uno.pdb	SUCCESS	
LINK.EXE:984	IRP_MJ_CLOSE	\bin\sax.uno.pdb	SUCCESS	
LINK.EXE:984	IRP_MJ_CREATE	\bin\sax.uno.pdb	SHARING VIOLATION	Options: OpenIf Acces
LINK.EXE:984	IRP_MJ_CREATE	\bin\sax.uno.pdb	SUCCESS	Options: Open Access

图 12-7　Filemon 监测的结果

发现当 Link.exe 使用 sax.uno.pdb 时，返回 SHARING VIOLATION 结果。然后使用 Process Explorer（另外一个工具）得知另外一个进程正在使用这个文件，关闭该进程即可解决 Link 错误的问题。

（2）作为性能分析工具。

磁盘作为存储信息的设备能够存储大量的数据，但是从磁盘中读取数据所花费的时间比从内存中读取数据所花费的时间要多很多。一般来说，读取 512 位数据，磁盘大约需要花费 10 ms，而 SRAM 需要花费 256 ns，DRAM 需要花费 4 000 ns。磁盘花费的时间是 SRAM 的 40 000 倍，是 DRAM 的 2 500 倍。因此从磁盘中访问文件的时间常常决定了应用程序的性能，特别是应用程序的启动性能。

应用程序的启动过程中不仅会把启动相关的代码从磁盘读到内存，而且可能会从配置文件中读取一些配置信息。如果应用程序的启动过程需要访问大量的配置信息，则会降低启动性能。可以使用 FileMon 来记录应用程序启动过程中所访问的文件，以及访问每个文件所需要的时间。根据这些客观数据来找出启动过程的瓶颈，从而找出解决问题的方案。

以下以 OpenOffice.org1.1.1 为例来说明用 FileMon 来寻找 OOo.org1.1.1 的启动瓶颈。（OpenOffice.org 是开放源代码的办公软件，访问 www.openoffice.org 网站可以获得更多详细信息）。

首先启动 FileMon。如图 12-8 所示。在 Filemon Filter 对话框中设置 FileMon 的过滤器为只记录包含 soffice.exe 的活动，并且排除所有包含 explorer.exe 的活动（如果通过资源管理器来启动 OpenOffice.org），然后通过工具栏中的 Duration/Clock 按钮把监视结果的 Time 列设置为活动的持续时间。

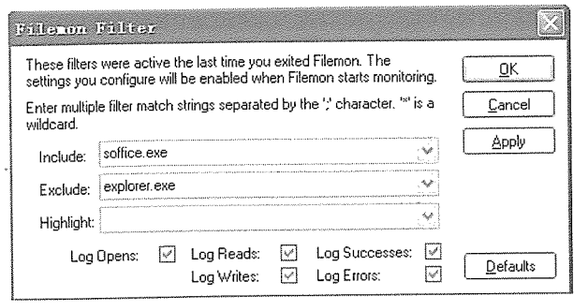


图 12-8　设置 FileMon 的过滤器

接着，从资源管理器中启动 OpenOffice.org。启动完成后，关闭 OpenOffice.org 并保存 FileMon 中记录的数据到 OOo.log 中。

最后，把 OOo.log 导入到 Excel、OOo.SpreadSheet 或者类似的工具中，分类处理数据，得到部分结果如图 12-9 所示：

time	Path
0.90190006	C:\Program Files\OpenOffice.org1.1.1\program\i18npool645mi.dll 汇总
0.30133694	C:\Program Files\OpenOffice.org1.1.1\program\vcl645mi.dll 汇总
0.29334116	C:\Program Files\OpenOffice.org1.1.1\program\services.rdb 汇总
0.24307152	C:\Program Files\OpenOffice.org1.1.1\program\MSVCP70.dll 汇总
0.23350136	C:\Program Files\OpenOffice.org1.1.1\program\svx645mi.dll 汇总
0.22661697	C:\Program Files\OpenOffice.org1.1.1\program\svt645mi.dll 汇总
0.21125591	C:\Program Files\OpenOffice.org1.1.1\program\svl645mi.dll 汇总
0.20552451	C:\Program Files\OpenOffice.org1.1.1\program\sfx645mi.dll 汇总
0.20547051	C:\Program Files\OpenOffice.org1.1.1\program\cfgmgr2.dll 汇总
0.2013491	C:\Program Files\OpenOffice.org1.1.1\program\dl645mi.dll 汇总
0.1709945	C:\Program Files\OpenOffice.org1.1.1\program\dnd.dll 汇总
0.13074542	C:\Program Files\OpenOffice.org1.1.1\program\tk645mi.dll 汇总
0.12335724	C:\Program Files\OpenOffice.org1.1.1\program\MSVCR70.dll 汇总
0.12182823	C:\Program Files\OpenOffice.org1.1.1\program\tl645mi.dll 汇总
0.11414021	C:\Program Files\OpenOffice.org1.1.1\program\comphelp3MSC.dll 汇总
0.11365254	C:\Program Files\OpenOffice.org1.1.1\program\types.rdb 汇总
0.10993086	C:\Program Files\OpenOffice.org1.1.1\program\fwe645mi.dll 汇总
0.10598521	C:\Program Files\OpenOffice.org1.1.1\program\so645mi.dll 汇总
0.10579524	C:\Program Files\OpenOffice.org1.1.1\program\icuuc22.dll 汇总
0.10448948	C:\Program Files\OpenOffice.org1.1.1\program\sal3.dll 汇总

图 12-9　OpenOffice.org 启动时 I/O 占时最多的 20 个文件

如果把所有的访问安装在 OOo 目录下 DLL 的时间累加，得到的时间占所有 I/O 时间的 70%。

在最耗 I/O 时间的前 20 个文件中，有两个类似于配置文件的 rdb 文件。

因此，可能的优化方案如下。

（1）减少访问 DLL 的 I/O 时间，可以采用 9.2.3 节介绍的优化代码布局的方法来减少启动过程中访问 DLL 中代码的数量，从而减少相关的 I/O。

（2）修改访问 services.rdb 和 types.rdb 的算法，减少启动过程中从这两个文件中读取数据的数量，从而减少相关的 I/O。

提示：

FileMon 可以给出的 I/O 的持续时间，但该数据只具有相对的参考价值。考虑到误差和四舍五入的原因，把所有 I/O 时间相加得到的时间可能远远大于程序的启动时间。但是，从 FileMon 记录的数据中可以得到某个文件的访问时间占整个 I/O 时间的比例，这一数据还是对性能分析具有一定的指导意义。

12.4 本章小结

本章介绍了 Windows 系统下的一个实时的 I/O 监测工具 FileMon，首先介绍了该工具的工作原理。然后详细地描述了该工具的使用方法，最后通过一些小例子来介绍如何使用 FileMon 来解决问题。尽管 FileMon 是一个小巧的工具，但是它可以提供一些非常有用的功能。因此，FileMon 不仅仅可以被用来解决一些与 I/O 相关的问题，它所产生的数据还可以用来分析应用程序的启动性能。

参 考 文 献

[1] IBM. AIX Version 3.2 for RISC System/6000 Optimizatoin and Tuning Guide for Fortran, C, and C++. IBM. 1993.

[2] R. Alexander, G. Bensley. C++ Footprint and Performance Optimization. Sams Publishing. 2000.

[3] Dov Bulka, David Mayhew. Efficient C++ Performance Programming Techniques. Addison Wesley. 1999.

[4] John Robbins. Improving Runtime Performance with the Smooth Working Set Tool. MSDN Magazine. 2000.

[5] Matt Pietrek. Inside Windows: An In-Depth Look into the Win32 Portable Executable File Format. MSDN Magazine. 2002.

[6] Matt Pietrek. Optimizing DLL Load Time Performance. MSDN Magazine. 2000.

[7] Edward G. Bradford. 运行时：Linux 和 Windows 2000 上的高性能编程技术-建立计时例程. IBM DeveloperWorks, 2001.

[8] Russ Osterlund. What Goes On Inside Windows 2000: Solving the Mysteries of the Loader. MSDN Magazine, 2002.

[9] Alex Farber. Using the Rebase utility in project makefile. website:The Code Project, 2001.

[10] Neelakanth Nadgir. Reducing Application Startup Time. Sun Developer Network Site, 2002.

[11] David A. Solomon. Mark E. Russinovich. Inside Microsoft Windows 2000. Microsoft Press, 2000.

[12] Jeffrey Richter. Programming Applications for Microsoft Windows. Fourth Edition. Microsoft Press, 1999.

[13] Andrew S. Tanenbrum. Modern Operation System. Second Edition. Prentice Hall, 2001.

[14] Abraham Silberschatz, Peter Baer Galvin, Greg Gagne. Operation System Concepts. Sixth Edition, 2002.

[15] Randy Kath. The Virtual-Memory Manager in Windows NT. MSDN, 1992.

[16] Randy Kath. Managing Virtual Memory in Win32. MSDN, 1993.