写作背景

在经历过几个大型的，失败的项目之后，我终于明白没有什么比高素质的程序员更能决定项目的成功了，无论什么过程，什么编程语言和开发工具，离开了高 素质的程序员，什么都是白费力气。毫无疑问，人是软件开发中最重要的因素，但不是每个人都重要，不是什么样的人都重要，只有那些高素质的程序员和那些对项 目有突出贡献的人才是重要的。

不过高素质的程序员并不多见，所以从我开始带人之际，就在思考团队成员培养的问题。我做过很多尝试，从小组内学习到整个部门一起上大课，最后又回到 对个人单独的辅导；从通过 codreview 做现场教育到制定过一个宏伟的培训计划，最后又回到一个朴素的培训过程。其中遇到了很多问题，开始是培训不够 系统，效果不甚理想，后来又因为计划过于”宏伟”而无法实施，直到最后行成一个朴素的，切实可行的培训方案，中间经过了好几年时间，直到去年，整个计划才 趋于完善。我把这个培训计划称为系统程序员成长计划，这就是我在这个系列中要写的。

培训内容不是来源于某本书，毕业八年来，我坚持不断的看书，家中放了 7 大储物箱，有 300 多本不同类型的书籍，其中囊括了大部分经典的 IT 图书。当 然也不是全部内容都来源书本，这几年我在开源软件吸取了大量的营养，一些思想和经验在 broncho 项目中也有充分的发挥，可以说是理论，经验和实践的结 合。但我不尝试阐述什么高深的道理，相反我是针对应届毕业生和业余爱好者写的，目的是要让初学者进阶为一个专业的程序员。

为什么叫 系统程序员成长计划？程序员的范围太广了, 虽然软件开发有很多相似之处，但是隔行如隔山，比如对于目前炙手可热的 WEB 开发，我完全是外行。想什么都讲一点，结果是什么都没有讲清楚，所以我得把培 训计划限定在我熟悉的范围之内。先定义一下系统程序员：从事操作系统内核、DBMS、GUI 系统，基础函数库，应用程序框架，编译器和虚拟机等基础软件开 发的程序员。这些培训同样适用于桌面软件和智能手机软件开发，我想对其它软件开发也会有一些启发作用。

草莓酱定律与果酱定律。第一次在咨询的奥秘中看到草莓酱定律时，我觉得非常有意思。当然这个系列也无法脱离草莓酱定律的魔法，利用这个系列中的内 容，我手把手的教了十多个同事，收到了良好的效果。当有数百个读者读这些文章时，我不敢期望有同样的效果。不过在果酱定律的鼓励下，我相信这个系列中至少 有部分内容的价值不会因为读者群的增大而消失，所以最终决定写出来。

中心思想

软件开发的困难在哪里？对于这个问题，不同的人有不同的答案，同一个人在不同职业阶段也会有不同的答案。作为一个系统程序员来说，我认为软件开发有两大难点：

一是控制软件的复杂度。软件的复杂度越来越高，而人类的智力基本保持不变，如何以有限的智力去控制无限膨胀的复杂度？我经历过几个大型项目，也分析 过不少现有的开源软件，我得出一个结论：没有单个难题和技术细节是我们无法搞定的，而所有这些问题出现在一个项目中时，其呈指数增长的复杂度往往让我们束 手无策。

二是隔离变化。用户需求在变化，应用环境在变化，新技术不断涌现，所有这些都要求软件开发能够射中移动的目标。即使是开发基础平台软件，在超过几年 时间的开发周期之后，

需求的变化也是相当惊人的。需求变化并不可怕，关键在于变化对系统的影响，如果牵一发而动全身，一点小小的变化可能对系统造成致命的 影响。

为了解决这两个问题，方法学家们几十年来不断努力，他们发明或改进软件的开发过程和设计方法。系统程序员面对的基础软件通常都是大型的复杂的软件，其通用性也要求能容纳更多变化，解决这两个问题也是系统程序员成长计划的主要目标。

文章特色

以引导读者思考为主。培训可以制造合格的程序员，却无法造就一流的高手。培训是一个被动的过程，我们都知道在大学里听课的效果，我不希望本系列文章 的成为单纯的培训教材，相反我们要变被动为主动，最大限度的提高学习的效果。大多数情况下，我先提出问题让读者去思考，让读者自己尝试去解决，能不能解决 这个问题不重要，重要的是在思考中提升自己。如果读者在一定时间无法找到解决问题的方法，后面会有专业程序员的参考做法(或许不是最优的)。

以简单的例子讲述复杂的设计方法。我曾经制定一个宏伟的培训计划，结果失败了，原因是我忘记了我在走路之前也艰难的爬行过。这次我吸取了教训，用简 单的示例讲述复杂的设计方法，而且不对读者的背景太多假设。里面不会出现复杂的数据结构和算法，也不会引入大型软件来唬人。即有足够的挑战，不会让读者感 到乏味；又一切尽在掌握之中，不会因为挫折而打击积极性。

技术能力与工作态度并重。古人说德才兼备者才是君子。同样，做一流的程序员，也要德才兼修才行。当我手把手的教别人的时候，我不但希望他能学会我讲 的知识点，更希望他能学习我的工作态度和作为程序员的道德素养。当然有些东西只可意会不可言传，未必能用文字表达出来，不管怎样这是我的初衷之一。

读者群

这个系列完全是针对初学者写的，初学者包括在校学生、应届毕业生和其他业余爱好者。我面试过很多应届毕业生，他们大多数都不具备编程能力，唯一的优 势就是对基本理论有一知半解的了解。本系列文章就是为他们量身定制的，经历十几个人的实践，取得令人满意的效果，大多数人在开始一行代码都写不出，到培训 结束时一般都能独立开发/维护一些几千行的小模块。本系列文章并不是金钥匙，最终学习的效果与个人的悟性和努力密切相关，但不管怎样，只要读完这些文章， 你都会有不小收获的。

如何使用

温伯格说过，医生的药方包括药物和服药的方法，两者缺一不可。同样的教材，不同的学习方法，效果也有很大差别。对于本系列文章的服用方法，我的建议 是，对于文中提出的问题，先自己想办法去解决，可以查资料，至少经过两三个小时的思考之后，再继续阅读，最后再按学到的方法独立写一遍，学习编程一定要多 写多练，否则效果会大打折扣。

Enjoy it!

对于是否写这样一章，我犹豫了很久，最后考虑到这个系列是针对新手而写的，不应该对读者做过多假设，这些基础知识是必须掌握的，不能不介绍一下。如果你已经了解它们，可以

放心的跳过本章。如果你是新手，请认眞学习本章提到的内容。

基础知识

C 语言。千万不要认为 C 语言过时了，它始终是开源社区，特别是系统软件和嵌入式系统中的王者，在可以预见的未来，C 语言将持续焕 发出生命力。有些外行认为 C 语言不适合开发大型软件，这是大错特错了，操作系统内核，虚拟机，数据库管理系统，图形引擎和 WEB 服务器等大型软件几乎都是 用 C 语言开发的。相反 C 语言不适合开发小程序，这时候脚本语言更能显出威力。C 语言能经久不衰，自有它的道理：

C 语言是最简单的语言之一，大部分编程语言在出现时都以其简单而获得好评，几乎全部都随着时间的推移变得越来越复杂，C 语言经过数十年的发展，却始 终保持其简洁和优美。初学者认为 C 语言难学，其实主要是对计算机本身不理解，花点时间去学习一下计算机组成原理和操作系统原理，再来学习 C 语言就很简单 了。一旦掌握了它，你会发现 C 语言的每项特性都是必须，常用的，根本不需要记忆任何不必要的东西，它的特性眞是减无可减了。

C 语言是运行时效率最高的编程语言之一。同样的算法，C 语言通常比其它语言更高效，这也它作为系统软件主流编程语言的原因之一。有些动态语言号称比较 C 语言更快，那都是骗人的，拿一个特定算法作为例子不足为证。选择是高效的算法是根本，但 C 语言更能把高效发挥到极致。

C 语言是最直观的语言之一。C 语言能够直观的表达程序员的想法，不像其它一些语言，一行简单的代码，你不清楚里面到底做了什么，不清楚它将花多少时 间执行。C 语言的直观性很好的满足了程序员好奇心，使用 C 语言你更能感觉编程是一种艺术。一切尽在掌握之中，更能满足你的成就感。

在系统程序员炼成计划中，前面部分都是使用 C 语言作为示例，读者应该找本 C 语言入门书籍看看，可以先通读一遍，不求甚解都可以，随着后面的课程而深入的学习。

数据结构与算法。不管使用什么设计方法和开发过程，数据结构与算法都是软件开发的基石。打好基础在以后的工作中会事半功倍。后继 课程也都是这些基本数据结构和算法为中心，讲述如何用这些基本的材料构建大型系统。读者暂时无需精通数据结构和算法，先找本书看看，了解一下像双向链表、 动态数组、队列、堆、栈、hash 表、排序和查找的基本原理就行了，后面我们会以这些数据结构题材反复的练习。

开发环境

本系列文章重点讲解软件开发的基础知识，这些知识不依赖于特定的平台和开发环境，读者可以根据自己喜好来选择，我们推荐读者使用下列开发环境：

操作系统使 Linux。Linux 是最适合程序员使用的操作系统，它是开源的，有多种不同的发行版可以免费使用，这些发行版默认安装就带了开发工具。学习 Linux 本身就需要一本书，如果你从来没接触过 Linux，也不用惊慌，花几个小时学会十来个常用的命令就够了，其它的以后慢慢再学。

编辑器使用 VIM。编辑器的功能是创建源文件，也就是把我们编写的代码输入到电脑中。vim 和 emacs 是 Linux 下最流行的 代码编辑器，vim 入门更简单，功能也很强大。它支持查找剪切替换等基本编辑功能，也支持符号跳转和代码补全等高级编辑特性。vimtutor 是最好

的入 门教材，初学者跟着这个 tutor 学习一遍就可以用它来编程了，等用得比较熟练之后，再去掌握那些高级功能。你掌握得越熟练，你就能更高效的工作，这个投 资是值得的。

编译器使用 gcc。编译器的功能是把源代码翻译成计算机可以“读懂”的机器语言。在 Linux 下可用的 C 编译器有好几个，gcc 是其中最流行的，大多数发行版都默认安装了 gcc。gcc 的参数很多，看起来很复杂，我们只掌握最简单的用法就好了，大概像这样的：gcc -g test.c -o test。

调试器使用 gdb。调试器的功能是帮助程序员定位错误，这是最后一招，也是最不期望的一招，使用调试器越多通常说明你的水平越 差，不过对初学者来说，掌握这个工具必不要可少的。gdb 的功能强大，推荐读者使用命令行的 gdb，它更灵活更方便。读者先掌握如何设置断点、显示变量和 继续执行等基本操作就行了。

工程管理使用 make。make 是 Linux 下最流行的工程管理工具，Makefile 是 make 的输入文件，它本身就相当于一 种编程语言，执行 make 相当于调用其中的函数。编写 Makefile 是一件繁琐无趣的工作，幸好我们不用学习它，后面我们会讲解 make 的改进版 automake，现在你能写出下面这种简单的 Makefile 就行了：

```
all:
    gcc -g test.c -o test
clean:
    rm -f test
```

在这里，你可以把 all 看作一个函数名，gcc -g test.c -o test 是函数体(前面加 tab)，它的功能是编译 test.c 成 test，在命令行运行 make all 就相当于调用这个函数。clean 是另外一个函数，它的功能是删除 test。如果你有时间学习一下 Makefile 当然更好，如果没有时间，了解这 么多也够了。

我在培训初学者时，如果他从来没用过 Linux，没有用 C 语言写过程序，我会给两到四周时间学习上述内容。如果读者处于类似的水平，也不急着看后面的课程，好好学习一下这里提到的内容。

需求简述

用 C 语言编写一个双向链表。如果你有一定的 C 语言编程经验，这自然是小菜一碟。有的读者可能连一个小程序都没有写过，那也不用害怕，可以参考任何一 本《数据结构》和 C 语言的书籍。先弄明白基本概念，把书上的代码看明白，再把代码抄到电脑里，保证编译过去，调试它到正常运行。反复这个过程，直到你能独 立完成它为止。写第一行代码是很痛苦的，我培训过好几个同事，他们不是计算机系毕业的，开始在电脑前坐一整天，一行代码都敲不出来，我最早写程序时的情况 也好不了多少，不过没有关系，迈出这一步就好了。

花 1-3 天时间，完成这个任务后，再继续往下阅读。

当 你读到这里的时候，相信你已经独立写出了一个双向链表。恭喜你！迈出这一步可是值得庆祝的，现在你已经走在通往程序员的光明大道上了。不过你还是个业余程 序员，那当然了，你才写出第一个程序呢！什么时候才能成为一个专业程序员呢？三年还是五年工作经验？其实不用的，你马上就可以了，我没有骗你，因为专业程 序员与业余程序员之分主要

在于一种态度，如果缺乏这种态度，拥有十年工作经验也还是业余的。

什么态度？专业态度！也就是星爷常说的专业精神。专业态度有多种表现形式，以后我们会一一介绍的。这里先介绍一下有关形象的态度，专业的程序员是很 注重自己的形象的，当然程序员的形象不是表现在衣着和言谈上，而是表现在代码风格上，代码就是程序员的社交工具，代码风格可是攸关形象的大事。

有人说过，傻瓜都可以写出机器能读懂的代码，但只有专业程序员才能写出人能读懂的代码。作为专业程序员，每当写下一行代码时，要记得程序首先是给人 读的，其次才是给机器读的。你要从一个业余程序员转向专业程序员，就要先从代码风格开始，并从此养成一种严谨的工作态度，生活上的不拘小节可不能带到编程 中来。

代码风格有很多种，Windows 和 Linux 都有自己主流的代码风格，每个团队，每个公司也可能有自己的代码风格，争论哪种风格好那种风格坏没有什么意义。只要有助于其他程序员理解的代码风格都是可以接受的，因为遵循特定代码风格的目的就是为了便于交流。

这里介绍一下作者本人喜欢的代码风格，这种代码风格也在作者所在团队中使用。这里的命名风格与 GTK+代码相近，排版风格 Linux 内核代码相近。

命名要展示对象的功能。

文件名：单词小写，多个单词用下划线分隔。
如: dlist.c (这里 d 代表 double，是通用的缩写方法)

注意: 文件名一定要能传达文件的内容信息，别人一看到文件名就是知道文件中放的是什么内容。只把一个类或者一类的代码放在一起是好的习惯，这样就很容易给文件取 一个直观的名字。业余爱好者常常把很多没关系的代码糅到一个文件中，结果造成代码杂乱无章，也很难给它取一个恰当的名字。

函数名：单词小写，多个单词用下划线分隔。
如：find_node

注意：同样，一个函数只完成单一功能，不要用代码的长度来衡量是不是要把一段代码独立成一个函数。即使只有几行代码，只要它完成的是一项独立的功 能，都应该提为一个单独的函数，而函数名可以直观的反应出它的功能。如果在给函数起名时遇到了困难，通常是函数设计不合理，应该仔细思考一下。

结构/枚举/联合名：首字母大写，多个单词连写。
如：struct _DListNode;

宏名：单词大写，多个单词下划线分隔
如：#define MAX_PATH 260

变量名：单词小写，多个单词下划线分隔。
如：DListNode* node = NULL;

面向对象的命名方式:

1.以对象为中心，采用主语(对象)+谓语(动作)，取代传统的谓语(动作)+宾语(目标)。
如：dlist_append

2.第一个参数为对象，并用 thiz 命名。
如：dlist_append(DList* thiz, void* value);

3.对象有自己的生命周期，都有 create 和 destroy 函数。

排版布局要美观大方。

合理使用空行：

1.函数体之间用空行分隔。

2.结构/联合/枚举声明空行分隔。

3.不同功能的代码块之间用空行分隔。

4.类似的代码放在一起，和其它部分用空行分隔。比如宏定义，类型定义，函数声明和全局变量放在一起。

5.使用空行时，一行就够了，不要使用连续多个空行，那样让人感觉空荡荡。

合理使用空格：

1.等号两边用空格。如：
如：int a = 100;

2.参数之间用空格。如：
如：test(int a, int b, int c)

3.语句末的分号与前面内容不要加空格。
如：test(a, b, c);

4.其它有助让代码更美观的地方。

合理使用括号：

1.用括号分隔子表达式，不要只靠默认优先级来判断。
如：((a && b) || (c && d))

2.用括号分隔 if/while/for 等语句的代码块，那怕代码只有一行。
如:

```
if(a > b)
{
        return c;
}
```

合理的缩进方式：

每一级都正常缩进，用 tab 缩进取代空格缩进(Linux kernel 也遵循此规则)。用空格缩进的目的是防止代码因编辑器的 tab 宽度不同而变乱，这个担心现在是多余的了，代码编辑器都支持 tab 宽度设置了。 如果缩进的居次太多（比如超过三层），可能是代码设计上出了问题。

如:

```
if(a > b)
```

```
{
        for(i = 0; i < 100; i++)
        {
                …
        }
}
```

遵从团队的习惯。这个是最重要的，一个团队就要像一个团队的样子，不管你的水平有多高，遵循团队的规则是一个程序员的基本素养。如果团队的规则确实不好，大家应该一起完善它。

做到这一点，你已经走近专业程序员了，重新做一遍练习吧。随着后面的学习，你就可以真正走进专业程序员这个行列了。

需求简述：

或许你还在欣赏用良好代码风格重新编写的双向链表，看起来不错，不是吗？不过这还远远不够，专业程序员要有精益求精的精神。至于要精到什么程度，与 具体需求有关，如果只是写个小程序验证一下某个想法，那完成需要的功能就行了，如果是开发一个基础程序库，那就要考虑更多了。侯捷先生说过，学从难处学， 用从易处用。这里我们是学习，就要精得不能再精为止，精到钻牛角尖为止。在后面的几章中，我们将对这个双向链表进行持续的重构，这个过程是循序渐近的，请 读者不要着急，稳扎稳打的学习是才最好的。在这一节中，我们要学习的是程序的封装性，请读者思考下面几个问题：

1· 什么是封装？
2· 为什么要封装？
3· 如何实现封装？

花上半小时去思考，尝试回答上述问题，然后按你的想法重写双向链表(多写多练，不要偷懒)。

1.什么封装？

人有隐私，程序也有隐私。有隐私不是什么坏事，没有隐私人就不是人了，程序也不成其为程序了。问题是隐私不应该让别人知道，否则伤害的不仅仅是自 己，相关人物也会跟着倒霉，“艳照门”就是个典型的例子。程序隐私的暴露，造成的伤害不一定有“艳照门”大，也不一定比它小，反正不要小看它就行了。封装 就是要保护好程序的隐私，不该让调用者知道的事，就坚决不要暴露出来。

2.为什么要封装？

总体来说，封装主要有以下两大好处(具体影响后面再说):

隔离变化。程序的隐私通常是程序最容易变化的部分，比如内部数据结构，内部使用的函数和全局变量等等，把这些代码封装起来，它们的变化不会影响系统的其它部分。

降低复杂度。接口最小化是软件设计的基本原则之一，最小化接口容易被理解和使用。封装内部实现细节，只暴露最小的接口，会让系统变得简单明了，在一定程度上降低了系统的

复杂度。

3.如何封装？

隐藏数据结构

暴露内部数据结构，会使头文件看起来杂乱无章，让调用者发蒙。其次是如果调用者图方便，直接访问这些数据结构的成员，会造成模块之间紧密耦合，给以 后的修改带来困难。隐藏数据结构的方法很简单，如果是内部数据结构，外面完全不会引用，则直接放在 C 文件中就好了，千万不要放在头文件里。如果该数据结构 在内外都要使用，则可以对外暴露结构的名字，而封装结构的实现细节，做法如下：

在头文件中声明该数据结构。

如：
struct _LrcPool;
typedef struct _LrcPool LrcPool;

在 C 文件中定义该数据结构。

```
struct _LrcPool
{
    size_t unit_size;
    size_t n_prealloc_units;
};
```

提供操作该数据结构的函数，哪怕只是存取数据结构的成员，也要包装成相应的函数。

如：
void* lrc_pool_alloc(LrcPool* thiz);
void lrc_pool_free(LrcPool* thiz, void* p);

提供创建和销毁函数。因为只是暴露了结构的名字，编译器不知道它的大小(所占内存空间)，外部可以访问结构的指针(指针的大小的固定的)，但不能直接声明结构的变量，所以有必要提供创建和销毁函数。

如：
这样是非法的：LrcPool lrc_pool;
应该对外提供创建和销毁函数。
LrcPool* lrc_pool_new(size_t unit_size, size_t n_prealloc_units);
void lrc_pool_destroy(LrcPool* thiz);

任何规则都有例外。有些数据结构纯粹是社交型的，为了提高性能和方便起见，常常不需要对它们进行封装，比如点(Point)和矩形(Rect)等。当然封装也不是坏事，MFC 就对它们作了封装，是否需要封装要根据具体情况而定。

隐藏内部函数

内部函数通常实现一些特定的算法(如果具有通用性，应该放到一个公共函数库里)，对调用者没有多大用处，但它的暴露会干扰调用者的思路，让系统看起 来比实际的复杂。函数名也会污染全局名字空间，造成重名问题。它还会诱导调用者绕过正规接口走捷径，造成不必要的耦合。隐藏内部函数的做法很简单：

在头文件中，只放最小接口函数的声明。

在 C 文件上，所有内部函数都加上 static 关键字。

禁止全局变量

除了为使用单件模式(只允许一个实例存在)的情况外，任何时候都要禁止使用全局变量。这一点我反复的强调，但发现初学者还是屡禁不止，为了贪图方便而使用全局变量。请读者从现在开始就记住这一准则。

全局变量始终都会占用内存空间，共享库的全局变量是按页分配的，那怕只有一个字节的全局变量也占用一个 page，所以这会造成不必要空间浪费。全局 变量也会给程序并发造成困难，想把程序从单线程改为多线程将会遇到麻烦。重要的是，如果调用者直接访问这些全局变量，会造成调用者和实现者之间的耦合。

在整个系统程序员成长计划中，我们都是以面向对象的方式来设计和实现的(封装就是面向对象的主要特点之一)。为了避免不必要的概念混淆，这里先解释一下对象和类：

关于对象：对象就是某一具体的事物，比如一个苹果, 一台电脑都是一个对象。每个对象都是唯一的实例，两个苹果，无论它们的外观有多么相像，内部成分有多么相似，两个苹果毕竟是两个苹果，它们是两个不同的对 象。对象可以是一个实物，也可以是一个概念，比如一个苹果对象是实物，而一项政策就是一个概念。在软件中，对象是一个运行时概念，它只存在于运行环境中， 比如：代码中并不存在窗口对象这样的东西，要创建一个窗口对象一定要运行起来才行。

关于类：对象可能是一个无穷的集合，用枚举的方式来表示对象集合不太现实。抽象出对象的特征和功能，按此标准将 对象进行分类，这就引入类的概念。类就是一类事物的统称，类实际上就是一个分类的标准，符合这个分类标准的对象都属于这个类。当然，为了方便起见，通常只 需要抽取那些对当前应用来说是有用的特征和功能。在软件中，类是一个设计时概念，它只存在于代码中，运行时并不存在某个类和某个类之间的交互。我们说，编 写一个双向链表，实际上指的是双向链表这个类。

C 语言里并没有类这个概念，我也不想因为引入这个概念让读者感到迷惑。在后面的讲述中，我不会刻意区分类和对象，我们说对象，可能是指单个对象，也 可能是指对象所属的类，要根据上下文进行区分(这种区分通常是很直观的)。我并不是这种做法的首创者，见过好几本书都是这样做的，希望挑剔的读者不要在这 个概念问题上纠缠。

好了，如果你实现的双向链表没有达到这个标准，请重做一遍练习吧。

需求简述

Write Once, Debug Everywhere。据说这是流传于 JAVA 程序员中间的一句笑话，Sun 公司用来形容 JAVA 的跨平台性的原话是 Write once, run anywhere(WORA) 。后者是理想的，前者才是现实。如果我们的双向链表可以到处运行，那就太好了。Write once, run anywhere(WORA)是我们的目标，不过我们先要面对现实，回到双向链表上，请读者思考下列问题：

1.专用双向链表和通用双向链表各自的特点与适用范围。

2.如何编写一个通用的双向链表？

花点时间思考一下，再尝试编写一个通用的双向链表，或许实现得不是那么优美，但是我们迈出了走向通用性的第一步。

1.专用链表和通用链表各自的特点与适用范围。

专用链表在这里是指它的实现和调用耦合在一起，只能被一个调用者使用，而不能单独在其它地方被重用。通用链表则相反，它具有通用性，可以在多处被重 复使用。尽管通用链表相对专用链表来说有很多优越之处，不过简单的断定通用链表比专用链表好也是不公正的，因为它们都有自己的优点和适用范围：

专用链表的优点：

更高性能。专用链表的实现和调用在一起，可以直接访问数据成员，省去了包装函数带来的性能开销，可以提高时间性能。专用链表无需实现完整的接口，只要满足自己的需要就行了，生成的代码更小，因此可以提高空间性能。

更少依赖。自己实现不用依赖于别人。有时候你要写一个规模不大的跨平台程序，比如想在展讯手机平台和 MTK 手机平台上运行，虽然有现存的库可用，但你又不想把整个库移植过去，那么实现一个专用链表是不错的选择。

实现简单。实现专用链表时，不需要考虑在各种复杂应用情况下的特殊要求，也不需要提供完整的接口，所以实现起来比通用链表更为简单。

通用链表的优点(从全局来看)：

可靠性更高。通用链表的实现要复杂得多，复杂的东西意味着不可靠。但它是可以重复使用的，其存在的问题会随每一次重用而被发现和改正，慢慢的就行成一个可靠的函数库。

开发效率更高。通用链表的实现要复杂得多，复杂的东西也意味着更高的开发成本。同样因为它是可以重复使用的，开发成本会随每一次重用而降低，从整个项目来看，会大大提高开发效率。

考虑到链表是最常用的数据结构之一，很多地方都会用到它，实现通用的链表会更有价值。接下来我们要实现一个通用的链表，不过请大家记住，实现通用的 链表并不是我们的目标，而是我们学习软件设计方法的手段。前面我许诺过要以简单的数据结构讲述复杂的软件设计方法，链表就是其中的载体之一。

2.如何编写一个通用的链表？

编写通用链表是一项复杂的任务，不可能在这一节中把它阐述清楚，这里我们先考虑三个问题：

存值还是存指针

通用链表首先是要做能够存放任何数据类型的数据，新手常见的做法是定义一个抽象数据类型，需要什么存放什么就定义成什么。如：

```
typedef int Type;
typedef struct _DListNode
{
```

```
    struct _DListNode* prev;
    struct _DListNode* next;
    Type data;
}DListNode;
```

这样的链表算不上是通用的，因为你存放整数时编译一次，存放字符串时，重义 Type 再编译一次，存放其它类型同样要重复这个过程。麻烦不说，关键是 没有办法同时使用多个数据类型。我们要找到一种同时可以表示不同数据类型的类型才行，有人说可以用 union，但是数据类型是无穷无尽的，不可能在 union 中表示它们的全部。

可行的办法有两种：

存值：

```
typedef struct _DListNode
{
    struct _DListNode* prev;
    struct _DListNode* next;
    void*  data;
    size_t length;
}DListNode;
```

存入时拷贝一份数据，保存数据的指针和长度。考虑到拷贝数据会带来性能开销，不合符 C 语言的风格，而且 C 语言中没有构造函数，实现深拷贝比较麻烦，所以在 C 语言中以这种方式实现的链表很少见。

存指针：

```
typedef struct _DListNode
{
    struct _DListNode* prev;
    struct _DListNode* next;
    void*  data;
}DListNode;
```

只是保存指向对象的指针，存取效率高，是 C 语言中常见的做法。在存放整数时，可以把 void*强制转换成整数使用，以避免内存分配(在现实中，90%以上的情况，链表都是存放结构的)。

让 C++可以调用

这不是一个重要的话题，只是顺便提一下。C++中允许同名函数存在，所以编译器会对函数名重新编码。C++代码包含 C 语言的头文件时，重新编码名字 与 C 语言库中的原函数名不一致，结果造成找不到函数的情况。为了让 C 语言实现的函数在 C++中可以调用，需要在头文件中加点东西才行：

```
#ifdef __cplusplus
extern "C" {
#endif
...
#ifdef __cplusplus
```

```
}
#endif
```

它表示如果在 C++中调用这里的函数，编译器不能对函数名进行重新编码。

完整的接口

作为一个通用的链表，接口要比较完整才行，否则无法满足各种情况的需要(提供完整的接口并不违背最小接口原则)。实现具有完整接口的链表不是件容易的事，读者先实现插入删除等基本操作就行了，后面我们会慢慢扩展它的功能。

(为了避免读起来拗口，本文把双向链表简写成链表了，希望读者不要介意)

需求简述

大部分初学者在编写双向链表时，为了验证相关函数工作是否正常，都会编写一个 dlist_print 的函数，它的功能是在屏幕上 打印出整个双向链表中的数据。从客观上讲，用 dlist_print 输出的信息来判断 dlist 的正确性不是最好的办法，不过脑袋里有质量概念总是值得表 扬的。当把专用的双向链表演化成通用的双向链表时，编写一个 dlist_print 已经不那么简单了。这里我们请读者写一个 dlist_printf 函 数，看看会遇到什么问题。

在专用双向链表中，dlist_printf 的实现非常简单，如果里面存放的是整数，用”%d”打印，存放的字符串，用”%s”打印。现在的麻烦在于双向链表是通用的，我们无法预知其中存在的数据类型，也就是我们要面对数据类型的变化。怎么办呢？初学者常见的做法有：

1.实现多个函数，需要哪个就用哪个。比如实现的有 dlist_print_int 用来打印存放整数的双向链表，dlist_print_string 用来打印存放字符串的双向链表，如此等等，其它类型都有自己的打印函数。

这种做法的缺点有：一是每个函数的实现方式类似，造成大量重复的代码。二是数据类型的种类不确定，每种数据类型都要写一个 print 函数，当要存放新的数据类型时，需要修改 dlist 的实现。

2.传入一个附加参数来决定如何打印。比如传入 1 表示按整数方式打印，传入 2 表示按字符串方式打印，以此类推。

这种做法比第一种好一点，至少不会造成大量重复的代码。但是同样存在增加新类型时要修改 dlist_print 函数的问题。

3 调用 dlist 的接口函数获取每一个位置的数据并打印出来。

它可以避免前面两种方法的缺点，而且是一种很直观的方式。奇怪的是偏偏很少有人这样去做，原因可能有两个，其一是太拘泥于传统的实现方式而没有想到这一种。其二是担心性能问题，因为通过索引取值，每一次都从头开始定位，其性能开销为 O(n*n)。

其实这种方法是可以接受的，dlist_print 是用于辅助测试，我们并不在乎它的性能开销，而且很少在链表中存放成千上万的数据，它带来的性能影响也没有想的那样严重。

不过在这里我们要介绍一种新的方法:

dlist_print 的大体框架为：

```
    DListNode* iter = thiz->first;
    while(iter != NULL)
    {
        print(iter->data);
        iter = iter->next;
    }
```

在上面代码中，我们主要是不知道如何实现 print(iter->data);这行代码。可是谁知道呢？很明显，调用者知道，因为调用者知道 里面存放的数据类型。OK，那让调用者来做好了，调用者调用 dlist_print 时提供一个函数给 dlist_print 调用，这种回调调用者提供的函 数的方法，我们可以称它为回调函数法。

调用者如何提供函数给 dlist_print 呢？当然是通过函数指针了。变量指针指向的是一块数据，指针指向不同的变量，则取到的是不同的数据。函 数指针指向的是一段代码（即函数），指针指向不同的函数，则具有不同的行为。函数指针是实现多态的手段，多态就是隔离变化的秘诀，这里只是一个开端，后面 我们会逐步的深入学习。

回到正题上，我们看如何实现 dlist_print：
定义函数指针类型：
typedef DListRet (*DListDataPrintFunc)(void* data);

声明 dlist_print 函数：
DListRet dlist_print(DList* thiz, DListDataPrintFunc print);

实现 dlist_print 函数：

```
DListRet dlist_print(DList* thiz, DListDataPrintFunc print)
{
    DListRet ret = DLIST_RET_OK;
    DListNode* iter = thiz->first;
    while(iter != NULL)
    {
        print(iter->data);
        iter = iter->next;
    }
    return ret;
}
```

调用方法

```
static DListRet print_int(void* data)
{
    printf("%d ", (int)data);
    return DLIST_RET_OK;
}
…
dlist_print(dlist, print_int);
```

所有问题都解决了，是不是很简单？我以前写过一篇关于函数指针的 BLOG，文中声称不懂

函数指针就不要自称是C语言高手，现在我仍然坚持这个观点。函数指针的概念本身很简单，关键在于灵活应用，这里是一个最简单的应用，希望读者仔细体会一下，后面将会有大量篇幅介绍。

我写了一个简单的示例，它的实现并不完善，不过用来演示我们到目前为止学到的内容已经够了。有兴趣的读者请到**这里**下载。

需求简述

这里我们请读者实现下列功能：

对一个存放整数的双向链表，找出链表中的最大值。
对一个存放整数的双向链表，累加链表中所有整数。

多写多练，不要偷懒，写完之后请仔细思考一下有无改进的余地。

实现这两个函数并不是件难事，但真正写好的人并不多。初学者通常的做法有两种：

1.各写一个独立的函数。dlist_find_max 用来找出最大值，dlist_sum 用来求和。这种做法和前面写 dlist_print 时所犯的错误一样，会造成重复的代码，让 dlist 的实现随着应用环境的变化而变化。

2.采用回调函数法。细心的初学者会发现，这两个函数的实现与 dlist_print 的实现很类似，无非是 print 那行代码要换成别的功能。能想 到这一点很好，不过在真正动手时，发现每个回调函数都要保存一些中间数据。大部分人选择了用全局变量来保存，这可以实现要求的功能，但违背了禁用全局变量 原则。

这两个函数没有什么实用价值，但是通过它们我们可以学习几点：

1.不要编写重复的代码

按传统的方法写出 dlist_find_max 之后，每个人都知道这个函数与 dlist_print 很类似，在写出 dlist_sum 之后，那种感 觉就更明显了。在这个时候，不应该停下来，而是要想办法把这些重复的代码抽出来。即使因为经验所限，也要极力去想思考和查资料。

写重复的代码很简单，甚至凭本能都可以写出来。但要想成为优秀的程序员，你一定要克服自己的惰情，因为重复的代码造成很多问题：

重复的代码更容易出错。在写类似代码的时候，几乎所有人(包括我)都会选择 Copy&Paste 的方法，这种方法很容易犯一些细节上的错误，如果某个地方修改不完整，那就留下了”不定时”的炸弹，说不定什么时候会暴露出来。

重复的代码经不起变化。无论是修改 BUG，还是增加新特性，往往你要修改很多地方，如果忘掉其中之一，你同样得为此付出代价。请记住古惑仔的话，出来混迟早是要还的。大师们说过，在软件中欠下的 BUG，你会为此还得更多。

去除重复代码往往不是件简单的事情，需要更多思考和更多精力，不过事实证明这是最值得的投资。在这里，我们要怎么抽取这些重复的代码呢？

这三个函数无非是要遍历双向链表并做一些事情，遍历双向链表我们可以提供一个

dlist_foreach 函数，至于要做什么，这是千变万化的行为，可以通过回调函数让调用者去做。

2.任何回调函数都要有上下文

大部分初学者都选择了回调函数法，不过都无一例外的选择了用全局变量来保存中间数据，这里我不想再强调全局变量的坏处了，记性不好的读者可以看看前面的内容。我们要说的是，在这种情况下，如何避免使用全局变量。

很简单，给回调函数传递额外的参数就行了。这个参数我们称为回调函数的上下文，变量名用 ctx(context 的缩写)。要在这个上下文中存放什么东西呢？那得根据具体的回调函数而定，为了能保存任何数据类型，我们选择 void*表示这个上下文。

下面我们看看怎么实现这个 dlist_foreach:

```
DListRet dlist_foreach(DList* thiz, DListVisitFunc visit, void* ctx)
{
    DListRet ret = DLIST_RET_OK;
    DListNode* iter = thiz->first;
    while(iter != NULL && ret != DLIST_RET_STOP)
    {
        ret = visit(ctx, iter->data);
        iter = iter->next;
    }
    return ret;
}
```

visit 是回调函数，ctx 就是我们说的上下文。要特别强调的一点是，ctx 应该作为回调函数的第一个参数。为什么呢？在前面我们讲过的面向对象 的函数命名规则中，我们以 thiz 作为函数的第一个参数，而 thiz 通常也就是函数的上下文。如果在这里恰好 ctx==thiz，就不需要因为参数顺序不 同而做转换了。

实现求和的回调函数：

```
static DListRet sum_cb(void* ctx, void* data)
{
    long long* result = ctx;
    *result += (int)data;
    return DLIST_RET_OK;
}
```

调用 foreach:

```
long long sum = 0;
dlist_foreach(thiz, sum_cb, &sum);
```

是不是很简单？以后在使用回调函数时，记得多加一个 ctx 参数，即使暂时用不着，留着方便以后扩展。好了，请读者用类似的方法实现查找最大值的功能吧。

3.只做份内的事

我见到不少任劳任怨的程序员，别人让他做什么他就做什么，不管是不是份内的事，不管是上司要求的还是同事要求的，都来者不拒。别人说需要一个 XXX 功能的函数，他就写一个

函数在他的模块里，日积月累后，他的模块变得乱七八糟的，成了大杂烩。我亲眼见过在系统设置和桌面两个模块里，提供很多毫不相干的 函数，这些函数造成不必要的耦合和复杂度。

在这里也是一样的，求和和求最大值不是 dlist 应该提供的功能，放在 dlist 里面实现是不应该的。为了能实现这些功能，我们提供一种满足这些需求的机制就好了。热心肠是好的，但一定不能违背原则，否则就费力不讨好了。

本节的示例请到这里**下载**。

需求简述

这里我们请读者实现下列功能：

对一个存放字符串的双向链表，把存放在其中的字符串转换成大写字母。

对 于初学者来说这道题有点难度，很少有人能完全做对的。不过没关系，我们的目标并不是要难倒读者，而是要刺激读者去思考，加深学习的印象。有了前面两次的经 验，我想没有人再去写一个 dlist_to_upper 的函数了，大家都会调用 dlist_foreach 来实现。不过新的问题又出现了，初学者常犯的错 误有以下几种：

1. 转换大写的方法不对。

```
    char* p = str;
    if(p != NULL)
    {
        while(*p != '\0')
        {
            if('a' <= *p && *p <= 'z')
            {
                *p = *p - ('a' - 'A');
            }
            p++;
        }
}
```

这是我们在课本里学到的写法，但在工程中是不能这样做的。因为大小写字母在不同语言中的定义是不一样的，’a’是一个字符常量，它的值在任何时候都 是 97，但在不同语言中，97 却不一定代表’a’。我们不能简单的认为在 97(‘a’)-122(‘z’)之间的字符就是小写字母，而是应该调用标准 C 函 数 islower 来判断，同样转换为大写应该调用 toupper 而不是减去一个常量。

2. 在双向链表中存放常量字符串，转换时出现段错误。

```
    DList* dlist = dlist_create();
    dlist_append(dlist, "It");
    dlist_append(dlist, "is");
    dlist_append(dlist, "OK");
    dlist_append(dlist, "!");
```

```
dlist_foreach(dlist, str_toupper, NULL);
dlist_destroy(dlist);
```

运行时会出现 Segmentation fault 错误。原因是”It”等字符串是常量，常量是不能修改的。

3. 在双向链表中存放的是临时变量，转换后发现所有字符串都一样。

```
char str[256] = {0};
DList* dlist = dlist_create();
strcpy(str, "It");
dlist_append(dlist, str);
strcpy(str, "is");
dlist_append(dlist, str);
strcpy(str, "OK");
dlist_append(dlist, str);
strcpy(str, "!");
dlist_append(dlist, str);
dlist_foreach(dlist, str_toupper, NULL);
dlist_foreach(dlist, str_print, NULL);
dlist_destroy(dlist);
```

运行时发现打印出几个感叹号。原因是 dlist_append 时没有拷贝一份，所以在 dlist 中存放的是同一个地址。而且这个 dlist 在当前函数返回后，里面保存的数据都无效了，因为这些数据指向的是临时变量。

4. 存放时拷贝了数据，但没有 free 分配的内存。

```
DList* dlist = dlist_create();
dlist_append(dlist, strdup("It"));
dlist_append(dlist, strdup("is"));
dlist_append(dlist, strdup("OK"));
dlist_append(dlist, strdup("!"));
dlist_foreach(dlist, str_toupper, NULL);
dlist_foreach(dlist, str_print, NULL);
dlist_destroy(dlist);
```

这里看起来工作正常了，但存在内存泄露的 BUG。strdup 调用 malloc 分配了内存，但没有地方去 free 它们。

初学者对内存和指针只有一知半解的认识，常常犯一些连自己都莫名其妙的错误。为了避免这些不必要的错误，今天我们要学习各种数据存放的位置以及它们的特性，让初学者对编程有更进一步的认识。在程序中，数据存放的位置主要有以下几个：

1.未初始化的全局变量(.bss 段)

已经记不清 bss 代表 Block Storage Start 还是 Block Started by Symbol。像我这种没有用过那些史前计算机的人，终究无法明白这样怪异的名字，记不住也是不足为奇的。不过没有关系，重要的是，我们要清楚什么数据是 存放在 bss 段中的，这些数据有什么样的特点以及如何使用它们。

通俗的说，bss 段是用来存放那些没有初始化的和初始化为 0 的全局变量的。它有什么特点

呢，让我们来看看一个小程序的表现。

```
int bss_array[1024 * 1024];
int main(int argc, char* argv[])
{
    return 0;
}
# gcc -g bss.c -o bss.exe
# ls -l bss.exe
-rwxrwxr-x 1 root root 5975 Nov 16 09:32 bss.exe
# objdump -h bss.exe |grep bss
24 .bss          00400020  080495e0  080495e0  000005e0  2**5
```

变量 bss_array 的大小为 4M，而可执行文件的大小只有 5K。 由此可见，bss 类型的全局变量只占运行时的内存空间，而不占用文件空间。

现代大多数操作系统，在加载程序时，会把所有的 bss 全局变量清零。但为保证程序的可移植性，手工把这些变量初始化为 0 也是一个好习惯，这样这些变量都有个确定的初始值。

当然作为全局变量，在整个程序的运行周期内，bss 数据是一直存在的。

2.初始化过的全局变量 (.data 段)

与 bss 相比，data 段就容易明白多了，它的名字就暗示着里面存放着数据。当然，如果数据全是零，为了优化考虑，编译器把它当作 bss 处理。通俗的说，data 段用来存放那些初始化为非零的全局变量。它有什么特点呢，我们还是来看看一个小程序的表现。

```
int data_array[1024 * 1024] = {1};
int main(int argc, char* argv[])
{
    return 0;
}
# ls -l data.exe
-rwxrwxr-x 1 root root 4200313 Nov 16 09:34 data.exe
# objdump -h data.exe |grep \\.data
23 .data         00400020  080495e0  080495e0  000005e0  2**5
```

仅仅是把初始化的值改为非零了，文件就变为 4M 多。由此可见，data 类型的全局变量是即占文件空间，又占用运行时内存空间的。

同样作为全局变量，在整个程序的运行周期内，data 数据是一直存在的。

3.常量数据 (.rodata 段)

rodata 的意义同样明显，ro 代表 read only，rodata 就是用来存放常量数据的。关于 rodata 类型的数据，要注意以下几点：

o 常量不一定就放在 rodata 里，有的立即数直接和指令编码在一起，存放在代码段(.text)中。

o 对于字符串常量，编译器会自动去掉重复的字符串，保证一个字符串在一个可执行文件(EXE/SO)中只存在一份拷贝。

o rodata 是在多个进程间是共享的，这样可以提高运行空间利用率。

o 在有的嵌入式系统中，rodata 放在 ROM(或者 norflash)里，运行时直接读取，无需加载到 RAM 内存中。

o 在嵌入式 linux 系统中，也可以通过一种叫作 XIP（就地执行）的技术，也可以直接读取，而无需加载到 RAM 内存中。

o 常量是不能修改的，修改常量在 linux 下会出现段错误。

由此可见，把在运行过程中不会改变的数据设为 rodata 类型的是有好处的：在多个进程间共享，可以大大提高空间利用率，甚至不占用 RAM 空间。同 时由于 rodata 在只读的内存页面(page)中，是受保护的，任何试图对它的修改都会被及时发现，这可以提高程序的稳定性。

字符串会被编译器自动放到 rodata 中，其它数据要放到 rodata 中，只需要加 const 关键字修饰就好了。

4.代码 (.text 段)

text 段存放代码(如函数)和部分整数常量，它与 rodata 段很相似，相同的特性我们就不重复了，主要不同在于这个段是可以执行的。

5. 栈(stack)

栈用于存放临时变量和函数参数。栈作为一种基本数据结构，我并不感到惊讶，用来实现函数调用，这也司空见惯的作法。直到我试图找到另外一种方式实现递归操作时，我才感叹于它的巧妙。要实现递归操作，不用栈不是不可能，只是找不出比它更优雅的方式。

尽管大多数编译器在优化时，会把常用的参数或者局部变量放入寄存器中。但用栈来管理函数调用时的临时变量（局部变量和参数）是通用做法，前者只是辅助手段，且只在当前函数中使用，一旦调用下一层函数，这些值仍然要存入栈中才行。

通常情况下，栈向下（低地址）增长，每向栈中 PUSH 一个元素，栈顶就向低地址扩展，每从栈中 POP 一个元素，栈顶就向高地址回退。一个有兴趣的问 题：在 x86 平台上，栈顶寄存器为 ESP，那么 ESP 的值在是 PUSH 操作之前修改呢，还是在 PUSH 操作之后修改呢？PUSH ESP 这条指令会向栈中存入什么数据呢？据说 x86 系列 CPU 中，除了 286 外，都是先修改 ESP，再压栈的。由于 286 没有 CPUID 指令，有的 OS 用 这种方法检查 286 的型号。

要注意的是，存放在栈中的数据只在当前函数及下一层函数中有效，一旦函数返回了，这些数据也自动释放了，继续访问这些变量会造成意想不到的错误。

6.堆(heap)

堆是最灵活的一种内存，它的生命周期完全由使用者控制。标准 C 提供几个函数：

malloc 用来分配一块指定大小的内存。
realloc 用来调整/重分配一块存在的内存。
free 用来释放不再使用的内存。
…

使用堆内存时请注意两个问题：

alloc/free 要配对使用。内存分配了不释放我们称为内存泄露(memory leak)，内存泄露多了迟早会出现 Out of memory 的错误，再分配内存就会失败。当然释放时也只能释放分配出来的内存，释放无效的内存，或者重复 free 都是不行的，会造成程序 crash。

分配多少用多少。分配了 100 字节就只能用 100 字节，不管是读还是写，都只能在这个范围内，读多了会读到随机的数据，写多了会造成的随机的破坏。这种情况我们称为缓冲区溢出(buffer overflow)，这是非常严重的，大部分安全问题都是由缓冲区溢出引起的。

手工检查有没有内存泄露或者缓冲区溢出是很困难的，幸好有些工具可以使用，比如 linux 下有 valgrind，它的使用方法很简单，大家下去可以试用一下，以后每次写完程序都应该用 valgrind 跑一遍。

最后，我们来看看在 linux 下，程序运行时空间的分配情况：

```
# cat /proc/self/maps
00110000-00111000 r-xp 00110000 00:00 0          [vdso]
009ba000-009d6000 r-xp 00000000 08:01 768759     /lib/ld-2.8.so
009d6000-009d7000 r--p 0001c000 08:01 768759     /lib/ld-2.8.so
009d7000-009d8000 rw-p 0001d000 08:01 768759     /lib/ld-2.8.so
009da000-00b3d000 r-xp 00000000 08:01 768760     /lib/libc-2.8.so
00b3d000-00b3f000 r--p 00163000 08:01 768760     /lib/libc-2.8.so
00b3f000-00b40000 rw-p 00165000 08:01 768760     /lib/libc-2.8.so
00b40000-00b43000 rw-p 00b40000 00:00 0
08048000-08050000 r-xp 00000000 08:01 993652     /bin/cat
08050000-08051000 rw-p 00007000 08:01 993652     /bin/cat
0805f000-08080000 rw-p 0805f000 00:00 0          [heap]
b7fe8000-b7fea000 rw-p b7fe8000 00:00 0
bfee7000-bfefc000 rw-p bffeb000 00:00 0          [stack]
```

每个区间都有四个属性：

r 表示可以读取。
w 表示可以修改。
x 表示可以执行。
p/s 表示是否为共享内存。

有文件名的内存区间，属性为 r—p 表示存放的是 rodata。
有文件名的内存区间，属性为 rw-p 表示存放的是 bss 和 data
有文件名的内存区间，属性为 r-xp 表示存放的是 text 数据。
没有文件名的内存区间，表示用 mmap 映射的匿名空间。
文件名为[stack]的内存区间表示是栈。
文件名为[heap]的内存区间表示是堆。

对内存的掌握是系统程序员必备的技能，希望读者多加体会。本节示例代码请到**这里下载**。

“ 快”是指开发效率高，“好”是指软件质量高。呵呵，写得又快又好的人就是高手了。记得这是林锐博士下的定义，读他那篇著名的《C/C++高质量编程》时， 我还是个初学者，

印象特别深。我现在仍然赞同他的观点，不过这里标题改为成为高手的秘诀，感觉就有点像标题党了，所以还是用比较通俗的说法吧。废话少说， 请读者回顾一下这段时间的编程经验，回答下面两个问题：

1.快与好是什么关系？写得快就不能写得好？写得好就不能写得快？还是写得好才能写得快？是不是绕晕了？不过这确实是值得思考的问题。

2.我们的时间花在哪里了？记得刚来深圳时到华为面试，面试的人是我的学长。他问我，你一天能写多少行代码？我想了想说，100 行吧。他用看外行的 眼光看着我说，能写 100 行吗？我知道说错话了，赶快补充说，嗯，从整个项目来看可能没有吧。他才点了点头。一天只写 100 行代码？初学者可能觉得不可思 议，以同时应付 10 个网友聊天的速度，写 100 行代码不用三分钟。不过，经过这段时间的练习后，我们想大家已经明白，敲代码不是花时间最多的地方，那时间 又花到哪里去了呢？

1.好与快的关系

几年前和一个朋友聊天时，他抱怨他的上司说，要我写得好又要写快，那怎么可能呢？我当时一愣，反问到，写不好怎么可能写得快？他也一愣。

传统观点认为在功能、成本(人*时间)和质量这个铁三角中，提高质量就意味投入更多成本或者减少一些功能。在功能不变的情况下，不可能在提高质量的同时降低开成成本(对个人来讲就是缩短开发时间)。我的朋友持的正是这种传统观点。

而根据我的经验来看，结论恰恰相反。每次我想以牺牲质量来加快速度的时候，结果反而花了更多时间，甚至可能到最后搞不定而放弃了。有了多次这样的经 验之后，我决定把每一步都做好，从开发的前期来看，我花的时间比别人多一点，但从整个任务来看，我反而能以别人几倍的速度完成任务。时间长了，我形成了这 样的观念：只有写得好才可能写得快。

两种观点截然相反，所以我们都愣了。虽然我相信我的经验没有错，但传统的铁三角定律是大师们总结出来的，也不可能出错。那是怎么回事呢？我开始到处 查资料，但是没有一个人支持我的观点。我又不想这样放弃，后来我用了一个简单的办法进行推理，结果证明两个观点都有各自的适用范围。

这个推理过程很简单，把两种观点推向极端：

先看看以牺牲质量来追求进度的例子。我以前参加过两个大项目，其一个项目的 BUG 总数达到 17000 多个，耗时近三年后项目被取消。另一个项目的 BUG 总数也超过 10000 个，三年之后带着很多 BUG 发布了，结果可想而知，产品很快从市场上消失了。这两个项目在开始阶段都制定了极其可笑的项目计 划，为了赶在这个根本不可能的最后期限前，都采用了牺牲质量的方式来提高开发速度，前期进展都很“顺利”，基本功能点很快就完成了，但是项目马上陷入了无 止境的 debug 之中，开发人员的士气一下跌到谷底，管理层开始暴跳如雷。

如果这两个项目有超过 170000 个 BUG，即使项目不取消，再做时间十年也做不完。由此可见：质量低到一定限度时，降低质量会延长项目时间，如果质量降到最低，那项目永远也不可能完成。这和我的观点一致：写不好就写不快。

再看看追求完美质量的例子。以前参与一个手机模拟器的开发，我们很快达到88%的眞实度，半年之后达到95%的眞实度，客户要98%的眞实度。但是怎么努力也达不到这个标准，花了极大的代价才达到96%多一点，到后来项目被取消了。

如果要达到99%的眞实度，即使项目不取消，再做时间十年也做不完。由此可见：质量高到一定程度，提高质量会延长项目时间，如果质量要高到最高，那任务远也不可能完成。这和传统观点一致，提高质量就要延长开发时间。

从两个极端往中间走，我们可以找到一个中间质量点。低于这个质量点，想以牺牲质量来赶进度，那只会适得其反，质量越低耗时越长。高于这个质量点，想提高质量就得增加成本，质量越高开发时间越长。这样两种观点就统一起来了。

如果在大多数项目中，这个中间质量点是可以作为高质量接受的，那我们就找到了又快又好的最佳方法。这个质量点到底是多少？呵，我可以告诉你，那是87.5。但是谁知道怎么去度量呢？没有人知道，只能凭感觉和经验了。

2.我们的时间花在哪里

经过这段时间的练习，大多数人都体会到敲代码不是耗费时间最多的地方，一个高效率的程序员，并不是打字比别人快，而他节省了别人浪费了的时间。我常 说达到别人五倍的效率并不难，因为在软件开发中，大部分人的大部分时间都浪费掉了，你只要把别人浪费的时间省下来，你的效率就提高上去了。像在优化软件性 能时采用的方法一样，要优化程序员的性能，我们要找出性能的瓶颈。也就是弄清楚我们的时间花在哪些地方，然后想办法省掉某些浪费了的时间。根据我的经验， 耗费时间最多的地方有：

o 分析

需求分析通常是SPEC工程师(或者所谓的系统分析员)的任务，程序员也会参与到这个过程中，但程序员的任务主要是理解需求，然后分析如何实现它 们，这个分析工作也就是软件设计。无论你是在计算机上用设计工具画出正规的软件架构图，还在纸上用自然语言描述出算法的逻辑，甚至在脑海中一闪而过的想法 都是设计。设计其实就是打草稿，把你的想法进行推敲，最后得到可行的方案。设计文档只是设计的结果，是设计的表现形式，没有写设计文档，并不代表没有做设 计(但是写设计文档可以加深你的思考)。

设计本身是一个思考过程，需要耗费大量时间，对于新手来说更是如此。前面几节中的需求并不难，理解它们只需要很少的时间，但要花不少时间去思考其实 现的方法。这个时间因人而异，有的读者到最后也没有想出办法，这没有关系，没有人天生就会的，不会的原因只是因为你暂时还不知道常用的设计方法，甚至连基 本数据结构和算法都不熟悉。

在后面的章节中，我们会一步步的深入学习各种常用设计方法，反复练习基本数据和算法，熟能生巧，软件设计也一样，在你什么都不懂的时候，不可能做出好的设计。你要学习各种经典的设计方法，开始可能生搬硬套，多写多练多思考，到后来就随心所欲了，设计的时间就会大大缩短。

o 测试

要写得好自然离不开测试，初学者都有这个概念。他们忠实的使用了教科书上讲的方法，用scanf输入数据，做些操作之后，用printf打印来，这 是一个完美的输入-处理-输出的过程。

测试也就是要保证正确的输入能产生正确的输出，这种方法的原理是没有错的，但它们确实耗费了我们大量时间。

如果测试只需要做一次，这种方法还是可取的，问题是每次修改之后都要重复这个过程，耗费的时间就更多了。这种工作单调乏味，而且很难坚持做下去，单 元测试做得不全面，就有更多 BUG 等着就调试了。时间久了，或者换人维护了，谁也搞不清楚什么样输入产生什么样的输出，结果可能是连测试也省了，那就等着 把大量的时间浪费在调试上吧。总而言之，这种测试方法不好，我们需要更有效的测试方法才行。

o 调试

测试时发现程序有 BUG，自然要用调试器了，对一些人来说，调试是一件充满挑战和乐趣的事。而对大部分人来说，特别是对我这种做过两年专职调试的人 来说，调试是件无趣无聊无用的工作。熟练使用调试器是必要的，在分析现有软件时，调试器是非常有用的工具。但在开发新软件时，调试器在浪费我们的时间。

调试器是最后一招，只有迫不得已时才使用。一位敏捷方法的高手说他已经记不得上次使用调试器是什么时候了，我想这就是为什么敏捷方法能够提高开发速度的原因吧。因为没有什么比一次性写好，不用调试器更快的方法了。

知道了浪费时间的地方，接下来几节中，我们将介绍避免浪费时间的方法。学完这些方法之后，我希望读者也能达到普通工程师五倍的效率，呵，读完本系列文章后之，希望你会达到更高。

代码阅读法

软件工程实践已经证明 Code Review 是提高代码质量最有效的手段之一，极限编程(XP)更是把 Code Review 推向极致，形成著名的结对编程工作方式，两个程序员在一台电脑前面工作，一个人编写程序，另一个 Review 输入每一行代码，写程序人的专注 于目前细节上的工作，Review 的人同时要从高层次考虑如何改进代码质量，两个人的角色会经常互换。

可惜我即没有结对编程的经验，也没有在 CMM3(及以上)团队中工作过。不过现在我要介绍比结对编程更敏捷更轻量级，但是同样有效的 Review 方 法。这种方法不需要其他程序员配合，有你自己就够了。为了把这种方法与传统的 Code Review 区分开来，我把它称为代码阅读法吧。

很多初学者包括一些有经验的程序员，在敲完代码的最后一个字符后，马上开始编译和运行，迫不急待的想看到自己的工作成果。快速反馈有助于满足自己的成就感，但是同时也会带来一些问题：

让编译器帮你检查语法错误可以省些时间，但程序员往往太专注这些错误了，以为改完这些错误就万事大吉了。其实不然，很多错误编译器是发现不了的，像 内存错误和线程死锁等等，这些错误可能逃过简单的测试而遗留在代码中，直到集成测试或者软件发布之后才暴露出来，那时就要花更大代价去修改它们了。

修改完编译错误之后就是运行程序了，运行起来有错误，就轮到调试器上场了。花了不少时间去调试，发现无非是些低级错误，或许你会自责自己粗心大意， 但是下次可能还是犯同

样的错误。更严重的是这种 debug & fix 的方法，往往是头痛医头脚痛医脚，导致低质量的软件。

让编译器帮你检查语法错误，让调试器帮你查 BUG，这是天经地义的事，但这确实是又慢又烂的方法。就像你要到离家东边 1000 米的地方开会，结果你 往西边走，又是坐车又是搭飞机，花了一周时间，也绕着地球转了一周，终于到了会议室，你还大发感慨说，现代的交通工具真是发达啊。其实你往东走，走路也只 要十多分钟就到了。不管你的调试技巧有多高，都不如一次性写好更高效。

我以前也一样，想赶时间结果花了更多时间，在经过很多痛苦的经历之后，我开始学会放松自己，让自己慢下来。写完程序之后，我会花些时间去阅读它，一 遍两遍甚至多遍之后，才开始编译它，只要有时间，在通过测试之后，我还会阅读它们，每读一遍都有不同的收获，有时候会发现一些错误，有时候会做些改进，有 时候也有新的想法。

下面是我在阅读自己代码时的一些方法：

o 检查常见错误。

第一遍阅读时主要关注语法错误、代码排版和命名规则等等问题，只要看不顺眼就修改它们。读完之后，你的代码很少有低级错误，看起来也比较干净清爽。 第二遍重点关注常见编程错误，比如内存泄露和可能的越界访问，变量没有初始化，函数忘记返回值等等，在后面的章节中，我会介绍这些常见错误，避免这些错误 可以为你省大量的时间。如果有时间，在测试完成之后，还可以考虑是否有更好的实现方法，甚至尝试重新去实现它们。说了读者可能不相信，在学习编程的前几 年，我经常重写整个模块，只我觉得能做得更好，能验证我的一些想法，或提高我的编程能力，即使连续几天加班到晚上十一点，我也要重写它们。

o 模拟计算机执行。

常见错误是比较死的东西，按照检查列表一条一条的做就行了。有些逻辑通常不是这么直观的，这时可以自己模拟计算机去执行，假想你自己是计算机，读入 这些代码时你会怎么处理。这种方法能有效的完善我们的思路，考虑不同的输入数据，各种边界值，这能帮助我们想到一些没有处理的情况，让程序的逻辑更严谨。

o 假想讲给朋友听。

据说在 Code Review 时发现错误的，往往不是 Review 的人而是程序员自己。我也有很多这样的经历，在讲给别人听的时候，别人还没有听明白，自己已经发现里面存 在的错误了。上大学时，我常常把写的或者学到的东西讲给隔壁寝室的一个同学听，他说他从我这里学到很多知识，其实我从讲的过程中，经常发现一些问题，对提 高自己的能力大有帮助。可惜并不是随时都能找到好的听众，幸好我们有另外一个替代办法，记得刚开始写程序时看过一本书(忘记名字了)，作者说他在写程序 时，常常把思路讲给他的布娃娃听。我没有布娃娃当听众，讲给鼠标听总是有点怪怪的 ，所以就假想旁边有个朋友，我把自己的思路讲给他听，同时也假想他来质 疑我。这种方法很效，能够让自己的思路更清晰，据说一些大师也经常使用这种方法。

这种代码阅读法会花你一些时间，但是可以省下更多调试时间，而且能够提高代码质量，可以说是名符其实的“又快又好的” 秘诀之一。至于读几遍合适，要根据情况而定，个人觉得读两到三遍是最佳的投资。

避免常见错误

在C语言中，内存错误是最为人诟病的。这些错误让项目延期或者被取消，引发无数的安全问题，甚至出现人命关天的灾难。抛开这些大道理不谈，它们确实 浪费了我们大量时间，这些错误引发的是随机现象，即使有一些先进工具的帮助，为了找到重现的路径，花上几天时间也不足为怪。如果能够在编写代码的时候避免 这些错误，开发效率至少提高一倍以上，质量可以提高几倍了。这里列举一些常见的内存错误，供新手参考。

o 内存泄露

大家都知道，在堆上分配的内存，如果不再使用了，应该把它释放掉，以便后面其它地方可以重用。在C/C++中，内存管理器不会帮你自动回收不再使用的内存。如果你忘了释放不再使用的内存，这些内存就不能被重用了，这就造成了所谓的内存泄露。

把内存泄露列为首位，倒并不是因为它有多么严重的后果，而因为它是最为常见的一类错误。一两处内存泄露通常不至于让程序崩溃，也不会出现逻辑上的错 误，加上进程退出时，系统会自动释放该进程所有相关的内存(共享内存除外)，所以内存泄露的后果相对来说还是比较温和的。但是，量变会导致质变，一旦内存 泄露过多以致于耗尽内存，后续内存分配将会失败，程序可能因此而崩溃。

现在PC机的内存够大了，加上进程有独立的内存空间，对于一些小程序来说，内存泄露已经不是太大的威胁。但对于大型软件，特别是长时间运行的软件，或者嵌入式系统来说，内存泄露仍然是致命的因素之一。

不管在什么情况下，采取谨慎的态度，杜绝内存泄露的出现，都是可取的。相反，认为内存有的是，对内存泄露放任自流都不是负责的。尽管一些工具可以帮助我们检查内存泄露问题，我认为还是应该在编程时就仔细一点，及早排除这类错误，工具只是用作验证的手段。

o 内存越界访问

内存越界访问有两种：一种是读越界，即读了不属于自己的数据，如果所读的内存地址是无效的，程度立刻就崩溃了。如果所读内存地址是有效的，在读的时 候不会出问题，但由于读到的数据是随机的，它会产生不可预料的后果。另外一种是写越界，又叫缓冲区溢出，所写入的数据对别人来说是随机的，它也会产生不可 预料的后果。

内存越界访问造成的后果非常严重，是程序稳定性的致命威胁之一。更麻烦的是，它造成的后果是随机的，表现出来的症状和时机也是随机的，让BUG的现象和本质看似没有什么联系，这给BUG的定位带来极大的困难。

一些工具可以够帮助检查内存越界访问的问题，但也不能太依赖于工具。内存越界访问通常是动态出现的，即依赖于测试数据，在极端的情况下才会出现，除 非精心设计测试数据，工具也无能为力。工具本身也有一些限制，甚至在一些大型项目中，工具变得完全不可用。比较保险的方法还是在编程是就小心，特别是对于 外部传入的参数要仔细检查。

我们来看一个例子：

```
#include <stdlib.h>
#include <string.h>
int main(int argc, char* argv[])
{
```

```
    char str[10];
    int array[10] = {0,1,2,3,4,5,6,7,8,9};
    int data = array[10];
    array[10] = data;
    if(argc == 2)
    {
        strcpy(str, argv[1]);
    }
    return 0;
}
```

这个例子中有两个错误是新手常犯的：

其一：int array[10] 定义了 10 个元素大小的数组，由于 C 语言中数组的索引是从 0 开始的，所以只能访问 array[0]到 array[9]，访问 array[10]就造成了越界错误。

其二：strcpy(str, argv[1]);这里是否存在越界错误依赖于外部输入的数据，这样的写法在正常下可能没有问题，但受到一点恶意攻击就完蛋了。除非你确定输入数据是在你 控制内的，否则不要用 strcpy、strcat 和 sprintf 之类的函数，而要用 strncpy、strncat 和 snprintf 代替。

o 野指针。

野指针是指那些你已经释放掉的内存指针。当你调用 free(p)时，你眞正清楚这个动作背后的内容吗？你会说 p 指向的内存被释放了。没错，p 本身有变化吗？答案是 p 本身没有变化。它指向的内存仍然是有效的，你继续读写 p 指向的内存，没有人能拦得住你。

释放掉的内存会被内存管理器重新分配，此时，野指针指向的内存已经被赋予新的意义。对野指针指向内存的访问，无论是有意还是无意的，都为此会付出巨大代价，因为它造成的后果，如同越界访问一样是不可预料的。

释放内存后立即把对应指针置为空值，这是避免野指针常用的方法。这个方法简单有效，只是要注意，当然指针是从函数外层传入的时，在函数内把指针置为 空值，对外层的指针没有影响。比如，你在析构函数里把 this 指针置为空值，没有任何效果，这时应该在函数外层把指针置为空值。

o 访问空指针。

空指针在 C/C++中占有特殊的地址，通常用来判断一个指针的有效性。空指针一般定义为 0。现代操作系统都会保留从 0 开始的一块内存，至于这块内存有多大，视不同的操作系统而定。一旦程序试图访问这块内存，系统就会触发一个异常/信号。

操作系统为什么要保留一块内存，而不是仅仅保留一个字节的内存呢？原因是：一般内存管理都是按页进行管理的，无法单纯保留一个字节，至少要保留一个页面。保留一块内存也有额外的好处，可以检查诸如 p=NULL; p[1]之类的内存错误。

在一些嵌入式系统(如 arm7)中，从 0 开始的一块内存是用来安装中断向量的，没有 MMU 的保护，直接访问这块内存好像不会引发异常。不过这块内存是代码段的，不是程序中有效的变量地址，所以用空指针来判断指针的有效性仍然可行。

o 引用未初始化的变量。

未初始化变量的内容是随机的(有的编译器会在调试版本中把它们初始化为固定值，如0xcc)，使用这些数据会造成不可预料的后果，调试这样的BUG也是非常困难的。

对于态度严谨的程度员来说，防止这类BUG非常容易。在声明变量时就对它进行初始化，是一个好的编程习惯。另外也要重视编译器的警告信息，发现有引用未初始化的变量，立即修改过来。

在下面这个例子中，全局变量g_count是确定的，因为它在bss段中，自动初始化为0了。临时变量a是没有初始化的，堆内存str是没有初始化 的。但这个例子有点特殊，因为程序刚运行起来，很多东西是确定的，如果你想把它们当作随机数的种子是不行的，因为它们还不够随机。

```
#include <stdlib.h>
#include <string.h>
int g_count;
int main(int argc, char* argv[])
{
    int a;
    char* str = (char*)malloc(100);
    return 0;
}
```

o 不清楚指针运算。

对于一些新手来说，指针常常让他们犯糊涂。

比如 int *p = …; p+1 等于(size_t)p + 1 吗

老手自然清楚，新手可能就搞不清了。事实上, p+n 等于 (size_t)p + n * sizeof(*p)

指针是C/C++中最有力的武器，功能非常强大，无论是变量指针还是函数指针，都应该非常熟练的掌握。只要有不确定的地方，马上写个小程序验证一下。对每一个细节了然于胸，在编程时会省下不少时间。

o 结构的成员顺序变化引发的错误。

在初始化一个结构时，老手可能很少像新手那样老老实实的，一个成员一个成员的为结构初始化，而是采用快捷方式，如：

```
Struct s
{
    int   1;
    char* p;
};
int main(int argc, char* argv[])
{
    struct s s1 = {4, "abcd"};
    return 0;
}
```

以上这种方式是非常危险的，原因在于你对结构的内存布局作了假设。如果这个结构是第三

方提供的，他很可能调整结构中成员的相对位置。而这样的调整往 往不会在文档中说明，你自然很少去关注。如果调整的两个成员具有相同数据类型，编译时不会有任何警告，而程序的逻辑可能相距十万八千里了。

正确的初始化方法应该是（当然，一个成员一个成员的初始化也行）：

```
struct s
{
    int   1;
    char* p;
};
int main(int argc, char* argv[])
{
    struct s s1 = {.1=4, .p = "abcd"};
    return 0;
}
```

(有的编译器可能不支持新标准)

o 结构的大小变化引发的错误。

我们看看下面这个例子：

```
struct base
{
    int n;
};
struct s
{
    struct base b;
    int m;
};
```

在 OOP 中，我们可以认为第二个结构继承了第一结构，这有什么问题吗？当然没有，这是 C 语言中实现继承的基本手法。

现在假设第一个结构是第三方提供的，第二个结构是你自己的。第三方提供的库是以 DLL 方式分发的，DLL 最大好处在于可以独立替换。但随着软件的进化，问题可能就来了。

当第三方在第一个结构中增加了一个新的成员 int k;，编译好后把 DLL 给你，你直接把它给了客户了，让他们替换掉老版本。程序加载时不会有任何问题，在运行逻辑可能完全改变！原因是两个结构的内存布局重叠了。

解决这类错误的唯一办法就是重新编译全部代码。由此看来，动态库并不见得可以动态替换，如果你想了解更多相关内容，建议你阅读《COM 本质论》。

o 分配/释放不配对。

大家都知道 malloc 要和 free 配对使用，new 要和 delete/delete[]配对使用，重载了类 new 操作，应该同时重载类的 delete/delete[]操作。这些都是书上反复强调过的，除非当时晕了头，一般不会犯这样的低级错误。

而有时候我们却被蒙在鼓里，两个代码看起来都是调用的 free 函数，实际上却调用了不同的实现。比如在 Win32 下，调试版与发布版，单线程与多线 程是不同的运行时库，不同的运行时库使用的是不同的内存管理器。一不小心链接错了库，那你就麻烦了。程序可能动则崩溃，原因在于在一个内存管理器中分配的 内存，在另外一个内存管理器中释放时就会出现问题。

o 返回指向临时变量的指针

大家都知道，栈里面的变量都是临时的。当前函数执行完成时，相关的临时变量和参数都被清除了。不能把指向这些临时变量的指针返回给调用者，这样的指针指向的数据是随机的，会给程序造成不可预料的后果。

下面是个错误的例子：

```
char* get_str(void)
{
    char str[] = {"abcd"};
    return str;
}
int main(int argc, char* argv[])
{
    char* p = get_str();
    printf("%s\n", p);
    return 0;
}
```

下面这个例子没有问题，大家知道为什么吗？

```
char* get_str(void)
{
    char* str = {"abcd"};
    return str;
}
int main(int argc, char* argv[])
{
    char* p = get_str();
    printf("%s\n", p);
    return 0;
}
```

o 试图修改常量

在函数参数前加上 const 修饰符，只是给编译器做类型检查用的，编译器禁止修改这样的变量。但这并不是强制的，你完全可以用强制类型转换绕过去，一般也不会出什么错。

而全局常量和字符串，用强制类型转换绕过去，运行时仍然会出错。原因在于它们是放在.rodata 里面的，而.rodata 内存页面是不能修改的。试图对它们修改，会引发内存错误。

下面这个程序在运行时会出错：

```
int main(int argc, char* argv[])
{
```

```
    char* p = "abcd";
    *p = '1';
    return 0;
}
```

o 误解传值与传引用

在 C/C++中，参数默认传递方式是传值的，即在参数入栈时被拷贝一份。在函数里修改这些参数，不会影响外面的调用者。如：

```
#include <stdlib.h>
#include <stdio.h>
void get_str(char* p)
{
    p = malloc(sizeof("abcd"));
    strcpy(p, "abcd");
    return;
}
int main(int argc, char* argv[])
{
    char* p = NULL;
    get_str(p);
    printf("p=%p\n", p);
    return 0;
}
```

在 main 函数里，p 的值仍然是空值。当然在函数里修改指针指向的内容是可以的。

o 重名符号。

无论是函数名还是变量名，如果在不同的作用范围内重名，自然没有问题。但如果两个符号的作用域有交集，如全局变量和局部变量，全局变量与全局变量之 间，重名的现象一定要坚决避免。gcc 有一些隐式规则来决定处理同名变量的方式，编译时可能没有任何警告和错误，但结果通常并非你所期望的。

下面例子编译时就没有警告：

t.c

```
#include <stdlib.h>
#include <stdio.h>
int count = 0;
int get_count(void)
{
    return count;
}
```

main.c

```
#include <stdio.h>
extern int get_count(void);
int count;
int main(int argc, char* argv[])
{
```

```
    count = 10;
    printf("get_count=%d\n", get_count());
    return 0;
}
```

如果把 main.c 中的 int count;修改为 int count = 0;，gcc 就会编辑出错，说 multiple definition of `count'。它的隐式规则比较奇妙吧，所以还是不要依赖它为好。

o 栈溢出。

我们在前面关于堆栈的一节讲过，在 PC 上，普通线程的栈空间也有十几 M，通常够用了，定义大一点的临时变量不会有什么问题。

而在一些嵌入式中，线程的栈空间可能只 5K 大小，甚至小到只有 256 个字节。在这样的平台中，栈溢出是最常用的错误之一。在编程时应该清楚自己平台的限制，避免栈溢出的可能。

o 误用 sizeof。

尽管 C/C++通常是按值传递参数，而数组则是例外，在传递数组参数时，数组退化为指针（即按引用传递），用 sizeof 是无法取得数组的大小的。

从下面这个例子可以看出：

```
void test(char str[20])
{
    printf("%s:size=%d\n", __func__, sizeof(str));
}
int main(int argc, char* argv[])
{
    char str[20]  = {0};
    test(str);
    printf("%s:size=%d\n", __func__, sizeof(str));
    return 0;
}
[root@localhost mm]# ./t.exe
test:size=4
main:size=20
```

o 字节对齐。

字节对齐主要目的是提高内存访问的效率。但在有的平台(如 arm7)上，就不光是效率问题了，如果不对齐，得到的数据是错误的。

所幸的是，大多数情况下，编译会保证全局变量和临时变量按正确的方式对齐。内存管理器会保证动态内存按正确的方式对齐。要注意的是，在不同类型的变量之间转换时要小心，如把 char*强制转换为 int*时，要格外小心。

另外，字节对齐也会造成结构大小的变化，在程序内部用 sizeof 来取得结构的大小，这就足够了。若数据要在不同的机器间传递时，在通信协议中要规定对齐的方式，避免对齐方式不一致引发的问题。

o 字节顺序。

字节顺序历来是设计跨平台软件时头疼的问题。字节顺序是关于数据在物理内存中的布局的问题，最常见的字节顺序有两种：大端模式与小端模式。

大端模式是高位字节数据存放在低地址处，低位字节数据存放在高地址处。

小端模式指低位字节数据存放在内存低地址处，高位字节数据存放在内存高地址处；

在普通软件中，字节顺序问题并不引人注目。而在开发与网络通信和数据交换有关的软件时，字节顺序问题就要特殊注意了。

o 多线程共享变量没有用 valotile 修饰。

关键字 valotile 的作用是告诉编译器，不要把变量优化到寄存器里。在开发多线程并发的软件时，如果这些线程共享一些全局变量，这些全局变量最好用 valotile 修饰。这样可以避免因为编译器优化而引起的错误，这样的错误非常难查。

o 忘记函数的返回值

函数需要返回值，如果你忘记 return 语句，它仍然会返回一个值，因为在 i386 上，EAX 用来保存返回值，如果没有明确返回，EAX 最后的内容被返回，所以 EAX 的内容是随机的。

自动测试

手工测试比没有测试强一点，但是它存在的问题让它很难在实践中应用：手工输入数据的过程单调乏味，很难长期坚持。每次都要重新输入数据，浪费大量时 间。测试用例不能累积，测试往往不完整。用人脑判断输出的正误，浪费人力也存在误差。要写得好测试自然不能省，要写得快就需要更好的测试方法。

更好的测试方法当然是自动测试了。幸运的是，刚进入这个行业我就接触了自动的测试 (呵，读本文的初学者就更幸运了)，我的第一份正式工作是在测试组写测试程序。当时测试组也算是人才济济了，居然有几个北大毕业的，不过她们都不懂 Linux，所以我被指派去为移植到 Linux 上的模块写测试程序。这些模块都有测试程序，但这些测试程序的功能太弱了，我的上司要求开发人员改进，但那 些开发人员太自以为是了，根本不理我们，所以我们只好自己重写这些测试程序。模块很多，大概有 50 多个模块，熟悉这些模块也需要不少时间，按每两个工作日 写一个测试程序，上司给我 5 个月时间。

记得第一个模块是 RDFParser，RDF(资源描述框架)是 XML 的一种应用，RDFParser 实际上是一个 XML 解析器，并包装成 RDF 要 求的接口。由于我对 C/C++还不太熟悉，对 RDF 更不熟悉了，花了两周时间才写出这个测试程序。运行起来有些不正常，我确信不是测试程序的问题，就去请 开发人员帮忙来看一下。负责 RDFParser 的那个程序员是人大毕业，我没有见过第二个比他更自以为是的程序员了，他刚在我座位上坐下就很大声说，你们 QA 的人太蠢了！

当时一听就愣了，不过我是新来的，见上司都没反应，自然就忍了。我列举了一些证据是模块里面的问题，他听也不听，只是不断重复的说，不可能是我程序 的问题，你们 QA 的人太蠢了，总是浪费我的时间。过了一会儿，他终于闭上了嘴巴，又等了一会儿才说，等会儿重新发个版本给你吧。后来又请他过来四五次，结 果每次都是他的问题。

之后我再没有听到他说过你们 QA 的人太蠢了的话。为了避免让他抓到把柄来嘲笑测试组，我决定请他来查问题之前做更详细的测试。当时我写的测试程序和 现在初学者写的测试程序没有两样，都是从教科书上学来的，先通过 scanf 从终端输入数据，调用被测函数，再把结果 printf 出来，这花了我太多时间。 想到后面还有 50 多个模块的测试程序要写，这样下去不行，一定得想个办法。

后来我把输入的数据和期望的结果都写到一个 INI 文件中，测试程序从这个文件中读入数据，运行测试，再和预期结果比较，整个过程都自动化了。写了一 个 INI 文件的解析器花了我一周时间，又重写了那个测试程序，整整花了我一个月时间完成 RDFParser 的测试程序。进度自然大落后了，还好上司知道后 并没有责备我，让我慢慢做就好了。

写第二个测试程序时把 INI 解析的代码拷贝过去，再加一些调用模块的代码就写好了，第三个也是如此。写了几个之后，我发现了 INI 解析有个 BUG， 结果每个测试程序我都要去修改，想到维护起来太麻烦了，就把 INI 解析器的接口规范化了，编译成一个独立共享库。又写了几个测试程序，我写烦了，原因是测 试程序无非就是读入数据，调用被测函数，再检查结果，这个过程太无聊了。想到后面还要把这个过程重复几十遍，郁闷了几天之后，突然灵机一动，我决定写了一 个代码产生器来产生这些代码。开始的代码产生器用 C 写的，用一个简单的规则来描述被测函数，通过这些规则来产生测试程序。我把这些东西和 INI 解析器放在 一个独立的库中，把它叫作 TesterFrameWork，经过几个测试程序的验证和完善，后来利用这个 TesterFrameWork，只要一两个小时 就能完成一个测试程序了。有次请开发人员那边一个高手帮我查一个问题，他看一会儿我的 TesterFrameWork 之后，盯着我说，你太聪明了。我笑了 笑说，刚刚开始写 C/C++程序。

一年之后我知道了有个 CPPUnit 之后，为了赶时髦我把 TesterFrameWork 改名为 CxxUnit，非典的时候放假无聊就把它重写了一遍放在 cosoft 上了(之后没有管过它，或许还在吧)。

一个大系统很难自动测试，而一个独立的模块则是最佳的自动测试单元。自动测试和单元测试几乎成了等价的概念，很多人都以为自动测试就是利用 CPPUnit 这样的单元测试框架写个测试程序而已，这完全是错误的，就像有人以为有个设计文档的模板，照着填空就能填出好设计一样。

我自己实现过单元测试框架，不是像有些人出于模仿去实现，而完全出于实际的需要，后来我也研究其它测试框架，应该说我对测试程序框架的认识比一般程 序员要深刻。我认为测试程序框架可以减化一些测试程序的工作，但它与自动测试没有密切关系，用不用测试程序框架完全是个人喜好。用测试程序框架未必能写出 好的测试程序，就像用 C++未必能写出好的面向对象的程序一样。

虽然我顺利的完成了那个写测试程序的任务，但我一直被一个问题困扰：如何写测试用例，如何去检测结果？这是测试程序框架帮不上忙的。写测试用例还好 说，通过边界值法，等价类法和路径覆盖法找到最常用的测试用例。检测结果呢？有人说很简单啊，判断返回值就好了。那我问一下 dlist_insert 返回 OK，就眞的 OK 了吗？如果一个函数根本没有返回值，那你怎么判断呢？

测试程序框架是敏捷论者提倡的，在我看来它根本不够敏捷：你要去学习它，了解它的运行机制，要包含它的头文件，链接它的库，有比不用它更敏捷么？重 要的是它根本帮不上什

么有用的忙。前面的问题折磨了我一段时间，于是得出一个可能有点偏激的结论：测试程序框架都是愚蠢的，你真正需要的，它根本帮不了你 (我知道这样说会得罪一些用测试程序框架的朋友，如果你想找我讨论的话，请看完本节的附带示例代码再说)。

就在那个时候，我看到了孟岩老师翻译的《契约式设计(Design by Contract)》，读完之后豁然开朗。或许我还没有明白契约式设计的本质，但我确实知道了写自动测试程序的方法，下面我介绍一下：

o 在设计时，每个函数只完成单一的功能。单一功能的函数容易理解，也容易预测其行为。对测试来说，给定一些输入数据，就知道它的输出和影响，这样函数是最容易测试的。

o 在设计时，把函数分为查询和命令两类。查询函数只查询对象的状态，而不改变对象的状态。命令函数则只修改对象的状态，只返回其操作是否成功的标志，而不返 回对象的状态。比如，dlist_length 查询双向链表的长度，它不修改双向链表的任何状态。dlist_delete 修改对象的状态(删除结点)， 并返回其操作是否成功，而不返回当前长度或者删除的结点之类的状态。

o 在设计时，把查询分为基本查询和复合查询两类。基本查询函数只查询单一的状态，而复合查询可以同时查询多个状态。比如，window_get_width 返回窗口的宽度，这是基本查询函数，widget_get_rect 返回窗口的左上角坐标，宽度和高度，这是复合查询函数。

o 在实现时，检验输入数据，确认使用者正确的调用了函数。契约式设计规定了调用者和实现者双方的责任，调用者需要使用正确的参数，才能保证有正确的 结果。政治家告诉我们，信任但要检查，所以作为实现者就需要检查输入参数是否违背了契约。那怎么检查呢？有人说，如果检查到无效参数就返回一个错误码。这 当然可以，只是不太好，因为大多数人都没有检查返回值的习惯，如果每个地方都检查函数的返回值，也是件很繁琐的事，代码看起来也比较乱。通常我们只检查一 些关键的地方，对于无效参数这样的错误，可能就无声无息的隐藏起来了，这样不好，因为隐藏得越深，发现的时间越晚，修改的代价越大。

在 C++和 JAVA 里，如果参数不正确，通常是 throw 一个无效参数之类的异常，C 语言里面没有异常这个概念，我们需要其它办法才行。有人推荐用 assert 来检查，这是一个好办法，assert 只在调试版本中有效(没有定义 NDEBUG)，这样任何无效调用都在调试版本中暴露出来了。如果配合前 面返回错误码的方法，在发布版本中也可能避免程序粗暴的死掉。使用方法如下：

```
assert(thiz != NULL);
if(thiz == NULL)
{
    return DLIST_RET_INVALID_PARAMS;
}
```

我一直使用这种方法，但是有个问题：无法用自动测试验证 assert 是否正常的触发了，当用错误的参数测试时，我期望 assert 被触发，但如果 assert 被触发了，自动程序测试就死掉了，自动测试程序死掉了，就无法继续验证下一个 assert。这是一个悖论！

后来我从 glib 里面学了一招，它检查时不用 assert，只是打印出一个警告，代码也简明了，按它的方式，我们这样检查：

```
return_val_if_fail(cursor != NULL, DLIST_RET_INVALID_PARAMS);
```

我们需要定义两个宏，一个用于无返回值的函数，一个用于有返回值的函数：

```
#define return_if_fail(p) if(!(p)) \
    {printf("%s:%d Warning: "#p" failed.\n", \
        __func__, __LINE__); return;}
#define return_val_if_fail(p, ret) if(!(p)) \
    {printf("%s:%d Warning: "#p" failed.\n",\
__func__, __LINE__); return (ret);}
```

这样一来，遇到无效参数时，可以看到一个警告信息，同时又不会影响自动测试。

o 在测试时，用查询来验证命令。命令一般都有返回值，但只检查返回值是不够的。比如 dlist_delete 返回 OK，它真的 OK 了吗？我们信任它，但还是要检查。怎么检查？很简单，用查询函数来检查对象的状态是不是预期的。

对于 dlist_delete，我们预期：

```
1.输入无效参数，期望返回 DLIST_RET_INVALID_PARAMS。
2.输入正确参数，期望：
        函数返回 DLIST_RET_OK
        双向链表的长度减一。
        删除的位置的下一个元素被移到删除的位置。
```

在测试程序中检查时，因为任何不符合期望的结果都是 BUG，所以我们用 assert 检查。这样有问题马上暴露出来了，定位错误比较容易，通常都不需要调试器。我们这样来检查：

```
    assert(dlist_length(dlist) == (n-i));
    assert(dlist_delete(dlist, 0) == DLIST_RET_OK);
    assert(dlist_length(dlist) == (n-i-1));
    if((i + 1) < n)
    {
         assert(dlist_get_by_index(dlist, 0, (void**)&data) == DLIST_RET_OK);
         assert((int)data == (i+1));
    }
```

(完整的例子请看本节的示例代码)

o 在测试时，用基本查询去验证复合查询。基本查询和复合查询返回的应该一致。比如：

```
    Rect rect = {0};
    widget_get_rect(widget, &rect);
    assert(widget_get_width(widget) == rect.width);
    assert(widget_get_height(widget)== rect.height);
```

o 在测试时，预期结果依赖其执行上下文，我们要按逻辑组织测试用例。前面调用的函数可能改变了对象的状态，为了简化测试，在每组测试用例开始时，都重置对象到初始状态。

o 在测试时，第一次只写基本的测试用例，以后逐渐累积，每次发现新的 BUG 就把相应的测试用例加进去。每次修改了代码就运行一遍自动测试，保证修改没有引起其它副作用。

按着上面的原则，应付正常模块的测试没有问题了，但是下面的情况仍然比较棘手：

o 带有 GUI 的应用程序。有 GUI 的程序会给自动的输入数据和检查结果带来困难，有些工具可以部分解决这个问题，特别是针对 Win32 下的 GUI，我很少在 Windows 下写程序，所以对这方面了解不多。不过最好的办法还是用 MVC 模型等分离界面和实现，因为界面通常相对比较简单，可以手工测试，而实现的逻 辑比较复杂，这部分可以自动测试。后面我们会专门讲解分离界面和实现的方法。

o 有随机数据输入。如果有些输入数据是内部随机产生的，那你根本无法预测它的输出结果和影响。比如游戏随机的步骤和无线网络信号的变化。对于我们可以控制的 随机数据，可以提供额外的函数去获取这些数据。对于无法控制的随机输入数据，可以把它们隔离开，在自动测试中，使用固定的数据。

o 多线程运行的程序。多线程的程序也很难自动测试，比如向链表中插入一个元素，当你检查的时候，根本无法知道链表的长度是否增加，也无法知道刚才插入的位置 是否是你插入的元素，因为这个时候，可能有另外一个线程已经把它删除了，或者又加入了新的数据。不过在单线程的自动测试通过之后，多线程的问题会大大减 少，剩下的问题我们可以通过其它方式加以避免。

写自动测试程序会花你一些时间，但这个投资能带来最大的回报：减少后面调试时的浪费，提高代码的质量，更重要的是你可以安稳的睡个觉了。

本节示例请到**这里**下载。

Save your work

“Ernst 和 Young 所在的小组决定使用正规的开发理论—他们常用削减法，分阶段进行开发并具有中途交付能力。他们的步骤包括细致的分析和设计 —正如本章描写的基本原则一样。而其他竞争者径直开始了编码，在开始几个小时里，Ernst 和 Young 小组落后了。但到中午时 Ernst 和 Young 小 组却是遥遥领先了，而到了这一天的最后，他们却失败了。导致失败的原因不是因为他们的正规方法，而是他们偶然错误的把工作文件覆盖了，最终他们比午餐时所 做的估计少交付了一些功能，他们是被没有使用有效的源程序版本控制这个典型的错误给打败了。”

—摘自《快速软件开发》

前段时间看探索频道的《荒野求生秘技(Man & wild)》，我很喜欢这个节目也喜欢那个英国佬，甚至连重播都不会放过。他展示在沙漠、丛林、冰河和雪山等各种环境的求生秘技，他吃蜘蛛、白蚁、蝎子和 蜥蜴，边吃边说这东西很恶心，但是里面含有非常的维生素，蛋白质和糖份，能够 Save your life，所以要吃下去。

在 Man & Code 的世界里，环境好多了，不用面临危险，寻找水源和食物根本不需要什么秘诀。这里我们不需要求生秘技去 Save your life，但我们需要一些习惯去 Save your work。我说过作为一名高效的程序员，不是因为他打字比别人快，而是因为他省下了别人浪费的时间，有什么比成果被毁，从头再来更浪费时间呢？下面我介绍 一些习惯，它们简单有效，根本算不上什么秘技，但它们能够 Save your work，让你的工作稳步前进。

o 随时存盘

每次停电时，我都会听到有人惊呼，完了，我的代码没有保存！补回半小时或一个小时的工作不难，在一个好的工作环境里，这种情况一年也就会遇到几次， 浪费的时间完全可以忽略不计。但是那种感觉很难受，会影响你的工作情绪，无缘无故的让你重做你的工作，和因为要改进去重做完全是两回事。在我以前工作过的 一个公司，有段时间经常跳闸，每周都要停好几次，怎么也找不到原因，后来请人来查，据说是线路太长，静电引起的跳闸。经过那段时间的折磨，我养成了一个习 惯：写代码的时候，平均 30 秒钟存盘一次。现在遇到停电，别人惊呼的时候，我开始闭目养神了。

o 使用版本控制系统

和一些老程序员聊天时(呵，其实我也老了)，他们经常问起我们项目有没有使用版本控制系统，我说当然有了，大二的时候就我用 Sourcesafe 来 管理用 powerbuilder 写的代码了，后来的工作中一直在使用不同的版本控制系统。接着他们开始讲述他们惨痛的经历…这些经历小则让他们项目延期， 大则导致整个项目失败。

版本控制系统有很多功能，但对我个人来说，它最重要的功能是备份代码。每完成一个小功能，我都会把它提交(checkin)进去，如果我不小心删除 了本地文件，或者某个做尝试的修改失败了，我可以恢复代码到前一个版本。不同团队有不同的规则，有的团队是不允许这样 checkin 的，他们只允许 checkin 经过严格测试的代码。如果是那样，你可以在本地建立自己的版本控制系统，初学者在学习时也可以这样做。现在有很多免费的版本控制系统可用，像 CVS、SVN 和 GIT 等等，我个人习惯用 CVS，SVN 是 CVS 的改进版，将来肯定会替代 SVN 的，所以推荐大家使用它。

o 定期备份

温伯格在《Quality Software Management: System Thinking》讲了一个有趣的故事，他以前去研究一些失败的案例，发现这些项目的失败都是因为欠佳的运气引起的：比如遭受到洪水、地震、火灾和流行感 冒等灾害，项目主管们把自己描述成外部问题的受害者。他又对另外一些成功的项目进行研究，发现其中有些项目同样经历这些自然灾害，但是他们成功的完成了任 务。区别只是在于成功项目的主管，采用积极预防措施，定期备份代码，把它们放到不同的地点。

以前在学校的时候，我有两台电脑，一台赛扬和一台 486。我经常在上面重装系统，一会儿装 Linux，一会儿装 NT，一会儿又装 Netware。虽 然我经常把代码备份到不同的分区上，结果还不小心把所有分区全干掉了，让我痛心不已。那只是写的一些小程序，重写一遍问题也不大，但是对于专业程序员或一 个软件团队来说，重写整个代码就不能接受了，所以需要更可靠的备份机制。

使用源代码管理系统还不能保证代码的安全，比如服务器硬盘损坏和办公室发生火灾等都是可能发生。团队里一定要有人负责定期备份源代码管理系统系统上 的资料，作为初学者也应该有这种意识。另外，我发现有些朋友把重要的资料放在邮箱里，现在的邮箱容量很大，因为提供商会定期备份，非常安全，这倒是一个不 错的主意。

o 状态不好就做点别的

女同胞有定期状态不佳的时候，男同胞也不是每天状态都很好。感冒了、丢东西了、或者家

人争吵了，都会影响你的状态。状态不好的时候做事，往往是进一 步退两步，甚至犯下严重的错误。有次我得了重感冒，居然在服务器的根目录下运行 rm * -rf(删除全部文件)，由于删除的时间太长，才让我发现删错地方了，吓得我出了一身冷汗，还好那台服务器不是运行着源代码管理系统，但还是浪费了我两天 时间去重建服务器上的环境。

状态不好的时候编程也会犯一些低级错误，让你花费更多时间去调试。总要言之，状态不好的去做重要的事有害无益，这时你不防去做点别做的，比如看看其它模块的代码之类的，甚至完全放松去休息都比犯下严重的错误强。

这 几年并发技术受到前所未有的关注：CPU 进入多核时代，连手机芯片都使用三核的 CPU(AP+BP+DSP 集成到一颗芯片)了。天生具有并发能力的语言 ErLang 逐渐成为热点。网格和云计算开始进入实用阶段。还有一些新技术更是让我闻所未闻，初学者也不用被这些铺天盖地的名词吓倒。据笔者的经验来看， 这些技术或许能够改变产业的格局，对人类生活造成重大影响，但从实现角度来看并不无多少革命，相反大部分都是传统技术的改进和应用。这几年我一直在研究开 源的基础软件，实际上我没有发现多少“新”东西或者核心技术。要说眞正的核心还是如序言中说的：战胜复杂度和应对变化。

作为系统程序员，掌握基础理论和经典的设计方法，比去追逐一些所谓的新技术要实用得多，基础打扎实了，学习新知识也是很容易的事。在接下来几节中，我们一起来学习传统的并发编程知识。在这里我们请读者完成下列任务：

了解 linux 下的多线程编程的基本方法，以双向链表为载体实现传统的生产者-消费者模型：一个线程往双向链表中加东西，另外一个线程从这里面取。

Linux 下的多线程编程使用 pthread(POSIX Thread)函数库，使用时包含头文件 pthread.h，链接共享库 libpthread.so。这里顺便说一下 gcc 链接共享库的方式：-L 用来指 定共享库所在目录，系统库目录不用指定。-l 用来指定要链接的共享库，只需要指定库的名字就行了，如：-lpthread，而不是 -llibpthread.so。看起来有点怪，这样做的原因是共享库通常带有版本号，指定全文件名就意味着你要绑定到特定版本的共享库上，只指定名字则 在可以运行时通过环境变量来选择要使用的共享库，这样能够给软件升级带来的方便。

pthread 函数库的使用相对比较简单，读者可以在终端下运行 man pthread_create 阅读相关函数的手册，也可以到网上找些例子参考。具体使用方法我们就不讲了，这里介绍一下初学者常犯的错误：

o 用临时变量作为线程参数的问题。

```
#include <stdio.h>
#include <pthread.h>
#include <assert.h>
void* start_routine(void* param)
{
    char* str = (char*)param;
    printf("%s:%s\n", __func__, str);
    return NULL;
}
pthread_t create_test_thread()
```

```
{
    pthread_t id = 0;
    char str[] = "it is ok!";
    pthread_create(&id, NULL, start_routine, str);
    return id;
}
int main(int argc, char* argv[])
{
    void* ret = NULL;
    pthread_t id = create_test_thread();
    pthread_join(id, &ret);
    return 0;
}
```

分析：由于新线程和当前线程是并发的，谁先谁后是无法预测的。可能 create_test_thread 已经执行完了，str 已经被释放了，新线程才拿到这参数，此时它的内容已经无法确定了，打印出的字符串自然是随机的。

o 线程参数共享的问题。

```
#include <stdio.h>
#include <pthread.h>
#include <assert.h>
void* start_routine(void* param)
{
    int index = *(int*)param;
    printf("%s:%d\n", __func__, index);
    return NULL;
}
#define THREADS_NR 10
void create_test_threads()
{
    int i = 0;
    void* ret = NULL;
    pthread_t ids[THREADS_NR] = {0};
    for(i = 0; i < THREADS_NR; i++)
    {
        pthread_create(ids + i, NULL, start_routine, &i);
    }
    for(i = 0; i < THREADS_NR; i++)
    {
        pthread_join(ids[i], &ret);
    }
    return ;
}
int main(int argc, char* argv[])
{
    create_test_threads();
    return 0;
}
```

分析：由于新线程和当前线程是并发的，谁先谁后是无法预测的。i 在不断变化，所以新线程拿到的参数值是无法预知的，打印出的字符串自然也是随机的。

o 虚假并发。

```
#include <stdio.h>
#include <pthread.h>
#include <assert.h>
void* start_routine(void* param)
{
    int index = *(int*)param;
    printf("%s:%d\n", __func__, index);
    return NULL;
}
#define THREADS_NR 10
void create_test_threads()
{
    int i = 0;
    void* ret = NULL;
    pthread_t ids[THREADS_NR] = {0};
    for(i = 0; i < THREADS_NR; i++)
    {
        pthread_create(ids + i, NULL, start_routine, &i);
        pthread_join(ids[i], &ret);
    }
    return ;
}
int main(int argc, char* argv[])
{
    create_test_threads();
    return 0;
}
```

分析：因为 pthread_join 会阻塞直到线程退出，所以这些线程实际上是串行执行的，一个退出了，才创建下一个。当年一个同事写了一个多线程的测试程序，就是这样写的，结果没有测试出一个潜伏的问题，直到产品运行时，这个问题才暴露出来。

访问线程共享的数据时要加锁，让访问串行化，否则就会出问题。比如，可能你正在访问的双向链表的某个结点时，它已经被另外一个线程删掉了。加锁的方 式有很多种，像互斥锁(mutex= mutual exclusive lock)，信号量(semaphore)和自旋锁(spin lock)等都是常用的，它们的使用同样很简单，我们就不多说了。

在加锁/解锁时，初学者常犯两个错误：

o 存在遗漏解锁的路径。初学者常见的做法就是，进入某个临界函数时加锁，在函数结尾的地方解锁，我甚至见过这种写法：

```
{
/*这里加锁*/
…
    return …;
```

```
/*这里解锁*/
}
```

如果你也犯了这种错误，应该好好反思一下。有时候，return 的地方太多，在某一处忘记解锁是可能的，就像内存泄露一样，只是忘记解锁的后果更严重。像下面这个例子：

```
Ret dlist_insert(DList* thiz, size_t index, void* data)
{
    DListNode* node = NULL;
    DListNode* cursor = NULL;
    return_val_if_fail(thiz != NULL, RET_INVALID_PARAMS);
    dlist_lock(thiz);
    if((node = dlist_create_node(thiz, data)) == NULL)
    {
        dlist_unlock(thiz);
        return RET_OOM;
    }
    if(thiz->first == NULL)
    {
        thiz->first = node;
        dlist_unlock(thiz);
        return RET_OK;
    }
    ...
    dlist_unlock(thiz);
    return RET_OK;
}
```

如果一个函数有五六个甚至更多的地方返回，遗忘一两个地方是很常见的，即使没有忘记，每个返回的地方都要去解锁和释放相关资源也是很麻烦的。在这种情况下，我们最好是实现单入口单出的函数，常见的做法有两种：

一种是使用 goto 语句（在 linux 内核里大量使用）。示例如下：

```
Ret dlist_insert(DList* thiz, size_t index, void* data)
{
    Ret ret = RET_OK;
    DListNode* node = NULL;
    DListNode* cursor = NULL;
    return_val_if_fail(thiz != NULL, RET_INVALID_PARAMS);
    dlist_lock(thiz);
    if((node = dlist_create_node(thiz, data)) == NULL)
    {
        ret = RET_OOM;
        goto done;
    }
    if(thiz->first == NULL)
    {
        thiz->first = node;
        goto done;
    }
```

```
    ...
done:
    dlist_unlock(thiz);
    return ret;
}
```

另外一种是使用 do{}while(0);语句，出于受教科书的影响(不要用 goto 语句)，我习惯了这种做法。示例如下：

```
Ret dlist_insert(DList* thiz, size_t index, void* data)
{
    Ret ret = RET_OK;
    DListNode* node = NULL;
    DListNode* cursor = NULL;
    return_val_if_fail(thiz != NULL, RET_INVALID_PARAMS);
    dlist_lock(thiz);
    do
    {
        if((node = dlist_create_node(thiz, data)) == NULL)
        {
            ret = RET_OOM;
            break;
        }
        if(thiz->first == NULL)
        {
            thiz->first = node;
            break;
        }
        ...
    }while(0);
    dlist_unlock(thiz);
    return ret;
}
```

o 加锁顺序的问题。有时候为了提高效率，常常降低加锁的粒度，访问时不是用一个锁锁住整个数据结构，而是用多个锁来控制数据结构各个部分。这样一个线程访问 数据结构的这部分时，另外一个线程还可以访问数据结构的其它部分。但是在有的情况下，你需要同时锁几个锁，这时就要注意了：所有线程一定要按相同的顺序加 锁，相反的顺序解锁。否则就可能出现死锁，两个线程都拿到对方需要的锁，结果出现互相等待的情况。

在 生产者-消费者的练习中，大部分人选择了由调用者来加锁：作为生产者，往双向链表里插入数据时，先加锁，插入数据，然后解锁。作为消费者，从双向链表里取 数据时，先加锁，删除数据，然后解锁。这是合理的，不过有点麻烦：每个调用者都要做这些动作，如果其中一个调用者忘记了解锁的步骤，就会造成死锁。而且调 用者必须要清楚自己是在多线程下工作，这些代码放到单线程的环境中就不能使用了。

在很多情况下由实现者来加锁是比较好的选择，那样对调用者更为友好，可以避免出现一些不必要的错误。比如像目前 Linux 下流行的 DBUS，它是一 套进程间通信框架，它支持单线程和多线程版本，但调用者不需要明确加锁/解锁，也不需要连接不同的库或者用宏来控

制，单线程版本和多线程版本的不同只是在 一个初始化函数上。

这里我们请读者对前面实现的双向链表做点改进：

o 支持多线程和单线程版本。对于多线程版本，由实现者(在链表)加锁/解锁，对于单线程版本，其性能不受影响(或很小)。
o 区分单线程版本和多线程版本时，不需要链接不同的库，或者要宏来控制，完全可以在运行时切换。
o 保持双向链表的通用性，不依赖于特定的平台。

面 对这个需求，一些初学者可能有点蒙了。以前在学校的时候，对于课本后面的练习，我总是信心百倍，原因很简单，我确信这些练习不管它的出现方式有多么不同， 但总是与前面学过的知识有关。记得《如何求解问题—现代启发式方法》中说过，正是这种练习的方式妨碍了我们解决问题的能力，在现实中解决问题时通常没有这 么幸运。在《系统程序员成长计划》我把练习放前面，目标就是刺激读者去思考，在学习知识的同时学习解决问题的方法。

这里我们应该怎么分析呢？要在双向链表里加锁，第一是要区分单线程和多线程，要链接同一个库，而且不能用宏来控制。第二是不能依赖于特定平台，而锁 本身恰恰又是依赖于平台的。怎么办？很明显这两个需求都要求锁的实现可以变化的：单线程版本它什么都不做，多线程版本中，不同的平台有不同的实现。

我们要做的就是隔离变化。变化怎么隔离？前面我们已经练习过几次用回调函数来隔离变化了，所有的读者都会想到这个方法，因为锁无非是具有两个功能：加锁和解锁，我们把它抽象成两个回调函数就行了。

这种方法是可行的。这里的情况与前面相比有点特殊：前面的回调函数都是些独立功能的函数，每个回调函数都有自己的上下文，而这里的多个回调函数具有 相关的功能，并且共享同一个上下文(锁)。其次是这里的上下文(锁)是一个对象，有自己的生命周期，完成自己的使命后就应该被销毁。

这里我们引入接口(interface)这个术语，接口其实就是一个抽象的概念，它只定义调用者和实现者之间的契约，而不规定实现的方法。比如这里 的锁就是一个抽象的概念，它有加锁/解锁两个功能，这是调用者和实现者之间的契约。但光有这个概念不能做任何事情，只有具体的锁才能被使用。至于具体的 锁，不同的平台有不同的实现，但调用者不用关心。正因为调用者不用关心接口的实现方法，接口成了隔离变化最有力的武器。

在这里，锁是一个接口，双向链表是锁的调用者，有基于不同方式实现的锁。通过接口，双向链表把锁的变化隔离开来：区分单线程和多线程，隔离平台相关性。在C语言中，接口的朴素定义是：一组相关的回调函数及其共享的上下文。我们看看锁这个接口怎么定义：

```
struct _Locker;
typedef struct _Locker Locker;
typedef Ret  (*LockerLockFunc)(Locker* thiz);
typedef Ret  (*LockerUnlockFunc)(Locker* thiz);
typedef void (*LockerDestroyFunc)(Locker* thiz);
struct _Locker
{
    LockerLockFunc    lock;
```

```
    LockerUnlockFunc  unlock;
    LockerDestroyFunc destroy;
    char priv[0];
};
```

这里要注意三个问题：

o 接口一定要足够抽象，不能依赖任何具体实现的数据类型。接口一旦与某个具体实现关联了，另外一种实现就会遇到麻烦。比如这里你使用了 pthread_mutex_t，那你要实现一个 win32 下的锁怎么办呢。

o 接口不能有 create 函数，但一定要有 destroy 函数。我们说过对象有自己的生命周期，创建它，使用它，然后销毁它。但接口只是一个概念，不可能通 过这个概念凭空创建一个对象出来，对象只能通过具体实现来创建，所以接口不应该出现 create 自己的函数。一旦对象被创建出来，使用者应该在不再需要它 时销毁它，在销毁对象时，如果还要知道它的实现方式才能销毁它，那就造成了调用者和实现者之间不必要的耦合，因此接口都要提供一个 destroy 函数，调 用者可以直接销毁它。

o 这里的 priv 用来存放上下文信息，也就是具体实现需要用到的数据结构。像前面的回调函数一样，我们可以用一个 void* ctx 的成员来保存上下文信息。我们使用的 char priv[0];技巧，有点额外的好处：只需要一次内存分配，而且可以分配刚好够用的长度(0 到任意长度)。

前面我们使用回调函数，调用时要判断回调函数是否为空，每个地方都要重复这个动作，所以我们把这些判断集中起来好了：

```
static inline Ret locker_lock(Locker* thiz)
{
    return_val_if_fail(thiz != NULL && thiz->lock != NULL, RET_INVALID_PARAMS);
    return thiz->lock(thiz);
}
static inline Ret locker_unlock(Locker* thiz)
{
    return_val_if_fail(thiz != NULL && thiz->unlock != NULL, RET_INVALID_PARAMS);
    return thiz->unlock(thiz);
}
static inline void locker_destroy(Locker* thiz)
{
    return_if_fail(thiz != NULL && thiz->destroy != NULL);
    thiz->destroy(thiz);
    return;
}
```

下面我们来看看基于 pthread_mutex 的实现：

o 在 locker_pthread.h 中，提供一个创建函数。

```
Locker* locker_pthread_create(void);
```

o 在 locker_pthread.c 中，实现这些回调函数：

定义私有数据结构：

```
typedef struct _PrivInfo
{
    pthread_mutex_t mutex;
}PrivInfo;
```

创建对象：

```
Locker* locker_pthread_create(void)
{
    Locker* thiz = (Locker*)malloc(sizeof(Locker) + sizeof(PrivInfo));
    if(thiz != NULL)
    {
        PrivInfo* priv = (PrivInfo*)thiz->priv;
        thiz->lock    = locker_pthread_lock;
        thiz->unlock  = locker_pthread_unlock;
        thiz->destroy = locker_pthread_destroy;
        pthread_mutex_init(&(priv->mutex), NULL);
    }
    return thiz;
}
```

实现几个回调函数：

```
static Ret  locker_pthread_lock(Locker* thiz)
{
    PrivInfo* priv = (PrivInfo*)thiz->priv;
    int ret = pthread_mutex_lock(&priv->mutex);
    return ret == 0 ? RET_OK : RET_FAIL;
}
...
```

我简单说一下里面几个问题：

o malloc(sizeof(Locker) + sizeof(PrivInfo)); 前面的 char priv[0]并不占空间，这是 C 语言新标准定义的，用于实现变长的 buffer，它在这里的长度由 sizeof(PrivInfo)决定。

o PrivInfo* priv = (PrivInfo*)thiz->priv; 这里的 thiz->priv 只是一个定位符，实际上等于 (size_t)thiz+sizeof(Locker)，帮我们定位到私有数据的内存地址上。

使用方法：

单线程版本：

```
DList* dlist = dlist_create(NULL, NULL, locker_pthread_create());
```

多线程版本：

```
DList* dlist = dlist_create(NULL, NULL, NULL);
```

接口在软件设计中占有非常重要的地位，它是隔离变化和降低复杂度最有力的武器，差不多所有的设计模式都与接口有关。后面我们会反复的练习，这里请读者仔细体会一下。

本节示例代码请到**这里**下载。

嵌套锁与装饰模式

在生产者-消费者的练习中，当由双向链表的实现者负责加锁时，一般都会遇到莫名其妙的死锁问题。有的读者可能已经查出来了原因是嵌套的加锁。比如在 dlist_insert 中调用了 dlist_length，进入 dlist_insert 时已经加了一次锁，再调用 dlist_length 时又加了一 次锁，这时就出现了死锁问题。

初学者遇到这个问题的时候，通常的做法是在调用 dlist_length 之前先解锁，调用完 dlist_length 后再重新加锁。这样是存在问题的：一个原子操作变成了几个原子操作，数据完整性得不到保证，在你重新加锁之前，其它线程可能利用这个空隙做了些别的事情。

有效解决这个问题的办法有两个，其一是实现一个内部版本的 dlist_length，它在里面不加锁。其二是使用嵌套锁，允许同一个线程多次加锁。 pthread 有嵌套锁的实现，不过我们在这里不用它，原因是我们要提供一个更通用的解决方案。现在我们不再满足于实现一个双向链表，而是要实现一个跨平 台的基础函数库。

在这里我们请读者实现一个嵌套锁，要求如下：

o 嵌套锁仍然兼容 Locker 接口。
o 嵌套锁的实现不依赖于特定平台。

嵌套锁与装饰模式

嵌套锁的实现算法

加锁：

o 如果没有任何线程加锁，就直接加锁，并且记录下当前线程的 ID。
o 如果是当前线程加过锁了，就不用加锁了，只是将加锁的计数增加一。
o 如果其它线程加锁了，那就等待直到加锁成功，后继步骤与第一种情况相同。

解锁：

o 如果不是当前线程加的锁或者没有人加锁，那这是错误的调用，直接返回。
o 如果是当前线程加锁了，将加锁的计数减一。如果计数大于 0，说明当前线程加了多次，直接返回就行了。如果计数为 0，说明当前线程只加了一次，则执行解锁动作。

这个逻辑很简单，要做到兼容 Locker 的接口和平台无关，我们还需要引入装饰模式这个概念。装饰模式的功能是在于不改变对象的本质(接口)的前提 下，给对象添加附加的功能。和继承不同的是，它不是针对整个类的，而只是针对单个对象的。装饰这个名字非常直观的表现它的意义：在你自己的显示器上做点了 装饰，比如贴上一张卡通画：第一是它没有改显示器的本质，显示器还是显示器。第二是只有你自己的显示器上多了张卡通画，其它显示器没有影响。

这里我们要对一把锁进行装饰，不改变它的接口，但给它加上嵌套的功能。下面我们看看在 C 语言里的实现方法：

o 创建函数的原型。由于获取当前线程 ID 的函数是平台相关的，我们要用回调函数来抽象

它。

typedef int (*TaskSelfFunc)(void);
Locker* locker_nest_create(Locker* real_locker, TaskSelfFunc task_self);

这里可以看出：传入的是一把锁，返回的还是一把锁，没有改变的接口，但是返回的锁已经具有嵌套调用的功能了。

o 嵌入锁的实现。

私有信息：拥有锁的线程 ID、加锁的计数，被装饰的锁和获取当前线程 ID 的回调函数。

```
typedef struct _PrivInfo
{
    int owner;
    int refcount;
    Locker* real_locker;
    TaskSelfFunc task_self;
}PrivInfo;
```

1.实现加锁函数：如果当前线程已经加锁，只是增加加锁计数，否则就加锁。

```
static Ret  locker_nest_lock(Locker* thiz)
{
    Ret ret = RET_OK;
    PrivInfo* priv = (PrivInfo*)thiz->priv;
    if(priv->owner == priv->task_self())
    {
        priv->refcount++;
    }
    else
    {
        if( (ret = locker_lock(priv->real_locker)) == RET_OK)
        {
            priv->refcount = 1;
            priv->owner = priv->task_self();
        }
    }
    return ret;
}
```

2.实现解锁函数：只有当前线程加的锁才能解锁，先减少加锁计数，计数为 0 时才眞正解锁，否则直接返回。

```
static Ret  locker_nest_unlock(Locker* thiz)
{
    Ret ret = RET_OK;
    PrivInfo* priv = (PrivInfo*)thiz->priv;
    return_val_if_fail(priv->owner == priv->task_self(), RET_FAIL);
    priv->refcount--;
    if(priv->refcount == 0)
    {
        priv->owner = 0;
```

```
        ret = locker_unlock(priv->real_locker);
    }
    return ret;
}
```

o 使用方法。除了创建方法稍有不同外，调用方法完全一样。

```
Locker* locker = locker_pthread_create();
Locker* nest_locker = locker_nest_create(locker, (TaskSelfFunc)pthread_self);
DList*  dlist = dlist_create(NULL, NULL, nest_locker);
```

装饰模式最有用的地方在于，它给单个对象增加功能，但不是影响调用者，即使加了多级装饰，调用者也不用关心。

本节示例代码请到**这里**下载。

读写锁

在前面的实现中，像 dlist_length 这类的查询函数也要加锁，那样才能保证在查询过程中对象的状态不会被其它线程所改变。加锁阻止了其它线 程修改对象，也阻止其它线程查询对象。如果大多数情况下，线程只是查询对象的状态而不修改它，这种设计不是一种高效的方法，因为它不允许多个线程同时查 询。我们能不能实现一种锁，它能串行化对数据结构的修改，而同时支持并行的查询呢？

这就是所谓的读写锁，也称为共享-互斥锁。这种锁在数据库管理系统中(DBMS)和操作系统内核中大量应用，作为系统程序员，了解它的实现机制是有必要的。这里我们请读者实现读写锁，要求如下：

o 不依赖于特定平台。
o 在任何情况下都不带来额外的性能开销。

记住多想多练不要偷懒，学习知识点不是我们最重要的目标，知识点能帮你解决别人解决过的问题，但对你解决新问题未必有多大好处，真正的程序员不应当 只是解决问题方案的贩卖者。不断从思考中学习解决问题的方法，加上灵活应用已经掌握的知识点，你的设计水平才会大大提高，这也是《系统程序员成长计划》努 力的目标。

读写锁

读写锁在加锁时，要区分是为了读而加锁还是为了写而加锁，所以和递归锁不同的是，它无法兼容 Locker 接口了。不过为了做到不依 赖于特定平台，我们可以利用 Locker 的接口来抽象锁的实现。利用现有的锁来实现读写锁。读写锁的可变的部分已经被 Locker 隔离了，所以读写锁本身 不需要做成接口。它只是一个普通对象而已：

```
struct _RwLocker;
typedef struct _RwLocker RwLocker;

RwLocker* rw_locker_create(Locker* rw_locker, Locker* rd_locker);

Ret rw_locker_wrlock(RwLocker* thiz);
Ret rw_locker_rdlock(RwLocker* thiz);
```

```
Ret rw_locker_unlock(RwLocker* thiz);

void rw_locker_destroy(RwLocker* thiz);
```

o 创建读写锁

```
RwLocker* rw_locker_create(Locker* rw_locker, Locker* rd_locker)
{
   RwLocker* thiz = NULL;
   return_val_if_fail(rw_locker != NULL && rd_locker != NULL, NULL);

   thiz = (RwLocker*)malloc(sizeof(RwLocker));
   if(thiz != NULL)
   {
      thiz->readers = 0;
      thiz->mode = RW_LOCKER_NONE;
      thiz->rw_locker = rw_locker;
      thiz->rd_locker = rd_locker;
   }

   return thiz;
}
```

读 写锁的基本要求是：写的时候不允许任何其它线程读或者写，读的时候允许其它线程读，但不允许其它线程写。所以在实现时，写的时候一定要加锁，第一个读的线 程要加锁，后面其它线程读时，只是增加锁的引用计数。我们需要两个锁：一个锁用来保存被保护的对象，一个锁用来保护引用计数。

o 加写锁

```
Ret rw_locker_wrlock(RwLocker* thiz)
{
   Ret ret = RET_OK;
   return_val_if_fail(thiz != NULL, RET_INVALID_PARAMS);

   if((ret = locker_lock(thiz->rw_locker)) == RET_OK)
   {
      thiz->mode = RW_LOCKER_WR;
   }

   return ret;
}
```

加写锁很简单，直接加保护受保护对象的锁，然后修改锁的状态为已加写锁。后面其它的线程想写，就会这个锁上等待，如果想读也要等待(见后面)。

o 加读锁

```
Ret rw_locker_rdlock(RwLocker* thiz)
{
```

```
    Ret ret = RET_OK;
    return_val_if_fail(thiz != NULL, RET_INVALID_PARAMS);

    if((ret = locker_lock(thiz->rd_locker)) == RET_OK)
    {
        thiz->readers++;
        if(thiz->readers == 1)
        {
            ret = locker_lock(thiz->rw_locker);
            thiz->mode = RW_LOCKER_RD;
        }
        locker_unlock(thiz->rd_locker);
    }

    return ret;
}
```

先尝试加保护引用计数的锁，增加引用计数。如果当前线程是第一个读，就要去加保护受保护对象的锁。如果此时已经有线程在写，就等待直到加锁成功，然后把锁的状态设置为已加读锁，最后解开保护引用计数的锁。

o 解锁

```
Ret rw_locker_unlock(RwLocker* thiz)
{
    Ret ret = RET_OK;
    return_val_if_fail(thiz != NULL, RET_INVALID_PARAMS);

    if(thiz->mode == RW_LOCKER_WR)
    {
        thiz->mode == RW_LOCKER_NONE;
        ret = locker_unlock(thiz->rw_locker);
    }
    else
    {
        assert(thiz->mode == RW_LOCKER_RD);
        if((ret = locker_lock(thiz->rd_locker)) == RET_OK)
        {
            thiz->readers--;
            if(thiz->readers == 0)
            {
                thiz->mode == RW_LOCKER_NONE;
                ret = locker_unlock(thiz->rw_locker);
            }
            locker_unlock(thiz->rd_locker);
        }
    }

    return ret;
}
```

解锁时根据状态来决定，解写读直接解保护受保护对象的锁。解读锁时，先要加锁保护引用计数的锁，引用计数减一。如果自己是最后一个读，才解保护受保护对象的锁，最后解开保护引用计数的锁。

从上面读写锁的实现，我们可以看出，读写锁要充分发挥作用，就要基于两个假设：

o 读写的不对称性，读的次数远远大于写的次数。像数据库就是这样，决大部分时间是在查询，而修改的情况相对少得多，所以数据库通常使用读写锁。

o 处于临界区的时间比较长。从上面的实现来看，读写锁实际上比正常加/解锁的次数反而要多，如果处于临界区的时间比较短，比如和修改引用计数差不多，使用读写锁，即使全部是读，它的效率也会低于正常锁。

本节示例请到**这里**下载。

无锁(lock-free)数据结构

提到并行计算通常都会想到加锁，事实却并非如此，大多数并发是不需要加锁的。比如在不同电脑上运行的代码编辑器，两者并发运行不需要加锁。在一台电 脑上同时运行的媒体播放放器和代码编辑器，两者并发运行不需要加锁(当然系统调用和进程调度是要加锁的)。在同一个进程中运行多个线程，如果各自处理独立 的事情也不需要加锁(当然系统调用、进程调度和内存分配是要加锁的)。在以上这些情况里，各个并发实体之间没有共享数据，所以虽然并发运行但不需要加锁。

多线程并发运行时，虽然有共享数据，如果所有线程只是读取共享数据而不修改它，也是不用加锁的，比如代码段就是共享的“数据”，每个线程都会读取， 但是不用加锁。排除所有这些情况，多线程之间有共享数据，有的线程要修改这些共享数据，有的线程要读取这些共享数据，这才是程序员需要关注的情况，也是本 节我们讨论的范围。

在并发的环境里，加锁可以保护共享的数据，但是加锁也会存在一些问题：

o 由于临界区无法并发运行，进入临界区就需要等待，加锁带来效率的降低。

o 在复杂的情况下，很容易造成死锁，并发实体之间无止境的互相等待。

o 在中断/信号处理函数中不能加锁，给并发处理带来困难。

o 优先级倒置造成实时系统不能正常工作。低级优先进程拿到高优先级进程需要的锁，结果是高/低优先级的进程都无法运行，中等优先级的进程可能在狂跑。

由于并发与加锁(互斥)的矛盾关系，无锁数据结构自然成为程序员关注的焦点，这也是本节要介绍的：

o CPU 提供的原子操作。

大约在七八年前，我们用 apache 的 xerces 来解析 XML 文件，奇怪的是多线程反而比单线程慢。他们找了很久也没有找出原因，只是证实使用多 进程代替多线程会快一个数量级，在 Windows 上他们就使用了多进程的方式。后来移植到 linux 时候，我发现 xerces 每创建一个

结点都会去更新一 些全局的统计信息，比如把结点的总数加一，它使用的 pthread_mutex 实现互斥。这就是问题所在：一个 XML 文档有数以万计的结点，以 50 个线程 并发为例，每个线程解析一个 XML 文档，总共要进行上百万次的加锁/解锁，几乎所有线程都在等待，你说能快得了吗？

当时我知道 Windows 下有 InterlockedIncrement 之类的函数，它们利用 CPU 一些特殊指令，保证对整数的基本操作是原子的。 查找了一些资源发现 Linux 下也有类似的函数，后来我把所有加锁去掉，换成这些原子操作，速度比多进程运行还快了几倍。下面我们看++和—的原子操作在 IA 架构上的实现：

```
#define ATOMIC_SMP_LOCK "lock ; "
typedef struct { volatile int counter; } atomic_t;
static __inline__ void atomic_inc(atomic_t *v)
{
    __asm__ __volatile__(
        ATOMIC_SMP_LOCK "inc1 %0"
        :"=m" (v->counter)
        :"m" (v->counter));
}
static __inline__ void atomic_dec(atomic_t *v)
{
    __asm__ __volatile__(
        ATOMIC_SMP_LOCK "dec1 %0"
        :"=m" (v->counter)
        :"m" (v->counter));
}
```

o 单入单出的循环队列。单入单出的循环队列是一种特殊情况，虽然特殊但是很实用，重要的是它不需要加锁。这里的单入是指只有一个线程向队列里追加数据 (push)，单出只是指只有一个线程从队列里取数据(pop)，循环队列与普通队列相比，不同之处在于它的最大数据储存量是事先固定好的，不能动态增 长。尽管有这些限制它的应用还是相当广泛的。这我们介绍一下它的实现：

数据下定义如下：

```
typedef struct _FifoRing
{
    int r_cursor;
    int w_cursor;
    size_t length;
    void* data[0];
}FifoRing;
```

r_cursor 指向队列头，用于取数据(pop)。w_cursor 指向队列尾，用于追加数据(push)。length 表示队列的最大数据储存量，data 表示存放的数据，[0]在这里表示变长的缓冲区(前面我们已经讲过)。

创建函数

```
FifoRing* fifo_ring_create(size_t length)
{
    FifoRing* thiz = NULL;
    return_val_if_fail(length > 1, NULL);
    thiz = (FifoRing*)malloc(sizeof(FifoRing) + length * sizeof(void*));
    if(thiz != NULL)
    {
        thiz->r_cursor = 0;
        thiz->w_cursor = 0;
        thiz->length   = length;
    }
    return thiz;
}
```

这里我们要求队列的长度大于1而不是大于0，为什么呢？排除长度为1的队列没有什么意义的原因外，更重要的原因是队列头与队列尾重叠 (r_cursor= =w_cursor) 时，到底表示是满队列还是空队列？这个要搞清楚才行，上次一个同事犯了这个错误，让我们查了很久。这里我们认为队列头与队列尾重叠时表示队列为空，这与队 列初始状态一致，后面在写的时候始终保留一个空位，避免队列头与队列尾重叠，这样可以消除歧义了。

追加数据(push)

```
Ret fifo_ring_push(FifoRing* thiz, void* data)
{
    int w_cursor = 0;
    Ret ret = RET_FAIL;
    return_val_if_fail(thiz != NULL, RET_FAIL);
    w_cursor = (thiz->w_cursor + 1) % thiz->length;
    if(w_cursor != thiz->r_cursor)
    {
        thiz->data[thiz->w_cursor] = data;
        thiz->w_cursor = w_cursor;
        ret = RET_OK;
    }
    return ret;
}
```

队列头和队列尾之间还有一个以上的空位时就追加数据，否则返回失败。

取数据(pop)

```
Ret fifo_ring_pop(FifoRing* thiz, void** data)
{
    Ret ret = RET_FAIL;
    return_val_if_fail(thiz != NULL && data != NULL, RET_FAIL);
    if(thiz->r_cursor != thiz->w_cursor)
    {
        *data = thiz->data[thiz->r_cursor];
        thiz->r_cursor = (thiz->r_cursor + 1)%thiz->length;
        ret = RET_OK;
    }
```

```
    return ret;
}
```

队列头和队列尾不重叠表示队列不为空，取数据并移动队列头。

o 单写多读的无锁数据结构。单写表示只有一个线程去修改共享数据结构，多读表示有多个线程去读取共享数据结构。前面介绍的读写锁可以有效的解决这个问题，但更高效的办法是使用无锁数据结构。思路如下：

就像为了避免显示闪烁而使用的双缓冲一样，我们使用两份数据结构，一份数据结构用于读取，所有线程都可以在不加锁的情况下读取这个数据结构。另外一份数据结构用于修改，由于只有一个线程会修改它，所以也不用加锁。

在修改之后，我们再交换读/写的两个函数结构，把另外一份也修改过来，这样两个数据结构就一致了。在交换时要保证没有线程在读取，所以我们还需要一个读线程的引用计数。现在我们看看怎么把前面写的双向链表改为单写多读的无锁数据结构。

为了保证交换是原子的，我们需要一个新的原子操作 CAS(compare and swap)。

```
#define CAS(_a, _o, _n)                                     \
({ __typeof__(_o) __o = _o;                                 \
   __asm__ __volatile__(                                    \
       "lock cmpxchg %3,%1"                                 \
       : "=a" (__o), "=m" (*(volatile unsigned int *)(_a)) \
       :  "0" (__o), "r" (_n) );                            \
   __o;                                                     \
})
```

数据结构

```
typedef struct _SwmrDList
{
    atomic_t rd_index_and_ref;
    DList* dlists[2];
}SwmrDList;
```

两个链表，一个用于读一个用于写。rd_index_and_ref 的最高字节记录用于读取的双向链表的索引，低 24 位用于记录读取线程的引用记数，最大支持 16777216 个线程同时读取，应该是足够了，所以后面不考虑它的溢出。

读取操作

```
int      swmr_dlist_find(SwmrDList* thiz, DListDataCompareFunc cmp, void* ctx)
{
    int ret = 0;
    return_val_if_fail(thiz != NULL && thiz->dlists != NULL, -1);
    atomic_inc(&(thiz->rd_index_and_ref));
    size_t rd_index = (thiz->rd_index_and_ref.counter>>24) & 0x1;
    ret = dlist_find(thiz->dlists[rd_index], cmp, ctx);
    atomic_dec(&(thiz->rd_index_and_ref));
    return ret;
```

```
}
```

修改操作

```
Ret swmr_dlist_insert(SwmrDList* thiz, size_t index, void* data)
{
    Ret ret = RET_FAIL;
    DList* wr_dlist = NULL;
    return_val_if_fail(thiz != NULL && thiz->dlists != NULL, ret);
    size_t wr_index = !((thiz->rd_index_and_ref.counter>>24) & 0x1);
    if((ret = dlist_insert(thiz->dlists[wr_index], index, data)) == RET_OK)
    {
        int rd_index_old = thiz->rd_index_and_ref.counter & 0xFF000000;
        int rd_index_new = wr_index << 24;
        do
        {
            usleep(100);
        }while(CAS(&(thiz->rd_index_and_ref), rd_index_old, rd_index_new));
        wr_index = rd_index_old>>24;
        ret = dlist_insert(thiz->dlists[wr_index], index, data);
    }
    return ret;
}
```

先修改用于修改的双向链表，修改完成之后等到没有线程读取时，交换读/写两个链表，再修改另一个链表，此时两个链表状态保持一致。

稍做改进，对修改的操作进行加锁，就可以支持多读多写的数据结构，读是无锁的，写是加锁的。

o 真正的无锁数据结构。Andrei Alexandrescu 的《Lock-FreeDataStructures》估计是这方面最经典的论文了，对他的方法我开始感到惊奇后来感到失望，惊 奇的是算法的巧妙，失望的是无锁的限制和代价。作者最后说这种数据结构只适用于 WRRMBNTM(Write-Rarely-Read-Many -But-Not-Too-Many)的情况。而且每次修改都要拷贝整个数据结构（甚至多次），所以不要指望这种方法能带来多少性能上的提高，唯一的好处 是能避免加锁带来的部分副作用。有兴趣的朋友可以看下这篇论文，这里我就不重复了。

本节示例请到**这里**下载。

双 向链表和动态数组是最简单的两种数据结构，在研读大量源代码之后，我发现正是这两种最简单的数据结构有着最广泛的应用。像平衡二叉树这样的复杂数据结构， 除了学习之外，你永远都不会写第二遍，甚至连使用的机会都没有，而双向链表和动态数组则会在开发中反复的运用。也正是由于这个原因，我们才选择双向链表和 动态数组作为学习的载体。

这里请读者做两件事：

o 思考双向链表和动态数组的特点及适用条件。

在学习完基本的数据结构和设计方法后，很多人在解决实际问题时束手无策：他们不知道什

么时候应该用什么数据结构和方法。所以在学习的过程中，应该经常问这个问题，这个东西在什么时候能派上用场？千万不要为了学习而学习，而是要为实际应用而学习。

o 按前面在双向链表中学习到的方法实现动态数组。

前面我们刚学习了并发编程的基础知识，由于我们现在的主要目的是学习，为了专注于我们要学习的内容，后面不再考虑并发问题。

(要回家过年了，年前 BLOG 不再更新，预祝大家春节快乐，谢谢大家对我的支持和鼓励)

对比双向链表和动态数组

在 C 语言中，数组的长度是事先确定的，不能在运行时动态调整。所谓动态数组就是它的长度可以根据存储数据多少自动调整，这需要我们用程序来实现。对比双向链表和动态数组，我们会发现：

o 动态数组本身占用一块连续的内存，而双向链表的每个结点要占用一块内存。在频繁增删数据的情况下，双向链表更容易造成内存碎片，具体影响与内存管理器的好坏有关。

o 动态数组的增删操作需要移动后面的元素，而双向链表只需要修改前后元素的指针。在存储大量数据的情况下，这种差异更为明显。

o 动态数组支持多种高效的排序算法，像快速排序、归并排序和堆排序等等，而这些算法在双向链表中的表现并不好，甚至不如冒泡排序来得快。

o 排序好的动态数组可以使用二分查找，而排序好的双向链表仍然只能使用顺序查找。主要原因是双向链表不支持随机定位，定位中间结点时只能一个一个的移动指针。

o 对于小量数据，使用动态数组还是双向链表没有多大区别，使用哪个只看个人的喜好了。

实现动态数组

在考虑存值还是指针时，我们同样选择存指针，所以这里我们实现的是指针数组。动态数组的功能和双向链表非常类似，所以它对外的接口也是类似的：

```
struct _DArray;
typedef struct _DArray DArray;
DArray* darray_create(DataDestroyFunc data_destroy, void* ctx);
Ret    darray_insert(DArray* thiz, size_t index, void* data);
Ret    darray_prepend(DArray* thiz, void* data);
Ret    darray_append(DArray* thiz, void* data);
Ret    darray_delete(DArray* thiz, size_t index);
Ret    darray_get_by_index(DArray* thiz, size_t index, void** data);
Ret    darray_set_by_index(DArray* thiz, size_t index, void* data);
size_t  darray_length(DArray* thiz);
int    darray_find(DArray* thiz, DataCompareFunc cmp, void* ctx);
Ret    darray_foreach(DArray* thiz, DataVisitFunc visit, void* ctx);
void darray_destroy(DArray* thiz);
```

动态数组的动态性如何实现了呢？其实很简单，借助标准 C 的内存管理函数 realloc，我们可以轻易改变数组的长度。函数 realloc 是比较费时 的，如果每插入/删除一个元素就要 realloc 一次，不但会带来性能的下降，而且可能造成内存碎片。为了解决这个问题，需要使用一个

称为预先分配的惯用 手法，预先分配实际上是用空间换时间的典型应用，下面我们看看它的实现：

扩展空间

在扩展数组时，不是一次扩展一个元素，而是一次扩展多个元素。至于应该扩展多少个，经验数据是扩展为现有元素个数的 1.5 倍。

```
#define MIN_PRE_ALLOCATE_NR 10
static Ret darray_expand(DArray* thiz, size_t need)
{
    return_val_if_fail(thiz != NULL, RET_INVALID_PARAMS);
    if((thiz->size + need) > thiz->alloc_size)
    {
        size_t alloc_size = thiz->alloc_size + (thiz->alloc_size>>1) + MIN_PRE_ALLOCATE_NR;
        void** data = (void**)realloc(thiz->data, sizeof(void*) * alloc_size);
        if(data != NULL)
        {
            thiz->data = data;
            thiz->alloc_size = alloc_size;
        }
    }
    return ((thiz->size + need) <= thiz->alloc_size) ? RET_OK : RET_FAIL;
}
Ret darray_insert(DArray* thiz, size_t index, void* data)
{
    Ret ret = RET_OOM;
    size_t cursor = index;
    return_val_if_fail(thiz != NULL, RET_INVALID_PARAMS);
    cursor = cursor < thiz->size ? cursor : thiz->size;
    if(darray_expand(thiz, 1) == RET_OK)
    {
        size_t i = 0;
        for(i = thiz->size; i > cursor; i--)
        {
            thiz->data[i] = thiz->data[i-1];
        }
        thiz->data[cursor] = data;
        thiz->size++;
        ret = RET_OK;
    }
    return ret;
}
```

扩展的大小由下列公式得出：

```
size_t alloc_size = (thiz->alloc_size + thiz->alloc_size>>1) + MIN_PRE_ALLOCATE_NR;
```

计算 1.5*thiz->alloc_size 时，我们不使用 1.5 * thiz->alloc_size，因为这样存在浮点数计算，在大多数嵌入式平台中，都没有硬件浮点数计算的支持，浮点数的计算比定点数的计算要慢上百倍。

我们也不使用 thiz->alloc_size+ thiz->alloc_size/2，如果编译器不做优化，除法指令也是比较慢的操作，特别是像在 ARM 这种没有除法指令的芯片中，需要很多条指令才能实现除法的计算。

这里我们使用(thiz->alloc_size + thiz->alloc_size>>1)，这是最快的方法。后面加上 MIN_PRE_ALLOCATE_NR 的原因是避免 thiz->alloc_size 为 0 时存在的错误。

减小空间

在删除元素时也不是马上释放空闲空间，而是等到空闲空间高于某个值时才释放它们。这里我们的做法时，空闲空间多于有效空间一倍时，将总空间调整为有效空间的 1.5 倍。

```
static Ret darray_shrink(DArray* thiz)
{
    return_val_if_fail(thiz != NULL, RET_INVALID_PARAMS);
    if((thiz->size < (thiz->alloc_size >> 1)) && (thiz->alloc_size > MIN_PRE_ALLOCATE_NR))
    {
        size_t alloc_size = thiz->size + (thiz->size >> 1);
        void** data = (void**)realloc(thiz->data, sizeof(void*) * alloc_size);
        if(data != NULL)
        {
            thiz->data = data;
            thiz->alloc_size = alloc_size;
        }
    }
    return RET_OK;
}
Ret darray_delete(DArray* thiz, size_t index)
{
    size_t i = 0;
    Ret ret = RET_OK;
    return_val_if_fail(thiz != NULL && thiz->size > index, RET_INVALID_PARAMS);
    darray_destroy_data(thiz, thiz->data[index]);
    for(i = index; (i+1) < thiz->size; i++)
    {
        thiz->data[i] = thiz->data[i+1];
    }
    thiz->size--;
    darray_shrink(thiz);
    return RET_OK;
}
```

为了避免极端情况下的出现频繁 resize 的情况，在总空间小于等于 MIN_PRE_ALLOCATE_NR 时，我们不减少空间的大小。

本节的示例请到这里下载。

排序

大多数高级排序算法都是针对数组实现的，接下来我们一起学习一下几种排序算法，学习算法本身只是我们的目标之一，最重要的是要从中学习一些思考问题的方法。对比不同算法的

特点，也有助于我们在设计时做出正确的选择。

这里我们请读者实现冒泡排序、快速排序和归并排序。要求如下：

o 算法同时支持升序和降序。按排序结果来看，有升序和降序两种。在升序排列中，前面的元素总是小于/等于后面的元素。在降序中，前面的元素总是大于/等于后面的元素。

o 算法同时支持多种数据类型。教科书上都是以整数排序为示例的，这种简化有且让学生集中精力在算法本身上。但在现实中，我们不能再满足了解算法本身了，而是要写出一些具有实用价值的程序来。

排序

对于前面提的两点额外要求：

o 算法同时支持升序和降序。

o 算法同时支持多种数据类型。

只要认真阅读过前面章节的读者，马上会想到用回调函数。这是对的。软件设计的关键在于熟能生巧，我们反复练习这些基本技巧也意在于此。熟到凭本能就可以运用正确的方法时，那也离所谓的高手不远了。言归正传，我们已经定义过比较回调函数的原型了：

typedef int (*DataCompareFunc)(void* ctx, void* data);

我们先实现一个整数的比较函数，升序比较函数的实现如下：

```
int int_cmp(void* a, void* b)
{
    return (int)a - (int)b;
}
```

降序比较函数的实现如下：

```
int int_cmp_invert(void* a, void* b)
{
    return (int)b - (int)a;
}
```

比较函数也不依赖于具体的数据类型，这样我们就把算法的变化部分独立出来了，是升序还是降序完全由回调函数决定。下面我们看看算法的具体实现。

冒泡排序

冒泡排序的名字很形象的表达了算法的原理，对于降序排列时，排序可以认为是轻的物体不断往上浮的过程。不过对于升序排序，排序认为是重的物体不断向 下沉更为合适。升序排列更符合人类的思考方式，这里我们按升序排序来实现冒泡排序(通过使用不同的比较函数，同样支持降序排序)。

```
Ret bubble_sort(void** array, size_t nr, DataCompareFunc cmp)
{
    size_t i     = 0;
    size_t max   = 0;
    size_t right = 0;
```

```
    return_val_if_fail(array != NULL && cmp != NULL, RET_INVALID_PARAMS);
    if(nr < 2)
    {
        return RET_OK;
    }
    for(right = nr - 1; right > 0; right--)
    {
        for(i = 1, max = 0; i < right; i++)
        {
            if(cmp(array[i], array[max]) > 0)
            {
                max = i;
            }
        }
        if(cmp(array[max], array[right]) > 0)
        {
            void* data = array[right];
            array[right] = array[max];
            array[max] = data;
        }
    }
    return RET_OK;
}
```

冒泡排序是最简单直观的排序算法，教科书上通常作为第一个排序算法来讲。从性能上来看，与其它高级排序算法相比，它似乎没有存在的理由。它除了教学目的之外，是否有实际价值呢？答案是有的。原因有两点：

o 实现简单，简单的程序通常更可靠。虽然我很多年没有写过冒泡排序算法了，从写代码、编译到测试，都一次性通过了。写快速排序时却出了好几次错误， 而且最后参考了教科书才完成。如果非要自己动手写排序算法时，我会先写一个冒泡排序算法，直到一定需要更快的算法时才会考虑其它算法。

o 在小量数据时，所有排序算法性能差别不大。有文章指出，高级排序算法在元素个数多于 1000 时，性能才出现显著提升。在 90%的情况下，我们存储的元素个数只有几十到上百个而已，比如进程数、窗口个数和配置信息等等的数量都不会很大，冒泡排序其实是更好的选择。

请记住：在完成同样任务的情况下，越简单越好。
同时记住：学从难处学，用从易处用。

快速排序

快速排序当然是以其性能优异出名了，而且它不需要额外的空间。如果数据量大而且全部在内存中时，快速排序是首选的排序方法。排序过程是先将元素分成 两个区，所有小于某个元素的值在第一个区，其它元素在第二区。然后分别对这两个区进行快速排序，直到所分的区只剩下一个元素为止。

```
void quick_sort_impl(void** array, size_t left, size_t right, DataCompareFunc cmp)
{
```

```
    size_t save_left  = left;
    size_t save_right = right;
    void* x = array[left];
    while(left < right)
    {
        while(cmp(array[right], x) >= 0 && left < right) right--;
        if(left != right)
        {
            array[left] = array[right];
            left++;
        }
        while(cmp(array[left], x) <= 0 && left < right) left++;
        if(left != right)
        {
            array[right] = array[left];
            right--;
        }
    }
    array[left] = x;
    if(save_left < left)
    {
        quick_sort_impl(array, save_left, left-1, cmp);
    }
    if(save_right > left)
    {
        quick_sort_impl(array, left+1, save_right, cmp);
    }
    return;
}
Ret quick_sort(void** array, size_t nr, DataCompareFunc cmp)
{
    Ret ret = RET_OK;
    return_val_if_fail(array != NULL && cmp != NULL, RET_INVALID_PARAMS);
    if(nr > 1)
    {
        quick_sort_impl(array, 0, nr - 1, cmp);
    }
    return ret;
}
```

战胜软件复杂度是《系统程序员成长计划》的中心思想之一。战胜软件复杂度包括防止复杂度增长和降低复杂度两个方面。降低复杂度的方法主要有抽象和分而治之两种，快速排序则是分而治之的具体体现。

归并排序

与快速排序一样，归并排序也是分而治之的应用。不同的是，它先让左右两部分进行排序，然后把它们合并起来。在排序左右两部分时，同样使用归并排序。快速排序可以认为是自顶向下的方法，而归并排序可以认为是自底向上的方法。

```
static Ret merge_sort_impl(void** storage, void** array, size_t low, size_t mid, size_t high,
```

```
DataCompareFunc cmp)
{
    size_t i = low;
    size_t j = low;
    size_t k = mid;
    if((low + 1) < mid)
    {
        size_t x = low + ((mid - low) >> 1);
        merge_sort_impl(storage, array, low, x, mid, cmp);
    }
    if((mid + 1) < high)
    {
        size_t x = mid + ((high - mid) >> 1);
        merge_sort_impl(storage, array, mid, x, high, cmp);
    }
    while(j < mid && k < high)
    {
        if(cmp(array[j], array[k]) <= 0)
        {
            storage[i++] = array[j++];
        }
        else
        {
            storage[i++] = array[k++];
        }
    }
    while(j < mid)
    {
        storage[i++] = array[j++];
    }
    while(k < high)
    {
        storage[i++] = array[k++];
    }
    for(i = low; i < high; i++)
    {
        array[i] = storage[i];
    }
    return RET_OK;
}
Ret merge_sort(void** array, size_t nr, DataCompareFunc cmp)
{
    void** storage = NULL;
    Ret ret = RET_OK;
    return_val_if_fail(array != NULL && cmp != NULL, RET_INVALID_PARAMS);
    if(nr > 1)
    {
        storage = (void**)malloc(sizeof(void*) * nr);
        if(storage != NULL)
        {
```

```
            ret = merge_sort_impl(storage, array, 0, nr>>1, nr, cmp);
            free(storage);
        }
    }
    return ret;
}
```

归并排序需要额外的存储空间，其为被排序的数组一样大。大部分示例代码里，都在每次递归调用中分配空间，这些会带来性能上的下降。这里我们选择了事先分配一块空间，在排序过程中重复使用，算法更简单，性能也得到提高。

根据把要排序的数组分成N个部分，可以把归并排序称为N路排序。上面实现的归并排序实际是归并算法一个特例：两路归并。看似归并排序与快速排序相比 几乎没有优势可言，但是归并排序更重要的能力在于处理大量数据的排序，它不要求被排序的数据全部在内存中，所以在数据大于内存的容纳能力时，归并排序就更 能大展身手了。归并排序最常用的地方是数据库管理系统(DBMS)，因为数据库中存储的数据通常无法全部加载到内存中来的。有兴趣的读者可以阅读相关资 料。

排序算法的测试

排序算法的实现不同，但它们的目的都一样：让数据处于有序状态。所以在写自动测试时，没有必要为每一种算法写一个测试程序。通过将排序算法作为回调函数传入，我们可以共用一个测试程序：

```
static void** create_int_array(int n)
{
    int i = 0;
    int* array = (int*)malloc(sizeof(int) * n);
    for(i = 0; i < n; i++)
    {
        array[i] = rand();
    }
    return (void**)array;
}
static void sort_test_one_asc(SortFunc sort, int n)
{
    int i = 0;
    void** array = create_int_array(n);
    sort(array, n, int_cmp);
    for(i = 1; i < n; i++)
    {
        assert(array[i] >= array[i-1]);
    }
    free(array);
    return;
}
void sort_test(SortFunc sort)
{
    int i = 0;
    for(i = 0; i < 1000; i++)
```

```
    {
        sort_test_one_asc(sort, i);
    }
    return ;
}
```

集成排序算法到动态数组中

把排序算法集中到动态数组并不合适，原因有：

o 绑定动态数组与特定算法不如让用户根据需要去选择。

o 在动态数组中实现排序算法不利于算法的重用。

所以我们给动态数组增加一个排序函数，但排序算法通过回调函数传入：

```
Ret    darray_sort(DArray* thiz, SortFunc sort, DataCompareFunc cmp);
```

本节示例代码请到**这里**下载。

有序数组的两个应用

前面我们学习了数组的排序方法，通常我们对数组排序不是为了排序而排序，而是为了其它的用途才排序的，这里了解一下有序数组的两个常见应用。

二分查找

二分查找也称为折半查找，它的前提是数组中的元素是有序的。算法过程如下(假定数组为升序)：先拿要查找的元素与数组中间位置的元素相比较，如果小 于则在数组的前半部分查找，大于则在数组的后半部分查找，相等则在找到了。重复这个过程直到找到或者数组被分成单个元素为止。实现如下：

```
int qsearch(void** array, size_t nr, void* data, DataCompareFunc cmp)
{
    int low    = 0;
    int mid    = 0;
    int high   = nr-1;
    int result = 0;
    return_val_if_fail(array != NULL && cmp != NULL, -1);
    while(low <= high)
    {
        mid  = low + ((high - low) >> 1);
        result = cmp(array[mid], data);
        if(result == 0)
        {
            return mid;
        }
        else if(result < 0)
        {
            low = mid + 1;
        }
        else
        {
```

```
            high = mid - 1;
        }
    }
    return -1;
}
```

在编写二分查找的代码时，除了算法本身外还要注意两个问题：

o 计算中间位置的方法。这里使用 mid = low + ((high - low) >> 1)代替(low+hign)/2，目的是为了避免整数溢出和除法计算。

o 边界值问题。在编写排序和查找的程序时，最容易犯边界值错误，写程序时一定要保持思路清晰。不防模拟计算机去执行你写的程序，用不同的输入观察所得的结果，最后加上自动测试，可以大大减少出错的概率。

去除重复元素

在工作中，我经常使用 linux 中的命令 sort 和 uniq 的组合。uniq 的功能是去除重复的元素，它的前提也是要求数据是有序的。下面我们写一个程序，它打印数组中不重复元素（整数）：

```
Ret  unique_print_int(void* ctx, void* data)
{
    if(*(int*)ctx != (int)data)
    {
        *(int*)ctx = (int)data;
        printf("%d ", (int)data);
    }
    return RET_OK;
}
darray_foreach(darray, unique_print_int, &data);
```

注意：记得把 data 初始化成不等于第一个元素的值，否则可能漏打第一个元素。这个算法当然同样适用链表，只要是有序的即可。

本节示例代码请到**这里**下载。

在《设计模式-可复用面向对象软件的基础》的序言里提到软件设计的两个基本原则：

针对接口编程，而不是针对实现编程。接口是抽象的，因为抽象所以简单。接口是对象的本质，因为是本质所以稳定。接口是降低复杂度和隔离变化的有力武 器，C 语言里没有接口的概念，不是因为接口不重要，而是 C 语言把它视为理所当然的东西(函数指针无所不在), 正所谓玫瑰不叫玫瑰，依然芳香如故。在前面我们已经看到，C 语言里的接口是相当直观和优雅的。

优先使用组合，而不是类继承。与组合相比，继承更为复杂，特别是多重继承和多层继承，甚至会带来一些歧义，给理解上造成困难。与组合相比，继承缺乏 灵活性，继承在编译时就绑定了父类和子类之间的关系，而组合却可以在运行时动态改变。C 语言没有继承的概念(当然可以实现继承)，自然大量使用组合来构建 大型系统，这在客观上恰恰与设计原则是一致的。

本章节将借助队列，栈和哈表来练习这种基本的重用方法。请读者实现队列，栈和哈表，要

求重用前面的链表实现。

队列

队列是一种很常用的数据，操作系统用队列来管理运行的进程，驱动程序用队列来管理要传输的数据包，GUI 框架用队列来管理各种 GUI 事件。队列是一 种先进先出(FIFO, First in First out)的数据结构，数据只能从队列头取，向队列尾追加。队列的名称很形象的表达了它的意义，就像排队上车一样，前面的先上，后来的站在后面排队。

队列主要的接口函数有：

o 创建队列：queue_create
o 取队列头的元素：queue_head
o 追加一个元素到队列尾：queue_push
o 删除队列头元素：queue_pop
o 取队列中元素个数(同时也可以判断队列是否为空)：queue_length
o 遍历队列中的元素：queue_foreach;
o 销毁队列 queue_destroy

队列看起来比链表和数组要高级一点，其实它只不过是链表和数组的一种特殊形式而已，下面我们重用链表来实现队列：

o 队列的数据结构

```
struct _Queue
{
    DList* dlist;
};
```

这里队列只是对双向链表的一个包装。

o 创建队列

```
Queue* queue_create(DataDestroyFunc data_destroy, void* ctx)
{
    Queue* thiz = (Queue*)malloc(sizeof(Queue));
    if(thiz != NULL)
    {
        if((thiz->dlist = dlist_create(data_destroy, ctx)) == NULL)
        {
            free(thiz);
            thiz = NULL;
        }
    }
    return thiz;
}
```

创建队列时，除了分配自己的空间外，就是简单的创建一个双向链表。

o 取队列头的元素

```
Ret      queue_head(Queue* thiz, void** data)
```

```
{
    return_val_if_fail(thiz != NULL && data != NULL, RET_INVALID_PARAMS);
    return dlist_get_by_index(thiz->dlist, 0, data);
}
```

我们认为链表的第一个元素是队列头，取队列头的元素就是取链表的第一个元素。

o 追加一个元素到队列尾

```
Ret      queue_push(Queue* thiz, void* data)
{
    return_val_if_fail(thiz != NULL, RET_INVALID_PARAMS);
    return dlist_append(thiz->dlist, data);
}
```

我们认为链表的最后一个元素是队列尾，追加一个元素到队列尾就是追加一个元素到链表尾。

o 删除队列头元素

```
Ret      queue_pop(Queue* thiz)
{
    return_val_if_fail(thiz != NULL, RET_INVALID_PARAMS);
    return dlist_delete(thiz->dlist, 0);
}
```

删除队列头的元素就是删除链表的第一个元素。

o 取队列中元素个数

```
size_t   queue_length(Queue* thiz)
{
    return_val_if_fail(thiz != NULL, 0);
    return dlist_length(thiz->dlist);
}
```

队列中元素的个数等同于链表元素的个数。

o 遍历队列中的元素

```
Ret      queue_foreach(Queue* thiz, DataVisitFunc visit, void* ctx)
{
    return_val_if_fail(thiz != NULL && visit != NULL, RET_INVALID_PARAMS);
    return dlist_foreach(thiz->dlist, visit, ctx);
}
```

遍历队列中的元素等同于遍历链表中的元素。

o 销毁队列

```
void queue_destroy(Queue* thiz)
{
    if(thiz != NULL)
    {
        dlist_destroy(thiz->dlist);
```

```
        thiz->dlist = NULL;
        free(thiz);
    }
    return;
}
```

销毁链表然后释放自身的空间。

用组合的方式实现队列很简单吧，不用半小时就可以写出来。虽然链表已经通过了自动测试，队列的自动测试也不能省，在这里我们就不列出代码了。

上面我们实现的是通用队列，除了通用队列外，还有几种特殊队列应用也非常广泛：

o 优先级队列。它的不同之外在于，插入元素时，不必追加到队尾，而是根据元素的优先级插到队列的适当位置。这个就像排队上车时，后面来了个老人，大家让他先上一样的。内核的进程管理器通常采用估先级队列管理进程。

o 循环队列。循环队列的特点是最大元素个数是固定的。这个我们在前面学习同步时已经提过，它的优点是两个并发的实体(线程或进程)，按一个读一个写的方式访问，不需要加锁。设备驱动程序经常使用循环队列来管理数据包。

本节示例代码请到**这里**下载。

栈

栈是一种后进先出(LIFO, last in first out)的数据结构，与队列的先进先出(FIFO)相比，这种规则似乎不太公平，计算机可不管这个。事实上，栈是最重要的数据结构之一：没有栈，基于下推 自动机的编译器不能工作，我们只能写汇编程序。没有栈，无法实现递归/多级函数调用，程序根本就无法工作。

栈主要的接口函数有：

o 创建栈 stack_create
o 取栈顶元素 stack_top
o 放入元素到栈顶 stack_push
o 删栈栈顶元素 stack_pop
o 取栈中元素个数 stack_length
o 遍历栈中的元素 stack_foreach
o 销毁栈 stack_destroy

栈同样是链表和数组的一种特殊形式而已，下面我们重用链表来实现栈：

o 栈的数据结构

```
struct _Stack
{
    DList* dlist;
};
```

这里和队列的数据结构一样，由一个链表组成。

o 创建栈

```
Stack* stack_create(DataDestroyFunc data_destroy, void* ctx)
{
    Stack* thiz = (Stack*)malloc(sizeof(Stack));
    if(thiz != NULL)
    {
        if((thiz->dlist = dlist_create(data_destroy, ctx)) == NULL)
        {
            free(thiz);
            thiz = NULL;
        }
    }
    return thiz;
}
```

创建栈时，除了分配自己的空间外，就是简单的创建一个双向链表。

o 取栈顶元素

```
Ret      stack_top(Stack* thiz, void** data)
{
    return_val_if_fail(thiz != NULL && data != NULL, RET_INVALID_PARAMS);
    return dlist_get_by_index(thiz->dlist, 0, data);
}
```

我们认为链表的第一个元素是栈顶，取栈顶的元素就是取链表的第一个元素。

o 放入元素到栈顶

```
Ret      stack_push(Stack* thiz, void* data)
{
    return_val_if_fail(thiz != NULL, RET_INVALID_PARAMS);
    return dlist_prepend(thiz->dlist, data);
}
```

放入元素到栈顶就是插入一个元素到链表头。

o 删栈栈顶元素

```
Ret      stack_pop(Stack* thiz)
{
    return_val_if_fail(thiz != NULL, RET_INVALID_PARAMS);
    return dlist_delete(thiz->dlist, 0);
}
```

删栈栈顶元素就是删除链表的第一个元素。

o 取栈中元素个数

```
size_t   stack_length(Stack* thiz)
{
    return_val_if_fail(thiz != NULL, 0);
```

```
    return dlist_length(thiz->dlist);
}
```

栈中的个数等同于链表的个数。

o 遍历栈中的元素

```
Ret      stack_foreach(Stack* thiz, DataVisitFunc visit, void* ctx)
{
    return_val_if_fail(thiz != NULL && visit != NULL, RET_INVALID_PARAMS);
    return dlist_foreach(thiz->dlist, visit, ctx);
}
```

遍历栈中的元素等同于遍历链表中的元素。

o 销毁栈

```
void stack_destroy(Stack* thiz)
{
    if(thiz != NULL)
    {
        dlist_destroy(thiz->dlist);
        thiz->dlist = NULL;
        free(thiz);
    }
    return;
}
```

销毁双向链表然后释放自身的空间。

栈是一个非常重要的数据，但奇怪的是我们很少有机会去写它。事实上，我从来没有在工作中写过一个栈。这是怎么回事呢？原因是我们的计算机本身是基于 栈的，很多事情计算机已经在我们不知道的情况下帮我们处理了，比如函数调用（特殊是递归调用），计算机帮我们处理了。用递归下降进行的语法分析利用了函数 调用的递归性，也不需要显式的构造栈。

哈希表

前面我们已经体会到了组合的威力，用短短几十行代码就搞定了队列和栈。现在轮到哈希表了，在此之前已经有几位读者向我抱怨，哈希表太难写了！其实哈 希表也很简单，前面我们说了队列和栈只不过是链表或者数组的特殊情况而已，哈希表当然不再是链表或者数组的特殊情况了，但是我们同样可以用组合的方式来实 现它。简单点说：

哈希表 = 数组 + 链表

有读者说，老兄，你在玩我吧。不，我是认真的。我说的“加”当然不是简单的叠加起来，组合也是需要技巧的，不同的组合得到的效果不一样，如何去组合也是需要花时间去学习的。

哈希表的基本接口有：

o 创建 hash_table_create
o 插入 hash_table_insert

o 删除 hash_table_delete
o 查找 hash_table_find
o 计算元素个数 hash_table_length
o 遍历所有元素 hash_table_foreach
o 销毁 hash_table_destroy

现在看看怎样用数组和链表组合出哈希表：

o 哈希表的数据结构

```
struct _HashTable
{
    DataHashFunc    hash;
    DList**         slots;
    size_t           slot_nr;
    DataDestroyFunc  data_destroy;
    void*           data_destroy_ctx;
};
```

hash 是一个函数指针，用来计算数据的哈希值。哈希函数的好坏基本上决定了哈希表的效率，好的哈希函数计算出的哈希值分布比较均匀。遗憾的是哈希 表的设计者们谁都不知道什么样的哈希函数是最好的，因为哈希函数的好坏只能动态评估，它与数据类型和应用环境密切相关。按照惯例，实现者不知道的事就应该 让调用者去实现，所以把哈希函数设计成回调函数，由调用者提供。

slots 是哈希表的主体，它是一个双向链表的指针数组。所以说哈希表 = 数组 + 链表是有道理的。由于这个数组不需要动态增长，所以用最简单的指针数组就好了。

slot_nr 是数组的大小，在哈希函数不变的情况下，slot_nr 的大小对哈希表的性能起决定作用。slot_nr 的值越大性能越高，但空间浪 费也越大，这又是一个时/空互换的例子。所以这个值也由调用者确定会好一点。很多书都认为这个值应该选择一个素数，我认为这没有什么理论根据，至少没有找 到严密的数学证明。

o 创建

```
HashTable* hash_table_create(DataDestroyFunc data_destroy, void* ctx, DataHashFunc hash, int
slot_nr
)
{
    HashTable* thiz = NULL;
    return_val_if_fail(hash != NULL && slot_nr > 1, NULL);
    thiz = (HashTable*)malloc(sizeof(HashTable));
    if(thiz != NULL)
    {
        thiz->hash = hash;
        thiz->slot_nr  = slot_nr;
        thiz->data_destroy_ctx = ctx;
        thiz->data_destroy = data_destroy;
        if((thiz->slots = (DList**)calloc(sizeof(DList*)*slot_nr, 1)) == NULL)
        {
```

```
            free(thiz);
            thiz = NULL;
        }
    }
    return thiz;
}
```

创建哈希表时，我们只是创建了数组，而链表则在第一次使用时再创建。这种延迟处理的手法在加快起动速度时是很常见的，这种做法也会减少一些不必要的开销(有些对象可能根本就不会用到)。

o 插入 hash_table_insert

```
Ret      hash_table_insert(HashTable* thiz, void* data)
{
    size_t index = 0;
    return_val_if_fail(thiz != NULL, RET_INVALID_PARAMS);
    index = thiz->hash(data)%thiz->slot_nr;
    if(thiz->slots[index] == NULL)
    {
        thiz->slots[index] = dlist_create(thiz->data_destroy, thiz->data_destroy_ctx);
    }
    return dlist_prepend(thiz->slots[index], data);
}
```

先计算元素所在链表，如果链表还没有创建就创建它，然后把元素插入到链表中。怎样？只是几行代码而已。

o 删除 hash_table_delete

```
Ret      hash_table_delete(HashTable* thiz, DataCompareFunc cmp, void* data)
{
    int index = 0;
    DList* dlist = NULL;
    return_val_if_fail(thiz != NULL && cmp != NULL, RET_INVALID_PARAMS);
    index = thiz->hash(data)%thiz->slot_nr;
    dlist = thiz->slots[index];
    if(dlist != NULL)
    {
        index = dlist_find(dlist, cmp, data);
        return dlist_delete(dlist, index);
    }
    return RET_FAIL;
}
```

先计算元素所在的链表，然后从链表中删除元素。

o 查找 hash_table_find

```
Ret    hash_table_find(HashTable* thiz, DataCompareFunc cmp, void* data, void** ret_data)
{
    int index = 0;
```

```
    DList* dlist = NULL;
    return_val_if_fail(thiz != NULL && cmp != NULL && ret_data != NULL, RET_INVALID_PARAMS);
    index = thiz->hash(data)%thiz->slot_nr;
    dlist = thiz->slots[index];
    if(dlist != NULL)
    {
        index = dlist_find(dlist, cmp, data);
        return dlist_get_by_index(dlist, index, ret_data);
    }
    return RET_FAIL;
}
```

先计算元素所在的链表，然后从链表中查找元素。

o 计算元素个数 hash_table_length

```
size_t   hash_table_length(HashTable* thiz)
{
    size_t i = 0;
    size_t nr = 0;
    return_val_if_fail(thiz != NULL, 0);
    for(i = 0; i < thiz->slot_nr; i++)
    {
        if(thiz->slots[i] != NULL)
        {
            nr += dlist_length(thiz->slots[i]);
        }
    }
    return nr;
}
```

这个麻烦一点，需要累加所有链表中元素个数。

o 遍历所有元素 hash_table_foreach

```
Ret      hash_table_foreach(HashTable* thiz, DataVisitFunc visit, void* ctx)
{
    size_t i = 0;
    return_val_if_fail(thiz != NULL && visit != NULL, RET_INVALID_PARAMS);
    for(i = 0; i < thiz->slot_nr; i++)
    {
        if(thiz->slots[i] != NULL)
        {
            dlist_foreach(thiz->slots[i], visit, ctx);
        }
    }
    return RET_OK;
}
```

依次调用每个链表的 dlist_foreach。

o 销毁 hash_table_destroy

```
void hash_table_destroy(HashTable* thiz)
{
    size_t i = 0;
    if(thiz != NULL)
    {
        for(i = 0; i < thiz->slot_nr; i++)
        {
            if(thiz->slots[i] != NULL)
            {
                dlist_destroy(thiz->slots[i]);
                thiz->slots[i] = NULL;
            }
        }
        free(thiz->slots);
        free(thiz);
    }
    return;
}
```

销毁所有链表，释放数组和哈希表本身。

哈希表的两种特殊情况：

o 哈希函数极差：所有元素计算出同一个哈希值。则哈希表退化成一个链表，查找的时间效率为 O(n)。

o 哈希函数极好：所有元素计算出不同的哈希值，而且元素个数为 slot_nr。则哈希表等同于一个数组，通过索引定位元素，查找的时间效率为 O(1)。

所以哈希表的查找效率在 O(1)到 O(n)之间，大部分人选择哈希表时，并没有仔细评估哈希函数的好坏，而又期望得到很高的查找效率，其实这只是一种幻觉。如果查找效率要求比较高，通常我会选择有序数组，用二分查找来做，至少它的查找效率是比较确定的。

本节示例代码请到**这里**下载。

系统程序员成长计划-容器与算法(一)（上）

前面我们通过组合的方法，用双向链表实现了队列。大家已经看到，它的实现非常简单。有的读者可能会问了：你说过双向链表和动态数组都有各自的适用范 围，在有的情况下，用双向链表合适，在有的情况下，用动态数组合适。你用双向链表实现了队列，是不是有必要用动态数组再实现一次呢？如果对栈同样如此，那 是不是做了太多重复的工作，有违背 DRY(don’t repeat yourself)原则呢？

问得好，这就是本章要学习的。我们请读者实现一个队列，要求如下：

o 由调用者决定用双向链表实现还是用动态数组实现，
o 在运行时决定，而不是在编译时决定（宏定义）。
o 如果调用者乐意，他以后还可以选择用单向链表来实现，而不必修改队列的实现代码。
o Don’t Repeat Yourself。

请读者不要急着看后面的内容，先仔细想想，尝试完成这个任务。

容器

在运行时动态选择实现方式，这个问题可是有点难度的。我很少见人能独立完成，最常见的做法是：用一个参数来决定调用双向链表的函数还是动态数组的函数。如：

```
Ret      queue_head(Queue* thiz, void** data)
{
    return_val_if_fail(thiz != NULL && data != NULL, RET_INVALID_PARAMS);
    if(thiz->use_dlist)
    {
        return dlist_get_by_index(thiz->dlist, 0, data);
    }
    else
    {
        return darray_get_by_index(thiz->darray, 0, data);
    }
}
```

这种方法满足了前面两个要求，但无法满足第三个要求，它限定了只能使用双向链表和动态数组两种方法来实现队列，要添加其它实现方法(如单向链表)就 要修改队列的本身了。当然你可以选择把目前所知的实现方式全部写在里面，但那样会带来复杂度上升，性能下降，空间浪费和错误增加等种种不利因素。更何况在 现实中，我们是无法预知所有可能出现的实现方式的。

在经过前面反复的练习之后，提到一个对象有多种不同的实现方式时，大部分读者都会想到用接口了，这是很好的。于是有人提出，抽象出一个队列的接口，分别用双向链表和动态数组来实现，这样就可以把调用者和队列的不同实现分开了。

这个想法不错，至少通过队列接口可以解决前面三个问题了。但问题是，用双向链表实现的队列和与用动态数组实现的队列，从逻辑上来看，它们完全是重复 的，从代码的角度来看，它们又有些差别。如果你亲手去写过，总感觉到里面有点不太对劲，这种感觉通常是需要改进的征兆。在仔细思考之后，我们会发现队列的 逻辑是不变的，变化的只是容器。也就是说应该抽象的是容器接口而不是队列接口。

最小粒度抽象原则。这个原则是我命名的，虽然不是出自大师之手，但它同样是有效的。接口隔离了调用者和具体实现，不管抽象的粒度大小，接口都能起到 隔离变化的作用。但是粒度越大造成的重复越多，上面的例子或许不太明显，假设我们要开发一个OS 内核，这个OS 可以在不同的硬件平台上运行，硬件平台是变 化的，我们只要对硬件平台进行抽象就好了，如果对OS 整体进行抽象，则意味着每个硬件平台写一个OS，那样就存在太多重复工作了。

前面我们在引入Locker 这个接口时，是先有这个接口然后再去实现它。这里不过是已经有实现好的双向链表和动态数组，我们再引入容器这个接口而已。它们的目的是一样的：隔离变化。回到正题，下面看看容器的接口：

```
struct _LinearContainer;
typedef struct _LinearContainer LinearContainer;
typedef Ret (*LinearContainerInsert)(LinearContainer* thiz, size_t index, void* data);
typedef Ret (*LinearContainerPrepend)(LinearContainer* thiz, void* data);
```

```
typedef Ret (*LinearContainerAppend)(LinearContainer* thiz, void* data);
typedef Ret (*LinearContainerDelete)(LinearContainer* thiz, size_t index);
typedef Ret (*LinearContainerGetByIndex)(LinearContainer* thiz, size_t index, void** data);
typedef Ret (*LinearContainerSetByIndex)(LinearContainer* thiz, size_t index, void* data);
typedef size_t (*LinearContainerLength)(LinearContainer* thiz);
typedef int    (*LinearContainerFind)(LinearContainer* thiz, DataCompareFunc cmp, void* ctx);
typedef Ret    (*LinearContainerForeach)(LinearContainer* thiz, DataVisitFunc visit, void* ctx);
typedef void   (*LinearContainerDestroy)(LinearContainer* thiz);
struct _LinearContainer
{
    LinearContainerInsert     insert;
    LinearContainerPrepend    prepend;
    LinearContainerAppend     append;
    LinearContainerDelete     del;
    LinearContainerGetByIndex get_by_index;
    LinearContainerSetByIndex set_by_index;
    LinearContainerLength     length;
    LinearContainerFind       find;
    LinearContainerForeach    foreach;
LinearContainerDestroy    destroy;
    char priv[0];
};
```

从上面的代码可以看出，容器接口、双向链表和动态数组的基本上是一致的。

下面我们看看基于双向链表的实现：

o 数据结构

```
typedef struct _PrivInfo
{
    DList* dlist;
}PrivInfo;
```

这是容器的私有数据，它只是对双向链表的包装，请不要与前面的“组合”搞混了：组合是为了实现新的功能，而这里是为了分隔接口和实现，让实现部分可以独立变化。

o 实现接口函数

```
static Ret linear_container_dlist_insert(LinearContainer* thiz, size_t index, void* data)
{
    PrivInfo* priv = (PrivInfo*)thiz->priv;
    return dlist_insert(priv->dlist, index, data);
}
static Ret linear_container_dlist_delete(LinearContainer* thiz, size_t index)
{
    PrivInfo* priv = (PrivInfo*)thiz->priv;
    return dlist_delete(priv->dlist, index);
}
...
```

同样这里只是对参数做简单转发，调用都是一一对应的，其它函数的实现类似，这里就不一

个一个的分析了。

o 创建函数

```
LinearContainer* linear_container_dlist_create(DataDestroyFunc data_destroy, void* ctx)
{
    LinearContainer* thiz = (LinearContainer*)malloc(sizeof(LinearContainer) +
sizeof(PrivInfo));
    if(thiz != NULL)
    {
        PrivInfo* priv = (PrivInfo*)thiz->priv;
        priv->dlist = dlist_create(data_destroy, ctx);
        thiz->insert        =  linear_container_dlist_insert;
        thiz->prepend       =  linear_container_dlist_prepend;
        thiz->append        =  linear_container_dlist_append;
        thiz->del           =  linear_container_dlist_delete;
        thiz->get_by_index  =  linear_container_dlist_get_by_index;
        thiz->set_by_index  =  linear_container_dlist_set_by_index;
        thiz->length        =  linear_container_dlist_length;
        thiz->find          =  linear_container_dlist_find;
        thiz->foreach       =  linear_container_dlist_foreach;
        thiz->destroy       =  linear_container_dlist_destroy;
        if(priv->dlist == NULL)
        {
            free(thiz);
            thiz = NULL;
        }
    }
}
```

创建函数的主要工作是把函数指针指向实际的函数。

有的读者可能会问，你用双向链表和动态数组实现了两个容器，我用双向链表和动态数组实现了两个队列，这没有什么差别啊，至少你的代码量并不会减少。能想到这点是不错的，所谓不怕不识货就怕货比货，这两种方法到底有什么差别呢？

前面我们说过，快速排序在元素很少的情况下，它的性能甚至不如冒泡排序高，但是随着元素个数的增加，它的优势就越来越明显了。好的设计也是一样的， 在小程序中，它的表现并不比普通设计强多少，但是随着规模的扩大，重用次数的增多，它的优势就会明显起来。单从实现队列来看，容器接口没有什么优势。眼光 再看远一点，我们会发现：用同样的方法实现栈不需要重写这些代码了;双向链表和动态数组可以共用一个测试程序;如果其它地方还有类似的需求，重用次数会越来越多，它的优势就更明显了。

下面我们来看看用容器来实现队列：

o 数据结构

```
struct _Queue
{
    LinearContainer* linear_container;
};
```

o 队列的创建函数

```
Queue* queue_create(LinearContainer* container)
{
    Queue* thiz = NULL;
    return_val_if_fail(container != NULL, NULL);
    thiz = (Queue*)malloc(sizeof(Queue));
    if(thiz != NULL)
    {
        thiz->linear_container = container;
    }
    return thiz;
}
```

容器对象由调用者传入，而不是在队列里硬编码，这样它的变化不会影响队列。

o 队列的实现函数

```
Ret      queue_head(Queue* thiz, void** data)
{
    return_val_if_fail(thiz != NULL && data != NULL, RET_INVALID_PARAMS);
    return linear_container_get_by_index(thiz->linear_container, 0, data);
}
Ret      queue_push(Queue* thiz, void* data)
{
    return_val_if_fail(thiz != NULL, RET_INVALID_PARAMS);
    return linear_container_append(thiz->linear_container, data);
}
...
```

用容器实现的队列和前面用双向链表实现的队列没有多大差别，只是函数名称不一样而已。

由此可见，队列与双向链表和动态数组没有任何关系，队列关心的只是 linear_container 这个接口，如果调用者想换成单向链表，用单向链 表去实现 linear_container 接口就好了。在 C＋＋的标准模板库(STL)里有类似的做法，STL 里的 Sequence 相当于这里的 LinearContainer 接口。

C 语言程序中广泛使用上述的做法，但我并不推荐读者这样去实现队列(当然也不反对)。原因是在通常情况下，队列中存放的元素并不多，使用双向链表、动态数组和其它容器都没有多大差别，用最简单的方法实现它就行了。

本节示例代码请到**这里**下载。

容器用来存储数据，算法用来处理数据。容器有多种，算法的种类更多，两者的组合数目就数不胜数了。如果同样的算法要为每种容器都写一遍，写的时候单调不说，维护起来也很困难。所以我们一直在寻找让算法独立于容器的方法，让同一种算法能适用于所有(或多种)的容器。

在开始的时候，我们通过 foreach 遍历函数＋回调函数的方法，第一次分离了算法和容器，算法不但可以独立于容器变化，而且同时适用于多种容器：大/小写转换函数只需要写一次，就可以在双向链表、动态数组、队列、栈和哈希表等多种容器中重用。

前面又我们抽象了容器的接口，容器的使用者不需要关心容器的实现，也就是说同一个算法可以适用于多种不同的容器(不是全部，前面只抽象了线形的容器)。我们用这种方面实现了队列和栈，还重用了双向链表和动态数组的测试程序。

可惜上面两种方法仍然不能解决让算法独立于容器的全部问题。这里请读者先实现这样一个算法：

o 反序排列容器中的元素：第一个元素与最后一元素交换，第二个元素与倒数第二个元素交换，以此类推。

o 这个算法同时适用双向链表和动态数组。

o 在双向链表和动态数组上的运行效率处于同一级别。

这个反序函数的原理很简单，有的读者很快就写出来了：

```
Ret invert(LinearContainer* linear_container)
{
    int i = 0;
    int j = 0;
    void* data1 = NULL;
    void* data2 = NULL;
    return_val_if_fail(linear_container != NULL, RET_INVALID_PARAMS);
    j = linear_container_length(linear_container) - 1;
    for(; i < j; i++, j--)
    {
        linear_container_get_by_index(linear_container, i, &data1);
        linear_container_get_by_index(linear_container, j, &data2);
        linear_container_set_by_index(linear_container, i, data2);
        linear_container_set_by_index(linear_container, j, data1);
    }
    return RET_OK;
}
```

这种实现利用我们抽象的 LinearContainer 接口，它同时适用于双向链表和动态数组，可惜无法满足第三个条件，双向链表和动态数组的性能差异很大。原因是，动态数组通 过偏移量可以定位，而双向链表则需要一个一个的移动指针才能定位，在循环内部调用 get_by_index/ set_by_index 更加剧它们在性能上的差异。

有的读者可能尝试了用 foreach 去实现，foreach 确实很好用，但它的缺点是只能按固定的顺序去遍历元素，不能同时即反序又顺序的遍历，所 以用 foreach 实现上述算法是有点困难的。这里我们引入一个新的概念：迭代器。迭代器是一种更灵活的遍历行为的抽象，它可以按任意顺序去访问容器内的 元素，而又不暴露容器的内部结构。

在引入迭代器之前，我们分析一下 invert 的算法，先考虑针对双向链表的实现：

1. 定义两个指针，一个指向链表的首结点，一个指向链表的尾结点。
2. 交换两个指针指向的内容。
3. 前面的指针向后移，后面的指针向前移，重复第二步直到两个指针相遇。

仔细看看上面的算法，我们发现在整个过程中，操作的只是两个指针。关于这个指针，它具

有下列行为：

o 获取指针指向的数据。
o 设置指针指向的数据。
o 指针向后移。
o 指针向前移。
o 比较的大小。

在前面的算法中，指针是依赖于双向链表的，离开了这个特定的容器，我们无法确定指针的行为。那么我们可以对这里指针进行抽象，让它针对不同容器有不同的实现，而算法只关心它的指针这个接口。为了遵循一些约定俗成的规则，我们把这里的指针称为迭代器(iterator)。

迭代器(iterator)的接口定义如下：

```
typedef Ret  (*IteratorSetFunc)(Iterator* thiz, void* data);
typedef Ret  (*IteratorGetFunc)(Iterator* thiz, void** data);
typedef Ret  (*IteratorNextFunc)(Iterator* thiz);
typedef Ret  (*IteratorPrevFunc)(Iterator* thiz);
typedef Ret  (*IteratorAdvanceFunc)(Iterator* thiz, int offset);
typedef int  (*IteratorOffsetFunc)(Iterator* thiz);
typedef Ret  (*IteratorCloneFunc)(Iterator* thiz, Iterator** cloned);
typedef void (*IteratorDestroyFunc)(Iterator* thiz);
struct _Iterator
{
    IteratorSetFunc     set;
    IteratorGetFunc     get;
    IteratorNextFunc    next;
    IteratorPrevFunc    prev;
    IteratorAdvanceFunc advance;
    IteratorCloneFunc   clone;
    IteratorOffsetFunc  offset;
    IteratorDestroyFunc destroy;
    char priv[0];
};
```

为了让迭代器具有更广的适应能力，在其它算法也可以使用，这里增加了几个接口函数：

o advance 随机定位迭代器位置，由当前位置和偏移量决定。
o offset 得到迭代器的偏移量，主要用作比较迭代器的大小。
o clone 拷贝一份迭代器。

现在我们来看针对双向链表的实现：

o 数据结构

```
typedef struct _PrivInfo
{
    DList* dlist;
    DListNode* cursor;
    int offset;
```

```
}PrivInfo;
```

一个指向双向链表的指针，它表明迭代器所属的容器。
一个指向当前结点的指针，它表明迭代器的位置。
一个标明当前偏移量的值，目的是提高获取偏移量的速度。

o 实现迭代器的函数

```
static Ret  dlist_iterator_set(Iterator* thiz, void* data)
{
    PrivInfo* priv = (PrivInfo*)thiz->priv;
    return_val_if_fail(priv->cursor != NULL && priv->dlist != NULL, RET_INVALID_PARAMS);
    priv->cursor->data = data;
    return RET_OK;
}
static Ret  dlist_iterator_next(Iterator* thiz)
{
    Ret ret = RET_FAIL;
    PrivInfo* priv = (PrivInfo*)thiz->priv;
    return_val_if_fail(priv->cursor != NULL && priv->dlist != NULL, RET_INVALID_PARAMS);
    if(priv->cursor->next != NULL)
    {
        priv->cursor = priv->cursor->next;
        priv->offset++;
        ret = RET_OK;
    }
    return ret;
}
...
```

从这些代码可以看出，双向链表的迭代器其实就是对双向链表结点的包装，经过这个包装之后，可以访问具体的结点而又不暴露双向链表的内部实现，并且调用者不再依赖双向链表了。

o 创建函数

```
Iterator* dlist_iterator_create(DList* dlist)
{
    Iterator* thiz = NULL;
    return_val_if_fail(dlist != NULL, NULL);
    if((thiz = malloc(sizeof(Iterator) + sizeof(PrivInfo))) != NULL)
    {
        PrivInfo* priv = (PrivInfo*)thiz->priv;
        thiz->set      =  dlist_iterator_set;
        thiz->get      =  dlist_iterator_get;
        thiz->next     =  dlist_iterator_next;
        thiz->prev     =  dlist_iterator_prev;
        thiz->advance  =  dlist_iterator_advance;
        thiz->clone    =  dlist_iterator_clone;
        thiz->offset   =  dlist_iterator_offset;
        thiz->destroy  =  dlist_iterator_destroy;
        priv->dlist  = dlist;
```

```
        priv->cursor = dlist->first;
        priv->offset = 0;
    }
    return thiz;
}
```

这里我们还遇到另外一个难题：双向链表的迭代器的实现放在哪里？放在 dlist.c 里还是单独的文件里，放在单独的文件里可读性更强，但是在访问双向链表的内部数据时会遇到一些麻烦。最后选择是：

o dlist_iterator.h 提供独立的文件。
o dlist_iterator.c 也提供独立的文件，但是它不直接参与编译，而是在 dlist.c 里 include 它。

这样就两全其美了：维护方便又可以访问双向链表的内部数据结构。

最后我们来看看用迭代器实现的 invert 函数：

```
Ret invert(Iterator* forward, Iterator* backward)
{
    void* data1 = NULL;
    void* data2 = NULL;
    return_val_if_fail(forward != NULL && backward != NULL, RET_INVALID_PARAMS);
    for(; iterator_offset(forward) < iterator_offset(backward); iterator_next(forward),
iterator_prev(backward))
    {
        iterator_get(forward, &data1);
        iterator_get(backward, &data2);
        iterator_set(forward, data2);
        iterator_set(backward, data1);
    }
    return RET_OK;
}
```

invert 算法同时适用于双向链表和动态数组，而且它们的运行效率相差不大，至少对于双向链表来说已经是比较高效的实现了。

迭代器模式是一种重要的模式，在 C＋＋的标准模板库(STL)中有大量的应用。老的 C 程序员通常是运行效率至上的，所以要在优雅的设计和性能之间做 出选择时，他们通常选择后者，所以很少看到 C 语言实现的容器提供迭代器，那也不足为奇。重要的是我们学习这种方法，并在适当的时候运用它。

本节示例代码请到**这里**下载。

前面我们通过容器接口抽象了双向链表和动态数组，这样队列的实现就不依赖于具体的容器了。但是作为队列的使用者，它仍然要在编译时决定使用哪个容器。队列的测试程序就是队列的使用者之一，它的实现代码如下：

```
    Queue* queue = queue_create(linear_container_dlist_create(NULL, NULL));
    for(i = 0; i < n; i++)
    {
        assert(queue_push(queue, (void*)i) == RET_OK);
```

```
        assert(queue_head(queue, (void**)&ret_data) == RET_OK);
        assert(queue_length(queue) == (i+1));
    }
  ...
```

这里必须明确指定是 linear_container_dlist_create 还是 linear_container_darray_create,假设使用者想要换一种容器，那还是要修改代码并重新编译才行。现在我们思考另外一个问 题，如何让使用者(如这里的测试程序)想换一种容器时，不需要修改代码也不需要重新编译。

在继续学习之前，我们先介绍几个概念：

静态库：在 Linux 下，静态库的扩展名为.a，a 代表 archive 的意思。正常情况下一个 C 源文件编译之后生成一个目标文件(.o)，目标文件 里存放的是程序的机器指令和数据，它不包含运行时的信息，所以不能直接运行。用命令 ar 把多个目标文件打包成一个文件就生成了所谓的静态库。可执行文件链 接静态库时，会把用到的函数和数据拷贝过去。多个可执行文件链接同一个静态库时，所用到的函数和数据就会拷贝多次，那就存在不少空间浪费。

共享库：顾名思义，共享库可以在多个可执行文件之间共享，链接时不用拷贝函数或数据，只是建立一个函数链接表(PLT)，在运行时通过这个表来确定 具体调用的函数。共享库可以有效的避免空间浪费，但它也不是免费的午餐，它在加载时存在额外的开销，在链接多个共享库时会比较明显。不过目前出现的 prelink 和 gnu hash 等技术，有效的缓解了这个问题。共享库另外一个好处就是可以单独升级，理论上，修改某个共享库不需要重新编译依赖它的应用程序(不过实际操作时与 它的接口变化和信息隐藏程度有关)。

可执行文件：Linux 下加 x 属性的文件都是可执行文件，这里可执行文件特指 ELF(Executable and Linking Format)格式的可执行文件。 可执行文件在编译时连接静态库，会把所用到的函数会被拷贝过来，运行时不再依赖于静态库。在编译时链接共享库，它不拷贝函数和数据，但在运行时会依赖于其 链接的共享库。

回到前面的问题：我们发现在编译时，无论是链接静态库还是共享库，它都绑定了调用者与使用者之间的关系。眞正要在运行时决定而不是编译时决定使用哪种容器，那就不能包含具体容器的头文件，链接具体容器所在的库了。今天我们要学习一种新的技术：在运行时动态加载共享库。

除了一些嵌入式环境下使用的实时操作系统(RTOS)外，现代操作系统都支持在运行时动态加载共享库的机制。Linux 下有 dlopen 系列函数，Windows 下有 LoadLibrary 系列函数，其它平台也有类似的函数。下面是使用 dlopen 的简单示例：

```
#include <stdio.h>
#include <stdlib.h>
#include <dlfcn.h>
int main(int argc, char **argv)
{
    void *handle = NULL;
    double (*cosine)(double) = NULL;
    /*加载共享库*/
    handle = dlopen("libm.so", RTLD_LAZY);
    /*通过函数名找到函数指针*/
```

```
    *(void **) (&cosine) = dlsym(handle, "cos");
    /*调用函数*/
    printf("%f\n", (*cosine)(2.0));
    /*卸载共享库*/
    dlclose(handle);
    return 0;
}
```

由于这些函数在每个平台的名称和参数都有所不同，直接使用这些函数会带来可移植性的问题，为此有必要对它们进行包装。不同平台有不同的函数，也就是 说存在多种不同的实现，那这是否意味着要用接口呢？答案是不用。原因是同一个平台只一种实现，而且不会出现潜在的变化。我们要做的只是加一个适配层，用它 来隔离不同平台就好了。

我们把这个对加载函数的适配层称为 Module，声明(module.h)如下：

```
struct _Module;
typedef struct _Module Module;
typedef enum _ModuleFlags
{
    MODULE_FLAGS_NONE,
    MODULE_FLAGS_DELAY = 1
}ModuleFlags;
Module* module_create(const char* file_name, ModuleFlags flags);
void*   module_sym(Module* thiz, const char* func_name);
void    module_destroy(Module* thiz);
```

这里它们的实现只是对 dl 系列函数做个简单包装。由于不同平台有不同的实现，为了维护方便，我们把不同的实现放在不同的文件中，比如 Linux 的实现放在 module_linux.c 里。

```
Module* module_create(const char* file_name, ModuleFlags flags)
{
    Module* thiz = NULL;
    return_val_if_fail(file_name != NULL, NULL);
    if((thiz = malloc(sizeof(Module))) != NULL)
    {
        thiz->handle = dlopen(file_name, flags & MODULE_FLAGS_DELAY ? RTLD_LAZY : RTLD_NOW);
        if(thiz->handle == NULL)
        {
            free(thiz);
            thiz = NULL;
            printf("%s\n", dlerror());
        }
    }
    return thiz;
}
```

我们再看看队列的测试程序怎么写：

```
#include "linear_container.h"
typedef LinearContainer* (*LinearContainerDarrayCreateFunc)(DataDestroyFunc data_destroy, void*
ctx);
```

```
int main(int argc, char* argv[])
{
    int i = 0;
    int n = 1000;
    int ret_data = 0;
    Queue* queue = NULL;
    Module* module = NULL;
    LinearContainerDarrayCreateFunc linear_container_create = NULL;
    if(argc != 3)
    {
        printf("%s sharelib linear_container_create\n", argv[0]);
        return 0;
    }
    module = module_create(argv[1], 0);
    return_val_if_fail(module != NULL, 0);
    linear_container_create = (LinearContainerDarrayCreateFunc)module_sym(module, argv[2]);
    return_val_if_fail(linear_container_create != NULL, 0);
    queue = queue_create(linear_container_create(NULL, NULL));
    ...
}
```

说明：

o 头文件只包含 linear_container.h，而不包含 linear_container_dlist.h。

o 通过 module_sym 获取容器的创建函数，而不是直接调用 linear_container_dlist_create。

编译时不再链接容器所在的共享库：

gcc -Wall -g module_linux.c queue.c queue_test.c -ldl -o queue_dynamic_test

运行决定要使用的容器：

./queue_dynamic_test ./libcontainer.so linear_container_dlist_create

这样一来，容器的调用者和使用者完全分隔开了。接口＋动态加载的方式也称为插件式设计，它是软件可扩展性的基础，像操作系统、办公软件、浏览器和 WEB 服务器等大型软件都无一例外的使用了这类技术。

本节示例代码请到**这里**下载。

到目前为止本书的上半部分已经完成了。在上半部分中，我们学习了基本的数据结构、算法和设计思想。在进行深入学习之前，我们把前面所写的代码整理成一个通用的函数库，这个函数可能在以后的工作中用得着。

前面我们写的 Makefile 非常简单，大概类似下面的内容：

```
all:
    gcc -Wall -g -DDARRAY_TEST darray.c -o darray_test
    gcc -Wall -g -DDLIST_TEST dlist.c -o dlist_test
    gcc -Wall -g linear_container_test.c -L./ -lcontainer -o container_test
    gcc -Wall -g invert_ng.c -DINVERT_TEST -L./ -lcontainer -o invert_ng_test
    gcc -Wall -g invert.c -DINVERT_TEST -L./ -lcontainer -o invert_test
    gcc -Wall -g -shared darray.c  dlist.c  linear_container_darray.c linear_container_dlist.c
-o libcontainer.so
clean:
```

```
rm *test *.so
```

这个简单的 Makefile，对于我们学习编程已经够了，因为我们写的函数库没有眞正的用户，只要测试程序通过就好了。但是作为一个眞正的软件包，你还要考虑下列因素：

o 依赖规则。前面我们没有写编译依赖规则，不管源文件有没有变化，反正全部编译一遍。对于小程序来说用不了多少时间，那没有问题，而一些大模块可能要花上几十分钟，那就不可接受了。

o 不同平台之间的差异。 同是支持 POSIX 标准的 unix-like 操作系统，多多少少都有些不同。即使同样是 Linux，发行版本不一样，或者 libc(除了 glibc 外还有 几种轻量级 libc)不一样，或者使用的桌面环境(如 KDE 和 GNOME)不一样，都会给编写跨平台软件来带一些困难。我们在编写软件时已经考虑到了可移 植性问题，但是去检查编译环境就也是件麻烦的事。

o 交叉编译。在嵌入式环境中，我们在 PC 上编译在设备上运行的程序，或者在 x86 上编译在 powerpc 上运行的程序，这都是属于交叉编译。交叉编译产生不同于编译机器的机器码，这要求编译时使用不同(版本)的编译器。

o 同时为多个平台编译。比如编译一个在 PC 上运行的模拟环境和在 ARM 板子上实际运行的版本。在前面的 Makefile 里，编译生成的文件全部放在一个目录下，一次只能编译一个版本，切换到另外一个版本时，你要清除所有文件重新编译。

o 调用者如何使用。一个软件包肯定不是孤独存在的，一定有另外的函数库或应用程序使用它。调用者怎么知道这个软件包安装在哪里呢？有的安装在/usr 下，有的安装在/usr/local 下，还有的安装到用户目录下。不知道安装在哪里，又怎样去链接它呢？

o Makefile 的复杂度。要用 Makefile 解决上述问题，Makefile 一定变得非常复杂。有兴趣的读者可以研究一下 Firefox、Linux 和 Android 的 Makefile，这些 Makefile 可不是一般人可以写出来的，普通开发人员通常不愿意花几周时间去写 Makefile 吧。

其实这些问题都不用担心，因为开源界的前辈们早就遇到过了，并用非常巧妙的方式解决了它们。接下来，我们一起学习 automake/autoconf，看看它是如何解决这些问题的。

系统程序员成长计划-工程管理(二)

HelloWorld

automake 比起 IDE 要复杂很多，这里我们先写一个 Hello World 例子，明白其中的基本概念后，再用它来管理实际的工程。

o 目录结构

最顶层目录名用模块名称，这里是 helloworld。
源文件放在模块下的 src 子目录里，即 helloworld/src。

这是惯例。有多个子模块时，各个子模块的源代码放在各自的目录里。

o 创建源文件

在 src 下创建源文件 main.c，内容就是一个简单的 Hello World 程序。

o 创建 Makefile 模板

创建 helloworld/Makefile.am，内容为：

```
SUBDIRS=src
```

这里只有简单的一行代码，表示其下有一个 src 的子目录，如果有多个子目录，用空格分开就行了。

创建 helloworld/src/Makefile.am，内容为：

```
bin_PROGRAMS=helloworld
helloworld_SOURCES=main.c
```

这里表示有一个可执行文件 helloworld，helloworld 由源文件 main.c 编译而来。

PROGRAMS 表示要产生的可执行文件，有多个可执行文件时，用空格分开，而 bin 表示可执行文件要安装的目录。 SOURCES 表示生成可执行文件需要的源文件，有多个源文件时，也用空格分开。

.am 扩展名是 automake 的简称，它是 automake 用来产生 Makefile.in 文件的模板。

o 创建 autoconf 的模板。

在 helloworld 下运行 autoscan，生成文件 configure.scan，把它改名为 configure.in。这是 autoconf 的模板文件，它的内容大概为：

```
#                                               -*- Autoconf -*-
# Process this file with autoconf to produce a configure script.
AC_PREREQ(2.61)
AC_INIT(FULL-PACKAGE-NAME, VERSION, BUG-REPORT-ADDRESS)
AC_CONFIG_SRCDIR([src/main.c])
AC_CONFIG_HEADER([config.h])
# Checks for programs.
AC_PROG_CC
# Checks for libraries.
# Checks for header files.
# Checks for typedefs, structures, and compiler characteristics.
# Checks for library functions.
AC_CONFIG_FILES([Makefile
                 src/Makefile])
AC_OUTPUT
```

这个文件由一系列的宏组成，这些宏最终由命令 m4 展开，得到一个脚本文件 configure。configure 的主要功能是探测系统的配置，然后 根据这些配置来产生相应的 Makefile 文件。比如 AC_PROG_CC 是用来检测编译器的，AC_CONFIG_FILES 和 AC_OUTPUT 是用来产生 Makefile 和其它数据文件的。

不过这个模板文件还不能直接使用，需要做下列修改：

把：
AC_INIT(FULL-PACKAGE-NAME, VERSION, BUG-REPORT-ADDRESS)

改为：
AC_INIT(helloworld, 0.1, xianjimli@hotmail.com)

FULL-PACKAGE-NAME 是模块的名称。

VERSION 是模块的版本号，初始版本号都用 0.1。对小模块来说用两级版本号就够了，小数点前的为主版本号，只有重大更新时才升级主版本号。小数点后的为次版本号，每次发布都应该升级它。一般升级到 0.9 后，可以继续升级到 0.10、0.11 等。

BUG-REPORT-ADDRESS 是作者或维护者的邮件地址。

再加上一行 automake 的初始化脚本：

AM_INIT_AUTOMAKE(helloworld, 0.1)

helloworld 是模块的名称。

0.1 是模块的版本号。

这里和前面的参数是重复的，AC_INIT 是初始化 autoconf 的，AM_INIT_AUTOMAKE 是初始 automake 的。在有的情况下，只是产生数据文件，而不需要编译文件时，那就不需要 AM_INIT_AUTOMAKE 了。

最后得到下面的文件：

```
#                                               -*- Autoconf -*-
# Process this file with autoconf to produce a configure script.
AC_PREREQ(2.61)
AC_INIT(helloworld, 0.1, xianjimli@hotmail.com)
AC_CONFIG_SRCDIR([src/main.c])
AC_CONFIG_HEADER([config.h])
AM_INIT_AUTOMAKE(helloworld, 0.1)
# Checks for programs.
AC_PROG_CC
# Checks for libraries.
# Checks for header files.
# Checks for typedefs, structures, and compiler characteristics.
# Checks for library functions.
AC_CONFIG_FILES([Makefile
                 src/Makefile])
AC_OUTPUT
```

o 拷贝所用到宏。

运行：aclocal

前面说了，configure.in 里是一系列的宏，这些宏由命令 m4 负责展开。m4 实际上就是 macro 的简称，4 代表 m 后面省略了 4 个字母。类似的还有 I18n (Internationalization)和 L10n(Localization)，其中的数字都是代表所省略的字母个数。

AC_PROG_CC 之类的宏是标准的宏(或说是内置的宏)，不需要我们自己去写它，但我们需要运行命令 aclocal，aclocal 把 configure.in 中所用到的 宏全部拷贝到我们的工程里来。在 helloworld 目录下运行 aclocal 之后，当前目录下出现了：

autom4te.cache 这是一个临时目录，只是用来加快宏展开的。
aclocal.m4 是 configure.in 中用到的宏的定义，有兴趣的读者可以看看。

o 产生配置头文件的模板。
运行：autoheader

配置头文件(config.h)是用来定义在 C/C++程序中可以引用的宏，像模块的名称和版本号等等。这些宏由 configure 脚本产生，但我 们要提供一个模板文件。这个模板文件可以用命令 autoheader 产生出来。在 helloworld 目录下运行 autoheader 之后，当前目录下产 生 config.h.in，一般情况不用修改它。

o 创建几个必要的文件。

README：描述模块的功能、用法和注意事项等。
NEWS：描述模块最新的动态。
AUTHORS：模块的作者及联系方式。
ChangeLog：记录模块的修改历史，它有固定的格式：
1.最新修改放在最上面。
2.对于每条记录，第一行写日期，修改者和联系方式。第二行开始以 tab 开头，再加一个星号，后面再写修改的原因和位置等。如：

```
2009-03-29 Li XianJing
    * Created
```

o 生成 Makefile.in 和所需要的脚本。
运行：automake -a

这个命令会建立 COPYING depcomp INSTALL install-sh missing 几个文件的链接，这些文件指向系统中的文件。automake 最重要的功能是以 Makefile.am 为模板产生 Makefile.in 文 件，Makefile.in 相对于 Makefile.am 要复杂很多倍了，所幸的是我们不需要了解它。

o 产生 configure 脚本。
运行：autoconf

autoconf 的功能是调用 m4 展开 configure.in 中的宏，生成 configure 脚本，这个脚本是最终运行的脚本。

o 产生最终的 Makefile。
运行：./configure

configure 有两个常用的参数：
–prefix 用来指定安装目录，Linux 下默认的安装目录是/usr/local。
–host 用于交叉编译，比如 x86 的 PC 机上编译在 ARM 板上运行的程序。

如：./configure –prefix=/home/lixianjing/work/arm-root/usr –host=arm-linux

o 编译
运行：make

o 安装
运行：make install

o 发布软件包
运行：make dist 或者 make distcheck

make dist 用来生成一个发布软件包，这里会产生一个名为 helloworld-0.1.tar.gz 的文件。通常，源代码管理系统(cvs/svn/git)中的源代码是处于开发中的，是不稳定的，而发布的软件包则是稳定的，可供用户使用的。

怎样，是不是有点晕了？这里主要是想读者了解其中的原理，在实际操作中，我们可以把 make 之前的部分动作放到一个脚本文件中，这个脚本文件通常取名为 autogen.sh 或者 bootstrap。

系统程序员成长计划-工程管理(三)

函数库

现在我们用 automake 来管理我们前面所建立的函数库，这是一个基础的函数库，我们就把它命名为 base 吧。

o 目录结构
base 根目录
base/src 源代码目录

o 创建 Makefile 模板
base/Makefile.am 内容为：
SUBDIRS=src

base/src/Makefile.am 内容为：

```
lib_LTLIBRARIES=libbase.la
libbase_la_SOURCES= darray.c \
    darray.h \
    dlist.c \
    dlist.h \
    darray_iterator.h \
    dlist_iterator.h \
    hash_table.c \
    hash_table.h \
    invert.c \
    iterator.h \
    linear_container_darray.c \
    linear_container_darray.h \
    linear_container_dlist.c \
    linear_container_dlist.h \
    linear_container.h \
    queue.c \
    queue.h \
    sort.c \
    sort.h \
```

```
    stack.c \
    stack.h \
    typedef.h
libbase_la_LDFLAGS=-lpthread
noinst_PROGRAMS=darray_test dlist_test
darray_test_SOURCES=darray.c
darray_test_CFLAGS=-DDARRAY_TEST
dlist_test_SOURCES=dlist.c
dlist_test_CFLAGS=-DDLIST_TEST
basedir=$(includedir)/base
base_HEADERS=darray.h dlist.h iterator.h linear_container_dlist.h typedef.h \
    darray_iterator.h  dlist_iterator.h  linear_container_darray.h  \
    linear_container.h
EXTRA_DIST=\
    linear_container_test.c \
    invert_ng.c \
    darray_iterator.c \
    dlist_iterator.c \
    test_helper.c
```

LTLIBRARIES 是关键字。LT 代表 libtool，libtool 是用来封装共享库在不同平台上差异的脚本，其具体实现我们不用关心。

libbase.la 是函数库的名称，扩展名用.la 而不是.so 或.a，同时会生成共享库和静态库。
libbase_la_SOURCES 是生成 libbase.la 所需要的源文件。
LDFLAGS 是关键字，用来指定链接时需要的参数，-lpthread 表示要链接 libpthread.so。

noinst_PROGRAMS 是关键字，表示不需要安装的可执行文件，通常是测试程序。为了简单明了，这里没有写出全部的测试程序。

CFLAGS 是关键字，用来指定编译和预处理时的参数。

HEADERS 是关键字，列出所要安装的头文件。xxx_HEADERS 和 xxxdir 要配套使用，后者表示要安装的位置。这里在 base_HEADERS 中列出的头文件会安装到 basedir 目录里。

o 创建 autoconf 的模板。
运行：
autoscan
mv configure.scan configure.in

然后按前面介绍的方法修改 configure.in，得到下面的内容：

```
AC_PREREQ(2.61)
AC_INIT(base, 0.1, xianjimli@hotmail.com)
AC_CONFIG_SRCDIR([src/invert.c])
AC_CONFIG_HEADER([config.h])
AM_INIT_AUTOMAKE(base, 0.1)
# Checks for programs.
AC_PROG_CC
AC_PROG_LIBTOOL
# Checks for libraries.
```

```
# FIXME: Replace `main' with a function in `-lpthread':
AC_CHECK_LIB([pthread], [main])
# Checks for header files.
AC_HEADER_STDC
AC_CHECK_HEADERS([stdlib.h string.h unistd.h])
# Checks for typedefs, structures, and compiler characteristics.
AC_C_INLINE
AC_TYPE_SIZE_T
# Checks for library functions.
AC_FUNC_MALLOC
AC_FUNC_REALLOC
AC_CONFIG_FILES([Makefile
                 src/Makefile])
AC_OUTPUT
```

与前面不同的是：
AC_PROG_LIBTOOL 用来检查 libtool 脚本。
AC_CHECK_LIB 用来检查共享库是否存在。
AC_CHECK_HEADERS 用来检查头文件是否存在。
AC_FUNC_MALLOC 用来检查标准的 malloc 函数是否存在。

o 收集用到的 m4 宏。
运行：aclocal

o 产生配置头文件的模板。
运行：autoheader

o 创建 README、NEWS、ChangeLog 和 AUTHORS 几个文件。

o 生成 libtool 需要的文件。
运行：libtoolize –force –copy

这个命令的主要功能是生成 ltmain.sh，而 ltmain.sh 用来产生 libtool 脚本。

o 生成 Makefile.in 和需要的脚本。
运行：automake -a

o 产生 configure 脚本。
运行：autoconf

o 产生最终的 Makefile。
运行：./configure –prefix=$HOME/usr

o 编译运行：make

o 安装
运行：make install

o 发布软件包
运行：make dist

我们编译好的文件安装到/home/lixianjing/usr/lib/目录下了：
libbase.a libbase.la libbase.so libbase.so.0 libbase.so.0.0.0

静态库：libbase.a
动态库：libbase.so
libtool 的包装：libbase.la

头文件和库都安装好了，调用者还需要知道下列信息才能使用：

头文件和库安装在哪里？
还依赖哪些其它模块？

为了解决这个问题，我们需要借助另外一个名为 pkg-config 的工具。pkg 是 package 的简写，pkg-config 负责查询指定软件包 的配置信息，如软件包的名称、说明、版本号、头文件、库和依赖关系等等。为了让 pkg-config 能正常工作，软件包的实现者需要提供一个扩展名为 pc 的配置文件。

系统中的 pkg-config 配置文件通常放在/usr/lib/pkgconfig/和/usr/local/lib/pkgconfig/下，下面是 gtk+-2.0.pc：

```
prefix=/usr
exec_prefix=/usr
libdir=/usr/lib
includedir=/usr/include
target=x11
gtk_binary_version=2.10.0
gtk_host=i386-redhat-linux-gnu
Name: GTK+
Description: GIMP Tool Kit (${target} target)
Version: 2.12.10
Requires: gdk-${target}-2.0 atk cairo
Libs: -L${libdir} -lgtk-${target}-2.0
Cflags: -I${includedir}/gtk-2.0
```

前面部分是定义的一些变量，后面是一些关键字：

Name: 名称
Description: 功能描述
Version: 版本号
Requires: 所依赖的软件包
Libs: 调用者的链接参数。
Cflags: 调用者的编译参数。

由于 prefix 之类的变量是在软件包 configure 时才决定的，不能直接写死在 pc 文件中。我们可以让 configure 根据模板文件来产生。

模板文件名为 base.pc.in，内容为：

```
prefix=@prefix@
exec_prefix=${prefix}
```

```
libdir=${prefix}/lib
includedir=${prefix}/include
Name: @PACKAGE_NAME@
Description: a basic library.
Version: @VERSION@
Requires:
Libs: -L${libdir} -lbase
Cflags: -I${includedir}/base
```

这个模板文件和 Makefile.in 的替换规则一样，用两个@@括起来的变量会替换成 configure 检测出来的值，@prefix@等变量是标准的变量。

修改一下 base/Makefile.am，增加下列两行代码：

```
pkgconfigdir=${libdir}/pkgconfig
pkgconfig_DATA=base.pc
```

这是安装数据文件的方法，pkgconfig 不是关键字，取个描述性的名称就好了。dir 和_DATA 是关键字，它们有相同的前缀，前者表示安装的目录，后者表示要安装的文件。按照惯例，pc 文件安装到${libdir}/pkgconfig 下。

修改 configure.in，增加输入出文件 base.pc

```
AC_OUTPUT([base.pc])
```

放到 AC_CONFIG_FILES 也可以，它告诉 configure 脚本要产生的文件。

重新运行 configure 后会生成 base.pc，内容为：

```
prefix=/home/lixianjing/usr/local
exec_prefix=${prefix}
libdir=${prefix}/lib
includedir=${prefix}/include
Name: base
Description: a basic library.
Version: 0.1
Requires:
Libs: -L${libdir} -lbase
Cflags: -I${includedir}/base
```

(prefix 与 configure 时指定的 prefix 参数一致。)

在下一节中，我们再学习调用者如何使用 pc 文件。

共享内存

大家都知道，进程的地址空间是独立的，它们之间互不影响。比如同样地址为 0xabcd1234 的内存，在不同的进程中，它们的数据是完全不同的。这 样做的好处有：首先是每个进程的地址空间变大了，让编写程序更为容易。其次是一个进程崩溃了，不会影响其它进程，提高了系统的稳定性。

要做到进程的地址空间独立，光靠软件是难以实现的，通常还依赖于硬件的帮助。这种硬件称为 MMU(Memory Manage Unit)，即所谓的内存管理单元。在这种体系结构下，内存分为

物理内存和虚拟内存两种。物理内存就是实际的内存，机器上装了多大内存条就有多大的物理内 存。而应用程序使用的是虚拟内存，访问内存数据时，由 MMU 根据页表把虚拟内存地址转换对应的物理内存地址。

MMU 把各个进程的虚拟内存映射到不同的物理内存上，这样就能保证进程的虚拟内存地址是独立的。由于物理内存远远少于各个进程的虚拟内存的总和，操 作系统会把暂时不用的内存数据写到磁盘上去，把腾出来的物理内存分配给有需要的进程使用。一般会创建一个专门的分区存放换出的内存数据，这个分区称为交换 分区。

从虚拟内存到物理内存的映射并不是一个字节一个字节映射的，而是以一个称为页(page)最小单位的进行的。页的大小视具体硬件平台而定，通常是 4K。当应用程序访问的虚拟内存的页面不在物理内存里时，MMU 产生一个缺页中断，并挂起当前进程，缺页中断处理函数负责把相应的数据从磁盘读入内存，然 后唤醒挂起的进程。

进程地址空间独立有它的好处，但我们需要在进程之间共享数据，比如两个进程共享同一段可执行代码或者共享其它数据，这就需要共享内存。实现共享内存非常容易，只要把两个进程的虚拟内存页面映射同一个物理内存页面就行了。

在程序中使用共享内存非常简单，操作系统或者函数库提供了一些 API。如 Linux 提供了：

void * mmap(void *start, size_t length, int prot , int flags, int fd, off_t offset);
int munmap(void *start, size_t length);

mmap 是 Linux 下创建共享内存最正统的方法，其它共享内存接口都是以它为基础进行包装的。 mmap 可以通过参数 start 指定映射的地址，但是这个不能保证成功，实际的地址应以返回值为准。如果需要在不同进程之间传递共享内存的地址，最好是在所有进程中，把共享内 存映射到同样的地址，为了保证映射成功，需要精心选择一些不常用的内存地址，并在程序初始化时就映射好。

示例：

下面我们用下循环队列(FIFO Ring)，在两个进程之间传递数据，由于循环队列在一个读一个写的情况下不需要加锁，暂时我们可以避开进程间同步的问题。

o 创建共享内存

```
static int g_shmem_fd = -1;
void* shmem_alloc(size_t size)
{
    g_shmem_fd = open("/tmp/shmem_demo", O_RDWR);
    if(g_shmem_fd < 0)
    {
        char* buffer = calloc(size, 1);
        g_shmem_fd = open("/tmp/shmem_demo", O_RDWR | O_CREAT, 0640);
        write(g_shmem_fd, buffer, size);
        free(buffer);
    }
    void* addr =  mmap(0, size, PROT_READ | PROT_WRITE, MAP_SHARED, g_shmem_fd, 0);
    return addr;
}
```

o 释放共享内存

```
void shmem_free(void* addr, size_t size)
{
    munmap(addr, size);
    close(g_shmem_fd);
    return;
}
```

这里用了一个全局变量 g_shmem_fd 来保存文件描述符，这里不好的，在后面的内存管理器中，我们会把它封装到内存管理器。

o 创建循环队列

```
FifoRing* fifo_ring_create(size_t length)
{
    FifoRing* thiz = NULL;
    return_val_if_fail(length > 1, NULL);
    thiz = (FifoRing*)shmem_alloc(sizeof(FifoRing) + length * sizeof(void*));
    if(thiz != NULL)
    {
        if(thiz->inited == 0)
        {
            thiz->r_cursor = 0;
            thiz->w_cursor = 0;
            thiz->length   = length;
            thiz->inited = 1;
        }
    }
    return thiz;
}
```

这里的 shmem_alloc 代替了以前的 malloc。

o 销毁循环队列

```
void fifo_ring_destroy(FifoRing* thiz)
{
    if(thiz != NULL)
    {
        shmem_free(thiz, sizeof(FifoRing) + thiz->length * sizeof(void*));
    }
    return;
}
```

这里的 shmem_free 代替了以前的 free。

循环队列的其它函数实现不变，有兴趣的读者可以参考完整的示例代码。

本节示例代码请到这里下载。

内存管理器

在前面学习共享内存的时候，我们重新实现了循环队列，两个实现的不同之处只是在于内存分配和释放上。对比一下 fifo_ring_create 的实现：

第一种实现用 malloc 分配内存。

```
FifoRing* fifo_ring_create(size_t length)
{
    FifoRing* thiz = NULL;
    return_val_if_fail(length > 1, NULL);
    thiz = (FifoRing*)malloc(sizeof(FifoRing) + length * sizeof(void*));
    if(thiz != NULL)
    {
        thiz->r_cursor = 0;
        thiz->w_cursor = 0;
        thiz->length   = length;
    }
    return thiz;
}
```

第二种实现用 shmem_alloc 分配内存。

```
FifoRing* fifo_ring_create(size_t length)
{
    FifoRing* thiz = NULL;
    return_val_if_fail(length > 1, NULL);
    thiz = (FifoRing*)shmem_alloc(sizeof(FifoRing) + length * sizeof(void*));
    if(thiz != NULL)
    {
        if(thiz->inited == 0)
        {
            thiz->r_cursor = 0;
            thiz->w_cursor = 0;
            thiz->length   = length;
            thiz->inited = 1;
        }
    }
    return thiz;
}
```

只是一行代码之差，就要把整个队列重写一遍，不符合我们前面倡导的 DRY(don’t repeat yourself)原则。这里可以看出，内存管理器有不同的实现，所以我们引入内存管理器这个接口来隔离变化。内存管理器的基本功能有：

- o 分配内存
- o 释放内存
- o 扩展/缩小已经分配的内存
- o 分配清零的内存

据此，我们定义 Allocator 接口如下：

```
struct  _Allocator;
```

```
typedef struct  _Allocator Allocator;
typedef void* (*AllocatorCallocFunc)(Allocator* thiz, size_t nmemb, size_t size);
typedef void* (*AllocatorAllocFunc)(Allocator* thiz, size_t size);
typedef void  (*AllocatorFreeFunc)(Allocator* thiz, void *ptr);
typedef void* (*AllocatorReallocFunc)(Allocator* thiz, void *ptr, size_t size);
typedef void  (*AllocatorDestroyFunc)(Allocator* thiz);
struct _Allocator
{
    AllocatorCallocFunc  calloc;
    AllocatorAllocFunc   alloc;
    AllocatorFreeFunc    free;
    AllocatorReallocFunc realloc;
    AllocatorDestroyFunc destroy;
    char priv[0];
};
```

基于 malloc 系统函数的实现：

```
static void*  allocator_normal_calloc(Allocator* thiz, size_t nmemb, size_t size)
{
    return calloc(nmemb, size);
}
...
Allocator* allocator_normal_create(void)
{
    Allocator* thiz = (Allocator*)calloc(1, sizeof(Allocator));
    if(thiz != NULL)
    {
        thiz->calloc  = allocator_normal_calloc;
        thiz->alloc   = allocator_normal_alloc;
        thiz->realloc = allocator_normal_realloc;
        thiz->free    = allocator_normal_free;
        thiz->destroy = allocator_normal_destroy;
    }
    return thiz;
}
```

这里函数与标准 C 函数一一对应，只需要简单包装就行了。

对于共享内存，通常的做法是先分配一大块内存，然后进行二次分配，此时需要编写自己的内存管理器。内存分配器是很奇特的，任何初学者都可以设计自己 的内存分配器，但同时任何高手都不敢说自己能设计出最好的内存分配器。为什么内存分配器很难写好呢？因为设计好的内存分配器需要考虑很多因素：

o 最大化兼容性

实现内存分配器时，先要定义出分配器的接口函数。接口函数没有必要标新立异，而是要遵循现有标准(如 POSIX 或者 Win32)，让使用者可以平滑的过度到新的内存分配器上。

o 最大化可移植性

通常情况下，内存分配器要向OS申请内存，然后进行二次分配，这要调用OS提供的函数才行。OS提供的函数则是因平台而异，尽量抽象出平台相关的代码，才能保证内存分配器的可移植性。

o 浪费最小的空间

内存分配器要管理内存，必然要使用一些自己的数据结构，这些数据结构本身也要占内存空间。在用户眼中，这些内存空间毫无疑问是浪费掉了，如果浪费在 内存分配器本身的内存太多，显然是不可以接受的。内存碎片也是浪费空间的罪魁祸首，内存碎片是一些不连续的小块内存，它们总量可能很大，但因为其非连续性 而无法使用，这些空间在某种程度上说也是浪费的。

o 最快的速度

内存分配/释放是常用的操作。按着2/8原则，常用函数的性能对系统的整体性能影响最大，所以内存分配器的速度越快越好。遗憾的是，最先匹配算法，最优匹配算法和最差匹配算法，谁也不能说谁更快，因为快与慢要依赖于具体的应用环境。

o 最大化可调性（以适应于不同的情况）

内存管理算法设计的难点就在于要适应不同的情况。事实上，如果缺乏应用的上下文，是无法评估内存管理算法的好坏的，因为专用算法总是在时/空性能上 有更优的表现。为每种情况都写一套内存管理算法，显然是不太合适的。我们不需要追求最优算法，那样代价太高，能达到次优就行了。设计一套通用内存管理算 法，通过一些参数对它进行配置，可以让它在特定情况也有相当出色的表现，这就是可调性。

o 最大化调试功能

作为一个C/C++程序员，内存错误可以说是我们的噩梦，上一次的内存错误一定还让你记忆犹新。内存分配器提供的调试功能，强大易用，特别对于嵌入式环境来说，内存错误检测工具缺乏的情况下，内存分配器提供的调试功能就更不可少了。

o 最大化适应性

前面说了最大化可调性，以便让内存分配器适用于不同的情况。但是，对于不同情况都要去调整配置，还是有点麻烦。尽量让内存分配器适用于很广的情况，只有极少情况下才去调整配置，这就是内存分配器的适应性。

设计是一个多目标优化的过程，而且有些目标之间存在着竞争。如何平衡这些竞争是设计的难点之一。在不同的情况下，这些目标的重要性是不一样的，所以根本不存在一个最好的内存分配器。换句话说，内存分配器的实现是变化的，要根据不同的应用环境而变化。

下面我们实现一个傻瓜型的内存管理器，按上面的标准来看，它没有什么实际的意义，但它的优点是简单，仅仅200多行代码，就展示了内存管理器的基本原理。

o 分配过程

所有空闲的内存块放在一个双向链表中，最初只有一块。分配时使用首次匹配算法，在第一个空闲块上进行分配。

```
static void*  allocator_self_manage_alloc(Allocator* thiz, size_t size)
```

```
{
    FreeNode* iter = NULL;
    size_t length = REAL_SIZE(size);
    PrivInfo* priv = (PrivInfo*)thiz->priv;
    /*查找第一个满足条件的空闲块*/
    for(iter = priv->free_list; iter != NULL; iter = iter->next)
    {
        if(iter->length > length)
        {
            break;
        }
    }
    return_val_if_fail(iter != NULL, NULL);
    /*如果找到的空闲块刚好满足需求，就从空闲块链表中移出它*/
    if(iter->length < (length + MIN_SIZE))
    {
        if(priv->free_list == iter)
        {
            priv->free_list = iter->next;
        }
        if(iter->prev != NULL)
        {
            iter->prev->next = iter->next;
        }
        if(iter->next != NULL)
        {
            iter->next->prev = iter->prev;
        }
    }
    else
    {
        /*如果找到的空闲块比较大，就把它拆成两个块，把多余的空闲内存放回去*/
        FreeNode* new_iter = (FreeNode*)((char*)iter + length);
        new_iter->length = iter->length - length;
        new_iter->next = iter->next;
        new_iter->prev = iter->prev;
        if(iter->prev != NULL)
        {
            iter->prev->next = new_iter;
        }
        if(iter->next != NULL)
        {
            iter->next->prev = new_iter;
        }
        if(priv->free_list == iter)
        {
            priv->free_list = new_iter;
        }
        iter->length = length;
    }
```

```
    /*返回可用的内存*/
    return (char*)iter + sizeof(size_t);
}
```

这里对空闲块的管理不占用有效内存空间，它只是强制的把空闲块转换成 FreeNode 结构，这要求任何空闲块的大小都不小于 sizeof(FreeNode)。

o 释放内存

释放时把内存块放回空闲链表，然后对相邻居的内存块进行合并。

```
static void   allocator_self_manage_free(Allocator* thiz, void *ptr)
{
    FreeNode* iter = NULL;
    FreeNode* free_iter = NULL;
    PrivInfo* priv = (PrivInfo*)thiz->priv;
    return_if_fail(ptr != NULL);
    free_iter = (FreeNode*)((char*)ptr - sizeof(size_t));
    free_iter->prev = NULL;
    free_iter->next = NULL;
    if(priv->free_list == NULL)
    {
        priv->free_list = free_iter;
        return;
    }
    /*根据内存块地址的大小，把它插入到适当的位置。*/
    for(iter = priv->free_list; iter != NULL; iter = iter->next)
    {
        if((size_t)iter > (size_t)free_iter)
        {
            free_iter->next = iter;
            free_iter->prev = iter->prev;
            if(iter->prev != NULL)
            {
                iter->prev->next = free_iter;
            }
            iter->prev = free_iter;
            if(priv->free_list == iter)
            {
                priv->free_list = free_iter;
            }
            break;
        }
        if(iter->next == NULL)
        {
            iter->next = free_iter;
            free_iter->prev = iter;
            break;
        }
    }
    /*对相邻居的内存进行合并*/
```

```
    allocator_self_manage_merge(thiz, free_iter);
    return;
}
```

有了 Allocator 接口，我们也可以通过装饰模式，为内存管理器加上越界/泄露检查等其它辅助功能。下面的 Allocator 是检查内存越界的装饰。

o 实现函数

```
static void*  allocator_checkbo_alloc(Allocator* thiz, size_t size)
{
    char* ptr = NULL;
    size_t total = size + 3 * sizeof(void*);
    PrivInfo* priv = (PrivInfo*)thiz->priv;
    if((ptr = allocator_alloc(priv->real_allocator, total)) != NULL)
    {
        *(size_t*)ptr = size;
        memset(ptr + sizeof(void*), BEGIN_MAGIC, sizeof(void*));
        memset(ptr + 2 * sizeof(void*) + size, END_MAGIC, sizeof(void*));
    }
    return ptr + 2 * sizeof(void*);
}
```

分配内存时：多分配 3 * sizeof(void*)个字节，分别用来记录内存块的长度及前后的 Magic 数据。然后调用实际的(即被装饰的)内存分配器分配内存，记录长度并填充 Magic 数据，最后返回内存指针给调用者。

```
static void   allocator_checkbo_free(Allocator* thiz, void *ptr)
{
    PrivInfo* priv = (PrivInfo*)thiz->priv;
    if(ptr != NULL)
    {
        char magic[sizeof(void*)];
        char* real_ptr = (char*)ptr - 2 * sizeof(void*);
        size_t size = *(size_t*)(real_ptr);
        memset(magic, BEGIN_MAGIC, sizeof(void*));
        assert(memcmp((char*)ptr - sizeof(void*), magic, sizeof(void*)) == 0);
        memset(magic, END_MAGIC, sizeof(void*));
        assert(memcmp((char*)ptr + size , magic, sizeof(void*)) == 0);
        allocator_free(priv->real_allocator, real_ptr);
    }
    return;
}
```

释放内存时：取得内存块的长度，然后检查前后的 magic 数据是否被修改，如果被修改则认为存在写越界的错误。

惯用手法

《POSA(面向模式的软件架构)》中根据模式粒度把模式分为三类：架构模式、设计模式和惯用手法。其中把分层模式、管道过滤器和微内核模式等归为 架构模式，把代理模式、命

令模式和出版-订阅模式等归为设计模式，而把引用计数等归为惯用手法。这三类模式间的界限比较模糊，在特定的情况下，有的设计模 式可以作为架构模式来用，有的把架构模式也作为设计模式来用。

一般来说，我们可以认为架构模式、设计模式和惯用手法，三者的重要性依次递减，毕竟整体决策比局部决策的影响面更大。但是任何整体都是局部组成的， 局部的决策也会影响整体。惯用手法的影响虽然通常是局部的，但其所起的作用仍然很重要。本文介绍几种关于内存使用的惯用手法：

1. 预分配

在前面的学习中，我们已经用到了这种手法。在实现了一个动态数组时，当向其中增加元素时，它会自动扩展(缩减)缓冲区的大小，无需要调用者关心。扩展缓冲区的过程如下：

o 先分配一块更大的缓冲区。
o 把数据从老的缓冲区拷贝到新的缓冲区。
o 释放老的缓冲区。

如果你使用 realloc 来实现，内存管理器可能会做些优化：如果老的缓冲区后面有连续的空闲空间，它只需要简单的扩展老的缓冲区，而跳过后面两个步骤。但在大多数情况下，它都要通过上述三个步骤来完成扩展。

由此可见，扩展缓冲区对调用者来说虽然是透明的，但决不是免费的。它得付出相当大的时间代价，以及由此产生的产生内存碎片问题。如果每次向 vector 中增加一个元素，都要扩展缓冲区，显然是不太合适的。

此时我们可以采用预分配机制，每次扩展时，不是需要多大就扩展多大，而是预先分配一大块内存。这一大块可以供后面较长一段时间使用，直到把这块内存全用完了，再继续用同样的方式扩展。

至于一次应该扩展多大，经验值是现有大小的 1.5 倍。这个经验值基于这样的假设：现在用得多意味着将来会用得更多。

预分配机制多见于一些带 buffer 的容器实现中，比如像 vector 和 string 等。

2. 对象引用计数

在面向对象的系统中，对象之间的协作关系非常复杂。所谓协作其实就调用对象的函数或者向对象发送消息，但不管调用函数还是发送消息，总是要通过某种方式知道目标对象才行。而最常见的做法就是保存目标对象的引用（指针）。

对象被别人引用了，但自己可能并不知道。此时麻烦就来了，如果对象被销毁了，对该对象的引用就变成了野针，系统随时可能因此而崩溃。不释放也不行，因为那样会出现内存泄露。怎么办呢？

此时我们可以采用对象引用计数，对象有一个引用计数器，不管谁要引用这个对象，就要把对象的引用计数器加 1，如果不再该引用了，就把对象的引用计数 器减 1。当对象的引用计数器被减为 0 时，说明没有其它对象引用它，该对象就可以安全的销毁了。这样，对象的生命周期就得到了有效的管理。

还有一种引用称为弱引用，它不增加对象的引用计数，只是要求对象在销毁时通知引用者。实现方法是调用者注册一个回调函数，在对象销毁时调用。

对象引用计数运用相当广泛。像在 Windows 下 COM(组件对象模型)和 GNOME 的 glib 里，对象引用计数都是作为对象系统的基本设施之一。

3. 写时拷贝(COW)

OS 内核创建子进程的过程是最常见而且最有效的写时拷贝(COW) 例子：创建子进程时，子进程要继承父进程内存空间中的数据。但继承之后，两者各自有独立的内存空间，修改各自的数据不会互相影响。

要做到这一点，最简单的办法就是直接把父进程的内存空间拷贝一份。这样做可行，但问题在于拷贝内容太多，无论是时间还是空间上的开销都让人无法接 受。况且，在大多数情况下，子进程只会使用少数继承过来的数据，而且多数是读取，只有少量是修改，也就说大部分拷贝的动作白做了。怎么办呢？

此时可以采用写时拷贝(COW)，COW 代表 Copy on Write。最初的拷贝只是个假象，并不是真正的拷贝，只是把引用计数加 1，并设置适当的标志。如果双方都只是读取这些数据，那好办，直接读就行了。而任 何一方要修改时，为了不影响另外一方，它要把数据拷贝一份，然后修改拷贝的这一份。也就是说在修改数据时，拷贝动作才真正发生。

当然，在真正拷贝的时候，你可以选择拷贝一部分数据或者全部数据。在上面的例子中，由于内存通常是按页面来管理的，拷贝时只拷贝相关的页面，而不是拷贝整个内存空间。

写时拷贝(COW)对性能上的贡献很大，差不多任何带 MMU 的 OS 都会采用。当然它不限于内核空间，在用户空间也可以使用，比如像一些 String 类的实现也采用了这种方法。

4. 固定大小分配

频繁的分配大量小块内存是设计内存管理器的挑战之一。

首先是空间利用率上的问题：由于内存管理本身的需要一些辅助内存，假设每块内存需要 16 字节用作辅助内存，那么即使只要分配 4 个字节这样的小块内存，仍然要分配 16 字节的内存。一块小内存不要紧，若存在大量小块内存，所浪费的空间就可观了。

其次是内存碎片问题：频繁分配大量小块内存，很容易造成内存碎片问题。这不但降低内存管理器的效率，同时由于这些内存不连续，虽然空闲却无法使用。

此时可以采用固定大小分配，这种方式通常也叫做缓冲池(pool)分配。缓冲池(pool)先分配一块或者多块连续的大块内存，把它们分成 N 块大小 相等的小块内存，然后进行二次分配。由于这些小块内存的大小是固定的，内存管理的开销就非常小了，往往只要一个标识位用于标识该单元是否空闲，或者甚至连 标识位都不需要。

缓冲池(pool)中所有这些小块内存分布在一块或者几块连接内存上，所以不会有内存碎片问题。

固定大小分配手法运用比较广泛，差不多所有的内存管理器都用这种方法来对付小块内存分配，比如 glibc、STLPort 和 linux 的 slab 等。

5. 会话缓冲池分配(Session Pool)

服务器要长时间运行，内存泄露是它的威胁之一，任何小概率的内存泄露，都可能会累积到具有破坏性的程度。从它们的运行模式来看，它们总是不断的重复某个过程，而在这个过程中，又要分配大量(次数)内存。

比如像 WEB 服务器，它不断的处理 HTTP 请求，我们把一次 HTTP 请求，称为一次会话。一次会话要经过多个阶段，在这个过程中要做各种处理，要多次分配内存。由于处理比较复杂，分配内存的地方又比较多，内存泄露可以说防不甚防。

针对这种情况，我们可以采用会话缓冲池分配。它基于多次分配一次释放的策略，在过程开始时创建会话缓冲池(Session Pool)，这个过程中所有内存分配都通过会话缓冲池(Session Pool)来分配，当这个过程完成时，销毁掉会话缓冲池(Session Pool)，即释放这个过程中所分配的全部内存。

因为只需要释放一次，内存泄露的可能大大降低了。会话缓冲池分配并不是太常见，apache 采用的这种用法。后来自己用过两次，感觉效果不错。

还有一些关于内存使用的惯用手法，这里不再多说了，有兴趣的读者可以参考相关资料。

变参函数的实现原理

C 语言要求函数调用者按照函数原型进行调用，如果调用参数与函数原型不一致，编译器就会发出警告。而变参函数的参数是不确定的，它允许同一个函数有 多种不同的参数组合，编译器不会对可变部分的参数做类型检查，因而在使用的时候拥有较大的灵活性(当然也容易出错)。本节我们将一起研究一下变参函数的实 现原理，先看一个例子程序：

o 使用变参函数，需要 libc 库支持，头文件 stdarg.h 里提供一些必要的宏定义。

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

o 实现变参函数。

```
int accumlate(int nr, ...)
{
    int i = 0;
    int result = 0;
    va_list arg = NULL;
    va_start(arg, nr);
    for(i = 0; i < nr; i++)
    {
        result += va_arg(arg, int);
    }
    va_end(arg);
    return result;
}
```

变参函数可变参数用…来表示。这里 accumlate 把多个整数累加起来，参数的个数由 accumlate 的第一个参数决定，然后返回计算结果。

o 调用变参函数。

```
int main(int argc, char* argv[])
{
    printf("%d\n", accumlate(1, 100));
    printf("%d\n", accumlate(2, 100, 200));
    printf("%d\n", accumlate(3, 100, 200, 300));
    return 0;
}
```

这里以三种方式调用了 accumlate。

在上面的例子中，va_list/ va_start/ va_arg/ va_end 几个宏搞定了所有的实现细节，它们是怎么实现的呢，我们先看看参数在内存里的布局：

用 gdb 调试上述程序，在函数 accumlate 里设置断点，然后显示 nr 相邻区域的数据：

Breakpoint 1, accumlate (nr=3) at varg.c:13
13 int i = 0;
(gdb) x /4w &nr
0xbf904440: 0×00000003 0×00000064 0×000000c8 0×0000012c

C 语言函数调用时，参数按值传递，并从最后一个参数开始压栈。先压入最后一个参数，再压入倒数第二参数，最后压入第一个参数。由于栈是向下增长的，也就是先压入的参数放在高地址，后压入的参数放在低地址。所以这里：

地址	内容	说明
0xbf904440	3	第一个参数
0xbf904444	100	第二个参数
0xbf904448	200	第三个参数
0xbf90444c	300	第四个参数

(这里的具体地址与运行环境有关，因操作系统而异)

这样一来，只要我们知道可变参数的前一个参数，就可以依次取出后面的参数了。

在前面的例子中，这两行代码让 arg 指向了第一个变参数：

```
    va_list arg = NULL;
    va_start(arg, nr);
```

va_list 是一个指针，由于参数的类型是不定的，它可以指向任意类型的指针，我们这样定义它：

```
#define va_list void*
```

为了让 arg 指向第一个可变参数，可以通过 nr 的地址加上 nr 的数据类型大小就行了，我们这样定义 va_start：

#define va_start(arg, start) arg = (va_list)(((char*)&(start)) + sizeof(start))

在前面的例子中，这行代码让 arg 指向了下一个变参数：

```
result += va_arg(arg, int)
```

为了让arg指向下一个可变参数，可以通过当前可变参数的地址加上当前可变参数的数据类型大小就行了，我们这样定义 va_arg：

```
#define va_arg(arg, type)    *(type*)arg; arg = (char*)arg + sizeof(type);
```

可变参数的实现原理简单吧，所以不用标准C的支持，我们也可以实现变参函数：

```
#include <stdio.h>
#define va_list void*
#define va_end(arg)
#define va_arg(arg, type)    *(type*)arg; arg = (char*)arg + sizeof(type);
#define va_start(arg, start) arg = (va_list)(((char*)&(start)) + sizeof(start))
int accumlate(int nr, ...)
{
    int i = 0;
    int result = 0;
    va_list arg = NULL;
    va_start(arg, nr);
    for(i = 0; i < nr; i++)
    {
        result += va_arg(arg, int);
    }
    va_end(arg);
    return result;
}
int main(int argc, char* argv[])
{
    printf("%d\n", accumlate(1, 100));
    printf("%d\n", accumlate(2, 100, 200));
    printf("%d\n", accumlate(3, 100, 200, 300));
    return 0;
}
```

对于变参函数还需要说明几点：

1.编译器优化。

如果加了编译优化标志，参数可能是通过寄存器传递的，那参数在内存里的布局与前面所展示的就不一样了。这个不用担心，编译器会处理的，它会保存变参函数所有的参数到内存里。我们可以看下ARM平台上汇编代码：

```
00000000 <accumlate>:
   0:   e92d000f        stmdb   sp!, {r0, r1, r2, r3}
   4:   e59dc000        ldr     ip, [sp]
   8:   e35c0000        cmp     ip, #0  ; 0x0
   c:   d3a00000        movle   r0, #0  ; 0x0
  10:   da000008        ble     38 <accumlate+0x38>
  14:   e3a00000        mov     r0, #0  ; 0x0
  18:   e28d1008        add     r1, sp, #8      ; 0x8
  1c:   e1a02000        mov     r2, r0
  20:   e5113004        ldr     r3, [r1, #-4]
```

```
  24:   e2822001        add     r2, r2, #1      ; 0x1
  28:   e15c0002        cmp     ip, r2
  2c:   e0800003        add     r0, r0, r3
  30:   e2811004        add     r1, r1, #4      ; 0x4
  34:   1afffff9        bne     20 <accumlate+0x20>
  38:   e28dd010        add     sp, sp, #16     ; 0x10
  3c:   e12fff1e        bx      lr
```

r0, r1, r2 和 r3 四个寄存器通常用来传递函数的前面四个参数，编译器在这里对此做了特殊处理，用 stmdb sp!, {r0, r1, r2, r3}这条指令把 r0 到 r3 的四个寄存器的值保存到内存里了。

2.关于参数结束标识的问题。

变参函数的参数个数是变化的，怎么知道实际参数的个数呢？通常的做法有三种：

o 指定参数的个数。比如这里 accumlate 的第一个参数指明了参数的个数。
o 用固定值(如-1 或 NULL)表示最后一个参数。
o 用格式化字符串。比如 printf 使用了格式化字符串。

3.变参函数至少要提供一个参数。

这很明显，没有可变参数前面的一个参数，我们是无法取得第一个可变参数的地址的。

谁在 call 我-backtrace 的实现原理

显示函数调用关系(backtrace/callstack)是调试器必备的功能之一，比如在 gdb 里，用 bt 命令就可以查看 backtrace。 在程序崩溃的时候，函数调用关系有助于快速定位问题的根源，了解它的实现原理，可以扩充自己的知识面，在没有调试器的情况下，也能实现自己 backtrace。更重要的是，分析 backtrace 的实现原理很有意思。现在我们一起来研究一下：

glibc 提供了一个 backtrace 函数，这个函数可以帮助我们获取当前函数的 backtrace，先看看它的使用方法，后面我们再仿照它写一个。

```
#include <stdio.h>
#include <stdlib.h>
#include <execinfo.h>
#define MAX_LEVEL 4
static void test2()
{
        int i = 0;
        void* buffer[MAX_LEVEL] = {0};
        int size = backtrace(buffer, MAX_LEVEL);
        for(i = 0; i < size; i++)
        {
                printf("called by %p\n",        buffer[i]);
        }
        return;
}
static void test1()
{
        int a=0x11111111;
        int b=0x11111112;
```

```
        test2();
        a = b;
        return;
}
static void test()
{
        int a=0x10000000;
        int b=0x10000002;
        test1();
        a = b;
        return;
}
int main(int argc, char* argv[])
{
        test();
        return 0;
}
```

编译运行它：
gcc -g -Wall bt_std.c -o bt_std
./bt_std

屏幕打印：
called by 0×8048440
called by 0×804848a
called by 0×80484ab
called by 0×80484c9

上面打印的是调用者的地址，对程序员来说不太直观，glibc 还提供了另外一个函数 backtrace_symbols，它可以把这些地址转换成源 代码的位置(通常是函数名)。不过这个函数并不怎么好用，特别是在没有调试信息的情况下，几乎得不什么有用的信息。这里我们使用另外一个工具 addr2line 来实现地址到源代码位置的转换：

运行：
./bt_std |awk ‘{print “addr2line “$3″ -e bt_std”}’>t.sh;. t.sh;rm -f t.sh

屏幕打印：
/home/work/mine/sysprog/think-in-compway/backtrace/bt_std.c:12
/home/work/mine/sysprog/think-in-compway/backtrace/bt_std.c:28
/home/work/mine/sysprog/think-in-compway/backtrace/bt_std.c:39
/home/work/mine/sysprog/think-in-compway/backtrace/bt_std.c:48

backtrace 是如何实现的呢？ 在 x86 的机器上，函数调用时，栈中数据的结构如下：

```
-------------------------------------------
参数N
参数…       函数参数入栈的顺序与具体的调用方式有关
参数 3
参数 2
参数 1
-------------------------------------------
EIP         完成本次调用后，下一条指令的地址
```

```
EBP          保存调用者的EBP，然后EBP指向此时的栈顶。
----------------新的EBP指向这里---------------
临时变量1
临时变量2
临时变量3
临时变量…
临时变量5
-------------------------------------------
```

(说明：下面低是地址，上面是高地址，栈向下增长的)

调用时，先把被调函数的参数压入栈中，C 语言的压栈方式是：先压入最后一个参数，再压入倒数第二参数，按此顺序入栈，最后才压入第一个参数。

然后压入 EIP 和 EBP，此时 EIP 指向完成本次调用后下一条指令的地址 ，这个地址可以近似的认为是函数调用者的地址。EBP 是调用者和被调函数之间的分界线，分界线之上是调用者的临时变量、被调函数的参数、函数返回地址 (EIP)，和上一层函数的 EBP，分界线之下是被调函数的临时变量。

最后进入被调函数，并为它分配临时变量的空间。gcc 不同版本的处理是不一样的，对于老版本的 gcc(如 gcc3.4)，第一个临时变量放在最高的 地址，第二个其次，依次顺序分布。而对于新版本的 gcc(如 gcc4.3)，临时变量的位置是反的，即最后一个临时变量在最高的地址，倒数第二个其次，依 次顺序分布。

为了实现 backtrace，我们需要：

1.获取当前函数的 EBP。
2.通过 EBP 获得调用者的 EIP。
3.通过 EBP 获得上一级的 EBP。
4.重复这个过程，直到结束。

通过嵌入汇编代码，我们可以获得当前函数的 EBP，不过这里我们不用汇编，而且通过临时变量的地址来获得当前函数的 EBP。我们知道，对于 gcc3.4 生成的代码，当前函数的第一个临时变量的下一个位置就是 EBP。而对于 gcc4.3 生成的代码，当前函数的最后一个临时变量的下一个位置就是 EBP。

有了这些背景知识，我们来实现自己的 backtrace:

```
#ifdef NEW_GCC
#define OFFSET 4
#else
#define OFFSET 0
#endif/*NEW_GCC*/
int backtrace(void** buffer, int size)
{
    int  n = 0xfefefefe;
    int* p = &n;
    int  i = 0;
    int ebp = p[1 + OFFSET];
    int eip = p[2 + OFFSET];
```

```
    for(i = 0; i < size; i++)
    {
        buffer[i] = (void*)eip;
        p = (int*)ebp;
        ebp = p[0];
        eip = p[1];
    }
    return size;
}
```

对于老版本的 gcc，OFFSET 定义为 0，此时 p+1 就是 EBP，而 p[1]就是上一级的 EBP，p[2]是调用者的 EIP。本函数总共有 5 个 int 的临时变量，所以对于新版本 gcc， OFFSET 定义为 5，此时 p+5 就是 EBP。在一个循环中，重复取上一层的 EBP 和 EIP，最终得到所有调用者的 EIP，从而实现了 backtrace。

现在我们用完整的程序来测试一下(bt.c)：

```
#include <stdio.h>
#define MAX_LEVEL 4
#ifdef NEW_GCC
#define OFFSET 4
#else
#define OFFSET 0
#endif/*NEW_GCC*/
int backtrace(void** buffer, int size)
{
        int      n = 0xfefefefe;
        int* p = &n;
        int      i = 0;
        int ebp = p[1 + OFFSET];
        int eip = p[2 + OFFSET];
        for(i = 0; i < size; i++)
        {
                buffer[i] = (void*)eip;
                p = (int*)ebp;
                ebp = p[0];
                eip = p[1];
        }
        return size;
}
static void test2()
{
        int i = 0;
        void* buffer[MAX_LEVEL] = {0};
        backtrace(buffer, MAX_LEVEL);
        for(i = 0; i < MAX_LEVEL; i++)
        {
                printf("called by %p\n",        buffer[i]);
        }
        return;
```

```
}
static void test1()
{
        int a=0x11111111;
        int b=0x11111112;
        test2();
        a = b;
        return;
}
static void test()
{
        int a=0x10000000;
        int b=0x10000002;
        test1();
        a = b;
        return;
}
int main(int argc, char* argv[])
{
        test();
        return 0;
}
```

写个简单的 Makefile:

```
CFLAGS=-g -Wall
all:
    gcc34 $(CFLAGS) bt.c -o bt34
    gcc $(CFLAGS) -DNEW_GCC  bt.c -o bt
    gcc $(CFLAGS) bt_std.c -o bt_std
clean:
    rm -f bt bt34 bt_std
```

编译然后运行：
make
./bt|awk '{print "addr2line "$3" -e bt"}'>t.sh;. t.sh;

屏幕打印：
/home/work/mine/sysprog/think-in-compway/backtrace/bt.c:37
/home/work/mine/sysprog/think-in-compway/backtrace/bt.c:51
/home/work/mine/sysprog/think-in-compway/backtrace/bt.c:62
/home/work/mine/sysprog/think-in-compway/backtrace/bt.c:71

对于可执行文件，这种方法工作正常。对于共享库，addr2line 无法根据这个地址找到对应的源代码位置了。原因是：addr2line 只能通过 地址偏移量来查找，而打印出的地址是绝对地址。由于共享库加载到内存的位置是不确定的，为了计算地址偏移量，我们还需要进程 maps 文件的帮助：

通过进程的 maps 文件(/proc/进程号/maps)，我们可以找到共享库的加载位置，如：
…
00c5d000-00c5e000 r-xp 00000000 08:05 2129013 /home/work/mine/sysprog/think-in-

compway/backtrace/libbt_so.so
00c5e000-00c5f000 rw-p 00000000 08:05 2129013 /home/work/mine/sysprog/think-in-compway/backtrace/libbt_so.so
…

libbt_so.so 的代码段加载到 0×00c5d000-0×00c5e000，而 backtrace 打印出的地址是：
called by 0xc5d4eb
called by 0xc5d535
called by 0xc5d556
called by 0×80484ca

这里可以用打印出的地址减去加载的地址来计算偏移量。如，用 0xc5d4eb 减去加载地址 0×00c5d000，得到偏移量 0×4eb，然后把 0×4eb 传给 addr2line：

addr2line 0×4eb -f -s -e ./libbt_so.so

屏幕打印：
/home/work/mine/sysprog/think-in-compway/backtrace/bt_so.c:38

栈里的数据很有意思，在上一节中，通过分析栈里的数据，我们了解了变参函数的实现原理。在这一节中，通过分析栈里的数据，我们又学到了 backtrace 的实现原理。

系统程序员成长计划－像机器一样思考(三)

hello world 的密秘

hello world 是最经典的入门程序，该程序因 Brian Kernighan 和 Dennis Ritchie 编写的《C 语言程序设计》(The C Programming Language)而广泛流传。hello world 同样也是深入研究计算机的极好题材，可以说我对计算机的理解，很大程度上归功于对 hello world 的研究。后来看了《深入理解计算机》和台湾著名黑客黄敬群老师的《深入浅出 Hello World》之后，对 hello world 又有了更深的认识。这里和大家分享一下 hello world 背后的密秘。

C 语言的 hello world 如下：

```
#include <stdio.h>


int main(int argc, char* argv[])

{

    printf("Hello World!\n");


    return 0;

}
```

o 密秘一：main 函数的原型

有些初学者是这样写 HelloWorld 的，编译也可以通过，运行也正常：

```
void main(void)

{

    printf("Hello World!\n");


    return;

}
```

如果用 gcc 来编译，你会发现，把 main 写成什么样子都行，只要函数名为 main，编译都可以通过(可能有警告)。但下面两种写法才是比较正规的：

```
int main(int argc, char* argv[])

int main(int argc, char* argv[], char* env[])
```

argc 是命令行参数的个数。
argv 是命令行参数，以 NULL 结束。
env 是环境变量，以 NULL 结束。

下面的程序可以显示 argv 和 env 的内容：

```
#include <stdio.h>


int main(int argc, char* argv[], char* env[])

{

        int i = 0;

        printf("Hello World!\n");


        for(i = 0; argv[i] != NULL; i++)

        {

                printf("argv[%d]=%s\n", i, argv[i]);

        }


        for(i = 0; env[i] != NULL; i++)

        {
```

```
            printf("env[%d]=%s\n", i, env[i]);
        }

        return 0;
}
```

编译后运行它：
./helloworld arg1 arg2

屏幕打印:

```
Hello World!

argv[0]=./helloworld

argv[1]=arg1

argv[2]=arg2

env[0]=SSH_AGENT_PID=2609

env[1]=HOSTNAME=lixj.linux

env[2]=DESKTOP_STARTUP_ID=

env[3]=TERM=xterm

...
```

环境变量是从父进程继承过来的，通过 setenv 和 getenv 等函数可以存取环境变量，但对环境变量的修改只会影响当前进程及子进程，而不会影响父进程。

o 密秘二：main 函数的返回值

正常情况下，我们调用一个函数之后，通过检查它的返回值来判断函数执行的结果。但 main 函数不是由程序员自己调用的，那么它的返回值会返回给谁呢？

答案是，main 函数的返回值是返回给父进程的，父进程调用下列函数来获取子进程的退出码(即 main 函数的返回值)：

```
pid_t wait(int *status);

pid_t waitpid(pid_t pid, int *status, int options);
```

在 bash 里，执行一个命令后(bash 是父进程，命令是子进程)，$?里存放的是这个命令的退出码，我们来测试一下：

```
int main(int argc, char* argv[])

{

    printf("Hello World!\n");


    return 100;

}
```

编译运行：
./helloworld_2;echo $?
屏幕打印：
Hello World!
100

100 正是我们的返回值。这里对上面的程序做点修改，让它返回 1000，看看它真的会返回 1000 吗？

```
int main(int argc, char* argv[])

{

    printf("Hello World!\n");


    return 1000;

}
```

编译运行：
./helloworld_2;echo $?

屏幕打印：
Hello World!
232

奇怪的是，这里打印的不是 1000，而是 232！不少朋友都碰到过类似的问题，我自己也遇到过。后来查资料才知道，main 函数的返回值虽然是 int 的，可以保存 32 位的整数，但实际上，系统只使用了一个字节来保存返回值，所以这是打印的是 232(即 1000 & 0xff)。

o 密秘三：被隐藏的细节

现在我们用 strace 来分析一下 HelloWorld 的执行过程：

strace ./helloworld_3

屏幕打印：

```
execve("./helloworld_3", ["./helloworld_3"], [/* 51 vars */]) = 0

brk(0)                                  = 0x8eda000

mmap2(NULL, 4096, PROT_READ|PROT_WRITE, MAP_PRIVATE|MAP_ANONYMOUS, -1, 0) = 0xb8029000

access("/etc/ld.so.preload", R_OK)      = -1 ENOENT (No such file or directory)

open("./tls/i686/sse2/libc.so.6", O_RDONLY) = -1 ENOENT (No such file or directory)

open("./tls/i686/libc.so.6", O_RDONLY)  = -1 ENOENT (No such file or directory)

open("./tls/sse2/libc.so.6", O_RDONLY)  = -1 ENOENT (No such file or directory)

open("./tls/libc.so.6", O_RDONLY)       = -1 ENOENT (No such file or directory)

open("./i686/sse2/libc.so.6", O_RDONLY) = -1 ENOENT (No such file or directory)

open("./i686/libc.so.6", O_RDONLY)      = -1 ENOENT (No such file or directory)

open("./sse2/libc.so.6", O_RDONLY)      = -1 ENOENT (No such file or directory)

open("./libc.so.6", O_RDONLY)           = -1 ENOENT (No such file or directory)

open("/etc/ld.so.cache", O_RDONLY)      = 3

fstat64(3, {st_mode=S_IFREG|0644, st_size=123031, ...}) = 0

mmap2(NULL, 123031, PROT_READ, MAP_PRIVATE, 3, 0) = 0xb800a000

close(3)                                = 0

open("/lib/libc.so.6", O_RDONLY)        = 3

read(3, "\177ELF\1\1\1\0\0\0\0\0\0\0\0\0\3\0\3\0\1\0\0\0@\307o\0004\0\0\0"..., 512) = 512

fstat64(3, {st_mode=S_IFREG|0755, st_size=1758448, ...}) = 0

mmap2(0x6e6000, 1476176, PROT_READ|PROT_EXEC, MAP_PRIVATE|MAP_DENYWRITE, 3, 0) = 0x6e6000

mmap2(0x849000, 12288, PROT_READ|PROT_WRITE, MAP_PRIVATE|MAP_FIXED|MAP_DENYWRITE, 3, 0x163) =
0x849000

mmap2(0x84c000, 9808, PROT_READ|PROT_WRITE, MAP_PRIVATE|MAP_FIXED|MAP_ANONYMOUS, -1, 0) =
0x84c000

close(3)                                = 0

mmap2(NULL, 4096, PROT_READ|PROT_WRITE, MAP_PRIVATE|MAP_ANONYMOUS, -1, 0) = 0xb8009000
```

```
set_thread_area({entry_number:-1 -> 6, base_addr:0xb80096c0, limit:1048575, seg_32bit:1,
contents:0, read_exec_only:0, limit_in_pages:1, seg_not_present:0, useable:1}) = 0

mprotect(0x849000, 8192, PROT_READ)     = 0

mprotect(0x6e2000, 4096, PROT_READ)     = 0

munmap(0xb800a000, 123031)              = 0

fstat64(1, {st_mode=S_IFCHR|0620, st_rdev=makedev(136, 1), ...}) = 0

mmap2(NULL, 4096, PROT_READ|PROT_WRITE, MAP_PRIVATE|MAP_ANONYMOUS, -1, 0) = 0xb8028000

write(1, "Hello World!\n", 13Hello World!

)          = 13

exit_group(0)                           = ?
```

这么一行简单的程序，居然做数十次系统调用。可见简单的程序并不简单，只是实现细节被操作系统和函数库封装起来了。

前面只是加载并执行 helloworld 程序，眞正打印字符串的是 write 函数：它把字符串写入文件描述符为 1 的文件里。在 C 语言中：

文件描述符 0 表示标准输入。
文件描述符 1 表示标准输出。
文件描述符 2 表示标准错误输出。

o 密秘四：printf 不见了。

我们看下 main 函数的汇编代码(x86)：

```
int main(int argc, char* argv[], char* env[])

{

 80483b4:       8d 4c 24 04             lea    0x4(%esp),%ecx

 80483b8:       83 e4 f0                and    $0xfffffff0,%esp

 80483bb:       ff 71 fc                pushl  -0x4(%ecx)

 80483be:       55                      push   %ebp

 80483bf:       89 e5                   mov    %esp,%ebp

 80483c1:       51                      push   %ecx
```

```
 80483c2:       83 ec 04                sub    $0x4,%esp

        printf("Hello World!\n");

 80483c5:       c7 04 24 a4 84 04 08    movl   $0x80484a4,(%esp)

 80483cc:       e8 1f ff ff ff          call   80482f0 <puts@plt>


        return 0;

 80483d1:       b8 00 00 00 00          mov    $0x0,%eax

}
```

奇怪的是这里并没有调用 printf，而且是调用的 puts。第一次见到这个代码，我想 printf 可能只是个宏，最终由 puts 来实现打印功能。不过后来证实 glibc 里确实有 printf 函数，那为什么这里变成了 puts 呢？

我对代码做了修改，不包含任何头文件，自己声明 printf 的函数原型，这样确保没有宏在做怪：

```
int printf(const char *format, ...);


int main(int argc, char* argv[], char* env[])

{

    printf("Hello World!\n");


    return 0;

}
```

反汇编出来的代码没有任何变化，由此可见，不是宏在做怪，而是 gcc 做了手脚。原因可能是：printf 要对格式字符进行分析，相对来说效率低下， 如果只有一个参数，printf 的功能和 puts 一致，于是 gcc 就用 puts 代替了它。为了证实这个观点，对代码再做一点修改，使用格式字符串打印：

```
#include <stdio.h>


int main(int argc, char* argv[], char* env[])

{
```

```
    printf("%s%s", "Hello", " World!\n");

    return 0;
}
```

这次汇编代码变成了：

```
int main(int argc, char* argv[], char* env[])
{
 80483c4:       8d 4c 24 04             lea    0x4(%esp),%ecx
 80483c8:       83 e4 f0                and    $0xfffffff0,%esp
 80483cb:       ff 71 fc                pushl  -0x4(%ecx)
 80483ce:       55                      push   %ebp
 80483cf:       89 e5                   mov    %esp,%ebp
 80483d1:       51                      push   %ecx
 80483d2:       83 ec 14                sub    $0x14,%esp
        printf("%s%s", "Hello", " World!\n");
 80483d5:       c7 44 24 08 c4 84 04    movl   $0x80484c4,0x8(%esp)
 80483dc:       08
 80483dd:       c7 44 24 04 cd 84 04    movl   $0x80484cd,0x4(%esp)
 80483e4:       08
 80483e5:       c7 04 24 d3 84 04 08    movl   $0x80484d3,(%esp)
 80483ec:       e8 03 ff ff ff          call   80482f4 <printf@plt>

        return 0;
 80483f1:       b8 00 00 00 00          mov    $0x0,%eax
}
```

看来真的是 gcc 做了优化。

o 密秘五：链接了哪些共享库

用 ldd 查看 helloworld 链接的共享库。

屏幕打印：

```
        linux-gate.so.1 =>  (0x002da000)

        libc.so.6 => /lib/libc.so.6 (0x006e6000)

        /lib/ld-linux.so.2 (0x006c6000)
```

libc.so.6 是 glibc，它实现了像 printf 这类标准 C 的函数。

ld-linux.so.2 是 elf 可执行文件的解释器，Linux 内核在执行 ELF 可执行文件时，其实是执行 ld-linux.so.2，然后由 ld-linux.so.2 去加载可执行文件及依赖的共享库。/lib/ld-linux.so.2 是共享库，但它又是可执行的，运行它，屏幕会打印：

```
Usage: ld.so [OPTION]... EXECUTABLE-FILE [ARGS-FOR-PROGRAM...]

You have invoked `ld.so', the helper program for shared library executables.

This program usually lives in the file `/lib/ld.so', and special directives

in executable files using ELF shared libraries tell the system's program

loader to load the helper program from this file.  This helper program loads

the shared libraries needed by the program executable, prepares the program

to run, and runs it.  You may invoke this helper program directly from the

command line to load and run an ELF executable file; this is like executing

that file itself, but always uses this helper program from the file you

specified, instead of the helper program file specified in the executable

file you run.  This is mostly of use for maintainers to test new versions

of this helper program; chances are you did not intend to run this program.


  --list              list all dependencies and how they are resolved

  --verify            verify that given object really is a dynamically linked

                      object we can handle
```

```
  --library-path PATH   use given PATH instead of content of the environment
                        variable LD_LIBRARY_PATH
  --inhibit-rpath LIST  ignore RUNPATH and RPATH information in object names
                        in LIST
```

从 ld-linux.so.2 的参数来看，它可以直接执行 ELF 文件:

/lib/ld-linux.so.2 ./helloworld_5

屏幕打印：

Hello World!

这和直接执行./helloworld_5 的效果一样。

linux-gate.so.1 则是有点奇怪了，它没有指向任何文件，而是指向一个地址 0×002da000。这个文件又称为虚拟动态共享库(vdso)，linux- gate.so.1 文件是不存在的，Linux 内核根据 CPU 类型，动态决定使用哪个共享库。它的功能主要是加速系统调用(syscall)：

我们知道，用户空间代码(如应用程序)和内核代码是运行在不同级别的，用户空间代码的运行权限较小，内核代码的运行权限最高。由用户空间进入内核空间，需要跨越一个门(gate)，这里 linux-gate.so.1 的功能就是提供这样一个门(gate)。

系统调用(syscall)需要跨越用户空间与内核空间之间的门。在 x86 系列 CPU 上，Linux 传统的做法是使用 80 中断(int 0×80)实现系统调用，不过它的执行效率较低。有的 CPU 提供了高效的 sysenter 指令，但不是所有 CPU 都支持，Linux 通过 VDSO 来兼容这 两种系统调用。于是提供了两个共享库：

```
ls -l /lib/modules/$(uname -r)/vdso
```

```
-rwxr-xr-x 1 root root 1764 03-24 12:10 vdso32-int80.so
```

```
-rwxr-xr-x 1 root root 1784 03-24 12:10 vdso32-sysenter.so
```

我们看 sysenter 的实现方式：

```
objdump -S vdso32-sysenter.so
```

```
00000400 <__kernel_sigreturn>:
 400:   58                      pop    %eax
 401:   b8 77 00 00 00          mov    $0x77,%eax
 406:   cd 80                   int    $0x80
```

```
 408:   90                      nop
 409:   8d 76 00                lea    0x0(%esi),%esi

0000040c <__kernel_rt_sigreturn>:
 40c:   b8 ad 00 00 00          mov    $0xad,%eax
 411:   cd 80                   int    $0x80
 413:   90                      nop

00000414 <__kernel_vsyscall>:
 414:   51                      push   %ecx
 415:   52                      push   %edx
 416:   55                      push   %ebp
 417:   89 e5                   mov    %esp,%ebp
 419:   0f 34                   sysenter
 41b:   90                      nop
 41c:   90                      nop
 41d:   90                      nop
 41e:   90                      nop
 41f:   90                      nop
 420:   90                      nop
 421:   90                      nop
 422:   eb f3                   jmp    417 <__kernel_vsyscall+0x3>
 424:   5d                      pop    %ebp
 425:   5a                      pop    %edx
 426:   59                      pop    %ecx
 427:   c3                      ret
```

里面实现了 kernel_sigreturn、kernel_rt_sigreturn 和 kernel_vsyscall。

vdso32-int80.so 也实现了这几个函数，只是方法不一样：

```
00000400 <__kernel_sigreturn>:

 400:   58                      pop    %eax

 401:   b8 77 00 00 00          mov    $0x77,%eax

 406:   cd 80                   int    $0x80

 408:   90                      nop

 409:   8d 76 00                lea    0x0(%esi),%esi


0000040c <__kernel_rt_sigreturn>:

 40c:   b8 ad 00 00 00          mov    $0xad,%eax

 411:   cd 80                   int    $0x80

 413:   90                      nop


00000414 <__kernel_vsyscall>:

 414:   cd 80                   int    $0x80

 416:   c3                      ret
```

o 密秘六：调用共享库中的函数

puts 是 libc 提供的函数，从反汇编代码中可以看到：

```
        printf("Hello World!\n");

 80483c5:       c7 04 24 a4 84 04 08    movl   $0x80484a4,(%esp)

 80483cc:       e8 1f ff ff ff          call   80482f0 <puts@plt>
```

从前面的分析中，我们已经知道，gcc 的优化把 printf 换成了 puts。但是这里也并没有直接调用 puts，而是调用的 puts@plt，这是怎么回事呢？puts@plt 显然是编译器加的一个中间函数，我们看一下这个函数对应的汇编代码：

```
080482f0 <puts@plt>:

 80482f0:   ff 25 1c 96 04 08       jmp    *0x804961c
```

```
 80482f6:   68 10 00 00 00          push   $0x10

 80482fb:   e9 c0 ff ff ff          jmp    80482c0 <_init+0x30>
```

现在我们用调试器来分析下：

```
gdb helloworld


(gdb) b main

Breakpoint 1 at 0x80483c5: file helloworld.c, line 5.

(gdb) r

Starting program: /home/work/mine/sysprog/think-in-compway/helloworld/helloworld


Breakpoint 1, main () at helloworld.c:5

5               printf("Hello World!\n");

Missing separate debuginfos, use: debuginfo-install glibc.i686
```

puts@plt 先跳到*0×804961c，我们看看*0×804961c 里有什么？

```
(gdb) x 0x804961c

0x804961c <_GLOBAL_OFFSET_TABLE_+20>:   0x080482f6
```

*0×804961c 等于 0×080482f6，这正是 puts@plt 中的第二行汇编代码的地址。也就是说 puts@plt 整个函数会顺序执行，直到跳转到 0×80482c0.

再来看看 0×80482c0 处有什么，通过汇编可以看到：

```
 080482c0 <__gmon_start__@plt-0x10>:

 80482c0:   ff 35 0c 96 04 08       push1  0x804960c

 80482c6:   ff 25 10 96 04 08       jmp    *0x8049610
```

又跳到了*0×8049610，转的弯眞多，没关系，我们再看*0×8049610：

```
(gdb) x 0x8049610

0x8049610 <_GLOBAL_OFFSET_TABLE_+8>:    0x006da4d0
```

```
(gdb) x /wa 0x006da4d0

0x6da4d0 <_dl_runtime_resolve>: 0x8b525150
```

原来转来转去就是为了调用函数_dl_runtime_resolve， _dl_runtime_resolve 的功能就是找到要调用函数(puts)的地址。

为什么不直接调用_dl_runtime_resolve，而要转这么多圈子呢？

先执行完 puts：

```
(gdb) n
```

再回头来看看 puts@plt 的第一行代码：

```
80482f0:   ff 25 1c 96 04 08       jmp    *0x804961c
```

```
(gdb) x 0x804961c

0x804961c <_GLOBAL_OFFSET_TABLE_+20>:   0x745af0 <puts>
```

对比前面的代码:

```
(gdb) x 0x804961c

0x804961c <_GLOBAL_OFFSET_TABLE_+20>:   0x080482f6
```

也就是说第一次执行时，通过_dl_runtime_resolve 解析到函数地址，并保存 puts 的地址到 0×804962c 里，以后执行时就直接调用了。

o 密秘七：函数的解析过程

LD_DEBUG 是一个很有用的环境变量，通过它，我们可以对 helloworld 做更深入的分析，按下列方式运行 helloworld：

```
LD_DEBUG=symbols ./helloworld
```

屏幕打印：

```
...

      7264: symbol=malloc;  lookup in file=./helloworld [0]

      7264: symbol=malloc;  lookup in file=/lib/libc.so.6 [0]

      7264: symbol=calloc;  lookup in file=./helloworld [0]
```

```
      7264: symbol=calloc;  lookup in file=/lib/libc.so.6 [0]

      7264: symbol=realloc;  lookup in file=./helloworld [0]

      7264: symbol=realloc;  lookup in file=/lib/libc.so.6 [0]

      7264: symbol=free;  lookup in file=./helloworld [0]

      7264: symbol=free;  lookup in file=/lib/libc.so.6 [0]

      7264: symbol=puts;  lookup in file=./helloworld [0]

      7264: symbol=puts;  lookup in file=/lib/libc.so.6 [0]

...
```

查找函数时，先在可执行文件中查找，然后依次到共享库中查找。使用共享库，可以节省空间，但调用共享库的函数需要额外的开销。我们用静态链接的方式链接 helloworld：

gcc -g -static helloworld.c -o helloworld_static

这样运行速度可能会快些，但是可执行文件的大小增加了不少：

```
ls -l helloworld_static helloworld
```

屏幕打印：

```
-rwxrwxr-x 1 root root   6128 05-16 17:40 helloworld

-rwxrwxr-x 1 root root 562578 05-16 17:40 helloworld_static
```

共享库的好处其实取决于共享的次数，共享的次数越多，因共享而节省的空间越多。如果一个函数库只有一个程序使用它，把它编译成静态库将是更好的选择。

o 密秘八：托梁换柱

下面的 helloworld 会在屏幕上打印出什么内容？

```
#include <stdio.h>


int main(int argc, char* argv[], char* env[])

{

    printf("Hello World!\n");


    return 0;
```

```
}
```

肯定是“Hello World!”，不是吗？下面我们来个托梁换柱：

preload.c

```
#include <stdio.h>

#include <ctype.h>

int puts(const char *s)

{

    const char* p = s;

    while(*p != '\0')

    {

        putc(toupper(*p), stdout);

        p++;

    }

    return 0;

}
```

编译：

```
gcc -g -shared preload.c -o libpreload.so
```

按下列方式运行 helloworld:

```
LD_PRELOAD=./libpreload.so ./helloworld
```

屏幕打印：
HELLO WORLD!

设置环境变量 LD_PRELOAD 之后，打印的内容变成大写了！原来，LD_PRELOAD 指定的共享库被预先加载，如果出现重名的函数，预先加载的函数将会被调用。通过这种方法，我

们可以在不需要修改源代码(有时候可能没有源代码)的情况下，来改变一个程序的行为。

o 密秘九：内存模型

对 helloworld 做点修改：

```
int main(int argc, char* argv[], char* env[])

{

    printf("Hello World!\n");


    getchar();


    return 0;

}
```

这里调用了 getchar ，让程序不会直接退出。利用程序等待输入的时间，我们看下它的内存布局(假设 3458 是 helloworld 的进程 ID)：

```
cat /proc/3458/maps


0041f000-00420000 r-xp 0041f000 00:00 0          [vdso]

006c6000-006e2000 r-xp 00000000 08:01 765655     /lib/ld-2.8.so

006e2000-006e3000 r--p 0001c000 08:01 765655     /lib/ld-2.8.so

006e3000-006e4000 rw-p 0001d000 08:01 765655     /lib/ld-2.8.so

006e6000-00849000 r-xp 00000000 08:01 765657     /lib/libc-2.8.so

00849000-0084b000 r--p 00163000 08:01 765657     /lib/libc-2.8.so

0084b000-0084c000 rw-p 00165000 08:01 765657     /lib/libc-2.8.so

0084c000-0084f000 rw-p 0084c000 00:00 0

08048000-08049000 r-xp 00000000 08:05 2129380    /home/work/mine/sysprog/think-in-
compway/helloworld/helloworld_9

08049000-0804a000 rw-p 00000000 08:05 2129380    /home/work/mine/sysprog/think-in-
compway/helloworld/helloworld_9

b7f87000-b7f89000 rw-p b7f87000 00:00 0
```

```
b7fa6000-b7fa8000 rw-p b7fa6000 00:00 0

bfc93000-bfca8000 rw-p bffeb000 00:00 0          [stack]
```

从这里我们可以看出：

这里最低有效地址是 0×0041f000，其下为保留区域。

可执行文件和共享库映射区，每个文件通常占 1-3 个区域，分别存放全局变量，常量和代码，它们有不同的属性。

这没有动态分配内存，所以没有堆内存。

栈是从上向下增长的， 这里栈底为 0xbfca8000，我做了多次测试，发现栈底并不总是在这个位置。不过对于 32 位系统，我们可以确信的是栈底总是小于 0xc0000000 的。

o 密秘十：main 函数不是第一个执行的函数

教科书告诉我们 main 函数是 C 语言程序的入口函数，实际上 main 并不是第一个被执行的函数。

我们对程序做点修改：

```
#include <stdio.h>


__attribute ((constructor)) void hello_init(void)

{

    printf("%s\n", __func__);


    return;

}


__attribute ((destructor)) void hello_fini(void)

{

    printf("%s\n", __func__);


    return;

}


int main(int argc, char* argv[], char* env[])
```

```
{
    printf("Hello World!\n");

    return 0;
}
```

编译并运行：

./helloworld_10

屏幕打印:

```
hello_init
Hello World!
hello_fini
```

在 main 函数之前执行了 hello_init ，在 main 函数之后执行了 hello_fini。在 gdb 里执行./helloworld_10，并在 hello_init 和 hello_fini 设置断点，看是谁调用了它们：

```
(gdb) bt
#0  hello_init () at helloworld.10.c:5
#1  0x0804849d in __do_global_ctors_aux ()
#2  0x080482bc in _init ()
#3  0x08048439 in __libc_csu_init ()
#4  0x006fc571 in __libc_start_main () from /lib/libc.so.6
#5  0x08048321 in _start ()

(gdb) bt
#0  hello_fini () at helloworld.10.c:12
#1  0x0804836f in __do_global_dtors_aux ()
#2  0x080484c4 in _fini ()
```

```
#3  0x006d4f7b in _dl_fini () from /lib/ld-linux.so.2

#4  0x00713b39 in exit () from /lib/libc.so.6

#5  0x006fc5de in __libc_start_main () from /lib/libc.so.6

#6  0x08048321 in _start ()
```

其实_start 才是程序的入口，它先构造进程中的全局对象，执行一些初始化函数，然后调用 main 函数，最后析构全局对象，执行一些退出函数。

系统程序员成长计划-文本处理(一)

状态机(1)

o 有穷状态机的形式定义

有穷状态机是一个五元组 (Q，Σ，δ，q0，F)，其中：
Q 是一个有穷集合，称为状态集。
Σ 是一个有穷集合，称为字母表。
δ: Q xΣ Q 称为状态转移函数。
q0 是初始状态。
F 是接受状态集。

教科书上是这样定义有穷自动机的，这个形式定义精确的描述了有穷状态机的含义。但是大部分人(包括我自己)第一次看到它时，反复的读上几遍，仍然不知道它在说什么。幸好通过一些实例，我们可以很容易明白有穷状态机的原理。

自动门是一个典型的有穷状态机：

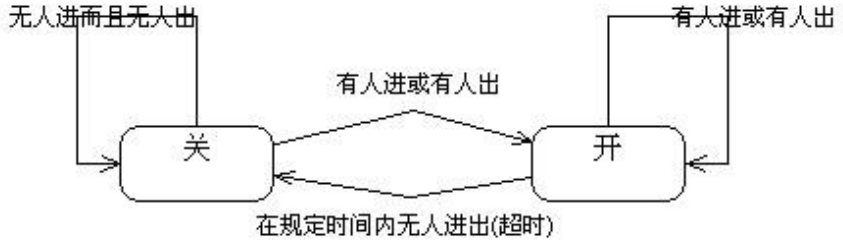
它有“开”和“关”两种状态，这就是它的状态集，也就是上面所说的 Q。

人可以从自动门进来或出去，当人进来或出去的时候，自动门会自动打开，如果在规定的时间内没有人进出，自动门会自动关上。人的进来、出去和超时三个 事件是自动门的字母表，也就是上面所说的 Σ。而自动门在当前状态下，对事件的响应，会引起状态的变化，这就是状态转换函数，也就是上面所说的 δ。

自动门刚安装好的时候，我们可以认为它是关上的，所以关闭状态是自动门的初始状态。

在理想情况下，自动门会一直运行，所以它没有接受状态，接受状态集 F 是空集。

有穷状态机的形式定义很精确，文字描述比较通俗，而图形表示则比较直观。通用建模语言（UML）里的状态图是状态机的常用图形表示方法。简单的状态 图包括一些状态，用圆角方框表示，里面有状态的名称。状态之间的转换，用箭头表示，上面可以加转换条件。自动门的状态机可以用下图表示：



有穷状态机很简单，在生活中可以找出很多这样的例子。但是教科书里讲得太复杂了，一会儿证明确定性有穷状态机和非确定性有穷状态机的等价性，一会儿 证明正则表达式的正则运算是封闭的，一会儿又来个泵引理。花了很长时间，我才明白这些原理，但两年之后，我又把它们忘得一干二净。

主要原因是工作中没有机会运用它们，这些理论的证明于编程没有太大用处，不过状态机本身却是文本处理利器，由于程序员在很多场合下都是在与文本数据 打交道，所以状态机是程序员必备的工具之一。这里我们将一起学习如何用状态机来处理文本数据，后面我们也会提到状态机的其它用途，不过不是本节的重点。

系统程序员成长计划-文本处理(一)

状态机(2)

o 用有穷状态机解一道面试题。

刚毕业的时候，我到一家外企面试，面试题里有这样一道题：

统计一篇英文文章里的单词个数。

有多种方法可以解这道题，这里我们选择用有穷状态机来解，做法如下：

先把这篇英文文章读入到一个缓冲区里，让一个指针从缓冲区的头部一直移到缓冲区的尾部，指针会处于两种状态：“单词内”或“单词外”，加上后面提到的初始状态和接受状态，就是有穷状态机的状态集。缓冲区中的字符集合就是有穷状态机的字母表。

如果当前状态为“单词内”，移到指针时，指针指向的字符是非单词字符(如标点和空格)，那状态会从“单词内”转换到“单词外”。如果当前状态为“单 词外”， 移到指针时，指针指向的字符是单词字符(如字母)，那状态会从“单词外”转换到“单词内”。这些转换规则就是状态转换函数。

指针指向缓冲区的头部时是初始状态。

指针指向缓冲区的尾部时是接受状态。

每次当状态从“单词内”转换到“单词外”时，单词计数增加一。
这个有穷状态机的图形表示如下：

下面我们看看程序怎么写：

```
int count_word(const char* text)
{
        /*定义各种状态，我们不关心接受状态，这里可以不用定义。*/
    enum _State
    {
        STAT_INIT,
        STAT_IN_WORD,
        STAT_OUT_WORD,
    }state = STAT_INIT;

    int count = 0;
    const char* p = text;

    /*在一个循环中，指针从缓冲区头移动缓冲区尾*/
    for(p = text; *p != '\0'; p++)
    {
        switch(state)
        {
            case STAT_INIT:
            {
                if(IS_WORD_CHAR(*p))
                {
                    /*指针指向单词字符，状态转换为单词内*/
                    state = STAT_IN_WORD;
                }
                else
```

```
    {
        /*指针指向非单词字符，状态转换为单词外*/
        state = STAT_OUT_WORD;
    }
    break;
}
case STAT_IN_WORD:
{
    if(!IS_WORD_CHAR(*p))
    {
        /*指针指向非单词字符，状态转换为单词外，增加单词计数*/
        count++;
        state = STAT_OUT_WORD;
    }
    break;
}
case STAT_OUT_WORD:
{
    if(IS_WORD_CHAR(*p))
    {
        /*指针指向单词字符，状态转换为单词内*/
        state = STAT_IN_WORD;
    }
    break;
}
```

```
            default:break;
        }
    }

    if(state == STAT_IN_WORD)
{
        /*如果由单词内进入接受状态，增加单词计数*/
        count++;
    }

    return count;
}
```

用状态机来解这道题目，思路清晰，程序简单，不易出错。

这道题目只是为了展示一些奇技淫巧，还是有一些实际用处呢？回答这个问题之前，我们先对上面的程序做点扩展，不只是统计单词的个数，而且要分离出里面的每个单词。

```
int word_segmentation(const char* text, OnWordFunc on_word, void* ctx)
{
    enum _State
    {
        STAT_INIT,
        STAT_IN_WORD,
        STAT_OUT_WORD,
    }state = STAT_INIT;

    int count = 0;
    char* copy_text = strdup(text);
    char* p = copy_text;
```

```
char* word = copy_text;


for(p = copy_text; *p != '\0'; p++)

{

    switch(state)

    {

        case STAT_INIT:

        {

            if(IS_WORD_CHAR(*p))

            {

                word = p;

                state = STAT_IN_WORD;

            }

            break;

        }

        case STAT_IN_WORD:

        {

            if(!IS_WORD_CHAR(*p))

            {

                count++;

                *p = '\0';

                on_word(ctx, word);

                state = STAT_OUT_WORD;

            }

            break;

        }
```

```
            case STAT_OUT_WORD:
            {
                if(IS_WORD_CHAR(*p))
                {
                    word = p;
                    state = STAT_IN_WORD;
                }
                break;
            }
            default:break;
        }
    }

    if(state == STAT_IN_WORD)
    {
        count++;
        on_word(ctx, word);
    }

    free(copy_text);

    return count;
}
```

状态机不变，只是在状态转换时，做是事情不一样。这里从“单词内”转换到其它状态时，增加单词计数，并分离出当前的单词。至于拿分离出的单词来做什么，由传入的回调函数决定，比如可以用来统计每个单词出现的频率。

但如果讨论还是限于英文文章，这个程序的意义仍然不大，现在来做进一步扩展。我们考虑

的文本不再是英文文章，而是一些文本数据，这些数据由一些分隔符分开，我们把数据称为token，现在我们要把这些token分离出来。

```
typedef void (*OnTokenFunc)(void* ctx, int index, const char* token);


#define IS_DELIM(c) (strchr(delims, c) != NULL)

int parse_token(const char* text, const char* delims, OnTokenFunc on_token, void* ctx)

{

    enum _State

    {

        STAT_INIT,

        STAT_IN,

        STAT_OUT,

    }state = STAT_INIT;


    int   count     = 0;

    char* copy_text = strdup(text);

    char* p         = copy_text;

    char* token     = copy_text;


    for(p = copy_text; *p != '\0'; p++)

    {

        switch(state)

        {

            case STAT_INIT:

            case STAT_OUT:

            {

                if(!IS_DELIM(*p))
```

```
            {
                token = p;
                state = STAT_IN;
            }
            break;
        }
        case STAT_IN:
        {
            if(IS_DELIM(*p))
            {
                *p = '\0';
                on_token(ctx, count++, token);
                state = STAT_OUT;
            }
            break;
        }
        default:break;
    }
}

if(state == STAT_IN)
{
    on_token(ctx, count++, token);
}

on_token(ctx, -1, NULL);
```

```
    free(copy_text);


    return count;

}
```

用分隔符分隔的文本数据有很多，如：

环境 PATH，它由‘:’分开的多个路径组成。如：
/usr/lib/qt-3.3/bin:/usr/kerberos/bin:/backup/tools/jdk1.5.0_18/bin/:/usr/lib/ccache:/usr/local/bin:/bin:/usr/bin:/home/lixianjing/bin

文件名，它由‘/’分开的路径组成。如：
/usr/lib/qt-3.3/bin

URL 中的参数，它‘&’分开的多个 key/value 对组成。
hl=zh-CN&q=limodev&btnG=Google+搜索&meta=&aq=f&oq=

所有这些数据都可以用上面的函数处理，所以这个小函数是颇具实用价值的。

系统程序员成长计划-文本处理(一)

状态机(3)

o INI 解析器

上面我们看了只有中间两个状态的状态机，现在我们来看一个稍微复杂一点的状态机。

INI 文件是 Windows 下常用的一种配置文件。它由多个分组组成，每个组有多个配置项，每个配置项又由名称和值组成。文件里还可以包含注释，注释通常以‘;’(或‘#’)开始，直到当前行结束。如 XP 下的 win.ini：

```
; for 16-bit app support

[fonts]

[extensions]

[mci extensions]

[files]

[MCI Extensions.BAK]

aif=MPEGVideo

aifc=MPEGVideo

aiff=MPEGVideo
```

```
asf=MPEGVideo

asx=MPEGVideo

au=MPEGVideo

mlv=MPEGVideo

m3u=MPEGVideo

mp2=MPEGVideo

mp2v=MPEGVideo

mp3=MPEGVideo

[annie]

CaptureFile=

VideoDevice=0

AudioDevice=0

FrameRate=333333

UseFrameRate=1

CaptureAudio=1

WantPreview=1

MasterStream=-1

[SciCalc]

layout=0
```

第一行是注释，后面有 fonts、extensions 和 mci extensions 三个空的分组，MCI Extensions.BAK、annie 和 SciCalc 三个分组包含有一个或多个配置项。

对于这样一个文件，我们应该怎样去解析它呢？按照前面的方法，先把数据读入到一个缓冲区中，让一个指针指向缓冲区的头部，然后移动指针，直到指向缓冲区的尾部。在这个过程中，指针可能指向的注释、分组的组名、配置项的名称、配置项的值或者一些如换行符之类的格式信息。

由此，我们可以这样来定义 INI 的状态机：

状态集合：

1. 分组的组名状态
2. 注释状态
3. 配置项的名称状态
4. 配置项的值状态
5. 空白状态

状态转换函数：
1. 初始状态为“空白”状态。
2. 在“空白”状态下，读入字符‘[’，进入“分组组名”状态。
3. 在“分组组名”状态下，读入字符‘]’,分组组名解析成功，回到“空白”状态。
4. 在“空白”状态下，读入字符‘;’，进入“注释”状态。
5. 在“注释”状态下，读入换行字符，结束“注释”状态，回到“空白”状态。
6. 在“空白”状态下，读入非空白字符，进入“配置项的名称”状态。
7. 在“配置项的名称”状态下，读入字符‘=’， 配置项的名称解析成功，进入“配置项的值”状态。
8. 在“配置项的值”状态下，读入换行字符，配置项的值解析成功，回到“空白”状态。

INI 状态机可以用下图来表示：

现在我们来看看程序实现：

```
static void ini_parse (char* buffer, char comment_char, char delim_char)

{

    char* p = buffer;

    char* group_start = NULL;

    char* key_start   = NULL;

    char* value_start = NULL;

    /*定义 INI 解析器的状态，初始状态为“空白”状态。*/

    enum _State

    {

        STAT_NONE = 0,

        STAT_GROUP,

        STAT_KEY,

        STAT_VALUE,
```

```
    STAT_COMMENT
}state = STAT_NONE;

for(p = buffer; *p != '\0'; p++)
{
    switch(state)
    {
        case STAT_NONE:
        {
            if(*p == '[')
            {
                /*在“空白”状态下，读入字符‘[’，进入“分组组名”状态。*/
                state = STAT_GROUP;
                group_start = p + 1;
            }
            else if(*p == comment_char)
            {
                /*在“空白”状态下，读入字符‘;’，进入“注释”状态。*/
                state = STAT_COMMENT;
            }
            else if(!isspace(*p))
            {
                /*在“空白”状态下，读入非空白字符，进入“配置项的名称”状态。*/
                state = STAT_KEY;
                key_start = p;
```

```
    }
    break;
}
case STAT_GROUP:
{
    /*在“分组组名”状态下，读入字符‘]’,分组组名解析成功，回到“空白”状态。*/
    if(*p == ']')
    {
        *p = '\0';
        state = STAT_NONE;
        strtrim(group_start);
        printf("[%s]\n", group_start);
    }
    break;
}
case STAT_COMMENT:
{
    /*在“注释”状态下，读入换行字符，结束“注释”状态，回到“空白”状态。*/
    if(*p == '\n')
    {
        state = STAT_NONE;
        break;
    }
    break;
}
```

```
            case STAT_KEY:
            {
                /*在“配置项的名称”状态下，读入字符‘=’， 配置项的名称解析成功，进入“配置项的值”状态。*/
                if(*p == delim_char || (delim_char == ' ' && *p == '\t'))
                {
                    *p = '\0';
                    state = STAT_VALUE;
                    value_start = p + 1;
                }
                break;
            }
            case STAT_VALUE:
            {
                /*在“配置项的值”状态下，读入换行字符，配置项的值解析成功，回到“空白”状态。*/
                if(*p == '\n' || *p == '\r')
                {
                    *p = '\0';
                    state = STAT_NONE;
                    strtrim(key_start);
                    strtrim(value_start);
                    printf("%s%c%s\n", key_start, delim_char, value_start);
                }
                break;
            }
            default:break;
```

```
        }
    }

    if(state == STAT_VALUE)
    {
        strtrim(key_start);
        strtrim(value_start);
        printf("%s%c%s\n", key_start, delim_char, value_start);
    }

    return;
}
```

ini 文件有几个变种：
1. 支持默认分组，如果只有一个分组，省略分组的组名，linux 下不少配置文件采用这种方式。
2. 注释符号，有的用‘;’，有的用‘#’，前者多用于 Windows 下，后面多用于 Linux 下。
3. 名称和值之间的分隔，有的用空格，有的用‘=’，有的‘:’。

不管哪种格式，它们的解析方法是一样的，在上面的程序中，我们使用了 comment_char 和 delim_char 两个参数，分别表示注释符号和分隔符号。

系统程序员成长计划-文本处理(一)

状态机(4)

XML 解析器

XML（Extensible Markup Language）即可扩展标记语言，也是一种常用的数据文件格式。相对于 INI 来说，它要复杂得多，INI 只能保存线性结构的数据，而 XML 可以保存树形结构的数据。先看下面的例子：

```
<?xml version="1.0" encoding="utf-8"?>
<mime-type xmlns="http://www.freedesktop.org/standards/shared-mime-info" type="all/all">
  <!--Created automatically by update-mime-database. DO NOT EDIT!-->
  <comment>all files and folders</comment>
```

</mime-type>

第一行称为处理指令(PI)，是给解析器用的。这里告诉解析器，当前的 XML 文件遵循 XML 1.0 规范，文件内容用 UTF-8 编码。

第二行是一个起始 TAG，TAG 的名称为 mime-type。它有两个属性，第一个属性的名称为 xmlns，值为 http://www.freedesktop.org/standards/shared-mime-info。第二个属性的名称为 type，值为 all/all。

第三行是一个注释。

第四行包括一个起始 TAG，一段文本和结束 TAG。

第五行是一个结束 TAG。

XML 本身的格式不是本文的重点，我们不详细讨论了。这里的重点是如何用状态机解析格式复杂的数据。

按照前面的方法，先把数据读入到一个缓冲区中，让一个指针指向缓冲区的头部，然后移动指针，直到指向缓冲区的尾部。在这个过程中，指针可能指向：起始 TAG，结束 TAG，注释，处理指令和文本。由此我们定义出状态机的主要状态：

1. 起始 TAG 状态
2. 结束 TAG 状态
3. 注释状态
4. 处理指令状态
5. 文本状态

由于起始 TAG、结束 TAG、注释和处理指令都在字符‘<’和‘>’之间，所以当读入字符‘<’时，我们还无法知道当前的状态， 为了便于处理，我们引入一个中间状态，称为“小于号之后”的状态。在读入字符‘<’和‘!’之后，还要读入两个‘-’，才能确定进入注释状态，为了 便于处理，再引入两个中间状态“注释前一”和“注释前二”。再引入一个“空”状态，表示不在上述任何状态中。

状态转换函数：
1. 在“空”状态下，读入字符‘<’，进入“小于号之后”状态。
2. 在“空”状态下，读入非‘<’非空白的字符，进入“文本”状态。
3. 在“小于号之后”状态下，读入字符‘！’，进入“注释前一” 状态。
4. 在“小于号之后”状态下，读入字符‘?’，进入“处理指令”状态。
5. 在“小于号之后”状态下，读入字符‘/’，进入“结束 TAG”状态。
6. 在“小于号之后”状态下，读入有效的 ID 字符，进入“起始 TAG”状态。
7. 在“注释前一” 状态下，读入字符‘-’， 进入“注释前二” 状态。
8. 在“注释前二” 状态下，读入字符‘-’， 进入“注释” 状态。
9. 在 “起始 TAG”状态、“结束 TAG”状态 、“文本” 状态、“注释”状态 和“处理指令”状态结束后，重新回到“空”状态下。

这个状态机的图形表示如下：

下面我们来看看代码实现：

```
void xml_parser_parse(XmlParser* thiz, const char* xml)

{

	/*定义状态的枚举值*/

	enum _State

	{

		STAT_NONE,

		STAT_AFTER_LT,

		STAT_START_TAG,

		STAT_END_TAG,

		STAT_TEXT,

		STAT_PRE_COMMENT1,

		STAT_PRE_COMMENT2,

		STAT_COMMENT,

		STAT_PROCESS_INSTRUCTION,

	}state = STAT_NONE;


	thiz->read_ptr = xml;

	/*指针从头移动到尾*/

	for(; *thiz->read_ptr != '\0'; thiz->read_ptr++)

	{

		char c = thiz->read_ptr[0];


		switch(state)

		{
```

```
    case STAT_NONE:
    {
            if(c == '<')
            {
/*在“空”状态下，读入字符‘<’，进入“小于号之后”状态。*/
                    xml_parser_reset_buffer(thiz);
                    state = STAT_AFTER_LT;
            }
            else if(!isspace(c))
            {
/*在“空”状态下，读入非‘<’非空白的字符，进入“文本”状态。*/
                    state = STAT_TEXT;
            }
            break;
    }
    case STAT_AFTER_LT:
    {
            if(c == '?')
            {
/*在“小于号之后”状态下，读入字符‘?’，进入“处理指令”状态。*/
                    state = STAT_PROCESS_INSTRUCTION;
            }
            else if(c == '/')
            {
/*在“小于号之后”状态下，读入字符‘/’，进入“结束TAG”状态。*/
```

```
                        state = STAT_END_TAG;
                }
                else if(c == '!')
                {
    /*在“小于号之后”状态下，读入字符‘！’，进入“注释前一” 状态*/
                        state = STAT_PRE_COMMENT1;
                }
                else if(isalpha(c) || c == '_')
                {
    /*在“小于号之后”状态下，读入有效的 ID 字符，进入“起始 TAG”状态。*/
                        state = STAT_START_TAG;
                }
                else
                {
                }
                break;
        }
        case STAT_START_TAG:
        {
/*进入子状态*/
                xml_parser_parse_start_tag(thiz);
                state = STAT_NONE;
                break;
        }
        case STAT_END_TAG:
```

```
        {
/*进入子状态*/
                xml_parser_parse_end_tag(thiz);
                state = STAT_NONE;
                break;
        }
        case STAT_PROCESS_INSTRUCTION:
        {
/*进入子状态*/
                xml_parser_parse_pi(thiz);
                state = STAT_NONE;
                break;
        }
        case STAT_TEXT:
        {
/*进入子状态*/
                xml_parser_parse_text(thiz);
                state = STAT_NONE;
                break;
        }
        case STAT_PRE_COMMENT1:
        {
                if(c == '-')
                {
    /*在“注释前一” 状态下，读入字符‘-’， 进入“注释前二” 状态。*/
```

```
                        state = STAT_PRE_COMMENT2;
                }
                else
                {
                }
                break;
        }
        case STAT_PRE_COMMENT2:
        {
                if(c == '-')
                {
    /*在“注释前二” 状态下，读入字符‘-’， 进入“注释” 状态。*/
                        state = STAT_COMMENT;
                }
                else
                {
                }
        }
        case STAT_COMMENT:
        {
/*进入子状态*/
                xml_parser_parse_comment(thiz);
                state = STAT_NONE;
                break;
        }
```

```
                default:break;
            }

            if(*thiz->read_ptr == '\0')
            {
                break;
            }
        }

        return;
}
```

解析并没有在此结束，原因是像“起始 TAG”状态和“处理指令”状态等，它们不是原子的，内部还包含一些子状态，如 TAG 名称，属性名和属性值等， 它们需要进一步分解。在考虑子状态时，我们可以忘掉它所处的上下文，只考虑子状态本身，这样问题会得到简化。下面看一下起始 TAG 的状态机。

假设我们要解析下面这样一个起始 TAG：
<mime-type xmlns=”http://www.freedesktop.org/standards/shared-mime-info” type=”all/all”>

我们应该怎样去做呢？还是按前面的方法，让一个指针指向缓冲区的头部，然后移动指针，直到指向缓冲区的尾部。在这个过程中，指针可能指向，TAG 名称，属性名和属性值。由此我们可以定义出状态机的主要状态：

1. “TAG 名称”状态
2. “属性名”状态
3. “属性值”状态

为了方便处理，再引两个中间状态，“属性名之前”状态和“属性值之前”状态。

状态转换函数：

初始状态为“TAG 名称”状态
1. 在“TAG 名称”状态下，读入空白字符，进入“属性名之前”状态。
2. 在“TAG 名称”状态下，读入字符‘/’或‘>’，进入“结束”状态。
3. 在“属性名之前”状态下，读入其它非空白字符，进入“属性名”状态。
4. 在“属性名”状态下，读入字符‘=’，进入“属性值之前”状态。
5. 在“属性值之前”状态下，读入字符‘“’，进入“属性值”状态。
6. 在“属性值”状态下，读入字符‘”’，成功解析属性名和属性值，回到“属性名之前”

状态。
7.在“属性名之前”状态下，读入字符‘/’或‘>’，进入“结束”状态。

由于处理指令(PI)里也包含了属性状态，为了重用属性解析的功能，我们把属性的状态再提取为一个子状态。这样，“起始 TAG”状态的图形表示如下：

下面我们看代码实现：

```
static void xml_parser_parse_attrs(XmlParser* thiz, char end_char)

{

        int i = 0;

        enum _State

        {

                STAT_PRE_KEY,

                STAT_KEY,

                STAT_PRE_VALUE,

                STAT_VALUE,

                STAT_END,

        }state = STAT_PRE_KEY;


        char value_end = '\"';

        const char* start = thiz->read_ptr;


        thiz->attrs_nr = 0;

        for(; *thiz->read_ptr != '\0' && thiz->attrs_nr < MAX_ATTR_NR; thiz->read_ptr++)

        {

                char c = *thiz->read_ptr;


                switch(state)

                {
```

```
                case STAT_PRE_KEY:
                {
                        if(c == end_char || c == '>')
                        {
                /*在“属性名之前”状态下，读入字符‘/’或‘>’，进入“结束”状态。*/
                                state = STAT_END;
                        }
                        else if(!isspace(c))
                        {
                /*在“属性名之前”状态下，读入其它非空白字符，进入“属性名”状态。*/
                                state = STAT_KEY;
                                start = thiz->read_ptr;
                        }
                }
                case STAT_KEY:
                {
                        if(c == '=')
                        {
                /*在“属性名”状态下，读入字符‘=’，进入“属性值之前”状态。*/
                                thiz->attrs[thiz->attrs_nr++] =
(char*)xml_parser_strdup(thiz, start, thiz->read_ptr - start);
                                state = STAT_PRE_VALUE;
                        }

                        break;
                }
```

```
                case STAT_PRE_VALUE:
                {
            /*在“属性值之前”状态下，读入字符‘“’，进入“属性值”状态。*/
                        if(c == '\"' || c == '\'')
                        {
                                state = STAT_VALUE;
                                value_end = c;
                                start = thiz->read_ptr + 1;
                        }
                        break;
                }
                case STAT_VALUE:
                {
            /*在“属性值”状态下，读入字符‘”’，成功解析属性名和属性值，回到“属性名之前”状态。*/
                        if(c == value_end)
                        {
                                thiz->attrs[thiz->attrs_nr++] =
(char*)xml_parser_strdup(thiz, start, thiz->read_ptr - start);
                                state = STAT_PRE_KEY;
                        }
                }
                default:break;
            }

            if(state == STAT_END)
            {
```

```
                break;
            }
        }

        for(i = 0; i < thiz->attrs_nr; i++)
        {
            thiz->attrs[i] = thiz->buffer + (size_t)(thiz->attrs[i]);
        }
        thiz->attrs[thiz->attrs_nr] = NULL;

        return;
}
```

记得在 XML 里，单引号和双引号都可以用来界定属性值，所以上面对此做了特殊处理。

```
static void xml_parser_parse_start_tag(XmlParser* thiz)
{
        enum _State
        {
            STAT_NAME,
            STAT_ATTR,
            STAT_END,
        }state = STAT_NAME;

        char* tag_name = NULL;
        const char* start = thiz->read_ptr - 1;

        for(; *thiz->read_ptr != '\0'; thiz->read_ptr++)
        {
```

```
char c = *thiz->read_ptr;

switch(state)
{
        case STAT_NAME:
        {
/*在“TAG名称”状态下，读入空白字符，属性子状态。*/
/*在“TAG名称”状态下，读入字符‘/’或‘>’，进入“结束”状态。*/
                if(isspace(c) || c == '>' || c == '/')
                {
                        state = (c != '>' && c != '/') ? STAT_ATTR : STAT_END;
                }
                break;
        }
        case STAT_ATTR:
        {
/*进入“属性”子状态*/
                xml_parser_parse_attrs(thiz, '/');
                state = STAT_END;

                break;
        }
        default:break;
}

if(state == STAT_END)
```

```
                {
                        break;
                }
        }

        for(; *thiz->read_ptr != '>' && *thiz->read_ptr != '\0'; thiz->read_ptr++);

        return;
}
```

处理指令的解析和起始TAG的解析基本上是一样的，这里只是看一下代码：

```
static void xml_parser_parse_pi(XmlParser* thiz)
{
        enum _State
        {
                STAT_NAME,
                STAT_ATTR,
                STAT_END
        }state = STAT_NAME;

        char* tag_name = NULL;
        const char* start = thiz->read_ptr;

        for(; *thiz->read_ptr != '\0'; thiz->read_ptr++)
        {
                char c = *thiz->read_ptr;

                switch(state)
                {
```

```
                case STAT_NAME:
                {
        /*在“TAG名称”状态下，读入空白字符，属性子状态。*/
        /*在“TAG名称”状态下，‘>’，进入“结束”状态。*/
                        if(isspace(c) || c == '>')
                        {
                                state = c != '>' ? STAT_ATTR : STAT_END;
                        }

                        break;
                }
                case STAT_ATTR:
                {
        /*进入“属性”子状态*/
                        xml_parser_parse_attrs(thiz, '?');
                        state = STAT_END;
                        break;
                }
                default:break;
        }

        if(state == STAT_END)
        {
                break;
        }
}
```

```
        tag_name = thiz->buffer + (size_t)tag_name;

        for(; *thiz->read_ptr != '>' && *thiz->read_ptr != '\0'; thiz->read_ptr++);

        return;
}
```

注释，结束TAG和文本的解析非常简单，这里结合代码看看就行了：

“注释”子状态的处理：

```
static void xml_parser_parse_comment(XmlParser* thiz)
{
    enum _State
    {
        STAT_COMMENT,
        STAT_MINUS1,
        STAT_MINUS2,
    }state = STAT_COMMENT;

    const char* start = ++thiz->read_ptr;
    for(; *thiz->read_ptr != '\0'; thiz->read_ptr++)
    {
        char c = *thiz->read_ptr;

        switch(state)
        {
            case STAT_COMMENT:
            {
```

```
    /*在“注释”状态下，读入‘-’，进入“减号一”状态。*/
    if(c == '-')
    {
        state = STAT_MINUS1;
    }
    break;
}
case STAT_MINUS1:
{
    if(c == '-')
    {
         /*在“减号一”状态下，读入‘-’，进入“减号二”状态。*/
        state = STAT_MINUS2;
    }
    else
    {
        state = STAT_COMMENT;
    }
    break;
}
case STAT_MINUS2:
{
    if(c == '>')
    {
        /*在“减号二”状态下，读入‘>’，结束解析。*/
```

```
                return;
            }
            else
            {
                state = STAT_COMMENT;
            }
        }
        default:break;
    }
}

    return;
}
```

“结束TAG”子状态的处理：

```
static void xml_parser_parse_end_tag(XmlParser* thiz)
{
    char* tag_name = NULL;
    const char* start = thiz->read_ptr;
    for(; *thiz->read_ptr != '\0'; thiz->read_ptr++)
{
    /*读入‘>’，结束解析。*/
        if(*thiz->read_ptr == '>')
        {
            break;
        }
    }
```

```
    return;
}
```

“文本”子状态的处理：

```
static void xml_parser_parse_text(XmlParser* thiz)
{
    const char* start = thiz->read_ptr - 1;
    for(; *thiz->read_ptr != '\0'; thiz->read_ptr++)
    {
        char c = *thiz->read_ptr;
        /*读入‘>’，结束解析。*/
        if(c == '<')
        {
            if(thiz->read_ptr > start)
            {
            }
            thiz->read_ptr--;
            return;
        }
        else if(c == '&')
        {
           /*读入‘&’，进入实体(entity)解析子状态。*/
            xml_parser_parse_entity(thiz);
        }
    }
```

```
    return;
}
```

实体(entity)子状态比较简单，这里不做进一步分析了，留给读者做练习吧。

Builder 模式

前面我们学习了状态机，并利用它来解析各种格式的文本数据。解析过程把线性的文本数据转换成一些基本的逻辑单元，但这通常只是任务的一部分，接下来 我们还要对这些解析出来的数据进一步处理。对于特定格式的文本数据，它的解析过程是一样的，但是对解析出来的数据的处理却是多种多样的。为了让解析过程能 被重用，就需要把数据的解析和数据的处理分开。

现在我们回过头来看一下前面写的函数 parse_token，这个函数把用分隔符分隔的文本数据，分离出一个一个的 token。

parse_token 的函数原型如下：

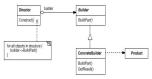
```
typedef void (*OnTokenFunc)(void* ctx, int index, const char* token);
int parse_token(const char* text, const char* delims, OnTokenFunc on_token, void* ctx)
```

parse_token 负责解析数据，但它并不关心数据代表的意义及用途。对数据的进一步处理由调用者提供的回调函数来完成，函数 parse_token 每解析到一个 token，就调用这个回调函数。parse_token 负责数据的解析，回调函数负责数据的处理，这样一来，数据的 解析和数据的处理就分开了。

parse_token 可以认为是 Builder 模式最朴素的应用。现在我们看看 Builder 模式：

Builder 模式的意图：将一个复杂对象的构建与它的表示分离，使得同样的构建过程可以创建不同的表示。“构建”其实就是前面的解析过程，而“表示”就是前面说的对数据的处理。

对象关系：



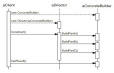
上面的 parse_token 与这里的 Director 对应。

上面的回调函数与这里的 Builder 对应。

具体的回调函数与这里的 ConcreteBuilder 对应。

对数据处理的结果就是 Product。

对象协作：



Client 是 parse_token 的调用者。

由于 parse_token 是按面向过程的方式设计的，所以 ConcreteBuilder 和 Director 的创建只是对应于一些初始化代码。

调用 parse_token 相当于调用 aDirector 的 Construct 函数。

调用回调函数相当于调用 aConcreteBuilder 的 BuildPart 函数。

回调函数可能把处理结果存在它的参数 ctx 中，GetResult 是从里面获取结果，这是可选的过程，依赖于具体回调函数所做的工作。

parse_token 的例子简单直接，对于理解 Builder 模式有较大的帮助，不过毕竟它是面向过程的。现在我们以前面的 XML 解析器为例来说 明 Builder 模式，虽然我们的代码是用 C 写的，但完全是用面向对象的思想来设计的。Builder 是一个接口，我们先把它定义出来：

```
struct _XmlBuilder;


typedef struct _XmlBuilder XmlBuilder;



typedef void (*XmlBuilderOnStartElementFunc)(XmlBuilder* thiz, const char* tag, const char**
attrs);


typedef void (*XmlBuilderOnEndElementFunc)(XmlBuilder* thiz, const char* tag);


typedef void (*XmlBuilderOnTextFunc)(XmlBuilder* thiz, const char* text, size_t length);


typedef void (*XmlBuilderOnCommentFunc)(XmlBuilder* thiz, const char* text, size_t length);


typedef void (*XmlBuilderOnPiElementFunc)(XmlBuilder* thiz, const char* tag, const char**
attrs);


typedef void (*XmlBuilderOnErrorFunc)(XmlBuilder* thiz, int line, int row, const char* message);


typedef void (*XmlBuilderDestroyFunc)(XmlBuilder* thiz);



struct _XmlBuilder

{

    XmlBuilderOnStartElementFunc on_start_element;
```

```
    XmlBuilderOnEndElementFunc    on_end_element;

    XmlBuilderOnTextFunc          on_text;

    XmlBuilderOnCommentFunc       on_comment;

    XmlBuilderOnPiElementFunc     on_pi_element;

    XmlBuilderOnErrorFunc         on_error;

    XmlBuilderDestroyFunc         destroy;



    char priv[1];

};



static inline void xml_builder_on_start_element(XmlBuilder* thiz, const char* tag, const char**
attrs)



{



    return_if_fail(thiz != NULL && thiz->on_start_element != NULL);



    thiz->on_start_element(thiz, tag, attrs);
```

```
    return;

}

static inline void xml_builder_on_end_element(XmlBuilder* thiz, const char* tag)

{

    return_if_fail(thiz != NULL && thiz->on_end_element != NULL);

    thiz->on_end_element(thiz, tag);

    return;

}

...
```

(其它 inline 函数不列在这里了)

XmlBuilder 接口要求实现下列函数：

on_start_element：解析器解析到一个起始 TAG 时调用它。
on_end_element：解析器解析到一个结束 TAG 时调用它。
on_text：解析器解析到一段文本时调用它。
on_comment：解析器解析到一个注释时调用它。
on_pi_element：解析器解析到一个处理指令时调用它。
on_error：解析器遇到错误时调用它。
destroy：用销毁 Builder 对象。

on_start_element 和 on_end_element 等函数相当于 Builder 模式中的 BuildPartX 函数。

XML 解析器相当于 Director，在前面我们已经写好了，不过它对解析出来的数据没有做任何处理。现在我们对它做些修改，让它调用 XmlBuilder 的函数。

XML 解析器对外提供下面几个函数：

o 构造函数。

```
XmlParser* xml_parser_create(void);
```

o 为 xmlParser 设置 builder 对象。

```
void       xml_parser_set_builder(XmlParser* thiz, XmlBuilder* builder);
```

o 解析 XML

```
void       xml_parser_parse(XmlParser* thiz, const char* xml);
```

o 析构函数

```
void       xml_parser_destroy(XmlParser* thiz);
```

在解析时，解析到相应的 tag，就调用 XmlBuilder 相应的函数：

o 解析到起始 tag 时调用 xml_builder_on_start_element

```
static void xml_parser_parse_start_tag(XmlParser* thiz)

{

        enum _State
```

```
{

        STAT_NAME,

        STAT_ATTR,

        STAT_END,

}state = STAT_NAME;



char* tag_name = NULL;

const char* start = thiz->read_ptr - 1;



for(; *thiz->read_ptr != '\0'; thiz->read_ptr++)

{

        char c = *thiz->read_ptr;



        switch(state)

        {

                case STAT_NAME:

                {

                        if(isspace(c) || c == '>' || c == '/')
```

```
                                {

                                        tag_name = (char*)xml_parser_strdup(thiz, start, thiz-
>read_ptr - start);

                                        state = (c != '>' && c != '/') ? STAT_ATTR : STAT_END;

                                }

                                break;

                        }

                        case STAT_ATTR:

                        {

                                xml_parser_parse_attrs(thiz, '/');

                                state = STAT_END;



                                break;

                        }

                        default:break;

                }



                if(state == STAT_END)
```

```
        {

                break;

        }

  }


  tag_name = thiz->buffer + (size_t)tag_name;

/*解析完成，调用 builder 的函数 xml_builder_on_start_element。*/

  xml_builder_on_start_element(thiz->builder, tag_name, (const char**)thiz->attrs);


  if(thiz->read_ptr[0] == '/')

  {

        /*如果 tag 以'/'结束，调用 builder 的函数 xml_builder_on_end_element。*/

        xml_builder_on_end_element(thiz->builder, tag_name);

  }


  for(; *thiz->read_ptr != '>' && *thiz->read_ptr != '\0'; thiz->read_ptr++);


  return;
```

```
}
```

o 解析到结束 tag 时调用 xml_builder_on_end_element

```
static void xml_parser_parse_end_tag(XmlParser* thiz)

{

        char* tag_name = NULL;

        const char* start = thiz->read_ptr;

        for(; *thiz->read_ptr != '\0'; thiz->read_ptr++)

        {

                if(*thiz->read_ptr == '>')

                {

                        tag_name = thiz->buffer + xml_parser_strdup(thiz, start, thiz->read_ptr-
start);

                        /*解析完成，调用 builder 的函数 xml_builder_on_end_element。*/

                        xml_builder_on_end_element(thiz->builder, tag_name);



                        break;

                }

        }
```

```
        return;

}
```

o 解析到文本时调用 xml_builder_on_text

```
static void xml_parser_parse_text(XmlParser* thiz)

{

        const char* start = thiz->read_ptr - 1;

        for(; *thiz->read_ptr != '\0'; thiz->read_ptr++)

        {

                char c = *thiz->read_ptr;



                if(c == '<')

                {

                        if(thiz->read_ptr > start)

                        {

                                /*解析完成，调用 builder 的函数 xml_builder_on_text。*/

                                xml_builder_on_text(thiz->builder, start, thiz->read_ptr-start);

                        }
```

```
                thiz->read_ptr--;

                return;

        }

        else if(c == '&')

        {

                xml_parser_parse_entity(thiz);

        }

    }



    return;

}
```

o 解析到注释时调用 xml_builder_on_comment

```
static void xml_parser_parse_comment(XmlParser* thiz)

{

    enum _State

    {

        STAT_COMMENT,
```

```
        STAT_MINUS1,

        STAT_MINUS2,

}state = STAT_COMMENT;



const char* start = ++thiz->read_ptr;

for(; *thiz->read_ptr != '\0'; thiz->read_ptr++)

{

        char c = *thiz->read_ptr;



        switch(state)

        {

                case STAT_COMMENT:

                {

                        if(c == '-')

                        {

                                state = STAT_MINUS1;

                        }
```

```
        break;

    }

    case STAT_MINUS1:

    {

        if(c == '-')

        {

            state = STAT_MINUS2;

        }

        else

        {

            state = STAT_COMMENT;

        }

        break;

    }

    case STAT_MINUS2:

    {

        if(c == '>')
```

```
                                {

                                        /*解析完成，调用 builder 的函数 xml_builder_on_comment。*/

                                        xml_builder_on_comment(thiz->builder, start, thiz-
>read_ptr-start-2);

                                        return;

                                }

                        }

                        default:break;

                }

        }


        return;

}
```

o 解析到处理指令时调用 xml_builder_on_pi_element

```
static void xml_parser_parse_pi(XmlParser* thiz)

{

        enum _State

        {
```

```
        STAT_NAME,

        STAT_ATTR,

        STAT_END

}state = STAT_NAME;



char* tag_name = NULL;

const char* start = thiz->read_ptr;



for(; *thiz->read_ptr != '\0'; thiz->read_ptr++)

{

        char c = *thiz->read_ptr;



        switch(state)

        {

                case STAT_NAME:

                {

                        if(isspace(c) || c == '>')

                        {
```

```
                                        tag_name = (char*)xml_parser_strdup(thiz, start, thiz->read_ptr - start);

                                        state = c != '>' ? STAT_ATTR : STAT_END;

                                }



                                break;

                        }

                        case STAT_ATTR:

                        {

                                xml_parser_parse_attrs(thiz, '?');

                                state = STAT_END;

                                break;

                        }

                        default:break;

                }



                if(state == STAT_END)

                {
```

```
                break;

            }

        }

        tag_name = thiz->buffer + (size_t)tag_name;

        /*解析完成，调用 builder 的函数 xml_builder_on_pi_element。*/

        xml_builder_on_pi_element(thiz->builder, tag_name, (const char**)thiz->attrs);

        for(; *thiz->read_ptr != '>' && *thiz->read_ptr != '\0'; thiz->read_ptr++);

        return;

}
```

从上面的代码可以看出，XmlParser 在适当的时候调用了 XmlBuilder 的接口函数，至于 XmlBuilder 在这些函数里做什么，要看具体的 Builder 实现了。

先看一个最简单的 XmlBuilder 实现，它只是在屏幕上打印出传递给它的数据：

o 创建函数

```
XmlBuilder* xml_builder_dump_create(FILE* fp)

{

        XmlBuilder* thiz = (XmlBuilder*)calloc(1, sizeof(XmlBuilder));
```

```
    if(thiz != NULL)

    {

        PrivInfo* priv = (PrivInfo*)thiz->priv;



        thiz->on_start_element   = xml_builder_dump_on_start_element;

        thiz->on_end_element    = xml_builder_dump_on_end_element;

        thiz->on_text                  = xml_builder_dump_on_text;

        thiz->on_comment         = xml_builder_dump_on_comment;

        thiz->on_pi_element      = xml_builder_dump_on_pi_element;

        thiz->on_error                = xml_builder_dump_on_error;

        thiz->destroy                  = xml_builder_dump_destroy;



        priv->fp = fp != NULL ? fp : stdout;

    }



    return thiz;

}
```

和其它接口的创建函数一样，它只是把接口要求的函数指针指到具体的实现函数上。

o 实现 on_start_element

```
static void xml_builder_dump_on_start_element(XmlBuilder* thiz, const char* tag, const char** attrs)

{

	int i = 0;

	PrivInfo* priv = (PrivInfo*)thiz->priv;

	fprintf(priv->fp, "<%s", tag);



	for(i = 0; attrs != NULL && attrs[i] != NULL && attrs[i + 1] != NULL; i += 2)

	{

		fprintf(priv->fp, " %s=\"%s\"", attrs[i], attrs[i + 1]);

	}

	fprintf(priv->fp, ">");



	return;

}
```

o 实现 on_end_element

```
static void xml_builder_dump_on_end_element(XmlBuilder* thiz, const char* tag)
```

```
{

        PrivInfo* priv = (PrivInfo*)thiz->priv;

        fprintf(priv->fp, "\n", tag);



        return;

}
```

o 实现 on_text

```
static void xml_builder_dump_on_text(XmlBuilder* thiz, const char* text, size_t length)

{

        PrivInfo* priv = (PrivInfo*)thiz->priv;

        fwrite(text, length, 1, priv->fp);



        return;

}
```

o 实现 on_comment

```
static void xml_builder_dump_on_comment(XmlBuilder* thiz, const char* text, size_t length)

{
```

```
        PrivInfo* priv = (PrivInfo*)thiz->priv;

        fprintf(priv->fp, "\n");



        return;

}
```

o 实现 on_pi_element

```
static void xml_builder_dump_on_pi_element(XmlBuilder* thiz, const char* tag, const char**
attrs)

{

        int i = 0;

        PrivInfo* priv = (PrivInfo*)thiz->priv;

        fprintf(priv->fp, "fp, " %s=\"%s\"", attrs[i], attrs[i + 1]);

        }

        fprintf(priv->fp, "?>\n");



        return;

}
```

o 实现 on_error

```
static void xml_builder_dump_on_error(XmlBuilder* thiz, int line, int row, const char* message)
```

```
{

        fprintf(stderr, "(%d,%d) %s\n", line, row, message);


        return;

}
```

上面的 XmlBuilder 实现简单，而且有一定的实用价值，我一般都会先写这样一个 Builder。它不但对于调试程序有不小的帮助，而且只要稍 做修改，就可以把它改进成一个美化数据格式的小工具，不管原始数据的格式(当然要合符相应的语法规则)有多乱，你都能以一种比较好看的方式打印出来。

下面我们再看一个比较复杂的 XmlBuilder 的实现，它根据接收的数据构建一棵 XML 树。

o 创建函数

```
XmlBuilder* xml_builder_tree_create(void)

{

        XmlBuilder* thiz = (XmlBuilder*)calloc(1, sizeof(XmlBuilder));


        if(thiz != NULL)

        {

                PrivInfo* priv = (PrivInfo*)thiz->priv;


                thiz->on_start_element   = xml_builder_tree_on_start_element;
```

```
        thiz->on_end_element    = xml_builder_tree_on_end_element;

        thiz->on_text                   = xml_builder_tree_on_text;

        thiz->on_comment         = xml_builder_tree_on_comment;

        thiz->on_pi_element      = xml_builder_tree_on_pi_element;

        thiz->on_error                = xml_builder_tree_on_error;

        thiz->destroy                  = xml_builder_tree_destroy;



        priv->root = xml_node_create_normal("__root__", NULL);

        priv->current = priv->root;

    }



    return thiz;

}
```

和其它接口的创建函数一样，它只是把接口要求的函数指针指到具体的实现函数上。这里还创建了一个根结点__root__，以保证整棵树只有一个根结点。

o 实现 on_start_element

```
static void xml_builder_tree_on_start_element(XmlBuilder* thiz, const char* tag, const char** attrs)

{

    XmlNode* new_node = NULL;
```

```
	PrivInfo* priv = (PrivInfo*)thiz->priv;

	new_node = xml_node_create_normal(tag, attrs);

	xml_node_append_child(priv->current, new_node);

	priv->current = new_node;

	return;
}
```

这里创建了一个新的结点，并追加为 priv->current 的子结点，然后让 priv->current 指向新的结点。

o 实现 on_end_element

```
static void xml_builder_tree_on_end_element(XmlBuilder* thiz, const char* tag)
{
	PrivInfo* priv = (PrivInfo*)thiz->priv;

	priv->current = priv->current->parent;

	assert(priv->current != NULL);

	return;
}
```

这里只是让 priv->current 指向它的父结点。

o 实现 on_text

```
static void xml_builder_tree_on_text(XmlBuilder* thiz, const char* text, size_t length)

{

        XmlNode* new_node = NULL;

        PrivInfo* priv = (PrivInfo*)thiz->priv;



        new_node = xml_node_create_text(text);

        xml_node_append_child(priv->current, new_node);



        return;

}
```

这里创建一个文本结点， 并追加为 priv->current 的子结点。

o 实现 on_comment

```
static void xml_builder_tree_on_comment(XmlBuilder* thiz, const char* text, size_t length)

{

        XmlNode* new_node = NULL;

        PrivInfo* priv = (PrivInfo*)thiz->priv;
```

```
        new_node = xml_node_create_comment(text);

        xml_node_append_child(priv->current, new_node);


        return;

}
```

这里创建一个注释结点， 并追加为 priv->current 的子结点。

o 实现 on_pi_element

```
static void xml_builder_tree_on_pi_element(XmlBuilder* thiz, const char* tag, const char**
attrs)

{

        XmlNode* new_node = NULL;

        PrivInfo* priv = (PrivInfo*)thiz->priv;


        new_node = xml_node_create_pi(tag, attrs);

        xml_node_append_child(priv->current, new_node);


        return;

}
```

这里创建一个处理指令结点， 并追加为 priv->current 的子结点。

o 实现 on_error

```
static void xml_builder_tree_on_error(XmlBuilder* thiz, int line, int row, const char* message)

{

        fprintf(stderr, "(%d,%d) %s\n", line, row, message);



        return;

}
```

下面我们再看 XmlNode 的数据结构和主要函数：

o 数据结构

```
typedef struct _XmlNode

{

    XmlNodeType type;

    union

    {

        char* text;

        char* comment;

        XmlNodePi pi;

        XmlNodeNormal normal;
```

```
    }u;

    struct _XmlNode* parent;

    struct _XmlNode* children;

    struct _XmlNode* sibling;

}XmlNode;
```

type 决定了结点的类型，可以是处理指令(XML_NODE_PI)、文本(XML_NODE_TEXT)、注释(XML_NODE_COMMENT)或普通 TAG(XML_NODE_NORMAL)。

联合体用于存放具体结点信息。

parent 指向父结点。

children 指向第一个子结点。

sibling 指向下一个兄弟结点。

o 创建普通 TAG 结点

```
XmlNode* xml_node_create_normal(const char* name, const char** attrs)

{

    XmlNode* node = NULL;

    return_val_if_fail(name != NULL, NULL);



    if((node = calloc(1, sizeof(XmlNode))) != NULL)

    {

        int i = 0;
```

```
        node->type = XML_NODE_NORMAL;

        node->u.normal.name = strdup(name);



        if(attrs != NULL)

        {

            for(i = 0; attrs[i] != NULL && attrs[i+1] != NULL; i += 2)

            {

                xml_node_append_attr(node, attrs[i], attrs[i+1]);

            }

        }

    }



    return node;

}
```

o 创建处理指令结点

```
XmlNode* xml_node_create_pi(const char* name, const char** attrs)

{

    XmlNode* node = NULL;
```

```
return_val_if_fail(name != NULL, NULL);

if((node = calloc(1, sizeof(XmlNode))) != NULL)
{
    int i = 0;
    node->type = XML_NODE_PI;
    node->u.pi.name = strdup(name);
    if(attrs != NULL)
    {
        for(i = 0; attrs[i] != NULL && attrs[i+1] != NULL; i += 2)
        {
            xml_node_append_attr(node, attrs[i], attrs[i+1]);
        }
    }
}

return node;
```

```
}
```

o 创建文本结点

```
XmlNode* xml_node_create_text(const char* text)

{

    XmlNode* node = NULL;

    return_val_if_fail(text != NULL, NULL);



    if((node = calloc(1, sizeof(XmlNode))) != NULL)

    {

        node->type = XML_NODE_TEXT;

        node->u.text = strdup(text);

    }



    return node;

}
```

o 创建注释结点

```
XmlNode* xml_node_create_comment(const char* comment)

{
```

```
    XmlNode* node = NULL;

    return_val_if_fail(comment != NULL, NULL);



    if((node = calloc(1, sizeof(XmlNode))) != NULL)

    {

        node->type = XML_NODE_COMMENT;

        node->u.comment = strdup(comment);

    }



    return node;

}
```

o 追加一个兄弟结点

```
XmlNode* xml_node_append_sibling(XmlNode* node, XmlNode* sibling)

{

    return_val_if_fail(node != NULL && sibling != NULL, NULL);



    if(node->sibling == NULL)

    {
```

```
        /*没有兄弟结点，让兄弟结点指向sibling */

       node->sibling = sibling;

    }

    else

    {

        /*否则，把sibling追加为最后一个兄弟结点*/

        XmlNode* iter = node->sibling;

        while(iter->sibling != NULL) iter = iter->sibling;

        iter->sibling = sibling;

    }

    /*让兄弟结点的父结点指向自己的父结点*/



    sibling->parent = node->parent;



    return sibling;

}
```

o 追加一个子结点

```
XmlNode* xml_node_append_child(XmlNode* node, XmlNode* child)

{

    return_val_if_fail(node != NULL && child != NULL, NULL);



    if(node->children == NULL)

    {

        /*没有子结点，让子结点指向child */

        node->children = child;

    }

    else

    {

        /*否则，把child 追加为最后一个子结点*/

        XmlNode* iter = node->children;

        while(iter->sibling != NULL) iter = iter->sibling;

        iter->sibling = child;

    }

    /*让子结点的父结点指向自己*/
```

```
    child->parent = node;



    return child;

}
```

回头再看一下 XmlParser，XmlBuilder 及几个具体的 XmlBuilder 的实现，我们可以看到，它们的实现都非常简单，其实这完全 得益于 Builder 模式的设计方法。它利用分而治之的思想，把数据的解析和数据的处理分开，降低了实现的复杂度。其次它利用了抽象的思想，从而数据的解 析只关心处理数据处理的接口，而不关心的它的实现，使得数据解析和数据处理可以独立变化。

分而治之和抽象是降低复杂度最有效的手段之一，它们在 Builder 模式里得到了很好的体现。初学者应该多花些时间去体会。

本节示例代码请到**这里** 下载。

系统程序员成长计划-文本处理（三）

管道过滤器（Pipe-And-Filter）模式

按照《POSA(面向模式的软件架构)》里的说法，管道过滤器（Pipe-And-Filter）应该属于架构模式，因为它通常决定了一个系统的基 本架构。管道过滤器和生产流水线类似，在生产流水线上，原材料在流水线上经一道一道的工序，最后形成某种有用的产品。在管道过滤器中，数据经过一个一个的 过滤器，最后得到需要的数据。

o 基本的管道过滤器：

管道负责数据的传递，它把原始数据传递给第一个过滤器，把一个过滤器的输出传递给下一个过滤器，作为下一个过滤器的输入，重复这个过程直到处理结 束。要注意的是，管道只是对数据传输的抽象，它可能是管道，也可能是其它通信方式，甚至什么都没有(所有过滤器都在原始数据基础上进行处理)。

过滤器负责数据的处理，过滤器可以有多个，每个过滤器对数据做特定的处理，它们之间没有依赖关系，一个过滤器不必知道其它过滤器的存在。这种松耦合 的设计，使得过滤器只需要实现单一的功能，从而降低了系统的复杂度，也使得过滤器之间依赖最小，从而以更加灵活的组合来实现新的功能。

编译器就是基于管道过滤器模式设计的：

输入：源程序
预处理：负责宏展开和去掉注释等工作。
编译：进行词法分析、语法分析、语义分析、代码优化和代码产生。
汇编：负责把汇编代码转换成机器指令，生成目标文件。
链接：负责把多个目标文件、静态库和共享库链接成可执行文件/共享库。
输出：可执行文件/共享库。

o 复合过滤器

过滤器可以由多个其它过滤器组合起来的，比如上面的“编译”过程可以认为是一个复合过滤器：

输入：预处理之后的源代码。
词法分析：负责将源程序分解成一个一个的 token，这些 token 是组成源程序的基本单元。
语法分析：把词法分析得到的 token 解析成语法树。
语义分析：对语法树进行类型检查等语义分析。
代码优化：对语法树进行重组和修改，以优化代码的速度和大小。
代码产生：根据语法树产生汇编代码。
输出：汇编代码。

o 支持多个输入的过滤器

过滤器可以有多个输入。比如上面“链接”，它接收多个输入：

“链接”过滤器能接收多个数据源，如目标文件、静态库和共享库。

o 具有多个输出的过滤器

过滤器可以有多个输出。如多媒体播放器的解码过程：

输入：AVI 文件，包括音频和视频数据。
分离器：把音频和视频数据分离成两个流，音频数据传递给音频解码器，视频数据传递给视频解码器。
音频解码器：把压缩的音频数据解码成原始的音频数据。
视频解码器：把压缩的视频数据解码成原始的图像数据。
输出：音频数据传递给声卡，图像数据传递给显示器。

管道过滤器是最贴近程序员生活的模式，也是 Unix-like 系统的基本设计理念之一。作为 Linux 下的程序员，我们天天都使用这个模式。比如：

o 删除当前目录及子目录下的目标文件。

find -name *.o|xargs rm –f
find 是过滤器：它找出所有目标文件，它不需要关心查找文件的目的。
rm 是过滤器：它删除找到目标文件，rm 不需要关心文件名是如何得来的。

o 查看某个进程的栈的大小
grep stack /proc/2976/maps|sed -e “s/-/ /”|awk ‘{print strtonum(”0x”$2)-strtonum(”0x”$1)}’

/proc/2976/maps 是进程 2976 内存映射表。内容可能如下：

```
00110000-00111000 r-xp 00110000 00:00 0          [vdso]

00111000-0011b000 r-xp 00000000 08:01 154857     /lib/libnss_files-2.8.so

0011b000-0011c000 r--p 0000a000 08:01 154857     /lib/libnss_files-2.8.so

0011c000-0011d000 rw-p 0000b000 08:01 154857     /lib/libnss_files-2.8.so

00907000-00923000 r-xp 00000000 08:01 157280     /lib/ld-2.8.so

00923000-00924000 r--p 0001c000 08:01 157280     /lib/ld-2.8.so

00924000-00925000 rw-p 0001d000 08:01 157280     /lib/ld-2.8.so

00927000-00a8a000 r-xp 00000000 08:01 157281     /lib/libc-2.8.so

00a8a000-00a8c000 r--p 00163000 08:01 157281     /lib/libc-2.8.so

00a8c000-00a8d000 rw-p 00165000 08:01 157281     /lib/libc-2.8.so

00a8d000-00a90000 rw-p 00a8d000 00:00 0

00abd000-00ac0000 r-xp 00000000 08:01 157284     /lib/libdl-2.8.so

00ac0000-00ac1000 r--p 00002000 08:01 157284     /lib/libdl-2.8.so

00ac1000-00ac2000 rw-p 00003000 08:01 157284     /lib/libdl-2.8.so
```

```
0383f000-03855000 r-xp 00000000 08:01 157307     /lib/libtinfo.so.5.6

03855000-03858000 rw-p 00015000 08:01 157307     /lib/libtinfo.so.5.6

08047000-080fa000 r-xp 00000000 08:01 1180910    /bin/bash

080fa000-080ff000 rw-p 000b3000 08:01 1180910    /bin/bash

080ff000-08104000 rw-p 080ff000 00:00 0

088bd000-088ff000 rw-p 088bd000 00:00 0          [heap]

b7bfb000-b7bfd000 rw-p b7bfb000 00:00 0

b7bfd000-b7c04000 r--s 00000000 08:01 237138     /usr/lib/gconv/gconv-modules.cache

b7c04000-b7d1e000 r--p 047d3000 08:01 237437     /usr/lib/locale/locale-archive

b7d1e000-b7d5e000 r--p 0236e000 08:01 237437     /usr/lib/locale/locale-archive

b7d5e000-b7f5e000 r--p 00000000 08:01 237437     /usr/lib/locale/locale-archive

b7f5e000-b7f60000 rw-p b7f5e000 00:00 0

bfe5e000-bfe73000 rw-p bffeb000 00:00 0          [stack]
```

grep 是过滤器：它从文件/proc/2976/maps 里找到下面这行数据。

bfe5e000-bfe73000 rw-p bffeb000 00:00 0 [stack]

sed 是过滤器：它把‘-’替换成‘ ’，数据变成下面的内容。
bfe5e000 bfe73000 rw-p bffeb000 00:00 0 [stack]

awk 是过滤器：它计算 0xbfe73000 和 0x bfe5e000 差值，并打印出来。

下面我们来看看，管道过滤器在程序里的实现方式。这里我们以 TinyMail 为例，TinyMail 是一款针对移动设备定制的邮件客户端软件。它使用 camel-lite 完成邮件内容解析和传输。

Camel-lite 对邮件内容的处理基本上基于管道过滤器模式的。

CamelMimeFilter 是过滤器接口，所有过滤器都要实现它要求的接口函数：

```
struct _CamelMimeFilterClass {

    CamelObjectClass parent_class;



    void (*filter)(CamelMimeFilter *f,

               char *in, size_t len, size_t prespace,

               char **out, size_t *outlen, size_t *outprespace);

    void (*complete)(CamelMimeFilter *f,

              char *in, size_t len, size_t prespace,

              char **out, size_t *outlen, size_t *outprespace);

    void (*reset)(CamelMimeFilter *f);

};
```

CamelMimeFilterClass 是从 CamelObjectClass 继承过来的。这里的接口定义和我们前面所讲的接口定义有些差别，但原理上都是一样，通过函数指针来抽象具体的功能。这里要求实现三个接口函数：

filter：过滤器的处理函数。
complete：过滤器的处理函数。与 filter 不同的是，调用 complete 之后不能调其它 filter。
reset：重置当前 filter 的状态。

Camel 实现了很多 Filter，其中 CamelMimeFilterBasic 实现了邮件基本的编码和解析功能。它的 filter 函数实现如下：

```
static void
```

```
filter(CamelMimeFilter *mf, char *in, size_t len, size_t prespace, char **out, size_t *outlen,
size_t *outprespace)

{

    CamelMimeFilterBasic *f = (CamelMimeFilterBasic *)mf;

    size_t newlen;



    switch(f->type) {

    case CAMEL_MIME_FILTER_BASIC_BASE64_ENC:

        /* wont go to more than 2x size (overly conservative) */

        camel_mime_filter_set_size(mf, len*2+6, FALSE);

        newlen = g_base64_encode_step((const guchar *) in, len, TRUE, mf->outbuf, &f->state, &f-
>save);

        g_assert(newlen <= len*2+6);

        break;

    case CAMEL_MIME_FILTER_BASIC_QP_ENC:

        /* *4 is overly conservative, but will do */

        camel_mime_filter_set_size(mf, len*4+4, FALSE);

        newlen = camel_quoted_encode_step((unsigned char *) in, len, (unsigned char *) mf-
>outbuf, &f->state, (gint *) &f->sa
```

```
ve);

        g_assert(newlen <= len*4+4);

        break;

    case CAMEL_MIME_FILTER_BASIC_UU_ENC:

        /* won't go to more than 2 * (x + 2) + 62 */

        camel_mime_filter_set_size (mf, (len + 2) * 2 + 62, FALSE);

        newlen = camel_uuencode_step ((unsigned char *) in, len, (unsigned char *) mf->outbuf,
f->uubuf, &f->state, (guint32

*) &f->save);

        g_assert (newlen <= (len + 2) * 2 + 62);

        break;

    case CAMEL_MIME_FILTER_BASIC_BASE64_DEC:

        /* output can't possibly exceed the input size */

        camel_mime_filter_set_size(mf, len+3, FALSE);

        newlen = g_base64_decode_step(in, len, (guchar *) mf->outbuf, &f->state, (guint *) &f-
>save);

        g_assert(newlen <= len+3);

        break;
```

```
    case CAMEL_MIME_FILTER_BASIC_QP_DEC:

        /* output can't possibly exceed the input size */

        camel_mime_filter_set_size(mf, len + 2, FALSE);

        newlen = camel_quoted_decode_step((unsigned char *) in, len, (unsigned char *) mf-
>outbuf, &f->state, (gint *) &f->sa

ve);

        g_assert(newlen <= len + 2);

        break;

    case CAMEL_MIME_FILTER_BASIC_UU_DEC:

        if (!(f->state & CAMEL_UUDECODE_STATE_BEGIN)) {

            register char *inptr, *inend;

            size_t left;



            inptr = in;

            inend = inptr + len;



            while (inptr < inend) {

                left = inend - inptr;
```

```
if (left < 6) {

    if (!strncmp (inptr, "begin ", left))

        camel_mime_filter_backup (mf, inptr, left);

    break;

} else if (!strncmp (inptr, "begin ", 6)) {

    for (in = inptr; inptr < inend && *inptr != '\n'; inptr++);

    if (inptr < inend) {

        inptr++;

        f->state |= CAMEL_UUDECODE_STATE_BEGIN;

        /* we can start uudecoding... */

        in = inptr;

        len = inend - in;

    } else {

        camel_mime_filter_backup (mf, in, left);

    }

    break;

}
```

```
            /* go to the next line */

            for ( ; inptr < inend && *inptr != '\n'; inptr++);


            if (inptr < inend)

                inptr++;

        }

    }


    if ((f->state & CAMEL_UUDECODE_STATE_BEGIN) && !(f->state & CAMEL_UUDECODE_STATE_END)) {

        /* "begin <mode> <filename>\n" has been found, so we can now start decoding */

        camel_mime_filter_set_size (mf, len + 3, FALSE);

        newlen = camel_uudecode_step ((unsigned char *) in, len, (unsigned char *) mf-
>outbuf, &f->state, (guint32 *) &f-

>save);

    } else {

        newlen = 0;

    }
```

```
        break;

    default:

        g_warning ("unknown type %u in CamelMimeFilterBasic", f->type);

        goto donothing;

    }



    *out = mf->outbuf;

    *outlen = newlen;

    *outprespace = mf->outpre;



    return;

donothing:

    *out = in;

    *outlen = len;

    *outprespace = prespace;

}
```

这个过滤器实现了下面三种编码方式的编码和解码：

1. UU(Unix-to-Unix encoding)：
2. Base64

3. QP(Quote-Printable)

Camel 还提供了其它一些过滤器，如：
CamelMimeFilterGZip：压缩和解压
CamelMimeFilterHTML：去掉 HTML Tag。
CamelMimeFilterCRLF：使用\r\n 作为换行符。
CamelMimeFilterToHTML：加上 HTML Tag。
CamelMimeFilterCharset：字符集转换。

所有的过滤器由 CamelStreamFilter 来组合，CamelStreamFilter 提供了下面两个函数：

增加过滤器：
int camel_stream_filter_add (CamelStreamFilter *stream, CamelMimeFilter *filter);

移除过滤器：
void camel_stream_filter_remove (CamelStreamFilter *stream, int id);

CamelStreamFilter 实现了 CamelStream 接口，这里应用了前面所讲的装饰模式，它不改变 CamelStream 的接口，但给 CamelStream 加上了数据转换功能。它的创建函数如下：

CamelStreamFilter *camel_stream_filter_new_with_stream (CamelStream *stream);

传入一个 CamelStream 对象，然后对这个对象进行装饰。在读写数据时，调用相应的 Filter，下面是写函数的实现：

```
do_read (CamelStream *stream, char *buffer, size_t n)

{

    CamelStreamFilter *filter = (CamelStreamFilter *)stream;

    struct _CamelStreamFilterPrivate *p = _PRIVATE(filter);

    ssize_t size;

    struct _filter *f;



    p->last_was_read = TRUE;



    g_check(p->realbuffer);
```

```
if (p->filteredlen<=0) {

    size_t presize = READ_PAD;



    size = camel_stream_read(filter->source, p->buffer, READ_SIZE);

    if (size <= 0) {

        /* this is somewhat untested */

        if (camel_stream_eos(filter->source)) {

            f = p->filters;

            p->filtered = p->buffer;

            p->filteredlen = 0;

            while (f) {

                camel_mime_filter_complete(f->filter, p->filtered, p->filteredlen,

                                presize, &p->filtered, &p->filteredlen, &presize);

                g_check(p->realbuffer);

                f = f->next;

            }
```

```
                size = p->filteredlen;

                p->flushed = TRUE;

            }

            if (size <= 0)

                return size;

        } else {

            f = p->filters;

            p->filtered = p->buffer;

            p->filteredlen = size;

            d(printf ("\n\nOriginal content (%s): '", ((CamelObject *)filter->source)->klass->name));

            d(fwrite(p->filtered, sizeof(char), p->filteredlen, stdout));

            d(printf("'\n"));



            while (f) {

                camel_mime_filter_filter(f->filter, p->filtered, p->filteredlen, presize,

                             &p->filtered, &p->filteredlen, &presize);

                g_check(p->realbuffer);
```

```
            d(printf ("Filtered content (%s): '", ((CamelObject *)f->filter)->klass->name));

            d(fwrite(p->filtered, sizeof(char), p->filteredlen, stdout));

            d(printf("'\n"));



            f = f->next;

        }

    }

}



size = MIN(n, p->filteredlen);

memcpy(buffer, p->filtered, size);

p->filteredlen -= size;

p->filtered += size;



g_check(p->realbuffer);



return size;
```

}

这里先调用 camel_stream_read 读取数据，然后依次调用 fitler 对数据进行处理，最后把数据返回给调用者。do_write 的过程类似，CamelStreamFilter 对编码和解码都支持，而使用者不用关心。

管道过滤器模式应用相当广泛，它不限于文本数据处理，任何以数据处理为中心的系统，都可以用管道过滤器模式作为基本架构。