

C 语言编程透视

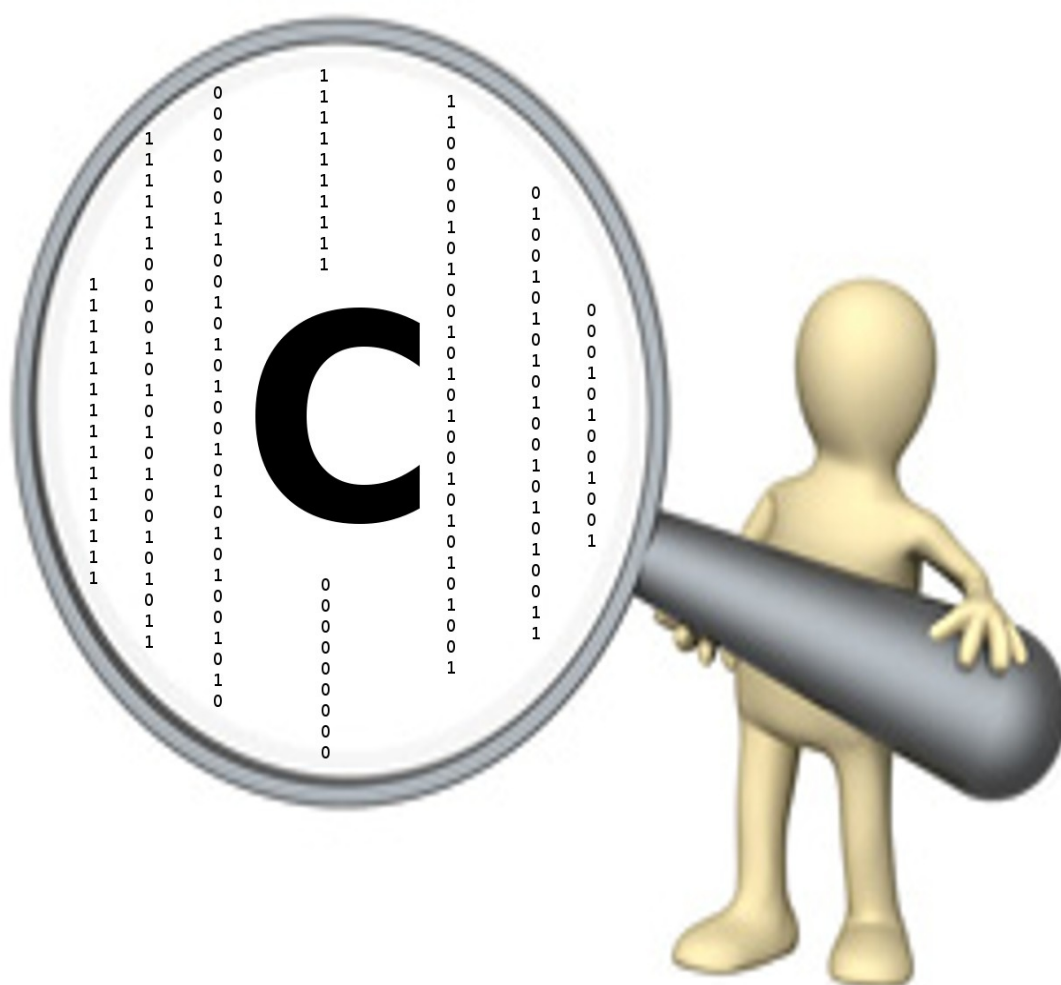


Table of Contents

1. [Introduction](#)
2. [版本修訂歷史](#)
3. [前言](#)
4. [把 Vim 打造成源代碼編輯器](#)
5. [Gcc 編譯的背後](#)
6. [程序執行的一剎那](#)
7. [動態符號鏈接的細節](#)
8. [緩衝區溢出與注入分析](#)
9. [進程的內存映像](#)
10. [進程和進程的基本操作](#)
11. [打造史上最小可執行ELF文件\(45字節\)](#)
12. [代碼測試、調試與優化](#)

本書來源：[開源書籍：C 語言編程透視](#) (by 泰曉科技)
報名參與：[Star/fork GitHub 倉庫](#) 併發送 *Pull Request*
關注我們：[掃描二維碼](#) 關注 @泰曉科技 微博和微信公眾號
贊助我們：[贊助 6.88 ¥](#), [更多原創開源書籍](#)需要您的支持 ^o^

C 語言編程透視

v 0.2

本書與《[深入淺出 Hello World](#)》有著類似的心路歷程，旨在以實驗的方式去探究類似 `Hello World` 這樣的小程序在開發與執行過程中的微妙變化，一層層揭開 C 語言程序開發過程的神祕面紗，透視背後的神秘，不斷享受醍醐灌頂的美妙。

介紹

- 項目首頁：<http://www.tinylab.org/hello-c-world>
- 代碼倉庫：<https://github.com/tinyclub/open-c-book>
- 在線閱讀：<http://tinylab.gitbooks.io/cbook>

更多背景和計劃請參考：[前言](#)。

安裝

以 Ubuntu 為例：

```
$ sudo aptitude install -y retext git nodejs npm
$ sudo ln -fs /usr/bin/nodejs /usr/bin/node
$ sudo aptitude install -y calibre fonts-arphic-gbsn00lp
$ sudo npm install gitbook-cli -g
```

下載

```
$ git clone https://github.com/tinyclub/open-c-book.git
$ cd open-c-book/
```

編譯

```
$ gitbook build // 編譯成網頁
$ gitbook pdf // 編譯成 pdf
```

糾錯

歡迎大家指出不足，如有任何疑問，請郵件聯繫 [wuzhangjin at gmail dot com](mailto:wuzhangjin@gmail.com) 或者直接修復並提交 Pull Request。

版權

本書採用  協議發佈，詳細版權信息請參考 [CC BY NC ND 4.0](#)。

關注我們

- [新浪微博](#)



- [微信公眾號](#)



贊助我們

- [微信掃碼贊助原創](#)



- 訪問 [泰曉開源小店](#) 支持心儀項目



更多原創開源書籍

- [Shell 編程範例](#)
- [嵌入式 Linux 知識庫\(eLinux.org 中文版\)](#)
- [Linux 內核文檔\(Linux Documentation/ 中文版\)](#)

版本修訂歷史

Revision	Author	From	Date	Description
0.2	@吳章金falcon	@泰曉科技	2015/07/23	調整格式，修復鏈接
0.1	@吳章金falcon	@泰曉科技	2014/01/19	初稿

前言

- 背景
- 現狀
- 計劃

背景

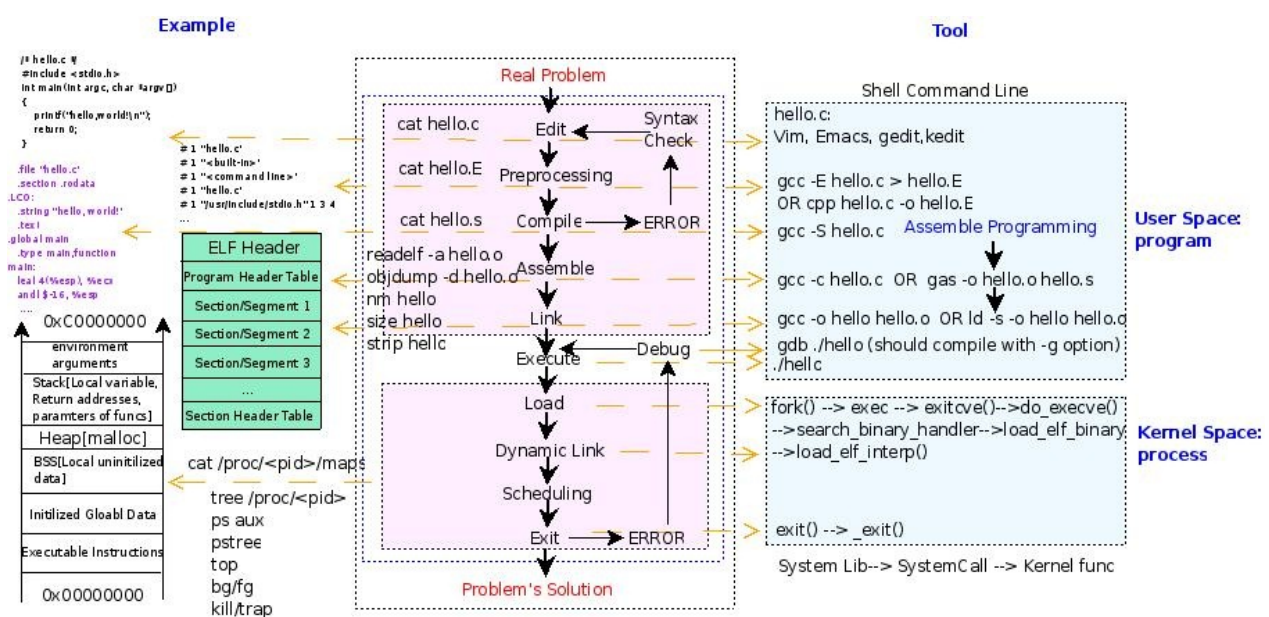
2007 年開始系統地學習 Shell 編程，並在[蘭大開源社區](#)寫了序列文章。

在編寫《Shell 編程範例》文章的《進程操作》一章時，為了全面瞭解進程的來龍去脈，對程序開發過程的細節、ELF 格式的分析、進程的內存映像等進行了全面地梳理，後來搞得“雪球越滾越大”，甚至脫離了 Shell 編程關注的內容。所以想了個小辦法，“大事化小，小事化了”，把涉及到的內容進行了分解，進而演化成另外一個完整的序列。

2008 年 3 月 1 日，當初步完成整個序列時，做了如下的小結：

到今天，關於“Linux 下 C 語言開發過程”的一個簡單視圖總算粗略地完成了，從寒假之前的一段時間到現在過了將近一個月左右吧。寫這個主題的目的源自“Shell 編程範例之進程操作”，當寫到這一章時，突然對進程的由來、本身和去向感到“迷惑不解”。所以想著好好花些時間來弄清楚它們，現在發現，這個由來就是這裡的程序開發過程，進程來自一個普通的文本文件，在這裡是 C 語言程序，C 語言程序經過編輯、預處理、編譯、彙編、鏈接、執行而成為一個進程；而進程本身呢？當一個可執行文件被執行以後，有了 exec 調用，被程序解釋器映射到了內存中，有了它的內存映像；而進程的去向呢？通過不斷地執行指令和內存映像的變化，進程完成著各項任務，等任務完成以後就可以退出了（exit）。

這樣一份視圖實際上是在寒假之前繪好的，可以從下圖中看到它；不過到現在才明白背後的很多細節。這些細節就是這個序列的每個篇章，可以對照“視圖”來閱讀它們。



C Language Programming Procedure in Linux

現狀

目前整個序列大部分都已經以 Blog 的形式寫完，大體結構目下：

- [《把 VIM 打造成源代碼編輯器》](#)
 - 源代碼編輯過程：用 VIM 編輯代碼的一些技巧
 - 更新時間：2008-2-22
- [《GCC 編譯的背後》](#)
 - 編譯過程：預處理、編譯、彙編、鏈接
 - 第一部分：《預處理和編譯》（更新時間：2008-2-22）
 - 第二部分：《彙編和鏈接》（更新時間：2008-2-22）
- [《程序執行的那一剎那》](#)
 - 執行過程：當從命令行輸入一個命令之後
 - 更新時間：2008-2-15
- [《進程的內存映像》](#)
 - 進程加載過程：程序在內存裡是個什麼樣子？
 - 第一部分（討論“緩衝區溢出和注入”問題）（更新時間：2008-2-13）
 - 第二部分（討論進程的內存分佈情況）（更新時間：2008-6-1）
- [《進程和進程的基本操作》](#)
 - 進程操作：描述進程相關概念和基本操作
 - 更新時間：2008-2-21
- [《動態符號鏈接的細節》](#)
 - 動態鏈接過程：函數 puts/printf 的地址在哪裡？
 - 更新時間：2008-2-26
- [《打造史上最小可執行ELF文件》](#)
 - ELF 詳解：從“減肥”的角度一層一層剖開 ELF 文件，最終獲得一個可打印 Hello World 的 45 字節 ELF 可執行文件
 - 更新時間：2008-2-23
- [《代碼測試、調試與優化小結》](#)
 - 程序開發過後：內存溢出了嗎？有緩衝區溢出？代碼覆蓋率如何測試呢？怎麼調試彙編代碼？有哪些代碼優化技巧和方法呢？
 - 更新時間：2008-2-29

計劃

考慮到整個 Linux 世界的蓬勃發展，Linux 和 C 語言的應用環境越來越多，相關使用群體會不斷增加，所以最近計劃把該序列重新整理，以自由書籍的方式不斷更新，以便惠及更多的讀者。

打算重新規劃、增補整個序列，並以開源項目的方式持續維護，並通過 [泰曉科技|TinLab.org](#) 平臺接受讀者的反饋，直到正式發行出版。

自由書籍將會維護在 [泰曉科技](#) 的[項目倉庫](#)中。項目相關信息如下：

- 項目首頁：<http://www.tinylab.org/project/hello-c-world/>
- 代碼倉庫：<https://github.com/tinyclub/open-c-book.git>

歡迎大家指出本書初稿中的不足，甚至參與到相關章節的寫作、校訂和完善中來。

如果有時間和興趣，歡迎參與。可以通過 [泰曉科技](#) 聯繫我們，或者直接關注微博 [@泰曉科技](#) 並私信我們。

把 Vim 打造成源代碼編輯器

- [前言](#)
- [常規操作](#)
 - [打開文件](#)
 - [編輯文件](#)
 - [保存文件](#)
 - [退出/關閉](#)
- [命令模式](#)
 - [編碼風格與 indent 命令](#)
 - [用 Vim 命令養成良好編碼風格](#)
- [相關小技巧](#)
- [後記](#)
- [參考資料](#)

前言

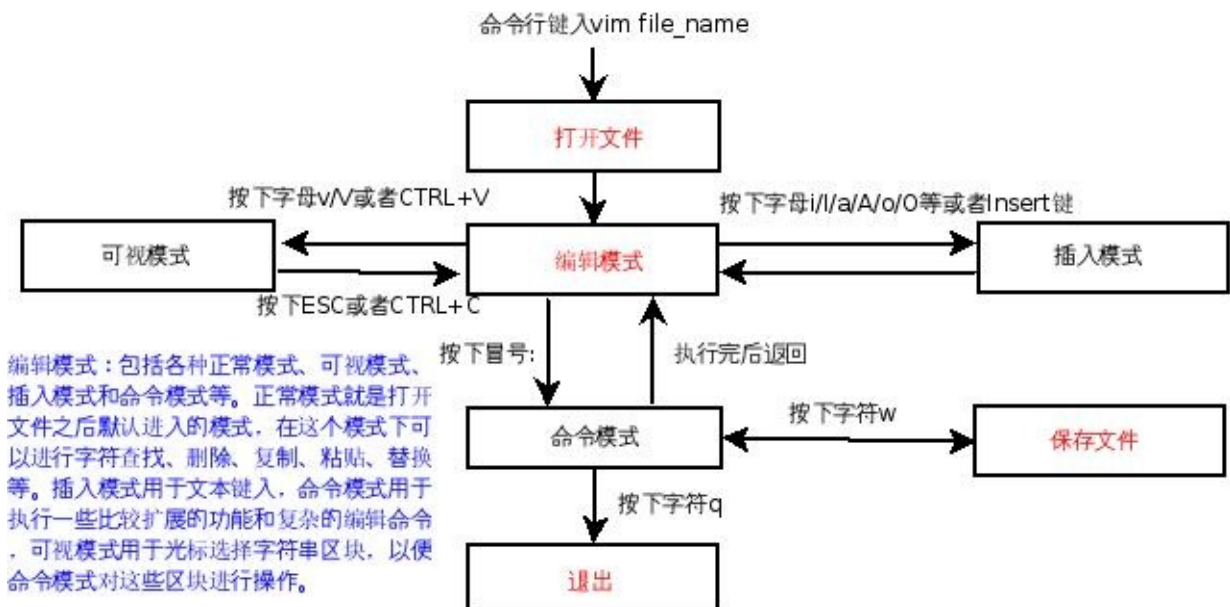
程序開發過程中，源代碼的編輯主要是為了實現算法，結果則是一些可閱讀的、便於檢錯的、可移植的文本文件。如何產生一份良好的源代碼，這不僅需要一些良好的編輯工具，還需要開發人員養成良好的編程修養。

Linux 下有很多優秀的程序編輯工具，包括專業的文本編輯器和一些集成開發環境（IDE）提供的編輯工具，前者的代表作有 Vim 和 Emacs，後者的代表作則有 Eclipse，Kdevelop，Anjuta 等，這裡主要介紹 Vim 的基本使用和配置。

常規操作

通過 Vim 進行文本編輯的一般過程包括：文件的打開、編輯、保存、關閉/退出，而編輯則包括插入新內容、替換已有內容、查找內容，還包括複製、粘貼、刪除等基本操作。

該過程如下圖：



下面介紹幾個主要操作：

打開文件

在命令行下輸入 `vim 文件名` 即可打開一個新文件並進入 Vim 的“編輯模式”。

編輯模式可以切換到命令模式（按下字符 `:`）和插入模式（按下字母 `a/A/i/I/o/O/s/S/c/C` 等或者 `Insert` 鍵）。

編輯模式下，Vim 會把鍵盤輸入解釋成 Vim 的編輯命令，以便實現諸如字符串查找（按下字母 `/`）、文本複製（按下字母 `yy`）、粘貼（按下字母 `pp`）、刪除（按下字母 `d` 等）、替換（`s`）等各種操作。

當按下 `a/A/i/I/o/O/s/S/c/C` 等字符時，Vim 先執行這些字符對應命令的動作（比如移動光標到某個位置，刪除某些字符），然後進入插入模式；進入插入模式後可以通過按下 `ESC` 鍵或者是 `CTRL+C` 返回到編輯模式。

在編輯模式下輸入冒號 `:` 後可進入命令模式，通過它可以完成一些複雜的編輯功能，比如進行正則表達式匹配替換，執行 Shell 命令（按下 `!` 命令）等。

實際上，無論是插入模式還是命令模式都是編輯模式的一種。而編輯模式卻並不止它們兩個，還有字符串查找、刪除、替換等。

需要提到的是，如果在編輯模式按下字母 `v/V` 或者是 `CTRL+V`，可以用光標選擇一個區塊，進而結合命令模式對這一個區塊進行特定的操作。

編輯文件

打開文件以後即可進入編輯模式，這時可以進行各種編輯操作，包括插入、複製、刪除、替換字符。其中兩種比較重要的模式經常被“獨立”出來，即上面提到的插入模式和命令模式。

保存文件

在退出之前需切換到命令模式，輸入命令 `w` 以便保存各種編輯後的內容，如果想取消某種操作，可以用 `u` 命令。如果打開 Vim 編輯器時沒有設定文件名，那麼在按下 `w` 命令時會提示沒有文件名，此時需要在 `w` 命令後加上需要保存的文件名。

退出/關閉

保存好內容後就可退出，只需在命令模式下鍵入字符 `q`。如果對文件內容進行了編輯，卻沒有保存，那麼 Vim 會提示，如果不想保存之前的編輯動作，那麼可按下字符 `q` 並且在之後跟上一個感嘆號 `!`，這樣會強制退出，不保存最近的內容變更。

命令模式

這裡需要著重提到的是 Vim 的命令模式，它是 Vim 擴展各種新功能的接口，用戶可以通過它啟用和撤銷某個功能，開發人員則可通過它為用戶提供新的功能。下面主要介紹通過命令模式這個接口定製 Vim 以便我們更好地進行源代碼的編輯。

編碼風格與 indent 命令

先提一下編碼風格。剛學習編程時，代碼寫得很“難看”（不方便閱讀，不方便檢錯，看不出任何邏輯結構），常常導致心情不好，而且排錯也很困難，所以逐漸意識到代碼編寫需要規範，即養成良好的編碼風格，如果換成俗話，那就是代碼的排版，讓代碼好看一些。雖說“編程的”（高雅一些則稱開發人員）不一定懂藝術，不過這個應該不是“搞藝術的”（高雅一些應該是文藝工作人員）的特權，而是我們應該具備的專業素養。在 Linux 下，比較流行的“行業”風格有 KR 的編碼風格、GNU 的編碼風格、Linux 內核的編碼風格（基於 KR 的，縮進是 8 個空格）等，它們都可以通過 `indent` 命令格式化，對應的選項分別是 `-kr`，`-gnu`，`-kr -i8`。下面演示用 `indent` 命令把代碼格式化成上面的三種風格。

這樣糟糕的編碼風格看著會讓人想“哭”，太難閱讀啦：

```
$ cat > test.c
/* test.c -- a test program for using indent */
#include<stdio.h>

int main(int argc, char *argv[])
{
    int i=0;
    if (i != 0) {i++; }
    else {i--; };
    for(i=0;i<5;i++)j++;
    printf("i=%d,j=%d\n",i,j);

    return 0;
}
```

格式化成 KR 風格，好看多了：

```
$ indent -kr test.c
$ cat test.c
/* test.c -- a test program for using indent */
```

```
#include<stdio.h>

int main(int argc, char *argv[])
{
    int i = 0;
    if (i != 0) {
        i++;
    } else {
        i--;
    };
    for (i = 0; i < 5; i++)
        j++;
    printf("i=%d,j=%d\n", i, j);
    return 0;
}
```

採用 GNU 風格，感覺不如 KR 的風格，處理 `if` 語句時增加了代碼行，卻並沒明顯改進效果：

```
$ indent -gnu test.c
$ cat test.c
/* test.c -- a test program for using indent */
#include<stdio.h>

int
main (int argc, char *argv[])
{
    int i = 0;
    if (i != 0)
    {
        i++;
    }
    else
    {
        i--;
    };
    for (i = 0; i < 5; i++)
        j++;
    printf ("i=%d,j=%d\n", i, j);
    return 0;
}
```

實際上 `indent` 命令有時候會不靠譜，也不建議“先汙染再治理”，而是從一開始就堅持“可持續發展”的觀念，在寫代碼時就逐步養成良好的風格。

需要提到的是，Linux 的編碼風格描述文件為內核源碼下的 [Documentation/CodingStyle](#)，而相應命令為 [scripts/Lindent](#)。

用 Vim 命令養成良好編碼風格

從演示中可看出編碼風格真地很重要，但是如何養成良好的編碼風格呢？經常練習，遵守某個編碼風格，一如既往。不過這還不夠，如果沒有一個好編輯器，習慣也很難養成。而 Vim 提供了很多輔助我們養成良好編碼習慣的功能，這些都通過它的命令模式提供。現在分開介紹幾個功能；

Vim 命令	功效
<code>:syntax on</code>	語法加“靚”（亮）
<code>:syntax off</code>	語法不加“靚”（亮）
<code>:set cindent</code>	C 語言自動縮進（可簡寫為 <code>set cin</code> ）
<code>:set sw=8</code>	自動縮進寬度（需要 <code>set cin</code> 才有用）
<code>:set ts=8</code>	設定 TAB 寬度
<code>:set number</code>	顯示行號
<code>:set nonumber</code>	不顯示行號
<code>:set sm</code>	括號自動匹配

這幾個命令對代碼編寫來說非常有用，可以考慮把它們全部寫到 `~/.vimrc` 文件（Vim 啟動時會去加載這個文件裡頭的內容）中，如：

```
$ cat ~/.vimrc
:set number
:set sw=8
:set ts=8
:set sm
:set cin
:syntax on
```

相關小技巧

需要補充的幾個技巧有：

- 對註釋自動斷行
 - 在編輯模式下，可通過 `gqap` 命令對註釋自動斷行（每行字符個數可通過命令模式下的 `set textwidth=個數` 設定）
- 跳到指定行
 - 命令模式下輸入數字可以直接跳到指定行，也可在打開文件時用 `vim +數字 文件名` 實現相同的功能。
- 把 C 語言輸出為 html
 - 命令模式下的 `TOhtml` 命令可把 C 語言輸出為 html 文件，結合 `syntax on`，可產生比較好的網頁把代碼發佈出去。
- 註釋掉代碼塊
 - 先切換到可視模式（編輯模式下按字母 `v` 可切換過來），用光標選中一片代碼，然後通過命令模式下的命令 `s#^#/#g` 把某這片代碼註釋掉，這非常方便調試某一片代碼的功能。
- 切換到粘貼模式解決 Insert 模式自動縮進的問題
 - 命令模式下的 `set paste` 可解決複製本來已有縮進的代碼的自動縮進問題，後可執行 `set nopaste` 恢復自動縮進。

- 使用 Vim 最新特性
 - 為了使用最新的 Vim 特性，可用 `set nocp` 取消與老版本的 Vi 的兼容。
- 全局替換某個變量名
 - 如發現變量命名不好，想在整個代碼中修改，可在命令模式下用 `%s#old_variable#new_variable#g` 全局替換。替換的時注意變量名是其他變量一部分的情況。
- 把縮進和 TAB 鍵都替換為空格
 - 可考慮設置 `expandtab`，即 `set et`，如果要把以前編寫的代碼中的縮進和 TAB 鍵都替換掉，可以用 `retab`。
- 關鍵字自動補全
 - 輸入一部分字符後，按下 `CTRL+P` 即可。比如先輸入 `prin`，然後按下 `CTRL+P` 就可以補全了。
- 在編輯模式下查看手冊
 - 可把光標定位在某個函數，按下 `Shift+k` 就可以調出 `man`，很有用。
- 刪除空行
 - 在命令模式下輸入 `g/^$/d`，前面 `g` 命令是擴展到全局，中間是匹配空行，後面 `d` 命令是執行刪除動作。用替換也可以實現，鍵入 `%s#^\n##g`，意思是把所有以換行開頭的行全部替換為空。類似地，如果要把多個空行轉換為一個可以輸入 `g/^\n$/d` 或者 `%s#^\n$##g`。
- 創建與使用代碼交叉引用
 - 注意利用一些有用的插件，比如 `ctags`，`cscope` 等，可以提高代碼閱讀、分析的效率。特別是開放的軟件。
- 回到原位置
 - 在用 `ctags` 或 `cscope` 時，當找到某個標記後，又想回到原位置，可按下 `CTRL+T`。

這裡特別提到 `cscope`，為了加速代碼的閱讀，還可以類似上面在 `~/.vimrc` 文件中通過 `map` 命令預定義一些快捷方式，例如：

```
if has("cscope")
    set csprg=/usr/bin/cscope
    set cst=0
    set cst
    set nocsverb
    " add any database in current directory
    if filereadable("cscope.out")
        cs add cscope.out
    " else add database pointed to by environment
    elseif $CSCOPE_DB != ""
        cs add $CSCOPE_DB
    endif
    set csverb
:map \ :cs find g <C-R>=expand("<C-R>")<CR><CR>
:map s :cs find s <C-R>=expand("<C-R>")<CR><CR>
:map t :cs find t <C-R>=expand("<C-R>")<CR><CR>
:map c :cs find c <C-R>=expand("<C-R>")<CR><CR>
:map C :cs find d <C-R>=expand("<C-R>")<CR><CR>
:map f :cs find f <C-R>=expand("<C-R>")<CR><CR>
```

```
endif
```

因為 `s, t, c, C, f` 這幾個 Vim 的默認快捷鍵用得不多，所以就把它們給作為快捷方式映射了，如果已經習慣它們作為其他的快捷方式就換別的字符吧。

注 上面很多技巧中用到了正則表達式，關於這部分請參考：[正則表達式 30 分鐘入門教程](#)。

更多的技巧可以看看後續資料。

後記

實際上，在源代碼編寫時還有很多需要培養的“素質”，例如源文件的開頭註釋、函數的註釋，變量的命名等。這方面建議看看參考資料裡的編程修養、內核編碼風格、網絡上流傳的《華為編程規範》，以及《C Traps & Pitfalls》，《C-FAQ》等。

參考資料

- Vim 官方教程，在命令行下鍵入 `vimtutor` 即可
- vim 實用技術序列
 - [實用技巧](#)
 - [常用插件](#)
 - [定製 Vim](#)
- [Graphical vi-vim Cheat Sheet and Tutorial](#)
- [Documentation/CodingStyle](#)
- [scripts/Lindent](#)。
- [正則表達式 30 分鐘入門教程](#)
- [也談 C 語言編程風格：完成從程序員到工程師的蛻變](#)
- Vim 高級命令集錦
- 編程修養
- C Traps & Pitfalls
- C FAQ

Gcc 編譯的背後

- [前言](#)
- [預處理](#)
 - [簡述](#)
 - [打印出預處理之後的結果](#)
 - [在命令行定義宏](#)
- [編譯（翻譯）](#)
 - [簡述](#)
 - [語法檢查](#)
 - [編譯器優化](#)
 - [生成彙編語言文件](#)
- [彙編](#)
 - [簡述](#)
 - [生成目標代碼](#)
 - [ELF 文件初次接觸](#)
 - [ELF 文件的結構](#)
 - [三種不同類型 ELF 文件比較](#)
 - [ELF 主體：節區](#)
 - [彙編語言文件中的節區表述](#)
- [鏈接](#)
 - [簡述](#)
 - [可執行文件的段：節區重排](#)
 - [鏈接背後的故事](#)
 - [用 ld 完成鏈接過程](#)
 - [C++ 構造與析構：crtbegin.o 和 crtend.o](#)
 - [初始化與退出清理：crti.o 和 crtn.o](#)
 - [C 語言程序真正的入口](#)
 - [鏈接腳本初次接觸](#)
- [參考資料](#)

前言

平時在 Linux 下寫代碼，直接用 `gcc -o out in.c` 就把代碼編譯好了，但是這背後到底做了什麼呢？

如果學習過《編譯原理》則不難理解，一般高級語言程序編譯的過程莫過於：預處理、編譯、彙編、鏈接。

`gcc` 在後臺實際上也經歷了這幾個過程，可以通過 `-v` 參數查看它的編譯細節，如果想看某個具體的編譯過程，則可以分別使用 `-E`，`-S`，`-c` 和 `-O`，對應的後臺工具則分別為 `cpp`，`cc1`，`as`，`ld`。

下面將逐步分析這幾個過程以及相關的內容，諸如語法檢查、代碼調試、彙編語言等。

預處理

簡述

預處理是 C 語言程序從源代碼變成可執程序的第一步，主要是 C 語言編譯器對各種預處理命令進行處理，包括頭文件的包含、宏定義的擴展、條件編譯的選擇等。

以前沒怎麼“深入”預處理，腦子對這些東西總是很模糊，只記得在編譯的基本過程（詞法分析、語法分析）之前還需要對源代碼中的宏定義、文件包含、條件編譯等命令進行處理。這三類的指令很常見，主要有 `#define`，`#include` 和 `#ifdef ... #endif`，要特別地注意它們的用法。

`#define` 除了可以獨立使用以便靈活設置一些參數外，還常常和 `#ifdef ... #endif` 結合使用，以便靈活地控制代碼塊的編譯與否，也可以用來避免同一個頭文件的多次包含。關於 `#include` 貌似比較簡單，通過 `man` 找到某個函數的頭文件，複製進去，加上 `<>` 就好。這裡雖然只關心一些技巧，不過預處理還是隱藏著很多潛在的陷阱（可參考《C Traps & Pitfalls》）也是需要注意的。下面僅介紹和預處理相關的幾個簡單內容。

打印出預處理之後的結果

```
$ gcc -E hello.c
```

這樣就可以看到源代碼中的各種預處理命令是如何被解釋的，從而方便理解和查錯。

實際上 `gcc` 在這裡調用了 `cpp`（雖然通過 `gcc -v` 僅看到 `cc1`），`cpp` 即 The C Preprocessor，主要用來預處理宏定義、文件包含、條件編譯等。下面介紹它的一個比較重要的選項 `-D`。

在命令行定義宏

```
$ gcc -Dmacro hello.c
```

這個等同於在文件的開頭定義宏，即 `#define macro`，但是在命令行定義更靈活。例如，在源代碼中有這些語句。

```
#ifdef DEBUG
printf("this code is for debugging\n");
#endif
```

如果編譯時加上 `-DDEBUG` 選項，那麼編譯器就會把 `printf` 所在的行編譯進目標代碼，從而方便地跟蹤該位置的某些程序狀態。這樣 `-DDEBUG` 就可以當作一個調試開關，編譯時加上它就可以用來打印調試信息，發佈時則可以通過去掉該編譯選項把調試信息去掉。

編譯（翻譯）

簡述

編譯之前，C 語言編譯器會進行詞法分析、語法分析，接著會把源代碼翻譯成中間語言，即彙編語言。如

果想看到這個中間結果，可以用 `gcc -S`。需要提到的是，諸如 Shell 等解釋語言也會經歷一個詞法分析和語法分析的階段，不過之後並不會進行“翻譯”，而是“解釋”，邊解釋邊執行。

把源代碼翻譯成彙編語言，實際上是編譯的整個過程中的第一個階段，之後的階段和彙編語言的開發過程沒有什麼區別。這個階段涉及到對源代碼的詞法分析、語法檢查（通過 `-std` 指定遵循哪個標準），並根據優化（`-O`）要求進行翻譯成彙編語言的動作。

語法檢查

如果僅僅希望進行語法檢查，可以用 `gcc` 的 `-fsyntax-only` 選項；如果為了使代碼有比較好的可移植性，避免使用 `gcc` 的一些擴展特性，可以結合 `-std` 和 `-pedantic`（或者 `-pedantic-errors`）選項讓源代碼遵循某個 C 語言標準的語法。這裡演示一個簡單的例子：

```
$ cat hello.c
#include <stdio.h>
int main()
{
    printf("hello, world\n")
    return 0;
}
$ gcc -fsyntax-only hello.c
hello.c: In function 'main':
hello.c:5: error: expected ';' before 'return'
$ vim hello.c
$ cat hello.c
#include <stdio.h>
int main()
{
    printf("hello, world\n");
    int i;
    return 0;
}
$ gcc -std=c89 -pedantic-errors hello.c      #默認情況下，gcc是允許在程序中間聲明變量的，但是turboC
hello.c: In function 'main':
hello.c:5: error: ISO C90 forbids mixed declarations and code
```

語法錯誤是程序開發過程中難以避免的錯誤（人的大腦在很多情況下都容易開小差），不過編譯器往往能夠通過語法檢查快速發現這些錯誤，並準確地告知語法錯誤的大概位置。因此，作為開發人員，要做的事情不是“恐慌”（不知所措），而是認真閱讀編譯器的提示，根據平時積累的經驗（最好總結一份常見語法錯誤索引，很多資料都提供了常見語法錯誤列表，如《C Traps&Pitfalls》和編輯器提供的語法檢查功能（語法加亮、括號匹配提示等）快速定位語法出錯的位置並進行修改。

編譯器優化

語法檢查之後就是翻譯動作，`gcc` 提供了一個優化選項 `-O`，以便根據不同的運行平臺和用戶要求產生經過優化的彙編代碼。例如，

```
$ gcc -o hello hello.c      # 採用默認選項，不優化
$ gcc -O2 -o hello2 hello.c # 優化等次是2
```

```

$ gcc -Os -o hellos hello.c      # 優化目標代碼的大小
$ ls -S hello hello2 hellos      # 可以看到, hellos 比較小, hello2 比較大
hello2  hello  hellos
$ time ./hello
hello, world

real    0m0.001s
user    0m0.000s
sys     0m0.000s
$ time ./hello2      # 可能是代碼比較少的緣故, 執行效率看上去不是很明顯
hello, world

real    0m0.001s
user    0m0.000s
sys     0m0.000s

$ time ./hellos      # 雖然目標代碼小了, 但是執行效率慢了點
hello, world

real    0m0.002s
user    0m0.000s
sys     0m0.000s

```

根據上面的簡單演示, 可以看出 `gcc` 有很多不同的優化選項, 主要看用戶的需求了, 目標代碼的大小和效率之間貌似存在一個“糾纏”, 需要開發人員自己權衡。

生成彙編語言文件

下面通過 `-S` 選項來看看編譯出來的中間結果: 彙編語言, 還是以之前那個 `hello.c` 為例。

```

$ gcc -S hello.c # 默認輸出是hello.s, 可自己指定, 輸出到屏幕`-o -`, 輸出到其他文件`-o file`
$ cat hello.s
cat hello.s
        .file "hello.c"
        .section      .rodata
.LC0:
        .string "hello, world"
        .text
.globl main
        .type  main, @function
main:
        leal   4(%esp), %ecx
        andl   $-16, %esp
        pushl  -4(%ecx)
        pushl  %ebp
        movl   %esp, %ebp
        pushl  %ecx
        subl   $4, %esp
        movl   $.LC0, (%esp)
        call   puts
        movl   $0, %eax
        addl   $4, %esp
        popl   %ecx
        popl   %ebp
        leal   -4(%ecx), %esp
        ret

```

```
.size    main, .-main
.ident   "GCC: (GNU) 4.1.3 20070929 (prerelease) (Ubuntu 4.1.2-16ubuntu2)"
.section .note.GNU-stack,"",@progbits
```

不知道看出來沒？和課堂裡學的 intel 的彙編語法不太一樣，這裡用的是 AT&T 語法格式。如果想學習 Linux 下的彙編語言開發，下一節開始的所有章節基本上覆蓋了 Linux 下彙編語言開發的一般過程，不過這裡不介紹彙編語言語法。

在學習後面的章節之前，建議自學舊金山大學的微機編程課程 CS630，該課深入介紹了 Linux/X86 平臺下的 AT&T 彙編語言開發。如果想在 Qemu 上做這個課程裡的實驗，可以閱讀本文作者寫的 [CS630: Linux 下通過 Qemu 學習 X86 AT&T 彙編語言](#)。

需要補充的是，在寫 C 語言代碼時，如果能夠對編譯器比較熟悉（工作原理和一些細節）的話，可能會很有幫助。包括這裡的優化選項（有些優化選項可能在彙編時採用）和可能的優化措施，例如字節對齊、條件分支語句裁減（刪除一些明顯分支）等。

彙編

簡述

彙編實際上還是翻譯過程，只不過把作為中間結果的彙編代碼翻譯成了機器代碼，即目標代碼，不過它還不可以運行。如果要產生這一中間結果，可用 `gcc -c`，當然，也可通過 `as` 命令處理彙編語言源文件來產生。

彙編是把彙編語言翻譯成目標代碼的過程，如果有在 Windows 下學習過彙編語言開發，大家應該比較熟悉 `nasm` 彙編工具(支持 Intel 格式的彙編語言)，不過這裡主要用 `as` 彙編工具來彙編 AT&T 格式的彙編語言，因為 `gcc` 產生的中間代碼就是 AT&T 格式的。

生成目標代碼

下面來演示分別通過 `gcc -c` 選項和 `as` 來產生目標代碼。

```
$ file hello.s
hello.s: ASCII assembler program text
$ gcc -c hello.s      #用gcc把彙編語言編譯成目標代碼
$ file hello.o        #file命令用來查看文件類型，目標代碼可重定位的(relocatable),
                      #需要通過ld進行進一步鏈接成可執行程序(executable)和共享庫(shared)
hello.o: ELF 32-bit LSB relocatable, Intel 80386, version 1 (SYSV), not stripped
$ as -o hello.o hello.s      #用as把彙編語言編譯成目標代碼
$ file hello.o
hello.o: ELF 32-bit LSB relocatable, Intel 80386, version 1 (SYSV), not stripped
```

`gcc` 和 `as` 默認產生的目標代碼都是 ELF 格式的，因此這裡主要討論 ELF 格式的目標代碼（如果有時間再回顧一下 `a.out` 和 `coff` 格式，當然也可以先了解一下，並結合 `objcopy` 來轉換它們，比較異同）。

ELF 文件初次接觸

目標代碼不再是普通的文本格式，無法直接通過文本編輯器瀏覽，需要一些專門的工具。如果想要了解更多目標代碼的細節，區分 `relocatable`（可重定位）、`executable`（可執行）、`shared library`（共享庫）的不同，我們得設法瞭解目標代碼的組織方式和相關的閱讀和分析工具。下面主要介紹這部分內容。

BFD is a package which allows applications to use the same routines to operate on object files whatever the object file format. A new object file format can be supported simply by creating a new BFD back end and adding it to the library.

`binutils`（GNU Binary Utilities）的很多工具都採用這個庫來操作目標文件，這類工具有 `objdump`，`objcopy`，`nm`，`strip` 等（當然，我們也可以利用它。如果深入瞭解ELF格式，那麼通過它來分析和編寫 Virus 程序將會更加方便），不過另外一款非常優秀的分析工具 `readelf` 並不是基於這個庫，所以也應該可以直接用 `elf.h` 頭文件中定義的相關結構來操作 ELF 文件。

下面將通過這些輔助工具（主要是 `readelf` 和 `objdump`），結合 ELF 手冊來分析它們。將依次介紹 ELF 文件的結構和三種不同類型 ELF 文件的區別。

ELF 文件的結構

```
ELF Header(ELF文件頭)
Program Headers Table(程序頭表，實際上叫段表好一些，用於描述可執行文件和可共享庫)
Section 1
Section 2
Section 3
...
Section Headers Table(節區頭部表，用於鏈接可重定位文件或可執行文件或共享庫)
```

對於可重定位文件，程序頭是可選的，而對於可執行文件和共享庫文件（動態鏈接庫），節區表則是可選的。可以分別通過 `readelf` 文件的 `-h`，`-l` 和 `-S` 參數查看 ELF 文件頭（ELF Header）、程序頭部表（Program Headers Table，段表）和節區表（Section Headers Table）。

文件頭說明了文件的類型，大小，運行平臺，節區數目等。

三種不同類型 ELF 文件比較

先來通過文件頭看看不同ELF的類型。為了說明問題，先來幾段代碼吧。

```
/* myprintf.c */
#include <stdio.h>

void myprintf(void)
{
    printf("hello, world!\n");
}
```

```
/* test.h -- myprintf function declaration */

#ifndef _TEST_H_
#define _TEST_H_
```

```
void myprintf(void);

#endif
```

```
/* test.c */
#include "test.h"

int main()
{
    myprintf();
    return 0;
}
```

下面通過這幾段代碼來演示通過 `readelf -h` 參數查看 ELF 的不同類型。期間將演示如何創建動態鏈接庫（即可共享文件）、靜態鏈接庫，並比較它們的異同。

編譯產生兩個目標文件 `myprintf.o` 和 `test.o`，它們都是可重定位文件（REL）：

```
$ gcc -c myprintf.c test.c
$ readelf -h test.o | grep Type
Type:                                REL (Relocatable file)
$ readelf -h myprintf.o | grep Type
Type:                                REL (Relocatable file)
```

根據目標代碼鏈接產生可執行文件，這裡的文件類型是可執行的(EXEC)：

```
$ gcc -o test myprintf.o test.o
$ readelf -h test | grep Type
Type:                                EXEC (Executable file)
```

用 `ar` 命令創建一個靜態鏈接庫，靜態鏈接庫也是可重定位文件（REL）：

```
$ ar rcsv libmyprintf.a myprintf.o
$ readelf -h libmyprintf.a | grep Type
Type:                                REL (Relocatable file)
```

可見，靜態鏈接庫和可重定位文件類型一樣，它們之間唯一不同是前者可以是多個可重定位文件的“集合”。

靜態鏈接庫可直接鏈接（只需庫名，不要前面的 `lib`），也可用 `-l` 參數，`-L` 指定庫搜索路徑。

```
$ gcc -o test test.o -lmyprintf -L./
```

編譯產生動態鏈接庫，並支持 `major` 和 `minor` 版本號，動態鏈接庫類型為 `DYN`：

```
$ gcc -Wall myprintf.o -shared -Wl,-soname,libmyprintf.so.0 -o libmyprintf.so.0.0
```

```
$ ln -sf libmyprintf.so.0.0 libmyprintf.so.0
$ ln -sf libmyprintf.so.0 libmyprintf.so
$ readelf -h libmyprintf.so | grep Type
Type:                                DYN (Shared object file)
```

動態鏈接庫編譯時和靜態鏈接庫類似：

```
$ gcc -o test test.o -lmyprintf -L./
```

但是執行時需要指定動態鏈接庫的搜索路徑，把 `LD_LIBRARY_PATH` 設為當前目錄，指定 `test` 運行時的動態鏈接庫搜索路徑：

```
$ LD_LIBRARY_PATH=./ ./test
$ gcc -static -o test test.o -lmyprintf -L./
```

在不指定 `-static` 時會優先使用動態鏈接庫，指定時則阻止使用動態鏈接庫，這時會把所有靜態鏈接庫文件加入到可執行文件中，使得執行文件很大，而且加載到內存以後會浪費內存空間，因此不建議這麼做。

經過上面的演示基本可以看出它們之間的不同：

- 可重定位文件本身不可以運行，僅僅是作為可執行文件、靜態鏈接庫（也是可重定位文件）、動態鏈接庫的“組件”。
- 靜態鏈接庫和動態鏈接庫本身也不可以執行，作為可執行文件的“組件”，它們兩者也不同，前者也是可重定位文件（只不過可能是多個可重定位文件的集合），並且在鏈接時加入到可執行文件中去。
- 而動態鏈接庫在鏈接時，庫文件本身並沒有添加到可執行文件中，只是在可執行文件中加入了該庫的名字等信息，以便在可執行文件運行過程中引用庫中的函數時由動態鏈接器去查找相關函數的地址，並調用它們。

從這個意義上說，動態鏈接庫本身也具有可重定位的特徵，含有可重定位的信息。對於什麼是重定位？如何進行靜態符號和動態符號的重定位，我們將在鏈接部分和《[動態符號鏈接的細節](#)》一節介紹。

ELF 主體：節區

下面來看看 ELF 文件的主體內容：節區（Section）。

ELF 文件具有很大的靈活性，它通過文件頭組織整個文件的總體結構，通過節區表（Section Headers Table）和程序頭（Program Headers Table 或者叫段表）來分別描述可重定位文件和可執行文件。但不管是哪種類型，它們都需要它們的主體，即各種節區。

在可重定位文件中，節區表描述的就是各種節區本身；而在可執行文件中，程序頭描述的是由各個節區組成的段（Segment），以便程序運行時動態裝載器知道如何對它們進行內存映像，從而方便程序加載和運行。

下面先來看看一些常見的節區，而關於這些節區（Section）如何通過重定位構成不同的段（Segments），以及有哪些常規的段，我們將在鏈接部分進一步介紹。

可以通過 `readelf -S` 查看 ELF 的節區。（建議一邊操作一邊看文檔，以便加深對 ELF 文件結構的理

解) 先來看看可重定位文件的節區信息，通過節區表來查看：

默認編譯好 `myprintf.c`，將產生一個可重定位的文件 `myprintf.o`，這裡通過 `myprintf.o` 的節區表查看節區信息。

```
$ gcc -c myprintf.c
$ readelf -S myprintf.o
There are 11 section headers, starting at offset 0xc0:
```

Section Headers:

[Nr]	Name	Type	Addr	Off	Size	ES	Flg	Lk	Inf	Al
[0]		NULL	00000000	000000	000000	00		0	0	0
[1]	.text	PROGBITS	00000000	000034	000018	00	AX	0	0	4
[2]	.rel.text	REL	00000000	000334	000010	08		9	1	4
[3]	.data	PROGBITS	00000000	00004c	000000	00	WA	0	0	4
[4]	.bss	NOBITS	00000000	00004c	000000	00	WA	0	0	4
[5]	.rodata	PROGBITS	00000000	00004c	00000e	00	A	0	0	1
[6]	.comment	PROGBITS	00000000	00005a	000012	00		0	0	1
[7]	.note.GNU-stack	PROGBITS	00000000	00006c	000000	00		0	0	1
[8]	.shstrtab	STRTAB	00000000	00006c	000051	00		0	0	1
[9]	.symtab	SYMTAB	00000000	000278	0000a0	10		10	8	4
[10]	.strtab	STRTAB	00000000	000318	00001a	00		0	0	1

Key to Flags:

W (write), A (alloc), X (execute), M (merge), S (strings)
 I (info), L (link order), G (group), x (unknown)
 0 (extra OS processing required) o (OS specific), p (processor specific)

用 `objdump -d` 可看反編譯結果，用 `-j` 選項可指定需要查看的節區：

```
$ objdump -d -j .text myprintf.o
myprintf.o: file format elf32-i386

Disassembly of section .text:

00000000 <myprintf>:
 0: 55          push    %ebp
 1: 89 e5       mov     %esp,%ebp
 3: 83 ec 08    sub     $0x8,%esp
 6: 83 ec 0c    sub     $0xc,%esp
 9: 68 00 00 00 00    push    $0x0
 e: e8 fc ff ff    call   f <myprintf+0xf>
13: 83 c4 10    add     $0x10,%esp
16: c9         leave
17: c3         ret
```

用 `-r` 選項可以看到有關重定位的信息，這裡有兩部分需要重定位：

```
$ readelf -r myprintf.o

Relocation section '.rel.text' at offset 0x334 contains 2 entries:
Offset      Info    Type           Sym.Value    Sym. Name
0000000a    00000501 R_386_32       00000000     .rodata
0000000f    00000902 R_386_PC32     00000000     puts
```

`.rodata` 節區包含只讀數據，即我們要打印的 `hello, world!`

```
$ readelf -x .rodata myprintf.o

Hex dump of section '.rodata':
0x00000000 68656c6c 6f2c2077 6f726c64 2100      hello, world!.
```

沒有找到 `.data` 節區，它應該包含一些初始化的數據：

```
$ readelf -x .data myprintf.o

Section '.data' has no data to dump.
```

也沒有 `.bss` 節區，它應該包含一些未初始化的數據，程序默認初始為 0：

```
$ readelf -x .bss      myprintf.o

Section '.bss' has no data to dump.
```

`.comment` 是一些註釋，可以看到是 `Gcc` 的版本信息

```
$ readelf -x .comment myprintf.o

Hex dump of section '.comment':
0x00000000 00474343 3a202847 4e552920 342e312e .GCC: (GNU) 4.1.
0x00000010 3200                                2.
```

`.note.GNU-stack` 這個節區也沒有內容：

```
$ readelf -x .note.GNU-stack myprintf.o

Section '.note.GNU-stack' has no data to dump.
```

`.shstrtab` 包括所有節區的名字：

```
$ readelf -x .shstrtab myprintf.o

Hex dump of section '.shstrtab':
0x00000000 002e7379 6d746162 002e7374 72746162 ..symtab..strtab
0x00000010 002e7368 73747274 6162002e 72656c2e ..shstrtab..rel.
0x00000020 74657874 002e6461 7461002e 62737300 text..data..bss.
0x00000030 2e726f64 61746100 2e636f6d 6d656e74 .rodata..comment
0x00000040 002e6e6f 74652e47 4e552d73 7461636b ..note.GNU-stack
0x00000050 00                                .
```

符號表 `.symtab` 包括所有用到的相關符號信息，如函數名、變量名，可用 `readelf` 查看：

```
$ readelf -symtab myprintf.o

Symbol table '.symtab' contains 10 entries:
   Num:      Value          Size Type      Bind     Vis      Ndx Name
   --:      -
   0: 00000000          0 NOTYPE   LOCAL   DEFAULT  UND
   1: 00000000          0 FILE    LOCAL   DEFAULT  ABS myprintf.c
   2: 00000000          0 SECTION LOCAL   DEFAULT    1
   3: 00000000          0 SECTION LOCAL   DEFAULT    3
   4: 00000000          0 SECTION LOCAL   DEFAULT    4
   5: 00000000          0 SECTION LOCAL   DEFAULT    5
   6: 00000000          0 SECTION LOCAL   DEFAULT    7
   7: 00000000          0 SECTION LOCAL   DEFAULT    6
   8: 00000000         24 FUNC     GLOBAL  DEFAULT    1 myprintf
   9: 00000000          0 NOTYPE   GLOBAL  DEFAULT  UND puts
```

字符串表 `.strtab` 包含用到的字符串，包括文件名、函數名、變量名等：

```
$ readelf -x .strtab myprintf.o

Hex dump of section '.strtab':
   0x00000000 006d7970 72696e74 662e6300 6d797072 .myprintf.c.mypr
   0x00000010 696e7466 00707574 7300      intf.puts.
```

從上表可以看出，對於可重定位文件，會包含這些基本節區 `.text`，`.rel.text`，`.data`，`.bss`，`.rodata`，`.comment`，`.note.GNU-stack`，`.shstrtab`，`.symtab` 和 `.strtab`。

彙編語言文件中的節區表述

為了進一步理解這些節區和源代碼的關係，這裡來看看 `myprintf.c` 產生的彙編代碼。

```
$ gcc -S myprintf.c
$ cat myprintf.s
        .file      "myprintf.c"
        .section   .rodata
.LC0:
        .string   "hello, world!"
        .text
.globl myprintf
        .type     myprintf, @function
myprintf:
        pushl    %ebp
        movl     %esp, %ebp
        subl     $8, %esp
        subl     $12, %esp
        pushl    $.LC0
        call     puts
        addl     $16, %esp
        leave
        ret
        .size    myprintf, .-myprintf
```

```
.ident "GCC: (GNU) 4.1.2"
.section .note.GNU-stack,"",@progbits
```

是不是可以從中看出可重定位文件中的那些節區和彙編語言代碼之間的關係？在上面的可重定位文件，可以看到有一個可重定位的節區，即 `.rel.text`，它標記了兩個需要重定位的項，`.rodata` 和 `puts`。這個節區將告訴編譯器這兩個信息在鏈接或者動態鏈接的過程中需要重定位，具體如何重定位？將根據重定位項的類型，比如上面的 `R_386_32` 和 `R_386_PC32`。

到這裡，對可重定位文件應該有了一個基本的瞭解，下面將介紹什麼是可重定位，可重定位文件到底是如何被鏈接生成可執行文件和動態鏈接庫的，這個過程除了進行一些符號的重定位外，還進行了哪些工作呢？

鏈接

簡述

重定位是將符號引用與符號定義進行鏈接的過程。因此鏈接是處理可重定位文件，把它們的各種符號引用和符號定義轉換為可執行文件中的合適信息（一般是虛擬內存地址）的過程。

鏈接又分為靜態鏈接和動態鏈接，前者是程序開發階段程序員用 `ld`（`gcc` 實際上在後臺調用了 `ld`）靜態鏈接器手動鏈接的過程，而動態鏈接則是程序運行期間系統調用動態鏈接器（`ld-linux.so`）自動鏈接的過程。

比如，如果鏈接到可執行文件中的是靜態鏈接庫 `libmprintf.a`，那麼 `.rodata` 節區在鏈接後需要被重定位到一個絕對的虛擬內存地址，以便程序運行時能夠正確訪問該節區中的字符串信息。而對於 `puts` 函數，因為它是動態鏈接庫 `libc.so` 中定義的函數，所以會在程序運行時通過動態符號鏈接找出 `puts` 函數在內存中的地址，以便程序調用該函數。在這裡主要討論靜態鏈接過程，動態鏈接過程見《[動態符號鏈接的細節](#)》。

靜態鏈接過程主要是把可重定位文件依次讀入，分析各個文件的文件頭，進而依次讀入各個文件的節區，並計算各個節區的虛擬內存位置，對一些需要重定位的符號進行處理，設定它們的虛擬內存地址等，並最終產生一個可執行文件或者是動態鏈接庫。這個鏈接過程是通過 `ld` 來完成的，`ld` 在鏈接時使用了一個鏈接腳本（`linker script`），該鏈接腳本處理鏈接的具體細節。

由於靜態符號鏈接過程非常複雜，特別是計算符號地址的過程，考慮到時間關係，相關細節請參考 ELF 手冊。這裡主要介紹可重定位文件中的節區（節區表描述的）和可執行文件中段（程序頭描述的）的對應關係以及 `gcc` 編譯時採用的一些默認鏈接選項。

可執行文件的段：節區重排

下面先來看看可執行文件的節區信息，通過程序頭（段表）來查看，為了比較，先把 `test.o` 的節區表也列出：

```
$ readelf -S test.o
There are 10 section headers, starting at offset 0xb4:

Section Headers:
```

[Nr]	Name	Type	Addr	Off	Size	ES	Flg	Lk	Inf	Al
[0]		NULL	00000000	000000	000000	00		0	0	0
[1]	.text	PROGBITS	00000000	000034	000024	00	AX	0	0	4
[2]	.rel.text	REL	00000000	0002ec	000008	08		8	1	4
[3]	.data	PROGBITS	00000000	000058	000000	00	WA	0	0	4
[4]	.bss	NOBITS	00000000	000058	000000	00	WA	0	0	4
[5]	.comment	PROGBITS	00000000	000058	000012	00		0	0	1
[6]	.note.GNU-stack	PROGBITS	00000000	00006a	000000	00		0	0	1
[7]	.shstrtab	STRTAB	00000000	00006a	000049	00		0	0	1
[8]	.symtab	SYMTAB	00000000	000244	000090	10		9	7	4
[9]	.strtab	STRTAB	00000000	0002d4	000016	00		0	0	1

Key to Flags:

W (write), A (alloc), X (execute), M (merge), S (strings)

I (info), L (link order), G (group), x (unknown)

0 (extra OS processing required) o (OS specific), p (processor specific)

\$ gcc -o test test.o myprintf.o

\$ readelf -l test

Elf file type is EXEC (Executable file)

Entry point 0x80482b0

There are 7 program headers, starting at offset 52

Program Headers:

Type	Offset	VirtAddr	PhysAddr	FileSiz	MemSiz	Flg	Align
PHDR	0x000034	0x08048034	0x08048034	0x000e0	0x000e0	R E	0x4
INTERP	0x000114	0x08048114	0x08048114	0x00013	0x00013	R	0x1
[Requesting program interpreter: /lib/ld-linux.so.2]							
LOAD	0x000000	0x08048000	0x08048000	0x0047c	0x0047c	R E	0x1000
LOAD	0x00047c	0x0804947c	0x0804947c	0x00104	0x00108	RW	0x1000
DYNAMIC	0x000490	0x08049490	0x08049490	0x000c8	0x000c8	RW	0x4
NOTE	0x000128	0x08048128	0x08048128	0x00020	0x00020	R	0x4
GNU_STACK	0x000000	0x00000000	0x00000000	0x00000	0x00000	RW	0x4

Section to Segment mapping:

Segment Sections...

00	
01	.interp
02	.interp .note.ABI-tag .hash .dynsym .dynstr .gnu.version .gnu.version_r .rel.dyn .rel.plt .init .plt .text .fini .rodata .eh_frame
03	.ctors .dtors .jcr .dynamic .got .got.plt .data .bss
04	.dynamic
05	.note.ABI-tag
06	

可發現，test 和 test.o，myprintf.o 相比，多了很多節區，如 .interp 和 .init 等。另外，上表也給出了可執行文件的如下幾個段（Segment）：

- PHDR：給出了程序表自身的大小和位置，不能出現一次以上。
- INTERP：因為程序中調用了 puts（在動態鏈接庫中定義），使用了動態鏈接庫，因此需要動態裝載器 / 鏈接器（ld-linux.so）
- LOAD：包括程序的指令，.text 等節區都映射在該段，只讀（R）
- LOAD：包括程序的數據，.data，.bss 等節區都映射在該段，可讀寫（RW）
- DYNAMIC：動態鏈接相關的信息，比如包含有引用的動態鏈接庫名字等信息
- NOTE：給出一些附加信息的位置和大小
- GNU_STACK：這裡為空，應該是和GNU相關的一些信息

這裡的段可能包括之前的一個或者多個節區，也就是說經過鏈接之後原來的節區被重排了，並映射到了不同的段，這些段將告訴系統應該如何把它加載到內存中。

鏈接背後的故事

從上表中，通過比較可執行文件 `test` 中擁有的節區和可重定位文件（`test.o` 和 `myprintf.o`）中擁有的節區後發現，鏈接之後多了一些之前沒有的節區，這些新的節區來自哪裡？它們的作用是什麼呢？先來通過 `gcc -v` 看看它的後臺鏈接過程。

把可重定位文件鏈接成可執行文件：

```
$ gcc -v -o test test.o myprintf.o
Reading specs from /usr/lib/gcc/i486-slackware-linux/4.1.2/specs
Target: i486-slackware-linux
Configured with: ../gcc-4.1.2/configure --prefix=/usr --enable-shared
--enable-languages=ada,c,c++,fortran,java,objc --enable-threads=posix
--enable-__cxa_atexit --disable-checking --with-gnu-ld --verbose
--with-arch=i486 --target=i486-slackware-linux --host=i486-slackware-linux
Thread model: posix
gcc version 4.1.2
/usr/libexec/gcc/i486-slackware-linux/4.1.2/collect2 --eh-frame-hdr -m
elf_i386 -dynamic-linker /lib/ld-linux.so.2 -o test
/usr/lib/gcc/i486-slackware-linux/4.1.2/../../../../crt1.o
/usr/lib/gcc/i486-slackware-linux/4.1.2/../../../../crti.o
/usr/lib/gcc/i486-slackware-linux/4.1.2/crtbegin.o
-L/usr/lib/gcc/i486-slackware-linux/4.1.2
-L/usr/lib/gcc/i486-slackware-linux/4.1.2
-L/usr/lib/gcc/i486-slackware-linux/4.1.2/../../../../i486-slackware-linux/lib
-L/usr/lib/gcc/i486-slackware-linux/4.1.2/../../../../ test.o myprintf.o -lgcc
--as-needed -lgcc_s --no-as-needed -lc -lgcc --as-needed -lgcc_s --no-as-needed
/usr/lib/gcc/i486-slackware-linux/4.1.2/crtend.o
/usr/lib/gcc/i486-slackware-linux/4.1.2/../../../../crtn.o
```

從上述演示看出，`gcc` 在鏈接了我們自己的目標文件 `test.o` 和 `myprintf.o` 之外，還鏈接了 `crt1.o`，`crtbegin.o` 等額外的目標文件，難道那些新的節區就來自這些文件？

用 ld 完成鏈接過程

另外 `gcc` 在進行了相關配置（`./configure`）後，調用了 `collect2`，卻並沒有調用 `ld`，通過查找 `gcc` 文檔中和 `collect2` 相關的部分發現 `collect2` 在後臺實際上還是去尋找 `ld` 命令的。為了理解 `gcc` 默認鏈接的後臺細節，這裡直接把 `collect2` 替換成 `ld`，並把一些路徑換成絕對路徑或者簡化，得到如下的 `ld` 命令以及執行的效果。

```
$ ld --eh-frame-hdr \
-m elf_i386 \
-dynamic-linker /lib/ld-linux.so.2 \
-o test \
/usr/lib/crt1.o /usr/lib/crti.o /usr/lib/gcc/i486-slackware-linux/4.1.2/crtbegin.o \
test.o myprintf.o \
-L/usr/lib/gcc/i486-slackware-linux/4.1.2 -L/usr/i486-slackware-linux/lib -L/usr/lib/ \
-lgcc --as-needed -lgcc_s --no-as-needed -lc -lgcc --as-needed -lgcc_s --no-as-needed \
```

```
/usr/lib/gcc/i486-slackware-linux/4.1.2/crtend.o /usr/lib/crtn.o
$ ./test
hello, world!
```

不出所料，它完美地運行了。下面通過 `ld` 的手冊（`man ld`）來分析一下這幾個參數：

- `--eh-frame-hdr`

要求創建一個 `.eh_frame_hdr` 節區(貌似目標文件test中並沒有這個節區，所以不關心它)。

- `-m elf_i386`

這裡指定不同平臺上的鏈接腳本，可以通過 `--verbose` 命令查看腳本的具體內容，如 `ld -m elf_i386 --verbose`，它實際上被存放在一個文件中（`/usr/lib/ldscripts` 目錄下），我們可以去修改這個腳本，具體如何做？請參考 `ld` 的手冊。在後面我們將簡要提到鏈接腳本中是如何預定義變量的，以及這些預定義變量如何在我們的程序中使用。需要提到的是，如果不是交叉編譯，那麼無須指定該選項。

- `-dynamic-linker /lib/ld-linux.so.2`

指定動態裝載器/鏈接器，即程序中的 `INTERP` 段中的內容。動態裝載器/鏈接器負責鏈接有可共享庫的可執行文件的裝載和動態符號鏈接。

- `-o test`

指定輸出文件，即可執行文件的名字

- `/usr/lib/crt1.o /usr/lib/crti.o /usr/lib/gcc/i486-slackware-linux/4.1.2/crtbegin.o`

鏈接到 `test` 文件開頭的一些內容，這裡實際上就包含了 `.init` 等節區。`.init` 節區包含一些可執行代碼，在 `main` 函數之前被調用，以便進行一些初始化操作，在 C++ 中完成構造函數功能。

- `test.o myprintf.o`

鏈接我們自己的可重定位文件

- `-L/usr/lib/gcc/i486-slackware-linux/4.1.2 -L/usr/i486-slackware-linux/lib -L/usr/lib/ \ -lgcc --as-needed -lgcc_s --no-as-needed -lc -lgcc --as-needed -lgcc_s --no-as-needed`

鏈接 `libgcc` 庫和 `libc` 庫，後者定義有我們需要的 `puts` 函數

- `/usr/lib/gcc/i486-slackware-linux/4.1.2/crtend.o /usr/lib/crtn.o`

鏈接到 `test` 文件末尾的一些內容，這裡實際上包含了 `.fini` 等節區。`.fini` 節區包含了一些可執行代碼，在程序退出時被執行，作一些清理工作，在 C++ 中完成析構造函數功能。我們往往可以通過 `atexit` 來註冊那些需要在程序退出時才執行的函數。

C++構造與析構：crtbegin.o和crtend.o

對於 `crtbegin.o` 和 `crtend.o` 這兩個文件，貌似完全是用來支持 C++ 的構造和析構工作的，所以可以

不鏈接到我們的可執行文件中，鏈接時把它們去掉看看，

```
$ ld -m elf_i386 -dynamic-linker /lib/ld-linux.so.2 -o test \
  /usr/lib/crti.o /usr/lib/crti.o test.o myprintf.o \
  -L/usr/lib -lc /usr/lib/crtn.o      #後面發現不用鏈接libgcc，也不用--eh-frame-hdr參數
$ readelf -l test

Elf file type is EXEC (Executable file)
Entry point 0x80482b0
There are 7 program headers, starting at offset 52

Program Headers:
  Type           Offset       VirtAddr    PhysAddr    FileSiz MemSiz  Flg Align
  PHDR           0x000034    0x08048034  0x08048034  0x000e0 0x000e0  R E  0x4
  INTERP         0x000114    0x08048114  0x08048114  0x00013 0x00013  R   0x1
      [Requesting program interpreter: /lib/ld-linux.so.2]
  LOAD           0x000000    0x08048000  0x08048000  0x003ea 0x003ea  R E  0x1000
  LOAD           0x0003ec    0x080493ec  0x080493ec  0x000e8 0x000e8  RW  0x1000
  DYNAMIC        0x0003ec    0x080493ec  0x080493ec  0x000c8 0x000c8  RW  0x4
  NOTE           0x000128    0x08048128  0x08048128  0x00020 0x00020  R   0x4
  GNU_STACK      0x000000    0x00000000  0x00000000  0x00000 0x00000  RW  0x4

Section to Segment mapping:
Segment Sections...
00
01      .interp
02      .interp .note.ABI-tag .hash .dynsym .dynstr .gnu.version .gnu.version_r
      .rel.dyn .rel.plt .init .plt .text .fini .rodata
03      .dynamic .got .got.plt .data
04      .dynamic
05      .note.ABI-tag
06
$ ./test
hello, world!
```

完全可以工作，而且發現 `.ctors`（保存著程序中全局構造函數的指針數組），`.dtors`（保存著程序中全局析構函數的指針數組），`.jcr`（未知），`.eh_frame` 節區都沒有了，所以 `crtbegin.o` 和 `crtend.o` 應該包含了這些節區。

初始化與退出清理：`crti.o` 和 `crtn.o`

而對於另外兩個文件 `crti.o` 和 `crtn.o`，通過 `readelf -S` 查看後發現它們都有 `.init` 和 `.fini` 節區，如果我們不需要讓程序進行一些初始化和清理工作呢？是不是就可以不鏈接這兩個文件？試試看。

```
$ ld -m elf_i386 -dynamic-linker /lib/ld-linux.so.2 -o test \
  /usr/lib/crti.o test.o myprintf.o -L/usr/lib/ -lc
/usr/lib/libc_nonshared.a(elf-init.oS): In function `__libc_csu_init':
(.text+0x25): undefined reference to `__init'
```

貌似不行，竟然有人調用了 `__libc_csu_init` 函數，而這個函數引用了 `__init`。這兩個符號都在哪裡呢？


```
$ readelf -s /usr/lib/crt1.o | grep __libc_csu_init
18: 00000000 0 NOTYPE GLOBAL DEFAULT UND __libc_csu_init
$ readelf -s /usr/lib/crti.o | grep _init
17: 00000000 0 FUNC GLOBAL DEFAULT 5 _init
```

竟然是 `crt1.o` 調用了 `__libc_csu_init` 函數，而該函數卻引用了我們沒有鏈接的 `crti.o` 文件中定義的 `_init` 符號。這樣的話不鏈接 `crti.o` 和 `crt0.o` 文件就不成了羅？不對吧，要不乾脆不用 `crt1.o` 算了，看看 `gcc` 額外鏈接進去的最後一個文件 `crt1.o` 到底幹了個啥子？

```
$ ld -m elf_i386 -dynamic-linker /lib/ld-linux.so.2 -o \
    test test.o myprintf.o -L/usr/lib/ -lc
ld: warning: cannot find entry symbol _start; defaulting to 000000000080481a4
```

這樣卻說沒有找到入口符號 `_start`，難道 `crt1.o` 中定義了這個符號？不過它給默認設置了一個地址，只是個警告，說明 `test` 已經生成，不管怎樣先運行看看再說。

```
$ ./test
hello, world!
Segmentation fault
```

貌似程序運行完了，不過結束時冒出個段錯誤？可能是程序結束時有問題，用 `gdb` 調試看看：

```
$ gcc -g -c test.c myprintf.c #產生目標代碼，非交叉編譯，不指定-m也可鏈接，所以下面可去掉-m
$ ld -dynamic-linker /lib/ld-linux.so.2 -o test \
    test.o myprintf.o -L/usr/lib -lc
ld: warning: cannot find entry symbol _start; defaulting to 000000000080481d8
$ ./test
hello, world!
Segmentation fault
$ gdb ./test
...
(gdb) l
1      #include "test.h"
2
3      int main()
4      {
5          myprintf();
6          return 0;
7      }
(gdb) break 7      #在程序的末尾設置一個斷點
Breakpoint 1 at 0x80481bf: file test.c, line 7.
(gdb) r            #程序都快結束了都沒問題，怎麼會到最後出個問題呢？
Starting program: /mnt/hda8/Temp/c/program/test
hello, world!

Breakpoint 1, main () at test.c:7
7      }
(gdb) n            #單步執行看看，怎麼下面一條指令是0x00000001，肯定是程序退出以後出了問題
0x00000001 in ?? ()
(gdb) n            #誒，當然找不到邊了，都跑到0x00000001了
Cannot find bounds of current function
(gdb) c
```

```
Continuing.
```

```
Program received signal SIGSEGV, Segmentation fault.
0x00000001 in ?? ()
```

原來是這麼回事，估計是 `return 0` 返回之後出問題了，看看它的彙編去。

```
$ gcc -S test.c #產生彙編代碼
$ cat test.s
...
    call    myprintf
    movl    $0, %eax
    addl    $4, %esp
    popl    %ecx
    popl    %ebp
    leal    -4(%ecx), %esp
    ret
...
```

後面就這麼幾條指令，難不成 `ret` 返回有問題，不讓它 `ret` 返回，把 `return` 改成 `_exit` 直接進入內核退出。

```
$ vim test.c
$ cat test.c    #就把return語句修改成_exit了。
#include "test.h"
#include <unistd.h> /* _exit */

int main()
{
    myprintf();
    _exit(0);
}
$ gcc -g -c test.c myprintf.c
$ ld -dynamic-linker /lib/ld-linux.so.2 -o test test.o myprintf.o -L/usr/lib -lc
ld: warning: cannot find entry symbol _start; defaulting to 000000000080481d8
$ ./test    #竟然好了，再看看彙編有什麼不同
hello, world!
$ gcc -S test.c
$ cat test.s    #貌似就把ret指令替換成了_exit函數調用，直接進入內核，讓內核處理了，那為什麼ret有問題呢
...
    call    myprintf
    subl    $12, %esp
    pushl    $0
    call    _exit
...
$ gdb ./test    #把代碼改回去（改成return 0;），再調試看看調用main函數返回時的下一條指令地址eip
...
(gdb) l
warning: Source file is more recent than executable.
1      #include "test.h"
2
3      int main()
4      {
5          myprintf();
6          return 0;
```

```

7      }
(gdb) break 5
Breakpoint 1 at 0x80481b5: file test.c, line 5.
(gdb) break 7
Breakpoint 2 at 0x80481bc: file test.c, line 7.
(gdb) r
Starting program: /mnt/hda8/Temp/c/program/test

Breakpoint 1, main () at test.c:5
5      myprintf();
(gdb) x/8x $esp
0xbf929510:    0xbf92953c    0x080481a4    0x00000000    0xb7eea84f
0xbf929520:    0xbf92953c    0xbf929534    0x00000000    0x00000001

```

發現 `0x00000001` 剛好是之前調試時看到的程序返回後的位置，即 `eip`，說明程序在初始化時，這個 `eip` 就是錯誤的。為什麼呢？因為根本沒有鏈接進初始化的代碼，而是在編譯器自己給我們，初始化了程序入口即 `00000000080481d8`，也就是說，沒有人調用 `main`，`main` 不知道返回哪裡去，所以，我們直接讓 `main` 結束時進入內核調用 `_exit` 而退出則不會有問題。

通過上面的演示和解釋發現只要把 `return` 語句修改為 `_exit` 語句，程序即使不鏈接任何額外的目標代碼都可以正常運行（原因是不鏈接那些額外的文件時相當於沒有進行初始化操作，如果在程序的最後執行 `ret` 彙編指令，程序將無法獲得正確的 `eip`，從而無法進行後續的動作）。但是為什麼會有“找不到 `_start` 符號”的警告呢？通過 `readelf -s` 查看 `crt1.o` 發現裡頭有這個符號，並且 `crt1.o` 引用了 `main` 這個符號，是不是意味著會從 `_start` 進入 `main` 呢？是不是程序入口是 `_start`，而並非 `main` 呢？

C 語言程序真正的入口

先來看看剛才提到的鏈接器的默認鏈接腳本（`ld -m elf_386 --verbose`），它告訴我們程序的入口（`entry`）是 `_start`，而一個可執行文件必須有一個入口地址才能運行，所以這就是說明了為什麼 `ld` 一定要提示我們“`_start` 找不到”，找不到以後就給默認設置了一個地址。

```
$ ld --verbose | grep ^ENTRY    #非交叉編譯，可不用-m參數；ld默認找_start入口，並不是main哦！
ENTRY(_start)
```

原來是這樣，程序的入口（`entry`）竟然不是 `main` 函數，而是 `_start`。那乾脆把彙編裡頭的 `main` 給改掉算了，看行不行？

先生成彙編 `test.s`：

```

$ cat test.c
#include "test.h"
#include <unistd.h>    /* _exit */

int main()
{
    myprintf();
    _exit(0);
}
$ gcc -S test.c

```

然後把彙編中的 `main` 改為 `_start`，即改程序入口為 `_start`：

```
$ sed -i -e "s#main#_start#g" test.s
$ gcc -c test.s myprintf.c
```

重新鏈接，發現果然沒問題了：

```
$ ld -dynamic-linker /lib/ld-linux.so.2 -o test test.o myprintf.o -L/usr/lib/ -lc
$ ./test
hello, world!
```

`_start` 竟然是真正的程序入口，那在有 `main` 的情況下呢？為什麼在 `_start` 之後能夠找到 `main` 呢？這個看看 alert7 大叔的[Before main 分析](#)吧，這裡不再深入介紹。

總之呢，通過修改程序的 `return` 語句為 `_exit(0)` 和修改程序的入口為 `_start`，我們的代碼不鏈接 `gcc` 默認鏈接的那些額外的文件同樣可以工作得很好。並且打破了一個學習 C 語言以來的常識：`main` 函數作為程序的主函數，是程序的入口，實際上則不然。

鏈接腳本初次接觸

再補充一點內容，在 `ld` 的鏈接腳本中，有一個特別的關鍵字 `PROVIDE`，由這個關鍵字定義的符號是 `ld` 的預定義字符，我們可以在 C 語言函數中擴展它們後直接使用。這些特別的符號可以通過下面的方法獲取，

```
$ ld --verbose | grep PROVIDE | grep -v HIDDEN
PROVIDE (__executable_start = 0x08048000); . = 0x08048000 + SIZEOF_HEADERS;
PROVIDE (__etext = .);
PROVIDE (_etext = .);
PROVIDE (etext = .);
_edata = .; PROVIDE (edata = .);
_end = .; PROVIDE (end = .);
```

這裡面有幾個我們比較關心的，第一個是程序的入口地址 `__executable_start`，另外三個是 `etext`，`edata`，`end`，分別對應程序的代碼段（text）、初始化數據（data）和未初始化的數據（bss）（可參考 `man etext`），如何引用這些變量呢？看看這個例子。

```
/* predefinevalue.c */
#include <stdio.h>

extern int __executable_start, etext, edata, end;

int main(void)
{
    printf ("program entry: 0x%x \n", &__executable_start);
    printf ("etext address(text segment): 0x%x \n", &etext);
    printf ("edata address(initialized data): 0x%x \n", &edata);
    printf ("end address(uninitilized data): 0x%x \n", &end);
}
```

```
    return 0;
}
```

到這裡，程序鏈接過程的一些細節都介紹得差不多了。在《[動態符號鏈接的細節](#)》中將主要介紹 ELF 文件的動態符號鏈接過程。

參考資料

- [Linux 彙編語言開發指南](#)
- [PowerPC 彙編](#)
- [用於 Power 體系結構的彙編語言](#)
- [Linux 中 x86 的內聯彙編](#)
- [Linux Assembly HOWTO](#)
- [Linux Assembly Language Programming](#)
- [Guide to Assembly Language Programming in Linux](#)
- [An beginners guide to compiling programs under Linux](#)
- [gcc manual](#)
- [A Quick Tour of Compiling, Linking, Loading, and Handling Libraries on Unix](#)
- [Unix 目標文件初探](#)
- [Before main\(\)分析](#)
- [A Process Viewing Its Own /proc//map Information](#)
- [UNIX 環境高級編程](#)
- [Linux Kernel Primer](#)
- [Understanding ELF using readelf and objdump](#)
- [Study of ELF loading and relocs](#)
- [ELF file format and ABI](#)
 - [\[1\]](#)
 - [\[2\]](#)
- [TN05.ELF.Format.Summary.pdf](#)
- [ELF文件格式\(中文\)](#)
- 關於 Gcc 方面的論文，請查看歷年的會議論文集
 - [2005](#)
 - [2006](#)
- [The Linux GCC HOW TO](#)
- [ELF: From The Programmer's Perspective](#)
- [C/C++ 程序編譯步驟詳解](#)
- [C 語言常見問題集](#)
- [使用 BFD 操作 ELF](#)
- [bfd document](#)
- [UNIX/LINUX 平臺可執行文件格式分析](#)
- [Linux 彙編語言快速上手：4大架構一塊學](#)
- [GNU binutils 小結](#)

程序執行的一剎那

- 什麼是命令行接口
- `/bin/bash` 是什麼時候啟動的
 - `/bin/login`
 - `/bin/getty`
 - `/sbin/init`
 - 命令啟動過程追本溯源
 - 誰啟動了 `/sbin/init`
- `/bin/bash` 如何處理用戶鍵入的命令
 - 預備知識
 - 哪種命令先被執行
 - 這些特殊字符是如何解析的： `|`, `>`, `<`, `&`
 - `/bin/bash` 用什麼魔法讓一個普通程序變成了進程
- 參考資料

當我們在 Linux 下的命令行輸入一個命令之後，這背後發生了什麼？

什麼是命令行接口

用戶使用計算機有兩種常見的方式，一種是圖形化的接口（GUI），另外一種則是命令行接口（CLI）。對於圖形化的接口，用戶點擊某個圖標就可啟動後臺的某個程序；對於命令行的接口，用戶鍵入某個程序的名字就可啟動某個程序。這兩者的基本過程是類似的，都需要查找程序文件在磁盤上的位置，加載到內存並通過不同的解釋器進行解析和運行。下面以命令行為例來介紹程序執行一剎那發生的一些事情。

首先來介紹什麼是命令行？命令行就是 `Command Line`，很直觀的概念就是系統啟動後的那個黑屏幕：有一個提示符，並有光標在閃爍的那樣一個終端，一般情況下可以用 `CTRL+ALT+F1-6` 切換到不同的終端；在 GUI 界面下也會有一些偽終端，看上去和系統啟動時的那個終端沒有什麼區別，也會有一個提示符，並有一個光標在閃爍。就提示符和響應用戶的鍵盤輸入而言，它們兩者在功能上是一樣的，實際上它們就是同一個東西，用下面的命令就可以把它們打印出來。

```
$ echo $SHELL # 打印當前SHELL，當前運行的命令行接口程序
/bin/bash
$ echo $$      # 該程序對應進程ID，$$是個特殊的環境變量，它存放了當前進程ID
1481
$ ps -C bash   # 通過PS命令查看
  PID TTY          TIME CMD
 1481 pts/0      00:00:00 bash
```

從上面的操作結果可以看出，當前命令行接口實際上是一個程序，那就是 `/bin/bash`，它是一個實實在在的程序，它打印提示符，接受用戶輸入的命令，分析命令序列並執行然後返回結果。不過 `/bin/bash` 僅僅是當前使用的命令程序之一，還有很多具有類似功能的程序，比如 `/bin/ash`，`/bin/dash` 等。不過這裡主要來討論 `bash`，討論它自己是怎麼啟動的，它怎麼樣處理用戶輸入的命令等後臺細節？

`/bin/bash` 是什麼時候啟動的

/bin/login

先通過 `CTRL+ALT+F1` 切換到一個普通終端下面，一般情況下看到的是 "XXX login: " 提示輸入用戶名，接著是提示輸入密碼，然後呢？就直接登錄到了我們的命令行接口。實際上正是你輸入正確的密碼後，那個程序才把 `/bin/bash` 給啟動了。那是什麼東西提示 "XXX login:" 的呢？正是 `/bin/login` 程序，那 `/bin/login` 程序怎麼知道要啟動 `/bin/bash`，而不是其他的 `/bin/dash` 呢？

`/bin/login` 程序實際上會檢查我們的 `/etc/passwd` 文件，在這個文件裡頭包含了用戶名、密碼和該用戶的登錄 Shell。密碼和用戶名匹配用戶的登錄，而登錄 Shell 則作為用戶登錄後的命令程序。看看 `/etc/passwd` 中典型的這麼一行：

```
$ cat /etc/passwd | grep falcon
falcon:x:1000:1000:falcon,,,:/home/falcon:/bin/bash
```

這個是我用的帳號的相關信息哦，看到最後一行沒？`/bin/bash`，這正是我登錄用的命令行解釋程序。至於密碼呢，看到那個 `x` 沒？這個 `x` 說明我的密碼被保存在另外一個文件裡頭 `/etc/shadow`，而且密碼是經過加密的。至於這兩個文件的更多細節，看手冊吧。

我們怎麼知道剛好是 `/bin/login` 打印了 "XXX login" 呢？現在回顧一下很早以前學習的那個 `strace` 命令。我們可以用 `strace` 命令來跟蹤 `/bin/login` 程序的執行。

跟上面一樣，切換到一個普通終端，並切換到 Root 用戶，用下面的命令：

```
$ strace -f -o strace.out /bin/login
```

退出以後就可以打開 `strace.out` 文件，看看到底執行了哪些文件，讀取了哪些文件。從中可以看到正是 `/bin/login` 程序用 `execve` 調用了 `/bin/bash` 命令。通過後面的演示，可以發現 `/bin/login` 只是在子進程裡頭用 `execve` 調用了 `/bin/bash`，因為在啟動 `/bin/bash` 後，可以看到 `/bin/login` 並沒有退出。

/bin/getty

那 `/bin/login` 又是怎麼起來的呢？

下面再來看一個演示。先是一個可以登陸的終端下執行下面的命令。

```
$ getty 38400 tty8 linux
```

`getty` 命令停留在那裡，貌似等待用戶的什麼操作，現在切回到第 8 個終端，是不是看到有 "XXX login:" 的提示了。輸入用戶名並登錄，之後退出，回到第一個終端，發現 `getty` 命令已經退出。

類似地，也可以用 `strace` 命令來跟蹤 `getty` 的執行過程。在第一個終端下切換到 Root 用戶。執行如下命令：

```
$ strace -f -o strace.out getty 38400 tty8 linux
```

同樣在 `strace.out` 命令中可以找到該命令的相關啟動細節。比如，可以看到正是 `getty` 程序用 `execve` 系統調用執行了 `/bin/login` 程序。這個地方，`getty` 是在自己的主進程裡頭直接執行了 `/bin/login`，這樣 `/bin/login` 將把 `getty` 的進程空間替換掉。

/sbin/init

這裡涉及到一個非常重要的東西：`/sbin/init`，通過 `man init` 命令可以查看到該命令的作用，它可是“萬物之王”（`init is the parent of all processes on the system`）哦。它是 Linux 系統默認啟動的第一個程序，負責進行 Linux 系統的一些初始化工作，而這些初始化工作的配置則是通過 `/etc/inittab` 來做的。那麼來看看 `/etc/inittab` 的一個簡單的例子吧，可以通過 `man inittab` 查看相關幫助。

需要注意的是，在較新版本的 Ubuntu 和 Fedora 等發行版中，一些新的 `init` 程序，比如 `upstart` 和 `systemd` 被開發出來用於取代 `System V init`，它們可能放棄了對 `/etc/inittab` 的使用，例如 `upstart` 會讀取 `/etc/init/` 下的配置，比如 `/etc/init/tty1.conf`，但是，基本的配置思路還是類似 `/etc/inittab`，對於 `upstart` 的 `init` 配置，這裡不做介紹，請通過 `man 5 init` 查看幫助。

配置文件 `/etc/inittab` 的語法非常簡單，就是下面一行的重複，

```
id:runlevels:action:process
```

- `id` 就是一個唯一的編號，不用管它，一個名字而言，無關緊要。
- `runlevels` 是運行級別，這個還是比較重要的，理解運行級別的概念很有必要，它可以有如下的取值：

```
0 is halt.
1 is single-user.
2-5 are multi-user.
6 is reboot.
```

不過，真正在配置文件裡頭用的是 `1-5` 了，而 `0` 和 `6` 非常特別，除了用它作為 `init` 命令的參數關機和重啟外，似乎沒有哪個“傻瓜”把它寫在系統的配置文件中，讓系統啟動以後就關機或者重啟。`1` 代表單用戶，而 `2-5` 則代表多用戶。對於 `2-5` 可能有不同的解釋，比如在 Slackware 12.0 上，`2,3,5` 被用來作為多用戶模式，但是默認不啟動 X windows（GUI 接口），而 `4` 則作為啟動 X windows 的運行級別。

- `action` 是動作，它也有很多選擇，我們關心幾個常用的
- `initdefault`：用來指定系統啟動後進入的運行級別，通常在 `/etc/inittab` 的第一條配置，如：

```
id:3:initdefault:
```


這個說明默認運行級別是 3，即多用戶模式，但是不啟動 X window 的那種。

- `sysinit`：指定那些在系統啟動時將被執行的程序，例如：

```
si:S:sysinit:/etc/rc.d/rc.S
```

在 `man inittab` 中提到，對於 `sysinit`，`boot` 等動作，`runlevels` 選項是不用管的，所以可以很容易解讀這條配置：它的意思是系統啟動時將默認執行 `/etc/rc.d/rc.S` 文件，在這個文件裡可直接或者間接地執行想讓系統啟動時執行的任何程序，完成系統的初始化。

- `wait`：當進入某個特別的運行級別時，指定的程序將被執行一次，`init` 將等到它執行完成，例如：

```
rc:2345:wait:/etc/rc.d/rc.M
```

這個說明無論是進入運行級別 2, 3, 4, 5 中哪一個，`/etc/rc.d/rc.M` 將被執行一次，並且有 `init` 等待它執行完成。

- `ctrlaltdel`，當 `init` 程序接收到 `SIGINT` 信號時，某個指定的程序將被執行，我們通常通過按下 `CTRL+ALT+DEL`，這個默認情況下將給 `init` 發送一個 `SIGINT` 信號。

如果我們想在按下這幾個鍵時，系統重啟，那麼可以在 `/etc/inittab` 中寫入：

```
ca::ctrlaltdel:/sbin/shutdown -t5 -r now
```

- `respawn`：這個指定的進程將被重啟，任何時候當它退出時。這意味著沒有辦法結束它，除非 `init` 自己結束了。例如：

```
c1:1235:respawn:/sbin/agetty 38400 tty1 linux
```

這一行的意思非常簡單，就是系統運行在級別 1, 2, 3, 5 時，將默認執行 `/sbin/agetty` 程序（這個類似於上面提到的 `getty` 程序），這個程序非常有意思，就是無論什麼時候它退出，`init` 將再次啟動它。這個有幾個比較有意思的問題：

- 在 Slackware 12.0 下，當默認運行級別為 4 時，只有第 6 個終端可以用。原因是什麼呢？因為類似上面的配置，因為那裡只有 1235，而沒有 4，這意味著當系統運行在第 4 級別時，其他終端下的 `/sbin/agetty` 沒有啟動。所以，如果想讓其他終端都可以用，把 1235 修改為 12345 即可。
- 另外一個有趣的問題就是：正是 `init` 程序在讀取這個配置行以後啟動了 `/sbin/agetty`，這就是 `/sbin/agetty` 的祕密。
- 還有一個問題：無論退出哪個終端，那個 "XXX login:" 總是會被打印，原因是 `respawn` 動作有趣的性質，因為它告訴 `init`，無論 `/sbin/agetty` 什麼時候退出，重新把它啟動起來，那跟 "XXX login:" 有什麼關係呢？從前面的內容，我們發現正是 `/sbin/getty`（同 `agetty`）啟動了 `/bin/login`，而 `/bin/login` 又啟動了 `/bin/bash`，即我們的命令程序。

命令啟動過程追本溯源

而 `init` 程序作為“萬物之王”，它是所有進程的“父”（也可能是祖父……）進程，那意味著其他進程最多隻能是它的兒子進程。而這個子進程是怎麼創建的，`fork` 調用，而不是之前提到的 `execve` 調用。前者創建一個子進程，後者則會覆蓋當前進程。因為我們發現 `/sbin/getty` 運行時，`init` 並沒有退出，因此可以判斷是 `fork` 調用創建一個子進程後，才通過 `execve` 執行了 `/sbin/getty`。

因此，可以總結出這麼一個調用過程：

```

      fork      execve      execve      fork      execve
init --> init --> /sbin/getty --> /bin/login --> /bin/login --> /bin/bash

```

這裡的 `execve` 調用以後，後者將直接替換前者，因此當鍵入 `exit` 退出 `/bin/bash` 以後，也就相當於 `/sbin/getty` 都已經結束了，因此最前面的 `init` 程序判斷 `/sbin/getty` 退出了，又會創建一個子進程把 `/sbin/getty` 啟動，進而又啟動了 `/bin/login`，又看到了那個 "XXX login:"。

通過 `ps` 和 `pstree` 命令看看實際情況是不是這樣，前者打印出進程的信息，後者則打印出調用關係。

```

$ ps -ef | egrep "/sbin/init|/sbin/getty|bash|/bin/login"
root      1      0  0 21:43 ?          00:00:01 /sbin/init
root     3957      1  0 21:43 tty4      00:00:00 /sbin/getty 38400 tty4
root     3958      1  0 21:43 tty5      00:00:00 /sbin/getty 38400 tty5
root     3963      1  0 21:43 tty3      00:00:00 /sbin/getty 38400 tty3
root     3965      1  0 21:43 tty6      00:00:00 /sbin/getty 38400 tty6
root     7023      1  0 22:48 tty1      00:00:00 /sbin/getty 38400 tty1
root     7081      1  0 22:51 tty2      00:00:00 /bin/login --
falcon   7092   7081  0 22:52 tty2      00:00:00 -bash

```

上面的結果已經過濾了一些不相干的數據。從上面的結果可以看到，除了 `tty2` 被替換成 `/bin/login` 外，其他終端都運行著 `/sbin/getty`，說明終端 2 上的進程是 `/bin/login`，它已經把 `/sbin/getty` 替換掉，另外，我們看到 `-bash` 進程的父進程是 `7081` 剛好是 `/bin/login` 程序，這說明 `/bin/login` 啟動了 `-bash`，但是它並沒有替換掉 `/bin/login`，而是成為了 `/bin/login` 的子進程，這說明 `/bin/login` 通過 `fork` 創建了一個子進程並通過 `execve` 執行了 `-bash`（後者通過 `strace` 跟蹤到）。而 `init` 呢，其進程 ID 是 1，是 `/sbin/getty` 和 `/bin/login` 的父進程，說明 `init` 啟動或者間接啟動了它們。下面通過 `pstree` 來查看調用樹，可以更清晰地看出上述關係。

```

$ pstree | egrep "init|getty|\\-bash|login"
init+-5*[getty]
    |-login---bash
    |-xfce4-terminal+-bash+-grep

```

結果顯示 `init` 是 5 個 `getty` 程序，`login` 程序和 `xfce4-terminal` 的父進程，而後兩者則是 `bash` 的父進程，另外我們執行的 `grep` 命令則在 `bash` 上運行，是 `bash` 的子進程，這個將是我們後面關心的問題。

從上面的結果發現，`init` 作為所有進程的父進程，它的父進程 ID 饒有興趣的是 0，它是怎麼被啟動的

呢？誰才是真正的“造物主”？

誰啟動了 /sbin/init

如果用過 Lilo 或者 Grub 這些操作系統引導程序，可能會用到 Linux 內核的一個啟動參數 `init`，當忘記密碼時，可能會把這個參數設置成 `/bin/bash`，讓系統直接進入命令行，而無須輸入帳號和密碼，這樣就可以方便地把登錄密碼修改掉。

這個 `init` 參數是個什麼東西呢？通過 `man bootparam` 會發現它的祕密，`init` 參數正好指定了內核啟動後要啟動的第一個程序，而如果沒有指定該參數，內核將依次查找

`/sbin/init`，`/etc/init`，`/bin/init`，`/bin/sh`，如果找不到這幾個文件中的任何一個，內核就要恐慌（panic）了，並掛（hang）在那裡一動不動了（注：如果 `panic=timeout` 被傳遞給內核並且 `timeout` 大於 0，那麼就不會掛住而是重啟）。

因此 `/sbin/init` 就是 Linux 內核啟動的。而 Linux 內核呢？是通過 Lilo 或者 Grub 等引導程序啟動的，Lilo 和 Grub 都有相應的配置文件，一般對應 `/etc/lilo.conf` 和 `/boot/grub/menu.lst`，通過這些配置文件可以指定內核映像文件、系統根目錄所在分區、啟動選項標籤等信息，從而能夠讓它們順利把內核啟動起來。

那 Lilo 和 Grub 本身又是怎麼被運行起來的呢？有了解 MBR 不？MBR 就是主引導扇區，一般情況下這裡存放著 Lilo 和 Grub 的代碼，而誰知道正好是這裡存放了它們呢？BIOS，如果你用光盤安裝過操作系統的話，那麼應該修改過 BIOS 的默認啟動設置，通過設置可以讓系統從光盤、硬盤、U 盤甚至軟盤啟動。正是這裡的設置讓 BIOS 知道了 MBR 處的代碼需要被執行。

那 BIOS 又是什麼時候被起來的呢？處理器加電後有一個默認的起始地址，一上電就執行到了這裡，再之前就是開機鍵按鍵後的上電時序。

更多系統啟動的細節，看看 `man boot-scripts` 吧。

到這裡，`/bin/bash` 的神祕面紗就被揭開了，它只是系統啟動後運行的一個程序而已，只不過這個程序可以響應用戶的請求，那它到底是如何響應用戶請求的呢？

/bin/bash 如何處理用戶鍵入的命令

預備知識

在執行磁盤上某個程序時，通常不會指定這個程序文件的絕對路徑，比如要執行 `echo` 命令時，一般不會輸入 `/bin/echo`，而僅僅是輸入 `echo`。那為什麼這樣 `bash` 也能夠找到 `/bin/echo` 呢？原因是 Linux 操作系統支持這樣一種策略：Shell 的一個環境變量 `PATH` 裡頭存放了程序的一些路徑，當 Shell 執行程序時有可能去這些目錄下查找。`which` 作為 Shell（這裡特指 `bash`）的一個內置命令，如果用戶輸入的命令是磁盤上的某個程序，它會返回這個文件的全路徑。

有三個東西和終端的關係很大，那就是標準輸入、標準輸出和標準錯誤，它們是三個文件描述符，一般對應描述符 0，1，2。在 C 語言程序裡，我們可以把它們當作文件描述符一樣進行操作。在命令行下，則可以使用重定向字符 `>`，`<` 等對它們進行操作。對於標準輸出和標準錯誤，都默認輸出到終端，對於標準輸入，也同樣默認從終端輸入。

哪種命令先被執行

在 C 語言裡頭要寫一段輸入字符串的命令很簡單，調用 `scanf` 或者 `fgets` 就可以。這個在 `bash` 裡頭應該是類似的。但是它獲取用戶的命令以後，如何分析命令，如何響應不同的命令呢？

首先來看看 `bash` 下所謂的命令，用最常見的 `test` 來作測試。

- 字符串被解析成命令

隨便鍵入一個字符串 `test1`，`bash` 發出響應，告知找不到這個程序：

```
$ test1
bash: test1: command not found
```

- 內置命令

而當鍵入 `test` 時，看不到任何輸出，唯一響應是，新命令提示符被打印了：

```
$ test
$
```

查看 `test` 這個命令的類型，即查看 `test` 將被如何解釋，`type` 告訴我們 `test` 是一個內置命令，如果沒有理解錯，`test` 應該是利用諸如 `case "test": do something;break;` 這樣的機制實現的，具體如何實現可以查看 `bash` 源代碼。

```
$ type test
test is a shell builtin
```

- 外部命令

這裡通過 `which` 查到 `/usr/bin` 下有一個 `test` 命令文件，在鍵入 `test` 時，到底哪一個被執行了呢？

```
$ which test
/usr/bin/test
```

執行這個呢？也沒什麼反應，到底誰先被執行了？

```
$ /usr/bin/test
```

從上述演示中發現一個問題？如果輸入一個命令，這個命令要麼就不存在，要麼可能同時是 Shell 的內置命令、也有可能是磁盤上環境變量 `PATH` 所指定的目錄下的某個程序文件。

考慮到 `test` 內置命令和 `/usr/bin/test` 命令的響應結果一樣，我們無法知道哪一個先被執行了，

怎麼辦呢？把 `/usr/bin/test` 替換成一個我們自己的命令，並讓它打印一些信息(比如 `hello, world!`)，這樣我們就知道到底誰被執行了。寫完程序，編譯好，命名為 `test` 放到 `/usr/bin` 下（記得備份原來那個）。開始測試：

鍵入 `test`，還是沒有效果：

```
$ test
$
```

而鍵入絕對路徑呢，則打印了 `hello, world!` 誒，那默認情況下肯定是內置命令先被執行了：

```
$ /usr/bin/test
hello, world!
```

由上述實驗結果可見，內置命令比磁盤文件中的程序優先被 `bash` 執行。原因應該是內置命令避免了不必要的 `fork/execve` 調用，對於採用類似算法實現的功能，內置命令理論上有更高運行效率。

下面看看更多有趣的內容，鍵盤鍵入的命令還有可能是什麼呢？因為 `bash` 支持別名（`alias`）和函數（`function`），所以還有可能是別名和函數，另外，如果 `PATH` 環境變量指定的不同目錄下有相同名字的程序文件，那到底哪個被優先找到呢？

下面再作一些實驗，

- 別名

把 `test` 命名為 `ls -l` 的別名，再執行 `test`，竟然執行了 `ls -l`，說明別名（`alias`）比內置命令（`builtin`）更優先：

```
$ alias test="ls -l"
$ test
total 9488
drwxr-xr-x 12 falcon falcon    4096 2008-02-21 23:43 bash-3.2
-rw-r--r--  1 falcon falcon 2529838 2008-02-21 23:30 bash-3.2.tar.gz
```

- 函數

定義一個名叫 `test` 的函數，執行一下，發現，還是執行了 `ls -l`，說明 `function` 沒有 `alias` 優先級高：

```
$ function test { echo "hi, I'm a function"; }
$ test
total 9488
drwxr-xr-x 12 falcon falcon    4096 2008-02-21 23:43 bash-3.2
-rw-r--r--  1 falcon falcon 2529838 2008-02-21 23:30 bash-3.2.tar.gz
```

把別名給去掉（`unalias`），現在執行的是函數，說明函數的優先級比內置命令也要高：

```
$ unalias test
$ test
hi, I'm a function
```

如果在命令之前跟上 `builtin`，那麼將直接執行內置命令：

```
$ builtin test
```

要去掉某個函數的定義，這樣就可以：

```
$ unset test
```

通過這個實驗我們得到一個命令的別名（`alias`）、函數（`function`），內置命令（`builtin`）和程序（`program`）的執行優先次序：

```
先    alias --> function --> builtin --> program    後
```

實際上，`type` 命令會告訴我們這些細節，`type -a` 會按照 `bash` 解析的順序依次打印該命令的類型，而 `type -t` 則會給出第一個將被解析的命令的類型，之所以要做上面的實驗，是為了讓大家加印象。

```
$ type -a test
test is a shell builtin
test is /usr/bin/test
$ alias test="ls -l"
$ function test { echo "I'm a function"; }
$ type -a test
test is aliased to `ls -l'
test is a function
test ()
{
    echo "I'm a function"
}
test is a shell builtin
test is /usr/bin/test
$ type -t test
alias
```

下面再看看 `PATH` 指定的多個目錄下有同名程序的情況。再寫一個程序，打印 `hi, world!`，以示和 `hello, world!` 的區別，放到 `PATH` 指定的另外一個目錄 `/bin` 下，為了保證測試的說服力，再寫一個放到另外一個叫 `/usr/local/sbin` 的目錄下。

先看看 `PATH` 環境變量，確保它有 `/usr/bin`，`/bin` 和 `/usr/local/sbin` 這幾個目錄，然後通過 `type -P`（`-P` 參數強制到 `PATH` 下查找，而不管是別名還是內置命令等，可以通過 `help type` 查看該參數的含義）查看，到底哪個先被執行。

```
$ echo $PATH /usr/local/sbin:/usr/local/bin:/usr/sbin:/usr/bin:/sbin:/bin:/usr/games $ type -P test
```

程序執行的一刹那

```
/usr/local/sbin/test
```

如上可以看到 `/usr/local/sbin` 下的先被找到。

把 `/usr/local/sbin/test` 下的給刪除掉，現在 `/usr/bin` 下的先被找到：

```
$ rm /usr/local/sbin/test
$ type -P test
/usr/bin/test
```

`type -a` 也顯示類似的結果：

```
$ type -a test
test is aliased to `ls -l`
test is a function
test ()
{
    echo "I'm a function"
}
test is a shell builtin
test is /usr/bin/test
test is /bin/test
```

因此，可以找出這麼一個規律：Shell 從 `PATH` 列出的路徑中依次查找用戶輸入的命令。考慮到程序的優先級最低，如果想優先執行磁盤上的程序文件 `test` 呢？那麼就可以用 `test -P` 找出這個文件並執行就可以了。

補充：對於 Shell 的內置命令，可以通過 `help command` 的方式獲得幫助，對於程序文件，可以查看用戶手冊（當然，這個需要安裝，一般叫做 `xxx-doc`），`man command`。

這些特殊字符是如何解析的：|, >, <, &

在命令行上，除了輸入各種命令以及一些參數外，比如上面 `type` 命令的各種參數 `-a`，`-P` 等，對於這些參數，是傳遞給程序本身的，非常好處理，比如 `if`，`else` 條件分支或者 `switch`，`case` 都可以處理。當然，在 `bash` 裡頭可能使用專門的參數處理函數 `getopt` 和 `getopt_long` 來處理它們。

而 `|`，`>`，`<`，`&` 等字符，則比較特別，Shell 是怎麼處理它們的呢？它們也被傳遞給程序本身嗎？可我們的程序內部一般都不處理這些字符的，所以應該是 Shell 程序自己解析了它們。

先來看看這幾個字符在命令行的常見用法，

`<` 字符表示：把 `test.c` 文件重定向為標準輸入，作為 `cat` 命令輸入，而 `cat` 默認輸出到標準輸出：

```
$ cat < ./test.c
#include <stdio.h>

int main(void)
{
    printf("hi, myself!\n");
```

程序執行的一刹那

```
    return 0;
}
```

> 表示把標準輸出重定向為文件 `test_new.c`，結果內容輸出到 `test_new.c`：

```
$ cat < ./test.c > test_new.c
```

對於 `>`，`<`，`>>`，`<<`，`<>` 我們都稱之為重定向（`redirect`），Shell 到底是怎麼進行所謂的“重定向”的呢？

這主要歸功於 `dup/fcntl` 等函數，它們可以實現：複製文件描述符，讓多個文件描述符共享同一個文件表項。比如，當把文件 `test.c` 重定向為標準輸入時。假設之前用以打開 `test.c` 的文件描述符是 5，現在就把 5 複製為了 0，這樣當 `cat` 試圖從標準輸入讀出內容時，也就訪問了文件描述符 5 指向的文件表項，接著讀出了文件內容。輸出重定向與此類似。其他重定向，諸如 `>>`，`<<`，`<>` 等雖然和 `>`，`<` 的具體實現功能不太一樣，但本質是一樣的，都是文件描述符的複製，只不過可能對文件操作有一些附加的限制，比如 `>>` 在輸出時追加到文件末尾，而 `>` 則會從頭開始寫入文件，前者意味著文件的大小會增長，而後者則意味文件被重寫。

那麼 `|` 呢？`|` 被形象地稱為“管道”，實際上它就是通過 C 語言裡頭的無名管道來實現的。先看一個例子，

```
$ cat < ./test.c | grep hi
printf("hi, myself!\n");
```

在這個例子中，`cat` 讀出了 `test.c` 文件中的內容，並輸出到標準輸出上，但是實際上輸出的內容卻只有一行，原因是這個標準輸出被“接到”了 `grep` 命令的標準輸入上，而 `grep` 命令只打印了包含“hi”字符串的一行。

這是怎麼被“接”上的。`cat` 和 `grep` 作為兩個單獨的命令，它們本身沒有辦法把兩者的輸入和輸出“接”起來。這正是 Shell 自己的“傑作”，它通過 C 語言裡頭的 `pipe` 函數創建了一個管道（一個包含兩個文件描述符的整形數組，一個描述符用於寫入數據，一個描述符用於讀入數據），並且通過 `dup/fcntl` 把 `cat` 的輸出複製到了管道的輸入，而把管道的輸出則複製到了 `grep` 的輸入。這真是一個奇妙的想法。

那 `&` 呢？當你在程序的最後跟上這個奇妙的字符以後就可以接著做其他事情了，看看效果：

```
$ sleep 50 & #讓程序在後臺運行
[1] 8261
```

提示符被打印出來，可以輸入東西，讓程序到前臺運行，無法輸入東西了，按下 `CTRL+Z`，再讓程序到後臺運行：

```
$ fg %1
sleep 50

[1]+  Stopped                  sleep 50
```


實際上 `&` 正是 Shell 支持作業控制的表徵，通過作業控制，用戶在命令行上可以同時作幾個事情（把當前不做的放到後臺，用 `&` 或者 `CTRL+Z` 或者 `bg`）並且可以自由地選擇當前需要執行哪一個（用 `fg` 調到前臺）。這在實現時應該涉及到很多東西，包括終端會話（`session`）、終端信號、前臺進程、後臺進程等。而在命令的後面加上 `&` 後，該命令將被作為後臺進程執行，後臺進程是什麼呢？這類進程無法接收用戶發送給終端的信號（如 `SIGHUP`，`SIGQUIT`，`SIGINT`），無法響應鍵盤輸入（被前臺進程佔用著），不過可以通過 `fg` 切換到前臺而享受作為前臺進程具有的特權。

因此，當一個命令被加上 `&` 執行後，Shell 必須讓它具有後臺進程的特徵，讓它無法響應鍵盤的輸入，無法響應終端的信號（意味忽略這些信號），並且比較重要的是新的命令提示符得打印出來，並且讓命令行接口可以繼續執行其他命令，這些就是 Shell 對 `&` 的執行動作。

還有什麼神祕的呢？你也可以寫自己的 Shell 了，並且可以讓內核啟動後就執行它 `1`，在 `lilo` 或者 `grub` 的啟動參數上設置 `init=/path/to/your/own/shell/program` 就可以。當然，也可以把它作為自己的登錄 Shell，只需要放到 `/etc/passwd` 文件中相應用戶名所在行的最後就可以。不過貌似到現在還沒介紹 Shell 是怎麼執程序，是怎樣讓程序變成進程的，所以繼續。

/bin/bash 用什麼魔法讓一個普通程序變成了進程

當我們從鍵盤鍵入一串命令，Shell 奇妙地響應了，對於內置命令和函數，Shell 自身就可以解析了（通過 `switch`，`case` 之類的 C 語言語句）。但是，如果這個命令是磁盤上的一個文件呢。它找到該文件以後，怎麼執行它的呢？

還是用 `strace` 來跟蹤一個命令的執行過程看看。

```
$ strace -f -o strace.log /usr/bin/test
hello, world!
$ cat strace.log | sed -ne "1p"    #我們對第一行很感興趣
8445  execve("/usr/bin/test", ["/usr/bin/test"], [/ * 33 vars */]) = 0
```

從跟蹤到的結果的第一行可以看到 `bash` 通過 `execve` 調用了 `/usr/bin/test`，並且給它傳了 33 個參數。這 33 個 `vars` 是什麼呢？看看 `declare -x` 的結果（這個結果只有 32 個，原因是 `vars` 的最後一個變量需要是一個結束標誌，即 `NULL`）。

```
$ declare -x | wc -l    #declare -x聲明的環境變量將被導出到子進程中
32
$ export TEST="just a test"    #為了認證declare -x和之前的vars的個數的關係，再加一個
$ declare -x | wc -l
33
$ strace -f -o strace.log /usr/bin/test    #再次跟蹤，看看這個關係
hello, world!
$ cat strace.log | sed -ne "1p"
8523  execve("/usr/bin/test", ["/usr/bin/test"], [/ * 34 vars */]) = 0
```

通過這個演示發現，當前 Shell 的環境變量中被設置為 `export` 的變量被複制到了新的程序裡頭。不過雖然我們認為 Shell 執行新程序時是在一個新的進程裡頭執行的，但是 `strace` 並沒有跟蹤到諸如 `fork` 的系統調用（可能是 `strace` 自己設計的時候並沒有跟蹤 `fork`，或者是在 `fork` 之後才跟蹤）。但是有一

個事實我們不得不承認：當前 Shell 並沒有被新程序的進程替換，所以說 Shell 肯定是先調用 `fork`（也有可能是 `vfork`）創建了一個子進程，然後再調用 `execve` 執行新程序的。如果你還不相信，那麼直接通過 `exec` 執行新程序看看，這個可是直接把當前 Shell 的進程替換掉的。

```
exec /usr/bin/test
```

該可以看到當前 Shell “譁”（聽不到，突然沒了而已）的一下就沒有了。

下面來模擬一下 Shell 執行普通程序。`multiprocess` 相當於當前 Shell，而 `/usr/bin/test` 則相當於通過命令行傳遞給 Shell 的一個程序。這裡是代碼：

```
/* multiprocess.c */
#include <stdio.h>
#include <sys/types.h>
#include <unistd.h>      /* sleep, fork, _exit */

int main()
{
    int child;
    int status;

    if( (child = fork()) == 0) {      /* child */
        printf("child: my pid is %d\n", getpid());
        printf("child: my parent's pid is %d\n", getppid());
        execlp("/usr/bin/test", "/usr/bin/test", (char *)NULL);
    } else if(child < 0){             /* error */
        printf("create child process error!\n");
        _exit(0);
    }                                 /* parent */
    printf("parent: my pid is %d\n", getpid());
    if ( wait(&status) == child ) {
        printf("parent: wait for my child exit successfully!\n");
    }
}
```

運行看看，

```
$ make multiprocess
$ ./multiprocess
child: my pid is 2251
child: my parent's pid is 2250
hello, world!
parent: my pid is 2250
parent: wait for my child exit successfully!
```

從執行結果可以看出，`/usr/bin/test` 在 `multiprocess` 的子進程中運行並不干擾父進程，因為父進程一直等到了 `/usr/bin/test` 執行完成。

再回頭看看代碼，你會發現 `execlp` 並沒有傳遞任何環境變量信息給 `/usr/bin/test`，到底是怎麼把環境變量傳送過去的呢？通過 `man exec` 我們可以看到一組 `exec` 的調用，在裡頭並沒有發現 `execve`，但

是通過 `man execve` 可以看到該系統調用。實際上 `exec` 的那一組調用都只是 `libc` 庫提供的，而 `execve` 才是真正的系統調用，也就是說無論使用 `exec` 調用中的哪一個，最終調用的都是 `execve`，如果使用 `execlp`，那麼 `execlp` 將通過一定的處理把參數轉換為 `execve` 的參數。因此，雖然我們沒有傳遞任何環境變量給 `execlp`，但是默認情況下，`execlp` 把父進程的環境變量複製給了子進程，而這個動作是在 `execlp` 函數內部完成的。

現在，總結一下 `execve`，它有有三個參數，

- 第一個是程序本身的絕對路徑，對於剛才使用的 `execlp`，我們沒有指定路徑，這意味著它會設法到 `PATH` 環境變量指定的路徑下去尋找程序的全路徑。
- 第二個參數是一個將傳遞給被它執行的程序的參數數組指針。正是這個參數把我們從命令行上輸入的那些參數，諸如 `grep` 命令的 `-v` 等傳遞給了新程序，可以通過 `main` 函數的第二個參數 `char argv[]` 獲得這些內容。
- 第三個參數是一個將傳遞給被它執行的程序的环境變量，這些環境變量也可以通過 `main` 函數的第三個變量獲取，只要定義一個 `char env[]` 就可以了，只是通常不直接用它罷了，而是通過另外的方式，通過 `extern char ** environ` 全局變量（環境變量表的指針）或者 `getenv` 函數來獲取某個環境變量的值。

當然，實際上，當程序被 `execve` 執行後，它被加載到了內存裡，包括程序的各種指令、數據以及傳遞給它的各種參數、環境變量等都被存放在系統分配給該程序的內存空間中。

我們可以通過 `/proc/<pid>/maps` 把一個程序對應的進程的內存映象看個大概。

```
$ cat /proc/self/maps #查看cat程序自身加載後對應進程的內存映像
08048000-0804c000 r-xp 00000000 03:01 273716 /bin/cat
0804c000-0804d000 rw-p 00003000 03:01 273716 /bin/cat
0804d000-0804e000 rw-p 0804d000 00:00 0 [heap]
b7c46000-b7e46000 r--p 00000000 03:01 87528 /usr/lib/locale/locale-archive
b7e46000-b7e47000 rw-p b7e46000 00:00 0
b7e47000-b7f83000 r-xp 00000000 03:01 466875 /lib/libc-2.5.so
b7f83000-b7f84000 r--p 0013c000 03:01 466875 /lib/libc-2.5.so
b7f84000-b7f86000 rw-p 0013d000 03:01 466875 /lib/libc-2.5.so
b7f86000-b7f8a000 rw-p b7f86000 00:00 0
b7fa1000-b7fbc000 r-xp 00000000 03:01 402817 /lib/ld-2.5.so
b7fbc000-b7fbe000 rw-p 0001b000 03:01 402817 /lib/ld-2.5.so
bfcdf000-bfcdf4000 rw-p bfcdf000 00:00 0 [stack]
ffffe000-fffff000 r-xp 00000000 00:00 0 [vdso]
```

關於程序加載和進程內存映象的更多細節請參考《[C 語言程序緩衝區注入分析](#)》。

到這裡，關於命令行的祕密都被“曝光”了，可以開始寫自己的命令行解釋程序了。

關於進程的相關操作請參考《[進程與進程的基本操作](#)》。

補充：上面沒有討論到一個比較重要的內容，那就是即使 `execve` 找到了某個可執行文件，如果該文件屬主沒有運行該程序的權限，那麼也沒有辦法運程序。可通過 `ls -l` 查看程序的權限，通過 `chmod` 添加或者去掉可執行權限。

文件屬主具有可執行權限時才可以執行某個程序：

```
$ whoami
```

```
falcon
$ ls -l hello  #查看用戶權限(第一個x表示屬主對該程序具有可執行權限)
-rwxr-xr-x 1 falcon users 6383 2000-01-23 07:59 hello*
$ ./hello
Hello World
$ chmod -x hello  #去掉屬主的可執行權限
$ ls -l hello
-rw-r--r-- 1 falcon users 6383 2000-01-23 07:59 hello
$ ./hello
-bash: ./hello: Permission denied
```

參考資料

- Linux 啟動過程: `man boot-scripts`
- Linux 內核啟動參數: `man bootparam`
- `man 5 passwd`
- `man shadow`
- 《UNIX 環境高級編程》，進程關係一章

動態符號鏈接的細節

- [前言](#)
- [基本概念](#)
 - [ELF](#)
 - [符號](#)
 - [重定位](#)：是將符號引用與符號定義進行鏈接的過程
 - [動態鏈接](#)
 - [動態鏈接庫](#)
 - [動態鏈接器（dynamic linker/loader）](#)
 - [過程鏈接表（plt）](#)
 - [全局偏移表（got）](#)
 - [重定位表](#)
- [動態鏈接庫的創建和調用](#)
 - [創建動態鏈接庫](#)
 - [隱式使用該庫](#)
 - [顯式使用庫](#)
- [動態鏈接過程](#)
- [參考資料](#)

前言

Linux 支持動態鏈接庫，不僅節省了磁盤、內存空間，而且[可以提高程序運行效率](#)。不過引入動態鏈接庫也可能會帶來很多問題，例如[動態鏈接庫的調試](#)、[升級更新](#)和潛在的安全威脅[\[1\]](#), [\[2\]](#)。這裡主要討論符號的動態鏈接過程，即程序在執行過程中，對其中包含的一些未確定地址的符號進行重定位的過程[\[1\]](#), [\[2\]](#)。

本篇主要參考資料[\[3\]](#)和[\[8\]](#)，前者側重實踐，後者側重原理，把兩者結合起來就方便理解程序的動態鏈接過程了。另外，動態鏈接庫的創建、使用以及調用動態鏈接庫的部分參考了資料[\[1\]](#), [\[2\]](#)。

下面先來看看幾個基本概念，接著就介紹動態鏈接庫的創建、隱式和顯示調用，最後介紹符號的動態鏈接細節。

基本概念

ELF

ELF 是 Linux 支持的一種程序文件格式，本身包含重定位、執行、共享（動態鏈接庫）三種類型（`man elf`）。

代碼：

```
/* test.c */
#include <stdio.h>

int global = 0;
```

```
int main()
{
    char local = 'A';

    printf("local = %c, global = %d\n", local, global);

    return 0;
}
```

演示：

通過 `-c` 生成可重定位文件 `test.o`，這裡不會進行鏈接：

```
$ gcc -c test.c
$ file test.o
test.o: ELF 32-bit LSB relocatable, Intel 80386, version 1 (SYSV), not stripped
```

鏈接後才可以執行：

```
$ gcc -o test test.o
$ file test
test: ELF 32-bit LSB executable, Intel 80386, version 1 (SYSV), dynamically linked (uses shared objects)
```

也可鏈接成動態鏈接庫，不過一般不會把 `main` 函數鏈接成動態鏈接庫，後面再介紹：

```
$ gcc -fpic -shared -Wl,-soname,libtest.so.0 -o libtest.so.0.0 test.o
$ file libtest.so.0.0
libtest.so.0.0: ELF 32-bit LSB shared object, Intel 80386, version 1 (SYSV), not stripped
```

雖然 `ELF` 文件本身就支持三種不同的類型，不過它有一個統一的結構。這個結構是：

```
文件頭部(ELF Header)
程序頭部表(Program Header Table)
節區1(Section1)
節區2(Section2)
節區3(Section3)
...
節區頭部表(Section Header Table)
```

無論是文件頭部、程序頭部表、節區頭部表，還是節區，它們都對應著 C 語言裡頭的一些結構體（`elf.h` 中定義）。文件頭部主要描述 `ELF` 文件的類型，大小，運行平臺，以及和程序頭部表和節區頭部表相關的信息。節區頭部表則用於可重定位文件，以便描述各個節區的信息，這些信息包括節區的名字、類型、大小等。程序頭部表則用於描述可執行文件或者動態鏈接庫，以便系統加載和執行它們。而節區主要存放各種特定類型的信息，比如程序的正文區（代碼）、數據區（初始化和未初始化的數據）、調試信息、以及用於動態鏈接的一些節區，比如解釋器（`.interp`）節區將指定程序動態裝載 / 鏈接器 `ld-linux.so` 的

位置，而過程鏈接表（`plt`）、全局偏移表（`got`）、重定位表則用於輔助動態鏈接過程。

符號

對於可執行文件除了編譯器引入的一些符號外，主要就是用戶自定義的全局變量，函數等，而對於可重定位文件僅僅包含用戶自定義的一些符號。

- 生成可重定位文件

```
$ gcc -c test.c
$ nm test.o
00000000 B global
00000000 T main
                U printf
```

上面包含全局變量、自定義函數以及動態鏈接庫中的函數，但不包含局部變量，而且發現這三個符號的地址都沒有確定。

注：`nm` 命令可用來查看 `ELF` 文件的符號表信息。

- 生成可執行文件

```
$ gcc -o test test.o
$ nm test | egrep "main$| printf|global$"
080495a0 B global
08048354 T main
                U printf@@GLIBC_2.0
```

經鏈接，`global` 和 `main` 的地址都已經確定了，但是 `printf` 卻還沒，因為它是動態鏈接庫 `glibc` 中定義的。

重定位：是將符號引用與符號定義進行鏈接的過程

從上面的演示可以看出，重定位文件 `test.o` 中的符號地址都是沒有確定的，而經過靜態鏈接（`gcc` 默認調用 `ld` 進行鏈接）以後有兩個符號地址已經確定了，這樣一個確定符號地址的過程實際上就是鏈接的實質。鏈接過後，對符號的引用變成了對地址（定義符號時確定該地址）的引用，這樣程序運行時就可通過訪問內存地址而訪問特定的數據。

我們也注意到符號 `printf` 在可重定位文件和可執行文件中的地址都沒有確定，這意味著該符號是一個外部符號，可能定義在動態鏈接庫中，在程序運行時需要通過動態鏈接器（`ld-linux.so`）進行重定位，即動態鏈接。

通過這個演示可以看出 `printf` 確實在 `glibc` 中有定義。

```
$ nm -D /lib/`uname -m`-linux-gnu/libc.so.6 | grep "\ printf$"
0000000000053840 T printf
```

除了 `nm` 以外，還可以用 `readelf -s` 查看 `.dynsym` 表或者用 `objdump -tT` 查看。

需要提到的是，用 `nm` 命令不帶 `-D` 參數的話，在較新的系統上已經沒有辦法查看 `libc.so` 的符號表了，因為 `nm` 默認打印常規符號表（在 `.symtab` 和 `.strtab` 節區中），但是，在打包時為了減少系統大小，這些符號已經被 `strip` 掉了，只保留了動態符號（在 `.dynsym` 和 `.dynstr` 中）以便動態鏈接器在執行程序時尋址這些外部用到的符號。而常規符號除了動態符號以外，還包含有一些靜態符號，比如說本地函數，這個信息主要是調試器會用，對於正常部署的系統，一般會用 `strip` 工具刪除掉。

關於 `nm` 與 `readelf -s` 的詳細比較，可參考：[nm vs “readelf -s”](#)。

動態鏈接

動態鏈接就是在程序運行時對符號進行重定位，確定符號對應的內存地址的過程。

Linux 下符號的動態鏈接默認採用 [Lazy Mode](#) 方式，也就是說在程序運行過程中用到該符號時才去解析它的地址。這樣一種符號解析方式有一個好處：只解析那些用到的符號，而對那些不用的符號則永遠不用解析，從而提高程序的執行效率。

不過這種默認是可以通過設置 `LD_BIND_NOW` 為非空來打破的（下面會通過實例來分析這個變量的作用），也就是說如果設置了這個變量，動態鏈接器將在程序加載後和符號被使用之前就對這些符號的地址進行解析。

動態鏈接庫

上面提到重定位的過程就是對符號引用和符號地址進行鏈接的過程，而動態鏈接過程涉及到的符號引用和符號定義分別對應可執行文件和動態鏈接庫，在可執行文件中可能引用了某些動態鏈接庫中定義的符號，這類符號通常是函數。

為了讓動態鏈接器能夠進行符號的重定位，必須把動態鏈接庫的相關信息寫入到可執行文件當中，這些信息是什麼呢？

```
$ readelf -d test | grep NEEDED
0x00000001 (NEEDED)                               Shared library: [libc.so.6]
```

ELF 文件有一個特別的節區：`.dynamic`，它存放了和動態鏈接相關的很多信息，例如動態鏈接器通過它找到該文件使用的動態鏈接庫。不過，該信息並未包含動態鏈接庫 `libc.so.6` 的絕對路徑，那動態鏈接器去哪裡查找相應的庫呢？

通過 `LD_LIBRARY_PATH` 參數，它類似 Shell 解釋器中用於查找可執行文件的 `PATH` 環境變量，也是通過冒號分開指定了各個存放庫函數的路徑。該變量實際上也可以通過 `/etc/ld.so.conf` 文件來指定，一行對應一個路徑名。為了提高查找和加載動態鏈接庫的效率，系統啟動後會通過 `ldconfig` 工具創建一個庫的緩存 `/etc/ld.so.cache`。如果用戶通過 `/etc/ld.so.conf` 加入了新的庫搜索路徑或者是把新庫加到某個原有的庫目錄下，最好是執行一下 `ldconfig` 以便刷新緩存。

需要補充的是，因為動態鏈接庫本身還可能引用其他的庫，那麼一個可執行文件的動態符號鏈接過程可能涉及到多個庫，通過 `readelf -d` 可以打印出該文件直接依賴的庫，而通過 `ldd` 命令則可以打印出所有依賴或者間接依賴的庫。


```
$ ldd test
linux-gate.so.1 => (0xffffe000)
libc.so.6 => /lib/libc.so.6 (0xb7da2000)
/lib/ld-linux.so.2 (0xb7efc000)
```

`libc.so.6` 通過 `readelf -d` 就可以看到的，是直接依賴的庫；而 `linux-gate.so.1` 在文件系統中並沒有對應的庫文件，它是一個虛擬的動態鏈接庫，對應進程內存映像的內核部分，更多細節請參考資料[11]；而 `/lib/ld-linux.so.2` 正好是動態鏈接器，系統需要用它來進行符號重定位。那 `ldd` 是怎麼知道 `/lib/ld-linux.so` 就是該文件的動態鏈接器呢？

那是因為 `ELF` 文件通過專門的節區指定了動態鏈接器，這個節區就是 `.interp`。

```
$ readelf -x .interp test

Hex dump of section '.interp':
0x08048114 2f6c6962 2f6c642d 6c696e75 782e736f /lib/ld-linux.so
0x08048124 2e3200                                .2.
```

可以看到這個節區剛好有字符串 `/lib/ld-linux.so.2`，即 `ld-linux.so` 的絕對路徑。

我們發現，與 `libc.so` 不同的是，`ld-linux.so` 的路徑是絕對路徑，而 `libc.so` 僅僅包含了文件名。原因是：程序被執行時，`ld-linux.so` 將最先被裝載到內存中，沒有其他程序知道去哪裡查找 `ld-linux.so`，所以它的路徑必須是絕對的；當 `ld-linux.so` 被裝載以後，由它來去裝載可執行文件和相關的共享庫，它將根據 `PATH` 變量和 `LD_LIBRARY_PATH` 變量去磁盤上查找它們，因此可執行文件和共享庫都可以不指定絕對路徑。

下面著重介紹動態鏈接器本身。

動態鏈接器（dynamic linker/loader）

Linux 下 `elf` 文件的動態鏈接器是 `ld-linux.so`，即 `/lib/ld-linux.so.2`。從名字來看和靜態鏈接器 `ld`（`gcc` 默認使用的鏈接器，見參考資料[10]）類似。通過 `man ld-linux` 可以獲取與動態鏈接器相關的資料，包括各種相關的環境變量和文件都有詳細的說明。

對於環境變量，除了上面提到過的 `LD_LIBRARY_PATH` 和 `LD_BIND_NOW` 變量外，還有其他幾個重要參數，比如 `LD_PRELOAD` 用於指定預裝載一些庫，以便替換其他庫中的函數，從而做一些安全方面的處理 [6]，[9]，[12]，而環境變量 `LD_DEBUG` 可以用來進行動態鏈接的相關調試。

對於文件，除了上面提到的 `ld.so.conf` 和 `ld.so.cache` 外，還有一個文件 `/etc/ld.so.preload` 用於指定需要預裝載的庫。

從上一小節中發現有一個專門的節區 `.interp` 存放有動態鏈接器，但是這個節區為什麼叫做 `.interp`（`interpreter`）呢？因為當 `Shell` 解釋器或者其他父進程通過 `exec` 啟動我們的程序時，系統會先為 `ld-linux` 創建內存映像，然後把控制權交給 `ld-linux`，之後 `ld-linux` 負責為可执行程序提供運行環境，負責解釋程序的運行，因此 `ld-linux` 也叫做 `dynamic loader`（或 `intepreter`）（關於程序的加載過程請參考資料 [13]）

那麼在 `exec（）` 之後和程序指令運行之前的過程是怎樣的呢？`ld-linux.so` 主要為程序本身創建了內

存映像（以下內容摘自資料 [8]），大體過程如下：

- 將可執行文件的內存段添加到進程映像中；
- 把共享目標內存段添加到進程映像中；
- 為可執行文件和它的共享目標（動態鏈接庫）執行重定位操作；
- 關閉用來讀入可執行文件的文件描述符，如果動態鏈接程序收到過這樣的文件描述符的話；
- 將控制轉交給程序，使得程序好像從 `exec()` 直接得到控制

關於第 1 步，在 `ELF` 文件的文件頭中就指定了該文件的入口地址，程序的代碼和數據部分會相繼 `map` 到對應的內存中。而關於可執行文件本身的路徑，如果指定了 `PATH` 環境變量，`ld-linux` 會到 `PATH` 指定的相關目錄下查找。

```
$ readelf -h test | grep Entry
Entry point address:          0x80482b0
```

對於第 2 步，上一節提到的 `.dynamic` 節區指定了可執行文件依賴的庫名，`ld-linux`（在這裡叫做動態裝載器或程序解釋器比較合適）再從 `LD_LIBRARY_PATH` 指定的路徑中找到相關的庫文件或者直接從 `/etc/ld.so.cache` 庫緩衝中加載相關庫到內存中。（關於進程的內存映像，推薦參考資料 [14]）

對於第 3 步，在前面已提到，如果設置了 `LD_BIND_NOW` 環境變量，這個動作就會在此時發生，否則將會採用 `lazy mode` 方式，即當某個符號被使用時才會進行符號的重定位。不過無論在什麼時候發生這個動作，重定位的過程大體是一樣的（在後面將主要介紹該過程）。

對於第 4 步，這個主要是釋放文件描述符。

對於第 5 步，動態鏈接器把程序控制權交還給程序。

現在關心的主要是第 3 步，即如何進行符號的重定位？下面來探求這個過程。期間會逐步討論到和動態鏈接密切相關的三個數據結構，它們分別是 `ELF` 文件的過程鏈接表、全局偏移表和重定位表，這三個表都是 `ELF` 文件的節區。

過程鏈接表（plt）

從上面的演示發現，還有一個 `printf` 符號的地址沒有確定，它應該在動態鏈接庫 `libc.so` 中定義，需要進行動態鏈接。這裡假設採用 `lazy mode` 方式，即執行到 `printf` 所在位置時才去解析該符號的地址。

假設當前已經執行到了 `printf` 所在位置，即 `call printf`，我們通過 `objdump` 反編譯 `test` 程序的正文段看看。

```
$ objdump -d -s -j .text test | grep printf
804837c:      e8 1f ff ff ff      call    80482a0 <printf@plt>
```

發現，該地址指向了 `plt`（即過程鏈接表）即地址 `80482a0` 處。下面查看該地址處的內容。

```
$ objdump -D test | grep "80482a0" | grep -v call
```

```
080482a0 <printf@plt>:
80482a0:      ff 25 8c 95 04 08      jmp     *0x804958c
```

發現 80482a0 地址對應的是一條跳轉指令，跳轉到 0x804958c 地址指向的地址。到底 0x804958c 地址本身在什麼地方呢？我們能否從 .dynamic 節區（該節區存放了和動態鏈接相關的數據）獲取相關的信息呢？

```
$ readelf -d test
```

```
Dynamic section at offset 0x4ac contains 20 entries:
```

Tag	Type	Name/Value
0x00000001	(NEEDED)	Shared library: [libc.so.6]
0x0000000c	(INIT)	0x8048258
0x0000000d	(FINI)	0x8048454
0x00000004	(HASH)	0x8048148
0x00000005	(STRTAB)	0x80481c0
0x00000006	(SYMTAB)	0x8048170
0x0000000a	(STRSZ)	76 (bytes)
0x0000000b	(SYMENT)	16 (bytes)
0x00000015	(DEBUG)	0x0
0x00000003	(PLTGOT)	0x8049578
0x00000002	(PLTRELSZ)	24 (bytes)
0x00000014	(PLTREL)	REL
0x00000017	(JMPREL)	0x8048240
0x00000011	(REL)	0x8048238
0x00000012	(RELSZ)	8 (bytes)
0x00000013	(RELENT)	8 (bytes)
0x6ffffffe	(VERNEED)	0x8048218
0x6fffffff	(VERNEEDNUM)	1
0x6ffffff0	(VERSYM)	0x804820c
0x00000000	(NULL)	0x0

發現 0x8049578 地址和 0x804958c 地址比較近，通過資料 [8] 查到前者正好是 .got.plt（即過程鏈接表）對應的全局偏移表的入口地址。難道 0x804958c 正好位於 .got.plt 節區中？

全局偏移表（got）

現在進入全局偏移表看看，

```
$ readelf -x .got.plt test
```

```
Hex dump of section '.got.plt':
```

```
0x08049578 ac940408 00000000 00000000 86820408 .....
0x08049588 96820408 a6820408 .....
```

從上述結果可以看出 0x804958c 地址（即 0x08049588+4）處存放的是 a6820408，考慮到我的實驗平臺是 i386，字節順序是 little-endian 的，所以實際數值應該是 080482a6，也就是說 *（0x804958c）的值是 080482a6，這個地址剛好是過程鏈接表的最後一項 call 80482a0printf@plt 中 80482a0 地址往後偏移 6 個字節，容易猜到該地址應該就是 jmp 指令的後一條地址。

```
$ objdump -d -d -s -j .plt test | grep "080482a0 <printf@plt>:" -A 3
080482a0 <printf@plt>:
80482a0:    ff 25 8c 95 04 08      jmp     *0x804958c
80482a6:    68 10 00 00 00         push    $0x10
80482ab:    e9 c0 ff ff ff        jmp     8048270 <_init+0x18>
```

80482a6 地址恰巧是一條 `push` 指令，隨後是一條 `jmp` 指令（暫且不管 `push` 指令入棧的內容有什麼意義），執行完 `push` 指令之後，就會跳轉到 8048270 地址處，下面看看 8048270 地址處到底有哪些指令。

```
$ objdump -d -d -s -j .plt test | grep -v "jmp     8048270 <_init+0x18>" | grep "08048270" -A 3
08048270 <__gmon_start__@plt-0x10>:
8048270:    ff 35 7c 95 04 08      pushl   0x804957c
8048276:    ff 25 80 95 04 08      jmp     *0x8049580
```

同樣是一條入棧指令跟著一條跳轉指令。不過這兩個地址 0x804957c 和 0x8049580 是連續的，而且都很熟悉，剛好都在 `.got.plt` 表裡頭（從上面我們已經知道 `.got.plt` 的入口是 0x08049578）。這樣的話，我們得確認這兩個地址到底有什麼內容。

```
$ readelf -x .got.plt test

Hex dump of section '.got.plt':
0x08049578 ac940408 00000000 00000000 86820408 .....
0x08049588 96820408 a6820408 .....
```

不過，遺憾的是通過 `readelf` 查看到的這兩個地址信息都是 0，它們到底是什麼呢？

現在只能求助參考資料 [8]，該資料的“3.8.5 過程鏈接表”部分在介紹過程鏈接表和全局偏移表相互合作解析符號的過程中的三步涉及到了這兩個地址和前面沒有說明的 `push $ 0x10` 指令。

- 在程序第一次創建內存映像時，動態鏈接器為全局偏移表的第二（0x804957c）和第三項（0x8049580）設置特殊值。
- 原步驟 5。在跳轉到 08048270 <__gmon_start__@plt-0x10>，即過程鏈接表的第一項之前，有一條壓入棧指令，即 `push $0x10`，0x10 是相對於重定位表起始地址的一個偏移地址，這個偏移地址到底有什麼用呢？它應該是提供給動態鏈接器的什麼信息吧？後面再說明。
- 原步驟 6。跳轉到過程鏈接表的第一項之後，壓入了全局偏移表中的第二項（即 0x804957c 處），“為動態鏈接器提供了識別信息的機會”（具體是什麼呢？後面會簡單提到，但這個並不是很重要），然後跳轉到全局偏移表的第三項（0x8049580，這一項比較重要），把控制權交給動態鏈接器。

從這三步發現程序運行時地址 0x8049580 處存放的應該是動態鏈接器的入口地址，而重定位表 0x10 位置處和 0x804957c 處應該為動態鏈接器提供瞭解析符號需要的某些信息。

在繼續之前先總結一下過程鏈接表和全局偏移表。上面的操作過程僅僅從“局部”看過了這兩個表，但是並沒有宏觀地看裡頭的內容。下面將宏觀的分析一下，對於過程鏈接表：

```
$ objdump -d -s -j .plt test
08048270 <__gmon_start__@plt-0x10>:
  8048270:    ff 35 7c 95 04 08    pushl  0x804957c
  8048276:    ff 25 80 95 04 08    jmp     *0x8049580
  804827c:    00 00                add     %al, (%eax)
  ...

08048280 <__gmon_start__@plt>:
  8048280:    ff 25 84 95 04 08    jmp     *0x8049584
  8048286:    68 00 00 00 00        push    $0x0
  804828b:    e9 e0 ff ff ff       jmp     8048270 <_init+0x18>

08048290 <__libc_start_main@plt>:
  8048290:    ff 25 88 95 04 08    jmp     *0x8049588
  8048296:    68 08 00 00 00        push    $0x8
  804829b:    e9 d0 ff ff ff       jmp     8048270 <_init+0x18>

080482a0 <printf@plt>:
  80482a0:    ff 25 8c 95 04 08    jmp     *0x804958c
  80482a6:    68 10 00 00 00        push    $0x10
  80482ab:    e9 c0 ff ff ff       jmp     8048270 <_init+0x18>
```

除了該表中的第一項外，其他各項實際上是類似的。而最後一項 080482a0 <printf@plt> 和第一項我們都分析過，因此不難理解其他幾項的作用。過程鏈接表沒有辦法單獨工作，因為它和全局偏移表是關聯的，所以在說明它的作用之前，先從總體上來看一下全局偏移表。

```
$ readelf -x .got.plt test

Hex dump of section '.got.plt':
 0x08049578 ac940408 00000000 00000000 86820408 .....
 0x08049588 96820408 a6820408 .....
```

比較全局偏移表中 0x08049584 處開始的數據和過程鏈接表第二項開始的連續三項中 push 指定所在的地 址，不難發現，它們是對應的。而 0x0804958c 即 push 0x10 對應的地址我們剛才提到過（下一節會進一步分析），其他幾項的作用類似，都是跳回到過程鏈接表的 push 指令處，隨後就跳轉到過程鏈接表的第一項，以便解析相應的符號（實際上過程鏈接表的第一個表項是進入動態鏈接器，而之前的連續兩個指令則傳送了需要解析的符號等信息）。另外 0x08049578 和 0x08049580 處分別存放有傳遞給動態鏈接庫的相關信息和動態鏈接器本身的入口地址。但是還有一個地址 0x08049578，這個地址剛好是 .dynamic 的入口地址，該節區存放了和動態鏈接過程相關的信息，資料 [8] 提到這個表項實際上保留給動態鏈接器自己使用的，以便在不依賴其他程序的情況下對自己進行初始化，所以下面將不再關注該表項。

```
$ objdump -D test | grep 080494ac
080494ac <_DYNAMIC>:
```

重定位表

這裡主要接著上面的 push 0x10 指令來分析。通過資料 [8] 發現重定位表包含如何修改其他節區的信息，以便動態鏈接器對某些節區內的符號地址進行重定位（修改為新的地址）。那到底重定位表項提供了什麼樣的信息呢？

動態符號鏈接的細節

- 每一個重定位項有三部分內容，我們重點關注前兩部分。
- 第一部分是 `r_offset`，這裡考慮的是可執行文件，因此根據資料發現，它的取值是被重定位影響（可以說改變或修改）到的存儲單元的虛擬地址。
- 第二部分是 `r_info`，此成員給出要進行重定位的符號表索引（重定位表項引用到的符號表），以及將實施的重定位類型（如何進行符號的重定位）。(Type)。

先來看看重定位表的具體內容，

```
$ readelf -r test

Relocation section '.rel.dyn' at offset 0x238 contains 1 entries:
  Offset      Info      Type           Sym.Value  Sym. Name
 08049574  00000106  R_386_GLOB_DAT  00000000  __gmon_start__

Relocation section '.rel.plt' at offset 0x240 contains 3 entries:
  Offset      Info      Type           Sym.Value  Sym. Name
 08049584  00000107  R_386_JUMP_SLOT  00000000  __gmon_start__
 08049588  00000207  R_386_JUMP_SLOT  00000000  __libc_start_main
 0804958c  00000407  R_386_JUMP_SLOT  00000000  printf
```

僅僅關注和過程鏈接表相關的 `.rel.plt` 部分，`0x10` 剛好是 $1 \times 16 + 0 \times 1$ ，即 16 字節，作為重定位表的偏移，剛好對應該表的第三行。發現這個結果中竟然包含了和 `printf` 符號相關的各種信息。不過重定位表中沒有直接指定符號 `printf`，而是根據 `r_info` 部分從動態符號表中計算出來的，注意觀察上述結果中的 `Info` 一列的 1, 2, 4 和下面結果的 `Num` 列的對應關係。

```
$ readelf -s test | grep ".dynsym" -A 6
Symbol table '.dynsym' contains 5 entries:
  Num:      Value      Size Type      Bind      Vis      Ndx Name
   0: 00000000         0 NOTYPE   LOCAL   DEFAULT   UND
   1: 00000000         0 NOTYPE   WEAK    DEFAULT   UND __gmon_start__
   2: 00000000       410 FUNC     GLOBAL   DEFAULT   UND __libc_start_main@GLIBC_2.0 (2)
   3: 08048474         4 OBJECT   GLOBAL   DEFAULT   14  _IO_stdin_used
   4: 00000000        57 FUNC     GLOBAL   DEFAULT   UND printf@GLIBC_2.0 (2)
```

也就是說在執行過程鏈接表中的第一項的跳轉指令（`jmp *0x8049580`）調用動態鏈接器以後，動態鏈接器因為有了 `push 0x10`，從而可以通過該重定位表項中的 `r_info` 找到對應符號（`printf`）在符號表（`.dynsym`）中的相關信息。

除此之外，符號表中還有 `Offset`（`r_offset`）以及 `Type` 這兩個重要信息，前者表示該重定位操作後可能影響的地址 `0804958c`，這個地址剛好是 `got` 表項的最後一項，原來存放的是 `push 0x10` 指令的地址。這意味著，該地址處的內容將被修改，而如何修改呢？根據 `Type` 類型 `R_386_JUMP_SLOT`，通過資料 [8] 查找到該類型對應的說明如下（原資料有誤，下面做了修改）：鏈接編輯器創建這種重定位類型主要是為了支持動態鏈接。其偏移地址成員給出過程鏈接表項的位置。動態鏈接器修改全局偏移表項的內容，把控制傳輸給指定符號的地址。

這說明，動態鏈接器將根據該類型對全局偏移表中的最有一項，即 `0804958c` 地址處的內容進行修改，修改為符號的實際地址，即 `printf` 函數在動態鏈接庫的內存映像中的地址。

到這裡，動態鏈接的宏觀過程似乎已經瞭然於心，不過一些細節還是不太清楚。

動態符號鏈接的細節

下面先介紹動態鏈接庫的創建，隱式調用和顯示調用，接著進一步澄清上面還不太清楚的細節，即全局偏移表中第二項到底傳遞給了動態鏈接器什麼信息？第三項是否就是動態鏈接器的地址？並討論通過設置 `LD_BIND_NOW` 而不採用默認的 `lazy mode` 進行動態鏈接和採用 `lazy mode` 動態鏈接的區別？

動態鏈接庫的創建和調用

在介紹動態符號鏈接的更多細節之前，先來了解一下動態鏈接庫的創建和兩種使用方法，進而引出符號解析的後臺細節。

創建動態鏈接庫

首先來創建一個簡單動態鏈接庫。

代碼：

```
/* myprintf.c */
#include <stdio.h>

int myprintf(char *str)
{
    printf("%s\n", str);
    return 0;
}
```

```
/* myprintf.h */
#ifndef _MYPRINTF_H
#define _MYPRINTF_H

int myprintf(char *);

#endif
```

演示：

```
$ gcc -c myprintf.c
$ gcc -shared -Wl,-soname,libmyprintf.so.0 -o libmyprintf.so.0.0.0 myprintf.o
$ ln -sf libmyprintf.so.0.0.0 libmyprintf.so.0
$ ln -fs libmyprintf.so.0 libmyprintf.so
$ ls
libmyprintf.so  libmyprintf.so.0  libmyprintf.so.0.0.0  myprintf.c  myprintf.h  myprintf.o
```

得到三個文件 `libmyprintf.so`，`libmyprintf.so.0`，`libmyprintf.so.0.0.0`，這些庫暫且存放在當前目錄下。這裡有一個問題值得關注，那就是為什麼要創建兩個符號鏈接呢？答案是為了在不影響兼容性的前提下升級庫 [5]。

隱式使用該庫

現在寫一段代碼來使用該庫，調用其中的 `myprintf` 函數，這裡是隱式使用該庫：在代碼中並沒有直接使用該庫，而是通過調用 `myprintf` 隱式地使用了該庫，在編譯引用該庫的可執行文件時需要通過 `-l` 參數指定該庫的名字。

```
/* test.c */
#include <stdio.h>
#include <myprintf.h>

int main()
{
    myprintf("Hello World");

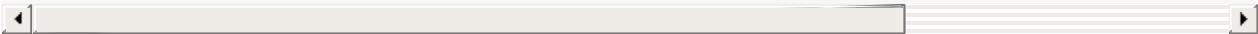
    return 0;
}
```

編譯：

```
$ gcc -o test test.c -lmyprintf -L./ -I./
```

直接運行 `test`，提示找不到該庫，因為庫的默認搜索路徑裡頭沒有包含當前目錄：

```
$ ./test
./test: error while loading shared libraries: libmyprintf.so: cannot open shared object file
```



如果指定庫的搜索路徑，則可以運行：

```
$ LD_LIBRARY_PATH=$PWD ./test
Hello World
```

顯式使用庫

`LD_LIBRARY_PATH` 環境變量使得庫可以放到某些指定的路徑下面，而無須在調用程序中顯式的指定該庫的絕對路徑，這樣避免了把程序限制在某些絕對路徑下，方便庫的移動。

雖然顯式調用有不便，但是能夠避免隱式調用搜索路徑的時間消耗，提高效率，除此之外，顯式調用為我們提供了一組函數調用，讓符號的重定位過程一覽無遺。

```
/* test1.c */

#include <dlfcn.h>      /* dlopen, dlsym, dlerror */
#include <stdlib.h>     /* exit */
#include <stdio.h>      /* printf */

#define LIB_SO_NAME    "./libmyprintf.so"
#define FUNC_NAME      "myprintf"
```



```
typedef int (*func)(char *);

int main(void)
{
    void *h;
    char *e;
    func f;

    h = dlopen(LIB_SO_NAME, RTLD_LAZY);
    if ( !h ) {
        printf("failed load library: %s\n", LIB_SO_NAME);
        exit(-1);
    }
    f = dlsym(h, FUNC_NAME);
    e = dlerror();
    if (e != NULL) {
        printf("search %s error: %s\n", FUNC_NAME, LIB_SO_NAME);
        exit(-1);
    }
    f("Hello World");

    exit(0);
}
```

演示：

```
$ gcc -o test1 test1.c -ldl
```

這種情況下，無須包含頭文件。從這個代碼中很容易看出符號重定位的過程：

- 首先通過 `dlopen` 找到依賴庫，並加載到內存中，再返回該庫的 `handle`，通過 `dlopen` 我們可以指定 `RTLD_LAZY` 採用 `lazy mode` 動態鏈接模式，如果採用 `RTLD_NOW` 則和隱式調用時設置 `LD_BIN_NOW` 類似。
- 找到該庫以後就是對某個符號進行重定位，這裡是確定 `myprintf` 函數的地址。
- 找到函數地址以後就可以直接調用該函數了。

關於 `dlopen`，`dlsym` 等後臺工作細節建議參考資料 [\[15\]](#)。

隱式調用的動態符號鏈接過程和上面類似。下面通過一些實例來確定之前沒有明確的兩個內容：即全局偏移表中的第二項和第三項，並進一步討論 `lazy mode` 和非 `lazy mode` 的區別。

動態鏈接過程

因為通過 `ELF` 文件，我們就可以確定全局偏移表的位置，因此為了確定全局偏移表位置的第三項和第四項的內容，有兩種辦法：

- 通過 `gdb` 調試。
- 直接在函數內部打印。

因為資料[\[3\]](#)詳細介紹了第一種方法，這裡試著通過第二種方法來確定這兩個地址的值。

```

/**
 * got.c -- get the relative content of the got(global offset table) of an elf file
 */

#include <stdio.h>

#define GOT 0x8049614

int main(int argc, char *argv[])
{
    long got2, got3;
    long old_addr, new_addr;

    got2=(long *)(GOT+4);
    got3=(long *)(GOT+8);
    old_addr=(long *)(GOT+24);

    printf("Hello World\n");

    new_addr=(long *)(GOT+24);

    printf("got2: 0x%0x, got3: 0x%0x, old_addr: 0x%0x, new_addr: 0x%0x\n",
          got2, got3, old_addr, new_addr);

    return 0;
}

```

在寫好上面的代碼後就需要確定全局偏移表的地址，然後把該地址設置為代碼中的宏 GOT。

```

$ make got
$ readelf -d got | grep PLTGOT
0x00000003 (PLTGOT)                0x8049614

```

注：這裡假設大家用的都是 i386 的系統，如果要在 x86_64 位系統上要編譯生成 i386 上的可執行文件，需要給 gcc 傳遞一個 -m32 參數，例如：

```
$ gcc -m32 -o got got.c
```

把地址 0x8049614 替換到上述代碼中，然後重新編譯運行，查看結果。

```

$ make got
$ Hello World
got2: 0xb7f376d8, got3: 0xb7f2ef10, old_addr: 0x80482da, new_addr: 0xb7e19a20
$ ./got
Hello World
got2: 0xb7f1e6d8, got3: 0xb7f15f10, old_addr: 0x80482da, new_addr: 0xb7e00a20

```

通過兩次運行，發現全局偏移表中的這兩項是變化的，並且 printf 的地址對應的 new_addr 也是變化的，說明 libc 和 ld-linux 這兩個庫啟動以後對應的虛擬地址並不確定。因此，無法直接跟蹤到那個地址處的內容，還得藉助調試工具，以便確認它們。

下面重新編譯 `got`，加上 `-g` 參數以便調試，並通過調試確認 `got2`，`got3`，以及調用 `printf` 前後 `printf` 地址的重定位情況。

```
$ gcc -g -o got got.c
$ gdb ./got
(gdb) l
5      #include <stdio.h>
6
7      #define GOT 0x8049614
8
9      int main(int argc, char *argv[])
10     {
11         long got2, got3;
12         long old_addr, new_addr;
13
14         got2=(long *)(GOT+4);
(gdb) l
15         got3=(long *)(GOT+8);
16         old_addr=(long *)(GOT+24);
17
18         printf("Hello World\n");
19
20         new_addr=(long *)(GOT+24);
21
22         printf("got2: 0x%0x, got3: 0x%0x, old_addr: 0x%0x, new_addr: 0x%0x\n",
23                got2, got3, old_addr, new_addr);
24
```

在第一個 `printf` 處設置一個斷點：

```
(gdb) break 18
Breakpoint 1 at 0x80483c3: file got.c, line 18.
```

在第二個 `printf` 處設置一個斷點：

```
(gdb) break 22
Breakpoint 2 at 0x80483dd: file got.c, line 22.
```

運行到第一個 `printf` 之前會停止：

```
(gdb) r
Starting program: /mnt/hda8/Temp/c/program/got

Breakpoint 1, main () at got.c:18
18         printf("Hello World\n");
```

查看執行 `printf` 之前的全局偏移表內容：

```
(gdb) x/8x 0x8049614
```

```

0x8049614 <_GLOBAL_OFFSET_TABLE_>:      0x08049548      0xb7f3c6d8      0xb7f33f10      0x00000000
0x8049624 <_GLOBAL_OFFSET_TABLE_+16>:    0xb7ddb20      0x080482ca      0x080482da      0x00000000

```

查看 GOT 表項的最有一項，發現剛好是 PLT 表中 push 指令的地址：

```

(gdb) disassemble 0x080482da
Dump of assembler code for function puts@plt:
0x080482d4 <puts@plt+0>:      jmp     *0x804962c
0x080482da <puts@plt+6>:      push    $0x18
0x080482df <puts@plt+11>:     jmp     0x8048294 <_init+24>

```

說明此時還沒有進行進行符號的重定位，不過發現並非 printf，而是 puts(1)。

接著查看 GOT 第三項的內容，剛好是 dl-linux 對應的代碼：

```

(gdb) disassemble 0xb7f33f10
Dump of assembler code for function _dl_runtime_resolve:
0xb7f33f10 <_dl_runtime_resolve+0>:      push    %eax
0xb7f33f11 <_dl_runtime_resolve+1>:      push    %ecx
0xb7f33f12 <_dl_runtime_resolve+2>:      push    %edx

```

可通過 `nm /lib/ld-linux.so.2 | grep _dl_runtime_resolve` 進行確認。

然後查看 GOT 表第二項處的內容，看不出什麼特別的信息，反編譯時提示無法反編譯：

```

(gdb) x/8x 0xb7f3c6d8
0xb7f3c6d8:      0x00000000      0xb7f39c3d      0x08049548      0xb7f3c9b8
0xb7f3c6e8:      0x00000000      0xb7f3c6d8      0x00000000      0xb7f3c9a4

```

在 `*(0xb7f33f10)` 指向的代碼處設置一個斷點，確認它是否被執行：

```

(gdb) break *(0xb7f33f10)
break *(0xb7f33f10)
Breakpoint 3 at 0xb7f3cf10
(gdb) c
Continuing.

Breakpoint 3, 0xb7f3cf10 in _dl_runtime_resolve () from /lib/ld-linux.so.2

```

繼續運行，直到第二次調用 printf：

```

(gdb) c
Continuing.
Hello World

Breakpoint 2, main () at got.c:22
22      printf("got2: 0x%x, got3: 0x%x, old_addr: 0x%x, new_addr: 0x%x\n",

```

再次查看 GOT 表項，發現 GOT 表的最後一項的值應該被修改：

```
(gdb) x/8x 0x8049614
0x8049614 <_GLOBAL_OFFSET_TABLE_>:      0x08049548      0xb7f3c6d8      0xb7f33f10      0x08049624 <_GLOBAL_OFFSET_TABLE_+16>:  0xb7ddb20      0x080482ca      0xb7e1ea20      0x080482ca
```

查看 GOT 表最後一項，發現變成了 puts 函數的代碼，說明進行了符號 puts 的重定位（2）：

```
(gdb) disassemble 0xb7e1ea20
Dump of assembler code for function puts:
0xb7e1ea20 <puts+0>:      push    %ebp
0xb7e1ea21 <puts+1>:      mov     %esp,%ebp
0xb7e1ea23 <puts+3>:      sub     $0x1c,%esp
```

通過演示發現一個問題（1）（2），即本來調用的是 printf，為什麼會進行 puts 的重定位呢？通過 gcc -S 參數編譯生成彙編代碼後發現，gcc 把 printf 替換成了 puts，因此不難理解程序運行過程為什麼對 puts 進行了重定位。

從演示中不難發現，當符號被使用到時才進行重定位。因為通過調試發現在執行 printf 之後，GOT 表項的最後一項才被修改為 printf（確切的說是 puts）的地址。這就是所謂的 lazy mode 動態符號鏈接方式。

除此之外，我們容易發現 GOT 表第三項確實是 ld-linux.so 中的某個函數地址，並且發現在執行 printf 語句之前，先進入了 ld-linux.so 的 _dl_runtime_resolve 函數，而且在它返回之後，GOT 表的最後一項才變為 printf（puts）的地址。

本來打算通過第一個斷點確認第二次調用 printf 時不再需要進行動態符號鏈接的，不過因為 gcc 把第一個替換成了 puts，所以這裡沒有辦法繼續調試。如果想確認這個，你可以通過寫兩個一樣的 printf 語句看看。實際上第一次鏈接以後，GOT 表的第三項已經修改了，當下次再進入過程鏈接表，並執行 jmp *(全局偏移表中某一個地址) 指令時，*(全局偏移表中某一個地址) 已經被修改為了對應符號的實際地址，這樣 jmp 語句會自動跳轉到符號的地址處運行，執行具體的函數代碼，因此無須再進行重定位。

到現在 GOT 表中只剩下第二項還沒有被確認，通過資料 [3] 我們發現，該項指向一個 link_map 類型的數據，是一個鑑別信息，具體作用對我們來說並不是很重要，如果了解，請參考資料 [16]。

下面通過設置 LD_BIND_NOW 再運行一下 got 程序並查看結果，比較它與默認的動態鏈接方式（lazy mode）的異同。

- 設置 LD_BIND_NOW 環境變量的運行結果

```
$ LD_BIND_NOW=1 ./got
Hello World
got2: 0x0, got3: 0x0, old_addr: 0xb7e61a20, new_addr: 0xb7e61a20
```

- 默認情況下的運行結果

```
$ ./got
Hello World
got2: 0xb7f806d8, got3: 0xb7f77f10, old_addr: 0x80482da, new_addr: 0xb7e62a20
```

通過比較容易發現，在非 `lazy mode`（設置 `LD_BIND_NOW` 後）下，程序運行之前符號的地址就已經被確定，即調用 `printf` 之前 `GOT` 表的最後一項已經被確定為了 `printf` 函數對應的地址，即 `0xb7e61a20`，因此在程序運行之後，`GOT` 表的第二項和第三項就保持為 0，因為此時不再需要它們進行符號的重定位了。通過這樣一個比較，就更容易理解 `lazy mode` 的特點了：在用到的時候才解析。

到這裡，符號動態鏈接的細節基本上就已經清楚了。

參考資料

- [Linux 系統中動態鏈接庫的創建與使用](#)
- [Linux 動態鏈接庫高級應用](#)
- [ELF 動態解析符號過程\(修訂版\)](#)
- [如何在 Linux 下調試動態鏈接庫](#)
- [Dissecting shared libraries](#)
- [關於 Linux 和 Unix 動態鏈接庫的安全](#)
- [Linux 系統下解析 ELF 文件 DT_RPATH 後門](#)
- [ELF 文件格式分析](#)
- [緩衝區溢出與注入分析\(第二部分：緩衝區溢出和注入實例\)](#)
- [Gcc 編譯的背後\(第二部分：彙編和鏈接\)](#)
- [程序執行的那一剎那](#)
- [What is Linux-gate.so.1: \[1\], \[2\], \[3\]](#)
- [Linux 下緩衝區溢出攻擊的原理及對策](#)
- [Intel 平臺下 Linux 中 ELF 文件動態鏈接的加載、解析及實例分析 \[part1\]\(#\), \[part2\]\(#\)](#)
- [ELF file format and ABI](#)

緩衝區溢出與注入分析

- [前言](#)
- [進程的內存映像](#)
 - [常用寄存器初識](#)
 - [call, ret 指令的作用分析](#)
 - [什麼是系統調用](#)
 - [什麼是 ELF 文件](#)
 - [程序執行基本過程](#)
 - [Linux 下程序的內存映像](#)
 - [棧在內存中的組織](#)
- [緩衝區溢出](#)
 - [實例分析：字符串複製](#)
 - [緩衝區溢出後果](#)
 - [緩衝區溢出應對策略](#)
 - [如何保護 ebp 不被修改](#)
 - [如何保護 eip 不被修改？](#)
 - [緩衝區溢出檢測](#)
- [緩衝區注入實例](#)
 - [準備：把 C 語言函數轉換為字符串序列](#)
 - [注入：在 C 語言中執行字符串化的代碼](#)
 - [注入原理分析](#)
 - [緩衝區注入與防範](#)
- [後記](#)
- [參考資料](#)

前言

雖然程序加載以及動態符號鏈接都已經很理解了，但是這夥卻被進程的內存映像給“糾纏”住。看著看著就一發不可收拾——很有趣。

下面一起來探究“緩衝區溢出和注入”問題（主要是關心程序的內存映像）。

進程的內存映像

永遠的 `Hello World`，太熟悉了吧，

```
#include <stdio.h>
int main(void)
{
    printf("Hello World\n");
    return 0;
}
```

如果要用內聯彙編 (`inline assembly`) 來寫呢？

```

1  /* shellcode.c */
2  void main()
3  {
4      __asm__ __volatile__("jmp forward;"
5                          "backward:"
6                          "popl    %esi;"
7                          "movl    $4, %eax;"
8                          "movl    $2, %ebx;"
9                          "movl    %esi, %ecx;"
10                         "movl    $12, %edx;"
11                         "int     $0x80;" /* system call 1 */
12                         "movl    $1, %eax;"
13                         "movl    $0, %ebx;"
14                         "int     $0x80;" /* system call 2 */
15                         "forward:"
16                         "call    backward;"
17                         ".string \"Hello World\\n\\n\"");
18  }
```

看起來很複雜，實際上就做了一個事情，往終端上寫了個 `Hello World`。不過這個非常有意思。先簡單分析一下流程：

- 第 4 行指令的作用是跳轉到第 15 行（即 `forward` 標記處），接著執行第 16 行。
- 第 16 行調用 `backward`，跳轉到第 5 行，接著執行 6 到 14 行。
- 第 6 行到第 11 行負責在終端打印出 `Hello World` 字符串（等一下詳細介紹）。
- 第 12 行到第 14 行退出程序（等一下詳細介紹）。

为了更好的理解上面的代碼和後續的分析，先來介紹幾個比較重要的內容。

常用寄存器初識

x86 處理器平臺有三個常用寄存器：程序指令指針、程序堆棧指針與程序基指針：

寄存器	名稱	註釋
EIP	程序指令指針	通常指向下一條指令的位置
ESP	程序堆棧指針	通常指向當前堆棧的當前位置
EBP	程序基指針	通常指向函數使用的堆棧頂端

當然，上面都是擴展的寄存器，用於 32 位系統，對應的 16 系統為 `ip`，`sp`，`bp`。

call, ret 指令的作用分析

- `call` 指令

跳轉到某個位置，並在之前把下一條指令的地址（`EIP`）入棧（為了方便“程序”返回以後能夠接著執行）。這樣的話就有：


```
call backward ==> push eip
                  jmp backward
```

- ret 指令

通常 `call` 指令和 `ret` 是配合使用的，前者壓入跳轉前的下一條指令地址，後者彈出 `call` 指令壓入的那條指令，從而可以在函數調用結束以後接著執行後面的指令。

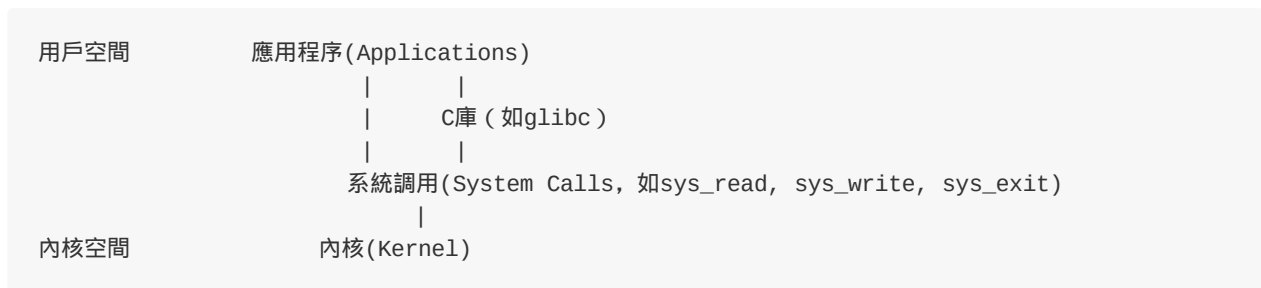
```
ret ==> pop eip
```

通常在函數調用後，還需要恢復 `esp` 和 `ebp`，恢復 `esp` 即恢復當前棧指針，以便釋放調用函數時為存儲函數的局部變量而自動分配的空間；恢復 `ebp` 是從棧中彈出一個數據項（通常函數調用過後的第一條語句就是 `push ebp`），從而恢復當前的函數指針為函數調用者本身。這兩個動作可以通過一條 `leave` 指令完成。

這三個指令對我們後續的解釋會很有幫助。更多關於 Intel 的指令集，請參考：[Intel 386 Manual](#), x86 Assembly Language FAQ: [part1](#), [part2](#), [part3](#).

什麼是系統調用（以 Linux 2.6.21 版本和 x86 平臺為例）

系統調用是用戶和內核之間的接口，用戶如果想寫程序，很多時候直接調用了 C 庫，並沒有關心繫統調用，而實際上 C 庫也是基於系統調用的。這樣應用程序和內核之間就可以通過系統調用聯繫起來。它們分別處於操作系統的用戶空間和內核空間（主要是內存地址空間的隔離）。



系統調用實際上也是一些函數，它們被定義在 `arch/i386/kernel/sys_i386.c`（老的在 `arch/i386/kernel/sys.c`）文件中，並且通過一張系統調用表組織，該表在內核啟動時就已經加載了，這個表的入口在內核源代碼的 `arch/i386/kernel/syscall_table.S` 裡頭（老的在 `arch/i386/kernel/entry.S`）。這樣，如果想添加一個新的系統調用，修改上面兩個內核中的文件，並重新編譯內核就可以。當然，如果要在應用程序中使用它們，還得把它寫到 `include/asm/unistd.h` 中。

如果要在 C 語言中使用某個系統調用，需要包含頭文件 `/usr/include/asm/unistd.h`，裡頭有各個系統調用的聲明以及系統調用號（對應於調用表的入口，即在調用表中的索引，為方便查找調用表而設立的）。如果是自己定義的新系統調用，可能還要在開頭用宏 `_syscall(type, name, type1, name1...)` 來聲明好參數。

如果要在彙編語言中使用，需要用到 `int 0x80` 調用，這個是系統調用的中斷入口。涉及到傳送參數的寄存器有這麼幾個，`eax` 是系統調用號（可以到 `/usr/include/asm-i386/unistd.h` 或者直接到

arch/i386/kernel/syscall_table.S 查到)，其他寄存器如 `ebx`，`ecx`，`edx`，`esi`，`edi` 一次存放系統調用的參數。而系統調用的返回值存放在 `eax` 寄存器中。

下面我們就很容易解釋前面的 `Shellcode.c` 程序流程的 2, 3 兩部分了。因為都用了 `int 0x80` 中斷，所以都用到了系統調用。

第 3 部分很簡單，用到的系統調用號是 1，通過查表（查 `/usr/include/asm-i386/unistd.h` 或 `arch/i386/kernel/syscall_table.S`）可以發現這裡是 `sys_exit` 調用，再從 `/usr/include/unistd.h` 文件看這個系統調用的聲明，發現參數 `ebx` 是程序退出狀態。

第 2 部分比較有趣，而且複雜一點。我們依次來看各個寄存器，首先根據 `eax` 為 4 確定（同樣查表）系統調用為 `sys_write`，而查看它的聲明（從 `/usr/include/unistd.h`），我們找到了參數依次為文件描述符、字符串指針和字符串長度。

- 第一個參數是 `ebx`，正好是 2，即標準錯誤輸出，默認為終端。
- 第二個參數是 `ecx`，而 `ecx` 的內容來自 `esi`，`esi` 來自剛彈出棧的值（見第 6 行 `popl %esi;`），而之前剛好有 `call` 指令引起了最近一次壓棧操作，入棧的內容剛好是 `call` 指令的下一條指令的地址，即 `.string` 所在行的地址，這樣 `ecx` 剛好引用了 `Hello World\\n` 字符串的地址。
- 第三個參數是 `edx`，剛好是 12，即 `Hello World\\n` 字符串的長度（包括一個空字符）。這樣，`Shellcode.c` 的執行流程就很清楚了，第 4, 5, 15, 16 行指令的巧妙之處也就容易理解了（把 `.string` 存放在 `call` 指令之後，並用 `popl` 指令把 `eip` 彈出當作字符串的入口）。

什麼是 ELF 文件

這裡的 ELF 不是“精靈”，而是 Executable and Linking Format 文件，是 Linux 下用來做目標文件、可執行文件和共享庫的一種文件格式，它有專門的標準，例如：[X86 ELF format and ABI](#)，[中文版](#)。

下面簡單描述 ELF 的格式。

ELF 文件主要有三種，分別是：

- 可重定位的目標文件，在編譯時用 `gcc` 的 `-c` 參數時產生。
- 可執行文件，這類文件就是我們後面要討論的可以執行的文件。
- 共享庫，這裡主要是動態共享庫，而靜態共享庫則是可重定位的目標文件通過 `ar` 命令組織的。

ELF 文件的大體結構：

```
ELF Header          #程序頭，有該文件的Magic number(參考man magic), 類型等
Program Header Table #對可執行文件和共享庫有效，它描述下面各個節(section)組成的段
Section1
Section2
Section3
.....
Program Section Table #僅對可重定位目標文件和靜態庫有效，用於描述各個Section的重定位信息等。
```

對於可執行文件，文件最後的 `Program Section Table`（節區表）和一些非重定位的 `Section`，比如 `.comment`，`.note.XXX.debug` 等信息都可以刪除掉，不過如果用 `strip`，`objcopy` 等工具刪除掉以後，就不可恢復了。因為這些信息對程序的運行一般沒有任何用處。

ELF 文件的主要節區 (section) 有 `.data` , `.text` , `.bss` , `.interp` 等, 而主要段 (segment) 有 `LOAD` , `INTERP` 等。它們之間 (節區和段) 的主要對應關係如下:

Section	解釋	實例
<code>.data</code>	初始化的數據	比如 <code>int a=10</code>
<code>.bss</code>	未初始化的數據	比如 <code>char sum[100]</code> ; 這個在程序執行之前, 內核將初始化為 0
<code>.text</code>	程序代碼正文	即可執行指令集
<code>.interp</code>	描述程序需要的解釋器 (動態連接和裝載程序)	存解釋器的全路徑, 如 <code>/lib/ld-linux.so</code>

而程序在執行以後, `.data` , `.bss` , `.text` 等一些節區會被 `Program header table` 映射到 `LOAD` 段, `.interp` 則被映射到了 `INTERP` 段。

對於 ELF 文件的分析, 建議使用 `file` , `size` , `readelf` , `objdump` , `strip` , `objcopy` , `gdb` , `nm` 等工具。

這裡簡單地演示這幾個工具:

```
$ gcc -g -o shellcode shellcode.c #如果要用gdb調試, 編譯時加上-g是必須的
shellcode.c: In function 'main':
shellcode.c:3: warning: return type of 'main' is not 'int'
f$ file shellcode #file命令查看文件類型, 想了解工作原理, 可man magic,man file
shellcode: ELF 32-bit LSB executable, Intel 80386, version 1 (SYSV),
dynamically linked (uses shared libs), not stripped
$ readelf -l shellcode #列出ELF文件前面的program head table, 後面是它描
                        #述了各個段(segment)和節區(section)的關係, 即各個段包含哪些節區。
Elf file type is EXEC (Executable file)
Entry point 0x8048280
There are 7 program headers, starting at offset 52

Program Headers:
  Type           Offset   VirtAddr   PhysAddr   FileSiz MemSiz  Flg Align
  PHDR           0x000034 0x08048034 0x08048034 0x000e0 0x000e0 R E 0x4
  INTERP         0x000114 0x08048114 0x08048114 0x00013 0x00013 R   0x1
      [Requesting program interpreter: /lib/ld-linux.so.2]
  LOAD           0x000000 0x08048000 0x08048000 0x0044c 0x0044c R E 0x1000
  LOAD           0x00044c 0x0804944c 0x0804944c 0x00100 0x00104 RW 0x1000
  DYNAMIC        0x000460 0x08049460 0x08049460 0x000c8 0x000c8 RW 0x4
  NOTE           0x000128 0x08048128 0x08048128 0x00020 0x00020 R   0x4
  GNU_STACK      0x000000 0x00000000 0x00000000 0x00000 0x00000 RW 0x4

Section to Segment mapping:
Segment Sections...
00
01      .interp
02      .interp .note.ABI-tag .hash .dynsym .dynstr .gnu.version .gnu.version_r
      .rel.dyn .rel.plt .init .plt .text .fini .rodata .eh_frame
03      .ctors .dtors .jcr .dynamic .got .got.plt .data .bss
04      .dynamic
05      .note.ABI-tag
06
$ size shellcode #可用size命令查看各個段 ( 對應後面將分析的進程內存映像 ) 的大小
```

```

text    data    bss    dec    hex filename
815     256      4    1075    433 shellcode
$ strip -R .note.ABI-tag shellcode #可用strip來給可執行文件“減肥”，刪除無用信息
$ size shellcode                    #“減肥”後效果“明顯”，對於嵌入式系統應該有很大的作用
text    data    bss    dec    hex filename
783     256      4    1043    413 shellcode
$ objdump -s -j .interp shellcode #這個主要工作是反編譯，不過用來查看各個節區也很厲害

shellcode:      file format elf32-i386

Contents of section .interp:
8048114 2f6c6962 2f6c642d 6c696e75 782e736f  /lib/ld-linux.so
8048124 2e3200                                .2.

```

補充：如果要刪除可執行文件的 Program Section Table，可以用 [A Whirlwind Tutorial on Creating Really Teensy ELF Executables for Linux](#) 一文的作者寫的 `elf kicker` 工具鏈中的 `sstrip` 工具。

程序執行基本過程

在命令行下，敲入程序的名字或者是全路徑，然後按下回車就可以啟動程序，這個具體是怎麼工作的呢？

首先要再認識一下我們的命令行，命令行是內核和用戶之間的接口，它本身也是一個程序。在 Linux 系統啟動以後會為每個終端用戶建立一個進程執行一個 Shell 解釋程序，這個程序解釋並執行用戶輸入的命令，以實現用戶和內核之間的接口。這類解釋程序有哪些呢？目前 Linux 下比較常用的有 `/bin/bash`。那麼該程序接收並執行命令的過程是怎麼樣的呢？

先簡單描述一下這個過程：

- 讀取用戶由鍵盤輸入的命令。
- 分析命令，以命令名作為文件名，並將其它參數改為系統調用 `execve` 內部處理所要求的形式。
- 終端進程調用 `fork` 建立一個子進程。
- 終端進程本身用系統調用 `wait4` 來等待子進程完成（如果是後臺命令，則不等待）。當子進程運行時調用 `execve`，子進程根據文件名（即命令名）到目錄中查找有關文件（這是命令解釋程序構成的文件），將它調入內存，執行這個程序（解釋這條命令）。
- 如果命令末尾有 `&` 號（後臺命令符號），則終端進程不用系統調用 `wait4` 等待，立即發提示符，讓用戶輸入下一個命令，轉 1）。如果命令末尾沒有 `&` 號，則終端進程要一直等待，當子進程（即運行命令的進程）完成處理後終止，向父進程（終端進程）報告，此時終端進程醒來，在做必要的判別等工作後，終端進程發提示符，讓用戶輸入新的命令，重複上述處理過程。

現在用 `strace` 來跟蹤一下程序執行過程中用到的系統調用。

```

$ strace -f -o strace.out test
$ cat strace.out | grep \(.*\)| sed -e "s#[0-9]* \([a-zA-Z0-9_]*\)(.*).*\#1#g"
execve
brk
access
open
fstat64
mmap2
close
open

```

```

read
fstat64
mmap2
mmap2
mmap2
mmap2
close
mmap2
set_thread_area
mprotect
munmap
brk
brk
open
fstat64
mmap2
close
close
close
exit_group

```

相關的系統調用基本體現了上面的執行過程，需要注意的是，裡頭還涉及到內存映射（`mmap2`）等。

下面再羅嗦一些比較有意思的內容，參考《深入理解 Linux 內核》的程序的執行（P681）。

Linux 支持很多不同的可執行文件格式，這些不同的格式是如何解釋的呢？平時我們在命令行下敲入一個命令就完了，也沒有去管這些細節。實際上 Linux 下有一個 `struct linux_binfmt` 結構來管理不同的可執行文件類型，這個結構中有對應的可執行文件的處理函數。大概的過程如下：

- 在用戶態執行了 `execve` 後，引發 `int 0x80` 中斷，進入內核態，執行內核態的相應函數 `do_sys_execve`，該函數又調用 `do_execve` 函數。`do_execve` 函數讀入可執行文件，檢查權限，如果沒問題，繼續讀入可執行文件需要的相關信息（`struct linux_binprm` 描述的）。
- 接著執行 `search_binary_handler`，根據可執行文件的類型（由上一步的最後確定），在 `linux_binfmt` 結構鏈表（`formats`，這個鏈表可以通過 `register_binfmt` 和 `unregister_binfmt` 註冊和刪除某些可執行文件的信息，因此註冊新的可執行文件成為可能，後面再介紹）上查找，找到相應的結構，然後執行相應的 `load_binary` 函數開始加載可執行文件。在該鏈表的最後一個元素總是對解釋腳本（`interpreted script`）的可執行文件格式進行描述的一個對象。這種格式只定義了 `load_binary` 方法，其相應的 `load_script` 函數檢查這種可執行文件是否以兩個 `#!` 字符開始，如果是，這個函數就以另一個可執行文件的路徑名作為參數解釋第一行的其餘部分，並把腳本文件名作為參數傳遞以執行這個腳本（實際上腳本程序把自身的內容當作一個參數傳遞給瞭解釋程序（如 `/bin/bash`），而這個解釋程序通常在腳本文件的開頭用 `#!` 標記，如果沒有標記，那麼默認解釋程序為當前 `SHELL`）。
- 對於 `ELF` 類型文件，其處理函數是 `load_elf_binary`，它先讀入 `ELF` 文件的頭部，根據頭部信息讀入各種數據，再次掃描程序段描述表（`Program Header Table`），找到類型為 `PT_LOAD` 的段（即 `.text`，`.data`，`.bss` 等節區），將其映射（`elf_map`）到內存的固定地址上，如果沒有動態連接器的描述段，把返回的入口地址設置成應用程序入口。完成這個功能的是 `start_thread`，它不啟動一個線程，而只是用來修改了 `pt_regs` 中保存的 `PC` 等寄存器的值，使其指向加載的應用程序的入口。當內核操作結束，返回用戶態時接著就執行應用程序本身了。

- 如果應用程序使用了動態連接庫，內核除了加載指定的可執行文件外，還要把控制權交給動態連接器（`ld-linux.so`）以便處理動態連接的程序。內核搜尋段表（`Program Header Table`），找到標記為 `PT_INTERP` 段中所對應的動態連接器的名稱，並使用 `load_elf_interp` 加載其映像，並把返回的入口地址設置成 `load_elf_interp` 的返回值，即動態鏈接器的入口。當 `execve` 系統調用退出時，動態連接器接著運行，它檢查應用程序對共享鏈接庫的依賴性，並在需要時對其加載，對程序的外部引用進行重定位（具體過程見《進程和進程的基本操作》）。然後把控制權交給應用程序，從 `ELF` 文件頭部中定義的程序進入點（用 `readelf -h` 可以出看到，`Entry point address` 即是）開始執行。（不過對於非 `LIB_BIND_NOW` 的共享庫裝載是在有外部引用請求時才執行的）。

對於內核態的函數調用過程，沒有辦法通過 `strace`（它只能跟蹤到系統調用層）來做的，因此要想跟蹤內核中各個系統調用的執行細節，需要用其他工具。比如可以通過 `Ftrace` 來跟蹤內核具體調用了哪些函數。當然，也可以通過 `ctags/csscope/LXR` 等工具分析內核的源代碼。

Linux 允許自己註冊我們自己定義的可執行格式，主要接口是 `/proc/sys/fs/binfmt_misc/register`，可以往裡頭寫入特定格式的字符串來實現。該字符串格式如下：

```
:name:type:offset:string:mask:interpreter:
```

- `name` 新格式的標示符
- `type` 識別類型（`M` 表示魔數，`E` 表示擴展）
- `offset` 魔數（`magic number`，請參考 `man magic` 和 `man file`）在文件中的啟始偏移量
- `string` 以魔數或者以擴展名匹配的字節序列
- `mask` 用來屏蔽掉 `string` 的一些位
- `interpreter` 程序解釋器的完整路徑名

Linux 下程序的內存映像

Linux 下是如何給進程分配內存（這裡僅討論虛擬內存的分配）的呢？可以從 `/proc/<pid>/maps` 文件中看到個大概。這裡的 `pid` 是進程號。

`/proc` 下有一個文件比較特殊，是 `self`，它鏈接到當前進程的進程號，例如：

```
$ ls /proc/self -l
lrwxrwxrwx 1 root root 64 2000-01-10 18:26 /proc/self -> 11291/
$ ls /proc/self -l
lrwxrwxrwx 1 root root 64 2000-01-10 18:26 /proc/self -> 11292/
```

看到沒？每次都不一樣，這樣我們通過 `cat /proc/self/maps` 就可以看到 `cat` 程序執行時的內存映像了。

```
$ cat -n /proc/self/maps
 1 08048000-0804c000 r-xp 00000000 03:01 273716 /bin/cat
 2 0804c000-0804d000 rw-p 00003000 03:01 273716 /bin/cat
 3 0804d000-0806e000 rw-p 0804d000 00:00 0 [heap]
 4 b7b90000-b7d90000 r--p 00000000 03:01 87528 /usr/lib/locale/locale-archive
 5 b7d90000-b7d91000 rw-p b7d90000 00:00 0
 6 b7d91000-b7ecd000 r-xp 00000000 03:01 466875 /lib/libc-2.5.so
 7 b7ecd000-b7ece000 r--p 0013c000 03:01 466875 /lib/libc-2.5.so
 8 b7ece000-b7ed0000 rw-p 0013d000 03:01 466875 /lib/libc-2.5.so
```

```

 9  b7ed0000-b7ed4000 rw-p b7ed0000 00:00 0
10  b7eeb000-b7f06000 r-xp 00000000 03:01 402817 /lib/ld-2.5.so
11  b7f06000-b7f08000 rw-p 0001b000 03:01 402817 /lib/ld-2.5.so
12  bfbf3000-bfbf8000 rw-p bfbf3000 00:00 0 [stack]
13  fffffe000-ffffff000 r-xp 00000000 00:00 0 [vdso]

```

編號是原文件裡頭沒有的，為了說明方便，用 `-n` 參數加上去的。我們從中可以得到如下信息：

- 第 1, 2 行對應的內存區是我們的程序（包括指令，數據等）
- 第 3 到 12 行對應的內存區是堆棧段，裡頭也映像了程序引用的動態連接庫
- 第 13 行是內核空間

總結一下：

- 前兩部分是用戶空間，可以從 `0x00000000` 到 `0xbfffffff`（在測試的 `2.6.21.5-smp` 上只到 `bfbf8000`），而內核空間從 `0xc0000000` 到 `0xffffffff`，分別是 3G 和 1G，所以對於每一個進程來說，共佔用 4G 的虛擬內存空間
- 從程序本身佔用的內存，到堆棧段（動態獲取內存或者是函數運行過程中用來存儲局部變量、參數的空間，前者是 `heap`，後者是 `stack`），再到內核空間，地址是從低到高的
- 棧頂並非 `0xc0000000` 下的一個固定數值

結合相關資料，可以得到這麼一個比較詳細的進程內存映像表（以 `Linux 2.6.21.5-smp` 為例）：

地址	內核空間	描述
0xC0000000		
	(program file) 程序名	execve 的第一個參數
	(environment) 環境變量	execve 的第三個參數，main 的第三個參數
	(arguments) 參數	execve 的第二個參數，main 的形參
	(stack) 棧	自動變量以及每次函數調用時所需保存的信息都
		存放在此，包括函數返回地址、調用者的
		環境信息等，函數的參數，局部變量都存放在此
	(shared memory) 共享內存	共享內存的大概位置
	...	
	...	
	(heap) 堆	主要在這裡進行動態存儲分配，比如 malloc, new 等。
	...	
	.bss (uninitialized data)	沒有初始化的數據（全局變量哦）
	.data (initialized global data)	已經初始化的全局數據（全局變量）
	.text (Executable Instructions)	通常是可執行指令
0x08048000		
0x00000000		...

光看沒有任何概念，我們用 `gdb` 來看看剛才那個簡單的程序。


```

$ gcc -g -o shellcode shellcode.c    #要用gdb調試，在編譯時需要加-g參數
$ gdb ./shellcode
(gdb) set args arg1 arg2 arg3 arg4  #為了測試，設置幾個參數
(gdb) l                               #瀏覽代碼
1 /* shellcode.c */
2 void main()
3 {
4     __asm__ __volatile__("jmp forward;"
5     "backward:"
6     "popl    %esi;"
7     "movl    $4, %eax;"
8     "movl    $2, %ebx;"
9     "movl    %esi, %ecx;"
10    "movl    $12, %edx;"
(gdb) break 4                        #在彙編入口設置一個斷點，讓程序運行後停到這裡
Breakpoint 1 at 0x8048332: file shellcode.c, line 4.
(gdb) r                             #運行程序
Starting program: /mnt/hda8/Temp/c/program/shellcode arg1 arg2 arg3 arg4

Breakpoint 1, main () at shellcode.c:4
4     __asm__ __volatile__("jmp forward;"
(gdb) print $esp                     #打印當前堆棧指針值，用於查找整個棧的棧頂
$1 = (void *) 0xbffe1584
(gdb) x/100s $esp+4000               #改變後面的4000，不斷往更大的空間找
(gdb) x/1s 0xbffe1fd9               #在 0xbffe1fd9 找到了程序名，這裡是該次運行時的棧頂
0xbffe1fd9:    "/mnt/hda8/Temp/c/program/shellcode"
(gdb) x/10s 0xbffe17b7              #其他環境變量信息
0xbffe17b7:    "CPLUS_INCLUDE_PATH=/usr/lib/qt/include"
0xbffe17de:    "MANPATH=/usr/local/man:/usr/man:/usr/X11R6/man:/usr/lib/java/man:/usr/sha
0xbffe1834:    "HOSTNAME=falcon.lzu.edu.cn"
0xbffe184f:    "TERM=xterm"
0xbffe185a:    "SSH_CLIENT=219.246.50.235 3099 22"
0xbffe187c:    "QTDIR=/usr/lib/qt"
0xbffe188e:    "SSH_TTY=/dev/pts/0"
0xbffe18a1:    "USER=falcon"
...
(gdb) x/5s 0xbffe1780               #一些傳遞給main函數的參數，包括文件名和其他參數
0xbffe1780:    "/mnt/hda8/Temp/c/program/shellcode"
0xbffe17a3:    "arg1"
0xbffe17a8:    "arg2"
0xbffe17ad:    "arg3"
0xbffe17b2:    "arg4"
(gdb) print init                    #打印init函數的地址，這個是/usr/lib/crti.o裡頭的函數，做一些初始化操作
$2 = {<text variable, no debug info>} 0xb7e73d00 <init>
(gdb) print fini                    #也在/usr/lib/crti.o中定義，在程序結束時做一些處理工作
$3 = {<text variable, no debug info>} 0xb7f4a380 <fini>
(gdb) print _start                  #在/usr/lib/crt1.o，這個才是程序的入口，必須的，ld會檢查這個
$4 = {<text variable, no debug info>} 0x8048280 <__libc_start_main@plt+20>
(gdb) print main                    #這裡是我們的main函數
$5 = {void ()} 0x8048324 <main>

```

補充：在進程的內存映像中可能看到諸如 `init`，`fini`，`_start` 等函數（或者是入口），這些東西並不是我們自己寫的啊？為什麼會跑到我們的代碼裡頭呢？實際上這些東西是鏈接的時候 `gcc` 默認給連接進去的，主要用來做一些進程的初始化和終止的動作。更多相關的細節可以參考資料[如何獲取當前進程之靜態影像文件](#)和“The Linux Kernel Primer”，P234，Figure 4.11，如果了解鏈接（ld）的具體過程，可以看

看本節參考《Unix環境高級編程編程》第7章 "Unix進程的環境", P127和P13, [ELF: From The Programmer's Perspective](#), [GNU-ld 連接腳本 Linker Scripts](#)。

上面的操作對堆棧的操作比較少，下面我們用一個例子來演示棧在內存中的情況。

棧在內存中的組織

這一節主要介紹一個函數被調用時，參數是如何傳遞的，局部變量是如何存儲的，它們對應的棧的位置和變化情況，從而加深對棧的理解。在操作時發現和參考資料的結果不太一樣（參考資料中沒有 `edi` 和 `esi` 相關信息，再第二部分的一個小程序裡頭也沒有），可能是 `gcc` 版本的問題或者是它對不同源代碼的處理不同。我的版本是 `4.1.2`（可以通過 `gcc --version` 查看）。

先來一段簡單的程序，這個程序除了做一個加法操作外，還複製了一些字符串。

```
/* testshellcode.c */
#include <stdio.h>      /* printf */
#include <string.h>     /* memset, memcpy */

#define BUF_SIZE 8

#ifndef STR_SRC
# define STR_SRC "AAAAAAA"
#endif

int func(int a, int b, int c)
{
    int sum = 0;
    char buffer[BUF_SIZE];

    sum = a + b + c;

    memset(buffer, '\0', BUF_SIZE);
    memcpy(buffer, STR_SRC, sizeof(STR_SRC)-1);

    return sum;
}

int main()
{
    int sum;

    sum = func(1, 2, 3);

    printf("sum = %d\n", sum);

    return 0;
}
```

上面這個代碼沒有什麼問題，編譯執行一下：

```
$ make testshellcode
cc      testshellcode.c  -o testshellcode
$ ./testshellcode
```

```
sum = 6
```

下面調試一下，看看在調用 `func` 後的棧的內容。

```
$ gcc -g -o testshellcode testshellcode.c #為了調試，需要在編譯時加-g選項
$ gdb ./testshellcode #啟動gdb調試
...
(gdb) set logging on #如果要記錄調試過程中的信息，可以把日誌記錄功能打開
Copying output to gdb.txt.
(gdb) l main #列出源代碼
20
21         return sum;
22     }
23
24     int main()
25     {
26         int sum;
27
28         sum = func(1, 2, 3);
29
(gdb) break 28 #在調用func函數之前讓程序停一下，以便記錄當時的ebp(基指針)
Breakpoint 1 at 0x80483ac: file testshellcode.c, line 28.
(gdb) break func #設置斷點在函數入口，以便逐步記錄棧信息
Breakpoint 2 at 0x804835c: file testshellcode.c, line 13.
(gdb) disassemble main #反編譯main函數，以便記錄調用func後的下一條指令地址
Dump of assembler code for function main:
0x0804839b <main+0>:    lea    0x4(%esp),%ecx
0x0804839f <main+4>:    and    $0xffffffff0,%esp
0x080483a2 <main+7>:    pushl  0xffffffffc(%ecx)
0x080483a5 <main+10>:   push   %ebp
0x080483a6 <main+11>:   mov    %esp,%ebp
0x080483a8 <main+13>:   push   %ecx
0x080483a9 <main+14>:   sub    $0x14,%esp
0x080483ac <main+17>:   push   $0x3
0x080483ae <main+19>:   push   $0x2
0x080483b0 <main+21>:   push   $0x1
0x080483b2 <main+23>:   call   0x8048354 <func>
0x080483b7 <main+28>:   add    $0xc,%esp
0x080483ba <main+31>:   mov    %eax,0xffffffff8(%ebp)
0x080483bd <main+34>:   sub    $0x8,%esp
0x080483c0 <main+37>:   pushl  0xffffffff8(%ebp)
0x080483c3 <main+40>:   push   $0x80484c0
0x080483c8 <main+45>:   call   0x80482a0 <printf@plt>
0x080483cd <main+50>:   add    $0x10,%esp
0x080483d0 <main+53>:   mov    $0x0,%eax
0x080483d5 <main+58>:   mov    0xffffffffc(%ebp),%ecx
0x080483d8 <main+61>:   leave
0x080483d9 <main+62>:   lea    0xffffffffc(%ecx),%esp
0x080483dc <main+65>:   ret
End of assembler dump.
(gdb) r #運行程序
Starting program: /mnt/hda8/Temp/c/program/testshellcode

Breakpoint 1, main () at testshellcode.c:28
28         sum = func(1, 2, 3);
(gdb) print $ebp #打印調用func函數之前的地址，即Previous frame pointer。
$1 = (void *) 0xbf84fdd8
```

```
(gdb) n #執行call指令並跳轉到func函數的入口

Breakpoint 2, func (a=1, b=2, c=3) at testshellcode.c:13
13      int sum = 0;
(gdb) n
16      sum = a + b + c;
(gdb) x/11x $esp #打印當前棧的內容，可以看出，地址從低到高，注意標記有藍色和紅色的值
#它們分別是前一個棧基地址(ebp)和call調用之後的下一條指令的指針(eip)
0xbf84fd94: 0x00000000 0x00000000 0x080482e0 0x00000000
0xbf84fda4: 0xb7f2bce0 0x00000000 0xbf84fdd8 0x080483b7
0xbf84fdb4: 0x00000001 0x00000002 0x00000003
(gdb) n #執行sum = a + b + c, 後，比較棧內容第一行，第4列，由0變為6
18      memset(buffer, '\0', BUF_SIZE);
(gdb) x/11x $esp
0xbf84fd94: 0x00000000 0x00000000 0x080482e0 0x00000006
0xbf84fda4: 0xb7f2bce0 0x00000000 0xbf84fdd8 0x080483b7
0xbf84fdb4: 0x00000001 0x00000002 0x00000003
(gdb) n
19      memcpy(buffer, STR_SRC, sizeof(STR_SRC)-1);
(gdb) x/11x $esp #緩衝區初始化以後變成了0
0xbf84fd94: 0x00000000 0x00000000 0x00000000 0x00000000
0xbf84fda4: 0xb7f2bce0 0x00000000 0xbf84fdd8 0x080483b7
0xbf84fdb4: 0x00000001 0x00000002 0x00000003
(gdb) n
21      return sum;
(gdb) x/11x $esp #進行copy以後，這兩列的值變了，大小剛好是7個字節，最後一個字節為'\0'
0xbf84fd94: 0x00000000 0x41414141 0x00414141 0x00000006
0xbf84fda4: 0xb7f2bce0 0x00000000 0xbf84fdd8 0x080483b7
0xbf84fdb4: 0x00000001 0x00000002 0x00000003
(gdb) c
Continuing.
sum = 6

Program exited normally.
(gdb) quit
```

從上面的操作過程，我們可以得出大概的棧分佈(func 函數結束之前)如下：

地址	值(hex)	符號或者寄存器	註釋
低地址			棧頂方向
0xbf84fd98	0x41414141	buf[0]	可以看出little endian(小端，重要的數據在前面)
0xbf84fd9c	0x00414141	buf[1]	
0xbf84fda0	0x00000006	sum	可見這上面都是func函數裡頭的局部變量
0xbf84fda4	0xb7f2bce0	esi	源索引指針，可以通過產生中間代碼查看，貌似沒什麼作用
0xbf84fda8	0x00000000	edi	目的索引指針
0xbf84fdac	0xbf84fdd8	ebp	調用func之前的棧的基地址，以便調用函數結束之後恢復
0xbf84fdb0	0x080483b7	eip	調用func之前的指令指針，以便調用函數結束之後繼續執行
0xbf84fdb4	0x00000001	a	第一個參數
0xbf84fdb8	0x00000002	b	第二個參數
0xbf84fdbc	0x00000003	c	第三個參數，可見參數是從最後一個開始壓棧的
高地址			棧底方向

先說明一下 `edi` 和 `esi` 的由來（在上面的調試過程中我們並沒有看到），是通過產生中間彙編代碼分析得出的。

```
$ gcc -S testshellcode.c
```

在產生的 `testShellcode.s` 代碼裡頭的 `func` 部分看到 `push ebp` 之後就 `push` 了 `edi` 和 `esi`。但是搜索了一下代碼，發現就這個函數裡頭引用了這兩個寄存器，所以保存它們沒什麼用，刪除以後編譯產生目標代碼後證明是沒用的。

```
$ cat testshellcode.s
...
func:
    pushl    %ebp
    movl     %esp, %ebp
    pushl    %edi
    pushl    %esi
    ...
    popl     %esi
    popl     %edi
    popl     %ebp
    ...
```

下面就不管這兩部分（`edi` 和 `esi`）了，主要來分析和函數相關的這幾部分在棧內的分佈：

- 函數局部變量，在靠近棧頂一端
- 調用函數之前的棧的基地址（`ebp`，Previous Frame Pointer），在中間靠近棧頂方向
- 調用函數指令的下一條指令地址（`eip`），在中間靠近棧底的方向
- 函數參數，在靠近棧底的一端，最後一個參數最先入棧

到這裡，函數調用時的相關內容在棧內的分佈就比較清楚了，在具體分析緩衝區溢出問題之前，我們再來看一個和函數關係很大的問題，即函數返回值的存儲問題：函數的返回值存放在寄存器 `eax` 中。

先來看這段代碼：

```
/**
 * test_return.c -- the return of a function is stored in register eax
 */

#include <stdio.h>

int func()
{
    __asm__ ("movl $1, %eax");
}

int main()
{
    printf("the return of func: %d\n", func());

    return 0;
}
```

```
}
```

編譯運行後，可以看到返回值為 1，剛好是我們在 `func` 函數中 `mov` 到 `eax` 中的“立即數” 1，因此很容易理解返回值存儲在 `eax` 中的事實，如果還有疑慮，可以再看看彙編代碼。在函數返回之後，`eax` 中的值當作了 `printf` 的參數壓入了棧中，而在源代碼中我們正是把 `func` 的結果作為 `printf` 的第二個參數的。

```
$ make test_return
cc      test_return.c  -o test_return
$ ./test_return
the return of func: 1
$ gcc -S test_return.c
$ cat test_return.s
...
    call    func
    subl    $8, %esp
    pushl   %eax          #printf的第二個參數，把func的返回值壓入了棧底
    pushl   $.LC0         #printf的第一個參數the return of func: %d\n
    call    printf
...
```

對於系統調用，返回值也存儲在 `eax` 寄存器中。

緩衝區溢出

實例分析：字符串複製

先來看一段簡短的代碼。

```
/* testshellcode.c */
#include <stdio.h>      /* printf */
#include <string.h>     /* memset, memcpy */

#define BUF_SIZE 8

#ifdef STR1
# define STR_SRC "AAAAAA\0\1\0\0\0"
#endif

#ifdef STR_SRC
# define STR_SRC "AAAAAA"
#endif

int func(int a, int b, int c)
{
    int sum = 0;
    char buffer[BUF_SIZE];

    sum = a + b + c;

    memset(buffer, '\0', BUF_SIZE);
    memcpy(buffer, STR_SRC, sizeof(STR_SRC)-1);
```

```

        return sum;
    }

    int main()
    {
        int sum;

        sum = func(1, 2, 3);

        printf("sum = %d\n", sum);

        return 0;
    }

```

編譯一下看看結果：

```

$ gcc -DSTR1 -o testshellcode testshellcode.c #通過-D定義宏STR1，從而採用第一個STR_SRC的值
$ ./testshellcode
sum = 1

```

不知道你有沒有發現異常呢？上面用紅色標記的地方，本來 `sum` 為 `1+2+3` 即 6，但是實際返回的竟然是 1。到底是什麼原因呢？大家應該有所瞭解了，因為我們在複製字符串 `AAAAAAA\0\1\0\0\0` 到 `buf` 的時候超出 `buf` 本來的大小。`buf` 本來的大小是 `BUF_SIZE`，8 個字節，而我們要複製的內容是 12 個字節，所以超出了四個字節。根據第一小節的分析，我們用棧的變化情況來表示一下這個複製過程（即執行 `memcpy` 的過程）。

```

memcpy(buffer, STR_SRC, sizeof(STR_SRC)-1);

( 低地址 )
複製之前      ==> 複製之後
0x00000000      0x41414141      #char buf[8]
0x00000000      0x00414141
0x00000006      0x00000001      #int sum
( 高地址 )

```

下面通過 `gdb` 調試來確認一下(只摘錄了一些片斷)。

```

$ gcc -DSTR1 -g -o testshellcode testshellcode.c
$ gdb ./testshellcode
...
(gdb) l
21
22         memset(buffer, '\0', BUF_SIZE);
23         memcpy(buffer, STR_SRC, sizeof(STR_SRC)-1);
24
25         return sum;
...
(gdb) break 23
Breakpoint 1 at 0x804837f: file testshellcode.c, line 23.
(gdb) break 25
Breakpoint 2 at 0x8048393: file testshellcode.c, line 25.

```

```
(gdb) r
Starting program: /mnt/hda8/Temp/c/program/testshellcode

Breakpoint 1, func (a=1, b=2, c=3) at testshellcode.c:23
23      memcpy(buffer, STR_SRC, sizeof(STR_SRC)-1);
(gdb) x/3x $esp+4
0xbfec6bd8:    0x00000000      0x00000000      0x00000006
(gdb) n

Breakpoint 2, func (a=1, b=2, c=3) at testshellcode.c:25
25      return sum;
(gdb) x/3x $esp+4
0xbfec6bd8:    0x41414141      0x00414141      0x00000001
```

可以看出，因為 C 語言沒有對數組的邊界進行限制。我們可以往數組中存入預定義長度的字符串，從而導致緩衝區溢出。

緩衝區溢出後果

溢出之後的問題是導致覆蓋棧的其他內容，從而可能改變程序原來的行為。

如果這類問題被“黑客”利用那將產生非常可怕的後果，小則讓非法用戶獲取了系統權限，把你的服務器當成“殭屍”，用來對其他機器進行攻擊，嚴重的則可能被人刪除數據（所以備份很重要）。即使不被黑客利用，這類問題如果放在醫療領域，那將非常危險，可能那個被覆蓋的數字剛好是用來控制治療癌症的輻射量的，一旦出錯，那可能導致置人死地，當然，如果在航天領域，那可能就是好多個 0 的 `money` 甚至航天員的損失，呵呵，“緩衝區溢出，後果很嚴重！”

緩衝區溢出應對策略

那這個怎麼辦呢？貌似[Linux下緩衝區溢出攻擊的原理及對策](#)提到有一個 `libsafe` 庫，可以至少用來檢測程序中出現的類似超出數組邊界的問題。對於上面那個具體問題，為了保護 `sum` 不被修改，有一個小技巧，可以讓求和操作在字符串複製操作之後來做，以便求和操作把溢出的部分給重寫。這個呆夥在下面一塊看效果吧。繼續看看緩衝區的溢出吧。

先來看看這個代碼，還是 `testShellcode.c` 的改進。

```
/* testshellcode.c */
#include <stdio.h>      /* printf */
#include <string.h> /* memset, memcpy */
#define BUF_SIZE 8

#ifdef STR1
# define STR_SRC "AAAAAAAa\1\0\0\0"
#endif
#ifdef STR2
# define STR_SRC "AAAAAAAa\1\0\0\0BBBBBBBB"
#endif
#ifdef STR3
# define STR_SRC "AAAAAAAa\1\0\0\0BBBBBBBBCCCC"
#endif
#ifdef STR4
# define STR_SRC "AAAAAAAa\1\0\0\0BBBBBBBBCCCCDDDD"
```

```

#endif

#ifndef STR_SRC
# define STR_SRC "AAAAAAA"
#endif

int func(int a, int b, int c)
{
    int sum = 0;
    char buffer[BUF_SIZE] = "";

    memset(buffer, '\0', BUF_SIZE);
    memcpy(buffer, STR_SRC, sizeof(STR_SRC)-1);

    sum = a + b + c;    //把求和操作放在複製操作之後可以在一定情況下“保護”求和結果

    return sum;
}

int main()
{
    int sum;

    sum = func(1, 2, 3);

    printf("sum = %d\n", sum);

    return 0;
}

```

看看運行情況：

```

$ gcc -D STR2 -o testshellcode testshellcode.c    #再多複製8個字節，結果和STR1時一樣
                                                    #原因是edi,esi這兩個沒什麼用的，覆蓋了也沒關係
$ ./testshellcode                                #看到沒？這種情況下，讓整數操作在字符串複製之後做可以“保護”整數結果
sum = 6
$ gcc -D STR3 -o testshellcode testshellcode.c    #再多複製4個字節，現在就會把ebp給覆蓋
                                                    #了，這樣當main函數再要用ebp訪問數據
                                                    #時就會出現訪問非法內存而導致段錯誤。

$ ./testshellcode
Segmentation fault

```

如果感興趣，自己還可以用gdb類似之前一樣來查看複製字符串以後棧的變化情況。

如何保護 **ebp** 不被修改

下面來做一個比較有趣的事情：如何設法保護我們的 **ebp** 不被修改。

首先要明確 **ebp** 這個寄存器的作用和“行為”，它是棧基地址，並且發現在調用任何一個函數時，這個 **ebp** 總是在第一條指令被壓入棧中，並在最後一條指令（**ret**）之前被彈出。類似這樣：

```

func:                                #函數
    pushl %ebp                       #第一條指令

```



```
...
popl %ebp          #倒數第二條指令
ret
```

還記得之前（第一部分）提到的函數的返回值是存儲在 `eax` 寄存器中的麼？如果我們在一個函數中僅僅做放這兩條指令：

```
popl %eax
pushl %eax
```

那不就剛好有：

```
func:                #函數
    pushl %ebp        #第一條指令
    popl %eax         #把剛壓入棧中的ebp彈出存放到eax中
    pushl %eax        #又把ebp壓入棧
    popl %ebp         #倒數第二條指令
    ret
```

這樣我們沒有改變棧的狀態，卻獲得了 `ebp` 的值，如果在調用任何一個函數之前，獲取這個 `ebp`，並且在任何一條字符串複製語句（可能導致緩衝區溢出的語句）之後重新設置一下 `ebp` 的值，那麼就可以保護 `ebp` 啦。具體怎麼實現呢？看這個代碼。

```
/* testshellcode.c */
#include <stdio.h>      /* printf */
#include <string.h> /* memset, memcpy */
#define BUF_SIZE 8

#ifdef STR1
# define STR_SRC "AAAAAAAa\1\0\0\0"
#endif
#ifdef STR2
# define STR_SRC "AAAAAAAa\1\0\0\0BBBBBBBB"
#endif
#ifdef STR3
# define STR_SRC "AAAAAAAa\1\0\0\0BBBBBBBBCCCC"
#endif
#ifdef STR4
# define STR_SRC "AAAAAAAa\1\0\0\0BBBBBBBBCCCCDDDD"
#endif

#ifndef STR_SRC
# define STR_SRC "AAAAAAA"
#endif

unsigned long get_ebp()
{
    __asm__ ("popl %eax;"
            "pushl %eax;");
}

int func(int a, int b, int c, unsigned long ebp)
```

```

{
    int sum = 0;
    char buffer[BUF_SIZE] = "";

    sum = a + b + c;
    memset(buffer, '\0', BUF_SIZE);
    memcpy(buffer, STR_SRC, sizeof(STR_SRC)-1);
    *(unsigned long *)(buffer+20) = ebp;
    return sum;
}

int main()
{
    int sum, ebp;

    ebp = get_ebp();
    sum = func(1, 2, 3, ebp);

    printf("sum = %d\n", sum);

    return 0;
}

```

這段代碼和之前的代碼的不同有：

- 給 `func` 函數增加了一個參數 `ebp`，（其實可以用全局變量替代的）
- 利用了剛介紹的原理定義了一個函數 `get_ebp` 以便獲取老的 `ebp`
- 在 `main` 函數中調用 `func` 之前調用了 `get_ebp`，並把它作為 `func` 的最後一個參數
- 在 `func` 函數中調用 `memcpy` 函數（可能發生緩衝區溢出的地方）之後添加了一條恢復設置 `ebp` 的語句，這條語句先把 `buffer+20` 這個地址（存放 `ebp` 的地址，你可以類似第一部分提到的用 `gdb` 來查看）強制轉換為指向一個 `unsigned long` 型的整數（4 個字節），然後把它指向的內容修改為老的 `ebp`。

看看效果：

```

$ gcc -D STR3 -o testshellcode testshellcode.c
$ ./testshellcode          #現在沒有段錯誤了吧，因為ebp得到了“保護”
sum = 6

```

如何保護 `eip` 不被修改？

如果我們複製更多的字節過去了，比如再多複製四個字節進去，那麼 `eip` 就被覆蓋了。

```

$ gcc -D STR4 -o testshellcode testshellcode.c
$ ./testshellcode
Segmentation fault

```

同樣會出現段錯誤，因為下一條指令的位置都被改寫了，`func` 返回後都不知道要訪問哪個“非法”地址啦。呵呵，如果是一個合法地址呢？

如果在緩衝區溢出時，`eip` 被覆蓋了，並且被修改為了一條合法地址，那麼問題就非常“有趣”了。如果這個地址剛好是調用`func`的那個地址，那麼整個程序就成了死循環，如果這個地址指向的位置剛好有一段關鍵代碼，那麼系統正在運行的所有服務都將被關掉，如果那個地方是一段更惡意的代碼，那就？你可以盡情想像哦。如果是黑客故意利用這個，那麼那些代碼貌似就叫做`shellcode`了。

有沒有保護 `eip` 的辦法呢？呵呵，應該是有的吧。不知道 `gas` 有沒有類似 `masm` 彙編器中 `offset` 的偽操作指令（查找了一下，貌似沒有），如果有的話在函數調用之前設置一個標號，在後面某個位置獲取，再加上一個可能的偏移（包括 `call` 指令的長度和一些 `push` 指令等），應該可以算出來，不過貌似比較麻煩（或許你靈感大作，找到好辦法了！），這裡直接通過 `gdb` 反彙編求得它相對 `main` 的偏移算出來得了。求出來以後用它來“保護”棧中的值。

看看這個代碼：

```
/* testshellcode.c */
#include <stdio.h>      /* printf */
#include <string.h> /* memset, memcpy */
#define BUF_SIZE 8

#ifdef STR1
# define STR_SRC "AAAAAAAa\1\0\0\0"
#endif
#ifdef STR2
# define STR_SRC "AAAAAAAa\1\0\0\0BBBBBBBB"
#endif
#ifdef STR3
# define STR_SRC "AAAAAAAa\1\0\0\0BBBBBBBBCCCC"
#endif
#ifdef STR4
# define STR_SRC "AAAAAAAa\1\0\0\0BBBBBBBBCCCCDDDD"
#endif

#ifndef STR_SRC
# define STR_SRC "AAAAAAA"
#endif

int main();
#define OFFSET 40

unsigned long get_ebp()
{
    __asm__ ("popl %eax;"
            "pushl %eax;");
}

int func(int a, int b, int c, unsigned long ebp)
{
    int sum = 0;
    char buffer[BUF_SIZE] = "";

    memset(buffer, '\0', BUF_SIZE);
    memcpy(buffer, STR_SRC, sizeof(STR_SRC)-1);

    sum = a + b + c;

    *(unsigned long *)(buffer+20) = ebp;
```

```

        *(unsigned long *)(buffer+24) = (unsigned long)main+OFFSET;
        return sum;
    }

    int main()
    {
        int sum, ebp;

        ebp = get_ebp();
        sum = func(1, 2, 3, ebp);

        printf("sum = %d\n", sum);

        return 0;
    }

```

看看效果：

```

$ gcc -D STR4 -o testshellcode testshellcode.c
$ ./testshellcode
sum = 6

```

這樣，EIP 也得到了“保護”（這個方法很糟糕的，呵呵）。

類似地，如果再多複製一些內容呢？那麼棧後面的內容都將被覆蓋，即傳遞給 func 函數的參數都將被覆蓋，因此上面的方法，包括所謂的對 sum 和 ebp 等值的保護都沒有任何意義了（如果再對後面的參數進行進一步的保護呢？或許有點意義，呵呵）。在這裡，之所以提出類似這樣的保護方法，實際上只是為了討論一些有趣的細節並加深對緩衝區溢出這一問題的理解（或許有一些實際的價值哦，算是拋磚引玉吧）。

緩衝區溢出檢測

要確實解決這類問題，從主觀上講，還得程序員來做相關的工作，比如限制將要複製的字符串的長度，保證它不超過當初申請的緩衝區的大小。

例如，在上面的代碼中，我們在 memcpy 之前，可以加入一個判斷，並且可以對緩衝區溢出進行很好的檢查。如果能夠設計一些比較好的測試實例把這些判斷覆蓋到，那麼相關的問題就可以得到比較不錯的檢查了。

```

/* testshellcode.c */
#include <stdio.h>          /* printf */
#include <string.h> /* memset, memcpy */
#include <stdlib.h>         /* exit */
#define BUF_SIZE 8

#ifdef STR4
# define STR_SRC "AAAAAAAa\1\0\0\0BBBBBBBBCCCCDDDD"
#endif

#ifndef STR_SRC
# define STR_SRC "AAAAAAA"

```

```

#endif

int func(int a, int b, int c)
{
    int sum = 0;
    char buffer[BUF_SIZE] = "";

    memset(buffer, '\0', BUF_SIZE);
    if ( sizeof(STR_SRC)-1 > BUF_SIZE ) {
        printf("buffer overflow!\n");
        exit(-1);
    }
    memcpy(buffer, STR_SRC, sizeof(STR_SRC)-1);

    sum = a + b + c;

    return sum;
}

int main()
{
    int sum;

    sum = func(1, 2, 3);

    printf("sum = %d\n", sum);

    return 0;
}

```

現在的效果如下：

```

$ gcc -DSTR4 -g -o testshellcode testshellcode.c
$ ./testshellcode      #如果存在溢出，那麼就會得到阻止並退出，從而阻止可能的破壞
buffer overflow!
$ gcc -g -o testshellcode testshellcode.c
$ ./testshellcode
sum = 6

```

當然，如果能夠在 C 標準裡頭加入對數組操作的限制可能會更好，或者在編譯器中擴展對可能引起緩衝區溢出的語法檢查。

緩衝區注入實例

最後給出一個利用上述緩衝區溢出來進行緩衝區注入的例子。也就是通過往某個緩衝區注入一些代碼，並把eip修改為這些代碼的入口從而達到破壞目標程序行為的目的。

這個例子來自[Linux 下緩衝區溢出攻擊的原理及對策](#)，這裡主要利用上面介紹的知識對它進行了比較詳細的分析。

準備：把 C 語言函數轉換為字符串序列

首先回到第一部分，看看那個 `Shellcode.c` 程序。我們想獲取它的彙編代碼，並以十六進制字節的形式輸出，以便把這些指令當字符串存放起來，從而作為緩衝區注入時的輸入字符串。下面通過 `gdb` 獲取這些內容。

```
$ gcc -g -o shellcode shellcode.c
$ gdb ./shellcode
GNU gdb 6.6
...
(gdb) disassemble main
Dump of assembler code for function main:
...
0x08048331 <main+13>:  push    %ecx
0x08048332 <main+14>:  jmp     0x8048354 <forward>
0x08048334 <main+16>:  pop     %esi
0x08048335 <main+17>:  mov     $0x4,%eax
0x0804833a <main+22>:  mov     $0x2,%ebx
0x0804833f <main+27>:  mov     %esi,%ecx
0x08048341 <main+29>:  mov     $0xc,%edx
0x08048346 <main+34>:  int     $0x80
0x08048348 <main+36>:  mov     $0x1,%eax
0x0804834d <main+41>:  mov     $0x0,%ebx
0x08048352 <main+46>:  int     $0x80
0x08048354 <forward+0>: call    0x8048334 <main+16>
0x08048359 <forward+5>: dec     %eax
0x0804835a <forward+6>: gs
0x0804835b <forward+7>: insb    (%dx),%es:(%edi)
0x0804835c <forward+8>: insb    (%dx),%es:(%edi)
0x0804835d <forward+9>: outsl   %ds:(%esi),(%dx)
0x0804835e <forward+10>: and     %dl,0x6f(%edi)
0x08048361 <forward+13>: jb      0x80483cf <__libc_csu_init+79>
0x08048363 <forward+15>: or      %fs:(%eax),%al
...
```

End of assembler dump.

(gdb) set logging on #開啟日誌功能，記錄操作結果

Copying output to gdb.txt.

(gdb) x/52bx main+14 #以十六進制單字節（字符）方式打印出shellcode的核心代碼

```
0x8048332 <main+14>:  0xeb  0x20  0x5e  0xb8  0x04  0x00  0x00  0x00
0x804833a <main+22>:  0xbb  0x02  0x00  0x00  0x00  0x89  0xf1  0xba
0x8048342 <main+30>:  0x0c  0x00  0x00  0x00  0xcd  0x80  0xb8  0x01
0x804834a <main+38>:  0x00  0x00  0x00  0xbb  0x00  0x00  0x00  0x00
0x8048352 <main+46>:  0xcd  0x80  0xe8  0xdb  0xff  0xff  0xff  0x48
0x804835a <forward+6>: 0x65  0x6c  0x6c  0x6f  0x20  0x57  0x6f  0x72
0x8048362 <forward+14>: 0x6c  0x64  0x0a  0x00
```

(gdb) quit

\$ cat gdb.txt | sed -e "s/^.*://g;s/\t/\\n/g;s/^\"/\"/g;s/\"/\$\"/g" #把日誌裡頭的內容處理一下，得：

```
"\xeb\x20\x5e\xb8\x04\x00\x00\x00"
"\xbb\x02\x00\x00\x00\x89\xf1\xba"
"\x0c\x00\x00\x00\xcd\x80\xb8\x01"
"\x00\x00\x00\xbb\x00\x00\x00\x00"
"\xcd\x80\xe8\xdb\xff\xff\xff\x48"
"\x65\x6c\x6c\x6f\x20\x57\x6f\x72"
"\x6c\x64\x0a\x00"
```

注入：在 C 語言中執行字符串化的代碼

得到上面的字符串以後我們就可以設計一段下面的代碼啦。

```
/* testshellcode.c */
char shellcode[]="\xeb\x20\x5e\xb8\x04\x00\x00\x00"
"\xbb\x02\x00\x00\x00\x89\xf1\xba"
"\x0c\x00\x00\x00\xcd\x80\xb8\x01"
"\x00\x00\x00\xbb\x00\x00\x00\x00"
"\xcd\x80\xe8\xdb\xff\xff\xff\x48"
"\x65\x6c\x6c\x6f\x20\x57\x6f\x72"
"\x6c\x64\x0a\x00";

void callshellcode(void)
{
    int *ret;
    ret = (int *)&ret + 2;
    (*ret) = (int)shellcode;
}

int main()
{
    callshellcode();

    return 0;
}
```

運行看看，

```
$ gcc -o testshellcode testshellcode.c
$ ./testshellcode
Hello World
```

竟然打印出了 Hello World，實際上，如果只是為了讓 Shellcode 執行，有更簡單的辦法，直接把 Shellcode 這個字符串入口強制轉換為一個函數入口，並調用就可以，具體見這段代碼。

```
char shellcode[]="\xeb\x20\x5e\xb8\x04\x00\x00\x00"
"\xbb\x02\x00\x00\x00\x89\xf1\xba"
"\x0c\x00\x00\x00\xcd\x80\xb8\x01"
"\x00\x00\x00\xbb\x00\x00\x00\x00"
"\xcd\x80\xe8\xdb\xff\xff\xff\x48"
"\x65\x6c\x6c\x6f\x20\x57\x6f\x72"
"\x6c\x64\x0a\x00";

typedef void (* func)();          //定義一個指向函數的指針func，而函數的返回值和參數均為void

int main()
{
    (* (func)shellcode)();

    return 0;
}
```

注入原理分析

這裡不那樣做，為什麼也能夠執行到 `Shellcode` 呢？仔細分析一下 `callShellcode` 裡頭的代碼就可以得到原因了。

```
int *ret;
```

這裡定義了一個指向整數的指針，`ret` 佔用 4 個字節（可以用 `sizeof(int *)` 算出）。

```
ret = (int *)&ret + 2;
```

這裡把 `ret` 修改為它本身所在的地址再加上兩個單位。首先需要求出 `ret` 本身所在的位置，因為 `ret` 是函數的一個局部變量，它在棧中偏棧頂的地方。然後呢？再增加兩個單位，這個單位是 `sizeof(int)`，即 4 個字節。這樣，新的 `ret` 就是 `ret` 所在的位置加上 8 個字節，即往棧底方向偏移 8 個字節的位置。對於我們之前分析的 `Shellcode`，那裡應該是 `edi`，但實際上這裡並不是 `edi`，可能是 `gcc` 在編譯程序時有不同的處理，這裡實際上剛好是 `eip`，即執行這條語句之後 `ret` 的值變成了 `eip` 所在的位置。

```
(*ret) = (int)shellcode;
```

由於之前 `ret` 已經被修改為了 `eip` 所在的位置，這樣對 `(*ret)` 賦值就會修改 `eip` 的值，即下一條指令的地址，這裡把 `eip` 修改為了 `Shellcode` 的入口。因此，當函數返回時直接去執行 `Shellcode` 裡頭的代碼，並打印了 `Hello World`。

用 `gdb` 調試一下看看相關變量的值的情況。這裡主要關心 `ret` 本身。`ret` 本身是一個地址，首先它所在的位置變成了 `EIP` 所在的位置（把它自己所在的位置加上 `2*4` 以後賦於自己），然後，`EIP` 又指向了 `Shellcode` 處的代碼。

```
$ gcc -g -o testshellcode testshellcode.c
$ gdb ./testshellcode
...
(gdb) l
8      void callshellcode(void)
9      {
10         int *ret;
11         ret = (int *)&ret + 2;
12         (*ret) = (int)shellcode;
13     }
14
15     int main()
16     {
17         callshellcode();
(gdb) break 17
Breakpoint 1 at 0x804834d: file testshell.c, line 17.
(gdb) break 11
Breakpoint 2 at 0x804832a: file testshell.c, line 11.
(gdb) break 12
Breakpoint 3 at 0x8048333: file testshell.c, line 12.
(gdb) break 13
Breakpoint 4 at 0x804833d: file testshell.c, line 13.
```



```

(gdb) r
Starting program: /mnt/hda8/Temp/c/program/testshell

Breakpoint 1, main () at testshell.c:17
17      callshellcode();
(gdb) print $ebp      #打印ebp寄存器裡的值
$1 = (void *) 0xbfcfd2c8
(gdb) disassemble main
...
0x0804834d <main+14>:  call    0x8048324 <callshellcode>
0x08048352 <main+19>:  mov     $0x0,%eax
...
(gdb) n

Breakpoint 2, callshellcode () at testshell.c:11
11      ret = (int *)&ret + 2;
(gdb) x/6x $esp
0xbfcfd2ac:  0x08048389      0xb7f4eff4      0xbfcfd36c      0xbfcfd2d8
0xbfcfd2bc:  0xbfcfd2c8      0x08048352
(gdb) print &ret      #分別打印出ret所在的地址和ret的值，剛好在ebp之上，我們發現這裡並沒有
                        #之前的testshellcode代碼中的edi和esi，可能是gcc在彙編的時候有不同處理。
$2 = (int **) 0xbfcfd2b8
(gdb) print ret
$3 = (int *) 0xbfcfd2d8 #這裡的ret是個隨機值
(gdb) n

Breakpoint 3, callshellcode () at testshell.c:12
12      (*ret) = (int)shellcode;
(gdb) print ret      #執行完ret = (int *)&ret + 2;後，ret變成了自己地址加上2*4，
                        #剛好是eip所在的位置。
$5 = (int *) 0xbfcfd2c0
(gdb) x/6x $esp
0xbfcfd2ac:  0x08048389      0xb7f4eff4      0xbfcfd36c      0xbfcfd2c0
0xbfcfd2bc:  0xbfcfd2c8      0x08048352
(gdb) x/4x *ret      #此時*ret剛好為eip, 0x8048352
0x8048352 <main+19>:  0x000000b8      0x8d5d5900      0x90c3fc61      0x89559090
(gdb) n

Breakpoint 4, callshellcode () at testshell.c:13
13      }
(gdb) x/6x $esp      #現在eip被修改為了shellcode的入口
0xbfcfd2ac:  0x08048389      0xb7f4eff4      0xbfcfd36c      0xbfcfd2c0
0xbfcfd2bc:  0xbfcfd2c8      0x8049560
(gdb) x/4x *ret      #現在修改了(*ret)的值，即修改了eip的值，使eip指向了shellcode
0x8049560 <shellcode>:  0xb85e20eb      0x00000004      0x000002bb      0xbaf18900

```

上面的過程很難弄，呵呵。主要是指針不大好理解，如果直接把它當地址繪出下面的圖可能會容易理解一些。

callshellcode棧的初始分佈：

```

ret=(int *)&ret+2=0xbfcfd2bc+2*4=0xbfcfd2c0
0xbfcfd2b8      ret(隨機值)                0xbfcfd2c0
0xbfcfd2bc      ebp(這裡不關心)
0xbfcfd2c0      eip(0x08048352)          eip(0x8049560 )

(*ret) = (int)shellcode;即eip=0x8049560

```

總之，最後體現為函數調用的下一條指令指針（`eip`）被修改為一段注入代碼的入口，從而使得函數返回時執行了注入代碼。

緩衝區注入與防範

這個程序裡頭的注入代碼和被注入程序竟然是一個程序，傻瓜才自己攻擊自己（不過有些黑客有可能利用程序中一些空間空間注入代碼哦），真正的緩衝區注入程序是分開的，比如作為被注入程序的一個字符串參數。而在被注入程序中剛好沒有做字符串長度的限制，從而讓這段字符串中的一部分修改了 `eip`，另外一部分作為注入代碼運行了，從而實現了注入的目的。不過這會涉及到一些技巧，即如何剛好用注入代碼的入口地址來修改 `eip`（即新的 `eip` 能夠指向注入代碼）？如果 `eip` 的位置和緩衝區的位置之間的距離是確定，那麼就比較好處理了，但從上面的兩個例子中我們發現，有一個編譯後有 `edi` 和 `esi`，而另外一個則沒有，另外，緩衝區的位置，以及被注入程序有多少個參數我們都無法預知，因此，如何計算 `eip` 所在的位置呢？這也會很難確定。還有，為了防止緩衝區溢出帶來的注入問題，現在的操作系統採取了一些辦法，比如讓 `esp` 隨機變化（比如和系統時鐘關聯起來），所以這些措施將導致注入更加困難。如果有興趣，你可以接著看看最後的幾篇參考資料並進行更深入的研究。

需要提到的是，因為很多程序可能使用 `strcpy` 來進行字符串的複製，在實際編寫緩衝區注入代碼時，會採取一定的辦法（指令替換），把代碼中可能包含的 `\0` 字節去掉，從而防止 `strcpy` 中斷對注入代碼的複製，進而可以複製完整的注入代碼。具體的技巧可以參考 [Linux下緩衝區溢出攻擊的原理及對策](#)，[Shellcode技術雜談](#)，[virus-writing-HOWTO](#)。

後記

實際上緩衝區溢出應該是語法和邏輯方面的雙重問題，由於語法上的不嚴格（對數組邊界沒有檢查）導致邏輯上可能出現嚴重缺陷（程序執行行為被改變）。另外，這類問題是對程序運行過程中的程序映像的棧區進行注入。實際上除此之外，程序在安全方面還有很多類似的問題。比如，雖然程序映像的正文區受到系統保護（只讀），但是如果內存（硬件本身，內存條）出現故障，在程序運行的過程中，程序映像的正文區的某些字節就可能被修改了，也可能發生非常嚴重的後果，因此程序運行過程的正文區檢查等可能的手段需要被引入。

參考資料

- [Playing with ptrace](#)
 - [how ptrace can be used to trace system calls and change system call arguments](#)
 - [setting breakpoints and injecting code into running programs](#)
 - [fix the problem of ORIG_EAX not defined](#)
- [《緩衝區溢出攻擊——檢測、剖析與預防》第五章](#)
- [Linux下緩衝區溢出攻擊的原理及對策](#)
- [Linux 彙編語言開發指南](#)
- [Shellcode 技術雜談](#)

進程的內存映像

- [前言](#)
- [進程內存映像表](#)
- [在程序內部打印內存分佈信息](#)
- [在程序內部獲取完整內存分佈信息](#)
- [後記](#)
- [參考資料](#)

前言

在閱讀《UNIX 環境高級編程》的第 14 章時，看到一個“打印不同類型的數據所存放的位置”的例子，它非常清晰地從程序內部反應了“進程的內存映像”，通過結合它與《[Gcc 編譯的背後](#)》和《[緩衝區溢出與注入分析](#)》的相關內容，可以更好地輔助理解相關的內容。

進程內存映像表

首先回顧一下《[緩衝區溢出與注入](#)》中提到的“進程內存映像表”，並把共享內存的大概位置加入該表：

地址	內核空間	描述
0xC0000000		
	(program file) 程序名	execve 的第一個參數
	(environment) 環境變量	execve 的第三個參數，main 的第三個參數
	(arguments) 參數	execve 的第二個參數，main 的形參
	(stack) 棧	自動變量以及每次函數調用時所需保存的信息都
		存放在此，包括函數返回地址、調用者的
		環境信息等，函數的參數，局部變量都存放在此
	(shared memory) 共享內存	共享內存的大概位置
	...	
	...	
	(heap) 堆	主要在這裡進行動態存儲分配，比如 malloc，new 等。
	...	
	.bss (uninitialized data)	沒有初始化的數據（全局變量哦）
	.data (initialized global data)	已經初始化的全局數據（全局變量）
	.text (Executable Instructions)	通常是可執行指令
0x08048000		
0x00000000		...

在程序內部打印內存分佈信息

為了能夠反應上述內存分佈情況，這裡在《UNIX 環境高級編程》的程序 14-11 的基礎上，添加了一個已經初始化的全局變量（存放在已經初始化的數據段內），並打印了它以及 `main` 函數(處在代碼正文部分)的位置。

```
/**
 * showmemory.c -- print the position of different types of data in a program in the memory
 */

#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/shm.h>
#include <stdio.h>
#include <stdlib.h>

#define ARRAY_SIZE 4000
#define MALLOC_SIZE 100000
#define SHM_SIZE 100000
#define SHM_MODE (SHM_R | SHM_W) /* user read/write */

int init_global_variable = 5; /* initialized global variable */
char array[ARRAY_SIZE]; /* uninitialized data = bss */

int main(void)
{
    int shmid;
    char *ptr, *shmptr;

    printf("main: the address of the main function is %x\n", main);
    printf("data: data segment is from %x\n", &init_global_variable);
    printf("bss: array[] from %x to %x\n", &array[0], &array[ARRAY_SIZE]);
    printf("stack: around %x\n", &shmid);

    /* shmid is a local variable, which is stored in the stack, hence, you
     * can get the address of the stack via it*/

    if ( (ptr = malloc(MALLOC_SIZE)) == NULL) {
        printf("malloc error!\n");
        exit(-1);
    }
    printf("heap: malloced from %x to %x\n", ptr, ptr+MALLOC_SIZE);

    if ( (shmid = shmget(IPC_PRIVATE, SHM_SIZE, SHM_MODE)) < 0) {
        printf("shmget error!\n");
        exit(-1);
    }

    if ( (shmptr = shmat(shmid, 0, 0)) == (void *) -1) {
        printf("shmat error!\n");
        exit(-1);
    }
    printf("shared memory: attached from %x to %x\n", shmptr, shmptr+SHM_SIZE);

    if (shmctl(shmid, IPC_RMID, 0) < 0) {
        printf("shmctl error!\n");
        exit(-1);
    }
}
```

```

    exit(0);
}

```

該程序的運行結果如下：

```

$ make showmemory
cc      showmemory.c  -o showmemory
$ ./showmemory
main: the address of the main function is 804846c
data: data segment is from 80498e8
bss: array[] from 8049920 to 804a8c0
stack: around bfe3e224
heap: malloced from 804b008 to 80636a8
shared memory: attached from b7da7000 to b7dbf6a0

```

上述運行結果反應了幾個重要部分數據的大概分佈情況，比如 `data` 段（那個初始化過的全局變量就位於這裡）、`bss` 段、`stack`、`heap`，以及 `shared memory` 和 `main`（代碼段）的內存分佈情況。

在程序內部獲取完整內存分佈信息

不過，這些結果還是沒有精確和完整地反應所有相關信息，如果要想在程序內完整反應這些信息，結合《[Gcc編譯的背後](#)》，就不難想到，我們還可以通過擴展一些已經鏈接到可執行文件中的外部符號來獲取它們。這些外部符號全部定義在可執行文件的符號表中，可以通過 `nm/readelf -s/objdump -t` 等查看到，例如：

```

$ nm showmemory
080497e4 d __DYNAMIC
080498b0 d __GLOBAL_OFFSET_TABLE__
080486c8 R __IO_stdin_used
          w __Jv_RegisterClasses
080497d4 d __CTOR_END__
080497d0 d __CTOR_LIST__
080497dc d __DTOR_END__
080497d8 d __DTOR_LIST__
080487cc r __FRAME_END__
080497e0 d __JCR_END__
080497e0 d __JCR_LIST__
080498ec A __bss_start
080498dc D __data_start
08048680 t __do_global_ctors_aux
08048414 t __do_global_dtors_aux
080498e0 D __dso_handle
          w __gmon_start__
0804867a T __i686.get_pc_thunk.bx
080497d0 d __init_array_end
080497d0 d __init_array_start
08048610 T __libc_csu_fini
08048620 T __libc_csu_init
          U __libc_start_main@@GLIBC_2.0
080498ec A __edata
0804a8c0 A __end

```

```

080486a8 T _fini
080486c4 R _fp_hw
08048328 T _init
080483f0 T _start
08049920 B array
08049900 b completed.1
080498dc W data_start
          U exit@@GLIBC_2.0
08048444 t frame_dummy
080498e8 D init_global_variable
0804846c T main
          U malloc@@GLIBC_2.0
080498e4 d p.0
          U printf@@GLIBC_2.0
          U shmat@@GLIBC_2.0
          U shmctl@@GLIBC_2.2
          U shmget@@GLIBC_2.0

```

第三列的符號在我們的程序中被擴展以後就可以直接引用，這些符號基本上就已經完整地覆蓋了相關的信息了，這樣就可以得到一個更完整的程序，從而完全反應上面提到的內存分佈表的信息。

```

/**
 * showmemory.c -- print the position of different types of data in a program in the memory
 */

#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/shm.h>
#include <stdio.h>
#include <stdlib.h>

#define ARRAY_SIZE 4000
#define MALLOC_SIZE 100000
#define SHM_SIZE 100000
#define SHM_MODE (SHM_R | SHM_W)          /* user read/write */

/* declare the address relative variables */
extern char _start, __data_start, __bss_start, etext, edata, end;
extern char **environ;

char array[ARRAY_SIZE];          /* uninitialized data = bss */

int main(int argc, char *argv[])
{
    int shmid;
    char *ptr, *shmptr;

    printf("==== memory map =====\n");
    printf(".text:\t0x%x->0x%x (_start, code text)\n", &_start, &etext);
    printf(".data:\t0x%x->0x%x (__data_start, initilized data)\n", &__data_start, &edata);
    printf(".bss: \t0x%x->0x%x (__bss_start, uninitilized data)\n", &__bss_start, &end);

    /* shmid is a local variable, which is stored in the stack, hence, you
     * can get the address of the stack via it*/

    if ( (ptr = malloc(MALLOC_SIZE)) == NULL) {
        printf("malloc error!\n");
    }
}

```

```

    exit(-1);
}

printf("heap: \t0x%x->0x%x (address of the malloc space)\n", ptr, ptr+MALLOC_SIZE);

if ( (shmid = shmget(IPC_PRIVATE, SHM_SIZE, SHM_MODE)) < 0) {
    printf("shmget error!\n");
    exit(-1);
}

if ( (shmptr = shmat(shmid, 0, 0)) == (void *) -1) {
    printf("shmat error!\n");
    exit(-1);
}

printf("shm : \t0x%x->0x%x (address of shared memory)\n", shmptr, shmptr+SHM_SIZE);

if (shmctl(shmid, IPC_RMID, 0) < 0) {
    printf("shmctl error!\n");
    exit(-1);
}

printf("stack: \t <--0x%x--> (address of local variables)\n", &shmid);
printf("arg: \t0x%x (address of arguments)\n", argv);
printf("env: \t0x%x (address of environment variables)\n", environ);

exit(0);
}

```

運行結果：

```

$ make showmemory
$ ./showmemory
===== memory map =====
.text:    0x8048440->0x8048754 (_start, code text)
.data:    0x8049a3c->0x8049a48 (__data_start, initialized data)
.bss:     0x8049a48->0x804aa20 (__bss_start, uninitialized data)
heap:     0x804b008->0x80636a8 (address of the malloc space)
shm :     0xb7db6000->0xb7dce6a0 (address of shared memory)
stack:    <--0xbff85b64--> (address of local variables)
arg:      0xbff85bf4 (address of arguments)
env:      0xbff85bfc (address of environment variables)

```

後記

上述程序完整地勾勒出了進程的內存分佈的各個重要部分，這樣就可以從程序內部獲取跟程序相關的所有數據了，一個非常典型的例子是，在程序運行的過程中檢查代碼正文部分是否被惡意篡改。

如果想更深度理解相關內容，那麼可以試著利用 `readelf`，`objdump` 等來分析 ELF 可執行文件格式的結構，並利用 `gdb` 來了解程序運行過程中的內存變化情況。

參考資料

- [Gcc 編譯的背後（第二部分：彙編和鏈接）](#)
- [緩衝區溢出與注入分析](#)
- 《Unix 環境高級編程》第 14 章，程序 14-11

進程和進程的基本操作

- [前言](#)
- [什麼是程序，什麼又是進程](#)
- [進程的創建](#)
 - [讓程序在後臺運行](#)
 - [查看進程 ID](#)
 - [查看進程的內存映像](#)
- [查看進程的屬性和狀態](#)
 - [通過 ps 命令查看進程屬性](#)
 - [通過 pstree 查看進程親緣關係](#)
 - [用 top 動態查看進程信息](#)
 - [確保特定程序只有一個副本在運行](#)
- [調整進程的優先級](#)
 - [獲取進程優先級](#)
 - [調整進程的優先級](#)
- [結束進程](#)
 - [結束進程](#)
 - [暫停某個進程](#)
 - [查看進程退出狀態](#)
- [進程通信](#)
 - [無名管道（pipe）](#)
 - [有名管道（named pipe）](#)
 - [信號（Signal）](#)
- [作業和作業控制](#)
 - [創建後臺進程，獲取進程的作業號和進程號](#)
 - [把作業調到前臺並暫停](#)
 - [查看當前作業情況](#)
 - [啟動停止的進程並運行在後臺](#)
- [參考資料](#)

前言

進程作為程序真正發揮作用時的“形態”，我們有必要對它的一些相關操作非常熟悉，這一節主要描述進程相關的概念和操作，將介紹包括程序、進程、作業等基本概念以及進程狀態查詢、進程通信等相關的操作。

什麼是程序，什麼又是進程

程序是指令的集合，而進程則是程序執行的基本單元。為了讓程序完成它的工作，必須讓程序運行起來成為進程，進而利用處理器資源、內存資源，進行各種 I/O 操作，從而完成某項特定工作。

從這個意思上說，程序是靜態的，而進程則是動態的。

進程有區別於程序的地方還有：進程除了包含程序文件中的指令數據以外，還需要在內核中有一個數據結

構用以存放特定進程的相關屬性，以便內核更好地管理和調度進程，從而完成多進程協作的任務。因此，從這個意義上可以說“高於”程序，超出了程序指令本身。

如果進行過多進程程序的開發，又會發現，一個程序可能創建多個進程，通過多個進程的交互完成任務。在 Linux 下，多進程的創建通常是通過 `fork` 系統調用來實現。從這個意義上來說程序則“包含”了進程。

另外一個需要明確的是，程序可以由多種不同程序語言描述，包括 C 語言程序、彙編語言程序和最後編譯產生的機器指令等。

下面簡單討論 Linux 下面如何通過 Shell 進行進程的相關操作。

進程的創建

通常在命令行鍵入某個程序文件名以後，一個進程就被創建了。例如，

讓程序在後臺運行

```
$ sleep 100 &
[1] 9298
```

查看進程 ID

用 `pidof` 可以查看指定程序名的進程ID：

```
$ pidof sleep
9298
```

查看進程的內存映像

```
$ cat /proc/9298/maps
08048000-0804b000 r-xp 00000000 08:01 977399      /bin/sleep
0804b000-0804c000 rw-p 00003000 08:01 977399      /bin/sleep
0804c000-0806d000 rw-p 0804c000 00:00 0          [heap]
b7c8b000-b7cca000 r--p 00000000 08:01 443354
...
bfbdb000-bfbdd000 rw-p bfbdb000 00:00 0          [stack]
ffffe000-fffff000 r-xp 00000000 00:00 0          [vdso]
```

程序被執行後，就被加載到內存中，成為了一個進程。上面顯示了該進程的內存映像（虛擬內存），包括程序指令、數據，以及一些用於存放程序命令行參數、環境變量的棧空間，用於動態內存申請的堆空間都被分配好。

關於程序在命令行執行過程的細節，請參考《[Linux 命令行下程序執行的一剎那](#)》。

實際上，創建一個進程，也就是說讓程序運行，還有其他的辦法，比如，通過一些配置讓系統啟動時自動啟動程序（具體參考 `man init`），或者是通過配置 `cron`（或者 `at`）讓它定時啟動程序。除此之外，

還有一個方式，那就是編寫 Shell 腳本，把程序寫入一個腳本文件，當執行腳本文件時，文件中的程序將被執行而成為進程。這些方式的細節就不介紹，下面瞭解如何查看進程的屬性。

需要補充一點的是：在命令行下執行程序，可以通過 `ulimit` 內置命令來設置進程可以利用的資源，比如進程可以打開的最大文件描述符個數，最大的棧空間，虛擬內存空間等。具體用法見 `help ulimit`。

查看進程的屬性和狀態

可以通過 `ps` 命令查看進程相關屬性和狀態，這些信息包括進程所屬用戶，進程對應的程序，進程對 `cpu` 和內存的使用情況等信息。熟悉如何查看它們有助於進行相關的統計分析等操作。

通過 `ps` 命令查看進程屬性

查看系統當前所有進程的屬性：

```
$ ps -ef
```

查看命令中包含某字符的程序對應的進程，進程 `ID` 是 1。 `TTY` 為 `?` 表示和終端沒有關聯：

```
$ ps -C init
  PID TTY          TIME CMD
    1 ?            00:00:01 init
```

選擇某個特定用戶啟動的進程：

```
$ ps -U falcon
```

按照指定格式輸出指定內容，下面輸出命令名和 `cpu` 使用率：

```
$ ps -e -o "%C %c"
```

打印 `cpu` 使用率最高的前 4 個程序：

```
$ ps -e -o "%C %c" | sort -u -k1 -r | head -5
 7.5 firefox-bin
 1.1 Xorg
 0.8 scim-panel-gtk
 0.2 scim-bridge
```

獲取使用虛擬內存最大的 5 個進程：

```
$ ps -e -o "%z %c" | sort -n -k1 -r | head -5
349588 firefox-bin
96612 xfce4-terminal
```

```
88840 xfdesktop
76332 gedit
58920 scim-panel-gtk
```

通過 `pstree` 查看進程親緣關係

系統所有進程之間都有“親緣”關係，可以通過 `pstree` 查看這種關係：

```
$ pstree
```

上面會打印系統進程調用樹，可以非常清楚地看到當前系統中所有活動進程之間的調用關係。

用 `top` 動態查看進程信息

```
$ top
```

該命令最大特點是可以動態地查看進程信息，當然，它還提供了一些其他的參數，比如 `-s` 可以按照累計執行時間的大小排序查看，也可以通過 `-u` 查看指定用戶啟動的進程等。

補充：`top` 命令支持交互式，比如它支持 `u` 命令顯示用戶的所有進程，支持通過 `k` 命令殺掉某個進程；如果使用 `-n 1` 選項可以啟用批處理模式，具體用法為：

```
$ top -n 1 -b
```

確保特定程序只有一個副本在運行

下面來討論一個有趣的問題：如何讓一個程序在同一時間只有一個在運行。

這意味著當一個程序正在被執行時，它將不能再被啟動。那該怎麼做呢？

假如一份相同的程序被複製成了很多份，並且具有不同的文件名被放在不同的位置，這個將比較糟糕，所以考慮最簡單的情況，那就是這份程序在整個系統上是唯一的，而且名字也是唯一的。這樣的話，有哪些辦法來回答上面的問題呢？

總的機理是：在程序開頭檢查自己有沒有執行，如果執行了則停止否則繼續執行後續代碼。

策略則是多樣的，由於前面的假設已經保證程序文件名和代碼的唯一性，所以通過 `ps` 命令找出當前所有進程對應的程序名，逐個與自己的程序名比較，如果已經有，那麼說明自己已經運行了。

```
ps -e -o "%c" | tr -d " " | grep -q ^init$    #查看當前程序是否執行
[ $? -eq 0 ] && exit    #如果在，那麼退出，$?表示上一條指令是否執行成功
```

每次運行時先在指定位置檢查是否存在一個保存自己進程 ID 的文件，如果不存在，那麼繼續執行，如果存在，那麼查看該進程 ID 是否正在運行，如果在，那麼退出，否則往該文件重新寫入新的進程 ID，並

繼續。

```
pidfile=/tmp/$0".pid"
if [ -f $pidfile ]; then
    OLDPID=$(cat $pidfile)
    ps -e -o "%p" | tr -d " " | grep -q "^$OLDPID$"
    [ $? -eq 0 ] && exit
fi

echo $$ > $pidfile

#... 代碼主體

#設置信號0的動作，當程序退出時觸發該信號從而刪除掉臨時文件
trap "rm $pidfile" 0
```

更多實現策略自己盡情發揮吧！

調整進程的優先級

在保證每個進程都能夠順利執行外，為了讓某些任務優先完成，那麼系統在進行進程調度時就會採用一定的調度辦法，比如常見的有按照優先級的時間片輪轉的調度算法。這種情況下，可以通過 `renice` 調整正在運行的程序的優先級，例如：

獲取進程優先級

```
$ ps -e -o "%p %c %n" | grep xfs
5089 xfs 0
```

調整進程的優先級

```
$ renice 1 -p 5089
renice: 5089: setpriority: Operation not permitted
$ sudo renice 1 -p 5089 #需要權限才行
[sudo] password for falcon:
5089: old priority 0, new priority 1
$ ps -e -o "%p %c %n" | grep xfs #再看看，優先級已經被調整過來了
5089 xfs 1
```

結束進程

既然可以通過命令行執程序，創建進程，那麼也有辦法結束它。可以通過 `kill` 命令給用戶自己啟動的進程發送某個信號讓進程終止，當然“萬能”的 `root` 幾乎可以 `kill` 所有進程（除了 `init` 之外）。例如，

結束進程

```
$ sleep 50 &    #啟動一個進程
[1] 11347
$ kill 11347
```

`kill` 命令默認會發送終止信號 (`SIGTERM`) 給程序，讓程序退出，但是 `kill` 還可以發送其他信號，這些信號的定義可以通過 `man 7 signal` 查看到，也可以通過 `kill -l` 列出來。

```
$ man 7 signal
$ kill -l
1) SIGHUP          2) SIGINT          3) SIGQUIT         4) SIGILL
5) SIGTRAP         6) SIGABRT         7) SIGBUS          8) SIGFPE
9) SIGKILL         10) SIGUSR1        11) SIGSEGV        12) SIGUSR2
13) SIGPIPE        14) SIGALRM        15) SIGTERM        16) SIGSTKFLT
17) SIGCHLD        18) SIGCONT        19) SIGSTOP        20) SIGTSTP
21) SIGTTIN        22) SIGTTOU        23) SIGURG         24) SIGXCPU
25) SIGXFSZ        26) SIGVTALRM      27) SIGPROF        28) SIGWINCH
29) SIGIO          30) SIGPWR         31) SIGSYS         34) SIGRTMIN
35) SIGRTMIN+1     36) SIGRTMIN+2     37) SIGRTMIN+3     38) SIGRTMIN+4
39) SIGRTMIN+5     40) SIGRTMIN+6     41) SIGRTMIN+7     42) SIGRTMIN+8
43) SIGRTMIN+9     44) SIGRTMIN+10    45) SIGRTMIN+11    46) SIGRTMIN+12
47) SIGRTMIN+13    48) SIGRTMIN+14    49) SIGRTMIN+15    50) SIGRTMAX-14
51) SIGRTMAX-13    52) SIGRTMAX-12    53) SIGRTMAX-11    54) SIGRTMAX-10
55) SIGRTMAX-9     56) SIGRTMAX-8     57) SIGRTMAX-7     58) SIGRTMAX-6
59) SIGRTMAX-5     60) SIGRTMAX-4     61) SIGRTMAX-3     62) SIGRTMAX-2
63) SIGRTMAX-1     64) SIGRTMAX
```

暫停某個進程

例如，用 `kill` 命令發送 `SIGSTOP` 信號給某個程序，讓它暫停，然後發送 `SIGCONT` 信號讓它繼續運行。

```
$ sleep 50 &
[1] 11441
$ jobs
[1]+  Running                  sleep 50 &
$ kill -s SIGSTOP 11441    #這個等同於我們對一個前臺進程執行CTRL+Z操作
$ jobs
[1]+  Stopped                  sleep 50
$ kill -s SIGCONT 11441    #這個等同於之前我們使用bg %1操作讓一個後臺進程運行起來
$ jobs
[1]+  Running                  sleep 50 &
$ kill %1                  #在當前會話(session)下，也可以通過作業號控制進程
$ jobs
[1]+  Terminated             sleep 50
```

可見 `kill` 命令提供了非常好的功能，不過它只能根據進程的 ID 或者作業來控制進程，而 `pkill` 和 `killall` 提供了更多選擇，它們擴展了通過程序名甚至是進程的用戶名來控制進程的方法。更多用法請參考它們的手冊。

查看進程退出狀態

當程序退出後，如何判斷這個程序是正常退出還是異常退出呢？還記得 Linux 下，那個經典 `hello world` 程序嗎？在代碼的最後總是有條 `return 0` 語句。這個 `return 0` 實際上是讓程序員來檢查進程是否正常退出的。如果進程返回了一個其他的數值，那麼可以肯定地說這個進程異常退出了，因為它都沒有執行到 `return 0` 這條語句就退出了。

那怎麼檢查進程退出的狀態，即那個返回的數值呢？

在 `Shell` 中，可以檢查這個特殊的變量 `$?`，它存放了上一條命令執行後的退出狀態。

```
$ test1
bash: test1: command not found
$ echo $?
127
$ cat ./test.c | grep hello
$ echo $?
1
$ cat ./test.c | grep hi
    printf("hi, myself!\n");
$ echo $?
0
```

貌似返回 0 成為了一個潛規則，雖然沒有標準明確規定，不過當程序正常返回時，總是可以從 `$?` 中檢測到 0，但是異常時，總是檢測到一個非 0 值。這就告訴我們在程序的最後最好是跟上一個 `exit 0` 以便任何人都可以通過檢測 `$?` 確定程序是否正常結束。如果有一天，有人偶爾用到你的程序，試圖檢查它的退出狀態，而你卻在程序的末尾莫名地返回了一個 `-1` 或者 `1`，那麼他將會很苦惱，會懷疑他自己編寫的程序到底哪個地方出了問題，檢查半天卻不知所措，因為他太信任你了，竟然從頭至尾都沒有懷疑你的編程習慣可能會與眾不同！

進程通信

為便於設計和實現，通常一個大型的任務都被劃分成較小的模塊。不同模塊之間啟動後成為進程，它們之間如何通信以便交互數據，協同工作呢？在《UNIX 環境高級編程》一書中提到很多方法，諸如管道（無名管道和有名管道）、信號（`signal`）、報文（`Message`）隊列（消息隊列）、共享內存（`mmap/munmap`）、信號量（`semaphore`，主要是同步用，進程之間，進程的不同線程之間）、套接口（`Socket`，支持不同機器之間的進程通信）等，而在 `Shell` 中，通常直接用到的就有管道和信號等。下面主要介紹管道和信號機制在 `Shell` 編程時的一些用法。

無名管道（`pipe`）

在 Linux 下，可以通過 `|` 連接兩個程序，這樣就可以用它來連接後一個程序的輸入和前一個程序的輸出，因此被形象地叫做個管道。在 C 語言中，創建無名管道非常簡單方便，用 `pipe` 函數，傳入一個具有兩個元素的 `int` 型的數組就可以。這個數組實際上保存的是兩個文件描述符，父進程往第一個文件描述符裡頭寫入東西后，子進程可以從第一個文件描述符中讀出來。

如果用多了命令行，這個管子 `|` 應該會經常用。比如上面有個演示把 `ps` 命令的輸出作為 `grep` 命令的輸入：

```
$ ps -ef | grep init
```

也許會覺得這個“管子”好有魔法，竟然真地能夠鏈接兩個程序的輸入和輸出，它們到底是怎麼實現的呢？實際上當輸入這樣一組命令時，當前 Shell 會進行適當的解析，把前面一個進程的輸出關聯到管道的輸出文件描述符，把後面一個進程的輸入關聯到管道的輸入文件描述符，這個關聯過程通過輸入輸出重定向函數 `dup`（或者 `fcntl`）來實現。

有名管道（named pipe）

有名管道實際上是一個文件（無名管道也像一個文件，雖然關係到兩個文件描述符，不過只能一邊讀另外一邊寫），不過這個文件比較特別，操作時要滿足先進先出，而且，如果試圖讀一個沒有內容的有名管道，那麼就會被阻塞，同樣地，如果試圖往一個有名管道里寫東西，而當前沒有程序試圖讀它，也會被阻塞。下面看看效果。

```
$ mkfifo fifo_test      #通過mkfifo命令創建一個有名管道
$ echo "fewefe" > fifo_test
#試圖往fifo_test文件中寫入內容，但是被阻塞，要另開一個終端繼續下面的操作
$ cat fifo_test          #另開一個終端，記得，另開一個。試圖讀出fifo_test的內容
fewefe
```

這裡的 `echo` 和 `cat` 是兩個不同的程序，在這種情況下，通過 `echo` 和 `cat` 啟動的兩個進程之間並沒有父子關係。不過它們依然可以通過有名管道通信。

這樣一種通信方式非常適合某些特定情況：例如有這樣一個架構，這個架構由兩個應用程序構成，其中一個通過循環不斷讀取 `fifo_test` 中的內容，以便判斷，它下一步要做什麼。如果這個管道沒有內容，那麼它就會被阻塞在那裡，而不會因死循環而耗費資源，另外一個則作為一個控制程序不斷地往 `fifo_test` 中寫入一些控制信息，以便告訴之前的那個程序該做什麼。下面寫一個非常簡單的例子。可以設計一些控制碼，然後控制程序不斷地往 `fifo_test` 裡頭寫入，然後應用程序根據這些控制碼完成不同的動作。當然，也可以往 `fifo_test` 傳入除控制碼外的其他數據。

- 應用程序的代碼

```
$ cat app.sh
#!/bin/bash

FIFO=fifo_test
while ;;
do
    CI=`cat $FIFO` #CI --> Control Info
    case $CI in
        0) echo "The CONTROL number is ZERO, do something ..."
            ;;
        1) echo "The CONTROL number is ONE, do something ..."
            ;;
        *) echo "The CONTROL number not recognized, do something else..."
            ;;
    esac
done
```


- 控制程序的代碼

```
$ cat control.sh
#!/bin/bash

FIFO=fifo_test
CI=$1

[ -z "$CI" ] && echo "the control info should not be empty" && exit

echo $CI > $FIFO
```

- 一個程序通過管道控制另外一個程序的工作

```
$ chmod +x app.sh control.sh      #修改這兩個程序的可執行權限，以便用戶可以執行它們
$ ./app.sh      #在一個終端啟動這個應用程序，在通過./control.sh發送控制碼以後查看輸出
The CONTROL number is ONE, do something ...      #發送1以後
The CONTROL number is ZERO, do something ...      #發送0以後
The CONTROL number not recognized, do something else... #發送一個未知的控制碼以後
$ ./control.sh 1      #在另外一個終端，發送控制信息，控制應用程序的工作
$ ./control.sh 0
$ ./control.sh 4343
```

這樣一種應用架構非常適合本地的多程序任務設計，如果結合 `web cgi`，那麼也將適合遠程控制的要求。引入 `web cgi` 的唯一改變是，要把控制程序 `./control.sh` 放到 `web` 的 `cgi` 目錄下，並對它作一些修改，以使它符合 `CGI` 的規範，這些規範包括文檔輸出格式的表示（在文件開頭需要輸出 `content-tpye: text/html` 以及一個空白行）和輸入參數的獲取（`web` 輸入參數都存放在 `QUERY_STRING` 環境變量裡頭）。因此一個非常簡單的 `CGI` 控制程序可以寫成這樣：

```
#!/bin/bash

FIFO=./fifo_test
CI=$QUERY_STRING

[ -z "$CI" ] && echo "the control info should not be empty" && exit

echo -e "content-type: text/html\n\n"
echo $CI > $FIFO
```

在實際使用時，請確保 `control.sh` 能夠訪問到 `fifo_test` 管道，並且有寫權限，以便通過瀏覽器控制 `app.sh`：

```
http://ipaddress\_or\_dns/cgi-bin/control.sh?0
```

問號 `?` 後面的內容即 `QUERY_STRING`，類似之前的 `$1`。

這樣一種應用對於遠程控制，特別是嵌入式系統的遠程控制很有實際意義。在去年的暑期課程上，我們就通過這樣一種方式來實現馬達的遠程控制。首先，實現了一個簡單的應用程序以便控制馬達的轉動，包括

轉速，方向等的控制。為了實現遠程控制，我們設計了一些控制碼，以便控制馬達轉動相關的不同屬性。

在 C 語言中，如果要使用有名管道，和 Shell 類似，只不過在讀寫數據時用 `read`，`write` 調用，在創建 `fifo` 時用 `mkfifo` 函數調用。

信號（Signal）

信號是軟件中斷，Linux 用戶可以通過 `kill` 命令給某個進程發送一個特定的信號，也可以通過鍵盤發送一些信號，比如 `CTRL+C` 可能觸發 `SIGINT` 信號，而 `CTRL+\` 可能觸發 `SIGQUIT` 信號等，除此之外，內核在某些情況下也會給進程發送信號，比如在訪問內存越界時產生 `SEGV` 信號，當然，進程本身也可以通過 `kill`，`raise` 等函數給自己發送信號。對於 Linux 下支持的信號類型，大家可以通過 `man 7 signal` 或者 `kill -l` 查看到相關列表和說明。

對於有些信號，進程會有默認的響應動作，而有些信號，進程可能直接會忽略，當然，用戶還可以對某些信號設定專門的處理函數。在 Shell 中，可以通過 `trap` 命令（Shell 內置命令）來設定響應某個信號的動作（某個命令或者定義的某個函數），而在 C 語言中可以通過 `signal` 調用註冊某個信號的處理函數。這裡僅僅演示 `trap` 命令的用法。

```
$ function signal_handler { echo "hello, world."; } #定義signal_handler函數
$ trap signal_handler SIGINT #執行該命令設定：收到SIGINT信號時打印hello, world
$ hello, world #按下CTRL+C，可以看到屏幕上輸出了hello, world字符串
```

類似地，如果設定信號 0 的響應動作，那麼就可以用 `trap` 來模擬 C 語言程序中的 `atexit` 程序終止函數的登記，即通過 `trap signal_handler SIGQUIT` 設定的 `signal_handler` 函數將在程序退出時執行。信號 0 是一個特別的信號，在 `POSIX.1` 中把信號編號 0 定義為空信號，這常被用來確定一個特定進程是否仍舊存在。當一個程序退出時會觸發該信號。

```
$ cat sigexit.sh
#!/bin/bash

function signal_handler {
    echo "hello, world"
}
trap signal_handler 0
$ chmod +x sigexit.sh
$ ./sigexit.sh #實際Shell編程會用該方式在程序退出時來做一些清理臨時文件的收尾工作
hello, world
```

作業和作業控制

當我們為完成一些複雜的任務而將多個命令通過 `|`，`>`，`<`，`;`，`(,)` 等組合在一起時，通常這個命令序列會啟動多個進程，它們間通過管道等進行通信。而有時在執行一個任務的同時，還有其他的任務需要處理，那麼就經常會在命令序列的最後加上一個 `&`，或者在執行命令後，按下 `CTRL+Z` 讓前一個命令暫停。以便做其他的任務。等做完其他一些任務以後，再通過 `fg` 命令把後臺任務切換到前臺。這樣一種控制過程通常被成為作業控制，而那些命令序列則被成為作業，這個作業可能涉及一個或者多個程序，一個或者多個進程。下面演示一下幾個常用的作業控制操作。

創建後臺進程，獲取進程的作業號和進程號

```
$ sleep 50 &
[1] 11137
```

把作業調到前臺並暫停

使用 Shell 內置命令 `fg` 把作業 1 調到前臺運行，然後按下 `CTRL+Z` 讓該進程暫停

```
$ fg %1
sleep 50
^Z
[1]+  Stopped                  sleep 50
```

查看當前作業情況

```
$ jobs                #查看當前作業情況，有一個作業停止
[1]+  Stopped          sleep 50
$ sleep 100 &         #讓另外一個作業在後臺運行
[2] 11138
$ jobs                #查看當前作業情況，一個正在運行，一個停止
[1]+  Stopped          sleep 50
[2]-  Running          sleep 100 &
```

啟動停止的進程並運行在後臺

```
$ bg %1
[2]+ sleep 50 &
```

不過，要在命令行下使用作業控制，需要當前 Shell，內核終端驅動等對作業控制支持才行。

參考資料

- 《UNIX 環境高級編程》

打造史上最小可執行 ELF 文件（45 字節，可打印字符串）

- 前言
- 可執行文件格式的選取
- 鏈接優化
- 可執行文件“減肥”實例（從6442到708字節）
 - 系統默認編譯
 - 不採用默認編譯
 - 刪除對程序運行沒有影響的節區
 - 刪除可執行文件的節區表
- 用匯編語言來重寫 Hello World（76字節）
 - 採用默認編譯
 - 刪除掉彙編代碼中無關緊要內容
 - 不默認編譯並刪除掉無關節區和節區表
 - 用系統調用取代庫函數
 - 把字符串作為參數輸入
 - 寄存器賦值重用
 - 通過文件名傳遞參數
 - 刪除非必要指令
- 合併代碼段、程序頭和文件頭（52字節）
 - 把代碼段移入文件頭
 - 把程序頭移入文件頭
 - 在非連續的空間插入代碼
 - 把程序頭完全合入文件頭
- 彙編語言極限精簡之道（45字節）
- 小結
- 參考資料

前言

本文從減少可執行文件大小角度分析了 ELF 文件，期間通過經典的 Hello World 實例逐步演示如何通過各種常用工具來分析 ELF 文件，並逐步精簡代碼。

為了能夠儘量減少可執行文件的大小，我們必須瞭解可執行文件的格式，以及鏈接生成可執行文件時的後臺細節（即最終到底有哪些內容被鏈接到了目標代碼中）。通過選擇合適的可執行文件格式並剔除對可執行文件的最終運行沒有影響的內容，就可以實現目標代碼的裁減。因此，通過探索減少可執行文件大小的方法，就相當於實踐性地探索了可執行文件的格式以及鏈接過程的細節。

當然，算法的優化和編程語言的選擇可能對目標文件的大小有很大的影響，在本文最後我們會跟參考資料 [1] 的作者那樣去探求一個打印 Hello World 的可執行文件能夠小到什麼樣的地步。

可執行文件格式的選取

可執行文件格式的選擇要滿足的一個基本條件是：目標系統支持該可執行文件格式，資料 [2] 分析和比較了

UNIX 平臺下的三種可執行文件格式，這三種格式實際上代表著可執行文件的一個發展過程：

- a.out 文件格式非常緊湊，只包含了程序運行所必須的信息（文本、數據、BSS），而且每個 section 的順序是固定的。
- coff 文件格式雖然引入了一個節區表以支持更多節區信息，從而提高了可擴展性，但是這種文件格式的重定位在鏈接時就已經完成，因此不支持動態鏈接（不過擴展的 coff 支持）。
- elf 文件格式不僅動態鏈接，而且有很好的擴展性。它可以描述可重定位文件、可執行文件和可共享文件（動態鏈接庫）三類文件。

下面來看看 ELF 文件的結構圖：

```
文件頭部(ELF Header)
程序頭部表(Program Header Table)
節區1(Section1)
節區2(Section2)
節區3(Section3)
...
節區頭部(Section Header Table)
```

無論是文件頭部、程序頭部表、節區頭部表還是各個節區，都是通過特定的結構體（struct）描述的，這些結構在 elf.h 文件中定義。文件頭部用於描述整個文件的類型、大小、運行平臺、程序入口、程序頭部表和節區頭部表等信息。例如，我們可以通過文件頭部查看該 ELF 文件的類型。

```
$ cat hello.c    #典型的hello, world程序
#include <stdio.h>

int main(void)
{
    printf("hello, world!\n");
    return 0;
}
$ gcc -c hello.c    #編譯，產生可重定向的目標代碼
$ readelf -h hello.o | grep Type    #通過readelf查看文件頭部找出該類型
Type:                                REL (Relocatable file)
$ gcc -o hello hello.o    #生成可執行文件
$ readelf -h hello | grep Type
Type:                                EXEC (Executable file)
$ gcc -fpic -shared -Wl,-soname,libhello.so.0 -o libhello.so.0.0 hello.o    #生成共享庫
$ readelf -h libhello.so.0.0 | grep Type
Type:                                DYN (Shared object file)
```

那節區頭部表（將簡稱節區表）和程序頭部表有什麼用呢？實際上前者只對可重定向文件有用，而後者只對可執行文件和可共享文件有用。

節區表是用來描述各節區的，包括各節區的名字、大小、類型、虛擬內存中的位置、相對文件頭的位置等，這樣所有節區都通過節區表給描述了，這樣連接器就可以根據文件頭部表和節區表的描述信息對各種輸入的可重定位文件進行合適的鏈接，包括節區的合併與重組、符號的重定位（確認符號在虛擬內存中的地址）等，把各個可重定向輸入文件鏈接成一個可執行文件（或者是可共享文件）。如果可執行文件中使

用了動態連接庫，那麼將包含一些用於動態符號鏈接的節區。我們可以通過 `readelf -S`（或 `objdump -h`）查看節區表信息。

```
$ readelf -S hello #可執行文件、可共享庫、可重定位文件默認都生成有節區表
...
Section Headers:
  [Nr] Name                Type              Addr      Off      Size    ES Flg Lk Inf Al
  [ 0]                     NULL              00000000  0000000  0000000  00      0  0  0
  [ 1] .interp                PROGBITS          08048114  000114  000013  00      A  0  0  1
  [ 2] .note.ABI-tag          NOTE              08048128  000128  000020  00      A  0  0  4
  [ 3] .hash                  HASH              08048148  000148  000028  04      A  5  0  4
  ...
  [ 7] .gnu.version           VERSYM            0804822a  00022a  00000a  02      A  5  0  2
  ...
  [11] .init                   PROGBITS          08048274  000274  000030  00     AX  0  0  4
  ...
  [13] .text                   PROGBITS          080482f0  0002f0  000148  00     AX  0  0 16
  [14] .fini                  PROGBITS          08048438  000438  00001c  00     AX  0  0  4
  ...
```

三種類型文件的節區（各個常見節區的作用請參考資料 [11]）可能不一樣，但是有幾個節區，例如 `.text`，`.data`，`.bss` 是必須的，特別是 `.text`，因為這個節區包含了代碼。如果一個程序使用了動態鏈接庫（引用了動態連接庫中的某個函數），那麼需要 `.interp` 節區以便告知系統使用什麼動態連接器程序來進行動態符號鏈接，進行某些符號地址的重定位。通常，`.rel.text` 節區只有可重定向文件有，用於鏈接時對代碼區進行重定向，而 `.hash`，`.plt`，`.got` 等節區則只有可執行文件（或可共享庫）有，這些節區對程序的運行特別重要。還有一些節區，可能僅僅是用於註釋，比如 `.comment`，這些對程序的運行似乎沒有影響，是可有可無的，不過有些節區雖然對程序的運行沒有用處，但是卻可以用來輔助對程序進行調試或者對程序運行效率有影響。

雖然三類文件都必須包含某些節區，但是節區表對可重定位文件來說才是必須的，而程序的執行卻不需要節區表，只需要程序頭部表以便知道如何加載和執行文件。不過如果需要對可執行文件或者動態連接庫進行調試，那麼節區表卻是必要的，否則調試器將不知道如何工作。下面來介紹程序頭部表，它可通過 `readelf -l`（或 `objdump -p`）查看。

```
$ readelf -l hello.o #對於可重定向文件，gcc沒有產生程序頭部，因為它對可重定向文件沒用

There are no program headers in this file.
$ readelf -l hello #而可執行文件和可共享文件都有程序頭部
...
Program Headers:
  Type           Offset      VirtAddr    PhysAddr    FileSiz MemSiz  Flg Align
  PHDR            0x000034    0x08048034  0x08048034  0x000e0 0x000e0 R E 0x4
  INTERP          0x000114    0x08048114  0x08048114  0x00013 0x00013 R   0x1
    [Requesting program interpreter: /lib/ld-linux.so.2]
  LOAD            0x000000    0x08048000  0x08048000  0x00470 0x00470 R E 0x1000
  LOAD            0x000470    0x08049470  0x08049470  0x0010c 0x00110 RW 0x1000
  DYNAMIC          0x000484    0x08049484  0x08049484  0x000d0 0x000d0 RW 0x4
  NOTE            0x000128    0x08048128  0x08048128  0x00020 0x00020 R   0x4
  GNU_STACK       0x000000    0x00000000  0x00000000  0x00000 0x00000 RW 0x4

Section to Segment mapping:
Segment Sections...
```

```

00
01      .interp
02      .interp .note.ABI-tag .hash .gnu.hash .dynsym .dynstr .gnu.version .gnu.version_r
03      .ctors .dtors .jcr .dynamic .got .got.plt .data .bss
04      .dynamic
05      .note.ABI-tag
06
$ readelf -l libhello.so.0.0  #節區和上面類似，這裡省略

```

從上面可看出程序頭部表描述了一些段（Segment），這些段對應著一個或者多個節區，上面的 `readelf -l` 很好地顯示了各個段與節區的映射。這些段描述了段的名称、類型、大小、第一個字節在文件中的位置、將佔用的虛擬內存大小、在虛擬內存中的位置等。這樣系統程序解釋器將知道如何把可執行文件加載到內存中以及進行動態鏈接等動作。

該可執行文件包含 7 個段，PHDR 指程序頭部，INTERP 正好對應 `.interp` 節區，兩個 LOAD 段包含程序的代碼和數據部分，分別包含有 `.text` 和 `.data`，`.bss` 節區，DYNAMIC 段包含 `.dynamic`，這個節區可能包含動態連接庫的搜索路徑、可重定位表的地址等信息，它們用於動態連接器。NOTE 和 GNU_STACK 段貌似作用不大，只是保存了一些輔助信息。因此，對於一個不使用動態連接庫的程序來說，可能只包含 LOAD 段，如果一個程序沒有數據，那麼只有一個 LOAD 段就可以了。

總結一下，Linux 雖然支持很多種可執行文件格式，但是目前 ELF 較通用，所以選擇 ELF 作為我們的討論對象。通過上面對 ELF 文件分析發現一個可執行的文件可能包含一些對它的運行沒用的信息，比如節區表、一些用於調試、註釋的節區。如果能夠刪除這些信息就可以減少可執行文件的大小，而且不會影響可執行文件的正常運行。

鏈接優化

從上面的討論中已經接觸了動態連接庫。ELF 中引入動態連接庫後極大地方便了公共函數的共享，節約了磁盤和內存空間，因為不再需要把那些公共函數的代碼鏈接到可執行文件，這將減少了可執行文件的大小。

與此同時，靜態鏈接可能會引入一些對代碼的運行可能並非必須的內容。你可以從《GCC 編譯的背後（第二部分：彙編和鏈接）》瞭解到 GCC 鏈接的細節。從那篇 Blog 中似乎可以得出這樣的結論：僅僅從是否影響一個 C 語言程序運行的角度上說，GCC 默認鏈接到可執行文件的幾個可重定位文件

（`crt1.o`，`rt1.o`，`crtbegin.o`，`crtend.o`，`crti.o`）並不是必須的，不過值得注意的是，如果沒有鏈接那些文件但在程序末尾使用了 `return` 語句，`main` 函數將無法返回，因此需要替換為 `_exit` 調用；另外，既然程序在進入 `main` 之前有一個入口，那麼 `main` 入口就不是必須的。因此，如果不採用默認鏈接也可以減少可執行文件的大小。

可執行文件“減肥”實例（從6442到708字節）

這裡主要是根據上面兩點來介紹如何減少一個可執行文件的大小。以 `Hello World` 為例。

首先來看看默認編譯產生的 `Hello World` 的可執行文件大小。

系統默認編譯

打造史上最小可執行ELF文件(45字節)

代碼同上，下面是一組演示，

```
$ uname -r    #先查看內核版本和gcc版本，以便和你的結果比較
2.6.22-14-generic
$ gcc --version
gcc (GCC) 4.1.3 20070929 (prerelease) (Ubuntu 4.1.2-16ubuntu2)
...
$ gcc -o hello hello.c    #默認編譯
$ wc -c hello    #產生一個大小為6442字節的可執行文件
6442 hello
```

不採用默認編譯

可以考慮編輯時就把 `return 0` 替換成 `_exit(0)` 并包含定義該函數的 `unistd.h` 頭文件。下面是從《GCC 編譯的背後（第二部分：彙編和鏈接）》總結出的 `Makefile` 文件。

```
#file: Makefile
#functin: for not linking a program as the gcc do by default
#author: falcon<zhangjinw@gmail.com>
#update: 2008-02-23

MAIN = hello
SOURCE =
OBSJ = hello.o
TARGET = hello
CC = gcc-3.4 -m32
LD = ld -m elf_i386

CFLAGSS += -S
CFLAGSc += -c
LDFLAGS += -dynamic-linker /lib/ld-linux.so.2 -L /usr/lib/ -L /lib -lc
RM = rm -f
SEDC = sed -i -e '/\#include[ "<]*unistd.h[ ">]*;/d;' \
        -i -e '1i \#include <unistd.h>' \
        -i -e 's/return 0;/_exit(0);/'
SEDS = sed -i -e 's/main/_start/g'

all: $(TARGET)

$(TARGET):
    @$(SEDC) $(MAIN).c
    @$(CC) $(CFLAGSS) $(MAIN).c
    @$(SEDS) $(MAIN).s
    @$(CC) $(CFLAGSc) $(MAIN).s $(SOURCE)
    @$(LD) $(LDFLAGS) -o $$ $(OBSJ)
clean:
    @$(RM) $(MAIN).s $(OBSJ) $(TARGET)
```

把上面的代碼複製到一個Makefile文件中，並利用它來編譯hello.c。

```
$ make    #編譯
$ ./hello    #這個也是可以正常工作的
Hello World
```



```
$ wc -c hello    #但是大小減少了4382個字節，減少了將近 70%
2060 hello
$ echo "6442-2060" | bc
4382
$ echo "(6442-2060)/6442" | bc -l
.68022353306426575597
```

對於一個比較小的程序，能夠減少將近 70% “沒用的”代碼。

刪除對程序運行沒有影響的節區

使用上述 `Makefile` 來編譯程序，不鏈接那些對程序運行沒有多大影響的文件，實際上也相當於刪除了一些“沒用”的節區，可以通過下列演示看出這個實質。

```
$ make clean
$ make
$ readelf -l hello | grep "0[0-9]\ \ "
00
01      .interp
02      .interp .hash .dynsym .dynstr .gnu.version .gnu.version_r .rel.plt .plt .text .rodata
03      .dynamic .got.plt
04      .dynamic
05
$ make clean
$ gcc -o hello hello.c
$ readelf -l hello | grep "0[0-9]\ \ "
00
01      .interp
02      .interp .note.ABI-tag .hash .gnu.hash .dynsym .dynstr .gnu.version .gnu.version_r
    .rel.dyn .rel.plt .init .plt .text .fini .rodata .eh_frame
03      .ctors .dtors .jcr .dynamic .got .got.plt .data .bss
04      .dynamic
05      .note.ABI-tag
06
```

通過比較發現使用自定義的 `Makefile` 文件，少了這麼多節區：`.bss .ctors .data .dtors .eh_frame .fini .gnu.hash .got .init .jcr .note.ABI-tag .rel.dyn`。再看看還有哪些節區可以刪除呢？通過之前的分析發現有些節區是必須的，那 `.hash? .gnu.version?` 呢，通過 `strip -R`（或 `objcopy -R`）刪除這些節區試試。

```
$ wc -c hello    #查看大小，以便比較
2060
$ time ./hello    #我們比較一下一些節區對執行時間可能存在的影響
Hello World

real    0m0.001s
user    0m0.000s
sys     0m0.000s
$ strip -R .hash hello    #刪除.hash節區
$ wc -c hello
1448 hello
$ echo "2060-1448" | bc    #減少了612字節
```

```

612
$ time ./hello          #發現執行時間長了一些（實際上也可能是進程調度的問題）
Hello World

real    0m0.006s
user    0m0.000s
sys     0m0.000s
$ strip -R .gnu.version hello  #刪除.gnu.version還是可以工作
$ wc -c hello
1396 hello
$ echo "1448-1396" | bc      #又減少了52字節
52
$ time ./hello
Hello World

real    0m0.130s
user    0m0.004s
sys     0m0.000s
$ strip -R .gnu.version_r hello  #刪除.gnu.version_r就不工作了
$ time ./hello
./hello: error while loading shared libraries: ./hello: unsupported version 0 of Verneed re

```

通過刪除各個節區可以查看哪些節區對程序來說是必須的，不過有些節區雖然並不影響程序的運行卻可能會影響程序的執行效率，這個可以上面的運行時間看出個大概。通過刪除兩個“沒用”的節區，我們又減少了 52+612，即 664 字節。

刪除可執行文件的節區表

用普通的工具沒有辦法刪除節區表，但是參考資料[1]的作者已經寫了這樣一個工具。你可以從[這裡](#)下載到那個工具，它是該作者寫的一序列工具 `ELFkickers` 中的一個。

下載並編譯（注：1.0 之前的版本才支持 32 位和正常編譯，新版本在代碼中明確限定了數據結構為 `Elf64`）：

```

$ git clone https://github.com/BR903/ELFkickers
$ cd ELFkickers/sstrip/
$ git checkout f0622afa  # 檢出 1.0 版
$ make

```

然後複製到 `/usr/bin` 下，下面用它來刪除節區表。

```

$ sstrip hello          #刪除ELF可執行文件的節區表
$ ./hello               #還是可以正常運行，說明節區表對可執行文件的運行沒有任何影響
Hello World
$ wc -c hello           #大小隻剩下708個字節了
708 hello
$ echo "1396-708" | bc  #又減少了688個字節。
688

```

通過刪除節區表又把可執行文件減少了 688 字節。現在回頭看看相對於 `gcc` 默認產生的可執行文件，通

過刪除一些節區和節區表到底減少了多少字節？減幅達到了多少？

```
$ echo "6442-708" | bc    #
5734
$ echo "(6442-708)/6442" | bc -l
.89009624340266997826
```

減少了 5734 多字節，減幅將近 90%，這說明：對於一個簡短的 `hello.c` 程序而言，`gcc` 引入了將近 90% 的對程序運行沒有影響的數據。雖然通過刪除節區和節區表，使得最終的文件只有 708 字節，但是打印一個 `Hello World` 真的需要這麼多字節麼？事實上未必，因為：

- 打印一段 `Hello World` 字符串，我們無須調用 `printf`，也就無須包含動態連接庫，因此 `.interp`，`.dynamic` 等節區又可以去掉。為什麼？我們可以直接使用系統調用 `_(sys_write)`來打印字符串。
- 另外，我們無須把 `Hello World` 字符串存放到可執行文件中？而是讓用戶把它當作參數輸入。

下面，繼續進行可執行文件的“減肥”。

用匯編語言來重寫"Hello World"（76字節）

採用默認編譯

先來看看 `gcc` 默認產生的彙編代碼情況。通過 `gcc` 的 `-S` 選項可得到彙編代碼。

```
$ cat hello.c #這個是使用_exit和printf函數的版本
#include <stdio.h>      /* printf */
#include <unistd.h>     /* _exit */

int main()
{
    printf("Hello World\n");
    _exit(0);
}
$ gcc -S hello.c      #生成彙編
$ cat hello.s         #這裡是彙編代碼
.file "hello.c"
.section .rodata
.LC0:
.string "Hello World"
.text
.globl main
.type main, @function
main:
    leal    4(%esp), %ecx
    andl    $-16, %esp
    pushl   -4(%ecx)
    pushl   %ebp
    movl    %esp, %ebp
    pushl   %ecx
    subl    $4, %esp
    movl    $.LC0, (%esp)
```

```

    call    puts
    movl    $0, (%esp)
    call    _exit
    .size   main, .-main
    .ident  "GCC: (GNU) 4.1.3 20070929 (prerelease) (Ubuntu 4.1.2-16ubuntu2)"
    .section .note.GNU-stack,"",@progbits
$ gcc -o hello hello.s    #看看默認產生的代碼大小
$ wc -c hello
6523 hello

```

刪除掉彙編代碼中無關緊要內容

現在對彙編代碼 `hello.s` 進行簡單的處理得到，

```

.LC0:
    .string "Hello World"
    .text
.globl main
    .type   main, @function
main:
    leal    4(%esp), %ecx
    andl    $-16, %esp
    pushl   -4(%ecx)
    pushl   %ebp
    movl    %esp, %ebp
    pushl   %ecx
    subl    $4, %esp
    movl    $.LC0, (%esp)
    call    puts
    movl    $0, (%esp)
    call    _exit

```

再編譯看看，

```

$ gcc -o hello.o hello.s
$ wc -c hello
6443 hello
$ echo "6523-6443" | bc    #僅僅減少了80個字節
80

```

不默認編譯並刪除掉無關節區和節區表

如果不採用默認編譯呢並且刪除掉對程序運行沒有影響的節區和節區表呢？

```

$ sed -i -e "s/main/_start/g" hello.s    #因為沒有初始化，所以得直接進入代碼，替換main為_start
$ as --32 -o hello.o hello.s
$ ld -melf_i386 -o hello hello.o --dynamic-linker /lib/ld-linux.so.2 -L /usr/lib -lc
$ ./hello
hello world!
$ wc -c hello
1812 hello

```

```
$ echo "6443-1812" | bc -l    #和之前的實驗類似，也減少了4k左右
4631
$ readelf -l hello | grep "\ [0-9][0-9]\ "
00
01      .interp
02      .interp .hash .dynsym .dynstr .gnu.version .gnu.version_r .rel.plt .plt .text
03      .dynamic .got.plt
04      .dynamic
$ strip -R .hash hello
$ strip -R .gnu.version hello
$ wc -c hello
1200 hello
$ sstrip hello
$ wc -c hello    #這個結果比之前的708（在刪除所有垃圾信息以後）個字節少了708-676，即32個字節
676 hello
$ ./hello
Hello World
```

容易發現這 32 字節可能跟節區 `.rodata` 有關係，因為剛才在鏈接完以後查看節區信息時，並沒有 `.rodata` 節區。

用系統調用取代庫函數

前面提到，實際上還可以不用動態連接庫中的 `printf` 函數，也不用直接調用 `_exit`，而是在彙編裡頭使用系統調用，這樣就可以去掉和動態連接庫關聯的內容。如果了解如何在彙編中使用系統調用，請參考資料 [9]。使用系統調用重寫以後得到如下代碼，

```
.LC0:
    .string "Hello World\xa\x0"
    .text
.global _start
_start:
    xorl    %eax, %eax
    movb    $4, %al                #eax = 4, sys_write(fd, addr, len)
    xorl    %ebx, %ebx
    incl    %ebx                  #ebx = 1, standard output
    movl    $.LC0, %ecx           #ecx = $.LC0, the address of string
    xorl    %edx, %edx
    movb    $13, %dl              #edx = 13, the length of .string
    int     $0x80
    xorl    %eax, %eax
    movl    %eax, %ebx            #ebx = 0
    incl    %eax                  #eax = 1, sys_exit
    int     $0x80
```

現在編譯就不再需要動態鏈接器 `ld-linux.so` 了，也不再需要鏈接任何庫。

```
$ as --32 -o hello.o hello.s
$ ld -melf_i386 -o hello hello.o
$ readelf -l hello

Elf file type is EXEC (Executable file)
Entry point 0x8048062
```

```
There are 1 program headers, starting at offset 52
```

```
Program Headers:
```

Type	Offset	VirtAddr	PhysAddr	FileSiz	MemSiz	Flg	Align
LOAD	0x000000	0x08048000	0x08048000	0x00007b	0x00007b	R E	0x1000

```
Section to Segment mapping:
```

```
Segment Sections...
```

```
00 .text
```

```
$ sstrip hello
```

```
$ ./hello #完全可以正常工作
```

```
Hello World
```

```
$ wc -c hello
```

```
123 hello
```

```
$ echo "676-123" | bc #相對於之前，已經只需要123個字節了，又減少了553個字節
```

```
553
```

可以看到效果很明顯，只剩下一個 `LOAD` 段，它對應 `.text` 節區。

把字符串作為參數輸入

不過是否還有辦法呢？把 `Hello World` 作為參數輸入，而不是硬編碼在文件中。所以如果處理參數的代碼少於 `Hello World` 字符串的長度，那麼就可以達到減少目標文件大小的目的。

先來看一個能夠打印程序參數的彙編語言程序，它來自參考資料[9]。

```
.text
.globl _start

_start:
    popl    %ecx          # argc
vnext:
    popl    %ecx          # argv
    test    %ecx, %ecx    # 空指針表明結束
    jz      exit
    movl    %ecx, %ebx
    xorl    %edx, %edx
strlen:
    movb    (%ebx), %al
    inc     %edx
    inc     %ebx
    test    %al, %al
    jnz     strlen
    movb    $10, -1(%ebx)
    movl    $4, %eax      # 系統調用號(sys_write)
    movl    $1, %ebx      # 文件描述符(stdout)
    int     $0x80
    jmp     vnext
exit:
    movl    $1,%eax       # 系統調用號(sys_exit)
    xorl    %ebx, %ebx    # 退出代碼
    int     $0x80
    ret
```

編譯看看效果，

```
$ as --32 -o args.o args.s
$ ld -melf_i386 -o args args.o
$ ./args "Hello World" #能夠打印輸入的字符串，不錯
./args
Hello World
$ sstrip args
$ wc -c args           #處理以後只剩下130字節
130 args
```

可以看到，這個程序可以接收用戶輸入的參數並打印出來，不過得到的可執行文件為 130 字節，比之前的 123 個字節還多了 7 個字節，看看還有改進麼？分析上面的代碼後，發現，原來的代碼有些地方可能進行優化，優化後得到如下代碼。

```
.global _start
_start:
    popl %ecx          #彈出argc
vnext:
    popl %ecx          #彈出argv[0]的地址
    test %ecx, %ecx    #空指針表明結束
    jz exit
    movl %ecx, %ebx     #複製字符串地址到ebx寄存器
    xorl %edx, %edx     #把字符串長度清零
strlen:
    #求輸入字符串的長度
    movb (%ebx), %al    #複製字符到al，以便判斷是否為字符串結束符\0
    inc %edx            #edx存放每個當前字符串的長度
    inc %ebx            #ebx存放每個當前字符的地址
    test %al, %al       #判斷字符串是否結束，即是否遇到\0
    jnz strlen
    movb $10, -1(%ebx)  #在字符串末尾插入一個換行符\0xa
    xorl %eax, %eax
    movb $4, %al        #eax = 4, sys_write(fd, addr, len)
    xorl %ebx, %ebx
    incl %ebx           #ebx = 1, standard output
    int $0x80
    jmp vnext
exit:
    xorl %eax, %eax
    movl %eax, %ebx     #ebx = 0
    incl %eax           #eax = 1, sys_exit
    int $0x80
```

再測試（記得先重新彙編、鏈接並刪除沒用的節區和節區表）。

```
$ wc -c hello
124 hello
```

現在只有 124 個字節，不過還是比 123 個字節多一個，還有什麼優化的辦法麼？

先來看看目前 `hello` 的功能，感覺不太符合要求，因為只需要打印 `Hello World`，所以不必處理所有的參數，僅僅需要接收並打印一個參數就可以。這樣的話，把 `jmp vnext`（2 字節）這個循環去掉，然後在打造史上最小可執行ELF文件(45字節)

第一個 `pop %ecx` 語句之前加一個 `pop %ecx` (1 字節) 語句就可以。

```
.global _start
_start:
    popl %ecx
    popl %ecx      #彈出argc[0]的地址
    popl %ecx      #彈出argv[1]的地址
    test %ecx, %ecx
    jz exit
    movl %ecx, %ebx
    xorl %edx, %edx
strlen:
    movb (%ebx), %al
    inc %edx
    inc %ebx
    test %al, %al
    jnz strlen
    movb $10, -1(%ebx)
    xorl %eax, %eax
    movb $4, %al
    xorl %ebx, %ebx
    incl %ebx
    int $0x80
exit:
    xorl %eax, %eax
    movl %eax, %ebx
    incl %eax
    int $0x80
```

現在剛好 123 字節，和原來那個代碼大小一樣，不過仔細分析，還是有減少代碼的餘地：因為在這個代碼中，用了一段額外的代碼計算字符串的長度，實際上如果僅僅需要打印 `Hello World`，那麼字符串的長度是固定的，即 12。所以這段代碼可去掉，與此同時測試字符串是否為空也就沒有必要（不過可能影響代碼健壯性！），當然，為了能夠在打印字符串後就換行，在串的末尾需要加一個回車（`$10`）並且設置字符串的長度為 `12+1`，即 13，

```
.global _start
_start:
    popl %ecx
    popl %ecx
    popl %ecx
    movb $10,12(%ecx) #在Hello World的結尾加一個換行符
    xorl %edx, %edx
    movb $13, %dl
    xorl %eax, %eax
    movb $4, %al
    xorl %ebx, %ebx
    incl %ebx
    int $0x80
    xorl %eax, %eax
    movl %eax, %ebx
    incl %eax
    int $0x80
```


再看看效果，

```
$ wc -c hello
111 hello
```

寄存器賦值重用

現在只剩下 111 字節，比剛才少了 12 字節。貌似到了極限？還有措施麼？

還有，仔細分析發現：系統調用 `sys_exit` 和 `sys_write` 都用到了 `eax` 和 `ebx` 寄存器，它們之間剛好有那麼一點巧合：

- `sys_exit` 調用時，`eax` 需要設置為 1，`ebx` 需要設置為 0。
- `sys_write` 調用時，`ebx` 剛好是 1。

因此，如果在 `sys_exit` 調用之前，先把 `ebx` 複製到 `eax` 中，再對 `ebx` 減一，則可減少兩個字節。

不過，因為標準輸入、標準輸出和標準錯誤都指向終端，如果往標準輸入寫入一些東西，它還是會輸出到標準輸出上，所以在上述代碼中如果在 `sys_write` 之前 `ebx` 設置為 0，那麼也可正常往屏幕上打印 `Hello World`，這樣的話，`sys_exit` 調用前就沒必要修改 `ebx`，而僅需把 `eax` 設置為 1，這樣就可減少 3 個字節。

```
.global _start
_start:
    popl %ecx
    popl %ecx
    popl %ecx
    movb $10,12(%ecx)
    xorl %edx, %edx
    movb $13, %dl
    xorl %eax, %eax
    movb $4, %al
    xorl %ebx, %ebx
    int $0x80
    xorl %eax, %eax
    incl %eax
    int $0x80
```

看看效果，

```
$ wc -c hello
108 hello
```

現在看一下純粹的指令還有多少？

```
$ readelf -h hello | grep Size
Size of this header:          52 (bytes)
Size of program headers:     32 (bytes)
```

```
Size of section headers:          0 (bytes)
$ echo "108-52-32" | bc
24
```

通過文件名傳遞參數

對於標準的 `main` 函數的兩個參數，文件名實際上作為第二個參數（數組）的第一個元素傳入，如果僅僅是為了打印一個字符串，那麼可以打印文件名本身。例如，要打印 `Hello World`，可以把文件名命名為 `Hello World` 即可。

這樣地話，代碼中就可以刪除掉一條 `popl` 指令，減少 1 個字節，變成 107 個字節。

```
.global _start
_start:
    popl %ecx
    popl %ecx
    movb $10,12(%ecx)
    xorl %edx, %edx
    movb $13, %dl
    xorl %eax, %eax
    movb $4, %al
    xorl %ebx, %ebx
    int $0x80
    xorl %eax, %eax
    incl %eax
    int $0x80
```

看看效果，

```
$ as --32 -o hello.o hello.s
$ ld -melf_i386 -o hello hello.o
$ sstrip hello
$ wc -c hello
107
$ mv hello "Hello World"
$ export PATH=./:$PATH
$ Hello\ World
Hello World
```

刪除非必要指令

在測試中發現，`edx`，`eax`，`ebx` 的高位即使不初始化，也常為 0，如果不考慮健壯性（僅這裡實驗用，實際使用中必須考慮健壯性），幾條 `xorl` 指令可以移除掉。

另外，如果只是為了演示打印字符串，完全可以不用打印換行符，這樣下來，代碼可以綜合優化成如下幾條指令：

```
.global _start
_start:
```

```

    popl %ecx    # argc
    popl %ecx    # argv[0]
    movb $5, %dl    # 設置字符串長度
    movb $4, %al    # eax = 4, 設置系統調用號, sys_write(fd, addr, len) : ebx, ecx, edx
    int $0x80
    movb $1, %al
    int $0x80

```

看看效果：

```

$ as --32 -o hello.o hello.s
$ ld -melf_i386 -o hello hello.o
$ sstrip hello
$ wc -c hello
96

```

合併代碼段、程序頭和文件頭（52字節）

把代碼段移入文件頭

純粹的指令只有 `96-84=12` 個字節了，還有辦法再減少目標文件的大小麼？如果看了參考資料 [\[1\]](#)，看樣子你又要蠢蠢欲動了：這 12 個字節是否可以插入到文件頭部或程序頭部？如果可以那是否意味著還可減少可執行文件的大小呢？現在來比較一下這三部分的十六進制內容。

```

$ hexdump -C hello -n 52      #文件頭(52bytes)
00000000  7f 45 4c 46 01 01 01 00  00 00 00 00 00 00 00 00 |.ELF.....|
00000010  02 00 03 00 01 00 00 00  54 80 04 08 34 00 00 00 |.....T...4...|
00000020  00 00 00 00 00 00 00 00  34 00 20 00 01 00 00 00 |.....4. ....|
00000030  00 00 00 00                |....|
00000034
$ hexdump -C hello -s 52 -n 32  #程序頭(32bytes)
00000034  01 00 00 00 00 00 00 00  00 80 04 08 00 80 04 08 |.....|
00000044  6c 00 00 00 6c 00 00 00  05 00 00 00 00 10 00 00 |1...1.....|
00000054
$ hexdump -C hello -s 84      #實際代碼部分(12bytes)
00000054  59 59 b2 05 b0 04 cd 80  b0 01 cd 80                |YY.....|
00000060

```

從上面結果發現 `ELF` 文件頭部和程序頭部還有好些空洞（0），是否可以把指令字節分散放入到那些空洞裡或者是直接覆蓋掉那些系統並不關心的內容？抑或是把代碼壓縮以後放入可執行文件中，並在其中實現一個解壓縮算法？還可以是通過一些代碼覆蓋率測試工具（`gcov`，`prof`）對你的代碼進行優化？

在繼續介紹之前，先來看一個 `dd` 工具，可以用來直接“編輯” `ELF` 文件，例如，

直接往指定位置寫入 `0xff`：

```

$ hexdump -C hello -n 16      # 寫入前，elf文件前16個字節
00000000  7f 45 4c 46 01 01 01 00  00 00 00 00 00 00 00 00 |.ELF.....|
00000010

```

```
$ echo -ne "\xff" | dd of=hello bs=1 count=1 seek=15 conv=notrunc # 把最後一個字節0覆蓋掉
1+0 records in
1+0 records out
1 byte (1 B) copied, 3.7349e-05 s, 26.8 kB/s
$ hexdump -C hello -n 16 # 寫入後果然被覆蓋
00000000 7f 45 4c 46 01 01 01 00 00 00 00 00 00 00 ff |.ELF.....|
00000010
```

- `seek=15` 表示指定寫入位置為第 15 個（從第 0 個開始）
- `conv=notrunc` 選項表示要保留寫入位置之後的內容，默認情況下會截斷。
- `bs=1` 表示一次讀/寫 1 個
- `count=1` 表示總共寫 1 次

覆蓋多個連續的值：

把第 12, 13, 14, 15 連續 4 個字節全部賦值為 `0xff`。

```
$ echo -ne "\xff\xff\xff\xff" | dd of=hello bs=1 count=4 seek=12 conv=notrunc
$ hexdump -C hello -n 16
00000000 7f 45 4c 46 01 01 01 00 00 00 00 00 ff ff ff ff |.ELF.....|
00000010
```

下面，通過往文件頭指定位置寫入 `0xff` 確認哪些部分對於可執行文件的執行是否有影響？這裡是逐步測試後發現依然能夠執行的情況：

```
$ hexdump -C hello
00000000 7f 45 4c 46 ff ff ff ff ff ff ff ff ff ff ff ff |.ELF.....|
00000010 02 00 03 00 ff ff ff ff 54 80 04 08 34 00 00 00 |.....T...4...|
00000020 ff ff ff ff ff ff ff ff 34 00 20 00 01 00 ff ff |.....4. ....|
00000030 ff ff ff ff 01 00 00 00 00 00 00 00 00 80 04 08 |.....|
00000040 00 80 04 08 60 00 00 00 60 00 00 00 05 00 00 00 |....`...`.....|
00000050 00 10 00 00 59 59 b2 05 b0 04 cd 80 b0 01 cd 80 |....YY.....|
00000060
```

可以發現，文件頭部分，有 30 個字節即使被篡改後，該可執行文件依然可以正常執行。這意味著，這 30 字節是可以寫入其他代碼指令字節的。而我們的實際代碼指令只剩下 12 個，完全可以直接移到前 12 個 `0xff` 的位置，即從第 4 個到第 15 個。

而代碼部分的起始位置，通過 `readelf -h` 命令可以看到：

```
$ readelf -h hello | grep "Entry"
Entry point address:          0x8048054
```

上面地址的最後兩位 `0x54=84` 就是代碼在文件中的偏移，也就是剛好從程序頭之後開始的，也就是用文件頭（52）+程序頭（32）個字節開始的 12 字節覆蓋到第 4 個字節開始的 12 字節內容即可。

上面的 `dd` 命令從 `echo` 命令獲得輸入，下面需要通過可執行文件本身獲得輸入，先把代碼部分移過去：

```
$ dd if=hello of=hello bs=1 skip=84 count=12 seek=4 conv=notrunc
12+0 records in
12+0 records out
12 bytes (12 B) copied, 4.9552e-05 s, 242 kB/s
$ hexdump -C hello
00000000  7f 45 4c 46 59 59 b2 05  b0 04 cd 80 b0 01 cd 80  |.ELFYY.....|
00000010  02 00 03 00 01 00 00 00  54 80 04 08 34 00 00 00  |.....T...4...|
00000020  00 00 00 00 00 00 00 00  34 00 20 00 01 00 00 00  |.....4. ....|
00000030  00 00 00 00 01 00 00 00  00 00 00 00 00 80 04 08  |.....|
00000040  00 80 04 08 60 00 00 00  60 00 00 00 05 00 00 00  |....`...`.....|
00000050  00 10 00 00 59 59 b2 05  b0 04 cd 80 b0 01 cd 80  |....YY.....|
00000060
```

接著把代碼部分截掉：

```
$ dd if=hello of=hello bs=1 count=1 skip=84 seek=84
0+0 records in
0+0 records out
0 bytes (0 B) copied, 1.702e-05 s, 0.0 kB/s
$ hexdump -C hello
00000000  7f 45 4c 46 59 59 b2 05  b0 04 cd 80 b0 01 cd 80  |.ELFYY.....|
00000010  02 00 03 00 01 00 00 00  54 80 04 08 34 00 00 00  |.....T...4...|
00000020  00 00 00 00 00 00 00 00  34 00 20 00 01 00 00 00  |.....4. ....|
00000030  00 00 00 00 01 00 00 00  00 00 00 00 00 80 04 08  |.....|
00000040  00 80 04 08 60 00 00 00  60 00 00 00 05 00 00 00  |....`...`.....|
00000050  00 10 00 00                                |....|
00000054
```

這個時候還不能執行，因為代碼在文件中的位置被移動了，相應地，文件頭中的 `Entry point address`，即文件入口地址也需要被修改為 `0x8048004`。

即需要把 `0x54` 所在的第 24 個字節修改為 `0x04`：

```
$ echo -ne "\x04" | dd of=hello bs=1 count=1 seek=24 conv=notrunc
1+0 records in
1+0 records out
1 byte (1 B) copied, 3.7044e-05 s, 27.0 kB/s
$ hexdump -C hello
00000000  7f 45 4c 46 59 59 b2 05  b0 04 cd 80 b0 01 cd 80  |.ELFYY.....|
00000010  02 00 03 00 01 00 00 00  04 80 04 08 34 00 00 00  |.....4...|
00000020  84 00 00 00 00 00 00 00  34 00 20 00 01 00 28 00  |.....4. ...(.|
00000030  05 00 02 00 01 00 00 00  00 00 00 00 00 80 04 08  |.....|
00000040  00 80 04 08 60 00 00 00  60 00 00 00 05 00 00 00  |....`...`.....|
00000050  00 10 00 00
```

修改後就可以執行了。

把程序頭移入文件頭

程序頭部分經過測試發現基本上都不能修改並且需要是連續的，程序頭有 32 個字節，而文件頭中連續的 `0xff` 可以被篡改的只有從第 46 個開始的 6 個了，另外，程序頭剛好是 `01 00` 開頭，而第 44, 45 個剛

好為 01 00，這樣地話，這兩個字節文件頭可以跟程序頭共享，這樣地話，程序頭就可以往文件頭裡頭移動 8 個字節了。

```
$ dd if=hello of=hello bs=1 skip=52 seek=44 count=32 conv=notrunc
```

再把最後 8 個沒用的字節刪除掉，保留 84-8=76 個字節：

```
$ dd if=hello of=hello bs=1 skip=76 seek=76
$ hexdump -C hello
00000000 7f 45 4c 46 59 59 b2 05 b0 04 cd 80 b0 01 cd 80 |.ELFYY.....|
00000010 02 00 03 00 01 00 00 00 04 80 04 08 34 00 00 00 |.....4...|
00000020 84 00 00 00 00 00 00 00 34 00 20 00 01 00 00 00 |.....4. ....|
00000030 00 00 00 00 00 80 04 08 00 80 04 08 60 00 00 00 |.....`...|
00000040 60 00 00 00 05 00 00 00 00 10 00 00                |`.....|
0000004c
```

另外，還需要把文件頭中程序頭的位置信息改為 44，即第 28 個字節，原來是 0x34，即 52 的位置。

```
$ echo "obase=16;ibase=10;44" | bc      # 先把44轉換是16進制的0x2C
2C
$ echo -ne "\x2C" | dd of=hello bs=1 count=1 seek=28 conv=notrunc      # 修改文件頭
1+0 records in
1+0 records out
1 byte (1 B) copied, 3.871e-05 s, 25.8 kB/s
$ hexdump -C hello
00000000 7f 45 4c 46 59 59 b2 05 b0 04 cd 80 b0 01 cd 80 |.ELFYY.....|
00000010 02 00 03 00 01 00 00 00 04 80 04 08 2c 00 00 00 |.....,....|
00000020 84 00 00 00 00 00 00 00 34 00 20 00 01 00 00 00 |.....4. ....|
00000030 00 00 00 00 00 80 04 08 00 80 04 08 60 00 00 00 |.....`...|
00000040 60 00 00 00 05 00 00 00 00 10 00 00                |`.....|
0000004c
```

修改後即可執行了，目前只剩下 76 個字節：

```
$ wc -c hello
76
```

在非連續的空間插入代碼

另外，還有 12 個字節可以放代碼，見 0xff 的地方：

```
$ hexdump -C hello
00000000 7f 45 4c 46 59 59 b2 05 b0 04 cd 80 b0 01 cd 80 |.ELFYY.....|
00000010 02 00 03 00 ff ff ff ff 04 80 04 08 2c 00 00 00 |.....,....|
00000020 ff ff ff ff ff ff ff ff 34 00 20 00 01 00 00 00 |.....4. ....|
00000030 00 00 00 00 00 80 04 08 00 80 04 08 60 00 00 00 |.....`...|
00000040 60 00 00 00 05 00 00 00 00 10 00 00                |`.....|
0000004c
```

不過因為空間不是連續的，需要用到跳轉指令作為跳板利用不同的空間。

例如，如果要利用後面的 `0xff` 的空間，可以把第 14, 15 位置的 `cd 80` 指令替換為一條跳轉指令，比如跳轉到第 20 個字節的位置，從跳轉指令之後的 16 到 20 剛好 4 個字節。

然後可以參考 [X86 指令編碼表](#)（也可以寫成彙編生成可執行文件後用 `hexdump` 查看），可以把 `jmp` 指令編碼為：`0xeb 0x04`。

```
$ echo -ne "\xeb\x04" | dd of=hello bs=1 count=2 seek=14 conv=notrunc
```

然後把原來位置的 `cd 80` 移動到第 20 個字節開始的位置：

```
$ echo -ne "\xcd\x80" | dd of=hello bs=1 count=2 seek=20 conv=notrunc
```

依然可以執行，類似地可以利用更多非連續的空間。

把程序頭完全合入文件頭

在閱讀參考資料 [1] 後，發現有更多深層次的探討，通過分析 Linux 系統對 `ELF` 文件頭部和程序頭部的解析，可以更進一步合併程序頭和文件頭。

該資料能夠把最簡的 `ELF` 文件（簡單返回一個數值）壓縮到 45 個字節，真地是非常極端的努力，思路可以充分借鑑。在充分理解原文的基礎上，我們進行更細緻地梳理。

首先對 `ELF` 文件頭部和程序頭部做更徹底的理解，並具體到每一個字節的含義以及在 Linux 系統下的實際解析情況。

先來看看 `readelf -a` 的結果：

```
$ as --32 -o hello.o hello.s
$ ld -melf_i386 -o hello hello.o
$ sstrip hello
$ readelf -a hello
ELF Header:
  Magic:   7f 45 4c 46 01 01 01 00 00 00 00 00 00 00 00 00
  Class:                                ELF32
  Data:                                      2's complement, little endian
  Version:                               1 (current)
  OS/ABI:                                UNIX - System V
  ABI Version:                           0
  Type:                                   EXEC (Executable file)
  Machine:                                Intel 80386
  Version:                                0x1
  Entry point address:                    0x8048054
  Start of program headers:               52 (bytes into file)
  Start of section headers:              0 (bytes into file)
  Flags:                                  0x0
  Size of this header:                    52 (bytes)
  Size of program headers:                32 (bytes)
  Number of program headers:              1
```

```

Size of section headers:      0 (bytes)
Number of section headers:    0
Section header string table index: 0

There are no sections in this file.

There are no sections to group in this file.

Program Headers:
Type           Offset      VirtAddr   PhysAddr   FileSiz MemSiz  Flg Align
LOAD           0x0000000 0x08048000 0x08048000 0x000060 0x000060 R E 0x1000

```

然後結合 `/usr/include/linux/elf.h` 分別做詳細註解。

首先是 52 字節的 `Elf` 文件頭的結構體 `elf32_hdr`：

變量類型	變量名	字節	說明	類型
unsigned char	e_ident[EI_NIDENT]	16	.ELF 前四個標識文件類型	必須
Elf32_Half	e_type	2	指定為可執行文件	必須
Elf32_Half	e_machine	2	指示目標機類型，例如：Intel 386	必須
Elf32_Word	e_version	4	當前只有一個版本存在，被忽略了	可篡改
Elf32_Addr	e_entry	4	代碼入口=加載地址(p_vaddr+.text偏移)	可調整
Elf32_Off	e_phoff	4	程序頭 Phdr 的偏移地址，用於加載代碼	必須
Elf32_Off	e_shoff	4	所有節區相關信息對文件執行無效	可篡改
Elf32_Word	e_flags	4	Intel 架構未使用	可篡改
Elf32_Half	e_ehsize	2	文件頭大小，Linux 沒做校驗	可篡改
Elf32_Half	e_phentsize	2	程序頭入口大小，新內核有用	必須
Elf32_Half	e_phnum	2	程序頭入口個數	必須
Elf32_Half	e_shentsize	2	所有節區相關信息對文件執行無效	可篡改
Elf32_Half	e_shnum	2	所有節區相關信息對文件執行無效	可篡改
Elf32_Half	e_shstrndx	2	所有節區相關信息對文件執行無效	可篡改

其次是 32 字節的程序頭（Phdr）的結構體 `elf32_phdr`：

變量類型	變量名	字節	說明	類型
Elf32_Word	p_type	4	標記為可加載段	必須
Elf32_Off	p_offset	4	相對程序頭的偏移地址	必須
Elf32_Addr	p_vaddr	4	加載地址, 0x0~0x80000000，頁對齊	可調整
Elf32_Addr	p_paddr	4	物理地址，暫時沒用	可篡改
Elf32_Word	p_filesz	4	加載的文件大小，>=real size	可調整
Elf32_Word	p_memsz	4	加載所需內存大小，>= p_filesz	可調整
Elf32_Word	p_flags	4	權限:read(4),exec(1), 其中一個暗指另外一個	可調整
Elf32_Word	p_align	4	PIC(共享庫需要)，對執行文件無效	可篡改

接著，咱們把 Elf 中的文件頭和程序頭部分可調整和可篡改的字節（52 + 32 = 84個）全部用特別的字體標記出來。

```
$ hexdump -C hello -n 84
```

```
00000000 7f 45 4c 46 01-01-01-00-00-00-00-00-00-00-00-00-00-00-00-00
00000010 02 00 03 00 01-00-00-00 54 80 04 08 34 00 00 00
00000020 84-00-00-00-00-00-00-00 34 00 20 00 01 00 28-00
00000030 05-00-02-00|01 00 00 00 00 00 00 00 00 80 04 08
00000040 00-80-04-08 60 00 00 00 60 00 00 00 05 00 00 00
00000050 00-10-00-00
00000054
```

上述 | 線之前為文件頭，之後為程序頭，之前的 000000xx 為偏移地址。

如果要把程序頭徹底合併進文件頭。從上述信息綜合來看，文件頭有 4 處必須保留，結合資料 [1]，經過對比發現，如果把第 4 行開始的程序頭往上平移 3 行，也就是：

```
00000000 ===== 01-01-01-00-00-00-00-00-00-00-00-00-00-00-00-00
00000010 02 00 03 00 01-00-00-00 54 80 04 08 34 00 00 00
00000020 84-00-00-00
00000030 ===== 01 00 00 00 00 00 00 00 00 80 04 08
00000040 00-80-04-08 60 00 00 00 60 00 00 00 05 00 00 00
00000050 00-10-00-00
00000054
```

把可直接合併的先合併進去，效果如下：

（文件頭）

```
00000000 ===== 01 00 00 00 00 00 00 00 00 80 04 08 (^ p_vaddr)
00000010 02 00 03 00 60-00-00-00 54 80 04 08 34 00 00 00
00000020 ===== ^ e_entry ^ e_phoff
```

（程序頭）

```
00000030 ===== 01 00 00 00 00 00 00 00 00 80 04 08 (^ p_vaddr)
```

```

00000040 02 00 03 00 60 00 00 00 60 00 00 00 05 00 00 00
00000050 ===== ^^ p_filesz ^^ p_memsz ^^p_flags
00000054

```

接著需要設法處理好可調整的 6 處，可以逐個解決，從易到難。

- 首先，合併 `e_phoff` 與 `p_flags`

在合併程序頭以後，程序頭的偏移地址需要修改為 4，即文件的第 4 個字節開始，也就是說 `e_phoff` 需要修改為 04。

而恰好，`p_flags` 的 `read(4)` 和 `exec(1)` 可以只選其一，所以，只保留 `read(4)` 即可，剛好也為 04。

合併後效果如下：

（文件頭）

```

00000000 ===== 01 00 00 00 00 00 00 00 00 80 04 08 (^ p_vaddr)
00000010 02 00 03 00 60 00 00 00 60 00 00 00 04 00 00 00
00000020 ===== ^^ e_entry

```

（程序頭）

```

00000030 ===== 01 00 00 00 00 00 00 00 00 80 04 08 (^ p_vaddr)
00000040 02 00 03 00 60 00 00 00 60 00 00 00 04 00 00 00
00000050 ===== ^^ p_filesz ^^ p_memsz
00000054

```

- 接下來，合併 `e_entry`，`p_filesz`，`p_memsz` 和 `p_vaddr`

從早前的分析情況來看，這 4 個變量基本都依賴 `p_vaddr`，也就是程序的加載地址，大體的依賴關係如下：

```

e_entry = p_vaddr + text offset = p_vaddr + 84 = p_vaddr + 0x54
p_memsz = e_entry
p_memsz >= p_filesz, 可以簡單取 p_filesz = p_memsz
p_vaddr = page alignment

```

所以，首先需要確定 `p_vaddr`，通過測試，發現 `p_vaddr` 最低必須有 64k，也就是 0x00010000，對應到 `hexdump` 的 `little endian` 導出結果，則為 00 00 01 00。

需要注意的是，為了儘量少了分配內存，我們選擇了一個最小的 `p_vaddr`，如果申請的內存太大，系統將無法分配。

接著，計算出另外 3 個變量：

```
e_entry = 0x00010000 + 0x54 = 0x00010054 即 54 00 01 00
p_memsz = 54 00 01 00
p_filesz = 54 00 01 00
```

完全合併後，修改如下：

（文件頭）

```
00000000 ===== 01 00 00 00 00 00 00 00 00 01 00

00000010 02 00 03 00 54 00 01 00 54 00 01 00 04 00 00 00

00000020 =====
```

好了，直接把內容燒入：

```
$ echo -ne "\x01\x00\x00\x00\x00\x00\x00\x00" \
"\x00\x00\x01\x00\x02\x00\x03\x00" \
"\x54\x00\x01\x00\x54\x00\x01\x00\x04" |\
tr -d ' ' |\
dd of=hello bs=1 count=25 seek=4 conv=notrunc
```

截掉代碼（ $52 + 32 + 12 = 96$ ）之後的所有內容，查看效果如下：

```
$ dd if=hello of=hello bs=1 count=1 skip=96 seek=96
$ hexdump -C hello -n 96
00000000 7f 45 4c 46 01 00 00 00 00 00 00 00 01 00 |.ELF.....|
00000010 02 00 03 00 54 00 01 00 54 00 01 00 04 00 00 00 |...T...T.....|
00000020 84 00 00 00 00 00 00 00 34 00 20 00 01 00 28 00 |.....4. ...(.|
00000030 05 00 02 00 01 00 00 00 00 00 00 00 80 04 08 00 |.....|
00000040 00 80 04 08 60 00 00 00 60 00 00 00 05 00 00 00 |....`...`.....|
00000050 00 10 00 00 59 59 b2 05 b0 04 cd 80 b0 01 cd 80 |....YY.....|
00000060
```

最後的工作是查看文件頭中剩下的可篡改的內容，並把代碼部分合併進去，程序頭已經合入，不再顯示。

```
00000000 7f 45 4c 46 01 00 00 00 00 00 00 00 01 00

00000010 02 00 03 00 54 00 01 00 54 00 01 00 04 00 00 00

00000020 84 00 00 00 00 00 00 00 34 00 20 00 01 00 28 00

00000030 05 00 02 00

00000040
```

```
00000050 ===== 59 59 b2 05 b0 04 cd 80 b0 01 cd 80
00000060
```

我們的指令有 12 字節，可篡改的部分有 14 個字節，理論上一定放得下，不過因為把程序頭搬進去以後，這 14 個字節並不是連續，剛好可以用上我們之前的跳轉指令處理辦法來解決。

並且，加入 2 個字節的跳轉指令，剛好是 14 個字節，恰好把代碼也完全包含進了文件頭。

在預留好跳轉指令位置的前提下，我們把代碼部分先合併進去：

```
00000000 7f 45 4c 46 01 00 00 00 00 00 00 00 00 01 00
00000010 02 00 03 00 54 00 01 00 54 00 01 00 04 00 00 00
00000020 59 59 b2 05 b0 04 00 00 34 00 20 00 01 00 cd 80
00000030 b0 01 cd 80
```

接下來設計跳轉指令，跳轉指令需要從所在位置跳到第一個 **cd 80** 所在的位置，相距 6 個字節，根據 `jmp` 短跳轉的編碼規範，可以設計為 `0xeb 0x06`，填完後效果如下：

```
00000000 7f 45 4c 46 01 00 00 00 00 00 00 00 00 01 00
00000010 02 00 03 00 54 00 01 00 54 00 01 00 04 00 00 00
00000020 59 59 b2 05 b0 04 eb 06 34 00 20 00 01 00 cd 80
00000030 b0 01 cd 80
```

用 `dd` 命令寫入，分兩段寫入：

```
$ echo -ne "\x59\x59\xb2\x05\xb0\x04\xeb\x06" | \
  dd of=hello bs=1 count=8 seek=32 conv=notrunc

$ echo -ne "\xcd\x80\xb0\x01\xcd\x80" | \
  dd of=hello bs=1 count=6 seek=46 conv=notrunc
```

代碼合入以後，需要修改文件頭中的代碼的偏移地址，即 `e_entry`，也就是要把原來的偏移 84 (0x54) 修改為現在的偏移，即 0x20。

```
$ echo -ne "\x20" | dd of=hello bs=1 count=1 seek=24 conv=notrunc
```

修改完以後恰好把合併進的程序頭 `p_memsz`，也就是分配給文件的內存改小了，`p_filesz` 也得相應改小。

```
$ echo -ne "\x20" | dd of=hello bs=1 count=1 seek=20 conv=notrunc
```

程序頭和代碼都已經合入，最後，把 52 字節之後的內容全部刪掉：

```
$ dd if=hello of=hello bs=1 count=1 skip=52 seek=52
$ hexdump -C hello
00000000  7f 45 4c 46 01 00 00 00 00 00 00 00 00 01 00 |.ELF.....|
00000010  02 00 03 00 20 00 01 00 20 00 01 00 04 00 00 00 |...T...T.....|
00000020  59 59 b2 05 b0 04 eb 06 34 00 20 00 01 00 cd 80 |YY.....4. ....|
00000030  b0 01 cd 80
$ export PATH=./:$PATH
$ hello
hello
```

代碼和程序頭部分合並進文件頭的彙總情況：

```
00000000 7f 45 4c 46 01 00 00 00 00 00 00 00 00 01 00
00000010 02 00 03 00 20 00 01 00 20 00 01 00 04 00 00 00
00000020 59 59 b2 05 b0 04 eb 06 34 00 20 00 01 00 cd 80
00000030 b0 01 cd 80
```

最後，我們的成績是：

```
$ wc -c hello
52
```

史上最小的可打印 `Hello World`（注：要完全打印得把代碼中的5該為13，並且把文件名該為該字符串）的 `Elf` 文件是 52 個字節。打破了資料 [1] 作者創造的紀錄：

```
$ cd ELFKickers/tiny/
$ wc -c hello
59 hello
```

需要特別提到的是，該作者創造的最小可執行 `Elf` 是 45 個字節。

但是由於那個程序只能返回一個數值，代碼更簡短，剛好可以直接嵌入到文件頭中間，而文件末尾的 7 個 0 字節由於 Linux 加載時會自動填充，所以可以刪掉，所以最終的文件大小是 52 - 7 即 45 個字節。

其大體可實現如下：

```
.global _start
_start:
    mov $42, %bl    # 設置返回值為 42
    xor %eax, %eax  # eax = 0
    inc %eax        # eax = eax+1, 設置系統調用號, sys_exit()
    int $0x80
```

保存為 ret.s，編譯和執行效果如下：

```
$ as --32 -o ret.o ret.s
$ ld -melf_i386 -o ret ret.o
$ ./ret
42
```

代碼字節數可這麼查看：

```
$ ld -melf_i386 --oformat=binary -o ret.bin ret.o
$ hexdump -C ret.bin
00000000 b3 2a 31 c0 40 cd 80
00000007
```

這裡只有 7 條指令，剛好可以嵌入，而最後的 6 個字節因為可篡改為 0，並且內核可自動填充 0，所以乾脆可以連續刪掉最後 7 個字節的 0：

```
00000000 7f 45 4c 46 01 00 00 00 00 00 00 00 00 01 00
00000010 02 00 03 00 54 00 01 00 54 00 01 00 04 00 00 00
00000020 b3 2a 31 c0 40 cd 80 00 34 00 20 00 01 00 00 00
00000030 00 00 00 00
```

可以直接用已經合併好程序頭的 `hello` 來做實驗，這裡一併截掉最後的 7 個 0 字節：

```
$ cp hello ret
$ echo -ne "\xb3\x2a\x31\xc0\x40\xcd\x80" | \
    dd of=ret bs=1 count=8 seek=32 conv=notrunc
$ dd if=ret of=hello bs=1 count=1 skip=45 seek=45
$ hexdump -C hello
00000000 7f 45 4c 46 01 00 00 00 00 00 00 00 00 01 00 |.ELF.....|
00000010 02 00 03 00 20 00 01 00 20 00 01 00 04 00 00 00 |.... ... ..|
00000020 b3 2a 31 c0 40 cd 80 06 34 00 20 00 01          |.*1.@...4. ..|
0000002d
$ wc -c ret
45 ret
$ ./ret
$ echo $?
42
```

如果想快速構建該 `Elf` 文件，可以直接使用下述 Shell 代碼：

```
#!/bin/bash
#
# generate_ret_elf.sh -- Generate a 45 bytes Elf file
#
# $ bash generate_ret_elf.sh
# $ chmod a+x ret.elf
```

```
# $ ./ret.elf
# $ echo $?
# 42
#

ret="\x7f\x45\x4c\x46\x01\x00\x00\x00"
ret=${ret}"\x00\x00\x00\x00\x00\x00\x01\x00"
ret=${ret}"\x02\x00\x03\x00\x20\x00\x01\x00"
ret=${ret}"\x20\x00\x01\x00\x04\x00\x00\x00"
ret=${ret}"\xb3\x2a\x31\xc0\x40xcd\x80\x06"
ret=${ret}"\x34\x00\x20\x00\x01"

echo -ne $ret > ret.elf
```

又或者是直接參照資料 [\[1\]](#) 的 `tiny.asm` 就行了，其代碼如下：

```
; ret.asm

BITS 32

org 0x00010000

    db 0x7F, "ELF"          ; e_ident
    dd 1                    ; p_type
    dd 0                    ; p_offset
    dd $$                   ; p_vaddr
    dw 2                    ; e_type   ; p_paddr
    dw 3                    ; e_machine
    dd _start               ; e_version ; p_filesz
    dd _start               ; e_entry  ; p_memsz
    dd 4                    ; e_phoff  ; p_flags

_start:
    mov bl, 42              ; e_shoff  ; p_align
    xor eax, eax
    inc eax                 ; e_flags
    int 0x80
    db 0
    dw 0x34                 ; e_ehsize
    dw 0x20                 ; e_phentsize
    db 1                    ; e_phnum
                           ; e_shentsize
                           ; e_shnum
                           ; e_shstrndx

filesize equ $ - $$
```

編譯和運行效果如下：

```
$ nasm -f bin -o ret ret.asm
$ chmod +x ret
$ ./ret ; echo $?
42
$ wc -c ret
45 ret
```

下面也給一下本文精簡後的 `hello` 的 `nasm` 版本：

```
; hello.asm

BITS 32

        org     0x00010000

        db      0x7F, "ELF"           ; e_ident
        dd      1                               ; p_type
        dd      0                               ; p_offset
        dd      $$                               ; p_vaddr
        dw      2                               ; e_type
        dw      3                               ; e_machine
        dd      _start                     ; e_version
        dd      _start                     ; e_entry
        dd      4                               ; e_phoff
_start:
        pop     ecx       ; argc
        pop     ecx       ; argv[0]
        mov     dl, 5      ; str len
        mov     al, 4      ; sys_write(fd, addr, len) : ebx, ecx, edx
        jmp     _next      ; jump to next part of the code
        dw      0x34        ; e_ehsize
        dw      0x20        ; e_phentsize
        dw      1           ; e_phnum
_next:
        int     0x80        ; syscall
        mov     al, 1       ; eax=1, sys_exit
        int     0x80        ; syscall

filesize    equ     $ - $$
```

編譯和用法如下：

```
$ ./nasm -f bin -o hello hello.asm a.
$ chmod a+x hello
$ export PATH=./:$PATH
$ hello
hello
$ wc -c hello
52
```

經過一番努力，`AT&T` 的完整 `binary` 版本如下：

```
# hello.s
#
# as --32 -o hello.o hello.s
# ld -melf_i386 --oformat=binary -o hello hello.o
#

.file "hello.s"
.global _start, _load
.equ    LOAD_ADDR, 0x00010000    # Page aligned load addr, here 64k
.equ    E_ENTRY, LOAD_ADDR + (_start - _load)
```



```

.equ    P_MEM_SZ, E_ENTRY
.equ    P_FILE_SZ, P_MEM_SZ

_load:
.byte   0x7F
.ascii  "ELF"                # e_ident, Magic Number
.long   1                    # p_type, loadable seg
.long   0                    # p_offset
.long   LOAD_ADDR            # p_vaddr
.word   2                    # e_type, exec # p_paddr
.word   3                    # e_machine, Intel 386 target
.long   P_FILE_SZ            # e_version   # p_filesz
.long   E_ENTRY              # e_entry     # p_memsz
.long   4                    # e_phoff     # p_flags, read(exec)
.text

_start:
popl    %ecx    # argc        # e_shoff     # p_align
popl    %ecx    # argv[0]
mov     $5, %dl # str len     # e_flags
mov     $4, %al # sys_write(fd, addr, len) : ebx, ecx, edx
jmp     next    # jump to next part of the code
.word   0x34     # e_ehsize = 52
.word   0x20     # e_phentsize = 32
.word   1        # e_phnum = 1
.text

_next:  int     $0x80    # syscall        # e_shentsize
        mov     $1, %al # eax=1,sys_exit # e_shnum
        int     $0x80    # syscall        # e_shstrndx

```

編譯和運行效果如下：

```

$ as --32 -o hello.o hello.s
$ ld -melf_i386 --oformat=binary -o hello hello.o
$ export PATH=./:$PATH
$ hello
hello
$ wc -c hello
52 hello

```

注：編譯時務必要加 `--oformat=binary` 參數，以便直接跟源文件構建一個二進制的 `Elf` 文件，否則會被 `ld` 默認編譯，自動填充其他內容。

彙編語言極限精簡之道（45字節）

經過上述努力，我們已經完全把程序頭和代碼都融入了 52 字節的 `Elf` 文件頭，還可以再進一步嗎？

基於資料一，如果再要努力，只能設法把 `Elf` 末尾的 7 個 0 字節刪除，但是由於代碼已經把 `Elf` 末尾的 7 字節 0 字符都填滿了，所以要想在這一塊努力，只能繼續壓縮代碼。

繼續研究下代碼先：

```
.global _start
```

```

_start:
    popl %ecx    # argc
    popl %ecx    # argv[0]
    movb $5, %dl  # 設置字符串長度
    movb $4, %al  # eax = 4, 設置系統調用號, sys_write(fd, addr, len) : ebx, ecx, edx
    int $0x80
    movb $1, %al
    int $0x80

```

查看對應的編碼：

```

$ as --32 -o hello.o hello.s
$ ld -melf_i386 -o hello hello.o --oformat=binary
$ hexdump -C hello
00000000  59 59 b2 05 b0 04 cd 80  b0 01 cd 80          |YY.....|
0000000c

```

每條指令對應的編碼映射如下：

指令	編碼	說明
popl %ecx	59	argc
popl %ecx	59	argv[0]
movb \$5, %dl	b2 05	設置字符串長度
movb \$4, %al	b0 04	eax = 4, 設置系統調用號, sys_write(fd, addr, len) : ebx, ecx, edx
int \$0x80	cd 80	觸發系統調用
movb \$1, %al	b0 01	eax = 1, sys_exit
int \$0x80	cd 80	觸發系統調用

可以觀察到：

- popl 的指令編碼最簡潔。
- int \$0x80 重複了兩次，而且每條都佔用了 2 字節
- movb 每條都佔用了 2 字節
- eax 有兩次賦值，每次佔用了 2 字節
- popl %ecx 取出的 argc 並未使用

根據之前通過參數傳遞字符串的想法，咱們是否可以考慮通過參數來設置變量呢？

理論上，傳入多個參數，通過 pop 彈出來賦予 eax, ecx 即可，但是實際上，由於從參數棧裡頭 pop 出來的參數是參數的地址，並不是參數本身，所以該方法行不通。

不過由於第一個參數取出的是數字，並且是參數個數，而且目前的那條 popl %ecx 取出的 argc 並沒有使用，那麼剛好可以用來設置 eax，替換後如下：

```

.global _start
_start:
    popl %eax    # eax = 4, 設置系統調用號, sys_write(fd, addr, len) : ebx, ecx, edx

```

```

    popl %ecx    # argv[0], 字符串
    movb $5, %dl # 設置字符串長度
    int $0x80
    movb $1, %al # eax = 1, sys_exit
    int $0x80

```

這裡需要傳入 4 個參數，即讓棧彈出的第一個值，也就是參數個數賦予 `eax`，也就是：`hello 5 4 1`。

難道我們只能把該代碼優化到 10 個字節？

巧合地是，當偶然改成這樣的情況下，該代碼還能正常返回。

```

.global _start
_start:
    popl %eax    # eax = 4, 設置系統調用號, sys_write(fd, addr, len) : ebx, ecx, edx
    popl %ecx    # argv[0], 字符串
    movb $5, %dl # 設置字符串長度
    int $0x80
    loop _start  # 觸發系統退出

```

注：上面我們使用了 `loop` 指令而不是 `jmp` 指令，因為 `jmp _start` 產生的代碼更長，而 `loop _start` 指令只有兩個字節。

這裡相當於刪除了 `movb $1, %al`，最後我們獲得了 8 個字節。但是這裡為什麼能夠工作呢？

經過分析 `arch/x86/ia32/ia32entry.S`，我們發現當系統調用號無效時（超過系統調用入口個數），內核為了健壯考慮，必須要處理這類異常，並通過 `ia32_badsys` 讓系統調用正常返回。

這個可以這樣驗證：

```

.global _start
_start:
    popl %eax    # argc, eax = 4, 設置系統調用號, sys_write(fd, addr, len) : ebx, ecx, edx
    popl %ecx    # argv[0], 文件名
    mov $5, %dl  # argv[1], 字符串長度
    int $0x80
    mov $0xffffffff, %eax # 設置一個非法調用號用於退出
    int $0x80

```

那最後的結果是，我們產生了一個可以正常打印字符串，大小隻有 45 字節的 `Elf` 文件，最終的結果如下：

```

# hello.s
#
# $ as --32 -o hello.o hello.s
# $ ld -melf_i386 --oformat=binary -o hello hello.o
# $ export PATH=./:$PATH
# $ hello 0 0 0
# hello
#

```

```

.file "hello.s"
.global _start, _load
.equ    LOAD_ADDR, 0x00010000    # Page aligned load addr, here 64k
.equ    E_ENTRY, LOAD_ADDR + (_start - _load)
.equ    P_MEM_SZ, E_ENTRY
.equ    P_FILE_SZ, P_MEM_SZ

_load:
.byte   0x7F
.ascii  "ELF"                # e_ident, Magic Number
.long   1                    # p_type, loadable seg
.long   0                    # p_offset
.long   LOAD_ADDR            # p_vaddr
.word   2                    # e_type, exec # p_paddr
.word   3                    # e_machine, Intel 386 target
.long   P_FILE_SZ            # e_version # p_filesz
.long   E_ENTRY              # e_entry   # p_memsz
.long   4                    # e_phoff   # p_flags, read(exec)
.text

_start:
popl    %eax    # argc    # e_shoff    # p_align
          # 4 args, eax = 4, sys_write(fd, addr, len) : ebx, ecx, edx
          # set 2nd eax = random addr to trigger bad syscall for exit
popl    %ecx    # argv[0]
mov     $5, %dl # str len # e_flags
int     $0x80
loop    _start  # loop to popup a random addr as a bad syscall number
.word   0x34    # e_ehsize = 52
.word   0x20    # e_phentsize = 32
.byte   1       # e_phnum = 1, remove trailing 7 bytes with 0 value
          # e_shentsize
          # e_shnum
          # e_shstrndx

```

效果如下：

```

$ as --32 -o hello.o hello.s
$ ld -melf_i386 -o hello hello.o --oformat=binary
$ export PATH=./:$PATH
$ hello 0 0 0
hello
$ wc -c hello
45 hello

```

到這裡，我們獲得了史上最小的可以打印字符串的 `ELF` 文件，是的，只有 45 個字節。

小結

到這裡，關於可執行文件的討論暫且結束，最後來一段小小的總結，那就是我們設法去減少可執行文件大小的意義？

實際上，通過這樣一個討論深入到了很多技術的細節，包括可執行文件的格式、目標代碼鏈接的過程、Linux 下彙編語言開發等。與此同時，可執行文件大小的減少本身對嵌入式系統非常有用，如果刪除那些對

程序運行沒有影響的節區和節區表將減少目標系統的大小，適應嵌入式系統資源受限的需求。除此之外，動態連接庫中的很多函數可能不會被使用到，因此也可以通過某種方式剔除 [8]，[10]。

或許，你還會發現更多有趣的意義，歡迎給我發送郵件，一起討論。

參考資料

- [A Whirlwind Tutorial on Creating Really Teensy ELF Executables for Linux](#)
- [UNIX/LINUX 平臺可執行文件格式分析](#)
- [C/C++ 程序編譯步驟詳解](#)
- [The Linux GCC HOW TO](#)
- [ELF: From The Programmer's Perspective](#)
- [Understanding ELF using readelf and objdump](#)
- [Dissecting shared libraries](#)
- [嵌入式 Linux 小型化技術](#)
- [Linux 彙編語言開發指南](#)
- [Library Optimizer](#)
- [ELF file format and ABI: \[1\], \[2\], \[3\], \[4\]](#)
- [i386 指令編碼表](#)

代碼測試、調試與優化

- 前言
- 代碼測試
 - 測試程序的運行時間 time
 - 函數調用關係圖 calltree
 - 性能測試工具 gprof & kprof
 - 代碼覆蓋率測試 gcov & ggcov
 - 內存訪問越界 catchsegv, libSegFault.so
 - 緩衝區溢出 libsafe.so
 - 內存洩露 Memwatch, Valgrind, mtrace
- 代碼調試
 - 靜態調試：printf + gcc -D (打印程序中的變量)
 - 交互式的調試（動態調試）：gdb（支持本地和遠程）/ald（彙編指令級別的調試）
 - 嵌入式系統調試方法 gdbserver/gdb
 - 彙編代碼的調試 ald
 - 實時調試：gdb tracepoint
 - 調試內核
- 代碼優化
- 參考資料

前言

代碼寫完以後往往要做測試（或驗證）、調試，可能還要優化。

- 關於測試（或驗證）

通常對應著兩個英文單詞 `Verification`和 `Validation`，在資料 [1] 中有關於這個的定義和一些深入的討論，在資料 [2] 中，很多人給出了自己的看法。但是正如資料 [2] 提到的：

The differences between verification and validation are unimportant except to the theorist; practitioners use the term V&V to refer to all of the activities that are aimed at making sure the software will function as required.

所以，無論測試（或驗證）目的都是為了讓軟件的功能能夠達到需求。測試和驗證通常會通過一些形式化（貌似可以簡單地認為有數學根據的）或者非形式化的方法去驗證程序的功能是否達到要求。

- 關於調試

而調試對應英文 `debug`，`debug` 叫“驅除害蟲”，也許一個軟件的功能達到了要求，但是可能會在測試或者是正常運行時出現異常，因此需要處理它們。

- 關於優化

`debug` 是為了保證程序的正確性，之後就需要考慮程序的執行效率，對於存儲資源受限的嵌入式系統，程序的大小也可能是優化的對象。

很多理論性的東西實在沒有研究過，暫且不說吧。這裡只是想把一些需要動手實踐的東西先且記錄和總結一下，另外很多工具在這裡都有提到和羅列，包括 Linux 內核調試相關的方法和工具。關於更詳細更深入的內容還是建議直接看後面的參考資料為妙。

下面的所有演示在如下環境下進行：

```
$ uname -a
Linux falcon 2.6.22-14-generic #1 SMP Tue Feb 12 07:42:25 UTC 2008 i686 GNU/Linux
$ echo $SHELL
/bin/bash
$ /bin/bash --version | grep bash
GNU bash, version 3.2.25(1)-release (i486-pc-linux-gnu)
$ gcc --version | grep gcc
gcc (GCC) 4.1.3 20070929 (prerelease) (Ubuntu 4.1.2-16ubuntu2)
$ cat /proc/cpuinfo | grep "model name"
model name      : Intel(R) Pentium(R) 4 CPU 2.80GHz
```

代碼測試

代碼測試有很多方面，例如運行時間、函數調用關係圖、代碼覆蓋度、性能分析（Profiling）、內存訪問越界（Segmentation Fault）、緩衝區溢出（Stack Smashing 合法地進行非法的內存訪問？所以很危險）、內存洩露（Memory Leak）等。

測試程序的運行時間 time

Shell 提供了內置命令 `time` 用於測試程序的執行時間，默認顯示結果包括三部分：實際花費時間（real time）、用戶空間花費時間（user time）和內核空間花費時間（kernel time）。

```
$ time pstree 2>&1 >/dev/null

real    0m0.024s
user    0m0.008s
sys     0m0.004s
```

`time` 命令給出了程序本身的運行時間。這個測試原理非常簡單，就是在程序運行（通過 `system` 函數執行）前後記錄了系統時間（用 `times` 函數），然後進行求差就可以。如果程序運行時間很短，運行一次看不到效果，可以考慮採用測試紙片厚度的方法進行測試，類似把很多紙張疊到一起來測試紙張厚度一樣，我們可以讓程序運行很多次。

如果程序運行時間太長，執行效率很低，那麼得考慮程序內部各個部分的執行情況，從而對代碼進行可能的優化。具體可能會考慮到這兩點：

對於 C 語言程序而言，一個比較宏觀的層次性的輪廓（profile）是函數調用圖、函數內部的條件分支構成的語句塊，然後就是具體的語句。把握好這樣一個輪廓後，就可以有針對性地去關注程序的各個部分，包括哪些函數、哪些分支、哪些語句最值得關注（執行次數越多越值得優化，術語叫 hotspots）。

對於 Linux 下的程序而言，程序運行時涉及到的代碼會涵蓋兩個空間，即用戶空間和內核空間。由於這兩個空間涉及到地址空間的隔離，在測試或調試時，可能涉及到兩個空間的工具。前者絕大多數是基於 `gcc`

的特定參數和系統的 `ptrace` 調用，而後者往往實現為內核的補丁，它們在原理上可能類似，但實際操作時後者顯然會更麻煩，不過如果你不去 hack 內核，那麼往往無須關心後者。

函數調用關係圖 `calltree`

`calltree` 可以非常簡單方便地反應一個項目的函數調用關係圖，雖然諸如 `gprof` 這樣的工具也能做到，不過如果僅僅要得到函數調用圖，`calltree` 應該是更好的選擇。如果要產生圖形化的輸出可以使用它的 `-dot` 參數。從[這裡](#)可以下載到它。

這裡是一份基本用法演示結果：

```
$ calltree -b -np -m *.c
main:
|   close
|   commitchanges
|   |   err
|   |   |   fprintf
|   |   ferr
|   |   ftruncate
|   |   lseek
|   |   write
|   ferr
|   getmemorysize
|   modifyheaders
|   open
|   printf
|   readelfheader
|   |   err
|   |   |   fprintf
|   |   ferr
|   |   read
|   readphdrtable
|   |   err
|   |   |   fprintf
|   |   ferr
|   |   malloc
|   |   read
|   truncatezeros
|   |   err
|   |   |   fprintf
|   |   ferr
|   |   lseek
|   |   read$
```

這樣一份結果對於“反向工程”應該會很有幫助，它能夠呈現一個程序的大體結構，對於閱讀和分析源代碼來說是一個非常好的選擇。雖然 `cscope` 和 `ctags` 也能夠提供一個函數調用的“即時”（在編輯 Vim 的過程中進行調用）視圖（view），但是 `calltree` 卻給了我們一個宏觀的視圖。

不過這樣一個視圖只涉及到用戶空間的函數，如果想進一步給出內核空間的宏觀視圖，那麼 `strace`，`KFT` 或者 `Ftrace` 就可以發揮它們的作用。另外，該視圖也沒有給出庫中的函數，如果要跟蹤呢？需要 `ltrace` 工具。

另外發現 `calltree` 僅僅給出了一個程序的函數調用視圖，而沒有告訴我們各個函數的執行次數等情況。

如果要關注這些呢？我們有 `gprof`。

性能測試工具 `gprof` & `kprof`

參考資料[3]詳細介紹了這個工具的用法，這裡僅挑選其中一個例子來演示。`gprof` 是一個命令行的工具，而 KDE 桌面環境下的 `kprof` 則給出了圖形化的輸出，這裡僅演示前者。

首先來看一段代碼（來自資料[3]），算 `Fibonacci` 數列的，

```
#include <stdio.h>

int fibonacci(int n);

int main (int argc, char **argv)
{
    int fib;
    int n;

    for (n = 0; n <= 42; n++) {
        fib = fibonacci(n);
        printf("fibonnaci(%d) = %d\n", n, fib);
    }

    return 0;
}

int fibonacci(int n)
{
    int fib;

    if (n <= 0) {
        fib = 0;
    } else if (n == 1) {
        fib = 1;
    } else {
        fib = fibonacci(n - 1) + fibonacci(n - 2);
    }

    return fib;
}
```

通過 `calltree` 看看這段代碼的視圖，

```
$ calltree -b -np -m *.c
main:
|  fibonacci
|  |  fibonacci ....
|  printf
```

可以看出程序主要涉及到一個 `fibonacci` 函數，這個函數遞歸調用自己。為了能夠使用 `gprof`，需要編譯時加上 `-pg` 選項，讓 `Gcc` 加入相應的調試信息以便 `gprof` 能夠產生函數執行情況的報告。

```
$ gcc -pg -o fib fib.c
$ ls
fib  fib.c
```

運程序並查看執行時間，

```
$ time ./fib
fibonnaci(0) = 0
fibonnaci(1) = 1
fibonnaci(2) = 1
fibonnaci(3) = 2
...
fibonnaci(41) = 165580141
fibonnaci(42) = 267914296

real    1m25.746s
user    1m9.952s
sys     0m0.072s
$ ls
fib  fib.c  gmon.out
```

上面僅僅選取了部分執行結果，程序運行了 1 分多鐘，代碼運行以後產生了一個 `gmon.out` 文件，這個文件可以用於 `gprof` 產生一個相關的性能報告。

```
$ gprof -b ./fib gmon.out
Flat profile:
```

Each sample counts as 0.01 seconds.

% time	cumulative seconds	self seconds	calls	self ms/call	total ms/call	name
96.04	14.31	14.31	43	332.80	332.80	fibonacci
4.59	14.99	0.68				main

Call graph

granularity: each sample hit covers 2 byte(s) for 0.07% of 14.99 seconds

index	% time	self	children	called	name
					<spontaneous>
[1]	100.0	0.68	14.31		main [1]
		14.31	0.00	43/43	fibonacci [2]

				2269806252	fibonacci [2]
		14.31	0.00	43/43	main [1]
[2]	95.4	14.31	0.00	43+2269806252	fibonacci [2]
				2269806252	fibonacci [2]

Index by function name

[2] fibonacci	[1] main
---------------	----------

從這份結果中可觀察到程序中每個函數的執行次數等情況，從而找出值得修改的函數。在對某些部分修改之後，可以再次比較程序運行時間，查看優化結果。另外，這份結果還包含一個特別有用的東西，那就是程序的動態函數調用情況，即程序運行過程中實際執行過的函數，這和 `calltree` 產生的靜態調用樹有所不同，它能夠反應程序在該次執行過程中的函數調用情況。而如果想反應程序運行的某一時刻調用過的函數，可以考慮採用 `gdb` 的 `backtrace` 命令。

類似測試紙片厚度的方法，`gprof` 也提供了一個統計選項，用於對程序的多次運行結果進行統計。另外，`gprof` 有一個 KDE 下圖形化接口 `kprof`，這兩部分請參考資料[3]。

對於非 KDE 環境，可以使用 `Gprof2Dot` 把 `gprof` 輸出轉換成圖形化結果。

關於 `dot` 格式的輸出，也可以考慮通過 `dot` 命令把結果轉成 `jpg` 等格式，例如：

```
$ dot -Tjpg test.dot -o test.jpg
```

`gprof` 雖然給出了函數級別的執行情況，但是如果想關心具體哪些條件分支被執行到，哪些語句沒有被執行，該怎麼辦？

代碼覆蓋率測試 `gcov` & `ggcov`

如果要使用 `gcov`，在編譯時需要加上這兩個選項 `-fprofile-arcs -ftest-coverage`，這裡直接用之前的 `fib.c` 做演示。

```
$ ls
fib.c
$ gcc -fprofile-arcs -ftest-coverage -o fib fib.c
$ ls
fib  fib.c  fib.gcno
```

運行程序，並通過 `gcov` 分析代碼的覆蓋度：

```
$ ./fib
$ gcov fib.c
File 'fib.c'
Lines executed:100.00% of 12
fib.c:creating 'fib.c.gcov'
```

12 行代碼 100% 被執行到，再查看分支情況，

```
$ gcov -b fib.c
File 'fib.c'
Lines executed:100.00% of 12
Branches executed:100.00% of 6
Taken at least once:100.00% of 6
Calls executed:100.00% of 4
fib.c:creating 'fib.c.gcov'
```

發現所有函數，條件分支和語句都被執行到，說明代碼的覆蓋率很高，不過資料[3] `gprof` 的演示顯示代碼的覆蓋率高並不一定說明代碼的性能就好，因為那些被覆蓋到的代碼可能能夠被優化成性能更高的代碼。那到底哪些代碼值得被優化呢？執行次數最多的，另外，有些分支雖然都覆蓋到了，但是這個分支的位置可能並不是理想的，如果一個分支的內容被執行的次數很多，那麼把它作為最後一個分支的話就會浪費很多不必要的比較時間。因此，通過覆蓋率測試，可以嘗試著剔除那些從未執行過的代碼或者把那些執行次數較多的分支移動到較早的條件分支裡頭。通過性能測試，可以找出那些值得優化的函數、分支或者是語句。

如果使用 `-fprofile-arcs -ftest-coverage` 參數編譯完代碼，可以接著用 `-fbranch-probabilities` 參數對代碼進行編譯，這樣，編譯器就可以對根據代碼的分支測試情況進行優化。

```
$ wc -c fib
16333 fib
$ ls fib.gcda #確保fib.gcda已經生成，這個是運行fib後的結果
fib.gcda
$ gcc -fbranch-probabilities -o fib fib.c #再次運行
$ wc -c fib
6604 fib
$ time ./fib
...
real    0m21.686s
user    0m18.477s
sys     0m0.008s
```

可見代碼量減少了，而且執行效率會有所提高，當然，這個代碼效率的提高可能還跟其他因素有關，比如 `Gcc` 還優化了一些跟平臺相關的指令。

如果想看看代碼中各行被執行的情況，可以直接看 `fib.c.gcov` 文件。這個文件的各列依次表示執行次數、行號和該行的源代碼。次數有三種情況，如果一直沒有執行，那麼用 `####` 表示；如果該行是註釋、函數聲明等，用 `-` 表示；如果是純粹的代碼行，那麼用執行次數表示。這樣我們就可以直接分析每一行的執行情況。

`gcov` 也有一個圖形化接口 `ggcov`，是基於 `gtk+` 的，適合 `Gnome` 桌面的用戶。

現在都已經關注到代碼行了，實際上優化代碼的前提是保證代碼的正確性，如果代碼還有很多 `bug`，那麼先要 `debug`。不過下面的這些“`bug`”用普通的工具確實不太方便，雖然可能，不過這裡還是把它們歸結為測試的內容，並且這裡剛好承接上 `gcov` 部分，`gcov` 能夠測試到每一行的代碼覆蓋情況，而無論是內存訪問越界、緩衝區溢出還是內存洩露，實際上是發生在具體的代碼行上的。

內存訪問越界 `catchsegv`, `libSegFault.so`

“Segmentation fault”是很頭痛的一個問題，估計“糾纏”過很多人。這裡僅僅演示通過 `catchsegv` 腳本測試段錯誤的方法，其他方法見後面相關資料。

`catchsegv` 利用系統動態鏈接的 `PRELOAD` 機制（請參考 `man ld-linux`），把庫 `/lib/libSegFault.so` 提前 `load` 到內存中，然後通過它檢查程序運行過程中的段錯誤。

```

$ cat test.c
#include <stdio.h>

int main(void)
{
    char str[10];

    sprintf(str, "%s", 111);

    printf("str = %s\n", str);
    return 0;
}
$ make test
$ LD_PRELOAD=/lib/libSegFault.so ./test #等同於catchsegv ./test
*** Segmentation fault
Register dump:

EAX: 0000006f   EBX: b7eecff4   ECX: 00000003   EDX: 0000006f
ESI: 0000006f   EDI: 0804851c   EBP: bff9a8a4   ESP: bff9a27c

EIP: b7e1755b   EFLAGS: 00010206

CS: 0073   DS: 007b   ES: 007b   FS: 0000   GS: 0033   SS: 007b

Trap: 0000000e   Error: 00000004   OldMask: 00000000
ESP/signal: bff9a27c   CR2: 0000006f

Backtrace:
/lib/libSegFault.so[0xb7f0604f]
[0xfffffe420]
/lib/tls/i686/cmov/libc.so.6(vsprintf+0x8c)[0xb7e0233c]
/lib/tls/i686/cmov/libc.so.6(sprintf+0x2e)[0xb7ded9be]
./test[0x804842b]
/lib/tls/i686/cmov/libc.so.6(__libc_start_main+0xe0)[0xb7dbd050]
./test[0x8048391]
...

```

從結果中可以看出，代碼的 `sprintf` 有問題。經過檢查發現它把整數當字符串輸出，對於字符串的輸出，需要字符串的地址作為參數，而這裡的 `111` 則剛好被解釋成了字符串的地址，因此 `sprintf` 試圖訪問 `111` 這個地址，從而發生了非法訪問內存的情況，出現“Segmentation Fault”。

緩衝區溢出 libsafe.so

緩衝區溢出是堆棧溢出（Stack Smashing），通常發生在對函數內的局部變量進行賦值操作時，超出了該變量的字節長度而引起對棧內原有數據（比如 `eip`, `ebp` 等）的覆蓋，從而引發內存訪問越界，甚至執行非法代碼，導致系統崩潰。關於緩衝區的詳細原理和實例分析見《[緩衝區溢出與注入分析](#)》。這裡僅僅演示該資料中提到的一種用於檢查緩衝區溢出的方法，它同樣採用動態鏈接的 `PRELOAD` 機制提前裝載一個名叫 `libsafe.so` 的庫，可以從[這裡](#)獲取它，下載後，再解壓，編譯，得到 `libsafe.so`，

下面，演示一個非常簡單的，但可能存在緩衝區溢出的代碼，並演示 `libsafe.so` 的用法。

```

$ cat test.c
$ make test

```

```
$ LD_PRELOAD=/path/to/libsafe.so ./test ABCDEFGHIJKLMNOP
ABCDEFGHIJKLMNOP
*** stack smashing detected ***: ./test terminated
Aborted (core dumped)
```

資料[7]分析到，如果不能夠對緩衝區溢出進行有效的處理，可能會存在很多潛在的危險。雖然 `libsafe.so` 採用函數替換的方法能夠進行對這類 Stack Smashing 進行一定的保護，但是無法根本解決問題，`alert7` 大蝦在資料[10]中提出了突破它的辦法，資料[11][11]提出了另外一種保護機制。

内存洩露 Memwatch, Valgrind, mtrace

堆棧通常會被弄在一起叫，不過這兩個名詞卻是指進程的內存映像中的兩個不同的部分，棧（Stack）用於函數的參數傳遞、局部變量的存儲等，是系統自動分配和回收的；而堆（heap）則是用戶通過 malloc 等方式申請而且需要用戶自己通過 free 釋放的，如果申請的內存沒有釋放，那麼將導致內存洩露，進而可能導致堆的空間被用盡；而如果已經釋放的內存再次被釋放（double-free）則也會出現非法操作。如果要真正理解堆和棧的區別，需要理解進程的內存映像，請參考[《緩衝區溢出與注入分析》](#)

這裡演示通過 Memwatch 來檢測程序中可能存在內存洩露，可以從[這裡](#)下載到這個工具。使用這個工具的方式很簡單，只要把它鏈接（ld）到可執行文件中，並在編譯時加上兩個宏開關 -DMEMWATCH -DMW_STDIO。這裡演示一個簡單的例子。

```
$ cat test.c
#include <stdlib.h>
#include <stdio.h>
#include "memwatch.h"

int main(void)
{
    char *ptr1;
    char *ptr2;

    ptr1 = malloc(512);
    ptr2 = malloc(512);

    ptr2 = ptr1;
    free(ptr2);
    free(ptr1);
}

$ gcc -DMEMWATCH -DMW_STDIO test.c memwatch.c -o test
$ cat memwatch.log
===== MEMWATCH 2.71 Copyright (C) 1992-1999 Johan Lindh =====

Started at Sat Mar  1 07:34:33 2008

Modes: __STDC__ 32-bit mwdWORD==(unsigned long)
mwROUNDALLOC==4 sizeof(mwData)==32 mwDataSize==32

double-free: <4> test.c(15), 0x80517e4 was freed from test.c(14)

Stopped at Sat Mar  1 07:34:33 2008

unfreed: <2> test.c(11), 512 bytes at 0x8051a14          {FE FE FE FE FE FE FE FE FE FE FE FI
```

```
Memory usage statistics (global):
N)umber of allocations made: 2
L)argest memory usage      : 1024
T)otal of all alloc() calls: 1024
U)nfreed bytes totals      : 512
```

通過測試，可以看到有一個 512 字節的空間沒有被釋放，而另外 512 字節空間卻被連續釋放兩次（double-free）。Valgrind 和 mtrace 也可以做類似的工作，請參考資料[4]，[5]和 mtrace 的手冊。

代碼調試

調試的方法很多，調試往往要跟蹤代碼的運行狀態，printf 是最基本的辦法，然後呢？靜態調試方法有哪些，非交互的呢？非實時的有哪些？實時的呢？用於調試內核的方法有哪些？有哪些可以用來調試彙編代碼呢？

靜態調試：printf + gcc -D（打印程序中的變量）

利用 gcc 的宏定義開關（-D）和 printf 函數可以跟蹤程序中某個位置的狀態，這個狀態包括當前一些變量和寄存器的值。調試時需要用 -D 開關進行編譯，在正式發佈程序時則可把 -D 開關去掉。這樣做比單純用 printf 方便很多，它可以避免清理調試代碼以及由此帶來的代碼誤刪除等問題。

```
$ cat test.c
#include <stdio.h>
#include <unistd.h>

int main(void)
{
    int i = 0;

#ifdef DEBUG
    printf("i = %d\n", i);

    int t;
    __asm__ __volatile__ ("movl %%ebp, %0;":"=r"(t)::"%ebp");
    printf("ebp = 0x%x\n", t);
#endif

    _exit(0);
}
$ gcc -DDEBUG -g -o test test.c
$ ./test
i = 0
ebp = 0xbfb56d98
```

上面演示瞭如何跟蹤普通變量和寄存器變量的辦法。跟蹤寄存器變量採用了內聯彙編。

不過，這種方式不夠靈活，我們無法“即時”獲取程序的執行狀態，而 gdb 等交互式調試工具不僅解決了這樣的問題，而且通過把調試器拆分成調試服務器和調試客戶端適應了嵌入式系統的調試，另外，通過預先設置斷點以及斷點處需要收集的程序狀態信息解決了交互式調試不適應實時調試的問題。

交互式的調試（動態調試）：**gdb**（支持本地和遠程）/**lald**（彙編指令級別的調試）

嵌入式系統調試方法 **gdbserver/gdb**

估計大家已經非常熟悉 GDB（Gnu DeBugger）了，所以這裡並不介紹常規的 **gdb** 用法，而是介紹它的服務器／客戶（**gdbserver/gdb**）調試方式。這種方式非常適合嵌入式系統的調試，為什麼呢？先來看看這個：

```
$ wc -c /usr/bin/gdbserver
56000 /usr/bin/gdbserver
$ which gdb
/usr/bin/gdb
$ wc -c /usr/bin/gdb
2557324 /usr/bin/gdb
$ echo "(2557324-56000)/2557324" | bc -l
.97810210986171482377
```

gdb 比 **gdbserver** 大了將近 97%，如果把整個 **gdb** 搬到存儲空間受限的嵌入式系統中是很不合適的，不過僅僅 5K 左右的 **gdbserver** 即使在只有 8M Flash 卡的嵌入式系統中也都足夠了。所以在嵌入式開發中，我們通常先在本地主機上交叉編譯好 **gdbserver/gdb**。

如果是初次使用這種方法，可能會遇到麻煩，而麻煩通常發生在交叉編譯 **gdb** 和 **gdbserver** 時。在編譯 **gdbserver/gdb** 前，需要配置(./configure)兩個重要的選項：

- **--host**，指定 **gdb/gdbserver** 本身的運行平臺，
- **--target**，指定 **gdb/gdbserver** 調試的代碼所運行的平臺，

關於運行平臺，通過 **\$MACHTYPE** 環境變量就可獲得，對於 **gdbserver**，因為要把它複製到嵌入式目標系統上，並且用它來調試目標平臺上的代碼，因此需要把 **--host** 和 **--target** 都設置成目標平臺；而 **gdb** 因為還是運行在本地主機上，但是需要用它調試目標系統上的代碼，所以需要把 **--target** 設置成目標平臺。

編譯完以後就是調試，調試時需要把程序交叉編譯好，並把二進制文件複製一份到目標系統上，並在本地需要保留一份源代碼文件。調試過程大體如下，首先在目標系統上啟動調試服務器：

```
$ gdbserver :port /path/to/binary_file
...
```

然後在本地主機上啟動**gdb**客戶端鏈接到 **gdb** 調試服務器，（**gdbserver_ipaddress** 是目標系統的IP地址，如果目標系統不支持網絡，那麼可以採用串口的方式，具體看手冊）

```
$ gdb
...
(gdb) target remote gdbserver_ipaddress:2345
...
```


其他調試過程和普通的gdb調試過程類似。

彙編代碼的調試 **ald**

用 `gdb` 調試彙編代碼貌似會比較麻煩，不過有人正是因為這個原因而開發了一個專門的彙編代碼調試器，名字就叫做 `assembly language debugger`，簡稱 `ald`，你可以從[這裡](#)下載到。

下載後，解壓編譯，我們來調試一個程序看看。

這裡是一段非常簡短的彙編代碼：

```
.global _start
_start:
    popl %ecx
    popl %ecx
    popl %ecx
    movb $10,12(%ecx)
    xorl %edx, %edx
    movb $13, %dl
    xorl %eax, %eax
    movb $4, %al
    xorl %ebx, %ebx
    int $0x80
    xorl %eax, %eax
    incl %eax
    int $0x80
```

彙編、鏈接、運行：

```
$ as -o test.o test.s
$ ld -o test test.o
$ ./test "Hello World"
Hello World
```

查看程序的入口地址：

```
$ readelf -h test | grep Entry
Entry point address:          0x8048054
```

接著用 `ald` 調試：

```
$ ald test
ald> display
Address 0x8048054 added to step display list
ald> n
eax = 0x00000000 ebx = 0x00000000 ecx = 0x00000001 edx = 0x00000000
esp = 0xBFBFDEB4 ebp = 0x00000000 esi = 0x00000000 edi = 0x00000000
ds  = 0x007B es  = 0x007B fs  = 0x0000 gs  = 0x0000
ss  = 0x007B cs  = 0x0073 eip = 0x08048055 eflags = 0x00200292
```

```
Flags: AF SF IF ID
```

```
Dumping 64 bytes of memory starting at 0x08048054 in hex
08048054:  59 59 59 C6 41 0C 0A 31 D2 B2 0D 31 C0 B0 04 31  YYY.A..1...1...1
08048064:  DB CD 80 31 C0 40 CD 80 00 2E 73 79 6D 74 61 62  ...1.@....symtab
08048074:  00 2E 73 74 72 74 61 62 00 2E 73 68 73 74 72 74  ..strtab..shstrt
08048084:  61 62 00 2E 74 65 78 74 00 00 00 00 00 00 00 00  ab..text.....

08048055                                59                                pop ecx
```

可見 `ald` 在啟動時就已經運行了被它調試的 `test` 程序，並且進入了程序的入口 `0x8048054`，緊接著單步執行時，就執行了程序的第一條指令 `popl ecx`。

`ald` 的命令很少，而且跟 `gdb` 很類似，比如這個幾個命令用法和名字都類似 `help,next,continue,set args,break,file,quit,disassemble,enable,disable` 等。名字不太一樣但功能對等的有：`examine` 對 `x`，`enter` 對 `set variable {int} 地址=數據`。

需要提到的是：Linux 下的調試器包括上面的 `gdb` 和 `ald`，以及 `strace` 等都用到 Linux 系統提供的 `ptrace()` 系統調用，這個調用為用戶訪問內存映像提供了便利，如果想自己寫一個調試器或者想hack一下 `gdb` 和 `ald`，那麼好好閱讀資料¹²和 `man ptrace` 吧。

如果確實需要用 `gdb` 調試彙編，可以參考：

- [Linux Assembly "Hello World" Tutorial, CS 200](#)
- [Debugging your Alpha Assembly Programs using GDB](#)

實時調試：gdb tracepoint

對於程序狀態受時間影響的程序，用上述普通的設置斷點的交互式調試方法並不合適，因為這種方式將由於交互時產生的通信延遲和用戶輸入命令的時延而完全改變程序的行為。所以 `gdb` 提出了一種方法以便預先設置斷點以及在斷點處需要獲取的程序狀態，從而讓調試器自動執行斷點處的動作，獲取程序的狀態，從而避免在斷點處出現人機交互產生時延改變程序的行為。

這種方法叫 `tracepoints`（對應 `breakpoint`），它在 `gdb` 的用戶手冊裡頭有詳細的說明，見 [Tracepoints](#)。

在內核中，有實現了相應的支持，叫 `KGTP`。

調試內核

雖然這裡並不會演示如何去 hack 內核，但是相關的工具還是需要簡單提到的，[這個資料](#)列出了絕大部分用於內核調試的工具，這些對你 hack 內核應該會有幫助的。

代碼優化

這部分暫時沒有準備足夠的素材，有待進一步完善。

暫且先提到兩個比較重要的工具，一個是 `Oprofile`，另外一個是 `Perf`。

實際上呢？“代碼測試”部分介紹的很多工具是為代碼優化服務的，更多具體的細節請參考後續資料，自己做實驗吧。

參考資料

- [VERIFICATION AND VALIDATION](#)
- [difference between verification and Validation](#)
- [Coverage Measurement and Profiling\(覆蓋度測量和性能測試,Gcov and Gprof\)](#)
- [Valgrind Usage](#)
 - [Valgrind HOWTO](#)
 - [Using Valgrind to Find Memory Leaks and Invalid Memory Use](#)
- [MEMWATCH](#)
- [Mastering Linux debugging techniques](#)
- [Software Performance Analysis](#)
- [Runtime debugging in embedded systems](#)
- [繞過libsafe的保護--覆蓋_dl_lookup_versioned_symbol技術](#)
- [介紹Propolice怎樣保護stack-smashing的攻擊](#)
- [Tools Provided by System: ltrace,mtrace,strace](#)
- [Process Tracing Using Ptrace](#)
- [Kernel Debugging Related Tools: KGDB, KGOV, KFI/KFT/Ftrace, GDB Tracepoint, UML, kdb](#)
- [用Graphviz 可視化函數調用](#)
- [Linux 段錯誤詳解](#)
- [源碼分析之函數調用關係繪製系列](#)
 - [源碼分析：靜態分析 C 程序函數調用關係圖](#)
 - [源碼分析：動態分析 C 程序函數調用關係](#)
 - [源碼分析：動態分析 Linux 內核函數調用關係](#)
 - [源碼分析：函數調用關係繪製方法與逆向建模](#)
- [Linux 下緩衝區溢出攻擊的原理及對策](#)
- [Linux 彙編語言開發指南](#)
- [緩衝區溢出與注入分析\(第一部分：進程的內存映像\)](#)
- [Optimizing C Code](#)
- [Performance programming for scientific computing](#)
- [Performance Programming](#)
- [Linux Profiling and Optimization](#)
- [High-level code optimization](#)
- [Code Optimization](#)