GCC 使用教程

目 录

gcc

makefile 写法

蓝森林 http://www.lslnet.com 2001 年 3 月 22 日 08:44

作 者： 许明彦

Abstract:

在 Unix 上写程式的人大概都碰过 Makefile，尤其是用 C 来开发程式的人。用 make 来开发和编译程式的确很方便，可是要写出一个 Makefile 就不简单了。偏偏介绍 Makefile 的文件不多，GNU Make 那份印出来要几百页的文件，光看完 Overview 就快阵亡了，难怪许多
人闻 Unix 色变。

本文将介绍如何利用 GNU Autoconf 及 Automake 这两套软体来协助我们『自动』产生 Makefile 档，并且让开发出来的软体可以像 Apache, MySQL 和常见的 GNU 软体一样，只要会 ``./configure'', ``make'', ``make install'' 就可以把程式安装到系统中。如果您有
心开发 Open Source 的软体，或只是想在 Unix 系统下写写程式。希望这份介绍文件能帮助您轻松地进入 Unix Programming 的殿堂。

1. 简介

Makefile 基本上就是『目标』(target), 『关连』(dependencies) 和『动作』三者所组成的一连串规则。而 make 就会根据 Makefile 的规则来决定如何编译 (compile) 和连结 (link) 程式。实际上，make 可做的不只是编译和连结程式，例如 FreeBSD 的 port collect
ion 中， Makefile 还可以做到自动下载原始程式套件，解压缩 (extract) ，修补 (patch)，设定，然後编译，安装至系统中。

Makefile 基本构造虽然简单，但是妥善运用这些规则就也可以变出许多不同的花招。却也因此，许多刚开始学习写 Makefile 时会感到没有规范可循，每个人写出来的 Makefile 长得都不太一样，不知道从何下手，而且常常会受限於自己的开发环境，只要环境变数不同或路
径改一下，可能 Makefile 就得跟着修改。虽然有 GNU Makefile Conventions (GNU Makefile 惯例) 订出一些使用 GNU 程式设计时撰写 Makefile 的一些标准和规范，但是内容很长而且很复杂，并且经常做些调整，为了减轻程式设计师维护 Makefile 的负担，因此有了 Automake。

程式设计师只需写一些预先定义好的巨集 (macro)，交给 Automake 处理後会产生一个可供

Autoconf 使用的 Makefile.in 档。再配合利用 Autoconf 产生的自动设定档 configure 即可产生一份符合 GNU Makefile 惯例的 Makeifle 了。

2. 上路之前

在开始试着用 Automake 之前，请先确认你的系统已经安装以下的软体：

1. GNU Automake
2. GNU Autoconf
3. GNU m4
4. perl
5. GNU Libtool (如果你需要产生 shared library)

我会建议你最好也使用 GNU C/C++ 编译器 、GNU Make 以及其它 GNU 的工具程式来做为开发的环境，这些工具都是属於 Open Source Software 不仅免费而且功能强大。如果你是使用 Red Hat Linux 可以找到所有上述软体的 rpm 档，FreeBSD 也有现成的 package 可以直接安装，或着你也可以自行下载这些软体的原始档回来 DIY。以下的范例是在 Red Hat Linux 5.2 + CLE2 的环境下所完成的。

3. 一个简单的例子

Automake 所产生的 Makefile 除了可以做到程式的编译和连结，也已经把如何产生程式文件(如 manual page, info 档及 dvi 档) 的动作，还有把原始程式包装起来以供散 的动作都考虑进去了，所以原始程式所存放的目录架构最好符合 GNU 的标准惯例，接下来我拿 hello.c 来做为例子。

在工作目录下建立一个新的子目录 ``devel''，再在 devel 下建立一个``hello'' 的子目录，这个目录将作为我们存放 hello 这个程式及其相关档案的地方：

```
% mkdir devel
% cd devel
% mkdir hello
% cd hello
```

用编辑器写个 hello.c 档，

```
#include stdio.h
int main(int argc, char** argv)
{
printf(``Hello, GNU! '');
return 0;
}
```

接下来就要用 Autoconf 及 Automake 来帮我们产生 Makefile 档了，

1. 用 autoscan 产生一个 configure.in 的雏型，执行 autoscan 後会产生一个 configure.scan 的档案，我们可以用它做为 configure.in 档的蓝本。

% autoscan
% ls
configure.scan hello.c

2. 编辑 configure.scan 档，如下所示，并且把它的档名改成 configure.in
dnl Process this file with autoconf to produce a con figure script.
AC_INIT(hello.c)
AM_INIT_AUTOMAKE(hello, 1.0)
dnl Checks for programs.
AC_PROG_CC
dnl Checks for libraries.
dnl Checks for header files.
dnl Checks for typedefs, structures, and compiler ch aracteristics.
dnl Checks for library functions.
AC_OUTPUT(Makefile)

3. 执行 aclocal 和 autoconf ，分别会产生 aclocal.m4 及 configure 两个档案
% aclocal
% autoconf
% ls
aclocal.m4 configure configure.in hello.c

4. 编辑 Makefile.am 档，内容如下
AUTOMAKE_OPTIONS= foreign
bin_PROGRAMS= hello
hello_SOURCES= hello.c

5. 执行 automake --add-missing ，Automake 会根据 Makefile.am 档产生一些档案，包含最重要的 Makefile.in
% automake --add-missing
automake: configure.in: installing `./install-sh'
automake: configure.in: installing `./mkinstalldirs'
automake: configure.in: installing `./missing'

6. 最後执行 ./configure ，
% ./configure
creating cache ./config.cache
checking for a BSD compatible install... /usr/bin/in stall -c
checking whether build environment is sane... yes
checking whether make sets ${MAKE}... yes
checking for working aclocal... found
checking for working autoconf... found
checking for working automake... found
checking for working autoheader... found
checking for working makeinfo... found

checking for gcc... gcc
checking whether the C compiler (gcc) works... yes
checking whether the C compiler (gcc) is a cross-co mpiler... no
checking whether we are using GNU C... yes
checking whether gcc accepts -g... yes
updating cache ./config.cache
creating ./config.status
creating Makefile

现在你的目录下已经产生了一个 Makefile 档，下个 ``make'' 指令就可以开始编译 hello.c 成执行档，执行 ./hello 和 GNU 打声招呼吧！

% make
gcc -DPACKAGE="hello" -DVERSION="1.0" -I. -I. -g -O2 -c he llo.c
gcc -g -O2 -o hello hello.o
% ./hello
Hello! GNU!

你还可以试试 ``make clean''，''make install''，''make dist'' 看看会有什麼结果。你也可以把产生出来的 Makefile 秀给你的老板，让他从此对你刮目相看 :-)

4. 一探究竟

上述产生 Makefile 的过程和以往自行编写的方式非常不一样，舍弃传统自行定义 make 的规则，使用 Automake 只需用到一些已经定义好的巨集即可。我们把巨集及目标 (target)写在 Makefile.am 档内，Automake 读入 Makefile.am 档後会把这一串已经定义好的巨集展
开并且产生对应的 Makefile.in 档， 然後再由 configure 这个 shell script 根据 Makefile.in 产生适合的 Makefile。

在此范例中可藉由 Autoconf 及 Automake 工具所产生的档案有 configure.scan、aclocal.m4、configure、Makefile.in，需要我们加入设定者为 configure.in 及 Makefile.am。

4.1 编辑 configure.in 档

Autoconf 是用来产生 'configure' 档的工具。'configure' 是一个 shell script，它可以自动设定原始程式以符合各种不同平台上 Unix 系统的特性，并且根据系统叁数及环境产生合适的 Makefile 档或是 C 的标头档 (header file)，让原始程式可以很方便地在这些不同
的平台上被编译出来。Autoconf 会读取 configure.in 档然後产生 'configure' 这个 shell script。

configure.in 档的内容是一连串 GNU m4 的巨集，这些巨集经过 autoconf 处理後会变成检查系统特徵的 shell script。configure.in 内巨集的顺序并没有特别的规定，但是每一个 configure.in 档必须在所有巨集前加入 AC_INIT 巨集，然後在所有巨集的最後面加上 AC
_OUTPUT 巨集。我们可先用 autoscan 扫描原始档以产生一个 configure.scan 档，再对 configure.scan 做些修改成 configure.in 档。在范例中所用到的巨集如下：

dnl

这个巨集後面的字不会被处理，可视为注解。

AC_INIT(FILE)

这个巨集用来检查原始码所在的路径，autoscan 会自动产生，我们不必修改它。

AM_INIT_AUTOMAKE(PACKAGE,VERSION)

这是使用 Automake 所必备的巨集，PACKAGE 是我们所要产生软体套件的名称，VERSION 是版本编号。

AC_PROG_CC

检查系统可用的 C 编译器，如果原始程式是用 C 写的就需要这个巨集。

AC_OUTPUT(FILE)

设定 configure 所要产生的档案，如果是 Makefile 的话，configure 便会把它检查出来的结果带入 Makefile.in 档然後产生
合适的 Makefile。

实际上，我们使用 Automake 时，还须要一些其它的巨集，这些额外的巨集我们用 aclocal 来帮我们产生。执行 aclocal 会产生 aclocal.m4 档，如果没有特别的用途，我们可以不必修改它，用 aclocal 所产生的巨集会告诉 Automake 怎麼做。

有了 configure.in 及 aclocal.m4 两个档案後，便可以执行 autoconf 来产生 configure 档了。

4.2 编辑 Makefile.am 档

接下来我们要编辑 Makefile.am 档，Automake 会根据 configure.in 中的巨集把 Makefile.am 转成 Makefile.in 档。Makefile.am 档定义我们所要产的目标：

AUTOMAKE_OPTIONS

设定 automake 的选项。Automake 主要是帮助开发 GNU 软体的人员维护软体套件，所以在执行 automake 时，会检查目录下是否存在标准 GNU 软体套件中应具备的文件档案，例如 'NEWS'、'AUTHOR'、'ChangeLog' 等文件档。设成 foreign 时，automake 会改用一般软
体套件的标准来检查。

bin_PROGRAMS

定义我们所要产生的执行档档名。如果要产生多个执行档，每个档名用空白字元隔开。

hello_SOURCES

定义 'hello' 这个执行档所需要的原始档。如果 'hello' 这个程式是由多个原始档所产生，必须把它所用到的原始档都列出来，以空白字元隔开。假设 'hello' 这个程式需要 'hello.c'、'main.c'、

'hello.h'

三个档案的话，则定义

hello_SOURCES= hello.c main.c hello.h

如果我们定义多个执行档，则对每个执行档都要定义相对的 filename_SOURCES。

编辑好 Makefile.am 档，就可以用 automake --add-missing 来产生 Makefile.in。加上 --add-missing 选项是告诉 automake 顺便帮我们加入包装一个软体套件所必备的档案。Automake 产生出来的 Makefile.in 档是完全符合 GNU Makefile 的惯例，我们只要执行 confi
gure 这个 shell script 便可以产生合适的 Makefile 档了。

4.3 使用 Makefile

利用 configure 所产生的 Makefile 档有几个预设的目标可供使用，我们只拿其中几个简述如下：

make all

产生我们设定的目标，即此范例中的执行档。只打 make 也可以，此时会开始编译原始码，然後连结，并且产生执行档。

make clean

清除之前所编译的执行档及目的档 (object file, *.o)。

make distclean

除了清除执行档和目的档外，也把 configure 所产生的 Makefile 也清除掉。

make install

将程式安装至系统中。如果原始码编译无误，且执行结果正确，便可以把程式安装至系统预设的执行档存放路径。如果我们用 bin_PROGRAMS 巨集的话，程式会被安装至 /usr/local/bin 这个目录。

make dist

将程式和相关的档案包装成一个压缩档以供散播 (distribution) 。执行完在目录下会产生一个以 PACKAGE-VERSION.tar.gz 为名称的档案。PACKAGE 和 VERSION 这两个变数是根据 configure.in 档中 AM_INIT_AUTOMAKE(PACKAGE, VERSION) 的定义。在此范例中会产生 'hello-1.0.tar.gz' 的档案。

make distcheck

和 make dist 类似，但是加入检查包装後的压缩档是否正常。这个目标除了把程式和相关档案包装成 tar.gz 档外，还会自动把这个压
缩档解开，执行 configure，并且进行 make all 的动作，确认编译无误後，会显示这个 tar.gz 档已经准备好可供散播了。这个检查非
常有用，检查过关的套件，基本上可以给任何一个具备 GNU 发展境的人去重新编译。就 hello-1.tar.gz 这个范例而言，除了在 Red
Hat Linux 上，在 FreeBSD 2.2.x 版也可以正确地重新编译。

要注意的是，利用 Autoconf 及 Automake 所产生出来的软体套件是可以在没有安装 Autoconf 及 Automake 的环境上使用的，因为 configure 是一个 shell script，它已被设计可以在一般 Unix 的 sh 这个 shell 下执行。但是如果要修改 configure.in 及 Makefile.a
m 档再产生新的 configure 及 Makefile.in 档时就一定要有 Autoconf 及 Automake 了。

5. 相关讯息

Autoconf 和 Automake 功能十分强大，你可以从它们所附的 info 档找到详细的用法。你也可以从许多现存的 GNU 软体或 Open Source 软体中找到相关的 configure.in 或 Makefile.am 档，它们是学习 Autoconf 及 Automake 更多技巧的最佳范例。

这篇简介只用到了 Autoconf 及 Automake 的皮毛罢了，如果你有心加入 Open Source 软体开发的行列，希望这篇文件能帮助你对产生 Makefile 有个简单的依据。其它有关开发 GNU 程式或 C 程式设计及 Makefile 的详细运用及技巧，我建议你从 GNU Coding Standards3
(GNU 编码标准规定) 读起，里面包含了 GNU Makefile 惯例，还有发展 GNU 软体套件的标准程序和惯例。这些 GNU 软体的线上说明文件可以在 http://www.gnu.org/ 这个网站上找到。

6. 结语

经由 Autoconf 及 Automake 的辅助，产生一个 Makefile 似乎不再像以前那麼困难了，而使用 Autoconf 也使得我们在不同平台上或各家 Unix 之间散播及编译程式变得简单，这对於在 Unix 系统上开发程式的人员来说减轻了许多负担。妥善运用这些 GNU 的工具软体，可
以帮助我们更容易去发展程式，而且更容易维护原始程式码。

一九九八年是 Open Source 运动风起云涌的一年，许多 Open Source 的软体普遍受到网路上大众的欢迎和使用。感谢所有为 Open Source 奉献的人们，也希望藉由本文能吸引更多的人加入『自由』、『开放』的 OpenSource 行列。

About this document ...

轻轻松松产生 Makefile1

This document was generated using the LaTeX2HTML translator Version 98.2 beta6 (August 14th, 1998) Copyright (C) 1993, 1994, 1995, 1996, Nikos Drakos, ComputerBased Learning Unit, University of Leeds.Copyright (C) 1997, 1998, Ross Moore, Mathematics Department,Macquarie University, Sydney.

The command line arguments were:
latex2html -split 0 -show_section_numbers automake.tex
The translation was initiated by on 1999-02-11

Footnotes
... itle1
本文件使用 ChiLaTeX 制作。
... CLE2
CLE (Chinese Linux Extension，Linux 中文延伸套件),
http://cle.linux.org.tw/

... Standards3
GNU Coding Standards, Richard Stallman.

gcc_egcs 使用

1.使用 egcs

Linux 中最重要的软件开发工具是 GCC。GCC 是 GNU 的 C 和 C++ 编译器。实际上，GCC 能够编译三种语言：C、C++ 和 Object C（C 语言的一种面向对象扩展）。利用 gcc 命令可同时编译并连接 C 和 C++ 源程序。

#DEMO#: hello.c

如果你有两个或少数几个 C 源文件，也可以方便地利用 GCC 编译、连接并生成可执行文件。例如，假设你有两个源文件 main.c 和 factorial.c 两个源文件，现在要编译生成一个计算阶乘的程序。

```
----------------------
清单 factorial.c
----------------------
#include <stdio.h>
#include <stdlib.h>

int factorial (int n)
{
   if (n <= 1)
      return 1;

   else
      return factorial (n - 1) * n;
}
----------------------

----------------------
清单  main.c
----------------------
#include <stdio.h>
#include <stdlib.h>

int factorial (int n);

int main (int argc, char **argv)
{
   int n;

   if (argc < 2) {
      printf ("Usage: %s n\n", argv [0]);
      return -1;
```

```
    }
    else {
        n = atoi (argv[1]);
        printf ("Factorial of %d is %d.\n", n, factorial (n));
    }

    return 0;
}
```

利用如下的命令可编译生成可执行文件，并执行程序：

```
$ gcc -o factorial main.c factorial.c
$ ./factorial 5
Factorial of 5 is 120.
```

GCC 可同时用来编译 C 程序和 C++ 程序。一般来说，C 编译器通过源文件的后缀名来判断是 C 程序还是 C++ 程序。在 Linux 中，C 源文件的后缀名为 .c，而 C++ 源文件的后缀名为 .C 或 .cpp。

但是，gcc 命令只能编译 C++ 源文件，而不能自动和 C++ 程序使用的库连接。因此，通常使用 g++ 命令来完成 C++ 程序的编译和连接，该程序会自动调用 gcc 实现编译。假设我们有一个如下的 C++ 源文件（hello.C）：

```
#include <iostream.h>

void main (void)
{
    cout << "Hello, world!" << endl;
}
```

则可以如下调用 g++ 命令编译、连接并生成可执行文件：

```
$ g++ -o hello hello.C
$ ./hello
Hello, world!
```

2. gcc/egcs 的主要选项

表 1-3 gcc 命令的常用选项

选项	解释
-ansi	只支持 ANSI 标准的 C 语法。这一选项将禁止 GNU C 的某些特色，例如 asm 或 typeof 关键词。
-c	只编译并生成目标文件。
-DMACRO	以字符串“1”定义 MACRO 宏。
-DMACRO=DEFN	以字符串“DEFN”定义 MACRO 宏。

-E 只运行 C 预编译器。
-g 生成调试信息。GNU 调试器可利用该信息。
-IDIRECTORY 指定额外的头文件搜索路径 DIRECTORY。
-LDIRECTORY 指定额外的函数库搜索路径 DIRECTORY。
-lLIBRARY 连接时搜索指定的函数库 LIBRARY。
-m486 针对 486 进行代码优化。
-o FILE 生成指定的输出文件。用在生成可执行文件时。
-O0 不进行优化处理。
-O 或 -O1 优化生成代码。
-O2 进一步优化。
-O3 比 -O2 更进一步优化，包括 inline 函数。
-shared 生成共享目标文件。通常用在建立共享库时。
-static 禁止使用共享连接。
-UMACRO 取消对 MACRO 宏的定义。
-w 不生成任何警告信息。
-Wall 生成所有警告信息。

#DEMO#

---

gdb 使用

1.简介

GNU 的调试器称为 gdb，该程序是一个交互式工具，工作在字符模式。在 X Window 系统中，有一个 gdb 的前端图形工具，称为 xxgdb。gdb 是功能强大的调试程序，可完成如下的调试任务：

* 设置断点；
* 监视程序变量的值；
* 程序的单步执行；
* 修改变量的值。

在可以使用 gdb 调试程序之前，必须使用 -g 选项编译源文件。可在 makefile 中如下定义 CFLAGS 变量：

CFLAGS = -g
运行 gdb 调试程序时通常使用如下的命令：

gdb progname

在 gdb 提示符处键入 help，将列出命令的分类，主要的分类有：

* aliases：命令别名
* breakpoints：断点定义；
* data：数据查看；

* files：指定并查看文件；
* internals：维护命令；
* running：程序执行；
* stack：调用栈查看；
* statu：状态查看；
* tracepoints：跟踪程序执行。

键入 help 后跟命令的分类名，可获得该类命令的详细清单。

2.gdb 的常用命令

表 1-4 常用的 gdb 命令

命令	解释
break NUM	在指定的行上设置断点。
bt	显示所有的调用栈帧。该命令可用来显示函数的调用顺序。
clear	删除设置在特定源文件、特定行上的断点。其用法为：clear FILENAME:NUM。
continue	继续执行正在调试的程序。该命令用在程序由于处理信号或断点而导致停止运行时。
display EXPR	每次程序停止后显示表达式的值。表达式由程序定义的变量组成。
file FILE	装载指定的可执行文件进行调试。
help NAME	显示指定命令的帮助信息。
info break	显示当前断点清单，包括到达断点处的次数等。
info files	显示被调试文件的详细信息。
info func	显示所有的函数名称。
info local	显示当函数中的局部变量信息。
info prog	显示被调试程序的执行状态。
info var	显示所有的全局和静态变量名称。
kill	终止正被调试的程序。
list	显示源代码段。
make	在不退出 gdb 的情况下运行 make 工具。
next	在不单步执行进入其他函数的情况下，向前执行一行源代码。
print EXPR	显示表达式 EXPR 的值。

3.gdb 使用范例

清单 一个有错误的 C 源程序 bugging.c

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

static char buff [256];
static char* string;
int main ()
{
```

```
    printf ("Please input a string: ");
    gets (string);

    printf ("\nYour string is: %s\n", string);
}
-----------------
```

上面这个程序非常简单，其目的是接受用户的输入，然后将用户的输入打印出来。该程序使用了一个未经过初始化的字符串地址 string，因此，编译并运行之后，将出现 Segment Fault 错误：

```
$ gcc -o test -g test.c
$ ./test
Please input a string: asfd
Segmentation fault (core dumped)
```

为了查找该程序中出现的问题，我们利用 gdb，并按如下的步骤进行：

1．运行 gdb bugging 命令，装入 bugging 可执行文件；
2．执行装入的 bugging 命令；
3．使用 where 命令查看程序出错的地方；
4．利用 list 命令查看调用 gets 函数附近的代码；
5．唯一能够导致 gets 函数出错的因素就是变量 string。用 print 命令查看 string 的值；
6．在 gdb 中，我们可以直接修改变量的值，只要将 string 取一个合法的指针值就可以了，为此，我们在第 11 行处设置断点；
7．程序重新运行到第 11 行处停止，这时，我们可以用 set variable 命令修改 string 的取值；
8．然后继续运行，将看到正确的程序运行结果。

---

gcc 常用选项对代码的影响

by alert7
2001-12-21
测试环境 redhat 6.2

★ 前言

本文讨论 gcc 的一些常用编译选项对代码的影响。当然代码变了，它的内存布局也就会变了，随之 exploit 也就要做相应的变动。
gcc 的编译选项实在太多，本文检了几个最常用的选项。

★ 演示程序

```
[alert7@redhat62 alert7]$ cat > test.c
#include
void hi(void)
{
```

```
printf("hi");
}
int main(int argc, char *argv[])
{
	hi();
	return 0;
}
```

一般情况

★ 一般情况

```
[alert7@redhat62 alert7]$ gcc -o test test.c
[alert7@redhat62 alert7]$ wc -c test
  11773 test
[alert7@redhat62 alert7]$ gdb -q test
(gdb) disass main
Dump of assembler code for function main:
0x80483e4 :	push	%ebp
0x80483e5 :	mov	%esp,%ebp
0x80483e7 :	call	0x80483d0
0x80483ec :	xor	%eax,%eax
0x80483ee :	jmp	0x80483f0
0x80483f0 :	leave
0x80483f1 :	ret
....
End of assembler dump.
(gdb) disass hi
Dump of assembler code for function hi:
0x80483d0 :	push	%ebp
0x80483d1 :	mov	%esp,%ebp
0x80483d3 :	push	$0x8048450
0x80483d8 :	call	0x8048308
0x80483dd :	add	$0x4,%esp
0x80483e0 :	leave
0x80483e1 :	ret
0x80483e2 :	mov	%esi,%esi
End of assembler dump.
```

来看看部分的内存映象

```
		(内存高址)
			+--------+
			|bffffbc4| argv 的地址(即 argv[0]的地址)
		0xbffffb84 +--------+
```

```
                |00000001| argc 的值
        0xbffffb80 +---------+
                |400309cb|main 的返回地址
        0xbffffb7c +---------+ <-- 调用 main 函数前的 esp
                |bffffb98| 调用 main 函数前的 ebp
        0xbffffb78 +---------+ <-- main 函数的 ebp
                |080483ec| hi()的返回地址
        0xbffffb74 +---------+
                |bffffb78| 调用 hi()前的 esp
        0xbffffb70 +---------+
                |08048450| "hi"的地址
        0xbffffb6c +---------+
                | ..... |
        (内存低址)
leave   指令所做的操作相当于 MOV ESP,EBP 然后 POP EBP
ret   指令所做的操作相当于 POP EIP
```

-O 编译选项

★ -O 编译选项

With `-O', the compiler tries to reduce code size and execution time.

When you specify `-O', the two options `-fthread-jumps' and `-fdefer-pop' are turned on

优化，减少代码大小和执行的时间

```
[alert7@redhat62 alert7]$ gcc -O -o test test.c
[alert7@redhat62 alert7]$ wc -c test
  11757 test
[alert7@redhat62 alert7]$ gdb -q test
(gdb) disass main
Dump of assembler code for function main:
0x80483d8 :      push   %ebp
0x80483d9 :    mov    %esp,%ebp
0x80483db :    call   0x80483c8
0x80483e0 :    xor    %eax,%eax
0x80483e2 :   leave
0x80483e3 :   ret
0x80483e4 :   nop
...
End of assembler dump.
(gdb) disass hi
Dump of assembler code for function hi:
0x80483c8 :      push   %ebp
0x80483c9 :      mov    %esp,%ebp
```

```
0x80483cb :     push   $0x8048440
0x80483d0 :     call   0x8048308
0x80483d5 :     leave
0x80483d6 :     ret
0x80483d7 :     nop
End of assembler dump.
```

在 main()中，把一条 jmp 指令优化掉了，很显然，这条指令是可以不需要的。
在 hi()中，把 add $0x4,%esp 优化掉了，这会不会使 stack 不平衡呢？

来看看部分的内存映象

```
(内存高址)
        +--------+
        |bffffbc4| argv 的地址(即 argv[0]的地址)
0xbffffb84 +--------+
        |00000001| argc 的值
0xbffffb80 +--------+
        |400309cb|main 的返回地址
0xbffffb7c +--------+ <-- 调用 main 函数前的 esp
        |bffffb98| 调用 main 函数前的 ebp
0xbffffb78 +--------+ <-- main 函数的 ebp
        |080483e0| hi()的返回地址
0xbffffb74 +--------+
        |bffffb78| 调用 hi()前的 esp
0xbffffb70 +--------+
        |08048440| "hi"的地址
0xbffffb6c +--------+
        | ..... |
(内存低址)
```

leave 指令所做的操作相当于把 MOV ESP,EBP 然后 POP EBP。看到 leave 指令操作了没有，先把 ebp-->esp，再 pop ebp，这样即使在过程内堆栈的 esp,ebp 是不平衡的，但只要返回时候碰到 leave 指令就会平衡了，所以把 add $0x4,%esp 优化掉也是没有问题的。

-O2 编译选项

★ -O2 编译选项

-O2

Optimize even more. Nearly all supported optimizations that do not involve a space-speed tradeoff are performed. Loop unrolling and function inlining are not done, for example. As compared to -O, this option increases both compilation time and the performance of the generated code.

```
[alert7@redhat62 alert7]$ gcc -O2 -o test test.c
[alert7@redhat62 alert7]$ wc -c test
  11757 test
[alert7@redhat62 alert7]$ gdb -q test
(gdb) disass main
Dump of assembler code for function main:
0x80483d8 :       push   %ebp
0x80483d9 :     mov    %esp,%ebp
0x80483db :     call   0x80483c8
0x80483e0 :     xor    %eax,%eax
0x80483e2 :    leave
0x80483e3 :    ret
...
0x80483ef :    nop
End of assembler dump.
(gdb) disass hi
Dump of assembler code for function hi:
0x80483c8 :       push   %ebp
0x80483c9 :       mov    %esp,%ebp
0x80483cb :       push   $0x8048440
0x80483d0 :       call   0x8048308
0x80483d5 :      leave
0x80483d6 :      ret
0x80483d7 :      nop
End of assembler dump.
```

由于程序比较简单，再优化也没有好优化的了，所以跟-O 出来的一样。

-fomit-frame-pointer 编译选项

★ -fomit-frame-pointer 编译选项

-fomit-frame-pointer

Don't keep the frame pointer in a register for functions that don't need one. This avoids the instructions to save, set up and restore frame pointers; it also makes an extra register available in many functions. It also makes debugging impossible on most machines.

忽略帧指针。这样在程序就不需要保存，安装，和恢复 ebp 了。这样 ebp 也就是一个 free 的 register 了，在函数中就可以随便使用了。

```
[alert7@redhat62 alert7]$ gcc -fomit-frame-pointer -o test test.c
[alert7@redhat62 alert7]$ wc -c test
  11773 test
```

```
[alert7@redhat62 alert7]$ gdb -q test
(gdb) disass main
Dump of assembler code for function main:
0x80483e0 :       call    0x80483d0
0x80483e5 :     xor     %eax,%eax
0x80483e7 :     jmp     0x80483f0
0x80483e9 :     lea     0x0(%esi,1),%esi
0x80483f0 :    ret
....
End of assembler dump.
(gdb) disass hi
Dump of assembler code for function hi:
0x80483d0 :       push    $0x8048450
0x80483d5 :       call    0x8048308
0x80483da :      add     $0x4,%esp
0x80483dd :      ret
0x80483de :      mov     %esi,%esi
End of assembler dump.
```

在 main()和 hi()中都去掉了以下指令

```
push    %ebp
mov     %esp,%ebp//这两条指令安装
leave//这条指令恢复
```

来看看部分的内存映象

```
                (内存高址)
                        +--------+
                        |bffffbc4| argv 的地址(即 argv[0]的地址)
                0xbffffb84 +--------+
                        |00000001| argc 的值
                0xbffffb80 +--------+
                        |400309cb|main 的返回地址
                0xbffffb7c +--------+
                        |080483e5| hi()的返回地址
                0xbffffb78 +--------+
                        |08048450|  "hi"字符串的地址
                0xbffffb74 +--------+
                        | ..... |
                (内存低址)
```

没有保存上层执行环境的 ebp.

-fomit-frame-pointer && -O2

★ -fomit-frame-pointer && -O2

-fomit-frame-pointer 编译选项去掉了
push %ebp
mov %esp,%ebp//这两条指令安装
leave//这条指令恢复
-O2 编译选项去掉了
add $0x4,%esp
两个加起来会不会这四条指令一起去掉，从而使 stack 不平衡呢？

```
[alert7@redhat62 alert7]$ gcc -fomit-frame-pointer -O2 -o test test.c
[alert7@redhat62 alert7]$ wc -c test
  11741 test
[alert7@redhat62 alert7]$ gdb -q test
(gdb) disass main
Dump of assembler code for function main:
0x80483d8 :       call    0x80483c8
0x80483dd :     xor     %eax,%eax
0x80483df :     ret
End of assembler dump.
(gdb) disass hi
Dump of assembler code for function hi:
0x80483c8 :       push    $0x8048430
0x80483cd :       call    0x8048308
0x80483d2 :     add     $0x4,%esp
0x80483d5 :     ret
0x80483d6 :     mov     %esi,%esi
End of assembler dump.
```

来看看部分的内存映象

```
            (内存高址)
                    +--------+
                    |bffffbc4| argv 的地址(即 argv[0]的地址)
            0xbffffb84 +--------+
                    |00000001| argc 的值
            0xbffffb80 +--------+
                    |400309cb|main 的返回地址
            0xbffffb7c +--------+
                    |080483dd| hi()的返回地址
            0xbffffb78 +--------+
                    |08048430|  "hi"字符串的地址
            0xbffffb74 +--------+
                    | ..... |
            (内存低址)
```

此时就没有把 add $0x4,%esp 优化掉，如果优化掉的话，整个 stack 就会变的不平衡，从而会导致程序出错。

-fPIC 编译选项

★ -fPIC 编译选项

-fPIC If supported for the target machine, emit position-independent code, suitable for dynamic linking,even if branches need large displacements.

产生位置无关代码（PIC），一般创建共享库时用到。

在 x86 上，PIC 的代码的符号引用都是通过 ebx 进行操作的。

```
[alert7@redhat62 alert7]$ gcc -fPIC -o test test.c
[alert7@redhat62 alert7]$ wc -c test
  11805 test
[alert7@redhat62 alert7]$ gdb -q test
(gdb) disass main
Dump of assembler code for function main:
0x80483f8 :       push   %ebp
0x80483f9 :     mov    %esp,%ebp
0x80483fb :     push   %ebx
0x80483fc :     call   0x8048401
0x8048401 :     pop    %ebx//取得该指令的地址
0x8048402 :     add    $0x1093,%ebx//此时 ebx 里面存放着是 GOT 表的地址
0x8048408 :     call   0x80483d0
0x804840d :     xor    %eax,%eax
0x804840f :    jmp    0x8048411
0x8048411 :     mov    0xfffffffc(%ebp),%ebx
0x8048414 :     leave
0x8048415 :     ret
...
End of assembler dump.
(gdb) disass hi
Dump of assembler code for function hi:
0x80483d0 :       push   %ebp
0x80483d1 :      mov    %esp,%ebp
0x80483d3 :      push   %ebx
0x80483d4 :      call   0x80483d9
0x80483d9 :      pop    %ebx
0x80483da :     add    $0x10bb,%ebx
0x80483e0 :     lea    0xffffefdc(%ebx),%edx
0x80483e6 :     mov    %edx,%eax
0x80483e8 :     push   %eax
0x80483e9 :     call   0x8048308
0x80483ee :     add    $0x4,%esp
```

```
0x80483f1 :     mov    0xfffffffc(%ebp),%ebx
0x80483f4 :     leave
0x80483f5 :     ret
0x80483f6 :     mov    %esi,%esi
End of assembler dump.
```

来看看部分的内存映象

```
  (内存高址)
        +--------+
        |bffffbc4| argv 的地址(即 argv[0]的地址)
  0xbffffb84 +--------+
        |00000001| argc 的值
  0xbffffb80 +--------+
        |400309cb|main 的返回地址
  0xbffffb7c +--------+ <-- 调用 main 函数前的 esp
        |bffffb98| 调用 main 函数前的 ebp
  0xbffffb78 +--------+ <-- main 函数的 ebp
        |401081ec| 保存的 ebx
  0xbffffb74 +--------+
        |0804840d| （存放过 call 0x8048401 的下一条指令地址）
  0xbffffb70 +--------+
        |bffffb78| 调用 hi()前的 esp
  0xbffffb6c +--------+
        |08049494| GOT 表地址
  0xbffffb68 +--------+
        |08048470|（存放过 call 0x80483d9 的下一条指令地址）
  0xbffffb64 +--------+
        | ..... |
   (内存低址)
```

-static 编译选项

★ -static 编译选项

```
-static
    On systems that support dynamic linking, this prevents
    linking with the shared libraries.  On other  systems,
    this option has no effect.
```

把一些函数都静态的编译到程序中，而无需动态链接了。

```
[alert7@redhat62 alert7]$ gcc -o test -static test.c
[alert7@redhat62 alert7]$ wc -c test
962808 test
[alert7@redhat62 alert7]$ gdb -q test
```

```
(gdb) disass main
Dump of assembler code for function main:
0x80481b4 :       push   %ebp
0x80481b5 :     mov    %esp,%ebp
0x80481b7 :     call   0x80481a0
0x80481bc :     xor    %eax,%eax
0x80481be :    jmp    0x80481c0
0x80481c0 :    leave
0x80481c1 :    ret
...
End of assembler dump.
(gdb) disass hi
Dump of assembler code for function hi:
0x80481a0 :       push   %ebp
0x80481a1 :       mov    %esp,%ebp
0x80481a3 :       push   $0x8071528
0x80481a8 :       call   0x804865c
0x80481ad :      add    $0x4,%esp
0x80481b0 :      leave
0x80481b1 :      ret
0x80481b2 :      mov    %esi,%esi
End of assembler dump.
[alert7@redhat62 alert7]$ ldd test

        not a dynamic executable
```

-static 出来的代码已经没有 PLT 了，GOT 虽然有，已经全部为 0 了。

AT&T 的汇编格式

一　基本语法

语法上主要有以下几个不同.

★ 寄存器命名原则

AT&T: %eax Intel: eax

★源/目的操作数顺序

AT&T: movl %eax,%ebx Intel: mov ebx,eax

★常数/立即数的格式

AT&T: movl $_value,%ebx Intel: mov eax,_value
把_value 的地址放入 eax 寄存器

AT&T: movl $0xd00d,%ebx Intel: mov ebx,0xd00d

★ 操作数长度标识

AT&T: movw %ax,%bx Intel: mov bx,ax

★寻址方式

AT&T: immed32(basepointer,indexpointer,indexscale)
Intel: [basepointer + indexpointer*indexscale + imm32)

Linux 工作于保护模式下，用的是 3 2 位线性地址，所以在计算地址时不用考虑 segment:offset 的问题．上式中的地址应为：
imm32 + basepointer + indexpointer*indexscale

下面是一些例子：

★直接寻址

AT&T: _booga ; _booga 是一个全局的 C 变量

注意加上$是表示地址引用，不加是表示值引用．
注：对于局部变量，可以通过堆栈指针引用．

Intel: [_booga]

★寄存器间接寻址

AT&T: (%eax)
Intel: [eax]

★变址寻址

AT&T: _variable(%eax)
Intel: [eax + _variable]

AT&T: _array(,%eax,4)
Intel: [eax*4 + _array]

AT&T: _array(%ebx,%eax,8)
Intel: [ebx + eax*8 + _array]

二　基本的行内汇编

·基本的行内汇编很简单，一般是按照下面的格式:

asm("statements");

例如：asm("nop"); asm("cli");

·asm　和　__asm__是完全一样的.

·如果有多行汇编，则每一行都要加上　"\n\t"

例如：

```
asm( "pushl %eax\n\t"
"movl $0,%eax\n\t"
"popl %eax");
```

实际上 gcc 在处理汇编时，是要把 asm(...)的内容"打印"到汇编文件中，所以格式控制字符是必要的.

再例如：

```
asm("movl %eax,%ebx");
asm("xorl %ebx,%edx");
asm("movl $0,_booga);
```

在上面的例子中，由于我们在行内汇编中改变了 edx 和 ebx 的值，但是由于 gcc 的特殊的处理方法，即先形成汇编文件，再交给 GAS 去汇编，所以 GAS 并不知道我们已经改变了 edx 和 ebx 的值，如果程序的上下文需要 edx 或 ebx 作暂存，这样就会引起严重的后果．对于变量_booga 也存在一样的问题．为了解决这个问题，就要用到扩展的行内汇编语法.

三　扩展的行内汇编

扩展的行内汇编类似于 Watcom.

基本的格式是：

```
asm ( "statements" : output_regs : input_regs : clobbered_regs);
```

clobbered_regs 指的是被改变的寄存器.

下面是一个例子(为方便起见，我使用全局变量）：

```
int count=1;
int value=1;
int buf[10];
void main()
{
asm(
"cld \n\t"
"rep \n\t"
```

```
"stosl"
:
: "c" (count), "a" (value) , "D" (buf[0])
: "%ecx","%edi" );
}
```

得到的主要汇编代码为：

```
movl count,%ecx
movl value,%eax
movl buf,%edi
#APP
cld
rep
stosl
#NO_APP
```

cld,rep,stos 就不用多解释了．这几条语句的功能是向 buf 中写上 count 个 value 值．冒号后的语句指明输入，输出和被改变的寄存器．通过冒号以后的语句，编译器就知道你的指令需要和改变哪些寄存器，从而可以优化寄存器的分配．

其中符号"c"(count)指示要把 count 的值放入 ecx 寄存器

类似的还有：

a eax
b ebx
c ecx
d edx
S esi
D edi
I 常数值，(0 - 31)
q,r 动态分配的寄存器
g eax,ebx,ecx,edx 或内存变量
A 把 eax 和 edx 合成一个 64 位的寄存器(use long longs)

我们也可以让 gcc 自己选择合适的寄存器．
如下面的例子：

```
asm("leal (%1,%1,4),%0"
: "=r" (x)
: "0" (x) );
```

这段代码实现 5*x 的快速乘法．
得到的主要汇编代码为：

```
movl x,%eax
#APP
```

```
leal (%eax,%eax,4),%eax
#NO_APP
movl %eax,x
```

几点说明：

1.使用 q 指示编译器从 eax,ebx,ecx,edx 分配寄存器．使用 r 指示编译器从 eax,ebx,ecx,edx,esi,edi 分配寄存器.

2.我们不必把编译器分配的寄存器放入改变的寄存器列表，因为寄存器已经记住了它们.

3."="是标示输出寄存器，必须这样用.

4.数字%n 的用法：

数字表示的寄存器是按照出现和从左到右的顺序映射到用"r"或"q"请求的寄存器．如果我们要重用"r"或"q"请求的寄存器的话，就可以使用它们.

5.如果强制使用固定的寄存器的话，如不用%1,而用 ebx,则 asm("leal (%%ebx,%%ebx,4),%0"

```
: "=r" (x)
: "0" (x) );
```

注意要使用两个%,因为一个%的语法已经被%n 用掉了.

下面可以来解释 letter 4854-4855 的问题：

1、变量加下划线和双下划线有什么特殊含义吗？
加下划线是指全局变量，但我的 gcc 中加不加都无所谓.

2、以上定义用如下调用时展开会是什么意思？

```
#define _syscall1(type,name,type1,arg1) \
type name(type1 arg1) \
{ \
long __res; \
/* __res 应该是一个全局变量  */
__asm__ volatile ("int $0x80" \
/* volatile 的意思是不允许优化，使编译器严格按照你的汇编代码汇编*/
: "=a" (__res) \
/* 产生代码  movl %eax, __res */
: "0" (__NR_##name),"b" ((long)(arg1))); \
/* 如果我没记错的话，这里##指的是两次宏展开.
   即用实际的系统调用名字代替"name",然后再把__NR_...展开.
   接着把展开的常数放入 eax，把 arg1 放入 ebx */
```

```
if (__res >= 0) \
return (type) __res; \
errno = -__res; \
return -1; \
}
```

x86 内联汇编

本文提供了在 Linux 平台上使用和构造 x86 内联汇编的概括性介绍。他介绍了内联汇编及其各种用法的基础知识，提供了一些基本的内联汇编编码指导，并解释了在 Linux 内核中内联汇编代码的一些实例。

如果您是 Linux 内核的开发人员，您会发现自己经常要对与体系结构高度相关的功能进行编码或优化代码路径。您很可能是通过将汇编语言指令插入到 C 语句的中间（又称为内联汇编的一种方法）来执行这些任务的。让我们看一下 Linux 中内联汇编的特定用法。（我们将讨论限制在 IA32 汇编。）

简述

GNU 汇编程序简述

让我们首先看一下 Linux 中使用的基本汇编程序语法。GCC（用于 Linux 的 GNU C 编译器）使用 AT&T 汇编语法。下面列出了这种语法的一些基本规则。（该列表肯定不完整；只包括了与内联汇编相关的那些规则。）

寄存器命名

寄存器名称有 % 前缀。即，如果必须使用 eax，它应该用作 %eax。

源操作数和目的操作数的顺序

在所有指令中，先是源操作数，然后才是目的操作数。这与将源操作数放在目的操作数之后的 Intel 语法不同。

mov %eax, %ebx, transfers the contents of eax to ebx.

操作数大小

根据操作数是字节 (byte)、字 (word) 还是长型 (long)，指令的后缀可以是 b、w 或 l。这并不是强制性的；GCC 会尝试通过读取操作数来提供相应的后缀。但手工指定后缀可以改善代码的可读性，并可以消除编译器猜测不正确的可能性。

```
movb %al, %bl -- Byte move
movw %ax, %bx -- Word move
movl %eax, %ebx -- Longword move
```

立即操作数

通过使用 $ 指定直接操作数。

movl $0xffff, %eax -- will move the value of 0xffff into eax register.

间接内存引用

任何对内存的间接引用都是通过使用 ( ) 来完成的。

movb (%esi), %al -- will transfer the byte in the memory
pointed by esi into al register

内联汇编

内联汇编

GCC 为内联汇编提供特殊结构，它具有以下格式：

```
asm ( assembler template
    : output operands              (optional)
    : input operands               (optional)
    : list of clobbered registers  (optional)
    );
```

本例中，汇编程序模板由汇编指令组成。输入操作数是充当指令输入操作数使用的 C 表达式。输出操作数是将对其执行汇编指令输出的 C 表达式。

内联汇编的重要性体现在它能够灵活操作，而且可以使其输出通过 C 变量显示出来。因为它具有这种能力，所以 "asm" 可以用作汇编指令和包含它的 C 程序之间的接口。

一个非常基本但很重要的区别在于简单内联汇编只包括指令，而扩展内联汇编包括操作数。要说明这一点，考虑以下示例：

内联汇编的基本要素

```
{
  int a=10, b;
  asm ("movl %1, %%eax;
       movl %%eax, %0;"
       :"=r"(b)  /* output */
       :"r"(a)       /* input */
      :"%eax"); /* clobbered register */
}
```

在上例中，我们使用汇编指令使 "b" 的值等于 "a"。请注意以下几点：

"b" 是输出操作数，由 %0 引用，"a" 是输入操作数，由 %1 引用。

"r" 是操作数的约束，它指定将变量 "a" 和 "b" 存储在寄存器中。请注意，输出操作数约束应该带有一个约束修饰符 "="，指定它是输出操作数。

要在 "asm" 内使用寄存器 %eax，%eax 的前面应该再加一个 %，换句话说就是 %%eax，因为 "asm" 使用 %0、%1 等来标识变量。任何带有一个 % 的数都看作是输入 / 输出操作数，而不认为是寄存器。
第三个冒号后的修饰寄存器 %eax 告诉将在 "asm" 中修改 GCC %eax 的值，这样 GCC 就不使用该寄存器存储任何其它的值。

movl %1, %%eax 将 "a" 的值移到 %eax 中，movl %%eax, %0 将 %eax 的内容移到 "b" 中。
因为 "b" 被指定成输出操作数，因此当 "asm" 的执行完成后，它将反映出更新的值。换句话说，对 "asm" 内 "b" 所做的更改将在 "asm" 外反映出来。

程序模板

汇编程序模板是一组插入到 C 程序中的汇编指令（可以是单个指令，也可以是一组指令）。每条指令都应该由双引号括起，或者整组指令应该由双引号括起。每条指令还应该用一个定界符结尾。有效的定界符为新行 (\n) 和分号 (;)。 '\n' 后可以跟一个 tab(\t) 作为格式化符号，增加 GCC 在汇编文件中生成的指令的可读性。 指令通过数 %0、%1 等来引用 C 表达式（指定为操作数）。

如果希望确保编译器不会在 "asm" 内部优化指令，可以在 "asm" 后使用关键字 "volatile"。如果程序必须与 ANSI C 兼容，则应该使用 __asm__ 和 __volatile__，而不是 asm 和 volatile。

操作数

C 表达式用作 "asm" 内的汇编指令操作数。在汇编指令通过对 C 程序的 C 表达式进行操作来执行有意义的作业的情况下，操作数是内联汇编的主要特性。

每个操作数都由操作数约束字符串指定，后面跟用括弧括起的 C 表达式，例如："constraint" (C expression)。操作数约束的主要功能是确定操作数的寻址方式。

可以在输入和输出部分中同时使用多个操作数。每个操作数由逗号分隔开。

在汇编程序模板内部，操作数由数字引用。如果总共有 n 个操作数（包括输入和输出），那么第一个输出操作数的编号为 0，逐项递增，最后那个输入操作数的编号为 n-1。总操作数的数目限制在 10，如果机器描述中任何指令模式中的最大操作数数目大于 10，则使用后者作为限制。

修饰寄存器列表

如果 "asm" 中的指令指的是硬件寄存器，可以告诉 GCC 我们将自己使用和修改它们。这样，GCC 就不会假设它装入到这些寄存器中的值是有效值。通常不需要将输入和输出寄存器列为 clobbered，因为 GCC 知道 "asm" 使用它们（因为它们被明确指定为约束）。不过，如果指令使用任何其它的寄存器，无论是明确的还是隐含的（寄存器不在输入约束列表中出现，也不在输出约束列表中出现），寄存器都必须被指定为修饰列表。修饰寄存器列在第三个冒号之后，其名称被指定为字符串。

至于关键字，如果指令以某些不可预知且不明确的方式修改了内存，则可能将 "memory" 关键字添加到修饰寄存器列表中。这样就告诉 GCC 不要在不同指令之间将内存值高速缓存在寄存器中。

操作数约束

前面提到过，"asm" 中的每个操作数都应该由操作数约束字符串描述，后面跟用括弧括起的 C 表达式。操作数约束主要是确定指令中操作数的寻址方式。约束也可以指定：

·是否允许操作数位于寄存器中，以及它可以包括在哪些种类的寄存器中
·操作数是否可以是内存引用，以及在这种情况下使用哪些种类的地址
·操作数是否可以是立即数

约束还要求两个操作数匹配。

常用约束

在可用的操作数约束中，只有一小部分是常用的；下面列出了这些约束以及简要描述。有关操作数约束的完整列表，请参考 GCC 和 GAS 手册。

寄存器操作数约束 (r)

使用这种约束指定操作数时，它们存储在通用寄存器中。请看下例：

```
asm ("movl %%cr3, %0\n" :"=r"(cr3val));
```

这里，变量 cr3val 保存在寄存器中，%cr3 的值复制到寄存器上，cr3val 的值从该寄存器更新到内存中。指定 "r" 约束时，GCC 可以将变量 cr3val 保存在任何可用的 GPR 中。要指定寄存器，必须通过使用特定的寄存器约束直接指定寄存器名。

a %eax
b %ebx
c %ecx
d %edx
S %esi
D %edi

内存操作数约束 (m)

当操作数位于内存中时，任何对它们执行的操作都将在内存位置中直接发生，这与寄存器约束正好相反，后者先将值存储在要修改的寄存器中，然后将它写回内存位置中。但寄存器约束通常只在对于指令来说它

们是绝对必需的，或者它们可以大大提高进程速度时使用。当需要在 "asm" 内部更新 C 变量，而您又确实不希望使用寄存器来保存其值时，使用内存约束最为有效。例如，idtr 的值存储在内存位置 loc 中：

```
("sidt %0\n" : :"m"(loc));
```

匹配（数字）约束

在某些情况下，一个变量既要充当输入操作数，也要充当输出操作数。可以通过使用匹配约束在 "asm" 中指定这种情况。

```
asm ("incl %0" :"=a"(var):"0"(var));
```

在匹配约束的示例中，寄存器 %eax 既用作输入变量，也用作输出变量。将 var 输入读取到 %eax，增加后将更新的 %eax 再次存储在 var 中。这里的 "0" 指定第 0 个输出变量相同的约束。即，它指定 var 的输出实例只应该存储在 %eax 中。该约束可以用于以下情况：

·输入从变量中读取，或者变量被修改后，修改写回到同一变量中
·不需要将输入操作数和输出操作数的实例分开
·使用匹配约束最重要的意义在于它们可以导致有效地使用可用寄存器。

示例

一般内联汇编用法示例

以下示例通过各种不同的操作数约束说明了用法。有如此多的约束以至于无法将它们一一列出，这里只列出了最经常使用的那些约束类型。

寄存器约束

"asm" 和寄存器约束 "r"

让我们先看一下使用寄存器约束 r 的 "asm"。我们的示例显示了 GCC 如何分配寄存器，以及它如何更新输出变量的值。

```
int main(void)
{
   int x = 10, y;
   asm ("movl %1, %%eax;
        "movl %%eax, %0;"
        :"=r"(y)  /* y is output operand */
        :"r"(x)       /* x is input operand */
        :"%eax"); /* %eax is clobbered register */
}
```

在该例中，x 的值复制为 "asm" 中的 y。x 和 y 都通过存储在寄存器中传递给 "asm"。为该例生成的汇编代码如下：

```
main:
pushl %ebp
movl %esp,%ebp
subl $8,%esp
movl $10,-4(%ebp)
movl -4(%ebp),%edx  /* x=10 is stored in %edx */
#APP   /* asm starts here */
movl %edx, %eax     /* x is moved to %eax */
movl %eax, %edx     /* y is allocated in edx and updated */
#NO_APP /* asm ends here */
movl %edx,-8(%ebp)  /* value of y in stack is updated with the value in %edx */
```

当使用 "r" 约束时，GCC 在这里可以自由分配任何寄存器。在我们的示例中，它选择 %edx 来存储 x。在读取了 %edx 中 x 的值后，它为 y 也分配了相同的寄存器。

因为 y 是在输出操作数部分中指定的，所以 %edx 中更新的值存储在 -8(%ebp)，堆栈上 y 的位置中。如果 y 是在输入部分中指定的，那么即使它在 y 的临时寄存器存储值 (%edx) 中被更新，堆栈上 y 的值也不会更新。

因为 %eax 是在修饰列表中指定的，GCC 不在任何其它地方使用它来存储数据。

输入 x 和输出 y 都分配在同一个 %edx 寄存器中，假设输入在输出产生之前被消耗。请注意，如果您有许多指令，就不是这种情况了。要确保输入和输出分配到不同的寄存器中，可以指定 & 约束修饰符。下面是添加了约束修饰符的示例。

```
int main(void)
{
   int x = 10, y;
   asm ("movl %1, %%eax;
        "movl %%eax, %0;"
       :"=&r"(y) /* y is output operand, note the & constraint modifier. */
       :"r"(x)       /* x is input operand */
       :"%eax"); /* %eax is clobbered register */
}
```

以下是为该示例生成的汇编代码，从中可以明显地看出 x 和 y 存储在 "asm" 中不同的寄存器中。

```
main:
pushl %ebp
movl %esp,%ebp
subl $8,%esp
movl $10,-4(%ebp)
```

```
movl -4(%ebp),%ecx  /* x, the input is in %ecx */
#APP
   movl %ecx, %eax
   movl %eax, %edx     /* y, the output is in %edx */
#NO_APP
movl %edx,-8(%ebp)
```

特定寄存器约束的使用

现在让我们看一下如何将个别寄存器作为操作数的约束指定。在下面的示例中，cpuid 指令采用 %eax 寄存器中的输入，然后在四个寄存器中给出输出：%eax、%ebx、%ecx、%edx。对 cpuid 的输入（变量 "op"）传递到 "asm" 的 eax 寄存器中，因为 cpuid 希望它这样做。在输出中使用 a、b、c 和 d 约束，分别收集四个寄存器中的值。

```
asm ("cpuid"
     : "=a" (_eax),
     "=b" (_ebx),
     "=c" (_ecx),
     "=d" (_edx)
     : "a" (op));
```

在下面可以看到为它生成的汇编代码（假设 _eax、_ebx 等... 变量都存储在堆栈上）：

```
movl -20(%ebp),%eax /* store 'op' in %eax -- input */
#APP
cpuid
#NO_APP
movl %eax,-4(%ebp)  /* store %eax in _eax -- output */
movl %ebx,-8(%ebp)  /* store other registers in
movl %ecx,-12(%ebp)
respective output variables */
movl %edx,-16(%ebp)
```

strcpy 函数可以通过以下方式使用 "S" 和 "D" 约束来实现：

```
asm ("cld\n
     rep\n
     movsb"
     : /* no input */
     :"S"(src), "D"(dst), "c"(count));
```

通过使用 "S" 约束将源指针 src 放入 %esi 中，使用 "D" 约束将目的指针 dst 放入 %edi 中。因为 rep 前缀需要 count 值，所以将它放入 %ecx 中。

在下面可以看到另一个约束，它使用两个寄存器 %eax 和 %edx 将两个 32 位的值合并在一起，然

后生成一个 64 位的值：

```
#define rdtscll(val) \
__asm__ __volatile__ ("rdtsc" : "=A" (val))
```

The generated assembly looks like this (if val has a 64 bit memory space).

```
#APP
rdtsc
#NO_APP
movl %eax,-8(%ebp)  /* As a result of A constraint
movl %edx,-4(%ebp)
%eax and %edx serve as outputs */
```

Note here that the values in %edx:%eax serve as 64 bit output.

匹配约束

使用匹配约束

在下面将看到系统调用的代码，它有四个参数：

```
#define _syscall4(type,name,type1,arg1,type2,arg2,type3,arg3,type4,arg4) \
type name (type1 arg1, type2 arg2, type3 arg3, type4 arg4) \
{ \
long __res; \
__asm__ volatile ("int $0x80" \
: "=a" (__res) \
: "0" (__NR_##name),"b" ((long)(arg1)),"c" ((long)(arg2)), \
"d" ((long)(arg3)),"S" ((long)(arg4))); \
__syscall_return(type,__res); \
}
```

在上例中，通过使用 b、c、d 和 S 约束将系统调用的四个自变量放入 %ebx、%ecx、%edx 和 %esi 中。请注意，在输出中使用了 "=a" 约束，这样，位于 %eax 中的系统调用的返回值就被放入变量 __res 中。通过将匹配约束 "0" 用作输入部分中第一个操作数约束，syscall 号 __NR_##name 被放入 %eax 中，并用作对系统调用的输入。这样，这里的 %eax 既可以用作输入寄存器，又可以用作输出寄存器。没有其它寄存器用于这个目的。另请注意，输入（syscall 号）在产生输出（syscall 的返回值）之前被消耗（使用）。

内存操作数约束

内存操作数约束的使用

请考虑下面的原子递减操作：

```
__asm__ __volatile__(
        "lock; decl %0"
        :"=m" (counter)
        :"m" (counter));
```

为它生成的汇编类似于：

```
#APP
    lock
    decl -24(%ebp) /* counter is modified on its memory location */
#NO_APP.
```

您可能考虑在这里为 counter 使用寄存器约束。如果这样做，counter 的值必须先复制到寄存器，递减，然后对其内存更新。但这样您会无法理解锁定和原子性的全部意图，这些明确显示了使用内存约束的必要性。

修饰寄存器

使用修饰寄存器

请考虑内存拷贝的基本实现。

```
asm ("movl $count, %%ecx;
    up: lodsl;
    stosl;
    loop up;"
    :           /* no output */
    :"S"(src), "D"(dst) /* input */
    :"%ecx", "%eax" );  /* clobbered list */
```

当 lodsl 修改 %eax 时，lodsl 和 stosl 指令隐含地使用它。%ecx 寄存器明确装入 count。但 GCC 在我们通知它以前是不知道这些的，我们是通过将 %eax 和 %ecx 包括在修饰寄存器集中来通知 GCC 的。在完成这一步之前，GCC 假设 %eax 和 %ecx 是自由的，它可能决定将它们用作存储其它的数据。请注意，%esi 和 %edi 由 "asm" 使用，它们不在修饰列表中。这是因为已经声明 "asm" 将在输入操作数列表中使用它们。这里最低限度是，如果在 "asm" 内部使用寄存器（无论是明确还是隐含地），既不出现在输入操作数列表中，也不出现在输出操作数列表中，必须将它列为修饰寄存器。

不同的 CPU 下最佳编译参数

gcc 在不同的体系机构/CPU 下编译效果有不同，需要使用不同的编译参数达到最佳效果。

一、1.2 版(gcc 2.9.x 版)

-pipe -fomit-frame-pointer"
CXXFLAGS="-march=i486 -O3 -pipe -fomit-frame-pointer"

Pentium, Pentium MMX+, Celeron (Mendocino) (Intel)
CHOST="i586-pc-linux-gnu"
CFLAGS="-march=pentium -O3 -pipe -fomit-frame-pointer"
CXXFLAGS="-march=pentium -O3 -pipe -fomit-frame-pointer"

Pentium Pro/II/III/4, Celeron (Coppermine), Celeron (Willamette?) (Intel)
CHOST="i686-pc-linux-gnu"
CFLAGS="-march=i686 -O3 -pipe -fomit-frame-pointer"
CXXFLAGS="-march=i686 -O3 -pipe -fomit-frame-pointer"

Eden C3/Ezra (Via)
CHOST="i586-pc-linux-gnu"
CFLAGS="-march=i586 -O3 -pipe -fomit-frame-pointer"
CXXFLAGS="-march=i586 -O3 -pipe -fomit-frame-pointer"

Quote : I did the original gentoo install using 1.2, with gcc 2.95 using -march=i586. i686 won't work.

K6 or beyond (AMD)
CHOST="i586-pc-linux-gnu"
CFLAGS="-march=k6 -O3 -pipe -fomit-frame-pointer"
CXXFLAGS="-march=k6 -O3 -pipe -fomit-frame-pointer"

(A Duron will report "Athlon" in its /proc/cpuinfo)

Athlon (AMD)
CHOST="i686-pc-linux-gnu"
CFLAGS="-march=k6 -O3 -pipe -fomit-frame-pointer"
CXXFLAGS="-march=k6 -O3 -pipe -fomit-frame-pointer"

For the following, i don't know of any flag that enhance performances..., do you ?

PowerPC
CHOST="powerpc-unknown-linux-gnu"
CFLAGS="-O3 -pipe -fomit-frame-pointer"

CXXFLAGS="-O3 -pipe -fomit-frame-pointer"

Sparc
CHOST="sparc-unknown-linux-gnu"
CFLAGS="-O3 -pipe -fomit-frame-pointer"
CXXFLAGS="-O3 -pipe -fomit-frame-pointer"

Sparc 64
CHOST="sparc64-unknown-linux-gnu"
CFLAGS="-O3 -pipe -fomit-frame-pointer"
CXXFLAGS="-O3 -pipe -fomit-frame-pointer"

二、1.4 版(gcc 3.x 版):

i386 (Intel), do you really want to install gentoo on that ?
CHOST="i386-pc-linux-gnu"
CFLAGS="-march=i386 -O3 -pipe -fomit-frame-pointer"
CXXFLAGS="-march=i386 -O3 -pipe -fomit-frame-pointer"

i486 (Intel), do you really want to install gentoo on that ?
CHOST="i486-pc-linux-gnu"
CFLAGS="-march=i486 -O3 -pipe -fomit-frame-pointer"
CXXFLAGS="-march=i486 -O3 -pipe -fomit-frame-pointer"

Pentium 1 (Intel)
CHOST="i586-pc-linux-gnu"
CFLAGS="-march=pentium -O3 -pipe -fomit-frame-pointer"
CXXFLAGS="-march=pentium -O3 -pipe -fomit-frame-pointer"

Pentium MMX (Intel)
CHOST="i586-pc-linux-gnu"
CFLAGS="-march=pentium-mmx -O3 -pipe -fomit-frame-pointer"
CXXFLAGS="-march=pentium-mmx -O3 -pipe -fomit-frame-pointer"

Pentium PRO (Intel)
CHOST="i686-pc-linux-gnu"
CFLAGS="-march=pentiumpro -O3 -pipe -fomit-frame-pointer"
CXXFLAGS="-march=pentiumpro -O3 -pipe -fomit-frame-pointer"

Pentium II (Intel)
CHOST="i686-pc-linux-gnu"
CFLAGS="-march=pentium2 -O3 -pipe -fomit-frame-pointer"

CXXFLAGS="-march=pentium2 -O3 -pipe -fomit-frame-pointer"

Celeron (Mendocino), aka Celeron1 (Intel)
CHOST="i686-pc-linux-gnu"
CFLAGS="-march=pentium2 -O3 -pipe -fomit-frame-pointer"
CXXFLAGS="-march=pentium2 -O3 -pipe -fomit-frame-pointer"

Pentium III (Intel)
CHOST="i686-pc-linux-gnu"
CFLAGS="-march=pentium3 -O3 -pipe -fomit-frame-pointer"
CXXFLAGS="-march=pentium3 -O3 -pipe -fomit-frame-pointer"

Celeron (Coppermine) aka Celeron2 (Intel)
CHOST="i686-pc-linux-gnu"
CFLAGS="-march=pentium3 -O3 -pipe -fomit-frame-pointer"
CXXFLAGS="-march=pentium3 -O3 -pipe -fomit-frame-pointer"

Celeron (Willamette?) (Intel)
CHOST="i686-pc-linux-gnu"
CFLAGS="-march=pentium4 -O3 -pipe -fomit-frame-pointer"
CXXFLAGS="-march=pentium4 -O3 -pipe -fomit-frame-pointer"

Pentium 4 (Intel)
CHOST="i686-pc-linux-gnu"
CFLAGS="-march=pentium4 -O3 -pipe -fomit-frame-pointer"
CXXFLAGS="-march=pentium4 -O3 -pipe -fomit-frame-pointer"

Eden C3/Ezra (Via)
CHOST="i586-pc-linux-gnu"
CFLAGS="-march=i586 -m3dnow -O3 -pipe -fomit-frame-pointer"
CXXFLAGS="-march=i586 -m3dnow -O3 -pipe -fomit-frame-pointer"

quote : the ezra doesn't have any special instructions that you could optimize for, just consider is a K6-3...basically
a p2 with 3dnow

K6 (AMD)
CHOST="i586-pc-linux-gnu"
CFLAGS="-march=k6 -O3 -pipe -fomit-frame-pointer"
CXXFLAGS="-march=k6 -O3 -pipe -fomit-frame-pointer"

K6-2 (AMD)
CHOST="i586-pc-linux-gnu"
CFLAGS="-march=k6-2 -O3 -pipe -fomit-frame-pointer"

CXXFLAGS="-march=k6-2 -O3 -pipe -fomit-frame-pointer"

K6-3 (AMD)
CHOST="i586-pc-linux-gnu"
CFLAGS="-march=k6-3 -O3 -pipe -fomit-frame-pointer"
CXXFLAGS="-march=k6-3 -O3 -pipe -fomit-frame-pointer"

Athlon (AMD)
CHOST="i686-pc-linux-gnu"
CFLAGS="-march=athlon -O3 -pipe -fomit-frame-pointer"
CXXFLAGS="-march=athlon -O3 -pipe -fomit-frame-pointer"

Athlon-tbird, aka K7 (AMD)
CHOST="i686-pc-linux-gnu"
CFLAGS="-march=athlon-tbird -O3 -pipe -fomit-frame-pointer"
CXXFLAGS="-march=athlon-tbird -O3 -pipe -fomit-frame-pointer"

Athlon-tbird XP (AMD)
CHOST="i686-pc-linux-gnu"
CFLAGS="-march=athlon-xp -O3 -pipe -fomit-frame-pointer"
CXXFLAGS="-march=athlon-xp -O3 -pipe -fomit-frame-pointer"

Athlon 4(AMD)
CHOST="i686-pc-linux-gnu"
CFLAGS="-march=athlon-4 -O3 -pipe -fomit-frame-pointer"
CXXFLAGS="-march=athlon-4 -O3 -pipe -fomit-frame-pointer"

Athlon XP (AMD)
CHOST="i686-pc-linux-gnu"
CFLAGS="-march=athlon-xp -O3 -pipe -fomit-frame-pointer"
CXXFLAGS="-march=athlon-xp -O3 -pipe -fomit-frame-pointer"

Athlon MP (AMD)
CHOST="i686-pc-linux-gnu"
CFLAGS="-march=athlon-mp -O3 -pipe -fomit-frame-pointer"
CXXFLAGS="-march=athlon-mp -O3 -pipe -fomit-frame-pointer"

603 (PowerPC)
CHOST="powerpc-unknown-linux-gnu"
CFLAGS="-O3 -pipe -fomit-frame-pointer -fsigned-char"
CXXFLAGS="-O3 -pipe -fomit-frame-pointer -fsigned-char"

603e (PowerPC)
CHOST="powerpc-unknown-linux-gnu"
CFLAGS="-O3 -pipe -fomit-frame-pointer -fsigned-char"

CXXFLAGS="-O3 -pipe -fomit-frame-pointer -fsigned-char"

604 (PowerPC)
CHOST="powerpc-unknown-linux-gnu"
CFLAGS="-O3 -pipe -fomit-frame-pointer -fsigned-char"
CXXFLAGS="-O3 -pipe -fomit-frame-pointer -fsigned-char"

604e (PowerPC)
CHOST="powerpc-unknown-linux-gnu"
CFLAGS="-O3 -pipe -fomit-frame-pointer -fsigned-char"
CXXFLAGS="-O3 -pipe -fomit-frame-pointer -fsigned-char"

750 aka as G3 (PowerPC)
CHOST="powerpc-unknown-linux-gnu"
CFLAGS="-mcpu=750 -O3 -pipe -fomit-frame-pointer
-fsigned-char"
CXXFLAGS="-mcpu=750 -O3 -pipe -fomit-frame-pointer
-fsigned-char"

Note: do not use -march=

7400, aka G4 (PowerPC)
CHOST="powerpc-unknown-linux-gnu"
CFLAGS="-mcpu=7400 -O3 -pipe -fomit-frame-pointer
-fsigned-char -maltivec"
CXXFLAGS="-mcpu=7400 -O3 -pipe -fomit-frame-pointer
-fsigned-char -maltivec"

Note: do not use -march=

7450, aka G4 second generation (PowerPC)
CHOST="powerpc-unknown-linux-gnu"
CFLAGS="-mcpu=7450 -O3 -pipe -fomit-frame-pointer
-fsigned-char -maltivec"
CXXFLAGS="-mcpu=7450 -O3 -pipe -fomit-frame-pointer
-fsigned-char -maltivec"

Note: do not use -march=

PowerPC (If you don't know which one)
CHOST="powerpc-unknown-linux-gnu"
CFLAGS="-O3 -pipe -fomit-frame-pointer -fsigned-char"
CXXFLAGS="-O3 -pipe -fomit-frame-pointer -fsigned-char"

Sparc

CHOST="sparc-unknown-linux-gnu"
CFLAGS="-O3 -pipe -fomit-frame-pointer"
CXXFLAGS="-O3 -pipe -fomit-frame-pointer"

Sparc 64
CHOST="sparc64-unknown-linux-gnu"
CFLAGS="-O3 -pipe -fomit-frame-pointer"
CXXFLAGS="-O3 -pipe -fomit-frame-pointer"

代码维护

简单 cvs

简单 CVS
一、所要解决的问题
由于软件项目越来越大，也增加了软件项目管理的难度。在开发组中，每个成员都要保留一个副本，在开发中非常容易引起冲突。CVS 就是为了解决这个问题的。

a、修改同步，防止一名开发人员的修改覆盖其他人的成果。（check out、read only）

b、维护不同的版本。（按 version 查找）

c、可查找历史记录。防止 bug 的再引入。（diff）

CVS 为了解决这个问题，采用的方式是：

当开发人员对源代码进行修改时，修改的内容被登记（check in）到了 CVS 仓库（repository）中。仓库中保存了代码的主控副本，以及历次修改的历史信息。它不保存文件的每个版本，而只是简单的记录发生在每个版本间的不同，节省磁盘空间。它能做到：

a、使开发人员的目录和仓库保持一致。可以把自己的修改提交（commit）给仓库，让仓库更新自己。

b、允许代码派生。可以进行测试，如果失败，可以消除所做的修改，维持原结果。

c、检索任何一个版本。

二、使用 CVS：

a、建立仓库：设置并 export CVSROOT 变量，并设置仓库目录。比如创建 /home/cvsroot 目录，

并合理设置权限，在 .bash_profile 中加入：export CVSROOT=$HOME/cvsroot；运行 cvs init；设置让用户输入日志信息的默认的编辑器：export EDITOR=vim。如果使用的是网络，则 CVSROOT 变量的形式是： export CVSROOT=:exit:user@server:/path，比如：export CVSROOT=:exit:david@power/home/projects/repository

b、导入文件或目录：cvs import filename_or_directory vender_tag release_tag；这三个参数的含义是：在仓库中这些导入的文件所在的目录、供应商标记、发行标记。比如 cvs import step1 david start。导入时的 N 表示所导入的文件都是新文件。

c、使用时导出文件，用：cvs checkout directory_name；参数含义是仓库中所在的目录。建立新的目录，而不是获取他人的改动

d、对文件修改后，保存修改到仓库：cvs commit。

e、如果要获得他人的修改，使用 cvs update，U 表示本地的一个文件已经被更新。如果已经对文件进行了修改，而此是他人已经修改了该文件并提交，cvs 将告诉用户发生冲突和冲突的位置。

f、添加一个文件：cvs add filename; cvs commit。

g、删除一个文件：先在本地删除，然后使用 cvs remove file_name; cvs commit。

三、使用标记

可以使用标记记录某个时刻文件的内容，这在制作发行版本的过程中非常有用：

cvs tag release1.0

改动后发现不正常，重新获得这个版本，则使用：cvs checkout -r release1.0

四、测试性代码：

当其中一个开发者对代码进行改进，但未来结果不能确定时，可以使用 cvs 产生出一个代码的分之，这并不改变主控代码：cvs tag -b for_test。导出该代码的命令是：cvs checkout -r for_test example。由于在已有的目录树中不会使用该分之，因此必须重新建立目录树。如果测试成功，则要求将主控代码和测试代码合并，则先导出主控代码，然后合并：

cvs checkout;cvs update -jfor_test;cvs commit。

---

automake

通常情况下，在写完自己代码后，使用 make 命令进行编译。make 命令其实什么也不做，知识读取一个叫 Makefile 的文件，从中获得 dependence and rule，然后调用 gcc 进行编译。但是 Makefile 比较复杂，变化技巧也比较多。对于一个大的工程项目来说，如果没有一个统一的风格，在工程延续的时

候改动 Makefile 会很麻烦，也容易出错误。所以这时就有了使用 automake 的需求。使用 automake，只需要掌握一点点规则，定义一些变量，就能自动生成 Makefile。而这些 Makefile 有着统一的格式和结构，便于修改。下面就如何使用 automake 举出一个实际的例子。

2.1 使用 automake 的前提条件

在使用 automake 前，请先确认在系统中安装了如下软件：

GNU Automake

GNU Autoconf

GNU m4

perl

GNU Libtool（如果需要产生 shared library）如果没有的话，请在发行版中找相应的 rpm 包。

2.2 制作 configure 脚本

autoconf 是用来生成自动配置软件源代码脚本（configure）的工具。configure 脚本独立于 autoconf 运行，而且在运行的过程中，不需要用户的干预，通常不需要附带参数。它是用来检验软件必须的参数的。autoconf 从一个列举编译软件时所需要各种参数的模板文件中创建 configure。autoconf 需要 GNU m4 来生成该脚本。

由 autoconf 生成的脚本一般起名为 configure。当运行时，configure 创建了多个文件，并对这些文件中的配置参数赋予适当的值。由 configure 创建生成的文件有：

1。一个或多个 Makefile，在软件源代码的每个目录中都生成一个 Makefile。

2。还可选的生成 C 头文件——configurable，包含了各种 #define 声明。

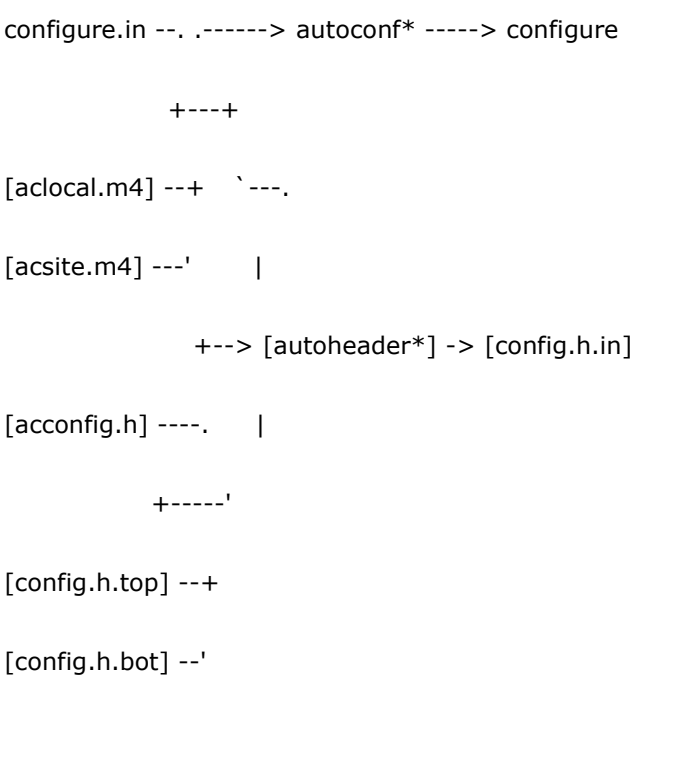
3。一个名为 config.status 的脚本，当运行时，重新生成上面的文件。

4。一个名为 config.cache 的脚本，保存运行检测的结果。

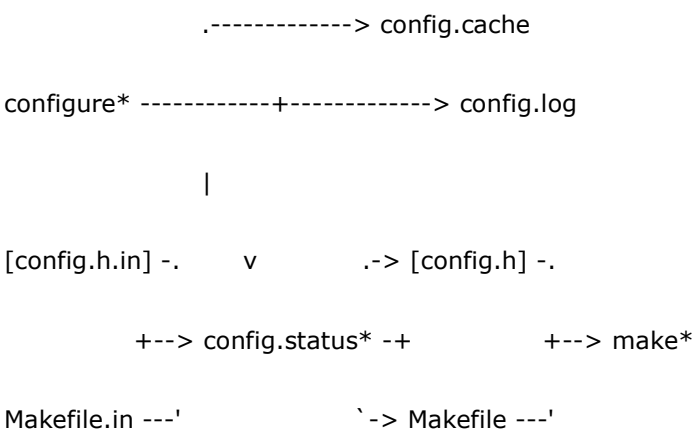
5。一个名为 config.log 的文件，保存有编译器生成的信息，用于调试 configure。

为了让 autoconf 生成 configure 脚本，需要以 configure.in 为参数调用 autoconf。如果要检测自己的各种参数，以作为对 autoconf 的补充，则需要写 aclocal.m4 和 acsite.m4 的文件。如果要使用 C 头文件，需要写 acconfig.h，并且将 autoconf 生成的 config.h.in 同软件一起发行。

your source files --> [autoscan*] --> [configure.scan] --> configure.in



Files used in configuring a software package:



编辑 configure.in 文件：

configure.in 文件中包含了对 autoconf 宏的调用，这些宏是用来检测软件所必须的各项参数的。为了能够得到 configure.in 文件，需要使用 autoscan。configure.in 文件中，在进行各项检测前，必须在最开始调用 AC_INIT，在最后调用 AC_OUTPUT。另外有些宏由于检测的关系是和在文件中的位置相关的。最好每一个宏占用一行。

使用 autoscan 创建 configure.in 文件

可以将目录做为参数调用 autoscan，如果不使用参数的化，则认为是当前目录。autoscan 将检查指定目录中的源文件，并创建 configure.scan 文件。在将 configure.scan 改名为 configure.in 文件前，需要手工改动它以进行调整。

使用 autoconf 创建 configure 脚本

不带任何参数的运行 autoconf。autoconf 将使用 m4 宏处理器和 autoconf 宏，来处理处理 configure.in 中的宏。

configure.in 中的宏：

AC_INIT（在源代码中唯一的一个文件）：configure 将检查该文件是否存在，并检查包含它的目录是否存在。

AC_OUTPUT（文件）：指定创建的输出文件。在 configure.in 文件中调用一次。文件名间用空格分开。比如：AC_OUTPUT(Makefile:templates/top.mk lib/Makefile:templates/lib.mk)

在 configure.in 中，有一些被 autoconf 宏预先定义的变量，重要的有如下几个：

bindir：安装可执行文件的目录。

includedir：C 头文件目录。

infodir：info 页安装目录。

mandir：安装手册页的目录。

sbindir：为管理员运行该该程序提供的安装路径。

srcdir：为 Makefile 提供的源代码路径。

top_srcdir：源代码的最上层目录。

LIBS：给连接程序的 -l 选项

LDFLAGS：给连接程序的 stripping（-s）和其他一些选项。

DEFS：给 C 编译器的 -D 选项。

CFLAGS：给 C 编译器的 debug 和优化选项。当调用了 AC_PROG_CC 才有效。

CPPFLAGS：头文件搜索路径（-I)和给 C 预处理器和编译器的其他选项。

CXXFLAGS：给 C++ 编译器的 debug 和优化选项。当调用了 AC_PROG_CXX 才有效。

如果在同一个目录下编译多个程序的话，使用 AC_CONFIG_SUBDIRS 宏，它的语法是：

AC_CONFIG_SUBDIRS(DIR....)：

其他重要的宏：

AC_PROG_CC：选择 C 编译器。如果在环境中不设置 CC 的话，则检测 gcc。

AC_PROG_CXX：选择 C++ 编译器。

参考文献：

Autoconf.htm

2.3 使用 automake

一般操作

Automake 工作时，读取一个叫'Makefile.am'的文件，并生成一个'Makefile.in'文件。Makefile.am 中定义的宏和目标,会指导 automake 生成指定的代码。例如，宏'bin_PROGRAMS'将导致编译和连接的目标被生成。

Makefile.am 中包含的目标和定义的宏被拷贝到生成的文件中去，这允许你添加任意代码到生成的 Makefile.in 文件中去。例如，使一个 Automake 发布中包含一个非标准的 dvs-dist 目标，Automake 的维护者用它来从它的源码控制系统制作一个发布。

请注意，GNU 生成的扩展名不被 Automake 所识别，在一个'Makefile.am'中使用这样一个扩展名会导致错误。

Automake 试图以一种聪明的方式将相邻的目标（或变量定义）注释重组。

通常,Makefile.am 中定义的目标会覆盖任何由 automake 自动生成的有相似名字的这样的目标。尽管这是种被支持的属性，但最好避免这么做，因为有些时候，生成的规则很严格。

类似的，Makefile.am 中定义的变量会覆盖任何由 automake 自动生成的变量定义。这一特性经常要比目标定义的覆盖能力更常用。请注意，很多 automake 生成的变量只用于内部使用，在将来发布时他们的名字可能会变化。

当测试一个变量定义时，Automake 降递归的测试在定义中引用的变量。例如，如果 Automake 看到这段 snippet 程序中的'foo_SOURCES'：

```
xs = a.c b.c

foo_SOURCES = c.c $(xs)
```

它将使用文件：'a.c','b.c'和'c.c' 作为 foo_SOURCES 的内容.

Automake 也允许不被拷贝到输出的注释形式,所有以'##'开头的行将被 Automake 完全忽略.

深度

Automake 支持三种目录层次:'flat', 'shallow', 'deep'.

flat: 所有的文件都在一个目录中. 相应的 Makefile.am 中缺少 SUBDIRS 宏. termutils 是一个例子.

deep: 所有的资源都在子目录中,指定曾目录主要包含配置信息.GNU cpio 是一个很好的例子.GNU tar.相应的最顶层 Makefile.am 中将包含一个 SUBDIR 宏,但没有其他的宏来定义要创建的对象.

shallow: 主资源存在于最顶层目录,而不同的部分(典型的,库函数)在子目录中.Automake 就是这样的一个包.

严格性

当 Automake 被 GNU 包维护者使用时,它的确努力去适应,但不要试图使用所有的 GNU 惯例.

目前,Automake 支持三种严格性标准:

foreign:Automake 将只检查绝对必须的东西.

gnu:Automake 将尽可能多的检查以适应 GNU 标准, 这是默认项.

gnits:Automake 将进行检查，以适应"尚未成文"的 Gnits 标准。 他们基于 GNU 标准，但更详尽。除非您是 Gnits 标准的制定者。建议您最好避免这个选项，指导该标准正式发布。

统一命名规范

Automake 变量一般遵循一套统一的命名规范以很容易的决定如何创建和安装程序（和其他派生对象）。给规范还支持 configure 时动态决定创建规则。

在 make 时，一些变量被用于决定那些对象要被创建。写变量叫做 primary variables。例如，PROGRAM 变量包括一个要被编译和连接的程序列表。

另一个变量集用于决定被创建变量被安装到哪里。这些变量以相应的主变量命名，但加一个前缀，表示那些标准目录应被用作安装路径。这些标准目录的名称规定在 GNU 标准中。Automake 用 pkglibdir，pkgincludedir 和 pkgdatadir 来展开这一列表。他们和没有 pkg 前缀的版本一样，只不过有'@PACHAGE@'扩展，PKGLIBDIR 被定义为 $(DATADIR)/@PACKAGE@.

对每一个主变量，有一个 EXTRA_前缀的变量。这个变量用于列出所有对象，至于哪些变量被创建，哪些变量不被创建则取决于 configure。之所以需要这个变量，是因为 Automake 必须静态的指导要创建对象的完整列表以便生成一个‘Makefile。in’文件。

例如，cpio 在 configure 时决定创建那些程序。一些程序被安装在 bindir，一些被安装在 sbindir：

EXTRA_PROGRAMS = mt rmt

bin_PROGRAMS = cpoi pax

sbin_PRGRAMS = @PROGRAMS@

定义没有前缀的主变量是错误的（如：PROGRAMS）。值得注意的是，“dir”在作为构成变量名时会被忽略。一次，我们写成 bin_PROGRAMS 而不是 bindir_PROGRAMS.

不是每一种对象都得以安装在每个目录下。Automake 将标记那些他认为是错误的尝试，他也能够诊断一些明显的目录名拼写错误。

有时标准目录--被 Automake 使用的--不过用。特别的，有时为了清晰，将对象安装在一些预定义的子目录，是很有用的。Automake 允许你增加安装目录。如果以一个变量名（如，zar）加上 dir 的另一个变量（如，zardir）被定义了，那么他也是合法的。

例如，如果 HTML 没支持 Automake 的一下部分，你就可以用他来安装 HTML 源文件：

htmldir = $(prefix)/html

html_DATA = automake.html

“noinst”前缀专门指定有问题的对象不被安装。

“check”前缀表示有问题的对象知道 make check 命令被执行猜被创建。

可用的主变量是 'PROGRAMS','LIBRARIES','LISP','SCRIPTS','DATA','HEADERS','MANS'和 'TEXINFOS' 导出变量是如何命名的

有时一个 Makefile 变量名有一些用户支持的文本导出。例如程序名被重写进 Makefile 宏名称。Automake 读取这些文本，所以他不必遵循命名规则。当生成后引用时名称中的字符除了字母，数字，下划线夺回被转换为下划线。例如，如果你的程序里有 sniff-glue,则导出变量名将会是 sniff_glue_SOURCES,而不是 sniff-glue_SOURCES.

一些例子

一个完整简单的例子

假设你写了一个名为 zardoz 的程序。

第一步，更新你的 configure.in 文件以包含 automake 所需的命令。最简单的办法就是在 AC_INIT 后加一个 AM_INIT_AUTOMAKE 调用：

AM_INIT_AUTOMAKE(zardoz, 1.0)

如果你的程序没有任何复杂的因素。这是最简单的办法。

现在，你必须重建‘configure’文件。这样做，你必须告诉 autoconf 如何找到你所用的新宏。最简单的方式是使用 aclocal 程序来 生成你的‘aclocal.m4’.aclocal 让你将你的宏加进‘acincluide.m4’,所以你只需重命名并运行他

mv aclocal.m4 acinclude.m4

aclocal

autoconf

现在是为你的 zardoz 写 Makefile.am 的时候了。zardoz 是一个用户程序，所以逆向将他安装在其他用户程序安装的目录。zardoz 还有 一些 Texinfo 文档。你的 configure.in 脚本使用 AC_REPLACE_FUNCS，所以你需要链接‘@LIBOBJS@’

bin_PROGRAMS = zardoz

zardoz_SOURCES = main.c head.c float.c vortex9.c gun.c

zardoz_LDADD = @LIBOBJS@

info_TEXINFOS = zardoz.texi

现在你可以运行 Automake 以生成你的 Makefile.in 文件。

一个经典的程序

hello 以其简单和多面幸而闻名。着一段将显示在 Hello 包 Automake 如何被使用。

下面是

dnl 用 autoconf 处理它以产生一个 configure 脚本.

AC_INIT(src/hello.c)

AM_INIT_AUTOMAKE(hello, 1.3.11)

AM_CONFIG_HEADER(config.h)

dnl Set of available languages

ALL_LINGUAS="de fr es ko nl no pl pt sl sv"

dnl Checks for programs.

AC_PROG_CC

AC_ISC_POSIX

dnl Checks for libraries.

dnl Checks for header files.

AC_STDC_HEADERS

AC_HAVE_HEADERS(string.h fcntl.h sys/file.h sys/param.h)

dnl Checks for library functions.

AC_FUNC_ALLOCA

dnl Check for st_blksize in struct stat

AC_ST_BLKSIZE

dnl internationalization macros

AM_GNU_GETTEXT

AC_OUTPUT([Makefile doc/Makefile intl/Makefile po/Makefile.in \ src/Makefile tests/Makefile tests/hello], [chmod +x tests/hello])

'AM_'宏由 Automake（或 Gettext 库）提供；其余的是 Autoconf 标准宏。

top-level ‘Makefile.am’:

EXTRA_DIST = BUGS ChangeLog.O

SUBDIRS = doc intl po src tests

--如你所见,这里所有的工作时在子目录中真正完成的.

--'po' 和 'intl'目录是用 gettextize 自动生成的,这里不做进一步讨论.

在'doc/Makefile.am'文件中我们看到:

info_TEXINFOS = hello.texi

hello_TEXINFOS = gpl.texi

--这已足以创建,安装和发布手册.

这里是'tests/Makefile.am'文件:

TESTS = hello

EXTRA_DIST = hello.in testdata

--脚本'hello'被 configure 创建,并且是唯一的测试.make check 将运行它.

最后是,'src/Makefile.am',所有的真正的工作是在这里完成的:

bin_PROGRAMS = hello

hello_SOURCES = hello.c version.c getopt.c getopt1.c getopt.h system.h

hello_LDADD = @INTLLIBS@ @ALLOCA@

localedir = $(datadir)/locale

INCLUDES = -I../intl -DLOCALEDIR=\"$(localedir)\"

创建一个'Makefile.in'文件

要为一个包创建所有的'Makefile.in',在顶层目录运行 automake 不带参数的程序.automake 将自动查找每个适当的'Makefile.am'文件,并声称相应的'Makefile.in'文件.请注意,automake 会简要的察看一下包的构成;它假定一个包在顶层只有一个'configure.in'文件.如果你的包有多个'configure.in'文件,那么你必须在每一个含有'configure.in'文件的目录运行一次'automake'程序.

你可以随意送给 automake 一个参数;'.am'会加到变量名的后面,并且作为输入文件的文件名.这一特

性通常被用来自动重建一个'过时的''Makefile.in'文件.注意,automake 必须在最顶级的目录运行,及时仅仅为了重新生成某个子目录中的'Makefile.in'文件.

automake 接受以下可选参数:

-a

--add-missing

在某些特殊情况下,Automake 需要某些公共文件存在.例如,如果'configure.in'运行 AC_CANONICAL_HOST 时,就需要'config.guess'.Automake 需要几个这样的文件,这一选项会让 automake 自动的将一些缺失文件加进包中(如果可能的话).一般的,如果 Automake 告诉你某个文件不存在,你可以试一试这个选项.

--amdir = dir

在 dir 所制定的目录而不是安装目录下寻找 Automake 的数据文件,经常用于调试.

--build-dir = dir

告诉 Automake 创建目录在哪里.这一选项只用于将'dependencies'加进一个由 make dist 生成的'Makefile.in'文件中.

--cygnus

将使得所生成的'Makefile.in'文件遵循 Cygnus 规则,而不是 GNU 或 Gnits 规则.

--foreign

设置全局严格性为'gnits'.

--gnu

设置全局严格性为'gnu'.

--help

帮助

-i

--include-deps

将所有自动创建时的依赖信息包括在'Makefile.in'中.通常在制作发行时用到.

--generate-deps

创建一个'.dep_segment'文件,它合并了所有的自动生成时的依赖信息,通常用于制作发行.这在维护'SMakefile'或其他平台上的'Makefile'(如 Makefile.Dos)时非常有用.它只能与--include-deps, --srcdir-name 和 --build-dir 联合使用.注意这一选项不作其他任何处理.

--no-force

一般的,automake 创建在'configure.in'中提及的所有'Makefile.in'文件.这一选项将导致,automake 只更新那些对于与自身相关的东西过了时的'Makefile.in'文件.

-o dir

--output-dir = dir

将生成的'Makefile.in'放在指定的 dir 目录中.通常,'Makefile.in'放在与之相对应的'Makefile.am'所在目录中的.在制作发布时使用.

--srcdir-name = dir

告诉 Automake 与当前创建动作相关的源文件目录的名字.个选项仅被用于将自动创建时的依赖信息包括在'Makefile.in'中.

-v

--verbose

时 Automake 打印出正在读取或创建的文件的信息.

--version

打印 Automake 的版本号.

扫描'configure.in'文件

Automake 扫描包的'configure.in'文件来决定某些关于包的信息.'configure.in'中必须定义一些变量并且需要一些 autoconf 宏.Automake 也会利用'configure.in'文件的信息来定制它的输出.

Automake 还支持一些 autoconf 宏以使维护更为简单.

配置需求

满足 Automake 基本要求的最简单方法是使用'AM_INIT_AUTOMAKE'宏.但如果你愿意,你可以手工

来做.

自动生成 aclocale.m4

Automake 包括很多 Autoconf 宏,其中有些在某些情况下游 Automake 使用,这些宏必须被定义在你的'aclocal.m4'中.否则,autoconf 将看不到他们.

acloal 程序将根据'configure.in'自动生成'aclocal.m4'.这提供了一个方便的途径来得到 Automake 提供的宏.

Automake 所支持的 Autoconf 宏

AM_CONFIG_HEADER

AM_CYGWIN32

AM_FUNC_STRTOD

AM_FUNC_ERROR_AT_LINE

AM_FUNC_MKTIME

AM_FUNC_OBSTACK

AM_C_PROTOTYPES

AM_HEADER_TOCGWINSZ_NEEDS_SYS_IOCTL

AM_INIT_AUTOMAKE

AM_PATH_LISPDIR

AM_PROG_CC_STDC

AM_PROG_INTALL

AM_PROG_LEX

AM_SANITY_CHECK

AM_SYS_POSIX_TERMIOS

AM_TYPE_PTRDIFF_T

AM_WITH_DMALLOC

AM_WITH_REGEX

编写你自己的 aclocal 宏

Aclocal 并没有特定的宏机制,因此你可以用你自己的宏来扩展它.这以特性经常被用以制作做那些想让自己的 Autoconf 宏被其他应用程序使用的库.例如,gettext 库支持一个 AM_GNU_GETTEXT 宏,它可以被任何使用 gettext 库的包所使用.当该库被安装后,它会安装这个宏,这样,aclocal 就能够找到他了.

宏的名称应以'.m4'结尾,这样的文件将被安装在'$(datadir)/aclocal'中.

最顶层'Makefile.am'

在 non-flat 包中,顶层'Makefile.am'必须告诉 Automake 哪些子目录将被创建.这是通过 SUBDIRS 变量定义的.

SUBDIRS 宏包含一个子目录列表,以规定各种创建的过程.'Makefile'中很多目标(如,all)将不止运行所在目录中,还要运行于所有指定的子目录中.注意,在 SUBDIRS 中列出的目录并不需要包含'Makefile.am'文件而只需要'Makefile'文件.这将允许包含位于不使用 Automake 的包(如,gettext)中的库.另外,SUBDIRS 之中的目录必须是当前目录的直接子目录.比如,你不能在 SUBDIRS 中指定'src/subdir'.

在 deep 型的包中,顶层'Makefile.am'文件通常非常短.例如,下面是 HELLO 发布中的'Makefile.am':

EXTRA_DIST = BUGS ChangeLog.O README-alpha

SUBDIRS = doc intl po src tests

如果你只想创建一个包的子集,你可以覆盖 SUBDIRS(就如同 GNU inetutils 的情形)在你的'Makefile.am'中包括:

SUBDIRS = @SUBDIRS@

让后在你的'configure.in'中你可以指定:

SUBDIRS = "src doc lib po"

AC_SUBST(SUBDIRS)

这样做的结果是 automake 被欺骗了,它会在指定目录创建包,但并不真正绑定那个列表直到运行 configure.

SUBDIRS 可以包含配置的替代(如,'@DIRS@');Automake 自己并不真正检测这一变量的内容.

如果 SUBDIRS 被定义了,那么你的'configure.in' 就必须包含 AC_PROG_MAKE_SET.

SUBDIRS 的使用并不局限于顶层'Makefile.am'.Automake 可被用来构造任意深度的包.

参考文献

Automake.htm

---

diff

diff

diff 是生成源码补丁的必备工具。其命令格式为：

diff [命令行选项] 原始文件 新文件

常用命令行选项如下：

-r 递归处理目录　　-u 输出统一格式(unified format)

-N patch 里包含新文件　　-a patch 里可以包含二进制文件

它的输出在 stdout 上，所以你可能需要把它重定向到一个文件。diff 的输出有“传统格式”和“统一格式”之分，现在大都使用统一格式：

传统格式示例：

[hahalee@builder]$ diff a.txt b.txt

1a2

> here we insert a new line

3d3

< why not this third line?

统一格式示例：

[hahalee@builder]$ diff -u a.txt b.txt

--- a.txt Thu Apr 6 15:58:34 2000

+++ b.txt Thu Apr 6 15:57:53 2000

@@ -1,3 +1,3 @@

This is line one

+here we insert a new line

and this is line two

-why not this third line?

通过比较可以看出，传统格式的 patch 文件比较小，除了要删除/插入的行外没有冗余信息。统一格式则保存了上下文（缺省是上下各三行，最少需要两行），这样，patch 的时候可以允许行号不精确匹配的情况出现。另外，在 patch 文件的开头明确地用---和+++标示出原始文件和当前文件，也方便阅读。要选用统一格式，用 u 开关。

通常，我们需要对整个软件包做修改，并生成一个 patch 文件，下面是典型的操作过程。这里就要用到前面介绍的几个命令行开关了：

tar xzvf software.tar.gz # 展开原始软件包，其目录为 software

cp _a software software-orig # 做个修改前的备份

cd software

[修改，测试……]

cd ..

diff _ruNa software-orig software > software-my.patch

现在我们就可以保存 software-my.patch 做为这次修改的结果，至于原始软件包，可以不必保存。等到下次需要再修改的时候，可以用 patch 命令把这个补丁打进原始包，再继续工作。比如是在 linux kernel 上做的工作，就不必每次保存几十兆修改后的源码了。这是好处之一，好处之二是维护方便，由于 unified patch 格式有一定的模糊匹配能力，能减少原软件包升级带来的维护工作量(见后)

patch

patch 命令跟 diff 配合使用，把生成的补丁应用到现有代码上。常用命令行选项：

patch [命令行选项] [待 patch 的文件[patch]]

-pn patch level(n 是数字) -b[后缀] 生成备份，缺省是.orig

为了说明什么是 patch level，这里看一个 patch 文件的头标记。

diff -ruNa xc.orig/config/cf/Imake.cf xc.bsd/config/cf/Imake.cf

--- xc.orig/config/cf/Imake.cf Fri Jul 30 12:45:47 1999

+++ xc.new/config/cf/Imake.cf Fri Jan 21 13:48:44 2000

这个 patch 如果直接应用，它会去找 xc.orig/config/cf 目录下的 Imake.cf 文件，假如你的源码树的根目录是缺省的 xc 而不是 xc.orig,除了 mv xc xc.orig 之外,有无简单的方法应用此 patch 呢？patch level 就是为此而设：patch 会把目标路径名砍去开头 patch level 个节(由/分开的部分)。在本例中，可以用下述命令：cd xc; patch _p1 < /pathname/xxx.patch 完成操作。注意，由于没有指定 patch 文件，patch 程序默认从 stdin 读入，所以用了输入重定向。

如果 patch 成功，缺省是不建备份文件的(注：FreeBSD 下的 patch 工具缺省是保存备份)，如果你需要，可以加上 b 开关。这样把修改前的文件以"原文件名.orig"的名字做备份。如果你喜欢其它后缀名，也可以用"b 后缀"来指定。

如果 patch 失败，patch 会把成功的 patch 行给 patch 上，同时（无条件）生成备份文件和一个.rej 文件。.rej 文件里是没有成功提交的 patch 行，需要手工 patch 上去。这种情况在原码升级的时候有可能会发生。

关于二进制文件的说明：binary 文件可以原始方式存入 patch 文件。diff 可以生成(加-a 选项),patch 也可以识别。如果觉得这样的 patch 文件太难看，解决方法之一是用 uuencode 处理该 binary 文件。

---

rcs

单个文件的版本控制/管理，适合对少量文件进行版本控制，不适合小组进行项目协作开发。优点：使用简便；缺点：功能有限。RCS 常用命令有 ci/co/rcsdiff。

rcs 用一个后缀为",v"的文件保存一文件的内容和所有修改的历史信息，你可以随时取出任意一个版本，用 rcs 保存程序就不必为不同版本分别备份。下面是一个",v"文件的例子：

（太长，忽略。请看演示或自己找一个样本）

rcs 文件里记载了每次更新的日期、作者、还有更新说明（Log）。文件的最新版本内容放在 Log 之后，再后面是历次版本与其后一版本的差别，按 check in 的时间做倒序排列。这么做的原因是因为新版本的 check out 机会大些，倒序排列可优化 check out 时间。

ci _ check in，保存新版本

此命令把指定文件添加到 rcs 历史文件中，同时把该文件删除。如果当前目录下有个 RCS 目录，ci 会把历史文件存在此处，否则放在当前目录下。

[hahalee@builder]$ mkdir RCS

[hahalee@builder]$ ci wood.txt

RCS/wood.txt,v <-- wood.txt

enter description, terminated with single '.' or end of file:

NOTE: This is NOT the log message!

>> initial checkin #NOTE: 这里是给本次 checkin 做的说明

>> .

initial revision: 1.1

done

[hahalee@builder]$ ls -l RCS/

总共 4

-r--r--r-- 1 hahalee hahalee 451 Apr 7 07:27 wood.txt,v

ci 也有丰富的命令行选项，比如，可以指定 check in 的版本号，甚至可以用字符串来做版本号，请查阅 ci 的 manpage。

co _ check out，取出当前（或任意）版本

常用命令行选项：

-r[rev] 指定版本的 checkout -l[rev] 指定版本，加锁的 checkout

如不加可选的版本号，缺省是最近的版本。如果只需要一份只读的拷贝，用-r（特殊情况，如需要一份只读的当前拷贝，可以不要任何选项）。如需要对 checkout 的文件进行修改，请用-l 选项。常见的操

作流程是：

ci xxx.c; co _l xxx.c; 编辑，测试; ci xxx.c .....

在每次 checkin 的时候，版本号会自动在最低位上加 1。为了在文件中反映版本控制信息，rcs 提供了几个特殊的关键字，这里介绍Id和Log，其它的请参考 info cvs。

Id代表文件的版本标识，由文件名/版本号/最后一次 checkin 时间/最后一次 checkin 的用户这几项组成，比如：

$Id: wood.txt,v 1.3 2000/04/07 00:06:52 hahalee Exp $

如果需要更详细的信息，可以用Log，Log被扩展为该文件的所有修改日期和备注，例：

```
/* $Log: wood.txt,v $
* Revision 1.2 2000/04/07 00:29:12 hahalee
* This is my second checkin
*
* Revision 1.1 2000/04/07 00:28:39 hahalee
* Initial revision
* /
```

顺便介绍一下 ident 命令。它的功能比较简单，就是从文件中搜索出 RCS 标示符并打印出来。可以用 ident /usr/sbin/sendmail 来看看。不用说，如果想在最终的 binary 文件里找到Id串，得要把它声明到一个字符串里去。很多程序里这么写：

```
#ifndef lint //这里是为了避免 lint 报告“变量未使用”
static const char rcsid[] =
"$Id: bin/sh.c,v 1.15 1999/08/27 23:13:43 wp Exp $"; //这是从 $Id$ 扩展出来的
#endif
```

rcsdiff _ 比较 revision 之间的差异.运行 diff 命令，生成 patch 文件

命令行格式：rcsdiff [选项] [-r 版本[-r 版本]] [diff 选项] 文件名

说明：如果没给出版本号，把上次 checkin 的内容同当前工作文件比较；如给出了一个版本号，就把那个版本的内容同当前工作文件比较；若给出了两个版本号，则用第一个来跟第二个比较。由于 rcsdiff

调用 diff 命令，所有的 diff 选项都可用。它的输出也是加了额外信息的 diff 格式内容，可以给 patch 使用。

rcs 里面还有 rcs，rcsclean，rlog，merge，rcsmerge 我们没有提到，有的特别简单有的特别繁琐且用得少。其中 rcs 命令可以用来对 rcs 文件进行各种精细的维护，最为复杂。

内核重编译常见故障

该程序只是简单地录一小段纯音频数据存储在 test.wav 中,用命令 cat test.wav >/dev/audio 就可以播放出来。因为程序要读写声卡的设备文件,所以你必须用相应的权限,或者你是 root

```
#include <sys/soundcard.h>
#include <stdio.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>

main()
{
  int id,fd,i,j;
  char testbuf[4096];
  id=open("/dev/audio",O_RDWR);
  //这里使用声卡驱动程序,linux 里当作一个文件读写
  fd=open("test.wav",O_RDWR);
  //一个文件

  if(id<0||fd<0){
    printf("error in open the device!\n");
    printf("id %d\t dsp%d\t seq%d\n",id,dsp,seq);
    exit(-1);
  }

  post=0;
  ioctl(id,SNDCTL_DSP_RESET,(char *)&i);
  ioctl(id,SNDCTL_DSP_SYNC,(char *)&i);
  i=1;
  ioctl(id,SNDCTL_DSP_NONBLOCK,(char *)&i);
  i=8000;
  ioctl(id,SNDCTL_DSP_SPEED,(char *)&i);
  i=1;
  ioctl(id,SNDCTL_DSP_CHANNELS,(char *)&i);
  i=8;
  ioctl(id,SNDCTL_DSP_SETFMT,(char *)&i);
```

```
  i=3;
  ioctl(id,SNDCTL_DSP_SETTRIGGER,(char *)&i);
  i=3;
  ioctl(id,SNDCTL_DSP_SETFRAGMENT,(char *)&i);
  i=1;
  ioctl(id,SNDCTL_DSP_PROFILE,(char *)&i);

  for(j=0;j<10;){
    i=read(id,testbuf,4096);
    if(i>0){
      write(fd,filebuf,i);
      j++;
    }
  }

  close(fd);
  close(id);

}
```

CVS

CVS，顾名思义，是个可以用在小组协作环境下的源码版本管理系统。同类的软件有 AT&T 的 SCCS（Source Code Control System），还有 PVCS 等。在 OpenSource 项目里 CVS 用得最为广泛，Linux kernel 不使用 CVS 来维护，下面我们将会参考 FreeBSD 的源码管理来做示例。CVS 是个相当复杂的系统，FreeBSD 甚至设有专门的"CVS 管理员"（CVS "Maester"）来全面负责项目的 CVS repository 维护。

下面介绍与 CVS 相关的若干概念和术语：

Repository: "源码仓库"，CVS 存放的项目源码历史档案

CVSROOT: 该环境变量指明 CVS Repository 存放的目录

Module: 模块。就是 CVSROOT 下的顶级目录名

Vendor Branch: 分支。在一套 Repository 里可以存放多个代码分支的历史

Release Tag: 发行标记。对于每一个版本，可以用符号来做标记

下面是一个 CVS repository 的版本衍生图，大致描绘了 FreeBSD 的版本发行情况。图中的 RELENG_3 和 RELENG_4 表示"Release Engine"，也就是 Vendor Branch，每个 Branch 分头发展，

等某个 Branch 的开发到了一定的质量水准，就做个 Release Tag。比如最近的 4.0-RELEASE 的 Release Tag 是 REL_4_0。

这些不同的 Branch 都存放在同一个 Repository Tree 里。

CVS 是个很复杂的系统，可以参考下面两个 URL 获得进一步的信息：

http://www.cyclic.com

http://www.loria.fr/~molli/cvs-index.html

（在 cvs 软件包里含有详细的文档，应当查阅 info 版本。几个 ps 文件都太老了）

下面介绍 CVS 的基本用法。

① Import 导入/创建 CVS Repository Tree

首先建一个目录作为你的 CVSROOT，然后用 cvs init 命令对其初始化（建立一系列 log,config 文件）。然后到工作目录下使用 cvs import 命令：

```
[hahalee@builder]$ mkdir /home/hahalee/CVS

[hahalee@builder]$ export CVSROOT=/home/hahalee/CVS

[hahalee@builder]$ cvs init

[hahalee@builder]$ cvs import _b 0.5.0 hftpd RELENG_0 REL_0

N hftpd/tar.h

N hftpd/auth.h

[blah...blah...blah...]

N hftpd/docs/rfcs/rfc0959.txt

N hftpd/docs/rfcs/rfc2428.txt

No conflicts created by this import
```

上述操作在$CVSROOT 下生成 hftpd 目录，可以看到里面都是后缀为“,v”的文件，这就是 import 进来的 Repository。RELENG_0 是 vendor-tag，REL_0 是 release-tag。vendor-tag 就是 vendor branch tag，可以理解为”code name”。

② Checkout 创建私有工作目录/Export

换一个空目录，运行 cvs checkout modules_name 即可：

```
[hahalee@builder]$ cvs checkout hftpd# hftpd 是我们的 module   name

cvs checkout: Updating hftpd

U hftpd/AUTHORS

U hftpd/COPYING

[blah blah blah] # 省略许多

[hahalee@builder t]$ ls -l

总共 0

drwxrwxr-x 5 hahalee hahalee 1253 Apr 7 20:08 hftpd

[hahalee@builder t]$ find ./ -type d
```

从最后一条命令的输出可看到，checkout 的工作目录里多了 CVS 目录。里面记载了 CVS 相关的信息，可以方便后续的 cvs 操作。如果纯粹是为了拷贝出最新的 source tree，可以用 export，此时不会建立 CVS 目录。

③ Update 更新

当你完成某一部分代码的时候，先不忙提交，可以把别人可能做了的其他修改 update 过来然后统一编译调试无误后再提交，这是 team work 的准则。在 checkout 出来的工作目录下（不管什么子目录），直接 cvsup update 就可以了，当然你要先把 CVSROOT 环境变量设置好。

④ Commit 提交

很简单，cvs commit。但你必须要在 checkout 出来的工作目录里提交才行：

```
[hahalee@builder]$ cvs commit

cvs commit: Examining .

cvs commit: Examining docs

cvs commit: Examining docs/man

cvs commit: Examining docs/rfcs

cvs commit: Examining tools
```

Checking in AUTHORS;

/home/hahalee/CVS/hftpd/AUTHORS,v <-- AUTHORS

new revision: 0.6; previous revision: 0.5

done

关于并发提交冲突：任何用户可以随意 checkout 他们自己的工作拷贝，commit 也是不受限制的。这样，当用户 a 和 b 分别 checkout 了 1.2 版的 c.c，然后各自对 c.c 做了修改，a 提交了他的修改，然后，当 b 提交的时候，冲突就产生了。

这时候，cvs 会做以下动作：

告诉用户 b，对 c.c 的提交发生冲突
对用户 b 当前的 c.c 做备份文件.#c.c.1.2
试图合并 a 和 b 的修改，生成新的 c.c
然后，用户 b 应当修改 c.c，去掉/合并冲突的行，并以版本 1.4 提交。

⑤ Diff

可以用类似 rcsdiff 的方法用 cvs 生成 patch，命令行语法也类似

[hahalee@builder]$ cvs diff -u -r0.5 AUTHORS

Index: AUTHORS

==========

RCS file: /home/hahalee/CVS/hftpd/AUTHORS,v

retrieving revision 0.5

retrieving revision 0.6

diff -u -r0.5 -r0.6

--- AUTHORS 2000/04/07 10:46:02 0.5

+++ AUTHORS 2000/04/07 14:05:57 0.6

@@ -1,3 +1,4 @@

+ah! let me in!

So then, who can't spell

Develloppopotamus?

Quite a lot of us.

还有一个 rdiff，用来生成两个不同的 release 之间的 patch。

⑥ 其他操作

cvs 的其他操作还包括有：

admin 管理功能

tag 对某一版本做符号标记

release 取消 checkout,删除工作目录（release 在这里是"释放"的意思）

add,remove 往 repository 里添加/删除文件

history 查看 repository 操作历史记录

⑦ CVS 的多平台特性以及 C/S 扩展

cvs 是多平台的，开发可以在多种平台比如，可以把 linux 上的 CVS Repository 通过 samba export 出来在 Windows 平台上做开发。现在很多软件包里包含有*NIX/Windows/MacOS 等多平台支持代码，cvs 的跨平台特性可提供最好的多平台开发支持。

不过，cvs 的操作是直接基于文件系统的，在需要大量远程协作的场合问题很多，远程的 NFS mount 效率太差，也会有安全问题。新版本的 cvs 自身内建了 Client/Server 支持，也可以利用 Unix 上传统的远程交互手段来通讯。

1，通过 rsh（也可用 ssh 替换）

2，使用 cvs 自带的 C/S 用户认证：pserver（缺省端口 2401）

3，使用 kerberos 的 gserver、kserver

共享库工具

strip：

nm：
size：
string：

1 创建和使用静态库

创建一个静态库是相当简单的。通常使用 ar 程序把一些目标文件（.o）组合在一起，
成为一个单独的库，然后运行 ranlib，以给库加入一些索引信息。

2 创建和使用共享库
特殊的编译和连接选项
-D_REENTRANT 使得预处理器符号 _REENTRANT 被定义，这个符号激活一些宏特性。
-fPIC 选项产生位置独立的代码。由于库是在运行的时候被调入，因此这个
选项是必需的，因为在编译的时候，装入内存的地址还不知道。如果
不使用这个选项，库文件可能不会正确运行。
-shared 选项告诉编译器产生共享库代码。
-Wl,-soname -Wl 告诉编译器将后面的参数传递到连接器。而 -soname 指定了
共享库的 soname。

可以把库文件拷贝到 /etc/ld.so.conf 中列举出的任何目录中，并以
root 身份运行 ldconfig；或者
运行 export LD_LIBRARY_PATH='pwd'，它把当前路径加到库搜索路径中去。

3 使用高级共享库特性
1> ldd 工具
ldd 用来显示执行文件需要哪些共享库，共享库装载管理器在哪里找到了需要的共享库.

2> soname

共享库的一个非常重要的，也是非常难的概念是 soname--简写共享目标名（short for shared object name）。这是一个为共享库（.so）文件而内嵌在控制数据中的名字。如前面提到的，每一个程序都有一个需要使用的库的清单。这个清单的内容是一系列库的 soname，如同 ldd 显示的那样，共享库装载器必须找到这个清单。

soname 的关键功能是它提供了兼容性的标准。当要升级系统中的一个库时，并且新库的 soname 和老的库的 soname 一样，用旧库连接生成的程序，使用新的库依然能正常运行。这个特性使得在 Linux 下，升级使用共享库的程序和定位错误变得十分容易。

在 Linux 中，应用程序通过使用 soname，来指定所希望库的版本。库作者也可以通过保留或者改变 soname 来声明，哪些版本是相互兼容的，这使得程序员摆脱了共享库版本冲突问题的困扰。

查看/usr/local/lib 目录，分析 MiniGUI 的共享库文件之间的关系

3> 共享库装载器

当程序被调用的时候，Linux 共享库装载器（也被称为动态连接器）也自动被调用。它的作用是保证程序所需要的所有适当版本的库都被调入内存。共享库装载器名字是 ld.so 或者是 ld-linux.so，这取决于 Linux libc 的版本，它必须使用一点外部交互，才能完成自己的工作。然而它接受在环境变量和配置文件中的配置信息。

文件 /etc/ld.so.conf 定义了标准系统库的路径。共享库装载器把它作为搜索路径。为了改变这个设置，必须以 root 身份运行 ldconfig 工具。这将更新 /etc/ls.so.cache 文件，这个文件其实是装载器内部使用的文件之一。

可以使用许多环境变量控制共享库装载器的操作（表 1-4+）。

表 1-4+ 共享库装载器环境变量

变量	含义
LD_AOUT_LIBRARY_PATH	除了不使用 a.out 二进制格式外，与 LD_LIBRARY_PATH 相同。
LD_AOUT_PRELOAD	除了不使用 a.out 二进制格式外，与 LD_PRELOAD 相同。
LD_KEEPDIR	只适用于 a.out 库；忽略由它们指定的目录。
LD_LIBRARY_PATH	将其他目录加入库搜索路径。它的内容应该是由冒号分隔的目录列表，与可执行文件的 PATH 变量具有相同的格式。如果调用设置用户 ID 或者进程 ID 的程序，该变量被忽略。
LD_NOWARN	只适用于 a.out 库；当改变版本号是，发出警告信息。
LD_PRELOAD	首先装入用户定义的库，使得它们有机会覆盖或者重新定义标准库。使用空格分开多个入口。对于设置用户 ID 或者进程 ID 的程序，只有被标记过的库才被首先装入。在 /etc/ld.so.perload 中指定了全局版本号，该文件不遵守这个限制。

4> 使用 dlopen

另外一个强大的库函数是 dlopen()。该函数将打开一个新库，并把它装入内存。该函数主要用来加载库中的符号，这些符号在编译的时候是不知道的。比如 Apache Web 服务器利用这个函数在运行过程中加载模块，这为它提供了额外的能力。一个配置文件控制了加载模块的过程。这种机制使得在系统中添加或者删除一个模块时，都不需要重新编译了。

可以在自己的程序中使用 dlopen()。dlopen() 在 dlfcn.h 中定义，并在 dl 库中实现。它需要两个参数：一个文件名和一个标志。文件名可以是我们学习过的库中的 soname。标志指明是否立刻计算库的依赖性。如果设置为 RTLD_NOW 的话，则立刻计算；如果设置的是 RTLD_LAZY，则在需要的时候才计算。另外，可以指定 RTLD_GLOBAL，它使得那些在以后才加载的库可以获得其中的符号。

当库被装入后，可以把 dlopen() 返回的句柄作为给 dlsym() 的第一个参数，以获得符号在库中的地址。使用这个地址，就可以获得库中特定函数的指针，并且调用装载库中的相应函数。

代码优化

Linux 是一个多用户系统，因此对用户的管理是系统管理的基本组成部分。安装 Linux 的用户很可能就是该系统的管理员，也就是权限最高的 root。通过对用户的管理，分清了用户之间的责、权、利，保证了系统安全。

对软件的评价：代码的稳定性、友好性、代码的易读性、统一的风格、技巧。

1。尽量少的使用全局变量

2。局部变量一定要初始化，特别是指针变量

3。成员函数功能单一，不要过分追求技巧，函数体不要过长。

{i = i + j;　　{int k = i;

j = i - j;　　i=j;

i = i = j;　　j=k;

}　　}

4。最好有头文件

5。关于变量名的长短问题

6。设计函数时考虑到通用性

7。申请内存时，一定先要释放。注意 if 问题。

8。对浮点数比较大小时注意不要使用 ==

9。最好不要用 goto 语句

10。所有成员函数要单出口单入口

11。函数中，要先检验参数的合法性

12。最好所有的函数都有返回值，表明错误的原因。

13。注释问题

14。类型转化一律用显示转换。

15。定义宏说，参数使用括号，结果也应该括起来

#define SUB(a,b) ((a)-(b))

3*SUB(3,4-5);

16。变量长度一定要用 sizeof 来求

17。malloc 后千万别忘 free 及使指针等于 NULL。

18。字符串拷贝时尽量少使用 sprintf，而使用 memcpy，最后别忘加上'\0'

19。慎重选择编译时的优化选项。

20。小组开发时，注意代码风格的统一。

GNU 编码标准

GNU 编码标准(Coding Standards)

原文在

http://gnu.clinux.org/prep/standards.html

看过之后随手记录了一点儿

Linux 命令行参数处理,
getopt();
getopt_long();
getopt_long_only();

在调用过程中多使用高层的接口;
eg. readdir;

signal handling facilities 信号处理:
1. BSD: signal the Best
#include <signal.h>
2. POSIX: sigaction
3. USG: signal

使用临时文件,请检查环境变量 TMPDIR
使用由它指定的目录

编码格式:

or, if you want to use ANSI C, format the definition like this:

```
static char *
concat (char *s1, char *s2)
{
  ...
}
```

In ANSI C, if the arguments don't fit nicely on one line, split it
like this:

```
int
lots_of_args (int an_integer, long a_long, short a_short,
              double a_double, float a_float)
...
```

Try to avoid having two operators of different precedence at the same
level of indentation. For example, don't write this:

```
mode = (inmode[j] == VOIDmode
        || GET_MODE_SIZE (outmode[j]) > GET_MODE_SIZE (inmode[j])
        ? outmode[j] : inmode[j]);
```

Instead, use extra parentheses so that the indentation shows the
nesting:

```
mode = ((inmode[j] == VOIDmode
         || (GET_MODE_SIZE (outmode[j]) > GET_MODE_SIZE (inmode[j])))
        ? outmode[j] : inmode[j]);
```

Insert extra parentheses so that Emacs will indent the code properly.
For example, the following indentation looks nice if you do it by hand,

```
v = rup->ru_utime.tv_sec*1000 + rup->ru_utime.tv_usec/1000
    + rup->ru_stime.tv_sec*1000 + rup->ru_stime.tv_usec/1000;
```

but Emacs would alter it. Adding a set of parentheses produces something
that looks equally nice, and which Emacs will preserve:

```
v = (rup->ru_utime.tv_sec*1000 + rup->ru_utime.tv_usec/1000
     + rup->ru_stime.tv_sec*1000 + rup->ru_stime.tv_usec/1000);
```

Format do-while statements like this:

```
do
```

```
  {
    a = foo (a);
  }
while (a > 0);
```

清洗的使用 C 语言的构造:
1.不要省略 int 类型的声明;
2.-Wall
3.不要在函数内部进行 extern 声明;
4.在函数中使用另外的形参命名方式;
5.不要在局部变量和参数中映射全局变量;

变量和函数的命名方法:
1.在定义全局变量和函数的时候,不要使用过于简单的命名方法,要通过名字反映它们的用途;
2.不要过分使用缩写;

3.使用下划线来分割名字中的单词;
4.使用枚举类型来定义 constant int,而不要用#define

不同系统间的可移植性:
1.使用 Autoconf 来进行配置;
2.define the "feature test macro" _GNU_SOURCE when compiling your C files.

调用系统函数:
1.不要使用 sprintf 的返回值;
2.vfprintf 不是都提供的;
3.main 要返回 int;
4.不要明确的声明系统函数;
5.如果必须定义系统函数的话,不要指明参数类型;
6.对于处理 string 的函数需要特别对待;

i18n,国际化:
1.要在每一个程序中使用 gettext 库;
  eg. printf (gettext ("Processing file `%s'..."));

程序的文档化:

发布过程:

Makefile 约定:

书籍

"Beginning Linux Programming"

published by Wrox Press
author: Neil Matthew and Richard Stones

"LINUX kernel Internals"

published by Addison Wesley

"Advanced Programming Language in the UNIX Environment"

published by Addison Wesley
ISBN 0-201-56317-7

author: W. Richard Stevens

"UNIX Network Programming"

published by Prentice Hall
ISBN 981-3026-53-7
author: W. Richard Stevens

"The UNIX C Shell field guide"

published by Prentice-Hall