[第二章 GNU工具链 2](#_Toc522105350)

[2．1 GNU的由来与发展 2](#_Toc522105351)

[2．2 编译器 2](#_Toc522105352)

[**2.2.1 GCC简介** 2](#_Toc522105353)

[**2.2.2 GCC特点** 2](#_Toc522105354)

[**2.2.3 GCC执行过程** 2](#_Toc522105355)

[**2.2.4 GCC基本用法与选项** 3](#_Toc522105356)

[**2.2.5 Gdb调试器** 4](#_Toc522105357)

[2．3 自动编译 6](#_Toc522105358)

[**2.3.1 Make工程管理** 6](#_Toc522105359)

[**2.3.2 Makefile结构** 6](#_Toc522105360)

[**2.3.3 makefile变量** 6](#_Toc522105361)

[**2.3.4 makefile规则** 9](#_Toc522105362)

[**2.3.5 makefile规则** 9](#_Toc522105363)

[**2.3.6 使用autotools** 10](#_Toc522105364)

[2．4 版本控制 10](#_Toc522105365)

[**2.4.1 版本管理模型** 10](#_Toc522105366)

[2.4.2常用术语介绍 11](#_Toc522105367)

[Compile 17](#_Toc522105368)

[arm-none-eabi-objcopy 17](#_Toc522105369)

[makefile 18](#_Toc522105370)

[Link 19](#_Toc522105371)

[DownLoad 20](#_Toc522105372)

# 第二章 GNU工具链

## 2．1 GNU的由来与发展

GNU 是由“GNU's Not Unix”所递规定义出的首字母缩写语。GNU计划是由Richard Stallman在1983年9月27日公开发起的。它的目标是创建一套完全自由的操作系统。Richard Stallman最早是在net.unix-wizards新闻组上公布该消息，并附带一份《GNU宣言》解释为何发起该计划的文章，其中一个重要的理由就是要“重现当年软件界合作互助的团结精神”。GNU工程已经开发了一个被称为“GNU”的、对Unix向上兼容的完整的自由软件系统(free software system)。由Richard Stallman完成的最初的GNU工程的文档被称为“GNU宣言”。

每个计算机的使用者都需要一个操作系统；如果没有自由的操作系统，那么如果他不求助于私有软件，就甚至不能开始使用一台计算机。所以自由软件议事日程的第一项就是自由的操作系统。一个操作系统不仅仅是一个内核；它还包括编译器、编辑器、电子邮件软件，和许多其他东西。因此，创作一个完整的操作系统是一项十分庞大的工作。由于Unix的全局设计已经得到认证并且广泛流传，所以GNU开发者们决定使操作系统与Unix兼容。同时这种兼容性使Unix的使用者可以容易地转移到GNU上来。

在1991年Linux的第一个版本公开发行时，GNU计划已经完成除操作系统内核之外的大部分软件，比如GNU Bash，GCC等等。于是Linus的内核和GNU软件的合理组合，构成了GNU/Linux这一优异的操作系统:一个基于Linux的GNU系统。

GCC(GNU Compiler Collection)是GNU组织开发的一个编译器。目前支持的语言有 C、C++、Objective-C、Fortran、Java和Ada等。

自由软件可以走多远？这没有限制，除非诸如版权法之类的法律完全地禁止自由软件。最终的目的是，让自由软件完成计算机用户希望完成的所有工作--从而导致自由软件的过时。

## 2．2 编译器

**2.2.1 GCC简介**

Linux系统下的GCC是GNU推出的功能强大、性能优越的多平台编译器，是GNU的代表作品之一。GCC是可以在多种硬体平台上编译出可执行程序的超级编译器，其执行效率与一般的编译器相比平均效率要高20%~30%。

最初，GCC只是一个C语言编译器，当时是“GNU C Compiler”的英文缩写。随着众多开发者的加入和GCC自身的发展，如今的GCC已经是一个包含众多语言的编译器了，其中包括 C，C++，Ada，Object-C和Java等。所以，GCC的全称也由原来的“GNU C Compiler”演变为现在的“GNU Compiler Collection”，即GNU编译器家族的意思。

**2.2.2 GCC特点**

GCC不仅是GNU/Linux上的标准编译器，而且它也是嵌入式系统开发的标准编译器，这是因为GCC支持各种不同的目标架构。本书将专注于FPGA平台的嵌入式系统开发，其中的软件部分运行在Microblaze或者PowerPC处理器上，为了使我们的应用程序能够运行在不同的目标机上，我们使用交叉编译工具对程序进行交叉编译。所谓交叉编译就是在某个主机平台上（比如PC上）编译出可在其他平台上（比如ARM上）运行代码的过程。GCC提供了40种不同的结构体系。其中包括X86，RS6000，Arm，PowerPC等等，用户可以根据实际的项目平台来进行应用程序的开发。

GCC编译器能将C、C++语言源程序、汇编程序和目标程序编译、链接成可执行文件，如果没有给出可执行文件的名字，GCC将生成一个名为a.out的文件。在Linux系统中，可执行文件没有统一的后缀，系统从文件的属性来区分可执行文件和不可执行文件。而GCC则通过后缀来区别输入文件的类别，下面我们来介绍GCC所遵循的部分约定规则：

.c为后缀的文件，C语言源代码文件；

.a为后缀的文件，是由目标文件构成的档案库文件；

.C、.cc或.cxx 为后缀的文件，是C++源代码文件；

.h为后缀的文件，是程序所包含的头文件；

.i 为后缀的文件，是已经预处理过的C源代码文件；

.ii为后缀的文件，是已经预处理过的C++源代码文件；

.m为后缀的文件，是Objective-C源代码文件；

.o为后缀的文件，是编译后的目标文件；

.s为后缀的文件，是汇编语言源代码文件；

.S为后缀的文件，是经过预编译的汇编语言源代码文件。

**2.2.3 GCC执行过程**

　　虽然我们称GCC是C语言的编译器，但使用GCC由C语言源代码文件生成可执行文件的过程不仅仅是编译的过程，而是要经历四个相互关联的步骤∶预处理(也称预编译，Preprocessing)、编译(Compilation)、汇编(Assembly)和链接(Linking)，如图2-1所示。在对程序进行开发的过程中，我们可以通过添加参数对程序单独执行其中的某个过程。

GCC

cpp

**cc1**

as(assembler)

ld(linker)

**g++**

图2-1 GCC执行过程

命令GCC首先调用cpp进行预处理，在预处理过程中，对源代码文件中的文件包含(include)、预编译语句(如宏定义define等)进行分析。接着调用cc1或g++进行编译，这个阶段根据输入文件生成以.o为后缀的目标文件。汇编过程是针对汇编语言的步骤，调用as进行工作，一般来讲，.s为后缀的汇编语言文件经过预编译和汇编之后都生成以.o为后缀的目标文件。当所有的目标文件都生成之后，GCC就调用ld命令来完成最后的关键性工作，这个阶段就是链接，当然，也可以使用GCC命令直接完成链接功能。在链接阶段，所有的目标文件被安排在可执行程序中的恰当的位置，同时，该程序所调用到的库函数也从各自所在的档案库中连到合适的地方。

**2.2.4 GCC基本用法与选项**

　在使用GCC编译器的时候，我们必须给出一系列必要的调用参数和文件名称。GCC编译器的调用参数大约有100多个，但其中多数参数很少会用到，所以这里只介绍其中最基本、最常用的参数。

　GCC最基本的用法是∶gcc [options] [filenames] ，其中options就是编译器所需要的参数，filenames给出相关的文件名称，表2-1列出了常用参数的意义。

|  |  |
| --- | --- |
| 选项 | 解释 |
| -ansi | 只支持 ANSI 标准的 C 语法 |
| -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 |
| -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 |
| -o FILE | 生成指定的输出文件 |
| -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 |
| -static | 禁止使用共享连接 |
| -w | 不生成任何警告信息 |
| -Wall | 生成所有警告信息 |

表2-1 GCC常用参数

上面我们简要介绍了GCC编译器最常用的功能和主要参数选项，更为详尽的资料可以参考http://gcc.gnu.org/。 假定我们有一个程序名为test.c的C语言源代码文件，要生成一个可执行文件，最简单的办法就是：

gcc test.c

　这时，预编译、编译链接一次完成，生成一个系统预设的名为a.out的可执行文件，对于稍为复杂的情况，比如有多个源代码文件、需要链接档案库或者有其他比较特别的要求，就要给定适当的调用选项参数。再看一个简单的例子。

　整个源代码程序由两个文件test1.c 和test2.c组成，程序中使用了系统提供的数学库，同时希望给出的可执行文件为test，这时的编译命令可以是∶

gcc test1.c test2.c -lm -o test

其中，-lm表示链接系统的数学库libm.a。

**2.2.5 Gdb调试器**

　　调试是所有程序员都会面临的问题。如何提高程序员的调试效率，更好更快的定位程序中的问题从而加快程序开发的进度，是大家共同面对的。就如读者熟知的Windows下的一些调试工具，如VC自带的如设置断点、单步跟踪等，都受到了广大用户的赞赏。那么，在Linux下有什么很好的调试工具呢？

　　Gdb调试器是一款GNU开发组织并发布的UNIX/Linux下的程序调试工具。虽然，它没有图形化的友好界面，但是它强大的功能也足以与微软的VC工具等媲美。

首先，打开Linux下的编辑器Vi或者Emacs，编辑如下代码。

/\*test.c\*/

#include <stdio.h>

int sum(int m);

int main()

{

      int i,n=0;

      sum(50);

      for(i=1; i<=50; i++)

       {

         n += i;

       }

      printf("The sum of 1-50 is %d \n", n );

 }

int sum(int m)

{

         int i,n=0;

         for(i=1; i<=m;i++)

            n += i;

         printf("The sum of 1-m is %d\n", n);

}

在保存退出后首先使用Gcc对test.c进行编译，注意一定要加上选项”-g”，这样编译出的可执行代码中才包含调试信息，否则之后Gdb无法载入该可执行文件。

[root@localhost Gdb]# gcc -g test.c -o test

虽然这段程序没有错误，但调试完全正确的程序可以更加了解Gdb的使用流程。接下来就启动Gdb进行调试。注意，Gdb进行调试的是可执行文件，而不是如”.c”的源代码，因此，需要先通过Gcc编译生成可执行文件才能用Gdb进行调试。

[root@localhost Gdb]# gdb test

GNU Gdb Red Hat Linux (6.3.0.0-1.21rh)

Copyright 2004 Free Software Foundation, Inc.

GDB is free software, covered by the GNU General Public License, and you are

welcome to change it and/or distribute copies of it under certain conditions.

Type "show copying" to see the conditions.

There is absolutely no warranty for GDB.  Type "show warranty" for details.

This GDB was configured as "i386-redhat-linux-gnu"...Using host libthread\_db library "/lib/libthread\_db.so.1".

(gdb)

可以看出，在Gdb的启动画面中指出了Gdb的版本号、使用的库文件等信息，接下来就进入了由“（gdb）”开头的命令行界面了。

（1）查看文件

在Gdb中键入”l”（list）就可以查看所载入的文件，如下所示：

(Gdb) l

1       #include <stdio.h>

2       int sum(int m);

3       int main()

4       {

5             int i,n=0;

6             sum(50);

7             for(i=1; i<=50; i++)

8              {

9                n += i;

10             }

(Gdb) l

11            printf("The sum of 1～50 is %d \n", n );

12

13      }

14      int sum(int m)

15      {

16                 int i,n=0;

17                 for(i=1; i<=m;i++)

18                      n += i;

19                 printf("The sum of 1～m is = %d\n", n);

20      }

可以看出，Gdb列出的源代码中明确地给出了对应的行号，这样就可以大大地方便代码的定位。

（2）设置断点

设置断点是调试程序中是一个非常重要的手段，它可以使程序到一定位置暂停它的运行。因此，程序员在该位置处可以方便地查看变量的值、堆栈情况等，从而找出代码的症结所在。

在Gdb中设置断点非常简单，只需在”b”后加入对应的行号即可（这是最常用的方式，另外还有其他方式设置断点）。如下所示：

(Gdb) b 6

Breakpoint 1 at 0x804846d: file test.c, line 6.

要注意的是，在Gdb中利用行号设置断点是指代码运行到对应行之前将其停止，如上例中，代码运行到第五行之前暂停（并没有运行第五行）。

（3）查看断点情况

在设置完断点之后，用户可以键入”info b”来查看设置断点情况，在Gdb中可以设置多个断点。

(Gdb) info b

Num Type           Disp Enb Address    What

1   breakpoint     keep y   0x0804846d in main at test.c:6

（4）运行代码

接下来就可运行代码了，Gdb默认从首行开始运行代码，可键入”r”（run）即可（若想从程序中指定行开始运行，可在r后面加上行号）。

(Gdb) r

Starting program: /root/workplace/Gdb/test

Reading symbols from shared object read from target memory...done.

Loaded system supplied DSO at 0x5fb000

Breakpoint 1, main () at test.c:6

6                 sum(50);

可以看到，程序运行到断点处就停止了。

（5）查看变量值

在程序停止运行之后，程序员所要做的工作是查看断点处的相关变量值。在Gdb中只需键入”p”＋变量值即可，如下所示：

(Gdb) p n

$1 = 0

(Gdb) p i

$2 = 134518440

在此处，为什么变量”i”的值为如此奇怪的一个数字呢？原因就在于程序是在断点设置的对应行之前停止的，那么在此时，并没有把”i”的数值赋为零，而只是一个随机的数字。但变量”n”是在第四行赋值的，故在此时已经为零。

（6）单步运行

单步运行可以使用命令”n”（next）或”s”（step），它们之间的区别在于：若有函数调用的时候，”s”会进入该函数而”n”不会进入该函数。因此，”s”就类似于VC等工具中的”step in”，”n”类似与VC等工具中的”step over”。它们的使用如下所示：

(Gdb) n

The sum of 1-m is 1275

7            for(i=1; i<=50; i++)

(Gdb) s

sum (m=50) at test.c:16

16              int i,n=0;

可见，使用”n”后，程序显示函数sum的运行结果并向下执行，而使用”s”后则进入到sum函数之中单步运行。

（7）恢复程序运行

在查看完所需变量及堆栈情况后，就可以使用命令”c”（continue）恢复程序的正常运行了。这时，它会把剩余还未执行的程序执行完，并显示剩余程序中的执行结果。以下是之前使用”n”命令恢复后的执行结果：

 (Gdb) c

Continuing.

The sum of 1-50 is :1275

 Program exited with code 031.

可以看出，程序在运行完后退出，之后程序处于“停止状态”。

## 2．3 自动编译

**2.3.1 Make工程管理**

到此为止，我们已经了解了如何在Linux下使用编辑器编写代码，如何使用Gcc把代码编译成可执行文件，还学习了如何使用Gdb来调试程序，那么，所有的工作看似已经完成了，为什么还需要Make这个工程管理器呢？

所谓工程管理器，顾名思义，是指管理较多的文件的。可以试想一下，有一个上百个文件的代码构成的项目，如果其中只有一个或少数几个文件进行了修改，按照之前所学的Gcc编译工具，就不得不把这所有的文件重新编译一遍，因为编译器并不知道哪些文件是最近更新的，而只知道需要包含这些文件才能把源代码编译成可执行文件。于是，程序员就不能不再重新输入数目如此庞大的文件名以完成最后的编译工作。

但是，仔细回想一下本书在2.2.3节中所阐述的编译过程，编译过程是分为编译、汇编、链接不同阶段的，其中编译阶段仅检查语法错误以及函数与变量的声明是否正确声明了，在链接阶段则主要完成是函数链接和全局变量的链接。因此，那些没有改动的源代码根本不需要重新编译，而只要把它们重新链接进去就可以了。所以，人们就希望有一个工程管理器能够自动识别更新了的文件代码，同时又不需要重复输入冗长的命令行，这样，Make工程管理器也就应运而生了。

实际上，Make工程管理器也就是个“自动编译管理器”，这里的“自动”是指它能够根据文件时间戳自动发现更新过的文件而减少编译的工作量，同时，它通过读入Makefile文件的内容来执行大量的编译工作。用户只需编写一次简单的编译语句就可以了。它大大提高了实际项目的工作效率，而且几乎所有Linux下的项目编程均会涉及到它。

**2.3.2 Makefile结构**

makefile描述了整个工程的编译规则,通过make命令自动化编译。

Make是一个解释makefile 中指令的命令工具，大多数的IDE都有这个命令，比如：

* + Delphi的make，
  + Visual C++的nmake
  + Linux下GNU的make

Makefile是Make读入的惟一配置文件，因此本节的内容实际就是讲述Makefile的编写规则。在一个Makefile中通常包含如下内容：

· **target**：是一个目标文件，可以是Object File，也可以是可执行文件，还可以是一个标签（Label）；

· **prerequisites**：要生成那个target所需要的文件或是目标；

· **Command**：make需要执行的命令 。

它的格式为：

Target：prerequisites

Command

例如，有两个文件分别为hello.c和hello.h，创建的目标体为hello.o，执行的命令为gcc编译指令：gcc –c hello.c，那么，对应的Makefile就可以写为：

hello.o: hello.c hello.h

       gcc –c hello.c –o hello.o

接着就可以使用make了。使用make的格式为：make target，这样make就会自动读入Makefile（也可以是首字母小写makefile）并执行对应target的command语句，并会找到相应的依赖文件。如下所示：

[root@localhost makefile]# make hello.o

gcc –c hello.c –o hello.o

[root@localhost makefile]# ls

hello.c  hello.h  hello.o  Makefile

 可以看到，Makefile执行了“hello.o”对应的命令语句，并生成了“hello.o”目标体。

**注意：每一个命令的第一个字符必须是“tab”键，不可使用8个“space”键替代，否则make会显示出错信息**

**2.3.3 makefile变量**

上面示例的Makefile在实际中是几乎不存在的，因为它过于简单，仅包含两个文件和一个命令，在这种情况下完全不必要编写Makefile而只需在Shell中直接输入即可，在实际中使用的Makefile往往是包含很多的文件和命令的，这也是Makefile产生的原因。下面就可给出稍微复杂一些的Makefile（2个头文件，5个C文件）进行讲解：

edit : main.o kbd.o   
 cc -o edit main.o kbd.o   
main.o : main.c defs.h  
 cc -c main.c  
kbd.o : kbd.c defs.h command.h  
 cc -c kbd.c  
clean :  
 rm edit main.o kbd.o

在这个Makefile中有三个目标体（target），分别为edit、main.o和kbd.o，其中第一个目标体的依赖文件就是后两个目标体。如果用户使用命令“make edit”，则make管理器就是找到edit目标体开始执行。

这时，make会自动检查相关文件的时间戳。首先，在检查“main.o”、“kbd.o”和“edit”三个文件的时间戳之前，它会向下查找那些把“main.o”或“kbd.o”做为目标文件的时间戳。比如，“main.o”的依赖文件为：“main.c”、“defs.h”。如果这些文件中任何一个的时间戳比“main.o”新，则命令“gcc –Wall –O -g –c main.c -o main.o”将会执行，从而更新文件“main.o”。在更新完“main.o”或“kbd.o”之后，make会检查最初的“main.o”、“kbd.o”和“edit”三个文件，只要文件“main.o”或“kbd.o”中的任比文件时间戳比“edit”新，则第二行命令就会被执行。这样，make就完成了自动检查时间戳的工作，开始执行编译工作。这也就是Make工作的基本流程。

接下来，为了进一步简化编辑和维护Makefile，make允许在Makefile中创建和使用变量。变量是在Makefile中定义的名字，用来代替一个文本字符串，该文本字符串称为该变量的值。在具体要求下，这些值可以代替目标体、依赖文件、命令以及makefile文件中其它部分。在Makefile中的变量定义有两种方式：一种是递归展开方式，另一种是简单方式。

递归展开方式定义的变量是在引用在该变量时进行替换的，即如果该变量包含了对其他变量的应用，则在引用该变量时一次性将内嵌的变量全部展开，虽然这种类型的变量能够很好地完成用户的指令，但是它也有严重的缺点，如不能在变量后追加内容（因为语句：CFLAGS = $(CFLAGS) -O在变量扩展过程中可能导致无穷循环）。

为了避免上述问题，简单扩展型变量的值在定义处展开，并且只展开一次，因此它不包含任何对其它变量的引用，从而消除变量的嵌套引用。

递归展开方式的定义格式为：VAR=var

简单扩展方式的定义格式为：VAR：=var

Make中的变量使用均使用格式为：$(VAR)

变量名是不包括“:”、“#”、“=”结尾空格的任何字符串。同时，变量名中包含字母、数字以及下划线以外的情况应尽量避免，因为它们可能在将来被赋予特别的含义。

变量名是大小写敏感的，例如变量名“foo”、“FOO”、和“Foo”代表不同的变量。

推荐在makefile内部使用小写字母作为变量名，预留大写字母作为控制隐含规则参数或用户重载命令选项参数的变量名。

下面给出了上例中用变量替换修改后的Makefile，这里用OBJS代替main.o和kbd.o，用CC代替Gcc，用CFLAGS代替“-Wall -O –g”。这样在以后修改时，就可以只修改变量定义，而不需要修改下面的定义实体，从而大大简化了Makefile维护的工作量。

经变量替换后的Makefile如下所示：

OBJS = main.o kbd.o

CC = cc

edit : $(OBJS)

       $(CC) $(OBJS) -o edit

main.o : main.c defs.h

       $(CC) -c main.c

kbd.o : kbd.c defs.h command.h

       $(CC) -c kbd.c

可以看到，此处变量是以递归展开方式定义的。

Makefile中的变量分为用户自定义变量、预定义变量、自动变量及环境变量。如上例中的OBJS就是用户自定义变量，自定义变量的值由用户自行设定，而预定义变量和自动变量为通常在Makefile都会出现的变量，其中部分有默认值，也就是常见的设定值，当然用户可以对其进行修改。

预定义变量包含了常见编译器、汇编器的名称及其编译选项。下表2-2列出了Makefile中常见预定义变量及其部分默认值。

|  |  |
| --- | --- |
| 命 令 格 式 | 含    义 |
| AR | 库文件维护程序的名称，默认值为ar |
| AS | 汇编程序的名称，默认值为as |
| CC | C编译器的名称，默认值为cc |
| CPP | C预编译器的名称，默认值为$(CC) –E |
| CXX | C++编译器的名称，默认值为g++ |
| FC | FORTRAN编译器的名称，默认值为f77 |
| RM | 文件删除程序的名称，默认值为rm –f |
| ARFLAGS | 库文件维护程序的选项，无默认值 |
| ASFLAGS | 汇编程序的选项，无默认值 |
| CFLAGS | C编译器的选项，无默认值 |
| CPPFLAGS | C预编译的选项，无默认值 |
| CXXFLAGS | C++编译器的选项，无默认值 |
| FFLAGS | FORTRAN编译器的选项，无默认值 |

表2-2 Makefile中常见预定义变量

可以看出，上例中的CC和CFLAGS是预定义变量，其中由于CC没有采用默认值，因此，需要把“CC=Gcc”明确列出来。

由于常见的Gcc编译语句中通常包含了目标文件和依赖文件，而这些文件在Makefile文件中目标体的一行已经有所体现，因此，为了进一步简化Makefile的编写，就引入了自动变量。自动变量通常可以代表编译语句中出现目标文件和依赖文件等，并且具有本地含义（即下一语句中出现的相同变量代表的是下一语句的目标文件和依赖文件）。下表2-3列出了Makefile中常见自动变量。

|  |  |
| --- | --- |
| 命 令 格 式 | 含    义 |
| $\* | 不包含扩展名的目标文件名称 |
| $+ | 所有的依赖文件，以空格分开，并以出现的先后为序，可能包含重复的依赖文件 |
| $< | 第一个依赖文件的名称 |
| $? | 所有时间戳比目标文件晚的依赖文件，并以空格分开 |
| $@ | 目标文件的完整名称 |
| $^ | 所有不重复的依赖文件，以空格分开 |
| $% | 如果目标是归档成员，则该变量表示目标的归档成员名称 |

表2-3 Makefile中常见自动变量

自动变量的书写比较难记，但是在熟练了之后会非常的方便，请读者结合下例中的自动变量改写的Makefile进行记忆。

OBJS = main.o kbd.o

CC = cc

edit : $(OBJS)

       $(CC) $^ -o $@

main.o : main.c defs.h

       $(CC) -c $< -o $@

kbd.o : kbd.c defs.h command.h

       $(CC) -c $< -o $@

另外，在Makefile中还可以使用环境变量。使用环境变量的方法相对比较简单，make在启动时会自动读取系统当前已经定义了的环境变量，并且会创建与之具有相同名称和数值的变量。但是，如果用户在Makefile中定义了相同名称的变量，那么用户自定义变量将会覆盖同名的环境变量。

|  |  |  |
| --- | --- | --- |
| edit : main.o kbd.o   cc -o edit main.o kbd.o  main.o : main.c defs.h  cc -c main.c kbd.o : kbd.c defs.h command.h  cc -c kbd.c clean :  rm edit main.o kbd.o | OBJS = main.o kbd.o  CC = cc  edit : $(OBJS)         $(CC) $(OBJS) -o edit  main.o : main.c defs.h         $(CC) -c main.c  kbd.o : kbd.c defs.h command.h         $(CC) -c kbd.c | OBJS = main.o kbd.o  CC = cc  edit : $(OBJS)         $(CC) $^ -o $@  main.o : main.c defs.h         $(CC) -c $< -o $@  kbd.o : kbd.c defs.h command.h         $(CC) -c $< -o $@ |

**2.3.4 makefile规则**

Makefile的规则是Make进行处理的依据，它包括了目标体、依赖文件及其之间的命令语句。一般的，Makefile中的一条语句就是一个规则。在上面的例子中，都显示地指出了Makefile中的规则关系，如“$(CC) $(CFLAGS) -c $< -o $@”，但为了简化Makefile的编写，make还定义了隐式规则和模式规则，下面就分别对其进行讲解。

1．隐式规则

隐含规则能够告诉make怎样使用传统的技术完成任务，这样，当用户使用它们时就不必详细指定编译的具体细节，而只需把目标文件列出即可。Make会自动搜索隐式规则目录来确定如何生成目标文件。如上例就可以写成：

 OBJS = main.o kbd.o

CC = cc

edit: $(OBJS)

      $(CC) $^ -o $@

main.o : main.c defs.h

kbd.o : kbd.c defs.h command.h

 为什么可以省略后两句呢？Make具有自动推导文件以及文件依赖关系后面的命令，没必要在每一个.o文件后写同名的.c文件，以及编译命令，此既是make的“隐式规则”。

下表2-4给出了常见的隐式规则目录：

|  |  |
| --- | --- |
| 对应语言后缀名 | 规    则 |
| C编译：.c变为.o | $(CC) –c $(CPPFLAGS) $(CFLAGS) |
| C++编译：.cc或.C变为.o | $(CXX) -c $(CPPFLAGS) $(CXXFLAGS) |
| Pascal编译：.p变为.o | $(PC) -c $(PFLAGS) |
| Fortran编译：.r变为-o | $(FC) -c $(FFLAGS) |

表2-4 Makefile中常见隐式规则目录

2．模式规则

模式规则是用来定义相同处理规则的多个文件的。它不同于隐式规则，隐式规则仅仅能够用make默认的变量来进行操作，而模式规则还能引入用户自定义变量，为多个文件建立相同的规则，从而简化Makefile的编写。

模式规则的格式类似于普通规则，这个规则中的相关文件前必须用“%”标明。使用模式规则修改后的Makefile的编写如下：

OBJS = main.o kbd.o

CC = cc

edit: $(OBJS)

      $(CC) $^ -o $@

%.o : %.c

      $(CC) -c $< -o $@

**2.3.5 makefile规则**

使用make管理器非常简单，只需在make命令的后面键入目标名即可建立指定的目标，如果直接运行make，则建立Makefile中的第一个目标。

此外make还有丰富的命令行选项，可以完成各种不同的功能。下表2-5列出了常用的make命令行选项。

|  |  |
| --- | --- |
| 命令格式 | 含    义 |
| -C dir | 读入指定目录下的Makefile |
| -f file | 读入当前目录下的file文件作为Makefile |
| -i | 忽略所有的命令执行错误 |
| -I dir | 指定被包含的Makefile所在目录 |
| -n | 只打印要执行的命令，但不执行这些命令 |
| -p | 显示make变量数据库和隐含规则 |
| -s | 在执行命令时不显示命令 |
| -w | 如果make在执行过程中改变目录，则打印当前目录名 |

表2-5 make的命令行选项

**2.3.6 使用autotools**

Makefile可以帮助make完成它的使命，但要承认的是，编写Makefile确实不是一件轻松的事，尤其对于一个较大的项目而言更是如此。那么，有没有一种轻松的手段生成Makefile而同时又能让用户享受make的优越性呢？本节要讲的autotools系列工具正是为此而设的，它只需用户输入简单的目标文件、依赖文件、文件目录等就可以轻松地生成Makefile了，这无疑是广大用户的所希望的。另外，这些工具还可以完成系统配置信息的收集，从而可以方便地处理各种移植性的问题。也正是基于此，现在Linux上的软件开发一般都用autotools来制作Makefile。

autotools是系列工具，读者首先要确认系统是否装了以下工具（可以用which命令进行查看）。

aclocal

autoscan

autoconf

autoheader

automake

使用autotools主要就是利用各个工具的脚本文件以生成最后的Makefile。其总体流程是这样的：

使用aclocal生成一个“aclocal.m4”文件，该文件主要处理本地的宏定义；

改写“configure.scan”文件，并将其重命名为“configure.in”，并使用autoconf文件生成configure文件。

用户不再需要定制不同的规则，而只需要输入简单的文件及目录名即可，这样就大大方便了用户的使用。下面的图2-1总结了上述过程：

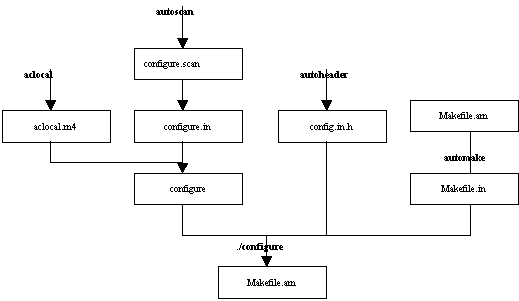


图2-1  autotools生成Makefile流程图

## 2．4 版本控制

版本控制（Revision Control），也被称为Version Control (System)或(Source) Code Management，用来管理同一信息单元的不同版本。它常用于软件开发过程中，用来管理诸如源代码、文档或其它被整个开发人员所共有的资源，藉以在开发的过程中，确保由不同人所编辑的同一档案都得到更新。版本控制会记录所有对源代码或文档的改动，并会用一个数字来加以标记，这个标记被称为版本(revision)。例如：一种简单的版本控制形式如下：最初的版本指定为“1”，当做了改变之后，版本编号增加为“2”，以此类推。

**2.4.1 版本管理模型**

1．集中式模型

传统的版本管理系统都是采用集中式模型，即所有的版本控制活动都发生在一个共享的中心版本库上。如果两个开发者在同一时间下尝试更改同一份文件，并且没有采取有效措施，那么他们很可能会互相覆盖各自的改动。我们有两种方法来解决这种情况，根据解决方法的不同，集中式版本管理系统又分为两种模型：文件锁定和版本合并。

1） 文件锁定

解决上述这个问题最简单的方法就是在一个时间段里版本库的一个文件只允许被一个人修改。首先在修改之前，先到的人会“锁定”这个文件，那么另一个人就无法修改这个文件，在先到的人释放这个“锁”之前，他只能阅读文件。这又被称为“锁定-修改-解锁”模型。早期的VSS就是使用这种模型。

锁定-修改-解锁模型的缺点就是限制太多：

* 如果一个人获得了“锁”，而他长时间没有释放“锁”，那么其他开发者就无法进行正常开发。
* 锁定可能导致无必要的线性化开发，如果两个开发者对同一文件的修改根本不冲突，他们完全可以同时对代码进行修改。

2）版本合并

许多版本控制系统，如CVS、Subversion，允许许多开发者在同一时间对同一文件进行修改。第一个提交的开发者在更改完后“提交”到中心版本库时总是成功的。在其它开发者提交的时候，版本控制系统有能力能把所有的修改合并进中心版本库。但是，如果有两个开发者的修改重叠了，那么就会引起“冲突”，这时你就会看到一组冲突集，你可以选择修改自己的代码或和其他开发者进行协商来解决冲突。

2．分布式模型

和集中式模型的客户-服务器的方法相反，分布式模型采用一种点对点的方法。通常的集中式管理系统，如 CVS，Subversion 已经得到广泛应用，但是集中式的管理存在相应的缺陷，例如对唯一的版本库过分依赖：一旦不能正常连接到集中式的版本库，整个系统陷入瘫痪。分布式模型最大的能力就在于可以维护分布式的版本库，分散的开发人员可以通过分布式版本管理建立远程的 CVS，Subversion，P4 协议的版本库镜像，选择工作在自己合适的镜像版本库，这个镜像甚至可以是本地的，整个工作可以离线进行，然后在需要的时候同步镜像版本库到主版本库。目前，采用分布式模型的版本管理系统有Bazaar、Git、Mercurial等。

### 2.4.2常用术语介绍

在具体介绍版本管理之前，首先介绍一下在进行版本管理时常用的术语：

* 分支(Branch): 在一个时间点，复制一份处于版本控制之下的文件，从这之后，这两份拷贝就可以独立的互不干扰的进行各自开发。
* 取出(Check-out): 一次“取出”，就是在本地创建一份仓库的工作拷贝。
* 提交(Commit): 一次“提交”，将本地的修改写回到仓库或合并到仓库。
* 冲突(Conflict): 当开发者们同时提交对同一文件的修改，而且版本系统不能把它们合并到一起，就会引起冲突，就需要人工来进行合并。
* 汇出(Export): 汇出和取出非常相似，只是汇出的文件不再处于版本控制之下，常用于发布代码。
* 汇入(Import): 汇入就是在第一次的时候，把本地的文件拷贝到仓库中，使它们处于版本控制之下。
* 合并(Merge): 合并就是把所有对文件的修改统一到文件里
* 仓库(Repository): 仓库就是当前的和历史的处于版本控制之下的文件所在的地方，通常在服务器端。
* 工作版本（Working copy）：从档案库中取出一个本地端的复制，所有在档案库中的档案更动，都是从一个工作版本中修改而来的，这也是这名称的由来。

**2.4.3 CVS的使用**

1． CVS在服务器端安装：

CVS服务器端可以工作在linux或windows下。windows下CVS服务器端软件为cvsnt。 RH linux通常情况下缺省安装了CVS，

　可以在终端输入：

　　　> rpm –qa | grep ‘cvs’ 来检查是否安装了cvs。

如果返回cvs版本号则表示已经安装。

若未安装cvs可以使用如下命令安装cvs包：

>rpm –ivh /mnt/cdrom/RedHat/RPMS/cvs-x.x.x-x.i386.rpm

创建一个目录作为cvsserver的根目录。

>mkdir -p /home/cvs/cvsroot

2． 初始化cvs 服务器，在系统中建立一个cvs用户组：

>groupadd pjt\_faceit

在系统中增加一个用户：

>useradd zhaofeng

并用passwd修改初始密码。

>passwd zhaofeng

修改cvs根目录的所有者和权限。

>chown zhaofeng.pjt\_faceit /home/cvs/cvsroot/ -R

>chown 775 /home/cvs/cvsroot/ -R

初始化cvs服务器：

>cvs –d /home/cvs/cvsroot/ init

初始化完成后，/home/cvs/cvsroot下生成CVSROOT目录，里面包含cvs服务器端的配置文件。

在系统服务中添加cvs服务：

使用文本编辑器打开/etc/services 查找如下内容

cvspserver 2401/tcp

cvspserver 2401/udp

注意：2401是cvs默认的通讯端口，tcp和udp为cvs所启用的网络服务。可以设定为其它闲置端口，并在客户端做同样设定。

如果没有上述内容则在文件中添加上述语句。

3．使用xinetd启用cvs服务

将cvs服务添加进xinetd，并设置启动参数：

在/etc/xinetd.d 目录下添加cvs服务的启动文件，文件名cvs或cvspserver。RH-linux通常包含一个名为cvs的文件，只需检查内容，并按自己要求加以修改。

文件内容：

service cvspserver   
{   
flags             = REUSE   
socket\_type       = stream   
wait              = no   
user              = root   
server            = /usr/bin/cvs   
server\_args       = -f --allow-root=/home/cvs/cvsroot pserver   
log\_on\_failure   += USERID   
disable           = no  
}

注意： 这里-allow-root参数的值应该和前面创建的cvsserver的根目录一致。否则后面login时会出现: no such repository的错误。如果系统本来就含有cvs配置文件，通常这个目录需要修改。flags的标志是用于setsockopt设置socket的一些属性，这里的REUSE表示socket关闭后可以立即重用，而不用等到超时后才能重用。如果配置文件已经存在一般只需要将disable的值修改为no，默认多为yes。server\_args后的-f与–allow-root之间要用space分隔，否则login时会出错。

重新启动xinetd服务

>/sbin/service xinetd restart

或 /etc/init.d/xinetd restart

注意：若xinetd没有安装，在重起服务时提示xinetd not recognized service。输入：yum install xinetd 安装xinetd服务，或者挂入RH的CD，使用rpm –ihv XXXX 安装

检查cvspserver服务是否已经启动

输入：

>netstat –l | grep cvspserver

应该有如下结果,表明cvs服务器安装成功:   
tcp 0 0 \*:cvspserver \*:\* LISTEN

4．设置CVSROOT环境变量

>export CVSROOT=:pserver:zhaofeng:123456@localhost:/home/cvs/cvsroot

>cvs login

输入刚才设定的密码登入服务器端，如果没有错误提示，则安装成功。

5． CVS的日常使用

* 取出文件到工作目录

cvs checkout project\_name

将在当前目录建立project\_name的工作目录

* 将所有文件同步到最新的版本

cvs update

* 提交修改写入到CVS库里

cvs commit -m "write some comments here" file\_name

-m 参数指定这次提交的注释

* 添加文件

创建好新文件后，比如：touch new\_file

cvs add new\_file

注意：对于图片，Word文档等非纯文本的项目，需要使用cvs add –kb

选项按2进制文件方式导入(k表示扩展选项，b表示binary)，否则有可能出现文件被破坏的情况。

比如：

cvs add -kb new\_file.gif

cvs add -kb readme.doc

然后提交修改并注释

cvs ci -m "write some comments here"

* 删除文件

将某个源文件物理删除掉，比如：rm file\_name

cvs rm file\_name

然后确认修改并注释

cvs ci -m "write some comments here"

以上面前2步合并的方法为：

cvs rm -f file\_name

cvs ci -m "why delete file"

*注意：很多cvs命令都有缩写形式：commit=>ci; update=>up; checkout=>co/get; remove=>rm;*

* 添加目录

cvs add dir\_name

* 查看修改历史

cvs log file\_name

cvs history file\_name

* 查看当前文件不同版本的区别

cvs diff -r1.3 -r1.5 file\_name

查看当前文件（可能已经修改了）和库中相应文件的区别

cvs diff file\_name

* 恢复旧版本的方法：

cvs update -p -r1.2 file\_name >file\_name  
恢复到版本号1.2

* 移动文件/文件重命名

cvs里没有cvs move或cvs rename，因为这两个操作是可以由先cvs remove old\_file\_name，然后cvs add new\_file\_name实现的。

* 汇出源代码文件

cvs export

* 给代码打上tag

cvs tag prj\_rc1

**2.4.4 Subversion**

1．Subversion简介

Subversion是一个自由/开源的版本控制系统。Subversion管理文件和目录，而且会记录所有对文件和目录的改动。于是我们就可以籍此将数据回复到以前的版本，并可以查看数据更改的历史。正因为如此，许多人认为版本控制系统是一种神奇的“时间机器”。

Subversion 的版本库可以通过网络访问，从而使用户可以在不同的电脑上进行操作。从某种意义上来说，这种让用户能在各自地方修改和管理同一组数据的能力促进了团队协作。因为不再是像在一个管子里流动，开发进度会进展迅速。此外，由于所有的工作都已版本化，也就不必担心由于错误的修改而影响软件质量—如果出现不正确的修改，仅仅要做的就是撤销那次修改。

Subversion是一个通用系统，可以处理**任何**类型的文件集。对你来说，这些文件只可能是源程序—而对别人，则可能是一个货物清单或者是数字电影。

Subversion将很多新特性引入到版本控制领域。为了更好的理解Subversion的好处，我们常常拿它和CVS作比较：

* 版本化目录

CVS只跟踪单个文件的历史，但是Subversion实现的“虚拟”版本化文件系统则一直跟踪所有对目录树作的修改。在Subversion中，文件和目录都是版本化的。

* 真实的版本历史

由于只跟踪单个文件的变更，CVS无法支持如文件拷贝和改名这些常见的操作--这些操作改变了目录的内容。同样，在CVS中，目录下的文件只要名字相同就拥有相同的历史，即使这些同名文件在历史上毫无关系。而在Subversion中，可以对文件或目录进行增加、拷贝和改名操作，也解决了同名而无关的文件之间的历史联系问题。

* 原子提交

一系列相关的修改，要么全部提交到版本库，要么一个也不提交。这样可以防止出现部分修改提交到版本库中，而另一部分则没有的情况。

* 版本化的元数据

每一个文件和目录都有自己的一组属性--键和它们的值。你可以根据需要建立并存储任何键/值对。和文件本身的内容一样，属性也在版本控制之下。

* 可选的网络层

Subversion在版本库的访问的实现上具有较高的抽象程度，利于人们实现新的网络访问机制。Subversion可以作为一个扩展模块嵌入到 Apache之中。这种方式在稳定性和交互性方面有很大的优势，可以直接使用服务器的成熟技术—认证、授权和传输压缩等等。此外，Subversion自身也实现了一个轻型的，可独立运行的服务器软件。这个服务器使用了一个特定的协议，这个协议可以轻松的用SSH封装。

* 一致的数据操作

Subversion用一个二进制差异算法描述文件的变化，对于文本（可读）和二进制（不可读）文件其操作方式是一致的。这两种类型的文件压缩存储在版本库中，而差异信息则在网络上双向传递。

* 高效的分支和标签操作

在Subversion中，分支与标签操作的开销与工程的大小无关。Subversion的分支和标签操作用只是一种类似于硬链接的机制拷贝整个

工程。因而这些操作通常只会花费很少且相对固定的时间。

1）仓库(repository)

Subversion也是使用传统的集中式管理模型，它的核心是“仓库(repository)”。仓库里存放了所有的数据，所有的修改最终都被提交到仓库，所有的客户端都是从“仓库”读取数据的。如下图2.4.4.1.1所示：

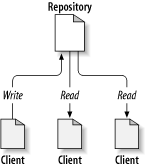


图2.4.4.1.1 一个典型的客户端/服务器端系统

2）修订版本(revision)

一次“提交”操作就是一次原子操作，它将所有对任意数量文件和目录的修改提交到版本库里。Subversion努力保持原子性以应对程序错误、系统错误、网络问题和其他用户行为， 确保一次“提交”要么所有的修改全部提交上去，要么仓库没有发生改变。每当版本仓库接受了一次提交，那么文件系统进入了一个新的状态，叫做一次修订（*revision*），每一个修订版本都被赋予一个独一无二的自然数，自然增长。版本仓库的初始修订号是0，只创建了一个空目录，没有任何内容。可以形象的把版本库看作一系列树，想象有一组修订号，从0开始，从左到右，每一个修订号有一个目录树挂在它下面，每一个树好像是一次提交后的版本库“快照”。如下图2.4.4.1.3所示：

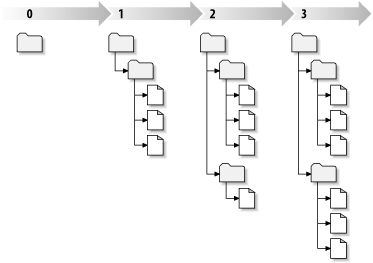


图2.4.4.1.3 版本仓库

2． Subversion快速入门

Subversion本身只是一个命令行工具集，我们常用到的Subversion的两个命令是：

1. svn -- 命令行的 Subversion 客户端程序
2. svnadmin -- 该工具用于创建、调整、以及修补 Subversion 存储库

1）安装

第一个步骤就是安装Subversion，Windows用户可以到Subversion的官方网站下载安装包: <http://subversion.tigris.org/>。如果你是使用Linux操作系统，大部分的发行版都会提供rpm包或deb包，请自行安装。下面以Ubuntu为例，安装Subversion：

|  |
| --- |
| sudo apt-get install subversion |

Windows用户还需把Subversion的命令行工具集添加到PATH环境变量里，这样就可以在命令行（cmd）下使用：

|  |
| --- |
| 1. 右击我的电脑-> 属性 -> 高级->环境变量，在系统变量里找到Path变量。 2. 在Path变量里添加：你的Subversion安装目录\bin，注意和其它路径用“；”隔开。 |

**svn**和**svnadmin**命令行有许多的选项和子命令，但是svn有个非常优秀的帮助系统，让我们不用担心不知道命令怎么用。我们可以用**svn help**，来查看svn支持的命令和选项。如果想知道的更多，譬如想知道svn mkdir怎么用，可以使用svn help mkdir，就可以获得详细的关于svn mkdir的帮助信息。下面的示例，都以在Windows环境下为准。

2）创建一个仓库

我们第一个要用到的命令就是**svnadmin**，这个命令行工具是Subversion的管理工具，常用来创建仓库、备份等等。下面我们来创建一个仓库，首先打开命令行，在Windows下，点击开始，然后运行，输入“cmd”，确定。

|  |
| --- |
| 1. cd d:\ 2. d: #改变目录到你想要建立仓库的地方。 3. svnadmin create repo #在本地创建名为repo的仓库，即仓库的位置: d:\repo。 |

这样，我们就创建了一个仓库，你的所有的文件、工程信息和版本信息都会存放在仓库里。所以仓库非常重要，你最好将仓库建立在一个安全的地方。

3）导入文件

现在我们有了仓库，下一步我们将使用**svn**工具来导入工程。首先，在仓库里创建一个文件夹：

|  |
| --- |
| 1. svn mkdir <file:///d:/repo/myproj> #替换<file:///d:/repo/myproj>为你的仓库路径和工程名。 |

Subversion会打开你的默认文本编辑器，你可以输入你的日志消息。

下一步，导入你的工程到仓库。改变你的目录到你的工程目录myproj，创建三个空文件夹trunk、tag和branch，如果你的工程目录已经有工程文件了，把他们全部剪切到trunk文件夹。创建trunk、tag和branch三个文件夹，是遵循一个约定俗成的规矩，就是trunk下放的是当前正在开发的版本，tag下放的是里程碑版本或是你自己打标签的版本，而branch下放的是分支版本，即用来测试新的想法、新的特性等等，都可以放在branch目录下，这样就可以在不影响主开方版本的情况下测试新的特性，等新特性成熟了，就可以“合并(merge)”到主开发版中。

|  |
| --- |
| 1. svn import <file:///d:/repo/myrpoj> |

这样就会把myproj下的文件和文件夹导入到仓库中。现在你的所有工程文件都在repo/myproj/trunk下了。

4）取出、修改、提交

首先，我们需要取出工程文件，在本地创建一个“工作副本”，然后就可以修改、添加等，首先改变目录到你希望创建“工作副本”的目录，下面所有的操作都是在“工作副本”目录下。

|  |
| --- |
| 1. svn checkout <file:///d:/repo/myproj/trunk> myproj |

这样就会创建一个myproj的文件夹，里面包含所有的工程文件。这个myporj是处于Subversion的管理之下的，你可以放心的将以前的没有处于Subversion管理之下的文件夹myproj彻底删除了。

如果，我想查看一个仓库里的内容，可以：

|  |
| --- |
| 1. svn list <file:///d:/repo/> #查看repo的内容 2. svn list <file:///d:/repo/myproj/trunk> #看看trunk下有什么 |

在我们对“工作副本”作了些修改，并且想保存这个时候的状态时，我们就需要把修改“提交”到仓库了。注意，没有提交到仓库里的修改，在将来的某个时候，我们是无法将它恢复的。

|  |
| --- |
| 1. svn commit #将修改提交到仓库 |

但是，如果我们在“工作副本”里添加了新的文件，并且，我们也希望它也被提交到仓库里，那么仅仅“svn commit”是不够的。

|  |
| --- |
| 1. svn add new\_item1 2. svn add new\_iterm2 3. svn commit |

５）同步

如果同时有其他开发者一起开发的化，你需要经常更新你的“工作版本”，以便把其他开发者提交的修改更新到你的工作版本中。

|  |
| --- |
| svn update #在修改前，尽可能先同步仓库，防止和其它开发者冲突  svn update –r R #更新到指定的revision |

查看文件或目录的历史：

|  |
| --- |
| svn log  svn log <file name>  svn log <directory nam> |

3．TortoiseSVN介绍

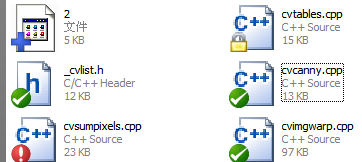
TortoiseSVN是*Subversion*版本控制系统在Windows下的一个免费开源客户端，它最大的特色是和Windows资源管理器紧密结合，让使用版本控制系统变得和操作普通文件夹一样简单。TortoiseSVNl另一个最直观的功能就是图标覆盖图，我们可以很清楚地看到各个文件的状态。如图2.4.4.3-1所示。我们可以到<http://tortoisesvn.tigris.org/>获得最新的版本。

图2.4.4.3-1 Subversion管理下各个文件的图标

安装好TortoiseSVN后，在系统的任意地方右击右键，就会出现TortoiseSVN的菜单，如果我们右击已经处于源代码管理下的文件夹，就会出现如update、commit等命令菜单，就可以直接对这个文件夹进行源代码管理。如下图2.4.4.3-2和2.4.4.3-3所示。

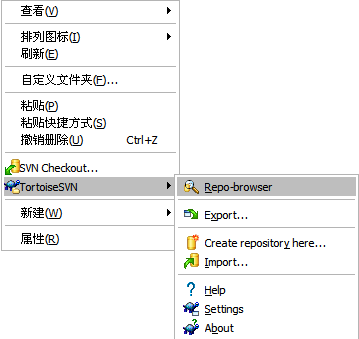
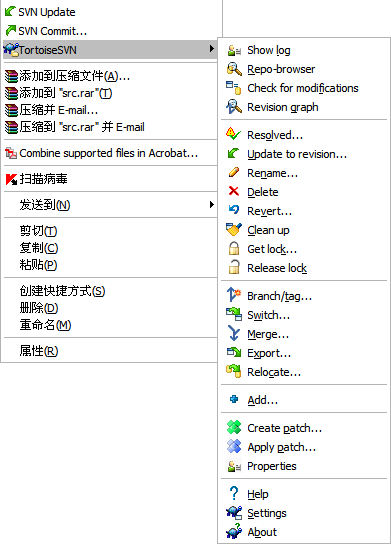


图2.4.4.3-2 TortoiseSVN菜单 图2.4.4.3-3 处于源代码管理下文件夹的菜单

1） Tortoise日常使用

（1）创建一个仓库

我们仍从创建仓库开始。首先，在你想要创建仓库的位置新建一个文件夹myrepo，**右击**这个文件夹，选择TortoiseSVN🡪Create repository here…，在弹出的对话框里选择Native filesystem(FSFS)，点击OK！仓库创建好了。不再需要敲击命令行，不需要记住命令，完完全全的Windows操作方式，通过敲击鼠标完成所有的操作。

（2）导入文件

在创建好仓库后，我们很自然的需要把原先的工程导入到仓库里。**右击**要导入的文件夹譬如prj，选择TortoiseSVN🡪Import…，在弹出的对话框中，通过浏览文件夹的方式选择仓库，还可以在Import message里写上信息，点击OK完成。注意：TortoiseSVN导入的是你右击的这个文件夹里面的内容，不包括这个文件夹本身。

（3）取出文件

从仓库里取出文件同样简单。在你想要创建工作目录的地方，**右击**空白地方，选择SVN Checkout…，在弹出的对话框里完成操作。

（4）更新文件等

在取出工作版本后，以后所有的源代码管理都可以通过右击工作目录或工作文件并点击相应的命令，非常方便。就不在一一叙述。

4． Google源代码托管服务

前面介绍了，通过TortoiseSVN，我们可以很方便对自己的文档、工程进行管理。下面，将简单介绍一下如何通过Google提供的源代码托管服务对整个项目组的工程进行管理。

在互联网流行的今天，不同地理位置的人需要协同起来对同一个项目进行开发的事非常普遍，因此，我们就需要一个能在互联网上创建仓库的地方，然后所有的开发者就可以通过互联网来访问仓库、更新仓库。Google源代码托管服务提供的正是这样一个功能。你可以通过<http://code.google.com/hosting/>来访问Google提供的这项服务。所有拥有Google帐号的用户都可以使用这项服务。

在<http://code.google.com/hosting/>页面点击[Create a new project](http://code.google.com/hosting/createProject)来创建一个项目。在项目创建好后，如果要添加其它开发者，在项目面板下选Administer🡪Project Members加入他们的Google帐号。在工程创建好后，属于你这个工程的SVN仓库也就创建好了。在项目面板下选Source，可以看到两条svn checkout语句，其中使用https协议的是给开发者用的，其它的人使用http协议。如果你安装了Subversion命令行，你可以在Windows下直接用命令行，输入Source面板里的svn checkout语句来取出工程。当然在Windows下，更方便的是使用我们前面介绍的TortoiseSVN图形客户端，在任意位置右击，选择svn Checkout…，在弹出的对话框里填上https或http网址，如果使用https网址，会要求你输入密码，直接点击Source面板上的***googlecode.com password***，就可以获得密码。这样我们就把Google仓库里的工程取出到了本地（一开始是空的），然后你在这个文件夹里进行添加、修改等操作，最后就可以把他们提交到Google仓库里。这样其它人在更新的时候就会获得你的修改。这些都和前面提到的操作没有什么不同。

## Compile

## arm-none-eabi-objcopy

|  |
| --- |
| ./arm-none-eabi-objcopy.exe --help |
| ./arm-none-eabi-objcopy.exe -I binary -O ihex C:/Users/su\_meide/Downloads/aaa.bin C:/Users/su\_meide/Downloads/aaa.hex  注：文件名不能包含空格 |
|  |
|  |
|  |

## makefile

|  |  |
| --- | --- |
| 选项 |  |
| -o | 要求输出的可执行文件名 |
| -c | 只要求编译器输出目标代码，而不必要输出可执行文件 |
| -g | 要求编译器在编译的时候提供我们以后对程序进行调试的信息 |
|  |  |
| -ansi | 只支持 ANSI 标准的 C 语法 |
| -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 |
| -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 |
| -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 |
| -static | 禁止使用共享连接 |
| -w | 不生成任何警告信息 |
| -Wall | 生成所有警告信息 |

|  |  |  |
| --- | --- | --- |
| 自动变量 | $@ | 目标文件 |
| $^ | 所有的依赖文件 |
| $< | 第一个依赖文件 |
|  |  |
|  |  |  |

## Link

## DownLoad

烧录的并不是hex文件，hex文件可以比flash大2~3倍

使用JFlash

注：根据mcu手册，选择烧录的首地址

