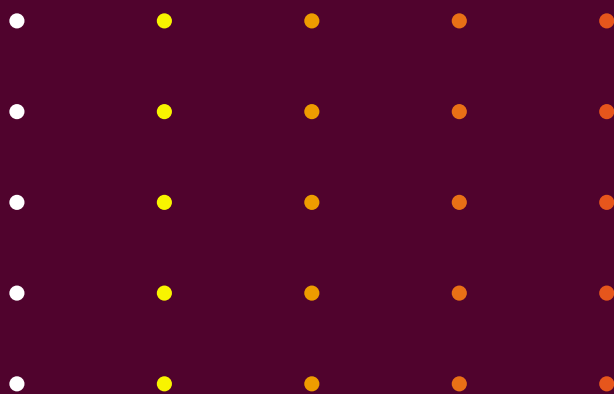


Chapter

5

第五章

程序是怎样炼成的



GCC

The GNU Compiler Collection

5.1 施工队

我发现主人最近似乎在被迫学习 Linux 下的 C 语言编程。作为一个软件，我觉得编程应该是一个很有意思的过程，因为编程就是创造软件，而且是随心所欲的，按照自己的意愿创造给自己干活的软件，这是多么有趣的一个事情阿。（就好像对你们人类来说，创造人类的过程也挺有趣的吧。）

然而主人似乎因此而觉得不是很爽，这从他的日记里可以看出来。（可不是我偷看别人日记阿，主人用 OO 老先生写的，是 OO 老先生告诉我们的。）不过不爽归不爽，主人做事还是雷厉风行的，写完这个日记的第二天，主人就开始创造自己的程序了。

要想创建一个程序也不是那么容易的，有很多的准备工作要做。首先就是需要工具，要创造些东西总要对其进行加工，所以总会需要工具的。你看木匠造个凳子还需要凿子、斧子、锯子好多工具呢是不？其次就是材料，要创造东西总是要把某种东西经过加工才变成成品的，总不可能凭空创造出来东西吧。你看木匠要造凳子得需要木头啊、钉子啊、或者胶水阿这些材料。根据爱因斯坦的物质守恒学说……

“2010 年 9 月 3 日 降雨

明天周末还要加班，真烦人。今天回家的时候路过一个工地，看到工头（大概是吧，我猜的。）拿着图纸在指挥别人干活，工人们按照指挥做着自己的事情。不禁竟然想到自己的工作，触景生情么？我不知道。我们被称为 IT 民工，不知道是不是从听到这个名词起，开始用看待同胞的眼神看待工地的民工了。我何尝不是和他们一样，每天在做着没有多少新意和变化的体力劳动。今天部门经理发彪，责怪我没把网络管理好，什么事情都手动来效率太低。看来我得学学编程，好好设几个自动化的软件来帮我干活了，要不全靠我手动管理服务器，确实太累了。然而，当年上学的时候就不喜欢编程，经验仅限于 Turbo C 下写个循环，算个 100 的累加什么的。本以为做网管不用编程，配配系统就好了，哪知道完全不是那么简单的。Linux 下有 Turbo C 么？我能学会编程么？”

哦，有点扯远了，总之，要创造程序，需要工具和材料。主人要创造一个程序，他需要的工具，就是编译器、链接器、编辑器之类的软件，需要的材料，就比如各种头文件、库文件这样的文件。创造程序之前，需要准备好这些东西才可以开始。我们的主人似乎已经进行了充分的学习，对这些理解的比较透彻，所以这天他就叫超级牛力来帮忙准备好这些工具和材料。为了方便用户们安装开发环境，超级牛力已经把创造程序需要的工具都打好了包，包名就叫做 `build-essential`。所以主人运行了 `sudo apt-get install build-essential`，超级牛力就去把这些东西从网络上拖回来并且安装好了。

要想写程序，首先得有个文本编辑器。您别听着这个名词觉得很高深，其实您早就认识并且使用过了，像是我们这里的 `gedit` 小弟，或者查皮那里的记事本，都是文本编辑器。`gedit` 小弟别看个头不大，论本事可比查皮那个记事本厉害多了。至少人家能够认识一些基本语言格式，什么 C 语言阿，脚本语言阿之类的。用 `gedit` 打开一个 C 源码文件的话，他会将程序中的一些关键词，常量，变量之类的用不同的颜色显示出来加以区分，这样看起来就比较清楚。

有人可能要问了：我在 Windows 下编过程序，也没用什么文本编辑器啊。只要装好 VC，在里面写就行了，没听说过要用记事本编程的。其实，你使用编辑器了，只不过不是独立的，而是集成在 VC 中的文本编辑器。他们 Windows 的软件就是这样，总是爱搞成个小团体，这 VC 好歹也是有点软公司的招牌软件嘛，对查皮系统中有什么基本的软件应该了如指掌，可明明知道有现成的记事本不用，非要自己带着一个编辑器来，这不是浪费资源么。这就不像我们 Linux，每个软件都顾全整个系统，从大局出发考虑问题。不过话说回来了，那个记事本的能力，也实在是没法写程序用，连个颜色都改不了，真要是写出程序来也得把人看晕了。`gedit` 小弟比记事本强点，不过其实也有限，写点小程序还行，真要是大程序，那还是有些应付不了，这时候就需要更强大的文本编辑器了。

不过有一点要注意：千万不要问谁是 Linux 下最强大的文本编辑器！



一直以来，在 Linux 这片自由的天空下，有两位公认的顶级的文本编辑器，一位是 vi，一位是 emacs。这两位谁也不服对方，都觉得自己才是空前绝后旷古烁今的全能文本编辑器（说到底还不是文本编辑器而已）。一旦有谁质疑一下这个“最强大文本编辑器”的地位，他们两个都会

第一时间跳出来，相互指摘对方的缺点，以确立自己在文本编辑器领域的不败地位。

vi 总是指责 emacs 说：“那么多的快捷键，记忆起来多麻烦。”

这时候一般 emacs 会反驳：“你呢？那么多命令难道容易记？”



“初期需要记住的命令确实多一些，但是总共就那么几个命令，记住之后就可以应用自如了。通过简单命令的组合可以实现各种复杂的操作。哪像你，每种操作都有快捷键要记忆，而且还分那么多模式。每个模式都有特定的快捷键，搞的人晕头转向。”



“你还好意思说我模式多？你自己最大的问题就是作为一个编辑器，还分什么编辑模式和命令模式。搞的新手不知如何是好，连退出都不知道怎么退出。你觉得我模式多？那是我灵活，那是我功能多。你能看邮件么？你能编写网页么？我都能，并且还远远不止这些。”

vi 会冷冷的说：“是啊……所以你才不是最强大的文本编辑器，因为你压根不是文本编辑器，你是个绑定了文本编辑功能的操作系统。”



“你连个图形界面都没有你还真好意思说我功能多？”

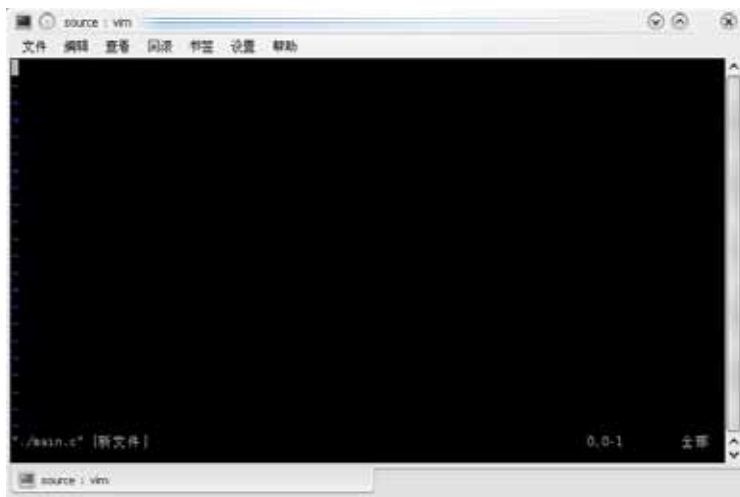


“那是我简洁，我省资源，我通过串口都能用，你行么？”

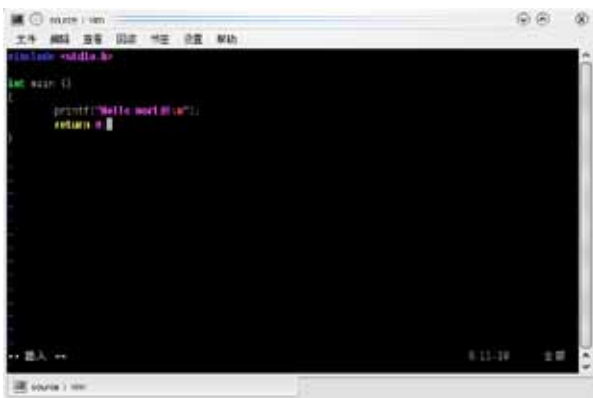
.....

哎～总之呢，一定不要让这两个遇到一起，更不能在有他们两个的时候提到谁是最好的什么什么之类的话题。否则就是：吵不关机死不休！

总的来说，这两个软件是各有千秋，用谁都行。主人选择了 vi，不过，我默认自带的这个 vi 是很古老很难用的，需要安装一个不那么变态的版本才好，他叫做 vim，怎么装就不用我多说了，主人很清楚：
`sudo apt-get install vim`。几分钟之后就装好了，主人首先是创建了一个叫做 test 的目录，然后 cd 进去，运行：`vim main.c`



进入 vim 的主界面之后，默认是在命令模式，这时候不能够输入任何东西，要按下 insert 键，或者 i 键进入编辑模式。进入之后在下边就可以看见“插入”的字样，这时候就可以输入文字内容了。



“: ”表示要输入命令，后面是具体的命令或命令的组合。w 就是保存的意思，主人输入了“:w”后按下回车，这个 main.c 文件就保存好了，

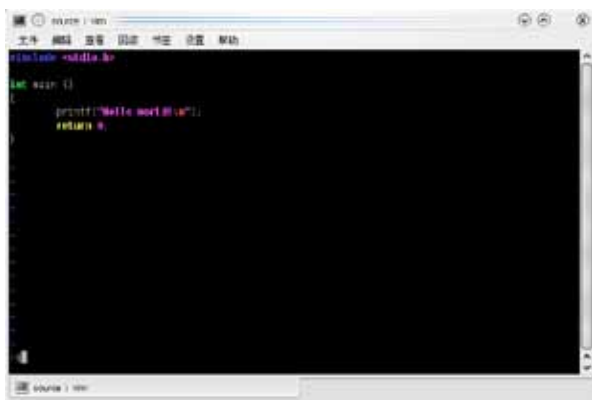
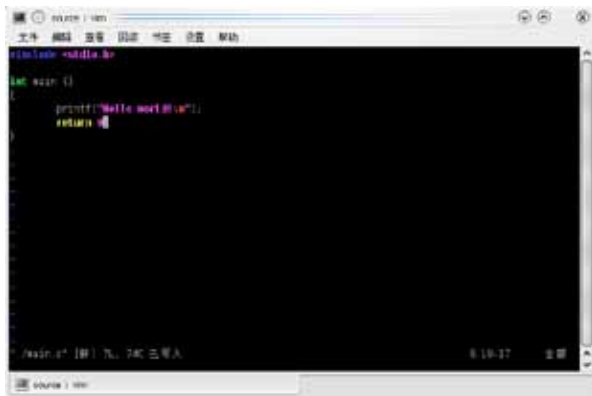
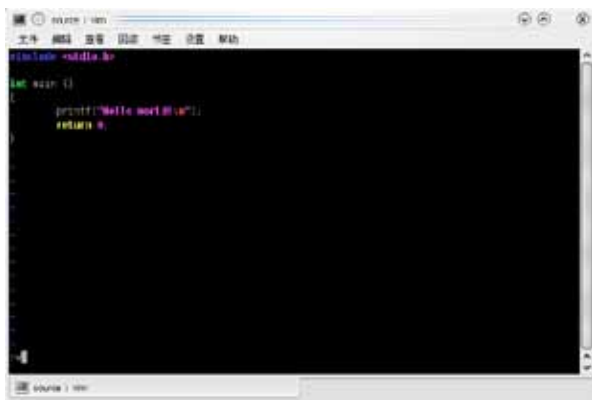
之后主人又输入“:q”，这是要退出的意思，其实也可以直接输入“:wq”，就是保存后退出，这大概就是 vim 所说的命令的灵活组合了吧。

到目前为止，一个简单的 C 程序的源码已经完成了，但是这只是第一步，离真正运行起来还远呢。一个 C 的源程序就好像是盖房子时候的图纸一样，有了图纸就可以了解整个建筑的结构，怎么建造，什么材料之类的。但是光有图纸是没用的，要想把图纸变成房子，需要施工，这时候就需要一个能够把图纸变成房子的施工队。我们这个系统里也有这么一个能够把源码变成程序的团体，他们就是以 gcc 为首的“施工队”。

施工队主要成员有 gcc、cpp、as、d 四个人，其中 gcc 是老大，其他几个干什么活都得听他调遣。主人一般也只跟 gcc 打交道，当写好了图纸之后（也就是源代码啦，就比如刚才主人写的 main.c），就

主人进入编辑模式，弹指如飞，瞬间，一个经典的程序写成了，就这样：

主人写了这一小段程序之后，想要保存，于是按了下 esc 键，这是要从编辑模式回到命令模式，因为只有在命令模式下才可以作保存、打开、退出这类的动作。进入命令模式后，主人输入了“:w”，这时候输入的这两个字符可不会在上面出现了，而是出现在左下角。



直接把图纸交给 gcc 去处理就好了，gcc 会去调动其他人进行各种处理。

一般来说，gcc 拿到图纸后，会首先叫来 cpp 进行预处理。预处理主要就是将文件里的宏定义进行展开。什么是宏定义呢？主人一般都比较懒，或者说，他们人类能力有限，不愿意写好多重复的，类似的东西，就把这些都定义成宏。比如，这么写：

```
#define TOTAL_NUMBER 18353226
```

就是定义总数为一千八百三十五万三千二百二十六，那么以后再要用这个总数的时候，就直接写 TOTAL_NUMBER 就好了，不用写那一大串数字。而且，如果总数变了，只要在最初 #define 的位置修改一次就可以，反正就是为了偷懒。那么 cpp 的任务就是把这类的宏定义都替换回去，把所有的 TOTAL_NUMBER 都替换成 18353226，否则他们老大 gcc 看不懂，老大看不懂，那就没法继续往下干了，因为经过 cpp 预处理之后的文件就要交给 gcc 去编译了。

编译又是怎么个意思呢？最初的图纸，也就是没有经过预处理的源代码，是人写的，一般懂相关语言（比如 C 语言）的人都能看懂。预处理之后的文件，虽然不那么直观了（TOTAL_NUMBER 看着是不是比 18353226 直观？光写个 18353226 还以为是谁的 QQ 号呢），但终究只是做了下替换，还是人类可以看懂的。这样的代码经过 gcc 的编译之后，就不是普通人类可以看懂的源代码了，而是只有终极牛人才能读懂的汇编代码。汇编代码就比较贴近底层的机器码了，里面描述的都是些基本的操作。打个比方吧，就比如描述切菜的过程，用 c 语言描述出来就像是“将黄瓜切片”，这么一句就搞定了。要是用汇编，那就是：

左手扶住黄瓜，右手拿起刀，移动刀到黄瓜顶部，刀落下，刀抬起。刀向黄瓜后部移动 4 毫米，刀落下，刀抬起。刀再向黄瓜后部移动 4 毫米，刀再落下，刀再抬起。放下刀，走出厨房，走进卧室，找到创可贴，贴在左手食指上……………

总之，汇编是一种面向机器的，很复杂的程序设计语言。gcc 的任务就是把 c 语言的源代码转换成贴近机器语言的汇编代码，为下一步 as 的工作做好准备。

as 拿到汇编代码后，对这样的代码再进行处理，得到真正的机器码，这个过程，也叫汇编。进行汇编之前的汇编代码是终极牛人更牛的人才能看懂的，那么汇编之后得到的机器码压根就不是人才能看懂的。汇编程序中至少还有些操作的助记符，比如什么 add 啊，mov 啊之类的。寄存器也是有名字的，比如叫 A，叫 R1 之类的。但是到了机器码，这些都没有了，这些都换成了各种各样的数字，半句人话都没了。还说切黄瓜的事，要是用机器码来描述，那就相当于说：

用 32 号设备扶住 87 号物体，24 号设备拿起 126 号物体，移动 126 号物体到 87 号物体顶部，做 2635 号动作，再做 2636 号动作……

好了，现在终于得到机器码了，机器码按说就是可以执行的代码了，但是，这时候的程序还是不能直接执行的，为什么？因为还有 ld 没有出场呢，他的工作叫：连接。光是一段机器码扔给机器去执行，机器照样摸不着头脑。而且，很多时候，一个程序不是一段机器码，而是由很多段机器码组成的，这些机器码分别存成很多的 .o 文件，这时候就需要 ld 出场了。ld 负责把这些机器码组装起来，并且写明了各段代码的地址，从哪里开始执行之类的。就像我们造个机器人，脑袋啦，胳膊啦，大腿啦之类的都做好了，ld 就是负责组装的。就算只有一段机器码，也就是只有一个 .o 文件，也要由 ld 进行一下处理，闹明白哪是头哪是尾，才能开始运行。

别看说的这么复杂，其实这些过程都会由 gcc 全权负责，主人并不需要操心，主人只需要跟 gcc 交流就好了，交流的方法也简单，就是运行一下 `gcc./main.c` 就可以了。



运行之后，以 gcc 为首的施工队赶紧起床，四个人拿起主人设计的图纸看了看，相互点点头，立即开工，流水线作业：cpp 先把图纸拿过来预处理，之后扔给 gcc 去编译，gcc 再递给 as 进行汇编，最后 as 交给 ld 进行连接，全部完成后，一个崭新的程序出现在了硬盘里。名字叫做 a.out——因为主人没有给他起名，gcc 就默认叫他 a.out。我们内存里的软件们都很好奇的凑过去看着这个刚刚出生的软件，只见他只有不到 10k

大小，看来主人的图纸里基本没有什么内容，也不知道这个家伙会干什么。我们正在胡思乱想呢，主人马上下达了命令 `./a.out` 这就是说要叫醒当前目录下的 a.out，也就是刚刚 gcc 施工队创造出来的这个程序了。

我慢慢的走过去，捅捅还冒着热气的 a.out（刚出锅嘛，可不冒热气，呵呵），温柔的对他说：这位新同学，起床干活啦。只见他一听我叫他，立刻飞身跳进内存，跑进内存后大喊一声：“I am a Rubbish!” 然后，跑回去继续睡觉了。-_-b 我说主人呐，您设计的这程序还真是挺谦虚哈。

后来我们都管这个程序叫做 Rubbish1 号。虽然他能干的事情不多，不过主人还是很满意，毕竟是自己第一次成功的创造了一个程序嘛。很快主人又拿来 Rubbish1 号的图纸改起来。10 分钟后，又把图纸交给施工队，施工队的哥儿几个凑在一起拿过图纸

来看看，点点头，立马开工，流水线作业，cpp 预处理→gcc 编译→as 汇编→ld 连接，之后，Rubbish2 号诞生！毫无悬念的，主人马上让我叫醒 Rubbish2 号起来干活，于是我走过去叫醒他，只见 Rubbish2 号立刻飞身跳进内存，跑进内存后大喊一声：“I am a Ru~Ru~Ru~Ru~Rubbish~~~~!”然后，跑回去继续睡觉。主人成功的用 for 循环创造了一个结巴。15 分钟后，Rubbish3 号的图纸再次毫无悬念的完成，图纸交给施工队，施工队哥儿几个凑在一起拿图纸看了看，点点头，开工，流水线作业，cpp 预处理→gcc 编译→as 汇编→ld 连接……很快，Rubbish3 号诞生，然后我去叫醒他，然后他立刻飞身跳进内存，对 metacity（Gnome 的窗口管理器）说：“我要一个窗口”。metacity 赶紧给他画好一个，然后他对着窗口喊“I am a Ru~Ru~Ru~Ru~Rubbish~~~~!”，然后毫无悬念的又回去睡觉了。图形界面的结巴……

当当当~~，Rubbish4 号诞生，这回没让 metacity 画窗口，而是在终端打印出了一句话：“Please Input a Number:”，然后就等着主人输入。主人输入了两个数：5 3，然后 4 号就大声喊：“I~I~I~I~I~ am a Ru~Ru~Rubbish~~~~!”——程控结巴！

5 号闪亮登场拉~他进来之后，紧闭双目，念动咒语“俺木哒咪咪呀~~分！”然后只见白光一闪，边成了两个！两个 5 号同时喊：

“I am a Ru~Ru~Ru~Ru~Rubbish~~~~!”

“I am a Ru~Ru~Ru~Ru~Rubbish~~~~!”

二重结巴！



5.2 修理工

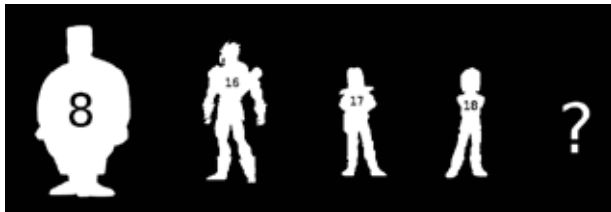


“2010 年 9 月 18 日 变天

终于又到周末了。今天在自己的机器上学习编程，逐渐的开始入门了。这一阵子最大的收获是学会了自学。通过网络，论坛，搜索，也能学到不少有用的东西。像学编程吧，开发环境的搭建，循环结构，文字输入，甚至创建进程，都尝试过了。看来编程也不是那么的困难嘛。不过我也知道这只是刚刚入门而已，真的写出个能用的程序和随便写个小程序玩玩还是有很大区别的。

今天我家的楼停电了，不知道哪里的问题，几家邻居慌手乱脚的也整不出个头绪来，最后物业的修理工来了，三下五除二就找到问题所在，没 5 分钟就搞定了，专业就是专业啊。 “

今天起床的时间似乎比平时晚了点，全体起床之后，主人按部就班的叫来 OO 老先生记录下一些文字，之后又继续去创造 rubbish 去了。这一段时间里，我们的主人真是笔耕不辍，哦，不对，应该是键盘不辍才对，先后制造了 18 个 rubbish 程序。不过 13 号以下的基本都被主人删除了，只留有一个 8 号。因为 8 号比较憨厚，性格温顺，不爱打架；然后印象比较深的就是 16 号。16 号很爱静，不爱多说话，有点冷漠，但是做起事情来一丝不苟，严格的执行命令；17 号很厉害，但是很自大，高傲，总跟别人发生矛盾。他好像和 18 号还有点什么关系，具体的我就知道了；最后的 18 号是个美眉，长的很可爱，一头金发，本事也不错，我们大家看他都很顺眼。现在主人正在制造 19 号，不知道为什么，我总觉得即将到来的 19 号将是一个邪恶的坏家伙……



没过多长时间，19 号出炉了。只见他起床之后，跑进内存，刚说了几句我是 rubbish19 号之类的话，就开始乱动别人东西，一会要去狐狸的内存空间里拿数据，一会又要往心有灵犀的地盘里存东西。当然，他的这些企图都没有得逞，要是连这样的小流氓的管不了，我还叫内核么。我们这工作间里面的空间管理是很严格的，谁的空间就是谁用，别人不能乱动。向 19 号这么目无法纪，影响他人工作的软件是不能容忍的。眼看着这邪恶的 19 号，和满工作间无辜受害的软件们，我终于忍无可忍，为了工作间的安宁，为了我稳定内核的荣誉，为了爱与正义，为了部落，为了世界和平，我代表月亮，我，我祭起屠龙刀，看准 19 号，手起刀落，只听的咔嚓一声——整个世界安静了，19 号被我斩为两段，然后我向主人汇报：您的程序出现了段错误。（因为他被砍成两段了，所以错误了）



主人似乎有些不明所以，不知道这个段错误是怎么回事，（因为太血腥了，所以我没直说是因为被我剁成两段。）于是就赶紧叫来狐狸上网查去。通过搜索知道了，段错误的情况有很多（很多种不老实的程序都会被我砍成两段），但大致上都是由于内存指针使用不当引起的，比如没有给指针赋值就去使用，或者虽然赋值但是访问越界等等。总之就是动了你不该动的内存就会段错误。可是到底这个 19 号是如何动了别人的空间呢？到底他为什么要去访问非法的地址呢？这些情况虽然我们内存里的软件们看得一清二楚，铁证如山，但是主人不知道，他没发钻进内存里来看。那么主人能有什么办法看清楚 19 号的一举一动呢？这时候就需要我们的软件修理工 gdb 闪亮登场～



gdb 是一个字符界面的调试工具，用过 vc 的应该知道在那里调程序的时候可以进入 debug 模式，单步执行之类的。我们 Linux 中，每个软件秉承着“只做一件事情，但做到最好”的原则，将调试这件事情交给了 gdb 来完成。gcc 编译出来的程序，可以通过 gdb 来运行，运行的时候，就可以设置断点，单步运行，查看变量，查看堆栈等等操作。有了 gdb，主人就可以监视程序在内存里面的一举一动了。不过 gdb 并不是像狗仔队那样想监视谁监视谁，像狐狸啦，gedit 这样的成品程序是不能被监视的，要想让某个程序被 gdb 监视，必须在制造他的时候——也就是编译的时候，留出给 gdb 控制的接口来，gdb 才能监视那个程序的一举一动。看过黑客帝国么？我们机器里的普通的程序就像是里面的正常的自然人，而可以被 gdb 调试的程序就像 Matrix 世界中的人一样，脑袋后面有个接口，可以接进去控制。那么怎么给程序装这么个接口呢？很简单，就是在编译的时候加上参数 -g。

```
Gcc -g ./main.c -o rubbish9_debug
```

主人运行了这么一句，创造出了脑袋后面有接口的 rubbish 19 号。之后就叫来 gdb 去运行他，就这样 `gdb ./rubbish9_debug`

```
GNU gdb (GDB) 7.0-ubuntu
Copyright (C) 2009 Free Software Foundation, Inc.
License GPLv3+: GNU GPL version 3 or later <http://gnu.org/licenses/gpl.html>
This is free software: you are free to change and redistribute it.
There is NO WARRANTY, to the extent permitted by law.  Type "show copying"
and "show warranty" for details.
This GDB was configured as "i486-linux-gnu".
For bug reporting instructions, please see:
<http://www.gnu.org/software/gdb/bugs/>...
Reading symbols from /home/saub/test/rubbish19_debug...done.
(gdb)
```

`gdb` 接到命令，赶快掏出各种仪器和工具，并把 19 号拖进内存里，然后从一个大机器上抽出一个长长的电缆，插进 19 号脑袋后面的接口里，一切准备就绪之后，向主人报告：一切准备就绪，可以开始了。

主人好像是之前看书学习过了，虽然就我所知这是他第一次使用 `gdb`，但是似乎还挺熟练，看到 `gdb` 的提示后，输入了 `r`，并且回车，意思就是：开始运行。（`run` 的首字母）于是，`gdb` 按下一个电钮，19 号的身体跟着腾的一下站了起来，启动了。插着电缆的 19 号像他上次启动一样，还是要去骚扰狐狸妹妹，访问人家的内存空间，于是我也只好举起屠龙刀，再次将其一刀两断。`gdb` 赶快向主人报告：19 号同学在按照您的图纸进行 `xxxx` 个动作时候的，由于侵占他人内存空间，触犯了内存治安法第 287 条，因此被处以断刑。然后还指出了 19 号的这种

```
Starting program: /home/saub/test/rubbish19_debug

Program received signal SIGSEGV, Segmentation fault.
0x080483f4 in main () at ./main.c:6
18*msg='R';
(gdb)
```

行为在主人的图纸中所在的位置，也就是代码行数了。

主人一看，18 行就出问题了，很打击自尊阿，可是这 18 行也看不出什么不对的来。应该是这个指针有什么问题，还是从头看看程序吧。于是主人输入了 `list` 命令，`gdb` 赶紧把 19 号的整个图纸——也就是全部程序的源代码打印了出来。程序一打印出来，我们就都看明白了，这个 `msg` 指针压根就没有初始化嘛，只是做了声明而已，也没给申请内存空间，就这么用，那不出问题才怪。

虽然我们都看出来了，但是主人似乎还是没明白过来，于是还不断的用 `b <linenum>`，这样的来设置断点（`break` 的首字母），然后用 `n` 命令来一步一步的运行（`next` 的首字母）。直到他用 `p` 命令（`print` 的首字母）来打印 `msg` 的值的时候才终于意识到：哦，好像没给分配空间哈，`msg` 的值一直是 `NULL` 呢。

故事外的事—— 内存管理

这一回跟您介绍了主人创造的 rubbish 19 号程序，这家伙由于不遵守《Linux 系统内存管理条例》被我干掉了。那么我们 Linux 的内存管理是怎么一个原则呢？都有什么规矩呢？下面我就跟您说说。

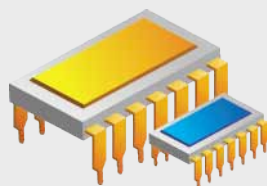
一个程序工作的时候需要用到的内存分为几个部分：代码段，数据段，BSS 段和堆栈段。其他的基本上程序一启动就确定好了的，没什么可说，也没太多要管理的，唯独这个堆栈里面的堆空间，是需要管理的。

堆空间是指程序在起床运行后向我申请来的空间，也是一个程序占用的最多的空间。一个程序如果要想使用工作间里的空间，他要向我提出要求，说我需要多大多大的一块内存空间——这个过程叫做申请。然后我根据工作间里的情况来分配，告诉他，哪块哪块归你，然后这个程序就去用去了。这时候那块地方就单独给这个程序使用，不许别的程序访问了。如果别的程序胆敢来访问这块空间，就像你去人家偷东西一样，必须依法剁成两段。（偷东西没这么大罪过吧.....）

那么这块内存空间分给这个程序使用之后就永远给他了么？想的美！你买房还只有 70 年产权呢，这么珍贵的内存空间怎么可能永久分给一个程序。申请到内存空间的这个程序在做完了相关的事情，不再需要这块空间的时候，他应该跟我报道，说空间我用完了，这块地方可以再给别的程序用了——这个过程叫做释放。

一个有知识有道德有理想的程序，在他回硬盘睡觉以前应该释放掉所有他申请过的空间的。如果遇到哪个不良程序，无德青年申请了很多空间不释放，我也没有办法，因为这不像是越界访问那样，访问了别人的地址那就是访问了，人赃俱获，无法抵赖。人家不释放也许是因为他一会还需要用这块空间呢，我没法证明他申请的空间用不着了。我唯一可以做的就是在他回硬盘睡觉的时候，检查所有他申请过的空间，如果有没释放的，就强制释放掉——你都睡觉去了，你申请的空间肯定用不着了嘛。但是如果这个程序是个长时间运行的程序，并且还不断的申请新的空间而不是放，那就麻烦了。内存空间会被他一点一点的消耗光，这就是最烦人的内存泄漏，在我们软件界，内存泄漏是和瓦斯泄漏同样严重的事故。《Linux 系统内存管理条例》第三条明确写着——禁止申请不释放！就在第四条禁止抽烟的上面。（当然不能抽烟，内存都冒烟了机器还能用么？）

其实不光是 Linux，查皮那里的程序一样需要遵守类似的原则，估计他们那里大概也有个什么《Windows 系统内存管理办法》之类的文件吧，反正大家的原理是一样的。



不过对于工作间的使用，查皮和我还是有点不同的。查皮总是喜欢尽量留出空间来，好给新起床的程序用。可是我总觉得，作为一个系统，我怎么能知道主人还会有什么程序要运行呢？要是没有程序要来了，工作间里还空那么大地方，不让正在工作的程序用，那不是浪费么？

我还是习惯尽可能的把东西都搬进工作间里。除了程序们申请多少内存就尽可能给多少之外，剩下的部分，我就把一些可能会用到的库啊，命令啊啥的统统都搬进来，能占多少占多少。那有人问，要是你把这里边都沾满了，待会有程序要进来咋办？很简单啊，我再搬出去呗！程序要进来，也不是一下子都进来，他也得把他的东西一点点搬进来，他往内存里搬的时候，我就往外搬，不耽误。

所以，当有程序要启动，跟我说：我要 10 平米的地方放东西。我就先答应他说，好，放吧，有地。然后在他往里搬的时候我再给他腾地方。也可能他要 10 平，但是先只用了 2 平，那我就先腾出 2 平来，等他再要我再腾。他们管我这个方法叫 Copy-on-write。

查皮就不同了，可能是因为他比较胖的缘故吧，他比较懒，不愿意搬来搬去这么折腾。基本上他只是在必须用啥东西的时候才把那东西搬到内存里，让内存留出尽可能多的空间。这样，当有程序管他申请内存的时候，他就可以用手一指：那块地，归你。然后就不用管了。实在内存不够用的时候就找个比较闲的程序，命令他：你，去硬盘里先忍会。

5.3 包工头

“2010 年 10 月 10 日 起风

今天好孤独，节后的只有一天的周末还要独自来加班。领导总是不顾员工的死活，就像剥削劳动人民的包工头一样。他让我无论如何要解决全公司计算机的杀毒问题，不管怎么解决，反正必须解决。这叫什么事儿嘛，也不调查一下实际情况，全公司那么多台电脑，我怎么给他们挨个杀毒？保证服务器不受病毒干扰那是我的责任，没的说，可我怎么才能保证每一个员工的电脑不中毒呢？谁知道他们每天都拿电脑装些什么乱七八糟的东西！真是气死我了。”

最近硬盘里的 rubbish 越来越多，已经排到 31 号了。虽然这些程序都不大，31 个加在一起也就几 M 的大小，可关键是这帮不着调的程序要是运行起来就乱了套了，指不定哪个就给你捅娄子玩。你像 19 号这个段错误倒是还好说，我能看得见，并且能第一时间让主人知道，主人也就想办法修改了。可是比如 23 号，有内存泄露，这个我没发第一时间通知主人，也加上我们这 4G 内存足够大，泄露了主人也感觉不出来，所以一直也没有修正这个错误。还有那 24 号，总是乱动别人的文件，那次就把狐狸妹妹的文件给弄乱了。人家狐狸妹妹工作的时候有许多东西要记录的，比如网页用什么字体显示啦，主人喜欢去那些网站啦之类的东西，狐狸妹妹都会写成配置文件存放在自己的那个目录里。结果那天那个 rubbish24 号一进工作间就上窜下跳，整的大家都不得安生，来来去去的要创建文件、修改文件、删除文件。你说你自

己折腾着玩谁也不管你，可是等他折腾完了会去睡觉了才发现，狐狸的配置文件让她给弄没了。闹得狐狸跟失忆了似的，主人再打开她的时候，模样也变了——因为不记得主人喜欢什么样子的了。主人说：去我最常去的那个网站。她眨眨眼问：哪啊？把主人气得鄙视了狐狸半天。我们真替狐狸冤枉阿。还有那个霸占着 CPU 不放的 27 号；在屏幕上乱写乱画的 17 号等等，罄竹难书阿～。结果今天 vim 传来一个更悲惨的消息：主人觉得这堆 rubbish 程序都没什么实用性，要把 31 个 rubbish 合并起来，创建一个能陪主人消磨时光的，有智力的 rubbish 合集！！我们一听头就大了，这主人是不是变形金刚看多了阿，31 个不着调的疯子程序合体能合出什么好玩意来？赶紧问 vim 消息可靠么？vim 说：可靠，主人打开了最初的那个 main.c 文件，正在把其他的 rubbish 的代码往里面拷贝，我们一听，完了……

只见主人首先拿来 rubbish1 号和 2 号的图纸，由于 1 号实在没什么本事，而且跟 2 号的动作差不多，于是就把这张图纸改成了整个合体后的 rubbish 的总体结构，（咱就把合体后的 rubbish 叫做 Rubbish 金刚吧）然后将 2 号的程序作为 rubbish 金刚的一部分加入到 rubbish 金刚的代码中。然后主人尝试编译 `gcc main.c -o all_in_one`，然后运行 `all_in_one`，效果不错。



之后主人继续让 rubbish 合体，要将 3 号的代码加进来。可是主人自己一想，这么多代码，要是都写在这一个 Main.c 里面多乱阿，对了，我把他们按照功能分别写成独立的 .c 文件吧。于是主人把代码结构整理了一下，设计了这么几个文件 main.c, board.c, board.h, ai.c, ai.h。

看这名字还真猜不出来主人要创造的这位 rubbish 金刚是干啥的。但估计不是什么好鸟，我们还是小心为妙。这样把结构调整了之后，主人再编译就要：`gcc main.c board.c ai.c -o all_in_one`，然后运行 `all_in_one`，恩，程序正常——因为 board.c 和 ai.c 里面还没什么东西呢，main.c 里面也不过是点打印语句当然正常。

再继续，主人继续把之前的一些 rubbish 的图纸往 board.c 和 ai.c 里面复制，复制了一点之后进行适当的修改，然后就编译试试。每次主人一编译，施工队那哥几个都得跑过来，把三张图纸都打开，拼在一起，然后开始从头施工。有时候主人只是在 ai.c 里面修改了很小的一点东西，编译的时候施工队的同志们也要从到尾的重新编译每一个部分。这就好像盖个楼，完工之后开发商说：一楼这个大门的门把手图纸上画错了，应该用圆的，怎么画成方的了？改了吧。施工方一听，赶紧下令：图纸画错啦！把楼炸了重新盖！虽然这样对于拉动 GDP 发展有很好的作用，但毕竟属于精神不正常的范畴。那么我们的 gcc 施工队为什么做这种很抽风的事呢？这不怪施工队，这是因为主人这么写的命令：

```
gcc main.c board.c ai.c -o all_in_one
```

就是要这么干，施工队也只是服从命令而已。所幸我们的主人觉悟还是比较高，神经还

算基本正常，很快就意识到了这么做可能有点浪费劳动力。三个文件里，改了哪个，就只编译哪个可以么？当然可以。这回主人这样编译：

```
gcc -c main.c board.c ai.c
```

这是告诉施工队，只把这些图纸分别编译成模块，并不组装起来。施工队按照指令施工完毕之后，目录里没有什么 `all_in_one` 程序，而是出现了 `main.o` `board.o` `ai.o` 三个文件。之后再 `gcc main.o board.o ai.o -o all_in_one`，才生成最终的程序。这样如果主人修改了 `ai.c`，只要运行 `gcc -c ai.c`，重新生成 `ai.o` 就可以了，`board.o` 和 `main.o` 就不用动了。这样施工队倒是省了不少事，可是主人又不平衡了：以前我只用一条命令 `gcc main.c board.c ai.c -o all_in_one` 就可以了，这下我每次修改之后都得 `gcc -c xxx.c`，然后 `gcc xxx.o xxx.o xxx.o -o yyyy`。这不是反倒麻烦了么？

哎，看来主人还真不是个搞编程的，要不怎么连这么点问题都不明白呢？这样编译虽然看上去多打了一次命令，但是节约的编译时间阿。现在主人的程序小，体会不到，要是编译个内核呢？明明只改了一点点，却要编译整个内核，浪费将近半个小时？再说，其实这些命令完全可以不让主人每次都敲的。那 `gcc` 施工队他们毕竟只是个施工队，你要是盖个小厨房，垒个猪圈啥的，这样的小东西直接找他们就没问题。直接一编译：`gcc 砖头 -o 猪圈`，这就出来了。可现在你要搞一个无敌全能不着调的 `rubbish` 合体，这可就不是猪圈了，这就是个 CBD 商圈，里边什么银行啊，商场啊，写字楼啊，炸油条的，卖臭豆腐，修理自行车的等等，一应俱全。这么大的一个工程，你光叫个施工队来那肯定搞不定阿。这得有人进行合理的统筹规划，设计施工方案，然后再让施工队去具体施工。而这个规划的人谁呢？按照主人现在的做法，这个规划人就是主人自己，但他自己又没这本事，怎么办呢？这时候他就需要专业的规划人——`make`。

`make` 也是一个程序，像上边说的一样，他就是负责控制整个施工过程的（也就是编译过程啊）。对于比较小的程序，就一两个 `.c` 文件，那根本用不着 `make` 出马，直接 `gcc` 施工队去编译就行了，因为源文件的结构关系不是很复杂。可是要稍大一点的程序，像狐狸妹妹啊，心有灵犀啊，星爷啊，基本上所有常用的软件吧，都足够复杂到需要 `make` 来对编译过程进行管理。当软件大了，编译的时候就不能像主人刚才似的简单的把一大堆 `.c` 的源文件统统一次性编译成一个二进制文件那么简单了。那样做的话费时费力，主人现在虽然还没有太体会到，但是以后要是程序再大点，他要想修改一下，可就费劲了。

比如说吧，有一个软件，源码由 20 个 `.c` 文件组成，分别是 `1.c`、`2.c`、`3.c`……`20.c`。这 20 个文件一股脑都交由 `gcc` 包工队，他们就会把这些文件都打开来，拼在一起，一次性的编译成一个叫做 `big` 的二进制文件。这时候发现了一些问题，需要修改 `3.c` 文件，修改之后得重新编译啊，那么 `gcc` 包工队又得把这 20 个文件全都打开，拼在一起，再从头到尾编译一次。而其实只有 `3.c` 文件修改了，完全不必这么兴师动众。那应该怎么做呢？一般的都是把这 20 个文件分别编译成 `.o` 文件，比如编译成 `1.o`、`2.o`、`3.o`……`20.o`，这样 20 个 `.o` 文件，然后再由 `ld` 把这些 `.o` 文件拼在一起，成为一个叫做 `big` 的二进制可执行文件。那么当要修改 `3.c` 的时候，只需要让 `gcc` 包工队重新将 `3.c` 编译为 `3.o`，再让 `ld` 重新连接一遍就好了，省去了很多时间。

而这个过程，如果让主人自己管理的话，会很麻烦，毕竟他们人类的大脑也不是那么靠谱的，搞着搞着就乱了。于是，`make` 义无反顾的挑起了这个重要的担子。当然 `make` 也不能靠凭空的想象就来指导包工队干活，什么事情总得有个规划不是。`make` 也需要一份施工的规划书，这分规划书就是 `Makefile`。

`Makefile`，顾名思义，就是 `make` 用的 `file`。这就相当于一份施工的规划，上面写着整个工程分为几个模块，先用哪几个文件编译成一个什么什么 `.o`，再用哪几个文件编译出一个 `.o`，再怎么怎么一连接，最后得到编译好的二进制程序。`make` 就根据这份文件来指导 `gcc` 他们进行施工。当有某个 `.c` 文件被修改之后，`make` 能够根据文件的修改时间智能的判断出哪些模块需要重新编译，重新连接，然后就去让 `gcc` 重新编译那些改过的文件，最终生成新的二进制程序。有了 `make` 和 `Makefile`，就省去了主人敲一大堆编译命令的烦恼，只要敲一个 `make`，其他的，就交给 `make` 去做吧，他办事，你放心。

`vim` 很快又来报告：主人新建了一个叫做 `Makefile` 的文件。

恩～看来主人终于想起 `make` 这个软件了。我刚要欣慰一下，忽然一想，不对！主人越是用合适的软件，那个 `rubish` 金刚就越早的来工作间捣乱阿！哎呀，怎么办啊？怎么办啊？可是咱们也不能拦着主人干活不是？只见主人用 `vim` 打开并创建了 `Makefile` 文件，然后在里面写道：

```
all:main.o board.o ai.o
gcc main.o board.o ai.o -o all_in_one

main.o: main.c
gcc -c main.c

board.o: board.c board.h
gcc -c board.c

ai.o: ai.c ai.h
gcc -c ai.c

clean:
rm ./*.o
rm all_in_one
```

这就是专门给 `make` 看的 `Makefile` 的内容，也就是相当于施工流程说明。这里面写了，最终的目标，是编译出一个叫做 `all_in_one` 的程序。要创造这个程序，就需要 `main.o`, `board.o`, `ai.o` 这三部分内容。有了这三个文件后如何创造呢？就是运行：`gcc main.o board.o ai.o -o all_in_one`。这也就是最上面，`all` 那一段写的意思。下面又分别写了如何建造出 `main.o`, `board.o`, 和 `ai.o` 这三个文件。比如 `main.o` 这段，意思就是，要想得到 `main.o`，那么需要 `main.c` 这个文件，有了这个文件如何得到 `main.o` 呢，就是运行 `gcc -c main.c` 这个命令，下面的 `board.o`, `ai.o` 也是类似。简单的说，这里的格式就是：

目标：原料

加工方法

但要注意，加工方法那一行必须以 `tab` 符号开头，而不能是空格（虽然都是看不见的一片白，但是决不能搞错），不知道 `make` 这家伙为什么有这样的怪癖。

有了 `Makefile` 之后，主人修改程序之后，编译就变得非常方便了。比如主人改了 `ai.c` 文件，存好之后，直接运行 `make`，就可以了。运行之后 `make` 会跑进工作间，拿起当前目录下的 `Makefile` 来看这个项目各个文件之间的关系。之后再去检查 `Makefile` 里面提到的这些文件的最后修改时间，一看就能发现，那个最终的产品，`all_in_one` 程序的最后修改时间是 3 点 15 分。`ai.c` 的时间是 3 点 20 分，其他的文件最后修改都是在 3 点 5 分左右，反正在 3 点 15 分之前，于是 `make` 马上通过严密的推理和判断，知道了——主人在 3 点 5 分左右写完了上一版代码，然后在 3 点 15 分编译出了最终的程序，之后在 3 点 20 分又修改了 `ai.c` 文件，先在想要重新编译一下。那么本着节约用电，不搞重复建设的原则，我应该只编译 `ai.c`，把他编译成 `ai.o`，之后在把新的 `ai.o` 和刚才已经有了的 `main.o` 和 `board.o` 组合成一个新的 `all_in_one` 程序。于是，`make` 就去叫来 `gcc` 施工队，下达命令：`gcc -c ai.c` 然后 `gcc main.o board.o ai.o -o all_in_one`。

当然，咱们是说的复杂，其实 `make` 想这些事情只是电光石火的一瞬间而已，一个 `make` 命令敲下去，马上就看到施工队们出来干活了，一个崭新的 `rubbish` 金刚就诞生了。

诞生之后，主人赶紧运行一下 `./all_in_one`，果然运行起来了，看来 `Makefile` 写的很成功。不过主人想了想，看别人的程序，好像 `make` 完了还得 `make install` 一下，要不不专业。`make` 之后程序就编译完成了，这个 `make install` 又是干什么的呢？赶紧叫过来狐狸妹妹就开始查。我在内存里看得这叫一个着急，你说你找个 `makefile` 看看不就得了么。再说，`make install`，这多好理解的名字阿，就是安装呗。我们 `Linux` 的软件都是按照文件的类别安装进系统的各个目录中的。二进制的文件放在 `/bin/usr/bin` 这样的目录，库文件放在 `/lib/usr/lib` 等等。你这个程序刚刚 `make` 完，只在当前这个目录下生成了二进制文件和各种其他库文件和存档文件。（当然了，我说的是一般的软件，主人这个弱智软件没那么复杂。只有一个二进制文件。）你总不能每次的 `cd` 进这个目录里然后运行 `./xxxxx` 这么用吧，很麻烦的。所以要安装，把二进制文件放进 `/usr/bin` 目录，库文件放在 `/usr/lib`，其他相关文件放在 `/usr/share` 之类的目录，这样你无论在哪，都可以调用这个程序了。那这个 `make install` 的过程，其实就是把编译好的文件复制到相应的目录去这么个动作，由于这是要往系统目录复制东西，所以需要在前面加上 `sudo`。还有的文件有 `make uninstal`，也是一样的道理，其实就是把相应的文件给删除。

主人查了半天终于明白了 `make install` 的道理，于是又加工了一下他的 `makefile`，加上了这么一段：

```
install:
cp ./all_in_one /usr/bin/
uninstall:
rm /usr/bin/all_in_one
```


之后，主人当然要试验一下，于是先运行 `make clean`。`make` 接到命令，赶紧打开 `makefile`，看到 `clean` 一段写的内容，于是照着执行，把当前目录下所有的 `.o` 文件和 `all_in_one` 这个祸害给删除了。清除了之前编译的内容之后，主人再运行 `make`，这回 `make` 接到命令，一看后面没有明确的参数，就按照自己的习惯在 `makefile` 里面找 `all:` 一段。找到之后发现要想完成这段，需要 `main.o`，`board.o` 和 `ai.o` 于是再一次找 `main.o:board.o:` 以及 `ai.o:` 三段内容，并依次按照上面些的命令运行，最终得到 `all_in_one`。（顺便说一句，如果 `make` 找不到 `all:` 一段，就会执行 `makefile` 里写在最前面的一段。）然后主人在运行 `make install`，`make` 一看 `install:` 段，简单，把文件拷贝过去了。主人特意跑到别的目录，运行 `all_in_one`，果然可以执行，看来是安装成功了。不过主人也知道这么个程序放在系统里不是很合适，于是又回来执行了 `make uninstall`。



5.4 分析师

“2010年11月15日 晴

昨天 mm 来我家，我给她看了我编的 rubbish 合集程序。mm 和那个程序玩了几把‘九宫格’的游戏，竟然还输了一回，说明我写的算法还是不错的，呵呵。看来编程也不是那么难，只要找到合适的切入点，潜心学习，没什么学不会的。编游戏看来是个不错的开始，虽然这个程序界面相当简陋，而且肯定有很多的潜在问题，不过对我自己来说还是个很好的开始。这个 Linux 系统学习编程好像不是很方便，不过它倒是让我能够比别人更了解程序编译的整个过程。虽然我写的程序肯定是很幼稚的，不过今天忽然有个想法——把它发布出去！我的程序也要开源，哈哈。这样让更多的同学们分享我的成果，并且告诉他们，新手也可以写出有用的东西来。”

哎～主人最近是越来越不着调了，他竟然想把那个超级无敌 rubbish 金刚发布到网上去！祸害我们一个系统还不够，还要祸害多少青春懵懂的 Ubuntu 啊。

其实那家伙也干不了啥正经事，主要就是能陪主人玩游戏，所以才这么受宠。说来也怪了，我们软件库里面那么多游戏主人都没兴趣，不知道为什么就对这么个简单的九个格子的游戏情有独钟。不过也难说，毕竟是他自己编的嘛，谁的孩子谁不爱呢。可是您自己喜欢那就偷偷摸摸自己玩就行了，干嘛非要发布到网上让他去祸害别的电脑呢？

这家伙经常的申请了内存不释放，有时候还假死，这要是不明所以的人用了这个程序没准还抱怨我们 Linux 系统不稳定呢。当然了，发牢骚归发牢骚，主人的命令我们还是得执行，这会狐狸妹妹就正忙着把主人的 rubbish 金刚传到网上去呢。

过了一会，狐狸妹妹忍着笑就过来了：“你知道主人怎么发布他那个程序嘛？主人直接把编译出来的 all_in_one 文件，贴到了论坛里面。哈哈，他以为这样直接就能运行呢。笑死了。”恩，看来主人还有很长的路要走阿。这个二进制文件看上去就是单一的文件，但其实他运行起来是需要很多库文件来协助的，不是拿到哪都能运行的。在他们 Windows 界其实也是这样，98 的程序直接拿到 xp 下不一定能运行，xp 的程序也有很多装不到 win7 上。但是由于他们的版本比较少，而且系统的各种库和接口啥的都比较统一，所以很多人觉得程序就是一个 exe 拷贝到哪里都可以运行。主人这个程序，如果拷贝到一个和我们相同的系统上，那肯定是可以运行（也肯定可以造成内存泄露和假死，哼哼）。可是不一定别人的系统就跟你的系统一模一样啊是不？

尤其我们 Linux，发行版是五花八门，就算相同的发行版，版本也不一定一样。如果系统里没有这个程序所依赖的那些库的话，这个程序肯定是运行不起来的。要想知道一个程序依赖于哪些库可以用 ldd 命令来查看。我趁主人不注意的时候叫来 ldd，运行了一下 ldd./all_in_one，看看这个软件都依赖什么。Ldd 向我汇报如下：

```
$ ldd ./all_in_one
linux-vdso.so.1 => (0x00007fff31dff000)
libc.so.6 => /lib/libc.so.6 (0x00007ff61215c000)
/lib64/ld-linux-x86-64.so.2 (0x00007ff6124cc000)
```

看来依赖的还不是很多，算是最基本的了，可也照样不能随便放到别的系统上运行啊。比如这个 ld-linux-x86-64.so.2，明显只有 64 位系统才可以。libc.so.6 这个虽然是个系统就有，但是版本也得合适，不合适也不行。这也是为什么 Linux 上发布的软件好多都是源码包的原因，因为系统环境实在是各式各样，还是把源码放到目标系统上编译一遍来的方便。然而我这些话也就跟您说说，我主人听不见我说话的，所以他还是把那个 rubbish 金刚直接贴到网上了。

几天之后，主人就收到了应有的回应：“这个程序在我这运行不了啊？”“楼主再好好看看吧。”“我是 32 位机啊，运行不了，楼主提供个 32 位的版本吧。”……主人终于意识到这二进制文件不是放到哪都能执行的，可要是他们每个人编译一个针对他们系统的版本也是不大可能，好吧，那就发布源码！可是问题又来了，源码发布的软件包应该是什么样子呢？为了不再次体现自己的无知，主人从网上随便下载了个软件的源码包，打算来学习一下。主人下的这个软件是 Lynx，这是一个字符界面的浏览器，倒不是主人想要用他，主要是这个体积不大，拿来体验编译安装的过程而已。

这个软件包下载下来叫做 lynx2.8.7.tar.gz。这种包是下载软件最经常找到的了。另外还有就是 tar.bz2 格式，跟 tar.gz 的没什么区别，只是压缩格式不同而已，就像一个 rar，一个是 zip。我经常听到很多其他的笨兔兔抱怨他们的主人围着个 tar.gz 包不知道该怎么办，自己急的直打英文字也没办法。还好我的主人了解的多一点，知道这样的包是怎么回事。其实事情是

这样的：话说有个软件叫 tar，基本上每个 linux 都会带着这么个软件，我这里也是。这个软件是干什么的呢？是个打包裹的，不过他可不是邮递公司的那种，不会把打好的包扔来扔去。他的能力有点像查皮那里的 winzip，他能把很多文件和目录收拾在一起，打成一个包裹，也就是生成一个 tar 包文件。可是跟 zip 不一样的是，tar 只管打包，不管压缩。原来那些零碎的小文件有多大，打成 tar 包之后还是多大，只是变成一个整个的文件了而已。有人说，那我想压缩怎么办？别急，我这还有另一个软件，叫 gzip。这个软件就是专门负责压缩的，但是他只能压缩一个文件，不能像 winzip 那样能压缩一个目录里的好多文件。这样，tar 和 gzip 就成黄金搭档了（有脑白金么？），要想实现 winzip 那样的功能，就得 tar 和 gzip 联手协作。

比如有个目录叫 aaaa，里面有好几十个文件，总共有 10M。想要压成 zip 那样的压缩包，那就先让 tar 出手，把 aaaa 目录打成一个包裹文件——因为 gzip 只能压缩一个文件嘛。这样 tar 就把这个目录打成了 aaaa.tar 文件，这个文件还是 10M 大。然后由 gzip 出场，把这个文件压缩，压缩完了得标明一下啊，所以就又把文件名改了，叫做 aaaa.tar.gz，表示这个文件经过了 gzip 压缩，这时候这个文件就小了，可能 5M，可能 7M 的就没准了。有时候还有叫 xxx.tgz 的包，也是一个意思，只是把 .tar.gz 扩展名合并了而已。这下就明白了吧，这个 tar.gz 包其实就相当于 rar 或者 zip 的压缩包。那下载来的 tar.gz 包的软件怎么装呢？那当是先把包解开再看了，得先解开压缩包看看里面是什么内容才能知道怎么装啊，就像我问你 RAR 包怎么装，你能知道么？

我们现在知道 tar 包就是个压缩包，就是个大包裹，里面有什么东西不一定。那一般拿到一个 tar 包的软件应该怎么办呢？

你收到一个包裹后怎么办？当然是先打开啦！先找剪子啦，小刀啦之类的工具把包裹拆开，然后看看里面有什么东西，根据里面东西的不同来决定怎么处理。里面要是家里寄来的松子核桃什么的，就赶快吃了；要是比较难吃的松子核桃什么的，就跟同事分着吃了；要是部手机，就赶快拿出来试试；要是下面还有把手枪，就赶紧拿刚才那手机报警。这些大概不用我说，智力正常的人都应该知道怎么做，其实 tar 包也是如此。拿到一个 tar 包之后，先用你的工具把 tar 包拆开。工具是啥？有道是解铃还须系铃人，tar 打的包，当然还用 tar 来解了。

当然，你也可以用那个叫做文档管理器的家伙，他的中文名字叫归档管理器（叫 gui ~ dang ~ guan~li~qi~~？那是小沈阳！），他的英文名字叫 file-roller。不过其实他只是个负责用图形界面和主人交流的家伙，真正干活的还得是 tar。tar 包解开后，一般会得到一个目录，里面有很多的文件。然后干什么呢？有的同学记起来了，看看里面的东西啊。

一般包里面应该有个 README 文件，里面写着这个软件是干什么用的、怎么安装、怎么用、作者是谁、干什么的、爱吃什么、身高多少，腰围裤长……等等信息吧。也可能安装的方法写在一个叫做 INSTALL 的文件里。总之，应该有相应的文档文件来告诉你这个软件怎么装。不过也有时候软件的作者不厚道，或者忘性大，没有写 README 或者 INSTALL 文件，或者文件有，但是没说清楚到底怎么装，那怎么办呢？用自己的头脑判断一下吧。

主人把下载来的软件包移动到了他的家目录下，也就是 ~ 目录，然后运行：

```
tar -xzf ./lynx2.8.7.tar.gz
```

把这个压缩包解了出来。解压之后是一个目录，叫做 lynx2-8-7。主人运行 `cd lynx2-8-7` 进入到这个目录里面，用 `ls` 看了一下，发现有很多文件，其中有个叫做 `INSTALLATION` 的，里面大约应该是写着安装方法吧。不过主人根本没打开这个文件，而是看到了 `configure` 脚本，于是用自己的头脑判断，直接运行了：

```
./configure
```

这个 `configure` 是干什么的呢？

我们知道 `gcc` 施工队听 `make` 包工头的指挥，`make` 包工头根据 `Makefile` 安排工作。这样，如果想把一堆源码编译成二进制的程序，只要执行一下 `make`。执行之后 `make` 会在当前目录下寻找 `Makefile`，然后按照上面写的方案，指挥施工队：在这盖个大裤衩、在那盖个水煮蛋、再跟中间垒个鸟窝……然后施工队按照命令一点点施工，直到最终完成任务。然而事情有时候并不是那么简单，没准包工头 `make` 下达大裤衩命令之后，施工队回来报告：这地方挨着鞭炮厂阿，盖大裤衩还不烧着了？包工头说：那先盖水煮蛋吧。施工队又报告：这地方长年干旱，地下水位也底，这点水连泡面都不够，别说煮蛋了。包工头只好说：那就先盖那鸟窝，总行了吧？施工队再说：鸟窝倒是能盖，就是这地方不通天然气，点不着窝里那火炬阿……

遇到这些问题，都是由于开工之前没有对这里的环境，现有的材料进行合理分析导致的，那么我们的这个 `configure`，就是这样一名分析师。

`configure` 跟 `make` 不一样，他并不是常驻在我这里的软件，而是每个源码发行的软件自带的一个脚本。简单点说，幸福的 `make` 都是一样的，不幸的 `configure` 各有各的不幸。有了 `configure` 之后，编译软件的步骤就多了一步——`./configure` 让这个分析师首先开始工作，他会检查当地的情况，有什么材料啊，什么库啊，什么编译器啊之类的，都检查一遍，然后因地制宜的设计一份 `Makefile`。如果有足够的水，才允许煮蛋，有远离火种的安全空间才能晾裤衩，如果条件不满足，`configure` 就会报告错误，告诉主人这里缺少什么，等主人想办法弄齐了再来编译。如果条件满足可以施工，`configure` 就会出一份 `Makefile`，注意，一般 `configure` 调查前，目录下是没有 `Makefile` 的（当然，没有 `configure` 的情况另说）。

主人运行 `./configure` 之后，`configure` 对我们系统进行了检查，发现可以施工，于是就生成了 `Makefile` 文件，主人接着运行 `Make`，开始编译，由于软件很小，马上就编译完了，最终主人运行 `make install`，把这个软件安装在了我们系统里。

有了这么一番感受之后，主人知道了一个源码发布的软件包的大概样子。于是照着人家这个软件包，对比一下自己的这个软件。`makefile` 是现成的，只要再增加一个 `configure` 脚本，检查一下系统中有没有 `gcc` 之类的编译工具，以及库文件啥的版本就好了。要写这么个脚本，那就得学习 `shell` 脚本编程了，主人之前可没学过，这会现学有些来不及，那怎么办呢？对，照猫画虎，看看那个 `lynx` 的 `configure` 怎么写的，跟着学就好了。主人想的挺好，叫来 `gedit` 打开 `configure` 一看就傻了！

——洋洋洒洒连注释带语句一共三万六千三百七十九行！这还不得看到猴年马月啊。



“2010 年 12 月 20 日 回冷

总算把代码发布到网上了。很多热心的网友提出了不错的建议和意见。才发现我写出来的程序还真的很白痴。经过了这么一个过程，确实在编程方面长进了不少，了解了很多以前不了解的事情。知道了原来好几千行的 `configure` 脚本都不是人写的，怪不得我写不出来呢，呵呵。

虽然在家编程很顺利，但是一上班还是头大。这个大企业说是管理正规，但是也太有点官僚主义了。hub 坏了想买一个都这么费劲，组长批，部门经理批，财务批，折腾一套下来，半个月都过去了。有这功夫我自己拿电线焊一个都焊出来了。”

主人的 `rubbish` 金刚终于还是放到网上去了，幸运的是，网络上的众多牛人们，为他修改了很多问题，把他调教的规规矩矩的，也不浪费内存了，也不乱改文件了，也不死机了，腰不酸了，背不疼了，现在我们都开始喜欢这个家伙了，他好，我们也好～

这大概就是开源的力量吧。我们现在都不好意思叫他 `rubbish` 了，就直接叫金刚。金刚发布前的那段时间，我们几个软件都很纠结，担心主人运行他，担心系统被他搞坏。那时候主人也很纠结，纠结的是怎么能够把他发布到网上去。那时候主人整天研究 `shell` 编程的技术，天天对着那三万多行的 `configure` 代码，出神的看着，嘴里念叨着：“这是哪位神仙大姐写的脚本啊……，这么多行得写多长时间啊……”，

直到有一天，他终于将眼睛聚焦在了 `configure` 文件前面的那段注释中的一句话：

```
# Generated by Autoconf 2.52.20081225
```

主人顿时如醍醐灌顶一般，看着这一行注释，看着这个 `Autoconf`，心里反复的呼喊，声音越来越强烈，直到终于爆发，脱口而出：“靠！原来是用软件自动生成滴！！！”

顿悟之后主人立刻叫来狐狸，本着“内事不明问老婆，外事不明问 google”的宗旨，直奔 `ww.google.com` 而去。

一番查找后，主人终于大致了解了 `autoconf` 这个软件。咱说 `gcc` 他们就像施工队，`make` 就是包工头，`configure` 就是分析师，那这个 `autoconf` 大概就是市政规划局了。有了他，什么 `Makefile`，`configure` 脚本，全都不用自己写，都由他一手代办。规划局的工作，就是根据源代码的结构和组成，来决定如何根据环境因地制宜，就地取材的施工，最终派出一个专门的分析师——也就是 `configure`。之后在安装的时候 `configure` 就可以根据目标系统的环境以及既定的几套施工方案，来写出 `Makefile`，再交给那里的 `make` 去指导施工。虽然软件叫做 `Autoconf`，但其实并不是只有他一个人。既然叫做规划局嘛，那就不可能是一个人，你见过哪地方的规划局就一个局长了？我们的规划局成员有四个：`aclocal`、`autoconf`、`automake`、`autoscan`。你要想自动创建 `makefile` 和 `configure` 脚本，就得跟这哥四个说。虽然可能你已经有 `makefile` 了，只是缺少 `configure`，但那不行，他们向来是买一送一，搭配销售。你写的 `makefile` 是没用的，必须得用他们创作的 `configure` 和 `makefile`，这个具体咱们待会再说。这四个人各司其职，其中，`autoscan` 是负责初步审查项目的。你的工程图纸画好了，得先拿给他看。他看了一遍之后，会给你写个报告。怎么还写报告？当然了，规划局嘛，审批个这处理个那的，不都是部门之间报告来，报告去的么。`autoscan` 写的这个报告叫做 `configure.scan`。

然后你要做的就是拿着这个报告去找 `aclocal` 审批。但是如果你真的直接拿着 `autoscan` 写出来的报告去，准过不了。`autoscan` 说是初审，但其实他本事一般，图纸都不一定看的懂。（那为啥他能在这规划局干活？听说主要因为他爸是啥刚。）他写出来的报告一般驴唇不对马嘴，你还得动手改改。改过的报告还得改个名字，叫做 `configure.in`。`aclocal` 也知道 `autoscan` 的那点事，都是圈里混的，谁还不明白谁阿。拿过 `configure.in` 来就知道，这是你改过的了，那就好好看看。看完了就批了么？一看你就没跟规划局干过，这才哪到哪阿。看完了他也会写个报告，叫做 `aclocal.m4`，之后你就要拿着这两个报告找到 `autoconf`。

`autoconf` 就是专门负责指派分析师的了。他看了两份报告后，一般会沉思一会，说说目前如何困难，人手不足之类的话，最终在你一再的苦苦哀求以及威逼利诱之下（一般威逼的不多），无可奈何的说：好，就给你派个分析师吧！于是，一个 `configure` 脚本诞生了。但这就完了么？当然没那么简单，这个 `configure` 是有条件的，他必须搭配规划局制定的 `Makefile.in` 才能工作。

有人问了，这个 `Makefile.in` 是什么阿？我已经有了 `Makefile` 还要他干什么？咱不是说了么，你的那个 `Makefile`，甭管写的多么的天花乱坠，也是白搭，人家规划局派出来的 `configure` 根本都不会瞧上一眼，人家 `configure` 要写自己的 `Makefile` 来用。你又得说了，那你

这 configure 就赶快写出来自己的 Makefile 阿。你看你，不讲道理了不是？这 makefile 那么复杂，哪能就这么凭空写出来阿，总得有个参考，有个蓝本，有个全市统一 Makefile 模板之类的吧。这个模板，就是 Makefile.in。

那么这个文件从哪来呢？嘿嘿，不是还有个 automake 没出场呢么？这回你该求他了。automake 就是专门写 configure 需要的 Makefile.in 的，您看咱规划局给您搭配的多好。那么 automake 直接就能写 makefile.in 么？也不是，人家比较忙，这个您也得理解，局里那么多事情，就算没啥正经事，也少不了大家一起出去会个餐，考个察什么的。就算不会餐不考察，谁也免不了得上个网，偷个菜，斗个地主扫个雷啥的吧？所以呢，automake 是没功夫从头给你写出一份 makefile.in 的，你得先给 automake 写好一个框架，然后人家才好动笔。这个框架呢，就叫做 Makefile.am。有了这个框架，交给 automake，他就可以给你写出 Makefile.in 了。之后就可以把这些文件连同源码一起打包发布了，用户拿到软件包之后只要：

```
./configure
make
make install
```

就可以把软件安装上了。

刚才说到哪了？哦对，主人终于闹明白了，原来他一直说的那个简直不是人写出来的 configure 脚本还真不是人写出来的。于是上网学习 autoconf 的用法，经过一番辛苦的学习，他终于理清了头绪，开始动手了。

只见主人来到存放金刚源码的那个目录，目录里面现在有 main.c、board.c、ai.c、board.h、ai.h 几个文件。金刚的功能也被主人删除的只剩下跟主人下棋这么一个功能——因为就这个功能还算有点用。来到目录之后，主人先是叫来 autoscan 来进行扫描。咱不是说了 autoscan 是初审么。只见 autoscan 同学大摇大摆来到主人的金刚源码目录，东瞅瞅，西看看，挨个打开每一个文件，终于搞清楚了各个文件之间的关系，然后按照一套很官方的格式，写了初步审核报告书，就是那个 configure.scan，之后就会去睡觉去了。主人拿到报告书，当然知道，这只是万里长征，才走完了第一步。赶紧叫来 gedit 小弟，修改报告书。把一些完全不着边际的东西删掉，或者注释掉（也就是在那行的前面加上 # 号。）修改后的报告书是这个样子的：

```
#
-*- Autoconf -*-
# Process this file with autoconf to produce a configure script.

#AC_PREREQ([2.64])
AC_INIT(main.c)
AM_INIT_AUTOMAKE(all_in_one, 1.0)
#AC_CONFIG_SRCDIR([ai.c])
#AC_CONFIG_HEADERS([config.h])
```

```
# Checks for programs.
AC_PROG_CC

# Checks for libraries.

# Checks for header files.
#AC_CHECK_HEADERS([stdlib.h])

# Checks for typedefs, structures, and compiler characteristics.

# Checks for library functions.

AC_OUTPUT(Makefile)
```

看着挺多，其实都是注释，真正有用的就这么几行：

`AC_INIT(main.c)` ——这句是说明这个工程的主要图纸是那个文件。

`AM_INIT_AUTOMAKE(all_in_one,1.0)` ——这行是汇报这个项目的名称，叫做 `all_in_one`（大概是一锅烩的意思。），版本是 1.0 版。

`AC_PROG_CC` ——这句是说，最终的 `configure` 需要检查 c 语言编译器是否正常。

`AC_OUTPUT(Makefile)` ——这行是说明，最终的 `configure` 需要产生的文件，叫做 `Makefile`。

其他的，都是废话。

主人把改好的报告改名为 `configure.in`，然后叫醒了 `aclocal` 来看报告。`aclocal` 不紧不慢的起床，伸着懒腰走进工作间，拿起桌上的茶水杯，掀开杯盖，撇一撇浮在水面上的茶叶，拿嘴吹一吹，喝一口，盖上杯盖，放下杯子，开始看报告。扫了两眼后，就知道是怎么回事了，也不用主人多说话，直接写了一份复查报告，叫做 `aclocal.m4`，扔给了主人，就赶紧回去继续睡觉去了。要说人家 `aclocal` 的办事效率那还真是一不错，毕竟人家写的 `aclocal.m4` 不用主人修改，直接就可以往上交了。

主要领导终于出场了，主人赶紧又叫来 `autoconf`，让他指派分析师 `configure`。`autoconf` 这回倒是没太耽搁，看了看两份报告后，就生成了 `configure` 脚本。

那么这个脚本得搭配他们专门的 `Makefile` 蓝本阿，所以主人又找来 `automake`。让她写，`automake` 拉着长声说：“这～个～我们这里～工作也比较忙啊，是不是？……要按说呢，这个文件我是应该给你写滴～不过我们这里每天这么多人来～我要是一个一个写呢～哪天才能写完啊？同志啊，为了帮助我们提高办事效率，也为了你自己早点拿到 `Makefile` 蓝本，更为了我们祖国的三个现代化，诶不对，五个，也不对，几个来着？咳，甭管多少个吧，反正呢，您是不是自己先写个草稿给我，我也好帮你赶快写出蓝本呀。”主人听得都快扔板砖了，心说不就你想犯懒这么点事么，至于跟我撒这么半天么。赶紧动手写草稿，这个草稿叫做 `Makefile.am`，内容很简单：

```
AUTOMAKE_OPTIONS=foreign
bin_PROGRAMS=all_in_one
all_in_one_SOURCES=main.c ai.c board.c
```

第一行呢，是行业规定，就这么写。（其实这是 automake 的选项啦。Automake 主要是帮助开发 GNU 软件的人员来维护软件，所以在执行 automake 时，会检查目录下是否存在标准 GNU 软件中应具备的文件，例如 'NEWS'、'AUTHOR'、'ChangeLog' 等文件。设置为 foreign，automake 就不会查这些了。）

第二行吧，就是说明编译之后的程序应该叫做 all_in_one。

第三行嘞，就是说这个工程包括 main.c ai.c board.c 这三个文件。

就这么简单，草稿就写完了，之后再把 automake 叫出来，总算是给写出了 makefile 的蓝本——Makefile.in。

做完了这些之后，这个工程就可以打包发布了。用户拿到这个包，解开之后，就直接 ./configure ;make ;make install 就把软件安装上了。主人欣喜的看着自己整出的这个像模像样的软件，看看 configure 脚本，四千多行！心想：不知道的人看见在这个脚本一定以为我是大牛呢吧，哈哈哈～

再运行一下 configure，看看生成的 Makefile，五百多行，哈哈，隐然感觉自己已经成为高手了一样～

于是，在主人的 YY 中，我们结束了那一天的工作

