They who can give up essential liberty to obtain a little temporary safety, deserve neither liberty nor safety.

—— *Benjamin Franklin*

2 搭建你的工作环境

我知道，现在你已经开始摩拳擦掌准备大干一场了，因为你发现，开头并不是那么难的。你可能想到了 Linus，或许他在写出第一个引导扇区并调试成功时也是同样的激动不已；你可能在想，有一天，我也要写出一个 Linux 那样伟大的操作系统！是的，这一切都有可能，因为一切伟大必定是从平凡开始的。我知道此刻你踌躇满志，已经迫不及待要进入操作系统的殿堂。

可是先不要着急，古人云：“工欲善其事，必先利其器”，你可能已经发现，如果每次我们编译好的东西都要写到软盘上，再重启计算机，不但费时费力，对自己的爱机简直是一种蹂躏。你一定不会满足于这样的现状，还好，我们有如此多的工具，比如前面提到过的 Bochs。

在介绍 Bochs 及其他工具之前，需要说明一点，这些工具并不是不可或缺的，介绍它们仅仅是为读者提供一些可供选择的方法，用以搭建自己的工作环境。但是，这并不代表这一章就不重要，因为得心应手的工具不但可以愉悦身心，并且可以起到让工作事半功倍的功效。

下面就从 Bochs 开始介绍。

2.1 虚拟计算机 Bochs

即便没有听说过虚拟计算机，你至少应该听说过磁盘映像。如果经历过 DOS 时代，你可能就曾经用 HD-COPY 把一张软盘做成一个 .IMG 文件，或者把一个 .IMG 文件恢复成一张软盘。虚拟计算机相当于此概念的外延，它与映像文件的关系就相当于计算机与磁盘。简单来讲，它相当于运行在计算机内的小计算机。

2.1.1 Bochs 初体验

我们先来看看 Bochs 是什么样子的，请看图2.1和图2.2这两个屏幕截图。

要看清楚哦，你看到的不是显示器，仅仅是窗口而已。如果你是第一次接触“虚拟机”这个东西的话，一定会感到很惊讶，你会惊叹：“啊，像真的一样！”没错，像真的一样，不过窗口的标题栏一行“Bochs x86-64 emulator”明白

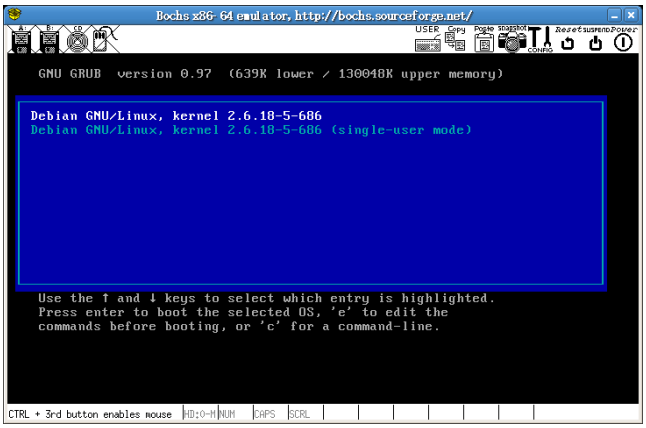

图 2.1 Bochs 中的 grub

图 2.2 Bochs 中的 Linux

无误地告诉我们，这仅仅是个“emulator”——模拟器而已。在本书中我们把这种模拟器称为虚拟机，因为这个词使用得更广泛一些。不管是模拟还是虚拟，我们要的就是它，有了它，我们不再需要频繁地重启计算机，即便程序有严重的问题，也丝毫伤害不到你的爱机。更加方便的是，你可以用这个虚拟机来进行操作系统的调试，在它面前，你就好像是上帝，你可以随时让时间停住，然后钻进这台计算机的内部，CPU 的寄存器、内存、硬盘，一切的一切都尽收眼底。这正是进行操作系统的开发实验所需要的。

好了，既然 Bochs 这么好，我们就来看看如何安装，以及如何使用。

2.1.2 Bochs 的安装

就像大部分软件一样，在不同的操作系统里面安装 Bochs 的过程是不同的，在 Windows 中，最方便的方法就是从 Bochs 的官方网站获取安装程序来安装

（安装时不妨将“DLX Linux Demo”选中，这样你可以参考它的配置文件）。在 Linux 中，不同的发行版（distribution）处理方法可能不同。比如，如果你用的是 Debian GNU/Linux 或其近亲（比如 Ubuntu），可以使用这样的命令：

```
▷ sudo apt-get install vgabios bochs bochs-x bximage
```

敲入这样一行命令，不一会儿就装好了。

很多 Linux 发行版都有自己的包管理机制，比如上面这行命令就是使用了 Debian 的包管理命令，不过这样安装虽然省事，但有个缺点不得不说，就是默认安装的 Bochs 很可能是没有调试功能的，这显然不能满足我们的需要，所以最好的方法还是从源代码安装，源代码同样位于 Bochs 的官方网站[1]，假设你下载的版本是 2.3.5，那么安装过程差不多是这样的：

```
▷ tar vxzf bochs-2.3.5.tar.gz
▷ cd bochs-2.3.5
▷ ./configure --enable-debugger --enable-disasm
▷ make
▷ sudo make install
```

注意“./configure”之后的参数便是打开调试功能的开关。在安装过程中，如果遇到任何困难，不要惊慌，其官方网站上有详细的安装说明。

2.1.3 Bochs 的使用

好了，Bochs 已经安装完毕，是时候来揭晓第 1 章的谜底了，下面我们就一步步来说明图1.1的画面是怎样来的。

在第 1 章我们提到过，硬件方面需要的是一台计算机和一张空白软盘，现在计算机有了——就是刚刚安装好的 Bochs，那么软盘呢？既然计算机都可以“虚拟”，软盘当然也可以。在刚刚装好的 Bochs 组件中，就有一个工具叫做 bximage，它不但可以生成虚拟软盘，还能生成虚拟硬盘，我们也称它们为磁盘映像。创建一个软盘映像的过程如下所示：

```
▷ bximage
========================================================================
                                 bximage
                  Disk Image Creation Tool for Bochs
        $Id: bximage.c,v 1.32 2006/06/16 07:29:33 vruppert Exp $
========================================================================

Do you want to create a floppy disk image or a hard disk image?
Please type hd or fd. [hd] fd ↩

Choose the size of floppy disk image to create, in megabytes.
Please type 0.16, 0.18, 0.32, 0.36, 0.72, 1.2, 1.44, 1.68, 1.72, or 2.88.
 [1.44] ↩
I will create a floppy image with
  cyl=80
  heads=2
  sectors per track=18
  total sectors=2880
  total bytes=1474560
```

[1]实际上通过命令行也可以获取源代码，只不过通常不是最新的，在此不做介绍。

```
What should I name the image?
[a.img] ↩

Writing: [] Done.

I wrote 1474560 bytes to a.img.

The following line should appear in your bochsrc:
  floppya: image="a.img", status=inserted
```

凡是有↩记号的地方，都是 bximage 提示输入的地方，如果你想使用默认值，直接按回车键就可以。在这里我们只有一个地方没有使用默认值，就是被问到创建硬盘还是软盘映像的时候，我们输入了“fd”。

完成这一步骤之后，当前目录下就多了一个 a.img，这便是我们的软盘映像了。所谓映像者，你可以理解为原始设备的逐字节复制，也就是说，软盘的第M个字节对应映像文件的第M个字节。

现在我们已经有了“计算机”，也有了“软盘”，是时候将引导扇区写进软盘了。我们使用 dd 命令[2]：

```
▷ dd if=boot.bin of=a.img bs=512 count=1 conv=notrunc
```

注意这里多用了一个参数“conv=notrunc”，如果不用它的话软盘映像文件 a.img 会被截断（truncated），因为 boot.bin 比 a.img 要小。第 1 章中我们使用这个命令时不需要此参数，因为真实的软盘不可能被“截断”——真的和假的总是会有一点区别。

现在一切准备就绪，该打开电源启动了。可电源在哪儿呢？不要慌，我们还剩一样重要的东西没有介绍，那就是 Bochs 的配置文件。为什么要有配置文件呢？因为你需要告诉 Bochs，你希望你的虚拟机是什么样子的。比如，内存多大啊、硬盘映像和软盘映像都是哪些文件啊等内容。不用怕，这配置文件也没什么难的，代码2.1就是一个 Linux 下的典型例子。

代码 2.1 bochsrc 示例

```
1  ###############################################################
2  # Configuration file for Bochs
3  ###############################################################
4
5  # how much memory the emulated machine will have
6  megs: 32
7
8  # filename of ROM images
9  romimage: file=/usr/share/bochs/BIOS-bochs-latest
10 vgaromimage: file=/usr/share/vgabios/vgabios.bin
11
12 # what disk images will be used
13 floppya: 1_44=a.img, status=inserted
14
15 # choose the boot disk.
16 boot: floppy
17
18 # where do we send log messages?
19 log: bochsout.txt
20
```

[2]如果你用 Windows，那么使用 Linux 常用命令需要额外一些劳动，比如安装一个 Cygwin，或者下载某个工具的 Windows 版本。在这里你可以简单下载一个“dd for Windows”，其下载地址为http://www.chrysocome.net/dd。

```
21 # disable the mouse
22 mouse: enabled=0
23
24 # enable key mapping, using US layout as default.
25 keyboard_mapping: enabled=1, map=/usr/share/bochs/keymaps/x11-pc-us.map
```

可以看到，这个配置文件本来就不长，除去注释之后内容就更少了，而且很容易理解，字面上稍微不容易理解的只有 romimage 和 vgaromimage[3]，它们指定的文件对应的其实就是真实机器的 BIOS 和 VGA BIOS，读者自己操作的时候要确保它们的路径是正确的，不然过一会儿虚拟机启动时可能会被提示“couldn’t open ROM image file”。读者还要注意 floppya 一项，它指定我们使用哪个文件作为软盘映像。

如果你在 Windows 下的话，romimage 和 vgaromimage 两项指定的文件应该是安装目录下的BIOS-bochs-latest和VGABIOS-lgpl-latest。当然，最保险的方法是参考安装程序自带的 DLX linux 的配置文件，将其稍作修改即可。

好了，现在一切准备就绪，是时候启动了，输入命令：

```
▷ bochs -f bochsrc
```

一个回车[4]，你想要的画面就呈现在眼前了。是不是很有趣呢？

顺便告诉你个窍门，如果你输入一个不带任何参数的 Bochs 并执行之，那么 Bochs 将在当前目录顺序寻找以下文件作为默认配置文件：

- .bochsrc
- bochsrc
- bochsrc.txt
- bochsrc.bxrc （仅对 Windows 有效）

所以刚才我们的“`-f bochsrc`”参数其实是可以省略的。读者在给配置文件命名时不妨从这些文件里选一个，这样可以省去许多输入命令的时间。

此外，Bochs 的配置文件还有许多其他选项，读者如果想详细了解的话，可以到其主页上看一看。由于本书中所用到的选项有限，在此不一一介绍。

2.1.4 用 Bochs 调试操作系统

如果单是需要一个虚拟机的话，你有许许多多的选择，本书下文也会对其他虚拟机有所介绍，之所以 Bochs 称为我们的首选，最重要的还在于它的调试功能。

假设你正在运行一个有调试功能的 Bochs，那么启动后，你会看到控制台出现若干选项，默认选项为“6. Begin simulation”，所以直接按回车键，Bochs 就

[3]Bochs 使用的 vgaromimage 来自于 vgabios 项目，如果读者感兴趣，可以去它的主页看看：http://www.nongnu.org/vgabios/。

[4]如果你正在使用的是自己编译的有调试功能的 Bochs，回车后还需要再一次回车，并在出现 Bochs 提示符之后输入“c”，再次回车。不要被这些输入吓怕了，下文有妙计可以让你不必总是这么辛苦。

启动了，不过既然是可调试的，Bochs 并没有急于让虚拟机进入运转状态，而是继续出现一个提示符，等待你的输入，这时，你就可以尽情操纵你的虚拟机了。

还是以我们那个最轻巧的引导扇区为例，假如你想让它一步步地执行，可以先在07c00h处设一个断点——引导扇区就是从这里开始执行的，所以这里就是我们的入口地址——然后单步执行，就好像所有其他调试工具一样。在任何时刻，你都可以查看 CPU 寄存器，或者查看某个内存地址处的内容。下面我就来模拟一下这个过程：

```
... ... ...
Next at t=0
(0) [0xfffffff0] f000:fff0 (unk. ctxt): jmp far f000:e05b         ; ea5be000f0
<bochs:1> b 0x7c00↩
<bochs:2> c↩
(0) Breakpoint 1, 0x00007c00 in ?? ()
Next at t=886152
(0) [0x00007c00] 0000:7c00 (unk. ctxt): mov ax, cs                ; 8cc8
<bochs:3> dump_cpu↩
eax:0x0fffaa55, ebx:0x00000000, ecx:0x00120001, edx:0x00000000
ebp:0x00000000, esp:0x0000fffe, esi:0x000088d2, edi:0x0000ffde
eip:0x00007c00, eflags:0x00000282, inhibit_mask:0
cs:s=0x0000, dl=0x0000ffff, dh=0x00009b00, valid=1
ss:s=0x0000, dl=0x0000ffff, dh=0x00009300, valid=7
ds:s=0x0000, dl=0x0000ffff, dh=0x00009300, valid=1
es:s=0x0000, dl=0x0000ffff, dh=0x00009300, valid=1
fs:s=0x0000, dl=0x0000ffff, dh=0x00009300, valid=1
gs:s=0x0000, dl=0x0000ffff, dh=0x00009300, valid=1
ldtr:s=0x0000, dl=0x0000ffff, dh=0x00008200, valid=1
tr:s=0x0000, dl=0x0000ffff, dh=0x00008300, valid=1
gdtr:base=0x00000000, limit=0xffff
idtr:base=0x00000000, limit=0xffff
dr0:0x00000000, dr1:0x00000000, dr2:0x00000000
dr3:0x00000000, dr6:0xffff0ff0, dr7:0x00000400
cr0:0x00000010, cr1:0x00000000, cr2:0x00000000
cr3:0x00000000, cr4:0x00000000
done
<bochs:4> x /64xb 0x7c00↩
[bochs]:
0x00007c00 <bogus+   0>:  0x8c  0xc8  0x8e  0xd8  0x8e  0xc0  0xe8  0x02
0x00007c08 <bogus+   8>:  0x00  0xeb  0xfe  0xb8  0x1e  0x7c  0x89  0xc5
0x00007c10 <bogus+  16>:  0xb9  0x10  0x00  0xb8  0x01  0x13  0xbb  0x0c
0x00007c18 <bogus+  24>:  0x00  0xb2  0x00  0xcd  0x10  0xc3  0x48  0x65
0x00007c20 <bogus+  32>:  0x6c  0x6c  0x6f  0x2c  0x20  0x4f  0x53  0x20
0x00007c28 <bogus+  40>:  0x77  0x6f  0x72  0x6c  0x64  0x21  0x00  0x00
0x00007c30 <bogus+  48>:  0x00  0x00  0x00  0x00  0x00  0x00  0x00  0x00
0x00007c38 <bogus+  56>:  0x00  0x00  0x00  0x00  0x00  0x00  0x00  0x00
<bochs:5> n↩
Next at t=886153
(0) [0x00007c02] 0000:7c02 (unk. ctxt): mov ds, ax                ; 8ed8
<bochs:6> trace-reg on↩
Register-Tracing enabled for CPU 0
<bochs:7> n↩
Next at t=886154
eax: 0x0fff0000 268369920
ecx: 0x00120001 1179649
edx: 0x00000000 0
ebx: 0x00000000 0
esp: 0x0000fffe 65534
ebp: 0x00000000 0
esi: 0x000088d2 35026
edi: 0x0000ffde 65502
eip: 0x00007c04
eflags 0x00000282
IOPL=0 id vip vif ac vm rf nt of df IF tf SF zf af pf cf
```

```
(0) [0x00007c04] 0000:7c04 (unk. ctxt): mov es, ax                ; 8ec0
<bochs:8> c↩
.........
```

以上带有↩符号并以加粗字体显示的是输入，其他均为 Bochs 的输出。如果你用过 GDB，你会觉得这个过程很亲切。没错，它跟用 GDB 调试程序的感觉是很相似的，最大的区别可能就在于在 Bochs 的调试模式下我们需要跟 CPU、内存、机器指令等内容打更多交道。

在上面的演示过程中，最开始的“b 0x7c00”在0x7c00处设置了断点，随后的命令“c”让代码继续执行，一直到我们设置的断点处停止，然后演示的是用“dump_cpu”指令查看 CPU 寄存器以及用“x”指令查看内存。随后用一个“n”指令让代码向下走了一步，“trace-reg on”的功能是让 Bochs 每走一步都显示主要寄存器的值。之所以选择演示这些命令，因为它们基本是调试过程中最常用到的。

如果你在调试过程中忘记了指令的用法，或者根本就忘记了该使用什么指令，可以随时使用help命令，所有命令的列表就呈现在眼前了。你将会发现 Bochs 的调试命令并不多，不需要多久就可以悉数掌握。表2.1列出了常用的指令以及其典型用法。

表 2.1 部分 Bochs 调试指令

行为	指令	举例
在某物理地址设置断点	b addr	b 0x30400
显示当前所有断点信息	info break	info break
继续执行，直到遇上断点	c	c
单步执行	s	s
单步执行（遇到函数则跳过）	n	n
查看寄存器信息	info cpu r fp sreg creg	info cpu r fp sreg creg
查看堆栈	print-stack	print-stack
查看内存物理地址内容	xp /nuf addr	xp /40bx 0x9013e
查看线性地址内容	x /nuf addr	x /40bx 0x13e
反汇编一段内存	u start end	u 0x30400 0x3040D
反汇编执行的每一条指令	trace-on	trace-on
每执行一条指令就打印 CPU 信息	trace-reg	trace-reg on

其中“xp /40bx 0x9013e”这样的格式可能显得有点复杂，读者可以用“help x”这一指令在 Bochs 中亲自看一下它代表的意义。

好了，虽然你可能还无法熟练运用 Bochs 进行调试，但至少你应该知道，即便你的操作系统出现了问题也并不可怕，有强大的工具可以帮助你进行调试。由于 Bochs 是开放源代码的，如果你愿意，你甚至可以通过读 Bochs 的源代码来间接了解计算机的运行过程——因为 Bochs 就是一台计算机。

2.2 QEMU

如果你选择在 Linux 下开发，其实 Bochs 自己就完全够用了。在本书中，今后的大部分例子均使用 Bochs 作为虚拟机来运行。如果你在 Windows 下开发的话，或许你还需要一个运行稍微快一点的虚拟机，它不是用来运行我们自己的操作系统的，而是用来运行 Linux 的，因为我们的代码是用 Linux 来编译的，并且生成的内核代码是 ELF 格式的。

之所以不用 Bochs 来装一个 Linux，是因为 Bochs 速度比较慢，这是由它的运行机制决定的，它完全模拟硬件及一些外围设备，而很多其他虚拟机大都采用一定程度的虚拟化（Virtualization）技术[5]，使得速度大大提高。

我以 QEMU 为例介绍速度较快的虚拟机，但它绝不是唯一的选择，除它之外，Virtual Box、Virtual PC、VM Ware 等都是很有名气的虚拟机。它们各有自己的优缺点，QEMU 的显著优势是它可以模拟较多的硬件平台，这对于一个操作系统爱好者而言是很具有吸引力的。

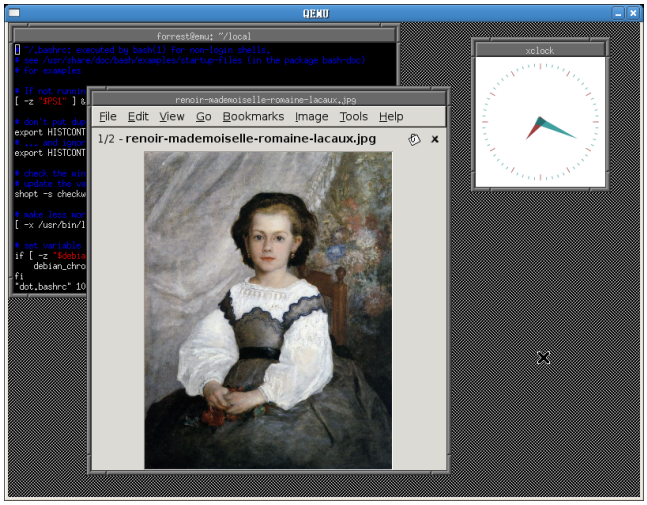

图 2.3 QEMU 中的 Linux

好了，百闻不如一见，图2.3就是 QEMU 上运行 Linux 的抓图。这里我安装的是 Debian 4.0，窗口管理器用的是 Fvwm。在我并不强大的个人电脑上，安装整个系统也没用很久，运行起来也相当顺畅。

与 Bochs 使用配置文件不同，QEMU 运行时需要的参数是通过命令行指定的[6]。比如，如果要用 QEMU 来引导我们之前已经做好的虚拟软盘，可以这样做：

```
▷ qemu -fda a.img
```

[5]读者如果对这一技术感兴趣，可在网上搜索相应资料，比如维基百科上就有个大致的介绍：http://en.wikipedia.org/wiki/Full_virtualization。

[6]实际上 Bochs 也可以用命令行指定参数，详见 Bochs 联机手册。

看起来比 Bochs 还要清爽，不是吗？其实如果你不需要调试的话，在 Linux 下也可以用 QEMU 来运行你的操作系统，这样每次都可以更快地看到结果——这便是我之前说过的“可以让你不必总是这么辛苦”的“妙计”了[7]。

2.3 平台之争：Windows 还是 *nix

读到这里，读者可能发现书中经常出现“如果你用的是 Windows”或者“如果你用的是 Linux”这样的字眼。有时这样的字眼甚至可能影响到你的阅读，如果真的这样请你原谅。我试图照顾尽量多的读者，但是对每一个人来讲，却必须面临一个选择——在什么平台下开发。本书第一版使用的是 Windows 平台，而在第二版中，我投诚了。接下来你会发现，虽然以后的行文会最大限度地兼顾 Windows，但总体是以 Linux 为默认平台的。

其实在什么平台下开发，有时纯粹是口味问题，或者是环境问题——你开始接触计算机时使用什么，很大程度上取决于你周围的人使用什么，而这往往对你的口味产生巨大而深远的影响。然而最早接触的未必是最适合的，在我亲身体会和比较之后，我决定从 Windows 彻底换到 Linux，我想在这里说说为什么。请注意这不是布道会，更不是你开发自己的操作系统必须阅读的章节，我仅仅是谈谈我个人的体会，希望能对你有所启发，同时解释一下为什么第二版会有这样的改动。

在第一版成书的时候，我已经在使用 Linux，但是用得并不多，主要是觉得用不习惯，而现在过了两三年，我已经基本不用 Windows，在 Windows 下我会觉得很不习惯。我的这一经历至少有两点启示：第一是 Linux 不好用是个误解（有一种说法是 Windows 的桌面更好用，这是个复杂的误解），好不好用是习惯问题；第二是如果你有兴趣使用一样你不熟悉的东西，不要因为刚开始的不习惯而放弃。

其实对于 Linux 和 Windows 的误解有很多，我把这种误解归结为操作系统文化上的差异。其实在提起两种系统时，人们往往拿一些具体的事情来做比较。比如比较它们的安装过程、使用方法，甚至是界面。但实际上隐藏在表面背后的是两种完全不同类型的文化，或者称之为不同的理念。

对于 Windows 而言，它的文化植根于微软公司的愿景，“让每个家庭的每个桌面上都有一台电脑”，当然他们希望此电脑内运行的是 Windows 操作系统。这个理想加上 Windows 作为商业软件的性质，决定了 Windows 具有相当程度的亲和力，用户界面显得相当友好。岂止友好，它简直友好到每个人——无论儿童还是老人，受过高等教育还是只念过小学——都能比较容易地开始使用电脑，这无疑是微软对这个社会的巨大贡献。但是界面友好并不一定就完美了，这一点暂且按下不表，我们先来说说 Linux。

Linux 的文化很大程度上来源于 UNIX，UNIX 所倡导和遵循的文化也被称为 UNIX 哲学[8]，其中很重要的一条原则叫做“做一件事并做好”[9]，这听上去

[7]其实妙计不止一条，你也可以在系统内安装两种 Bochs，一种是打开调试功能的，一种是没有打开的，你可以自由选择运行哪一种。

[8]简单的介绍可参见http://en.wikipedia.org/wiki/Unix_philosophy；若想较全面地了解，建议读者阅读 Eric S. Raymond 所著的《UNIX 编程艺术》。

[9]原文作“Do one thing, do it well”。理解这一原则的内涵及外延是理解 UNIX 世界的基本条件。

跟 Windows 的界面友好说的不是一码事，但其实仔细分析起来大有关联。做一件事并做好意味着两件事情，第一件事就是工具之间可以协同作战，不然各人做各人的，无法完成复杂应用；第二件事就是接口要统一，不然无法做到协同。这个统一的接口就是文本流（text stream），这也就意味着，命令行是 UNIX 文化的核心。而 Windows 的做法大有不同，因为要界面友好，于是不能指望用户开始就知道怎么把工具串接在一起，所以 Windows 选择任何应用都自己完成所有功能——至少让用户看起来如此，这使得每个工具都各自为战，从而增加了每个程序的复杂性和开发成本。不仅如此，由于功能都是软件开发者定好的，所以你基本上不能指望大部分的程序具备可扩展性，而在 UNIX 下，大部分的程序都可以跟其他程序协同起来完成程序不曾“设计”的功能。这也是上文我说“界面友好并不一定完美”的原因，友好是有代价的。

那么 UNIX 是一个“不友好”的系统吗？这个问题其实没有看起来那么简单。首先是 UNIX 下流行的桌面环境正在越来越“友好”，你甚至可以将其配置得看上去跟 Windows 别无二致，不过关键点不在于此，而在于长期来看，UNIX 的学习成本并不比 Windows 要高，但收益却要高得多。我们刚刚提到，友好是有代价的，而且代价比想像中要高。对于一个初学者，开始的简单容易使他产生错觉，认为电脑是个简单器械，但实际情况并非如此，一旦遇到麻烦，用户很容易陷入束手无策的境地，一旦有一件事情没有现成的软件可以解决，你马上一筹莫展。而 UNIX 不同，它的学习曲线比较陡峭，但是你一旦入门，就会发现自己的工作可以变得如此轻松而且有趣。在 Windows 中，虽然使用一个工具第一步往往很容易，但很快你就容易迷失在一堆嵌套很深名字晦涩的菜单里面，学习这些菜单可不是一件容易的事情，而且在一个工具里学会的东西到了另一个工具里可能就变了样。如果你想看看程序的帮助，有时也是件困难的事情，因为为了达到“友好”的效果，帮助经常也是一层一层的，很难找到自己需要的内容。而在 UNIX 中，所有的工具都有个手册（Manual），可以通过统一的命令“man”来查看，而且这些手册都是平坦的，你可以一口气从头看到尾，可以随时查看你所要的关键字。此外，除了少数极其复杂的工具，手册基本上是够用的。简而言之，在 UNIX 中，软件使用看起来复杂了，实际上如果你想真正掌握一个东西，用的时间不会比 Windows 中更多。况且，在 Windows 中你很难真正掌握一个东西。

我并非故意贬低 Windows，我说过它对社会的贡献巨大。对于一个平常只用电脑来收收邮件看看电影的用户，它的易用性绝对是巨大的优点，但你我不是这样的用户。我相信阅读本书的人都是程序员，而且都是像我一样喜欢探索的程序员——不喜欢探索的程序员很难有心思写自己的操作系统做消遣。一个程序员的要求和普通用户是不同的，程序员需要了解他的电脑，掌握它，并且可以熟练地让它帮助自己完成工作，从这个角度上讲，UNIX 无疑具有巨大的优势。它里面的每个工具都很锋利，你可以组合着使用，持久地使用，而且许多年都不会过时。

在这里我可以举一个我自己遇到过的例子。在我编写操作系统的文件系统时，需要多次查看某几个扇区的内容，并对其中的数据进行分析。在 Linux 中，我可以很容易地将 od、grep、sed 和 awk 等工具[10]串在一起完成这项工作，我也可以编写一个简单的脚本，将命令放在脚本中方便取用。而在 Windows 中，我

[10]这些都是 UNIX 下的常用工具，读者可以通过联机手册查看它们的用法。更多 UNIX 下的工具介绍可参考http://en.wikipedia.org/wiki/List_of_Unix_utilities。

通常只能在窗口间反复地单击鼠标，费时费力而且效率低下。类似的例子不胜枚举，你一旦熟悉了这些工具，就会发现通过组合它们，你能得到比任何图形界面工具都多的功能。而在 Windows 下就不得不看具体菜单的眼色了。不仅如此，UNIX 下的工具往往学习一次就能长久使用，很少过时。比如刚才提到的几个工具大部分有 20 年以上的历史，到现在它们依然被广泛使用，即便它们学起来会稍微难一点，平摊在 20 年里面，成本也是极低的。这就是 UNIX 哲学，你不需要重复学习，每个工具都好用，而且可以因为跟其他工具结合而发挥多种作用。

大多数 Windows 下的软件都有个毛病，它经常试图隐藏一些东西。它的本意是好的，就是让界面更“友好”，但这对程序员有时是件坏事，因为它让人难以透彻地理解软件的所作所为。或许你会说，如果你想理解，你总能理解的。没错，这跟“在 UNIX 下能做的事情在 Windows 下都能做”是相似的命题，甚至于，只要安装一些额外的软件（比如 Cygwin[11]），你可以在 Windows 下使用 UNIX 的命令。问题是即便能做，你也未必去做，这就是所谓文化的力量。理论上你在任何地方都能读书学习，但效率最高的地方还是教室和书房，在客厅舒服的沙发上，你不自主地就拿起了电视遥控器。

所以以我自己的体会而言，一个程序员最好还是使用类 UNIX 的操作系统。它能在日常生活中帮你提高自己的水平和工作效率。这一点与摄影有点类似，市面上数量最多的是傻瓜相机，但一个专业的摄影师总是会选择功能复杂的专业级设备，不是傻瓜相机不好，而是适应的人群不同，如果你想成为好的摄影师，那么上手容易的傻瓜相机一定不是你的最终选择。不是好不好的问题，是适不适合的问题。

上面论及的是两类操作系统文化上的差异，其实即便是纯粹应用层的，也有诸多误解，比如以下几条：

误解一. Linux 难安装。如果你曾被 Linux 的安装难倒过，我建议你下次试试 Ubuntu[12]。在本书第一版开始写作之时，Ubuntu 的第一个版本还没有发布[13]，但短短几年时间，它已经变成全世界最流行的发行版[14]，这与它的易装和易用性是分不开的。笔者本人大规模地使用 Linux 也是从 Ubuntu 开始的，它的安装过程一点也不比 Windows 的难，而且中文资料也相当丰富，很容易找到志同道合的人。Ubuntu 的另一特点是它的驱动程序很丰富，支持很多的硬件，大部分情况下驱动程序都能自动安装好，甚至不需要用户参与，在这一点上它甚至比 Windows 更“友好”。

误解二. Linux 难学。希望你永远记住，电脑不是个简单器械，无论是硬件、操作系统还是应用软件，都经常比看上去复杂得多。所以易用未必是好事，它肯定向你隐瞒了些什么，花一点时间绝对是值得的，尤其是当你想做一个程序员的时候。况且 Linux 也没那么难学，且不说它的图形界面越来越好用，就是完全用命令行的话，入门也相当容易。而且 Linux 世界的文档齐全且易于检索，更有高度发达的社区文化，在这里你学到的往往比预期的还要多。

误解三. Linux 难用。再强调一下，Linux 的学习曲线是陡峭的，然而一旦你度过了开始的适应期，就会发现原来命令行可以这么好用，原来有这么丰富的

[11] Cygwin 官方网站为 http://www.cygwin.com/。
[12] Ubuntu 官方网站为http://www.ubuntu.com/。
[13] Ubuntu 的第一个版本（代号 Warty Warthog）发行于 2004 年 10 月。
[14] 根据 2008 年 8 月的数据。及时情况可参考http://distrowatch.com/。

工具来提高效率，而且这些好用的工具居然都是自由的[15]！你不需要向作者付费，甚至他们鼓励你使用和传播，你甚至可以随便修改这些工具的源代码，遇到问题可以发一封邮件反馈给作者。这在 Windows 下都是很难做到的，在那片土地上你很难体会什么叫“自由”。不仅如此，当你熟悉之后会发现，同样一件事情，其实 Linux 下的解决方案往往比 Windows 下要简单。就比如我们提到过的安装 Bochs 一例吧，在 Windows 下你通常需要先到 Bochs 网站，在数次单击之后找到下载链接，然后下载，再然后是双击安装程序来安装。在 Linux 下呢，你看到了，只需要一个命令行就可以了，即便你打字的速度再慢，也比那些鼠标单击操作要快。

我在此并非贬低鼠标的好处，我每天都使用鼠标，它绝对是个伟大的发明。但是我们应该只在需要它的时候使用它，而不是试图用它来解决所有事情。这就好比图形界面是个好东西，如果你的工作是图形图像处理，很难想像没有图形界面该怎么做，但并不是图形界面在任何时候都是好的。我们应该分辨每一类工作最适合的工具是什么，而不是用一种思维解决所有问题。这也是我说“Windows 桌面好用是个复杂的误解”的原因，图形界面是个好东西，Windows 把它用得极端了。

误解四. Linux 下软件少。这是最大的一个误区，事实上很少有事情你在 Linux 下是做不到的。而且 Linux 的发行版通常都有发达的包管理工具，无论是关键字查找、安装、更新还是卸载都可以用一组统一的命令来完成。这使得你在需要某种软件时，使用简单的命令就可以找到它，很多时候你能找到不止一种。如果你想看看除了 od 之外还有哪些二进制查看器，在 Ubuntu 或者 Debian 下通过一个 `apt-cache search 'hex.*(view|edit)'`[16]命令就能找到十几种，这些都是自由软件，有命令行的也有图形界面的，而在 Windows 下，怕是又要在浩瀚的互联网中搜索了。

在 Linux 中找一些 Windows 下软件的替代品是很容易的，虽然这种对应有时并非必要。比如字处理软件就有 OpenOffice.org、KOffice、AbiWord 等选择；图像处理软件有 GIMP；多媒体播放软件有 MPlayer、Totem 等。如果你喜欢玩游戏，Linux 下的游戏数量也会让你大吃一惊，不信你可以来这里看一看：http://en.wikipedia.org/wiki/Category:Linux_games。

其实 Linux 的好处还远不止这些，众所周知的一个优点是它基本没有病毒的烦恼。不是 Linux 中开发不出病毒来[17]，而是因为 Linux 系统有自身的权限机制作保障，加上软件来源都可信赖，且大部分都是源代码开放的（其中相当一部分都是自由软件），所以说 Linux 下没有病毒烦恼并非夸张。想想你在与病毒做斗争的过程中浪费了多少时间吧，我已经很久没有这种烦恼了。Windows 或许正变得越来越稳定，但 Linux 一直都很稳定，而且你不需要整天重启你的电脑，笔者的电脑就有时几十天不重启。除非你要升级内核，否则没有很多关机和重启的理由（不管是安装还是卸载，或是对系统内的包进行升级，都不需要重启电脑）。

作为一个操作系统爱好者，使用 Linux 的理由还有一条，那就是 Linux 的内核是“自由”的，注意它不仅仅是“开放源代码”的，你不仅可以获取其源代

[15]注意这里没用“免费”这个词。Free Software 的 Free 是“自由”之意，它比“免费”一词包含了更多意义。欲获得更详细的内容请访问http://www.fsf.org/。

[16]apt-cache 是个 Debian 家族中常用的包管理命令，可以使用正则表达式来搜索软件包。

[17]关于 Linux 系统下的病毒，读者可以参考：http://en.wikipedia.org/wiki/Linux_malware。

码，而且可以自由地复制、修改、传播它，当然也包括学习它。如果你也想加入到内核黑客[18]的队伍，那么就先从使用它开始吧。Linux 不是完美的，它的问题有很多，但每个问题都是你参与的机会，而这种参与可能是你成为顶尖高手的开始。

笔者本人完全使用 Linux 来工作的时间其实很短，几年而已，但我已经深深体会到它给我带来的好处。我并非想说服你，人不能被说服，除非他自己愿意相信。我只是希望你能尝试着去用一用 Linux，或者 UNIX 的其他变种，然后用自己的判断去选择。

如果你坚持使用 Windows，没问题，两者之中都可以很容易地搭建起开发环境，本章后面的部分将会就 Linux 和 Windows 分别来做介绍。

2.4 GNU/Linux 下的开发环境

在工作环境中，虚拟机是个重头戏，所以在本章的前面单独做了介绍。除了虚拟机之外，还有几样重要的东西，分别是编辑器、编译器和自动化工具 GNU Make。

许多在 Linux 下工作的人会使用 Vi 或者 Emacs 作为编辑器。如果你有兴趣尝试，那么还是那句建议，“不要因为刚开始的不习惯而放弃”，因为它们的确是编辑器中的经典，而且和 Linux 一样，具有陡峭的学习曲线。许多人一旦学会使用就爱上它们，这其中也包括笔者自己。当然，学习它们并不是必需的，而且你的选择范围比操作系统要大多了，相信会有一款能让你满意。

对于编译器，我们选择 GCC 和 NASM 分别来编译 C 代码和汇编代码。选择 GCC 的原因很简单，它是 Linux 世界编译器的事实标准。GCC 的全称是 GNU Compiler Collection，在这里我们只用到其中的 C 编译器，所以对我们而言它的全部意义仅为 GNU C Compiler——这也正是它原先的名字。之前提到过，我们使用的 Linux 其实应该叫做 GNU/Linux，所以使用 GCC 是比较顺理成章的，那么为什么不能使用 GCC 来编译我们的汇编代码呢？何苦再用个 NASM 呢？原因在于 GCC 要求汇编代码是 AT&T 格式的，它的语法对于习惯了 IBMPC 汇编的读者而言会显得很奇怪，我猜大部分读者可能都跟我一样，学习汇编语言时使用的教材里介绍的是 IBMPC 汇编。NASM 的官方网站位于http://nasm.sourceforge.net/，你还可以在上面找到详细的文档资料。

关于 GNU Make 的介绍见本书第 5 章。

还是以 Debian 作为示例，安装 GCC 和 NASM 可以通过以下命令来完成：

```
▷ sudo apt-get install build-essential nasm
```

注意这里的 build-essential 软件包中包含 GCC 和 GNU Make。

好了，现在可以总结一下了，如果你想要搭建一个基于 Linux 的开发环境，那么你需要做的工作有以下这些：

- 安装一个 Linux 发行版，如果你对 Linux 不甚熟悉，推荐使用 Ubuntu。

[18] 如果你对成为黑客感兴趣，或许可以读一读 Eric S. Raymond 的“How To Become A Hacker”。

- 通过 Linux 发行版的包管理工具或者通过下载源代码手工操作的方式来安装以下内容：
  - 一个你喜欢的编辑器，比如 Emacs。
  - 用于编译 C 语言代码的 GCC。
  - 用于编译汇编代码的 NASM。
  - 用于自动化编译和链接的 GNU Make。
  - 一个用于运行我们的操作系统的虚拟机，推荐使用 Bochs。

再次强调，如果你在安装或使用它们时遇到困难，不要着急，也不要气馁，因为一帆风顺的情形在现实生活中着实很少见。你或许可以试试以下的解决方案（这些方法也适用于其他在自由软件的安装配置及使用等方面的问题）：

- 向身边的朋友求助。
- 使用搜索引擎看看是不是有人遇到类似的问题，那里或许已经给出解决方案。
- 仔细阅读相应资料（不要怕英文），比如安装说明，或是 FAQ。
- 订阅相应的邮件列表（Mailing List），只要能将问题描述清楚[19]，通常你能在几小时内得到答复。
- 到论坛提问。
- 如果实在是疑难杂症，你可以试着联系软件的开发者，通常也是通过邮件列表的方式（同一个项目可能有多个邮件列表，开发者邮件列表通常与其他分离）。
- 自己阅读源代码并独立解决——这或许是个挑战，然而一旦解决了问题，你将获得知识、经验以及成功的喜悦。

将来，如果一切顺利的话，你编写操作系统时的步骤很可能是这样的：

1. 用编辑器编写代码。
2. 用 Make 调用 GCC、NASM 及其他 Linux 下的工具来生成内核并写入磁盘映像。
3. 用 Bochs 来运行你的操作系统。
4. 如果有问题的话
   (a) 用各种方法来调试，比如用 Bochs；
   (b) 返回第 1 步。

[19]关于提问的技巧，请参考 Eric S. Raymond 的“How To Ask Questions The Smart Way”。

2.5 Windows 下的开发环境

我们在介绍 QEMU 时提到过，在 Windows 下你需要一个虚拟的 Linux 来帮你编译操作系统的源代码。将操作系统内核编译链接成 ELF 格式有诸多好处，我们不但可以用 Linux 下现成的工具[20]来分析编译好的内核，还可以在必要时参考 Linux 内核的源代码来帮助我们自己的开发，总之这拉近了我们与 Linux 之间的距离。所以不要因为在 Windows 下也离不开 Linux 这件事而沮丧，况且装一个 Linux 是件很容易的事情。

不过装一个*虚拟的* Linux 跟装一个*真实的* Linux 还是有所不同，主要在于两点。一是我们仅仅想用这个 Linux 来做编译链接的工作，所以在选择组件的时候尽量去除不必要的内容，这样可以节省时间和空间；二是要确保你选择的虚拟机容易跟宿主机进行网络通信，因为你需要将宿主机上的源代码拿给虚拟机来编译。

安装方法可以有多种选择，比较简单的方法是通过光盘安装，当然这个光盘也可以是“虚拟”的，也就是一个光盘映像。首先到你所中意的 Linux 发行版的官方网站下载一个安装光盘的映像，有些发行版还提供免费或付费的邮寄服务，读者可以根据自己的喜欢自行选择。这里假设你得到的是光盘映像，文件名为 inst.iso。

有了光盘映像，我们还缺少一个硬盘映像，读者可以用前文提到过的 bximage 来生成它，也可以使用下面的命令：

```
▷ qemu-img create hd.img 1500M
```

这样就能生成一个大小约为 1.5GB 的硬盘映像了。

接下来就可以进行安装了：

```
▷ qemu -cdrom inst.iso -hda hd.img -boot d
```

安装过程从略，注意尽量精简你的组件，不要安装太多无用的东西。这些组件对我们是必需的：GCC、GNU Make、NASM、Samba。如果它们在安装时默认没有装上，那么你需要在系统安装结束后将它们安装上。由于目前大多数虚拟机都具有好用的网络功能，所以安装它们并非难事。

装完之后，我们还需要解决让宿主机和虚拟机通信的问题。其实你可以把它们看成是局域网中的两台机器，局域网中适用的方法这里同样适用，所以 Samba 就很适合。

首先在 Windows 中以可读写方式共享一个文件夹，假设叫做 OrangeS，然后在虚拟的 Linux 上运行下面这条命令：

```
▷ sudo mount -t smbfs -o username=user,password=blah \
//10.0.2.2/OrangeS /mnt
```

其中假设你的宿主机 IP 地址为 10.0.2.2。这样在 Linux 的 /mnt 目录下就能看到 Windows 共享文件夹下的内容了，你可以在虚拟机中随意读写，就像对待本地文件一样。

[20] 比如 readelf。

这样一来，你的编译环境就安装完成了，接下来，如同在Linux下一样，你还需要一个编辑器。据说始终有一部分人使用记事本（notepad）来编写代码，不管基于何种理由，希望你不要这样做，因为你可以找到许多比notepad更适合编写代码的编辑器，有收费的，也有免费的，它们通常都具备关键字颜色，自动缩进等方便开发者的功能，可以大大提高工作效率。

总结一下的话，搭建一个Windows下的开发环境，你需要做以下工作：

- 安装Windows。
- 安装Bochs（安装程序可到其官方网站获取）。
- 安装一个你喜欢的编辑器用来编写代码。
- 安装一个速度较快的虚拟机，如QEMU（安装程序可到其官方网站获取[21]）。
- 在速度较快的虚拟机上安装一个Linux。
- 在虚拟的Linux中安装GCC、GNU Make、NASM、Samba——如果它们没有默认被安装上的话。
- 在虚拟的Linux和宿主机之间共享一个可读写的文件夹。

将来你的开发过程看起来很可能是这样的：

1. 在Windows中用编辑器编写代码。
2. 在虚拟Linux中用Make调用GCC、NASM及其他工具来生成内核并写入磁盘映像。
3. 在Windows中用Bochs来运行你的操作系统。
4. 如果有问题的话。
   (a) 用各种方法来调试，比如用Bochs；
   (b) 返回第1步。

2.6 总结

好了，到这里相信读者已经知道如何搭建自己的开发环境了，说白了它跟开发一个普通的软件区别基本就在一个虚拟机上。它既是我们的“硬件”，又是我们的调试器，有了它我们安心多了。那是不是马上就可以开始我们的操作系统开发之旅了呢？很遗憾，还不能那么着急，因为你知道，操作系统是跟硬件紧密相连的，如果想实现一个运行在使用IA32架构的IBM PC上的操作系统，免不了要具备相关的知识。其中的重头戏就是32位Intel CPU的运行机制，毕竟CPU是一台计算机的大脑，也是整个计算机体系的核心。

[21] QEMU的官方网站位于http://bellard.org/qemu/。

所以紧接着我们要学习的，就是要了解 IA32 保护模式。掌握了保护模式，我们才知道 Intel 的 CPU 如何运行在 32 位模式之下，从而才有可能写出一个 32 位的操作系统。

如果读者已经掌握了保护模式的内容，可以直接跳到第 4 章。