《Linux 设备驱动开发详解》（第 2 版）

作者：华清远见

第 1 章 Linux 设备驱动概述及开发环境构建

本章目标

本章将介绍 Linux 设备驱动开发的基本概念，并对本书所基于的平台和开发环境进行讲解。

1.1 节阐明了设备驱动的概念和作用。

1.2 节和 1.3 节分别讲解在无操作系统情况下和有操作系统情况下设备驱动的设计，通过对两者不同的分析讲解设备驱动与硬件和操作系统的关系。

1.4 节对 Linux 操作系统的设备驱动进行了概要性的介绍，给出了设备驱动与整个软硬件系统的关系，分析了 Linux 设备驱动的重点、难点和学习方法。

1.5 节对本书所基于的 LDD6410 ARM11 开发板和开发环境的安装进行了介绍。

本章的最后给出了一个设备驱动的“Hello World”实例，即最简单的 LED 驱动在无操作系统情况下和 Linux 操作系统下的实现。

1.1 设备驱动的作用

任何一个计算机系统的运转都是系统中软硬件共同努力的结果，没有硬件的软件是空中楼阁，而没有软件的硬件则只是一堆废铁。硬件是底层基础，是所有软件得以运行的平台，代码最终会落实为硬件上的组合逻辑与时序逻辑。软件则实现了具体应用，它按照各种不同的业务需求而设计，完成了用户的最终诉求。硬件较固定，软件则很灵活，可以适应各种复杂多变的应用。可以说，计算机系统的软硬件互相成就了对方。

但是，软硬件之间同样存在着悖论，那就是软件和硬件不应该互相渗透入对方的领地。为尽可能快速地完成设计，应用软件工程师不想也不必关心硬件，而硬件工程师也难有足够的闲暇和能力来顾及软件。譬如，应用软件工程师在调用套接字发送和接收数据包的时候，不必关心网卡上的中断、寄存器、存储空间、I/O 端口、片选以及其他任何硬件词汇；在使用 printf()函数输出信息的时候，他不用知道底层究竟是怎样把相应的信息输出到屏幕或者串口。

也就是说，应用软件工程师需要看到一个没有硬件的纯粹的软件世界，硬件必须被透明地呈现给他。谁来实现硬件对应用软件工程师的隐形？这个光荣而艰巨的任务就落在了驱动工程师的头上。

对设备驱动最通俗的解释就是“驱使硬件设备行动”。驱动与底层硬件直接打交道，按照硬件设备的具体工作方式，读写设备的寄存器，完成设备的轮询、中断处理、DMA 通信，进行物理内存向虚拟内存的映射等，最终让通信设备能收发数据，让显示设备能显示文字和画面，让存储设备能记录文件和数据。

由此可见，设备驱动充当了硬件和应用软件之间的纽带，它使得应用软件只需要调用系统软件的应用编程接口（API）就可让硬件去完成要求的工作。在系统中没有操作系统的情况下，工程师可以根据硬件设备的特点自行定义接口，如对串口定义 SerialSend()、SerialRecv()，对 LED 定义 LightOn()、LightOff()，对 Flash 定义 FlashWrite()、FlashRead()等。而在有操作系统的情况下，驱动的架构则由相应的操作系统定义，驱动工程师必须按照相应的架构设计驱动，这样，驱动才能良好地整合入操作系统的内核。

驱动程序沟通着硬件和应用软件，而驱动工程师则沟通着硬件工程师和应用软件工程师。目前，随着通信、电子行业的迅速发展，全世界每天都会有大量的新芯片被生产，大量的新电路板被设计，也因此，会有大量设备驱动需要开发。这些驱动，或运行在简单的单任务环境，或运行在 VxWorks、Linux、Windows 等多任务操作系统环境，发挥着不可替代的作用。

1.2 无操作系统时的设备驱动

并不是任何一个计算机系统都一定要运行操作系统，在许多情况下，操作系统都不必存在。对于功能比较单一、控制并不复杂的系统，譬如 ASIC 内部、公交车的刷卡机、电冰箱、微波炉、简单的手机和小灵通等，并不需要多任务调度、文件系统、内存管理等复杂功能，用单任务架构完全可以良好地支持它们的工作。一个无限循环中夹杂对设备中断的检测或者对设备的轮询是这种系统中软件的典型架构，如代码清单 1.1。

代码清单 1.1　单任务软件典型架构

```
1  int main(int argc, char* argv[])
```

```
{
  while (1)
  {
    if (serialInt == 1)
    /*有串口中断*/
    {
      ProcessSerialInt(); /*处理串口中断*/
      serialInt = 0; /*中断标志变量清 0*/
    }
    if (keyInt == 1)
    /*有按键中断*/
    {
      ProcessKeyInt(); /*处理按键中断*/
      keyInt = 0; /*中断标志变量清 0*/
    }
    status = CheckXXX();
    switch (status)
    {
      ...
    }
    ...
  }
}
```

在这样的系统中，虽然不存在操作系统，但是设备驱动则无论如何都必须存在。一般情况下，每一种设备驱动都会定义为一个软件模块，包含.h 文件和.c 文件，前者定义该设备驱动的数据结构并声明外部函数，后者进行驱动的具体实现。譬如，可以如代码清单 1.2 那样定义一个串口的驱动。

代码清单 1.2 无操作系统情况下串口的驱动

```
/**********************
 *serial.h 文件
 **********************/
extern void SerialInit(void);
extern void SerialSend(const char buf*,int count);
extern void SerialRecv(char buf*,int count);

/**********************
 *serial.c 文件
 **********************/
/*初始化串口*/
void SerialInit(void)
{
  ...
}
/*串口发送*/
void SerialSend(const char buf*,int count)
{
  ...
}
/*串口接收*/
void SerialRecv(char buf*,int count)
{
  ...
```

```
25 }
26 /*串口中断处理函数*/
27 void SerialIsr(void)
28 {
29  ...
30  serialInt = 1;
31 }
```

其他模块想要使用这个设备的时候，只需要包含设备驱动的头文件 serial.h，然后调用其中的外部接口函数。如我们要从串口上发送“Hello World”字符串，使用语句 SerialSend（“Hello World”，11）即可。

由此可见，在没有操作系统的情况下，设备驱动的接口被直接提交给了应用软件工程师，应用软件没有跨越任何层次就直接访问了设备驱动的接口。驱动包含的接口函数也与硬件的功能直接吻合，没有任何附加功能。图 1.1 所示为无操作系统情况下硬件、驱动与应用软件的关系。

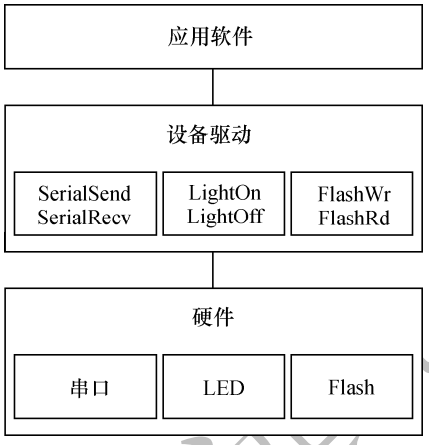

图 1.1　无操作系统时硬件、驱动和应用软件的关系

有的工程师把单任务系统设计成了如图 1.2 所示的结构，即设备驱动和具体的应用软件模块之间平等，驱动中包含了业务层面上的处理，这显然是不合理的，不符合软件设计中高内聚、低耦合的要求。

另一种不合理的设计是直接在应用中操作硬件的寄存器，而不单独设计驱动模块，如图 1.3 所示。这种设计意味着系统中不存在或未能充分利用可被重用的驱动代码。

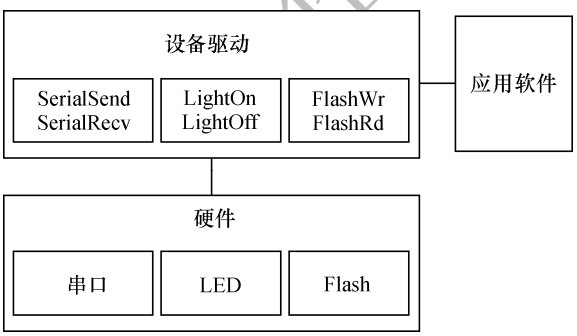

图 1.2　驱动与应用高耦合的不合理设计

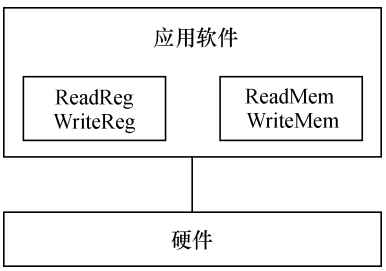

图 1.3　应用直接访问硬件的不合理设计

1.3 有操作系统时的设备驱动

1.2 节中我们看到一个干净利落的设备驱动，它直接运行在硬件之上，不与任何操作系统关联。当系统中包含操作系统后，设备驱动会变得怎样？

首先，无操作系统时设备驱动的硬件操作工作仍然是必不可少的，没有这一部分，驱动不可能与硬件打交道。

其次，我们还需要将驱动融入内核。为了实现这种融合，必须在所有设备的驱动中设计面向操作系统内核的接口，这样的接口由操作系统规定，对一类设备而言结构一致，独立于具体的设备。

由此可见，当系统中存在操作系统的时候，驱动变成了连接硬件和内核的桥梁。如图 1.4，操作系统的存在势必要求设备驱动附加更多的代码和功能，把单一的“驱使硬件设备行动”变成了操作系统内与硬件交互的模块，它对外呈现为操作系统的 API，不再给应用软件工程师直接提供接口。

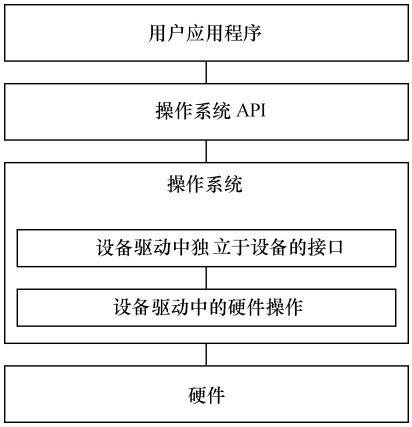

图 1.4　硬件、驱动、操作系统和应用程序的关系

那么我们要问，有了操作系统之后，驱动反而变得复杂，那要操作系统干什么？

首先，一个复杂的软件系统需要处理多个并发的任务，没有操作系统，想完成多任务并发是很困难的。

其次，操作系统给我们提供内存管理机制。一个典型的例子是，对于多数含 MMU 的处理器而言，Windows、Linux 等操作系统可以让每个进程都可以独立地访问 4GB 的内存空间。

上述优点似乎并没有体现在设备驱动身上，操作系统的存在给设备驱动究竟带来了什么实质的好处？

简而言之，操作系统通过给驱动制造麻烦来达到给上层应用提供便利的目的。当驱动都按照操作系统给出的独立于设备的接口而设计，那么，应用程序将可使用统一的系统调用接口来访问各种设备。对于类 UNIX 的 VxWorks、Linux 等操作系统而言，当应用程序通过 write()、read()等函数读写文件就可访问各种字符设备和块设备，而不论设备的具体类型和工作方式，那将是怎样的便利？

1.4 Linux 设备驱动

1.4.1　设备的分类及特点

计算机系统的硬件主要由 CPU、存储器和外设组成。随着 IC 制作工艺的发展，目前，芯片的集成度越来越高，往往在 CPU 内部就集成了存储器和外设适配器。譬如，相当多的 ARM、PowerPC、MIPS 等处理器都集成了 UART、I^2C 控制器、USB 控制器、SDRAM 控制器等，有的处理器还集成了片内 RAM 和 Flash。

驱动针对的对象是存储器和外设（包括 CPU 内部集成的存储器和外设），而不是针对 CPU 核。Linux 将存储器和外设分为 3 个基础大类。

- 字符设备。
- 块设备。
- 网络设备。

字符设备指那些必须以串行顺序依次进行访问的设备，如触摸屏、磁带驱动器、鼠标等。块设备可以用任意顺序进行访问，以块为单位进行操作，如硬盘、软驱等。字符设备不经过系统的快速缓冲，而块设备经过系统的快速缓冲。但是，字符设备和块设备并没有明显的界限，如对于 Flash 设备，符合块设备的特点，但是我们仍然可以把它作为一个字符设备来访问。

字符设备和块设备的驱动设计呈现出很大的差异，但是对于用户而言，他们都使用文件系统的操作接口 open()、close()、read()、write()等进行访问。

在 Linux 系统中，网络设备面向数据包的接收和发送而设计，它并不对应于文件系统的节点。内核与网络设备的通信与内核和字符设备、网络设备的通信方式完全不同。

另外一种设备分类方法中所称的 I^2C 驱动、USB 驱动、PCI 驱动、LCD 驱动等本身可归纳入 3 个基础大类，但是对于这些复杂的设备，Linux 也定义了独特的驱动体系结构。

1.4.2 Linux 设备驱动与整个软硬件系统的关系

如图 1.5 所示，除网络设备外，字符设备与块设备都被映射到 Linux 文件系统的文件和目录，通过文件系统的系统调用接口 open()、write()、read()、close()等即可访问字符设备和块设备。所有的字符设备和块设备都被统一地呈现给用户。块设备比字符设备复杂，在它上面会首先建立一个磁盘/Flash 文件系统，如 FAT、EXT3、YAFFS2、JFFS2、UBIFS 等。FAT、EXT3、YAFFS2、JFFS2、UBIFS 定义了文件和目录在存储介质上的组织。

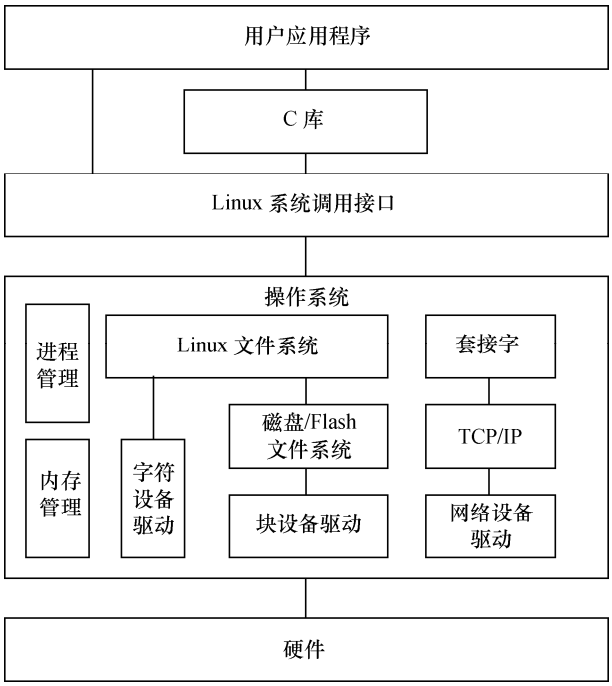

图 1.5 Linux 设备驱动与整个软硬件系统的关系

应用程序可以使用 Linux 的系统调用接口编程，但也可使用 C 库函数，出于代码可移植性的目的，后者更值得推荐。C 库函数本身也通过系统调用接口而实现，如 C 库函数 fopen()、fwrite()、fread()、fclose()分别会调用操作系统的 API open()、write()、read()、close()。

1.4.3 Linux 设备驱动的重点、难点

Linux 设备驱动的学习是一项浩繁的工程，包含如下的重点、难点。

- 编写 Linux 设备驱动要求工程师有非常好的硬件基础，懂得 SRAM、Flash、SDRAM、磁盘的读写方式，UART、I^2C、USB 等设备的接口以及轮询、中断、DMA 的原理，PCI 总线的工作方式以及 CPU 的内存管理单元（MMU）等。
- 编写 Linux 设备驱动要求工程师有非常好的 C 语言基础，能灵活地运用 C 语言的结构体、指针、函数指针及内存动态申请和释放等。
- 编写 Linux 设备驱动要求工程师有一定的 Linux 内核基础，虽然并不要求工程师对内核各个部分有深入的研究，但至少要明白驱动与内核的接口。尤其是对于块设备、网络设备、Flash 设备、串口设备等复杂设备，内核定义的驱动体系架构本身就非常复杂。

- 编写 Linux 设备驱动要求工程师有非常好的多任务并发控制和同步的基础，因为在驱动中会大量使用自旋锁、互斥、信号量、等待队列等并发与同步机制。

上述经验值的获取并非朝夕之事，因此要求我们有足够的学习恒心和毅力。对这些重点、难点，本书都会有相应章节进行讲解。

动手实践永远是学习任何软件开发的最好方法，学习 Linux 设备驱动也不例外。因此，本书专门配备了一款基于 S3C6410 的 ARM11 开发板 LDD6410（全称 Linux Device Drivers 6410，即 Linux 设备驱动开发 6410 专用板），本书中的所有实例均可在该电路板上直接执行。

阅读经典书籍和参与 Linux 社区的讨论也是非常好的学习方法。Linux 内核源代码中包含了一个 Documentation 目录，其中包含了一批内核设计的文档，全部是文本文件。很遗憾，这些文档的组织不太好，内容也不够细致。本书的参考目录中给出了一些优秀的参考书籍和 Linux 网站，并进行了简单的介绍。

学习 Linux 设备驱动的一个注意事项是要避免管中窥豹、只见树木不见森林，因为各类 Linux 设备驱动都从属于一个 Linux 设备驱动的架构，单纯而片面地学习几个函数、几个数据结构是不可能理清驱动中各组成部分之间的关系的。因此，Linux 驱动的分析方法是点面结合，将对函数和数据结构的理解放在整体架构的背景之中。这是本书各章节讲解驱动的方法。

1.5 Linux 设备驱动开发环境构建

1.5.1 PC 上的 Linux 环境

本书配套光盘提供了一个 Ubuntu 的 VirtualBox 虚拟机映像，该虚拟机上安装了所有本书涉及的源代码、工具链和各种开发工具，读者无需再安装和配置任何环境。该虚拟机可运行于 Windows 等操作系统中，运行方法如下。

（1）解压缩安装盘内的虚拟机磁盘映像 virtual-disk.rar 到本地硬盘得到 virtual-disk.vdi（至少需要 16GB 的空闲磁盘空间）。

（2）安装安装盘内的 VirtualBox 虚拟机软件。

（3）建立一个虚拟机。

① 单击“新建”按钮，指定虚拟机使用 Linux Ubuntu 系统，如图 1.6 所示。

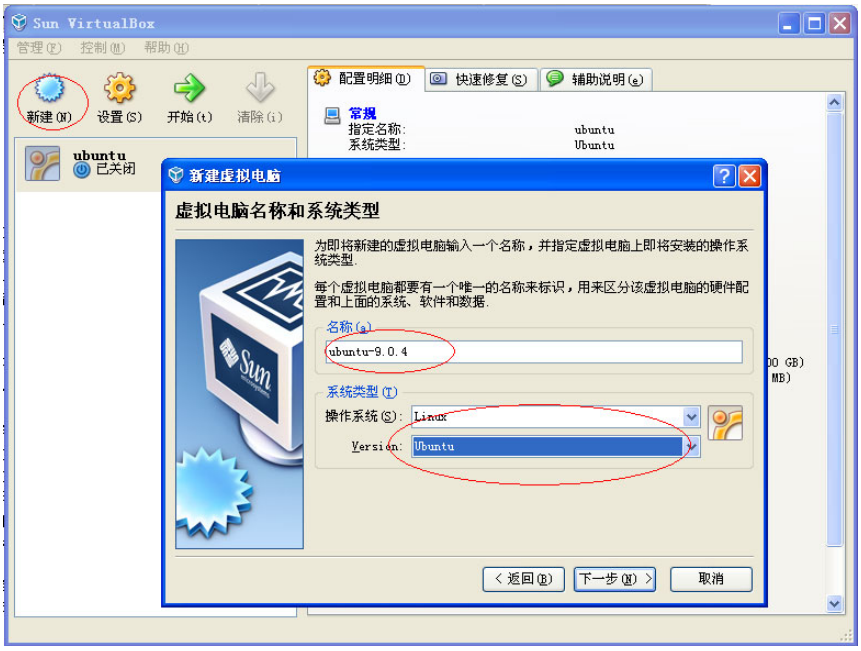

图 1.6 VirtualBox 指定使用 Ubuntu

② 单击“下一步”按钮，如图 1.7 所示，使用推荐的内存 384MB。

图 1.7　VirtualBox 中内存设定

③ 指定虚拟机磁盘映像为第一步解压缩得到的 virtual-disk.vdi，如图 1.8 所示。

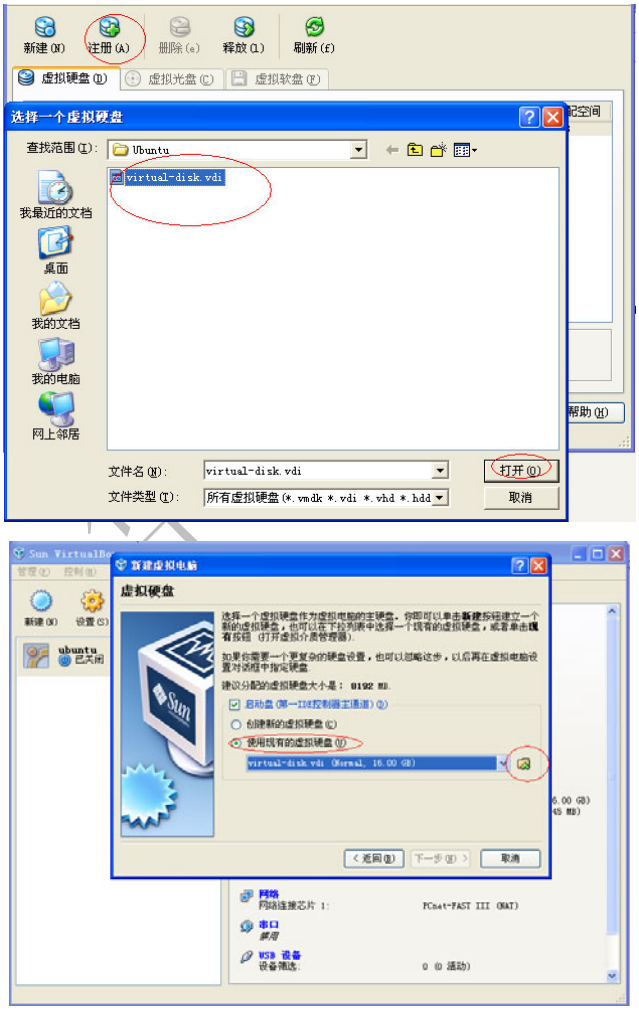

图 1.8　VirtualBox 中磁盘设定

④ 完成设置，如图 1.9 所示。

之后就可以启动虚拟机，账号和密码都是“lihacker”。本书配套源代码都位于 lihacker 主目录的 develop 目录下，几个主要项目针对/home/lihacker/develop/的子目录如下。

LDD6410 开发板内核源代码：svn/ldd6410-2-6-28-read-only/linux-2.6.28-samsung。

LDD6410 开发板 U-BOOT 源代码：svn/ldd6410-read-only/s3c-u-boot-1.1.6。

图 1.9　VirtualBox 中完成设定

LDD6410 开发板文件系统用的 busybox、jpegview、mplayer、appweb 等：svn/ldd6410-read- only/utils。
LDD6410 开发板及常用 Linux 用户空间驱动测试程序：svn/ldd6410-read-only/tests。
书中 globalmem、globalfifo 等驱动实例：svn/ldd6410-read-only/training/kernel。
Android 的源代码：git/myandroid。
NDK：android-ndk-r3。
eclipse：单击桌面上的“android-eclipse”图标，即可运行附带 ADT 的 eclipse 开发工具。

1.5.2　LDD6410 开发板

LDD6410 是本书专配的一款高端 ARM11 处理器开发板（其结构如图 1.10 所示，实物如图 1.11 所示），采用三星公司最新推出 S3C6410 处理器，芯片拥有强大的内部资源和视频处理能力，板上集成了丰富的外围接口，其主要特点如下。

（1）运行于 533MHz 的 ARM11 处理器（最高主频可达到 667MHz）。
（2）运行于 266MHz 的 DDR 内存，128MB。
（3）1MB NOR Flash。
（4）256MB NAND Flash。
（5）WM9714 AC97 声卡。
（6）VGA 输出接口（可达 1024×768@60Hz）。
（7）TV 输出接口。
（8）USB 2.0 OTG 接口及 USB 1.1 host 接口。

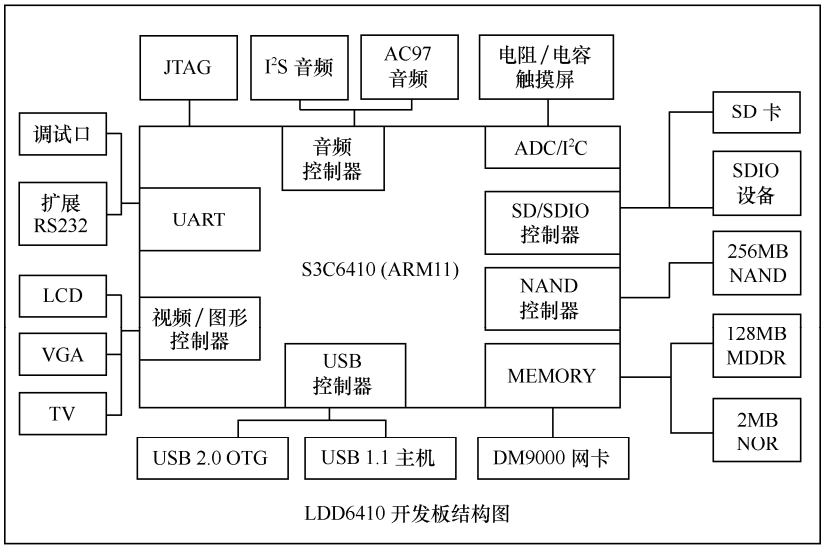

图 1.10　LDD6410 的结构图

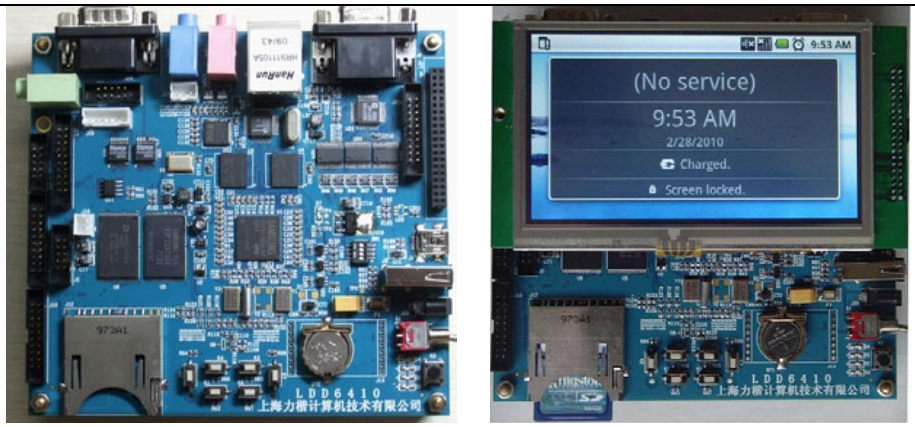

图 1.11　LDD6410 实物图

（9）SD/SDIO 接口，支持 SD 卡和 SDIO 设备。

（10）DM9000 百兆网卡。

（11）4.3 寸 LCD（分辨率为 480×272）、触摸屏。

（12）S3C6410 芯片内嵌图形加速，JPEG、多媒体编解码。

（13）6 个 GPIO 按键。

（14）可扩展 Camera、WiFi、3G modem 等模块。

（15）可扩展外部矩阵键盘。

配套电路板提供了如下软件。

（1）工具链：提供了 arm-linux-gcc、arm-linux-gdb、gdbserver、strace 用于 Android 开发的 eclipse（带 ADT 插件）、JDK 和 NDK。

（2）U-BOOT：U-BOOT 源代码包含独立的 LDD6410 文件，支持从 SD 卡、NAND 启动，支持 DM9000 网卡引导。

（3）Linux 内核、BSP 和驱动：Linux 2.6.28 内核、源代码，包含独立的 LDD6410 BSP 和完整的设备驱动。

（4）文件系统：基于新版 Busybox 1.15.1，文件系统集成 jpegview、mplayer、appweb 等大量应用，集成了按键、鼠标、触摸屏、LCD 等测试程序，作为驱动的用户应用案例。

（5）Android：提供 Android 源代码和文件系统、内核电源管理补丁源代码、内核 Android 驱动源代码。LDD6410 的 Android 系统支持按键、触摸屏和鼠标操作，支持使用 LCD 和 VGA 进行显示。

（6）QT：LDD6410 支持 Qt/Embedded 4.5.3，移植了 Ts_lib 和 Tslib, ts_calibration，支持使用触摸屏进行操作。

LDD6410 支持从 SD 卡或 NAND 启动，通过电路板上的 SW1 可设置 LDD6410 的启动模式。从 SD 卡启动设备为全 ON；从 NAND 启动时，将 1、2 设置为 ON，3、4 设置为 OFF。

LDD6410 开发板的详细使用方法，请见配套光盘中的“LDD6410 开发板用户手册”。

1.5.3　工具链安装

本书配套光盘的虚拟机映像中已经安装好了 LDD6410 的工具链，读者如果想在其他环境中安装，只需要从 http://ldd6410.googlecode.com/files/cross-4.2.2-eabi.tar.bz2 下载。LDD6410 开发板工具链为 S3C6410X-ToolChain4.2.2-EABI-V0.0-cross-4.2.2-eabi.tar。安装步骤如下。

（1）解压上述工具链获得文件夹：4.2.2-eabi/。

（2）在/usr/local/下面创建目录 arm/（注意，最好是放到这个目录，不然在以后的编译过程中可能出现一些错误）。

（3）将目录 4.2.2-eabi/移动到/usr/local/arm/下面。

（4）设置环境变量。

编辑/etc/profile 文件，在文件末尾添加：

```
PATH="$PATH:/usr/local/arm/4.2.2-eabi/usr/bin"
export PATH
```

使环境变量生效，在终端输入命令：

```
source /etc/profile
```

另外，也可以通过修改 home 目录的.bashrc 来将/usr/local/arm/4.2.2-eabi/usr/bin 添加到 PATH：

```
export PATH=/usr/local/arm/4.2.2-eabi/usr/bin/:$PATH
```

（5）测试环境变量是否设置成功。

在终端输入：echo $PATH，如果输出的路径中包含了/usr/local/arm/4.2.2-eabi/usr/bin，则说明环境变量设置成功。

（6）测试交叉编译工具链。

在终端输入“arm-linux-gcc –v”，显示如下：

```
Using built-in specs.
Target: arm-unknown-linux-gnueabi
Configured with:
/home/scsuh/workplace/coffee/buildroot-20071011/toolchain_build_arm
/gcc-4.2.2/configure --prefix=/usr --build=i386-pc-linux-gnu --host=i386-pc-linux-gnu
--target=arm-unknown-linux-gnueabi --enable-languages=c,c++ --with-sysroot=/usr/local
/arm/4.2.2-eabi/ --with-build-time-tools=/usr/local/arm/4.2.2-eabi//usr/arm-unknown-linux-
gnueabi/bin --disable-cxa_atexit --enable-target-optspace --with-gnu-ld --enable-shared
--with-gmp=/usr/local/arm/4.2.2-eabi/gmp --with-mpfr=/usr/local/arm/4.2.2-eabi//mpfr
--disable-nls --enable-threads --disable-multilib --disable-largefile --with-arch=armv4t
--with-float=soft --enable-cxx-flags=-msoft-float
Thread model: posix gcc version 4.2.2
```

说明交叉编译工具链已经安装成功。

ldd6410-debug-tools.tar.gz 调试工具包包含了 strace、gdbserver 和 arm-linux-gdb，其中 strace、gdbserver 用于目标板文件系统，arm-linux-gdb 运行于主机端，对目标板上的内核、内核模块应用程序进行调试。

下载地址为 http://ldd6410.googlecode.com/files/ldd6410-debug-tools.tar.gz，光盘目录为 toolchains/ldd6410-debug-tools.tar.gz。

解压 ldd6410-debug-tools.tar.gz，将其中的 arm-linux-gdb 放入主机上 arm-linux-gcc 所在的目录/usr/local/arm/4.2.2-eabi/usr/bin/。

而 strace、gdbserver 则可根据需要放入目标机根文件系统的/usr/sbin 目录。

1.5.4 主机端 nfs 和 tftp 服务安装

本书配套光盘的虚拟机映像中已经安装好了 nfs 和 tftp，LDD6410 可使用 tftp 或 nfs 文件系统与主机通过网口交互。如果用户想在其他环境下自行安装，对于 Ubuntu 或 Debian 用户而言，在主机端可通过如下方法安装 tftp 服务：

```
sudo apt  get install tftpd  hpa
```

开启 tftp 服务：

```
sudo /etc/init.d/tftpdhpa start
Starting HPA's tftpd: in.tftpd.
```

对于 Ubuntu 或 Debian 用户而言，在主机端可通过如下方法安装 nfs 服务：

```
apt-get install nfs-kernel-server
sudo mkdir /home/nfs
sudo chmod 777 /home/nfs
```

运行“sudo vim /etc/exports”或“sudo gedit /etc/exports”，修改该文件内容为：

```
/home/nfs *(sync,rw)
```

运行 exportfs rv 开启 NFS 服务：

```
/etc/init.d/nfs-kernel-server restart
```

1.5.5 源代码阅读和编辑

源代码是学习 Linux 的最权威资料，在 Windows 上阅读 Linux 源代码的最佳工具是 Source Insight，在其中建立一个工程，并将 Linux 的所有源代码加入该工程，同步这个工程之后，我们将可以非常方便地在代码之间进行关联阅读，如图 1.12 所示。

网站 http://lxr.linux.no/提供了内核版本 2.6.11 到最新版 Linux 源代码的交叉索引，在其中输入 Linux 内核中的函数、数据结构或变量的名称就可以直接得到以超链接形式给出的定义和引用它的所有位置。还有一些网站也提供了 Linux 内核中函数、变量和数据结构的搜索能力，在 google 中搜索“linux identifier search”可得。

在 Linux 主机上阅读和编辑 Linux 源码的常用方式是 vim + cscope 或者 vim + ctags，vim 是一个文本编辑器，而 cscope 和 ctags 则可建立代码索引，建议读者尽快使用基于文本界面全键盘操作的 vim 编辑器，如图 1.13 所示。

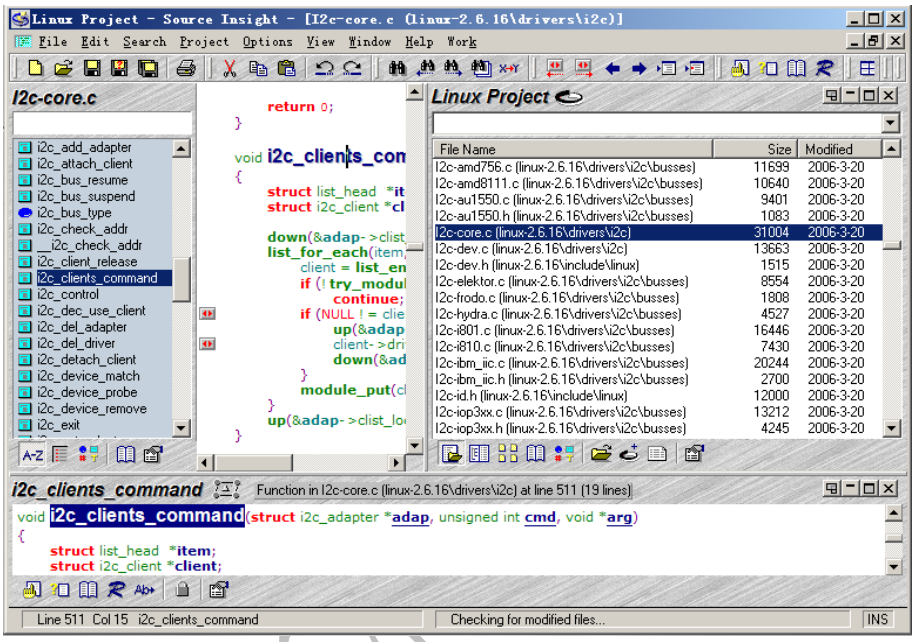

图 1.12　在 Source Insight 中阅读 Linux 源代码

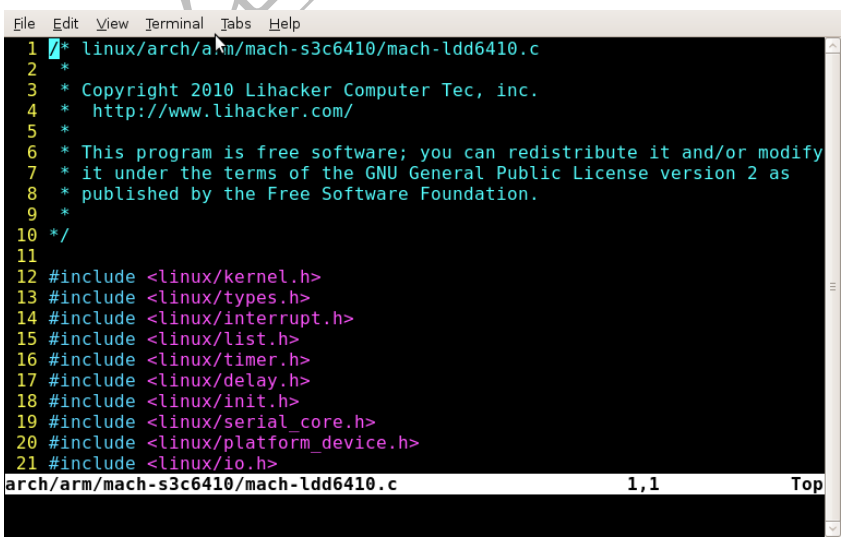

图 1.13　vim 编辑器

1.6 设备驱动 Hello World：LED 驱动

1.6.1 无操作系统时的 LED 驱动

在嵌入式系统的设计中，LED 一般直接由 CPU 的 GPIO（通用可编程 I/O 口）控制。GPIO 一般由两组寄存器控制，即一组控制寄存器和一组数据寄存器。控制寄存器可设置 GPIO 口的工作方式为输入或是输出。当引脚被设置为输出时，向数据寄存器的对应位写入 1 和 0 会分别在引脚上产生高电平和低电平；当引脚设置为输入时，读取数据寄存器的对应位可获得引脚上的电平为高或低。

在本例子中，我们屏蔽具体 CPU 的差异，假设在 GPIO_REG_CTRL 物理地址处的控制寄存器处的第 *n* 位写入 1 可设置 GPIO 为输出，在地址 GPIO_REG_DATA 物理地址处的数据寄存器的第 *n* 位写入 1 或 0 可在引脚上产生高或低电平，则无操作系统的情况下，设备驱动为代码清单 1.3。

代码清单 1.3 无操作系统时的 LED 驱动

```
1  #define reg_gpio_ctrl *(volatile int *)(ToVirtual(GPIO_REG_CTRL))
2  #define reg_gpio_data *(volatile int *)(ToVirtual(GPIO_REG_DATA))
3  /*初始化 LED*/
4  void LightInit(void)
5  {
6    reg_gpio_ctrl |= (1 << n); /*设置 GPIO 为输出*/
7  }
8
9  /*点亮 LED*/
10 void LightOn(void)
11 {
12   reg_gpio_data |= (1 << n); /*在 GPIO 上输出高电平*/
13 }
14
15 /*熄灭 LED*/
16 void LightOff(void)
17 {
18   reg_gpio_data &= ~(1 << n); /*在 GPIO 上输出低电平*/
19 }
```

上述程序中的 LightInit()、LightOn()、LightOff()都直接作为驱动提供给应用程序的外部接口函数。程序中 ToVirtual()的作用是当系统启动了硬件 MMU 之后，根据物理地址和虚拟地址的映射关系，将寄存器的物理地址转化为虚拟地址。

1.6.2 Linux 下的 LED 驱动

在 Linux 下，可以使用字符设备驱动的框架来编写对应于代码清单 1.3 的 LED 设备驱动（这里仅仅是为了讲解的方便，到后文我们会发现，内核中实际实现了一个提供 sysfs 结点的 GPIO LED 驱动，位于 drivers/leds/leds-gpio.c），操作硬件的 LightInit()、LightOn()、LightOff()函数仍然需要，但是，遵循 Linux 编程的命名习惯，重新将其命名为 light_init()、light_on()、light_off()。这些函数将被 LED 设备驱动中独立于设备的针对内核的接口进行调用，代码清单 1.4 给出了 Linux 下 LED 的驱动，此时读者并不需要能读懂这些代码。

代码清单 1.4 Linux 操作系统下 LED 的驱动

```
1  #include .../*包含内核中的多个头文件*/

2  /*设备结构体*/
3  struct light_dev {
4      struct cdev cdev; /*字符设备 cdev 结构体*/
5      unsigned char vaule; /*LED 亮时为 1，熄灭时为 0，用户可读写此值*/
6  };

7  struct light_dev *light_devp;
8  int light_major = LIGHT_MAJOR;
```

```
 9 MODULE_AUTHOR("Barry Song <21cnbao@gmail.com>");
10 MODULE_LICENSE("Dual BSD/GPL");
11 /*打开和关闭函数*/
12 int light_open(struct inode *inode, struct file *filp)
13 {
14     struct light_dev *dev;
15     /* 获得设备结构体指针 */
16     dev = container_of(inode->i_cdev, struct light_dev, cdev);
17     /* 让设备结构体作为设备的私有信息 */
18     filp->private_data = dev;
19     return 0;
20 }

21 int light_release(struct inode *inode, struct file *filp)
22 {
23     return 0;
24 }

25 /*读写设备:可以不需要 */
26 ssize_t light_read(struct file *filp, char __user *buf, size_t count,
27     loff_t *f_pos)
28 {
29     struct light_dev *dev = filp->private_data; /*获得设备结构体 */
30     if (copy_to_user(buf, &(dev->value), 1))
31         return  -EFAULT;

32     return 1;
33 }

34 ssize_t light_write(struct file *filp, const char __user *buf, size_t count,
35     loff_t *f_pos)
36 {
37     struct light_dev *dev = filp->private_data;

38     if (copy_from_user(&(dev->value), buf, 1))
39         return  -EFAULT;

40     /*根据写入的值点亮和熄灭 LED*/
41     if (dev->value == 1)
42         light_on();
43     else
44         light_off();

45     return 1;
46 }

47 /* ioctl 函数 */
48 int light_ioctl(struct inode *inode, struct file *filp, unsigned int cmd,
49     unsigned long arg)
50 {
51     struct light_dev *dev = filp->private_data;
```

```
52      switch (cmd) {
53      case LIGHT_ON:
54          dev->value = 1;
55          light_on();
56          break;
57      case LIGHT_OFF:
58          dev->value = 0;
59          light_off();
60          break;
61      default:
62          /* 不能支持的命令 */
63          return  -ENOTTY;
64      }

65      return 0;
66  }

67  struct file_operations light_fops = {
68      .owner = THIS_MODULE,
69      .read = light_read,
70      .write = light_write,
71      .ioctl = light_ioctl,
72      .open = light_open,
73      .release = light_release,
74  };

75  /*设置字符设备 cdev 结构体*/
76  static void light_setup_cdev(struct light_dev *dev, int index)
77  {
78      int err, devno = MKDEV(light_major, index);
79      cdev_init(&dev->cdev, &light_fops);
80      dev->cdev.owner = THIS_MODULE;
81      dev->cdev.ops = &light_fops;
82      err = cdev_add(&dev->cdev, devno, 1);
83      if (err)
84          printk(KERN_NOTICE "Error %d adding LED%d", err, index);
85  }

86  /*模块加载函数*/
87  int light_init(void)
88  {
89      int result;
90      dev_t dev = MKDEV(light_major, 0);
91      /* 申请字符设备号*/
92      if (light_major)
93          result = register_chrdev_region(dev, 1, "LED");
94      else {
95          result = alloc_chrdev_region(&dev, 0, 1, "LED");
96          light_major = MAJOR(dev);
97      }
98      if (result < 0)
```

```
 99          return result;

100      /* 分配设备结构体的内存 */
101      light_devp = kmalloc(sizeof(struct light_dev), GFP_KERNEL);
102      if (!light_devp) {
103          result =  -ENOMEM;
104          goto fail_malloc;
105      }
106      memset(light_devp, 0, sizeof(struct light_dev));
107      light_setup_cdev(light_devp, 0);
108      light_gpio_init();
109      return 0;

110 fail_malloc:
111      unregister_chrdev_region(dev, light_devp);
112      return result;
113 }
114 /*模块卸载函数*/
115 void light_cleanup(void)
116 {
117      cdev_del(&light_devp->cdev); /*删除字符设备结构体*/
118      kfree(light_devp); /*释放在 light_init 中分配的内存*/
119      unregister_chrdev_region(MKDEV(light_major, 0), 1); /*删除字符设备*/
120 }
121 module_init(light_init);
122 module_exit(light_cleanup);
```

上述代码的行数与代码清单 1.3 已经不能比拟，除了代码清单 1.3 中的硬件操作函数仍然需要外，代码清单 1.4 中还包含了大量对我们暂时陌生的元素，如结构体 file_operations、cdev，Linux 内核模块声明用的 MODULE_AUTHOR、MODULE_LICENSE、module_init、module_exit，以及用于字符设备注册、分配和注销用的函数 register_chrdev_region()、alloc_chrdev_region()、unregister_chrdev_region()等。我们也不能理解为什么驱动中要包含 light_init ()、light_cleanup ()、light_read()、light_write()等函数。

此时，我们只需要有一个感性认识，那就是，上述暂时陌生的元素都是 Linux 内核给字符设备定义的为实现驱动与内核接口而定义的。Linux 对各类设备的驱动都定义了类似的数据结构和函数。

1.7 全书结构

本书第 1 篇给您打下 Linux 设备驱动的基础。第 1 章简要地介绍了设备驱动的作用，并从无操作系统的设备驱动引出了 Linux 操作系统下的设备驱动，介绍了本书所基于的开发环境。第 2 章系统地讲解了一个 Linux 驱动工程师应该掌握的硬件知识，为工程师打下 Linux 驱动编程的硬件基础，讲解了各种类型的 CPU、存储器和常见的外设，并阐述了硬件时序分析方法和数据手册阅读方法。第 3 章将 Linux 设备驱动放在 Linux 2.6 内核背景中进行讲解，说明 Linux 内核的编程方法。由于驱动编程也在内核编程的范畴，因此，这一章实质是为编写 Linux 设备驱动打下软件基础。

第 2 篇讲解 Linux 设备驱动编程的基础理论、字符设备驱动及设备驱动设计中涉及的并发控制、同步等问题。第 4、5 章分别讲解 Linux 内核模块和 Linux 设备文件系统，第 6～9 章以虚拟设备 globalmem 和 globalfifo 为主线，逐步给其添加高级控制功能，第 10、11 章分别阐述 Linux 驱动编程中所涉及的中断和定时器、内核和 I/O 操作处理方法，本篇的第 12 章讲解了 Linux 设备驱动工程化的一些问题，属于承前启后的一章。

第 3 篇剖析复杂设备驱动的体系架构，每一章都给出了具体的实例。所涉及的设备包括块设备、终端设备、I^2C 适配器与 I^2C 设备、网络设备、PCI 设备、USB 设备、LCD 设备、Flash 设备等。这一部分的讲解方法是抽象与具体相结合，先以模板的形式给出各种设备驱动的设计，然后用具体实例设备的驱动填充对应的模板。

第 4 篇分析了 Linux 设备驱动的调试和移植方法。由于在 Linux 设备驱动的设计工作中人们强调多快好省，因此，如果能方便地把现有的其他平台中的驱动移植到 Linux 2.6 平台，或者将类似设备的驱动进行简单修改就运用于新的设备，那将会极大地缩短工程的实施时间。本书的最后几章对 Linux 设备驱动移植中涉及的理论以及移植的技巧进行了讲解。

联系方式

集团官网：www.hqyj.com　　嵌入式学院：www.embedu.org　　移动互联网学院：www.3g-edu.org

企业学院：www.farsight.com.cn　　物联网学院：www.topsight.cn　　研发中心：dev.hqyj.com

集团总部地址：北京市海淀区西三旗悦秀路北京明园大学校内 华清远见教育集团

北京地址：北京市海淀区西三旗悦秀路北京明园大学校区，电话：010-82600386/5

上海地址：上海市徐汇区漕溪路 250 号银海大厦 11 层 B 区，电话：021-54485127

深圳地址：深圳市龙华新区人民北路美丽 AAA 大厦 15 层，电话：0755-25590506

成都地址：成都市武侯区科华北路 99 号科华大厦 6 层，电话：028-85405115

南京地址：南京市白下区汉中路 185 号鸿运大厦 10 层，电话：025-86551900

武汉地址：武汉市工程大学卓刀泉校区科技孵化器大楼 8 层，电话：027-87804688

西安地址：西安市高新区高新一路 12 号创业大厦 D3 楼 5 层，电话：029-68785218

广州地址：广州市天河区中山大道 268 号天河广场 3 层，电话：020-28916067