第 3 章 CHAPTER 3 基于 Linux 物联网网关系统构建及开发

3.1 网关平台介绍

物联网网关的开发要基于特定的平台，本书中涉及的案例都是基于北京赛佰特科技有限公司开发的全功能物联网教学科研平台。该平台是基于物联网多功能、全方位教学科研需求，推出的一款集无线 ZigBee、IPv6、Bluetooth、WiFi、RFID 和智能传感器等通信模块于一体的全功能物联网教学科研平台，以强大的 Cortex-A9 嵌入式处理器作为核心智能终端，支持多种无线传感器通信模块组网方式，可支持 Linux/Android/Windows CE 操作系统。

全功能物联网教学科研平台的外观如图 3-1 所示。

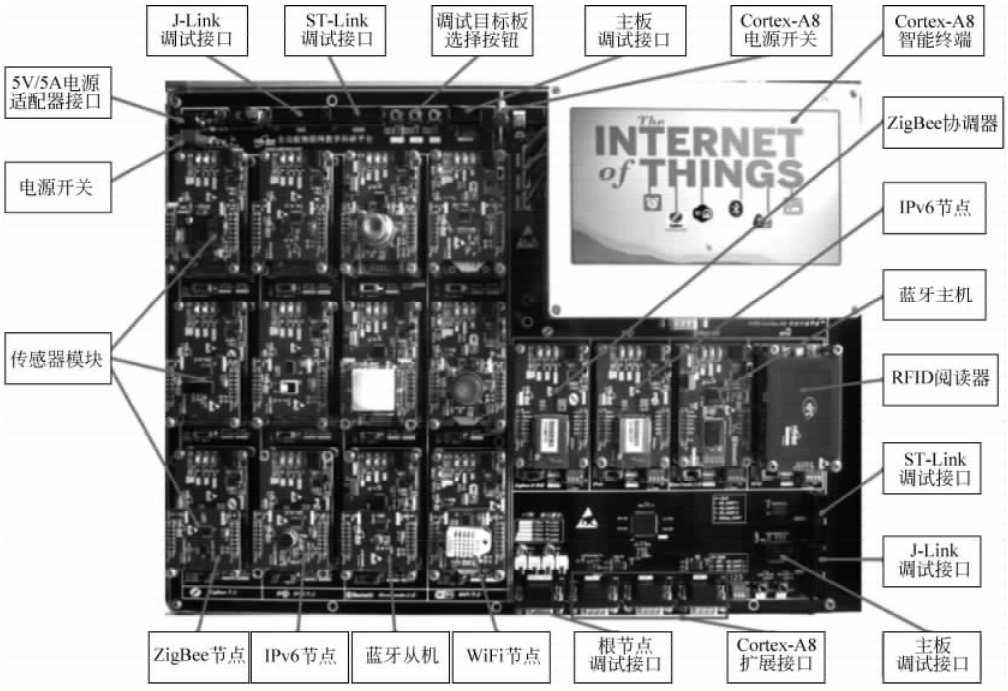

图 3-1　全功能物联网教学科研平台外观图

全功能物联网教学科研平台的应用结构拓扑图如图 3-2 所示。

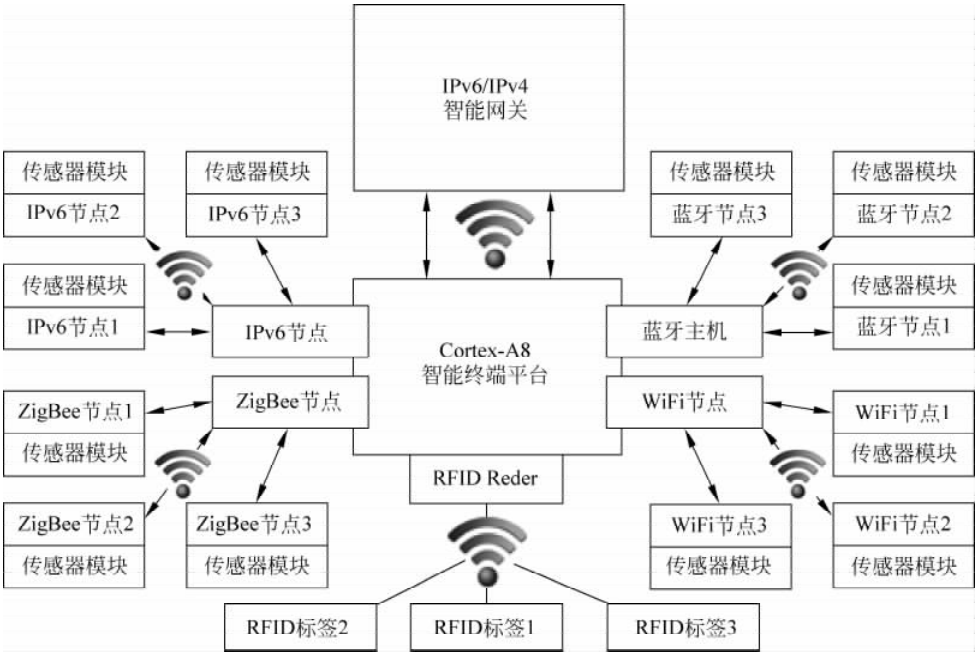

图 3-2　全功能物联网教学科研平台应用结构拓扑图

3.1.1　平台硬件资源

全功能物联网教学科研平台硬件由 Cortex-A9 智能终端、无线通信模块和智能传感器模块几部分构成。

智能终端硬件构成如表 3-1 所示。

表 3-1　Cortex-A9 智能终端组成

部件	性能指标参数
CPU 处理器	处理器 Samsung Exynos4412，基于 CortexM-A9，运行主频 1.5GHz
	内置 PowerVR SGX540 高性能图形引擎
	支持流畅的 2D/3D 图形加速
	最高可支持 1080p@30fps 硬件解码视频流畅播放，格式可为 MPEG4、H.263、H.264 等
	最高可支持 1080p@30fps 硬件编码(Mpeg-2/VC1)视频输入
显示	7 寸 LCD 液晶电阻触摸屏
接口	1 路 HDMI 输出
	4 路串口，RS232×2、TTL 电平×4
	USB Host 2.0、mini USB Slave 2.0 接口
	3.5mm 立体声音频(WM8960 专业音频芯片)输出接口、板载麦克风
	1 路标准 SD 卡座
	10/100M 自适应 DM9000AEP 以太网 RJ45 接口

续表

部件	性能指标参数
接口	SDIO 接口
	CMOS 摄像头接口
	AD 接口×6，其中 AIN0 外接可调电阻，用于测试
	I2C-EEPROM 芯片(256byte)，主要用于测试 I2C 总线
	用户按键(中断式资源引脚) ×8
	PWM 控制蜂鸣器
	板载实时时钟备份电池
电源	电源适配器 5V(支持睡眠唤醒)

无线通信模块资源如表 3-2 所示。

表 3-2 无线通信模块组成

节点类型	性能指标参数
ZigBee 节点	处理器 CC2530，内置增强型 8 位 51 单片机和 RF 收发器，符合 IEEE 802.15.4/ZigBee 标准规范，频段范围为 2045～2483.5MHz，板载天线
	存储器：256KB 闪存和 8KB RAM
	射频数据速率：250Kbps，可编程的输出功率高达 4.5dB
	用户自定制：按键×2，LED×2
	供电电压：2～3.6V，支持电池供电
	扩展调试接口
IPv6 节点	处理器 STM32W108，基于 ARM Cortex-M3 高性能的微处理器，集成了 2.4GHz IEEE 802.15.4 射频收发器，板载天线
	存储器：128KB 闪存和 8KB RAM
	用户自定制：按键×1，LED×2
	供电电压：3.7V，收发电流：27mA/40mA，支持电池供电
	扩展 J-Link 调试接口
蓝牙节点	BF-10 蓝牙模块，BlueCore4-Ext 芯片，板载天线
	处理器 STM32F103 基于 ARM Cortex-M3 内核，主频 72MHz
	完全兼容蓝牙 2.0 规范，硬件支持数据和语音传输，最高可支持 3M 调制模式
	支持 UART 透传，IO 配置
	扩展 J-Link 接口，外设主从开关，支持一键主从模式转换
	支持电池供电
WiFi 节点	型号：嵌入式 WiFi 模块(支持 802.11b/g/n 无线标准)，内置板载天线
	处理器 STM32F103 基于 ARM Cortex-M3 内核，主频 72MHz
	支持多种网络协议：TCP/IP/UD，支持 UART/以太网数据通信接口
	支持 STA 和 AP 两种工作模式，支持路由和桥接两种网络架构
	支持透明协议数据传输模式，支持串口 AT 指令
	扩展 J-Link 接口
	支持电池供电

续表

节点类型	性能指标参数
RFID 阅读器	MF RC531(高集成非接触读写卡芯片)支持 ISO/IEC 14443A/B 和 MIFARE 经典协议
	处理器 STM8S105 高性能 8 位架构的微控制器,主频 16MHz
	支持 mifare1 S50 等多种卡类型
	用户自定制：按键×1,LED×1
	最大工作距离：100mm,最高波特率：424Kb/s
	Crypto1 加密算法并含有安全的非易失性内部密钥存储器
	扩展 ST-Link 接口

传感器模块资源如表 3-3 所示。

表 3-3　传感器模块组成

部件	性能指标参数	
处理器	STM8S103 高性能 8 位框架结构的微控制器,主频 1MHz	
外设	LED 灯、UART 串口及电源接口	
传感器种类	① 磁检测传感器	⑥ 三轴加速度传感器
	② 光照传感器	⑦ 声响检测传感器
	③ 红外对射传感器	⑧ 温湿度传感器
	④ 红外反射传感器	⑨ 烟雾传感器
	⑤ 结露传感器	⑩ 振动检测传感器

外扩辅助模块资源如表 3-4 所示。

表 3-4　外扩辅助模块组成

部件	性能指标参数
USB-UART 扩展板	核心芯片：FT232RL
	功能：连接 PC 与网络节点串口调试功能
	接口：VCC GND TXD GND RXD
电池模块	功能：锂电池供电,提供低电压报警,提示用户充电
	接口：3.7V
电池充电板	5V 电源适配器,双路锂电池充电
调试工具	ST-Link 仿真调试工具、J-Link 仿真调试工具

3.1.2　平台软件资源

全功能物联网教学科研平台有丰富的软件资源,具体如表 3-5 所示。

表 3-5　全功能物联网教学科研平台软件资源

模块	软 件 资 源
Cortex-A9 智能终端平台	操作系统：Linux-3.5 ＋ Qt4.7/Qtopia2/Qtopia4、Android 4.1.2 功能：可进行 Linux 系统嵌入式编程开发,包括开发环境搭建、Bootloader 开发、嵌入式操作系统移植、驱动程序调试与开发、应用程序的移植与项目开发等

续表

模块	软 件 资 源
IPv6 智能网关	操作系统：Openwrt，实现 IPv6 网络的全部功能、IPv4 到 IPv6 的自动转换 开发工具：Linux(RHEL6)、openwrt 源码包 功能：可进行 Linux 编程开发，包括 Linux 内核移植与裁剪、文件系统定制、驱动程序调试与开发、应用程序的移植与开发、交叉编译、Shell 编程、网络通信、防火墙技术、路由技术、Web 配置系统、数据库技术等
ZigBee 通信节点	开发环境：基于 IAR for 8051 协议：ZigBee PRO 协议(Z-Stack2007 协议栈) 功能：自动组网、自动路由、无线数据传输等
IPv6 通信节点	操作系统：Contiki 2.5 协议：基于 Contiki OS 在 802.15.4 平台上实现完整的 IPv6 协议(Contiki OS uIPv6 协议栈) 功能：自动组网、自动路由、无线数据传输等
蓝牙通信节点	协议：完整的蓝牙通信 2.0 协议 功能：蓝牙模块组网、SPP 蓝牙串行服务、无线数据传输等
WiFi 通信节点	网络类型：Station/AP 模式 安全机制：WEP/WAP-PSK/WAP2-PSK/WAPI 加密类型：WEP64/WEP128/TKIP/AES 工作模式：透明传输模式、协议传输模式 串口命令：AT＋命令结构 网络协议：TCP/UDP/ARP/ICMP/DHCP/DNS/HTTP 最大 TCP 连接数：32 功能：自动组网、支持 AP 模式/AT 命令、无线数据传输等
RFID 阅读器	功能：支持与节点通信、组网，支持快速 CRYPTO1 加密算法、IC 卡识别、IC 卡读写
传感器模块	功能：基于 IAR for STM8 的开发环境，实现传感器数据采集与串口协议通信

3.2 网关交叉编译环境

全功能物联网教学科研平台中的智能终端平台，在后续的物联网系统案例中充当网关的作用，该平台可支持 Linux 和 Android 操作系统。本节介绍在嵌入式操作系统 Linux 下开发应用程序的方法。首先需要明确交叉编译的概念。

3.2.1 交叉编译的概念

交叉编译，简单地说，就是在一个平台上生成另一个平台上的可执行代码。同一个体系结构可以运行不同的操作系统；同样，同一个操作系统也可以在不同的体系结构上运行。举例来说，我们常说的 x86 Linux 平台实际上是 Intel x86 体系结构和 Linux for x86 操作系统的简称；而 x86 WinNT 平台实际上是 Intel x86 体系结构和 Windows NT for x86 操作系统的简称。

一个经常会被问到的问题是，“既然已经有了主机编译器，为什么还要交叉编译呢?”其实答案很简单。有时是因为目标平台上不允许或不能够安装我们所需要的编译器，而我们又需要这个编译器的某些特征；有时是因为目标平台上的资源贫乏，无法运行我们所需要的编译器；有时又是因为目标平台还没有建立，连操作系统都没有，根本谈不上运行什么编译器。

交叉编译这个概念的出现和流行是和嵌入式系统的广泛发展同步的。我们常用的计算机软件，都需要通过编译的方式，把使用高级计算机语言编写的代码编译成计算机可以识别和执行的二进制代码。例如，在 Windows 平台上，可使用 Visual C++开发环境，编写程序并编译成可执行程序。在这种方式下，我们使用 PC 平台上的 Windows 工具开发针对 Windows 本身的可执行程序，这种编译过程称为本机编译。然而，在进行嵌入式系统的开发时，运行程序的目标平台通常具有有限的存储空间和运算能力，例如常见的 ARM 平台，其一般的静态存储空间大概是 16～32MB，而 CPU 的主频大概是 100～500MHz。在这种情况下，在 ARM 平台上进行本机编译就不太可能了，这是因为一般的编译工具链需要很大的存储空间，并需要很强的 CPU 运算能力。为了解决这个问题，交叉编译工具应运而生。通过交叉编译工具，就可以在 CPU 能力很强、存储空间足够的主机平台上编译出针对其他平台的可执行程序。

要进行交叉编译，需要在主机平台上安装对应的交叉编译工具链，然后用这个交叉编译工具链编译源代码，最终生成可在目标平台上运行的代码。

在此，我们将在 Linux PC 上，利用 arm-linux-gcc 编译器，编译出针对 Linux ARM 平台的可执行代码。

3.2.2　交叉编译环境的搭建

在做实际工作之前，首先介绍一些关于交叉编译的基本知识。

宿主机(host)：编辑和编译程序的平台，一般是基于 x86 的 PC，通常也被称为主机。

目标机(target)：用户开发的系统，通常都是非 x86 平台。host 编译得到的可执行代码在 target 上运行。

我们在主机平台上开发程序，并在这个平台上运行交叉编译器，编译程序；而由交叉编译器生成的程序将在目标平台上运行。平台描述的完整格式是：CPU-制造厂商-操作系统，如 sparc-sun-sunos4.1.4，说明平台所使用的 CPU 是 sparc，制造厂商是 Sun，上面运行的操作系统是 SunOS，版本是 4.1.4。也可以使用短格式，短格式中有选择地去除了制造厂商、软件版本等信息，因此可以用 sparc-sunos 或 sparc-sunos-sunos4 来描述这个平台。

对于交叉编译器，可以从网上下载，也可以自己生成，需要准备足够的磁盘空间和编辑器的源代码，然后配置一些信息，需要花费一定的时间来进行编辑。具体编译过程本书不再赘述，读者可参考网上资料自行实现。

下面介绍宿主机的配置。在宿主机上安装 VMware 虚拟机，然后安装 Linux 映像文件，具体安装过程读者自行完成。本书中介绍的 Linux 映像文件版本为 Red Hat Enterprise Linux 6，安装的交叉编译器版本为 arm-linux4.5.1。图 3-3 显示的是该版本交叉编译工具链包含的文件。

```
[root@localhost bin]# ls
arm-linux-addr2line  arm-none-linux-gnueabi-addr2line
arm-linux-ar         arm-none-linux-gnueabi-ar
arm-linux-as         arm-none-linux-gnueabi-as
arm-linux-c++        arm-none-linux-gnueabi-c++
arm-linux-cc         arm-none-linux-gnueabi-cc
arm-linux-c++filt    arm-none-linux-gnueabi-c++filt
arm-linux-cpp        arm-none-linux-gnueabi-cpp
arm-linux-g++        arm-none-linux-gnueabi-g++
arm-linux-gcc        arm-none-linux-gnueabi-gcc
arm-linux-gcc-4.5.1  arm-none-linux-gnueabi-gcc-4.5.1
arm-linux-gccbug     arm-none-linux-gnueabi-gccbug
arm-linux-gcov       arm-none-linux-gnueabi-gcov
arm-linux-gprof      arm-none-linux-gnueabi-gprof
arm-linux-ld         arm-none-linux-gnueabi-ld
arm-linux-ldd        arm-none-linux-gnueabi-ldd
arm-linux-nm         arm-none-linux-gnueabi-nm
arm-linux-objcopy    arm-none-linux-gnueabi-objcopy
arm-linux-objdump    arm-none-linux-gnueabi-objdump
arm-linux-populate   arm-none-linux-gnueabi-populate
arm-linux-ranlib     arm-none-linux-gnueabi-ranlib
arm-linux-readelf    arm-none-linux-gnueabi-readelf
arm-linux-size       arm-none-linux-gnueabi-size
arm-linux-strings    arm-none-linux-gnueabi-strings
arm-linux-strip      arm-none-linux-gnueabi-strip
```

图 3-3　arm-linux4.5.1 交叉编译工具链工具显示

在宿主机上通过交叉编译器来编译源程序，得到可执行程序后，在目标机上运行可执行程序。那么如何实现把宿主机上的文件共享到目标机上呢？可采用 TFTP、NFS 及 U 盘挂载等方式。在此介绍最常用的方法——NFS 共享和 TFTP。图 3-4 显示了宿主机与目标机的连接方式。

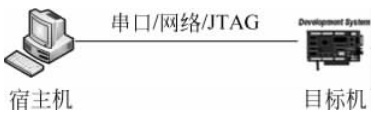

图 3-4　宿主机与目标机连接示意图

1. NFS 服务

网络中同为 Linux 或 UNIX 操作系统的主机可通过 NFS 服务实现文件的共享。

NFS 可以将远程文件系统载入到本地文件系统下。远程的硬盘、目录和光驱都可以变成本地主机目录树中的一个子目录。载入后与处理自己的文件系统一样使用即可。不仅方便，还节省了重复保存文件的空间、传输文件的时间及网络带宽。NFS 基于 C/S 体系结构，即服务器端和客户端。服务器端提供共享的文件系统，必须把文件系统输出(export)出去；客户端则要把文件系统载入到自己的系统下；使用 NFS，需要在服务器端设置输出，在客户端设置载入。

NFS 共享实现了将宿主机 RHEL6 系统的目录设置为共享目录，在 ARM Linux 系统中使用 mount 命令挂载的方式进行访问和执行目标程序。下面讲解该共享的设置和使用方法。

(1) 添加 NFS 共享目录并设置权限。

```
[root@localhost ~]# vi /etc/exports
```

修改内容如下：

```
/CBT - SuperIOT * (rw)
```

退出保存即可。该行语句表明，将系统的/CBT-SuperIOT 目录设置成共享，“*”代表

任意机器都可以访问，rw 表示具有读写权限。

（2）关闭系统防火墙。

```
[root@localhost ~]# /etc/init.d/iptables stop
iptables: 清除防火墙规则：  [确定]
iptables: 将链设置为政策 ACCEPT: filter      [确定]
iptables: 正在卸载模块：  [确定]
```

（3）启动 NFS 共享服务。

```
[root@localhost ~]# /etc/init.d/nfs restart
关闭 NFS mountd: [确定]
关闭 NFS 守护进程: [确定]
关闭 NFS quotas: [确定]
启动 NFS 服务: [确定]
关掉 NFS 配额: [确定]
启动 NFS 守护进程: [确定]
启动 NFS mountd: [确定]
```

（4）在 ARM Linux 系统中访问宿主机端 NFS 共享。

```
[root@Cyb-Bot /]# mount -t nfs -o nolock 192.168.1.7:/CBT-SuperIOT /mnt/nfs/
```

mount 命令在目标机上的 ARM Linux 系统中使用。挂载成功后，即可在 ARM 系统中访问远程宿主机端 NFS 共享目录了。

本书实验环境中 RHEL6 宿主机 IP 为 192.168.1.7，目标机 ARM Linux 系统默认 IP 地址为 192.168.1.230。

注意：在搭建 NFS 共享服务时，确保实验网络环境设置正确，如 RHEL6 宿主机的 IP 地址和 ARM Linux 系统的 IP 地址保持同一个网段，并使用网线连接好宿主机和目标机系统。

2. TFTP 服务

TFTP 服务主要是基于网络的文件传输，通常需要在宿主机 RHEL6 中安装 tftp-server，之后就可以使用 ARM Linux 系统的 tftp 命令从宿主机端下载文件了。以下是宿主机 RHEL6 下 tftp 软件的设置和使用方法。

（1）安装 tftp-server。

如果宿主机 Linux 系统没有安装 tftp-server 软件，则需要利用网络进行安装，安装前要确保宿主机系统可以上网，且 yum 仓库源也设置好。

```
[root@localhost ~]# yum install tftp-server
```

（2）配置 tftp。

修改/etc/xinetd.d/tftp 文件，更改 tftp 下载目录和开启服务。

```
service tftp
{
```

```
        socket_type = dgram
        protocol = udp
        wait = yes
        user = root
        server = /usr/sbin/in.tftpd
        server_args =- s /tftpboot      /* 更改默认下载目录为/tftpboot */
        disable = no                    /* 开启服务 */
        per_source = 11
        cps = 100 2
        flags = IPv4
    }
```

(3) 重启服务。

```
[root@localhost ~]# service xinetd restart
停止 xinetd:                                                [确定]
正在启动 xinetd:                                            [确定]
```

(4) 下载文件。

在宿主机端开启 tftp-server 服务后,将要下载的文件复制到/tftpboot 目录下,就可以用 ARM Linux 系统的 tftp 命令下载宿主机/tftpboot 目录下的文件了。例如:

```
[root@Cyb-Bot /]# tftp -r test.txt -g 192.168.1.7
```

上面的 tftp 下载命令是在目标机 Cortex-A9 智能终端的 ARM Linux 系统中使用的。

如何访问目标机系统呢? 首先宿主机要通过串口线连接目标机,在连接上串口线后,在系统提示下安装 USB 转串口的驱动程序,即可看到串口号,如图 3-5 所示。

图 3-5 USB 转串口设备名称显示

安装超级终端或 xshell 软件,打开 xshell,设置串口协议,如图 3-6 所示。

启动 xshell,目标机实验箱上电启动后,就可以操作目标机系统了,如图 3-7 所示。

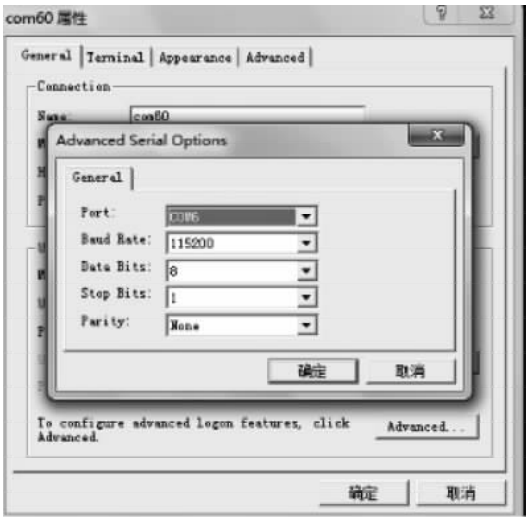

图 3-6　设置串口协议

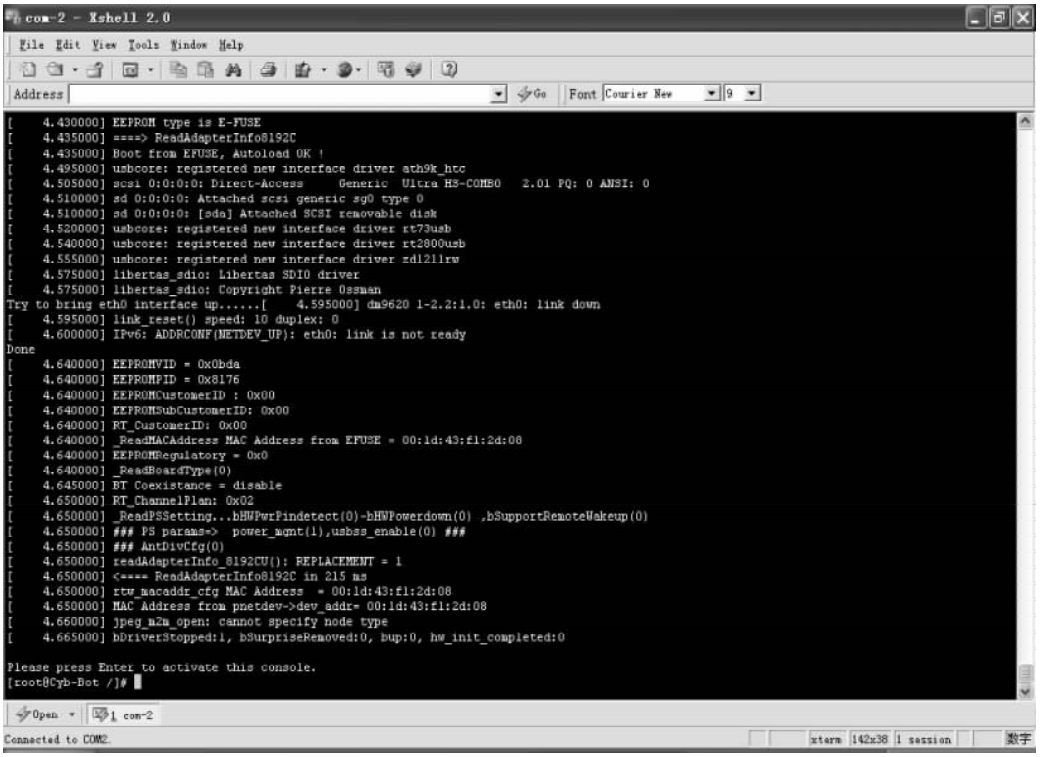

图 3-7　目标机 Linux 系统启动过程

3.3 GCC 编译器

GCC(GNU C Compiler)是 GNU 推出的功能强大、性能优越的多平台编译器,可以在多种硬体平台上编译出可执行程序,其执行效率与一般的编译器相比平均要高 20%～30%。GCC 支持 C、C++、Java 等多种编程语言。

GCC 将源代码文件生成可执行文件的过程分为 4 个相互关联的步骤。

(1) 预处理(也称预编译,Preprocessing):对头文件(include)、预编译语句(如 define 等)进行分析[预处理器 cpp]。

(2) 编译(Compilation):将预处理后的文件转换成汇编语言,生成.s 文件[编译器 ccl]。

(3) 汇编(Assembly):由汇编变为目标代码,生成.o 文件[汇编器 as]。

(4) 链接(Linking):连接目标代码,生成可执行程序[链接器 ld]。

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、预编译语句等进行分析,这个阶段根据输入文件生成以.i 为后缀的文件。接着调用 ccl 进行编译,生成以.s 为后缀的文件。汇编过程调用 as 进行工作,将以.s 为后缀的汇编语言文件经过汇编之后生成以.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 来完成最后的关键性工作,这个阶段就是链接。在链接阶段,所有的目标文件都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库中连到合适的地方。

在使用 GCC 编译器时,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,这里只介绍其中最基本、最常用的参数。具体可参考 GCC 使用手册。

GCC 最基本的用法是:

```
gcc [options] [filenames]
```

其中,options 是编译的参数;filenames 是相关的文件名称。

-c:只编译,不链接生成可执行文件,编译器只是由输入的.c 等源代码文件生成以.o 为后缀的目标文件,通常用于编译不包含主程序的子程序文件。

-o output_filename:确定输出文件的名称为 output_filename,同时这个名称不能和源文件同名。如果不给出这个选项,gcc 就给出预设的可执行文件 a.out。

-E:对源代码进行预编译。

-S:此编译选项告诉 gcc 产生了汇编语言文件后停止编译。

-g:如需对源代码进行调试,就必须加入这个选项。

-O:对程序进行优化编译、连接。采用这个选项,整个源代码会在编译、连接过程中进行优化处理,这样产生的可执行文件的执行效率可以提高,但是,编译、连接的速度就相应地要慢一些。

-O2：比-O 更好地优化编译、连接，整个编译、连接过程会更慢。

-l 库文件：编译时加载库文件。

gcc 一般使用默认路径/usr/include、/usr/lib 查找头文件和库文件。如果文件所用的头文件或库文件不在缺省目录下，则编译时要指定它们的查找路径。

-I 选项：指定头文件的搜索目录。

例如：gcc - I /home/export - o test1 test1.c

-L 选项：指定库文件的搜索目录。

例如：gcc - L /usr/X11/R6/lib - o test1 test1.c

gcc 所遵循的部分约定规则如下：

- .c 为后缀的文件，是 C 语言源代码文件；
- .h 为后缀的文件，是程序所包含的头文件；
- .i 为后缀的文件，是已经过预处理的 C 原始程序；
- 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。

3.4　Make 工具

GNU Make 是程序自动维护工具。在大型的开发项目中，通常有几十到上百个源文件，如果每次均手工输入 gcc 命令进行编译，会非常不方便。因此，通常利用 make 工具自动完成编译工作。这些工作包括：

(1) 如果仅修改了某几个源文件，则只需要重新编译这几个源文件。

(2) 如果某个头文件被修改了，则重新编译所有包含该头文件的源文件。

GNU Make 的主要工作是读 Makefile 文件。GNU Make 工具要依靠一个 Makefile 文件，Makefile 文件告诉 make 命令如何编译和链接程序。Makefile 文件由符合基本规则的语句组成，每组语句包含目标、依赖文件、命令三部分，由命令执行依赖文件来实现目标。依赖文件中如果有一个以上的文件比目标文件新，命令就会被执行。

3.4.1　Makefile 文件基本结构

Makefile 由一系列规则组成，规则格式如下：

```
target : prerequisites          依赖关系
<Tab> command                   命令
```

其中 target：表示要创建的项目；通常是目的文件和可执行文件；prerequisites：注明要创建的项目依赖于哪些文件；command：是创建每个项目时需要运行的命令。

注：命令前面需要是 Tab 键，而不是空格。

3.4.2 Makefile 实例

【例 3-1】 Makefile 文件实例。

```
myprog : foo.o bar.o
    gcc foo.o bar.o -o myprog
foo.o : foo.c foo.h
    gcc -c foo.c -o foo.o
bar.o : bar.c bar.h
    gcc -c bar.c -o bar.o
clean:
    rm *.o myprog
```

上面是 Makefile 的一个实例，管理的是一个包含 foo.c、bar.c、foo.h、bar.h 四个文件的项目。

第一行中 myprog 为目标，依赖于 foo.o 和 bar.o 文件。

foo.o 和 bar.o 文件又有各自的依赖规则。

Makefile 中一般都有 clean 规则，在重新编译之前删除以前生成的各个文件，此条规则没有依赖文件。使用 make 工具执行 Makefile 的命令为：

```
make
```

默认文件名为当前目录下的 makefile、Makefile 或 GNUmakefile，也可以使用命令行参数-f 指定文件名：

```
make -f  filename
```

如果没有“-f”参数，在 Linux 中，GNU Make 工具在当前工作目录中按照 GNUmakefile、Makefile、makefile 的顺序搜索 Makefile 文件。

通过命令行参数中的 target，可指定 make 要编译的目标，并且允许同时定义编译多个目标，操作时按照从左向右的顺序依次编译 target 选项中指定的目标文件。

如果命令行中没有指定目标，则系统默认 target 指向描述文件中的第一个目标文件。例如：

```
make
make clean
```

为简化 Makefile 文件的编写和编辑，Makefile 中可以使用环境变量。环境变量可以表示以下几种信息：

(1) 存储文件名列表。

(2) 存储可执行文件名。

(3) 存储编译器标识。

(4) 存储参数列表。

设置了环境变量后,前文介绍的 Makefile 文件例子简化成如下形式:

```
OBJS = foo.o bar.o
CC = gcc
CFLAGS =- Wall -O -g
EXEC = myprog
$(EXEC): $(OBJS)
    $(CC) $(OBJS) -o $(EXEC)
foo.o:foo.c foo.h
    $(CC) $(CFLAGS) -c foo.c -o foo.o
bar.o:bar.c bar.h
    $(CC) $(CFLAGS) -c bar.c -o bar.o
```

在 Makefile 文件中也可以使用以下内部变量。

(1) $@:扩展成当前规则的目标文件名。

(2) $<:扩展成依赖列表中的第一个依赖文件。

(3) $^:扩展成整个依赖列表的所有文件。

上述的 Makefile 文件使用内部变量后,简化的文件如下所示:

```
OBJS = foo.o bar.o
CC = gcc
CFLAG =- Wall -O -g
myprog: $(OBJS)
    $(CC) $^ -o $@
foo.o:foo.c foo.h
    $(CC) $(CFLAG) - c $< -o $@
bar.o:bar.c bar.h
    $(CC) $(CFLAG) - c $< -o $@
```

对于不同的项目,在使用 Makefile 文件进行项目管理时各不相同,有为单个文件编写 Makefile,有为多个文件编写 Makefile,有为不同目录文件编写 Makefile,有为多个子模块编写 Makefile,读者可以根据各自的需求参考相关资料学习,本书不再赘述。

3.5　Linux 多线程编程

3.5.1　多线程概述

线程是进程的一条执行路径。每个线程共享其所附属的进程的所有资源,包括打开的文件、信号标识及动态分配的内存等。

线程是属于进程的,线程运行在进程空间内,同一进程所产生的线程共享同一物理内存空间,当进程退出时该进程所产生的线程都会被强制退出并清除。

为什么有了进程的概念后，还要再引入线程呢？使用多线程到底有哪些好处？什么系统应该选用多线程？

(1) 使用多线程的理由之一是和进程相比，它是一种非常节俭的多任务操作方式。在Linux系统下，启动一个新的进程必须分配给它独立的地址空间，建立众多的数据表来维护它的代码段、堆栈段和数据段，这是一种昂贵的多任务工作方式。而运行于一个进程中的多个线程，它们彼此之间使用相同的地址空间，共享大部分数据，启动一个线程所花费的空间远远小于启动一个进程所花费的空间，而且，线程间彼此切换所需的时间也远远小于进程间彼此切换所需要的时间。一个进程的开销大约是一个线程开销的30倍左右。

(2) 使用多线程的理由之二是线程间通信机制比较方便。对不同进程来说，它们具有独立的数据空间，要进行数据的传递只能通过通信的方式进行，这种方式不仅费时，而且很不方便。线程则不然，由于同一进程下的线程之间共享数据空间，所以一个线程的数据可以直接为其他线程所用，这不仅快捷，而且方便。当然，数据的共享也带来其他一些问题，有的变量不能同时被两个线程所修改，有的子程序中声明为static的数据更有可能给多线程程序带来灾难性的打击，这些正是编写多线程程序时最需要注意的地方。

除了以上所说的优点外，多线程程序作为一种多任务、并发的工作方式，还有以下的优点：

(1) 提高应用程序响应时间。这对图形界面的程序尤其有意义，当一个操作耗时很长时，整个系统都会等待这个操作，此时程序不会响应键盘、鼠标、菜单的操作，而使用多线程技术，将耗时长的操作置于一个新的线程，可以避免这种尴尬的情况。

(2) 使多CPU系统更加有效。操作系统会保证当线程数不大于CPU数目时，不同的线程运行于不同的CPU上。

(3) 改善程序结构。一个既长又复杂的进程可以考虑分为多个线程，成为几个独立或半独立的运行部分，这样的程序会利于理解和修改。

3.5.2 Linux多线程API

Linux系统的多线程遵循POSIX线程接口，称为pthread。编写Linux的多线程程序，需要使用头文件pthread.h，连接时需要使用库libpthread.a。LIBC中的pthread库提供了大量的API函数，为用户编写应用程序提供支持。下面对比较重要的函数做一些详细的说明。

1. 线程创建函数

```
int pthread_create (pthread_t * thread_id, __const pthread_attr_t * __attr,
                    void * ( * __start_routine) (void * ), void * __restrict __arg)
```

其中，第一个参数为指向线程标识符的指针；第二个参数用来设置线程属性；第三个参数是线程运行函数的起始地址；最后一个参数是运行函数的参数。当创建线程成功时，函数返回0，若不为0则说明创建线程失败。常见的错误返回代码为EAGAIN和EINVAL。前者表示系统限制创建新的线程，例如线程数目过多了；后者表示第二个参数

代表的线程属性值非法。创建线程成功后，新创建的线程运行第三个参数和第四个参数确定的函数，原来的线程则继续运行下一行代码。

2. 获得父进程 ID

```
pthread_t pthread_self (void)
```

3. 测试两个线程号是否相同

```
int pthread_equal (pthread_t __thread1, pthread_t __thread2)
```

4. 等待指定的线程结束

```
int pthread_join (pthread_t __th, void ** __thread_return)
```

其中，第一个参数为被等待的线程标识符；第二个参数为用户定义的指针，它可以用来存储被等待线程的返回值。这个函数是一个线程阻塞的函数，调用它的函数将一直等待到被等待的线程结束为止，当函数返回时，被等待线程的资源被收回。

5. 线程退出

```
void pthread_exit (void * __retval)
```

线程的结束方式有两种：一种是使用上面介绍的 pthread_join()函数，函数结束了，线程也就结束了；另一种是使用 pthread_exit()函数，该函数唯一的参数是线程退出以后的返回值。

6. 互斥量初始化

```
pthread_mutex_init (pthread_mutex_t * ,__const pthread_mutexattr_t * )
```

7. 销毁互斥量

```
int pthread_mutex_destroy (pthread_mutex_t * __mutex)
```

8. 再试一次获得对互斥量的锁定（非阻塞）

```
int pthread_mutex_trylock (pthread_mutex_t * __mutex)
```

9. 锁定互斥量（阻塞）

```
int pthread_mutex_lock (pthread_mutex_t * __mutex)
```

10. 解锁互斥量

```
int pthread_mutex_unlock (pthread_mutex_t * __mutex)
```

下面介绍有关条件变量的函数。使用互斥锁可实现线程间数据的共享和通信，互斥锁的一个明显的缺点是它只有两种状态：锁定和非锁定。而条件变量通过允许线程阻塞和等待另一个线程发送信号的方法弥补了互斥锁的不足，它常和互斥锁一起使用。使用时，条件变量被用来阻塞一个线程，当条件不满足时，线程往往解开相应的互斥锁并等待条件发生变化。一旦其他的某个线程改变了条件变量，它将通知相应的条件变量唤醒一个或多个正被此条件变量阻塞的线程。这些线程将重新锁定互斥锁并重新测试条件是否满足。一般来说，条件变量被用来进行线程间的同步。

11. 条件变量初始化

```
int pthread_cond_init (pthread_cond_t * __restrict __cond,
__const pthread_condattr_t * __restrict __cond_attr)
```

该函数用来初始化一个条件变量。其中，cond 是一个指向结构 pthread_cond_t 的指针；cond_attr 是一个指向结构 pthread_condattr_t 的指针。结构 pthread_condattr_t 是条件变量的属性结构，和互斥锁一样可以用来设置条件变量是进程内可用还是进程间可用，默认值是 PTHREAD_PROCESS_PRIVATE，即此条件变量被同一进程内的各个线程使用。

12. 销毁条件变量 COND

```
int pthread_cond_destroy (pthread_cond_t * __cond)
```

13. 唤醒线程等待条件变量

```
int pthread_cond_signal (pthread_cond_t * __cond)
```

该函数作用是发送条件变量 cond，以唤醒被阻塞在条件变量 cond 上的一个线程。多个线程阻塞在此条件变量上时，哪一个线程被唤醒是由线程的调度策略所决定的。要注意的是，必须用保护条件变量的互斥锁来保护这个函数，否则条件变量可能在测试条件和调用 pthread_cond_wait 函数之间被发出，从而造成无限制的等待。

14. 等待条件变量(阻塞)

```
int pthread_cond_wait (pthread_cond_t * __restrict __cond, pthread_mutex_t * __restrict __
mutex)
```

该函数使线程阻塞在一个条件变量上。线程解开 mutex 指向的锁，并被条件变量 cond 阻塞。线程可以被函数 pthread_cond_signal 和函数 pthread_cond_broadcast 唤醒，但是要注意的是，条件变量只是起阻塞和唤醒线程的作用，具体的判断条件还需用户给出。线程被唤醒后，它将重新判断条件是否满足，如果还不满足，一般来说线程仍阻塞在这里，等待被下一次唤醒。

15. 在指定的时间到达前等待条件变量

```
int pthread_cond_timedwait (pthread_cond_t * __restrict __cond,
pthread_mutex_t * __restrict __mutex,  __const struct timespec * __restrict __abstime)
```

该函数为另一个用来阻塞线程的函数。它比函数 pthread_cond_wait()多了一个时间参数，经历 abstime 段时间后，即使条件变量不满足，阻塞也被解除。

3.5.3　Linux 多线程例程

下面介绍 Linux 多线程编程的典型实例：生产者－消费者问题模型的实现。在主程序中分别启动生产者线程和消费者线程。生产者线程不断地顺序将 0～1000 的数字写入共享的循环缓冲区，同时消费者线程不断地从共享的循环缓冲区读取数据。生产者—消费者问题流程图如图 3-8 所示。

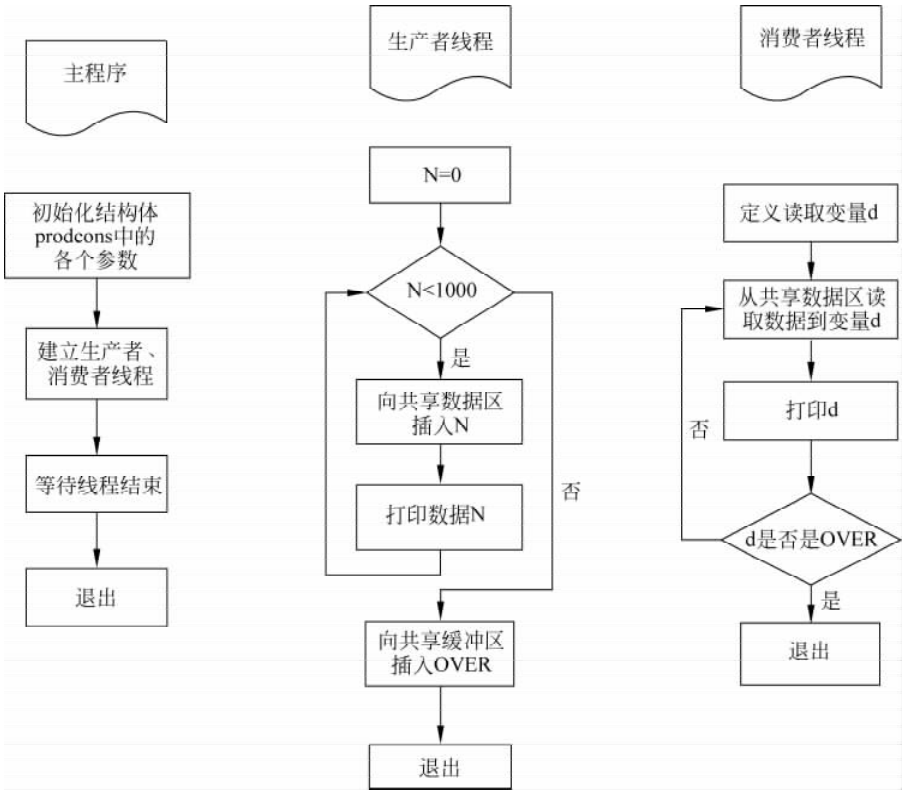

图 3-8　生产者—消费者实验源代码结构流程图

【例 3-2】 Linux 下生产者—消费者多线程应用：pthread.c。

```
#include <stdio.h>
#include <stdlib.h>
#include <time.h>
#include "pthread.h"
#define BUFFER_SIZE 16
/* 设置一个整数的圆形缓冲区 */
struct prodcons {
  int buffer[BUFFER_SIZE];        /* 缓冲区数组 */
  pthread_mutex_t lock;           /* 互斥锁 */
  int readpos, writepos;          /* 读写的位置 */
  pthread_cond_t notempty;        /* 缓冲区非空信号 */
  pthread_cond_t notfull;         /* 缓冲区非满信号 */
};
/* ---------------------------------------------------------- */
/* 初始化缓冲区 */
void init(struct prodcons * b)
{
  pthread_mutex_init(&b->lock, NULL);
  pthread_cond_init(&b->notempty, NULL);
  pthread_cond_init(&b->notfull, NULL);
  b->readpos = 0;
  b->writepos = 0;
}
/* ---------------------------------------------------------- */
#define OVER (-1)
struct prodcons buffer;
/* ---------------------------------------------------------- */
void * producer(void * data)
{
    int n;
    for (n = 0; n < 1000; n++) {
    printf(" put-->%d\n", n);
    put(&buffer, n);
    }
  put(&buffer, OVER);
  printf("producer stopped!\n");
  return NULL;
}
/* ---------------------------------------------------------- */
void * consumer(void * data)
{
  int d;
  while (1) {
    d = get(&buffer);
    if (d == OVER ) break;
    printf("%d-->get\n", d);
  }
  printf("consumer stopped!\n");
```

```
    return NULL;
}
/* ------------------------------------------------------------ */
int main(void)
{
    pthread_t th_a, th_b;
    void * retval;
    init(&buffer);
    pthread_create(&th_a, NULL, producer, 0);
    pthread_create(&th_b, NULL, consumer, 0);
  /* 等待生产者和消费者结束 */
    pthread_join(th_a, &retval);
    pthread_join(th_b, &retval);
    return 0;
}
```

生产者写入缓冲区和消费者从缓冲区读数的具体流程是：生产者首先获得互斥锁，并且判断写指针＋1 后是否等于读指针，如果相等则进入等待状态，等待条件变量 notfull；如果不等则向缓冲区中写一个整数，并且设置条件变量为 notempty，最后释放互斥锁。消费者线程与生产者线程类似，流程图如图 3-9 所示。

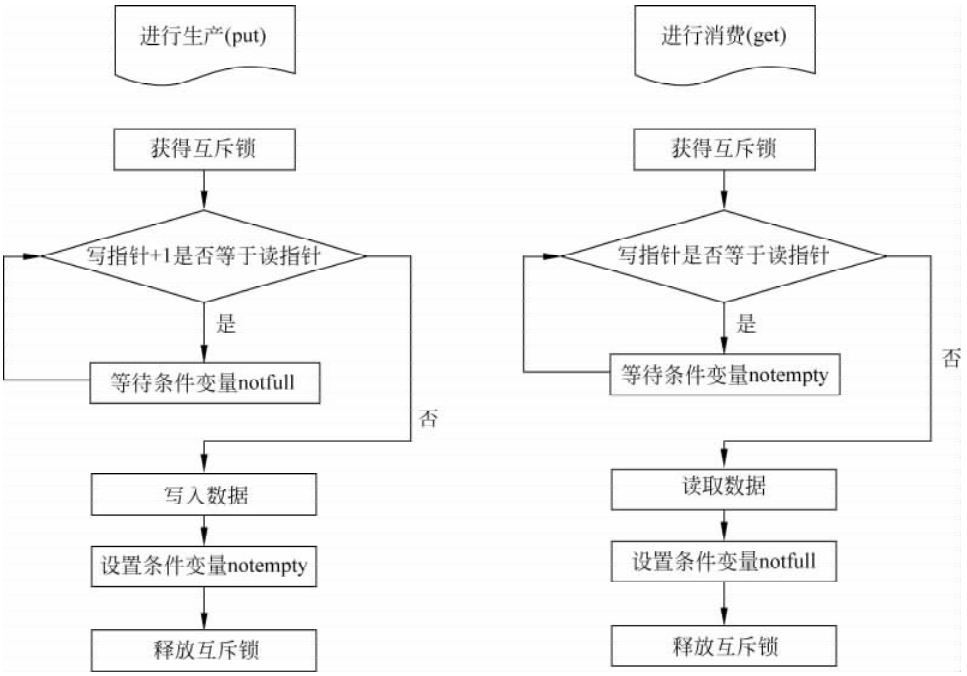

图 3-9　生产消费流程图

生产者写入共享的循环缓冲区函数 put，参考代码如下所示。

```
/* 向缓冲区中写入一个整数 */
void put(struct prodcons * b, int data)
```

```
{
        pthread_mutex_lock(&b->lock);          //获取互斥锁
    /*等待缓冲区非满*/
        while ((b->writepos+1) % BUFFER_SIZE==b->readpos) {
            printf("wait for not full\n");
        pthread_cond_wait(&b->notfull, &b->lock);
                                               //等待条件变量 b->notfull,不满则跳出阻塞
    }
        /*写数据并且指针前移*/
        b->buffer[b->writepos] = data;
        b->writepos++;
        if (b->writepos>=BUFFER_SIZE) b->writepos = 0;
        /*设置缓冲区非空信号*/
        pthread_cond_signal(&b->notempty);     //发送条件变量
        pthread_mutex_unlock(&b->lock);        //释放互斥锁
}
```

消费者读取共享的循环缓冲区函数 get 的参考代码如下所示。

```
/*从缓冲区中读出一个整数 */
int get(struct prodcons * b)
{
    int data;
    pthread_mutex_lock(&b->lock);              //获取互斥锁
    /* 等待缓冲区非空*/
    while (b->writepos==b->readpos) {          //如果读写位置相同
        printf("wait for not empty\n");
        pthread_cond_wait(&b->notempty, &b->lock);
                                  //等待条件变量 b->notempty,不空则跳出阻塞,否则无数据可读
    }
    /* 读数据并且指针前移 */
    data=b->buffer[b->readpos];
    b->readpos++;
    if (b->readpos>=BUFFER_SIZE) b->readpos = 0;
    /* 设置缓冲区非满信号*/
    pthread_cond_signal(&b->notfull);          //发送条件变量
    pthread_mutex_unlock(&b->lock);            //释放互斥锁
    return data;
}
```

运行结果参考如下：

```
[root@Cyb-Botpthread]# ./pthread
put-->0
put-->1
put-->2
put-->3
put-->4
```

```
put --> 5
…
wait for not full
0 --> get
1 --> get
2 --> get
3 --> get
4 --> get
5 --> get
…
wait for not empty
15 --> get
…
wait for not empty
put --> 20
20 --> get
…
```

3.6　Linux 串口编程

3.6.1　串口简介

串口是计算机的一种常用的接口，具有连接线少、通信简单的特点，因此得到广泛的应用。常用的串口是 RS-232-C 接口（又称 EIA RS-232-C），它是在 1970 年由美国电子工业协会（EIA）联合贝尔系统、调制解调器厂家及计算机终端生产厂家共同制定的用于串行通信的标准。串口通信指计算机以位（bit）为单位来传送数据，串行通讯的使用范围很广，是物联网及嵌入式系统开发过程中经常用到的通信方式之一。

串行 I/O 方式是将传输数据的每个字符一位接一位地传送，因此串行 I/O 可以减少信号连线，最少用一对线即可。同一根线上一连串的数字信号，首先要分割成位，再按位组成字符。在微型计算机中大量使用异步串行 I/O 方式，通信双方使用各自的时钟信号，而且允许时钟频率有一定误差，因此实现较容易。但是由于每个字符都要有独立的起始位和结束位，字符和字符间还有长度不定的空闲时间，因此效率相对较低。

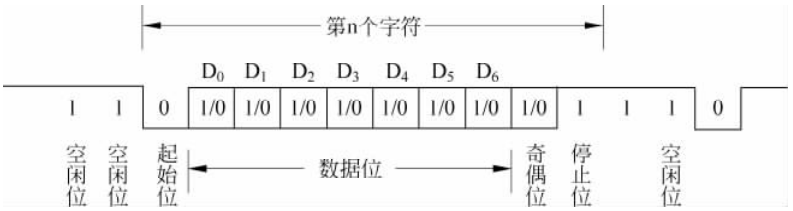

图 3-10　串行通信字符格式

图 3-10 给出异步串行通信中一个字符的传送格式。开始前，线路处于空闲状态，送出连续个“1”。传送开始时首先发一个“0”作为起始位，然后出现在通信线上的是字符的二进制编码数据。每个字符的数据位长可以约定为 5 位、6 位、7 位或 8 位，一般采用 ASCII 编码。后面是奇偶校验位，根据约定，用奇偶校验位将所传字符中为“1”的位数凑成奇数个或偶数个，也可以约定不要奇偶校验，这样就取消奇偶校验位。最后是表示停止位的“1”信号，这个停止位可以约定持续 1 位、1.5 位或 2 位的时间宽度。至此，一个字符传送完毕，线路又进入空闲，持续为“1”。异步串行通信中，常用的波特率为 50、95、110、150、300、600、1200、2400、4800、9600 等。

接收方按约定的格式接收数据，并进行检查，可以查出以下 3 种错误。

(1) 奇偶错：在约定奇偶检查的情况下，接收到的字符奇偶状态和约定不符。

(2) 帧格式错：一个字符从起始位到停止位的总位数不对。

(3) 溢出错：先接收的字符尚未被读取，后面的字符又传送过来，则产生溢出错。

每一种错误都会给出相应的出错信息，提示用户处理。一般串口调试都使用空的 Modem 连接电缆，其连接方式如图 3-11 所示。

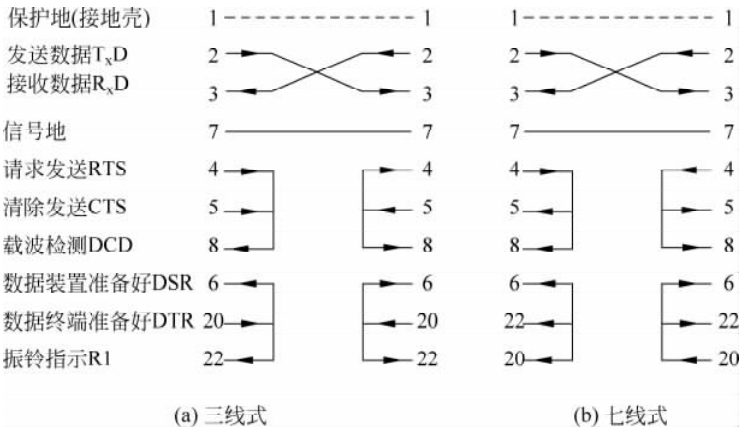

图 3-11 实用 RS-232C 通信连线

3.6.2 Linux 串口操作流程

Linux 对设备的访问是通过设备文件进行的，串口也是这样，为了访问串口，只需打开其设备文件即可。在 Linux 系统下，每一个串口设备都有设备文件与其关联，设备文件位于系统的/dev 目录下。本书所介绍的物联网平台的 Exynos4412 处理器自带 4 个串行端口控制器，系统的/dev/ttySAC0、/dev/ttySAC1 文件分别是串口 0 和串口 1 的设备文件。

1. 打开串口

打开串口是通过使用标准的文件打开函数 open()实现的。open()函数的原型为：

```
int open( const char * path, int flags)
```

其中

path 参数：串口设备文件路径。

flags 参数：打开文件的方式。

返回值：成功返回文件描述符，失败返回－1。

参考例子如下：

```
int fd;
/* 以读写方式打开串口 */
fd = open( "/dev/ttySAC0", O_RDWR);
if ( -1 == fd){
/*  不能打开串口一 */
perror(" 提示错误!");
}
```

这个例子中的 open()函数的第一个参数为串口设备节点文件；第二个参数是打开的属性，O_RDWR 表示以读写方式打开设备文件。

2. 设置串口

Linux 系统最基本的串口设置方式包括波特率设置，校验位设置和停止位设置。这些属性定义在结构体 struct termios 中。要在程序中使用该结构体，需要包含头文件 termios.h，该头文件定义了结构体 struct termios。termios 提供了一个常规的终端接口，用于控制非同步通信端口。这个结构至少包含以下成员。

```
struct termios
{
unsigned short c_iflag;              /* 输入模式标志 */
unsigned short c_oflag;              /* 输出模式标志 */
unsigned short c_cflag;              /* 控制模式标志 */
unsigned short c_lflag;              /* local mode flags */
unsigned char c_line;                /* line discipline */
unsigned char c_cc[NCC];             /* control characters */
};
```

1）波特率设置

```
struct termios Option;
tcgetattr(fd, &Option);
cfsetispeed(&Option,B19200);        /* 设置为 19200Bps */
cfsetospeed(&Option,B19200);
tcsetattr(fd,TCANOW,& Option);
```

2）校验位设置

校验位设置的参考代码如表 3-6 所示。

表 3-6 校验位设置的参考代码

校验方式	数据位数	设 置 代 码
无校验	8 位	Option. c_cflag &= ~PARENB; Option. c_cflag &= ~CSTOPB; Option. c_cflag &= ~CSIZE; Option. c_cflag |= ~CS8;
奇校验(Odd)	7 位	Option. c_cflag |= ~PARENB; Option. c_cflag &= ~PARODD; Option. c_cflag &= ~CSTOPB; Option. c_cflag &= ~CSIZE; Option. c_cflag |= ~CS7;
偶校验(Even)	7 位	Option. c_cflag &= ~PARENB; Option. c_cflag |= ~PARODD; Option. c_cflag &= ~CSTOPB;
Space 校验	7 位	Option. c_cflag &= ~PARENB; Option. c_cflag &= ~CSTOPB; Option. c_cflag &= &~CSIZE; Option. c_cflag |= CS8;

3) 停止位设置

停止位设置的参考代码如表 3-7 所示。

表 3-7 停止位设置的参考代码

停止位位数	设 置 代 码
1 位	Option. c_cflag &= ~CSTOPB
2 位	Option. c_cflag |= CSTOPB

3. 读写串口

设置串口属性之后,就可以读写串口进行数据收发了,把串口当作文件读写。

1) 发送数据

发送数据使用 write()函数,该函数的原型为:

```
int write( int fd, const void * buf, int nbytes )
```

其中:

fd 参数:文件描述符。

buf 参数:数据区首地址。

nbytes 参数:数据字节个数。

返回值:成功返回已发送字节数,失败返回-1。

功能:将 buf 缓冲区的 nbytes 个字节数据从 fd 标识的串口发送出去。

参考代码如下：

```
char buffer[1024];
int Length;
int nByte;
nByte = write(fd, buffer ,Length)
```

2）接收数据

接收数据使用 read()函数，该函数的原型为：

```
int read(int fd, void * buf, int nbytes )
```

其中：

fd 参数：文件描述符。

buf 参数：保存读取数据的缓冲区。

nbytes 参数：读取数据的字节数。

返回值：成功返回已读取字节数，失败返回－1。

功能：从 fd 标识的串口接收 nbytes 个字节数据存储至 buf 缓冲区。

参考代码如下：

```
char  buff[1024];
int    Len;
int  readByte = read(fd,buff,Len);
```

在操作串口数据时，除了基本的 read()和 write()函数实现数据收发外，还可以使用函数 fcntl()、select()等来实现异步读取。

4. 关闭串口

关闭串口就是关闭文件，使用 close()函数实现，该函数的原型为：

```
int close(int fd )
```

其中：

fd 参数：文件描述符。

返回值：成功返回 0，失败返回－1。

功能：关闭文件，释放文件描述符，使之可再利用。

参考代码如下：

```
close(fd)
```

3.6.3　Linux 串口操作实例

本节介绍一个通过串口实现双机通信的实例。通信的双机为宿主机和目标机。宿主机通过串口调试工具 AccessPort 实现数据的收发；目标机则在默认终端实现数据的收发。

该实例的功能为：目标机发送“1234567890”字符串到宿主机，并接收从宿主机发送过来的数据，并在默认终端实现数据的接收。该实例的参考代码如下。

【例 3-3】 Linux 串口通信：term.c。

```
#include <termios.h>
#include <stdio.h>
#include <unistd.h>
#include <stdlib.h>
#include <fcntl.h>
#include <sys/signal.h>
#include <pthread.h>

#define BAUDRATE B115200
#define COM1 "/dev/ttySAC0"
#define COM2 "/dev/ttySAC3"
volatile int fd;

void * receive(void * data)
{
    int c;
    printf("read modem\n");

    while (1)
    {
        read(fd,&c,1);                  /* com port */
        write(1,&c,1);                  /* stdout */
            if(c == '0')
                break;
    }

    printf("exit from reading modem\n");
    return NULL;
}

void * send(void * data)
{
    int c = '0';
    printf("send data\n");
    char buf0[11] = "1234567890";
    write(fd,buf0,11);
    return NULL; /* wait for child to die or it will become a zombie */
}

int main(int argc,char ** argv)
{
    struct termios oldtio,newtio;
    struct sigaction sa;
    int ok;
    pthread_t th_a, th_b;
```

```
    void * retval;
    fd = open(COM2, O_RDWR );
    if (fd < 0) {
        perror(COM1);
        exit( - 1);
    }
    printf("\nOpen COM Port Successfull\n");
    tcgetattr(fd,&oldtio);           /* save current modem settings */
    tcflush(fd,TCIOFLUSH);
    cfsetispeed(&newtio,BAUDRATE);
    cfsetospeed(&newtio,BAUDRATE);
    newtio.c_cflag &=~CSIZE;
    newtio.c_cflag |= CS8;
    newtio.c_cflag &=~PARENB; /* set the PARENB bit to zero -------- disable parity checked */
    newtio.c_iflag &=~ INPCK; /* set the INPCK bit to zero -------- INPCK means
inparitycheck(not paritychecked) */
    newtio.c_cflag &=~CSTOPB;
    newtio.c_cc[VMIN] = 1;
    newtio.c_cc[VTIME] = 0;

    tcflush(fd, TCIFLUSH);
    if(tcsetattr(fd,TCSANOW,&newtio)!= 0){
    perror("\n");
        return 0;
}

    pthread_create(&th_a, NULL, receive, 0);
    pthread_create(&th_b, NULL, send, 0);
    pthread_join(th_a, &retval);
    pthread_join(th_b, &retval);

    tcsetattr(fd,TCSANOW,&oldtio);   /* restore old modem setings */
    close(fd);
    exit(0);
}
```

在宿主机上用交叉编译器编译该源程序，生成可执行程序 term。

```
[root@localhost tty]# arm - linux - gcc - o term term.c - lpthread
```

通过 NFS 服务把可执行程序 term 加载到目标机上，并运行。

```
[root@localhost tty]#./term
```

目标机端运行结果如图 3-12 所示。

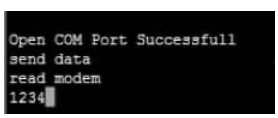

图 3-12　目标机端运行结果界面

宿主机端通过串口工具实现数据收发的显示结果如图 3-13 所示。

图 3-13 宿主机端串口收发结果界面

3.7 嵌入式数据库

在物联网网关及嵌入式系统中，经常需要存储一些数据。对于简单的数据，可以文件的形式进行存储。但应用程序需要对数据进行复杂的操作时，文件就不能满足要求了。而且物联网网关及嵌入式系统的硬件资源的有限性使得数据的存储空间有一定的局限性。如采用大型的关系型数据库在物联网网关及嵌入式系统中实现数据存储，会使系统变得庞大，影响系统的性能。这样一来，嵌入式数据库的优势就体现出来了。

3.7.1 嵌入式数据库的特点

嵌入式数据库是一种具备了基本数据库特性的数据文件，它与传统数据库的区别是：嵌入式数据库通常不需要独立运行数据库的引擎，而是和操作系统及其具体的应用集成在一起。只要在运行的操作系统平台上有相应的嵌入式数据库的驱动，应用程序直接调用相应的 API 去实现对数据的存取操作就可以了。而传统数据库则采用引擎响应方式驱动。嵌入式数据库的体积通常很小，这使得嵌入式数据库常常应用在移动设备上。由于性能卓越，所以在高性能的应用上也经常见到嵌入式数据库的身影。

嵌入式数据库具有以下特点。

（1）体积小。嵌入式数据库经过编译后也不过几十 KB，占用内存少，这使得它可以支持嵌入式 Linux、Windows CE、Palm OS 等多种嵌入式操作系统。

（2）可定制性。从目前嵌入式应用的发展趋势来看，嵌入式数据库的实现必须充分体现系统的可定制性，即系统选择的技术路线要面向具体的行业应用，因此，研究源码开放的

嵌入式数据库具有特殊意义。嵌入式数据库提供功能定制，根据其应用的环境来定制数据库的系统功能。

(3) 支持 SQL 查询语言。嵌入式数据库支持 SQL 查询语言，提供数据库及数据表的管理功能，能够方便地实现对数据的操作，不需要花很多的时间来重新学习嵌入式数据库。

(4) 提供接口函数。嵌入式数据库提供在高级语言中调用的接口函数，以方便地实现对数据库的操作及管理。

(5) 实时性。嵌入式系统的实时性，使得嵌入式数据库的操作应具有定时限制的特性。

(6) 底层控制能力。管理嵌入式数据时，要有一定的底层控制能力，因为嵌入式系统和底层的硬件环境密不可分，如对磁盘的操作等。底层控制的能力是决定数据库管理操作的关键。

(7) 标准化。随着专业化的发展，嵌入式数据库的功能越来越强大，也越来越专业化，将向标准化发展。市场的发展要求嵌入式数据库进一步规范。

3.7.2　SQLite 数据库

SQLite 是一种采用 C 语言开发的嵌入式数据库。SQLite 的目标是尽量简单，因此它抛弃了传统企业级数据库的种种复杂特性，只实现那些对于数据库而言非常必要的功能。尽管简单性是 SQLite 追求的首要目标，但是其功能和性能都非常出色。

SQLite 的源代码完全开放。SQLite 的第一个 Alpha 版本发布于 2000 年 5 月。SQLite 具有以下特性：

(1) 支持 ACID 事务。

(2) 零配置，即无须安装和管理配置。

(3) 是存储在单一磁盘文件中的一个完整的数据库。

(4) 数据库文件可以在不同字节顺序的机器间自由共享。

(5) 支持 2TB 的数据库。

(6) 程序体积足够小，全部 C 语言源码大致 3 万行，共 250KB。

(7) 比目前流行的大多数数据库对数据的操作要快。

(8) 提供了对事务功能和并发处理的支持，既保证了数据的完整性，又提高了运行速度。

(9) 程序独立运行，没有额外依赖。

SQLite 的 SQL 语言很大程度上实现了 ANSI SQL92 标准，特别是支持视图、触发器、事务，并支持 SQL 嵌套。它通过 SQL 编译器实现 SQL 语言对数据库的操作，支持大部分的 SQL 命令。

SQLite 的最大特点是其数据类型是无类型(typelessness)。无论表中每列声明的数据类型是什么，SQLite 并不做任何检查，可以将任何类型的数据保存到想要保存的任何列中。开发人员主要靠自己的程序控制输入与输出数据的类型。这里有一个特例，即当主键为整型值时，如果插入一个非整型值会产生异常。SQLite 允许忽略数据类型，但是仍然建议在创建表时指定数据类型，这有利于增强程序的可读性。目前，SQLite 的版本为 SQLite3。

3.7.3 SQLite3 的数据类型

3.7.2 节介绍了 SQLite 的字段是无数据类型的，在 SQLite 中，当将某个值插入数据库时，SQLite 将检查它的类型，如果该类型与关联的列不匹配，则 SQLite 会尝试将该值转换成列类型；如果不能转换，则该值将作为其本身具有的类型存储。但是有一种情况是例外的，即字段类型为 Integer Primary Key 时。

为了增加 SQLite 数据库和其他数据库的兼容性，SQLite 支持列的“类型亲和性”。列的“类型亲和性”是为该列所存储的数据建议一个类型。从理论上讲，任何列仍然是可以存储任何类型数据的，只是针对某些列，如果给出建议类型的话，数据库将按所建议的类型存储。这个被优先使用的数据类型被称为“亲和类型”。

SQLite3 支持 NULL、INTEGER、REAL、TEXT 和 BLOB 数据类型。数据库中的每一列都被定义为这几个亲和类型中的一种。

(1) NULL：表示该值为空值。

(2) INTERGER：表示值被标识为整数。

(3) REAL：表示值是浮动的数值，被存储为 8 字节浮动标记序号。

(4) TEXT：表示值为文本字符串，使用数据库编码存储。

(5) BLOB：表示值是 BLOB 数据，如何输入就如何存储，不改变格式。

3.7.4 SQLite3 的 API 函数

SQLite3 提供了 API 供 C/C++应用程序调用，以实现对 SQLite3 的操作。常用的 API 函数有以下几个。

1. 打开数据库

```
int sqlite3_open(
const char * filename,          /* 数据库的名字 */
sqlite3 ** ppDb                 /* 输出参数: SQLite 数据库句柄 */
);
```

该函数用来打开或者创建一个 SQLite3 数据库，如果在包含该函数的文件所在的路径下有同名的数据库(*.db)，则打开数据库；如果没有同名的数据库，则创建一个同名的数据库。如果打开或者创建数据库成功，则该函数返回 0 值，输出参数为 sqlite3 类型的变量。后续对该数据库的操作，通过该参数进行传递。

2. 关闭数据库

```
int sqlite3_close(sqlite3 * db);
```

当要结束对数据库的操作时，调用该函数来实现该数据库的关闭，该函数的一个参数是

成功打开数据库时输出参数 sqlite3 类型的变量。

3. 执行函数

```
int sqlite3_exec(
sqlite3 * ,                    /* 打开的数据库的名字 */
const char * sql,              /* 要执行的 SQL 语句 */
sqlite_callback,               /* 回调函数 */
void * ,                       /* 回调函数的参数 */
char ** errmsg                 /* 错误信息 */
);
```

在对数据库进行操作时，可以通过调用该函数来完成。sql 参数为具体操作数据库的 SQL 语句。在执行过程中，如果出现错误，相应错误信息可以存放在 errmsg 变量中。

4. 释放内存函数

```
void sqlite3_free(char * z);
```

在对数据库进行操作时，如果需要释放在中间过程中保存在内存中的数据（即清除内存空间），可以通过该函数来完成。

5. 显示错误信息

```
const char * sqlite3_errmsg(sqlite3 * );
```

通过 API 函数实现对数据库操作的过程中，出现的错误信息，可以通过该函数给出。

6. 获取结果集

```
int sqlite3_get_table(
  sqlite3 * ,                  /* 打开的数据库的名字 */
  const char * sql,            /* 要执行的 SQL 语句 */
  char *** resultp,            /* 结果集 */
  int * nrow,                  /* 结果集的行数 */
  int * ncolumn,               /* 结果集的列数 */
  char ** errmsg               /* 错误信息 */
);
```

在对数据库进行查询操作时，可以通过该函数来获取结果集。该函数的入口参数为查询的 SQL 语句。该函数的出口参数有：二维数据指针，指示查询结果的内容；结果集的行数和列数，行数为纯记录的条数，但是 resultp 数组中包含一行字段名的值。在具体操作时需要特殊关注。

7. 释放结果集

```
void sqlite3_free_table(char ** result);
```

释放 sqlite3_get_table()函数所分配的内存空间。

8. 声明 SQL 语句

```
int sqlite3_prepare(sqlite3 * , const char * , int, sqlite3_stmt ** , const char ** );
```

该函数把一条 SQL 语句编译成字节码留给后面的执行函数。使用该函数访问数据库是当前比较好的一种方法。

9. 销毁 SQL 声明

```
int sqlite3_finalize(sqlite3_stmt * );
```

该函数将销毁一个准备好的 SQL 声明。在数据库关闭之前,所有准备好的声明都必须被释放销毁。

10. 重置 SQL 声明

```
int sqlite3_reset(sqlite3_stmt * );
```

该函数用来重置一个 SQL 声明的状态,使得它可以被再次执行。

3.7.5 SQLite3 的应用

下面通过一个例子,介绍 SQLite3 中常用 API 函数的使用。

【例 3-4】 SQLite3API 函数的使用:sqlitetest.c。

```
#include<stdio.h>
#include<sqlite3.h>

int main()
{
    sqlite3 *db=NULL;
    int rc;
    char *Errormsg;
    int nrow;
    int ncol;
    char **Result;
    int i=0;

    rc=sqlite3_open("test.db",&db);
```

```
    if(rc){
        fprintf(stderr,"can't open database: % s\n", sqlite3_errmsg(db));
        sqlite3_close(db);
        return 1;
    }else
        printf("open database successly!\n");

    char * sql = "create table teacher (id integer primary key,name varchar(10))";
    sqlite3_exec(db,sql,0,0,&Errormsg);

    sql = "insert into teacher values(1,'sunjm')";
    sqlite3_exec(db,sql,0,0,&Errormsg);

    sql = "insert into teacher values(2,'zhangy')";
    sqlite3_exec(db,sql,0,0,&Errormsg);

    sql = "select * from teacher";
    sqlite3_get_table(db,sql,&Result,&nrow,&ncol,&Errormsg);

    printf("row = % d column = % d\n",nrow,ncol);
    printf("the result is:\n");
    for( i = 0;i <(nrow + 1) * ncol;i++)
        printf("Result[ % d] = % s\n",i,Result[i]);
    sqlite3_free(Errormsg);
    sqlite3_free_table(Result):
    sqlite3_close(db);
    return 0;
}
```

应用程序在编写完成之后,需要加载到目标机上运行,接下来介绍如何实现该应用程序在目标机上的运行过程。

1) 交叉编译 SQLite 源码

从官网上下载 SQLite3 的源码压缩包 sqlite-autoconf-3070500.tar.gz,解压后进行交叉编译,具体操作过程如下所示。

```
[root@localhost public]# tar -zxvf sqlite-autoconf-3070500.tar.gz
[root@localhost public]# cd sqlite-autoconf-3070500
[root@localhost sqlite-autoconf-3070500]# mkdir arm
[root@localhost sqlite-autoconf-3070500]# cd arm
[root@localhost arm]#
../configure --prefix=/home/public/sqlite-autoconf-3070500/arm/ --host=arm-linux
[root@localhost arm]#make
[root@localhost arm]#make install
```

安装之后会在/home/public/sqlite-autoconf-3070500/arm/目录下生成交叉编译的库文件和头文件。

2) 交叉编译源程序

如果没有将 include 下的文件和 lib 下的文件复制到/usr/lib 和/usr/include,则需要在编译时指定头文件和库文件所在的路径。含有 SQLite3 的 API 函数的应用程序,在链接时

需要加上-lsqlite3 参数。

```
arm-linux-gcc -o sqlitetest sqlitetest.c -lsqlite3
    -L /home/public/sqlite-autoconf-3070500/arm/lib
        -I /home/public/sqlite-autoconf-3070500/arm/include/
```

3）下载可执行程序到目标机并运行

配置 NFS 服务，进入目标机，将 lib 下的库文件和 include 下的头文件复制到/usr/lib 和/usr/include，并建立软链接：

```
ln -sf libsqlite3.so.0.8.6 libsqlite3.so
ln -sf libsqlite3.so.0.8.6 libsqlite3.so.0
```

运行结果如下：

```
[root@localhost sqlite3test]# ./sqlitetest
open database successly!
row = 2 column = 2
the result is:
Result[0] = id
Result[1] = name
Result[2] = 1
Result[3] = sunjm
Result[4] = 2
Result[5] = zhangy
```

习题 3

1. 简述交叉编译的概念。

2. 简述 GCC 的编译过程。

3. 如果在一个 C 语言的工程中含有 file1.c、file2.c、file3.c 三个文件，编写交叉编译该工程的 Makefile 文件，并运用环境变量，将交叉编译好的可执行程序下载到目标机运行。

4. 利用 Linux 串口编程实现双机通信。

5. 利用 SQLite3 的 API 函数实现学生成绩管理系统的基本功能。