

**Linux内核分析课程报告**

报告题目： Linux驱动程序设计及内核调试分析技巧

|  |  |
| --- | --- |
| 姓 名： | 涂 浩 新 |
| 学 号： | 31717019 |
| 专 业： | 软件工程 |
| 完成日期： | 2018.5 |

目 录

[1 Linux I2C总线驱动 2](#_Toc22862)

[1.1 I2C体系结构 3](#_Toc51)

[1.2 Linux I2C驱动源码分析 4](#_Toc27080)

[1.2.1 Linux I2C源码文件及目录说明 4](#_Toc32174)

[1.2.2 Linux I2C重要结构体及相互关系 5](#_Toc19905)

[1.2.3 代码层次调用图 8](#_Toc296)

[1.3 I2C设备驱动设计方法 9](#_Toc26433)

[1.3.1 Linux I2C设备驱动模块的加载与卸载 9](#_Toc12927)

[1.3.2 Linux I2C设备驱动的数据传输 10](#_Toc7384)

[1.3.3 Linux 的i2c-dev.c文件分析 11](#_Toc27521)

[1.4 小结 13](#_Toc16537)

[2 Linux内核调试环境搭建及调试技巧 14](#_Toc32652)

[2.1 搭建方法一：基于Qemu+Eclipse+Gdb 14](#_Toc17330)

[2.1.1 方案一：命令行界面QEMU+GDB 14](#_Toc308)

[2.1.2 方案二：图形界面QEMU+GDB+Eclipse 16](#_Toc17407)

[2.2 搭建方法二：基于Docker+Qemu 18](#_Toc32280)

[2.3 Linux内核调试技巧 21](#_Toc11026)

[2.3.1 printk 21](#_Toc10821)

[2.3.2 printk 23](#_Toc4730)

[3 Linux内核分析工具 23](#_Toc32735)

[3.1 系统分析工具 23](#_Toc32599)

[3.1.1 strace 23](#_Toc15592)

[3.2 内存分析工具 25](#_Toc22396)

[3.3 设备分析工具 25](#_Toc14423)

[3.4 文件分析工具 25](#_Toc17582)

[3.5 网络分析工具 25](#_Toc4326)

[3.6 安全分析工具 25](#_Toc7867)

[4 总结 26](#_Toc2870)

[参考 26](#_Toc31622)

[致谢 26](#_Toc25153)

# **Linux I2C总线驱动**

I2C总线是PHLIPS公司推出的一种串行总线，是具备多主机系统所需的包括总线裁决和高低速器件同步功能的高性能串行总线。I2C总线只有两根双向信号线。一根是数据线SDA，另一根是时钟线SCL。如图1.1所示。每个I2C总线上的器件的地址唯一，主机就是通过寻找唯一的地址来进行数据传输。当主机为发送器的时候，器件接收数据，当主机为接收器的时候，器件发送数据。

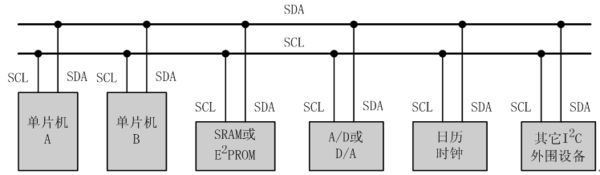


图1.1 I2C总线设备连接

在linux系统下编写I2C驱动，目前主要有两种方法，一种是把I2C设备当作一个普通的字符设备来处理，另一种是利用linux下I2C驱动体系结构来完成。

下面比较下这两种方法：

第一种方法：

优点：

思路比较直接，不需要花很多时间去了解linux中复杂的I2C子系统的操作方法。

缺点：

1. 要求工程师不仅要对I2C设备的操作熟悉，而且要熟悉I2C的适配器(I2C控制器)操作。
2. 要求工程师对I2C的设备器及I2C的设备操作方法都比较熟悉，最重要的是写出的程序可以移植性差。
3. 对内核的资源无法直接使用，因为内核提供的所有I2C设备器以及设备驱动都是基于I2C子系统的格式。

第一种方法的优点就是第二种方法的缺点，

第一种方法的缺点就是第二种方法的优点。

## **I2C体系结构**

Linux的I2C体系结构如图1.2所示。

Linux的I2C体系结构分为3个组成部分。

第一部分：I2C核心。I2C核心提供了I2C总线驱动和设备驱动的注册，注销方法，I2C通信方法(”algorithm”)上层的，与具体适配器无关的代码以及探测设备，检测设备地址的上层代码等。

第二部分：I2C总线驱动。I2C总线驱动是对I2C硬件体系结构中适配器端的实现，适配器可由CPU控制，甚至可以直接集成在CPU内部。

第三部分：I2C设备驱动。I2C设备驱动(也称为客户驱动)是对I2C硬件体系结构中设备端的实现，设备一般挂接在受CPU控制的I2C适配器上，通过I2C适配器与CPU交换数据。

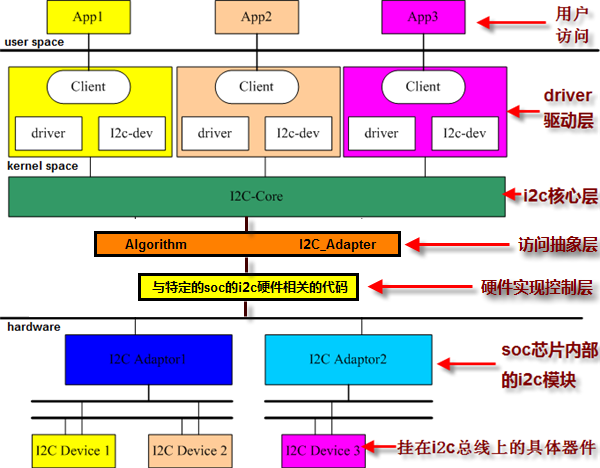


图1.2 I2C体系结构

上图完整的描述了linux i2c驱动架构，虽然I2C硬件体系结构比较简单，但是i2c体系结构在linux中的实现却相当复杂。下面首先说明I2C体系架构分层的细节，然后说明如何编写特定i2c接口器件的驱动程序以及上述架构中的哪些部分需要我们完成，而哪些是linux内核已经完善的或者是芯片提供商已经提供的。

第一层：提供i2c adapter的硬件驱动，探测、初始化i2c adapter（如申请i2c的io地址和中断号），驱动soc控制的i2c adapter在硬件上产生信号（start、stop、ack）以及处理i2c中断。覆盖图中的硬件实现层。

第二层：提供i2c adapter的algorithm，用具体适配器的xxx\_xferf()函数来填充i2c\_algorithm的master\_xfer函数指针，并把赋值后的i2c\_algorithm再赋值给i2c\_adapter的algo指针。覆盖图中的访问抽象层、i2c核心层。

第三层：实现i2c设备驱动中的i2c\_driver接口，用具体的i2c device设备的attach\_adapter()、detach\_adapter()方法赋值给i2c\_driver的成员函数指针。实现设备device与总线（或者叫adapter）的挂接。覆盖图中的driver驱动层。

第四层：实现i2c设备所对应的具体device的驱动，i2c\_driver只是实现设备与总线的挂接，而挂接在总线上的设备则是千差万别的，所以要实现具体设备device的write()、read()、ioctl()等方法，赋值给file\_operations，然后注册字符设备（多数是字符设备）。覆盖图中的driver驱动层。

第一层和第二层又叫i2c总线驱动(bus)，第三第四属于i2c设备驱动(device driver)。

在linux驱动架构中，几乎不需要驱动开发人员再添加bus，因为linux内核几乎集成所有总线bus，如usb、pci、i2c等等。并且总线bus中的(与特定硬件相关的代码)已由芯片提供商编写完成，例如三星的s3c-2440平台i2c总线bus为/drivers/i2c/buses/i2c-s3c2410.c。

第三第四层与特定device相干的就需要驱动工程师来实现了，这部分就是我们需要编写实现的内容。

## **1.2 Linux I2C驱动源码分析**

本小节以最新的内核版本为4.15的I2C主要驱动程序进行分析，为后面I2C适配器驱动和设备驱动设计打下基础。

### 1.2.1 Linux I2C源码文件及目录说明

在Linux内核源代码中的driver目录下包含一个i2c目录，而在此目录下又包含如下文件和文件夹，如图1.3所示。

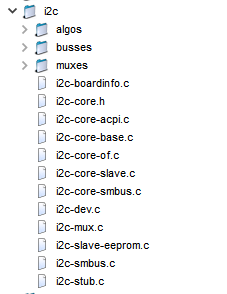


图1.3 Linux/driver/i2c目录

i2c-core.c这个文件实现了I2C核心的功能以及/proc/bus/i2c\*接口。  
　   i2c-dev.c实现了I2C适配器设备文件的功能，每一个I2C适配器都被分配一个设备。通过适配器访设备时的主设备号都为89，次设备号为0-255。I2c-dev.c并没有针对特定的设备而设计，只是提供了通用的read(),write(),和ioctl()等接口，应用层可以借用这些接口访问挂接在适配器上的I2C设备的存储空间或寄存器，并控制I2C设备的工作方式。  
　　busses文件夹这个文件中包含了一些I2C总线的驱动，如针对S3C2410，S3C2440，S3C6410等处理器的I2C控制器驱动为i2c-s3c2410.c。  
　　algos文件夹实现了一些I2C总线适配器的algorithm。

muxes文件夹实现了一种特殊的单向复用器，从2015年开始加入到内核中。

### 1.2.2 Linux I2C重要结构体及相互关系

下面主要介绍几个重要的结构体。

I2C Core，定义了一些函数和数据结构，用于支持I2C控制器驱动（I2C Adapter drvier）和I2C设备驱动（I2C client driver）。I2C Core的存在使I2C控制器驱动和I2C设备驱动独立开来，具有更好的可移置性，同时简化了驱动开发的工作量。但是，I2C Core的存在也使Linux I2C体系结构理解起来更有难度。

I2C Adapter，代表一个I2C控制器，用struct i2c\_adapter来表示。

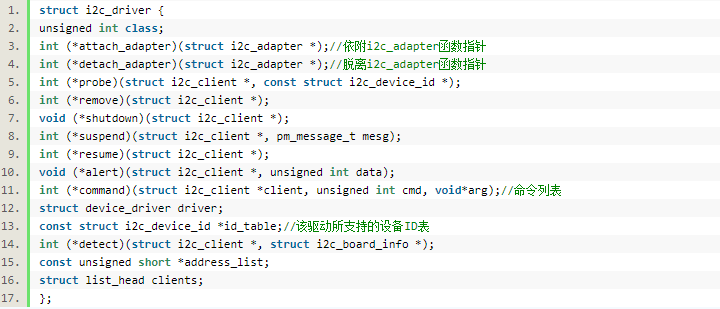
Algorithm，访问I2C控制器的接口，Algorithm直接操作I2C控制器的硬件寄存器来执行数据的发送和接收。它提供操作I2C控制器的最底层操作函数。每个I2C Adapter都有自己的Algorithm，每个I2C Adapter通过自己的Algorithm与挂接在该控制器上的I2C从设备进行通信。

I2C Client，代表一个挂接在I2C总线上的I2C设备。

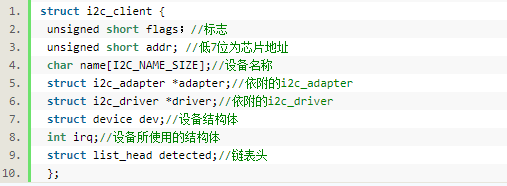
I2C设备驱动（I2C client driver）

I2C-dev，I2C控制器的设备文件，通常命名为i2c-0、i2c-1等等,是I2C控制器的应用层访问接口。各部分源程序如下。

1. i2c\_driver



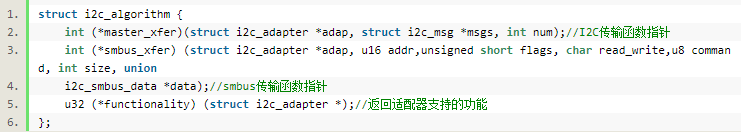
1. i2c\_client



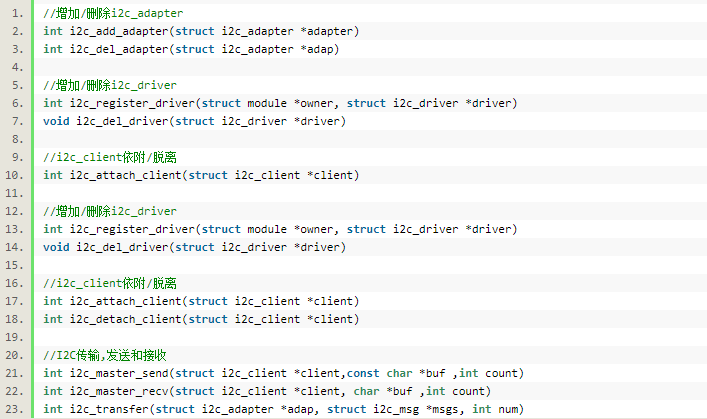
1. i2c\_adapter



1. i2c\_algorithm



1. i2c\_core主要操作。



各结构体的作用与它们之间的关系如下。

i2c\_adapter与i2c\_algorithm。i2c\_adapter对应与物理上的一个适配器，而i2c\_algorithm对应一套通信方法，一个i2c适配器需要i2c\_algorithm中提供的（i2c\_algorithm中的又是更下层与硬件相关的代码提供）通信函数来控制适配器上产生特定的访问周期。缺少i2c\_algorithm的i2c\_adapter什么也做不了，因此i2c\_adapter中包含其使用i2c\_algorithm的指针。

i2c\_algorithm中的关键函数master\_xfer()用于产生i2c访问周期需要的start stop ack信号，以i2c\_msg（即i2c消息）为单位发送和接收通信数据。

i2c\_driver和i2c\_client。i2c\_driver对应一套驱动方法，其主要函数是attach\_adapter()和detach\_client()。i2c\_client对应真实的i2c物理设备device，每个i2c设备都需要一个i2c\_client来描述。i2c\_driver与i2c\_client的关系是一对多。一个i2c\_driver上可以支持多个同等类型的i2c\_client。

i2c\_adapter和i2c\_client。i2c\_adapter和i2c\_client的关系与i2c硬件体系中适配器和设备的关系一致，即i2c\_client依附于i2c\_adapter,由于一个适配器上可以连接多个i2c设备，所以i2c\_adapter中包含依附于它的i2c\_client的链表。

从i2c驱动架构图中可以看出，linux内核对i2c架构抽象了一个叫核心层core的中间件，它分离了设备驱动device driver和硬件控制的实现细节（如操作i2c的寄存器），core层不但为上面的设备驱动提供封装后的内核注册函数，而且还为小面的硬件事件提供注册接口（也就是i2c总线注册接口），可以说core层起到了承上启下的作用。

### 1.2.3 代码层次调用图

有时候代码比任何文字描述都来得直接，但是过多的代码展示反而让人觉得枯燥。这个时候，需要一幅图来梳理一下上面的内容，如图1.4所示。

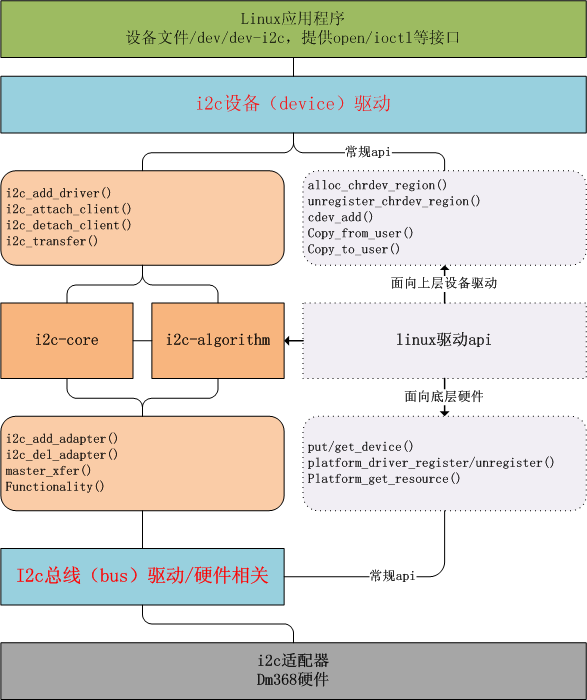


图1.4 代码层次调用图

上面这些代码的展示告诉我们：linux内核和芯片提供商为我们的的驱动程序提供了 i2c驱动的框架，以及框架底层与硬件相关的代码的实现。

剩下的就是针对挂载在i2c两线上的i2c设备了device，而编写的即具体设备驱动了，这里的设备就是硬件接口外挂载的设备，而非硬件接口本身（soc硬件接口本身的驱动可以理解为总线驱动）。

## **1.3 I2C设备驱动设计方法**

### 1.3.1 Linux I2C设备驱动模块的加载与卸载

I2C设备驱动模块加载函数通用的方法是在I2C设备驱动模块加载函数中完成两件事：

（1）通过register\_chrdev()函数将I2C设备注册为一个字符设备。

（2）通过I2C核心的i2c\_add\_driver()函数添加i2c\_driver。

在模块卸载函数中需要做相反的两件事：

（1）通过I2C核心的i2c\_del\_driver()函数删除i2c\_driver。

（2）通过unregister\_chrdev()函数注销字符设备。

加载与卸载的执行函数如图1.5所示。



图1.5 模块的加载与卸载

### 1.3.2 Linux I2C设备驱动的数据传输

I2C设备上的读写数据的时序且数据通常通过i2c\_msg消息数组进行组织，最后通过i2c\_transfer函数完成，如图1.6所示。

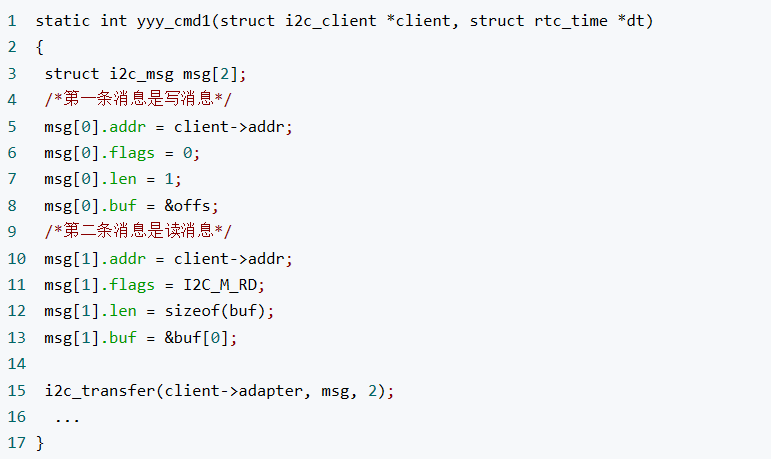


图1.6 i2c 设备驱动数据传输

### 1.3.3 Linux 的i2c-dev.c文件分析

i2c-dev.c文件可以被看作一个I2C设备驱动，它实现的一个i2c\_client是虚拟的、临时的，随着设备文件的打开而产生，并随设备文件的关闭而撤销，并没有被添加到i2c\_adapter的clients链表中。

i2c-dev.c针对每个I2C适配器生成一个主设备为89的设备文件，实现了i2c\_driver的成员函数以及文件操作接口

i2c-dev.c的主体是“i2c\_driver成员函数 + 字符设备驱动”。

i2c-dev.c中提供i2cdev\_read()、i2cdev\_write()函数来对应用户空间要使用的read()和 write()文件操作接口，这两个函数分别调用I2C核心的i2c\_master\_recv()和i2c\_master\_send()函数来构造1条 I2C消息并引发适配器algorithm通信函数的调用，完成消息的传输

i2c-dev.c中i2cdev\_read()和i2cdev\_write()函数不具备太强的通用性，没有太大的实用价值，只能适用于非RepStart模式的情况。

对于2条以上消息组成的读写，在用户空间需要组织i2c\_msg消息数组并调用I2C\_RDWR IOCTL命令。

图1.7代码给出了i2cdev\_ioctl()函数的框架，其中详细列出了I2C\_RDWR命令的处理过程。

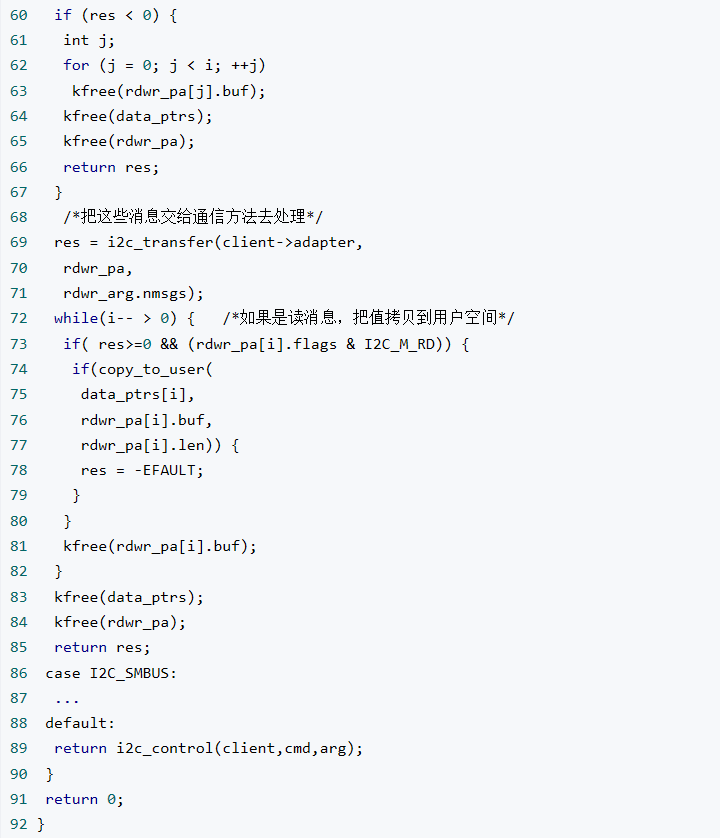
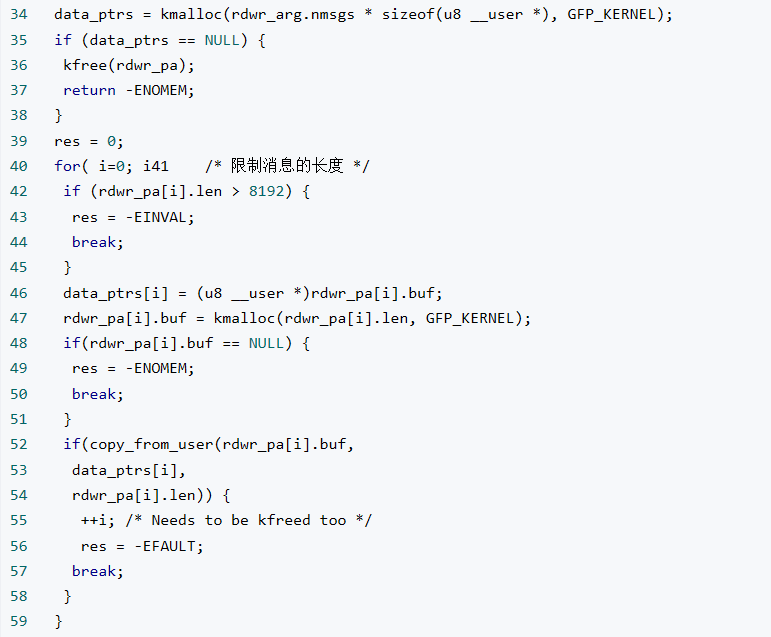
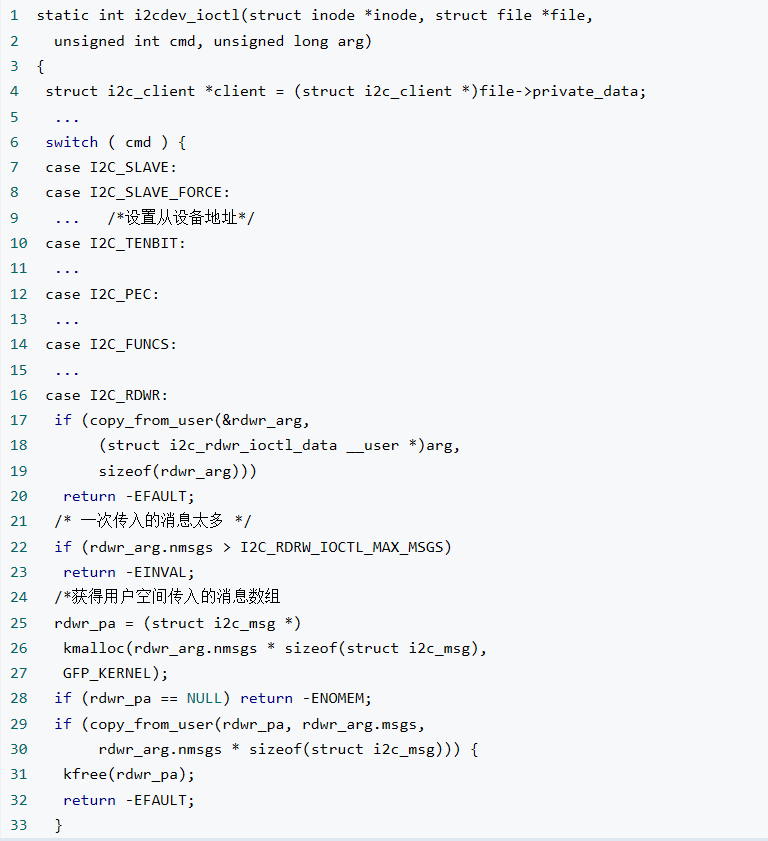


图1.7 i2c-dev分析

## **1.4 小结**

编写具体的I2C驱动时，工程师需要处理的主要工作如下：

（1）提供I2C适配器的硬件驱动，探测，初始化I2C适配器(如申请I2C的I/O地址和中断号)，驱动CPU控制的I2C适配器从硬件上产生。

（2）提供I2C控制的algorithm, 用具体适配器的xxx\_xfer()函数填充i2c\_algorithm的master\_xfer指针，并把i2c\_algorithm指针赋给i2c\_adapter的algo指针。

（3）实现I2C设备驱动中的i2c\_driver接口，用具体yyy的yyy\_probe()，yyy\_remove()，yyy\_suspend(),yyy\_resume()函数指针和i2c\_device\_id设备ID表赋给i2c\_driver的probe,remove,suspend,resume和id\_table指针。

（4）实现I2C设备所对应类型的具体驱动，i2c\_driver只是实现设备与总线的挂接。

上面的工作中前两个属于I2C总线驱动，后面两个属于I2C设备驱动。

# **Linux内核调试环境搭建及调试技巧**

本节主要介绍linux内核调试环境搭建方法，使用基于Qemu+Gdb+Eclipse的平台和虚拟化技术Docker实现内核调试环境的搭建。

## **2.1 搭建方法一：基于Qemu+Eclipse+Gdb**

此方法主要有两个可行方案，一个是在完全命令行下进行调试，如果你是一个命令行爱好者，这将是一个不错的选择。另一种方案加上了eclipse作为图形化调试环境的基础，如果你喜欢图形化界面可以选择方案二来搭建内核调试环境。

开始之前首先执行如下命令安装相关的工具包。

$sudo apt-get install qemu libncurses5-dev gcc-arm-linux-gnueabi build-essential gdb-arm-none-eabi gcc-aarch64-linux-gnu eclipse-cdt libdw-dev systemtap systemtap-runtime

接下来就可以进行相应的环境搭建了。

## 2.1.1 方案一：命令行界面QEMU+GDB

用GDB来调试用户态程序是一个方便快捷的定位问题的方法，极大的缩短了调试程序和定位问题的时间。而对于内核或者驱动ko的调试或者我们想了解内核运行的某些过程，我们也可以借助于gdb工具。

 GDB+QEMU的方式是一种比较常见的调试内核和驱动的方法，这里是参考《奔跑吧linux内核》第六章，一个使用QEMU来单步调试ARM32/ARM64的Linux内核的实验平台，但是默认内核编译的时候默认使用“-02”的GCC编译优化选项。

实验平台为Ubuntu16.04，内核版本为4.13。

详细的调试环境搭建步骤如下：

（1）下载已修改优化选项的内核源码

$ git clone https:github.com/figozhang/runninglinuxkernel\_4.0.git

（2）编译ARM32

$ export ARCH=arm

$ export CROSS\_COMPILE=arm-linux-gnueabi-

$ cd \_install\_arm32/dev/

$ sudo mknod console c 5 1 (注意，不要遗漏该步骤)

$ cd runninglinuxkernel\_4.0

$ make vexpress\_defconfig (在runninglinuxkernel\_4.0目录下输入make命令)

$ make bzImage –j4

$ make dtbs

（3）进入ARM32内核环境

$ qemu-system-arm -M vexpress-a9 -smp 4 -m 1024M -kernel arch/arm/boot/zImage -append "rdinit=/linuxrc console=ttyAMA0 loglevel=8" -dtb arch/arm/boot/dts/vexpress-v2p-ca9.dtb -nographic

（4）运行QEMU

$ qemu-system-arm -M vexpress-a9 -smp 4 -m 1024M -kernel arch/arm/boot/zImage -append "rdinit=/linuxrc console=ttyAMA0 loglevel=8" -dtb arch/arm/boot/dts/vexpress-v2p-ca9.dtb -nographic -S -s

（5）使用GDB连接QEMU，进入调试环境

$ arm-none-eabi-gdb --tui vmlinux

$ target remote localhost:1234

至此，方案一基于Qemu+Gdb的内核调试环境搭建完毕，图2.1是搭建完成的效果截图。

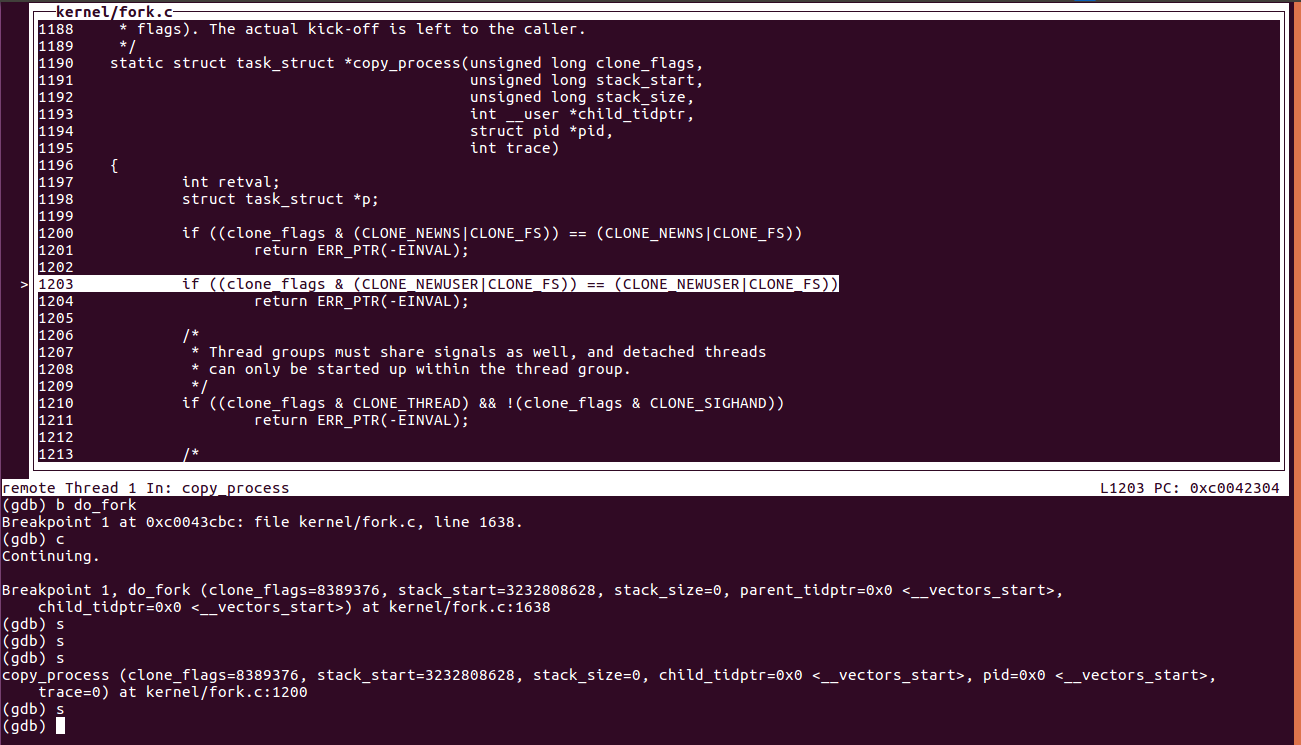


图2.1 Qemu+Gdb调试环境

## 2.1.2 方案二：图形界面QEMU+GDB+Eclipse

1. 如图2.2，配置Debug选项，打开Eclipse,进入run/debug configurations。在main窗口下的c/c++ Application下选择目录下的vmlinux。

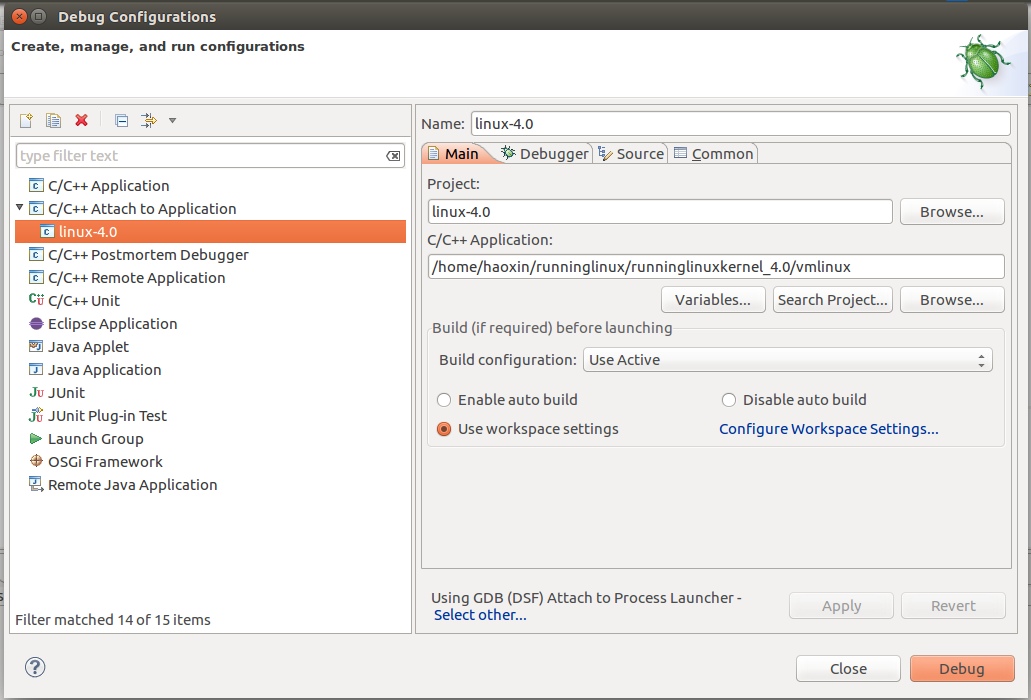
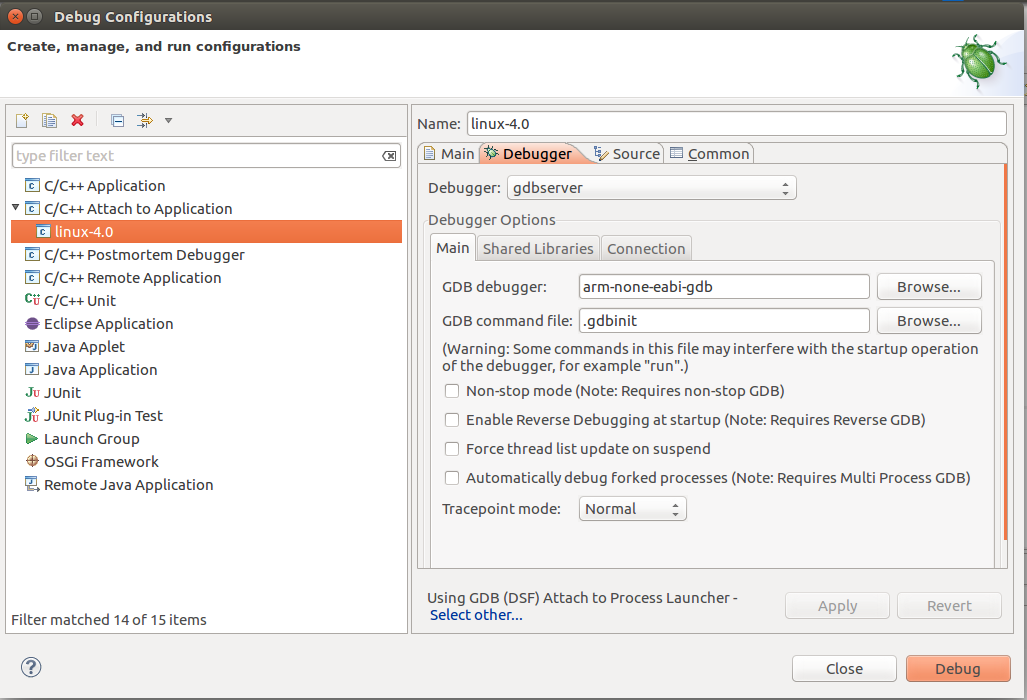


图2.2 配置Debug/main

（2）如图2.3，配置Debug选项，打开Eclipse,进入run/debug configurations。在debugger窗口下的Debugger下选择gdbserver，GDB debugger选择arm-none-eabi-gdb。并在Debugger/Connection下选择TCP协议，IP地址localhost和端口号1234。



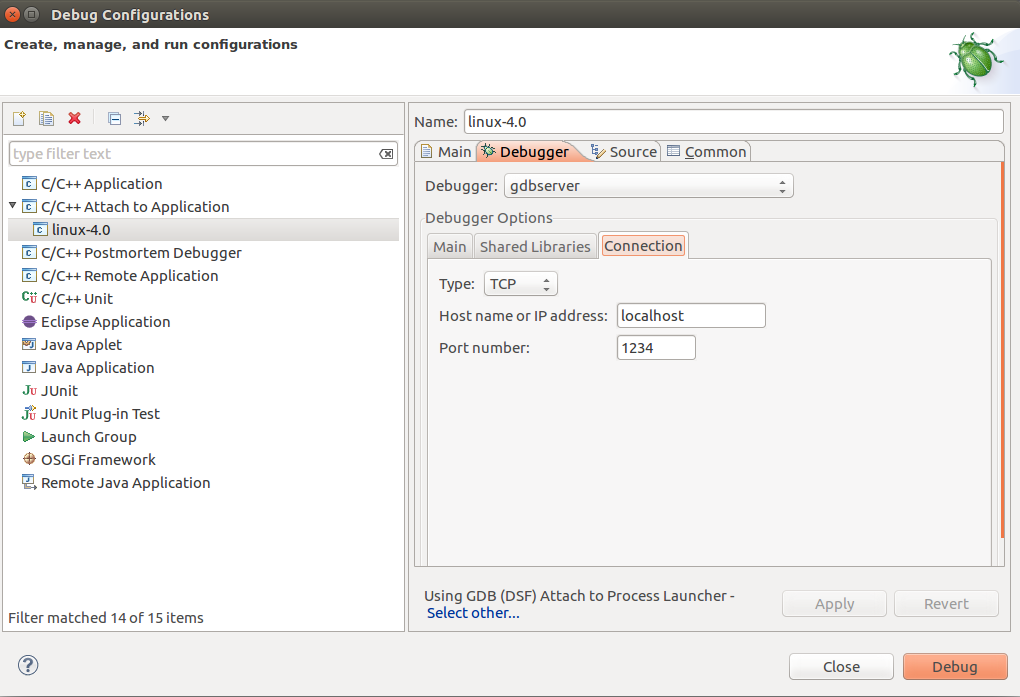


图2.3 配置Debugger

## **2.2 搭建方法二：基于Docker+Qemu**

提到虚拟化技术，我们首先想到的一定是 Docker，经过四年的快速发展 Docker 已经成为了很多公司的标配，也不再是一个只能在开发阶段使用的玩具了。作为在生产环境中广泛应用的产品，Docker 有着非常成熟的社区以及大量的使用者，代码库中的内容也变得非常庞大。

Linux调试环境搭建不应该是复杂的，这个时候借助Docker技术无疑是一种好的选择。那么，我们需要搭建一个实验环境，得基于 Qemu，天生要支持一大堆的免费开发板和处理器架构，使得自然而然地获得一个完整的 Linux 开发板仓库，可以方便各种开发、调试与测试。另外一个是，基于 Docker，可以快速构建和复制这个实验环境，避免一条一条命令反反复复地敲，节约生命。

下面就来开始介绍搭建Docker+Qemu来实现linux内核调试的步骤。这里主要使用泰晓科技提供的开源代码仓库支持。

首先，我们需要在系统中安装Docker。在此以 Ubuntu16.04 和 Qemu 为例。其他 Linux 和 Mac OSX 系统请安装 Docker CE。

1. 获取最新版本的 Docker 安装包

$ wget -qO- https://get.docker.com/ | sh

1. 输入当前用户的密码后，就会下载脚本并且安装Docker及依赖包。安装完成之后，如果想免 sudo 使用 linux lab，请务必把用户加入到 docker 用户组并重启系统。

$ sudo usermod -aG docker $USER

1. 由于 docker 镜像文件比较大，有 1G 左右，下载时请耐心等待。另外，为了提高下载速度，建议通过配置 docker 更换镜像库为本地区的，更换完记得重启 docker 服务。

$ grep registry-mirror /etc/default/docker

DOCKER\_OPTS="$DOCKER\_OPTS --registry-mirror=https://docker.mirrors.ustc.edu.cn"

$ service docker restart

1. 如果 docker 默认的网络环境跟本地的局域网环境地址冲突，请通过如下方式更新 docker 网络环境，并重启 docker 服务。

$ grep bip /etc/default/docker

DOCKER\_OPTS="$DOCKER\_OPTS --bip=10.66.0.10/16"

$ service docker restart

至此，Ubuntu下的Docker环境已经搭建完毕，可以通过docker -v命令查看是否安装成功。

接着介绍如何在docker下搭建linux内核调试环境。

1. 下载GitHub代码。

$ git clone https://github.com/tinyclub/cloud-lab.git

$ cd cloud-lab && tools/docker/choose linux-lab

1. 安装实验环境。

$ tools/docker/pull # Pull from docker hub

$ tools/docker/run # 加载镜像，拉起一个 Linux Lab 容器

1. 启动实验环境。执行 tools/docker/vnc 后会打开一个 VNC 网页，根据 console 提示输入密码登陆即可，之后打开桌面的 Linux Lab 控制台并执行：make boot。默认会启动一个 versatilepb 的 ARM 板子，要指定一块开发板，可以用：

$ make list # 查看支持的列表

$ make BOARD=malta # 这里选择一块 MIPS 板子：malta

$ make boot

1. 配置实验环境。

$ make root-defconfig # 配置根文件系统

$ make kernel-checkout # 从代码仓库检出特定的内核版本（执行前请确保本地修改有备份）

$ make kernel-defconfig # 配置内核

$ make root-menuconfig # 手动配置根文件系统

$ make kernel-menuconfig # 手动配置内核

1. 编译内核。

$ make root # 编译根文件系统，稍微有点慢，需要下载带 sysroot 的编译器

$ make kernel # 编译内核，采用 Ubuntu 和 emdebian.org 提供的交叉编译器

1. 保存所有改动。

$ make save # 保存新的配置和新产生的镜像

$ make kconfig-save # 保存到 boards/BOARD/

$ make rconfig-save

$ make root-save # 保存到 prebuilt/

$ make kernel-save

1. 启动新的根文件系统和内核。需要打开 boards/BOARD/Makefile 屏蔽已经编译的 KIMAG 和 ROOTFS，此时会启动 output/ 目录下刚编译的 rootfs 和内核：

$ vim boards/versatilepb/Makefile

#KIMAGE=$(PREBUILT\_KERNEL)/$(XARCH)/$(BOARD)/$(LINUX)/zImage

#ROOTFS=$(PREBUILT\_ROOTFS)/$(XARCH)/$(CPU)/rootfs.cpio.gz

$ make boot

1. 扩展用法。通过添加或者修改 boards/BOARD/Makefile，可以灵活配置开发板、内核版本以及 BuildRoot等信息。通过它可以灵活打造自己特定的 Linux 实验环境。

$ cat boards/versatilepb/Makefile

ARCH=arm

XARCH=$(ARCH)

CPU=arm926t

MEM=128M

LINUX=2.6.35

NETDEV=smc91c111

SERIAL=ttyAMA0

ROOTDEV=/dev/nfs

ORIIMG=arch/$(ARCH)/boot/zImage

CCPRE=arm-linux-gnueabi-

KIMAGE=$(PREBUILT\_KERNEL)/$(XARCH)/$(BOARD)/$(LINUX) /zImage

ROOTFS=$(PREBUILT\_ROOTFS)/$(XARCH)/$(CPU)/rootfs.cpio.gz

1. 更多用法可以使用make help命令查看帮助。

实验的效果如图2.4所示。

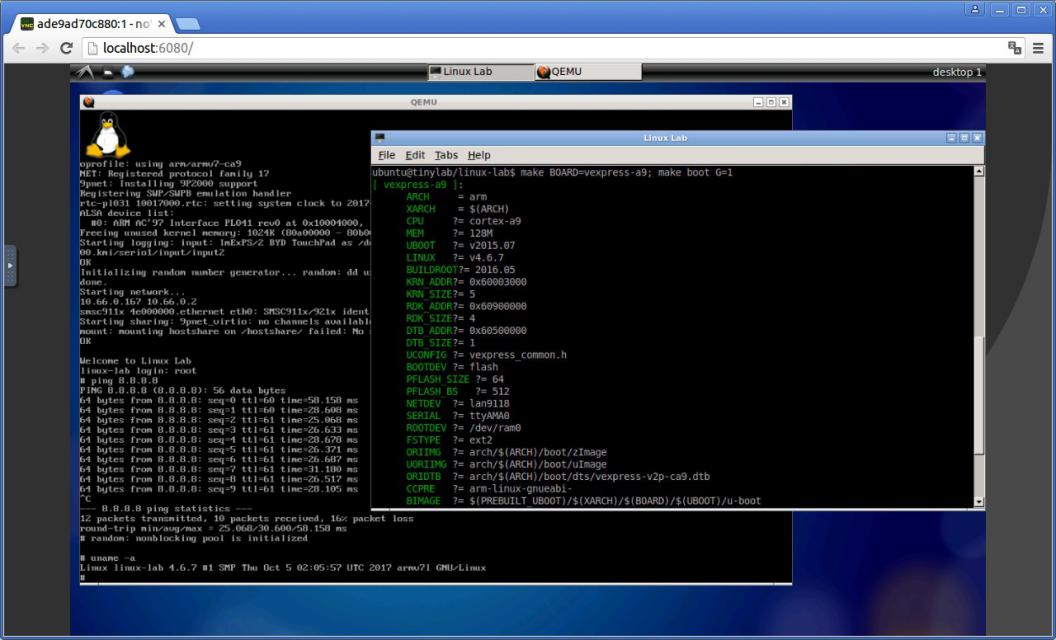


图2.4 Docker+Qemu实验环境

## **2.3 Linux内核调试技巧**

## 2.3.1 printk

内核调试技术之中, 最简单的就是 printk 的使用, 它的用法和C语言应用程序中的 printf 使用类似，但是比printf函数多了一个日志级别，内核中最常见的日志输出都是通过调用printk来实现的，其打印级别有8种可能的记录字串, 在头文件 <linux/kernel.h> 宏定义。在应用程序中依靠的是 stdio.h 中的库, 而在 linux 内核中没有这个库, 所以在 linux 内核中, 实现了自己的一套库函数, printk 就是标准的输出函数。

有时调试内核模块，打印信息太多了，可以通过修改/proc/sys/kernel/printk文件内容来控制。可通过命令cat /proc/sys/kernel/printk查看。默认的设置为：7 4 1 7。

该文件有四个数字值，它们根据日志记录消息的重要性，定义将其发送到何处。关于不同日志级别的更多信息，请查阅syslog(2)联机帮助。上面显示的4个数据分别对应：

控制台日志级别：优先级高于该值的消息将被打印至控制台。

默认的消息日志级别：将用该优先级来打印没有优先级的消息。

最低的控制台日志级别：控制台日志级别可被设置的最小值(最高优先级)。

默认的控制台日志级别：控制台日志级别的缺省值。

这四个数值越小，优先级越高，这四个值是在kernel/printk.c 中被定义：

int console\_printk[4] = {

DEFAULT\_CONSOLE\_LOGLEVEL, /\* console\_loglevel \*/

DEFAULT\_MESSAGE\_LOGLEVEL, /\* default\_message\_loglevel \*/

MINIMUM\_CONSOLE\_LOGLEVEL, /\* minimum\_console\_loglevel \*/

DEFAULT\_CONSOLE\_LOGLEVEL, /\* default\_console\_loglevel \*/

};

内核通过printk() 输出的信息具有日志级别，日志级别是通过在printk() 输出的字符串前加一个带尖括号的整数来控制的，如printk("<6>Hello, world!\n");。内核中共提供了八种不同的日志级别，在 linux/kernel.h 中有相应的宏对应。 #define KERN\_EMERG "<0>" /\* systemis unusable \*/

#define KERN\_ALERT "<1>" /\* actionmust be taken immediately \*/

#define KERN\_CRIT "<2>" /\*critical conditions \*/

#define KERN\_ERR "<3>" /\* errorconditions \*/

#define KERN\_WARNING "<4>" /\* warning conditions \*/

#define KERN\_NOTICE "<5>" /\* normalbut significant \*/

#define KERN\_INFO "<6>" /\*informational \*/

#define KERN\_DEBUG "<7>" /\*debug-level messages \*/

所以printk() 可以这样用：printk(KERN\_INFO"Hello, world!\n")。未指定日志级别的printk() 采用的默认级别是DEFAULT\_MESSAGE\_LOGLEVEL，这个宏在kernel/printk.c 中被定义为整数4，即对应KERN\_WARNING。

如果要想在内核启动过程中打印少的信息，就可以根据自己的需要在kernel/printk.c中修改以上数值，重新编译即可，如下。

/\* printk's without a loglevel use this.. \*/

#define DEFAULT\_MESSAGE\_LOGLEVEL 4 /\* KERN\_WARNING \*/

了解了上面的这些知识后，我们就应该知道如何手动控制printk打印了。例如，想屏蔽掉所有的内核printk打印，只需要把第一个数值调到最小值1或者0。

# echo 1 4 1 7 > /proc/sys/kernel/printk

或者# echo 0 4 0 7 > /proc/sys/kernel/printk

## 2.3.2 printk

# **3 Linux内核分析工具**

## **3.1 系统分析工具**

### **3.1.1 strace**

strace命令是一个集诊断、调试、统计与一体的工具，我们可以使用strace对应用的系统调用和信号传递的跟踪结果来对应用进行分析，以达到解决问题或者是了解应用工作过程的目的。当然strace与专业的调试工具比如说gdb之类的是没法相比的，因为它不是一个专业的调试器。

strace的最简单的用法就是执行一个指定的命令，在指定的命令结束之后它也就退出了。在命令执行的过程中，strace会记录和解析命令进程的所有系统调用以及这个进程所接收到的所有的信号值。

strace基本语法如下：

strace [ -dffhiqrtttTvxx ] [ -acolumn ] [ -eexpr ] ...

[ -ofile ] [-ppid ] ... [ -sstrsize ] [ -uusername ]

[ -Evar=val ] ... [ -Evar ]...

[ command [ arg ... ] ]

strace -c [ -eexpr ] ... [ -Ooverhead ] [ -Ssortby ]

[ command [ arg... ] ]

strace选项如下：

-c 统计每一系统调用的所执行的时间,次数和出错的次数等.

-d 输出strace关于标准错误的调试信息.

-f 跟踪由fork调用所产生的子进程.

-ff 如果提供-o filename,则所有进程的跟踪结果输出到相应的filename.pid中,pid是各进程的进程号.

-F 尝试跟踪vfork调用.在-f时,vfork不被跟踪.

-h 输出简要的帮助信息.

-i 输出系统调用的入口指针.

-q 禁止输出关于脱离的消息.

-r 打印出相对时间关于,,每一个系统调用.

-t 在输出中的每一行前加上时间信息.

-tt 在输出中的每一行前加上时间信息,微秒级.

-ttt 微秒级输出,以秒了表示时间.

-T 显示每一调用所耗的时间.

-v 输出所有的系统调用.一些调用关于环境变量,状态,输入输出等调用由于使用频繁,默认不输出.

-V 输出strace的版本信息.

-x 以十六进制形式输出非标准字符串

-xx 所有字符串以十六进制形式输出.

-a column 设置返回值的输出位置.默认 为40.

-e expr 指定一个表达式,用来控制如何跟踪.格式：[qualifier=][!]value1[,value2]...

qualifier只能是 trace,abbrev,verbose,raw,signal,read,write其中之一.value是用来限定的符号或数字.默认的 qualifier是 trace.感叹号是否定符号.例如:-eopen等价于 -e trace=open,表示只跟踪open调用.而-etrace!=open 表示跟踪除了open以外的其他调用.有两个特殊的符号 all 和 none. 注意有些shell使用!来执行历史记录里的命令,所以要使用\\.

-e trace=set 只跟踪指定的系统 调用.例如:-e trace=open,close,rean,write表示只跟踪这四个系统调用.默认的为set=all.

-e trace=file 只跟踪有关文件操作的系统调用.

-e trace=process 只跟踪有关进程控制的系统调用.

-e trace=network 跟踪与网络有关的所有系统调用.

-e strace=signal 跟踪所有与系统信号有关的 系统调用

-e trace=ipc 跟踪所有与进程通讯有关的系统调用

-e abbrev=set 设定strace输出的系统调用的结果集.-v 等与 abbrev=none.默认为abbrev=all.

-e raw=set 将指定的系统调用的参数以十六进制显示.

-e signal=set 指定跟踪的系统信号.默认为all.如 signal=!SIGIO(或者signal=!io),表示不跟踪SIGIO信号.

-e read=set 输出从指定文件中读出 的数据.例如: -e read=3,5

-e write=set 输出写入到指定文件中的数据.

-o filename 将strace的输出写入文件filename

-p pid 跟踪指定的进程pid.

-s strsize 指定输出的字符串的最大长度.默认为32.文件名一直全部输出.

-u username 以username的UID和GID执行被跟踪的命令

如对简单的C程序，使用gcc编译后使用strace ./a.out，效果如图3.1。

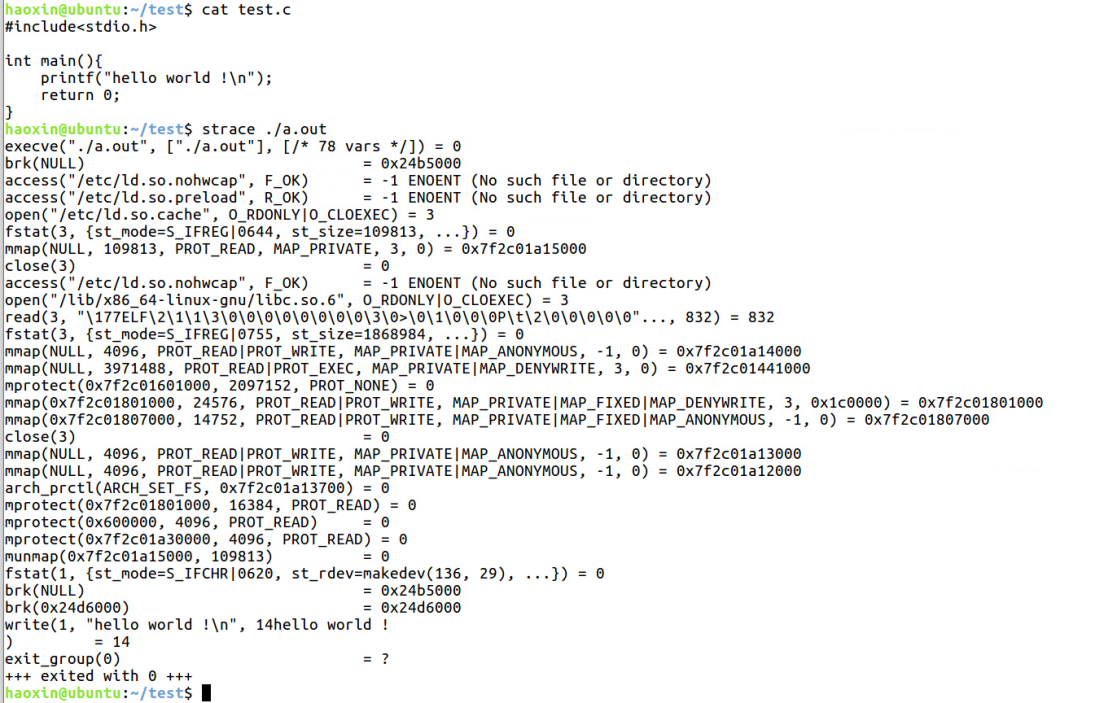


图3.1 strace执行效果

### **3.1.2 ltrace**

ltrace命令是用来跟踪进程调用库函数的情况。

基本语法如下：ltrace [option ...] [command [arg ...]]

ltrace选项有：

-a 对齐具体某个列的返回值。

-c 计算时间和调用，并在程序退出时打印摘要。

-C 解码低级别名称（内核级）为用户级名称。

-d 打印调试信息。

-e 改变跟踪的事件。

-f 跟踪子进程。

-h 打印帮助信息。

-i 打印指令指针，当库调用时。

-l 只打印某个库中的调用。

-L 不打印库调用。

-n, --indent=NR 对每个调用级别嵌套以NR个空格进行缩进输出。

-o, --output=file 把输出定向到文件。

-p PID 附着在值为PID的进程号上进行ltrace。

-r 打印相对时间戳。

-s STRLEN 设置打印的字符串最大长度。

-S 显示系统调用。

-t, -tt, -ttt 打印绝对时间戳。

-T 输出每个调用过程的时间开销。

-u USERNAME 使用某个用户id或组ID来运行命令。

-V, --version 打印版本信息，然后退出。

-x NAME treat the global NAME like a library subroutine.（求翻译）

最基本应用，不带任何参数的情况，输出调用时间开销的情况以及显示系统调用的情况如图3.2所示。

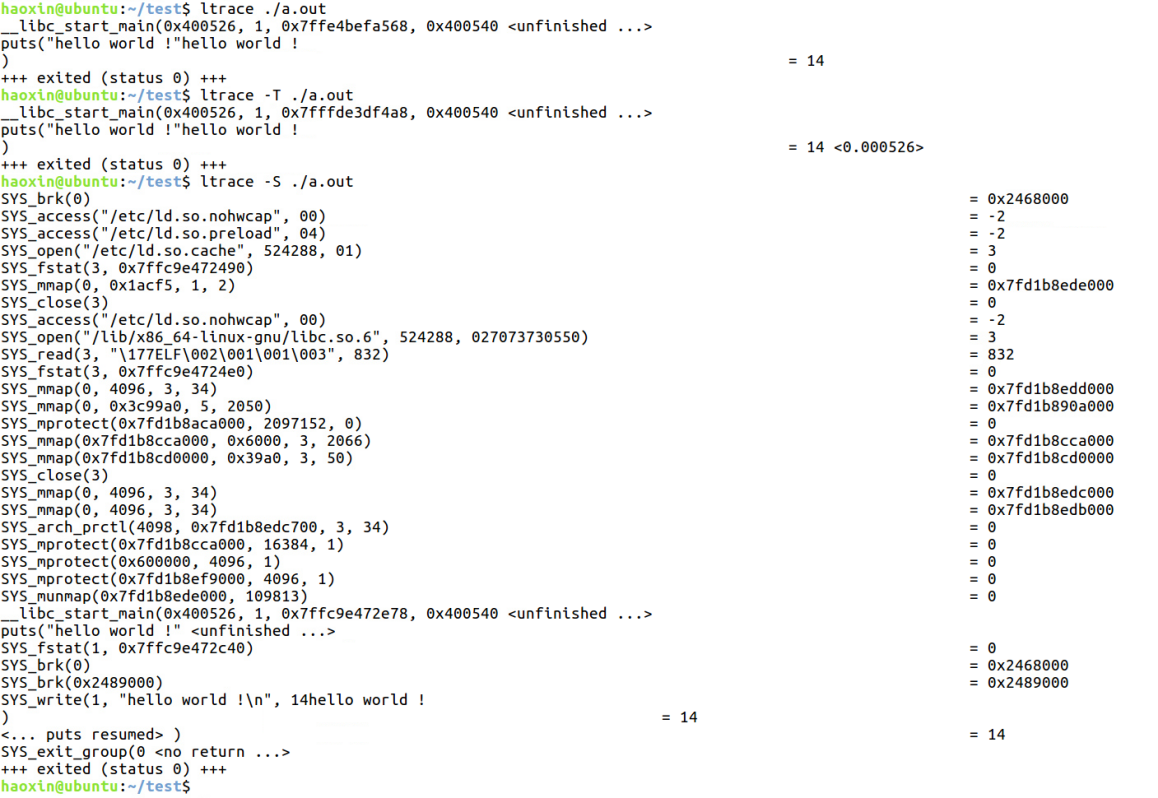


图3.2 ltrace 执行效果

### **3.1.3 pstree**

stree命令以树状图的方式展现进程之间的派生关系，显示效果比较直观。

基本语法为：pstree [option]。

pstree的选项如下：

-a：显示每个程序的完整指令，包含路径，参数或是常驻服务的标示；

-c：不使用精简标示法；

-G：使用VT100终端机的列绘图字符；

-h：列出树状图时，特别标明现在执行的程序；

-H<程序识别码>：此参数效果和指定"-h"参数类似，但特别标明指定的程序；

-l：采用长列格式显示树状图；

-n：用程序识别码排序。预设是以程序名称来排序；

-p：显示程序识别码；

-u：显示用户名称；

-U：使用UTF-8列绘图字符；

-V：显示版本信息。

pstree基本使用情况效果部分截图如图3.3所示。

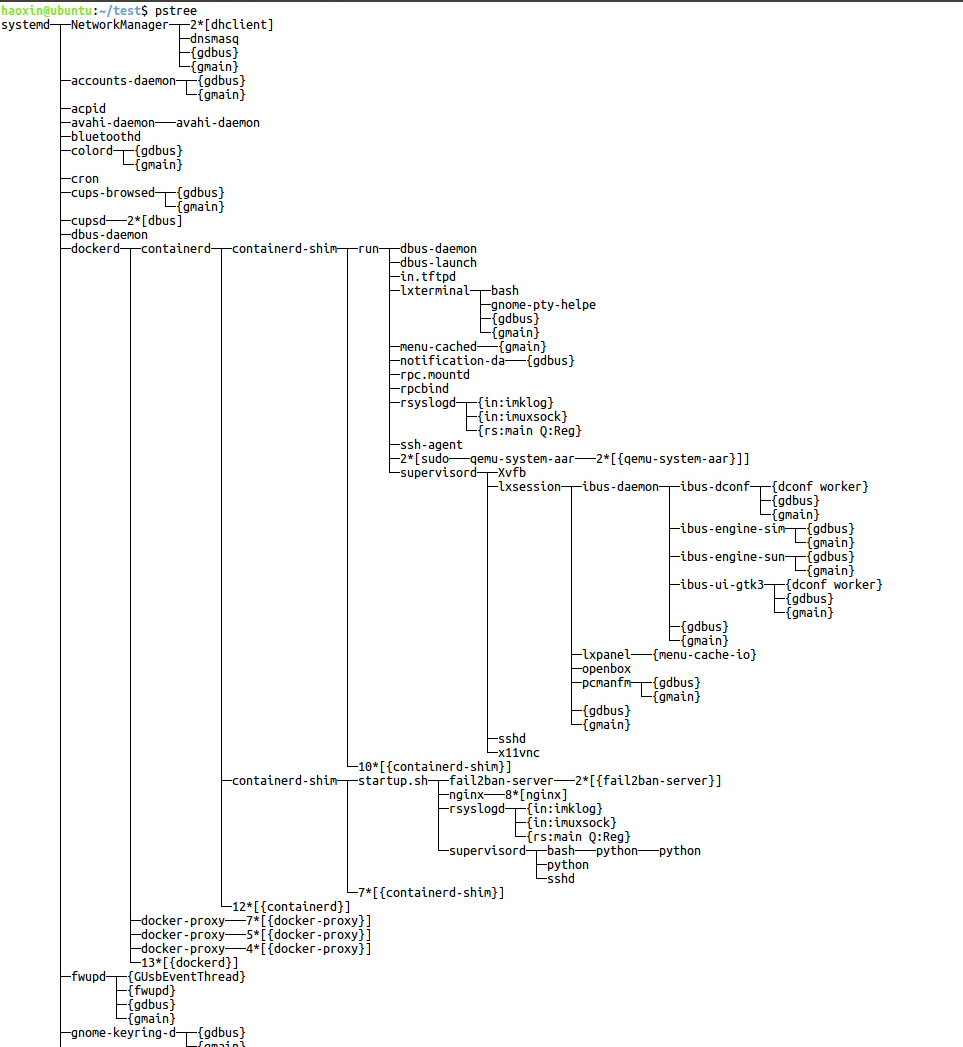


图3.3 pstree 执行效果部分

## **3.2 内存分析工具**

## **3.3 设备分析工具**

## **3.4 文件分析工具**

## **3.5 网络分析工具**

## **3.6 安全分析工具**

# **4 总结**

**参考**

**致谢**