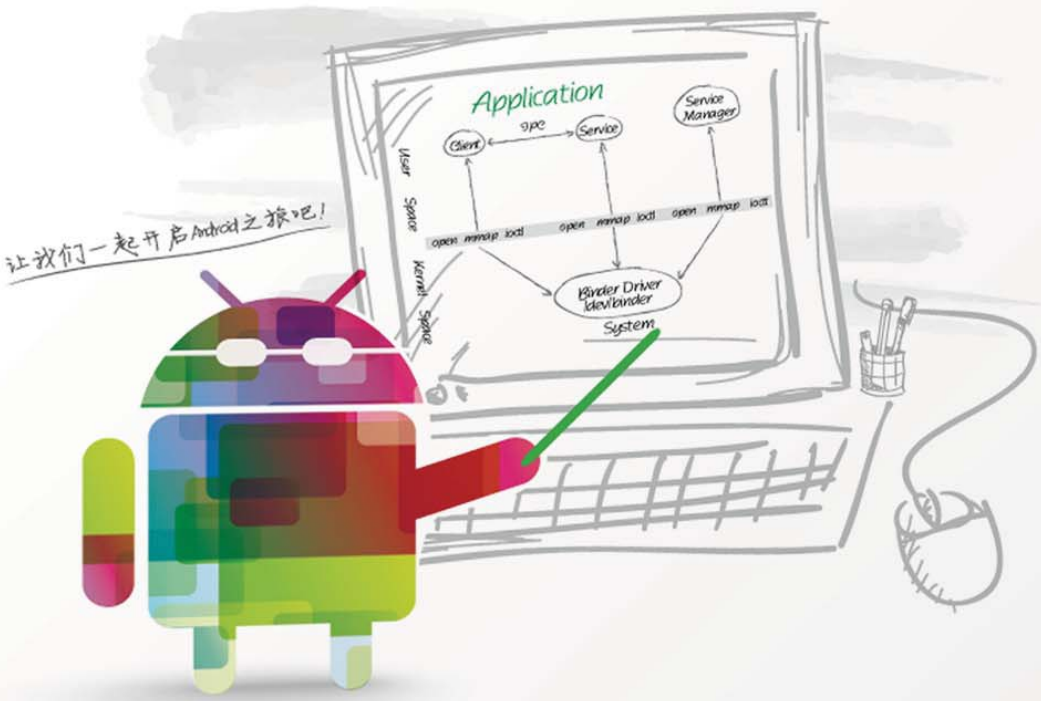
Broadview
www.broadview.com.cn

全面、深入、细致地掌握Android系统，引领移动互联网新时代！

Android
系统源代码情景分析

◎罗升阳 著◎



電子工業出版社
PUBLISHING HOUSE OF ELECTRONICS INDUSTRY
http://www.phei.com.cn

Android系统源代码情景分析
電子工業出版社

内 容 简 介

在内容上，本书结合使用情景，全面、深入、细致地分析了Android系统的源代码，涉及到Linux内核层、硬件抽象层（HAL）、运行时库层（Runtime）、应用程序框架层（Application Framework）以及应用程序层（Application）。

在组织上，本书将上述内容划分为初识Android系统、Android专用驱动系统和Android应用程序框架三大篇。初识Android系统篇介绍了参考书籍、基础知识以及实验环境搭建；Android专用驱动系统篇介绍了Logger日志驱动程序、Binder进程间通信驱动程序以及Ashmem匿名共享内存驱动程序；Android应用程序框架篇从组件、进程、消息以及安装四个维度对Android应用程序的框架进行了深入的剖析。

通过上述内容及其组织，本书能使读者既能从整体上把握Android系统的层次结构，又能从细节上掌握每一个层次的要点。

图书在版编目（CIP）数据

Android系统源代码情景分析 / 罗升阳著. —北京：电子工业出版社，2012.10
ISBN 978-7-121-18108-5

Ⅰ. ①A… Ⅱ. ①罗… Ⅲ. ①移动终端—应用程序—程序设计 Ⅳ. ①TN929.53

中国版本图书馆CIP数据核字（2012）第204380号

策划编辑：符隆美
责任编辑：葛　娜
印　　刷：北京东光印刷厂
装　　订：三河市鹏成印业有限公司
出版发行：电子工业出版社
　　北京市海淀区万寿路173信箱　　邮编：100036
开　　本：787 × 1092　1 /16　　印张：52.5　　字数：1570千字
印　　次：2012年9月第1次印刷
印　　数：0000册　　定价：00.00元（含光盘1张）

凡所购买电子工业出版社图书有缺损问题，请向购买书店调换。若书店售缺，请与本社发行部联系，联系及邮购电话：（010）88254888。

质量投诉请发邮件至zlts@phei.com.cn，盗版侵权举报请发邮件至dbqq@phei.com.cn。

服务热线：（010）88258888。

前　言

虽然Android系统自2008年9月发布第一个版本1.0以来，截至2011年10月发布最新版本4.0，一共存在十多个版本，但是据官方统计，截至2012年3月5日，占据首位的是Android 2.3，市场占有率达到66.5%；其次是Android 2.2，市场占有率为25.3%；第三位是Android 2.1，市场占有率为6.6%；而最新发布的Android 3.2和Android 4.0的市场占有率只有3.3%和2%。因此，在本书中，我们选择了Android 2.3的源代码来分析Android系统的实现，一是因为从目前来说，它的基础架构是最稳定的；二是因为它是使用最广泛的。

本书内容

全书分为初识Android系统篇、Android专用驱动系统篇和Android应用程序框架篇三个部分。

初识Android系统篇包含三个章节的内容，主要介绍Android系统的基础知识。第1章介绍与Android系统有关的参考书籍，以及Android源代码工程环境的搭建方法；第2章介绍Android系统的硬件抽象层；第3章介绍Android系统的智能指针。读者可能会觉得奇怪，为什么一开始就介绍Android系统的硬件抽象层呢？因为涉及硬件，它似乎是一个深奥的知识点。其实不然，Android系统的硬件抽象层无论是从实现上，还是从使用上，它的层次都是非常清晰的，而且从下到上涵盖了整个Android系统，包括Android系统在用户空间和内核空间的实现。内核空间主要涉及硬件驱动程序的编写方法，而用户空间涉及运行时库层、应用程序框架层及应用程序层。因此，尽早学习Android系统的硬件抽象层，有助于我们从整体上去认识Android系统，以便后面可以更好地分析它的源代码。在分析Android系统源代码的过程中，经常会碰到智能指针，第3章我们就重点分析Android系统智能指针的实现原理，也是为了后面可以更好地分析Android系统源代码。

Android专用驱动系统篇包含三个章节的内容。我们知道，Android系统是基于Linux内核来开发的，但是由于移动设备的CPU和内存配置都要比PC低，因此，Android系统并不是完全在Linux内核上开发的，而是在Linux内核里面添加了一些专用的驱动模块来使它更适合于移动设备。这些专用的驱动模块同时也形成了Android系统的坚实基础，尤其是Logger日志驱动程序、Binder进程间通信驱动程序，以及Ashmem匿名共享内存驱动程序，它们在Android系统中被广泛地使用。在此篇中，我们分别在第4章、第5章和第6章分析Logger日志系统、Binder进程间通信系统和Ashmem共享内存系统的实现原理，为后面深入分析Android应用程序的框架打下良好的基础。

Android应用程序框架篇包含十个章节的内容。我们知道，在移动平台中，Android系统、iOS系统和Windows Phone系统正在形成三足鼎立之势，谁的应用程序更丰富、质量更高、用户体验更好，谁就能取得最终的胜利。因此，每个平台都在尽最大努力吸引第三方开发者来为其开发应用程序。这就要求平台必须提供良好的应用程序架构，以便第三方开发者可以将更多的精力集中在应用程序的业务逻辑上，从而开发出数量更多、质量更高和用户体验更好的应用程序。在此篇中，我们将从组件、进程、消息和安装四个维度来分析Android应用程序的实现框架。第7章到第10章分析Android应用程序

四大组件Activity、Service、Broadcast Receiver和Content Provider的实现原理；第11章和第12章分析Android应用程序进程的启动过程；第13章到第15章分析Android应用程序的消息处理机制；第16章分析Android应用程序的安装和显示过程。学习了这些知识之后，我们就可以掌握Android系统的精髓了。

本书特点

本书从初学者的角度出发，结合具体的使用情景，在纵向和横向上对Android系统的源代码进行了全面、深入、细致的分析。在纵向上，采用从下到上的方式，分析的源代码涉及了Android系统的内核层（Linux Kernel）、硬件抽象层（HAL）、运行时库层（Runtime）、应用程序框架层（Application Framework）以及应用程序层（Application），这有利于读者从整体上掌握Android系统的架构。在横向上，从Android应用程序的组件、进程、消息以及安装四个角度出发，全面地剖析了Android系统的应用程序框架层，这有利于读者深入地理解Android应用程序的架构以及运行原理。

作　者

目　　录

第1篇　初识Android系统

第2篇 Android专用驱动系统

第 2 章

硬件抽象层

Android系统的硬件抽象层（Hardware Abstract Layer，HAL）运行在用户空间中，它向下屏蔽硬件驱动模块的实现细节，向上提供硬件访问服务。通过硬件抽象层，Android系统分两层来支持硬件设备，其中一层实现在用户空间中，另一层实现在内核空间中。传统的Linux系统把对硬件的支持完全实现在内核空间中，即把对硬件的支持完全实现在硬件驱动模块中。

Android系统为什么要把对硬件的支持划分为两层来实现呢？我们知道，一方面，Linux内核源代码是遵循GPL[1]协议的，即如果我们在Android系统所使用的Linux内核中添加或者修改了代码，那么就必须将它们公开。因此，如果Android系统像其他的Linux系统一样，把对硬件的支持完全实现在硬件驱动模块中，那么就必须将这些硬件驱动模块源代码公开，这样就可能会损害移动设备厂商的利益，因为这相当于暴露了硬件的实现细节和参数。另一方面，Android系统源代码是遵循Apache License[2]协议的，它允许移动设备厂商添加或者修改Android系统源代码，而又不必公开这些代码。因此，如果把对硬件的支持完全实现在Android系统的用户空间中，那么就可以隐藏硬件的实现细节和参数。然而，这是无法做到的，因为只有内核空间才有特权操作硬件设备。一个折中的解决方案便是将对硬件的支持分别实现在内核空间和用户空间中，其中，内核空间仍然是以硬件驱动模块的形式来支持，不过它只提供简单的硬件访问通道；而用户空间以硬件抽象层模块的形式来支持，它封装了硬件的实现细节和参数。这样就可以保护移动设备厂商的利益了。

本章介绍Android系统的硬件抽象层的目的在于认识Android系统的体系结构，因为它的实现和使用依次涉及Android系统的硬件驱动模块、硬件抽象层、外部库和运行时库层、应用程序框架层和应用程序层等，如图2-1所示。

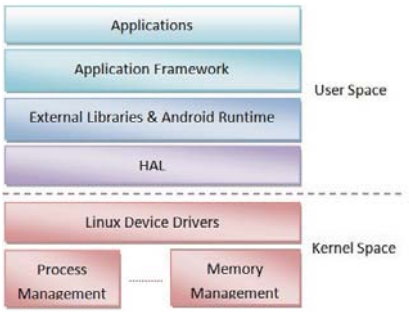


图2-1　Android系统的体系结构

1　GNU General Public License，即GNU通用公共许可证，简称为GPL，是一个被广泛使用的自由软件许可证，具体可以参考http://www.gnu.org/copyleft/gpl.html。

2　Apache Licence是一种对商业应用友好的许可，由非盈利开源组织Apache创建，具体可以参考http://www.apache.org/licenses/LICENSE-2.0.html。

接下来，我们就以硬件抽象层为中心，首先在Android系统的内核空间中为一个硬件开发驱动程序，接着在用户空间中为该硬件添加一个硬件抽象层模块，并且在应用程序框架层中添加一个硬件访问服务，最后开发一个应用程序来访问该硬件服务。这样我们就可以从下到上来认识Android系统的体系结构。

2.1 开发Android硬件驱动程序

为了方便描述，我们将为一个虚拟的字符硬件设备开发驱动程序。这个虚拟的字符硬件设备只有一个寄存器，它的大小为4字节，可读可写。由于这个字符设备是虚拟的，且只有一个寄存器，因此，我们将它称为“fake register”，并且将对应的驱动程序命名为freg。

在Android系统中开发硬件驱动程序的方法与一般Linux系统是一样的，因此，本节假设读者已经对Linux设备驱动有一定的了解，具体可以参考在前面1.1小节中介绍的*Linux Device Drivers*一书。接下来，我们首先实现驱动程序freg，然后介绍它的编译方法，最后验证它的功能正确性。

2.1.1 实现内核驱动程序模块

驱动程序freg的目录结构如下：

```
~/Android/kernel/goldfish
----drivers
     ----freg
          ----freg.h
          ----freg.c
          ----Kconfig
          ----Makefile
```

它由四个文件组成，其中freg.h和freg.c是源代码文件，Kconfig是编译选项配置文件，Makefile是编译脚本文件。下面我们就分别介绍这四个文件的实现。

freg.h

```
01 #ifndef _FAKE_REG_H_
02 #define _FAKE_REG_H_
03
04 #include <linux/cdev.h>
05 #include <linux/semaphore.h>
06
07 #define FREG_DEVICE_NODE_NAME  "freg"
08 #define FREG_DEVICE_FILE_NAME  "freg"
09 #define FREG_DEVICE_PROC_NAME  "freg"
10 #define FREG_DEVICE_CLASS_NAME "freg"
11
12 struct fake_reg_dev {
13         int val;
14         struct semaphore sem;
15         struct cdev dev;
16 };
17
18 #endif
```

这个文件定义了四个字符串常量，分别用来描述虚拟硬件设备freg在设备文件系统中的名称。此外，此文件还定义了一个结构体fake_reg_dev，用来描述虚拟硬件设备freg。在结构体fake_reg_dev中，成员变量val用来描述一个虚拟寄存器，它的类型为int；成员变量sem是一个信号量，用来同步访问虚

拟寄存器val；成员变量dev是一个标准的Linux字符设备结构体变量，用来标志该虚拟硬件设备freg的类型为字符设备。

freg.c

这个文件是驱动程序freg的实现文件，它向用户空间提供了三个接口来访问虚拟硬件设备freg中的寄存器val。第一个是proc文件系统接口，第二个是传统的设备文件系统接口，第三个是devfs文件系统接口。下面我们就分段介绍该驱动程序的实现。

文件开头包含了必要的头文件，以及定义了一些相关的变量和函数原型，它们的含义可以参考代码中的注释。

```
001 #include <linux/init.h>
002 #include <linux/module.h>
003 #include <linux/types.h>
004 #include <linux/fs.h>
005 #include <linux/proc_fs.h>
006 #include <linux/device.h>
007 #include <asm/uaccess.h>
008
009 #include "freg.h"
010
011 /*主设备号和从设备号变量*/
012 static int freg_major = 0;
013 static int freg_minor = 0;
014
015 /*设备类别和设备变量*/
016 static struct class* freg_class = NULL;
017 static struct fake_reg_dev* freg_dev = NULL;
018
019 /*传统的设备文件操作方法*/
020 static int freg_open(struct inode* inode, struct file* filp);
021 static int freg_release(struct inode* inode, struct file* filp);
022 static ssize_t freg_read(struct file* filp, char __user *buf, size_t count, loff_t* f_pos);
023 static ssize_t freg_write(struct file* filp, const char __user *buf, size_t count, loff_t* f_pos);
024
025 /*传统的设备文件操作方法表*/
026 static struct file_operations freg_fops = {
027         .owner = THIS_MODULE,
028         .open = freg_open,
029         .release = freg_release,
030         .read = freg_read,
031         .write = freg_write,
032 };
033
034 /*devfs文件系统的设备属性操作方法*/
035 static ssize_t freg_val_show(struct device* dev, struct device_attribute* attr,  char* buf);
036  static ssize_t freg_val_store(struct device* dev, struct device_attribute* attr, const
char* buf, size_t count);
037
038 /*devfs文件系统的设备属性*/
039 static DEVICE_ATTR(val, S_IRUGO | S_IWUSR, freg_val_show, freg_val_store);
```

接下来，定义用来访问虚拟硬件设备freg的传统设备文件系统接口，主要是实现freg_open、freg_release、freg_read和freg_write四个函数，其中，前面两个函数用来打开和关闭虚拟硬件设备freg，而后面两个函数用来读取和写入虚拟硬件设备freg中的寄存器val。这些函数的实现细节可以参考代码中的注释，如下所示。

```
040 /*打开设备方法*/
041 static int freg_open(struct inode* inode, struct file* filp) {
```

```
042         struct fake_reg_dev* dev;
043
044         /*将自定义设备结构体保存在文件指针的私有数据域中，以便访问设备时可以直接拿来用*/
045         dev = container_of(inode->i_cdev, struct fake_reg_dev, dev);
046         filp->private_data = dev;
047
048         return 0;
049 }
050
051 /*设备文件释放时调用，空实现*/
052 static int freg_release(struct inode* inode, struct file* filp) {
053         return 0;
054 }
055
056 /*读取设备的寄存器val的值*/
057 static ssize_t freg_read(struct file* filp, char __user *buf, size_t count, loff_t* f_pos) {
058         ssize_t err = 0;
059         struct fake_reg_dev* dev = filp->private_data;
060
061         /*同步访问*/
062         if(down_interruptible(&(dev->sem))) {
063                 return -ERESTARTSYS;
064         }
065
066         if(count < sizeof(dev->val)) {
067                 goto out;
068         }
069
070         /*将寄存器val的值拷贝到用户提供的缓冲区中*/
071         if(copy_to_user(buf, &(dev->val), sizeof(dev->val))) {
072                 err = -EFAULT;
073                 goto out;
074         }
075
076         err = sizeof(dev->val);
077
078 out:
079         up(&(dev->sem));
080         return err;
081 }
082
083 /*写设备的寄存器val的值*/
084 static ssize_t freg_write(struct file* filp, const char __user *buf, size_t count, loff_t* f_pos) {
085         struct fake_reg_dev* dev = filp->private_data;
086         ssize_t err = 0;
087
088         /*同步访问*/
089         if(down_interruptible(&(dev->sem))) {
090                 return -ERESTARTSYS;
091         }
092
093         if(count != sizeof(dev->val)) {
094                 goto out;
095         }
096
097         /*将用户提供的缓冲区的值写到设备寄存器中*/
098         if(copy_from_user(&(dev->val), buf, count)) {
099                 err = -EFAULT;
100                 goto out;
101         }
102
103         err = sizeof(dev->val);
104
```

```
105 out:
106         up(&(dev->sem));
107         return err;
108 }
```

接下来，我们继续定义用来访问虚拟硬件设备freg的devfs文件系统接口。这种硬件访问接口将虚拟硬件设备freg的寄存器val当作设备的一个属性，通过读写这个属性就可以达到访问设备的目的，这是通过调用freg_val_show和freg_val_store这两个函数来实现的。为了方便后面编写proc文件系统接口来访问虚拟硬件设备freg，我们定义了两个内部使用的函数__freg_get_val和__freg_set_val，它们分别用来读写虚拟硬件设备freg的寄存器val。它们的实现如下所示。

```
109 /*将寄存器val的值读取到缓冲区buf中，内部使用*/
110 static ssize_t __freg_get_val(struct fake_reg_dev* dev, char* buf) {
111         int val = 0;
112 
113         /*同步访问*/
114         if(down_interruptible(&(dev->sem))) {
115                 return -ERESTARTSYS;
116         }
117 
118         val = dev->val;
119         up(&(dev->sem));
120 
121         return snprintf(buf, PAGE_SIZE, "%d\n", val);
122 }
123 
124 /*把缓冲区buf的值写到设备寄存器val中，内部使用*/
125 static ssize_t __freg_set_val(struct fake_reg_dev* dev, const char* buf, size_t count) {
126         int val = 0;
127 
128         /*将字符串转换成数字*/
129         val = simple_strtol(buf, NULL, 10);
130 
131         /*同步访问*/
132         if(down_interruptible(&(dev->sem))) {
133                 return -ERESTARTSYS;
134         }
135 
136         dev->val = val;
137         up(&(dev->sem));
138 
139         return count;
140 }
141 
142 /*读设备属性val的值*/
143 static ssize_t freg_val_show(struct device* dev, struct device_attribute* attr, char* buf) {
144         struct fake_reg_dev* hdev = (struct fake_reg_dev*)dev_get_drvdata(dev);
145 
146         return __freg_get_val(hdev, buf);
147 }
148 
149 /*写设备属性val的值*/
150 static ssize_t freg_val_store(struct device* dev, struct device_attribute* attr, const char* buf, size_t count) {
151          struct fake_reg_dev* hdev = (struct fake_reg_dev*)dev_get_drvdata(dev);
152 
153         return __freg_set_val(hdev, buf, count);
154 }
```

接下来，我们继续定义用来访问虚拟硬件设备freg的proc文件系统接口，主要是实现freg_proc_read和freg_proc_write这两个函数。同时，我们还定义了用来在proc文件系统中创建和删除/proc/freg文件的函数freg_create_proc和freg_remove_proc。它们的实现如下所示。

```
155 /*读取设备寄存器val的值，保存到page缓冲区中*/
156 static ssize_t freg_proc_read(char* page, char** start, off_t off, int count, int* eof, void* data) {
157         if(off > 0) {
158                 *eof = 1;
159                 return 0;
160         }
161
162         return __freg_get_val(freg_dev, page);
163 }
164
165 /*把缓冲区的值buff保存到设备寄存器val中*/
166 static ssize_t freg_proc_write(struct file* filp, const char __user *buff, unsigned long len, void* data) {
167         int err = 0;
168         char* page = NULL;
169
170         if(len > PAGE_SIZE) {
171                 printk(KERN_ALERT"The buff is too large: %lu.\n", len);
172                 return -EFAULT;
173         }
174
175         page = (char*)__get_free_page(GFP_KERNEL);
176         if(!page) {
177                 printk(KERN_ALERT"Failed to alloc page.\n");
178                 return -ENOMEM;
179         }
180
181         /*先把用户提供的缓冲区的值拷贝到内核缓冲区中*/
182         if(copy_from_user(page, buff, len)) {
183                 printk(KERN_ALERT"Failed to copy buff from user.\n");
184                 err = -EFAULT;
185                 goto out;
186         }
187
188         err = __freg_set_val(freg_dev, page, len);
189
190 out:
191         free_page((unsigned long)page);
192         return err;
193 }
194
195 /*创建/proc/freg文件*/
196 static void freg_create_proc(void) {
197         struct proc_dir_entry* entry;
198
199         entry = create_proc_entry(FREG_DEVICE_PROC_NAME, 0, NULL);
200         if(entry) {
201                 entry->owner = THIS_MODULE;
202                 entry->read_proc = freg_proc_read;
203                 entry->write_proc = freg_proc_write;
204         }
205 }
206
207 /*删除/proc/freg文件*/
208 static void freg_remove_proc(void) {
209         remove_proc_entry(FREG_DEVICE_PROC_NAME, NULL);
210 }
```

最后，我们定义驱动程序freg的模块加载与卸载函数freg_init和freg_exit。函数freg_init主要用来注册和初始化虚拟硬件设备freg，而函数freg_exit用来反注册和释放虚拟硬件设备freg。它们的实现如下所示。

```
211 /*初始化设备*/
212 static int  __freg_setup_dev(struct fake_reg_dev* dev) {
213         int err;
214         dev_t devno = MKDEV(freg_major, freg_minor);
215 
216         memset(dev, 0, sizeof(struct fake_reg_dev));
217 
218         /*初始化字符设备*/
219         cdev_init(&(dev->dev), &freg_fops);
220         dev->dev.owner = THIS_MODULE;
221         dev->dev.ops = &freg_fops;
222 
223         /*注册字符设备*/
224         err = cdev_add(&(dev->dev),devno, 1);
225         if(err) {
226                 return err;
227         }
228 
229         /*初始化信号量和寄存器val的值*/
230         init_MUTEX(&(dev->sem));
231         dev->val = 0;
232 
233         return 0;
234 }
235 
236 /*模块加载方法*/
237 static int __init freg_init(void) {
238         int err = -1;
239         dev_t dev = 0;
240         struct device* temp = NULL;
241 
242         printk(KERN_ALERT"Initializing freg device.\n");
243 
244         /*动态分配主设备号和从设备号*/
245         err = alloc_chrdev_region(&dev, 0, 1, FREG_DEVICE_NODE_NAME);
246         if(err < 0) {
247                 printk(KERN_ALERT"Failed to alloc char dev region.\n");
248                 goto fail;
249         }
250 
251         freg_major = MAJOR(dev);
252         freg_minor = MINOR(dev);
253 
254         /*分配freg设备结构体*/
255         freg_dev = kmalloc(sizeof(struct fake_reg_dev), GFP_KERNEL);
256         if(!freg_dev) {
257                 err = -ENOMEM;
258                 printk(KERN_ALERT"Failed to alloc freg device.\n");
259                 goto unregister;
260         }
261 
262         /*初始化设备*/
263         err = __freg_setup_dev(freg_dev);
264         if(err) {
265                 printk(KERN_ALERT"Failed to setup freg device: %d.\n", err);
266                 goto cleanup;
267         }
268 
269         /*在/sys/class/目录下创建设备类别目录freg*/
270         freg_class = class_create(THIS_MODULE, FREG_DEVICE_CLASS_NAME);
271         if(IS_ERR(freg_class)) {
272                 err = PTR_ERR(freg_class);
273                 printk(KERN_ALERT"Failed to create freg device class.\n");
```

```
274                 goto destroy_cdev;
275         }
276
277         /*在/dev/目录和/sys/class/freg目录下分别创建设备文件freg*/
278         temp = device_create(freg_class, NULL, dev, "%s", FREG_DEVICE_FILE_NAME);
279         if(IS_ERR(temp)) {
280                 err = PTR_ERR(temp);
281                 printk(KERN_ALERT"Failed to create freg device.\n");
282                 goto destroy_class;
283         }
284
285         /*在/sys/class/freg/freg目录下创建属性文件val*/
286         err = device_create_file(temp, &dev_attr_val);
287         if(err < 0) {
288                 printk(KERN_ALERT"Failed to create attribute val of freg device.\n");
289                 goto destroy_device;
290         }
291
292         dev_set_drvdata(temp, freg_dev);
293
294         /*创建/proc/freg文件*/
295         freg_create_proc();
296
297         printk(KERN_ALERT"Succedded to initialize freg device.\n");
298
299         return 0;
300
301 destroy_device:
302         device_destroy(freg_class, dev);
303 destroy_class:
304         class_destroy(freg_class);
305 destroy_cdev:
306         cdev_del(&(freg_dev->dev));
307 cleanup:
308         kfree(freg_dev);
309 unregister:
310         unregister_chrdev_region(MKDEV(freg_major, freg_minor), 1);
311 fail:
312         return err;
313 }
314
315 /*模块卸载方法*/
316 static void __exit freg_exit(void) {
317         dev_t devno = MKDEV(freg_major, freg_minor);
318
319         printk(KERN_ALERT"Destroy freg device.\n");
320
321         /*删除/proc/freg文件*/
322         freg_remove_proc();
323
324         /*销毁设备类别和设备*/
325         if(freg_class) {
326                 device_destroy(freg_class, MKDEV(freg_major, freg_minor));
327                 class_destroy(freg_class);
328         }
329
330         /*删除字符设备和释放设备内存*/
331         if(freg_dev) {
332                 cdev_del(&(freg_dev->dev));
333                 kfree(freg_dev);
334         }
335
336         /*释放设备号资源*/
```

```
337         unregister_chrdev_region(devno, 1);
338 }
339
340 MODULE_LICENSE("GPL");
341 MODULE_DESCRIPTION("Fake Register Driver");
342
343 module_init(freg_init);
344 module_exit(freg_exit);
```

驱动程序freg的源文件准备好之后，接下来就要为它编写编译选项配置文件和编译脚本文件了。

Kconfig

```
1 config FREG
2         tristate "Fake Register Driver"
3         default n
4         help
5         This is the freg driver for android system.
```

这个文件定义了驱动程序freg的编译选项。在编译驱动程序freg之前，我们可以通过执行make menuconfig命令来设置这些编译选项，以便可以指定驱动程序freg的编译方式。从这个配置文件就可以看出，驱动程序freg可以以三种方式来编译。第一种方式是直接内建在内核中；第二种方式是编译成内核模块；第三种方式是不编译到内核中。默认的编译方式为n，即不编译到内核中，因此，在编译驱动程序freg之前，我们需要执行make menuconfig命令来修改它的编译选项，以便可以将驱动程序freg内建到内核中或者以模块的方式来编译。

Makefile

```
1 obj-$(CONFIG_FREG) += freg.o
```

这是驱动程序freg的编译脚本文件，其中，$(CONFIG_FREG)是一个变量，它的值与驱动程序freg的编译选项有关。如果选择将驱动程序freg内建到内核中，那么变量$(CONFIG_FREG)的值为y；如果选择以模块的方式来编译驱动程序freg，那么变量$(CONFIG_FREG)的值为m；如果变量$(CONFIG_FREG)的值既不为y，也不为m，那么驱动程序freg就不会被编译。

至此，我们就为虚拟硬件设备freg开发了一个驱动程序。接下来，我们还需要修改内核中的Kconfig和Makefile文件来支持驱动程序freg的编译。

2.1.2　修改内核Kconfig文件

虽然我们在2.1.1小节中为驱动程序freg编写了一个Kconfig文件，但是在默认情况下，在执行make menuconfig命令配置内核编译选项时，编译系统是无法找到这个Kconfig文件的。这时候，我们需要修改内核的根Kconfig文件，使得编译系统能够找到驱动程序freg的Kconfig文件。

当执行make menuconfig命令时，编译系统会读取arch/$(ARCH)目录下的Kconfig文件，其中，$(ARCH)指向编译的目标CPU体系架构。在前面的1.4.2小节中，我们将$(ARCH)的值设置为arm，因此，就需要修改arch/arm目录下的Kconfig文件，使得编译系统可以找到驱动程序freg的Kconfig文件。打开arch/arm/Kconfig文件，找到以下两行内容：

```
menu "Device Drivers"
……
endmenu
```

在这两行内容之间添加下面一行内容，将驱动程序freg的Kconfig文件包含进来。

```
menu "Device Drivers"
source "drivers/freg/Kconfig"
……
endmenu
```

这样，当我们执行make menuconfig命令来配置内核编译选项时，编译系统就可以找到驱动程序freg中的Kconfig文件了，这时候，我们就可以配置驱动程序freg的编译方式了。

一般来说，各个CPU体系架构目录下的Kconfig文件都会通过source "drivers/Kconfig"命令把drivers目录下的Kconfig文件包含进去。例如，x86体系架构下的Kconfig文件有下面一行内容：

```
……
source "drivers/Kconfig"
……
```

这意味着当我们在内核中新增了一个驱动程序时，只需要将它的Kconfig文件包含drivers目录下的Kconfig文件就可以了。但是arm体系架构比较特殊，它没有将drivers目录下的Kconfig文件包含到自身的Kconfig文件中，而是将drivers/Kconfig文件的内容原封不动地拷贝到它的Kconfig文件中。因此，考虑到兼容其他体系架构，我们最好也以同样的方式来修改drivers/Kconfig文件，即在里面添加一行：source "drivers/freg/Kconfig"。

2.1.3 修改内核Makefile文件

与Kconfig文件相似，虽然我们在2.1.1小节中为驱动程序freg编写了一个编译脚本文件Makefile，但是在默认情况下，在执行make命令编译内核时，编译系统是无法找到这个Makefile文件的。这时候，我们需要修改drivers目录下的Makefile文件，使得编译系统能够找到驱动程序freg的Makefile文件。打开drivers/Makefile文件，在里面添加以下一行内容：

```
……
obj-$(CONFIG_FREG)+= freg/
```

这样，当我们执行make命令来编译内核时，编译系统就会对驱动程序freg进行编译。

2.1.4 编译内核驱动程序模块

前面提到，在编译驱动程序freg之前，我们需要执行make menuconfig命令来配置它的编译方式。

```
USER@MACHINE:~/Android/kernel/goldfish$ make menuconfig
```

在弹出来的第一个配置界面中用上下箭头键选择“Device Drivers”项，按Enter键；接着在弹出来的第二个配置界面中继续用上下箭头键选择“Fake Register Driver”项，按Y键或者M键，就可以看到选项前面方括号中的字符变成“*”或者“M”符号，它们分别表示将驱动程序freg编译到内核中或者以模块的方式来编译。

```
| |     [*] Fake Register Driver                                    | |
……
```

注意

如果我们要以模块的方式来编译驱动程序freg，那么就必须先在第一个配置界面中选择“Enable loadable module support”选项，并且按Y键将它的值设置为true，即使得内核可以支持动态加载模块，这样才能在第二个配置界面中按M键来配置“Fake Register Driver”选项。同样，如果要使得内核支持动态卸载模块，那么就要在第一个配置界面中选择“Enable loadable module support”选项中的子选项“Module unloading”，并且按Y键将它的值设置为true。

配置完成后，保存编译配置选项，退出make menuconfig命令，然后就可以执行make命令来编译驱动程序freg了。

```
USER@MACHINE:~/Android/kernel/goldfish$ make
```

当看到下面的输出时，就说明驱动程序freg编译成功了。

```
OBJCOPY arch/arm/boot/zImage
Kernel: arch/arm/boot/zImage is ready
```

编译得到的内核镜像文件zImage保存在arch/arm/boot目录下，我们可以使用它来启动Android模拟器。

2.1.5　验证内核驱动程序模块

编译了驱动程序freg之后，我们就可以通过proc文件系统和devfs文件系统来验证它的功能是否正确。我们首先使用前面在2.1.4小节中得到的内核镜像文件zImage来启动Android模拟器，然后用adb工具连接上它，最后就可以使用cat和echo命令来读写/proc/freg文件或者/sys/class/freg/freg/val文件的内容了，即读写虚拟硬件设备freg的寄存器val的内容。如果读出来的内容与上次写入的内容相同，就说明我们为虚拟硬件设备freg所编写的驱动程序freg是正确的。

在读写虚拟硬件设备freg的寄存器val的内容之前，我们需要检查设备上的/dev目录下是否存在一个设备文件freg。如果存在，就说明驱动程序freg成功地将虚拟硬件设备freg注册到设备文件系统中了。

```
USER@MACHINE: ~/Android$ emulator -kernel kernel/goldfish/arch/arm/boot/zImage &
USER@MACHINE: ~/Android$ adb shell
root@android:/ # cd dev
root@android:/dev # ls freg
freg
```

接下来，我们就进入到/proc目录中，首先使用cat命令读取文件freg的内容，然后使用echo命令往文件freg中写入一个新的内容，最后使用cat命令将文件freg的内容读取出来，看看是否与上次写入的内容相同。

```
root@android:/proc # cat freg
0
root@android:/proc # echo '5' > freg
root@android:/proc # cat freg
5
```

如果能看到上面的输出，就说明我们能够使用proc文件系统接口来访问虚拟硬件设备freg的寄存器val的内容，即说明前面所开发的驱动程序freg的功能是正确的。

最后，我们进入到/sys/class/freg/freg中，首先使用cat命令读取val文件的内容，然后使用echo命令往文件val中写入一个新的内容，最后使用cat命令将文件val中的内容读取出来，同样是检查它是否与上次写入的内容相同。

```
root@android:/sys/class/freg/freg # cat val
5
root@android:/sys/class/freg/freg # echo '0' > val
root@android:/sys/class/freg/freg # cat val
0
```

如果能看到上面的输出，就说明我们能够使用devfs文件系统接口来访问虚拟硬件设备freg的寄存器val的内容，同样说明前面所开发的驱动程序freg的功能是正确的。

以上两种方法只验证了驱动程序freg所提供的proc和devfs文件系统访问接口是正确的，我们还需要进一步验证它所提供的dev文件系统访问接口也是正确的，即能正常读写设备文件/dev/freg的内容。由于设备文件/dev/freg的内容是二进制格式的，因此，使用cat和echo命令来读写它的内容不够直观，在接下来的2.2小节中，我们将通过编写一个C可执行程序来直观地验证它的dev文件系统访问接口的正确性。

2.2 开发C可执行程序验证Android硬件驱动程序

在本节中，我们将通过编写一个C可执行程序来验证驱动程序freg所提供的dev文件系统接口的正确性，这是通过调用read和write函数读写设备文件/dev/freg的内容来实现的。对于Android应用程序开发者来说，可能会觉得奇怪，怎么能在Android系统中编写C语言程序呢？其实在Android源代码工程环境中，不仅可以用C/C++语言来开发可执行程序，还可以开发动态链接库，即so文件。使用adb工具命令连接上Android模拟器之后，进入到/system/bin或者/system/lib目录中，就可以看到很多可执行程序或者动态链接库文件。在接下来的2.3小节中，我们为虚拟硬件设备freg所编写的硬件抽象层模块接口其实就是一个动态链接库文件。

在Android源代码工程环境中开发的C可执行程序源文件一般保存在external目录中，因此，我们进入到external目录中，并且创建一个freg目录，用来保存我们将要开发的C可执行程序源文件。它的目录结构如下：

```
~/Android
----external
    ----freg
        ----freg.c
        ----Android.mk
```

这个C应用程序只有一个源文件 freg.c和一个编译脚本文件Android.mk。下面我们就详细分析这两个文件的内容。

freg.c

```
01 #include <stdio.h>
02 #include <stdlib.h>
03 #include <fcntl.h>
04
05 #define FREG_DEVICE_NAME "/dev/freg"
06
07 int main(int argc, char** argv) {
08         int fd = -1;
09         int val = 0;
10
11         fd = open(FREG_DEVICE_NAME, O_RDWR);
12         if(fd == -1) {
13                 printf("Failed to open device %s.\n", FREG_DEVICE_NAME);
14                 return -1;
15         }
16
17         printf("Read original value:\n");
18         read(fd, &val, sizeof(val));
19         printf("%d.\n\n", val);
20
21         val = 5;
22         printf("Write value %d to %s.\n\n", val, FREG_DEVICE_NAME);
23         write(fd, &val, sizeof(val));
```

```
24
25          printf("Read the value again:\n");
26          read(fd, &val, sizeof(val));
27          printf("%d.\n\n", val);
28
29          close(fd);
30
31          return 0;
32  }
```

第11行通过调用open函数，并且以读写方式打开设备文件/dev/freg；接着第18行调用read函数读取它的内容，即读取虚拟硬件设备freg的寄存器val的内容，并且将它的内容打印出来。

第23行调用write函数将一个整数5写入到虚拟硬件设备freg的寄存器val中；接着第26行和第27行再调用read和print函数将这个整数5读取并且打印出来。假设虚拟硬件设备freg的寄存器val是第一次被访问，那么如果一切正常的话，两次打印出来的内容就应该分别为0和5。

Android.mk

```
1 LOCAL_PATH := $(call my-dir)
2 include $(CLEAR_VARS)
3 LOCAL_MODULE_TAGS := optional
4 LOCAL_MODULE := freg
5 LOCAL_SRC_FILES := $(call all-subdir-c-files)
6 include $(BUILD_EXECUTABLE)
```

这是源文件freg.c的编译脚本文件，它与使用Java语言开发的Android应用程序的编译脚本的不同之处在于include命令后面所带的参数。对于C可执行程序来说，它的编译脚本中的include命令后面跟的参数的值为$(BUILD_EXECUTABLE)，表示当前要编译的是一个可执行应用程序模块，并且将编译结果保存在out/target/product/gerneric/system/bin目录中。

准备好这两个文件之后，我们就可以通过mmm和make snod命令来编译和打包这个C可执行程序了。

```
USER@MACHINE:~/Android$ mmm ./external/freg/
USER@MACHINE:~/Android$ make snod
```

编译成功后，就可以在out/target/product/gerneric/system/bin目录下看到一个freg文件；而当打包成功后，这个文件就会包含在out/target/product/gerneric目录下的Android系统镜像文件system.img中。

最后，我们就可以使用编译和打包后得到的system.img文件启动Android模拟器，然后使用adb工具连接上它，并且进入到/system/bin目录中，执行里面的freg文件来验证驱动程序freg的dev文件系统访问接口的正确性。

```
USER@MACHINE: ~/Android$ emulator -kernel kernel/goldfish/arch/arm/boot/zImage &
USER@MACHINE: ~/Android$ adb shell
root@android:/ # cd system/bin
root@android:/system/bin # ./freg
Read original value:
0.
Write value 5 to /dev/freg.
Read the value again:
5.
```

如果能够看到上面的输出，就说明驱动程序freg所提供的dev文件系统访问接口是正确的。

接下来，我们继续介绍如何为虚拟硬件设备freg编写硬件抽象层模块接口。

2.3 开发Android硬件抽象层模块

Android系统为硬件抽象层中的模块接口定义了编写规范，我们必须按照这个规范来编写自己的硬件模块接口，否则就会导致无法正常访问硬件。下面我们首先介绍硬件抽象层模块接口的编写规范，然后再按照这个规范为虚拟硬件设备freg开发硬件抽象层模块接口，并且分析硬件抽象层模块的加载过程，最后讨论硬件设备的访问权限问题。

2.3.1 硬件抽象层模块编写规范

Android系统的硬件抽象层以模块的形式来管理各个硬件访问接口。每一个硬件模块都对应有一个动态链接库文件，这些动态链接库文件的命令需要符合一定的规范。同时，在系统内部，每一个硬件抽象层模块都使用结构体hw_module_t来描述，而硬件设备则使用结构体hw_device_t来描述。接下来，我们就分别描述硬件抽象层模块文件的命名规范以及结构体hw_module_t和hw_device_t的定义。

2.3.1.1 硬件抽象层模块文件命名规范

硬件抽象层模块文件的命名规范定义在hardware/libhardware/hardware.c文件中，如下所示。

hardware/libhardware/hardware.c

```
01 /**
02  * There are a set of variant filename for modules. The form of the filename
03  * is "<MODULE_ID>.variant.so" so for the led module the Dream variants
04  * of base "ro.product.board", "ro.board.platform" and "ro.arch" would be:
05  *
06  * led.trout.so
07  * led.msm7k.so
08  * led.ARMV6.so
09  * led.default.so
10  */
11
12 static const char *variant_keys[] = {
13     "ro.hardware",  /* This goes first so that it can pick up a different
14                        file on the emulator. */
15     "ro.product.board",
16     "ro.board.platform",
17     "ro.arch"
18 };
```

这段代码和注释的意思是，硬件抽象层模块文件的命名规范为“<MODULE_ID>.variant.so”，其中，MODULE_ID表示模块的ID，variant表示四个系统属性ro.hardware、ro.product.board、ro.board.platform和ro.arch之一。系统在加载硬件抽象层模块时，依次按照ro.hardware、ro.product.board、ro.board.platform和ro.arch的顺序来取它们的属性值。如果其中的一个系统属性存在，那么就把它的值作为variant的值，然后再检查对应的文件是否存在，如果存在，那么就找到要加载的硬件抽象层模块文件了；否则，就继续查找下一个系统属性。如果这四个系统属性都不存在，或者对应于这四个系统属性的硬件抽象层模块文件都不存在，那么就使用“<MODULE_ID>.default.so”来作为要加载的硬件抽象层模块文件的名称。

系统属性ro.hardware是在系统启动时，由init进程负责设置的。它首先会读取/proc/cmdline文件，检查里面有没有一个名称为androidboot.hardware的属性，如果有，就把它的值作为属性ro.hardware的值；否则，就将/proc/cpuinfo文件的内容读取出来，并且将里面的硬件信息解析出来，即将Hardware

字段的内容作为属性ro.hardware的值。例如，在Android模拟器中，从/proc/cpuinfo文件读取出来的Hardware字段内容为goldfish，于是，init进程就会将属性ro.hardware的值设置为“goldfish”。系统属性ro.product.board、ro.board.platform和ro.arch是从/system/build.prop文件读取出来的。文件/system/build.prop是由编译系统中的编译脚本build/core/Makefile和Shell脚本build/tools/buildinfo.sh生成的，有兴趣的读者可以研究一下这两个文件，这里就不深入分析了。

2.3.1.2 硬件抽象层模块结构体定义规范

结构体hw_module_t和hw_device_t及其相关的其他结构体定义在文件hardware/libhardware/include/hardware/hardware.h中。下面我们就分别介绍这些结构体的定义。

hw_module_t

hardware/libhardware/include/hardware/hardware.h

```
01 /*
02  * Value for the hw_module_t.tag field
03  */
04
05 #define MAKE_TAG_CONSTANT(A,B,C,D) (((A) << 24) | ((B) << 16) | ((C) << 8) | (D))
06 #define HARDWARE_MODULE_TAG MAKE_TAG_CONSTANT('H', 'W', 'M', 'T')
07
08 /**
09  * Name of the hal_module_info
10  */
11 #define HAL_MODULE_INFO_SYM         HMI
12
13 /**
14  * Every hardware module must have a data structure named HAL_MODULE_INFO_SYM
15  * and the fields of this data structure must begin with hw_module_t
16  * followed by module specific information.
17  */
18 typedef struct hw_module_t {
19     /** tag must be initialized to HARDWARE_MODULE_TAG */
20     uint32_t tag;
21
22     /** major version number for the module */
23     uint16_t version_major;
24
25     /** minor version number of the module */
26     uint16_t version_minor;
27
28     /** Identifier of module */
29     const char *id;
30
31     /** Name of this module */
32     const char *name;
33
34     /** Author/owner/implementor of the module */
35     const char *author;
36
37     /** Modules methods */
38     struct hw_module_methods_t* methods;
39
40     /** module's dso */
41     void* dso;
42
43     /** padding to 128 bytes, reserved for future use */
44     uint32_t reserved[32-7];
45
46 } hw_module_t;
```

结构体hw_module_t中的每一个成员变量在代码中都有详细的解释，这里不再重复。不过，有五点是需要注意的。

（1）在结构体hw_module_t的定义前面有一段注释，意思是，硬件抽象层中的每一个模块都必须自定义一个硬件抽象层模块结构体，而且它的第一个成员变量的类型必须为hw_module_t。

（2）硬件抽象层中的每一个模块都必须存在一个导出符号HAL_MODULE_IFNO_SYM，即“HMI”，它指向一个自定义的硬件抽象层模块结构体。后面我们在分析硬件抽象层模块的加载过程时，将会看到这个导出符号的意义。

（3）结构体hw_module_t的成员变量tag的值必须设置为HARDWARE_MODULE_TAG，即设置为一个常量值（'H' <<24 | 'W'<<16 | 'M'<<8 | 'T'），用来标志这是一个硬件抽象层模块结构体。

（4）结构体hw_module_t的成员变量dso用来保存加载硬件抽象层模块后得到的句柄值。前面提到，每一个硬件抽象层模块都对应有一个动态链接库文件。加载硬件抽象层模块的过程实际上就是调用dlopen函数来加载与其对应的动态链接库文件的过程。在调用dlclose函数来卸载这个硬件抽象层模块时，要用到这个句柄值，因此，我们在加载时需要将它保存起来。

（5）结构体hw_module_t的成员变量methods定义了一个硬件抽象层模块的操作方法列表，它的类型为hw_module_methods_t，接下来我们就介绍它的定义。

hw_module_methods_t

hardware/libhardware/include/hardware/hardware.h

```
1 typedef struct hw_module_methods_t {
2     /** Open a specific device */
3     int (*open)(const struct hw_module_t* module, const char* id,
4             struct hw_device_t** device);
5
6 } hw_module_methods_t;
```

结构体hw_module_methods_t只有一个成员变量，它是一个函数指针，用来打开硬件抽象层模块中的硬件设备。其中，参数module表示要打开的硬件设备所在的模块；参数id表示要打开的硬件设备的ID；参数device是一个输出参数，用来描述一个已经打开的硬件设备。由于一个硬件抽象层模块可能会包含多个硬件设备，因此，在调用结构体hw_module_methods_t的成员变量open来打开一个硬件设备时，我们需要指定它的ID。硬件抽象层中的硬件设备使用结构体hw_device_t来描述，接下来我们就介绍它的定义。

hw_device_t

hardware/libhardware/include/hardware/hardware.h

```
01 #define HARDWARE_DEVICE_TAG MAKE_TAG_CONSTANT('H', 'W', 'D', 'T')
02
03 /**
04  * Every device data structure must begin with hw_device_t
05  * followed by module specific public methods and attributes.
06  */
07 typedef struct hw_device_t {
08     /** tag must be initialized to HARDWARE_DEVICE_TAG */
09     uint32_t tag;
10
11     /** version number for hw_device_t */
12     uint32_t version;
13
14     /** reference to the module this device belongs to */
15     struct hw_module_t* module;
16
17     /** padding reserved for future use */
```

```
18     uint32_t reserved[12];
19
20     /** Close this device */
21     int (*close)(struct hw_device_t* device);
22
23 } hw_device_t;
```

结构体hw_device_t中的每一个成员变量在代码中都有详细的解释，这里不再重复。不过，有三点是需要注意的。

（1）硬件抽象层模块中的每一个硬件设备都必须自定义一个硬件设备结构体，而且它的第一个成员变量的类型必须为hw_device_t。

（2）结构体hw_device_t的成员变量tag的值必须设置为HARDWARE_DEVICE_TAG，即设置为一个常量值（'H' <<24 | 'W'<<16 | 'D'<<8 | 'T'），用来标志这是一个硬件抽象层中的硬件设备结构体。

（3）结构体hw_device_t的成员变量close是一个函数指针，它用来关闭一个硬件设备。

注意

硬件抽象层中的硬件设备是由其所在的模块提供接口来打开的，而关闭则是由硬件设备自身提供接口来完成的。

至此，硬件抽象层模块接口的编写规范就介绍完了。接下来，我们就可以为虚拟硬件设备freg编写硬件抽象层模块接口了。

2.3.2　编写硬件抽象层模块接口

每一个硬件抽象层模块在内核中都对应有一个驱动程序，硬件抽象层模块就是通过这些驱动程序来访问硬件设备的，它们是通过读写设备文件来进行通信的。

硬件抽象层中的模块接口源文件一般保存在hardware/libhardware目录中。为了方便起见，我们将虚拟硬件设备freg在硬件抽象层中的模块名称定义为freg，它的目录结构如下：

```
~/Android/hardware/libhardware
----include
     ----hardware
          ----freg.h
Modules
     ----freg
          ----freg.cpp
          ----Android.mk
```

它由三个文件组成，其中，freg.h和freg.cpp是源代码文件，而Android.mk是模块的编译脚本文件。下面我们就分别介绍这三个文件的内容。

freg.h

```
01 #ifndef ANDROID_FREG_INTERFACE_H
02 #define ANDROID_FREG_INTERFACE_H
03
04 #include <hardware/hardware.h>
05
06 __BEGIN_DECLS
07
08 /*定义模块ID*/
09 #define FREG_HARDWARE_MODULE_ID "freg"
10
11 /*定义设备ID*/
12 #define FREG_HARDWARE_DEVICE_ID "freg"
```

```
13
14 /*自定义模块结构体*/
15 struct freg_module_t {
16         struct hw_module_t common;
17 };
18
19 /*自定义设备结构体*/
20 struct freg_device_t {
21         struct hw_device_t common;
22         int fd;
23         int (*set_val)(struct freg_device_t* dev, int val);
24         int (*get_val)(struct freg_device_t* dev, int* val);
25 };
26
27 __END_DECLS
28
29 #endif
```

这个文件中的常量和结构体都是按照硬件抽象层模块编写规范来定义的。宏FREG_HARDWARE_MODULE_ID和FREG_HARDWARE_DEVICE_ID分别用来描述模块ID和设备ID。结构体freg_module_t用来描述自定义的模块结构体，它的第一个成员变量的类型为hw_module_t。结构体freg_device_t用来描述虚拟硬件设备freg，它的第一个成员变量的类型为freg_device_t。此外，结构体freg_device_t还定义了其他三个成员变量，其中，成员变量fd是一个文件描述符，用来描述打开的设备文件/dev/freg，成员变量set_val和get_val是函数指针，它们分别用来写和读虚拟硬件设备freg的寄存器val的内容。

freg.cpp

这是硬件抽象层模块freg的实现文件，我们分段来阅读。

文件首先包含相关头文件并且定义相关结构体变量。

```
01 #define LOG_TAG "FregHALStub"
02
03 #include <hardware/hardware.h>
04 #include <hardware/freg.h>
05
06 #include <fcntl.h>
07 #include <errno.h>
08
09 #include <cutils/log.h>
10 #include <cutils/atomic.h>
11
12 #define DEVICE_NAME "/dev/freg"
13 #define MODULE_NAME "Freg"
14 #define MODULE_AUTHOR "shyluo@gmail.com"
15
16 /*设备打开和关闭接口*/
17 static int freg_device_open(const struct hw_module_t* module, const char* id, struct hw_device_t** device);
18 static int freg_device_close(struct hw_device_t* device);
19
20 /*设备寄存器读写接口*/
21 static int freg_get_val(struct freg_device_t* dev, int* val);
22 static int freg_set_val(struct freg_device_t* dev, int val);
23
24 /*定义模块操作方法结构体变量*/
25 static struct hw_module_methods_t freg_module_methods = {
26         open: freg_device_open
27 };
28
29 /*定义模块结构体变量*/
```

```
30 struct freg_module_t HAL_MODULE_INFO_SYM = {
31         common: {
32                 tag: HARDWARE_MODULE_TAG,
33                 version_major: 1,
34                 version_minor: 0,
35                 id: FREG_HARDWARE_MODULE_ID,
36                 name: MODULE_NAME,
37                 author: MODULE_AUTHOR,
38                 methods: &freg_module_methods,
39         }
40 };
```

在这段代码中，最值得关注的就是模块变量HAL_MODULE_INFO_SYM的定义。按照硬件抽象层模块编写规范，每一个硬件抽象层模块必须导出一个名称为HAL_MODULE_INFO_SYM的符号，它指向一个自定义的硬件抽象层模块结构体，而且它的第一个类型为hw_module_t的成员变量的tag值必须设置为HARDWARE_MODULE_TAG。除此之外，还初始化了这个硬件抽象层模块结构体的版本号、ID、名称、作者和操作方法列表等。

虚拟硬件设备freg的打开和关闭分别由函数freg_device_open和freg_device_close来实现，如下所示。

```
41 static int freg_device_open(const struct hw_module_t* module, const char* id, struct hw_device_t** device) {
42         if(!strcmp(id, FREG_HARDWARE_DEVICE_ID)) {
43                 struct freg_device_t* dev;
44
45                 dev = (struct freg_device_t*)malloc(sizeof(struct freg_device_t));
46                 if(!dev) {
47                         LOGE("Failed to alloc space for freg_device_t.");
48                         return -EFAULT;
49                 }
50
51                 memset(dev, 0, sizeof(struct freg_device_t));
52
53                 dev->common.tag = HARDWARE_DEVICE_TAG;
54                 dev->common.version = 0;
55                 dev->common.module = (hw_module_t*)module;
56                 dev->common.close = freg_device_close;
57                 dev->set_val = freg_set_val;
58                 dev->get_val = freg_get_val;
59
60                 if((dev->fd = open(DEVICE_NAME, O_RDWR)) == -1) {
61                         LOGE("Failed to open device file /dev/freg -- %s.", strerror(errno));
62                         free(dev);
63                         return -EFAULT;
64                 }
65
66                 *device = &(dev->common);
67
68                 LOGI("Open device file /dev/freg successfully.");
69
70                 return 0;
71         }
72
73         return -EFAULT;
74 }
75
76 static int freg_device_close(struct hw_device_t* device) {
77         struct freg_device_t* freg_device = (struct freg_device_t*)device;
78         if(freg_device) {
79                 close(freg_device->fd);
80                 free(freg_device);
81         }
82
83         return 0;
84 }
```

前面提到，一个硬件抽象层模块可能会包含多个硬件设备，而这些硬件设备的打开操作都是由函数freg_device_open来完成的，因此，函数freg_device_open会根据传进来的参数id来判断要打开哪一个硬件设备。

在硬件抽象层模块freg中，只有一个虚拟硬件设备freg，它使用结构体freg_device_t来描述。因此，函数freg_device_open发现参数id与虚拟硬件设备freg的ID值匹配以后，就会分配一个freg_device_t结构体，并且对它的成员变量进行初始化。按照硬件抽象层模块编写规范，硬件抽象层中的硬件设备标签（dev->common.tag）必须设置为HARDWARE_DEVICE_TAG。除此之外，我们还将虚拟硬件设备freg的关闭函数设置为freg_device_close，并且将它的读写函数设置为freg_get_val和freg_set_val。

初始化完成用来描述虚拟硬件设备freg的结构体freg_device_t之后，我们就可以调用open函数来打开虚拟硬件设备文件/dev/freg了，并且将得到的文件描述符保存在结构体freg_device_t的成员变量fd中。

虚拟硬件设备freg的关闭函数freg_device_close的实现比较简单，它主要是关闭设备文件/dev/freg，以及释放设备在打开时所分配的资源。

虚拟硬件设备freg的读写函数freg_get_val和freg_set_val的实现如下所示。

```
85 static int freg_get_val(struct freg_device_t* dev, int* val) {
86         if(!dev) {
87                 LOGE("Null dev pointer.");
88                 return -EFAULT;
89         }
90
91         if(!val) {
92                 LOGE("Null val pointer.");
93                 return -EFAULT;
94         }
95
96         read(dev->fd, val, sizeof(*val));
97
98         LOGI("Get value %d from device file /dev/freg.", *val);
99
100         return 0;
101 }
102
103 static int freg_set_val(struct freg_device_t* dev, int val) {
104         if(!dev) {
105                 LOGE("Null dev pointer.");
106                 return -EFAULT;
107         }
108
109         LOGI("Set value %d to device file /dev/freg.", val);
110         write(dev->fd, &val, sizeof(val));
111
112         return 0;
113 }
```

这两个函数分别通过调用read和write函数来实现读写虚拟硬件设备freg的寄存器val的内容。

Android.mk

```
1 LOCAL_PATH := $(call my-dir)
2 include $(CLEAR_VARS)
3 LOCAL_MODULE_TAGS := optional
4 LOCAL_PRELINK_MODULE := false
5 LOCAL_MODULE_PATH := $(TARGET_OUT_SHARED_LIBRARIES)/hw
6 LOCAL_SHARED_LIBRARIES := liblog
7 LOCAL_SRC_FILES := freg.cpp
8 LOCAL_MODULE := freg.default
9 include $(BUILD_SHARED_LIBRARY)
```

这是硬件抽象层模块freg的编译脚本文件。第9行指定include命令的参数为$(BUILD_SHARED_LIBRARY)，表示要将该硬件抽象层模块编译成一个动态链接库文件，名称为freg.default，并且保存在$(TARGET_OUT_SHARED_LIBRARIES)/hw目录下，即out/target/product/generic/system/lib/hw目录下。

注意

我们将硬件抽象层模块freg对应的文件名称定义为freg.default，编译成功后，系统就会自动在后面加后缀.so，于是就得到了一个freg.default.so文件。根据硬件抽象层模块文件的命名规范，当我们要加载硬件抽象层模块freg时，只需要指定它的ID值，即“freg”，系统就会根据一定的规则成功地找到要加载的freg.default.so文件。

硬件抽象层模块freg的所有文件都准备好之后，我们就可以执行mmm命令对它进行编译和打包了。

```
USER@MACHINE:~/Android$ mmm ./hardware/libhardware/freg/
USER@MACHINE:~/Android$ make snod
```

最终就可以在out/target/product/generic/system/lib/hw目录下得到一个freg.default.so文件。

2.3.3　硬件抽象层模块的加载过程

学习Android硬件抽象层模块的加载过程有助于理解它的编写规范以及实现原理。Android系统中的硬件抽象层模块是由系统统一加载的，当调用者需要加载这些模块时，只要指定它们的ID值就可以了。

在Android硬件抽象层中，负责加载硬件抽象层模块的函数是hw_get_module，它的原型如下：

hardware/libhardware/include/hardware/hardware.h

```
1 /**
2  * Get the module info associated with a module by id.
3  * @return: 0 == success, <0 == error and *pHmi == NULL
4  */
5 int hw_get_module(const char *id, const struct hw_module_t **module);
```

它有id和module两个参数。其中，id是输入参数，表示要加载的硬件抽象层模块ID；module是输出参数，如果加载成功，那么它指向一个自定义的硬件抽象层模块结构体。函数的返回值是一个整数，如果等于0，则表示加载成功；如果小于0，则表示加载失败。

下面我们就开始分析hw_get_module函数的实现。

hardware/libhardware/hardware.c

```
01 /** Base path of the hal modules */
02 #define HAL_LIBRARY_PATH1 "/system/lib/hw"
03 #define HAL_LIBRARY_PATH2 "/vendor/lib/hw"
04
05 static const char *variant_keys[] = {
06     "ro.hardware",  /* This goes first so that it can pick up a different
07                        file on the emulator. */
08     "ro.product.board",
09     "ro.board.platform",
10     "ro.arch"
11 };
12
13 static const int HAL_VARIANT_KEYS_COUNT =
14     (sizeof(variant_keys)/sizeof(variant_keys[0]));
15
16 int hw_get_module(const char *id, const struct hw_module_t **module)
17 {
18     int status;
19     int i;
20     const struct hw_module_t *hmi = NULL;
21     char prop[PATH_MAX];
```

```
22      char path[PATH_MAX];
23
24      /*
25       * Here we rely on the fact that calling dlopen multiple times on
26       * the same .so will simply increment a refcount (and not load
27       * a new copy of the library).
28       * We also assume that dlopen() is thread-safe.
29       */
30
31      /* Loop through the configuration variants looking for a module */
32      for (i=0 ; i<HAL_VARIANT_KEYS_COUNT+1 ; i++) {
33          if (i < HAL_VARIANT_KEYS_COUNT) {
34              if (property_get(variant_keys[i], prop, NULL) == 0) {
35                  continue;
36              }
37
38              snprintf(path, sizeof(path), "%s/%s.%s.so",
39                      HAL_LIBRARY_PATH1, id, prop);
40              if (access(path, R_OK) == 0) break;
41
42              snprintf(path, sizeof(path), "%s/%s.%s.so",
43                       HAL_LIBRARY_PATH2, id, prop);
44              if (access(path, R_OK) == 0) break;
45          } else {
46              snprintf(path, sizeof(path), "%s/%s.default.so",
47                       HAL_LIBRARY_PATH1, id);
48              if (access(path, R_OK) == 0) break;
49          }
50      }
51
52      status = -ENOENT;
53      if (i < HAL_VARIANT_KEYS_COUNT+1) {
54          /* load the module, if this fails, we're doomed, and we should not try
55           * to load a different variant. */
56          status = load(id, path, module);
57      }
58
59      return status;
60  }
```

在前面的2.3.1.1小节中，我们已经见过数组variant_keys的定义了，它用来组装要加载的硬件抽象层模块的文件名称。常量HAL_VARIANT_KEYS_COUNT表示数组variant_keys的大小。

宏HAL_LIBRARY_PATH1和HAL_LIBRARY_PATH2用来定义要加载的硬件抽象层模块文件所在的目录。在前面的2.3.2小节中提到，编译好的模块文件位于out/target/product/generic/system/lib/hw目录中，而这个目录经过打包后，就对应于设备上的/system/lib/hw目录。宏HAL_LIBRARY_PATH2所定义的目录为/vendor/lib/hw，用来保存设备厂商所提供的硬件抽象层模块接口文件。

函数第32行到第50行的for循环根据数组variant_keys在HAL_LIBRARY_PATH1和HAL_LIBRARY_PATH2目录中检查对应的硬件抽象层模块文件是否存在，如果存在，则结束for循环；第56行调用load函数来执行加载硬件抽象层模块的操作。

我们以在Android模块器中加载硬件抽象层模块freg为例，来分析硬件抽象层模块的加载过程。在调用hw_get_module函数加载硬件抽象层模块freg时，传入的参数id的值为FREG_HARDWARE_MODULE_ID，即“freg”。在上面的for循环中，首先找到通过property_get函数获得的系统属性“ro.hardware”的值。在Android模拟器中，这个属性的值定义为“goldfish”，于是通过第38和39两行的snprintf函数，就得到变量path的值为“/system/lib/hw/freg.goldfish.so”。第40行调用access函数判断文件/system/lib/hw/freg.goldfish.so是否存在，如果存在，就跳出循环；否则，再通过第42行到第44行的代码来判断文件/vendor/lib/hw/freg.goldfish.so是否存在，如果存在，那么也会跳出循环，因为要加载的硬件抽象层模

块文件已经找到了。如果这两个文件都不存在，那么按照相同的方法来依次查找数组variant_keys中其他元素所对应的硬件抽象层模块文件是否存在。如果数组variant_keys中的所有元素对应的硬件抽象层模块文件都不存在，那么第46行到第48行的代码就会在“/system/lib/hw”目录中检查是否存在一个freg.default.so文件。如果也不存在，那么硬件抽象层模块freg的加载就失败了。

找到了硬件抽象层模块文件之后，第56行就调用load函数来执行硬件抽象层模块的加载操作，它的实现如下所示。

hardware/libhardware/hardware.c

```
01 static int load(const char *id,
02         const char *path,
03         const struct hw_module_t **pHmi)
04 {
05     int status;
06     void *handle;
07     struct hw_module_t *hmi;
08
09     /*
10      * load the symbols resolving undefined symbols before
11      * dlopen returns. Since RTLD_GLOBAL is not or'd in with
12      * RTLD_NOW the external symbols will not be global
13      */
14     handle = dlopen(path, RTLD_NOW);
15     if (handle == NULL) {
16         char const *err_str = dlerror();
17         LOGE("load: module=%s\n%s", path, err_str?err_str:"unknown");
18         status = -EINVAL;
19         goto done;
20     }
21
22     /* Get the address of the struct hal_module_info. */
23     const char *sym = HAL_MODULE_INFO_SYM_AS_STR;
24     hmi = (struct hw_module_t *)dlsym(handle, sym);
25     if (hmi == NULL) {
26         LOGE("load: couldn't find symbol %s", sym);
27         status = -EINVAL;
28         goto done;
29     }
30
31     /* Check that the id matches */
32     if (strcmp(id, hmi->id) != 0) {
33         LOGE("load: id=%s != hmi->id=%s", id, hmi->id);
34         status = -EINVAL;
35         goto done;
36     }
37
38     hmi->dso = handle;
39
40     /* success */
41     status = 0;
42
43     done:
44     if (status != 0) {
45         hmi = NULL;
46         if (handle != NULL) {
47             dlclose(handle);
48             handle = NULL;
49         }
50     } else {
51         LOGV("loaded HAL id=%s path=%s hmi=%p handle=%p",
52                 id, path, *pHmi, handle);
53     }
54
```

```
55      *pHmi = hmi;
56
57      return status;
58  }
```

前面提到，硬件抽象层模块文件实际上是一个动态链接库文件，即so文件。因此，第14行调用dlopen函数将它加载到内存中。加载完成这个动态链接库文件之后，第24行就调用dlsym函数来获得里面名称为HAL_MODULE_INFO_SYM_AS_STR的符号。这个HAL_MODULE_INFO_SYM_AS_STR符号指向的是一个自定义的硬件抽象层模块结构体，它包含了对应的硬件抽象层模块的所有信息。HAL_MODULE_INFO_SYM_AS_STR是一个宏，它的值定义为“HMI”。根据硬件抽象层模块的编写规范，每一个硬件抽象层模块都必须包含一个名称为“HMI”的符号，而且这个符号的第一个成员变量的类型必须定义为hw_module_t，因此，第24行可以安全地将模块中的HMI符号转换为一个hw_module_t结构体指针。

获得了这个hw_module_t结构体指针之后，第32行调用strcmp函数来验证加载得到的硬件抽象层模块ID是否与所要求加载的硬件抽象层模块ID一致。如果不一致，就说明出错了，函数返回一个错误值-EINVAL。最后，第38行将成功加载后得到的模块句柄值handle保存在hw_module_t结构体指针hmi的成员变量dso中，然后将它返回给调用者。

至此，硬件抽象层模块的加载过程就介绍完了。接下来，我们分析在硬件抽象层模块的加载过程中，经常会碰到的一个权限问题，即无法调用open函数打开对应的设备文件。

2.3.4 处理硬件设备访问权限问题

在硬件抽象层模块中，我们是调用open函数来打开对应的设备文件的。例如，在2.3.2小节中开发的硬件抽象层模块freg中，函数freg_device_open调用open函数来打开设备文件/dev/freg。

```
60 if((dev->fd = open(DEVICE_NAME, O_RDWR)) == -1) {
61        LOGE("Failed to open device file /dev/freg -- %s.", strerror(errno));
62        free(dev);
63        return -EFAULT;
64 }
```

如果不修改设备文件/def/freg的访问权限，那么应用程序调用freg_device_open函数打开设备文件/def/freg就会失败，从第61行的日志输出可以看到下面的内容：

```
Failed to open /dev/hello -- Permission denied.
```

这表示当前用户没有权限打开设备文件/dev/freg。在默认情况下，只有root用户才有权限访问系统的设备文件。由于一般的应用程序是没有root用户权限的，因此，这里就会提示没有权限打开设备文件/dev/freg。

解决这个问题的办法是，赋予root之外的其他用户访问设备文件/dev/freg的权限。我们知道，在Linux系统中，可以通过udev规则在系统启动时修改设备文件的访问权限[3]。然而，Android系统并没有实现udev机制，因此，我们就不可以通过定义udev规则来赋予root之外的其他用户访问设备文件/dev/freg的权限。不过，Android提供了另外的一个uevent机制，可以在系统启动时修改设备文件的访问权限。

在system/core/rootdir目录下有一个名为ueventd.rc的配置文件，我们可以在里面增加以下一行内容

3 udev是Linux 2.6内核新增的一个功能，用来替代原来的devfs，是Linux系统默认的设备管理工具。udev 机制以守护进程的形式运行，通过侦听内核发出来的uevent来管理/dev目录下的设备文件，包括添加或者删除设备文件、修改设备文件的访问权限等。

来修改设备文件/dev/freg的访问权限。

```
/dev/freg                 0666   root       root
```

这表示所有的用户均可以访问设备文件/dev/freg，即可以打开设备文件/dev/freg，以及读写它的内容。这样，除了root用户之外，系统中的其他用户也可以调用freg_device_open函数来打开设备文件/dev/freg。

修改了ueventd.rc文件后，需要重新编译Android源代码工程，这样新修改的设备文件/dev/freg的访问权限才能生效。这里，我们介绍一种不必重新编译Android源代码工程就可以使得修改后的设备文件/dev/freg的访问权限生效的方法。

在编译Android源代码工程时，文件system/core/rootdir/ueventd.rc会被拷贝到out/target/product/generic/root目录下，并且最终打包在ramdisk.img镜像文件中。当Android系统启动时，会把ramdisk.img镜像文件中的ueventd.rc文件安装在设备根目录中，并且由init进程来解析它的内容和修改相应的设备文件的访问权限。因此，只要我们能够修改ramdisk.img镜像文件中ueventd.rc文件的内容，就可以修改设备文件/dev/freg的访问权限。接下来就详细介绍修改ramdisk.img镜像文件中ueventd.rc文件的方法。

1. 解压ramdisk.img镜像文件

镜像文件ramdisk.img是一个gzip文件，因此，我们可以执行gunzip命令对它进行解压。

```
USER@MACHINE:~/Android$ mv ./out/target/product/generic/ramdisk.img ./ramdisk.img.gz
USER@MACHINE:~/Android$ gunzip ./ramdisk.img.gz
```

我们先将ramdisk.img改名为ramdisk.img.gz，然后调用gunzip命令对它进行解压。解压后得到的ramdisk.img文件保存在~/Android目录中。

2. 还原ramdisk.img镜像文件

解压后得到的ramdisk.img文件是一个cpio[4]格式的归档文件，因此，我们可以执行cpio命令对它解除归档。

```
USER@MACHINE:~/Android $ mkdir ramdisk
USER@MACHINE:~/Android $ cd ./ramdisk/
USER@MACHINE:~/Android /ramdisk$ cpio -i -F ../ramdisk.img
```

解除归档后得到的文件保存在~/Android/ramdisk目录中。

3. 修改ueventd.rc文件

进入到~/Android/ramdisk目录中，找到ueventd.rc文件，并且往里面增加以下一行内容：

```
/dev/freg                 0666   root       root
```

这一行内容赋予了系统中的所有用户访问设备文件/dev/freg的权限。

4. 重新打包ramdisk.img镜像文件

重新打包ramdisk.img镜像文件的过程其实就是第1步和第2步的逆过程，即先把ramdisk目录归档成cpio文件，然后压缩成gzip文件。

```
USER@MACHINE:~/Android/ramdisk$ rm -f ../ramdisk.img
USER@MACHINE:~/Android/ramdisk$ find . | cpio -o -H newc > ../ramdisk.img.unzip
USER@MACHINE:~/Android/ramdisk$ cd ..
USER@MACHINE:~/Android$ gzip -c ./ramdisk.img.unzip > ./ramdisk.img.gz
```

4　cpio是一种包含其他文件和有关信息的归档文件，具体可以参考http://www.gnu.org/software/cpio/。

```
USER@MACHINE:~/Android$ rm -f ./ramdisk.img.unzip
USER@MACHINE:~/Android$ rm -R ./ramdisk
USER@MACHINE:~/Android$ mv ./ramdisk.img.gz ./out/target/product/generic/ramdisk.img
```

这样，重新打包后得到的ramdisk.img镜像文件中的ueventd.rc文件就修改好了，系统在启动之后就会通过init进程来赋予系统中的所有用户访问设备文件/dev/freg的权限。

2.4 开发Android硬件访问服务

开发好硬件抽象层模块之后，我们通常还需要在应用程序框架层中实现一个硬件访问服务。硬件访问服务通过硬件抽象层模块来为应用程序提供硬件读写操作。由于硬件抽象层模块是使用C++语言开发的，而应用程序框架层中的硬件访问服务是使用Java语言开发的，因此，硬件访问服务必须通过Java本地接口（Java Native Interface，JNI）来调用硬件抽象层模块的接口。

Android系统的硬件访问服务通常运行在系统进程System[5]中，而使用这些硬件访问服务的应用程序运行在另外的进程中，即应用程序需要通过进程间通信机制来访问这些硬件访问服务。Android系统提供了一种高效的进程间通信机制——Binder进程间通信机制[6]，应用程序就是通过它来访问运行在系统进程System中的硬件访问服务的。Binder进程间通信机制要求提供服务的一方必须实现一个具有跨进程访问能力的服务接口，以便使用服务的一方可以通过这个服务接口来访问它。因此，在实现硬件访问服务之前，我们首先要定义它的服务接口。

在本节接下来的内容中，我们将为在2.1小节中介绍的虚拟硬件设备freg开发一个硬件访问服务FregService，它所实现的硬件访问服务接口为IFregService。首先介绍硬件访问服务接口IFregService的定义，然后介绍硬件访问服务FregService的实现，包括它的内部实现以及调用硬件抽象层模块的JNI实现，最后介绍如何在系统进程System中启动硬件访问服务FregService。

2.4.1 定义硬件访问服务接口

Android系统提供了一种描述语言来定义具有跨进程访问能力的服务接口，这种描述语言称为Android接口描述语言（Android Interface Definition Language，AIDL）。以AIDL定义的服务接口文件是以aidl为后缀名的，在编译时，编译系统会将它们转换成一个Java文件，然后再对它们进行编译。在本节中，我们将使用AIDL来定义硬件访问服务接口IFregService。

在Android系统中，通常把硬件访问服务接口定义在frameworks/base/core/java/android/os目录中，因此，我们把定义了硬件访问服务接口IFregService的文件IFregService.aidl也保存在这个目录中，它的内容如下所示。

frameworks/base/core/java/android/os/IFregService.aidl

```
1 package android.os;
2
3 interface IFregService {
4         void setVal(int val);
5         int getVal();
6 }
```

IFregService服务接口只定义了两个成员函数，它们分别是setVal和getVal。其中，成员函数setVal用来往虚拟硬件设备freg的寄存器val中写入一个整数，而成员函数getVal用来从虚拟硬件设备freg的寄

5 第11章将详细分析Android系统的系统进程System。

6 第5章将详细分析Android系统的Binder进程间通信机制。

存器val中读出一个整数。

由于服务接口IFregService是使用AIDL语言描述的，因此，我们需要将其添加到编译脚本文件中，这样编译系统才能将其转换为Java文件，然后再对它进行编译。进入到frameworks/base目录中，打开里面的Android.mk文件，修改LOCAL_SRC_FILES变量的值。

```
LOCAL_SRC_FILES += \
    ......
    voip/java/android/net/sip/ISipService.aidl \
    core/java/android/os/IFregService.aidl
```

修改这个编译脚本文件之后，我们就可以使用mmm命令对硬件访问服务接口IFregService进行编译了。

```
USER@MACHINE:~/Android$ mmm ./frameworks/base/
```

编译后得到的framework.jar文件就包含有IFregService接口，它继承了android.os.IInterface接口。在IFregService接口内部，定义了一个Binder本地对象类Stub，它实现了IFregService接口，并且继承了android.os.Binder类。此外，在IFregService.Stub类内部，还定义了一个Binder代理对象类Proxy，它同样也实现了IFregService接口。

前面提到，用AIDL定义的服务接口是用来进行进程间通信的，其中，提供服务的进程称为Server进程，而使用服务的进程称为Client进程。在Server进程中，每一个服务都对应有一个Binder本地对象，它通过一个桩（Stub）来等待Client进程发送进程间通信请求。Client进程在访问运行Server进程中的服务之前，首先要获得它的一个Binder代理对象接口（Proxy），然后通过这个Binder代理对象接口向它发送进程间通信请求。

在接下来的2.4.2小节中，我们就将硬件访问服务FregService从IFregService.Stub类继承下来，并且实现IFregService接口的成员函数setVal和getVal。在2.5小节中，我们再介绍如何在应用程序中获得硬件访问服务FregService的一个Binder代理对象接口，即IFregService.Stub.Proxy接口。

2.4.2　实现硬件访问服务

在Android系统中，通常把硬件访问服务实现在frameworks/base/services/java/com/android/server目录中。因此，我们把实现了硬件访问服务FregService的FregService.java文件也保存在这个目录中，它的内容如下所示。

frameworks/base/services/java/com/android/server/FregService.java

```
01 package com.android.server;
02
03 import android.content.Context;
04 import android.os.IFregService;
05 import android.util.Slog;
06
07 public class FregService extends IFregService.Stub {
08         private static final String TAG = "FregService";
09
10         private int mPtr = 0;
11
12         FregService() {
13                 mPtr = init_native();
14
15                 if(mPtr == 0) {
16                         Slog.e(TAG, "Failed to initialize freg service.");
17                 }
18         }
```

```
19
20          public void setVal(int val) {
21                  if(mPtr == 0) {
22                          Slog.e(TAG, "Freg service is not initialized.");
23                          return;
24                  }
25
26                  setVal_native(mPtr, val);
27          }
28
29          public int getVal() {
30                  if(mPtr == 0) {
31                          Slog.e(TAG, "Freg service is not initialized.");
32                          return 0;
33                  }
34
35                  return getVal_native(mPtr);
36          }
37
38          private static native int init_native();
39          private static native void setVal_native(int ptr, int val);
40          private static native int getVal_native(int ptr);
41  };
```

硬件访问服务FregService继承了IFregService.Stub类，并且实现了IFregService接口的成员函数setVal和getVal。其中，成员函数setVal通过调用JNI方法setVal_native来写虚拟硬件设备freg的寄存器val，而成员函数getVal调用JNI方法getVal_native来读虚拟硬件设备freg的寄存器val。此外，硬件访问服务FregService在启动时，会通过调用JNI方法init_native来打开虚拟硬件设备freg，并且获得它的一个句柄值，保存在成员变量mPtr中。如果硬件访问服务FregService打开虚拟硬件设备freg失败，那么它的成员变量mPtr的值就等于0；否则，就得到一个大于0的句柄值。这个句柄值实际上是指向虚拟硬件设备freg在硬件抽象层中的一个设备对象，硬件访问服务FregService的成员函数 setVal和getVal在访问虚拟硬件设备freg的寄存器val时，必须要指定这个句柄值，以便硬件访问服务FregService的JNI实现可以知道它所要访问的是哪一个硬件设备。

硬件访问服务FregService编写完成之后，就可以执行mmm命令来重新编译Android系统的services模块了。

```
USER@MACHINE:~/Android$ mmm ./frameworks/base/services/java/
```

编译后得到的services.jar文件就包含有FregService类。下面我们继续介绍硬件访问服务FregService的JNI实现。

2.4.3 实现硬件访问服务的JNI方法

在Android系统中，通常把硬件访问服务的JNI方法实现在frameworks/base/services/jni目录中，因此，我们把实现了硬件访问服务FregService的JNI方法的com_android_server_FregService.cpp文件也保存在这个目录中，它的内容如下所示。

frameworks/base/services/jni/com_android_server_FregService.cpp

```
01 #define LOG_TAG "FregServiceJNI"
02
03 #include "jni.h"
04 #include "JNIHelp.h"
05 #include "android_runtime/AndroidRuntime.h"
06
07 #include <utils/misc.h>
```

```
08 #include <utils/Log.h>
09 #include <hardware/hardware.h>
10 #include <hardware/freg.h>
11
12 #include <stdio.h>
13
14 namespace android
15 {
16         /*设置虚拟硬件设备freg的寄存器的值*/
17         static void freg_setVal(JNIEnv* env, jobject clazz, jint ptr, jint value) {
18                 /*将参数ptr转换为freg_device_t结构体变量*/
19                 freg_device_t* device = (freg_device_t*)ptr;
20                 if(!device) {
21                         LOGE("Device freg is not open.");
22                         return;
23                 }
24
25                 int val = value;
26
27                 LOGI("Set value %d to device freg.", val);
28
29                 device->set_val(device, val);
30         }
31
32         /*读取虚拟硬件设备freg的寄存器的值*/
33         static jint freg_getVal(JNIEnv* env, jobject clazz, jint ptr) {
34                 /*将参数ptr转换为freg_device_t结构体变量*/
35                 freg_device_t* device = (freg_device_t*)ptr;
36                 if(!device) {
37                         LOGE("Device freg is not open.");
38                         return 0;
39                 }
40
41                 int val = 0;
42
43                 device->get_val(device, &val);
44
45                 LOGI("Get value %d from device freg.", val);
46
47                 return val;
48         }
49
50         /*打开虚拟硬件设备freg*/
51         static inline int freg_device_open(const hw_module_t* module, struct freg_device_t** device) {
52                 return module->methods->open(module, FREG_HARDWARE_DEVICE_ID, (struct hw_device_t**)device);
53         }
54
55         /*初始化虚拟硬件设备freg*/
56         static jint freg_init(JNIEnv* env, jclass clazz) {
57                 freg_module_t* module;
58                 freg_device_t* device;
59
60                 LOGI("Initializing HAL stub freg......");
61
62                 /*加载硬件抽象层模块freg*/
63                 if(hw_get_module(FREG_HARDWARE_MODULE_ID, (const struct hw_module_t**)&module) == 0) {
64                         LOGI("Device freg found.");
65
66                                 /*打开虚拟硬件设备freg*/
67                                 if(freg_device_open(&(module->common), &device) == 0) {
68                                   LOGI("Device freg is open.");
69
70                                   /*将freg_device_t接口转换为整型值返回*/
```

```
71                                  return (jint)device;
72                          }
73
74                          LOGE("Failed to open device freg.");
75                          return 0;
76                  }
77
78                  LOGE("Failed to get HAL stub freg.");
79
80                  return 0;
81          }
82
83          /*Java本地接口方法表*/
84          static const JNINativeMethod method_table[] = {
85                  {"init_native", "()I", (void*)freg_init},
86                  {"setVal_native", "(II)V", (void*)freg_setVal},
87                  {"getVal_native", "(I)I", (void*)freg_getVal},
88          };
89
90          /*注册Java本地接口方法*/
91          int register_android_server_FregService(JNIEnv *env) {
92                      return jniRegisterNativeMethods(env, "com/android/server/FregService",
method_table, NELEM(method_table));
93          }
94 };
```

在函数freg_init中，首先通过Android硬件抽象层提供的hw_get_module函数来加载模块ID为FREG_HARDWARE_MODULE_ID的硬件抽象层模块。在2.3.3小节中，我们已经介绍过hw_get_module函数是如何根据FREG_HARDWARE_MODULE_ID来加载Android硬件抽象层模块freg的了。函数hw_get_module最终返回一个hw_module_t接口给freg_init函数，这个hw_module_t接口实际指向的是自定义的一个硬件抽象层模块对象，即一个freg_module_t对象。

函数freg_init接着调用函数freg_device_open来打开设备ID为FREG_HARDWARE_DEVICE_ID的硬件设备，而后者又是通过调用前面获得的hw_module_t接口的操作方法列表中的open函数来打开指定的硬件设备的。在2.3.2小节中，我们将硬件抽象层模块freg的操作方法列表中的open函数设置为freg_device_open，这个函数最终返回一个freg_device_t接口给freg_init函数。

函数freg_init最后把获得的freg_device_t接口转换成一个整型句柄值，然后返回给调用者。

函数freg_setVal和freg_getVal都是首先把参数ptr转换为一个freg_device_t接口，然后分别调用它的成员函数set_val和get_val来访问虚拟硬件设备freg的寄存器val的值。

注意

在调用freg_setVal和freg_getVal这两个JNI方法之前，调用者首先要调用JNI方法freg_init打开虚拟硬件设备freg，以便可以获得一个freg_device_t接口。

文件接着定义了一个JNI方法表method_table，分别将函数freg_init、freg_setVal和freg_getVal的JNI方法注册为init_native、setVal_native和getVal_native。文件最后调用了jniRegisterNativeMethods函数把JNI方法表method_table注册到Java虚拟机中，以便提供给硬件访问服务FregService类使用。

硬件访问服务FregService的JNI方法编写完成之后，我们还需要修改frameworks/base/services/jni目录下的onload.cpp文件，在里面增加register_android_server_FregService函数的声明和调用。

frameworks/base/services/jni/onload.cpp

```
01 #include "JNIHelp.h"
02 #include "jni.h"
03 #include "utils/Log.h"
04 #include "utils/misc.h"
```

```
05
06 namespace android {
07 ......
08
09 int register_android_server_FregService(JNIEnv* env);
10 };
11
12 using namespace android;
13
14 extern "C" jint JNI_OnLoad(JavaVM* vm, void* reserved)
15 {
16     ......
17
18     register_android_server_FregService(env);
19
20     return JNI_VERSION_1_4;
21 }
```

onload.cpp文件实现在libandroid_servers模块中。当系统加载libandroid_servers模块时，就会调用实现在onload.cpp文件中的JNI_OnLoad函数。这样，就可以将前面定义的三个JNI方法init_native、setVal_native和getVal_native注册到Java虚拟机中。

最后，进入到frameworks/base/services/jni目录中，打开里面的Android.mk文件，修改变量LOCAL_SRC_FILES的值。

```
LOCAL_SRC_FILES += \
    ......
    com_android_server_FregService.cpp \
    onload.cpp
```

修改好这个文件之后，我们就可以执行mmm命令来重新编译libandroid_servers模块了。

```
USER@MACHINE:~/Android$ mmm ./frameworks/base/services/jni/
```

编译后得到的libandroid_servers.so文件就包含有init_native、setVal_native和getVal_native这三个JNI方法了。

至此，硬件访问服务FregService的实现就介绍完了。下面我们继续介绍如何在系统进程System中启动它。

2.4.4　启动硬件访问服务

前面提到，Android系统的硬件访问服务通常是在系统进程System中启动的，而系统进程System是由应用程序孵化器进程Zygote[7]负责启动的。由于应用程序孵化器进程Zygote是在系统启动时启动的，因此，把硬件访问服务运行在系统进程System中，就实现了开机时自动启动。

在本节中，我们把硬件访问服务FregService运行在系统进程System中，因此，进入到frameworks/base/services/java/com/android/server目录中，打开里面的SystemServer.java文件，修改ServerThread类的成员函数run的实现，如下所示。

```
01 class ServerThread extends Thread {
02         ......
03
04         @Override
05           public void run() {
06                 ......
07
```

7　第11章将详细分析应用程序孵化器进程Zygote。

```
08                  if (factoryTest != SystemServer.FACTORY_TEST_LOW_LEVEL) {
09                          ......
10
11                          try {
12                              Slog.i(TAG, "Freg Service");
13                              ServiceManager.addService("freg", new FregService());
14                            } catch (Throwable e) {
15                              Slog.e(TAG, "Failure starting Freg Service", e);
16                            }
17                  }
18
19                  ......
20          }
21
22          ......
23  }
```

系统进程System在启动时，会创建一个ServerThread线程来启动系统中的关键服务，其中就包括一些硬件访问服务。在ServerThread类的成员函数run中，首先创建一个FregService实例，然后把它注册到Service Manager中。Service Manager是Android系统的Binder进程间通信机制的一个重要角色，它负责管理系统中的服务对象。注册到Service Manager中的服务对象都有一个对应的名称，使用这些服务的Client进程就是通过这些名称来向Service Manager请求它们的Binder代理对象接口的，以便可以访问它们所提供的服务。硬件访问服务FregService注册到Service Manager之后，它的启动过程就完成了。

最后，我们需要执行mmm命令来重新编译services模块。

```
USER@MACHINE:~/Android$ mmm ./frameworks/base/services/java/
```

编译后得到的services.jar文件就包含有硬件访问服务FregService，并且在系统启动时，将它运行在系统进程System中。

至此，硬件访问服务FregService就完全实现好了。我们可以执行make snod命令来重新打包Android系统镜像文件system.img。

```
USER@MACHINE:~/Android$ make snod
```

在接下来的2.5小节中，我们将开发一个Android应用程序来访问虚拟硬件设备freg的寄存器val的值，这是通过调用硬件访问服务FregService的成员函数setVal和getVal实现的。

2.5 开发Android应用程序来使用硬件访问服务

在Android应用程序框架层开发硬件访问服务的目的是为了让上层的Android应用程序能够访问对应的硬件设备。在本节中，我们将在Android源代码工程环境中开发一个应用程序Freg，它通过硬件访问服务FregService来访问虚拟硬件设备freg的寄存器val的值。

由于这个应用程序是实验性质的，因此，我们将它放在packages/experimental目录中，对应的工程为Freg。它的目录结构如下：

```
~/Android/packages/experimental/Freg
----AndroidManifest.xml
----Android.mk
----src
    ----shy/luo/freg
        ----Freg.java
----res
    ----layout
        ----main.xml
```

```
----values
    ----strings.xml
----drawable
    ----icon.png
```

它包含一个源代码目录src、一个资源目录res、一个配置文件AndroidManifest.xml和一个编译脚本文件Android.mk。下面我们就分别介绍每一个文件的内容。

Freg.java

```
01 package shy.luo.freg;
02
03 import android.app.Activity;
04 import android.os.ServiceManager;
05 import android.os.Bundle;
06 import android.os.IFregService;
07 import android.os.RemoteException;
08 import android.util.Log;
09 import android.view.View;
10 import android.view.View.OnClickListener;
11 import android.widget.Button;
12 import android.widget.EditText;
13
14 public class Freg extends Activity implements OnClickListener {
15         private final static String LOG_TAG = "shy.luo.freg.FregActivity";
16
17          private IFregService fregService = null;
18
19          private EditText valueText = null;
20          private Button readButton = null;
21          private Button writeButton = null;
22          private Button clearButton = null;
23
24          @Override
25          public void onCreate(Bundle savedInstanceState) {
26                    super.onCreate(savedInstanceState);
27                    setContentView(R.layout.main);
28
29                    fregService = IFregService.Stub.asInterface(
30                              ServiceManager.getService("freg"));
31
32                    valueText = (EditText)findViewById(R.id.edit_value);
33                    readButton = (Button)findViewById(R.id.button_read);
34                    writeButton = (Button)findViewById(R.id.button_write);
35                    clearButton = (Button)findViewById(R.id.button_clear);
36
37                    readButton.setOnClickListener(this);
38                    writeButton.setOnClickListener(this);
39                    clearButton.setOnClickListener(this);
40
41                    Log.i(LOG_TAG, "Freg Activity Created");
42          }
43
44          @Override
45          public void onClick(View v) {
46                    if(v.equals(readButton)) {
47                              try {
48                              int val = fregService.getVal();
49                              String text = String.valueOf(val);
50                              valueText.setText(text);
51                         } catch (RemoteException e) {
52                              Log.e(LOG_TAG, "Remote Exception while reading value from freg service.");
53                              }
```

```
54                      } else if(v.equals(writeButton)) {
55                              try {
56                              String text = valueText.getText().toString();
57                              int val = Integer.parseInt(text);
58                              fregService.setVal(val);
59                          } catch (RemoteException e) {
60                              Log.e(LOG_TAG, "Remote Exception while writing value to freg service.");
61                          }
62                      } else if(v.equals(clearButton)) {
63                              String text = "";
64                              valueText.setText(text);
65                      }
66          }
67 }
```

文件定义了一个Activity组件Freg，它是应用程序Freg的主界面。在Activity组件Freg的界面上，有一个编辑框和三个按钮Read、Write和Clear，其中，编辑框用来显示或者输入虚拟硬件设备freg的寄存器val的值；按钮Read和Write分别用来读写虚拟硬件设备freg的寄存器，而按钮Clear用来清空编辑框。

在Activity组件Freg的成员函数onCreate中，第30行通过Service Manager获得一个名称为“freg”的服务的Binder代理对象接口。从2.4.4小节的内容可以知道，这个服务就对应于运行在系统进程System中的硬件访问服务FregService。因此，第29行就可以安全地将这个Binder代理对象接口转换为一个FregService代理对象接口，并且保存在Activity组件Freg的成员变量fregService中。有了这个FregService代理对象接口之后，应用程序Freg就可以通过调用它的成员函数setVal和getVal来访问虚拟硬件设备freg的寄存器val的值了，如Activity组件Freg的成员函数onClick所示。

main.xml

```
01 <?xml version="1.0" encoding="utf-8"?>
02 <LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
03     android:orientation="vertical"
04     android:layout_width="fill_parent"
05     android:layout_height="fill_parent"
06     >
07     <LinearLayout
08         android:layout_width="fill_parent"
09         android:layout_height="wrap_content"
10         android:orientation="vertical"
11         android:gravity="center">
12         <TextView
13                 android:layout_width="wrap_content"
14                 android:layout_height="wrap_content"
15                 android:text="@string/value">
16         </TextView>
17         <EditText
18                 android:layout_width="fill_parent"
19                 android:layout_height="wrap_content"
20                 android:id="@+id/edit_value"
21                 android:hint="@string/hint">
22         </EditText>
23     </LinearLayout>
24     <LinearLayout
25         android:layout_width="fill_parent"
26         android:layout_height="wrap_content"
27         android:orientation="horizontal"
28         android:gravity="center">
29         <Button
30                 android:id="@+id/button_read"
31                 android:layout_width="wrap_content"
32                 android:layout_height="wrap_content"
```

```
33                android:text="@string/read">
34        </Button>
35        <Button
36                android:id="@+id/button_write"
37                android:layout_width="wrap_content"
38                android:layout_height="wrap_content"
39                android:text="@string/write">
40        </Button>
41        <Button
42                android:id="@+id/button_clear"
43                android:layout_width="wrap_content"
44                android:layout_height="wrap_content"
45                android:text="@string/clear">
46        </Button>
47    </LinearLayout>
48 </LinearLayout>
```

这是应用程序Freg的主界面配置文件，在屏幕中显示一个TextView控件和三个Button控件。

strings.xml

```
1 <?xml version="1.0" encoding="utf-8"?>
2 <resources>
3     <string name="app_name">Freg</string>
4     <string name="value">Value</string>
5     <string name="hint">Please input a value...</string>
6     <string name="read">Read</string>
7     <string name="write">Write</string>
8     <string name="clear">Clear</string>
9 </resources>
```

这是应用程序Freg的字符串资源文件，定义了在应用程序中使用到的各个字符串。

icon.png

这是应用程序Freg的图标文件，可以根据需要来放置不同的图片文件。

AndroidManifest.xml

```
01 <?xml version="1.0" encoding="utf-8"?>
02 <manifest xmlns:android="http://schemas.android.com/apk/res/android"
03       package="shy.luo.freg"
04       android:versionCode="1"
05       android:versionName="1.0">
06       <application android:icon="@drawable/icon" android:label="@string/app_name">
07            <activity android:name=".Freg"
08                 android:label="@string/app_name">
09                 <intent-filter>
10                      <action android:name="android.intent.action.MAIN" />
11                      <category android:name="android.intent.category.LAUNCHER" />
12                 </intent-filter>
13            </activity>
14       </application>
15 </manifest>
```

这是应用程序Freg的配置文件，由于应用程序Freg定义了一个Activity组件Freg，因此，要在这个配置文件中对它进行配置。

Android.mk

```
1 LOCAL_PATH:= $(call my-dir)
2 include $(CLEAR_VARS)
3 LOCAL_MODULE_TAGS := optional
```

```
4 LOCAL_SRC_FILES := $(call all-subdir-java-files)
5 LOCAL_PACKAGE_NAME := Freg
6 include $(BUILD_PACKAGE)
```

这是应用程序Freg的编译脚本文件，指定程序的名称为“Freg”。

在应用程序Freg的各个文件都准备好以后，就可以对它进行编译和打包了。

```
USER@MACHINE:~/Android$ mmm ./packages/experimental/Freg/
USER@MACHINE:~/Android$ make snod
```

打包后得到的Android系统镜像文件system.img就包含有应用程序Freg了。

最后，我们使用这个新的Android系统镜像文件system.img来启动Android模拟器。

```
USER@MACHINE:~/Android$ emulator -kernel kernel/goldfish/arch/arm/boot/zImage
```

Android模拟器运行起来之后，我们就可以在应用程序启动器中启动应用程序Freg了，它的界面如图2-2所示。

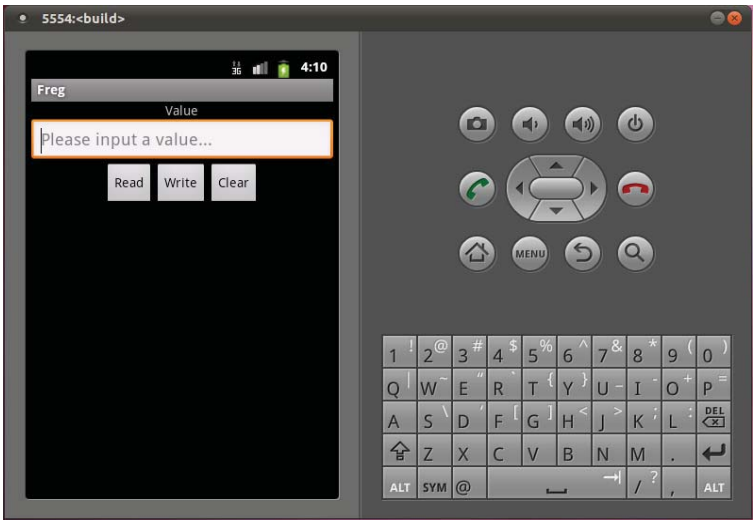


图2-2　应用程序Freg的界面

应用程序Freg启动起来之后，通过点击Read按钮，就可以通过硬件访问服务FregService来读取虚拟硬件设备freg的寄存器val的内容了，而通过在编辑框中输入一个整数值，并且点击Write按钮，就可以通过硬件访问服务FregService将这个整数值写入到虚拟硬件设备freg的寄存器val中了。