《ANDROID 多媒体编程从初学到精通》

作者：华清远见

第 3 章　多核通信

合抱之木，生于毫末。 -- 《老子•道德经》

在智能手机平台中，为了保证多媒体业务在平台上的流畅运行，多采用专用的处理器来处理多媒体业务。如在 Qucalcomm MSM 7K 平台上，就包含了 4 个处理器内核，Qucalcomm MSM 7K 平台采用 ARM 9（mARM，modem ARM）处理基带业务；采用 mDSP（Modem DSP）来处理协议栈业务；采用 ARM 11（aARM，application ARM）来负责 Linux 操作系统的运行；采用 aDSP（Application DSP）来处理多媒体业务方面的编/解码加速工作。

在 2010 年末，智能终端的 CPU 得到了快速发展，高端的智能终端已经采用了 Cortex-A8 的双核处理器。而在平板电脑上，Nvidia 开发的基于 Cortex-A9 的双核处理器 Tegra 2 则成了市场的宠儿。

通信离不开内存的操作，在 Qucalcomm 平台上，内存一般分为 3 种：基带内存（Modem Memory）、应用内存（Application Memory）和共享内存，其中系统 MPU 保护基带内存不被 aARM 接入，ARM MMU 保护应用内存不被 mARM 接入。在本章中，将主要介绍基于共享内存的多核通信。

3.1 共享内存

在 Linux 中，实现进程通信的机制有很多种，如信号、管道、信号量、消息队列、共享内存和套接字等，但共享内存的方式效率最高。

在 Aurora 中，共享内存是多核通信的物理基础，其实现主要包括 3 个部分：共享内存驱动（SMD，Shared Memory Driver）、共享内存状态机（SMSM，Shared Memory State Machine）和共享内存管理器（SMEM，Shared Memory Manager）。其中 SMD 用于多核之间的数据通信；SMSM 用于多核之间的状态通信；SMEM 是一个底层的协议，是物理 RAM 共享内存的管理接口，是 SMD 和 SMSM 的基础。

SMEM 具有两种分配模式：动态 SMEM 和静态 SMEM，动态 SMEM 根据需要实时分配，静态 SMEM 则会预先分配。SMEM 的主要接口为：smem_alloc()、smem_find()、smem_init()等。

SMEM、SMD、SMSM 的实现都需要硬件平台厂商提供支持。

3.1.1 同步与互斥

在 Aurora 中，共享内存用到了自旋锁和互斥锁的概念。

自旋锁是 Linux 内核的同步机制之一，与互斥锁类似，但自旋锁使用者一般保持锁的时间非常短。自旋锁的效率远高于互斥锁。自旋锁的定义位于 aurora\msm\工程 msm\include\linux\spinlock_*.h 文件中。在 Aurora 中，并未引入在 Kernel 2.6.25 中才引入的排队自旋锁（FIFO Ticket Spinlock）概念。排队自旋锁（FIFO Ticket Spinlock）通过保存执行线程申请锁的顺序信息，可以解决传统自旋锁的“不公平”问题，在设计之初仅支持 X86 架构。

自旋锁的定义如下：

```
typedef struct {
    volatile unsigned int lock; //无符号整数
} raw_spinlock_t;
```

lock 虽然被定义为无符号整数，但是实际上被当做有符号整数使用。slock 值为 1 代表锁未被占用，值为 0 或负数代表锁被占用。初始化时 slock 被置为 1。

与信号量和读写信号量导致调用者睡眠不同，自旋锁不会引起调用者睡眠。如果自旋锁已被别的执行单元保持，调用者就一直循环以确定是否该自旋锁的保持者已经释放了锁。

由于自旋锁适用的访问共享资源的时间非常短， 导致自旋锁通常应用于中断上下文访问和对共享资源访问文件非常短的场景中，如果被保护的共享资源在进程上下文访问，则应使用信号量。

与信号量和读写信号量在保持期间可以被抢占的情况不同，自旋锁在保持期间是抢占失效的，自旋锁只有在内核可抢占或 SMP（Symmetrical Multi-Processing）的情况下才真正需要，在单 CPU 且不可抢占的内核下，自旋锁的所有操作都是空操作。

由于智能终端平台上通常存在多个 CPU 或 DSP，自旋锁的运用就显得非常重要。在 Aurora 的 SMD 和 SMSM 的实现上，自旋锁主要运用于中断处理、信道列表和信道状态的变更过程中，自旋锁的定义如下：

```
static DEFINE_SPINLOCK(smd_lock);
static DEFINE_SPINLOCK(smem_lock);
```

互斥锁主要用于实现 Linux 内核中的互斥访问功能，在 Aurora 的 SMD 的实现上，互斥锁主要用于 SMD 信道的打开或关闭过程。定义如下：

```
static DEFINE_MUTEX(smd_creation_mutex);
```

关于自旋锁和互斥锁的更多内容请参考文献[2]。

3.1.2 SMD 数据通信

在 Linux 中，基于 SMD 的数据通信是以信道的形式作为一个设备存在的，作为一种双向信道，其接口的实现遵循 Linux 设备驱动规范。在 Qucalcomm 平台上，SMD 的缓冲大小为 8192bit，最大信道数为 64，

SMD 的头大小为 20bit。

SMD 的相关代码实现主要位于 aurora\msm\msm\arch\arm\mach-msm 目录下。主要文件包括：smd.c、smd_nmea.c 、 smd_qmi.c 、 smd_rpcrouter.c 、 smd_rpcrouter_clients.c 、 smd_ rpcrouter_device.c 、smd_rpcrouter_servers.c、smd_tty.c 等。

SMD 信道需要同时维护接收信道、发送信道的状态和数据信息，SMD 的信道定义如下：

```
struct smd_channel {
    volatile struct smd_half_channel *send;     //发送握手信道
    volatile struct smd_half_channel *recv;     //接收握手信道
    unsigned char *send_buf;                    //发送信道数据
    unsigned char *recv_buf;                    //接收信道数据
    unsigned buf_size;
    struct list_head ch_list;                   //信道列表
    unsigned current_packet;
    unsigned n;
    void *priv;
    void (*notify)(void *priv, unsigned flags);
    int (*read)(smd_channel_t *ch, void *data, int len);          //读数据
    int (*write)(smd_channel_t *ch, const void *data, int len);   //写数据
    int (*read_avail)(smd_channel_t *ch);                         //是否可读
    int (*write_avail)(smd_channel_t *ch);                        //是否可写
    int (*read_from_cb)(smd_channel_t *ch, void *data, int len);
    void (*update_state)(smd_channel_t *ch);
    unsigned last_state;
    char name[20];
    struct platform_device pdev;
    unsigned type;
};
```

共享信道的信道状态在其握手信道中记录，握手信道的定义如下：

```
struct smd_half_channel {
    unsigned state;
    unsigned char fDSR;
    unsigned char fCTS;
    unsigned char fCD;
    unsigned char fRI;
    unsigned char fHEAD;      //头部
    unsigned char fTAIL;      //尾部
    unsigned char fSTATE;     //状态
    unsigned char fUNUSED;
    unsigned tail;
    unsigned head;
};
```

在实际实现中，SMD 信道分配被封装在 SMEM（Shared Memory Manager）模块中，系统提供了 smem_init()、smem_alloc()、smem_get_entry()等内存操作函数供 SMD 和 SMSM 操作。

SMD 的状态共有 SMD_SS_CLOSED、SMD_SS_OPENING、SMD_SS_OPENED、SMD_SS_FLUSHING、SMD_SS_CLOSING、SMD_SS_RESET、SMD_SS_RESET_OPENING 等。其变化过程如图 3-1 所示。

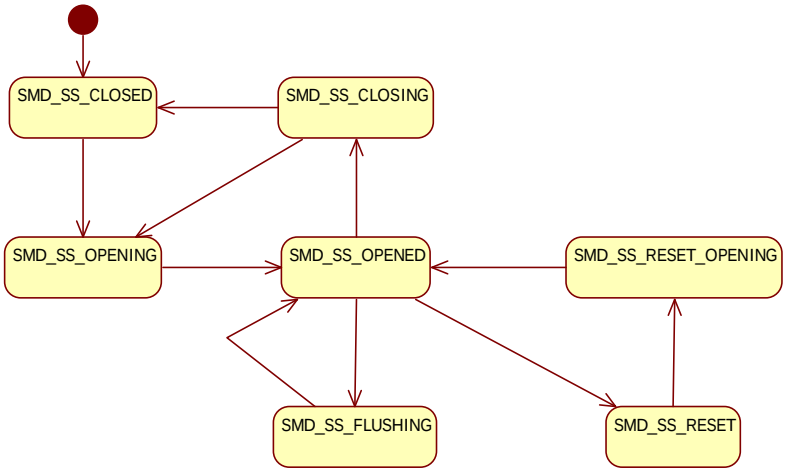

图 3-1　SMD 的状态机

下面结合 SMD 信道的实现简要介绍 SMD 信道的分配、打开、读取、写入、关闭等操作。

1．分配信道

SMD 信道根据数据的类型可以分为流信道和包信道，其中包信道具有比流信道更强的流控制能力，包含头信息。在创建 SMD 信道时，会根据信道类型的不同，确定创建的是 FIFO 信道还是普通信道，是流信道还是包信道，然后为 SMD 进行设备注册。SMD 分配信道的实现如下：

代码 3-1　SMD 分配信道的过程

```
static void smd_alloc_channel(struct smd_alloc_elm *alloc_elm)
{
    struct smd_channel *ch;
    uint32_t *smd_ver;
//分配 SMEM 内存
    smd_ver=smem_alloc(SMEM_VERSION_SMD, 32 * sizeof(uint32_t));
    if (smd_ver && ((smd_ver[VERSION_MODEM]>>16)>=1))
        ch=smd_alloc_channel_v2(alloc_elm->cid);        //FIFO 信道
    else
        ch=smd_alloc_channel_v1(alloc_elm->cid);        //普通信道
    if (ch==0)
        return;
    ch->type=SMD_CHANNEL_TYPE(alloc_elm->type);
    memcpy(ch->name, alloc_elm->name, 20);
    ch->name[19]=0;
    if (smd_is_packet(alloc_elm)) {                     //包信道
        ch->read=smd_packet_read;
        ch->write=smd_packet_write;
        ch->read_avail=smd_packet_read_avail;
        ch->write_avail=smd_packet_write_avail;
        ch->update_state=update_packet_state;
        ch->read_from_cb=smd_packet_read_from_cb;
    } else {                                            //流信道
        ch->read=smd_stream_read;
        ch->write=smd_stream_write;
        ch->read_avail=smd_stream_read_avail;
        ch->write_avail=smd_stream_write_avail;
        ch->update_state=update_stream_state;
        ch->read_from_cb=smd_stream_read;
    }
    ch->pdev.name=ch->name;
    ch->pdev.id=ch->type;
    pr_info("smd_alloc_channel() '%s' cid=%d\n",
        ch->name, ch->n);
```

```
    mutex_lock(&smd_creation_mutex);                    //互斥锁
//将信道添加到“smd_ch_closed_list”列表中
    list_add(&ch->ch_list, &smd_ch_closed_list); mutex_unlock(&smd_creation_mutex);
    platform_device_register(&ch->pdev);                //注册设备
}
```

2．打开信道

为了打开一个信道，首先要判断 SMD 信道是否已经初始化。如果 SMD 信道已经初始化，就根据信道名获得信道，将信道加入到“smd_ch_list”信道列表中并设置该信道的状态为 SMD_SS_OPENING，然后调用 notify_other_smd()函数通知其他的信道该信道已经激活。在默认情况下，其信道类型为 SMD_APPS_MODEM，打开一个 SMD 信道的实现如下：

代码 3-2　SMD 打开信道的过程

```
int smd_named_open_on_edge(const char *name, uint32_t edge,
                smd_channel_t **_ch,
                void *priv, void (*notify)(void *, unsigned))
{
    struct smd_channel *ch;
    unsigned long flags;
    if (smd_initialized==0) {               //判断 SMD 信道是否已初始化
        printk(KERN_INFO "smd_open() before smd_init()\n");
        return -ENODEV;
    }
    D("smd_open('%s', %p, %p)\n", name, priv, notify);
    ch=smd_get_channel(name, edge);             //获取信道
    if (!ch)
        return -ENODEV;
    if (notify==0)
        notify=do_nothing_notify;
    ch->notify=notify;
    ch->current_packet=0;
    ch->last_state=SMD_SS_CLOSED;
    ch->priv=priv;
    *_ch=ch;
    D("smd_open: opening '%s'\n", ch->name);
    spin_lock_irqsave(&smd_lock, flags);        //自旋锁
    list_add(&ch->ch_list, &smd_ch_list);       //将信道添加到“smd_ch_list”列表中
    D("%s: opening ch %d\n", __func__, ch->n);
    smd_state_change(ch, ch->last_state, SMD_SS_OPENING);      //信道状态变更
    spin_unlock_irqrestore(&smd_lock, flags);
    return 0;
}
```

3．关闭信道

关闭信道的操作相对简单，首先将信道从“smd_ch_list”信道列表中删除，然后将信道状态设置为 SMD_SS_CLOSED，并将信道添加到“smd_ch_closed_list”信道列表中即可。关闭 SMD 信道的实现如下：

代码 3-3　SMD 关闭信道的过程

```
int smd_close(smd_channel_t *ch)
{
    unsigned long flags;
    printk(KERN_INFO "smd_close(%p)\n", ch);
    if (ch==0)
        return -1;
    spin_lock_irqsave(&smd_lock, flags);            //自旋锁
    ch->notify=do_nothing_notify;
    list_del(&ch->ch_list);                         //从打开信道列表中去除该信道
```

```
    ch_set_state(ch, SMD_SS_CLOSED);                //设置信道状态
    spin_unlock_irqrestore(&smd_lock, flags);
    mutex_lock(&smd_creation_mutex);                //互斥锁
    list_add(&ch->ch_list, &smd_ch_closed_list);    //将信道添加至关闭信道列表
    mutex_unlock(&smd_creation_mutex);
    return 0;
}
```

4. 信道读取

包信道的内容读取涉及缓冲的复制、与其他 SMD 的消息通信和包状态的更新。从包信道读取数据的实现如下：

代码 3-4　SMD 包信道读取数据的过程

```
static int smd_packet_read(smd_channel_t *ch, void *data, int len)
{
    unsigned long flags;
    int r;
    if (len<0)
        return -EINVAL;
    if (len>ch->current_packet)
        len=ch->current_packet;
    r=ch_read(ch, data, len);               //读取数据
    if (r>0)
        notify_other_smd(ch->type);
    spin_lock_irqsave(&smd_lock, flags);        //自旋锁
    ch->current_packet-=r;
    update_packet_state(ch);                    //更新包状态
    spin_unlock_irqrestore(&smd_lock, flags);
    return r;
}
```

流信道的内容读取非常简单，只需要调用 ch_read()函数读取数据并通知其他 SMD 该信道处于打开状态即可。流信道读取的实现如下：

代码 3-5　SMD 流信道读取数据的过程

```
static int smd_stream_read(smd_channel_t *ch, void *data, int len)
{
    int r;
    if (len<0)
        return -EINVAL;
    r=ch_read(ch, data, len);               //读取数据
    if (r>0)
        notify_other_smd(ch->type);         //通知其他 SMD，该信道处于激活状态。
    return r;
}
```

流信道和包信道在读取信道数据时，都需要调用 ch_read()函数来实现真正的数据读取，ch_read()函数的实现如下：

代码 3-6　SMD 信道读取数据的过程

```
static int ch_read(struct smd_channel *ch, void *_data, int len)
{
    void *ptr;
    unsigned n;
    unsigned char *data=data;
    int orig_len=len;
    while (len>0) {
        n=ch_read_buffer(ch, &ptr);         //读取缓冲
        if (n==0)
            break;
```

```
        if (n>len)
            n=len;
        if (data)
            memcpy(data, ptr, n);          //数据复制
        data+=n;
        len-=n;
        ch_read_done(ch,n);                //读取完成
    }
    return orig_len-len;
}
```

5. 信道写入

在信道的数据写入方面，同样分为流信道数据的写入和包信道数据的写入两种类型。

向包信道中写入数据的过程和读取数据不同，在写入数据前，要首先利用 smd_stream_write_avail()函数判断是否有数据可供写入，在确定有可供写入的数据的情况下才调用 smd_stream_write()函数执行数据的写入操作。包信道写入数据的实现如下：

代码 3-7　SMD 包信道写入数据的过程

```
static int smd_packet_write(smd_channel_t *ch, const void *_data, int len)
{
    int ret;
    unsigned hdr[5];
    D("smd_packet_write() %d->ch%d\n",len, ch->n);
    if (len<0)
        return-EINVAL;
    if (smd_stream_write_avail(ch)<(len+SMD_HEADER_SIZE))    //判断数据写入进度
        return-ENOMEM;
    hdr[0]=len;
    hdr[1]=hdr[2]=hdr[3]=hdr[4]=0;
    ret=smd_stream_write(ch, hdr, sizeof(hdr));              //流写入
    if (ret<0 || ret!=sizeof(hdr)) {
        D("%s failed to write pkt header: "
          "%d returned\n", __func__, ret);
        return -1;
    }
    ret=smd_stream_write(ch, _data, len);
    if (ret<0 || ret != len) {
        D("%s failed to write pkt data: "
          "%d returned\n", __func__, ret);
        return ret;
    }
    return len;
}
```

流信道数据的写入和包信道数据的写入不同，其首先获得下一段可用缓冲的指针，然后执行内存复制，流信道写入数据的实现如下：

代码 3-8　SMD 流信道写入数据的过程

```
static int smd_stream_write(smd_channel_t *ch, const void *_data, int len)
{
    void *ptr;
    const unsigned char *buf=data;
    unsigned xfer;
    int orig_len=len;
    D("smd_stream_write() %d->ch%d\n", len, ch->n);
    if (len<0)
        return -EINVAL;
    while ((xfer=ch_write_buffer(ch, &ptr)) != 0) { //写入数据
        if (!ch_is_open(ch))
```

```
                break;
            if (xfer>len)
                xfer=len;
            memcpy(ptr, buf, xfer);                         //内存复制
            ch_write_done(ch, xfer);                        //完成写入数据
            len-=xfer;
            buf+=xfer;
            if (len==0)
                break;
        }
        if (orig_len-len)
            notify_other_smd(ch->type);             //通知其他 SMD，该信道处于激活状态
        return orig_len-len;
    }
```

SMD 是多核通信的基础，在 SMD 之上是一个叫做 RPC 路由器（RPC Router）的封装。关于 RPC 路由器，将在 3.2 节过程调用中详细介绍。

通过 SMD，可以为系统提供 RPC、DIAG、AT 命令、NMEA（GPS 数据）、数据服务、拨号等服务。

3.1.3 SMSM 状态通信

SMSM 为处理状态而非数据的共享内存，主要就电源管理、共享内存、定时器、进程状态等信息在多核之间进行通信。SMSM 的实现框架和 SMD 类似。

SMSM 的状态共有：SMSM_INIT、SMSM_OSENTERED、SMSM_SMDWAIT、SMSM_SMDINIT、SMSM_RPCWAIT、SMSM_RPCINIT、SMSM_RESET、SMSM_RSA、SMSM_RUN、SMSM_PWRC、SMSM_TIMEWAIT、SMSM_TIMENIT、SMSM_PWRC_EARLY_EXIT、SMSM_WFPI、SMSM_SLEEP、SMSM_SLEEPEXIT、SMSM_OEMSBL_RELEASE、SMSM_APPS_REBOOT、SMSM_SYSTEM_POWER_DOWN、SMSM_SYSTEM_REBOOT、SMSM_SYSTEM_DOWNLOAD、SMSM_PWRC_SUSPEND、SMSM_APPS_SHUTDOWN、SMSM_SMD_LOOPBACK、SMSM_RUN_QUIET、SMSM_MODEM_WAIT、SMSM_MODEM_BREAK、SMSM_MODEM_CONTINUE、SMSM_UNKNOWN 等。

在实际应用上，SMSM 主要在处理器发生状态变化，以及发生中断、重设基带处理器时运行，下面结合处理器状态变化和中断等两种情况简要介绍 SMSM 的处理过程。

处理处理器的变化，首先判断接入点的有效性，是应用处理器还是基带处理器等。如果接入点有效，就分配共享内存，并将相应的信息通知给其他的 SMSM，下面是 smsm_change_state()函数的实现：

代码 3-9　SMSM 改变状态的过程

```
int smsm_change_state(uint32_t smsm_entry,
                      uint32_t clear_mask, uint32_t set_mask)
{
    unsigned long flags;
    uint32_t  *smsm;
    uint32_t  old_state;
    if (smsm_entry >= SMSM_NUM_ENTRIES) {       //判断有效性
        printk(KERN_ERR "smsm_change_state: Invalid entry %d",
              smsm_entry);
        return -EINVAL;
    }
    spin_lock_irqsave(&smem_lock, flags);
    smsm=smem_alloc(ID_SHARED_STATE,            //分配共享内存
                    SMSM_NUM_ENTRIES * sizeof(uint32_t));
    if (smsm) {
        old_state=smsm[smsm_entry];
        smsm[smsm_entry]=(smsm[smsm_entry] & ~clear_mask) | set_mask;
        if (msm_smd_debug_mask & MSM_SMSM_DEBUG)
            printk(KERN_INFO "smsm_change_state %x\n",
```

```
                    smsm[smsm_entry]);
            notify_other_smsm(SMSM_APPS_STATE, old_state, smsm[smsm_entry]);
        }                       //通知其他 SMSM
        spin_unlock_irqrestore(&smem_lock, flags);
        if (smsm==NULL) {
            printk(KERN_ERR "smsm_change_state<SM NO STATE>\n");
            return -EIO;
        }
        return 0;
    }
```

当 INT_A9_M2A_5 和 INT_ADSP_A11 中断发生时，会触发 SMSM 的中断处理函数，SMSM 处理中断的过程如下：

代码 3-10　SMSM 处理中断的过程

```
    static irqreturn_t smsm_irq_handler(int irq, void *data)
    {
        unsigned long flags;
        uint32_t *smsm;
        static uint32_t prev_smem_q6_apps_smsm;
        if (irq==INT_ADSP_A11) {
            smsm=smem_alloc(SMEM_SMD_SMSM_INTR_MUX,     //分配内存
                            SMSM_NUM_INTR_MUX * sizeof(uint32_t));
            if (!smsm ||
               (smsm[SMEM_Q6_APPS_SMSM]==prev_smem_q6_apps_smsm))
                return IRQ_HANDLED;
            prev_smem_q6_apps_smsm=smsm[SMEM_Q6_APPS_SMSM];
        }
        spin_lock_irqsave(&smem_lock, flags);
        smsm=smem_alloc(ID_SHARED_STATE,                    //分配内存
                        SMSM_NUM_ENTRIES * sizeof(uint32_t));
        if (smsm==0) {
            printk(KERN_INFO "<SM NO STATE>\n");
        } else {
            unsigned old_apps, apps;
            unsigned modm=smsm[SMSM_MODEM_STATE];
            old_apps=apps=smsm[SMSM_APPS_STATE];
            if (msm_smd_debug_mask & MSM_SMSM_DEBUG)
              printk(KERN_INFO "<SM %08x %08x>\n", apps, modm);
            if (apps & SMSM_RESET) {
                apps &=~SMSM_RESET;
                smd_fake_irq_handler(0);            //让 SMD 响应假中断
                modem_queue_start_reset_notify(); //发起重设提醒
            } else if (modm & SMSM_RESET) {
                apps |=SMSM_RESET;
            } else {
                apps|=SMSM_INIT;
                if (modm & SMSM_SMDINIT)
                    apps|=SMSM_SMDINIT;
                if (modm & SMSM_RPCINIT)
                    apps|=SMSM_RPCINIT;
                if ((apps & (SMSM_INIT|SMSM_SMDINIT|SMSM_RPCINIT))==
                    (SMSM_INIT|SMSM_SMDINIT|SMSM_RPCINIT))
                    apps|=SMSM_RUN;
            }
            if (smsm[SMSM_APPS_STATE]!=apps) {
                if (msm_smd_debug_mask & MSM_SMSM_DEBUG)
                    printk(KERN_INFO "<SM %08x NOTIFY>\n", apps);
                smsm[SMSM_APPS_STATE]=apps;
                do_smd_probe();
                notify_other_smsm(SMSM_APPS_STATE, old_apps, apps); //通知其他 SMSM
            }
```

```
        }
        spin_unlock_irqrestore(&smem_lock, flags);
        return IRQ_HANDLED;
    }
```

3.2 过程调用

在 Android 中，远程过程调用是基于 ONC RPC 来实现的，在 Android 上层的接口为 IBinder,在框架层面，其实现主要包括 RPC 路由器（RPC Router）、RPC 服务器（RPC Server）和 RPC 管道等 3 部分。

关于多核之间远程过程调用的内容，在 Qucalcomm 平台上，相关的代码实现主要位于 aurora\msm\msm\arch\arm\mach-msm 和 aurora\msm\msm\net\sunrpc 目录下。关于 ONC RPC 协议的实现位于 aurora\msm7k\msm7k\librpc 目录下。关于 RDMA 的内容位于 aurora\msm\msm\net\sunrpc\xprtrdma 目录下。协议的具体实现不是本书关注的重点，这里就不再进行过多的描述，有兴趣的读者可以参考文献[34]。

远程过程调用物理上是基于 3.1.2 节数据通信中介绍的共享内存（SMD，Shared Memory Driver）的。

事实上，Aurora 的鼠标、键盘、全速 USB、高速 USB 等设备都是基于 ONC RPC 来实现通信的。相关的实现分布在 rpcmouse.c 、rpckbd.c 、rpc_fsusb.c、rpc_hsusb.c 等文件中。

3.2.1 RPC 路由器

RPC 路由器对服务器端和客户端的通信提供了支持。在整个 ONC RPC 架构上，ONC RPC 主要由 3 部分构成：客户端、框架层、服务器端。其中框架层由 RPC 路由器和 ONC RPC 协议栈构成，ONC RPC 协议栈是基于 ONC RPC 协议的实现，执行接口描述语言的解析、自动产生 RPC 通信所需的框架代码等工作。ONC RPC 架构如图 3-2 所示。

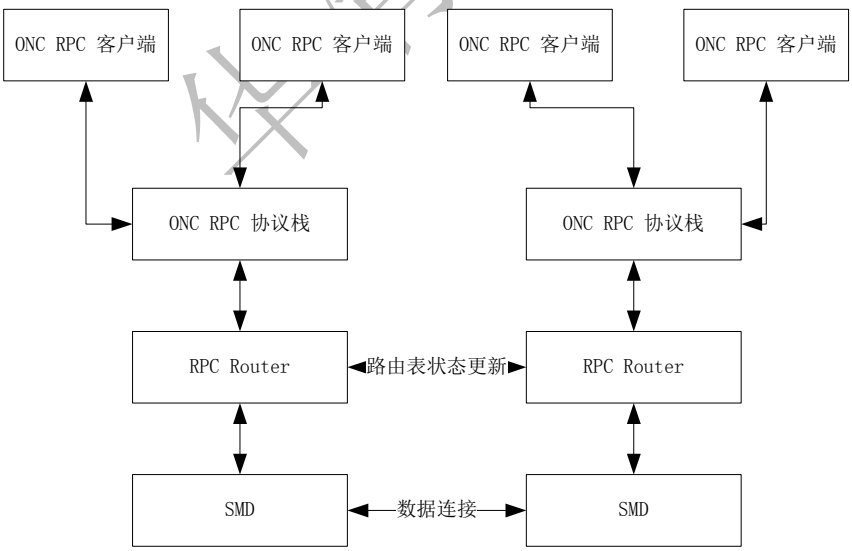

图 3-2 ONC RPC 架构

关于 RPC 路由器的实现主要分布在 smd_rpcrouter.c、smd_rpcrouter_device.c、smd_rpcrouter_servers.c、smd_rpcrouter_clients.c 等文件中。RPC 路由器起着 RPC 服务器查询、RPC 服务器和 RPC 客户端的注册和销毁，以及底层通信的封装功能，类似于 TCP 协议。

在实际的实现中，RPC 路由器和 RPC 服务器均是作为一个虚拟的字符型设备来存在的。

下面是 RPC 路由器的创建过程：

代码 3-11 RPC 路由器的创建过程

```
int msm_rpcrouter_init_devices(void)
{
    int rc;
    int major;
    msm_rpcrouter_class=class_create(THIS_MODULE, "oncrpc"); //创建设备节点
    if (IS_ERR(msm_rpcrouter_class)) {
        rc=-ENOMEM;
        printk(KERN_ERR
             "rpcrouter: failed to create oncrpc class\n");
        goto fail;
    }
    rc=alloc_chrdev_region(&msm_rpcrouter_devno, 0, //作为字符型设备分配资源
                              RPCROUTER_MAX_REMOTE_SERVERS + 1, "oncrpc");
    if (rc<0) {
        printk(KERN_ERR
              "rpcrouter: Failed to alloc chardev region (%d)\n", rc);
        goto fail_destroy_class;
    }
    major=MAJOR(msm_rpcrouter_devno);
    rpcrouter_device=device_create(msm_rpcrouter_class, NULL,    //创建设备
                      msm_rpcrouter_devno, NULL, "%.8x:%d",
                      0, 0);
    if (IS_ERR(rpcrouter_device)) {
        rc=-ENOMEM;
        goto fail_unregister_cdev_region;
    }
    cdev_init(&rpcrouter_cdev, &rpcrouter_router_fops);  //字符型设备初始化
    rpcrouter_cdev.owner=THIS_MODULE;
    rc=cdev_add(&rpcrouter_cdev, msm_rpcrouter_devno, 1);
    if (rc<0)
        goto fail_destroy_device;
    return 0;
fail_destroy_device:
    device_destroy(msm_rpcrouter_class, msm_rpcrouter_devno);//销毁设备
fail_unregister_cdev_region:
    unregister_chrdev_region(msm_rpcrouter_devno,            //去注册
                  RPCROUTER_MAX_REMOTE_SERVERS + 1);
fail_destroy_class:
    class_destroy(msm_rpcrouter_class);
fail:
    return rc;
}
```

如图 3-3 所示为 RPC 客户端向 RPC 路由器进行注册和销毁的过程。

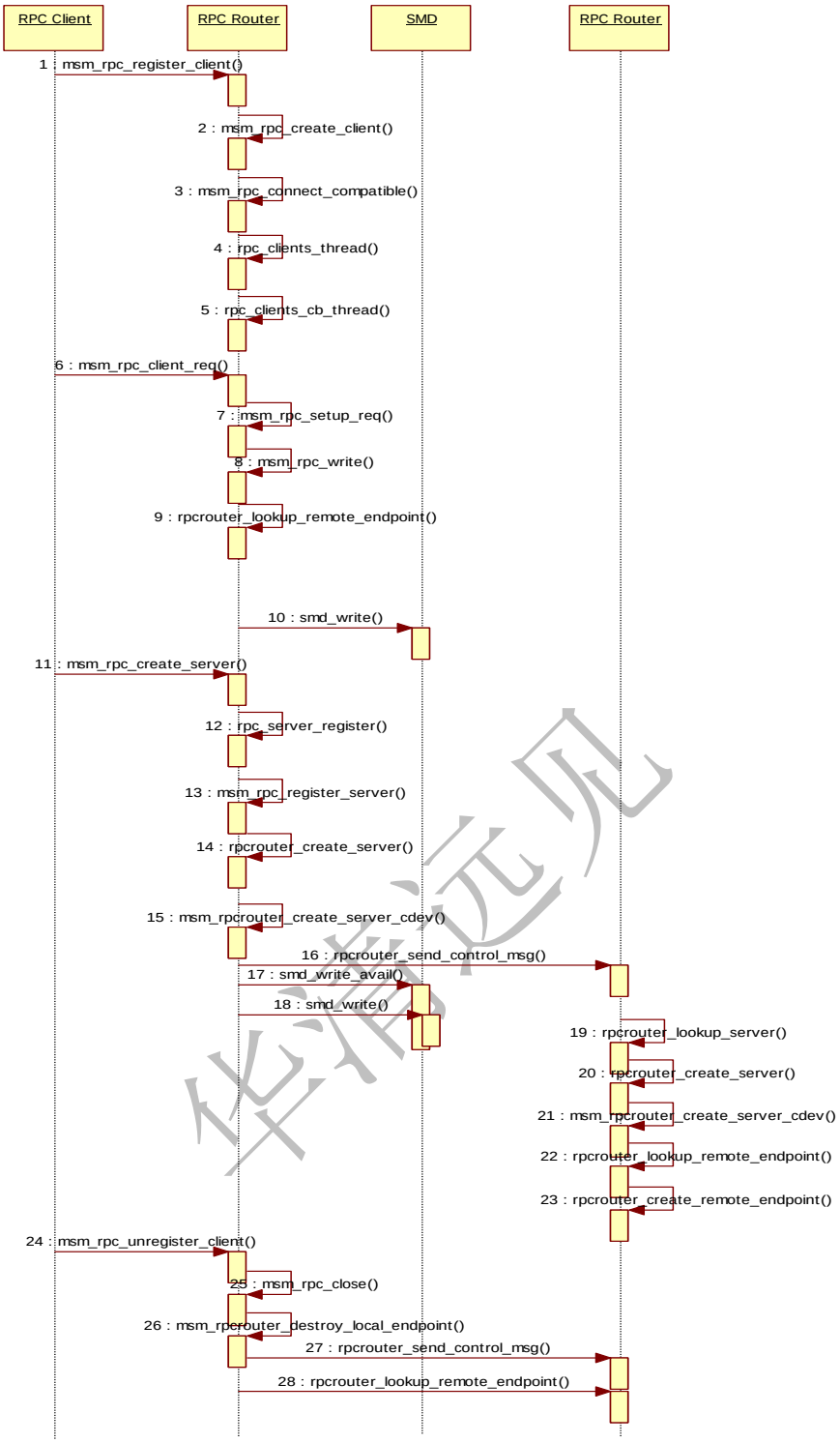

图 3-3　RPC 客户端注册和销毁过程

以 msm-handset 驱动为例，这是一个处理按键和 LCD 事件的驱动，在驱动创建时，会调用 hs_rpc_init()函数进行 RPC 客户端和 RPC 服务器的创建工作。

为了创建“hs”RPC 客户端，首先要完成 RPC 客户端的注册。在 RPC 客户端向 RPC 路由器注册的过程中，会首先调用 msm_rpc_create_client()函数进行 RPC 客户端的创建，随后尝试查询 RPC 服务器进行链接。在完成这些工作后，会创建一个读取数据的线程和一个回调线程。

在读取数据的线程中，会循环执行 msm_rpc_read()函数进行操作，监听来自 RPC 服务器的响应。

为了和 RPC 服务器进行通信，首先要调用 msm_rpc_setup_req()函数创建一个 rpc_request_hdr，然后通过 msm_rpc_write()函数发送消息。在发送消息前，要首先调用 rpcrouter_lookup_remote_endpoint()函数完成

远端服务器的查找工作，找到 RPC 服务器后，调用 msm_rpc_write_pkt()函数，通过 smd_write()函数向共享内存写入数据。

为了创建“hs”RPC 服务器，首先要调用 msm_rpc_create_server()函数通过 rpc_server_register()函数完成 RPC 服务器的创建和注册工作。

创建服务器的过程为，首先通过 rpcrouter_create_server()函数完成 PRC 服务器的创建，然后发送“RPCROUTER_CTRL_CMD_NEW_SERVER”命令通知 RPC 路由器。相关的路由器命令在 process_control_msg()函数中进行处理。

当销毁 RPC 客户端时，需要调用 msm_rpc_unregister_client()函数完成本地端点的销毁和回调函数的注销。同时向 RPC 路由器发送“RPCROUTER_CTRL_CMD_REMOVE_ CLIENT”命令。

3.2.2 RPC 管道

在 Linux 中，由于没有对应的物理设备，RPC 管道文件系统是作为一种虚拟的文件系统存在的，其在系统中的注册和注销等操作都必须遵循 Linux 文件系统规范。关于 RPC 管道的实现主要分布在 rpc_pipe.c 文件中。下面是 RPC 管道文件系统在 sunrpc_syms.c 文件中的初始化过程：

代码 3-12 RPC 管道文件系统的初始化过程

```
static int __init  init_sunrpc(void)          //__init 表示该函数只在初始化过程中调用
{
    int err=register_rpc_pipefs();            //注册 RPC 管道文件系统“rpc_pipefs”
    if (err)
        goto out;
    err=rpc_init_mempool();                   //初始化内存池
    if (err) {
        unregister_rpc_pipefs();              //注销 RPC 管道文件系统“rpc_pipefs”
        goto out;
    }
#ifdef RPC_DEBUG
    rpc_register_sysctl();                    //注册 SYSCTL，方便用户读取或调整系统设置
#endif
#ifdef CONFIG_PROC_FS
    rpc_proc_init();                          //初始化 PROC，方便用户获取运行时信息
#endif
    cache_register(&ip_map_cache);                          //注册缓冲
    cache_register(&unix_gid_cache);
    svc_init_xprt_sock();
    init_socket_xprt();                                     //初始化套接字
    rpcauth_init_module();                                  //初始化 RPC 鉴权模块
out:
    return err;
}
```

和其他文件系统一样，一个 RPC 管道是作为 RPC 管道文件系统下的一个文件存在的，下面是 RPC 管道的接口定义：

```
static const struct file_operations rpc_pipe_fops={
    .owner=THIS_MODULE,
    .llseek=no_llseek,
    .read=rpc_pipe_read,            //读取管道数据
    .write=rpc_pipe_write,          //写入管道数据
    .poll=rpc_pipe_poll,            //轮询管道
    .ioctl=rpc_pipe_ioctl,          //管道的 ioctl
    .open=rpc pipe_open,            //打开管道
    .release=rpc_pipe_release,      //释放管道
};
```

一个 RPC 管道就是 RPC 管道文件系统的一个节点，下面是 RPC 节点的定义：

```
struct rpc_inode {
    struct inode vfs_inode;                   //继承 VFS 节点
```

```
        void *private;
        struct list_head pipe;
        struct list_head in_upcall;             //通知上层节点
        struct list_head in_downcall;           //通知下层节点
        int pipelen;
        int nreaders;                           //读数据线程数
        int nwriters;                           //写数据线程数
        int nkern_readwriters;
        wait_queue_head_t waitq;                //等待队列
    #define RPC_PIPE_WAIT_FOR_OPEN 1
        int flags;
        struct rpc_pipe_ops *ops;               //管道选项
        struct delayed_work queue_timeout;
    };
```

作为文件节点，当然拥有自己的选项，下面是 RPC 管道选项的定义：

```
    struct rpc_pipe_ops {
        ssize_t (*upcall)(struct file *, struct rpc_pipe_msg *, char __user *, size_t);
        ssize_t (*downcall)(struct file *, const char __user *, size_t);
        void (*release_pipe)(struct inode *);           //释放管道
        int (*open_pipe)(struct inode *);               //打开管道
        void (*destroy_msg)(struct rpc_pipe_msg *);     //销毁管道消息
    };
```

为了利用 RPC 在 Linux 内核和用户空间之间进行通信，需要创建一个 RPC 管道，下面是创建一个 RPC 管道的实现过程：

代码 3-13　创建 RPC 管道的过程

```
    struct dentry * rpc_mkpipe(struct dentry *parent, const char *name, void *private, struct rpc_pipe_ops
*ops, int flags)
    {
        struct dentry *dentry;
        struct inode *dir, *inode;
        struct rpc_inode *rpci;
        dentry=rpc_lookup_create(parent, name, strlen(name), 0); //创建目录
        if (IS_ERR(dentry))
            return dentry;
        dir=parent->d_inode;
        if (dentry->d_inode) {
            rpci=RPC_I(dentry->d_inode);
            if (rpci->private !=private ||
                    rpci->ops !=ops ||
                    rpci->flags !=flags) {
                dput (dentry);
                dentry=ERR_PTR(-EBUSY);
            }
            rpci->nkern_readwriters++;
            goto out;
        }
        inode=rpc_get_inode(dir->i_sb, S_IFIFO | S_IRUSR | S_IWUSR); //配置权限
        if (!inode)
            goto err_dput;
        inode->i_ino=iunique(dir->i_sb, 100);
        inode->i_fop=&rpc_pipe_fops;
        d_instantiate(dentry, inode);
        rpci=RPC_I(inode);
        rpci->private=private;
        rpci->flags=flags;
        rpci->ops=ops;
        rpci->nkern_readwriters=1;
        fsnotify_create(dir, dentry);
        dget(dentry);
    out:
```

```
        mutex_unlock(&dir->i_mutex);
        return dentry;
    err_dput:
        dput(dentry);
        dentry=ERR_PTR(-ENOMEM);
        printk(KERN_WARNING "%s: %s() failed to create pipe %s/%s (errno=%d)\n",
    __FILE__, __func__, parent->d_name.name, name,-ENOMEM);
        goto out;
    }
```

通过 RPC 管道文件系统，调用者可以像操作其他文件那样进行 RPC 管道的操作，常见的操作有读取数据、写入数据、查询管道信息、创建路径、删除路径、创建 RPC 管道等。

需要说明的是，当管道数据可读时，需要调用 rpc_queue_upcall()函数通知用户空间，当管道可写入数据时，则不需要通知用户空间。在通知用户空间 RPC 管道的状态时，需要将消息封装在 rpc_pipe_msg 结构体中。下面是 rpc_queue_upcall()函数的实现过程：

代码 3-14　RPC 通知用户空间的过程

```
    int rpc_queue_upcall(struct inode *inode, struct rpc_pipe_msg *msg)
    {
        struct rpc_inode *rpci=RPC_I(inode);
        int res=-EPIPE;
        spin_lock(&inode->i_lock);     //自旋锁
        if (rpci->ops==NULL)
            goto out;
        if (rpci->nreaders) {
            list_add_tail(&msg->list, &rpci->pipe); //添加到消息队列
            rpci->pipelen+=msg->len;
            res=0;
        } else if (rpci->flags & RPC_PIPE_WAIT_FOR_OPEN) {   //如果 RPC 管道等待打开
            if (list_empty(&rpci->pipe))
                queue_delayed_work(rpciod_workqueue,        //添加到 rpciod_workqueue
                        &rpci->queue_timeout,
                        RPC_UPCALL_TIMEOUT);
            list_add_tail(&msg->list, &rpci->pipe);         //添加到消息队列
            rpci->pipelen += msg->len;
            res=0;
        }
    out:
        spin_unlock(&inode->i_lock);
        wake_up(&rpci->waitq);                              //唤醒等待队列
        return res;
    }
```

在 auth_gss.c 文件中，有 RPC 管道的运用实例，感兴趣的读者可以自行研读。

联系方式

深圳地址：深圳市龙华新区人民北路美丽 AAA 大厦 15 层，电话：0755-22193762

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

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

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

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