

博客

登录 | 注册

Q

u010246947的专栏

目录视图

摘要视图

RSS 订阅

个人资料

u010246947

+ 加关注

发私信

访问：32296次

积分：928

等级：BLOG > 3

排名：千里之外

原创：63篇

转载：12篇

译文：0篇

评论：1条

文章搜索

Q

文章分类

IPC (1)

linux设备模型 (4)

linux内核 (5)

水滴石穿 (26)

跳吧

博客Markdown编辑器上线啦

那些年我们追过的Wrox精品红皮计算机图书

PMBOK第五版精讲视频教程

火星敏捷开发1001问

原

基于systemV的消息队列的多进程间CS通讯实现

分类：IPC

2013-05-13 14:30

360人阅读

评论(0)

收藏

举报

进程间通讯

IPC

消息队列

systemV

CS架构

目录(?)

[+]

因工作需要，需要整改一个C/S架构的进程间通讯，以systemV接口的方式，因为跑在marvell一个芯片上，这个东西起名字叫MIPC，废话少说，如下：

SystemV消息队列的报文结构要求如下：

```
typedef struct sysV_msg
{
    long type;-----消息类型

    XXXX msg_payload;-----消息内容
}sysV_msg_t;
```

现有约五六个用户进程，每个进程背后是一个自己的业务功能，每个进程都需要随时能访问对方的资源，这样就是说每个进程都向其他进程提供服务，也需要其他进程为自己提供服务，所以也就是C/S机制，而且每个进程对其他进程的访问都可能是同步的，也就是说需要被访问者给予访问者以回复，当然也可以是异步的；除此以外，每个进程可能会有不同的线程，它们也许都会向某一进程发起访问。

基于以上的要求，在采用systemV方式的前提下，需要一种机制能够满足以上的要求，根据systemV接口特点，计划如下：

- 每段路 (3)
- 网络 (1)
- linux网络协议栈 (29)
- 网络编程 (1)
- 数据库---MySQL (1)
- lua (1)
- boost (3)
- python (1)

文章存档

- 2015年01月 (1)
- 2014年12月 (3)
- 2014年06月 (1)
- 2014年04月 (2)
- 2014年02月 (10)

展开

阅读排行

- arm-linux内存页表创建 (1367)
- linux进程地址空间(3) 内 (785)
- linux进程地址空间(2) 缺 (722)
- ARM架构内核启动分析-t (675)
- linux网络协议栈(四)链路 (649)
- linux网络协议栈(四)链路 (645)
- linux arm的高端内存映射 (639)
- linux进程地址空间(1) for (596)
- TCP/IP网络层(转) (570)
- arm-linux之uboot向内核 (567)

评论排行

- linux设备模型____相关 (1)
- 自己编写Python连接MyS (0)
- linux网络协议栈(五)网络 (0)
- linux网络协议栈(五)网络 (0)
- linux网络协议栈(五)网络 (0)
- linux网络协议栈(五)网络 (0)

systemV方式的消息队列以消息队列ID、消息类型为区分，根据这个特点，可以让需要访问其他进程资源的每一个线程(注意是线程)注册一个消息队列ID，作为其自身的“通讯地址”，每次访问其他进程后，被访问者都要以该ID的消息队列回复给这个访问者线程，同时根据访问资源的不同，确定不同的消息类型，这样就可以将收发报文清晰的区分，具体如下：

上面说了，每一个线程都要注册一个消息队列ID，为了能够统一的管理，让每一个线程都知道给提供服务的是谁，怎么发送消息给他，需要一个机制，我们把形成这个机制所需的函数放在一个叫mipc.c的文件里，并且把该文件编成一个动态库给所有进程连接，这里的第一个机制就是注册，所谓注册就是把本线程的信息注册在一个条目表项中，这个表项的结构是：

```
typedef struct
{
    int flag_is_hold;-----标志位，表示是否是有效条目

    pthread_t thread_id;-----所在线程ID，重要区分标识

    char name[MIPC_NAME_MAX_LEN];-----条目名称，重要区分标识

    int mqd;-----消息队列ID

} MIPC_THREAD_INFO_T;
```

这样，当线程A需要访问另一进程的线程B的资源时，只需知道线程B的条目名称即可，就可以查表知道它的消息队列ID，这样就能够知道该把消息发到哪个消息队列，这里每个线程的条目名称是预知的。

也许你会问，动态库对于不同进程是同一份拷贝，那为什么线程A还可以查表获取另一个进程的线程B的条目内容，这是需要通过一个收发包进程实现，线程A需要发消息给另一个进程的线程B时，首先需要发消息队列给收发包进程，而发送报文的内容包括线程B的条目名称，由收发包进程解析报文内容获取线程A发送报文的目的地是那里，然后由收发包进程再把消息发送给线程B；线程B接收报文并做处理后，需要回复消息给线程A，也是发送给收发包进程，并由收发包进程解析报文内容获取到报文目的地是线程A，再把报文发送给线程A，线程A就可以收到另一个进程的线程B的回复了。这样一轮收发消息队列，就实现了线程A和另一个进程的线程B的同步访问。

下面是目前规定的报文格式：

```
typedef struct sysV_msg
{
```

- linux网络协议栈(六)传输层 (0)
- linux网络协议栈(六)传输层 (0)
- linux网络协议栈(六)传输层 (0)
- linux网络协议栈(五)网络层 (0)

推荐文章

- * [【ShaderToy】开篇](#)
- * [FFmpeg源代码简单分析：avio_open2\(\)](#)
- * [技能树之旅: 从模块分离到测试](#)
- * [Qt5官方demo解析集36——Wiggly Example](#)
- * [Unity3d HDR和Bloom效果（高动态范围图像和泛光）](#)

最新评论

linux设备模型——相关函数

crk_13: 这么好的文章，居然没人评论，是博主限制了么？小弟今年刚毕业，正在学习Linux设备模...

linux内核的TCP/IP协议栈

本专栏细致入微的描述了从网卡驱动、链路层、网络层、传输层、linux网络安全、socket网络编程、应用层的整个网络部分的原理和linux源码实现，没有一个字是ctrl V的，完全个人一字一字写出来，下面是目录： 1、网卡驱动，包括网卡驱动职责和原理和重要注意地方，一个最简单的网卡驱动的实现(工程实际使用，几十万台设备实际出货)，和广义网卡驱动 2、链路层，包括链路层原理、网桥原理与实现分析、邻居子系统与ARP的原理与实现分析、原始套接字分析、vlan分析、eoip L2隧道的源码实现分析 3、网络层，包括网络层原理、路由实现与使用分析、L3的qos、L3的分帧、IPV6、L3的各类VPN、NAT穿越 4、网络安全，包括linux的netfilter框架、实际使用分析 5、传输层，包括传输层原理、TCP/UDP/RAW IP的

long type;

sysV_msg_payload_t msg_payload;

}sysV_msg_t;

sysV_msg_t是顶层的消息队列报文格式：前四个字节type标识了发送线程发送的消息队列类型，同时也可以接收线程要接收的消息队列类型，这是MIPC能达到不同进程多个MIPC条目做到同步通讯的基础；msg_payload是报文内容，见接下来的描述：

typedef struct sysV_msg_payload

{

char src_name[MIPC_NAME_MAX_LEN];

char dst_name[MIPC_NAME_MAX_LEN];

MIPC_MSG_HDR_T msg_hdr;

unsigned char payload[MIPC_MSG_MAX_LEN];

}sysV_msg_payload_t;

sysV_msg_payload_t是报文具体内容的结构格式：

src_name填入发送线程注册时的条目名称；dst_name是报文发送目的地接收线程注册时的名称；msg_hdr是报文负载的头，记录该报文类型、长度、是否需要回复；payload是报文实际负载，目前规定最大长度为2048字节。

typedef struct

{

unsigned int msg_type; //-1:notice(add a new item); other:api_id

unsigned int msg_size;

int sync_flag;//0:no sync, >0:sync, timeout value; -1:sync, wait forever

} MIPC_MSG_HDR_T;

msg_hdr是报文负载的头，记录该报文类型、长度、是否需要回复

技术问答

快速回复

分析 6、socket，包括socket层面的内核实现、网络编程分析 7、应用层，简单的描述了典型的应用层协议，包括ftp、tftp、dhcp、dns等

每个进程的每个需要访问其他进程的线程，都需要注册一个MIPC条目，否则发起的进程间通讯将被丢弃掉，注册函数为mipc_init，注册过程是给收发中转进程发送一个特殊的消息，其代码片段如下：

```
while(root_msg_mqd < 0)

{

    root_msg_mqd = msgget(MIPC_MSG_ROOT_KEY, 0);

}
```

root_msg_mqd是收发中转进程的消息队列ID，收发中转进程必须首先启动建立这个消息队列，否则其他进程无法通过MIPC通讯；当该消息队列建立好后，其他进程将给收发中转进程发送特殊消息要求注册一个MIPC条目；

```
memset(&mipc_info, 0, sizeof(MIPC_THREAD_INFO_T));

mipc_info.flag_is_hold = MIPC_TRUE;

mipc_info.thread_id = pthread_self();

memcpy(mipc_info.name ,name, strlen(name));

mipc_info.mqd = mipc_mq_create();
```

上面这些就是要发送给收发中转进程的特殊的注册消息，其实质就是把要注册的内容按前面描述过的MIPC_THREAD_INFO_T结构类型写入报文的实际负载中，包括置位有效标志、写入自己的线程ID、写入自己的条目名称、写入自己的接收消息队列ID，函数mipc_mq_create就是调用msgget系统调用从内核申请一个消息队列ID；

```
msgq->type = MIPC_MSG_TYPE_NOTICE;

memcpy(msgq->msg_payload.src_name, name, strlen(name));

memcpy(msgq->msg_payload.dst_name, "mipc_main", strlen("mipc_main"));

msgq->msg_payload.msg_hdr.msg_size = sizeof(MIPC_THREAD_INFO_T);

msgq->msg_payload.msg_hdr.msg_type = MIPC_MSG_TYPE_NOTICE;

msgq->msg_payload.msg_hdr.sync_flag = 0;
```

技术问答

快速回复

```
memcpy(msgq->msg_payload.payload, &mipc_info, sizeof(MIPC_THREAD_INFO_T));
```

上面这段是填写报文的头的过程，在消息类型域中写入MIPC_MSG_TYPE_NOTICE，这是个特殊的类型标识是要注册一个条目；src_name域填写要注册条目的名称，这也是mipc_init函数的唯一参数；dst_name域填写mipc_main，这是预设的中转收发进程的名字；然后报文长度为结构MIPC_THREAD_INFO_T的大小；报文类型同样填写MIPC_MSG_TYPE_NOTICE；同步标识置为0意味中转收发进程收到后无需回复，因为没有太大回复的必要；最后把注册消息的内容复制到实际负载域中。

```
size += msgq->msg_payload.msg_hdr.msg_size;
```

```
ret = mipc_mq_send(root_msg_mqd, buffer, size, 0);
```

最后把整个报文长度重新更新好，并发送，函数mipc_mq_send实际调用系统调用msgsnd发送消息队列；中转收发进程在收到消息后就把该条目内容写入表中用于以后查询。

仅仅上面这些是不够的，因为一个进程可能会有多个线程都要发起对其他进程的访问，也就是每个线程都需要注册一个条目，如果每个线程发起访问时还需要接收回复，那么它必须知道自己所在线程条目的消息队列ID，每个线程很难做到随时可以找出属于自己的消息队列ID，所以还需要再次注册在属于本进程的一个条目表，如下：

```
user_thread_mipc_info_t user_thread_info[MIPC_USER_INFO_NUM];
```

```
ret = mipc_user_thread_info_register(name, mipc_info.mqd, pthread_self());
```

这个方法利用的是，库代码的数据段，对于，每个进程都有一份拷贝，所以每个进程都利用该函数配置libmipc.so中的全局变量数值user_thread_info[MIPC_USER_INFO_NUM]是不会冲突的，即是各自进程独有的；mipc_user_thread_info_register函数实现很简单，仍然是操作线程ID、消息队列ID、条目名称、有效标志的一个表，代码片段如下：

```
for (index = 0; index < MIPC_USER_INFO_NUM; index++)  
  
{  
  
    if (!user_thread_info[index].flag)  
  
    {  
  
        find_flag = 1;  
  
        break;  
  
    }
```

技术问答

快速回复

```
}
```

在这个全局变量数组中找到一个空的条目，该数值预设条目个数为10，应该肯定能满足每个进程的需要；

```
if (find_flag)
```

```
{
```

```
    if (src_name)
```

```
        memcpy(user_thread_info[index].myname, src_name, strlen(src_name));
```

```
    user_thread_info[index].mqd = mqd;
```

```
    user_thread_info[index].thread_id = tid;
```

```
    user_thread_info[index].flag = 1;
```

```
    return MIPC_OK;
```

```
}
```

```
return MIPC_ERROR;
```

如果真的找不到空余条目，则只能返回错误即不能增加条目；正常情况下是可以找到的，这时就把线程ID、消息队列ID、条目名称、有效标志写入该数组条目即可。

上面就完成了整个注册过程，所注册的线程在需要访问其他进程时，只需要首先查询自己的消息队列ID，然后以被访问线程的条目名称为参数就可以实现进程间通讯

每个进程都有自己的一个唯一的server，它是该进程唯一的server，由一个线程用MIPC条目接收处理对它的访问，也就是说，其他所有线程多该进程的访问都必须访问该server所在的MIPC条目，访问该进程其他MIPC条目无法得到正确的回复；每个进程的server所在MIPC条目代码片段如下：

```
osTaskCreate(&apm_mq_thread,
```

```
    "apm_mq_thread",
```

```
    (GLFUNCPTR) apm_mq_rcv,
```

技术问答

快速回复


```
0,

0,

47 - 1,0x2800) != IAMBA_OK)
```

每个进程都必须先建立一个线程，然后在这个线程内再建立MIPC条目用于接收处理所有对该进程的访问，上面就是先建立一个线程apm_mq_recv，其函数内容片段如下：

```
int mipc_fd = mipc_init("new_apm");
```

首先建立一个MIPC条目，这就是之后用于接收处理对该进程全部访问的MIPC条目；

```
mthread_register_mq(mipc_fd, apm_new_mipc_server_handler, &thread_index_apm, &mq_index_apm);
```

然后调用函数mthread_register_mq注册收到报文后的hook处理函数，这里处理函数是apm_new_mipc_server_handler；

```
while(1)

{

    size = mipc_mq_receive(mipc_fd, buffer, MIPC_DEFAULT_MAX_MSG_SIZE, 0,
MIPC_MSG_SYNC_WAIT_FOREVER);

    if (size > 0)

    {

        mthread_check_mq_msg(thread_index_apm, mq_index_apm, (char *)msgq, size);

        memset(buffer, 0, sizeof(buffer));

    }

}
```

最后进入永久循环，将阻塞地接收所有发给该MIPC条目的消息队列ID的全部报文，注意函数mipc_mq_receive的第四个参数为0，意为接收任何发送给该消息队列ID的第一个报文，也就是不区分消息类型；每收到报文都调用上一步注册的hook处理函数去处理报文。实际的处理函数，就是每个进程自己的server api函数，这些函数最后都会根据报文中的src_name、msg_type确定应该以什么样的消息类型回复给哪一个消息队列ID，这样做到了报文不会混淆，不会导致需要接收回复的收不到，不需要接收回复的反而收到的现象。

技术问答

快速回复

最后描述一下收发中转进程，之所以需要改进程，是因为，每个发起进程间通讯的MIPC条目所在线程，无法获悉目的进程的消息队列ID，因为每个进程依靠调用动态库libmipc.so建立条目，而库代码的数据段对于每个进程是独有的，所以它只能知道自己进程内部线程所建立的条目，无法知道其他进程的条目，所以将无法实现跨进程间的通讯；这就需要有一个能够维护所有条目表的东西，这就是收发中转进程，由它来记录维护全部注册的条目，而发起进程间通讯的任何线程只需知道被访问者的MIPC条目名称即可，下面是收发中转进程的代码片段：

osTaskCreate(&mipc_process_thread,

```
"mipc_process_thread",  
  
(GLFUNCPTR) mipc_process,  
  
0,  
  
0,  
  
47 - 1,  
  
0x2800) != IAMBA_OK)
```

首先建立一个线程只用于处理报文，主线程只负责接收，如下；

```
while (1)  
  
{  
  
    size = msgrcv(root_msg_mqd, (void *)buffer, MIPC_DEFAULT_MAX_MSG_SIZE, 0, 0);  
  
    if (size > 0)  
  
    {  
  
        ret = mipc_msg_check(buffer, size);  
  
        if (ret)  
  
        {  
  
            fprintf(stderr, "wrong mpic msg\n\r");  
  
            continue;  
  
        }  
  
    }
```

技术问答

快速回复


```
        //fprintf(stderr, "mipc_main: recv an msg, size:%d!\n\r", size);

        mipc_msg_list_add(buffer, size);

    }

}
```

主线程以阻塞方式接收所有发给它的报文，对报文进行简单判断后就加入到报文接收链表中，这样是为了提高效率，避免报文被积压在内核队列中，函数mipc_msg_list_add代码片段如下：

```
list_mipc_msg_t *newmsg = (list_mipc_msg_t *)malloc(sizeof(list_mipc_msg_t));

INIT_LIST_HEAD(&newmsg->list);

memcpy(newmsg->buffer, buffer, size);

newmsg->size = size;

pthread_mutex_lock(&mipc_msg_sem);

LIST_ADD_TAIL(&newmsg->list, &mipc_msg_list.list);

mipc_msg_list.addnum++;

mipc_msg_list.curnum++;

pthread_mutex_unlock(&mipc_msg_sem);
```

为每一个报文都动态申请一个报文结构，然后把它加入到链表尾部，并更新统计值；

报文处理线程的代码片段如下；

```
while(1)

{

    if (LIST_EMPTY(&mipc_msg_list.list))

        usleep(50);

    else

    {
```

技术问答

快速回复

```
mipc_msg_list_del(buffer, &size);

ret = mipc_msg_process(buffer, size);

if (ret)

{

    fprintf(stderr, "mipc msg process error\n\r");

}

memset(buffer, 0, MIPC_DEFAULT_MAX_MSG_SIZE);

}

}
```

它在报文接收链表不为空时，取出报文链表头部开始的第一个报文，将其拷贝到一个固定缓存处，同时释放该报文动态申请的空间，调用函数mipc_msg_process处理报文，报文的处理方式非常简单，除了特殊的注册和解注册消息外，其余都是根据报文的目的是MIPC名称查表找到对应的消息队列ID并发送。

示例1：

下面是一个同步进程间通讯的示例：

某两个线程已经分别注册了MIPC条目，名称分别为"main_apm"和"alarm_apm"，现在条目名称为"main_apm"对"alarm_apm"发起进程间通讯，调用测试函数mipc_ping：

```
mipc_ping("alarm_apm", MIPC_MSG_SYNC_WAIT_FOREVER);
```

第二个参数MIPC_MSG_SYNC_WAIT_FOREVER的意思是“将一直等待直到收到回复为止”，mipc_ping的实现如下：

```
mipc_send_sync_msg(dest_name, MIPC_MSG_TYPE_PING, NULL, 0, &result, sizeof(int), timeout);
```

mipc_ping的实现就是调用函数mipc_send_sync_msg，参数dest_name就是"alarm_apm"即被访问线程的条目名称，第二个参数代码了报文类型为“PING”，第三个参数是实际负载的指针，这里为NULL表示实际负载无需任何内容，第四个参数是实际负载的长度，这里为0表示实际负载无需任何内容，第5个参数代表用于接收回复的指针，这里是个int型变量result的地址，第6个参数代表接收回复的长度，这里是int型变量的长度；下面是函数mipc_send_sync_msg的实现片段：

技术问答

快速回复

mipc_user_thread_info_find(src_name, &mqd, pthread_self());

首先调用函数mipc_user_thread_info_find，找到自己所在线程的MIPC条目的条目名称和消息队列ID，依靠的就是自己的线程ID(这个可以随时获取到)，通过查表找到；

mipc_clear_resp_msgq(src_name, mqd, api_id);

然后根据自己的消息队列ID和所要发送报文的消息类型，先将发给自己的这样的报文全部读取干净，这是为了避免已有的发给自己的这样消息类型的报文和一会收到的回复报文混淆；

mipc_send_msg(src_name, dest_name, api_id, inMsg, inSize, timeout)

接下来是发送报文，注意函数mipc_send_msg实际是把报文发送给收发中转进程，因为它不知道被访问进程对应的消息队列ID，它只能把自己的MIPC名称和被访问进程的MIPC名称作为索引，由收发中转进程去判断应该发给哪个消息队列；

mipc_receive_response_msg(src_name, mqd, api_id, outMsg, outSize, timeout)

最后是等待回复，该函数实际内容是通过自己的MIPC条目名称得出自己的MIPC条目消息队列ID，接收发给该消息队列ID且报文类型为api_id的消息，这样就避免了收错消息；当收到回复报文后把报文的实际负载内容取出拷贝到接收回复的地址处，这样就完成了一次同步的进程间通讯；另外可以设置超时值，默认为-1即永久等待；

中转收发进程在收到消息后，根据报文中指示的dst_name查表得出应该发往的消息队列ID并立即发送，这时相应进程的server处理线程就会收到该报文，并调用其hook处理函数处理，任何hook处理函数的子分支都会根据报文中的msg_type域找到应该调用的下一级处理函数，处理后得出返回值填入报文的实际负载，以对“PING”类型的报文处理为例：

case MIPC_MSG_TYPE_PING:

```
{

    int result = 0;

    mipc_response_msg(pMsg->msg_payload.src_name, pMsg->msg_payload.dst_name,
msgq_header->msg_type, &result, sizeof(int));

    return MIPC_OK;

}
```

上面就是每个进程的server处理线程的hook处理函数对于“PING“的处理，它调用函数mipc_response_msg发

技术问答

快速回复

送回复报文，同样发送给中转收发进程并且把回复内容填入实际负载，这里就是一个值为0的int型变量，符合发送者的要求；调用结束后，server处理完毕。

示例2：

下面的例子更加实用，以使用最多的中间件middleware为例，在文件middleware_mipc_client.c文件中包含了每一个访问middleware进程资源的client api函数，以其中一个为例：

ONU_STATUS middleware_insert_entry(MIDWARE_TABLE_ID_E table_id, VOID* entry, UINT32 size)

```
{

    middleware_insert_entry_IN_T input;

    middleware_insert_entry_OUT_T output;

    memset(&output, 0, sizeof(output));

    input.table_id = table_id;

    if (NULL != entry)

        memcpy(&input.entry, entry, size);

    input.size = size;

    // Call message queue or socket to send out the parameters

    if (MIPC_OK != mipc_send_sync_msg("middleware", 1, &input, sizeof(input), &output,
    sizeof(output), -1))

    {

        printf("%s: failed to send message\n", __FUNCTION__);

    }

    return output.ret;
```

技术问答

快速回复

```
}
```

任何一个注册了MIPC条目的线程都可以调用该api函数访问middleware进程，每一个api都是在调用mipc_send_sync_msg函数实现访问middleware进程，第一个参数标识了被访问MIPC条目的名称，第二个参数是消息类型也是报文类型，第三、第四个参数分别是输入参数内容及长度，第五、第六参数是接收回复的地址和长度，它们都是要被填入实际负载域；

由2.3节已知server进程是如何收到报文和大致的处理流程，下面是处理细节：

```
if (MIPC_ERROR != mipc_receive_msg(source_module, &api_id, inMsg))
```

```
{
```

```
    // Call APIs
```

```
    middleware_mipc_server_call(source_module, dest_module, api_id, msgq->msg_payload.payload);
```

```
}
```

```
.....,
```

```
void middleware_mipc_server_call(char* source_module, char* dest_module, int api_id, char* input)
```

```
{
```

```
    switch (api_id)
```

```
    {
```

```
        case 1:
```

```
        {
```

```
            middleware_insert_entry_IN_T* in = (middleware_insert_entry_IN_T*)input;
```


```
            middleware_insert_entry_OUT_T out;
```

```
            //实际执行需要做的处理
```

```
            out.ret = middleware_insert_entry(in->table_id, (void*)&in->entry, in->size);
```

```
            //把回复内容回复给发送者
```

 技术问答

 快速回复

```
// send back everyting in out data structure

if (MIPC_OK != mipc_response_msg(source_module, dest_module, 1, &out, sizeof(out)))

{

    printf("%s: failed to send response message\n", __FUNCTION__);

}

break;

}
```

首先通过调用函数mipc_receive_msg解析出报文的发送MIPC条目名称，这个是用来回复报文使用的；如何根据报文类型得出应该执行哪个具体处理函数，这里只列出case 1的情况，事实上有非常多的报文类型，在执行完毕后将回复内容作为参数调用函数mipc_response_msg，回复内容将被填入实际负载并发送给中转收发进程，中转收发进程会把该报文发给正在阻塞等待的访问middleware进程的线程MIPC条目。

总结：

上面是整改后marvell OMCI SDK内部使用的进程间通讯方式的描述，目前测试过程中运转正常，GPON也可以注册，但整改过程大幅度增删了原有代码，所以还有待于进一步检验。这种进程间通讯方式的效率并不高(进程A到进程B需要发送两次消息队列，如果需要接收回复则需要发送四次消息队列)，但总体而言比较安全稳定，实现了类似我们1.45代码的RPC方式的功能，即任何代码只要注册了client都可以向其他进程的server发送访问，保证了顺序访问同一进程，此外中转收发进程的报文接收和处理分开，最大程度的提高效率避免积压。



▼ 下一篇 linux设备模型____宏观印象

主题推荐

- 通讯
- 全局变量
- structure
- 中间件
- 结构

猜你在找

- 流行的通讯库消息中间件
 - linux 进程间通信--systemV 消息队列 实例
 - 多进程编程 Perl与C进程间的消息队列通信
 - Android -- 网络图片查看器网络html查看器 消息机制 消
- 消息中间件 activeMQ的源码分析 之 TCP通讯机制
 - 多进程间使用消息队列通信
 - 多进程编程Perl与C进程间的消息队列通信
 - 消息队列中间件的技术选型分析

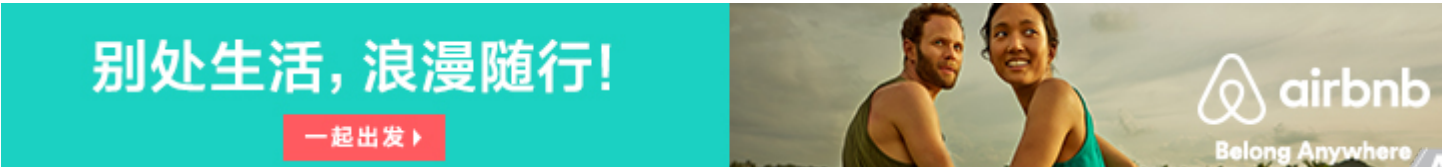
技术问答

快速回复

▪ 消息队列中间件ActiveMQ 准备好了么？！

▪ Apache Kafka 分布式消息队列中间件安装与配置 更多职位尽在 CSDN JOB

▪ linux高级工程师（即时通讯后台）	我要跳槽	▪ C++工程师（通信 通讯）	我要跳槽
深圳市云之讯网络技术有限公司	13-20K/月	深圳市三三得玖通信技术有限公司	7-14K/月
▪ linux高级工程师（VOIP通讯后台）	我要跳槽	▪ Java中级工程师（通信 通讯）	我要跳槽
深圳市云之讯网络技术有限公司	13-20K/月	深圳市三三得玖通信技术有限公司	12-17K/月



查看评论

暂无评论

您还没有登录,请[\[登录\]](#)或[\[注册\]](#)

* 以上用户言论只代表其个人观点，不代表CSDN网站的观点或立场

核心技术类目

全部主题 Hadoop AWS 移动游戏 Java Android iOS Swift 智能硬件 Docker OpenStack
VPN Spark ERP IE10 Eclipse CRM JavaScript 数据库 Ubuntu NFC WAP jQuery
BI HTML5 Spring Apache .NET API HTML SDK IIS Fedora XML LBS Unity
Splashtop UML components Windows Mobile Rails QEMU KDE Cassandra CloudStack FTC
coremail OPhone CouchBase 云计算 iOS6 Rackspace Web App SpringSide Maemo
Compuware 大数据 aptech Perl Tornado Ruby Hibernate ThinkPHP HBase Pure Solr
Angular Cloud Foundry Redis Scala Django Bootstrap

公司简介 | 招贤纳士 | 广告服务 | 银行汇款帐号 | 联系方式 | 版权声明 | 法律顾问 | 问题报告 | 合作伙伴 | 论坛反馈

网站客服 杂志客服 微博客服 webmaster@csdn.net 400-600-2320 | 北京创新乐知信息技术有限公司 版权所有 | 江苏乐知网络技术有限公司 提供商务支持

京 ICP 证 070598 号 | Copyright © 1999-2014, CSDN.NET, All Rights Reserved

技术问答

快速回复