从 API 开始理解 QNX

2009-05-25

个人来讲，喜欢通过 API 来理解一个系统。所以这篇文章也直接从讲解 API 开始。文章的对象是那些对微内核的结构有些了解后，而想实实在在的编些程序的年轻 IT 民工们。

如果你的实际工作是在 QNX 上编程的话，刚开头的这部份，可能还不能立刻派上用场。不过，理解这部份知识，对于将来理解更上层的 API 会有很大的帮助；而且，有兴趣的同学，也可以从这里管中窥豹地看一看 QNX 是如何做到"商用微内核"系统的。

消息传递

大家都知道 QNX 是个基于微内核结构的操作系统，靠的是进程间通讯来实现整个系统功能的。整个 QNX 上的服务，基本上都是服务器进程与客户端进程的通信来完成的。当然很多时候，一个进程既是服务器，也是客户端。

那么具体到写一个程序的时候，到底这个通讯是如何完成的呢？这章就是具体介绍最底层的消息传递 API 的。

消息传递是通过内核进行的，所以所谓的 API，实际也就是最底层的内核调用了。需要再次指出的是，真正在 QNX 上写程序的时候，很少会直接用到这些 API，而是利用更高层的 API，不过，知道这些底层的 API 对于将来理解建立在这些 API 上的界面，应该会有帮助的。下面讲解中，也会穿插一些实际的例子。

频道（Channel）与连接（Connect）

消息传递是基于服务器与客户端的模式来进行的，那么客户端怎样才能与服务器端通讯呢？最简单的，当然是指定对方的进程号。要发送的一方，将消息加一个头，告诉内核"把这个消息发给进程号为 12345 的进程"就行了。

但是 QNX 完整支持 POSIX 线程后，这种方法会有问题的。想一想，如果服务器有两个线程，同时提供两个不同的服务，那客户端要怎么发送给特定的服务呢？或者你会说"把这个消息发给进程 12345 的线程 3"就

行了。可是，如果某一个服务，不是由单一线程来进行服务的，而是有一组线程进行的，那又怎么办呢？

为此，QNX 抽象出了"频道"（Channel）这个概念。一个频道，就是一个服务的入口；至于这个频道到底具体有多少线程为其服务，那都是服务器端自己的事情。一个服务器如果有多个服务，它也可以开多个频道。这个概念还是比较好理解的，拿 Socket 编程来比方，"频道"就有点像端口号，一个端口号对应了一个服务。可以用多个线程 listen()在同一个端口上，并发处理服务。

而客户端，在向特定"频道"发送消息前，需要先建立连接（Connection），然后将消息在连接上发出去。这样同一个客户端，如果需要，可以与同一个频道建立多个连接。这个过程，有 Socket 编程经验的同学应该不难理解。

所以，大致上通讯的准备过程是这样的：

服务器

```
ChannelId = ChannelCreate(Flags);
```

客户端

```
ConnectionId = ConnectAttach(Node, Pid, Chid, Index, Flag);
```

服务器端就不用解释了，客户端要建立连接的话，它需要 Node，这个就是机器号。如果过 QNX 的 QNET 网络（透明分布处理）进行消息传递时，这个值决定了哪一台机器；如果客户端与服务器在同一台机器里时，这个数字是 0，或者说 ND_LOCAL_NODE；pid 是服备器的进程号；而 chid 就是服务器调用 ChannelCreate()后得到的频道号了。Index 与 Flag 以后再讨论。基本上客户端就是同 " Node 这台机器里的，Pid 这个进程的，Chid 频道 " 做一个连接。有了连接以后，就可以进行消息传递了。

连接的终止是 ConnectDetach()，而频道的结束则是 ChannelDestroy()了。不过，一般服务器都是长久存在的，不大有需要 ChannelDestroy()的时候。

发送(Send)，接收(Receive)和应答(Reply)

QNX 的消息传递，与我们传统常见的进程间通讯最大的不同，就是这是一个 " 同步的 " 消息传递。一个消

息传递，都要经过发送，接收和应答三个部份，所谓的 SRR 过程。这三步的每一步都会发生阻塞，这是 QNX 一个很重要也是很基本的概念。

具体来说，客户端在连接上服务器后，就可以"发送"消息；一旦发送，不论服务器端收到没有，客户端会被阻塞。

服务器端会接收到消息，进行处理，最后，将处理结果"应答"给客户端；只有服务器"应答"了以后，客户端的 REPLY_BLOCK 阻塞状态才会被解除。客户端得以继续执行"发送消息"的下一条语句。

这种同步的过程，严格保证了客户端与服务器端的时序，也大大简化了编程。具体用 API 来说，就是这样。

服务器

```
ReceiveId = MsgReceive(ChannelId, ReceiveBuffer, ReceiveBufLength, &MsgInfo);
(...  检查 Buffer 里的消息进行处理 ...)
MsgReply(RceeiveId, ReplyStatus, ReplyBuf, ReplyLen);
```

客户端

```
MsgSend(ConnectionId, SendBuf, SendLen, ReplyBuf, ReplyLen);
(... 由 OS 将这个线程挂起 ...)
(... 当服务器 MsgReply()后，OS 解除线程的阻塞状态, 客户端可以检查自己的 ReceiveBuf 看看应答结果 ...)
```

服务器端在频道上进行接收，处理消息，完成后应答；客户端则是在连接上发送，要注意在发送的同时，客户端还同时已经提供了接收应答用的缓冲。

如果你细心的话，或许你会问，服务器端的 MsgReceive()与客户端的 MsgSend()没有同步，会不会有问题呢？比如，如果 MsgSend()时，服务器没有在 MsgReceive()，会出什么事呢？答案是 OS 依然会把发送线程挂起，发送线程从执行状态(RUNNING)转入"发送阻塞"状态(SEND BLOCK)，一直等到服务器来 MsgReceive()时，再将 SendBuf 里的东西复制到 ReceiveBuffer 里去，同时发送线程的状态变成 "应答阻塞"(REPLY BLOCK)。在 QNX 上用 pidin 查看进程、线程状态时，可以看到很多线程在 REPLY 状态下，就是这个原因。说明该线程向服务器发送了消息，但还没有得到应答。在 QNX 里这是非常常见的情况，比如你去 read()一个串口，串口驱动就是你的服务器，当没有数据进来时，串口驱动就不应答你，造成你的程序一直在 read() 里等着。

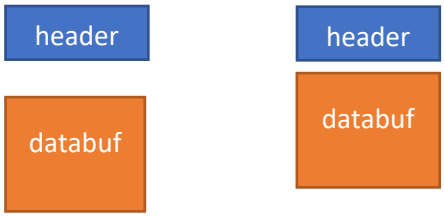
一般不太容易看到 SEND_BLOCK 的线程，因为这意味着服务器已经不能正常提供服务了。

同样的，如果服务器调用 MsgReceive()时，没有客户端，服务器线程也会被挂起，进入"接收阻塞"状态(RECEIVE BLOCK)。这个也可以在 pidin 里经常看到。

在应答时，还可以用 MsgError()来告诉发送方有错误发生了。因为 MsgReply()也可以返回一个状态，或许你会问这两者之间有什么区别？MsgReply(rcvid, EINVAL, 0, 0);的结果是，MsgSend() 这个函数的返回值是 22(EINVAL)；而 MsgError(rcvid, EINVAL)；的结果，是 MsgSend()返回-1，而 errno 被设为 EINVAL。

数据区与 iov

除了用线性的缓冲区进行消息传递以外，为了方便使用，QNX 还提供了用 iov_t 来"汇集"数据。也就是说，可以一次传送几块数据。好象下面的图这样子。虽然在客户端蓝色的 Header 同红色的 databuf 是两块不相邻的内存，但传递到服务器端的 ReceiveBuffer 里，就是连续的了。也就是说在服务器端，要想得到原来 databuf 里的数据，只需要(ReceiveBuffer + sizeof(header))就可以了。（要注意数据结构对齐）



客户端　　服务器端

客户端

```
SETIOV(&iov[0], &header, sizeof(header));
SETIOV(&iov[1], databuf, datalen);
MsgSendvs(ConnectionId, iov, 2, Replybf, ReplyLen);
```

"header" 与 "databuf" 是不连续的两块数据。

服务器接收后，"header"与"databuf"被连续地存在 ReceiveBuffer 里。

```
ReceiveId = MsgReceive(ChannelId, ReceiveBuffer, ReceiveBufLength, &MsgInfo);

header = (struct header *)ReceiveBUffer;
databuf = (char *)((char *)header + sizeof(*header));
```

例子

好了，有了以上这些基本函数（内核调用），我们就可以写一个客户端和一个服务器端，进行最基本的通信了。
服务器：这个服务器，准备好频道后，就从频道上接收信息。如果信息是字符串"Hello"的话，这个服务器应答一个"World"字符串。如果收到的信处是字符串"Ni Hao", 那么它会应答"Zhong Guo"，其它任何消息都用 MsgError()回答一个错误。

```
$ cat simple_server.c

// Simple server
#include <errno.h>
#include <stdio.h>
#include <string.h>
#include <sys/neutrino.h>

int main()
{
    int chid, rcvid, status;
    char buf[128];

    if ((chid = ChannelCreate(0)) == -1) {
        perror("ChannelCreate");
        return -1;
    }

    printf("Server is ready, pid = %d, chid = %d\n", getpid(), chid);

    for (;;) {
        if ((rcvid = MsgReceive(chid, buf, sizeof(buf), NULL)) == -1) {
            perror("MsgReceive");
            return -1;
        }

        printf("Server: Received '%s'\n", buf);

        /* Based on what we receive, return some message */
        if (strcmp(buf, "Hello") == 0) {
            MsgReply(rcvid, 0, "World", strlen("World") + 1);
        } else if (strcmp(buf, "Ni Hao") == 0) {
```

```
                MsgReply(rcvid, 0, "Zhong Guo", strlen("Zhong Guo") + 1);
            } else {
                MsgError(rcvid, EINVAL);
            }
        }

        ChannelDestroy(chid);
        return 0;
}
```

客户端：客户端通过从命令行得到的服务器的进程号与频道号，与服务器建立连接。然后向服务器发送三遍"Hello"和”Ni Hao"，并检查返回值。最后发一个”unknown"看是不是 MsgSend()会得到一个出错返回。

```
$ cat simple_client.c

//simple client
#include <stdio.h>
#include <string.h>
#include <sys/neutrino.h>

int main(int argc, char **argv)
{
        pid_t spid;
        int chid, coid, i;
        char buf[128];

        if (argc < 3) {
                fprintf(stderr, "Usage: simple_client <pid> <chid>\n");
                return -1;
        }

        spid = atoi(argv[1]);
        chid = atoi(argv[2]);

        if ((coid = ConnectAttach(0, spid, chid, 0, 0)) == -1) {
                perror("ConnectAttach");
                return -1;
        }

        /* sent 3 pairs of "Hello" and "Ni Hao" */
        for (i = 0; i < 3; i++) {
```

```
        sprintf(buf, "Hello");
        printf("client: sent '%s'\n", buf);
        if (MsgSend(coid, buf, strlen(buf) + 1, buf, sizeof(buf)) != 0) {
            perror("MsgSend");
            return -1;
        }
        printf("client: returned '%s'\n", buf);

        sprintf(buf, "Ni Hao");
        printf("client: sent '%s'\n", buf);
        if (MsgSend(coid, buf, strlen(buf) + 1, buf, sizeof(buf)) != 0) {
            perror("MsgSend");
            return -1;
        }
        printf("client: returned '%s'\n", buf);
    }

    /* sent a bad message, see if we get an error */
    sprintf(buf, "Unknown");
    printf("client: sent '%s'\n", buf);
    if (MsgSend(coid, buf, strlen(buf) + 1, buf, sizeof(buf)) != 0) {
        perror("MsgSend");
        return -1;
    }

    ConnectDetach(coid);

    return 0;
}
```

分别编译后的执行结果是这样的：

服务器：

```
$ ./simple_server
Server is ready, pid = 36409378, chid = 2
Server: Received 'Hello'
Server: Received 'Ni Hao'
Server: Received 'Hello'
Server: Received 'Ni Hao'
Server: Received 'Hello'
Server: Received 'Ni Hao'
Server: Received 'Unknown'
```

```
Server: Received ''
```

客户端：

```
$ ./simple_client 36409378 2
client: sent 'Hello'
client: returned 'World'
client: sent 'Ni Hao'
client: returned 'Zhong Guo'
client: sent 'Hello'
client: returned 'World'
client: sent 'Ni Hao'
client: returned 'Zhong Guo'
client: sent 'Hello'
client: returned 'World'
client: sent 'Ni Hao'
client: returned 'Zhong Guo'
client: sent 'Unknown'
MsgSend: Invalid argument
```

可变消息长度

从上面的程序也可以看出来，消息传递的实质是把数据从一个缓冲，复制到（另一个进程的）另一个缓冲里去。问题是，如何确定缓冲的大小呢？上述的例子里，服务器端用了一个 128 字节的缓冲，万一客户端发送一个比如说 512 字节的消息，是不是消息传递就会出错了呢？

答案是，消息传递依然成功，但是，只有 SendBuffer 的最初的 128 个字节的数据会被复制。这个设计背后的思想是，服务器预先是不可能知道客户端要发多少数据的，所以它一定需要正确地发现这样的情形，并设法取得完整的数据。

在 MsgRecieve()时，第四个参数是一个 struct _msg_info。内核会在进行消息传递的同时，填充这个结构，从而告诉让你得到一些信息。在这个结构中，"msglen"告诉你这次消息传递你实际收到了多少字节（在我们的例子里，就是 128），而"srcmsglen"则告诉你发送方的实际 Buffer 会有多大（在我们的例子里，是 512）。通过比较这两个值，服务器端就可以判断有没有收到全部数据。

一旦服务器知道了还有更多的数据没有收到，那该怎么办呢？QNX 提供了 MsgRead()这个特殊函数。服务器端可以用这个函数，从发送缓冲中"读取"数据。MsgRead()基本上就是告诉内核，从发送缓冲的某个指定偏移开始，读取一定长的数据回来。所以服务器端这部份的代码基本上是这样的。

```
int rcvid;
struct _msg_info info;
char buf[128], *totalmsg;

...

rcvid = MsgReceive(chid, buf, 128, &info);
...
if (info->srcmsglen > info->msglen) {
    totalmsg = malloc(info->srcmsglen);
    if (!totalmsg) {
        MsgError(rcvid, ENOMEM);
        continue;
    }
    memcpy(totalmsg, buf , 128);
    if (MsgRead(rcvid, &totalmsg[128], 128, info->srcmsglen - info->msglen) == -1) {
        MsgError(rcvid, EINVAL);
        continue;
    }
} else {
    totalmsg = buf;
}

/* Now totalmsg point to a full message, don't forget to free() it later on,
 * if totalmsg is malloc()'d here
 */
```

你或者会问，为什么服务器都已经接收数据了，怎么还能再去读取客户端的数据？这是因为从一开始我们就提到的，QNX 的消息传递是"同步"的。还记得吗？在服务器端"应答"之前，客户端是被阻塞的；也说是说客户端的发送缓冲会一直保留在那里，不会变化。（另外再开个线程去把这个缓冲搞乱甚至 free 掉？当然可以。不过，这是你客户端程序的 BUG 了）

与此相近的，有的时候，服务器需要返回大量的数据给客户端（比如说 1M)。服务器不希望 malloc(1024 * 1024)，然后 MsgReply()，然后再 free()。（在嵌入式程序里，经常地进行 malloc()/free()不是一个很好的习惯）那么服务器也可以用一个小的定长缓冲，比方说 16K，然后把数据"一部份一部份地写回"客户端的应答缓冲里。好象下面的样子。要记得最后还是要做一个 MsgReply() 以让客户端继续运行。

```
char *buf[16 * 1024];
unsigned offset;
```

```
for  (offset = 0; offset < 1024 * 1024; offset += 16 * 1024) {
    /* moving data into buffer */
    MsgWrite(rcvid, buffer, 16 * 1024, offset);
}
/* 1MB returned, Reply() to let client go */
MsgReply(rcvid, 0, 0, 0);
```

实例

QNX 公司大约在 2007 年时，在网上公布过包括内核、C 库和大量系统应用的源码。以下是 QNX 的 C 库中的 read()和 write()函数实装，有了前面的基础，应该很好理解了。

先不管 fd 是如何得到的，只要理解 fd 就是 ConnectAttach()返回的连接号就可以了。虽然 read()是从服务器取得数据，而 write()是向服务器输出数据，但实质上，它们都是向服务器提出一个请求，由服务器来应答。而对于 write()来说，这是一个 io_write_t，一个 MsgWritev()把请求与要传递的数据一起发给服务器；而对于 read()来说，请求被封装在 io_read_t 里，MsgSend()把这请求传给服务器，read()的结果缓冲，则做为应答缓冲，由服务器 MsgReply()时填入。

read():

```
/*
 * $QNXLicenseC:
 * Copyright 2007, QNX Software Systems. All Rights Reserved.
 *
 * You must obtain a written license from and pay applicable license fees to QNX
 * Software Systems before you may reproduce, modify or distribute this software,
 * or any work that includes all or part of this software. Free development
 * licenses are available for evaluation and non-commercial purposes. For more
 * information visit http://licensing.qnx.com or email licensing@qnx.com.
 *
 * This file may contain contributions from others. Please review this entire
 * file for other proprietary rights or license notices, as well as the QNX
 * Development Suite License Guide at http://licensing.qnx.com/license-guide/
 * for other information.
 * $
 */
```

```
#include <unistd.h>
#include <sys/iomsg.h>

ssize_t read(int fd, void *buff, size_t nbytes) {
        io_read_t       msg;

        msg.i.type = _IO_READ;
        msg.i.combine_len = sizeof msg.i;
        msg.i.nbytes = nbytes;
        msg.i.xtype = _IO_XTYPE_NONE;
        msg.i.zero = 0;
        return MsgSend(fd, &msg.i, sizeof msg.i, buff, nbytes);
}
```

write():

```
/*
 * $QNXLicenseC:
 * Copyright 2007, QNX Software Systems. All Rights Reserved.
 *
 * You must obtain a written license from and pay applicable license fees to QNX
 * Software Systems before you may reproduce, modify or distribute this software,
 * or any work that includes all or part of this software. Free development
 * licenses are available for evaluation and non-commercial purposes. For more
 * information visit http://licensing.qnx.com or email licensing@qnx.com.
 *
 * This file may contain contributions from others. Please review this entire
 * file for other proprietary rights or license notices, as well as the QNX
 * Development Suite License Guide at http://licensing.qnx.com/license-guide/
 * for other information.
 * $
 */




#include <unistd.h>
#include <sys/iomsg.h>

ssize_t write(int fd, const void *buff, size_t nbytes) {
```

```
    io_write_t          msg;
    iov_t       iov[2];

    msg.i.type = _IO_WRITE;
    msg.i.combine_len = sizeof msg.i;
    msg.i.xtype = _IO_XTYPE_NONE;
    msg.i.nbytes = nbytes;
    msg.i.zero = 0;
    SETIOV(iov + 0, &msg.i, sizeof msg.i);
    SETIOV(iov + 1, buff, nbytes);
    return MsgSendv(fd, iov, 2, 0, 0);
}
```

服务器端应该是怎样进行处理的？想想 MsgRead()/MsgWrite()，你应该不难想像服务器端是如何工作的吧。

另外可以看到，“消息发送一直阻塞到收到数据”这个特性，很好地符合了 Posix 关于 read()/write()一直阻塞到函数完成的定义。你调用 read()，等函数返回时，要么正确地读到了数据，要么出错了，对吧。

脉冲(Pulse)

脉冲其实更像一个短消息，也是在“连接”上发送的。脉冲最大的特点是它是异步的。发送方不必要等接收方应答，直接可以继续执行。但是，这种异步性也给脉冲带来了限制。脉冲能携带的数据量有限，只有一个 8 位的"code"域用来区分不同的脉冲，和一个 32 位的“value"域来携带数据。脉冲最主要的用途就是用来进行“通知”(Notification)。不仅是用户程序，有时候内核也会生成特殊的“系统脉冲”发送到用户程序，以通知某一特殊情况的发生。
脉冲的接收比较简单，如果你有一个频道只接受脉冲的话，可以用 MsgReceivePulse()来接收脉冲；如果频道既可以接收消息，也可以接收脉冲时，就直接用 MsgReceive()，只要确保接收缓冲(ReveiveBuf)至少可以容下一个脉冲（sizeof struct _pulse)就可以了。在后一种情况下，如果 MsgReceive()返回的 rcvid 是 0，就代表接收到了一个脉冲，反之，则收到了一个消息。所以，一个既接收脉冲，又接收消息的服务器，可以是这样的。

```
union {
    struct _pulse pulse;
    msg_header   header;
} msgs;

...

if ((rcvid = MsgReceive(chid, &msgs, sizeof(msgs), &info)) == -1) {
    perror("MsgReceive");
```

```
        continue;
    }

    if (rcvid == 0) {
        process_pulse(&msgs, &info);
    } else {
        process_message(&msgs, &info);
    }
```

脉冲的发送，最直接的就是 MsgSendPulse()。不过，这个函数一般不太会直接用到。比较可能的用法是在一个进程中，有一个线程要通知另一个服务器线程什么事情的情形。在更多的时候，以及跨进程的时候，通常不会用到这个函数，而是用到下面将要提到的 MsgDeliverEvent()。
与消息传递相比，消息传递永远是在用户进程间进行的。也就是说，不会有内核向某一个进程发送数据的情形。而脉冲就不一样，除了用户进程间可以发脉冲以外，内核也会向用户进程发送"系统脉冲"来通知某一事件的发生。

消息传递的方向与 MsgDeliverEvent()

从一开始就提到，QNX 的消息传递是客户端、服务器型的。也就是说，总是由客户端向服务器端发送请求，等待被回复的。但在现实情况中，客户端与服务器端并不是很容易区分开来的。有的服务器端为了处理客户端的请求，本身就需要向别的服务器发送消息；有的客户端需要从不同的服务器那里得到服务，而不能阻塞在某一特定的服务器上；还有的时候，两个进程间的数据是互相流动的，这应该怎么办呢？

也许有人认为，两个进程互为通讯就可以了。每个进程都建立自己的频道，然后都与对方的频道建一个连接就好了；这样，需要的时候，就可以直接通过连接向对方发送消息了。就好象管道(pipe)或是 socketpair 一样。请注意，这种设计在 QNX 的消息传递中是应该避免的。因为很容易就造成死锁。一个常见的情形是这样的。

进程 A：MsgSend() 到进程 B
进程 B：MsgReceive()接收到消息
进程 B：处理消息，然后 MsgSend()给进程 A

因为进程 A 正在阻塞状态中，无法接收并处理 B 的请求；所以 A 会在 REPLY_BLOCK 里，而 B 则会因 MsgSend()但无人接收数据而进入 SEND_BLOCK，两个进程就互为死锁住了。当然，如果 A 和 B 都使用多

线程，专门用一个线程来 MsgReceive()，这个情形或许可以避免；但你要保证 MsgReceive()的线程不会去 MsgSend()，否则一样会死锁。在程序简单的时候或许你还有控制，如果程序变得复杂，又或者你写的只是一个程序库，别人怎么来用你完全没有控制，那么最好还是不要用这种设计。

在 QNX 中，正确的方法是这样的。

客户端： 准备一个"通知事件"(Notification Event)，并把这个事件用 MsgSend()发给服务器端，意思是："如果 xxx 情况发生的话，请用这个事件通知我"。
服务器： 收到这个消息后，记录下当时的 rcvid，和传过来的事件，然后应答"好的，知道了"。
客户端： 因为有了服务器的应答，客户端不再阻塞，可以去做别的事
……
服务器： 在某个时刻，客户端所要求的"xxx 情况"满足了，服务器调用 MsgDeliverEvent(rcvid, event);以通知客户端
客户端： 收到通知，再用 MsgSend()发关"xxx 情况的数据在哪里？"
服务器： 用 MsgReply()把数据返回给客户端

具体的例子，可以参考 MsgDeliverEvent()的文档说明。

路径名（Path Name）

现在来回想一下我们最初的例子，客户端与服务器是怎样取得连接的？客户端需要服务器的 nd, pid, chid，才能与服务器正确地建立连接。在我们的例子里，我们是让服务器显示这几个数，然后在客户端的启动时，通过命令行里传给客户端。但是，在一个现实的系统里，进程不断地启动、终止；服务器与客户端的起动过程也无法控制，这种方法显然是行不通的。

QNX 的解决办法，是把"路径名"与上述的"服务频道"概念巧妙地结合起来。让服务器进程可以注册一个路径名，与服务频道的 nd, pid, chid 关联起来。这样，客户端就不需要知道服务器的 nd, pid, chid，而只要请求连接版务器路径名就可以了。具体来说 name_attach()就是用来建立一个频道，并为频道注册一个名字的；而 name_open()则是用来连接注册过的服务器频道；具体的例子，可以在 name_attach()的文档里找到，这里就不再重复了。

除了 name_attach()/name_open(),在资源管理器时还会讲到资源管理器是如何获取路径名的。同时路径名（在资源管理器时也称做命名空间 name space）或者说命名空间的管理，水也相当地深。有机会再另开章节讨论吧。