QNX 资源管理器

资源管理器(resource manager)应该是 QNX 编程中最常用的，如果没写过几个资源管理器，大概都不好意思说自已做过 QNX 编程。唯其重要，QNX 也有很多官方文档解释。这篇文章详细解释了 QNX 资源管理器的基本设计，管理器框架，并辅以代码。希望能够帮助读者掌握 QNX 上的资源管理器，从而更容易地搭建系统。

文章里提到的资源管理器示例，完整代码可以在下面的链接里找到
https://github.com/xtang2010/articles/tree/master/QNX_Resource_Manager

什么是资源管理器

资源管理器顾名思义就是管理“资源”的服务器，这里问题是，到底什么是“资源”呢？在 QNX 上，“资源”可以是一个硬件(硬件资源管理器其实就是我们常说的硬件驱动)，“资源”也可以是一种服务，比如 TCPIP 网络服务，或者 ntfs 文件系统服务；“资源”甚至可以是一个文件(或者目录)。

如果你还记得，Unix 的基本思想，就是“把驱动当成文件”，那资源管理器就非常有用了。所以/dev/ser1 是一个管理串口的资源管理器，而/dev/random 则是一个提供随机数的资源管理器。甚至传统 Unix 里那些 mount point，像是根目录 /，或者用户目录 /home 在 QNX 里也可以是一个个资源管理器。

好，实战最重要，假设我们要写一个 md5 的资源管理器, 要怎么办呢？md5 是这样一个服务，假设客户端传给它一串数据, 它计算 md5 值, 然后让客户端来取。

一个最基本的资源管理器

一个资源管理器基本上来说就是一个服务器。只要看过《从 API 开始理解 QNX》那篇的同学应该很容易就可以想到最基本的资源管理器，应该就是下面这样的:

(这里只是示意代码，完整的可以在 QNX 上编译运行的代码 md5name 可以在 GitHub 上找到)

```
#define MD5_SEND_DATA   0x00000001
#define MD5_RECV_DIGEST 0x00000002

typedef struct {
   int msgtype;
   int msglen;
} md5_msg_t;

name_attach(...);
for (;;){
   revid = MsgReceive(chid, &msg, sizeof(msg), &info);
   switch msg.msgtype {
   case MD_SEND_DATA:
       MsgRead(rcvid, ...);
        ....
       MsgReply(rcvid, ...);
       break;
```

```
    case MD_RECV_DIGEST :
        ....
        MsgReply(rcvid, …)
        break;
    }
}
```

对的, 就是一个循环不断收信息, 然后按定好的信息类型进行处理就好了。客户端要怎样获取这个服务呢?

```
int md5_send(int fd, unsigned char *data, int len)
{
    md5_msg_t msg;
    iov_t iov[2];

    msg.msgtype = MD5_SEND_DATA;
    msg.msglen = len;
    SETIOV(&iov[0], &msg, sizeof(msg));
    SETIOV(&iov[1], data, len);
    return MsgSendv(fd, iov, 2, 0, 0);
}

int md5_recv(int fd, unsigned char *digest, int len)
{
    md5_msg_t msg;

    if (len < 16) {
        errno = EINVAL;
        return -1;
    }

    msg.msgtype = MD5_RECV_DIGEST;
    msg.msglen = len;
    return MsgSend(fd, &msg, sizeof(msg), digest, len);
}
```

当然做为一个资源管理器的开发者, 你会把 md5_send(), md5_recv() 函数打包成一个库，让别人来使用你的服务。

```
int main(int argc, char **argv)
{
    if ((fd = name_open("md5name", 0)) == -1) {
        perror("name_open");
        return -1;
    }

    if ((cfd = open(argv[1], O_RDONLY)) == -1) {
        perror("open");
        return -1;
    }

    total = 0;
```

```
    for (;;) {
        n = read(cfd, buf, 16 * 1024);
        if (n <= 0)
              break;

        if (md5_send(fd, buf, n) <= 0) {
              perror("md5_send");
              return -1;
        }

        total += n;
    }

    md5_recv(fd, digest, 16);
    printf("%10d\t\t\t", total);
    for (n = 0; n < 16; n++) {
        printf(" %02X", digest[n]);
    }
    printf("\n");
    return 0;
}
```

看上去一切都完成了，但是，没有问题吗?

首先这个服务器只能一次处理一个请求，如果有两个程序同时申请这服务时，逻辑上有错误。另外，这个解决方案，意味着每个想要使用这个 md5 服务的客户端都要链接一个专用的库, 难道没有办法做得通用一点吗?

答案当然是有的。既然 md5 服务是通过名字来提供服务的，而 POSIX 对于文件可以进行的操作是有标准定义的呀, 对于我们这个服务, 立刻可以想到的就是客户端可以直接 write() 数据到比如 /dev/md5, 然后 read() 结果的呀。

那么，资源管理器要怎么做呢，如果客户端 write() 时, 我会收到什么信息? QNX 已经为你准备好了，这就是，资源管理器框架。

资源管理器框架

因为一个资源管理器都是由一个路经名作为入口的，而 POSIX 又定义了对一个文件可进行的操作，所以 QNX 就替大家预定义了这些操作所需要传递的消息类型和数据格式

QNX 提供的资源管理器框架大致可以分成 4 个部份：

- **iofunc (iomsg) 层**，这一层提供了所有 POSIX 对文件可以进行的 io 操作 (sys/iofuncs.h，sys/iomsg.h)
- **resmgr 层**，这一层提供了登记路径名，接收数据并分发给 iofunc 执行具体操作。iofunc 和 resmgr，是写一个资源管理器的基础

- **dispatch 层**，在一些复杂的资源管理器里，“外来的消息传递”并不是唯一需要处理的。也有可能需要处理“脉冲”，或者有时候一个“信号”。dispatch 层会主动识别不同的输入信息，然后转给不同的处理函数进行处理
- **thread pool 层**，这一层提供了一个线程池管理，可以配置实现多个线程进行资源管理。

下面我们深入地看一下各层

iofunc (iomsg) 层:

QNX 总结了总共 34 个对文件操作，基本上 POSIX 对文件的处理，都可以通过这 34 个操作进行。而资源管理器的 iofunc 层，其实也就是准备回调函数，通过响应这些操作请求，来提供服务。

这 34 个回调函数，又根据性质不同，被分为 8 个 “connect" 回调函数，和 26 个 "io" 回调函数。这些函数在 io_func.h 里都有定义，分别是：

Connect Functions	IO Functions
open	read
unlink	write
rename	close_ocb
mknod	stat
readlink	notify
link	devctl
unblock	unblock
mount	pathconf
	lseek
	chmod
	chown
	utime
	openfd
	fdinfo
	lock
	space
	shutdown
	mmap
	msg
	umount
	dup
	close_dup
	lock_ocb
	unlock_ocb
	sync
	power

iofunc 层的每一个回调函数，都有一个对应的数据结构供客户端和服务器端进行消息传递，在 sys/iomsg.h 里。

比如计对 io_read，就有一个：

```
typedef union {
      struct _io_read i;
      /* unsigned char data[nbytes]; */
} io_read_t;
```

可以看出来 io_read，有一个从客户端发到服务器端的 struct _io_read I; 而服务器端到客户端，没有特别的消息结构，就是读到的数据直接返回了。

再比如 io_stat，就是：

```
typedef union {
      struct _io_stat    i;
      struct  stat        o;
} io_stat_t;
```

可以看到，io_stat 是从客户端向服务器端发送一个 stauct _io_stat; 而服务器端返回的，直接就是一个 struct stat o; 这个 struct stat, 就是 POSIX 的 stat() 函数取到的返回值。

如果你仔细观察 struct _io_read; struct _io_stat 这些数据结构，你会发现它们都是开始于：

```
{
      _Uint16t  type;
      _Uint16t  combine_len;
      …
}
```

这里，”type”可以告诉收到这个数据结构的服务器，这是个什么信息，“combined_len”则通知了服务器端这个消息（以及它可能携带的可变长参数）一共有多长。

"type"的定义，已经在 sys/iomsg.h 里定义好了, 比如“_IO_READ", "_IO_WRITE", "_IO_STAT"…

我们在《从 API 开始理解 QNX》一文里解释过一个 read()函数是怎样发送数据的了。再看一下：

```
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);
}
```

所以，对照 iomsg.h 里的数据结构，你不难想像 libc 里那些 POSIX 标准文件函数是怎样构造出来的吧，write() / stat() / lseek() / …

所有 QNX 上的程序都调用这些 libc 里的函数，而在函数里，它们都转化成一个个消息，通过 fd，发送到相关的资源服务器去了。

普通的 QNX 上的开发者，只是调用了这些 read()/write()/stat()函数，然后得到结果，并不关心里面是怎么实现的了。这也是为什么好多基本的 Unix 上的程序，都可以轻松在 QNX 上重新编译的原因。

在资源服务器端，根据 iomsg 里的 type，就能判断出这是一个什么样的请求，然后调用 iofunc 里相应的处理程序处理就好了。

也许你已经想到下一步这个问题了，每写一个资源管理器，都需要准备三十几个回调函数，这也太复杂了。而且，很多时候，资源管理器只是通过路径名提供一个服务，并不需要提供所有的 POSIX 标准处理。比如 /dev/random 是提供随机数的一个服务，应该没有必要 lseek() 它吧。那么，从资源管理器的角度，可不可以不提供 IO_SEEK 回调函数码?

答案当然是肯定的，这就会介绍到 resmgr 层了。

resmgr 层

如果 iofunc 层提供了各种回调函数的话，resmgr 层就是那个“循环按收信息并调用 iofunc”的那一层。几个重要的函数差不多就是这样。

```
resmgr_attach(….)
ctp = resmgr_context_alloc()
for (;;) {
      ctp = resmgr_block(ctp);
      resmgr_handler(ctp)
}
```

resmgr 层和 iofunc 层给合，就可以搭建一个资源管理器。让我们用一个简单的例子来看一下。

我们这次打算写一个 "/dev/now" 的资源管理器，当有人读它时，它会把当前的时间用字符串返回。让我们看一下代码：

```
/* 建一个 dispatcher，这个可以想像就是一个接收数据的频道 */
dispatch = dispatch_create();

/* 资源管理器本身的一些参数，下面这个就是指定了资源管理器最多一次可以处理 10 个 iov_t
*/
memset( &res_attr, 0, sizeof( res_attr ) );
res_attr.nparts_max = 10;
res_attr.msg_max_size = 0;

/* io_attr 其实可以想像成一个文件相关的参数，比如读写权限等等 */
iofunc_attr_init(&io_attr, 0666 | S_IFCHR, 0, 0);
```

```
/* 初始化 iofunc 层回调 */
iofunc_func_init( _RESMGR_CONNECT_NFUNCS, &connect_funcs,
            _RESMGR_IO_NFUNCS, &io_funcs );

/* 建立起资源管理层，同时注册路径 */
rmgid = resmgr_attach(dispatch, &res_attr, "/dev/now", _FTYPE_ANY, 0,
                &connect_funcs, &io_funcs,&io_attr);

/* 准备一个资源管理层的 context 以备使用 */
ctp = resmgr_context_alloc(dispatch);

while (1)
{
      /* resmgr_block() 相当于做了一个 MsgReceive() */
      ctp = resmgr_block(ctp);

      /* 根据收到的信息，调用相应的回调程序 */
      resmgr_handler(ctp);
}
```

如果你把上述几行代码，放在一个 main() 里编译执行的话，你就会发现系统里多了一个 /dev/now 的文件。

```
$ ls -la /dev/now
crw-rw-rw-   1 root       root        0,  1 Jun 01 01:46 /dev/now
```

这个服务，因为我们根本还没有写 iofunc 的回调函数，当然什么服务也不会提供。

但是，等一下？如果一个回调函数都没写，ls -la 是怎么成功的？"ls -la /dev/now" 基本上做的就是：

```
fd = open("/dev/now", O_RDONLY);
stat(fd, &mystat);
close (fd)
printf()
```

也就是说，resmgr 至少已经帮我们实现了 io_open(), io_stat(), io_close() 这几个回调函数。对吧。而 "crw-rw-rw-" 其实就是因为我们初始化 io_attr 时给出的参数。

让我们加一个回调函数，来实现 /dev/now 的功能。

```
static int now_read(resmgr_context_t *ctp, io_read_t *msg, RESMGR_OCB_T
*ocb)
{
      iofunc_ocb_t *o = (iofunc_ocb_t *)ocb;
      char nowstr[128];
      time_t t;
      int n;
```

```
        /* 取当前时间，写入字符串 */
        t = time(NULL);
        n = strftime(nowstr, 128, "%Y-%m-%d %H:%M:%S\n", localtime(&t));

        /* 把字符串返回给客户端 */
        MsgReply(ctp->rcvid, n, nowstr, n);

        /* 告诉 resmgr 层，我们已经做过 Reply 了，不要再 reply 客户端了 */
        return _RESMGR_NOREPLY;
}
```

然后在 resmgr 初始化的时候，把这个回调函数加上：

```
iofunc_attr_init(&io_attr, 0666 | S_IFCHR, 0, 0);
iofunc_func_init( _RESMGR_CONNECT_NFUNCS, &connect_funcs,
                          _RESMGR_IO_NFUNCS, &io_funcs );

io_funcs.read = now_read;
```

现在，再执行 devnow 程序启动 /dev/now 服务。

```
$ cat /dev/now
2008-09-12 10:23:06
2008-09-12 10:23:06
2008-09-12 10:23:06
2008-09-12 10:23:06
2008-09-12 10:23:06
2008-09-12 10:23:06
…
```

虽然我们正确得到了时间显示，但是为什么会持续不断地显示下去？这是因为 cat 的工作原理就是循环 read() 直到读到文件结尾。所以我们要怎么样跟用户程序说 " 已经读到文件结尾了 " ？ 其实就是 read()函数返回个长度 0。也就是说我们的 io_read()回调函数里，需要区分第一次 read（返回时间字符串）和第二次 read() （返回长度 0）。我们可以通过记录文件读取的 offset 来区分。第一次读时，offset 还是 0，但接下来再读的话，offset 就不是 0 了。

看下面修改的代码。

```
static int now_read(resmgr_context_t *ctp, io_read_t *msg, RESMGR_OCB_T
*ocb)
{
        iofunc_ocb_t *o = (iofunc_ocb_t *)ocb;
        char nowstr[128];
        time_t t;
        int n;

        /* 判断是不是文件 open 后第一次 read()，不是的话，回复 0，这样客户端的
        read()就会返回 0，以示读到了文件尾 */
        if (o->offset != 0) {
                return 0;
        }
```

```
        /* 取当前时间，写入字符串 */
        t = time(NULL);
        n = strftime(nowstr, 128, "%Y-%m-%d %H:%M:%S\n", localtime(&t));

        o->offset += n;

        /* 把字符串返回给客户端 */
        MsgReply(ctp->rcvid, n, nowstr, n);

        /* 告诉 resmgr 层，我们已经做过 Reply 了，不要再 reply 客户端了 */
        return _RESMGR_NOREPLY;
}
```

现在，再 cat 的话：

```
$ cat /dev/now
2008-09-12 10:25:28
$
```

devnow 的完整代码在 Github 的 devnow 下面可以找到。

iofunc_default_ 回调函数

如果资源管理器，并不需要呈现标准的 POSIX 文件属性，那么如上面的例子，事情还比较简单。但如果资源管理器需要支持完整的 POSIX 文件，事情就比较复杂了。

POSIX 对于文件处理，有一些比较重要的内在联系。比如在 io_open() 回调中，不光要检查客户端有没有正确的读写权限，也要记住打开文件的模式，并反应在将来的回调中（显然一个 O_RDONLY 打开文件的客户端是不能 io_write()的）；文件管理器还得不断追踪读写的位置，以保证将来的 lseek() / tell() 调用能返回正确的值；另外每一次 read() / write() 都需要更新 stat 结构里的 st_mtime; st_atime 等等…

为了帮助用户正确处理 POSIX 文件，QNX 在 iofunc 层还提供了 iofunc_default_* 函数：

```
iofunc_open_default()
iofunc_chmod_default()
iofunc_devctl_default()
…
```

还是资源管理器的例子，我们加几行代码，注册一个 /posix_file 的文件：

```
dispatch = dispatch_create();

memset( &res_attr, 0, sizeof( res_attr ) );
res_attr.nparts_max = 10;
res_attr.msg_max_size = 0;

iofunc_attr_init(&io_attr, 0666 | S_IFCHR, 0, 0);
iofunc_func_init( _RESMGR_CONNECT_NFUNCS, &connect_funcs,
                  _RESMGR_IO_NFUNCS, &io_funcs );
```

```
connect_funcs.open  = iofunc_open_default;
io_funcs.read           = iofunc_read_default;
io_funcs.write          = iofunc_write_default;
io_funcs.chmod      = iofunc_chmod_default;
io_funcs.chown      = iofunc_chown_default;
io_funcs.stat           = iofunc_stat_default;
io_funcs.close_ocb  = iofunc_close_ocb_default;

rmgid = resmgr_attach(dispatch, &res_attr, "/posix_file", _FTYPE_ANY,
                      0, &connect_funcs, &io_funcs, &io_attr);

ctp = resmgr_context_alloc(dispatch);
while (1)
{
      ctp = resmgr_block(ctp);
      resmgr_handler(ctp);
}
```

具体代码在 Github 上的 posix_file 下面。这个资源管理器我们可以做一些操作：

```
# ls -lc /posix_file
crw-rw-rw-  1 root      root         0,   1 Jun 05 08:34 /posix_file
```

可以看到文件的 owner 是 root，mode 是 666，文件的 update time 是 08:34

```
# chown xtang /posix_file
# chmod 777 /posix_file
# echo "hello" >/posix_file
# ls -lc /posix_file
crwxrwxrwx  1 xtang     root         0,   1 Jun 05 08:35 /posix_file
```

可以看到，文件的 mode, owner, update time 都正确地变化了。

这个示例代码，我们是直接挂接了 iofunc_ chmod_default, iofunc_chown_default 等函数，虽然可以处理标准 POSIX 文件命令，但实际上没有什么功能。在实际情况下，一般都是挂自己的函数进行资源管理器的处理，然后调用 iofunc_*_default 函数来进行相关 POSIX 的处理。就是说:

```
iofuncs.write = my_write;
int my_write(resmgr_context_t *ctp, io_wriet_t msg, iofunc_ocb_t *ocb)
{
      /* do special operation */

      return iofunc_write_default(ctp, msg, ocb);
}
```

现在，我们来试试用 iofunc + resmgr 来重写我们的 md5 服务器。按照需要，我们只要挂 io_write() 和 io_read() 两个回调函数。客户端 write()时，不断把“写进来”的数据输入 MD5_Update()，一直到客户端 read()，把到目前为止的 digest 算出来，返回给用户。

这里涉及到一个问题。MD5 记算，是有一个 MD5_Context_t 的数据结构的，用 MD5_Init() 初始化后，对于数据，需要不断用 MD5_Update() 去计算；最后 read() 时，你会得到用 MD5_Final 返回的 digest.

问题来了，这些 io_read()/io_write() 函数是回调函数，怎么保证陆续进来的几个 write()和 read()处理中，这个 Context 结构是贯穿始终的?

答案是扩展 OCB 。

扩展 **OCB**

OCB 是 Open Context Block. 当文件被 Open 时，一个 ocb 会准备好。这个 ocb 等于是绑定了 fd，只要文件没有 CLOSE，在同一 fd 上的任何 iofunc 回调，都会回传这个 ocb。如果同时有几个 fd 被 open 了以后，每一个 fd，都有一个独自的 ocb。显然，如果我们的 md5_context_t 可以嵌入在 ocb 里的话，我们就会每个 fd 都有一个 MD5_context_t 了。具体怎么做呢?

首先定义一个扩展的 ocb 结构，如下。

```
/* extend iofunc_ocb_t */
typedef struct {
      iofunc_ocb_t ocb;
      int          total_len;
      MD5_CTX      md5_ctx;
} md5mgr_ocb_t;
```

请注意，结构里第一个元素一定是 iofunc_ocb_t，这样保证我们的扩展结构，可以直接当作 iofunc_ocb_t 操作。如果需要，我们依然可以调用 iofunc_*_default 函数来做 POSIX 相关的默认处理。这个结构后面的 "total_len"和“md5_ctx" 就是扩展的部份了。

接下来，这个扩展的结构，要怎样分配内存呢？有两种办法，一种是替换 iofunc 层的 ocb_calloc() 回调函数。

```
iofunc_funcs_t ocb_funcs = {
      _IOFUNC_NFUNCS,
      md5mgr_ocb_calloc,
      md5mgr_ocb_free
 };
iofunc_mount_t mountpoint = { 0, 0, 0, 0, &ocb_funcs };
iofunc_attr_init(&io_attr, 0666 | S_IFCHR, 0, 0);
io_attr.mount = &mountpoint;
```

另一种更直接的办法，就是自己准备一个 io_open()的回调，当客户端 open()时，就会进入我们的回调，在回调里，可以自己 malloc()自己的结构，然后用 resmgr_bind()把这结构当作 ocb 跟 fd 绑定。这样，接下来的所有 io 回调中，都会有这个 ocb 结构传进来。

这个新 md5 服务完整代码在 Github 的 md5mgr 下面。

dispatch 层

虽然大多数情况下，用 iofunc 层+resmgr 层已经可以构建一个完整的资源管理器了，但是有时候资源管理器需要同时处理别的消息，这时就需要 dispatch 层了。dispatch 可以处理其他形态的信息，除了可以用 resmgr_attch() 来挂接 resmgr 和 iofunc 以外，还可以做这些 (sys/dispatch.h)：

message_attach()

```
int message_attach(dispatch_t *dpp, message_attr_t *attr, int low, int high,
                   int (*func)(message_context_t *ctp, int code, unsigned flags, void *handle),
                   void *handle);
```

有时候资源管理器在使用标准的 iomsg 以外，还想要定义自己的消息类型，那就会用到这个 message_attach(). 这个函数意思是说，如果收到一个消息，它的消息类型在 low 和 high 之间的话，那就回调 func 来处理。

当然，要保证 low/high 不会与 iomsg 重叠，不然就会有歧义。在 iomsg.h 里已经定义了所有的 iomsg 在 _IO_BASE 和 _IO_MAX 之间，所以只要保证 low > _IO_MAX 就可以了。

pulse_attach()

```
int pulse_attach(dispatch_t *dpp, int flags, int code,
                 int (*func)(message_context_t *ctp, int code, unsigned flags, void *handle),
                 void *handle);
```

有许多管理硬件资源的管理器（驱动程序），除了提供 iomsg 消息，来对应客户端的 io 请求以外，也会很常见需要接收“脉冲”来处理中断 (InterruptAttachEvent)。这时，用 pulse_attach() 就可以把指定的脉冲号 (code)，绑定到回调函数 func 上。

select_attach()

```
int select_attach(void *dpp, select_attr_t *attr, int fd, unsigned flags,
                  int (*func)(select_context_t *ctp, int fd, unsigned flags, void *handle),
                  void *handle);
```

很多资源管理器，在处理客户端请求时，还需要向别的资源管理器发送一些请求。而很多时候这些请求可能不能立即返回结果，通常情况下，可以用 select() 来处理，但第一我们无法使用会阻塞的 select()，因为我们是一个资源管理器的服务函数，如果被阻塞无法返回，就意味着我们无法处理客户端请求了；第二我们也无法用 select() 轮询，因为我们一旦返回，就会进入 等待客户端消息的阻塞状态，没有新消息来时，不会退出阻塞状态，也就没有机会再去轮询 select()了。

当然，你可以自设一个时钟，每隔一定时间就给自己发一个脉冲，等于自己把自己叫醒，然后再轮询 select()。做是做得到，但这样就无端增加了许多系统开销。

select_attach() 就是为了这个目的设的，针对一个特定的 fd, 这里的 unsigned flags，决定了你想要 select() 的事件（Read? Write? Except?)。这意思是说，如果对于 fd, 我选择的 flags 事件发生了的话，调用 func 回调函数。

在使用 dispatch 时，进行特殊的 *_attach() 挂接以后，只要把 resmgr 层的几个函数替换成 dispath 层的几个函数 就可以了，比如这样：

```
ctp = dispatch_context_alloc(dispatch);
while (1)
{
        ctp = dispatch_block(ctp);
        dispatch_handler(ctp);
}
```

dispath_block() 相当于阻塞并等待，而 dispatch_handle() 则根据不同的挂接，调用不同的回调函数进行处理。

另一个比较常用的 dispatch 函数是 dispatch_create_channel()。有时候，你希望自己用 ChannelCreate() 创建频道，而不是让 dispatch_create()自动为你创建频道，就可以用这个函数。之所以需要自己创建频道，是因为有时候希望在频道上设一些特殊的标志（ChannelCreate() 的 flags)。

thread pool 层

上面这些用 while (1) 来循环处理消息的资源管理器，明显都是“单线程”资源管理器，一共只有一个线程来处理客户端请求。

对于一些需要频繁处理客户请求的资源管理器，自然会想到用“多线程”资源管理器。基本就是用几个线程来执行上面的 while(1)循环。

线程池（Thread Pool）就是用来实现这个的。使用起来也比较简单，先配置 poot_attr，然后创建并启动线程池。

```
memset(&pool_attr, 0x00, sizeof pool_attr);
if(!(pool_attr.handle = dpp = dispatch_create())) {
      perror("dispatch_create");
      return EXIT_FAILURE;
}
pool_attr.context_alloc = dispatch_context_alloc;
pool_attr.block_func    = dispatch_block;
pool_attr.handler_func  = dispatch_handler;
pool_attr.context_free  = dispatch_context_free;
pool_attr.lo_water      = 2;
pool_attr.hi_water      = 5;
pool_attr.increment     = 2;
pool_attr.maximum       = 10;

tpp = thread_pool_create( &pool_attr, POOL_FLAG_EXIT_SELF)
thread_pool_start( tpp );
```

pool_attr 前面几个回调函数都比较简单，后面的 lo_water, hi_water, increment, maximum 简要说明一下。

maxmimum 是最多池里可以建多少线程，

hi_water 是最多这些线程可以等待任务

lo_water 是至少应该有多少线程需要在等待任务

increment 则是一次递增的线程数。

拿上面的例子来说，这个线程池一旦启动，首先会创建 5 个线程，都在 dispatch_block()上等待接受任务。

当有一个请求来时，1 号线程为其服务，假设这个服务线程在服务过程中，还需要向别的线程请求数据，一时回不来，那就还剩下 4 个线程等待任务。

如些再来两个请求，我们会变成 3 个线程在进行服务，2 个线程等待任务的情形。

这时，当第 4 个请求来时，又一个线程去进行服务，这时只有 1 个线程在等待任务了，比 lo_water 少，所以线程池会自动再新建 2 个（increment) 线程，把他们放在等待任务队列中。所以这时我们有 4 个线程在服务，3 个线程在等待。

如果服务线程无法回收，而新的请求又进来，导致等待线程数又低于 lo_water 的话，那么，线层池还会继续增加线程以保证有足够线程在等待服务，但是服务线程数与等待线程数的和，不会超过 10（maximum).

假如在 4 个线程服务，3 个线程等待的状态下，有 2 个服务线程结束了服务，它们会被还回线程池，线程池就会把这 2 个线程继续放入等待队列。也就是变有还有 2 个服务线程，5 个等待线程的状态。

现在，如果又有 1 个线程结束服务，回归线程池了。如果把这个线程再放入等待队列，那就会有 6 个线程等待服务，这个超过了 hi_water，所以线程池会结束这个线程。这样，我们就会有 1 个线程在服务，5 个线程在等待的情形，线程总数下降到了 6

综上，使用线程池可以动态地灵活配置多线程资源管理器，这样，当大量服务同时拥来时，线程分分钟投入服务；当空闲线程较少时，线程池会预先再多开些线程，以备服务。然后，当线程服务结束后，线程池也会停掉一些多余的线程。

结语

大家可以看到资源管理器在 QNX 上的重要，几乎所有的服务都是通过资源管理器来实现的。而且用资源管理器的概念，可以很好地模块化系统。所以正确地理解资源管理器的概念，熟练运用 QNX 提供的资源管理器，是在 QNX 上进行开发的重要技能。当然，资源管理器也有其自身的弱点，通常一个管理器需要跟别的管理器协同工作，才能完成系统的功能；这时，犹其需要注意单线程的管理器不要有被阻塞不能提供服务的时候，多线程管理器虽然不担心阻塞，但是更需要当心线程间同步。

另外，系统设计上一个很重要的因素，还在于正确设计资源管理的细分化。分得太细，会造成一个任务需要穿过多个资源管理器才能实现，会严重损失性能。分得太粗，当然就失去了模块分割，很容易变成一个复杂系统而增加了调试和出错管理的难度。