MOC (Music On Console)

1. 配置选项

选项的值有两种类型：int，str。由枚举定义如下：

```
enum option_type
{
    OPTION_INT,
    OPTION_STR,
    OPTION_ANY
};
```

值的表示则使用 union 来定义：

```
union option_value
{
    char *str;
    int num;
};
```

每个选项除了值，值的类型，还有名字等字段：

```
struct option
{
    char name[OPTION_NAME_MAX];
    enum option_type type;
    union option_value value;
    int ignore_in_config;
    int set_in_config;
};
```

其中，ignore_in_config 表示是否忽略配置文件里的此选项。因为有些选项可以通过命令行参数指定（比如 MOCDir），命令行参数上指定了的选项，其 ignore_in_config 就设为1，这样在装载配置文件时，就可以跳过了。

所有选项存在一个全局数组里，叫 options，大小为128。

配置文件的格式比较简单，解析配置文件的代码也就一个 while 循环，100来行代码搞定。

选项的名字没有定义成宏，导致同一个字面字符串在代码中多处出现，这一点不太好。

2. 错误处理

MOC 的错误主要分两种：内存分配，网络通信。

这两种错误都不太可能发生，因为 MOC 是个非常小巧高效的程序，极少的内存就可以运行，而网络通信采用的不是一般的 socket，而是高效稳定的 UNIX Domain Socket。

鉴于这种情况，一旦有错误发生，调用 exit()退出程序便是可取的。

内存分配相关的函数有 malloc，calloc，realloc，strdup 等，分别做了简单的封装：xmalloc，xcalloc，xrealloc，xstrdup，x~会检查~是否成功，失败了就调用 fatal()，fatal()向 stderr 打出

log，然后调用 exit()退出程序。

3. main()

函数 main()依次做如下事情：

- 初始化所有选项，用缺省值，比如 Shuffle = 0，MOCDir = "~/.moc"。
- 用 getopt 获取命令行参数。

  命令行参数先存在 parameters 结构中，随后会传给其他函数，要么启动 server 和/或 client，要么向 server 直接发送命令。

```
struct parameters
{
    int debug; // 对应于 -D, --debug
    int only_server; // 对应于 -S, --server
    int foreground; // 对应于 -F, --foreground
    // ...
};
```

  有些命令行参数直接存在选项里，比如 M（moc-dir）的值对应于选项 MOCDir 。而 C（config）指定了配置文件，存在局部变量里，随后即用。
- 检查命令行参数的合法性。比如 foreground（在前台运行 server）必须与 server（只运行 server）一起使用。
- 解析配置文件。

  配置文件可由命令行参数 C（config）指定，如果命令行参数没有指定（多数情况如此），就用 MOCDir 目录里的配置文件（也可能尚不存在）。
- 检查 MOCDir 是否存在，不存在则创建之。
- 最后，如果命令行参数指定了如下几个命令之一，则向 server 发送相应的命令，然后退出程序。否则，如果必要，启动 server 或 client：
  - s (stop)
  - f (next)
  - r (previous)
  - x (exit)
  - P (pause)
  - U (unpause)
  - G (toggle-pause)

4. Client / Server

MOC 使用 client/server 结构，因为 client 和 server 在同一台主机上，所以 MOC 使用的不是一般的 socket，而是效率比较高的 UNIX Domain Socket。UNIX Domain Socket 用于 IPC（进程间通信）在效率上的优点是：不需要经过网络协议栈，不需要打包拆包、计算校验和、维护序号和应答等，只是将应用层数据从一个进程拷贝到另一个进程。

关于 UNIX Domain Socket IPC：http://learn.akae.cn/media/ch37s04.html。

4.1. 协 议

之前在 start_moc()里提到 server 的启动分两步：

- server_init()，返回 server 的 socket fd，此 fd 将保存下来，传给界面等留作后用。
- server_loop()

不管是在后台还是前台启动，都是如此。

Server 与 client 之间要通过 socket 通信，需要定义一组协议，来规定 client 可以向 server 发哪些命令，server 又可以向 client 发哪些事件。

从 client 到 server 的命令

Client 每一个可能的操作，都对应于一个命令，比如播放对应于 CMD_PLAY（0x00），其他命令有：

- CMD_STOP：停止播放
- CMD_PAUSE：暂停
- CMD_NEXT：播放下一首
- CMD_GET_BITRATE：获取比特率
- CMD_PING：Ping server
- CMD_DISCONNET：与 server 断开
- 等等

从 server 到 client 的事件

Server 发生了什么，有时需要通知 client(s)，最简单的情况比如 server 退出了（EV_EXIT），client 自然有必要知道。这些事件定义如下：

- EV_STATE：server 改变了状态（server 的状态有三种：PLAY，STOP，PAUSE）
- EV_CTIME：歌曲的当前时间已改
- EV_BUSY：另一个 client 正在连接 server
- EV_PONG：命令 CMD_PING 的应答
- 等等

4.2. Server/Client 间数据的打包、发送

```
// main.c
static void server_command (struct parameters *params)
```

此函数向 server 发送参数 params 所指定的请求（暂停播放，退出程序，等）。params 里的内容来自命令行参数。这个函数在 main 函数的末尾调用，当即不需要启动 server 也不需要启动 client 时，而只是想向 server 发送命令时。

```
// main.c
static int ping_server (int sock)
```

此函数向 server 发一个 CMD_PING 命令，如果 server 返回一个 EV_PONG 事件（调用 recv 从 server 的 socket 接受一个 int，CMD_PING 和 EV_PONG 等都是 int 值），表示 server 可用。

```
// main.c
static int server_connect ()
```

Server 启动时，依次调用 socket()，bind()，listen()完成 socket server 的创建。Server 调用 bind()时绑定的地址，和 client 调用 connect()时连接的地址相同。

```
// protocol.c
int send_int (int sock, int i)
int get_int (int sock, int *i)
```

函数 send_int()调用 send()向给定的 socket 发送一个整型值。函数 get_int()调用 recv()从给定的 socket 接收一个整型值。

```
// protocol.c
int send_str (int sock, const char *str)
char *get_str (int sock)
```

函数 send_str()向给定的 socket 发送一个字符串。首先调用 send_int()发送字符串的长度，然后调用 send 发送字符串。

函数 get_str()从给定的 socket 接收一个字符串。首先调用 get_int()获得字符串的长度，然后分配相应大小的内存，再连续调用 recv 获取完整的字符串。使用时要注意释放内存。

```
// protocol.c
int send_time (int sock, time_t i)
int get_time (int sock, time_t *i)
```

函数 send_time()调用 send()向给定的 socket 发送一个时间值。函数 get_time()调用 recv()从给定的 socket 接收一个时间值。

4.3. interface_loop()

```
// interface.c
void interface_loop ()
{
    while (want_quit == NO_QUIT) {
        // fds 和 ret 的声明可以放在 while 外面。因为 select() 会改变 timeout，
        // 所以如果把 timeout 也声明在外面的话，记得每次循环时重新赋值。
        fd_set fds;
        int ret;
        struct timeval timeout = { 1, 0 };

        FD_ZERO (&fds);
        FD_SET (srv_sock, &fds); // 把 server 的 socket fd 加到 fd set 里。
        FD_SET (STDIN_FILENO, &fds); // 把 stdin 的 fd(0) 加到 fd set 里（以处理键盘输入）。

        dequeue_events ();
        // 监视 fds 里的各 fd，直到它们（一或多个）可读（ready to read）。
        ret = select (srv_sock + 1, &fds, NULL, NULL, &timeout);

        iface_tick ();

        if (ret == 0) // timeout
            do_silent_seek ();
        else if (ret == -1 && !want_quit && errno != EINTR) // select 失败!
            interface_fatal ("select() failed: %s",
                    strerror(errno));
```

```
#ifdef SIGWINCH
        if (want_resize)
            do_resize ();
#endif

        if (ret > 0) {
            if (FD_ISSET(STDIN_FILENO, &fds)) { // stdin 可读，处理键盘输入。
                struct iface_key k;

                iface_get_key (&k);

                clear_interrupt ();
                menu_key (&k);
            }

            if (!want_quit) {
                if (FD_ISSET(srv_sock, &fds))
                    get_and_handle_event (); // 从 server 获取并处理事件。
                do_silent_seek ();
            }
        }
        else if (user_wants_interrupt()) // CTRL-C was pressed?
            handle_interrupt ();

        if (!want_quit)
            update_mixer_value ();
    }
}
```

4.4. server_loop()

```
// server.c
// 参数 list_sock 为 server 的 socket fd，由 server_init() 返回。
void server_loop (int list_sock)
{
    struct sockaddr_un client_name;
    socklen_t name_len = sizeof (client_name);
    int end = 0;

    do {
        int res;
        fd_set fds_write, fds_read;

        FD_ZERO (&fds_read);
        FD_ZERO (&fds_write);

        // list_sock 为 server 的 socket fd；可读表示有新连接。详见 accept 的手册。
        FD_SET (list_sock, &fds_read);

        // wake_up_pipe 是为了在另一个线程（thread）里把 server 从 select 调用中唤醒。
```

```
        // 详解稍后。
        FD_SET (wake_up_pipe[0], &fds_read);

        // 将现有的各 client 也加到 read/write 的 fd set 里。
        // 这样，当它们可读时，便能从其接收命令；当它们可写时，便能向其发送事件。
        add_clients_fds (&fds_read, &fds_write);

        if (!server_quit)
            res = select (max_fd(list_sock)+1, &fds_read,
                  &fds_write, NULL, NULL);
        else
            res = 0;

        if (res == -1 && errno != EINTR && !server_quit) {
            // select 失败，输出日志并退出。
        }
        else if (!server_quit && res >= 0) {
            if (FD_ISSET(list_sock, &fds_read)) { // 有连接请求。
                int client_sock;
                // 接收连接请求并创建 socket，返回新建 socket 的 fd。
                client_sock = accept (list_sock,
                       (struct sockaddr *)&client_name,
                       &name_len);

                // accept 失败则退出。代码从略。

                // 将 socket 保存到 client 列表里。目前可最多有10个 client，所以如果超过10个，
                // 就会失败，这时候调用 busy() 发送 EV_BUSY 事件给这个 client，然后关闭连接。
                if (!add_client(client_sock))
                    busy (client_sock);
            }

// 现在来说说 wake_up_pipe 唤醒 server 的作用。
// TODO：感觉并不十分必要，因为 client 可写，自然 select 成功，进而向其发送事件队列中的事件。
// 假如某 client 向 server 发送 CMD_PAUSE 命令，server 于是从 select 中返回，通过下面的
// handle_clients 接收并处理这个命令。处理 CMD_PAUSE 命令除了要设置当前状态为 STATE_PAUSE
// 外，还需要向所有 client 发送 EV_STATE 事件通知它们状态已经改变。这个通知不是在这一次 select
// 中完成的，而是通过往 wake_up_pipe[1] 里写1，继而触发下一次 select 来完成的，即函数
// send_events。
            if (FD_ISSET(wake_up_pipe[0], &fds_read)) {
                // 象征性地读 wake_up_pipe[0]，从略。
            }

            // 对每一个 client，如果可写，则将其事件队列中的事件发送过去。
            send_events (&fds_write);
            // 对每一个 client，如果可读，则接收并处理来自于其的命令。
            handle_clients (&fds_read);
        }
    } while (!end && !server_quit);
```

```
    // 关闭所有 client，关闭 server 的 socket，关闭 server，等。代码从略。
}
```

5. Audio, Player and IO

5.1. 音频相关知识补充

5.1.1. ALSA, OSS, JACK, etc.

https://bbs.archlinux.org/viewtopic.php?id=48385 (freakcode)

I see a lot of developers saying good things about OSS API - in fact, OSS API is so better that some of them use ALSA OSS emulation instead of ALSA own API, which is regarded as poorly documented and messy.

PulseAudio promisses a lot, but delivers little. There's a pile of Fedora and Ubuntu users reporting PulseAudio broken compatibility with everything else, that it make sound lags, and that uses too much CPU. Also, PulseAudio comes to fix a broken level below, that is ALSA's lack for native mixing and per-application volume levels.

OSS, on the other hand, is a good api, UNIX compatible (instead of ALSA that is Linux specific), with native mixing support and per-application volume levels. If you ever used FreeBSD, you may also know it's ridiculous simple to setup - if your soundcard is supported, you only load the driver, it just works. Finally, OSS is the main choice for third-party and commercial applications (like Skype, TeamSpeak, games like Quake 4, etc.), as they win UNIX compatibility (instead of only supporting Linux), and a better, more documented API to use.

Jack (which is awesome, btw) doesn't enter on the discussion, as I see it aiming more features for professional software (like low latency, input/output redirection), that most people won't use on daily basis.

Also, not to confuse ALSA and OSS (sound APIs) with PulseAudio and Jack (sound servers).

5.2. CURL

MOC 可以播放 URL 指定的文件，正是通过 libcurl 来实现的。

详见：http://curl.haxx.se/

5.3. 播放流程

考虑最简单的情形：用户选中播放列表中的某个文件，然后按回车开始播放。

基本的流程是这样的：

- interface 分别调用 send_int_to_srv()和 send_str_to_srv()向 server 发送 CMD_PLAY 命令和文件名。
- Server 在 server_loop()里调用 handle_command()时接收到 CMD_PLAY 命令和文件名，然后调用 audio_play()来播放这个文件。

- audio_play()创建一个播放线程，线程函数是 play_thread()。play_thread()从当前文件开始播放，按照一定的顺序（视 Shuffle 选项而定），播放整个播放列表。
- 具体来说，play_thread()是通过调用 player()来播放文件的。player()的签名如下，它的任务是打开当前文件（参数 file），将之解码，然后把解码的输出放到给定的 buffer 里（参数 out_buf），最后，它会提前 cache 下一个播放文件（参数 next_file）。

```
void player (const char *file, const char *next_file, struct out_buf *out_buf)
```

5.4. Decoder Plugins

上面提到 player()的主要任务是解码文件，那么解码某一类文件时，就需要相应的解码器，比如 mp3 文件需要 mp3 解码器。

MOC 的解码器以共享库的形式，通过插件的方式进行管理。MOC 在启动时会装载所有的解码器：

```
// main.c
static void start_moc (const struct parameters *params, char **args,
       const int arg_num)
{
   decoder_init (params->debug);
   ...
```

decoder_init()遍历插件目录里的每一个共享库文件，打开并进行初始化，用下面的 struct plugin 表示每一个插件：

```
static struct plugin {
   lt_dlhandle handle;
   struct decoder *decoder;
} plugins[8];
```

因为 struct plugin 数组大小为8，所以最多只能装载8个插件。

MOC 在操作共享库时没有直接使用 dlopen()，dlsym()等函数，而是使用了 libltdl 这个程序库。libltdl 的好处是可以跨平台，甚至还能支持 Windows 的 LoadLibrary()。