DirectFB 源码解读

DirectFB 就是一个全能系统，麻雀虽小五脏俱全。DirecttFB 源码，可以了解很多方面的技术，包括 Framebuffer, Graphics Accelerate Card,鼠标及键盘等外设的事件处理，Font, Graphics Drawing 等，例外还可以看到一下很有用的编程技巧，例如 C++思想在 C 语言中的运用，动态加载链接库，双 buffer 的具体实现，进程通信，多进程的控制和管理等。

DirectFB 源码解读之初始化-1

对于任何一个 DirectFB 程序，头两句有关 DirectFB 的函数调用一定是：
DirectFBInit(&argc, &argv);
DirectFBCreate(&dfb);

它的作用是初始化整个 DirectFB 库，为后面的画图等实际操作做好准备，那 DirectFB 是如何完成初始化的呢？它的初始工作包括那些呢？现在我们就从这两个函数开始，进入 DirectFB 源代码。

（一）
DirectFBInit（）定义在【src\Directfb.c】中，主要是读取系统的和用户定义的 configuration、处理命令行，读取环境变量等，我们暂且跳过这个函数，但是由于它的重要性，我们会在以后专为此开一个专题。

DirectFBCreate（）也定义在【src\Directfb.c】中，从用户的角度看，这个函数的作用就是返回一个类型为 IDirectFB 的指针，而这个指针是在后续 API 调用中必须的，例如 CreateSurface() ，CreateFont()等。从 DirectFB 本省来说，这个函数完成一系列的初始化，并将各种需要的信息保存在 IDirectFB 指向的数据结构中。它主要会调用三个函数：

- direct_initialize();//线程以及信号量的初始化
- dfb_core_create();//完成 directfb 的各个子系统的初始化
- IDirectFB_Construct();//为 DirectFB 的根接口设置函数指针。

dfb_core_create() 主要完成 directfb 各个子系统的初始化，完事以后，所有的信息都会保存在一个 CoreDFB 的数据结构中，它是 IDirectFB 的一部分。

另外，DirectFB 是支持多进程的，由于其中涉及的内容多而杂，因此我们先研究单进程的情况，所有涉及的对线程的代码我们都先忽略，后面会单独开辟专题研究多进程是如何工作的。

（二）
现在我们就看看 dfb_core_create（）是如何工作的：
它依次调用下面的几个函数：

- direct_initialize()；//初始化线程锁及注册信号处理函数
- dfb_system_lookup();//在指定的目录中，搜寻存在的系统实现
- direct_find_best_memcpy();//如果有必要，DirectFB 将使用用户指定的 memory-copy 函数（例如为特定平台优化的 memcpy）
- fusion_enter(); //分配 Main pool
- fusion_arena_enter();//调用 dfb_core_arena_initialize，完成各个子系统的初始化。

我们现在逐一分析后四个函数。

DirectFB 源码解读之初始化-2

dfb_system_lookup( ) [src\core\System.c]是搜寻是否存在 system 的实现，在 DirectFB 中，system 概念特指 graphics system， 不同的系统意味着显示输出设备和接口是不同的。例如：

- fbdev: 输出到 frame buffer。
- osx: 输出到 mac os 上。
- vnc： 输出到 Virtual Network Computing（类似于微软远程桌面的一个协议）
- x11： 输出到 X Window 上。
- sdl： 输出到 Simple DirectMedia Layer。

为了统一各个 graphics system 之间的差异，DirectFB 定义了一套统一的接口，也就是说各个 graphics system 必须实现这些接口，才能使 DirectFB 运行在其上。这些接口实际上就是一些 API, 具体定义在【src\core\Core_system.h】:

```
static CoreSystemFuncs system_funcs = {
    .GetSystemInfo      = system_get_info,
    .Initialize         = system_initialize,
    .Join               = system_join,
    .Shutdown           = system_shutdown,
    .Leave              = system_leave,
```

```
    .Suspend             = system_suspend,
    .Resume              = system_resume,
    .GetModes            = system_get_modes,
    .GetCurrentMode      = system_get_current_mode,
    .ThreadInit          = system_thread_init,
    .InputFilter         = system_input_filter,
    .MapMMIO             = system_map_mmio,
    .UnmapMMIO           = system_unmap_mmio,
    .GetAccelerator      = system_get_accelerator,
    .VideoMemoryPhysical = system_video_memory_physical,
    .VideoMemoryVirtual  = system_video_memory_virtual,
    .VideoRamLength      = system_videoram_length,
    .AuxMemoryPhysical   = system_aux_memory_physical,
    .AuxMemoryVirtual    = system_aux_memory_virtual,
    .AuxRamLength        = system_auxram_length,
    .GetBusID            = system_get_busid,
    .GetDeviceID         = system_get_deviceid
};
```

上面这些函数包括：系统初始化，系统 memory map，系统退出等。注意这个结构是在定义的同时复制的，用的是圆点， 可以对一部分成员变量赋值。

DirectFB 本身自带了 fbDev、x11 等多个 graphics system 的实现，每个实现就是一个动态链接库。
这些动态链接库很可能同时存在的，只有在运行时才会根据实际情况自动加载（用户可以在 config file 中指定， system=<system>，或者缺省的就是 fbDev）。在运行时，这些动态链接库必须放在特定的目录中，例如 directfb/lib/directfb/systems，该目录在编译的时候指定的。

动态链接库的名字有一定的约定， 以 libdirectfb_开头，以.so 结尾的，例如 libdirectfb_devmem.so，libdirectfb_fbdev.so 等。

dfb_system_lookup() 调用的第一个函数是 direct_modules_explore_directory()，这个函数我们还会经常接触，它的作用就是在指定的目录中，搜索 modules（一般是动态链接库的形式存在），打开这是 module，调用 module 中初始化函数完成初始化工作，最后把所有满足条件的 module 加入到一个链表中。

在看 direct_modules_explore_directory()代码之前，我们先看看它的输入参数 dfb_core_systems，这是一个全局变量，它是连接 DirectFB 与各个不同的 graphics system 具体实现之间的桥梁。它的定义在 DEFINE_MODULE_DIRECTORY 中，展开这个宏定义可见，dfb_core_systems 实际上是 DirectModuleDir 的结构，

```
struct __D_DirectModuleDir {
    pthread_mutex_t     lock;
    const char         *path;
    unsigned int        abi_version;
    DirectLink         *entries;
    DirectModuleEntry *loading;
};
```

其中:

path 是 moduel 所在的路径

entries 就是包含所有 graphics system 实现的链表。

那么这种连接是如何建立起来的呢？

首先 dfb_core_systems 是以宏的形式定义在 directfb core 中的【src\core\System.c】， 而在<core/system.h>中 dfb_core_systems 定义为一个外部变量【extern】，这个头文件实际为各个 graphics system 实现所包含<core/system.h>。这样 core 中定义的变量就可以为各个动态库所使用。下一章，我们就具体分析 direct_modules_explore_directory()的代码。

DirectFB 源码解读之初始化-3

现在我们开始阅读 direct_modules_explore_directory（）【lib\direct\Modules. C】.

```
Int direct_modules_explore_directory ( DirectModuleDir *directory )
{
#ifdef DYNAMIC_LINKING
#一般情况下，driver 都是以动态链接库的形式存在的
    int            dir_len;
    DIR            *dir;
    struct dirent *entry = NULL;
```

```
     struct dirent   tmp;
     int             count = 0;

     D_ASSERT( directory != NULL );
     D_ASSERT( directory->path != NULL );

     D_DEBUG_AT( Direct_Modules, "%s( '%s' )\n", __FUNCTION__, directory->path );
     dir_len = strlen( directory->path );
#打开目录，path  就是上面宏定义中设定的
     dir     = opendir( directory->path );
     if (!dir) {
          D_DEBUG_AT( Direct_Modules, "   -> ERROR opening directory: %s!\n", strerror(errno) );
          return 0;
     }
     while (readdir_r( dir, &tmp, &entry ) == 0 && entry)
{
#读取目录下的每一个文件
#opendir, readdir_r 等都是 c 语言中操作目录的函数

          void              *handle;
          DirectModuleEntry *module;
          int                entry_len = strlen(entry->d_name);
#只处理后缀是 so 的文件，即动态连接库
          if (entry_len < 4 ||
              entry->d_name[entry_len-1] != 'o' ||
              entry->d_name[entry_len-2] != 's')
               continue;
#查看一下这个文件是否已经在链表中了，即这个 driver 是否已经遍历过了，如果是，则跳过
          if (lookup_by_file( directory, entry->d_name ))
               continue;

#为新的 module 分配一个 DirectModuleEntry  的数据结构，这个结构将来要加入的到链表中
          module = D_CALLOC( 1, sizeof(DirectModuleEntry) );
          if (!module)
               continue;

          module->directory = directory;
          module->dynamic    = true;
          module->file       = D_STRDUP( entry->d_name );

          directory->loading = module;

#打开这个动态链接库，实际调用的是 dlopen。
#需要注意的是这时候这 module 并没有加入到 list 中，只是通过 directory->loading 暂时记录起来。
          if ((handle = open_module( module )) != NULL) {
#如果一切顺利，这个 module 应该已经注册成功并加入到 directory 的链表中，而 module->loaded  也设为了 true，你可能纳闷，因为 open_module  中
并没有这些代码呀，别急，我们后面会讲
               if (!module->loaded)
               {
                    int     len;
                    void (*func)();
                   #如果进入了这里就意味的 module 自己注册没有成功，这是需要手动注册。
                    D_ERROR( "Direct/Modules: Module '%s' did not register itself after loading! "
                             "Trying default module constructor...\n", entry->d_name );

                    len = strlen( entry->d_name );
                  #去掉后缀“.so”
                    entry->d_name[len-3] = 0;
                      #掉前缀”lib”,例如“libdirectfb_fbdev.so”就变成了”directfb_fbdev”，

                    func = dlsym( handle, entry->d_name + 3 );
                     #在动态库中搜索函数 directfb_fbdev
                    #可是所有的 Fbdev 源码中好像并没有 directfb_fbdev 这个函数的定义，这是怎么回事呢？
                     #原来在  libdirectfb_fbdev.so  的源码  Fbdev.c  中有个宏定义  DFB_CORE_SYSTEM（fbdev）,展开这个 宏定义，一切都明白了。而
directfb_fbdev 实际调用的是 direct_modules_register，它的工作就是自动注册的过程，我们在此我详述，稍后会讲
                    if (func)
                     {
```

```
    ＃执行这个函数，
    func();
    ＃上述函数执行完毕后，loaded 应为 true，否则就意味着注册失败
     if (!module->loaded)
      {
        D_ERROR( "Direct/Modules: ... even did not register after "
                "explicitly calling the module constructor!\n" );
      }
  }
 else
{
    #func 为 NULL，即不存在构造函数，则什么也不做
        D_ERROR( "Direct/Modules: ... default contructor not found!\n" );
 }
if (!module->loaded)
{
   #如果手工注册也失败则仍然将该动态链接库加入到链表中，
  #只不过 disabled 设为 true，  表示不可用
    module->disabled = true;
     D_MAGIC_SET( module, DirectModuleEntry );
    direct_list_prepend( &directory->entries,
                         &module->link );
        }
    }

    if (module->disabled)
    {
         ＃对于状态是 disable 的 module，也就是自动注册和手动注册都没有成功的，则将其关闭
        dlclose( handle );
        module->loaded = false;
    }
    else
   {
       ＃对于注册成功的 module，  记录动态库的 handle
      ＃以后就可以根据这个 handle 调用库中的函数了，
      ＃注意的是这些动态库是处于打开状态，
        module->handle = handle;
        count++;
    }
}＃endif(handle=open_module(…))
else
{
  #如果 module 打开失败，则仍然将其加入链表中，disabled 设为 true，表示不可用
    module->disabled = true;
    D_MAGIC_SET( module, DirectModuleEntry );
    direct_list_prepend( &directory->entries, &module->link );
        }

        directory->loading = NULL;#当前没有试图加载的 module，为加载下一个 module 做准备
    }//endof while(readdir_r)

    closedir( dir );

    return count;
#else
    return 0;
#endif
}
```

DirectFB 源码解读之初始化-4

现在我们解决上一节中那个悬而未决的问题：dlopen()中能够到底做了些什么？
dlopen()是一个标准的 C 函数，在 dlopen()函数说明中有这样一段话：
“Instead, libraries should export routines using the __attribute__((constructor)) and __attribute__((destructor)) function attributes. See the gcc info pages for information on these. Constructor routines are executed before dlopen() returns, and destructor routines are executed before dlclose() returns.”

也就是在 dlopen()返回前，它将执行动态链接库中被__attribute__((constructor)) 修饰的函数。

现在再回头看一下 src\core\Core_system.h 的定义，确实有一个被__attribute__((constructor))修饰的函数:

```
#define DFB_CORE_SYSTEM(shortname)                          \
__attribute__((constructor)) void directfb_##shortname( void ); \
                                                            \
void                                                        \
directfb_##shortname( void )                                \
{                                                           \
     direct_modules_register( &dfb_core_systems,            \
                              DFB_CORE_SYSTEM_ABI_VERSION,  \
                              #shortname, &system_funcs );  \
}
```

不论具体的 system 是什么（也就是 shortname 是什么,每个 system 的实现中都会调用 DFB_CORE_SYSTEM 这个宏），其最终都会调用 direct_modules_register()。
direct_modules_register（）【lib\direct\Modules.c】才真正完成将当前 module 加入各个链表的工作，例如我们当前遍历的 system 链表，它就将每一个 system 实现挂在 dfb_core_systems 这个链表中，同时设置‘entry->loaded = true'表示该 module 已经加载成功。

至此 dfb_system_lookup（）中 direct_modules_explore_directory(&dfb_core_systems)全部完成。这时候实际上所有的 system 实现都是打开的同时挂在 dfb_core_systems 链表中，dfb_system_lookup（）根据 configure 中指定 system 确定最终的选用 system 是哪个（比较 module 的名字,上面标红的字段），然后将没用的 module 关闭。而选中的 sytem 及其函数表会记录在两个全局变量中：system_module（）和 system_funcs。

另外，它还会调用 system_funcs 函数表中的 GetSystemInfo(),得到该 graphics system 的一些基本信息,这些信息也记录在一个全局变量 system_info 中， 其实这些信息中我们最关心的就是其中的 caps，它记录了该系统是否支持硬件加速。

DirectFB 源码解读之初始化-5

继续 dfb_core_create()的源码之旅，接着调用的是 direct_find_best_memcpy（），它的作用就是寻找性能最优的 memory copy 的实现， 不同的平台可能有不同的 memcpy 的优化， 用户在编译 DFB 的时候可以指定某一种 memcpy。
DFB 本省包含了为部分平台优化过的 memory_copy【lib\direct\Memcpy.c】中，其中就有 memcpy()。如果用户没有指定某种 memory_copy 实现，那么 DFB 会比较当前的几个 memory_copy 的性能，选出一个最有的实现。其方法就是调用每种实现完成内存拷贝，比较执行时间。
不论是用户指定的，还是 DFB 自己选定的，最终这个实现会保存在 direct_memcpy 这个全局的函数指针中，在以后涉及内存拷贝的时候， DFB 都是使用 direct_memcpy。

DirectFB 源码解读之初始化-6

接着 dfb_core_create（）将完成多进程相关的初始化。
第一个概念是 fusion， 意为“融合”，是 Linux 提供的一个用于进程间通信的模块，分为内核态和用户态两部分，DFB 包中包含了 fusion 的用户态的源码，编译后也是一个动态链接库。fusion 只是 IPC 的一种，DFB 除了支持 fusion， 还支持基于 socket 的进程间通信。

另一概念是 arena，意为“竞技场”，这是 DFB 特有的概念，第一个进程负责初始化一个 arena，以后的所有进程都只是加入这个 arena。由于我们暂时对考虑多进程的情况，这部分内容暂且略过。

dfb_core_create（）中第一个有关多进程的函数是 fusion_enter()【lib\fusion\Fusion.c】，实际上在 Fusion.c 这个源文件中有三套函数， 对应三种情况：

```
#if FUSION_BUILD_MULTI
    #if FUSION_BUILD_KERNEL
            fusion_enter();//DirectFB 支持多进程，且 linux 内核中有 Fusion 模块
    #else
            fusion_enter();//DirectFB 支持多进程，但无 Fusion 模块，使用基于 socket IPC
    #endif
#else
```

fusion_enter();//DirectFB 不支持多进程。
#endif

Note:
(1). Fusion 的内核态模块需要 Linux 内核的支持，如有必要可能需要从网上下载并做移植。
(2). 在编译前运行 configure 时，通过指定"--enable-multi"使 DFB 支持多进程， 缺省是不支持的。

对于当进程的情况，fusion_enter()只是分配了一个 main pool 就返回了。

dfb_core_create（）接着调用的是 fusion_arena_enter（），这个函数会根据当前进程是不是不第一个，而调用 initialize 或 join，对于当进程的情况，它总是调用 initialize 函数，即 dfb_core_arena_initialize（），而它的工作除了创建各个 pool 的结构，就是调用 dfb_core_initialize（）完成 DFB 各个核心模块的初始化。

DirectFB 源码解读之初始化-7

上一节说到 dfb_core_initialize（）将完成各个核心部件的初始化。DFB 中，将以下几个方面作为 core part，即核心部件，它们是：

clipboard：管理剪切板，DFB 内部维护一块共享内存作为剪切板，并提供了 GetClipboardData 和 SetClipboardData 两个 API 给上层应用使用这个剪切板。

colorhash：DFB 支持 RGB, 也支持颜色的 Hash 表， 创建 surface 时如果指定的 pixel format 是 DSPF_LUT8 或 DSPF_ALUT44 ， 则该 surface 有关颜色的操作都是通过 Hash 表。用户使用的一般步骤是：
（1）创建一个 surface（pixel format 必须为 DSPF_LUT8 或 DSPF_ALUT44）
（2）创建一个 palette
（3）surface->setPalette
（4）surface->SetColor 等颜色操作。

surface： 写写画画的主战场。
system： 前面已经讲过了，特指 graphics system, 例如 fb,X11 等。
input: 对各种外设的抽象，包括鼠标、键盘、滚轮等，注意是这里的 input 概念与具体的输入驱动是不同的。在 DFB 包中有一个 inputdrivers 的目录包含了各种具体设备的驱动。而两者之间又是有关联的：上层应用在调用 DFB 中有关外设输入的接口，如 CreateInputEventBuffer，总是先进入抽象层，再由此进入具体的设备。
graphics：对各种 gfx driver 的抽象，与上面的 input 类似。
screen：屏幕的抽象
layer：层的抽象
wm：窗口管理

几个容易混淆的概念：surface, screen, layer。
surface：是用户作图的一块方形区域，对应一块内存，用户可以任意创建多个 surface，surface 与屏幕没有任何联系。
screen: 就是用户看到的屏幕，实际上对应的是系统中特定的一块内存，写到这块内存的东西会自动显示到屏幕上，这块内存可以通过 grahics system 操作，如 frame buffer 等， DFB 的用户并不能直接操作这块内存。surface 上的东西只有拷贝到这块内存中才会显示出来。
layer: 一般是一个与 graphics driver 有关的物理特性，不同的硬件可能支持的 layer 个数不同，种类也不同。不同的 layer 对应不同的 graphics card 中的不同内存。一个典型的应用是下面的一个 layer 显示 video，上面的 layer 显示字幕。layers 上的内容在现实到屏幕之前需要做混合等处理。

DirectFB 源码解读之初始化-8

下面看看 dfb_core_initialize()的源码【src\core\Core.c】:

```
 dfb_core_initialize( CoreDFB *core )
{

.....
     for (i=0; i<D_ARRAY_SIZE(core_parts); i++) {
          DFBResult ret;
          if ((ret = dfb_core_part_initialize( core, core_parts[i] ))) {
               dfb_core_shutdown( core, true );
               return ret;
          }
     }
     return DFB_OK;
```

}

这个函数及其调用的 dfb_core_part_initialize（）都很简单，就是调用各个 core_part 的初始化函数完成各个核心部件的初始化。

其中 core_parts[]定义在同一文件中：

```
static CorePart *core_parts[] = {
    &dfb_clipboard_core,
    &dfb_colorhash_core,
    &dfb_surface_core,
    &dfb_system_core,
    &dfb_input_core,
    &dfb_graphics_core,
    &dfb_screen_core,
    &dfb_layer_core,
    &dfb_wm_core
};
```

可是搜遍了整个 DFB 源码，好像并没有这些 core_part 的定义，怎么回事呢？原来 DFB 是通过宏 DFB_CORE_PART 来定义这些 core_part 的。每一个核心部件的源码中都有一个 DFB_CORE_PART，例如：

DFB_CORE_PART(clipboard_core, ClipboardCore);【src\core\Clipper.c】
DFB_CORE_PART(graphics_core, GraphicsCore);【src\core\Gfxcard.c】
DFB_CORE_PART(input_core, InputCore);【src\core\Input.c】
等等。

这个宏实际上是定义了一些函数，也就是每个核心部件都需要实现的接口：

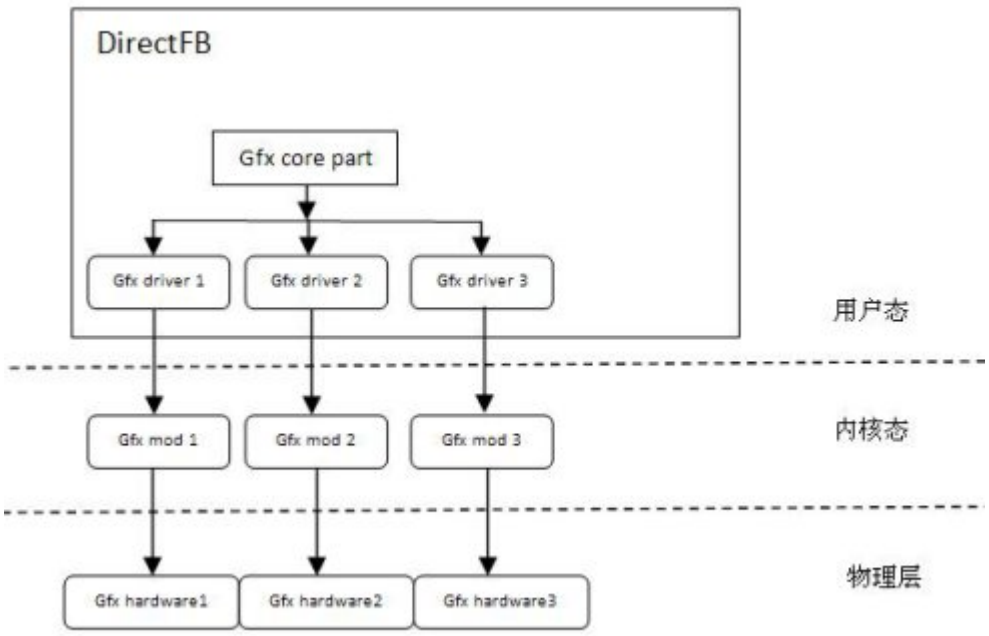
```
#define DFB_CORE_PART(part,Type)                                        \
.....
CorePart dfb_##part = {                                             \
    .name         = #part,                                          \
                                                                    \
    .size_local   = sizeof(DFB##Type),                              \
    .size_shared = sizeof(DFB##Type##Shared),                       \
                                                                    \
    .Initialize   = (void*)dfb_##part##_initialize,                 \
    .Join         = (void*)dfb_##part##_join,                       \
    .Shutdown     = (void*)dfb_##part##_shutdown,                   \
    .Leave        = (void*)dfb_##part##_leave,                      \
    .Suspend      = (void*)dfb_##part##_suspend,                    \
    .Resume       = (void*)dfb_##part##_resume,                     \
}
```

通过这个宏定义，实现了各个不同的核心部件对外接口的统一。

现在再看 dfb_core_part_initialize（）就很简单了，它调用各个核心部件的 Initialize()函数，也就是 dfb_##part##_initialize（）， 例如对于 graphics_core, 就是 dfb_graphics_core_initialize;对于 input_core，就是 dfb_input_core_initialize。从下一节开始，我们将选取几个核心部件看看其中它们初始化的流程。

DirectFB 源码解读之初始化-9

我们先看 graphics_core 这个核心部件的初始化。graphics_core 是一个与画图息息相关的一个核心部件，每个画图调用都会进入这个部件，并在此决定调用软件实现或是硬件实现。这个核心部件与 gfx driver 的关系如下：



有关上面的图， 有以下几点说明：

（1） graphics driver 分为内核态和用户态，为了区分我们称用户态为 gfx driver 而内核态的为 gfx module，内核态的 module 也就是传统意义上的设备驱动，负责直接操作 graphics 硬件，它的接口和实现与 DFB 无关。

（2）用户态的 gfx driver 是 DFB 的一部分，它向上的接口是统一的，而向下的接口则可能不同。每个 gfx driver 都是以动态链接库的形式存在。

（3）DFB 本身包含了一些 gfx driver 的实现，支持一些硬件，如果这些不能满足你的需求，你可以实现自己的 gfx driver 但是必须符合 DFB 的要求，即实现特定的一些接口。

（4）上图中出现多个 gfx driver 和 gfx module，是指在 DFB 编译时可以有多个 gfx driver, 即多个 gfx driver 的动态链接库，但运行时系统一般只有一个其作用，因为一般只有一个图形加速卡。

而这个部件的初始化，总结起来分为几个工作：

（1）找到与当前系统硬件匹配的 gfx driver

（2）调用 gfx driver 中的函数初始化该 driver,设置硬件绘图函数

（3）调用 gfx driver 中的函数初始化该 gfx 硬件

上面三个工作的核心是第一步，只要找到了正确的 driver，后面两步就水到渠成了。

```
static DFBResult
dfb_graphics_core_initialize( CoreDFB              *core,
                              DFBGraphicsCore      *data,
                              DFBGraphicsCoreShared *shared )
{
 .....
//card 是一个全局变量，代表系统的 graphics card, 如果下面找到了合适的 gfx driver，那么 card 中就记录了它的所有信息， 包括最重要的函数表。
 card = data;
 ......
//下面的这个函数我们在 dfb_system_lookup()的时候遇到过，它的作用是搜索并打开
//dfb_graphics_drivers 指定的目录中的所有动态链接库，并建立一个 modules 链表
//在打开动态库时，自动调用一个统一的注册函数，与 system 类似的是，DFB 通过宏定义
//DFB_GRAPHICS_DRIVER 统一了各种不同的 gfx driver，具体细节与之前相同，略过。
 direct_modules_explore_directory( &dfb_graphics_drivers );
 .....
//dfb_system_caps()等到当前系统的 caps 信息，其中的值表示系统是否支持硬件加速。
//dfb_gfxcard_find_driver()遍历上面的链表中的每一个 gfx driver， 调用它们的 probe 函
//数，即 driver_probe(), 而它最终调用的 system_funcs->GetAccelerator()，也就是当前
//系统的 GetAccelerator 函数，以 fbdev 为例，它返回的就是 dfb_fbdev->shared->fix.accel，
//这就是 fbdev 在初始化时，通过 ioctl 得到的 fbdev 的固定信息。
//至此我们得到了当前系统中存在的 graphics 硬件信息（实际就是一个整数值），每个 gfx driver
//将自己支持的硬件信息与当前系统的硬件信息比较，如果相同，则匹配成功。

 if (dfb_system_caps() & CSCAPS_ACCELERATION)
  dfb_gfxcard_find_driver( core );
 if (data->driver_funcs)
 {

   ...
      //进入这里，表示找到了匹配的 gfx driver
      //card->driver_funcs 指向的是 gfx driver 的统一接口，每个 driver 都是一样，如
InitDriver（），InitDevice（）等。
```

```
    //card->funcs 指向的是各个 gfx driver  的内部接口，每个 driver 是不同。
     //调用该 driver 的初始化函数完成 driver 的初始化，其中重要的步骤就是为 card->funcs 中的各个函数指针赋值，使其指向 gfx driver 自己的函数，以后上层用户需要画图时，DFB 就会通过这些函数指针，调用相应的硬件实现，完成硬件加速功能。
      ret = funcs->InitDriver( card, &card->funcs,
                               card->driver_data, card->device_data, core );
  ....
    //调用 InitDevice，一般是设置 gfx 硬件的寄存器，为正式的作图做好准备。
       ret = funcs->InitDevice( data, &shared->device_info,
                               data->driver_data, data->device_data );
    }

  //即使系统存在硬件加速，DFB  仍然提供了上层用户绕过硬件加速而只使用纯软件的画图。这在调试硬件或比较软硬件性能时很有帮助。这是通过 directfbrc 中的'hardware'指定的。
 if (dfb_config->software_only)
 {

  if (data->funcs.CheckState)
  {
         //在每个做图函数中都会查看 CheckState，  如果这个变量是  NULL, 则直接调用 DFB 自带的软件画图
            data->funcs.CheckState = NULL;
             D_INFO( "DirectFB/Graphics: Acceleration disabled (by 'no-hardware')\n" );
  }
 }
 .....
 return DFB_OK;
}
```

DirectFB 源码解读之初始化-10

现在我们看看 input_core 的初始化。

在进入具体的代码之前，我们先总结一下 input_core 这个核心部件的主要功能。我们知道计算机系统的外设有很多，不同的外设，接口不同，功能不同，提供的数据类型也不尽相同。例如键盘的事件是 KEY_RELEASE 或 KEY_PRESS, 而鼠标的事件是 BUTTON_PRESS 或 BUTTON_RELEAASE,还有触摸屏，游戏杆等等。所以 input_core 的功能之一就是统一不同输入设备的差异。第二，上层可能有多个进程在等待某个输入，而下层的输入设备并不知道，这种向多个进程分发事件的功能也是 input core 完成的。第三，系统中可能有多个设备对应同一个 driver，DFB 需要建立设备与 driver 的对应关系。

根据以前的解读，我们可以直接跳到 dfb_input_core_initialize（）开始 input_core 的初始化。这个函数做两件事：

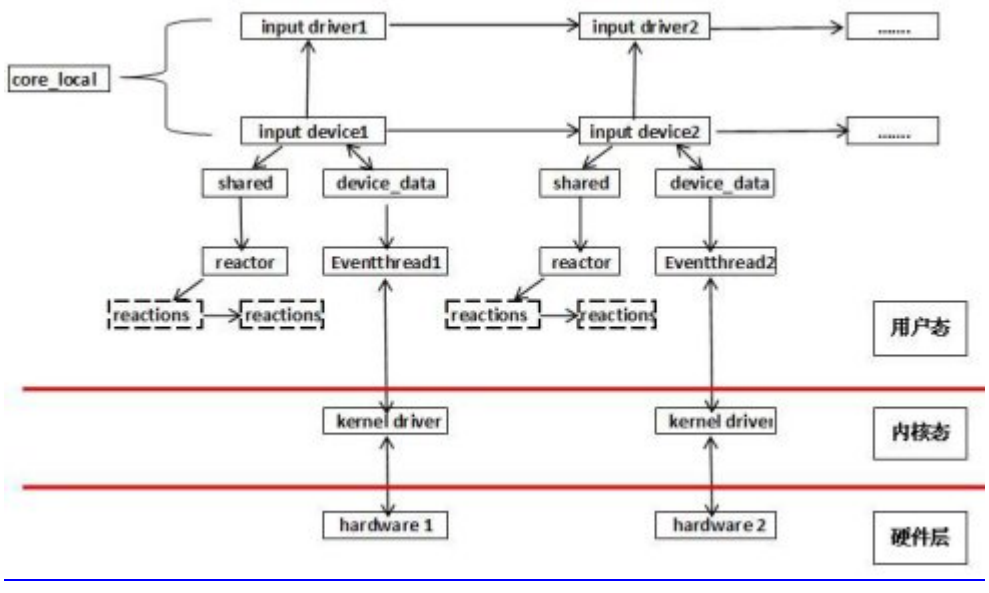
（1）direct_modules_explore_directory（）：这个函数又出现了，它就是在指定的目录中搜索所有的动态链接库，依次打开这些库，将这些库的信息（主要是函数表）记录在一个结构中，并挂在一个链表里。

（2）init_devices（）：初始化 input device。init_devices（）的主体代码如下：

```
static void init_devices( CoreDFB *core )
{
   //遍历每一个 module，一个 modeule 对应一个 input driver
     direct_list_foreach_safe (module, next, dfb_input_modules.entries)
     {
        //得到该 module 的函数表
          funcs = direct_module_ref( module );
             if (!funcs)
                continue;

          driver = D_CALLOC( 1, sizeof(InputDriver) );
        //得到当前系统中，该 driver 支持的设备总数，不同的设备 driver 实现自己的 GetAvailabe
        //GetAvailabe 的实现分两类，一是调用 access 看节点是否存在，而是调用 open 看是否打开成功。
       //同一 driver 可能对应多个设备。
            driver->nr_devices = funcs->GetAvailable();

            driver->module = module;
             driver->funcs   = funcs;
       //所有有效的 input driver 会记录在全局变量 core_local 的 drivers 链表中。
        direct_list_prepend( &core_local->drivers, &driver->link );
```

```
        for (n=0; n<driver->nr_devices; n++)
         {
           device = D_CALLOC( 1, sizeof(CoreInputDevice) );
             shared = SHCALLOC( pool, 1, sizeof(InputDeviceShared) );
             device->core = core;

        //打开设备，在成功打开设备后调用 direct_thread_create（），为每个设备指定一个事件处理线程, 这个线程负责读取设备的事件，完成初次转发。
             funcs->OpenDevice( device, n, &device_info, &driver_data );
        //为每个设备创建一个 reactor 的结构，其中有一个 reactions 的链表将来会记录所有关心该设备事件的对象，事件的转发会最终在此完成。
             shared->reactor = fusion_reactor_new( sizeof(DFBInputEvent), buf, dfb_core_world(core) );
        //下面这一步与当前分析没有太多联系，略过。
             fusion_call_init( &shared->call, input_device_call_handler,   device, dfb_core_world(core) );
            //device 与 driver，device 与 share（即 reactor）联系起来
              device->shared       = shared;
               device->driver       = driver;
            //所有的 device 都会记录在全局变量 core_local 的 devices 链表中。
              input_add_device( device );
      }
   }
}
```

上面有关 fusion 和 reactor 时，我们仍然只考虑单进程的情况。
在 input_core_part 初始化完成后，我们最终得到一个数据结构 core_local，将 input drivers, input devices, reactors 联系起来。其结构图如下：



有关上面的图， 有几点说明：
（1）DFB 中的 input driver 与 gfx driver 类似，只是 DFB 中的一个概念，不是真正意义的设备驱动。而上图中的 input driver 和 input device 只是对真正设备和驱动的一个抽象，在 DFB 中也数据结构的形式，将它们串联起来。
(2) 每一个设备对应一个 EventThread, 一经创建，即开始工作，不断从设备中读取事件，并开始处理分发这些事件。
（3）图中的 reactor 指向的是一个 reactions 链表，虚线表示这时并没有实际的 reactions 接入，也就是没有对这些设备及其事件感兴趣的应用。所以在上一步中，即使读到数据，在分发时都扔掉了。
（4）上图中的 driver 与 device 是一一对应的，但也可以是一对多的关系（一个 driver，多个 device）
（5）DFB 中自带了各种 input driver, 在实际运行时，只有系统中存在的设备的 driver 才会挂到上面的结构中。
（6）DFB 中定义了一个 DFBInputEvent 的数据结构，各种输入设备的事件都需要 mapping 到这个结构中。因此 EventThread 在开始分发事件前，需要做 mapping 的工作。
（7） 上图只针对单进程的情况。

下节我们看看输入设备事件传送的流程。

DirectFB 源码解读之外设输入处理流程

接着上一节，我们看看输入设备的事件是怎样从硬件传到 DFB 的最终用户的。下面的代码是 DFB 用户处理外设事件的一种方法：

```
DirectFBInit( &argc, &argv );
DirectFBCreate( &dfb );
dfb->EnumInputDevices( dfb, enum_input_device, &devices );//枚举得到系统存在并被 DFB 支持的所有外设，每找到
                                        //一个都会调用一次 callback 函数，你可以在这个
                                        //callback 函数中记录每个 device 的详细信息。
dfb->CreateInputEventBuffer( dfb, DICAPS_ALL, DFB_TRUE, &events ));//创建一个 InputEvent Buffer，第二个参
                                          //数表示该 buffer 关心的那种事件，
                                          //DFB 将所有的外设事件分为三类：key,
                                          //axis, button，你可以在程序中指定
                                          //三种的任何组合
if (events->WaitForEventWithTimeout( events, 10, 0 ) != DFB_TIMEOUT) //等待事件的发生。
{
        while (1)
        {
            DFBInputEvent evt;
            while (events->GetEvent( events, DFB_EVENT(&evt) ) == DFB_OK) //获取所有的事件
            {
              //do something.....                          //处理该事件
             }
            events->WaitForEvent( events );                 //继续等待事件
        }
}
```

需要注明的是：上面只是 input event 的一种使用方法，你也可以创建一个 input device，通过这个 device 创建一个属于它自己的一般的 Event buffer (而不是 input event buffer)。

现在我们就是顺着上面的代码看看 DFB 内部做了些什么？

（1）前两步DirectFBInit和DirectFBCreate完成DFB的初始化，与这个例子有关的就是input core part 的初始化，其结果如上一节所述《DirectFB 源码解读之初始化-10》。需要补充的是，在调用OpenDevice打开设备的时候，会得到device_info，它的内容是每个driver填写的，其中的一项desc.caps表明了该设备的类型：KEY, BUTTON 或AXIS。当然有的设备可以既有button，也有axis，如鼠标。

（2）dfb->EnumInputDevices 也很简单，根据上一节，所有 device 和 driver 的信息都记录在 core_local 这个全局变量的链表中，遍历这个链表即可。

（3）dfb->CreateInputEventBuffer,这一步做两件事情， 一是创建一个 Event buffer 对象（就如同用户调用 dfb->CreateEventBuffer 一样），包括初始化它的函数接口等。二是与 device 绑定。

先看看 Event buffer 这个对象包含了那些内容：IDirectFBEventBuffer 的私有数据即 IDirectFBEventBuffer_data，有两个我们关心的链表，一个是 devices 记录了与该 eventbuffer 关联的所有设备，一个是 events 记录了这个 eventbuffer 中所有的 event。

而 devices 与 eventbuffer 的关联，就是填充《DirectFB 源码解读之初始化-10》图中的虚线 reaction，每个 reaction，包含两个指针：IDirectFBEventBuffer_data和处理函数（统一为IDirectFBEventBuffer_InputReact），前一个表示谁关心该设备的事件，后一个表示有了事件后如何处理。链接工作具体是在IDirectFBEventBuffer_AttachInputDevice中完成的。调用流程如下：

前一节，我们说到，DFB 为每个有效的输入设备创建一个线程，该线程在得到下层的事件后，做一个简单的事件匹配，调用 dfb_input_dispatch，开始分发。它会调用 fusion_reactor_dispatch，而它会调用该 device 上的每个 reaction 的处理函数（IDirectFBEventBuffer_InputReact），该函数将事件挂到 event buffer 的 events 链表中，同时等待的线程（如果存在的话）。

（4）events->WaitForEventWithTimeout。如果 events 链表中有数据，则直接返回；否则调用 pthread_cond_timedwait,阻塞当前线程，直到发生事件或超时。

（5）events->GetEvent 就是从链表中取得 event。

DirectFB 源码解读之gfx硬件加速如何工作

刚刚说了 gfx core 及 gfx driver 的初始化，有必要趁热打铁看看 gfx 硬件加速的工作流程。

我们先看一下用户是如何调 DirectFB 来画线的,一般流程如下：

- DirectFBInit(&argc, &argv);
- DirectFBCreate(&dfb);
- dfb->CreateSurface(dfb, &sdsc, &primary);
- primary->DrawLine(primary, 0, 0, 100, 200);

上面的代码就是创建一个 surface 并在其上画一条从（0,0）到（100,200）的斜线。

我们可以沿着上述流程看看系统是如何一步步的调到硬件的呢？

1. DirectFBCreate 在 dfb_core_create 完成各个 core 的初始化后，会调用 IDirectFB_Construct 设置 DirectFB 的总接口：IDirectFB。 IDirectFB 也就是上面 DirectFBCreate 返回的那个参数，DFB 所有的其他接口那么是这个总接口直接创建，要么是它的子孙创建的。在 DFB 中，接口实际上是用 C 语言实现的一个类，里面有私有数据，还有函数等。设置总接口的任务之一是设置它的函数指针。
其中有一个函数指针是：thiz->CreateSurface = IDirectFB_CreateSurface;

2. 当用户调用 dfb->CreateSurface 时，实际调到的是 IDirectFB_CreateSurface，而它又会创建 surface 这个对象并调用 IDirectFBSurface_Construct 完成 surface 接口中各个函数指针的赋值工作，其中有一个函数指针是: thiz->DrawLine = IDirectFBSurface_DrawLine。

3. 当用户调用 primary->DrawLine，实际调到的是 IDirectFBSurface_DrawLine
IDirectFBSurface_DrawLine 会调用 dfb_gfxcard_drawlines()，而它首先会调用 dfb_gfxcard_state_check 看看系统是否支持硬件加速，如果 card->funcs.CheckState等于NULL，则表示不支持或不用硬件加速这直接返回，调用软件实现， 这个我们在《DirectFB 源码解读之初始化-9》讲过 。

4. 如果通过了 dfb_gfxcard_state_check 的检查，则继续调用 dfb_gfxcard_state_acquire，设置下面硬件寄存器的状态和值，包括颜色、目标地址、源地址等等（通过 card->funcs.SetState）。

5. 在第 3 步和第 4 步都成功后，就可以调用真正的作图函数，如card->funcs.DrawLine了，根据《DirectFB 源码解读之初始化-9》我们知道，这个函数指向的就是gfx driver中的作图函数，即硬件加速。例如，对于ATI128，它指向的就是ati128DrawLine（）【gfxdrivers/ati128/Ati128.c】

至此我们完成了整个 graphics driver 中的硬件加速的画图流程。我们虽以 DrawLine 为例，但实际上所有其他的画图流程完全类似。唯一有区别的是：操作不同，涉及的寄存器不同。例如对于画线，需要设置画线的颜色；而对于 blit 操作，需要知道 src surface 的地址等。

DirectFB 源码解读之双缓存实现

双缓存是画图时一个常用的技术，它的基本原理是在其中一个缓存中作图，完成后提交显示，同时在另一块缓存中继续作图，这样两块缓存交替画图-显示，实现了两者的同步进行，提高了效率。

在 DirectFB 中一个缓存实际就是一块内存。DFB 支持两种缓存分配方式:(1)用户自己分配，并在 createSurface 是将该内存地址传递给 DFB，这种方式需要在 createSurface 时指定 DSCAPS_PREMULTIPLIED 属性（2）DFB 自动分配，大部分用户使用这种方式。

DirectFB 支持双缓存或三缓存，用户编程时，只需在调用 dfb->CreateSurface 时指定 DSCAPS_DOUBLE 或 DSCAPS_TRIPLE 即可。而除此以外，多缓存对于用户是透明的。

下面是一个简单实现动画的例子：一条横线自上而下的移动，在这里我们让 DFB 自动分配缓存并设定它是双缓存的：

```
 dfb->CreateSurface(dfb, &sdsc, &surface);
for(i=0;i<100;i++)
{
        surface->DrawLine(surface, 100, 100+i, 200, 100+i);
```

```
        sleep(1);
        surface->Flip(surface, NULL, DSFLIP_WAITFORSYNC);
    }
    surface->Release(surface);
```

第一步 CreateSurface（），创建一个 Surface 对象并初始化这个对象的函数指针，同时设置该 surface 的一些基本属性如大小，格式（pixel format）等。但是并没有为该 surface 创建缓存，也就是没有实质的内存分配。

第一次调用 drawline 时，首先会调用 dfb_gfxcard_state_check 检查和设置各个状态，其中会调用 dfb_surface_get_buffer 得到目标 buffer，即在那个缓存上画。它的代码如下：

```
dfb_surface_get_buffer( CoreSurface            *surface, CoreSurfaceBufferRole   role )
{
  return surface->buffers[ surface->buffer_indices[(surface->flips + role) % surface->num_buffers] ];
}
```

关于这个函数：
#role 即 uffer 的角色，有三种 CSBR_FRONT（0）， CSBR_BACK（1），CSBR_IDLE（2），分别表示前缓存，后缓存和闲置缓存。
#通常，总是在前缓存中画，即 role 总是 CSBR_FRONT。另外，前缓存和后缓存只是一个逻辑概念，它所指向的 buffer 是交替变化的，
#surface->buffers[]是一个 buffer 的数组，buffers[0]指向第一个 buffer 对象，buffers[1]指向第二个 buffer 对象，等等。该数组的大小是 MAX_SURFACE_BUFFERS，即 6 个。
#surface->buffer_indices[]是一个整数数组，它的大小也是 6，我个人认为不需要这个变量，仍然可以工作，看不出他的真正用途是什么。
#surface->num_buffers 记录了这个 surface 有效的 buffer 数量，如果指定了 DSCAPS_DOUBLE，那它就是 2；如果指定了 DSCAPS_TRIPLE，那它就是 3.
#surface->flips 是一个整数值，系统每次调用 Flip 函数，这个值就会加 1。

这样，我们就很容易理解上面的那个函数。例如，第一次 drawline 时，调用 dfb_surface_get_buffer（）， role 是 FRONE，即 0， flips = 0，则返回的就是第一个缓存；调用 Flip 显示之后，第二次画图时，role 仍然为 FRONT,因为用户总是在前 buffer 中作图，而 flips 变成了 1，这时上面的函数返回的是第二个缓存。第三次画图时，又返回第二个 buffer，依次类推。

最终得到的 buffer 内存地址将传递给实际的作图函数，并由它在内存中完成作图。

这就是 DFB 双（多）缓存的实现。

DirectFB 源码解读之字体-1

我们在 DirectFB 初始化中了解到 gfx driver, input driver 等都是在 DirectFBCreate（）时完成初始化，也就是说在用户真正使用之前，这些 driver 已经准备就绪。

而字体（font）与此不同，只有用户明确使用字体时，才会进行初始化及资源分配，类似还有 Image 和 Video，在 DirectFB 中它们通称为 Interface。源码对应的目录就是 DirectFB-1.4.0/Interfaces。

在 DirectFB 运行环境中，interface 的存在形式也是动态链接库。对应的目录是：lib/directfb-1.4.0/interfaces/IDirectFBFont 等。

当前的 directfb 支持三种字体文件：

（1）FreeType2。有关freetype2 的资料，可以参考官方网站：http://freetype.sourceforge.net/。

(2) DGIFF 字体。 DGIFF 是 DirectFB Glyph Image File Format 的简称，从名字就可以看出，这是 DirectFB 所特有的一种字体格式。 DFB 在 tools 目录中有一个 mkdgiff 可以将 TrueType 的字体文件转化为一个 DGIFF 字体文件。（命令为：./mkdgiff -f A8 -s 10,20,30 one.ttf > one.dgiff，将字体文件 one.ttf 转化为 DGIFF 格式，结果保存在 one.dgiff 中，指定字体的格式是 A8，大小支持 10，20，30），DGIFF 与 FreeType2 的一个重要区别是 FreeType2 可以支持无限大小的字体，而 DGIFF 只支持一定个数的字体大小，例如对于上面的 one.dgiff 它只支持 10，20 或 30，三种大小的字体。

(3) 缺省字体。如果系统不支持 FreeType2，也不想是使用 DGIFF,则可以使用 DFB 中提供了一种缺省字体，这种字体固定大小的，也就是说指定是 non-scalable 的字体。

DFB 使用字体的一个例子：

```
    DirectFBInit( argc, argv );
    DirectFBCreate( &dfb );
//创建字体对象
   font_dsc.flags = DFDESC_HEIGHT;
   font_dsc.height = 10;
   dfb->CreateFont (dfb, "my.ttf", &font_dsc, &font_1);
```

```
    dfb->CreateSurface(dfb, &sdsc, &f_surface);
    f_surface->Clear(f_surface, 0x0, 0, 00, 0);
//将字体对象与 surface 关联
     f_surface->SetFont(f_surface, font_1);
    f_surface->SetColor(f_surface, 0xff, 0, 0, 0);
//画字体
    f_surface->DrawString(f_surface, "1234567890", -1, 0, 50, DSTF_LEFT);
    f_surface->Flip(f_surface, NULL, DSFLIP_WAITFORSYNC );
```

如果在调用 CreateFont() 时,字体文件设为 NULL,则 DFB 会自动调用 Default font 即缺省字体，而这时的 font_dsc.height 是不起作用的。

DirectFB 源码解读之字体-2

我们以 DFB 中的 FT2 为例，研究一下 DFB 与字体库之间的分工。

先看一个使用 FT2 的例子：

```
  FT_Init_FreeType( &library );
  FT_New_Face( library, "example.ttf", 0, &face );
  FT_Set_Char_Size( face, 20*64, 0,72, 0 );
  FT_Load_Char( face, 'M', FT_LOAD_RENDER );
 //to show the Char
  FT_Done_Face     ( face );
  FT_Done_FreeType( library );
```

总结下来，FT2 画一个字符，需要经过的步骤：

（1）初始化 FT 库。

（2）根据指定的字体文件创建一个 FACE。如果字体文件不是 FT2 支持的，则创建失败。

(3) 设定字体的大小。字体的高度或宽度单位是 1/64 像素，而 resolution 的单位是 dpi

（4）加载指定的字符。其结果存放在 face->glyph->bitmap 中，这时就可以对一个字符进行一些处理如显示等

（5）释放资源。

从上面 FT2 的处理过程我们看出，有一些事情 FT2 并没有处理：

（1）字符的显示。这是涉及到字符显示的颜色，字符显示的格式转换，字符显示的位置等

（2）字符的缓存。在上面的例子中 FT_Load_Char（）是相对比较耗时的操作，而在使用字体是，往往很多字符是重复的，如果对于重复的字符，将 bitmap 的结果保存下来，就不需要每次都调用 FT_Load_Char

（3）字符串的分解。FT2 处理的是单个字符，而用户往往使用的是字符串，FT 不支持字符串分解为字符的操作。

上面这三种工作是所有的字体库都不支持但却是必须的，他们会由 DFB 完成。另外，作为一个框架，DFB 还支持多种字体库。

DirectFB 源码解读之字体-3

前面我们了解了 DFB 字体的用法以及 DFB 与字体库的关系，现在我们进入代码，看看 DFB 是如何具体管理字体的。

实际上，DFB 所有与字体有关的逻辑，几乎可以被下面这几句调用所覆盖：

```
(1)  dfb->CreateFont(dfb, "myfont.ttf", desc, &myfont);
(2)  mysurface_1->SetFont(mysurface_1, myfont);
(3)  mysurface_1->DrawString(mysurface_1, "aAbB", -1, 50, 100, DSTF_LEFT);
(4)  mysurface_2->SetFont(mysurface_2, myfont);
(5)  mysurface_2->DrawString(mysurface_2, "aAbB", -1, 50, 150, DSTF_LEFT);
```

第一句，创建字体对象，需要指定一个字体文件，以及在 desc 中指定字体大小

第二句，将字体与 surface_1 关联，以后该 surface 上调用 DrawString()或 DrawGlyph()都会用这个字体

第三句，在 surface_1 的指定位置（50，100）输出字符串

第四句，将字体与 surface_2 关联

第五句，在 surface_2 的指定位置（50, 150）输出同样的字符串，虽然第三句和第五局的调用几乎完全相同 ，但在 DFB 中的流程却完全不一样，第五句与字体库没有任何关系，直接是从 DFB 缓存中提取字体图像的。

创建字体对象

我们知道， DFB 支持三种字体：DGIFF, FT2 和 default font。在创建字体对象之前，首先调用 DirectGetInterface（），找到合适的字体 interface。其过程与 gfxdriver 及 input driver 类似，就是依次打开 lib\Direct-1.4.0\Interface\IDirectFBFont 目录中的每一个动态链接库（每个动态链接库实现了一种字体 interface，对应一种字体库）， 然后调用动态库中的 Probe 函数来判断当前的动态库是否与 CreateFont 中指定的字体文件匹配。

对于 Default Font, 如果字体文件为 NULL, 则匹配

对于 DGIFF, 如果字体文件头中包含字符串"DGIFF", 则匹配

对于 FT2, 如果 FT_NEW_FACE()调用成功，则匹配。（FT_NEW_FACE()是 FT2 中的一个标准函数，其作用是根据字体文件创建一个 FACE），因此对于 FT2，DFB 直接将问题直接踢给了 FT2。

在找到合适的字体 interface 后，就会调用该 interface 的 Construct()函数。我们以 FT2 为例看看 Construct 的实现。

FT2 的 Construct()首先会调用 FT_Init_FreeType()初始化 FT2 字体库，接着调用 FT_New_Face（）创建一个 FACE，然后调用 FT_Set_Char_Size（）设置字体大小，这些调用都是与 FT2 字体库有关的。

接着 Constrct()调用 dfb_font_create（）创建一个 CoreFont，初始化这个数据结构。为了以后的代码解读，我们有必要看看这个数据结构的一些重要字段：

```
struct _CoreFont {
.............
     DFBSurfaceBlittingFlags       blittingflags;
     CardState                     state;
     DFBSurfacePixelFormat         pixel_format;
     DFBSurfaceCapabilities        surface_caps;
//CoreFontCacheRow 是 DFB 字体中的一个重要结构，是实际存放字体图像的地方。每个 CoreFontCacheRow 包含一个 surface（实际上就是一块 buffer），每个字符在ＦＴ２中对应一张 bitmap 图像，在将这个图像绘制到最终的 surface 之前，需要先将其拷贝到这个字体对象内部的 surface 上，以后用户如果再画同样的字符，就直接从这个 surface 中取，这就是 DFB 所谓的对字体有缓存
     CoreFontCacheRow            **rows;

     int                         row_width;
//每个 row 对应的 surface 的缓存大小是一定的，即存放的字符是一定的，如果字体库中的字体数量很多，一个 row 的 surface 可能不够，需要创建新的 row，max_row  记录了 DFB 支持的最大 row 的数量（当前值是 5）
     int                         max_rows;
//num_rows 记录了当前 rows 的数量，num_rows<=max_rows
     int                         num_rows;
//active_row, 记录了当前活动的 row，
     int                         active_row;
//row_stamp 用于记录每个 row 的使用频率，每次绘制的字符如果属于每个 row，则该 row 的 row_stamp 会加 1，这个变量的作用在于 row 的删除替换。当 row 的数量达到 max_rows 时，而又有新的字符需要绘制，这时显然不能分配新的 row 了，需要现存的某个 row 进行删除，而其依据就是 row_stamp。
     unsigned int                row_stamp;
//rows 是存放字体图像的地方，就像是一个仓库，如何从仓库中快速拿到所需的货物，也是 DFB 需要考虑的问题。为此，DFB 提供了两种途径，一是数组，二是哈希表，他们中的每一项都存放了 rows 中的一个字符图像的索引，两者的区别是数组的大小固定，为 128，存放的是前 128 个字符（实际上，大部分字符都在这个范围内），而哈希表的大小是可变的，存放的是除去数组之外的其他字符。
     DirectHash                  *glyph_hash;
     CoreGlyphData               *glyph_data[128];
//下面这两个函数负责将字符图像拷贝到指定的 surface 中，与各个实现有关
     DFBResult                   (* GetGlyphData) ( CoreFont      *thiz,
                                                    unsigned int    index,
                                                    CoreGlyphData *data );
     DFBResult                   (* RenderGlyph)   ( CoreFont      *thiz,
                                                    unsigned int    index,
                                                    CoreGlyphData *data );
.........
};
```

综上所述，在用户调用 CreateFont()时， DFB 会调用 Probe 和 Construct(这也是每个字体 interface 需要实现的接口)完成字体库的初始化及字体对象的创建。

DirectFB 源码解读之字体-4

字体关联

字体与 surface 的关联是在 SetFont 中完成的。其过程很简单，就是将字体对象记录在 surface 的一个指针中，以后该 surface 上画字符，就会找到相应的字体对象。

需要说明的是，一个字体对象可以同时属于多个 surface，而每个 surface 在某个时刻最多拥有一个 surface 对象。

第一次绘制字符串"aAbB"

用户在调用 surface->DrawString(）时，对应 DFB 中的函数是 dfb_gfxcard_drawstring（），这也是绘制字符串的主函数。

它的工作包含以下几步：

（1）对需要绘制的字符串进行解码，解码的结果是得到每个字符的索引值。字符的索引值的定义依赖于不同的字体库，对于 FT2,最终是调用 FT_Get_Char_Index（）得到索引值，而对于 default font，其索引值就是字符对应的整数值。总之，索引值与字符是一一对应的。

（2）对于每一个字符，首先根据它的索引值在字体对象的字形数组或字形 HASH 表中查找它的字形对象（DFB 中称为 CoreGlyphData），因为是第一次画，必然找不到。（如果索引值小于 128，在数组中查找，否则在哈希表中查找）。字形数组和 hash 表的定义如下：

（3）接着，调用 dfb_font_get_glyph_data（）得到字符的字形对象。该对象的数据结构如下：

```
typedef struct {
    DirectLink      link;
    unsigned int    index;
    unsigned int    row;
    CoreSurface    *surface;
    int             start;
    int             width;
    int             height;
    int             left;
    int             top;
    int             advance;
    int             magic;
} CoreGlyphData;
```

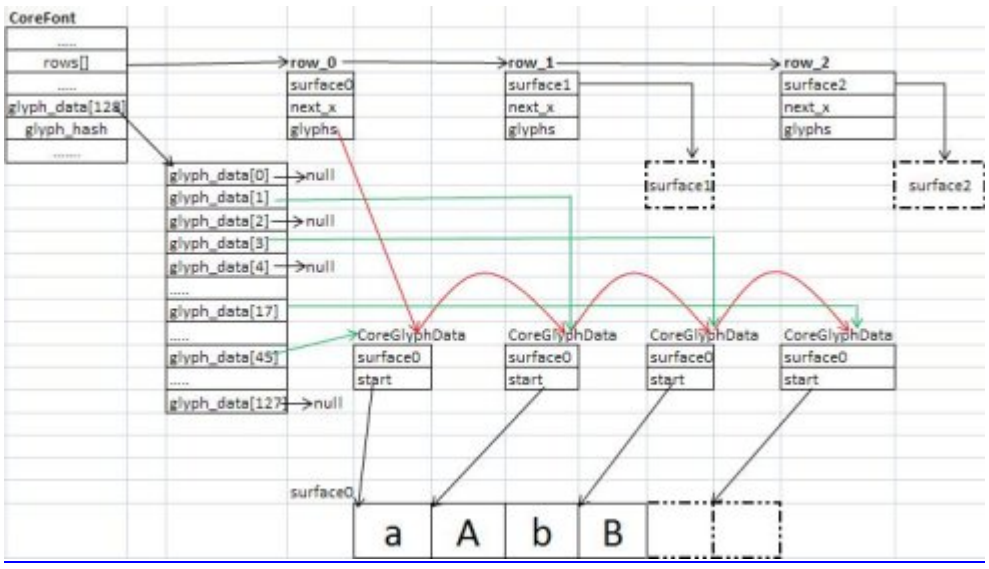
首先，为字符分配一个新的字形对象：CoreGlyphData。

然后，调用 font->GetGlyphData（）绘制字符，这时绘制的字符仍然存在字体库的 buffer 中。

其次，分配一个新的 row（包含一个 surface）用于存放该字形。

再次，调用 font->RenderGlyph（）将该字符绘制到 row 中的 surfaceace 上，其中 CoreGlyphData->surface 就指向这个内部的 surface, CoreGlyphData->start 指向了这个字符在 surface 上的起始位置。CoreGlyphData->width 和 CoreGlyphData->height 指向的是字符的宽度和高度，这几个变量就可以确定 surface 上的包含该字符一个矩形

最后，将该字符对应的字形对象存放到了 hash 表或字形数组中。所以这时的 CoreFont, CoreGlyphData 和 CoreFontCacheRow 三者的关系如下：



有关上面这个图，有几点需要说明：

@每个 CoreFontCacheRow 都拥有一个 surface 用来绘制字符，CoreFontCacheRow->next_x 指向 surface 中下一个可以绘图的地址，每次绘制一

个字符，next_x 都会增加。

@如果 next_x 指到了该 surface 的最后，或者 surface 所剩的空间不足以绘制一个字符，则 DFB 会分配一个新的 CoreFontCacheRow 并为其分配新的 surface，而系统中最多支持 5 个 CoreFontCacheRow（这个值是在创建字体对象 CoreFont 时指定的）

@CoreFontCacheRow->glyphs 是一个链表，链表中的每一项都是一个 CoreGlyphData 对象，我们知道字符与 CoreGlyphData 是一一对应的，因此该链表记录了所有绘制在该 row 上的字形对象。

@如果 5 个 CoreFontCacheRow 都用完了，还是不能存放所有的字符，这时就需要替换，替换的原则是 LRU(最近最少用)，CoreFontCacheRow->stamp 记录了该 row 的使用情况，每次属于该 row 的字符被绘制，该值加 1，例如 row0 中有字符 a, row1 中有字符 M, 用户调用 surface->DrawString(surface, "aMMMaaaMMMM", ...)后,row0 的 stamp 将加 4， 而 row1 的 stamp 将加 7.

@在 row 的替换时，需要释放原来 row 的资源，包括它的 surface，以及 row->glyphs 链表中所有字形对象

@CoreFont->glyph_data[128]是一个指针数组，初始时，每个元素都是 NULL, 在绘制字符时，该字符对应的字形对象 CoreGlyphData 会记录在这个数组中，而记录的位置就是它的索引值，以后再绘制同样的字符，就可以从数组中直接找到。

@如果字符个数多于 128，则要用到 hash 表。

（4）将字符从 font 对象内部的 surface，拷贝到目标 surface，即调用 DrawString 的 surface。拷贝的过程就是 blit 的过程，根据不同的系统，可以调用硬件支持 blit 或软件 blit。

（5）转向（2）绘制下一个字符，在绘制接下来的几个字符"AbB"时，就不需要分配新的 row 了，因为第一次分配的 row 足够放下这几个字符了。

DirectFB 1.4 移植及运行

1. download the source package from www.directfb.org
2. unzip the package and run "./configure --prefix=/11 --exec-prefix=/12 --host=arm-linux
"
3.run "make" to compile DFB.它报一个 error "/DirectFB-1.4.0/gfxdrivers/matrox", 如果 matrox 不是必须的，可以直接在 graphics/Makefile 中将 MATROX_DIR 置空即可。
4. make install
所有的头文件将拷贝到/11 中，所有的库文件将拷贝到/12 中。如果编写应用程序，只需将路径指向这两个目录即可。
5. 程序运行
（1）将--exec-prefix 指定的目录复制到开发板上，对于上面的例子，就是将 12 及其子目录拷贝到开发板的根目录中，否则程序会报错， 例如“DirectFB/core/system: No system found”等。
(2)export LD_LIBRARY_PATH=$LD_LIBRARY_PATH:/12/lib，否则系统会报错，例如“error while loading shared libraries: libdirectfb-1.4.so.0”
(3)运行程序，即可。