Understanding X From Source Code

THOSS X Group
blc03@mails.tsinghua.edu.cn

December 21, 2006

Contents

前言

这篇文档将从X的源码出发深入的考察X服务器各种基本概念及其实现方式。从中，我们将看到一个桌面系统的基础框架是如何设计和实现，深入的分析将使我们学到珍贵的系统设计的各种方法、编程的各种技巧、方案取舍的各种原则。

由于各种概念的交叉，加上文档的不成熟，这篇文档最佳的阅读方式如下：

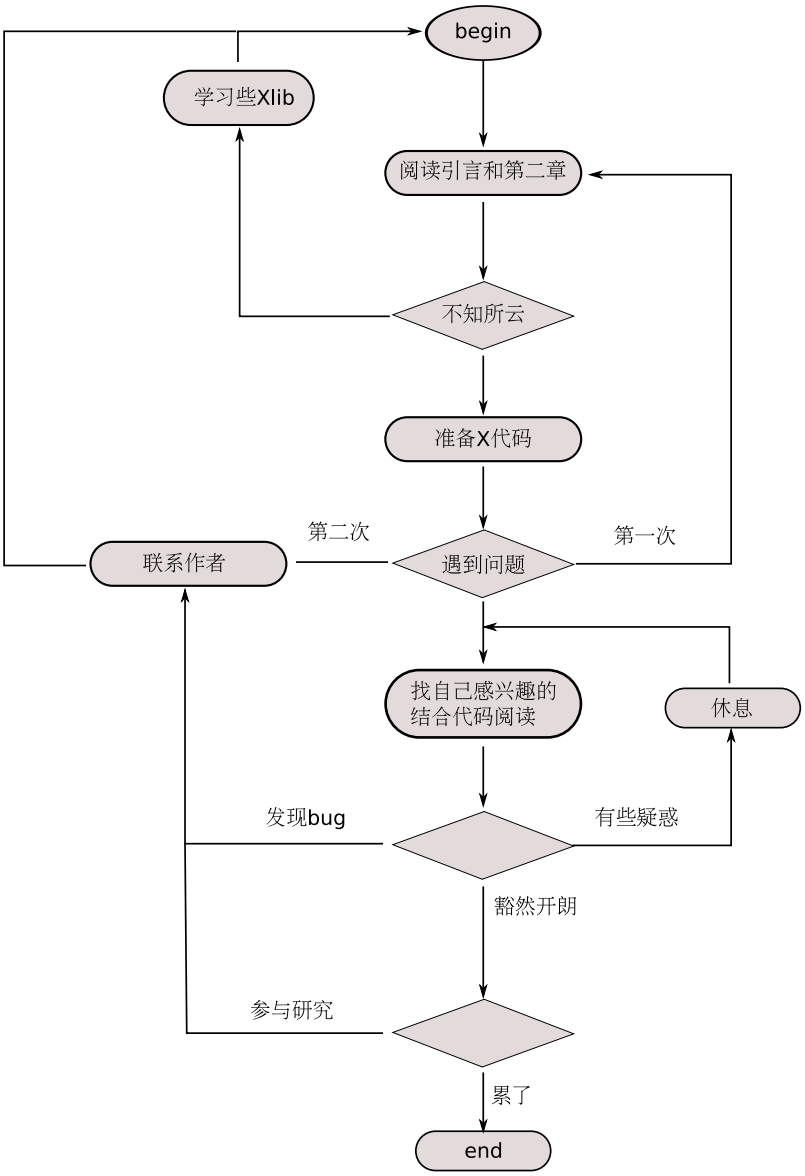

Figure 1: 阅读方法

在后续的章节中，我们在每一章的开头提出这一章将要解决的问题，然后再结合源码的分析和阐释中回答这些问题。整篇文档的目标包括：

- 深入理解X概念
- 掌握X服务器的代码

- 学习系统设计方法, 熟悉常用的编程技巧,学会实现方案的取舍原则

1 引言

在第一章中，我们将看看X的基本图景：X的基本概念和基本结构。考虑到这是一篇关于X源码的文档，因此，这里不会给出太多的细节，并且我们会结合代码来考察这一基本图景。

1.1 X系统基本概念

X是一个复杂的系统，但是它所基于的基本概念却简单易懂:

- Displays和Screens: 显示系统和屏幕
- Server-Clinet Model: 服务器-客户端模型
- Window Management: 窗口管理
- Events: 事件
- X Extensions: X扩展

显示系统和屏幕

在X系统中，显示系统被定义为由一个键盘、一个鼠标、一个或多个屏幕组成的工作站。

服务器-客户端模型

X是面向网络的窗口系统，一个应用程序不需要在实际完成显示的工作站上运行，它可以作为客户端运行在远程机器上。X服务器运行于本地, 控制程序的显示并获得用户的输入。两者间的交互通过相应的协议在网络上传输。客户端运行于远程，服务器运行于本地，这是与普通的服务器-客户端模型的重要区别。

控制显示的程序被称为一个服务器(server), 它需要完成的工作包括:

1. 控制显示系统，允许多个客户程序的访问。
2. 把用户输入传递给相应的客户。
3. 解释并执行来自客户的请求。
4. 维护复杂的数据结构。

窗口管理

在X系统中，一个应用程序并不实际控制窗口在哪里显示、大小是多少等等事情。由于多个客户的存在，一个客户不能依赖于特定的窗口配置，它只能告诉它所希望的显示的位置和大小。屏幕的布局以及用户的交互方式由一个单独的程序控制，称之为窗口管理器(`window manager`)。

窗口管理器实际上也是使用了Xlib编写的一个应用程序，不过它被赋予了特殊的权限，以便管理屏幕上窗口布局。

事件

X扩展

X系统是可扩展的，它定义了一套扩展机制。并不是每个服务器都支持所有的扩展，所以应用程序在使用一个扩展的时候，必须首先询问服务器是否提供该扩展。

1.2 X系统软件体系结构

1.3 准备源码

可以到Xorg站点参考源码的下载和编译方法:

http://wiki.x.org/wiki/ModularDevelopersGuide

欢迎把编译中遇到的问题放到版上[1]讨论。并将讨论的结果加入到这篇文档中来。

推荐使用kscope阅读源码，它依赖于**cscope, dot, ctags**三个程序。

1.4 源码目录结构

app 这个目录下包含一些X应用程序，包括twm这个最简单的窗口管理器。

doc 这个目录下包含的文档包括X协议规范，X部分模块(比如安全模块)等的设计文档，但是缺乏对Xserver整体设计的描述。另外，Xlib的参考手册可以在`xorg-docs/hardcopy/X11` 下找到。

xserver 这个目录下包含xserver的源代码。

lib 包含X服务器需要使用的库，有些库实际上只能被X服务器代码使用，将它们独立出来是为了更好的模块化。

[1] http://www.newsmth.org/bbsdoc.php?board=THOSS

2 XServer代码框架

xserver目录下包含多个X服务器的源码，包括**Xorg, Xnest**和基于**kdrive**的多个服务器。

X服务器的代码分为四个部分组织：

- 设备无关层(DIX), 这一部分的代码被所有Xserver的实现共享。
- 操作系统层(OS), 这一部分的代码随着操作系统的不同而不同，但是被这个操作系统上的图形设备共享。
- 设备相关层(DDX), 这一部分随着操作系统和图形设备的组合的不同而不同。
- 扩展接口，这一部分为用统一的方式向X server加入新的功能提供支持。

DIX层有如下的八类资源:

- Window
- Pixmap
- Screen
- Device
- Colormap
- Font
- Cursor
- Graphics Contexts

每种资源用相应的struct表示，其中包括三个部分：

- 属性段，用于保存相应的信息。
- 函数指针
- 私有字段，DDX代码用来保存似有数据。

这些结构定义在有对应名称的两个.h文件内，比如screen对应于`screenint.h`和`scrnintstr.h`。`screenint.h`定义外部使用时需要的结构和函数原型，但不暴露内部的实现; `scrnintstr.h` 中定义内部实现用的数据结构。

DIX通过调用结构的函数指针来完成它的工作。这些指针由DDX直接或间接的设定。属性段一般由DIX层设定，DDX不能修改它们。

DIX代码放置在`dix/`目录下，保括

> main.c window.c colormap.c, dispatch.c, events.c ...

每个X概念，基本上都有一个对应的实现文件。

服务器相关部分的代码放置在`hw/`目录下。其中，`hw/xfree86/`目录下包含了Xorg的代码;`hw/kdrive`下包括了基于kdrive的多个服务器。由于基于kdrive的服务器比较的简单，我们将从其中的一个`Xephyr`入手。关于这个服务器的说明，可以参见`hw/kdrive/ephyr/README`, 这里引用一部分:

> Xephyr is a a kdrive server that outputs to a window on a pre-existing 'host' X display. Think Xnest but with support for modern extensions like composite, damage and randr.
>
> Unlike Xnest which is an X proxy, i.e. limited to the capabilities of the host X server, Xephyr is a real X server which uses the host X server window as "framebuffer" via fast SHM XImages.

It also has support for 'visually' debugging what the server is painting.

kdrive的介绍可以参见相应的man文件,`hw/kdrive/Xkdrive.man`:

```
man ./Xdrive.man
```

2.1 八类资源

下面我们通过展示相应的数据结构来搞清楚八种资源是如何在X系统中维护。

Screen

X支持一个显示系统包含多个屏幕。处理屏幕所需要的数据信息被包含在ScreenRec 中(`xserver/-include/scrnintstr.h`):

```
typedef struct _Screen {
    int                 myNum;  /* index of this instance in Screens[] */
    ATOM                id;
    short               width, height;
    short               mmWidth, mmHeight;
    short               numDepths;
    unsigned char       rootDepth;
    DepthPtr            allowedDepths;
    unsigned long       rootVisual;
    unsigned long       defColormap;
    ...

    /* anybody can get a piece of this array */
    DevUnion    *devPrivates;
    ...

    /* Random screen procedures */
    CloseScreenProcPtr      CloseScreen;
    ...
    /* Window Procedures */
    /* Pixmap procedures */
    /* Font procedures */
    /* Cursor Procedures */
    ...
} ScreenRec;
```

如前面所介绍的，Screen对应的结构包含三个部分：

- 属性段: 保存信息。
- 函数指针: 指向完成相应的操作所用的例程，由DDX代码设定。
- 私有字段，DDX代码用来保存私有数据。

全局变量`screenInfo`保存了指向所有`ScreenRec`的指针数组:

```
typedef struct _ScreenInfo {
    int         imageByteOrder;
    int         bitmapScanlineUnit;
    int         bitmapScanlinePad;
    int         bitmapBitOrder;
```

```
    int         numPixmapFormats;
    PixmapFormatRec
                formats[MAXFORMATS];
    int         arraySize;
    int         numScreens;
    ScreenPtr   screens[MAXSCREENS];
    int         numVideoScreens;
} ScreenInfo;

extern ScreenInfo screenInfo;
```

同时，screenInfo也保存了诸如bitmap的bit-order等的全局配置。

Device

Window,Pixmap,Drawable

Window和Pixmap都是Drawable的子类，drawable可以视为一个可以在其表面画图的平面，它或者是一个屏幕上的window, 或者是一个内存中pixmap。

drawable对应的数据结构如下(`include/pixmapstr.h`):

```
typedef struct _Drawable {
    unsigned char       type;   /* DRAWABLE_<type> */
    unsigned char       class;  /* specific to type */
    unsigned char       depth;
    unsigned char       bitsPerPixel;
    XID                 id;     /* resource id */
    short               x;      /* window: screen absolute, pixmap: 0 */
    short               y;      /* window: screen absolute, pixmap: 0 */
    unsigned short      width;
    unsigned short      height;
    ScreenPtr           pScreen;
    unsigned long       serialNumber;
} DrawableRec;
```

type字段的取值为`DRAWABLE_PIXMAP`, `DRAWABLE_WINDOW`, `UNDRAWABLE_WINDOW`;`UNDRAWABLE_WINDOW`用于标识`InputOnly`类型的window。serialNumber 字段的取值在所有的drawable中唯一，它用于确认GC中裁剪信息的有效性。Screen中相应的函数指针指向了用于操作`drawable`的例程。pScreen字段指向这个drawable关联的screen。

pixmap结构定义如下：

```
typedef struct _Pixmap {
    DrawableRec         drawable;
    int                 refcnt;
    int                 devKind;
    DevUnion            devPrivate;
#ifdef COMPOSITE
    short               screen_x;
    short               screen_y;
#endif
} PixmapRec;
```

通过`drawable`字段，pixmap成了drawable的子类，这是C中实现类继承的常用方法。

Colormap

Graphics Contexts

Font

Cursor

2.2 其他数据结构

Client

每一个连接服务器的客户程序对应于一个Client数据结构，它定义于include/dixstruct.h:

```
typedef struct _Client {
    int         index;
    Mask        clientAsMask;
    pointer     requestBuffer;
    pointer     osPrivate;        /* for OS layer, including scheduler */
    Bool        swapped;
    ReplySwapPtr pSwapReplyFunc;
    XID         errorValue;
    int         sequence;
    int         closeDownMode;
    int         clientGone;
    int         noClientException;      /* this client died or needs to be
                                         * killed */
    DrawablePtr lastDrawable;
    Drawable    lastDrawableID;
    GCPtr       lastGC;
    GContext    lastGCID;
    SaveSetElt        *saveSet;
    int         numSaved;
    pointer     screenPrivate[MAXSCREENS];
    int         (**requestVector) (
                   ClientPtr /* pClient */);
    CARD32        req_len;                /* length of current request */
    Bool        big_requests;                /* supports large requests */
    int                priority;
    ClientState clientState;
    DevUnion        *devPrivates;

    unsigned long replyBytesRemaining;

    struct _FontResolution * (*fontResFunc) (    /* no need for font.h */
                ClientPtr           /* pClient */,
                int *               /* num */);

} ClientRec;
```

requestBuffer用于放置来自客户的请求。osPrivate 用于保存os层需要使用的私有数据，这在OS层一章有所涉及。

2.3 资源管理

资源管理模块负责管理所有的资源(dix/resource.c)，它定义了以下的结构和宏：

```
#define SERVER_MINID 32

#define INITBUCKETS 64
#define INITHASHSIZE 6
#define MAXHASHSIZE 11

#define NullResource ((ResourcePtr)NULL)

typedef struct _Resource {
    struct _Resource *next;
    XID                 id;
    RESTYPE             type;
    pointer             value;
} ResourceRec, *ResourcePtr;

typedef struct _ClientResource {
    ResourcePtr *resources;
    int                elements;
    int                buckets;
    int                hashsize;        /* log(2)(buckets) */
    XID                fakeID;
    XID                endFakeID;
    XID                expectID;
} ClientResourceRec;

RESTYPE lastResourceType;
static RESTYPE lastResourceClass;
RESTYPE TypeMask;

static DeleteType *DeleteFuncs = (DeleteType *)NULL;
```

每种资源都属于一个类，X中定义了如下的资源类和类型：

```
#define RC_VANILLA        ((RESTYPE)0)
#define RC_CACHED        ((RESTYPE)1<<31)
#define RC_DRAWABLE        ((RESTYPE)1<<30)
/*  Use class RC_NEVERRETAIN for resources that should not be retained
 *  regardless of the close down mode when the client dies.  (A client's
 *  event selections on objects that it doesn't own are good candidates.)
 *  Extensions can use this too!
 */
#define RC_NEVERRETAIN        ((RESTYPE)1<<29)
#define RC_LASTPREDEF        RC_NEVERRETAIN
#define RC_ANY                (~(RESTYPE)0)

#define RT_WINDOW        ((RESTYPE)1|RC_CACHED|RC_DRAWABLE)
#define RT_PIXMAP        ((RESTYPE)2|RC_CACHED|RC_DRAWABLE)
#define RT_GC                ((RESTYPE)3|RC_CACHED)
#undef RT_FONT
#undef RT_CURSOR
#define RT_FONT                ((RESTYPE)4)
```

```
#define RT_CURSOR        ((RESTYPE)5)
#define RT_COLORMAP        ((RESTYPE)6)
#define RT_CMAPENTRY        ((RESTYPE)7)
#define RT_OTHERCLIENT        ((RESTYPE)8|RC_NEVERRETAIN)
#define RT_PASSIVEGRAB        ((RESTYPE)9|RC_NEVERRETAIN)
#define RT_LASTPREDEF        ((RESTYPE)9)
#define RT_NONE                ((RESTYPE)0)
```

每个资源都有唯一的XID, 它实际上是一个int, 它的32个位被加以划分:

```
/* bits and fields within a resource id */
#define RESOURCE_AND_CLIENT_COUNT   29                          /* 29 bits for XIDs */
#if MAXCLIENTS == 512
#define RESOURCE_CLIENT_BITS        9
#endif
/* client field offset */
#define CLIENTOFFSET            (RESOURCE_AND_CLIENT_COUNT - RESOURCE_CLIENT_BITS)
/* resource field */
#define RESOURCE_ID_MASK        ((1 << CLIENTOFFSET) - 1)
#define RESOURCE_CLIENT_MASK        (((1 << RESOURCE_CLIENT_BITS) - 1) << CLIENTOFFSET)

/* extract the client mask from an XID */
#define CLIENT_BITS(id) ((id) & RESOURCE_CLIENT_MASK)
/* extract the client id from an XID */
#define CLIENT_ID(id) ((int)(CLIENT_BITS(id) >> CLIENTOFFSET))
#define SERVER_BIT                (Mask)0x40000000        /* use illegal bit */
```

DeleteFuncs函数指针集合指向了每种类型资源被delete的时候所要调用的处理函数。每个客户所使用的资源用一个全局的数组加以记录:

```
    ClientResourceRec clientTable[MAXCLIENTS];
```

在每个client结构建立的时候，它对应的资源结构也被设置:

```
Bool
InitClientResources(ClientPtr client)
{
    register int i, j;

    if (client == serverClient)
    {
        lastResourceType = RT_LASTPREDEF;
        lastResourceClass = RC_LASTPREDEF;
        TypeMask = RC_LASTPREDEF - 1;
        if (DeleteFuncs)
            xfree(DeleteFuncs);
        DeleteFuncs = (DeleteType *)xalloc((lastResourceType + 1) *
                                           sizeof(DeleteType));
        if (!DeleteFuncs)
            return FALSE;
        DeleteFuncs[RT_NONE & TypeMask] = (DeleteType)NoopDDA;
        DeleteFuncs[RT_WINDOW & TypeMask] = DeleteWindow;
        DeleteFuncs[RT_PIXMAP & TypeMask] = dixDestroyPixmap;
        DeleteFuncs[RT_GC & TypeMask] = FreeGC;
        DeleteFuncs[RT_FONT & TypeMask] = CloseFont;
        DeleteFuncs[RT_CURSOR & TypeMask] = FreeCursor;
        DeleteFuncs[RT_COLORMAP & TypeMask] = FreeColormap;
```

```
        DeleteFuncs[RT_CMAPENTRY & TypeMask] = FreeClientPixels;
        DeleteFuncs[RT_OTHERCLIENT & TypeMask] = OtherClientGone;
        DeleteFuncs[RT_PASSIVEGRAB & TypeMask] = DeletePassiveGrab;
    }
    clientTable[i = client->index].resources =
        (ResourcePtr *)xalloc(INITBUCKETS*sizeof(ResourcePtr));
    if (!clientTable[i].resources)
        return FALSE;
    clientTable[i].buckets = INITBUCKETS;
    clientTable[i].elements = 0;
    clientTable[i].hashsize = INITHASHSIZE;
    /* Many IDs allocated from the server client are visible to clients,
     * so we don't use the SERVER_BIT for them, but we have to start
     * past the magic value constants used in the protocol.  For normal
     * clients, we can start from zero, with SERVER_BIT set.
     */
    clientTable[i].fakeID = client->clientAsMask |
                            (client->index ? SERVER_BIT : SERVER_MINID);
    clientTable[i].endFakeID = (clientTable[i].fakeID | RESOURCE_ID_MASK) + 1;
    clientTable[i].expectID = client->clientAsMask;
    for (j=0; j<INITBUCKETS; j++)
    {
        clientTable[i].resources[j] = NullResource;
    }
    return TRUE;
}
```

serverClient是第一个被创建的client，在它初始化的时候，DeleteFunc数组被加以设置。

2.4 资源的组织

由全局的变量(dix/globals.c)不难看出全局资源的组织情况:

```
ScreenInfo screenInfo;
ClientPtr *clients;
ClientPtr  serverClient;
int  currentMaxClients;   /* current size of clients array */
WindowPtr *WindowTable;
```

clients数组保存了所有客户的信息，它默认的大小是500。clients[0]是特殊的元素，它对应于根窗口，并且有:

```
serverClient = clients[0];
```

3 DIX层

设备无关层(DIX)搭建了服务器的框架。它构造全局数据结构，调用DDX层和OS层为相应的数据结构设置对应的回调函数。在完成初始化后，DIX层进入dispatch事件循环，读取和响应客户请求，并在适当的时候通过回调函数调用DDX层和OS层。我们需要解决的问题包括：

- 初始化的过程中，全局的数据结构如何被设置？设置后呈怎样的图像？
- DDX层相关的部分如何被绑定到DIX层？DDX和DIX的接口在哪，包含什么？
- Dispatch循环如何响应客户的请求？

3.1 main函数

我们先从main函数出发，它定义在dix/main.c中：

```
int
main(int argc, char *argv[], char *envp[])
{
    int         i, j, k, error;
    char        *xauthfile;
    HWEventQueueType        alwaysCheckForInput[2];

    display = "0";

    InitGlobals();
    CheckUserParameters(argc, argv, envp);
    CheckUserAuthorization();
    InitConnectionLimits();

    argcGlobal = argc;
    argvGlobal = argv;
    /* prep X authority file from environment; this can be overriden by a
    * command line option */
    xauthfile = getenv("XAUTHORITY");
    if (xauthfile) InitAuthorization (xauthfile);
    ProcessCommandLine(argc, argv);

    alwaysCheckForInput[0] = 0;
    alwaysCheckForInput[1] = 1;

    while (1) {
        ...
    }
    return 0;
}
```

InitGlobals()的意图是给os层和ddx层提供初始化全局变量的机会。实际上现在只提供了空实现。InitGlobals()定义在os/util.c中：

```
void InitGlobals(void)
{
    ddxInitGlobals();
}
```

ddxInitGlobals()定义于(hw/kdrive/src/kdrive.c):

```
void ddxInitGlobals(void) { /* THANK YOU XPRINT */ }
```

接下来我们看看InitConnectionLimits()函数。在X中，每个client的连接对应于一个文件描述符，而所有client被组织在一个数组里，全局的整型数组ConnectionTranslation定义了从描述符到clients数组索引的转换表。InitConnectionLimits()定义于os/connection.c, 它初始化ConnectionTranslation数组，由于是os相关，实际的函数含有许多的宏，简化如下:

```
/* Set MaxClients and lastfdesc, and allocate ConnectionTranslation */
void
InitConnectionLimits(void)
{
    lastfdesc = -1;

    lastfdesc = sysconf(_SC_OPEN_MAX) - 1;
    /* This is the fallback */
    if (lastfdesc < 0)
        lastfdesc = MAXSOCKS;

    if (lastfdesc > MAXSELECT)
        lastfdesc = MAXSELECT;

    if (lastfdesc > MAXCLIENTS)
    {
        lastfdesc = MAXCLIENTS;
    }
    MaxClients = lastfdesc;

#if !defined(WIN32)
    ConnectionTranslation = (int *)xnfalloc(sizeof(int)*(lastfdesc + 1));
#else
    InitConnectionTranslation();
#endif
}
```

ConnectionTranslation在同一文件中定义为：

```
int *ConnectionTranslation = NULL;
```

另外，该文件定义了如下的宏：

```
#undef MAXSOCKS
#define MAXSOCKS 500
#undef MAXSELECT
#define MAXSELECT 500
#define MAXFD 500
```

下面我们展开while循环:

```
main
{
    ...
    while(1)
    {
        serverGeneration++;
        ScreenSaverTime = defaultScreenSaverTime;
        ScreenSaverInterval = defaultScreenSaverInterval;
```

```
        ScreenSaverBlanking = defaultScreenSaverBlanking;
        ScreenSaverAllowExposures = defaultScreenSaverAllowExposures;
        InitBlockAndWakeupHandlers();

        ...
        Dispatch();
        ...

        xfree(ConnectionInfo);
        ConnectionInfo = NULL;
    }
    return(0);
}
```

循环的第一步是完成初始化工作，第二步是进入Dispatch()事件处理循环，第三步是释放所有的资源。这个大循环的退出即意味着main函数的退出。初始化的内容包括:

初始化内容	函数	实现于
Block, Wakeup Handler	InitBlockAndWakeupHandlers	DIX
OS	OsInit()	OS
Socket	CreateWellKnownSockets()	OS
事件处理函数数组	InitProcVectors	DIX
client数组		main
screenInfo,WindowTable		main
Atom	InitAtom()	DIX
	InitGlyphCaching()	
private		DIX
通用Callback管理	InitCallbackManager()	DIX
	InitVisualWrap()	
输出	InitOutput()	DDX
扩展	InitExtensions()	
输入	InitInput()	DDX
字体	InitFonts()	
根窗口	InitRootWindow()	DIX

3.1.1 初始化中的各层交互

DIX、OS、kdrive和ephyr四层相互独立，但又相互交互。DIX构建服务器的框架，其他各层通过注册回调函数、设置DIX层中相应数据结构的函数指针实现与它的配合。各层交互的初始化工作主要在OsInit()、InitOutput()、InitInput() 三个函数中完成。图2对这三个函数的调用链给出了描述。

3.1.2 InitExtensions

InitExtensions()在mi/miinitext.c中定义，定义了两个版本，通过宏来选择，下面简要的展示其中的一个定义:

```
#ifndef XFree86LOADER

/*ARGSUSED*/
void
InitExtensions(argc, argv)
    int         argc;
```

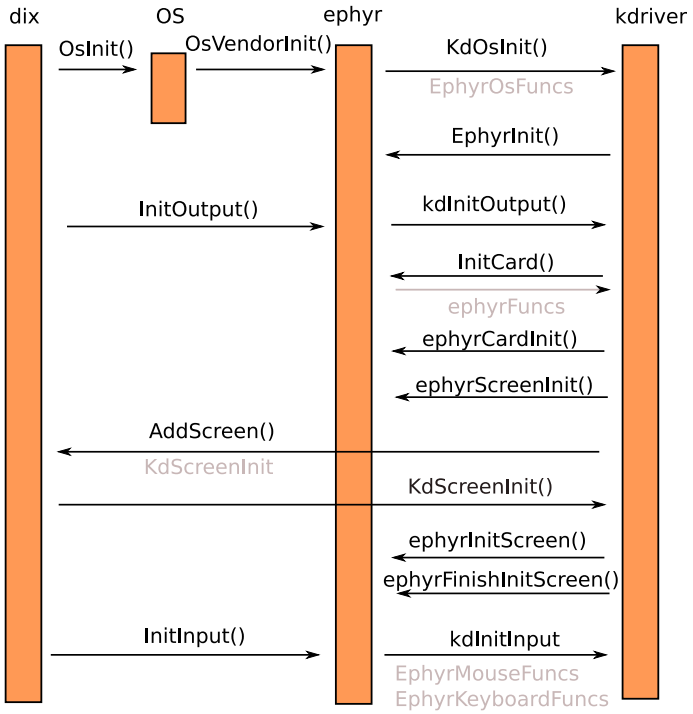

Figure 2: 四层初始化交互过程

```
    char        *argv[];
{
#ifdef XCSECURITY
    SecurityExtensionSetup();
#endif
    ...
#ifdef DMXEXT
    DMXExtensionInit(); /* server-specific extension, cannot be disabled */
#endif
#ifdef XEVIE
    if (!noXevieExtension) XevieExtensionInit();
#endif
#ifdef COMPOSITE
    if (!noCompositeExtension) CompositeExtensionInit();
#endif
#ifdef DAMAGE
    if (!noDamageExtension) DamageExtensionInit();
#endif
}
```

在这里我们看到实现3D桌面效果必须的DamageExtension、CompositeExtension 两个扩展的初始化位置。

3.2 Dispatch事件循环

下面首先给出Dispatch事件循环经过简化后的代码:

```
#define MAJOROP ((xReq *)client->requestBuffer)->reqType
```

```
void
Dispatch(void)
{
    register int        *clientReady;     /* array of request ready clients */
    register int        result;
    register ClientPtr  client;
    register int        nready;
    register HWEventQueuePtr* icheck = checkForInput;

    nextFreeClientID = 1;
    InitSelections();
    nClients = 0;

    clientReady = (int *) ALLOCATE_LOCAL(sizeof(int) * MaxClients);
    if (!clientReady)
        return;

    while (!dispatchException)
    {
        if (*icheck[0] != *icheck[1])
        {
            ProcessInputEvents();
            FlushIfCriticalOutputPending();
        }

        nready = WaitForSomething(clientReady);

       /*****************
        *  Handle events in round robin fashion, doing input between
        *  each round
        *****************/
        while (!dispatchException && (--nready >= 0))
        {
            client = clients[clientReady[nready]];
            if (! client)
            {
                /* KillClient can cause this to happen */
                continue;
            }
            /* GrabServer activation can cause this to be true */
            if (grabState == GrabKickout)
            {
                grabState = GrabActive;
                break;
            }
            isItTimeToYield = FALSE;

            requestingClient = client;
            while (!isItTimeToYield)
            {
                if (*icheck[0] != *icheck[1])
                {
                    ProcessInputEvents();
```

```
                    FlushIfCriticalOutputPending();
                }
                /* now, finally, deal with client requests */
                result = ReadRequestFromClient(client);
                if (result <= 0)
                {
                    if (result < 0)
                        CloseDownClient(client);
                    break;
                }
                client->sequence++;
                result = (* client->requestVector[MAJOROP])(client);
            }
            FlushAllOutput();
            requestingClient = NULL;
        }
        dispatchException &= ~DE_PRIORITYCHANGE;
    }
    KillAllClients();
    DEALLOCATE_LOCAL(clientReady);
    dispatchException &= ~DE_RESET;
}
```

clientReady是一个int数组，它做为参数传给`WaitForSomething()`, 返回的结果中包含了等待处理的客户索引。之后的循环依次处理每个客户的请求。首先`ReadRequestFromClient()` 从客户读入请求，放置于client的requestBuffer，然后通过语句

```
result = (* client->requestVector[MAJOROP])(client);
```

调用相应的函数处理这一请求。

4 OS 层

OS层负责管理客户的连接和调度工作，同时它也提供了字体文件，字体名和文件名的翻译，另外，它还提供了底层的内存管理。

定义于os/osinit.c的OsInit()函数负责完成OS相关的初始化工作。

4.1 调度和请求交付

DIX的主事件循环营造了不同窗口间多任务的假象，实际上服务器是单进程的。事件循环把对每个客户端的工作分解成多个部分。不同客户的事件和请求互相交叉，所以真正的多任务是没有必要的。主事件循环调用WaitForSomething()得到事件的来源。WaitForSomething()函数阻塞服务器进程直到下面的事之一发生：

- 用户或硬件触发了一个输入事件(参考SetInputCheck()).
- 有未被处理的请求
- 有新的客户连接请求

在WaitForSomething()开始调用select、poll或其他类似的功能之前，它必须检查在工作队列上是否有东西，如果有，它必须调用DIX层的ProcessWorkQueue()过程。如果WaitForSomething()决定它要做一些可能会导致度阻塞的事情(在select或poll前)，它必须调用DIX的BlockHandler()。WaitForSomething()代码由两个部分组成，第一部分包含在while语句中，该部分代码的核心是select语句；第二部分从select返回的fd_set中计算出等待处理的client索引，并放置于pClientReady中，函数范围一个intclient的数目(os/WaitFor.c):

```
int
WaitForSomething(int *pClientsReady)
{
    int i;
    struct timeval waittime, *wt;
    INT32 timeout = 0;
    fd_set clientsReadable;
    fd_set clientsWritable;
    int curclient;
    int selecterr;
    int nready;
    fd_set devicesReadable;
    CARD32 now = 0;

    FD_ZERO(&clientsReadable);

    /* We need a while loop here to handle
       crashed connections and the screen saver timeout */
    while (1)
    {
        ... // select related code
    }
    ...

    nready = 0;
    if (XFD_ANYSET (&clientsReadable))
    {
        for (i=0; i<howmany(XFD_SETSIZE, NFDBITS); i++) {
```

```
            int highest_priority = 0;

            while (clientsReadable.fds_bits[i]) {
                int client_priority, client_index;

                curclient = ffs (clientsReadable.fds_bits[i]) - 1;
                client_index = /* raphael: modified */
                    ConnectionTranslation[curclient + (i * (sizeof(fd_mask) * 8))];
                pClientsReady[nready++] = client_index;
                clientsReadable.fds_bits[i] &= ~(((fd_mask)1L) << curclient);
            }
        }
    }
    return nready;
}
```

while代码比较长，我们看看select调用前的部分:

```
int
WaitForSomething(int *pClientsReady)
{
    while (1)
    {
        /* deal with any blocked jobs */
        if (workQueue)
            ProcessWorkQueue();
        if (XFD_ANYSET (&ClientsWithInput)) {
             XFD_COPYSET (&ClientsWithInput, &clientsReadable);
             break;
        }
        wt = NULL;
        if (timers) {
            now = GetTimeInMillis();
            timeout = timers->expires - now;
            if (timeout < 0)
                timeout = 0;
            waittime.tv_sec = timeout / MILLI_PER_SECOND;
            waittime.tv_usec = (timeout % MILLI_PER_SECOND) *
                               (1000000 / MILLI_PER_SECOND);
            wt = &waittime;
        }
        XFD_COPYSET(&AllSockets, &LastSelectMask);
        BlockHandler((pointer)&wt, (pointer)&LastSelectMask);
        if (NewOutputPending)
            FlushAllOutput();
        /* keep this check close to select() call to minimize race */
        if (dispatchException)
            i = -1;
        else if (AnyClientsWriteBlocked) {
            XFD_COPYSET(&ClientsWriteBlocked, &clientsWritable);
            i = Select (MaxClients, &LastSelectMask, &clientsWritable, NULL, wt);
        } else {
            i = Select (MaxClients, &LastSelectMask, NULL, NULL, wt);
        }
```

```
        selecterr = GetErrno();
        ...
        }
    }

    nready = 0;
    ...
}
```

workQueue在后面的章节介绍。ClientsWithInput定义于os/connection.c:

```
    fd_set ClientsWithInput;    /* clients with FULL requests in buffer */
```

从注释上来看，它记录了那些请求队列已经满了的客户。如果有这样的客户，则break,直接进入第二部分。接下去调用BlockHandler()以调用那些已经注册的需要在select()阻塞前被调用的例程。

select后LastSelectMask记录下了所有可读的描述符，clientsWritable记录了所有可写的描述符，下面看看select()后的代码:

```
int
WaitForSomething(int *pClientsReady)
{
    while (1)
    {
        ...
        selecterr = GetErrno();
        WakeupHandler(i, (pointer)&LastSelectMask);
        if (i <= 0) /* An error or timeout occurred */
        {
            ...
        } else {
            fd_set tmp_set;

            if (*checkForInput[0] == *checkForInput[1]) {
                if (timers) {
                    int expired = 0;
                    now = GetTimeInMillis();
                    if ((int) (timers->expires - now) <= 0)
                        expired = 1;

                    while (timers && (int) (timers->expires - now) <= 0)
                        DoTimer(timers, now, &timers);

                    if (expired)
                        return 0;
                }
            } if (AnyClientsWriteBlocked && XFD_ANYSET (&clientsWritable)) {
                NewOutputPending = TRUE;
                XFD_ORSET(&OutputPending, &clientsWritable, &OutputPending);
                XFD_UNSET(&ClientsWriteBlocked, &clientsWritable);
                if (! XFD_ANYSET(&ClientsWriteBlocked))
                    AnyClientsWriteBlocked = FALSE;
            }

            XFD_ANDSET(&devicesReadable, &LastSelectMask, &EnabledDevices);
```

```
            XFD_ANDSET(&clientsReadable, &LastSelectMask, &AllClients);
            XFD_ANDSET(&tmp_set, &LastSelectMask, &WellKnownConnections);
            if (XFD_ANYSET(&tmp_set))
                QueueWorkProc(EstablishNewConnections, NULL,
                    (pointer)&LastSelectMask);
            if (XFD_ANYSET (&devicesReadable) || XFD_ANYSET (&clientsReadable))
                break;
        }
    }
    ...
}
```

在理解上面的代码之前，我们看看下面的定义和注释：

```
fd_set EnabledDevices;              /* mask for input devices that are on */
fd_set AllSockets;              /* select on this */
fd_set AllClients;              /* available clients */
fd_set LastSelectMask;              /* mask returned from last select call */
fd_set ClientsWithInput;        /* clients with FULL requests in buffer */
fd_set ClientsWriteBlocked;        /* clients who cannot receive output */
fd_set OutputPending;              /* clients with reply/event data ready to go */
int MaxClients = 0;
Bool NewOutputPending;              /* not yet attempted to write some new output */
Bool AnyClientsWriteBlocked;        /* true if some client blocked on write */
```

所以在上面的代码处理后，devicesReadable记录下了所有可读的设备，clientsReadable记录下了所有可读的客户连接。

4.2 新客户连接的建立

对新客户的处理起始于WaitForSomething()函数发现在监听端口上的描述符被设置:

```
            if (XFD_ANYSET(&tmp_set))
                QueueWorkProc(EstablishNewConnections, NULL,
                    (pointer)&LastSelectMask);
            if (XFD_ANYSET (&devicesReadable) || XFD_ANYSET (&clientsReadable))
                break;
```

X服务器将EstablishNewConnections放置于WorkQueue上，这一函数在下次调用WaitForSomething()的时候被执行。

EstablishNewConnections()首先去掉所有游荡的客户，然后从select返回的fd集合中计算出有连接请求的fd记录于curconn中。XtransConnInfo结构定义于Xtrans库中，这一个库对网络传输的处理进行了包裹，XtransConnInfo 保存了网络传输需要的信息和函数指针[2]。trans_conn保存了监听端口的信息，X服务器调用_XSERVTransAccept() 完成新的连接，连接信息保存在new_trans_conn中:

```
Bool
EstablishNewConnections(ClientPtr clientUnused, pointer closure)
{
    fd_set  readyconnections;     /* set of listeners that are ready */
    int curconn;                  /* fd of listener that's ready */
    register int newconn;         /* fd of new client */
    CARD32 connect_time;
    register int i;
    register ClientPtr client;
```

[2]Xtrans库的文档可以参见doc/xorg-docs/hardcopy/xtrans/Xtrans.PS.gz

```
    register OsCommPtr oc;
    fd_set tmask;

    XFD_ANDSET (&tmask, (fd_set*)closure, &WellKnownConnections);
    XFD_COPYSET(&tmask, &readyconnections);
    if (!XFD_ANYSET(&readyconnections))
        return TRUE;
    connect_time = GetTimeInMillis();
    /* kill off stragglers */
    for (i=1; i<currentMaxClients; i++)
    {
        if ((client = clients[i]))
        {
            oc = (OsCommPtr)(client->osPrivate);
            if ((oc && (oc->conn_time != 0) &&
                (connect_time - oc->conn_time) >= TimeOutValue) ||
                (client->noClientException != Success && !client->clientGone))
                    CloseDownClient(client);
        }
    }

    for (i = 0; i < howmany(XFD_SETSIZE, NFDBITS); i++)
    {
      while (readyconnections.fds_bits[i])
      {
        XtransConnInfo trans_conn, new_trans_conn;
        int status;

        curconn = ffs (readyconnections.fds_bits[i]) - 1;
        readyconnections.fds_bits[i] &= ~((fd_mask)1 << curconn);
        curconn += (i * (sizeof(fd_mask)*8));
        if ((trans_conn = lookup_trans_conn (curconn)) == NULL)
            continue;
        if ((new_trans_conn = _XSERVTransAccept (trans_conn, &status)) == NULL)
            continue;

        newconn = _XSERVTransGetConnectionNumber (new_trans_conn);
        _XSERVTransSetOption(new_trans_conn, TRANS_NONBLOCKING, 1);
        AllocNewConnection (new_trans_conn, newconn, connect_time)
      }
    }
    return TRUE;
}
```

EstablishNewConnections调用AllocNewConnection()分配并设置新的client结构，同时设置ConnectionTranslation转换表，这一转换表从fd找到client的index:

```
static ClientPtr
AllocNewConnection (XtransConnInfo trans_conn, int fd, CARD32 conn_time)
{
    OsCommPtr        oc;
    ClientPtr        client;

    oc = (OsCommPtr)xalloc(sizeof(OsCommRec));
```

```
    oc->trans_conn = trans_conn;
    oc->fd = fd;
    oc->input = (ConnectionInputPtr)NULL;
    oc->output = (ConnectionOutputPtr)NULL;
    oc->auth_id = None;
    oc->conn_time = conn_time;
    if (!(client = NextAvailableClient((pointer)oc))) {
        xfree (oc);
        return NullClient;
    }
    ConnectionTranslation[fd] = client->index;

    if (GrabInProgress) {
        FD_SET(fd, &SavedAllClients);
        FD_SET(fd, &SavedAllSockets);
    } else {
        FD_SET(fd, &AllClients);
        FD_SET(fd, &AllSockets);
    }

    return client;
}
```

在前面我们看到，X Server在全局数组`clients`中(2.4)维护所有的client信息。这一数组在初始化的时候被分以固定的大小(500)，**NextAvailableClient()**在这一数组中找下一个可用位置(`dix/dispatch.c`):

```
/************************
 * int NextAvailableClient(ospriv)
 *
 * OS dependent portion can't assign client id's because of CloseDownModes.
 * Returns NULL if there are no free clients.
 ************************/

ClientPtr NextAvailableClient(pointer ospriv)
{
    register int i;
    register ClientPtr client;
    xReq data;

    i = nextFreeClientID;
    if (i == MAXCLIENTS)
        return (ClientPtr)NULL;
    clients[i] = client = (ClientPtr)xalloc(totalClientSize);
    if (!client)
        return (ClientPtr)NULL;
    InitClient(client, i, ospriv);
    InitClientPrivates(client);
    if (!InitClientResources(client))
    {
        xfree(client);
        return (ClientPtr)NULL;
    }
    data.reqType = 1;
    data.length = (sz_xReq + sz_xConnClientPrefix) >> 2;
```

```
    if (!InsertFakeRequest(client, (char *)&data, sz_xReq))
    {
        FreeClientResources(client);
        xfree(client);
        return (ClientPtr)NULL;
    }
    if (i == currentMaxClients)
        currentMaxClients++;
    while ((nextFreeClientID < MAXCLIENTS) && clients[nextFreeClientID])
        nextFreeClientID++;
    if (ClientStateCallback)
    {
        NewClientInfoRec clientinfo;

        clientinfo.client = client;
        clientinfo.prefix = (xConnSetupPrefix *)NULL;
        clientinfo.setup = (xConnSetup *) NULL;
        CallCallbacks((&ClientStateCallback), (pointer)&clientinfo);
    }
    return(client);
}
```

4.3 BlockHandler

在WaitForSomething调用select之前，它调用BlockHandler()。

5 DDX 层

DDX层大部分的工作都在前面提过的函数指针所指向的函数中完成, 它们大部分和图形显示相关，其余的和处理用户从输入设备的输入。下面结合Xephyr分析DDX层。

5.1 数据结构

在kdrive中，结构KdCardInfo用于描述显卡, 结构KdScreenInfo用于描述屏幕，它们与DIX层的Display 和Screen相对应，它们保存了kdrive用于处理显卡和屏幕所需要的信息。KdCardAttr用于描述显卡的特性，KdFrameBuffer描述了一个屏幕对应的帧缓存。结构KdPrivScreenRec被加到DIX层Screen结构的Private字段中，用于连接DIX和kdrive中的数据结构。

```
typedef struct _KdCardAttr KdCardAttr;
typedef struct _KdCardInfo KdCardInfo;
typedef struct _KdFrameBuffer KdFrameBuffer;
typedef struct _KdScreenInfo KdScreenInfo;
struct  KdPrivScreenRec;

typedef struct _KdOffscreenArea KdOffscreenArea;
typedef void (*KdOffscreenSaveProc)
            (ScreenPtr pScreen, KdOffscreenArea *area);
typedef enum _KdOffscreenState KdOffscreenState;


KdCardInfo *kdCardInfo;
```

全局变量kdCardInfo用于组织kdrive中所有的数据。如图3所示，kdCardInfo 指向KdCardInfo的列表, 每个KdCardInfo和一个KdScreenInfo列表对应, 每个KdScreenInfo 含有一个KdFrameBuffer数组，一般来说这个数组的元素为2，对应于双缓冲。在ephyr中,KdFrameBuffer 指向XImage结构中的图像存储区。KdCardInfo和KdScreenInfo都在driver字段中保存了ephyr的私有数据。另外，KdCardInfo的cfunc字段指向ephyr提供的一个回调函数集合。 KdScreenInfo中的pscreen字段指向DIX层的screen结构，后者的私有字段指向KdPrivScreenRec。KdPrivScreenRec包含指向KdScreenInfo和KdCardInfo的指针，因而当DIX以指向screen的指针为参数调用kdrive例程的时候，kdrive可以获得对应的KdScreenInfo和KdCardInfo。

5.2 输入

DIX在一个叫做InputInfo的数据结构中保存所有输入设备的信息。对每个设备，都有一个对应的设备结构DeviceRec. DIX能够通过InputInfo定位每一个DeviceRec。另外，它有一个特殊的指针标明主鼠标设备，另一个特殊的指针标明主键盘。

输入设备结构DeviceRec(Xserver/include/input.h)是一个设备无关的结构，保存了输入设备的状态。DevicePtr是指向DeviceRec的指针。

xEvent用于描述服务器向客户报告的事件，它在Xproto.h中定义，包含一个struct的union，用于表示各种事件。

DDX主要的输入函数接口是

```
void ProcessInputEvents()
```

当有逗留的输入时, DIX会调用这个函数，在没有逗留的输入时，DIX 也可能调用这个函数。DDX必须完成这个函数，从每个设备得到输入事件，并把它交付给DIX, 交付的过程通过调用一下的函数：

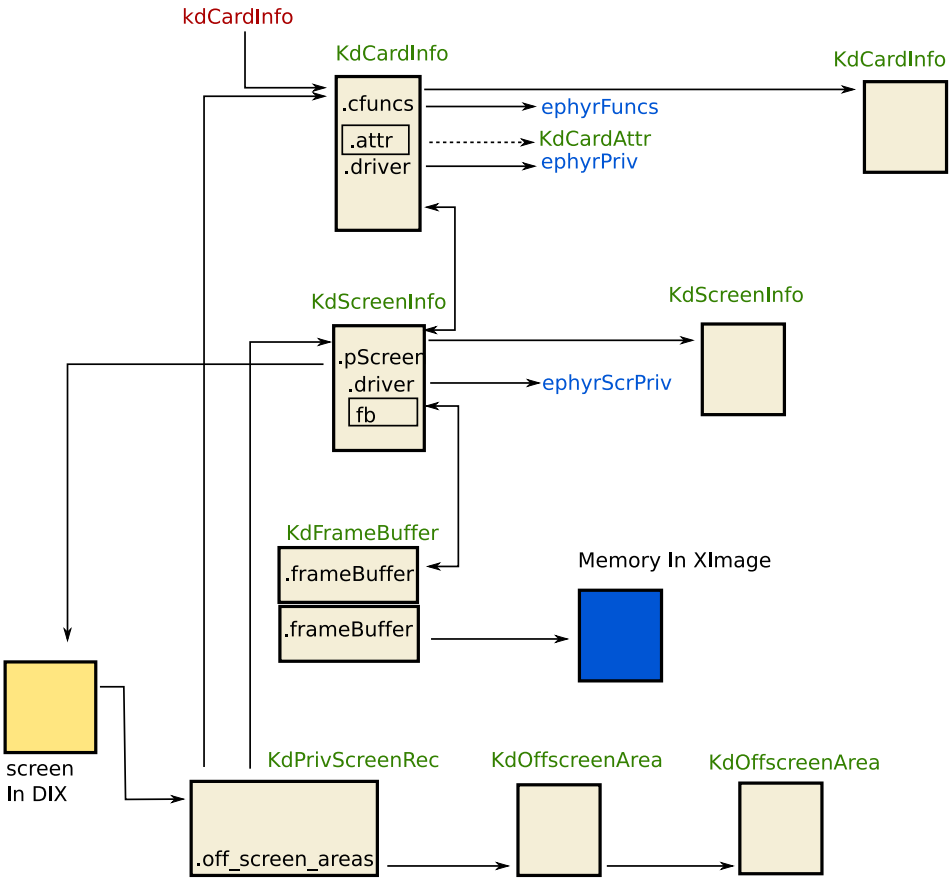

Figure 3: kdrive数据组织

```
void DevicePtr->processInputProc(xEventPtr pEvent,
    DeviceIntPtr device, int count);
```

在ProcessInputEvents()第一次被调用的时候，DIX会填充这个函数指针指向它自己的过程。有多少输入事件就掉用多少次这个函数，DIX会把这些事件缓存起来，并在必要的时候发送出去。

举个例子，我们的ProcessInputEvents()过程可能检查鼠标和键盘。如果键盘有多个击键在队列中，那我们需要多次的调用键盘的processInputProc直到把队列清空。

在服务器的主事件循环中，每当WaitForSomething()调用之前和返回的时候，DIX都要检查下是否有逗留的输入，如果有，DIX调用DDX的ProcessInputEvents()。

```
HWEventQueuePtr* icheck = checkForInput;

if (*icheck[0] != *icheck[1])
{
    ProcessInputEvents();
    FlushIfCriticalOutputPending();
}
```

检查的过程必须非常的快，它仅仅比较两个指针的内容。

```
typedef int                HWEventQueueType
typedef HWEventQueueType* HWEventQueuePtr;
HWEventQueuePtr checkForInput[2];
```

main函数中传给设置这两个指针, 这通过下面的函数完成：

```
    void SetInputCheck(long *p1, long *p2);

main()
{
    HWEventQueueType        alwaysCheckForInput[2];
    alwaysCheckForInput[0] = 0;
    alwaysCheckForInput[1] = 1;

    SetInputCheck(&alwaysCheckForInput[0], &alwaysCheckForInput[1]);
}
```

输入初始化

输入的初始化从InitInput()函数开始，它在DDX层实现，原型如下：

```
    void InitInput (int argc, char **argv);
```

在初始化中，我们需要两次调用AddInputDevice()，一次为鼠标，一次为键盘。这一函数返回指向输入设备的指针，我们需要以它们为参数调用RegisterKeyboardDevice()和RegisterPointerDevice()，分别标志主键盘和主鼠标。在调用AddInputDevice()的时候我们传递一个回调函数，它需要调用InitKeyboardDeviceStruct() 或InitPointerDeviceStruct()。AddInputDevice()原型如下：

```
    DeviceIntPtr AddInputDevice(DeviceProc deviceProc, Bool autoStart);
```

键盘映射和键码

一个键码(KeyCode)代表了一个物理按键，它的范围是[8, 255]。键和键码之间的映射由物理设备决定，不能被改变。键码对应的符号称为键符(KeySym)[3]。每个键码对应一个键符列表，当击键时，这个键码对应的键符由modifier键决定(比如shift)。

ephyr输入初始化

在ephyr代码中，InitInput()函数实现在文件ephyrinit.c中，它通过调用KdInitInput() 完成：

```
    void
    InitInput (int argc, char **argv)
    {
        KdInitInput (&EphyrMouseFuncs, &EphyrKeyboardFuncs);
    }
```

EphyrMouseFuncs，EphyrKeyboardFuncs定义在ephyr.c中：

```
    KdMouseFuncs EphyrMouseFuncs = {
        MouseInit,
        MouseFini,
    };

    KdKeyboardFuncs        EphyrKeyboardFuncs = {
        EphyrKeyboardLoad,
        EphyrKeyboardInit,
        EphyrKeyboardLeds,
```

[3] xlib 12章7节

```
    EphyrKeyboardBell,
    EphyrKeyboardFini,
    0,
};
```

除了EphyrKeyboardLoad(), 其他的函数只提供空实现：

```
static void
EphyrKeyboardLoad (void)
{
    EPHYR_DBG("mark");

    hostx_load_keymap();
}
```

hostx_load_keymap()定义在hostx.c中, 它从宿主机获得键码与键符的映射关系。

5.3 输出

InitOutput()函数为DIX层设置screenInfo、screen结构，为kdrive设置kdCardInfo, kdScreenInfo。它的调用链如图4所示。

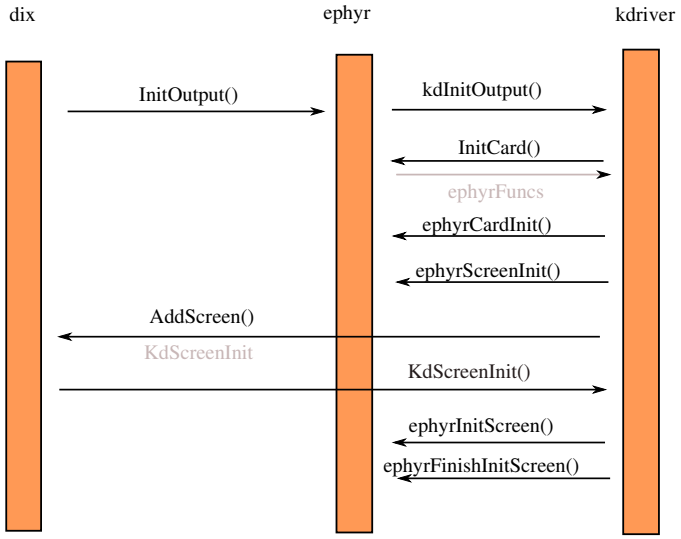

Figure 4: 输出初始化

InitOutput()hw/kdrive/ephyr/ephyrinit.c中定义。

```
void
InitOutput (ScreenInfo *pScreenInfo, int argc, char **argv)
{
    KdInitOutput (pScreenInfo, argc, argv);
}
```

KdInitOutput定义于kdrive.c:

```
void
KdInitOutput (ScreenInfo    *pScreenInfo,
              int     argc,
```

```
                char      **argv)
{
    KdCardInfo      *card;
    KdScreenInfo    *screen;

    if (!kdCardInfo)
    {
        InitCard (0);
        if (!(card = KdCardInfoLast ()))
            FatalError("No matching cards found!\n");
        screen = KdScreenInfoAdd (card);
        KdParseScreen (screen, 0);
    }
    /*
     * Initialize all of the screens for all of the cards
     */
    for (card = kdCardInfo; card; card = card->next)
    {
        int ret=1;
        if(card->cfuncs->cardinit)
            ret=(*card->cfuncs->cardinit) (card);
        if (ret)
        {
             for (screen = card->screenList; screen; screen = screen->next)
                KdInitScreen (pScreenInfo, screen, argc, argv);
        }
    }

    /*
     * Merge the various pixmap formats together, this can fail
     * when two screens share depth but not bitsPerPixel
     */
    if (!KdSetPixmapFormats (pScreenInfo))
        return;

    /*
     * Add all of the screens
     */
    for (card = kdCardInfo; card; card = card->next)
        for (screen = card->screenList; screen; screen = screen->next)
            KdAddScreen (pScreenInfo, screen, argc, argv);
}
```

InitCard函数在ephyrinit.c中定义:

```
    void
    InitCard (char *name)
    {
        KdCardAttr        attr;
        EPHYR_DBG("mark");
        KdCardInfoAdd (&ephyrFuncs, &attr, 0);
    }
```

我们看看ephyrFuncs:

```
KdCardFuncs ephyrFuncs = {
```

```
    ephyrCardInit,          /* cardinit */
    ephyrScreenInit,        /* scrinit */
    ephyrInitScreen,        /* initScreen */
    ephyrFinishInitScreen,  /* finishInitScreen */
    ephyrCreateResources,   /* createRes */
    ephyrPreserve,          /* preserve */
    ephyrEnable,            /* enable */
    ephyrDPMS,              /* dpms */
    ephyrDisable,           /* disable */
    ephyrRestore,           /* restore */
    ephyrScreenFini,        /* scrfini */
    ephyrCardFini,          /* cardfini */

    0,                      /* initCursor */
    0,                      /* enableCursor */
    0,                      /* disableCursor */
    0,                      /* finiCursor */
    0,                      /* recolorCursor */

    0,                      /* initAccel */
    0,                      /* enableAccel */
    0,                      /* disableAccel */
    0,                      /* finiAccel */

    ephyrGetColors,         /* getColors */
    ephyrPutColors,         /* putColors */
};
```

可以看出，ephyr通过向kdrive注册一系列的函数来实现它们的交互。KdCardInfoAdd添加了新的Card信息。之后是添加screen信息，KdScreenInfoAdd (card)。

接下去分析kdInitOutput的代码:

```
    /*
     * Initialize all of the screens for all of the cards
     */
    for (card = kdCardInfo; card; card = card->next)
    {
        int ret=1;
        if(card->cfuncs->cardinit)
            ret=(*card->cfuncs->cardinit) (card);
        if (ret)
        {
            for (screen = card->screenList; screen; screen = screen->next)
                KdInitScreen (pScreenInfo, screen, argc, argv);
        }
    }
```

(*card->cfuncs->cardinit) (card)调用了以下的函数:

```
Bool
ephyrCardInit (KdCardInfo *card)
{
  EphyrPriv        *priv;

  priv = (EphyrPriv *) xalloc (sizeof (EphyrPriv));
```

```
  if (!priv)
    return FALSE;

  if (!ephyrInitialize (card, priv))
    {
      xfree (priv);
      return FALSE;
    }
  card->driver = priv;

  return TRUE;
}
```

由于返回值为TRUE, KdInitScreen被调用:

```
void
KdInitScreen (ScreenInfo    *pScreenInfo,
              KdScreenInfo  *screen,
              int           argc,
              char        **argv)
{
    KdCardInfo        *card = screen->card;

    (*card->cfuncs->scrinit) (screen);

    if (!card->cfuncs->initAccel)
        screen->dumb = TRUE;
    if (!card->cfuncs->initCursor)
        screen->softCursor = TRUE;
}
```

看看ephyrScreenInit:

```
Bool
ephyrScreenInit (KdScreenInfo *screen)
{
  EphyrScrPriv *scrpriv;

  scrpriv = xalloc (sizeof (EphyrScrPriv));

  if (!scrpriv)
    return FALSE;

  memset (scrpriv, 0, sizeof (EphyrScrPriv));
  screen->driver = scrpriv;

  if (!ephyrScreenInitialize (screen, scrpriv))
    {
      screen->driver = 0;
      xfree (scrpriv);
      return FALSE;
    }

  return TRUE;
}
```

ephyrScreenInitialize函数比较长，我们知道Xephyr类似于Xnest,这个函数在另一个X server上建立一个窗口，并将它当作屏幕。

下面为剩下的kdInitOutput的代码:

```
/*
 * Merge the various pixmap formats together, this can fail
 * when two screens share depth but not bitsPerPixel
 */
if (!KdSetPixmapFormats (pScreenInfo))
    return;

/*
 * Add all of the screens
 */
for (card = kdCardInfo; card; card = card->next)
    for (screen = card->screenList; screen; screen = screen->next)
        KdAddScreen (pScreenInfo, screen, argc, argv);
```

5.4 ephyr

ephyr在已有的一个X服务器上模拟出另一个服务器。它在内存中创建XImage，并将XImage的存储区传给kdrive，虚拟出显卡的显存。它在宿主上创建一个窗口，截获窗口的鼠标和键盘事件并向kdrive传递，以此模拟鼠标和键盘。ephyr在什么时候、在什么地方捕获宿主的事件、在宿主上绘制结果?

我们知道服务器本身是单线程的，由于不可能为虚拟的鼠标和键盘创建对应的文件描述符，所以不可能在WaitForSomething()的select中等待输入事件; 实际上，ephyr采用了注册BlockHandler回调函数的方法，这一回调函数在select堵塞前被调用；ephyr不能堵塞在等待虚拟的鼠标和键盘事件中，否则无法处理客户的请求，所以它采用Poll的方法，首先检查宿主窗口有没有逗留事件，没有逗留事件的时候，它立即返回。ephyr向BlockHandler注册的另一个函数是ephyrInternalDamageBlockHandle(), 它将XImage中的内容显示到宿主窗口上。ephyr与其他层的关系如下：

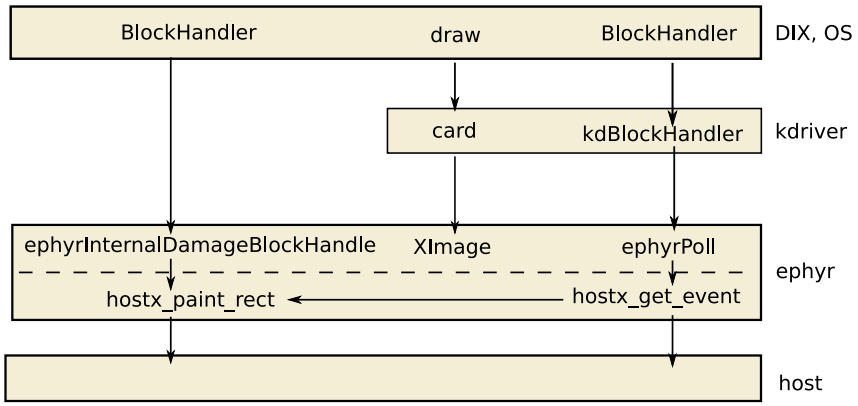

Figure 5: ephyr与其他层的关系

HostX

与宿主交互所需的所有信息被包含在HostX对象中(hostx.c)。

```
struct EphyrHostXVars
{
  char           *server_dpy_name;
  Display        *dpy;
  int             screen;
  Visual         *visual;
  Window          win, winroot;
  Window          win_pre_existing;   /* Set via -parent option like xnest */
  GC              gc;
  int             depth;
  int             server_depth;
  XImage         *ximg;
  int             win_width, win_height;
  Bool            use_host_cursor;
  Bool            use_fullscreen;
  Bool            have_shm;

  long            damage_debug_msec;

  unsigned char  *fb_data;           /* only used when host bpp != server bpp */
  unsigned long   cmap[256];

  XShmSegmentInfo shminfo;
};
```

各个字段的说明如下：

字段	说明
server_dpy_name	宿主机display描述(比如":1")
dpy	宿主机display
screen	宿主机屏幕号
win	在宿主机上建立的窗口(宿主窗口)
winroot	宿主机根窗口
win_pre_existing	父窗口，如果在创建的时候指定-parent选项
gc	在宿主窗口上绘制时使用的GC
depth	模拟服务器的颜色深度
server_depth	宿主机颜色深度
ximg	XImage数组，模拟显卡缓存。每个Image和宿主窗口的尺寸相同
use_host_cursor	使用宿主机的光标
use_fullscreen	全屏显示
have_shm	宿主机支持share memory

HostX的初始化设置由dix通过os层间接调用，main函数调用OsInit(), OsInit调用ephyr中的OsVendorInit(), 它设置鼠标回调函数，并将回调函数集EphyrOsFuncs传给kdOsInit。

```
void
OsVendorInit (void)
{
  if (hostx_want_host_cursor()) {
      ephyrFuncs.initCursor   = &ephyrCursorInit;
      ephyrFuncs.enableCursor = &ephyrCursorEnable;
  }
```

```
  KdOsInit (&EphyrOsFuncs);
}

KdOsFuncs   EphyrOsFuncs = {
    EphyrInit,
    EphyrEnable,
    EphyrSpecialKey,
    EphyrDisable,
    EphyrFini,
    ephyrPoll
};
```

除了ephyrPoll、EphyrInit, 其他几个函数只提供了空实现：

```
static int
EphyrInit (void)
{
  return hostx_init();
}
```

hostx_init()在宿主上创建窗口，完成对HostX的设置。

```
int hostx_init(void)
{
  XSetWindowAttributes  attr;
  Cursor                empty_cursor;
  Pixmap                cursor_pxm;
  XColor                col;

  attr.event_mask =
    ButtonPressMask
    |ButtonReleaseMask
    |PointerMotionMask
    |KeyPressMask
    |KeyReleaseMask
    |ExposureMask;

  if ((HostX.dpy = XOpenDisplay(getenv("DISPLAY"))) == NULL) {
    fprintf(stderr, "\nXephyr cannot open host display. Is DISPLAY set?\n");
    exit(1);
  }

  HostX.screen  = DefaultScreen(HostX.dpy);
  HostX.winroot = RootWindow(HostX.dpy, HostX.screen);
  HostX.gc      = XCreateGC(HostX.dpy, HostX.winroot, 0, NULL);
  HostX.depth   = DefaultDepth(HostX.dpy, HostX.screen);
  HostX.visual  = DefaultVisual(HostX.dpy, HostX.screen);

  HostX.server_depth = HostX.depth;


  HostX.win = XCreateWindow(HostX.dpy,
                            HostX.winroot,
                            0,0,100,100, /* will resize  */
                            0,
                            CopyFromParent,
```

```
                          CopyFromParent,
                          CopyFromParent,
                          CWEventMask,
                          &attr);

  hostx_set_win_title("( ctrl+shift grabs mouse and keyboard )");

  if (HostX.use_fullscreen) {
       HostX.win_width  = DisplayWidth(HostX.dpy, HostX.screen);
       HostX.win_height = DisplayHeight(HostX.dpy, HostX.screen);

       hostx_set_fullscreen_hint();
  }

  XParseColor(HostX.dpy, DefaultColormap(HostX.dpy,HostX.screen), "red", &col);
  XAllocColor(HostX.dpy, DefaultColormap(HostX.dpy, HostX.screen), &col);
  XSetForeground(HostX.dpy, HostX.gc, col.pixel);

  HostX.ximg   = NULL;

  /* Try to get share memory ximages for a little bit more speed */

  if (!XShmQueryExtension(HostX.dpy) || getenv("XEPHYR_NO_SHM")) {
      fprintf(stderr, "\nXephyr unable to use SHM XImages\n");
      HostX.have_shm = False;
  } else {
        ...
  }

  XFlush(HostX.dpy);

  /* Setup the pause time between paints when debugging updates */
  HostX.damage_debug_msec = 20000; /* 1/50 th of a second */
  return 1;
}
```

XParseColor函数得到绘制红色对应的rgb值，XAllocColor利用这个rgb获取绘制红色对应的pixel值(在debug中，ephyr在窗口外圈绘制红色框)。

6 其他

6.1 Wrappers 和devPrivates

6.1.1 devPrivates

devPrivates存在于许多数据结构之中(Screens, GCs, Windows和Pixmaps), 它是一个数组的数组，里层的数组包含了一组数值，它们可能是int、pointer也可能是结构，能被用作各种目的。每个数组在服务器初始化的时候动态分配大小，各个模块又分配了它们自己要使用的部分。

Screen devPrivates 我们看看ScreenRec结构中的devPrivates (include/screenint.h, include/miscstruc.h):

```
typedef struct _Screen {
    /* anybody can get a piece of this array */
    DevUnion        *devPrivates;
} ScreenRec;

typedef union _DevUnion {
    pointer             ptr;
    long                val;
    unsigned long       uval;
    RegionPtr           (*fptr)(
        DrawablePtr         /* pSrcDrawable */,
        DrawablePtr         /* pDstDrawable */,
        GCPtr               /* pGC */,
        int                 /* srcx */,
        int                 /* srcy */,
        int                 /* width */,
        int                 /* height */,
        int                 /* dstx */,
        int                 /* dsty */,
        unsigned long       /* bitPlane */);
} DevUnion;
```

我们调用AllocateScreenPrivateIndex()来分配一个新的devPrivates元素, devPrivates数组的大小会被加以调整以容纳新的元素。返回的值是新元素的下标, 我们可以使用如下:

```
    private = (PrivatePointer) pScreen->devPrivates[screenPrivateIndex].ptr;
```

当然上面的指针需要自己初始化。

Window devPrivates

GC和Pixmap devPrivates

6.1.2 Wrappers

6.2 工作队列WorkQueue

在一些情况下，我们需要设置X Server，使其在所有的客户都处于一个稳定的状态时(WaitForSomething()中调用select之前)调用一些例程。WorkQueue提供了这样的机制，它定义在dix/dixutil.c:

```
/*
 * A general work queue.  Perform some task before the server
 * sleeps for input.
 */
WorkQueuePtr            workQueue;
static WorkQueuePtr     *workQueueLast = &workQueue;
```

看看WorkQueuePtr(include/dixstruc.h):

```
typedef struct _WorkQueue        *WorkQueuePtr;

typedef struct _WorkQueue {
    struct _WorkQueue *next;
    Bool        (*function) (
                ClientPtr       /* pClient */,
                pointer         /* closure */
                );
    ClientPtr   client;
    pointer     closure;
} WorkQueueRec;
```

我们使用QueueWorkProc()往这一队列添加工作例程(dix/dixutils.c), 这一函数的实现简单易懂:

```
Bool
QueueWorkProc (
    Bool (*function)(ClientPtr /* pClient */, pointer /* closure */),
    ClientPtr client, pointer closure)
{
    WorkQueuePtr    q;

    q = (WorkQueuePtr) xalloc (sizeof *q);
    if (!q)
        return FALSE;
    q->function = function;
    q->client = client;
    q->closure = closure;
    q->next = NULL;
    *workQueueLast = q;
    workQueueLast = &q->next;
    return TRUE;
}
```

如果WorkQueue非空，WaitForSomething()函数在调用select之前，首先调用ProcessWorkQueue():

```
void
ProcessWorkQueue(void)
{
    WorkQueuePtr    q, *p;

    p = &workQueue;
    /*
     * Scan the work queue once, calling each function.  Those
     * which return TRUE are removed from the queue, otherwise
     * they will be called again.  This must be reentrant with
     * QueueWorkProc.
     */
```

```
    while ((q = *p))
    {
        if ((*q->function) (q->client, q->closure))
        {
            /* remove q from the list */
            *p = q->next;    /* don't fetch until after func called */
            xfree (q);
        }
        else
        {
            p = &q->next;    /* don't fetch until after func called */
        }
    }
    workQueueLast = p;
}
```