第一篇　服务器

先援引一段 RT-Thread 编程指南 GUI 部分的介绍作为开头

“RT-Thread/GUI 采用传统的客户端/服务端(C/S)的结构，但和传统的客户端/服务端构架，把绘画操作放于服务端不同的是，绘画操作完全有客户端自行完成。服务端仅维护着各个客户端的位置信息。”

在初始化线程中调用 rtgui_system_server_init()进行 rtgui 服务器端的初始化

```
void rtgui_system_server_init()
{
    rt_mutex_init(&_screen_lock, "screen", RT_IPC_FLAG_FIFO);
    rtgui_system_image_init();// 初始化图像系统　其中至少包含硬件 DC 的注册
    rtgui_font_system_init();  // 初始化字体系统
    /*从前面液晶的驱动程序得到 整个屏幕的大小参数 存在全局变量_mainwin_rect */
    rtgui_graphic_driver_get_rect(rtgui_graphic_driver_get_default(), &_mainwin_rect);
    rtgui_topwin_init();//主要是初始化一个双向链表_rtgui_topwin_list (见后面 topwin 部分)
    rtgui_server_init();
    rtgui_system_theme_init();/* init theme */
}
```

rtgui_server_init()是整个函数最重要的部分，它创建并启动了 rtgui 服务器线程，下面是服务器线程的入口:

```
static void rtgui_server_entry(void *parameter)
{
1   rtgui_server_application = rtgui_app_create(rtgui_server_tid,"rtgui");
2   rtgui_object_set_event_handler(RTGUI_OBJECT(rtgui_server_application),
                                   rtgui_server_event_handler);
3   rtgui_app_run(rtgui_server_application);
4   rtgui_app_destroy(rtgui_server_application);
5   rtgui_server_application = RT_NULL;
}
```

第 1 行首先创建 1 个 app 应用，这里有必要先了解一下全局指针 rtgui_server_application 指向的数据结构 struct rtgui_app

```
struct rtgui_app
{
    struct rtgui_object parent;      // 表示 rtgui_app 继承于 rtgui_object
    unsigned char *name;             // app 的名字
    rt_thread_t tid;                 // 需要绑定的线程 也即创建此 app 的线程
    rt_thread_t server;              // rtgui 服务器端的线程
    rt_mq_t mq;                      // 此线程的消息队列指针
    rt_uint8_t event_buffer[sizeof(union rtgui_event_generic)]; // 事件缓冲区
};
```

通过函数 rtgui_app_create，设置好 rtgui_server_application 的名字 name = "rtgui"，tid= rtgui_server_tid(即服务器线程)　并创建其消息队列。在函数中，还会将此线程(在这里

是服务器线程)的 user_data 指针指向此 app，所以其实从这里还可以看出，**一个线程只能创建 1 个 rtgui 应用**

第 2 行的作用是设置这个 app 的 OBJECT 级别的事件处理函数为 rtgui_server_event_handler，在那里面，根据接收到的事件 event 类型和参数调用不同的函数进行处理。

第 3 行启动这个 app，跟踪进去会发现该函数调用_rtgui_application_event_loop(app) 启动这个 app 的事件循环，再跟踪进去会发现此函数在**循环**调用事件接收函数(后面有例子说明)

```
{
        result = rtgui_recv(event, sizeof(union rtgui_event_generic));
        if (result == RT_EOK)
            RTGUI_OBJECT(app)->event_handler(RTGUI_OBJECT(app), event);
}
```

跟踪进 rtgui_recv 会看到

```
r = rt_mq_recv(app->mq, event, event_size, RT_WAITING_FOREVER);
```

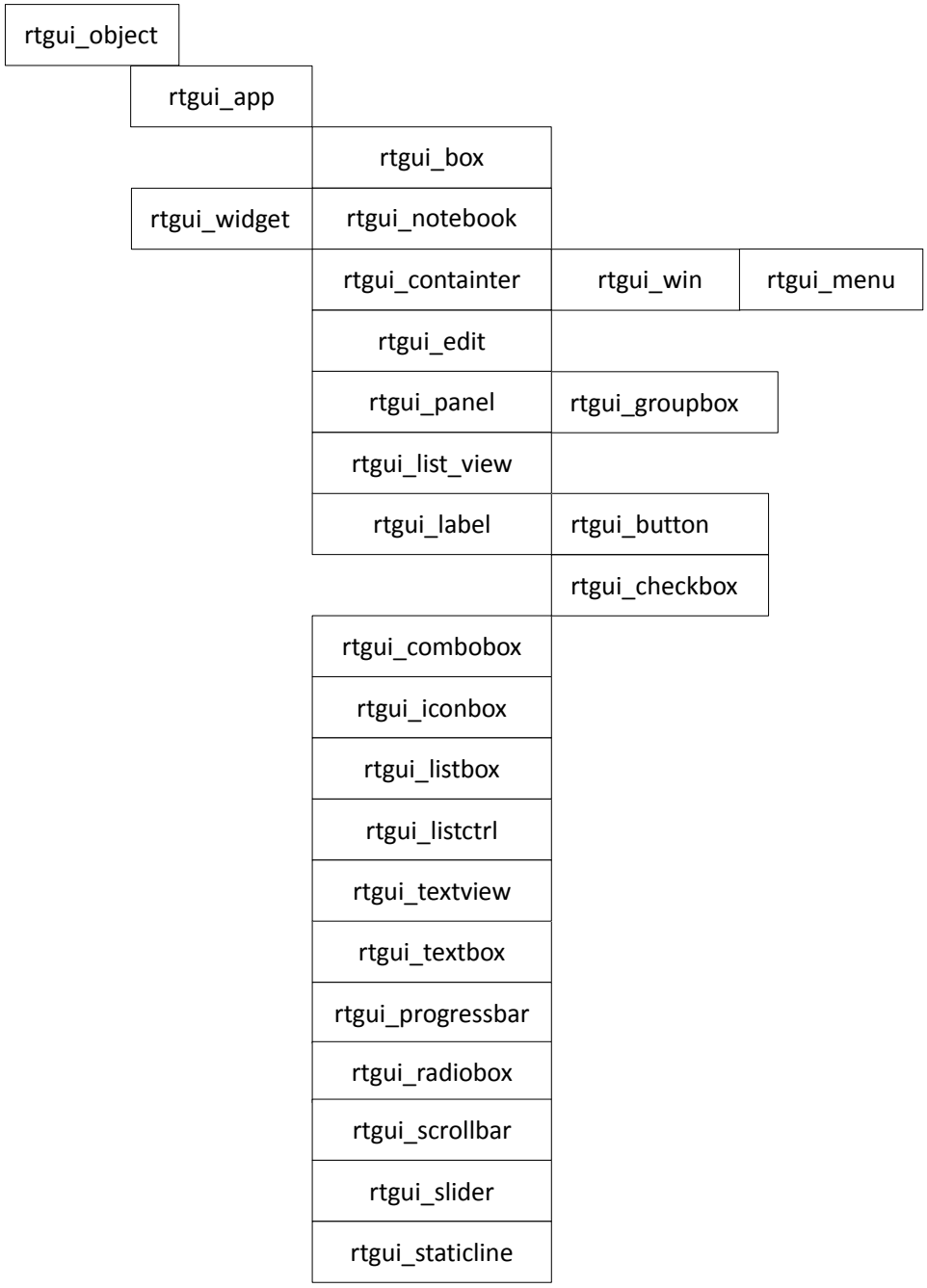
可见，线程会阻塞在这里，从消息队列中接收事件。

在接收到事件以后，调用 app 父类()的事件处理函数 即 rtgui_server_event_handler。

在 app 结束时，会执行第 4-5 行，销毁 app 所占用的内存资源。

第二篇　层次分明的数据结构

1. rtgui 实例对象



上图显示了以 rtgui_object 为 root 的一棵继承树，rtgui 中定义了许多宏用来进行子对象向父对象的指针转换，这一点在源码中几乎随处可见。

如在第一篇中出现的一行代码

```
rtgui_object_set_event_handler(RTGUI_OBJECT(rtgui_server_application),
                               rtgui_server_event_handler);
```

上面这条语句中，便是通过宏 RTGUI_OBJECT 将一个 rtgui_app 对象指针转换为一个 rt_object 指针，并设置其 object 级别的事件处理函数。

以下这个数据结构在 rtgui 中表示类型

```
struct rtgui_type
{
        char *name;                          //名字
        struct rtgui_type *parent;           //父类型
        rtgui_constructor_t constructor;     // 构造函数
        rtgui_destructor_t destructor;       // 析构函数
        int size;                            // 类型的大小  用于在创建对象时 申请内存
};
```

举个例子 比如我要创建 1 个 struct rtgui_win 对象，最后获得其指针

```
struct rtgui_win *win;
win = RTGUI_WIN(rtgui_widget_create(RTGUI_WIN_TYPE));
```

可以看出 要创建 rtgui_win 对象会先在 rtgui_widget 级别创建，再转换为 rtgui_win 类型

宏定义:

```
#define RTGUI_WIN_TYPE                  (RTGUI_TYPE(win))
#define RTGUI_TYPE(type)                (struct rtgui_type*)&(_rtgui_##type)
```

在宏定义中是连接符

所以 rtgui_widget_create(RTGUI_WIN_TYPE)就是

```
rtgui_widget_create((struct rtgui_type*)&(_rtgui_win))
```

而_rtgui_win 实际上是一个常量 定义在 window.c 中

```
const struct rtgui_type _rtgui_win =
{
    "win",
    RTGUI_CONTAINER_TYPE, // 实际上是常量结构体_rtgui_container 的指针
    _rtgui_win_constructor,
    _rtgui_win_destructor,
    sizeof(struct rtgui_win)
}
```

PS 在 rtgui 中 像这样的常量还有很多!

来看看 rtgui_widget_create 函数内部,注意此时输入参数是(struct rtgui_type*)&(_rtgui_win)

```
rtgui_widget_t *rtgui_widget_create(rtgui_type_t *widget_type)
{
    struct rtgui_widget *widget;
    widget = RTGUI_WIDGET(rtgui_object_create(widget_type));
    return widget;
}
```

可知 要创建 rtgui_widget 对象会先在 rtgui_object 级别创建，再转换为 rtgui_widget 类型

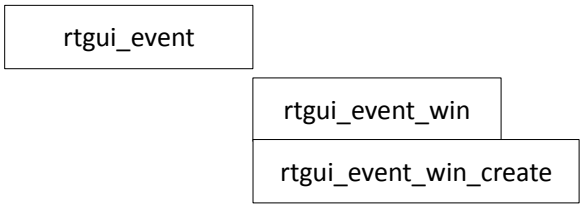
注意 rtgui_object_create 的输入参数依然是是(struct rtgui_type*)&(_rtgui_win)

```
rtgui_object_t *rtgui_object_create(rtgui_type_t *object_type)
{
    rtgui_object_t *new_object;
//申请内存 申请大小即在本例中就是_rtgui_win 中定义的 sizeof(struct rtgui_win)
    new_object = rtgui_malloc(object_type->size);
```

```
    new_object->type = object_type;
    rtgui_type_object_construct(object_type, new_object);// 递归的调用各个类的构造函数
    return new_object;
}
void rtgui_type_object_construct(const rtgui_type_t *type, rtgui_object_t *object)
{
    if (type->parent != RT_NULL) /* first call parent's type */
        rtgui_type_object_construct(type->parent, object);
    if (type->constructor) type->constructor(object);
}
```

所以整个过程将依次调用
_rtgui_object_constructor,
_rtgui_widget_constructor,
_rtgui_container_constructor
_rtgui_win_constructor
最终将创建一个 rtgui_win 对象

2. 事件 event 对象

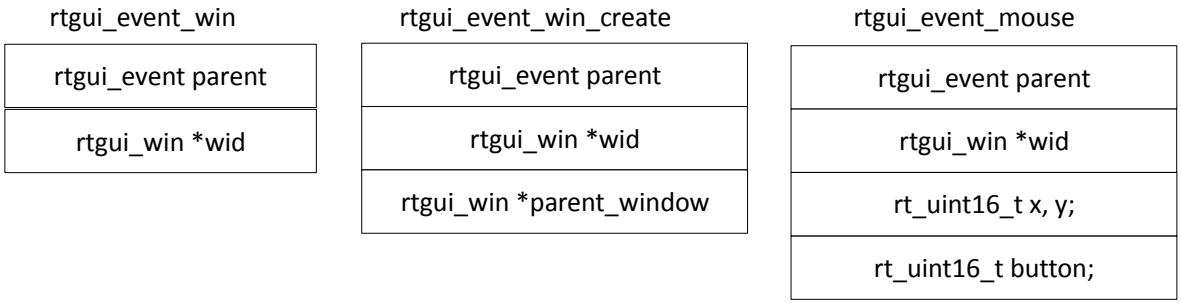


详细说一下基类 rt_event

```
struct rtgui_event
{
    enum _rtgui_event_type type; /* the event type */
    rt_uint16_t user; /* user field of event */
    rt_thread_t sender; /* the event sender */
    rt_mailbox_t ack; /* mailbox to acknowledge request */
};
```

第一项枚举型变量 type 表示了该事件的类型，一般在初始化事件时就会指定好，所有的 event 枚举类型在 event.h 文件中都可以找到
第二项事件的私有数据。
第三项指明此事件的发送者是由哪个线程发送的。
第四项一般用于服务器接收到客户端的某些请求所作出应答。例如当服务器收到 RTGUI_EVENT_WIN_CREATE 事件，在作出相应的处理后，便会调用 rtgui_ack，跟踪进去就会发现实际上是服务器向该事件所指定的邮箱发送了一封应答邮件。

```
    case RTGUI_EVENT_WIN_CREATE:
        if (rtgui_topwin_add((struct rtgui_event_win_create *)event) == RT_EOK)
            rtgui_ack(event, RTGUI_STATUS_OK);
```



可以看出，其他 event 不过是在基类 rtgui_event 的基础上多出一些特有的域，在服务器端的事件处理函数 rtgui_server_event_handler 中，首先得到事件的类型，在根据事件的类型，将事件指针转化为子类特定的事件指针。

第三篇　从 calibration 过程简单了解客户端与服务器的交互

calibration_init()是 calibration 过程的初始化函数，其中主要完成

1-- calibration 实体内存的申请

2-- calibration 设备(即触摸设备)的 calibration_func 函数的设置

3-- calibration 线程的创建和启动。

calibration 线程的入口是 calibration_entry。

```
void calibration_entry(void *parameter)
{
1     struct rtgui_app *app;
2     struct rtgui_win *win;
3     struct rtgui_rect rect;
4     app = rtgui_app_create(rt_thread_self(), "cali");
5     rtgui_graphic_driver_get_rect(rtgui_graphic_driver_get_default(), &rect);
6     calibration_ptr->width = rect.x2;
7     calibration_ptr->height = rect.y2;
8     win = rtgui_win_create(RT_NULL,
          "calibration", &rect, RTGUI_WIN_STYLE_NO_TITLE | RTGUI_WIN_STYLE_NO_BORDER);
9     calibration_ptr->wid = win;
10    rtgui_object_set_event_handler(RTGUI_OBJECT(win), calibration_event_handler);
11    rtgui_win_show(win, RT_TRUE);
12    rtgui_win_destroy(win);
13    rtgui_app_destroy(app);
14    rt_device_control(calibration_ptr->device,RT_TOUCH_CALIBRATION_DATA,
&calibration_ptr->data);
15    rt_device_control(calibration_ptr->device, RT_TOUCH_NORMAL, RT_NULL);
16    rt_free(calibration_ptr); /* release memory */
17    calibration_ptr = RT_NULL;
}
```

第 4 行创建一个名为”cali”的 app 应用，绑定到本线程(calibration)，下面贴出 rtgui_app_create 的代码

```
struct rtgui_app *rtgui_app_create(rt_thread_t tid,const char *title)
{
    rt_thread_t srv_tid;
    struct rtgui_app *app;
    struct rtgui_event_application event;
    app = RTGUI_APP(rtgui_object_create(RTGUI_APP_TYPE));//创建一个 app
    app->tid = tid;                          //这一句和下一句将此 app 和此线程进行一个绑定
    tid->user_data = (rt_uint32_t)app;
    app->mq = rt_mq_create("rtgui",    //创建这个 app 所用的消息队列
                  sizeof(union rtgui_event_generic), 32, RT_IPC_FLAG_FIFO);
    srv_tid = rtgui_get_server();          // 获取服务器的线程 ID
    if (srv_tid == rt_thread_self())       // 只有在创建服务器线程时才会相等
        return app;
```

```
    RTGUI_EVENT_APP_CREATE_INIT(&event);// 初始化一个 RTGUI_EVENT_APP_CREATE 事件
    event.app = app;
    if (rtgui_send_sync(srv_tid, RTGUI_EVENT(&event), sizeof(event)) == RT_EOK)
    {
        return app;
    }
}
```

红色语句表示向服务器线程发送这一事件(APP_CREATE 事件)，从 rtgui_send_sync 的名字中的 sync(同步)可以猜出，这个函数需要服务器端的应答才会继续执行下去。

```
rt_err_t rtgui_send_sync(rt_thread_t tid, rtgui_event_t *event, rt_size_t event_size)
{
    rt_err_t r;
    struct rtgui_app *app;
    rt_int32_t ack_buffer, ack_status;
    struct rt_mailbox ack_mb;
    rtgui_event_dump(tid, event);
    r = rt_mb_init(&ack_mb, "ack", &ack_buffer, 1, 0);
    app = (struct rtgui_app *)(tid->user_data);
    if (app == RT_NULL)
    event->ack = &ack_mb;
    r = rt_mq_send(app->mq, event, event_size);
    r = rt_mb_recv(&ack_mb, (rt_uint32_t *)&ack_status, RT_WAITING_FOREVER);
    rt_mb_detach(&ack_mb);
    return r;
}
```

从蓝色代码可知，rtgui_send_sync(srv_tid, RTGUI_EVENT(&event), sizeof(event))将事件放到服务器应用的消息队列中，并永久地等待服务器的应答。

在 server.c 中经过 switch (event->type)后，执行下面的语句

```
    case RTGUI_EVENT_APP_CREATE:
            rtgui_ack(event, RTGUI_STATUS_OK);
```

如此 便是客户端和服务器最简单的一次交互。

第四篇　topwin 系统

topwin 系统处于 rtgui 的服务器端，它只负责管理各个窗口的位置信息和层叠关系，而具体的绘图操作是由客户端自己完成的。一般来说，服务器通过向客户端应用的消息队列投递事件，客户端根据事件作出相应操作。

topwin 系统将要管理的窗口组成一棵棵树，每棵树的根又通过双向链表连接起来，它管理 rtgui 中所有的窗口并将它们放在恰当的树恰当的位置。在 topwin.c 开头的那段注释中已经写的比较清楚了

```
struct rtgui_topwin
{
    enum rtgui_topwin_flag flag;
    rt_uint32_t mask; /* event mask */
    struct rtgui_wintitle *title;
    struct rtgui_win *wid;     // 每一个 topwin 结构都关联了一个实际的 rtgui_win
    rt_thread_t tid;       /* the thread id */
    rtgui_rect_t extent; /* the extent information */
    struct rtgui_topwin *parent;
    struct rtgui_dlist_node list;
    struct rtgui_dlist_node child_list;
    rtgui_list_t monitor_list;
};
```

其中，双向链表的节点定义如下

```
struct rtgui_dlist_node
{
    struct rtgui_dlist_node *next;    /* point to next node. */
    struct rtgui_dlist_node *prev;    /* point to prev node. */
};
```

在 topwin.c 中定义了一个节点 static struct rtgui_dlist_node _rtgui_topwin_list 它始终处于 topwin 树的最顶上那一层(即树根层)。一般情况下整个 topwin 的树状结构可以示意如下图

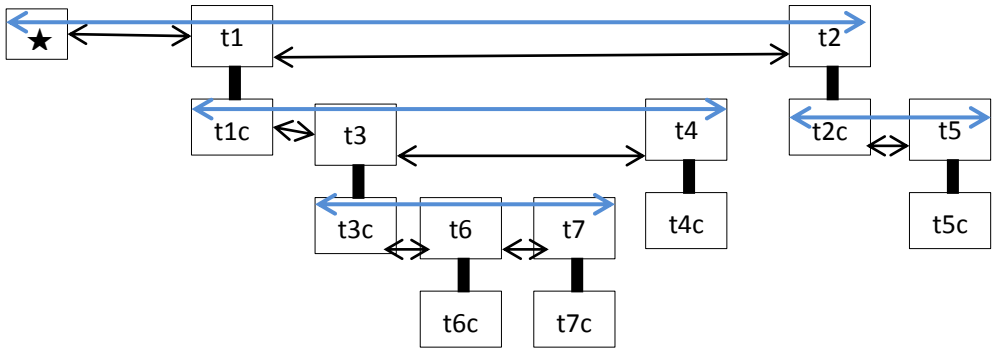


图 3.1

一共画了 2 棵树(根节点分别为 t1,t2)，不过也应该能说明　其中上下两个数字相同的的矩形

代表 1 个 topwin，其中上面部分代表数据结构中的 list，下面部分代表 child_list。★代表 _rtgui_topwin_list。

为方便描述　以下不区分 rtgui_topwin 对象和它所对应的节点，另外，将一层双向链表中在中心节点都画在**最右端**，将中心节点右边第一个节点视为最前，将中心节点右方最后一个的节点视为最后。

举例来说，上图中在根节点层，t1 在最前，t2 在最后；在 t1 这课树中，在 t1c 这一层，t3 在最前，t4 在最后。在 t3c 这一层，t6 在最前，t7 在最后。这样就比较好理解在 topwin.c 开头注释的最后一段

```
/
* Thus, the left most leaf of the tree is the top most window and the right
* most root node is the bottom window.    The hidden part have no specific
* order.
*/
```

结合上图，再看 topwin.c 的函数就应该比较好理解了。用以下函数举几个例子

rt_err_t rtgui_topwin_add(struct rtgui_event_win_create *event)

函数功能:根据 event 指向的信息创建一个 rtgui_topwin 对象，将其插入 topwin 树中合适的位置.

举例来说，假设我要在图 3.1 所示现有的 topwin 系统中插入一个 t9，它所关联的窗口的父窗口是 t1 关联的窗口。

那么函数执行过后的整个 topwin 系统就为

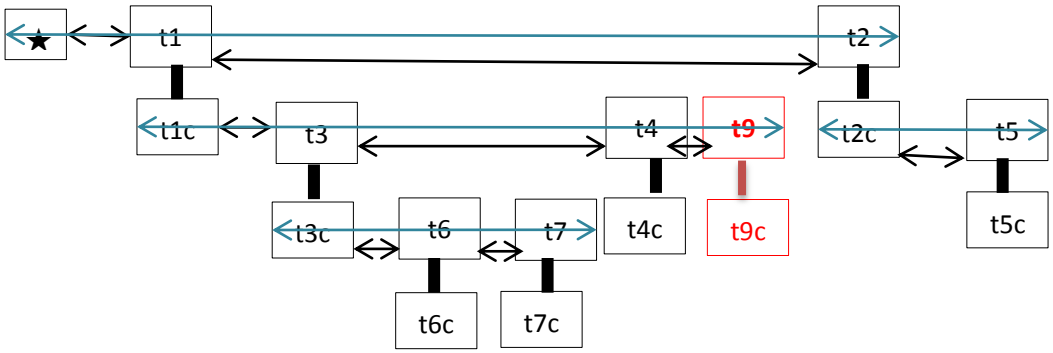


图 3.2

static struct rtgui_topwin *_rtgui_topwin_get_topmost_child_shown(struct rtgui_topwin *topwin)

该函数是找到 topwin 的子孙中可以显示在最顶上的 rtgui_topwin 对象。

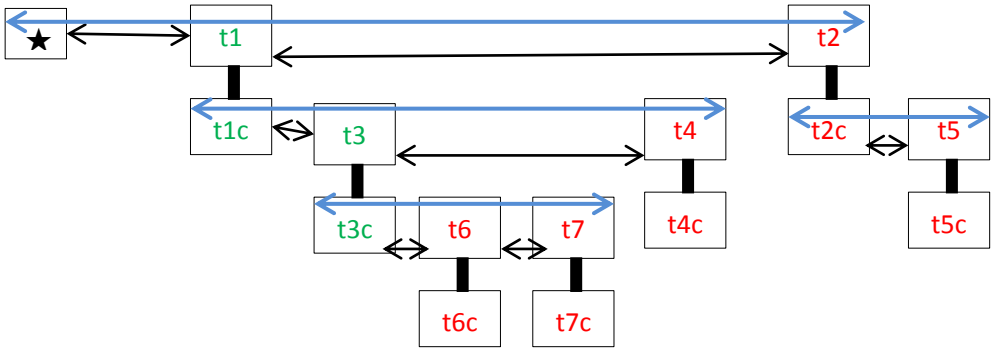


图 3.3

假设此时 topwin 系统如图，其中绿色字体的 rtgui_topwin 的 WINTITLE_SHOWN 标志置位
红色字体的 rtgui_topwin 的 WINTITLE_SHOWN 标志没有置位。则执行
_rtgui_topwin_get_topmost_child_shown(t1)后返回的是 t3。
topwin.c 中有几个函数都是对数结构的操作，比如:
static void _rtgui_topwin_move_whole_tree2top(struct rtgui_topwin *topwin)
static void _rtgui_topwin_raise_in_sibling(struct rtgui_topwin *topwin)
static void _rtgui_topwin_raise_tree_from_root(struct rtgui_topwin *topwin)
要了解每个函数的作用，还是自己动笔在纸上画一画最好。

第五篇　剪切域

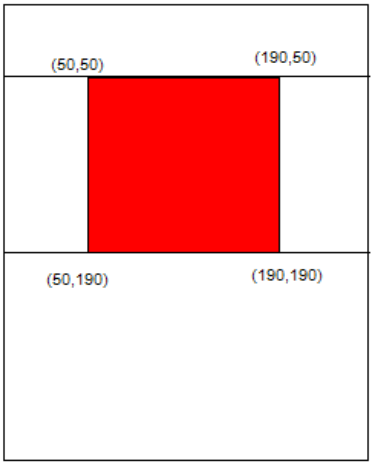
这部分参考了 amsl 的发表的帖子 http://www.rt-thread.org/phpBB3/viewtopic.php?t=882, 在这里先感谢他！

先列出两个数据结构

```
typedef struct rtgui_region
{
    rtgui_rect_t            extents;
    rtgui_region_data_t    *data;
} rtgui_region_t;
struct rtgui_region_data
{
    rt_uint32_t size;
    rt_uint32_t numRects;
};
```

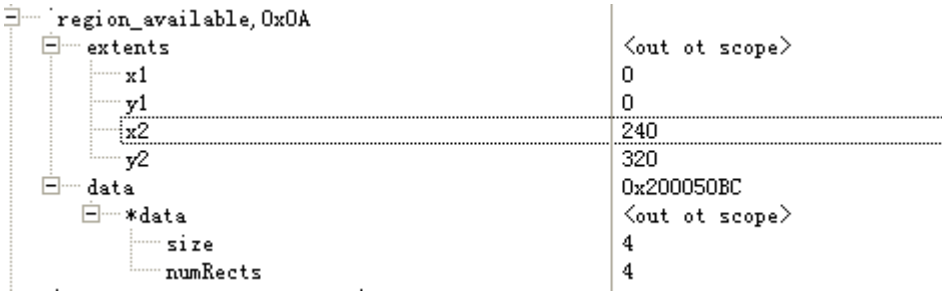
以上两个数据结构是非常重要的，先看看如何用这两个数据结构表示一个区域:

假设屏幕尺寸是 240x320。

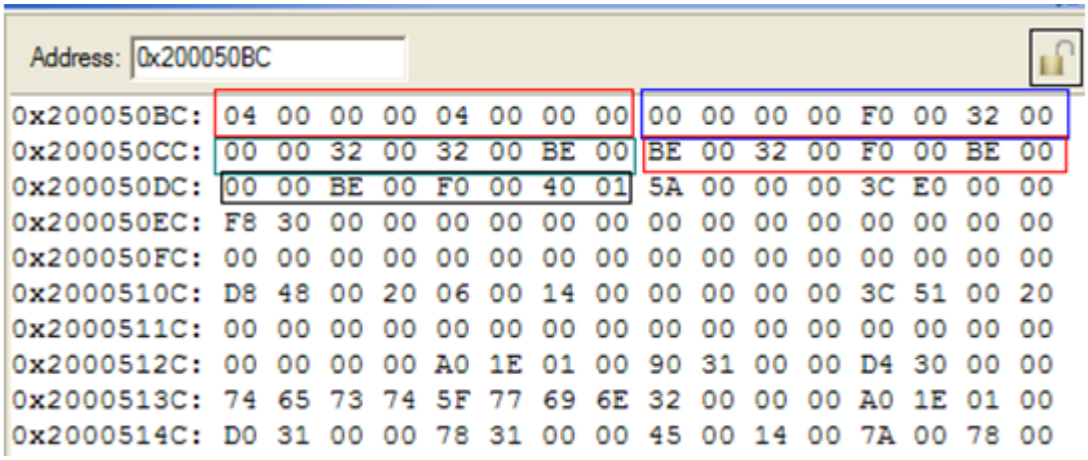


假设我要描述屏幕中除了红色矩形之外的区域，将其定义为 rtgui_region_t　region_available;

那么在'剪去'中间红色矩形后 其各个域的值如下图



由 numRects 可知　这个 region_available 被分成了 4 个矩形，那么我们来看看内存 0x200050BC 处的情况



如上图所示　第一个框中的数据表示 size 和 numRects,后面紧跟的四个框中的数据即为 4 个

rect 的坐标信息:
第一个矩形: x1=0x00; y1=0x00; x2=0xF0; y2=0x32;
第二个矩形: x1=0x00; y1=0x32; x2=0x32; y2=0xBE;
第三个矩形: x1=0xBE; y1=0x32; x2=0xF0; y2=0xBE;
第四个矩形: x1=0x00; y1=0xBE; x2=0xF0; y2=0x140;

其实 不管多复杂剪切域,总能通过这种方式进行描述,只是所有矩形数目比较多。

让我们再来回头看看数据结构 struct rtgui_widget 中的一些东西

```
struct rtgui_widget
{
        ......
        rtgui_rect_t extent; /* the widget extent */
        rtgui_region_t clip; /* the rect clip */
        ......
};
```

其中 extent 表示的矩形记录该 widget 的原始区域，即如果不被其他东西遮挡而应该在屏幕上显示的区域。当剪切域是整个 widget 的可绘区域时，使用 clip.extents 记录该可绘区域。当剪切域有多个矩形区域构成时，则这些信息记录到 clip.data 所指向的内存。

下面用一个简单的例子来说明 rtgui 的服务器客户端的交互流程和一些 topwin 和剪切域的具体过程。

例子内容:在屏幕上用按键控制显示两个部分重叠的窗口。
下面列出代码

```
void application_entry(void *parameter)
{

1    struct rtgui_app *app;
2    struct rtgui_rect rect;
3    rtgui_button_t *button1;
4    rtgui_button_t *button2;
5    struct rtgui_win *main_win;
6    struct rtgui_win *test_win1;
7    struct rtgui_win *test_win2;
8    app = rtgui_app_create(rt_thread_self(), "win_test");
     /* create a full screen window */
9    rtgui_graphic_driver_get_rect(rtgui_graphic_driver_get_default(), &rect);
10   main_win = rtgui_win_create(RT_NULL, "win_test", &rect,
                                 RTGUI_WIN_STYLE_NO_BORDER | RTGUI_WIN_STYLE_NO_TITLE);
11   rect.x1 = 50;
12   rect.x2 = rect.x1 + 100;
13   rect.y1 = 50;
14   rect.y2 = rect.y1 + 40;
```

```
15    test_win1 = rtgui_win_create(main_win, "win1", &rect, RTGUI_WIN_STYLE_DEFAULT);
16    rect.x1 = 100;
17    rect.x2 = rect.x1 + 100;
18    rect.y1 = 70;
19    rect.y2 = rect.y1 + 40;
20    test_win2 = rtgui_win_create(main_win, "win2", &rect,RTGUI_WIN_STYLE_DEFAULT);
21    rect.x1 = 5;
22    rect.x2 = rect.x1 + 100;
23    rect.y1 = 200;
24    rect.y2 = rect.y1 + 20;
     /* 创建按钮用于显示正常窗口 */
25    button1 = rtgui_button_create("show win1");
26    rtgui_widget_set_rect(RTGUI_WIDGET(button1), &rect);
27    rtgui_container_add_child(RTGUI_CONTAINER(main_win), RTGUI_WIDGET(button1));
28    rtgui_button_set_onbutton(button1,show_test_win1);
29    rect.x1 = 5;
30    rect.x2 = rect.x1 + 100;
31    rect.y1 = 260;
32    rect.y2 = rect.y1 + 20;
     /* 创建按钮用于显示正常窗口 */
33    button2 = rtgui_button_create("show win2");
34    rtgui_widget_set_rect(RTGUI_WIDGET(button2), &rect);
35    rtgui_container_add_child(RTGUI_CONTAINER(main_win), RTGUI_WIDGET(button2));
36    rtgui_button_set_onbutton(button2,show_test_win2);

37    rtgui_win_show(main_win, RT_FALSE);
      /* 执行 app 事件循环 */
38    rtgui_app_run(app);      // _rtgui_application
39    rtgui_app_destroy(app);
}
```

第 8 行 创建一个 app，前面提到过，这里就不详说了，只要记住此 app 与该线程相互绑定，其事件处理函数为 rtgui_app_event_handler

第 10 行 创建主窗口 main_win。

```
rtgui_win_t *rtgui_win_create(struct rtgui_win *parent_window,
                              const char *title,
                              rtgui_rect_t *rect,
                              rt_uint16_t style)
{
    struct rtgui_win *win;
    /* 分配内存 并调用_rtgui_win 的构造函数 指定事件处理函数*/
    win = RTGUI_WIN(rtgui_widget_create(RTGUI_WIN_TYPE));
    win->parent_window = parent_window;              //设置父窗口
```

```
    win->title = rt_strdup(title);                          //设置窗口 title
    rtgui_widget_set_rect(RTGUI_WIDGET(win), rect); //设置窗口位置
    win->style = style;                                     //设置窗口样式
    if (_rtgui_win_create_in_server(win) == RT_FALSE)
    {
        goto __on_err;
    }
    return win;
__on_err:
    rtgui_widget_destroy(RTGUI_WIDGET(win));
    return RT_NULL;
}
```

进入_rtgui_win_create_in_server 可以看到:客户端每创建一个窗口，都会将窗口信息以事件的形式发送给服务器端，服务器作出应答并”注册”一个 topwin，并将其加入到 topwin 系统中。

回到**主线程**

11-20 行分别创建两个子窗口 test_win1 和 test_win2。

21-36 行分别创建两个 button 并设置他们的点击函数。

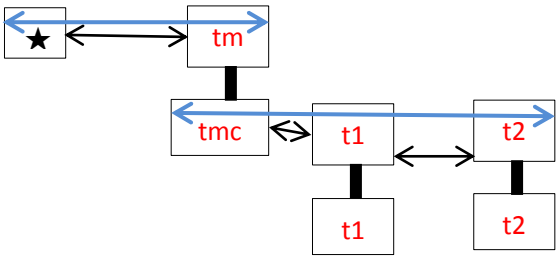
37 行重点来了 调用 rtgui_win_show(main_win, RT_FALSE);

```
rt_base_t rtgui_win_show(struct rtgui_win *win, rt_bool_t is_modal)
{
    rt_base_t exit_code = -1;
    struct rtgui_app *app;
    struct rtgui_event_win_show eshow;
    app = win->app;          // win 的 app 是在创建窗口时,构造函数所指定的,为该线程的 app
    RTGUI_EVENT_WIN_SHOW_INIT(&eshow);
    eshow.wid = win;
    /* set window unhidden before notify the server */
    rtgui_widget_show(RTGUI_WIDGET(win));
    if (rtgui_server_post_event_sync(RTGUI_EVENT(&eshow),
                                     sizeof(struct rtgui_event_win_show)) != RT_EOK)
    {
        /* It could not be shown if a parent window is hidden. */
        rtgui_widget_hide(RTGUI_WIDGET(win));
        return exit_code;
    }
    if (win->focused_widget == RT_NULL)
        rtgui_widget_focus(RTGUI_WIDGET(win));
    /* set main window */
    if (app->main_object == RT_NULL)
        rtgui_app_set_main_win(win);
```

```
    return exit_code;
}
```

蓝色部分的代码表示构造一个 rtgui_event_win_show 事件，并填入相关窗口信息，然后通过 rtgui_server_post_event_sync 与服务器端进行交互。

现在的 topwin 树结构如下:



其中 ★代表_rtgui_topwin_list。tm 表示 main_win 对应的 topwin，t1 和 t2 分别表示 test_win1 和 test_win2 对应的 topwin。

服务器端根据事件类型会用 rtgui_topwin_show((struct rtgui_event_win *)event)

```
rt_err_t rtgui_topwin_show(struct rtgui_event_win *event)
{
1     struct rtgui_topwin *topwin;
2     struct rtgui_win *wid = event->wid;
3     topwin = rtgui_topwin_search_in_list(wid, &_rtgui_topwin_list);
4     if (!_rtgui_topwin_could_show(topwin))
5     {
6           topwin->flag |= WINTITLE_SHOWN;
7           _rtgui_topwin_raise_in_sibling(topwin);
8           return -RT_ERROR;
9     }
10    _rtgui_topwin_preorder_map(topwin, _rtgui_topwin_mark_shown);
11    rtgui_topwin_activate_topwin(topwin);
12    return RT_EOK;
}
```

第 3 行 找到 mian_win 对应的 topwin。

第 4-9 行 查看该 topwin 能否显示，其实是检查其父窗口能否显示。

第 10 行 设置该 topwin 的 WINTITLE_SHOWN 标志，以及对应窗口的 RTGUI_WIDGET _FLAG_SHOWN 标志。

第 11 行 详细说说 rtgui_topwin_activate_topwin(topwin)，来看看其代码

```
rt_err_t rtgui_topwin_activate_topwin(struct rtgui_topwin *topwin)
{
1       struct rtgui_event_paint epaint;
2       RTGUI_EVENT_PAINT_INIT(&epaint);
3       _rtgui_topwin_raise_tree_from_root(topwin);
4       rtgui_topwin_update_clip();
```

```
5       _rtgui_topwin_only_activate(topwin);
6       _rtgui_topwin_draw_tree(topwin, &epaint);
7       return RT_EOK;
}
```

第 1-2 行初始化了一个 RTGUI_EVENT_PAINT 事件。

第 3 行_rtgui_topwin_raise_tree_from_root 的作用是移动 topwin 所在的整课树到前方并将 topwin 移动到所在层的最前方。

第 4 行 更新剪切域

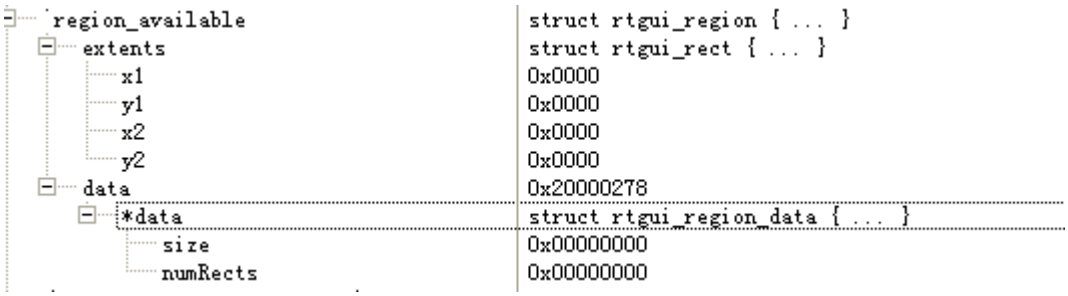
```
static void rtgui_topwin_update_clip(void)
{
1       struct rtgui_topwin *top;
2       struct rtgui_event_clip_info eclip;
3       struct rtgui_region region_available;
4       RTGUI_EVENT_CLIP_INFO_INIT(&eclip);
5       rtgui_region_init_rect(&region_available, 0, 0,
                                        rtgui_graphic_driver_get_default()->width,
                                        rtgui_graphic_driver_get_default()->height);
6       top = _rtgui_topwin_get_topmost_window_shown(0);
7       while (top != RT_NULL)
8       {
9             _rtgui_topwin_clip_to_region(&region_available, top);
10            rtgui_region_subtract_rect(&region_available, &region_available, &top->extent);
11             eclip.wid = top->wid;
12             rtgui_send(top->tid, &(eclip.parent), sizeof(struct rtgui_event_clip_info));
13             if (top->parent == RT_NULL)
14                   if (top->list.next != &_rtgui_topwin_list &&get_topwin_from_list(top->list.next)->
                                        flag & WINTITLE_SHOWN)
15                         top = _rtgui_topwin_get_topmost_child_shown(get_topwin_from_list
                                  (top->list.next));
16                   else
17                         break;
            else if (top->list.next != &top->parent->child_list &&
                              get_topwin_from_list(top->list.next)->flag & WINTITLE_SHOWN)
                  top = _rtgui_topwin_get_topmost_child_shown(get_topwin_from_list
                           (top->list.next));
            else      top = top->parent;
      }
}
```

第 4 行初始化一个 RTGUI_EVENT_CLIP_INFO 事件,准备后面发送给客户端。

第 5 行 得到 region_available，其值是整个屏幕

第 6 行 会得到 main_win 对应的 topwin

第 9-10 行 用该 topwin 的尺寸信息’剪切’ region_available。由于我们的 main_win 是全屏的，所以这两句执行之后，region_available 的值如下



第 12 行 将此 RTGUI_EVENT_CLIP_INFO 事件投递到应用 app 的消息队列。(以后会说在哪里处理)

13 行以后的代码 由于本例子太简单 很快回到 17 行的 break 跳出循环。

回到 rtgui_topwin_activate_topwin

```
rt_err_t rtgui_topwin_activate_topwin(struct rtgui_topwin *topwin)
{
1       struct rtgui_event_paint epaint;
2       RTGUI_EVENT_PAINT_INIT(&epaint);
3       _rtgui_topwin_raise_tree_from_root(topwin);
4       rtgui_topwin_update_clip();
5       _rtgui_topwin_only_activate(topwin);
6       _rtgui_topwin_draw_tree(topwin, &epaint);
7       return RT_EOK;
}
```

第 5 行 跟踪进_rtgui_topwin_only_activate 会发现里面设置 WINTITLE_ACTIVATE 标志，并向 app 的消息队列投递第 2 次事件，事件类型是 RTGUI_EVENT_WIN_ACTIVATE。

第 6 行 跟踪进_rtgui_topwin_draw_tree 可知函数向 app 的消息队列投递第 3 次事件，事件类型是 RTGUI_EVENT_PAINT 事件，并遍历该 topwin 的子节点，递归的调用本函数。

如此之后 便会回到主线程 线程函数执行至此 尽管调用了 rtgui_win_show 但屏幕上什么都没有显示，原因就是**客户端 app 还没有处理消息队列中收到的事件**!!

回到主线程

```
void application_entry(void *parameter)
{
      ......
37    rtgui_win_show(main_win, RT_FALSE);
      /*  执行 app 事件循环  */
38    rtgui_app_run(app);
39    rtgui_app_destroy(app);
}
```

跟踪进第 38 行

```
rt_base_t rtgui_app_run(struct rtgui_app *app)
{
      _rtgui_application_check(app);
      app->state_flag &= ~RTGUI_APP_FLAG_EXITED;
```

```
    _rtgui_application_event_loop(app);
    if (app->ref_count == 0)
        app->state_flag |= RTGUI_APP_FLAG_EXITED;
    return app->exit_code;
}
```

跟踪进_rtgui_application_event_loop

```
rt_inline void _rtgui_application_event_loop(struct rtgui_app *app)
{
    rt_err_t result;
    rt_uint16_t current_ref;
    struct rtgui_event *event;
    event = (struct rtgui_event *)app->event_buffer;
    current_ref = ++app->ref_count;
    while (current_ref <= app->ref_count)
    {
        if (app->on_idle != RT_NULL)
        {
            result = rtgui_recv_nosuspend(event, sizeof(union rtgui_event_generic));
            if (result == RT_EOK)
                RTGUI_OBJECT(app)->event_handler(RTGUI_OBJECT(app), event);
            else if (result == -RT_ETIMEOUT)
                app->on_idle(RTGUI_OBJECT(app), RT_NULL);
        }
        else
        {
            result = rtgui_recv(event, sizeof(union rtgui_event_generic));
            if (result == RT_EOK)
                RTGUI_OBJECT(app)->event_handler(RTGUI_OBJECT(app), event);
        }
    }
}
```

跟踪进 result = rtgui_recv(event, sizeof(union rtgui_event_generic))

```
rt_err_t rtgui_recv(rtgui_event_t *event, rt_size_t event_size)
{
    struct rtgui_app *app;
    rt_err_t r;
    app = (struct rtgui_app *)(rt_thread_self()->user_data);
    r = rt_mq_recv(app->mq, event, event_size, RT_WAITING_FOREVER);
    return r;
}
```

可以看到 程序在这里接收线程 app 消息队列的事件。

回到_rtgui_application_event_loop

```
{
        ......
```

```
        result = rtgui_recv(event, sizeof(union rtgui_event_generic));
        if (result == RT_EOK)
        RTGUI_OBJECT(app)->event_handler(RTGUI_OBJECT(app), event);
}
```

当从消息队列接收到事件之后，调用 app 的 OBJECT 级别的事件处理函数 event_handler，这个函数在 app 的构造函数中被指定为:

```
rt_bool_t rtgui_app_event_handler(struct rtgui_object *object, rtgui_event_t *event)
{
    struct rtgui_app *app;
    app = RTGUI_APP(object);
    switch (event->type)
    {
    case RTGUI_EVENT_PAINT:                 //收到的第 3 个事件的类型
    case RTGUI_EVENT_CLIP_INFO:             //收到的第 1 个事件的类型
    case RTGUI_EVENT_WIN_ACTIVATE:      //收到的第 2 个事件的类型
    case RTGUI_EVENT_WIN_DEACTIVATE:
    case RTGUI_EVENT_WIN_CLOSE:
    case RTGUI_EVENT_WIN_MOVE:
    case RTGUI_EVENT_KBD:
        _rtgui_application_dest_handle(app, event);
        break;
    ......
}
rt_inline rt_bool_t _rtgui_application_dest_handle(
    struct rtgui_app *app,
    struct rtgui_event *event)
{
    struct rtgui_event_win *wevent = (struct rtgui_event_win *)event;
    struct rtgui_object *dest_object = RTGUI_OBJECT(wevent->wid);
    if (dest_object->event_handler != RT_NULL)
        return dest_object->event_handler(RTGUI_OBJECT(dest_object), event);          、
    else
        return RT_FALSE;
}
```

可以看出，app 的事件处理函数并不直接处理某些类型的事件，取而代之的是将其派发到事件相关窗口自己的事件处理函数,这个函数在窗口创建时由构造函数指定为:

```
rt_bool_t rtgui_win_event_handler(struct rtgui_object *object, struct rtgui_event *event)
```

此函数处理收到的第一个 RTGUI_EVENT_CLIP_INFO 类型的事件将调用

```
    rtgui_win_update_clip(win)
```

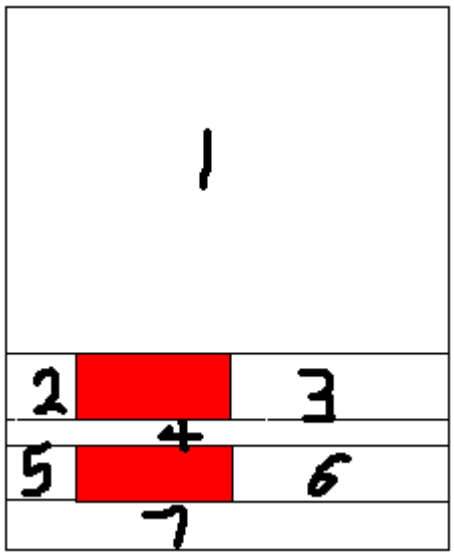
此函数的作用是更新 win 及其作为 container 的 children 的剪切域。

具体的到本例，便是更新 main_win 以及 button1 和 button2 的剪切域。

这是执行过后 main_win 的 clip

clip	struct rtgui_region { ... }
extents	struct rtgui_rect { ... }
x1	0
y1	0
x2	240
y2	320
data	0x200054B4
*data	struct rtgui_region_data { ... }
size	8
numRects	7

即 main_win 的 clip 被分割成了 7 个矩形



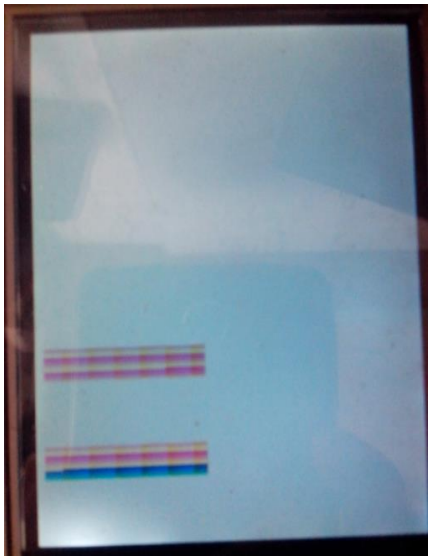
接下来处理收到的第 2 个 RTGUI_EVENT_WIN_ACTIVATE 类型的事件，主要是置位窗口的 RTGUI_WIN_FLAG_ACTIVATE 标志；

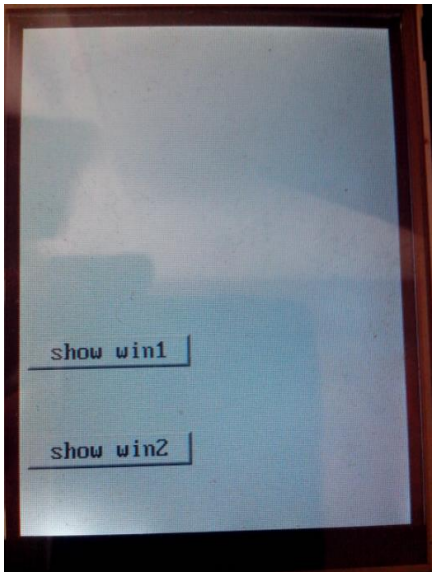
接着处理收到的第 3 个 RTGUI_EVENT_PAINT 类型的事件，处理函数将调用

rtgui_win_ondraw(win)

```
static rt_bool_t rtgui_win_ondraw(struct rtgui_win *win)
{
1       struct rtgui_dc *dc;
2       struct rtgui_rect rect;
3       struct rtgui_event_paint event;
4       dc = rtgui_dc_begin_drawing(RTGUI_WIDGET(win));
5       rtgui_widget_get_rect(RTGUI_WIDGET(win), &rect);
6       rtgui_dc_fill_rect(dc, &rect);
7       RTGUI_EVENT_PAINT_INIT(&event);
8       event.wid = RT_NULL;
9       rtgui_container_dispatch_event(RTGUI_CONTAINER(win),
                                       (rtgui_event_t *)&event);
10       rtgui_dc_end_drawing(dc);
11       return RT_FALSE;
}
```

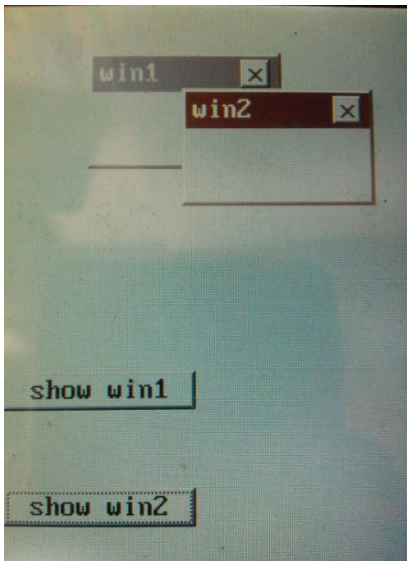
在执行完 1-6 行后，实际上我们的 main_win 已经在屏幕上绘制出来,绘制区域即为自己的剪切域，绘制结果如下图(左):





第 7 行是将此事件分发给 win 的所有子 widget，子 widget 在收到事件后会执行自己的绘制过程。绘制完成如上图(右)所示:

本来后面还有按键处理的一些东西，但思来想去与前面介绍的也是只有少许地方有不同，于是偷个懒就不写了 O(∩_∩)O 哈哈~，最后的结果就是下面这样。



<全文完>　by　cmjs