# 对象结构

资源线：

UIApplication

SkinMgr

SkinRes

ImageMgr

FontMgr

ColorMgr

I18nMgr

StyleMgr

ImageRes

FontRes

ColorRes

I18nRes

StyleRes

SkinRes

对象线：

IMessage

IObject

IControl

IButton

ICheckButton

Message

Object

Control

Button

CheckButton

UISDK

UICTRL

# 注册控件

interface IobjectDescription

// 注册控件

UICTRL\_RegisterUIObject(m\_pUIApp);

UICef\_RegisterUIObject(m\_pUIApp);

# Xml书写

资源xml的入口为skin.xml

需要引入其它配置，加<include>xxx.xml</include>标签

## 字体

第一个配置的字体将作为整个系统的默认字体

<font>

<font id="normal" face="宋体" size="10"/>

</font>

## 图片

<image>

<image id="icon" path="icon\_32x32.png"/>

<imagelist id="btn" path="btn.png" layout="h" count="3"/>

</image>

图片如果需要支持dpi自适应，应该再添加一张icon\_32x32@2x.png的图片，系统会根据当前的dpi自动选取一张合适的图片。

## 颜色

## 国际化

将string attribute指定国际化标志，它的xml内容如果以#前缀开始，则将会做国际化资料匹配。

s.AddString(XML\_TEXT, this,

memfun\_cast<pfnStringSetter>(&Button::SetText),

memfun\_cast<pfnStringGetter>(&Button::SaveText))

**->Internationalization()**

->AsData();

<Button id="btn" width="100" height="100" tooltip="100\*100" **text="#ok"**>

<i18n>

**<string id="ok" zh\_cn="确定" en\_us="OK"/>**

</i18n>

如何支持多语言环境下的字体配置？

将字体id进行国际化(在字体id前面加上#)

<font id = "hyperlink.font.zh\_cn" face = "微软雅黑" size = "10" / >

<font id = "hyperlink.font.en\_us" face = "微软雅黑" size = "8" / >

<i18n>

<string id="hyperlink.font"

zh\_cn="hyperlink.font.zh\_cn"

en\_us="hyperlink.font.en\_us"/>

</i18n>

<prop key = "text.render" type = "colorlist" font = "#hyperlink.font"

## 样式

类似于css的功能，将一些控件都共用的属性抽出来，作成style

Style支持： class, tag, id

## 属性

对象的属性由 UI\_MSG\_SERIALIZE 消息控制，该消息有三种作用：

* 加载属性
* 保存属性
* 在编辑器中显示属性

void Button::OnSerialize(SERIALIZEDATA\* pData)

{

DO\_PARENT\_PROCESS(IButton, IButtonBase);

AttributeSerializerWrap s(pData, TEXT("Button"));

s.AddI18nString(XML\_TEXT,

[this](LPCTSTR t, int s){ m\_strText.assign(t, s); },

[this](){ return m\_strText.c\_str(); })

->AsData();

s.AddLong(XML\_BUTTON\_ICON\_TEXT\_SPACE, m\_lIconMarginText);

s.AddBool(XML\_BUTTON\_DEFPUSHBUTTON, this,

memfun\_cast<pfnBoolSetter>(&Button::SetDefaultPushButtonStyle),

memfun\_cast<pfnBoolGetter>(&Button::HasDefaultPushButtonStyle))

->SetDefault(false);

属性的类型有很多种: string, i18n string, bool, long, enum, flags, color, renderbase, textrender ...

属性的绑定有三种方式：属性引用、getter/setter、lambda

## 属性下移

有些属性需要写多个xml attribute才能表达出来，为了使这个类型的属性更清楚的表现在xml当中，允许将这些属性下移到<prop>结点下面。

这个写法是借鉴了mac下面的xib写法。

<CustomWindow id="Window" width="500" height="500" bkgnd.render.type="color" bkgnd.render.color="0xFFF">

可以写成：

<CustomWindow id="Window" width="500" height="500" >

<prop key="back.render" type="color" color="0xFFF"/>

</CustomWindow>

在xib下面的写法为：

<view clearsContextBeforeDrawing="NO" contentMode="scaleToFill" id="i5M-Pr-FkT">

<rect key="frame" x="0.0" y="20" width="768" height="1004"/>

</view>

如果配置了key属性，则会将key属性作为前缀，添加在其它属性的前面。如果没有配置，则直接读取该属性

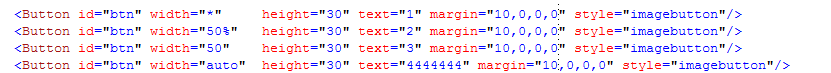
## 布局

为了便于书写<prop>，增加了兼容key字段

width，兼容layout.width=””

height，兼容layout.height=””

水平布局宽度支持：平均、比例、指定、自动。



C:\Users\Administrator.SKY-20160823UPD\Documents\Tencent Files\329075202\Image\C2C\8C]R(01L6TYJJU0MHBUF(UN.png

## 控件背景配置

让按钮支持更多的样式，如颜色、图片、系统样式等等，由一个统一的接口提供基础渲染功能，这个接口被称为renderbase。

Renderbase 现在支持： color, colorlist, image, imagelist。

每个控件都从object中继承了两个renderbase，分别是 back.render、fore.render。

文字的绘制由 textrender提供。

<class id="imagebutton"

back.render.type="imagelist"

back.render.image="btn"

back.render.image.drawtype="stretch"

back.render.image.9region="10"/>

# 特性

## 事件绑定

除了使用IMessage接口进行控件通知传递外，一些常用的通知采用了绑定的方式来实现。

采用绑定的方式，只需要在初始化的时候调用一次，不需要再单独保存这个控件的指针用于后续的事件判断了，简化了代码逻辑。

UI::IButton\* pBtn = (UI::IButton\*)m\_pWindow->FindChildObject(TEXT("btn"));

if (pBtn)

{

pBtn->ClickedEvent().connect(this, &MainWindow::on\_click);

}

void on\_click(UI::IButton\*)

{

::MessageBox(NULL, TEXT("on\_click"), TEXT("info"), MB\_OK);

}

目前事件绑定只提供了单一绑定，一个signal最多只能绑定一个slot。

支持绑定参数全省略，但不支持部分省略，绑定时需改为调用connect0，例如：

pBtn->ClickedEvent().connect**0**(this, &LoginWindow::on\_click);

void LoginWindow::on\_click(/\*UI::IButton\*\*/)

{

}

## 数据驱动

（目前支持的不是很完善，也不支持跨线程）

以往的控件编程经验：控件提供set方法，将要显示的文本内容set进去。因此业务层收到变更后，要先存一份，然后发一个通知到界面层，通知控件更新。

这样方式会导致整个代码中充满了这种胶水代码。

数据驱动是指是修改了某个数据时，自动会更新到控件当中的技术。

声明数据成员

UI::bindable\_int m\_nCount;

UI::bindable\_string m\_strName;

将控件属性绑定到该数据上面

UI::ILabel\* pLabel = (UI::ILabel\*)m\_pWindow->FindChildObject(TEXT("label1"));

if (pLabel)

{

m\_nCount.AddObserver(pLabel->GetListener(), UI::ILabel::PROPERTY\_TEXT, 0, 0);

}

pLabel = (UI::ILabel\*)m\_pWindow->FindChildObject(TEXT("label2"));

if (pLabel)

{

m\_strName.AddObserver(pLabel->GetListener(), UI::ILabel::PROPERTY\_TEXT, 0, 0);

}

之后再修改m\_nCount/m\_strName时，会自动触发控件的更新。

## 多皮肤资源包

为了支持模块插件化，每个模块的皮肤也允许单独存放。

每个Object都会属于一个SkinRes资源对象。

SkinRes之间可以设置父子关系，在子SkinRes中，可以直接引用父SkinRes中的资源对象。

// 顶级资源

TCHAR szResPath[MAX\_PATH] = { 0 };

UI::Util::PathInBin(L"Skin\\Root", szResPath);

UI::ISkinRes\* pRootSkinRes = pUIApp->LoadSkinRes(szResPath);

// 登录插件资源

UI::Util::PathInBin(L"Skin\\Plugin\\Login", szResPath);

UI::ISkinRes\* pLoginSkinRes = pUIApp->LoadSkinRes(szResPath);

// 绑定

pLoginSkinRes->SetParentSkinRes(pRootSkinRes);

LoginWindow m;

m.Create(pLoginSkinRes);

## 图片拉伸抗锯齿

默认使用alphablend是不支持抗锯齿的。在GdiRenderTarget::DrawBitmapEx中，如果发现当前是拉伸模式，则会采用 gdiplus 的方式进行绘制。

## 窗口透明

1. 分层窗口

<CustomWindow id="Window" width="300" height="300">

<prop key="back.render" type="color" color="0x8FFFFFFF"/>

<prop key="transparent" type="layered"/>

1. 磨砂效果

<CustomWindow id="Window" width="300" height="300">

<prop keyx="back.render" type="color" color="0x8FFFFFFF"/>

<prop key="transparent" type="aero" aero.blur.region="10" aero.disable="layered"/>

1. 外发光效果（待完善）

## 列表控件

### ListBox基本配置

Listbox里的滚动条需要自己单独配置

<ListBox id="listbox" border="1">

<prop key="layout" width="200" height="100" top="50" left="10"/>

<prop key="back.render" type="color" border="1" border.color="#000"/>

<prop key="fore.render" type="colorlist" color=";;;;#03B;#03B;#03B;;"/>

<prop key="text.render" type="colorlist" color=";;;;#FFF;#FFF;#FFF;;"/>

<VScrollBar ncchild="1" id="\_\_vscrollbar" width="11">

<Button id="\_\_scrollbar\_thumb" back.render.type="color" back.render.color="#eee"/>

</VScrollBar>

</ListBox>

### 自定义列表项配置

允许在xml中配置一个列表项模板，然后在代码中进行加载。如下。

<ListItem id="listitem">

<Button id="btn" style="imagebutton" width="50" height="30" layout.right="10" layout.top="0"/>

</ListItem>

UI::ICustomListItem\* pListItem = (UI::ICustomListItem\*)

pListBox->InsertCustomItem(TEXT("listitem"), i);

if (pListItem)

{

UI::IButton\* pBtn = (UI::IButton\*)

pListItem->FindControl(TEXT("btn"));

pBtn->SetText(szText);

pBtn->ClickedEvent().connect(this, &LoginWindow::on\_listitem\_btn\_click);

}

### 菜单加载

首先需要配置菜单模板，并支持在模板中直接配置菜单项，如下。

<CustomWindow id="Menu.Default" padding="7">

<prop key="back.render" type="image" image="menu.back" image.drawtype="stretch" image.9region="35,10,10,10"/>

<prop key="transparent" type="layered"/>

<Menu icongutterwidth="25" separatorheight="3">

<prop key="fore.render" type="imagelist" image="menu.fore" image.drawtype="stretch" image.9region="3"/>

<prop key="text.render" type="colorlist" color="0xF000;0xFFFF;0xFFFF"/>

<prop key="separator.render" type="image" image="menu.sep" image.drawtype="stretch" image.9region="1"/>

<String text="TestMenuItem"/>

</Menu>

</CustomWindow>

UI::LoadMenuData data = { 0 };

data.pSkinRes = m\_pWindow->GetSkinRes();

data.hWndParent = m\_pWindow->GetHWND();

UI::CMenuPtr pMenu = UILoadMenu(&data);

pMenu->AppendString(L"AAAA", 1);

pMenu->AppendSeparator(0);

pMenu->AppendString(L"BBBB", 2);

POINT pt;

GetCursorPos(&pt);

pMenu->TrackPopupMenu(TPM\_RETURNCMD, pt.x, pt.y, 0);