# Xml书写

资源xml的入口为skin.xml

## 字体

第一个配置的字体将作为整个系统的默认字体

<font>

<item 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>

## 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。

其中下移后结点tag名称尚未使用，可任意。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>

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

</prop>

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=””

width.type，兼容layout.width.type=””

height.type，兼容layout.height.type=””

# 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);

}

## 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>

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

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

</prop>

1. 磨砂效果

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

<prop>

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

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

</prop>

1. 外发光效果（待完善）