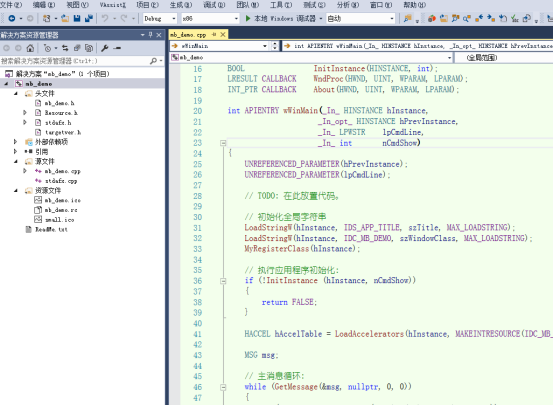
**新手入门**

Miniblink的上手非常简单，下面我们将用最简单的步骤跑起一个web页面。

Miniblink只有一个dll：node.dll。很多人问这不是nodejs的dll吗，是的，因为miniblink有electron模式，所以需要伪装成nodejs的dll，便改成了这个名字。

首先第一步，从https://github.com/weolar/miniblink49/releases里下载最新的包，

第二步，在vs里新建一个win32工程mb\_demo



然后引入包里的头文件（wke.h的具体位置和引用路径看你喜好了）：

#include "wke.h"

再写入下面5行代码：

wkeSetWkeDllPath(L"E:\\mycode\\miniblink49\\trunk\\out\\Release\_vc6\\node.dll");

wkeInitialize();

wkeWebView window = wkeCreateWebWindow(WKE\_WINDOW\_TYPE\_POPUP, NULL, 0, 0, 1080, 680);

wkeLoadURL(window, "qq.com");

wkeShowWindow(window, TRUE);

文件编译运行：



所以一切都是这么的简单，5行代码即可。

有眼尖的人发现了，整个过程没看到填入lib。是的，我在wke.h头文件里玩了点小技巧，所有导出接口都自动帮你load好了，你只要引入wke.h，再通过wkeSetWkeDllPath设置node.dll的路径即可（wkeSetWkeDllPath如果不写全路径，写成相对路径，则是相对exe的工作目录）。

**离屏渲染**

上面这个演示的是wkeCreateWebWindow创建窗口。这个接口表示mb内部帮你创建好了窗口，但很多时候，我们需要更细致的控制mb的功能。例如在游戏中，我们不需要窗口，此时可以用wkeCreateWebView()这个接口。这个接口没有参数，可以用来实现游戏中的离屏渲染。

下面代码实现了一个离屏渲染的简单示例：

RegWndClass(CLS\_WINDOW, CS\_HREDRAW | CS\_VREDRAW);

::wkeInitialize();

wkeWebView view = wkeCreateWebView();

HWND hWnd = ::CreateWindowEx(WS\_EX\_APPWINDOW, CLS\_WINDOW, NULL, WS\_OVERLAPPEDWINDOW, 0, 0, 840, 680, NULL, NULL, ::GetModuleHandle(NULL), NULL);

::SetProp(hWnd, L"wke", (HANDLE)view);

wkeSetHandle(view, hWnd);

wkeOnPaintUpdated(view, HandlePaintUpdatedCallback, hWnd);

RECT rc = { 0 };

::GetClientRect(hWnd, &rc);

wkeResize(view, rc.right, rc.bottom);

wkeMoveToCenter(view);

wkeShowWindow(view, TRUE);

wkeLoadURL(view, "http://news.sina.com.cn");

其实流程和普通流程差距不大。最大的两点差距：

1. 外部创建窗口，并设置给mb。
2. 绑定了wkeOnPaintUpdated回调。

这里有个问题，上面不是说了有可能不需要窗口吗，怎么还要创建窗口并设置给mb。答案是你不创建也可以，但有几个功能mb会工作不正常。一个是下拉框。因为mb的下拉框是独立窗口，如果外部创建窗口并且设置给mb的话，无法计算需要在屏幕的哪个部位弹出。二是flash的window模式也需要窗口句柄。

还有个问题，为何需要-绑定wkeOnPaintUpdated回调？

原因是mb在发现使用离屏模式后，不会主动发消息给父窗口通知窗口需要刷新，所以需要通过此回调来通知。完整示例可见github。

**JS绑定**

Mb导出了一组js开头的接口，如jsCallGlobal等。现在有个最基本的问题：如何让js和c++进行交互？

最简单的办法就是用jsBindFunction：

jsBindFunction("onJsCall", onJsCall, 3);

这句话就绑定了一个C函数给js。第一个参数表示js里的全局函数名，jsBindFunction是直接绑定到主frame的window对象上的（所以新手比较容易犯的一个错是在子frame里也调用onJsCall）。第二个参数是C的回调。第三个参数是js会传入多少个参数。

Js里可以用window.onJsCall(1, "str", 1.3) 来调用。

C里，可以这样写：

jsValue JS\_CALL onJsCall(jsExecState es) {

jsValue val0 = jsArg(es, 0);

jsValue val1 = jsArg(es, 1);

jsValue val2 = jsArg(es, 2);

int int0 = jsToInt(es, val0);

const utf8\* str1 = jsToTempString(es, val1);

float float2 = jsToFloat(es, val2);

return jsString(es, "I am return value;");

}

可以看到int0，str1，float2就是从js里获取到的值。最后还返回了

"I am return value;"字符串给js。

上面是js调用c，下面再看如何c里调用js。

目前有三种方法实现从C调用JS，一种是：

jsValue wkeRunJsByFrame(wkeWebView webView, wkeWebFrameHandle frameId, const utf8\* script, bool isInClosure)

第二种是：

jsValue jsEvalExW(jsExecState es, const wchar\_t\* str, bool isInClosure)

第三种是：

jsValue jsCall(jsExecState es, jsValue func, jsValue thisValue, jsValue\* args, int argCount)

wkeRunJsByFrame可以针对某个frame运行js。如果是主frame，第二个参数传wkeWebFrameGetMainFrame 数返回的值。

wkeRunJsByFrame(window, wkeWebFrameGetMainFrame(window), "window.onJsCall(1, \"str\", 1.3);", true);

最后一个参数表示js代码是否外部被包裹在一个闭包中。如果为true，其实最后传给mb的js代码是这样:

"functiuon () {window.onJsCall(1, \"str\", 1.3);}"

这样做的原因是为了防止js的变量污染。

下面是个完整的例子：

jsValue JS\_CALL onJsCall(jsExecState es) {

jsValue val0 = jsArg(es, 0);

jsValue val1 = jsArg(es, 1);

jsValue val2 = jsArg(es, 2);

int int0 = jsToInt(es, val0);

const utf8\* str1 = jsToTempString(es, val1);

float float2 = jsToFloat(es, val2);

return jsString(es, "I am return value;");

}

void onDocumentReady(wkeWebView webView, void\* param, wkeWebFrameHandle frameId) {

wkeWebFrameHandle mainFrame = wkeWebFrameGetMainFrame(webView);

jsValue v = wkeRunJsByFrame(webView, mainFrame, "window.onJsCall(1, \"str\", 1.3);", false);

jsExecState es = wkeGetGlobalExecByFrame(webView, mainFrame);

const utf8\* str1 = jsToTempString(es, v);

str1 = str1;

}

之所以要在onDocumentReady里做演示，是因为此时dom 树已创建完毕，以后要加功能，操作dom什么的，比较方便。虽然我们这个例子用不到。

这里有个小注意点：wkeRunJsByFrame里执行的js，如果想拿到返回值，要分最后一个参数是否为true。如果为true，按照上面描述的，会被封装在闭包里，所以要加return。如果是false，则不用加return。

jsEvalExW类似，就不讲了。

jsCall表示调用一个js函数。下面是一个复杂点的示例：

jsValue jsFunctionCallback(jsExecState es, jsValue object, jsValue\* args, int argCount) {

jsData \*jsObjData = ::jsGetData(es, args[0]);

printf("jsObjData:%p\n", jsObjData); //取不到native对象的数据

return ::jsUndefined();

}

void jsFunctionFinalizeCallback(jsData\* data) {

printf("jsFunctionFinalizeCallback\n");

delete data;

}

jsValue jsObjectGetPropertyCallback(jsExecState es, jsValue object, const char\* propertyName) {

printf("jsObjectGetPropertyCallback:%s\n", propertyName);

jsData \*jsFuncData = new jsData;

::ZeroMemory(jsFuncData, sizeof(jsData));

jsFuncData->callAsFunction = jsFunctionCallback;

jsFuncData->finalize = jsFunctionFinalizeCallback;

return ::jsFunction(es, jsFuncData);

}

void jsObjectFinalizeCallback(jsData\* data) {

printf("jsObjectFinalizeCallback\n");

delete data;

}

void onDocumentReady(wkeWebView webView, void\* param, wkeWebFrameHandle frameId) {

jsExecState es = ::wkeGlobalExec(webView);

jsData \*jsObjData = new jsData;

::ZeroMemory(jsObjData, sizeof(jsData));

jsObjData->propertyGet = jsObjectGetPropertyCallback;

jsObjData->finalize = jsObjectFinalizeCallback;

jsValue jsObj = ::jsObject(es, jsObjData);

jsValue jsArgs[1] = { jsObj };

::jsCallGlobal(es, ::jsGetGlobal(es, "test"), jsArgs, 1);

}

这段代码演示了调用window.test方法，并传入一个C++构造的对象。注意的是这里用的是jsCallGlobal，和jsCall 的区别是jsCallGlobal调用window上的js函数，而jsCall更灵活些，可以调用任意js对象的成员函数。实际上，jsCallGlobal内部也是用jsCall实现的。

**模拟标题栏、拖动窗口**

实战中，经常需要把窗口设置为无边框窗口，但这样有个小问题：标题栏也没了，怎么拖动窗口？

解决方案很简单，你可以对某个你需要的div绑定一个js监听器：

divElement.addEventListener("mousedown", function() { jsCall("mousedown"); }, false);

收到mousedown消息后，用上面讲到的js通知c++方法，通知c++。再到c++中调用

::PostMessage(hWnd, WM\_SYSCOMMAND, SC\_MOVE | HTCAPTION, 0);

来实现模拟消息拖动窗口。