CEF 文档翻译-alpha版

CEF General Usage(CEF3预览)

介绍

CEF全称Chromium Embedded Framework,是一个基于Google Chromium 的开源项目。Google Chromium项目主要是为Google Chrome应用开发的，而CEF的目标则是为第三方应用提供可嵌入浏览器支持。CEF隔离底层Chromium和Blink的复杂代码，并提供一套产品级稳定的API，发布跟踪具体Chromium版本的分支，以及二进制包。CEF的大部分特性都提供了丰富的默认实现，让使用者做尽量少的定制即可满足需求。在本文发布的时候，世界上已经有很多公司和机构采用CEF，CEF的安装量超过了100万。[CEF wikipedia]页面上有使用CEF的公司和机构的不完全的列表。CEF的典型应用场景包括：

- 嵌入一个兼容HTML5的浏览器控件到一个已经存在的本地应用。
- 创建一个轻量化的壳浏览器，用以托管主要用Web技术开发的应用。
- 有些应用有独立的绘制框架，使用CEF对Web内容做离线渲染。
- 使用CEF做自动化Web测试。

CEF3是基于Chomuim Content API多进程构架的下一代CEF，拥有下列优势：

- 改进的性能和稳定性（JavaScript和插件在一个独立的进程内执行）。
- 支持Retina显示器
- 支持WebGL和3D CSS的GPU加速
- 类似WebRTC和语音输入这样的前卫特性。
- 通过DevTools远程调试协议以及ChromeDriver2提供更好的自动化UI测试
- 更快获得当前以及未来的Web特性和标准的能力

本文档介绍CEF3开发中涉及到的一般概念。

开始

使用二进制包

CEF3的二进制包可以在这个页面下载。其中包含了在特定平台（Windows，Mac OS X 以及Linux）编译特定版本CEF3所需的全部文件。不同平台拥有共同的结构：

- **cefclient**
- **Debug**
- **include**

- **libcef_dll**
- **Release**
- **Resources**
- **tools**

每个二进制包包含一个README.txt文件和一个LICENSE.txt文件，README.txt用以描述平台相关的细节，而LICENSE.txt包含CEF的BSD版权说明。如果你发布了基于CEF的应用，则应该应用程序的某个地方包含该版权声明。例如，你可以在”关于”和“授权”页面列出该版权声明，或者单独一个文档包含该版权声明。“关于”和“授权”信息也可以分别在CEF浏览器的”about:license”和”about:credits”页面查看。

基于CEF二进制包的应用程序可以使用每个平台上的经典编译工具。包括Windows平台上的Visual Studio，Mac OSX平台上的Xcode，以及Linux平台上的gcc/make编译工具链。CEF项目的下载页面包含了这些平台上编译特定版本CEF所需的编译工具的版本信息。在Linux上编译CEF时需要特别注意依赖工具链。

Tutorial Wiki页面有更多关于如何使用CEF3二进制包创建简单应用程序的细节。

从源码编译(Building from Source Code)

CEF可以从源码编译，用户可以使用本地编译系统或者像TeamCity这样的自动化编译系统编译。首先你需要使用svn或者git下载Chromium和CEF的源码。由于Chromium源码很大，只建议在内存大于4GB的现代机器上编译。编译Chromium和CEF的细节请参考BranchesAndBuilding页面。

示例应用程序(Sample Application)

cefclient是一个完整的CEF客户端应用程序示例，并且它的源码包含在CEF每个二进制发布包中。使用CEF创建一个新的应用程序，最简单的方法是先从cefclient应用程序开始，删除你不需要的部分。本文档中许多示例都是来源于cefclient应用程序。

重要概念(Important Concepts)

在开发基于CEF3的应用程序前，有一些重要的基础概念应该被理解。

C++ 封装(C++ Wrapper)

libcef 动态链接库导出 C API 使得使用者不用关心CEF运行库和基础代码。
libcef_dll_wrapper 工程把 C API 封装成 C++ API同时包含在客户端应用程序工程中，与cefclient一样，源代码作为CEF二进制发布包的一部份共同发布。C/C++ API的转换层代码是由转换工具自动生成。UsingTheCAPI 页面描述了如何使用C API。

进程(Processes)

CEF3是多进程架构的。”browser”被定义为主进程，负责窗口管理，界面绘制和网络交互。Blink的渲染和Js的执行被放在一个独立的”render”
进程中；除此之外，render进程还负责Js Binding和对Dom节点的访问。
默认的进程模型中，会为每个标签页创建一个新的”render”进程。其他进程按需创建，象管理

插件的进程和处理合成加速的进程。

默认情况下，主应用程序会被多次启动运行各自独立的进程。这是通过传递不同的命令行参数给CefExecuteProcess函数做到的。如果主应用程序很大，加载时间比较长，或者不能在非浏览器进程里使用，则宿主程序可使用独立的可执行文件去运行这些进程。这可以通过配置CefSettings.browser_subprocess_path变量做到。更多细节请参考`Application Structure`一节。

CEF3的进程之间可以通过IPC进行通信。”Browser”和”Render”进程可以通过发送异步消息进行双向通信。甚至在Render进程可以注册在Browser进程响应的异步JavaScript API。更多细节，请参考`Inter-Process Communication`一节。

通过设置命令行的”–single-process”，CEF3就可以支持用于调试目的的单进程运行模型。支持的平台为：Windows，Mac OS X 和Linux。

线程(Threads)

在CEF3中，每个进程都会运行多个线程。完整的线程类型表请参照cef_thread_id_t。例如，在browser进程中包含如下主要的线程：

- **TID_UI** 线程是浏览器的主线程。如果应用程序在调用调用CefInitialize()时，传递CefSettings.multi_threaded_message_loop=false，这个线程也是应用程序的主线程。
- **TID_IO** 线程主要负责处理IPC消息以及网络通信。
- **TID_FILE** 线程负责与文件系统交互。

由于CEF采用多线程架构，有必要使用锁和闭包来保证在多不同线程安全的传递数据。IMPLEMENT_LOCKING定义提供了Lock()和Unlock()方法以及AutoLock对象来保证不同代码块同步访问数据。CefPostTask函数组支持简易的线程间异步消息传递。更多信息，请参考”Posting Tasks”章节。

判断当前工作线程可以通过使用CefCurrentlyOn()方法，cefclient工程使用下面的定义来确保方法在期望的线程中被执行。

```
#define REQUIRE_UI_THREAD()   ASSERT(CefCurrentlyOn(TID_UI));
#define REQUIRE_IO_THREAD()   ASSERT(CefCurrentlyOn(TID_IO));
#define REQUIRE_FILE_THREAD() ASSERT(CefCurrentlyOn(TID_FILE));
```

引用计数(Reference Counting)

所有的框架类从CefBase继承，实例指针由CefRefPtr管理，CefRefPtr通过调用AddRef()和Release()方法自动管理引用计数。框架类的实现方式如下：

```
class MyClass : public CefBase {
 public:
  // Various class methods here...

 private:
  // Various class members here...
```

```
  IMPLEMENT_REFCOUNTING(MyClass);  // Provides atomic refcounting implementati
on.
};

// References a MyClass instance
CefRefPtr<MyClass> my_class = new MyClass();
```

字符串(Strings)

CEF为字符串定义了自己的数据结构。下面是这样做的理由：
- libcef包和宿主程序可能使用不同的运行时，对堆管理的方式也不同。所有的对象，包括字符串，需要确保和申请堆内存使用相同的运行时环境。
- libcef包可以编译为支持不同的字符串类型(UTF8，UTF16以及WIDE)。默认采用的是UTF16，默认字符集可以通过更改cef_string.h文件中的定义，然后重新编译来修改。当使用宽字节集的时候，切记字符的长度由当前使用的平台决定。

UTF16字符串结构体示例如下：

```
typedef struct _cef_string_utf16_t {
  char16* str;  // Pointer to the string
  size_t length;  // String length
  void (*dtor)(char16* str);  // Destructor for freeing the string on the corr
ect heap
} cef_string_utf16_t;
```

通过typedef来设置常用的字符编码。

```
typedef char16 cef_char_t;
typedef cef_string_utf16_t cef_string_t;
```

CEF提供了一批C语言的方法来操作字符串(通过#define的方式来适应不同的字符编码)

- **cef_string_set** 对制定的字符串变量赋值(支持深拷贝或浅拷贝)。
- **cef_string_clear** 清空字符串。
- **cef_string_cmp** 比较两个字符串.

CEF也提供了所有的字符编码的字符串格式之间相互转换的方法。具体函数列表请查阅cef_string.h和cef_string_types.h文件。

在C++中，通常使用CefString类来管理CEF的字符串。CefString支持与std::string(UTF8)、std::wstring(wide)类型的相互转换。也可以用来包裹一个cef_string_t结构来对其进行赋值。

和std::string的相互转换：

```
std::string str = "Some UTF8 string";

// Equivalent ways of assigning |str| to |cef_str|. Conversion from UTF8 will
occur if necessary.
```

```
CefString cef_str(str);
cef_str = str;
cef_str.FromString(str);

// Equivalent ways of assigning |cef_str| to |str|. Conversion to UTF8 will oc
cur if necessary.
str = cef_str;
str = cef_str.ToString();
```

和std::wstring的相互转换：

```
std::wstring str = “Some wide string”;

// Equivalent ways of assigning |str| to |cef_str|. Conversion from wide will
occur if necessary.
CefString cef_str(str);
cef_str = str;
cef_str.FromWString(str);

// Equivalent ways of assigning |cef_str| to |str|. Conversion to wide will oc
cur if necessary.
str = cef_str;
str = cef_str.ToWString();
```

如果是ASCII编码，使用FromASCII进行赋值：

```
const char* cstr = “Some ASCII string”;
CefString cef_str;
cef_str.FromASCII(cstr);
```

一些结构体(比如CefSettings)含有cef_string_t类型的成员，CefString支持直接赋值给这些成员。

```
CefSettings settings;
const char* path = “/path/to/log.txt”;

// Equivalent assignments.
CefString(&settings.log_file).FromASCII(path);
cef_string_from_ascii(path, strlen(path), &settings.log_file);
```

命令行参数(Command Line Arguments)

在CEF3和Chromium中许多特性可以使用命令行参数进行配置。这些参数采用”–some-argument[=optional-param]”形式，并通过CefExecuteProcess()和CefMainArgs结构（参考下面的”应用资源布局”章节）传递给CEF。在传递CefSettings结构给CefInitialize()之前，我们可以设置CefSettings.command_line_args_disabled为false来禁用对命令行参数的处理。如果想指定命令行参数传入主应用程序，实现CefApp::OnBeforeCommandLineProcessing()方法。更多关于如何查找已支持的命令行选

项的信息，请查看client_switches.cpp文件的注释。

应用程序文件结构(Application Layout)

应用资源布局依赖于平台，有很大的不同。比如，在Mac OS X上，你的资源布局必须遵循特定的app bundles结构；Window与Linux则更灵活，允许你定制CEF库文件与资源文件所在的位置。为了获取到特定地、可以正常工作的示例，你可以从工程的下载页面，下载到一个client压缩包，每个平台对应的README.txt文件详细说明了哪些文件是可选的，哪些文件是必须的。

Windows操作系统(Windows)

在Windows平台上，默认的资源布局将libcef库文件、相关资源与可执行文件放置在同级目录，文件夹结构如下类似：

```
Application/
    cefclient.exe  <= cefclient application executable
    libcef.dll <= main CEF library
    icudt.dll <= ICU unicode support library
    ffmpegsumo.dll <= HTML5 audio/video support library
    libEGL.dll, libGLESv2.dll, … <= accelerated compositing support libraries
    cef.pak, devtools_resources.pak <= non-localized resources and strings
    locales/
        en-US.pak, … <= locale-specific resources and strings
```

使用结构体CefSettings可以定制CEF库文件、资源文件的位置（查看README.txt文件或者本文中CefSettings部分获取更详细的信息）。虽然在Windows平台上，cefclient项目将资源文件以二进制形式编译进cefclient.rc文件，但是改为从文件系统加载资源也很容易。

Linux操作系统(Linux)

在Linux平台上，默认的资源布局将libcef库文件、相关资源与可执行文件放置在同级目录。注意：在你编译的版本与发行版本应用程序中，libcef.so的位置是有差异的，此文件的位置取决于编译可执行程序时，编译器rpath的值。比如，编译选项为“-Wl,-rpath,.”（“.”意思是当前文件夹），这样libcef.so与可执行文件处于同级目录。libcef.so文件的路径可以通过环境变量中的“`LD_LIBRARY_PATH`”指定。

```
Application/
    cefclient  <= cefclient application executable
    libcef.so <= main CEF library
    ffmpegsumo.so <-- HTML5 audio/video support library
    cef.pak, devtools_resources.pak <= non-localized resources and strings
    locales/
        en-US.pak, … <= locale-specific resources and strings
    files/
        binding.html, … <= cefclient application resources
```

使用结构体CefSettings可以定制CEF库文件、资源文件（查看README.txt文件或者本文中CefSettings部分获取更详细的信息）。

Mac X平台(Mac OS X)

在Mac X平台上，app bundles委托给了Chromium实现，因此不是很灵活。文件夹结构如下类似：

```
cefclient.app/
    Contents/
        Frameworks/
            Chromium Embedded Framework.framework/
                Libraries/
                    ffmpegsumo.so <= HTML5 audio/video support library
                    libcef.dylib <= main CEF library
                Resources/
                    cef.pak, devtools_resources.pak <= non-localized resources
and strings
                    *.png, *.tiff <= Blink image and cursor resources
                    en.lproj/, … <= locale-specific resources and strings
            libplugin_carbon_interpose.dylib <= plugin support library
            cefclient Helper.app/
                Contents/
                    Info.plist
                    MacOS/
                        cefclient Helper <= helper executable
                    Pkginfo
            cefclient Helper EH.app/
                Contents/
                    Info.plist
                    MacOS/
                        cefclient Helper EH <= helper executable
                    Pkginfo
            cefclient Helper NP.app/
                Contents/
                    Info.plist
                    MacOS/
                        cefclient Helper NP <= helper executable
                    Pkginfo
        Info.plist
        MacOS/
            cefclient <= cefclient application executable
        Pkginfo
        Resources/
            binding.html, … <= cefclient application resources
```

列表中的“Chromium Embedded Framework.framework”，这个未受版本管控的框架包含了所有的CEF库文件、资源文件。使用install_name_tool与@executable_path，将cefclient，cefclient helper等可执行文件，连接到了libcef.dylib上。

应用程序cefclient helper用来执行不同特点、独立的进程（renderer，plugin等），这些进程需要独立的资源布局与Info.plist等文件，它们没有显示停靠图标。用来启动插件进程的EH Helper清除了MH_NO_HEAP_EXECUTION标志位，这样就允许一个可执行堆。只能用来启动NaCL插件进程的NP Helper，清除了MH_PIE标志位，这样就禁用了ASLR。这些都是tools文

件夹下面，用来构建进程脚本的一部分。为了理清脚本的依赖关系，更好的做法是检查发行版本中的Xcode工程或者原始文件cefclient.gyp。

应用程序结构(Application Structure)

每个CEF3应用程序都是相同的结构

- 提供入口函数，用于初始化CEF、运行子进程执行逻辑或者CEF消息循环。
- 提供CefApp实现，用于处理进程相关的回调。
- 提供CefClient实现，用于处理browser实例相关的回调
- 执行CefBrowserHost::CreateBrowser()创建一个browser实例，使用CefLifeSpanHandler管理browser对象生命周期。

入口函数(Entry-Point Function)

像本文中进程章节描述的那样，一个CEF3应用程序会运行多个进程，这些进程能够使用同一个执行器或者为子进程定制的、单独的执行器。进程的执行从入口函数开始，示例cefclient_win.cc、cefclient_gtk.cc、cefclient_mac.mm分别对应Windows、Linux和Mac OS-X平台下的实现。

当执行子进程是，CEF将使用命令行参数指定配置信息，这些命令行参数必须通过CefMainArgs结构体传入到CefExecuteProcess函数。CefMainArgs的定义与平台相关，在Linux、Mac OS X平台下，它接收main函数传入的argc和argv参数值。

```
CefMainArgs main_args(argc, argv);
```

在Windows平台下，它接收wWinMain函数传入的参数：实例句柄（HINSTANCE），这个实例能够通过函数GetModuleHandle(NULL)获取。

```
CefMainArgs main_args(hInstance);
```

单一执行体(Single Executable)

当以单一执行体运行时，根据不同的进程类型，入口函数有差异。Windows、Linux平台支持单一执行体架构，Mac OS X平台则不行。

```
int main(int argc, char* argv[]) {
  // Structure for passing command-line arguments.
  // The definition of this structure is platform-specific.
  CefMainArgs main_args(argc, argv);

  // Optional implementation of the CefApp interface.
  CefRefPtr<MyApp> app(new MyApp);

  // Execute the sub-process logic, if any. This will either return immediatel
y for the browser
  // process or block until the sub-process should exit.
  int exit_code = CefExecuteProcess(main_args, app.get());
```

```
  if (exit_code >= 0) {
    // The sub-process terminated, exit now.
    return exit_code;
  }

  // Populate this structure to customize CEF behavior.
  CefSettings settings;

  // Initialize CEF in the main process.
  CefInitialize(main_args, settings, app.get());

  // Run the CEF message loop. This will block until CefQuitMessageLoop() is c
alled.
  CefRunMessageLoop();

  // Shut down CEF.
  CefShutdown();

  return 0;
}
```

独立的子进程执行体(Separate Sub-Process Executable)

当使用独立的子进程执行体时，你需要2个分开的可执行工程和2个分开的入口函数。

主程序的入口函数：

```
// Program entry-point function.
// 程序入口函数
int main(int argc, char* argv[]) {
  // Structure for passing command-line arguments.
  // The definition of this structure is platform-specific.
  // 传递命令行参数的结构体。
  // 这个结构体的定义与平台相关。
  CefMainArgs main_args(argc, argv);

  // Optional implementation of the CefApp interface.
  // 可选择性地实现CefApp接口
  CefRefPtr<MyApp> app(new MyApp);

  // Populate this structure to customize CEF behavior.
  // 填充这个结构体，用于定制CEF的行为。
  CefSettings settings;

  // Specify the path for the sub-process executable.
  // 指定子进程的执行路径
  CefString(&settings.browser_subprocess_path).FromASCII("/path/to/subprocess"
);

  // Initialize CEF in the main process.
  // 在主进程中初始化CEF
  CefInitialize(main_args, settings, app.get());
```

```
  // Run the CEF message loop. This will block until CefQuitMessageLoop() is c
alled.
  // 执行消息循环，此时会堵塞，直到CefQuitMessageLoop()函数被调用。
  CefRunMessageLoop();

  // Shut down CEF.
  // 关闭CEF
  CefShutdown();

  return 0;
}
```

子进程程序的入口函数：

```
// Program entry-point function.
// 程序入口函数
int main(int argc, char* argv[]) {
  // Structure for passing command-line arguments.
  // The definition of this structure is platform-specific.
  // 传递命令行参数的结构体。
  // 这个结构体的定义与平台相关。
  CefMainArgs main_args(argc, argv);

  // Optional implementation of the CefApp interface.
  // 可选择性地实现CefApp接口
  CefRefPtr<MyApp> app(new MyApp);

  // Execute the sub-process logic. This will block until the sub-process shou
ld exit.
  // 执行子进程逻辑，此时会堵塞直到子进程退出。
  return CefExecuteProcess(main_args, app.get());
}
```

集成消息循环(Message Loop Integration)

CEF可以不用它自己提供的消息循环，而与已经存在的程序中消息环境集成在一起，有两种方式可以做到：

1. 周期性执行CefDoMessageLoopWork()函数，替代调用CefRunMessageLoop()。CefDoMessageLoopWork()的每一次调用，都将执行一次CEF消息循环的单次迭代。需要注意的是，此方法调用次数太少时，CEF消息循环会饿死，将极大的影响browser的性能，调用次数太频繁又将影响CPU使用率。

2. 设置CefSettings.multi_threaded_message_loop=true（Windows平台下有效），这个设置项将导致CEF运行browser UI运行在单独的线程上，而不是在主线程上，这种场景下CefDoMessageLoopWork()或者CefRunMessageLoop()都不需要调用，CefInitialze()、CefShutdown()仍然在主线程中调用。你需要提供主程序线程通信的机制（查看cefclient_win.cpp中提供的消息窗口实例）。在Windows平台下，你可以通过命令行参数“–multi-threaded-message-loop”测试上述消息模型。

CefSettings

CefSettings结构体允许定义全局的CEF配置，经常用到的配置项如下：

- **single_process** 设置为true时，browser和renderer使用一个进程。此项也可以通过命令行参数“single-process”配置。查看本文中“进程”章节获取更多的信息。
- **browser_subprocess_path** 设置用于启动子进程单独执行器的路径。参考本文中“单独子进程执行器”章节获取更多的信息。
- **cache_path** 设置磁盘上用于存放缓存数据的位置。如果此项为空，某些功能将使用内存缓存，多数功能将使用临时的磁盘缓存。形如本地存储的HTML5数据库只能在设置了缓存路径才能跨session存储。
- **locale** 此设置项将传递给Blink。如果此项为空，将使用默认值“en-US”。在Linux平台下此项被忽略，使用环境变量中的值，解析的依次顺序为：LANGUAE，LC_ALL，LC_MESSAGES和LANG。此项也可以通过命令行参数“lang”配置。
- **log_file** 此项设置的文件夹和文件名将用于输出debug日志。如果此项为空，默认的日志文件名为debug.log，位于应用程序所在的目录。此项也可以通过命令参数“log-file”配置。
- **log_severity** 此项设置日志级别。只有此等级、或者比此等级高的日志的才会被记录。此项可以通过命令行参数“log-severity”配置，可以设置的值为“verbose”，“info”，“warning”，“error”，“error-report”，“disable”
- **resources_dir_path** 此项设置资源文件夹的位置。如果此项为空，Windows平台下cef.pak、Linux平台下devtools_resourcs.pak、Mac OS X下的app bundle Resources目录必须位于组件目录。此项也可以通过命令行参数“resource-dir-path”配置。
- **locales_dir_path** 此项设置locale文件夹位置。如果此项为空，locale文件夹必须位于组件目录，在Mac OS X平台下此项被忽略，pak文件从app bundle Resources目录。此项也可以通过命令行参数“locales-dir-path”配置。
- **remote_debugging_port** 此项可以设置1024-65535之间的值，用于在指定端口开启远程调试。例如，如果设置的值为8080，远程调试的URL为http://localhost:8080。CEF或者Chrome浏览器能够调试CEF。此项也可以通过命令行参数“remote-debugging-port”配置。

CefBrowser和CefFrame

CefBrowser和CefFrame对象被用来发送命令给浏览器以及在回调函数里获取状态信息。每个CefBrowser对象包含一个主CefFrame对象，主CefFrame对象代表页面的顶层frame；同时每个CefBrowser对象可以包含零个或多个的CefFrame对象，分别代表不同的子Frame。例如，一个浏览器加载了两个iframe，则该CefBrowser对象拥有三个CefFrame对象（顶层frame和两个iframe）。

下面的代码在浏览器的主frame里加载一个URL：

```
browser->GetMainFrame()->LoadURL(some_url);
```

下面的代码执行浏览器的回退操作：

```
browser->GoBack();
```

下面的代码从主frame里获取HTML内容：

```
// Implementation of the CefStringVisitor interface.
class Visitor : public CefStringVisitor {
 public:
  Visitor() {}

  // Called asynchronously when the HTML contents are available.
  virtual void Visit(const CefString& string) OVERRIDE {
    // Do something with |string|...
  }

  IMPLEMENT_REFCOUNTING(Visitor);
};

browser->GetMainFrame()->GetSource(new Visitor());
```

CefBrowser和CefFrame对象在Browser进程和Render进程都有对等的代理对象。在Browser进程里，Host（宿主）行为控制可以通过CefBrowser::GetHost()方法控制。例如，浏览器窗口的原生句柄可以用下面的代码获取：

```
// CefWindowHandle is defined as HWND on Windows, NSView* on Mac OS X
// and GtkWidget* on Linux.
CefWindowHandle window_handle = browser->GetHost()->GetWindowHandle();
```

其他方法包括历史导航，加载字符串和请求，发送编辑命令，提取text/html内容等。请参考支持函数相关的文档或者CefBrowser的头文件注释。

CefApp

CefApp接口提供了不同进程的可定制回调函数。毕竟重要的回调函数如下：

- **OnBeforeCommandLineProcessing** 提供了以编程方式设置命令行参数的机会，更多细节，请参考 `Command Line Arguments` 一节。
- **OnRegisterCustomSchemes** 提供了注册自定义schemes的机会，更多细节，请参考 `Request Handling` 一节。
- **GetBrowserProcessHandler** 返回定制Browser进程的Handler，该Handler包括了诸如OnContextInitialized的回调。
- **GetRenderProcessHandler** 返回定制Render进程的Handler，该Handler包含了JavaScript相关的一些回调以及消息处理的回调。更多细节，请参考 `JavascriptIntegration` 和 `Inter-Process Communication` 两节。

CefApp子类的例子：

```
// MyApp implements CefApp and the process-specific interfaces.
class MyApp : public CefApp,
              public CefBrowserProcessHandler,
```

```
                public CefRenderProcessHandler {
 public:
  MyApp() {}

  // CefApp methods. Important to return |this| for the handler callbacks.
  virtual void OnBeforeCommandLineProcessing(
      const CefString& process_type,
      CefRefPtr<CefCommandLine> command_line) {
    // Programmatically configure command-line arguments...
  }
  virtual void OnRegisterCustomSchemes(
      CefRefPtr<CefSchemeRegistrar> registrar) OVERRIDE {
    // Register custom schemes...
  }
  virtual CefRefPtr<CefBrowserProcessHandler> GetBrowserProcessHandler()
      OVERRIDE { return this; }
  virtual CefRefPtr<CefRenderProcessHandler> GetRenderProcessHandler()
      OVERRIDE { return this; }

  // CefBrowserProcessHandler methods.
  virtual void OnContextInitialized() OVERRIDE {
    // The browser process UI thread has been initialized...
  }
  virtual void OnRenderProcessThreadCreated(CefRefPtr<CefListValue> extra_info
)
                                            OVERRIDE {
    // Send startup information to a new render process...
  }

  // CefRenderProcessHandler methods.
  virtual void OnRenderThreadCreated(CefRefPtr<CefListValue> extra_info)
                                     OVERRIDE {
    // The render process main thread has been initialized...
    // Receive startup information in the new render process...
  }
  virtual void OnWebKitInitialized(CefRefPtr<ClientApp> app) OVERRIDE {
    // WebKit has been initialized, register V8 extensions...
  }
  virtual void OnBrowserCreated(CefRefPtr<CefBrowser> browser) OVERRIDE {
    // Browser created in this render process...
  }
  virtual void OnBrowserDestroyed(CefRefPtr<CefBrowser> browser) OVERRIDE {
    // Browser destroyed in this render process...
  }
  virtual bool OnBeforeNavigation(CefRefPtr<CefBrowser> browser,
                                  CefRefPtr<CefFrame> frame,
                                  CefRefPtr<CefRequest> request,
                                  NavigationType navigation_type,
                                  bool is_redirect) OVERRIDE {
    // Allow or block different types of navigation...
  }
  virtual void OnContextCreated(CefRefPtr<CefBrowser> browser,
                                CefRefPtr<CefFrame> frame,
```

```
                                CefRefPtr<CefV8Context> context) OVERRIDE {
    // JavaScript context created, add V8 bindings here...
  }
  virtual void OnContextReleased(CefRefPtr<CefBrowser> browser,
                                 CefRefPtr<CefFrame> frame,
                                 CefRefPtr<CefV8Context> context) OVERRIDE {
    // JavaScript context released, release V8 references here...
  }
  virtual bool OnProcessMessageReceived(
      CefRefPtr<CefBrowser> browser,
      CefProcessId source_process,
      CefRefPtr<CefProcessMessage> message) OVERRIDE {
    // Handle IPC messages from the browser process...
  }

  IMPLEMENT_REFCOUNTING(MyApp);
};
```

CefClient

CefClient提供访问browser-instance-specific的回调接口。单实例CefClient可以共数任意数量的浏览器进程。以下为几个重要的回调：

- Handlers for things like browser life span, context menus, dialogs, display notifications, drag events, focus events, keyboard events and more. The majority of handlers are optional. See the documentation in cef_client.h for the side effects, if any, of not implementing a specific handler.
- **OnProcessMessageReceived** which is called when an IPC message is received from the render process. See the “Inter-Process Communication” section for more information.

- 比如处理browser的生命周期，右键菜单，对话框，通知显示， 拖曳事件，焦点事件，键盘事件等等。如果没有对某个特定的处理接口进行实现会造成什么影响，请查看cef_client.h文件中相关说明。

- **OnProcessMessageReceived**在Render进程收到进程间消息时被调用。更多细节，请参考`Inter-Process Communication`一节。

CefClient子类的例子：

```
// MyHandler implements CefClient and a number of other interfaces.
class MyHandler : public CefClient,
                  public CefContextMenuHandler,
                  public CefDisplayHandler,
                  public CefDownloadHandler,
                  public CefDragHandler,
                  public CefGeolocationHandler,
                  public CefKeyboardHandler,
                  public CefLifeSpanHandler,
                  public CefLoadHandler,
                  public CefRequestHandler {
```

```
public:
 MyHandler();

 // CefClient methods. Important to return |this| for the handler callbacks.
 virtual CefRefPtr<CefContextMenuHandler> GetContextMenuHandler() OVERRIDE {
   return this;
 }
 virtual CefRefPtr<CefDisplayHandler> GetDisplayHandler() OVERRIDE {
   return this;
 }
 virtual CefRefPtr<CefDownloadHandler> GetDownloadHandler() OVERRIDE {
   return this;
 }
 virtual CefRefPtr<CefDragHandler> GetDragHandler() OVERRIDE {
   return this;
 }
 virtual CefRefPtr<CefGeolocationHandler> GetGeolocationHandler() OVERRIDE {
   return this;
 }
 virtual CefRefPtr<CefKeyboardHandler> GetKeyboardHandler() OVERRIDE {
   return this;
 }
 virtual CefRefPtr<CefLifeSpanHandler> GetLifeSpanHandler() OVERRIDE {
   return this;
 }
 virtual CefRefPtr<CefLoadHandler> GetLoadHandler() OVERRIDE {
   return this;
 }
 virtual CefRefPtr<CefRequestHandler> GetRequestHandler() OVERRIDE {
   return this;
 }
 virtual bool OnProcessMessageReceived(CefRefPtr<CefBrowser> browser,
                                       CefProcessId source_process,
                                       CefRefPtr<CefProcessMessage> message)
                                       OVERRIDE {
   // Handle IPC messages from the render process...
 }

 // CefContextMenuHandler methods
 virtual void OnBeforeContextMenu(CefRefPtr<CefBrowser> browser,
                                  CefRefPtr<CefFrame> frame,
                                  CefRefPtr<CefContextMenuParams> params,
                                  CefRefPtr<CefMenuModel> model) OVERRIDE {
   // Customize the context menu...
 }
 virtual bool OnContextMenuCommand(CefRefPtr<CefBrowser> browser,
                                   CefRefPtr<CefFrame> frame,
                                   CefRefPtr<CefContextMenuParams> params,
                                   int command_id,
                                   EventFlags event_flags) OVERRIDE {
   // Handle a context menu command...
 }
```

```
    // CefDisplayHandler methods
    virtual void OnLoadingStateChange(CefRefPtr<CefBrowser> browser,
                                      bool isLoading,
                                      bool canGoBack,
                                      bool canGoForward) OVERRIDE {
      // Update UI for browser state...
    }
    virtual void OnAddressChange(CefRefPtr<CefBrowser> browser,
                                 CefRefPtr<CefFrame> frame,
                                 const CefString& url) OVERRIDE {
      // Update the URL in the address bar...
    }
    virtual void OnTitleChange(CefRefPtr<CefBrowser> browser,
                               const CefString& title) OVERRIDE {
      // Update the browser window title...
    }
    virtual bool OnConsoleMessage(CefRefPtr<CefBrowser> browser,
                                  const CefString& message,
                                  const CefString& source,
                                  int line) OVERRIDE {
      // Log a console message...
    }

    // CefDownloadHandler methods
    virtual void OnBeforeDownload(
        CefRefPtr<CefBrowser> browser,
        CefRefPtr<CefDownloadItem> download_item,
        const CefString& suggested_name,
        CefRefPtr<CefBeforeDownloadCallback> callback) OVERRIDE {
      // Specify a file path or cancel the download...
    }
    virtual void OnDownloadUpdated(
        CefRefPtr<CefBrowser> browser,
        CefRefPtr<CefDownloadItem> download_item,
        CefRefPtr<CefDownloadItemCallback> callback) OVERRIDE {
      // Update the download status...
    }

    // CefDragHandler methods
    virtual bool OnDragEnter(CefRefPtr<CefBrowser> browser,
                             CefRefPtr<CefDragData> dragData,
                             DragOperationsMask mask) OVERRIDE {
      // Allow or deny drag events...
    }

    // CefGeolocationHandler methods
    virtual void OnRequestGeolocationPermission(
        CefRefPtr<CefBrowser> browser,
        const CefString& requesting_url,
        int request_id,
        CefRefPtr<CefGeolocationCallback> callback) OVERRIDE {
      // Allow or deny geolocation API access...
    }
```

```
// CefKeyboardHandler methods
virtual bool OnPreKeyEvent(CefRefPtr<CefBrowser> browser,
                           const CefKeyEvent& event,
                           CefEventHandle os_event,
                           bool* is_keyboard_shortcut) OVERRIDE {
  // Perform custom handling of key events...
}

// CefLifeSpanHandler methods
virtual bool OnBeforePopup(CefRefPtr<CefBrowser> browser,
                           CefRefPtr<CefFrame> frame,
                           const CefString& target_url,
                           const CefString& target_frame_name,
                           const CefPopupFeatures& popupFeatures,
                           CefWindowInfo& windowInfo,
                           CefRefPtr<CefClient>& client,
                           CefBrowserSettings& settings,
                           bool* no_javascript_access) OVERRIDE {
  // Allow or block popup windows, customize popup window creation...
}
virtual void OnAfterCreated(CefRefPtr<CefBrowser> browser) OVERRIDE {
  // Browser window created successfully...
}
virtual bool DoClose(CefRefPtr<CefBrowser> browser) OVERRIDE {
  // Allow or block browser window close...
}
virtual void OnBeforeClose(CefRefPtr<CefBrowser> browser) OVERRIDE {
  // Browser window is closed, perform cleanup...
}

// CefLoadHandler methods
virtual void OnLoadStart(CefRefPtr<CefBrowser> browser,
                         CefRefPtr<CefFrame> frame) OVERRIDE {
  // A frame has started loading content...
}
virtual void OnLoadEnd(CefRefPtr<CefBrowser> browser,
                       CefRefPtr<CefFrame> frame,
                       int httpStatusCode) OVERRIDE {
  // A frame has finished loading content...
}
virtual void OnLoadError(CefRefPtr<CefBrowser> browser,
                         CefRefPtr<CefFrame> frame,
                         ErrorCode errorCode,
                         const CefString& errorText,
                         const CefString& failedUrl) OVERRIDE {
  // A frame has failed to load content...
}
virtual void OnRenderProcessTerminated(CefRefPtr<CefBrowser> browser,
                                       TerminationStatus status) OVERRIDE {
  // A render process has crashed...
}
```

```
  // CefRequestHandler methods
  virtual CefRefPtr<CefResourceHandler> GetResourceHandler(
      CefRefPtr<CefBrowser> browser,
      CefRefPtr<CefFrame> frame,
      CefRefPtr<CefRequest> request) OVERRIDE {
    // Optionally intercept resource requests...
  }
  virtual bool OnQuotaRequest(CefRefPtr<CefBrowser> browser,
                              const CefString& origin_url,
                              int64 new_size,
                              CefRefPtr<CefQuotaCallback> callback) OVERRIDE {
    // Allow or block quota requests...
  }
  virtual void OnProtocolExecution(CefRefPtr<CefBrowser> browser,
                                   const CefString& url,
                                   bool& allow_os_execution) OVERRIDE {
    // Handle execution of external protocols...
  }

  IMPLEMENT_REFCOUNTING(MyHandler);
};
```

Browser生命周期(Browser Life Span)

Browser生命周期从执行 CefBrowserHost::CreateBrowser() 或者 CefBrowserHost::CreateBrowserSync() 开始。可以在 CefBrowserProcessHandler::OnContextInitialized() 回调或者特殊平台例如windows的 WM_CREATE 中方便的执行业务逻辑。

```
// Information about the window that will be created including parenting, size
, etc.
// The definition of this structure is platform-specific.

// 定义的结构体与平台相关

CefWindowInfo info;
// On Windows for example...
info.SetAsChild(parent_hwnd, client_rect);

// Customize this structure to control browser behavior.
CefBrowserSettings settings;

// CefClient implementation.
CefRefPtr<MyClient> client(new MyClient);

// Create the browser asynchronously. Initially loads the Google URL.
CefBrowserHost::CreateBrowser(info, client.get(), “http://www.google.com”, set
tings);

The CefLifeSpanHandler class provides the callbacks necessary for managing bro
wser life span. Below is an extract of the relevant methods and members.
```

```
CefLifeSpanHandler 类提供管理 browser生合周期必需的回调。以下为相关方法和成员。

class MyClient : public CefClient,
                 public CefLifeSpanHandler,
                 ... {
  // CefClient methods.
  virtual CefRefPtr<CefLifeSpanHandler> GetLifeSpanHandler() OVERRIDE {
    return this;
  }

  // CefLifeSpanHandler methods.
  void OnAfterCreated(CefRefPtr<CefBrowser> browser) OVERRIDE;
  bool DoClose(CefRefPtr<CefBrowser> browser) OVERRIDE;
  void OnBeforeClose(CefRefPtr<CefBrowser> browser) OVERRIDE;

  // Member accessors.
  CefRefPtr<CefBrowser> GetBrower() { return m_Browser; }
  bool IsClosing() { return m_bIsClosing; }

 private:
  CefRefPtr<CefBrowser> m_Browser;
  int m_BrowserId;
  int m_BrowserCount;
  bool m_bIsClosing;

  IMPLEMENT_REFCOUNTING(MyHandler);
  IMPLEMENT_LOCKING(MyHandler);
};
```

当browser对象创建后OnAfterCreated() 方法立即执行。宿主程序可以用这个方法来保持对browser对象的引用。

```
void MyClient::OnAfterCreated(CefRefPtr<CefBrowser> browser) {
  // Must be executed on the UI thread.
  REQUIRE_UI_THREAD();
  // Protect data members from access on multiple threads.
  AutoLock lock_scope(this);

  if (!m_Browser.get())   {
    // Keep a reference to the main browser.
    m_Browser = browser;
    m_BrowserId = browser->GetIdentifier();
  }

  // Keep track of how many browsers currently exist.
  m_BrowserCount++;
}
```

执行CefBrowserHost::CloseBrowser()销毁browser对象。

```
// Notify the browser window that we would like to close it. This will result
```

```
in a call to
// MyHandler::DoClose() if the JavaScript 'onbeforeunload' event handler allow
s it.
browser->GetHost()->CloseBrowser(false);
```

browser对象的关闭事件来源于他的父窗口的关闭方法（比如，在父窗口上点击X控钮。）。父窗口需要调用 CloseBrowser(false) 并且等待操作系统的第二个关闭事件来决定是否允许关闭。如果在JavaScript ‘onbeforeunload’事件处理或者 DoClose()回调中取消了关闭操作，则操作系统的第二个关闭事件可能不会发送。注意一下面示例中对IsCloseing()的判断-它在第一个关闭事件中返回false，在第二个关闭事件中返回true(当 DoCloase 被调用后)。

Windows平台下，在父窗口的WndProc里处理WM_ClOSE消息：

```
case WM_CLOSE:
  if (g_handler.get() && !g_handler->IsClosing()) {
    CefRefPtr<CefBrowser> browser = g_handler->GetBrowser();
    if (browser.get()) {
      // Notify the browser window that we would like to close it. This will r
esult in a call to
      // MyHandler::DoClose() if the JavaScript 'onbeforeunload' event handler
allows it.
      browser->GetHost()->CloseBrowser(false);

      // Cancel the close.
      return 0;
    }
  }

  // Allow the close.
  break;

case WM_DESTROY:
  // Quitting CEF is handled in MyHandler::OnBeforeClose().
  return 0;
}
```

Linux平台下，处理`delete_event`信号:

```
gboolean delete_event(GtkWidget* widget, GdkEvent* event,
                      GtkWindow* window) {
  if (g_handler.get() && !g_handler->IsClosing()) {
    CefRefPtr<CefBrowser> browser = g_handler->GetBrowser();
    if (browser.get()) {
      // Notify the browser window that we would like to close it. This will r
esult in a call to
      // MyHandler::DoClose() if the JavaScript 'onbeforeunload' event handler
allows it.
      browser->GetHost()->CloseBrowser(false);

      // Cancel the close.
      return TRUE;
```

```
    }
  }

  // Allow the close.
  return FALSE;
}
```

MacOS X平台下，处理windowShouldClose选择器:

```
// Called when the window is about to close. Perform the self-destruction
// sequence by getting rid of the window. By returning YES, we allow the windo
w
// to be removed from the screen.
- (BOOL)windowShouldClose:(id)window {
  if (g_handler.get() && !g_handler->IsClosing()) {
    CefRefPtr<CefBrowser> browser = g_handler->GetBrowser();
    if (browser.get()) {
      // Notify the browser window that we would like to close it. This will r
esult in a call to
      // MyHandler::DoClose() if the JavaScript 'onbeforeunload' event handler
allows it.
      browser->GetHost()->CloseBrowser(false);

      // Cancel the close.
      return NO;
    }
  }

  // Try to make the window go away.
  [window autorelease];

  // Clean ourselves up after clearing the stack of anything that might have t
he
  // window on it.
  [self performSelectorOnMainThread:@selector(cleanup:)
                         withObject:window
                      waitUntilDone:NO];

  // Allow the close.
  return YES;
}
```

DoClose方法设置m_bIsClosing 标志位为true，并返回false以再次发送操作系统的关闭事件。

```
bool MyClient::DoClose(CefRefPtr<CefBrowser> browser) {
  // Must be executed on the UI thread.
  REQUIRE_UI_THREAD();
  // Protect data members from access on multiple threads.
  AutoLock lock_scope(this);
```

```
  // Closing the main window requires special handling. See the DoClose()
  // documentation in the CEF header for a detailed description of this
  // process.
  if (m_BrowserId == browser->GetIdentifier()) {
    // Notify the browser that the parent window is about to close.
    browser->GetHost()->ParentWindowWillClose();

    // Set a flag to indicate that the window close should be allowed.
    m_bIsClosing = true;
  }

  // Allow the close. For windowed browsers this will result in the OS close
  // event being sent.
  return false;
}
```

当操作系统捕捉到第二次关闭事件，它才会允许父窗口真正关闭。该动作会先触发OnBeforeClose()回调，请在该回调里释放所有对浏览器对象的引用。

```
void MyHandler::OnBeforeClose(CefRefPtr<CefBrowser> browser) {
  // Must be executed on the UI thread.
  REQUIRE_UI_THREAD();
  // Protect data members from access on multiple threads.
  AutoLock lock_scope(this);

  if (m_BrowserId == browser->GetIdentifier()) {
    // Free the browser pointer so that the browser can be destroyed.
    m_Browser = NULL;
  }

  if (--m_BrowserCount == 0) {
    // All browser windows have closed. Quit the application message loop.
    CefQuitMessageLoop();
  }
}
```

完整的流程，请参考cefclient例子里对不同平台的处理。

离屏渲染(Off-Screen Rendering)

在离屏渲染模式下，CEF不会创建原生浏览器窗口。CEF为宿主程序提供无效的区域和像素缓存区，而宿主程序负责通知鼠标键盘以及焦点事件给CEF。离屏渲染目前不支持混合加速，所以性能上可能无法和非离屏渲染相比。离屏浏览器将收到和窗口浏览器同样的事件通知，例如前一节介绍的生命周期事件。下面介绍如何使用离屏渲染：

- 实现CefRenderHandler接口。除非特别说明，所有的方法都需要覆写。
- 调用CefWindowInfo::SetAsOffScreen()，将CefWindowInfo传递给CefBrowserHost::CreateBrowser()之前还可以选择设置CefWindowInfo::SetTransparentPainting()。如果没有父窗口被传递给SetAsOffScreen,则有些类似上下文菜单这样的功能将不可用。

- CefRenderHandler::GetViewRect方法将被调用以获得所需要的可视区域。
- CefRenderHandler::OnPaint() 方法将被调用以提供无效区域（脏区域）以及更新过的像素缓存。cefclient程序里使用OpenGL绘制缓存，但你可以使用任何别的绘制技术。
- 可以调用CefBrowserHost::WasResized()方法改变浏览器大小。这将导致对GetViewRect()方法的调用，以获取新的浏览器大小，然后调用OnPaint()重新绘制。
- 调用CefBrowserHost::SendXXX()方法通知浏览器的鼠标、键盘和焦点事件。
- 调用CefBrowserHost::CloseBrowser()销毁浏览器。

使用命令行参数 `--off-screen-rendering-enabled` 运行cefclient，可以测试离屏渲染的效果。

投递任务(Posting Tasks)

任务(Task)可以通过CefPostTask在一个进程内的不同的线程之间投递。CefPostTask有一系列的重载方法，详细内容请参考cef_task.h头文件。任务将会在被投递线程的消息循环里异步执行。例如，为了在UI线程上执行MyObject::MyMethod方法，并传递两个参数，代码如下：

```
CefPostTask(TID_UI, NewCefRunnableMethod(object, &MyObject::MyMethod, param1,
param2));
```

为了在IO线程在执行MyFunction方法，同时传递两个参数，代码如下：

```
CefPostTask(TID_IO, NewCefRunnableFunction(MyFunction, param1, param2));
```

参考cef_runnable.h头文件以了解更多关于NewCefRunnable模板方法的细节。

如果宿主程序需要保留一个运行循环的引用，则可以使用CefTaskRunner类。例如，获取UI线程的任务运行器(task runner)，代码如下：

```
CefRefPtr<CefTaskRunner> task_runner = CefTaskRunner::GetForThread(TID_UI);
```

进程间通信(Inter-Process Communication (IPC))

由于CEF3运行在多进程环境下，所以需要提供一个进程间通信机制。CefBrowser和CefFrame对象在Borwser和Render进程里都有代理对象。CefBrowser和CefFrame对象都有一个唯一ID值绑定，便于在两个进程间定位匹配的代理对象。

处理启动消息(Process Startup Messages)

为了给所有的Render进程提供一样的启动信息，请在Browser进程实现CefBrowserProcessHander::OnRenderProcessThreadCreated()方法。在这里传入的信息会在Render进程的CefRenderProcessHandler::OnRenderThreadCreated()方法里接受。

处理运行时消息(Process Runtime Messages)

在进程声明周期内，任何时候你都可以通过CefProcessMessage类传递进程间消息。这些信息和特定的CefBrowser实例绑定在一起，用户可以通过CefBrowser::SendProcessMessage()

方法发送。进程间消息可以包含任意的状态信息，用户可以通过CefProcessMessage::GetArgumentList()获取。

```
// Create the message object.
CefRefPtr<CefProcessMessage> msg= CefProcessMessage::Create(“my_message”);

// Retrieve the argument list object.
CefRefPtr<CefListValue> args = msg>GetArgumentList();

// Populate the argument values.
args->SetString(0, “my string”);
args->SetInt(0, 10);

// Send the process message to the render process.
// Use PID_BROWSER instead when sending a message to the browser process.
browser->SendProcessMessage(PID_RENDERER, msg);
```

一个从Browser进程发送到Render进程的消息将会在CefRenderProcessHandler::OnProcessMessageReceived()方法里被接收。一个从Render进程发送到Browser进程的消息将会在CefClient::OnProcessMessageReceived()方法里被接收。

```
bool MyHandler::OnProcessMessageReceived(
    CefRefPtr<CefBrowser> browser,
    CefProcessId source_process,
    CefRefPtr<CefProcessMessage> message) {
  // Check the message name.
  const std::string& message_name = message->GetName();
  if (message_name == “my_message”) {
    // Handle the message here...
    return true;
  }
  return false;
}
```

我们可以调用CefFrame::GerIdentifier()获取CefFrame的ID，并通过进程间消息发送给另一个进程，然后在接收端通过CefBrowser::GetFrame()找到对应的CefFrame。通过这种方式可以将进程间消息和特定的CefFrame联系在一起。

```
// Helper macros for splitting and combining the int64 frame ID value.
#define MAKE_INT64(int_low, int_high) \
    ((int64) (((int) (int_low)) | ((int64) ((int) (int_high))) << 32))
#define LOW_INT(int64_val) ((int) (int64_val))
#define HIGH_INT(int64_val) ((int) (((int64) (int64_val) >> 32) & 0xFFFFFFFFL)
)

// Sending the frame ID.
const int64 frame_id = frame->GetIdentifier();
args->SetInt(0, LOW_INT(frame_id));
args->SetInt(1, HIGH_INT(frame_id));
```

```
// Receiving the frame ID.
const int64 frame_id = MAKE_INT64(args->GetInt(0), args->GetInt(1));
CefRefPtr<CefFrame> frame = browser->GetFrame(frame_id);
```

异步JavaScript绑定(Asynchronous JavaScript Bindings)

JavaScritp被集成在Render进程，但是需要频繁和Browser进程交互。 JavaScript API应该被设计成可使用闭包异步执行。

通用消息转发(Generic Message Router)

从1574版本开始，CEF提供了在Render进程执行的JavaScript和在Browser进程执行的C++代码之间同步通信的转发器。应用程序通过C++回调函数（OnBeforeBrowse, OnProcessMessageRecieved, OnContextCreated等）传递数据。Render进程支持通用的JavaScript回调函数注册机制，Browser进程则支持应用程序注册特定的Handler进行处理。

下面的代码示例在JavaScript端扩展window对象，添加cefQuery函数：

```
// Create and send a new query.
var request_id = window.cefQuery({
    request: 'my_request',
    persistent: false,
    onSuccess: function(response) {},
    onFailure: function(error_code, error_message) {}
});

// Optionally cancel the query.
window.cefQueryCancel(request_id);
```

The C++ handler looks like this:
对应的C++ Handler代码如下：

```
class Callback : public CefBase {
 public:
  ///
  // Notify the associated JavaScript onSuccess callback that the query has
  // completed successfully with the specified |response|.
  ///
  virtual void Success(const CefString& response) =0;

  ///
  // Notify the associated JavaScript onFailure callback that the query has
  // failed with the specified |error_code| and |error_message|.
  ///
  virtual void Failure(int error_code, const CefString& error_message) =0;
};

class Handler {
 public:
```

```
  ///
  // Executed when a new query is received. |query_id| uniquely identifies the
  // query for the life span of the router. Return true to handle the query
  // or false to propagate the query to other registered handlers, if any. If
  // no handlers return true from this method then the query will be
  // automatically canceled with an error code of -1 delivered to the
  // JavaScript onFailure callback. If this method returns true then a
  // Callback method must be executed either in this method or asynchronously
  // to complete the query.
  ///
  virtual bool OnQuery(CefRefPtr<CefBrowser> browser,
                       CefRefPtr<CefFrame> frame,
                       int64 query_id,
                       const CefString& request,
                       bool persistent,
                       CefRefPtr<Callback> callback) {
    return false;
  }

  ///
  // Executed when a query has been canceled either explicitly using the
  // JavaScript cancel function or implicitly due to browser destruction,
  // navigation or renderer process termination. It will only be called for
  // the single handler that returned true from OnQuery for the same
  // |query_id|. No references to the associated Callback object should be
  // kept after this method is called, nor should any Callback methods be
  // executed.
  ///
  virtual void OnQueryCanceled(CefRefPtr<CefBrowser> browser,
                               CefRefPtr<CefFrame> frame,
                               int64 query_id) {}
};
```

完整的用法请参考wrapper/cef_message_router.h

自定义实现(Custom Implementation)

一个CEF应用程序也可以提供自己的异步JavaScript绑定。典型的实现如下：

1. Render进程的JavaScript传递一个回调函数。

```
// In JavaScript register the callback function.
app.setMessageCallback('binding_test', function(name, args) {
  document.getElementById('result').value = "Response: "+args[0];
});
```

1. Render进程的C++端通过一个map持有JavaScript端注册的回调函数。

```
// Map of message callbacks.
typedef std::map<std::pair<std::string, int>,
                 std::pair<CefRefPtr<CefV8Context>, CefRefPtr<CefV8Value> > >
```

```
                CallbackMap;
CallbackMap callback_map_;

// In the CefV8Handler::Execute implementation for “setMessageCallback”.
if (arguments.size() == 2 && arguments[0]->IsString() &&
    arguments[1]->IsFunction()) {
  std::string message_name = arguments[0]->GetStringValue();
  CefRefPtr<CefV8Context> context = CefV8Context::GetCurrentContext();
  int browser_id = context->GetBrowser()->GetIdentifier();
  callback_map_.insert(
      std::make_pair(std::make_pair(message_name, browser_id),
                     std::make_pair(context, arguments[1])));
}
```

1. Render进程发送异步进程间通信到Browser进程。
2. Browser进程接收到进程间消息，并处理。
3. Browser进程处理完毕后，发送一个异步进程间消息给Render进程，返回结果。
4. Render进程接收到进程间消息，则调用最开始保存的JavaScript注册的回调函数处理之。

```
// Execute the registered JavaScript callback if any.
if (!callback_map_.empty()) {
  const CefString& message_name = message->GetName();
  CallbackMap::const_iterator it = callback_map_.find(
      std::make_pair(message_name.ToString(),
                     browser->GetIdentifier()));
  if (it != callback_map_.end()) {
    // Keep a local reference to the objects. The callback may remove itself
    // from the callback map.
    CefRefPtr<CefV8Context> context = it->second.first;
    CefRefPtr<CefV8Value> callback = it->second.second;

    // Enter the context.
    context->Enter();

    CefV8ValueList arguments;

    // First argument is the message name.
    arguments.push_back(CefV8Value::CreateString(message_name));

    // Second argument is the list of message arguments.
    CefRefPtr<CefListValue> list = message->GetArgumentList();
    CefRefPtr<CefV8Value> args = CefV8Value::CreateArray(list->GetSize());
    SetList(list, args);  // Helper function to convert CefListValue to CefV8V
alue.
    arguments.push_back(args);

    // Execute the callback.
    CefRefPtr<CefV8Value> retval = callback->ExecuteFunction(NULL, arguments);
```

```
    if (retval.get()) {
      if (retval->IsBool())
        handled = retval->GetBoolValue();
    }

    // Exit the context.
    context->Exit();
  }
}
```

1. 在CefRenderProcessHandler::OnContextReleased()里释放JavaScript注册的回调函数以及其他V8资源。

```
void MyHandler::OnContextReleased(CefRefPtr<CefBrowser> browser,
                                  CefRefPtr<CefFrame> frame,
                                  CefRefPtr<CefV8Context> context) {
  // Remove any JavaScript callbacks registered for the context that has been
released.
  if (!callback_map_.empty()) {
    CallbackMap::iterator it = callback_map_.begin();
    for (; it != callback_map_.end();) {
      if (it->second.first->IsSame(context))
        callback_map_.erase(it++);
      else
        ++it;
    }
  }
}
```

同步请求(Synchronous Requests)

某些特殊场景下，也许会需要在Browser进程和Render进程做进程间同步通信。这应该被尽可能避免，因为这会对Render进程的性能造成负面影响。然而如果你一定要做进程间同步通信，可以考虑使用XMLHttpRequest，XMLHttpRequest在等待Browser进程的网络响应的时候会等待。Browser进程可以通过自定义scheme Handler或者网络交互处理XMLHttpRequest。更多细节，请参考`Network Layer`一节。

网络层(Network Layer)

默认情况下，CEF3的网络请求会被宿主程序手工处理。然而CEF3也暴露了一系列网络相关的函数用以处理网络请求。

网络相关的回调函数可在不同线程被调用，因此要注意相关文档的说明，并对自己的数据进行线程安全保护。

自定义请求(Custom Requests)

通过CefFrame::LoadURL()方法可简单加载一个url:

```
browser->GetMainFrame()->LoadURL(some_url);
```

如果希望发送更复杂的请求，则可以调用CefFrame::LoadRequest()方法。该方法接受一个CefRequest对象作为唯一的参数。

```
// Create a CefRequest object.
CefRefPtr<CefRequest> request = CefRequest::Create();

// Set the request URL.
request->SetURL(some_url);

// Set the request method. Supported methods include GET, POST, HEAD, DELETE a
nd PUT.
request->SetMethod("POST");

// Optionally specify custom headers.
CefRequest::HeaderMap headerMap;
headerMap.insert(
    std::make_pair("X-My-Header", "My Header Value"));
request->SetHeaderMap(headerMap);

// Optionally specify upload content.
// The default "Content-Type" header value is "application/x-www-form-urlencod
ed".
// Set "Content-Type" via the HeaderMap if a different value is desired.
const std::string& upload_data = "arg1=val1&arg2=val2";
CefRefPtr<CefPostData> postData = CefPostData::Create();
CefRefPtr<CefPostDataElement> element = CefPostDataElement::Create();
element->SetToBytes(upload_data.size(), upload_data.c_str());
postData->AddElement(element);
request->SetPostData(postData);
```

浏览器无关请求(Browser-Independent Requests)

应用程序可以通过CefURLRequest类发送和浏览器无关的网络请求。并实现CefURLRequestClient接口处理响应。CefURLRequest可以在Browser和Render进程被使用。

```
class MyRequestClient : public CefURLRequestClient {
 public:
  MyRequestClient()
    : upload_total_(0),
      download_total_(0) {}

  virtual void OnRequestComplete(CefRefPtr<CefURLRequest> request) OVERRIDE {
    CefURLRequest::Status status = request->GetRequestStatus();
    CefURLRequest::ErrorCode error_code = request->GetRequestError();
    CefRefPtr<CefResponse> response = request->GetResponse();

    // Do something with the response...
  }
```

```
  virtual void OnUploadProgress(CefRefPtr<CefURLRequest> request,
                                uint64 current,
                                uint64 total) OVERRIDE {
    upload_total_ = total;
  }

  virtual void OnDownloadProgress(CefRefPtr<CefURLRequest> request,
                                  uint64 current,
                                  uint64 total) OVERRIDE {
    download_total_ = total;
  }

  virtual void OnDownloadData(CefRefPtr<CefURLRequest> request,
                              const void* data,
                              size_t data_length) OVERRIDE {
    download_data_ += std::string(static_cast<const char*>(data), data_length)
;
  }

 private:
  uint64 upload_total_;
  uint64 download_total_;
  std::string download_data_;

 private:
  IMPLEMENT_REFCOUNTING(MyRequestClient);
};
```

To send the request:
下面的代码发送一个请求:

```
// Set up the CefRequest object.
CefRefPtr<CefRequest> request = CefRequest::Create();
// Populate |request| as shown above...

// Create the client instance.
CefRefPtr<MyRequestClient> client = new MyRequestClient();

// Start the request. MyRequestClient callbacks will be executed asynchronousl
y.
CefRefPtr<CefURLRequest> url_request = CefURLRequest::Create(request, client.g
et());
// To cancel the request: url_request->Cancel();
```

可以通过CefRequest::SetFlags定制请求的行为，这些标志位包括：

- **UR_FLAG_SKIP_CACHE** 如果设置了该标志位，则处理请求响应时，缓存将被跳过。
- **UR_FLAG_ALLOW_CACHED_CREDENTIALS** 如果设置了该标志位，则可能会发送cookie并在响应端被保存。同时UR_FLAG_ALLOW_CACHED_CREDENTIALS标志位也必须被设置。

- **UR_FLAG_REPORT_UPLOAD_PROGRESS** 如果设置了该标志位，则当请求拥有请求体时，上载进度事件将会被触发。
- **UR_FLAG_REPORT_LOAD_TIMING** 如果设置了该标志位，则时间信息会被收集。
- **UR_FLAG_REPORT_RAW_HEADERS** 如果设置了该标志位，则头部会被发送，并且接收端会被记录。
- **UR_FLAG_NO_DOWNLOAD_DATA** 如果设置了该标志位，则CefURLRequestClient::OnDownloadData方法不会被调用。
- **UR_FLAG_NO_RETRY_ON_5XX** 如果设置了该标志位，则5xx重定向错误会被交给相关Observer去处理，而不是自动重试。这个功能目前只能在Browser进程的请求端使用。

例如，为了跳过缓存并不报告下载数据，代码如下：

```
request->SetFlags(UR_FLAG_SKIP_CACHE | UR_FLAG_NO_DOWNLOAD_DATA);
```

请求响应(Request Handling)

CEF3 支持两种方式处理网络请求。一种是实现scheme Handler，这种方式允许为特定的(sheme+domain)请求注册特定的请求响应。另一种是请求拦截，允许处理任意的网络请求。

注册自定义scheme（有别于`HTTP`，`HTTPS`等）可以让CEF按希望的方式处理请求。例如，如果你希望特定的shceme被当策划那个HTTP一样处理，则应该注册一个`standard`的scheme。如果你的自定义shceme可被跨域执行，则应该考虑使用使用HTTP scheme代替自定义scheme以避免潜在问题。如果你希望使用自定义scheme，实现CefApp::OnRegisterCustomSchemes回调。

```
void MyApp::OnRegisterCustomSchemes(CefRefPtr<CefSchemeRegistrar> registrar) {
  // Register "client" as a standard scheme.
  registrar->AddCustomScheme("client", true, false, false);
}
```

Scheme响应(Scheme Handler)

通过CefRegisterSchemeHandlerFactory方法注册一个scheme响应，最好在CefBrowserProcessHandler::OnContextInitialized()方法里调用。例如，你可以注册一个”client://myapp/”的请求：

```
CefRegisterSchemeHandlerFactory("client", “myapp”, new MySchemeHandlerFactory(
));
```

scheme Handler类可以被用在内置shcme(HTTP,HTTPS等），也可以被用在自定义scheme上。当使用内置shceme，选择一个对你的应用程序来说唯一的域名。实现CefSchemeHandlerFactory和CefResoureHandler类去处理请求并返回响应数据。可以参考cefclient/sheme_test.h的例子。

```
// Implementation of the factory for creating client request handlers.
class MySchemeHandlerFactory : public CefSchemeHandlerFactory {
```

```
 public:
  virtual CefRefPtr<CefResourceHandler> Create(CefRefPtr<CefBrowser> browser,
                                               CefRefPtr<CefFrame> frame,
                                               const CefString& scheme_name,
                                               CefRefPtr<CefRequest> request)
                                               OVERRIDE {
    // Return a new resource handler instance to handle the request.
    return new MyResourceHandler();
  }

  IMPLEMENT_REFCOUNTING(MySchemeHandlerFactory);
};

// Implementation of the resource handler for client requests.
class MyResourceHandler : public CefResourceHandler {
 public:
  MyResourceHandler() {}

  virtual bool ProcessRequest(CefRefPtr<CefRequest> request,
                              CefRefPtr<CefCallback> callback)
                              OVERRIDE {
    // Evaluate |request| to determine proper handling...
    // Execute |callback| once header information is available.
    // Return true to handle the request.
    return true;
  }

  virtual void GetResponseHeaders(CefRefPtr<CefResponse> response,
                                  int64& response_length,
                                  CefString& redirectUrl) OVERRIDE {
    // Populate the response headers.
    response->SetMimeType("text/html");
    response->SetStatus(200);

    // Specify the resulting response length.
    response_length = ...;
  }

  virtual void Cancel() OVERRIDE {
    // Cancel the response...
  }

  virtual bool ReadResponse(void* data_out,
                            int bytes_to_read,
                            int& bytes_read,
                            CefRefPtr<CefCallback> callback)
                            OVERRIDE {
    // Read up to |bytes_to_read| data into |data_out| and set |bytes_read|.
    // If data isn't immediately available set bytes_read=0 and execute
    // |callback| asynchronously.
    // Return true to continue the request or false to complete the request.
    return …;
  }
```

```
 private:
  IMPLEMENT_REFCOUNTING(MyResourceHandler);
};
```

如果响应数据类型是已知的，则CefStreamResourceHandler类提供了CefResourceHandler类的默认实现。

```
// CefStreamResourceHandler is part of the libcef_dll_wrapper project.
#include “include/wrapper/cef_stream_resource_handler.h”

const std::string& html_content = “<html><body>Hello!</body></html>”;

// Create a stream reader for |html_content|.
CefRefPtr<CefStreamReader> stream =
    CefStreamReader::CreateForData(
        static_cast<void*>(const_cast<char*>(html_content.c_str())),
        html_content.size());

// Constructor for HTTP status code 200 and no custom response headers.
// There’s also a version of the constructor for custom status code and respon
se headers.
return new CefStreamResourceHandler("text/html", stream);
```

请求拦截(Request Interception)

CefRequestHandler::GetResourceHandler()方法支持拦截任意请求。参考client_handler.cpp。

```
CefRefPtr<CefResourceHandler> MyHandler::GetResourceHandler(
      CefRefPtr<CefBrowser> browser,
      CefRefPtr<CefFrame> frame,
      CefRefPtr<CefRequest> request) {
  // Evaluate |request| to determine proper handling...
  if (...)
    return new MyResourceHandler();

  // Return NULL for default handling of the request.
  return NULL;
}
```

其他回调(Other Callbacks)

CefRequestHander接口还提供了其他回调函数以定制其他网络相关事件。包括授权、cookei处理、外部协议处理、证书错误等。

Proxy Resolution

CEF3使用类似Google Chrome一样的方式，通过命令行参数传递代理配置。

```
--proxy-server=host:port
      Specify the HTTP/SOCKS4/SOCKS5 proxy server to use for requests. An indi
vidual proxy
      server is specified using the format:

        [<proxy-scheme>://]<proxy-host>[:<proxy-port>]

      Where <proxy-scheme> is the protocol of the proxy server, and is one of:

        "http", "socks", "socks4", "socks5".

      If the <proxy-scheme> is omitted, it defaults to "http". Also note that
"socks" is equivalent to
      "socks5".

      Examples:

        --proxy-server="foopy:99"
            Use the HTTP proxy "foopy:99" to load all URLs.

        --proxy-server="socks://foobar:1080"
            Use the SOCKS v5 proxy "foobar:1080" to load all URLs.

        --proxy-server="sock4://foobar:1080"
            Use the SOCKS v4 proxy "foobar:1080" to load all URLs.

        --proxy-server="socks5://foobar:66"
            Use the SOCKS v5 proxy "foobar:66" to load all URLs.

      It is also possible to specify a separate proxy server for different URL
types, by prefixing
      the proxy server specifier with a URL specifier:

      Example:

        --proxy-server="https=proxy1:80;http=socks4://baz:1080"
            Load https://* URLs using the HTTP proxy "proxy1:80". And load htt
p://*
            URLs using the SOCKS v4 proxy "baz:1080".

--no-proxy-server
      Disables the proxy server.

--proxy-auto-detect
      Autodetect  proxy  configuration.

--proxy-pac-url=URL
      Specify proxy autoconfiguration URL.
```

如果代理请求授权，CefRequestHandler::GetAuthCredentials()回调会被调用。如果isProxy参数为true，则需要返回用户名和密码。

```
bool MyHandler::GetAuthCredentials(
    CefRefPtr<CefBrowser> browser,
    CefRefPtr<CefFrame> frame,
    bool isProxy,
    const CefString& host,
    int port,
    const CefString& realm,
    const CefString& scheme,
    CefRefPtr<CefAuthCallback> callback) {
  if (isProxy) {
    // Provide credentials for the proxy server connection.
    callback->Continue("myuser", "mypass");
    return true;
  }
  return false;
}
```

网络内容加载可能会因为代理而有延迟。为了更好的用户体验，可以考虑让你的应用程序先显示一个闪屏，等内容加载好了再通过meta refresh显示真实网页。可以指定`--no-proxy-server`禁用代理并做相关测试。代理延迟也可以通过chrome浏览器重现，方式是使用命令行传参：`chrome -url=...`.