Chromium/content v8::Extension

[fanfeilong@gmail.com](mailto:fanfeilong@gmail.com)

@幻灰龙

目录

[1 注册v8::Extension 2](#_Toc373142664)

[2 注册v8::Extension的内部流程 2](#_Toc373142665)

[3 理解v8的设计理念和重要概念 6](#_Toc373142666)

[3.1 v8的设计文档 6](#_Toc373142671)

[3.1.1 属性的快速读取 6](#_Toc373142676)

[3.1.2 动态代码生成 7](#_Toc373142677)

[3.1.3 高效垃圾回收。 8](#_Toc373142678)

[3.2 v8的hello world。 8](#_Toc373142679)

[3.3 v8的重要概念。 9](#_Toc373142680)

[3.3.1 句柄（Handles）和垃圾回收（Garbage Collection） 10](#_Toc373142681)

[3.3.2 上下文(Context) 12](#_Toc373142682)

[3.3.3 模板(Templates) 12](#_Toc373142683)

[3.3.4 存取器（Accessor） 13](#_Toc373142684)

[3.3.5 拦截器（Interceptor） 15](#_Toc373142685)

[3.3.6 安全（Security） 16](#_Toc373142686)

[3.3.7 继承（Inheritance） 17](#_Toc373142687)

[3.3.8 Isolate，Locker以及GC设置 17](#_Toc373142688)

[3.3.9 v8的类型体系 18](#_Toc373142689)

[3.4 v8 API手册 19](#_Toc373142690)

[4 CEF的v8::Extension扩展机制 19](#_Toc373142691)

[4.1 CEF类型:CefV8Value 19](#_Toc373142692)

[4.2 CefValueImpl的存储 21](#_Toc373142693)

[4.3 基本内置类型 24](#_Toc373142694)

[4.4 Object和UserData 25](#_Toc373142695)

[4.5 InitFromV8Value和GetV8Value 27](#_Toc373142696)

[4.6 函数 29](#_Toc373142697)

[4.7 字典 36](#_Toc373142698)

[4.8 异常 45](#_Toc373142699)

[4.9 内存控制 46](#_Toc373142700)

[4.10 Cef的对象跟踪:V8TrackObject 47](#_Toc373142701)

[4.11 CefIsolateManager 50](#_Toc373142702)

[4.11.1 Isolate 55](#_Toc373142703)

[4.11.2 CefContextState 55](#_Toc373142704)

[4.11.3 SetUncaughtExceptionStackSize 59](#_Toc373142705)

[4.12 CefV8Context 60](#_Toc373142706)

[4.13 CefV8Handle 64](#_Toc373142707)

[4.14 CefV8Hanlder 65](#_Toc373142708)

[4.15 ExtensionWrapper 65](#_Toc373142709)

[4.16 注册v8扩展 68](#_Toc373142710)

# 注册v8::Extension

用法：

|  |
| --- |
| content::RenderThread::Get()->RegisterExtension(extension); |

接口：

|  |
| --- |
| class CONTENT\_EXPORT RenderThread : public IPC::Sender {  // Registers the given V8 extension with WebKit.  virtual void RegisterExtension(v8::Extension\* extension) = 0;  } |

# 注册v8::Extension的内部流程

Content的RenderThread::RegisterExtension(v8::Extension\* extension)对上层暴露注册v8扩展API.

|  |
| --- |
| class CONTENT\_EXPORT RenderThread : public IPC::Sender {  // Registers the given V8 extension with WebKit.  virtual void RegisterExtension(v8::Extension\* extension) = 0;  } |

Content的RenderThreadImpl则具体实现该API:

|  |
| --- |
| // The RenderThreadImpl class represents a background thread where RenderView  // instances live. The RenderThread supports an API that is used by its  // consumer to talk indirectly to the RenderViews and supporting objects.  // Likewise, it provides an API for the RenderViews to talk back to the main  // process (i.e., their corresponding WebContentsImpl).  //  // Most of the communication occurs in the form of IPC messages. They are  // routed to the RenderThread according to the routing IDs of the messages.  // The routing IDs correspond to RenderView instances.  class CONTENT\_EXPORT RenderThreadImpl : public RenderThread,  public ChildThread,  public GpuChannelHostFactory {  virtual void RegisterExtension(v8::Extension\* extension) OVERRIDE;  } |

该方法内部则调用了WebKit的WebScriptController方法：

|  |
| --- |
| void RenderThreadImpl::RegisterExtension(v8::Extension\* extension) {  WebScriptController::registerExtension(extension);  } |

查看WebKit的WebScriptController的registerExtension方法：

|  |
| --- |
| class WebScriptController {  public:  // Registers a v8 extension to be available on webpages. Will only affect  // v8 contexts initialized after this call. Takes ownership of the  // v8::Extension object passed.  WEBKIT\_EXPORT static void registerExtension(v8::Extension\*);  } |

实现如下：

|  |
| --- |
| void WebScriptController::registerExtension(v8::Extension\* extension)  {  ScriptController::registerExtensionIfNeeded(extension);  } |

WebScriptController内部调用了v8的ScriptController的方法：

|  |
| --- |
| class ScriptController {  // Registers a v8 extension to be available on webpages. Will only  // affect v8 contexts initialized after this call. Takes ownership of  // the v8::Extension object passed.  static void registerExtensionIfNeeded(v8::Extension\*);  } |

实现如下：

|  |
| --- |
| void ScriptController::registerExtensionIfNeeded(v8::Extension\* extension)  {  const V8Extensions& extensions = registeredExtensions();  for (size\_t i = 0; i < extensions.size(); ++i) {  if (extensions[i] == extension)  return;  }  v8::RegisterExtension(extension);  registeredExtensions().append(extension);  } |

可见v8内部会先判断待注册的扩展是否已存在，如果已存在则直接返回，否则注册并添加到扩展集合里。我们看下v8::RegisterExtension里面做了什么。

|  |
| --- |
| void RegisterExtension(Extension\* that) {  RegisteredExtension\* extension = new RegisteredExtension(that);  RegisteredExtension::Register(extension);  }  void RegisteredExtension::Register(RegisteredExtension\* that) {  that->next\_ = first\_extension\_;  first\_extension\_ = that;  } |

可见，该方法只是将v8::Extension转换成v8::RegisteredExtension，然后串到v8的扩展链表上。我们看下RegisteredExtension这个类：

|  |
| --- |
| class RegisteredExtension {  public:  explicit RegisteredExtension(Extension\* extension);  static void Register(RegisteredExtension\* that);  static void UnregisterAll();  Extension\* extension() { return extension\_; }  RegisteredExtension\* next() { return next\_; }  RegisteredExtension\* next\_auto() { return next\_auto\_; }  static RegisteredExtension\* first\_extension() { return first\_extension\_; }  private:  Extension\* extension\_;  RegisteredExtension\* next\_;  RegisteredExtension\* next\_auto\_;  static RegisteredExtension\* first\_extension\_;  }; |

该类只是一个v8:Extension的单链表。从而，我们只需重点研究v8::Extension的类结构、方法，以及在v8引擎里的什么地方调用v8::Extension即可。首先查看v8::Extension的结构：

|  |
| --- |
| class V8EXPORT Extension { // NOLINT  public:  // Note that the strings passed into this constructor must live as long  // as the Extension itself.  Extension(const char\* name,  const char\* source = 0,  int dep\_count = 0,  const char\*\* deps = 0,  int source\_length = -1);  virtual ~Extension() { }  virtual v8::Handle<v8::FunctionTemplate>  GetNativeFunction(v8::Handle<v8::String> name) {  return v8::Handle<v8::FunctionTemplate>();  }  const char\* name() const { return name\_; }  size\_t source\_length() const { return source\_length\_; }  const String::ExternalAsciiStringResource\* source() const {  return &source\_; }  int dependency\_count() { return dep\_count\_; }  const char\*\* dependencies() { return deps\_; }  void set\_auto\_enable(bool value) { auto\_enable\_ = value; }  bool auto\_enable() { return auto\_enable\_; }  private:  const char\* name\_;  size\_t source\_length\_; // expected to initialize before source\_  ExternalAsciiStringResourceImpl source\_;  int dep\_count\_;  const char\*\* deps\_;  bool auto\_enable\_;  // Disallow copying and assigning.  Extension(const Extension&);  void operator=(const Extension&);  }; |

从v8:Extension可以看出，最重要的是实现GetNativeFunction这个虚方法，描述如下：

|  |  |
| --- | --- |
| 参数v8::Handle<v8::String> name | 本地函数名字 |
| 返回值v8::Handle<v8::FunctionTemplate> | V8函数 |

则重点看下v8::Handle<v8::FunctionTemplate>这个类，这个类的注释比较多，具体直接查看src\v8\include\v8.h里的注释。我们只摘录该类唯一以及最重要的一个静态方法：

|  |
| --- |
| typedef Handle<Value> (\*InvocationCallback)(const Arguments& args);  class V8EXPORT FunctionTemplate : public Template {  public:  /\*\* Creates a function template.\*/  static Local<FunctionTemplate> New(  InvocationCallback callback = 0,  Handle<Value> data = Handle<Value>(),  Handle<Signature> signature = Handle<Signature>(),  int length = 0);  } |

V8::Hanlde<FunctionTemplate>::New静态方法是用来将Native方法转换成v8引擎内部使用的方法模板上。从参数列表里可知用户的本地函数只需声明为InvocationCallback所示的函数原型就可以。参数列表为v8::Hanle<Arguments>，返回值是v8::Hanlde<v8::Value>。

这里有个老版本的示例：

<http://code.google.com/p/v8/source/browse/trunk/src/extensions/?r=8431>

# 理解v8的设计理念和重要概念



## v8的设计文档

首先需要对v8引擎有个整体了解，v8引擎用于高效解析和执行JavaScript。下面的文档描述了v8引擎高效的几个设计。文档上有详细的描述，此处我们简要翻译。

<https://developers.google.com/v8/design>



### 属性的快速读取

JavaScript的对象是允许动态增减属性的，文档上说大部分其他JavaScript引擎都是是用字典来实现属性的动态增减，然而这样的话相比于Java和SmllTalk这样的静态类型语言来说就减低了访问效率。具体来说静态类型在编译时就确定了属性的偏移位置，所以访问只是直接做指针偏移，而使用字典的设计则每次访问都需要做一次查询，也就是间接寻址。为了达到静态类型的属性访问效率，同时保留属性的动态性，v8采用了基于原型编程语言设计。

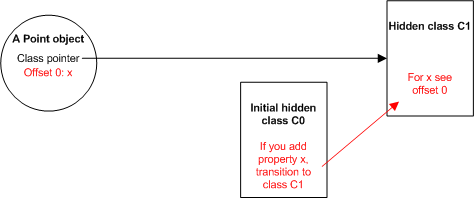
假设下面是一段JavaScript函数，用于动态创建一个对象：

|  |
| --- |
| function Point(x, y) {   this.x = x;   this.y = y; } |

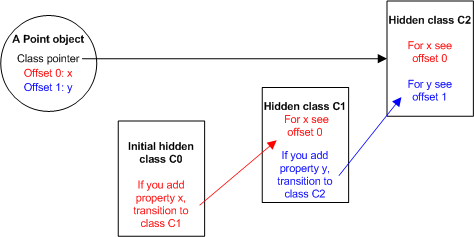
第一步：当执行new Point(x,y)时，一个新的Point对象被创建，当v8首次创建一个Point对象时，会同时创建一个Point的隐藏类C0，结构图如下：



第二步：执行函数里的第一行代码this.x=x;此时，v8会以隐藏类C0为原型创建新的隐藏类C1，C1拥有属性x，偏移位置为0。然后通过类型转移将之前的Point对象指向的C0隐藏类改为指向C1隐藏类。此时Point的隐藏类是C1。



第三步：执行函数里的第二行代码this.y=y;此时，v8会以隐藏类C1为原型创建新的隐藏类C2，C2拥有属性x，y，x的偏移位置依然是0，y的偏移位置为1。然后通过类型转移将之前的Point对象指向的C1隐藏类改为指向C2隐藏类。此时Point的隐藏类是C2。



这个过程本身并不高效，但v8引擎通过缓存这些隐藏类，在下一次创建Point对象时复用这些隐藏类从而达到高效创建隐藏类。下一次创建Point对象时的过程如下：

第一步，初始化Point对象，指向隐藏类C0.

第二步，添加x属性，Point指向隐藏类C1.

第三步，添加y属性，Point指向隐藏类C2.

从而，v8达到高效创建隐藏类，进而达到对对象属性的快速访问能力，这是v8高效于其他JavaScript引擎的设计之一。

### 动态代码生成

在隐藏类的基础上，v8引擎进一步通过JIT技术提高属性的访问效率。具体来说，访问代码在首次执行时直接被编译成了目标机器代码，属性访问直接内联了缓存的目标机器代码。如果隐藏类发生改变，缓存的目标机器代码会被更新。

具体来说，以point.x为例，首次编译时生成的代码如下：

|  |
| --- |
| # ebx = the point object  cmp [ebx,<hidden class offset>],<cached hidden class>  jne <inline cache miss>  mov eax,[ebx, <cached x offset>] |

如果Point的隐藏类跟缓存的隐藏类不匹配，则指令跳转到v8运行时进行即时编译，否则直接返回指定偏移位置的x的值。

### 高效垃圾回收。

v8引擎设计了高效的垃圾回收机制，以保证快速的对象内存分配、短暂的垃圾回收暂停、无内存碎片。具体来说：

在垃圾回收时，停止程序运行。

在大部分垃圾回收周期内只处理一小部分堆内存，这样最大限度减少程序暂停时间。

保持对所有对象和指针在内存的位置信息，从而杜绝内存泄漏。

## v8的hello world。

这个例子首先给出不添加任何新概念的示例代码，然后给出添加了v8的对象生命周期管理的示例代码以引出v8的对象生命周期相关的几个重要概念。

<https://developers.google.com/v8/get_started>

首先是一个裸代码：

|  |
| --- |
| int main(int argc, char\* argv[]) {   // Create a string containing the JavaScript source code.   String source = String::New("'Hello' + ', World'");    // Compile the source code.   Script script = Script::Compile(source);      // Run the script to get the result.   Value result = script->Run();    // Convert the result to an ASCII string and print it.   String::AsciiValue ascii(result);   printf("%s\n", \*ascii);   return 0; } |

与所有的脚本引擎类似，上述代码只是简单的将JavaScript脚本送进v8引擎编译，运行并返回结果。下面我们看添加了v8的对象生命周期管理的代码：

|  |
| --- |
| #include <v8.h>  using namespace v8;  int main(int argc, char\* argv[]) {   // Get the default Isolate created at startup.   Isolate\* isolate = Isolate::GetCurrent();    // Create a stack-allocated handle scope.   HandleScope handle\_scope(isolate);    // Create a new context.   Handle<Context> context = Context::New(isolate);    // Here's how you could create a Persistent handle to the context, if needed.   Persistent<Context> persistent\_context(isolate, context);      // Enter the created context for compiling and   // running the hello world script.    Context::Scope context\_scope(context);    // Create a string containing the JavaScript source code.   Handle<String> source = String::New("'Hello' + ', World!'");    // Compile the source code.   Handle<Script> script = Script::Compile(source);      // Run the script to get the result.   Handle<Value> result = script->Run();      // The persistent handle needs to be eventually disposed.   persistent\_context.Dispose();    // Convert the result to an ASCII string and print it.   String::AsciiValue ascii(result);   printf("%s\n", \*ascii);   return 0; } |

上述代码引进了几个重要的概念，他们都是与v8的对象生命周期管理相关的，下面先简要给出说明，详细介绍在下一小节给出。

v8::Handle是v8的智能指针，负责垃圾回收机制下的对象生命周期管理。

v8::HandleScope是v8的智能指针容器，当对象的Handle生命周期结束时，通过HandleScope批量删除对象来替代逐个删除Handle。

V8::Context是v8用来隔离JavaScript代码运行的机制，v8的JavaScript代码运行必须指定Context。

## v8的重要概念。

v8 API不仅提供了编译和运行JavaScript代码的功能，还提供了其他与C++交互的功能，包括函数和数据结构的注册，错误处理，安全检查等。C++应用程序可以将v8当作一个普通类库使用，只需引用v8.h即可。下面这个链接是C++里使用v8类库的指南，同样的，我们做简要翻译，以理解v8的几个重要概念。

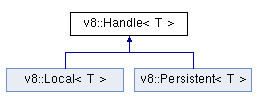
<https://developers.google.com/v8/embed?csw=1#interceptors>

### 句柄（Handles）和垃圾回收（Garbage Collection）

一个句柄对象(v8::Handle<T>)引用了JavaScript对象在堆上的位置。v8垃圾回收会将没有被引用的对象的内存回收掉。在垃圾回收的过程中，堆上的JavaScript对象会被移动，垃圾回收器会自动更新所有引用了被移动对象的句柄。

如果一个对象不能在JavaScript里访问，也没有句柄指向它，则被认为是垃圾对象。垃圾回收器在每次垃圾回收时移除垃圾对象。v8的垃圾回收机制是v8引擎高效的关键因素。

v8一共有两种句柄，局部句柄和持久化句柄。



#### 局部句柄(v8::Local<T>)

局部句柄(v8::Local<T>)在栈上，在析构函数调用时失效。局部句柄的生命周期由v8::HandleScope确定。一般在函数的开头创建v8::HandleScope，则当v8::HandleScope被删除时，垃圾回收器就可以释放在v8::HandleScope管理下的局部句柄对象，从而这些对象不可以在JavaScript代码里访问或者被其他句柄持有。

需要注意的是，句柄所在的栈并不是C++的函数调用栈，而是由v8::HandleScope的生命周期所确定的一个逻辑上的栈。另外，v8::HandleScope只可以在C++栈上分配，不可以使用new在堆上分配。

#### 持久句柄(v8::Persistent<T>)

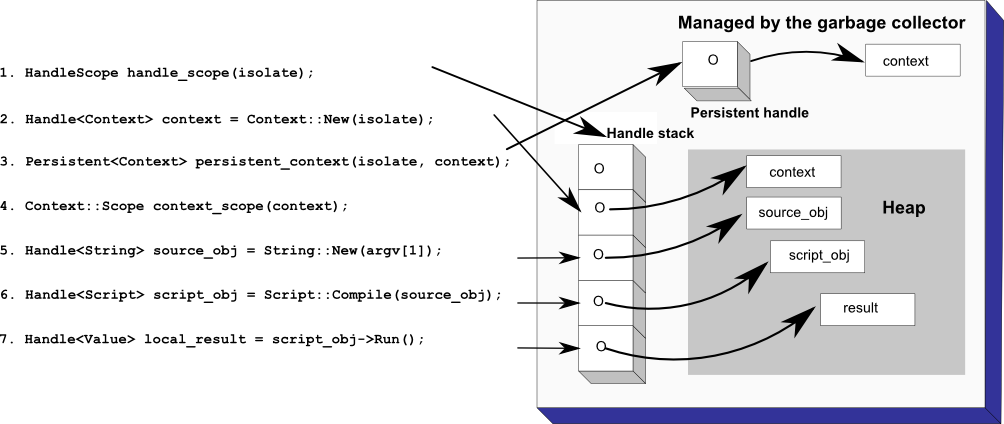
持久句柄（v8::Persistent<T>）并不在栈上，并且必须显示释放才会被删除。与局部句柄类似，持久句柄指向堆上的对象。如果你需要在多个函数里使用一个堆上的对象，则可以使用持久句柄。如果你的对象的生命周期与C++的对象生命周期范围不一致，也可以使用持久句柄。例如，在Google Chrome里使用持久句柄引用DOM节点。

持久局部直接通过构造函数创建，通过v8::Persistent<T>::Dispose()释放。我们可以通过v8::Persistent<T>::MakeWeak()来创建弱持久句柄。

#### 句柄范围(v8::HandleScope)

每次创建一个对象就创建一个句柄会产生大量的句柄对象，因此v8引入了句柄范围这个概念(v8::HandleScope)，句柄范围可以看做是局部句柄(v8::Local<T>)的容器，在句柄范围的析构函数被调用后，所有在容器内的句柄都被从栈上移除，从而可以被垃圾回收。

以3.2的例子，下图给出了对象在内存中的示例图：



当函数退出时，HandleScope::~HandleScope被调用，则句柄容器管理的所有局部句柄都被删除，从而垃圾回收器将从堆上移除source\_obj和script\_obj，因为它们既没有句柄指向他们，也无法从JavaScript里访问。而persistent是一个持久句柄，必须手工调用Dispose方法释放之。函数里面声明的局部句柄不能直接返回给函数调用者，如果需要返回，需要调用Handle::Scope::Close(handle)。下面是一个例子：

|  |
| --- |
| // This function returns a new array with three elements, x, y, and z. Handle<Array> NewPointArray(int x, int y, int z) {   v8::Isolate\* isolate = v8::Isolate::GetCurrent();   // We will be creating temporary handles so we use a handle scope.   HandleScope handle\_scope(isolate);   // Create a new empty array.   Handle<Array> array = Array::New(3);   // Return an empty result if there was an error creating the array.   if (array.IsEmpty())     return Handle<Array>();   // Fill out the values   array->Set(0, Integer::New(x));   array->Set(1, Integer::New(y));   array->Set(2, Integer::New(z));   // Return the value through Close.   return handle\_scope.Close(array); } |

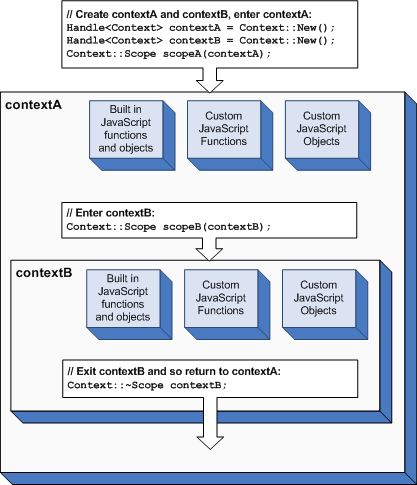
### 上下文(Context)

v8的上下文(Context)是独立的JavaScript执行环境，通过使用上下文允许JavaScript应用程序跑在不同的v8实例上。执行一段JavaScript代码必须显示指定上下文。

为什么必须指定上下文呢？这是因为JavaScript提供了一系列内置辅助函数和全局对象，它们可以被JavaScript代码调用和修改。如果两段不相关的JavaScript代码同时修改全局对象，可能会导致用户不希望看到的结果。

从节省CPU和内存的角度考虑，重复的创建内置函数和全局对象的开销并不低。然而，得益于v8可扩展的缓存机制，只会在首次创建上下文时产生大的开销，后续的子上下文创建的开销低的多。这是由于首次创建上下文时，v8需要创建内置对象，同时转换内置JavaScript代码，而后续的上下文创建则只需创建内置对象即可。进一步，得益于v8的快照特性，首次创建上下文也很高效。这是由于快照包含了已序列化的堆数据，里面有已经编译好的内置JavaScript代码。

v8的上下文通过Enter和Exist进入和退出，并且v8的上下文可以嵌套使用，比如在进入上下文A后接着进入上下文B，然后退出上下文B回到上下文A，最后退出上下文A。面的图示例来这个过程。



### 模板(Templates)

V8里通过模板将C++函数和数据结构封装成JavaScript对象，从而使得JavaScript代码可以使用C++函数和数据结构。例如，Google Chrome使用模板将DOM节点封装成JavaScript对象，以及通过模板添加全局函数。你可以创建一系列模板，然后在每个新的上下文里同时使用它们。需要注意到是，你可以创建不限个数的模板，但是在一个给定的上下文里每个模板只会有一个实例。

在JavaScript语言里，函数和对象是强对偶的。在C++或者Java语言里，创建一个新类型的对象，你通常需要定义一个新的类型。但是在JavaScript里，你只需创建一个函数就可以，将这个函数作为类型的构造函数，从而创建类型的实例。JavaScript对象的布局和功能和创建对象的函数非常接近。这个特点直接反映在v8的模板设计里。

v8有两种类型的模板：函数模板(Function templates)和对象模板(Object templates)。

#### 函数模板（Function templates）

在v8上下文里，你可以调用函数模板的GetFunction创建一个JavaScript模板实例。你也可以将C++回调函数绑定到函数模板，从而使得JavaScrpit拥有调用C++本地函数的能力。

#### 对象模板（Object templates）

每个函数模板有一个伴随的对象模板，在JavaScript调用函数创建新对象时，伴随的对象模板可以被用来配置相关的对象。你可以将两种类型的C++回调函数绑定到对象模板。

存取器回调函数（Accessor），用于访问指定的对象属性。

拦截器回调函数（Interceptor），用户访问任意的对象属性。

下面的代码示例了如何创建一个全局对象模板，并设置内置全局函数

|  |
| --- |
| // Create a template for the global object and set the // built-in global functions. Handle<ObjectTemplate> global = ObjectTemplate::New(); global->Set(String::New("log"), FunctionTemplate::New(LogCallback));  // Each processor gets its own context so different processors // do not affect each other. Persistent<Context> context = Context::New(NULL, global); |

### 存取器（Accessor）

一个存取器是一个C++回调函数，当JavaScript代码调用某个对象的属性时，该回调函数负责计算并返回对应的属性值。模板对象通过该SetAccessor设置存取器。

#### 访问全局静态变量

假设x和y是两个C++整型变量，我们希望在某个上下文里将x和y设置为JavaScript的全局变量。为了达到这个目的，我们需要为对象模板设置C++存取器回掉函数，以便JavaScript读写x和y时调用。存取器回调函数将C++整型变量通过Integer::New转为JavaScript的整型变量，通过Int32Value将JavaScript整型变量转为C++整型变量。示例代码如下：

|  |
| --- |
| Handle<Value> XGetter(Local<String> property, const AccessorInfo& info) {     return Integer::New(x);   }        void XSetter(Local<String> property, Local<Value> value, const AccessorInfo& info) {     x = value->Int32Value();   }           // YGetter/YSetter are so similar they are omitted for brevity   Handle<ObjectTemplate> global\_templ = ObjectTemplate::New();   global\_templ->SetAccessor(String::New("x"), XGetter, XSetter);   global\_templ->SetAccessor(String::New("y"), YGetter, YSetter);   Persistent<Context> context = Context::New(NULL, global\_templ); |

#### 访问动态变量

上一小节的例子是静态全局变量，更多的时候，我们希望在JavaScript里使用动态变量，比如浏览器的DOM树。假设x和y是C++类Point的两个成功变量。

|  |
| --- |
| class Point {    public:     Point(int x, int y) : x\_(x), y\_(y) { }     int x\_, y\_;   } |

为了在JavaScript里使用任意数量的C++ Point实例，我们需要为每个C++ Point实例创建JavaScript封装。在v8里，我们通过外部值(External values)和对象内部成员(internal object fileds)来做到这点。

首先，我们需要创建一个point对象模板：

|  |
| --- |
| Handle<ObjectTemplate> point\_templ = ObjectTemplate::New(); |

其次，每个JavaScript point对象保存一个引用，指向的C++对象被封装成point对象模板的内部成员。point对象模板可以指定任意数量的成员变量：

|  |
| --- |
| point\_templ->SetInternalFieldCount(1); |

接着封装一个C++ Point实例，并且让point\_templ的第0个成员变量指向它：

|  |
| --- |
| Point\* p = ...;   Local<Object> obj = point\_templ->NewInstance();   obj->SetInternalField(0, External::New(p)); |

然后，我们设置x和y属性的存取器：

|  |
| --- |
| point\_templ.SetAccessor(String::New("x"), GetPointX, SetPointX);   point\_templ.SetAccessor(String::New("y"), GetPointY, SetPointY);    Handle<Value> GetPointX(Local<String> property,const AccessorInfo &info) {     Local<Object> self = info.Holder();     Local<External> wrap = Local<External>::Cast(self->GetInternalField(0));     void\* ptr = wrap->Value();     int value = static\_cast<Point\*>(ptr)->x\_;     return Integer::New(value);   }   void SetPointX(Local<String> property, Local<Value> value, const AccessorInfo& info) {     Local<Object> self = info.Holder();     Local<External> wrap = Local<External>::Cast(self->GetInternalField(0));     void\* ptr = wrap->Value();     static\_cast<Point\*>(ptr)->x\_ = value->Int32Value();   } |

与全局静态变量相比，这个做法实际上只是让对象模板保持动态变量的引用，从而在属性存取器里可以读写相应的动态变量。

### 拦截器（Interceptor）

在v8里，我们也可以设置拦截器回调函数，使得JavaScript代码可以访问任意的对象属性，有两种高效的拦截器：命名属性拦截器和索引属性拦截器。

#### 命名属性拦截器

使用字符串访问对象属性，比如：document.theFormName.elementName。

#### 索引属性拦截器

使用索引访问对象属性，比如：document.forms.elements[0].

下面给出拦截器的例子：

|  |
| --- |
| Handle<ObjectTemplate> result = ObjectTemplate::New(); result->SetNamedPropertyHandler(MapGet, MapSet);  Handle<Value> JsHttpRequestProcessor::MapGet(Local<String> name,                                              const AccessorInfo &info) {   // Fetch the map wrapped by this object.   map<string, string> \*obj = UnwrapMap(info.Holder());    // Convert the JavaScript string to a std::string.   string key = ObjectToString(name);    // Look up the value if it exists using the standard STL idiom.   map<string, string>::iterator iter = obj->find(key);    // If the key is not present return an empty handle as signal.   if (iter == obj->end()) return Handle<Value>();    // Otherwise fetch the value and wrap it in a JavaScript string.   const string &value = (\*iter).second;   return String::New(value.c\_str(), value.length()); } |

存取器和拦截器的区别在于，存取器只针对某个具体的属性，而拦截器则可以处理所有的属性。

3.3.7、异常（Exception）

v8会在遇到错误时抛出异常，并返回空句柄。所以异常处理的惯用法如下：

|  |
| --- |
| TryCatch trycatch;   Handle<Value> v = script->Run();   if (v.IsEmpty()) {       Handle<Value> exception = trycatch.Exception();     String::AsciiValue exception\_str(exception);     printf("Exception: %s\n", \*exception\_str);     // ...   } |

### 安全（Security）

同源策略（same orign policy，在Netscape Navigator 2.0里首次引入）防止某个源的文档或脚本读取其他源的设置属性。这里的同源是指域名、协议、端口共同决定的。

在v8引擎里，源被定义为一个上下文（Context），默认情况下v8禁止访问代码跨上下文访问。如果需要跨上下文访问，你需要使用安全口令或者安全回调。安全口令可以任意的值，通常是一个唯一符号串。在创建一个上下文时，可以通过SetSerurityToken设置安全口令。如果上下文没有设置口令，v8会自动为其生成安全口令。当尝试去访问一个全局对象时，v8会比较全局对象的安全口令和访问代码的安全口令，如果口令不匹配，v8会调用一个回调函数以确认是否允许访问。所以你可以通过SetAccessCheckCallbacks来控制对象的访问权限。

### 继承（Inheritance）

我们知道JavaScript是通过prototype实现继承机制的。相应的一个函数模板可以获取原型模板以动态修改属性，并且v8通过了Inherit函数实现函数模板的继承：

|  |
| --- |
| Handle<FunctionTemplate> biketemplate = FunctionTemplate::New();  biketemplate->PrototypeTemplate().Set(      String::New("wheels"),      FunctionTemplate::New(MyWheelsMethodCallback)->GetFunction();  )  void Inherit(Handle<FunctionTemplate> parent); |

### Isolate，Locker以及GC设置

多线程环境下，每个线程运行一个独立的v8虚拟机的话，就需要在线程初始化的时候创建Isolate::New()，在线程退出的时候调用Isolate::Dispose()。

如果每个线程都创建独立的Isolate运行独立的v8虚拟机的话，内存开销会很大，所以可以多线程共用Isolate，但这样的话就得加锁，使用Locker加锁。

v8的GC只有在托管内存超过一定阈值后才会触发垃圾回收，这对于嵌入到v8的对象来说，GC是无法知道对象的外部资源内存大小，有可能会造成内存紧张，所以需要手工触发GC。

AdjustAmountOfExternalAllocatedMemory(int change\_in\_bytes);使用这个API调整注册内存的真实数量，以让GC在合适的时机发起垃圾回收。

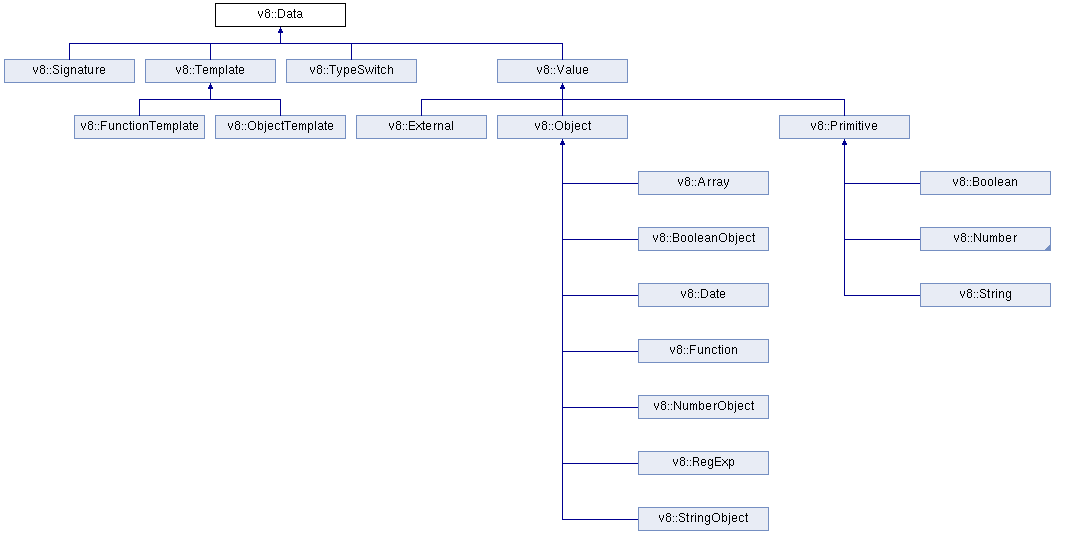
IdleNotification()当嵌入器空闲的时候进行资源清理；LowMemoryNotification()当内存过低时进行资源清理。

启动时设置内存相关参数SetFlagsFromCommandLine()和SetFlagsFromString()，

SetFlagsFromCommandLine()/SetFlagsFromString()应该在任何v8的API调用前调用。

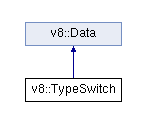
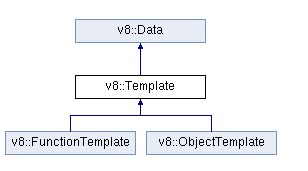
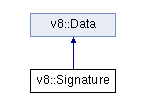
SetResourceContraints()可以在v8 vm初始化前调用设置Isolate相关Heap的大小，一旦vm初始化后，就无法再进行调整了。

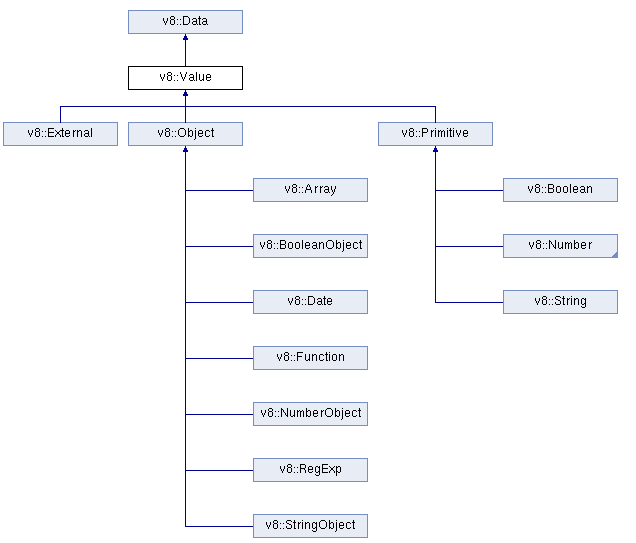
### v8的类型体系



v8的类型体系继承关系图如上，基类是v8::Data，v8::Data下有v8::Signature,v8::Template,v8::TypeSwitch,v8::Value几个子类。其中v8::Template下有两个子类， 分别是前几节介绍过的v8::FunctionTemplate和v8::ObjectTemplate。而v8::Value下面则有v8:External，v8::Object，v8::Primitive三个子类。v8::External我们在前面介绍v8::ObjectTemplate的内部成员持有外部对象的引用时有介绍过；而v8::Primitive是基本值类型，包括Boolean、Number、String；剩下的Arrray、BooleanObject、Date、Function、NumberObject、RegExp、StringObject都属于对象类型。

上图有点小，我们分开列出：





## v8 API手册

<http://www.bespin.cz/~ondras/html/annotated.html>

# CEF的v8::Extension扩展机制

CEF在content api基础上定制v8扩展，在上层封装了一套自己的类型和对象系统和对象生命周期管理，我们将逐一研究。

## CEF类型:CefV8Value

CEF定制了自己的类型系统，相关类：

接口类：CefV8Value

实现类：CefV8ValueImpl

|  |
| --- |
| class CefV8ValueImpl : public CefV8Value {  public:  CefV8ValueImpl();  explicit CefV8ValueImpl(v8::Handle<v8::Value> value);  virtual ~CefV8ValueImpl();  // Used for initializing the CefV8ValueImpl. Should be called a single time  // after the CefV8ValueImpl is created.  void InitFromV8Value(v8::Handle<v8::Value> value);  void InitUndefined();  void InitNull();  void InitBool(bool value);  void InitInt(int32 value);  void InitUInt(uint32 value);  void InitDouble(double value);  void InitDate(const CefTime& value);  void InitString(CefString& value);  void InitObject(v8::Handle<v8::Value> value, CefTrackNode\* tracker);  // Creates a new V8 value for the underlying value or returns the existing  // object handle.  v8::Handle<v8::Value> GetV8Value(bool should\_persist);  virtual bool IsValid() OVERRIDE;  virtual bool IsUndefined() OVERRIDE;  virtual bool IsNull() OVERRIDE;  virtual bool IsBool() OVERRIDE;  virtual bool IsInt() OVERRIDE;  virtual bool IsUInt() OVERRIDE;  virtual bool IsDouble() OVERRIDE;  virtual bool IsDate() OVERRIDE;  virtual bool IsString() OVERRIDE;  virtual bool IsObject() OVERRIDE;  virtual bool IsArray() OVERRIDE;  virtual bool IsFunction() OVERRIDE;  virtual bool IsSame(CefRefPtr<CefV8Value> value) OVERRIDE;  virtual bool GetBoolValue() OVERRIDE;  virtual int32 GetIntValue() OVERRIDE;  virtual uint32 GetUIntValue() OVERRIDE;  virtual double GetDoubleValue() OVERRIDE;  virtual CefTime GetDateValue() OVERRIDE;  virtual CefString GetStringValue() OVERRIDE;  virtual bool IsUserCreated() OVERRIDE;  virtual bool HasException() OVERRIDE;  virtual CefRefPtr<CefV8Exception> GetException() OVERRIDE;  virtual bool ClearException() OVERRIDE;  virtual bool WillRethrowExceptions() OVERRIDE;  virtual bool SetRethrowExceptions(bool rethrow) OVERRIDE;  virtual bool HasValue(const CefString& key) OVERRIDE;  virtual bool HasValue(int index) OVERRIDE;  virtual bool DeleteValue(const CefString& key) OVERRIDE;  virtual bool DeleteValue(int index) OVERRIDE;  virtual CefRefPtr<CefV8Value> GetValue(const CefString& key) OVERRIDE;  virtual CefRefPtr<CefV8Value> GetValue(int index) OVERRIDE;  virtual bool SetValue(const CefString& key, CefRefPtr<CefV8Value> value,  PropertyAttribute attribute) OVERRIDE;  virtual bool SetValue(int index, CefRefPtr<CefV8Value> value) OVERRIDE;  virtual bool SetValue(const CefString& key, AccessControl settings,  PropertyAttribute attribute) OVERRIDE;  virtual bool GetKeys(std::vector<CefString>& keys) OVERRIDE;  virtual bool SetUserData(CefRefPtr<CefBase> user\_data) OVERRIDE;  virtual CefRefPtr<CefBase> GetUserData() OVERRIDE;  virtual int GetExternallyAllocatedMemory() OVERRIDE;  virtual int AdjustExternallyAllocatedMemory(int change\_in\_bytes) OVERRIDE;  virtual int GetArrayLength() OVERRIDE;  virtual CefString GetFunctionName() OVERRIDE;  virtual CefRefPtr<CefV8Handler> GetFunctionHandler() OVERRIDE;  virtual CefRefPtr<CefV8Value> ExecuteFunction(  CefRefPtr<CefV8Value> object,  const CefV8ValueList& arguments) OVERRIDE;  virtual CefRefPtr<CefV8Value> ExecuteFunctionWithContext(  CefRefPtr<CefV8Context> context,  CefRefPtr<CefV8Value> object,  const CefV8ValueList& arguments) OVERRIDE;  protected:  //Other  } |

## CefValueImpl的存储

CefV8ValueImpl内部用一个Union存储实际的C++基本内置类型，用内置的Handle类型存储对象、函数和数组，注意这个Handle对象并不是v8::Handle，而是在内部持有v8::Handle。这点跟Lua的TValue类似：

|  |
| --- |
| union {  bool bool\_value\_;  int32 int\_value\_;  uint32 uint\_value\_;  double double\_value\_;  cef\_time\_t date\_value\_;  cef\_string\_t string\_value\_;  };  // Used with Object, Function and Array types.  scoped\_refptr<Handle> handle\_; |

同时用一个枚举标识当前存储的C++类型：

|  |
| --- |
| enum {  TYPE\_INVALID = 0,  TYPE\_UNDEFINED,  TYPE\_NULL,  TYPE\_BOOL,  TYPE\_INT,  TYPE\_UINT,  TYPE\_DOUBLE,  TYPE\_DATE,  TYPE\_STRING,  TYPE\_OBJECT,  } type\_; |

CefV8ValueImpl需要注册给JavaScript环境，那么也需要遵循我们在第三章描述的注册类型给JavaScript需要的注册方法。

首先我们知道需要使用句柄持有CefV8ValueImpl对象，那么我们知道句柄有两种，一种是局部的，一种是持久的。CefV8ValueImpl内部使用封装了Handle内部类用来持有对象，Handle内部用一个持久句柄保存对象的引用，通过GetNewHandle返回局部句柄给外部使用，当然也可以通过GetPersistentHandle直接获取持久句柄。

|  |
| --- |
| class Handle : public CefV8HandleBase {  public:  typedef v8::Handle<v8::Value> handleType;  typedef v8::Persistent<v8::Value> persistentType;  //构造的时候指定了v8::Context，以及CefTrackNode,CefTrackNode在后面解析  Handle(v8::Handle<v8::Context> context, handleType v, CefTrackNode\* tracker)  : CefV8HandleBase(context), //将context交给父类管理  handle\_(isolate(), v), //通过父类方法获取v8::Isolate  tracker\_(tracker),  should\_persist\_(false) {  }  virtual ~Handle();  //创建一个新的局部句柄给外部使用。  handleType GetNewV8Handle(bool should\_persist) {  DCHECK(IsValid());  if (should\_persist && !should\_persist\_)//是否应该持久  should\_persist\_ = true;  return handleType::New(isolate(), handle\_); //通过父类方法获取v8::Isolate  }  persistentType& GetPersistentV8Handle() {  return handle\_;  }  private:  persistentType handle\_;  // For Object and Function types, we need to hold on to a reference to their  // internal data or function handler objects that are reference counted.  CefTrackNode\* tracker\_;  // True if the handle needs to persist due to it being passed into V8.  bool should\_persist\_;  DISALLOW\_COPY\_AND\_ASSIGN(Handle);  }; |

从v8的概念我们知道局部句柄用v8::HandleScope管理生命周期，执行JavaScript脚本则需要在v8::Context里，而多线程环境下需要使用Isolate隔离不同的v8虚拟机。从Handle的构造函数我们看到Context被设置给了父类CefHandleBase的构造函数。我们看下该类：

|  |
| --- |
| // Base class for V8 Handle types.  class CefV8HandleBase :  public base::RefCountedThreadSafe<CefV8HandleBase,  CefV8DeleteOnMessageLoopThread> {  public:  virtual ~CefV8HandleBase();  // Returns true if there is no underlying context or if the underlying context  // is valid.  bool IsValid() const {  return (!context\_state\_.get() || context\_state\_->IsValid());  }  bool BelongsToCurrentThread() const;  v8::Isolate\* isolate() const { return isolate\_; }  scoped\_refptr<base::SequencedTaskRunner> task\_runner() const {  return task\_runner\_;  }  protected:  // |context| is the context that owns this handle. If empty the current  // context will be used.  explicit CefV8HandleBase(v8::Handle<v8::Context> context);  protected:  v8::Isolate\* isolate\_;  scoped\_refptr<base::SequencedTaskRunner> task\_runner\_;  scoped\_refptr<CefV8ContextState> context\_state\_;  };  //我们重点看这个构造函数  CefV8HandleBase::CefV8HandleBase(v8::Handle<v8::Context> context) {  //获取CefV8IsolateManager，我们在4.1.3剖析  CefV8IsolateManager\* manager = GetIsolateManager();  DCHECK(manager);    //获取Isolate  isolate\_ = manager->isolate();  //获取TaskRunner，这个在后面解析，暂时不理会  task\_runner\_ = manager->task\_runner();  //将构造函数传入的Context封装成CefContextState  context\_state\_ = manager->GetContextState(context);  } |

## 基本内置类型

对于值类型，CefV8ValueImpl提供了简单的初始化、判断、获取、设置等方法：

|  |
| --- |
| void CefV8ValueImpl::InitDouble(double value) {  DCHECK\_EQ(type\_, TYPE\_INVALID);  type\_ = TYPE\_DOUBLE;  double\_value\_ = value;  }  bool CefV8ValueImpl::IsDouble() {  CEF\_V8\_REQUIRE\_ISOLATE\_RETURN(false);  return (type\_ == TYPE\_INT || type\_ == TYPE\_UINT || type\_ == TYPE\_DOUBLE);  }  double CefV8ValueImpl::GetDoubleValue() {  CEF\_V8\_REQUIRE\_ISOLATE\_RETURN(0.);  if (type\_ == TYPE\_DOUBLE)  return double\_value\_;  else if (type\_ == TYPE\_INT)  return int\_value\_;  else if (type\_ == TYPE\_UINT)  return uint\_value\_;  return 0.;  } |

而在CefV8Value里提供静态函数用于创建基本内置类型：

|  |
| --- |
| // static  CefRefPtr<CefV8Value> CefV8Value::CreateDouble(double value) {  CEF\_V8\_REQUIRE\_ISOLATE\_RETURN(NULL);  CefRefPtr<CefV8ValueImpl> impl = new CefV8ValueImpl();  impl->InitDouble(value);  return impl.get();  } |

## Object和UserData

对于对象类型，CefV8ValueImpl则需要通过Handle存储。

|  |
| --- |
| void CefV8ValueImpl::InitObject(v8::Handle<v8::Value> value, CefTrackNode\* tracker) {  DCHECK\_EQ(type\_, TYPE\_INVALID);  type\_ = TYPE\_OBJECT;  handle\_ = new Handle(v8::Handle<v8::Context>(), value, tracker);  }  bool CefV8ValueImpl::IsObject() {  CEF\_V8\_REQUIRE\_ISOLATE\_RETURN(false);  return (type\_ == TYPE\_OBJECT);  } |

下面的方法允许将用户自定义类型（继承自CefBase）存储在TYPE\_OBJECT类型的CefV8Value里。

|  |
| --- |
| bool CefV8ValueImpl::SetUserData(CefRefPtr<CefBase> user\_data) {  CEF\_V8\_REQUIRE\_OBJECT\_RETURN(false);  v8::HandleScope handle\_scope(handle\_->isolate());  v8::Handle<v8::Value> value = handle\_->GetNewV8Handle(false);  v8::Handle<v8::Object> obj = value->ToObject();  V8TrackObject\* tracker = V8TrackObject::Unwrap(obj);  if (tracker) {  tracker->SetUserData(user\_data);  return true;  }  return false;  }  CefRefPtr<CefBase> CefV8ValueImpl::GetUserData() {  CEF\_V8\_REQUIRE\_OBJECT\_RETURN(NULL);  v8::HandleScope handle\_scope(handle\_->isolate());  v8::Handle<v8::Value> value = handle\_->GetNewV8Handle(false);  v8::Handle<v8::Object> obj = value->ToObject();  V8TrackObject\* tracker = V8TrackObject::Unwrap(obj);  if (tracker)  return tracker->GetUserData();  return NULL;  }  bool CefV8ValueImpl::IsUserCreated() {  CEF\_V8\_REQUIRE\_OBJECT\_RETURN(false);  v8::HandleScope handle\_scope(handle\_->isolate());  v8::Handle<v8::Value> value = handle\_->GetNewV8Handle(false);  v8::Handle<v8::Object> obj = value->ToObject();  V8TrackObject\* tracker = V8TrackObject::Unwrap(obj);  return (tracker != NULL);  } |

CefV8Value类提供了静态方法用于创建TYPE\_OBJECT类型的CefV8Value：

|  |
| --- |
| CefRefPtr<CefV8Value> CefV8Value::CreateObject(  CefRefPtr<CefV8Accessor> accessor) {  CEF\_V8\_REQUIRE\_ISOLATE\_RETURN(NULL);  //与基本内置类型不同，TYPE\_OBJECT不是值类型，所以需要在栈上使用局部句柄引用之，  //从而需要句柄范围自动管理局部句柄的生命周期。  v8::HandleScope handle\_scope(v8::Isolate::GetCurrent());  //获取v8虚拟机的当前上下文  v8::Local<v8::Context> context = v8::Context::GetCurrent();  if (context.IsEmpty()) {  NOTREACHED() << "not currently in a V8 context";  return NULL;  }  // 创建一个v8::Object,返回栈上的局部句柄  v8::Local<v8::Object> obj = v8::Object::New();  // 创建一个V8TrackObject,用于保存存取器(accessor).  // 后面解析字典（数组）的相关函数的时候会用到这个  // Create a tracker object that will cause the user data and/or accessor  // reference to be released when the V8 object is destroyed.  V8TrackObject\* tracker = new V8TrackObject;  tracker->SetAccessor(accessor);  // Attach the tracker object.  tracker->AttachTo(obj); //将tracker和obj绑定在一起  CefRefPtr<CefV8ValueImpl> impl = new CefV8ValueImpl();  impl->InitObject(obj, tracker); //初始化CefV8ValueImpl  return impl.get();  } |

V8TrackObject我们在后面单独解析。目前只需知道在创建Object时，同时创建了一个V8TrackObject，用于保存obj的Accessor，然后将obj绑定到tracker，最后将obj和tracker同时组合到CefV8ValueImpl里面，在后续的Object相关的操作中，会用到tracker对象。比如获取obj的accessor。

## InitFromV8Value和GetV8Value

从而，CefV8ValueImpl内部将v8::Value转换成了内部的类型体系：

|  |
| --- |
| void CefV8ValueImpl::InitFromV8Value(v8::Handle<v8::Value> value) {  if (value->IsUndefined()) {  InitUndefined();  } else if (value->IsNull()) {  InitNull();  } else if (value->IsTrue()) {  InitBool(true);  } else if (value->IsFalse()) {  InitBool(false);  } else if (value->IsBoolean()) {  InitBool(value->ToBoolean()->Value());  } else if (value->IsInt32()) {  InitInt(value->ToInt32()->Value());  } else if (value->IsUint32()) {  InitUInt(value->ToUint32()->Value());  } else if (value->IsNumber()) {  InitDouble(value->ToNumber()->Value());  } else if (value->IsDate()) {  // Convert from milliseconds to seconds.  InitDate(CefTime(value->ToNumber()->Value() / 1000));  } else if (value->IsString()) {  CefString rv;  GetCefString(value->ToString(), rv);  InitString(rv);  } else if (value->IsObject()) {  InitObject(value, NULL);  }  }  v8::Handle<v8::Value> CefV8ValueImpl::GetV8Value(bool should\_persist) {  switch (type\_) {  case TYPE\_UNDEFINED:  return v8::Undefined();  case TYPE\_NULL:  return v8::Null();  case TYPE\_BOOL:  return v8::Boolean::New(bool\_value\_);  case TYPE\_INT:  return v8::Int32::New(int\_value\_);  case TYPE\_UINT:  return v8::Uint32::New(uint\_value\_);  case TYPE\_DOUBLE:  return v8::Number::New(double\_value\_);  case TYPE\_DATE:  // Convert from seconds to milliseconds.  return v8::Date::New(CefTime(date\_value\_).GetDoubleT() \* 1000);  case TYPE\_STRING:  return GetV8String(CefString(&string\_value\_));  case TYPE\_OBJECT:  return handle\_->GetNewV8Handle(should\_persist);  default:  break;  } |

同时，CefV8ValueImpl通过先比较类型，再比较具体的值的方式比较两个CefV8Value是否相等：

|  |
| --- |
| bool CefV8ValueImpl::IsSame(CefRefPtr<CefV8Value> that) {  CEF\_V8\_REQUIRE\_MLT\_RETURN(false);  CefV8ValueImpl\* thatValue = static\_cast<CefV8ValueImpl\*>(that.get());  if (!thatValue || !thatValue->IsValid() || type\_ != thatValue->type\_)  return false;  switch (type\_) {  case TYPE\_UNDEFINED:  case TYPE\_NULL:  return true;  case TYPE\_BOOL:  return (bool\_value\_ == thatValue->bool\_value\_);  case TYPE\_INT:  return (int\_value\_ == thatValue->int\_value\_);  case TYPE\_UINT:  return (uint\_value\_ == thatValue->uint\_value\_);  case TYPE\_DOUBLE:  return (double\_value\_ == thatValue->double\_value\_);  case TYPE\_DATE:  return (CefTime(date\_value\_).GetTimeT() ==  CefTime(thatValue->date\_value\_).GetTimeT());  case TYPE\_STRING:  return (CefString(&string\_value\_) ==  CefString(&thatValue->string\_value\_));  case TYPE\_OBJECT: {  return (handle\_->GetPersistentV8Handle() ==  thatValue->handle\_->GetPersistentV8Handle());  }  default:  break;  }  return false;  } |

## 函数

Cef也将函数、字典、数组（下标为0-base连续整数的字典）存储在Object里，我们首先看下如果Object是一个v8::Function，如何获取函数名和函数体：

|  |
| --- |
| CefString CefV8ValueImpl::GetFunctionName() {  CefString rv;  CEF\_V8\_REQUIRE\_OBJECT\_RETURN(rv);  v8::HandleScope handle\_scope(handle\_->isolate());  v8::Handle<v8::Value> value = handle\_->GetNewV8Handle(false);  //根据v8::Value的方法判断是否是函数类型  if (!value->IsFunction()) {  NOTREACHED() << "V8 value is not a function";  return rv;  }  //通过v8::Value::ToObject()将v8::Value转成v8::Object  //通过v8::Cast将v8::Object转成v8::Function  v8::Handle<v8::Object> obj = value->ToObject();  v8::Handle<v8::Function> func = v8::Handle<v8::Function>::Cast(obj);  GetCefString(v8::Handle<v8::String>::Cast(func->GetName()), rv);  return rv;  }  CefRefPtr<CefV8Handler> CefV8ValueImpl::GetFunctionHandler() {  CEF\_V8\_REQUIRE\_OBJECT\_RETURN(NULL);  //校验是否是v8::Function对象  v8::HandleScope handle\_scope(handle\_->isolate());  v8::Handle<v8::Value> value = handle\_->GetNewV8Handle(false);  if (!value->IsFunction()) {  NOTREACHED() << "V8 value is not a function";  return 0;  }  //通过Cef提供的V8TrackObject将v8::Function转成CefV8Handler  //Cef的TrackObject系统会在后面介绍  v8::Handle<v8::Object> obj = value->ToObject();  V8TrackObject\* tracker = V8TrackObject::Unwrap(obj);  if (tracker)  return tracker->GetHandler();  return NULL;  } |

同样的，CefV8Value提供了静态方法用于创建函数：

|  |
| --- |
| CefRefPtr<CefV8Value> CefV8Value::CreateFunction(  const CefString& name,  CefRefPtr<CefV8Handler> handler) {  CEF\_V8\_REQUIRE\_ISOLATE\_RETURN(NULL);  if (!handler.get()) {  NOTREACHED() << "invalid parameter";  return NULL;  }  v8::HandleScope handle\_scope(v8::Isolate::GetCurrent());  v8::Local<v8::Context> context = v8::Context::GetCurrent();  if (context.IsEmpty()) {  NOTREACHED() << "not currently in a V8 context";  return NULL;  }    //创建函数模板  // Create a new V8 function template.  v8::Local<v8::FunctionTemplate> tmpl = v8::FunctionTemplate::New();  //将CefV8Handler对象保存到一个External对象里  //注意这个是CefV8Handler，不是CefV8Handle，这是两个不同功能的类！  v8::Local<v8::Value> data = v8::External::New(handler.get());  //设置函数模板的回调函数FunctionCallbackImp以及数据data  //在FunctionCallbackImpl函数里会解析data数据  // Set the function handler callback.  tmpl->SetCallHandler(FunctionCallbackImpl, data);  //从函数模板里获取函数对象  // Retrieve the function object and set the name.  v8::Local<v8::Function> func = tmpl->GetFunction();  if (func.IsEmpty()) {  NOTREACHED() << "failed to create V8 function";  return NULL;  }  func->SetName(GetV8String(name));  //在tracker里保留handler  // Create a tracker object that will cause the user data and/or handler  // reference to be released when the V8 object is destroyed.  V8TrackObject\* tracker = new V8TrackObject;  tracker->SetHandler(handler);  // Attach the tracker object.  tracker->AttachTo(func);//绑定到func  //将func和tracker初始化到CefV8Value  // Create the CefV8ValueImpl and provide a tracker object that will cause  // the handler reference to be released when the V8 object is destroyed.  CefRefPtr<CefV8ValueImpl> impl = new CefV8ValueImpl();  impl->InitObject(func, tracker);  return impl.get();  } |

CreateFunction的代码里设置了函数回调,这个函数如下:

|  |
| --- |
| // V8 function callback.  // 这个函数回调与第二章翻译的v8的教程里的函数回调原型有所不一样，应该属于正常的代码  // 变动，不过并不影响我们对代码的理解。  void FunctionCallbackImpl(const v8::FunctionCallbackInfo<v8::Value>& info) {  //TODO:V8RecursionScope是干什么的？  WebCore::V8RecursionScope recursion\_scope(  WebCore::toExecutionContext(v8::Context::GetCurrent()));  //获取CreateFunction里设置到模板里的External数据，并转回CefV8Handler  //从这里看是被保存早FunctionCallbackInfo里  CefV8Handler\* handler =  static\_cast<CefV8Handler\*>(v8::External::Cast(\*info.Data())->Value());    //从info里获取函数参数  CefV8ValueList params;  for (int i = 0; i < info.Length(); i++)  params.push\_back(new CefV8ValueImpl(info[i]));  //从info里获取函数名字  CefString func\_name;  GetCefString(v8::Handle<v8::String>::Cast(info.Callee()->GetName()),  func\_name);  //从info里获取函数的接收者  CefRefPtr<CefV8Value> object = new CefV8ValueImpl(info.This());    //声明返回值和异常  CefRefPtr<CefV8Value> retval;  CefString exception;  //调用Handler的Execute执行函数并返回结果或处理异常  if (handler->Execute(func\_name, object, params, retval, exception)) {  if (!exception.empty()) {  info.GetReturnValue().Set(  v8::ThrowException(v8::Exception::Error(GetV8String(exception))));  return;  } else {  CefV8ValueImpl\* rv = static\_cast<CefV8ValueImpl\*>(retval.get());  if (rv && rv->IsValid()) {  info.GetReturnValue().Set(rv->GetV8Value(true));  return;  }  }  }  info.GetReturnValue().SetUndefined();  } |

后面我们会解析如何将CreateFunction注册个v8的JavaScript环境。此次我们只要理解JavaScript代码在调用我们注册给它的函数时，会调用此处的函数回调，从上面代码可知，FunctionCallbackImpl提供了一个标准的回调函数，在内部将函数名字，函数接收者、函数参数，返回值，异常等数据转发给CefV8Handle的Execute方法，所以我们只需在CefV8Hanlde的Execute方法里处理JavaScript的函数调用即可。

另一方方面，我们也希望在C++环境里调用JavaScript的代码。其基本思想还是将C++的函数参数类型（比如CefValue）转成v8的相应参数类型，然后通过v8的API来执行v8::Function，并返回结果。我们逐一分析这个过程，首先是参数类型转换过程：

|  |
| --- |
| CefRefPtr<CefV8Value> CefV8ValueImpl::ExecuteFunction(  CefRefPtr<CefV8Value> object,  const CefV8ValueList& arguments) {  // An empty context value defaults to the current context.  CefRefPtr<CefV8Context> context;  return ExecuteFunctionWithContext(context, object, arguments);  }  //上面的函数将调用转发给ExecuteFunctionWithContext，我们知道v8的JavaScript代码  //必须在独立的Context里执行，同时如果需要多线程隔离的话，需要为Context指定独立的  //Isolate  CefRefPtr<CefV8Value> CefV8ValueImpl::ExecuteFunctionWithContext(  CefRefPtr<CefV8Context> context,  CefRefPtr<CefV8Value> object,  const CefV8ValueList& arguments) {  CEF\_V8\_REQUIRE\_OBJECT\_RETURN(NULL);  //如前所述，我们需要一个HandleScope来自动管理局部栈上的对象生命周期  v8::HandleScope handle\_scope(handle\_->isolate());  v8::Handle<v8::Value> value = handle\_->GetNewV8Handle(false);  if (!value->IsFunction()) {  NOTREACHED() << "V8 value is not a function";  return 0;  }    //下面分别校验上下文、调用对象、参数列表的有效性  if (context.get() && !context->IsValid()) {  NOTREACHED() << "invalid V8 context parameter";  return NULL;  }  if (object.get() && (!object->IsValid() || !object->IsObject())) {  NOTREACHED() << "invalid V8 object parameter";  return NULL;  }  int argc = arguments.size();  if (argc > 0) {  for (int i = 0; i < argc; ++i) {  if (!arguments[i].get() || !arguments[i]->IsValid()) {  NOTREACHED() << "invalid V8 arguments parameter";  return NULL;  }  }  }  //将CefV8Context转换成v8::Context  v8::Local<v8::Context> context\_local;  if (context.get()) {  CefV8ContextImpl\* context\_impl =  static\_cast<CefV8ContextImpl\*>(context.get());  context\_local = context\_impl->GetV8Context();  } else {  context\_local = v8::Context::GetCurrent();  }  //TODO：说明作用；创建上下文范围  v8::Context::Scope context\_scope(context\_local);  //将自身从Object类型转成v8::Function  v8::Handle<v8::Object> obj = value->ToObject();  v8::Handle<v8::Function> func = v8::Handle<v8::Function>::Cast(obj);  //将调用对象object转成v8::Object类型，  v8::Handle<v8::Object> recv;  // Default to the global object if no object was provided.  if (object.get()) {  CefV8ValueImpl\* recv\_impl = static\_cast<CefV8ValueImpl\*>(object.get());  recv = v8::Handle<v8::Object>::Cast(recv\_impl->GetV8Value(true));  } else {  recv = context\_local->Global();  }  //将参数列表转成v8::Array  v8::Handle<v8::Value> \*argv = NULL;  if (argc > 0) {  argv = new v8::Handle<v8::Value>[argc];  for (int i = 0; i < argc; ++i) {  argv[i] =  static\_cast<CefV8ValueImpl\*>(arguments[i].get())->GetV8Value(true);  }  }    //在v8::TryCatch环境下执行代码，并返回结果  CefRefPtr<CefV8Value> retval;  {  v8::TryCatch try\_catch;  try\_catch.SetVerbose(true);//TODO：说明作用  v8::Local<v8::Value> func\_rv =  CallV8Function(context\_local, func, recv, argc, argv, handle\_->isolate());  if (!HasCaught(try\_catch) && !func\_rv.IsEmpty())  retval = new CefV8ValueImpl(func\_rv);  }  if (argv)  delete [] argv;  return retval;  } |

上面的代码创建了句柄范围，创建了上下文范围，然后将调用对象、参数列表转换成v8的类型系统，最后在v8的异常处理环境下调用函数。最后黄色标注的那一行将函数调用转发给CallV8Function，我们进一步看下这个函数做了什么。

|  |
| --- |
| v8::Local<v8::Value> CallV8Function(v8::Handle<v8::Context> context,  v8::Handle<v8::Function> function,  v8::Handle<v8::Object> receiver,  int argc,  v8::Handle<v8::Value> args[],  v8::Isolate\* isolate) {  v8::Local<v8::Value> func\_rv;  // Execute the function call using the ScriptController so that inspector  // instrumentation works.  if (CEF\_CURRENTLY\_ON\_RT()) {  //如果当前线程在RenderThread上，直接通过Frame获取 ScriptController  //然后调用函数  RefPtr<WebCore::Frame> frame = WebCore::toFrameIfNotDetached(context);  DCHECK(frame);  if (frame &&  frame->script().canExecuteScripts(WebCore::AboutToExecuteScript)) {  func\_rv = frame->script().callFunction(function, receiver, argc, args);  }  } else {  //如果不在RunerThread主线程上，则通过WorkerScriptController执行函数  //这需要将context，isolate传入到工作线程。  WebCore::WorkerScriptController\* controller =  WebCore::WorkerScriptController::controllerForContext();  DCHECK(controller);  if (controller) {  func\_rv = WebCore::ScriptController::callFunction(  controller->workerGlobalScope().executionContext(),  function, receiver, argc, args, isolate);  }  }  return func\_rv;  } |

至此，Cef完成了C++和JavaScript函数互相调用的机制。

## 字典

CefV8Value的Object亦可以是字典，字典的下标可以是整数或字符串，所以可以用字典模拟数组。

首先看下是否包含某个键的判断：

|  |
| --- |
| bool CefV8ValueImpl::HasValue(const CefString& key) {  CEF\_V8\_REQUIRE\_OBJECT\_RETURN(false);  //通过HandleScope管理局部句柄对象生命周期  //此处并无调用v8::Function，所以不需要使用上下文  v8::HandleScope handle\_scope(handle\_->isolate());  v8::Handle<v8::Value> value = handle\_->GetNewV8Handle(false);  v8::Handle<v8::Object> obj = value->ToObject();  return obj->Has(GetV8String(key));  }  bool CefV8ValueImpl::HasValue(int index) {  CEF\_V8\_REQUIRE\_OBJECT\_RETURN(false);  if (index < 0) {  NOTREACHED() << "invalid input parameter";  return false;  }    //通过HandleScope管理局部句柄对象生命周期  //此处并无调用v8::Function，所以不需要使用上下文  v8::HandleScope handle\_scope(handle\_->isolate());  v8::Handle<v8::Value> value = handle\_->GetNewV8Handle(false);  v8::Handle<v8::Object> obj = value->ToObject();  return obj->Has(index);  } |
|  |

基本上就是将handle转换成v8::Object后，通过v8::Object::Has方法判断是否含有字符串键或者整形键。Has是v8::Object的成员方法：

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| |  |  | | --- | --- | | V8EXPORT bool | [**Has**](http://izs.me/v8-docs/classv8_1_1Object.html#a12a53d31423d7da6da3960cde76bfea5) (uint32\_t index) | | V8EXPORT bool | [**Has**](http://izs.me/v8-docs/classv8_1_1Object.html#ada7a824246df2e8a9b7476bd63a54b2e) ([**Handle**](http://izs.me/v8-docs/classv8_1_1Handle.html)< [**String**](http://izs.me/v8-docs/classv8_1_1String.html) > key) | |

下面是获取键值的代码：

|  |
| --- |
| CefRefPtr<CefV8Value> CefV8ValueImpl::GetValue(const CefString& key) {  CEF\_V8\_REQUIRE\_OBJECT\_RETURN(NULL);    //通过HandleScope管理局部句柄对象生命周期  //此处并无调用v8::Function，所以不需要使用上下文  v8::HandleScope handle\_scope(handle\_->isolate());  v8::Handle<v8::Value> value = handle\_->GetNewV8Handle(false);  v8::Handle<v8::Object> obj = value->ToObject();  //字典的键可能不存在，此时会抛出异常，所以需要用v8::TryCatch做出错处理  v8::TryCatch try\_catch;  try\_catch.SetVerbose(true);  v8::Local<v8::Value> ret\_value = obj->Get(GetV8String(key));  if (!HasCaught(try\_catch) && !ret\_value.IsEmpty())  return new CefV8ValueImpl(ret\_value);  return NULL;  }  CefRefPtr<CefV8Value> CefV8ValueImpl::GetValue(int index) {  CEF\_V8\_REQUIRE\_OBJECT\_RETURN(NULL);  if (index < 0) {  NOTREACHED() << "invalid input parameter";  return NULL;  }    //通过HandleScope管理局部句柄对象生命周期  //此处并无调用v8::Function，所以不需要使用上下文  v8::HandleScope handle\_scope(handle\_->isolate());  v8::Handle<v8::Value> value = handle\_->GetNewV8Handle(false);  v8::Handle<v8::Object> obj = value->ToObject();  //字典的键可能不存在，此时会抛出异常，所以需要用v8::TryCatch做出错处理  v8::TryCatch try\_catch;  try\_catch.SetVerbose(true);  v8::Local<v8::Value> ret\_value = obj->Get(v8::Number::New(index));  if (!HasCaught(try\_catch) && !ret\_value.IsEmpty())  return new CefV8ValueImpl(ret\_value);  return NULL;  } |

转型成v8::Object后调用v8::Object::Get的方法：

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| |  |  | | --- | --- | | V8EXPORT [**Local**](http://izs.me/v8-docs/classv8_1_1Local.html)< [**Value**](http://izs.me/v8-docs/classv8_1_1Value.html) > | [**Get**](http://izs.me/v8-docs/classv8_1_1Object.html#a59bd849c1cdb2aa1033423ee21b2ede9) ([**Handle**](http://izs.me/v8-docs/classv8_1_1Handle.html)< [**Value**](http://izs.me/v8-docs/classv8_1_1Value.html) > key) | | V8EXPORT [**Local**](http://izs.me/v8-docs/classv8_1_1Local.html)< [**Value**](http://izs.me/v8-docs/classv8_1_1Value.html) > | [**Get**](http://izs.me/v8-docs/classv8_1_1Object.html#a7310ce3ee18744c0971356dad8e2bed1) (uint32\_t index) | |

下面是设置指定键的值，与GetValue相比，：

|  |
| --- |
| bool CefV8ValueImpl::SetValue(const CefString& key,  CefRefPtr<CefV8Value> value,  PropertyAttribute attribute) {  CEF\_V8\_REQUIRE\_OBJECT\_RETURN(false);  CefV8ValueImpl\* impl = static\_cast<CefV8ValueImpl\*>(value.get());  if (impl && impl->IsValid()) {  v8::HandleScope handle\_scope(handle\_->isolate());  v8::Handle<v8::Value> value = handle\_->GetNewV8Handle(false);  v8::Handle<v8::Object> obj = value->ToObject();  v8::TryCatch try\_catch;  try\_catch.SetVerbose(true);  bool set = obj->Set(GetV8String(key), impl->GetV8Value(true),  static\_cast<v8::PropertyAttribute>(attribute));  return (!HasCaught(try\_catch) && set);  } else {  NOTREACHED() << "invalid input parameter";  return false;  }  }  bool CefV8ValueImpl::SetValue(int index, CefRefPtr<CefV8Value> value) {  CEF\_V8\_REQUIRE\_OBJECT\_RETURN(false);  if (index < 0) {  NOTREACHED() << "invalid input parameter";  return false;  }  CefV8ValueImpl\* impl = static\_cast<CefV8ValueImpl\*>(value.get());  if (impl && impl->IsValid()) {  v8::HandleScope handle\_scope(handle\_->isolate());  v8::Handle<v8::Value> value = handle\_->GetNewV8Handle(false);  v8::Handle<v8::Object> obj = value->ToObject();  v8::TryCatch try\_catch;  try\_catch.SetVerbose(true);  bool set = obj->Set(index, impl->GetV8Value(true));  return (!HasCaught(try\_catch) && set);  } else {  NOTREACHED() << "invalid input parameter";  return false;  }  } |

同样，最后都转发到了v8::Object::Set方法：

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| |  |  | | --- | --- | | V8EXPORT bool | [**Set**](http://izs.me/v8-docs/classv8_1_1Object.html#aa588c4de7f0e4db47a2f22acfaa774fd) (uint32\_t index, [**Handle**](http://izs.me/v8-docs/classv8_1_1Handle.html)< [**Value**](http://izs.me/v8-docs/classv8_1_1Value.html) > value) | | V8EXPORT bool | [**Set**](http://izs.me/v8-docs/classv8_1_1Object.html#a97717c7b7fdc556c3a7fad14877ca912) ([**Handle**](http://izs.me/v8-docs/classv8_1_1Handle.html)< [**Value**](http://izs.me/v8-docs/classv8_1_1Value.html) > key, [**Handle**](http://izs.me/v8-docs/classv8_1_1Handle.html)< [**Value**](http://izs.me/v8-docs/classv8_1_1Value.html) > value, [**PropertyAttribute**](http://izs.me/v8-docs/namespacev8.html#a05f25f935e108a1ea2d150e274602b87) attribs=None) | |

其中，PropertyAttribute是一个枚举，配置value类型，一个有下面四种：

|  |
| --- |
| None、ReadOnly 、DontEnum、DontDelete |

SetValue还有一个重载函数，提供了设置访问控制的能力：

|  |
| --- |
| bool CefV8ValueImpl::SetValue(const CefString& key, AccessControl settings,  PropertyAttribute attribute) {  CEF\_V8\_REQUIRE\_OBJECT\_RETURN(false);  v8::HandleScope handle\_scope(handle\_->isolate());  v8::Handle<v8::Value> value = handle\_->GetNewV8Handle(false);  v8::Handle<v8::Object> obj = value->ToObject();  CefRefPtr<CefV8Accessor> accessorPtr;  ///获取V8TrackObject，回忆下CefV8Value::CreateObject里的tracker->AttachTo(obj)  V8TrackObject\* tracker = V8TrackObject::Unwrap(obj);  //获取访问存取器  //注意tracker->GetAccessor所获得的存取器是在CefV8Value::CreateObject里设置的  if (tracker)  accessorPtr = tracker->GetAccessor();  //如果访问控制器不存在则返回  if (!accessorPtr.get())  return false;    //设置存取器的getter和setter  v8::AccessorGetterCallback getter = AccessorGetterCallbackImpl;  v8::AccessorSetterCallback setter =  (attribute & V8\_PROPERTY\_ATTRIBUTE\_READONLY) ?  NULL : AccessorSetterCallbackImpl; //如果只读，则不提供setter  //错误处理  v8::TryCatch try\_catch;  try\_catch.SetVerbose(true);  //设置存取器  bool set = obj->SetAccessor(GetV8String(key), getter, setter, obj,  static\_cast<v8::AccessControl>(settings),  static\_cast<v8::PropertyAttribute>(attribute));  return (!HasCaught(try\_catch) && set);  } |

其中，AccessControl是一个枚举，有下面四种值：

|  |
| --- |
| DEFAULT、ALL\_CAN\_READ、ALL\_CAN\_WRITE、PROHIBITS\_OVERWRITING |

上述代码里用到了AccessorGetterCallbackImpl和AccessorSetterCallbackImpl。与FunctionCallbackImpl类似，这是Cef提供的通用存取器Getter和Setter回调函数，内部必然需要将具体的操作做适当转发，我们实际探究下代码：

|  |
| --- |
| // V8 Accessor callbacks  void AccessorGetterCallbackImpl(  v8::Local<v8::String> property,  const v8::PropertyCallbackInfo<v8::Value>& info) {  //TODO:WebCore::V8RecursionScope的作用？  WebCore::V8RecursionScope recursion\_scope(  WebCore::toExecutionContext(v8::Context::GetCurrent()));  //与FunctionCallbackImpl一样，info.This()获取存取器的接收者  v8::Handle<v8::Object> obj = info.This();  //获取在SetValue里绑定到obj的tracker,然后通过tracker获取在  //SetValue里设置到tracker的accessor  CefRefPtr<CefV8Accessor> accessorPtr;  V8TrackObject\* tracker = V8TrackObject::Unwrap(obj);  if (tracker)  accessorPtr = tracker->GetAccessor();    //下面的代码可以看到最终将代码转发到SetValue里设置的Accessor的Get方法  if (accessorPtr.get()) {  CefRefPtr<CefV8Value> retval;  CefRefPtr<CefV8Value> object = new CefV8ValueImpl(obj);  CefString name, exception;  GetCefString(property, name);  if (accessorPtr->Get(name, object, retval, exception)) {  if (!exception.empty()) {  info.GetReturnValue().Set(  v8::ThrowException(v8::Exception::Error(GetV8String(exception))));  return;  } else {  CefV8ValueImpl\* rv = static\_cast<CefV8ValueImpl\*>(retval.get());  if (rv && rv->IsValid()) {  info.GetReturnValue().Set(rv->GetV8Value(true));  return;  }  }  }  }  return info.GetReturnValue().SetUndefined();  }  void AccessorSetterCallbackImpl(  v8::Local<v8::String> property,  v8::Local<v8::Value> value,  const v8::PropertyCallbackInfo<void>& info) {  // TODO:WebCore::V8RecursionScope的作用？  WebCore::V8RecursionScope recursion\_scope(  WebCore::toExecutionContext(v8::Context::GetCurrent()));  //与FunctionCallbackImpl一样，info.This()获取存取器的接收者  v8::Handle<v8::Object> obj = info.This();  //获取在SetValue里绑定到obj的tracker,然后通过tracker获取在  //SetValue里设置到tracker的accessor  CefRefPtr<CefV8Accessor> accessorPtr;  V8TrackObject\* tracker = V8TrackObject::Unwrap(obj);  if (tracker)  accessorPtr = tracker->GetAccessor();  //下面的代码可以看到最终将代码转发到SetValue里设置的Accessor的Set方法  if (accessorPtr.get()) {  CefRefPtr<CefV8Value> object = new CefV8ValueImpl(obj);  CefRefPtr<CefV8Value> cefValue = new CefV8ValueImpl(value);  CefString name, exception;  GetCefString(property, name);  accessorPtr->Set(name, object, cefValue, exception);  if (!exception.empty()) {  v8::ThrowException(v8::Exception::Error(GetV8String(exception)));  return;  }  }  } |

对于以字符串为键的字典来说，需要提供获取所有键的方法，以便于遍历字典：

|  |
| --- |
| bool CefV8ValueImpl::GetKeys(std::vector<CefString>& keys) {  CEF\_V8\_REQUIRE\_OBJECT\_RETURN(false);  v8::HandleScope handle\_scope(handle\_->isolate());  v8::Handle<v8::Value> value = handle\_->GetNewV8Handle(false);  v8::Handle<v8::Object> obj = value->ToObject();  //获取Object所有的属性名字，这是一个v8::Array  //TODO：为什么使用Local，而不是Handle？  v8::Local<v8::Array> arr\_keys = obj->GetPropertyNames();  uint32\_t len = arr\_keys->Length();  for (uint32\_t i = 0; i < len; ++i) {  v8::Local<v8::Value> value = arr\_keys->Get(v8::Integer::New(i));  CefString str;  GetCefString(value->ToString(), str);  keys.push\_back(str);  }  return true;  } |

当然，应该提供删除键值对的方法：

|  |
| --- |
| bool CefV8ValueImpl::DeleteValue(const CefString& key) {  CEF\_V8\_REQUIRE\_OBJECT\_RETURN(false);  v8::HandleScope handle\_scope(handle\_->isolate());  v8::Handle<v8::Value> value = handle\_->GetNewV8Handle(false);  v8::Handle<v8::Object> obj = value->ToObject();  v8::TryCatch try\_catch;  try\_catch.SetVerbose(true);  bool del = obj->Delete(GetV8String(key));  return (!HasCaught(try\_catch) && del);  }  bool CefV8ValueImpl::DeleteValue(int index) {  CEF\_V8\_REQUIRE\_OBJECT\_RETURN(false);  if (index < 0) {  NOTREACHED() << "invalid input parameter";  return false;  }  v8::HandleScope handle\_scope(handle\_->isolate());  v8::Handle<v8::Value> value = handle\_->GetNewV8Handle(false);  v8::Handle<v8::Object> obj = value->ToObject();  v8::TryCatch try\_catch;  try\_catch.SetVerbose(true);  bool del = obj->Delete(index);  return (!HasCaught(try\_catch) && del);  } |

如果CefV8Value是一个数组，则可以使用下面的方法获取数组长度：

|  |
| --- |
| int CefV8ValueImpl::GetArrayLength() {  CEF\_V8\_REQUIRE\_OBJECT\_RETURN(0);  v8::HandleScope handle\_scope(handle\_->isolate());  v8::Handle<v8::Value> value = handle\_->GetNewV8Handle(false);  if (!value->IsArray()) {  NOTREACHED() << "V8 value is not an array";  return 0;  }  v8::Handle<v8::Object> obj = value->ToObject();  v8::Local<v8::Array> arr = v8::Handle<v8::Array>::Cast(obj);  return arr->Length();  } |

当然，CefV8Value提供了静态方法用于创建数组，也就是这里的字典：

|  |
| --- |
| CefRefPtr<CefV8Value> CefV8Value::CreateArray(int length) {  CEF\_V8\_REQUIRE\_ISOLATE\_RETURN(NULL);  v8::HandleScope handle\_scope(v8::Isolate::GetCurrent());  v8::Local<v8::Context> context = v8::Context::GetCurrent();  if (context.IsEmpty()) {  NOTREACHED() << "not currently in a V8 context";  return NULL;  }  //创建V8TrackObject  // Create a tracker object that will cause the user data reference to be  // released when the V8 object is destroyed.  V8TrackObject\* tracker = new V8TrackObject;    //创建v8::Array  // Create the new V8 array.  v8::Local<v8::Array> arr = v8::Array::New(length);  // Attach the tracker object.  tracker->AttachTo(arr);//将arr绑定到tracker  //将arr和tracker初始化到CefV8Value  CefRefPtr<CefV8ValueImpl> impl = new CefV8ValueImpl();  impl->InitObject(arr, tracker);  return impl.get();  } |

## 异常

前面的代码在异常处理的地方都是用了一个函数HasCaught,该方法如下:

|  |
| --- |
| bool CefV8ValueImpl::HasCaught(v8::TryCatch& try\_catch) {  if (try\_catch.HasCaught()) {  last\_exception\_ = new CefV8ExceptionImpl(try\_catch.Message());  if (rethrow\_exceptions\_)  try\_catch.ReThrow();  return true;  } else {  if (last\_exception\_.get())  last\_exception\_ = NULL;  return false;  }  } |

此处将v8的异常信息转换成了CefExceptionImpl，CefExceptionImpl继承自CefException接口，实现如下，只是一个数据封装类：

|  |
| --- |
| class CefV8ExceptionImpl : public CefV8Exception {  public:  explicit CefV8ExceptionImpl(v8::Handle<v8::Message> message)  : line\_number\_(0),  start\_position\_(0),  end\_position\_(0),  start\_column\_(0),  end\_column\_(0) {  if (message.IsEmpty())  return;  GetCefString(message->Get(), message\_);  GetCefString(message->GetSourceLine(), source\_line\_);  if (!message->GetScriptResourceName().IsEmpty())  GetCefString(message->GetScriptResourceName()->ToString(), script\_);  line\_number\_ = message->GetLineNumber();  start\_position\_ = message->GetStartPosition();  end\_position\_ = message->GetEndPosition();  start\_column\_ = message->GetStartColumn();  end\_column\_ = message->GetEndColumn();  }  virtual CefString GetMessage() OVERRIDE { return message\_; }  virtual CefString GetSourceLine() OVERRIDE { return source\_line\_; }  virtual CefString GetScriptResourceName() OVERRIDE { return script\_; }  virtual int GetLineNumber() OVERRIDE { return line\_number\_; }  virtual int GetStartPosition() OVERRIDE { return start\_position\_; }  virtual int GetEndPosition() OVERRIDE { return end\_position\_; }  virtual int GetStartColumn() OVERRIDE { return start\_column\_; }  virtual int GetEndColumn() OVERRIDE { return end\_column\_; }  protected:  CefString message\_;  CefString source\_line\_;  CefString script\_;  int line\_number\_;  int start\_position\_;  int end\_position\_;  int start\_column\_;  int end\_column\_;  IMPLEMENT\_REFCOUNTING(CefV8ExceptionImpl);  }; |

## 内存控制

根据3.3.8节点内容，非v8托管内存需要手工通知v8的GC系统外部资源的内存变动，CefV8Value提供了下面两个方法：

|  |
| --- |
| int CefV8ValueImpl::GetExternallyAllocatedMemory() {  CEF\_V8\_REQUIRE\_OBJECT\_RETURN(0);  v8::HandleScope handle\_scope(handle\_->isolate());  v8::Handle<v8::Value> value = handle\_->GetNewV8Handle(false);  v8::Handle<v8::Object> obj = value->ToObject();  V8TrackObject\* tracker = V8TrackObject::Unwrap(obj);  if (tracker)  return tracker->GetExternallyAllocatedMemory();  return 0;  }  int CefV8ValueImpl::AdjustExternallyAllocatedMemory(int change\_in\_bytes) {  CEF\_V8\_REQUIRE\_OBJECT\_RETURN(0);  v8::HandleScope handle\_scope(handle\_->isolate());  v8::Handle<v8::Value> value = handle\_->GetNewV8Handle(false);  v8::Handle<v8::Object> obj = value->ToObject();  V8TrackObject\* tracker = V8TrackObject::Unwrap(obj);  if (tracker)  return tracker->AdjustExternallyAllocatedMemory(change\_in\_bytes);  return 0;  } |

基本上，CefV8Value将Object对象的内存调整放到了与Object所伴随的Tracker对象，上述两个方法都只是简单转发。V8TrackeObject（这是一个Cef的类，但是奇怪的是没有加Cef前缀，也许Cef的作者认为这个类应该由V8提供？）与CefV8Value紧密相连，我们单独在4.10里解析。

## Cef的对象跟踪:V8TrackObject

从4.2的代码里可以看到，当一个CefV8Value对象obj的类型标记为TYPE\_OBJECT的时候，CefV8Value::CreateObject内部会为每个obj同时创建一个V8TrackObject对象tracker。tracker的作用是绑定obj对象，并保存obj的其他辅助信息，比如当obj是一个Function时，tracker可以保存obj的CefV8Handle对象；当obj是一个Array时，tracker保存obj的Accessor对象；当obj是一个UserData时，tracker保存obj的userData对象。并且无论哪种具体的obj，都可以通过tracker->AdjustExternallyAllocateMemory通知v8的GC系统调整外部资源的内存信息。

V8TrackObject如此重要，在4.2剖析CefV8Value代码的过程中，我们已经了解了它的主要功能和用法，我们可以在V8TrackObject的源码里映照上述过程。我们在代码里直接注释关键代码的作用。

|  |
| --- |
| class V8TrackObject : public CefTrackNode {  public:  V8TrackObject()  : external\_memory\_(0) {  //构造函数里将V8TrackObject的大小通知v8的GC系统  v8::V8::AdjustAmountOfExternalAllocatedMemory(  static\_cast<int>(sizeof(V8TrackObject)));  }  ~V8TrackObject() {  //析构函数里通知v8的GC系统，减少V8TrackObject管理的外部内存字节  //包括V8TrackObject自身的大小和external\_memory  v8::V8::AdjustAmountOfExternalAllocatedMemory(  -static\_cast<int>(sizeof(V8TrackObject)) - external\_memory\_);  }  //返回外部内存大小  inline int GetExternallyAllocatedMemory() {  return external\_memory\_;  }  //增加外部内存大小  int AdjustExternallyAllocatedMemory(int change\_in\_bytes) {  //计算调整后V8TrackObject管理的外部内存大小  int new\_value = external\_memory\_ + change\_in\_bytes;  if (new\_value < 0) {  NOTREACHED() << "External memory usage cannot be less than 0 bytes";  change\_in\_bytes = -(external\_memory\_);  new\_value = 0;  }    //通知v8的GC调整外部内存大小  if (change\_in\_bytes != 0)  v8::V8::AdjustAmountOfExternalAllocatedMemory(change\_in\_bytes);  external\_memory\_ = new\_value;  return new\_value;  }    //设置存取器，绑定数组时用到  inline void SetAccessor(CefRefPtr<CefV8Accessor> accessor) {  accessor\_ = accessor;  }  //获取存取器，在数组的存取器回调函数里用到  inline CefRefPtr<CefV8Accessor> GetAccessor() {  return accessor\_;  }  //设置CefV8Handler，绑定函数时用到  inline void SetHandler(CefRefPtr<CefV8Handler> handler) {  handler\_ = handler;  }    //获取CefV8Handler，执行函数模板的回调函数时用到  inline CefRefPtr<CefV8Handler> GetHandler() {  return handler\_;  }    //设置UserData  inline void SetUserData(CefRefPtr<CefBase> user\_data) {  user\_data\_ = user\_data;  }    //获取UserData  inline CefRefPtr<CefBase> GetUserData() {  return user\_data\_;  }  //设置对象的隐藏对象  // Attach this track object to the specified V8 object.  void AttachTo(v8::Handle<v8::Object> object) {  object->SetHiddenValue(v8::String::New(kCefTrackObject),  v8::External::New(this));  }  //获取对象的隐藏对象  // Retrieve the track object for the specified V8 object.  static V8TrackObject\* Unwrap(v8::Handle<v8::Object> object) {  v8::Local<v8::Value> value =  object->GetHiddenValue(v8::String::New(kCefTrackObject));  if (!value.IsEmpty())  return static\_cast<V8TrackObject\*>(v8::External::Cast(\*value)->Value());  return NULL;  }  private:  CefRefPtr<CefV8Accessor> accessor\_; //存取器  CefRefPtr<CefV8Handler> handler\_; //函数对象  CefRefPtr<CefBase> user\_data\_; //用户定义类  int external\_memory\_;  }; |

## CefIsolateManager

4.2里我们看到TYPE\_OBJECT的CefV8Value使用Handle存储，而Handle继承CefHandleBase类，CefHandleBase负责管理Context、TaskRunner以及Isolate，再次看下其构造函数：

|  |
| --- |
| CefV8HandleBase::CefV8HandleBase(v8::Handle<v8::Context> context) {  CefV8IsolateManager\* manager = GetIsolateManager();  DCHECK(manager);  isolate\_ = manager->isolate();  task\_runner\_ = manager->task\_runner();  context\_state\_ = manager->GetContextState(context);  } |

本节我们重点关注CefV8IsolateManager，顾名思义这是一个管理v8::Isolate的类，根据v8的说明，独立的v8虚拟机需要各自的Isolate，我们看下代码：

|  |
| --- |
| // Manages memory and state information associated with a single Isolate.  class CefV8IsolateManager {  public:  //构造函数，初始化了下面几个成员：  //1、当前线程的Isolate  //2、TaskRunner  //3、上下文安全类型，这是一个枚举  //4、是否注册了消息监听器  //5、工作线程id  CefV8IsolateManager()  : isolate\_(v8::Isolate::GetCurrent()),  task\_runner\_(CefContentRendererClient::Get()->GetCurrentTaskRunner()),  context\_safety\_impl\_(IMPL\_HASH),  message\_listener\_registered\_(false),  worker\_id\_(0) {  DCHECK(isolate\_);  DCHECK(task\_runner\_.get());    //查找进程的参数信息，更改上下文类型  const CommandLine& command\_line = \*CommandLine::ForCurrentProcess();  if (command\_line.HasSwitch(switches::kContextSafetyImplementation)) {  std::string value = command\_line.GetSwitchValueASCII(  switches::kContextSafetyImplementation);  int mode;  if (base::StringToInt(value, &mode)) {  if (mode < 0)  context\_safety\_impl\_ = IMPL\_DISABLED;  else if (mode == 1)  context\_safety\_impl\_ = IMPL\_VALUE;  }  }  }  //析构函数  ~CefV8IsolateManager() {  DCHECK\_EQ(isolate\_, v8::Isolate::GetCurrent());  DCHECK(context\_map\_.empty());  }  //获取context对应的CefV8ContextState  //可见一个context在Cef里被映射到一个ContextState  //TODO：为什么要这么做？  scoped\_refptr<CefV8ContextState> GetContextState(  v8::Handle<v8::Context> context) {  DCHECK\_EQ(isolate\_, v8::Isolate::GetCurrent());  DCHECK(context.IsEmpty() || isolate\_ == context->GetIsolate());    //如果禁用IMPL，直接返回空的CefV8ContextState  if (context\_safety\_impl\_ == IMPL\_DISABLED)  return scoped\_refptr<CefV8ContextState>();  //IMPL可用，但context为空  if (context.IsEmpty()) {  if (v8::Context::InContext())  context = v8::Context::GetCurrent();//当前上下文  else  return scoped\_refptr<CefV8ContextState>();//空上下文  }  if (context\_safety\_impl\_ == IMPL\_HASH) {  //IMPL\_HASH类型，直接查表返回  int hash = context->Global()->GetIdentityHash();  ContextMap::const\_iterator it = context\_map\_.find(hash);  if (it != context\_map\_.end())  return it->second;  scoped\_refptr<CefV8ContextState> state = new CefV8ContextState();  context\_map\_.insert(std::make\_pair(hash, state));  return state;  } else {  //IMPL\_VALUE类型，返回全局CefContextState对象  v8::Handle<v8::String> key = v8::String::New(kCefContextState);  //先查找是否已创建，如果有则返回  //可见全局CefV8ContextState也类似V8TrackObject被保存在  //context->Global()对象的隐藏External对象里  v8::Handle<v8::Object> object = context->Global();  v8::Handle<v8::Value> value = object->GetHiddenValue(key);  if (!value.IsEmpty()) {  return static\_cast<CefV8ContextState\*>(  v8::External::Cast(\*value)->Value());  }  //首次获取，创建并保存到context->Global()的隐藏External对象里  scoped\_refptr<CefV8ContextState> state = new CefV8ContextState();  object->SetHiddenValue(key, v8::External::New(state.get()));  //增加引用计数，在ReleaseContext里减少引用计数  // Reference will be released in ReleaseContext.  state->AddRef();  return state;  }  }  //释放Context  void ReleaseContext(v8::Handle<v8::Context> context) {  DCHECK\_EQ(isolate\_, v8::Isolate::GetCurrent());  //IMPL\_DISABLED类型，不需要释放  if (context\_safety\_impl\_ == IMPL\_DISABLED)  return;  if (context\_safety\_impl\_ == IMPL\_HASH) {  //IMPL\_HASH类型，从字典里删除即可  int hash = context->Global()->GetIdentityHash();  ContextMap::iterator it = context\_map\_.find(hash);  if (it != context\_map\_.end()) {  it->second->Detach();//注意此处，调用CefContextState解除所管理的跟踪对象  context\_map\_.erase(it);  }  } else {  //IMPL\_VALUE类型，删除全局对象  v8::Handle<v8::String> key = v8::String::New(kCefContextState);  v8::Handle<v8::Object> object = context->Global();  v8::Handle<v8::Value> value = object->GetHiddenValue(key);  if (value.IsEmpty())  return;  scoped\_refptr<CefV8ContextState> state =  static\_cast<CefV8ContextState\*>(v8::External::Cast(\*value)->Value());  state->Detach();//删除所管理的跟踪对象  object->DeleteHiddenValue(key);  //减少引用计数  // Match the AddRef in GetContextState.  state->Release();  }  }  //添加全局跟踪对象  void AddGlobalTrackObject(CefTrackNode\* object) {  DCHECK\_EQ(isolate\_, v8::Isolate::GetCurrent());  global\_manager\_.Add(object);  }  //删除全局跟踪对象  void DeleteGlobalTrackObject(CefTrackNode\* object) {  DCHECK\_EQ(isolate\_, v8::Isolate::GetCurrent());  global\_manager\_.Delete(object);  }  //设置未捕获的异常栈大小  void SetUncaughtExceptionStackSize(int stack\_size) {  if (stack\_size <= 0)  return;  //如果未注册消息监听回调函数，则注册  //用于处理未捕获的异常信息  if (!message\_listener\_registered\_) {  v8::V8::AddMessageListener(&MessageListenerCallbackImpl);  message\_listener\_registered\_ = true;  }  v8::V8::SetCaptureStackTraceForUncaughtExceptions(true,  stack\_size, v8::StackTrace::kDetailed);  }  //设置工作id和url  void SetWorkerAttributes(int worker\_id, const GURL& worker\_url) {  worker\_id\_ = worker\_id;  worker\_url\_ = worker\_url;  }  //获取v8::Isolate  v8::Isolate\* isolate() const { return isolate\_; }  scoped\_refptr<base::SequencedTaskRunner> task\_runner() const {  return task\_runner\_;  }  //获取工作id  int worker\_id() const {  return worker\_id\_;  }  //获取工作url  const GURL& worker\_url() const {  return worker\_url\_;  }  private:  v8::Isolate\* isolate\_;  scoped\_refptr<base::SequencedTaskRunner> task\_runner\_;  enum ContextSafetyImpl {  IMPL\_DISABLED,  IMPL\_HASH,  IMPL\_VALUE,  };  ContextSafetyImpl context\_safety\_impl\_;  // Used with IMPL\_HASH.  typedef std::map<int, scoped\_refptr<CefV8ContextState> > ContextMap;  ContextMap context\_map\_;  // Used for globally tracked objects that are not associated with a particular  // context.  CefTrackManager global\_manager\_;  // True if the message listener has been registered.  bool message\_listener\_registered\_;  // Attributes associated with WebWorker threads.  int worker\_id\_;  GURL worker\_url\_;  }; |

### Isolate

首先CefIsolateManager持有一份Isolate，供所有需要的地方共用。

### CefContextState

其次CefIsolateManager拥有GetContextState和ReleaseContextState两个方法，根据conxt\_safty\_impl的枚举值，支持禁用、哈希映射以及全局共享的方式使用上下文。要理解这点我们需要进一步看下CefContextState这个类：

|  |
| --- |
| // Used to detach handles when the associated context is released.  class CefV8ContextState : public base::RefCounted<CefV8ContextState> {  public:  CefV8ContextState() : valid\_(true) {}  virtual ~CefV8ContextState() {}  bool IsValid() { return valid\_; }    //删除所有的跟踪对象  void Detach() {  DCHECK(valid\_);  valid\_ = false;  track\_manager\_.DeleteAll();  }  //添加跟踪对象  void AddTrackObject(CefTrackNode\* object) {  DCHECK(valid\_);  track\_manager\_.Add(object);  }  //删除跟踪对象  void DeleteTrackObject(CefTrackNode\* object) {  DCHECK(valid\_);  track\_manager\_.Delete(object);  }  private:  bool valid\_;  CefTrackManager track\_manager\_;  }; |

可以看到，CefContextState可以添加和删除跟踪对象，并且提供了一个Datch方法批量删除所添加到跟踪对象，在CefIsolateManager::ReleaseContext里用到这个。理解了这点，也就能理解CefIsolateManager的AddGlobalTrackObject和DeleteGlobalTrackObject方法。都是用来管理TrackObject的，只是一个是Context管理的，一个全局的。TrackObject的管理类则是CefTrackManager类，这是一个Cef通用跟踪节点管理类，管理CefTrackNode对象，而我们关心的V8TrackObject继承自CefTrackNode。

在4.15和4.16里，我们会看到CefIsolateManager::AddGlobalTrackObject被用于持有扩展方法的名字、JavaScript代码以及JavaScript回调委托（CefV8Handler）。而CefV8ValueImpl::Handle的构造函数里将上下文赋值给父类的构造函数，也就是CefHanldeBase的构造函数，后者通过CefIsolateManager::GetContextState返回一个CefContextState实例，那么这个对象被构造出来的作用是什么呢？很简单，谁持有它谁需要它！我们跟踪CefHandleBase类，没看到直接使用它的地方，进而我们跟踪CefV8ValueImpl::Handle类，在它的析构函数里看到了线索：

|  |
| --- |
| CefV8ValueImpl::Handle::~Handle() {  // Persist the handle (call MakeWeak) if:  // A. The handle has been passed into a V8 function or used as a return value  // from a V8 callback, and  // B. The associated context, if any, is still valid.  if (should\_persist\_ && (!context\_state\_.get() || context\_state\_->IsValid())) {  handle\_.MakeWeak(  (tracker\_ ? new CefV8MakeWeakParam(context\_state\_, tracker\_) : NULL),  TrackDestructor);  } else {  handle\_.Reset();  handle\_.Clear();  if (tracker\_)  delete tracker\_;  }  tracker\_ = NULL;  } |

可见，CefV8ValueImpl::Handle内部的should\_persist布尔变量指示了在handle\_被析构时是否持久化所持有的对象。

从代码可以看出，在should\_persist为true并且context\_state\_有效时，调用handle\_.MakeWeak。否则，直接调用handle\_.Reset()和handle\_.Clear()重置并清空对象。

我们知道handle\_本身是一个v8::Persistent<T>对象，MakeWeak则是它的一个方法：

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| |  |  | | --- | --- | | void | [**MakeWeak**](http://izs.me/v8-docs/classv8_1_1Persistent.html#ab04609812113450bece2640ad0b27658) (void \*parameters, [**WeakReferenceCallback**](http://izs.me/v8-docs/namespacev8.html#a9065f8994925d59dc82ae88667d93904) callback) | |  | Make the reference to this object weak. | |

用来将其设置为弱引用，这样在v8的GC回收时，如果这个对象没有在其他地方被引用，则可以回收对象内存，从而避免内存泄漏。这里可以传入一个额外的parameter指针，以及一个回调函数，额外的parameter指针是个void\*，所以可以传入任意的用户数据，而回调函数显然是在v8的GC回收弱引用对象时调用。WeakReferenceCallback如下：

|  |
| --- |
| **typedef void(\*** [**v8::WeakReferenceCallback**](http://izs.me/v8-docs/namespacev8.html#a9065f8994925d59dc82ae88667d93904)**)(**[**Persistent**](http://izs.me/v8-docs/classv8_1_1Persistent.html)**<** [**Value**](http://izs.me/v8-docs/classv8_1_1Value.html) **> object, void \*parameter)** |

可见，回调函数里会将v8::Persistent<T>对象以及额外的用户数据parameter对象指针传入，将真正的内存释放动作交给用户处理。

我们分别看下CefV8MakeWeakParam类以及TrackObjectDestructor函数：

|  |
| --- |
| // Manages the life span of a CefTrackNode associated with a persistent Object  // or Function.  class CefV8MakeWeakParam {  public:  CefV8MakeWeakParam(scoped\_refptr<CefV8ContextState> context\_state,  CefTrackNode\* object)  : context\_state\_(context\_state),  object\_(object) {  DCHECK(object\_);  v8::V8::AdjustAmountOfExternalAllocatedMemory(  static\_cast<int>(sizeof(CefV8MakeWeakParam)));  if (context\_state\_.get()) {  // |object\_| will be deleted when:  // A. The associated context is released, or  // B. TrackDestructor is called for the weak handle.  DCHECK(context\_state\_->IsValid());  context\_state\_->AddTrackObject(object\_);  } else {  // |object\_| will be deleted when:  // A. The process shuts down, or  // B. TrackDestructor is called for the weak handle.  GetIsolateManager()->AddGlobalTrackObject(object\_);  }  }  ~CefV8MakeWeakParam() {  if (context\_state\_.get()) {  // If the associated context is still valid then delete |object\_|.  // Otherwise, |object\_| will already have been deleted.  if (context\_state\_->IsValid())  context\_state\_->DeleteTrackObject(object\_);  } else {  GetIsolateManager()->DeleteGlobalTrackObject(object\_);  }  v8::V8::AdjustAmountOfExternalAllocatedMemory(  -static\_cast<int>(sizeof(CefV8MakeWeakParam)));  }  private:  scoped\_refptr<CefV8ContextState> context\_state\_;  CefTrackNode\* object\_;  }; |

这个CefV8MakeWeakParam类的构造函数里，tracker对象要么被CefIsolateManger::AddGlobalTrackObject管理，要么被CefContextState::AddTrackObject管理。而析构函数里被删除，由于这个对象会被v8管理，所以也需要在构造函数和析构函数里调整v8外部内存的大小。

TrackDestructor则用来在v8的垃圾回收时真正释放对象的内存：

|  |
| --- |
| // Callback for weak persistent reference destruction.  void TrackDestructor(v8::Isolate\* isolate,  v8::Persistent<v8::Value>\* object,  CefV8MakeWeakParam\* parameter) {  if (parameter)  delete parameter;  object->Reset();  object->Clear();  } |

可以看到，在弱引用回调函数里，先析构CefV8MakeWeakParam对象，再将handle本身重置并清空。从而正确的释放内存。

经过这一轮分析，可知CefV8ValueImpl::Hanlde，CefV8HanldeBase，CefIsolateManager，CefContextState，CefMakeWeakParam，TrackDestructor他们精密配合，完成了对CefV8Value类型为TYPE\_OBJECT时的句柄管理和对象生命周期管理。

可见,一切的复杂性都是由于v8的句柄管理之麻烦所导致的。v8::Object可以是Array、Function，UserData，而Array需要Accessor（进而需要AccessorGetterCallbackImpl，AccessorSetterCallbackImpl），Function需要函数模板及其相关的回调（进而需要FunctionTemplateCallbackImpl），UserData需要存取用户定义数据。这些数据只能以v8::Object的隐藏外部数据形式存储，在CEF里统一通过v8TrackObject存取这几种数据。又由于v8的句柄有局部句柄和持久句柄，同时需要Context和Isoalte等信息，所以一个TYPE\_OBJECT的CefVa8Value需要在内部用CefV8ValueImpl::Hanlde将object,tracker,context都装进去。CefV8ValueImpl::Hanlde内部默认以v8::Persistent<T>持有object，在外部使用则需要通过GetNewV8Hanlde(bool should\_persist)返回v8::Hanlde局部句柄。 外部使用时可以指定一个额外的参数should\_persist，如果用户指定了shold\_persist为true，则在CefV8ValueImpl析构函数被调用时，并不直接释放内存，而是调用v8::Persist<T>的成员函数MakeWeak将handle转成弱引用，同时将保存object辅助数据的tracker以CefV8MakeWeakParam封装，并传给MakeWeak。从而，当外部代码不再持有handle时，v8的GC将调用MakeWeak设置的回调函数释放内存，这个回调函数在CEF里是TrackObjectDestructor这个函数，在这里将之前封装的CefV8MakeWeakParam释放，同时真正释放handle。

### SetUncaughtExceptionStackSize

CefIsolateManager通过SetUncaughtExceptionStackSize方法设置v8未捕获异常的监听回调函数，这个回调函数是MessageListenerCallbackImpl：

|  |
| --- |
| void MessageListenerCallbackImpl(v8::Handle<v8::Message> message,  v8::Handle<v8::Value> data) {  //获取CefContentClient对象，进而获取CefApp对象  CefRefPtr<CefApp> application = CefContentClient::Get()->application();  if (!application.get())  return;  //获取CefRenderProcessHandler对象  CefRefPtr<CefRenderProcessHandler> handler =  application->GetRenderProcessHandler();  if (!handler.get())  return;  //获取当前上下文  CefRefPtr<CefV8Context> context = CefV8Context::GetCurrentContext();  //获取消息的堆栈信息  v8::Handle<v8::StackTrace> v8Stack = message->GetStackTrace();  DCHECK(!v8Stack.IsEmpty());  //将v8::StackTrace转成CefStackTraceImpl对象  CefRefPtr<CefV8StackTrace> stackTrace = new CefV8StackTraceImpl(v8Stack);    //将v8::Message转成CefExceptionImpl对象  CefRefPtr<CefV8Exception> exception = new CefV8ExceptionImpl(message);  if (CEF\_CURRENTLY\_ON\_RT()) {  //如果当前线程是Render进程的主线程则调用Render进程的全局异常处理函数  handler->OnUncaughtException(context->GetBrowser(), context->GetFrame(),  context, exception, stackTrace);  }  } |

从而，我们接触到一个新类：CefStackTrace，和CefException类一样，这也是一个数据封装类：

|  |
| --- |
| class CefV8StackFrameImpl : public CefV8StackFrame {  public:  explicit CefV8StackFrameImpl(v8::Handle<v8::StackFrame> handle);  virtual ~CefV8StackFrameImpl();  virtual bool IsValid() OVERRIDE;  virtual CefString GetScriptName() OVERRIDE;  virtual CefString GetScriptNameOrSourceURL() OVERRIDE;  virtual CefString GetFunctionName() OVERRIDE;  virtual int GetLineNumber() OVERRIDE;  virtual int GetColumn() OVERRIDE;  virtual bool IsEval() OVERRIDE;  virtual bool IsConstructor() OVERRIDE;  protected:  CefString script\_name\_;  CefString script\_name\_or\_source\_url\_;  CefString function\_name\_;  int line\_number\_;  int column\_;  bool is\_eval\_;  bool is\_constructor\_;  IMPLEMENT\_REFCOUNTING(CefV8StackFrameImpl);  DISALLOW\_COPY\_AND\_ASSIGN(CefV8StackFrameImpl);  }; |

## CefV8Context

在4.6里剖析函数的代码里，ExecuteFunction转调用ExecuteFunctionWithContext，而后者代码里用到了CefV8Context类，这个类在ExecuteFunctionWithContext里貌似没什么作用，但实际上根据v8的规范，执行JavaScript函数必须在Context之内，所以调用ExecuteFunction的前我们必须进入Context，执行完毕后必须退出Context。

这个功能是由CefV8Context提供的，最重要的成员有Enter、Exist、Eval等。我们将只关注这三个方法。

|  |
| --- |
| class CefV8ContextImpl : public CefV8Context {  public:  explicit CefV8ContextImpl(v8::Handle<v8::Context> context);  virtual ~CefV8ContextImpl();  virtual CefRefPtr<CefTaskRunner> GetTaskRunner() OVERRIDE;  virtual bool IsValid() OVERRIDE;  virtual CefRefPtr<CefBrowser> GetBrowser() OVERRIDE;  virtual CefRefPtr<CefFrame> GetFrame() OVERRIDE;  virtual CefRefPtr<CefV8Value> GetGlobal() OVERRIDE;  virtual bool Enter() OVERRIDE;  virtual bool Exit() OVERRIDE;  virtual bool IsSame(CefRefPtr<CefV8Context> that) OVERRIDE;  virtual bool Eval(const CefString& code,  CefRefPtr<CefV8Value>& retval,  CefRefPtr<CefV8Exception>& exception) OVERRIDE;  v8::Handle<v8::Context> GetV8Context();  blink::WebFrame\* GetWebFrame();  protected:  typedef CefV8Handle<v8::Context> Handle;  scoped\_refptr<Handle> handle\_;  #ifndef NDEBUG  // Used in debug builds to catch missing Exits in destructor.  int enter\_count\_;  #endif  IMPLEMENT\_REFCOUNTING(CefV8ContextImpl);  DISALLOW\_COPY\_AND\_ASSIGN(CefV8ContextImpl);  }; |

首先是构造函数，将v8::Context使用Handle类管理，这个Handle是一个typedef，真正的类型是CefV8Handle<T>，CefV8Handle<T>与CefV8Value::Handle的实现一样，内部用持久句柄引用对象，提供获取局部句柄和持久句柄的接口。

|  |
| --- |
| CefV8ContextImpl::CefV8ContextImpl(v8::Handle<v8::Context> context)  : handle\_(new Handle(context, context))  #ifndef NDEBUG  , enter\_count\_(0)  #endif  { // NOLINT(whitespace/braces)  } |

其次，Enter和Exit方法，封装了执行JavaScript代码所需的进入和退出上下文的代码，这也是遵守v8执行JavaScript代码的规范。

|  |
| --- |
| bool CefV8ContextImpl::Enter() {  CEF\_V8\_REQUIRE\_VALID\_HANDLE\_RETURN(false);    //声明句柄范围，自动管理局部句柄  v8::HandleScope handle\_scope(handle\_->isolate());  //添加Context递归层，TODO：进一步了解  WebCore::V8PerIsolateData::current()->incrementRecursionLevel();  //获取v8::Context的局部句柄，然后调用v8::Context::Exter  handle\_->GetNewV8Handle()->Enter();  #ifndef NDEBUG  ++enter\_count\_;  #endif  return true;  }  bool CefV8ContextImpl::Exit() {  CEF\_V8\_REQUIRE\_VALID\_HANDLE\_RETURN(false);  //声明句柄范围，自动管理局部句柄  v8::HandleScope handle\_scope(handle\_->isolate());  DLOG\_ASSERT(enter\_count\_ > 0);  //获取v8::Context的局部句柄，然后调用v8::Context::Exit  handle\_->GetNewV8Handle()->Exit();  //减少Context递归层  WebCore::V8PerIsolateData::current()->decrementRecursionLevel();  #ifndef NDEBUG  --enter\_count\_;  #endif  return true;  } |

在Enter和Exit之间，我们可以调用CefV8Value::ExecuteFunction或者调用CefV8Context::Eval方法解析JavaScript代码。我们进入Eval方法一窥奥妙，不过在此之前我们介绍下另一个方法：GetV8Context:

|  |
| --- |
| v8::Handle<v8::Context> CefV8ContextImpl::GetV8Context() {  return handle\_->GetNewV8Handle();  } |

GetV8Context只是对handle\_->GetNewV8Handle()的封装。

|  |
| --- |
| bool CefV8ContextImpl::Eval(const CefString& code,  CefRefPtr<CefV8Value>& retval,  CefRefPtr<CefV8Exception>& exception) {  CEF\_V8\_REQUIRE\_VALID\_HANDLE\_RETURN(false);  if (code.empty()) {  NOTREACHED() << "invalid input parameter";  return false;  }  //声明句柄范围，自动管理局部句柄  v8::HandleScope handle\_scope(handle\_->isolate());  //获取v8::Context  v8::Local<v8::Context> context = GetV8Context();  //进入上下文范围，并获取上下文的全局对象  v8::Context::Scope context\_scope(context);  v8::Local<v8::Object> obj = context->Global();  //从上下文里获取全局辅助函数”eval”，我们之前在v8的介绍里提到过每个Context会预先  //准备一堆全局对象和辅助函数，在这行代码里得到了验证。  // Retrieve the eval function.  v8::Local<v8::Value> val = obj->Get(v8::String::New("eval"));  if (val.IsEmpty() || !val->IsFunction())  return false;  //将eval对象转成v8::Function  v8::Local<v8::Function> func = v8::Local<v8::Function>::Cast(val);  v8::Handle<v8::Value> code\_val = GetV8String(code);  //添加异常处理  v8::TryCatch try\_catch;  try\_catch.SetVerbose(true);  retval = NULL;  exception = NULL;    //调用4.1.1.4节介绍过的CallV8Function执行eval函数，并返回结果或异常信息  v8::Local<v8::Value> func\_rv =  CallV8Function(context, func, obj, 1, &code\_val, handle\_->isolate());  if (try\_catch.HasCaught()) {  exception = new CefV8ExceptionImpl(try\_catch.Message());  return false;  } else if (!func\_rv.IsEmpty()) {  retval = new CefV8ValueImpl(func\_rv);  }  return true;  } |

## CefV8Handle

上一节提到过CefV8Context内部使用了CefV8Handle类与CefV8Value::Handle功能一样，但实际上为什么需要两个不同的类呢？很简单，CefV8Value::Handle是一个只针对v8::Value的Handle类，而CefV8Context是一个模板类，仅此而已。

|  |
| --- |
| // Template for V8 Handle types. This class is used to ensure that V8 objects  // are only released on the render thread.  template <typename v8class>  class CefV8Handle : public CefV8HandleBase {  public:  typedef v8::Handle<v8class> handleType;  typedef v8::Persistent<v8class> persistentType;  CefV8Handle(v8::Handle<v8::Context> context, handleType v)  : CefV8HandleBase(context),  handle\_(isolate(), v) {  }  virtual ~CefV8Handle() {  handle\_.Reset();  handle\_.Clear();  }  handleType GetNewV8Handle() {  DCHECK(IsValid());  return handleType::New(isolate(), handle\_);  }  persistentType& GetPersistentV8Handle() {  return handle\_;  }  protected:  persistentType handle\_;  DISALLOW\_COPY\_AND\_ASSIGN(CefV8Handle);  }; |

## CefV8Hanlder

有了前面的一系列介绍，我们终于可以引入CefV8Handler这个类的介绍了。首先，这个是CefV8Hanlder，不是CefV8Handle，少一个r都不行！

CefV8Handler是一个纯接口类，只有一个方法，你可以继承它，并提供相应的实现，在随后介绍的注册v8扩展方法时会使用到它。

|  |
| --- |
| ///  // Interface that should be implemented to handle V8 function calls. The methods  // of this class will be called on the thread associated with the V8 function.  ///  /\*--cef(source=client)--\*/  class CefV8Handler : public virtual CefBase {  public:  ///  // Handle execution of the function identified by |name|. |object| is the  // receiver ('this' object) of the function. |arguments| is the list of  // arguments passed to the function. If execution succeeds set |retval| to the  // function return value. If execution fails set |exception| to the exception  // that will be thrown. Return true if execution was handled.  ///  /\*--cef()--\*/  virtual bool Execute(const CefString& name,  CefRefPtr<CefV8Value> object,  const CefV8ValueList& arguments,  CefRefPtr<CefV8Value>& retval,  CefString& exception) =0;  }; |

## ExtensionWrapper

好了，在介绍完CefV8Hanlder类之后，我们将焦点集中在ExtensionWrappper类，这是CEF完成注册v8扩展的最后一环。按我们之前的习惯，我们先将这个类的源码过一遍。

|  |
| --- |
| //继承自v8::Extension,注册v8扩展当然得遵守v8的规矩，不是么？  class ExtensionWrapper : public v8::Extension {  public:  //构造函数希望提供：  //1、待注册的扩展名字  //2、注册扩展所需的JavaScript代码，我们会在后面告诉你需要怎样的代码才可以  //3、CefV8Handler，用来处理JavaScript调用的委托  ExtensionWrapper(const char\* extension\_name,  const char\* javascript\_code,  CefV8Handler\* handler)  : v8::Extension(extension\_name, javascript\_code), handler\_(handler) {  if (handler) {  //看到了没？将handler添加到全局CefIsolateManager管理  //AddGlobalTrackObject派上用场了。  // The reference will be released when the process exits.  V8TrackObject\* object = new V8TrackObject;  object->SetHandler(handler);  GetIsolateManager()->AddGlobalTrackObject(object);  }  }  //获取本地函数，覆写基类的方法总是必须的  virtual v8::Handle<v8::FunctionTemplate> GetNativeFunction(  v8::Handle<v8::String> name) {  //如果JavaScript调用委托不存在，应该返回空函数模板  if (!handler\_)  return v8::Handle<v8::FunctionTemplate>();    //否则，将4.6节介绍的FunctionCallbackImpl以及handler组合在一起  //创建一个新的函数模板，并返回。  //由此我们猜测v8在调用该函数模板后最后会调用FunctionCallbackImpl，  //而后者显然在内部将调用最后转发给了handler的Execute  //所以，JavaScript调用最终转发到了你手中的Handler的Execure方法  //在那里，你可以执行本地代码调用，然后返回适当的值。  //在那里，你当然可以通过CefV8Value::ExecuteFunction调用JavaScript的函数。  return v8::FunctionTemplate::New(FunctionCallbackImpl,  v8::External::New(handler\_));  }  private:  CefV8Handler\* handler\_;  }; |

传入的javascript\_code被设置给父类v8::Extension，我们在第2章看到过这个类，我们回顾下：

|  |
| --- |
| class V8EXPORT Extension { // NOLINT  public:  // Note that the strings passed into this constructor must live as long  // as the Extension itself.  Extension(const char\* name,  const char\* source = 0,  int dep\_count = 0,  const char\*\* deps = 0,  int source\_length = -1);  virtual ~Extension() { }  virtual v8::Handle<v8::FunctionTemplate>  GetNativeFunction(v8::Handle<v8::String> name) {  return v8::Handle<v8::FunctionTemplate>();  }  const char\* name() const { return name\_; }  size\_t source\_length() const { return source\_length\_; }  const String::ExternalAsciiStringResource\* source() const {  return &source\_; }  int dependency\_count() { return dep\_count\_; }  const char\*\* dependencies() { return deps\_; }  void set\_auto\_enable(bool value) { auto\_enable\_ = value; }  bool auto\_enable() { return auto\_enable\_; }  private:  const char\* name\_;  size\_t source\_length\_; // expected to initialize before source\_  ExternalAsciiStringResourceImpl source\_;  int dep\_count\_;  const char\*\* deps\_;  bool auto\_enable\_;  // Disallow copying and assigning.  Extension(const Extension&);  void operator=(const Extension&);  }; |

可见，父类只是简单讲代码保存了起来。我们查看

## 注册v8扩展

注册v8扩展的代码很简单：

|  |
| --- |
| bool CefRegisterExtension(const CefString& extension\_name,  const CefString& javascript\_code,  CefRefPtr<CefV8Handler> handler) {  // Verify that this method was called on the correct thread.  CEF\_REQUIRE\_RT\_RETURN(false);  //将扩展名字添加到全局跟踪对象  V8TrackString\* name = new V8TrackString(extension\_name);  GetIsolateManager()->AddGlobalTrackObject(name);  //将注册的JavaScript代码添加到全局跟踪对象  V8TrackString\* code = new V8TrackString(javascript\_code);  GetIsolateManager()->AddGlobalTrackObject(code);  //将扩展名字，JavaScript代码，以及JavaScript调用委托传入ExtensionWrapper  //构造一个v8::Extension的子类实例  ExtensionWrapper\* wrapper = new ExtensionWrapper(name->GetString(),  code->GetString(), handler.get());  //调用RenderThread::RegisterExtension  //正如第1章所描述的，它最终会调用v8::RegisterExtension  content::RenderThread::Get()->RegisterExtension(wrapper);  return true;  } |

CEF在这个方法接口的注释里有详细描述了注册扩展的JavaScript代码的作用以及示例，通过这个注释可以大致了解javascript\_code的作用和去处。

|  |
| --- |
| ///  // Register a new V8 extension with the specified JavaScript extension code and  // handler. Functions implemented by the handler are prototyped using the  // keyword 'native'. The calling of a native function is restricted to the scope  // in which the prototype of the native function is defined. This function may  // only be called on the render process main thread.  //  // Example JavaScript extension code:  // <pre>  // // create the 'example' global object if it doesn't already exist.  // if (!example)  // example = {};  // // create the 'example.test' global object if it doesn't already exist.  // if (!example.test)  // example.test = {};  // (function() {  // // Define the function 'example.test.myfunction'.  // example.test.myfunction = function() {  // // Call CefV8Handler::Execute() with the function name 'MyFunction'  // // and no arguments.  // native function MyFunction();  // return MyFunction();  // };  // // Define the getter function for parameter 'example.test.myparam'.  // example.test.\_\_defineGetter\_\_('myparam', function() {  // // Call CefV8Handler::Execute() with the function name 'GetMyParam'  // // and no arguments.  // native function GetMyParam();  // return GetMyParam();  // });  // // Define the setter function for parameter 'example.test.myparam'.  // example.test.\_\_defineSetter\_\_('myparam', function(b) {  // // Call CefV8Handler::Execute() with the function name 'SetMyParam'  // // and a single argument.  // native function SetMyParam();  // if(b) SetMyParam(b);  // });  //  // // Extension definitions can also contain normal JavaScript variables  // // and functions.  // var myint = 0;  // example.test.increment = function() {  // myint += 1;  // return myint;  // };  // })();  // </pre>  // Example usage in the page:  // <pre>  // // Call the function.  // example.test.myfunction();  // // Set the parameter.  // example.test.myparam = value;  // // Get the parameter.  // value = example.test.myparam;  // // Call another function.  // example.test.increment();  // </pre>  ///  /\*--cef(optional\_param=handler)--\*/  bool CefRegisterExtension(const CefString& extension\_name,  const CefString& javascript\_code,  CefRefPtr<CefV8Handler> handler); |

如果想要进一步了解v8::RegisterExtension背后的故事，则需要深入v8引擎，这将是另外一个故事，我们有时间再探究。