Chromium/content v8::Extension

fanfeilong@gmail.com

@幻灰龙

# 目录


1 注册 v8::Extension ..... 2
2 注册 v8::Extension 的内部流程 ..... 2
3 理解 v8 的设计理念和重要概念 ..... 6
3.1 v8 的设计文档 ..... 6
3.1.1 属性的快速读取 ..... 6
3.1.2 动态代码生成 ..... 7
3.1.3 高效垃圾回收。 ..... 8
3.2 v8 的 hello world。 ..... 8
3.3 v8 的重要概念。 ..... 9
3.3.1 句柄（Handles）和垃圾回收（Garbage Collection） ..... 10
3.3.2 上下文(Context) ..... 12
3.3.3 模板(Templates) ..... 12
3.3.4 存取器（Accessor） ..... 13
3.3.5 拦截器（Interceptor） ..... 15
3.3.6 安全（Security） ..... 16
3.3.7 继承（Inheritance） ..... 17
3.3.8 Isolate，Locker 以及 GC 设置 ..... 17
3.3.9 v8 的类型体系 ..... 18
3.4 v8 API 手册 ..... 19
4 CEF 的 v8::Extension 扩展机制 ..... 19
4.1 CEF 类型:CefV8Value ..... 19
4.2 CefValueImpl 的存储 ..... 21
4.3 基本内置类型 ..... 24
4.4 Object 和 UserData ..... 25
4.5 InitFromV8Value 和 GetV8Value ..... 27
4.6 函数 ..... 29
4.7 字典 ..... 36
4.8 异常 ..... 45
4.9 内存控制 ..... 46
4.10 Cef 的对象跟踪:V8TrackObject ..... 47
4.11 CefIsolateManager ..... 50
4.11.1 Isolate ..... 55
4.11.2 CefContextState ..... 55
4.11.3 SetUncaughtExceptionStackSize ..... 59
4.12 CefV8Context ..... 60
4.13 CefV8Handle ..... 64
4.14 CefV8Hanlder ..... 65
4.15 ExtensionWrapper ..... 65
4.16 注册 v8 扩展 ..... 68

# 1 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;
}
```

# 2 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

# 3 理解 v8 的设计理念和重要概念

## 3.1 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

### 3.1.1 属性的快速读取

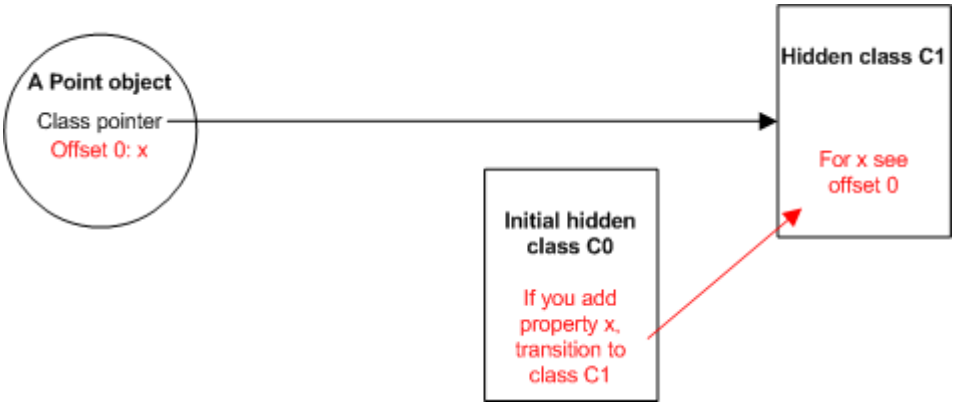
JavaScript 的对象是允许动态增减属性的，文档上说大部分其他 JavaScript 引擎都是是用字典来实现属性的动态增减，然而这样的话相比于 Java 和 SmllTalk 这样的静态类型语言来说就减低了访问效率。具体来说静态类型在编译时就确定了属性的偏移位置，所以访问只是直接做指针偏移，而使用字典的设计则每次访问都需要做一次查询，也就是间接寻址。为了达到静态类型的属性访问效率，同时保留属性的动态性，v8 采用了基于原型编程语言设计。

假设下面是一段 JavaScript 函数，用于动态创建一个对象：

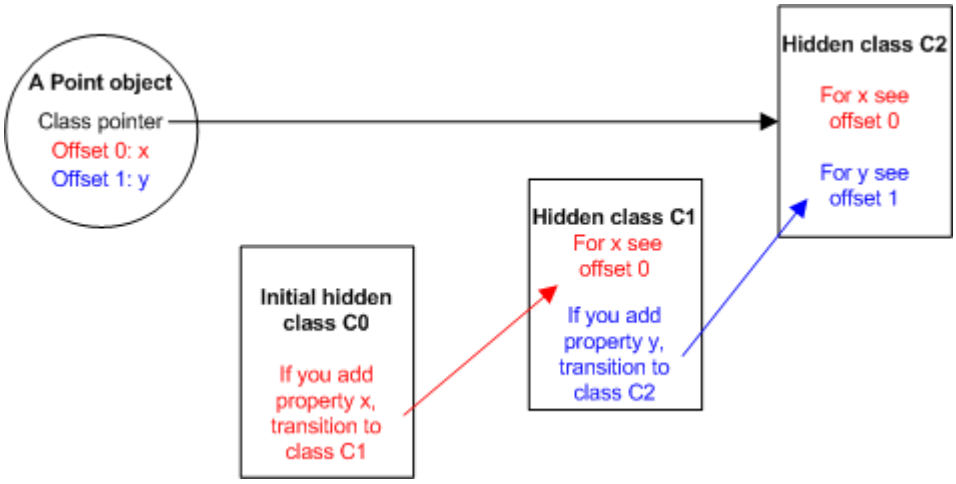
```
function Point(x, y) {
  this.x = x;
  this.y = y;
}
```

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。

## 3.1.2 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。

### 3.1.3 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。

## 3.2 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。

## 3.3 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。

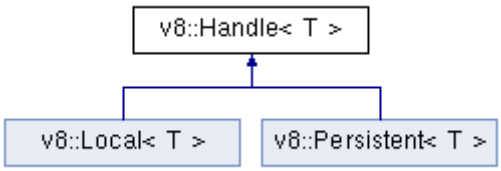
https://developers.google.com/v8/embed?csw=1#interceptors

## 3.3.1 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。




### 3.3.1.1 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。

### 3.3.1.2 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。

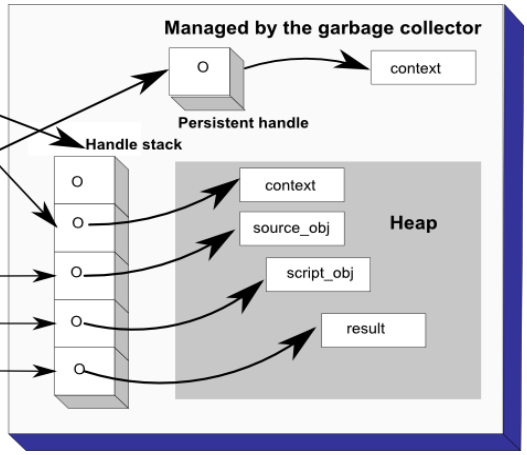
持久局部直接通过构造函数创建，通过 v8::Persistent<T>::Dispose()释放。我们可以通过 v8::Persistent<T>::MakeWeak()来创建弱持久句柄。

### 3.3.1.3 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：

```
1. HandleScope handle_scope(isolate);
2. Handle<Context> context = Context::New(isolate);
3. Persistent<Context> persistent_context(isolate, context);
4. Context::Scope context_scope(context);
5. Handle<String> source_obj = String::New(argv[1]);
6. Handle<Script> script_obj = Script::Compile(source_obj);
7. Handle<Value> local_result = script_obj->Run();
```




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);
}
```

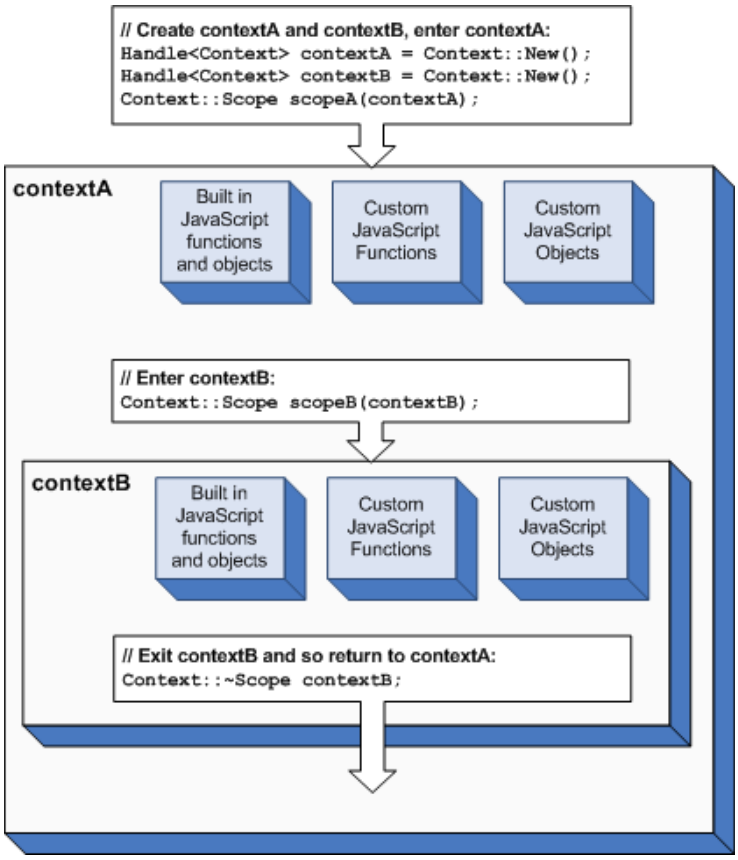
## 3.3.2 上下文(Context)

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。




## 3.3.3 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)。

#### 3.3.3.1 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。

#### 3.3.3.2 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);
```

### 3.3.4 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。

### 3.3.4.1 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);
```

### 3.3.4.2 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。

## 3.3.5 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。

### 3.3.5.1 命名属性拦截器

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

### 3.3.5.2 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);
    // ...
  }
```

## 3.3.6 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。

## 3.3.7 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);
```

## 3.3.8 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。

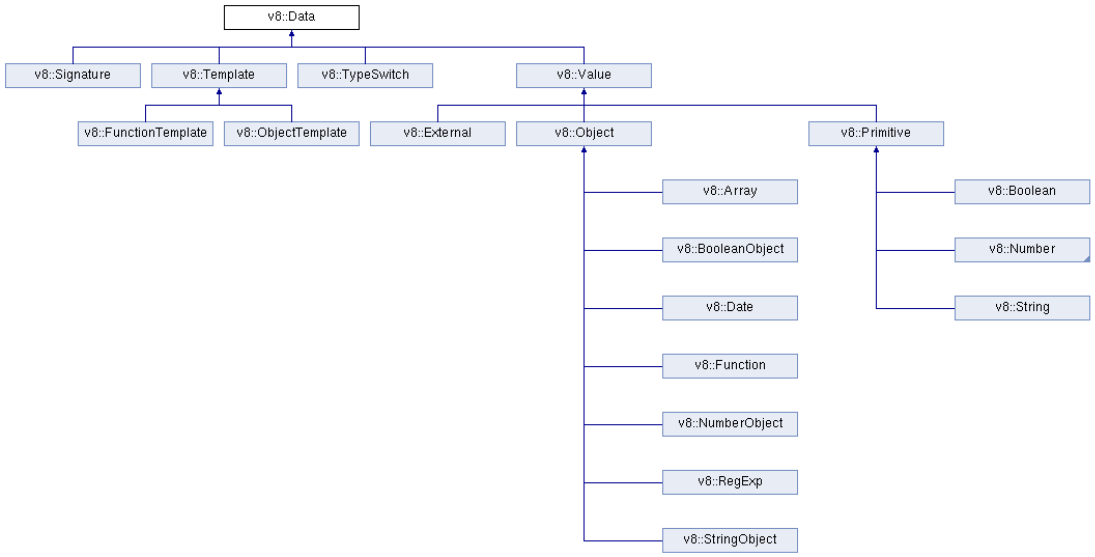
AdjustAmountOfExternalAllocatedMemory(int change_in_bytes);使用这个 API 调整注册内存的真实数量，以让 GC 在合适的时机发起垃圾回收。

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。

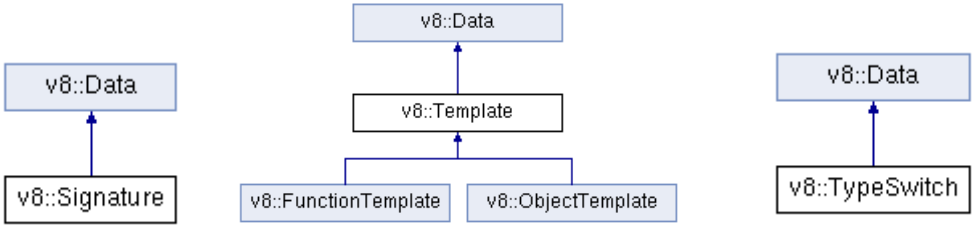
SetResourceContraints()可以在 v8 vm 初始化前调用设置 Isolate 相关 Heap 的大小，一旦 vm 初始化后，就无法再进行调整了。

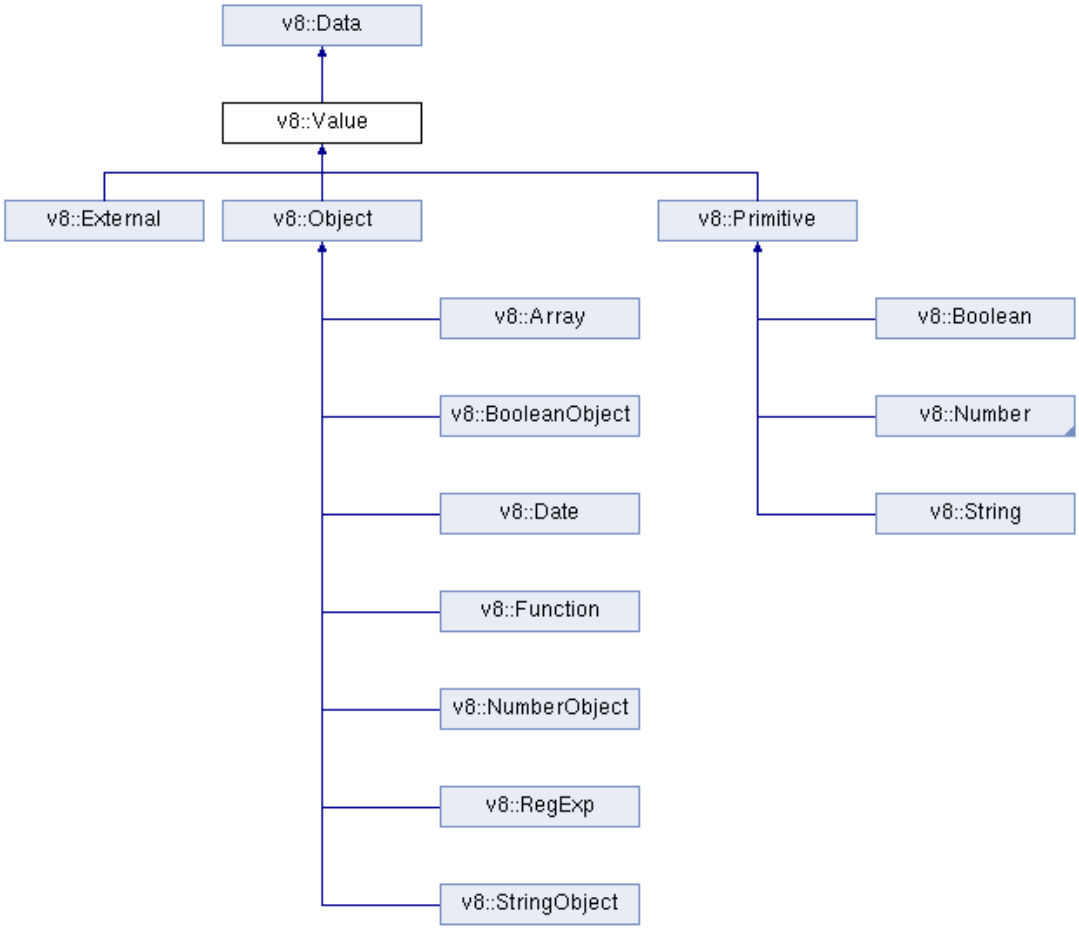
## 3.3.9 v8 的类型体系




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：






## 3.4 v8 API 手册

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

# 4 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。

## 4.1 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
}
```

## 4.2 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);
  }
```

## 4.3 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();
}
```

## 4.4 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。

## 4.5 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;
}
```

## 4.6 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。

## 4.7 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** (uint32_t index) |
| V8EXPORT bool | **Has** (**Handle**< **String** > 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**< **Value** > | **Get** (**Handle**< **Value** > key) |
| V8EXPORT **Local**< **Value** > | **Get** (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** (uint32_t index, **Handle**< **Value** > value) |
| V8EXPORT bool | **Set** (**Handle**< **Value** > key, **Handle**< **Value** > value, **PropertyAttribute** 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();
}
```

## 4.8 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);
};
```

## 4.9 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。

## 4.10Cef 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_;
};
```

## 4.11 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_;
};
```

## 4.11.1 Isolate

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

## 4.11.2 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** (void *parameters, **WeakReferenceCallback** 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)(Persistent< Value > 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。

## 4.11.3 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);
};
```

## 4.12 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;
}
```

## 4.13 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);
};
```

## 4.14 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;
};
```

## 4.15 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

# 4.16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。