# 更多的示例代码

下面的示例演示了如何使用JSAPI实现一些不同的效果。

请注意，最重要的示例位于上面的“最小示例”部分。更多JSAPI代码示例出现在JSAPI常用语手册中。

## 定义对象和属性

/\* Statically initialize a class to make "one-off" objects. \*/

JSClass my\_class = {

"MyClass",

/\* All of these can be replaced with the corresponding JS\_\*Stub

function pointers. \*/

my\_addProperty, my\_delProperty, my\_getProperty, my\_setProperty,

my\_enumerate, my\_resolve, my\_convert, my\_finalize

};

JSObject \*obj;

/\*

\* Define an object named in the global scope that can be enumerated by

\* for/in loops. The parent object is passed as the second argument, as

\* with all other API calls that take an object/name pair. The prototype

\* passed in is null, so the default object prototype will be used.

\*/

obj = JS\_DefineObject(cx, globalObj, "myObject", &my\_class, nullptr,

JSPROP\_ENUMERATE);

/\*

\* Define a bunch of properties with a JSPropertySpec array statically

\* initialized and terminated with a null-name entry. Besides its name,

\* each property has a "tiny" identifier (MY\_COLOR, e.g.) that can be used

\* in switch statements (in a common my\_getProperty function, for example).

\*/

enum my\_tinyid {

MY\_COLOR, MY\_HEIGHT, MY\_WIDTH, MY\_FUNNY, MY\_ARRAY, MY\_RDONLY

};

static JSPropertySpec my\_props[] = {

{"color", MY\_COLOR, JSPROP\_ENUMERATE},

{"height", MY\_HEIGHT, JSPROP\_ENUMERATE},

{"width", MY\_WIDTH, JSPROP\_ENUMERATE},

{"funny", MY\_FUNNY, JSPROP\_ENUMERATE},

{"array", MY\_ARRAY, JSPROP\_ENUMERATE},

{"rdonly", MY\_RDONLY, JSPROP\_READONLY},

{0}

};

JS\_DefineProperties(cx, obj, my\_props);

/\*

\* Given the above definitions and call to JS\_DefineProperties, obj will

\* need this sort of "getter" method in its class (my\_class, above). See

\* the example for the "It" class in js.c.

\*/

static JSBool

my\_getProperty(JSContext \*cx, JSObject \*obj, jsval id, jsval \*vp)

{

if (JSVAL\_IS\_INT(id)) {

switch (JSVAL\_TO\_INT(id)) {

case MY\_COLOR: \*vp = . . .; break;

case MY\_HEIGHT: \*vp = . . .; break;

case MY\_WIDTH: \*vp = . . .; break;

case MY\_FUNNY: \*vp = . . .; break;

case MY\_ARRAY: \*vp = . . .; break;

case MY\_RDONLY: \*vp = . . .; break;

}

}

return true;

}

## 定义类

通过定义一个构造函数、一个原型对象以及原型和构造函数的属性(所有这些都通过一个API调用)，可以将上述API元素组合在一起。

通过定义构造函数、原型以及每个实例和每个类的属性来初始化类。后者在下文中与Java类似，称为“静态”。它们在构造函数对象的作用域中定义，因此MyClass。myStaticProp与新MyClass()一起工作。

JS\_InitClass接受很多参数，但是如果没有这样的属性或方法，则可以为最后四个参数中的任何一个传递nullptr。

注意,你不需要叫JS\_InitClass class-otherwise的一个新实例,将会有一个“鸡生蛋还是蛋生鸡”问题的全局对象,但是你应该叫JS\_InitClass如果您需要一个脚本作者调用构造函数通过新的和/或类原型对象(MyClass.prototype)作者与新的属性在运行时扩展。通常，如果希望支持多个共享行为的实例，请使用JS\_InitClass。

protoObj = JS\_InitClass(cx, globalObj, nullptr, &my\_class,

/\* native constructor function and min arg count \*/

MyClass, 0,

/\* prototype object properties and methods -- these

will be "inherited" by all instances through

delegation up the instance's prototype link. \*/

my\_props, my\_methods,

/\* class constructor properties and methods \*/

my\_static\_props, my\_static\_methods);

## 运行脚本

/\* These should indicate source location for diagnostics. \*/

char \*filename;

unsigned int lineno;

/\*

\* The return value comes back here -- if it could be a GC thing, you must

\* add it to the GC's "root set" with JS\_AddRoot(cx, &thing) where thing

\* is a JSString \*, JSObject \*, or jsdouble \*, and remove the root before

\* rval goes out of scope, or when rval is no longer needed.

\*/

jsval rval;

JSBool ok;

/\*

\* Some example source in a C string. Larger, non-null-terminated buffers

\* can be used, if you pass the buffer length to JS\_EvaluateScript.

\*/

char \*source = "x \* f(y)";

ok = JS\_EvaluateScript(cx, globalObj, source, strlen(source),

filename, lineno, &rval);

if (ok) {

/\* Should get a number back from the example source. \*/

jsdouble d;

ok = JS\_ValueToNumber(cx, rval, &d);

. . .

}

## 调用函数

/\* Call a global function named "foo" that takes no arguments. \*/

ok = JS\_CallFunctionName(cx, globalObj, "foo", 0, 0, &rval);

jsval argv[2];

/\* Call a function in obj's scope named "method", passing two arguments. \*/

argv[0] = . . .;

argv[1] = . . .;

ok = JS\_CallFunctionName(cx, obj, "method", 2, argv, &rval);

## JSContext

您不应该为每个JSRuntime创建一个以上的JSContext。将来将删除对每个JSRuntime的多个jscontext的支持。

如果应用程序创建多个运行时，应用程序可能需要知道上下文与哪个运行时相关联。在本例中，使用JS\_GetRuntime。

使用JS\_SetContextPrivate和JS\_GetContextPrivate将特定于应用程序的数据与上下文关联起来。

## 初始化内置和全局JS对象

有关SpiderMonkey提供的内置对象的完整列表，请参见JS\_InitStandardClasses。

应用程序提供给脚本的全局对象在很大程度上决定了这些脚本可以做什么。例如，Firefox浏览器使用它自己的全局对象window。要更改应用程序的全局对象，请调用JS\_SetGlobalObject。

## 创建和初始化自定义对象

除了使用引擎的内置对象外，还将创建、初始化和使用自己的JS对象。如果您使用带有脚本的JS引擎来自动化您的应用程序，这一点尤其正确。自定义JS对象可以提供直接的程序服务，也可以作为程序服务的接口。例如，提供直接服务的自定义JS对象可以处理应用程序的所有网络访问，也可以充当数据库服务的中介代理。或者，反映应用程序中已经存在的数据和函数的JS对象可以为C代码提供面向对象的接口，严格地说，这本身并不是面向对象的。这样的自定义对象充当应用程序本身的接口，将值从应用程序传递给用户，并在将用户输入返回给应用程序之前接收和处理用户输入。这样的对象还可以用来提供对应用程序底层函数的访问控制。

有两种方法可以创建自定义对象，JS引擎可以使用:

编写一个JS脚本，创建对象、对象的属性、方法和构造函数，然后在运行时将脚本传递给JS引擎。

在应用程序中嵌入代码，定义对象的属性和方法，调用引擎初始化新对象，然后通过其他引擎调用设置对象的属性。此方法的一个优点是，应用程序可以包含直接操作对象嵌入的本机方法。

在这两种情况下，如果您创建了一个对象，然后希望它在运行时能够被其他脚本使用，那么您必须通过调用JS\_AddRoot或JS\_AddNamedRoot来根化该对象。使用这些函数可以确保JS引擎跟踪对象，并在垃圾收集期间清理它们(如果合适的话)。

## 从脚本创建对象

从脚本创建自定义JS对象的一个原因是，只要使用它的脚本在执行，就需要一个对象存在。要创建跨脚本调用持久保存的对象，可以将对象代码嵌入到应用程序中，而不是使用脚本。

注意:您还可以使用脚本创建持久对象。

使用脚本创建自定义对象:

定义并指定对象。它的目的是什么?它的数据成员(属性)是什么?它的方法(函数)是什么?它需要运行时构造函数吗?

编写定义和创建对象的JS脚本。例如:函数myfun(){var x = newObject();。注意:使用JavaScript的对象脚本编写发生在应用程序中嵌入JS引擎的上下文之外。有关对象脚本的更多信息，请参见客户端JavaScript指南和服务器端JavaScript指南。在应用程序中嵌入适当的JS引擎调用来编译和执行脚本。您有两种选择:1.)编译并执行一个脚本，只调用JS\_EvaluateScript、JS\_EvaluateUCScript或2.)编译脚本一次，调用JS\_CompileScript或JS\_CompileUCScript，然后使用对JS\_ExecuteScript的单独调用重复执行脚本。这些调用的“UC”版本提供了对unicode编码脚本的支持。

使用脚本创建的对象只能在脚本的生命周期内可用，或者可以在脚本执行完成后创建持久对象。通常，一旦脚本执行完成，它的对象就会被销毁。在许多情况下，这种行为正是应用程序所需要的。然而，在其他情况下，您将希望跨脚本或应用程序的生命周期保持对象持久性。在这些情况下，您需要将对象创建代码直接嵌入到应用程序中，或者您需要将对象直接绑定到全局对象，这样，只要全局对象本身持续存在，它就会持续存在。

## 自定义对象

应用程序可以创建自定义对象，而不需要使用JSClass:

用C或c++实现自定义对象的getter、setter和方法。为每个getter或setter编写JSPropertyOp。为每个方法编写一个JSNative或JSFastNative。

声明一个JSPropertySpec数组，其中包含关于自定义对象属性的信息，包括getter和setter。

声明一个包含有关自定义对象方法的信息的JSFunctionSpec数组。

调用JS\_NewObject、JS\_ConstructObject或JS\_DefineObject来创建对象。

调用JS\_DefineProperties来定义对象的属性。

调用JS\_DefineFunctions来定义对象的方法。

还可以使用JS\_SetProperty在对象上创建属性。它创建的属性没有getter或setter;它们是普通的JavaScript属性。

## 为对象提供私有数据

与上下文类似，您可以将大量数据与对象关联，而不必将数据存储在对象本身中。调用JS\_SetPrivate来为对象建立指向私有数据的指针，并调用JS\_GetPrivate来检索指针，以便访问数据。您的应用程序负责创建和管理这个可选的私有数据。

要创建私有数据并将其与对象关联:

建立私有数据，就像建立一个普通的C空指针变量一样。

调用JS\_SetPrivate，指定用于建立私有数据的对象，并指定指向数据的指针。

例如:

JS\_SetPrivate(cx, obj, pdata);

要在稍后检索数据，请调用JS\_GetPrivate，并将对象作为参数传递。这个函数返回指向对象私有数据的指针:

pdata = JS\_GetPrivate(cx, obj);

# 特别的主题

## Unicode

要在JavaScript和本机代码之间传递Unicode数据，请在内存中表示UTF-16中的数据。JavaScript字符串、属性名和程序都由jschars组成，jschars是16位无符号整数。

许多JSAPI函数都对以null结尾的8位char字符串进行操作。这些函数通过将每个8位字符从零开始扩展到16位，将它们的char \*参数转换为16位字符串，除非定义了JS\_C\_STRINGS\_ARE\_UTF8或调用了JS\_SetCStringsAreUTF8，否则在这种情况下，每个char \*字符串都被解释为UTF-8 Unicode文本。

JSAPI提供了许多基于jschar的API函数版本，这些API函数操作字符串、对象属性和JavaScript代码。

|  |  |
| --- | --- |
| **char-based function** | **jschar-based function** |
| *Unicode data* | |
| JS\_GetStringBytes Obsolete since JavaScript 1.8.5 | [JS\_GetStringChars](https://developer.mozilla.org/en/SpiderMonkey/JSAPI_Reference/JS_GetStringChars) |
| [JS\_NewString](https://developer.mozilla.org/en/SpiderMonkey/JSAPI_Reference/JS_NewString) | [JS\_NewUCString](https://developer.mozilla.org/en/SpiderMonkey/JSAPI_Reference/JS_NewString) |
| [JS\_NewStringCopyN](https://developer.mozilla.org/en/SpiderMonkey/JSAPI_Reference/JS_NewStringCopyN) | [JS\_NewUCStringCopyN](https://developer.mozilla.org/en/SpiderMonkey/JSAPI_Reference/JS_NewStringCopyN) |
| [JS\_NewStringCopyZ](https://developer.mozilla.org/en/SpiderMonkey/JSAPI_Reference/JS_NewStringCopyZ) | [JS\_NewUCStringCopyZ](https://developer.mozilla.org/en/SpiderMonkey/JSAPI_Reference/JS_NewStringCopyZ) |
| [JS\_InternString](https://developer.mozilla.org/en/SpiderMonkey/JSAPI_Reference/JS_InternString) | JS\_InternUCString, JS\_InternUCStringN |
| [JS\_ReportErrorNumber](https://developer.mozilla.org/en/SpiderMonkey/JSAPI_Reference/JS_ReportErrorNumber) | [JS\_ReportErrorNumberUC](https://developer.mozilla.org/en/SpiderMonkey/JSAPI_Reference/JS_ReportErrorNumber) |
| [JS\_ReportErrorFlagsAndNumber](https://developer.mozilla.org/en/SpiderMonkey/JSAPI_Reference/JS_ReportErrorNumber) | [JS\_ReportErrorFlagsAndNumberUC](https://developer.mozilla.org/en/SpiderMonkey/JSAPI_Reference/JS_ReportErrorNumber) |
| *Unicode property names* | |
| [JS\_DefineProperty](https://developer.mozilla.org/en/SpiderMonkey/JSAPI_Reference/JS_DefineProperty) | [JS\_DefineUCProperty](https://developer.mozilla.org/en/SpiderMonkey/JSAPI_Reference/JS_DefineProperty) |
| [JS\_DefinePropertyWithTinyId](https://developer.mozilla.org/en/SpiderMonkey/JSAPI_Reference/JS_DefinePropertyWithTinyId) | [JS\_DefineUCPropertyWithTinyId](https://developer.mozilla.org/en/SpiderMonkey/JSAPI_Reference/JS_DefinePropertyWithTinyId) |
| [JS\_DefineFunction](https://developer.mozilla.org/en/SpiderMonkey/JSAPI_Reference/JS_DefineFunction) | [JS\_DefineUCFunction](https://developer.mozilla.org/en/SpiderMonkey/JSAPI_Reference/JS_DefineFunction) |
| [JS\_HasProperty](https://developer.mozilla.org/en/SpiderMonkey/JSAPI_Reference/JS_HasProperty) | [JS\_HasUCProperty](https://developer.mozilla.org/en/SpiderMonkey/JSAPI_Reference/JS_HasProperty) |
| [JS\_LookupProperty](https://developer.mozilla.org/en/SpiderMonkey/JSAPI_Reference/JS_LookupProperty) | [JS\_LookupUCProperty](https://developer.mozilla.org/en/SpiderMonkey/JSAPI_Reference/JS_LookupProperty) |
| [JS\_GetProperty](https://developer.mozilla.org/en/SpiderMonkey/JSAPI_Reference/JS_GetProperty) | [JS\_GetUCProperty](https://developer.mozilla.org/en/SpiderMonkey/JSAPI_Reference/JS_GetProperty) |
| [JS\_GetPropertyAttributes](https://developer.mozilla.org/en/SpiderMonkey/JSAPI_Reference/JS_GetPropertyAttributes) | [JS\_GetUCPropertyAttributes](https://developer.mozilla.org/en/SpiderMonkey/JSAPI_Reference/JS_GetPropertyAttributes) |
| [JS\_GetPropertyAttrsGetterAndSetter](https://developer.mozilla.org/en/SpiderMonkey/JSAPI_Reference/JS_GetPropertyAttrsGetterAndSetter) | [JS\_GetUCPropertyAttrsGetterAndSetter](https://developer.mozilla.org/en/SpiderMonkey/JSAPI_Reference/JS_GetPropertyAttrsGetterAndSetter) |
| [JS\_SetProperty](https://developer.mozilla.org/en/SpiderMonkey/JSAPI_Reference/JS_SetProperty) | [JS\_SetUCProperty](https://developer.mozilla.org/en/SpiderMonkey/JSAPI_Reference/JS_SetProperty) |
| [JS\_SetPropertyAttributes](https://developer.mozilla.org/en/SpiderMonkey/JSAPI_Reference/JS_SetPropertyAttributes) | [JS\_SetUCPropertyAttributes](https://developer.mozilla.org/en/SpiderMonkey/JSAPI_Reference/JS_SetPropertyAttributes) |
| [JS\_DeleteProperty2](https://developer.mozilla.org/en/SpiderMonkey/JSAPI_Reference/JS_DeleteProperty2) | [JS\_DeleteUCProperty2](https://developer.mozilla.org/en/SpiderMonkey/JSAPI_Reference/JS_DeleteProperty2) |
| [JS\_AlreadyHasOwnProperty](https://developer.mozilla.org/en/SpiderMonkey/JSAPI_Reference/JS_AlreadyHasOwnProperty) | [JS\_AlreadyHasOwnUCProperty](https://developer.mozilla.org/en/SpiderMonkey/JSAPI_Reference/JS_AlreadyHasOwnProperty) |
| *Unicode JavaScript source* | |
| [JS\_CompileScript](https://developer.mozilla.org/en/SpiderMonkey/JSAPI_Reference/JS_CompileScript) | [JS\_CompileUCScript](https://developer.mozilla.org/en/SpiderMonkey/JSAPI_Reference/JS_CompileScript) |
| [JS\_CompileScriptForPrincipals](https://developer.mozilla.org/en/SpiderMonkey/JSAPI_Reference/JS_CompileScriptForPrincipals) | [JS\_CompileUCScriptForPrincipals](https://developer.mozilla.org/en/SpiderMonkey/JSAPI_Reference/JS_CompileScriptForPrincipals) |
| [JS\_CompileFunction](https://developer.mozilla.org/en/SpiderMonkey/JSAPI_Reference/JS_CompileFunction) | [JS\_CompileUCFunction](https://developer.mozilla.org/en/SpiderMonkey/JSAPI_Reference/JS_CompileFunction) |
| [JS\_CompileFunctionForPrincipals](https://developer.mozilla.org/en/SpiderMonkey/JSAPI_Reference/JS_CompileFunctionForPrincipals) | [JS\_CompileUCFunctionForPrincipals](https://developer.mozilla.org/en/SpiderMonkey/JSAPI_Reference/JS_CompileFunctionForPrincipals) |
| [JS\_EvaluateScript](https://developer.mozilla.org/en/SpiderMonkey/JSAPI_Reference/JS_EvaluateScript) | [JS\_EvaluateUCScript](https://developer.mozilla.org/en/SpiderMonkey/JSAPI_Reference/JS_EvaluateScript) |
| [JS\_EvaluateScriptForPrincipals](https://developer.mozilla.org/en/SpiderMonkey/JSAPI_Reference/JS_EvaluateScriptForPrincipals) | [JS\_EvaluateUCScriptForPrincipals](https://developer.mozilla.org/en/SpiderMonkey/JSAPI_Reference/JS_EvaluateScriptForPrincipals) |

基于jschar的函数的工作原理与基于字符的同名函数完全相同，只是传统函数使用char \*参数，Unicode版本使用jschar \*参数，通常使用一个单独的参数在jschars中指定字符串的长度。

## 编译脚本

运行脚本最简单的方法是使用JS\_EvaluateScript，它一次编译并执行脚本。

但有时应用程序需要多次运行脚本。在这种情况下，编译脚本一次并执行多次可能更快。

JSAPI提供了一种类型JSScript，它表示编译后的脚本。JSScript的生命周期是这样的:

应用程序使用JS\_CompileScript、JS\_CompileUTF8File或JS\_CompileFileHandle编译一些JavaScript代码。这些函数返回一个指向新JSScript的指针。

应用程序调用JS\_ExecuteScript(或JS\_ExecuteScriptPart)任意次数。在多个不同的上下文中使用JSScript是安全的，但是只能在创建它的JSRuntime和全局上下文中使用。

下面是一些使用编译脚本的示例代码:

/\*

\* Compile a script and execute it repeatedly until an

\* error occurs. (If this ever returns, it returns false.

\* If there's no error it just keeps going.)

\*/

JSBool compileAndRepeat(JSContext \*cx, const char \*filename)

{

JSScript \*script;

script = JS\_CompileUTF8File(cx, JS\_GetGlobalObject(cx), filename);

if (!script)

return false; /\* compilation error \*/

for (;;) {

jsval result;

if (!JS\_ExecuteScript(cx, JS\_GetGlobalObject(cx), script, &result))

break;

JS\_MaybeGC(cx);

}

return false;

}

编译脚本的生存期与JavaScript对象的生存期绑定在一起，垃圾收集器将在脚本不再可访问时销毁该脚本。JSAPI通过JS\_NewScriptObject函数提供该特性。使用此功能的脚本的生命周期如下:

应用程序编译一些JavaScript代码。

为了保护编译后的脚本不被垃圾收集，应用程序通过调用JS\_NewScriptObject创建一个编译后的脚本对象，并使用JS\_SetProperty、JS\_SetReservedSlot、JS\_AddRoot或其他一些函数使该对象gc可访问。

应用程序可以任意次数地执行编译后的脚本。

随着应用程序的发展，最终它不再需要编译后的脚本，并且编译后的脚本对象变得不可访问。

垃圾收集器最终收集不可到达的脚本及其组件。

下面是演示该技术的示例代码—但是请注意，这种情况并不复杂到足以保证使用JS\_NewScriptObject。上面的例子更直接地做了同样的事情。

/\*

\* Compile a script and execute it repeatedly until an

\* error occurs. (If this ever returns, it returns false.

\* If there's no error it just keeps going.)

\*/

JSBool compileAndRepeat(JSContext \*cx, const char \*filename)

{

JSScript \*script;

JSObject \*scriptObj;

script = JS\_CompileUTF8File(cx, JS\_GetGlobalObject(cx), filename);

if (!script)

return false; /\* compilation error \*/

scriptObj = JS\_NewScriptObject(cx, script);

if (!scriptObj) {

JS\_DestroyScript(cx, script);

return false;

}

if (!JS\_AddNamedObjectRoot(cx, &scriptObj, "compileAndRepeat script object"))

return false;

for (;;) {

jsval result;

if (!JS\_ExecuteScript(cx, JS\_GetGlobalObject(cx), script, &result))

break;

JS\_MaybeGC(cx);

}

JS\_RemoveObjectRoot(cx, &scriptObj); /\* scriptObj becomes unreachable

and will eventually be collected. \*/

return false;

}

## 安全

许多应用程序使用SpiderMonkey运行不受信任的代码。在设计这种应用程序时，提前考虑安全性问题是很重要的。您的应用程序将需要做几件事。

部署安全更新——Firefox会自动安装更新，所以一旦安全补丁可用，就会立即部署。除非您还定期部署SpiderMonkey安全更新，否则坚定的黑客可能会使用引擎中已知的bug攻击您的应用程序。注意，我们在这里讨论的攻击类型是黑客使用JavaScript攻击引擎本身的c++代码(或您的嵌入)。这个列表中的其他项目讨论JavaScript本身出现的安全问题，即使引擎正常工作。

阻止简单的拒绝服务攻击——像while(true){}这样的程序不应该挂起您的应用程序。要停止运行时间过长的脚本的执行，可以使用JS\_SetOperationCallback。同样，类似函数f(){f();}的函数不应该因为堆栈溢出而导致应用程序崩溃。要阻止这种情况，可以使用JS\_SetNativeStackQuota。

控制对敏感数据的访问——您的应用程序可能会将数据暴露给其他脚本不应该看到的一些脚本。例如，您可以让您的客户编写脚本来操作他们自己的数据，而不是其他客户的数据。必须以某种方式强制执行这些访问规则。

控制对危险功能的访问——假设您的应用程序有一个只能由管理员使用的方法deleteUserAccount()。显然，如果不受信任的代码可以使用该方法，那么就存在安全问题。

前两个问题很重要，但相当直接，这里不再讨论。本节的其余部分将说明如何控制脚本对数据和功能的访问。

最好的安全就是没有安全(真的)

你曾经担心过你的蛇会吃掉你的老鼠吗?没有?如果你没有同时拥有一条蛇和一只老鼠，你就不会有这个问题。

同样，如果您的应用程序既没有不受信任的用户(snake)，也没有敏感数据或暴露在JavaScript(鼠标)中的危险功能，那么您就不需要进一步阅读。JS\_InitStandardClasses创建的函数和对象是安全的。它们不提供对文件、网络或任何与浏览器相关的东西的访问。它们暴露给脚本的最敏感的信息是当前日期和时间。

Object-capabilities-based安全

防止蛇吃老鼠的一种方法是把老鼠和蛇关在单独的笼子里。

防止用户A访问用户B的敏感数据或危险函数的一种方法是将每个用户的代码保存在单独的沙箱中。也就是说，为每个用户创建一个单独的JSContext和全局对象，并始终在适当的上下文中运行每个脚本。在设置新的全局对象时，不要定义用户不应该访问的任何函数。这种方法称为对象功能安全性。想要了解更多，可以看电影或看书。

这个比喻在某种程度上是误导人的:蛇不能够到老鼠，因为有一个物理屏障挡住了它。而蜘蛛猴的情况则更为微妙。没有障碍;从这里到那是不可能的。恶意脚本如何从另一个沙箱获得对对象的引用?这就像在一个平行宇宙中一样。甚至全局变量都是每个沙箱的。没有getObjectFromOtherSandbox()函数。您的应用程序只需要注意不要向脚本公开任何这样的函数。

简而言之，永远不要将一个用户的数据和对象传递给另一个用户的代码，这样就不会有访问控制问题。如果你不这么做，蜘蛛猴是不会这么做的。

权衡。对象功能安全性是没有运行时安全性检查的安全性。它易于实现，易于推理，而且速度快。但公平地说，也有一些缺点。首先，故障模式相当严重。如果你不小心把一个物体从一个沙箱漏到另一个沙箱，妖怪就从瓶子里出来了。一旦恶意脚本获得对沙箱中单个对象的引用，它就可以使用该对象来获得对沙箱的全局对象的引用，并且从那里，几乎可以获得沙箱中任何其他对象或函数的引用。除了破坏两个沙箱并重新开始之外，没有其他方法可以修复它。第二个缺点是系统不会自动响应用户权限的更改。假设用户A不是管理员，因此您设置了没有管理员功能的沙箱A。如果将用户A提升为管理员，SpiderMonkey不会神奇地更新沙箱A，使其具有以前没有定义的管理员类和函数。您的应用程序必须显式地做到这一点。相反，如果您想剥夺用户A的管理员特权，但是您已经为用户A的脚本提供了管理员功能，那就更糟了。您别无选择，只能销毁用户A的沙箱并重新开始。

|  |  |
| --- | --- |
| **Function** | **Purpose** |
| [JS\_CompileScriptForPrincipals,](https://developer.mozilla.org/en/SpiderMonkey/JSAPI_Reference/JS_CompileScriptForPrincipals) | [Compile a script with security information. (To execute a compiled script, use JS\_ExecuteScript.)](https://developer.mozilla.org/en/SpiderMonkey/JSAPI_Reference/JS_ExecuteScript) |
| [JS\_CompileUCScriptForPrincipals,](https://developer.mozilla.org/en/SpiderMonkey/JSAPI_Reference/JS_CompileScriptForPrincipals) |
| [JS\_CompileFileHandleForPrincipals](https://developer.mozilla.org/en/SpiderMonkey/JSAPI_Reference/JS_CompileFileHandle) |
| [JS\_CompileFunctionForPrincipals,](https://developer.mozilla.org/en/SpiderMonkey/JSAPI_Reference/JS_CompileFunctionForPrincipals) | Create a JavaScript function with security information. |
| [JS\_CompileUCFunctionForPrincipals](https://developer.mozilla.org/en/SpiderMonkey/JSAPI_Reference/JS_CompileFunctionForPrincipals) |
| [JS\_EvaluateScriptForPrincipals,](https://developer.mozilla.org/en/SpiderMonkey/JSAPI_Reference/JS_EvaluateScriptForPrincipals) | Compile and execute a script with security information. |
| [JS\_EvaluateUCScriptForPrincipals](https://developer.mozilla.org/en/SpiderMonkey/JSAPI_Reference/JS_EvaluateScriptForPrincipals) |

实现访问检查回调函数(参见JSClass)。checkAccess和JS\_SetCheckObjectAccessCallback)。这些函数将从JS\_CheckAccess调用，有时在JavaScript引擎中调用。访问检查回调函数可以使用jsdbgapi.h函数(如JS\_FrameIterator和js\_stackframeprincipal)来获取试图执行检查操作的代码的主体。然后它决定是否允许操作继续进行。

## 跟踪和分析

JSAPI提供了一些特性，这些特性使得实现JavaScript跟踪器和分析器更加容易。

跟踪功能

如果您配置了——enable-trace-jscalls，那么您可以使用JS\_SetFunctionCallback()来设置一个C函数，以便在JavaScript函数即将被调用或已经执行完毕时调用:

void funcTransition(const JSFunction \*func,

const JSScript \*scr,

const JSContext \*const\_cx,

JSBool entering)

{

JSContext \*cx = const\_cast<JSContext\*>(const\_cx);

JSString \*name = JS\_GetFunctionId((JSFunction\*)func);

const char \*entExit;

const char \*nameStr;

/\* build a C string for the function's name \*/

if (!name) {

nameStr = "Unnamed function";

} else {

nameStr = JS\_EncodeString(cx, name);

}

/\* build a string for whether we're entering or exiting \*/

if (entering) {

entExit = "Entering";

} else {

entExit = "Exiting";

}

/\* output information about the trace \*/

printf("%s JavaScript function: %s at time: %ld", entExit, nameStr, clock());

}

void enableTracing(JSContext \*cx) {

JS\_SetFunctionCallback(cx, funcTransition);

}

void disableTracing(JSContext \*cx) {

JS\_SetFunctionCallback(cx, nullptr);

}