本文档解释了如何将Mozilla JavaScript引擎SpiderMonkey嵌入到您的c++程序中。

JavaScript广泛用于在浏览器中运行的客户端脚本。但是Mozilla的JavaScript引擎是一个库，可以链接到任何c++程序，而不仅仅是一个浏览器。许多应用程序都可以从脚本中获益。这些程序可以使用SpiderMonkey API从c++执行JavaScript代码。

注意:FOSS wiki页面包含了一些指向其他库和程序的链接，这些库和程序在使用Spidermonkey和JSAPI时可以变得更简单。

# 什么SpiderMonkey doesSection

JavaScript引擎编译并执行包含JavaScript语句和函数的脚本。引擎为执行脚本所需的对象处理内存分配，并清理垃圾收集—不再需要的对象。

JavaScript这个词可能会让人想起事件处理程序(如onclick)、DOM对象和窗口等特性。开放和XMLHttpRequest。但是在Mozilla中，所有这些特性实际上都是由其他组件提供的，而不是SpiderMonkey引擎本身。SpiderMonkey提供了一些核心JavaScript数据类型(数字、字符串、数组、对象等)和一些方法，比如Array.push。它还使每个应用程序可以很容易地将自己的一些对象和函数公开给JavaScript代码。浏览器公开DOM对象。您的应用程序将公开与您想要编写的脚本类型相关的对象。由应用程序开发人员决定向脚本公开哪些对象和方法。

## 你好，世界

**使用SpiderMonkey库**

要从源代码构建SpiderMonkey，请参阅SpiderMonkey构建文档。

一些系统(如Debian)提供SpiderMonkey作为预构建包。而是从源代码构建它。您的程序将更容易调试。

c++代码通过包含头部“JSAPI .h”来通过JSAPI访问SpiderMonkey。下面是JSAPI功能的概述。有关更多细节，请参见JSAPI引用。

**SpiderMonkey的宇宙**

为了在SpiderMonkey中运行任何JavaScript代码，应用程序必须有三个关键元素:JSRuntime、JSContext和全局对象。本节将描述这些内容。下一节将解释如何使用JSAPI函数设置它们。

**运行时。**JSRuntime或运行时是分配应用程序使用的JavaScript变量、对象、脚本和上下文的空间。应用程序中的每个JSContext和每个对象都生活在JSRuntime中。它们不能移动到其他运行时，也不能跨运行时共享。每个使用JSAPI的线程至少需要一个JSRuntime。

**上下文。**JSContext就像一台小机器，可以做许多涉及JavaScript代码和对象的事情。它可以编译和执行脚本、获取和设置对象属性、调用JavaScript函数、将JavaScript数据从一种类型转换为另一种类型、创建对象等等。几乎所有JSAPI函数都需要JSContext \*作为第一个参数，就像大多数h>函数需要一个文件\*。

**全局对象。**最后，全局对象包含JavaScript代码可以使用的所有类、函数和变量。每当JavaScript代码执行诸如window.open(“http://www.mozilla.org/”)之类的操作时，它都在访问一个全局属性，在本例中是window。JSAPI应用程序完全控制全局属性脚本可以看到什么。应用程序首先创建一个对象并用标准JavaScript类(如Array和object)填充它。然后它添加应用程序想要提供的任何自定义类、函数和变量(如窗口);参见下面的自定义对象。每次应用程序运行JS脚本(例如，使用JS\_EvaluateScript)时，它都会为脚本提供要使用的全局对象。当脚本运行时，它可以创建自己的全局函数和变量。所有这些函数、类和变量都存储为全局对象的属性。

## 一个很小的例子

上一节描述的三个关键元素都需要一些JSAPI调用:

**运行时:**使用JS\_NewRuntime创建它，当您完成时，使用JS\_DestroyRuntime清理它。当您的应用程序完全使用SpiderMonkey完成时，使用JS\_ShutDown来释放剩余的缓存资源。(如果这个过程无论如何都要退出，那么这只是一个小细节。但是，由于并不总是这样，所以调用JS\_Shutdown是一个好习惯。)

**上下文:**使用JS\_NewContext和JS\_DestroyContext。为了最大限度地符合ECMAScript标准，应用程序还应该使用JS\_SetOptions来启用JSOPTION\_VAROBJFIX。要获得最新的JavaScript语言特性，应用程序可以使用JS\_SetVersion。错误报告也是针对上下文的，并使用JS\_SetErrorReporter启用。

**全局对象:**要创建这个对象，首先需要一个带有JSCLASS\_GLOBAL\_FLAGS选项的JSClass。下面的示例定义了一个非常基本的JSClass(名为global\_class)，它没有自己的方法或属性。使用JS\_NewGlobalObject创建一个全局对象。使用JS\_InitStandardClasses用标准的JavaScript全局变量填充它。

对于一个简单的应用程序来说，这似乎包含了很多内容。它相当于大约80行代码，如下所示。但是JSAPI被设计成可伸缩到需要许多全局对象的应用程序。它是一个细粒度的API，支持许多不同的部件组合，并为应用程序提供了对SpiderMonkey行为的精确控制。

下面是一个最小的JSAPI应用程序所需的样板代码。它包含了上面描述的所有内容。

#include "jsapi.h"

using namespace JS;

// The class of the global object.

static JSClass globalClass = {

"global",

JSCLASS\_GLOBAL\_FLAGS,

JS\_PropertyStub,

JS\_DeletePropertyStub,

JS\_PropertyStub,

JS\_StrictPropertyStub,

JS\_EnumerateStub,

JS\_ResolveStub,

JS\_ConvertStub,

nullptr, nullptr, nullptr, nullptr,

JS\_GlobalObjectTraceHook

};

// The error reporter callback.

void reportError(JSContext \*cx, const char \*message, JSErrorReport \*report) {

fprintf(stderr, "%s:%u:%s\n",

report->filename ? report->filename : "[no filename]",

(unsigned int) report->lineno,

message);

}

int run(JSContext \*cx) {

// Enter a request before running anything in the context.

JSAutoRequest ar(cx);

// Create the global object and a new compartment.

RootedObject global(cx);

global = JS\_NewGlobalObject(cx, &globalClass, nullptr,

JS::DontFireOnNewGlobalHook);

if (!global)

return 1;

// Enter the new global object's compartment.

JSAutoCompartment ac(cx, global);

// Populate the global object with the standard globals, like Object and

// Array.

if (!JS\_InitStandardClasses(cx, global))

return 1;

// Your application code here. This may include JSAPI calls to create your

// own custom JS objects and run scripts.

return 0;

}

int main(int argc, const char \*argv[]) {

// Initialize the JS engine.

if (!JS\_Init())

return 1;

// Create a JS runtime.

JSRuntime \*rt = JS\_NewRuntime(8L \* 1024L \* 1024L);

if (!rt)

return 1;

// Create a context.

JSContext \*cx = JS\_NewContext(rt, 8192);

if (!cx)

return 1;

JS\_SetErrorReporter(rt, reportError);

int status = run(cx);

// Shut everything down.

JS\_DestroyContext(cx);

JS\_DestroyRuntime(rt);

JS\_ShutDown();

return status;

}

每个JSNative都有相同的签名，不管它期望从JavaScript接收什么参数。

函数的JavaScript参数在argc和vp中给出。argc告诉调用者传递了多少实际参数，JS\_ARGV(cx, vp)返回这些参数的数组。参数没有像int和float这样的原生c++类型;相反，它们是jsvals, JavaScript值。本机函数使用JS\_ConvertArguments将参数转换为c++类型并将它们存储在本地变量中。本机函数使用JS\_SET\_RVAL(cx, vp, val)来存储它的JavaScript返回值。

成功时，JSNative必须调用JS\_SET\_RVAL并返回true。传递给JS\_SET\_RVAL的值将返回给JavaScript调用者。

失败时，JSNative调用错误报告函数，在本例中是JS\_ReportError，并返回false。这会引发JavaScript异常。调用者可以使用JavaScript try/catch语句捕捉异常。

要从JavaScript调用本机函数，请声明一个描述函数的JSFunctionSpecs表。然后调用JS\_DefineFunctions。

static JSFunctionSpec myjs\_global\_functions[] = {

JS\_FS("rand", myjs\_rand, 0, 0),

JS\_FS("srand", myjs\_srand, 0, 0),

JS\_FS("system", myjs\_system, 1, 0),

JS\_FS\_END

};

...

if (!JS\_DefineFunctions(cx, global, myjs\_global\_functions))

return false;

...

一旦在global中定义了函数，任何使用global作为全局对象的脚本都可以调用它们，就像任何web页面都可以调用alert一样。在我们创建的环境中，“hello world”脚本如下所示:

system("echo hello world");

## JSAPI的概念

本节的目的是填补到目前为止介绍的JSAPI的主要空白。要使用SpiderMonkey做任何有用的事情，您必须阅读所有三个部分。

**JavaScript的价值观**

主要文章:JS:价值

JavaScript是一种动态类型语言:变量和属性没有在编译时固定的类型。静态类型语言，如C或c++，所有变量都有类型，如何与JavaScript交互?JSAPI提供了一个数据类型JS::Value(也使用了一个不推荐的jsval类型定义)，它可以包含任何类型的JavaScript值。值可以是数字、字符串、布尔值、对对象的引用(如对象、数组、日期或函数)，也可以是一个特殊值null或undefined。

对于整数和布尔值，jsval本身包含值。在其他情况下，jsval是指向对象、字符串或数字的指针。

警告:与c++指针一样，与JavaScript vars不同的是，JS::Value不会自动初始化为一个安全值，并且可能成为一个悬空指针!

悬空指针是一个指针，用于指向一个有效的对象，但由于该对象不再存在而不再指向它。使用悬空指针会导致c++程序崩溃(或者更糟)。在JS::Value的情况下，JavaScript垃圾收集器回收没有被使用的对象、字符串和数字，而JS::Value本身并不能保护它的引用不受垃圾收集器的影响。有关如何安全地使用JS::值的关键信息，请参阅下面的垃圾收集。

Value包含成员函数来测试JavaScript数据类型。。它们是isObject()、isNumber()、isInt32()、isDouble()、isString()、isBoolean()、isNull()和isUndefined()。

如果一个JS::Value包含一个JSObject、double或JSString，您可以分别使用toObject()、toDouble()和toString()成员函数将其转换为它的底层数据类型。在应用程序或JSAPI函数需要特定数据类型的变量或参数而不是JS::Value的情况下，这很有用。类似地，可以使用JS::ObjectValue(jsobject&)、JS::DoubleValue(double)或JS::StringValue(JSString\*)创建一个JS::Value包装一个JSObject、double或JSString指针来指向JS::Value。

**垃圾收集**

在运行时，JavaScript代码隐式地为对象、字符串、变量等分配内存。垃圾收集是JavaScript引擎检测哪些内存片段不再可用(即它们不可能再次被使用)并回收内存的过程。

对于JSAPI应用程序，垃圾收集有两个重要的结果。首先，应用程序必须非常小心，确保它需要的任何值都是gc可访问的。垃圾收集器对它的工作相当热心。如果您不告诉JSAPI您仍然在使用它，那么您留下的任何对象都将被销毁。其次，应用程序应该采取步骤减少垃圾收集对性能的影响。

**保持对象活着**

如果您的JSAPI应用程序崩溃，很可能是由于gc相关的错误。应用程序必须确保垃圾收集器能够访问仍然在使用的所有对象、数字和字符串。否则，GC将释放这些值占用的内存，导致程序下一次尝试使用这些值时可能崩溃。

有许多方法可以确保一个值是gc可访问的。

如果只需要该值在JSNative调用期间保持可访问性，则将其存储在\*rval或argv数组的元素中。存储在这些位置中的值总是可访问的。要获得额外的argv插槽，请使用JSFunctionSpec.extra。

如果自定义对象需要某些值保留在内存中，只需将这些值存储在对象的属性中。只要对象是可访问的，它的属性就保持可访问。如果这些值不能从JavaScript访问，则使用保留的插槽。或者将值存储在私有数据中并实现JSClass.mark。

如果一个函数创建了新的对象、字符串或数字，它可以使用JS\_EnterLocalRootScope和JS\_LeaveLocalRootScope在函数运行期间保持这些值为活动的。

要使值永久存活，请将其存储在GC根中。

尽管如此，GC bug还是会发生。这两个函数都只在调试构建中可用，对于调试gc相关的崩溃特别有用:

使用js\_setgc启用额外的垃圾收集。GC热情通常会导致GC相关崩溃更快(更接近其原因)和更可靠地发生。它只用于开发和调试，因为额外的垃圾收集使JS非常慢。

使用JS\_DumpHeap转储SpiderMonkey堆或其中特定的有趣部分。

有关详细信息，请参阅SpiderMonkey垃圾收集技巧。

**GC的性能**

过于频繁的垃圾收集可能会导致性能问题。一些应用程序可以通过增加JSRuntime的初始大小来降低垃圾收集的频率。

也许最好的技术是在空闲时间执行垃圾收集，因为空闲时间对用户的影响最小。默认情况下，JavaScript引擎在别无选择的情况下执行垃圾收集，只能增长进程。这意味着垃圾收集通常发生在内存密集型代码运行时，这可能是最糟糕的时间。应用程序可以通过调用JS\_GC或JS\_MaybeGC在更方便的时间触发垃圾收集。JS\_GC强制执行垃圾收集。JS\_MaybeGC只在可能回收有价值的内存时才执行垃圾收集。

**错误和异常**

当然，检查JSAPI函数返回值的重要性是不言而喻的。几乎每个接受JSContext \*参数的JSAPI函数都可能失败。系统可能会耗尽内存。脚本中可能有语法错误。或者脚本可能显式地抛出异常。

JavaScript语言有异常，c++也有异常，但它们不是一回事。SpiderMonkey对任何东西都不使用c++异常。JSAPI函数从不抛出c++异常，当SpiderMonkey调用应用程序回调时，回调必须不抛出c++异常。

**抛出和捕获异常**

我们已经看到了如何从JSNative函数抛出异常的例子。只需调用JS\_ReportError，使用printf样式的参数，并返回false。

rc = system(cmd);

if (rc != 0) {

/\* Throw a JavaScript exception. \*/

JS\_ReportError(cx, "Command failed with exit code %d", rc);

return false;

}

这很像JavaScript语句抛出新错误(“退出代码命令失败”+ rc);。再次注意，调用JS\_ReportError不会引发c++异常。它只创建一个新的JavaScript错误对象，并将其作为当前挂起的异常存储在上下文中。应用程序还必须返回false。

一旦c++函数返回false, JavaScript引擎就开始展开JavaScript堆栈，寻找要执行的catch或block。但是SpiderMonkey的堆栈展开从未从堆栈中删除应用程序的c++函数。相反，SpiderMonkey只向应用程序返回false或nullptr，应用程序可以根据自己的选择处理错误，或者返回false，让它在堆栈中进一步传播。

在JSAPI常用语手册中可以找到更多抛出和捕获异常的例子。

**错误报告**

执行自定义错误报告器

报告错误时的待办事项

**自动处理未捕获异常**

在某些情况下，JS\_Compile\*、JS\_Call\*、JS\_Execute\*和JS\_Evaluate\*函数会自动将异常传递给错误报告器。在返回之前，每个函数都会检查当前JSContext中是否挂起了异常。如果是，那么它将检查该JSContext中的堆栈上是否有其他JavaScript脚本或函数。如果是，那么异常可能仍然被捕获，因此SpiderMonkey什么也不做，返回false，允许异常传播。但是如果JavaScript堆栈上什么都没有，那么未捕获异常将传递给错误报告器，并清除挂起的异常。

最基本的结果是，顶级应用程序代码可以设置一个错误报告器并开始调用JSAPI函数。它从不需要显式地处理未捕获的异常;将自动调用错误报告器。应用程序可以使用JSOPTION\_DONT\_REPORT\_UNCAUGHT选项禁用自动异常处理,但它必须处理未捕获的异常显式地调用JS\_IsExceptionPending JS\_GetPendingException JS\_ReportPendingException和/或JS\_ClearPendingException每当JSAPI nullptr或函数返回false。

**只捕捉不到的错误**

JSNative回调报告错误的另一种方法是:

if (!p) {

JS\_ReportOutOfMemory(cx);

return false;

}

这与JS\_ReportError的功能略有不同。

大多数错误，包括JS\_ReportError提出的错误，都表示为JavaScript异常，因此与JavaScript异常处理语言特性交互，try、catch，最后。然而，在某些情况下，我们不希望脚本能够捕获错误;我们希望脚本执行立即终止。如果系统在脚本执行过程中耗尽内存，我们不希望最后执行块，因为几乎所有脚本执行的操作都需要至少一点内存，而我们没有内存。如果一个脚本运行的时间太长，我们想要终止它，那么抛出一个异常是没有好处的——脚本可以捕获它并继续运行。

因此，JS\_ReportOutOfMemory(cx)不设置挂起异常。这是一个无法捕捉的错误。

如果SpiderMonkey耗尽内存，或者JSAPI回调返回false而没有异常挂起，这将被视为不可捕获的错误。除catch和最后的块被忽略外，JavaScript堆栈按正常方式展开。最近的JSAPI调用向应用程序返回false或nullptr。

一个无法捕获的错误使JSContext处于良好状态。它可以马上再次使用。就JSAPI而言，应用程序不需要做任何事情来从错误中“恢复”。(当然，如果错误是系统内存不足，那么这个问题仍然需要处理。)

下面是一些抛出不可捕获错误的示例代码。

/\* Call the error reporter, if any. This part is optional. \*/

JS\_ReportError(cx, "The server room is on fire!");

JS\_ReportPendingException(cx);

/\* Make sure the error is uncatchable. \*/

JS\_ClearPendingException(cx);

return false;