

윈도우 프로그래밍 테크닉

SpiderMonkey를 사용한 자바스크립트 임베딩

목차

목차.....	1
License.....	1
소개.....	1
연재 가이드.....	1
필자소개.....	2
필자 메모.....	2
Introduction.....	2
SpiderMonkey.....	3
Hello, World.....	4
런타임, 컨텍스트.....	6
타입 변환.....	7
자바스크립트 코드 실행시키기.....	9
C 함수를 자바스크립트에서 호출하기.....	10
자바스크립트 함수를 C에서 호출하기.....	12
에러 핸들링.....	13
도전 과제.....	16
참고자료.....	18

License

Copyright © 2007, 신영진

이 문서는 Creative Commons 라이선스를 따릅니다.

<http://creativecommons.org/licenses/by-nc-nd/2.0/kr>

소개

프로그램을 프로그래밍 하는 시대가 오고 있다. 이러한 시대 변화의 선두에는 스크립트 언어들이 있다. 이번 시간에는 SpiderMonkey라는 자바스크립트 엔진을 사용해서 C/C++ 프로그램에 자바스크립트 기능을 탑재하는 방법에 대해서 살펴본다.

연재 가이드

운영체제: 윈도우 2000/XP

개발도구: Visual Studio 2005

기초지식: C/C++, Win32 API, Javascript

응용분야: 스크립트 기능을 포함한 응용 프로그램

필자소개

신영진 pop@jiniya.net, <http://www.jiniya.net>

A의 아들. B의 동생. C의 친구. D의 후배. E의 선배. F의 사원. G의 직장 동료. H의 회원. I의 필자. J의 운영자. A, B, C, D, E, F, G, H, I, J에게 모두 미안한 요즘이다.

필자 메모

흔히 하는 속된 말로 "강한 놈이 살아 남는 게 아니라 살아남는 놈이 강하다"라는 이야기가 있다. 그렇다면 살아남기 위해서 가장 중요한 것은 무엇일까? 바로 변화에 대처하는 능력이다. 프로그램도 마찬가지다. 엑셀도, 워드도, 인터넷 익스플로러도 처음부터 시장 점유율이 높은 프로그램은 아니었다. 경쟁자들이 하나씩 없어질 때에도 꾸준히 업그레이드를 하고 시장의 요구 사항에 맞게 변화해 나갔기 때문에 지금의 위치에 있는 것이다.

프로그램이 이렇게 변화에 대처하기 쉽도록 만드는 기술 중의 하나가 스크립팅이다. 스크립트 언어를 포함시키면 유지보수 작업이 간단해지고, 기능 확장을 손쉽게 할 수 있다. SpiderMonkey 엔진을 사용해서 C/C++ 프로그램에 이러한 날개를 달아보자.

Introduction

UCC 광풍이 불고 있다. UCC 광고는 TV에서 시작해서, 인터넷 포털, 지하철에 이르기까지 없는 곳이 없다. 점심 시간의 잡담에도 빠지지 않고 등장하는 단어가 UCC다. UCC란 User Created Content의 약자로 사용자가 직접 제작한 콘텐츠를 말한다. 과거에는 막대한 자금을 투입해 제작한 미디어를 모두가 바라보는 시대였다. 라디오가 그랬고, TV도 그랬고, 영화, 음반도 그랬다. 하지만 요즘의 미디어는 이러한 방향성이 없다. 제작자가 소비자가 되기도 하고, 소비자가 제작자가 되기도 하는 것이다.

이런 UCC 열풍이 단지 동영상 분야에, 블로그에 국한되어 있다고 생각하는 것은 큰 오산이다. 느리긴 하지만 프로그래밍이란 분야에도 UCC 바람이 불어오고 있기 때문이다. 블리자드에서 제작하고, 블리자드코리아에서 서비스하고 있는 월드오브워크래프트(WoW)란 게임 속에서 그러한 현상을 쉽게 찾아볼 수 있다. WoW는 사용자가 프로그래밍을 해서 변경할 수 있는 두 가지 요소가 있다. 매크로와 UI가 그것이다. 매크로는 게임 내부에서 사용되는 프로그램으로 일련의 작업을 한번에 하도록 만드는 간단한 배치 파일 같은 것이다. UI는 Lua 스크립트로 WoW가 동작하는 외부 인터페이스를 정의하는 일을 한다. 중요한 것은 이런 것을 제작하는 사용자들이 많고, 그들을 중심으로 커뮤니티가 형성되고 있다는 점이다.

시대는 변하고 있다. 프로그램도 이제 고급 개발자 몇 명이 모여서 만들던 시대가 아니다. 최적화 프로그램을 생각해보자. 개발자 몇 명이 생각할 수 있는 최적화 방법엔 한계가 있다. 하지만 그것을 최적화에 사용되는 기본적인 함수를 가진 스크립트 기능을 포함한 프로그램으로 만들었다면 그 한계가 없어진다. 개발자들이 생각하지도 못했던 기상천외한 최적화 방법들을 담은 스크립트들이 나올 것이고, 커뮤니티도 만들어질 것이다. 이제 스크립팅은 더 이상 부가 기능이 아니다. 킬러 애플리케이션의 필수 기능이다.

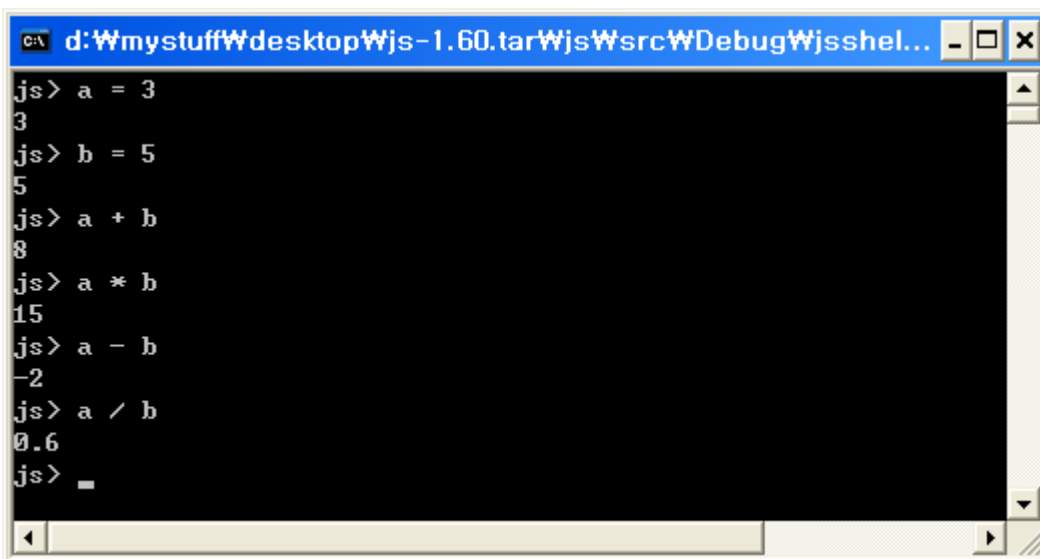
SpiderMonkey를 사용한 자바스크립트 임베딩

우리는 이번 시간에 SpiderMonkey라는 잘 만들어진 자바스크립트 엔진을 사용해서 C/C++에 자바스크립트를 임베딩 시키는 방법에 대해서 배울 것이다. 이 간단한 작업을 배우고 나면 자바스크립트를 C/C++에서 탑재하는 일이 얼마나 간단한지, 그렇게 함으로써 얻을 수 있는 확장성이 얼마나 큰지를 몸으로 느낄 수 있을 것이다. 이를 통해서 우리가 얻을 수 있는 확장성은 무궁무진 하다. 전적으로 여러분의 상상력에 달려있다.

SpiderMonkey

SpiderMonkey는 모질라(Mozilla)에서 나온 C/C++용 자바스크립트 엔진이다. 오픈 소스 프로젝트로 진행 중이기 때문에 누구나 소스 코드를 다운 받고, 컴파일해서 사용해 볼 수 있다. SpiderMonkey의 공식 페이지는 <http://www.mozilla.org/js/spidermonkey>이고, 프로젝트의 소스는 <http://ftp.mozilla.org/pub/mozilla.org/js/>에서 받을 수 있다. 이번 연재에 사용된 모든 소스 코드는 해당 페이지에서 다운 받을 수 있는 js-1.60.tar.gz에 기반을 두고 있다.

코드를 다운받은 다음 압축을 해제하면 폴더가 여러 개 생긴다. 읽기 전용 속성이 적용되어 있기 때문에 먼저 해당 속성을 해제해 준다. src 폴더로 들어가 보자. 방대한 분량의 소스코드에 놀랄 것이다. 하지만 주눅들 필요 없다. 이름있는 오픈 소스 프로젝트인 만큼 소스 코드 관리도 잘 되어 있기 때문이다. src 폴더의 js.mdp를 열면 프로젝트가 Visual Studio에 로드된다. 솔루션 다시 빌드를 하면 전체 프로젝트가 새로 빌드 된다. 간혹 라이브러리 링크가 잘못되는 경우가 있다. 이 때는 당황하지 말고 프로젝트 속성에 가서 각각 필요한 라이브러리를 연결해준다. js 프로젝트는 fdlibm 라이브러리가 필요하고, jshell 프로젝트는 js32 라이브러리가 필요하다. 컴파일이 완료되었다면 컴파일된 폴더로 이동해서 jshell을 실행시켜 보자. jshell은 자바스크립트를 테스트해 볼 수 있는 콘솔 프로그램이다. <화면 1>은 jshell에서 간단하게 사칙 연산을 수행해 본 화면이다.



```
C:\d:\Wmystuff\Wdesktop\Wjs-1.60.tar\Wjs\Wsrc\WDebug\Wjshell... - □ ×
js> a = 3
3
js> b = 5
5
js> a + b
8
js> a * b
15
js> a - b
-2
js> a / b
0.6
js> _
```

화면 1 jshell 실행 화면

Hello, World

"The C Programming Language" 이후로 새로운 프로그래밍 언어나 환경을 익힐 때 가장 먼저 작성하는 프로그램의 표준은 Hello, World가 되었다. Hello, World의 가장 큰 의미는 프로그램의 골격을 익힌다는 점과 실제로 동작하는 프로그램을 만들어 본다는데 있다. 이러한 의미를 가장 잘 살리기 위해서는 반드시 코드를 직접 입력해서 실행해 보는 것이 좋다.

<리스트 1>에 SpiderMonkey를 사용한 Hello, World 프로그램이 나와있다. 모르는 함수나 구조체에 당황하지 말고 전체적인 구조를 파악하는데 집중하도록 하자. 소스 코드를 컴파일하기 위해서는 빌드된 js32.dll 파일과 src 폴더에 있는 헤더들이 필요하다. 환경 구축이 힘들다면 이달의 디스크에 포함된 lib 폴더의 내용을 복사해서 사용하면 된다.

리스트 1 SpiderMonkey를 사용한 Hello, World 프로그램

```
#include <windows.h>

// 윈도우 환경
#define XP_WIN
#include "jsapi.h"

// 라이브러리 링크
#pragma comment(lib, "js32.lib")

// 자바스크립트에 제공해 줄 OutputDebugString에 대한 인터페이스 함수
JSBool
JAlert(JSContext *ctx, JSObject *obj, uintN argc, jsval *argv, jsval *rval)
{
    char *msg = NULL;
    char *boolMsg[2] = { "false\n", "true\n" };
    *rval = BOOLEAN_TO_JSVAL(FALSE);

    // 인자 개수가 1보다 작은 경우엔 바로 리턴
    if(argc < 1)
        return TRUE;

    // 문자열인 경우
    if(JSVAL_IS_STRING(argv[0]))
    {
        msg = JS_GetStringBytes(JS_ValueToString(ctx, argv[0]));
    }
    // 불(bool) 값인 경우
    else if(JSVAL_IS_BOOLEAN(argv[0]))
    {
        JSBool b = JSVAL_TO_BOOLEAN(argv[0]);
        msg = boolMsg[b];
    }

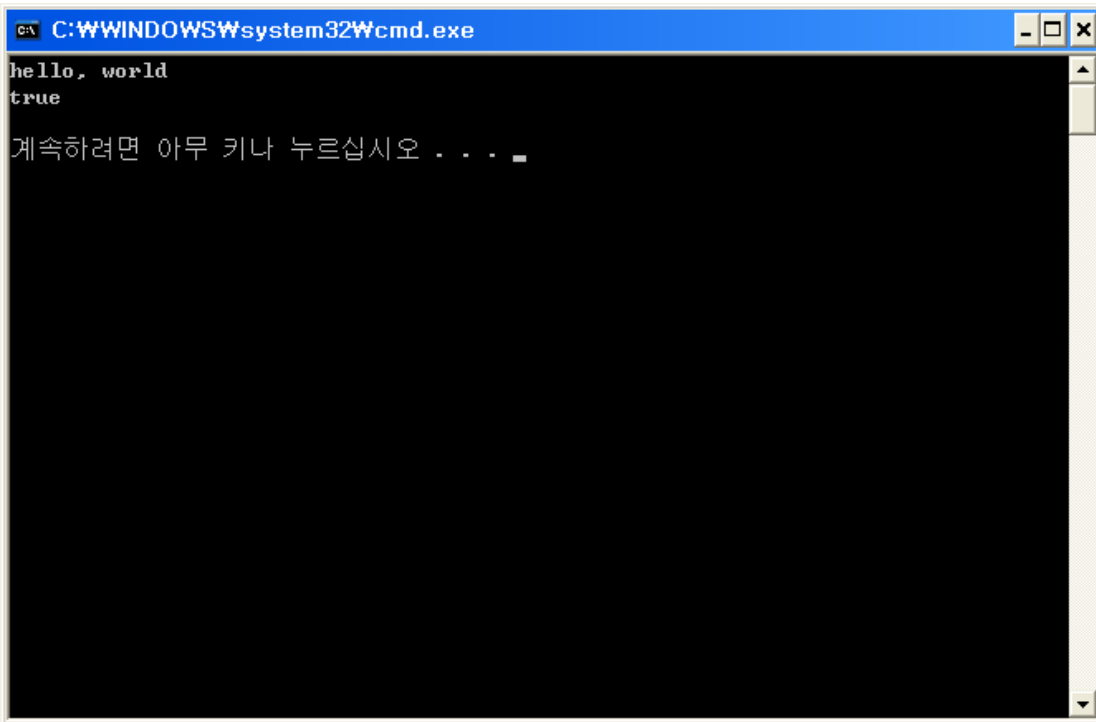
    if(msg)
    {
        // 메시지를 출력하고 리턴 값 설정
        puts(msg);
        *rval = BOOLEAN_TO_JSVAL(TRUE);
    }

    return TRUE;
}
```

SpiderMonkey를 사용한 자바스크립트 임베딩

```
}  
  
int _tmain(int argc, _TCHAR* argv[])  
{  
    JSRuntime *rt;  
    JSContext *ctx;  
    JSObject *global;  
  
    // 런타임 생성  
    rt = JS_NewRuntime(1024 * 1024);  
  
    // 컨텍스트 생성  
    ctx = JS_NewContext(rt, 4096);  
  
    // 글로벌 객체 생성  
    global = JS_NewObject(ctx, NULL, NULL, NULL);  
  
    // 글로벌 클래스 및 함수 초기화  
    JS_InitStandardClasses(ctx, global);  
  
    // alert 함수 정의  
    JS_DefineFunction(ctx, global, "alert", Jalert, 1, 0);  
  
    // 자바스크립트 코드 실행  
    JSBool ok;  
    jsval rval;  
    char *script = "var s = alert('hello, world')\n\n    s = alert(s);"  
  
    uintN size = strlen(script);  
    ok = JS_EvaluateScript(ctx, global, script, size, NULL, 0, &rval);  
  
    // 컨텍스트 및 런타임 파괴  
    JS_DestroyContext(ctx);  
    JS_DestroyRuntime(rt);  
    JS_ShutDown();  
  
    return 0;  
}
```

<화면 2>는 <리스트 1>의 코드를 컴파일해서 실행한 화면이다. <화면 2>와 같이 hello, world와 true가 출력된다면 정상적으로 동작한 것이다. script의 변수 내용을 이리저리 변경해 가면서 메시지를 바꿔서 출력해 보도록 하자.



화면 2 프로그램 실행 화면

런타임, 컨텍스트

SpiderMonkey를 사용하기 전에 가장 먼저 이해해야 할 것은 런타임(run time)과 컨텍스트(context), 객체(object)의 관계다. 런타임은 자바스크립트 코드를 수행하기 위해서 필요한 컨텍스트, 변수, 객체 등을 저장하기 위한 공간이다. 컨텍스트는 코드의 실행 상태를 저장하기 위해서 사용된다. 동시에 실행되는 코드는 반드시 별도의 컨텍스트에서 실행되어야 한다. 컨텍스트에서 스크립트가 실행되기 위해서 필요한 마지막 요소는 객체다. 객체는 스크립트가 실행되는 스코프를 의미한다고 이해하면 된다. 일반적으로 전역으로 실행되는 스크립트라면 제일 처음 생성한 global 객체와 함께 실행되면 된다. 이 객체들은 컨텍스트에 국한된 요소가 아니기 때문에 별도의 컨텍스트라 하더라도 같이 사용할 수 있다. 이러한 관계가 <그림 1>에 잘 나와있다.

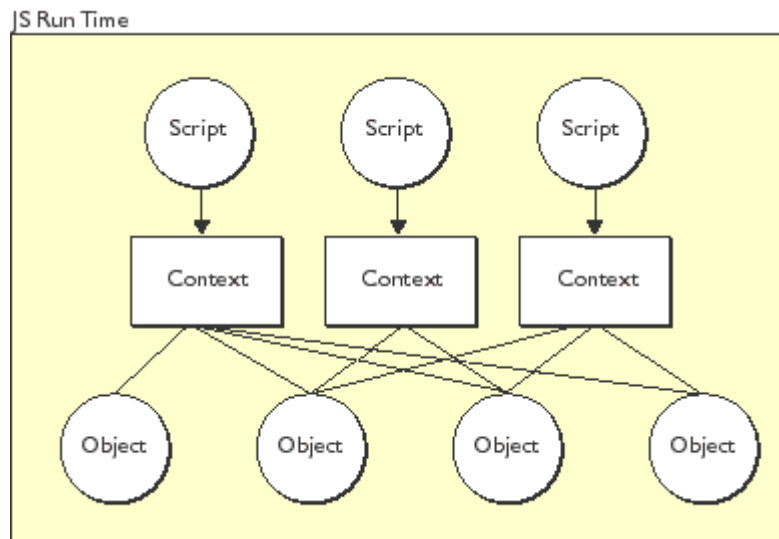


그림 1 런타임, 컨텍스트, 오브젝트의 관계

SpiderMonkey 엔진을 사용하기 위해서 가장 먼저 호출해야 하는 함수는 런타임을 생성 시키는 JS_NewRuntime이다. 이 함수의 인자는 가비지 컬렉터가 동작하기 위한 임계 메모리 양이다. <리스트 1>에서는 1024x1024를 넣었기 때문에 1MB 이상의 메모리를 사용하려고 하면 가비지 컬렉터가 동작한다. JS_NewRuntime의 리턴 값은 생성된 런타임의 포인터다. 런타임을 다 사용하고 더 이상 필요가 없는 경우에는 JS_DestroyRuntime 함수를 호출해서 런타임을 파괴시켜야 한다.

런타임 생성 다음으로 할 일은 코드가 수행될 컨텍스트를 생성하는 일이다. 컨텍스트 생성은 JS_NewContext 함수가 한다. JS_NewContext의 첫 번째 인자는 컨텍스트가 포함될 런타임의 포인터가, 두 번째 인자는 스택 청크의 사이즈다. 이 사이즈는 스택 크기를 의미하지 않기 때문에 크게 잡을 필요는 없다. 보통 8192를 넣어주면 된다. 컨텍스트도 런타임과 마찬가지로 더 이상 필요하지 않다면 JS_DestroyContext를 호출해서 파괴시켜야 한다.

끝으로 초기화의 마지막 단계는 전역 객체를 만들어주는 과정이다. 일반적으로 자바스크립트에서 사용하는 Math, String, Number 등의 객체는 JS_InitStandardClasses를 호출하면 만들어진다. JS_InitStandardClasses의 두 번째 인자는 해당 컨텍스트의 전역 객체를 지정한다. 이 객체는 JS_NewObject를 사용해서 만들면 된다.

타입 변환

뭐든지 var에 저장되고, 아무렇게나 인자로 넘겨도 예상대로 동작하는 자바스크립트는 느슨한 타입 체크를 하는 언어다. 하지만 이와는 달리 C/C++은 엄격한 타입 검사를 한다. 따라서 자바스크립트의 변수를 C/C++에서 사용하려면 적절한 타입 변환이 필요하다.

SpiderMonkey에서 자바스크립트와 인터페이스되는 모든 변수는 jsval 형태로 되어있다. jsval은 정수가 재정의 된 것으로 정수와 boolean 타입은 바로 저장되고 다른 타입인 경우에는 해당 데이터가 할당된 메모리를 가리키는 포인터가 된다. jsval을 C/C++에서 사용하는 데이터 타입으로 변환하기 위해서는 우선 jsval에 저장된 데이터가 어떤 형태인지를 <표 1>에 나와있는 매크로를 통해서 검사하고, 해당 데이터에 맞는 변환 함수를(<표 2> 참고) 사용해서 실제 저장된 데이터를 구한다. 반대로 C언어의 데이터를 jsval로 변환하기 위해서는 <표 3>과 <표 4>에 나와있는 함수와 매크로를 사용하면 된다. 추가적으로 JSString에서 C/C++에서 사용하는 char *로 변환을 위해서는 <표 5>에 나타난 함수들을 사용하면 된다. <리스트 2>에 간단한 함수와 매크로의 사용 예가 나와있다.

표 1 jsval의 타입을 검사하는 매크로

매크로 이름	기능
JSVAL_IS_BOOLEAN	jsval이 Boolean 형인지 검사한다.
JSVAL_IS_DOUBLE	jsval이 double 형인지 검사한다.
JSVAL_IS_GCTHING	jsval이 가비지 컬렉션의 대상인지 검사한다.

SpiderMonkey를 사용한 자바스크립트 임베딩

JSVAL_IS_INT	jsval이 int 형인지 검사한다.
JSVAL_IS_NULL	jsval이 null인지 검사한다.
JSVAL_IS_NUMBER	jsval이 Number 형인지 검사한다.
JSVAL_IS_OBJECT	jsval이 객체인지 검사한다.
JSVAL_IS_PRIMITIVE	jsval이 자바스크립트 원시 데이터 형인지 검사한다. 원시 데이터 형에는 undefined, null, boolean, 숫자, 문자가 포함된다.
JSVAL_IS_STRING	jsval이 JSString 형인지 검사한다.
JSVAL_IS_VOID	jsval이 void 형인지 검사한다.

표 2 jsval의 타입을 변환하는 함수

함수 이름	기능
JS_ValueToBoolean	jsval을 Boolean 형으로 변환한다.
JS_ValueToECMAInt32	jsval을 ECMA 표준에 맞는 32비트 정수 형으로 변환한다.
JS_ValueToECMAUint32	jsval을 ECMA 표준에 맞는 부호 없는 32비트 정수 형으로 변환한다.
JS_ValueToFunction	jsval을 Function 객체로 변환한다.
JS_ValueToInt32	jsval을 32비트 정수 형으로 변환한다.
JS_ValueToNumber	jsval을 double 형으로 변환한다.
JS_ValueToObject	jsval을 객체로 변환한다.
JS_ValueToString	jsval을 JSString으로 변환한다.
JS_ValueToUint16	jsval을 부호 없는 16비트 정수로 변환한다.

표 3 새로운 jsval 변수를 생성하는 함수

함수 이름	기능
JS_NewArrayObject	배열 객체를 생성한다.
JS_NewDoubleValue	double 변수를 생성한다.
JS_NewNumberValue	
JS_NewFunction	function 객체를 생성한다.
JS_NewObject	새로운 객체를 생성한다.
JS_NewString	JSString을 생성한다.
JS_NewUCString	유니코드 기반의 JSString을 생성한다.

표 4 새로운 jsval 변수를 생성하는 매크로

매크로 이름	기능
BOOLEAN_TO_JSVAL	boolean 값을 jsval 형태로 변환한다.
INT_TO_JSVAL	int 값을 jsval 형태로 변환한다.
STRING_TO_JSVAL	JSString을 jsval로 변환한다.
OBJECT_TO_JSVAL	객체를 jsval로 변환한다.

표 5 JSString을 다루는 함수들

SpiderMonkey를 사용한 자바스크립트 임베딩

함수이름	기능
JS_GetStringBytes	JSString에서 C형태의 char *을 구한다.
JS_GetStringChars	JSString에서 유니코드 문자열을 구한다.
JS_GetStringLength	JSString의 길이를 구한다.

리스트 2 변환 함수와 매크로의 사용 예

```
// val을 char *로 변환
if(JSVAL_IS_STRING(val))
    char *str = JS_GetStringBytes(JS_ValueToString(ctx, val));

// val을 double로 변환
if(JSVAL_IS_NUMBER(val))
{
    double d;
    JS_ValueToNumber(ctx, val, &d);
}

// 새로운 hello 문자열을 생성
jsval val = STRING_TO_JSVAL(JS_NewString(ctx, "hello", 5));
```

자바스크립트 코드 실행시키기

SpiderMonkey에서는 자바스크립트 코드를 수행시키기 위해서 크게 두 종류의 함수를 제공한다. <오류! 참조 원본을 찾을 수 없습니다.>에서 사용한 JS_EvaluateScript가 좀 더 쓰기 편한 함수다. 이 함수는 별도의 스크립트 객체 생성 필요 없이 문자열로 된 스크립트를 바로 실행시키는 기능을 한다. 인자로 넘겨주는 filename과 lineno는 스크립트 실행 중에 오류가 발생했을 때 보고 하기 위해 사용된다. obj는 스크립트와 관련된 객체다. 전역으로 실행하는 경우라면 제일 처음 생성한 global 객체를 전달해주면 된다.

```
JSBool JS_EvaluateScript( JSContext *cx           // 컨텍스트
                        , JSObject *obj         // 스크립트와 연관된 오브젝트
                        , const char *bytes     // 스크립트
                        , uintN length         // 스크립트 길이
                        , const char *filename // 에러 출력 시 파일명
                        , uintN lineno        // 에러 출력 시 라인 번호
                        , jsval *rval );      // 리턴 값
```

JS_EvaluateScript 보다 조금 원시적인 함수가 JS_ExecuteScript다. JS_ExecuteScript는 직접 스크립트 객체를 생성한 경우에만 사용할 수 있다. 실행시킬 스크립트 객체를 script 인자로 넘겨주면 된다.

```
JSBool JS_ExecuteScript( JSContext *cx // 컨텍스트
                       , JSObject *obj // 스크립트 실행 오브젝트
                       , JSScript *script // 스크립트 객체
                       , jsval *rval ) // 리턴 값
```

SpiderMonkey를 사용한 자바스크립트 임베딩

스크립트 객체는 아래에 나와있는 JS_CompileScript 함수를 사용해서 만든다. JS_CompileScript 함수로 들어가는 인자의 의미는 JS_EvaluateScript의 인자와 같은 의미를 가진다. 성공한 경우에는 생성된 스크립트 객체가 반환된다. 스크립트 객체가 더 이상 필요하지 않다면 JS_DestroyScript 함수를 호출해서 스크립트 객체를 제거해 준다.

JSScript *

```
JS_CompileScript( JSContext *cx           // 컨텍스트
                 , JSObject *obj         // 오브젝트
                 , const char *bytes     // 스크립트 내용
                 , size_t length         // 스크립트 길이
                 , const char *filename  // 파일 명
                 , uintN lineno )       // 줄 번호
```

<리스트 3>에 JS_ExecuteScript를 사용하는 예제가 나와있다. <리스트 1>의 JS_EvaluateScript 코드 부분을 대체시킨 것이다. 실제로 JS_EvaluateScript는 내부적으로 JS_CompileScript와 JS_ExecuteScript의 조합으로 이루어져 있다.

리스트 3 JS_ExecuteScript를 사용 예제

```
JSScript *s = JS_CompileScript(ctx, global, script, size, NULL, 0);
JS_ExecuteScript(ctx, global, s, &rval);
JS_DestroyScript(ctx, s);
```

C 함수를 자바스크립트에서 호출하기

순수 자바스크립트는 사실 계산하는 기계에 불과하다. 실제로 시스템과 의사소통을 하도록 만들기 위해서는 C언어에서 함수를 만들어서 자바스크립트에서 사용할 수 있도록 만들어 주어야 한다.

JS_DefineFunction은 새로운 함수를 자바스크립트에서 사용할 수 있도록 정의하는 API다. 원형은 아래와 같다. 컨텍스트에는 수행될 자바스크립트의 컨텍스트 정보를, obj로는 해당 함수가 수행될 때 this를 통해 참조할 객체 포인터를 전달해 주면 된다. 전역 함수인 경우에는 가장 먼저 생성했던 global 객체를 전달하면 된다. flags값은 일반적으로 0을 전달하면 된다.

```
JSFunction * JS_DefineFunction( JSContext *cx // 컨텍스트
                               , JSObject *obj // this 포인터
                               , const char *name // 함수 이름
                               , JSNative call // 실제 함수 포인터
                               , uintN nargs // 인자 개수
                               , uintN flags ); // 함수 특성
```

한번에 함수를 여러 개 등록하는 API로 JS_DefineFunctions가 있다. 원형은 아래와 같고 사용 방

SpiderMonkey를 사용한 자바스크립트 임베딩

법은 JS_DefineFunction과 유사하다. 정의할 함수 목록에 대한 정보를 fs로 전달해 준다.

```
JSTool JS_DefineFunctions( JSContext *cx // 컨텍스트
                          , JSObject *obj // this 포인터
                          , JSFunctionSpec *fs ); // 정의할 함수 목록
```

JFunctionSpec 구조체의 원형은 아래와 같다. 들어가는 정보는 JS_DefineFunction에 전달하는 정보와 동일한 의미를 가진다.

```
struct JSFunctionSpec {
    const char *name; // 함수 이름
    JSNative call; // 함수 포인터
    uint8 nargs; // 인자 개수
    uint8 flags; // 함수 특성
    uint16 extra; // 추후 기능을 위해 예약됨
};
```

실제 호출되는 C함수인 JSNative는 아래와 같이 정의되어 있다. argv로 인자가 넘어오고, rval에 리턴 값을 전달하면 된다. 함수가 성공한 경우에는 JS_TRUE를 리턴 하고, 실패한 경우에는 JS_FALSE를 리턴 해야 한다. JS_FALSE를 리턴 하면 엔진은 현재 진행중인 스크립트를 중지한다.

```
typedef JSTool (*JSNative)( JSContext *cx // 컨텍스트
                            , JSObject *obj // 함수가 수행될 때의 this 포인터
                            , unsigned int argc // 인자 개수
                            , jsval *argv // 인자
                            , jsval *rval ) // 함수 리턴 값
```

리스트 4 DeleteFile에 대한 인터페이스 함수

```
JSTool
JDeleteFile(JSContext *ctx, JSObject *obj, uintN argc, jsval *argv, jsval *rval)
{
    *rval = BOOLEAN_TO_JSVAL(FALSE);

    // 인자 개수가 작은 경우는 바로 리턴
    if(argc < 1)
        return JS_TRUE;

    // 인자가 문자열인 경우만 처리한다
    if(JSVAL_IS_STRING(argv[0]))
    {
        // 문자열을 변환하고 실제 DeleteFile 호출한다
        LPCWSTR path = (LPCWSTR) JS_GetStringChars(JS_ValueToString(ctx, argv[0]));
        BOOL ret = DeleteFileW(path);

        // 리턴 값을 설정한다
        *rval = BOOLEAN_TO_JSVAL(ret);
    }
}
```

```
}  
    return JS_TRUE;  
}  
  
// 함수를 정의 한다  
JS_DefineFunction(ctx, global, "DeleteFile", JDeleteFile, 1, 0);  
  
// 스크립트를 실행한다  
char *script = "DeleteFile('c:\\\\a.txt')";  
uintN size = strlen(script);  
JSVal rval;  
JS_EvaluateScript(ctx, global, script, size, NULL, 0, &rval);
```

자바스크립트 함수를 C에서 호출하기

C함수 인터페이스를 자바스크립트에 제공하는 것과 마찬가지로 자바스크립트의 함수를 사용하는 것도 간단한 함수로 되어있다. 여기에 사용되는 함수로는 JS_CallFunction, JS_CallFunctionName, JS_CallFunctionValue가 있다.

JS_CallFunction은 가장 원시적인 함수 호출 API다. 원형은 아래와 같다. 호출하고 싶은 함수 객체를 obj로 전달해주면 된다. 컨텍스트와 인자 정보는 나머지 인자들을 통해서 전달한다. 함수 호출 결과는 rval을 통해서 리턴 된다. JS_CallFunction의 리턴 값은 함수 호출이 성공했는지, 실패했는지를 나타낸다.

```
JSBool JS_CallFunction( JSContext *cx // 컨텍스트  
                        , JSObject *obj // this 포인터  
                        , JSFunction *fun // 함수 객체  
                        , uintN argc, jsval *argv, jsval *rval); // 인자 개수, 인자, 리턴 값
```

JS_CallFunctionName은 JS_CallFunction과 하는 일은 동일하다. 단지 차이점은 호출할 함수 객체를 직접 전달하는 것이 아니라, 함수 이름을 전달하면 해당하는 함수를 수행해 준다는 점이다. 함수 원형은 아래와 같다.

```
JSBool JS_CallFunctionName( JSContext *cx // 컨텍스트  
                            , JSObject *obj // this 포인터  
                            , const char *name // 함수 이름  
                            , uintN argc, jsval *argv, jsval *rval ); // 인자 개수, 인자, 리턴 값
```

마지막 함수인 JS_CallFunctionValue는 인자로 넘어온 함수를 호출할 때 사용하면 편리한 함수다. 함수 정보로 함수 객체를 저장하고 있는 jsval을 넘겨주기 때문이다. C함수에서 자바스크립트 함수를 콜백으로 사용하는 경우에 넘어온 인자가 함수 객체이지만 확인한 다음, 이 함수를 사용하면 해당 함수를 바로 호출할 수 있다.

```
JSBool JS_CallFunctionValue(JSContext *cx // 컨텍스트
```

SpiderMonkey를 사용한 자바스크립트 임베딩

```
, JSObject *obj // this 포인터
, jsval fval // 함수 객체를 저장하고 있는 jsval 변수
, uintN argc, jsval *argv, jsval *rval); // 인자 개수, 인자, 리턴 값
```

리스트 5 자바스크립트의 main 함수를 호출하는 코드

```
char *script = "\n\
function main() \n\
{ \n\
    alert('hello, world')\n\
} \n\
";

uintN size = strlen(script);
ok = JS_EvaluateScript(ctx, global, script, size, NULL, 0, &rval);
JS_CallFunctionName(ctx, global, "main", 0, NULL, &rval);
```

에러 핸들링

스크립트 기능을 프로그램에 포함시킬 때 가장 신경 써야 하는 부분은 스크립트의 오류 처리이다. 스크립트 파일은 언제나 손상될 수 있고, 사용자는 언제나 잘못된 스크립트를 작성한다고 생각하는 게 정신 건강에 이롭다.

SpiderMonkey에는 이러한 오류 처리를 위해서 콜백 함수를 사용한다. 컨텍스트의 오류 처리 함수를 콜백으로 등록해두면 스크립트를 수행하다 오류가 난 경우에 해당 콜백 함수를 호출해 준다. 이렇게 오류처리 콜백을 등록하는 함수가 JS_SetErrorReporter 함수다. 원형은 아래와 같다. cx에 오류처리 핸들러를 등록할 컨텍스트를, er로 오류처리 콜백 함수를 전달해주면 된다. 리턴 값은 이전에 설정된 오류처리 콜백 함수다.

```
JSErrorReporter JS_SetErrorReporter(JSContext *cx, JSErrorReporter er);
```

```
typedef void
```

```
(* JS_DLL_CALLBACK JSErrorReporter)(JSContext *cx, const char *message, JSErrorReport *report);
```

콜백으로 넘어가는 JSErrorReporter는 위와 같이 정의되어 있다. cx에는 오류가 발생한 컨텍스트가, message로 오류 메시지가, report에는 오류와 관계된 자세한 정보가 넘어온다. JSErrorReporter에는 아래와 같은 내용이 들어있다. tokenptr은 linebuf의 내용 중에 어떤 토큰에서 에러가 발생했는지를 가리키고 있다. 따라서 tokenptr과 linebuf의 차이가 오류가 발생한 위치의 열 번호가 된다. JSErrorReport의 falgs 값은 <표 6>에 나와있는 값이 조합해서 사용된다.

```
struct JSErrorReport {
    const char    *filename;    /* 파일 명 */
    uintN        lineno;       /* 줄 번호 */
    const char    *linebuf;     /* 오류가 발생한 줄 내용 */
    const char    *tokenptr;    /* 오류가 발생한 토큰 */
    const jschar  *uclinebuf;   /* 유니코드 linebuf */
};
```

SpiderMonkey를 사용한 자바스크립트 임베딩

```
const jschar *uctokenptr; /* 유니코드 tokenptr */
uintN flags; /* 에러, 경고 같은 플래그 */
uintN errorNumber; /* 에러 번호 */
const jschar *ucmessage; /* 에러 메시지 */
const jschar **messageArgs; /* 에러 메시지 인자 */
};
```

표 6 에러 플래그 값

이름	값	의미
JSREPORT_ERROR	0	기본 값
JSREPORT_WARNING	1	JS_ReportWarning에 의해 보고된 경우
JSREPORT_EXCEPTION	2	예외가 던져진 경우
JSREPORT_STRICT	4	strict 옵션 때문에 발생한 경고

JS_SetErrorReporter 함수를 사용해서 오류 처리를 하고 있는 프로그램이 <리스트 6>에 나와있다. <리스트 1> 프로그램에서 alert 함수를 등록하는 부분만 제거하고 오류처리 콜백 함수만 추가한 것이다. JError 함수에서 오류 내용을 어떤 식으로 출력하는지 살펴보자. 프로그램의 실행 화면이 <화면 3>에 나와있다.

리스트 6 jserr 소스 코드

```
#include <windows.h>

// 윈도우 환경
#define XP_WIN
#include "jsapi.h"

// 라이브러리 링크
#pragma comment(lib, "js32.lib")

void
JError(JSContext *ctx, LPCSTR msg, JSErrorReport *report)
{
    printf("자바스크립트 오류\n");
    printf("종류:");

    if(JSREPORT_IS_WARNING(report->flags))
        printf(" WARNING");
    if(JSREPORT_IS_STRICT(report->flags))
        printf(" STRICT");
    if(JSREPORT_IS_EXCEPTION(report->flags))
        printf(" EXCEPTION");

    printf("\n파일 명: %s\n", report->filename ? report->filename : "알수 없음");
    printf("위치: %d, %d\n", report->lineno, report->tokenptr - report->linebuf);
    printf("%s\n", msg);
}

int _tmain(int argc, _TCHAR* argv[])
{
    JSRuntime *rt;
```

SpiderMonkey를 사용한 자바스크립트 임베딩

```
JSContext *ctx;
JSObject *global;

// 런타임 생성
rt = JS_NewRuntime(1024 * 1024);

// 컨텍스트 생성
ctx = JS_NewContext(rt, 4096);

// 글로벌 객체 생성
global = JS_NewObject(ctx, NULL, NULL, NULL);

// 글로벌 클래스 및 함수 초기화
JS_InitStandardClasses(ctx, global);

// 에러 리포터 설정
JS_SetErrorReporter(ctx, JError);

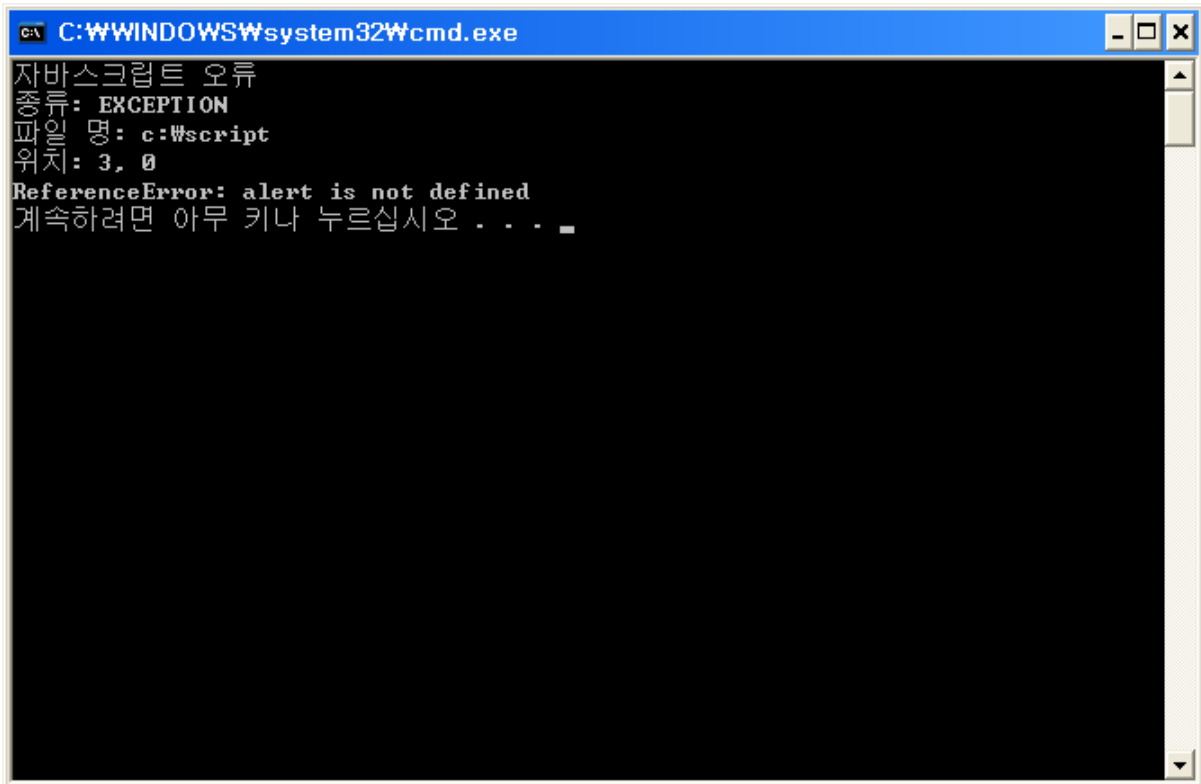
// 자바스크립트 코드 실행
JSBool ok;
jsval rval;

char *script = "\
    function main() \n\
    { \n\
        alert('hello, world')\n\
    } \n\
";

uintN size = strlen(script);
ok = JS_EvaluateScript(ctx, global, script, size, "c:\\script", 1, &rval);
JS_CallFunctionName(ctx, global, "main", 0, NULL, &rval);

// 컨텍스트 및 런타임 파괴
JS_DestroyContext(ctx);
JS_DestroyRuntime(rt);
JS_ShutDown();

return 0;
}
```



화면 3 jserr 실행 화면

도전 과제

GDI와 자바스크립트를 접목해서 간단한 드로잉 기계를 만들어 보자. 이 기계는 스택을 기반으로 그림을 그린다. PushRect, PushCircle, PushTriangle, PushText 등의 함수로 그리고자 하는 객체를 스택에 집어넣고, Draw를 호출하면 그림이 그려진다. Clear는 스택을 비우는 함수이고, Pop은 가장 마지막에 추가된 객체를 반환하는 함수다. Wait는 s초만큼 대기를 하는 기능을 한다. 함수 구현을 다 했으면 드로잉 스크립트를 만들어 보자. Wait 기능이 있기 때문에 애니메이션도 만들 수 있다. 그리기 기능이 빈약하다고 느껴진다면 함수를 더 추가해 보자.

PushRect(x, y, w, h, c) // x, y: 좌표, w: 너비, h: 높이, c: 색깔

PushCircle(x, y, r, c) // x, y: 좌표, r: 반지름, c: 색깔

PushTriangle(x, y, d, c) // x, y: 좌표, d: 길이, c: 색깔

PushText(x, y, str, c) // x, y: 좌표, str: 문자열, c: 색깔

Pop() // 마지막에 추가된 객체 제거. 객체가 없으면 아무 일도 하지 않음

Draw() // 스택 내용을 화면에 그림

Clear() // 스택 내용을 비움

Wait(s) // s: 대기 시간(초 단위)

참고자료

SpiderMonkey API Reference

http://developer.mozilla.org/en/docs/JSAPI_Reference

SpiderMonkey Embedder's guide

http://developer.mozilla.org/en/docs/JavaScript_C_Engine_Embedder%27s_Guide

SpiderMonkey Embedding Tutorial

<http://www.mozilla.org/js/spidermonkey/tutorial.html>