SpiderMonkey Internals

> This document is talking about partially obsolete things.

Design walk-through

At heart, SpiderMonkey is a fast interpreter that runs an untyped bytecode and operates on values of type `JS::Value`—type-tagged values that represent the full range of JavaScript values. In addition to the interpreter, SpiderMonkey contains a Just-In-Time (JIT) compiler, a garbage collector, code implementing the basic behavior of JavaScript values, a standard library implementing ECMA 262-5.1 §15 with various extensions, and a few public APIs.

Interpreter

Like many portable interpreters, SpiderMonkey's interpreter is mainly a single, tremendously long function that steps through the bytecode one instruction at a time, using a `switch` statement (or faster alternative, depending on the compiler) to jump to the appropriate chunk of code for the current instruction. A JS-to-JS function call pushes a JavaScript stack frame without growing the C stack. But since JS-to-C-to-JS call stacks are common, the interpreter is reentrant.

Some SpiderMonkey bytecode operations have many special cases, depending on the type of their arguments. Common cases are inlined in the interpreter loop, breaking any abstractions that stand in the way. So optimizations such as dense arrays and the property cache are, alas, *not* transparently tucked away in the `jsarray.*` and `jsobj.*` files. Both guest-star in `jsinterp.cpp` (to thunderous applause from Firefox users).

All state associated with an interpreter instance is passed through formal parameters to the interpreter entry point; most implicit state is collected in a type named `JSContext`. Therefore, almost all functions in SpiderMonkey, API or not, take a `JSContext` pointer as their first argument.

Compiler

The compiler consumes JavaScript source code and produces a *script* which contains bytecode, source annotations, and a pool of string, number, and identifier literals. The script also contains objects, including any functions defined in the source code, each of which has its own, nested script.

The compiler consists of: a random-logic rather than table-driven lexical scanner, a recursive-descent parser that produces an AST, and a tree-walking code generator. Semantic and lexical feedback are used to disambiguate hard cases such as missing semicolons, assignable expressions ("lvalues" in C parlance), and whether `/` is the division symbol or the start of a regular expression. The compiler attempts no error recovery; it bails out on the first error. The emitter does some constant folding and a few codegen optimizations; about the fanciest thing it does is to attach source notes to the script for the decompiler's benefit.

The decompiler implements `Function.toSource()`, which reconstructs a function's source code. It translates postfix bytecode into infix source by consulting a separate byte-sized code, called *source notes*, to disambiguate bytecodes that result from more than one grammatical production.

Garbage collector

The GC is a mark-and-sweep, non-conservative (exact) collector. It is used to hold JS objects and string descriptors (JSString), but not string bytes. It runs automatically only when maxbytes (as passed to `JS_NewRuntime`) bytes of GC things have been allocated and another thing-allocation request is made. JS API users should call `JS_GC` or `JS_MaybeGC` between script executions or from the operation callback, as often as necessary.

Because the GC is exact, C/C++ applications must ensure that all live objects, strings, and numbers are GC-reachable.

JavaScript values

The type `JS::Value` represents a JavaScript value.

The representation is 64 bits and uses NaN-boxing on all platforms, although the exact NaN-boxing format depends on the platform. NaN-boxing is a technique based on the fact that in IEEE-754 there are 2**53-2 different bit patterns that all represent NaN. Hence, we can encode any floating-point value as a C++ `double` (noting that JavaScript NaN must be represented as one canonical NaN format). Other values are encoded as a value and a type tag:

- On x86, ARM, and similar 32-bit platforms, we use what we call "nunboxing", in which non-`double` values are a 32-bit type tag and a 32-bit payload, which is normally either a pointer or a signed 32-bit integer. There are a few special values: `NullValue()`, `UndefinedValue()`, `TrueValue()` and `FalseValue()`.
- On x64 and similar 64-bit platforms, pointers are longer than 32 bits, so we can't use the nunboxing format. Instead, we use "punboxing", which has 17 bits of tag and 47 bits of payload.

Only JIT code really depends on the layout--everything else in the engine interacts with values through functions like `val.isDouble()`. Most parts of the JIT also avoid depending directly on the layout: the files `PunboxAssembler.h` and `NunboxAssembler.h` are used to generate native code that depends on the value layout.

Objects consist of a possibly shared structural description, called the map or scope; and unshared property values in a vector, called the slots. Each property has an id, either a nonnegative integer or an atom (unique string), with the same tagged-pointer encoding as a `jsval`.

The atom manager consists of a hash table associating strings uniquely with scanner/parser information such as keyword type, index in script or function literal pool, etc. Atoms play three roles: as literals referred to by unaligned 16-bit immediate bytecode operands, as unique string descriptors for efficient property name hashing, and as members of the root GC set for exact GC.

Standard library

The methods for arrays, booleans, dates, functions, numbers, and strings are implemented using the JS API. Most are `JSFastNative`s. Most string methods are customized to accept a primitive string as the `this` argument. (Otherwise, SpiderMonkey converts primitive values to objects before invoking their methods, per ECMA 262-3 §11.2.1.)

Error handling

SpiderMonkey has two interdependent error-handling systems: JavaScript exceptions (which are *not* implemented with, or even compatible with, any kind of native C/C++ exception handling) and error reporting. In general, both functions inside SpiderMonkey and JSAPI callback functions signal errors by calling `JS_ReportError` or one of its variants, or `JS_SetPendingException`, and returning `JS_FALSE` or `NULL`.

Public APIs

The public C/C++ interface, called the JSAPI, is in most places a thin (but source-compatible across versions) layer over the implementation. See the JSAPI User Guide. There is an additional public API for JavaScript debuggers, JSDBGAPI, but `js/jsd/jsdebug.h` might be a better API for debuggers. Another API, JSXDRAPI, provides serialization for JavaScript scripts. (XUL Fastload uses this.)

Just-In-Time compiler

SpiderMonkey contains a baseline compiler as first tier. A second tier JIT, code-named *IonMonkey* was enabled in Firefox 18. IonMonkey is an optimizing compiler.

Self-hosting of built-in functions in JS

Starting with Firefox 17, SpiderMonkey has the ability to implement built-in functions in self-hosted JS code. This code is compiled in a special compilation mode that gives it access to functionality that's not normally exposed to JS code, but that's required for safe and specification-conformant implementation of built-in functions.

All self-hosted code lives in `.js` files under `builtin/`. For details on implementing self-hosted built-ins, see self-hosting.

File walkthrough

jsapi.cpp, jsapi.h

The public API to be used by almost all client code.

jspubtd.h, jsprvtd.h

These files exist to group struct and scalar typedefs so they can be used everywhere without dragging in struct definitions from N different files. The `jspubtd.h` file contains public typedefs, and is included automatically when needed. The `jsprvtd.h` file contains private typedefs and is included by various .h files that need type names, but not type sizes or declarations.

jsdbgapi.cpp, jsdbgapi.h

The debugging API. Provided so far:

Traps, with which breakpoints, single-stepping, step over, step out, and so on can be implemented. The debugger will have to consult jsopcode.def on its own to figure out where to plant trap instructions to implement functions like step out, but a future jsdbgapi.h will provide convenience interfaces to do these things. At most one trap per bytecode can be set. When a script (`JSScript`) is destroyed, all traps set in its bytecode are cleared.

Watchpoints, for intercepting set operations on properties and running a debugger-supplied function that receives the old value and a pointer to the new one, which it can use to modify the new value being set.

Line number to PC and back mapping functions. The line-to-PC direction "rounds" toward the next bytecode generated from a line greater than or equal to the input line, and may return the PC of a for-loop update part, if given the line number of the loop body's closing brace. Any line after the last one in a script or function maps to a PC one byte beyond the last bytecode in the script. An example, from perfect.js:

```
1 function perfect(n) {
2   print("The perfect numbers up to " + n + " are:");
```

```
  // We build sumOfDivisors[i] to hold a string expression for
  // the sum of the divisors of i, excluding i itself.
  var sumOfDivisors = new ExprArray(n + 1, 1);
  for (var divisor = 2; divisor <= n; divisor++) {
    for (var j = divisor + divisor; j <= n; j += divisor) {
      sumOfDivisors[j] += " + " + divisor;
    }
    // At this point everything up to 'divisor' has its sumOfDivisc
    // expression calculated, so we can determine whether it's perf
    // already by evaluating.
    if (eval(sumOfDivisors[divisor]) == divisor) {
      print("" + divisor + " = " + sumOfDivisors[divisor]);
    }
  }
  delete sumOfDivisors;
  print("That's all.");
}
```

The line number to PC and back mappings can be tested using the js program with the following script:

```
load("perfect.js");
print(perfect);
dis(perfect);
print();
for (var ln = 0; ln <= 40; ln++) {
    var pc = line2pc(perfect, ln);
    var ln2 = pc2line(perfect, pc);
    print("\tline " + ln + " => pc " + pc + " => line " + ln2);
}
```

The result of the for loop over lines 0 to 40 inclusive is:

```
line 0 => pc 0 => line 16
line 1 => pc 0 => line 16
line 2 => pc 0 => line 16
line 3 => pc 0 => line 16
line 4 => pc 0 => line 16
```

```
line 5 => pc 0 => line 16
line 6 => pc 0 => line 16
line 7 => pc 0 => line 16
line 8 => pc 0 => line 16
line 9 => pc 0 => line 16
line 10 => pc 0 => line 16
line 11 => pc 0 => line 16
line 12 => pc 0 => line 16
line 13 => pc 0 => line 16
line 14 => pc 0 => line 16
line 15 => pc 0 => line 16
line 16 => pc 0 => line 16
line 17 => pc 19 => line 20
line 18 => pc 19 => line 20
line 19 => pc 19 => line 20
line 20 => pc 19 => line 20
line 21 => pc 36 => line 21
line 22 => pc 53 => line 22
line 23 => pc 74 => line 23
line 24 => pc 92 => line 22
line 25 => pc 106 => line 28
line 26 => pc 106 => line 28
line 27 => pc 106 => line 28
line 28 => pc 106 => line 28
line 29 => pc 127 => line 29
line 30 => pc 154 => line 21
line 31 => pc 154 => line 21
line 32 => pc 161 => line 32
line 33 => pc 172 => line 33
line 34 => pc 172 => line 33
line 35 => pc 172 => line 33
line 36 => pc 172 => line 33
line 37 => pc 172 => line 33
line 38 => pc 172 => line 33
line 39 => pc 172 => line 33
line 40 => pc 172 => line 33
```

jsconfig.h

Various configuration macros defined as 0 or 1 depending on how `JS_VERSION` is defined (as 10 for JavaScript 1.0, 11 for JavaScript 1.1, etc.). Not all macros are tested around related code yet. In particular, JS 1.0 support is missing from SpiderMonkey.

js.cpp, jsshell.msg

The "JS shell", a simple interpreter program that uses the JS API and more than a few internal interfaces (some of these internal interfaces could be replaced by `jsapi.h` calls). The js program built from this source provides a test vehicle for evaluating scripts and calling functions, trying out new debugger primitives, etc.

A look at the places where `jsshell.msg` is used in `js.cpp` shows how error messages can be handled in JSAPI applications. These messages can be localized at compile time by replacing the `.msg` file; or, with a little modification to the source, at run time.

More information on the JavaScript shell.

js.msg

SpiderMonkey error messages.

jsarray.*, jsbool.*, jsdate.*, jsfun.*, jsmath.*, jsnum.*, jsstr.*

These file pairs implement the standard classes and (where they exist) their underlying primitive types. They have similar structure, generally starting with class definitions and continuing with internal constructors, finalizers, and helper functions.

jsobj.*, jsscope.*

These two pairs declare and implement the JS object system. All of the following happen here:

- creating objects by class and prototype, and finalizing objects;
- defining, looking up, getting, setting, and deleting properties;
- creating and destroying properties and binding names to them.

The details of a native object's map (scope) are mostly hidden in `jsscope.[ch]`.

jsatom.cpp, jsatom.h

The atom manager. Contains well-known string constants, their atoms, the global atom hash table and related state, the js_Atomize() function that turns a counted string of bytes into an atom, and literal pool (`JSAtomMap`) methods.

jsarena.cpp, jsarena.h

Last-In-First-Out allocation macros that amortize malloc costs and allow for en-masse freeing. See the paper mentioned in `jsarena.h`'s major comment.

jsgc.cpp, jsgc.h

The garbage collector and tracing routines.

jsinterp.*, jscntxt.*, jsinvoke.cpp

The bytecode interpreter, and related functions such as Call and AllocStack, live in *jsinterp.cpp*. The JSContext constructor and destructor are factored out into *jscntxt.cpp* for minimal linking when the compiler part of JS is split from the interpreter part into a separate program.

`jsinvoke.cpp` is a build hack used on some platforms to build `js_Interpret` under different compiler options from the rest of `jsinterp.cpp`.

jsemit.*, jsopcode.tbl, jsopcode.*, jsparse.*, jsscan.*, jsscript.*

Compiler and decompiler modules. The *jsopcode.tbl* file is a C preprocessor source that defines almost everything there is to know about JS bytecodes. See its major comment for how to use it. For now, a debugger will use it and its dependents such as *jsopcode.h* directly, but over time we intend to extend *jsdbgapi.h* to hide uninteresting details and provide conveniences. The code generator is split across paragraphs of code in *jsparse.cpp*, and the utility methods called on `JSCodeGenerator` appear in *jsemit.cpp*. Source notes generated by *jsparse.cpp* and *jsemit.cpp* are used in *jsscript.cpp* to map line number to program counter and back.

jstypes.h

Fundamental representation types and utility macros. This file alone among all .h files in SpiderMonkey must be included first by .cpp files. It is not nested in .h files, as other prerequisite .h files generally are, since it is also a direct dependency of most .cpp files and would be over-included if nested in addition to being directly included.

jsbit.h, jslog2.cpp

Bit-twiddling routines. Most of the work here is selectively enabling compiler-specific intrinsics such as GCC's `__builtin_ctz`, which is useful in calculating base-2 logarithms of integers.

jsutil.cpp, jsutil.h

The `JS_ASSERT` macro is used throughout the source as a proof device to make invariants and preconditions clear to the reader, and to hold the line during maintenance and evolution against regressions or violations of assumptions that it would be too expensive to test unconditionally at run-time. Certain assertions are followed by run-time tests that cope with assertion failure, but only where I'm too smart or paranoid to believe the assertion will never fail...

jsclist.h

Doubly-linked circular list struct and macros.

jscpucfg.cpp

This standalone program generates *jscpucfg.h*, a header file containing bytes per word and other constants that depend on CPU architecture and C compiler type model. It tries to discover most of these constants by running its own experiments on the build host, so if you are cross-compiling, beware.

jsdtoa.cpp, jsdtoa.h, dtoa.c

dtoa.c contains David Gay's portable double-precision floating point to string conversion code, with Permission To Use notice included. jsdtoa.cpp `#include`s this file.

jshash.cpp, jshash.h, jsdhash.cpp, jsdhash.h

Portable, extensible hash tables. These use multiplicative hash for strength reduction over division hash, yet with very good key distribution over power of two table sizes. jshash resolves collisions via chaining, so each entry burns a malloc and can fragment the heap. jsdhash uses open addressing.

jslong.cpp, jslong.h

64-bit integer emulation, and compatible macros that use intrinsic C types, like `long long`, on platforms where they exist (most everywhere, these days).

jsprf.*

Portable, buffer-overrun-resistant sprintf and friends. For no good reason save lack of time, the %e, %f, and %g formats cause your system's native sprintf, rather than `JS_dtoa()`, to be used. This bug doesn't affect SpiderMonkey, because it uses its own `JS_dtoa()` call in `jsnum.cpp` to convert from double to string, but it's a bug that we'll fix later, and one you should be aware of if you intend to use a `JS_*printf()` function with your own floating type arguments - various vendor sprintf's mishandle NaN, +/-Inf, and some even print normal floating values inaccurately.

prmjtime.c, prmjtime.h

Time functions. These interfaces are named in a way that makes local vs. universal time confusion likely. Caveat emptor, and we're working on it. To make matters worse, Java (and therefore JavaScript) uses "local" time numbers (offsets from the epoch) in its Date class.

jsfile.cpp, jsfile.h, jsfile.msg

Obsolete. Do not use these files.

Makefile.in, build.mk

Mozilla makefiles. If you're building Gecko or Firefox, the larger build system will use these files. They are also used for current standalone builds.

Makefile.ref, rules.mk, config.mk, config/*

Obsolete SpiderMonkey standalone makefiles from 1.8 and earlier. See SpiderMonkey Build Documentation.

See also

- jsd

Last modified: Nov 30, 2019, by MDN contributors

Related Topics

SpiderMonkey

References:

- JSAPI reference
- Debugger-API

Guides:

- General
- SpiderMonkey internals
  - 64-bit Compatibility
  - Bytecode Descriptions
  - Bytecodes
  - Functions
  - Garbage collection
  - Invariants
  - Property cache
  - Self-hosted builtins in SpiderMonkey
  - SpiderMonkey Internals: Thread Safety
  - Tracing JIT

Contributing to SpiderMonkey:

- Getting started
- Tests

Releases:

- Release notes

Documentation:

- Useful lists
- Contribute