28 minutes — Written by amarekano

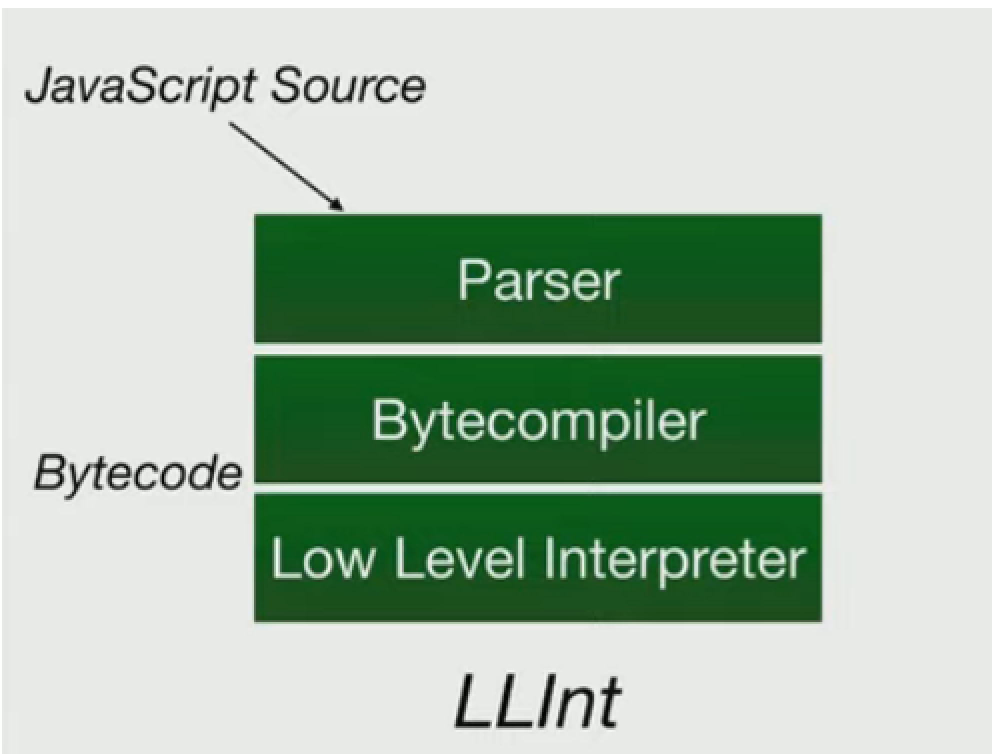
JavaScriptCore Internals Part I: Tracing JavaScript Source to Bytecode

Table of Contents

Introduction

Fuzzing Webkit's JavaScriptCore (JSC) with Fuzzilli proved to be quite successful and produced a fair number of crashes over time. However, once a crash was detected, triaging the crashes for exploitability took a fair bit of time due to unfamiliarity with the WebKit codebase and the lack of easily available documentation on navigating the codebase. This motivated the creation of this blog series to dig into the internals of JSC and hopefully be useful to others who wish to bootstrap their knowledge on the engine. This blogpost series is also aimed at security researchers to help them navigate aspects of the engine that are relevant for vulnerability research.

Part I of this series explores how source code is parsed and converted to bytecode and trace this journey through the codebase. The image, reproduced from a presentation on JSC[1], below describes the three aspects of this pipeline that will be cover as part of the blog post.



This post will cover the source code parser in JSC, the bytecode compiler which takes an AST (Abstract Syntax Tree) generated at the end of the parsing phase and emit bytecode from it. The bytecode is the source of truth for the engine and is one of the key inputs to the various Just-In-Time (JIT) compilers in JSC. This post will explore generated bytecode and help understand some of the opcodes and their operands. Finally, the post concludes by touching upon bytecode execution by the Low Level Interpreter (LLInt). Part II of this blog series will dive into the details of Low Level Interpreter (LLInt) and the Baseline JIT.

Existing Work

An excellent talk on JSC architecture and JIT tiers that is highly recommended is Michael Saboff — JavaScriptCore, many compilers make this engine perform. Whilst it does not go into the internals of each JIT tier, it does provide an overview and the various optimisation techniques and profiling methods employed by the engine.

Another useful blog that on navigating the codebase was this WebKit wiki. This did provide a useful highlevel overview of the engine but lacked sufficient detail.

Saelo's phrack paper on "Attacking JavaScript Engines: A case study of JavaScriptCore and CVE-2016-4622" is another good read to familiarise oneself with the JSC runtime which he discusses in the various sections of his research.

Getting Started

This section will demonstrate setting up a debugging environment and compile a debug build of the `jsc` shell utility. A working debugging environment will be important in being able to navigate the JSC codebase and examine various aspects of the engine execution at runtime.

Generating a debug build

The instructions below will clone the webkit repository mirrored on github and compile a debug build for the `jsc` shell.

```
$ git clone git://git.webkit.org/WebKit.git && cd WebKit
$ Tools/gtk/install-dependencies
$ Tools/Scripts/build-webkit --jsc-only --debug
$ cd WebKitBuild/Debug/bin/
$ ./jsc
```

Setting up a debugging environment

Once a debug binary has been generated, it is time to configure an IDE (Integrated Developement Environment) and debugger for code review and stepping through the execution of the engine. This post will use vscode with ccls for code review and integrated with gdb for interactive debugging. However, the reader is free to use an IDE and debugger that they are most comfortable with. Should the reader decide to continue with vscode and ccls, the following launch task will need to be added to `launch.json` in vscode. Note: Do ensure that the file paths (i.e. *program* and *args*) listed in the snippet below are appropriately modified to reflect the target debugging environment.

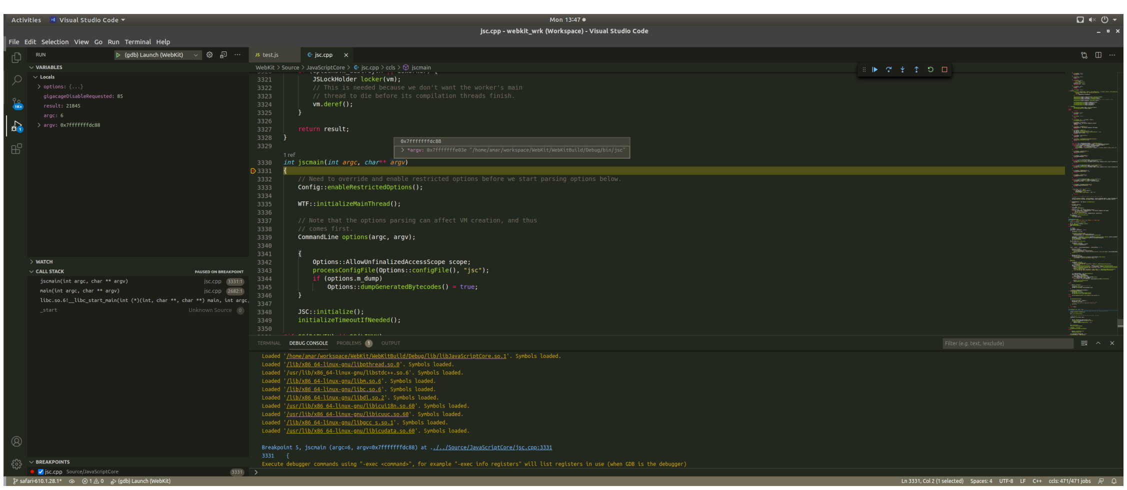
```
{
    "version": "0.2.0",
    "configurations": [
        {
            "name": "(gdb) Launch",
            "type": "cppdbg",
            "request": "launch",
            "program": "/home/amar/workspace/WebKit/WebKitE
            "args": ["--dumpGeneratedBytecodes=true", "--us
            "stopAtEntry": false,
            "cwd": "${workspaceFolder}",
            "environment": [],
            "externalConsole": false,
            "MIMode": "gdb",
            "setupCommands": [
                {
                    "description": "Enable pretty-printing
                    "text": "-enable-pretty-printing",
```

```
                "ignoreFailures": true
            }
        ],
        "preLaunchTask": "WebKit Debug Build"
    }
  ]
}
```

An optional but handy build task was added to launch configuration that would build `jsc` before launching the debugger. This step is optional but it's generally a good idea to have the codebase in sync with the debug builds being generated. The build task added to tasks.json is listed below:

```
{
    "version": "2.0.0",
    "tasks": [
        {
            "label": "WebKit Debug Build",
            "type": "shell",
            "command": "/home/amar/workspace/WebKit/Tools/S
        }
    ]
}
```

Should all go to plan, the debugging environment should now allow launching gdb (*F5* in vscode) and hit breakpoints that have been setup like in the screenshot shown below:



JSC shell

This section will discuss the `jsc` shell that was generated in the previous section and its importance in being able to understand the engine internals. The `jsc` shell allows both researchers and developers to test JavaScriptCore as an independent library without the need to build the entire WebKit project. The `jsc` shell provides a Repeat-Eval-Print-Loop (REPL) environment for the javascript engine. In addition it also allows js scripts to be passed via the commandline, which is read by the `jsc` shell, parsed and executed by the engine.

The source code to the shell can be found in jsc.cpp

The entry point to the shell is jscmain. This function is responsible for initialising Web Template Framework (WTF), which is a set of commonly used functions from the Webkit codebase, and parsing options before a JSC vm can be created.

Initialisation of JSC begins with the call to runJSC which when invoked allocates memory for the `VM` object as well as initialises and retrieves a `GlobalObject` reference.

```
int runJSC(const CommandLine& options, bool isWorker, const
{
    //... code truncated for brevity

    VM& vm = VM::create(LargeHeap).leakRef();
    //... code truncated for brevity

    GlobalObject* globalObject = nullptr;
    {
    //... code truncated for brevity
        globalObject = GlobalObject::create(vm, GlobalObjec
        globalObject->setRemoteDebuggingEnabled(options.m_e
        func(vm, globalObject, success);
        //... code truncated for brevity
    }
    //... code truncated for brevity
```

`GlobalObject::create` eventually ends up calling `JSGlobalObject::init(VM& )` which is responsible to initialising the `VM` with the required builtins and other runtime setup activities. This post won't go into the details of how the builtin code is parsed and linked to the VM but the interested reader should explore JavaScriptCore/builtins/ for all the builtin objects/constructors that form part of the JSC runtime. Alternatively, setting breakpoints and stepping through the execution of `JSGlobalObject::init` would be another approach.

Once `VM` and `GlobalObject` have been initialised, the lambda function, passed to `runJSC` is executed. The lambda function calls runWithOptions which takes three arguments; a pointer to the initialised `GlobalObject`, the commandline options passed to the jsc shell and a status variable.

`runWithOptions`'s primary goal is to create the necessary buffers to store the raw javascript that is supplied to the `jsc` shell. For the purpose of the blog the following script file (*test.js*) will be passed as a commandline parameter to `jsc`:

```
$ cat test.js
let x = 10;
let y = 20;
let z = x + y;

$ ./WebKitBuild/Debug/bin/jsc test.js
```

Once the backing buffers have been setup and populated, the shell will now being evaluating and executing the script with a call to `evaluate`:

```
NakedPtr<Exception> evaluationException;
JSValue returnValue = evaluate(globalObject, jscSource(scri
```

In the snippet above `scriptBuffer` stores the contents of *test.js* passed via the commandline; `sourceOrigin` stores the URI to the script, which in this case is the absolute path to the script passed on the commandline. `jscSource` is a helper function that generates a SouceCode object from the `scriptBuffer`. The SourceCode object encapsulates the raw script data.

Runtime Setup

Now that the source code has been loaded into the engine via the `jsc` shell, the next step is to hand this over to the JSC engine and initiate processing of the loaded script. The function `evaluate` which is defined in runtime/Completion.cpp invokes `executeProgram` which is the point where the JSC engine takes over and begins processing:

```
JSValue evaluate(JSGlobalObject* globalObject, const Source
{
    VM& vm = globalObject->vm();

    //... code truncated for brevity

    JSObject* thisObj = jsCast<JSObject*>(thisValue.toThis(
    JSValue result = vm.interpreter->executeProgram(source,

    //... code truncated for brevity

    return result;
}
```

The function `Interpreter::executeProgram` performs three important tasks that will be the focal points of discussion throughout this blog post. The main tasks are as follows:

1. Initiating lexing and parsing of the sourcecode,
2. Generation of bytecode,
3. Execution of bytecode.

These have been highlighted in the truncated function code below:

```
JSValue Interpreter::executeProgram(const SourceCode& sourc
{
    JSScope* scope = thisObj->globalObject()->globalScope()
    VM& vm = scope->vm();

    //.. truncated code

    ProgramExecutable* program = ProgramExecutable::create(

    //... code truncated for brevity

    VMEntryScope entryScope(vm, globalObject);

    // Compile source to bytecode if necessary:
    JSObject* error = program->initializeGlobalProperties(v

    //... code truncated for brevity

    ProgramCodeBlock* codeBlock;
    {
        CodeBlock* tempCodeBlock;
        Exception* error = program->prepareForExecution<Pro
        //... code truncated for brevity
        codeBlock = jsCast<ProgramCodeBlock*>(tempCodeBlock
    }

    RefPtr<JITCode> jitCode;
    ProtoCallFrame protoCallFrame;
    {
        DisallowGC disallowGC; // Ensure no GC happens. GC
        jitCode = program->generatedJITCode();
        protoCallFrame.init(codeBlock, globalObject, global
    }

    // Execute the code:
    //... code truncated for brevity
    JSValue result = jitCode->execute(&vm, &protoCallFrame)
    return checkedReturn(result);
}
```

- The `executeProgram` beings by first allocating memory for the `ProgramExecutable` object and then calls the `ProgramExecutable` constructor with a call to `ProgramExecutable::create`. The call inturn calls the the base constructor (i.e. `GlobalExecutable`) which has the signature shown below:

```
GlobalExecutable(Structure* structure, VM& vm, const Source
        : Base(structure, vm, sourceCode, isInStrictContext
    {
    }
```

- `GlobalExecutable` inturn calls its base constructor `ScriptExecutable`. The ScriptExecutable constructor performs two functions, it first initalises the ExecutableBase and initialise several other class members. `ExecutableBase` calls the `JSCell` constructor which effectively generates a `JSCell` for the `ProgramExecutable`.

The `ExecutableBase` and its derived classes (e.g. `ProgramExecutable`) is important to this discussion as it stores references to JIT code which gets executed at later stages.

Lets now return to `Interpreter::executeProgram` and continue the discussion on this function; once the `ProgramExecutable` object `program` has been initialised, the function performs a range of validation checks to evaluate whether the supplied script is a JSON script. Since *test.js* does not contain any JSON, these checks can be ignored for now and this has been truncated in the code snippet documented previously. The next interesting instruction that needs to be considered is `ProgramExecutable::initializeGlobalProperties`:

```
// Compile source to bytecode if necessary:
JSObject* error = program->initializeGlobalProperties(vm, g
```

The function `ProgramExecutable::initializeGlobalProperties` uses the source object to generate an `UnlinkedProgramCodeBlock`:

```
UnlinkedProgramCodeBlock* unlinkedCodeBlock = vm.codeCache(
```

A `CodeCache` according to the developer comments in the source is a cache for top-level code such as `<script>`, `window.eval()`, `new Function`, and `JSEvaluateScript()`. The CodeCache is initalised when the VM object is instantiated. The function call `getUnlinkedProgramCodeBlock` inturn calls `getUnlinkedGlobalCodeBlock`:

```
template <class UnlinkedCodeBlockType, class ExecutableType
UnlinkedCodeBlockType* CodeCache::getUnlinkedGlobalCodeBloc
{
    //... code truncated for brevity

    VariableEnvironment variablesUnderTDZ;
    unlinkedCodeBlock = generateUnlinkedCodeBlock<UnlinkedC

    //... code truncated for brevity

    return unlinkedCodeBlock;
}
```

The call to `generateUnlinkedCodeBlock` eventually leads to a call to `CodeCache::generateUnlinkedCodeBlockImpl`. This function is responsible for initiating parsing of the script as well as bytecode generation. The call stack upto this point in the execution will look similar to the one below:

```
libJavaScriptCore.so.1!JSC::generateUnlinkedCodeBlockImpl<J
libJavaScriptCore.so.1!JSC::generateUnlinkedCodeBlock<JSC::
libJavaScriptCore.so.1!JSC::CodeCache::getUnlinkedGlobalCod
libJavaScriptCore.so.1!JSC::CodeCache::getUnlinkedProgramCo
libJavaScriptCore.so.1!JSC::ProgramExecutable::initializeGl
libJavaScriptCore.so.1!JSC::Interpreter::executeProgram(JSC
libJavaScriptCore.so.1!JSC::evaluate(JSC::JSGlobalObject *
runWithOptions(GlobalObject * globalObject, CommandLine & c
operator()(const struct {...} * const __closure, JSC::VM &
runJSC<jscmain(int, char**)::<lambda(JSC::VM&, GlobalObject
jscmain(int argc, char ** argv) (/home/amar/workspace/WebKi
```

```
main(int argc, char ** argv) (/home/amar/workspace/WebKit/S
libc.so.6!__libc_start_main(int (*)(int, char **, char **)
_start (Unknown Source:0)
```

Lexing and Parsing

This section will now explore how the source code loaded into the engine is lexed and parsed by the engine to generate an AST. Lexing and parsing is a process where raw source code is tokenised and the tokens generated are then parsed to build an AST. This processing will also identify syntax and semantic errors that may be present in the supplied js script by validating the script against the ECMA spec. This beings with the call to function `CodeCache::generateUnlinkedCodeBlockImpl` which is described below and unimportant code truncated.

```
UnlinkedCodeBlockType* generateUnlinkedCodeBlockImpl(VM& vm
{
    typedef typename CacheTypes<UnlinkedCodeBlockType>::Roo
    bool isInsideOrdinaryFunction = executable && executabl
    std::unique_ptr<RootNode> rootNode = parse<RootNode>(
        vm, source, Identifier(), JSParserBuiltinMode::NotE

    //... code truncated for brevity

    ExecutableInfo executableInfo(usesEval, false, false, C

    UnlinkedCodeBlockType* unlinkedCodeBlock = UnlinkedCode
    unlinkedCodeBlock->recordParse(rootNode->features(), rc

    //... code truncated for brevity

    error = BytecodeGenerator::generate(vm, rootNode.get(),

    if (error.isValid())
        return nullptr;
```

```
    return unlinkedCodeBlock;
}
```

Parsing is initiated with a call to `parse` which is defined in parser/Parser.h:

```
std::unique_ptr<RootNode> rootNode = parse<RootNode>(
        vm, source, Identifier(), JSParserBuiltinMode::NotE
```

The following lines of code within the parse function are responsible for setting up the parser and analysing the source script:

```
Parser<Lexer<LChar>> parser(vm, source, builtinMode, strict
result = parser.parse<ParsedNode>(error, name, parseMode, i
```

The first line creates a parser object whose constructor, among other activities, instantiates a lexer object with the unlinked source code:

```
m_lexer = makeUnique<LexerType>(vm, builtinMode, scriptMode
m_lexer->setCode(source, &m_parserArena);
```

Additionally, the constructor also sets up the details of the first token location:

```
 m_token.m_location.line = source.firstLine().oneBasedInt()
 m_token.m_location.startOffset = source.startOffset();
 m_token.m_location.endOffset = source.startOffset();
 m_token.m_location.lineStartOffset = source.startOffset();
```

Once these parameters have been initialised, it makes a call to `next` :

```
ALWAYS_INLINE void next(OptionSet<LexerFlags> lexerFlags =
    {
```

```
        int lastLine = m_token.m_location.line;
        int lastTokenEnd = m_token.m_location.endOffset;
        int lastTokenLineStart = m_token.m_location.lineSta
        m_lastTokenEndPosition = JSTextPosition(lastLine, l
        m_lexer->setLastLineNumber(lastLine);
        m_token.m_type = m_lexer->lex(&m_token, lexerFlags,
}
```

The function `m_lexer->lex` eventually calls the function [`Lexer<T>::lexWithoutClearingLineTerminator`]. This function lexes the next token in the source and returns a JSToken object to the caller.

```
JSTokenType Lexer<T>::lexWithoutClearingLineTerminator(JSTo
```

Anyone interested in the workings of the lexer should review the functions within Lexer.cpp

Once the `parser` object has been initialised, the function `parse` is invoked which beings the process of parsing. The parsing function, parse, invokes `parseInner`:

```
auto parseResult = parseInner(calleeName, parseMode, parsir
```

The function `parseInner` begins by setting up a context object for `ASTBuilder` called `context`. `context` now has references to the source code, the `parserArena` and the vm. After a series of checks parseInner eventually calls `parseSouceElements`:

```
sourceElements = parseSourceElements(context, CheckForStric
```

The function `parseSourceElements` beings by creating a `sourceElements` object which serves as a store for statements that have been parsed.

```
template <typename LexerType>
template <class TreeBuilder> TreeSourceElements Parser<Lexe
```

```
{
    const unsigned lengthOfUseStrictLiteral = 12; // "use s
    TreeSourceElements sourceElements = context.createSourc

    //... code truncated for brevity

    while (TreeStatement statement = parseStatementListItem
        if (shouldCheckForUseStrict) {
           //... code truncated for brevity
        }
        context.appendStatement(sourceElements, statement);
    }

    propagateError();
    return sourceElements;
}
```

The function then iterates over the statements in the source code to lex and parse the unlinkedSourceCode referenced by `context` with a call to `parseStatementListItem` .

```
while (TreeStatement statement = parseStatementListItem(con
```

The function `parseStatementListItem` is responsible for continuing the lexing and parsing of the source code to construct an AST. Parsing of tokens to generate `TreeStatement` nodes; the interested reader can explore this by reviewing the functions within Parser.cpp. An example of a variable declaration parsing function can be found here.

```
template <class TreeBuilder> TreeStatement Parser<LexerType
{
    ASSERT(match(VAR) || match(LET) || match(CONSTTOKEN));
    JSTokenLocation location(tokenLocation());
    int start = tokenLine();
    int end = 0;
    int scratch;
    TreeDestructuringPattern scratch1 = 0;
    TreeExpression scratch2 = 0;
    JSTextPosition scratch3;
```

```
    bool scratchBool;
    TreeExpression variableDecls = parseVariableDeclaratior
    propagateError();
    failIfFalse(autoSemiColon(), "Expected ';' after variat

    return context.createDeclarationStatement(location, var
}
```

The `StatementNodes` returned at the end of the call to `parseStatementListItem` are then validated and added to the `sourceElements` object.

```
context.appendStatement(sourceElements, statement);
```

`parseSourceElements` returns by creating an AST of `ParsedNode` elements. When `parse` returns without any syntax or semantic parsing errors, we have a valid AST with `rootNode` pointing to the root of the tree. The various node types that form an AST are defined in the parser/NodeContructors.h and parser/Nodes.h

Bytecode

This section dives into the details of bytecode generation from the AST generated in the previous section. It will be worth the readers time to review the webkit blog[2] on the latest changes to the bytecode format in JSC and background reading on why these changes were introduced. Bytecode is the source of truth for the engine and the discussion in this section is perhaps the most important to the rest of the blog series.

Generation

Once an AST has been generated, the next step before bytecode generation is to create an `UnlikedCodeBlock` object.

```
UnlinkedCodeBlockType* unlinkedCodeBlock = UnlinkedCodeBloc
unlinkedCodeBlock->recordParse(rootNode->features(), rootNc
```

The generated `unlinkedCodeBlock` is then populated with unlinked bytecode with the call to `BytecodeGenerator::generate`

```
error = BytecodeGenerator::generate(vm, rootNode.get(), sou
```

The function `BytecodeGenerator::generate` initialises a `BytecodeGenerator` object with the supplied AST (i.e. the root node reference) and then beings generating bytecode for the AST:

```
template<typename Node, typename UnlinkedCodeBlock>
static ParserError generate(VM& vm, Node* node, const Sourc
{
    //... code truncated for brevity

    DeferGC deferGC(vm.heap);
    auto bytecodeGenerator = makeUnique<BytecodeGenerator>(
    auto result = bytecodeGenerator->generate();

    //... code truncated for brevity

    return result;
}
```

First a `BytecodeGenerator` object is initialised, by calling the `BytecodeGenerator` constructor. This constructor in addition to initialising several aspects of the generator also emits bytecode for the program prologue (e.g. the program entry point).

The call to generate, initiates bytecode generation for various function initalisation constructs before emitting bytecode for the global scope.

```
ParserError BytecodeGenerator::generate()
{
    //... code truncated for brevity

    m_codeBlock->setThisRegister(m_thisRegister.virtualRegi

    //... code truncated for brevity

    if (m_restParameter)
        m_restParameter->emit(*this);

    {
        RefPtr<RegisterID> temp = newTemporary();
        RefPtr<RegisterID> tolLevelScope;
        for (auto functionPair : m_functionsToInitialize) {
            FunctionMetadataNode* metadata = functionPair.f
            FunctionVariableType functionType = functionPai
            emitNewFunction(temp.get(), metadata);

            //... code truncated for brevity
        }
    }

    bool callingClassConstructor = false;

    //... code truncated for brevity

    if (!callingClassConstructor)
        m_scopeNode->emitBytecode(*this);
    else {
        emitUnreachable();
    }

    for (auto& handler : m_exceptionHandlersToEmit) {
        Ref<Label> realCatchTarget = newLabel();
        TryData* tryData = handler.tryData;

        OpCatch::emit(this, handler.exceptionRegister, hand

        //... code truncated for brevity

        m_codeBlock->addJumpTarget(m_lastInstruction.offset
```

```
        emitJump(tryData->target.get());
        tryData->target = WTFMove(realCatchTarget);
    }

    m_staticPropertyAnalyzer.kill();

    for (auto& range : m_tryRanges) {
        int start = range.start->bind();
        int end = range.end->bind();

        if (end <= start)
            continue;

        UnlinkedHandlerInfo info(static_cast<uint32_t>(star
            static_cast<uint32_t>(range.tryData->target->bi
        m_codeBlock->addExceptionHandler(info);
    }

    //... code truncated for brevity

    m_codeBlock->finalize(m_writer.finalize());

    //... code truncated for brevity

    return ParserError(ParserError::ErrorNone);
}
```

The function `emitBytecode`, called by generate, ends up calling `emitProgramNodeBytecode`, which as the name suggests is responsible for generating bytecode for the program node by traversing the AST.

```
static void emitProgramNodeBytecode(BytecodeGenerator& gene
{
    generator.emitDebugHook(WillExecuteProgram, scopeNode.s

    RefPtr<RegisterID> dstRegister = generator.newTemporary
    generator.emitLoad(dstRegister.get(), jsUndefined());
    generator.emitProfileControlFlow(scopeNode.startStartOf
    scopeNode.emitStatementsBytecode(generator, dstRegister
```

```
    generator.emitDebugHook(DidExecuteProgram, scopeNode.la
    generator.emitEnd(dstRegister.get());
}
```

The various opcodes are defined in BytecodeList.rb which at compile time is used to generate `BytecodeStructs.h` which is referenced by he `BytecodeGenerator` to emit the relevant opcodes. The structs for the various opcodes also define several helper functions, one of which allows dumping bytecodes to stdout in a human readable format. `BytecodeStructs.h` is typically located under `<build-directory>/Debug/DerivedSources/JavaScriptCore/BytecodeStructs.h`. An example of the `OpAdd` instruction is shown below:

```
struct OpAdd : public Instruction {
    static constexpr OpcodeID opcodeID = op_add;
    static constexpr size_t length = 6;

    template<typename BytecodeGenerator>
    static void emit(BytecodeGenerator* gen, VirtualRegiste
    {
        emitWithSmallestSizeRequirement<OpcodeSize::Narrow,
    }

    //... code truncated for brevity

private:
    //... code truncated for brevity

    template<OpcodeSize __size, bool recordOpcode, typename
    static bool emitImpl(BytecodeGenerator* gen, VirtualReg
    {

        if (__size == OpcodeSize::Wide16)
            gen->alignWideOpcode16();
        else if (__size == OpcodeSize::Wide32)
            gen->alignWideOpcode32();
        if (checkImpl<__size>(gen, dst, lhs, rhs, operandTy
            if (recordOpcode)
                gen->recordOpcode(opcodeID);
```

```
            if (__size == OpcodeSize::Wide16)
                gen->write(Fits<OpcodeID, OpcodeSize::Narro
            else if (__size == OpcodeSize::Wide32)
                gen->write(Fits<OpcodeID, OpcodeSize::Narro
            gen->write(Fits<OpcodeID, OpcodeSize::Narrow>::
            gen->write(Fits<VirtualRegister, __size>::conve
            gen->write(Fits<VirtualRegister, __size>::conve
            gen->write(Fits<VirtualRegister, __size>::conve
            gen->write(Fits<OperandTypes, __size>::convert(
            gen->write(Fits<unsigned, __size>::convert(__me
            return true;
        }
        return false;
    }

public:
    void dump(BytecodeDumperBase* dumper, InstructionStream
    {
        //... code truncated for brevity
    }

    //... code truncated for brevity

    VirtualRegister m_dst;
    VirtualRegister m_lhs;
    VirtualRegister m_rhs;
    OperandTypes m_operandTypes;
    unsigned m_metadataID;
};
```

The Domain Specific Language (DSL) used to define `BytecodeList.rb` can be found under JavaScriptCore/generator.

In addition to function initialisers and emitting bytecode for the program node, `generate` also emits bytecode for exception handlers and try-catch nodes.

Finally the call to `finalise` which completes the writing of instruction bytes as unlinked bytecode to the allocated codeblock.

```
m_codeBlock->finalize(m_writer.finalize());
```

Returning back to our calling function, `Interpreter::executeProgram` , after the unlinked codeblock has been generated, the bytecode can now be linked and executed. The unlinked bytecode is first encoded with a call to prepareForExecution:

```
CodeBlock* tempCodeBlock;
Exception* error = program->prepareForExecution<ProgramExec
```

Through a series of function calls, prepareForExecution eventually ends up calling `CodeBlock::finishCreation` . From the developer notes, this function is responsible for converting the unlinked bytecode to linked bytecode.

```
// The main purpose of this function is to generate linked
// of linking is taking an abstract representation of bytec
// chain. For example, this process allows us to cache the
// outside of this CodeBlock's compilation unit. It also al
// we can't generate during unlinked bytecode generation. T
// flow or introduce new locals. The reason for this is we
// all the CodeBlocks of an UnlinkedCodeBlock. We rely on t
// inside UnlinkedCodeBlock.
bool CodeBlock::finishCreation(VM& vm, ScriptExecutable* ow
    JSScope* scope)
{
```

The function iterates over the unlinked instructions in the codeblock and links them based on the opcode retrieved.

```
const InstructionStream& instructionStream = instructions()
    for (const auto& instruction : instructionStream) {
        OpcodeID opcodeID = instruction->opcodeID();
        m_bytecodeCost += opcodeLengths[opcodeID];
        switch (opcodeID) {
        LINK(OpHasIndexedProperty)
```

```
LINK(OpCallVarargs, profile)
LINK(OpTailCallVarargs, profile)
//... code truncated for brevity
```

The process of linking and updating the metadata table is described in the webkit blog on the new bytecode format. Adding the `dumpGeneratedBytecodes` or the shortened version `-d` commandline option to the `jsc` shell allows dumping the generated bytecodes to stdout.

```
$ cat test.js
let x = 10;
let y = 20;
let z = x + y;

$ ./WebKitBuild/Debug/bin/jsc --dumpGeneratedBytecodes=true
```

The bytecodes generated from parsing *test.js* are as follows:

```
<global>#AccRYt:[0x7fffee4bc000->0x7fffeeecb848, NoneGlobal

bb#1
[   0] enter
[   1] get_scope          loc4
[   3] mov                loc5, loc4
[   6] check_traps
[   7] mov                loc6, Undefined(const0)
[  10] resolve_scope      loc7, loc4, 0, GlobalProperty, 0
[  17] put_to_scope       loc7, 0, Int32: 10(const1), 10485
[  25] resolve_scope      loc7, loc4, 1, GlobalProperty, 0
[  32] put_to_scope       loc7, 1, Int32: 20(const2), 10485
[  40] resolve_scope      loc7, loc4, 2, GlobalProperty, 0
[  47] resolve_scope      loc8, loc4, 0, GlobalProperty, 0
[  54] get_from_scope     loc9, loc8, 0, 2048<ThrowIfNotFou
[  62] mov                loc8, loc9
[  65] resolve_scope      loc9, loc4, 1, GlobalProperty, 0
[  72] get_from_scope     loc10, loc9, 1, 2048<ThrowIfNotFo
[  80] add                loc8, loc8, loc10, OperandTypes(1
[  86] put_to_scope       loc7, 2, loc8, 1048576<DoNotThrow
[  94] end                loc6
```

```
Successors: [ ]


Identifiers:
  id0 = x
  id1 = y
  id2 = z

Constants:
   k0 = Undefined
   k1 = Int32: 10: in source as integer
   k2 = Int32: 20: in source as integer
```

Opcodes

The previous section provided an explanation on how bytecode is generated and how one can go about tracing the bytecode emission process in a debugger. It also introduced at a handy commandline flag that allows dumping generated bytecode to stdout. This section discusses how to read and understand the dumped bytecode.

Every program has a prologue and epilogue bytecode that the generator emits. This can test this by creating an empty javascript file and passing it to the `jsc` shell. The resulting bytecodes are as follows:

```
$ touch empty.js && ./jsc -d empty.js

<global>#EW7Aoi:[0x7f42d2bc4000->0x7f43135cb768, NoneGlobal

bb#1
[   0] enter
[   1] get_scope          loc4
[   3] mov                loc5, loc4
[   6] check_traps
[   7] mov                loc6, Undefined(const0)
[  10] end                loc6
Successors: [ ]
```

```
Constants:
   k0 = Undefined

End: undefined
```

The first line of the output contains information about the codeblock. The dumper function `CodeBlock::dumpAssumingJITType` prints details about the CodeBlock associated with the generated bytecode:

```
<global>#EW7Aoi:[0x7f42d2bc4000->0x7f43135cb768, NoneGlobal
```

`<global>` here is the codeType which in this case refers to the global program. Bytecode for user-defined functions would have the function name instead of `<global>`. `#EW7Aoi` is the hash of the source code string that the codeblock was created for. The two memory address that follow (i.e. `0x7f42d2bc4000` and `0x7f43135cb768`) represent the target and target offset for the executable. `None` describes the JITType and `Global` the codeType. The number `12` represents the number of instructions in the codeblock. The remaining part of the header, prints statistics about the generated bytecode, such as number of instructions, parameters, callee registers, variables and lastly the location to the scope register.

What follows the header is a dump of the bytecode graph. This is essentially a for-loop that iterates over the basic blocks in the graph and prints out the instructions in the basic block.

```
bb#1
[   0] enter
[   1] get_scope          loc4
[   3] mov                loc5, loc4
[   6] check_traps
[   7] mov                loc6, Undefined(const0)
[  10] end                loc6
Successors: [ ]
```

In the snippet above, `bb#` identifies the basic block in the graph. The first column represents the offset of the instruction in the instruction stream. The next column lists the various opcodes and the last column the operands passed to the opcode. Take the following snippet of the `mov` opcode:

```
[   3] mov                loc5, loc4
```

Here, `loc5` represents the destination register and `loc4` the source register. One can infer this by looking up the `OpMov::dump` function defined in `DerivedSources/JavaScriptCore/BytecodeStructs.h` .

```
void dump(BytecodeDumperBase* dumper, InstructionStream::Of
    {
        dumper->printLocationAndOp(__location, &"**mov"[2 -
        dumper->dumpOperand(m_dst, true);
        dumper->dumpOperand(m_src, false);
    }
```

At the end of the basic block is a list of `Successor` blocks that can be reached from the current basic block. In the dump above we don't have any successor blocks since there are no control flow edges in *empty.js*.

Towards the end of the bytecode dump is the footer which typically contains information about *Identifiers*, *Constants*, *ExceptionHandlers* and *JumpTables*. In the dump snippet, there exists one constant which is the value returned at the end of program execution:

```
Constants:
   k0 = Undefined
```

Lets now attempt to explore the bytecode with a more interesting program. For this exercise lets use a *fibonacci* sequence generator:

```
function fibonacci(num) {
  if (num <= 1) return 1;

  return fibonacci(num - 1) + fibonacci(num - 2);
}

let fib = fibonacci(5)

print(fib)
```

The dumped bytecode is as follows:

```
<global>#BDIvjt:[0x7fffee4bc000->0x7fffeeecb848, NoneGlobal

bb#1
[   0] enter
[   1] get_scope          loc4
[   3] mov                loc5, loc4
[   6] check_traps
[   7] new_func           loc6, loc4, 0
[  11] resolve_scope      loc7, loc4, 0, GlobalProperty, 0
[  18] mov                loc8, loc7
[  21] put_to_scope       loc8, 0, loc6, 2048<ThrowIfNotFou
[  29] mov                loc6, Undefined(const0)
[  32] resolve_scope      loc7, loc4, 1, GlobalProperty, 0
[  39] resolve_scope      loc12, loc4, 0, GlobalProperty, 0
[  46] get_from_scope     loc8, loc12, 0, 2048<ThrowIfNotFo
[  54] mov                loc11, Int32: 5(const1)
[  57] call               loc8, loc8, 2, 18
[  63] put_to_scope       loc7, 1, loc8, 1048576<DoNotThrow
[  71] mov                loc6, Undefined(const0)
[  74] resolve_scope      loc10, loc4, 2, GlobalProperty, 0
[  81] get_from_scope     loc7, loc10, 2, 2048<ThrowIfNotFo
[  89] resolve_scope      loc9, loc4, 1, GlobalProperty, 0
[  96] get_from_scope     loc11, loc9, 1, 2048<ThrowIfNotFo
[ 104] mov                loc9, loc11
[ 107] call               loc6, loc7, 2, 16
[ 113] end                loc6
Successors: [ ]
```

```
Identifiers:
  id0 = fibonacci
  id1 = fib
  id2 = print

Constants:
   k0 = Undefined
   k1 = Int32: 5: in source as integer

fibonacci#AcXBvC:[0x7fffee4bc130->0x7fffee4e5100, NoneFunct

bb#1
[   0] enter
[   1] get_scope          loc4
[   3] mov                loc5, loc4
[   6] check_traps
[   7] jnlesseq           arg1, Int32: 1(const0), 6(->13)
Successors: [ #3 #2 ]

bb#2
[  11] ret                Int32: 1(const0)
Successors: [ ]

bb#3
[  13] resolve_scope      loc10, loc4, 0, GlobalProperty, 0
[  20] get_from_scope     loc6, loc10, 0, 2048<ThrowIfNotFc
[  28] sub                loc9, arg1, Int32: 1(const0), Ope
[  34] call               loc6, loc6, 2, 16
[  40] resolve_scope      loc10, loc4, 0, GlobalProperty, 0
[  47] get_from_scope     loc7, loc10, 0, 2048<ThrowIfNotFc
[  55] sub                loc9, arg1, Int32: 2(const1), Ope
[  61] call               loc7, loc7, 2, 16
[  67] add                loc6, loc6, loc7, OperandTypes(12
[  73] ret                loc6
Successors: [ ]


Identifiers:
  id0 = fibonacci

Constants:
   k0 = Int32: 1: in source as integer
   k1 = Int32: 2: in source as integer
```

```
8
```

The dumped output contains bytecode for the the main program as well as the function `fibonacci` defined in the script. There are several new instructions that have been emitted by the bytecode generator, such as `new_func` which indicates a function declaration, `call` and `ret` which indicates that a function is being called and when a function returns, `jnlesseq` which is a conditional jump instruction if the *lhs* is less than or equal to the *rhs*, arithmetic opcodes such as `add` and `sub`, etc. To learn more about an opcode and it's operands, one approach would be to add a breakpoint and the `dump` function in `BytecodeStructs.h` and inspect the operands and trace their origins while debugging.

The function `fibonacci` is composed of three basic blocks: `bb#1`, `bb#2` and `bb#3`. Basic block `bb#1` has two successors `bb#3` and `bb#2`. This indicates that the block `bb#1` has two control flow edges, one that leads to `bb#2` and the other that leads to `bb#3`.

The dumped footers for the two codeblocks `<global>` and `fibonacci` list the various Identifiers and Constants that are referenced by the bytecode. For example the footer for the main program is as follows:

```
Identifiers:
  id0 = fibonacci
  id1 = fib
  id2 = print

Constants:
   k0 = Undefined
   k1 = Int32: 5: in source as integer
```

Those familiar with x86 or arm assembly will find the opcode syntax to be very similar and can make an educated guess on some of the actions performed by the opcodes. For example the `mov` opcode is similar to the x86 mov, which takes the form `mov`

`<dst>` `<src>` . However, there are some opcodes that may not be intuitive, and to determine the opcode action one would need to trace the execution of the opcodes in the LLInt or evaluate LLInt assembly to understand their operation.

Execution

The linked `codeBlock` is now ready to be consumed and executed by the interpreter. This beings at the point when the `codeBlock` is passed back up the call stack to `ScriptExecutable::prepareForExecutionImpl` .

```
Exception* ScriptExecutable::prepareForExecutionImpl(VM& vm
{
    //... code truncated for brevity

    Exception* exception = nullptr;
    CodeBlock* codeBlock = newCodeBlockFor(kind, function,
    resultCodeBlock = codeBlock;
    //... code truncated for brevity

    if (Options::useLLInt())
        setupLLInt(codeBlock);
    else
        setupJIT(vm, codeBlock);

    installCode(vm, codeBlock, codeBlock->codeType(), codeE
    return nullptr;
}
```

Within this function, the `codeBlock` is passed to `setupLLInt` which eventually calls `LLInt::setProgramEntrypoint` which sets up the entry point to the program for the LLInt to being executing from:

```
static void setProgramEntrypoint(CodeBlock* codeBlock)
{
    //... code truncated for brevity

    static NativeJITCode* jitCode;
```

```
        static std::once_flag onceKey;
        std::call_once(onceKey, [&] {
            jitCode = new NativeJITCode(getCodeRef<JSEntryPtrTa
        });
        codeBlock->setJITCode(makeRef(*jitCode));
    }
```

The call to `program->generatedJITCode()` retrieves a reference pointer to the interpreted code which is then used to initialise a `ProtoCallFrame`. Finally the call to `jitCode->execute` executes the interpreted bytecode. The snippet below shows the relevant section in `Interpreter::executeProgram`.

```
    RefPtr<JITCode> jitCode;
    ProtoCallFrame protoCallFrame;
    {
        DisallowGC disallowGC; // Ensure no GC happens. GC can r
        jitCode = program->generatedJITCode();
        protoCallFrame.init(codeBlock, globalObject, globalCalle
    }

    // Execute the code:
    throwScope.release();
    ASSERT(jitCode == program->generatedJITCode().ptr());
    JSValue result = jitCode->execute(&vm, &protoCallFrame);
    return checkedReturn(result);
```

Conclusion

In this post we explored how JSC turns javascript source code to bytecode and long this journey we've traced and documented the engine execution as it parses and emits bytecode. We've also reviewed the generated bytecode and analysed bytecode dumps. We concluded with briefly describing how the LLInt is loaded with bytecode and a highlevel overview on how the interpreter executes the bytecode. In Part II of this blog series we will dive into the details of how bytecode gets interpreted by the LLInt and the Baseline JIT.

We hope you've found this post informative, if you have questions, spot something that's incorrect or have suggestions on improving this writeup do reach out to the author @amarekano or @Zon8Research on Twitter. We are more than happy to discuss this at length with anyone interested in this subject and would love to hear your thoughts on it.

Appendix

1. https://www.youtube.com/watch?v=mtVBAcy7AKA ↩

2. https://webkit.org/blog/9329/a-new-bytecode-format-for-javascriptcore/ ↩

#JSC #Safari #WebKit

5815 Words

2020-11-08 00:00 +0000

READ OTHER POSTS

← JavaScriptCore I… JavaScript Engin… →