How To Hack JavaScript Engine

TA: 藤澤 (B4)

2017/09/26

もくじ

JavaScript Engine

JavaScript Engine (1/3)

みなさんが毎日使ってるヤツ

ブラウザ	JavaScript Engine
Mozilla Firefox	SpiderMonkey
Google Chrome	V8
Safari (WebKit)	JavaScriptCore
Edge	ChakraCore

JavaScript Engine (2/3)

SpiderMonkey
Firefox の JavaScript 処理系

`https://developer.mozilla.org/en-US/docs/Mozilla/Projects/SpiderMonkey`

- スタックマシンベースのインタープリタ
- 二段階の JIT コンパイラ Baseline + IonMonkey
  (x86, x64, arm, arm64 (+ mips32, mips64))
- Incremental/Generational/Compacting GC

JavaScript Engine (3/3)

ホットな話題

- 高速化
- メモリ使用量削減
- ECMAScript Spec 追従 (新機能追加, 変更)
- WebAssembly

開発環境構築

開発環境構築

Firefox の開発環境構築と途中まで同じ

https://developer.mozilla.org/en-US/docs/Mozilla/Developer_guide/Build_Instructions/Simple_Firefox_build

```
wget https://hg.mozilla.org/mozilla-central/raw-file/default/python/
        mozboot/bin/bootstrap.py && python bootstrap.py
```

あとは質問に答えるだけ

ビルド

ビルド (1/2)

Firefox... ではなくて, js shell をビルド

Firefox 全体のビルド ⇒ 30 分から数時間
js shell 全体のビルド ⇒ 5 分くらい

JavaScript Engine に閉じてる変更 ⇒ js shell 上で開発

ビルド (2/2)

ビルドの部分から Firefox のドキュメントとは別

https://developer.mozilla.org/en-US/docs/Mozilla/Projects/SpiderMonkey/Build_Documentation

optimized ビルド

```
cd js/src
cp configure.in configure
chmod 0755 configure
mkdir build_OPT.OBJ
cd build_OPT.OBJ
../configure
make
```

debug ビルド

```
cd js/src
cp configure.in configure
chmod 0755 configure
mkdir build_DBG.OBJ
cd build_DBG.OBJ
../configure --enable-debug \
  --disable-optimize
make
```

js というバイナリができあがる

js shell

js shell (1/5)

ビルドできたら動かしてみる

```
./dist/bin/js
```

JavaScript を打ち込むと実行できる

```
js> 1 + 2
3
js> [...!+[]+[]].reduce(_=>_+_,+{}+[])
"NaNNaNNaNNaNNaNNaNNaNNaNNaNNaNNaNNaNNaNNaNNaN"
```

js shell (2/5)

テスト用の関数がいろいろある
help() で一覧

```
js> help()
JavaScript-C57.0a1
version([number])
  Get or force a script compilation version number.
options([option ...])
  Get or toggle JavaScript options.
load(['foo.js' ...])
  Load files named by string arguments. Filename is relative to the
      current working directory.
...
```

js shell (3/5)

便利なテスト用関数

`load()`	ファイルをロードして実行
`print()`	値を出力
`parse()`	文字列をパースしてパースツリーを出力
`dis()`	関数のバイトコードを出力
`parseRegExp()`	正規表現をパースしてパースツリーを出力
`disRegExp()`	正規表現のバイトコードを出力

js shell (4/5)

パースツリーの見方 - parse()

```
js> parse("for (let x of y) { x() }");
(STATEMENTLIST [(LEXICALSCOPE [x]
                  (FOR (FOROF (LET [x])
                              #NULL
                              y)
                       (LEXICALSCOPE []
                         (STATEMENTLIST [(SEMI (CALL [x]))])))))])
```

ノードの種類

https://dxr.mozilla.org/mozilla-central/rev/

13d241d08912be31884f9d0d0e805b25343d6c0a/js/src/frontend/ParseNode.h#207

js shell (5/5)

バイトコードの見方 - dis()

```
js> dis(x => x * 2)
dis(x => x * 2)
flags: LAMBDA EXPRESSION_CLOSURE ARROW
loc     op
-----   --
main:
00000:  getarg 0                        # x
00003:  int8 2                          # x 2
00005:  mul                             # (x * 2)
00006:  return                          #
00007:  retrval                         # !!! UNREACHABLE !!!

Source notes:
 ofs line    pc  delta desc     args
---- ---- ----- ------ -------- ------
  0:    3     0 [   0] colspan 9
  2:    3     6 [   6] colspan 5
```

リファレンス

https://developer.mozilla.org/en-US/docs/Mozilla/Projects/SpiderMonkey/Internals/Bytecode

変更すべき箇所の探しかた

変更すべき箇所の探しかた (1/2)

- デバッガで実行を追う
  - クラッシュなどのバグ修正では一番簡単
  - debug ビルドを使うと動きが追いやすい
- プロファイラを使う
  - ボトルネックを調べる
  - 高速化, 最適化向け
  - opt ビルドを使う (debug ビルドは余分なコードが動いている)
- ソフト内蔵のデバッグ用の機能を使う
  - テスト用の関数とか

変更すべき箇所の探しかた (2/2)

- 関連してそうなテキストでソースコードを検索
  - UI のラベルとかメッセージとか
  - http://dxr.mozilla.org/
  - http://searchfox.org/
- 既に修正された関連する Bug のパッチを見る
  - 関連する修正では似たような部分を触っているハズ
  - `https://bugzilla.mozilla.org/`
- 人に聞く
  - 聞けば分かる事なら聞いてしまうのも手
  - https://wiki.mozilla.org/IRC
  - 英語の勉強にもなるかも
  - 時間帯に注意 (だいたい返事はすぐには来ない)

例題 A - パフォーマンス改善

例題 A - パフォーマンス改善 (1/5)

プロファイラを使った例

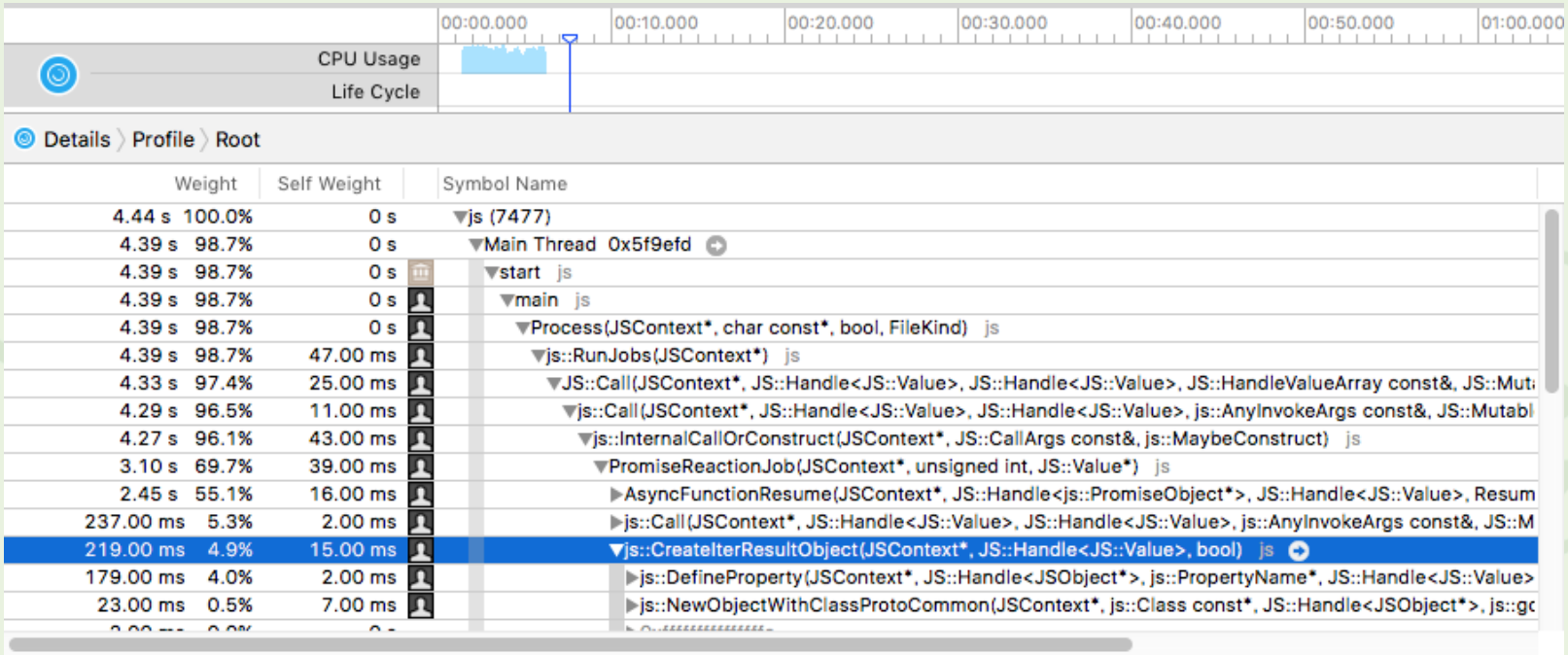
お題: あるテストケースがとても遅い

戦略

1. 時間がかかっているコードを探す
2. その中で手が加えられそうなものを選ぶ
3. 時間がかかる理由を調べる

例題 A - パフォーマンス改善 (2/5)

テストケースを js shell で走らせてプロファイラを使って時間のかかっている関数を調べる
macOS の場合



Instruments.app の Time Profiler

例題 A - パフォーマンス改善 (2/5)

Linux の場合

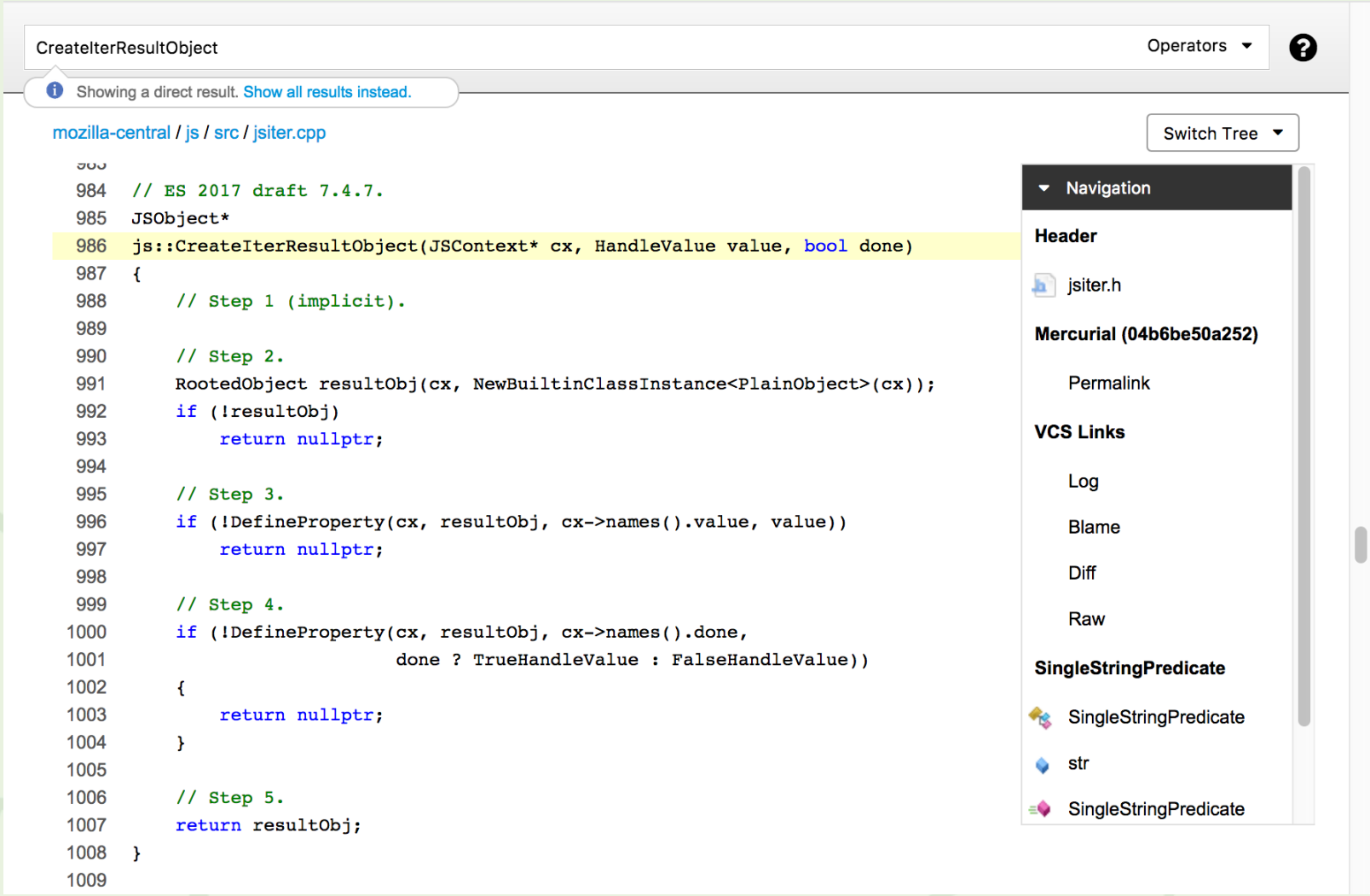
```
$ sudo perf record -F 999 -g -- ./dist/bin/js test.js
$ sudo perf report
```

```
Samples: 58  of event 'cycles:p', Event count (approx.): 484935868688
  Children      Self  Command    Shared Object       Symbol
-  100.00%     0.00%  js         js                  [.] main
   - main
      - 50.44% js::RunJobs
           JS::Call
           js::Call
         - js::InternalCallOrConstruct
            - 50.44% PromiseReactionJob
                 AsyncFunctionResume
                 js::CallSelfHostedFunction
                 js::Call
                 js::InternalCallOrConstruct
                 js::RunScript
                 js::jit::EnterBaselineMethod
                 EnterBaseline
                 0xf43fe5438aa
               + 0xf43fe54fd73
For a higher level overview, try: perf report --sort comm,dso
```

perf report

例題 A - パフォーマンス改善 (4/5)

遅い関数が分かったらその定義箇所を探す



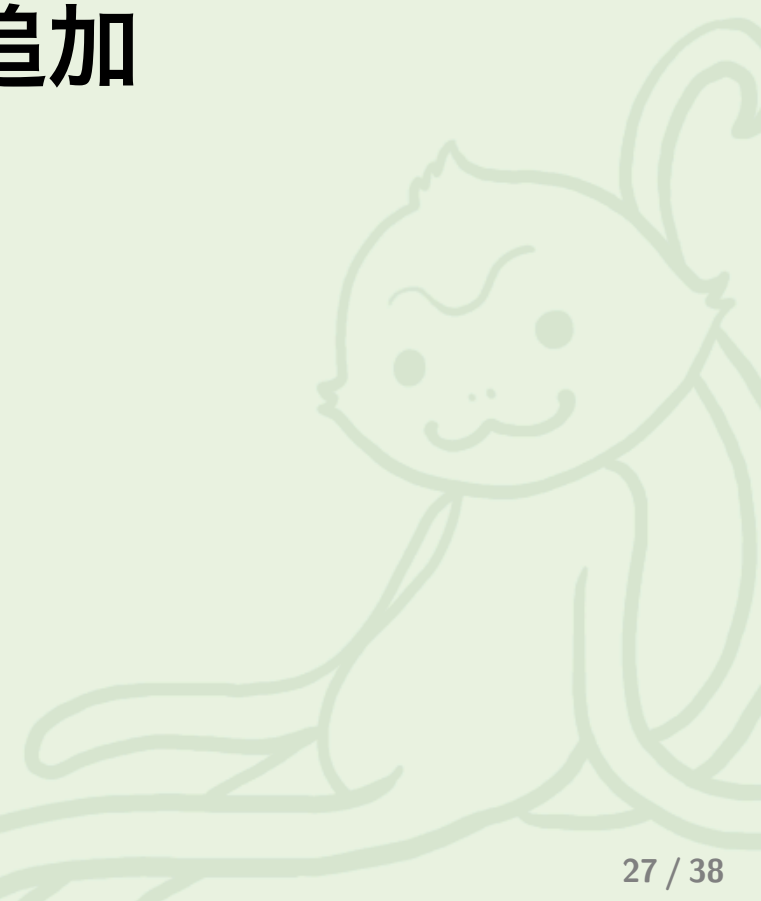
DXR で関数の定義を検索

例題 A - パフォーマンス改善 (5/5)

コードを読んだりデバッガで実行を追ったりして遅い原因と改善方法を考える

```
$ lldb dist/bin/js
...
(lldb) b CreateIterResultObject
(lldb) run testcase.js
...
Process 30661 stopped
...
   1046     // Step 1 (implicit).
   1047
   1048     // Step 2.
-> 1049     RootedObject resultObj(cx, NewBuiltinClassInstance<
          PlainObject>(cx));
   1050     if (!resultObj)
   1051         return nullptr;
   1052
(lldb)
```

例題 B - 機能追加



例題 B - 機能追加 (1/6)

テスト用関数を使った例

お題: 新しい二項演算子を足す

手順

1. Tokenizer を新しいトークンに対応させる
2. Parser で新しいトークンからノードを作る
3. Bytecode Compiler で新しいノードからバイトコードを作る
4. Interpreter を新しいバイトコードに対応させる
5. (JIT を新しいバイトコードに対応させる)

例題 B - 機能追加 (2/6)

似た挙動をする既存の演算子がどう扱われているか見る

```
js> parse("x ** y");
(STATEMENTLIST [(SEMI (POW [x
                            y]))])
```

** 演算子は PNK_POW ノードになる

```
js> dis(() => x ** y);
...
00010:  pow                                 # (x ** y)
...
```

** 演算子は op コード JSOP_POW になる

⇒ PNK_POW, JSOP_POW を生成する所と消費する所をマネすればいい

例題 B - 機能追加 (3/6) - Tokenizer

```
    /* \
     * Binary operators. \
     * These must be in the same order as TOK_OR and friends in
         TokenStream.h. \
     */ \
...
    F(POW) \
```

js/src/frontend/ParseNode.h

PNK_POW は TOK_POW に対応している

```
    switch (c) {
...
      case '*':
        if (matchChar('*'))
            tp->type = matchChar('=') ? TOK_POWASSIGN : TOK_POW;
        else
            tp->type = matchChar('=') ? TOK_MULASSIGN : TOK_MUL;
        goto out;
```

js/src/frontend/TokenStream.cpp

例題 B - 機能追加 (4/6) - Parser

二項演算子のパースは優先順位と対応する op コードを指定するだけで動く

```
static const JSOp ParseNodeKindToJSOp[] = {
...
    JSOP_POW,
...
static const int PrecedenceTable[] = {
...
    11  /* PNK_POW */
```

js/src/frontend/Parser.cpp

例題 B - 機能追加 (5/6) - Bytecode Compiler

結合則に合わせて Left か Right を呼ぶ

```
case PNK_MOD:
  if (!emitLeftAssociative(pn))
      return false;
  break;

case PNK_POW:
  if (!emitRightAssociative(pn))
      return false;
  break;
```

js/src/frontend/BytecodeEmitter.cpp

必要ならその実装も書き換える

例題 B - 機能追加 (6/6) - Interpreter

スタックから値を 2 つ取り出して, 処理してスタックに返している

```
CASE(JSOP_POW)
{
    ReservedRooted<Value> lval(&rootValue0, REGS.sp[-2]);
    ReservedRooted<Value> rval(&rootValue1, REGS.sp[-1]);
    MutableHandleValue res = REGS.stackHandleAt(-2);
    if (!math_pow_handle(cx, lval, rval, res))
        goto error;
    REGS.sp--;
}
END_CASE(JSOP_POW)
```

js/src/vm/Interpreter.cpp

テスト

テスト (1/2)

- 既存の機能を壊してないか調べる
- 追加/変更した機能が意図どおりに動くか調べる

テスト (2/2)

自動テスト

- jstests (言語仕様関連)

```
cd path/to/mozilla-unified/js/src/tests
./jstests.py
```

- jit-test (JIT 関連)

```
cd path/to/mozilla-unified/js/src/jit-test
./jit_test.py
```

その他ヒント

https://developer.mozilla.org/en-US/docs/Mozilla/Projects/SpiderMonkey/Hacking_Tips

Happy Hacking!