《Chrome V8 源码》37. String.prototype.match 源码分析

1 介绍

字符串是 JavaScript 中的重要数据类型，其重要性不仅体现在字符串是应用最多最广泛的数据类型，更体现在V8中使用了大量的技术手段来修饰和优化字符串的操作。接下来的几篇文章将集中讲解字符串的相关操作。本文先讲解 String.prototype.match 的源码以及相关数据结构，再通过测试用例演示 String.prototype.match 的调用、加载和执行过程。

注意 （1）Sea of Nodes 是本文的先导知识，请参考 Cliff 1993年发表的论文 From Quads to Graphs。（2）本文所用环境为：V8 7.9、win10 x64、VS2019。

2 String.prototype.match 源码

测试用例代码如下：

```
var str="1 plus 2 equal 3";
str.match(/\d+/g);
```

match() 采用 TF_BUILTIN 实现，concet() 在 V8 中的函数名是 StringPrototypeMatch，编号是 591，源码如下：

```
1.  TF_BUILTIN(StringPrototypeMatch, StringMatchSearchAssembler) {
2.  TNode<Object> receiver = CAST(Parameter(Descriptor::kReceiver));
```

```
3.  TNode<Object> maybe_regexp = CAST(Parameter(Descriptor::kRegexp));
4.  TNode<Context> context = CAST(Parameter(Descriptor::kContext));
5.  Generate(kMatch, "String.prototype.match", receiver, maybe_regexp, context);}
6.  //分隔..............
7.  void Generate(Variant variant, const char* method_name,TNode<Object> receiver,
TNode<Object> maybe_regexp, TNode<Context> context) {
8.    Label call_regexp_match_search(this);
9.     Builtins::Name builtin;
10.      Handle<Symbol> symbol;
11.      DescriptorIndexNameValue property_to_check;
12.      if (variant == kMatch) {
13.        builtin = Builtins::kRegExpMatchFast;
14.        symbol = isolate()->factory()->match_symbol();
15.        property_to_check = DescriptorIndexNameValue{
16.            JSRegExp::kSymbolMatchFunctionDescriptorIndex,
17.            RootIndex::kmatch_symbol, Context::REGEXP_MATCH_FUNCTION_INDEX};
18.      } else {//省略.....................
19.     }
20.      RequireObjectCoercible(context, receiver, method_name);//省
略.................
21.      { RegExpBuiltinsAssembler regexp_asm(state());
22.        TNode<String> receiver_string = ToString_Inline(context, receiver);
23.        TNode<NativeContext> native_context = LoadNativeContext(context);
24.        TNode<HeapObject> regexp_function = CAST(
25.            LoadContextElement(native_context,
Context::REGEXP_FUNCTION_INDEX));
26.        TNode<Map> initial_map = CAST(LoadObjectField(
27.            regexp_function, JSFunction::kPrototypeOrInitialMapOffset));
28.        TNode<Object> regexp = regexp_asm.RegExpCreate(
29.            context, initial_map, maybe_regexp, EmptyStringConstant());
30.        Label fast_path(this), slow_path(this);
31.        regexp_asm.BranchIfFastRegExp(context, CAST(regexp), initial_map,
32.            PrototypeCheckAssembler::kCheckPrototypePropertyConstness,
33.            property_to_check, &fast_path, &slow_path);
34.        BIND(&fast_path);
35.        Return(CallBuiltin(builtin, context, regexp, receiver_string));
36.        BIND(&slow_path);
37.        {
38.          TNode<Object> maybe_func = GetProperty(context, regexp, symbol);
39.          Callable call_callable = CodeFactory::Call(isolate());
40.          Return(CallJS(call_callable, context, maybe_func, regexp,
41.                        receiver_string));
42.        } } }
```

上述代码中，第 1-5 行是 match() 的入口函数；Generate()（第 7 行代码）用于实现 match 功能，参数 variant 的值只能是 Match 或 Search，这说明了 Search 也由 Generate() 实现。参数 receiver 是字符串（测试用例中的 str），maybe_regexp 是正则字符串（测试用例中的 /\d+/g）；

第 13-17 行代码准备 Builtins::kRegExpMatchFast、symbol 和 property_to_check 三个参数，其中 kRegExpMatchFast 和 symbol 在快速正则时会被用到；

第 22 行代码把 receiver 转换成字符串并存储到 receiver_string 中；

第 23-28 行代码使用字符串（/\d+/g）创建正则表达式 regexp；

第 31 行代码判断是否满足快速正则匹配的使用条件，如果满足则执行第 35 行代码，否则执行第 36-40 行代码；
第 35 行代码执行快速正则匹配；**提示：**使用 Builtin 实现的正则叫做快速正则匹配；
第 36-40 行代码执行慢速正则匹配。
下面说明 Generate() 中的重要函数：

（1） Builtins::kRegExpMatchFast 用于实现快速正则匹配，源码如下：

```
1.  TF_BUILTIN(RegExpMatchFast, CodeStubAssembler) {
2.    compiler::CodeAssemblerState* state_ = state();  compiler::CodeAssembler
ca_(state());
3.    TNode<Context> parameter0 = UncheckedCast<Context>
(Parameter(Descriptor::kContext));
4.    USE(parameter0);
5.    compiler::TNode<JSRegExp> parameter1 = UncheckedCast<JSRegExp>
(Parameter(Descriptor::kReceiver));
6.    USE(parameter1);
7.    compiler::TNode<String> parameter2 = UncheckedCast<String>
(Parameter(Descriptor::kString));
8.    USE(parameter2);
9.    compiler::CodeAssemblerParameterizedLabel<Context, JSRegExp, String>
block0(&ca_, compiler::CodeAssemblerLabel::kNonDeferred);
10.      ca_.Goto(&block0, parameter0, parameter1, parameter2);
11.    if (block0.is_used()) {
12.      compiler::TNode<Context> tmp0;
13.      compiler::TNode<JSRegExp> tmp1;
14.      compiler::TNode<String> tmp2;
15.      ca_.Bind(&block0, &tmp0, &tmp1, &tmp2);
16.      ca_.SetSourcePosition("../../../src/builtins/regexp-match.tq", 27);
17.      compiler::TNode<Object> tmp3;
18.      USE(tmp3);
19.      tmp3 = FastRegExpPrototypeMatchBody_322(state_, compiler::TNode<Context>
{tmp0}, compiler::TNode<JSRegExp>{tmp1}, compiler::TNode<String>{tmp2});
20.      CodeStubAssembler(state_).Return(tmp3);
21.    }
22.  }
```

上述代码中，第 3-8 行定义上下文（parameter0）、正则（parameter1）以及字符串 （parameter2）三个参数；
第 10-14 行代码 Goto 用于跳转到 block0。其中 tmp1 表示 parameter1，tmp2 表示 parameter2;
第 19 行代码 FastRegExpPrototypeMatchBody_322() 是入口函数，在该函数中调用 RegExpBuiltinsAssembler::RegExpPrototypeMatchBody 完成正则匹配，后续文章单独讲解。

（2） BranchIfFastRegExp 判断是否符合满足快速正则条件，源码如下：

```
1.  void RegExpBuiltinsAssembler::BranchIfFastRegExp(/*省略...*/) {
2.    CSA_ASSERT(this, TaggedEqual(LoadMap(object), map));
3.    GotoIfForceSlowPath(if_ismodified);
4.    TNode<NativeContext> native_context = LoadNativeContext(context);
5.    GotoIf(IsRegExpSpeciesProtectorCellInvalid(native_context), if_ismodified);
6.    TNode<JSFunction> regexp_fun =
```

```
7.        CAST(LoadContextElement(native_context,
Context::REGEXP_FUNCTION_INDEX));
8.    TNode<Map> initial_map = CAST(
9.        LoadObjectField(regexp_fun, JSFunction::kPrototypeOrInitialMapOffset));
10.    TNode<BoolT> has_initialmap = TaggedEqual(map, initial_map);
11.    GotoIfNot(has_initialmap, if_ismodified);
12.    TNode<Object> last_index = FastLoadLastIndexBeforeSmiCheck(CAST(object));
13.    GotoIfNot(TaggedIsPositiveSmi(last_index), if_ismodified);
14.    // Verify the prototype.
15.    TNode<Map> initial_proto_initial_map = CAST(
16.        LoadContextElement(native_context,
Context::REGEXP_PROTOTYPE_MAP_INDEX));
17.    DescriptorIndexNameValue properties_to_check[2];
18.    int property_count = 0;
19.    properties_to_check[property_count++] = DescriptorIndexNameValue{
20.        JSRegExp::kExecFunctionDescriptorIndex, RootIndex::kexec_string,
21.        Context::REGEXP_EXEC_FUNCTION_INDEX};
22.    if (additional_property_to_check) {
23.      properties_to_check[property_count++] = *additional_property_to_check;
24.    }
25.    PrototypeCheckAssembler prototype_check_assembler(
26.        state(), prototype_check_flags, native_context,
initial_proto_initial_map,
27.        Vector<DescriptorIndexNameValue>(properties_to_check, property_count));
28.    TNode<HeapObject> prototype = LoadMapPrototype(map);
29.    prototype_check_assembler.CheckAndBranch(prototype, if_isunmodified,
30.                                             if_ismodified);
31.  }
```

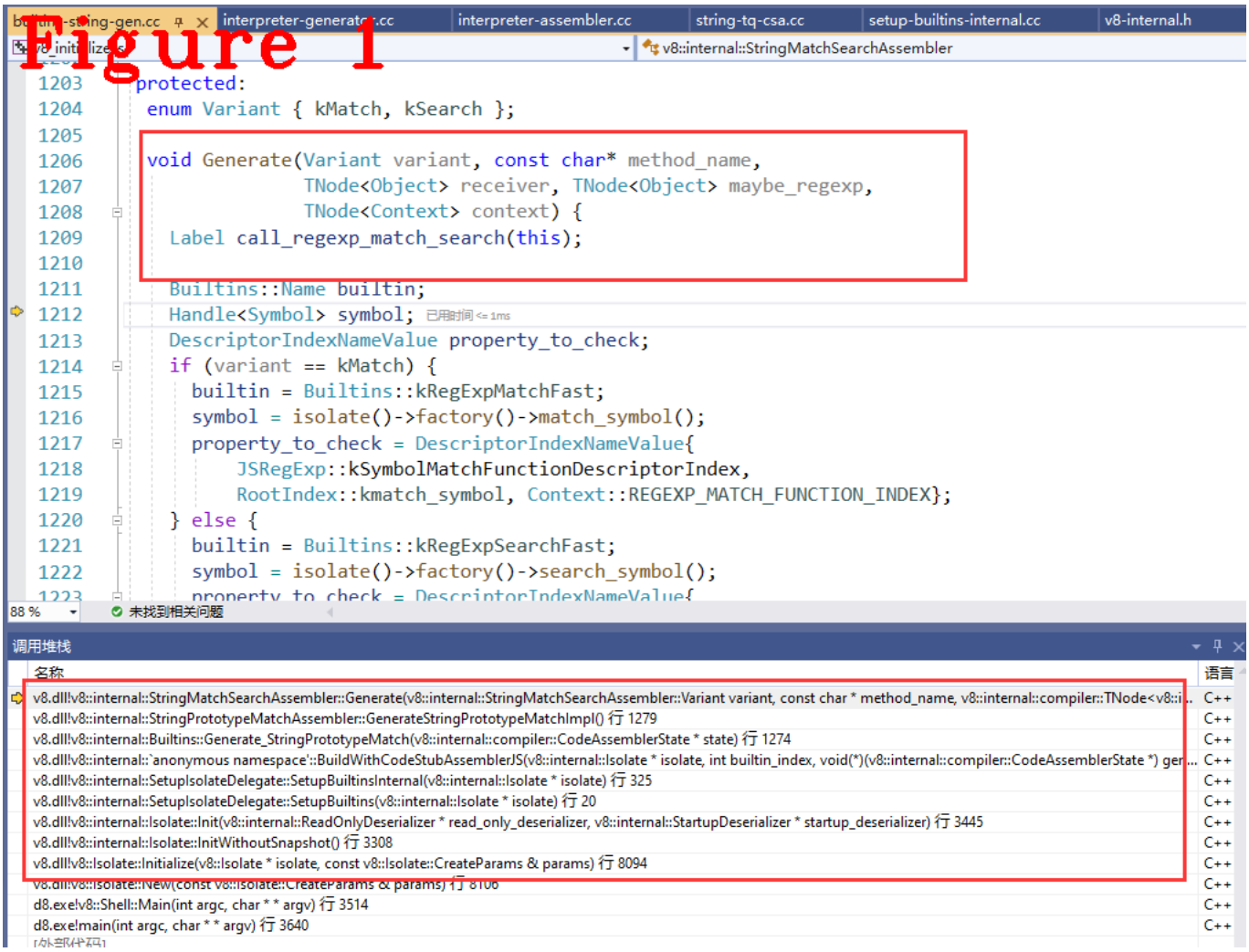
上述代码中 if_ismodified 代表慢速正则；第 3 行代码 GotoIfForceSlowPath 根据 V8_ENABLE_FORCE_SLOW_PATH 判断是否使用慢速正则；
第 2 行代码检测正则表达式对象map的tag标记；
第 10-11 行代码判断正则表式对象的 tag 与 native_context中的 regexp_fun的tag 是否相等；
第 15-29 行代码检测 prototype 属性，并根据检测结果决定是否使用快速正则。

（3） MaybeCallFunctionAtSymbol 方法源码如下：

```
1.  void StringBuiltinsAssembler::MaybeCallFunctionAtSymbol(
2.      Node* const context, Node* const object, Node* const maybe_string,
3.      Handle<Symbol> symbol,
4.      DescriptorIndexNameValue additional_property_to_check,
5.      const NodeFunction0& regexp_call, const NodeFunction1& generic_call) {
6.    Label out(this);
7.    // Smis definitely don't have an attached symbol.
8.    GotoIf(TaggedIsSmi(object), &out);
9.    {
10.      Label stub_call(this), slow_lookup(this);
11.      GotoIf(TaggedIsSmi(maybe_string), &slow_lookup);
12.      GotoIfNot(IsString(maybe_string), &slow_lookup);
13.      RegExpBuiltinsAssembler regexp_asm(state());
14.      regexp_asm.BranchIfFastRegExp(
15.          CAST(context), CAST(object), LoadMap(object),
```

```
16.          PrototypeCheckAssembler::kCheckPrototypePropertyConstness,
17.          additional_property_to_check, &stub_call, &slow_lookup);
18.      BIND(&stub_call);
19.  .
20.      regexp_call();
21.      BIND(&slow_lookup);
22.    }
23.    GotoIf(IsNullOrUndefined(object), &out);
24.    TNode<Object> const maybe_func = GetProperty(context, object, symbol);
25.    GotoIf(IsUndefined(maybe_func), &out);
26.    GotoIf(IsNull(maybe_func), &out);
27.    // Attempt to call the function.
28.    generic_call(maybe_func);
29.    BIND(&out);
30.  }
```

上述代码中第 11-12 行判断正则表达式是否为 SMI 或 String，判断结果为真则执行慢速正则；
第 14 行代码 BranchIfFastRegExp 判断原型链属性是否满足快速正则条件；
第 23、25、26 行代码分别判断字符串是否为空、正则表达式是否未定义或为空。
图 1 给出了 Generate 的函数调用堆栈。

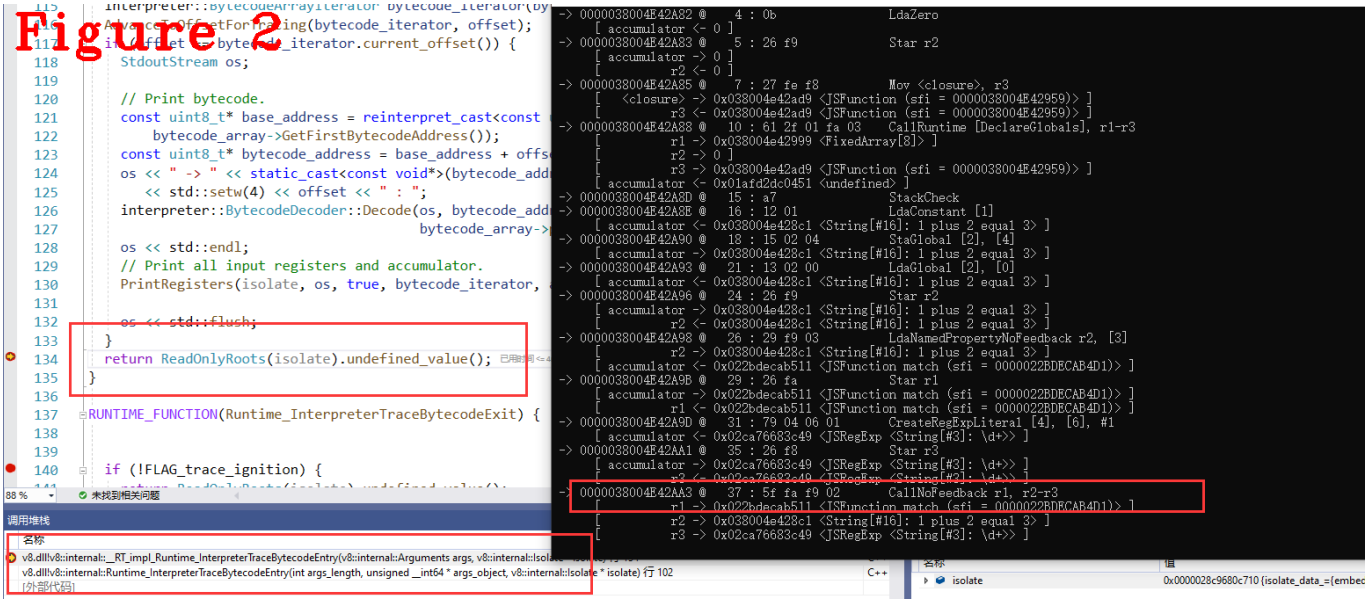


3 String.prototype.match 测试

测试用例的字节码如下：

```
1.  //省略.............
2.  0000038004E42A8E @   16 : 12 01             LdaConstant [1]
3.  0000038004E42A90 @   18 : 15 02 04          StaGlobal [2], [4]
4.  0000038004E42A93 @   21 : 13 02 00          LdaGlobal [2], [0]
5.  0000038004E42A96 @   24 : 26 f9             Star r2
6.  0000038004E42A98 @   26 : 29 f9 03          LdaNamedPropertyNoFeedback r2, [3]
7.  0000038004E42A9B @   29 : 26 fa             Star r1
8.  0000038004E42A9D @   31 : 79 04 06 01       CreateRegExpLiteral [4], [6], #1
9.  0000038004E42AA1 @   35 : 26 f8             Star r3
10.  0000038004E42AA3 @   37 : 5f fa f9 02       CallNoFeedback r1, r2-r3
11.  0000038004E42AA7 @   41 : 15 05 07          StaGlobal [5], [7]
12.  0000038004E42AAA @   44 : 13 06 09          LdaGlobal [6], [9]
13.  0000038004E42AAD @   47 : 26 f9             Star r2
14.  0000038004E42AAF @   49 : 29 f9 07          LdaNamedPropertyNoFeedback r2,
[7]
15.  0000038004E42AB2 @   52 : 26 fa             Star r1
16.  0000038004E42AB4 @   54 : 13 05 02          LdaGlobal [5], [2]
17.  0000038004E42AB7 @   57 : 26 f8             Star r3
18.  0000038004E42AB9 @   59 : 5f fa f9 02       CallNoFeedback r1, r2-r3
19.  0000038004E42ABD @   63 : 26 fb             Star r0
20.  0000038004E42ABF @   65 : ab                Return
21.  Constant pool (size = 8)
22.  0000038004E429F9: [FixedArray] in OldSpace
23.   - map: 0x01afd2dc0169 <Map>
24.   - length: 8
25.  0: 0x038004e42999 <FixedArray[8]>
26.  1: 0x038004e428c1 <String[#16]: 1 plus 2 equal 3>
27.  2: 0x038004e428a9 <String[#3]: str>
28.  3: 0x022bdecab4b9 <String[#5]: match>
29.  4: 0x038004e428f9 <String[#3]: \d+>
30.  5: 0x038004e428e1 <String[#3]: res>
31.  6: 0x022bdecb3699 <String[#7]: console>
32.  7: 0x022bdecb2cd9 <String[#3]: log>
33.  Handler Table (size = 0)
```

上述代码中，第 2-5 行代码加载 "1 plus 2 equal 3" 到 r2 寄存器；
第 6 行代码获取字符串方法 match，并存储到 r1 寄存器；
第 8 行代码为字符串 \d+ 创建正表达式对象，并存储到 r3 寄存器；
第 10 行代码 CallNoFeedback 调用 match方法（r1 寄存器），并传递 r2、r3 两个参数给 match 方法。
图 2 给出了字节码 CallNoFeedback 的入口，从此处开始跟踪可以看到正则表达的匹配过程。



技术总结

（1） 快速正则是采用 Builtins::kRegExpMatchFast 实现的快速匹配；

（2） 使用快速正则的判断条件包括：字符串类型是否正确、正则表达式的类型、V8_ENABLE_FORCE_SLOW_PATH 等。

好了，今天到这里，下次见。

个人能力有限，有不足与纰漏，欢迎批评指正

微信：qq9123013 备注：v8交流 知乎：www.zhihu.com/people/v8blink

本文由灰豆原创发布

转载出处： https://www.anquanke.com/post/id/263786

安全客 - 有思想的安全新媒体