《Chrome V8 源码》38. replace 技术细节、性能影响因素

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.replace 的源码以及相关数据结构，再通过测试用例演示 String.prototype.replace 的调用、加载和执行过程。我们会看字符串的类型、长度等因素如何影响 replace 的实现方式和效率。
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.replace 源码

测试用例代码如下：

```
var str="Visit Microsoft!Visit Microsoft!";
var res=str.replace("Microsoft","Runoob");
console.log(res);
```

replace() 采用 TF_BUILTIN 实现，replace() 在 V8 中的函数名是 StringPrototypeReplace，编号是 594，源码如下：

```
1.  TF_BUILTIN(StringPrototypeReplace, StringBuiltinsAssembler) {
2.  Label out(this);
3.  TNode<Object> receiver = CAST(Parameter(Descriptor::kReceiver));
4.  Node* const search = Parameter(Descriptor::kSearch);
5.  Node* const replace = Parameter(Descriptor::kReplace);
6.  TNode<Context> context = CAST(Parameter(Descriptor::kContext));
7.  TNode<Smi> const smi_zero = SmiConstant(0);
8.  RequireObjectCoercible(context, receiver, "String.prototype.replace");
9.  MaybeCallFunctionAtSymbol(/*省略.....*/);
10.   TNode<String> const subject_string = ToString_Inline(context, receiver);
11.   TNode<String> const search_string = ToString_Inline(context, search);
12.   TNode<IntPtrT> const subject_length =
LoadStringLengthAsWord(subject_string);
13.   TNode<IntPtrT> const search_length = LoadStringLengthAsWord(search_string);
14.   {
15.     Label next(this);
16.     GotoIfNot(WordEqual(search_length, IntPtrConstant(1)), &next);
17.     GotoIfNot(IntPtrGreaterThan(subject_length, IntPtrConstant(0xFF)), &next);
18.     GotoIf(TaggedIsSmi(replace), &next);
19.     GotoIfNot(IsString(replace), &next);
20.     TNode<Uint16T> const subject_instance_type =
21.         LoadInstanceType(subject_string);
22.     GotoIfNot(IsConsStringInstanceType(subject_instance_type), &next);
23.     GotoIf(TaggedIsPositiveSmi(IndexOfDollarChar(context, replace)), &next);
24.     Return(CallRuntime(Runtime::kStringReplaceOneCharWithString, context,
25.                        subject_string, search_string, replace));
26.     BIND(&next); }
27.   TNode<Smi> const match_start_index =
28.       CAST(CallBuiltin(Builtins::kStringIndexOf, context, subject_string,
29.                        search_string, smi_zero));
30.   {
31.     Label next(this), return_subject(this);
32.     GotoIfNot(SmiIsNegative(match_start_index), &next);
33.     GotoIf(TaggedIsSmi(replace), &return_subject);
34.     GotoIf(IsCallableMap(LoadMap(replace)), &return_subject);
35.     ToString_Inline(context, replace);
36.     Goto(&return_subject);
37.     BIND(&return_subject);
38.     Return(subject_string);
39.     BIND(&next); }
40.   TNode<Smi> const match_end_index = SmiAdd(match_start_index,
SmiFromIntPtr(search_length));
41.   VARIABLE(var_result, MachineRepresentation::kTagged, EmptyStringConstant());
42.   {
43.     Label next(this);
44.     GotoIf(SmiEqual(match_start_index, smi_zero), &next);
45.     TNode<Object> const prefix =
46.         CallBuiltin(Builtins::kStringSubstring, context, subject_string,
47.                     IntPtrConstant(0), SmiUntag(match_start_index));
48.     var_result.Bind(prefix);
49.     Goto(&next);
50.     BIND(&next); }
51.   Label if_iscallablereplace(this), if_notcallablereplace(this);
```

```
52.   GotoIf(TaggedIsSmi(replace), &if_notcallablereplace);
53.   Branch(IsCallableMap(LoadMap(replace)), &if_iscallablereplace,
54.          &if_notcallablereplace);
55.   BIND(&if_iscallablereplace);
56.   {
57.     Callable call_callable = CodeFactory::Call(isolate());
58.     Node* const replacement =
59.         CallJS(call_callable, context, replace, UndefinedConstant(),
60.                search_string, match_start_index, subject_string);
61.     TNode<String> const replacement_string =
62.         ToString_Inline(context, replacement);
63.     var_result.Bind(CallBuiltin(Builtins::kStringAdd_CheckNone, context,
64.                                 var_result.value(), replacement_string));
65.     Goto(&out); }
66.   BIND(&if_notcallablereplace);
67.   {
68.     TNode<String> const replace_string = ToString_Inline(context, replace);
69.     Node* const replacement =
70.         GetSubstitution(context, subject_string, match_start_index,
71.                         match_end_index, replace_string);
72.     var_result.Bind(CallBuiltin(Builtins::kStringAdd_CheckNone, context,
73.                                 var_result.value(), replacement));
74.     Goto(&out);}
75.   BIND(&out);
76.   {
77.     TNode<Object> const suffix =
78.         CallBuiltin(Builtins::kStringSubstring, context, subject_string,
79.                     SmiUntag(match_end_index), subject_length);
80.     TNode<Object> const result = CallBuiltin(
81.         Builtins::kStringAdd_CheckNone, context, var_result.value(), suffix);
82.      Return(result);
83.    }}
```

上述代码第 3 行代码 receiver 代表测试用例字符串 "Visit Microsoft!Visit Microsoft!"；
第 4 行代码 search 代表测试用例字符串 "Microsoft"；
第 5 行代码 replace 代表测试用例字符串 "Runoob"；
第 9 行代码当 search 为正则表达式时，使用 MaybeCallFunctionAtSymbol() 实现 replace 功能。
下面将 replace 源码划分为五个子功能单独说明：
（1） 功能 1：receiver 长度大于 0xFF、search 长度大于 1 以及 replace 为 ConsString，这三个条件同时成立时
第 10-13 行代码转换 search 和 replace 的数据类型，并计算他们的长度；
第 16 行代码判断 search 的长度是否等于 1，不等于则跳转到第 26 行；
第 17 行代码判断 receiver 的长度是否大于 0xFF，小于等于则跳转到第 26 行；
第 18-19 行判断 replace 是否为小整数或者 replace 不是字符串，结果为真则跳转到第 26 行；
第 20-22 行判断 replace 是否为 ConsString 类型，结果为假则跳转到第 26 行；**提示** ConsString不是一个独立的字符串，它是使用指针把两个字符串拼接在一起的字符串对。在 V8 中，两个字符串相加的结果常是 ConsString 类型的字符串对。
第 23 行判断 PositiveSmi 类型；
第 24 行采用 Runtime::kStringReplaceOneCharWithString() 完成 replace。
（2） 功能 2：计算 receiver 中的前缀字符串

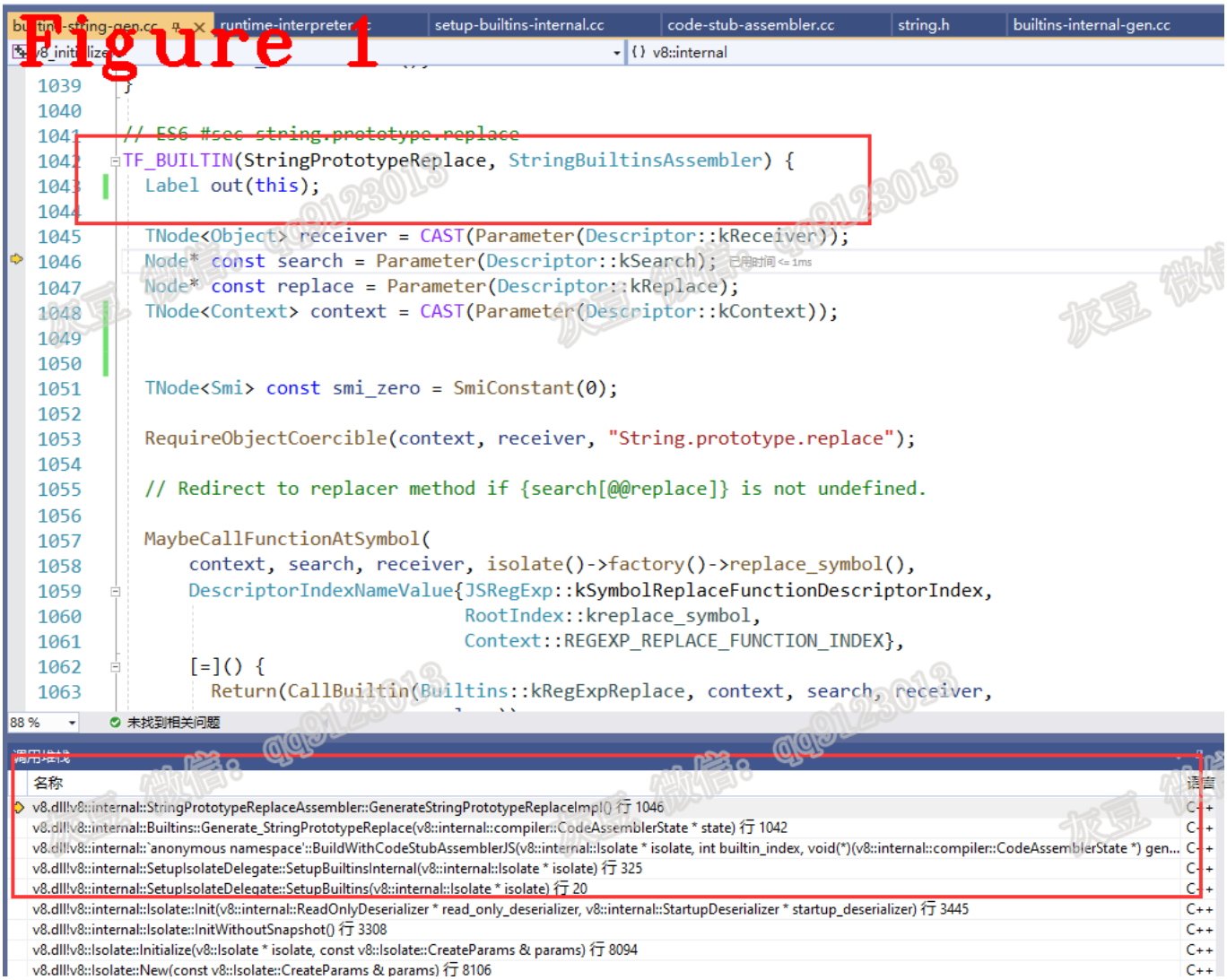
第 27-32 行计算 search的第一个字符在 receiver 中的位置 match_start_index，如果 match_start_index 不是负数则跳转到 39 行；
第 33-35 行判断 replace 是否为 SMI 或 函数，是则跳转到 37 行；
第 40 行计算 match_end_index；
第 44 行如果 match_start_index = 0（也就是 search[0]=receiver[0]），则跳转到 49 行；
第 45 行取出 receiver 中 match_start_index 位置之前的字符，保存为 prefix；也就是获取测试用例中的 “Visit ” 字符串；
第 50-54 行判断 replace 是否为 SMI 或 函数，根据判断结果跳转到相应的行号。
（3） 功能 3：replace 是函数类型
第 58-63 行计算 replace 的结果，将该结果拼接在 prefix 后面组成新的字符串 var_result;
（4） 功能 4：replace 是字符串类型
第 58-72 行将 replace 拼接在 prefix 后面组成新的字符串 var_result;
（5） 计算 receiver 中的后缀字符串
第 77-82 行取出 receiver 中 match_end_index 之后的字符串 suffix，将 suffix 拼接在 var_result 后面组成并返回新的字符串，replace 完毕。

下面简单说明 replace 中使用的 runtime 方法：

```
1.  RUNTIME_FUNCTION(Runtime_StringReplaceOneCharWithString) {
2.    HandleScope scope(isolate);
3.    DCHECK_EQ(3, args.length());
4.    CONVERT_ARG_HANDLE_CHECKED(String, subject, 0);
5.    CONVERT_ARG_HANDLE_CHECKED(String, search, 1);
6.    CONVERT_ARG_HANDLE_CHECKED(String, replace, 2);
7.    const int kRecursionLimit = 0x1000;
8.    bool found = false;
9.    Handle<String> result;
10.    if (StringReplaceOneCharWithString(isolate, subject, search, replace,
&found,
11.                                       kRecursionLimit).ToHandle(&result)) {
12.      return *result;
13.    }
14.    if (isolate->has_pending_exception())
15.      return ReadOnlyRoots(isolate).exception();
16.    subject = String::Flatten(isolate, subject);
17.    if (StringReplaceOneCharWithString(isolate, subject, search, replace,
&found,
18.                                       kRecursionLimit).ToHandle(&result)) {
19.      return *result;
20.    }
21.    if (isolate->has_pending_exception())
22.      return ReadOnlyRoots(isolate).exception();
23.    return isolate->StackOverflow();
24.  }
```

上述代码中第 10 执行 StringReplaceOneCharWithString 方法，该方法实现了 replace 功能;
第 14 行代码检测到异常情况时，先执行 String::Flatten，将ConsString字符处理成为单一字符串之后再次执行 StringReplaceOneCharWithString 方法。

图1给出 StringPrototypeReplace 的函数调用堆栈。



String.prototype.replace 测试

测试用例的字节码如下：

```
1.  //省略........
2.  000001A2EE982AC6 @   16 : 12 01             LdaConstant [1]
3.  000001A2EE982AC8 @   18 : 15 02 04          StaGlobal [2], [4]
4.  000001A2EE982ACB @   21 : 13 02 00          LdaGlobal [2], [0]
5.  000001A2EE982ACE @   24 : 26 f9             Star r2
6.  000001A2EE982AD0 @   26 : 29 f9 03          LdaNamedPropertyNoFeedback r2, [3]
7.  000001A2EE982AD3 @   29 : 26 fa             Star r1
8.  000001A2EE982AD5 @   31 : 12 04             LdaConstant [4]
9.  000001A2EE982AD7 @   33 : 26 f8             Star r3
10.  000001A2EE982AD9 @   35 : 12 05             LdaConstant [5]
11.  000001A2EE982ADB @   37 : 26 f7             Star r4
12.  000001A2EE982ADD @   39 : 5f fa f9 03       CallNoFeedback r1, r2-r4
13.  000001A2EE982AE1 @   43 : 15 06 06          StaGlobal [6], [6]
14.  000001A2EE982AE4 @   46 : 13 07 08          LdaGlobal [7], [8]
15.  000001A2EE982AE7 @   49 : 26 f9             Star r2
16.  000001A2EE982AE9 @   51 : 29 f9 08          LdaNamedPropertyNoFeedback r2,
```

```
[8]
17.  000001A2EE982AEC @   54 : 26 fa             Star r1
18.  000001A2EE982AEE @   56 : 13 06 02          LdaGlobal [6], [2]
19.  000001A2EE982AF1 @   59 : 26 f8             Star r3
20.  000001A2EE982AF3 @   61 : 5f fa f9 02       CallNoFeedback r1, r2-r3
21.  000001A2EE982AF7 @   65 : 26 fb             Star r0
22.  000001A2EE982AF9 @   67 : ab                Return
23.  Constant pool (size = 9)
24.  000001A2EE982A29: [FixedArray] in OldSpace
25.   - map: 0x022d5e100169 <Map>
26.   - length: 9
27.  0: 0x01a2ee9829c9 <FixedArray[8]>
28.  1: 0x01a2ee9828c1 <String[#32]: Visit Microsoft!Visit Microsoft!>
29.  2: 0x01a2ee9828a9 <String[#3]: str>
30.  3: 0x02749a2ab821 <String[#7]: replace>
31.  4: 0x01a2ee982909 <String[#9]: Microsoft>
32.  5: 0x01a2ee982929 <String[#6]: Runoob>
33.  6: 0x01a2ee9828f1 <String[#3]: res>
34.  7: 0x02749a2b3699 <String[#7]: console>
35.  8: 0x02749a2b2cd9 <String[#3]: log>
```

上述代码中，第 2-5 行读取字符串 “Visit Microsoft!Visit Microsoft!” 并保存到 r2 寄存器中；
第 6-7 行获取 replace 函数地址并保存到 r1 寄存器中；
第 8-11 行读取 “Microsoft” 和 “Runoob” 并分别保存到 r3 和 r4 寄存器中；
第 12 行调用 repalce 方法；
图 2 给出了 CallNoFeedback 的调用堆栈。

技术总结

（1） receiver 长度大于 0xFF、search 长度大于 1 以及 replace为ConsString，三个条件同时成立时采用低效率的 runtime 方式处理； **（2）** replace 的原理是计算前、后缀，并与 newvalue 拼接组成最终结果。

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/263871
安全客 - 有思想的安全新媒体