《Chrome V8 源码》36. String.prototype.concat 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.concat 的源码以及相关数据结构，再通过测试用例演示 String.prototype.concat 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.concat 源码

测试用例代码如下：

```
var txt1 = "he ", txt2="is ", txt3="HuiDou ", txt4=".";
var bio = txt1.concat(txt2,txt3,txt4);
console.log(bio);
```

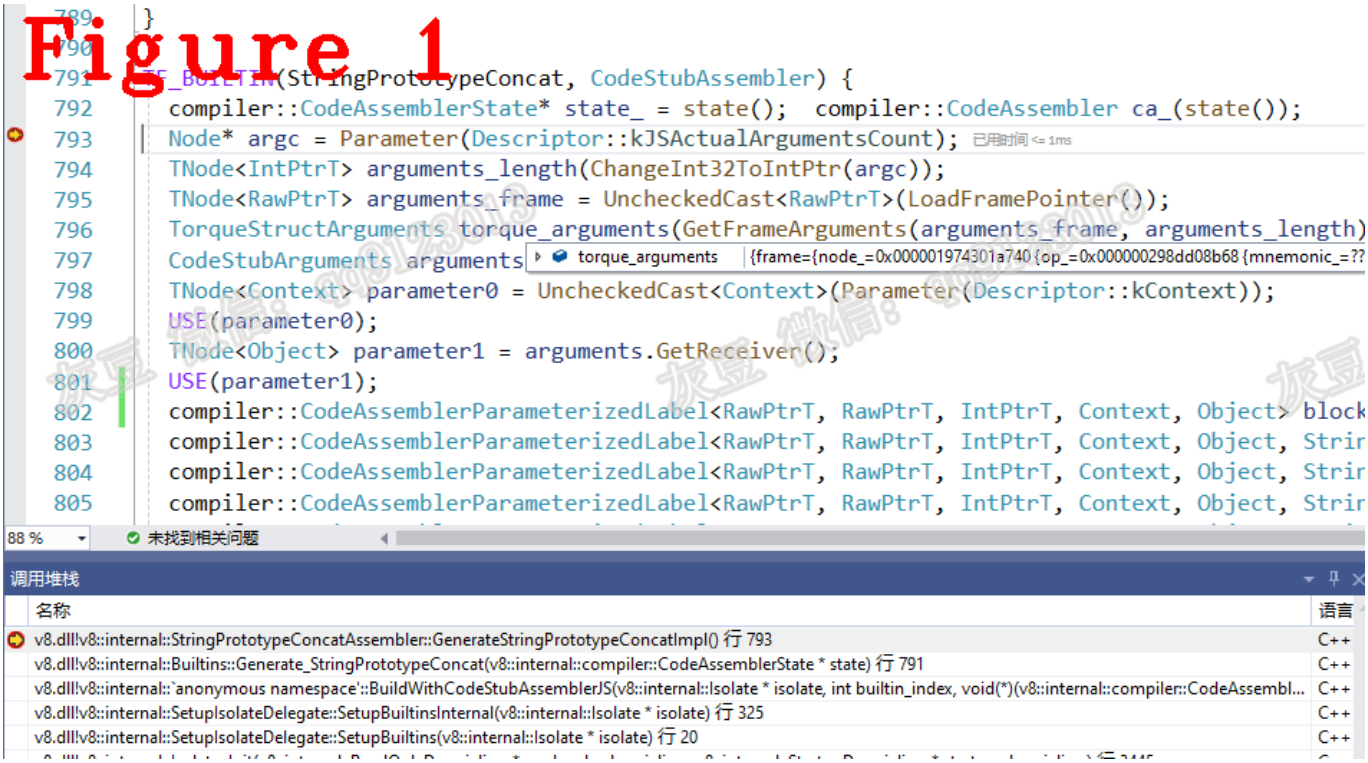
concet() 是用 TF_BUILTIN 实现的，concet() 在 V8 中的函数名是 StringPrototypeConcat，编号是 888，源码如下：

```
1.  TF_BUILTIN(StringPrototypeConcat, CodeStubAssembler) {
2.  ca_.Goto(&block0, torque_arguments.frame, torque_arguments.base,
torque_arguments.length, parameter0, parameter1);
3.    if (block0.is_used()) {//省略.............
4.      ca_.Bind(&block0, &tmp0, &tmp1, &tmp2, &tmp3, &tmp4);
5.      ca_.SetSourcePosition("../../../src/builtins/string.tq", 128);
6.      compiler::TNode<String> tmp5;
7.      USE(tmp5);
8.      tmp5 = FromConstexpr6String18ATconstexpr_string_156(state_,
"String.prototype.concat");
9.      compiler::TNode<String> tmp6;
10.      USE(tmp6);
11.      tmp6 = CodeStubAssembler(state_).ToThisString(compiler::TNode<Context>
{tmp3}, compiler::TNode<Object>{tmp4}, compiler::TNode<String>{tmp5});
12.      ca_.SetSourcePosition("../../../src/builtins/string.tq", 131);
13.      compiler::TNode<IntPtrT> tmp7;
14.      USE(tmp7);
15.      tmp7 = Convert8ATintptr8ATintptr_1494(state_, compiler::TNode<IntPtrT>
{tmp2});
16.      ca_.SetSourcePosition("../../../src/builtins/string.tq", 132);
17.      compiler::TNode<IntPtrT> tmp8;
18.      USE(tmp8);
19.      tmp8 = FromConstexpr8ATintptr17ATconstexpr_int31_150(state_, 0);
20.      ca_.Goto(&block3, tmp0, tmp1, tmp2, tmp3, tmp4, tmp6, tmp7, tmp8);
21.    }
22.    if (block3.is_used()) {//省略.............
23.      ca_.Bind(&block3, &tmp9, &tmp10, &tmp11, &tmp12, &tmp13, &tmp14, &tmp15,
&tmp16);
24.      compiler::TNode<BoolT> tmp17;
25.      USE(tmp17);
26.      tmp17 = CodeStubAssembler(state_).IntPtrLessThan(compiler::TNode<IntPtrT>
{tmp16}, compiler::TNode<IntPtrT>{tmp15});
27.      ca_.Branch(tmp17, &block1, &block2, tmp9, tmp10, tmp11, tmp12, tmp13,
tmp14, tmp15, tmp16);
28.    }
29.    if (block1.is_used()) {//省略.............
30.      ca_.Bind(&block1, &tmp18, &tmp19, &tmp20, &tmp21, &tmp22, &tmp23, &tmp24,
&tmp25);
31.      ca_.SetSourcePosition("../../../src/builtins/string.tq", 133);
32.      compiler::TNode<Object> tmp26;
33.      USE(tmp26);
34.      tmp26 =
CodeStubAssembler(state_).GetArgumentValue(TorqueStructArguments{compiler::TNode<R
awPtrT>{tmp18}, compiler::TNode<RawPtrT>{tmp19}, compiler::TNode<IntPtrT>{tmp20}},
compiler::TNode<IntPtrT>{tmp25});
35.      compiler::TNode<String> tmp27;
36.      USE(tmp27);
37.      tmp27 =
CodeStubAssembler(state_).ToString_Inline(compiler::TNode<Context>{tmp21},
compiler::TNode<Object>{tmp26});
38.      ca_.SetSourcePosition("../../../src/builtins/string.tq", 134);
39.      compiler::TNode<String> tmp28;
40.      USE(tmp28);
```

```
41.       tmp28 = StringAdd_82(state_, compiler::TNode<Context>{tmp21},
compiler::TNode<String>{tmp23}, compiler::TNode<String>{tmp27});
42.       ca_.SetSourcePosition("../../../src/builtins/string.tq", 132);
43.       ca_.Goto(&block4, tmp18, tmp19, tmp20, tmp21, tmp22, tmp28, tmp24,
tmp25);
44.     }
45.     if (block4.is_used()) {//省略..............
46.       ca_.Bind(&block4, &tmp29, &tmp30, &tmp31, &tmp32, &tmp33, &tmp34, &tmp35,
&tmp36);
47.       compiler::TNode<IntPtrT> tmp37;
48.       USE(tmp37);
49.       tmp37 = FromConstexpr8ATintptr17ATconstexpr_int31_150(state_, 1);
50.       compiler::TNode<IntPtrT> tmp38;
51.       USE(tmp38);
52.       tmp38 = CodeStubAssembler(state_).IntPtrAdd(compiler::TNode<IntPtrT>
{tmp36}, compiler::TNode<IntPtrT>{tmp37});
53.       ca_.Goto(&block3, tmp29, tmp30, tmp31, tmp32, tmp33, tmp34, tmp35,
tmp38);
54.     }
55.     if (block2.is_used()) {//省略..............
56.       ca_.Bind(&block2, &tmp39, &tmp40, &tmp41, &tmp42, &tmp43, &tmp44, &tmp45,
&tmp46);
57.       ca_.SetSourcePosition("../../../src/builtins/string.tq", 136);
58.       arguments.PopAndReturn(tmp44);
59.     }
60.   }
```

上述代码中定义了四个 block（block0-block4），它们的作用是：
block0 创建三个变量：初始字符串 tmp6（即测试用例中的“he”）、拼接总数 tmp7（测试用例中的数量为 3）和完成拼接的数量 tmp8（初始值为 0）；
block1 把两个字符串拼接在一起生成一个新字符串；
block2 返回最终结果；
block3 判断完成拼接的数量是否小于拼接总数，如果小于跳转到 block1，如果不小于跳转到 blcok2；
block4 把完成拼接的数量加 1。
从这些 block 在代码中的分布位置可以看出，他们使用 while 循环方式实现了字符串的拼接。图 1 给出了 StringPrototypeConcat 源码的位置。

Figure 1



下面说明 StringPrototypeConcat 用到的重要函数：

(1) ToThisString（第 11 行代码）的作用是把对象转换成字符串，源码如下：

```
1.  TNode<String> CodeStubAssembler::ToThisString(/*省略*/) {
2.  BIND(&if_valueisnotsmi);//省略....
3.  { TNode<Uint16T> value_instance_type = LoadInstanceType(CAST(value));
4.    Label if_valueisnotstring(this, Label::kDeferred);
5.    Branch(IsStringInstanceType(value_instance_type), &if_valueisstring,
6.           &if_valueisnotstring);
7.     BIND(&if_valueisnotstring);
8.     { Label if_valueisnullorundefined(this, Label::kDeferred);
9.       GotoIf(IsNullOrUndefined(value), &if_valueisnullorundefined);
10.       var_value.Bind(CallBuiltin(Builtins::kToString, context, value));
11.       Goto(&if_valueisstring);
12.       BIND(&if_valueisnullorundefined);
13.       {ThrowTypeError(context, MessageTemplate::kCalledOnNullOrUndefined,
14.                       method_name);}
15.     }
16.   }
17.   BIND(&if_valueissmi);
18.   {var_value.Bind(CallBuiltin(Builtins::kNumberToString, context, value));
19.     Goto(&if_valueisstring);  }
20.   BIND(&if_valueisstring);
21.   return CAST(var_value.value());
```

上述代码中第 2-16 行用于把非 small integer 数据转换成字符串，非 small integer 可以是数组、浮点数等。其中转换操作的具体功能由第 10 行的 kToString 实现；第 17-19 行用于把 small integer 数据转换成字符串，转换操作的具体功能由第 18 行的 kNumberToString 实现。后续文章会单独讲解 kToString 和 kNumberToString 方法。

(2) IntPtrAdd（第 52 行）实现整型数据的加法，源码如下：

```
1.  TNode<WordT> CodeAssembler::IntPtrAdd(SloppyTNode<WordT> left,
2.                                        SloppyTNode<WordT> right) {
3.  intptr_t left_constant;
4.  bool is_left_constant = ToIntPtrConstant(left, &left_constant);
5.  intptr_t right_constant;
6.  bool is_right_constant = ToIntPtrConstant(right, &right_constant);
7.  if (is_left_constant) {
8.    if (is_right_constant) {
9.      return IntPtrConstant(left_constant + right_constant);}
10.     if (left_constant == 0) { return right;}
11.   } else if (is_right_constant) {
12.     if (right_constant == 0) {return left;}
13.     }
14.   return UncheckedCast<WordT>(raw_assembler()->IntPtrAdd(left, right));}
```

上述代码中第 7-13 行用于常量的加法运算，其中第 7-8 行判断加法的左右值，如果都为常量则直接计算结果（第 9 行）；第 10 行代码判断左值，如果为零则直接返回右值；第 14 行代码表示当左、右值均为变量时，则需要添加计算节点。

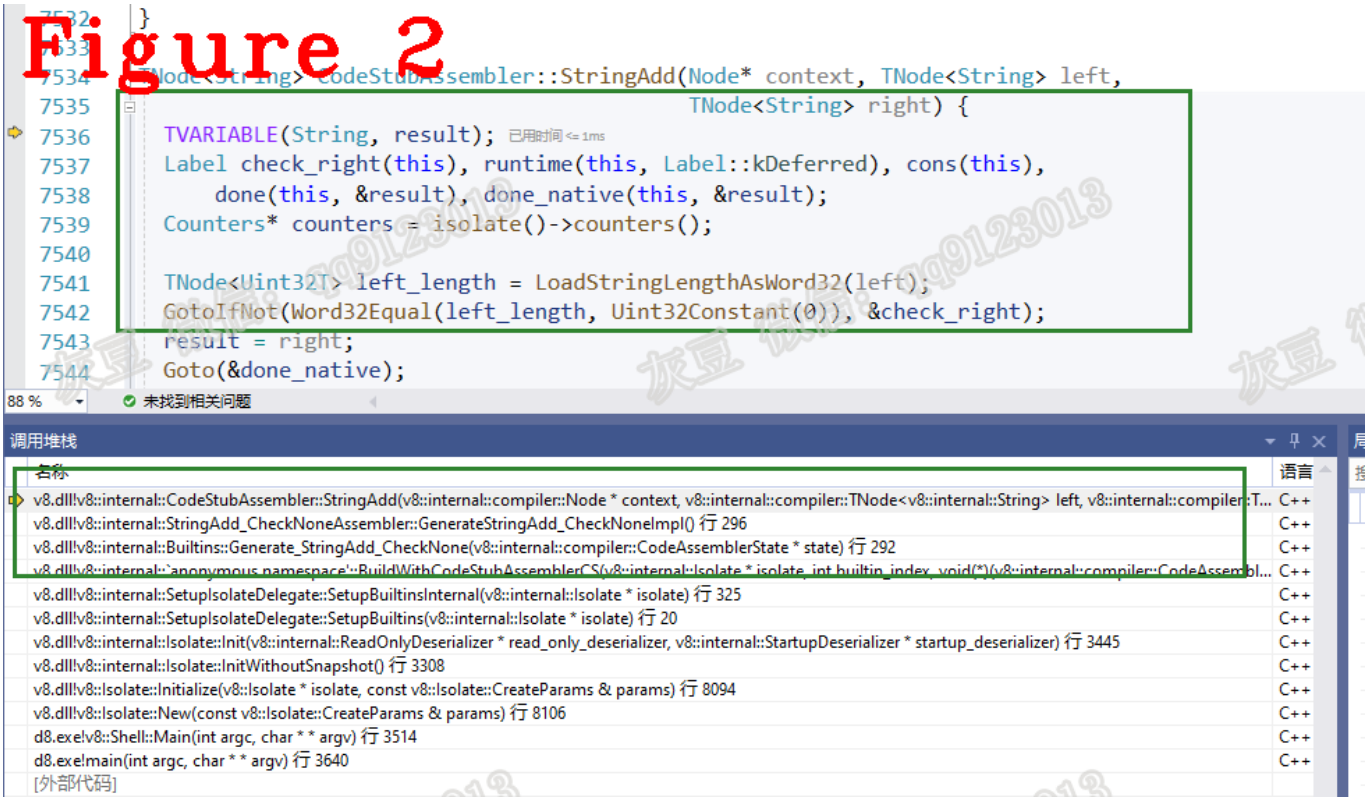
（3） StringAdd_82（第 41 行）中调用 Builtins::kStringAdd_CheckNone 方法，该方法使用 CodeStubAssembler::StringAdd 完成字符串的加法运算，StringAdd 的源码如下：

```
1.  TNode<String> CodeStubAssembler::StringAdd(Node* context, TNode<String> left,
2.                                             TNode<String> right) {
3.  TNode<Uint32T> left_length = LoadStringLengthAsWord32(left);
4.  GotoIfNot(Word32Equal(left_length, Uint32Constant(0)), &check_right);
5.  result = right;
6.  Goto(&done_native);
7.  BIND(&check_right);
8.  TNode<Uint32T> right_length = LoadStringLengthAsWord32(right);
9.  GotoIfNot(Word32Equal(right_length, Uint32Constant(0)), &cons);
10.   result = left;
11.   Goto(&done_native);
12.   BIND(&cons);
13.   {
14.     TNode<Uint32T> new_length = Uint32Add(left_length, right_length);
15.     GotoIf(Uint32GreaterThan(new_length, Uint32Constant(String::kMaxLength)),
16.            &runtime);
17.     TVARIABLE(String, var_left, left);
18.     TVARIABLE(String, var_right, right);
19.     Variable* input_vars[2] = {&var_left, &var_right};
20.     Label non_cons(this, 2, input_vars);
21.     Label slow(this, Label::kDeferred);
22.     GotoIf(Uint32LessThan(new_length, Uint32Constant(ConsString::kMinLength)),
23.            &non_cons);
24.     result =AllocateConsString(new_length, var_left.value(),
var_right.value());
25.     Goto(&done_native);
26.     BIND(&non_cons);
27.     TNode<Int32T> left_instance_type = LoadInstanceType(var_left.value());
28.     TNode<Int32T> right_instance_type = LoadInstanceType(var_right.value());
```

```
29.     TNode<Int32T> ored_instance_types =
30.         Word32Or(left_instance_type, right_instance_type);
31.     TNode<Word32T> xored_instance_types =
32.         Word32Xor(left_instance_type, right_instance_type);
33.     GotoIf(IsSetWord32(xored_instance_types, kStringEncodingMask), &runtime);
34.     GotoIf(IsSetWord32(ored_instance_types, kStringRepresentationMask),
&slow);
35.     TNode<IntPtrT> word_left_length = Signed(ChangeUint32ToWord(left_length));
36.     TNode<IntPtrT> word_right_length =
Signed(ChangeUint32ToWord(right_length));
37.     Label two_byte(this);
38.     GotoIf(Word32Equal(Word32And(ored_instance_types,
39.                                  Int32Constant(kStringEncodingMask)),
40.                        Int32Constant(kTwoByteStringTag)),&two_byte);
41.     result = AllocateSeqOneByteString(new_length);
42.     CopyStringCharacters(/*拷贝左字符串*/);
43.     CopyStringCharacters(/*拷贝右字符串*/);
44.     Goto(&done_native);
45.     BIND(&two_byte);
46.     {
47.       result = AllocateSeqTwoByteString(new_length);
48.       CopyStringCharacters(/*拷贝左字符串*/);
49.       CopyStringCharacters(/*拷贝右字符串*/);
50.       Goto(&done_native); }
51.     BIND(&slow);
52.     {
53.       MaybeDerefIndirectStrings(&var_left, left_instance_type, &var_right,
54.                                 right_instance_type, &non_cons);
55.       Goto(&runtime);}}
56.   BIND(&runtime);
57.   {//省略.....
58.     Goto(&done); }
59.   BIND(&done_native);
60.   {    Goto(&done);  }
61.   BIND(&done);
62.    return result.value();}
```

上述代码中，第 3-6 行代码判断左值长度是否为零，如果为零则直接返回右值；
第 7-11 行代码判断右值长度是否为零，如果为零则直接返回左值；
第 14-15 行代码判断新字符串的长度是否大于 V8 规定的字符串的最大长度，如果大于则使用 runtime 方式处理；
第 22 行代码判断新字符串的长度是否小于 V8 规定的字符串的最小长度，如果小于则使用 slow 或 runtime 方式处理；
第 27-34 行代码判断左、右值的编码是否一致且是否都为 sequential string，如是结果为真则执行第 35-50 行代码；
第 35-50 行代码根据单、双字节的不同，采用不同的方式创建并返回新的字符串，函数执行完毕；
第 51-58 行代码采用 slow 和 runtime 方式处理字符串。

图 2 给出了 StringAdd() 的调用堆栈。



3.String.prototype.concat 测试

测试用例的字节码如下：

```
1.  //省略........
2.     11 S> 000001332DAC2C86 @   16 : 12 01             LdaConstant [1]
3.     11 E> 000001332DAC2C88 @   18 : 15 02 0a          StaGlobal [2], [10]
4.  //省略........
5.     64 S> 000001332DAC2C9A @   36 : 13 02 00          LdaGlobal [2], [0]
6.           000001332DAC2C9D @   39 : 26 f9             Star r2
7.     69 E> 000001332DAC2C9F @   41 : 29 f9 09
LdaNamedPropertyNoFeedback r2, [9]
8.  //省略........
9.     69 E> 000001332DAC2CB3 @   61 : 5f fa f9 04       CallNoFeedback r1, r2-r5
10.  //省略........
11.  Constant pool (size = 13)
12.  000001332DAC2BC9: [FixedArray] in OldSpace
13.   - map: 0x00df20ec0169 <Map>
14.   - length: 13
15.            0: 0x01332dac2b09 <FixedArray[20]>
16.            1: 0x01332dac29d9 <String[#3]: he >
17.            2: 0x01332dac29c1 <String[#4]: txt1>
18.            3: 0x01332dac2a09 <String[#3]: is >
19.            4: 0x01332dac29f1 <String[#4]: txt2>
20.            5: 0x01332dac2a39 <String[#7]: HuiDou >
21.            6: 0x01332dac2a21 <String[#4]: txt3>
22.            7: 0x00df20ec4369 <String[#1]: .>
23.            8: 0x01332dac2a51 <String[#4]: txt4>
```

```
24.              9: 0x037ebea28b49 <String[#6]: concat>
25.             10: 0x01332dac2a69 <String[#3]: bio>
26.             11: 0x037ebea336f1 <String[#7]: console>
27.             12: 0x037ebea32d31 <String[#3]: log>
```

上述代码中，第 2-3 行加载并存储字符串 “he”；第 5-6 行把字符串 “he” 保存到 r2 寄存器中； 第 7 行加载 concat 方法；第 9 行调用 concat 方法，r1 寄存器的值是 concat 的地址，r2-r5 依次是 “he ”、“is ”、“HuiDou ” 和 “.”。debug测试方法：从 CallNoFeedback 开始进行汇编跟踪，如图 3 所示。

Figure 3

```
125         << std::setw(4) << offset << " : ";
126     interpreter::BytecodeDecoder::Decode(os, bytecode_a
127                                          bytecode_array
128     os << std::endl;
129     // Print all input registers and accumulator.
130     PrintRegisters(isolate, os, true, bytecode_iterator
131
132     os << std::flush;
133   }
134   return ReadOnlyRoots(isolate).undefined_value();
135 }
136
137 RUNTIME_FUNCTION(Runtime_InterpreterTraceBytecodeExit)
138
139
140   if (!FLAG_trace_ignition) {
141     return ReadOnlyRoots(isolate).undefined_value();
142   }
143
144
```

```
     [ accumulator -> 0x01332dac29d9 <String[#3]: he > ]
     [          r2 <- 0x01332dac29d9 <String[#3]: he > ]
-> 000001332DAC2C9F @   41 : 29 f9 09          LdaNamedPropertyNoFeedback r2, [9]
     [          r2 -> 0x01332dac29d9 <String[#3]: he > ]
     [ accumulator <- 0x037ebea2af71 <JSFunction concat (sfi = 0000037EBEA2AF31)> ]
-> 000001332DAC2CA2 @   44 : 26 fa             Star r1
     [ accumulator -> 0x037ebea2af71 <JSFunction concat (sfi = 0000037EBEA2AF31)> ]
     [          r1 <- 0x037ebea2af71 <JSFunction concat (sfi = 0000037EBEA2AF31)> ]
-> 000001332DAC2CA4 @   46 : 13 04 02          LdaGlobal [4], [2]
     [ accumulator <- 0x01332dac2a09 <String[#3]: is > ]
-> 000001332DAC2CA7 @   49 : 26 f8             Star r3
     [ accumulator -> 0x01332dac2a09 <String[#3]: is > ]
     [          r3 <- 0x01332dac2a09 <String[#3]: is > ]
-> 000001332DAC2CA9 @   51 : 13 06 04          LdaGlobal [6], [4]
     [ accumulator <- 0x01332dac2a39 <String[#7]: HuiDou > ]
-> 000001332DAC2CAC @   54 : 26 f7             Star r4
     [ accumulator -> 0x01332dac2a39 <String[#7]: HuiDou > ]
     [          r4 <- 0x01332dac2a39 <String[#7]: HuiDou > ]
-> 000001332DAC2CAE @   56 : 13 08 06          LdaGlobal [8], [6]
     [ accumulator <- 0x00df20ec4369 <String[#1]: .> ]
-> 000001332DAC2CB1 @   59 : 26 f6             Star r5
     [ accumulator -> 0x00df20ec4369 <String[#1]: .> ]
     [          r5 <- 0x00df20ec4369 <String[#1]: .> ]
-> 000001332DAC2CB3 @   61 : 5f fa f9 04       CallNoFeedback r1, r2-r5
     [          r1 -> 0x037ebea2af71 <JSFunction concat (sfi = 0000037EBEA2AF31)> ]
     [          r2 -> 0x01332dac29d9 <String[#3]: he > ]
     [          r3 -> 0x01332dac2a09 <String[#3]: is > ]
     [          r4 -> 0x01332dac2a39 <String[#7]: HuiDou > ]
     [          r5 -> 0x00df20ec4369 <String[#1]: .> ]
```

技术总结

（1） 拼接前要判断左、右字符串的类型、编码以及单双字节是否一致；

（2） 拼接采用循环方式对字符串两两拼接；

（3） 字符串的最大长度是 String::kMaxLength（1073741799）。

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 邮箱：v8blink@outlook.com

本文由灰豆原创发布

转载出处： https://www.anquanke.com/post/id/263381

安全客 - 有思想的安全新媒体