《Chrome V8 源码》35. String.prototype.CharAt 源码分析

1 介绍

本文先讲解 String.prototype.CharAt 源码以及相关数据结构，再通过测试用例演示 String.prototype.CharAt 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。
本文的测试用例如下：

```
1.  var s = "hello V8";
2.  var c = s.charAt(2);
3.  console.log(c);
```

2 CharAt 源码

String.prototype.CharAt 由 TF_Bultin 实现，编号是 885。String.prototype.CharAt 源码中使用了TF_BUILTIN宏模板，宏模板展开后的源码如下：

```
1.    class StringPrototypeCharAtAssembler : public CodeStubAssembler {
2.     public:
```

```
3.      using Descriptor = Builtin_StringPrototypeCharAt_InterfaceDescriptor;
4.      explicit StringPrototypeCharAtAssembler(compiler::CodeAssemblerState*
state)
5.          : CodeStubAssembler(state) {}
6.      void GenerateStringPrototypeCharAtImpl();
7.      Node* Parameter(Descriptor::ParameterIndices index) {/*省略*/      }
8.    };
9.    void Builtins::Generate_StringPrototypeCharAt(compiler::CodeAssemblerState*
state) {
10.      StringPrototypeCharAtAssembler assembler(state);
11.        //省略.....
12.      assembler.GenerateStringPrototypeCharAtImpl();
13.    }
14.    void StringPrototypeCharAtAssembler::GenerateStringPrototypeCharAtImpl(){
15.  //省略.....
16.    if (block0.is_used()) {
17.      compiler::TNode<Context> tmp0;
18.      compiler::TNode<Object> tmp1;
19.      compiler::TNode<Object> tmp2;
20.      ca_.Bind(&block0, &tmp0, &tmp1, &tmp2);
21.      ca_.SetSourcePosition("../../../src/builtins/string.tq", 77);
22.      compiler::TypedCodeAssemblerVariable<String> result_0_0(&ca_);
23.      compiler::TypedCodeAssemblerVariable<IntPtrT> result_0_1(&ca_);
24.      compiler::TypedCodeAssemblerVariable<IntPtrT> result_0_2(&ca_);
25.      compiler::CodeAssemblerLabel label0(&ca_);
26.      compiler::CodeAssemblerLabel label1(&ca_);
27.      GenerateStringAt_336(state_, compiler::TNode<Context>{tmp0},
compiler::TNode<Object>{tmp1}, compiler::TNode<Object>{tmp2},
"String.prototype.charAt", &label0, &result_0_0, &result_0_1, &result_0_2,
&label1);
28.      if (label0.is_used()) {
29.        ca_.Bind(&label0);
30.        ca_.Goto(&block5, tmp0, tmp1, tmp2, tmp1, tmp2, result_0_0.value(),
result_0_1.value(), result_0_2.value());
31.      }  }
32.    if (block5.is_used()) {
33.  //省略.....
34.      ca_.Bind(&block5, &tmp3, &tmp4, &tmp5, &tmp6, &tmp7, &tmp8, &tmp9,
&tmp10);
35.      ca_.Goto(&block4, tmp3, tmp4, tmp5, tmp8, tmp9, tmp10);
36.    }
37.    if (block4.is_used()) {
38.  //省略.....
39.      ca_.Bind(&block4, &tmp16, &tmp17, &tmp18, &tmp19, &tmp20, &tmp21);
40.      ca_.SetSourcePosition("../../../src/builtins/string.tq", 81);
41.      compiler::TNode<Int32T> tmp22;
42.      USE(tmp22);
43.      tmp22 =
CodeStubAssembler(state_).StringCharCodeAt(compiler::TNode<String>{tmp19},
compiler::TNode<IntPtrT>{tmp20});
44.      ca_.SetSourcePosition("../../../src/builtins/string.tq", 82);
45.      compiler::TNode<String> tmp23;
46.      USE(tmp23);
47.      tmp23 =
```

```
CodeStubAssembler(state_).StringFromSingleCharCode(compiler::TNode<Int32T>
{tmp22});
48.       CodeStubAssembler(state_).Return(tmp23);
49.     }}
```

编译 String.prototype.CharAt 时，先调用 Builtins::Generate_StringPrototypeCharAt() 方法（第 9 行）生成中间结果，然后 compiler::CodeAssembler::GenerateCode() 方法再将中间结果编译成二进制代码并存储在 butiltin_数组中。 StringPrototypeCharAtAssembler::GenerateStringPrototypeCharAtImpl() （第 14 行）的关键功能如下：

（1） 第 18 行代码 tmp1 代表测试用例中的字符串 "hello V8"；

（2） 第 19 行代码 tmp2 代表测试用例中的 position， position 的值为 2；

（3） 第 27 行代码 GenerateStringAt_336() 判断 tmp1 的类型是否为 string 且 tmp2 是否小于 tmp1 的长度，判断结果为真则执行 32 行代码；

（4） 第 34-35 行代码绑定参数，跳转到第 37 行；

（5） 第 40-43 行代码调用 StringCharCodeAt() 获取指定位置的字符（tmp2）并存储在 tmp22 中。使用 StringFromSingleCharCode() 方法把 tmp22 转换为最终结果，并在第 48 行返回此结果。至此，CharAt 执行完毕。

下面说明 GenerateStringPrototypeCharAtImpl 用到的几个重要方法。

（1） GenerateStringAt_336() 判断字符串的类型和参数的长度是否正确，源码如下：

```
1.  void GenerateStringAt_336(/*省略*/) {
2.  ca_.Goto(&block0, p_context, p_receiver, p_position);
3.  //省略........
4.   if (block0.is_used()) {
5.     ca_.Bind(&block0, &tmp0, &tmp1, &tmp2);
6.      compiler::TNode<String> tmp4;
7.      USE(tmp4);
8.      tmp4 = CodeStubAssembler(state_).ToThisString(compiler::TNode<Context>
{tmp0}, compiler::TNode<Object>{tmp1}, compiler::TNode<String>{tmp3});
9.      ca_.SetSourcePosition("../../../src/builtins/string.tq", 65);
10.      compiler::TNode<Number> tmp5;
11.      USE(tmp5);
12.      tmp5 =
CodeStubAssembler(state_).ToInteger_Inline(compiler::TNode<Context>{tmp0},
compiler::TNode<Object>{tmp2},
CodeStubAssembler::ToIntegerTruncationMode::kTruncateMinusZero);
13.      ca_.SetSourcePosition("../../../src/builtins/string.tq", 64);
14.      ca_.SetSourcePosition("../../../src/builtins/string.tq", 66);
15.      compiler::TNode<BoolT> tmp6;
16.      USE(tmp6);
17.      tmp6 = CodeStubAssembler(state_).TaggedIsNotSmi(compiler::TNode<Object>
{tmp5});
18.      ca_.Branch(tmp6, &block3, &block4, tmp0, tmp1, tmp2, tmp4, tmp5);
19.    }//省略........
20.    if (block4.is_used()) {
21.  //省略..........
22.      ca_.Branch(tmp22, &block5, &block6, tmp12, tmp13, tmp14, tmp15, tmp16,
tmp18, tmp19);
23.    }
```

```
24.     if (block6.is_used()) {ca_.Goto(&block1, tmp33, tmp35, tmp36);}
25.     if (block1.is_used()) {ca_.Goto(label_IfInBounds);}
26.     if (block2.is_used()) {
27.       ca_.Bind(&block2);
28.       ca_.Goto(label_IfOutOfBounds);} }
```

上述代码中，第 8 行 把 this（测试用例中的字符串）的类型转换为字符串；
第 9-12 行代码判断 position 的长度（测试用例中的 2）是否需要截断；
第 17 行代码判断 position 的长度是否符合规定，判断结果存储在 tmp6 中；
第 18 行代码会根据 tmp6 的结果进行跳转，若结果为真则跳转到 block4；
第 21 行代码判断 position 是否小于字符串的长度，小于代表 CharAt() 操作没有越界（InBound），否则代表 OutBound，OutBound 意味着最终的结果为空。

(2) StringCharCodeAt() 定义在类 CodeStubAssembler 中，源码如下：

```
1.  TNode<Int32T> CodeStubAssembler::StringCharCodeAt(SloppyTNode<String> string,
2.                                                    SloppyTNode<IntPtrT> index)
{
3.    CSA_ASSERT(this, IsString(string));
4.    CSA_ASSERT(this, IntPtrGreaterThanOrEqual(index, IntPtrConstant(0)));
5.    CSA_ASSERT(this, IntPtrLessThan(index, LoadStringLengthAsWord(string)));
6.    TVARIABLE(Int32T, var_result);
7.    Label return_result(this), if_runtime(this, Label::kDeferred),
8.        if_stringistwobyte(this), if_stringisonebyte(this);
9.    ToDirectStringAssembler to_direct(state(), string);
10.    to_direct.TryToDirect(&if_runtime);
11.    TNode<IntPtrT> const offset = IntPtrAdd(index, to_direct.offset());
12.    TNode<Int32T> const instance_type = to_direct.instance_type();
13.    TNode<RawPtrT> const string_data = to_direct.PointerToData(&if_runtime);
14.    // Check if the {string} is a TwoByteSeqString or a OneByteSeqString.
15.    Branch(IsOneByteStringInstanceType(instance_type), &if_stringisonebyte,
16.           &if_stringistwobyte);
17.    BIND(&if_stringisonebyte);
18.    {
19.      var_result =
20.          UncheckedCast<Int32T>(Load(MachineType::Uint8(), string_data,
offset));
21.      Goto(&return_result);
22.    }
23.    BIND(&if_stringistwobyte);
24.    {
25.      var_result =
26.          UncheckedCast<Int32T>(Load(MachineType::Uint16(), string_data,
27.                                     WordShl(offset, IntPtrConstant(1))));
28.      Goto(&return_result);
29.    }
30.    BIND(&if_runtime);
31.    {
32.      TNode<Object> result = CallRuntime(
33.          Runtime::kStringCharCodeAt, NoContextConstant(), string,
SmiTag(index));
34.      var_result = SmiToInt32(CAST(result));
```

```
35.       Goto(&return_result);
36.     }
37.     BIND(&return_result);
38.     return var_result.value();
39.   }
```

上述代码中，第 2-5 行判断字符串类型是否正确、index 是否大于等于零且小于字符串长度；
第 6 行代码申请 Int32 类型的变量 var_result，用于存储返回值；
第 7 行代码申请四个标签变量，将来该变量会配合 Branch 完成函数跳转；
第 10 行代码 to_direct.TryToDirect(&if_runtime) 的作用是将 flat、thin 或 slice 类型的 indirect string 转换成 direct string。若转换失败则跳转到第 30 行代码，使用 runtime 处理字符串；若转换成功则返回到 string_data;
第 15 行代码判断字符串类型是单字节还是双字节；
第 19 行代码使用单字节方式从 string_data 的偏移位置读取字节并存储到 var_result 中；
第 25 行代码使用双字节方式从 string_data 的偏移位置读取字节并存储到 var_result中；
第 32 行代码使用 runtime 方式实现 StringChartAt 功能，后续文章另做讲解。
（3） StringFromSingleCharCode() 定义在 CodeStubAssembler 中，作用是把 Int 数值转换为字符串。

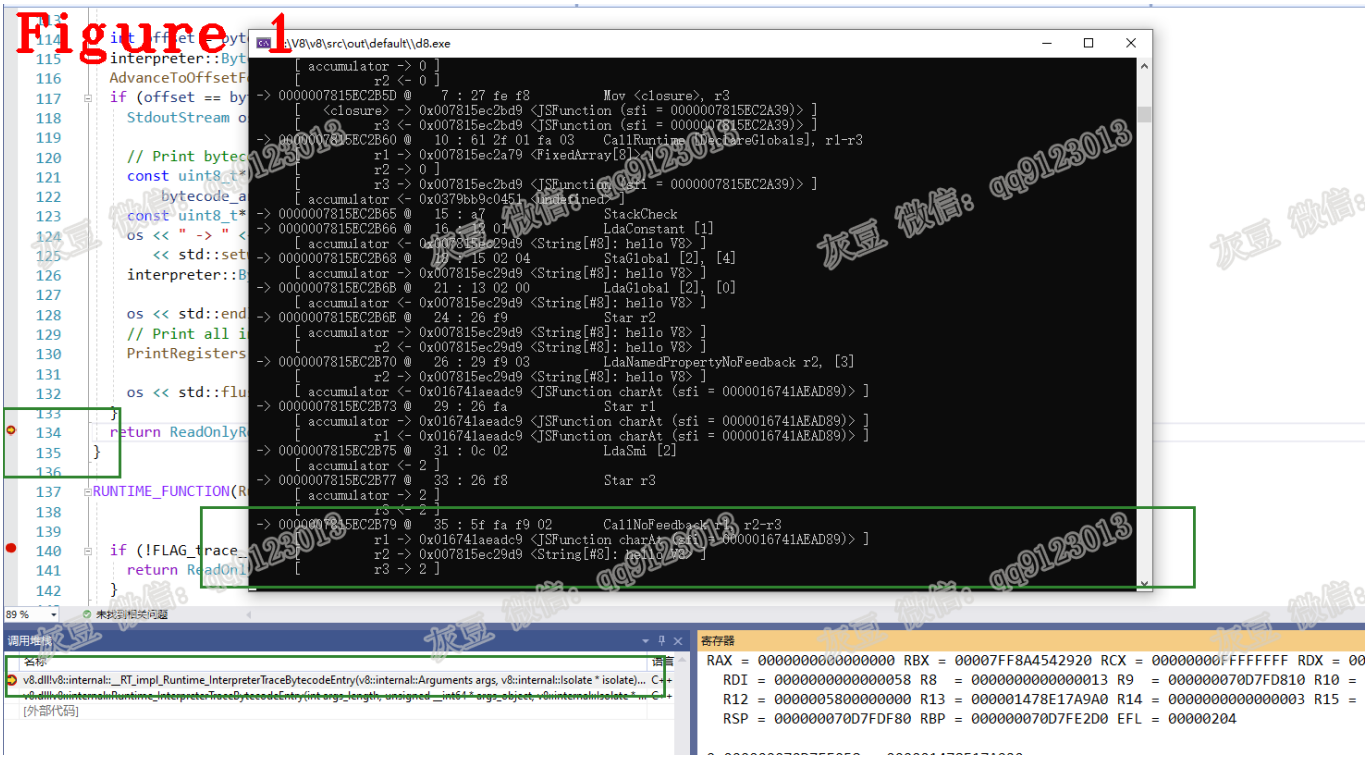
3 CharAt 测试

测试代码的字节码如下：

```
1.  //分隔线.............//省略了部分字节码
2.       8 S> 0000012A281C2B26 @   16 : 12 01             LdaConstant [1]
3.       8 E> 0000012A281C2B28 @   18 : 15 02 04          StaGlobal [2], [4]
4.      29 S> 0000012A281C2B2B @   21 : 13 02 00          LdaGlobal [2], [0]
5.            0000012A281C2B2E @   24 : 26 f9             Star r2
6.      31 E> 0000012A281C2B30 @   26 : 29 f9 03
LdaNamedPropertyNoFeedback r2, [3]
7.            0000012A281C2B33 @   29 : 26 fa             Star r1
8.            0000012A281C2B35 @   31 : 0c 02             LdaSmi [2]
9.            0000012A281C2B37 @   33 : 26 f8             Star r3
10.     31 E> 0000012A281C2B39 @   35 : 5f fa f9 02       CallNoFeedback r1, r2-r3
11.     29 E> 0000012A281C2B3D @   39 : 15 04 06          StaGlobal [4], [6]
12.     43 S> 0000012A281C2B40 @   42 : 13 05 08          LdaGlobal [5], [8]
13.           0000012A281C2B43 @   45 : 26 f9             Star r2
14.     51 E> 0000012A281C2B45 @   47 : 29 f9 06
LdaNamedPropertyNoFeedback r2, [6]
15.           0000012A281C2B48 @   50 : 26 fa             Star r1
16.     55 E> 0000012A281C2B4A @   52 : 13 04 02          LdaGlobal [4], [2]
17.           0000012A281C2B4D @   55 : 26 f8             Star r3
18.     51 E> 0000012A281C2B4F @   57 : 5f fa f9 02       CallNoFeedback r1, r2-r3
19.           0000012A281C2B53 @   61 : 26 fb             Star r0
20.     58 S> 0000012A281C2B55 @   63 : ab                Return
21.  Constant pool (size = 7)
22.  0000012A281C2A99: [FixedArray] in OldSpace
23.   - map: 0x0148906c0169 <Map>
24.   - length: 7
25.           0: 0x012a281c2a39 <FixedArray[8]>
26.           1: 0x012a281c2999 <String[#8]: hello V8>
```

```
27.              2: 0x012a281c2981 <String[#1]: s>
28.              3: 0x00024f52ad19 <String[#6]: charAt>
29.              4: 0x00024f53e069 <String[#1]: c>
30.              5: 0x00024f533699 <String[#7]: console>
31.              6: 0x00024f532cd9 <String[#3]: log>
```

上述代码中，第 2 行代码把常量 hello v8 加载到累加寄存器；第 3-5 行代码存储并读取常量 hello v8 ，该常量最终被存储到 r2 寄存器中；第 6 行代码加载 String.prototype.CharAt()，第 10 行代码调用 String.prototype.CharAt；第 25-31 行是常量池，用于存储运行期间需要使用的常量。



调试字节码 CallNoFeedback 的执行可以看到 String.prototype.CharAt 的执行过程。调试方法为：设置 FLAG_trace_ignition = true；在 Runtime_InterpreterTraceBytecodeEntry 中设置断点；在图 1 中的 “CallNoFeedback r1, r2-r3” 位置使用汇编进行调试。

技术总结

（1） 编译 String.prototype.CharAt 生成的二进制代码（code）存放在 builtin_ 数组中，解释执行 JavaScript 源码时会使用 code，而不是使用本文所讲的 StringCharAt 的源码;

（2） V8 中字符串细分为单字节、双字节、flat 等类型；

（3） indirect 类型的字符串只能使用 runtime 处理，先用 to_direct.TryToDirect() 判断字符串类型可以省去 runtime 不必要的操作；

（4） String.prototype.CharAt 的工作流程是：判断字符串类型和位置是否正确、判断位置是否越界、直接字符串采用 Builtin 方式处理、间接字符串采用 runtime 方式处理。

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：https://www.zhihu.com/people/v8blink

本文由灰豆原创发布

转载出处： https://www.anquanke.com/post/id/263308

安全客 - 有思想的安全新媒体