《Chrome V8 源码》41. Runtime_StringTrim 源码、触发条件

1 介绍

Runtime 是一系列采用 C++ 语言编写的功能方法，它实现了大量 JavaScript 运行期间需要的 native 功能。接下来几篇文章将介绍一些 Runtime 方法。本文分析 Runtime_StringTrim 方法的源码和重要数据结构，讲解 Runtime_StringTrim 方法的触发条件。
注意： Runtime 方法的加载、调用以及 RUNTIME_FUNCTION 宏模板请参见第十六篇文章。--allow-natives-syntax 和 %-prefix 不是本文的讲解重点。

2 StringTrim 测试用例

编写可以触发特定的 V8 内部功能的 JavaScript 测试用例，可以帮助我们更好地理解 V8 的内部工作原理，达到事半功倍的效果。下面讲解 Runtime_StringTrim 测试用例的编写思路：
字符串的 Trim 方法由 TF_BUILTIN(StringPrototypeTrim, StringTrimAssembler) 函数实现，这个函数设置了一些字符串检测条件，如果满足检测条件就会启动 Runtime_StringTrim 方法。因此，我们需要从 TF_BUILTIN(StringPrototypeTrim, StringTrimAssembler) 开始分析，源码如下：

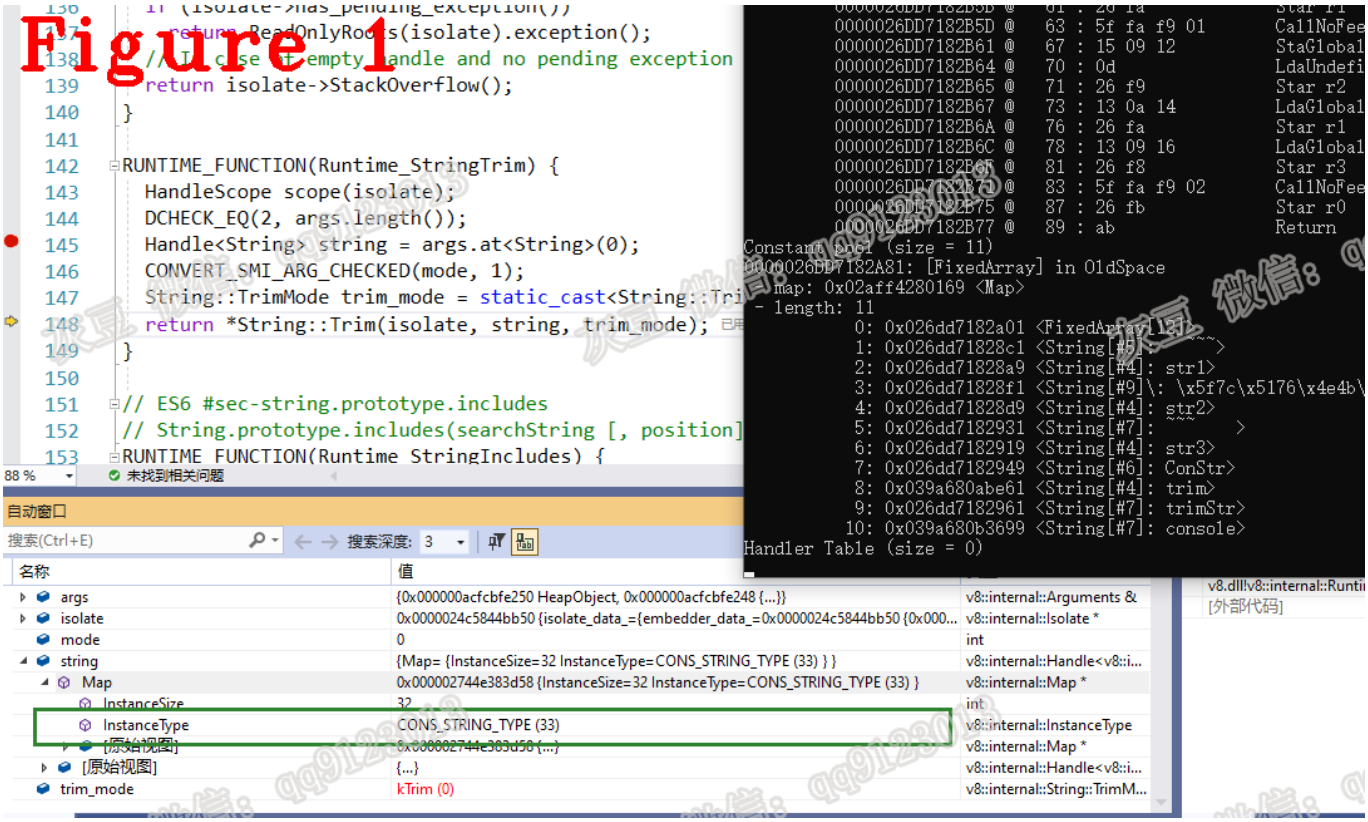
```
1.  TF_BUILTIN(StringPrototypeTrim, StringTrimAssembler) {
2.    TNode<IntPtrT> argc =
3.        ChangeInt32ToIntPtr(Parameter(Descriptor::kJSActualArgumentsCount));
4.    TNode<Context> context = CAST(Parameter(Descriptor::kContext));
5.    Generate(String::kTrim, "String.prototype.trim", argc, context);
```

```
6.  }
7.  //分隔.................
8.  void StringTrimAssembler::Generate(String::TrimMode mode,
9.                                     const char* method_name, TNode<IntPtrT>
argc,
10.                                     TNode<Context> context) {
11.    Label return_emptystring(this), if_runtime(this);
12.    CodeStubArguments arguments(this, argc);
13.    TNode<Object> receiver = arguments.GetReceiver();
14.    TNode<String> const string = ToThisString(context, receiver, method_name);
15.    TNode<IntPtrT> const string_length = LoadStringLengthAsWord(string);
16.    ToDirectStringAssembler to_direct(state(), string);
17.    to_direct.TryToDirect(&if_runtime);
18.    TNode<RawPtrT> const string_data = to_direct.PointerToData(&if_runtime);
19.    TNode<Int32T> const instance_type = to_direct.instance_type();
20.    TNode<BoolT> const is_stringonebyte =
21.        IsOneByteStringInstanceType(instance_type);
22.    TNode<IntPtrT> const string_data_offset = to_direct.offset();
23.    TVARIABLE(IntPtrT, var_start, IntPtrConstant(0));
24.    TVARIABLE(IntPtrT, var_end, IntPtrSub(string_length, IntPtrConstant(1)));
25.  //省略.............
26.    arguments.PopAndReturn(
27.        SubString(string, var_start.value(),
28.                  IntPtrAdd(var_end.value(), IntPtrConstant(1))));
29.    BIND(&if_runtime);
30.    arguments.PopAndReturn(
31.        CallRuntime(Runtime::kStringTrim, context, string, SmiConstant(mode)));
32.    BIND(&return_emptystring);
33.    arguments.PopAndReturn(EmptyStringConstant());
34.  }
```

上述代码中，第 5 行代码调用了 Generate() 方法；
第 11 行代码定义 runtime 标签；
第 14-15 行代码获取字符串以及它的长度；
第 16-17 行 TryToDirect 把字符串转换为直接字符串，如果 TryToDirect 失败将采用 Runtime 方式处理；
第 29 行绑定 runtime 标签；
第 31 行调用 Runtime::kStringTrim 方法。
runtime 标签仅在第 17 行被使用一次，由此我们可知：构造一段 “TryToDirect 失败” 的 JavaScript 源码是触发 Runtime 的条件。TryToDirect() 的原理和失败条件在之前的文章中讲过。V8 的字符串类型包括：SeqString、ConsString、SliceString、ThinString、ExternalString。直接给出结论：一个单字节串和两个双字节串组成的 ConsString 串可以导致 “TryToDirect 失败”，源码如下：

```
var str1 = "  ~~~"; //前面有空格
var str2 = "彼其之子、美如玉。";
var str3 ="~~~    "; //后面有空格
ConStr = str1+str2+str3;
trimStr = ConStr.trim();
console.log(trimStr);
```

图 1 中可以看到 ConStr InstanceType 的值是 CONS_STRING_TYPE，它导致 “TryToDirect 失败” 并启动 Runtime。



3 StringTrim 源码

源码如下：

```
1.  RUNTIME_FUNCTION(Runtime_StringTrim) {
2.    HandleScope scope(isolate);
3.    DCHECK_EQ(2, args.length());
4.    Handle<String> string = args.at<String>(0);
5.    CONVERT_SMI_ARG_CHECKED(mode, 1);
6.    String::TrimMode trim_mode = static_cast<String::TrimMode>(mode);
7.    return *String::Trim(isolate, string, trim_mode);
8.  }
9.  //分隔线.............
10.  Handle<String> String::Trim(Isolate* isolate, Handle<String> string,
11.                              TrimMode mode) {
12.    string = String::Flatten(isolate, string);
13.    int const length = string->length();
14.    // Perform left trimming if requested.
15.    int left = 0;
16.    if (mode == kTrim || mode == kTrimStart) {
17.      while (left < length && IsWhiteSpaceOrLineTerminator(string->Get(left)))
{
18.        left++;
19.      }
20.    }
21.    // Perform right trimming if requested.
```

```
22.    int right = length;
23.    if (mode == kTrim || mode == kTrimEnd) {
24.      while (right > left &&
25.             IsWhiteSpaceOrLineTerminator(string->Get(right - 1))) {
26.        right--;
27.      }
28.    }
29.    return isolate->factory()->NewSubString(string, left, right);
30.  }
```

上述代码中，第 4 行代码获取字符串，也就是测试用例的 ConStr；
第 6 行代码调用 *String::Trim(isolate, string, trim_mode) 以完成 Trim 功能；
第 12 行代码对 ConStr 进行 Flatten 处理，结果保存为连续存储的字符串 string。因为 ConStr 由三个子串组成，所以 Flatten 方法中会使用递归调用来处理 ConStr，详见上篇文章。
第 16-17 行代码从 string 的头部依次判断每个字符是否为空格或行结尾符，记录不是空格或行结尾符的位置 left；
第 24-26 行代码从 string 的尾部依次判断每个字符是否为空格或行结尾符，记录不是空格或行结尾符的位置 right；
第 29 行代码调用 NewSubString 生成新的字符串。正如 ECMA 所说的那样：Trim 不会改变原字符串，而是生成新的字符串。
NewSubString 中调用 NewProperSubString 以生成最终的结果，NewProperSubString 源码分析参见上一篇文章。
下面给出判断空格和行结尾符的函数源码：

```
bool IsWhiteSpaceOrLineTerminator(uc32 c) {
  if (!IsInRange(c, 0, 127)) return IsWhiteSpaceOrLineTerminatorSlow(c);
  DCHECK_EQ(
      IsWhiteSpaceOrLineTerminatorSlow(c),
      static_cast<bool>(kAsciiCharFlags[c] & kIsWhiteSpaceOrLineTerminator));
  return kAsciiCharFlags[c] & kIsWhiteSpaceOrLineTerminator;
}
```

首先判断字符是否在 0-127 区间，如果不在区间内使用 Slow 方式判断，源码如下：

```
inline bool IsWhiteSpaceOrLineTerminatorSlow(uc32 c) {
  return IsWhiteSpaceSlow(c) || unibrow::IsLineTerminator(c);
}
//.....................分隔线.................
// ES#sec-white-space White Space
// gC=Zs, U+0009, U+000B, U+000C, U+FEFF
bool IsWhiteSpaceSlow(uc32 c) {
  return (u_charType(c) == U_SPACE_SEPARATOR) ||
         (c < 0x0D && (c == 0x09 || c == 0x0B || c == 0x0C)) || c == 0xFEFF;
}
//.....................分隔线.................
// LineTerminator:       'JS_Line_Terminator' in point.properties
// ES#sec-line-terminators lists exactly 4 code points:
```

```
// LF (U+000A), CR (U+000D), LS(U+2028), PS(U+2029)
V8_INLINE bool IsLineTerminator(uchar c) {
  return c == 0x000A || c == 0x000D || c == 0x2028 || c == 0x2029;
}
```

上述代码分为三部分，第二、三部实现 ECMA 规范，第一部分是他们的入口函数。
IsWhiteSpaceOrLineTerminator() 中的 kAsciiCharFlags 数组定义 Ascii 字符，kAsciiCharFlags 数组中又引用了 BuildAsciiCharFlags() 方法，该方法说明了 \t、\v 是空格、还是行结尾符，也就是 BuildAsciiCharFlags() 方法影响 Strint.trim() 的结果。相关源码如下：

```
const constexpr uint8_t kAsciiCharFlags[128] = {
#define BUILD_CHAR_FLAGS(N) BuildAsciiCharFlags(N),
    INT_0_TO_127_LIST(BUILD_CHAR_FLAGS)
#undef BUILD_CHAR_FLAGS
};
//.................分隔线.........................
constexpr uint8_t BuildAsciiCharFlags(uc32 c) {
  return ((IsAsciiIdentifier(c) || c == '\\')
              ? (kIsIdentifierPart |
                 (!IsDecimalDigit(c) ? kIsIdentifierStart : 0))
              : 0) |
         ((c == ' ' || c == '\t' || c == '\v' || c == '\f')
              ? kIsWhiteSpace | kIsWhiteSpaceOrLineTerminator
              : 0) |
         ((c == '\r' || c == '\n') ? kIsWhiteSpaceOrLineTerminator : 0);
}
//................分隔线.......................
#define INT_0_TO_127_LIST(V)                                          \
V(0)   V(1)   V(2)   V(3)   V(4)   V(5)   V(6)   V(7)   V(8)   V(9)   \
V(10)  V(11)  V(12)  V(13)  V(14)  V(15)  V(16)  V(17)  V(18)  V(19)  \
V(20)  V(21)  V(22)  V(23)  V(24)  V(25)  V(26)  V(27)  V(28)  V(29)  \
V(30)  V(31)  V(32)  V(33)  V(34)  V(35)  V(36)  V(37)  V(38)  V(39)  \
V(40)  V(41)  V(42)  V(43)  V(44)  V(45)  V(46)  V(47)  V(48)  V(49)  \
V(50)  V(51)  V(52)  V(53)  V(54)  V(55)  V(56)  V(57)  V(58)  V(59)  \
V(60)  V(61)  V(62)  V(63)  V(64)  V(65)  V(66)  V(67)  V(68)  V(69)  \
V(70)  V(71)  V(72)  V(73)  V(74)  V(75)  V(76)  V(77)  V(78)  V(79)  \
V(80)  V(81)  V(82)  V(83)  V(84)  V(85)  V(86)  V(87)  V(88)  V(89)  \
V(90)  V(91)  V(92)  V(93)  V(94)  V(95)  V(96)  V(97)  V(98)  V(99)  \
V(100) V(101) V(102) V(103) V(104) V(105) V(106) V(107) V(108) V(109) \
V(110) V(111) V(112) V(113) V(114) V(115) V(116) V(117) V(118) V(119) \
V(120) V(121) V(122) V(123) V(124) V(125) V(126) V(127)
```

上述代码分为三部分，他们共同完成 kAsciiCharFlags 数组的定义。
下面给出从字符串中读取字符的函数源码，也就是 IsWhiteSpaceOrLineTerminator(string->Get(left)) 中的 “Get” 方法，源码如下：

```
uint16_t String::Get(int index) {
  DCHECK(index >= 0 && index < length());
```

```
  class StringGetDispatcher : public AllStatic {
   public:
#define DEFINE_METHOD(Type)                                    \
  static inline uint16_t Handle##Type(Type str, int index) { \
    return str.Get(index);                                     \
  }
    STRING_CLASS_TYPES(DEFINE_METHOD)
#undef DEFINE_METHOD
    static inline uint16_t HandleInvalidString(String str, int index) {
      UNREACHABLE();
    }
  };

  return StringShape(*this)
      .DispatchToSpecificType<StringGetDispatcher, uint16_t>(*this, index);
}
```

Get 方法用于读取 index 位置的字符。从 String 中读取字符时，要根据 String Header 的长度计算字符串的首位置，然后再加上 index 读取相应的字符。

技术总结

（1） Runtime Trim 的效率比 TF_BUILTIN(StringPrototypeTrim) 低很多；

（2） 字符串的类型影响 TryToDirect 的成败。

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