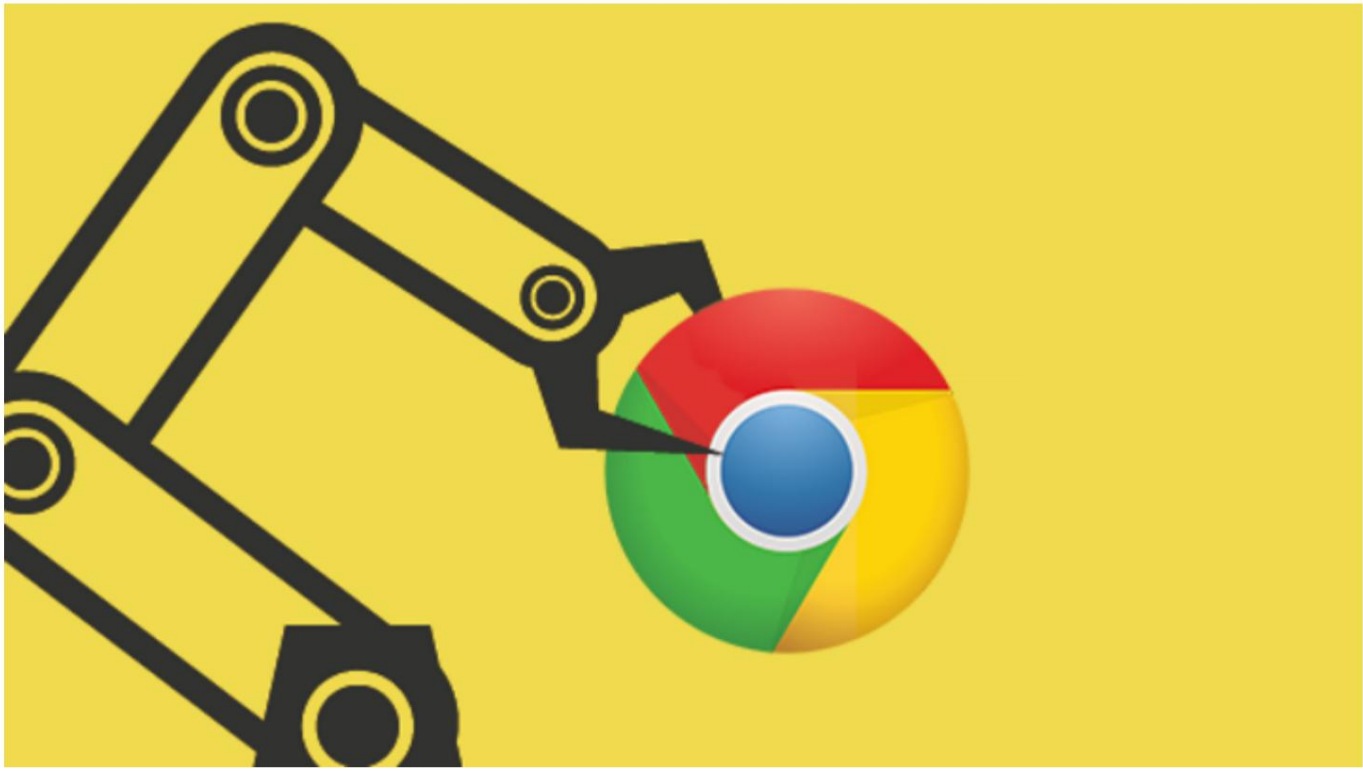
"Chrome V8 Source Code" 41. Runtime_StringTrim source code, touch

issuance conditions



1 Introduction

Runtime is a series of functional methods written in C++ language, which implements a large number of native functions required by JavaScript during runtime. catch
The next few articles will introduce some Runtime methods. This article analyzes the source code and important data structure of the Runtime_StringTrim method and explains
Trigger conditions for the Runtime_StringTrim method.

Note: Please refer to the sixteenth article for the loading and calling of Runtime methods and the RUNTIME_FUNCTION macro template. --allow-natives-syntax and %-prefix are not the focus of this article.

2 StringTrim test case

Writing JavaScript test cases that can trigger specific V8 internal functions can help us better understand the internal working principles of V8 and achieve
Get twice the result with half the effort. The following explains the ideas for writing Runtime_StringTrim test cases:

The Trim method of string is implemented by TF_BUILTIN(StringPrototypeTrim, StringTrimAssembler) function. This function sets some
String detection conditions. If the detection conditions are met, the Runtime_StringTrim method will be started. Therefore, we need to start from
TF_BUILTIN(StringPrototypeTrim, StringTrimAssembler) starts analysis, the source code is as follows:

```
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. //Separation.....................
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 =
twenty one.         IsOneByteStringInstanceType(instance_type);
twenty two.     TNode<IntPtrT> const string_data_offset = to_direct.offset();
twenty three.   TVARIABLE(IntPtrT, var_start, IntPtrConstant(0));
24.     TVARIABLE(IntPtrT, var_end, IntPtrSub(string_length, IntPtrConstant(1)));
25. //Omit.....................
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. }
```

In the above code, the 5th line of code calls the Generate() method;

Line 11 of code defines the runtime tag;

Lines 14-15 of code get the string and its length;

Lines 16-17 TryToDirect converts the string into a direct string. If TryToDirect fails, the Runtime method will be used;

Line 29 binds the runtime tag;

Line 31 calls the Runtime::kStringTrim method.

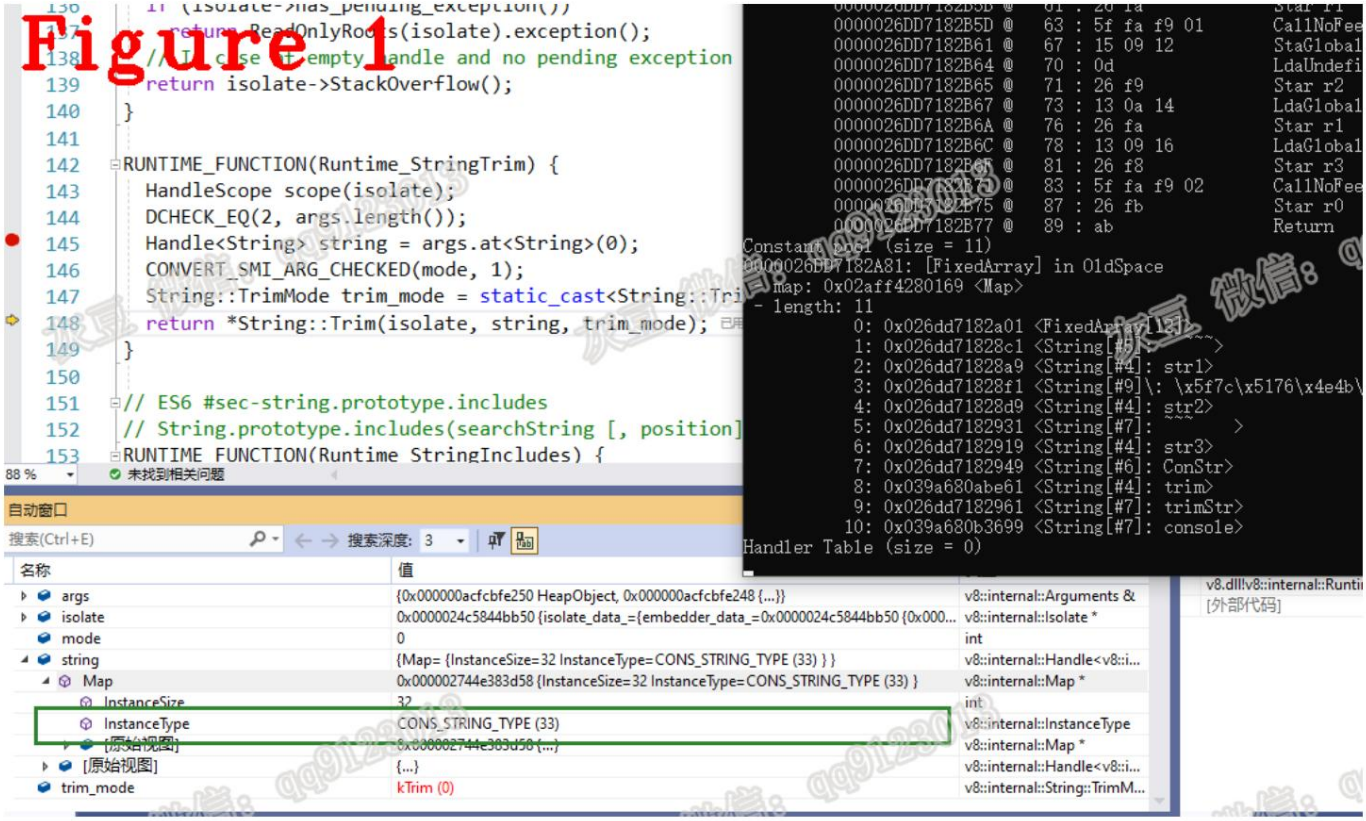
The runtime tag is only used once in line 17. From this we can know that constructing a piece of JavaScript source code for "TryToDirect failed" is the trigger Runtime conditions. The principle and failure conditions of TryToDirect() have been discussed in previous articles. V8's string types include: SeqString, ConsString, SliceString, ThinString, ExternalString. Give the conclusion directly: it consists of a single-byte string and two double-byte strings.

ConsString string can cause "TryToDirect failure", the source code is as follows:

```
var str1 =        "~~~"; //There is a space in front of it
var str2 = "His son is as beautiful as jade.";
var str3 ="~~~              "; //There is a space after
ConStr = str1+str2+str3;
trimStr = ConStr.trim();
console.log(trimStr);
```

In Figure 1 you can see that the value of ConStr InstanceType is CONS_STRING_TYPE, which causes "TryToDirect failed" and starts Runtime.

Figure 1



3 StringTrim source code

The source code is as follows:

```
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. //Separator line.....
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.       }
twenty one.       // Perform right trimming if requested.
```

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

In the above code, the 4th line of code obtains the string, which is the ConStr of the test case;

the 6th line of code calls *String::Trim(isolate, string, trim_mode) to complete the Trim function; the 12th line

of code flattens the ConStr , the result is saved as a continuously stored string string. Because ConStr consists of three substrings, recursive calls are used

in the Flatten method to process ConStr. See the previous article for details. Lines 16-17 of the code

determine whether each character is a space or line terminator from the head of the string, and record the position left of the character that is not a space or line

terminator; Lines 24-26 of the code determine each character in sequence from the end of the string. Whether it is a space or a line terminator, record the position

right that is not a space or a line terminator; Line 29 of the code calls NewSubString to generate a new string. As ECMA says: Trim does not change the original string, but generates a new string.

NewProperSubString is called in NewSubString to generate the final result. For NewProperSubString source code analysis, please refer to the previous article. The

following is the function source code for judging spaces and line endings:

```
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;
}
```

First, determine whether the character is in the range 0-127. If it is not within the range, use the Slow method. The source code is as follows:

```
inline bool IsWhiteSpaceOrLineTerminatorSlow(uc32 c) {
    return IsWhiteSpaceSlow(c) || unibrow::IsLineTerminator(c);

} //.....separator line.....
// 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;

} //.....separator line.....
// 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;
}
```

The above code is divided into three parts. The second and third parts implement the ECMA specification, and the first part is their entry function.

The kAsciiCharFlags array in IsWhiteSpaceOrLineTerminator() defines Ascii characters. The kAsciiCharFlags array also refers to the BuildAsciiCharFlags() method. This method specifies whether \t and \v are spaces or line terminators. That is, the BuildAsciiCharFlags() method affects Strint.trim( ) the result of. The relevant source code is as follows:

```
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 }; //.....separator lines.....
constexpr uint8_t BuildAsciiCharFlags(uc32 c) { return
    ((IsAsciiIdentifier(c) || c == '\\')
                    ? (kIsIdentifierPart |
                        (!IsDecimalDigit(c) ? kIsIdentifierStart : 0)) : 0) | ' || c == '\t' ||
                    c ==
                        '\v' || c == '\f') ((c == ? kIsWhiteSpace |
                kIsWhiteSpaceOrLineTerminator : 0) |

                ((c == '\r' || c == '\n') ? kIsWhiteSpaceOrLineTerminator : 0);

} //.....Separator line..... #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)
```

The above code is divided into three parts, and together they complete the definition of the kAsciiCharFlags array. The following is the function source code for reading characters from a string, which is the "Get" method in IsWhiteSpaceOrLineTerminator(string->Get(left)). The source code is as follows:

```
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);
}
```

The Get method is used to read the character at index position. When reading characters from String, calculate the first position of the string based on the length of the String Header, and then add the index to read the corresponding characters.

Technical summary

(1) The efficiency of Runtime Trim is much lower than TF_BUILTIN(StringPrototypeTrim); **(2)** The type of string affects the success or failure of TryToDirect. Okay, that's it for today, see you next time.

Personal abilities are limited and there are shortcomings and mistakes. Welcome to criticize and correct me. WeChat: qq9123013 Note: v8 communication email:

v8blink@outlook.com This article was originally published by Huidou. Source: https://www.anquanke.com/post/id/264385 Ankangke - Thoughtful new safety media