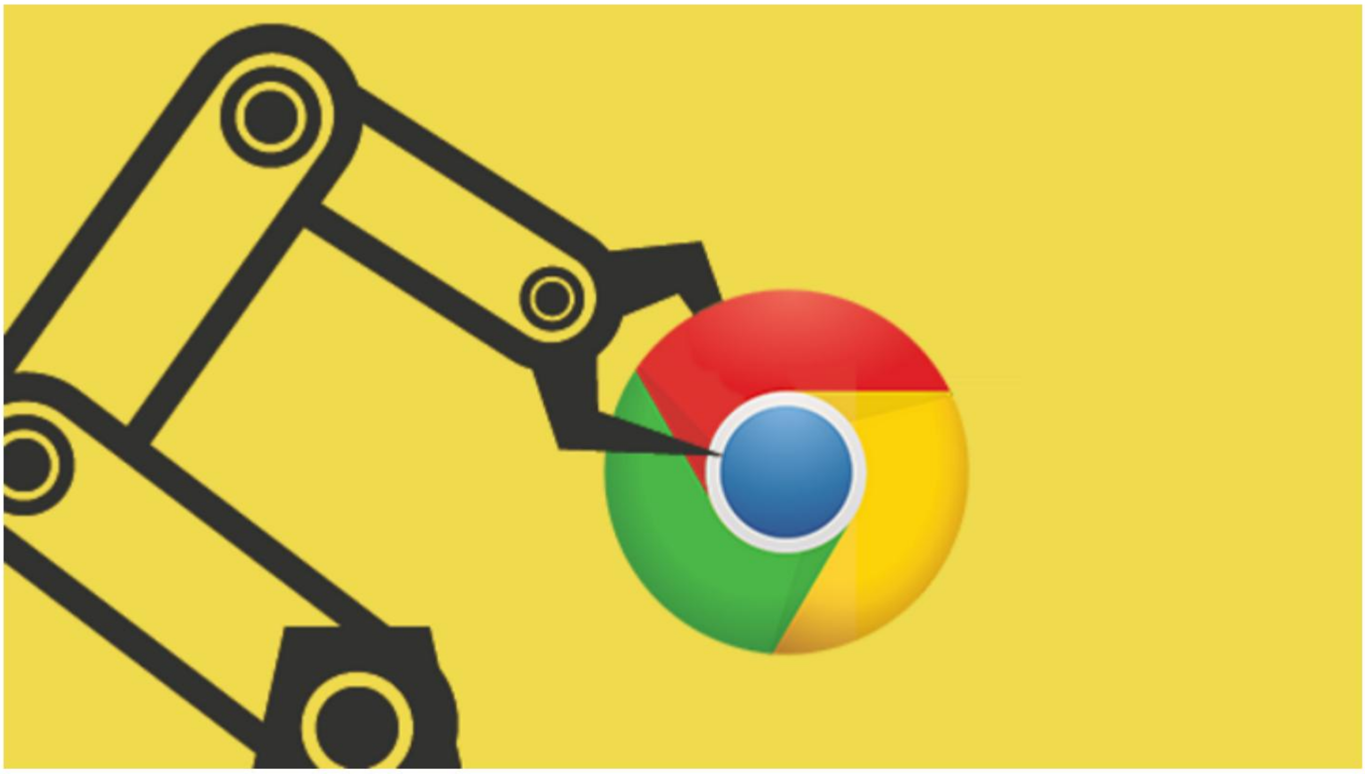
"Chrome V8 Source Code" 40.Runtime substring detailed explanation



1 Introduction

Runtime is a series of functional methods written in C++ language. It implements a large number of native functions required during JavaScript running, such as Such as String add, String split. The next few articles will introduce some Runtime methods. This article analyzes Runtime_StringSubstring The source code and important data structure of the method are explained, and the trigger conditions of the Runtime_StringSubstring method are explained.

Note that for the loading and calling of the Runtime method and the RUNTIME_FUNCTION macro template, please refer to the sixteenth article.

2 Runtime_StringSubstring source code analysis

The source code is as follows:

```
RUNTIME_FUNCTION(Runtime_StringSubstring) {
        HandleScope scope(isolate);
        DCHECK_EQ(3, args.length());
        CONVERT_ARG_HANDLE_CHECKED(String, string, 0);
        CONVERT_INT32_ARG_CHECKED(start, 1);
        CONVERT_INT32_ARG_CHECKED(end, 2);
        DCHECK_LE(0, start);
        DCHECK_LE(start, end);
        DCHECK_LE(end, string->length());
          isolate->counters()->sub_string_runtime()->Increment();
          return *isolate->factory()->NewSubString(string, start, end);
}
```

In the above code, the third line of code detects whether the number of parameters is 3, if not, an error will be reported;

Lines 4-6 of the code obtain three parameters: string, start, and end respectively;

Lines 7-9 of code respectively detect whether the length of string is less than 0, whether the value of end is less than start, and whether the length of string is less than end.

If the result is true, an error will be reported;

The 11th line of code NewSubString(string, start, end) function internally calls the NewProperSubString(str, begin, end) method.

The source code is as follows:

```
Handle<String> Factory::NewProperSubString(Handle<String> str, int begin,
                                                                int end) {
      DCHECK(begin > 0 || end < str->length());
      str = String::Flatten(isolate(), str);
      int length = end - begin;
      if (length <= 0) return empty_string();
      if (length == 1) {
         return LookupSingleCharacterStringFromCode(str->Get(begin)); }
      if (length == 2) {
            uint16_t c1 = str->Get(begin);
            uint16_t c2 = str->Get(begin + 1);
            return MakeOrFindTwoCharacterString(isolate(), c1, c2); }
        if (!FLAG_string_slices || length < SlicedString::kMinLength) {
            if (str->IsOneByteRepresentation()) {
               Handle<SeqOneByteString> result =
                     NewRawOneByteString(length).ToHandleChecked();
               DisallowHeapAllocation no_gc;
               uint8_t* dest = result->GetChars(no_gc);
               String::WriteToFlat(*str, dest, begin, end);
               return result;
            } else {
               Handle<SeqTwoByteString> result =
                     NewRawTwoByteString(length).ToHandleChecked();
               DisallowHeapAllocation no_gc;
               uc16* dest = result->GetChars(no_gc);
               String::WriteToFlat(*str, dest, begin, end);
               return result; }
        }
        int offset = begin;
        if (str->IsSlicedString()) {
            Handle<SlicedString> slice = Handle<SlicedString>::cast(str);
            str = Handle<String>(slice->parent(), isolate());
            offset += slice->offset(); }
        if (str->IsThinString()) {
            Handle<ThinString> thin = Handle<ThinString>::cast(str);
            str = handle(thin->actual(), isolate()); }
        DCHECK(str->IsSeqString() || str->IsExternalString());
        Handle<Map> map = str->IsOneByteRepresentation()
                                          ? sliced_one_byte_string_map()
                                          : sliced_string_map();
        Handle<SlicedString> slice(
               SlicedString::cast(New(map, AllocationType::kYoung)), isolate());
        slice->set_hash_field(String::kEmptyHashField);
        slice->set_length(length);
        slice->set_parent(*str);
46.         slice->set_offset(offset);
47.         return slice;
48. }
```

In the above code, the 4th line of code flattens the string, which will be explained later;

Lines 5-6 of code determine whether to return an empty string;

Lines 7-8 of code handle the case where the length value is 1, which is substring(1);

Lines 9-12 of code handle the case where the length value is 2, which is substring(2);

Line 13 of code! FLAG_string_slices does not allow slicing. For example, in result = example.substring(x,y); in this statement, it cannot be used.

Use **offset** and **length** to represent the result; length < SlicedString::kMinLength means that when the length of the result is less than the maximum length of the slice,

When the limit is small, slicing is not used to represent the result;

Lines 14-19 of code process single-byte strings, apply for length-length heap space (dest), and then copy the characters between begin and end.

into dest;

Lines 22-26 of code process double-byte strings in the same way as above;

Important note: In "result = example.substring(x,y)", the substring processing method when example is of slice or Thin type:

(1) Slice type: Use **offset** and **length** to obtain a partially sliced string in the parent string (parent)

After the code in lines 30-31 detects that example is of slice type, it converts it into a slice type string;

Line 32 of the code obtains the parent string (str) of the slice string;

Line 33 of the code slices the starting position (offset) of the string in str plus begin to get the super-starting position of substring in str;

Lines 41-47 of the code apply for a new slice string heap space, point the parent pointer of the heap space to the parent (str), and then set the new offset and

length, return result;

(2) Thin type: a string that directly references another string object

After the code in lines 34-36 detects that example is of Thin type, it will be converted into a Thin type string; the referenced string is obtained;

Lines 41-47 are the same as above.

The following explains the important methods used in NewProperSubString:

(1) Flatten method, the source code is as follows:

```
1. Handle<String> String::Flatten(Isolate* isolate, Handle<String> string,
2.                                       AllocationType allocation) {
3. if (string->IsConsString()) {
4.         Handle<ConsString> cons = Handle<ConsString>::cast(string);
5.         if (cons->IsFlat()) {
6.             string = handle(cons->first(), isolate);
7.         } else {
8.             return SlowFlatten(isolate, cons, allocation); } }
9. if (string->IsThinString()) {
10.            string = handle(Handle<ThinString>::cast(string)->actual(), isolate);
11.            DCHECK(!string->IsConsString()); }
12. return string;}
```

The above code converts ConsString and ThinString into contiguously stored strings. The SlowFlatten method in line 8 uses a loop to

The two substrings of ConsString are combined into a continuous string. The source code is as follows:

```
1. Handle<String> String::SlowFlatten(Isolate* isolate, Handle<ConsString> cons,
2.                                           AllocationType allocation) {
3.         while (cons->first().length() == 0) {
```

```
4.            if (cons->second().IsConsString() && !cons->second().IsFlat()) {
5.                cons = handle(ConsString::cast(cons->second()), isolate);
6.            } else {
7.                return String::Flatten(isolate, handle(cons->second(), isolate));
8.            }
9.        }
10.         int length = cons->length();
11.         allocation =
12.                 ObjectInYoungGeneration(*cons) ? allocation : AllocationType::kOld;
13.         Handle<SeqString> result;
14.         if (cons->IsOneByteRepresentation()) {
15.             Handle<SeqOneByteString> flat =
16.                     isolate->factory()
17.                             ->NewRawOneByteString(length, allocation)
18.                             .ToHandleChecked();
19.             DisallowHeapAllocation no_gc;
20.             WriteToFlat(*cons, flat->GetChars(no_gc), 0, length);
twenty one.             result = flat;
twenty two.         } else {
twenty three.             Handle<SeqTwoByteString> flat =
twenty four.                     isolate->factory()
25.                             ->NewRawTwoByteString(length, allocation)
26.                             .ToHandleChecked();
27.             DisallowHeapAllocation no_gc;
28.             WriteToFlat(*cons, flat->GetChars(no_gc), 0, length);
29.             result = flat;
30.         }
31.         cons->set_first(*result);
32.         cons->set_second(ReadOnlyRoots(isolate).empty_string());
33.         DCHECK(result->IsFlat());
34.         return result;
35. }
```

In the above code, lines 3-7 use while to process the two substrings of ConsString respectively. Because substrings can also be of type ConsString, So the Flatten method may be called again;

Lines 10-34 of code handle single-byte and double-byte types of ConsString respectively. The solution is to apply for heap space and copy both parts of ConsString. string into the heap space just applied for, and return the result.

3 Runtime_StringSubstring trigger condition

V8 official documentation explains that --allow-natives-syntax and %-prefix can invoke the Runtime method, but what we need to learn is in what In this case V8 will use the Runtime method to process JavaScript source code. Let's start with CodeStubAssembler::SubString(). Because this method is the first method called by V8, the Runtime method will only be used when this method fails. For a detailed explanation of this method, see Article 28 chapter. Part of the source code of CodeStubAssembler::SubString() is given below:

```
1. TNode<String> CodeStubAssembler::SubString(TNode<String> string,
2.                                             TNode<IntPtrT> from,
3.                                             TNode<IntPtrT> to) {
4.     Label original_string_or_invalid_length(this);
5.        GotoIf(UintPtrGreaterThanOrEqual(substr_length, string_length),
6.                 &original_string_or_invalid_length);
7.        TNode<String> direct_string = to_direct.TryToDirect(&runtime);
8.        TNode<IntPtrT> offset = IntPtrAdd(from, to_direct.offset());
9.        TNode<Int32T> const instance_type = to_direct.instance_type();
10.         BIND(&original_string_or_invalid_length);
11. {//Omit very...
12.            CSA_ASSERT(this, IntPtrEqual(substr_length, string_length));
13.            GotoIf(UintPtrGreaterThan(from, IntPtrConstant(0)), &runtime);
14.         }
15.         BIND(&runtime);
16.         {
17.            var_result =
18.                 CAST(CallRuntime(Runtime::kStringSubstring, NoContextConstant(),
string,
19.                                          SmiTag(from), SmiTag(to)));
20.            Goto(&end);
twenty one.    }}
```

In the above code, the Runtime method will be called when the 7th line of code fails to convert the string into a direct string; the 10th line of code satisfies

The Runtime method will also be called when the original_string_or_invalid_length condition is met; in addition, external strings are also processed using the Runtime method.

(omitted from code). The test cases we constructed are as follows:

```
var str1="hello ~~~~~~~~~";
var str2="Gray beans~~~~~~~~";
var str3="Runtime substring detailed explanation";
var example = str1+str2+str3;
console.log(example.substring(5,15));
```

In this code, example is a ConsString type, and its two substrings are ConsString and double-byte types. example of

The TryToDirect conversion failed causing the Runtime method to be triggered. The constant pool data of the test case code is given below:

```
Constant pool (size = 11)
0000032277382AB1: [FixedArray] in OldSpace
  - map: 0x02c6f4e80169 <Map>
  - length: 11
                   0: 0x032277382a11 <FixedArray[16]>
                   1: 0x0322773828c1 <String[#15]: hello ~~~~~~~~~>
                   2: 0x0322773828a9 <String[#4]: str1>
                   3: 0x0322773828f9 <String[#10]\: \x7070\x8c46~~~~~~~~>
                   4: 0x0322773828e1 <String[#4]: str2>
                   5: 0x032277382939 <String[#19]\: Runtime substring\x8be6\x89e3>
                   6: 0x032277382921 <String[#4]: str3>
                   7: 0x032277382971 <String[#7]: example>
                   8: 0x030903133699 <String[#7]: console>
                   9: 0x030903132cd9 <String[#3]: log>
                  10: 0x03090312bc41 <String[#9]: substring>
Handler Table (size = 0)
```

We can see that str1 is a single-byte string, str2 and str3 are double-byte strings. Figure 1

shows the call stack of Runtime_StringSubstring.

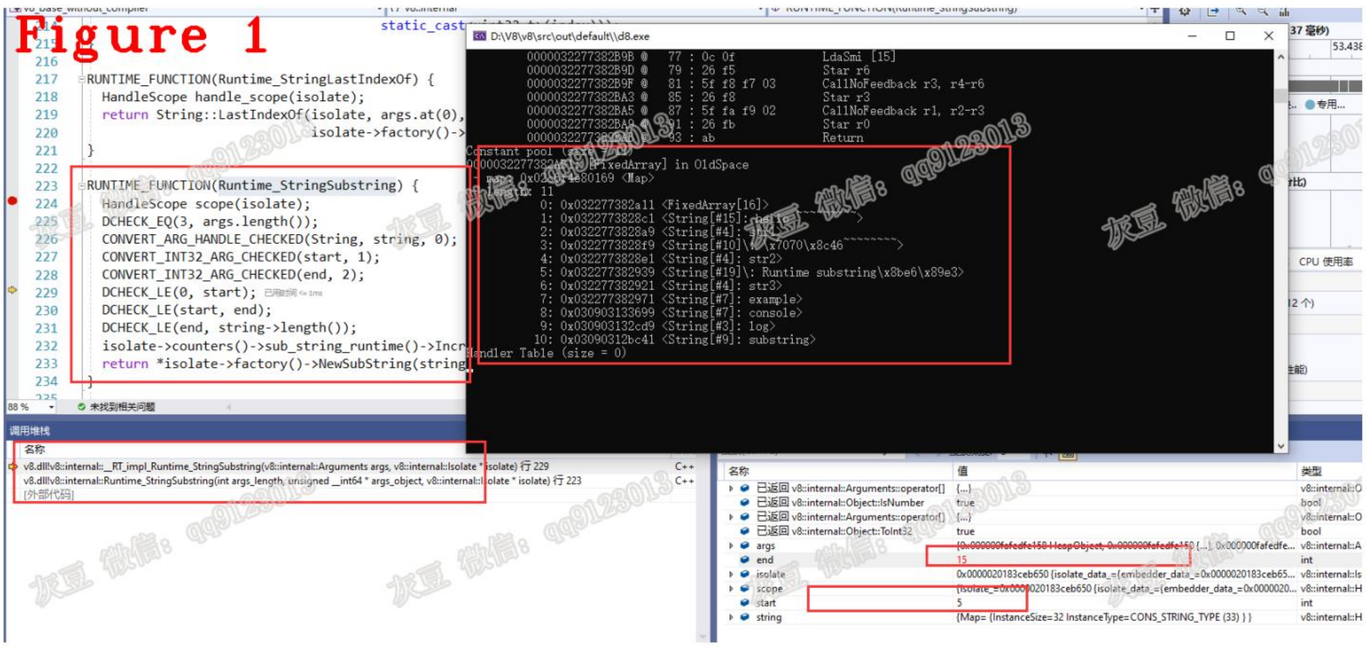


Figure 1

```
RUNTIME_FUNCTION(Runtime_StringLastIndexOf) {
  HandleScope handle_scope(isolate);
  return String::LastIndexOf(isolate, args.at(0),
                             isolate->factory()->
}

RUNTIME_FUNCTION(Runtime_StringSubstring) {
  HandleScope scope(isolate);
  DCHECK_EQ(3, args.length());
  CONVERT_ARG_HANDLE_CHECKED(String, string, 0);
  CONVERT_INT32_ARG_CHECKED(start, 1);
  CONVERT_INT32_ARG_CHECKED(end, 2);
  DCHECK_LE(0, start);
  DCHECK_LE(start, end);
  DCHECK_LE(end, string->length());
  isolate->counters()->sub_string_runtime()->Incr
  return *isolate->factory()->NewSubString(string
}
```

D:\V8\v8\src\out\default\\d8.exe

```
        000003227738 2B9B @   77 : 0c 0f              LdaSmi [15]
        0000032277382B9D @   79 : 26 f5              Star r6
        0000032277382B9F @   81 : 5f f8 f7 03        CallNoFeedback r3, r4-r6
        0000032277382BA3 @   85 : 26 f8              Star r3
        0000032277382BA5 @   87 : 5f fa f9 02        CallNoFeedback r1, r2-r3
        0000032277382BA9 @   91 : 26 fb              Star r0
        0000032277382BAB @   93 : ab                 Return
Constant pool (size = 11)
0000032277382A81: [FixedArray] in OldSpace
 - map: 0x0230f4e80169 <Map>
 - length: 11
           0: 0x032277382a11 <FixedArray[16]>
           1: 0x0322773828c1 <String[#15]: hello~~~~~~~~~>
           2: 0x0322773828a9 <String[#4]: str1>
           3: 0x0322773828f9 <String[#10]\: \x7070\x8c46~~~~~~~~>
           4: 0x0322773828e1 <String[#4]: str2>
           5: 0x032277382939 <String[#19]\: Runtime substring\x8be6\x89e3>
           6: 0x032277382921 <String[#4]: str3>
           7: 0x032277382971 <String[#7]: example>
           8: 0x030903133699 <String[#7]: console>
           9: 0x030903132cd9 <String[#3]: log>
          10: 0x03090312bc41 <String[#9]: substring>
Handler Table (size = 0)
```

调用堆栈

```
名称
v8.dll!v8::internal::__RT_impl_Runtime_StringSubstring(v8::internal::Arguments args, v8::internal::Isolate * isolate) 行 229    C++
v8.dll!v8::internal::Runtime_StringSubstring(int args_length, unsigned __int64 * args_object, v8::internal::Isolate * isolate) 行 223    C++
[外部代码]
```

名称	值	类型
已返回 v8::internal::Arguments::operator[]	{...}	v8::internal::O
已返回 v8::internal::Object::IsNumber	true	bool
已返回 v8::internal::Arguments::operator[]	{...}	v8::internal::O
已返回 v8::internal::Object::ToInt32	true	bool
args	{0x000000fafedfe158 HeapObject, 0x000000fafedfe150 {...}, 0x000000fafedfe...	v8::internal::A
end	15	int
isolate	0x0000020183ceb650 {isolate_data_={embedder_data_=0x0000020183ceb65...	v8::internal::Is
scope	{isolate_=0x0000020183ceb650 {isolate_data_={embedder_data_=0x0000020...	v8::internal::H
start	5	int
string	{Map= {InstanceSize=32 InstanceType=CONS_STRING_TYPE (33) } }	v8::internal::H

You can see in the figure that the values of start and end are 5 and 15 respectively.

Technical

summary (1) SeqString, a continuously stored string in the V8 heap, divided into two categories: OneByte and

TwoByte; **(2)** ConsString, a spliced (first + second) string expressed in the form of a pointer; **(3)**

SliceString, using offset and length are strings that express the partial slice content of the parent string (parent); **(4)** ThinString,

a string that directly references another string object; **(5)** ExternalString, a

string outside the V8 heap space. 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/264196 Ankangke - Thoughtful new safety media