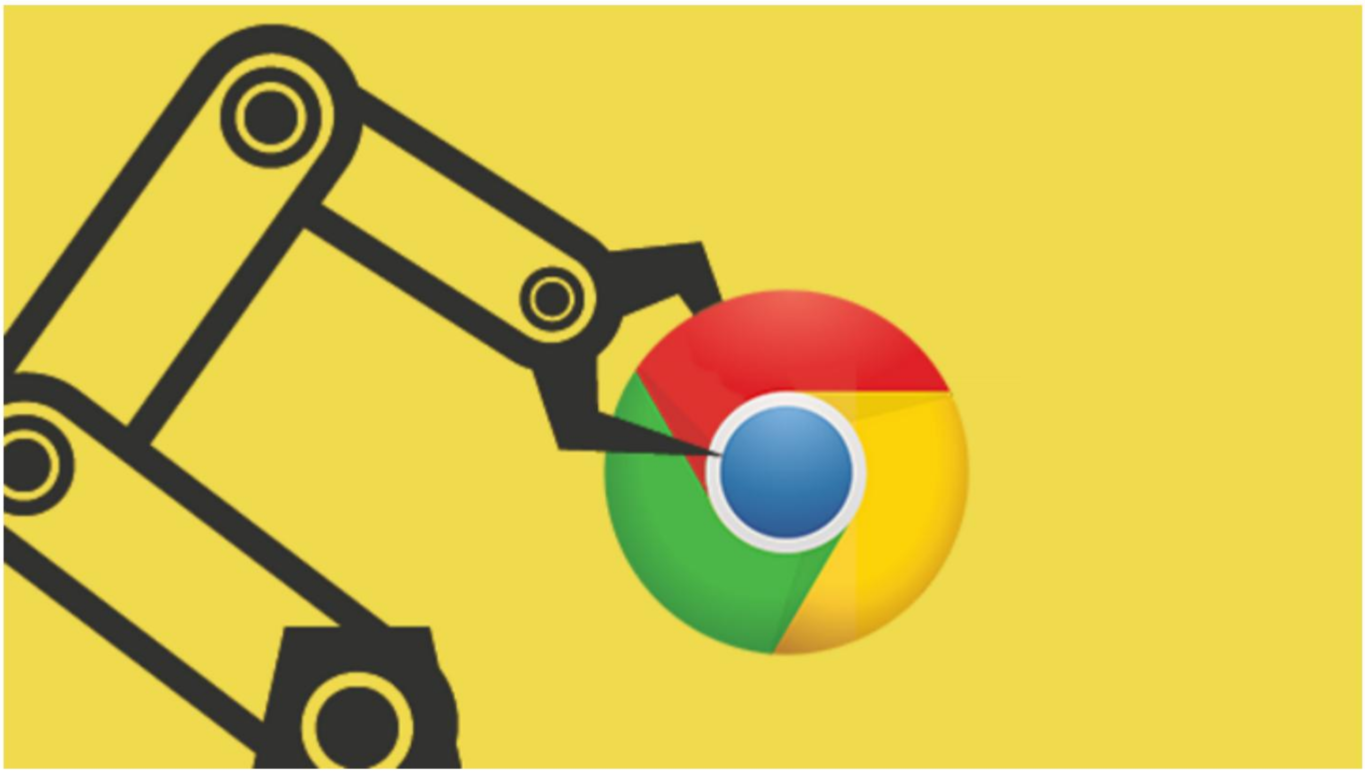
"Chrome V8 Source Code" 35. String.prototype.CharAt source code analysis



1 Introduction

This article first explains the String.prototype.CharAt source code and related data structures, and then demonstrates the calling, loading and execution process of String.prototype.CharAt

through test cases. **Note** (1) Sea of Nodes is the leading knowledge of this article. Please refer to Cliff's 1993 paper From Quads to Graphs. (2) The environment used in this article is: V8 7.9, win10 x64, VS2019.

The test examples for this article are as follows:

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

2 CharAt source code

String.prototype.CharAt is implemented by TF_Bultin and the number is 885. The TF_BUILTIN macro template is used in the String.prototype.CharAt source code. The source code after the macro template is expanded is as follows:

```
class StringPrototypeCharAtAssembler : public CodeStubAssembler {
  public:
    using Descriptor = Builtin_StringPrototypeCharAt_InterfaceDescriptor;
    explicit StringPrototypeCharAtAssembler(compiler::CodeAssemblerState*
state)
        : CodeStubAssembler(state) {}
    void GenerateStringPrototypeCharAtImpl(); Node*
    Parameter(Descriptor::ParameterIndices index) {/*omitted*/ }; void        }

  Builtins::Generate_StringPrototypeCharAt(compiler::CodeAssemblerState*
state) {
      StringPrototypeCharAtAssembler assembler(state); //Omit...

      assembler.GenerateStringPrototypeCharAtImpl();

   } void StringPrototypeCharAtAssembler::GenerateStringPrototypeCharAtImpl(){
//Omit...
   if (block0.is_used()) {
      compiler::TNode<Context> tmp0;
      compiler::TNode<Object> tmp1;
      compiler::TNode<Object> tmp2;
      ca_.Bind(&block0, &tmp0, &tmp1, &tmp2);
      ca_.SetSourcePosition("../../../src/builtins/string.tq", 77);
      compiler::TypedCodeAssemblerVariable<String> result_0_0(&ca_);
      compiler::TypedCodeAssemblerVariable<IntPtrT> result_0_1(&ca_);
      compiler::TypedCodeAssemblerVariable<IntPtrT> result_0_2(&ca_);
      compiler::CodeAssemblerLabel label0(&ca_);
      compiler::CodeAssemblerLabel label1(&ca_);
      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);
      if (label0.is_used()) {
         ca_.Bind(&label0);
         ca_.Goto(&block5, tmp0, tmp1, tmp2, tmp1, tmp2, result_0_0.value(),
result_0_1.value(), result_0_2.value());
      } }
   if (block5.is_used()) {
//Omit...
      ca_.Bind(&block5, &tmp3, &tmp4, &tmp5, &tmp6, &tmp7, &tmp8, &tmp9,
&tmp10);
      ca_.Goto(&block4, tmp3, tmp4, tmp5, tmp8, tmp9, tmp10);
   }
   if (block4.is_used()) {
//Omit...
      ca_.Bind(&block4, &tmp16, &tmp17, &tmp18, &tmp19, &tmp20, &tmp21);
      ca_.SetSourcePosition("../../../src/builtins/string.tq", 81);
      compiler::TNode<Int32T> tmp22;
      USE(tmp22);
      tmp22 =
CodeStubAssembler(state_).StringCharCodeAt(compiler::TNode<String>{tmp19},
compiler::TNode<IntPtrT>{tmp20});
      ca_.SetSourcePosition("../../../src/builtins/string.tq", 82);
      compiler::TNode<String> tmp23;
      USE(tmp23);
      tmp23 =
CodeStubAssembler(state_).StringFromSingleCharCode(compiler::TNode<Int32T>
{tmp22});
48.           CodeStubAssembler(state_).Return(tmp23);
49.        }}
```

When compiling String.prototype.CharAt, first call the Builtins::Generate_StringPrototypeCharAt() method (line 9) to generate the intermediate result, and then the compiler::CodeAssembler::GenerateCode() method compiles the intermediate result into binary code and stores it in butiltin_ in the array. The key functions of StringPrototypeCharAtAssembler::GenerateStringPrototypeCharAtImpl() (line 14) are as follows Down:

(1) Line 18 of code tmp1 represents the string "hello V8" in the test case;

(2) Line 19 of code tmp2 represents the position in the test case, and the value of position is 2;

(3) Line 27 of the code GenerateStringAt_336() determines whether the type of tmp1 is string and whether tmp2 is less than the length of tmp1. If the judgment result is true, execute 32 lines of code;

(4) Lines 34-35 of code bind parameters and jump to line 37;

(5) Lines 40-43 of the code call StringCharCodeAt() to obtain the character (tmp2) at the specified position and store it in tmp22. use The StringFromSingleCharCode() method converts tmp22 to the final result and returns this result on line 48. At this point, CharAt has been executed complete.

The following explains several important methods used by GenerateStringPrototypeCharAtImpl.

(1) GenerateStringAt_336() determines whether the string type and parameter length are correct. The source code is as follows:

```
1. void GenerateStringAt_336(/*omitted*/) {
2. ca_.Goto(&block0, p_context, p_receiver, p_position);
3. //Omit........
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.        }//Omit........
20.        if (block4.is_used()) {
21. //Omit.........
twenty two.   ca_.Branch(tmp22, &block5, &block6, tmp12, tmp13, tmp14, tmp15, tmp16,
tmp18, tmp19);
twenty three. }
twenty four.     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);} }
```

In the above code, line 8 converts the type of this (the string in the test case) to a string;

Lines 9-12 of code determine whether the length of position (2 in the test case) needs to be truncated;

Line 17 of the code determines whether the length of position meets the requirements, and the result is stored in tmp6;

Line 18 of the code will jump based on the result of tmp6. If the result is true, it will jump to block4;

The 21st line of code determines whether the position is less than the length of the string. If it is less than the length of the string, it means that the CharAt() operation is not out of bounds (InBound), otherwise it means OutBound, OutBound means that the final result is empty.

(2) StringCharCodeAt() is defined in class CodeStubAssembler. The source code is as follows:

```
TNode<Int32T> CodeStubAssembler::StringCharCodeAt(SloppyTNode<String> string,
                                                  SloppyTNode<IntPtrT> index)
{
    CSA_ASSERT(this, IsString(string));
    CSA_ASSERT(this, IntPtrGreaterThanOrEqual(index, IntPtrConstant(0)));
    CSA_ASSERT(this, IntPtrLessThan(index, LoadStringLengthAsWord(string)));
    TVARIABLE(Int32T, var_result);
    Label return_result(this), if_runtime(this, Label::kDeferred),
        if_stringistwobyte(this), if_stringisonebyte(this);
    ToDirectStringAssembler to_direct(state(), string);
     to_direct.TryToDirect(&if_runtime);
     TNode<IntPtrT> const offset = IntPtrAdd(index, to_direct.offset());
     TNode<Int32T> const instance_type = to_direct.instance_type();
     TNode<RawPtrT> const string_data = to_direct.PointerToData(&if_runtime);
     // Check if the {string} is a TwoByteSeqString or a OneByteSeqString.
     Branch(IsOneByteStringInstanceType(instance_type), &if_stringisonebyte,
                &if_stringistwobyte);
     BIND(&if_stringisonebyte);
     {
         var_result =
             UncheckedCast<Int32T>(Load(MachineType::Uint8(), string_data, offset));
         Goto(&return_result);
     }
     BIND(&if_stringistwobyte);
     {
         var_result =
             UncheckedCast<Int32T>(Load(MachineType::Uint16(), string_data,
                                        WordShl(offset, IntPtrConstant(1))));
         Goto(&return_result);
     }
     BIND(&if_runtime);
     {
         TNode<Object> result = CallRuntime(
             Runtime::kStringCharCodeAt, NoContextConstant(), string, SmiTag(index));
         var_result = SmiToInt32(CAST(result));
35.         Goto(&return_result);
36.       }
37.       BIND(&return_result);
38.       return var_result.value();
39. }
```

In the above code, lines 2-5 determine whether the string type is correct and whether the index is greater than or equal to zero and less than the string length;

Line 6 of code applies for the Int32 type variable var_result, which is used to store the return value;

Line 7 of the code applies for four label variables, which will cooperate with Branch to complete function jumps in the future;

The function of line 10 of the code to_direct.TryToDirect(&if_runtime) is to convert the indirect string of flat, thin or slice type into direct string. If the conversion fails, jump to the 30th line of code and use runtime to process the string; if the conversion is successful, return to string_data;

Line 15 of code determines whether the string type is single byte or double byte;

Line 19 of code uses single-byte mode to read bytes from the offset position of string_data and store them in var_result;

Line 25 of code uses double-byte mode to read bytes from the offset position of string_data and store them in var_result;

The 32nd line of code uses the runtime method to implement the StringChartAt function, which will be explained in a subsequent article.

(3) StringFromSingleCharCode() is defined in CodeStubAssembler, and its function is to convert Int values into strings.

3 CharAt test

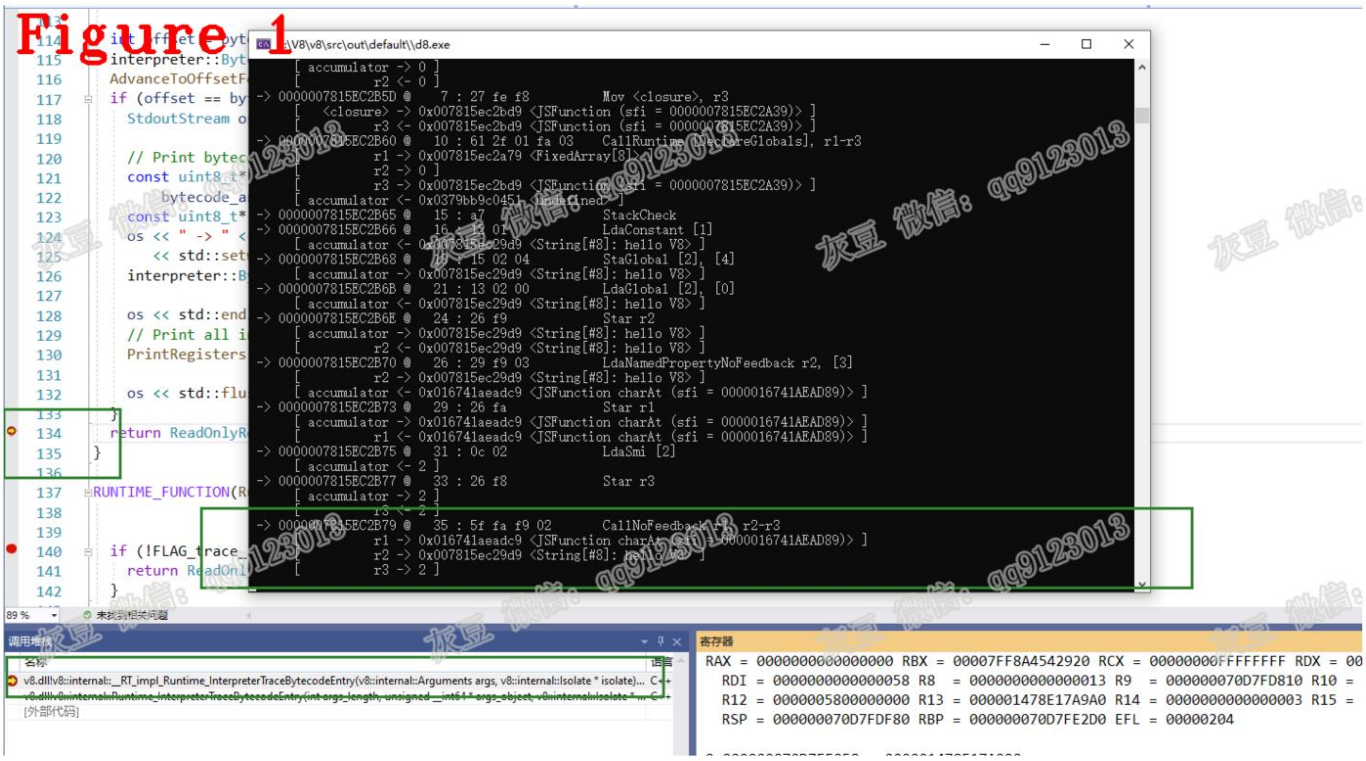
The bytecode of the test code is as follows:

```
//Separator line............//Omitted part of the bytecode
         8 S> 0000012A281C2B26 @ 16 : 12 01                   LdaConstant [1]
         8 E> 0000012A281C2B28 @ 18 : 15 02 04                StaGlobal [2], [4]
        29 S> 0000012A281C2B2B @ 21 : 13 02 00                LdaGlobal [2], [0]
              0000012A281C2B2E @ 24 : 26 f9                   Star r2
        31 E> 0000012A281C2B30 @ 26 : 29 f9 03
LdaNamedPropertyNoFeedback r2, [3]
              0000012A281C2B33 @ 29 : 26 fa                   Star r1
              0000012A281C2B35 @ 31 : 0c 02                   LdaSmi [2]
              0000012A281C2B37 @ 33 : 26 f8 31 E>             Star r3
      0000012A281C2B39 @ 35 : 5f fa f9 02 29 E> 0             CallNoFeedback r1, r2-r3
      000012A281C2B3D @ 39 : 15 04 06 43 S>                   StaGlobal [4], [6]
      0000012A281C2B40 @ 42 : 13 05 08 0000012A281C2B43       LdaGlobal [5], [8]
              @ 45 : 26 f9 51 E> 0000012A281C2B45 @           Star r2
      47 : 29 f9 06 LdaNamedPropertyNoFeedback r2, [6]

              0000012A281C2B48 @ 50 : 26 fa                   Star r1
      55 E> 0000012A281C2B4A @ 52 : 13 04 02                  LdaGlobal [4], [2]
              0000012A281C2B4D @ 55 : 26 f8                   Star r3
      51 E> 0000012A281C2B4F @ 57 : 5f fa f9 02               CallNoFeedback r1, r2-r3
              0000012A281C2B53 @ 61 : 26 fb                   Star r0
      58 S> 0000012A281C2B55 @ 63 : ab                        Return
Constant pool (size = 7)
0000012A281C2A99: [FixedArray] in OldSpace
    - map: 0x0148906c0169 <Map>
    - length: 7
            0: 0x012a281c2a39 <FixedArray[8]>
            1: 0x012a281c2999 <String[#8]: hello V8>
```

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

In the above code, the 2nd line of code loads the constant hello v8 into the accumulation register; the 3-5 lines of code store and read the constant hello v8, which is eventually stored in the r2 register; the 6th line of code loads String.prototype. CharAt(), line 10 of the code calls String.prototype.CharAt; lines 25-31 are constant pools used to store constants that need to be used during runtime.



```
    [ accumulator -> 0 ]
    [          r2 <- 0 ]
 -> 0000007815EC2B5D @    7 : 27 fe f8          Mov <closure>, r3
    [    <closure> -> 0x007815ec2bd9 <JSFunction (sfi = 0000007815EC2A39)> ]
    [          r3 <- 0x007815ec2bd9 <JSFunction (sfi = 0000007815EC2A39)> ]
 -> 0000007815EC2B60 @   10 : 61 2f 01 fa 03    CallRuntime [DeclareGlobals], r1-r3
    [          r1 -> 0x007815ec2a79 <FixedArray[8]> ]
    [          r2 -> 0 ]
    [          r3 -> 0x007815ec2bd9 <JSFunction (sfi = 0000007815EC2A39)> ]
    [ accumulator <- 0x0379bb9c0451 <undefined> ]
 -> 0000007815EC2B65 @   15 : a7                StackCheck
 -> 0000007815EC2B66 @   16 : 12 01             LdaConstant [1]
    [ accumulator <- 0x007815ec29d9 <String[#8]: hello V8> ]
 -> 0000007815EC2B68 @   18 : 15 02 04          StaGlobal [2], [4]
    [ accumulator -> 0x007815ec29d9 <String[#8]: hello V8> ]
 -> 0000007815EC2B6B @   21 : 13 02 00          LdaGlobal [2], [0]
    [ accumulator <- 0x007815ec29d9 <String[#8]: hello V8> ]
 -> 0000007815EC2B6E @   24 : 26 f9             Star r2
    [ accumulator -> 0x007815ec29d9 <String[#8]: hello V8> ]
    [          r2 <- 0x007815ec29d9 <String[#8]: hello V8> ]
 -> 0000007815EC2B70 @   26 : 29 f9 03          LdaNamedPropertyNoFeedback r2, [3]
    [          r2 -> 0x007815ec29d9 <String[#8]: hello V8> ]
    [ accumulator <- 0x016741aeadc9 <JSFunction charAt (sfi = 000016741AEAD89)> ]
 -> 0000007815EC2B73 @   29 : 26 fa             Star r1
    [ accumulator -> 0x016741aeadc9 <JSFunction charAt (sfi = 000016741AEAD89)> ]
    [          r1 <- 0x016741aeadc9 <JSFunction charAt (sfi = 000016741AEAD89)> ]
 -> 0000007815EC2B75 @   31 : 0c 02             LdaSmi [2]
    [ accumulator <- 2 ]
 -> 0000007815EC2B77 @   33 : 26 f8             Star r3
    [ accumulator -> 2 ]
    [          r3 <- 2 ]
 -> 0000007815EC2B79 @   35 : 5f fa f9 02       CallNoFeedback r1, r2-r3
    [          r1 -> 0x016741aeadc9 <JSFunction charAt (sfi = 000016741AEAD89)> ]
    [          r2 -> 0x007815ec29d9 <String[#8]: hello V8> ]
    [          r3 -> 2 ]
```

```
名称                                                                                                   语言
v8.dll!v8::internal::_RT_impl_Runtime_InterpreterTraceBytecodeEntry(v8::internal::Arguments args, v8::internal::Isolate * isolate)...  C++
~~v8.dll!v8::internal::Runtime_InterpreterTraceBytecodeEntry(int args_length, unsigned __int64 * args_object, v8::internal::Isolate * ...~~  C++
[外部代码]
```

```
RAX = 0000000000000000 RBX = 00007FF8A4542920 RCX = 00000000FFFFFFFF RDX = 00
  RDI = 0000000000000058 R8  = 0000000000000013 R9  = 000000070D7FD810 R10 =
  R12 = 0000005800000000 R13 = 000001478E17A9A0 R14 = 0000000000000003 R15 =
  RSP = 000000070D7FDF80 RBP = 000000070D7FE2D0 EFL = 00000204
```

Debugging the execution of bytecode CallNoFeedback can see the execution process of String.prototype.CharAt. The debugging method is: Setting FLAG_trace_ignition = true; set a breakpoint in Runtime_InterpreterTraceBytecodeEntry; use assembly for debugging at the "CallNoFeedback r1, r2-r3" position in Figure 1.

Technical summary

(1) The binary code (code) generated by compiling String.prototype.CharAt is stored in the builtin_ array. When interpreting and executing JavaScript source code, code will be used instead of using the source code of StringCharAt discussed in this article; **(2)** String details in V8 Divided into single-byte, double-byte, flat and other types; **(3)** Indirect type strings can only be processed by runtime. Using to_direct.TryToDirect() to determine the string type first can save unnecessary runtime operations;

(4) The working process of String.prototype.CharAt is: determine whether the string type and position are correct, determine whether the position is out of bounds, use the Builtin method to process direct strings, and use the runtime method to process indirect strings.

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 Zhihu: https:// www.zhihu.com/people/

v8blink This article was originally published by Gray Bean and reprinted from: https:// www. anquanke.com/post/id/263308 Anquanke - Thoughtful new security media