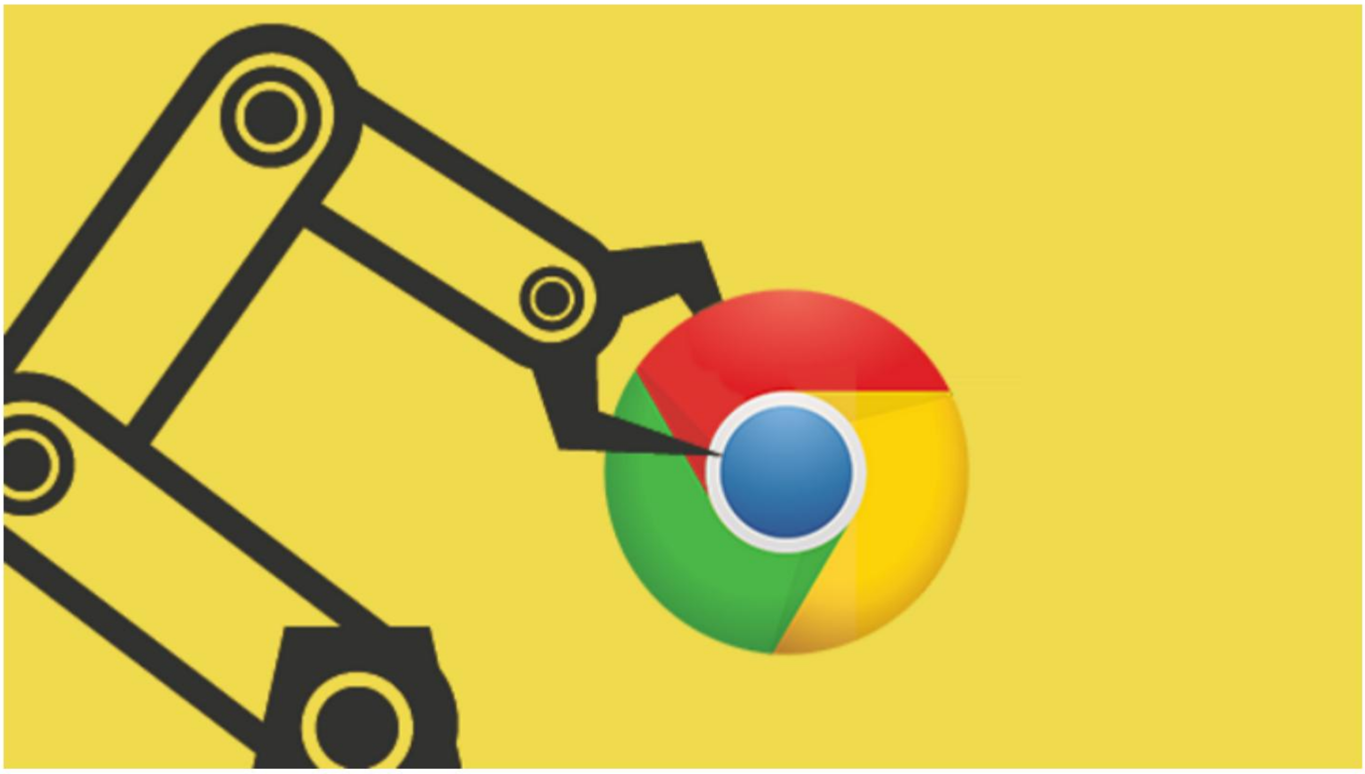
"Chrome V8 Source Code" 29. Detailed explanation of the CallBuiltin() calling process



1 abstract

This article is the fifth of the Builtin topic and analyzes the calling process of Builtin in detail. It is the most common situation to use CallBuiltin() to call Builtin in Bytecode handler. This article will explain the source code and related data structure of CallBuiltin() in detail. The content of this article is organized as follows: important data structures (Chapter 2); CallBuiltin() source code (Chapter 3).

2 Data structure

Tip: Just-In-Time Compiler is the introductory knowledge of this article, please read it by yourself.

The calling process of Builtin is mainly divided into two parts: querying the Builtin table to find the corresponding entry function and calculating the calldescriptor. The following explains the

relevant data structure: **(1)** Builtin name (for example, Builtin::kStoreGlobalIC). The name is an enumeration type variable. CallBuiltin() uses this name to query the Builtin table and find the corresponding entry function. The source code is as follows:

```
class Builtins
{ //.............Omitted............................
    enum Name : int32_t { #define
DEF_ENUM(Name, ...) k##Name,
        BUILTIN_LIST(DEF_ENUM, DEF_ENUM, DEF_ENUM, DEF_ENUM, DEF_ENUM, DEF_ENUM,
                                 DEF_ENUM)
#undef DEF_ENUM
                builtin_count,
}
```

After expansion, it looks like this:

```
enum Name:int32_t{kRecordWrite, kEphemeronKeyBarrier,
kAdaptorWithBuiltinExitFrame, kArgumentsAdaptorTrampoline,......}
```

(2) Builtin table stores Builtin address. The location of the Builtin table is isoate->isolate_data_->builtins_. The source code is as follows:

```
class IsolateData final {
  public:
    explicit IsolateData(Isolate* isolate) : stack_guard_(isolate) {}
//............omitted............................
    Address* builtins() { return builtins_; }
}
```

builtins_ is an array of Address type. Used in conjunction with enum Name:int32_t{}, the corresponding Builtin address can be queried (the second line below represents code to complete the address query), the source code is as follows:

```
1.Callable Builtins::CallableFor(Isolate* isolate, Name name) {
2. Handle<Code> code = isolate->builtins()->builtin_handle(name);
3. return Callable{code, CallInterfaceDescriptorFor(name)};
4.}
```

CallInterfaceDescriptorFor in line 3 of the above code returns Builtin's calling information, which together with the code constitutes Callable.

(3) Code class, which includes the Builtin address, the start and end of the instruction, and filling information. One of its functions is to create a snapshot file.

The source code is as follows:

```
class Code : public HeapObject {
public:
#define CODE_KIND_LIST(V) \
      V(OPTIMIZED_FUNCTION) V(BYTECODE_HANDLER)              \
      V(STUB) V(BUILTIN) V(REGEXP) V(WASM_FUNCTION) V(WASM_TO_CAPI_FUNCTION)
\
        V(WASM_TO_JS_FUNCTION) V(JS_TO_WASM_FUNCTION) V(JS_TO_JS_FUNCTION)
\
        V(WASM_INTERPRETER_ENTRY) V(C_WASM_ENTRY)
        inline int builtin_index() const;
        inline int handler_table_offset() const;
          inline void set_handler_table_offset(int offset);
          // The body of all code objects has the following layout.
          // +--------------------------+ <-- raw_instruction_start()
          // | |                  instructions // | |
                                          ...
          // +--------------------------+
          // | | <-- safepoint_table_offset() embedded metadata
17.        // | //                    ...                   | <-- handler_table_offset()
18.        | // | //                                        | <-- constant_pool_offset()
19.        | //                                             | <-- code_comments_offset()
20.                                                         |
twenty one.        +--------------------------+ <-- raw_instruction_end()
twenty two.        // If has_unwinding_info() is false, raw_instruction_end() points to the
first
twenty three.        // memory location after the end of the code object. Otherwise, the body
twenty four.        // continues as follows:
25.        // +--------------------------+
26.        // | |          padding to the next // | 8-
27.        byte aligned address |
28.        // +--------------------------+ <-- raw_instruction_end()
29.        // | [unwinding_info_size] |
30.        // | |                 as uint64_t //
31.        +--------------------------+ <-- unwinding_info_start()
32.        // | unwinding info // | //                           |
33.                                       ...                   |
34.        +--------------------------+ <-- unwinding_info_end()
35.        // and unwinding_info_end() points to the first memory location after the
end
36.        // of the code object.
37.     };
```

Lines 3-7 of the above code explain which instruction type the current Code is; codes 9-10 are exception handlers; comments on lines 11-36 explain the Code

Memory layout. The memory layout will change slightly when writing snapshot files. For details, please refer to the mksnapshot.exe source code.

(4) CallInterfaceDescriptor describes the register parameters, stack parameters, return values and other information of the Builtin entry function. Call

This information will be used when Builtin, the source code is as follows:

```
class V8_EXPORT_PRIVATE CallInterfaceDescriptor {
public:
      Flags flags() const { return data()->flags(); }
      bool HasContextParameter() const {return (flags() &
CallInterfaceDescriptorData::kNoContext) == 0;}
      int GetReturnCount() const { return data()->return_count(); }
      MachineType GetReturnType(int index) const {return data()-
>return_type(index);}
      int GetParameterCount() const { return data()->param_count(); }
int GetRegisterParameterCount() const {return data()-
>register_param_count();}
      int GetStackParameterCount() const {return data()->param_count() - data()-
>register_param_count();}
       Register GetRegisterParameter(int index) const {return data()-
>register_param(index);}
       MachineType GetParameterType(int index) const {return data()-
>param_type(index);}
       RegList allocatable_registers() const {return data()-
>allocatable_registers();}
//............omitted...........................
private :
```

```
15. const CallInterfaceDescriptorData* data_;
16. }
```

Line 5 of the above code is the number of return values of Builtin; line 6 is the type of return value; line 7 is the number of parameters; line 8 is the number of register parameters.

amount; line 9 is the number of stack parameters; lines 10-12 are to obtain parameters; line 15 of code CallInterfaceDescriptorData stores the above code

The information required in the code, that is, the number and type of return values, is as follows:

```
1. class V8_EXPORT_PRIVATE CallInterfaceDescriptorData {
2.private :
3.        bool IsInitializedPlatformSpecific() const {
4. //.........omitted.......................
5.        }
6.        bool IsInitializedPlatformIndependent() const {
7. //.........omitted.......................
8.        }
9.        int register_param_count_ = -1;
10.        int return_count_ = -1;
11.        int param_count_ = -1;
12.        Flags flags_ = kNoFlags;
13.        RegList allocatable_registers_ = 0;
14.        Register* register_params_ = nullptr;
15.        MachineType* machine_types_ = nullptr;
16.        DISALLOW_COPY_AND_ASSIGN(CallInterfaceDescriptorData);
17. };
```

The variables defined in lines 9-15 of the above code are the return values, parameters and other information mentioned in CallInterfaceDescriptor. The above content is

The main data structure used by CallBuiltin().

3 CallBuiltin()

Let's look at the following usage scenarios:

```
IGNITION_HANDLER(LdaNamedPropertyNoFeedback, InterpreterAssembler) {
   TNode<Object> object = LoadRegisterAtOperandIndex(0);
   TNode<Name> name = CAST(LoadConstantPoolEntryAtOperandIndex(1));
   TNode<Context> context = GetContext();
   TNode<Object> result = CallBuiltin(Builtins::kGetProperty, context, object,
name);
   SetAccumulator(result);
   Dispatch();
}
```

The function of LdaNamedPropertyNoFeedback is to obtain attributes, such as getting the getelementbyID method from the document attribute.

The method is obtained by CallBuiltin calling Builtins::kGetProperty. The source code is as follows:

```
template <class... TArgs>
TNode<Object> CallBuiltin(Builtins::Name id, SloppyTNode<Object> context,
                                                     TArgs... args) {
    return CallStub<Object>(Builtins::CallableFor(isolate(), id), context,
                                                     args...);
}
```

In the above code, id represents the name of Builtin, which is the enumeration value mentioned earlier; args has two members: args[0] represents object (in the above example document), args[1] represents name (getelementbyID method). The source code of CallStub() is as follows:

```
1.          template <class T = Object, class... TArgs>
2.          TNode<T> CallStub(Callable const& callable, SloppyTNode<Object> context,
3.                                           TArgs... args) {
4.              TNode<Code> target = HeapConstant(callable.code());
5.              return CallStub<T>(callable.descriptor(), target, context, args...);
6.          }
7. //............separator line.......................
8.          template <class T = Object, class... TArgs>
9.          TNode<T> CallStub(const CallInterfaceDescriptor& descriptor,
10.                                          SloppyTNode<Code> target, SloppyTNode<Object> context,
11.                                          TArgs... args) {
12.               return UncheckedCast<T>(CallStubR(StubCallMode::kCallCodeObject,
descriptor,
13.                                                                    1, target, context, args...));
14.           }
15. //............Separator line............
16.           template <class... TArgs>
17.           Node* CallStubR(StubCallMode call_mode,
18.                                        const CallInterfaceDescriptor& descriptor, size_t
result_size,
19.                                        SloppyTNode<Object> target, SloppyTNode<Object> context,
20.                                        TArgs... args) {
twenty one.          return CallStubRImpl(call_mode, descriptor, result_size, target, context,
twenty two.                                            {args...});
twenty three.     }
```

The 4th line of the above code creates the target object, which is the entry address of Builtin; the 5th line of code calls the CallStub() method (line 9), and finally Enter CallStubR(). Call CallStubRImpl() in CallStubR(), the source code is as follows:

```
1. Node* CodeAssembler::CallStubRImpl( ) {
2.        DCHECK(call_mode == StubCallMode::kCallCodeObject ||
3.                      call_mode == StubCallMode::kCallBuiltinPointer);
4.        constexpr size_t kMaxNumArgs = 10;
5.        DCHECK_GE(kMaxNumArgs, args.size());
6.          NodeArray<kMaxNumArgs + 2> inputs;
7.          inputs.Add(target);
8.          for (auto arg : args) inputs.Add(arg);
9.          if (descriptor.HasContextParameter()) {
```

```
10.         inputs.Add(context);
11.       }
12.       return CallStubN(call_mode, descriptor, result_size, inputs.size(),
13.                        inputs.data());
14. }
```

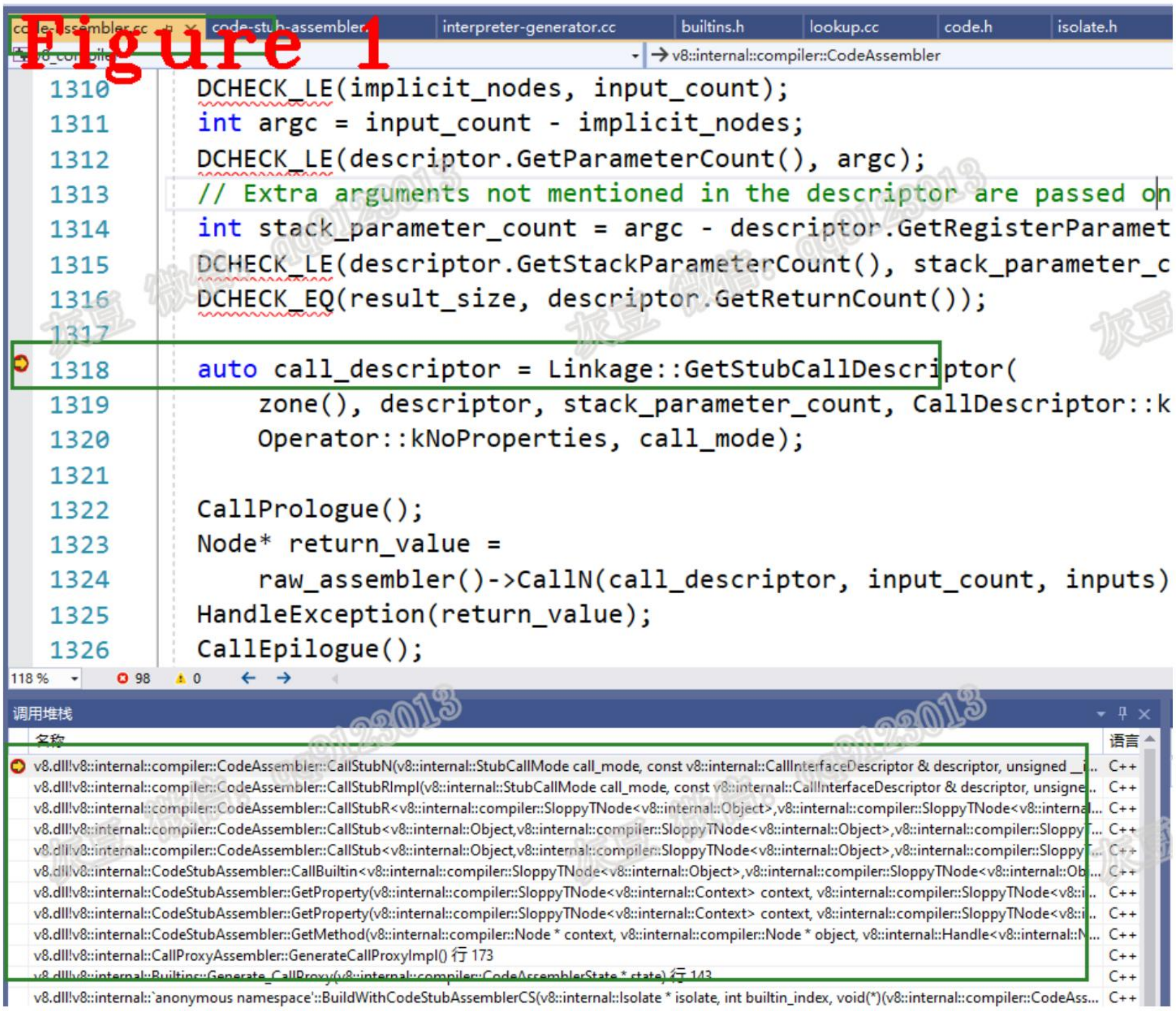
Lines 7-10 of the above code add all parameters to the array inputs. The contents of the inputs are: Builtin's entry address (code type), object, name, context. Entering the 12th line of code, the source code of CallStubN() is as follows:

```
1. Node* CodeAssembler::CallStubN() {
2.     // implicit nodes are target and optionally context.
3.     int implicit_nodes = descriptor.HasContextParameter() ? 2 : 1;
4.     int argc = input_count - implicit_nodes;
5.     // Extra arguments not mentioned in the descriptor are passed on the stack.
6.     int stack_parameter_count = argc - descriptor.GetRegisterParameterCount();
7.     auto call_descriptor = Linkage::GetStubCallDescriptor(
8.         zone(), descriptor, stack_parameter_count, CallDescriptor::kNoFlags,
9.         Operator::kNoProperties, call_mode);
10.     CallPrologue();
11.     Node* return_value =
12.         raw_assembler()->CallN(call_descriptor, input_count, inputs);
13.     HandleException(return_value);
14.     CallEpilogue();
15.     return return_value;
16. }
```

The return value type of call_descriptor in line 7 above is as follows:

```
CallDescriptor( kind, //              // --
    kind
    target_type, // target MachineType
    target_loc, // target location
    locations.Build(), // location_sig
    stack_parameter_count, // stack_parameter_count
    properties, // properties
    kNoCalleeSaved, // callee-saved registers
    kNoCalleeSaved, // callee-saved fp
    CallDescriptor::kCanUseRoots | flags, // flags
    descriptor.DebugName(),                  // debug name
    descriptor.allocatable_registers())
```

The above information prepares for calling Builtin. The 11th line of code in CallStubN(): Complete the call to Builtin, the 13th line of code: Exception handling reason. Figure 1 shows the call stack of CodeAssembler(). Builtin is being established at this time. The establishment of Builtin occurs in the Isolate initialization phase.



```
DCHECK_LE(implicit_nodes, input_count);
int argc = input_count - implicit_nodes;
DCHECK_LE(descriptor.GetParameterCount(), argc);
// Extra arguments not mentioned in the descriptor are passed on
int stack_parameter_count = argc - descriptor.GetRegisterParamet
DCHECK_LE(descriptor.GetStackParameterCount(), stack_parameter_c
DCHECK_EQ(result_size, descriptor.GetReturnCount());

auto call_descriptor = Linkage::GetStubCallDescriptor(
    zone(), descriptor, stack_parameter_count, CallDescriptor::k
    Operator::kNoProperties, call_mode);

CallPrologue();
Node* return_value =
    raw_assembler()->CallN(call_descriptor, input_count, inputs)
HandleException(return_value);
CallEpilogue();
```

```
v8.dll!v8::internal::compiler::CodeAssembler::CallStubN(v8::internal::StubCallMode call_mode, const v8::internal::CallInterfaceDescriptor & descriptor, unsigned __i...   C++
v8.dll!v8::internal::compiler::CodeAssembler::CallStubRImpl(v8::internal::StubCallMode call_mode, const v8::internal::CallInterfaceDescriptor & descriptor, unsigne...   C++
v8.dll!v8::internal::compiler::CodeAssembler::CallStubR<v8::internal::compiler::SloppyTNode<v8::internal::Object>,v8::internal::compiler::SloppyTNode<v8::internal...   C++
v8.dll!v8::internal::compiler::CodeAssembler::CallStub<v8::internal::Object,v8::internal::compiler::SloppyTNode<v8::internal::Object>,v8::internal::compiler::SloppyT...   C++
v8.dll!v8::internal::compiler::CodeAssembler::CallStub<v8::internal::Object,v8::internal::compiler::SloppyTNode<v8::internal::Object>,v8::internal::compiler::SloppyT...   C++
v8.dll!v8::internal::CodeStubAssembler::CallBuiltin<v8::internal::compiler::SloppyTNode<v8::internal::Object>,v8::internal::compiler::SloppyTNode<v8::internal::Ob...   C++
v8.dll!v8::internal::CodeStubAssembler::GetProperty(v8::internal::compiler::SloppyTNode<v8::internal::Context> context, v8::internal::compiler::SloppyTNode<v8::i...   C++
v8.dll!v8::internal::CodeStubAssembler::GetProperty(v8::internal::compiler::SloppyTNode<v8::internal::Context> context, v8::internal::compiler::SloppyTNode<v8::i...   C++
v8.dll!v8::internal::CodeStubAssembler::GetMethod(v8::internal::compiler::Node * context, v8::internal::compiler::Node * object, v8::internal::Handle<v8::internal::N...   C++
v8.dll!v8::internal::CallProxyAssembler::GenerateCallProxyImpl() 行 173   C++
v8.dll!v8::internal::Builtins::Generate_CallProxy(v8::internal::compiler::CodeAssemblerState * state) 行 143   C++
v8.dll!v8::internal::`anonymous namespace'::BuildWithCodeStubAssemblerCS(v8::internal::Isolate * isolate, int builtin_index, void(*)(v8::internal::compiler::CodeAss...   C++
```

Technical summary

(1) There is a one-to-one correspondence between the Builtin name and the Builtin entry. This relationship is represented by isoate->isolate_data_->builtins_. builtins_ is the Address array created during the Isolate initialization process; **(2)** inputs array In addition to parameters, there are also target and context; **(3)** Detailed descriptions of Builtin parameters, return values and other information can be viewed in the BUILTIN_LIST macro.

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 is published by Gray Bean Original Reprint Statement, indicate the source: https: //www.anquanke.com/post/id/260901 Anquanke - Thoughtful new security media