《Chrome V8源码》29.CallBuiltin()调用过程详解

1 摘要

本篇文章是Builtin专题的第五篇，详细分析Builtin的调用过程。在Bytecode handler中使用`CallBuiltin()`调用Builtin是最常见的情况，本文将详解`CallBuiltin()`源码和相关的数据结构。本文内容组织方式：重要数据结构（章节2）；`CallBuiltin()`源码（章节3）。

2 数据结构

提示： Just-In-Time Compiler是本文的前导知识，请读者自行查阅。
Builtin的调用过程主要分为两部分：查询Builtin表找到相应的入口函数、计算calldescriptor。 下面解释相关的数据结构：
（1） Builtin名字（例如Builtin::kStoreGlobalIC），名字是枚举类型变量，`CallBuiltin()`使用该名字查询Builtin表，找到相应的入口函数，源码如下：

```
class Builtins {
//..............省略.....................
  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,
}
```

展开之后如下：

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

(2) Builtin表存储Builtin的地址。Builtin表的位置是isoate->isolate_data_->builtins_，源码如下：

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

builtins_是Address类型的数组，与enum Name:int32_t{}配合使用可查询对应的Builtin地址（下面的第2行代码就完成了地址查询），源码如下：

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

上述代码第3行CallInterfaceDescriptorFor返回Builtin的调用信息，该信息与code共同组成了Callable。

(3) Code类，该类包括Builtin地址、指令的开始和结束以及填充信息，它的作用之一是创建snapshot文件，源码如下：

```
1.  class Code : public HeapObject {
2.   public:
3.  #define CODE_KIND_LIST(V)   \
4.    V(OPTIMIZED_FUNCTION) V(BYTECODE_HANDLER)       \
5.    V(STUB)  V(BUILTIN)  V(REGEXP)  V(WASM_FUNCTION)   V(WASM_TO_CAPI_FUNCTION)
\
6.     V(WASM_TO_JS_FUNCTION)   V(JS_TO_WASM_FUNCTION)   V(JS_TO_JS_FUNCTION)
\
7.     V(WASM_INTERPRETER_ENTRY)   V(C_WASM_ENTRY)
8.     inline int builtin_index() const;
9.     inline int handler_table_offset() const;
10.     inline void set_handler_table_offset(int offset);
11.     // The body of all code objects has the following layout.
12.     //  +--------------------------+  <-- raw_instruction_start()
13.     //  |       instructions       |
14.     //  |           ...            |
15.     //  +--------------------------+
16.     //  |     embedded metadata    |  <-- safepoint_table_offset()
```

```
17.     //  |           ...             |  <-- handler_table_offset()
18.     //  |                           |  <-- constant_pool_offset()
19.     //  |                           |  <-- code_comments_offset()
20.     //  |                           |
21.     //  +---------------------------+  <-- raw_instruction_end()
22.     // If has_unwinding_info() is false, raw_instruction_end() points to the
first
23.     // memory location after the end of the code object. Otherwise, the body
24.     // continues as follows:
25.     //  +---------------------------+
26.     //  |    padding to the next    |
27.     //  |  8-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.   };
```

上述代码第3-7行说明了当前Code是哪种指令类型；第9-10代码是异常处理程序；第11-36行注释说明了Code的内存布局。写snapshot文件时内存布局会有细微变化，详情请参考mksnapshot.exe源码。

（4） CallInterfaceDescriptor描述了Builtin入口函数的寄存器参数、堆栈参数和返回值等信息，调用Builtin时会使用这些信息，源码如下：

```
1.  class V8_EXPORT_PRIVATE CallInterfaceDescriptor {
2.   public:
3.    Flags flags() const { return data()->flags(); }
4.    bool HasContextParameter() const {return (flags() &
CallInterfaceDescriptorData::kNoContext) == 0;}
5.    int GetReturnCount() const { return data()->return_count(); }
6.    MachineType GetReturnType(int index) const {return data()-
>return_type(index);}
7.    int GetParameterCount() const { return data()->param_count(); }
8.    int GetRegisterParameterCount() const {return data()-
>register_param_count();}
9.    int GetStackParameterCount() const {return data()->param_count() - data()-
>register_param_count();}
10.    Register GetRegisterParameter(int index) const {return data()-
>register_param(index);}
11.    MachineType GetParameterType(int index) const {return data()-
>param_type(index);}
12.    RegList allocatable_registers() const {return data()-
>allocatable_registers();}
13.  //.............省略.................
14.  private:
```

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

上述代码第5行是Builtin的返回值数量；第6行是返回值的类型；第7行是参数数量；第8代是寄存器参数的数量；第9行是栈参数的数量；第10-12行是获取参数；第15行代码`CallInterfaceDescriptorData`存储上述代码中所需的信息，即返回值数量、类型等信息，源码如下：

```
1.  class V8_EXPORT_PRIVATE CallInterfaceDescriptorData {
2.   private:
3.    bool IsInitializedPlatformSpecific() const {
4.  //.........省略..............
5.    }
6.    bool IsInitializedPlatformIndependent() const {
7.  //.........省略..............
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.  };
```

上述代码第9-15行定义的变量就是`CallInterfaceDescriptor`中提到的返回值、参数等信息。 以上内容是`CallBuiltin()`使用的主要数据结构。

3 CallBuiltin()

来看下面的使用场景：

```
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();
}
```

`LdaNamedPropertyNoFeedback`的作用是获取属性，例如从`document`属性中获取`getelementbyID`方法，该方法的获取由`CallBuiltin`调用`Builtins::kGetProperty`实现，源码如下：

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

上述代码中id代表Builtin的名字，即前面提到的枚举值；args有两个成员：args[0]代表object（上述例子中的document），args[1]代表name（getelementbyID方法）。CallStub()源码如下：

```
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.  //..............分隔线.................
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.  //..............分隔线.................
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) {
21.      return CallStubRImpl(call_mode, descriptor, result_size, target, context,
22.                           {args...});
23.    }
```

上述代码第4行创建target对象，该对象是Builtin的入口地址；第5行代码调用CallStub()方法（第9行），最终进入CallStubR()。在CallStubR()中调用CallStubRImpl(),源码如下：

```
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.   }
```

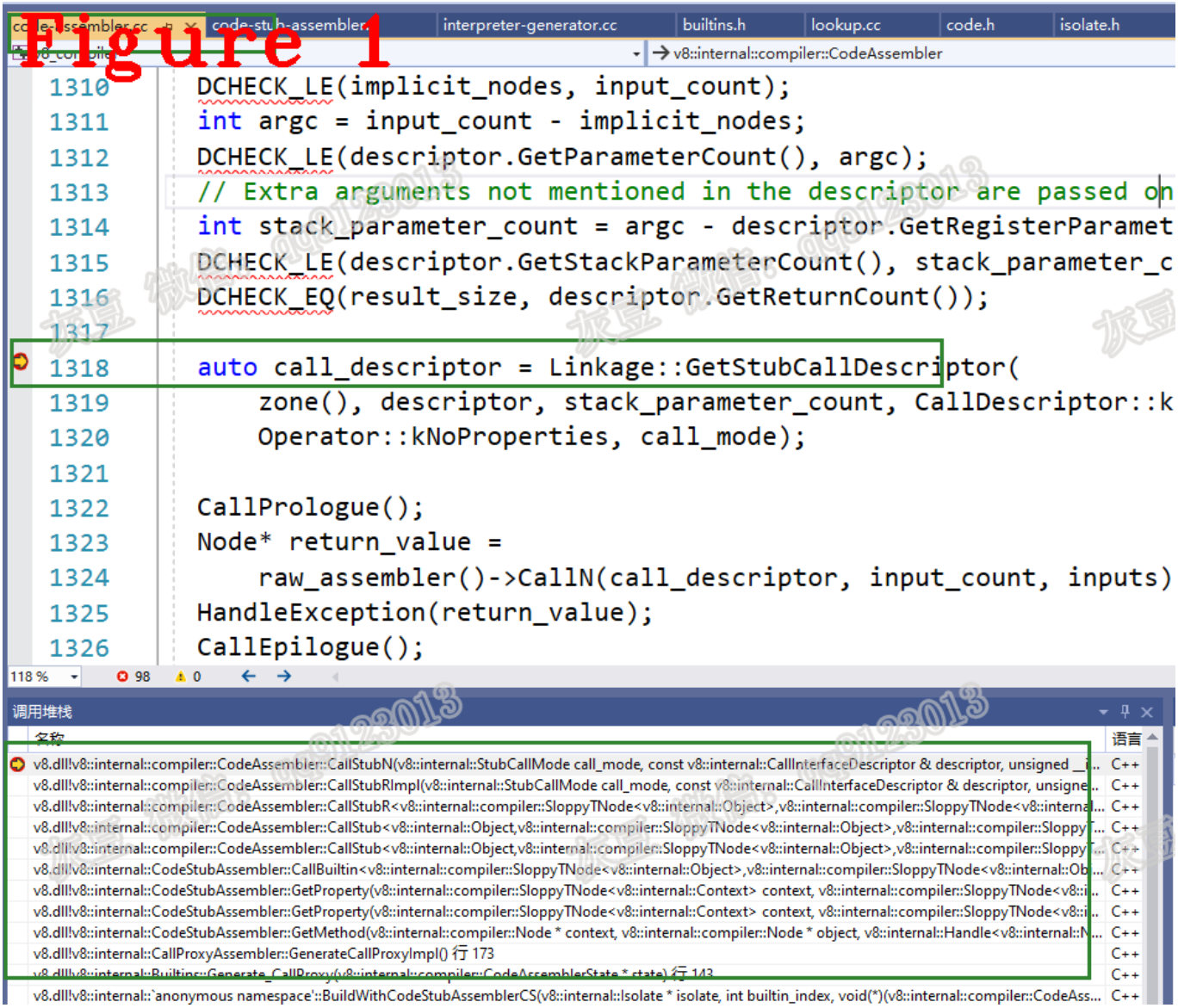
上述代码第7-10行将所有参数添加到数组inputs中，inputs内容依次为：Builtin的入口地址（code类型）、object、name、context。进入第12行代码，CallStubN()源码如下：

```
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.  }
```

上述第7行代码call_descriptor的返回值类型如下：

```
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())
```

上述信息为调用Builtin做准备工作。CallStubN()中第11行代码：完成Builtin的调用，第13行代码：异常处理。图1给出了CodeAssembler()的调用堆栈，此时正在建立Builtin，Builtin的建立发生在Isolate初始化阶段。



技术总结

(1) Builtin名字与Builtin的入口之间存在一一对应的关系，这种关系由isoate->isolate_data_->builtins_表示，builtins_是在Isolate初始化过程中创建的`Address`数组；

(2) inputs数组除了包括参数之外还有`target`和`context`；

(3) Builtin的参数、返回值等信息的详细说明可在`BUILTIN_LIST`宏中查看。

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 知乎：https://www.zhihu.com/people/v8blink

本文由灰豆原创发布

转载声明，注明出处： https://www.anquanke.com/post/id/260901 安全客 - 有思想的安全新媒体