《Chrome V8源码》27.神秘又简单的dispatch_table_

1 摘要

本篇文章是Builtin专题的第三篇，讲解Bytecode的执行、数据结构以及Dispatch。dispatch_table_是连接Bytecode之间的纽带，它记录了每条Bytecode handler的地址，Ignition通过dispatch_table_查找并执行相应的Bytecode。本文内容组织方法：Bytecode的执行和数据结构（章节2）；Bytecode的调度（章节3）。

2 Bytecode的执行

在V8中，负责执行Bytecode的解释器是Ignition，Ignition执行Bytecode时要做很多复杂的准备工作，这些"准备工作"后续文章讲解，我们重点说明Bytecode的执行。
Bytecode以JavaScript函数为粒度生成并存储在Bytecode array中，即Bytecode array是存储Bytecode的数组，源码如下：

```
class BytecodeArray : public FixedArrayBase {
 public:
  static constexpr int SizeFor(int length) {
    return OBJECT_POINTER_ALIGN(kHeaderSize + length);
  }
  inline byte get(int index) const;
  inline void set(int index, byte value);
  inline Address GetFirstBytecodeAddress();
//省略...................
 };
```

我们仅说明与本文有关的两点内容：
（1） 第3行代码`SizeFor(int length)`计算Bytecode array的长度，参数`length`的值是编译JavaScript函数后得到的Bytecode的数量，length+Bytecode array需要的空间等于Bytecode array的长度。在创建Bytecode array时，使用`SizeFor(int length)`计算申请内存的长度；
（2） 第8行代码`GetFirstBytecodeAddress()`获取Bytecode的首地址。把Parser生成的Bytecode拷贝到Bytecode array时会用到该函数。
在`Factory::NewBytecodeArray()`中，使用`SizeFor(int length)`的返回值申请内存，用`CopyBytes()`把Bytecode拷贝到首地址中。下面是一段Bytecode源码:

```
1.  a7                StackCheck
2.  12 00             LdaConstant [0]
3.  15 01 00          StaGlobal [1], [0]
4.  13 01 02          LdaGlobal [1], [2]
5.  26 f9             Star r2
6.  29 f9 02          LdaNamedPropertyNoFeedback r2, [2]
7.  26 fa             Star r1
8.  0c 05             LdaSmi [5]
9.  Constant pool (size = 6)
10.  0000005EAE403019: [FixedArray] in OldSpace
11.   - map: 0x03d0be000169 <Map>
12.   - length: 6
13.            0: 0x005eae402f59 <String[#22]: ignoreCase here we go!>
14.            1: 0x038ee90c3cc1 <String[#1]: a>
15.            2: 0x01b92bc2bde1 <String[#9]: substring>
16.            3: 0x038ee90c3fa9 <String[#1]: b>
17.            4: 0x01b92bc33839 <String[#7]: console>
18.            5: 0x01b92bc32e79 <String[#3]: log>
```

第2行代码`12 00 LdaConstant [0]`：`12`是LdaConstant的编号，这个编号也是`LdaConstant`的枚举值，即`Bytecode[0x12]=kLdaConstant`，源码如下：

```
enum class Bytecode : uint8_t {

        kWide, kExtraWide, kDebugBreakWide, kDebugBreakExtraWide, kDebugBreak0,
kDebugBreak1, kDebugBreak2, kDebugBreak3, kDebugBreak4, kDebugBreak5,
kDebugBreak6, kLdaZero, kLdaSmi, kLdaUndefined, kLdaNull, kLdaTheHole, kLdaTrue,
kLdaFalse, kLdaConstant, kLdaGlobal, kLdaGlobalInsideTypeof, kStaGlobal,
kPushContext, kPopContext, kLdaContextSlot, kLdaImmutableContextSlot,
kLdaCurrentContextSlot, kLdaImmutableCurrentContextSlot, kStaContextSlot,
kStaCurrentContextSlot, kLdaLookupSlot, kLdaLookupContextSlot,
kLdaLookupGlobalSlot, kLdaLookupSlotInsideTypeof,
kLdaLookupContextSlotInsideTypeof, kLdaLookupGlobalSlotInsideTypeof,
kStaLookupSlot, kLdar, kStar, kMov, kLdaNamedProperty,
kLdaNamedPropertyNoFeedback, kLdaKeyedProperty, kLdaModuleVariable,
kStaModuleVariable, kStaNamedProperty, kStaNamedPropertyNoFeedback,
kStaNamedOwnProperty, kStaKeyedProperty, kStaInArrayLiteral,
kStaDataPropertyInLiteral, kCollectTypeProfile, kAdd, kSub, kMul, kDiv, kMod,
kExp, kBitwiseOr, kBitwiseXor, kBitwiseAnd, kShiftLeft, kShiftRight,
kShiftRightLogical, kAddSmi, kSubSmi, kMulSmi, kDivSmi, kModSmi, kExpSmi,
kBitwiseOrSmi, kBitwiseXorSmi, kBitwiseAndSmi, kShiftLeftSmi, kShiftRightSmi,
kShiftRightLogicalSmi, kInc, kDec, kNegate, kBitwiseNot, kToBooleanLogicalNot,
kLogicalNot, kTypeOf, kDeletePropertyStrict, //省略..................
}
```

V8规定：fb代表寄存器R0，fa代表寄存器R1，以此类推。在`29 f9 02 LdaNamedPropertyNoFeedback r2, [2]`中，`f9`代表寄存器R2，`02`代表常量池[2]。执行`LdaNamedPropertyNoFeedback`时，Ignition通过Isolate获取dispatch_table的base address，再通过`base address+0x29`得到`LdaNamedPropertyNoFeedback`的handler，源码如下：

```
// Calls the GetProperty builtin for <object> and the key in the accumulator.
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();
}
```

3 Dispatch

`Dispatch_table`是指针数组，Bytecode的枚举值代表它在数组中的位置，该位置存储了对应的Bytecode handler的地址。`Dispatch_table`的初始化如下：

```
void Interpreter::Initialize() {
  Builtins* builtins = isolate_->builtins();
  // Set the interpreter entry trampoline entry point now that builtins are
  // initialized.
  Handle<Code> code = BUILTIN_CODE(isolate_, InterpreterEntryTrampoline);
  DCHECK(builtins->is_initialized());
  DCHECK(code->is_off_heap_trampoline() ||
         isolate_->heap()->IsImmovable(*code));
  interpreter_entry_trampoline_instruction_start_ = code->InstructionStart();
  // Initialize the dispatch table.
  Code illegal = builtins->builtin(Builtins::kIllegalHandler);
  int builtin_id = Builtins::kFirstBytecodeHandler;
  ForEachBytecode([=, &builtin_id](Bytecode bytecode,
                                   OperandScale operand_scale) {
    Code handler = illegal;
    if (Bytecodes::BytecodeHasHandler(bytecode, operand_scale)) {
#ifdef DEBUG
      std::string builtin_name(Builtins::name(builtin_id));
      std::string expected_name =
          Bytecodes::ToString(bytecode, operand_scale, "") + "Handler";
21.       DCHECK_EQ(expected_name, builtin_name);
22. #endif
23.       handler = builtins->builtin(builtin_id++);
24.     }
25.     SetBytecodeHandler(bytecode, operand_scale, handler);
26.   });
27.   DCHECK(builtin_id == Builtins::builtin_count);
28.   DCHECK(IsDispatchTableInitialized());
29. }
```

上述13-26行代码是匿名函数，其中25行代码初始化`Dispatch_table`，源码如下：

```
1.  void Interpreter::SetBytecodeHandler(Bytecode bytecode,
2.                                       OperandScale operand_scale, Code handler)
{
3.    DCHECK(handler.kind() == Code::BYTECODE_HANDLER);
4.    size_t index = GetDispatchTableIndex(bytecode, operand_scale);
5.    dispatch_table_[index] = handler.InstructionStart();
6.  }
7.  //.........分隔线............................................
8.  size_t Interpreter::GetDispatchTableIndex(Bytecode bytecode,
9.                                            OperandScale operand_scale) {
10.   static const size_t kEntriesPerOperandScale = 1u << kBitsPerByte;
11.   size_t index = static_cast<size_t>(bytecode);
12.   return index + BytecodeOperands::OperandScaleAsIndex(operand_scale) *
13.                      kEntriesPerOperandScale;
14. }
```

上述第5行代码`dispatch_table_`就是我们念念已久的存储dispatch table的成员变量；第4行代码`GetDispatchTableIndex()`计算Bytecode handler在dispatch_table中的位置，这个位置与`enum class Bytecode`是相同的。图1给出了`SetBytecodeHandler`的调用堆栈。

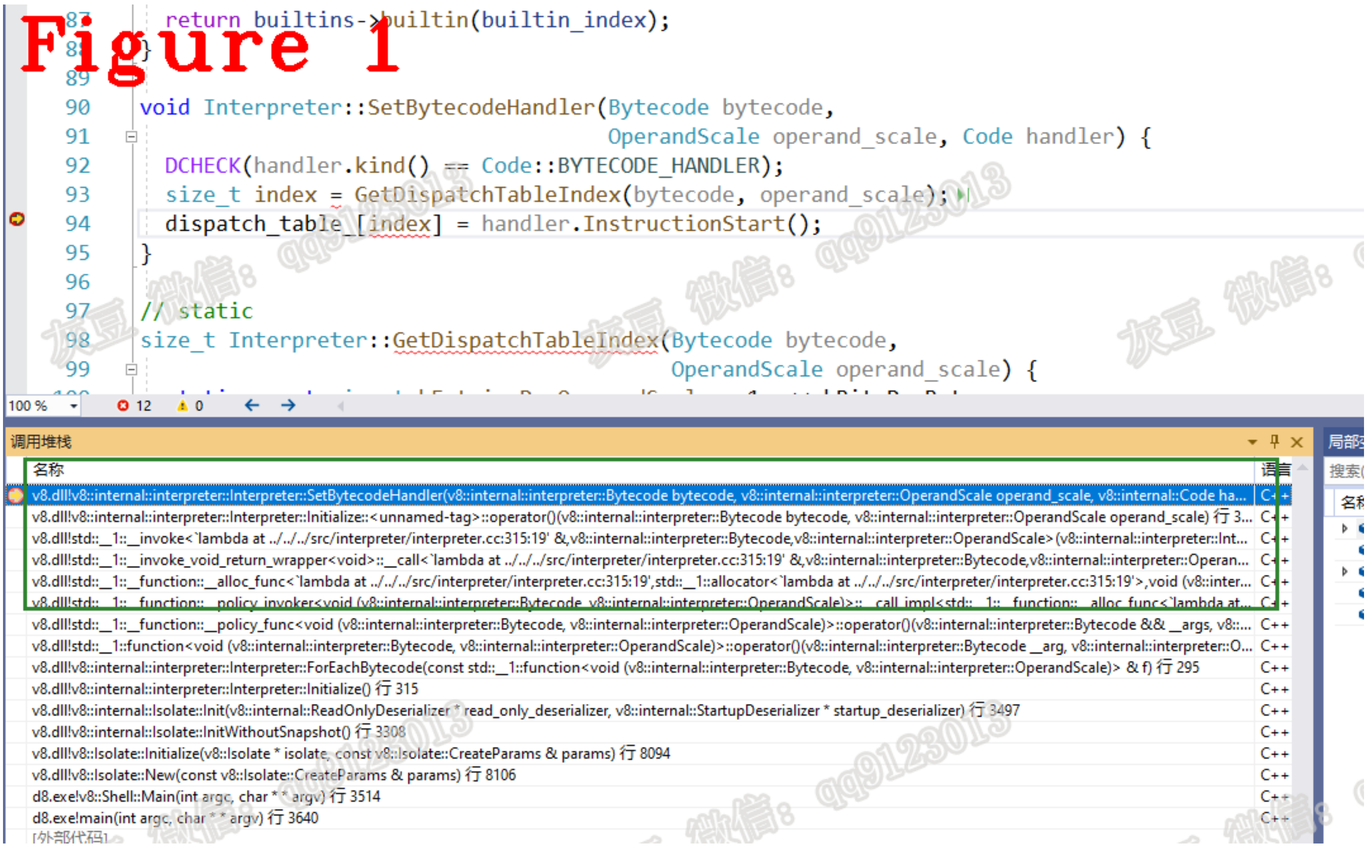

Figure 1

```
    return builtins->builtin(builtin_index);
}

void Interpreter::SetBytecodeHandler(Bytecode bytecode,
                                     OperandScale operand_scale, Code handler) {
  DCHECK(handler.kind() == Code::BYTECODE_HANDLER);
  size_t index = GetDispatchTableIndex(bytecode, operand_scale);
  dispatch_table_[index] = handler.InstructionStart();
}

// static
size_t Interpreter::GetDispatchTableIndex(Bytecode bytecode,
                                          OperandScale operand_scale) {
```

```
名称                                                                                                                                 语言
v8.dll!v8::internal::interpreter::Interpreter::SetBytecodeHandler(v8::internal::interpreter::Bytecode bytecode, v8::internal::interpreter::OperandScale operand_scale, v8::internal::Code ha...   C++
v8.dll!v8::internal::interpreter::Interpreter::Initialize::<unnamed-tag>::operator()(v8::internal::interpreter::Bytecode bytecode, v8::internal::interpreter::OperandScale operand_scale) 行 3...   C++
v8.dll!std::__1::__invoke<`lambda at ../../../src/interpreter/interpreter.cc:315:19' &,v8::internal::interpreter::Bytecode,v8::internal::interpreter::OperandScale>(v8::internal::interpreter::Int...   C++
v8.dll!std::__1::__invoke_void_return_wrapper<void>::__call<`lambda at ../../../src/interpreter/interpreter.cc:315:19' &,v8::internal::interpreter::Bytecode,v8::internal::interpreter::Operan...   C++
v8.dll!std::__1::__function::__alloc_func<`lambda at ../../../src/interpreter/interpreter.cc:315:19',std::__1::allocator<`lambda at ../../../src/interpreter/interpreter.cc:315:19'>,void (v8::inter...   C++
v8.dll!std::__1::__function::__policy_invoker<void (v8::internal::interpreter::Bytecode, v8::internal::interpreter::OperandScale)>::__call_impl<std::__1::__function::__alloc_func<`lambda at...   C++
v8.dll!std::__1::__function::__policy_func<void (v8::internal::interpreter::Bytecode, v8::internal::interpreter::OperandScale)>::operator()(v8::internal::interpreter::Bytecode && __args, v8::...   C++
v8.dll!std::__1::function<void (v8::internal::interpreter::Bytecode, v8::internal::interpreter::OperandScale)>::operator()(v8::internal::interpreter::Bytecode __arg, v8::internal::interpreter::O...   C++
v8.dll!v8::internal::interpreter::Interpreter::ForEachBytecode(const std::__1::function<void (v8::internal::interpreter::Bytecode, v8::internal::interpreter::OperandScale)> & f) 行 295   C++
v8.dll!v8::internal::interpreter::Interpreter::Initialize() 行 315   C++
v8.dll!v8::internal::Isolate::Init(v8::internal::ReadOnlyDeserializer * read_only_deserializer, v8::internal::StartupDeserializer * startup_deserializer) 行 3497   C++
v8.dll!v8::internal::Isolate::InitWithoutSnapshot() 行 3308   C++
v8.dll!v8::Isolate::Initialize(v8::Isolate * isolate, const v8::Isolate::CreateParams & params) 行 8094   C++
v8.dll!v8::Isolate::New(const v8::Isolate::CreateParams & params) 行 8106   C++
d8.exe!v8::Shell::Main(int argc, char * * argv) 行 3514   C++
d8.exe!main(int argc, char * * argv) 行 3640   C++
[外部代码]
```

Interpreter的源码如下：

```
1.  class Interpreter {
2.      //............省略.................
3.   private:
4.    // Get dispatch table index of bytecode.
5.    static size_t GetDispatchTableIndex(Bytecode bytecode,
6.                                        OperandScale operand_scale);
7.    static const int kNumberOfWideVariants =
BytecodeOperands::kOperandScaleCount;
8.    static const int kDispatchTableSize = kNumberOfWideVariants * (kMaxUInt8 +
1);
9.    static const int kNumberOfBytecodes = static_cast<int>(Bytecode::kLast) + 1;
10.    Isolate* isolate_;
11.    Address dispatch_table_[kDispatchTableSize];
12.    std::unique_ptr<uintptr_t[]> bytecode_dispatch_counters_table_;
13.    Address interpreter_entry_trampoline_instruction_start_;
14.    DISALLOW_COPY_AND_ASSIGN(Interpreter);
15.  };
```

上述第11行代码dispatch_table_是Interpreter的成员变量。Interpreter是Isolate的成员变量，源码如下：

```
1.  class Isolate final : private HiddenFactory {
2.  //省略...............
3.    const AstStringConstants* ast_string_constants_ = nullptr;
4.    interpreter::Interpreter* interpreter_ = nullptr;
5.    compiler::PerIsolateCompilerCache* compiler_cache_ = nullptr;
```

```
  Zone* compiler_zone_ = nullptr;
  CompilerDispatcher* compiler_dispatcher_ = nullptr;
  friend class heap::HeapTester;
  friend class TestSerializer;
   DISALLOW_COPY_AND_ASSIGN(Isolate);
 };
```

通过上述代码可以看出：`Isolate->interpreter_->dispatch_table_`获取`dispatch_table_`。 下面是在Bytecode handler中调用的`Dispatch()`的源码：

```
void InterpreterAssembler::Dispatch() {
  Comment("========= Dispatch");
  DCHECK_IMPLIES(Bytecodes::MakesCallAlongCriticalPath(bytecode_), made_call_);
  TNode<IntPtrT> target_offset = Advance();
  TNode<WordT> target_bytecode = LoadBytecode(target_offset);
  if (Bytecodes::IsStarLookahead(bytecode_, operand_scale_)) {
    target_bytecode = StarDispatchLookahead(target_bytecode);
  }
  DispatchToBytecode(target_bytecode, BytecodeOffset());
  }
  //......分隔线..................................
  void InterpreterAssembler::DispatchToBytecode(
      TNode<WordT> target_bytecode, TNode<IntPtrT> new_bytecode_offset) {
    if (FLAG_trace_ignition_dispatches) {
      TraceBytecodeDispatch(target_bytecode);
    }
    TNode<RawPtrT> target_code_entry = Load<RawPtrT>(
        DispatchTablePointer(), TimesSystemPointerSize(target_bytecode));
    DispatchToBytecodeHandlerEntry(target_code_entry, new_bytecode_offset);
  }
  //..........分隔线..................................
  void InterpreterAssembler::DispatchToBytecodeHandlerEntry(
      TNode<RawPtrT> handler_entry, TNode<IntPtrT> bytecode_offset) {
    // Propagate speculation poisoning.
    TNode<RawPtrT> poisoned_handler_entry =
        UncheckedCast<RawPtrT>(WordPoisonOnSpeculation(handler_entry));
    TailCallBytecodeDispatch(InterpreterDispatchDescriptor{},
                             poisoned_handler_entry, GetAccumulatorUnchecked(),
                             bytecode_offset, BytecodeArrayTaggedPointer(),
                             DispatchTablePointer());
  }
   //..........分隔线..................................
 void CodeAssembler::TailCallBytecodeDispatch(
     const CallInterfaceDescriptor& descriptor, TNode<RawPtrT> target,
     TArgs... args) {
   DCHECK_EQ(descriptor.GetParameterCount(), sizeof...(args));
   auto call_descriptor = Linkage::GetBytecodeDispatchCallDescriptor(
       zone(), descriptor, descriptor.GetStackParameterCount());
   Node* nodes[] = {target, args...};
40.    CHECK_EQ(descriptor.GetParameterCount() + 1, arraysize(nodes));
41.    raw_assembler()->TailCallN(call_descriptor, arraysize(nodes), nodes);
42.  }
```

上述三个方法共同实现Bytecode的dispatch。第5行代码计算target_bytecode；第17行代码计算target_bytecode_entry；第27行代码开始跳转；第34行代码创建`call discriptor`；第41行代码生成Node节点，并把该节点添加到当前基本块的尾部，至此跳转完成。`TailCallN()`的详细讲解参见第十一篇文章。图2给出了`Dispatch()`的调用堆栈。

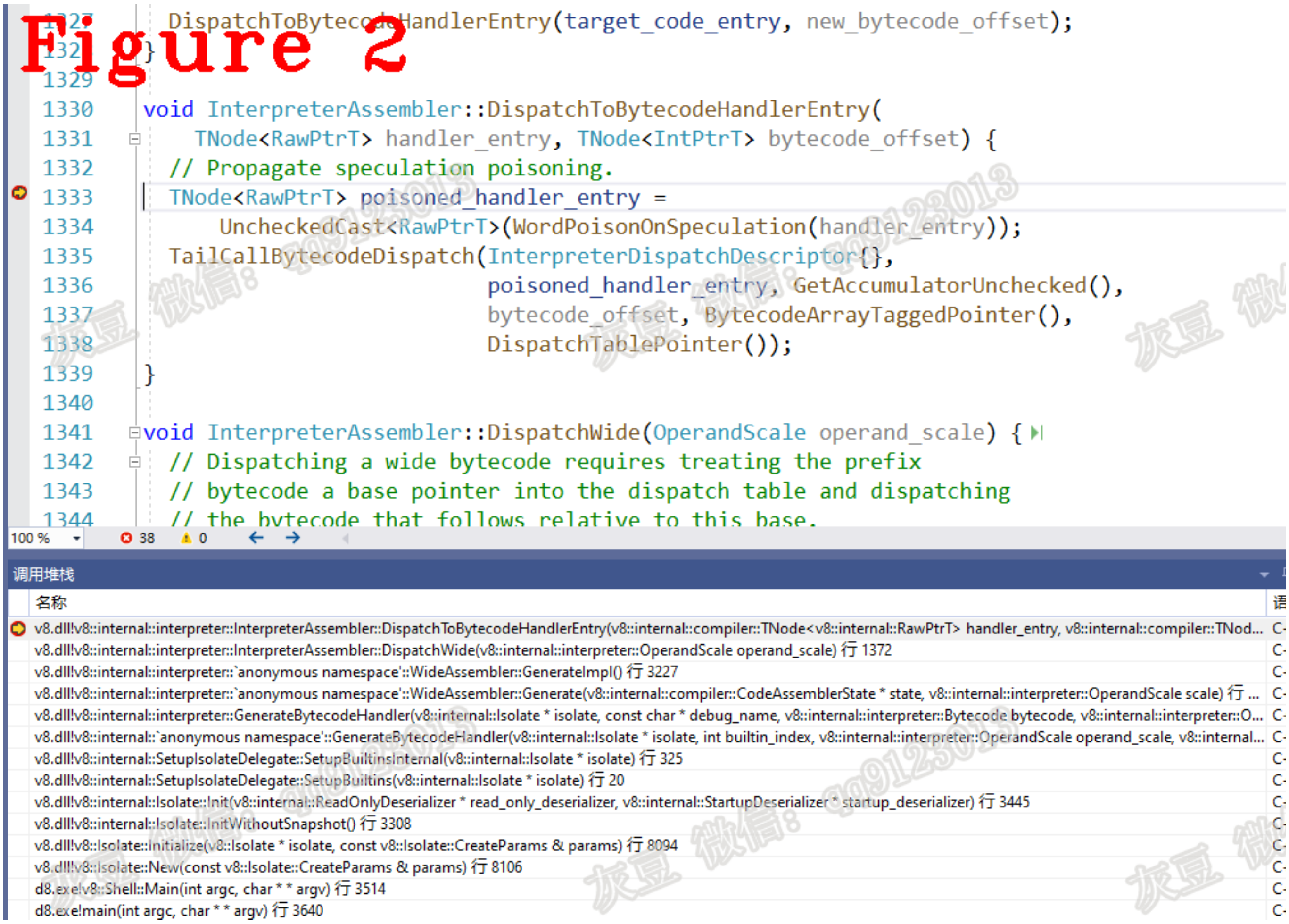

Figure 2

```
  DispatchToBytecodeHandlerEntry(target_code_entry, new_bytecode_offset);
}

void InterpreterAssembler::DispatchToBytecodeHandlerEntry(
    TNode<RawPtrT> handler_entry, TNode<IntPtrT> bytecode_offset) {
  // Propagate speculation poisoning.
  TNode<RawPtrT> poisoned_handler_entry =
      UncheckedCast<RawPtrT>(WordPoisonOnSpeculation(handler_entry));
  TailCallBytecodeDispatch(InterpreterDispatchDescriptor{},
                           poisoned_handler_entry, GetAccumulatorUnchecked(),
                           bytecode_offset, BytecodeArrayTaggedPointer(),
                           DispatchTablePointer());
}

void InterpreterAssembler::DispatchWide(OperandScale operand_scale) {
  // Dispatching a wide bytecode requires treating the prefix
  // bytecode a base pointer into the dispatch table and dispatching
  // the bytecode that follows relative to this base.
```

```
v8.dll!v8::internal::interpreter::InterpreterAssembler::DispatchToBytecodeHandlerEntry(v8::internal::compiler::TNode<v8::internal::RawPtrT> handler_entry, v8::internal::compiler::TNod...  C-
v8.dll!v8::internal::interpreter::InterpreterAssembler::DispatchWide(v8::internal::interpreter::OperandScale operand_scale) 行 1372  C-
v8.dll!v8::internal::interpreter::`anonymous namespace'::WideAssembler::GenerateImpl() 行 3227  C-
v8.dll!v8::internal::interpreter::`anonymous namespace'::WideAssembler::Generate(v8::internal::compiler::CodeAssemblerState * state, v8::internal::interpreter::OperandScale scale) 行 ...  C-
v8.dll!v8::internal::interpreter::GenerateBytecodeHandler(v8::internal::Isolate * isolate, const char * debug_name, v8::internal::interpreter::Bytecode bytecode, v8::internal::interpreter::O...  C-
v8.dll!v8::internal::`anonymous namespace'::GenerateBytecodeHandler(v8::internal::Isolate * isolate, int builtin_index, v8::internal::interpreter::OperandScale operand_scale, v8::internal...  C-
v8.dll!v8::internal::SetupIsolateDelegate::SetupBuiltinsInternal(v8::internal::Isolate * isolate) 行 325  C-
v8.dll!v8::internal::SetupIsolateDelegate::SetupBuiltins(v8::internal::Isolate * isolate) 行 20  C-
v8.dll!v8::internal::Isolate::Init(v8::internal::ReadOnlyDeserializer * read_only_deserializer, v8::internal::StartupDeserializer * startup_deserializer) 行 3445  C-
v8.dll!v8::internal::Isolate::InitWithoutSnapshot() 行 3308  C-
v8.dll!v8::Isolate::Initialize(v8::Isolate * isolate, const v8::Isolate::CreateParams & params) 行 8094  C-
v8.dll!v8::Isolate::New(const v8::Isolate::CreateParams & params) 行 8106  C-
d8.exe!v8::Shell::Main(int argc, char * * argv) 行 3514  C-
d8.exe!main(int argc, char * * argv) 行 3640  C-
```

技术总结

（1） Bytecode的编号是Bytecode handler在数组`dispatch_table_`中的下标；

（2） dispatch_table_的初始化在Isolate启动时完成；

（3） 使用固定的物理寄存器保存`dispatch_table_`的优点是：避免不必要的入栈和出栈，简化Bytecode的设计，提高了Dispatch的效率；

提示：我调试V8时，`dispatch_table_`始终保存在物理寄存器R15中，调试方法参见第18篇文章。

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
转载出处： https://www.anquanke.com/post/id/260182
安全客 - 有思想的安全新媒体