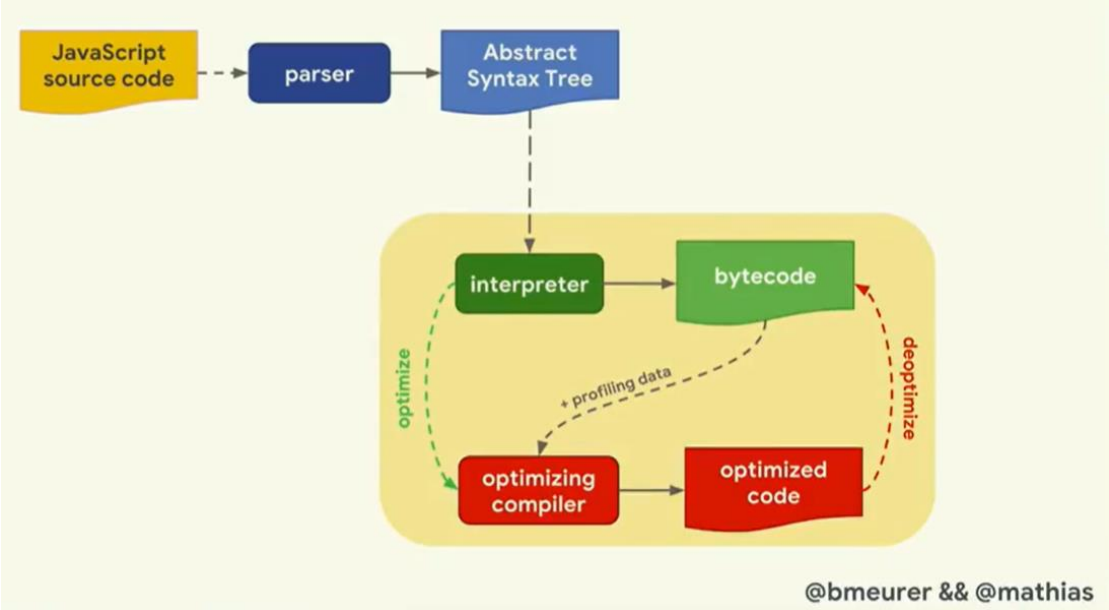
V8 是一个 javascript 引擎，用于编译并执行 javascript 代码。V8 的架构经过很多次变化，目前整个工作过程可以在这幅图上体现出来：




https://www.youtube.com/watch?v=5nmpokoRaZI

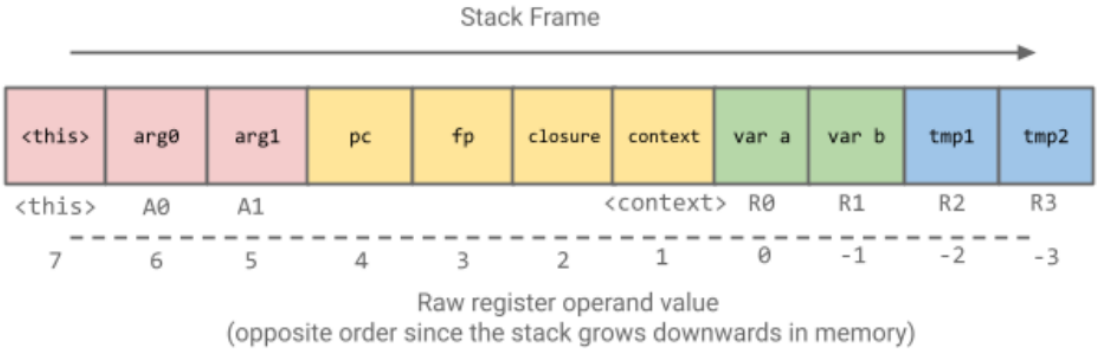
javascript 代码经过 Parser（语法分析器）生成 AST（抽象语法树），然后 Ignition 解释器根据 AST 生成字节码并且执行它，在执行的过程中，Ignition 解释器会注意到一些经常被执行的函数，并且收集这些函数的相关信息。生成的字节码，以及收集到的信息，都会被喂到 turbofan 优化编译器，turbofan 会根据它们生成优化后的机器码。相比于解释执行字节码，执行机器码的速度会更快。但是这些机器码是基于一些假设生成的，如果在执行的过程中，遇到了违反假设的情况，此时只能返回到解释执行字节码的阶段，这就是反优化。

本部分主要描述 Ignition 解释器是如何工作的。

Ignition 解释器的一个任务是生成字节码。这个工作主要是由 BytecodeGenerator 完成的，它会遍历 AST，为每个 AST 节点生成字节码。一个函数的字节码会被存储到这个函数的 SharedFunctionInfo 对象里，顾名思义，这个对象描述了一个 javascript 函数的相关信息，这些信息可以被多个函数的实例共享。

Ignition 解释器是基于寄存器的，这里的寄存器并不是真实的机器寄存器，而是一个 register file 内特定的 slot，register file 是作为函数的 stack frame 的一部分被分配的。特别的，为了减少字节码流的大小，Ignition 解释器使用了一个特殊的寄存器 Accumulator，它并不是 stack frame 上的 register file 的一部分，而是 Ignition 解释器维护的一个机器寄存器，某些字节码会使用 Accumulator 寄存器作为默认寄存器，这样就可以省略掉一个操作数。

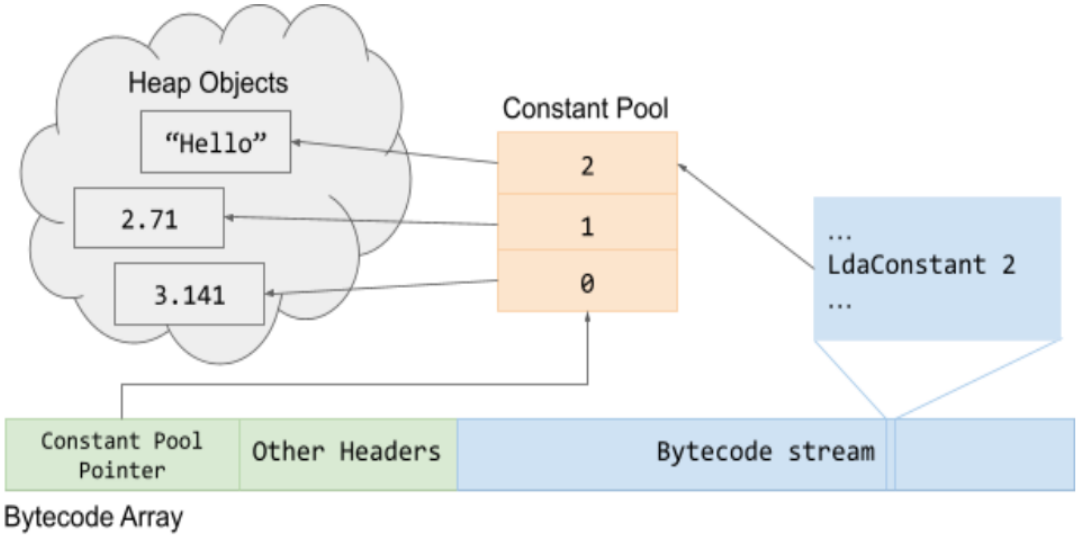
BytecodeGenerator 在生成字节码的过程中会进行寄存器的分配工作。每个局部变量在遍历 AST 的初始阶段被映射到 register file 内的一个 slot；另外，在 expression evaluation（表达式计算）时需要 temporary，所以 BytecodeGenerator 也会为它们分配寄存器。但是这些 temporary 的生命周期不会横跨多个 statement（语句），所以 BytecodeGenerator 结束对某个 statement 的访问时，它分配的所有 temporary 寄存器都会被释放。BytecodeGenerator 会注意所需的 temporary 寄存器的最大数量，从而在 register file 中留出足够的 slot。




https://docs.google.com/document/d/11T2CRex9hXxoJwbYqVQ32yIPMh0uouUZLdyrtmMoL44/edit#

图中展示了 Ignition 解释器的 stack frame 的布局，绿色的部分是局部变量对应的寄存器，蓝色的部分是 temporary 对应的寄存器。

生成的字节码会引用一些常量，所以在字节码生成阶段，Ignition 解释器还需要构建常量池。




https://docs.google.com/document/d/11T2CRex9hXxoJwbYqVQ32yIPMh0uouUZLdyrtmMoL44/edit#

常量池是用于存储堆对象和 Smis（小整数）的，每个 BytecodeArray 有自己的常量池，常量池是一个指针数组。

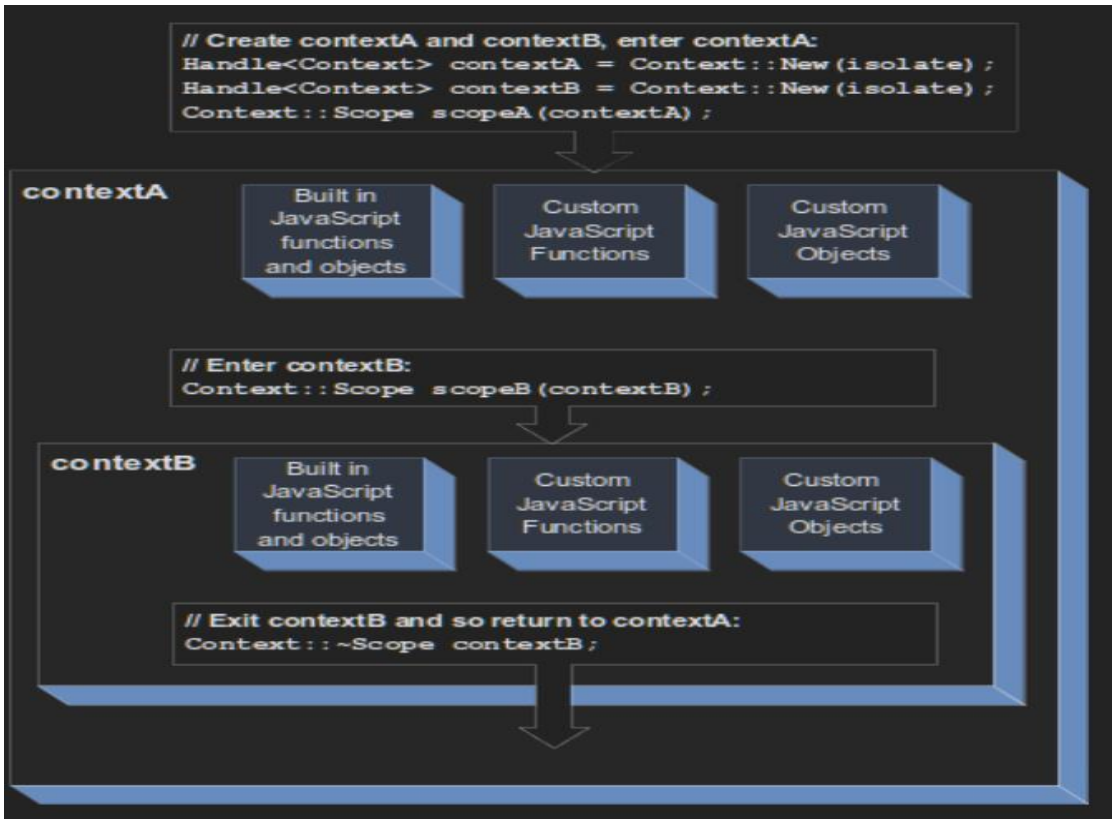
常量池的构建过程是这样的：当一个字节码需要加载一个常量时，ConstantArrayBuilder 会检查常量池里是否已经存在该常量，如果是，返回该常量的索引；如果否，将该常量加入常量池，并且返回其索引。

另外，因为 jump 字节码的特殊性，常量池还会存储 jump 字节码的偏移量。由于在 jump 字节码被生成时，它的 jump displacement（位移）是未知的，所以需要在字节码流中为其预留出一定的空间。当它的位移已知时，BytecodeArrayBuilder 会对 jump 字节码进行补足。当预留的空间足够容纳位移的值时，位移的值会被填入空间内；但是当预留的空间不足以容纳位移的值时，位移的值会被放入常量池里，然后 jump 字节码会更新为另一个字节码，并且其操作数会被更新为常量池中的对应到位移的一项。

在对控制流结构生成字节码时，由于 Ignition 解释器支持 label（标签）的概念，最后生成的字节码的控制逻辑和 javascript 代码是一致的。

除了生成字节码，Ignition 解释器还会追踪当前的上下文对象，在之前的 stack frame 里，存

在一个 context slot，其中存放着上下文对象。




https://v8.dev/docs/embed

上下文是一个执行环境，每个上下文有自己独立的 javascript 的内置函数和对象，这样就允许多个不同的 javascript 应用在同一个 V8 实例里运行，而不会相互干扰。当我们在一个上下文里时，可以进入另一个的上下文，也可以选择退出当前上下文，这样就形成了一个嵌套的结构，由于允许嵌套，我们得到了一个上下文链。

当一个上下文对象被创建时，BytecodeGenerator 会分配一个 ContextScope 对象来追踪嵌套的上下文链，这样 BytecodeGenerator 就可以展开嵌套的上下文链。当访问一个分配在某个内部上下文的变量时，Ignition 解释器可以直接访问，而不必沿着上下文链访问。

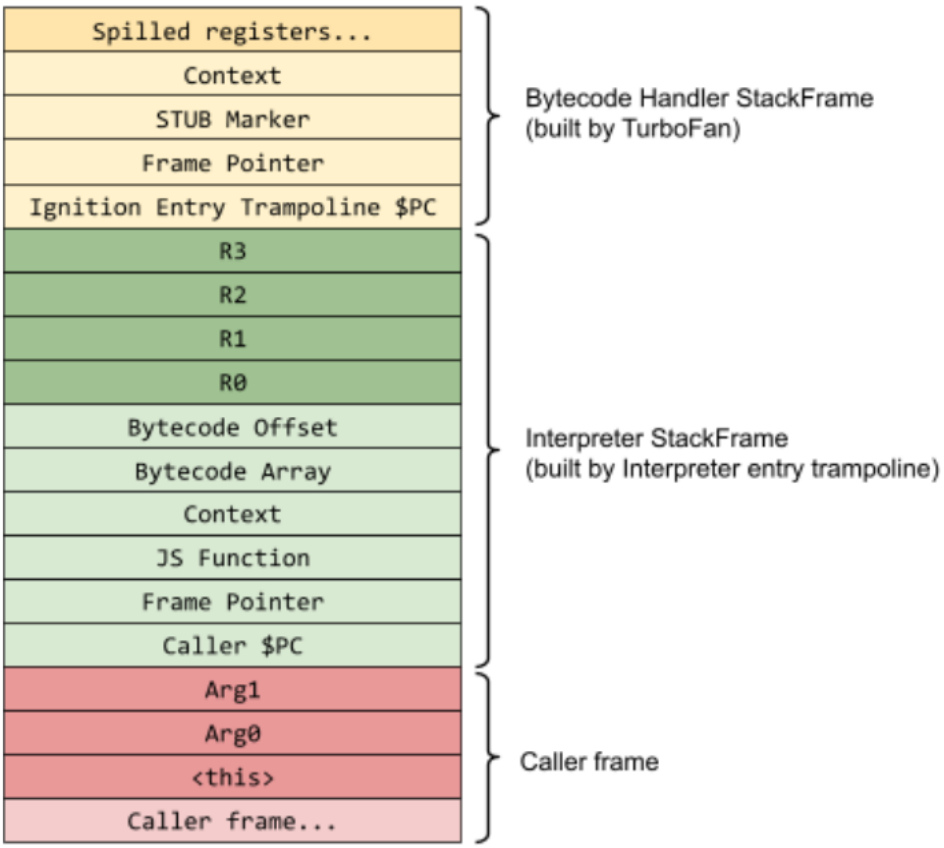
更具体的，当一个 ContextScope 对象被分配时，当前的上下文对象会被移到另一个寄存器里，然后新的上下文对象会被移到当前上下文寄存器里，也就是 context slot 里；类似的，当 ContextScope 对象离开了它对应的范围时，上一个上下文对象会被移回当前上下文寄存器里。这样，如果要访问分配在某个特定的上下文的变量，Ignition 解释器不必沿着上下文链寻找该变量对应的上下文，BytecodeGenerator 会找到该变量对应的 ContextScope，从而知道对应的上下文对象被分配在哪个寄存器里。由于当前上下文寄存器的存在，BytecodeGenerator 不需要记住当前的上下文对象在哪个寄存器里，它一定就在当前上下文寄存器里。

Ignition 解释器的另一个任务是解释执行字节码，因此可以把 Ignition 解释器看作是一个 bytecode handler 的集合，不同的字节码由不同的 bytecode handler 来处理。bytecode handler 由架构无关的汇编语言编写，由 turbofan 编译，生成对应特定架构的机器码，所以只需要编写一次。

和 bytecode handler 相关的一个概念是全局分派表，每个 V8 引擎的 isolated 的实例都有一个全局分派表，表中存储的是 code object 指针，每个指针指向一个 bytecode handler。

```
// Mov <src> <dst>
//
// Stores the value of register <src> to register <dst>.
IGNITION_HANDLER(Mov, InterpreterAssembler) {
  TNode<Object> src_value = LoadRegisterAtOperandIndex(0);
  StoreRegisterAtOperandIndex(src_value, 1);
  Dispatch();
}
```

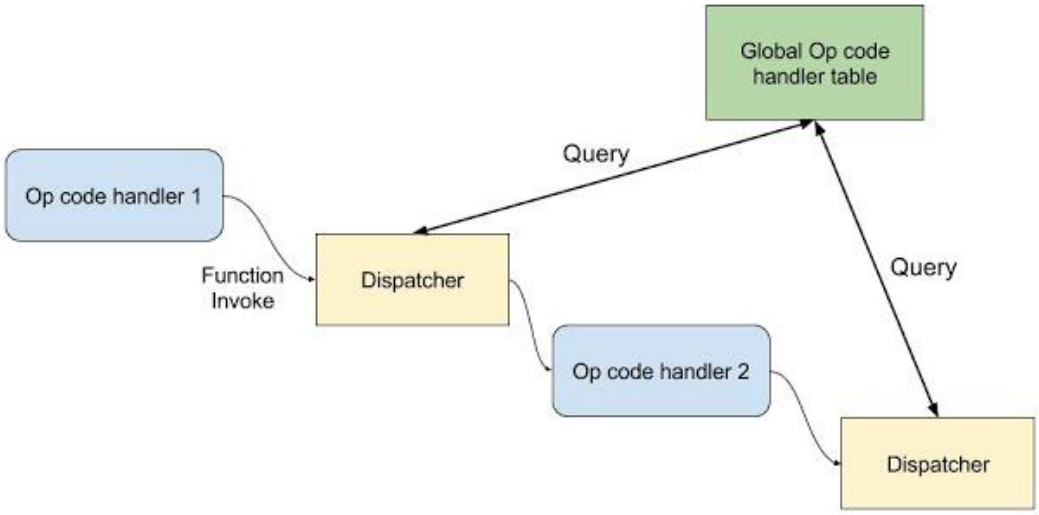
图中是一个对应到 Mov 这个字节码的 bytecode handler 的例子。首先从一个寄存器中加载值，然后把值存入另一个寄存器中，最后调用一个 Dispatch 函数。实际上，每个 bytecode handler 的末尾都会调用 Dispatch 函数。




https://docs.google.com/document/d/11T2CRex9hXxoJwbYqVQ32yIPMh0uouUZLdyrtmMoL44/edit#

当一个函数在运行时被调用时，由于函数的代码入口地址被设置在 InterpreterEntryTrampoline 这个内置的 stub，所以这个 stub 会在函数开始执行前做一些准备工作。它会为 Ignition 解释器建立一个合适的 stack frame，并且初始化一些固定的机器寄存器，分配 register file，然后分派一个 bytecode handler 来处理函数的第一个字节码。每个

bytecode handler 在最后会调用 Dispatch 函数来分派下一个字节码对应的 bytecode handler，所以整个执行过程是一个流式的过程，不会出现某个被调用者执行完毕后返回到调用者的情况 。




http://mshockwave.blogspot.com/2016/03/ignition-interpreter-in-v8-javascript.html
具体到代码，整个的调用链是这样的：Bytecode handler->Dispatch->DispatchToBytecodeWithOptionalStarLookahead->DispatchToBytecode->DispatchToBytecodeHandlerEntry->TailCallBytecodeDispatch->TailCallN->Next bytecode handler，在两个 bytecode handler 之间的部分可以看做是一个分派器。

```
// Dispatch to the bytecode.
void InterpreterAssembler::Dispatch() {
  Comment("========= Dispatch");
  DCHECK_IMPLIES(Bytecodes::MakesCallAlongCriticalPath(bytecode_), made_call_);
  // Updates and returns BytecodeOffset() advanced by the current bytecode's
  // size. Traces the exit of the current bytecode.
  TNode<IntPtrT> target_offset = Advance();
  // Load the bytecode at |bytecode_offset|.
  TNode<WordT> target_bytecode = LoadBytecode(target_offset);
  DispatchToBytecodeWithOptionalStarLookahead(target_bytecode);
}
```

Dispatch 函数首先把当前的字节码偏移量前进一个字节码长度，这样就得到了下一个要处理的字节码的偏移量，然后从这个偏移量加载对应的字节码，得到目标字节码，最后调用 DispatchToBytecodeWithOptionalStarLookahead 函数

```
// Dispatches to |target_bytecode| at BytecodeOffset(). Includes short-star
// lookahead if the current bytecode_ is likely followed by a short-star
// instruction.
void InterpreterAssembler::DispatchToBytecodeWithOptionalStarLookahead(
    TNode<WordT> target_bytecode) {

  if (
      // Returns true if the handler for |bytecode| should look ahead and inline a
      // dispatch to a Star bytecode.
      Bytecodes::IsStarLookahead(bytecode_, operand_scale_)) {
    StarDispatchLookahead(target_bytecode);
  }
  DispatchToBytecode(target_bytecode,
                     // Returns the offset from the BytecodeArrayPointer
                     // of the current bytecode.
                     BytecodeOffset());
}
```

DispatchToBytecodeWithOptionalStarLookahead 函数对一种特殊情况做了处理，它基于特定的目标字节码再往前看一步，如果目标字节码的后面跟随了一个 short star 指令，那么就需要在当前的字节码偏移量处为 short star 构建代码，short star 并不会由它对应的 bytecode handler 处理，而是把对应的 bytecode handler 的处理逻辑内联到此处。一般的 star 指令指的是把 Accumulator 寄存器中的值存储到某个寄存器中，如果目标寄存器位 r0，那么对应 short star 指令就是 star0，从而省略了 r0 这个操作数；如果不满足这种特殊情况，则不需要额外处理，直接用目标字节码和它对应的字节码偏移量作为参数调用 DispatchToBytecode 函数。

```
// Dispatch to |target_bytecode| at |new_bytecode_offset|.
// |target_bytecode| should be equivalent to loading from the offset.
void InterpreterAssembler::DispatchToBytecode(
    TNode<WordT> target_bytecode, TNode<IntPtrT> new_bytecode_offset) {
  if (FLAG_trace_ignition_dispatches) {
    TraceBytecodeDispatch(target_bytecode);
  }
  //Load a pointer to the target entry in the interpreter dispatch table
  //based on the target bytecode.
  TNode<RawPtrT> target_code_entry = Load<RawPtrT>(
      // Returns a pointer to first entry in the interpreter dispatch table.
      DispatchTablePointer(),
      TimesSystemPointerSize(target_bytecode));

  DispatchToBytecodeHandlerEntry(target_code_entry, new_bytecode_offset);
}
```

DispatchToBytecode 函数主要做的事情是从全局分派表里，根据字节码来加载表中对应的一项，实际得到的是一个指针，指向表中的一项。现在已知应该用哪个 bytecode handler 来处理目标字节码，接下来调用 DispatchToBytecodeHandlerEntry 函数。

```
// Dispatch to the bytecode handler with code entry point |handler_entry|.
void InterpreterAssembler::DispatchToBytecodeHandlerEntry(
    TNode<RawPtrT> handler_entry, TNode<IntPtrT> bytecode_offset) {
  // Propagate speculation poisoning.
  TNode<RawPtrT> poisoned_handler_entry =
      UncheckedCast<RawPtrT>(
          // Poison |value| on speculative paths.
          WordPoisonOnSpeculation(handler_entry));
  TailCallBytecodeDispatch(InterpreterDispatchDescriptor{},
                           poisoned_handler_entry, GetAccumulatorUnchecked(),
                           bytecode_offset,
                           // Returns a pointer to
                           // the current function's BytecodeArray object.
                           BytecodeArrayTaggedPointer(),
                           DispatchTablePointer());
}
```

DispatchToBytecodeHandlerEntry 函数首先对 bytecode handler 项做了一个关于分支预测的安全处理，处理结束以后，会调用 TailCallBytecodeDispatch 函数，此时把调用描述符，bytecode handler 项，Accumulator 寄存器，字节码偏移量等参数传过去。

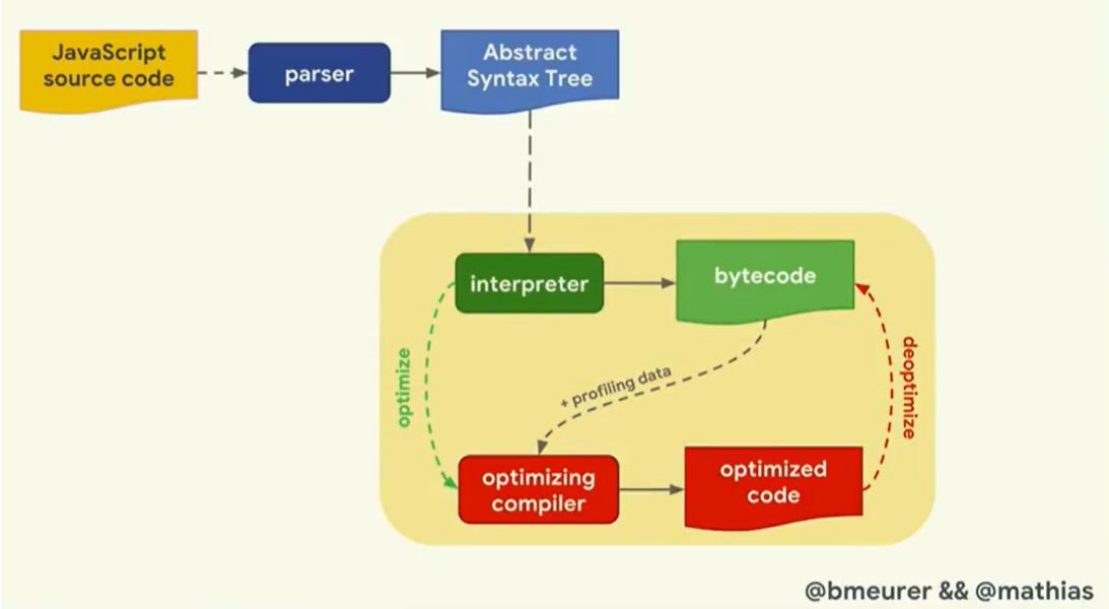
```
template <class... TArgs>
void CodeAssembler::TailCallBytecodeDispatch(
    const CallInterfaceDescriptor& descriptor, TNode<RawPtrT> target,
    TArgs... args) {
  DCHECK_EQ(descriptor.GetParameterCount(), sizeof...(args));
  auto call_descriptor = Linkage::GetBytecodeDispatchCallDescriptor(
      zone(), descriptor, descriptor.GetStackParameterCount());

  Node* nodes[] = {target, args...};
  CHECK_EQ(descriptor.GetParameterCount() + 1, arraysize(nodes));
  raw_assembler()->TailCallN(call_descriptor, arraysize(nodes), nodes);
}
```

TailCallBytecodeDispatch 函数把除了描述符之外的参数全部放到一个数组里面，然后把调用描述符和数组作为参数调用 TailCallN 函数。

```
// Tail call a given call descriptor and the given arguments.
// The call target is passed as part of the {inputs} array.
void RawMachineAssembler::TailCallN(CallDescriptor* call_descriptor,
                                    int input_count, Node* const* inputs) {
  // +1 is for target.
  DCHECK_EQ(input_count, call_descriptor->ParameterCount() + 1);
  Node* tail_call =
      //common(): return a pointer to CommonOperatorBuilder,
      // which builds common operators that can be used at any level of IR
      //TailCall: return an Operator
      MakeNode(common()->TailCall(call_descriptor), input_count, inputs);
  // BasicBlock building: add a tailcall at the end of current execution block {block}.
  schedule()->AddTailCall(CurrentBlock(), tail_call);
  current_block_ = nullptr;
}
```

最后在 TailCallN 函数中，首先根据调用描述符得到一个操作符，然后根据接收到的参数构建一个节点，最后把这个节点放到当前执行块的末尾。

这就是整个分派器做的事，当执行到这个末尾的节点的时候，实际上就是用对应的 bytecode handler 来处理下一个目标字节码，从而可以如流水一般不断地执行下去。




https://www.youtube.com/watch?v=5nmpokoRaZI

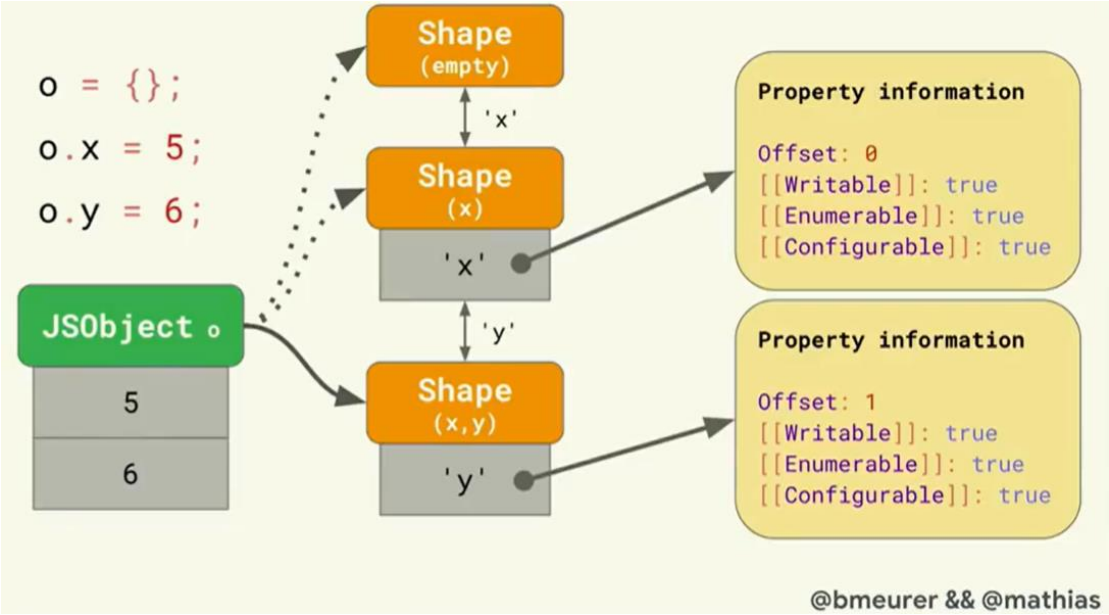
除了生成字节码和解释执行字节码，Ignition 解释器同时还有收集函数执行过程中的反馈信息的任务。比如，在程序运行的同时，Ignition 解释器会注意到对象的数据类型，并通过存储相关信息将其反映出来，以便后续的优化。这部分工作主要是由 inline cache 负责。

| IC Slot | IC Type | State |
| --- | --- | --- |
| ... | ... | ... |
| 10 | LOAD | MONO(M) |
| 11 | LOAD | UNINITIALIZED |

https://slides.com/ripsawridge/deck

更具体的，收集到的反馈信息是存储在 FeedbackVector 内的，FeedbackVector 可以被理解为一个带有头部信息的 slot 序列。Bytecode handler 会把 FeedbackVector 传给 inline cache stub，inline cache stub 可以更新 FeedbackVector 里的内容，这些反馈信息会被 turbofan 这个优化编译器利用，用来生成优化后的机器码。

采用了这种机制后的一个好处是可以减少访问对象属性的时间。




https://www.youtube.com/watch?v=5nmpokoRaZI

首先我们需要了解对象的属性信息是怎么组织的。为了节省存储空间，我们使用 map，或者说 hidden class，也可以说是 shape，来描述对象的结构，shape 不涉及具体的对象属性值，但是可以表示每个属性的相关性质，比如说某个属性的偏移量，是否可写等信息，这样就允许多个结构相同的对象共享同一个 map。采用了 map 机制后，一般的访问属性的过程是这样的：如果要访问一个对象 o 的属性 x，那么首先找到 map 中 x 的部分，然后找到 x 对应的属性信息，得到 x 的偏移量是 0，最后在 JSObject 的偏移量为 0 的地方找到 x 的值。

但是因为 FeedbackVector 存储了收集到的函数执行过程中的反馈信息，我们可以把上述访问对象属性的过程变得更加快捷。当字节码涉及对对象的属性的加载或者存储操作时，字节码对应的 bytecode handler 会调用 loadIC stub 或者 storeIC stub。在加载属性的情况下，loadIC stub 被调用，它会去查询 TypeFeedbackVector，找到对应的 slot，如果状态是

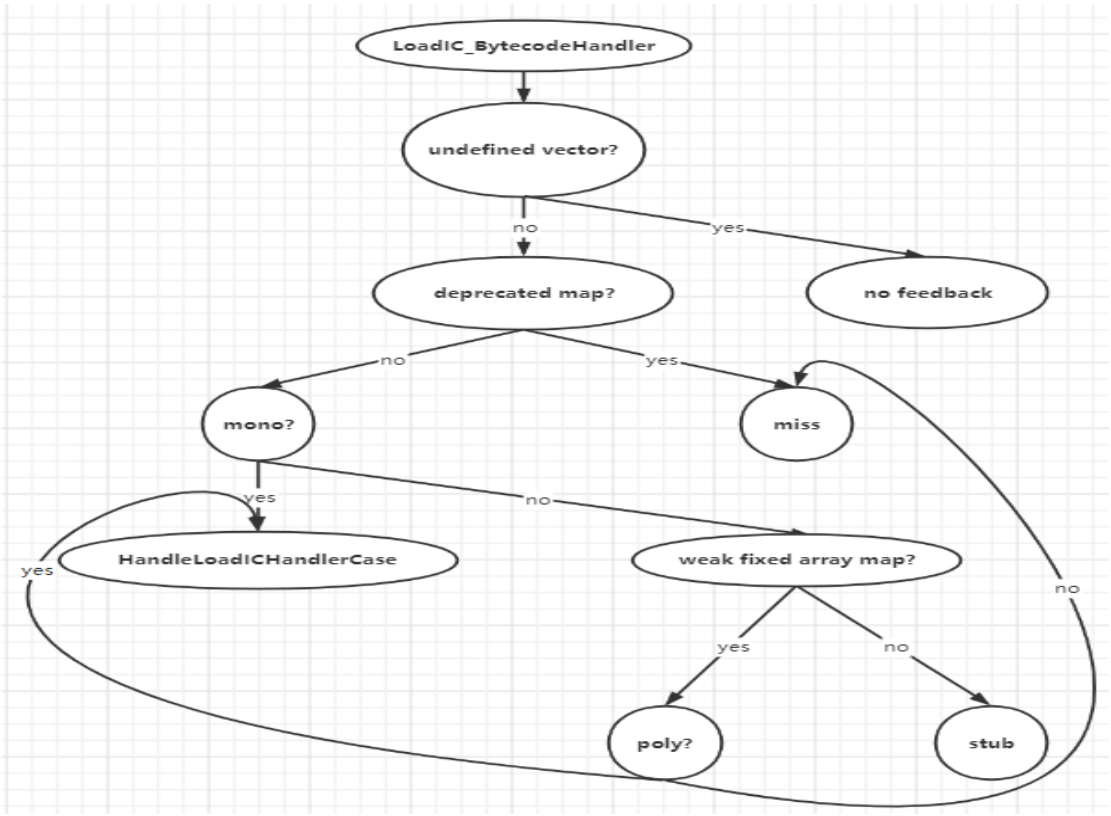
monomorphic，那么继续检查对象的 map，和 slot 中存储的 map 是否一致，如果一致的话，就可以使用 slot 中存储的偏移量来获取对象的属性值；如果不一致的话，就只能按一般的方法去查询对象的属性值，这样速度会更慢，而且还需要更新 FeedbackVector 里对应的 slot。每个 slot 中存储一个会被使用的 inline cache stub 以及它的状态。状态 monomorphic 指的是 inline cache stub 处理过的对象的 map 都是相同的。状态 polymorphic 指的是 inline cache stub 处理过少量 map 不同的对象。如果原来 inline cache stub 的状态是 monomorphic，一但遇到一个对象的 map 和之前不一样就需要把 monomorphic 更新成 polymorphic。如果 inline cache stub 处理非常多不同 map 的对象，那么 polymorphic 还会被更新成其他状态。实际上一个函数的 FeedbackVector 在生成 AST 阶段就已经被确定了要存储哪些 inline cache stub 的状态，或者说信息，只不过对应的状态值是未初始化的。

在解释执行字节码的过程中，如果某个 inline cache stub 被第一次执行，此时是没有办法利用 FeedbackVector 中的信息的，所以只能按一般的方法，首先找到对象的 map，然后查询属性的对应的信息，得到对应的偏移量后根据偏移量从 JSObject 中获取属性的值，经过这些步骤之后，就可以在 TypeFeedbackVector 里把状态值从未初始化的更新为 monomorphic 状态，并且更新 map，偏移量等信息。

当这个 inline cache stub 被第二次执行的时候，FeedbackVector 已经有了关于它的一些信息，此时可以检查是否满足一些条件，满足的话就可以直接使用 FeedbackVector 里存储的偏移量来从 JSObject 获取属性值，相当于省略掉了根据 map 中对应的属性查询其相关信息的步骤。

以上过程都是属于解释执行过程，我们还需要决定是否要生成优化代码。如果某个函数它经常被调用，而且在解释执行的过程中，FeedbackVector 已经包含了足够多的信息，那么我们就可以优化它，这部分就是 turbofan 的工作了。但实际上，turbofan 生成的机器码的逻辑和上述查询过程大致上是一样的，只不过当有条件不满足的时候就进行反优化。

具体到代码中，以加载对象属性为例，对应的 bytecode handler 的任务是把某一个对象的指定属性的值加载到 Accumulator 寄存器中。它首先获取当前函数的 FeedbackVector，然后获取属性名称，上下文，还有 FeedbackVector 中对应的 slot，接着把这些信息传递给 loadIC stub，最后把得到的属性值存到 Accumulator 寄存器里，最后调用 Dispatch 函数分派下一个 bytecode handler。




图中展示了 loadIC stub 的控制逻辑结构。首先它会判断收到的 FeedbackVector 有没有被定义，如果没有，就转到 no feedback 分支，这里会调用一个名为 kLoadIC_NoFeedback 的内置函数来处理没有 feedback 的情况。如果 FeedbackVector 确实被定义了，然后再检查对象的 map 是否被弃用，如果被弃用，就转到 miss 分支，这里会调用一个名为 kLoadIC_Miss 的运行时函数来处理这种情况。如果 map 没有被弃用，那么它就会检查是否满足 monomorphic 的情况，如果满足，就得到 FeedbackVector 中对应的一项，最后调用 HandleLoadICHandlerCase 这个函数来根据反馈信息加载属性值。如果不满足 monomorphic 的情况，那么就只能退而求其次，检查是否满足 polymorphic 的情况，如果满足，那么也能在 FeedbackVector 中找到相关的信息，这部分是用一个循环来寻找的，最后同样调用 HandleLoadICHandlerCase 这个函数加载属性值。如果还是不满足 polymorphic 的情况，那就无法利用 FeedbackVector 中的信息，一种情况是转到之前的 miss 分支，一种情况是转到 stub 分支，调用一个名为 kLoadIC_Noninlined 的内置函数来处理这种情况。

综上所述，Ignition 解释器在生成字节码的过程中涉及寄存器的分配，常量池的构建，还有上下文的跟踪等方面；Ignition 解释器在解释执行字节码的过程中涉及全局分派表，Dispatch 函数的工作原理等方面；最后，Ignition 解释器在函数执行时收集反馈信息的过程中涉及 map，inline cache，FeedbackVector 等方面。