深入剖析 JavaScriptCore

⭐ ming1016.github.io/2018/04/21/deeply-analyse-javascriptcore

前言

最近开始涉及 JS 的解析和处理工作，所以专门研究了下这块。特别是动态类型的处理以及不同引擎对于平台无关的字节码的设计和处理会有很大的帮助。

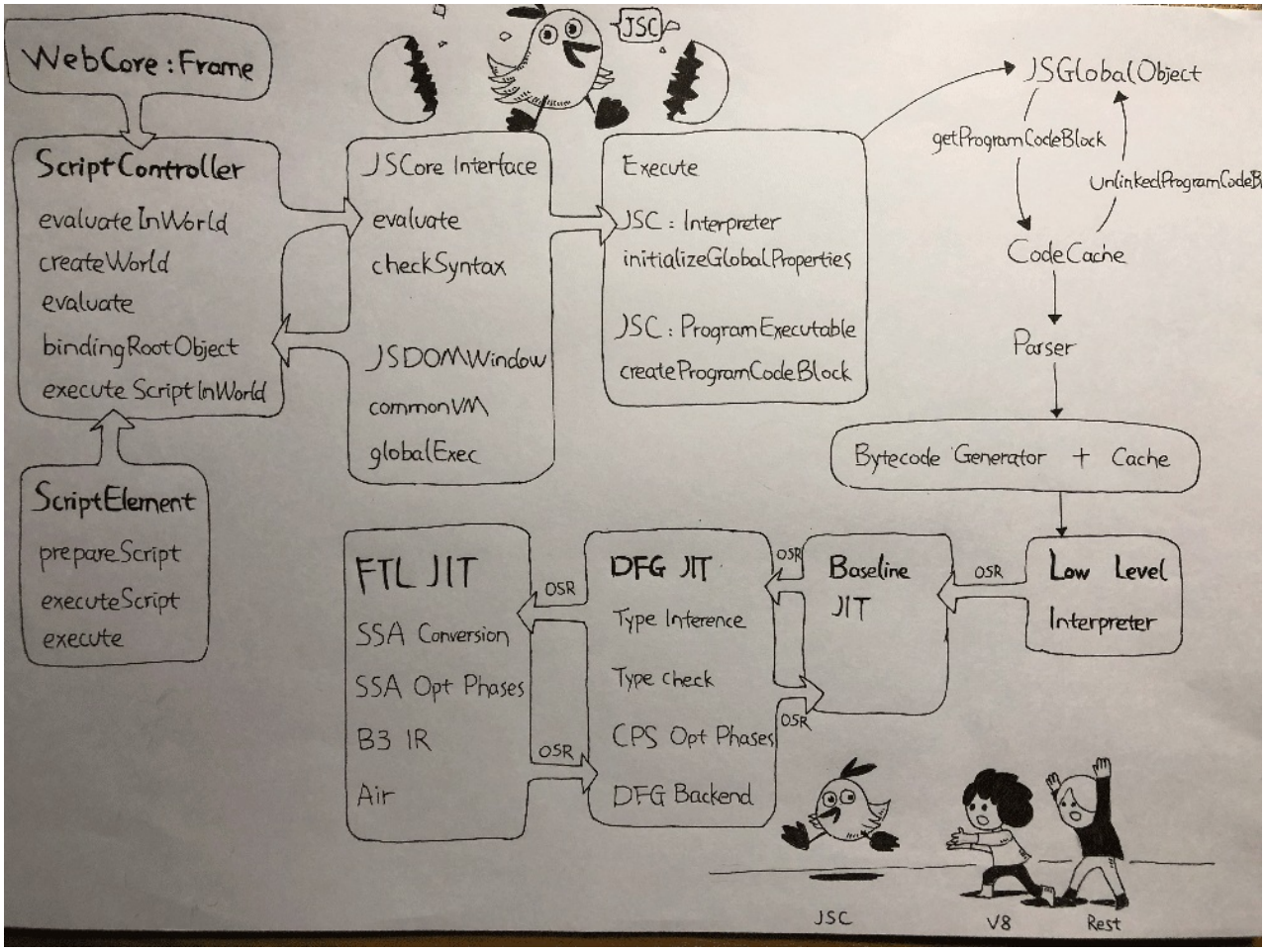
JavaScriptCore 介绍

JavaScriptCore 是 JavaScript 引擎，通常会被叫做虚拟机，专门设计来解释和执行 JavaScript 代码。最开始的 JavaScriptCore 是从 KJS（KDE 的 JavaScript 引擎）以及 PCRE 正则表达式的基础上开发的，是基于抽象语法树的解释器。2008 年重写了，叫做 SquirrelFish，后来是 SquirrelFish Extreme，又叫 Nitro。目前 JavaScript 引擎还有 Google 的 V8 ，Mozilla 的 SpiderMonkey。

JavaScriptCore 还能够在 Objective-C 程序中来执行 JavaScript 的代码，也可以在 JavaScript 环境中插入自定义对象。

JavaScriptCore 全貌

下面是解析 JavaScript 源码解析编译的详细流程图：



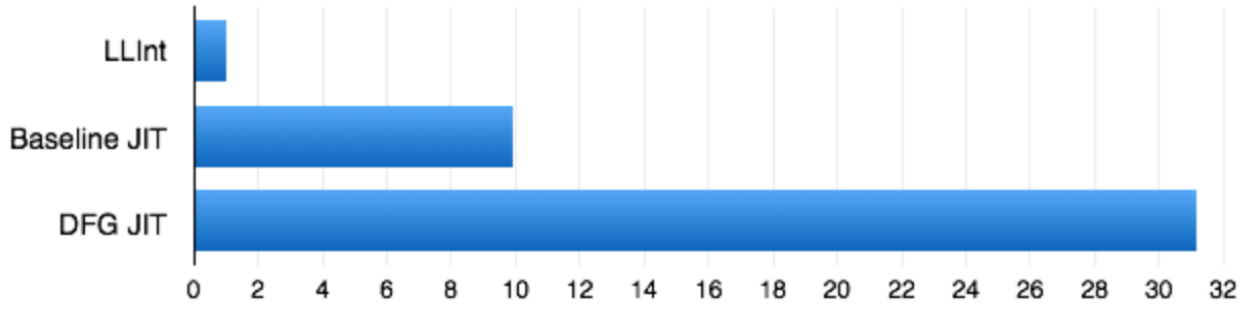
主要模块

- Lexer：词法分析器，生成 tokens，大部分代码都在 parser/Lexer.cpp 里。
- Parser：语法分析，基于 Lexer 的 tokens 生成语法树。手写了个 recusive descent parser 递归下降解析器，代码主要在 parser/Parser.cpp 里。
- LLInt：Low Level Interpreter 执行 Parser 生成的 Byte code。代码在 llint/ 里，使用汇编，在 offlineasm/ 里，可以编译为 x86 和 ARMv7 的汇编和 C 代码。LLInt 希望达成除了词法和语法分析外零启动消耗，同时遵守用 JIT 在调用，堆栈和起存器的约定。
- Baseline JIT：实时编译，性能不好用这个。在函数调用了 6 次，或者某段代码循环了大于100次会被触发。BaseLine JIT 的代码在 jit/ 里。BaseLine JIT 还对几乎所有堆的访问执行了复杂的多态内联高速缓存（Polymorphic inline caches）。多态内联缓存是 Smalltalk 社区优化动态分发的一个经典技术。
- DFG JIT：低延迟优化 JIT，更差性能就用这个生成更优化的机器码来执行。在函数被调用了60次或者代码循环了1000次会触发。在 LLInt 和 Baseline JIT 中会收集一些包括最近参数，堆以及返回值中的数据等轻量级的性能信息，方便 DFG 进行类型判断。先获取类型信息可以减少大量的类型检查，推测失败 DFG 会取消优化，也叫 OSR exit。取消可以是同步的也

可以是异步的。取消后会回到 Baseline JIT，退回一定次数会进行重新优化，收集更多统计信息，看情况再次调用 DFG。重新优化使用的是指数式回退策略应对一些怪异的代码。DFG 代码在 dfg/ 里。

- FTL：高吞吐量优化 JIT，全称 Faster Than Light，DFG 高层优化配合 B3 底层优化。以前全称是 Fourth Tier LLVM 底层优化使用的是 LLVM。B3 对 LLVM 做了裁剪，对 JavaScriptCore 做了特性处理，B3 IR 的接口和 LLVM IR 很类似。B3 对 LLVM 的替换主要是考虑减少内存开销，LLVM 主要是针对编译器，编译器在这方面优化动力必然没有 JIT 需求高。B3 IR 将指针改成了更紧凑的整数来表示引用关系。不可变的常用的信息使用固定大小的结构来表示，其它的都放到另外的地方。紧凑的数据放到数组中，更多的数组更少的链表。这样形成的 IR 更省内存。Filip Pizlo 主导的这个改动，DFG JIT 也是他弄的，为了能够更多的减少内存上的开销，他利用在 DFG 里已经做的 InsertionSet 将 LLVM IR 里的 def-use 干掉了，大概思路是把单向链表里批量插入新 IR 节点先放到 InsertionSet 里，在下次遍历 IR 时再批量插入。Filip Pizlo 还把 DFG 里的 UpsilonValue 替代 LLVM SSA 组成部分。B3 后面会把 LLVM 的寄存器分配算法 Greedy 一直到 B3 中。

执行速度的对比：



相对速度，越高表示越好。

更多的说明可以参看 WebKit 官网 JavaScriptCore 的 Wiki 首页部分：https://trac.webkit.org/wiki/JavaScriptCore

主要的源码目录结构

- API：JavaScriptCore 对外的接口类
- assembler：不同 CPU 的汇编生成，比如 ARM 和 X86
- b3：ftl 里的 Backend
- bytecode：字节码的内容，比如类型和计算过程
- bytecompiler：编译字节码
- Configurations：Xcode 的相关配置
- Debugger：用于测试脚本的程序
- dfg：DFG JIT 编译器

- disassembler：反汇编
- heap：运行时的堆和垃圾回收机制
- ftl：第四层编译
- interpreter：解释器，负责解析执行 ByteCode
- jit：在运行时将 ByteCode 转成机器码，动态及时编译。
- llint：Low Level Interpreter，编译四层里的第一层，负责解释执行低效字节码
- parser：词法语法分析，构建语法树
- profiler：信息收集，能收集函数调用频率和消耗时间。
- runtime：运行时对于 js 的全套操作。
- wasm：对 WebAssembly 的实现。
- yarr：Yet Another Regex Runtime，运行时正则表达式的解析

JavaScriptCore 与 WebCore

ScriptController 会调用 JavaScriptCore 的 evaluate 和 checkSyntax 两个接口。DOM 节点的 JSBindings 通过回溯到 JSC::JSNonFinalObject 实现和 JavaScriptCore 的绑定。

VM 是 JavaScript 的 Runtime 环境。GlobalObject 是全剧对象，负责管理执行环境和 Object 的。ExecState 是执行脚本的对象，由 GlobalObject 管理的，负责记录脚本执行上下文。

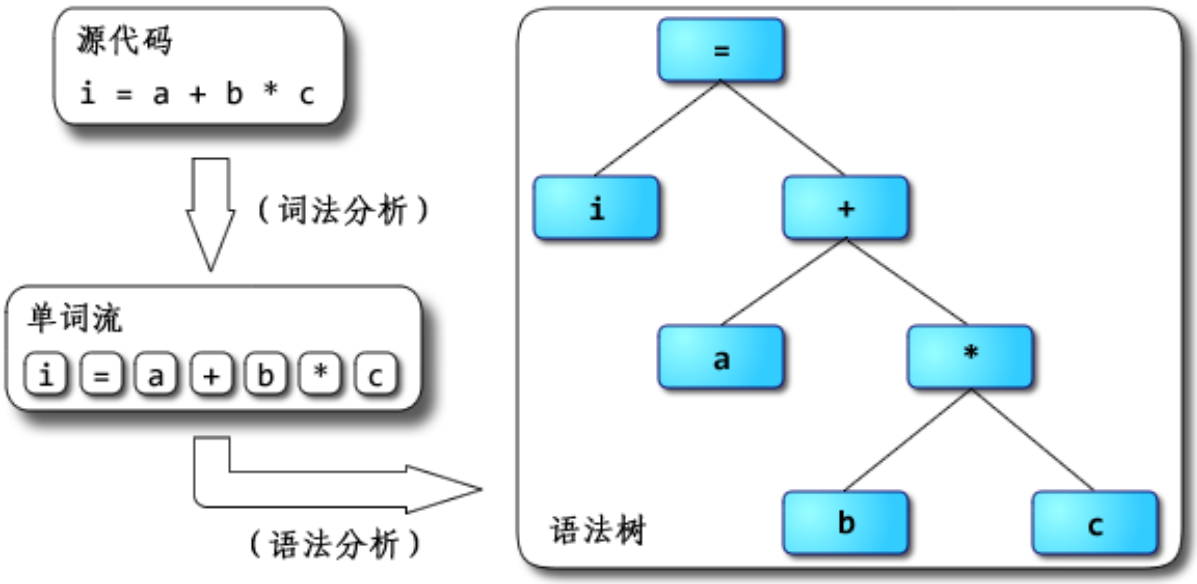
接口执行脚本，创建一个 ProgramExecutable 对象来表示要执行的脚本负责编译成 ByteCode，调用 Interpreter 执行这个 ByteCode。

Binding 是 WebCore 为 JavaScriptCore 提供的封装层。这一层的定义可以在这里找到：https://trac.webkit.org/wiki/WebKitIDL。WebKit 参照的是 W3C Web IDL https://www.w3.org/TR/WebIDL-1/。

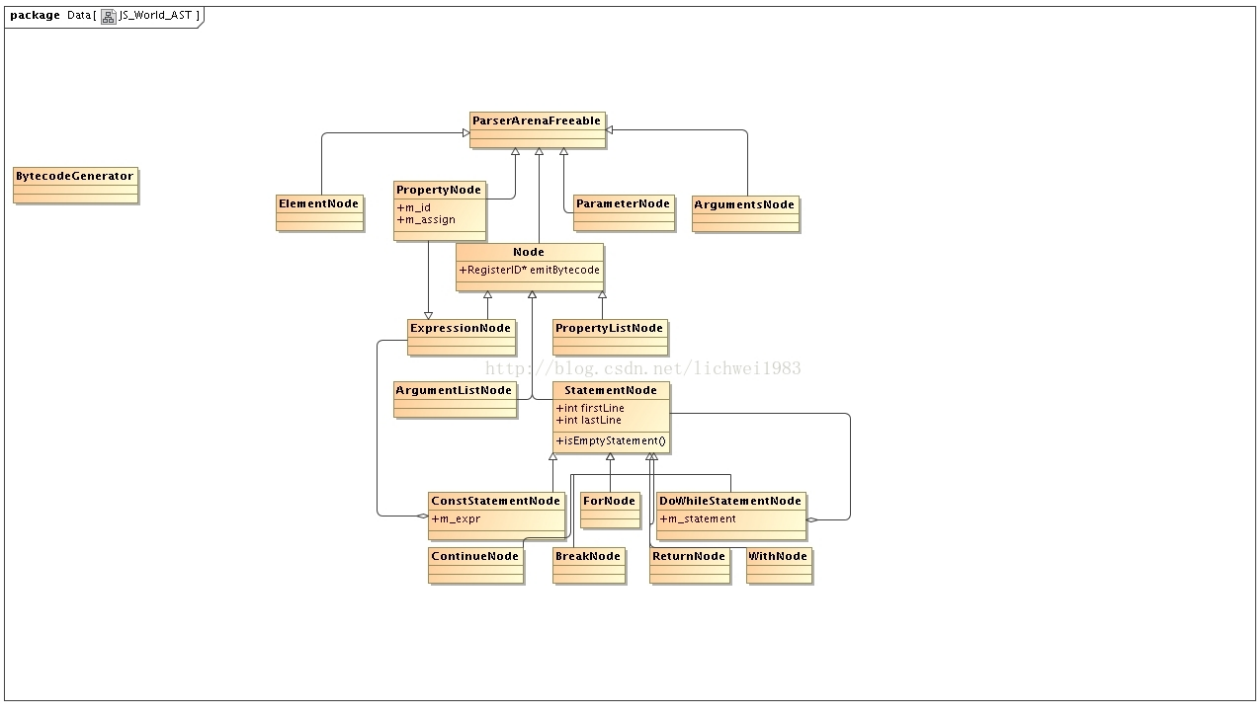
使用 IDL 定义接口规范，WebKit 使用一组 perl 脚本来转换 IDL，初始脚本是 generate-binding.pl。生成接口与 DOM 组件关联是通过 JSNode 来的。执行脚本和 Frame 的 setDocument 会更新 document 对象到 JavaScriptCore，通过 JSDomWindowBase::updateDocument 更新到 JSC::Heap 里。

词法语法分析

词法和语法分析程序 JavaScriptCore 是自己编写的。词法分析会把文本理解成很多 token，比如 a = 5; 就会被识别成下面这些 token，VARIABLE EQUAL CONSTANT END。然后通过语法分析，输出语法树。

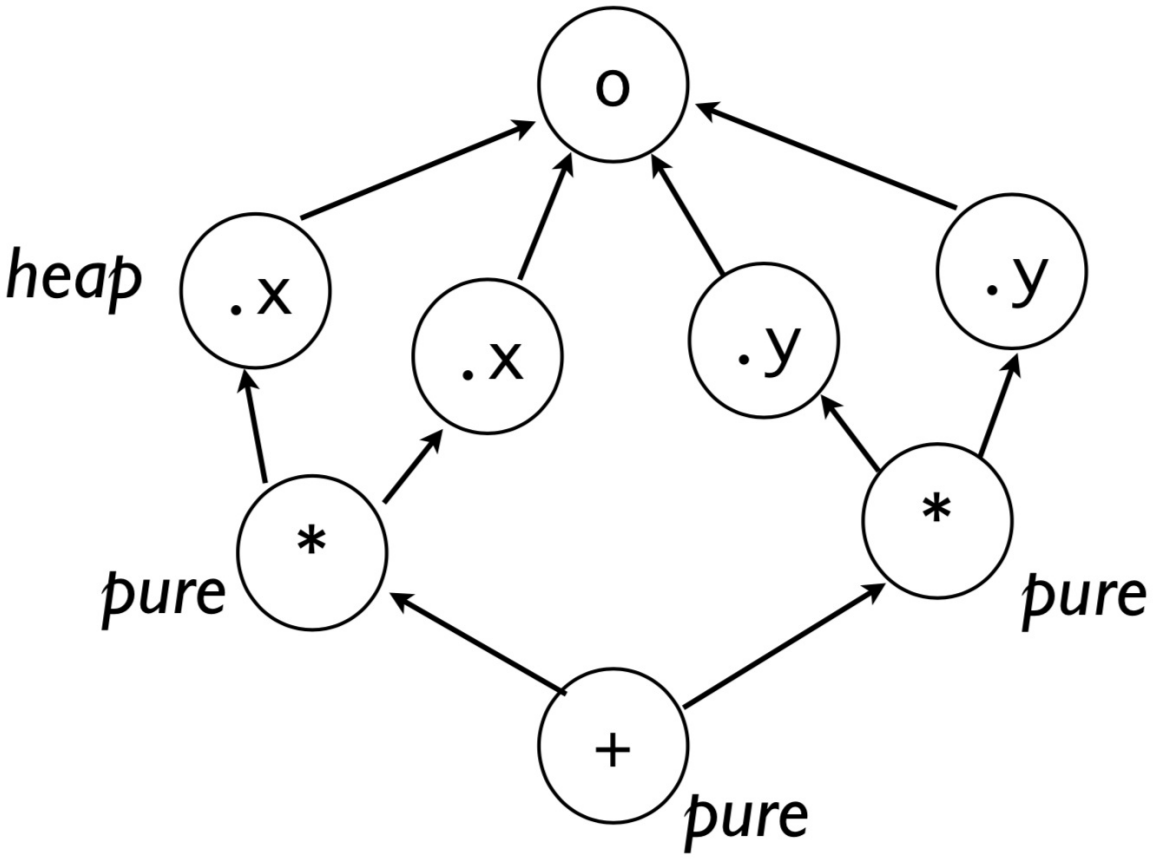


需要先设计好 Node，Node 的类关系图如下：



举个包含加和乘的 statement 是如何组成 AST 的

```
1  o.x * o.x +
   o.y * o.y
```

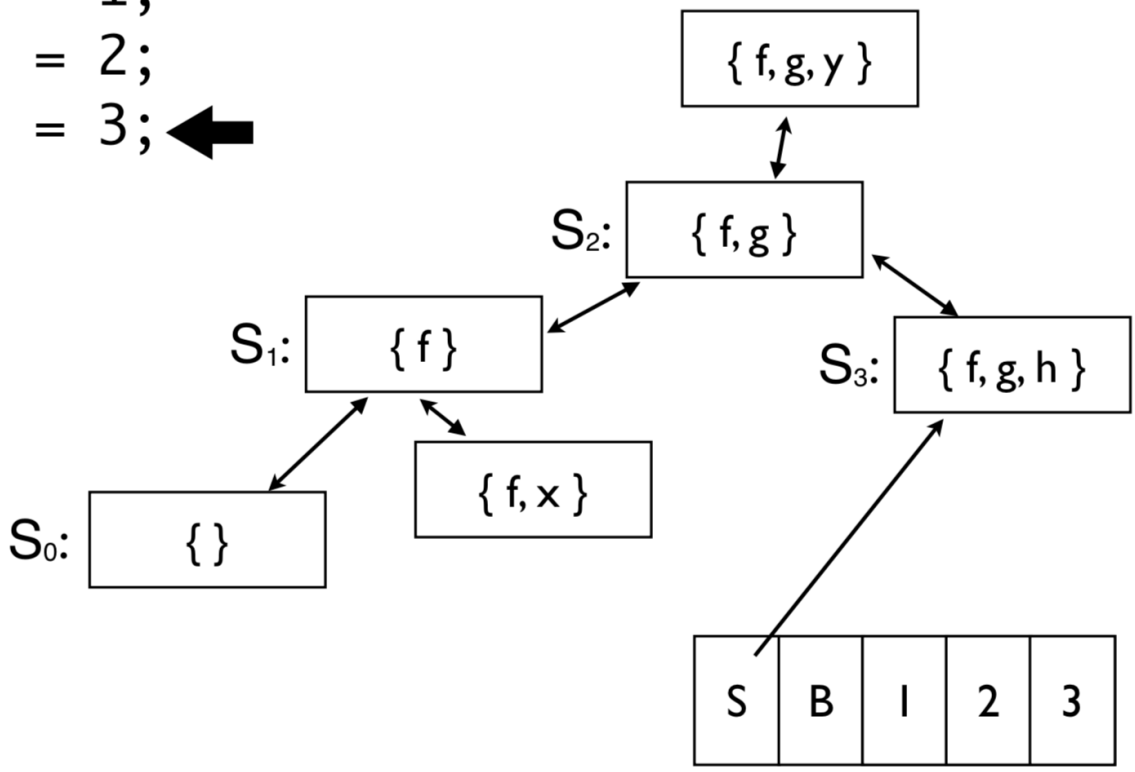


heap 代表的是可见数据，pure 代表的是抽象说明。

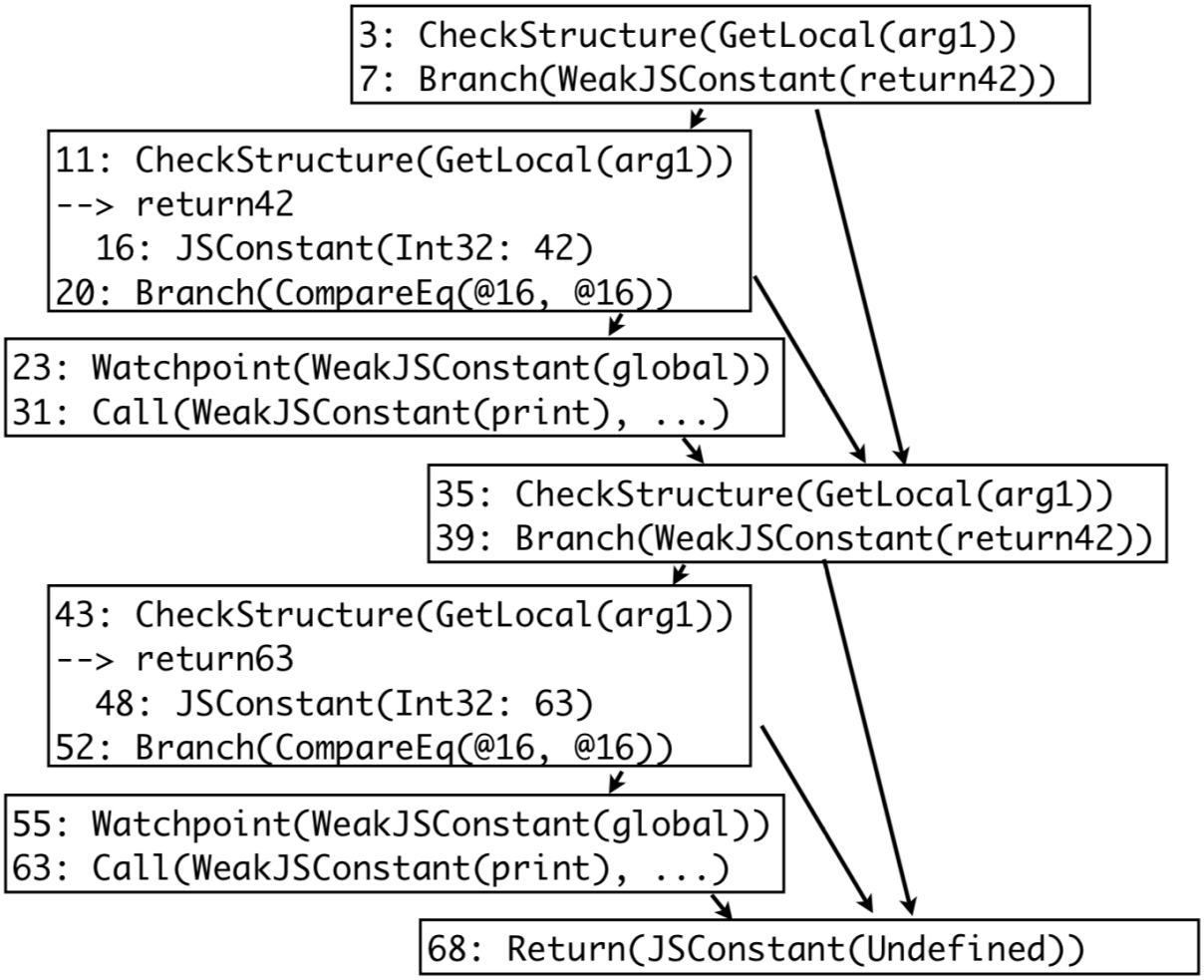
下面看看 Object 的构造

```
1  var o = new
2  Object();
3  o.f = 1;
4  o.g = 2;
   o.h = 3;
```

```
var o = new Object();
o.f = 1;
o.g = 2;
o.h = 3;
```



接下来看看 function 的



还有些其它的 parser 比如 Esprima 提供了一个在线 demo 可以在线 Parser。实现的代码在这里：https://github.com/jquery/esprima/tree/master/src 主要使用的是递归下降和运算符优先级混合式来做的 parser。

我在 HTN （https://github.com/ming1016/HTN）项目中做了个 js 的 AST 的 builder：https://github.com/ming1016/HTN/tree/master/Sources/Core/JavaScript 分词和语法树生成使用的是状态机处理递归下降。

WebKit 的性能目录里有用 ES6 标准实现的 ECMA-55 BASIC 这个词法语法分析的测试。代码在：https://trac.webkit.org/browser/trunk/PerformanceTests/ARES-6/Basic?rev=211697

程序会被表示成树，每个节点都有与其相关的代码，这些代码都可以递归的调用树中的子节点。Basic 的节点的结构是这样子的：

```
1  {evaluate: Basic.NumberPow, left: primary, right:
   parsePrimary()}
```

Basic 是以不那么常见的方式使用生成器的，比如多个生成器函数，很多 yield points 和递归生成器调用。Basic 也有 for-of，类，Map 和 WeakMap。所以上的测试 ES6 程序通过 js 写的程序帮助理解 ES6 的解析过程。

代码到 JIT 的过程

ProgramExecutable 的初始化会生成 Lexer，Parser 和字节码。入口是从 JS Binding 那层里调用 ScriptController::evaluateInWorld 进来，这个方法里的参数 sourceCode 就是 js 的代码的来源，方法内通过调用 runtime/Completion.cpp 里的方法 evaluate 来进入 executeProgram 方法的。总的来说 ProgramExecutable 主要是把代码编成字节码，Interpreter 是来执行字节码的。

```
1  JSValue result = vm.interpreter->executeProgram(source, exec,
   thisObj);
```

executeProgram 方法将源码生成 ProgramExecutable 这个对象。

```
1  ProgramExecutable* program = ProgramExecutable::create(callFrame,
   source);
```

这个对象里有 StringView 对象，program->source().view() 这样就可以对源码进行操作了。在那之前会先判断下是否这段 js 代码仅仅只是一个 JSON 对象，如果是就地按照 JSON 对象处理，不然就按照普通的 js 代码来处理。

普通处理会先编译成字节码，过程是先初始化全局属性：

```
1  JSObject* error = program->initializeGlobalProperties(vm, callFrame,
   scope);
```

处理的结果都会记录在 callFrame 里。主要是通过 JSGlobalObject 这个对象的 addFunction 和 addVar 方法记录 Parser 出那些在全局空间的那些 let，const 和 class 的全局属性，或者 var，let 和 const 的全局变量。

接下来会创建一个 codeBlock：

```
ProgramCodeBlock* codeBlock;
    {
        CodeBlock* tempCodeBlock;
        JSObject* error = program->prepareForExecution<ProgramExecutable>(vm, nullptr, scope, CodeForCall,
tempCodeBlock);
        EXCEPTION_ASSERT(throwScope.exception() == reinterpret_cast<Exception*>(error));
        if (UNLIKELY(error))
            return checkedReturn(error);
        codeBlock = jsCast<ProgramCodeBlock*>(tempCodeBlock);
    }
```

这里 ProgramExecutable 会通过 prepareForExecution 方法里调用 prepareForExecutionImpl 来创建一个新的 CodeBlock。

```
CodeBlock* codeBlock = newCodeBlockFor(kind, function, scope, exception);
resultCodeBlock = codeBlock;
EXCEPTION_ASSERT(!!throwScope.exception() == !codeBlock);
if (UNLIKELY(!codeBlock))
    return exception;
if (Options::validateBytecode())
    codeBlock->validate();
if (Options::useLLInt())
    setupLLInt(vm, codeBlock);
else
    setupJIT(vm, codeBlock);
installCode(vm, codeBlock, codeBlock->codeType(), codeBlock-
>specializationKind());
```

这里看到，如果 Options 是指定 LLInt 的话就会调 setupLLInt 方法去设置这个 codeBlock，不然就通过 JIT 来。LLInt 在 LLIntSlowPaths.cpp 里通过 C 函数封装了 LowLevelInterpreter.asm 里的执行指令的汇编。触发 JIT 优化是通过热点探测方法，LLInt 会在字节码的 loop_hint 循环计数和 ret 函数返回指令的时候进行统计，结果保存在 ExcutionCounter 里。当函数或循环体执行一定次数时，通过 checkIfThresholdCrossedAndSet 方法得到布尔值结果来决定是否用 JIT 编译。

CodeBlock 有 GlobalCode，EvalCode，FunctionCode，ModuleCode 这些类型，用于 LLInt 和 JIT。编译后的 ByteCode 会存在 UnlinkedCodeBlock 里。

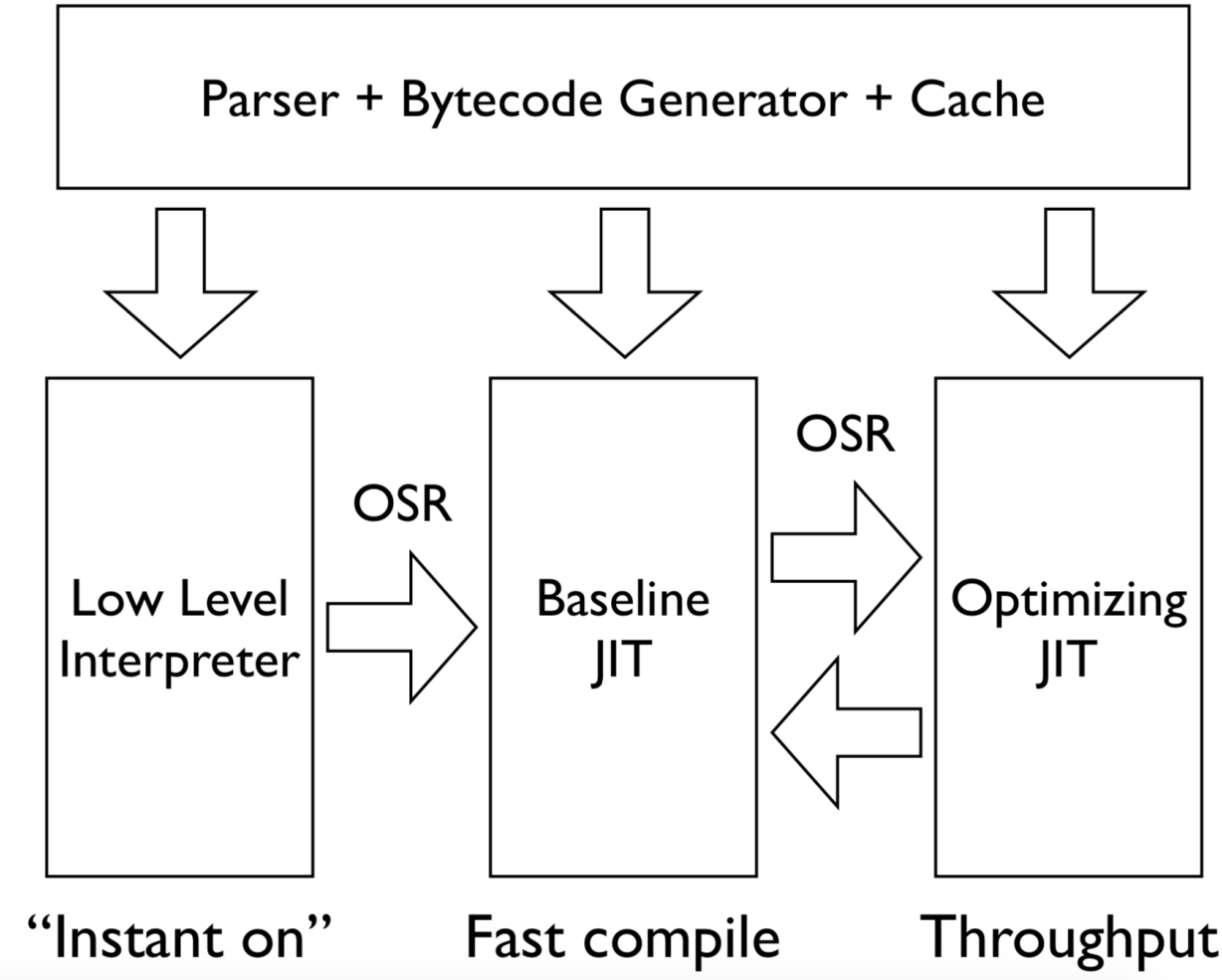
分层编译

总的来说 JavaScriptCore 是采用了类型推测和分层编译的思想，解析成字节码后 LLInt 的作用就是让 js 代码能够早点执行，由于解释的效率不高，所以达到一定条件可以并行的通过 Baseline JIT 编译成更高效的字节码来解释，如果到了更加容易使得 LLInt 解释效率差的情况就会并行使用

DFG JIT ，DFG 编译的结果也有代表 On stack replacement 的 osrExitSites 数组，这样的字节码的解释性能会更好，如果假设解释失败还可以再回到 Baseline，也不会有什么问题。启用 FTL 的条件就更高了，可能函数调用了好几万次也都不会开启。

分层编译的主要思想是先把源码解释成一种内部的表示也就是字节码，将其中使用率高的字节码转成汇编代码，汇编代码可以由 CPU 直接执行来最大程度的提高性能。

LLInt，Baseline JIT 和 DFG 三者会同时运行。可以在 jsc.cpp 里添加日志观察他们的工作状态



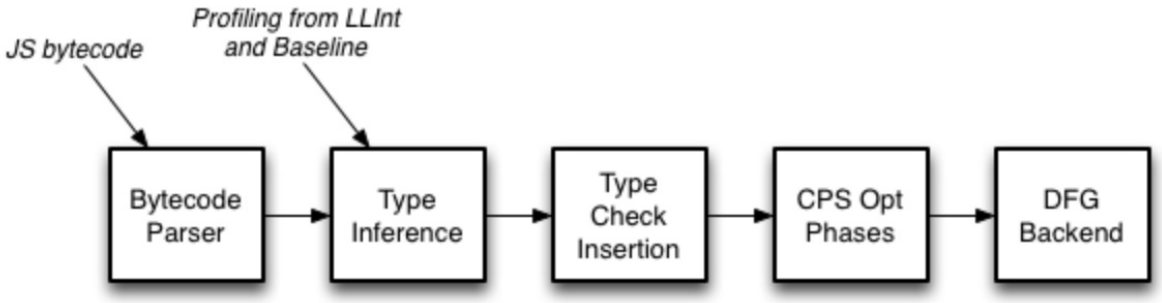
箭头指示的是堆栈替换 on-stack replacement，简称 OSR。这个技术可以将执行转移到任何 statement 的地方。OSR 可以不管执行引擎是什么都在去解析字节码状态，并且能够重构它去让其它引擎继续执行，OSR entry 是进入更高层优化，OSR exit 是降至低层。LLInt 到 Baseline JIT 时 OSR 只需要跳转到相应的机器代码地址就行，因为 Baseline JIT 每个指令边界上的所有变量的表示和 LLInt 是一样的。进入 DFG JIT 就会复杂些，DFG 通过将函数控制流图当作多个进入点，一个是函数函数启动入口一个是循环的入口。

如果每个函数都用像 DFG JIT 来优化那么消耗太大，这就类似每次运行就来个完整的源码编译成本地应用那样。如果每个语句都只执行一次，那么 JIT 编译就会比 LLInt 是时间要长，如果执行的次数越多，那么 LLInt 这种解释方式就会比编译方式要差，LLInt 的大部分时间都消耗在分派下个字

节码指令的操作上了。

DFG JIT

全称 data flow graph JIT 数据流图 JIT。是一种推测优化的技术。会开始对一个类型做出一个能够对性能好的假设，先编译一个版本，如果后面发现假设不对就会跳转回原先代码，称为 Speculation failure。DFG 是并发编译器，DFG pipeline 的每个部分都是同时运行的，包括字节码解析和分析。



上图是 DFG JIT 的优化 pipeline。开始 DFG 会把字节码转成 DFG CPS 格式，这个格式会描述变量和临时数据之间的数据流关系。再用分析信息来推测类型，通过推测的类型减少类型的检查。接着进行传统的编译器方面的优化，最后编译器通过 DFG CPS 格式直接生成机器码。

DFG JIT 会将字节码解析成 SSA 形式，按执行路径收集类型信息。会使用 inlining 和 value profiling 等一些静态分析技术。类型推导把 value profile 里的常用的类型作为后面用的类型来预测，在 SpeculatedType.h 里定义里一些数据类型，符合 Data-flow analysis 规范，具体实现在 DFGPredictionPropagationPhase.cpp 里。Baseline JIT 次数超过一定数量就会生成一个新的类型，可能会触发 DFG 也可能会让 Baseline JIT 再执行几次。

DFG 的 register allocator 中汇编程序使用的是二地址，不用 op dest, src1, src2 这样三地址形式，只需要 op src1, src2 这样的二地址形式，结果存在第一个操作数里。

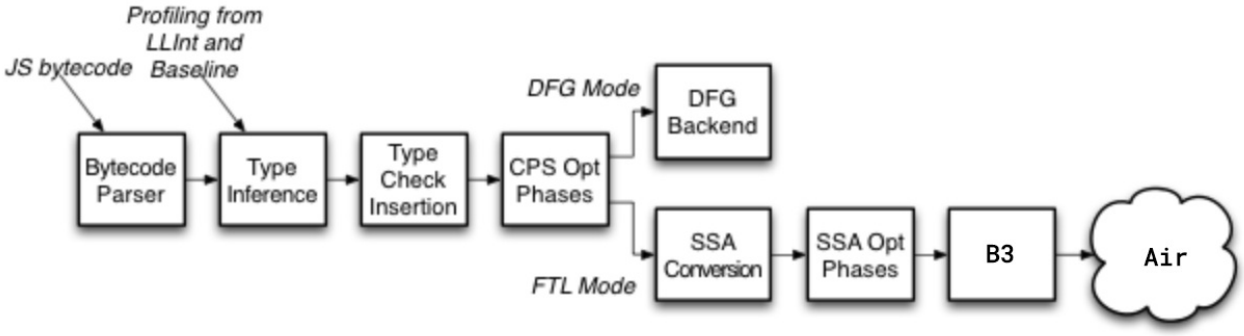
DFG JIT 将低效字节码转成更优化的高效形式。DFG 结合传统编译器的优化，比如寄存器的分配，控制流图的简化，公共子表达式消除，死代码消除和稀疏条件常量传播等。但是正儿八经的编译器是需要知道变量类型和堆里面的对象的结构的，这样才能更好的优化。DFG JIT 使用的是 LLInt 和 Baseline JIT 里分析出的变量类型来推断的。比如下面的例子

```
1  function
2  plusThe(x) {
3      return x + 1;
   }
```

这里的 x 从代码上看是看不出类型的，它可能是字符串也可能是 integer 或者 double。LLInt 和 Baseline JIT 会从函数参数或者来自堆的读取来收集每个变量的值信息。DFG 不会一开始就编译 plusThe 这个函数，它会等到这个函数被执行了很多次，让 LLInt 和 Baseline JIT 尽可能的把类型信息收集全。根据这些信息来假设一个类型，一个确定的类型会节省空间和执行时间。如果假设的检查失败，就通过 OSR 把执行转移到 Baseline JIT 上。

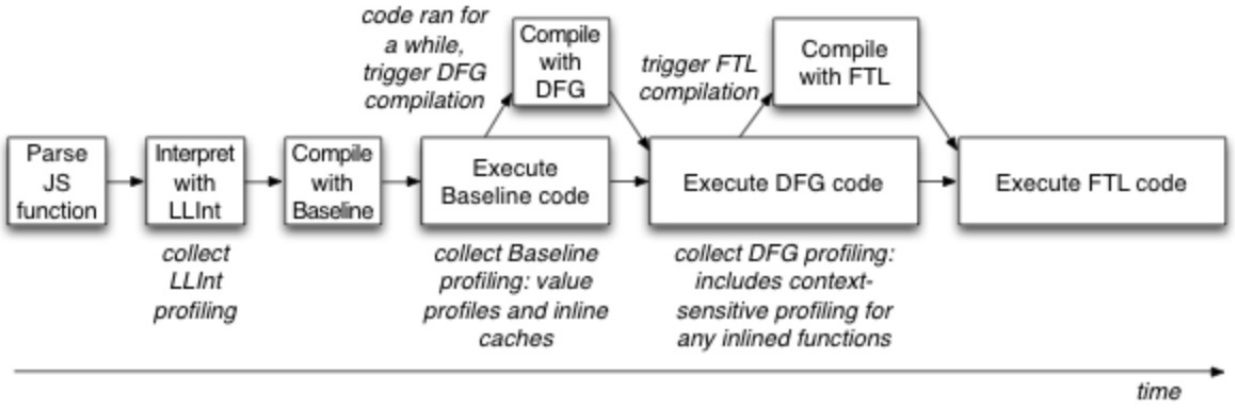
FTL JIT

DFG JIT 可以优化 long-running 运行的代码性能，但是会影响 short-running 的程序，DFG 编译的时间和 DFG 生成代码的质量相比较来说还是过大。想让 DFG 生成更好的代码都会降低代码速度，还会增加 shorter-running 代码的延迟。DFG 不可能同时成为低延迟优化和 high-throughput 代码编译器。这时就需要再多一层来做较重的优化，同时让 DFG 保持自己的轻体重，使得 longer-running 和 shorter-running 代码能够保持平衡。



新的一层 FTL 实际上是 DFG Backend 的替换。会先在 DFG 的 JavaScript 函数表示转换为静态单一指派（SSA） 格式上做些 JavaScript 特性的优化。接着把 DFG IR 转换成 FTL 里用到的 B3 的 IR。最后生成机器码。

DFG 的机器代码生成器很快，但没有低级优化。对这块的优化采用先转成 SSA 形式，执行比如自动执行循环不变量代码来提高执行速率的循环不变量代码移动（loop-invariant code motion）的优化技术，这些优化完成后就可以 straight-forward linear 和 一对多（DFG SSA 的每个指令都可以产生）的转换为也基于 SSA 的没有 JavaScript 语言特性的 B3 的 IR。总的来说过程就是把源码生成字节码，接着变成 DFG CPS IR，再就是 DFG SSA IR，最后成 B3 的 IR，JavaScript 的动态性就是在这些过程中一步步被消除掉的。



代码在 LLInt，Baseline JIT 和 DFG JIT 运行一段时间才会调用 FTL。FTL 在并发线程中会从它们那收集分析信息。short-running 代码是不会导致 FTL 编译的，一般超过10毫秒的函数会触发 FTL 编译。

FTL 通过并发编译减小对启动速度的影响。

FTL Backend B3

B3 的全称 Bare Bones Backend，实现了大吞吐量同时缩短总体 FTL 编译时间。B3 的 IR 是一个低级别的中间表示，低级别意味着需要大量内存去表示每个函数，大量的内存意味着编译器在分析函数时需要扫描大量内存。B3 有两个 IR，一个叫 B3 IR 另一个机器级别的叫 Assembly IR，简称 Air。这两个 IR 的目标是代表低级操作，同时能最大限度的减少分析代码所需要的内存访问。达到这个目标需要减少 IR 对象整体大小，减少表示典型操作的 IR 对象数量，减少 IR 的 pointer chasing，最后时减少 IR 的总数。

B3 转换成 Air 会通过执行反向贪婪模式匹配来巧妙地针对每个 B3 操作选择正确的 Air 序列。具体的实现可以参看 B3LowerToAir.cpp 的代码。

下面来看看 B3 IR 的所做的优化。

- Strength reduction：主要包括控制流程图的简化，常量合并，消除死代码，剪除整数溢出的检查，还有其它简化规则。实现文件是 B3ReduceStrength.cpp。
- Flow-sensitive 常量合并：在 B3FoldPathConstants.cpp 文件里。
- 全局共用子表达式消除：实现文件 B3EliminateCommonSubexpressions.cpp。
- Tail duplication：B3DuplicateTails.cpp
- SSA 修正：B3FixSSA.cpp
- 优化常量实际的位置：B3MoveConstants.cpp

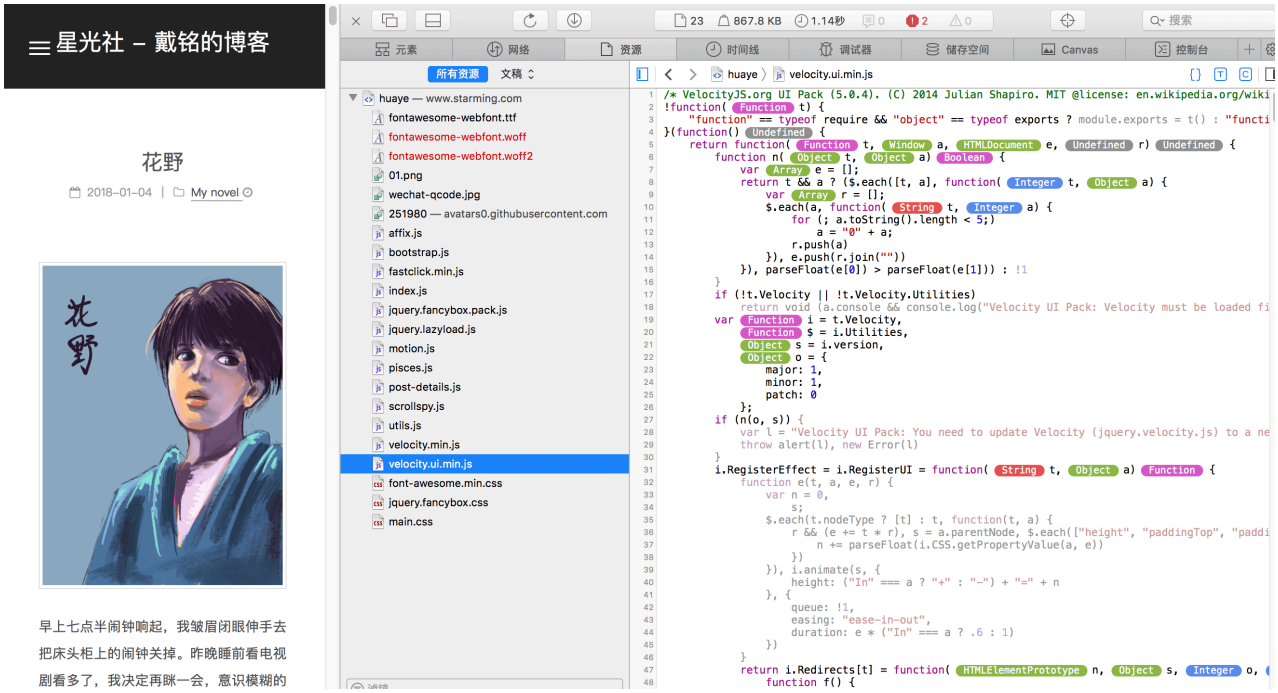
Air 里做的优化包括去除死代码，控制流图的简化以及修正部分寄存器 stalls 和溢出代码症状。Air 最重要的优化是寄存器分配。

B3 的动态语言支持靠的是 Patchpoints，还有一半功劳是操作码的 Check family，用来实现栈上替换，即 OSR。JavaScript 是种动态语言，没有可以快速执行的类型和操作，FTL JIT 就使用推测行为方式把这些操作转换成快速代码。

Air 在寄存器分配器的选择上使用了经典的图形着色寄存器分配器，叫迭代寄存器合并，简称 IRC。

类型分析

Web Inspector 工具是 WebKit 里提供的调试工具，通过这个工具可以很好的观察哪些函数甚至哪些条件执行了或者没有执行。最重要的是可以直观的观察变量类型，还能够跟踪值的继承链。



JavaScript 是动态类型语言，任何变量都可以是任何类型，表达式可以有任何类型，函数可以返回任何类型等等。由于不能立刻得到类型就没法确定地址字节大小，需要在运行时去反复推断。静态语言就不一样，一开始明确了类型，比如目标平台 int 类型占用4个字节，它对应的对象是个地址，偏移量是0，访问 int 时需要将对象地址加上4个字节。所以语言解释系统只要用数组和位移来存变量和方法地址就行了，这样几个机器语言的指令就可以对其执行各种操作。

下面来个例子：

```
let x = 3;
x = ["These", "is", "array with
string"];
const justReturn = (x) => {
return x; };
justReturn(10);
justReturn([1, 2, 3]);
justReturn("I am bad!");

```

JavaScript 没有类型限制，语法不错就没问题，如果是静态类型语言的话就没法像上面的例子一样把数组赋给数字类型的变量。在 Type Profiler 中会显示类型名为 String? 和 Integer? 这样后面跟着问号符号的表示，这是在有多个不同类型被分配给相同变量，函数参数传递和函数返回时发生的。查看时可以看到所有这些被收集的类型的信息。

那么 JavaScriptCore 是如何分析变量类型的呢？先看段 js 代码：

```
function
add(x, y) {
    return x +
y;
}
```

对应的字节码：

```
function add: 10 m_instructions; 3 parameter(s); 1
variable(s)
[ 0] enter
[ 1] get_scope       loc0
[ 3] add             loc1, arg1, arg2
[ 8] ret             loc1
```

下面看看开启了 Type Profiler 后的字节码是什么样的：

```
function add: 40 m_instructions; 3 parameter(s); 1
variable(s)
[ 0] enter
[ 1] get_scope         loc0
// 分析函数参数
[ 3] op_profile_type   arg1
[ 9] op_profile_type   arg2
// 分析 add 表达式的操作数
[15] op_profile_type   arg1
[21] op_profile_type   arg2
[27] add               loc1, arg1, arg2
// 分析返回语句，收集这个函数的返回类型信息
[32] op_profile_type   loc1
[38] ret               loc1
```

下面是进入 DFG 时转换成 DFG IR 的代码：

```
0:  SetArgument(this)
1:  SetArgument(arg1)
2:  SetArgument(arg2)
3:  JSConstant(JS|PureInt, Undefined)
4:  MovHint(@3, loc0)
5:  SetLocal(@3, loc0)
6:  JSConstant(JS|PureInt, Weak:Cell: 0x10f458ca0 Function)
7:  JSConstant(JS|PureInt, Weak:Cell: 0x10f443800
GlobalScopeObject)
8:  MovHint(@7, loc0)
9:  SetLocal(@7, loc0)
10: GetLocal(JS|MustGen|PureInt, arg1)
11: ProfileType(@10)
12: GetLocal(JS|MustGen|PureInt, arg2)
13: ProfileType(@12)
14: ProfileType(@10)
15: ProfileType(@12)
16: ValueAdd(@10, @12, JS|MustGen|PureInt)
17: MovHint(@16, loc1)
18: SetLocal(@16, loc1)
19: ProfileType(@16)
20: Return(@16)
```

现在还有很多 ProfileType，这些操作会有很大的消耗，DFG 会推测参数类型是整数，接下来会在这个假设下将 ProfileType 的那些操作移除掉。

```
1  1:  SetArgument(arg1)
2  2:  SetArgument(arg2)
3  3:  JSConstant(JS|PureInt, Undefined)
4  4:  MovHint(@3, loc0)
5  7:  JSConstant(JS|PureInt, Weak:Cell: 0x10f443800
6  GlobalScopeObject)
7  8:  MovHint(@7, loc0)
8  10: GetLocal(@1, arg1)
9  12: GetLocal(@2, arg2)
1  16: ArithAdd(Int32:@10, Int32:@12)
0  17: MovHint(@16, loc1)
1  20: Return(@16)
1
```

在对类型的处理上 V8 使用了一个结合 C++ 使用类和偏移位置思想的隐藏类来解决通过字符串匹配找属性的算法。做法是将有相同属性名和属性值的对象保存在同一个组的隐藏类里，这些属性在隐藏类里有着同样的偏移值，这样这个组里的对象能够共用这个隐藏类的信息。访问属性的过程是得到隐藏类的地址，根据属性名得到偏移值，通过偏移值和隐藏类地址得到属性地址。那么这个过程是否可以加速呢？答案是肯定的，通过 Inline Cache 缓存之前查找结果来减少方法和属性的哈希表查找时间。当一个对象或属性类型出现多种时缓存就会不断更新，V8 就会退到先前按照哈希表查找的方式来。

指令集架构

JavaScriptCore 是基于寄存器的虚拟机 register-based VM。这种实现方式不用频繁入栈，出栈和三地址的指令集，所以效率高，但移植性弱点。

基于寄存器指令集架构

三地址和二地址的指令集，基本都是使用基于寄存器的架构来实现的，要求除了 load 和 store 外的运算指令的源都要是寄存器。

下面的代码

```
1  i =
   a
   +
   b;
```

上面这句转换成机器指令样子

```
1  add
   i, a,
   b
```

这样的形式就是三地址指令，很多的代码都是这样的二元运算然后再赋值，三地址正好可以分配两地址给二元运算的两个源，剩下一个地址给赋值目标。ARM 处理器的主要指令集就是三地址的。那么二地址是怎么处理的呢？将上面的代码换成下面的样子：

```
1  i
2  +
   =
   a
   ;
   i
   +
   =
   b
   ;
```

机器指令形式就变成

```
1  a
2  d
   d
   i,
   a
   a
   d
   d
   i,
   b
```

这样一个源同时也作为赋值目标，X86 系列的处理器就是采用的二地址。

有了二地址和三地址，那么有一地址么。

```
1  a
2  d
   d
   a
   a
   d
   d
   b
```

这就是一地址，只有操作源，那么目标呢？目标 i 是隐藏目标，这种运算的目标称为累加器的专用寄存器，运算都是依赖更新累加器的状态来完成。

基于栈指令集架构

那么零地址呢？JVM 就是采用的这种。那么就先看下一段 JAVA 代码：

```
class QuickCalculate {
    byte onePlusOne()
{
        byte x = 1;
        byte y = 1;
        byte z = (byte) (x
+ y);
        return z;
    }
}
```

转换成字节码：

```
iconst_1   // Push 整数常量1
istore_1    // Pop 到局部变量1，相当于 byte x = 1; 这
句
iconst_1   // 再 Push 整数常量1
istore_2    // Pop 到局部变量2，相当于 byte y = 1; 这
句
iload_1     // Push x，这时 x 已经作为整数存在局部变量
1里。
iload_2     // Push y，这时 y 已经作为整数存在局部变量
1里。
iadd        // 执行加操作，栈顶就变成了 x + y，结果为
一个整数
int2byte   // 将整数转化成字节，结果还是占32位
istore_3   // Pop 到局部变量3里，byte z = (byte)(x +
y)
iload_3    // Push z 的值使之可以被返回
ireturn     // 返回结果，return z
```

整个过程我放到了注释里。可以看到零地址形式的指令集就是基于栈架构的。这种架构的优势是可以用更少的空间存更多的指令，所以空间不是很富足时这种架构是可取的，不过零地址要完成一件事会比基于寄存器指令架构的二地址，三地址指令要多很多指令，执行效率还会低。

指令集架构演化和比较

JavaScriptCore 采用 SquirrleFish 之前的解释器他们都是树遍历式的，解释器会递归遍历树，在遍历树上的每个节点的操作都是根据解释其每个字节点返回的值来的。这种操作即不是基于栈也不是基于寄存器。举个例子

```
i = x
+ y *
z
```

按 AST 的后序遍历，最上面的 = 符号依赖子节点 + 符号节点返回的值，+ 符号依赖 x 节点和 *符号节点的值，依此递归下去，这样最开始获取到值的就是最低一级的运算，在这个例子里就是 y 和 z 的运算的结果返回给* 符号节点。这种就是典型的后序遍历。CRuby 1.9 之前也是用的这种方式解释执行的。

赋值符号 = 符号的左侧叫做左值，右侧的值叫做右值。左值也可能是复杂的表达式，比如数组或者结构体。根据求值顺序，对于二元运算的节点，是先遍历左子节点的。所以当左值是复杂表达式需要计算时是会优先进行计算的。再看个左值是数组的例子：

```
1  public class LeftFirstTest {
2      public static void main(String[]
3  args) {
4          int[] arr = new int[1];
5          int x = 3;
6          int y = 5;
7          arr[0] = x + y;
8      }
   }
```

arr[0] = x + y; 对应的字节码：

```
1  // 左值，数组下标
2  aload_1
3  iconst_0
4
5  // 右值
6  iload_2 // x
7  iload_3 // y
8  iadd
9
1  iastore  // 赋值符号节点进
0  行赋值
```

可以看到左值是先计算的。

从树遍历到基于栈的解释，实际上是一个将 AST 树打平的过程。方法是后序遍历 AST 时使用 Reverse Polish Notation 这种后缀记法生成一个序列，成为一个线性结构，后面再解释执行这个操作序列。

JVM 是 Java 语言的虚机。Android 也是用的 Java 语言，那么它的虚机也是 JVM 吗？答案是否定的，Android 使用的虚机叫 Dalvik VM)，这款虚机在很多设计上都与 JVM 兼容，字节码是二地址和三地址并用的方式，是基于寄存器的架构。Dalvik VM 用在移动端为了能够更加高效，开始就没有顾及太多可移植性，这样基于寄存器架构的优势就能够更好的发挥出来了。想更多了解 Dalvik VM 可以通过 Dan Bornstein 做的一个 Dalvik 的实现原理的演讲 Dalvik VM Internals。

JVM 和 Dalvik VM 的主要区别是后者字节码指令数量和内存更少。JVM 每个线程有一个 Java 栈用来记录方法调用的 activation record，每调用一个方法就会分配一个新栈帧，方法返回就 Pop 出

栈帧。每个栈帧会有局部变量区，istore 这样的指令用来移动局部变量和参数到局部变量区。每个栈帧还会有求值栈，这个栈用来存储求值的中间结果和调用其他方法的参数等，使用 iconst 这样的指令来进行数据的移动，还可以通过 iadd，imul 这样的指令在求值栈中 Pop 出值进行求值，然后再把结果 Push 到栈里。

Dalvik VM 的每个线程有程序计数器和调用栈，方法的调用和 JVM 一样是会分配一个新的帧，不同的是 Dalvik VM 使用的是虚拟寄存器来替代 JVM 里的局部变量区和求值栈。方法调用会有一组自己的虚拟寄存器，常用的是 v0 - v15，有些指令可以使用 v0 - v255。只在虚拟寄存器中进行指令操作，数据移动少多了，保存局部变量的存储单元也会少很多。Dalvik VM 的寄存器每次方法调用会一组自己的，不过在 X86 架构中寄存器是全局的，这样 X86 需要考虑 calling converntion，就是需要保护一些寄存器的状态，在调用时需要处理这些，而 Dalvik VM 调用完方法后，那些寄存器值会恢复成调用前的可以很好的避免这样的问题。

V8 没有中间的字节码，而是直接编译 JavaScript 生成机器码。不过在内部也用了表达式栈来简化代码生成，在编译过程中使用虚拟栈帧来记录局部变量和栈的状态。生成代码的过程中会有窥孔优化用来去除多余的入栈和出栈，把栈操作转成寄存器操作，生成的代码看起来就和基于寄存器的代码类似了。

这就有个疑问了，零地址要做多次入栈和出栈操作，执行效率低，那么为什么还会有虚拟机比如 JVM 字节码还要用零地址形式呢？

像 X86 刚开始时的寄存器很多不是通用寄存器，由于让编译器去决定程序里那么多的变量该怎么装到寄存器里，那些应该应该映射到一个寄存器，哪些应该换出，这并不容易，JVM 采用零指令这种堆栈结构的原因就是不信任编译器的寄存器分配，所以使用堆栈结构，这样就可以避开寄存器分配这样的难题。不过后来 IBM 公开了他们的图染色寄存器分配算法，这样编译器的分配能力得到了很大的进步，所以现在都是编译器来主导寄存器分配。

很多主流高级语言虚拟机（high-level language virtual machine，简称 HLL VM）比如 JVM，CPython 都采用的基于栈的架构。这样做主要是因为不用考虑临时变量分配空间，只需要求值栈来做，这样的编译器更容易实现。还有就是可以更容易在硬件较差的机器上运行，前面也讲到基于栈这种架构指令对于存储空间的要求更少。最后就是考虑移植，复杂指令计算机（Complex Instruction Set Computer，简称 CISC）的通用寄存器比较少，32位只有8个32位通用寄存器。精简指令集计算机（Reduced Instruction Set Computer，简称 RISC）通用寄存器数量会多些，32位有16个寄存器。源架构寄存器数量通常和实际机器通用寄存器数量不一致而实现映射麻烦，基于栈架构里是没有通用寄存器的，实现虚机时可以就可以很容易自由的分配实际机器寄存器，移植起来自然也就容易多了。

处理器来说多是基于寄存器架构的，但是对于虚机来说，由于基于栈架构需要执行更多的 load 或 store 这样的指令，这样 Instruction dispatch 的次数和内存访问的次数会更多，所以基于寄存器的架构在虚机里同样也更优些，这也是为什么对于闭环的苹果来说会在 JavaScriptCore 的虚机里采用性能更好的基于寄存器的架构，而不去顾及移植性。

各大引擎的介绍

各个 JavaScript 引擎介绍：

- SpiderMonkey：用于 Mozilla Firefox，是最早的 JavaScript 引擎，基于栈的字节码。Parser 使用的手写纯递归下降式。
- KJS：KDE 的引擎，用于 Konqueror 浏览器。树遍历解释器。Parser 使用的是 bison。
- Rhino：使用 Java编写，开发源码，也是 Mozilla 的，相当于 Java 版的 SpiderMonkey，当时的想法是想把 JavaScript 做成服务端的脚本语言来用。Parser 是手写的纯递归下降式 rhino/src/org/mozilla/javascript/Parser.java 。
- Chakra：也叫 JScript 微软的 Internet Explorer 和 Microsoft Edge 在使用。
- JavaScriptCore：苹果开发的开源 JavaScript 引擎，用在 Safari 等浏览器中。Safari 是带有 JIT 的 JavaScriptCore 的，而 UIWebView 是没有的。
- V8：Google 开发的开源引擎，用于 Chrome。Parser 使用的是手写纯递归下降加运算符优先级混合式。V8 使用了 2-pass，会先收集一些上下文信息增加预测的准确性。V8 开始时是不用字节码会直接编译成机器码，Dart VM 也是这样设计的，他们一个问题的回答 Why didn't Google build a bytecode VM targetable by multiple languages including Dart? 同样适用于 V8。不过 V8 5.9 启用了 Ignition 字节码解释器，自此几大 JS 引擎都用了字节码。启用字节码的考虑主要是希望能够减少机器码对内存空间的占用。由于机器码占用的空间也很大，所以不好都缓存下来，不然内存和磁盘都吃不消，序列化和反序列化时间都太长，这样每次编译的机器码都不是完整的只会缓存最外的一层，如果代码最外层包了一层，启动代码每次都是不同的调用就会每次都编译，会导致缓存没有作用。由于字节码通过比较好的设计后能够做到比机器码紧凑，这样引入 Ignition 后内存明显的下降了。TurboFan 再对 Ignition 字节码来进行解释。

各个引擎之间通用技术

根据下面关键子去查找，可以了解更多的相关的虚机技术。

源码到中间代码

- Recusive-descent parser：递归下降式 parser
- Operator precedence parser：运算符优先级 parser
- Deferred parser：延迟 parser

中间代码到目标代码

- Tiered compilation：**多层编译**
- Background compilation：**后台编译**
- Type feedback：**类型反馈**
- Type specialization：**类型特化**
- SSA-form IR：**静态单赋值形式**