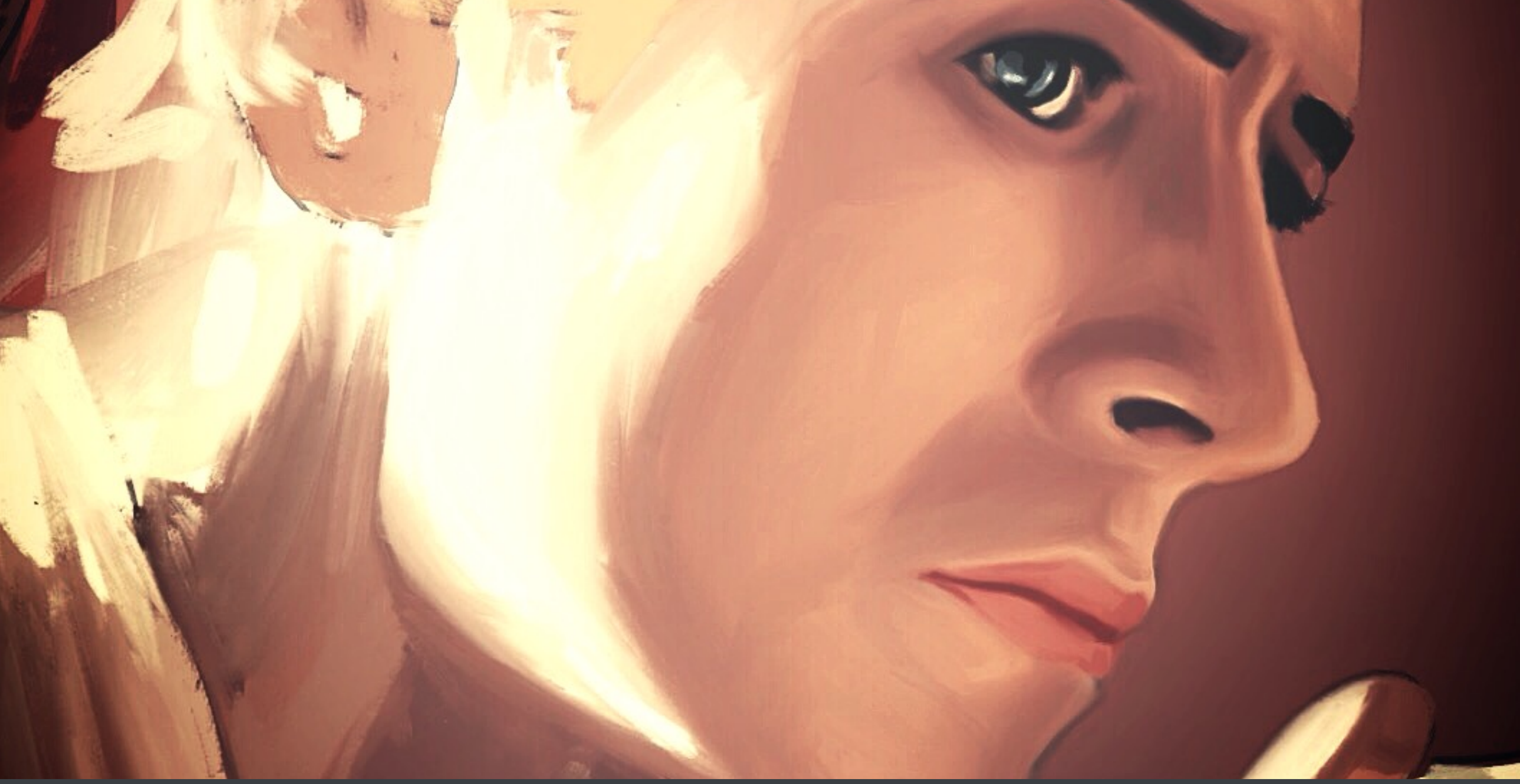
学习 iOS 编译
能做哪些有意思的事情

— 戴铭

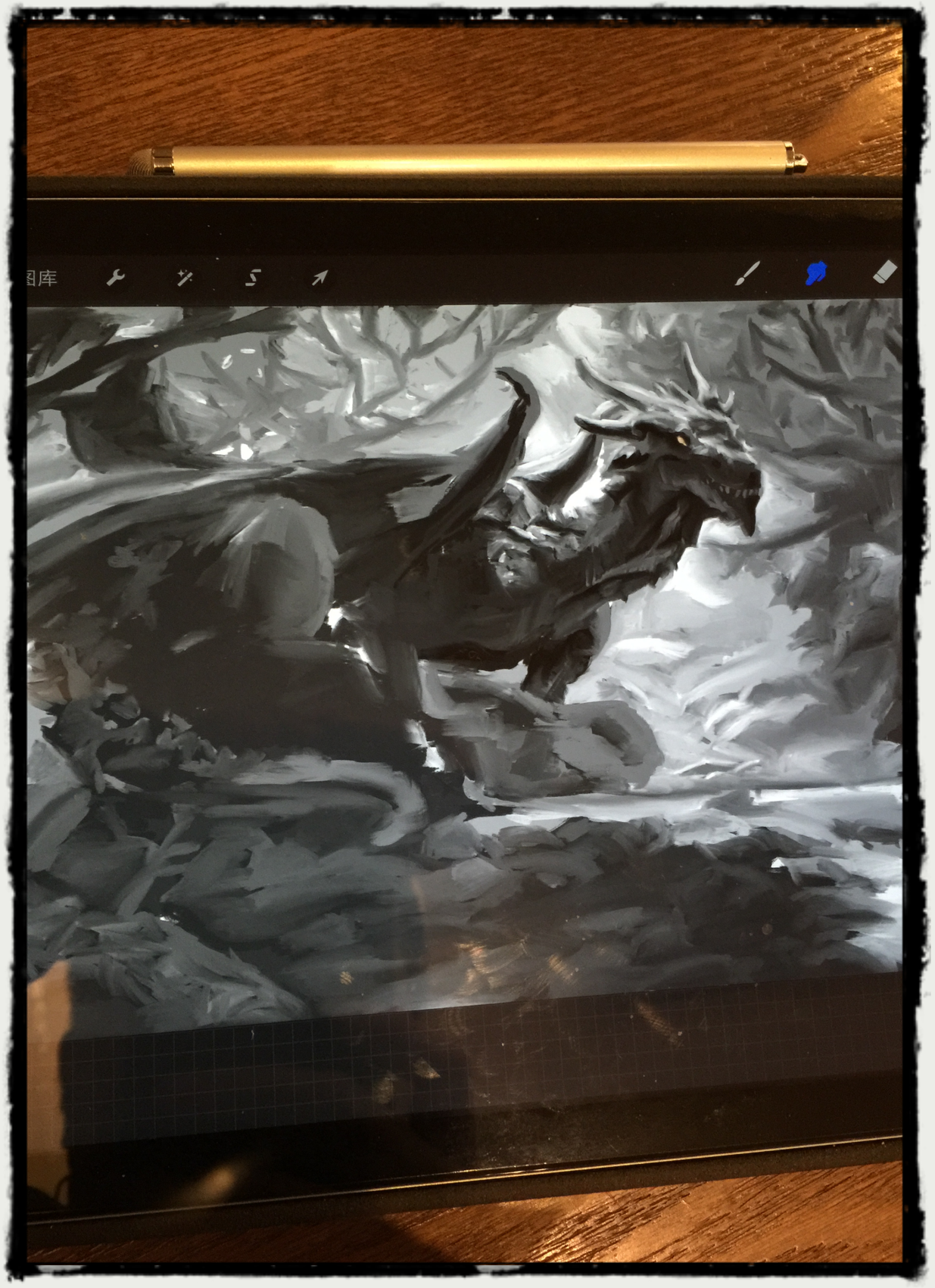


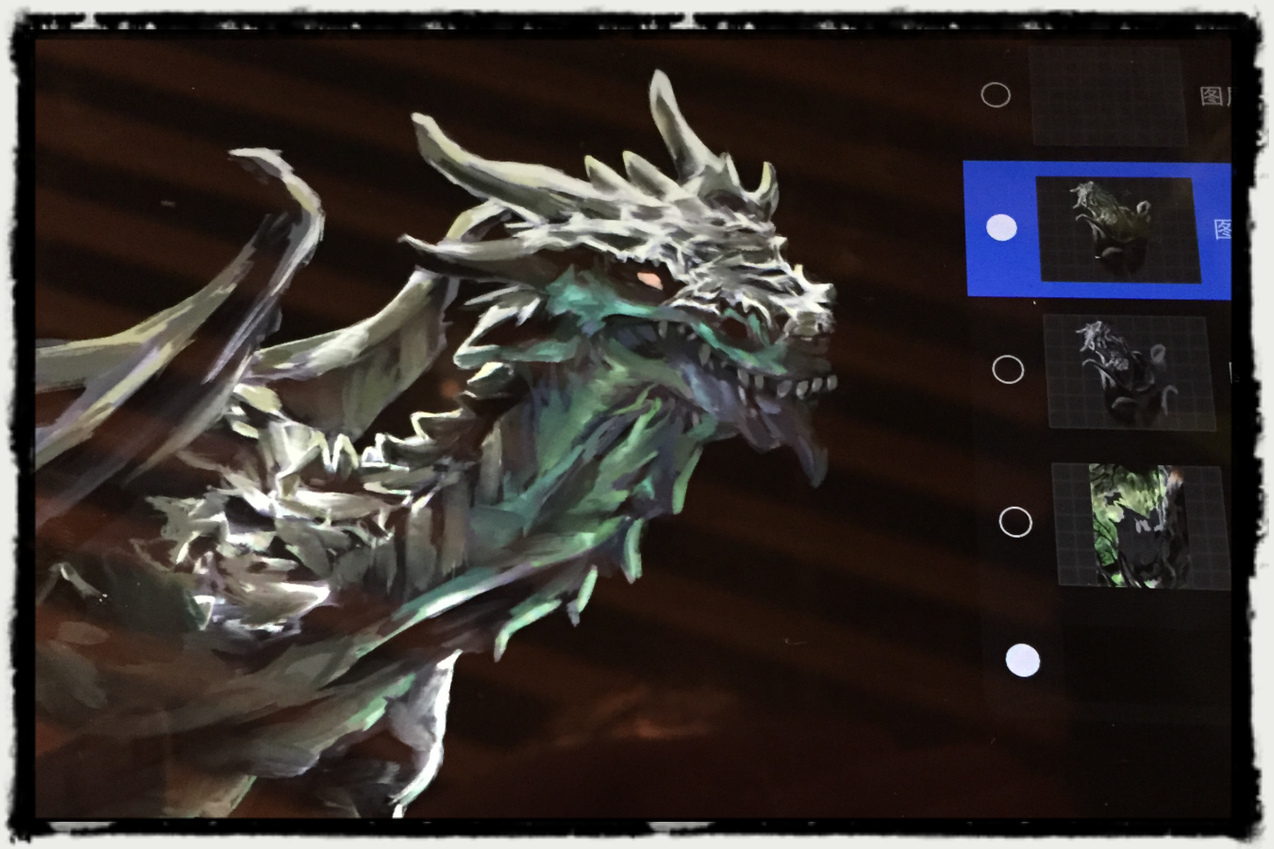
自我介绍

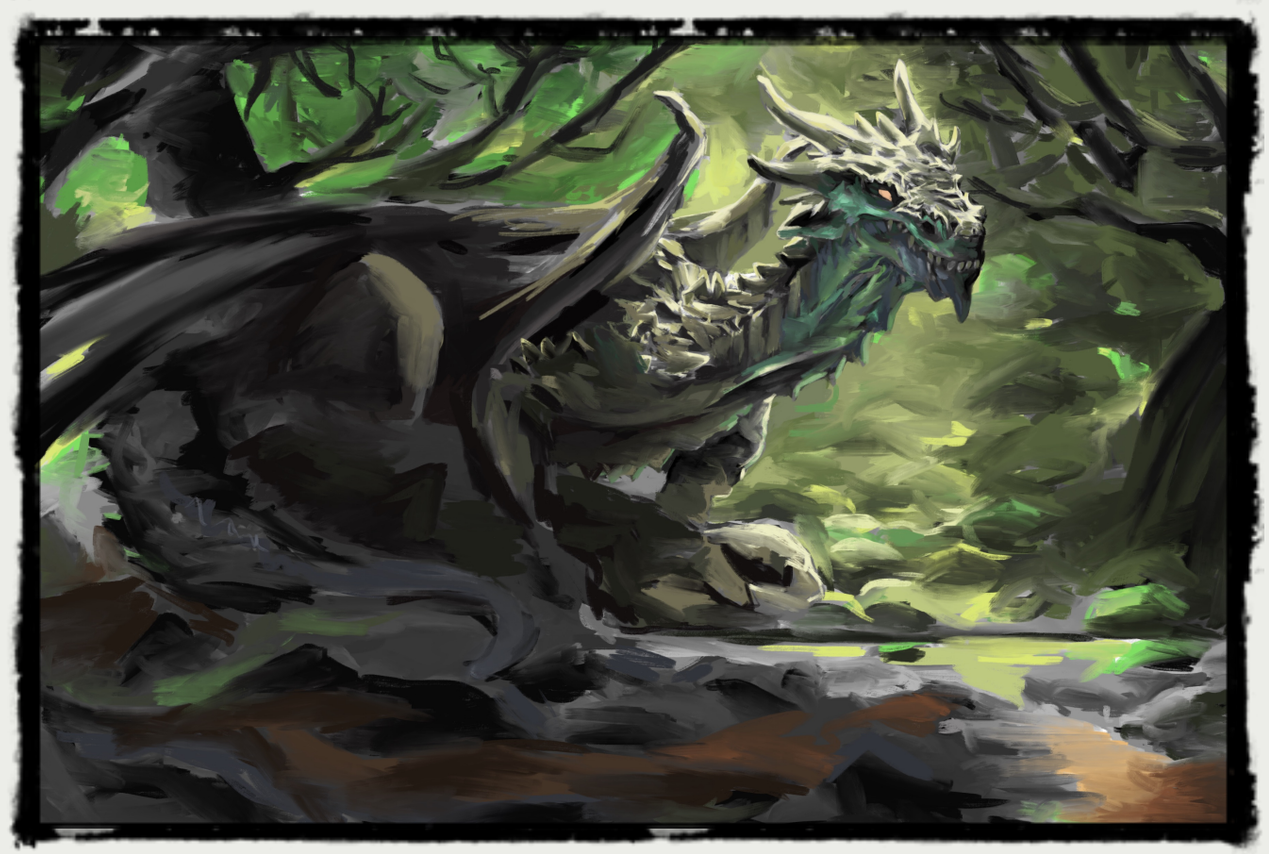
微博： @戴铭

Github：https://github.com/ming1016

在画画和开发的切换过程中
发现了很多相通的东西


图库





龙母

掌握结构原理和光影原理能够使画更加立体和真实

Tywin Lannister

那么编译
编程语言的原理是什么呢?

看个 DEMO

先写个一个基于 FlexBox 布局的 H5

```
<style>
  #six {
    /*order: -1; 排列顺序，数值越小，排列越靠前，默认为0*/
    flex-grow: 0; /*放大比例，默认为0*/
    flex-basis: 150pt; /*占据的主轴空间 <length> | auto default auto */
    align-self: flex-end; /*单个项目有与其他项目不一样的对齐方式 auto | flex-start | flex-end | center | baseline | stretch*/
  }
  .big {
    width: 150pt; height: 200pt; background-color: black; margin: 10pt; color: white;
  }
  .small {
    width: 100pt;
    height: 40pt;
    background-color: orange;
    margin: 10pt;
    color: white;
    display: flex;
    flex-direction: row;
  }
  .tinyBox {
    width: 10pt; height: 10pt; background-color: red; margin: 5pt;
  }
  #main {
    display: flex;
    flex-direction: row; /*垂直横向排列顺序 row | row-reverse | column | column-reverse*/
    flex-wrap: wrap; /*nowrap | wrap | wrap-reverse*/
    flex-flow: row wrap; /*简化写法 <flex-direction> || <flex-wrap>*/
    justify-content: flex-start; /*主轴对齐方式 flex-start | flex-end | center | space-between | space-around*/
    align-items: center; /*交叉轴对齐方式 flex-start | flex-end | center | baseline | stretch*/
    align-content: flex-start; /*多轴对齐方式 flex-start | flex-end | center | space-between | space-around | stretch*/
  }
  ul {
    display: flex;
    flex-direction: column;
    justify-content: center;
    list-style-type: none;
    align-items: center;
    flex-wrap: wrap;
    -webkit-padding-start: 0pt;
  }
</style>
```

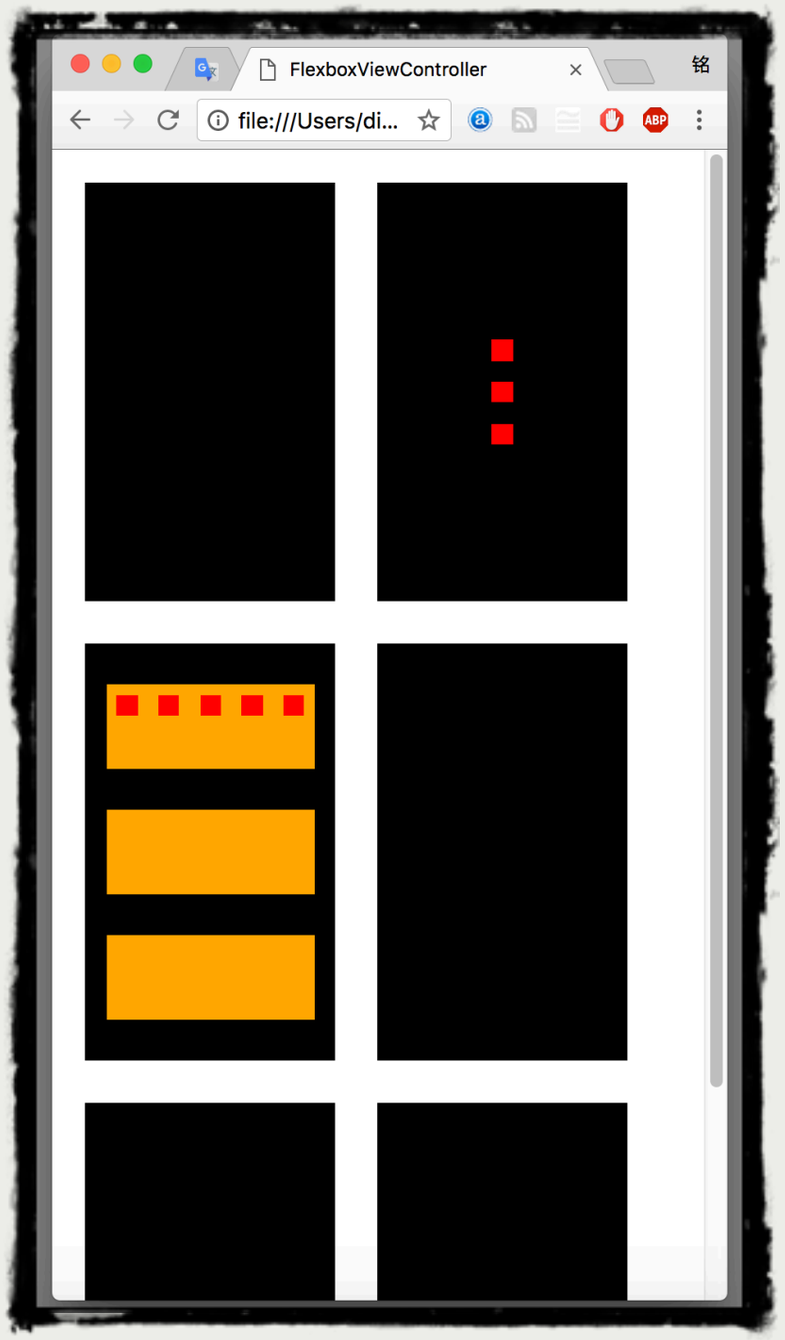
```
<body>
  <div id="main">
    <div class="big" id="smallbox"></div>
    <ul class="big" id="more">
      <li class="tinyBox"></li>
      <li class="tinyBox"></li>
      <li class="tinyBox"></li>
    </ul>
    <ul class="big" id="more">
      <li class="small">
        <div class="tinyBox"></div>
        <div class="tinyBox"></div>
        <div class="tinyBox"></div>
        <div class="tinyBox"></div>
      </li>
      <li class="small"></li>
      <li class="small"></li>
    </ul>
    <div class="big" id="six"></div>
    <div class="big"></div>
    <div class="big"></div>
  </div>
</body>
```

为什么选 FlexBox 布局

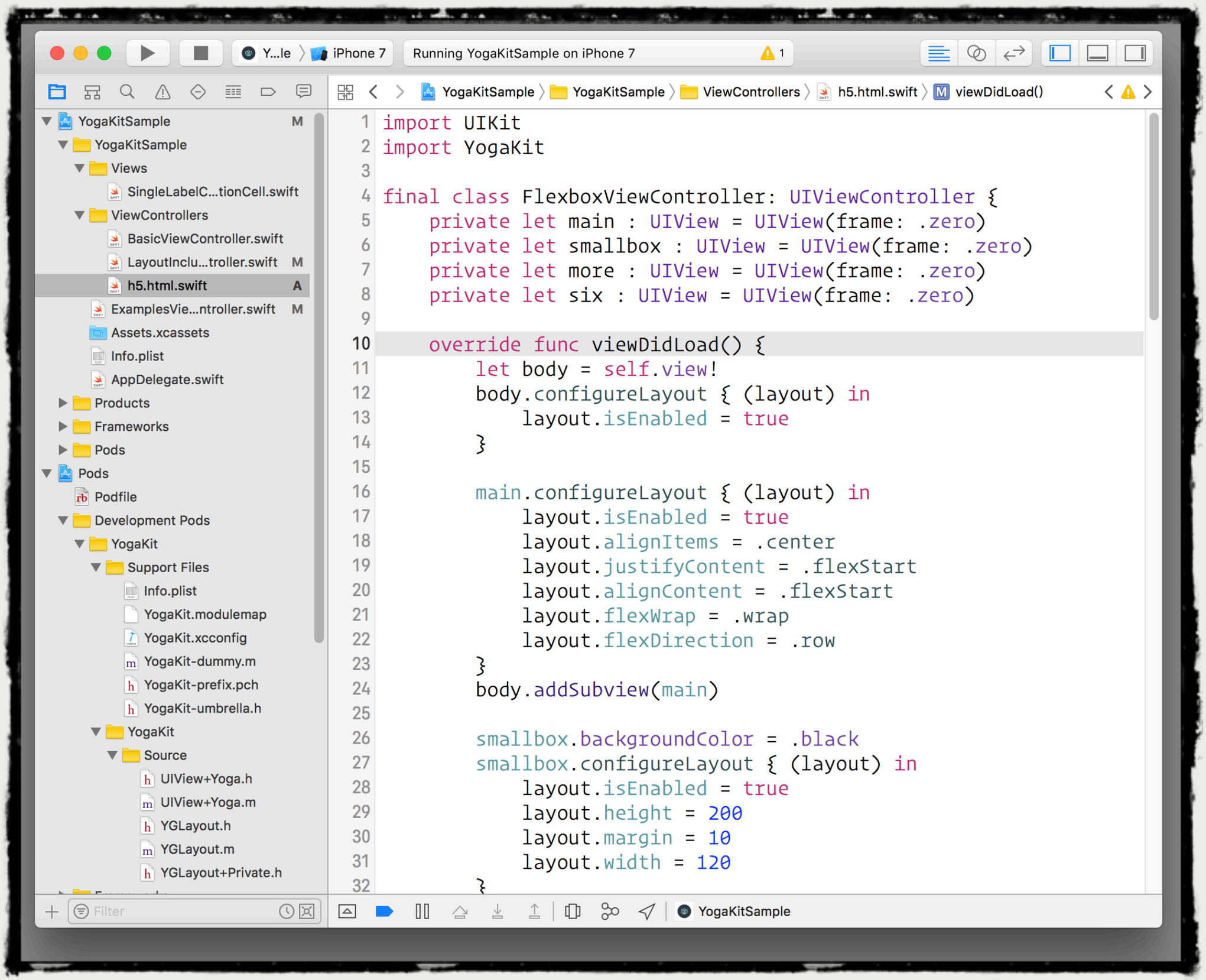
W3C 标准 https://www.w3.org/TR/css-flexbox-1/

运用于 React Native 和 Weex 里

各个平台都有相关实现的库，iOS 有 Facebook 的 Yoga，Android 有 Google 的 FlexboxLayout

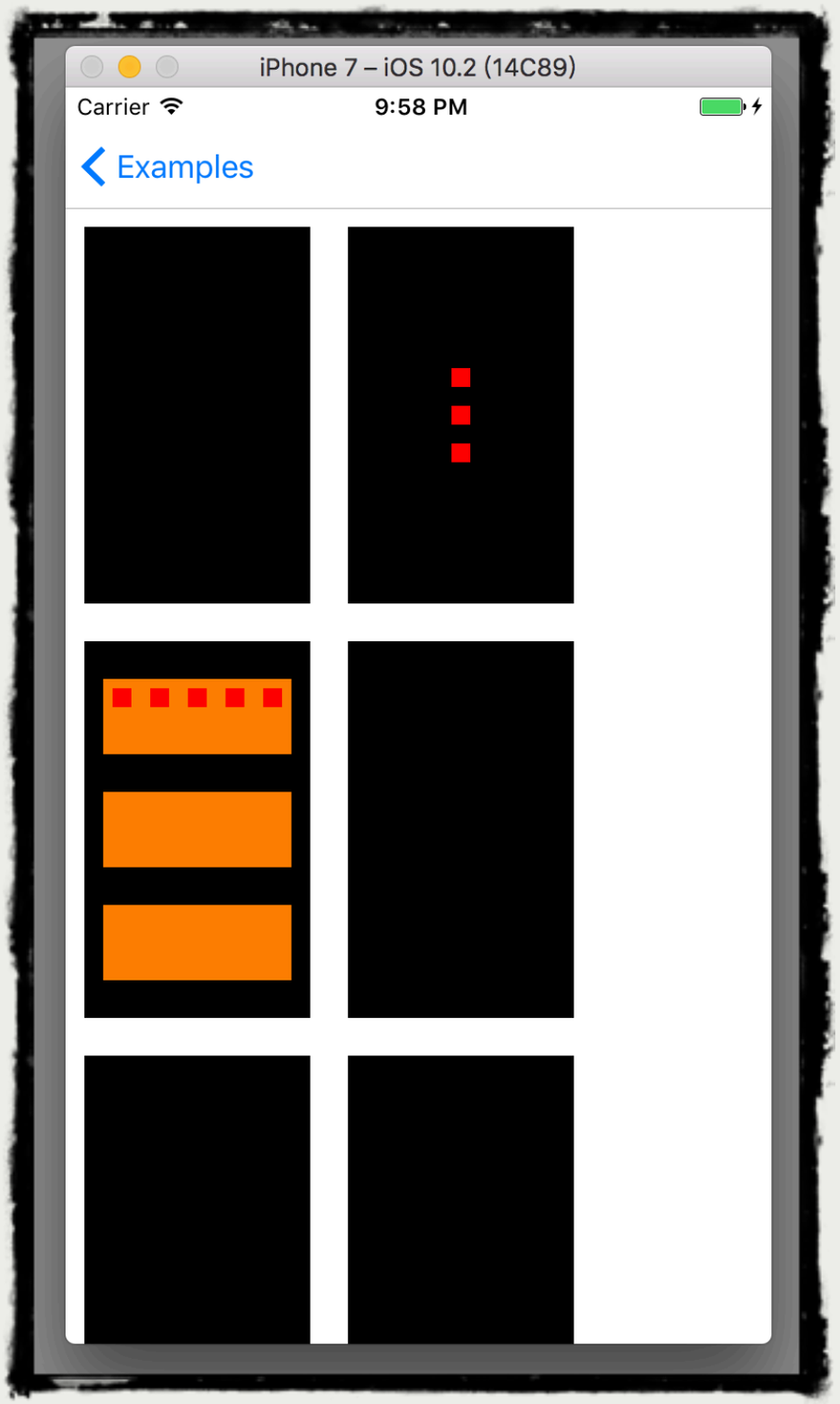


H5 自动转成 Swift 语言




```swift
import UIKit
import YogaKit

final class FlexboxViewController: UIViewController {
    private let main : UIView = UIView(frame: .zero)
    private let smallbox : UIView = UIView(frame: .zero)
    private let more : UIView = UIView(frame: .zero)
    private let six : UIView = UIView(frame: .zero)

    override func viewDidLoad() {
        let body = self.view!
        body.configureLayout { (layout) in
            layout.isEnabled = true
        }

        main.configureLayout { (layout) in
            layout.isEnabled = true
            layout.alignItems = .center
            layout.justifyContent = .flexStart
            layout.alignContent = .flexStart
            layout.flexWrap = .wrap
            layout.flexDirection = .row
        }
        body.addSubview(main)

        smallbox.backgroundColor = .black
        smallbox.configureLayout { (layout) in
            layout.isEnabled = true
            layout.height = 200
            layout.margin = 10
            layout.width = 120
        }
```

Run 起来看看效果

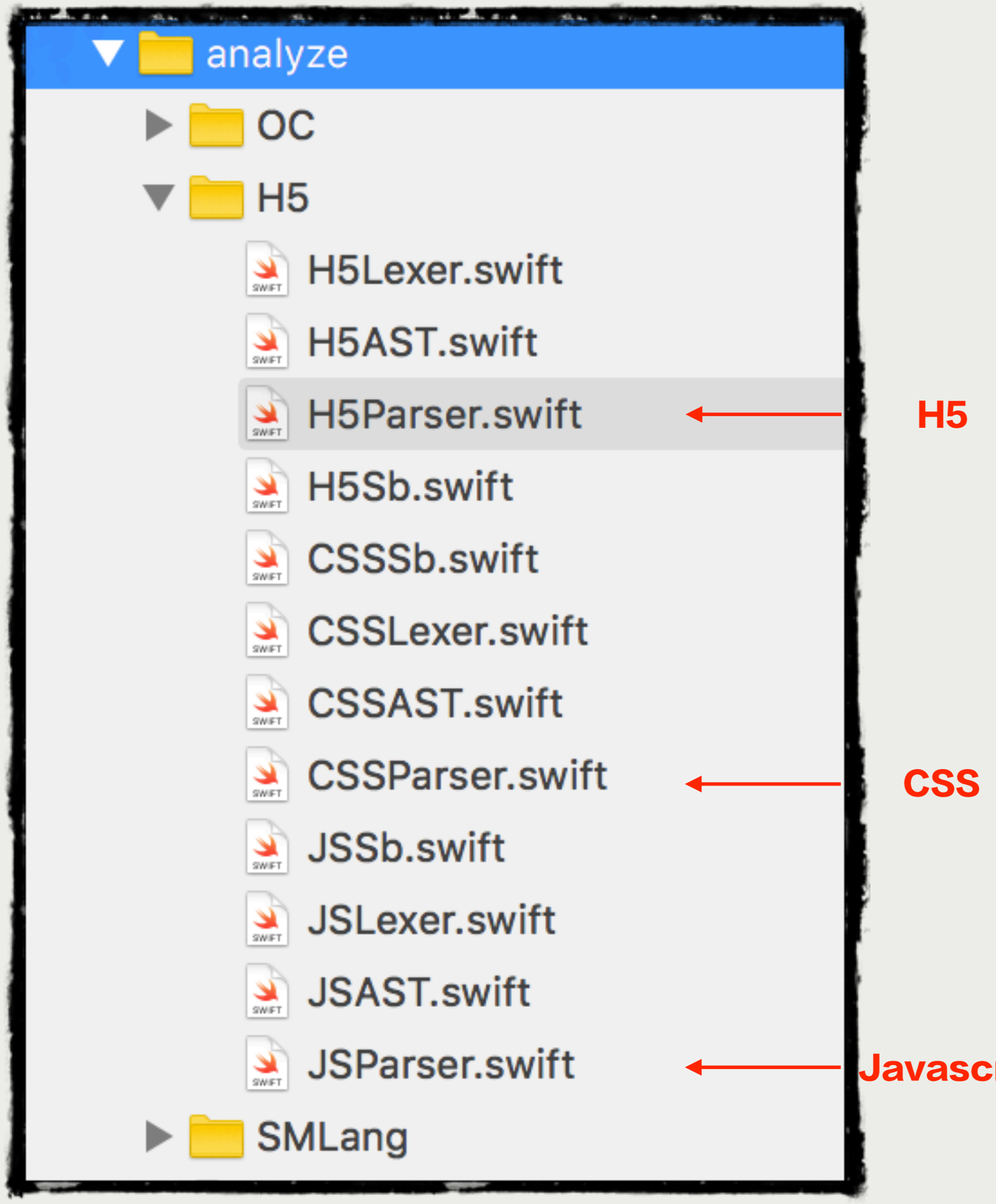


怎么做到的?

是不是以后只用写 H5 了?

编译前端

Lexer
AST
Parser



Backus normal form (BNF) for EcmaScript
JavaScript 巴科斯范式

BNF for EcmaScript.jj

NON-TERMINALS

```
PrimaryExpression::="this"
                  | ObjectLiteral
                  | ( "(" Expression ")" )
                  | Identifier
                  | ArrayLiteral
                  | Literal
        Literal::=( <DECIMAL_LITERAL> | <HEX_INTEGER_LITERAL> | <STRING_LITERAL> | <BOOLEAN_LITERAL> |
                    <NULL_LITERAL> | <REGULAR_EXPRESSION_LITERAL> )
     Identifier::=<IDENTIFIER_NAME>
   ArrayLiteral::="[" ( ( Elision )? "]" | ElementList Elision "]" | ( ElementList )? "]" )
   ElementList::=( Elision )? AssignmentExpression ( Elision AssignmentExpression )*
      Elision::=( "," )+
ObjectLiteral::="{" ( PropertyNameAndValueList )? "}"
PropertyNameAndValueList::=PropertyNameAndValue ( "," PropertyNameAndValue | "," )*
PropertyNameAndValue::=PropertyName ":" AssignmentExpression
    PropertyName::=Identifier
                  | <STRING_LITERAL>
                  | <DECIMAL_LITERAL>
MemberExpression::=( ( FunctionExpression | PrimaryExpression ) ( MemberExpressionPart )* )
                  | AllocationExpression
MemberExpressionForIn::=( ( FunctionExpression | PrimaryExpression ) ( MemberExpressionPart )* )
AllocationExpression::=( "new" MemberExpression ( ( Arguments ( MemberExpressionPart )* )* ) )
MemberExpressionPart::=( "[" Expression "]" )
                  | ( "." Identifier )
  CallExpression::=MemberExpression Arguments ( CallExpressionPart )*
CallExpressionForIn::=MemberExpressionForIn Arguments ( CallExpressionPart )*
CallExpressionPart::=Arguments
                  | ( "[" Expression "]" )
                  | ( "." Identifier )
     Arguments::="(" ( ArgumentList )? ")"
  ArgumentList::=AssignmentExpression ( "," AssignmentExpression )*
LeftHandSideExpression::=CallExpression
                  | MemberExpression
eftHandSideExpressionForIn::=CallExpressionForIn
                  | MemberExpressionForIn
 PostfixExpression::=LeftHandSideExpression ( PostfixOperator )?
  PostfixOperator::=( "++" | "--" )
  UnaryExpression::=( PostfixExpression | ( UnaryOperator UnaryExpression )+ )
   UnaryOperator::=( "delete" | "void" | "typeof" | "++" | "--" | "+" | "-" | "~" | "!" )
MultiplicativeExpression::=UnaryExpression ( MultiplicativeOperator UnaryExpression )*
```

JSAllocationExpression

JSFunctionDeclaration

JSIfStatement

VUE.JS

```
22  <script type="text/javascript">
23    var watchExampleVM = new Vue({
24      el: '#watch-example',
25      data: {
26        question: '',
27        answer: 'I cannot give you an answer until you ask a question!'
28      },
29      watch: {
30        question: function (newQuestion) {
31          this.answer = 'Waiting for you to stop typing...'
32          this.getAnswer()
33        }
34      },
35      methods: {
36        getAnswer: _.debounce(
37          function () {
38            if (this.question.indexOf('?') === -1) {
39              this.answer = 'Questions usually contain a question mark. ;-)'
40              return
41            }
42            this.answer = 'Thinking...'
43            var vm = this
44            axios.get('https://yesno.wtf/api')
45              .then(function (response) {
46                vm.answer = _.capitalize(response.data.answer)
47              })
48              .catch(function (error) {
49                vm.answer = 'Error! Could not reach the API. ' + error
50              })
51          },
52          500
53        )
54      }
55    })
56  </script>
```

为什么要做这个

iOS 上有自动布局

Android 上有可组合容器

Web 上有 CSS

每个平台都有不同布局思路，这样开发团队为了界面布局就需要花费大量的时间去解决，而且布局解决方案也难以统一。

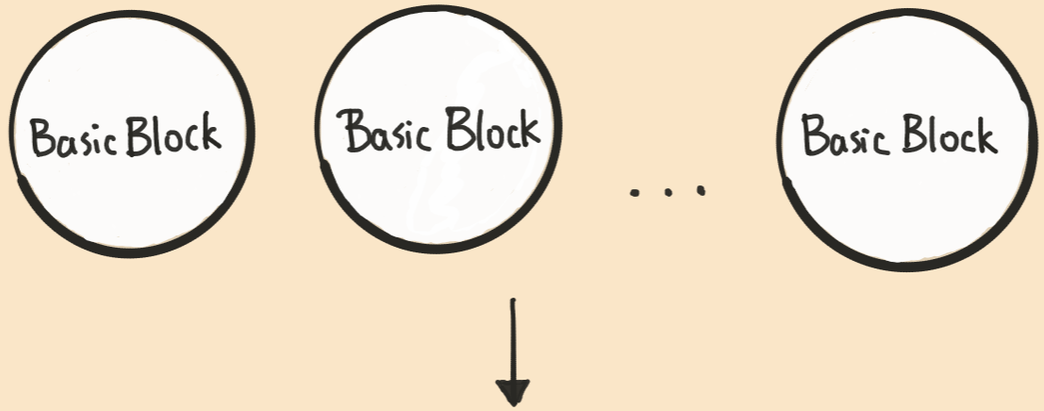
能够规范代码，保障多端代码业务逻辑一致

SWIFT WRITE LLVM IR RUN IN JIT DEMO

using LLVM_C
convert any language to bitcode running in JIT

Module

Function



Basic Block

Instruction Instruction ... Instruction

ll.swift

```swift
import LLVM_C

let module = LLVMModuleCreateWithName("Hello")

let int32 = LLVMInt32Type()
let returnType = int32
var paramTypes: UnsafeMutablePointer<LLVMTypeRef?> = UnsafeMutablePointer.allocate(capacity:2)
paramTypes.initialize(from:[int32, int32])

let functionType = LLVMFunctionType(returnType, paramTypes, 2, 0)
let sumFunction = LLVMAddFunction(module, "sum", functionType)

let builder = LLVMCreateBuilder()
let entryBlock = LLVMAppendBasicBlock(sumFunction, "entry")
LLVMPositionBuilderAtEnd(builder, entryBlock)

let a = LLVMGetParam(sumFunction, 0)
let b = LLVMGetParam(sumFunction, 1)

let result = LLVMBuildAdd(builder, a, b, "entry")
LLVMBuildRet(builder, result)

let engineSize = MemoryLayout<LLVMExecutionEngineRef?>.stride
let engine = UnsafeMutablePointer<LLVMExecutionEngineRef?>.allocate(capacity:engineSize)

let errorSize = MemoryLayout<UnsafeMutablePointer<Int8>?>.stride
let error = UnsafeMutablePointer<UnsafeMutablePointer<Int8>?>.allocate(capacity:errorSize)

LLVMLinkInMCJIT()
LLVMInitializeNativeTarget()
LLVMInitializeNativeAsmPrinter()

let res = LLVMCreateExecutionEngineForModule(engine,module,error)
if res != 0 {
    let msg = String(cString:error.pointee!)
    print("\(msg)")
    exit(1)
}
```

Module

Sum function

JIT

```
41    func runSumFunction(a: Int, _ b: Int) -> Int {
42        let returnType = int32
43        let functionType = LLVMFunctionType(returnType, nil, 0, 0)
44        let wrapperFunction = LLVMAddFunction(module, "", functionType)
45        let entryBlock = LLVMAppendBasicBlock(wrapperFunction, "entry")
46        LLVMPositionBuilderAtEnd(builder, entryBlock)
47        let argumentsSize = MemoryLayout<LLVMValueRef>.stride * 2
48        let arguments = UnsafeMutablePointer<LLVMValueRef?>.allocate(capacity:argumentsSize)
49
50        let argA = LLVMConstInt(int32, UInt64(a), 0)
51        let argB = LLVMConstInt(int32, UInt64(b), 0)
52        arguments.initialize(from:[argA, argB])
53        let callTemp = LLVMBuildCall(builder, sumFunction, arguments, 2, "sum_temp")
54        LLVMBuildRet(builder, callTemp)
55
56        let value = LLVMRunFunction(engine.pointee, wrapperFunction, 0, nil)
57        let result  = LLVMGenericValueToInt(value, 0)
58
59        LLVMDumpModule(module)
60    //    LLVMDisposeModule(module)
61
62        return Int(result)
63    }
64    LLVMWriteBitcodeToFile(module, "ll.bc")
65
66    print("\(runSumFunction(a:15, 6))")
67
```

← Add run function

← Module dump can be excution

运行

xcrun -sdk macosx swiftc ll.swift -I /Users/didi/Documents/llvm/trunk/include/ -I /Users/didi/Documents/llvm/trunk/build/include -Xcc -D__STDC_CONSTANT_MACROS -Xcc -D__STDC_LIMIT_MACROS `/Users/didi/Documents/llvm/trunk/build/bin/llvm-config --libs mcjit native` -L /Users/didi/

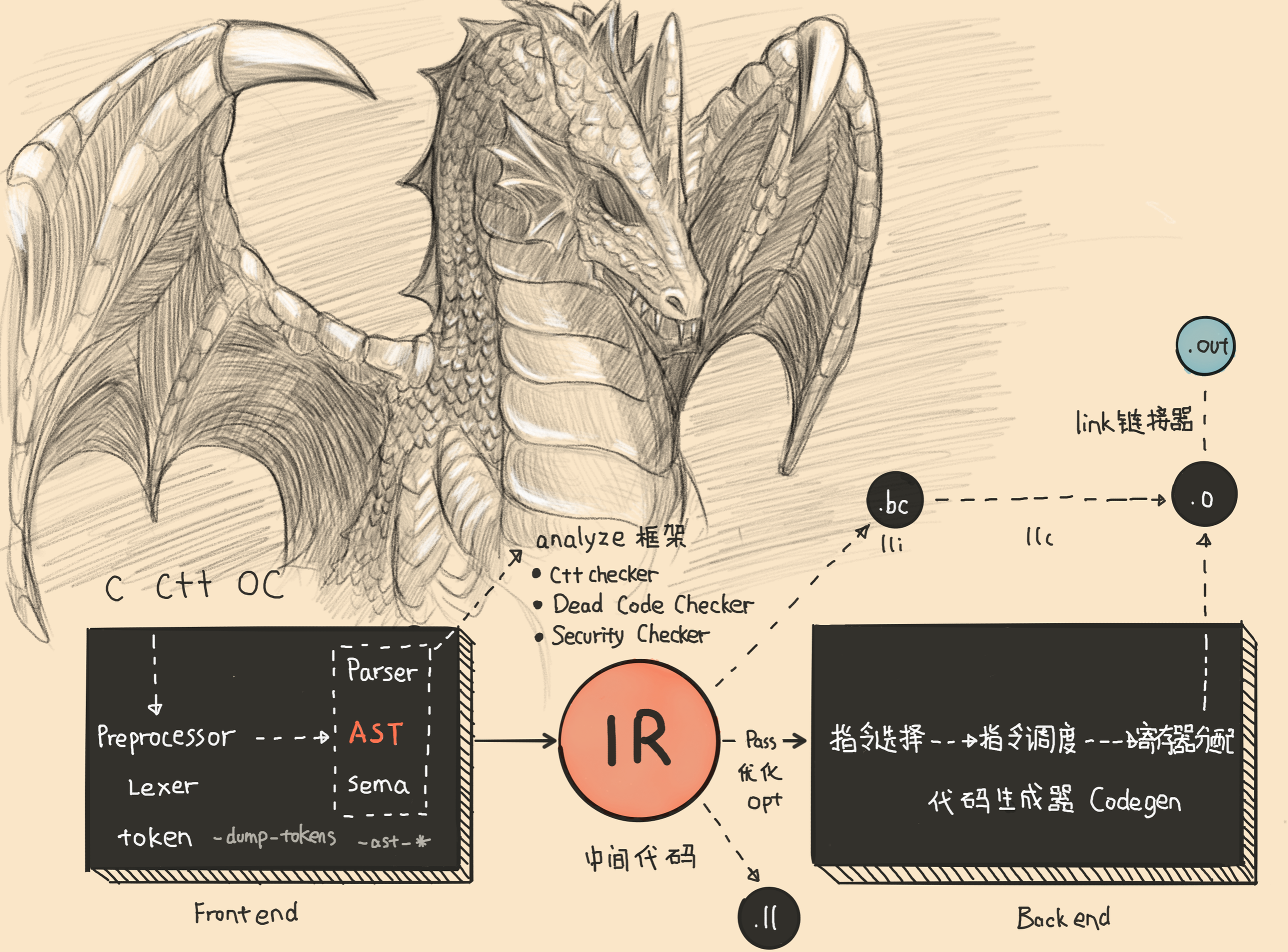
```
➜ llvmswift xcrun -sdk macosx swiftc ll.swift -I /Users/didi/Documents/llvm/trunk/include/ -
I /Users/didi/Documents/llvm/trunk/build/include -Xcc -D__STDC_CONSTANT_MACROS -Xcc -D__STDC_
LIMIT_MACROS `/Users/didi/Documents/llvm/trunk/build/bin/llvm-config --libs mcjit native` -L
/Users/didi/Documents/llvm/trunk/build/lib -lc++ -lncurses
➜ llvmswift ./ll
; ModuleID = 'Hello'
source_filename = "Hello"
target datalayout = "e-m:o-i64:64-f80:128-n8:16:32:64-S128"

define i32 @sum(i32, i32) {
entry:
  %entry1 = add i32 %0, %1
  ret i32 %entry1
}

define i32 @0() {
entry:
  %sum_temp = call i32 @sum(i32 15, i32 6)
  ret i32 %sum_temp
}

; Function Attrs: nounwind
declare void @llvm.stackprotector(i8*, i8**) #0

attributes #0 = { nounwind }
21
➜ llvmswift
```

Sum function

Run function

Print result

LLVM BACKEND


.out
link 链接器
.bc
.o
llvm
llc
llvm
analyze 框架
• Ctt checker
• Dead Code Checker
• Security Checker
C C++ OC
Parser
Preprocessor
AST
Lexer
Sema
token
-dump-tokens
-ast-*
IR
Pass
优化
opt
中间代码
指令选择 --> 指令调度 --> 寄存器分配
代码生成器 Codegen
.ll
Front end
Back end

Pipeline

LLVM IR

IR Pass

ISel

DAG
Combine2

Legalize

Lower

DAG
Combine1

Instruction
Scheduling/Post-RA

Register
Allocation

Instruction
Scheduling/Pre-RA

MC Stream

Object File
Assembler
BinaryCode

CodeGen 阶段

Instruction Selection 指令选择

将IR转化成目标平台指令组成的定向非循环图 DAG（Directed Acyclic Graph）

Scheduling and Formation 调度与排序

读取 DAG，将 DAG 的指令排成 MachineInstr 的队列

SSA 优化

多个基于 SSA（Static Single Assignment）的 Pass 组成。比如 modulo-scheduling 和 peephole optimization 都是在这个阶段完成的

Register allocation 寄存器分配

将 Virtual Register 映射到 Physical Register 或内存上

Prolog / Epilog 生成

确定所需堆栈大小

Machine Code 晚期优化

最后一次优化机会

Code Emission

输出代码，可以选择汇编或者二进制机器码。

SelectionDAG

构建最初的 DAG

把 IR 里的 add 指令转成 SelectionDAG 的 add 节点

优化构建好的 DAG

把一些平台支持的 meta instructions 比如 Rotates，div / rem 指令识别出

Legalization SelectionDAG 类型

比如某些平台只有64位浮点和32位整数运算指令，那么就需要把所有 f32 都提升到 f64，i1/i8/i16 都提升到 i32，同时还要把 i64 拆分成两个 i32 来存储，操作符的合法化，比如 SDIV 在 x86 上回转成 SDIVREM。这个过程结果可以通过 llc -view-dag-combine2-dags sum.ll 看到

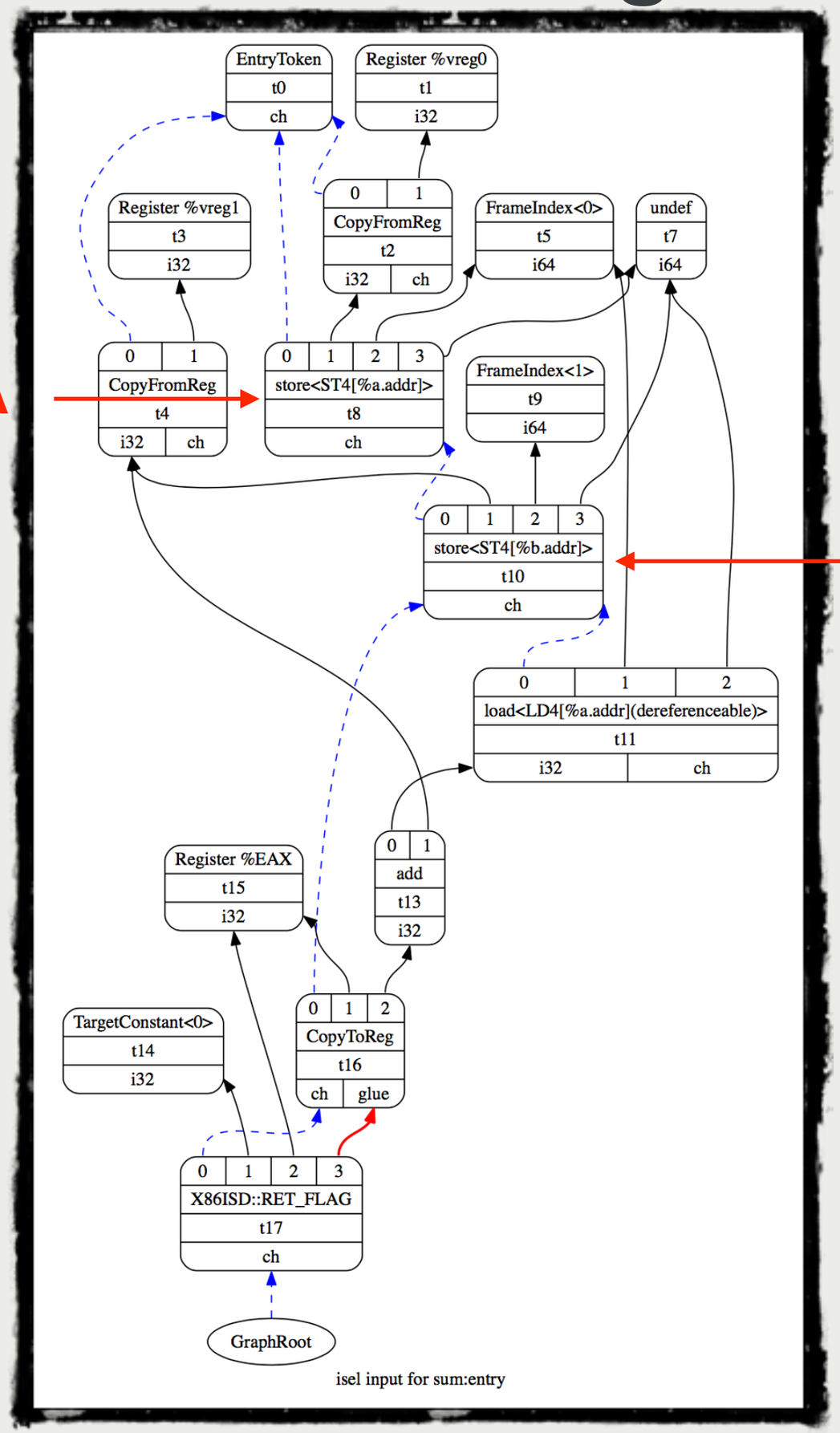
指令选择 instruction selector（ISel）

将平台无关的 DAG 通过 TableGen 读入 .tb 文件并且生成对应的模式匹配代码从而转成平台相关的 DAG

SelectionDAG Scheduling and Formation

因为 CPU 是没法执行 DAG，所以需要将指令从 DAG 中提取依据一定规则比如 minimal register pressure 或隐藏指令延迟重排成指令队列。（DAG -> linear list（SSA form）-> MachineInstr -> MC Layer API MCInst MCStreamr -> MCCodeEmitter -> Binary Instr）

llc –view–isel–dags sum.ll



查看 DAG 状态

-view-dag-combine1-dags	可以显示没有被优化的 DAG
-view-legalize-dags	合法化之前的 DAG
-view-dag-cmobine2-dags	第二次优化前
-view-isel-dags	显示指令选择前的 DAG
-view-sched-dags	在 Scheduler 之前 ISel 之后
-view-sunit-dags	可以显示 Scheduler 的依赖图

Register Allocation 寄存器分配

寄存器在 LLVM 中的表达

物理寄存器在 LLVM 里均有 1 – 1023 范围内的编号。在 GenRegisterNames.inc 里找到，比如 lib/Target/X86/X86GenRegisterInfo.inc

虚拟寄存器到物理寄存器的映射

直接映射使用 TargetRegisterInfo 和 MachineOperand 中的 API。间接映射的API用 VirtRegMap 以正确插入读写指令实现内存调度

LLVM 自带的寄存器分配算法 llc –regalloc=Greedy add.bc –o ln.s

Fast – debug 默认，尽可能保存寄存器。

Basic – 增量分配

Greedy – LLVM 默认寄存器分配算法，对 Basic 算法变量生存期进行分裂进行高度优化

PBQP – 将寄存器分配描述成分区布尔二次规划

MI

MC InstLowing

ASMPrinter <-- X86Asm Printer ...

MC Inst

MCStreamer

ASM API

Binary Intr

MCCode Emitter

Assembler

MCAsmStremer

MCInstPrinter

Assembly

X86 Intel InstPrinter ...

MCObject Streamer

MCELFStreamer ...

X86MCELFStreamer ...

学 LLVM BACKEND 能够干什么？

FRACTURE
AN ARCHITECTURE-INDEPENDENT DECOMPILER TO LLVM IR

https://github.com/draperlaboratory/fracture

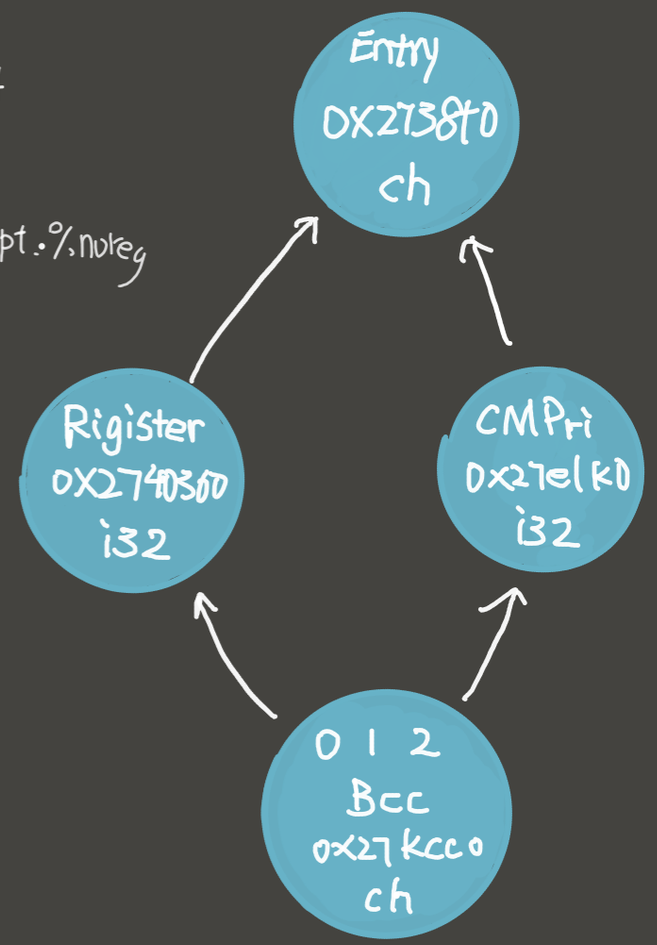
Machine Binary → ASM → MCInst → DAG

x86 PowerPC ARM

Instruction Object

```
00000038 : 10 48 2D E9 push {r4, r11, lr}
0000003C : 08 B0 8D E2 add r11, sp, #0x8
00000040 : 0C D0 4D E2 sub sp, sp, #0xC
...
```

```
%SP<def,tied1> = STMDB_UPD %SP<tied0>, pred:14
    pred:%noreg, %R4, %R11, %LR
%R11<def> = ADDri %SP, 8, pred:14, pred:%noreg, opt:%noreg
...
```



反向 LLVM 编译过程

LLVM MC API　MCDirector

用 LLVM API 解 symbols 和内存方法

Disassembler　Machine Inst

使用 TableGen 中的定义反向 DAG 操作表

LLVM CodeGen API　InvlSelDAG (Fracture)

IR