

# V8 For RISC-V：从何处来，向何处去

讲述 PLCT 2020 V8-RISCV 移植

PLCT V8 Team : 吴伟 邹小芳 陆亚涵 邱吉 陈家友 杨文章 陶立强

报告人 : 邱吉 [qiuji@iscas.ac.cn](mailto:qiuji@iscas.ac.cn)



# 目录

01 背景、动机和时间线

02 V8整体架构和移植所需工作介绍

03 技术路线图 ( vs Futurewei )

04 项目的输出、现状和未来

## 背景介绍-JavaScript语言和引擎

- JavaScript 是一种基于原型的函数式编程语言
- JavaScript虚拟机/引擎：简称JS引擎
  - 解释器 ( Interpreter )
  - 及时编译 ( JIT , Just-in-Time compile )
  - 预先编译 ( AOT , Ahead-of-Time compile )
- JavaScript 引擎负责将JavaScript程序进行 “解释执行” (interpreter) 或者 “即时编译成本地代码然后执行” (Just-In-Time compile to native code)
- 21世纪前十年，第二次浏览器大战，各大浏览器争夺市场份额，JS引擎技术不断发展改进
- 端侧：浏览器依赖于JS引擎：Google Chrome 在当前浏览器市场中占据了绝对份额优势（接近70%）
- 云侧：服务器上JavaScript的执行也依赖JS引擎

<https://zh.wikipedia.org/wiki/浏览器大战>

[https://en.wikipedia.org/wiki/Usage\\_share\\_of\\_web\\_browser](https://en.wikipedia.org/wiki/Usage_share_of_web_browser)

## 背景介绍-V8是什么

- V8 是 Google 浏览器 Chrome的 JavaScript 引擎，也支撑Node.js的运行
  - 200+万行代码
- V8位于Google的开源浏览器项目Chromium中
  - 一些主流的浏览器和APP，也是基于Chromium内核（使用了V8引擎）
    - 2020年2月微软发布的Windows10 Edge
    - 360浏览器/搜狗浏览器
- Chromium是Chrome浏览器的开源项目名称， Chrome浏览器基于Chromium的代码
- V8 is one of the best JavaScript Engine in the world

## 背景介绍-2020年初V8的RISC-V现状

- RISC-V是一个“刚满10岁”的“新兴”指令集体体系结构
- 彼时V8还没有RISC-V的porting



## Help Wanted

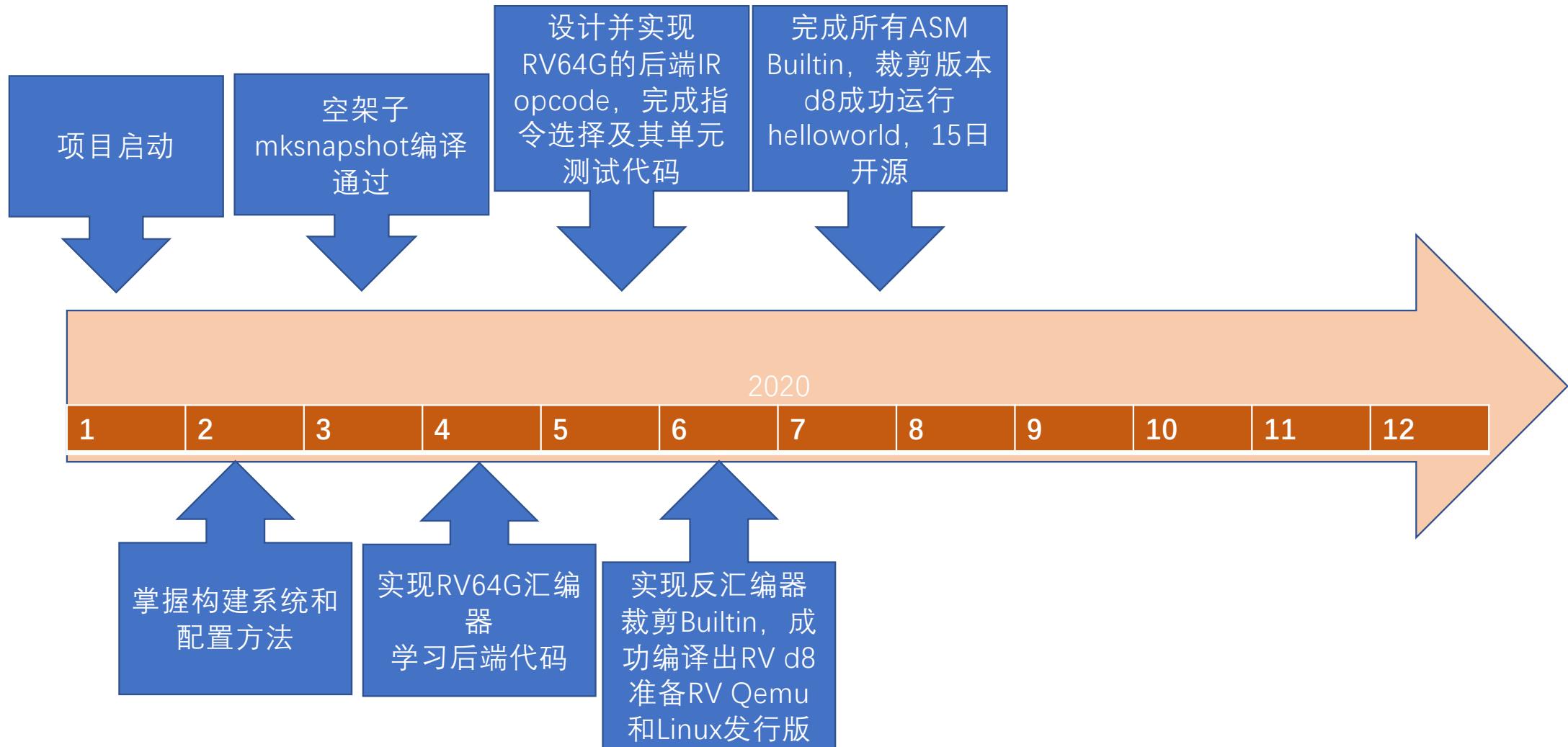
- V8
- Node.js
- Dart

<https://riscv.org/2020/05/happy-10th-birthday-risc-v/risc-v-rollout-at-hotchips-26/>  
<https://riscv.org/software-status/>

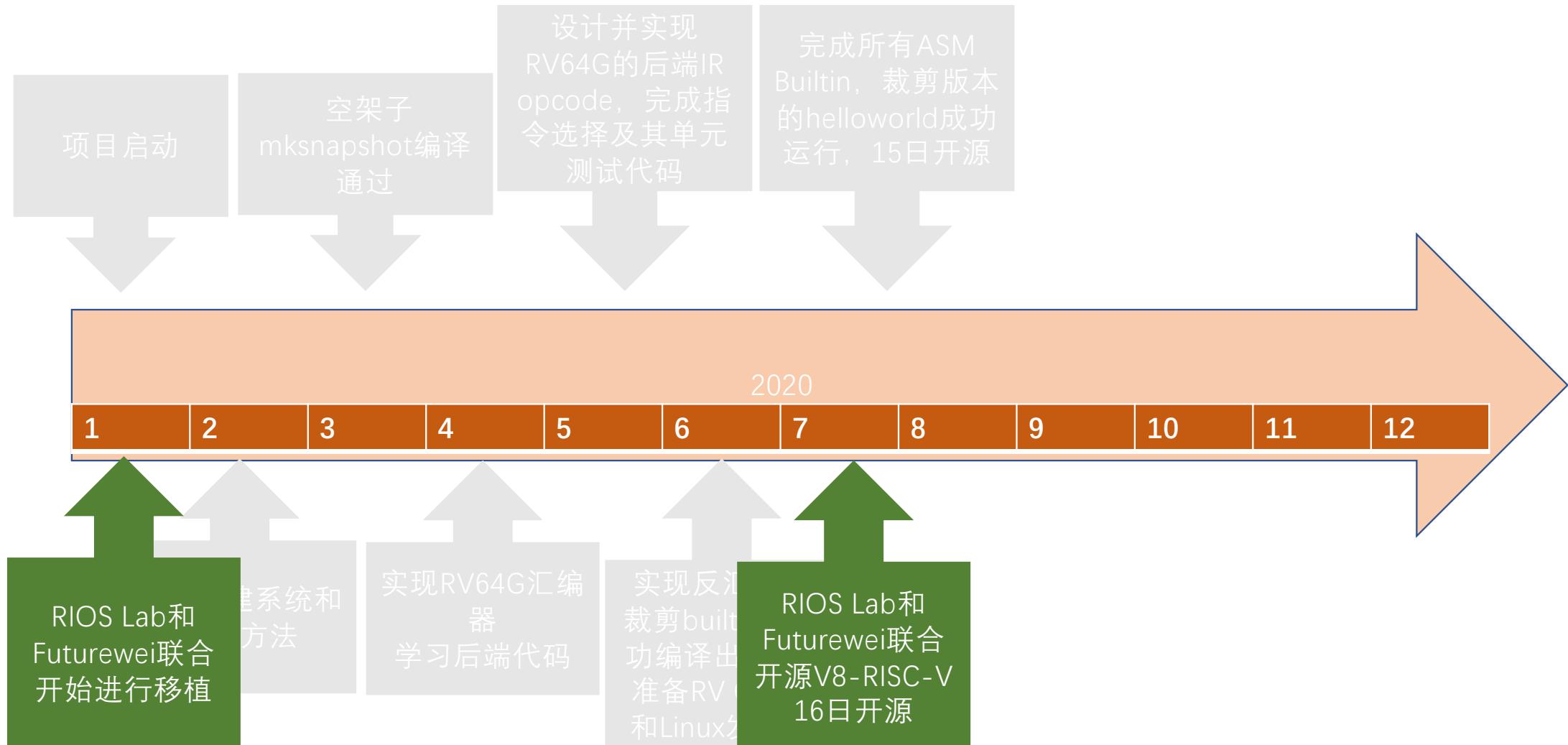
# 动机和目标

- 等待V8社区自己支持RISC-V后端？
- PLCT做起来的意义：
  - 锻炼维护V8代码体系的能力
  - 推动国内的RISC-V生态中关键、基础的部分
- 是否担心力量太小，**快不过、好不过**Google或其他大团队？
  - 不，作业总要自己写，学渣才有可能成长
- 现实的资源
  - 2020年初开始阅读代码和文档
  - 团队概况：全职工工程师3~4人，实习生2人
  - 全职成员掌握工具链开发，后端适配和Firefox JS引擎JIT移植经验
- 目标：6月底将能够在RISC-V平台运行helloworld.js的V8代码开源到github

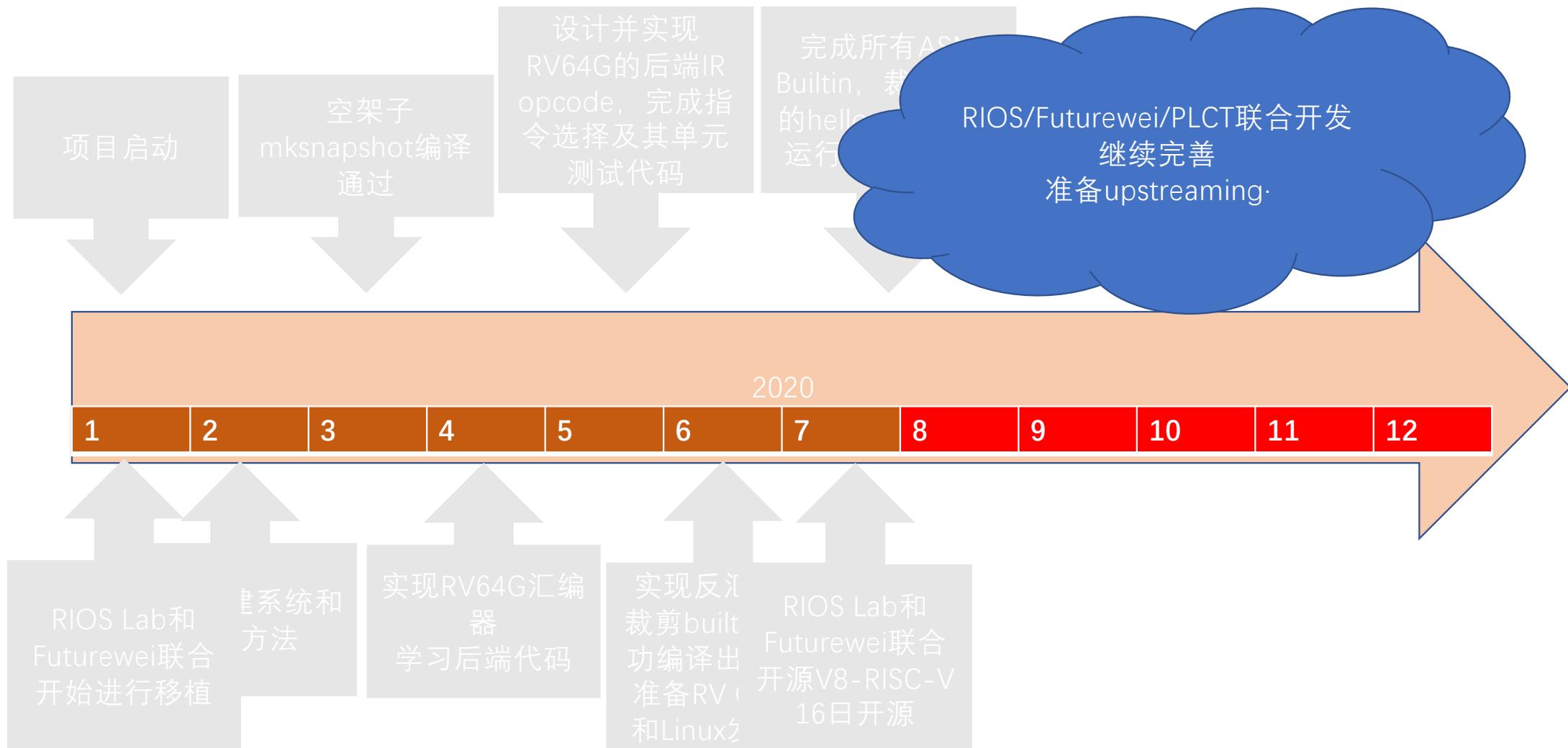
## 时间线



# 时间线-RIOS & Futurewei



# 时间线-RIOS & Futurewei & PLCT



# 目录

01 背景、动机和时间线

02 V8整体架构和移植所需工作介绍

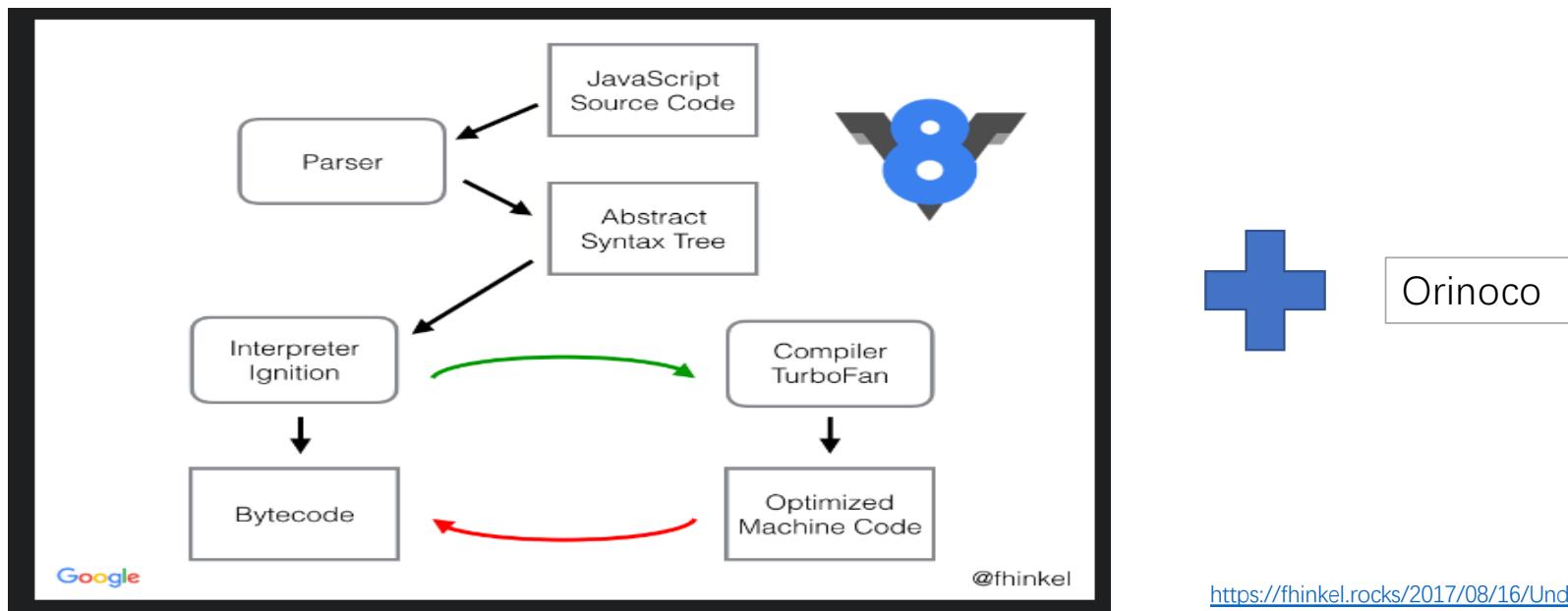
03 技术路线图 ( vs Futurewei )

04 项目的输出、现状和未来

# V8架构概述

## ● V8的主要子模块

- Parser : 负责将JavaScript源码转换为Abstract Syntax Tree (AST)
- Ignition : “解释器” , 负责将AST转换为Bytecode , 执行AOT生成的Bytecode Handler ; 并在运行中剖析热点信息 , 启动优化编译过程
- TurboFan : AOT和JIT编译器 , 在运行前AOT编译Bytecode Handler,、内置库函数、汇编代码stub和内联cache片段等 , 在运行中进行基于剖析反馈信息的优化编译
- Orinoco : 垃圾回收器



# TurboFan: V8的AOT和JIT编译器

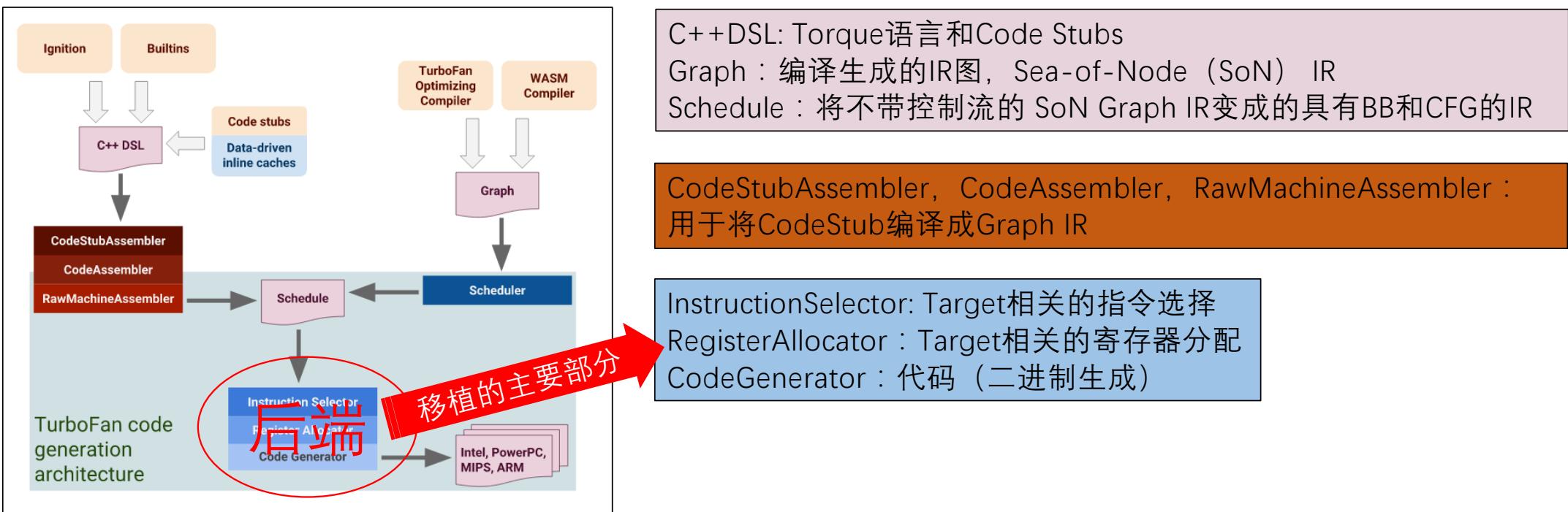
Ignition : V8的bytecode解释器，每一个bytecode对应一段预先编译成为unoptimized target binary的bytecode handler

Builtins : JavaScript内置函数，提供标准库操作，如RegExp、Array等，使用V8特有的Torque DSL编写

TurboFan Optimizing Compiler : 在解释执行阶段，如果发现热点代码，会再次进行优化编译

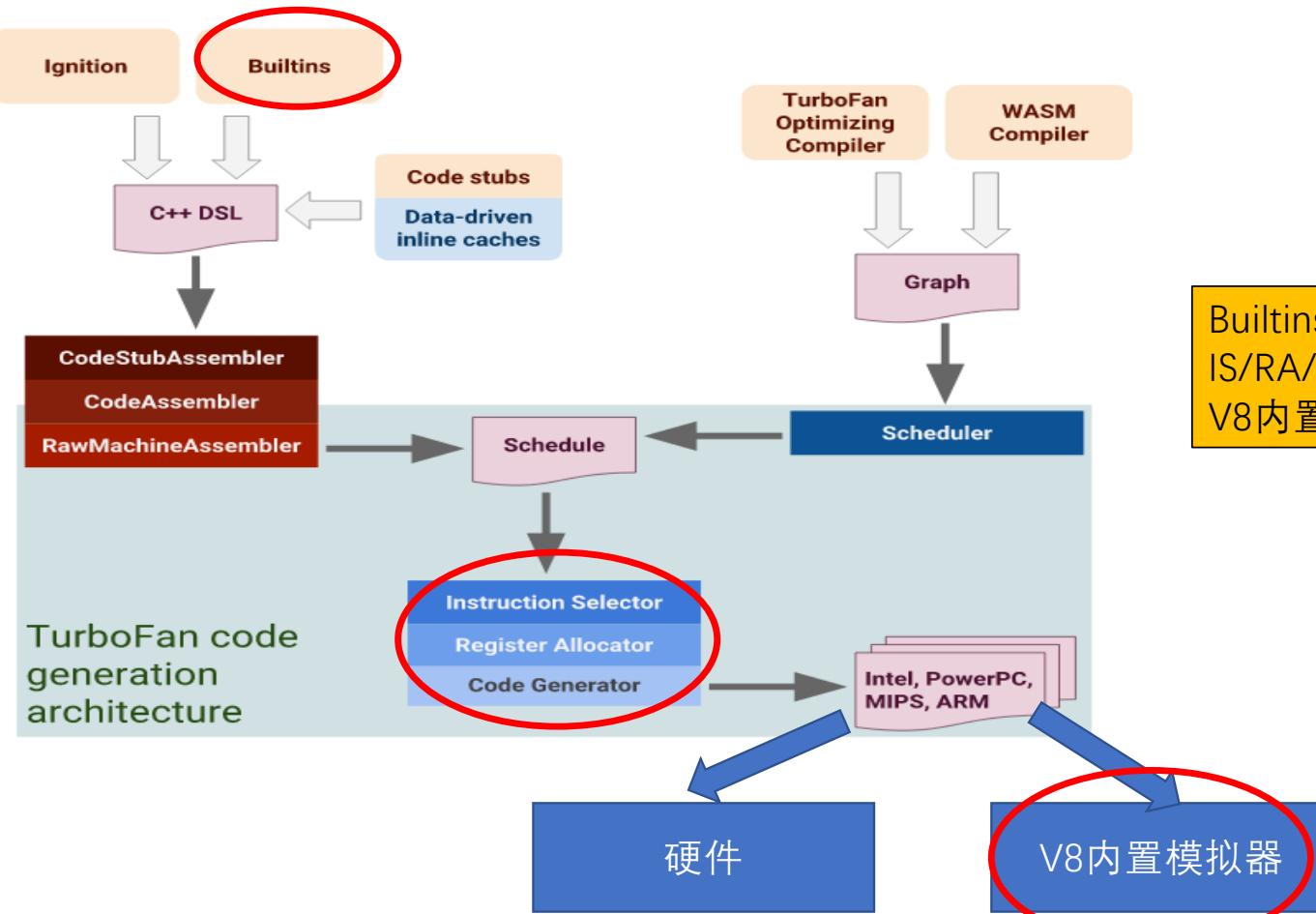
WASM Compiler : WASM的编译器

Code Stubs : V8特有的一种DSL，low-level的平台无关汇编语言，用于编写Inline Cache/Trampoline等底层操作



<https://benediktmeurer.de/2017/03/01/v8-behind-the-scenes-february-edition>

# 移植所需的工作

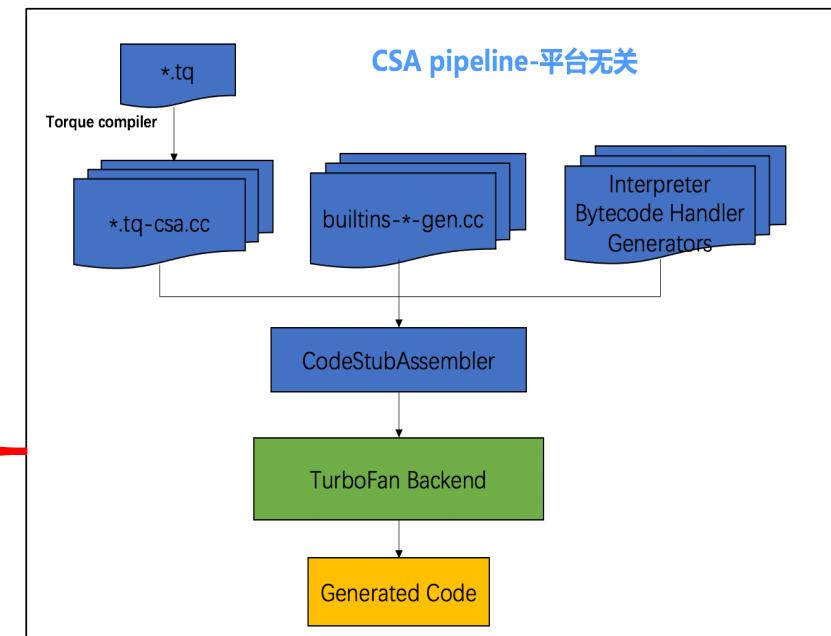
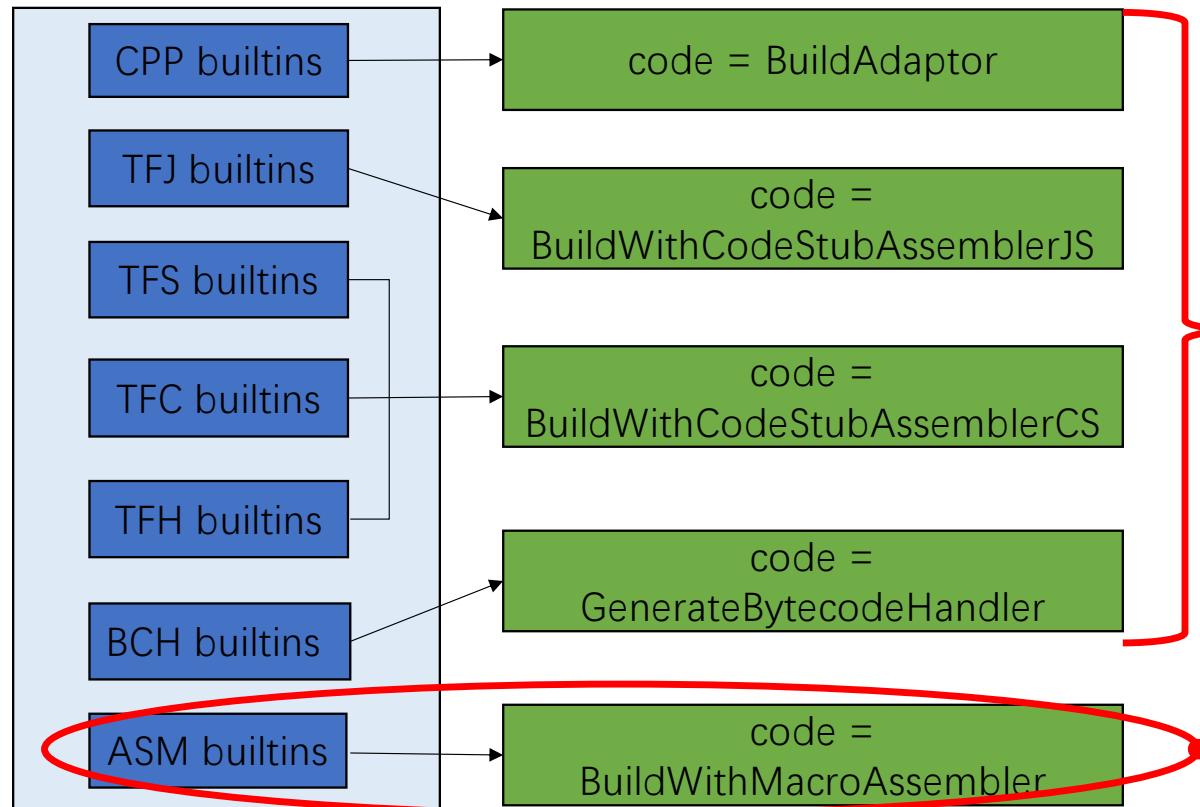


Builtins : 类似C库中平台相关部分  
IS/RA/CG : 类似GCC/LLVM的后端部分  
V8内置模拟器 : 类似Qemu/Spike和Gdb

<https://benediktmeurer.de/2017/03/01/v8-behind-the-scenes-february-edition>

# 移植内容1:Builtins

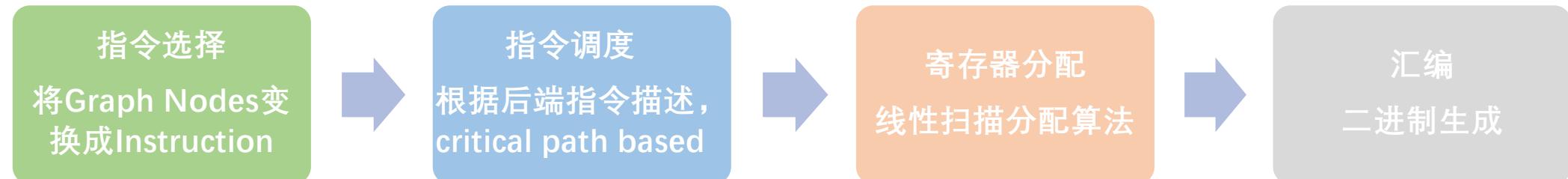
## Builtins handler的不同类型和移植内容



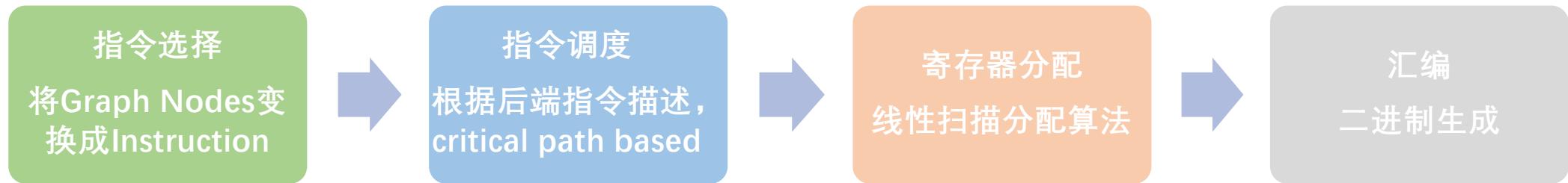
因为需要调用MacroAssembler API  
经由Assembler生成二进制  
所以需要重写  
\* 类似C库中汇编写的memcpy

## 移植内容2: 后端-全览

SelectInstructionsAndAssemble() @ pipeline.cc				
SelectInstructions()		AssembleCode()		
InstructionSelectionPhase.Run()	AllocateRegisters()	AssembleCodePhase		
SelectInstructions() @ instruction-selector.cc			AssembleCode()	
VisitBlock()	1. StartBlock() 2. AddInstructions() 3. AddTerminator() 4. EndBlock() <a href="#">@instructionscheduler.cc</a>		@code-generator.cc	
VisitControl() VisitNode()			AssembleBlock()	
VisitFooNonArchInst			AssembleInstruction()	
VisitFooArchInst <a href="#">@instruction-selector-[arch].cc</a>	GetInstructionLatency() GetInstructionFlags() <a href="#">@instruction-scheduler-[arch].cc</a>			
Emit()			AssembleArchInstruction() AssembleArchJump() AssembleArchBranch() AssembleArchDeoptBranch() AssembleArchBoolean() AssembleArchTrap() AssembleArchBinarySearchSwitchRange() AssembleArchBinarySearchSwitch() AssembleArchLookupSwitch() AssembleArchTableSwitch() <a href="#">@code-generator-[arch].cc</a>	
instructions_.push_back()			_ [MacroInst]() @ <a href="#">macro-assembler-[arch].cc</a> _ inst() @ <a href="#">assembler-[arch].cc</a>	



## 移植内容2: 后端 移植内容



```
IrOpcode::kWord32And  
VisitWordAnd32(Node)  
--IS-->  
Emit(kRiscvAnd32,⋯)
```

设计并实现RV64G的  
ARCH\_OPCODE及其  
lower过程

描述ARCH\_OPCODE的属  
性和延迟

描述CallDescriptor和  
linkage属性（类似ABI）  
描述RV64的寄存器文  
件

```
kRiscvAnd32  
--codegen-->  
__ And(dstreg,src1reg,src2opd)  
__ SII32(dstreg, dstreg, 0x0)  
--MacroAssembler-->  
andi(r,r,i) or and(r,r,r)  
--Assembler-->  
GenInstrALU_ri(0b111,rd,rs1,im12)
```

## 移植内容3: 反汇编器/模拟器-用于支撑debug

### ● 反汇编器

```
case RO_ANDI:{  
    Format(instr, "andi      'rd, ' rs1, 'imm12x'  ")  
    break;  
}
```

### ● 模拟器

```
case RO_ANDI:{  
    set_rd(imm12() & rs1());  
    break;  
}
```

# 目录

01 背景、动机和时间线

02 V8整体架构和移植所需工作介绍

**03 技术路线图 ( vs Futurewei )**

04 项目的输出、现状和未来

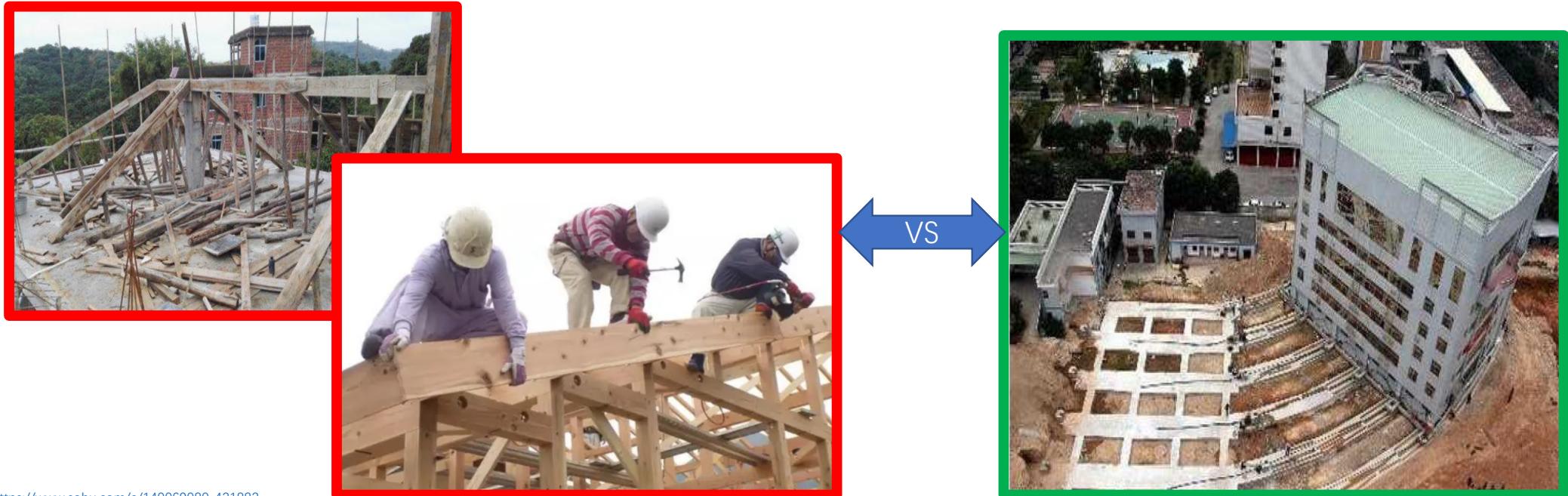
# 技术路线-与Futurewei的对比

第一里程碑 (开源时目标)	Native Run “Hello world”	Pass most of the regression test on Simulator mode
第一阶段	拷贝ARM64后端，重命名文件名、函数名、宏定义名到RV64，在构建系统中添加RV64配置， <b>尝试删减或者注释掉大部分代码和函数体，只保留会阻碍mksnapshot编译通过的部分</b>	拷贝MIPS64后端、反汇编器、模拟器，重命名文件名、函数名、宏定义名到RV64，在构建系统中添加RV64配置， <b>保留所有代码，形成RINO (RISC-V in Name Only) 后端</b> ，通过Cross build编译，能够产生X64 d8并在内置模拟器运行
第二阶段	前后夹挤 1. 前： <b>运行最小功能mksnapshot</b> 。尝试只走通一个Builtin (Increment) 的AOT流程，逐步补充后端在第一阶段被注释空的函数 2. 后： <b>从0实现汇编器</b>	在保留所有MIPS64汇编器和宏汇编器的API名称前提下，在二进制层面替换进入RV64 1. 汇编器：用RV64的指令二进制码替代 2. 修改内置Simulator和反汇编器的二进制译码逻辑
第三阶段	大桥合龙： 1. 前： <b>运行最小功能d8</b> 。面向helloworld.js 裁剪bytecode,builtin,lrOpcode，并设计ArchOpcode。 2. 后：宏汇编器和汇编器按照“前”的需求完成所有API实现 3. 后勤保障：反汇编器实现/后端单元测试	根据RV64G ISA特征修改汇编器、宏汇编器、模拟器和反汇编器，接入单元测试、模块测试等回归测试集
第四阶段	NA	1. 回归测试集Bug fix 2. Upstream

## 对比：不同技术路线的总结和反思

### ● 分析和综合

- 🐘 面对庞大的V8架构，采取了每一步都只分析和修改一小部分，先尝试拆除次要架构保证主框架不倒，再添新砖加新瓦（难度小，摸着石头过河）
- 🐘 对后端的整体流程有深入理解，也充分考虑到了调试是移植中最难的点，选择了“整体平移”“重新装修”的路线（需要有足够的全局观）



[https://www.sohu.com/a/140069080\\_421883](https://www.sohu.com/a/140069080_421883)

[http://www.precast.com.cn/index.php/subject\\_detail\\_id-3301.html](http://www.precast.com.cn/index.php/subject_detail_id-3301.html)

<http://www.civilcn.com/jianzhu/jzlw/jzjs/1342074980163514.html>

# 目录

01 背景、动机和过程

02 V8整体架构和移植所需工作介绍

03 技术路线图 ( vs Futurewei )

04 项目输出、现状和未来

## 项目输出

- 开源了V8 RISC-V代码：
  - <https://github.com/isrc-cas/v8-riscv>
- 参与RIOS/Futurewei的v8-riscv联合开发，成为项目maintainer
  - <https://github.com/v8-riscv/v8>
- 软件工程师的训练和培养
  - 实习生4名+
  - 员工3名+
- 原创文档、报告和高校课程的输出，以及参考资料的整理
  - <https://github.com/isrc-cas/PLCT-Open-Reports>
  - <https://space.bilibili.com/296494084/video?keyword=V8>

# 现状

- Code status :

- 回归测试集  
cctest/unittests/mjsunit/intl/message/inspector/mkgrokdump/debugger/wasm-  
js/wasm-spec-tests/wasm-api-tests pass on the latest final-rebase branch
- benchmark : SunSpider/Octane/Kraken pass

- Upstreaming status :

- <https://chromium-review.googlesource.com/c/v8/v8/+/2569387/>

## 未来

- 联合开发 & upstreaming (计划12月底完成)
- 性能优化 (2021年底达到与ARM64同等性能)
  - 指令选择的优化
  - ABI linkage的优化
  - 通过扩展指令进行性能优化 (WASM)
- RV32G后端
- Add ISA-extensions : C/V
- 欢迎实习生和社区贡献者加入我们：
  - <https://github.com/v8-riscv/v8>
    - 贡献代码、测试结果、加入Slack、参与双周会
  - 加入PLCT：
    - <https://github.com/lazyparser/weloveinterns>

# Q & A

2020/12