

# QuickJS 源码解读（一）：虚拟机的实现

Posted at 2019-08-27

编译 qjs 引擎本身

简介 qjsc 编译器

使用 XCode 进行 Debug

QuickJS 是 Fabrice Bellard 今年发布的一款 JavaScript 引擎，具有以下特性：

• 轻量而且易于嵌入：只需几个C文件，没有外部依赖，一个x86下的简单的“hello world”程序只要180 KiB。

• 具有极低启动时间的快速解释器：在一台单核的台式PC上，大约在100秒内运行ECMAScript 测试套件1 56000次。运行时实例的完整生命周期在不到300微秒的时间内完成。

相关阅读

- 几乎完整实现ES2019支持，包括：模块，异步生成器和完整Annex B支持 (传统的Web兼容性)。
- 通过100%的ECMAScript Test Suite测试。
- 可以将Javascript源编译为没有外部依赖的可执行文件。
- 使用引用计数（以减少内存使用并具有确定性行为）的垃圾收集与循环删除。
- 数学扩展：BigInt, BigFloat, 运算符重载, bigint模式, math模式。
- 在Javascript中实现的具有上下文着色和完成的命令行解释器。
- 采用C包装库构建的内置标准库。

本文主要记录我阅读 QuickJS 源码时记录的一些心得，主要用于学习用途。

## 编译

源码下载地址：<https://bellard.org/quickjs/>

## 编译 qjs 引擎本身

qjs 本身用于直接执行 JavaScript 代码：

```
$ make qjs
```

## 编译 qjsc 编译器

qjsc 编译器可以把 JavaScript 代码编译成 QuickJS 虚拟机的字节码（可直接通过 QuickJS 虚拟机执行）。也可以把 JavaScript 代码编译成一个 C 语言的 .c 文件，这个文件包含了字节码：

编译 qjs 引擎本身

编译 qjsc 编译器

使用 XCode 进行 Debug

OPCode

Runtime

命令执行

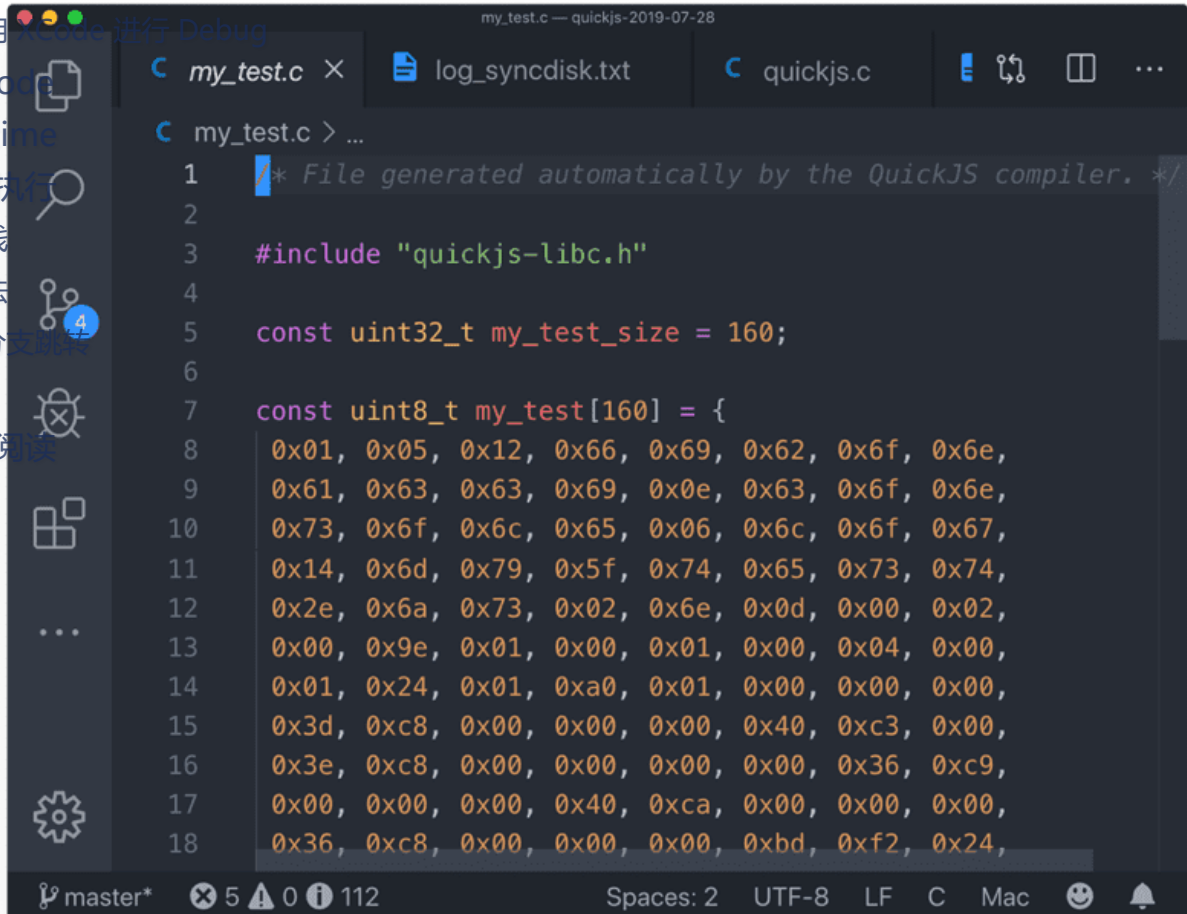
入栈

加法

If 分支跳转

总结

相关阅读



```
my_test.c — quickjs-2019-07-28
C my_test.c × log_syncdisk.txt quickjs.c
C my_test.c > ...
1  /* File generated automatically by the QuickJS compiler. */
2
3  #include "quickjs-libc.h"
4
5  const uint32_t my_test_size = 160;
6
7  const uint8_t my_test[160] = {
8      0x01, 0x05, 0x12, 0x66, 0x69, 0x62, 0x6f, 0x6e,
9      0x61, 0x63, 0x63, 0x69, 0x0e, 0x63, 0x6f, 0x6e,
10     0x73, 0x6f, 0x6c, 0x65, 0x06, 0x6c, 0x6f, 0x67,
11     0x14, 0x6d, 0x79, 0x5f, 0x74, 0x65, 0x73, 0x74,
12     0x2e, 0x6a, 0x73, 0x02, 0x6e, 0x0d, 0x00, 0x02,
13     0x00, 0x9e, 0x01, 0x00, 0x01, 0x00, 0x04, 0x00,
14     0x01, 0x24, 0x01, 0xa0, 0x01, 0x00, 0x00, 0x00,
15     0x3d, 0xc8, 0x00, 0x00, 0x00, 0x40, 0xc3, 0x00,
16     0x3e, 0xc8, 0x00, 0x00, 0x00, 0x00, 0x36, 0xc9,
17     0x00, 0x00, 0x00, 0x40, 0xca, 0x00, 0x00, 0x00,
18     0x36, 0xc8, 0x00, 0x00, 0x00, 0xbd, 0xf2, 0x24,
```

但是编译 qjsc 编译器首先需要编译 libquickjs 库：

```
$ make libquickjs.a
```

```
$ make qjsc
```

使用 qjsc 编译器把 my\_test.js 文件编译成 my\_test.c 文件：

```
$ ./qjsc -e -o my_test.c my_test.js
```

## 使用 XCode 进行 Debug

如果你想很清楚地了解整个虚拟机的执行过程，可能你需要 XCode 来进行单步调试：

## 简介

## 编译

编译 qjs 引擎本身

编译 qjsc 编译器

使用 XCode 进行 Debug

## OPCode

## Runtime

## 命令执行

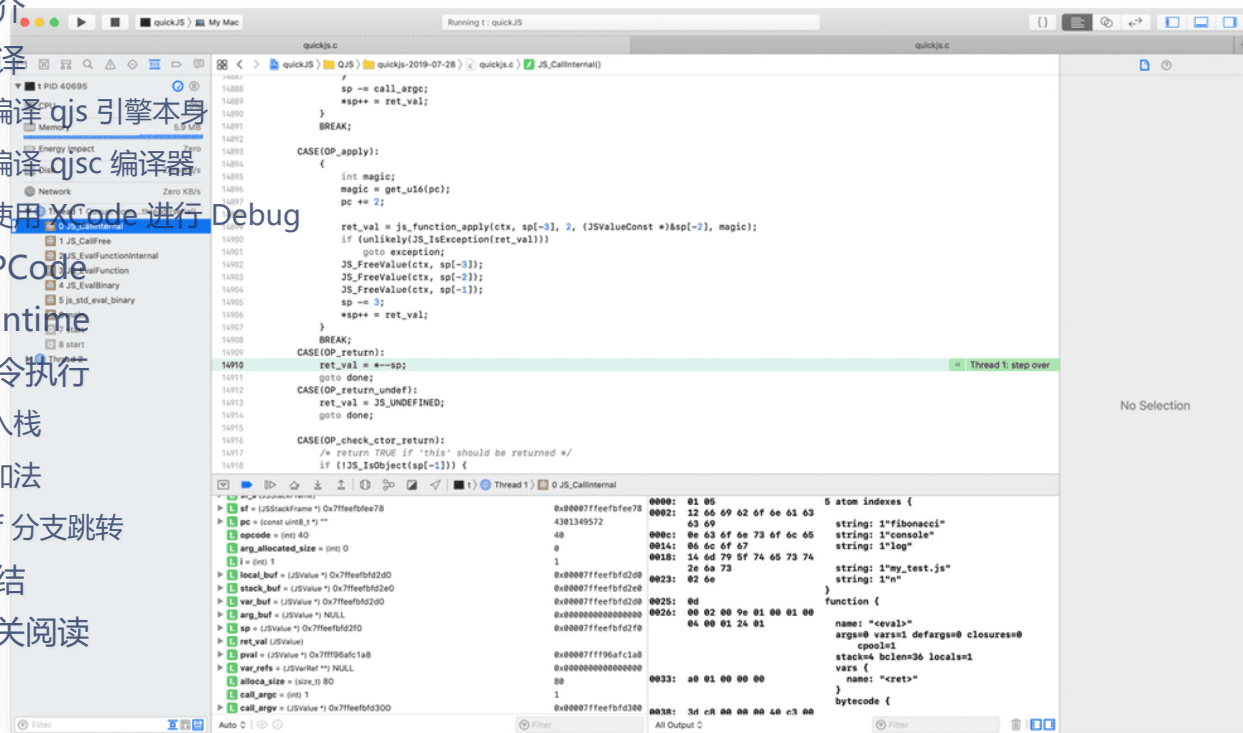
入栈

加法

if 分支跳转

## 总结

## 相关阅读



为了能在 XCode 下进行调试，你可能需要先在 Makefile 里面把优化给去掉。打开 Makefile，在里面找到这一句：

```
CFLAGS_OPT=$(CFLAGS) -O2
```

改成：

```
CFLAGS_OPT=$(CFLAGS) -O0
```

接下来就好办了，我们照着 [这篇教程](#) 走就可以在 XCode 里面 debug 了。

## OPCode

QuickJs 的虚拟机使用栈式虚拟机。对于什么式栈虚拟机推荐阅读 [这篇文章](#) 了解。

QuickJS 的 OPCode 十分简洁和紧凑。所有 OP 码的定义都放在 quickjs-opcode.h 文件里面：

简介

编译

编译 qjs 引擎

编译 qjs 库

使用 XCode 进行 Debug

OPCode

Runtime

命令执行

入栈

加法

If 分支跳转

总结

相关阅读

```

h quickjs-opcode.h > ...
55 #undef FMT
56 #endif /* FMT */
57
58 #ifdef DEF
59
60 #ifndef def
61 #define def(id, size, n_pop, n_push, f) DEF(id, size, n_pop, n_push, f)
62 #endif
63
64 def(invalid, 1, 0, 0, none) /* never emitted */
65
66 /* push values */
67 DEF(    push_i32, 5, 0, 1, i32)
68 DEF(    push_const, 5, 0, 1, const)
69 DEF(    fclosure, 5, 0, 1, const) /* must follow push_const */
70 DEF(push_atom_value, 5, 0, 1, atom)
71 DEF(    undefined, 1, 0, 1, none)
72 DEF(    null, 1, 0, 1, none)
73 DEF(    push_this, 1, 0, 1, none) /* only used at the start of a function */
74 DEF(    push_false, 1, 0, 1, none)
75 DEF(    push_true, 1, 0, 1, none)
76 DEF(    object, 1, 0, 1, none)

```

从上图可以知道 OPCode 定义分为这几个部分：

- id: OPCode 的名字
- size: OPCode 的字节大小。比如说 push\_i32 指令的 size 是 5. 那么第一个字节用来存 OPCode 本身，后面四个字节用来存 32 位的整型，也就是四个字节。
- n\_pop: 从栈上面弹出的元素的数量。和下面的 n\_push 一起用于统计函数需要分配的栈的大小。
- n\_push: 从栈上面插入元素的数量。
- f: 字节码的格式。

## Runtime

在虚拟机内部，运算的最基本单位是 JSValue，一个 JSValue 可以用以下 Tag 来标示：

```

enum {
    /* all tags with a reference count are negative */
    JS_TAG_FIRST      = -10, /* first negative tag */
    JS_TAG_BIG_INT    = -10,
    JS_TAG_BIG_FLOAT  = -9,
    JS_TAG_SYMBOL     = -8,
    JS_TAG_STRING     = -7,
    JS_TAG_SHAPE      = -6, /* used internally during GC */
    JS_TAG_ASYNC_FUNCTION = -5, /* used internally during GC */
    JS_TAG_VAR_REF    = -4, /* used internally during GC */
    JS_TAG_MODULE     = -3, /* used internally */
    JS_TAG_FUNCTION_BYTECODE = -2, /* used internally */
    JS_TAG_OBJECT     = -1,
    JS_TAG_INT        = 0,
    JS_TAG_BOOL       = 1,
    JS_TAG_NULL       = 2,
    JS_TAG_UNDEFINED  = 3,
    JS_TAG_UNINITIALIZED = 4,
    JS_TAG_CATCH_OFFSET = 5,
    JS_TAG_EXCEPTION  = 6,
    JS_TAG_FLOAT64     = 7,
    /* any larger tag is FLOAT64 if JS_NAN_BOXING */
};

```

那么一个 JSValue 的表示是这样的:

```

typedef union JSValueUnion {
    int32_t int32;
    double float64;
    void *ptr;
} JSValueUnion;

typedef struct JSValue {
    JSValueUnion u;
    int64_t tag;
} JSValue;

```



从这个结构体可以看出一个 JSValue 占用 16 字节的内存。tag 用来表示这个 Value 的类型。而 JSValueUnion 则是用来储存真正的值。这个值可以是 int/float 或者一个指针指向一个真正的对象。

简介

编译

编译 QuickJS 引擎本身

计数器:

编译 QuickJS 编译器

使用 XCode 进行 Debug

OPCode

typedef struct JSRefCountHeader {

Runtime

int ref\_count;

命令执行

} JSRefCountHeader;

入栈

加法

命令执行

分支跳转

知道了上述背景之后，我们就可以看看 QuickJS 虚拟机具体的执行过程了。需要知道的是，一个栈虚拟机，需要两个很重要的指针，分别是 pc 和 sp。

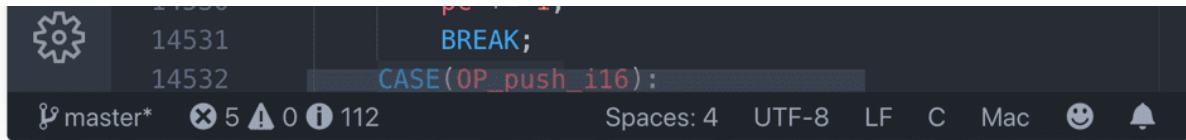
pc 指的是程序计数器。指向正在读取的 OPCode，读取之后 pc 会 ++，指向下一个字节。

sp 是栈指针。指向栈机器栈顶的下一个元素，用来进行栈相关的操作。

## 入栈

```

quickjs.c — quickjs-2019-07-28
quickjs.c × my_test.c log_syncdisk.txt
quickjs.c > ...
14515         BREAK;
14516     f SHORT_OPCODES
14517         CASE(OP_push_minus1):
14518             CASE(OP_push_0):
14519             CASE(OP_push_1):
14520             CASE(OP_push_2):
14521             CASE(OP_push_3):
14522             CASE(OP_push_4):
14523             CASE(OP_push_5):
14524             CASE(OP_push_6):
14525             CASE(OP_push_7):
14526                 *sp++ = JS_NewInt32(ctx, opcode - OP_push_0);
14527                 BREAK;
14528             CASE(OP_push_i8):
14529                 *sp++ = JS_NewInt32(ctx, get_i8(pc));
14530                 pc += 1;
  
```



## 简介

## 编译

编译 quickjs 引擎本身  
入栈操作就是把读入的值变成一个 JSInt32，然后赋值到 sp 指针指向的内存，然后 sp++。就完成了入栈操作。我们可以看到 QuickJS 的字节码实际是十分冗余的，具体为什么这么设计我不太清楚。估计是 0-7 之间的数可以用一个字节搞定，节省空间，但是我个人感觉也太省了，对实际运行应该帮助不大。

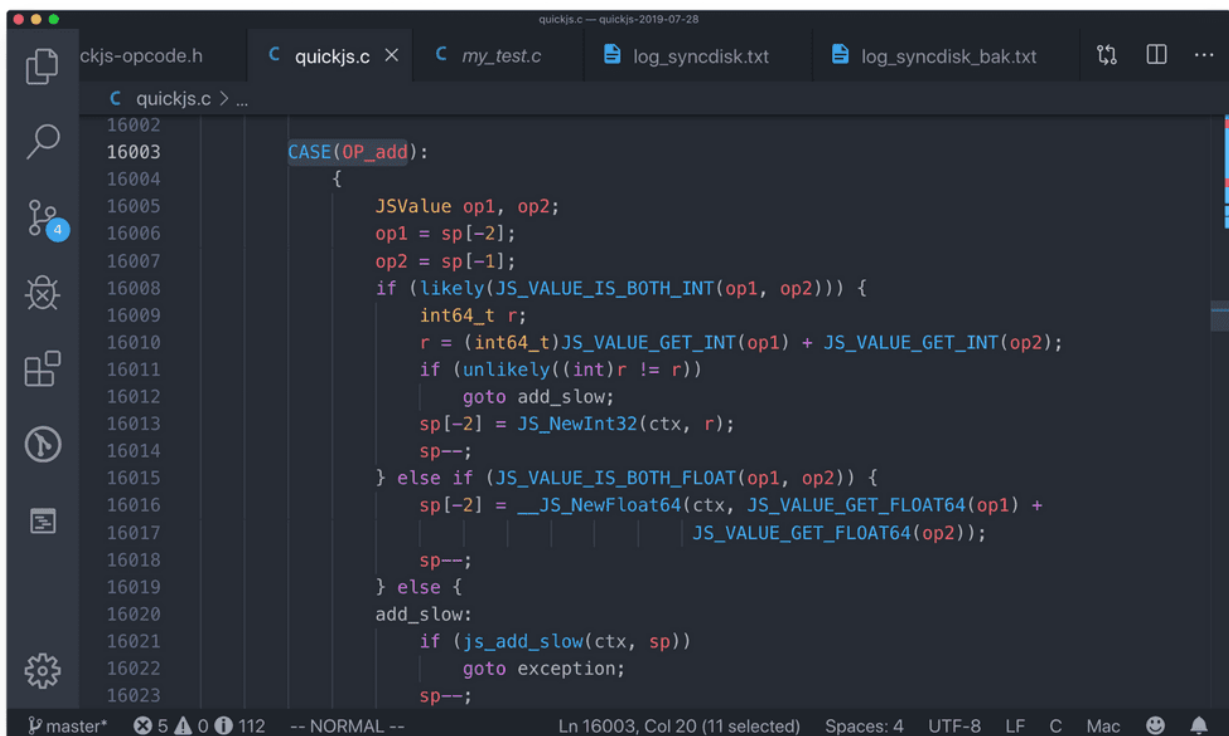
## 命令执行

OP\_push\_i8 是一个两字节的 OP 码，第二个字节就是一个 8 位的整数，同理  
OP\_push\_i16 就是一个三个字节 的指令，入栈一个 16 位的整数。

If 分支跳转

## 加法

## 相关阅读



栈虚拟机上的加法就是从栈弹出两个元素，相加，然后入栈。这里首先从栈顶取出 op1 和 op2。然后判断是不是 int。这里的 int 类型是 32 位的。所以它先把他们切到 64 位进行相加。加完再切回 32 位的 int，看看数据是不是一样。若不一样，则说明加法产生了溢出，这个时候就丢到 add\_slow 进行处理。

若两个 op 都是 float64 则进行浮点数的加法。

## If 分支跳转

If 分支跳转是程序里面很重要的一步，实现了分支跳转，for 和 if 语句都可以实现。QuickJs 有 goto 指令，比较简单，这里就不说了，这里介绍一下

OP\_if\_true :

编译 qjs 引擎本身

编译 qjsc 编译器

使用 C 对 Code 进行 Decode

OPCode

Runtime

命令执行

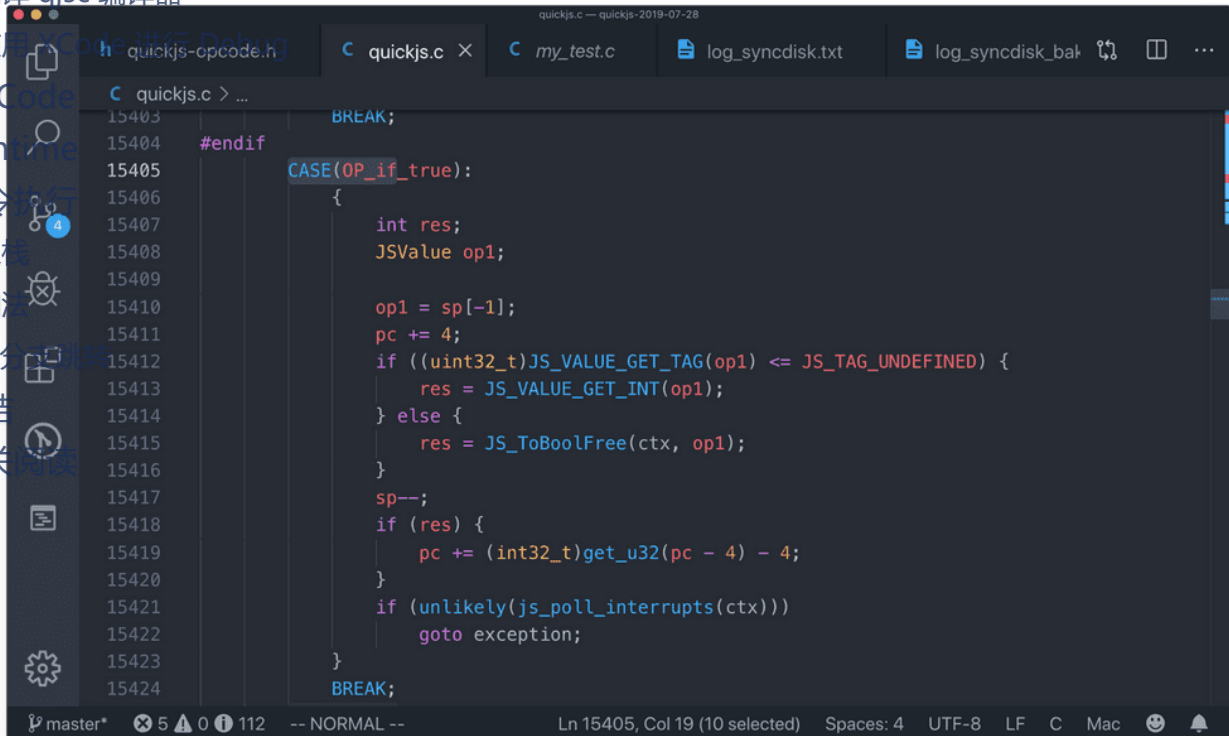
入栈

加法

If 分支跳转

总结

相关阅读



```
quickjs.c — quickjs-2019-07-28
quickjs.c x my_test.c log_syncdisk.txt log_syncdisk_bak
quickjs.c > ...
15403 BREAK;
15404 #endif
15405 CASE(OP_if_true):
15406 {
15407     int res;
15408     JSValue op1;
15409
15410     op1 = sp[-1];
15411     pc += 4;
15412     if (((uint32_t)JS_VALUE_GET_TAG(op1) <= JS_TAG_UNDEFINED) {
15413         res = JS_VALUE_GET_INT(op1);
15414     } else {
15415         res = JS_ToBoolFree(ctx, op1);
15416     }
15417     sp--;
15418     if (res) {
15419         pc += (int32_t)get_u32(pc - 4) - 4;
15420     }
15421     if (unlikely(js_poll_interrupts(ctx)))
15422         goto exception;
15423 }
15424 BREAK;
```

If 分支跳转主要是看栈顶元素是不是为 true。若为 true，则把 pc 加等于相应的 offset。这个 offset 在 QuickJS 里面是 32 位的，这意味着 OP\_if\_true 是一个 5 个字节的 OPCode。这个 offset 在编译的过程中就可以被算出来。

图中有一个 tag 是否小于等于 JS\_TAG\_UNDEFINED 的判断。这个判断根据上面的 tag 的定义。我们知道只有这几个类型是不符合这个判断的：

- JS\_TAG\_UNINITIALIZED = 4
- JS\_TAG\_CATCH\_OFFSET = 5
- JS\_TAG\_EXCEPTION = 6
- JS\_TAG\_FLOAT64 = 7

以上几个类型需要用 JS\_ToBoolFree，进行判断，其他类型则可以从 JSValue 读出真正的 bool 值。



## 总结

QuickJS 源码实现非常简单易懂，是一个非常适合学习的 JS 引擎。有一点谈不上好坏的就是 像 Bellard 这样的 hacker 出来的代码非常的骚气。比如说通篇的 goto，不过幸好不是很难懂，有些逻辑确实 goto 写起来比较爽这一点，可以理解。另一方面不太好的就是大部分代码都写尽了一个 .c 文件 里面了，读函数调用跳来跳去比较麻烦。

编译 qjsc 编译器  
使用 XCode 进行 Debug  
总的来说是比较适合阅读的源码了。

## 相关阅读


- QuickJS 源码解读（二）：基础设施和标准库
- QuickJS 官方文档: <https://bellard.org/quickjs/quickjs.pdf>
- 总结栈式虚拟机: <https://www.iteye.com/blog/rednaxelafx-492667>
- X86 架构: <https://blog.csdn.net/ajigegege/article/details/17055779>
- 使用 Xcode 进行 Debug:  
<https://blog.csdn.net/u011577874/article/details/73000207>


在 VINCENT'S PERSONAL BLOG 上还有


<b>webpack 如何通过作用域分析消除无用代码</b> 3 年前 • 1条评论 Vincent Chan's Bus Station. Blog of Vincent.	<b>写一个光线追踪渲染器</b> 4 年前 • 2条评论 Vincent Chan's Bus Station. Blog of Vincent.	<b>转换网易 ncm 格式到 mp3</b> 2 年前 • 2条评论 Vincent Chan's Bus Station. Blog of Vincent.	<b>在 I Spi</b> 3 年 Vinc Blog
--	---	---	---------------------------------------

What do you think?

2条回复

 Upvote

 Funny

 Sad

简介

编译

编译 qjs 引擎本身

评论 在线社区

编译 qjsc 编译器

隐私政策

1 登录

使用 XCode 进行 Debug

推荐

推文

分享

评分最高

OPCode

Runtime

加入讨论...

命令执行

通过以下方式登录

或注册一个 DISQUS 帐号

加法

If 分支跳转

总结

相关阅读

danta • 2 years ago



^ | v • 回复 • 分享

订阅

在您的网站上使用 Disqus 添加 Disqus 添加

不要出售我的数据