QuickJS架构及源代码分析

by 丁乐华@20200607

目录

一、QuickJS项目介绍

JavaScript简介

- **关键特性：动态类型、函数式、基于对象编程、原型链**
- **函数闭包**：函数定义 + 定义时的上下文（词法作用域）

```
/* 1. 动态类型 */
let a = 1;
a = '123';

/* 2. 隐式类型转换 */
let b = 1 + '23'; // b为：'123'

/* 3. 函数跟普通数据类型一样赋值给变量add */
let add = () => {
  let a = 1;
  return (b) => {
    return a + b;
  };
};
let add1 = add(); // add1是函数闭包
add1(2); // 输出为：3
```

```
/* 4. 对象与原型 */
let obj1 = {
  a: 1,
  add: function () {
    return this.a + this.b;
  }
};
let obj2 = {
  b: 2
};
// 设置obj2的原型为obj1
Object.setPrototypeOf(obj2, obj1);
obj2.add(); // 输出为：3
```

QuickJS项目简介

- QuickJS是JavaScript语言的一个实现，实现的语言标准为最新的ES2020（绝大部分特性）
- 项目地址：**bellard.org/quickjs**，由QEMU的作者Fabrice Bellard等人维护
- 对外提供了以下几个内容：
  - **qjsc编译器**——生成字节码，链接QuickJS执行库就可以生成可执行文件
  - **qjs解释器**——直接编译并执行生成的字节码，支持REPL交互
  - **quickjs-lib.h库**——可以很方便地给C程序调用
- 主要特性：**小巧启动速度快、垃圾回收器、支持大数计算、友好的REPL交互式解释器**

QuickJS项目的目录结构

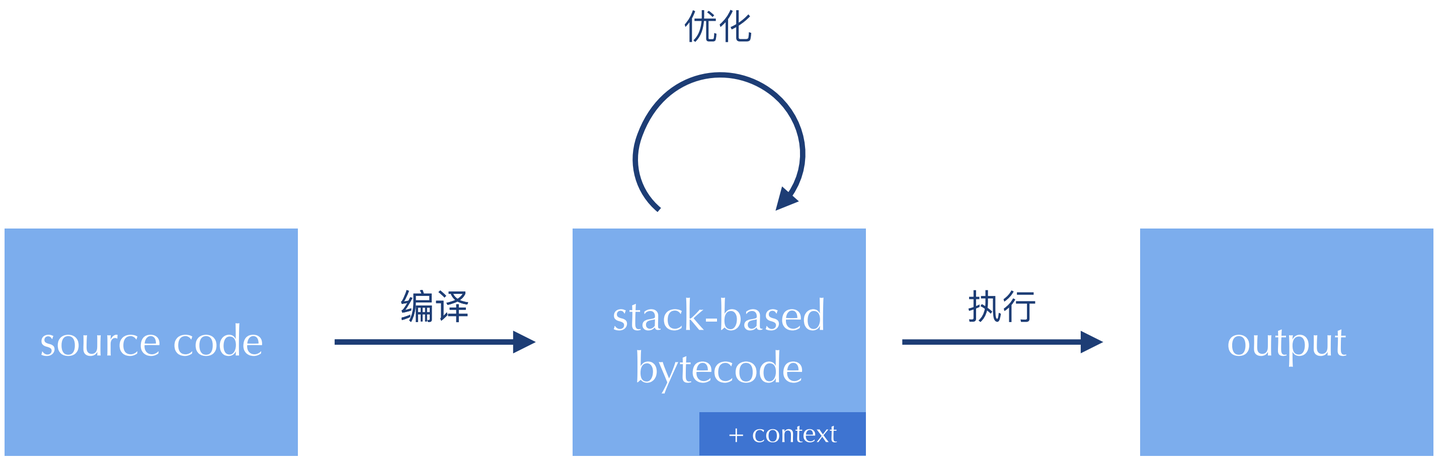
编译和安装命令：make install

qjsc.c	编译器程序qjsc的入口
qjs.c	REPL交互程序qjs的入口
repl.js	REPL的实现
qjscalc.js	数学计算器应用，支持任意长度的整数和浮点数、分数、复数、多项式、矩阵计算
quickjs.c quickjs.h	主程序位置（编译器和解释器都在里面）
quickjs-atom.h	预定义的字符串
quickjs-opcode.h	字节码中的操作符定义
cutils.c cutils.h	辅助函数
list.h	klist实现
libbf.c libbf.h	BigFloat实现
libregexp.c libregexp.h libregexp-opcode.h	正则表达式实现
libunicode.c libunicode.h libunicode-table.h	Unicode编码的支持
quickjs-lib.c quickjs-lib.h	暴露给C程序使用的API

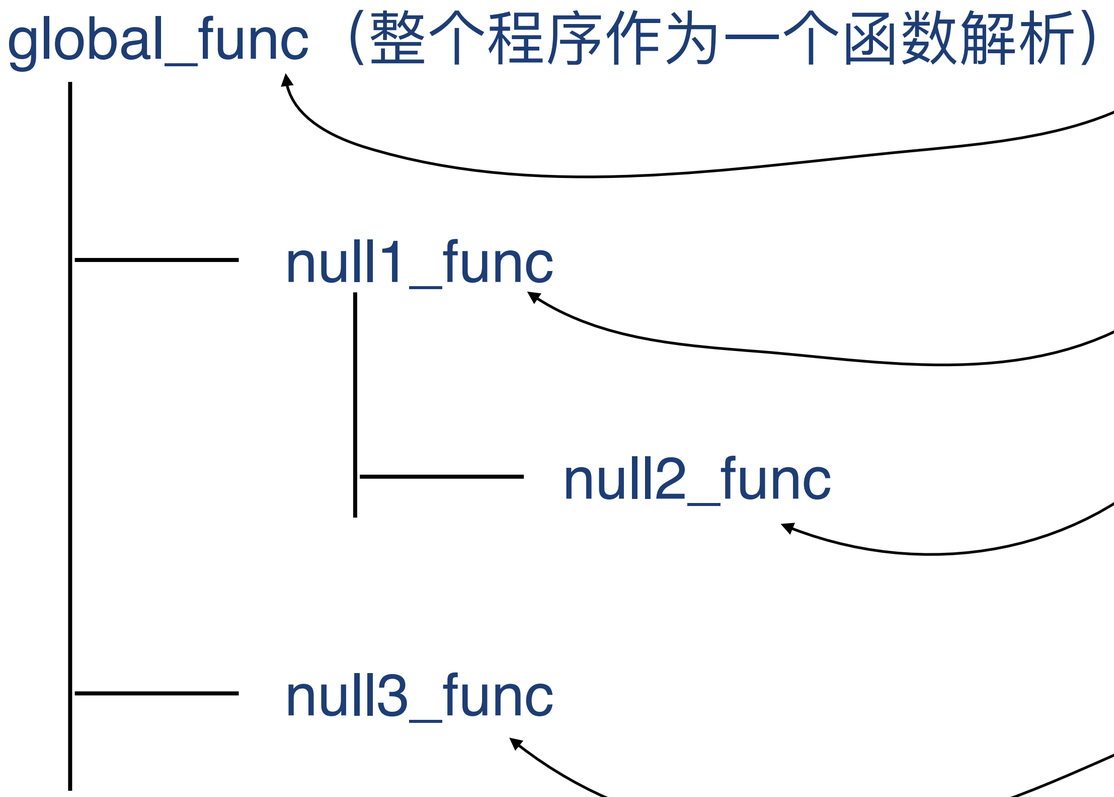
examples/	示例JS程序
tests/	测试程序
doc/	文档
Changelog/	程序修改记录
readme.txt	
Makefile	
TODO	
VERSION	

二、QuickJS架构介绍

QuickJS的架构图

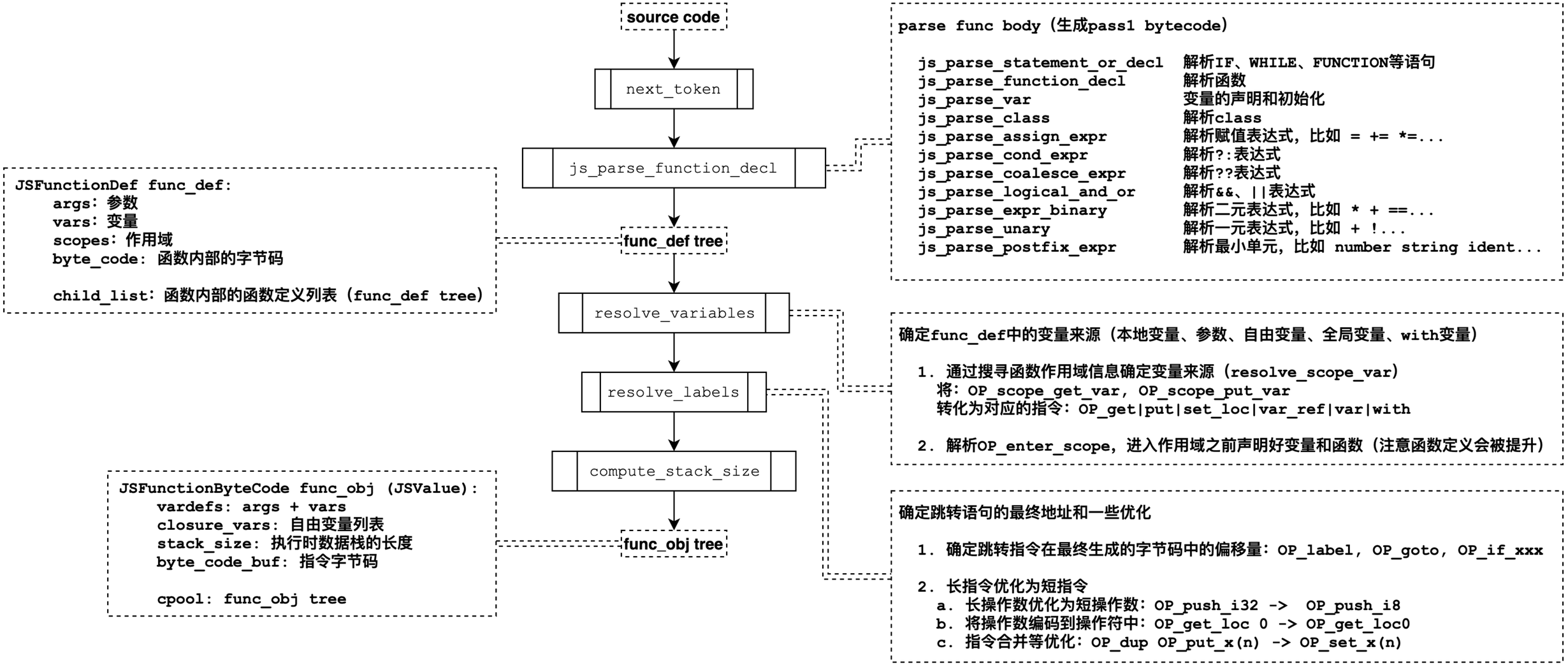


解析单元——函数




```
/* 1. 动态类型 */
let a = 1;
a = '123';

/* 2. 隐式类型转换 */
let b = 1 + '23'; // b为: '123'

/* 3. 函数跟普通数据类型一样赋值给变量add */
let add = () => {
  let a = 1;
  return (b) => {
    return a + b;
  };
};
let add1 = add(); // add1和add2都是函数闭包
add1(2); // 输出为: 3

/* 4. 对象与原型 */
let obj1 = {
  a: 1,
  add: function () {
    return this.a + this.b;
  }
};
let obj2 = {
  b: 2
};
// 设置obj2的原型为obj1
Object.setPrototypeOf(obj2, obj1);
obj2.add(); // 输出为: 3
```

编译&优化


source code
next_token
js_parse_function_decl
func_def tree
resolve_variables
resolve_labels
compute_stack_size
func_obj tree
JSFunctionDef func_def:
args：参数
vars：变量
scopes：作用域
byte_code： 函数内部的字节码
child_list：函数内部的函数定义列表（func_def tree）
parse func body（生成pass1 bytecode）
js_parse_statement_or_decl 解析IF、WHILE、FUNCTION等语句
js_parse_function_decl 解析函数
js_parse_var 变量的声明和初始化
js_parse_class 解析class
js_parse_assign_expr 解析赋值表达式，比如 = += *=...
js_parse_cond_expr 解析?:表达式
js_parse_coalesce_expr 解析??表达式
js_parse_logical_and_or 解析&&、||表达式
js_parse_expr_binary 解析二元表达式，比如 * + ==...
js_parse_unary 解析一元表达式，比如 + !...
js_parse_postfix_expr 解析最小单元，比如 number string ident...
确定func_def中的变量来源（本地变量、参数、自由变量、全局变量、with变量）
1. 通过搜寻函数作用域信息确定变量来源（resolve_scope_var）
将：OP_scope_get_var, OP_scope_put_var
转化为对应的指令：OP_get|put|set_loc|var_ref|var|with
2. 解析OP_enter_scope，进入作用域之前声明好变量和函数（注意函数定义会被提升）
JSFunctionByteCode func_obj (JSValue):
vardefs: args + vars
closure_vars: 自由变量列表
stack_size: 执行时数据栈的长度
byte_code_buf: 指令字节码
cpool: func_obj tree
确定跳转语句的最终地址和一些优化
1. 确定跳转指令在最终生成的字节码中的偏移量：OP_label, OP_goto, OP_if_xxx
2. 长指令优化为短指令
a. 长操作数优化为短操作数：OP_push_i32 -> OP_push_i8
b. 将操作数编码到操作符中：OP_get_loc 0 -> OP_get_loc0
c. 指令合并等优化：OP_dup OP_put_x(n) -> OP_set_x(n)

示例：

编译

resolve_variables

resolve_labels

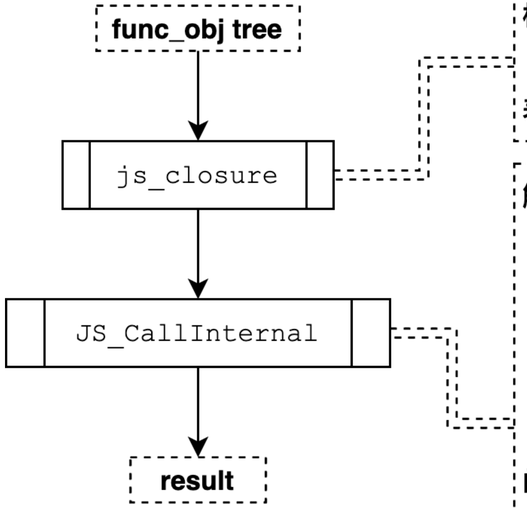
```
function abs(a) {
  if (a < 0) {
    a = -a;
  }
  return a
}
```

pass1
enter_scope 1
line_num 2
enter_scope 2
scope_get_var a,2
push_i32 0
lt
if_false 0:77
enter_scope 3
line_num 3
scope_make_ref a,1:61,3
scope_get_var a,3
neg
label 1:61
61: insert3
put_ref_value
drop
leave_scope 3
line_num 4
label 0:77
77: leave_scope 2
line_num 5
scope_get_var a,1
return

pass2
line_num 2
get_arg 0: a
push_i32 0
lt
if_false 0:43
line_num 3
get_arg 0: a
neg
dup
put_arg 0: a
drop
line_num 4
label 0:43
43: line_num 5
get_arg 0: a
return

pass3
get_arg0 0: a
push_0 0
lt
if_false8 8
get_arg0 0: a
neg
put_arg0 0: a
8: get_arg0 0: a
return

解释执行



根据函数定义和执行上下文，生成函数闭包对象：
1. 生成函数对象，设置函数原型、标准属性
2. 确定自由变量哪些来自父函数的本地变量和参数，哪些来自父函数的自由变量列表。生成当前函数的var_refs（get_var_ref）

解释每一条指令：
1. 分配内存，存放参数、本地变量和执行时的数据栈（大小为stack_size）
2. 依次执行每条指令
   - 数据入栈指令：push_xxx, fclosure
   - 跳转指令：goto, if_false, if_true
   - 调用指令：call, call_method
   - 数据栈转换指令：drop, nip, dup, insert, perm, rot, swap, ...
   - 计算指令：add, sub, mul, div, mod, pow, neg, or, ...

函数：
- 生成JSValue：JS_MKVAL, JS_MKPTR（INT, BOOL, OBJECT, FUNCTION...）
- 引用计数相关：JS_DupValue, JS_FreeValue
- 类型转化：JS_ToObject, JS_ToString, JS_ToNumeric, JS_ToBool
- 数据创建：JS_NewBool, JS_NewString, JS_NewObject, JS_NewArray
- 垃圾回收：js_trigger_gc, add_gc_object

示例：

```
bytecode:
    get_arg0 0: a
    push_0 0
    lt
    if_false8 8
    get_arg0 0: a
    neg
    put_arg0 0: a
8:  get_arg0 0: a
    return
```

+

```
args: a
locals:
stack_size: 2
```

执行 → `abs(-1);`

1

执行`abs(-1);`时的数据栈：

get_arg0 0: a	push_0 0	lt	if_false8 8	get_arg0 0: a	neg	put_arg0 0: a	get_arg0 0: a	return
-1	-1	TRUE	false则跳到8	-1	1	修改参数a的值为1	1	将1返回
	0							

三、QuickJS源码解读

1. 闭包的实现原理

什么是闭包?

```
let add = () => {
  let a = 1;
  return (b) => {
    return a + b;
  };
};
let add1 = add(); // add1是函数闭包
add1(2); // 输出为: 3
```

实现原理

① 解析变量时，**收集函数中定义的变量和作用域(scope_level)信息**，生成scope_get_var和scope_put_var操作符

② 根据收集到的变量及作用域信息，将scope_get_var和scope_put_var转化为更确切的操作符，**并生成自由变量列表(closure_var)**。本地变量为get|put_loc，自由变量为get|put_var_ref

③ 在执行创建函数闭包指令(OP_fclosure)时，根据函数的closure_var信息和创建时的上下文，**生成自由变量的引用(var_refs)**

④ 执行自由变量获取指令时(OP_get_var_ref)，**根据索引从var_refs中直接获取对应的自由变量**

添加和使用变量

```
int add_scope_var(JSContext *ctx,
                  JSFunctionDef *fd,
                  JSAtom name,
                  JSVarKindEnum var_kind)
{
    int idx = add_var(ctx, fd, name);
    if (idx >= 0) {
        JSVarDef *vd = &fd->vars[idx];
        vd->var_kind = var_kind;
        vd->scope_level = fd->scope_level;
        vd->scope_next = fd->scope_first;
        fd->scopes[fd->scope_level].first = idx;
        fd->scope_first = idx;
    }
    return idx;
}
```

```
int add_var(JSContext *ctx,
            JSFunctionDef *fd,
            JSAtom name)
{
    JSVarDef *vd;
    vd = &fd->vars[fd->var_count++];
    /* 这里有个关键点：会将scope_level设置为0，
     * 这是必须的，对于var声明的变量来说。
     * 而对于let声明的变量，则会在后续显式设置scope_level */
    memset(vd, 0, sizeof(*vd));
    vd->var_name = JS_DupAtom(ctx, name);
    return fd->var_count - 1;
}
```

```
int js_parse_postfix_expr(JSParseState *s,
                          BOOL accept_lparen)
{
    switch(s->token.val) {
        case TOK_IDENT:
            name = JS_DupAtom(s->ctx, s->token.u.ident.atom);
            emit_op(s, OP_scope_get_var);
            emit_u32(s, name);
            emit_u16(s, s->cur_func->scope_level);
            break;
    }
    return 0;
}
```

确定哪些变量是自由变量

get_closure_var会给s和fd之间的函数都添加自由变量

```
int resolve_scope_var(JSContext *ctx, JSFunctionDef *s,
                      JSAtom var_name, int scope_level, int op,
                      DynBuf *bc, uint8_t *bc_buf,
                      LabelSlot *ls, int pos_next, int arg_valid)
{
    /* ... */
    /* 当前函数中没有找到变量，则在父函数中寻找 */
    for (fd = s; fd->parent;) {
        scope_level = fd->parent_scope_level;
        fd = fd->parent;
        for (idx = fd->scopes[scope_level].first; idx >= 0;) {
            vd = &fd->vars[idx];
            if (vd->var_name == var_name) {
                var_idx = idx;
                break;
            }
            idx = vd->scope_next;
        }
        if (var_idx >= 0)
            break;

        var_idx = find_var(ctx, fd, var_name);
        if (var_idx >= 0) {
            break;
        }
    }
    if (var_idx >= 0) {
        fd->vars[var_idx].is_captured = 1;
        idx = get_closure_var(ctx, s, fd,
                              FALSE, var_idx,
                              var_name,
                              fd->vars[var_idx].is_const,
                              fd->vars[var_idx].is_lexical,
                              fd->vars[var_idx].var_kind);
        if (idx >= 0) {
            switch (op) {
                case OP_scope_get_var:
                    dbuf_putc(bc, OP_get_var_ref);
                    dbuf_put_u16(bc, idx);
                    break;
            }
            goto done;
        }
    }
    /* ... */
    done:
    return pos_next;
}
```

创建函数闭包

直接从父函数的var_refs中获取

```
JSValue js_closure2(JSContext *ctx, JSValue func_obj,
                    JSFunctionBytecode *b,
                    JSVarRef **cur_var_refs,
                    JSStackFrame *sf) /* 表示sf代表的函数用到的自由变量 */
{
    JSObject *p;
    JSVarRef **var_refs;
    int i;

    p = JS_VALUE_GET_OBJ(func_obj);
    p->u.func.function_bytecode = b;
    p->u.func.var_refs = NULL;
    if (b->closure_var_count) {
        var_refs = js_mallocz(ctx, sizeof(var_refs[0]) * b->closure_var_count);
        p->u.func.var_refs = var_refs; /* 代表当前函数使用到的自由变量引用列表 */
        for(i = 0; i < b->closure_var_count; i++) {
            JSClosureVar *cv = &b->closure_var[i];
            JSVarRef *var_ref;
            if (cv->is_local) {
                /* 自由变量来自父函数 */
                var_ref = get_var_ref(ctx, sf, cv->var_idx, cv->is_arg);
            } else {
                /* 说明自由变量来自更外层 */
                var_ref = cur_var_refs[cv->var_idx];
                var_ref->header.ref_count++;
            }
            var_refs[i] = var_ref;
        }
    }
    return func_obj;
}
```

闭包变量获取和修改

通过var_refs获取最终value

修改自由变量

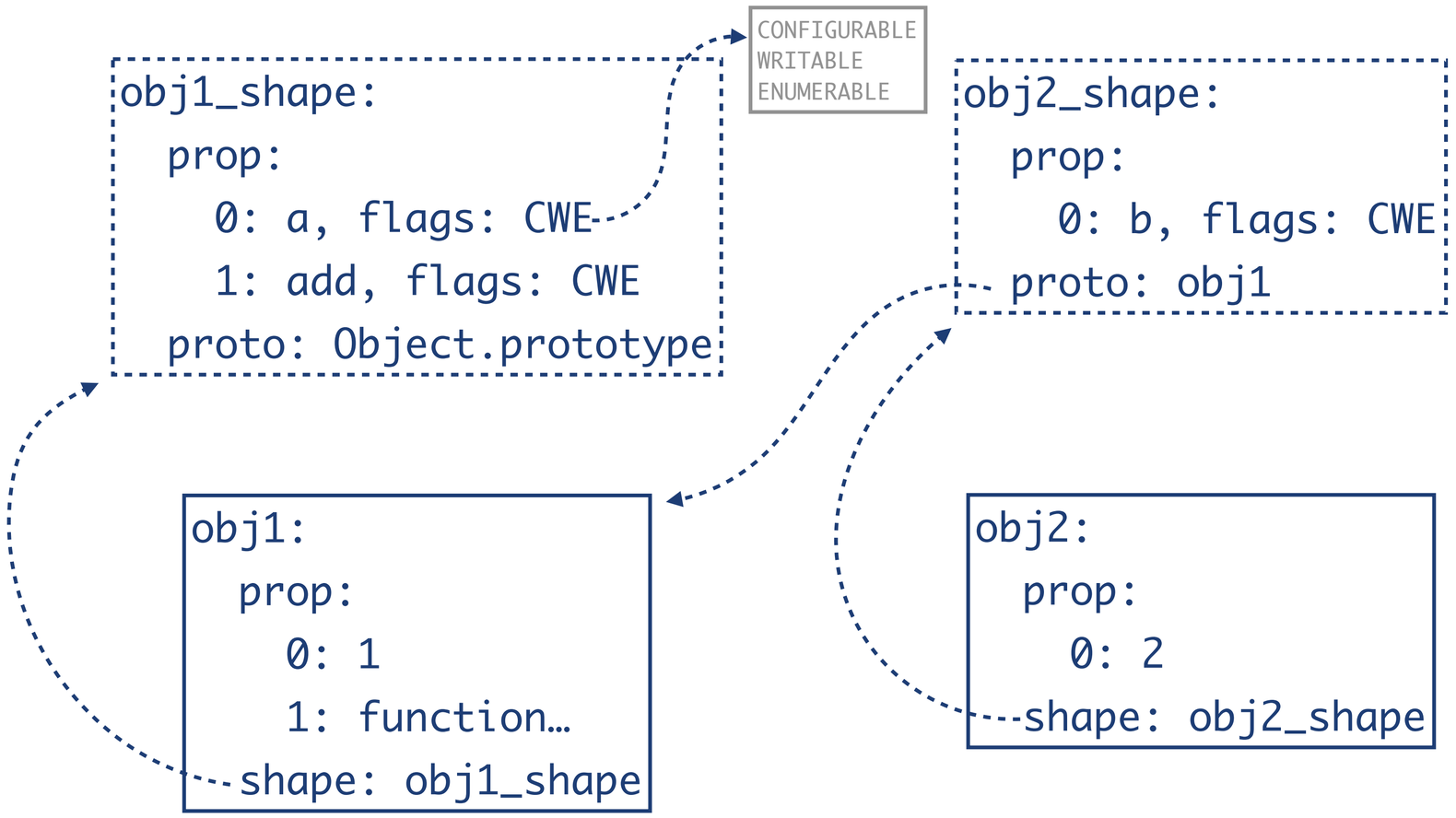
```
JSValue JS_CallInternal(JSContext *ctx, JSValueConst func_obj,
                        JSValueConst this_obj, JSValueConst new_target,
                        int argc, JSValue *argv, int flags)
{
    JSObject *p;
    JSVarRef **var_refs;

    p = JS_VALUE_GET_OBJ(func_obj);
    var_refs = p->u.func.var_refs;

    for(;;) {
        SWITCH(pc) {
        CASE(OP_get_var_ref):
            int idx;
            JSValue val;
            idx = get_u16(pc);
            pc += 2;
            val = *var_refs[idx]->pvalue;
            sp[0] = JS_DupValue(ctx, val);
            sp++;
            BREAK;
        CASE(OP_put_var_ref):
            int idx;
            idx = get_u16(pc);
            pc += 2;
            set_value(ctx, var_refs[idx]->pvalue, sp[-1]);
            sp--;
            BREAK;
        }
    }
}
```

2. 原型链的实现原理

什么是原型链?

```
let obj1 = {
  a: 1,
  add: function () {
    return this.a + this.b;
  }
};
let obj2 = {
  b: 2
};
// 设置obj2的原型为obj1
Object.setPrototypeOf(obj2, obj1);
obj2.add(); // 输出为: 3
```

Object与Shape的关系



数据结构

```
struct JSObject {
    JSShape *shape;
    JSProperty *prop;
    ...
};

struct JSShape {
    JSObject *proto;
    JSShapeProperty *prop;
    ...
};
```

创建Shape

设置原型对象

```
JSShape *js_new_shape2(JSContext *ctx, JSObject *proto,
                                              int hash_size, int prop_size)
{
    JSRuntime *rt = ctx->rt;
    void *sh_alloc;
    JSShape *sh

    sh_alloc = js_malloc(ctx, get_shape_size(hash_size, prop_size));
    sh = get_shape_from_alloc(sh_alloc, hash_size);
    sh->header.ref_count = 1;
    sh->proto = proto;
    memset(sh->prop_hash_end - hash_size, 0, sizeof(sh->prop_hash_end[0]) *
           hash_size);
    sh->prop_hash_mask = hash_size - 1;
    sh->prop_count = 0;
    sh->prop_size = prop_size;

    /* insert in the hash table */
    sh->hash = shape_initial_hash(proto);
    sh->is_hashed = TRUE;
    sh->has_small_array_index = FALSE;
    js_shape_hash_link(ctx->rt, sh);
    return sh;
}
```

创建对象

设置对象的shape

```
JSValue JS_NewObjectFromShape(JSContext *ctx, JSShape *sh, JSClassID class_id)
{
    JSObject *p;

    js_trigger_gc(ctx->rt, sizeof(JSObject));
    p = js_malloc(ctx, sizeof(JSObject));

    p->shape = sh;
    p->prop = js_malloc(ctx, sizeof(JSProperty) * sh->prop_size);

    p->header.ref_count = 1;
    add_gc_object(ctx->rt, &p->header, JS_GC_OBJ_TYPE_JS_OBJECT);
    return JS_MKPTR(JS_TAG_OBJECT, p);
}
```

获取对象属性

当前对象上没找到的话
就去原型对象上寻找

```
JSValue JS_GetPropertyInternal(JSContext *ctx, JSValueConst obj,
                               JSAtom prop, JSValueConst this_obj,
                               BOOL throw_ref_error)
{
    JSObject *p;
    JSProperty *pr;
    JSShapeProperty *prs;

    p = JS_VALUE_GET_OBJ(obj);
    for(;;) {
        prs = find_own_property(&pr, p, prop);
        if (prs) {
            return JS_DupValue(ctx, pr->u.value);
        }
        p = p->shape->proto;
        if (!p)
            break;
    }

    return JS_UNDEFINED;
}
```

3. 垃圾回收的实现原理

下面的几个对象会在什么时候被回收?

```
let f1 = () => {
    let obj1 = {}
    return;
}
let f2 = () => {
    let obj1 = {};
    let obj2 = {obj1: obj1};
    obj1.obj2 = obj2;
    return;
}

f1();
f2();
```

对象之间互相引用怎么解决?

垃圾回收时机

① 函数返回或者重新赋值，会对可访问的对象的引用计数建一，如果发现为0了就则销毁

② 对于对象的互相引用的销毁，则是在创建新对象的时候。超过某个阈值会触发GC检测。GC会给对象的属性进行解引用。通过不断的解引用之后，如果最终的引用计数都为0，则说明这个环可以被安全的移除

```
void JS_RunGC(JSRuntime *rt)
{
    /* 遍历gc_obj_list中的对象，
       将每个对象的属性引用计数减1
       减完之后如果计数为0，则存放到
       temp_obj_list中 */
    gc_decref(rt);

    /* 再次变量gc_obj_list中的对象
       将每个对象的属性引用计数加1
       如果属性的引用计数等于1，
       则从temp_obj_list中移除，
       添加回gc_obj_list。并且后面还会
       继续检查它的属性 */
    gc_scan(rt);

    /* 释放还存在于temp_obj_list的对象 */
    gc_free_cycles(rt);
}
```

引用计数减一

```
void gc_decref(JSRuntime *rt)
{
    struct list_head *el, *el1;
    JSGCObjectHeader *p;

    init_list_head(&rt->tmp_obj_list);

    list_for_each_safe(el, el1, &rt->gc_obj_list) {
        p = list_entry(el, JSGCObjectHeader, link);
        assert(p->mark == 0);
        mark_children(rt, p, gc_decref_child);
        p->mark = 1;
        if (p->ref_count == 0) {
            list_del(&p->link);
            list_add_tail(&p->link, &rt->tmp_obj_list);
        }
    }
}
```

```
void gc_decref_child(JSRuntime *rt,
                     JSGCObjectHeader *p)
{
    assert(p->ref_count > 0);
    p->ref_count--;
    if (p->ref_count == 0 && p->mark == 1) {
        list_del(&p->link);
        list_add_tail(&p->link, &rt->tmp_obj_list);
    }
}
```

引用计数加一

```c
void gc_scan(JSRuntime *rt)
{
    struct list_head *el;
    JSGCObjectHeader *p;

    /* */
    list_for_each(el, &rt->gc_obj_list) {
        p = list_entry(el, JSGCObjectHeader, link);
        assert(p->ref_count > 0);
        p->mark = 0;
        /* 注意这里有可能会添加元素到gc_obj_list中 */
        mark_children(rt, p, gc_scan_incref_child);
    }
}
```

```c
void gc_scan_incref_child(JSRuntime *rt,
                          JSGCObjectHeader *p)
{
    p->ref_count++;
    if (p->ref_count == 1) {
        list_del(&p->link);
        list_add_tail(&p->link, &rt->gc_obj_list);
        p->mark = 0;
    }
}
```

添加完之后，后面将会被遍历到

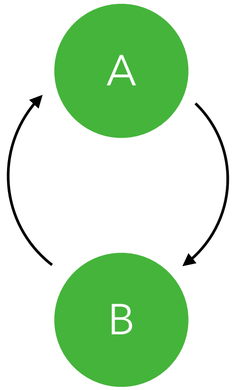
去环成功的例子

```
let a = {b: undefined};
let b = {a: a};
a.b = b;
```

存放在gc_obj_list

存放在temp_obj_list

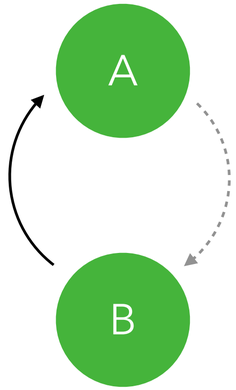
gc_decref



引用计数

A: 1

B: 1

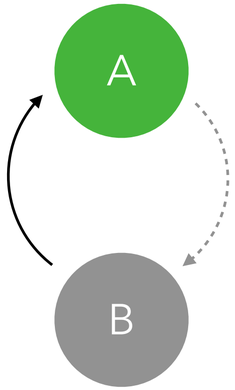


引用计数

A: 1

B: 0

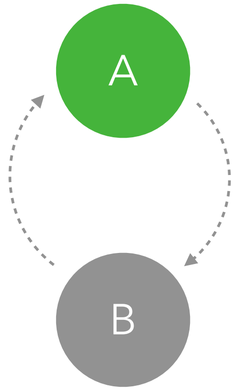
B将移动到temp_obj_list



引用计数

A: 1

B: 0

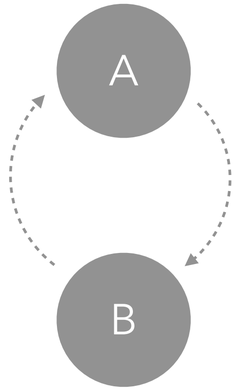


引用计数

A: 0

B: 0

A将移动到temp_obj_list



引用计数

A: 0

B: 0

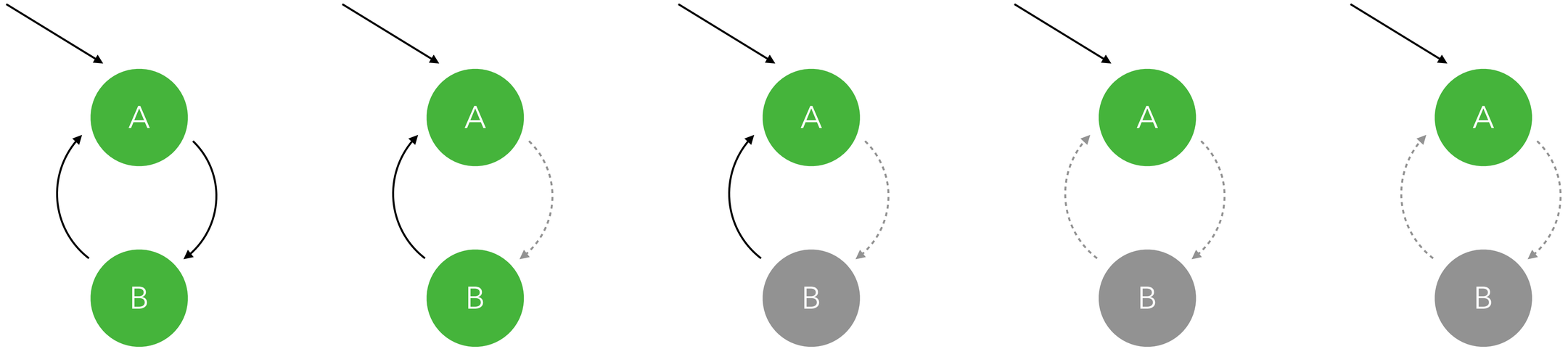
去环失败的例子

```
let a = {b: undefined};
let b = {a: a};
a.b = b;
global.a = a;
```

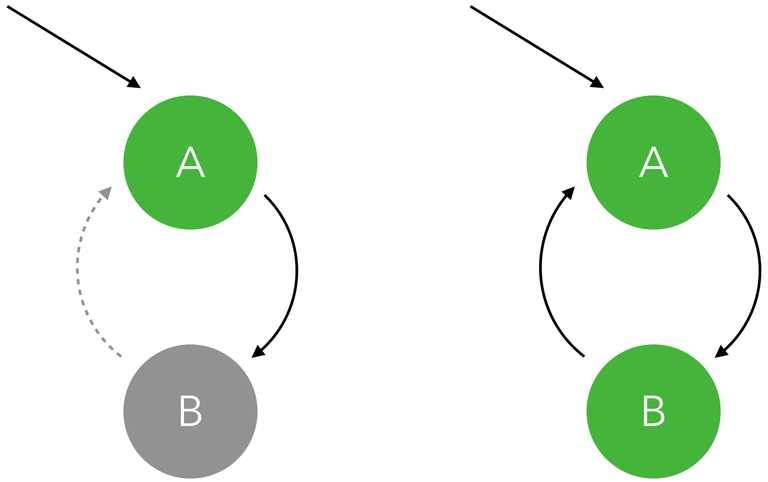
存放在gc_obj_list

存放在temp_obj_list

gc_decref



gc_scan



因为A还被其他地方引用，不能被移动到temp_obj_list
而移过去的B因为A的引用又给添加回来了

待改进的点

- 粒度不够，函数中的块级作用域中的变量需要等到整个函数退出时才有机会被回收。实际上可以在块级作用域结束后就进行回收了。
- 环形对象的回收检查，需要等待申请新对象时才会进行，即使该环形对象只存在于一个函数中。如果在函数退出的时候先进行局部变量的回收，对于回收不了的再进行环形对象探测，会让对象的回收时机提前。

Thank You

Q&A

PPT&代码注释可前往：https://gitee.com/lhtin/quickjs