

QuickJS 源码解读（二）：基础设施和标准库

基础设施

标准库

Posted at 2020-12-26

console.log

上一篇文章 里面主要解释了 QuickJS 虚拟机的运作。第二篇文章打算介绍一下 QuickJS 里面 JavaScript 基础设施的实现。

基础设施

注意，使用 QuickJS 新建 JSContext 的时候，默认是不带基础设施的 (比如说 JSON 解析、Object、等等)。这时候，可以调用以下命令进行添加，这些内建对象的支持都是内置的：

```
void JS_AddIntrinsicBaseObjects(JSContext *ctx);
void JS_AddIntrinsicDate(JSContext *ctx);
void JS_AddIntrinsicEval(JSContext *ctx);
void JS_AddIntrinsicStringNormalize(JSContext *ctx);
void JS_AddIntrinsicRegExpCompiler(JSContext *ctx);
void JS_AddIntrinsicRegExp(JSContext *ctx);
void JS_AddIntrinsicJSON(JSContext *ctx);
void JS_AddIntrinsicProxy(JSContext *ctx);
void JS_AddIntrinsicMapSet(JSContext *ctx);
void JS_AddIntrinsicTypedArrays(JSContext *ctx);
void JS_AddIntrinsicPromise(JSContext *ctx);
void JS_AddIntrinsicBigInt(JSContext *ctx);
void JS_AddIntrinsicBigFloat(JSContext *ctx);
void JS_AddIntrinsicBigDecimal(JSContext *ctx);
```

至于这些内置方法，比如 Object 是怎么实现，我们可以直接点开 JS_AddIntrinsicBaseObjects 代码

```
/* Object */
obj = JS_NewGlobalCConstructor(ctx, "Object", js_object_construct,
                               ctx->class_proto[JS_CLASS_OBJECT])
JS_SetPropertyFunctionList(ctx, obj, js_object_funcs, countof(js_
```

```
JS_SetPropertyFunctionList(ctx, ctx->class_proto[JS_CLASS_OBJECT]
                           js_object_proto_funcs, countof(js_obj
```

基础设施

这两行分别给 `Object` 和 `Object.prototype` 设置了相应的函数，函数的定义在 `js_object_proto_funcs` 这个静态变量里面。

```
js_object_proto_funcs
```

我们可以跟踪看看 `js_object_proto_funcs` 分别定义了什么函数：

```
static const JSCFunctionListEntry js_object_proto_funcs[] = {
    JS_CFUNC_DEF("toString", 0, js_object_toString ),
    JS_CFUNC_DEF("toLocaleString", 0, js_object_toLocaleString ),
    JS_CFUNC_DEF("valueOf", 0, js_object_valueOf ),
    JS_CFUNC_DEF("hasOwnProperty", 1, js_object_hasOwnProperty ),
    JS_CFUNC_DEF("isPrototypeOf", 1, js_object_isPrototypeOf ),
    JS_CFUNC_DEF("propertyIsEnumerable", 1, js_object_propertyIsE
JS_CGETSET_DEF("__proto__", js_object_get__proto__, js_objec
JS_CFUNC_MAGIC_DEF("__defineGetter__", 2, js_object__defineG
JS_CFUNC_MAGIC_DEF("__defineSetter__", 2, js_object__defineG
JS_CFUNC_MAGIC_DEF("__lookupGetter__", 1, js_object__lookupG
JS_CFUNC_MAGIC_DEF("__lookupSetter__", 1, js_object__lookupG
};
```

可以看到这些内置方法都是 C 实现的方法。对应的函数都是在 `quickjs.c` 里面定义。

知道了这些，我们就可以动手修改 QuickJS 代码，比如说我们想改变 `Object` 的 `toString` 的表现。让它输出更详细的信息，那我们更改 `js_object_toString` 的实现就可以了。

标准库

标准库的实现，不是语言的一部分。这一部分的实现内容放在了 `quickjs-libc.c` 这个文件。这里稍微提一句，Bellard 实现标准库用了不少 POSIX 方法，这样实现起来代码会比较简单，但是同时导致了代码无法在 Windows 上编译（除非用 MingW），所以想要在 Windows 上独立编译通过的话，需要做不少改动。

我自己也做了一份 Fork, 修复了 Windows 的编译问题, 同时支持 CMake:
<https://github.com/vincentdchan/quickjs>

需要的可以自取。

标准库

`console.log`
`setTimeout`

写过 JS 的人估计都用过 `console.log` 吧。那么在 QuickJS 里面, `console.log` 怎么实现呢。答案就在 `quickjs-libc.c` 这个文件的 `js_std_add_helpers` 这个函数里面:

```
JSValue global_obj, console, args;
int i;

global_obj = JS_GetGlobalObject(ctx);
console = JS_NewObject(ctx);
JS_SetPropertyStr(ctx, console, "log",
                  JS_NewCFunction(ctx, js_print, "log", 1));
JS_SetPropertyStr(ctx, global_obj, "console", console);
```

可以看到实现方法就是 `js_print` 这个函数:

```
static JSValue js_print(JSContext *ctx, JSValueConst this_val,
                        int argc, JSValueConst *argv)
{
    int i;
    const char *str;

    for(i = 0; i < argc; i++) {
        if (i != 0)
            putchar(' ');
        str = JS_ToCString(ctx, argv[i]);
        if (!str)
            return JS_EXCEPTION;
        fputs(str, stdout);
        JS_FreeCString(ctx, str);
    }
}
```

```

    putchar('\n');
    return JS_UNDEFINED;
}

```

基础设施

我们可以看到底层调用的是 C 语言的 `fputs` 方法输出到 `stdout`。如果你想让 `console.log` 输出到自己的文件，或者数据库，那么你就可以更改 `js_print` 这个方法了。

我们还可以看到，Bellard 只实现了 `console.log`，但是没有输出 `console.error`。那么我们就可以把 `console.error` 给实现上，输出到 `stderr`，也是轻而易举了。

setTimeout

`setTimeout` 的实现会稍微复杂一点。要了解 `setTimeout` 的实现，就要了解 QuickJS 的事件循环。要了解 QuickJS 的事件循环，其实只要看懂一个函数：

```

/* main loop which calls the user JS callbacks */
void js_std_loop(JSContext *ctx)
{
    JSContext *ctx1;
    int err;

    for(;;) {
        /* execute the pending jobs */
        for(;;) {
            err = JS_ExecutePendingJob(JS_GetRuntime(ctx), &ctx1);
            if (err <= 0) {
                if (err < 0) {
                    js_std_dump_error(ctx1);
                }
                break;
            }
        }

        if (!os_poll_func || os_poll_func(ctx))
            break;
    }
}

```

QuickJS 里面维护着一个队列。而主循环就是一一直从这个队列里面捞任务出来进行处理。JS_ExecutePendingJob 就是执行一个 JS 任务。这个 JS 任务可能添加了一个系统任务。比如如果代码调用了 setTimeout 那么 QuickJS 会往系统添加一个定时器任务。

随后这个循环调用 os_poll_func 这个方法，会一直阻塞，等到有任务完成，这个函数会往 QuickJS 队列添加一个回调，然后返回。

返回后进入下一次循环，就会执行 setTimeout 的回调，这样就完成了 setTimeout 的调用。


这里和我们熟知的 v8 不一样，v8 使用的是 libuv 作为事件循环的库。而 QuickJS 为了轻量化，简单的封装了一下系统的信号，有兴趣的同学可以深入了解 os_poll_func 函数的实现。


在 VINCENT'S PERSONAL BLOG 上还有


<div><div>在 Mojave 下编译 SpiderMonkey</div><div>3 年前 · 4条评论</div><div>Vincent Chan's Bus Station. Blog of Vincent.</div></div>	<div><div>多线程 SQLite with C++ 踩坑汇总</div><div>2 年前 · 3条评论</div><div>Vincent Chan's Bus Station. Blog of Vincent.</div></div>	<div><div>QuickJS 源码解读 (一)</div><div>2 年前 · 1条评论</div><div>Vincent Chan's Bus Station. Blog of Vincent.</div></div>	<div><div>使用 Jav</div><div>2 年</div><div>Vinc Blog</div></div>
--	---	---	--

What do you think?

0条回复

 Upvote

 Funny

 Sad

基础设施
标准库

[console.log](#)
[0条评论](#)
[setTimeout](#)

Vincent's personal blog

 Disqus [隐私政策](#)

 1 登录 ▾

 推荐  推文  分享

评分最高 ▾



开始讨论...

通过以下方式登录

或注册一个 DISQUS 帐号 

姓名

来做第一个留言的人吧！

 订阅  在您的网站上使用 Disqus添加 Disqus添加  不要出售我的数据

© 2019, Built with [Gatsby](#)