No.js---基于V8和io_uring的JS运行时

theanarkh

已关注

前言：阅读Node.js的源码已经有一段时间了，最近也看了一下新的JS运行时Just的一些实现，就产生了自己写一个JS运行时的想法，虽然几个月前就基于V8写了一个简单的JS运行时，但功能比较简单，这次废弃了之前的代码，重新写了一遍，写这个JS运行时的目的最主要是为了学习，事实也证明，写一个JS运行时的确可以学到很多东西。本文介绍运行时No.js的一些设计和实现，取名No.js一来是受Node.js的影响，二来是为了说明不仅仅是JS，也就是利用V8拓展了JS的功能，同时，前端开发者要学习的知识也不仅仅是JS了。

1 为什么选io_uring

io_uring是Linux下新一代的高性能异步IO框架，也是No.js的核心。在No.js中，io_uring用于实现事件循环。为什么不选用epoll呢？因为epoll不支持文件IO，如果选用epoll，还需要自己实现一个线程池，还需要实现线程和主线程的通信，以及线程池任务和事件循环的融合，No.js希望把事件变得纯粹，简单。而io_uring是支持异步文件IO的，并且io_uring是真正的异步IO框架，支持的功能也非常丰富，比如在epoll里我们监听一个socket后，需要把socket fd注册到epoll中，等待有连接时执行回调，然后调用accept获取新的fd，而io_uring直接就帮我们获取新的fd，io_uring通知我们的时候，我们就已经拿到新的fd了，epoll时代，epoll通知我们可以做什么事情了，然后我们自己去做，io_uring时代，io_uring通知我们什么事情完成了。

2 No.js框架的设计

No.js目前的实现比较清晰简单，所有的功能都通过c和c++实现，然后通过V8暴露给JS实现。No.cc是初始化的入口，core目录是所有功能实现的地方，core下面按照模块功能划分。下面我们看看整体的框架实现。

```
int main(int argc, char* argv[]) {
  // ...
  Isolate* isolate = Isolate::New(create_params);
  {
    Isolate::Scope isolate_scope(isolate);
    HandleScope handle_scope(isolate);
    // 创建全局对象✦
    Local<ObjectTemplate> global = ObjectTemplate::New(isolate);
    // 创建执行上下文
    Local<Context> context = Context::New(isolate, nullptr, global);
    Environment * env = new Environment(context);
    Context::Scope context_scope(context);
    // 创建No，核心对象
    Local<Object> No = Object::New(isolate);
    // 注册c、c++模块
    register_builtins(isolate, No);
    // 获取全局对象
    Local<Object> globalInstance = context->Global();
    // 设置全局属性
    globalInstance->Set(context, String::NewFromUtf8Literal(isolate, "No",
    NewStringType::kNormal), No);
    // 设置全局属性global指向全局对象
    globalInstance->Set(context, String::NewFromUtf8Literal(isolate,
      "global",
      NewStringType::kNormal), globalInstance).Check();
    {
      // 打开文件
      int fd = open(argv[1], O_RDONLY);
      struct stat info;
      // 取得文件信息
      fstat(fd, &info);
      // 分配内存保存文件内容
      char *ptr = (char *)malloc(info.st_size + 1);
      read(fd, (void *)ptr, info.st_size);
      // 要执行的js代码✦
      Local<String> source = String::NewFromUtf8(isolate, ptr,
                             NewStringType::kNormal,
                             info.st_size).ToLocalChecked();

      // 编译
      Local<Script> script = Script::Compile(context, source).ToLocalChecked();
      // 解析完应该没用了，释放内存
      free(ptr);
      // 执行
      Local<Value> result = script->Run(context).ToLocalChecked();
      // 进入事件循环
      Run(env->GetIOUringData());
    }
}
```

```
  return 0;
}
```

大部分逻辑都是V8初始化的标准流程，添加的内容主要包括注册c、c++模块、挂载No到全局作用域、开启事件循环。

2.1 注册模块

No在初始化的时候会把所有C++模块注册到No中，因为No是全局属性，所以在JS里可以直接访问C++模块，不需要require。我们看看register_builtins。

```
void No::Core::register_builtins(Isolate * isolate, Local<Object> target) {
    FS::Init(isolate, target);
    TCP::Init(isolate, target);
    Process::Init(isolate, target);
    Console::Init(isolate, target);
    IO::Init(isolate, target);
    Net::Init(isolate, target);
    UDP::Init(isolate, target);
    UNIX_DOMAIN::Init(isolate, target);
    Signal::Init(isolate, target);
    Timer::Init(isolate, target);
}
```

register_builtins会调用各个模块的Init函数，各个模块自己实现需要挂载的功能，从代码中可以看到目前实现的功能。我们随便找一个模块看看初始化的逻辑。

```
void No::FS::Init(Isolate* isolate, Local<Object> target) {
  Local<ObjectTemplate> fs = ObjectTemplate::New(isolate);
  setMethod(isolate, fs, "open", No::FS::Open);
  setMethod(isolate, fs, "openat", No::FS::OpenAt);
  setMethod(isolate, fs, "close", No::IO::Close);
  setMethod(isolate, fs, "read", No::IO::Read);
  setMethod(isolate, fs, "write", No::IO::Write);
  setMethod(isolate, fs, "readv", No::IO::ReadV);
  setMethod(isolate, fs, "writev", No::IO::WriteV);
  setObjectValue(isolate, target, "fs", fs->NewInstance(isolate->GetCurrentCont
}
```

挂载的逻辑就是新建一个对象，然后设置对象的属性，最后把这个对象作为No对象的一个属性挂载到No中，最后形成如下一个结构。

```
var No = {
    fs: {},
    tcp: {}
}
```

这就完成了所有核心模块的注册。

2.2 执行JS

注册完核心模块后就是执行业务JS。我们随便看个例子。

```
const {
    fs,
    console
} = No;
const fd = fs.open('./test/file/1.txt');
const arr = new ArrayBuffer(100);
fs.readv(fd,arr , 0, (res) => {console.log(res)});
console.log(new Uint8Array(arr));
```

以上是读取一个文件的例子，从中也可以看到No的使用方式。No没有实现类似Node.js的Buffer，是直接使用V8的ArrayBuffer的，ArrayBuffer使用的是V8堆外内存，readv是C++层实现的函数，我们一会单独介绍。

2.3 开启事件循环

执行完JS后，最后进入事件循环。

```
void No::io_uring::RunIOUring(struct io_uring_info *io_uring_data) {
    struct io_uring* ring = &io_uring_data->ring;
    struct io_uring_cqe* cqe;
    struct request* req;
    while(io_uring_data->stop != 1 && io_uring_data->pending != 0) {
        // 提交请求给内核
        int count = io_uring_submit_and_wait(ring, 1);
        // 处理每一个完成的请求
        while (1) {
            io_uring_peek_cqe(ring, &cqe);
            if (cqe == NULL)
                break;
            --io_uring_data->pending;
            // 拿到请求上下文
            req = (struct request*) (uintptr_t) cqe->user_data;
```

```
                req->res = cqe->res;
                io_uring_cq_advance(ring, 1);
                // 执行回调
                if (req->cb != nullptr) {
                    req->cb((void *)req);
                }
            }
        }
}
```

从事件循环的代码中大致可以看到原理，首先判断事件循环是不是停止或者可以停止了，如果还没有停止，则等待任务完成，然后取出任务执行任务的对象。

3 任务的封装和处理

io_uring的任务是以结构体io_uring_sqe表示的，但是io_uring_sqe只是记录了和io_uring框架本身相关的一些数据结构，因为是异步的模式，所以在任务完成的时候，我们需要知道，这个任务关联的上下文和回调。io_uring_sqe提供了user_data字段用于保存请求对应的上下文。流程如下。 设置和提交请求

```
// 获取一个io_uring的请求结构体
 struct io_uring_sqe *sqe = io_uring_get_sqe(&io_uring_data->ring);
 // 自定义结构体
 struct io_request * file_req = (struct io_request *)req;
 // 设置请求的字段
 io_uring_prep_read(sqe, file_req->fd, file_req->buf, file_req->len, file_req->
 // 保存请求上下文，响应的时候用
 io_uring_sqe_set_data(sqe, (void *)req);
 // 提交请求
 io_uring_submit(&io_uring_data->ring);
```

我们看到提交请求的时候，设置了请求上下文是我们自定义的结构体，具体结构体类型✦根据操作类型而不同。我们看看请求完成时是如何处理的。

```
struct io_uring_cqe* cqe;
io_uring_peek_cqe(ring, &cqe);
// 拿到请求上下文
req = (struct request*) (uintptr_t) cqe->user_data;
// 记录请求结果
req->res = cqe->res;
req->cb((void *)req);
```

以上就是一个No请求和响应的处理过程。No为不同的操作类型封装了不同的结构体。首先封装了一个请求的基类。

```
#define REQUEST \
        int op; \
        // io_uring执行的回调
        request_cb cb; \
        // io_uring请求的结果
        int res;\
        // 业务上下文
        void * data; \
        int flag;
```

类似io_uring通过user_data字段关联请求响应上下文。REQUEST 里通过data关联请求和响应上下文，通过user_data字段，我们在任务完成时可以执行应该执行哪个回调以及对应的上下文。但是执行某个回调时，该回调函数✦需要的上下文可能不仅仅是io_uring返回的结果，这时候就可以使用data字段记录额外的上下文。一会会具体介绍。基于REQUEST，针对不同的操作封装了不同的结构体，比如文件请求。

```
struct io_request {
    REQUEST
    int fd;
    int offset;
    void *buf;
    int len;
};
```

下面我们分析一个具体请求的过程，这里以read为例。

```
void read_write_request(V8_ARGS, int op) {
    V8_ISOLATE
    int fd = args[0].As<Uint32>()->Value();
    int offset = 0;
    if (args.Length() > 2 && args[2]->IsNumber()) {
        offset = args[2].As<Integer>()->Value();
    }
    Local<ArrayBuffer> arrayBuffer = args[1].As<ArrayBuffer>();
    std::shared_ptr<BackingStore> backing = arrayBuffer->GetBackingStore();
    V8_CONTEXT
    Environment *env = Environment::GetEnvByContext(context);
    struct io_uring_info *io_uring_data = env->GetIOUringData();
    struct request *req;
    // 文件操作对应的request结构体
    struct io_request *io_req = (struct io_request *)malloc(sizeof(struct io_re
    memset(io_req, 0, sizeof(*io_req));
    io_req->buf = backing->Data();
    io_req->len = backing->ByteLength();
    io_req->fd = fd;
    io_req->offset = offset;
```

```
    req = (struct request *)io_req;
    // JS层回调
    req->cb = makeCallback<onread>;
    req->op = op;
    // 保存回调上下文
    if (args.Length() > 3 && args[3]->IsFunction()) {
        Local<Object> obj = Object::New(isolate);
        Local<String> key = newStringToLcal(isolate, onread);
        obj->Set(context, key, args[3].As<Function>());
        req->data = (void *)new RequestContext(env, obj);
    } else {
        req->data = (void *)new RequestContext(env, Local<Function>());
    }
    // 提交请求
    SubmitRequest((struct request *)req, io_uring_data);
}
```

初始化请求的上下文后，调用SubmitRequest提交任务和io_uring。我们看看SubmitRequest。

```
void No::io_uring::SubmitRequest(struct request * req, struct io_uring_info *io
    // 获取一个io_uring的请求结构体
    struct io_uring_sqe *sqe = io_uring_get_sqe(&io_uring_data->ring);
    // 填充请求
    switch (req->op)
    {

        case IORING_OP_READ:
            {
                struct io_request * file_req = (struct io_request *)req;
                io_uring_prep_read(sqe, file_req->fd, file_req->buf, file_req->
                break;
            }
        default:
            return;
    }
    ++io_uring_data->pending;
    // 保存请求上下文，响应的时候用
    io_uring_sqe_set_data(sqe, (void *)req);
    io_uring_submit(&io_uring_data->ring);
}
```

SubmitRequest根据不同的操作设置io_uring的请求结构体，并保存对应的请求上下文。当任务完成时执行回调makeCallback。makeCallback是模板函数。

```
template <const char * event>
  void makeCallback(void * req) {
       struct request * _req = (struct request *)req;
       RequestContext* ctx =(RequestContext *)_req->data;
       if (!ctx->object.IsEmpty()) {
           Local<Object> object = ctx->object.Get(ctx->env->GetIsolate());
           Local<Value> cb;
           Local<Context> context = ctx->env->GetContext();
           Local<String> onevent = newStringToLcal(ctx->env->GetIsolate(), ever
           object->Get(context, onevent).ToLocal(&cb);
           if (cb->IsFunction()) {
               Local<Value> argv[] = {
                   Integer::New(context->GetIsolate(), _req->res)
               };
               // 执行JS层回调
               cb.As<v8::Function>()->Call(context, object, 1, argv);
           }
       }
  };
```

makeCallback做的事情就是执行JS回调。

4 非io_uring的处理

io_uring目前已经支持了非常多的操作，但我们也不可避免地会碰到io_uring不支持的操作，比如信号的处理。No里目前定时器和信号不是使用io_uring处理的。定时器目前使用内核的posix timer实现的，io_uring有个timeout类型的请求，后续可能会使用io_uring的，信号处理io_uring就无能无力了。因为No是单线程的架构，所以非io_uring的任务完成后也需要通过io_uring事件循环执行，下面看一下非io_uring支持的操作如何处理的。在业务里，我们可能需要监听一个信号。

```
const {
    signal,
    console,
    process,
    timer,
} = No;

signal.on(signal.constant.SIG.SIGUSR1, () => {
    process.exit();
});

// for keep process alive
timer.setInterval(() => {},10000, 10000);
```

可以通过signal模块的on实现监听信号，接下来看看具体实现。

```
void No::Signal::RegisterSignal(V8_ARGS) {
    V8_ISOLATE
    V8_CONTEXT
    Environment *env = Environment::GetEnvByContext(context);
    Local<Object> obj = Object::New(isolate);
    Local<String> key = newStringToLcal(isolate, onsignal);
    obj->Set(context, key, args[1].As<Function>());
    int sig = args[0].As<Integer>()->Value();
    // 新建一个上下文
    shared_ptr<SignalRequestContext> ctx = make_shared<SignalRequestContext>(en
    auto ret = signalMap.find(sig);
    // 是否在map里，不是则新建一个vector，否则直接追加
    if (ret == signalMap.end()) {
        signal(sig, signalHandler);
        vector<shared_ptr<SignalRequestContext>> vec;
        vec.push_back(ctx);
        signalMap.insert(map<int, vector<shared_ptr<SignalRequestContext>>>::va
        return;
    }
    ret->second.push_back(ctx);
}
```

No使用一个map管理信号和监听函数，因为支持多个监听函数，所以map的key是信号的值，value是一个回调函数数组。如果是第一次注册该信号，则调用signal注册该信号的处理函数，所有信号的处理函数都是signalHandler。接着看信号产生时的处理逻辑。

```
static void signalHandler(int signum)
{
    auto vec = signalMap.find(signum);
    if (vec != signalMap.end()) {
        vector<shared_ptr<SignalRequestContext>>::iterator it;
        for(it=vec->second.begin();it!=vec->second.end(); it++)
        {
            struct signal_request * req = (struct signal_request *)malloc(sizeo
            memset(req, 0, sizeof(*req));
            req->cb = signal_cb;
            req->data = (void *)(*it).get();
            req->op = IORING_OP_NOP;
            SubmitRequest((struct request *)req, (*it).get()->env->GetIOUringDa
        }
    }
}
```

信号产生时，从map中找到对应的处理函数列表，然后生成一个io_uring请求，这样在事件循环时就会被执行，也实现了非io_uring任务和io_uring任务的整合，这里主要是利用了io_uring提供了nop类型的请求，这个类型的请求不做任何操作，主要是用于测试io_uring请求和响应链路，利用这点恰好可以实现我们的需求。从代码中可以看到io_uring事件循环时会执行信号处理的回调

signal_cb，signal_cb会回调JS层。

5 上下文的设计

因为No各种请求都是异步的，所以避免不了需要保持请求和响应的上下文。类似Node.js，No里也存在一个env作为整个进程级的上下文。

```
enum {
    CONTEXT_INDEX
} ENV_INDEX;

class Environment {
    public:
        Environment(Local<Context> context);
        static Environment * GetEnvByContext(Local<Context> context);
        struct io_uring_info * GetIOUringData() {
            return io_uring_data;
        }
        Isolate * GetIsolate() const {
            return _isolate;
        }
         Local<Context> GetContext() const {
            return PersistentToLocal::Strong(_context);
        }
    private:
        struct io_uring_info *io_uring_data;
        Global<Context> _context;
        Isolate * _isolate;
};
```

env目前的功能还不多，只要负责管理context、isolate、io_uring等数据结构。另外还有一些和具体操作相关的上下文。

```
struct RequestContext {
   RequestContext(Environment * passEnv, Local<Object> _object)
   : env(passEnv),  object(passEnv->GetIsolate(), _object) {}
   ~RequestContext() {
       if (!object.IsEmpty()) {
           object.Reset();
       }
   }
   Environment * env;
   Global<Object> object;
};

struct SignalRequestContext: public RequestContext
{
   SignalRequestContext(Environment * passEnv, Local<Object> _object, int _sig)
   : RequestContext(passEnv, _object), sig(_sig) {}
   int sig;
};
```

前面介绍过io_uring层的上下文request，request主要是用于io_uring任务完成时，知道执行哪个回调函数，并且记录了少量的上下文，但是reuqest的字段不一定够用，所以RequestContext主要记录额外的上下文，其实把RequestContext的字段合进request也是可以的。

6 事件循环的设计

No的事件循环是io_uring实现的，事件循环的本质就是在一个循环里不断等待任务和执行任务，那么什么时候结束呢?

```
while(io_uring_data->stop != 1 && io_uring_data->pending != 0) {
        // 等待和处理任务
}
```

目前可以通过设置stop直接停止事件循环，正常情况下，没有任务了就会结束事件循环，通过pending字段记录，比如发起一个读取文件的请求，pending就是1，读完后就会减一，这时候，事件循环就会结束，相对Node.js的handle和request，No里是没有的，No里通过控制pending的值去控制事件循环的状态。

7 如何使用

No是基于Linux的io_uring的，目前在Linux5.5及以上的系统可以运行，可以安装ubuntu21.04及以上的虚拟机使用，具体可以参考仓库说明（github.com/theanarkh/No...）。目前支持了TCP、UCP、Unix域、文件、信号、定时器、log，进程还没有写完，总体只是支持一些简单的操作，后续慢慢更新。

8 后记

写No是一个让人非常深刻的过程，已经很多年没有正经写过c、c++代码，或许代码里有不对的用法，但是整个过程里的思考、编码和调试让我学到了很多东西，也给我了一段深刻的时光。目前实现的功能还不多，也不足以用起来，还有很多事情需要做，纸上得来终觉浅，绝知此事要躬行。后续慢慢学习、慢慢思考、慢慢更新！