怎样阅读源代码

January 25, 2016

原题目：How To Read Source Code，原作者：Aria Stewart

中文翻译：

这篇文章基于我在Oneshot Nodeconf Christchurch的一个演讲。

我本来没有想要写这篇文章。程序员不读源代码听起来似乎是很荒谬的。然后我真遇到了一群不读源代码的程序员。接着我又跟更多的人进行了交谈，发现他们除了读代码示例或测试脚本之外什么也不看。最重要的是，我遇到过很多新手程序员，对他们来说，找到从哪个地方开始阅读是非常困难的。

当我们说读源代码的时候，我们要表达什么意思？

我们是为了什么去读源代码？为了理解它。为了找bug，为了知道这些代码和系统中的其他软件是怎样交互的。我们还会为了回顾、品评而去读。为了找出其中的接口信息，为了理解和找到不同模块之间的界线，为了学习，我们都会去读源代码。

读代码不是一个线性过程

读代码的过程不是线性的。人们往往认为读源代码就像读一本书一样：先搞定简介或者README，然后从第一章开始一章一章的读，直到结束。其实并不是这样的。我们甚至都不能确定一个程序有没有结束，很多程序是不会终止的。我们应该在章节、模块之间跳转，反复阅读。我们也可以选择通读单个模块，但是这样我们就无法理解这个模块所引用的其他模块的代码。我们也可以根据程序的执行顺序去阅读，但是我们最后并不会清楚程序会向哪里执行。

读的顺序

要从一个库的入口开始读吗？对于Node库来说，即从index.js或者主脚本开始？

在浏览器中的情况呢？即使是找到程序入口，弄清楚载入了哪些文件，是怎样载入的都是一个很关键的事情。去研究文件之间是怎样关联起来的是一个很好的着手点。

除此之外，还可以从最大的一个源文件开始读。或者尝试在debugger中设置一个“浅”断点，通过函数调用去追溯源代码。亦或在某些复杂晦涩的地方设置一个深断点，然后去读函数调用栈里面的每一个函数。

代码的分类

我通常习惯于以语言去区分源代码，如Javascript, C++, ES6, Befunge, Forth或者LISP。熟悉的语言读起来相对更容易，对于不太熟悉的语言，我们往往就不会去看了。

这里还有另外一种看待源代码的方法，就是基于每一的模块的功能去分类：

- 连接类代码（Glue）
- 接口定义类代码
- 实现类
- 算法类
- 配置类
- 任务类

关于算法类的代码已经有很多研究了，因为学术界就是干这个的。算法就是用数学模型去处理一件事情的方法。其他种类的代码就要模糊很多了，但是我认为它们更加有趣。它们才是绝大多数人在编程的时候写的代码。

当然，很多时候一个模块可以同时做以上的很多事情。而读代码首先要做的事情之一恰恰就是弄清楚每一个部分是在做什么。

什么是连接类代码

并不是所有的接口都能很好的工作在一起。连接类代码是将各模块连在一起的管道。中间件、Promise和绑定回调函数的代码、将参数导入object中或者解析object的代码，这些都属于连接类代码。

怎样阅读连接类代码

关键要弄清楚两个接口是怎样以不同方式构建起来的，以及它们共通的地方。

下面的例子来自于Ben Drucker的`stream-to-promise`

```
internals.writable = function (stream) {
  return new Promise(function (resolve, reject) {
    stream.once('finish', resolve);
    stream.once('error', reject);
  });
};
```

可以看到，上面两个接口分别是Node的stream和Promise接口。

它们的共同点在于两者完成的时候都会执行一个操作，在Node中通过事件（event）来实现，而在Promise中通过调用resolve函数实现。

可以看到，Promise只能执行一次，但是stream可以发出同样的事件很多次。

更多连接类代码的例子：

```
var record = {
    name: (input.name || '').trim(),
    age: isNaN(Number(input.age)) ? null : Number(input.age),
```

```
    email: validateEmail(input.email.trim())
}
```

在读连接类代码的时候，还可以思考的是报错情况下的处理。

一段程序会抛出错误吗？能删除坏值吗？或者可以将其强制转换成可以接受的值？对于它们执行的环境来说，这些是正确的选择吗？类型转换是否是有损的？这些问题都是很值得思考的。

什么是接口定义类代码

可能你写过一些只在一两个地方用得到的模块，它们基本是在内部使用的，你不希望有人费很大劲去读这些模块。

而接口定义类代码却恰恰和上述情况相反。它们是模块之间泾渭分明的边界线。

下面的例子来自于Node的events.js：

```
exports.usingDomains = false;

function EventEmitter() { }
exports.EventEmitter = EventEmitter;

EventEmitter.prototype.setMaxListeners = function setMaxListeners(n) { };
EventEmitter.prototype.emit = function emit(type) { };
EventEmitter.prototype.addListener = function addListener(type, listener) {
EventEmitter.prototype.on = EventEmitter.prototype.addListener;
EventEmitter.prototype.once = function once(type, listener) { };
EventEmitter.prototype.removeListener = function removeListener(type, listen
EventEmitter.prototype.removeAllListeners = function removeAllListeners(type
EventEmitter.prototype.listeners = function listeners(type) { };
EventEmitter.listenerCount = function(emitter, type) { };
```

上面的代码在定义EventEmitter的接口。

关于这类代码可以思考的问题包括：它们是否完全？能提供哪些保证？内部的细节信息会暴露给用户吗？

在一个有严格接口定义的架构中，上面就是这类代码应该出现的地方。

就像连接类代码一样，我们可以思考它是怎样处理错误和报错的。处理方法前后一致吗？能区分出因为内部不一致而出现的错误和因为用户使用不当而出现的错误吗？

实现类代码

```
startRouting: function() {
  this.router = this.router || this.constructor.map(K);

  var router = this.router;
  var location = get(this, 'location');
  var container = this.container;
  var self = this;
  var initialURL = get(this, 'initialURL');
  var initialTransition;

  // Allow the Location class to cancel the router setup while it refreshes
  // the page
  if (get(location, 'cancelRouterSetup')) {
    return;
  }

  this._setupRouter(router, location);
```

```
    container.register('view:default', _MetamorphView);
    container.register('view:toplevel', EmberView.extend());

    location.onUpdateURL(function(url) {
      self.handleURL(url);
    });

    if (typeof initialURL === "undefined") {
      initialURL = location.getURL();
    }
    initialTransition = this.handleURL(initialURL);
    if (initialTransition && initialTransition.error) {
      throw initialTransition.error;
    }
  },
```

上面的示例取自Ember.Router。

这里往往是需要在“为什么”上作更多说明的地方。

读这类代码的时候应着重思考它是怎样跟更大的整体相融合的。

从公共接口中传入的值是怎样的？哪些需要验证(validation)？包含我们认为该有的值吗？会影响到其他哪些部分？当需要改写以加入新功能的时候会有哪些潜在危险？可能会导致程序崩溃的地方有哪些？有测试代码来对其进行测试吗？变量的生命周期是什么？

算法

算法类代码是实现类代码的一种，通常是封装起来不对外暴露的。它可以说是程序的血肉。也是一款软件的业务逻辑和主进程所在。

```
function Grammar(rules) {
  // Processing The Grammar
  //
  // Here we begin defining a grammar given the raw rules, terminal
  // symbols, and symbolic references to rules
  //
  // The input is a list of rules.
  //
  // The input grammar is amended with a final rule, the 'accept' rule,
  // which if it spans the parse chart, means the entire grammar was
  // accepted. This is needed in the case of a nulling start symbol.
  rules.push(Rule('_accept', [Ref('start')]));
  rules.acceptRule = rules.length - 1;
```

上面的代码出自一个叫做lotsawa的解析引擎。

人们常说好的注释应该告诉读者为什么这件事情要这样来做，而不是一段代码在做什么。算法类代码通常需要更多的解释，因为如果是一个很简单的算法的话，那通常它就已经被吸纳进标准库中了。

下面这段代码是计算机系学生喜欢的（或者有糟糕记忆的）：

```
// Build a list of all the symbols used in the grammar so they can be number
// by name, and therefore their presence can be represented by a single bit
  function censusSymbols() {
    var out = [];
    rules.forEach(function(r) {
      if (!~out.indexOf(r.name)) out.push(r.name);

      r.symbols.forEach(function(s, i) {
        var symNo = out.indexOf(s.name);
        if (!~out.indexOf(s.name)) {
```

```
            symNo = out.length;
            out.push(s.name);
          }

          r.symbols[i] = symNo;
        });

        r.sym = out.indexOf(r.name);
      });

      return out;
    }

    rules.symbols = censusSymbols();
```

读起来像是数学论文，对吗？

在读算法类代码的时候需要关注的事情之一就是其运用的数据结构。上面的程序建了一个符号列表，并确保其中没有重复元素。

读的时候同时也要注意有关程序时间复杂度的线索。上面的代码中，有两个循环。用大O来记的话，时间复杂度就是O(n * m)。但已经有人注意到，循环之中还有`indexOf`的调用。这在Javascript中也是循环操作。因此这又在时间复杂度上加了一个因子。还好这段代码并不是该算法的主要部分。

配置

源代码和配置文件之间的界线非常的窄。对配置文件来说，表达力强、可读性强和直接之间永远是冲突的。

```
app.configure('production', 'staging', function() {
  app.enable('emails');
```

```
});

app.configure('test', function() {
  app.disable('emails');
});
```

上面是一个用Javascript进行配置的例子。

在这里可能会遇到的问题是不同选项的组合爆炸性增长。应该配置多少个环境？在每一个环境实例中又配置哪些东西？我们很容易就会过度配置，去考虑所有的情况，但是bug可能只出现于其中一种情况中。时刻注意配置文件给我们多少自由度是非常有用的。

```
"express": {
    "env": "", // NOTE: `env` is managed by the framework. This value will b
    "x-powered-by": false,
    "views": "path:./views",
    "mountpath": "/"
},

"middleware": {

    "compress": {
        "enabled": false,
        "priority": 10,
        "module": "compression"
    },

    "favicon": {
        "enabled": false,
        "priority": 30,
        "module": {
            "name": "serve-favicon",
```

```
            "arguments": [ "resolve:kraken-js/public/favicon.ico" ]
        }
    },
```

上面是一小段kraken的配置文件。

Kraken采取了“低功耗(low power)语言”的路，其配置文件采用JSON。更多一点“配置”，而相对少一点“源代码”。其目的之一就是让可选择项的数目可控。很多工具都采用简单的key-value对或者ini类的文件来做配置是有道理的，即使这样会使配置文件的表达力受限。

配置类的代码有如下一些值得思考的有趣而独特的限制：

- 生命周期
- 机器可写性
- 对一段代码负有责任的人会多很多
- 怎样适用在一些奇怪的地方，比如环境变量
- 往往有涉及安全的敏感信息

任务类

对50张信用卡计费，用编译器和其他构建工具开发一个复杂的软件，发出100封电子邮件。这些事情有什么共同点？

事务性。通常一个系统的某一部分只需严格执行一次，而遇到错误的话则完全不执行。编译器遗留下的错误构建文件是bug的一大来源。对客户重复收费是很糟糕的。因为重试而向用户邮箱滥发邮件也很可怕。

重启性。能根据系统状态，在之前退出的地方继续运行。

序列性。对于不是严格线性的程序，通常都有一个方向明晰的执行流程。循环是其

中一大块。

阅读杂乱代码

一般人会怎样入手下面的一段代码：

```
      DuplexCombination.prototype.on = function(ev, fn) {
    switch (ev) {
  case 'data':
  case 'end':
  case 'readable':
this.reader.on(ev, fn);
return this
  case 'drain':
  case 'finish':
this.writer.on(ev, fn);
return this
  default:
return Duplex.prototype.on.call(this, ev, fn);
}
};
```

对，这就是反向缩进，要怪就怪Issac吧。

美化一下！

standard -F dc.js

```
DuplexCombination.prototype.on = function (ev, fn) {
  switch (ev) {
    case 'data':
    case 'end':
```

```
    case 'readable':
      this.reader.on(ev, fn)
      return this
    case 'drain':
    case 'finish':
      this.writer.on(ev, fn)
      return this
    default:
      return Duplex.prototype.on.call(this, ev, fn)
  }
}
```

读代码的时候使用工具是很好的。

又比如下面这段代码：

```
(function(t,e){if(typeof define==="function"&&define.amd){define(["underscor
jquery","exports"],function(i,r,s){t.Backbone=e(t,s,i,r)})}else if(typeof ex
s!=="undefined"){var i=require("underscore");e(t,exports,i)}else{t.Backbone=
{},t._,t.jQuery||t.Zepto||t.ender||t.$)}})(this,function(t,e,i,r){var s=t.Ba
ne;var n=[];var a=n.push;var o=n.slice;var h=n.splice;e.VERSION="1.1.2";e.$=
noConflict=function(){t.Backbone=s;return this};e.emulateHTTP=false;e.emula
N=false;var u=e.Events={on:function(t,e,i){if(!c(this,"on",t,[e,i])||!e)retu
his;this._events||(this._events={});var r=this._events[t]||(this._events[t]=
r.push({callback:e,context:i,ctx:i||this});return this},once:function(t,e,r)
!c(this,"once",t,[e,r])||!e)return this;var s=this;var n=i.once(function(){
(t,n);e.apply(this,arguments)});n._callback=e;return this.on(t,n,r)},off:fun
n(t,e,r){var s,n,a,o,h,u,l,f;if(!this._events||!c(this,"off",t,[e,r]))return
s;if(!t&&!e&&!r){this._events=void 0;return this}o=t?[t]:i.keys(this._events
r(h=0,u=o.length;h<u;h++){t=o[h];if(a=this._events[t]){this._events[t]=s=[];
||r){for(l=0,f=a.length;l<f;l++){n=a[l];if(e&&e!==n.callback&&e!==n.callback
llback||r&&r!==n.context){s.push(n)}}}if(!s.length)delete this._events[t]}}r
n this},trigger:function(t){if(!this._events)return this;var e=o.call(argum
```

```
1);if(!c(this,"trigger",t,e))return this;var i=this._events[t];var r=this._
s.all;if(i)f(i,e);if(r)f(r,arguments);return this},stopListening:function(t,
{var s=this._listeningTo;if(!s)return this;var n=!e&&!r;if(!r&&typeof e==="
```

uglifyjs -b < backbone-min.js

```
(function(t, e) {
    if (typeof define === "function" && define.amd) {
        define([ "underscore", "jquery", "exports" ], function(i, r, s) {
            t.Backbone = e(t, s, i, r);
        });
    } else if (typeof exports !== "undefined") {
        var i = require("underscore");
        e(t, exports, i);
    } else {
        t.Backbone = e(t, {}, t._, t.jQuery || t.Zepto || t.ender || t.$);
    }
})(this, function(t, e, i, r) {
    var s = t.Backbone;
    var n = [];
    var a = n.push;
    var o = n.slice;
    var h = n.splice;
    e.VERSION = "1.1.2";
    e.$ = r;
    e.noConflict = function() {
```

人的部分

猜测代码作者的意图有很多的方法。

找**guards**和**coercions**

```
if (typeof arg != 'number') throw new TypeError("arg must be a number");
```

看上去函数的定义域是数值型。

```
arg = Number(arg)
```

强制转换为数值类型。和前面的一样，只是不再抛出错误。可能会有NaN出现。

找默认值

```
arg = arg || {}
```

默认为空。

```
arg = (arg == null ? true : arg)
```

如果没有相关参数传进来的话，则默认为true。

找层（**layers**）

Express中的req和res是跟web相连的。它们会作用到哪一层呢？能够找到接口的边界吗？

找轨迹（**tracing**）

有检查点吗？有debug日志吗？它们是一个完整的功能，还是之前的bug残留下来的

呢？

找自反性（**reflexivity**）

识别符是动态生成的吗？有eval、元编程和新函数定义吗？func.toString()是一个很好的切入点。

找生命周期

- 被谁初始化的
- 什么时候会改变
- 被谁改变
- 上述信息会在其他地方出现吗
- 会出现不一致性吗

某个时间，某个人输入了一个值。在另外的某个地方，某个时候这个值会对其他人产生影响，他们是谁？是什么或者谁来决定的？这个值会改变吗？由谁来改变？

找隐藏状态机（**hidden state machines**）

有时候布尔变量会被当作解构了的状态机使用。

例如，变量isReadied和isFinished可能隐含如下状态：

```
var isReadied = false;
var isFinished = false;
```

或： START -> READY -> FINISHED

```
isReadied | isFinished | state
```

```
----------|------------|------------
false     | false      | START
false     | true       | invalid
true      | false      | READY
true      | true       | FINISHED
```

找构造（**composition)**和继承

这段代码有我认识的部分吗？它们有名字吗？

找相同的操作

`map`, `reduce`, cross-join。

是时候开始读源代码了，Enjoy!

Github:

https://github.com/freedombird9/how-to-read-source-code