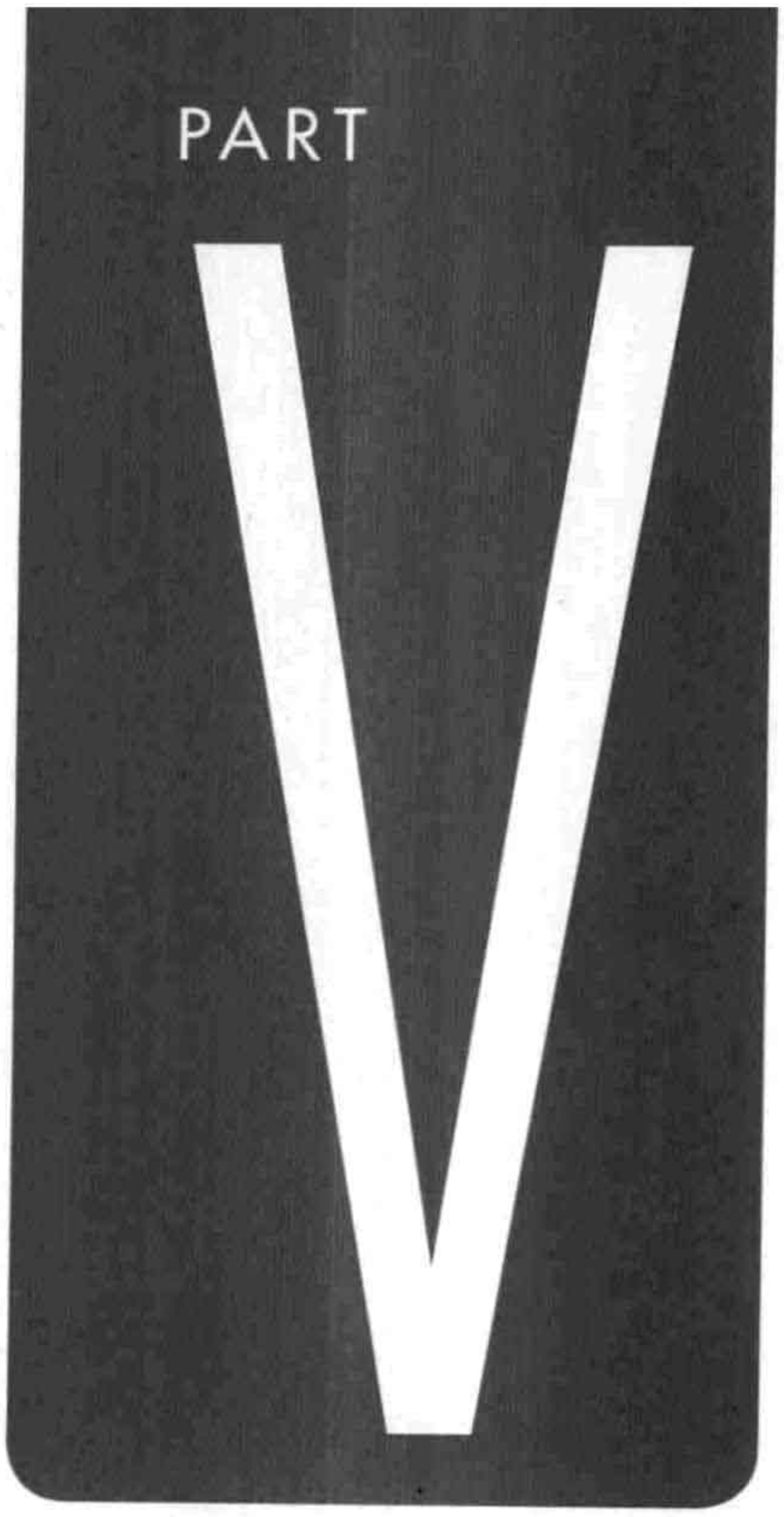

测试

CHAPTER

15 代码共享

在本书介绍部分，我就提过，Node.js最重要的特点之一就是：它所使用的语言——JavaScript，是浏览器唯一支持的语言。

尽管现在我们已经写了很多独立的JavaScript代码，减少了在开发 Web应用时总要在不同语言之间切换上下文的痛苦。我们还没有享受到一次编码处处运行的好处。

本章分析适合代码共享的最佳用例，以及如何解决常见的语言兼容性问题。本章还会介绍如何书写模块化Node代码的最佳实践，并将其通过`browserbuild`编译后在浏览器中直接运行。

什么样的代码可以共享

回答代码是否可以在浏览器和服务器端共享的最简单的方式就是把问题拆为如下两个问题：

- 是否值得在两个环境中运行同一份代码？
- 代码依赖的API在两个环境中是否都有？如果不是，那么是否能够很容易地找到替换它们的方法（称为模拟实现法）。

回答第一个问题比较简单，答案本身取决于程序和项目本身。回答第二个问题就有点困难了。

在第2章中，我们介绍了有些特定的和JavaScript有关联的API并非是语言的一部分，而是浏览器提供的标准API。例如：XMLHttpRequest、WebSocket、DOM、2d画布 API，等等。

尽管Node.js核心并未提供XMLHttpRequest API，但是我们可以通过使用Node.js的HTTP客户端API来替代。在NPM有个名为xmlhttprequest的项目就是起这个作用的。

在其他的场景中，某些模块只和原生API交互，这类代码就很容易写成可以在两个环境中运行。

这类例子如下所示。

- 日期操作工具集：它们通常简单地处理原生Date API或者对其做相应扩展。
- 模板引擎：它们通常接收一个字符串，通过正则表达式或者循环来解析，并生成一个（编译）函数，用于输出编译后的字符串。
- 数学及加密库：它们通常处理Number和与数学相关的内容。
- 面向对象框架：它们提供一些在JavaScript中书写类的语法糖。换句话说，提供一些API，使得在JavaScript中能和在其他典型的面向对象语言中一样书写面向对象代码。

书写兼容的JavaScript代码

第一个挑战就是要解决在Node.js模块系统中书写的JavaScript代码无法在浏览器中运行的问题。

导出模块

第一个问题是浏览器环境中没有module全局变量。哪怕文件本身没有依赖，但是，要想让它能为其他Node程序所用，需要通过module.exports或者exports。

281 假设，我们已熟悉一个简单的两数求和的函数：

add.js

```
module.exports = function (a, b) {
  return a + b;
}
```

在Node中，只需简单地从另外一个文件中调用require('./add')就可以了。在浏览器中，我们更希望将其暴露为一个全局变量add。

为了避免修改已有代码，我们可以在浏览器中，通过伪造一个module对象来模拟module.exports：

```
if ('undefined' == typeof module) {
  module = { exports: {} };
}
```

模块代码的最后，对象生成后，就可以将其暴露给window全局对象了。

```
if ('undefined' != typeof window) {
  window.add = module.exports;
}
```

和Node.js不同，在浏览器中，文件都是在全局作用于下执行的。因此，我们需要为程序引入一个全局的module变量。

将模块代码包裹在一个自执行的函数中是个不错的主意。

```
(function (module) {
  module.exports = function (a, b) {
    return a + b;
  }

  if ('undefined' != typeof window) {
    window.add = module.exports;
  }
})('undefined' == typeof module ? { module: { exports: {} } } : module);
```

太好了！模块还是可以通过require获得：

```
$ node
> require('./add')(1,2)
3
```

同时，还能在DOM中引入：

282

add.html

```
<script src="add.js"><script>
<script>
  console.log(add(1,2));
</script>
```

模拟实现ECMA API

下一个挑战就是，一些主流浏览器中的特性在其他浏览器及JavaScript引擎中都没有。

有时，Node中的v8引擎甚至比诸多电脑所用的Google Chrome浏览器的引擎还要新。然而，对于Function#bind这样的API，一些主流引擎，如Safari引擎JavaScriptCore VM就还不支持。所以，要想让代码处处运行，对于使用了哪些功能要特别小心。

对于缺失的这类功能，有两个解决办法：通过扩展原型来提供这类功能的实现或者使用工具函数。

扩展原型

假设我们书写了一个同时在Node和浏览器中运行的模块，它用到了Function#bind方法。

```
var myfn = fn.bind(this);
```

由于bind方法并不一定在所有环境中都存在，所以当它不存在时，我们要创建一个出来：

```
if (!Function.prototype.bind) {
  Function.prototype.bind = function () {
    // code that replicates bind behavior
  }
}
```

有一个名为es5-shim的项目做了类似上面的事情，它把所有浏览器中缺少的ECMA标准API都实现了。要了解该项目详情，可以参考https://github.com/kriskowal/es5-shim。

这种技术有个很明显的好处就是在加入填补的方法后，几乎不需要修改源代码。

283 缺点就是这么做会破坏原型对象，影响其他使用者，可能还会更糟。

工具函数

另一种解决方法就是定义简单的函数，接收原生对象作为参数，如果该对象上的函数已经实现了，就直接使用，否则就实现一次。

例如，Object.keys就是个很好的例子，在Node中会经常使用此函数，但是在很多浏览器中却还没有提供其实现。

接着，我们就通过如下方式来定义一个工具函数：

```
var keys = Object.keys || function (obj) {
  var ret = [];
  for (var i in obj) {
    if (Object.prototype.hasOwnProperty.call(obj, i)) {
      ret.push(i);
    }
  }
  return ret;
};
```

这种技术的好处就是随处都能用，没有隐患，并且对于开发者来说，能够一眼就看明白哪些方法是模拟的，哪些是原生的，这对于在某些浏览器中检测代码性能很有帮助。

缺点就是，我们要记住是用工具函数而不是用原生的。

除此之外，有的时候，最终书写出来的代码会有些冗长。比如，旧版本的IE不支持Array#forEach、Array#map及Array#filter，要想写出兼容的代码，可能要写成如下形式：

```
arr.filter(function () {}).map(function(){ }).forEach(function () {})
```

不过，上述代码远没有使用工具函数来得清晰：

```
each(map(filter(arr, function() {}), function () {}), function (){})
```

模拟实现Node API

有些如EventEmitter这样的Node API一般会在自定义的类中使用。幸运的是，Node社区书写了可以在所有环境下运行的Node API。

EventEmitter

284

对node EventEmitter的实现可以参考https://github.com/Wolfy87/EventEmitter和https://github.com/tmpvar/node-eventemitter。

assert

支持浏览器的assert模块可以在这里找到： https://github.com/Jxck/assert。

模拟实现浏览器端API

很多情况下，我们希望在Node中也能运行浏览器支持的方法。

XMLHttpRequest

node-XMLHttpRequest项目（https://github.com/driverdan/node-XMLHttpRequest）为Node提供了XMLHttpRequest的模拟实现，通过如下方式就可以引入该模块了：

```
var XMLHttpRequest = require('xmlhttprequest')
```

DOM

一个完整并且测试完全的DOM I、DOM II以及 DOM III的实现在这里：https://github.com/tmpvar/jsdom。

WebSocket

Node中也有WebSocket客户端的API实现（https://github.com/einaros/ws）：

```
var WebSocket = require('ws')
```

node-canvas

用于图片操作的2D画布上下文也有在Node中的实现，参考node-canvas：https://github.com/learnboost/node-canvas。

跨浏览器的继承实现

在模块中，经常需要将一个类继承自另一个类。第2章中介绍的.__proto__可以用于实现继承，但却很难模拟实现。

一个简单的解决方法是通过如下方式定义一个工具函数：

```
/**
 * 用于实现继承的工具函数
 *
 * @param {Function} 构造器

285      * @param {Function} 父类构造器
 * @api private
 */

function inherits (a, b) {
  function c () {};
  c.prototype = b.prototype;
  a.prototype = new c;
};
```

然后就可以在Node和浏览器中使用了。

```
function A () {}
function B () {}

inherits(A, B);
// instead of A.prototype.__proto__ = B.prototype
```

集成到一起：browserbuild

通过自执行函数将模块代码包裹起来，同时到处都得执行typeof检测多少有点麻烦。

我创建了一个名为browserbuild的项目，它的作用就是将以Node风格书写的代码（即使用了require、module.exports以及exports，并且代码都分属不同文件），通过运行简单的命令，就编译为浏览器端可执行的版本。

browserbuild可以允许你以Node风格书写此前的求和模块，通过编译后，就会生成一个可以直接在浏览器端通过<script>嵌入执行的版本。

browserbuild同时还是个NPM模块，它提供了browserbuild命令行脚本。要全局安装该模块，可以采用如下方式：

```
npm install -g browserbuild
browserbuild --version
```

上述第二行命令用于展示所安装browserbuild的版本号。这里所用的版本号是0.4.8。

还可以直接安装到工作目录中，通过.bin目录来访问：

```
npm install browserbuild
./node_modules/browserbuild/.bin/browserbuild --version
```

基础案例

286

本例中，我们要书写一个模块，该模块依赖于日志工具。

首先，书写一个main.js文件：

main.js

```
var log = require('./log'):
module.exports = function () {
log('Executed my module');
}
```

在使用特定模块时，main文件不管是在NPM模块系统中，还是在单独的文件中，都可以通过require来引入。

log.js

```
module.exports = function (str) {
  return console.log(str);
}
```

在Node中，可以直接写成；

node.js

```
var mymodule = require('./main')
mymodule();
```

对于浏览器端，我们需要其导出成全局的mymodule变量。所以，我们需要在工作目录下运行browserbuild命令，同时提供全局变量名以及main文件。

```
browserbuild --main main --global mymodule main.js log.js > out.js.
```

上述代码的含义是告诉browserbuild：编译main.js和log.js，并将main模块导出为mymodule全局变量。

除此之外，注意，命令行最后我们使用了>compiled.js。这是将输出的结果保存到compiled.js文件中。

要将其作为库引入到浏览器中，只需添加一个<script>标签，并指向compiled.js脚本即可：

```
<script src='compiled.js">
```

287 生成的脚本如下所示：

compiled.js

```
(function(){var global = this;function debug(){return debug};function require(p,
 parent){ var path = require.resolve(p) , mod = require.modules[path]; if (!mod)
 throw new Error('failed to require "' + p + '" from ' + parent); if (!mod.exports)
 { mod.exports = {}; mod.call(mod.exports, mod, mod.exports, require.relative(path),
 global); } return mod.exports;}require.modules = {};require.resolve =
 function(path){ var orig = path , reg = path + '.js' , index = path + '/index.js';
 return require.modules[reg] && reg || require.modules[index] && index ||
 orig;};require.register = function(path, fn){ require.modules[path] = fn;};require.
 relative = function(parent) { return function(p){ if ('debug' == p) return debug;
 if ('.' != p.charAt(0)) return require(p); var path = parent.split('/') , segs =
 p.split('/'); path.pop(); for (var i = 0; i < segs.length; i++) { var seg =
 segs[i]; if ('..' == seg) path.pop(); else if ('.' != seg) path.push(seg); } return
 require(path.join('/'), parent); };};require.register("main.js", function(module,
 exports, require, global){

var log = require('./log');

module.exports = function () {
  log('Executed my module');
}

});require.register("log.js", function(module, exports, require, global){
module.exports = function (str) {
  return console.log(str);
}

});mymodule = require('main');
})();
```

上述脚本中，第一部分实现了一个require函数供后续代码使用。编译后的脚本全部压缩在一行代码中，这样文件长度对于编译过程的影响可以减到最小。

另一部分比较有意思的代码是，require.register函数提供了一个global参数。这样做是为了避免你去检测window对象是否存在，以此来提供另一个global变量。书写代码时可以直接看作是Node.js的global对象，要是在浏览器中，该对象会变成window对象。

最后，上述代码的最后一部分是导出了全局变量（mymodule），这也解释了运行browserbuild的时候要指定是main模块的原因：

```
mymodule = require('main');
```

browserbuild另一个有意思的特性是if node代码块，它允许你使用JavaScript注释来告诉编译器，该代码块中的代码不需要编译到浏览器端版本中： 288

nodeonly.js

```
// if node
process.exit(1);
// end

console.log('browser and node');
```

如果运行如下命令编译上述代码，if node代码块中的代码就不会被编译进去：

```
$ browserbuild --main nodeonly nodeonly.js
```

你会注意到process.exit代码行不在里面。

```
// …
require.register("nodeonly.js", function(module, exports, require, global){
console.log('browser and node');
});nodeonly = require('nodeonly');
})();
```

要获取更多browserbuild命令的选项，可以执行browserbuild --help，或者参看项目主页： http://github.com/learnboost/browserbuild。

小结

正如全书一直在强调的，Node.js做了一项很伟大的工作就是将 JavaScript带到了服务器端。

这一创新依赖于其模块系统，而浏览器端则没有对应的默认模块系统。

本章从介绍如何利用运行时的方法检测来书写同时能在服务器端和浏览器端运行的代码。通过执行typeof检查，可以实现模块系统的特性检查，并为浏览器端也提供一个导出机制，比如通过全局对象。

然而，在所有的文件中都要进行手动地通过自执行的函数将代码包裹起来，并执行typeof进行检查，这显然有悖于Node中require系统的简洁性，browserbuild就是为了解决这类问题而生的。有了它，就可以很轻松地书写Node.js模块，并编译到浏览器端进行运行了。

这种方法的显著的好处就是，模块是完全以全局变量的方式在浏览器端导出的，就和jQuery和IO一样，也就是说，并没有引入一个特定的模块系统API来让终端用户在浏览器环境中使用。