



深入浅出 Node JS 知识点总结

Node简介

Node的特点

1. 异步I/O

在Node中，绝大多数的操作都以异步的方式调用。

从文件读取到网络请求。这样的意义在于，我们可以从语言层面上很自然的进行并行I/O操作。

每个调用无需等待之前的I/O调用结束。在编程模型上可以极大的提升效率。

2. 事件与回调函数

无论在前端还是后端，事件都是常用的。

事件的编程有轻量级、轻耦合、只关心事物特点的优势，但是在多个任务的场景下，事件与事件之间各自独立，如何协作是一个问题。

3. 单线程

Node保持了JavaScript在浏览器中单线程的特点。而且在Node中，JavaScript与其他线程是无法共享任何状态的。

单线程的最大好处是不用像多线程编程那样处处在意状态的同步问题，这里没有死锁的存在，也没有线程上下文所带来的性能上的开销。

线程的弱点：无法利用多核CPU；错误会引起整个应用退出；大量的计算占用CPU导致无法异步调用CPU。

4. 跨平台

Node的应用场景

1. I/O 密集型

2. CPU密集型业务：适当调整和分解大型运算任务为多个小任务，使得运算能够释放，不阻塞I/O之间的阻塞发起，这样既能享受并行异步I/O的好处，又可以充分利用CPU。

3. 与遗留系统可以和平共处

4. 分布式应用

模块机制

CommonJS 规范

CommonJS对模块的定义主要分为：模块引用，模块定义和模块标识三个部分。

1.模块引用

示例代码：

```
let math = require('math');
```

在CommonJS规范中，存在**require()**方法，这个方法接受模块标识，以此引入一个模块的API到当前的上下文中。

2.模块定义

对应引入的功能，上下文提供**exports对象**用于导出当前模块的方法或变量，并且它是唯一的导出的出口。在模块中，还存在一个**module对象**，它代表模块自身，而**exports**是模块的属性。

在Node中，一个文件就是一个模块，将方法挂接在**exports对象**上作为属性即可定义导出的方式。

```
// math.js
exports.add = function () {
    let sum = 0,
        i = 0,
        args = arguments,
        l = args.length;
    while (i < l) {
        sum += args[i++];
    }
    return sum
}
```

在另一个文件中，我们通过**require()**方法引入模块后，就能调用定义的属性或方法了。

```
//program.js
var math = require('math');
exports.increment = function (val){
    return math.add(val, 1);
};
```

3.模块标识

模块标识是传递给require()方法的参数，它必须符合小驼峰命名的字符串，或者以..开头的相对路径，或者绝对路径。它可以没有文件名后缀js。

Node的模块实现

Node在实现exports, require 和 module 的过程中，经历了三个步骤：1) 路径分析，2) 文件定位，3) 编译执行。

在Node中，模块分为两类：一类是Node提供的模块，为**核心模块**；另一类是用户编写的模块，称为**文件模块**。

核心模块是在Node源代码的编译过程中，编译进了二进制执行文件。在Node启动时，部分核心模块就直接被加载进内存中。

文件模块则是在运行时动态加载，需要完整的路径分析，文件定位，编译执行过程，速度比核心模块慢。

模块调用栈

核心模块包括：底层的c/c++内建模块和JavaScript模块。其中C/C++内建模块主要提供API给JavaScript核心模块和第三方JavaScript文件块。

文件模块包括：C/C++扩展模块和JavaScript模块。

包与NPM

包实际上是一个存档文件，安装后解压后可解压还原为目录。完全符合CommonJS规范的包目录包含：

`package.json`: 包描述文件。

`bin`: 用于存放可执行的二进制文件的目录。

`lib`: 用于存放文档的目录。

`test`: 用于存放单元测试用例的代码。

包描述文件用于表达非代码的相关的信息，是一个JSON格式的文件。

CommonJS 为 `package.json` 文件定义了如下一些必需的字段：

`name`。包名。

`description`。包简介。

`version`。版本号。

`keywords`。关键词数组。

`maintainers`。包维护者列表。

`licenses`。当前包使用的许可证列表。

`dependencies`。使用当前包所需要依赖的包列表。这个属性十分重要，NPM会通过这个属性帮助自动加载依赖的包。

`scripts`: 脚本说明对象。它主要用被包管理器用来安装、编译、测试和卸载包。

```
"scripts":{  
  "install": "install.js",  
  "uninstall": "uninstall.js",  
  "build": "build.js",  
  "doc": "make-doc.js",  
  "test": "test.js"  
}
```

CommonJS包规范是理论，NPM是其中的一种实践。对于NODE而言，NPM帮助完成了第三方模块的发布、安装和依赖。

借助NPM，Node与第三方模块之间形成了很好的一个生态系统。

NPM常用功能

\$ npm -v (查看版本)

\$ npm (查看帮助)

\$ npm install express (安装包依赖)

\$ npm install express -g (全局安装包依赖，根据bin字段配置，将实际脚本链接到与Node可执行文件相同的路径下)

\$ npm install (从本地安装)

\$ npm install underscore --registry =http://registry.url (从非官方源安装)

编写模块：

```
exports.sayHello = function(){  
  return "Hello world";  
};
```

将这段代码保存为hello.js即可。

\$ npm init (初始化包描述文件，帮助生成package.json文件)

\$ npm adduser (注册包仓库账号，只有使用仓库账号才允许将包发布到仓库中。)

\$ npm publish (上传包：在这个过程中，npm会将目录打包成一个存档文件，然后上传到官方源仓库中)

\$ npm owner (管理包权限：可以添加或删除拥有者)

\$ npm ls (分析包：可以分析出当前路径下能够通过模块路径找到的所有包，并生成依赖树)

AMD规范

AMD规范是CommonJS模块规范的一个延伸，它更适用于前端的应用场景。它的模块定义如下：

```
define(id? dependencies?, factory);
```

它的模块id和依赖是可选的，与Node模块相似的地方在于factory的内容就是实际代码的内容。

```
define(function(){
    var exports = {};
    exports.sayHello = function(){
        alert('Hello from module: ' + module.id);
    };
    return exports;
});
```

不同之处是AMD模块需要**define**来明确定义一个模块，而Node实现中是隐形包装的，目的是进行作用域隔离，避免过去那种全局变量或者全局命名空间的方式，**以免变量污染**和不小心被修改。

另一个区别是内容需要通过返回的方式实现导出。

CMD规范

CMD规范与AMD规范的主要区别在定义模块和依赖引入的部分。在依赖部分，支持动态引入。

```
define(factory);

define(function(require, exports, module)){
    // module code
}
```

require, exports和module通过形参传递给模块，在需要依赖模块时，随时调用require()引入。

异步I/O

为什么要异步I/O

Node 面向网络的设计，Web 应用不再是单台服务器就能满足的，而