|  |
| --- |
|  |
|  |
| 需求规格说明书 |
| 基于Node.js的分析与应用 |
|  |
|  |
| 北京航空航天大学  2016-03-27 |

版本变更历史

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 版本 | 提交日期 | 主要编制人 | 审核人 | 版本说明 |
| V1.0 | 2017/3/23 | 温元祯 |  | 初稿 |
| V1.1 | 2017/3/27 | 王春柳 | 李岳檑 | 一稿 |
|  |  |  |  |  |
|  |  |  |  |  |
|  |  |  |  |  |
|  |  |  |  |  |
|  |  |  |  |  |
|  |  |  |  |  |
|  |  |  |  |  |

目录

[**1.** **引言** 3](#_Toc478397330)

[**1.1** **目的** 3](#_Toc478397331)

[**1.2** **标识** 3](#_Toc478397332)

[**1.3** **系统概述** 3](#_Toc478397333)

[**1.4** **文档概述** 4](#_Toc478397334)

[**1.5** **术语和缩略词** 4](#_Toc478397335)

[**2.** **引用文档** 5](#_Toc478397336)

[**3.** **功能需求** 5](#_Toc478397337)

[**3.1** **模块和包** 5](#_Toc478397338)

[**3.1.1** **创建和加载模块** 6](#_Toc478397339)

[**3.1.2** **创建和调用包** 7](#_Toc478397340)

[**3.1.3** **管理包** 8](#_Toc478397341)

[**3.2** **网络通信** 8](#_Toc478397342)

[**3.2.1** **构建TCP服务** 9](#_Toc478397343)

[**3.2.2** **构建UDP服务** 9](#_Toc478397344)

[**3.2.3** **构建HTTP服务** 9](#_Toc478397345)

[**3.2.4** **构建WebSocket服务** 10](#_Toc478397346)

[**3.3** **文件系统** 11](#_Toc478397347)

[**3.3.1** **文件操作** 11](#_Toc478397348)

[**3.3.2** **Buffer** 11](#_Toc478397349)

[**4.** **数据需求** 12](#_Toc478397350)

[**5.** **非功能需求** 12](#_Toc478397351)

[**5.1** **兼容性** 12](#_Toc478397352)

[**5.1.1** **浏览器兼容** 12](#_Toc478397353)

[**5.1.2** **操作系统兼容** 12](#_Toc478397354)

[**5.2** **高效性** 12](#_Toc478397355)

[**5.3** **容错性** 12](#_Toc478397356)

[**5.4** **可扩展性** 12](#_Toc478397357)

[**6.** **运行需求** 13](#_Toc478397358)

[**6.1** **硬件接口** 13](#_Toc478397359)

[**6.2** **软件接口** 13](#_Toc478397360)

[**7.** **应用场景** 13](#_Toc478397361)

[**7.1** **I/O 密集型** 13](#_Toc478397362)

[**7.2** **CPU密集型** 13](#_Toc478397363)

[**8.** **参考文献** 13](#_Toc478397364)

1. **引言**
   1. **目的**

本文档的编写目的是为了协调组内成员开展后期的工作，对项目提出需求，以指导后期的开发、测试等工作。

* 1. **标识**

Node.js版本号：Node.js6.10.1

需求报告版本：V1.1

* 1. **系统概述**

Node.js是一个可以让JavaScript运行在服务器端的平台，它可以让JavaScript脱离浏览器的束缚运行在一般的服务器环境下。

Node.js在诞生之初就充分考虑了在事实响应、超大规模数据要求下架构的可扩展性，实现了诸如文件系统、模块、包、操作系统API、网络通信等Core JavaScript没有或者不完善的功能。历史上将JavaScript移植到浏览器外的计划不止一个，但Node.js是最出色的一个。Node.js的JavaScript引擎是V8,来自Google Chrome项目。V8号称是目前世界上最快的JavaScript引擎，它的即时编译执行速度已经快到了接近本地代码的执行速度。Node.js不运行在浏览器中，所以也不存在JavaScript的浏览器兼容性问题。

Node.js内建了HTTP服务器支持，用户可以轻松地实现一个网站和服务器的组合。这个服务器不仅可以用来调试代码，而且它本身就可以部署到产品环境。Node.js还可以部署到非网络应用的环境下，比如一个命令行工具。Node.js还可以调用C/C++的代码，这样可以充分利用已有的诸多函数库，也可以将对性能要求非常高的部分用C/C++来实现。

Node.js用异步式I/O和事件驱动代替多线程，带来了可观的性能提升。Node.js除了使用V8作为JavaScript引擎之外，还使用了高效的libev和libeio库支持事件驱动和异步式I/O。图1是Node.js架构的示意图。



图 1 Node.js架构图

* 1. **文档概述**

本文档是基于Node.js的各模块需求分析和规格说明书，确定Node.js的功能需求、非功能需求、数据需求以及运行需求。

1. 以用例图的形式给出 Node.js功能需求的分解结构，并对用例模型中的参与者和用例进行详细的描述。
2. 描述了与此次系统实施相关的硬件环境的一些要求。
3. 描述了与此系统实施相关的软件环境的要求
   1. **术语和缩略词**

关于本文档中出现的一些专业术语、缩略语如表1所示。

|  |  |
| --- | --- |
| 术语/缩略语 | 描述 |
| JavaScript | JavaScript一种直译式脚本语言 |
| API | Application Programming Interface，应用程序编程接口 |
| HTTP | HyperText Transfer Protocol，[超文本传输协议](http://baike.baidu.com/item/%E8%B6%85%E6%96%87%E6%9C%AC%E4%BC%A0%E8%BE%93%E5%8D%8F%E8%AE%AE" \t "_blank) |
| libev | 一个[事件驱动](http://baike.baidu.com/subview/536048/536048.htm" \t "_blank)的编程框架 |
| libeio | 提供全套一部文件操作的接口 |
| libuv | libuv 是 Node 的新跨平台抽象层 |
| epoll | epoll是[Linux内核](http://baike.baidu.com/item/Linux%E5%86%85%E6%A0%B8" \t "_blank)为处理大批量[文件描述符](http://baike.baidu.com/item/%E6%96%87%E4%BB%B6%E6%8F%8F%E8%BF%B0%E7%AC%A6)而作了改进的poll，能显著提高程序在大量[并发连接](http://baike.baidu.com/item/%E5%B9%B6%E5%8F%91%E8%BF%9E%E6%8E%A5)中只有少量活跃的情况下的系统CPU利用率。 |
| kqueue | Kqueue功能同epoll，存在于许多unix系统中。 |
| POSIX | Portable Operating System Interface，一套操作系统API规范 |
| CommonJS | Javascript标准规范 |

表 1 专业术语/缩略图描述表

1. **引用文档**

【1】<https://cnodejs.org/topic/551200e6d792542a29789a43>

1. **功能需求**
   1. **模块和包**

[JavaScript](http://lib.csdn.net/base/javascript)原生态是一个全局的世界，所有如setTimeout，document等这样在浏览器中使用的API，都是全局定义的。然而js没有命名空间，不像其他语言通过命名空间可以有效的避免重名问题，当文件之间有复杂的依赖关系的时候就很容易出现一些变量的属性或方法被覆盖或改写，导致变量污染。CommonJS规范提出了模块规范，将函数和方法包含在不同的模块中，避免了相互污染。

Node.js摒弃了js在全局定义上的缺点，引入了模块和包机制，使得用户完全不必考虑变量污染、命名空间等问题。Node.js中模块和包机制的用例图如图3.1所示。

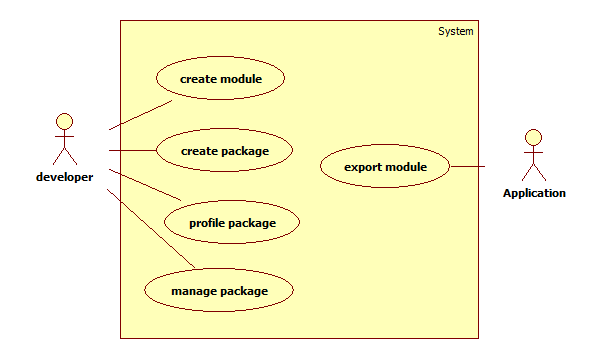


图3.1 模块和包机制用例图

* + 1. **创建和加载模块**

在Node.js中创建一个模块非常简单，因为一个文件就是一个模块。创建模块时需要一个对外公开的接口exports，外部加载模块时利用require获取该模块的接口。开发人员创建和加载模块的用例如3.2图和3.3图所示。



图3.2 创建模块用例描述



图3.3 加载模块用例描述

* + 1. **创建和调用包**

包通常是一些模块的集合，在模块的基础上提供了更高层的抽象，相当于提供了一些固定接口的函数库。创建包时需要有一个模块，或者在创建包之后在包内创建一个新的模块。调用时依旧使用require方法来获取包的接口。创建和调用包的用例如下图3.4和3.5所示。



图3.4 创建包用例描述



图3.5 调用包用例描述

* + 1. **管理包**

NPM是Node.js发布的包管理工具，用于包的发布、传播、依赖控制。NPM提供了命令行工具，使用户方便地下载、安装、升级、删除包，也可以让开发者发布并维护包。

* 1. **网络通信**

在使用PHP等脚本语言开发网站时，必须先搭建一个Apache之类的HTTP服务器，然后通过HTTP服务器的模块加载或CGI调用，才能将PHP脚本执行结果呈现给用户。这样打破了过去js只能在浏览器运行的局面，前后端实现了统一。Node提供了net、dgram、http、https这4个模块，分别用于处理TCP、UDP、HTTP、HTTPS，适用于服务端和客户端。

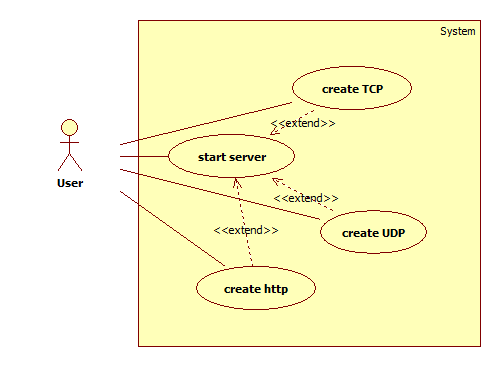


图3.6 网络通信用例图

* + 1. **构建TCP服务**

通过net.createServer(listener)即可创建一个TCP服务器，listener是连接事件connection的侦听器。服务器可以同时和多个客户端保持连接，stream对象用于服务器端和客户端的通信。



图3.7 构建TCP服务用例描述

* + 1. **构建UDP服务**

通过dgram.createSocket(“Udp4”)即可创建UDP套接字。若想让UDP套接字接收网络消息，只需要调用dgram.bind()方法对网卡和端口进行绑定即可。之后可以创建一个用户来和服务器端进行对话。



图3.8 构建UDP服务用例描述

* + 1. **构建HTTP服务**

通过http. createServer()方法即可实现一个HTTP服务器。对于TCP连接的读操作，http模块将其封装为ServerRequest对象。HTTP的相应机制封装了对底层连接的写操作，将其看成一个可写的流对象。对于报头的写入，分为setHeader()和writeHead()两个步骤。报文体部分则是调用res.write()和res.end()方法实现。客户端的实现同服务器端。



图3.9 构建HTTP服务用例描述

* + 1. **构建WebSocket服务**

首先new WebSocket()新建一个WebSocket,然后用socket.onopen()方法在浏览器与服务器端创建WebSocket协议请求，并规定向服务器端发送数据的时间。同时可以通过onmessage()方法接收服务器端传来的数据。

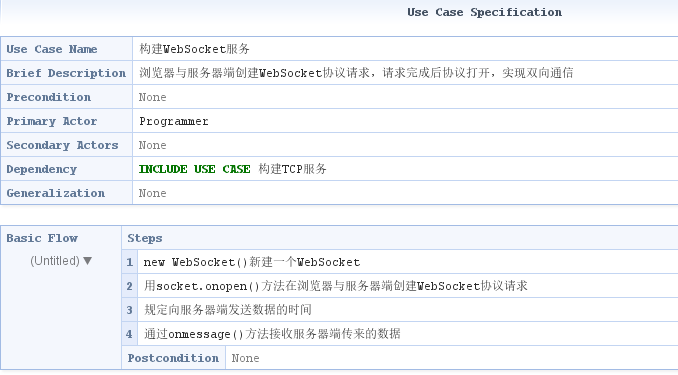


图3.10 构建WebSocket服务用例描述

* 1. **文件系统**

文件系统是文件操作的封装，它提供了文件的读取、写入、更名、删除等文件操作。文件I/O 是由简单封装的标准 POSIX 函数提供的。通过require(‘fs’)使用该模块。 所有的方法都有异步和同步的形式。异步形式始终以完成回调作为它最后一个参数。传给完成回调的参数取决于具体方法，但第一个参数总是留给异常。如果操作成功完成，则第一个参数会是null或undefined。当使用同步形式时，任何异常都会被立即抛出。 可以使用try/catch 来处理异常，或让它们往上冒泡。

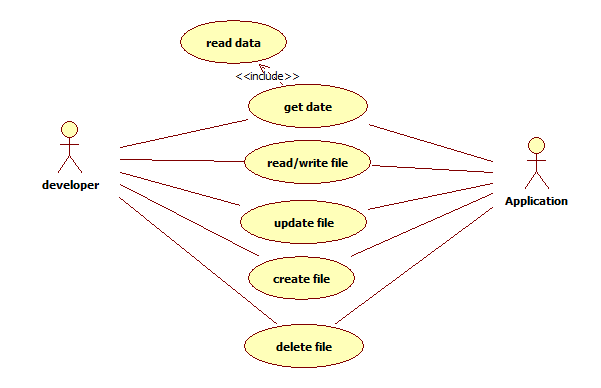


图3.11 文件系统用例图

* + 1. **文件操作**

Fs提供文件的读取、写入、更名、删除、遍历目录、链接等文件操作。每种操作都有异步和同步两种形式。例如，读文件可以用fs.open()和fs.read()来实现，也可以用fs.readFile()来实现。前者相比于后者提供了更底层的接口。

* + 1. **Buffer**

在 ECMAScript 2015 (ES6) 引入TypedArray之前，JavaScript 语言没有读取或操作二进制数据流的机制。Buffer类被引入作为 Node.js API 的一部分，使其可以在 TCP 流和文件系统操作等场景中处理二进制数据流。现在TypedArray已经被添加进 ES6 中，Buffer类以一种更优与更适合 Node.js 用例的方式实现了Unit8Array API。Buffer类可以实现编码类型的转换、Buffer的拼接等功能。

1. **数据需求**

暂无

1. **非功能需求**
   1. **兼容性**
      1. **浏览器兼容**

Node.js不运行在浏览器中，所以不存在JavaScript的浏览器兼容性问题，可以放心使用JavaScript语言的所有特性。

* + 1. **操作系统兼容**

Node.js兼容Windows、Linux、Mac OS X操作系统。

* 1. **高效性**

Node.js非阻塞模式的IO处理给Node.js带来在相对低系统资源耗用下的高性能与出众的负载能力，非常适合用作依赖其它IO资源的中间层服务。Node.js轻量高效，可以认为是数据密集型分布式部署环境下的实时应用系统的完美解决方案。

* 1. **容错性**

JavaScript提供uncaughtException容错机制，在一定程度上提高可程序的稳定性。但是Nodejs在容错方面仍存在一些不足。例如，当异步回调中出现异常，不管相应的 error 的事件有没有被订阅，整个进程都会挂掉。虽然在早期Node.js 中提出Domain 机制，用于隔离错误域，希望解决这个问题，但是由于机制方面的问题，导致这套机制无法像预期的那样完美运作。

* 1. **可扩展性**

Node.js支持C/C++扩展模块的编写。用户可根据自己的需求编写相应的功能模块放入核心模块中，以提高效率。

1. **运行需求**
   1. **硬件接口**

官方暂无明确要求

* 1. **软件接口**

官方暂无明确要求

1. **应用场景**
   1. **I/O 密集型**

从单线程的角度来说，Node处理I/O的能力是很出色的。Node面向网络且擅长并行I/O，能够有效地组织起更多的硬件资源，从而提供更多好的服务。I/0密集的优势主要在于Node利用事件循环的处理能力，而不是启动每一个线程为每一个请求服务，资源占用极少。

* 1. **CPU密集型**

Node.js的引擎是V8，实际上V8的执行效率是非常高的。关于CPU密集型应用，Node的异步I/O已经解决了在单线程上CPU与I/O之间阻塞无法重叠利用的问题，I/O阻塞造成的性能浪费远比CPU的影响小。

对于长时间的运算或者纯运算的场景，或许应该采用多线程的方式进行计算。Node.js虽然没有提供多线程用于计算支持，但也有自己的解决方案。Node.js可以通过编写C/C++扩展的方式更高效的利用CPU,将一些V8不能做到性能极致的地方通过C/C++来实现。如果单线程的Node不能满足需求，可以通过子进程的方式，将一部分Node进程当作常驻服务进程用于计算，然后利用进程间的消息来传递结果，将计算和I/O分离，这样能充分利用多CPU。

* 1. **分布式应用**

分布式应用意味着对可伸缩性的要求非常高。数据平台通常要在一个数据库集群中去寻找需要的数据。Node高效利用并行I/O的过程，也是高效实用数据库的过程

1. **参考文献**

【1】《Node.js开发指南》,BYVoid,人民邮电出版社

【2】《深入浅出Node.js》，朴灵，人民邮电出版社