|  |
| --- |
| buaa_1 |
|  |
| 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  2016-04-05 |

版本变更历史

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| 版本 | 提交日期 | 编制人 | 修改说明 | 审核人 | 版本说明 |
| V1.0 | 2017/3/23 | 温元祯 |  | 李岳檑 | 初稿 |
| V1.1 | 2017/3/27 | 王春柳 | 添加用例说明。 | 李岳檑 | 一稿 |
| V1.2 | 2017/3/31 | 李岳檑  王春柳 | 修改文档格式；  补充文档内容；  修改用例说明。 | 谭伟良 | 一稿修订 |
| V1.3 | 2017/4/05 | 温元祯  李岳檑  王春柳 | 修改用例图；  补全方向和业务需求等内容；修改用例说明 | 谭伟良 | 定稿 |
| V1.4 | 2017/4/21 | 温元祯 | 修改用例图；添加非功能性需求用例描述 | 李岳檑 | 最终定稿 |
|  |  |  |  |  |  |
|  |  |  |  |  |  |
|  |  |  |  |  |  |
|  |  |  |  |  |  |

、

目录

[**1.** **引言** 3](#_Toc479183541)

[**1.1** **目的** 3](#_Toc479183542)

[**1.2** **标识** 3](#_Toc479183543)

[**1.3** **系统概述** 3](#_Toc479183544)

[**1.4** **文档概述** 4](#_Toc479183545)

[**1.5** **术语和缩略词** 4](#_Toc479183546)

[**2.** **业务需求** 5](#_Toc479183547)

[**2.1 构建高性能Web服务器** 5](#_Toc479183548)

[**2.2 使用一门门槛低，符合事件驱动的语言来开发** 5](#_Toc479183549)

[**3.** **工作重点** 6](#_Toc479183550)

[**3.1 异步并发控制** 6](#_Toc479183551)

[**3.2 解决方案初步** 6](#_Toc479183552)

[**4.** **功能需求** 6](#_Toc479183553)

[**4.1** **模块和包** 6](#_Toc479183554)

[**4.1.1** **创建和加载模块** 7](#_Toc479183555)

[**4.1.2** **创建和调用包** 8](#_Toc479183556)

[**4.1.3** **管理包** 9](#_Toc479183557)

[**4.2** **网络通信** 9](#_Toc479183558)

[**4.2.1** **构建TCP服务** 10](#_Toc479183559)

[**4.2.2** **构建UDP服务** 10](#_Toc479183560)

[**4.2.3** **构建HTTP服务** 11](#_Toc479183561)

[**4.2.4** **构建WebSocket服务** 11](#_Toc479183562)

[**4.3** **文件系统** 12](#_Toc479183563)

[**4.3.1** **文件操作** 13](#_Toc479183564)

[**4.3.2** **Buffer** 13](#_Toc479183565)

[**5.** **数据需求** 13](#_Toc479183566)

[**6.** **非功能需求** 13](#_Toc479183567)

[**6.1** **兼容性** 14](#_Toc479183568)

[**6.2** **高效性** 14](#_Toc479183569)

[**6.3** **容错性** 14](#_Toc479183570)

[**6.4** **可扩展性** 14](#_Toc479183571)

[**7.** **运行需求** 14](#_Toc479183572)

[**7.1** **硬件接口** 14](#_Toc479183573)

[**7.2** **软件接口** 14](#_Toc479183574)

[**8.** **应用场景** 14](#_Toc479183575)

[**8.1** **I/O 密集型** 14](#_Toc479183576)

[**9.** **参考文献** 15](#_Toc479183577)

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：V1.3

* 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[1]。

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[3]。



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。

**表 1 专业术语/缩略图描述表**

|  |  |
| --- | --- |
| 术语/缩略语 | 描述 |
| JavaScript | JavaScript一种直译式脚本语言,简称js。 |
| 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" \t "_blank)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 |
| WebSocket | WebSocket协议是基于TCP的一种新的协议。WebSocket最初在HTML5规范中被引用为TCP连接，作为基于TCP的套接字API的占位符。它实现了浏览器与服务器全双工(full-duplex)通信。 |
| NPM | NPM是随同NodeJS一起安装的包管理工具 |
| HTTPs | HTTPS（全称：Hyper Text Transfer Protocol over Secure Socket Layer），是以安全为目标的HTTP通道，简单讲是HTTP的安全版。 |
| UDP | UDP 是User Datagram Protocol的简称， 中文名是用户数据报协议，是OSI（Open System Interconnection，开放式系统互联） 参考模型中一种无连接的传输层协议。 |

1. **业务需求**

**2.1 构建高性能Web服务器**

在Web领域，大多数的编程语言需要专门的Web服务器作为容器，如ASP需要IIS作为服务器，PHP需要搭载Apache，JSP需要tomcat服务器。Node需要做的是构建一个高性能的，事件驱动，非阻塞I/O的Web服务器[2]。

**2.2 使用一门门槛低，符合事件驱动的语言来开发**

Javascript开发门槛低，在后端部分一直没有市场，可以说历史包袱为零，为其导入非阻塞I/O库没有额外阻力，另外，Javascript在浏览器中有广泛的事件驱动方面的应用，暗合Node基于事件驱动的需求。

1. **工作重点**

**3.1 异步并发控制**

异步编程解决的问题无外乎是保持异步的性能优势，提升编程体验，但仍然会有别的问题，那就是并发量太大导致下层服务器崩溃，所以需要过载保护。

**3.2 解决方案初步**

1.通过一个队列来控制并发量

2.如果当前的异步调用不超过规定的最大值，就从队列中取出执行，

3．如果达到最大值，就讲异步调用暂时放在队列中。

4.一旦有异步调用结束，就从队列中取一个新的异步调用。

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。

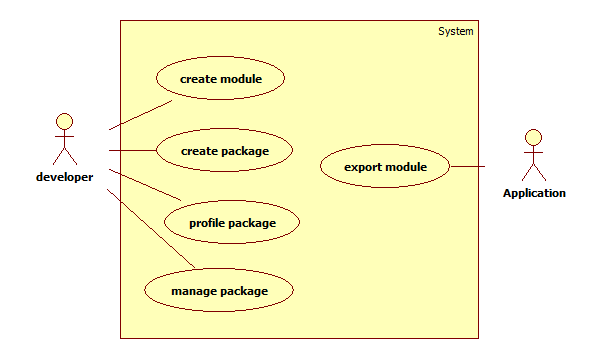


图4.1 模块和包机制用例图

* + 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[1]。



图4.2 创建模块用例描述



图4.3 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4.4 创建包用例描述



图4.5 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。

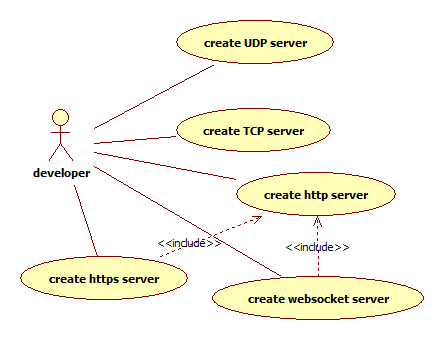


图4.6 创建服务器用例图

* + 1. **构建TCP服务器**

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4.7 构建TCP服务器用例描述

* + 1. **构建UDP服务器**

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4.8 构建UDP服务器用例描述

* + 1. **构建HTTP服务器**

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[2]。



图4.9 构建HTTP服务器用例描述



图4.10 构建HTTPS服务器用例描述

* + 1. **构建WebSocket服务器**

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4.11 构建WebSocket服务器用例描述

* 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。

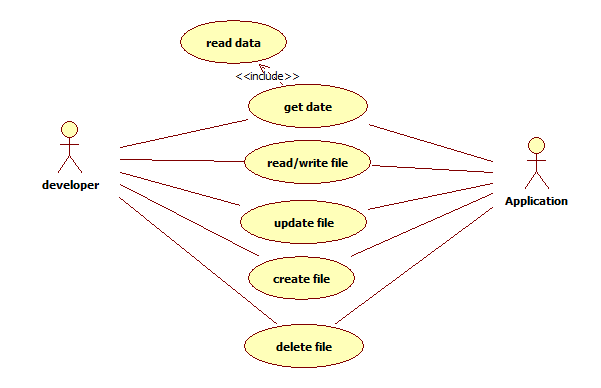


图4.12 文件系统用例图

* + 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]。

* + 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**

开发者在使用Node时，不需要考虑操作系统的差别，Node在windows,linux上可以实现底层的支持。

表6.1 兼容性用例描述

|  |  |
| --- | --- |
| 场景 | 可能的值 |
| 源 | web开发人员 |
| 刺激 | 在不同操作系统上使用该系统的需求 |
| 制品 | 产出不同操作系统下的系统版本 |
| 环境 | 不同操作系统下 |
| 响应 | Nodejs在操作系统和上层模块系统之间构建了一层libuv平台架构。在不同操作系统下，Nodejs应该无需采用该软件准备的配置文件之外的规定就可以直接使用 |
| 响应度量 | 可移植的程度、在不同操作系统下需要更改的配置文件个数、除了配置文件之外所需的其它文件 |

* 1. **高效性**

开发者在使用Node时,可以从语言层面很自然的进行并行I/O操作，每个调用无需等待之前的I/O调用结束，从而在编程模型上极大提升效率。

表6.2 高效性用例描述

|  |  |
| --- | --- |
| 场景 | 可能的值 |
| 源 | Web开发人员、时间、响应 |
| 刺激 | 有快速响应、或高效处理并发事件的需求 |
| 制品 | 高效的处理事件的机制 |
| 环境 | 正常模式 |
| 响应 | Nodejs保持单线程，解决死锁问题，减少性能的开销；Nodejs使用异步I/O和事件回调机制，极大的提高了处理事件的效率 |
| 响应度量 | 最大吞吐量 |

* 1. **容错性**

当程序出现错误时，开发者会被通知到发生了什么错误，而不是直接将这个错误的上下文丢掉或者伴随着错误而退出程序。

表6.3 容错性用例描述

|  |  |
| --- | --- |
| 场景 | 可能的值 |
| 源 | Web开发人员、响应 |
| 刺激 | 开发人员代码编写出错、系统中出现未处理的异常 |
| 制品 | 在开发人员出错的情况下，系统能继续响应，不会出现崩溃的现象 |
| 环境 | 系统出现异常的情况 |
| 响应 | Nodejs采用child\_process来应对单线程在健壮性的问题。通过catch/exception机制来通知开发者出现异常 |
| 响应度量 | 容错成功的概率 |

* 1. **可扩展性**

开发者可根据自己的需求编写相应的功能模块放入核心模块中。

表6.4 可扩展性用例描述

|  |  |
| --- | --- |
| 场景 | 可能的值 |
| 源 | Web开发人员 |
| 刺激 | 有添加功能或改进功能的需求 |
| 制品 | 扩展模块 |
| 环境 | 在设计时、开发时、编译时 |
| 响应 | Nodejs应该提供可扩展模块的接口、编写规范、以及加载、运行方法 |
| 响应度量 | 规范的可扩展模块可执行的概率 |

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。

如果有数百万玩家同时在线玩游戏，而且他们处于游戏中的不同位置，那么很快就会生成海量信息。Node 是这种场景的一种很好的解决方案，因为它能采集游戏生成的数据，对数据进行最少的合并，然后对数据进行排队，以便将它们写入数据库。

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

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