|  |
| --- |
| buaa_1 |
| 测试需求规格说明书 |
| 基于Node.js的分析与应用 |
|  |
|  |
| 北京航空航天大学 |
| 2017-05-15 |

版本变更历史

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| 版本 | 提交日期 | 主要编制人 | 内容说明 | 审核人 | 版本说明 |
| V1.0.0 | 2017/5/15 | 王春柳 |  |  | 初稿 |
|  |  |  |  |  |  |
|  |  |  |  |  |  |
|  |  |  |  |  |  |
|  |  |  |  |  |  |
|  |  |  |  |  |  |
|  |  |  |  |  |  |
|  |  |  |  |  |  |
|  |  |  |  |  |  |

目录

[1.前言 5](#_Toc482807667)

[1.1目的 5](#_Toc482807668)

[1.2软件测试的主要内容 5](#_Toc482807669)

[1.3文档描述 5](#_Toc482807670)

[1.4测试用例与需求用例参照表 5](#_Toc482807671)

[1.5基于异步过载保护的DNS系统测试 6](#_Toc482807672)

[2.功能性需求测试描述 7](#_Toc482807673)

[2.1创建模块 7](#_Toc482807674)

[2.1.1创建模块测试描述 7](#_Toc482807675)

[2.1.2测试用例 7](#_Toc482807676)

[2.2加载模块 8](#_Toc482807677)

[2.2.1加载模块测试描述 8](#_Toc482807678)

[2.2.2测试用例 8](#_Toc482807679)

[2.3创建包 9](#_Toc482807680)

[2.3.1创建包测试描述 9](#_Toc482807681)

[2.3.2测试用例 9](#_Toc482807682)

[2.4加载包 10](#_Toc482807683)

[2.4.1加载包测试描述 10](#_Toc482807684)

[2.4.2测试用例 11](#_Toc482807685)

[2.5管理包 12](#_Toc482807686)

[2.5.1管理包测试描述 12](#_Toc482807687)

[2.5.2测试用例 12](#_Toc482807688)

[2.6构建TCP服务器 13](#_Toc482807689)

[2.6.1构建TCP服务器测试描述 13](#_Toc482807690)

[2.6.2测试用例 13](#_Toc482807691)

[2.7构建UDP服务器 14](#_Toc482807692)

[2.7.1构建UDP服务器测试描述 14](#_Toc482807693)

[2.7.2测试用例 14](#_Toc482807694)

[2.8构建HTTP服务器 16](#_Toc482807695)

[2.8.1构建HTTP服务器测试描述 16](#_Toc482807696)

[2.8.2测试用例 16](#_Toc482807697)

[2.9构建WebSocket服务器 17](#_Toc482807698)

[2.9.1构建WebSocket服务器测试描述 17](#_Toc482807699)

[2.9.2测试用例 18](#_Toc482807700)

[2.10文件系统 19](#_Toc482807701)

[2.10.1文件系统测试描述 19](#_Toc482807702)

[2.10.2测试用例 19](#_Toc482807703)

[3.非功能需求测试描述 21](#_Toc482807704)

[3.1兼容性测试 21](#_Toc482807705)

[3.1.1测试策略描述 21](#_Toc482807706)

[3.1.2测试用例 21](#_Toc482807707)

[3.2高效性测试 22](#_Toc482807708)

[3.2.1测试策略描述 22](#_Toc482807709)

[3.2.2测试用例 22](#_Toc482807710)

[3.3容错性测试 23](#_Toc482807711)

[3.3.1测试策略描述 23](#_Toc482807712)

[3.3.2测试用例 23](#_Toc482807713)

[3.4可扩展性 24](#_Toc482807714)

[3.4.1测试策略描述 24](#_Toc482807715)

[3.4.2测试用例 24](#_Toc482807716)

[4.基于异步过载保护的DNS系统测试 26](#_Toc482807717)

[4.1 node单元测试介绍 26](#_Toc482807718)

[4.2对DNS系统的单元测试 27](#_Toc482807719)

[4.2.1代码覆盖率测试 27](#_Toc482807720)

[4.2.2系统功能测试 27](#_Toc482807721)

**1.前言**

**1.1目的**

本文档主要用于分析本软件工程综合实验小组对Node.js以及基于异步过载保护模块的DNS系统进行测试的主要内容，在此基础上对测试用例进行了初步的设计。

**1.2软件测试的主要内容**

软件测试是一种实际输出与预期输出间的审核或者比较过程。软件测试的经典定义是：在规定的条件下对程序进行操作，以发现程序错误，衡量软件质量，并对其是否能满足设计要求进行评估的过程。本次软件测试阶段的主要工作如下：

总结项目实现的内容，结合需求分析阶段的内容，分析测试需求，编写测试计划及测试规格说明书；

①编写有效的、覆盖面广的测试用例；

②研究相关测试技术；

③按计划实施测试工作，提交测试报告。

**1.3文档描述**

本次测试需求规格说明书主要参照《需求规格说明书》以及已经实现的项目内容，给出了需求用例与测试用例的对照表，对于每个测试用例，先给出测试策略的描述，然后按照RUCM4test标准说明测试用例。主要通过功能需求以及非功能需求两个大的方面进行测试。

**1.4测试用例与需求用例参照表**

本次测试严格遵照《需求规格说明书》以及项目实现内容来进行设计与实现，测试用例与需求用例的对照表如下：

|  |  |  |
| --- | --- | --- |
| 模块 | 需求用例 | 测试用例 |
| 模块与包 | 创建模块 | 创建模块测试 |
| 加载模块 | 加载模块测试 |
| 创建包 | 创建包测试 |
| 加载包 | 加载包测试 |
| 管理包 | 管理包测试 |
| 网络通信 | 构建TCP服务器 | 构建TCP服务器测试 |
| 构建UDP服务器 | 构建UDP服务器测试 |
| 构建HTTP服务器 | 构建HTTP服务器测试 |
| 构建HTTPS服务器 | 构建HTTPS服务器测试 |
| 构建WebSocket服务器 | 构建WebSocket服务器测试 |
| 文件系统 | 写入文件 | 写入文件测试 |
| 读取文件 | 读取文件测试 |
| 更名文件 | 更名文件测试 |
| 删除文件 | 删除文件测试 |
| 创建目录 | 创建目录测试 |
| 删除目录 | 删除目录测试 |
| 非功能性 | 兼容性 | 兼容性测试 |
| 高效性 | 高效性测试 |
| 容错性 | 容错性测试 |
| 可扩展性 | 可扩展性测试 |

**1.5基于异步过载保护的DNS系统测试**

除了上述对node.js的整体测试，我们将基于之前扩展的异步过载保护模块的DNS系统进行单元测试，单元测试可以包括两个方面，结构测试和功能测试。我们可以对异步过载保护模块进行单元测试，也可以对整个基于异步过载保护的DNS系统来进行单元测试。第四章将对关于node的单元测试进行详细说明。

**2.功能性需求测试描述**

**2.1创建模块**

**2.1.1创建模块测试描述**

测试员通过创建一个新的模块并对其加载来测试创建模块功能能否完成。例如本次实验扩展部分编写的异步过载保护模块，即为创建一个新的模块。若加载模块失败，可能加载模块功能异常，也有可能创建模块失败，在这里将创建与加载分开描述完成与需求规格说明书中内容的一一对应。

**2.1.2测试用例**

|  |  |  |
| --- | --- | --- |
| Test Case Specification | | |
| Name | 创建模块测试 | |
| Brief Description | 测试能否在系统中创建一个新的模块 | |
| Precondition | 系统已启动 | |
| Tester | --- | |
| Dependency | None | |
| Basic Flow  (Test Sequence) | Steps | |
| 1 | 程序员VALIDATES THAT系统启动成功 |
| 2 | 测试员编写新模块 |
| 3 | 系统VALIDATES THAT加载模块成功 |
| Postcondition  (Test Oracle) | 模块创建成功 |
| Specific Alternative Flows  (Test Sequence) | RFS 1 | |
| 1 | 系统启动失败 |
| 2 | ABORT |
| Postcondition  (Test Sequence) | 无法测试系统能否创建新模块 |
| Specific Alternative Flows  (Test Sequence) | RFS 3 | |
| 1 | 编写的新模块不能够被加载 |
| 2 | ABORT |
| Postcondition  (Test Sequence) | 创建模块失败的可能情况被测试 |

**2.2加载模块**

**2.2.1加载模块测试描述**

测试员通过加载已经被创建的模块来进行测试，可以通过加载自己创建的模块，例如本次实验扩展部分的异步过载保护模块，也可以通过加载Node.js管理包中的模块进行测试，以排除加载模块失败的可能原因是创建模块失败引发的可能性。

**2.2.2测试用例**

|  |  |  |
| --- | --- | --- |
| Test Case Specification | | |
| Name | 加载模块测试 | |
| Brief Description | 测试系统能否完成加载模块的功能 | |
| Precondition | 系统已启动 | |
| Tester | --- | |
| Dependency | None | |
| Basic Flow  (Test Sequence) | Steps | |
| 1 | 测试员VALIDATES THAT系统启动成功 |
| 2 | 测试员准备好待测试的模块以及调用该模块的程序 |
| 3 | 测试员运行程序 |
| 4 | 程序员VALIDATES THAT显示结果正确 |
| Postcondition  (Test Oracle) | 模块加载成功 |
| Specific Alternative Flows  (Test Sequence) | RFS 1 | |
| 1 | 系统启动失败； |
| 2 | ABORT |
| Postcondition  (Test Sequence) | 无法测试系统能否加载模块 |
| Specific Alternative Flows  (Test Sequence) | RFS 4 | |
| 1 | 显示的结果与正常结果不相符或报错 |
| 2 | ABORT |
| Postcondition  (Test Sequence) | 模块加载失败的可能情况被测试 |

**2.3创建包**

**2.3.1创建包测试描述**

包中可以包括一个或多个模块，包是模块的集合，包中必须包含包描述文件。如果在我们编写的异步过载保护模块中添加上包描述文件，那么它们的整体就可以称之为一个完整的包了。测试员可以通过在异步过载保护模块的基础上添加包描述文件并对其进行加载来测试可能创建包失败的发生情况。创建包失败可能与模块对错有关也可能与包描述文件对错有关，在两者都正确的基础上出现创建包失败的情况即为系统本身不能满足该功能，即测试失败。

**2.3.2测试用例**

|  |  |  |
| --- | --- | --- |
| Test Case Specification | | |
| Name | 创建包测试 | |
| Brief Description | 测试能否在系统中创建一个新的包 | |
| Precondition | 系统已启动 | |
| Tester | --- | |
| Dependency | None | |
| Basic Flow  (Test Sequence) | Steps | |
| 1 | 程序员VALIDATES THAT系统启动成功 |
| 2 | 测试员在模块的基础上添加包描述文件 |
| 3 | 系统VALIDATES THAT加载包成功 |
| Postcondition  (Test Oracle) | 模块创建已完成 |
| Specific Alternative Flows  (Test Sequence) | RFS 1 | |
| 1 | 系统启动失败； |
| 2 | ABORT |
| Postcondition  (Test Sequence) | 无法测试系统能否创建新模块 |
| Specific Alternative Flows  (Test Sequence) | RFS 3 | |
| 1 | 创建的包不能够被加载 |
| 2 | ABORT |
| Postcondition  (Test Sequence) | 创建包失败的可能情况被测试 |

**2.4加载包**

**2.4.1加载包测试描述**

在完成包创建之后需要对其进行加载，才能够知道创建的包是否可用，如果一个满足规范的包文件不能够被正常使用说明创建的包可能不正确，同时也有可能是加载过程出现了问题，创建和加载的测试是一个相辅相成的关系。对于加载包的测试可以通过require()方法进行对包的加载，程序结果执行不正确或不能被执行表示错误原因可能是加载包功能异常，表示测试失败，但这只是可能原因，不排除包创建失败的可能情况。

**2.4.2测试用例**

|  |  |  |
| --- | --- | --- |
| Test Case Specification | | |
| Name | 加载包测试 | |
| Brief Description | 测试系统能否完成加载包的功能 | |
| Precondition | 系统已启动 | |
| Tester | --- | |
| Dependency | None | |
| Basic Flow  (Test Sequence) | Steps | |
| 1 | 测试员VALIDATES THAT系统启动成功 |
| 2 | 测试员准备好待测试的包以及调用该包的程序 |
| 3 | 测试员运行程序 |
| 4 | 程序员VALIDATES THAT显示结果正确 |
| Postcondition  (Test Oracle) | 包加载成功 |
| Specific Alternative Flows  (Test Sequence) | RFS 1 | |
| 1 | 系统启动失败； |
| 2 | ABORT |
| Postcondition  (Test Sequence) | 无法测试系统能否加载包 |
| Specific Alternative Flows  (Test Sequence) | RFS 4 | |
| 1 | 显示的结果与正常结果不相符或报错 |
| 2 | ABORT |
| Postcondition  (Test Sequence) | 加载包失败的可能情况被测试 |

**2.5管理包**

**2.5.1管理包测试描述**

NPM是Node.js发布的包管理工具，这里提到的管理包包括发布、下载、安装、升级、维护和删除包等功能，NPM提供命令行工具来方便用户完成这几个功能。测试人员可以通过对上述几个功能的挨个试验来完成对管理包这个整体功能的测试，下面仅对安装包这一功能进行测试用例的描述，其他内容相似可省略。

**2.5.2测试用例**

|  |  |  |
| --- | --- | --- |
| Test Case Specification | | |
| Name | 安装包测试 | |
| Brief Description | 测试系统能否完成下载包的功能 | |
| Precondition | NPM已安装成功 | |
| Tester | --- | |
| Dependency | None | |
| Basic Flow  (Test Sequence) | Steps | |
| 1 | 测试员VALIDATES THAT命令行工具启动成功 |
| 2 | 测试员使用npm install指令安装需要的包 |
| 3 | 命令行VALIDATES THAT显示安装包的信息 |
| Postcondition  (Test Oracle) | 包安装成功 |
| Specific Alternative Flows  (Test Sequence) | RFS 1 | |
| 1 | 命令行工具启动失败； |
| 2 | ABORT |
| Postcondition  (Test Sequence) | 无法测试能否安装包 |
| Specific Alternative Flows  (Test Sequence) | RFS 3 | |
| 1 | 未显示安装包的信息或显示安装失败的提示 |
| 2 | ABORT |
| Postcondition  (Test Sequence) | 安装包失败的情况被测试 |

**2.6构建TCP服务器**

**2.6.1构建TCP服务器测试描述**

因为通过net.createServer(listener)即可创建一个TCP服务器，listener表示连接事件connection的侦听器，所以测试人员可以通过实践这一指令并且与客户端成功进行交流来检测系统能够成功构建TCP服务器。

**2.6.2测试用例**

|  |  |  |
| --- | --- | --- |
| Test Case Specification | | |
| Name | 构建TCP服务器测试 | |
| Brief Description | 测试系统能否完成构建TCP服务器这一功能 | |
| Precondition | 系统已经启动 | |
| Tester | --- | |
| Dependency | None | |
| Basic Flow  (Test Sequence) | Steps | |
| 1 | 测试员VALIDATES THAT系统启动成功 |
| 2 | 测试员利用net.createServer(listener)创建服务器 |
| 3 | 编写服务器端的相关代码 |
| 4 | 测试人员利用Telnet工具与服务器进行交流 |
| 5 | 系统VALIDATES THAT显示编写的服务器端的内容 |
| Postcondition  (Test Oracle) | 构建TCP服务器成功 |
| Specific Alternative Flows  (Test Sequence) | RFS 1 | |
| 1 | 系统启动失败； |
| 2 | ABORT |
| Postcondition  (Test Sequence) | 无法测试能否创建TCP服务器 |
| Specific Alternative Flows  (Test Sequence) | RFS 5 | |
| 1 | 显示的内容不是构建的服务器端的内容或抛出异常 |
| 2 | ABORT |
| Postcondition  (Test Sequence) | 构建TCP服务器失败的情况被测试 |

**2.7构建UDP服务器**

**2.7.1构建UDP服务器测试描述**

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，这就完成了UDP服务器端的创建。

测试人员可以通过对上述构建UDP服务器的流程进行实践来完成对Node.js系统这一功能的测试。完整的测试还需要包括构建UDP客户端，通过客户端与服务器端的连接交互来反映服务器端构建成功与否，在这里客户端的测试不在我们需要测试的范围之内。

**2.7.2测试用例**

|  |  |  |
| --- | --- | --- |
| Test Case Specification | | |
| Name | 构建UDP服务器测试 | |
| Brief Description | 测试系统能否完成构建UDP服务器这一功能 | |
| Precondition | 系统已经启动 | |
| Tester | --- | |
| Dependency | None | |
| Basic Flow  (Test Sequence) | Steps | |
| 1 | 测试员VALIDATES THAT系统启动成功 |
| 2 | 测试员创建UDP套接字 |
| 3 | 测试员调用dgram.bind()方法对网卡与端口进行绑定 |
| 4 | 测试人员构建UDP客户端与服务器端进行对话 |
| 5 | 系统VALIDATES THAT显示编写的服务器端的内容 |
| Postcondition  (Test Oracle) | 构建UDP服务器成功 |
| Specific Alternative Flows  (Test Sequence) | RFS 1 | |
| 1 | 系统启动失败； |
| 2 | ABORT |
| Postcondition  (Test Sequence) | 无法测试能否创建UDP服务器 |
| Specific Alternative Flows  (Test Sequence) | RFS 5 | |
| 1 | 显示的内容不是构建的服务器端的内容或抛出异常 |
| 2 | ABORT |
| Postcondition  (Test Sequence) | 构建UDP服务器失败的情况被测试 |

**2.8构建HTTP服务器**

**2.8.1构建HTTP服务器测试描述**

通过http. createServer()方法即可实现一个HTTP服务器。我们可以采用curl工具中的获取报文方法来显示该网络通信的所有报文信息。正确的报文信息应该包括四个部分, ，一部分为经典的 TCP的3次握手过程，第二部分是在完成握手之后，客户端向服务器端发送请求报文，第三部分是服务器端处理完后向客户端发送响应内容，最后部分是结束回话的信息。

测试人员对于创建完的HTTP服务器可以采用获取报文的方式来获取该服务器的所有报文信息，通过检测创建的服务器是否满足报文的正确格式来检测创建的服务器正确与否。

HTTPS服务器是在HTTP服务器的基础上添加了SSL加密协议，构建过程类似,下文将省略对创建HTTPS服务器的测试描述。

**2.8.2测试用例**

|  |  |  |
| --- | --- | --- |
| Test Case Specification | | |
| Name | 构建HTTP服务器测试 | |
| Brief Description | 测试系统能否完成构建HTTP服务器这一功能 | |
| Precondition | 系统已经启动 | |
| Tester | --- | |
| Dependency | None | |
| Basic Flow  (Test Sequence) | Steps | |
| 1 | 测试员VALIDATES THAT系统启动成功 |
| 2 | 测试员通过http. createServer()方法创建服务器 |
| 3 | 测试员通过curl工具获取报文信息 |
| 4 | 系统VALIDATES THAT显示报文信息 |
| 5 | 测试员VALIDATES THAT检测报文信息正确 |
| Postcondition  (Test Oracle) | 构建HTTP服务器成功 |
| Specific Alternative Flows  (Test Sequence) | RFS 1 | |
| 1 | 系统启动失败； |
| 2 | ABORT |
| Postcondition  (Test Sequence) | 无法测试能否创建HTTP服务器 |
| Specific Alternative Flows  (Test Sequence) | RFS 4 | |
| 1 | 未显示报文信息或抛出异常、错误提示 |
| 2 | ABORT |
| Postcondition  (Test Sequence) | 构建HTTP服务器失败的情况被测试 |
| Specific Alternative Flows  (Test Sequence) | RFS 5 | |
| 1 | 显示的报文信息与构建的服务器信息不符 |
| 2 | ABORT |
| Postcondition  (Test Sequence) | 构建HTTP服务器失败的情况被测试 |

**2.9构建WebSocket服务器**

**2.9.1构建WebSocket服务器测试描述**

根据需求规格书中的内容可知构建WebSocket的主要流程为：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()方法接收客户端传来的数据，send()方法向客户端发送数据。

WebSocket的很大优点就是WebSocket服务器端可以推送数据到客户端，远比HTTP请求响应模式更灵活高效。对于构建WebSocket服务器的测试我们可以采用一种极为简单的方式：测试人员将编写的构建服务器的代码保存在.js文件中，并在该目录下创建与之相关的index.htm文件，通过命令行工具执行.js文件，再在浏览器中打开编码构建服务器时使用的地址，若出现.htm文件中的内容说明测试通过，反之则失败。

**2.9.2测试用例**

|  |  |  |
| --- | --- | --- |
| Test Case Specification | | |
| Name | 构建WebSocket服务器测试 | |
| Brief Description | 测试系统能否完成构建WebSocket服务器这一功能 | |
| Precondition | 系统已经启动 | |
| Tester | --- | |
| Dependency | None | |
| Basic Flow  (Test Sequence) | Steps | |
| 1 | 测试员VALIDATES THAT系统启动成功 |
| 2 | 测试员编写创建WebSocket服务器代码 |
| 3 | 测试员将代码保存为.js文件 |
| 4 | 测试员编写.htm文件，包含要显示在浏览器中的内容 |
| 5 | 测试员执行.js文件 |
| 6 | 在浏览器中打开服务器地址 |
| 7 | 测试员VALIDATES THAT显示内容与.htm中内容相符 |
| Postcondition  (Test Oracle) | 构建WebSocket服务器成功 |
| Specific Alternative Flows  (Test Sequence) | RFS 1 | |
| 1 | 系统启动失败； |
| 2 | ABORT |
| Postcondition  (Test Sequence) | 无法测试能否创建HTTP服务器 |
| Specific Alternative Flows  (Test Sequence) | RFS 7 | |
| 1 | 内容不符或抛出异常、错误提示 |
| 2 | ABORT |
| Postcondition  (Test Sequence) | 构建WebSocket服务器失败的情况被测试 |

**2.10文件系统**

**2.10.1文件系统测试描述**

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。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。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。

对于文件系统的测试就是根据需求规格说明书中的内容对文件系统的几个功能一一进行实际操作，检测是否可以完成操作，由于几个操作都极其类似，所以现对文件删除操作进行测试用例描述，对于其他几个操作的测试用例的描述说明可以省略。

**2.10.2测试用例**

|  |  |  |
| --- | --- | --- |
| Test Case Specification | | |
| Name | 删除文件测试 | |
| Brief Description | 测试系统能否完成删除文件的功能 | |
| Precondition | 系统已经启动 | |
| Tester | --- | |
| Dependency | None | |
| Basic Flow  (Test Sequence) | Steps | |
| 1 | 测试员VALIDATES THAT系统启动成功 |
| 2 | 测试员确认文本库中要删除的文件 |
| 3 | 测试员将命令行中使用删除语句选定文件进行删除 |
| 4 | 测试员VALIDATES THAT文本库中文件不存在 |
| Postcondition  (Test Oracle) | 构建WebSocket服务器成功 |
| Specific Alternative Flows  (Test Sequence) | RFS 1 | |
| 1 | 系统启动失败； |
| 2 | ABORT |
| Postcondition  (Test Sequence) | 无法测试能否删除文件 |
| Specific Alternative Flows  (Test Sequence) | RFS 4 | |
| 1 | 文件依然存在或抛出异常 |
| 2 | ABORT |
| Postcondition  (Test Sequence) | 删除文件失败的情况被测试 |

**3.非功能需求测试描述**

**3.1兼容性测试**

**3.1.1测试策略描述**

为了测试系统运行于不同操作系统的能力，测试员分别将系统部署在不同的平台上，分别就基本需求及实现需求模块进行功能测试，正常情况下，系统能通过所有功能测试；失败情况下，系统的功能测试存在未能通过的项目。

**3.1.2测试用例**

|  |  |  |
| --- | --- | --- |
| Test Case Specification | | |
| Name | 兼容性测试 | |
| Brief Description | 分别将系统部署在Windows及Linux操作系统上，测试系统能否通过所有功能测试 | |
| Precondition | 操作系统运行正常，不同系统中部署的项目代码一致 | |
| Tester | ---- | |
| Dependency | None | |
| Test Setup | Name | 兼容性测试准备 |
| Description | 在相应的操作系统中部署本项目系统，配置相应的参数，准备基础数据集及测试数据集 |
| Basic Flow  (Test Setup) | Steps | |
| 1 | 分别在Windows及Linux系统下部署系统 |
| 2 | 配置相应参数 |
| 3 | 准备基础数据集 |
| 4 | 准备测试数据集 |
| Postcondition  (Test Oracle) | 系统成功部署 |
| Basic Flow  (Test Sequence) | Steps | |
| 1 | 测试员在Windows系统中测试所有功能模块项目 |
| 2 | 测试员在Linux系统中测试所有功能模块项目 |
| Postcondition  (Test Oracle) | 系统响应正常 |

**3.2高效性测试**

**3.2.1测试策略描述**

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。测试人员通过输入一个I/O密集型事件将node.js使用并行I/O操作与不使用并行I/O操作的时间长短进行对比。

**3.2.2测试用例**

|  |  |  |
| --- | --- | --- |
| Test Case Specification | | |
| Name | 高效性测试 | |
| Brief Description | 测试人员通过输入一个I/O密集型事件将node.js使用并行I/O操作与不使用并行I/O操作的时间长短进行对比。 | |
| Precondition | 系统运行正常 | |
| Tester | ---- | |
| Dependency | None | |
| Test Setup | Name | 高效性测试准备 |
| Description | 准备相应的I/O密集型事件作为测试的用例 |
| Basic Flow  (Test Setup) | Steps | |
| 1 | 测试员分别启动相应系统 |
| 2 | 测试员准备测试脚本 |
| Postcondition  (Test Oracle) | 系统启动成功且相应测试脚本准备完毕 |
| Basic Flow  (Test Sequence) | Steps | |
| 1 | 测试员在Node.js系统中执行I/O密集型事件，记录所用时长 |
| 2 | 测试员在未使用并行I/O操作的系统中执行I/O密集型事件，记录所用时长 |
| Postcondition  (Test Oracle) | 系统响应正常，第一个的时间大于第二个的时间 |

**3.3容错性测试**

**3.3.1测试策略描述**

容错性表示当代码编写错误时，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。测试人员通过输入一段错误的代码来检验node.js是否会成功响应容错机制抛出错误提示。正常情况下，系统发生异常时会继续响应或给出提示而不会发生系统错误；失败情况下，系统出错甚至出现崩溃情况。

**3.3.2测试用例**

|  |  |  |
| --- | --- | --- |
| Test Case Specification | | |
| Name | 容错性测试 | |
| Brief Description | 测试人员通过输入异常数据来检验系统是否会出错甚至崩溃。 | |
| Precondition | 系统运行正常 | |
| Tester | ---- | |
| Dependency | None | |
| Test Setup | Name | 容错性测试准备 |
| Description | 准备相应的测试用例进行测试 |
| Basic Flow  (Test Setup) | Steps | |
| 1 | 测试员启动系统 |
| 2 | 测试员准备测试用例 |
| Postcondition  (Test Oracle) | 系统启动成功且相应测试用例准备完毕 |
| Basic Flow  (Test Sequence) | Steps | |
| 1 | 测试员在系统执行过程中输入多个不合逻辑或不合法的值 |
| Postcondition  (Test Oracle) | 系统提示输入有误、没有发生崩溃情况 |

**3.4可扩展性**

**3.4.1测试策略描述**

可扩展性表示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，Node.js应该提供可扩展模块的接口、编写规范、以及加载、运行方法。测试人员测试自己编写的规范的可执行的扩展模块能否顺序执行。正常情况下，扩展模块可以正常执行；失败情况下，该扩展模块不能够执行成功。

**3.4.2测试用例**

|  |  |  |
| --- | --- | --- |
| Test Case Specification | | |
| Name | 可扩展性测试 | |
| Brief Description | 测试人员通过测试自己编写的模块能否正常使用来测试Node.js的可扩展性 | |
| Precondition | 系统运行正常 | |
| Tester | ---- | |
| Dependency | None | |
| Test Setup | Name | 可扩展性测试准备 |
| Description | 编写扩展模块作为测试对象 |
| Basic Flow  (Test Setup) | Steps | |
| 1 | 测试员启动系统 |
| 2 | 测试员编写扩展模块 |
| 3 | 测试员编写需要调用该模块才能运行的代码 |
| Postcondition  (Test Oracle) | 系统启动成功且相应测试代码准备完毕 |
| Basic Flow  (Test Sequence) | Steps | |
| 1 | 测试员在Node.js系统中执行编写的程序代码 |
| Postcondition  (Test Oracle) | 系统响应正常，程序执行结果正确 |

**4.基于异步过载保护的DNS系统测试**

**4.1 node单元测试介绍**

单元测试主要包含断言、测试框架、测试用例、测试覆盖率、mock等几个方面。下面对node单元测试中的这几个方面进行具体的介绍。

①断言：用于检查程序在运行时是否满足期望，在node中一旦assert.equal()不满足期望，就会抛出AssertionError异常，整个程序停止执行。

②测试框架：与断言相反，记录下抛出的异常并继续执行，最后生成测试报告，这些任务的承担者就是测试框架。Node中主要使用的单元测试框架是mocha。默认的报告格式为dot，还有json、spec等格式。执行mocha-R命令即可采用这些报告。Json报告因为格式非常通用，多用于将结果传递给其他程序进行处理。而html-cov则用于可视化的观察代码覆盖率。

③测试用例：一个行为或一个功能需要有完善的、多方面的测试用例，一个测试用例包含至少一个断言。测试用例需要通过正向测试和反向测试来保证测试队功能的覆盖，这是最基本的测试用例。

④测试覆盖率：通过不断地给代码添加测试用例，将会不断地覆盖代码的分支和不同的情况。但是如何判断单元测试对代码的覆盖情况我们需要直观的工具来体现。测试覆盖率是单元测试中一个重要指标，它能够概括性的给出整体的覆盖度，也能明确的统计到行的覆盖情况。在node中我们可以选择jscover模块工具，通过npm install jscover -g的方式可以安装该模块。调用jscover lib lib-cov进行源代码的编译。每一行原始代码的前面都有一些\_$jscoverage的出现，它们将会在执行时统计每一行代码被执行了多少次，也即除了统计是否执行外，还能统计次数。在测试中还用到html-cov报告，帮助我们生成一张HTML页面，具体的标出哪一行未执行到，整体覆盖率为多少。通常我们往往会不经意的遗漏掉一些异常情况的覆盖，构造一个错误的输入可以覆盖错误情况，再次执行测试用例以达到100%的覆盖率。除了上述模块也可以采用blanket模块进行覆盖率的测试，其基本原理与上述模块基本相同，但实现过程有所不同。

⑤Mock：对于异常的情况较难实现，大多数异常与输入数据并没有绝对的关系，所以异常相对难以模拟，在测试领域中模拟异常有一个特殊的名词即为mock，我们通过伪造被调用代码方式来测试上层代码的健壮性。Mock的过程比较繁琐，我们使用muk模块解决此事。通过模拟底层方法出现异常的情况，只要检测调用方的输出值是否符合期望即可，无需关注是否是真正的异常。

**4.2对DNS系统的单元测试**

**4.2.1代码覆盖率测试**

我们采用前文提到的blanket模块来进行系统的代码覆盖率测试，blanket将编译的步骤注入在require中，而不需要额外的编译成文件，执行测试时再去引用编译后的文件，它的技巧在require中，它的配置比jscover要简单，只需要在所有测试用例运行之前通过—require选项引入它即可：

mocha --require blanket -R html-cov > coverage.html

当在测试文件中通过require引入一个文件模块时，它将判断这个文件的实际路径，如果符合这个匹配规则就对它进行编译，它的编译与jscover不同，原理与文件模块编译相同。编译流程如下图1所示。使用blanket之后，就无需配置环境变量了，也无需根据环境去判断引入哪种代码。

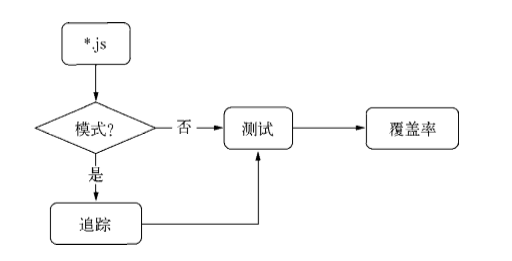


图1 blanket的编译流程

**4.2.2系统功能测试**

DNS（Domain Name System，域名系统），本系统的主要功能就是实现了域名查询功能，对于域名不正确的时候会调用异常处理机制，提示不正确信息请重新输入内容。对于其功能的测试需要遵照测试用例的基本准则，采用正向测试和反向测试的方式：输入正确的域名、输入不正确的域名。

对于正确的域名，系统需要根据输入的域名而输出正确的IP地址，对于不正确的域名，系统需要给出类似于‘输入内容不正确，请重新输入’的提示信息，如下图2所示：

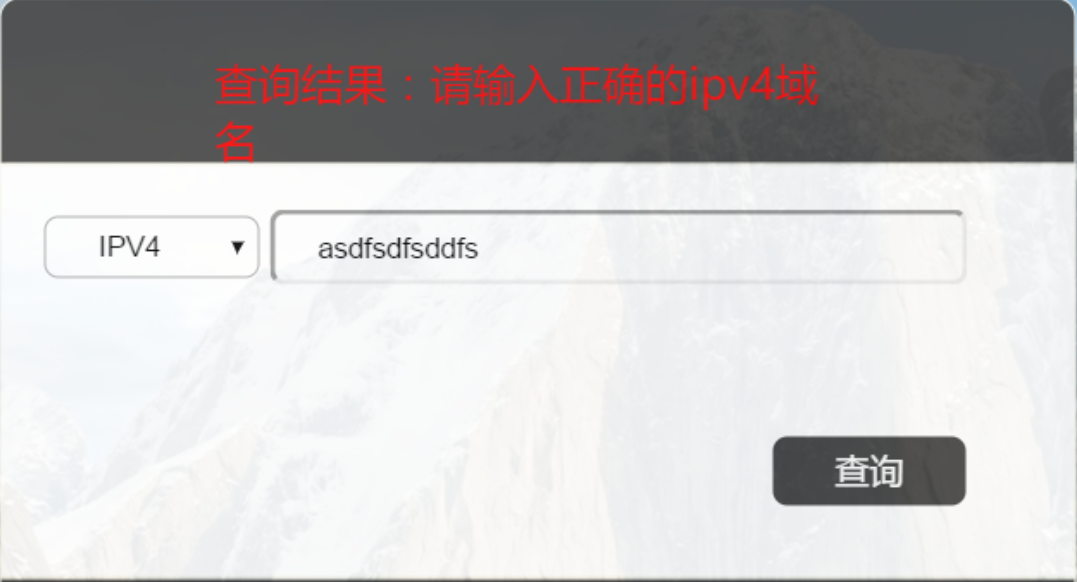


图2 系统对于错误域名所显示的内容