Json Messaging 自述文件

简介

Json Messaging 是使用 node.js 技术构建的发布/订阅类型的消息服务器，具有如下特性：

1、支持 TCP 和 WebSocket 协议；

2、传输帧使用 Json 格式；

3、可以使用正则表达式订阅消息目的地，正则表达式中可以包含“捕获”，所有目的地匹配该正则表达式的消息，连同目的地的“捕获”都将发送到订阅方；

4、一个客户端可以订阅多个消息目的地；

5、为了简化设计，服务器端不持久化消息。

致谢

Json Messaging 消息服务器使用了很多第三方的框架和技术，感谢他们辛勤的工作。

Json：http://www.json.org

node.js：http://nodejs.org

node-uuid：https://github.com/broofa/node-uuid

WebSocket-Node：https://github.com/Worlize/WebSocket-Node

使用方法

1、编译安装 node.js；

2、打开 server/config.js，可以配置 TCP 和 WebSocket 端口；

3、启动消息服务器：node server/server.js。

例子

使用最新版本的 Firefox 或者 Chrome 打开下面的 HTML 文件可以发送和接收消息。

```
<!DOCTYPE html>
<html>
<head>
    <meta charset="utf-8">
    <title>Json Messaging Example</title>
    <style>
        div#output {
            border: 1px solid #000;
            width: 960px;
            height: 450px;
            overflow: auto;
            background-color: #333;
            color: #6cf;
        }

        strong {
            color: #f66;
        }

        input#input {
            border: 1px solid #000;
            width: 640px;
        }

        button {
            border: 1px solid #000;
            width: 100px;
        }
    </style>
    <script>
        // connect to the Json Messaging server and return an 'connection'
object
        function connect(host, port, messageListener, errorListener) {
            window.WebSocket = window.WebSocket || window.MozWebSocket;

            if (!window.WebSocket) {
                alert('Your browser does not support WebSocket.');
                return null;
            }

            var connection = new WebSocket('ws://' + host + ':' + port);

            connection.onmessage = function(message) {
                try {
                    var parsed = JSON.parse(message.data);
                    switch (parsed.type) {
                        case 'message':
                            if (messageListener) {
                                messageListener(parsed.content, parsed.match);
                            }
                            break;
                        case 'error':
                            if (errorListener) {
```

```
                        errorListener(parsed.content);
                    }
                    break;
                default:
                    throw new Error('Unknown message type ' +
parsed.type);
                    break;
            }
        } catch (e) {
            console.warn(e);
            alert(e);
        }
    };

    connection.publish = function(content, destination) {
        connection.send(JSON.stringify({
            type: 'publish',
            destination: destination,
            content: content
        }));
    };

    connection.subscribe = function(destination) {
        connection.send(JSON.stringify({
            type: 'subscribe',
            destination: destination
        }));
    };

    connection.unsubscribe = function(destination) {
        connection.send(JSON.stringify({
            type: 'unsubscribe',
            destination: destination
        }));
    };

    return connection;
}

// the 'connection' object
```

```
       var connection = null;

       var output = null;

       var input = null;

       // initialize
       window.onload = function() {
           output = document.getElementById('output');
           input = document.getElementById('input');

           // connect to the local server
           connection = connect(
                   'localhost',
                   8155,
                   // message handler
                   function(content, match) {
                       output.innerHTML += ('<strong>Message: </strong>' +
content + '<br>\n');
                   },
                   // error handler
                   function(content) {
                       output.innerHTML += ('<strong>Error: </strong>' +
content + '<br>\n');
                   }
           );

           // subscribe a topic
           connection.onopen = function() {
               connection.subscribe('test');
           };
       };

       function _send() {
           connection.publish(input.value, 'test');
       }

       function _clear() {
           output.innerHTML = '';
       }
```

```
   </script>
</head>
<body>
<div id="output"></div>
<input type="text" id="input">
<button id="send" onclick="_send()">Send</button>
<button id="clear" onclick="_clear()">Clear</button>
</body>
</html>
```

下面的 C 程序发送三条 HelloWorld 消息，第一条是英文，第二条是中文，第三条是 Unicode 转义的中文。注意源代码必须以 UTF-8 编码保存。

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

#include <unistd.h>
#include <fcntl.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <arpa/inet.h>

int main(int argc, char ** argv)
{
   int fd;
   struct sockaddr_in addr;
   int ret;
   const char publish_frame_1[] =
   "{\"type\":\"publish\",\"destination\":\"test\",\"content\":\"Hello
World\"}";
   const char publish_frame_2[] =
   "{\"type\":\"publish\",\"destination\":\"test\",\"content\":\"你好世界
\"}";
   const char publish_frame_3[] =
   "{\"type\":\"publish\",\"destination\":\"test\",\"content\":\"\\u4f60\\u
597d\\u4e16\\u754c\"}";


   fd = socket(AF_INET, SOCK_STREAM, IPPROTO_TCP);


   memset(&addr, 0, sizeof(addr));
   addr.sin_family = AF_INET;
   addr.sin_addr.s_addr = inet_addr("127.0.0.1");
```

```
    addr.sin_port = htons(8153);

    ret = connect(fd, (struct sockaddr *)&addr, sizeof(addr));
    printf("%d\n", ret);

    ret = write(fd, publish_frame_1, sizeof(publish_frame_1));
    printf("%d\n", ret);

    ret = write(fd, publish_frame_2, sizeof(publish_frame_2));
    printf("%d\n", ret);

    ret = write(fd, publish_frame_3, sizeof(publish_frame_3));
    printf("%d\n", ret);
}
```

帧格式

消息服务器的应用层数据帧采用 Json 格式，使用 UTF-8 编码的纯文本，在 TCP 协议中，使用'\0'作为帧间分隔，在 WebSocket 协议中遵循 WebSocket 草案标准。

帧格式有 5 类，其中，客户端到服务器端的 3 类，服务器端到客户端的 2 类。

客户端到服务器端

发布帧

客户端发送一条消息到服务器的目的地中，所有连接到服务器并且订阅了该目的地（正则表达式匹配）的客户端都能接收到该消息。

帧格式为：

```
{
    "type": "publish",
    "destination": <消息目的地>,
    "content": <消息内容>
}
```

其中，消息目的地为字符串类型；消息内容同样也必须是 Json 格式的。

订阅帧

客户端订阅服务器的一个目的地，所有匹配该目的地的消息都会发送到该客户端。

帧格式为：

```
{
    "type": "subscribe",
    "destination": <消息目的地>
}
```

其中，消息目的地可以是正则表达式，且正则表达式中可以含有“捕获”，服务器会使用该正则表达式匹配发送的消息目的地，如果符合，则会把该消息连同匹配结果一并发给客户端，在下面的“消息帧”介绍中有具体的例子。

同一个客户端可以订阅多个目的地。

取消订阅帧

客户端取消订阅服务器的一个目的地。

帧格式为：

```
{
   "type": "subscribe",
   "destination": <消息目的地>
}
```

其中，消息目的地等于订阅帧中的消息目的地。

当客户端断开连接后，服务器端会自动取消该客户端的所有订阅。

服务器端到客户端

消息帧

一旦消息目的地匹配，服务器端会把匹配结果连同消息内容发给客户端。

帧格式为：

```
{
   "type": "message",
   "match": <匹配结果>,
   "content": <消息内容>
}
```

匹配结果为一个数组，至少包含一个元素，即订阅的消息目的地；如果订阅的消息目的地是正则表达式且其中含有“捕获”，那么从第二往后的元素为捕获结果，参考 JavaScript 正则表达式规范。

举例：

假设设备的网口状态信息在消息服务中发布，规定目的地格式为：“/devices/<设备名>/<网口名>”；消息内容为：“down”表示停止、“up”表示启动。

下面两个发布帧，表示设备 a 的第 1 个网口停止了，而设备 b 的第 0 个网口启动了：

```
{"type":"publish","destination":"/devices/a/if1","content":"down"}
{"type":"publish","destination":"/devices/b/if0","content":"up"}
```

如果客户端订阅目的地为“/devices/.*”，那么它将能收到所有设备的所有网口的状态消息，接收到的消息帧如下：

```
{"type":"message","match":["/devices/a/if1"],"content":"down"}
{"type":"message","match":["/devices/b/if0"],"content":"up"}
```

如果想在程序中更方便地对设备和网口做分类处理，可以把订阅目的地改为“/devices/(.*)/(.*)”，其中小括号即为“捕获”。

接收到的消息帧会变为：

```
{"type":"message","match":["/devices/a/if1","a","if1"],"content":"down"}
```

```
{"type":"message","match":["/devices/b/if0","b","if0"],"content":"up"}
```

可以看到，match 中增加了捕获的结果。

错误帧

如果服务器端产生错误，例如客户端发送的帧超长、非 Json 格式等，将会向客户端返回错误帧。

帧格式为：

```
{
    "type": "error",
    "content": <错误内容>
}
```

客户端可以对错误进行相应的处理。

源代码结构

server.js

程序入口。

config.js

全局配置，其中的 udpPort 并没有使用，因为 UDP 难以知晓客户端状态，所以不打算实现 UDP 协议。

log.js

控制台日志，相比其它第三方的日志模块的特点是使用简单，而且能够输出日志产生的源代码的位置，便于调试。

protocol.js

协议帧的包装。

exchange.js

负责处理发布和订阅的消息，是服务器代码的核心部分。

tcp.js

TCP 协议的实现。

ws.js

WebSocket 协议的实现。

Json Messaging Readme

Introduction

Json Messaging is a pub/sub messaging server built with node.js, which has following features:

1. Support both TCP and WebSocket protocol.

2. Use Json as frame format.

3. The message destination can be subscribed by regular expression, and the regular expression may contains "capture". Messages matched by the regular expression will be sent to the subscriber, (including capture result if exists).

4. One client can subscribe multiple destinations.

5. The server doesn't persist any message.

Thanks

Many third party frameworks and technology are used by Json Messaging Server. Thanks for their hard work.

Json: http://www.json.org

node.js: http://nodejs.org

node-uuid: https://github.com/broofa/node-uuid

WebSocket-Node：https://github.com/Worlize/WebSocket-Node

Usage

1. Build and install the latest version of node.js.

2. Customize TCP and WebSocket port and other options in "server/config.js".

3. Start the server using command: "node server/server.js".

Examples

Open the following HTML files by latest version of Firefox or Chrome and send/receive messages.

```
<!DOCTYPE html>
<html>
<head>
    <meta charset="utf-8">
    <title>Json Messaging Example</title>
    <style>
        div#output {
            border: 1px solid #000;
            width: 960px;
            height: 450px;
            overflow: auto;
            background-color: #333;
            color: #6cf;
        }

        strong {
            color: #f66;
        }

        input#input {
            border: 1px solid #000;
            width: 640px;
        }

        button {
            border: 1px solid #000;
            width: 100px;
        }
    </style>
    <script>
        // connect to the Json Messaging server and return an 'connection'
object
        function connect(host, port, messageListener, errorListener) {
            window.WebSocket = window.WebSocket || window.MozWebSocket;

            if (!window.WebSocket) {
                alert('Your browser does not support WebSocket.');
                return null;
            }

            var connection = new WebSocket('ws://' + host + ':' + port);

            connection.onmessage = function(message) {
                try {
                    var parsed = JSON.parse(message.data);
                    switch (parsed.type) {
                        case 'message':
                            if (messageListener) {
                                messageListener(parsed.content, parsed.match);
                            }
                            break;
                        case 'error':
                            if (errorListener) {
                                errorListener(parsed.content);
                            }
                            break;
                        default:
```

```
                        throw new Error('Unknown message type ' +
parsed.type);
                        break;
                }
            } catch (e) {
                console.warn(e);
                alert(e);
            }
        };

        connection.publish = function(content, destination) {
            connection.send(JSON.stringify({
                type: 'publish',
                destination: destination,
                content: content
            }));
        };

        connection.subscribe = function(destination) {
            connection.send(JSON.stringify({
                type: 'subscribe',
                destination: destination
            }));
        };

        connection.unsubscribe = function(destination) {
            connection.send(JSON.stringify({
                type: 'unsubscribe',
                destination: destination
            }));
        };

        return connection;
    }

    // the 'connection' object
    var connection = null;

    var output = null;
```

```
       var input = null;

       // initialize
       window.onload = function() {
          output = document.getElementById('output');
          input = document.getElementById('input');

          // connect to the local server
          connection = connect(
                'localhost',
                8155,
                // message handler
                function(content, match) {
                   output.innerHTML += ('<strong>Message: </strong>' +
content + '<br>\n');
                },
                // error handler
                function(content) {
                   output.innerHTML += ('<strong>Error: </strong>' +
content + '<br>\n');
                }
          );

          // subscribe a topic
          connection.onopen = function() {
             connection.subscribe('test');
          };
       };

       function _send() {
          connection.publish(input.value, 'test');
       }

       function _clear() {
          output.innerHTML = '';
       }
    </script>
</head>
<body>
<div id="output"></div>
<input type="text" id="input">
```

```
<button id="send" onclick="_send()">Send</button>
<button id="clear" onclick="_clear()">Clear</button>
</body>
</html>
```

The following C program sends three "Hello World" messages, the first is english, the second is simplified chinese and the third is unicode escaped simplified chinese. Notice these code must be saved using UTF-8 encoding.

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

#include <unistd.h>
#include <fcntl.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <arpa/inet.h>

int main(int argc, char ** argv)
{
   int fd;
   struct sockaddr_in addr;
   int ret;
   const char publish_frame_1[] =
   "{\"type\":\"publish\",\"destination\":\"test\",\"content\":\"Hello 
World\"}";
   const char publish_frame_2[] =
   "{\"type\":\"publish\",\"destination\":\"test\",\"content\":\"你好世界
\"}";
   const char publish_frame_3[] =
   "{\"type\":\"publish\",\"destination\":\"test\",\"content\":\"\\u4f60\\u
597d\\u4e16\\u754c\"}";

   fd = socket(AF_INET, SOCK_STREAM, IPPROTO_TCP);

   memset(&addr, 0, sizeof(addr));
   addr.sin_family = AF_INET;
   addr.sin_addr.s_addr = inet_addr("127.0.0.1");
   addr.sin_port = htons(8153);

   ret = connect(fd, (struct sockaddr *)&addr, sizeof(addr));
   printf("%d\n", ret);
```

```
    ret = write(fd, publish_frame_1, sizeof(publish_frame_1));
    printf("%d\n", ret);

    ret = write(fd, publish_frame_2, sizeof(publish_frame_2));
    printf("%d\n", ret);

    ret = write(fd, publish_frame_3, sizeof(publish_frame_3));
    printf("%d\n", ret);
}
```

Frame Format

Message frames are packed by Json format, and encoded by UTF-8. In TCP protocol, frames are splited by '\0'.

There are 5 types of frame, 3 for client side and 2 for server side.

Client Side

Publish Frame

The client publish a message to the server's destination. All clients connected to the server and subscribed the destination(may be regular expression) will receive the message.

The format is:

```
{
    "type": "publish",
    "destination": <message destination>,
    "content": <message content>
}
```

The message destination is a string, and the message content must also be packed by Json format.

Subscribe Frame

The client subscribe a destination, and all the messages matched this destination will be sent to the subscriber.

The format is:

```
{
    "type": "subscribe",
    "destination": <message destination>
}
```

The message destination may be a regular expression, and may include "capture". If matched, the message, the regular expression and the capture(if exists) will be sent to the subscriber. See examples in "Message Frame".

One client can subscribe multiple destinations.

Unsubscribe Frame

The client unsubscribe a message destination.

The format is:

```
{
   "type": "subscribe",
   "destination": <message destination>
}
```

The message destination is equal to subscribe frame.

The server will automatically unsubscribe all the destinations of the client if it is disconnected.

Server Side

Message Frame

Once the message destination is matched, the server will send the message and the match result to the subscriber.

The format is:

```
{
   "type": "message",
   "match": <match result>,
   "content": <message content>
}
```

The match result is an array, which includes at least one element - the subscribe destination. If the subscribe destination is a regular expression and contains "capture", elements from the second to the last will be capture results. See specification of JavaScript regular expression.

For example:

Assume some devices' network interface status needs to be broadcast. The status message destination is formatted as "/devices/<device name>/<interface name>" and the message content is "down" or "up".

The following two publish frames indicete the first network interface of device "a" is down and the zero interface of device "b" is up.

```
{"type":"publish","destination":"/devices/a/if1","content":"down"}
{"type":"publish","destination":"/devices/b/if0","content":"up"}
```

If some clients subscribe the destination "/devices/.*", they will receive status of all interfaces, all devices. The received message frames are:

```
{"type":"message","match":["/devices/a/if1"],"content":"down"}
{"type":"message","match":["/devices/b/if0"],"content":"up"}
```

If client wants to get detailed information of which device and which interface, it can changed the subscribe destination to "/devices/(.*)/(.*)", and the received message frames will be:

```
{"type":"message","match":["/devices/a/if1","a","if1"],"content":"down"}
{"type":"message","match":["/devices/b/if0","b","if0"],"content":"up"}
```

And you can see the "match" field now contains capture result.

Error Frame

If the server encountered errors, for example: client side frames too long, or invalid Json format, an error frame will be sent to the client.

The format is:

```
{
    "type": "error",
    "content": <error content>
}
```

The client may handle the error.

Source Code Structure

server.js

The server entrance.

config.js

The global configuration. Because UDP protocol is stateless and hard to get client status(connected/disconnected), UDP is not supported and "udpPort" is currently not used.

log.js

The console log utility. Compare to other third party log modules, it is simple and it can display the source code position where log generates.

protocol.js

The wrapper of the protocol frames.

exchange.js

The core part of the server which handles the pub/sub.

tcp.js

TCP implementation.

ws.js

WebSocket implementation.