有且仅有的路

学而不思则罔 思而不学则殆

个人资料

有且仅有

关注 发私信

访问： 126428次

积分： 2221

等级： BLOG 5

排名： 第17190名

原创： 82篇 转载： 4篇

译文： 1篇 评论： 33条

文章搜索

文章分类

1/12 Computer System (2)

2/12 Programming Language (4)

3/12 Operate System (0)

4/12 Software Engineering (0)

5/12 Design Develop (0)

6/12 Network Multimedia (1)

7/12 Database (13)

8/12 Data Structrue (2)

9/12 Algorithm (5)

10/12 Object-Oriented (3)

11/12 Standard Intellectual (0)

12/12 Analysis Design (0)

Java (21)

JavaEE (4)

Spring (2)

JVM (4)

Android (1)

JavaScript (2)

.NET (1)

API (1)

Design Pattern (3)

Project Build Tools (5)

WEB即时通讯/消息推送

标签： 即时通讯 IM WebSocket SocketIO

2017-03-09 22:12 2199人阅读 评论(1)

分类：

JavaEE (3) JavaScript (1)

目录(?) [+]

写在前面

通常进行的Web开发都是由客户端主动发起的请求，然后服务器对请求做出响应并返回给客户端。但是在很多情况下，你也许会希望由服务器主动对客户端发送一些数据。

那么，该如何实现这个需求，或者说该如何向网页推送消息呢？

一、推送方式

我们知道， HTTP/HTTPS 协议是被设计基于“请求-相应”模型的，尽管 HTTP/HTTPS 可以在任何互联网协议或网络上实现，但这里我们只讨论在Internet网上的万维网中的情况。

由于在Internet中， HTTP 协议在传输层使用的是 TCP 协议。由此可知，只要我们能保持 TCP 连接不随一次“请求-响应”结束而结束，使得服务器可以主动发送数据，那么我们就能够实现向网页的数据推送。事与愿违，在2011年 *WebSocket* (详见下文) 出现之前我们对此是无能为力的。

不过，在那时虽然不能直接实现推送，但是还是有曲线救国路线的，基本上有4类这种间接方式。当然现在我们还有了1种直接方式- *WebSocket* ，接下来我来依次介绍下。

模拟推送

1. 轮询 (Polling)

AJAX 定时 (可以使用js的 setTimeout 函数) 去服务器查询是否有新消息，从而进行增量式的更新。这种方式间隔多长时间再查询是个问题，因为性能和即时性是反比关系。间隔太短，海量的请求会拖垮服务器，间隔太长，服务器上的新数据就需要更长的时间才能到达客户机。

关闭

IDE-Eclipse (8)
Server (9)
Linux (5)
English (1)
Office (1)
Common (2)
Good Job (1)
Sth else (2)

文章存档

2017年07月 (1)
2017年04月 (2)
2017年03月 (2)
2017年02月 (1)
2016年12月 (1)

展开

阅读排行

RPC框架与Dubbo完整使用 (14117)
Java为什么要设置环境变量、... (8431)
C语言、编程语言发展史 (7403)
Win10 + VMware-CentOS7... (6493)
Servlet的历史与规范 (5624)
Linux、开源软件发展史 (5008)
Linux中Apache(httpd)安装... (4951)
哈希表、Java中HashMap (3829)
Java中char和String 的深入理... (3505)
生产者消费者问题、Java实现 (3004)

评论排行

Java中char和String 的深入理... (5)
Linux、开源软件发展史 (5)
哈希表、Java中HashMap (5)
Servlet的历史与规范 (4)
CSND使用Markdown“写新... (4)
Apache2.4+Tomcat8负载均衡 (3)
MySQL实现类似Oracle的序... (2)
RPC框架与Dubbo完整使用 (2)
Java内存分配 (2)
SQL面试题总结、解答 (1)

推荐文章

* CSDN日报20170824——《你为什么跳槽？真正原因找到了吗？》
* Linux的任督二脉：进程调度和内存管理
* 秒杀系统的一点思考
* TCP网络通讯如何解决分包粘包问题
* 技术与技术人员的价值
* GitChat·人工智能 | 除了深度学习，机器翻译还需要啥？

最新评论

哈希表、Java中HashMap
lao5net ：写得很好

Linux、开源软件发展史
lao5net ：写的不错，继续努力。

CSND使用Markdown“写新文章”时出...

- 优点：服务端逻辑简单；
- 缺点：大多数请求是无效请求，在轮询很频繁的情况下对服务器的压力很大；

所以，除了一些简单练习项目外，这种方式不能被用于生产。

Comet

2和3属于：Comet (web技术)，是广大开发者想出来的比较可行的推送技术。

2. 长轮询（Long-Polling）

客户端向服务器发送AJAX请求，服务器接到请求后hold住连接，直到有新消息或超时（设置了多返回响应信息并关闭连接，客户端处理完响应信息后再向服务器发送新的请求。

- 优点：任意浏览器都可用；实时性好，无消息的情况下不会进行频繁的请求；
- 缺点：连接创建销毁操作还是比较频繁，服务器维持着连接比较消耗资源；

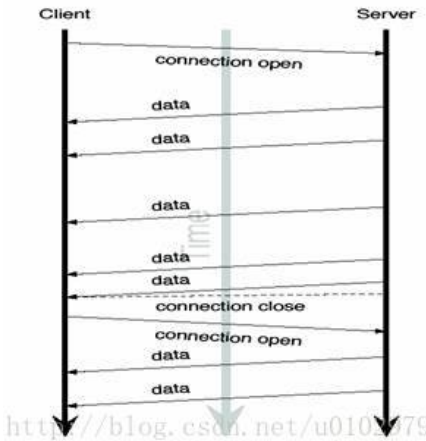
微信网页版使用的就是这种方式，据我观察：

- 微信把25秒作为超时时间；
- 用两个请求来完成长轮询，一个用于25秒超时获取是否有新消息，当有新消息时会用另一个AJAX请求来获取具体数据。

这种方式是可以被用于生产的，并且已经被实践检验有比较高的可用性。

3. 基于iframe的方式

iframe 是很早就存在的一种 HTML 标记， 通过在 HTML 页面里嵌入一个隐藏帧，然后将这个隐藏帧的 src 属性设为对一个长连接的请求，服务器端就能源源不断地往客户端输入数据。



iframe 服务器端并不返回直接显示在页面的数据，而是返回对客户端 **JavaScript** 函数的调用，如 `<script type="text/javascript">js_func("data from server")</script>`。服务器端将返回的数据作为客户端 JavaScript 函数的参数传递；客户端浏览器的 Javascript 引擎在收到服务器返回的 JavaScript 调用时就会去执行代码。

关闭

每次数据传送不会关闭连接，连接只会在通信出现错误时，或是连接重建时关闭（一些防火墙常被设置为丢弃过长的连接， 服务器端可以设置一个超时时间， 超时后通知客户端重新建立连接，并关闭原来的连接）。

- 优点：消息能够实时到达；
- 缺点：使用 iframe 请求一个长连接有一个很明显的不足之处：IE、Morzilla Firefox 下端的进度栏都会显示加载没有完成，而且 IE 上方的图标会不停的转动，表示加载正在进行；

Google公司在一些产品中使用了iframe流，如Google Talk。

局限性方式

4. 插件提供的Socket方式

利用Flash XMLSocket，**Java** Applet套接口，Activex包装的socket。

- 优点：原生socket的支持，和PC端和移动端的实现方式相似；
- 缺点：浏览器端需要装相应的插件；

---

5. WebSocket

2011年，WebSocket被IETF定为标准RFC 6455，WebSocket API也被W3C定为标准。

WebSocket 使得客户端和服务器之间的数据交换变得更加简单，允许服务端主动向客户端推送数据。在 WebSocket API 中，浏览器和服务器只需要完成一次握手，两者之间就直接可以创建持久性的连接，并进行双向数据传输。

WebSocket自然是极好的，更多细节我在下一节详细说明。

到这里，我们已经对WEB上的消息推送机制有了一个整体的了解。不过，仅仅只有了解对于我们来说显然还不够，由于我是**Java**程序员，接下来我将继续介绍WebSocket，并且用Java做服务端来做一个例子。

二、WebSocket

WebSocket 是独立的、创建在 TCP 上的协议。Websocket 通过 HTTP/1.1 协议的101状态码进行握手。为了创建Websocket连接，需要通过浏览器发出请求，之后服务器进行回应，这个过程通常称为“握手”（handshaking）。

1. ws请求

关闭

一个典型的WebSocket请求如下：

```
GET wss://xxx.xxx.com/push/ HTTP/1.1
Host: xxx.xxx.com:port
Connection:Upgrade
Upgrade:websocket
Sec-WebSocket-Extensions:permessage-deflate; client_max_window_bits
Sec-WebSocket-Key:rZGX8zZKTrdkhIJTCuW54Q==
Sec-WebSocket-Version:13

// Connection必须为：Upgrade，表示client希望升级连接；
// Upgrade必须为：websocket，表示client希望升级到Websocket协议；
// Sec-WebSocket-Key：是随机字符串，服务端会将其做一定运算，最后在Res
// Sec-WebSocket-Version：表示支持的Websocket版本。RFC6455要求使用的
```

响应如下：

```
HTTP/1.1 101 Switching Protocols
Upgrade:websocket
Connection:upgrade
Sec-WebSocket-Accept:QJsTRym36zHnArQ7FCmSdPhuK78=

// Connection:upgrade 升级被服务器同意
// Upgrade:websocket 指示客户端升级到websocket
// Sec-WebSocket-Accept：参考上面请求的Sec-WebSocket-Key的注释
```

上面只是比较重要的点，其实只知道这些暂时就够了，更详细的细节请参看：

RFC 6455 WebSocket

wikipedia WebSocket

2. WebSocket在Java中

JavaEE 7的JSR-356：**java** API for WebSocket，已经对WebSocket做了支持。不少Web容器，如Tomcat、Jetty等都支持WebSocket。Tomcat从7.0.27开始支持WebSocket，从7.0.47开始支持JSR-356。

但是如果使用**Java EE**的WebSocket API的话，还有很多自己需要封装的地方。**所以接下来我要说的并不是Java官方的API，而是目前正在接触的一种推送框架：Socket.IO以及其Server端的Java实现netty-socketio。这个框架不仅支持WebSocket，还支持Long-Polling模式。**

> 注意Socket.IO并不是一个标准的WebSocket的实现，只是说Socket.IO使用并很好的支持了WebSocket协议而已。

下面就说一下这两个框架。

3. SOCKET.IO

Socket.IO enables real-time bidirectional event-based communication. It consists in:

- a Node.js server (this repository)
- a Javascript client library for the browser (or a Node.js client)

SOCKET.IO - 官网地址

SOCKET.IO - github地址

由于其Server端是用**Node.js**实现的，又没有提供Java版本的Server，所以我找到了一个比较流行的第三方实现：netty-socketio。

4. netty-socketio

netty-socketio - github地址

This project is an open-source Java implementation of Socket.IO server. Based on Netty server framework.

netty-socketio是一个开源的Socket.IO Server的Java实现，基于Netty。

接下来我就使用netty-socketio来做一个demo。

三、netty-socketio实例

建议先大致读一下Socket.IO和netty-socketio的官方网站相关信息，以有个整体的概念，然后再做Demo，我就不把那些搬过来了。

Socket.IO中的一些重要概念。

1. `Server` ：代表一个服务端服务器；
2. `Namespace` ：一个 `Server` 中可以包含多个 `Namespace` 。见名知意， `Namespace` 代表一个个独立的空间。
3. `Socket` / `Client` ：基本上这两个词是一个概念。
    - 在 `JavaScript` 客户端叫 `Socket` ，在创建时必须确定加入哪个 `Namespace` ，使用 `Socket` 可以让你和服务器通信。注意这个和 `伯克利Socket` 是不同的，只是开发者借用了一样的名字、功能相似。
    - 在 `Java` 服务端用 `Client` 来表示连接上服务器的链接，它就代表了 `JavaScript` 连接时创建的那个 `Socket` 。
4. `room` ：在服务端，一个 `Namespace` 中你可以创建任意个房间，房间就是给 `Client` 进行分组，以进行组范围的通信。 `Client` 可以选择加入某个房间，也可以不加入。

代码实例：两个Namespace，广播通讯。

1. Java服务端

```
public static void main(String[] args) throws InterruptedException {

Configuration config = new Configuration();
config.setHostname("localhost");
config.setPort(9092);

// 可重用地址，防止处于重启时处于TIME_WAIT的TCP影响服务启动
final SocketConfig socketConfig = new SocketConfig();
socketConfig.setReuseAddress(true);
```

关闭

```
config.setSocketConfig(socketConfig);

final SocketIOServer server = new SocketIOServer(config);
final SocketIONamespace chat1namespace = server.addNamespace("/chat1");
chat1namespace.addEventListener("message", ChatObject.class, new DataListener<Ch
    @Override
    public void onData(SocketIOClient client, ChatObject data, AckRequest ackReq
        // broadcast messages to all clients
        chat1namespace.getBroadcastOperations().sendEvent("message", data);
    }
});

final SocketIONamespace chat2namespace = server.addNamespace("/chat2");
chat2namespace.addEventListener("message", ChatObject.class, ne                Ch
    @Override
    public void onData(SocketIOClient client, ChatObject data,                 eq
        // broadcast messages to all clients
        chat2namespace.getBroadcastOperations().sendEvent("mess
    }
});

server.start();

Thread.sleep(Integer.MAX_VALUE);

server.stop();
}
```

2. JS客户端

引用到的JS文件：

js文件github下载页面

时间格式化JS

```
<!DOCTYPE html>
<html>
<head>
    <title>Demo Chat</title>
    <link href="bootstrap.css" rel="stylesheet">
<style>
body {
    padding: 20px;
}
.console {
    height: 400px;
    overflow: auto;
}
.username-msg {
    color: orange;
}
.connect-msg {
    color: green;
}
.disconnect-msg {
    color: red;
}
.send-msg {
    color: #888
}
</style>

<script src="js/socket.io/socket.io.js"></script>
<script src="js/moment.min.js"></script>
<script src="http://code.jquery.com/jquery-1.10.1.min.js"></script>

<script>
```

关闭

```
33      var userName1 = 'user1_' + Math.floor((Math.random() * 1000) + 1);
34      var userName2 = 'user2_' + Math.floor((Math.random() * 1000) + 1);
35
36      var chat1Socket = io.connect('http://localhost:9092/chat1');
37      var chat2Socket = io.connect('http://localhost:9092/chat2');
38
39      function connectHandler(parentId) {
40          return function() {
41              output('<span class="connect-msg">Client has connected to the server
42          }
43      }
44
45      function messageHandler(parentId) {
46          return function(data) {
47              output('<span class="username-msg">' + data.userNam
48                      + data.message, parentId);
49          }
50      }
51
52      function disconnectHandler(parentId) {
53          return function() {
54              output('<span class="disconnect-msg">The client has ¨          ¨</s
55          }
56      }
57
58      function sendMessageHandler(parentId, userName, chatSocket) {
59          var message = $(parentId + ' .msg').val();
60          $(parentId + ' .msg').val('');
61
62          var jsonObject = {'@class': 'com.ddupa.service.push.model.ChatObject',
63                  userName: userName,
64                  message: message};
65          chatSocket.json.send(jsonObject);
66      }
67
68      chat1Socket.on('connect', connectHandler('#chat1'));
69      chat2Socket.on('connect', connectHandler('#chat2'));
70
71      chat1Socket.on('message', messageHandler('#chat1'));
72      chat2Socket.on('message', messageHandler('#chat2'));
73
74      chat1Socket.on('disconnect', disconnectHandler('#chat1'));
75      chat2Socket.on('disconnect', disconnectHandler('#chat2'));
76
77      function sendDisconnect1() {
78          chat1Socket.disconnect();
79      }
80
81      function sendDisconnect2() {
82          chat2Socket.disconnect();
83      }
84
85      function sendMessage1() {
86          sendMessageHandler('#chat1', userName1, chat1Socket);
87      }
88
89      function sendMessage2() {
90          sendMessageHandler('#chat2', userName2, chat2Socket);
91      }
92
93      function output(message, parentId) {
94          var currentTime = "<span class='time'>"
95                  + moment().format('HH:mm:ss.SSS') + "</span>";
96          var element = $("<div>" + currentTime + " " + message + "</div>");
97          $(parentId + ' .console').prepend(element);
98      }
99
100     $(document).keydown(function(e) {
101         if (e.keyCode == 13) {
102             $('#send').click();
103         }
```

关闭

```
    });
</script>
</head>
<body>
    <h1>Namespaces demo chat</h1>
    <br />
    <div id="chat1" style="width: 49%; float: left;">
        <h4>chat1</h4>
        <div class="console well"></div>

        <form class="well form-inline" onsubmit="return false;">
            <input class="msg input-xlarge" type="text"
                placeholder="Type something..." />
            <button type="button" onClick="sendMessage1()" clas                d"
            <button type="button" onClick="sendDisconnect1()" c                on
        </form>
    </div>

    <div id="chat2" style="width: 49%; float: right;">
        <h4>chat2</h4>
        <div class="console well"></div>

        <form class="well form-inline" onsubmit="return false;">
            <input class="msg input-xlarge" type="text"
                placeholder="Type something..." />
            <button type="button" onClick="sendMessage2()" class="btn" id="send"
            <button type="button" onClick="sendDisconnect2()" class="btn">Discon
        </form>
    </div>
</body>

</html>
```

到这里，我们学习了一个能用于生产的推送框架的基本使用。不过，以上只是一个简单例子，仅做引路入门，更多参考可以直接去官方网站找到，我再写就是赘述了：

- Socket.IO服务端API点这里
- Socket.IO JS客户端API点这里
- netty-socketio服务端API点这里
- netty-socketio Demo点这里

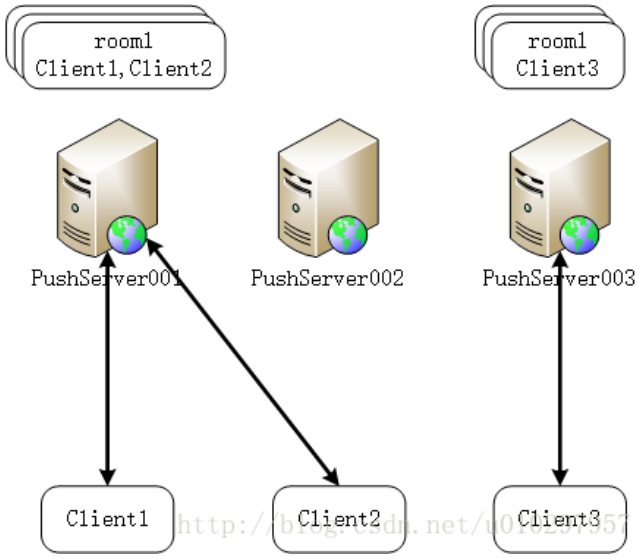
例外的一点是，由于分布式netty-socketio的部署方式文档中描述的不太清晰，且这部分实际中比较重要，我会在下面再继续描述下。

四、分布式服务器实例

1. 分布式环境下的问题

在分布式部署环境下假设有3台服务器分别为： PushServer001 、 PushServer002 和 PushServer003 。有3个 Client 连接上了服务器且他们都在一个命名空间下的同一个 room 中（叫 room1 ）。连接关系如下：

- Client1 <————> PushServer001
- Client2 <————> PushServer001
- Client3 <————> PushServer003



此时 Client1 发送了一条消息， PushServer集群 收到消息后显然需要将其推到 Client2 和 Client3 上。

- Client2 好说：它和 Client1 连接的是同一个 PushServer001 ， PushServer001 通过 Client1 可以获取到 room ，继而通过 room 获取到其下的 所有Clients （ 其中必有 Client2 ），然后推送即可。
- Client3 **怎么办呢？它连接的是** PushServer003 **，而** 003 **并没有收到** Client1 **的推送事件。**

2. 解决方案

其实解决方案也很简单，就是用**发布/订阅** 模式。

1. 首先需要引入一个第三方的发布/订阅系统，比如这里使用Redis-PUB/SUB。*（如果**Redis**是主从复制的，注意PUB只能由Master做，SUB则Master和Slaves都行）*
2. 其次，每当服务器需要发送消息时：
    - 先将消息发送给 本Server 保存的 某room 中的 所有Client ；
    - 接着再立即发布一个通知，例如叫 PubSubStore.DISPATCH ，并将消息内容放入其中。

```
// 本服务器推送
try {
    Iterable<SocketIOClient> clients = pushNamespace.getRoomClients(room);
    for (SocketIOClient socketIOClient : clients) {
        socketIOClient.send(packet);
    }
} catch (Exception e) {
    logger.error("当前服务直接推送失败", e);
}

// 分发消息(当前服务不会向client推送自己分发出去的消息)
try {
    pubSubStore.publish(PubSubStore.DISPATCH, new DispatchMessage(userId, packet
} catch (Exception e) {

```

```
16      logger.error("分发消息失败", e);
    }
```

3. 最后，每台服务器启动时都订阅通知 PubSubStore.DISPATCH 。每当当前服务器收到此类订阅通知时，就将其中的消息分发到同一个房间名的 所有Client 去。

   在 com.corundumstudio.socketio.store.pubsub.BaseStoreFactory.init(*) 时：

```
pubSubStore().subscribe(PubSubStore.DISPATCH, new PubSubListener<D      tchMessage
    @Override
    public void onMessage(DispatchMessage msg) {
        String room = msg.getRoom();
        namespacesHub.get(msg.getNamespace()).dispatch(room, ms
    }
}, DispatchMessage.class);
```

   如此便能解决此问题。附上netty-socket.io相关话题Wiki：How-To:-create-netty-socketio-servers。

其它一些事

1. HTTP持久连接

所谓HTTP持久连接即是：HTTP persistent connection，意即TCP连接重用技术。HTTP 1.0 的连接本来是“短连接”：建立一次TCP做完请求-响应即关闭，这样频繁的创建、关闭TCP连接显然是很低效比较浪费资源。

所以HTTP协议后来就做了升级，允许使用一个请求和响应头 Connection:keep-alive ，来祈使服务器能够保持连接不中断。如此，一个TCP连接就能在你对同一个网站进行访问的时候被多次复用，请求网页HTML本身、网页中的JS、CSS和图片等都用这一个连接。

不过，到了HTTP 1.1 以上连接默认就是持久化的了。

值得注意的是HTTP服务器一般都有超时机制，服务器不可能容忍你一直不释放连接的。例如：Apache httpd 1.3/2.0是15秒、2.2是5秒。

持久连接做的是**连接复用**的工作，并不是解决全双工通讯、推送的。

- 上一篇 Redis 主从、哨兵Sentinel、Jedis
- 下一篇 多媒体-声音

查看评论

accelerator 1楼 2017-05-02

赞

发表评论

用户名： qq_36596145

评论内容：

提交

* 以上用户言论只代表其个人观点，不代表CSDN网站的观点或立场

关闭