WebSocket相关笔记

目 录

1. websocket
   1. Websocket概述

Browser和WebServer间的实时数据传输是一个很重要的需求，但最早只能通过AJAX轮询方式实现。在WebSocket标准没有推出之前，AJAX轮询是一种可行的方案。

AJAX轮询原理是设置定时器，定时通过AJAX同步服务端数据。这种方式存在延时且对服务端造成很大负载。直至2011年，IETF才标准化WebSocket - 一种基于TCP套接字进行收发数据的协议。

Socket.io将数据传输部分独立出来形成engine.io，engine.io对WebSocket和AJAX轮询进行了封装，形成了一套API，屏蔽了细节差异和兼容性问题，实现了跨浏览器/跨设备进行双向数据通信。

* + 1. WebSocket 协议

WebSocket是HTML5新增的一种通信协议，其特点是服务端可以主动向客户端推送信息，客户端也可以主动向服务端发送信息，是真正的双向平等对话，属于服务器推送技术的一种。

在WebSocket API中，浏览器和服务器只需要做一个握手的动作，然后浏览器和服务端之间就形成了一条快速通道，两者之间就直接可以数据相互传送，带来的好处是

相互沟通的Header很小，大概只有2Bytes。

服务器不再被动的接收到浏览器的请求之后才返回数据，而是在有新数据时就主动推送给浏览器。

为了建立一个WebSocket连接，浏览器首先要向服务器发起一个HTTP请求，这个请求和通常的HTTP请求不同，包含了一些附加头信息，其中附加头信息Upgrade: WebSocket表明这是一个申请协议升级的HTTP请求。服务端解析这些头信息，然后产生应答信息返回给客户端，客户端和服务端的WebSocket连接就建立起来了。双方就可以通过这个连接通道自由的传递信息，并且这个连接会持续直到客户端或者服务端的某一方主动关闭连接。

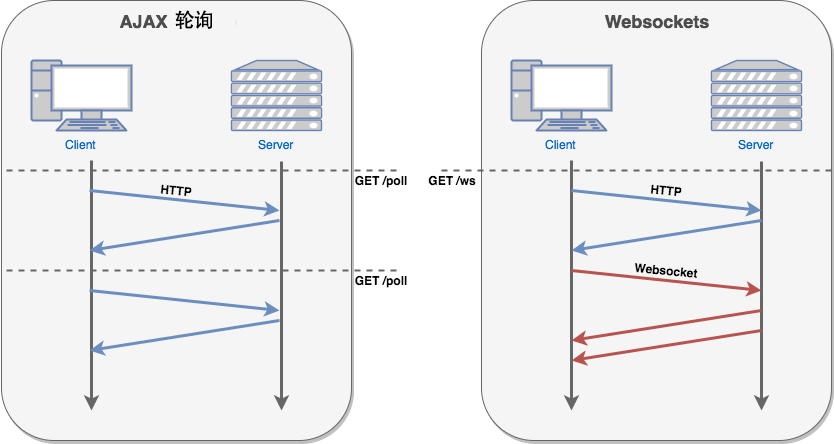
* + 1. 为什么要使用WebSocket呢？

Browser已经支持HTTP协议，为什么还要开发一种新的WebSocket协议呢？

我们知道HTTP协议是一种单向的网络协议，在建立连接后，仅允许Browser/UserAgent向WebServer发出请求资源后，WebServer才能返回对应的数据，而WebServer不能主动的推送数据给Browser/UserAgent。

最初这么设计HTTP协议的原因是，假设WebServer能主动的推送数据给Browser/UserAgent，那么Browser/UserAgent就太容易受到攻击了，一些广告商也会主动把广告在不经意间强行的传输给客户端，这不能不说是一个灾难。那么单向的HTTP协议给Web应用开发带哪些问题呢？

现在假设我们要开发一个基于Web的应用去获取当前WebServer的实时数据。例如股票实时行情、火车票剩余票数等。这就需要Browser/UserAgent与WebServer之间反复进行HTTP通信，Browser/UserAgent不断的发送请求去获取当前的实时数据。



* + 1. 常见的方式
       1. Polling

Polling轮询是通过Browser/UserAgent定时向WebServer发送HTTP请求，WebServer收到请求后把最新的数据发回给Browser/UserAgent，Browser/UserAgent得到数据后将其显示，然后再定期重复此过程。

虽然这样可以满足需求，但仍存在问题，例如某段时间内WebServer没有更新的数据，但Browser/UserAgent仍然会定时发送请求过来询问，WebServer可以把以前的老数据再传送过去，Browser/UserAgent把这些没有变化的数据再显示出来。这样既浪费网络带宽，有浪费CPU利用率。

如果说把Browser/UserAgent发送请求的周期调大一些，就可以缓解这个问题，但如果WebServer的数据更新很快时，这样又不能保证Web应用获取数据的实时性。

* + - 1. LongPolling

LongPolling是对Polling的一种改进。

Browser/UserAgent发送HTTP请求到WebServer，此时WebServer可以做2件事情：

1. 如果WebServer有新的数据需要传送，就立即把数据发回给Browser/UserAgent，Browser/UserAgent收到数据后，立即再发送HTTP请求给WebServer。
2. 如果WebServer没有新数据需要传送，这里与Polling的方式不同的是，WebServer不是立即发送回应给Browser/UserAgent，而是将这个请求保持住，等待有新的数据来到，再去响应这个请求。当然，如果WebServer的数据长期没有更新，一段时间后，这个HTTP请求就会超时，Browser/UserAgent收到超时信息后，在立即发送一个新的HTTP请求给服务器，然后依次循环这个过程。

LongPolling的方式虽然在某种程度上减少了网络带宽和CPU利用率等问题，但仍存在缺陷。

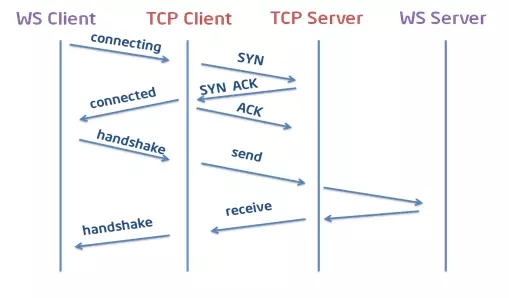
例如WebServer的数据更新速度较快，WebServer在传送一个数据包给Browser/UserAgent后必须等待Browser的下一个HTTP请求到来，才能传递第二个更新的数据包给Browser。这样的话，Browser显示实时数据最快的时间为2 xRTT（往返时间）。另外在网络拥堵的情况下，这个应该是不能让用户接受的。另外，由于HTTP数据包的头部数据量很大（通常有400多个字节），但真正被服务器需要的数据却很少（有时只有10个字节左右），这样的数据包在网络上周期性传输，难免对网络带宽是一种浪费。

综上所述，要是在Browser有一种新的网路一些，能支持客户端和服务端的双向通信，而且协议的头部又不那么庞大就very nice了。WebSocket正是肩负这样的使命登上了Web的舞台。

* + 1. WebSocket 原理

WebSocket是一种双向通信协议，它建立在TCP之上，同HTTP一样通过TCP来传输数据，但与HTTP最大不同的是：

1. WebSocket是一种双向通信协议，在建立连接后，WebSocket服务器和Browser/UserAgent都能主动的向对象发送或接收数据，就像Socket一样，不同的是WebSocket是一种建立在Web基础上的简单模拟Socket的协议。
2. WebSocket需要通过握手连接，类似TCP也需要客户端和服务端进行握手连接，连接成功后才能相互通信。



建立握手的时序图

简单说明下WebSocket握手的过程

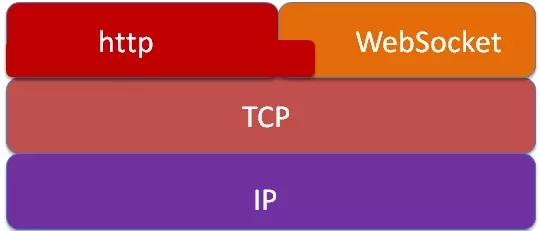
当Web应用端调用new WebSocket(url)接口时，Browser就开始了与地址为URL的WebServer建立握手连接的过程。

1. Browser与WebSocket服务器通过TCP三次握手建立连接，如果这个建立连接失败，那么后面的过程就不会执行，Web应用将收到错误消息通知。
2. 在TCP建立连接成功后，Browser/UserAgent通过HTTP协议传送WebSocket支持的版本号、协议的字版本号、原始地址、主机地址等一系列字段给服务端。
3. WebSocket服务器收到Browser/UserAgent发送来的握手请求后，如果数据包数据和格式正确，客户端和服务端的协议版本匹配等，就接受本次握手连接，并给出对应的数据回复，同样回复的数据包也是采用HTTP协议传输。
4. Browser收到服务器回复的数据包后，如果数据包内容、格式都没有问题的话，就表示本次连接成功，触发onopen消息，此时Web开发者就可以在此时通过send接口向服务器发送数据。否则，握手连接失败，Web应用会收到onerror消息，并且能知道连接失败的原因。
   * 1. WebSocket与TCP、HTTP的关系

WebSocket与HTTP协议一样都是基于TCP的，所以它们都是可靠的协议，Web开发者调用的WebSocket的send函数在Browser的实现中最终都是通过TCP的系统接口进行传输的。

WebSocket和HTTP协议样都属于应用层协议，那么它们之间有没有什么关系呢？

答案是肯定的，WebSocket在建立握手连接时，数据是通过HTTP协议传输的。但在建立连接之后，真正的数据传输阶段是不需要HTTP参与的。



WebSocket与TCP和HTTP的关系

* + 1. WebSocket Server

如果要搭建一个WebServer，我们会有很多选择，市场上也有很多成熟的产品供我们是使用。例如开源的Apache，安装配置后即可工作。但如果想要搭建一个WebSocket服务器就没有那么轻松，因为WebSocket是一种新的通信协议，目前还是草案，没有成为标准，市场上也没有成熟的WebSocket服务器或Library实现WebSocket协议，我们必须自己手动编码去解析和组装WebSocket的数据包。要完成一个WebSocket服务器，估计所有的人都想放弃，不过市场上有几款比较好的开源Library可供使用。例如PyWebSocket、WebSocket-Node、LibWebSockets等，这些Library已经实现了WebSocket数据包的封装和解析，我们可以调用这些接口，这在很大程度上减少了我们的工作量。

* 1. API

用API创建 WebSocket 对象

* + 1. 创建Websocket实例

var Socket = new WebSocket(url, [protocol] );

第一个参数 url，指定连接的 URL。

第二个参数 protocol 是可选的，指定了可接受的子协议。

* + 1. Websocket属性

|  |  |
| --- | --- |
| **属性** | **描述** |
| Socket.readyState | 只读属性 readyState 表示连接状态，可以是以下值：  0 - 表示连接尚未建立。  1 - 表示连接已建立，可以进行通信。  2 - 表示连接正在进行关闭。  3 - 表示连接已经关闭或者连接不能打开。 |
| Socket.bufferedAmount | 只读属性 bufferedAmount 已被 send() 放入正在队列中等待传输，但是还没有发出的 UTF-8 文本字节数。 |

* + 1. Websocket事件

|  |  |  |
| --- | --- | --- |
| **事件** | **事件处理程序** | **描述** |
| open | Socket.onopen | 连接建立时触发 |
| message | Socket.onmessage | 客户端接收服务端数据时触发 |
| error | Socket.onerror | 通信发生错误时触发 |
| close | Socket.onclose | 连接关闭时触发 |

* + 1. Websocket方法

|  |  |
| --- | --- |
| **方法** | **描述** |
| Socket.send() | 使用连接发送数据 |
| Socket.close() | 关闭连接 |

1. Socket.io

**使用socket.io模块**

后端使用socket.io

前端使用socket.io-client

* 1. 推荐学习网址

<https://www.w3cschool.cn/socket/socket-1olq2egc.html>

https://socket.io/docs/

https://socket.io/docs/client-api/

https://socket.io/docs/server-api/

* 1. 概述

socket.io不是由websocket实现的。

尽管Socket.io确实在可能的情况下使用WebSocket作为传输，但它会为每个数据包添加一些元数据：数据包类型，命名空间和需要消息确认时的确认ID。这就是为什么WebSocket客户端无法成功连接到Socket.IO服务器，而Socket.IO客户端也无法连接到WebSocket服务器。

* + 1. 发送和接收事件

Socket.IO允许您发出和接收自定义事件。此外connect，message和disconnect，你可以发出自定义事件：

//注意，io（<port>）将为你创建一个http服务器

var io = require（'socket.io'）（80）;

io.on（'connection'，function（socket） {

io.emit（'this'，{ will：'be everyone by everyone' }）;

socket.on（'悄悄话'，函数（从，MSG） { 控制台的.log（'我接收到的由一个私人消息'，从，'说'，MSG）; }）;

socket.on（'disconnect'，function（） {

io.emit（'user disconnected'）;

}）;

}）;

* + 1. 限制自定义命名空间

如果可以控制对特定应用发送的所有消息和事件，则可使用默认/命名空间。如果您想利用第三方代码，或给他人分享代码，socket.io提供了一种socket命名空间的方法。

它具有只是用一个可进行复数个用户(multiplexing)连接单个用户的好处。而不是使用两个WebSocket连接的socket.io。

server.js

var io = require('socket.io')(80);

var chat = io

.of('/chat')

.on('connection', function (socket) {

socket.emit('a message', {

that: 'only'

, '/chat': 'will get'

});

chat.emit('a message', {

everyone: 'in'

, '/chat': 'will get'

});

});

var news = io

.of('/news')

.on('connection', function (socket) {

socket.emit('item', { news: 'item' });

});

index.html

<script>

var chat = io.connect('http://localhost/chat')

, news = io.connect('http://localhost/news');

chat.on('connect', function () {

chat.emit('hi!');

});

news.on('news', function () {

news.emit('woot');

});

</script>

* + 1. 发送和获取数据（确认）

有时，您可能希望在客户端确认消息接收时收到回调。

为此，只需将函数作为.send或的最后一个参数传递.emit。更重要的是，当您使用时.emit，确认由您完成，这意味着您还可以传递数据：

app.js

var io = require（'socket.io'）（80）;

io.on（'connection'，function（socket） {

socket.on（'ferret'，function（name，word，fn） {

fn（name + '表示' + word）;

}）;

}）;

index.html

< script >

var socket = io(); //提示：没有args的io（）会自动发现

socket.on('connect'，function() { //提示：你可以避免收听`connect`并直接听事件！

socket.emit('ferret'，'tobi'，'woot'，function(data) { // args被发送以确认功能

console.log(data); //数据将是'tobi说woot' }) ;

});

</ script >

* + 1. 广播消息

要广播，只需添加一个broadcast标记emit和send方法调用。广播意味着向除了启动它的socket之外的所有人发送消息。

app.js

var io = require（'socket.io'）（80）;

io.on（'connection'，function（socket） {

socket.broadcast.emit（'user connected'）;

}）;

* 1. 命名空间 Namespace
     1. 默认命名空间 Default Namespace

/ —— 默认地址

io.sockets —简写—> io

io.sockets.emit('hi', 'everyone');

io.emit('hi', 'everyone'); // 简写

每个命名空间都会发出一个connection事件，将每个Socket实例作为参数接收。

io.on('connection'，function(socket) {

socket.on('disconnect'，function() {})

});

* + 1. 自定义命名空间 Custom Namespaces

要设置自定义命名空间，可以of在服务器端调用该函数：

const nsp = io.of('/my-namespace');

nsp.on('connection', function(socket){

console.log('someone connected');

});

nsp.emit('hi', 'everyone!');

在客户端，告诉Socket.IO客户端连接到该命名空间：

const socket = io('/my-namespace');

重要说明：命名空间是Socket.IO协议的实现细节，与底层传输的实际URL无关，默认为/socket.io/…。

* 1. 房间 Room

每一个命名的Room内，可以任意加入和离开

* + 1. 加入房间 Joining and leaving

可以将任意socket加入任意通道中

io.on('connection', function(socket){

socket.join('some room');

});

当发送或广播内容相同时，可用 to 或 in 简写

io.to('some room').emit('some event');

离开通道的方式 leave 与加入通道的方式 join 相同

* + 1. 默认房间 Default room

每一个socket都有一个随机的、唯一的socket#id。

每一个socket将自动通过该id来判断是否加入Room。

io.on('connection', function(socket){

socket.on('say to someone', function(id, msg){

socket.broadcast.to(id).emit('my message', msg);

});

});

* + 1. 断开 Disconnection

断开连接时，所有通道将自动leave，不需要专门拆除(teardown)

* + 1. 注意

注意：以下事件是保留的，不应被应用程序用作事件名称：

error

connect

connection

disconnect

disconnecting

newListener

removeListener

ping

pong

* 1. 后端常用api

https://socket.io/docs/server-api/

* + 1. 创建服务
       1. 默认设置

新服务器（httpServer [，options]）

httpServer （http.Server）要绑定的服务器。

options （宾语）

const io = require('socket.io')();

//或

const Server = require('socket.io');

const io = new Server();

* + - 1. 手动配置

新服务器（端口[，选项]）

port （Number）要侦听的端口（http.Server将创建一个新端口）

options （宾语）

const io = require('socket.io')(3000, {

path: '/test',

serveClient: false,

// below are engine.IO options

pingInterval: 10000,

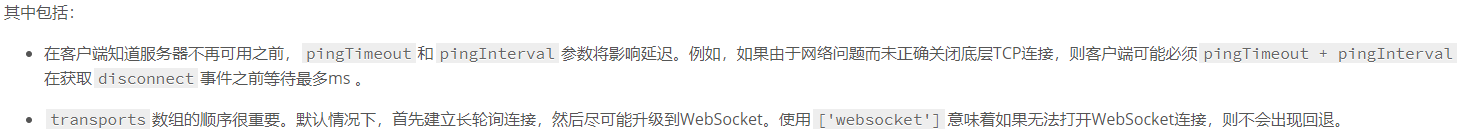
pingTimeout: 5000,

cookie: false

});







* + 1. 断开服务

server.close（[回调]）

callback （功能）

关闭socket.io服务器。该callback参数是可选的，在关闭所有连接将被调用。

const Server = require（'socket.io'）;

const PORT = 3030 ;

const server = require（'http'）。Server（）;

const io =服务器（PORT）;

io.close（）; //关闭当前服务器

server.listen（PORT）; // PORT可以免费使用

io =服务器（服务器）;

* + 1. emit — 发送

socket.emit（eventName [，... args] [，ack]）

（覆盖EventEmitter.emit）

eventName （String）

args

ack （Function）

返回 Socket

将一个事件发送到由字符串名称标识的套接字。可以包括任何其他参数。支持所有可序列化的数据结构，包括Buffer。

该ack参数是可选的，并且将被客户端的答案调用。

在发生特定情况时发送一个action命令

命令可以只有一个名字

io.emit('自命名')

命令可以除了名字还有数据

io.emit('自命名',a)

命令可以除了名字还有多个数据

io.emit('自命名',a，b，c，...，d)

命令可以除了名字还有回调函数

io.emit('自命名',data，function(a,b){ } )

* + 1. send — 发送

socket.send([…args][, ack])

args

ack (Function)

返回Socket

Sends a message event. See socket.emit(eventName[, …args][, ack]).

send仅能将信息传输给默认监听事件(message)

emit可将信息传输给自定义监听事件

* + 1. on — 接收(监听)

socket.on（eventName，callback）

（继承自EventEmitter）

eventName （String）

callback （Function）

返回 Socket

为给定的事件注册一个新的处理程序。

与上述一一对应

命令可以只有一个名字

socket.on('自命名'，function(){ } )

命令可以除了名字还有数据

socket.on('自命名', function(data){ } )

命令可以除了名字还有多个数据

socket.on('自命名', function(a,b,c,...,d){ } )

命令可以除了名字数据还有回调函数

socket.on('自命名', function(data，fn){ fn(a,b); })

* + 1. once — 仅监听一次

socket.once（eventName，listener）

只监听一次，会在回调函数执行完毕后，取消监听

* + 1. join — 加入房间

socket.join（room [，callback]）

room （String）

callback （Function）

Socket链接返回

将客户端添加到room，并且可选地启动带有err签名的回调（如果有）。

* + 1. leave — 离开房间

socket.leave（room [，callback]）

room （String）

callback （Function）

Socket链接返回

从中删除客户端room，并可选地启动带有err签名的回调（如果有）。

断开后房间自动关闭。

* + 1. of — 命名空间地址

socket.of(nsp)

nsp (String|RegExp|Function)

返回Namespace

初始化和检索这个命名空间在nsp命名列表中。

若该命名空间已经初始化，则返回命名空间地址。

const adminNamespace = io.of('/admin');

* + 1. to / in — 指定房间

socket.to（room）

room （String）

Socket链接返回

为随后的事件发射设置一个修饰符，事件只会广播到已加入给定的客户端room（套接字本身被排除）。

要发射到多个房间，可以to多次打电话。

注意：广播时不支持确认。

* + 1. 事件(Event)
       1. conn/disconnect — 连接/断开连接

socket.disconnect（close）

close （Boolean）是否关闭底层连接

返回 Socket

断开此客户端。如果值为close true，则关闭底层连接。否则，它只是断开命名空间。

socket.on('connect', function(data) {}) // 连接

socket.on('disconnect', function(data) {}) // 断开连接

connect与connection同义

disconnecting

reason （字符串）断开连接的原因（客户端或服务器端）

当客户端将被断开连接时被触发（但尚未离开它rooms）。

io.on('connection', (socket) => {

socket.on('disconnecting', (reason) => {

let rooms = Object.keys(socket.rooms);

// ...

});

});

* + - 1. error

error(Object)错误对象

发生错误时触发。

io.on('connection', (socket) => {

socket.on('error', (error) => {

// ...

});

});

* + 1. 命名空间（Room）

namespace.name

(String)

The namespace identifier property.

namespace.connected

(Object)

socket连接到此命名空间的对象的哈希值，索引为id

namespace.adapter

(Adapter)

该Adapter用于命名空间。在使用Adapter基于Redis的方法时很有用，因为它公开了管理集群中socket和room的方法。

注意：可以使用主要命名空间的Adapter访问io.of('/').adapter。

namespace.to(room)

room (String)

返回Namespace的连接

为后续事件设置修饰符，该事件将仅广播到已加入给定的客户端room。

要发送多个room，你可以to多次。

const io = require('socket.io')();

const adminNamespace = io.of('/admin');

adminNamespace.to('level1').emit('an event', { some: 'data' });

namespace.in(room)

同namespace.to(room)

namespace.emit(eventName[, …args])

eventName (String)

args

向所有连接的客户端发送事件。

const io = require（'socket.io'）（）;

io.emit（'发送给所有连接客户的事件'）; //主命名空间

或

const chat = io.of（'/ chat'）;

chat.emit（'发送给聊天命名空间中所有连接客户端的事件'）;

注意：从命名空间发出时不支持确认回调。

* 1. 前端常用api

https://socket.io/docs/client-api/

* + 1. 创建客户端 Manager

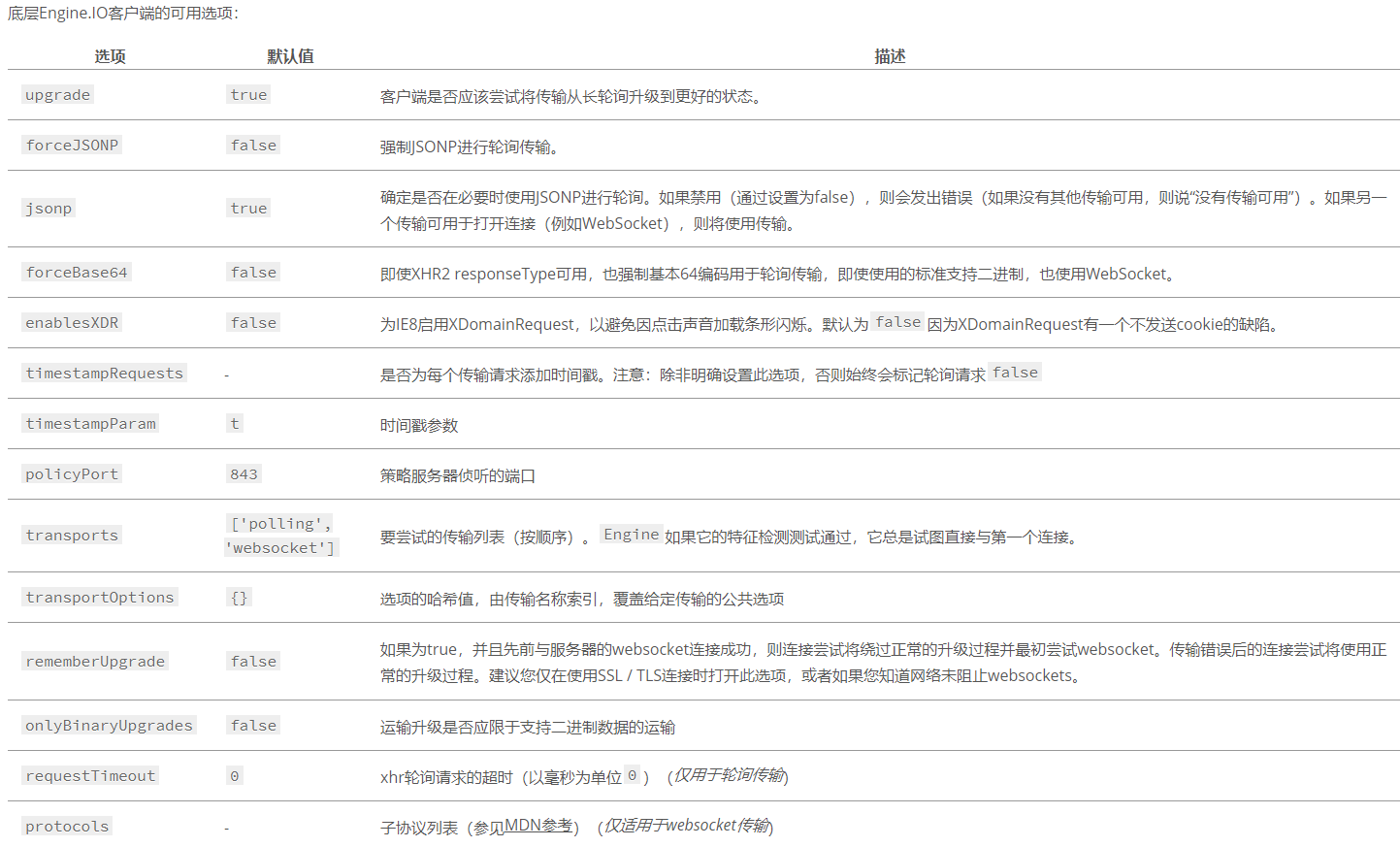
new Manager(url[, options])

url (String)

options (Object)

返回Manager







* + 1. 连接

manager.open（[回调]）

callback （功能）

返回 Manager

如果管理者与发起autoConnect到false，启动一个新的连接尝试。

该callback参数是可选的，一旦尝试失败将被调用/成功。

manager.connect（[回调]）

manager.open（[callback]）的同义词。

* + 1. 创建命名空间

manager.socket（nsp，options）

nsp （String）

options （Object）

返回 Socket

Socket为给定的命名空间创建一个新的。

* + 1. 事件(Event)
       1. connect

连接成功，包括成功重新连接。

socket.on('connect', () => {

// ...

});

// note: you should register event handlers outside of connect,

// so they are not registered again on reconnection

socket.on('myevent', () => {

// ...

});

* + - 1. connect\_error

error （对象）错误对象

因连接错误而被解雇。

* + - 1. connect\_timeout

在连接超时时触发。

* + 1. socket
       1. id

socket.id

(String)

socket会话的唯一标识符。connect事件触发后设置，并在reconnect事件发生后更新。

const socket = io('http://localhost');

console.log(socket.id); // undefined

socket.on('connect', () => {

console.log(socket.id); // 'G5p5...'

});

* + - 1. 是否连接

socket.connected

（布尔）

socket是否连接到服务器。

const socket = io('http://localhost');

socket.on('connect', () => {

console.log(socket.connected); // true

})

* + - 1. 是否断开连接

socket.disconnected

（布尔）

socket是否与服务器断开连接。

const socket = io('http://localhost');

socket.on('connect', () => {

console.log(socket.disconnected); // false

});

* + - 1. 连接open / connect

socket.open（）

返回 Socket

手动打开socket。

const socket = io({

autoConnect: false

});

// ...

socket.open();

它也可以用于手动重新连接：

socket.on('disconnect', () => {

socket.open();

});

socket.connect（）

socket.open（）的同义词。

* + - 1. 发送send / emit

ocket.send（[... args] [，ack]）

args

ack （function）

返回 Socket

发送message事件。请参阅socket.emit（eventName [，... args] [，ack]）。

socket.emit（eventName [，... args] [，ack]）

eventName （string）

args

ack （function）

返回 Socket

将事件发送到由string名称标识的socket。可以包括任何其他参数。支持所有可序列化的数据结构，包括Buffer。

socket.emit('hello', 'world');

socket.emit('with-binary', 1, '2', { 3: '4', 5: new Buffer(6) };

该ack参数是可选的，并会与服务器的响应被调用。

socket.emit('ferret', 'tobi', (data) => {

console.log(data); // data will be 'woot'

});

// server:

// io.on('connection', (socket) => {

// socket.on('ferret', (name, fn) => {

// fn('woot');

// });

// });

* + - 1. 监听事件on

socket.on(eventName, callback)

eventName (String)

callback (Function)

返回Socket

为给定事件注册新的处理程序。

socket.on('news', (data) => {

console.log(data);

});

// with multiple arguments

socket.on('news', (arg1, arg2, arg3, arg4) => {

// ...

});

// with callback

socket.on('news', (cb) => {

cb(0);

});

* + - 1. 断开连接close / disconnect

socket.close（）

返回 Socket

手动断开socket。

socket.disconnect（）

socket.close（）的同义词。