link: null
title: 珠峰架构师成长计划
description: Http是客户端/服务器模式中请求-响应所用的协议，在这种模式中，客户端(一般是web浏览器)向服务器提交HTTP请求，服务器响应请求的资源
keywords: null
author: null
date: null
publisher: 珠峰架构师成长计划
stats: paragraph=87 sentences=155, words=1287

1. HTTP的架构模式

Http是客户端/服务器模式中请求-响应所用的协议，在这种模式中，客户端(一般是web浏览器)向服务器提交HTTP请求，服务器响应请求的资源

1.1. HTTP的特点

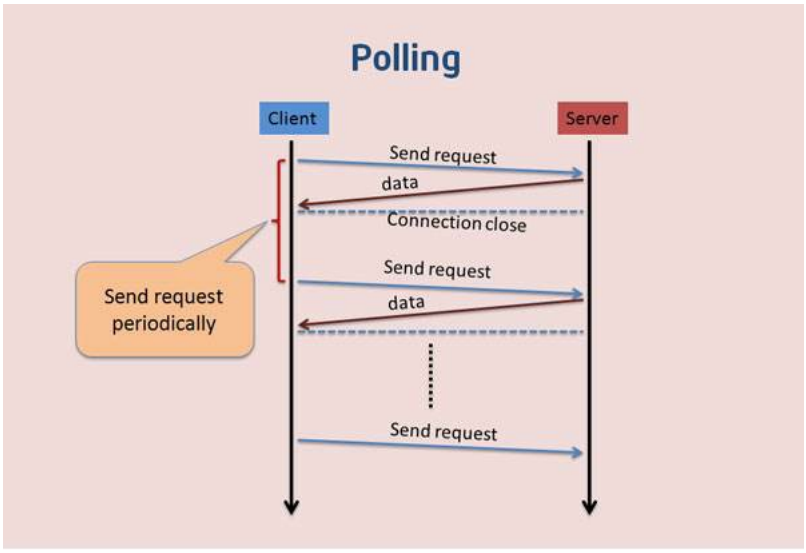
- HTTP是半双工协议，也就是说，在同一时刻数据只能单向流动，客户端向服务器发送请求(单向的)，然后服务器响应请求(单向的)。
- 服务器不能主动推送数据给浏览器。

2. 双向通信

Comet是一种用于web的推送技术，能使服务器能实时地将更新的信息传送到客户端，而无须客户端发出请求，目前有三种实现方式:轮询（polling） 长轮询（long-polling）和iframe流（streaming）。

2.1 轮询

- 轮询是客户端和服务器之间会一直进行连接，每隔一段时间就询问一次
- 这种方式连接数会很多，一个接受，一个发送。而且每次发送请求都会有Http的Header，会很耗流量，也会消耗CPU的利用率



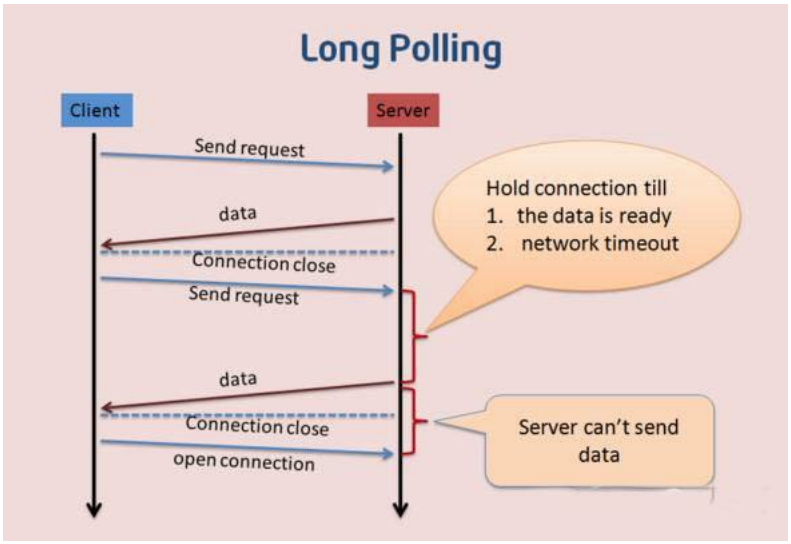
server.js

```
let express = require('express');
let app = express();
app.use(express.static(__dirname));
app.use(function(req,res,next){
    res.header('Access-Control-Allow-Origin', 'http://localhost:8000');
    res.end(new Date().toLocaleTimeString());
});
app.listen(8080);
```

```
        setInterval(function () {
            let xhr = new XMLHttpRequest();
            xhr.open('GET', 'http://localhost:8080', true);
            xhr.onreadystatechange = function () {
                if (xhr.readyState == 4 && xhr.status == 200) {
                    document.querySelector('#clock').innerHTML = xhr.responseText;
                }
            }
            xhr.send();
        }, 1000);
```

1.2 长轮询

- 长轮询是对轮询的改进版，客户端发送HTTP给服务器之后，看有没有新消息，如果没有新消息，就一直等待
- 当有新消息的时候，才会返回给客户端。在某种程度上减小了网络带宽和CPU利用率等问题。
- 由于http数据包的头部数据量往往很大（通常有400多个字节），但是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的传输，难免对网络带宽是一种浪费



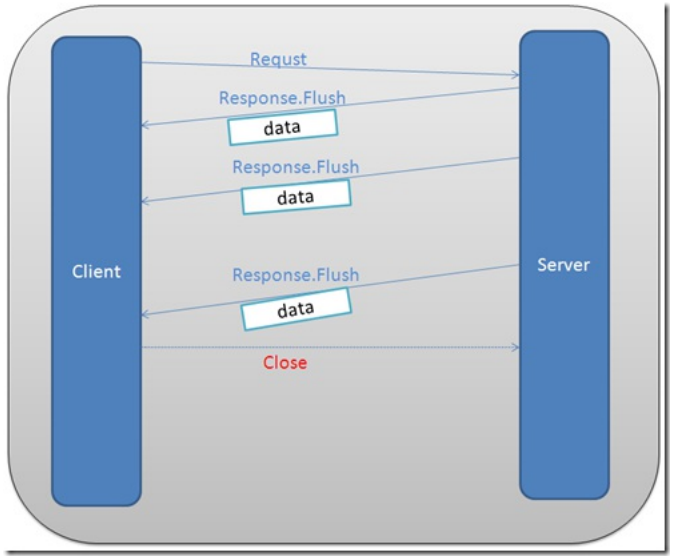
clock.html

```
    (function poll() {
            let xhr = new XMLHttpRequest();
            xhr.open('GET', 'http://localhost:8080', true);
            xhr.onreadystatechange = function () {
                if (xhr.readyState == 4 && xhr.status == 200) {
                    document.querySelector('#clock').innerHTML = xhr.responseText;
                    poll();
                }
            }
            xhr.send();
    })();
```

> long poll 需要有很高的并发能力

1.3 iframe流

- 通过在HTML页面里嵌入一个隐藏的iframe,然后将这个iframe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推送数据。



server.js

```
const express = require('express');
const app = express();
app.use(express.static(__dirname));
app.get('/clock', function (req, res) {
    setInterval(function () {
        res.write(`

                parent.document.getElementById('clock').innerHTML = "<span class="hljs-subst">${<span class="hljs-keyword">new</span> <span class="hljs-
built_in">Date</span>().toLocaleTimeString()}</span>";

        `);
    }, 1000);
});
app.listen(8080);
```

client.html

```
    <div id="clock">div>
    <iframe src="/clock" style=" display:none" />
```

1.4 EventSource流

- HTML5规范中提供了服务端事件EventSource，浏览器在实现了该规范的前提下创建一个EventSource连接后，便可收到服务端的发送的消息，这些消息需要遵循一定的格式，对于前端开发人员而言，只需在浏览器中侦听对应的事件皆可
- SSE的简单模型是：一个客户端去从服务器端订阅一条 流，之后服务端可以发送消息给客户端直到服务端或者客户端关闭该"流"，所以eventsource也叫作 "server-sent-event`
- EventSource流的实现方式对客户端开发人员而言非常简单，兼容性良好
- 对于服务端，它可以兼容老的浏览器，无需upgrade为其他协议，在简单的服务端推送的场景下可以满足需求

1.4.1 浏览器端

- 浏览器端，需要创建一个 EventSource对象，并且传入一个服务端的接口URI作为参
- 默认EventSource对象通过侦听 message事件获取服务端传来的消息
- open事件则在http连接建立后触发
- error事件会在通信错误（连接中断、服务端返回数据失败）的情况下触发
- 同时 EventSource规范允许服务端指定自定义事件，客户端侦听该事件即可

```
var eventSource = new EventSource('/eventSource');
eventSource.onmessage  = function(e){
    console.log(e.data);
}
eventSource.onerror  = function(err){
    console.log(err);
}
```

1.4.2 服务端

- 事件流的对应MIME格式为 text/event-stream，而且其基于HTTP长连接。针对HTTP1.1规范默认采用长连接，针对HTTP1.0的服务器需要特殊设置。
- event-source必须编码成 utf-8的格式，消息的每个字段使用"\n"来做分割，并且需要下面4个规范定义好的字段：
    - Event: 事件类型
    - Data: 发送的数据
    - ID: 每一条事件流的ID
    - Retry： 告知浏览器在所有的连接丢失之后重新开启新的连接等待的时间，在自动重新连接的过程中，之前收到的最后一个事件流ID会被发送到服务端

```
let  express = require('express');
let app = express();
app.use(express.static(__dirname));
let sendCount = 1;
app.get('/eventSource',function(req,res){
    res.header('Content-Type','text/event-stream',);
    setInterval(() => {
      res.write(`event:message\nid:${sendCount++}\ndata:${Date.now()}\n\n`);
    }, 1000)
});
app.listen(8888);
```

```
let  express = require('express');
let app = express();
app.use(express.static(__dirname));
const SseStream = require('ssestream');
let sendCount = 1;
app.get('/eventSource',function(req,res){
    const sseStream = new SseStream(req);
    sseStream.pipe(res);
    const pusher = setInterval(() => {
      sseStream.write({
        id: sendCount++,
        event: 'message',
        retry: 20000,
        data: {ts: new Date().toTimeString()}
      })
    }, 1000)

    res.on('close', () => {
      clearInterval(pusher);
      sseStream.unpipe(res);
    })
});
app.listen(8888);
```

2.websocket

- WebSockets API (https://developer.mozilla.org/en-US/docs/Web/API/WebSockets_API) 规范定义了一个 API 用以在网页浏览器和服务器建立一个 socket 连接。通俗地讲：在客户端和服务器保有一个持久的连接，两边可以在任意时间开始发送数据。
- HTML5开始提供的一种浏览器与服务器进行全双工通讯的网络技术
- 属于应用层协议，它基于TCP传输协议，并复用HTTP的握手通道。

2.1 websocket 优势

- 支持双向通信，实时性更强。
- 更好的二进制支持。
- 较少的控制开销。连接创建后，ws客户端、服务端进行数据交换时，协议控制的数据包头部较小。

2.2 websocket实战

2.2.1 服务端

```
let express = require('express');
const path = require('path');
let app = express();
let server = require('http').createServer(app);
app.get('/', function (req, res) {
    res.sendFile(path.resolve(__dirname, 'index.html'));
});
app.listen(3000);

let WebSocketServer = require('ws').Server;
let wsServer = new WebSocketServer({ port: 8888 });
wsServer.on('connection', function (socket) {
    console.log('连接成功');
    socket.on('message', function (message) {
        console.log('接收到客户端消息:' + message);
        socket.send('服务器回应:' + message);
    });
});
```

2.2.2 客户端

```
    <script>
        let ws = new WebSocket('ws://localhost:8888');
        ws.onopen = function () {
            console.log('客户端连接成功');
            ws.send('hello');
        }
        ws.onmessage = function (event) {
            console.log('收到服务器的响应 ' + event.data);
        }
    script>
```

2.3 如何建立连接

WebSocket复用了HTTP的握手通道。具体指的是，客户端通过HTTP请求与WebSocket服务端协商升级协议。协议升级完成后，后续的数据交换则遵照WebSocket的协议。

2.3.1 客户端：申请协议升级

首先，客户端发起协议升级请求。可以看到，采用的是标准的HTTP报文格式，且只支持GET方法。

```
GET ws:
Host: localhost:8888
Connection: Upgrade
Upgrade: websocket
Sec-WebSocket-Version: 13
Sec-WebSocket-Key: IHfMdf8a0aQXbwQOlpkGdA==
```

- Connection: Upgrade：表示要升级协议
- Upgrade: websocket：表示要升级到websocket协议
- Sec-WebSocket-Version: 13：表示websocket的版本
- Sec-WebSocket-Key：与后面服务端响应首部的Sec-WebSocket-Accept是配套的，提供基本的防护，比如恶意的连接，或者无意的连接。

2.3.2 服务端：响应协议升级

服务端返回内容如下，状态代码101表示协议切换。到此完成协议升级，后续的数据交互都按照新的协议来。

```
HTTP/1.1 101 Switching Protocols
Upgrade: websocket
Connection: Upgrade
Sec-WebSocket-Accept: aWAY+V/uyz5ILZEoWuWdxjnlb7E=
```

2.3.3 Sec-WebSocket-Accept的计算

Sec-WebSocket-Accept根据客户端请求首部的Sec-WebSocket-Key计算出来。 计算公式为：

- 将Sec-WebSocket-Key跟258EAFA5-E914-47DA-95CA-C5AB0DC85B11拼接。
- 通过SHA1计算出摘要，并转成base64字符串

```
const crypto = require('crypto');
const number = '258EAFA5-E914-47DA-95CA-C5AB0DC85B11';
const webSocketKey = 'IHfMdf8a0aQXbwQOlpkGdA==';
let websocketAccept = require('crypto').createHash('sha1').update(webSocketKey + number).digest('base64');
console.log(websocketAccept);
```

2.3.4 Sec-WebSocket-Key/Accept的作用

- 避免服务端收到非法的websocket连接
- 确保服务端理解websocket连接
- 用浏览器里发起ajax请求，设置header时，Sec-WebSocket-Key以及其他相关的header是被禁止的
- Sec-WebSocket-Key主要目的并不是确保数据的安全性，因为Sec-WebSocket-Key、Sec-WebSocket-Accept的转换计算公式是公开的，而且非常简单，最主要的作用是预防一些常见的意外情况（非故意的）

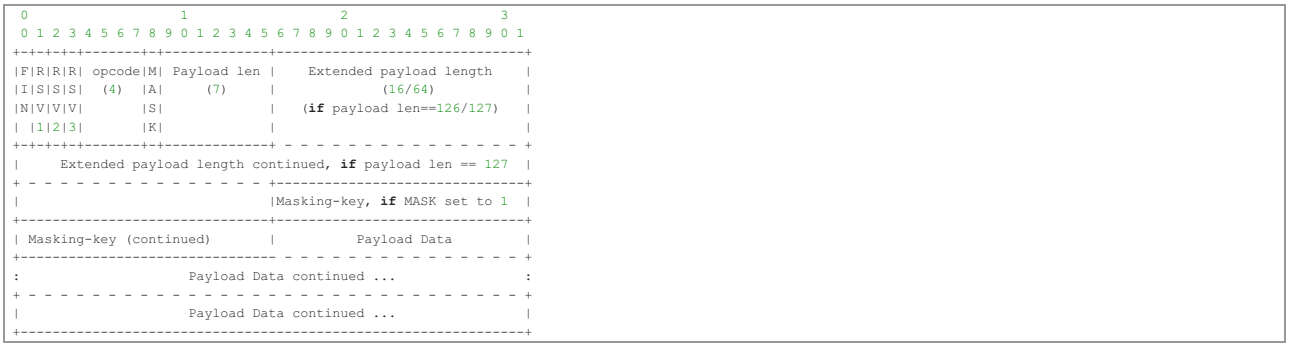
2.4 数据帧格式

WebSocket客户端、服务端通信的最小单位是帧 (https://tools.ietf.org/html/rfc6455#section-5.2)，由1个或多个帧组成一条完整的消息（message）。

- 发送端：将消息切割成多个帧，并发送给服务端
- 接收端：接收消息帧，并将关联的帧重新组装成完整的消息

2.4.1 数据帧格式

单位是比特。比如FIN、RSV1各占据1比特，opcode占据4比特



- FIN：1个比特 如果是1，表示这是消息（message）的最后一个分片（fragment），如果是0，表示不是是消息（message）的最后一个分片（fragment）
- RSV1, RSV2, RSV3：各占1个比特。一般情况下全为0。当客户端、服务端协商采用WebSocket扩展时，这三个标志位可以非0，且值的含义由扩展进行定义。如果出现非零的值，且并没有采用WebSocket扩展，连接出错。
- Opcode: 4个比特。操作代码，Opcode的值决定了应该如何解析后续的数据载荷（data payload）。如果操作代码是不认识的，那么接收端应该断开连接（fail the connection）
    - %x0：表示一个延续帧。当Opcode为0时，表示本次数据传输采用了数据分片，当前收到的数据帧为其中一个数据分片。
    - %x1：表示这是一个文本帧（frame）
    - %x2：表示这是一个二进制帧（frame）
    - %x3-7：保留的操作代码，用于后续定义的非控制帧。
    - %x8：表示连接断开。
    - %x9：表示这是一个ping操作。
    - %xA：表示这是一个pong操作。
    - %xB-F：保留的操作代码，用于后续定义的控制帧。
- Mask: 1个比特。表示是否要对数据载荷进行掩码操作
    - 从客户端向服务端发送数据时，需要对数据进行掩码操作；从服务端向客户端发送数据时，不需要对数据进行掩码操作.如果服务端接收到的数据没有进行过掩码操作，服务端需要断开连接。
    - 如果Mask是1，那么在Masking-key中会定义一个掩码键（masking key），并用这个掩码键来对数据载荷进行反掩码。所有客户端发送到服务端的数据帧，Mask都是1。
- Payload length：数据载荷的长度，单位是字节。为7位，或7+16位，或7+64位。
    - Payload length=x为0~125：数据的长度为x字节。
    - Payload length=x为126：后续2个字节代表一个16位的无符号整数，该无符号整数的值为数据的长度。
    - Payload length=x为127：后续8个字节代表一个64位的无符号整数（最高位为0），该无符号整数的值为数据的长度。
    - 如果payload length占用了多个字节的话，payload length的二进制表达采用网络序（big endian，重要的位在前)
- Masking-key：0或4字节(32位) 所有从客户端传送到服务端的数据帧，数据载荷都进行了掩码操作，Mask为1，且携带了4字节的Masking-key。如果Mask为0，则没有Masking-key。载荷数据的长度，不包括mask key的长度
- Payload data：(x+y) 字节
    - 载荷数据：包括了扩展数据、应用数据。其中，扩展数据x字节，应用数据y字节。
    - 扩展数据：如果没有协商使用扩展的话，扩展数据数据为0字节。所有的扩展都必须声明扩展数据的长度，或者可以如何计算出扩展数据的长度。此外，扩展如何使用必须在握手阶段就协商好。如果扩展数据存在，那么载荷数据长度必须将扩展数据的长度包含在内。
    - 应用数据：任意的应用数据，在扩展数据之后（如果存在扩展数据），占据了数据帧剩余的位置。载荷数据长度 减去 扩展数据长度，就得到应用数据的长度。

2.4.2 掩码算法

掩码键（Masking-key）是由客户端挑选出来的32位的随机数。掩码操作不会影响数据载荷的长度。掩码、反掩码操作都采用如下算法：

- 对索引i模以4得到j,因为掩码一共就是四个字节
- 对原来的索引进行异或对应的掩码字节
- 异或就是两个数的二进制形式，按位对比，相同取0，不同取1

```
function unmask(buffer, mask) {
     const length = buffer.length;
     for (let i = 0; i < length; i++) {
         buffer[i] ^= mask[i & 3];
     }
 }
```

2.4.3 服务器实战

```
const net = require('net');
const crypto = require('crypto');
const CODE = '258EAFA5-E914-47DA-95CA-C5AB0DC85B11';
let server = net.createServer(function (socket) {
    socket.once('data', function (data) {
        data = data.toString();
        if (data.match(/Upgrade: websocket/)) {
            let rows = data.split('\r\n');
            rows = rows.slice(1, -2);
            const headers = {};
            rows.forEach(row => {
                let [key, value] = row.split(': ');
                headers[key] = value;
            });
            if (headers['Sec-WebSocket-Version'] == 13) {
                let wsKey = headers['Sec-WebSocket-Key'];
                let acceptKey = crypto.createHash('sha1').update(wsKey + CODE).digest('base64');
                let response = [
                    'HTTP/1.1 101 Switching Protocols',
                    'Upgrade: websocket',
                    `Sec-WebSocket-Accept: ${acceptKey}`,
                    'Connection: Upgrade',
                    '\r\n'
                ].join('\r\n');
                socket.write(response);
                socket.on('data', function (buffers) {
                    let _fin = (buffers[0] & 0b10000000) === 0b10000000;
                    let _opcode = buffers[0] & 0b00001111;
                    let _masked = buffers[1] & 0b100000000 === 0b100000000;
                    let _payloadLength = buffers[1] & 0b01111111;
                    let _mask = buffers.slice(2, 6);
                    let payload = buffers.slice(6);

                    unmask(payload, _mask);

                    let response = Buffer.alloc(2 + payload.length);
                    response[0] = _opcode | 0b10000000;
                    response[1] = payload.length;
                    payload.copy(response, 2);
                    socket.write(response);
                });
            }

        }
    });
    function unmask(buffer, mask) {
        const length = buffer.length;
        for (let i = 0; i < length; i++) {
            buffer[i] ^= mask[i & 3];
        }
    }
    socket.on('end', function () {
        console.log('end');
    });
    socket.on('close', function () {
        console.log('close');
    });
    socket.on('error', function (error) {
        console.log(error);
    });
});
server.listen(9999);
```

参考

- eventsource (https://blog.5udou.cn/blog/JSShi-Shi-Tong-Xin-San-Ba-Fu-Xi-Lie-Zhi-San-eventsource55)
- 服务端事件EventSource (https://www.cnblogs.com/accordion/p/7764460.html)