HTTP VS WebSocket：基于实时股票价格更新demo的协议对比

HTTP协议长期以来一直是客户端和服务器之间通信的主流协议。然而，随着实时Web应用的需求，出现了为双向、低延迟的通信而专门设计的WebSocket协议。本文将通过一个实时股票价格更新的具体示例，深入比较WebSocket和HTTP这两种协议在实现、性能、适用场景等方面的差异。

假设我们正在开发一个股票交易应用。需要为用户实时提供股票价格，接下来我们将分析采用WebSocket协议和采用http协议时的通讯过程。

1. 使用WebSocket的通信过程:

a) 建立连接:

* 客户端向服务器发送WebSocket握手请求
* 服务器响应并完成握手
* WebSocket连接建立

b) 数据传输:

* 服务器持续监控股票价格变化
* 一旦价格发生变化,服务器立即通过WebSocket连接将新价格推送给客户端
* 客户端接收数据并更新显示,无需额外请求

c) 保持连接:

* WebSocket连接持续开放
* 可以进行双向通信,客户端也可以随时向服务器发送数据(如下单指令)

d) 关闭连接:

* 当用户关闭应用时,WebSocket连接关闭

WebSocket实现代码

后端代码 (Python):

```
from flask import Flask, render_template
from flask_socketio import SocketIO, emit
import random
import time

app = Flask(__name__)
socketio = SocketIO(app, cors_allowed_origins="*")

@app.route('/')
def index():
    return render_template('index.html')

def generate_stock_price():
    while True:
        price = round(random.uniform(100, 200), 2)
        socketio.emit('price_update', {'price': price})
        time.sleep(1)

@socketio.on('connect')
def handle_connect():
    print('Client connected')
    socketio.start_background_task(generate_stock_price)

if __name__ == '__main__':
    socketio.run(app, debug=True)
```

前端代码 (JavaScript):

```
<!DOCTYPE html>
<html>
<head>
    <title>Real-time Stock Price</title>
    <script
```

```
src="https://cdnjs.cloudflare.com/ajax/libs/socket.io/4.0.1/socket.io.js"
></script>
</head>
<body>
    <h1>Stock Price: <span id="price">-</span></h1>
    <script>
        const socket = io();

        socket.on('connect', () => {
            console.log('Connected to server');
        });

        socket.on('price_update', (data) => {
            document.getElementById('price').textContent = data.price;
        });
    </script>
</body>
</html>
```

```

WebSocket通信过程解析：
----------

1. 建立连接：

* 客户端通过 `const socket = io();` 初始化WebSocket连接。
* 服务器端的 `@socketio.on('connect')` 处理新的连接。
2. 实时数据传输：

* 服务器使用 `socketio.start_background_task(generate_stock_price)` 启动后台任务生成价格。
* `socketio.emit('price_update', {'price': price})` 持续向所有连接的客户端推送新价格。
* 客户端通过 `socket.on('price_update', (data) => {...})` 监听并处理价格更新。
3. 长连接维护：

* WebSocket连接保持打开状态，无需重复建立连接。
* 服务器可以随时向客户端推送数据，客户端也可以随时向服务器发送数据。

2. 使用HTTP协议的通信过程
==========
```

a) 初始请求:

* 客户端向服务器发送HTTP GET请求获取初始股票价格
* 服务器响应当前价格数据
* HTTP连接关闭

b) 轮询更新:

* 客户端定期(如每5秒)向服务器发送新的HTTP GET请求
* 服务器响应最新价格数据
* 每次请求后HTTP连接关闭

c) 用户操作:

* 当用户需要执行操作(如下单)时,客户端发送HTTP POST请求
* 服务器处理请求并响应
* HTTP连接关闭

HTTP实现代码

后端代码 (Python):

```
from flask import Flask, jsonify
import random

app = Flask(__name__)

@app.route('/')
def index():
    return app.send_static_file('index.html')

@app.route('/get_price')
def get_price():
    price = round(random.uniform(100, 200), 2)
    return jsonify({'price': price})
```

```
if __name__ == '__main__':
    app.run(debug=True)
```

前端代码 (JavaScript):

```
<!DOCTYPE html>
<html>
<head>
    <title>Stock Price (HTTP)</title>
</head>
<body>
    <h1>Stock Price: <span id="price">-</span></h1>
    <script>
        function updatePrice() {
            fetch('/get_price')
                .then(response => response.json())
                .then(data => {
                    document.getElementById('price').textContent = data.price;
                })
                .catch(error => console.error('Error:', error));
        }

        setInterval(updatePrice, 1000);
    </script>
</body>
</html>
```

HTTP通信过程解析：

1. 短连接：

* 每次 `fetch('/get_price')` 都建立一个新的HTTP连接。
* 服务器响应后连接立即关闭。

2. 轮询机制：

* 客户端使用 `setInterval(updatePrice, 1000)` 定期发送请求。
* 服务器的 `@app.route('/get_price')` 处理每个新请求。

3. 请求-响应模式：

* 客户端必须主动发起请求才能获取新数据。
* 服务器无法主动推送数据到客户端。

深入比较WebSocket和HTTP
==========

1. 协议层面：

* WebSocket：使用ws://或wss://协议，在TCP之上实现全双工通信。
* HTTP：使用[http://或https://协议，基于请求-响应模型。
](http://cxyroad.com/”http://%E6%88%96https://%E5%8D%8F%E8%AE%AE%EF%BC%8C%E5%9F%BA%E4%BA%8E%E8%AF%B7%E6%B1%82-%E5%93%8D%E5%BA%94%E6%A8%A1%E5%9E%8B%E3%80%82”)
2. 连接生命周期：

* WebSocket：建立一次连接后保持长时间开放，直到明确关闭。
* HTTP：每次请求都是独立的，连接在响应后关闭（除非使用HTTP/1.1的keep-alive）。
3. 头部开销：

* WebSocket：初始握手后，后续通信的头部很小。
* HTTP：每次请求都包含完整的HTTP头，增加了数据传输量。
4. 实时性：

* WebSocket：可以实现真正的实时通信，延迟最小。
* HTTP：依赖轮询，存在固有延迟，且难以保证所有客户端的同步性。
5. 服务器资源利用：

* WebSocket：长连接可能占用服务器资源，但减少了频繁建立连接的开销。
* HTTP：短连接模式下，服务器需要频繁处理新的连接请求。
6. 复杂性：

* WebSocket：需要特殊的服务器支持（如Flask-SocketIO），客户端库也相对特殊。
* HTTP：使用标准的Web服务器和客户端技术，实现相对简单。
7. 防火墙穿透：

* WebSocket：可能被某些严格的防火墙阻止。
* HTTP：几乎总是被允许通过防火墙。
8. 错误处理和重连：

* WebSocket：需要自行实现断线重连机制。
* HTTP：每个请求相对独立，错误处理较为简单。

由此发现WebSocket在实时性要求高、数据频繁更新的场景下具有显著优势，而HTTP则在简单的请求–响应模式和广泛的兼容性方面更有优势。

原文链接: https://juejin.cn/post/7384652811554488371