

第十章 WebSockets API

内容安排

- 7.1 WebSockets概述
- 7.2 编写简单的Echo WebSockets服务器
- 7.3 使用HTML5 WebSockets API
- 7.4 创建HTML5 WebSockets应用程序
- 7.5 课后思考
- 7.6 小结

7.1 WebSockets概述

- 实时和HTTP
- 解读WebSockets

□ 实时和HTTP

■ 问题:

高并发与用户实时响应是Web应用经常面临的问题，比如金融证券的实时信息，Web导航应用中的地理位置获取，社交网络的实时消息推送等。

■ 解决方案:

传统的请求-响应模式的Web开发在处理此类业务场景时，通常采用实时通讯方案，常见的是轮询和comet技术等。

□ 实时和HTTP

■ 轮询

轮询，原理简单易懂，就是客户端通过一定的时间间隔以频繁请求的方式向服务器发送请求，来保持客户端和服务端的数据同步。问题很明显，当客户端以固定频率向服务器端发送请求时，服务器端的数据可能并没有更新，带来很多无谓请求，浪费带宽，效率低下。

□ 实时和HTTP

■ Comet技术

Comet技术，又可以分为长轮询和流技术。

- 长轮询

长轮询改进了上述的轮询技术，减少了无用的请求。它会为某些数据设定过期时间，当数据过期后才会向服务端发送请求；这种机制适合数据的改动不是特别频繁的情况。

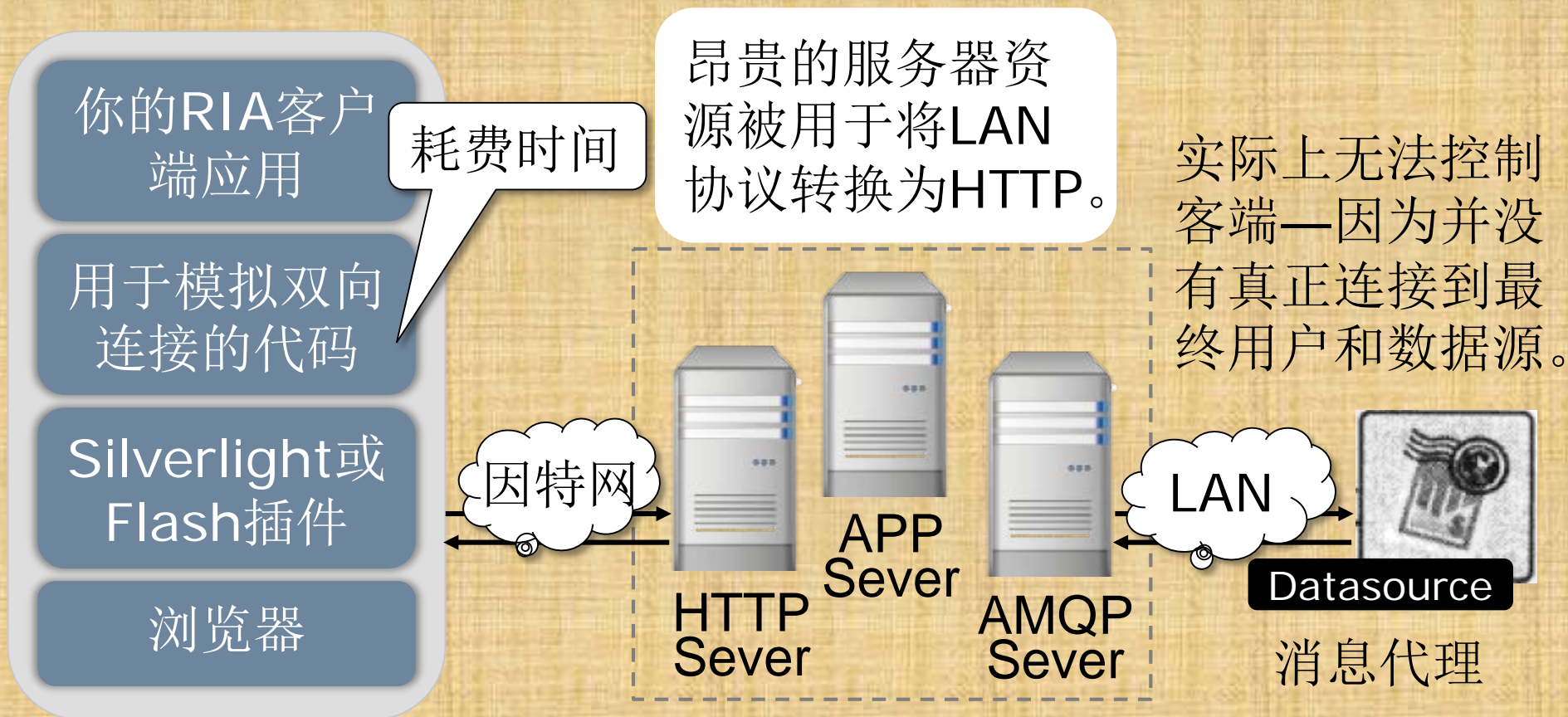
□ 实时和HTTP

■ Comet技术

- 流技术

流技术通常是指客户端使用一个隐藏的窗口与服务端建立一个HTTP长连接，服务端会不断更新连接状态以保持HTTP长连接存活；这样的话，服务端就可以通过长连接主动将数据发送给客户端；流技术在大并发环境下，可能会考验到服务端的性能。

□ 实时和HTTP



混乱、缓慢、易出错的HTTP长轮询等。

图：实时HTTP应用程序的复杂性

□ 实时和HTTP

■ 结论

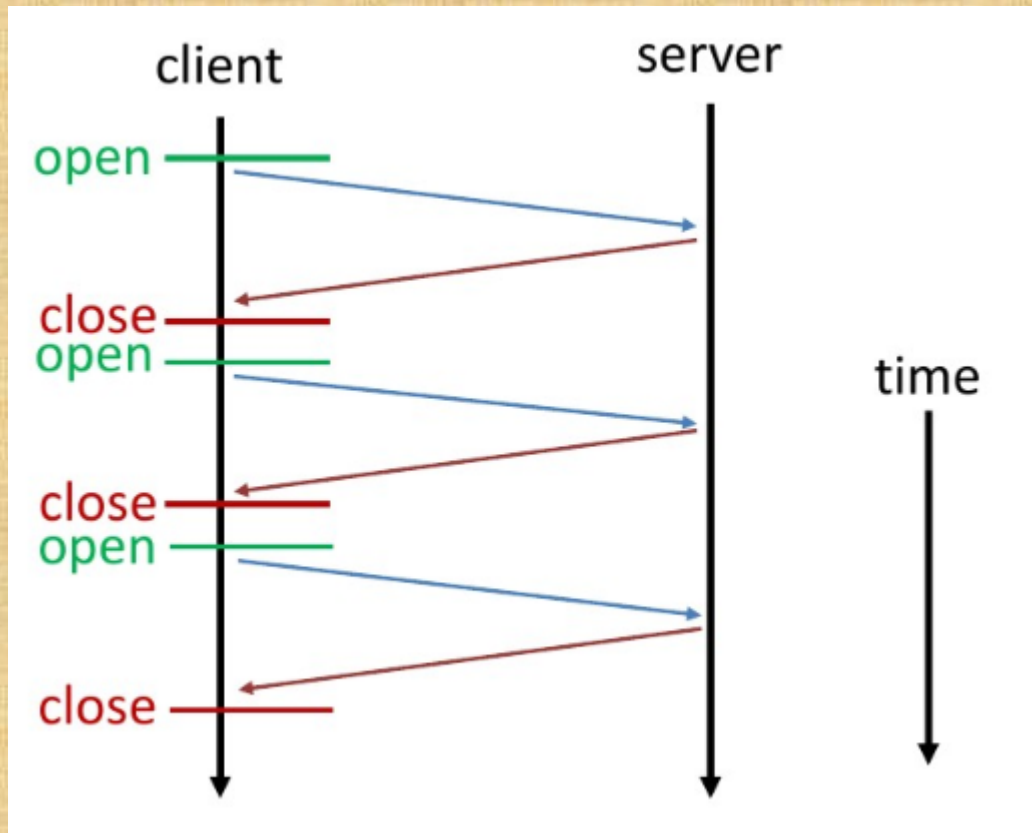
传统Web模式在处理高并发及实时性需求的时候，会遇到难以逾越的瓶颈，我们需要一种高效节能的双向通信机制来保证数据的实时传输。在此背景下，基于HTML5规范的、有Web TCP之称的WebSocks应运而生。

□ 解读WebSockets

■ 什么是WebSocket

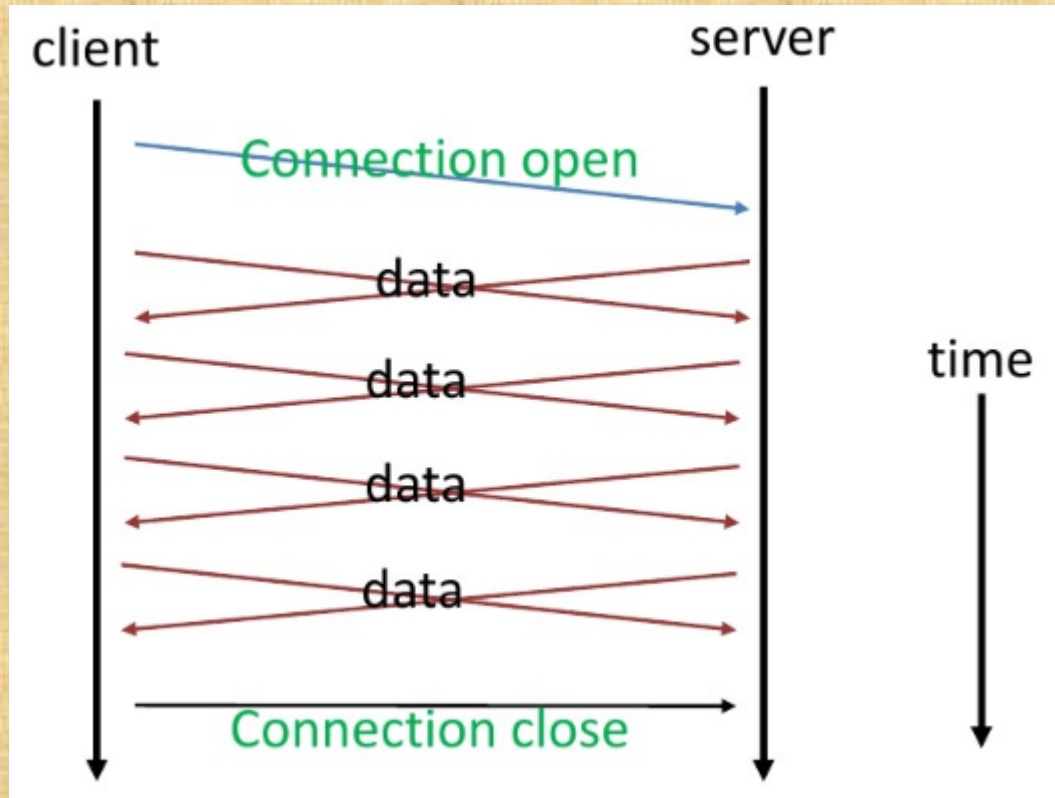
- 是一种网络通信协议-- Web应用程序的传输协议;
- 建立在传输层TCP协议之上;
- 可以实现浏览器与服务器全双工（full-duplex）通信，即允许服务器主动发送信息给客户端;
- 高层协议能够在WebSocket上运行;
- WebSocket连接的是URL，而非因特网上的主机和端口。

□ 解读WebSockets



图：传统HTTP请求响应客户端服务器交互图

□ 解读WebSockets



图：WebSocket请求响应客户端服务器交互图

□ 实时和HTTP

在 WebSocket 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，客户端和服务端之间的数据交换变得更加简单。

□ 实时和HTTP

1.WebSocket握手

- 从客户端到服务器:

GET /chat HTTP/1.1

Host: example.com

Connection: Upgrade

Upgrade: websocket

Sec-WebSocket-Protocol: sample

Sec-WebSocket-Version: 13

Sec-WebSocket-Key: 7cxQRnWs91xJW9T0QLSuVQ==

Origin: http://example.com

将HTTP协议升级到
WebSocket协议

□ 实时和HTTP

1.WebSocket握手

- 从服务器到客户端:

```
HTTP/1.1 101 WebSocket Protocol Handshake
```

```
Upgrade: websocket
```

```
Connection: Upgrade
```

```
Sec-WebSocket-Accept: 7cxQRnWs91xJW9T0QLSuVQ==
```

```
WebSocket-Protocol: sample
```

- 一旦连接建立成功，就可以在全双工模式下在客户端和服务器之间来回传送WebSocket消息。

□ 实时和HTTP

2.WebSocket接口

- WebSocket接口代码:

```
[
    Constructor(DOMString url, optional DOMString protocols),
    Constructor(DOMString url, optional DOMString[] protocols)
]
interface WebSocket : EventTarget {
    readonly attribute DOMString url;

    //就绪状态
    const unsigned short CONNECTING = 0;
    const unsigned short OPEN = 1;
    const unsigned short CLOSING = 2;
    const unsigned short CLOSED = 3;
    readonly attribute unsigned short readState;
    readonly attribute unsigned long bufferedAmount;
```

□ 实时和HTTP

2.WebSocket接口

- WebSocket接口代码（续）：

```
//网络
[TreatNonCallableAsNull] attribute function ? onopen;
[TreatNonCallableAsNull] attribute function ? onerror;
[TreatNonCallableAsNull] attribute function ? onclose;
readonly attribute DOMString extensions;
readonly attribute DOMString protocol;
void close([Clamp] optional unsigned short code, optional DOMString reason);

//消息
[TreatNonCallableAsNull] attribute function ? onmessage;
void send(DOMString data);
void send(ArrayBuffer data);
void send(Blob data);
}
```

7.2 编写简单的Echo WebSocket服务器

在使用WebSockets API之前，先完成WebSocket“Echo”服务器的编写，具体步骤如下：

1、启动Python WebSocket Echo服务器，接收ws://localhost:8080/echos上的连接。

打开命令行窗口，转到该文件所在的文件夹，然后执行以下命令：

```
Python websocket.py
```


7.2 编写简单的Echo WebSocket服务器

2、启动广播服务器(broadcast server)，接收
`ws://localhost:8080/ broadcast`上的连接。

打开命令行窗口，转到该文件所在的文件夹，然后
执行以下命令：

```
Python broadcast.py
```

7.2 编写简单的Echo WebSocket服务器

3、浏览器向WebSocket URL发出一个请求。服务器会返回报头来完成WebSocket握手。

```
#填写我们自己的响应报头
self.send_bytes("HTTP/1.1 101 Switching Protocols\r\n")
    self.send_bytes("Upgrade: WebSocket\r\n")
    self.send_bytes("Connection: Upgrade\r\n")
    self.send_bytes("Sec-WebSocket-Accept: %s\r\n" %
self.hash_key(key))
    if "Sec-WebSocket-Protocol" in headers:
        protocol = headers["Sec-WebSocket-Protocol"]
        elf.send_bytes("Sec-WebSocket-Accept: %s\r\n" %
protocol)
```

7.2 编写简单的Echo WebSocket服务器

- WebSocket.py的完整代码:

```
import asyncio
import socket
import struct
import time
import hashlib

class WebSocketConnection(asyncio.dispatcher_with_send):

    def __init__(self, conn, server):
        asyncio.dispatcher_with_send.__init__(self, conn)

        self.server = server
        self.server.sessions.append(self)
        self.readystate = "connecting"
        self.buffer = ""

    def handle_read(self):
        data = self.recv(1024)
        self.buffer += data
        if self.readystate == "connecting":
            self.parse_connecting()
        elif self.readystate == "open":
            self.parse_frame_type()
```

7.2 编写简单的Echo WebSocket服务器

- WebSocket.py的完整代码（续）：

```
def handle_close(self):
    self.server.sessions.remove(self)
    self.close()

def parse_connecting(self):
    header_end = self.buffer.find("\r\n\r\n")
    if header_end == -1:
        return
    else:
        header = self.buffer[:header_end]
        # remove header and four bytes of line endings from buffer
        self.buffer = self.buffer[header_end+4:]
        header_lines = header.split("\r\n")
        headers = {}

        # validate HTTP request and construct location
        method, path, protocol = header_lines[0].split(" ")
        if method != "GET" or protocol != "HTTP/1.1" or path[0] != "/":
            self.terminate()
            return
```

7.2 编写简单的Echo WebSocket服务器

- WebSocket.py的完整代码（续）：

```
# parse headers
for line in header_lines[1:]:
    key, value = line.split(": ")
    headers[key] = value

headers["Location"] = "ws://" + headers["Host"] + path

self.readystate = "open"
self.handler = self.server.handlers.get(path, None)(self)

if "Sec-WebSocket-Key1" in headers.keys():
    self.send_server_handshake_76(headers)
else:
    self.send_server_handshake_75(headers)

def terminate(self):
    self.ready_state = "closed"
    self.close()

def send_server_handshake_76(self, headers):
    """
    Send the WebSocket Protocol v.76 handshake response
    """
```


7.2 编写简单的Echo WebSocket服务器

- WebSocket.py的完整代码（续）：

```
key1 = headers["Sec-WebSocket-Key1"]
key2 = headers["Sec-WebSocket-Key2"]
# read additional 8 bytes from buffer
key3, self.buffer = self.buffer[:8], self.buffer[8:]

response_token = self.calculate_key(key1, key2, key3)

# write out response headers
self.send_bytes("HTTP/1.1 101 Web Socket Protocol Handshake\r\n")
self.send_bytes("Upgrade: WebSocket\r\n")
self.send_bytes("Connection: Upgrade\r\n")
self.send_bytes("Sec-WebSocket-Origin: %s\r\n" % headers["Origin"])
self.send_bytes("Sec-WebSocket-Location: %s\r\n" % headers["Location"])

if "Sec-WebSocket-Protocol" in headers:
    protocol = headers["Sec-WebSocket-Protocol"]
    self.send_bytes("Sec-WebSocket-Protocol: %s\r\n" % protocol)

self.send_bytes("\r\n")
# write out hashed response token
self.send_bytes(response_token)
```

7.2 编写简单的Echo WebSocket服务器

- WebSocket.py的完整代码（续）：

```
def calculate_key(self, key1, key2, key3):
    # parse keys 1 and 2 by extracting numerical characters
    num1 = int("".join([digit for digit in list(key1) if digit.isdigit()]))
    spaces1 = len([char for char in list(key1) if char == " "])
    num2 = int("".join([digit for digit in list(key2) if digit.isdigit()]))
    spaces2 = len([char for char in list(key2) if char == " "])

    combined = struct.pack(">II", num1/spaces1, num2/spaces2) + key3
    # md5 sum the combined bytes
    return hashlib.md5(combined).digest()

def send_server_handshake_75(self, headers):
    """
    Send the WebSocket Protocol v.75 handshake response
    """

    self.send_bytes("HTTP/1.1 101 Web Socket Protocol Handshake\r\n")
    self.send_bytes("Upgrade: WebSocket\r\n")
    self.send_bytes("Connection: Upgrade\r\n")
    self.send_bytes("WebSocket-Origin: %s\r\n" % headers["Origin"])
    self.send_bytes("WebSocket-Location: %s\r\n" % headers["Location"])
```

7.2 编写简单的Echo WebSocket服务器

- WebSocket.py的完整代码（续）：

```
if "Protocol" in headers:
    self.send_bytes("WebSocket-Protocol: %s\r\n" % headers["Protocol"])

self.send_bytes("\r\n")

def parse_frametype(self):
    while len(self.buffer):
        type_byte = self.buffer[0]
        if type_byte == "\x00":
            if not self.parse_textframe():
                return

def parse_textframe(self):
    terminator_index = self.buffer.find("\xFF")
    if terminator_index != -1:
        frame = self.buffer[1:terminator_index]
        self.buffer = self.buffer[terminator_index+1:]
        s = frame.decode("UTF8")
        self.handler.dispatch(s)
        return True
    else:
        # incomplete frame
        return false
```

7.2 编写简单的Echo WebSocket服务器

- WebSocket.py的完整代码（续）：

```
def send(self, s):
    if self.readystate == "open":
        self.send_bytes("\x00")
        self.send_bytes(s.encode("UTF8"))
        self.send_bytes("\xFF")

def send_bytes(self, bytes):
    asyncio.dispatcher_with_send.send(self, bytes)

class EchoHandler(object):
    """
    The EchoHandler repeats each incoming string to the same Web Socket.
    """
    def __init__(self, conn):
        self.conn = conn

    def dispatch(self, data):
        self.conn.send("echo: " + data)
```

7.2 编写简单的Echo WebSocket服务器

- WebSocket.py的完整代码（完）：

```
class WebSocketServer(asyncore.dispatcher):
    def __init__(self, port=80, handlers=None):
        asyncore.dispatcher.__init__(self)
        self.handlers = handlers
        self.sessions = []
        self.port = port
        self.create_socket(socket.AF_INET, socket.SOCK_STREAM)
        self.set_reuse_addr()
        self.bind(("", port))
        self.listen(5)

    def handle_accept(self):
        conn, addr = self.accept()
        session = WebSocketConnection(conn, self)

if __name__ == "__main__":
    print "Starting WebSocket Server"
    WebSocketServer(port=8080, handlers={"/echo": EchoHandler})
    asyncore.loop()
```


7.2 编写简单的Echo WebSocket服务器

- broadcast.py的完整代码:

```
#!/usr/bin/env python
import asyncio
from websocket import WebSocketServer
class BroadcastHandler(object):
    """
    The BroadcastHandler repeats incoming strings to every connected
    WebSocket.
    """
    def __init__(self, conn):
        self.conn = conn
    def dispatch(self, data):
        for session in self.conn.server.sessions:
            session.send(data)
if __name__ == "__main__":
    print "Starting WebSocket broadcast server"
    WebSocketServer(port=8080, handlers={"/broadcast": BroadcastHandler})
    asyncio.loop()
```

7.3 使用HTML5 WebSockets API

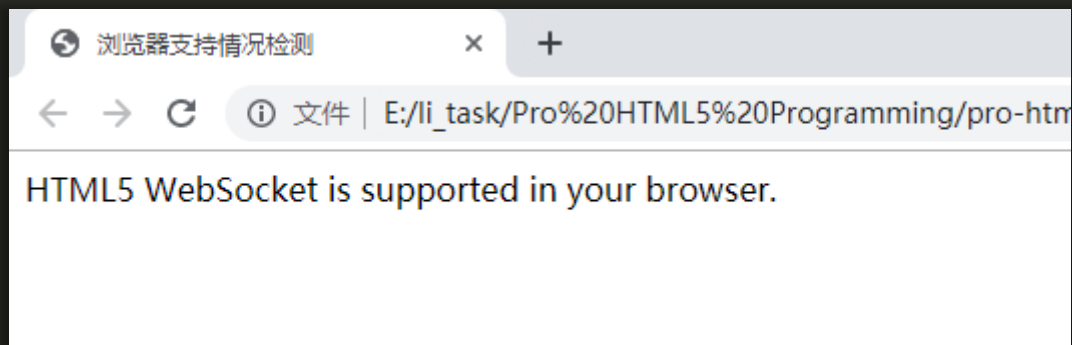
- ❑ 浏览器支持情况检测
- ❑ API的基本用法

□ 浏览器支持情况检测

■ 检测浏览器支持情况方法一

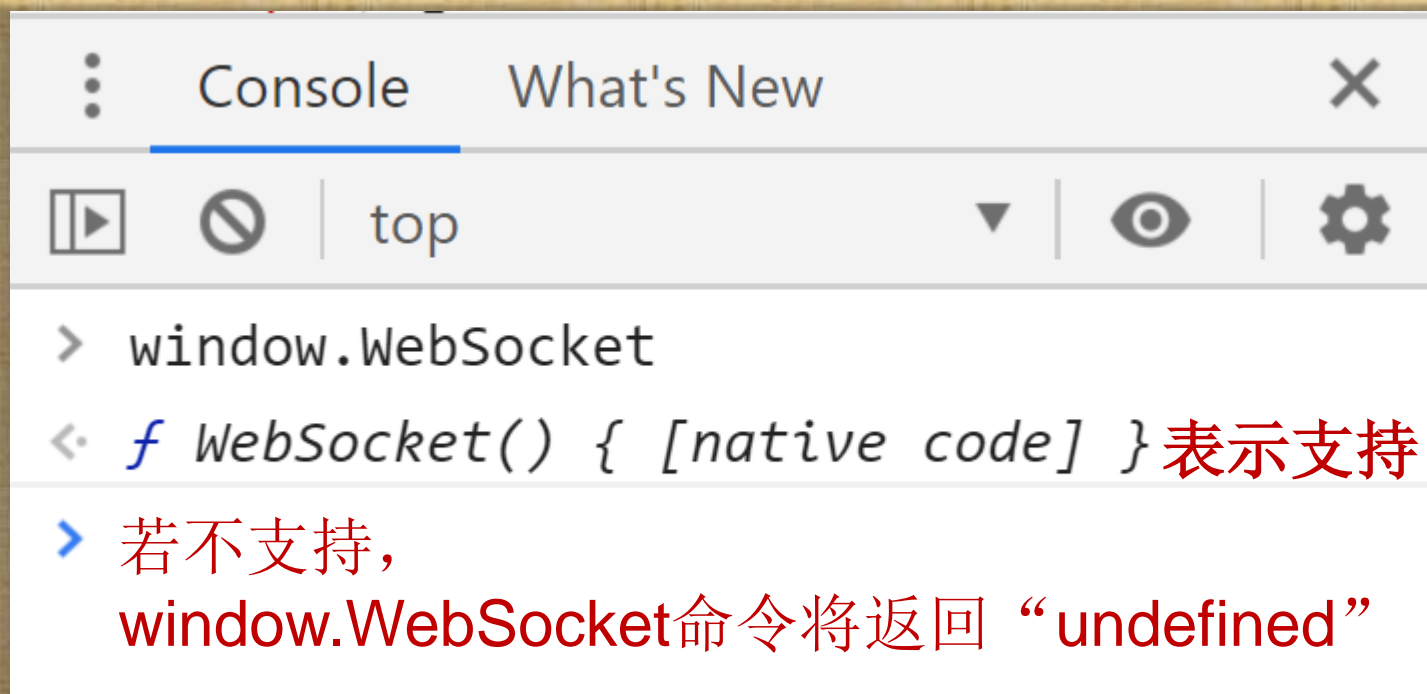
```
<!DOCTYPE html>
<html>
<head>
  <meta charset="utf-8"/>
  <title>浏览器支持情况检测</title>
</head>
<body>
  <div id="support"></div>
<script type="text/javascript">
  loadDemo();
```

```
function loadDemo() {
  if (window.WebSocket) {
    document.getElementById('support').innerHTML = "HTML5 WebSocket is supported in your browser."
  } else {
    document.getElementById('support').innerHTML = "HTML5 WebSocket is not supported in your browser."
  }
}
</script>
</body>
</html>
```



❑ 浏览器支持情况检测

■ 检测浏览器支持情况方法二



图：通过**Google Chrome**开发工具来检测**WebSocket**支持性

□ API的基本用法

■ API用于创建WebSocket对象

第一个参数url，指定连接的URL。

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

第二个参数protocol是可选的，指定了可接受的子协议。可同时用多个协议，它们之间用逗号隔开。

XMPP(extensible messaging and presence protocol, 或称jabber)

□ API的基本用法

■ 可使用的协议包括:

- **XMPP**(extensible messaging and presence protocol, 或称jabber)
- **AMQP**(advanced message queuing protocol)
- **RFB**(remote frame buffer, 或称VNC)
- **STOMP**(streaming text oriented messaging protocol)

□ API的基本用法

■ WebSocket属性

属性	描述
Socket.readyState	只读属性readyState表示连接状态。值： <ul style="list-style-type: none">• 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。

□ API的基本用法

■ 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

■ WebSocket方法

方法	描述
Socket.send()	使用连接发送数据
Socket.close()	关闭发送

□ API的基本用法

1. WebSockets对象的创建及其与WebSockets服务器的连接

ws://和wss://前缀分别表示WebSocket连接和安全的WebSocket连接。

```
url = "ws://localhost:8080/broadcast";  
w = new WebSocket(url);           //创建新的WebSocket实例
```

希望连接的对端的URL

□ API的基本用法

2. 添加事件监听器

WebSocket编程遵循异步编程模型；打开socket后，只需要等待事件发生，而不需要主动向服务器轮询，所以需要在WebSocket对象中添加回调函数来监听事件。

□ API的基本用法

2. 添加事件监听器

```
// 连接建立时触发open事件
w.onopen = function() {
    log("open");
    w.send("thank you for accepting this WebSocket request");
}
// 客户端接收服务端数据时触发message事件
w.onmessage = function(e) {
    log(e.data);
}
// 连接关闭时触发close事件
w.onclose = function(e) {
    log("closed");
}
// 通信发生错误时触发error事件
w.onerror = function(e) {
    log("error");
}
```

□ API的基本用法

2. 添加事件监听器 — 消息处理器

```
w.binaryType = "arraybuffer";  
w.onmessage = function(e) {  
    // data属性值可以是Blob或ArrayBuffer,  
    // 具体取决于WebSocket中binaryType属性的值。  
    log(e.data);  
}
```

□ API的基本用法

3. 发送消息

```
document.getElementById("sendButton").onclick = function() {  
    w.send(document.getElementById("inputMessage").value);  
}
```

□ API的基本用法

- 帶有WebSocket代码的整个HTML页面代码

```
<!DOCTYPE html>
<html>
<head>
  <meta charset="utf-8"/>
  <title>websockets API</title>
</head>
<body>
  <input type="text" id="inputMessage" value="hello, WebSocket!">
  <button id="sendButton">Send</button>
  <pre id="output"></pre>

  <script type="text/javascript">
    var log = function(s) {
      if (document.readyState !== "complete") {
        log.buffer.push(s);
      } else {
        document.getElementById("output").textContent += (s + "\n"
          );
      }
    }
  </script>
</body>
</html>
```

□ API的基本用法

- 帶有WebSocket代码的整个HTML页面代码（续）

```
log.buffer = [];  
  
if (this.MozWebSocket) {  
    WebSocket = MozWebSocket;  
}  
// 1.WebSockets对象的创建及其与WebSockets服务器的连接  
url = "ws://localhost:8080/echo";  
w = new WebSocket(url);           //创建新的WebSocket实例  
  
// 连接建立时触发open事件  
w.onopen = function() {  
    log("open");  
    // WebSocket可以发送字符串  
    w.binaryType = "arraybuffer";  
    w.send("thank you for accepting this WebSocket request");  
    // WebSocket API支持以二进制数据的形式发送Blob和ArrayBuffer  
    var a = new Uint8Array([8,6,7,5,3,0,9]);  
    w.send(a.buffer);  
}
```


□ API的基本用法

- 带有WebSocket代码的整个HTML页面代码（完）

```
// 客户端接收服务端数据时触发message事件
w.onmessage = function(e) {
    // data属性值可以是Blob或ArrayBuffer,
    // 具体取决于WebSocket中binaryType属性的值。
    log(e.data.toString());
}
// 连接关闭时触发close事件
w.onclose = function(e) {
    log("closed");
}
// 通信发生错误时触发error事件
w.onerror = function(e) {
    log("error");
}

window.onload = function() {
    log(log.buffer.join("\n"));
    document.getElementById("sendButton").onclick = function() {
        w.send(document.getElementById("inputMessage").value);
    }
}

</script>
</body>
</html>
```

□ API的基本用法

4. 运行WebSocket页面

1. 打开命令行窗口，转到WebSocket代码所在目录下，输入以下命令启动服务器：

```
python -m SimpleHTTPServer 9999
```

2. 打开另一个命令行窗口，转到WebSocket代码所在目录，输入以下命令来启动WebSocket服务：

```
python websocket.py
```

3. 打开支持WebSockets的浏览器，浏览

```
http://localhost:9999/websocket.html
```

□ API的基本用法

- 运行效果

Hello, Web Socket!

Send

open

echo: thank you for accepting this websocket request

echo: Hello, WebSocket!

□ API的基本用法

4. 运行WebSocket页面

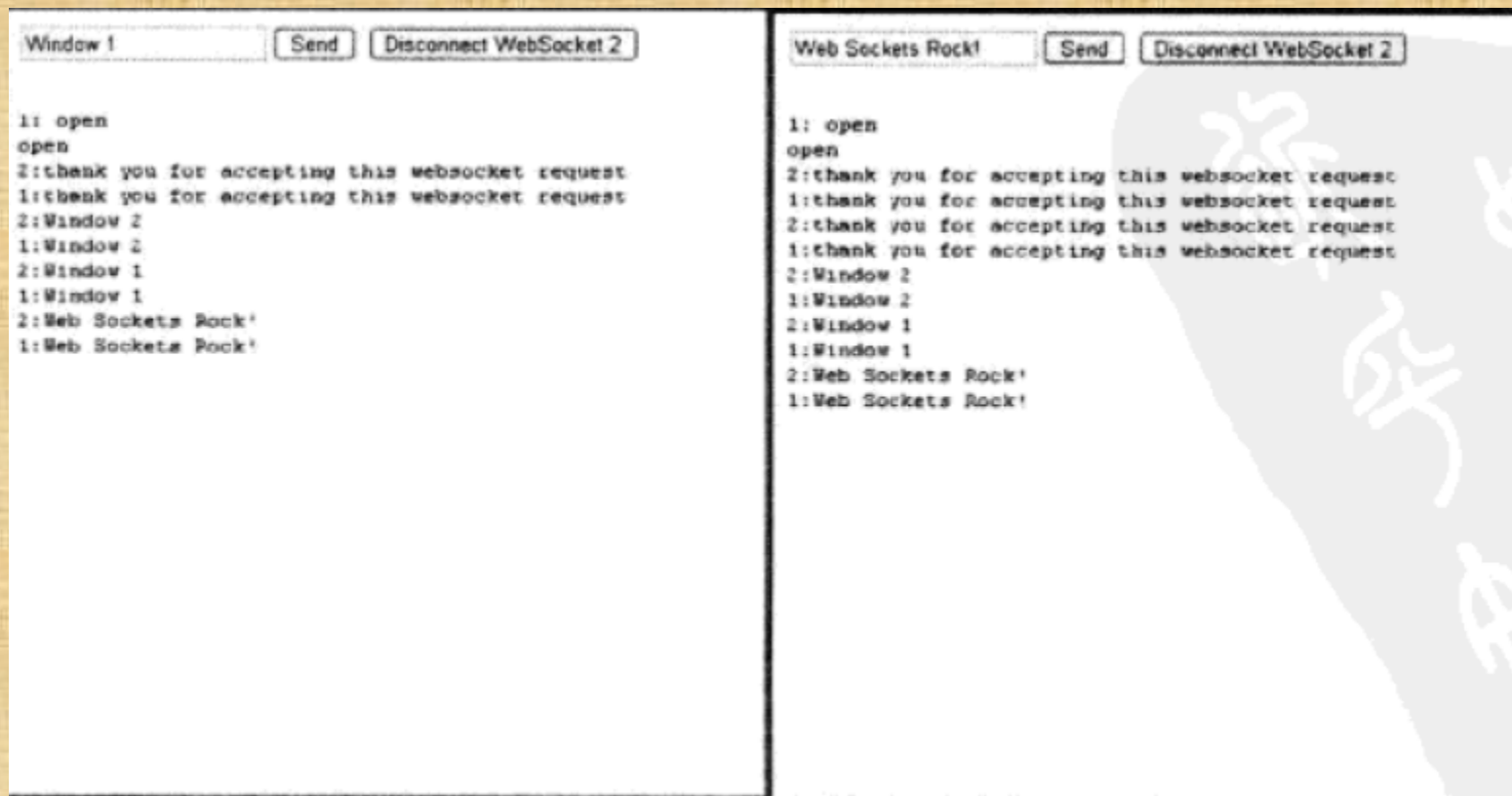
4. 连接到之前创建的broadcast服务。具体步骤如下：先将正在运行WebSocket服务器的命令行窗口关闭，然后转到WebSocket代码所在目录，输入如下命令启动Python WebSocket服务器。

```
Python broadcast.py
```

5. 分别打开两个支持WebSocket的浏览器，统一转到 <http://localhost:9999/broadcast.html>

□ API的基本用法

- 运行效果



```
Window 1      Send  Disconnect WebSocket 2

1: open
open
2:thank you for accepting this websocket request
1:thank you for accepting this websocket request
2:Window 2
1:Window 2
2:Window 1
1:Window 1
2:Web Sockets Rock!
1:Web Sockets Rock!

Web Sockets Rock!  Send  Disconnect WebSocket 2

1: open
open
2:thank you for accepting this websocket request
1:thank you for accepting this websocket request
2:thank you for accepting this websocket request
1:thank you for accepting this websocket request
2:Window 2
1:Window 2
2:Window 1
1:Window 1
2:Web Sockets Rock!
1:Web Sockets Rock!
```


7.4 创建HTML5 WebSockets应用程序

- 编写HTML文件
- 添加WebSockets代码
- 添加Geolocation代码
- 合并所有内容
- 最终代码

7.4 创建HTML5 WebSockets应用程序

- tracker.html完整代码

```
<!DOCTYPE html>
<html>
<head>
  <meta charset="utf-8"/>
  <title>HTML5 WebSocket / Geolocation Tracker</title>
  <link rel="stylesheet" href="styles.css">
</head>
<body onload="loadDemo()">
  <h1>HTML5 WebSocket / Geolocation Tracker</h1>
  <div>
    <strong>Geolocation</strong>:
    <p id="geoStatus">HTML5 Geolocation is <strong>not</strong> supported in your
    browser.</p>
  </div>
  <div>
    <strong>WebSocket</strong>:
    <p id="socketStatus">HTML5 Web Sockets are <strong>not</strong> supported in
    your browser.</p>
  </div>
```

7.4 创建HTML5 WebSockets应用程序

- tracker.html完整代码（续）

```
<script>
// WebSocket的引用
var socket;
// 为该会话生成唯一的随机ID
var myId = Math.floor(100000*Math.random());
// 当前显示的行数
var rowCount = 0;
// 更新状态信息
function updateSocketStatus(message) {
    document.getElementById("socketStatus").innerHTML = message;
}
// 更新地理位置状态信息
function updateGeolocationStatus(message) {
    document.getElementById("geoStatus").innerHTML = message;
}
// Geolocation异常处理信息
function handleLocationError(error) {
    switch(error.code) {
        case 0:
            updateGeolocationStatus("There was an error while retrieving your
            location: " + error.message);
            break;
```

7.4 创建HTML5 WebSockets应用程序

- tracker.html完整代码（续）

```
        case 1:
            updateGeolocationStatus("The user prevented this page from retrieving
                a location.");
            break;
        case 2:
            updateGeolocationStatus("The browser was unable to determine your
                location: " + error.message);
            break;
        case 3:
            updateGeolocationStatus("The browser timed out before retrieving the
                location.");
            break;
    }
}
// 页面初始加载时会调用loadDemo
function loadDemo() {
    // 进行检测，确保浏览器支持sockets
    if (window.WebSocket) {
        // broadcast WebSocket server服务器位置
        url = "ws://localhost:8080";
        // 实例化WebSocket
        socket = new WebSocket(url);
```

7.4 创建HTML5 WebSockets应用程序

- tracker.html完整代码（续）

```
// 告诉用户已成功建立连接
socket.onopen = function() {
    updateSocketStatus("Connected to WebSocket tracker server");
}
// 告知用户消息已送达
socket.onmessage = function(e) {
    updateSocketStatus("Updated location from " + dataReturned(e.data));
}
}
// 检测Geolocation服务的浏览器支持情况
var geolocation;
if(navigator.geolocation) {
    geolocation = navigator.geolocation;
    updateGeolocationStatus("HTML5 Geolocation is supported in your browser.")
    ;
}
// 使用Geolocation API注册位置更新处理函数
geolocation.watchPosition(updateLocation,
                           handleLocationError,
                           {maximumAge:20000});
}
```


7.4 创建HTML5 WebSockets应用程序

- tracker.html完整代码（续）

```
// 当新位置可用时，更新位置信息
function updateLocation(position) {
    var latitude = position.coords.latitude;    //经度
    var longitude = position.coords.longitude;  //纬度
    var timestamp = position.timestamp;        //时间戳

    updateGeolocationStatus("Location updated at " + timestamp);

    // 通过WebSocket发送我的位置
    var toSend = JSON.stringify([myId, latitude, longitude]);
    sendMyLocation(toSend);
}

// 将位置信息发送给服务器
function sendMyLocation(newLocation) {
    if (socket) {
        socket.send(newLocation);
    }
}
```

7.4 创建HTML5 WebSockets应用程序

- tracker.html完整代码（续）

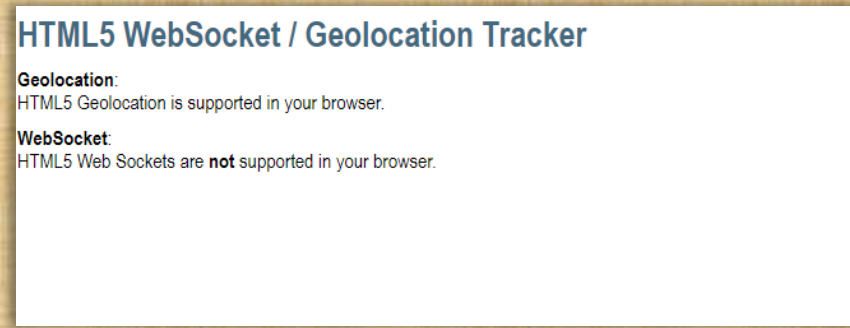
```
function dataReturned(LocationData) {  
    // 从数据中拆分出 ID, 经度, 和纬度  
    var allData = JSON.parse(locationData);  
    var incomingId    = allData[1];  
    var incomingLat   = allData[2];  
    var incomingLong  = allData[3];  
  
    // 根据ID定位到HTML元素  
    // 如果不存在, 就创建  
    var incomingRow = document.getElementById(incomingId);  
    if (!incomingRow) {  
        incomingRow = document.createElement('div');  
        incomingRow.setAttribute('id', incomingId);  
  
        incomingRow.userText = (incomingId == myId) ?  
                                'Me' :  
                                'User ' + rowCount;  
  
        rowCount++;  
        document.body.appendChild(incomingRow);  
    }  
}
```

7.4 创建HTML5 WebSockets应用程序

- tracker.html完整代码（完）

```
// 使用新的值更新对应行文本
incomingRow.innerHTML = incomingRow.userText + " \\ Lat: " +
    incomingLat + " \\ Lon: " +
    incomingLong;

return incomingRow.userText;
}
</script>
</body>
</html>
```



7.5 课后思考

- 阐述WebSocket、Socket、TCP、HTTP的异同点。
- 列举WebSocket 的使用场景。
- 阐述WebSocket 的优势。

7.6 小结

1. 了解HTML5 WebSocket协议本身的特性，对比基于轮询的通信策略和WebSocket通信技术；
2. 搭建WebSocket服务器演示WebSocket的运行情况
3. 创建一个综合Geolocation和WebSocket的示例应用。