为什么面试官都爱问websocket?

为什么面试官都爱问websocket?

什么是webSocket?

> WebSocket 是一种在单个 TCP 连接上进行全双工通信的应用层协议，它弥补 HTTP 协议在持久通信能力上的不足，允许服务器主动向客户端推送数据，并且简化了客户端和服务器之间的数据交换。
> 在 WebSocket 通讯中，浏览器和服务器只需完成一次握手，便可创建持久性连接。WebSocket最大特点是服务器可以主动向客户端推送信息，同时客户端也能够主动向服务器发送信息，实现了真正的双向平等对话，属于服务器推送技术的一种。

简单来说：**webSocket就是服务器和客户端相互主动传输信息的约定协议。**

优点:

- **基于TCP协议**： WebSocket建立在TCP之上，这使得服务器端的实现相对容易。
- **与HTTP兼容性良好**： WebSocket与HTTP协议兼容，使用HTTP协议进行握手阶段，因此默认端口与HTTP相同（80和443），且不易被屏蔽。这意味着它可以通过各种HTTP代理服务器，增加了通信的灵活性。
- **轻量级数据格式和高效通信**： 在连接创建后，持久保存连接状态，并且交换数据时，用于协议控制的数据包头部相对较小；
- **支持文本和二进制数据**： WebSocket不仅可以发送文本数据，还可以发送二进制数据，相对HTTP，可以更轻松地处理二进制内容。
- **无同源限制**： 与传统的AJAX请求不同，WebSocket没有同源限制，客户端可以与任意服务器通信，不需要处理跨域
- **标识符简单明了**： WebSocket的协议标识符是"ws"（如果加密则为"wss"），而服务器网址就是URL本身，这使得其使用和理解都相对简单直观。
- **支持扩展性**： WebSocket 定义了扩展，用户可以扩展协议、实现部分自定义的子协议

缺点:

- **安全性**：WebSocket 使用的是持久性连接，连接建立后会长时间保持打开状态。会增加服务器资源的消耗。
- **兼容性**：旧版浏览器中可能会出现兼容性问题。
- **数据包大小的限制**： WebSocket 协议发送的数据包不能超过 2GB。

应用场景

- 即时聊天通信
- 多玩家游戏
- 在线协同编辑/编辑
- 实时地图位置
- 即时Web应用程序

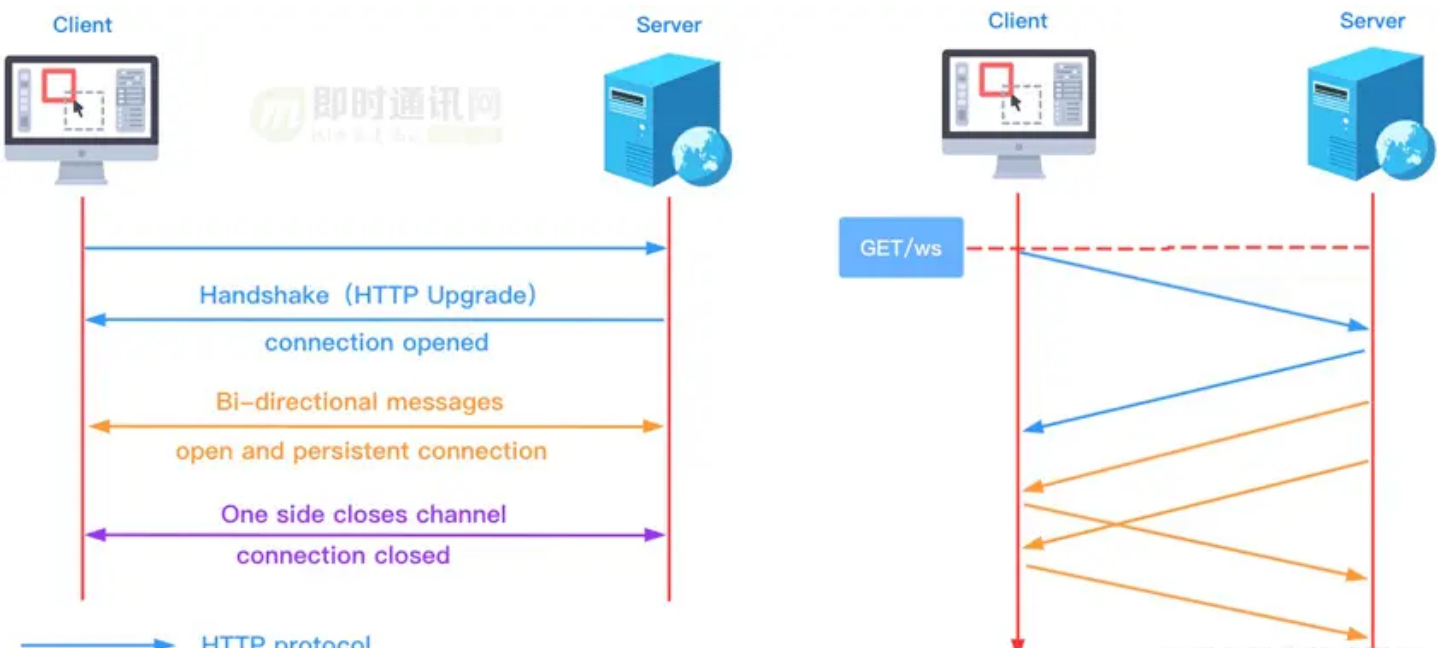
websocket的具体使用

5分钟从0到1，学会webSocket的使用

WebSocket 原理

WebSocket是如何建立连接?

WebSocket 连接的生命周期



从上图可知：在使用 WebSocket 实现全双工通信之前，客户端与服务器之间需要先进行握手（Handshake），，`WebSocket 复用了 HTTP 的握手通道`，即客户端通过 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。

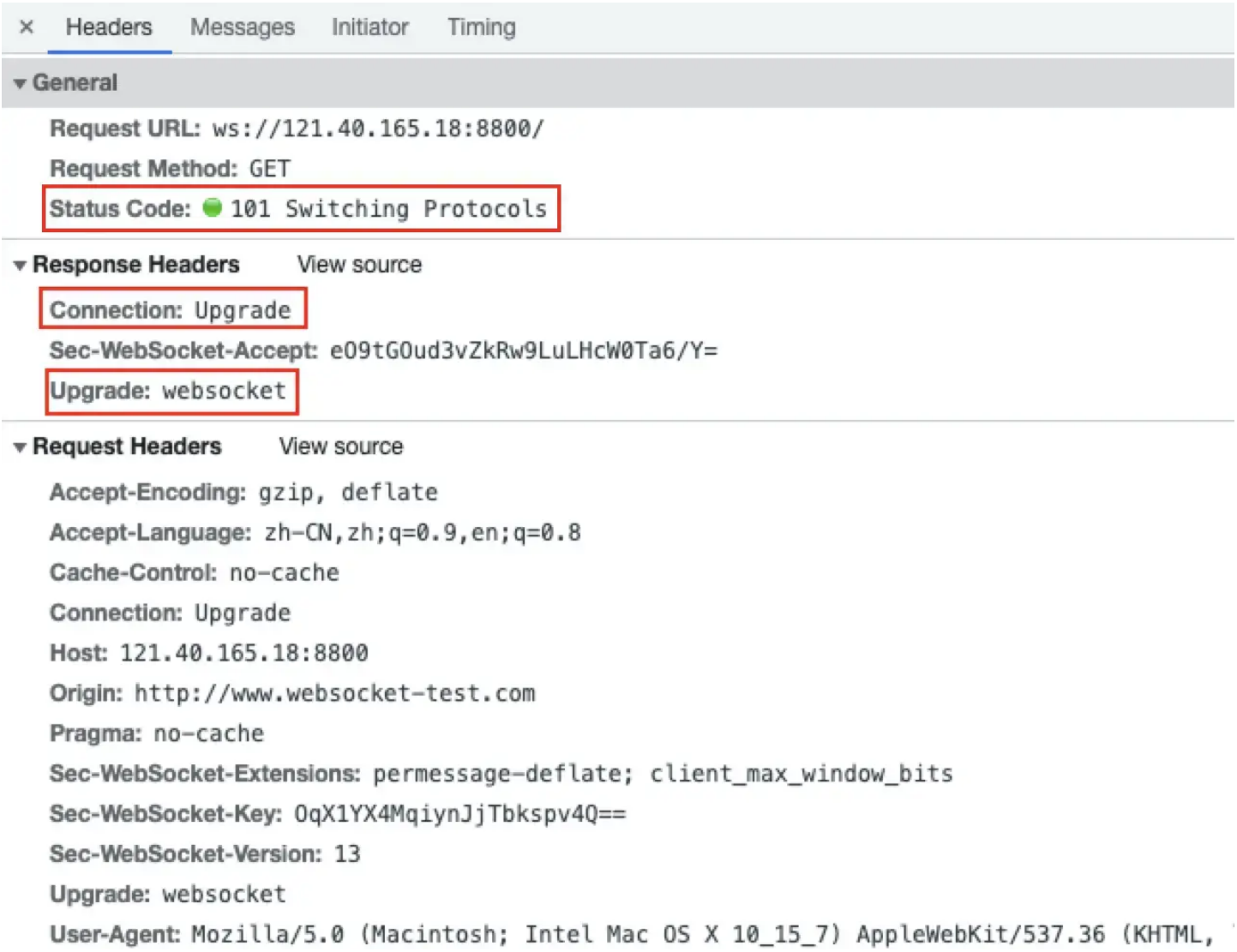
握手是在通信电路创建之后，信息传输开始之前。

那你可能会问，利用 HTTP 完成握手有什么好处呢?

1. 确保 WebSocket 和 HTTP 基础设备的兼容性，因为它们都可以运行在常见的 80 端口或 443 端口

2. 利用了 HTTP 的 Upgrade 机制，简化了协议升级过程，使通信双方能够快速而可靠地建立 WebSocket 连接。

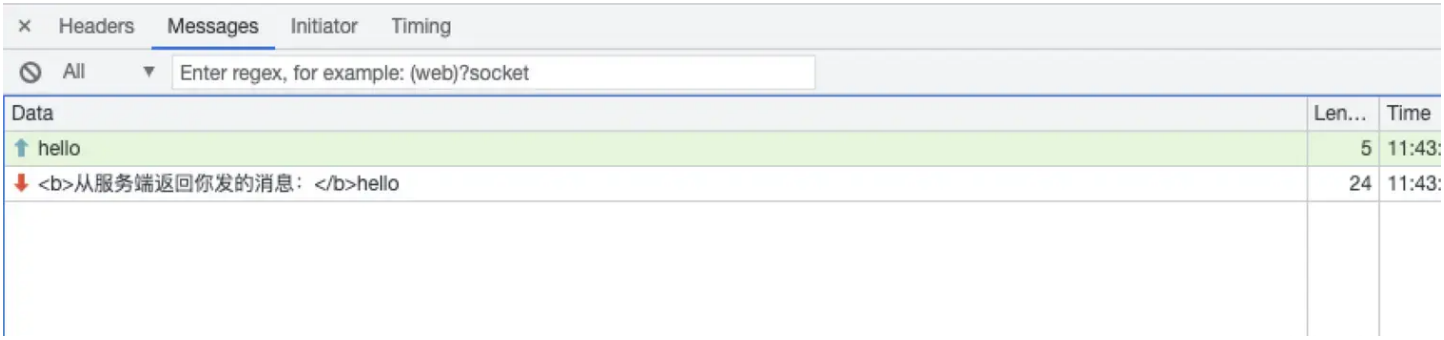
看个具体的请求（在网上找了个**在线测试**）：



图中有`几个比较关键的点`：

- `101 状态码`，表示协议切换；
- `Connection: Upgrade` 表示要升级协议；
- `Upgrade: websocket` 表示要升级到 websocket 协议；
- `Sec-WebSocket-Key`：与服务端响应头部的 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；这里的“配套”指的是：Sec-WebSocket-Accept 是根据请求头部的 Sec-WebSocket-Key 计算而来，计算过程大致为基于 SHA1 算法得到摘要并转成 base64 字符串。

图中的请求已经完成握手并正常工作了：



如何交换数据?

具体的数据格式是怎么样的呢?

WebSocket 的每条消息可能会被切分成多个数据帧（最小单位）。发送端会将消息切割成多个帧发送给接收端，接收端接收消息帧，并将关联的帧重新组装成完整的消息。

看一个来自 MDN 上的示例：

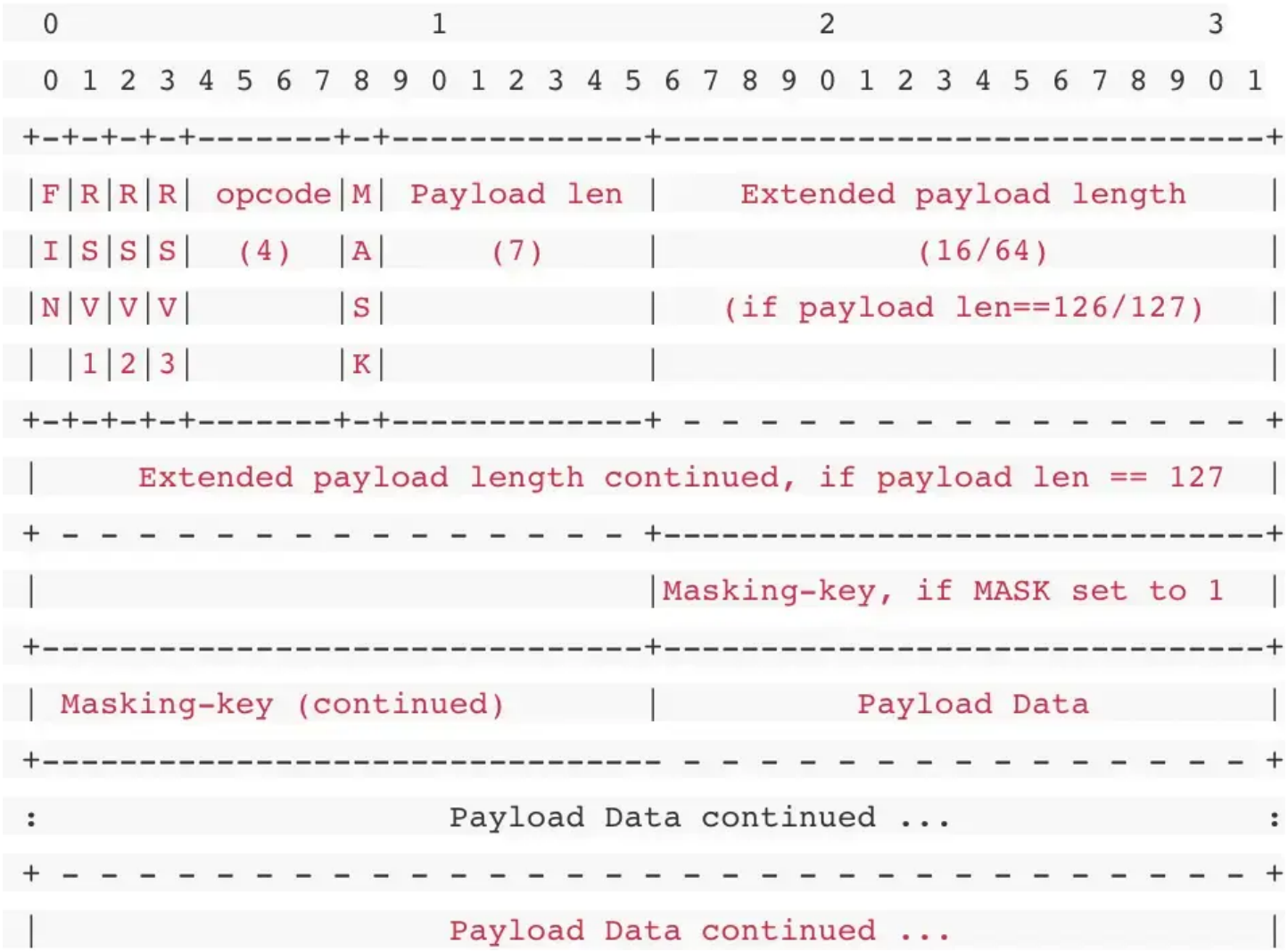
```
// Client 发送第一条消息，FIN=1 表示这是消息的最后一帧，opcode=0x1 表示传输的是文本数据，msg="hello" 表示消息内容为 "hello"
Client: FIN=1, opcode=0x1, msg="hello"

// Server 接收到消息并立即处理，返回响应消息 "Hi."
Server: (process complete message immediately) Hi.

// Client 发送第二条消息，FIN=0 表示这不是消息的最后一帧，opcode=0x1 表示传输的是文本数据，msg="and a" 表示消息内容为 "and a"
Client: FIN=0, opcode=0x1, msg="and a"

// Server 正在监听，等待接收到完整的消息
Server: (listening, new message containing text started)

// Client 继续发送第三条消息的第一个片段，FIN=0 表示这不是消息的最后一帧，opcode=0x0 表示这是一个延续帧，msg="happy new" 表示消息内容的一部分
Client: FIN=0, opcode=0x0, msg="happy new"

// Server 正在监听，将接收到的片段连接到之前的消息中
Server: (listening, payload concatenated to previous message)

// Client 发送第三条消息的最后一个片段，FIN=1 表示这是消息的最后一帧，opcode=0x0 表示这是一个延续帧，msg="year!" 表示消息内容的最后一部分
Client: FIN=1, opcode=0x0, msg="year!"

// Server 接收到完整的消息并进行处理，返回响应消息 "Happy new year to you too!"
Server: (process complete message) Happy new year to you too!


```

在该示例中，客户端向服务器发送了两条消息，第一个消息在单个帧中发送，而第二个消息跨三个帧发送。当 WebSocket 的接收方收到一个数据帧时，会根据 FIN 字段值来判断是否收到消息的最后一个数据帧。利用 FIN 和 Opcode，我们就可以实现跨帧发送消息。

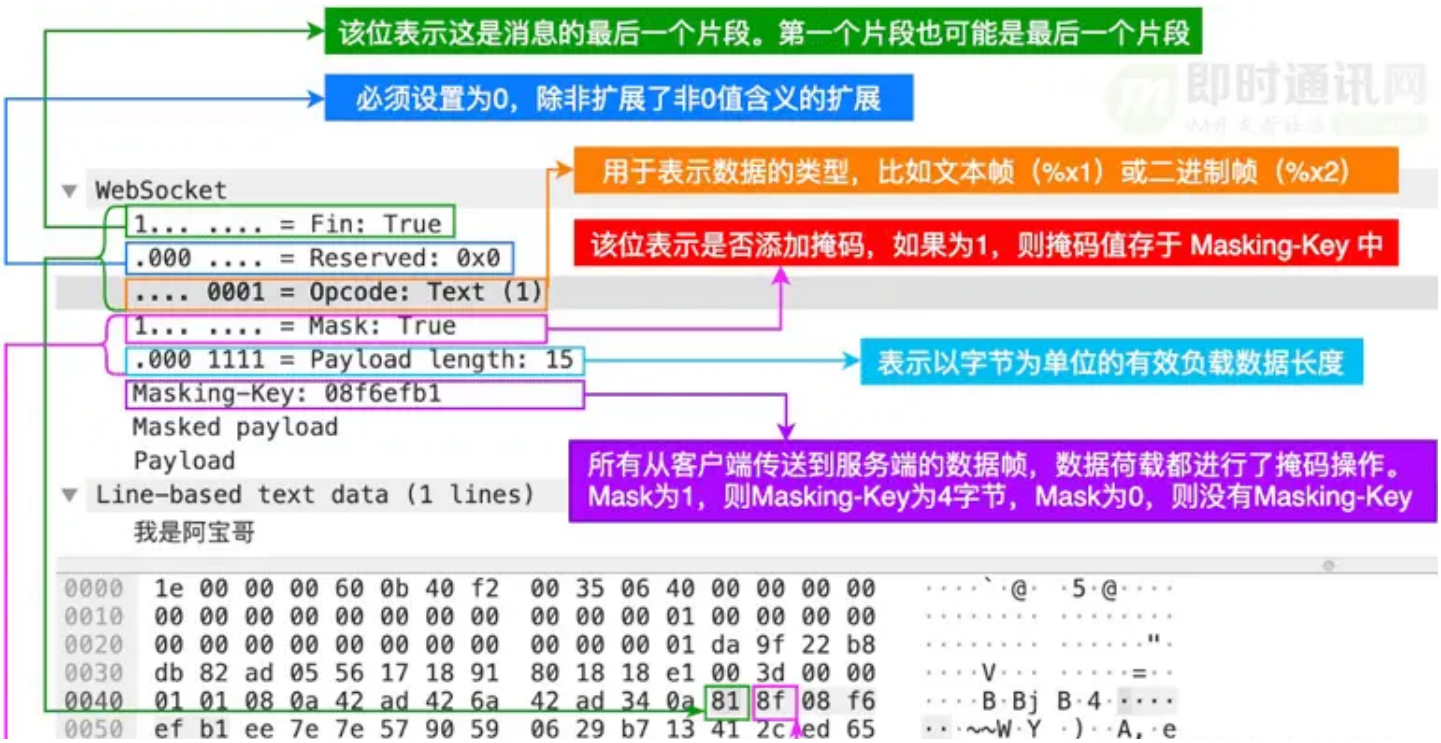
其中 Opcode 表示操作码，它的可能值有：

- 0x1，传输数据是文本；
- 0x2，传输数据是二进制数据；
- 0x0，表示该帧是一个延续帧（这意味着服务器应该将帧的数据连接到从该客户端接收到的最后一个帧）；
- 0x3-7：保留的操作代码，用于后续定义的非控制帧；
- 0x8：表示连接断开；
- 0x9：表示这是一个心跳请求（ping）；
- 0xA：表示这是一个心跳响应（pong）；
- 0xB-F：保留的操作代码，用于后续定义的控制帧；

具体的数据帧格式大概长下面这样（从左到右，单位是比特）：



我们来结合实际的数据帧一起来看一下



具体看下每一个字段：

- FIN：1 个比特，值为 1 表示这是消息的最后一帧，为 0 则不是；
- RSV1, RSV2, RSV3：各占 1 个比特，一般情况下全为 0，非零值表示采用 WebSocket 扩展；
- Mask: 1 个比特，表示是否要对数据进行掩码操作；
- Payload length：数据负载的长度，单位是字节。为 7 位，或 7+16 位，或 1+64 位；
- 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；
- Payload data：具体数据；

如何维持连接

如果我们使用 WebSocket 进行通信，建立连接之后怎么判断连接正常没有断开或者服务是否可用呢？

如何判断在线离线？

客户端首次发送请求至服务端时会携带一个唯一标识和时间戳，服务端可以根据唯一标识查询数据库或缓存，若不存在则将该请求信息存储。

当客户端定时再次发送请求时，依然携带同一唯一标识和时间戳。服务端再次检查数据库或缓存，若存在该唯一标识，便取出上次存储的时间戳。然后，服务端使用当前时间戳减去上次的时间戳，得到的毫秒数判断是否超过指定的时间阈值。如果未超过阈值，则认为客户端仍在线；否则，客户端被视为离线。

如何解决断线问题

1. **心跳检测（Heartbeat）**：通过定期发送心跳消息，可以确保客户端和服务器之间的连接处于活跃状态。如果一段时间内未收到来自客户端的心跳消息，服务器可以认为客户端已经断线，并采取相应的措施，例如关闭连接或重新建立连接。
2. **自动重连（Automatic Reconnection）**：在客户端检测到连接断开后，可以自动尝试重新连接服务器。可以通过实现指数退避策略（exponential backoff）来控制重连尝试的频率，以避免对服务器造成过大的负载。
3. **断线重连策略（Reconnection Strategy）**：可以根据具体情况制定不同的断线重连策略。例如，在一段时间内连续尝试重连多次失败后，可以采取延迟重连的策略，以避免过度频繁地尝试重连。
4. **连接状态管理（Connection State Management）**：在客户端代码中维护连接的状态信息，以便及时检测连接的断开和重新连接的状态变化。这样可以使应用程序更容易地处理连接断开和重新连接时的逻辑。
5. **异常处理（Exception Handling）**：及时捕获和处理在连接过程中可能出现的异常情况，例如网络超时、连接被拒绝等，以提高系统的稳定性和可靠性。

心跳检测（Heartbeat）

针对websocket断线我们来分析一下，

- 断线的可能原因1：websocket超时没有消息自动断开连接，应对措施：
- 这时候我们就需要知道服务端设置的超时时长是多少，在小于超时时间内发送心跳包，在实现心跳包时，可以通过两种方式之一来发送心跳包：客户端主动发送上行心跳包或者服务端主动发送下行心跳包。
- 下面主要讲一下客户端也就是前端如何实现心跳包：
- 首先了解一下心跳包机制
- 跳包之所以叫心跳包是因为：它像心跳一样每隔固定时间发一次，以此来告诉服务器，这个客户端还活着。实际上，这种机制是为了维持长连接而设计的。通常情况下，心跳包的内容并没有特别的规定，通常是一个很小的数据包，甚至可能只包含包头的空包。
- 在TCP的机制里面，本身是存在有心跳包的机制的，也就是TCP的选项：SO_KEEPALIVE。系统默认是设置的2小时的心跳频率。但是这种心跳包机制并不能检测到诸如机器断电、网线拔出或防火墙等造成的断线情况。此外，即使检测到了断线，逻辑层处理断线的效果可能也不是十分可靠。通常情况下，如果只是简单地用于保活连接，TCP 的心跳包机制仍然是可行的。
- 心跳包通常通过在逻辑层发送空的 echo 包来实现。**具体而言，服务器会在一定时间间隔内发送一个空的数据包给客户端，然后期待客户端回复一个相同的空包。如果服务器在一定时间内未收到客户端发送的回复包，就会判定客户端已经掉线。这种机制确保了连接的持续性和稳定性，因为它能够及时发现并处理连接状态的变化。。**

- 在长连接下，有可能会出现很长一段时间都没有数据往来的情况。但是理论上，这个连接是应该一直保持连接的，但是实际情况中，如果中间节点出现什么故障我们都是难以知道的。更要命的是，有的节点(防火墙)会自动把一定时间之内没有数据交互的连接给断掉。在这个时候，就需要我们的心跳包了，用于维持长连接，保持连接的活跃性。
- 心跳检测步骤：
  a. 客户端会定期发送一个探测包（心跳包）给服务器，以确保连接的活跃性。
  b. 在发送探测包的同时，客户端会启动一个超时定时器，以便在规定的时间内等待服务器的响应。
  c. 当服务器接收到客户端发送的探测包时，会立即回应一个确认包，以表明服务器正常接收到了探测包。
  d. 如果客户端收到服务器的应答包，则说明服务器正常：客户端在收到服务器的确认包后，会立即删除之前启动的超时定时器，表明服务器正常运行。
  e. 如果客户端的超时定时器超时，仍未收到应答包，则说明服务器挂了：如果客户端的超时定时器到期时仍未收到服务器的确认包，则客户端会认为服务器已经挂了，进而采取相应的措施，例如重新连接或者进行错误处理。

```

    // 前端解决方案：心跳检测
var heartCheck = {
    timeout: 30000, //30秒发一次心跳，设置心跳间隔为30秒
    timeoutObj: null, // 定义超时定时器对象，用于发送心跳包
    serverTimeoutObj: null, // 定义服务器超时定时器对象，用于检测服务器是否响应心跳包
    reset: function(){ // 重置定时器函数，用于清除之前的定时器对象
        clearTimeout(this.timeoutObj); // 清除发送心跳包的定时器
        clearTimeout(this.serverTimeoutObj); // 清除检测服务器响应的定时器
        return this; // 返回当前对象以支持链式调用
    },
    start: function(){ // 启动心跳检测函数
        var self = this; // 缓存当前对象的引用
        this.timeoutObj = setTimeout(function(){ // 设置定时器，定时发送心跳包
            // 在定时器回调函数中发送心跳包
            // 一般情况下，后端收到心跳包后会返回一个心跳响应消息
            ws.send("ping"); // 发送心跳包消息给服务器
            console.log("ping!"); // 打印日志，表示发送了心跳包

            // 设置服务器响应超时定时器
            self.serverTimeoutObj = setTimeout(function(){ // 如果超过一定时间还没收到心跳响应，说明服务器已断开连接
                ws.close(); // 手动关闭 WebSocket 连接
            }, self.timeout); // 设置超时时间为心跳间隔
        }, this.timeout); // 设置发送心跳包的定时器间隔为心跳间隔
```

```
    }
}

```

断线重连策略（Reconnection Strategy）

- 断线的可能原因2：websocket异常包括服务端出现中断，交互切屏等等客户端异常中断等等
- 当若服务端宕机了，客户端怎么做、服务端再次上线时怎么做?
- 客户端在检测到连接断开时应该通过 `onclose` 事件来关闭连接。而当服务端重新上线时，需要清除之前存储的数据，否则会导致所有发送到服务端的请求被视为离线。
- 为了处理这种异常情况，通常会采用重连方案。一种常见的方法是使用 JavaScript 库来处理重连，例如引入 `reconnecting-websocket.min.js` 库，然后使用该库的 API 方法来建立 WebSocket 连接。

```
var ws = new ReconnectingWebSocket(url); // 使用 ReconnectingWebSocket 类创建一个 WebSocket 连接，并传入连接的 URL
// 断线重连:
reconnectSocket(){ // 定义一个重连方法
    if ('ws' in window) { // 检查当前浏览器是否支持原生 WebSocket
        ws = new ReconnectingWebSocket(url); // 如果支持原生 WebSocket，则使用 ReconnectingWebSocket 类来创建 WebSocket 连接
    } else if ('MozWebSocket' in window) { // 如果当前浏览器支持 MozWebSocket
       ws = new MozWebSocket(url); // 则使用 MozWebSocket 类来创建 WebSocket 连接
    } else { // 如果当前浏览器都不支持 WebSocket
      ws = new SockJS(url); // 则使用 SockJS 来创建 WebSocket 连接
    }
}

```

断网监测支持使用js库：offline.min.js

```
onLineCheck(){ // 定义在线状态检测函数
    Offline.check(); // 调用 Offline 库的 check 方法来检测网络状态
    console.log(Offline.state,'---Offline.state'); // 打印当前网络状态到控制台
    console.log(this.socketStatus,'---this.socketStatus'); // 打印 WebSocket 连接状态到控制台

    if(!this.socketStatus){ // 如果 WebSocket 连接状态为断开
        console.log('网络连接已断开! '); // 打印提示信息：网络连接已断开
        if(Offline.state === 'up' && websocket.reconnectAttempts > websocket.maxReconnectInterval){ // 如果网络连接状态为恢复，并且重连尝试次数大于重连间
```

```
隔
            window.location.reload(); // 刷新页面
        }
        reconnectSocket(); // 调用重连函数来重新建立 WebSocket 连接
    }else{ // 如果 WebSocket 连接状态为连接中
        console.log('网络连接成功! '); // 打印提示信息：网络连接成功
        websocket.send("heartBeat"); // 发送心跳包给服务器以保持连接
    }
}

// 使用：在 WebSocket 断开连接时调用网络中断监测
websocket.onclose = () => { // 当 WebSocket 连接关闭时执行以下操作
    onLineCheck(); // 调用在线状态检测函数
};


```

面试常问问题

WebSocket 与 HTTP 有什么关系?

> WebSocket 和 HTTP 是两种不同的协议。两者都位于 OSI 模型的应用层，并且都依赖于传输层的 TCP 协议。 **虽然它们不同，但是 RFC 6455 中规定：** WebSocket 被设计为在 HTTP 80 和 443 端口上工作，并支持 HTTP 代理和中介，从而使其与 HTTP 协议兼容。 为了实现兼容性，WebSocket 握手使用 HTTP Upgrade 头，从 HTTP 协议更改为 WebSocket 协议。

协议类型	全双工、双向通信	单向请求-响应、无状态
用途	实时、交互式应用程序	网页、REST API
连接建立	需要握手	简单的请求-响应
消息格式	二进制或文本帧	文本格式（HTML、JSON、XML等）
开销	低	高（由于请求-响应头部）
持久连接	是	否（除非使用HTTP/2、长轮询等技术）
带宽效率	高	中等到高
使用情景	聊天应用、在线游戏、实时数据流	网页浏览、数据传输、API调用

WebSocket 与长轮询有什么区别?

> **长轮询**是一种技术，它的实现原理是客户端向服务器发送一个请求，服务器收到请求后不会立即响应，而是暂时挂起请求。服务器会持续检查请求的资源是否有更新，一旦有新数据可用，服务器就会立即响应请求。如果没有新数据可用，服务器会等待一定时间后再返回响应。长轮询的本质仍然基于 HTTP 协议，它使用了 HTTP 的请求-响应模式。
>
> 相比之下，**WebSocket**则提供了全双工的通信通道，建立连接后可以实现双向实时通信。一旦握手成功，WebSocket 就变成了一个持久的 TCP 连接，允许服务器和客户端之间随时发送数据，而不需要等待客户端的请求。

实时性	非常高	较低
性能开销	低	较高
适用场景	实时通信，双向通信	实时通知，较低实时性需求
浏览器支持	现代浏览器支持	几乎所有浏览器支持

Socket 是什么?

网络上的两个程序可以通过一个双向的通信连接来进行数据的交换。这个连接的一端被称为一个 Socket（套接字），因此建立网络通信连接至少需要一对端口号。

Socket 的本质是对 TCP/IP 协议栈的封装，它提供了一个针对 TCP 或者 UDP 编程的接口，而不是另一种独立的通信协议。通过 Socket，开发人员可以方便地使用 TCP/IP 协议进行网络通信。

关于 Socket，可以总结如下：

1. 它提供了底层通信的实现，几乎所有的应用层通信都是通过 Socket 进行的。
2. Socket 封装了 TCP/IP 协议，作为应用层协议的中间抽象层，为应用层提供了便捷的通信接口。
3. 在 TCP/IP 协议族中，传输层存在两种通用协议：TCP 和 UDP，因此根据不同的需求，可以选择使用不同参数的 Socket 来实现通信。