WebSocket 协议的实现

一、实验介绍

1.1 实验内容

这本实验中，我们将实现一个简陋版本的 WebSocket 协议。

1.2 实验知识点

- Go 的基本语法和使用
- WebSocket 协议的握手过程
- WebSocket 协议数据帧
- WebSocket 协议中的一些算法
- 掩码处理
- WebSocket 链接的实现

1.3 实验环境

- Go 1.2.1
- Xfce终端

1.4 适合人群

本课程难度为较难，属于中级级别课程，适合具有GO基础的用户，熟悉Go基础知识加深巩固，并且加深对于计算机网络的认识与理解。

1.5 代码获取

该实验的所有代码可以在终端中输入如下的代码来获取。

```
$ wget http://labfile.oss.aliyuncs.com/courses/510/websocket.tgz
```

二、实验原理

WebSocket 协议分析

WebSocket 协议解决了浏览器和服务器之间的全双工通信问题。在WebSocket出现之前，浏览器如果需要从服务器及时获得更新，则需要不停的对服务器主动发起请求，也就是 Web 中常用的 `poll` 技术。这样的操作非常低效，这是因为每发起一次新的 HTTP 请求，就需要单独开启一个新的 TCP 链接，同时 HTTP 协议本身也是一种开销非常大的协议。为了解决这些问题，所以出现了 WebSocket 协议。WebSocket 使得浏览器和服务器之间能通过一个持久的 TCP 链接就能完成数据的双向通信。关于 WebSocket 的 RFC 提案，可以参看 RFC6455 (http://tools.ietf.org/html/rfc6455)。

WebSocket 和 HTTP 协议一般情况下都工作在浏览器中，但 WebSocket 是一种完全不同于 HTTP 的协议。尽管，浏览器需要通过 HTTP 协议的 `GET` 请求，将 HTTP 协议升级为 WebSocket 协议。升级的过程被称为 `握手(handshake)`。当浏览器和服务器成功握手后，则可以开始根据 WebSocket 定义的通信帧格式开始通信了。像其他各种协议一样，WebSocket 协议的通信帧也分为控制数据帧和普通数据帧，前者用于控制 WebSocket 链接状态，后者用于承载数据。下面我们将一一分析 WebSocket 协议的握手过程以及通信帧格式。

2.1 WebSocket 协议的握手过程

握手的过程也就是将 HTTP 协议升级为 WebSocket 协议的过程。前面我们说过，握手开始首先由浏览器端发送一个 `GET` 请求开发，该请求的 HTTP 头部信息如下:

```
GET /chat HTTP/1.1
Host: server.example.com
Upgrade: websocket
Connection: Upgrade
Sec-WebSocket-Key: dGhlIHNhbXBsZSBub25jZQ==
Origin: http://example.com
Sec-WebSocket-Protocol: chat, superchat
Sec-WebSocket-Version: 13
```

当服务器端，成功验证了以上信息后，则会返回一个形如以下信息的响应:

```
HTTP/1.1 101 Switching Protocols
Upgrade: websocket
Connection: Upgrade
Sec-WebSocket-Accept: s3pPLMBiTxaQ9kYGzzhZRbK+xOo=
Sec-WebSocket-Protocol: chat
```

可以看到，浏览器发送的 **HTTP** 请求中，增加了一些新的字段，其作用如下所示:

- `Upgrade` : 规定必需的字段，其值必需为 `websocket` , 如果不是则握手失败;
- `Connection` : 规定必需的字段，值必需为 `Upgrade` , 如果不是则握手失败;
- `Sec-WebSocket-Key` : 必需字段，一个随机的字符串;
- `Sec-WebSocket-Protocol` : 可选字段，可以用于标识应用层的协议;
- `Sec-WebSocket-Version` : 必需字段，代表了 **WebSocket** 协议版本，值必需是 `13` , 否则握手失败;

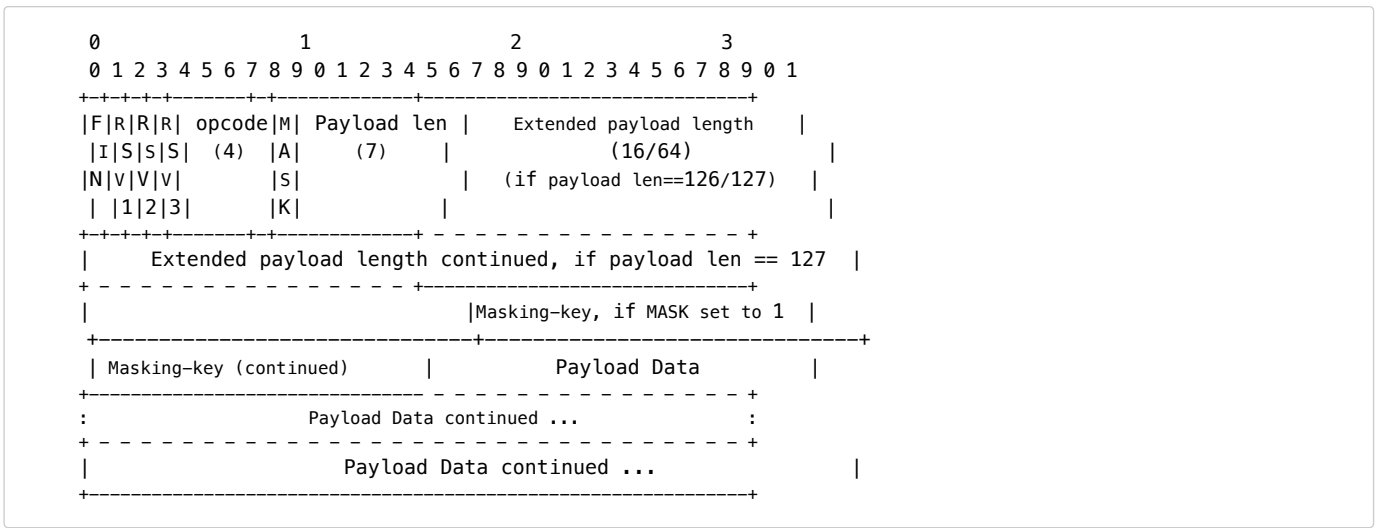
返回的响应中，如果握手成功会返回状态码为 `101` 的 **HTTP** 响应。同时其他字段说明如下:

- `Upgrade` : 规定必需的字段，其值必需为 `websocket` , 如果不是则握手失败;
- `Connection` : 规定必需的字段，值必需为 `Upgrade` , 如果不是则握手失败;
- `Sec-WebSocket-Accept` : 规定必需的字段，该字段的值是通过固定字符串 `258EAFA5-E914-47DA-95CA-C5AB0DC85B11` 加上请求中 `Sec-WebSocket-Key` 字段的值，然后再对其结果通过 **SHA1** 哈希算法求出的结果。
- `Sec-WebSocket-Protocol` : 对应于请求中的 `Sec-WebSocket-Protocol` 字段;

当浏览器和服务器端成功握手后，就可以传送数据了，传送数据是按照 **WebSocket** 协议的数据格式生成的。

2.2 WebSocket 协议数据帧

数据帧的定义类似于 **TCP/IP** 协议的格式定义，具体看下图:



以上这张图，一行代表 **32 bit (位)** ，也就是 **4 bytes**。总体上包含两份，帧头部和数据内容。每个从 **WebSocket** 链接中接收到的数据帧，都要按照以上格式进行解析，这样才能知道该数据帧是用于控制的还是用于传送数据的。关于以上数据帧的各个比特位的解释如下:

- `FIN` : **1 bit,** 当该比特位值为 **%x0** 时，表示后面还有更多的数据帧，**%x1** 时表示这是最后一个数据帧;
- `RSV1` , `RSV2` , `RSV3` : 各占 **1 bit,** 保留使用，一般情况下应该设置为 **%x0,** 用于
- `opcode` : **4 bit,** 用于表明数据帧的类型，一共可以表示 **16** 种帧类型，如下所示:
  - **%x0** : 表示这是一个分片的帧，它属于前面帧的后续帧;
  - **%x1** : 表示该数据帧携带的数据类型是文本类型，且编码是 **utf-8;**
  - **%x2** : 表示携带的是二进制数据;
  - **%x3-7** : 保留未使用;
  - **%x8** : 表示该帧用于关闭 **WebSocket** 链接;
  - **%x9** : 表示该帧代表了 `ping` 操作;
  - **%xA** : 表示该帧代表了 `pong` 回应;

- %xB-F : 保留未使用；

- `MASK` : 1 bit, %x0 表示数据帧没有经过掩码计算，而 %x1 则表示数据帧已经经过掩码计算，得到真正的数据需要解码。一般情况下，只有浏览器发送给服务器的数据帧才需要进行掩码计算；

- `Payload len` : 7 bit, 表示了数据帧携带的数据长度，7 bit 代表的值最大为 127, 按照 WebSocket 协议的规定， 这 7 bit 的值根据三种情况，帧的解析有所不同：

  - %x0 - 7D : 也就是从 0 到 125，表示数据长度, 数据总长度也就是 7 bit 代表的长度；
  - %x7E : 7 bit 的值是 126 时，则后续的 2 个字节（16 bit)表示的一个 16 位无符号数，这个数用来表示数据的长度；
  - %x7F : 7 bit 的值是 127 时，则后续的 8 个字节（64 bit)表示的一个 64 位无符号数，这个数用来表示数据的长度；

- `Masking-key` : 32 bit, 表示了用于解码的 key，只有当 `MASK` 比特位的值为 %x1 是，才有该数据；

- `Payload Data` : 余下的比特位用于存储具体的数据；

通过以上分析可以看出，WebSocket 协议数据帧的最大头部为 `2 + 8 + 4 = 14 bytes` 也就是 14 个字节。同时我们要实现 WebSocket 协议，最主要的工作就是实现对数据帧的解析。

2.3 WebSocket 协议中的一些算法

在分析 WebSocket 协议握手过程和数据帧格式过程中，我们讲到了一些算法，下面我们讲解下具体实现。

2.3.1 Sec-WebSocket-Accept 的计算方法

从上面的分析中，我们知道字段的值是通过固定字符串 `258EAFA5-E914-47DA-95CA-C5AB0DC85B11` 加上请求中 `Sec-WebSocket-Key` 字段的值，然后再对其结果通过 SHA1 哈希算法求出的结果。可以通过以下 golang 代码实现：

```
var keyGUID = []byte("258EAFA5-E914-47DA-95CA-C5AB0DC85B11")

func computeAcceptKey(challengeKey string) string {
    h := sha1.New()
    h.Write([]byte(challengeKey))
    h.Write(keyGUID)
    return base64.StdEncoding.EncodeToString(h.Sum(nil))
}
```

2.3.2 掩码处理

浏览器发送给服务器的数据帧是经过掩码处理的，那怎么样对数据进行解码呢？以下是来自 RFC6455 (http://tools.ietf.org/html/rfc6455) 文档的解释：

> The masking does not affect the length of the "Payload data". To
>
> convert masked data into unmasked data, or vice versa, the following
>
> algorithm is applied. The same algorithm applies regardless of the
>
> direction of the translation, e.g., the same steps are applied to
>
> mask the data as to unmask the data.
>
> Octet i of the transformed data ("transformed-octet-i") is the XOR of
>
> octet i of the original data ("original-octet-i") with octet at index
>
> i modulo 4 of the masking key ("masking-key-octet-j"):
>
> ```
> j                   = i MOD 4
> transformed-octet-i = original-octet-i XOR masking-key-octet-j
> ```
>
> The payload length, indicated in the framing as frame-payload-length,
>
> does NOT include the length of the masking key. It is the length of
>
> the "Payload data", e.g., the number of bytes following the masking
>
> key.

具体的流程是：将传输的数据按字节 byte 处理，同时将 `Masking-key` 代表的值也按字节处理。假如 `data-byte-i` 代表的是数据的第 `i` 个字节，那么 `j = i MOD 4` ，然后从 `Maksing-key` 中(一共有4个字节）取出第 `j` 个字节 `mask-key-byte-j` ，然后将 `data-byte-i` 和 `mask-key-byte-j` 代表的字节进行异或操作，取得结果就是最终的结果。该操作可以用如下 golang 代码实现：

```
func maskBytes(key [4]byte, pos int, b []byte) int {
    for i := range b {
        b[i] ^= key[pos&3]
        pos++
    }
    return pos & 3
}
```

注意以上的操作， pos & 3 这里代表的操作是 pos % 4，因为 a %（2^n）等价于 a &（2^n – 1），在这里之所以使用这种晦涩的方式是因为按位与操作更快。

三、开发准备

环境设置

首先，需要创建工作目录，同时设置 GOPATH 环境变量：

```
$ cd /home/shiyanlou/
$ mkdir -p golang/src
$ cd golang
$ export GOPATH=`/home/shiyanlou/golang`
```

以上步骤中，我们创建了 /home/shiyanlou/golang 目录，并将它设置为 GOPATH，也是后面实验的工作目录。

四、实验步骤

WebSocket 虽然是一个简单的协议，但是由于篇幅有限，所以我们需要做一些约定，只实现一个简陋版本。约定如下:

- 不支持数据分片；
- 只支持发送文本数据；
- 只实现用于服务器端的版本；

有了以上约定，就可以开始实现代码了。我们先从最底层开始实现，也就是说我们先实现 WebSocket 数据接收发送功能，然后再实现 WebSocket 协议的握手过程。

4.1 WebSocket 链接的实现

我们可以将 WebSocket 封装成一个结构体，然后通过在这个结构体上绑定接收和发送数据的方法，这样基本的数据接收功能就实现。

```
onst (
        // 是否是最后一个数据帧
        finalBit = 1 << 7
        // 是否需要进行掩码处理
        maskBit = 1 << 7

        // 文本数据帧类型
        TextMessage = 1
        // 关闭数据帧类型
        CloseMessage = 8
)

// WebSocket 链接
type Conn struct {
        writeBuf []byte
        maskKey  [4]byte

        conn net.Conn
}
```

以上代码中，先定义了一些常量，这些常量用于表示 FIN，MASK 比特位，同时由于我们实现的版本只支持文本协议，所以这里我们只定义了 TextMessage 和 CloseMessage 类型。接着定义了一个 Conn 结构体，该结构体表示一个 WebSocket 链接，其中最主要的字段是 conn，是一个 net.Conn 类型，也就是底层的 TCP 链接。

接着我们实现发送数据功能。由于我们实现的是用于服务器端版本的 WebSocket 协议，同时不支持分片操作，那这里的实现就非常简单了，具体代码如下:

```
// 发送数据，只支持发送文本数据，且不支持分片
func (c *Conn) SendData(data []byte) {
        length := len(data)
        c.writeBuf = make([]byte, 10+length)

        // 数据开始和结束的位置
        payloadStart := 2

        // 数据帧的第一个字节，不支持分片，且值能发送文本类型数据
        // 所以二进制位为 %b1000 0001
        // b0 := []byte{0x81}
        c.writeBuf[0] = byte(TextMessage) | finalBit

        // 数据帧第二个字节，服务器发送的数据不需要进行掩码处理
        switch {
        case length >= 65536:
                c.writeBuf[1] = byte(0x00) | 127
                binary.BigEndian.PutUint64(c.writeBuf[payloadStart:], uint64(length))
                // 需要 8 byte 来存储数据长度
                payloadStart += 8
        case length > 125:
                c.writeBuf[1] = byte(0x00) | 126
                binary.BigEndian.PutUint16(c.writeBuf[payloadStart:], uint16(length))
                // 需要 2 byte 来存储数据长度
                payloadStart += 2
        default:
                c.writeBuf[1] = byte(0x00) | byte(length)
        }
        copy(c.writeBuf[payloadStart:], data[:])
        c.conn.Write(c.writeBuf[:payloadStart+length])
}
```

以上代码的总体逻辑是，通过计算发送数据 `data` 的长度，来生成不同的协议头部，然后再进行数据发送。比如当数据长度 `125 < length < 65536` 时， `Payload len` 比特位的值应该为 `%x7E` ，这时候需要 2 字节来存储数据的真实长度。可以看到以上代码中，也未对数据帧进行掩码处理，这是因为从服务器端发送数据给客户端，不要求进行掩码处理。

接着我们就可以事先数据的接收功能了，具体代码如下：

```
// 读取数据
func (c *Conn) ReadData() (data []byte, err error) {

        var b [8]byte
        // 读取数据帧的前两个字节
        if _, err := c.conn.Read(b[:2]); err != nil {
                return nil, err
        }

        // 开始解析第一个字节, 是否还有后续数据帧
        final := b[0]&finalBit != 0
        // 不支持数据分片
        if !final {
                log.Println("Recived fragmented frame, not support")
                return nil, errors.New("not support fragmented message")
        }

        // 数据帧类型
        frameType := int(b[0] & 0xf)
        // 如果关闭类型，则关闭链接
        if frameType == CloseMessage {
                c.conn.Close()
                log.Println("Recived closed message, connection will be closed")
                return nil, errors.New("recived closed message")
        }
        if frameType != TextMessage {
                return nil, errors.New("only support text message")
        }
        // 检查数据帧是否被掩码处理
        mask := b[1]&maskBit != 0

        // 数据长度
        payloadLen := int64(b[1] & 0x7F)
        dataLen := int64(payloadLen)
        // 根据payload length 判断数据的真实长度
        switch payloadLen {
        case 126:
                if _, err := c.conn.Read(b[:2]); err != nil {
                        return nil, err
                }
                dataLen = int64(binary.BigEndian.Uint16(b[:2]))
        case 127:
                if _, err := c.conn.Read(b[:8]); err != nil {
                        return nil, err
                }
                dataLen = int64(binary.BigEndian.Uint64(b[:8]))
        }

        log.Printf("Read data length: %d, payload length %d", payloadLen, dataLen)
        // 读取 mask key
        if mask {
                        if _, err := c.conn.Read(c.maskKey[:]); err != nil {
                        return nil, err
                }
        }

        // 读取数据内容
        p := make([]byte, dataLen)
        if _, err := c.conn.Read(p); err != nil {
                return nil, err
        }
        if mask {
                maskBytes(c.maskKey, 0, p)
        }
        return p, nil
}
```

接收数据的过程也就是解析数据帧的过程。以上代码中，首先我们读取数据帧的前两个字节，因为从前面的协议分析中我们知道，前两个字节包含了数据帧的很多信息，比如这是不是最后一个数据帧，数据帧的类型，数据帧是否需要进行掩码处理，数据帧包含的数据长度。我们根据这些信息，对数据帧进行不同的处理。需要注意的是，由于我们只支持接收文本类型的数据帧，所以我们如果检测到其他类型的数据帧就直接错误返回了。

4.2 握手过程的实现

根据之前的协议分析，我知道握手的过程其实就是检查 HTTP 请求头部字段的过程，值得注意的一点就是需要针对客户端发送的 `Sec-WebSocket-Key` 生成一个正确的 `Sec-WebSocket-Accept` 只。关于生成的 `Sec-WebSocket-Accpet` 的实现，可以参考之前的分析。握手过程的具体代码如下：

```
// 将 http 链接升级到 websocket 链接
func Upgrade(w http.ResponseWriter, r *http.Request) (c *Conn, err error) {
	// 是否是 GET 方法
	if r.Method != "GET" {
		http.Error(w, http.StatusText(http.StatusMethodNotAllowed), http.StatusMethodNotAllowed)
		return nil, errors.New("websocket: method not GET")
	}
	// 检查 Sec-WebSocket-Version 版本
	if values := r.Header["Sec-Websocket-Version"]; len(values) == 0 || values[0] != "13" {
		http.Error(w, http.StatusText(http.StatusBadRequest), http.StatusBadRequest)
		return nil, errors.New("websocket: version != 13")
	}

	// 检查 Connection 和 Upgrade
	if !tokenListContainsValue(r.Header, "Connection", "upgrade") {
		http.Error(w, http.StatusText(http.StatusBadRequest), http.StatusBadRequest)
		return nil, errors.New("websocket: could not find connection header with token 'upgrade'")
	}
	if !tokenListContainsValue(r.Header, "Upgrade", "websocket") {
		http.Error(w, http.StatusText(http.StatusBadRequest), http.StatusBadRequest)
		return nil, errors.New("websocket: could not find connection header with token 'websocket'")
	}

	// 计算 Sec-WebSocket-Accpet 的值
	challengeKey := r.Header.Get("Sec-Websocket-Key")
	if challengeKey == "" {
		http.Error(w, http.StatusText(http.StatusBadRequest), http.StatusBadRequest)
		return nil, errors.New("websocket: key missing or blank")
	}

	var (
		netConn net.Conn
		br      *bufio.Reader
	)

	h, ok := w.(http.Hijacker)
	if !ok {
		http.Error(w, http.StatusText(http.StatusInternalServerError), http.StatusInternalServerError)
		return nil, errors.New("websocket: response dose not implement http.Hijacker")
	}
	var rw *bufio.ReadWriter
	netConn, rw, err = h.Hijack()
	if err != nil {
		http.Error(w, http.StatusText(http.StatusInternalServerError), http.StatusInternalServerError)
		return nil, err
	}
	br = rw.Reader

	if br.Buffered() > 0 {
		netConn.Close()
		return nil, errors.New("websocket: client sent data before handshake is complete")
	}

	// 构造握手成功后返回的 response
	p := []byte{}
	p = append(p, "HTTP/1.1 101 Switching Protocols\r\nUpgrade: websocket\r\nConnection: Upgrade\r\nSec-WebS
ocket-Accept: "...)
	p = append(p, computeAcceptKey(challengeKey)...)
	p = append(p, "\r\n\r\n"...)

	if _, err = netConn.Write(p); err != nil {
		netConn.Close()
		return nil, err
	}
	log.Println("Upgrade http to websocket successfully")
	conn := newConn(netConn)
	return conn, nil
}
```

握手过程的代码比较直观，就不多做解释了。

到这里 **WebSocket** 的实现就基本完成了，可以看到有了之前的各种约定，我们实现 **WebSocket** 协议也是比较简单的。

4.3 实验测试

为了测试我们实现的 **WebSocket** 协议，最简单的例子就是实现一个 `echo` 服务了：我们通过浏览器 **WebSocket** 链接将消息发送给服务器，然后服务器再通过 **WebSocket** 链接将消息返回回来。再次强调我们需要实现的 **WebSocket** 服务器。代码如下

动手实践是学习编程技术最有效的方式，可以开始实验

```
package main

import (
        "html/template"
        "log"
        "net/http"

        "websocket"
)

// WebSocket 处理器
func echo(w http.ResponseWriter, r *http.Request) {
        // 协议升级
        c, err := websocket.Upgrade(w, r)
        if err != nil {
                log.Print("Upgrade error:", err)
                return
        }
        defer c.Close()
        // 循环处理数据，接收数据，然后返回
        for {
                message, err := c.ReadData()
                if err != nil {
                        log.Println("read:", err)
                        break
                }
                log.Printf("recv: %s", message)
                c.SendData(message)
        }
}

// index 页面处理器
func home(w http.ResponseWriter, r *http.Request) {
        homeTemplate.Execute(w, "ws://"+r.Host+"/echo")
}

func main() {
        log.SetFlags(0)
        // 注册 handler
        http.HandleFunc("/echo", echo)
        http.HandleFunc("/", home)
        log.Fatal(http.ListenAndServe("0.0.0.0:8080", nil))
}

// index 页面内容
var homeTemplate = template.Must(template.New("").Parse(`
// 省略了页面内容...
`))
```

以上代码中我们省略了页面模板内容。可以看到以上代码中，我们实现了两个处理器， `home` 处理器处理 对 `/` 的访问， `echo` 处理器实现了 **WebSocket** 访问。

4.4 运行测试

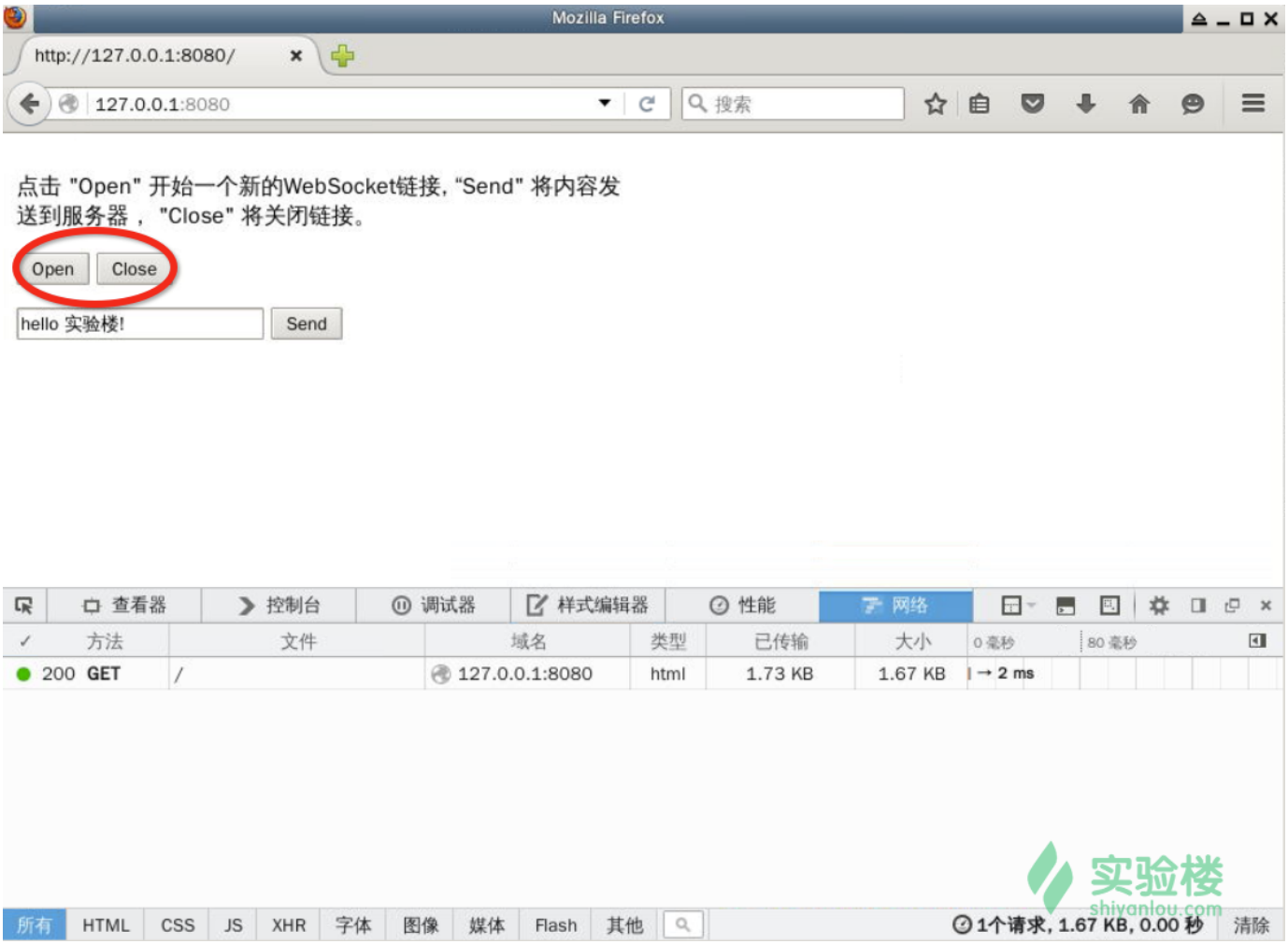
首先需要安装本实验的开始说明设置 **golang** 的开发环境，如果你还未设置，可以按照以下步骤设置：

```
$ cd /home/shiyanlou/
$ mkdir -p golang/src
$ cd golang
$ export GOPATH=`/home/shiyanlou/golang
```

然后通过 源码地址 **(http://labfile.oss.aliyuncs.com/courses/510/websocket.tgz)** 下载本实验代码, 并运行示例 ， 过程如下：

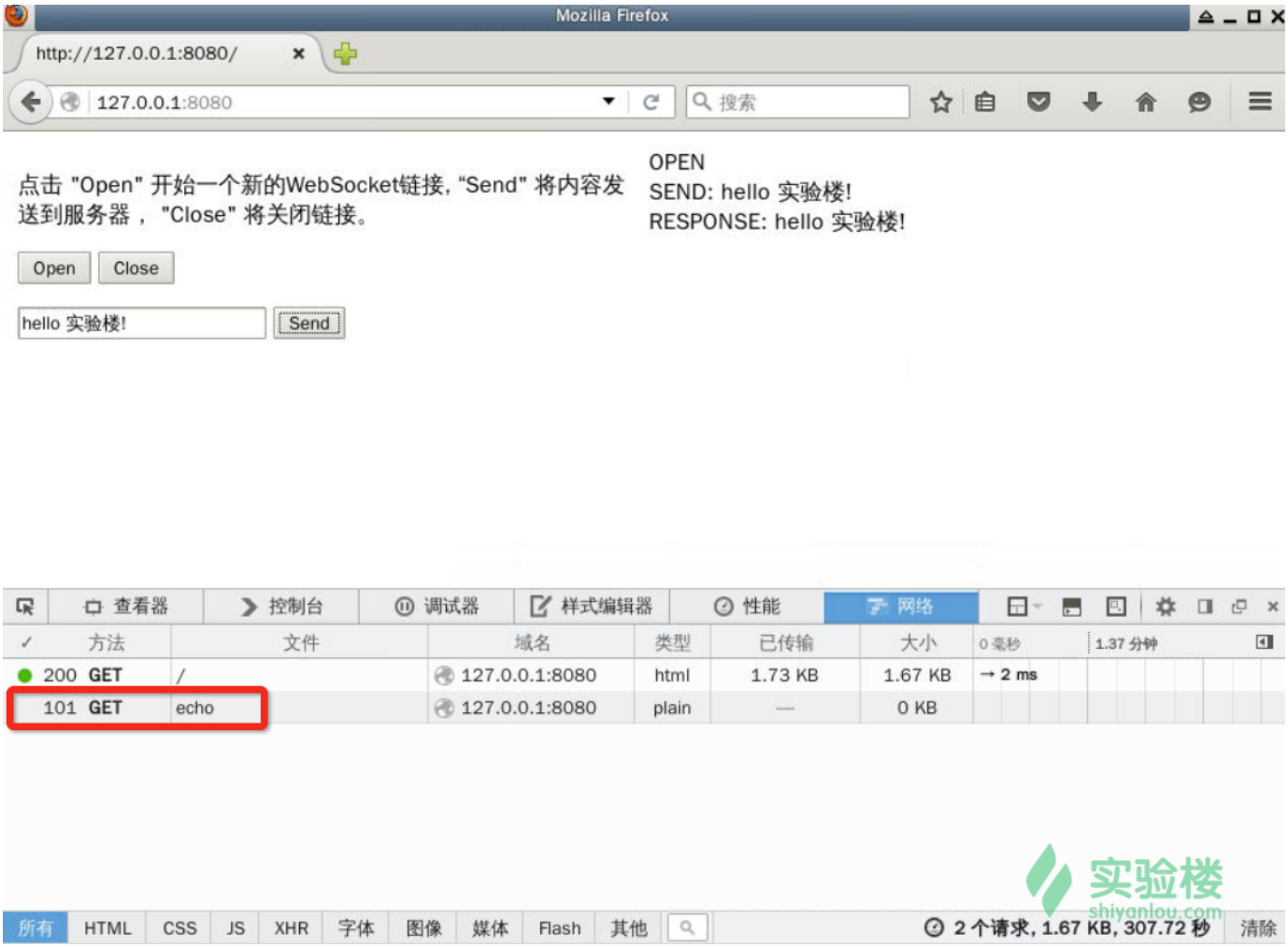
```
$ wget http://labfile.oss.aliyuncs.com/courses/510/websocket.tgz
$ tar xvf websocket.tgz
$ mv websocket /home/shiyanlou/golang/src/
$ cd /home/shiyanlou/golang/src/websocket/examples
$ go run echo.go
```

现在 `echo` 服务已经运行起来了，通过浏览器访问 `http://127.0.0.1:8080` 可以看到以下效果：



然后通过 Open 按钮开启 WebSocket 通信，接着在输入框内输入字符，点击 Send 就可以将内容发送到服务器，服务器再讲这些内容返回回来，具体效果如下图：

动手实践是学习 IT 技术最有效的方式！ 开始实验



五、实验总结

至此，本课程到这里就结束了。我们实现的 WebSocket 协议没有实现分片处理功能， 你能实现它吗？如果感觉有难度可以阅读下 Gorilla 实现的版本，项目地址为 https://github.com/gorilla/websocket。 (https://github.com/gorilla/websocket。)

课程教师

aiden0z

共发布过**6**门课程

查看老师的所有课程 > (/teacher/5348)

动手做实验，轻松学IT

(http://weibo.com/shiyanlou2013)

公司

关于我们 (/aboutus)
联系我们 (/contact)
加入我们 (http://www.simplecloud.cn/jobs.html)
技术博客 (https://blog.shiyanlou.com)

合作

我要投稿 (/contribute)
教师合作 (/labs)
高校合作 (/edu/)
友情链接 (/friends)
开发者 (/developer)

服务

企业版 (/saas)
实战训练营 (/bootcamp/)
会员服务 (/vip)
实验报告 (/courses/reports)

学习路径

Python学习路径 (/paths/python)
Linux学习路径 (/paths/linuxdev)
大数据学习路径 (/paths/bigdata)
Java学习路径 (/paths/java)

动手实践是学习 IT 技术最有效的方式！ 开始实验

常见问题 (/questions/?tag=%E5%B8%B8%E8%A7%81%E9%97%AE%E9%A2%98)
隐私条款 (/privacy)
PHP学习路径 (/paths/php)
全部 (/paths/)