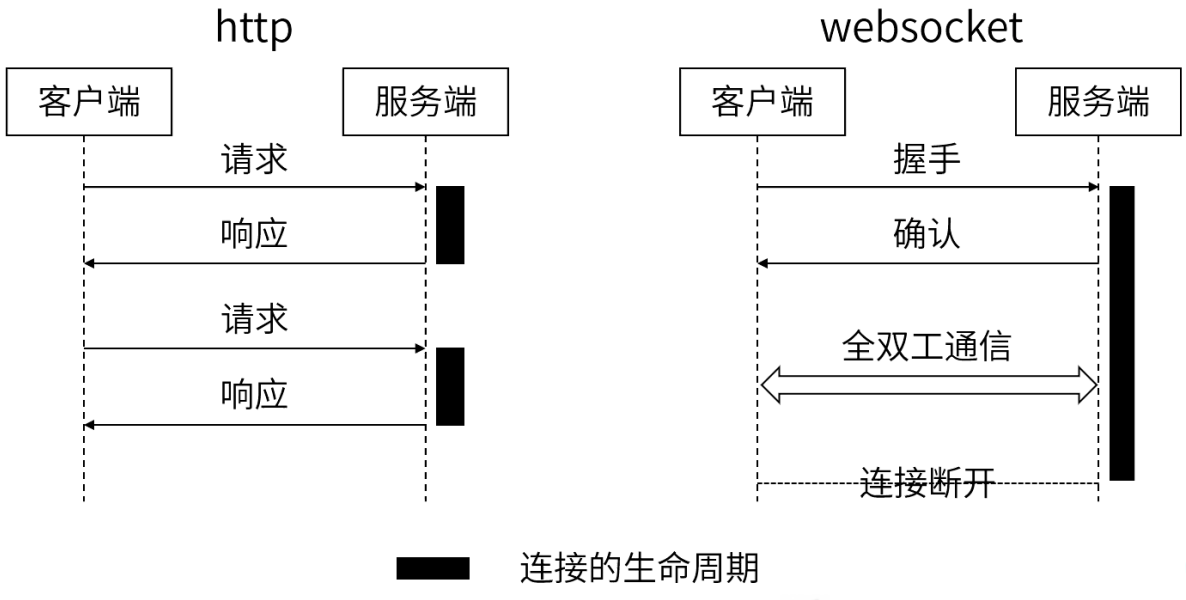
WebSocket

协议



HTTP协议，必须由客户端主动发起请求，服务器端收到请求后被动响应请求返回信息给客户端，而且底层建立的TCP连接用完即断，并不维持很久。但有些场景下，例如服务器端需要为某些客户端主动发送请求，HTTP协议做不到，解决方案只能定时发起请求来轮询服务器，效率低下。例如客户端需要快速刷新数据，依然需要连续地和服务器端建立连接，HTTP协议会频繁建立、断开TCP连接，成本很高。
HTTP每一次发起请求的请求报文或响应的响应报文，都需要携带请求头或响应头，两端通信也存在很多冗余数据。

- 应用层协议，底层采用TCP协议
- 因为需要在浏览器中使用，使用HTTP协议完成一次握手
- 全双工通信通道

应用场景

- 聊天室
- 在线协同编辑
- 实时数据更新
- 弹幕
- 股票等数据实时报价

服务器端实现

Gorilla WebSocket，Go实现的快速且应用较广泛的WebSocket库。

参考：https://github.com/gorilla/websocket/tree/master/examples/echo

server参考：https://github.com/gorilla/websocket/blob/master/examples/echo/server.go

下例，先使用http库完成WEB Server实现

```go
package main

import (
    "fmt"
    "net/http"

    "github.com/gorilla/websocket"
)

// 参考WEB服务器实现
var html = `<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <title>magedu</title>
</head>
<body>
    <h1>马哥教育www.magedu.com -- http库</h1><br>
    WebSocket测试 ws://127.0.0.1:9999/wsecho
</body>
</html>`

var upgrader = websocket.Upgrader{} // 使用缺省选项

func home(w http.ResponseWriter, request *http.Request) {
    //
    fmt.Printf("请求=%v\n", request)
    w.Header().Add("X-Server", "magedu.com") // 响应头
    w.Write([]byte(html))                    // 响应的内容
}

func wsecho(w http.ResponseWriter, request *http.Request) {
    // TODO
}

func main() {
    // URL映射到handler，handler函数2个参数一进一出
    http.HandleFunc("/", home)               // HTTP协议处理
    http.HandleFunc("/", wsecho)             // ws协议处理
    http.ListenAndServe("0.0.0.0:9999", nil) // http协议监听9999端口
}
```

由于HTML内容较为复杂，这里使用文件IO读取一个HTML文件。

index.html，网页浏览器端代码参考MDN https://developer.mozilla.org/zh-CN/docs/Web/API/WebSocket

```html
<!DOCTYPE html>
<html lang="en">
  <head>
    <meta charset="UTF-8" />
    <meta http-equiv="X-UA-Compatible" content="IE=edge" />
    <meta name="viewport" content="width=device-width, initial-scale=1.0" />
    <title>magedu</title>
  </head>
```

```
  <body>
    <h1>马哥教育www.magedu.com -- http库、WebSocket使用</h1>
    <br />
    WebSocket测试 ws://127.0.0.1:9999/wsecho
  </body>
</html>
<script>
  // 利用http协议先握手，请求头中有Sec-Websocket-Key:[1g4Tvo2jzXGnsGDgN0JdEA==]
Sec-Websocket-Version:[13] Upgrade:[websocket]
  const ws = new WebSocket("ws://127.0.0.1:9999/wsecho");

  // Connection opened
  ws.addEventListener("open", function (event) {
    ws.send("Hello Server!");
  });

  // Listen for messages
  ws.addEventListener("message", function (event) {
    console.log("Message from server ", event.data);
  });

  //
  ws.addEventListener("error", (event) => {
    console.log("连接错误: ", event);
  });

  // 连接关闭回调
  ws.onclose = (event) => {
    console.log("连接关闭");
  };
</script>
```

```
package main

import (
    "fmt"
    "log"
    "net/http"
    "os"

    "github.com/gorilla/websocket"
)

var upgrader = websocket.Upgrader{} // 使用缺省选项

func home(w http.ResponseWriter, request *http.Request) {
    // fmt.Printf("请求=%v\n", request)
    w.Header().Add("X-Server", "magedu.com") // 响应头
    http.ServeFile(w, request, "index.html") // 响应的内容
}

func wsecho(w http.ResponseWriter, request *http.Request) {
    // fmt.Printf("请求=%v\n", request)
```

```
    //
https://github.com/gorilla/websocket/blob/master/examples/echo/server.go/ech
o
    c, err := upgrader.Upgrade(w, request, nil)
    if err != nil {
        log.Print("upgrade:", err)
        return
    }
    defer c.Close()
    for {
        mt, message, err := c.ReadMessage()
        if err != nil {
            log.Println("read:", err)
            break
        }
        log.Printf("recv: %s", message)
        err = c.WriteMessage(mt, message)
        if err != nil {
            log.Println("write:", err)
            break
        }
    }
}

func main() {
    // URL映射到handler，handler函数2个参数一进一出
    http.HandleFunc("/", home)                  // HTTP协议处理
    http.HandleFunc("/wsecho", wsecho)          // ws协议处理
    http.ListenAndServe("0.0.0.0:9999", nil) // http协议监听9999端口
}
```

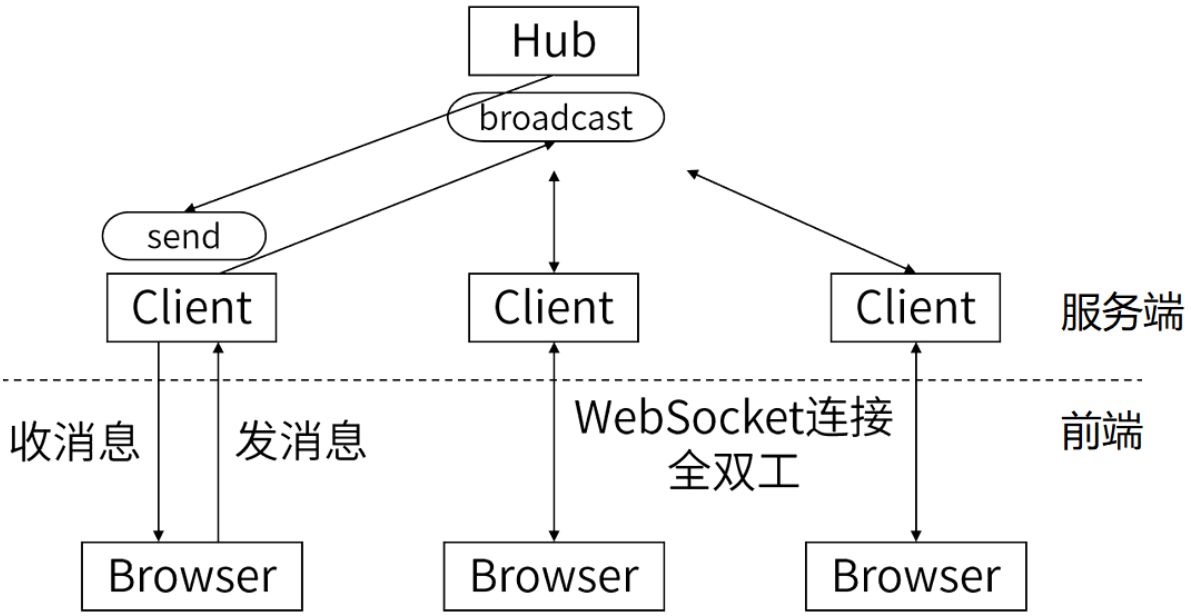
客户端实现

客户端参考：https://github.com/gorilla/websocket/blob/master/examples/echo/client.go

客户端代码大家可以参考以上链接，但多数情况下，更多是使用浏览器作为客户端。

聊天室实现

https://github.com/gorilla/websocket/tree/master/examples/chat




```go
// https://github.com/gorilla/websocket/blob/master/examples/chat/hub.go
package main

// Hub就是为了广播消息给每一个客户端，因此要维护着一个客户端集合
type Hub struct {
    clients map[*Client]bool // 记录客户端的容器

    // Inbound messages from the clients.
    broadcast chan []byte

    // Register requests from the clients.
    register chan *Client

    // Unregister requests from clients.
    unregister chan *Client
}

func newHub() *Hub {
    return &Hub{
        broadcast:  make(chan []byte),
        register:   make(chan *Client),
        unregister: make(chan *Client),
        clients:    make(map[*Client]bool),
    }
}

func (h *Hub) run() {
    for { // 死循环
        select { // 监听注册、注销、消息通道
        case client := <-h.register:
            h.clients[client] = true // 注册到map中
        case client := <-h.unregister:
            if _, ok := h.clients[client]; ok {
                delete(h.clients, client) // 从map中删除
                close(client.send)
            }
        case message := <-h.broadcast: // 如果有消息，遍历所有客户端
            for client := range h.clients {
```

```
                select {
                case client.send <- message:
                default:
                    close(client.send)
                    delete(h.clients, client)
                }
            }
        }
    }
}
```

```go
// https://github.com/gorilla/websocket/blob/master/examples/chat/client.go

// Client is a middleman between the websocket connection and the hub.
type Client struct {
    hub *Hub

    // The websocket connection.
    conn *websocket.Conn

    // Buffered channel of outbound messages.
    send chan []byte
}


// 每一个持久连接的客户端到来就调用serveWs
func serveWs(hub *Hub, w http.ResponseWriter, r *http.Request) {
    conn, err := upgrader.Upgrade(w, r, nil)
    if err != nil {
        log.Println(err)
        return
    }
    // 为当前连接创建新的客户端结构体，send是一个容量256的通道
    client := &Client{hub: hub, conn: conn, send: make(chan []byte, 256)}
    client.hub.register <- client // 注册

    // Allow collection of memory referenced by the caller by doing all
work in
    // new goroutines.
    // 为每一个客户端读写创建单独的协程
    go client.writePump()
    go client.readPump()
}

// readPump pumps messages from the websocket connection to the hub.
// 读取泵从连接上获取到消息把它打入hub中
// The application runs readPump in a per-connection goroutine. The
application
// ensures that there is at most one reader on a connection by executing
all
// reads from this goroutine.
func (c *Client) readPump() {
    defer func() {
        c.hub.unregister <- c
        c.conn.Close()
```

```
    }()
    c.conn.SetReadLimit(maxMessageSize)
    c.conn.SetReadDeadline(time.Now().Add(pongWait))
    c.conn.SetPongHandler(func(string) error { c.conn.SetReadDeadline(time.Now().Add(pongWait)); return nil })
    for {
        _, message, err := c.conn.ReadMessage()
        if err != nil {
            if websocket.IsUnexpectedCloseError(err, websocket.CloseGoingAway, websocket.CloseAbnormalClosure) {
                log.Printf("error: %v", err)
            }
            break
        }
        message = bytes.TrimSpace(bytes.Replace(message, newline, space, -1))
        c.hub.broadcast <- message // 打到hub中让hub遍历map
    }
}

// writePump pumps messages from the hub to the websocket connection.
// 写入泵从hub中把消息打入到连接中
// A goroutine running writePump is started for each connection. The
// application ensures that there is at most one writer to a connection by
// executing all writes from this goroutine.
func (c *Client) writePump() {
    ticker := time.NewTicker(pingPeriod) // 定时器定时ping等pong回来
    defer func() {
        ticker.Stop()
        c.conn.Close()
    }()
    for {
        select {
        case message, ok := <-c.send:
            c.conn.SetWriteDeadline(time.Now().Add(writeWait))
            if !ok {
                // The hub closed the channel.
                c.conn.WriteMessage(websocket.CloseMessage, []byte{})
                return
            }

            w, err := c.conn.NextWriter(websocket.TextMessage)
            if err != nil {
                return
            }
            w.Write(message)

            // Add queued chat messages to the current websocket message.
            n := len(c.send)
            for i := 0; i < n; i++ {
                w.Write(newline)
                w.Write(<-c.send)
            }

            if err := w.Close(); err != nil {
```

```
                return
            }
        case <-ticker.C:
            c.conn.SetWriteDeadline(time.Now().Add(writeWait))
            if err := c.conn.WriteMessage(websocket.PingMessage, nil); err
 != nil {
                return
            }
        }
    }
}
```

```
// https://github.com/gorilla/websocket/blob/master/examples/chat/main.go
package main

import (
    "flag"
    "log"
    "net/http"
)

var addr = flag.String("addr", ":8080", "http service address")

func serveHome(w http.ResponseWriter, r *http.Request) {
    log.Println(r.URL)
    if r.URL.Path != "/" {
        http.Error(w, "Not found", http.StatusNotFound)
        return
    }
    if r.Method != http.MethodGet {
        http.Error(w, "Method not allowed", http.StatusMethodNotAllowed)
        return
    }
    http.ServeFile(w, r, "home.html")
}

func main() {
    flag.Parse()
    hub := newHub()
    go hub.run() // 启动hub
    http.HandleFunc("/", serveHome)
    http.HandleFunc("/ws", func(w http.ResponseWriter, r *http.Request) {
        serveWs(hub, w, r)
    })
    err := http.ListenAndServe(*addr, nil)
    if err != nil {
        log.Fatal("ListenAndServe: ", err)
    }
}
```

TLS

很多应用层协议都采用明文传输，之前我们编写WEB服务器的时候，就是用明文和浏览器传输数据。

安全套阶层协议（SecureSockets Layer， SSL）是有网景公司研发，用于增强因特网通信安全的协议。

传输安全层协议（Transfer Layer Security，TLS）在SSL v3.0基础上，提供了一些增强功能，两者差异很小。

浏览器端和服务器端通信，如果使用明文，在某些场景下，数据不能保证安全，所以，通信数据在必要时需要加密。

浏览器端和Server端

- 采用对称加密
  - 如果所有浏览器和Server采用同一个密码，根本没有安全性可言
  - 如果浏览器A和Server可以采用一个密码，浏览器B和Server采用另外一个密码，不同浏览器和Server之间采用不同的密码。这样做，增加了Server的密码管理负担，且密码较容易破解
  - 加密效率较好，密文传输，但容易破解
- 采用非对称加密RSA密钥系统
  - 服务器端产生公钥、私钥
  - 如果公钥对数据加密，就只能使用私钥才能解密；如果使用私钥对数据加密，也只能使用公钥解密
  - 把服务器端公钥发给浏览器端，让浏览器端使用公钥加密、解密数据
  - 为了提高效率，可以在通信中，使用对称加密

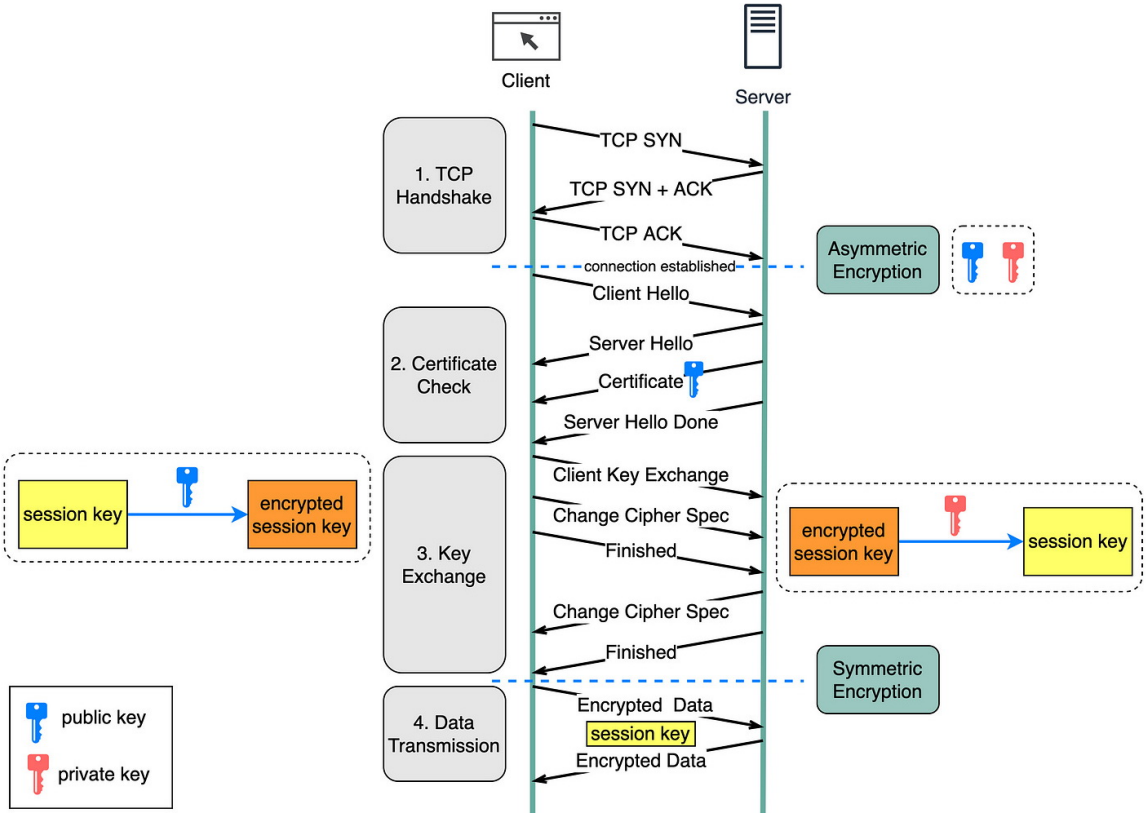
由此可见，在浏览器端和Server端采用非对称加密方式结合对称加密是较好的方式。

问题是，为什么信任服务器端的公钥？这就需要一个可信的第三方CA（Certificate Authority）机构，由它签发一个对该服务器的可信证书，里面有该服务器端的域名、证书有效期、颁发机构、公钥等信息，以及可信的签名来防止篡改信息。签名本质上就是一串hash值。

问题又来了，如何信任这个第三方CA机构？计算机系统中预安装了一些第三方机构根证书，从而可以验证证书是否可信。也可以本地下载手动强行确认安装不可信的证书，后果自负。

HTTPS协议

可以认为HTTPS是在应用层和传输层之间加入了一层，用来解决应用层数据加密、解密传输。



1. HTTP协议基于TCP协议，先建立TCP连接，需要3次握手
2. 客户端发送client hello到服务器端
    1. hello信息包括最后的TLS版本、支持的加密算法套件等
    2. 服务端响应server hello给客户端，回复它支持的加密算法和TLS版本
    3. 接着服务端在发送SSL证书到浏览器端，浏览器端校验证书是否可信
    4. 服务端发送server hello done
3. 校验证书通过后，浏览器端使用公钥加密一个新产生的会话key，和确认的加密算法及版本的信息一起发给服务端。服务端收到加密后的session key的密文使用自己私钥解密得到session key
4. 接下来，两端就使用这个session key来对称加密交换的数据。session key每次会话都不一样，就当前会话用

```
1 http.ListenAndServeTLS(addr, certFile, keyFile string, handler Handler) error
  {}
```

- certFile：服务器端证书文件
    - 可以付费或免费申请证书
    - 本地证书，但不可信，所以要在浏览器端手动安装证书
- keyFile：与证书相关的私钥文件