HTTP

- `http` 是我们前后台交互的时候的传输协议（即超文本传输协议）

HTTP 的工作流程

1. 和服务器建立链接
2. 建立链接后，发送一个请求给服务器（请求）
3. 服务器接受到请求以后进行相应的处理并给出一个回应（响应）
4. 断开于服务器的链接

和服务器建立链接

- 怎么和服务器建立链接呢？
- 需要保证客户端的接受和发送正常，服务器端的接受和发送正常
- 这里就涉及到一个东西叫做 `TCP/IP` 协议
- 建立链接的主要步骤叫做 `三次握手`

i. 客户端发送一个消息给到服务端

```
1. 此时：
2.  服务端知道了 客户端可以正常发送消息
3.  服务端知道了 服务端可以正常接受消息
```

ii. 服务端回给客户端一个消息

```
1. 此时：
2.  服务端知道了 客户端可以正常发送消息
3.  服务端知道了 服务端可以正常接受消息
4.  客户端知道了 客户端可以正常发送消息
5.  客户端知道了 客户端可以正常接受消息
6.  客户端知道了 服务端可以正常接受消息
7.  客户端知道了 服务端可以正常发送消息
```

iii. 客户端再回给服务端一个消息

```
1. 此时：
2.  服务端知道了 客户端可以正常发送消息
```

```
服务端知道了 服务端可以正常接受消息
客户端知道了 客户端可以正常发送消息
客户端知道了 客户端可以正常接受消息
客户端知道了 服务端可以正常接受消息
客户端知道了 服务端可以正常发送消息
服务端知道了 服务端可以正常发送消息
服务端知道了 客户端可以正常接受消息
```

- 至此，依照 `TCP/IP` 协议的建立链接就建立好了
- 双方都知道双方可以正常收发消息
- 就可以进入到第二步，通讯了

发送一个请求

- 建立完链接以后就是发送请求的过程
- 我们的每一个请求都要把我们的所有信息都包含请求
- 每一个请求都会有一个 `请求报文`
- 在 `请求报文` 中会包含我们所有的请求信息（也就是我们要和服务端说的话都在里面）
- 我们的请求报文中会包含几个东西

i. 请求行

```
POST /user HTTP/1.1
# POST 请求方式
# /user 请求URL（不包含域名）
# HTTP/1.1 请求协议版本
```

ii. 请求头（请求头都是键值对的形式出现的）

```
user-agent: Mozilla/5.0 # 产生请求的浏览器信息
accept: application/json # 表示客户端希望接受的数据类型
Content-Type: application/x-www-form-urlencoded # 客户端发送的实体数据格式
Host: 127.0.0.1 # 请求的主机名（IP）
```

iii. 请求空行（请求头和请求主体之间要留一个空白行）

```
# 就是一个空行
```

iv. 请求体（本次请求携带的数据）

iv. 请求体（本次请求携带的数据）

```
# GET 请求是没有请求体数据的
# POST 请求才有请求体数据
```

- 接下来看一个完整的请求报文

```
POST /user HTTP/1.1 # 请求行
Host: www.user.com
Content-Type: application/x-www-form-urlencoded
accept: application/json
User-agent: Mozilla/5.0. # 以上是首部
#（此处必须有一空行） # 空行分割header和请求内容
name=world # 请求体
```

接受一个响应

- 客户端的请求发送到服务端以后
- 服务端进行对应的处理
- 会给我们返回一个响应
- 每一个响应都会有一个 `响应报文`
- 在 `响应报文` 中会包含我们所有的响应信息（也就是服务端在接受到客户端请求以后，给我们的回信）
- 我们的 `响应报文` 中会包含几个信息

i. 状态行

```
HTTP/1.1 200 OK
# HTTP/1.1 服务器使用的 HTTP 协议版本
# 200 响应状态码
# OK 对响应状态码的简单解释
```

ii. 响应头

```
Date: Jan, 14 Aug 2019 12:42:30 GMT # 服务器时间
Server: Apache/2.4.23 (Win32) OpenSSL/1.0.2j PHP/5.4.45 # 服务器类型
Content-Type: text/html # 服务端给客户端的数据类型
Content-Length: 11 # 服务端给客户端的数据长度
```

iii. 响应体

iii. 响应体

```
1. hello world
2. # 服务端给客户端的响应数据
```

断开于服务端的链接

- 之前我们的建立链接是基于 `TCP/IP` 协议的 `三次握手`
- 我们的断开链接是基于 `TCP/IP` 协议的 `四次挥手`
  i. 客户端发送一个我要断开的消息给服务端
  ii. 服务端接受到以后发送一个消息告诉客户端我已经进入关闭等待状态
  iii. 服务端再次发送一个消息告诉客户端，这个是我的最后一次消息给你，当我再接受到消息的时候就会关闭
  iv. 客户端接受到服务端的消息以后，告诉服务器，我已经关闭，这个是给你的最后一个消息

完成一个 HTTP 请求

- 至此，一个 HTTP 请求就完整的完成了
- 一个 HTTP 请求必须要包含的四个步骤就是
  i. 建立链接
  ii. 发送请求
  iii. 接受响应
  iv. 断开链接
- 在一个 HTTP 请求中，请求的部分有请求报文，接受响应的部分有响应报文
- 请求报文包含
  i. 请求行
  ii. 请求头
  iii. 请求空行
  iv. 请求体
- 响应报文
  i. 状态行
  ii. 响应头
  iii. 响应体

常见的 HTTP 响应状态码

- 在一个 HTTP 请求的响应报文中的状态行会有一个响应状态码
- 这个状态码是用来描述本次响应的状态的
- 通常会出现五种状态码
  i. 100 ~ 199
  ii. 200 ~ 299

iii. 300 ~ 399
iv. 400 ~ 499
v. 500 ~ 599

100 ~ 199

- 一般我们看不到，因为表示请求继续
- 100： 继续请求，前面的一部分内容服务端已经接受到了，正在等待后续内容
- 101： 请求者已经准备切换协议，服务器页表示同意

200 ~ 299

- 2 开头的都是表示成功，本次请求成功了，只不过不一样的状态码有不一样的含义（语义化）
- 200： 标准请求成功（一般表示服务端提供的是网页）
- 201： 创建成功（一般是注册的时候，表示新用户信息已经添加到数据库）
- 203： 表示服务器已经成功处理了请求，但是返回的信息可能来自另一源
- 204： 服务端已经成功处理了请求，但是没有任何数据返回

300 ~ 399

- 3 开头也是成功的一种，但是一般表示重定向
- 301： 永久重定向
- 302： 临时重定向
- 304： 使用的是缓存的数据
- 305： 使用代理

400 ~ 499

- 4 开头表示客户端出现错误了
- 400： 请求的语法服务端不认识
- 401： 未授权（你要登录的网站需要授权登录）
- 403： 服务器拒绝了你的请求
- 404： 服务器找不到你请求的 URL
- 407： 你的代理没有授权
- 408： 请求超时
- 410： 你请求的数据已经被服务端永久删除

500 ~ 599

- 5 开头的表示服务端出现了错误
- 500： 服务器内部错误
- 503： 服务器当前不可用（过载或者维护）
- 505： 请求的协议服务器不支持

常见的 HTTP 请求方式

- 每一个 HTTP 请求在请求行里面会有一个东西叫做请求方式
- 不同的请求方式代表的含义不同
  i. GET： 一般用于获取一些信息使用（获取列表）
  ii. POST： 一般用于发送一些数据给服务端（登录）
  iii. PUT： 一般用于发送一些数据给服务当让其添加新数据（注册）
  iv. DELETE： 一般用域删除某些数据
  v. HEAD： 类似于 GET 的请求，只不过一般没有响应的具体内容，用于获取报文头
  vi. CONNECT： HTTP/1.1 中预留的方式，一般用于管道链接改变为代理的时候使用
  vii. PATCH： 是和 PUT 方式类似的一个方式，一般用于更新局部数据
  viii. OPTIONS： 允许客户端查看服务端性能
- 我们比较常用的就是 GET 和 POST

GET 请求

- 参数以 `querystring` 的形式发送，也就是直接拼接在 请求路径的后面
- GET 请求会被浏览器主动缓存
- GET 请求根据不同的浏览器对长度是有限制的
  - IE： 2083 个字符
  - FireFox： 65536 个字符
  - Safari： 80000 个字符
  - Opera： 190000 个字符
  - Chrome： 8182 个字符
  - APACHE(server)： 理论上接受的最大长度是 8192 个字符（有待商榷）
- 对参数的类型有限制，只接受 ASCII 码的格式
- GET 请求是明文发送，相对不安全

POST 请求

- 参数以 `request body` 的形式发送，也就是放在请求体中
- POST 请求不会被浏览器主动缓存，除非手动设置
- POST 请求理论上是没有限制的，除非服务端做了限制
- 对参数类型没有限制，理论上可以传递任意数据类型，只不过要和请求头对应
- POST 请求是密文发送，相对安全

COOKIE

- `cookie` 是一个以字符串的形式存储数据的位置
- 每一个 HTTP 请求都会在请求头中携带 cookie 到服务端
- 每一个 HTTP 响应都会在响应头中携带 cookie 到客户端
- 也就是说，cookie 是不需要我们手动设置，就会自动在 客户端 和 服务端之间游走的数据
- 我们只是需要设置一下 cookie 的内容就可以

COOKIE 的存储形式

- cookie 是以字符串的形式存储，在字符串中以 `key=value` 的形式出现
- 每一个 `key=value` 是一条数据
- 多个数据之间以 `;` 分割

```
// cookie 的形态
'a=100; b=200; c=300;'
```

COOKIE 的特点

1. 存储大小有限制，一般是 4 KB 左右
2. 数量有限制，一般是 50 条左右
3. 有时效性，也就是有过期时间，一般是 **会话级别**（也就是浏览器关闭就过期了）
4. 有域名限制，也就是说谁设置的谁才能读取

使用方式

- 读取 cookie 的内容使用 `document.cookie`

```
const cookie = document.cookie
console.log(cookie) // 就能得到当前 cookie 的值
```

- 设置 cookie 的内容使用 `document.cookie`

```
// 设置一个时效性为会话级别的 cookie
document.cookie = 'a=100'

// 设置一个有过期时间的 cookie
```

```
document.cookie = 'b=200;expires=Thu, 18 Dec 2043 12:00:00 GMT";'
// 上面这个 cookie 数据会在 2043 年 12 月 18 日 12 点以后过期，过期后会自动消失
```

- 删除 cookie 的内容使用 `document.cookie`

```
// 因为 cookie 不能直接删除
// 所以我们只能把某一条 cookie 的过期时间设置成当前时间之前
// 那么浏览器就会自动删除 cookie
document.cookie = 'b=200;expires=Thu, 18 Dec 2018 12:00:00 GMT";'
```

COOKIE 操作封装

- 因为 js 中没有专门操作 COOKIE 增删改查的方法
- 所以需要我们自己封装一个方法

设置 cookie

```
/**
 * setCookie 用于设置 cookie
 * @param {STRING} key 要设置的 cookie 名称
 * @param {STRING} value 要设置的 cookie 内容
 * @param {NUMBER} expires 过期时间
 */
function setCookie (key, value, expires) {
  const time = new Date()
  time.setTime(time.getTime() - 1000 * 60 * 60 * 24 * 8 + expires) // 用于设置过期时间

  document.cookie = `${key}=${value};expires=${time};`
}
```

读取 cookie

```
/**
 * getCookie 获取 cookie 中的某一个属性
 * @param {STRING} key 你要查询的 cookie 属性
 * @return {STRING} 你要查询的那个 cookie 属性的值
 */
function getCookie(key) {
  const cookieArr = document.cookie.split(';')

  let
```

```
  let value =

  cookieArr.forEach(item => {
  if (item.split('=')[0] === key) {
  value = item.split('=')[1]
  }
  })

  return value
}
```

删除 cookie

```
/**
 * delCookie 删除 cookie 中的某一个属性
 * @param {STRING} name 你要删除的某一个 cookie 属性的名称
 */
function delCookie(name) {
 setCookie(name, 1, -1)
}
```