HTTP协议

在网络通信中，2台计算机之间发送和接收的数据都是二进制流，二进制流可理解为只包含0和1的一个长串。单纯的二进制流是没有意义的，除非人为的对其进行规定。比如一个有爱的路人甲定义了一套规则（规则甲），在这套规则里，`00`表示`我`，`11`表示`你`，`01`表示`爱`，那么当计算机A向计算机B发送一串二进制流`000111`，若A和B都遵循规则甲。那么B就知道，A对她说了`我爱你`。如果另一个心怀怨恨的路人乙定义了一套规则（规则乙），其中`00`表示`我`，`11`表示`你`，`01`表示`恨`，如果A和B的通信，遵循的是规则乙，那么B接收到上面的二进制流，就知道A对她说了`我恨你`。同样的二进制流，若采用不同的规则，解读出来的是完全不同的含义。所以要让网络中的计算机互相通信，需要有一套标准规则，或者协议。HTTP协议就是网络通信中最基本的一种协议。浏览器通过发起HTTP请求给服务器（传送二进制流给服务器，并告诉服务器这段二进制流采用的协议是HTTP），服务器收到请求后，用HTTP协议对二进制流进行解析，明白了浏览器需要访问的具体是哪一个页面后，找到对应的页面资源，根据HTTP协议转化为二进制流，再传送回给浏览器。这样就完成了一次网页的请求和响应。

在通信时，HTTP协议会区分请求者，和响应者。或者客户端，和服务端。由客户端，发送HTTP请求给服务端，告知需要的资源。由服务端，发送HTTP响应给客户端，返回客户端需要的资源。遵循HTTP协议的二进制流，称为HTTP报文。客户端向服务端发送的报文，称为**HTTP请求报文**。服务端向客户端发送的报文，称为**HTTP响应报文**。它们的格式分别定义如下。

HTTP请求报文

先来看一个实际的HTTP请求报文。打开FireFox浏览器，按下F12打开开发者工具，在地址栏输入`www.baidu.com`，敲下回车，能够看到发出了如下的HTTP请求。

```
GET / HTTP/1.1
Host: www.baidu.com
User-Agent: Mozilla/5.0 (Windows NT 10.0; Win64; x64; rv:84.0) Gecko/20100101
Firefox/84.0
Accept:
text/html,application/xhtml+xml,application/xml;q=0.9,image/webp,*/*;q=0.8
Accept-Language: zh-CN,zh;q=0.8,zh-TW;q=0.7,zh-HK;q=0.5,en-US;q=0.3,en;q=0.2
Accept-Encoding: gzip, deflate, br
Connection: keep-alive
Cookie: BAIDUID=BD904AF3599D963DAF4E36903E9D0561;
```

报文的第一行称为**请求行**，由3部分组成：**请求方法**，**请求的URL**（关于URL），**HTTP协议版本**。上面的请求行表示，请求的方法是`GET`，请求的URL是`/`，使用的协议是HTTP 1.1版本。随后的每一行，都是以键值对的形式出现，左侧是键，右侧是值，键值之间用冒号`:`分隔。这些键值对称为**请求头**，它主要描述关于此次请求的一些元数据。比如`Host: www.baidu.com`表示请求的域名是`www.baidu.com`，`User-Agent: Mozilla/5.0....`说明了浏览器的信息，这里表示使用的Mozilla的火狐浏览器，`Accept: text/html....`表示了此次请求预期接收的响应体的格式，这里表明预期接收到的是纯文本或者HTML格式的数据。

上面的请求报文不包含**请求体**，下面在JSP中提交表单数据，来发起一个POST请求，查看一下请求报文。

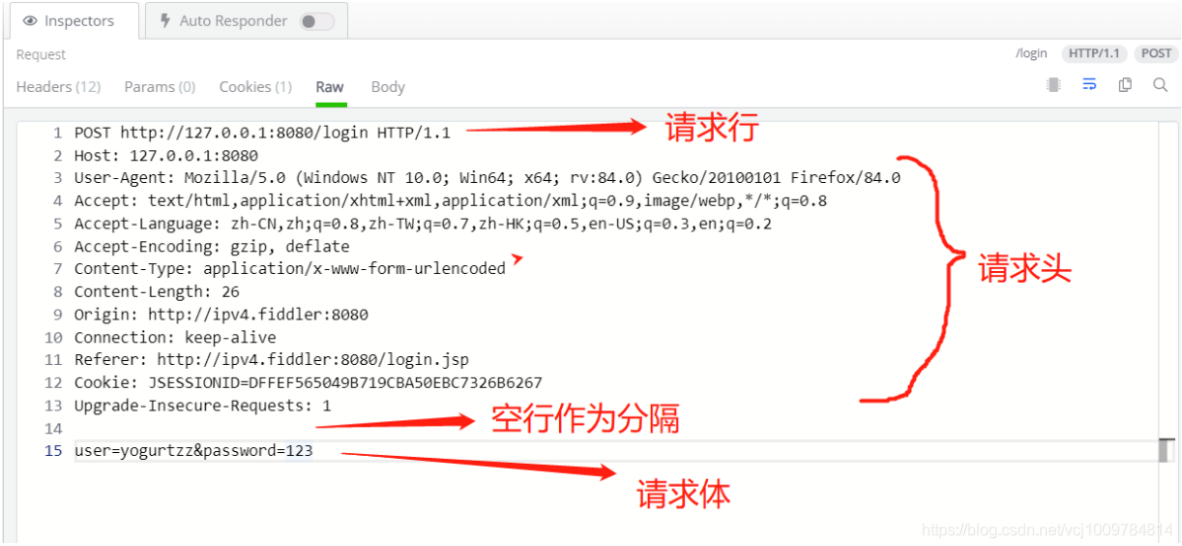
```
POST /login HTTP/1.1
Host: localhost:8025
User-Agent: Mozilla/5.0 (Windows NT 10.0; Win64; x64; rv:84.0) Gecko/20100101
Firefox/84.0
Accept-Language: zh-CN,zh;q=0.8,zh-TW;q=0.7,zh-HK;q=0.5,en-US;q=0.3,en;q=0.2
Accept-Encoding: gzip, deflate
Content-Type: application/x-www-form-urlencoded; charset=UTF-8
Content-Length: 26
Connection: keep-alive

user=yogurtzz&password=123
```

可以看到在多行请求头之后，以**一个空行**作为分隔，多出了一部分内容，这些内容即是**请求体**

所以，HTTP请求报文由3部分组成：

- 请求行：包含了请求方法，请求的URL，使用的HTTP协议版本（三者之间以**空格**分隔）
- 请求头：包含了请求的**元数据**（关于元数据）
- 请求体：包含了请求的主体数据（请求头与请求体之间以一个**空行**分隔）



HTTP响应报文

同样，先看一个实际的HTTP响应报文

```
HTTP/1.1 200 OK
Content-Type: text/html;
Transfer-Encoding: chunked
Date: Wed, 30 Dec 2020 03:20:06 GMT
Keep-Alive: timeout=60
Connection: keep-alive

<html>
  <head>
  </head>
  <body>
  success
  </body>
</html>
```

与HTTP请求报文类似，响应报文也由3个部分组成

- 响应行：包含**HTTP协议版本**，**HTTP状态码**，**状态码描述**
- 响应头：包含了响应的元数据
- 响应体：响应的主体数据

浏览器收到上面的响应报文后，会自动渲染出HTML页面，如下

HTTP响应状态码

状态码用来描述HTTP响应的情况，由三位数字组成，第一个数字表明了响应的类别，共有如下几类

	类别	说明
1xx	Informational（信息性状态码）	接收的请求正在处理
2xx	Success（成功性状态码）	请求正常处理完毕
3xx	Redirection（重定向状态码）	需要进行附加操作以完成请求
4xx	Client Error（客户端错误）	服务器无法处理请求
5xx	Server Error（服务端错误）	服务器处理请求出错

状态码定义可以参考RFC2616

通常来说，只需要熟悉以下的几种常见状态码即可，其中加粗的几种状态码非常常见。

状态码	状态码描述	说明
200	OK	请求成功
301	Move Permanently	永久重定向
302	Found	临时重定向
400	Bad Request	请求报文中存在语法错误
401	Unauthorized	需要认证
403	Forbidden	请求被服务器拒绝，可能是IP未授权等原因
404	Not Found	服务器上找不到请求的资源
500	Internal Server Error	服务器内部错误

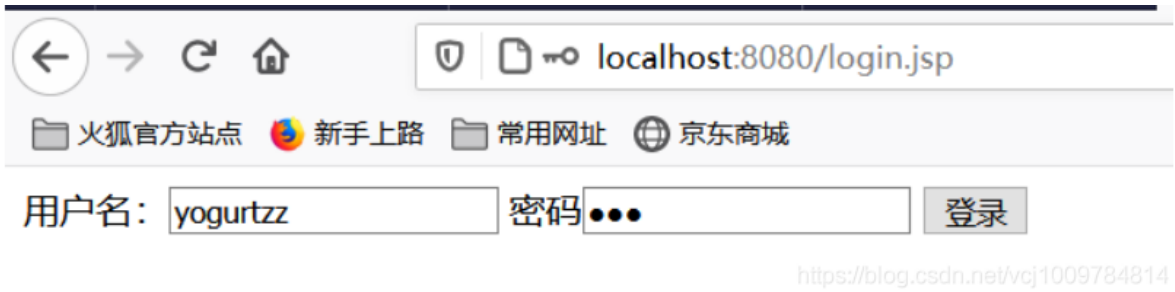
HTTP请求方法

HTTP请求方法用来告知服务器，客户端的意图，常用的有如下几种

请求方法	意图	HTTP版本
GET	获取资源	1.0/1.1
POST	传输实体	1.0/1.1
PUT	传输文件	1.0/1.1
HEAD	获得响应头部	1.0/1.1
DELETE	删除文件	1.0/1.1
OPTIONS	询问支持的请求方法	1.1
TRACE	追踪路径	1.1
CONNECT	要求建立隧道	1.1

在进行web开发时，通常只会用到GET和POST两种方法。通俗地讲，GET的请求参数是放在URL中，且通常没有请求体（但没有规定GET请求不能携带请求体，只是通常不这么做）。假设有如下表单

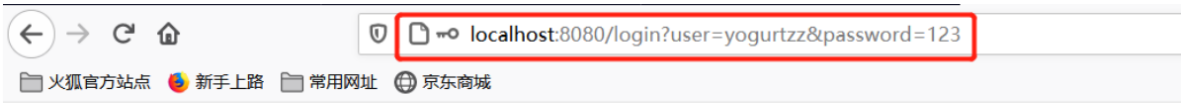
```
<form action="login" method="get">
    用户名：<input type="text" name="user"/>
    密码：<input type="password" name="password">
    <input type="submit" value="登录"/>
</form>
```

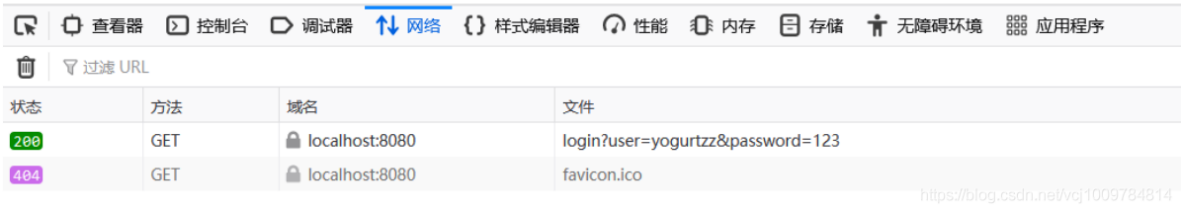


按F12打开开发者工具，能够看到请求的报文如下

```
GET /login?user=yogurtzz&password=123 HTTP/1.1
Host: localhost:8080
User-Agent: Mozilla/5.0 (Windows NT 10.0; Win64; x64; rv:84.0) Gecko/20100101
Firefox/84.0
Accept:
text/html,application/xhtml+xml,application/xml;q=0.9,image/webp,*/*;q=0.8
Accept-Language: zh-CN,zh;q=0.8,zh-TW;q=0.7,zh-HK;q=0.5,en-US;q=0.3,en;q=0.2
Accept-Encoding: gzip, deflate
Connection: keep-alive
Referer: http://localhost:8080/login.jsp
Cookie: JSESSIONID=FD82B6FF51C8B2B22407AC0892BCBA86
Upgrade-Insecure-Requests: 1
```

可以看到用GET方式提交的表单数据，是通过追加到URL中的queryString来传送的，所以请求的参数，会直接反映到浏览器的地址栏中，如下图所示。所以我们会说，GET请求是不太安全的，它把请求地数据直接显示了出来。





我们把表单的请求方法改为POST

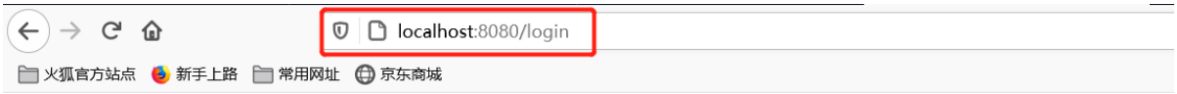
```
<form action="login" method="post">
    用户名：<input type="text" name="user"/>
    密码：<input type="password" name="password">
    <input type="submit" value="登录"/>
</form>
```

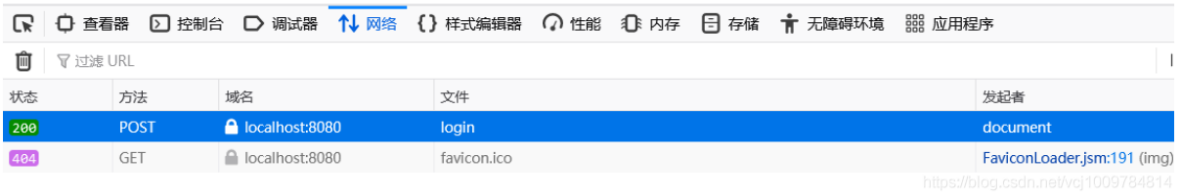
再填写用户名，密码，点击登录。得到的请求报文如下

```
POST /login HTTP/1.1
Host: localhost:8080
User-Agent: Mozilla/5.0 (Windows NT 10.0; Win64; x64; rv:84.0) Gecko/20100101
Firefox/84.0
Accept:
text/html,application/xhtml+xml,application/xml;q=0.9,image/webp,*/*;q=0.8
Accept-Language: zh-CN,zh;q=0.8,zh-TW;q=0.7,zh-HK;q=0.5,en-US;q=0.3,en;q=0.2
Accept-Encoding: gzip, deflate
Content-Type: application/x-www-form-urlencoded
Content-Length: 26
Origin: http://localhost:8080
Connection: keep-alive
Referer: http://localhost:8080/login.jsp
Cookie: JSESSIONID=A061396E89A92B916AB1FA8298354BAE
Upgrade-Insecure-Requests: 1

user=yogurtzz&password=123
```

请求的参数是放在请求体中的，不会明文显示在浏览器地址栏，所以我们说POST相对更加安全。





HTTP小结

- 定义：HTTP，全称 Hyper Text Transfer Protocol，超文本传输协议
- 作用：规范了客户端和服务端的数据交互格式
- 报文格式：
  - 请求行（响应行）
  - 请求头（响应头）
  - 请求体（响应体）
- 特点
  - 简单

    HTTP协议简单，不复杂
  - 灵活

    HTTP协议可以发送任何类型的数据（MIME类型），只要客户端和服务端能够处理这种数据。这可以通过`Content-Type`头来指定，请求头和响应头中都可以设置该头部。
  - 无连接

    在早期HTTP版本里，使用的是短链接，即每次请求和响应都会单独建立一个HTTP连接，完成一次请求响应后，连接断开。这么做的原因是，早期的网络，流量没有很大，请求具有间歇性和突发性，维持持久连接会使得连接通道长时间处于空闲状态，占用网络资源。而如今的网络请求具有频繁性和持续性，一次会话中可能会连续发送好几个HTTP请求，若为每一次的请求和响应单独建立一个HTTP连接，则频繁的建立和关闭连接会无谓地消耗性能。HTTP 1.1 之前的版本，可以通过设置请求头`Connection: keep-alive`来建立一个HTTP长连接（长连接即持久连接）。HTTP 1.1版本的默认连接则都是长连接。客户端可以在一个HTTP长连接上发送多次请求。若想断开连接，设置`Connection: close`即可。
  - 无状态

    HTTP协议不具备记忆能力。也就是说，每个HTTP请求都是独立的，完整的。后一个请求无法了解前一个请求的情况。这样设计使得HTTP协议足够简单，快速，因为不需要维护状态信息，不需要在状态之间进行转换。缺点也很明显，就是单个请求的所有信息都必须要完整地包含在一次请求中，使得单个请求的数据量可能会很大，并且先前请求中传输过的相同数据，可能在后续的请求中需要**重复传输**（如果后续的请求中需要用到前面请求的信息）。

    在某些场景下，需要让HTTP协议记住一些状态。比如用户登录的场景，用户应该只登录一次，就能够访问一个网站的全部资源，而不是每次请求资源都要重新登录。这种需要维持一个会话的场景，可通过Cookie和Session技术进行解决。

关于**有状态**（stateful）和**无状态**（stateless）的生活小例子：

- 有状态

  A：你中午吃的啥呀？

  B：吃的土豆烧排骨。

  A：味道怎么样？

  B：香的一匹。

- 无状态

  A：你中午吃的啥呀？

  B：吃的土豆烧排骨。

  A：味道怎么样？

  B：什么味道怎么样？

无状态的每次请求都是独立的，不存在上下文环境。有状态则存在上下文环境。

附录

元数据

元数据，就是数据的数据，或者可以理解为**对数据的描述**。比如客户端发起的HTTP请求报文中，请求体为`name=yogurt`，那么可以在请求头中可能有一项为：`Content-Length: 11`，以说明请求体的长度（字节数）为11。请求时传送的数据是`name=yogurt`，而请求头`Content-Length: 11`则描述了数据的长度，此为对数据的描述，即**元数据**。再比如，`Content-Type`头，这个请求头说明了请求体数据的格式。在前后端对接开发时，前端请求后端，传递参数时，最常用的两种`Content-Type`便是`application/x-www-form-urlencoded`和`application/json`，前者是表单数据，后端可以用Spring的`@RequestParam`来接收请求参数，后者是json数据，后端可以用`@RequestBody`来接收。

URL

Uniform Resource Locator，**统一资源定位符**。在使用浏览器访问web页面时需要输入的网页地址，就是URL，比如`http://www.baidu.com`

与URL相关的，还有一个概念，叫URI（Uniform Resource Identifier，**统一资源标识符**）。URI用一个字符串来**唯一标识**网络上的某一资源，而URL则额外描述了该资源的地点，即在哪里能找到这个资源。

比如一个身份证可以用来唯一标识某个具体的人，但你需要这个人的地址才能找到他。

一个绝对URI的格式如下

`http://user:pass@www.xxx.com:8808/dir/index.html?date=2021#chapter1`

scheme	user info（可选）	host	port（可选）	path	query（可选）	fragment（可选）
协议名称	登录信息(用户+密码)	服务器地址	端口号	资源路径	查询字符串	片段标识符
`http`	`user:pass`	`www.xxx.com`	`8808`	`/dir/index.html`	`date=2021`	`chapter1`

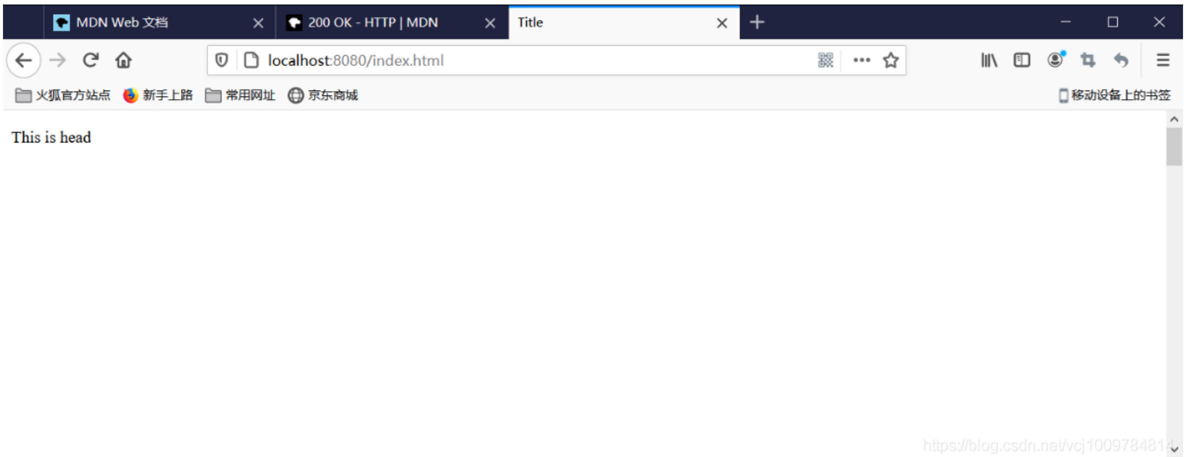
其中，**服务器地址**可以是`www.xxx.com`这种能被DNS解析的域名，或者`192.168.0.133`这种IP地址（ipv4），或者`[0:0:0:0:0:0:0:1]`这样用方括号括起来的ipv6地址。

关于**片段标识符**，假设`index.html`内容如下

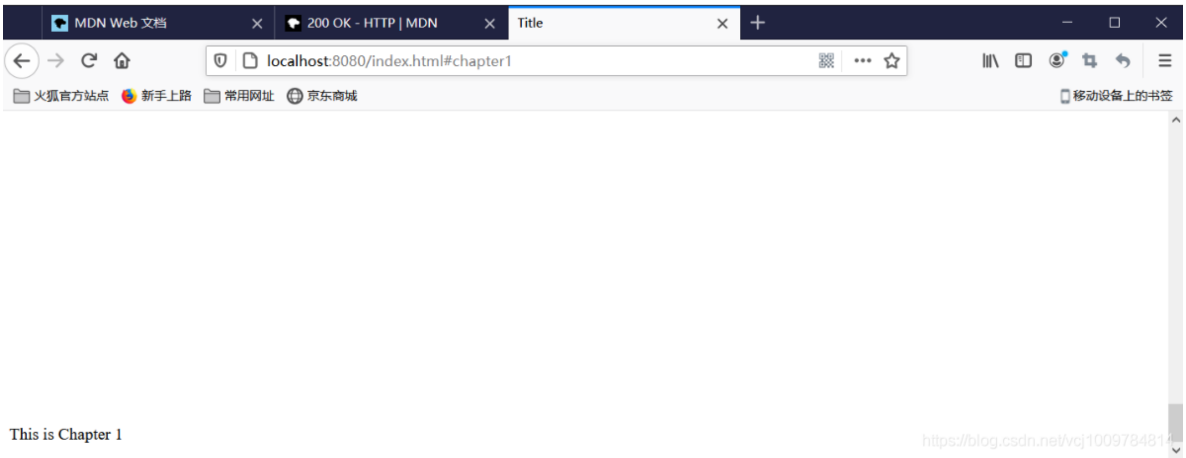
```
<!DOCTYPE html>
```

```html
<html lang="en">
<head>
    <meta charset="UTF-8">
    <title>Title</title>
</head>
<body>
    <p>This is head</p>
    <!-- 此处省略1万字 -->
    <br/><br/><br/><br/><br/><br/><br/><br/><br/><br/><br/><br/><br/><br/><br/>
<br/><br/><br/><br/><br/>
    <br/><br/><br/><br/><br/><br/><br/><br/><br/><br/><br/><br/><br/><br/><br/>
<br/><br/><br/><br/><br/>
    <br/><br/><br/><br/><br/><br/><br/><br/><br/><br/><br/><br/><br/><br/><br/>
<br/><br/><br/><br/><br/>
    <br/><br/><br/><br/><br/><br/><br/><br/><br/><br/><br/><br/><br/><br/><br/>
<br/><br/><br/><br/><br/>
    <br/><br/><br/><br/><br/><br/><br/><br/><br/><br/><br/><br/><br/><br/><br/>
<br/><br/><br/><br/><br/>
    <br/><br/><br/><br/><br/><br/><br/><br/><br/><br/><br/><br/><br/><br/><br/>
<br/><br/><br/><br/><br/>
    <p id="chapter1"> This is Chapter 1</p>
</body>
</html>
```

访问`http://localhost:8080/index.html`，结果如下。若需要看到`This is Chapter1`，则需要将滚动条往下拖



若加上片段标识符，访问`http://localhost:8080/index.html#chapter1`，则会自动定位到网页的锚点`chapter1`的位置（自动定位到一个`id`属性为`chapter1`的元素的位置，对于`<a>`元素，设置其`name`属性为`chapter1`也能够达到同样的效果）



锚点在编写markdown文档时的妙用：可以使用锚点自由的链接到文档的其他位置。比如在本篇markdown文档中，我在`HTTP请求报文`的位置，添加了一个锚点`request`

在这个地方，我只需要创建一个链接，指向`request`这个锚点，查看文档时，就可以点击链接，定位到锚点所处的位置。比如我用markdown语法，

`[点击这里](#request)`，来创建一个链接，点击就可以直接定位到锚点所处的位置，非常方便。可以试试 => 点击这里

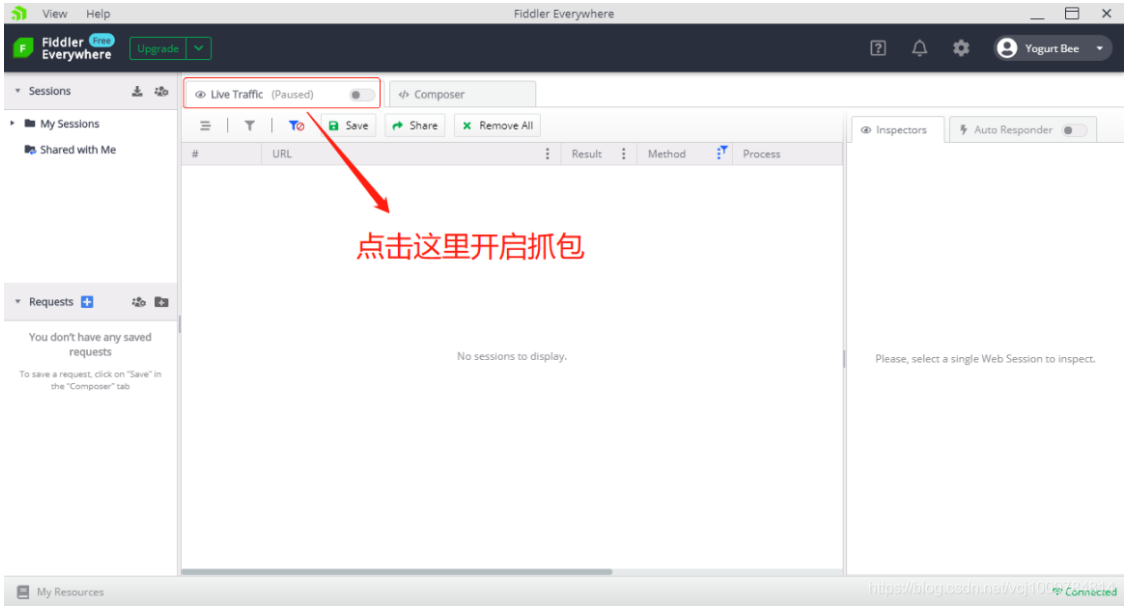
HTTP抓包体验

为了更好的学习HTTP协议，可以尝试使用抓包工具来查看HTTP报文。这里推荐`Fiddler`，它是一款免费的HTTP分析和调试工具。支持对HTTP数据包进行截取，编辑，存储等。它的操作也非常简便。下面对使用`Fiddler`进行抓包做一个简单介绍

1. 下载&安装

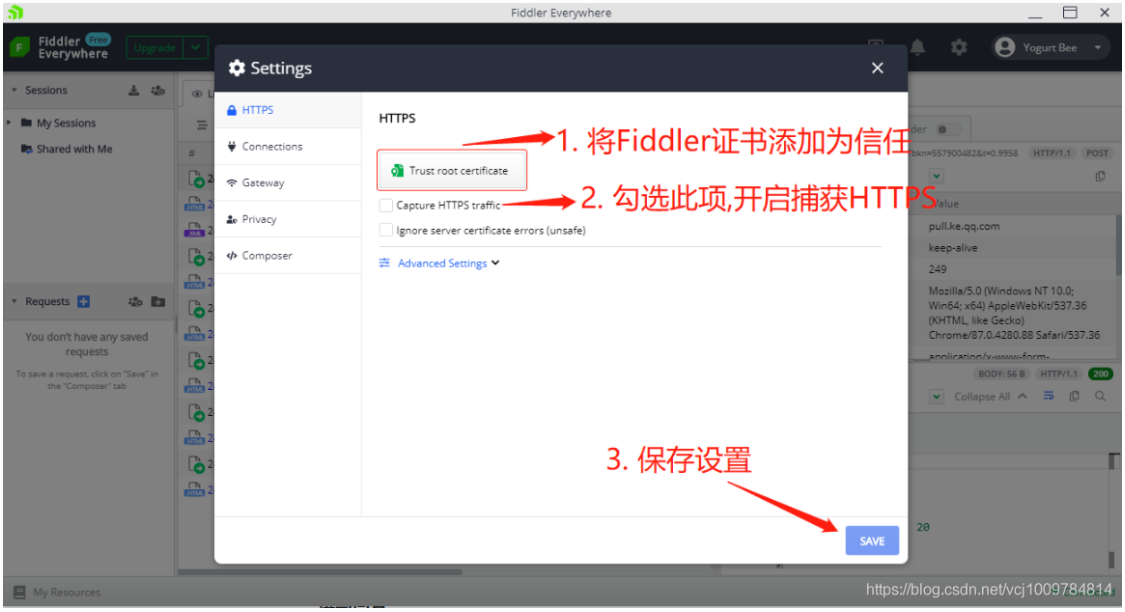
   访问`https://www.telerik.com/download/fiddler-everywhere`下载`Fiddler`，并安装

2. 启动`Fiddler`，点击按钮开始抓包

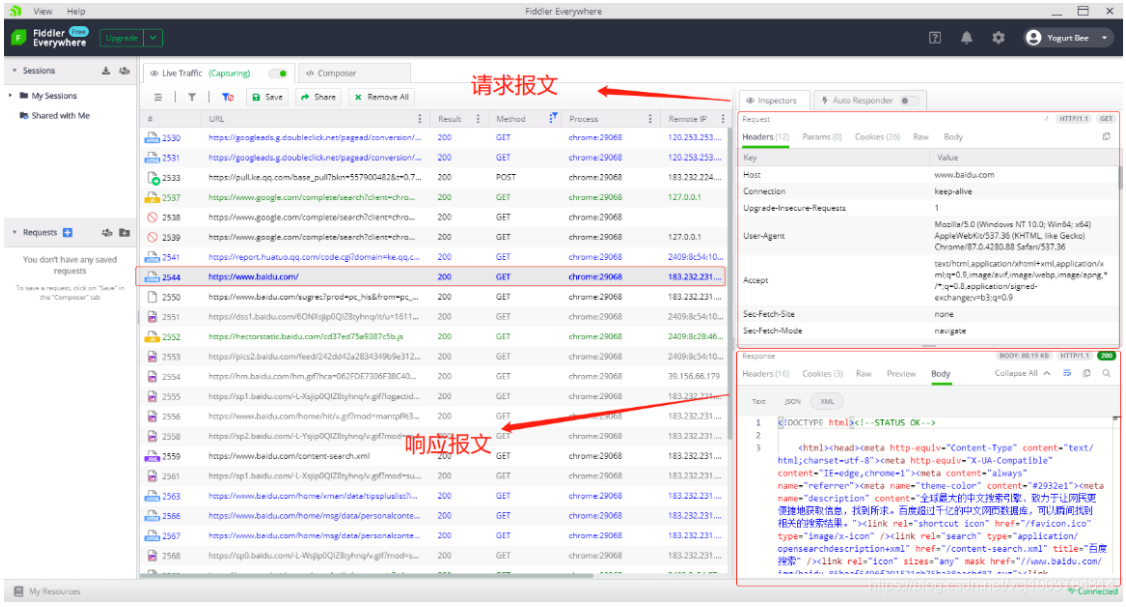


3. 打开浏览器随便访问一个页面，如`www.baidu.com`

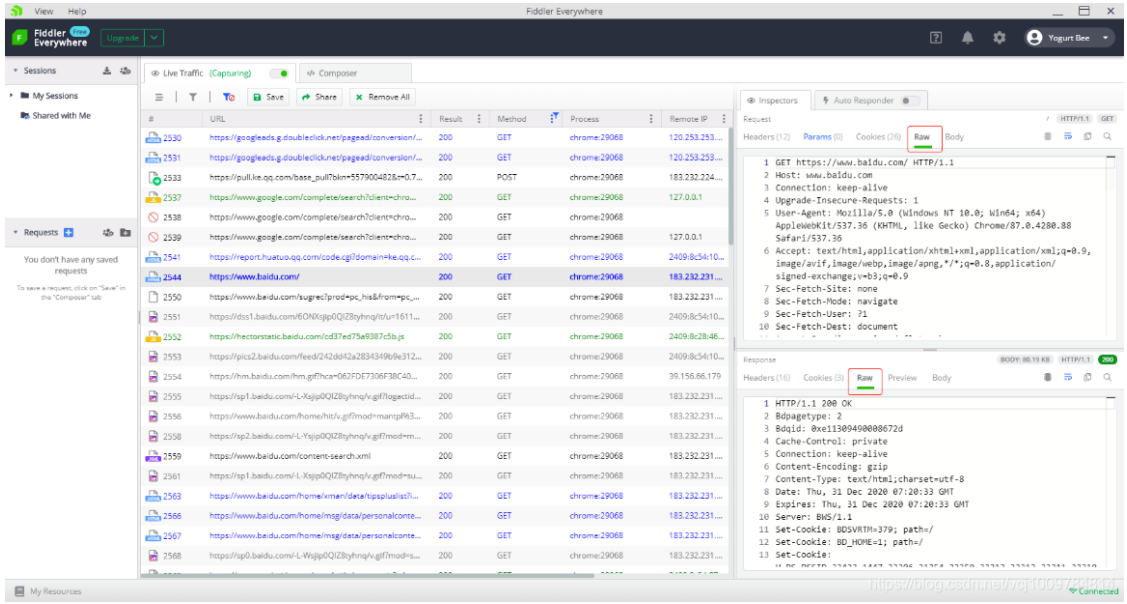
   一开始可能会奇怪，为什么在`Fiddler`中看不到HTTP报文，这可能是因为访问外部网站采用的多是HTTPS协议，而没有对Fiddler进行HTTPS配置，所以查看不到，点击右上角的`设置`按钮，进行HTTPS的设置即可。设置完毕后记得重启一下`Fiddler`



4. 再次开启抓包，就能看到访问`www.baidu.com`的HTTP报文了，选择并点击要查看的某一条HTTP请求，右侧`Inspectors`中会展示具体的请求报文和响应报文。



可以看到已经自动对报文的各个部分进行了拆分展示，如报文头头部，报文主体等。若要查看原始的报文信息，点击`Raw`即可



由于开发时，经常要对本地服务进行调试，本地服务经常使用`http://localhost`或者`http://127.0.0.1`来访问，故需要`Fiddler`抓取请求本地服务的HTTP报文。但不经任何配置，会发现在`Fiddler`中无法抓取到访问本地服务的HTTP请求。

推荐如下2种解决方式：

1. 使用`ipv4.fiddler`替换`localhost`

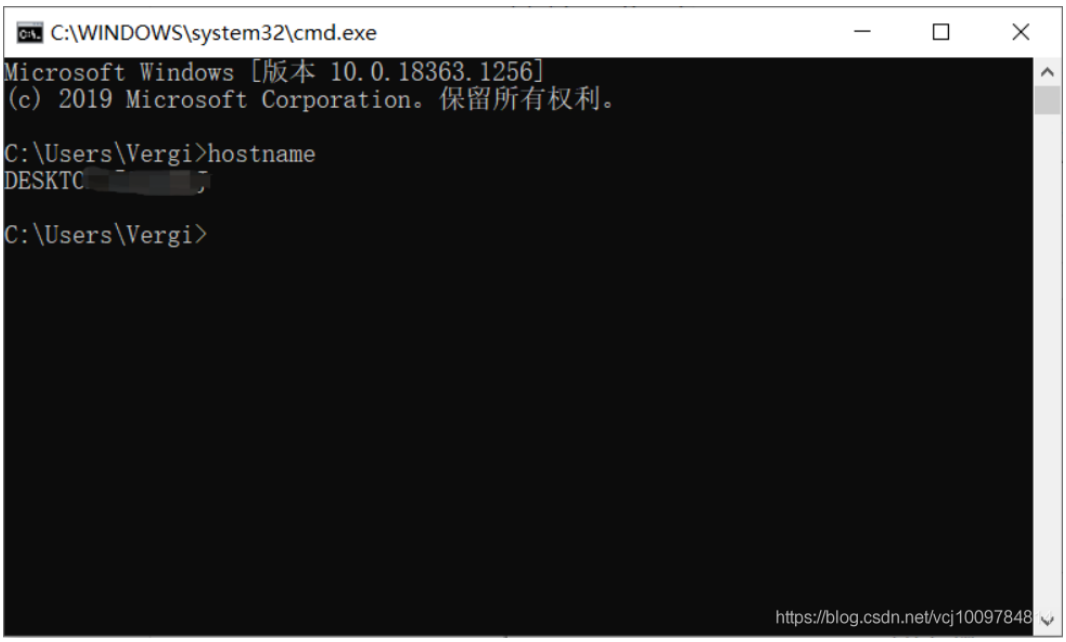
   假设要访问的本地服务地址为`http://localhost:8080/demo`，将其中的`localhost`替换`ipv4.fiddler`，则URL变为`http://ipv4.fiddler:8080/demo`

   由于Fiddler内部拦截了`ipv4.fiddler`这个域名，并自动转换到`localhost`，所以开启Fiddler时，访问`ipv4.fiddler`就等同于访问`localhost`

2. 使用计算机名替换`localhost`

   计算机名的查看方式：打开 控制面板 -> 系统，便能够看到计算机名

   或者直接打开CMD，命令提示符，敲命令`hostname`，得到的也是计算机名



   假设要访问的本地服务地址为`http://localhost:8080/demo`，将其中的`localhost`替换计算机名`DESKTOP-XXXX`，则URL变为`http://DESKTOP-XXXX:8080/demo`

   参考链接：

   https://www.racecoder.com/archives/904/

https://docs.telerik.com/fiddler/Observe-Traffic/Troubleshooting/NoTrafficToLocalhost

对本地服务的访问进行抓包测试如下