当我们在 web 浏览器的地址栏中输入：www.baidu.com，具体发生了什么（一次完整的 HTTP 服务过程）：

1.对 www.baidu.com 这个网址进行 DNS 域名解析：

a）首先会搜索浏览器自身的 DNS（高速）缓存（缓存时间比较短，大概只有 1 分钟，且只能容纳 1000 条缓存）

b）如果浏览器自身的缓存里面没有找到，那么浏览器会搜索系统自身的 DNS（高速）缓存

c）如果还没有找到，那么尝试从 hosts 文件里面去找

d）在前面三个过程都没获取到的情况下，就递归地去域名服务器【有根域名服务器》COM 顶级域名服务器》baidu.com 域名服务器（named.conf 找定义》找区域文件》找关于 www 的主机记录》找到返回）】去查找。

2.得到对应的 IP 地址后，根据这个 IP，找到对应的服务器，发起 TCP 的三次握手：

拿到域名对应的 IP 地址之后，User-Agent（一般指浏览器）会以一个随机端口（1024<端口<65535）向服务器的 WEB 程序（常用的有 httpd，nginx）等的 80 端口。这个连接请求（原始的 http 请求经过 TCP/IP 4 层模型的层层封包）到达服务器端后（这中间有各种路由设备，局域网内除外），进入到网卡，然后是进入到内核的 TCP/IP 协议栈（用于识别连接请求，解封包，一层一层的剥开），还有可能要经过 Netfilter 防火墙（属于内核的模块）的过滤，最终达到 WEB 程序，最终建立了 TCP/IP 的连接。

3.建立 TCP 连接后发起 HTTP 请求（tcp 是比 http 更底层一个连接协议）（ip 是 tcp 下面一层）：

而 HTTP 请求报文由三部分组成：请求行，请求头、空行 / 请求正文

请求行：用于描述客户端的请求方式（GET/POST 等），请求的资源名称(URL)以及使用的 HTTP 协议的版本号

请求头：用于描述客户端请求哪台主机及其端口，以及客户端的一些环境信息等

空行：空行就是\r\n (POST 请求时候有)

请求正文：当使用 POST 等方法时，通常需要客户端向服务器传递数据，这些数据就储存在请求正文中，且请求头中会多了一项 Content-Length 用于表示消息体的字节数，这样服务器才能知道请求是否发送结束。（GET 方式是保存在 url 地址后面，不会放到这里）。

4.服务器响应 HTTP 请求，浏览器得到 html 代码

5.浏览器解析 html 代码，并请求 html 代码中的资源，如 js、css、图片等（先得到 html 代码，才能去找这些资源）：

浏览器拿到 html 文件后，就开始解析其中的 html 代码，遇到 js/css/image 等静态资源时，就向服务器端去请求下载（会使用多线程下载，每个浏览器的线程数不一样），这时候就用上 keep-alive 特性了，建立一次 HTTP 连接，可以请求多个资源，下载资源的顺序就是按照代码里面的顺序，但是由于每个资源大小不一样，而浏览器又是多线程请求请求资源，所以这里显示的顺序并不一定是代码里面的顺序。

6.浏览器对页面进行渲染呈现给用户

7.服务器关闭关闭 TCP 连接：

一般情况下，一旦 Web 服务器向浏览器发送了请求数据，它就要关闭 TCP 连接，然后如果浏览器或者服务器在其头信息加入了这行代码：Connection:keep-alive ，那 TCP 连接在发送后将仍然保持打开状态，于是浏览器可以继续通过相同的连接发送请求，保持连接节省了为每个请求建立新连接所需的时间，还节约了网络带宽。

1.DNS 怎么找到域名的？

DNS 域名解析采用的是递归查询的方式，过程是，先去找 DNS 缓存->缓存找不到就去找根域名服务器->根域名又会去找下一级，这样递归查找之后，找到了，给我们的 web 浏览器

2.为什么 HTTP 协议要基于 TCP 来实现？

TCP 是一个端到端的可靠的面相连接的协议，HTTP 基于传输层 TCP 协议不用担心数据传输的各种问题（当发生错误时，会重传）

3.最后一步浏览器是如何对页面进行渲染的？

a）解析 html 文件构成 DOM 树

b）解析 CSS 文件构成渲染树

c）边解析，边渲染

d）JS 单线程运行，JS 有可能修改 DOM 结构，意味着 JS 执行完成前，后续所有资源的下载是没有必要的，所以 JS 是单线程，会阻塞后续资源下载

4.DNS 优化两个方面？DNS 缓存、DNS 负载均衡

5.起始行（请求行）中的请求方法有哪些种呢？

GET: 完整请求一个资源 （常用） HEAD: 仅请求响应首部

POST：提交表单 （常用） PUT: (webdav) 上传文件（但是浏览器不支持该方法）

DELETE：(webdav) 删除　　OPTIONS：返回请求的资源所支持的方法的方法
TRACE: 追求一个资源请求中间所经过的代理（该方法不能由浏览器发出）
6.什么是 URI、URL、URN？
URI Uniform Resource Identifier 统一资源标识符 ；
URL Uniform Resource Locator 统一资源定位符 ；
URN Uniform Resource Name 统一资源名称。
而 URL 和 URN 都属于 URI，为了方便就把 URL 和 URI 暂时都通指一个东西。
7. HTTP 响应也由三部分组成：状态行，响应头，空格和消息体。具体：
a）状态行包括：协议版本、状态码、状态码描述
B）状态码：状态码用于表示服务器对请求的处理结果：1xx：指示信息——表示请求已经接受，继续处理
2xx：成功——表示请求已经被成功接收、理解、接受。 3xx：重定向——要完成请求必须进行更进一步的操作
4xx：客户端错误——请求有语法错误或请求无法实现 5xx：服务器端错误——服务器未能实现合法的请求。
列举几种常见的：200（没有问题） 302（要你去找别人） 304（要你去拿缓存） 307（要你去拿缓存）
403（有这个资源，但是没有访问权限） 404（服务器没有这个资源） 500（服务器这边有问题）
C）响应头：响应头用于描述服务器的基本信息，以及客户端如何处理数据。
响应头中的 Content-Length 同样用于表示消息体的字节数。
Content-Type 表示消息体的类型，通常浏览网页其类型是 HTML，当然还会有其他类型，比如图片、视频等。
D）空格：CRLF（即 \r\n）分割
E）消息体：服务器返回给客户端的数据

Web 缓存大致可以分为：数据库缓存、服务器端缓存（代理服务器缓存、CDN 缓存）、浏览器缓存。
浏览器缓存也包含很多内容： HTTP 缓存、indexDB、cookie、localstorage 等等。这里只讨论 HTTP 缓存相关内容。
1.缓存命中率：从缓存中得到数据的请求数与所有请求数的比率。理想状态是越高越好。
2.过期内容：超过设置的有效时间，被标记为“陈旧”的内容。通常过期内容不能用于回复客户端的请求，必须重新向源服务器请求新的内容或者验证缓存的内容是否仍然准备。
3.验证：验证缓存中的过期内容是否仍然有效，验证通过的话刷新过期时间。
4.失效：失效就是把内容从缓存中移除。当内容发生改变时就必须移除失效的内容。
5.浏览器缓存主要是 HTTP 协议定义的缓存机制。HTML meta 标签，如
<META HTTP-EQUIV="Pragma" CONTENT="no-store"> 含义是让浏览器不缓存当前页面。但代理服务器不解析 HTML 内容，一般应用广泛的是用 HTTP 头信息控制缓存。

浏览器缓存（这里指 HTTP 缓存）分为强缓存和协商缓存：
强缓存通过 expires 和 cache-control 控制 协商缓存 通过 last-Modify 和 E-tag 控制。
浏览器加载一个页面的流程：
1.浏览器先根据这个资源的 http 头信息来判断是否命中强缓存。如果命中则直接加在缓存中的资源，并不会将请求发送到服务器。（强缓存）在 Chrome 的开发者工具中看到 http 的返回码是 200，但在 Size 列会显示为(from cache)。强缓存是利用 http 的返回头中的 Expires 或者 Cache-Control 两个字段来控制的，用来表示资源的缓存时间。
A）Expires：缓存过期时间，指定资源到期的时间，是服务器端的具体的时间点。~~即 Expires=max-age+请求时间，需要和 Last-modified 结合使用。~~Expires 是 Web 服务器响应消息头字段，在响应 http 请求时告诉浏览器在过期时间前浏览器可以直接从浏览器缓存取数据，而无需再次请求。这种方式有一个明显的缺点，由于失效时间是一个绝对时间，所以当客户端本地时间被修改以后，服务器与客户端时间偏差变大以后，就会导致缓存混乱。
B）Cache-Control：相对时间，是与客户端时间比较，所以服务器与客户端时间偏差也不会导致问题。Cache-Control 与 Expires 可以在服务端配置同时启用或者启用任意一个，同时启用的时候 Cache-Control 优先级高。Cache-Control 可以由多个字段组合而成，主要有以下几个取值：
max-age 指定一个时间长度，在这个时间段内缓存是有效的，单位是 s。
s-maxage 同 max-age，覆盖 max-age、Expires，但仅适用于共享缓存，在私有缓存中被忽略。
public 表明响应可以被任何对象（发送请求的客户端、代理服务器等等）缓存。
private 表明响应只能被单个用户（可能是操作系统用户、浏览器用户）缓存，非共享的，不能被代理服务器缓存。
no-cache 强制缓存了该响应的用户在使用已缓存的数据前，发送带验证器的请求到服务器。不是字面上的不缓存。

no-store 禁止缓存，每次请求都要向服务器重新获取数据。

must-revalidate 指定如果页面是过期的，则去服务器进行获取。这个指令并不常用。

2.如果未命中强缓存，则浏览器会将资源加载请求发送到服务器。服务器来判断浏览器本地缓存是否失效。若可以使用，则服务器并不会返回资源信息，浏览器继续从缓存加载资源。（协商缓存）服务器根据 http 头信息中的 Last-Modify/If-Modify-Since 或 Etag/If-None-Match 来判断是否命中协商缓存。如果命中，则 http 返回码为 304，浏览器从缓存中加载资源。

A）浏览器第一次请求一个资源的时候，服务器返回的响应头 header 中会加上 Last-Modify，Last-modify 是一个时间标识，表示该资源的最后修改时间。

当浏览器再次请求该资源时，发送的请求头中会包含 If-Modify-Since，该值为缓存之前返回的 Last-Modify。服务器收到 If-Modify-Since 后，根据资源的最后修改时间判断是否命中缓存。

如果命中协商缓存，则返回 http304，并且不会返回资源内容，并且不会返回 Last-Modify。由于对比的服务端时间，所以客户端与服务端时间差距不会导致问题。但是有时候通过最后修改时间来判断资源是否修改还是不太准确（资源变化了最后修改时间也可以一致，不能精确到秒）。

B）HTTP1.1 中 Etag 的出现主要是为了解决几个 Last-Modified 比较难解决的问题：Last-Modified 标注的最后修改只能精确到秒级，如果某些文件在 1 秒钟以内，被修改多次的话，它将不能准确标注文件的修改时间；如果某些文件会被定期生成，当有时内容并没有任何变化，但 Last-Modified 却改变了，导致文件没法使用缓存；有可能存在服务器没有准确获取文件修改时间，或者与代理服务器时间不一致等情形。

Etag 是服务器自动生成或者由开发者生成的对应资源在服务器端的唯一标识符，能够更加准确的控制缓存。Last-Modified 与 ETag 是可以一起使用的，服务器会优先验证 ETag，一致的情况下，才会继续比对 Last-Modified，最后才决定是否返回 304。

以 Apache 为例，ETag 靠以下几种因子生成：文件的 i-node 编号（非 iNode），是 Linux/Unix 用来识别文件的编号，不是文件名，使用命令’ls －I’可以看到；文件最后修改时间；文件大小。生成 Etag 的时候，可以使用其中一种或几种因子，使用抗碰撞散列函数来生成。所以，理论上 ETag 也是会重复的，只是概率小到可以忽略。

3.如果未命中协商缓存，则服务器会将完整的资源返回给浏览器，浏览器加载新资源，并更新缓存。（新的请求）

Ajax 技术，通常意思是基于 XMLHttpRequest 的 Ajax，它是一种能够有效改进页面通信的技术。Ajax 的兴起是由于 Google 的 Gmail 所带动的，开发者已经默认将 XMLHttpRequest 作为了当前 Web 应用与远程资源进行通信的基础。XMLHttpRequest 的最新替代技术 Fetch API，是由 whatwg 组织提出的，是 W3C 的正式标准。

fetch 是浏览器提供的 api，axios 是社区封装的一个组件。

fetch 优势：语法简洁，更加语义化；基于标准 Promise 实现，支持 async/await；更加底层，提供的 API 丰富（request, response）；脱离了 XHR，是 ES 规范里新的实现方式。

fetch 存在问题 ：

1.fetch 是一个低层次的 API，可把它考虑成原生的 XHR，所以使用起来并不是那么舒服，需要进行封装；

2.fetch 只对网络请求报错，对 400、500 都当做成功的请求，服务器返回 400、500 错误码时并不会 reject，只有网络错误这些导致请求不能完成时，fetch 才会被 reject；

3.fetch 默认不会带 cookie，需要添加配置项，fetch(url, {credentials: 'include'}) ；fetch 不支持 abort，不支持超时控制，使用 setTimeout 及 Promise.reject 的实现的超时控制并不能阻止请求过程继续在后台运行，造成了流量的浪费；

4.fetch 没有办法原生监测请求的进度，而 XHR 可以。

```
fetch 例子：fetch('http://example.com/movies.json')  //第二个参数  指定 post  get
  .then(function(response) {
    return response.json();
  })
  .then(function(myJson) {
    console.log(myJson);
  });
```

axios 是一个基于 Promise 的用于浏览器和 nodejs 的 HTTP 客户端，本质上也是对原生 XHR 的封装，只不过它是

Promise 的实现版本，符合最新 ES 规范，具有以下特征：从浏览器中创建 XMLHttpRequest；支持 Promise API；客户端支持防止 CSRF；提供了一些并发请求的接口，方便了很多的操作；从 node.js 创建 http 请求；拦截请求和响应；转换请求和响应数据；取消请求；自动转换 JSON 数据。
axios 既提供了并发的封装，也没有 fetch 的各种问题，而且体积也较小，当之无愧现在最应该选用的请求的方式。

浏览器内多个标签页之间的通讯：
1.浏览器存储：
A）调用 localStorage：
在一个标签页调用 localStorage.setItem(name,val)保存数据或 localStorage.removeItem(name)删除数据的时候会触发'storage'事件。在另外一个标签页监听 document 对象的 storage 事件，在事件 event 对象属性中获取信息，包含 domain、newValue、oldValue、key。如
标签页 1：

```
        window.onload = function () {
            var btnEle = document.getElementById('btn');
            var nameEle = document.getElementById('name');
            btnEle.onclick = function () {
                var name = nameEle.value;
                localStorage.setItem("name", name);
            }
        }
```

标签页 2：

```
        window.onload = function () {
            window.addEventListener("storage", function (event) {
                console.log(event.key + "=" + event.newValue);
            });
        }
```

B）调用 cookie+setInterval()：
将要传递的信息（A 页面）存储在 cookie 中，每隔一定时间（B 页面）读取 cookie 信息，即可获取要传递的信息。
页面 1：

```
$(function(){
$("#btn").click(function(){
var name=$("#name").val();
document.cookie="name="+name;
}); });
```

页面 2：

```
$(function(){
function getCookie(key) {
return JSON.parse("{\"" + document.cookie.replace(/;\s+/gim,"\",\"").replace(/=/gim, "\":\"") + "\"}")[key];
}
setInterval(function(){
console.log("name=" + getCookie("name"));
}, 10000);
});
```

2.监听服务器事件：
A）websocket 通讯：
WebSocket 是全双工(full-duplex)通信自然可以实现多个标签页之间的通信。WebSocket 是 HTML5 新增的协议，它的目的是在浏览器和服务器之间建立一个不受限的双向通信的通道，如服务器可以在任意时刻发送消息给浏览器。
为什么传统的 HTTP 协议不能做到 WebSocket 实现的功能？这是因为 HTTP 协议是一个请求－响应协议，请求必须先由浏览器发给服务器，服务器才能响应这个请求，再把数据发送给浏览器。

虽然表面上 HTTP 协议能实现 WebSocket 实现的功能，如用轮询或者 Comet。但是轮询机制的缺点是实时性不够，且频繁的请求会给服务器带来极大的压力；Comet 本质上也是轮询，但是在没有消息的情况下，服务器先拖一段时间，等到有消息了再回复，Comet 机制暂时地解决了实时性问题，但是它带来了新的问题是以多线程模式运行的服务器会让大部分线程大部分时间都处于挂起状态，极大地浪费服务器资源，另外，一个 HTTP 连接在长时间没有数据传输的情况下，链路上的任何一个网关都可能关闭这个连接，而网关是我们不可控的，这就要求 Comet 连接必须定期发一些 ping 数据表示连接“正常工作”。

WebSocket 并不是全新的协议，而是利用了 HTTP 协议来建立连接。为什么 WebSocket 连接可以实现全双工通信而 HTTP 连接不行呢？实际上 HTTP 协议是建立在 TCP 协议之上的，TCP 协议本身就实现了全双工通信，但是 HTTP 协议的请求一应答机制限制了全双工通信而已。安全的 WebSocket 连接机制和 HTTPS 类似。首先，浏览器用 wss://xxx 创建 WebSocket 连接时，会先通过 HTTPS 创建安全的连接，然后，该 HTTPS 连接升级为 WebSocket 连接，底层通信走的仍然是安全的 SSL/TLS 协议。

WebSocket 连接必须由浏览器发起，这样做的特点：

（1）建立在 TCP 协议之上，服务器端的实现比较容易。

（2）与 HTTP 协议有着良好的兼容性。默认端口也是 80 和 443，并且握手阶段采用 HTTP 协议，因此握手时不容易屏蔽，能通过各种 HTTP 代理服务器。

（3）数据格式比较轻量，性能开销小，通信高效。

（4）可以发送文本，也可以发送二进制数据。

（5）没有同源限制，客户端可以与任意服务器通信。

（6）协议标识符是 ws（如果加密，则为 wss），服务器网址就是 URL。

```
//示例：  浏览器端代码：
// Create WebSocket connection.
const socket = new WebSocket('ws://localhost:8080');
// Connection opened
socket.addEventListener('open', function (event) {
    socket.send('Hello Server!');
});
// Listen for messages
socket.addEventListener('message', function (event) {
    console.log('Message from server ', event.data);
});
```

B）html5 浏览器的新特性 SharedWorker：

普通的 webworker 直接使用 new Worker()即可创建，这种 webworker 是当前页面专有的。

另外有种共享 worker(SharedWorker)，这种是可以多个标签页、iframe 共同使用的。SharedWorker 可以被多个 window 共同使用，但必须保证这些标签页都是同源的(相同的协议，主机和端口号)

```
// 新建一个 js 文件 worker.js，sharedWorker 所要用到的 js 文件，不必打包到项目中，直接放到服务器即可
let data = '';
let onconnect = function (e) {
  let port = e.ports[0];
  port.onmessage = function (e) {
    if (e.data === 'get') {
      port.postMessage(data)
    } else {
      data = e.data
    } } }
```

webworker 端的代码如上，只需注册一个 onmessage 监听信息的事件，客户端(即使用 sharedWorker 的标签页)发送 message 时就会触发。注意 webworker 无法在本地使用，出于浏览器本身的安全机制，要放在服务器上，worker.js 和 index.html 在同一目录。

因为客户端和 webworker 端的通信不像 websocket 那样是全双工的，所以客户端发送数据和接收数据要分成两步来

处理，即向 sharedWorker 发送数据的请求以及获取数据的请求，但他们本质上都是相同的事件，即发送消息。webworker 端会进行判断，传递的数据为'get'时，就把变量 data 的值回传给客户端，其他情况，则把客户端传递过来的数据存储到 data 变量中。

```
// 客户端的代码，打开页面后注册 SharedWorker，显示指定 worker.port.start()方法建立与 worker 间的连接
    if (typeof Worker === "undefined") {
      alert('当前浏览器不支持 webworker')
    } else {
      let worker = new SharedWorker('worker.js')
      worker.port.addEventListener('message', (e) => {
        console.log('来自 worker 的数据：', e.data);
      }, false);
      worker.port.start();
      window.worker = worker;
    }
// 获取和发送消息都是调用 postMessage 方法，我这里约定的是传递'get'表示获取数据。
window.worker.port.postMessage('get')
window.worker.port.postMessage('发送信息给 worker')
```

页面 A 发送数据给 worker，然后打开页面 B，调用 window.worker.port.postMessage('get')，即可收到页面 A 发送给 worker 的数据。

在 Web 安全领域中，XSS 和 CSRF 是最常见的攻击方式。

XSS，即 Cross Site Script，跨站脚本攻击；其原本缩写是 CSS，但为了和层叠样式表(Cascading Style Sheet)有所区分，因而在安全领域叫做 XSS。XSS 攻击是指攻击者在网站上注入恶意的客户端代码，通过恶意脚本对客户端网页进行篡改，从而在用户浏览网页时，对用户浏览器进行 控制或者获取用户隐私数据 的一种攻击方式。

攻击者对客户端网页注入的恶意脚本一般包括 JavaScript，有时也会包含 HTML 和 Flash。有很多种方式进行 XSS 攻击，但它们的共同点为：将一些隐私数据像 cookie、session 发送给攻击者，将受害者重定向到一个由攻击者控制的网站，在受害者的机器上进行一些恶意操作。

XSS 攻击可以分为 3 类：反射型（非持久型）、存储型（持久型）、基于 DOM：

1、反射型 （Reflected XSS ） 发出请求时，XSS 代码出现在 url 中，作为输入提交到服务器端，服务器端解析后响应，XSS 代码随响应内容一起传回给浏览器，最后浏览器解析执行 XSS 代码。这个过程像一次反射，所以叫反射型 XSS。

2、存储型存 Stored XSS，具有攻击性的脚本被保存到了服务器端（数据库，内存，文件系统）并且可以被普通用户完整的从服务的取得并执行，从而获得了在网络上传播的能力。

3、DOM 型 （DOM-based or local XSS） 即基于 DOM 或本地的 XSS 攻击：其实是一种特殊类型的反射型 XSS，它是基于 DOM 文档对象模型的一种漏洞。可以通过 DOM 来动态修改页面内容，从客户端获取 DOM 中的数据并在本地执行。基于这个特性，就可以利用 JS 脚本来实现 XSS 漏洞的利用。

防御措施：

（1）输入过滤，避免 XSS 的方法之一主要是将用户输入的内容进行过滤。对所有用户提交内容进行可靠的输入验证，包括对 URL、查询关键字、POST 数据等，仅接受指定长度范围内、采用适当格式、采用所预期的字符的内容提交，对其他的一律过滤。(客户端和服务器都要)

（2）输出转义。例如： 往 HTML 标签之间插入不可信数据的时候，首先要做的就是对不可信数据进行 HTML Entity 编码。

```
function htmlEncodeByRegExp   (str){
        var s = "";
        if(str.length == 0) return "";
        s = str.replace(/&/g,"&amp;");
        s = s.replace(/</g,"&lt;");
        s = s.replace(/>/g,"&gt;");
```

```
        s = s.replace(/ /g," ");
        s = s.replace(/\'/g,"&#39;");
        s = s.replace(/\"/g,"&quot;");
        return s;
 }
var tmpStr="<p>123</p>";
var html=htmlEncodeByRegExp (tmpStr)
console.log(html) //&lt;p&gt;123&lt;/p&gt;
document.querySelector(".content").innerHTML=html; //<p>123</p>
```

（3）使用 HttpOnly Cookie

将重要的 cookie 标记为 httponly，这样的话当浏览器向 Web 服务器发起请求的时就会带上 cookie 字段，但是在 js 脚本中却不能访问这个 cookie，这样就避免了 XSS 攻击利用 JavaScript 的 document.cookie 获取 cookie。

（4）现代 web 开发框架如 vue.js、react.js 等，在设计的时候就考虑了 XSS 攻击对 html 插值进行了更进一步的抽象、过滤和转义，只要熟练正确地使用他们，就可以在大部分情况下避免 XSS 攻击。

CSRF，即 Cross-site request forgery，跨站请求伪造；CSRF，伪造请求，冒充用户在站内的正常操作。绝大多数网站是通过 cookie 等方式辨识用户身份（包括使用服务器端 Session 的网站，因为 Session ID 也是大多保存在 cookie 里面的），再予以授权的。所以要伪造用户的正常操作，最好的方法是通过 XSS 或链接欺骗等途径，让用户在本机（即拥有身份 cookie 的浏览器端）发起用户所不知道的请求。CSRF 攻击攻击原理及过程如下：

1.用户 C 打开浏览器，访问受信任网站 A，输入用户名和密码请求登录网站 A；

2.在用户信息通过验证后，网站 A 产生 Cookie 信息并返回给浏览器，此时用户登录网站 A 成功，可以正常发送请求到网站 A；

3.用户未退出网站 A 之前，在同一浏览器中，打开一个 TAB 页访问网站 B；

4.网站 B 接收到用户请求后，返回一些攻击性代码，并发出一个请求要求访问第三方站点 A；

5.浏览器在接收到这些攻击性代码后，根据网站 B 的请求，在用户不知情的情况下携带 Cookie 信息，向网站 A 发出请求，而网站 A 并不知道该请求其实是由 B 发起的，所以会根据用户 C 的 Cookie 信息以 C 的权限处理该请求，导致来自网站 B 的恶意代码被执行。

防范 CSRF：

（1）验证 HTTP Referer 字段，利用 HTTP 头中的 Referer 判断请求来源是否合法，Referer 记录了该 HTTP 请求的来源地址。优点：简单易行，只需要在最后给所有安全敏感的请求统一增加一个拦截器来检查 Referer 的值就可以。特别是对于当前现有的系统，不需要改变当前系统的任何已有代码和逻辑，没有风险，非常便捷。 缺点：Referer 的值是由浏览器提供的，不可全信，低版本浏览器下 Referer 存在伪造风险。用户自己可以设置浏览器使其在发送请求时不再提供 Referer 时，网站将拒绝合法用户的访问。

（2）在请求地址中添加 token 并验证。CSRF 攻击之所以能够成功，是因为黑客可以完全伪造用户的请求，该请求中所有的用户验证信息都是存在于 cookie 中，因此黑客可以在不知道这些验证信息的情况下直接利用用户自己的 cookie 来通过安全验证。要抵御 CSRF，关键在于在请求中放入黑客所不能伪造的信息，并且该信息不存在于 cookie 之中。可以在 HTTP 请求中以参数的形式加入一个随机产生的 token，并在服务器端建立一个拦截器来验证这个 token，如果请求中没有 token 或者 token 内容不正确，则认为可能是 CSRF 攻击而拒绝该请求。优点：这种方法要比检查 Referer 要安全一些，token 可以在用户登陆后产生并放于 session 之中，然后在每次请求时把 token 从 session 中拿出，与请求中的 token 进行比对。缺点：对所有请求都添加 token 比较困难，且难以保证 token 本身的安全，依然会被利用获取到 token。

（3）在 HTTP 头中自定义属性并验证。这种方法也是使用 token 并进行验证，和上一种方法不同的是，这里并不是把 token 以参数的形式置于 HTTP 请求之中，而是把它放到 HTTP 头中自定义的属性里。通过 XMLHttpRequest 这个类，可以一次性给所有该类请求加上 csrftoken 这个 HTTP 头属性，并把 token 值放入其中。这样解决了上种方法在请求中加入 token 的不便，同时，通过 XMLHttpRequest 请求的地址不会被记录到浏览器的地址栏，也不用担心 token 会透过 Referer 泄露到其他网站中去。优点：统一管理 token 输入输出，可以保证 token 的安全性。缺点：有局限性，无法在非异步的请求上实施。