Web 安全

XSS

- **概念：** XSS(Cross-Site Scripting) 跨站脚本攻击，一种代码注入攻击。攻击者通过在目标上注入恶意脚本，使之在用户的浏览器上运行。利用这些恶意脚本，攻击者可获取用户的敏感信息如 Cookie、SessionID，进而危害数据安全。
- 常见注入方法：
    - 在 HTML 中内嵌的文本中，恶意内容以 script 标签形成注入
    - 内联的 Javascript 中，拼接的数据突破了原本的限制（字符串，变量，方法名）
    - 在标签属性中，恶意内容包含引号，从而突破属性值的限制，注入其他属性或标签。
    - 在标签的 href、src 等属性中，包含可执行代码
    - 在 onload、onerror、onclick 等事件中，注入不受控制代码
- 分类
    - 存储型(论坛发帖、商品评论、用户私信)
        - 攻击者将恶意代码提交到目标网站数据库
        - 用户打开目标网站时，网站服务端从数据库中取出恶意代码，拼接在 HTML 中返回给浏览器
        - 用户浏览器接收到响应后解析执行，恶意代码也被执行
        - 恶意代码窃取用户数据并发送到攻击者的网站，或者冒充用户行为，调用目标网站接口执行攻击者指定的操作。

      例如攻击者在评论框中输入了，这段代码就被提交到数据库中，下次客户端请求数据时就会解析执行。

      ```
      "><script>alert('xss')</script>;
      ```

    - 反射型(网站搜索、跳转)
        - 攻击者诱导用户访问一个带有恶意代码的 URL
        - 用户打开带有恶意 URL 后，服务端将恶意代码从 URL 取出并处理，拼接在 HTML 返回给浏览器
        - 浏览器接收到响应后将这段带有 xss 攻击的数据当做脚本执行

- 恶意代码窃取用户数据并发送到攻击者的网站，或者冒充用户的行为，调用目标网站接口执行攻击者指定的操作。

例如攻击者输入这条 url 来访问网站

```
http://www.seauning.com?q=
<script>alert('xss攻击')</script>
```

- DOM 型
  - 攻击者构造出特殊的 URL，包含恶意代码。
  - 用户打开带有恶意代码的 URL
  - 接收到响应后解析执行，前端 JavaScript 取出 URL 中的恶意代码并执行
  - 恶意代码窃取用户数据并发送到攻击者的网站，或者冒充用户行为，调用目标网站接口执行攻击者指定的操作。

- 预防方法：
  - 存储型和反射型
    - 改为纯前端渲染(需注意避免 DOM 型 XSS 漏洞)
      - 先加载一个静态 HTML，不包含任何业务相关数据
      - JavaScript 通过 Ajax 加载业务数据，调用 DOM API 更新到页面上

      补充： 纯前端渲染中，会明确的告诉浏览器：下面要设置的内容是文本(.innerText)，还是属性(.setAttribute)，还是样式(.style)等等。浏览器不会被轻易的被欺骗，执行预期外的代码了。

    - 对 HTML 作充分转义。

  - DOM 型
    - 在使用 .innerHTML、.innerText 时要特别小心，不要把不可信的数据作为 HTML 插到页面上，而应尽量使用 .textContent、.setAttribute() 等
    - 如果使用 Vue 的 v-html，在前端 render 阶段避免 innerHTML、outerHTML 的 XSS 隐患
    - DOM 中的内联事件监听器，如 location、onclick、onerror、onload、onmouseover 等，`<a>` 标签的 href 属性，JavaScript 的 eval()、setTimeout()、setInterval() 等，都能把字符串作为代码运行。

- 其他防范方法
    - httpOnly: 在 cookie 中设置 HttpOnly 属性后，js 脚本将无法读取到 Cookie 的值。
    - 输入过滤: 一般是用于对于输入格式的检查，例如：邮箱，电话号码，按照规定的格式输入。
    - 转义 HTML：对于引号，尖括号，斜杠进行转义(转义库)
    - 使用 CSP (Content Security Policy 内容安全政策) 建立一个白名单，告诉浏览器哪些外部资源可以 `加载和执行` ，从而防止恶意代码的注入攻击。
        - 开启方式
            - 通过 HTTP 首部中的 Content-Security-Policy
            - 通过设置 meta 标签
        - 禁止加载外域代码，防止复杂的攻击逻辑。
        - 禁止外域提交，网站被攻击后，用户的数据不会泄露到外域。
        - 禁止内联脚本执行。
        - 禁止未授权的脚本执行。
        - 合理使用上报可以及时发现 XSS，利于尽快修复问题。

CSRF

- CSRF（Cross-site request forgery）跨站请求伪造：攻击者诱导受害者进入第三方网站，在第三方网站中，向被攻击网站发送跨站请求。利用受害者在被攻击网站已经获取的注册凭证，绕过后台的用户验证，达到冒充用户对被攻击的网站执行某项操作的目的。
- 类型
    - GET 类型，只需要一个 HTTP 请求。在受害者访问含有这个 img 的页面后，浏览器会自动向接口发出一次 HTTP 请求。bank.example 就会收到包含受害者登录信息的一次跨域请求。

```
<img src="http://bank.example/withdraw?amount=10000&for=hacker" />
```

    - POST 类型，一个隐藏的自动提交的表单。访问该页面后，表单会自动提交，模拟用户完成了一次 POST 操作。

```
<form
action="http://bank.example/withdraw"
method="POST">
  <input type="hidden" name="account"
value="xiaoming" />
  <input type="hidden" name="amount"
value="10000" />
  <input type="hidden" name="for"
value="hacker" />
</form>
<script>
  document.forms[0].submit();
</script>
```

- 链接类型，用户点击链接才会触发。以广告的形式诱导用户点击恶意链接，由于之前用户登录了信任的网站 A，并且保存登录状态，只要用户主动访问的这个 PHP 页面，则攻击成功。

```
<a
href="http://test.com/csrf/withdraw.php?
amount=1000&for=hacker" taget="_blank">
重磅消息！！</a>
```

- 防护策略
  - 同源检测，禁止不受信任的域名对网站发起请求。HTTP 协议中，每一个异步请求都会携带两个字段 Origin、Referer，用于标记来源域名。
    - 使用 Origin 头部字段确定来源域名，包含请求的域名（不包含 path 及 query）

      存在两种情况没有 Origin，IE11 同源策略：IE11 不会在跨站 CORS 请求上添加 Origin 标头；302 重定向：在 302 重定向之后 Origin 不包含在重定向的请求中，因为 Origin 可能会被认为是其他来源的敏感信息。
    - 使用 Referer 头部字段确定来源域名

      Referrer Policy 的策略设置成 same-origin，对于同源的链接和引用，会发送 Referer，Referer 值为 Host 不带 Path

      如何设置 Referrer Policy
      - 在 CSP 设置
      - 页面头部添加 meta 标签
      - a 标签增加 Referrer policy 属性

        - 当 Origin 和 Referer 都不存在时，直接阻止请求。Referer 可以被伪造，同时会屏蔽搜索引擎的链接，所以一般网站会运行搜索引擎的页面请求，但是相应的页面请求的请求方式也可能被攻击者利用。

    - CSRF Token

      提交时要求附加本域才能获取的信息。CSRF 攻击之所以能够成功，是因为服务器误把攻击者请发送的请求当成了用户自己的请求。那么我们可以要求所有的用户请求都携带一个 CSRF 攻击者无法获取到的 Token。服务器通过校验请求是否携带正确的 Token，来把正求区常的请求和攻击的请分开，也可以防范 CSRF 的攻击。

      用户登录之后，服务端向客户端返回一个 Token ，Token 中包括一些用户信息以及一个随机数，并且服务端会对这个 Token 进行加密，在之后客户端每次发起请求时都在请求头中携带这个 Token。一般向服务器发起请求都是通过事件的方式，这样我们可以统一拦截 xhr 的发送，并在请求头中统一携带 Token。

      服务器验证 Token 是否正确性，验证过程是先解密 Token，对比加密字符串以及时间戳，如果加密字符串一致且时间未过期，那么这个 Token 就是有效的。

    - 设置 Cookie 时，设置 Samesite 字段
        - Strict 严格模式，完全禁止第三方请求携带 Cookie。
        - Lax 宽松模式，只能在 GET 方法提交表单，或者 a 标签发送 GET 请求时能携带 Cookie
        - None，默认携带

    - 双重 Cookie 验证
        - 在用户访问网站页面时，向请求域名注入一个 Cookie 内容为随机字符串
        - 在客户端端向服务端端发起请求时，取出 Cookie 中的随机字符串，并添加到 URL 参数中
        - 服务端接口验证 Cookie 中的字段与 URL 参数中的字段是否一致，不一致则拒绝。

同源策略

1. 源及是协议、域名和端口号
   如 `http://www.seauning.com:80`
   若地址里面的协议、域名和端口号均相同则属于同源

   以下是相对http://www.seauning.com:80 的同源检测

    - http://www.seauning.com/dir/page.html --- 成功
    - http://www.seaun.com/dir/page.html --- 失败，域名不同

- https://www.seauning.com/dir/page.html --- 失败，协议不同
- https://www.seauning.com:8080/dir/page.html --- 失败，端口号不同

2. 同源策略的概念

   同源策略是浏览器的一个安全功能，不同源的客户端脚本在没有明确授权的情况下，不能读写对方资源。所以 seauning.com 下的 js 脚本采用 ajax 读取 seaun.com 里面的文件数据是会报错的。

   不受同源策略限制的：

   1. 页面中的链接，重定向以及表单提交是不会受到同源策略限制的。
   2. 跨域资源的引入是可以的。但是 js 不能读写加载的内容。如嵌入到页面中的 `<script src="..."></script>`，`<img>`，`<link>`，`<iframe>` 等。

3. 为什么浏览器会有同源策略

   为了限制其他源文档或脚本与当前源的资源进行交互。

   主要有两个地方：

   - 一个是 iframe 节点访问控制，如果没有同源策略限制的话，iframe 可以随意访问其他非同源 iframe 的 dom 节点，如钓鱼网站嵌套了一个银行网站的 iframe ，从外部就可以读取到内部密码输入框的值。
   - 另一个是 http 响应控制，如果没有同源策略限制的话，在第三方网站直接对其他网站发起请求，可以读取到响应，这样就可以获取用户的个人信息，造成隐私泄露。

跨域

想要访问另一个源下的资源是就需要跨域。

- window.name + iframe(成功)
  - window.name 属性在不同的页面（甚至不同域名）加载后依旧存在，并且支持非常长的 name 值（2MB）。
  - a.html、b.html，http://127.0.0.1:3000；
    c.html，http://127.0.0.1:4000

```
// a.html
<iframe
src="http://localhost:4000/c.html"
frameborder="0" onload="load()">
</iframe>
<script>
  let first = true
  // onload事件触发两次,第一次加载跨域页，并留存数据于window.name
  function load() {
    if(first) {
      // 第一次 onload(跨域页) 成功后,切换到同域代理页面
      let iframe = document.querySelector('iframe');
      iframe.src = 'http://localhost:3000/b.html';
      first = false
    } else {
      // 第二次 onload(同域页) 成功后，读取同域 window.name 中数据

console.log(iframe.contentWindow.name)
    }
  }
</script>
// b.html为中间代理页，与a.html同域，内容为空
 // c.html
<script>
  window.name = 'xxxxxxx'
</script>
```

  - 总结：通过 iframe 的 src 属性由外域转向本地域，跨域数据即由 iframe 的 window.name 从外域传递到本地域。

- document.domain + iframe(不好使)
  只能用于主域名相同的情况下，比如 a.test.com 和 b.test.com 适用于该方式

```
// http://www.domain.com/a.html
<iframe id="iframe"
src="http://child.domain.com/b.html"></iframe>
<script>
  document.domain = 'domain.com';
  var user = 'admin';
  let iframe =
document.querySelector('#iframe');
  alert(`get js data from child --->
${iframe.contentWindow.name}`);
</script>
// http://child.domain.com/b.html
<script>
  document.domain = 'domain.com';
  let name = 'tom';
  // 获取父窗口中变量
  alert('get js data from parent ---> ' +
window.parent.user);
</script>
```

总结：两个页面都通过 js 强制设置 document.domain 为主域，就实现了同域。

- location.hash + iframe(不好使)
  a 欲与 c 跨域相互通信，通过中间页 b 来实现。三个页面，不同域之间利用 iframe 的 location.hash 传值，相同域之间直接 js 访问来通信

```
 // a.html
 // a 给 c 传了一个 hash 值
<iframe
src="http://localhost:4000/c.html#iloveyou">
</iframe>
<script>
  window.onhashchange = function () {
    //检测 hash 的变化,中间介 b 帮 c 传达的
    console.log(location.hash);
  }
</script>
// b.html
<script>
  window.onhashchange = function () {
    window.parent.parent.location.hash =
location.hash
    // b 将 c 的话放在 a 的 hash 中
  }
</script>
// c.html
<script>
  window.onhashchange = function () {
    // c 拿到 a 发送来的 hash
    console.log(location.hash);
    let iframe =
document.createElement('iframe');
    // c 向 b 传达了向发送给 a 的信息
    iframe.src =
'http://localhost:3000/b.html#idontloveyou';
    document.body.appendChild(iframe)
  }
</script>
```

总结：不同域的 a 和 c ，a 直接通过 hash 向 c 传达消息；c 通过 hash 向中间者 b(a 同域) 传达要发送给 a 的消息，b 拿到 消息 放到 a 的 hash 中。

- JSONP，利用 script 标签没有跨域限制的漏洞。通过 script 指向一个需要访问的地址并提供一个回调函数来接收数据当需要通讯时。$.Ajax 中的 jsonp 采用了这个原理，使用简单且兼容性不错，但是只限于 get 请求(因为只能通过拼接 url 传递数据)，并且 jsonp 需要服务器端支持。

```
// index.html(实现简易的jsonp,并用promise包装)
function jsonp({ url, params, callback }) {
  return new Promise((resolve, reject) => {
    let script =
document.createElement("script");
    window[callback] = function (data) {
      // 回调函数其实在服务器返回响应时就调用了
      resolve(data);
      // 移除该请求脚本
      document.body.removeChild(script);
    };
    // 将回调函数加入参数
    params = { ...params, callback }; //
wd=b&callback=show
    let arrs = [];
    // 遍历所有参数
    for (let key of Object.keys(params)) {
      arrs.push(`${key}=${params[key]}`);
    }
    // 拼接参数设置为 script 的属性
    script.src = `${url}?${arrs.join("&")}`;
    // 添加到文档中发送请求
    document.body.appendChild(script);
  });
}
jsonp({
  url: "http://localhost:3000",
  params: { wd: "Iloveyou" },
  callback: "show"
}).then(data => {
  console.log(data); // show
});

// server.js
let express = require("express");
let app = express();
app.get("/", function (req, res) {
  let { wd, callback } = req.query;
  console.log(wd); // Iloveyou
  console.log(callback); // show
  // 注意，返回给script标签，浏览器直接把这部分字符串
执行
  res.end(`${callback}('我不爱你')`);
});
app.listen(3000);
```

- CORS(Cross-Origin ResourceSharing) 跨域资源共享(跨域根本解决方案)

  CORS

  使用自定义的 HTTP 头部让浏览器与服务器进行沟通，从而决定请求或是响应应是成功还是失败。 整个 CORS 通讯过程由浏览器自动完成。浏览器一旦发现 AJAX 请求跨源，就会自动添加一些附加的头信息，甚至多出一次附加的请求。

    - 简单请求，满足下列条件
      - 请求方法为以下中任意
        - GET
        - HEAD
        - POST
      - 请求头仅包含以下中任意
        - Accept
        - Accept-Language
        - Content-Language
        - Content-Type
        - DPR
        - Downlink
        - Save-Data
        - Viewport-Width
        - Width
      - Content-Type 仅属于以下中任意
        - text/plain
        - multipart/form-data
        - application/x-www-form-urlencoded

      对于简单请求，浏览器会采用先请求后判断的方式，浏览器会在请求头中增加 Origin 字段。用于向服务器说明本次请求来自哪个源（协议://域名:端口），有服务器判断是否允许这个源的访问，(个人理解：服务器判断的依据是这个请求的资源是否为用户的私有性资源，如果是需要判断这个源是否在白名单列表中，如果用户登录过了服务器会在白名单列表中添加该用户的 ip 地址，而如果访问的不是私有性资源，一般白名单设置为*，表示任意源访问都允许)。

      如果该源不在服务器允许的范围内，就返回一个正常的 HTTP 响应，浏览器判断响应头中是否包含 Access-Control-Allow-Origin 字段，如果没有，浏览器就知道服务器是不允许跨域访问的，就会抛出错误。这种错误一般无法通过状态码识别，因为 HTTP 的响应状态码有可能是 200。

      如果 Origin 在服务器的允许范围内，返回的响应状态码一般包含下面 4 个字段。

    - 相关字段

- Access-Control-Allow-Origin，必须，值为请求头 Origin 字段值,或者为 * 表示接受任意域名的请求。
- Access-Control-Allow-Credentials，是否允许发送 Cookie。默认 Cookie 不包括在 CORS 请求中。需要配合 AJAX 打开 XMLHttpRequest 对象的 withCredentials 且 Access-Control-Allow-Origin 不能设为星号，**必须指定明确的、与请求网页一致的域名**。此时 Cookie 依然遵循同源策略，只有用该服务器域名设置的 Cookie 才会上传，其他域名的 Cookie 不会上传。
- Access-Control-Allow-Methods，允许浏览器在 CORS 跨域中使用的方法。
- Access-Control-Expose-Headers，CORS 请求时，XMLHttpRequest 对象的 getResponseHeader() 如果想拿到除了默认字段外的属性，需要在这里指定。默认字段：Cache-Control、Content-Language、Content-Type、Expires、Last-Modified、Pragma。

- 复杂请求，不满足以上条件。

  对于复杂请求，浏览器会使用预检请求，询问服务器是否支持跨域请求。在正式发送资源请求之前，发送一次额外的 OPTIONS 方法的请求，询问服务器当前发送请求的 ip 地址是否在服务器的许可名单之中，并且在请求头添加一些字段用于询问可以使用哪些 HTTP 方法和头字段。

  服务器在收到预检请求后，进行与简单请求相同的检查操作，并在相应头中添加 Access-Control-Max-Age 字段，单位为秒。用于告诉浏览器在此期间可以直接发送正式请求，不用再发送预检请求。并添加一些其他字段，用于告诉浏览器跨域通信时可以传输的媒体类型、请求方法等信息。

- postMessage

  HTML5 XMLHttpRequest Level 2 中的 API。window.postMessage() 方法可以安全地实现跨源通讯，可以解决以下问题

  - 页面和其打开的新窗口的数据(跨域/同域)传递
  - 多窗口之间消息(跨域/同域)传递
  - 页面与嵌套的 iframe 消息(跨域/同域)传递
  - 语法

    ```
    otherWindow.postMessage(message, targetOrigin, [transfer]);
    ```

    - message，要发送的数据
    - targetOrigin ，决定哪些窗口能接受到消息事件，可以是'*'(无限制)或 URI。如果目标窗口的协议、主机地址或端口任意一项与 targetOrigin 不匹配不会发送。

    - transfer，一串和 message 同时传递的 Transferable 对象. 这些对象的所有权将被转移给消息的接收方，而发送一方将不再保有所有权。
    - 接收方通过给 message 添加监听器，传入一个 event，带有 data、origin(发送方窗口的 URL)、source(发送消息的窗口对象的引用) 三个属性

- 跨域

```
// http://www.domain1.com/a.html)
<iframe id="iframe"
src="http://www.domain2.com/b.html"
style="display:none;"></iframe>
<script>
var iframe =
document.getElementById('iframe');
iframe.onload = function() {
    var data = {
        name: 'aym'
    };
    // 向domain2传送跨域数据

iframe.contentWindow.postMessage(JSON.st
ringify(data),
'http://www.domain2.com');
};
// 接受domain2返回数据
window.addEventListener('message',
function(e) {
    alert('data from domain2 ---> ' +
e.data);
}, false);
</script>
// http://www.domain2.com/b.html
<script>
// 接收domain1的数据
window.addEventListener('message',
function(e) {
    alert('data from domain1 ---> ' +
e.data);
    var data = JSON.parse(e.data);
    if (data) {
        data.number = 16;
        // 处理后再发回domain1

e.source.postMessage(JSON.stringify(data
), e.origin);
    }
}, false);
</script>
```

- Websocket(应用层协议和 HTTP 一样基于 TCP)

  HTML5 持久化协议，实现了浏览器服务器全双工通信，解决跨域。

  - 连接流程

- 客户端发送 GET 请求报文，告诉服务器，我要发起 Websocket 协议，我不是 HTTP

  Upgrade: websocket
  Connection: Upgrade
- 服务器收到协议，返回一个 101 Switching Protocol，连接成功
- 接下来的通信都是 Websocket

- 与 HTTP 区别
  - 使用 Websocket 协议的连接，服务器可以主动向客户端推送信息，客户端也可以主动向服务器发送信息，属于服务器推送技术
  - HTTP 协议通信只能由客户端发起，服务端不能主动推送消息

- 跨域
  webSocket 本身不存在跨域问题，所以可以利用 webSocket 来进行非同源之间的通信。

```
// socket.html
<script>
  let socket = new
WebSocket('ws://localhost:8080');
  socket.addEventListener('open',
function() {
    socket.send("建立成功1");
  });
  socket.addEventListener('message',
function(e) {
    console.log(e.data);
  });
</script>;
// server.js
const WebSocket = require("ws");
const wss = new WebSocket.Server({ port:
8080 });
wss.on("connection", function (ws) {
  ws.on("message", function (data) {
    console.log(data);
    ws.send("建立成功2");
  });
});
```

总结：Websocket 只有在建立连接时需要借助 HTTP 协议，建立好连接后的双向通讯就与 HTTP 无关了

- Node 中间件(正向代理)
  - 代理服务器流程

- 接受客户端请求 。
- 将请求 转发给服务器。
- 拿到服务器 响应 数据。
- 将 响应 转发给客户端。

- 跨域

```
  // index.html(http://127.0.0.1:5500)
  <script
src="https://cdn.bootcss.com/jquery/3.3.
1/jquery.min.js"></script>
  <script>
    $.ajax({
      url: 'http://localhost:3000',
      type: 'post',
      data: {name: 'xiamen', password:
'123'},  // 客户端要发送的数据
      contentType:
'application/json;charset=utf-8',
      success:function(res) {
        console.log(res)
      },
      error: function(e) {
        console.log(e)
      }
    })
  </script>
  // server1.js 代理服务器
(http://localhost:3000)
  const http = require('http')
  // 第一步：接受客户端请求
  const server =
http.createServer((request, response) =>
{
    // 代理服务器，直接和浏览器直接交互，需要
设置CORS的首部字段
    response.writeHead(200, {
      'Access-Control-Allow-Origin':
'*',
      'Access-Control-Allow-Header':
'Content-Type',
      'Access-Control-Allow-Methods':
'*'
    })
    const proxyRequest = http.request(
      { // 第二步，向服务器发送请求
        host: '127.0.0.1',
        port: 4000,
        url: '/',
        method: request.method,
        headers: request.headers,
        data: request.body.data
      },
      serverResponse => {
        // 第四步，收到服务器响应
```

```
          // 第四步，收到服务器响应
          let body = ''
          serverResponse.on('data', chunk
=> {
            body += chunk
          })
          // 第五步，转发服务器响应
          serverResponse.on('end', () => {
            console.log(`The data is
${body}`)
            response.end(body)
          })
        }
      )
    })
    server.listen(3000, ()=>{
      console.log('The proxyServer is
running in 3000')
    })
    // server2.js (http://localhost:4000)
    const http = require('http')
    // 服务器要发送的数据
    const data = {title: 'fronted',
password: '123'}
    const server =
http.createServer((request, response)=>{
      if(request.url !== '/') return
      console.log(response.body.data)
      response.end(JSON.stringify(data))
    })
    server.listen(4000, () => {
      console.log('The server is running
in 4000')
    })
```

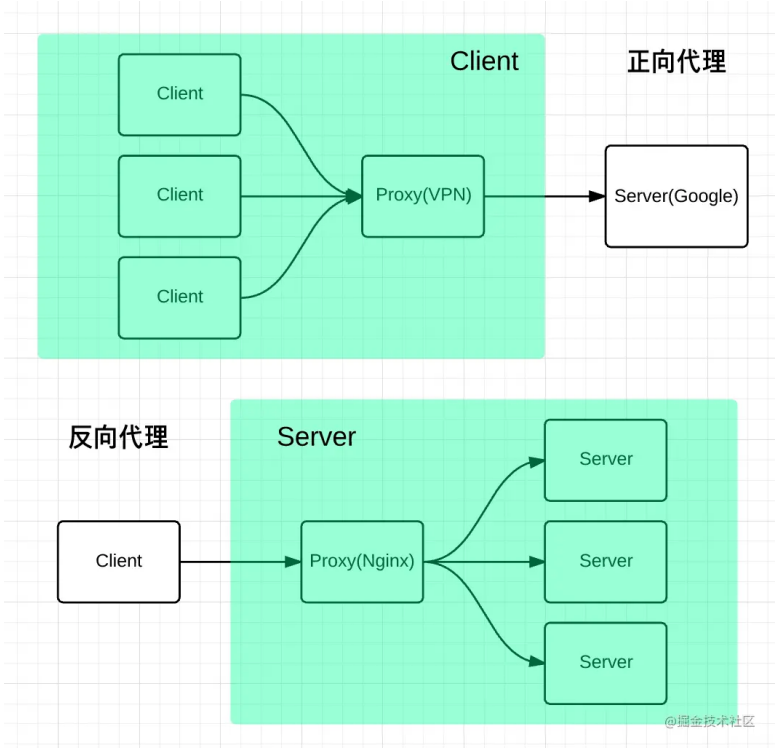
总结：同源策略是浏览器需要遵循的标准，而如果是服务器向服务器请求就无需遵循同源策略。虽然浏览器向代理服务器发送请求的过程也是跨域，但是在代理服务器中开启 CORS 因此能够实现跨域。

- nginx(反向代理)

  正向代理帮助客户端访问客户端自己访问不到的服务器，然后将结果返回给客户端。

  反向代理拿到客户端的请求，将请求转发给其他的服务器，意思就是反向代理帮其它的服务器拿到请求，然后选择一个合适的服务器，将请求转交给它。

  正向代理服务器是帮客户端做事情，而反向代理服务器是帮其它的服务器做事情。



Nginx 相当于起了一个跳板机，这个跳板机的域名也是 client.com，让客户端首先访问 client.com/api，这当然没有跨域，然后 Nginx 服务器作为反向代理，将请求转发给 server.com，当响应返回时又将响应给到客户端，这就完成整个跨域请求的过程。

重点：CORS、JSONP、Node、Nginx，了解 postMessage、WebSocket

前端鉴权

---

sesson-cookie

1. 服务端在客户端首次登录时在服务端创建 session，然后保存 session 在 redis 中，
   并给这个 session 生成一个唯一标识 sessionid，并在返回的响应中携带该 sessionid。
2. 签名，对 sessionid 进行加密处理
3. 浏览器中收到请求响应的时候就会解析响应头，将 session 保存在本地 cookie 中，并且在之后的请求中会在请求头中携带上该域名下的 cookie。
4. 服务端在接受客户端请求时会去解析请求头中 cookie 字段保存的 sessionid，然后根据这个 sessionid 去找服务端保存的该客户端的 session，判断请求是否合法。

这样做有几个缺点，用户每做一次登录认证，就会在服务端保存一次记录，以方便用户下一次请求，随着登录用户数增加，会导致服务器内存消耗大的问题。

将 sessionid 存放在 cookie 中会遭到 CSRF 攻击，基于 cookie 的内容来识别用户 cookie 的值很容易被截获用于伪造身份。

如果浏览器禁用了 cookie，一般会把 sessionid 跟在 url 参数后面重写 url。

移动端对 cookie 的支持不大好，而且 session 需要基于 cookie 实现，所以移动端一般采用 Token。

token

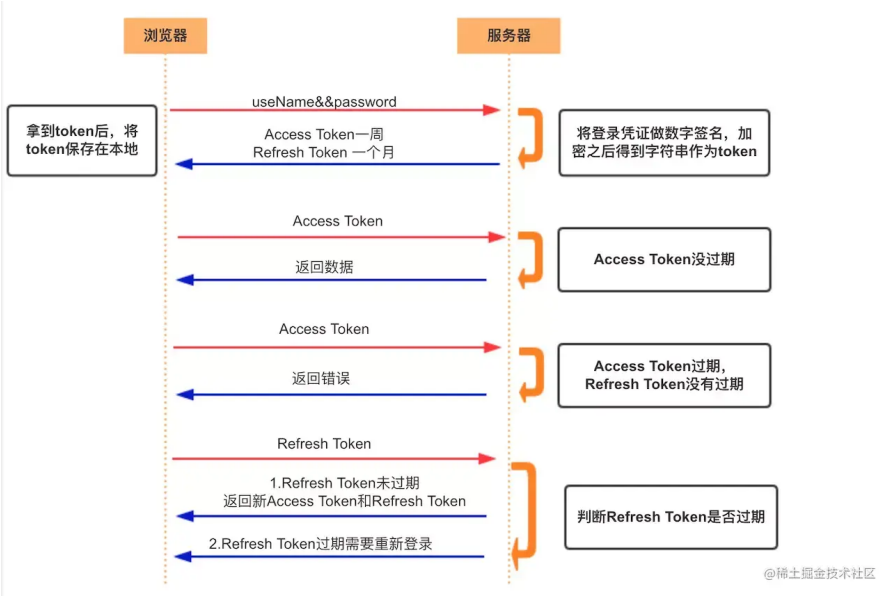
Token 原理当用户登录后，服务器生成一个 token，并在返回的响应中携带该 token 返回给客户端。客户端在 localStorage 和 Vuex 中保存，并在以后每次发送请求时携带上这个 token 请求数据。服务端收到用户请求后拿到请求头中的 Token 并进行解密，解密完成后与数据库中的用户信息对比，如果对比成功则通过认证。

1. 客户端使用用户名和密码进行登录
2. 服务端接收到请求，验证用户名和密码
3. 验证成功服务端签发一个 Token，并在返回的响应中携带这个 Token
4. 客户端接收到 Token 后把它存在在 Cookie 或 localStorage 和 Vuex 中
5. 客户端每次向服务端请求资源的时候都需要携带服务端签发的 Token
6. 服务端收到请求，去验证客户端请求携带的 Token，如果验证成功就向客户端返回请求的数据。如果不合法就返回 401 鉴权失败。

Token 由应用管理可以避开同源策略，并且 Token 也可以避免 CSRF 攻击

Token 是一种时间换空间的认证方式，服务端不用存放 Token 数据，用解析 Token 的计算时间换取 session 的存储空间。

Token 的生存时间比较短，但如果每次一过期就让用户重新登录用户体验不大好，一般通过 Refresh Token 去重新获取 Access Token。在请求时如果服务器发现用户的 Token 过期就会返回一个错误，提示客户端 Token 过期，这时候客户端会再次发送一个请求携带 Refresh Token，服务器校验 Refresh Token 未过期这时候就会返回新的 Access Token 和 Refresh Token。



JWT

JWT 是一种常用的 Token

JWT 包括三个字段分别是 Header 头部、Payload 负载、Signature 签名

头部是一个 JSON 对象，包括签名的算法 alg、以及 Token 令牌类型 typ，JWT 令牌就写 JWT。头部的 JSON 对象会通过 Base64URL 算法转换成字符串。

负载也是一个 JSON 对象，用来存放实际要传输的数据，比如过期时间、生效时间和编号，同样也会采用 Base64 算法转换成字符串。JWT 默认是不加密的，所以不能存放秘密信息。

签名是对前面两部分的签名，防止数据篡改。服务器需要有一个只有自己知道的秘钥，然后通过 头部字段中指定的签名算法对前面两部分进行加密。将下面加密后的内容作为签名。

```
HMACSHA256(
  base64UrlEncode(header) + "." +
  base64UrlEncode(payload),
  secret)
```

JWT 一般都放在请求头的 Authorization 中，这样既可以避开同源策略也能够避免 CSRF 攻击。

首次登录时，后端服务器判断用户账号密码正确之后，根据用户 id、用户名、定义好的秘钥、过期时间生成 token ，返回给前端；
前端拿到后端返回的 token ,存储在 localStroage 和 Vuex 里；
前端每次路由跳转，判断 localStroage 有无 token ，没有则跳转到登录页，有则请求获取用户信息，改变登录状态；
每次请求接口，在 Axios 请求头里携带 token;

后端接口判断请求头有无 token，没有或者 token 过期，返回 401；
前端得到 401 状态码，重定向到登录页面。

后端可以通过维护一份 Token 黑名单，Token 失效就将该 Token 加入到黑名单中。

服务端主动让 token 失效的方法

后端在每次签发 JWT 的时候在负载部分加入一个随机字符的字段 tokenid。

在服务端的缓存中保存一份黑名单。如果用户的 jwt 重置密码等需要作废已经签发但未过期的 jwt 时，就将该之前用户的 tokenid 存入到黑名单中。并分配给他一个新的 tokenid。

存入到黑名单中的 tokenid 会设置一个过期时间，过期后自动从黑名单中删除。

服务器做 JWT 校验的时候，除了校验过期时间，还要查询内存中的黑名单列表。若在黑名单中，则判定该 JWT 为失效。