一次意外的百度二面

自我介绍

首先上来是一段自我介绍，主要围绕前端的开发经验进行了自我介绍。

看简历

之后就是看简历，面试官主要围绕项目进行询问。

技术难点

这个主要是根据个人经历询问，在此就不赘述了。但是针对技术难点，面试官往往会进行深挖。

工作过程中有没有遇到一些问题并且提出解决方案

大文件上传切割后如何上传的

可以看一下**这篇文章**，详细记录了针对大文件上传的一些追问。

讲一下xss和scrf

XSS 攻击

Cross-Site Scripting（跨站脚本攻击）简称 XSS，是一种**「代码注入攻击」**。攻击者通过在目标网站上注入恶意脚本，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 等，进而危害数据安全。

XSS 的本质是：恶意代码未经过滤，与网站正常的代码混在一起；浏览器无法分辨哪些脚本是可信的，导致恶意脚本被执行。

而由于直接在用户的终端执行，恶意代码能够直接获取用户的信息，或者利用这些信息冒充用户向网站发起攻击者定义的请求。

在部分情况下，由于输入的限制，注入的恶意脚本比较短。但可以通过引入外部的脚本，并由浏览器执行，来完成比较复杂的攻击策略。

XSS 攻击可分为存储型、反射型和 DOM 型三种：

存储型 XSS

存储型 XSS 的攻击步骤：

1. 攻击者将恶意代码提交到目标网站的数据库中。
2. 用户打开目标网站时，网站服务端将恶意代码从数据库取出，拼接在 HTML 中返回给浏览器。
3. 用户浏览器接收到响应后解析执行，混在其中的恶意代码也被执行。
4. 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。

这种攻击常见于带有用户保存数据的网站功能，如论坛发帖、商品评论、用户私信等。

反射型 XSS

反射型 XSS 的攻击步骤：

1. 攻击者构造出特殊的 URL，其中包含恶意代码。
2. 用户打开带有恶意代码的 URL 时，网站服务端将恶意代码从 URL 中取出，拼接在 HTML 中返回给浏览器。
3. 用户浏览器接收到响应后解析执行，混在其中的恶意代码也被执行。
4. 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。

反射型 XSS 跟存储型 XSS 的区别是：存储型 XSS 的恶意代码存在数据库里，反射型 XSS 的恶意代码存在 URL 里。

反射型 XSS 漏洞常见于通过 URL 传递参数的功能，如网站搜索、跳转等。

由于需要用户主动打开恶意的 URL 才能生效，攻击者往往会结合多种手段诱导用户点击。

POST 的内容也可以触发反射型 XSS，只不过其触发条件比较苛刻（需要构造表单提交页面，并引导用户点击），所以非常少见。

DOM 型 XSS

DOM 型 XSS 的攻击步骤：

1. 攻击者构造出特殊的 URL，其中包含恶意代码。
2. 用户打开带有恶意代码的 URL。
3. 用户浏览器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。
4. 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。

DOM 型 XSS 跟前两种 XSS 的区别：DOM 型 XSS 攻击中，取出和执行恶意代码由浏览器端完成，属于前端 JavaScript 自身的安全漏洞，而其他两种 XSS 都属于服务端的安全漏洞。

如何防御

XSS 攻击有两大要素：

1. 攻击者提交恶意代码。
2. 浏览器执行恶意代码。

输入过滤

对于明确的输入类型，例如数字、URL、电话号码、邮件地址等等内容，在用户输入和写入数据库前进行输入过滤还是必要的。

输入侧过滤能够在某些情况下解决特定的 XSS 问题，但会引入很大的不确定性和乱码问题。
我们举一个例子，一个正常的用户输入了 `5 < 7` 这个内容，在写入数据库前，被转义，变成了 `5 < 7`。

问题是：在提交阶段，我们并不确定内容要输出到哪里。

这里的“并不确定内容要输出到哪里”有两层含义：

1. 用户的输入内容可能同时提供给前端和客户端，而一旦经过了 `escapeHTML()`，客户端显示的内容就变成了乱码(`5 < 7`)。
2. 在前端中，不同的位置所需的编码也不同。

- 当 `5 < 7` 作为 HTML 拼接页面时，可以正常显示：

```ini
<div title="comment">5 &lt; 7</div>
```

- 当 `5 < 7` 通过 Ajax 返回，然后赋值给 JavaScript 的变量时，前端得到的字符串就是转义后的字符。这个内容不能直接用于 Vue 等模板的展示，也不能直接用于内容长度计算。不能用于标题、alert 等。

既然输入过滤并非完全可靠，我们就要通过“防止浏览器执行恶意代码”来防范 XSS。这部分分为两类：

- 防止 HTML 中出现注入。
- 防止 JavaScript 执行时，执行恶意代码。

预防存储型和反射型 XSS 攻击

存储型和反射型 XSS 都是在服务端取出恶意代码后，插入到响应 HTML 里的，攻击者刻意编写的“数据”被内嵌到“代码”中，被浏览器所执行。

预防这两种漏洞，有两种常见做法：

- 改成纯前端渲染，把代码和数据分隔开。
- 对 HTML 做充分转义。

纯前端渲染 5

纯前端渲染的过程：

1. 浏览器先加载一个静态 HTML，此 HTML 中不包含任何跟业务相关的数据。
2. 然后浏览器执行 HTML 中的 JavaScript。
3. JavaScript 通过 Ajax 加载业务数据，调用 DOM API 更新到页面上。

但纯前端渲染还需注意避免 DOM 型 XSS 漏洞（例如 `onload` 事件和 `href` 中的 `javascript:xxx` 等

转义 HTML 5

如果拼接 HTML 是必要的，就需要采用合适的转义库，对 HTML 模板各处插入点进行充分的转义。

常用的模板引擎，如 doT.js、ejs、FreeMarker 等，对于 HTML 转义通常只有一个规则，就是把 `& < > " ' /` 这几个字符转义掉，确实能起到一定的 XSS 防护作用，但并不完善

预防 DOM 型 XSS 攻击

DOM 型 XSS 攻击，实际上就是网站前端 JavaScript 代码本身不够严谨，把不可信的数据当作代码执行了。

在使用 `.innerHTML`、`.outerHTML`、`document.write()` 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。

如果用 Vue/React 技术栈，并且不使用 `v-html`/`dangerouslySetInnerHTML` 功能，就在前端 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。如果不可信的数据拼接到字符串中传递给这些 API，很容易产生安全隐患，请务必避免。

其他安全措施

- HTTP-only Cookie: 禁止 JavaScript 读取某些敏感 Cookie，攻击者完成 XSS 注入后也无法窃取此 Cookie。
- 验证码：防止脚本冒充用户提交危险操作。

CSRF

「跨站请求伪造」（英语：Cross-site request forgery），也被称为 **「one-click attack」**或者 **「session riding」**，通常缩写为 **「CSRF」** 或者 **「XSRF」**， 是一种挟制用户在当前已登录的 Web 应用程序上执行非本意的操作的攻击方法。 跟XSS相比，**「XSS」** 利用的是用户对指定网站的信任，CSRF 利用的是网站对用户网页浏览器的信任。

简单点说，CSRF 就是利用用户的登录态发起恶意请求。

CSRF 分为GET型，POST型(利用自动提交的表单)和链接型（不常见，只有打开链接才会触发）

一个典型的CSRF攻击有着如下的流程：

- 受害者登录a.com，并保留了登录凭证（Cookie）。
- 攻击者引诱受害者访问了b.com。
- b.com 向 a.com 发送了一个请求：a.com/act=xx。浏览器会默认携带a.com的Cookie。
- a.com接收到请求后，对请求进行验证，并确认是受害者的凭证，误以为是受害者自己发送的请求。
- a.com以受害者的名义执行了act=xx。
- 攻击完成，攻击者在受害者不知情的情况下，冒充受害者，让a.com执行了自己定义的操作。

如何防御

同源检测

既然CSRF大多来自第三方网站，那么我们就直接禁止外域（或者不受信任的域名）对我们发起请求。

那么问题来了，我们如何判断请求是否来自外域呢？

在HTTP协议中，每一个异步请求都会携带两个Header，用于标记来源域名：

- Origin Header
- Referer Header

这两个Header在浏览器发起请求时，大多数情况会自动带上，并且不能由前端自定义内容。服务器可以通过解析这两个Header中的域名，确定请求的来源域。

如果Origin存在，那么直接使用Origin中的字段确认来源域名就可以。

但是Origin在以下两种情况下并不存在：

- **「IE11同源策略：」** IE 11 不会在跨站CORS请求上添加Origin标头，Referer头将仍然是唯一的标识。最根本原因是因为IE 11对同源的定义和其他浏览器有不同，有两个主要的区别，可以参考**MDN Same-origin_policy#IE_Exceptions****
- **「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。对于302重定向的情况来说都是定向到新的服务器上的URL，因此浏览器不想将Origin泄漏到新的服务器上。

origin不存在时使用Referer Header确定来源域名：

根据HTTP协议，在HTTP头中有一个字段叫Referer，记录了该HTTP请求的来源地址。 对于Ajax请求，图片和script等资源请求，Referer为发起请求的页面地址。对于页面跳转，Referer为打开页面历史记录的前一个页面地址。因此我们使用Referer中链接的Origin部分可以得知请求的来源域名。

这种方法并非万无一失，Referer的值是由浏览器提供的，虽然HTTP协议上有明确的要求，但是每个浏览器对于Referer的具体实现可能有差别，并不能保证浏览器自身没有安全漏洞。使用验证 Referer 值的方法，就是把安全性都依赖于第三方（即浏览器）来保障，从理论上来讲，这样并不是很安全。在部分情况下，攻击者可以隐藏，甚至修改自己请求的Referer。

2014年，W3C的Web应用安全工作组发布了Referrer Policy草案，对浏览器该如何发送Referer做了详细的规定。截止现在新版浏览器大部分已经支持了这份草案，我们终于可以灵活地控制自己网站的Referer策略了。新版的Referrer Policy规定了五种Referer策略：No Referrer、No Referrer When Downgrade、Origin Only、Origin When Cross-origin、和 Unsafe URL。之前就存在的三种策略：never、default和always，在新标准里换了个名称。。他们的对应关系如下：

策略名称	属性值（新）	属性值（旧）
No Referrer	no-Referrer	never
No Referrer When Downgrade	no-Referrer-when-downgrade	default
Origin Only	(same or strict) origin	origin
Origin When Cross Origin	(strict) origin-when-crossorigin	-
Unsafe URL	unsafe-url	always

根据上面的表格因此需要把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；跨域访问则不携带Referer。

设置Referrer Policy的方法有三种：

1. 在CSP设置
2. 页面头部增加meta标签

3. a标签增加referrerpolicy属性

「上面说的这些比较多，但我们可以知道一个问题：攻击者可以在自己的请求中隐藏Referer。」

另外在以下情况下Referer没有或者不可信：

1. IE6、7下使用window.location.href=url进行界面的跳转，会丢失Referer。
2. IE6、7下使用window.open，也会缺失Referer。
3. HTTPS页面跳转到HTTP页面，所有浏览器Referer都丢失。
4. 点击Flash上到达另外一个网站的时候，Referer的情况就比较杂乱，不太可信。

如果Origin和Referer都不存在，建议直接进行阻止，特别是如果您没有使用随机CSRF Token（参考下方）作为第二次检查。

「综上所述：同源验证是一个相对简单的防范方法，能够防范绝大多数的CSRF攻击。但这并不是万无一失的，对于安全性要求较高，或者有较多用户输入内容的网站，我们就要对关键的接口做额外的防护措施。」

CSRF Token

前面讲到CSRF的另一个特征是，攻击者无法直接窃取到用户的信息（Cookie，Header，网站内容等），仅仅是冒用Cookie中的信息。

而CSRF攻击之所以能够成功，是因为服务器误把攻击者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，来把正常的请求和攻击的请求区分开，也可以防范CSRF的攻击。

原理 5

CSRF Token的防护策略分为三个步骤：

「1. 将CSRF Token输出到页面中」

首先，用户打开页面的时候，服务器需要给这个用户生成一个Token，该Token通过加密算法对数据进行加密，一般Token都包括随机字符串和时间戳的组合，显然在提交时Token不能再放在Cookie中了，否则又会被攻击者冒用。因此，为了安全起见Token最好还是存在服务器的Session中，之后在每次页面加载时，使用JS遍历整个DOM树，对于DOM中所有的a和form标签后加入Token。这样可以解决大部分的请求，但是对于在页面加载之后动态生成的HTML代码，这种方法就没有作用，还需要程序员在编码时手动添加Token。

「2. 页面提交的请求携带这个Token」

对于GET请求，Token将附在请求地址之后，这样URL 就变成 **http://url/?csrftoken=tokenvalue%E3%80%82** 而对于 POST 请求来说，要在 form 的最后加上：

```xml
<input type="hidden" name="csrftoken" value="tokenvalue"/>
```

这样，就把Token以参数的形式加入请求了。

「3. 服务器验证Token是否正确」

当用户从客户端得到了Token，再次提交给服务器的时候，服务器需要判断Token的有效性，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。

这种方法要比之前检查Referer或者Origin要安全一些，Token可以在产生并放于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，但这种方法的比较麻烦的在于如何把Token以参数的形式加入请求。

「Token是一个比较有效的CSRF防护方法，只要页面没有XSS漏洞泄露Token，那么接口的CSRF攻击就无法成功。」

「验证码和密码其实也可以起到CSRF Token的作用哦，而且更安全。」

Samesite Cookie属性

防止CSRF攻击的办法已经有上面的预防措施。为了从源头上解决这个问题，Google起草了一份草案来改进HTTP协议，那就是为Set-Cookie响应头新增Samesite属性，它用来标明这个Cookie是个“同站 Cookie”，同站Cookie只能作为第一方Cookie，不能作为第三方Cookie，Samesite 有两个属性值，分别是 Strict 和 Lax，下面分别讲解：

Samesite=Strict 5

这种称为严格模式，表明这个 Cookie 在任何情况下都不可能作为第三方 Cookie，绝无例外。比如说 b.com 设置了如下 Cookie：

```ini
Set-Cookie: foo=1; Samesite=Strict
Set-Cookie: bar=2; Samesite=Lax
Set-Cookie: baz=3
```

我们在 a.com 下发起对 b.com 的任意请求，foo 这个 Cookie 都不会被包含在 Cookie 请求头中，但 bar 会。举个实际的例子就是，假如淘宝网站用来识别用户登录与否的 Cookie 被设置成了 Samesite=Strict，那么用户从百度搜索页面甚至天猫页面的链接点击进入淘宝后，淘宝都不会是登录状态，因为淘宝的服务器不会接受到那个 Cookie，其它网站发起的对淘宝的任意请求都不会带上那个 Cookie。

Samesite=Lax 5

这种称为宽松模式，比 Strict 放宽了点限制：假如这个请求是这种请求（改变了当前页面或者打开了新页面）且同时是个GET请求，则这个Cookie可以作为第三方Cookie。比如说 b.com 设置了如下Cookie：

```ini
Set-Cookie: foo=1; Samesite=Strict
Set-Cookie: bar=2; Samesite=Lax
Set-Cookie: baz=3
```

当用户从 a.com 点击链接进入 b.com 时，foo 这个 Cookie 不会被包含在 Cookie 请求头中，但 bar 和 baz 会，也就是说用户在不同网站之间通过链接跳转是不受影响了。但假如这个请求是从 a.com 发起的对 b.com 的异步请求，或者页面跳转是通过表单的 post 提交触发的，则bar也不会发送。

我们应该如何使用SamesiteCookie 5

如果SamesiteCookie被设置为Strict，浏览器在任何跨域请求中都不会携带Cookie，新标签重新打开也不携带，所以说CSRF攻击基本没有机会。

但是跳转子域名或者是新标签重新打开刚登陆的网站，之前的Cookie都不会存在。尤其是有登录的网站，那么我们新打开一个标签进入，或者跳转到子域名的网站，都需要重新登录。对于用户来讲，可能体验不会很好。

如果SamesiteCookie被设置为Lax，那么其他网站通过页面跳转过来的时候可以使用Cookie，可以保障外域连接打开页面时用户的登录状态。但相应的，其安全性也比较低。

另外一个问题是Samesite的兼容性不是很好，现阶段除了从新版Chrome和Firefox支持以外，Safari以及iOS Safari都还不支持，现阶段看来暂时还不能普及。

而且，SamesiteCookie目前有一个致命的缺陷：不支持子域。例如，种在topic.a.com下的Cookie，并不能使用a.com下种植的SamesiteCookie。这就导致了当我们网站有多个子域名时，不能使用SamesiteCookie在主域名存储用户登录信息。每个子域名都需要用户重新登录一次。

总之，SamesiteCookie是一个可能替代同源验证的方案，但目前还并不成熟，其应用场景有待观望。

总结

简单总结一下上文的防护策略：

- CSRF自动防御策略：同源检测（Origin 和 Referer 验证）。
- CSRF主动防御措施：Token验证 或者 双重Cookie验证 以及配合Samesite Cookie。
- 保证页面的幂等性，后端接口不要在GET页面中做用户操作。

xss是前端做还是后端做

防范存储型和反射型 XSS 是后端的责任。而 DOM 型 XSS 攻击不发生在后端，是前端的责任。防范 XSS 是需要后端和前端共同参与的系统工程。

转义应该在输出 HTML 时进行，而不是在提交用户输入时

安全问题主要参考**这两篇文章**

说一下前端性能优化

性能优化在一面中问到过，可以直接**跳转查看详情**

针对这个问题进行了如下询问

什么场景引起重绘，什么场景引起重排

重绘和回流是渲染步骤中的一小节，但是这两个步骤对于性能影响很大。

- 重绘是当节点需要更改外观而不会影响布局的，比如改变 `color` 会引起重绘
- 回流是布局或者几何属性需要改变就会引起回流。

回流必定会发生重绘，重绘不一定会引发回流。回流所需的成本比重绘高的多，改变深层次的节点很可能导致父节点的一系列回流。

业务中必须要重绘重排如何进行优化

- 增加多个节点使用documentFragment
- 使用 `visibility` 替换 `display: none` ，因为前者只会引起重绘，后者会引发回流（改变了布局）
- 将频繁运行的动画变为图层，图层能够阻止该节点回流影响别的元素。比如对于 `video` 标签，浏览器会自动将该节点变为图层。
- 把 DOM 离线后修改，比如：先把 DOM 给 `display:none` (有一次 Reflow)，然后你修改 100 次，然后再把它显示出来

vue如何实现多次渲染的优化

异步更新视图：只要观察到数据变化，Vue将开启一个队列，并缓冲在同一事件循环中发生的所有数据变化。如果同一个watcher被多次触发，只会被推入到队列中一次。这种在缓冲时去除重复数据对于避免不必要的计算和DOM操作上非常重要。然后，在下一个事件循环在“tick”中，Vue刷新队列并执行实际（已去重）的工作

代码题

```js
const result = ['1', '3', '3'].map(parseInt);
// 这里会打印出什么呢?
console.log( result );
```

map的第一个参数callback的入参依次是：

- 当前值
- 当前值下标
- 当前数组

parseInt(解析一个字符串并返回指定基数的十进制整数)的入参依次是：

- 要转化的数字
- 第一个参数的基数：
  - 是一个2-36的数字，如果是1或者大于36会返回NaN，假如指定 0 或未指定，基数将会根据字符串的值进行推算
  - 如果第一个参数按照第二个定义的基数是无效的也会返回NaN

所以上述代码的返回结果为：

```js
1,NaN,NaN
```

代码题

```js
function changeArg(x) { x = 200 }

let num = 100
changeArg(num)
console.log('changeArg num', num) // 100

let obj = { name: '双越' }
changeArg(obj)
console.log('changeArg obj', obj) // { name: '双越' }

function changeArgProp(x) {
    x.name = '张三'
}
changeArgProp(obj)
console.log('changeArgProp obj', obj) //{name:'张三'}
```

js中所有函数传递都是**「按值传递的」**，不会按引用传递。所谓的值，就是指直接保存在变量上的值，如果把对象作为参数传递，那么这个值就是这个对象的引用，而不是对象本身。这里实

际上是一个隐式的赋值过程，所以给函数传递参数时，相当于**「从一个变量赋值到另一个变量」**。

其实可以理解为默认存在一个复制的过程：

```js
changeArg(num) // 可以看做:x=num

changeArg(obj) // 可以看做:x=obj
```

代码题：实现一个函数，入参是一个fn，延迟5s执行，并且拿到返回值

```js
function sleep(fn) {
    return new Promise(resolve => {
        setTimeout(() => {
            let res = fn()
            resolve(res)
        }, 5000)
    })
}

function f() {
    return 1
}

console.log(Date.now())
sleep(f).then(res => {
    console.log(res, Date.now())
})
```