22.2 CSRF

> 涉及面试题：什么是 CSRF 攻击？如何防范 CSRF 攻击？

> CSRF 中文名为跨站请求伪造 。原理就是攻击者构造出一个后端请求地址，诱导用户点击或者通过某些途径自动发起请求 。如果用户是在登录状态下的话，后端就以为是用户在操作，从而进行相应的逻辑。

举个例子，假设网站中有一个通过 GET 请求提交用户评论的接口 ，那么攻击者就可以在钓鱼网站中加入一个图片， 图片的地址就是评论接口

```html
<img src="http://www.domain.com/xxx?comment= 'attack '"/>
```

> 那么你是否会想到使用 POST 方式提交请求是不是就没有这个问题了呢？其实并不是，使用这种方式也不是百分百安全的， 攻击者同样可以诱导用户进入某个页面，在页面中通过表单提交 POST 请求。

如何防御

- Get 请求不对数据进行修改
- 不让第三方网站访问到用户 Cookie
- 阻止第三方网站请求接口
- 请求时附带验证信息， 比如验证码或者 Token

SameSite

> 可以对 Cookie 设置 SameSite 属性 。该属性表示 Cookie 不随着跨域请求发送， 可以很大程度减少 CSRF 的攻击，但是该属性目前并不是所有浏览器都兼容。

验证 Referer

> 对于需要防范 CSRF 的请求， 我们可以通过验证 Referer 来判断该请求是否为第三方网站发起的。

Token

> 服务器下发一个随机 Token ，每次发起请求时将 Token 携带上， 服务器验证 Token 是否有效

22.3 点击劫持

> 涉及面试题：什么是点击劫持？如何防范点击劫持？

> 点击劫持是一种视觉欺骗的攻击手段 。攻击者将需要攻击的网站通过 iframe 嵌套的方式嵌入自己的网页中， 并将 iframe 设置为透明，在页面中透出一个按钮诱导用户点击

> 对于这种攻击方式，推荐防御的方法有两种

1. X-FRAME-OPTIONS

> X-FRAME-OPTIONS 是一个 HTTP 响应头，在现代浏览器有一个很好的支持 。这个 HTTP 响应头 就是为了防御用 iframe 嵌套的点击劫持攻击。

> 该响应头有三个值可选，分别是

- DENY ，表示页面不允许通过 iframe 的方式展示
- SAMEORIGIN ，表示页面可以在相同域名下通过 iframe 的方式展示
- ALLOW-FROM ，表示页面可以在指定来源的 iframe 中展示

2. JS 防御

> 对于某些远古浏览器来说， 并不能支持上面的这种方式，那我们只有通过 JS 的方式来防御点击劫持了。

```html
<head>
  <style id="click-jack">
    html {
      display: none !important;
    }
  </style>
</head>
<body>
  <script>
    if (self == top) {
      var style = document.getElementById( 'click-jack')
      document.body.removeChild(style)
    } else {
      top.location = self.location
    }
  </script>
</body>
```

> 以上代码的作用就是当通过 `iframe` 的方式加载页面时， 攻击者的网页直接不显示所有内容了

23 从 V8 中看 JS 性能优化

> 注意：该知识点属于性能优化领域。

23.1 测试性能工具

> `Chrome` 已经提供了一个大而全的性能测试工具 `Audits`

> 点我们点击 `Audits` 后， 可以看到如下的界面

> 在这个界面中， 我们可以选择想测试的功能然后点击 `Run audits` ， 工具就会自动运行帮助我们测试问题并且给出一个完整的报告

> 上图是给掘金首页测试性能后给出的一个报告， 可以看到报告中分别为性能、体验 、SEO 都给出了打分， 并且每一个指标都有详细的评估

评估结束后， 工具还提供了一些建议便于我们提高这个指标的分数

我们只需要一条条根据建议去优化性能即可。

> 除了 `Audits` 工具之外， 还有一个 `Performance` 工具也可以供我们使用。

> 在这张图中， 我们可以详细的看到每个时间段中浏览器在处理什么事情， 哪个过程最消耗时间，便于我们更加详细的了解性能瓶颈

23.2 JS 性能优化

> `JS` 是编译型还是解释型语言其实并不固定 。首先 `JS` 需要有引擎才能运行起来， 无论是浏览器还是在 `Node` 中， 这是解释型语言的特性 。但是在 V8 引擎下， 又引入了 `TurboFan` 编译器，他会在特定的情况下进行优化，将代码编译成执行效率更高的 `Machine Code` ， 当然这个编译器并不是 `JS` 必须需要的， 只是为了提高代码执行性能，所以总的来说 `JS` 更偏向于解释型语言。

那么这一小节的内容主要会针对于 `Chrome` 的 `V8` 引擎来讲解。

> 在这一过程中， JS 代码首先会解析为抽象语法树（ AST ）， 然后会通过解释器或者编译器转化为 Bytecode 或者 Machine Code

> 从上图中我们可以发现， JS 会首先被解析为 AST ，解析的过程其实是略慢的 。代码越多，解析的过程也就耗费越长， 这也是我们需要压缩代码的原因之一 。另外一种减少解析时间的方式是预解析，会作用于未执行的函数， 这个我们下面再谈

这里需要注意一点，对于函数来说，应该尽可能避免声明嵌套函数（类也是函数)， 因为这样会造成函数的重复解析

```js
function test1() {
  // 会被重复解析
  function test2() {}
}
```

> 然后 Ignition 负责将 AST 转化为 Bytecode ， TurboFan 负责编译出优化后的 Machine Code ， 并且 Machine Code 在执行效率上优于 Bytecode

> 那么我们就产生了一个疑问，什么情况下代码会编译为 Machine Code ？

> JS 是一门动态类型的语言， 并且还有一大堆的规则 。简单的加法运算代码，内部就需要考虑好几种规则， 比如数字相加 、字符串相加 、对象和字符串相加等等 。这样的情况也就势必导致了内部要增加很多判断逻辑， 降低运行效率。

```js
function test(x) {
  return x + x
}

test(1)
test(2)
```

```
test(3)
test(4)
```

- 对于以上代码来说， 如果一个函数被多次调用并且参数一直传入 number 类型，那么 V8 就会认为该段代码可以编译为 Machine Code ， 因为你固定了类型，不需要再执行很多判断逻辑了。
- 但是如果一旦我们传入的参数类型改变，那么 Machine Code 就会被 DeOptimized 为 Bytecode ， 这样就有性能上的一个损耗了 。所以如果我们希望代码能多的编译为 Machine Code 并且 DeOptimized 的次数减少，就应该尽可能保证传入的类型一致。
- 那么你可能会有一个疑问， 到底优化前后有多少的提升呢，接下来我们就来实践测试一下到底有多少的提升

```js
const { performance, PerformanceObserver } = require('perf_hooks')

function test(x) {
  return x + x
}
// node 10 中才有 PerformanceObserver
// 在这之前的 node 版本可以直接使用 performance 中的 API
const obs = new PerformanceObserver((list, observer) => {
  console.log(list.getEntries())
  observer.disconnect()
})
obs.observe({ entryTypes: ['measure'], buffered: true })

performance.mark('start')

let number = 10000000
// 不优化代码
%NeverOptimizeFunction(test)

while (number--) {
  test(1)
}

performance.mark('end')
performance.measure('test', 'start', 'end')
```

> 以上代码中我们使用了 performance API ， 这个 API 在性能测试上十分好用 。不仅可以用来测量代码的执行时间， 还能用来测量各种网络连接中的时间消耗等等， 并且这个 API 也可以在浏览器中使

> 从上图中我们可以发现，优化过的代码执行时间只需要 9ms ，但是不优化过的代码执行时间却是前者的二十倍， 已经接近 200ms 了 。在这个案例中， 我相信大家已经看到了 V8 的性能优化到底有多强， 只需要我们符合一定的规则书写代码， 引擎底层就能帮助我们自动优化代码。

> 另外，编译器还有个骚操作 Lazy-Compile ， 当函数没有被执行的时候，会对函数进行一次预解析， 直到代码被执行以后才会被解析编译 。对于上述代码来说， test 函数需要被预解析一次，然后在调用的时候再被解析编译 。但是对于这种函数马上就被调用的情况来说，预解析这个过程其实是多余的，那么有什么办法能够让代码不被预解析呢?

```js
(function test(obj) {
  return x + x
})
```

> 但是不可能我们为了性能优化， 给所有的函数都去套上括号， 并且也不是所有函数都需要这样做 。我们可以通过 optimize-js 实现这个功能， 这个库会分析一些函数的使用情况，然后给需要的函数添加括号， 当然这个库很久没人维护了， 如果需要使用的话， 还是需要测试过相关内容的。

其实很简单， 我们只需要给函数套上括号就可以了

24 性能优化

> 总的来说性能优化这个领域的很多内容都很碎片化， 这一章节我们将来学习这些碎片化的内容。

24.1 图片优化

计算图片大小

> 对于一张 100 * 100 像素的图片来说， 图像上有 10000 个像素点， 如果每个像素的值是 RGBA 存储的话，那么也就是说每个像素有 4 个通道，每个通道 1 个字节（ 8 位 = 1 个字节)， 所以该图片大小大概为 39KB （ 10000 * 1 * 4 / 1024 ）。

- 但是在实际项目中，一 张图片可能并不需要使用那么多颜色去显示， 我们可以通过减少每个像素的调色板来相应缩小图片的大小 。
- 了解了如何计算图片大小的知识，那么对于如何优化图片，想必大家已经有 2 个思路了：

1. 减少像素点
2. 减少每个像素点能够显示的颜色

24.2 图片加载优化

- 不用图片 。很多时候会使用到很多修饰类图片， 其实这类修饰图片完全可以用 CSS 去代替。
- 对于移动端来说， 屏幕宽度就那么点， 完全没有必要去加载原图浪费带宽 。一般图片都用 CDN 加载， 可以计算出适配屏幕的宽度，然后去请求相应裁剪好的图片。
- 小图使用 base64 格式
- 将多个图标文件整合到一张图片中（雪碧图)
- 选择正确的图片格式：
  - 对于能够显示 WebP 格式的浏览器尽量使用 WebP 格式 。因为 WebP 格式具有更好的图像数据压缩算法， 能带来更小的图片体积， 而且拥有肉眼识别无差异的图像质量，缺点就是兼容性并不好
  - 小图使用 PNG ， 其实对于大部分图标这类图片， 完全可以使用 SVG 代替
  - 照片使用 JPEG

24.3 DNS 预解析

> DNS 解析也是需要时间的， 可以通过预解析的方式来预先获得域名所对应的 IP 。

```html
<link rel="dns-prefetch" href="//blog.poetries.top">
```

24.4 节流

> 考虑一个场景， 滚动事件中会发起网络请求，但是我们并不希望用户在滚动过程中一直发起请求， 而是隔一段时间发起一次，对于这种情况我们就可以使用节流。

理解了节流的用途， 我们就来实现下这个函数

```js
// func是用户传入需要防抖的函数
// wait是等待时间
const throttle = (func, wait = 50) => {
  // 上一次执行该函数的时间
  let lastTime = 0
  return function(...args) {
    // 当前时间
    let now = +new Date()
    // 将当前时间和上一次执行函数时间对比
    // 如果差值大于设置的等待时间就执行函数
    if (now - lastTime > wait) {
      lastTime = now
      func.apply(this, args)
    }
  }
}

setInterval(
  throttle(() => {
    console.log(1)
  }, 500),
  1
)
```

24.5 防抖

> 考虑一个场景，有一个按钮点击会触发网络请求，但是我们并不希望每次点击都发起网络请求， 而是当用户点击按钮一段时间后没有再次点击的情况才去发起网络请求，对于这种情况我们就可以使用防抖。

理解了防抖的用途， 我们就来实现下这个函数

```js
// func是用户传入需要防抖的函数
// wait是等待时间
const debounce = (func, wait = 50) => {
  // 缓存一个定时器id
  let timer = 0
  // 这里返回的函数是每次用户实际调用的防抖函数
  // 如果已经设定过定时器了就清空上一次的定时器
  // 开始一个新的定时器，延迟执行用户传入的方法
  return function(...args) {
    if (timer) clearTimeout(timer)
    timer = setTimeout(() => {
      func.apply(this, args)
    }, wait)
  }
}
```

24.6 预加载

- 在开发中， 可能会遇到这样的情况。有些资源不需要马上用到，但是希望尽早获取， 这时候就可以使用预加载。
- 预加载其实是声明式的 `fetch` ， 强制浏览器请求资源， 并且不会阻塞 `onload` 事件，可以使用以下代码开启预加载

```html
<link rel="preload" href="http://blog.poetries.top">
```

> 预加载可以一定程度上降低首屏的加载时间， 因为可以将一些不影响首屏但重要的文件延后加载， 唯一缺点就是兼容性不好。

24.7 预渲染

> 可以通过预渲染将下载的文件预先在后台渲染， 可以使用以下代码开启预渲染

```html
<link rel="prerender" href="http://blog.poetries.top">
```

> 预渲染虽然可以提高页面的加载速度，但是要确保该页面大概率会被用户在之后打开， 否则就是白白浪费资源去渲染。

24.8 懒执行

> 懒执行就是将某些逻辑延迟到使用时再计算 。该技术可以用于首屏优化，对于某些耗时逻辑并不需要在首屏就使用的，就可以使用懒执行 。懒执行需要唤醒，一 般可以通过定时器或者事件的调用来唤醒。

24.9 懒加载

- 懒加载就是将不关键的资源延后加载。
- 懒加载的原理就是只加载自定义区域（通常是可视区域，但也可以是即将进入可视区域）内需要加载的东西 。对于图片来说， 先设置图片标签的 `src` 属性为一张占位图，将真实的图片资源放入一个自定义属性中， 当进入自定义区域时，就将自定义属性替换为 `src` 属性， 这样图片就会去下载资源， 实现了图片懒加载。
- 懒加载不仅可以用于图片，也可以使用在别的资源上 。比如进入可视区域才开始播放视频等等。

24.10 CDN

> `CDN` 的原理是尽可能的在各个地方分布机房缓存数据， 这样即使我们的根服务器远在国外，在国内的用户也可以通过国内的机房迅速加载资源。

> 因此， 我们可以将静态资源尽量使用 `CDN` 加载， 由于浏览器对于单个域名有并发请求上限， 可以考虑使用多个 `CDN` 域名 。并且对于 `CDN` 加载静态资源需要注意 `CDN` 域名要与主站不同， 否则每次请求都会带上主站的 `Cookie` ， 平白消耗流量

25 Webpack 性能优化

> 在这部分的内容中， 我们会聚焦于以下两个知识点， 并且每一个知识点都属于高频考点：

- 有哪些方式可以减少 Webpack 的打包时间
- 有哪些方式可以让 Webpack 打出来的包更小

25.1 减少 Webpack 打包时间

1. 优化 Loader

> 对于 Loader 来说，影响打包效率首当其冲必属 Babel 了 。因为 Babel 会将代码转为字符串生成 AST ，然后对 AST 继续进行转变最后再生成新的代码，项目越大，转换代码越多，效率就越低 。当然了， 我们是有办法优化的

> 首先我们可以优化 Loader 的文件搜索范围

```js
module.exports = {
  module: {
    rules: [
      {
        // js 文件才使用 babel
        test: /\.js$/,
        loader: 'babel-loader',
        // 只在 src 文件夹下查找
        include: [resolve( 'src')],
        // 不会去查找的路径
        exclude: /node_modules/
      }
    ]
  }
}
```

> 对于 Babel 来说， 我们肯定是希望只作用在 JS 代码上的，然后 node_modules 中使用的代码都是编译过的，所以我们也完全没有必要再去处理一遍

- 当然这样做还不够， 我们还可以将 Babel 编译过的文件缓存起来，下次只需要编译更改过的代码文件即可， 这样可以大幅度加快打包时间

```
loader:  ' babel- loader? cacheDirectory= true'
```

2. HappyPack

> 受限于 Node 是单线程运行的，所以 Webpack 在打包的过程中也是单线程的，特别是在执行 Loader 的时候， 长时间编译的任务很多， 这样就会导致等待的情况。

> HappyPack 可以将 Loader 的同步执行转换为并行的， 这样就能充分利用系统资源来加快打包效率了

```
module: {
  loaders: [
    {
      test: /\.js$/,
      include: [resolve( 'src')],
      exclude: /node_modules/,
      // id 后面的内容对应下面
      loader: 'happypack/loader?id=happybabel'
    }
  ]
},
plugins: [
  new HappyPack({
    id: 'happybabel',
    loaders: [ 'babel-loader?cacheDirectory'],
    // 开启 4 个线程
    threads: 4
  })
]
```

3. DllPlugin

> DllPlugin 可以将特定的类库提前打包然后引入 。这种方式可以极大的减少打包类库的次数， 只有当类库更新版本才有需要重新打包， 并且也实现了将公共代码抽离成单独文件的优化方案。

接下来我们就来学习如何使用 DllPlugin

```js
// 单独配置在一个文件中
// webpack.dll.conf.js
const path = require( 'path')
const webpack = require( 'webpack')
module.exports = {
  entry: {
    // 想统一打包的类库
    vendor: [ 'react']
  },
  output: {
    path: path.join(__dirname, 'dist'),
    filename: '[name].dll.js',
    library: '[name]- [hash]'
  },
  plugins: [
    new webpack.DllPlugin({
      // name 必须和 output.library 一致
      name: '[name]- [hash]',
      // 该属性需要与 DllReferencePlugin 中一致
      context: __dirname,
      path: path.join(__dirname, 'dist', '[name]-manifest.json')
    })
  ]
}
```

然后我们需要执行这个配置文件生成依赖文件，接下来我们需要使用 DllReferencePlugin 将依赖文件引入项目中

```js
// webpack.conf.js
module.exports = {
  // ...省略其他配置
  plugins: [
    new webpack.DllReferencePlugin({
      context: __dirname,
      // manifest 就是之前打包出来的 json 文件
      manifest: require( './dist/vendor-manifest.json'),
    })
  ]
}
```

4. 代码压缩

> 在 Webpack3 中， 我们一般使用 UglifyJS 来压缩代码，但是这个是单线程运行的， 为了加快效率， 我们可以使用 webpack-parallel-uglify-plugin 来并行运行 UglifyJS ，从而提高效率。

> 在 Webpack4 中， 我们就不需要以上这些操作了， 只需要将 mode 设置为 production 就可以默认开启以上功能 。代码压缩也是我们必做的性能优化方案， 当然我们不止可以压缩 JS 代码， 还可以压缩 HTML 、 CSS 代码，并且在压缩 JS 代码的过程中， 我们还可以通过配置实现比如删除 console.log 这类代码的功能。

5. 一些小的优化点

> 我们还可以通过一些小的优化点来加快打包速度

- resolve.extensions ：用来表明文件后缀列表， 默认查找顺序是 ['.js', '.json'] ， 如果你的导入文件没有添加后缀就会按照这个顺序查找文件 。我们应该尽可能减少后缀列表长度，然后将出现频率高的后缀排在前面
- resolve.alias ：可以通过别名的方式来映射一个路径， 能让 Webpack 更快找到路径
- module.noParse ：如果你确定一个文件下没有其他依赖，就可以使用该属性让 Webpack 不扫描该文件， 这种方式对于大型的类库很有帮助

25.2 减少 Webpack 打包后的文件体积

1. 按需加载

> 想必大家在开发 SPA 项目的时候，项目中都会存在十几甚至更多的路由页面 。如果我们将这些页面全部打包进一个 JS 文件的话， 虽然将多个请求合并了，但是同样也加载了很多并不需要的代码，耗费了更长的时间 。那么为了首页能更快地呈现给用户， 我们肯定是希望首页能加载的文件体积越小越好， 这时候我们就可以使用按需加载，将每个路由页面单独打包为一个文件 。当然不仅仅路由可以按需加载，对于 loadash 这种大型类库同样可以使用这个功能。

> 按需加载的代码实现这里就不详细展开了， 因为鉴于用的框架不同， 实现起来都是不一样的 。当然了， 虽然他们的用法可能不同，但是底层的机制都是一样的 。都是当使用的时候再去下载对应文件， 返回一个 `Promise` ， 当 `Promise` 成功以后去执行回调。

2. Scope Hoisting

> `Scope Hoisting` 会分析出模块之间的依赖关系，尽可能的把打包出来的模块合并到一个函数中去。

比如我们希望打包两个文件

```js
// test.js
export const a = 1

// index.js
import { a } from './test.js'
```

> 对于这种情况， 我们打包出来的代码会类似这样

```js
[
  /* 0 */
  function (module, exports, require) {
    //...
  },
  /* 1 */
  function (module, exports, require) {
    //...
  }
]
```

> 但是如果我们使用 `Scope Hoisting` 的话，代码就会尽可能的合并到一个函数中去，也就变成了这样的类似代码

```js
[
  /* 0 */
```

```
  function (module, exports, require) {
    //...
  }
]
```

> 这样的打包方式生成的代码明显比之前的少多了。如果在 Webpack4 中你希望开启这个功能，只需要启用 optimization.concatenateModules 就可以了。

```js
module.exports = {
  optimization: {
    concatenateModules: true
  }
}
```

3. Tree Shaking

> Tree Shaking 可以实现删除项目中未被引用的代码，比如

```js
// test.js
export const a = 1
export const b = 2
// index.js
import { a } from './test.js'
```

- 对于以上情况，test 文件中的变量 b 如果没有在项目中使用到的话，就不会被打包到文件中。
- 如果你使用 Webpack 4 的话，开启生产环境就会自动启动这个优化功能。

26 实现小型打包工具

> 该工具可以实现以下两个功能

- 将 ES6 转换为 ES5
- 支持在 JS 文件中 import CSS 文件

> 通过这个工具的实现，大家可以理解到打包工具的原理到底是什么

实现

> 因为涉及到 `ES6` 转 `ES5` ，所以我们首先需要安装一些 `Babel` 相关的工具

```
yarn add babylon babel-traverse babel-core babel-preset-env
```

接下来我们将这些工具引入文件中

```js
const fs = require( 'fs')
const path = require( 'path')
const babylon = require( 'babylon')
const traverse = require( 'babel-traverse').default
const { transformFromAst } = require( 'babel-core')
```

首先， 我们先来实现如何使用 `Babel` 转换代码

```js
function readCode( filePath) {
  // 读取文件内容
  const content = fs.readFileSync(filePath, 'utf-8')
  // 生成 AST
  const ast = babylon.parse(content, {
    sourceType: 'module'
  })
  // 寻找当前文件的依赖关系
  const dependencies = []
  traverse(ast, {
    ImportDeclaration: ({ node }) => {
      dependencies.push(node.source.value)
    }
  })
  // 通过 AST 将代码转为 ES5
  const { code } = transformFromAst(ast, null, {
    presets: [ 'env']
  })
  return {
    filePath,
    dependencies,
    code
```

```
  }
}
```

- 首先我们传入一个文件路径参数，然后通过 `fs` 将文件中的内容读取出来
- 接下来我们通过 `babylon` 解析代码获取 `AST` ， 目的是为了分析代码中是否还引入了别的文件
- 通过 `dependencies` 来存储文件中的依赖，然后再将 `AST` 转换为 `ES5` 代码
- 最后函数返回了一个对象，对象中包含了当前文件路径 、当前文件依赖和当前文件转换后的代码

> 接下来我们需要实现一个函数， 这个函数的功能有以下几点

- 调用 `readCode` 函数，传入入口文件
- 分析入口文件的依赖
- 识别 `JS` 和 `CSS` 文件

```js
function getDependencies(entry) {
  // 读取入口文件
  const entryObject = readCode(entry)
  const dependencies = [entryObject]
  // 遍历所有文件依赖关系
  for (const asset of dependencies) {
    // 获得文件目录
    const dirname = path.dirname(asset.filePath)
    // 遍历当前文件依赖关系
    asset.dependencies.forEach(relativePath => {
      // 获得绝对路径
      const absolutePath = path.join(dirname, relativePath)
      // CSS 文件逻辑就是将代码插入到 `style` 标签中
      if (/\.css$/.test(absolutePath)) {
        const content = fs.readFileSync(absolutePath, 'utf-8')
        const code = `
          const style = document.createElement('style')
          style.innerText = ${JSON.stringify(content).replace(/\\r\\n/g, ''
          document.head.appendChild(style)
        `
        dependencies.push({
          filePath: absolutePath,
          relativePath,
          dependencies: [],
          code
        })
      } else {
```