精简

一、CSS相关

1.1 左边定宽，右边自适应方案：float + margin，float + calc

```
/* 方案1 */
.left {
  width: 120px;
  float: left;
}
.right {
  margin-left: 120px;
}
/* 方案2 */
.left {
  width: 120px;
  float: left;
}
.right {
  width: calc(100% - 120px);
  float: left;
}
```

1.2 左右两边定宽，中间自适应：float，float + calc，圣杯布局（设置BFC，margin负值法），flex

```
.wrap {
  width: 100%;
  height: 200px;
}
.wrap > div {
  height: 100%;
}
/* 方案1 */
.left {
```

```
  width: 120px;
  float: left;
}
.right {
  float: right;
  width: 120px;
}
.center {
  margin: 0 120px;
}
/* 方案2 */
.left {
  width: 120px;
  float: left;
}
.right {
  float: right;
  width: 120px;
}
.center {
  width: calc(100% - 240px);
  margin-left: 120px;
}
/* 方案3 */
.wrap {
  display: flex;
}
.left {
  width: 120px;
}
.right {
  width: 120px;
}
.center {
  flex: 1;
}
```

1.3 左右居中

- 行内元素: `text-align: center`
- 定宽块状元素: 左右 `margin` 值为 `auto`
- 不定宽块状元素: `table`布局, `position + transform`

```
/* 方案1 */
```

```
.wrap {
  text-align: center
}
.center {
  display: inline;
  /* or */
  /* display: inline-block; */
}
/* 方案2 */
.center {
  width: 100px;
  margin: 0 auto;
}
/* 方案2 */
.wrap {
  position: relative;
}
.center {
  position: absulote;
  left: 50%;
  transform: translateX(-50%);
}
```

1.4 上下垂直居中

- 定高：`margin`，`position + margin`(负值)
- 不定高：`position` + `transform`，`flex`，`IFC + vertical-align:middle`

```
/* 定高方案1 */
.center {
  height: 100px;
  margin: 50px 0;
}
/* 定高方案2 */
.center {
  height: 100px;
  position: absolute;
  top: 50%;
  margin-top: -25px;
}
/* 不定高方案1 */
.center {
  position: absolute;
  top: 50%;
  transform: translateY(-50%);
```

```
}
/* 不定高方案2 */
.wrap {
  display: flex;
  align-items: center;
}
.center {
  width: 100%;
}
/* 不定高方案3 */
/* 设置 inline-block 则会在外层产生 IFC，高度设为 100% 撑开 wrap 的高度 */
.wrap::before {
  content: '';
  height: 100%;
  display: inline-block;
  vertical-align: middle;
}
.wrap {
  text-align: center;
}
.center {
  display: inline-block;
  vertical-align: middle;
}
```

1.5 盒模型：content（元素内容） + padding（内边距） + border（边框） + margin（外边距）

> 延伸：`box-sizing`

- `content-box`：默认值，总宽度 = `margin` + `border` + `padding` + `width`
- `border-box`：盒子宽度包含 `padding` 和 `border`，`总宽度 = margin + width`
- `inherit`：从父元素继承 `box-sizing` 属性

1.6 BFC、IFC、GFC、FFC：FC（Formatting Contexts），格式化上下文

> `BFC`：块级格式化上下文，容器里面的子元素不会在布局上影响到外面的元素，反之也是如此(按照这个理念来想，只要脱离文档流，肯定就能产生 `BFC`)。产生 `BFC` 方式如下

- `float` 的值不为 `none`。
- `overflow` 的值不为 `visible`。

- `position` 的值不为 `relative` 和 `static`。
- `display` 的值为 `table-cell`，`table-caption`，`inline-block`中的任何一个

> 用处？常见的多栏布局，结合块级别元素浮动，里面的元素则是在一个相对隔离的环境里运行

> `IFC`：内联格式化上下文，`IFC` 的 `line box`（线框）高度由其包含行内元素中最高的实际高度计算而来（不受到竖直方向的 `padding/margin` 影响)。

> `IFC`中的`line box`一般左右都贴紧整个 `IFC`，但是会因为 `float` 元素而扰乱。`float` 元素会位于 IFC 与 `line box` 之间，使得 `line box` 宽度缩短。同个 `ifc` 下的多个 `line box` 高度会不同。 `IFC`中时不可能有块级元素的，当插入块级元素时（如 `p` 中插入 `div`）会产生两个匿名块与 `div` 分隔开，即产生两个 `IFC` ，每个 `IFC` 对外表现为块级元素，与 `div` 垂直排列。

用处?

- 水平居中：当一个块要在环境中水平居中时，设置其为 `inline-block` 则会在外层产生`IFC`，通过 `text-align` 则可以使其水平居中。
- 垂直居中：创建一个 `IFC`，用其中一个元素撑开父元素的高度，然后设置其 `vertical-align`:`middle`，其他行内元素则可以在此父元素下垂直居中

> - **GFC**：网格布局格式化上下文（`display: grid`）
> - **FFC**：自适应格式化上下文（`display: flex`）

二、JS 基础（ES5）

2.1 原型

> 这里可以谈很多，只要围绕 `[[ prototype ]]` 谈，都没啥问题

2.2 闭包

> 牵扯作用域，可以两者联系起来一起谈

2.3 作用域

> 词法作用域，动态作用域

2.4 this

> 不同情况的调用，`this`指向分别如何。顺带可以提一下 `es6` 中箭头函数没有 `this`, `arguments`, `super` 等，这些只依赖包含箭头函数最接近的函数

2.5 call，apply，bind 三者用法和区别

> 参数、绑定规则（显示绑定和强绑定），运行效率（最终都会转换成一个一个的参数去运行）、运行情况（`call`，`apply` 立即执行，`bind` 是`return` 出一个 `this` “固定”的函数，这也是为什么 `bind` 是强绑定的一个原因）

> 注：“固定”这个词的含义，它指的固定是指只要传进去了 `context`，则 `bind` 中 `return` 出来的函数 `this` 便一直指向 `context`，除非 `context` 是个变量

2.6 变量声明提升

> `js` 代码在运行前都会进行 `AST` 解析，函数申明默认会提到当前作用域最前面，变量申明也会进行提升。但赋值不会得到提升。关于 `AST` 解析，这里也可以说是形成词法作用域的主要原因

三、JS 基础（ES6）

3.1 let，const

> `let` 产生块级作用域（通常配合 `for` 循环或者 `{}` 进行使用产生块级作用域），`const` 申明的变量是常量（内存地址不变）

3.2 Promise

> 这里你谈 `promise`的时候，除了将他解决的痛点以及常用的 `API` 之外，最好进行拓展把 `eventloop` 带进来好好讲一下，`microtask`(微任务)、`macrotask`(任务) 的执行顺序，如果看过 `promise` 源码，最好可以谈一谈 原生 `Promise` 是如何实现的。`Promise` 的关键点在于`callback` 的两个参数，一个是 `resovle`，一个是 `reject`。还有就是 `Promise` 的链式调用（`Promise.then()`，每一个 `then` 都是一个责任人）

3.3 Generator

> 遍历器对象生成函数，最大的特点是可以交出函数的执行权

- `function` 关键字与函数名之间有一个星号；

- 函数体内部使用 `yield` 表达式，定义不同的内部状态；
- `next` 指针移向下一个状态

> 这里你可以说说 `Generator` 的异步编程，以及它的语法糖 `async` 和 `awiat`，传统的异步编程。`ES6` 之前，异步编程大致如下

- 回调函数
- 事件监听
- 发布/订阅

> 传统异步编程方案之一：协程，多个线程互相协作，完成异步任务。

3.4 async、await

> `Generator` 函数的语法糖。有更好的语义、更好的适用性、返回值是 `Promise`。

- `async => *`
- `await => yield`

```
// 基本用法

async function timeout (ms) {
  await new Promise((resolve) => {
    setTimeout(resolve, ms)
  })
}
async function asyncConsole (value, ms) {
  await timeout(ms)
  console.log(value)
}
asyncConsole('hello async and await', 1000)
```

> 注：最好把2，3，4 连到一起讲

3.5 AMD，CMD，CommonJs，ES6 Module：解决原始无模块化的痛点

- **AMD**：`requirejs` 在推广过程中对模块定义的规范化产出，提前执行，推崇依赖前置
- **CMD**：`seajs` 在推广过程中对模块定义的规范化产出，延迟执行，推崇依赖就近
- **CommonJs**：模块输出的是一个值的 `copy`，运行时加载，加载的是一个对象（`module.exports` 属性），该对象只有在脚本运行完才会生成

- **ES6 Module**：模块输出的是一个值的引用，编译时输出接口，`ES6`模块不是对象，它对外接口只是一种静态定义，在代码静态解析阶段就会生成。

四、框架相关

4.1 数据双向绑定原理：常见数据绑定的方案

- `Object.defineProperty（vue）`：劫持数据的`getter`和`setter`
- 脏值检测（`angularjs`）：通过特定事件进行轮循 发布/订阅模式：通过消息发布并将消息进行订阅

4.2 VDOM：三个 part

- 虚拟节点类，将真实`DOM`节点用`js`对象的形式进行展示，并提供`render`方法，将虚拟节点渲染成真实`DOM`
- 节点`diff`比较：对虚拟节点进行`js`层面的计算，并将不同的操作都记录到`patch`对象
- `re-render`：解析`patch`对象，进行`re-render`

补充1：VDOM 的必要性?

- **创建真实DOM的代价高**：真实的`DOM`节点`node`实现的属性很多，而`vnode`仅仅实现一些必要的属性，相比起来，创建一个`vnode`的成本比较低。
- **触发多次浏览器重绘及回流**：使用`vnode`，相当于加了一个缓冲，让一次数据变动所带来的所有`node`变化，先在`vnode`中进行修改，然后`diff`之后对所有产生差异的节点集中一次对`DOM tree`进行修改，以减少浏览器的重绘及回流。

补充2：vue 为什么采用 vdom?

> 引入`Virtual DOM`在性能方面的考量仅仅是一方面。

- 性能受场景的影响是非常大的，不同的场景可能造成不同实现方案之间成倍的性能差距，所以依赖细粒度绑定及`Virtual DOM`哪个的性能更好还真不是一个容易下定论的问题。
- `Vue`之所以引入了`Virtual DOM`，更重要的原因是为了解耦`HTML`依赖，这带来两个非常重要的好处是：

> - 不再依赖`HTML`解析器进行模版解析，可以进行更多的`AOT`工作提高运行时效率：通过模版`AOT`编译，`Vue`的运行时体积可以进一步压缩，运行时效率可以进一步提升；

> - 可以渲染到 `DOM` 以外的平台，实现 `SSR`、同构渲染这些高级特性，`Weex` 等框架应用的就是这一特性。

> 综上，`Virtual DOM` 在性能上的收益并不是最主要的，更重要的是它使得 `Vue` 具备了现代框架应有的高级特性。

4.3 vue 和 react 区别

- 相同点：都支持 `ssr`，都有 `vdom`，组件化开发，实现 `webComponents` 规范，数据驱动等
- 不同点：`vue` 是双向数据流（当然为了实现单数据流方便管理组件状态，`vuex` 便出现了），`react` 是单向数据流。`vue`的 `vdom` 是追踪每个组件的依赖关系，不会渲染整个组件树，`react` 每当应该状态被改变时，全部子组件都会 `re-render`

4.4 为什么用 vue

> 简洁、轻快、舒服

五、网络基础类

5.1 跨域

> 很多种方法，但万变不离其宗，都是为了搞定同源策略。重用的有 `jsonp`、`iframe`、`cors`、`img`、H`TML5 postMessage`等等。其中用到 `html` 标签进行跨域的原理就是 `html` 不受同源策略影响。但只是接受 `Get` 的请求方式，这个得清楚。

> **延伸1：img iframe script 来发送跨域请求有什么优缺点?**

1. `iframe`

- 优点：跨域完毕之后`DOM`操作和互相之间的`JavaScript`调用都是没有问题的
- 缺点：1.若结果要以`URL`参数传递，这就意味着在结果数据量很大的时候需要分割传递，巨烦。2.还有一个是`iframe`本身带来的，母页面和`iframe`本身的交互本身就有安全性限制。

2. script

- 优点：可以直接返回`json`格式的数据，方便处理
- 缺点：只接受`GET`请求方式

3. 图片ping

- 优点：可以访问任何`url`，一般用来进行点击追踪，做页面分析常用的方法
- 缺点：不能访问响应文本，只能监听是否响应

> **延伸2：配合 webpack 进行反向代理?**

`webpack` 在 `devServer` 选项里面提供了一个 `proxy` 的参数供开发人员进行反向代理

```
'/api': {
  target: 'http://www.example.com', // your target host
  changeOrigin: true, // needed for virtual hosted sites
  pathRewrite: {
    '^/api': ''  // rewrite path
  }
},
```

> 然后再配合 `http-proxy-middleware` 插件对 `api` 请求地址进行代理

```
const express = require('express');
const proxy = require('http-proxy-middleware');
// proxy api requests
const exampleProxy = proxy(options); // 这里的 options 就是 webpack 里面的 proxy 选项对应的每个选项

// mount `exampleProxy` in web server
const app = express();
app.use('/api', exampleProxy);
app.listen(3000);
```

> 然后再用 `nginx` 把允许跨域的源地址添加到报头里面即可

> 说到 `nginx`，可以再谈谈 `CORS` 配置，大致如下

```
location / {
  if ($request_method = 'OPTIONS') {
    add_header 'Access-Control-Allow-Origin' '*';
    add_header 'Access-Control-Allow-Methods' 'GET, POST, OPTIONS';
    add_header 'Access-Control-Allow-Credentials' 'true';
    add_header 'Access-Control-Allow-Headers' 'DNT, X-Mx-ReqToken, Keep-Alive, User-Agent, X-Requested-With, If-Modified-Since, Cache-Control, Content-Type';
    add_header 'Access-Control-Max-Age' 86400;
    add_header 'Content-Type' 'text/plain charset=UTF-8';
    add_header 'Content-Length' 0;
    return 200;
  }
}
```

5.2 http 无状态无连接

- `http` 协议对于事务处理没有记忆能力
- 对同一个`url`请求没有上下文关系
- 每次的请求都是独立的，它的执行情况和结果与前面的请求和之后的请求是无直接关系的，它不会受前面的请求应答情况直接影响，也不会直接影响后面的请求应答情况
- 服务器中没有保存客户端的状态，客户端必须每次带上自己的状态去请求服务器
- 人生若只如初见，请求过的资源下一次会继续进行请求

http协议无状态中的 状态 到底指的是什么？！

- 【状态】的含义就是：客户端和服务器在某次会话中产生的数据
- 那么对应的【无状态】就意味着：这些数据不会被保留
- 通过增加`cookie`和`session`机制，现在的网络请求其实是有状态的
- 在没有状态的`http`协议下，服务器也一定会保留你每次网络请求对数据的修改，但这跟保留每次访问的数据是不一样的，保留的只是会话产生的结果，而没有保留会话

5.3 http-cache：就是 http 缓存

1. 首先得明确 http 缓存的好处

- 减少了冗余的数据传输，减少网费
- 减少服务器端的压力
- `web` 缓存能够减少延迟与网络阻塞，进而减少显示某个资源所用的时间
- 加快客户端加载网页的速度

2. 常见 http 缓存的类型

- 私有缓存（一般为本地浏览器缓存）
- 代理缓存

3. 然后谈谈本地缓存

> 本地缓存是指浏览器请求资源时命中了浏览器本地的缓存资源，浏览器并不会发送真正的请求给服务器了。它的执行过程是

- 第一次浏览器发送请求给服务器时，此时浏览器还没有本地缓存副本，服务器返回资源给浏览器，响应码是`200 OK`，浏览器收到资源后，把资源和对应的响应头一起缓存下来
- 第二次浏览器准备发送请求给服务器时候，浏览器会先检查上一次服务端返回的响应头信息中的`Cache-Control`，它的值是一个相对值，单位为秒，表示资源在客户端缓存的最大有效期，过期时间为第一次请求的时间减去`Cache-Control`的值，过期时间跟当前的请求时间比较，如果本地缓存资源没过期，那么命中缓存，不再请求服务器
- 如果没有命中，浏览器就会把请求发送给服务器，进入缓存协商阶段。

> 与本地缓存相关的头有：`Cache-Control`、`Expires`，`Cache-Control`有多个可选值代表不同的意义，而`Expires`就是一个日期格式的绝对值。

3.1 Cache-Control

> `Cache-Control`是`HTPP`缓存策略中最重要的头，它是`HTTP/1.1`中出现的，它由如下几个值

- `no-cache`：不使用本地缓存。需要使用缓存协商，先与服务器确认返回的响应是否被更改，如果之前的响应中存在`ETag`，那么请求的时候会与服务端验证，如果资源未被更改，则可以避免重新下载
- `no-store`：直接禁止游览器缓存数据，每次用户请求该资源，都会向服务器发送一个请求，每次都会下载完整的资源
- `public`：可以被所有的用户缓存，包括终端用户和`CDN`等中间代理服务器。
- `private`：只能被终端用户的浏览器缓存，不允许`CDN`等中继缓存服务器对其缓存。
- `max-age`：从当前请求开始，允许获取的响应被重用的最长时间（秒）。

```
# 例如：

Cache-Control: public, max-age=1000
# 表示资源可以被所有用户以及代理服务器缓存，最长时间为1000秒。
```

3.2 Expires

> `Expires`是`HTTP/1.0`出现的头信息，同样是用于决定本地缓存策略的头，它是一个绝对时间，时间格式是如`Mon, 10 Jun 2015 21:31:12 GMT`，只要发送请求时间是在`Expires`之前，那么本地缓存始终有效，否则就会去服务器发送请求获取新的资源。如果同时出现`Cache-Control: max-age`和`Expires`，那么`max-age`优先级更高。他们可以这样组合使用

```
Cache-Control: public
Expires: Wed, Jan 10 2018 00:27:04 GMT
```

3.3 所谓的缓存协商

> 当第一次请求时服务器返回的响应头中存在以下情况时

- 没有 `Cache-Control` 和 `Expires`
- `Cache-Control` 和 `Expires` 过期了
- `Cache-Control` 的属性设置为 `no-cache` 时

> 那么浏览器第二次请求时就会与服务器进行协商，询问浏览器中的缓存资源是不是旧版本，需不需要更新，此时，服务器就会做出判断，如果缓存和服务端资源的最新版本是一致的，那么就无需再次下载该资源，服务端直接返回`304 Not Modified` 状态码，如果服务器发现浏览器中的缓存已经是旧版本了，那么服务器就会把最新资源的完整内容返回给浏览器，状态码就是`200 Ok`，那么服务端是根据什么来判断浏览器的缓存是不是最新的呢？其实是根据`HTTP`的另外两组头信息，分别是：`Last-Modified/If-Modified-Since` 与 `ETag/If-None-Match`。

Last-Modified 与 If-Modified-Since

- 浏览器第一次请求资源时，服务器会把资源的最新修改时间`Last-Modified:Thu, 29 Dec 2011 18:23:55 GMT`放在响应头中返回给浏览器
- 第二次请求时，浏览器就会把上一次服务器返回的修改时间放在请求头`If-Modified-Since:Thu, 29 Dec 2011 18:23:55`发送给服务器，服务器就会拿这个时间跟服务器上的资源的最新修改时间进行对比

> 如果两者相等或者大于服务器上的最新修改时间，那么表示浏览器的缓存是有效的，此时缓存会命中，服务器就不再返回内容给浏览器了，同时`Last-Modified`头也不会返回，因为资源没被修改，返回了也没什么意义。如果没命中缓存则最新修改的资源连同`Last-Modified`头一起返回

```
# 第一次请求返回的响应头
Cache-Control:max-age=3600
Expires: Fri, Jan 12 2018 00:27:04 GMT
Last-Modified: Wed, Jan 10 2018 00:27:04 GMT
# 第二次请求的请求头信息
If-Modified-Since: Wed, Jan 10 2018 00:27:04 GMT
```

> 这组头信息是基于资源的修改时间来判断资源有没有更新，另一种方式就是根据资源的内容来判断，就是接下来要讨论的 `ETag` 与 `If-None-Match`

ETag与If-None-Match

> `ETag/If-None-Match`与`Last-Modified/If-Modified-Since`的流程其实是类似的，唯一的区别是它基于资源的内容的摘要信息（比如`MD5 hash`）来判断

> 浏览器发送第二次请求时，会把第一次的响应头信息`ETag`的值放在`If-None-Match`的请求头中发送到服务器，与最新的资源的摘要信息对比，如果相等，取浏览器缓存，否则内容有更新，最新的资源连同最新的摘要信息返回。用`ETag`的好处是如果因为某种原因到时资源的修改时间没改变，那么用`ETag`就能区分资源是不是有被更新。

```
# 第一次请求返回的响应头：

Cache-Control: public, max-age=31536000
ETag: "15f0fff99ed5aae4edffdd6496d7131f"
# 第二次请求的请求头信息：

If-None-Match: "15f0fff99ed5aae4edffdd6496d7131f"
```

5.4 cookie 和 session

- `session`： 是一个抽象概念，开发者为了实现中断和继续等操作，将 `user agent`和 `server` 之间一对一的交互，抽象为“会话”，进而衍生出“会话状态”，也就是 `session` 的概念
- `cookie`：它是一个世纪存在的东西，`http` 协议中定义在 `header` 中的字段，可以认为是 `session` 的一种后端无状态实现

> 现在我们常说的 `session`，是为了绕开 `cookie` 的各种限制，通常借助`cookie`本身和后端存储实现的，一种更高级的会话状态实现

```
session` 的常见实现要借助`cookie`来发送 `sessionID
```

5.5 安全问题，如 XSS 和 CSRF

- `XSS`：跨站脚本攻击，是一种网站应用程序的安全漏洞攻击，是代码注入的一种。常见方式是将恶意代码注入合法代码里隐藏起来，再诱发恶意代码，从而进行各种各样的非法活动

> 防范：记住一点 “所有用户输入都是不可信的”，所以得做输入过滤和转义

- `CSRF`：跨站请求伪造，也称 `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。与 `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。

> 防范：用户操作验证（验证码），额外验证机制（`token`使用）等