JavaScript 进阶面试题

1. 说说 ECMAScript6 怎么写 class?

- 这个语法糖可以让有 `OOP` 基础的人更快上手 `js`，至少是一个官方的实现了。
- 对熟悉 `js` 的人来说，这个东西没啥大影响；一个 `Object.creat()` 搞定继承，比 `class` 简洁清晰的多。

2. 说说什么是面向对象编程及面向过程编程，它们的异同和优缺点?

- 面向过程就是分析出解决问题所需要的步骤，然后用函数把这些步骤一步一步实现，使用的时候一个一个依次调用就可以了。
- 面向对象是把构成问题事务分解成各个对象，建立对象的目的不是为了完成一个步骤，而是为了描叙某个事物在整个解决问题的步骤中的行为。
- 面向对象是以功能来划分问题，而不是步骤。

3. 说说异步编程的实现方式?

- 回调函数
  - 优点：简单、容易理解
  - 缺点：不利于维护，代码耦合高
- 事件监听(采用时间驱动模式，取决于某个事件是否发生)：
  - 优点：容易理解，可以绑定多个事件，每个事件可以指定多个回调函数
  - 缺点：事件驱动型，流程不够清晰
- 发布/订阅(观察者模式)
  - 类似于事件监听，但是可以通过'消息中心'，了解现在有多少发布者，多少订阅者
- Promise 对象
  - 优点：可以利用 then 方法，进行链式写法；可以书写错误时的回调函数
  - 缺点：编写和理解，相对比较难
- Generator 函数
  - 优点：函数体内外的数据交换、错误处理机制
  - 缺点：流程管理不方便

- async 函数
  - 优点：内置执行器、更好的语义、更广的适用性、返回的是 Promise、结构清晰
  - 缺点：错误处理机制

4. 说说面向对象编程思想？

- 基本思想是使用对象，类，继承，封装等基本概念来进行程序设计
- 优点
  - 易维护
  - 易扩展
  - 开发工作的重用性、继承性高，降低重复工作量。
  - 缩短了开发周期

5. 说说 Gulp 是什么？

- `gulp` 是前端开发过程中一种基于流的代码构建工具，是自动化项目的构建利器；它不仅能对网站资源进行优化，而且在开发过程中很多重复的任务能够使用正确的工具自动完成
- Gulp 的核心概念：流
- 流：就是建立在面向对象基础上的一种抽象的处理数据的工具。在流中，定义了
- 一些处理数据的基本操作，如读取数据，写入数据等，程序员是对流进行所有操作的，而
- 不用关心流的另一头数据的真正流向
- gulp 正是通过流和代码优于配置的策略来尽量简化任务编写的工作

6. 想实现一个对页面某个节点的拖曳？如何做？

- 给需要拖拽的节点绑定 `mousedown` , `mousemove` , `mouseup` 事件
- `mousedown` 事件触发后，开始拖拽
- `mousemove` 时，需要通过 `event.clientX` 和 `clientY` 获取拖拽位置，并实时更新位置
- `mouseup` 时，拖拽结束
- 需要注意浏览器边界的情况

7. 封装一个函数，参数是定时器的时间，then 执行回调函数？

```
function sleep(time) {
    return new Promise((resolve) => setTimeout(resolve, time));
}
```

8. 怎么判断两个对象相等？

```
obj = {a: 1,b: 2}
obj2 = {a: 1,b: 2}
obj3 = {a: 1,b: '2'}
JSON.stringify(obj)==JSON.stringify(obj2);//true
JSON.stringify(obj)==JSON.stringify(obj3);//false
```

9. Javascript 全局函数和全局变量？

- 全局变量
  - `Infinity` 变量代表正的无穷大的数值
  - `NaN` 指示某个值是不是数字值
  - `undefined` 指示未定义的值
- 全局函数
  - `decodeURI()`解码某个编码的 `URI` 。
  - `decodeURIComponent()` 解码一个编码的 `URI` 组件。
  - `encodeURI()` 把字符串编码为 URI。
  - `encodeURIComponent()` 把字符串编码为 `URI` 组件。
  - `escape()` 对字符串进行编码。
  - `eval()` 计算 `JavaScript` 字符串，并把它作为脚本代码来执行。
  - `isFinite()` 检查某个值是否为有穷大的数。
  - `isNaN()` 检查某个值是否是数字。
  - `Number()` 把对象的值转换为数字。
  - `parseFloat()` 解析一个字符串并返回一个浮点数。
  - `parseInt()` 解析一个字符串并返回一个整数。
  - `String()` 把对象的值转换为字符串。
  - `unescape()` 对由 `escape()` 编码的字符串进行解码

10. 项目性能优化有哪些？

- 减少 `HTTP` 请求数
- 减少 `DNS` 查询
- 使用 `CDN`
- 避免重定向
- 图片懒加载
- 减少 `DOM` 元素数量

- 减少 DOM 操作
- 使用外部 JavaScript 和 CSS
- 压缩 JavaScript、CSS、字体、图片等
- 优化 CSS Sprite
- 使用 iconfont
- 字体裁剪
- 多域名分发划分内容到不同域名
- 尽量减少 iframe 使用
- 避免图片 src 为空
- 把样式表放在 link 中
- 把 JavaScript 放在页面底部

11. 说出你对 Electron 的理解？

- 减少 HTTP 请求数最最重要的一点， electron 实际上是一个套了 Chrome 的 nodeJS 序
- Chrome 无各种兼容性问题
- NodeJS

12. WebSocket 的理解？

- 历史来源：序由于 http 存在一个明显的弊端（消息只能有客户端推送到服务器端，而服务器端不能主动推送到客户端），导致如果服务器如果有连续的变化，这时只能使用轮询，而轮询效率过低，并不适合。于是 WebSocket 被发明出来
- 支持双向通信，实时性更强；
- 可以发送文本，也可以二进制文件
- 协议标识符是 ws，加密后是 wss
- 较少的控制开销。连接创建后，ws 客户端、服务端进行数据交换时，协议控制的数据包头部较小。在不包含头部的情况下，服务端到客户端的包头只有 2~10 字节（取决于数据包长度），客户端到服务端的的话，需要加上额外的 4 字节的掩码。而 HTTP 协议每次通信都需要携带完整的头部；
- 支持扩展。ws 协议定义了扩展，用户可以扩展协议，或者实现自定义的子协议。（比如支持自定义压缩算法等）
- 无跨域问题

13. 什么是单线程，和异步的关系？

- 单线程 - 只有一个线程，只能做一件事
- 原因 - 避免 DOM 渲染的冲突
  - 浏览器需要渲染 DOM

- JS 可以修改 DOM 结构
- JS 执行的时候，浏览器 DOM 渲染会暂停
- 两段 JS 也不能同时执行（都修改 DOM 就冲突了）
- webworker 支持多线程，但是不能访问 DOM

- 解决方案 - 异步

14. 说说负载均衡？

- 单台服务器共同协作，不让其中某一台或几台超额工作，发挥服务器的最大作用
- http 重定向负载均衡：调度者根据策略选择服务器以 302 响应请求，缺点只有第一次有效果，后续操作维持在该服务器 dns 负载均衡：解析域名时，访问多个 ip 服务器中的一个（可监控性较弱）原因 - 避免 DOM 渲染的冲突
- 反向代理负载均衡：访问统一的服务器，由服务器进行调度访问实际的某个服务器，对统一的服务器要求大，性能受到 服务器群的数量

15. webpack 的一些 plugin，怎么使用 webpack 对项目进行优化？

- 构建优化：
  - 减少编译体积 ContextReplacementPugin、IgnorePlugin、babel-plugin-import、babelplugin-transform-runtime
  - 并行编译 happypack、thread-loader、uglifyjsWebpackPlugin 开启并行
  - 缓存 cache-loader、hard-source-webpack-plugin、uglifyjsWebpackPlugin 开启缓存、babel-loader 开启缓存
  - 预编译 dllWebpackPlugin && DllReferencePlugin 、auto-dll-webapck-plugin
- 原因 - 避免 DOM 渲染的冲突
  - 减少编译体积 Tree-shaking 、Scope Hositing
  - hash 缓存 webpack-md5-plugin
  - 拆包 splitChunksPlugin 、import() 、require.ensure

16. 编写一个 loader？

- loader 就是一个 node 模块，它输出了一个函数。当某种资源需要用这个
- loader 转换时，这个函数会被调用。并且，这个函数可以通过提供给它的
- this 上下文访问 Loader API。 reverse-txt-loader

```
module.exports = {
  module: {
    rules: [
      { test: /\.css$/, use: 'css-loader' },
      { test: /\.ts$/, use: 'ts-loader' }
    ]
  }
};
```

17. webpack 打包体积 优化思路？

- 提取第三方库或通过引用外部文件的方式引入第三方库
- 代码压缩插件 `UglifyJsPlugin`
- 服务器启用 gzip 压缩
- 按需加载资源文件 `require.ensure`
- 优化 `devtool` 中的 `source-map`
- 剥离 `css` 文件，单独打包
- 去除不必要插件，通常就是开发环境与生产环境用同一套配置文件导致

18. webpack 打包效率？

- 开发环境采用增量构建，启用热更新
- 开发环境不做无意义的工作如提取 `css` 计算文件 hash 等
- 配置 `devtool`
- 选择合适的 `loader`
- 个别 `loader` 开启 `cache` 如 `babel-loader`
- 第三方库采用引入方式
- 提取公共代码
- 优化构建时的搜索路径 指明需要构建目录及不需要构建目录
- 模块化引入需要的部分

19. 原型 / 构造函数 / 实例？

- 原型(`prototype`): 一个简单的对象，用于实现对象的 属性继承。可以简单的理解成对象的爹。在 `Firefox` 和 `Chrome` 中，每个 `JavaScript` 对象中都包含一个`__proto__`(非标准)的属性指向它爹(该对象的原型)，可 `obj.__proto__`进行访问。
- 构造函数: 可以通过 `new` 来 新建一个对象 的函数。
- 实例: 通过构造函数和 `new` 创建出来的对象，便是实例。 实例通过`__proto__`指向原型，通过 `constructor` 指向构造函数。
- 以 `Object` 为例，我们常用的 `Object` 便是一个构造函数，因此我们可以通过它构建实例:

```
// 实例
const instance = new Object()
```

- 则此时， 实例为 `instance`, 构造函数为 `Object`，我们知道，构造函数拥有一个 `prototype` 的属性指向原型，因此原型为:

```
// 实例
const prototype = Object.prototype
```

- 原型 / 构造函数 / 实例三者的关系
  - 实例.__proto__ === 原型
  - 原型.constructor === 构造函数
  - 构造函数.prototype === 原型

```
// 这条线其实是是基于原型进行获取的，可以理解成一条基于原型的映射线
// 例如:
// const o = new Object()
// o.constructor === Object    --> true
// o.__proto__ = null;
// o.constructor === Object    --> false
实例.constructor === 构造函数
```

20. 原型链？

- 原原型链是由原型对象组成，每个对象都有__proto__ 属性，指向了创建该对象的构造函数的原型， __proto__将对象连接起来组成了原型链。是一个用来实现继承和共享属性的有限的对象链。
- 属性查找机制: 当查找对象的属性时，如果实例对象自身不存在该属性，则沿着原型链往上一级查找，找到时则输出，不存在时，则继续沿着原型链往上一级查找，直至最顶级的原型对象 Object.prototype ，如还是没找到，则输出 undefined。
- 属性修改机制: 只会修改实例对象本身的属性，如果不存在，则进行添加该属性，如果需要修改原型的属性时，则可以用: b.prototype.x = 2 ；但是这样会造成所有继承于该对象的实例的属性发生改变。

21. 作用域链？

原我们知道，我们可以在执行上下文中访问到父级甚至全局的变量，这便是作用域链的功劳。作用域链可以理解为一组对象列表，包含 父级和自身的变量对象，因此我们便能通过作用域链访问到父级里声明的变量或者函数。

由两部分组成

- [[scope]] 属性: 指向父级变量对象和作用域链，也就是包含了父级的[[scope]] 和 AO。
- AO ：自身活动对象。

如此[[scopr]]包含[[scope]]，便自上而下形成一条 链式作用域。

22. 对象的拷贝？

浅拷贝: 以赋值的形式拷贝引用对象，仍指向同一个地址，修改时原对象也会受到影响

- Object.assign
- 展开运算符(...)

深拷贝: 完全拷贝一个新对象，修改时原对象不再受到任何影响

- `JSON.parse(JSON.stringify(obj))` ：性能最快
- 具有循环引用的对象时，报错
- 当值为函数、 `undefined` 、或 `symbol` 时，无法拷贝
- 递归进行逐一赋值

23. new 运算符的执行过程？

- 新生成一个对象
- 链接到原型: `obj.__proto__ = Con.prototype`
- 绑定 `this: apply`
- 返回新对象(如果构造函数有自己 `retrun` 时，则返回该值)

24. instanceof 原理？

- 新能在实例的 原型对象链 中找到该构造函数的 `prototype` 属性所指向的 原型对象，就返回 `true`。即:

```
// __proto__: 代表原型对象链
instance.[__proto__...] === instance.constructor.prototype

// return true
```

25. 类型判断？

原我判断 `Target` 的类型，单单用 `typeof` 并无法完全满足，这其实并不是 `bug`，本质原因是 `JS` 的万物皆对象的理论。因此要真正完美判断时，我们需要区分对待

- 基本类型(`null`): 使用 `String(null)`。
- 基本类型(`string/number/boolean/undefined`) +`function` :- 直接使用 `typeof` 即可
- 其余引用类型(`Array/Date/RegExp Error`): 调用 `toString` 后根据`[object XXX]` 进行判断

26. 模块化？

原模块化开发在现代开发中已是必不可少的一部分，它大大提高了项目的可维护、可拓展和可协作性。通常，我们 在浏览器中使用 `ES6` 的模块化支持，在 `Node` 中使用 `commonjs` 的模块化支持

- 分类:
  - `es6: import / export`
  - `commonjs: require / module.exports / exports`
  - `amd: require / defined`

- require 与 import 的区别:
  - ✧ require 支持 动态导入，import 不支持，正在提案 (babel 下可支持)
  - ✧ require 是 同步 导入， impor t属于 异步 导入
  - ✧ require 是 值拷贝，导出值变化不会影响导入值； import 指向 内存地址，导入值会随导出值而变化

27. 防抖与节流？

防抖与节流函数是一种最常用的 高频触发优化方式，能对性能有较大的帮助。

- 防抖 (debounce): 将多次高频操作优化为只在最后一次执行，通常使用的场景是：用户输入，只需再输入完成后做一次输入校验即可:

```
function debounce (fn, wait, immediate) {
    let timer = null
    return function () {
        let args = arguments
        let context = this
        if (immediate && ! timer) {
            fn.apply (context, args)
        }
        if (timer) clearTimeout (timer)
        timer = setTimeout (() =>{
            fn. apply (context, args)
        }, wait)
    }
}
```

- 节流(throttle): 每隔一段时间后执行一次，也就是降低频率，将高频操作优化成低频操作，通常使用场景: 滚动条事件 或者 resize 事件，通常每隔 100~500 ms 执行一次即可

```
function throttle (fn, wait, immediate) {
    let timer = null
    return function () {
        let args = arguments
        let context = this
        if (callNow) {
            fn.apply (context, args)
            callNow = false
        }
        if (!timer) {
            timer = setTimeout (() =>{
                fn. apply (context, args)
            }, wait)
        }
    }
}
```

28. 函数执行改变 this？

- 由于 JS 的设计原理: 在函数中，可以引用运行环境中的变量。因此就需要一个机制来让我们可以在函数体内部获取当前的运行环境，这便是 `this` 。
- `obj.fn()` ，便是 `obj` 调用了函数，既函数中的 `this === obj`
- `fn()` ，这里可以看成 `window.fn()` ，因此 `this === window`
- 三种方式可以手动修改 `this` 的指向:
  - `call: fn.call(target, 1, 2)`
  - `apply: fn.apply(target, [1, 2])`
  - `bind: fn.bind(target)(1,2)`

29. ES6/ES7？

声明

- `let / const` ：块级作用域、不存在变量提升、暂时性死区、不允许重复声明
- `const` ：声明常量，无法修改

解构赋值

- `calss/extend:` 类声明与继承
- `Set/Map:` 新的数据结构

异步解决方案

- `Promise` 的使用与实现
- `generator: 1. yield`: 暂停代码 `2.next()`：继续执行代码
- `await/async`: 是 `generator` 的语法糖， `babel` 中是基于 `promise` 实现。

29. 说说你对 promise 的了解？

- 依照 `Promise/A+`的定义，`Promise` 有四种状态：
  - `pending:`初始状态, 非 `fulfilled` 或 `rejected.`
  - `ulfilled:` 成功的操作.
  - `rejected:` 失败的操作.
  - `settled: Promise` 已被 `fulfilled` 或 `rejected` ，且不是 `pending`
- 另外，`fulfilled` 与 `rejected` 一起合称 `settled`
- `Promise` 对象用来进行延迟(`deferred`) 和异步(`asynchronous`) 计算

Promise 的构造函数

- 构造一个 `Promise` ，最基本的用法如下:

```
var promise = new Promise(function(resolve, reject) {
    if (...) {  // succeed
      resolve(result);
    } else {   // fails
      reject(Error(errMessage))
    }
})
```

- Promise 实例拥有 then 方法（具有 then 方法的对象，通常被称为 thenable ）。它的使用方法如下：

```
promise.then(onFulfilled, onRejected)
```

- 接收两个函数作为参数，一个在 fulfilled 的时候被调用，一个在 rejected 的时候被调用，接收参数就是 future ， onFulfilled 对应 resolve ， onRejected 对应 reject

30. 谈谈你对 AMD、CMD 的理解？

- CommonJS 是服务器端模块的规范，Node.js 采用了这个规范。CommonJS 规范加载模块是同步的，也就是说，只有加载完成，才能执行后面的操作。AMD 规范则是非同步加载模块，允许指定回调函数。
- AMD 推荐的风格通过返回一个对象做为模块对象，CommonJS 的风格通过对 module.exports 或 exports 的属性赋值来达到暴露模块对象的目的。

es6 模块 CommonJS、AMD、CMD

- CommonJS 的规范中，每个 JavaScript 文件就是一个独立的模块上下文 modulecontext），在这个上下文中默认创建的属性都是私有的。也就是说，在一个文件定义的变量（还包括函数和类），都是私有的，对其他文件是不可见的。
- CommonJS 是同步加载模块,在浏览器中会出现堵塞情况，所以不适用 AMD 异步，需要定义回调 define 方式 es6 一个模块就是一个独立的文件，该文件内部的所有变量，外部无法获取。如果你希望外部能够读取模块内部的某个变量，就必须使用 export 关键字输出该变量 es6 还可以出类、方法，自动适用严格模式。