前端知识梳理

- 从浏览器中输入URL到页面渲染的整个流程
  - DNS查找
    - 首先查找本地浏览器缓存中是否有 => 本机的host文件 => DNS服务商找到对应的ip
  - TCP连接
    - 三次握手，ACK(报文到达确认机制)保持可靠性
  - 负载均衡服务器，将请求合理的分发到多台服务器上
  - 服务端收到请求
    - 根据请求进行响应的后端逻辑处理
  - 服务端发出响应
  - 浏览器收到服务端响应结果，首先浏览器会判断状态码，如果是200就继续解析，如果是400或者500就会报错，如果300的话会进行重定向
  - 浏览器开始解析文件，如果是gzip的话会先解析一下
  - 浏览器开始下载所需资源文件，首先会下载完HTML，然后开始按照HTML中的文件顺序，分别开始下载CSS、图片、JS，此时页面已经开始渲染，首先会渲染HTML DOM树，然后再渲染CSS OM树，下载CSS、图片不会block渲染流程，但是如果遇到了JS，如果JS中没有async属性的话，则会block渲染进程，直到执行完JS中的内容后继续开始渲染，这也是为什么要将JS放到尾部的原因
    - HTTP2.0 的多路复用特性可以加快图片的下载，也就是可以通过一个链接下载多个图片
- HTTP2.0新特性
  - 二进制分帧
  - 首部压缩
  - 流量控制
  - 多路复用
    - 多路复用对性能优化工作的贡献
可以并行交错的发送请求和响应，这些请求和响应之间互不影响
只使用一个链接即可并行发送多个请求和响应
消除不必要的延迟，从而减少页面加载的时间
不必再为绕过HTTP1.x限制而多做很多工作
  - 请求优先级
  - 服务器推送
  -

https://juejin.im/post/5a4dfb2ef265da43305ee2d0

- web安全防护
  - XSS攻击
    - 通俗说就是想尽办法将可以执行的代码注入到网页中
    - 持久型和非持久型
      - 持久型的话，攻击的代码被服务端写入进数据库中，这种攻击危害性较大
      - 非持久型的话，一般通过修改URL参数的方式加入攻击代码，诱导用户访问链接从而进行攻击
      - 通常通过两种方式来进行防御
        - 转义字符
        - CSP
          - 本质上是建立白名单，明确的告诉浏览器哪些外部资源是可以加载和执行的，我们只需要配置规则，如何拦截是由浏览器自己实现的，可以通过这个方式来尽量减少XSS攻击,以下两种方式可以来开启CSP
            - 设置HTTP Header中的Content-Security-Policy
              - 只允许加载本站资源
                - Content-Security-Policy: default-src 'self'
              - 只允许加载HTTPS协议图片
                - Content-Security-Policy: img-src https://*
              - 允许加载任何来源框架
                - Content-Security-Policy: child-src 'none'
            - 设置meta标签的方式 <meta http-equiv="Content-Security-Policy"
  - CSRF
    - 跨站请求伪造，原理就是攻击者构造出一个后端请求地址，诱导用户点击或通过某些途径自动发起请求，从而进行相应的逻辑
    - 如何防御
      - 可以遵循以下几种规则：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 => 对Cookie设置SameSite属性，该属性标识Cookie不随着跨域请求发送，有一定兼容性问题
        - 验证Referer => 验证Referer来判断该请求是否为第三方发起的
        - 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
  - 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

- X-FRAME-OPTIONS是一个HTTP响应头，它有三个值：DENY（不允许通过iframe的方式展示);SAMEORIGIN(标识页面可以在相同域名下通过iframe的方式展示);ALLOW-FORM(标识页面可以在制定来源的iframe中展示)

- JS防御
  - 对于比较远古的浏览器，并不能支持上面的方式，就只有通过JS的方式来防御点击了
    - 可以直接通过js document.body.removeChild(iframeDOM)移除
- 中间人攻击
  - 攻击方同时与服务端和客户端建立起了连接，并让对方认为连接是安全的，实际上整个通信过程都被攻击者控制了，攻击者不仅能获得双方的通信信息，还能修改通信信息。通常来说不建议使用公共WIFI;另外建议整站开启HTTPS

- JS基础
  - 原始数据类型
    - 6种：boolean、null、undefined、number、string、symbol
  - 对象类型
    - 数组、对象、函数
      - typeof 与 intsanceof
  - 闭包
    - 闭包就是能够读取其他函数内部变量的函数
      - 作用域链：先在自己的变量范围中查找，如果找不到，就会沿着作用域链去查找
      - 词法作用域：函数是通过词法来划分作用域的，而不是动态地划分作用域的，也就是说作用域链是在定义的时候就已经确定了，不是动态的根据后续来确认的
    - js垃圾回收机制中只要变量被另外一个作用域所引用就不会被回收，所以这个内部变量会存在，被外部所访问
  - 深浅拷贝
    - 浅拷贝
      - Object.assign
      - 展开运算符...
    - 深拷贝
      - JSON.parse(JSON.stringify(object))
  - 原型
  - var、let、const
    - 函数提升优先于变量提升，函数提升会把整个函数提升到作用域顶部，变量提升只会把声明提升到作用域顶部，var存在提升，let、const因为暂时性死区的原因，不能在声明前使用，var在全局作用域下声明变量会导致挂载在window上，其他两者不会

- 继承
  - class只是语法糖，本质上还是函数
  - 原型继承
    - 组合继承
      - 代码示例

```
function Parent(value) {
    this.val = value
}
Parent.prototype.getValue = function() {
    console.log(this.val)
}
function Child(value) {
    Parent.call(this, value)
}
Child.prototype = new Parent()

const child = new Child(1)
child.getValue() // 1
child instanceof Parent // true
```

      - 以上继承的方式核心是在子类的构造函数中通过Parent.call(this)继承父类的属性，然后改变子类的原型为new Parent()来继承父类的函数,这样的优点在于构造函数可以传参，不会与父类引用属性共享，可以复用父类的函数，缺点是就是在继承父类函数的时候调用了父类构造函数，导致子类的原型上多了不需要的父类属性。
    - 寄生组合继承
      - 这种方式对组合继承进行了优化，解决了无用的父类属性问题，还能正确的找到子类的构造函数。
      - 代码示例
  - Class继承
    - 代码示例

```
class Parent {
    constructor(value) {
        this.val = value
    }
    getValue() {
        console.log(this.val)
    }
}
class Child extends Parent {
    constructor(value) {
        super(value)
    }
}
let child = new Child(1)
child.getValue() // 1
```

```
child instanceof Parent
```

- 模块化
  - 立即执行函数
  - AMD
    - 代码示例

```
define(['./a', './b'], function(a,b) {
    a.do()
    b.do()
})
```

  - CMD
    - 代码示例

```
define（function(require, exports, module) {
    var a = require('./a')
    a.do()
})
```

  - CommonJS
  - ES Module
- Proxy
  - Proxy可以来实现数据响应式，替换Object.defineProperty
- 数组高阶方法
  - map、filter、reduce
- JS异步编程
  - 并发、并行
  - 回调函数
  - Generator
  - Promise
    - 手写Promise
  - async及await
  - setTimeout
- Event Loop
  - 进程与线程
    - JS是单线程执行的，在浏览器中来说，当你打开一个Tab页时，其实就是创建一个进程，一个进程中可以有多个线程，比如渲染线程、JS引擎线程、HTTP请求线程等。当你发起一个请求时，其实就是创建了一个线程，当请求结束后，该线程可能就会被销毁；上面提到了JS引擎线程和渲染线程，大家应该都知道，在JS运行

的时候可能会阻止UI渲染，这说明了两个线程是互斥的。因为JS可以修改DOM，如果在JS执行的时候UI线程还在工作，就可能导致不能安全的渲染UI，这其实也是一个单线程的好处。

- 执行栈
  - 是一个存储函数调用的栈结构，遵循先进后出的原则
- 浏览器中的Event Loop
  - 任务源可以分为微任务（microtask）和宏任务（macrotask），在ES6规范中，microtask称为jobs，macrotask称为task
  - 执行顺序如下：首先执行同步代码，这属于宏任务；当执行完所有同步代码后，执行栈为空，查询是否有异步代码需要执行；执行所有微任务；当执行完所有微任务后，如果有必要会渲染页面；然后开始下一轮Event Loop，执行宏任务中的异步代码，也就是setTimeout中的回调函数；所以虽然setTimeout写在Promise之前，但是因为Promise属于微任务而setTimeout属于宏任务，所以会是先执行Promise，再执行setTimeout；微任务包括process.nextTick、promise、MutationObserver; 宏任务包括script、setTimeout、I/O、UI rendering、setInterval、setImmediate
- Node中的Event Loop

Vue深度解析

运行机制全局概览

初始化及挂载

- 在new Vue()之后，Vue会调用_init函数进行初始化，也就是这里的init过程，它会初始化生命周期、事件、props、methods、data、computed与watch等，其中最重要的是通过Object.defineProperty设置setter与getter函数，用来实现响应式以及依赖收集。初始化之后调用$mount来挂载组件，如果是运行时编译，即不存在render function但是存在template的情况，需要执行编译步骤

编译

- compile编译可以分成parse、optimize与generate三个阶段，最终需要得到render function
- parse会用正则等方式解析template模版中的指令、class、style等数据，形成AST
- optimize的主要作用是标记static静态节点，这是vue在编译过程中的一处优化，后面当update更新界面时，会有一个patch的过程，diff算法会直接跳过静态节点，从而减少了比较的过程，优化了patch的性能
- generate是将AST转化成render function字符串的过程，得到结果是render的字符串以及staticRenderFns字符串
- 在经历过parse、optimize与generate这三个阶段后，组件中就会存在渲染VNode所需的render function了

响应式

- getter跟setter在init的时候通过Object.defineProperty进行绑定，也就是当被设置的对象被读取时执行getter函数，而在被赋值的时候执行setter函数

- 当render function被渲染的时候，因为会读取所需对象的值，所以会触发getter函数进行依赖收集，依赖收集会将观察者Watcher对象存放到当前闭包中的订阅者Dep的subs中。
- 在修改对象的值的时候，会触发对应的setter，setter通知之前依赖收集得到的Dep中的每一个Watcher，告诉它们自己的值改变了，需要重新渲染视图，这时候这些watcher就会开始调用update来更新视图，当然中间还有一个patch的过程以及使用队列来异步更新的策略。

- Virtual DOM
  - render function会被转化成VNode节点，Virtual DOM其实就是一棵以Javascript对象（VNode节点）作为基础的树，用对象属性来描述节点，实际上它只是一层对真实DOM的抽象，最终可以通过一系列操作使这棵树映射到真实环境上，由于Virtual DOM是以Javascript对象为基础而不依赖真实平台环境，所以使它具有了跨平台的能力，比如说浏览器平台、Weex、Node等。
- 更新视图
  - 在修改一个对象值的时候，会通过setter --> Watcher --> update的流程来修改对应的视图，当数据变化的后，执行render function就可以得到一个新的VNode节点，如果想要得到新的视图，最简单粗暴的方法就是直接解析这个新的VNode节点，然后用innerHTML直接渲染到真实DOM中，但是其实我们只对其中一小块内容进行了修改，这样就有些太浪费来，所以这就引出来patch，我们将新的VNode与旧的VNode一起传入patch进行比较，经过diff算法得出它们的差异，最后我们将这些有差异的对应DOM进行修改即可

响应式系统的基本原理

- Object.defineproperty
  - function defineReactive (obj, key, val) {
    Object.defineProperty(obj, key, {
    enumerable: true, /* 属性可枚举 */
    configurable: true, /* 属性可被修改或删除 */
    get: function reactiveGetter () {
    return val; /* 实际上会依赖收集，下一小节会讲 */
    },
    set: function reactiveSetter (newVal) {
    if (newVal === val) return;
    cb(newVal);
    }
    });
    }

响应式系统的依赖收集追踪原理

- 订阅者 Dep
  - 它的主要作用是用来存放 Watcher 观察者对象。
- 观察者 Watcher
-

- 依赖收集
  - 在闭包中增加了一个 Dep 类的对象，用来收集 Watcher 对象。在对象被「读」的时候，会触发 reactiveGetter 函数把当前的 Watcher 对象（存放在 Dep.target 中）收集到 Dep 类中去。之后如果当该对象被「写」的时候，则会触发 reactiveSetter 方法，通知 Dep 类调用 notify 来触发所有 Watcher 对象的 update 方法更新对应视图。

- VNode
  - VNode 归根结底就是一个 JavaScript 对象，只要这个类的一些属性可以正确直观地描述清楚当前节点的信息即可。我们来实现一个简单的 VNode 类，加入一些基本属性，为了便于理解，我们先不考虑复杂的情况。
  - class VNode {
    constructor (tag, data, children, text, elm) {
    /*当前节点的标签名*/
    this.tag = tag;
    /*当前节点的一些数据信息，比如props、attrs等数据*/
    this.data = data;
    /*当前节点的子节点，是一个数组*/
    this.children = children;
    /*当前节点的文本*/
    this.text = text;
    /*当前虚拟节点对应的真实dom节点*/
    this.elm = elm;
    }
    }

- 数据状态更新时的差异 diff 及 patch 机制
  - 数据更新视图
  - patch
    - 核心 diff 算法，我们用 diff 算法可以比对出两颗树的「差异」，我们来看一下，假设我们现在有如下两颗树，它们分别是新老 VNode 节点，这时候到了 patch 的过程，我们需要将他们进行比对。
    - diff 算法是通过同层的树节点进行比较而非对树进行逐层搜索遍历的方式，所以时间复杂度只有 O(n)，是一种相当高效的算法，

- 批量异步更新策略及 nextTick 原理
  - 在Vue.js 中修改 data 中的数据后修改视图其实就是一个“setter -> Dep -> Watcher -> patch -> 视图”的过程。
  - Vue.js在默认情况下，每次触发某个数据的 setter 方法后，对应的 Watcher 对象其实会被 push 进一个队列 queue 中，在下一个 tick 的时候将这个队列 queue 全部拿出来 run（ Watcher 对象的一个方法，用来触发 patch 操作） 一遍。
  - nextTick
    - Vue.js 实现了一个 nextTick 函数，传入一个 cb ，这个 cb 会被存储到一个队列

中，在下一个 tick 时触发队列中的所有 cb 事件。

因为目前浏览器平台并没有实现 nextTick 方法，所以 Vue.js 源码中分别用 Promise、setTimeout、setImmediate 等方式在 microtask（或是task）中创建一个事件，目的是在当前调用栈执行完毕以后（不一定立即）才会去执行这个事件。

- Watcher
  - 我们使用一个叫做 has 的 map，里面存放 id -> true (false) 的形式，用来判断是否已经存在相同的 Watcher 对象 （这样比每次都去遍历 queue 效率上会高很多）。

    如果目前队列 queue 中还没有这个 Watcher 对象，则该对象会被 push 进队列 queue 中去。

    waiting 是一个标记位，标记是否已经向 nextTick 传递了 flushSchedulerQueue 方法，在下一个 tick 的时候执行 flushSchedulerQueue 方法来 flush 队列 queue，执行它里面的所有 Watcher 对象的 run 方法。

- Vuex工作原理
  - Store
  - commit 完成mutation
  - dispatch 去触发action
- 复用性
  - 混入（mixins）
    - 全局混入
      - // 为自定义的选项 'myOption' 注入一个处理器。
        Vue.mixin({
          created: function () {
            var myOption = this.$options.myOption
            if (myOption) {
              console.log(myOption)
            }
          }
        })

        new Vue({
          myOption: 'hello!'
        })
    - 选项合并
      - var mixin = {
          data: function () {
            return {

```
      message: 'hello',
      foo: 'abc'
    }
  }
}

new Vue({
  mixins: [mixin],
  data: function () {
    return {
      message: 'goodbye',
      bar: 'def'
    }
  },
  created: function () {
    console.log(this.$data)
    // => { message: "goodbye", foo: "abc", bar: "def" }
  }
})
```

- 自定义指令
  - ```
    // 注册一个全局自定义指令 `v-focus`
    Vue.directive('focus', {
      // 当被绑定的元素插入到 DOM 中时……
      inserted: function (el) {
        // 聚焦元素
        el.focus()
      }
    })
    ```
- 插件
  - 插件通常用来为 Vue 添加全局功能。插件的功能范围没有严格的限制——一般有下面几种：

    添加全局方法或者属性。如: vue-custom-element

    添加全局资源：指令/过滤器/过渡等。如 vue-touch

    通过全局混入来添加一些组件选项。如 vue-router

    添加 Vue 实例方法，通过把它们添加到 Vue.prototype 上实现。

    一个库，提供自己的 API，同时提供上面提到的一个或多个功能。如 vue-router
  - Vue.js 的插件应该暴露一个 install 方法。这个方法的第一个参数是 Vue 构造器，第二个参数是一个可选的选项对象：
  -
    ```

过滤器

- filters: {
    capitalize: function (value) {
      if (!value) return ''
      value = value.toString()
      return value.charAt(0).toUpperCase() + value.slice(1)
    }
  }

- 生命周期
  - beforeCreate
  - created
  - beforeMount
  - mounted
  - beforeUpdate
  - updated
  - beforeDestroy
  - destroyed

- 设计模式
  - 工厂模式
    - 可以想象一个场景。假设有一份很复杂的代码需要用户去调用，但是用户并不关心这些复杂的代码，只需要你提供给我一个接口去调用，用户只负责传递需要的参数，至于这些参数怎么使用，内部有什么逻辑是不关心的，只需要你最后返回我一个实例。这个构造过程就是工厂。
    - 工厂起到的作用就是隐藏了创建实例的复杂度，只需要提供一个接口，简单清晰。
    - class Man {
      constructor(name) {
        this.name = name
      }
      alertName() {
        alert(this.name)
      }
    }

    class Factory {
      static create(name) {
        return new Man(name)
      }
    }

Factory.create('yck').alertName()

- 单例模式
  - 单例模式很常用，比如全局缓存、全局状态管理等等这些只需要一个对象，就可以使用单例模式。

    单例模式的核心就是保证全局只有一个对象可以访问。因为 JS 是门无类的语言，所以别的语言实现单例的方式并不能套入 JS 中，我们只需要用一个变量确保实例只创建一次就行，以下是如何实现单例模式的例子
  - 在 Vuex 源码中，你也可以看到单例模式的使用，虽然它的实现方式不大一样，通过一个外部变量来控制只安装一次 Vuex

    ```
    let Vue // bind on install

    export function install (_Vue) {
      if (Vue && _Vue === Vue) {
        // 如果发现 Vue 有值，就不重新创建实例了
        return
      }
      Vue = _Vue
      applyMixin(Vue)
    }
    ```
- 适配器模式
  - 适配器用来解决两个接口不兼容的情况，不需要改变已有的接口，通过包装一层的方式实现两个接口的正常协作。
  - class Plug {
      getName() {
        return '港版插头'
      }
    }

    class Target {
      constructor() {
        this.plug = new Plug()
      }
      getName() {
        return this.plug.getName() + ' 适配器转二脚插头'
      }
    }

    let target = new Target()
    target.getName() // 港版插头 适配器转二脚插头
  - 在 Vue 中，我们其实经常使用到适配器模式。比如父组件传递给子组件一个时间戳属

性，组件内部需要将时间戳转为正常的日期显示，一般会使用 computed 来做转换这件事情，这个过程就使用到了适配器模式。

- 装饰模式
  - 装饰模式不需要改变已有的接口，作用是给对象添加功能。就像我们经常需要给手机戴个保护套防摔一样，不改变手机自身，给手机添加了保护套提供防摔功能。
  - function readonly(target, key, descriptor) {
    descriptor.writable = false
    return descriptor
    }

    class Test {
    @readonly
    name = 'yck'
    }

    let t = new Test()

    t.yck = '111' // 不可修改
- 代理模式
  - 代理是为了控制对对象的访问，不让外部直接访问到对象。在现实生活中，也有很多代理的场景。比如你需要买一件国外的产品，这时候你可以通过代购来购买产品。
  - <ul id="ul">
      1
      2
      3
      4
      5

    <script>
      let ul = document.querySelector('#ul')
      ul.addEventListener('click', (event) => {
        console.log(event.target);
      })
    </script>
    因为存在太多的 li，不可能每个都去绑定事件。这时候可以通过给父节点绑定一个事件，让父节点作为代理去拿到真实点击的节点。
- 发布订阅模式
  - 发布-订阅模式也叫做观察者模式。通过一对一或者一对多的依赖关系，当对象发生改变时，订阅方都会收到通知。在现实生活中，也有很多类似场景，比如我需要在购物网站上购买一个产品，但是发现该产品目前处于缺货状态，这时候我可以点击有货通知的按钮，让网站在产品有货的时候通过短信通知我。
  - <ul id="ul">

```
<script>
    let ul = document.querySelector('#ul')
    ul.addEventListener('click', (event) => {
        console.log(event.target);
    })
</script>
```

- 在 Vue 中，如何实现响应式也是使用了该模式。对于需要实现响应式的对象来说，在 get 的时候会进行依赖收集，当改变了对象的属性时，就会触发派发更新。

- 外观模式
  - 提供一个接口，隐藏了内部的逻辑，更加方便外部调用。
    对于不同的浏览器，添加事件的方式可能会存在兼容问题。如果每次都需要去这样写一遍的话肯定是不能接受的，所以我们将这些判断逻辑统一封装在一个接口中，外部需要添加事件只需要调用 addEvent 即可。
  - function addEvent(elm, evType, fn, useCapture) {
    if (elm.addEventListener) {
    elm.addEventListener(evType, fn, useCapture)
    return true
    } else if (elm.attachEvent) {
    var r = elm.attachEvent("on" + evType, fn)
    return r
    } else {
    elm["on" + evType] = fn
    }
    }

- React相关
  - HOOK
    - state Hook => useState
    - effect Hook => useEffect
  - 生命周期
    - 组件初始化阶段
      - constructor做一些组件的初始化工作
        super(props)用来调用基类的构造方法，也将父组件的props注入到子组件
    - getDefaultProps()
    - getInitialState()
    - componentWillMount()
    - render()
    - componentDidMount()
    - componentWillReceiveProps(nextProps)
    -

- shouldComponentUpdate(nextProps, nextState)
  - componentWillUpdata(nextProps, nextState)
  - componentDidUpdate()
  - componentWillUnmount()
- Fiber
  - 增量渲染

- 常见数据结构
  - 栈
    - 先进后出
  - 队列
    - 先进先出
  - 链表
  - 树
    - 二叉树
    - 二分搜索树
  - 堆

- Typescript
  - 基础类型
    - 布尔值、数字、字符串、数组、元祖、枚举、Any、Void、Null和undefined、Never、Object
  - 变量声明
  - 接口 interface
  - 类class
    - public、private、protected、static、abstract、readonly
  - 函数
  - 泛型
  - 模块
  - 命名空间
  - 装饰器

- 常用算法
  - 排序
    - 冒泡排序 O(n*n)
      - function bubble(array) {

```
    checkArray(array);
    for (let i = array.length - 1; i > 0; i--) {
      // 从 0 到 `length - 1` 遍历
      for (let j = 0; j < i; j++) {
        if (array[j] > array[j + 1]) swap(array, j, j + 1)
      }
    }
    return array;
  }
```

- 冒泡排序的原理如下，从第一个元素开始，把当前元素和下一个索引元素进行比较。如果当前元素大，那么就交换位置，重复操作直到比较到最后一个元素，那么此时最后一个元素就是该数组中最大的数。下一轮重复以上操作，但是此时最后一个元素已经是最大数了，所以不需要再比较最后一个元素，只需要比较到 length - 2 的位置。

- ```
  function checkArray(array) {
    return Array.isArray(array)
  }
  function swap(array, left, right) {
    let rightValue = array[right]
    array[right] = array[left]
    array[left] = rightValue
  }
  ```

- 插入排序

  - ```
    function insertion(array) {
      if (!checkArray(array)) return
      for (let i = 1; i < array.length; i++) {
        for (let j = i - 1; j >= 0 && array[j] > array[j + 1]; j--)
          swap(array, j, j + 1);
      }
      return array;
    }
    ```

  - 插入排序的原理如下。第一个元素默认是已排序元素，取出下一个元素和当前元素比较，如果当前元素大就交换位置。那么此时第一个元素就是当前的最小数，所以下次取出操作从第三个元素开始，向前对比，重复之前的操作。

- 选择排序

  - 选择排序的原理如下。遍历数组，设置最小值的索引为 0，如果取出的值比当前最小值小，就替换最小值索引，遍历完成后，将第一个元素和最小值索引上的值交换。如上操作后，第一个元素就是数组中的最小值，下次遍历就可以从索引 1 开始重复上述操作。

  - ```
    function selection(array) {
      if (!checkArray(array)) return
      for (let i = 0; i < array.length - 1; i++) {
    ```
    ```

```
    let minIndex = i;
    for (let j = i + 1; j < array.length; j++) {
      minIndex = array[j] < array[minIndex] ? j : minIndex;
    }
    swap(array, i, minIndex);
  }
  return array;
}
```

- 快速排序
  - 快排的原理如下。随机选取一个数组中的值作为基准值，从左至右取值与基准值对比大小。比基准值小的放数组左边，大的放右边，对比完成后将基准值和第一个比基准值大的值交换位置。然后将数组以基准值的位置分为两部分，继续递归以上操作。
  - function quickSort(arr){
    if(arr.length<1){
    return arr;
    }
    var pivotIndex=Math.floor(arr.length/2);//找到那个基准数
    var pivot=arr.splice(pivotIndex,1)[0]; //取出基准数，并去除，splice返回值为数组。
    var left=[];
    var right=[];
    for(var i=0;i<arr.length;i++){
    if(arr[i]<pivot){
    left.push(arr[i]);
    }else{
    right.push(arr[i]);
    }
    }
    return quickSort(left).concat([pivot],quickSort(right)); //加入基准数
    }
    arr=[2,1,5,8,3,7,4,6,9];
    console.log(quickSort(arr)); //[1, 2, 3, 4, 5, 6, 7, 8, 9]

- 浏览器渲染原理
  - 浏览器接收到 HTML 文件并转换为 DOM 树
  - 将 CSS 文件转换为 CSSOM 树
  - 生成渲染树
    - 当我们生成 DOM 树和 CSSOM 树以后，就需要将这两棵树组合为渲染树。
  - 为什么操作 DOM 慢
    - 因为 DOM 是属于渲染引擎中的东西，而 JS 又是 JS 引擎中的东西。当我们通过 JS 操作 DOM 的时候，其实这个操作涉及到了两个线程之间的通信，那么势必会带来一

些性能上的损耗。操作 DOM 次数一多，也就等同于一直在进行线程之间的通信，并且操作 DOM 可能还会带来重绘回流的情况，所以也就导致了性能上的问题。

- 经典面试题：插入几万个 DOM，如何实现页面不卡顿?
  - 对于这道题目来说，首先我们肯定不能一次性把几万个 DOM 全部插入，这样肯定会造成卡顿，所以解决问题的重点应该是如何分批次部分渲染 DOM。大部分人应该可以想到通过 requestAnimationFrame 的方式去循环的插入 DOM，其实还有种方式去解决这个问题：虚拟滚动（virtualized scroller）。

    这种技术的原理就是只渲染可视区域内的内容，非可见区域的那就完全不渲染了，当用户在滚动的时候就实时去替换渲染的内容。

- 什么情况阻塞渲染
  - 首先渲染的前提是生成渲染树，所以 HTML 和 CSS 肯定会阻塞渲染。如果你想渲染的越快，你越应该降低一开始需要渲染的文件大小，并且扁平层级，优化选择器。

    然后当浏览器在解析到 script 标签时，会暂停构建 DOM，完成后才会从暂停的地方重新开始。也就是说，如果你想首屏渲染的越快，就越不应该在首屏就加载 JS 文件，这也是都建议将 script 标签放在 body 标签底部的原因。

    当然在当下，并不是说 script 标签必须放在底部，因为你可以给 script 标签添加 defer 或者 async 属性。

    当 script 标签加上 defer 属性以后，表示该 JS 文件会并行下载，但是会放到 HTML 解析完成后顺序执行，所以对于这种情况你可以把 script 标签放在任意位置。

    对于没有任何依赖的 JS 文件可以加上 async 属性，表示 JS 文件下载和解析不会阻塞渲染。

- 重绘和回流(重排)
  - 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 就叫称为重绘
    回流是布局或者几何属性需要改变就称为回流。
    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，改变父节点里的子节点很可能会导致父节点的一系列回流。
  - 重绘是指一个元素外观的改变所触发的浏览器行为，浏览器会根据元素的新属性重新绘制，使元素呈现新的外观。

    触发重绘的条件：改变元素外观属性。如：color，background-color等。
  - 触发重排的条件：任何页面布局和几何属性的改变都会触发重排，比如：

    1、页面渲染初始化；(无法避免)

    2、添加或删除可见的DOM元素；

3、元素位置的改变，或者使用动画；

4、元素尺寸的改变——大小，外边距，边框；

5、浏览器窗口尺寸的变化（resize事件发生时）；

6、填充内容的改变，比如文本的改变或图片大小改变而引起的计算值宽度和高度的改变；

7、读取某些元素属性：（offsetLeft/Top/Height/Width, clientTop/Left/Width/Height, scrollTop/Left/Width/Height, width/height, getComputedStyle(), currentStyle(IE) ）

- （1）直接改变元素的className

（2）使用visibility代替display
（3）不要经常访问浏览器的flush队列属性；如果一定要访问，可以利用缓存。将访问的值存储起来，接下来使用就不会再引发回流；

（4）使用cloneNode(true or false) 和 replaceChild 技术，引发一次回流和重绘；

（5）将需要多次重排的元素，position属性设为absolute或fixed，元素脱离了文档流，它的变化不会影响到其他元素；

（6）如果需要创建多个DOM节点，可以使用DocumentFragment创建完后一次性的加入document；
（7）尽量不要使用table布局。

- AST
  - 抽象语法树（Abstract Syntax Tree）
- render整体流程图