前端面试

课堂目标w

1. 个人核心竞争力
2. 如何规划自己的学习路径
3. 软实力(三面)
4. 如何学习

知识要点

1. html
2. css
3. javascript
4. 技术选型
5. 软实力
6. 刻意练习

7.

资源

起步

学习路径

1. 基础知识
2. 职场或者面试发现短板
3. 刻意练习
4. 不要闭门造车

建立知识架构

检查点

html

1. 元素
2. 文档
3. 连接
4. 表单
5. 表格
6. 语义化

标签	说明
small	之前表示字体缩小的废弃标签，HTML5救回来表示补充评论。
s	之前表示划线的废弃标签，HTML5救回来表示错误的内容，经常用于电商领域表示打折前的价格。
i	之前表示斜体的废弃标签，HTML5救回来表示读的时候变调。
b	之前表示黑体的废弃标签，HTML5救回来表示关键字。
u	之前表示下划线的废弃标签，HTML5救回来表示避免歧义的注记。
data	跟time标签类似，给机器阅读的内容，意义广泛，可以自由定义。
var	变量，多用于计算机和数学领域。
kbd	用户输入，表示键盘按键居多。
sub	下标，多用于化学/物理/数学领域。
sup	上标，多用于化学/物理/数学领域。
bdi, bdo	用于多语言混合时指定语言或者书写方向（左到右或者右到左）。
mark	表示高亮，这里并非指显示为高亮，而是从读者角度希望的高亮（注意与strong的区分）。
wbr	表示可以换行的位置，主要是英文等文字不允许单词中间换行，这个标签一般在把多个单词粘成很长的单词时候用。
menu	ul的变体，用于功能菜单时使用。
dl，dd，dt	一般出现较为严肃的文章，对一些术语进行定义，dt和dd其实并不总是成对出现，两者是多对多的关系。
main	整个页面只出现一个，表示页面的主要内容，可以理解为特殊的div。

css

1. 选择器
2. 单位
3. 布局
4. 文档
5. 颜色和形状
6. 动画
7. 预处理器

Javascript

1. 变量
2. 数据结构
3. 函数
4. 循环
5. 语句

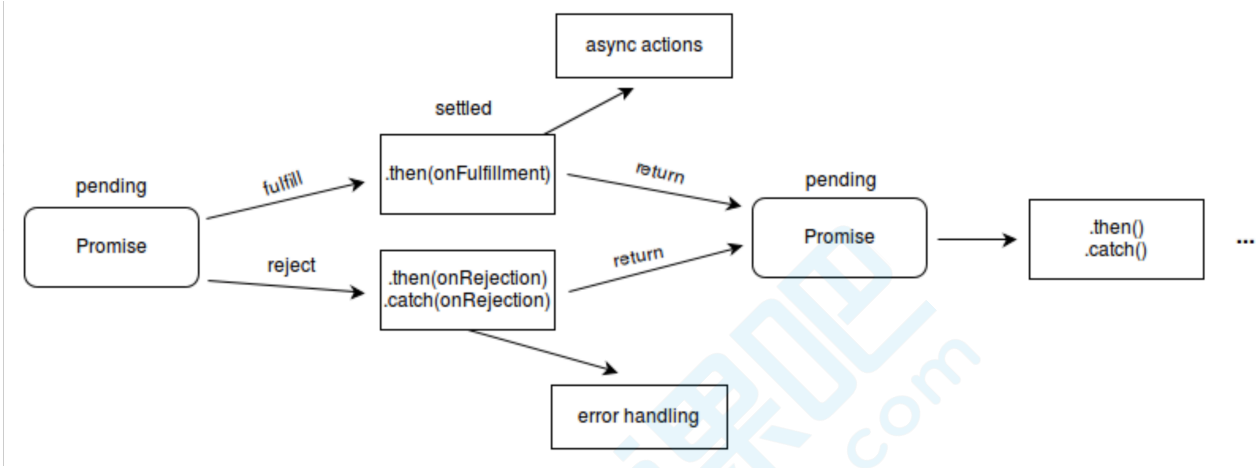
6. 对象
7. this
8. 闭包

典型面试题

手写promise

https://promisesaplus.com/

promises-aplus-tests 来检测是否符合promiseA+规范



解决回掉地域

```
let fs = require('fs')
fs.readFile('./a.txt','utf8',function(err,data){
  fs.readFile(data,'utf8',function(err,data){
    fs.readFile(data,'utf8',function(err,data){
      console.log(data)
    })
  })
})
```

```
axios.get('xx')
    .then(data=>{
        return data.data
    })
    .then(data=>{
        console.log(data)
    })
    .catch(e=>{
        console.log(e)
    })
```

```
new Promise((resolve, reject)=>{
    if(xx){
        resolve('123')
    }else{
        reject('error')
    }
})
```

一个Promise的当前状态必须是以下三种状态中的一种: **等待状态（Pending） 执行状态（Fulfilled）** 和 **拒绝状态（Rejected）** 。

1. pending
    1. 初始化的状态
    2. 可以变成完成或者拒绝
2. fulfilled
    1. 不能改变状态
3. rejected
    1. 不能改变

promise的使用方法

`promise.then(onFulfilled, onRejected)`

```
const promise = new Promise((resolve, reject) => {
    resolve('fulfilled') // 状态由 pending => fulfilled
 })
 promise.then(result => { // onFulfilled
     console.log(result) // 'fulfilled'
 }, reason => { // onRejected 不会被调用
    console.log('reject', reason)

 })

 // 失败
 const promise = new Promise((resolve, reject) => {
    reject('rejected'); // 状态由 pending => rejected
 });
 promise.then(result => { // onFulfilled 不会被调用
     console.log(result)

 }, reason => { // onRejected
     console.log('reject',reason); // 'rejected'
```

```
})
```

catch处理报错 并且是可以链式写

```
promise.catch(onRejected)
相当于
promise.then(null, onRrejected);
```

```
class KPromise{
  // 构造器
  constructor(executor){
    // 成功
    let resolve = () => { };
    // 失败
    let reject = () => { };
    // 立即执行
    executor(resolve, reject);
  }
```

```
const PENDING = 1
const FULFILLED = 2
const REJECTED = 3

class KPromise{
    constructor(cb){
        this.state = PENDING
        // 完成后的传值
        this.value = null
        // 失败后的原因
        this.reason = null
        this.fulfilledCbs = []
        this.rejectCbs = []
        // this.fn = fn
        let reslove = data=>{
            setTimeout(()=>{
            // 这个执行后，修改状态
                if(this.state==PENDING){
                    this.state = FULFILLED
```

```
                    this.value = data
                    this.fulfilledCbs.forEach(v=>v(data))
                }
            })

        }
        let reject = reason=>{
            setTimeout(()=>{
            // 这个执行后，修改状态
            if(this.state==PENDING){
                this.state = REJECTED
                this.reason = reason
                this.rejectCbs.forEach(v=>v(reason))

            }
            })

        }
        cb(reslove, reject)
    }
    then(onFufilled,onRejected){
        if(typeof onFufilled=='function'){
            // 成功回掉
            this.fulfilledCbs.push(onFufilled)
        }
        if(typeof onRejected=='function'){
            // 失败回掉
            this.rejectCbs.push(onRejected)
        }

    }
}
let promise = new KPromise((resolve,reject)=>{
    if(2<1){
        resolve('hi')
    }else{
        reject('出错了')
    }
}).then(data=>{
    console.log(data)
}, reason=>{
    throw new Error(reason)
})
```

链式

then返回一个新的promise对象，

1、为了达成链式，我们默认在第一个then里返回一个promise。[就是在then里面返回一个新的promise,称为promise2：`promise2 = new KPromise((resolve, reject)=>{})`

- 将这个promise2返回的值传递到下一个then中
- 如果返回一个普通的值，则将普通的值传递给下一个then中

2、当我们在第一个then中`return`了一个参数（参数未知，需判断）。这个return出来的新的promise就是onFulfilled()或onRejected()的值

onFulfilled()或onRejected()的值，即第一个then返回的值，叫做x，判断x的函数叫做resolvePromise

- 首先，要看x是不是promise。
- 如果是promise，则取它的结果，作为新的promise2成功的结果
- 如果是普通值，直接作为promise2成功的结果
- 所以要比较x和promise2
- resolvePromise的参数有promise2（默认返回的promise）、x（我们自己`return`的对象）、resolve、reject
- resolve和reject是promise2的

```
then(onFulfilled,onRejected) {
  // 声明返回的promise2
  let promise2 = new Promise((resolve, reject)=>{
    if (this.state === 'fulfilled') {
      let x = onFulfilled(this.value);
      // resolvePromise函数，处理自己return的promise和默认的promise2的关系
      resolvePromise(promise2, x, resolve, reject);
    };
    if (this.state === 'rejected') {
      let x = onRejected(this.reason);
      resolvePromise(promise2, x, resolve, reject);
    };
    if (this.state === 'pending') {
      this.onResolvedCallbacks.push(()=>{
        let x = onFulfilled(this.value);
        resolvePromise(promise2, x, resolve, reject);
      })
      this.onRejectedCallbacks.push(()=>{
        let x = onRejected(this.reason);
        resolvePromise(promise2, x, resolve, reject);
      })
    }
  });
  // 返回promise，完成链式
  return promise2;
}
```

resolvePromise

1. 如果x是普通值 直接resolve
2. then = x.then

静态方法

1. Promise.all 接收一个promise对象数组为参数 全部完成，执行fulfilled

```
const p1 = new Promise((resolve, reject) => {
    resolve(1);
});

const p2 = new Promise((resolve, reject) => {
    resolve(2);
});

const p3 = new Promise((resolve, reject) => {
    resolve(3);
});

Promise.all([p1, p2, p3]).then(data => {
    console.log(data); // [1, 2, 3] 结果顺序和promise实例数组顺序是一致的
}, err => {
    console.log(err);
});
```

```
    static all(promises){
        return new KPromise((resolve,reject)=>{
            let count = 0
            let values = []
            function done(){
                count += 1
                if(count==promises.length){
                    resolve(values)
                }
            }
            promises.forEach((p,i)=>{
                p.then(val=>{
                    values[i] = val
                    done()
                },reject)
            })
        })
    }
```

扩展： 实际then会return另一个promise 可以思考下 怎么实现

前端路由原理

1. hash模式 hashchange

vue中使用

```
    const Home = { template: '<div>home</div>' };
    const Book = { template: '<div>book</div>' };
    const Movie = { template: '<div>movie</div>' };

    const routes = [
      { path: '/', component: Home },
      { path: '/book', component: Book },
      { path: '/movie', component: Movie }
    ];

    const router = new VueRouter(Vue, {
      routes
    });

    new Vue({
      el: '#app',
      router
    });
```

```
class VueRouter {
  constructor (Vue, options) {
    this.$options = options;
    this.routeMap = {};
    this.app = new Vue({
      data: {
        current: '#/'
      }
    });

    this.init();
    this.createRouteMap(this.$options);
    this.initComponent(Vue);
  }

  // 绑定事件
  init () {
    window.addEventListener('load', this.onHashChange.bind(this), false);
    window.addEventListener('hashchange', this.onHashChange.bind(this),
false);
```

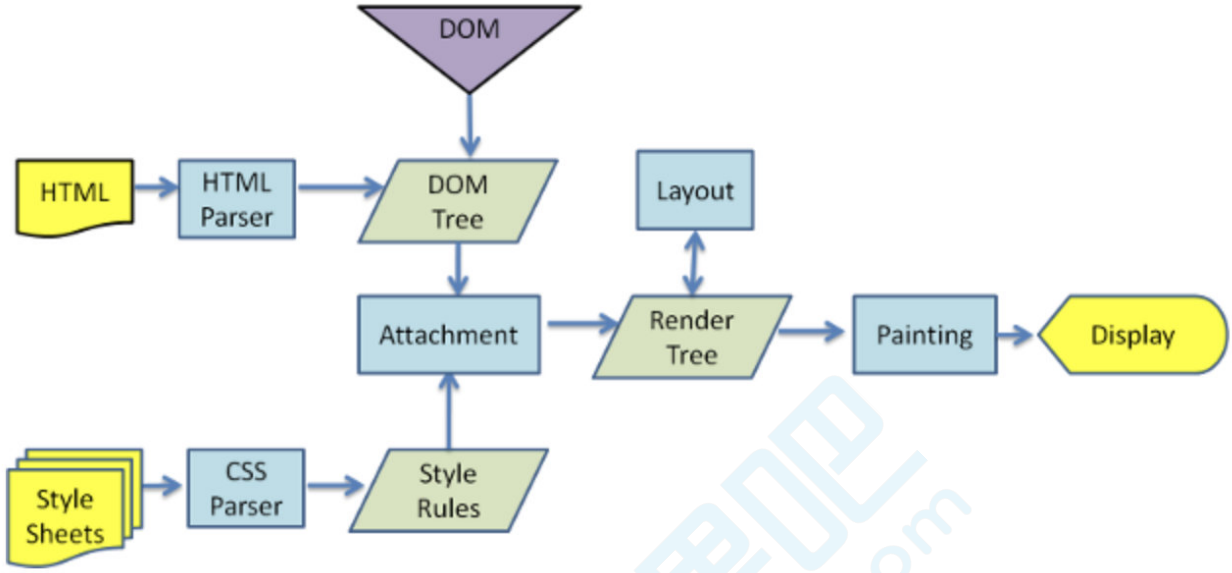
```
  }

  // 路由映射表
  createRouteMap (options) {
    options.routes.forEach(item => {
      this.routeMap[item.path] = item.component;
    });
  }

  // 注册组件
  initComponent (Vue) {
    Vue.component('router-link', {
      props: {
        to: String
      },
      template: '<a :href="to"><slot></slot></a>'
    });

    const _this = this;
    Vue.component('router-view', {
      render (h) {
        var component = _this.routeMap[_this.app.current];
        return h(component);
      }
    });
  }

  // 获取当前 hash 串
  getHash () {
    return window.location.hash.slice(1) || '/';
  }

  // 设置当前路径
  onHashChange () {
    this.app.current = this.getHash();
  }
}
```

借助了vue的响应式能力

浏览器

1. 解析
2. 构建dom树
3. 计算css
4. 合成render tree

5. 绘制
6. dom方法
7. bom方法
8. 事件机制
    1. 捕获、冒泡



前端工程化

1. 工具链
2. 持续集成
3. 预编译
4. 性能优化

框架

1. vuejs
    1. 生命周期
    2. 组件设计
2. reactjs
    1. 组件设计
    2. 单向数据流
    3. fiber

计算机基础

1. 网络协议
    1. tcp http
2. 操作系统
3. 数据库

4. 算法数据结构
5. 编译

1. el-form
    1. 负责提供全局的配置rules
    2. 统一执行所有的item的validate 获取最终的校验结果
2. el-form-item
    1. 使用inject从form哪里获取rules
    2. 校验输入
    3. 显示错误信息
    4. 对外提供validate方法
3. el-input
    1. 双向数据绑定，对面发布input时间 blur input
    2. 决定校验的时机
4.

谈钱不丢人

如何谈钱

职业生涯

自己的规划

英语

第一手资料 打造自己的信息流

如何变成一个高手

1. 刻意练习
2. 任务分解
3. 核心知识
    1. 组件化设计
        1. element源码 or ant.design rc-form
        2. 表单组件设计
        3. 弹窗如何设计
        4. 表格如何设计
        5. 给组件什么数据 props
        6. 组件对外怎么通知 event

7. 如何扩展

2. 流行库的源码
    1. vue
        1. 如何实现响应式
        2. 虚拟dom如何工作的
        3. 模板怎么解析的 怎么收集的依赖
        4. 单文件组件如何工作的
    2. react
        1. 虚拟dom是怎么工作的
        2. setState如何工作的
        3. hooks怎么工作的
    3. 过时的ng1
        1. 实现一个完整的编译器 如何编译语法树
    4. 过时jquery
        1. $（）到底怎么执行的

```
$('#app .test')和$('#app').find('.test') 什么区别
```

```

```

3. 计算机基础
    1. 网络
    2. 算法
    3. 数据结构
    4. 操作系统
    5. 数据库
    6. 编译原理
4. 开源分
    1. 提高影响力
        1. 做开源
        2. 写博客
        3. 写书

回顾

前端面试

- 课堂目标w
- 知识要点
- 资源
- 起步

- 学习路径
- 建立知识架构
  - html
- css
- Javascript
- 典型面试题
  - 手写promise
  - resolvePromise
- 前端路由原理
- 浏览器
- 前端工程化
- 框架
- 计算机基础
- 谈钱不丢人
- 职业生涯
- 英语
  - 如何变成一个高手

回顾