温故而知新

前端知识梳理

承认差距与错误，才能去改正！

目录

当下，正面临着近几年来的最严重的互联网寒冬，听得最多的一句话便是：相见于江湖~。缩减HC、裁员不绝于耳，大家都是人心惶惶，年前如此，年后想必肯定又是一场更为惨烈的江湖厮杀。但博主始终相信，**寒冬之中，人才更是尤为珍贵。只要有过硬的操作和装备，在逆风局下，同样也能来一波收割翻盘。**面试固然有技巧，但绝不是伪造与吹流弊，通过一段短时间沉下心来闭关修炼，出山收割，步入大厂，薪资翻番，岂不爽哉？🤓

修炼原则

想必大家很厌烦笔试和考察知识点。因为其实在平时实战中，讲究的是开发效率，很少会去刻意记下一些细节和深挖知识点，脑海中都是一些分散的知识点，无法系统性地关联成网，一直处于似曾相识的状态。不知道多少人和博主一样，至今每次写阻止冒泡都需要谷歌一番如何拼写。。

以如此的状态，定然是无法在面试的战场上纵横的。其实面试就犹如考试，大家回想下高考之前所做的事，无非就是 **理解** 和 **系统性关联记忆**。本秘籍的知识点较多，花点时间一个个理解并记忆后，自然也就融会贯通，无所畏惧。

由于本秘籍为了便于记忆，快速达到应试状态，类似于复习知识大纲。知识点会尽量的精简与提炼知识脉络，并不去展开深入细节，面面俱到。有兴趣或者有疑问的童鞋可以自行谷歌下对应知识点的详细内容。

CSS

0.基础

1.　　块级元素主要有： 、<ul>、<li>、<p>、<fieldset>、<form>、<h1>、<h2>、<h3>、<h4>、<h5>、<h6>、<hr>、<iframe>、<ol>、<pre>、<table>、<tr>、<td>
　　特点：垂直分布，宽度自适应，占满父元素的剩余空间
2.　行内元素主要有：、<a>、<b>、<img>、
、、、<textarea>、<select>
　　特点：横着分布，不能设置宽高。宽度由内容撑开，但是img可以设置宽高
3. CSS3有哪些新特性?
　　CSS3实现圆角（azborder-radius），阴影（box-shadow），色调反转(filter)
　　对文字加特效（text-shadow、），线性渐变（gradient），旋转（transform）
　　transform:rotate(9deg) scale(0.85,0.90) translate(0px,-30px) skew(-9deg,0deg);
　　//旋转　　缩放　　定位　　倾斜
4. 介绍一下CSS的盒子模型？IE 盒子模型、标准 W3C 盒子模型；
　　1）有两种， IE的content = border + pading;
　　2）盒模型： 内容(content)、填充(padding)、边界(margin)、 边框(border).
5. 新的语义化标签，比如header、footer、nav、article等

1. 盒模型

页面渲染时，dom 元素所采用的 **布局模型**。可通过box-sizing进行设置。根据计算宽高的区域可分为：

- content-box (W3C 标准盒模型)
- border-box (IE 盒模型)
- padding-box
- margin-box (浏览器未实现)

2. BFC

块级格式化上下文，是一个独立的渲染区域，让处于 BFC 内部的元素与外部的元素相互隔离，使内外元素的定位不会相互影响。

IE下为 Layout，可通过 zoom:1 触发

- 触发条件:
    - 根元素

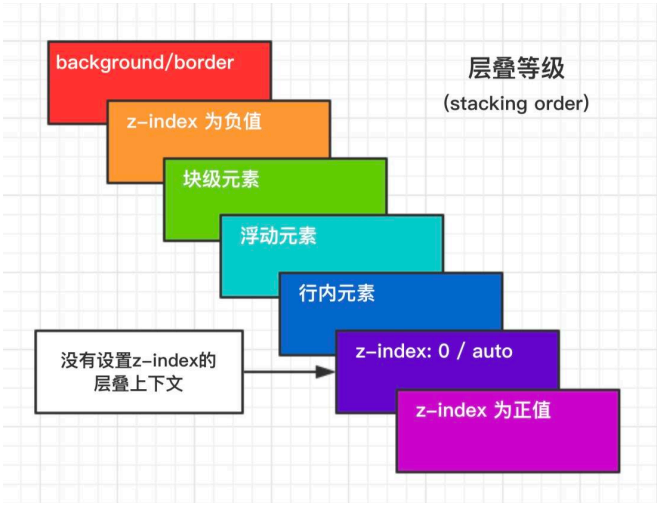
    - position: absolute/fixed
    - display: inline-block / table
    - float 元素
    - ovevflow !== visible
- 规则:
    - 属于同一个 BFC 的两个相邻 Box 垂直排列
    - 属于同一个 BFC 的两个相邻 Box 的 margin 会发生重叠
    - BFC 中子元素的 margin box 的左边， 与包含块 (BFC) border box的左边相接触 (子元素 absolute 除外)
    - BFC 的区域不会与 float 的元素区域重叠
    - 计算 BFC 的高度时，浮动子元素也参与计算
    - 文字层不会被浮动层覆盖，环绕于周围
- 应用:
    - 阻止margin重叠
    - 可以包含浮动元素 —— 清除内部浮动(清除浮动的原理是两个div都位于同一个 BFC 区域之中)
    - 自适应两栏布局
    - 可以阻止元素被浮动元素覆盖

3.层叠上下文

元素提升为一个比较特殊的图层，在三维空间中 **(z轴)** 高出普通元素一等。

- 触发条件
    - 根层叠上下文(html)
    - position
    - css3属性
        - flex
        - transform
        - opacity
        - filter
        - will-change
        - -webkit-overflow-scrolling
- 层叠等级：层叠上下文在z轴上的排序

- 在同一层叠上下文中，层叠等级才有意义
- z-index的优先级最高



4. 居中布局

- 水平居中
  - 行内元素: text-align: center
  - 块级元素: margin: 0 auto
  - absolute + transform
  - flex + justify-content: center

    calc居中通过计算属性居中
- 垂直居中
  - line-height: height
  - absolute + transform
  - flex + align-items: center
  - table
- 水平垂直居中
  - absolute + transform
  - flex + justify-content + align-items

5. 选择器优先级

- !important > 行内样式 > #id > .class > tag > * > 继承 > 默认
- 选择器 **从右往左** 解析

6.去除浮动影响，防止父级高度塌陷

- 通过增加尾元素清除浮动
    - :after /
 : clear: both
- 创建父级 BFC
- 父级设置高度

7.link 与 @import 的区别

- link功能较多，可以定义 RSS，定义 Rel 等作用，而@import只能用于加载 css
- 当解析到link时，页面会同步加载所引的 css，而@import所引用的 css 会等到页面加载完才被加载
- @import需要 IE5 以上才能使用
- link可以使用 js 动态引入，@import不行

8. CSS预处理器(Sass/Less/Postcss)

CSS预处理器的原理: 是将类 CSS 语言通过 **Webpack 编译** 转成浏览器可读的真正 CSS。在这层编译之上，便可以赋予 CSS 更多更强大的功能，常用功能:

- 嵌套
- 变量
- 循环语句
- 条件语句
- 自动前缀
- 单位转换
- mixin复用

面试中一般不会重点考察该点，一般介绍下自己在实战项目中的经验即可~

9.CSS动画

- transition: 过渡动画
    - transition-property: 属性
    - transition-duration: 间隔
    - transition-timing-function: 曲线
    - transition-delay: 延迟
    - 常用钩子: transitionend
- animation / keyframes
    - animation-name: 动画名称，对应@keyframes
    - animation-duration: 间隔
    - animation-timing-function: 曲线
    - animation-delay: 延迟
    - animation-iteration-count: 次数
        - infinite: 循环动画
    - animation-direction: 方向
        - alternate: 反向播放
    - animation-fill-mode: 静止模式
        - forwards: 停止时，保留最后一帧
        - backwards: 停止时，回到第一帧
        - both: 同时运用 forwards / backwards
    - 常用钩子: animationend
- 动画属性: 尽量使用动画属性进行动画，能拥有较好的性能表现
    - translate
    - scale
    - rotate
    - skew
    - opacity
    - color

经验

通常，CSS 并不是重点的考察领域，但这其实是由于现在国内业界对 CSS 的专注不够导致的，真正精通并专注于 CSS 的团队和人才并不多。因此如果能在 CSS 领域有自己的见解和经验，反而会为相当的加分和脱颖而出。

JavaScript

1. 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**。

说了一大堆，大家可能有点懵逼，这里来举个栗子，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:

```
// 原型
const prototype = Object.prototype
```

这里我们可以来看出三者的关系:

实例.__proto__ === 原型

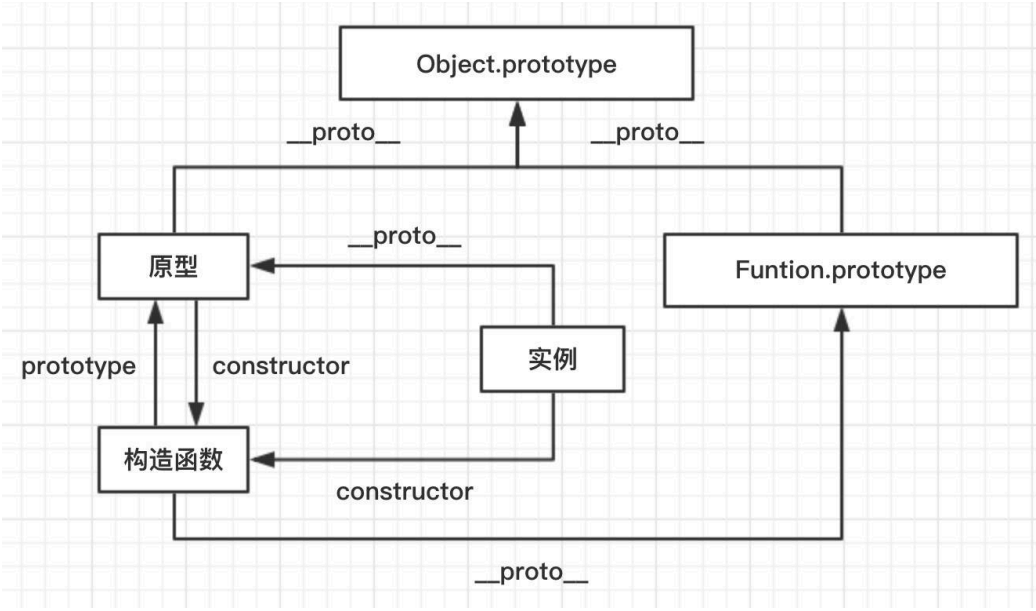
原型.constructor === 构造函数

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
```

放大来看，我画了张图供大家彻底理解:



2.原型链：

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。

3. 执行上下文(EC)

执行上下文可以简单理解为一个对象:

- 它包含三个部分:

    - 变量对象(VO)
    - 作用域链(词法作用域)
    - this指向
- 它的类型:
    - 全局执行上下文
    - 函数执行上下文
    - eval执行上下文
- 代码执行过程:
    - 创建 **全局上下文** (global EC)
    - 全局执行上下文 (caller) 逐行 **自上而下** 执行。遇到函数时，**函数执行上下文** (callee) 被push到执行栈顶层
    - 函数执行上下文被激活，成为 active EC, 开始执行函数中的代码，caller 被挂起
    - 函数执行完后，callee 被pop移除出执行栈，控制权交还全局上下文 (caller)，继续执行

2.变量对象

变量对象，是执行上下文中的一部分，可以抽象为一种 **数据作用域**，其实也可以理解为就是一个简单的对象，它存储着该执行上下文中的所有 **变量和函数声明(不包含函数表达式)**。

活动对象 (AO): 当变量对象所处的上下文为 active EC 时，称为活动对象。

3. 作用域

执行上下文中还包含作用域链。理解作用域之前，先介绍下作用域。作用域其实可理解为该上下文中声明的 **变量和声明的作用范围**。可分为 **块级作用域** 和 **函数作用域**

特性:

- **声明提前**: 一个声明在函数体内都是可见的, 函数优先于变量
- 非匿名自执行函数，函数变量为 **只读** 状态，无法修改

```
let foo = function() { console.log(1) };
(function foo() {
    foo = 10  // 由于foo在函数中只为可读，因此赋值无效
    console.log(foo)
}())

```

```
// 结果打印：  ƒ foo() { foo = 10 ; console.log(foo) }
```

4.作用域链

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**。

5. 闭包

闭包属于一种特殊的作用域，称为 **静态作用域**。它的定义可以理解为: **父函数被销毁** 的情况下，返回出的子函数的[[scope]]中仍然保留着父级的单变量对象和作用域链，因此可以继续访问到父级的变量对象，这样的函数称为闭包。

- 闭包会产生一个很经典的问题:
  - 多个子函数的[[scope]]都是同时指向父级，是完全共享的。因此当父级的变量对象被修改时，所有子函数都受到影响。
- 解决:
  - 变量可以通过 **函数参数的形式** 传入，避免使用默认的[[scope]]向上查找
  - 使用setTimeout包裹，通过第三个参数传入
  - 使用 **块级作用域**，让变量成为自己上下文的属性，避免共享

6. script 引入方式：

- html 静态<script>引入
- js 动态插入<script>
- <script defer>: 延迟加载，元素解析完成后执行
- <script async>: 异步加载，但执行时会阻塞元素渲染

7. 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

8. 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)

9. instanceof原理

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
```

10. 代码的复用

当你发现任何代码开始写第二遍时，就要开始考虑如何复用。一般有以下的方式:

- 函数封装
- 继承
- 复制extend
- 混入mixin
- 借用apply/call

11. 继承

在 JS 中，继承通常指的便是 **原型链继承**，也就是通过指定原型，并可以通过原型链继承原型上的属性或者方法。

- 最优化: **圣杯模式**

```
var inherit = (function(c,p){
    var F = function(){};
    return function(c,p){
        F.prototype = p.prototype;
        c.prototype = new F();
        c.uber = p.prototype;
        c.prototype.constructor = c;
    }
})();
```

- 使用 ES6 的语法糖 class / extends

12. 类型转换

大家都知道 JS 中在使用运算符号或者对比符时，会自带隐式转换，规则如下:

- -、*、/、% ： 一律转换成数值后计算
- +:
    - 数字 + 字符串 = 字符串， 运算顺序是从左到右
    - 数字 + 对象， 优先调用对象的valueOf -> toString
    - 数字 + boolean/null -> 数字
    - 数字 + undefined -> NaN
- [1].toString() === '1'
- {}.toString() === '[object object]'
- NaN !== NaN 、+undefined 为 NaN

13. 类型判断

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

很稳的判断封装:

```
let class2type = {}
'Array Date RegExp Object Error'.split(' ').forEach(e => class2type[ '[object ' + e + ']' ] = e.toLowerCase())

function type(obj) {
    if (obj == null) return String(obj)
    return typeof obj === 'object' ? class2type[ Object.prototype.toString.call(obj) ] || 'object' : typeof obj
}
```

14. 模块化

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，import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

15. 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 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)
    }
}
```

16. 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。

因此要明白 this 指向，其实就是要搞清楚 函数的运行环境，说人话就是，谁调用了函数。例如:

- 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

但这种机制并不完全能满足我们的业务需求，因此提供了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)

17. ES6/ES7

由于 Babel 的强大和普及，现在 ES6/ES7 基本上已经是现代化开发的必备了。通过新的语法糖，能让代码整体更为简洁和易读。

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

class / 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:

yield: 暂停代码

next(): 继续执行代码

```
function* helloWorld() {
  yield 'hello';
  yield 'world';
  return 'ending';
}

const generator = helloWorld();
```

```
8.
9.  generator.next()  // { value: 'hello', done: false }
10.
11. generator.next()  // { value: 'world', done: false }
12.
13. generator.next()  // { value: 'ending', done: true }
14.
15. generator.next()  // { value: undefined, done: true }
```

- await / async: 是generator的语法糖， babel中是基于promise实现。

```
function* helloWorld() {
  yield 'hello';
  yield 'world';
  return 'ending';
}

const generator = helloWorld();

generator.next()  // { value: 'hello', done: false }

generator.next()  // { value: 'world', done: false }

generator.next()  // { value: 'ending', done: true }

generator.next()  // { value: undefined, done: true }
```

18. AST

抽象语法树 (Abstract Syntax Tree)，是将代码逐字母解析成 **树状对象** 的形式。这是语言之间的转换、代码语法检查，代码风格检查，代码格式化，代码高亮，代码错误提示，代码自动补全等等的基础。例如:

```
1.  function square(n){
2.      return n * n
3.  }
```

通过解析转化成的AST如下图:

```
- FunctionDeclaration  {
      type: "FunctionDeclaration"
      start: 0
      end: 35
    + id: Identifier {type, start, end, name}
      expression: false
      generator: false
    + params: [1 element]
    + body: BlockStatement {type, start, end, body}
  }
]
```

19. babel编译原理

- babylon 将 ES6/ES7 代码解析成 AST
- babel-traverse 对 AST 进行遍历转译，得到新的 AST
- 新 AST 通过 babel-generator 转换成 ES5

20. 函数柯里化

在一个函数中，首先填充几个参数，然后再返回一个新的函数的技术，称为函数的柯里化。通常可用于在不侵入函数的前提下，为函数 **预置通用参数**，供多次重复调用。

```
const add = function add(x) {
    return function (y) {
        return x + y
    }
}

const add1 = add(1)

add1(2) === 3
add1(20) === 21
```

21. 数组(array)

- map: 遍历数组，返回回调返回值组成的新数组
- forEach: 无法break，可以用try/catch中throw new Error来停止
- filter: 过滤
- some: 有一项返回true，则整体为true
- every: 有一项返回false，则整体为false
- join: 通过指定连接符生成字符串
- push / pop: 末尾推入和弹出，改变原数组， 返回推入/弹出项
- unshift / shift: 头部推入和弹出，改变原数组，返回操作项
- sort(fn) / reverse: 排序与反转，改变原数组
- concat: 连接数组，不影响原数组， 浅拷贝
- slice(start, end): 返回截断后的新数组，不改变原数组
- splice(start, number, value...): 返回删除元素组成的数组，value 为插入项，改变原数组
- indexOf / lastIndexOf(value, fromIndex): 查找数组项，返回对应的下标
- reduce / reduceRight(fn(prev, cur)， defaultPrev): 两两执行，prev 为上次化简函数的return值，cur 为当前值(从第二项开始)
- 数组乱序：

```
var arr = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10];
arr.sort(function () {
    return Math.random() – 0.5;
});
```

    - 数组拆解: flat: [1,[2,3]] –-> [1, 2, 3]

```
Array.prototype.flat = function() {
    return this.toString().split(',').map(item => +item )
}
```

浏览器

1. 跨标签页通讯

不同标签页间的通讯，本质原理就是去运用一些可以 **共享的中间介质**，因此比较常用的有以下方法:

- 通过父页面window.open()和子页面postMessage
    - 异步下，通过 window.open('about: blank') 和 tab.location.href = '*'
- 设置同域下共享的localStorage与监听window.onstorage
    - 重复写入相同的值无法触发
    - 会受到浏览器隐身模式等的限制
- 设置共享cookie与不断轮询脏检查(setInterval)
- 借助服务端或者中间层实现

2. 浏览器架构

- 用户界面
- 主进程
- 内核
    - 渲染引擎
    - JS 引擎
        - 执行栈
    - 事件触发线程
        - 消息队列
            - 微任务
            - 宏任务
    - 网络异步线程
    - 定时器线程

3. 浏览器下事件循环(Event Loop)

事件循环是指: 执行一个宏任务，然后执行清空微任务列表，循环再执行宏任务，再清微任务列表

- 微任务 microtask(jobs): promise / ajax / Object.observe(该方法已废弃)
- 宏任务 macrotask(task): setTimout / script / IO / UI Rendering

4. 从输入 url 到展示的过程

- DNS 解析
- TCP 三次握手
- 发送请求，分析 url，设置请求报文(头，主体)
- 服务器返回请求的文件 (html)
- 浏览器渲染
    - HTML parser --> DOM Tree
        - 标记化算法，进行元素状态的标记
        - dom 树构建
    - CSS parser --> Style Tree
        - 解析 css 代码，生成样式树
    - attachment --> Render Tree
        - 结合 dom树 与 style树，生成渲染树
    - layout: 布局
    - GPU painting: 像素绘制页面

5. 重绘与回流

当元素的样式发生变化时，浏览器需要触发更新，重新绘制元素。这个过程中，有两种类型的操作，即重绘与回流。

- **重绘(repaint)**: 当元素样式的改变不影响布局时，浏览器将使用重绘对元素进行更新，此时由于只需要UI层面的重新像素绘制，因此 **损耗较少**
- **回流(reflow)**: 当元素的尺寸、结构或触发某些属性时，浏览器会重新渲染页面，称为回流。此时，浏览器需要重新经过计算，计算后还需要重新页面布局，因此是较重的操作。会触发回流的操作:
    - 页面初次渲染
    - 浏览器窗口大小改变
    - 元素尺寸、位置、内容发生改变
    - 元素字体大小变化
    - 添加或者删除可见的 dom 元素

    - 激活 CSS 伪类（例如：:hover）
    - 查询某些属性或调用某些方法
        - clientWidth、clientHeight、clientTop、clientLeft
        - offsetWidth、offsetHeight、offsetTop、offsetLeft
        - scrollWidth、scrollHeight、scrollTop、scrollLeft
        - getComputedStyle()
        - getBoundingClientRect()
        - scrollTo()

回流必定触发重绘，重绘不一定触发回流。重绘的开销较小，回流的代价较高。

最佳实践:

- css
    - 避免使用table布局
    - 将动画效果应用到position属性为absolute或fixed的元素上
- javascript
    - 避免频繁操作样式，可汇总后统一 **一次修改**
    - 尽量使用class进行样式修改
    - 减少dom的增删次数，可使用 **字符串** 或者 documentFragment 一次性插入
    - 极限优化时，修改样式可将其display: none后修改
    - 避免多次触发上面提到的那些会触发回流的方法，可以的话尽量用 **变量存住**

6. 存储

我们经常需要对业务中的一些数据进行存储，通常可以分为 短暂性存储 和 持久性储存。

- 短暂性的时候，我们只需要将数据存在内存中，只在运行时可用
- 持久性存储，可以分为 浏览器端 与 服务器端
    - 浏览器:
        - cookie: 通常用于存储用户身份，登录状态等
            - http 中自动携带， 体积上限为 4K， 可自行设置过期时间

        - localStorage / sessionStorage: 长久储存/窗口关闭删除，体积限制为 4~5M
        - indexDB
    - 服务器:
        - 分布式缓存 redis
        - 数据库

7. Web Worker

现代浏览器为JavaScript创造的 **多线程环境**。可以新建并将部分任务分配到worker线程并行运行，两个线程可 **独立运行，互不干扰**，可通过自带的 **消息机制** 相互通信。

基本用法:

```
// 创建 worker
const worker = new Worker('work.js');

// 向主进程推送消息
worker.postMessage('Hello World');

// 监听主进程来的消息
worker.onmessage = function (event) {
  console.log('Received message ' + event.data);
}
```

限制:

- 同源限制
- 无法使用 document / window / alert / confirm
- 无法加载本地资源

8. V8垃圾回收机制

垃圾回收: 将内存中不再使用的数据进行清理，释放出内存空间。V8 将内存分成 **新生代空间** 和 **老生代空间**。

- **新生代空间**: 用于存活较短的对象
    - 又分成两个空间: from 空间 与 to 空间
    - Scavenge GC算法: 当 from 空间被占满时，启动 GC 算法
        - 存活的对象从 from space 转移到 to space
        - 清空 from space
        - from space 与 to space 互换

        - 完成一次新生代GC
- **老生代空间**: 用于存活时间较长的对象
    - 从 新生代空间 转移到 老生代空间 的条件
        - 经历过一次以上 Scavenge GC 的对象
        - 当 to space 体积超过25%
    - **标记清除算法**: 标记存活的对象，未被标记的则被释放
        - 增量标记: 小模块标记，在代码执行间隙执，GC 会影响性能
        - 并发标记(最新技术): 不阻塞 js 执行
    - **压缩算法**: 将内存中清除后导致的碎片化对象往内存堆的一端移动，解决**内存的碎片化**

9. 内存泄露

- 意外的**全局变量**: 无法被回收
- **定时器**: 未被正确关闭，导致所引用的外部变量无法被释放
- **事件监听**: 没有正确销毁 (低版本浏览器可能出现)
- **闭包**: 会导致父级中的变量无法被释放
- **dom 引用**: dom 元素被删除时，内存中的引用未被正确清空

可用 chrome 中的 timeline 进行内存标记，可视化查看内存的变化情况，找出异常点。

服务端与网络

1. http/https 协议

- 1.0 协议缺陷:
    - 无法复用链接，完成即断开，**重新慢启动和 TCP 3次握手**
    - head of line blocking: **线头阻塞**，导致请求之间互相影响
- 1.1 改进:
    - **长连接**(默认 keep-alive)，复用
    - host 字段指定对应的虚拟站点
    - 新增功能:
        - 断点续传
        - 身份认证

        - 状态管理
        - cache 缓存
            - Cache-Control
            - Expires
            - Last-Modified
            - Etag
- 2.0:
    - 多路复用
    - 二进制分帧层: 应用层和传输层之间
    - 首部压缩
    - 服务端推送
- https: 较为安全的网络传输协议
    - 证书(公钥)
    - SSL 加密
    - 端口 443
- TCP:
    - 三次握手
    - 四次挥手
    - 滑动窗口: 流量控制
    - 拥塞处理
        - 慢开始
        - 拥塞避免
        - 快速重传
        - 快速恢复
- 缓存策略: 可分为 **强缓存** 和 **协商缓存**
    - Cache-Control/Expires: 浏览器判断缓存是否过期，未过期时，直接使用强缓存，**Cache-Control的 max-age 优先级高于 Expires**
    - 当缓存已经过期时，使用协商缓存
        - 唯一标识方案: Etag(response 携带) & If-None-Match(request 携带，上一次返回的 Etag): 服务器判断资源是否被修改，
        - 最后一次修改时间: Last-Modified(response) & If-Modified-Since (request，上一次返回的Last-Modified)
            - 如果一致，则直接返回 304 通知浏览器使用缓存
            - 如不一致，则服务端返回新的资源

    - Last-Modified 缺点：
        - 周期性修改，但内容未变时，会导致缓存失效
        - 最小粒度只到 s， s 以内的改动无法检测到
    - Etag 的优先级高于 Last-Modified

2. 常见状态码

- 1xx: 接受，继续处理
- 200: 成功，并返回数据
- 201: 已创建
- 202: 已接受
- 203: 成为，但未授权
- 204: 成功，无内容
- 205: 成功，重置内容
- 206: 成功，部分内容
- 301: 永久移动，重定向
- 302: 临时移动，可使用原有URI
- 304: 资源未修改，可使用缓存
- 305: 需代理访问
- 400: 请求语法错误
- 401: 要求身份认证
- 403: 拒绝请求
- 404: 资源不存在
- 500: 服务器错误

3. get / post

- get: 缓存、请求长度受限、会被历史保存记录
    - 无副作用(不修改资源)，幂等(请求次数与资源无关)的场景
- post: 安全、大数据、更多编码类型

两者详细对比如下图:

	GET	POST
后退按钮/刷新	无害	数据会被重新提交（浏览器应该告知用户数据会被重新提交）。
书签	可收藏为书签	不可收藏为书签
缓存	能被缓存	不能缓存
编码类型	application/x-www-form-urlencoded	application/x-www-form-urlencoded 或 multipart/form-data。为二进制数据使用多重编码。
历史	参数保留在浏览器历史中。	参数不会保存在浏览器历史中。
对数据长度的限制	是的。当发送数据时，GET 方法向 URL 添加数据；URL 的长度是受限制的（URL 的最大长度是 2048 个字符）。	无限制。
对数据类型的限制	只允许 ASCII 字符。	没有限制。也允许二进制数据。
安全性	与 POST 相比，GET 的安全性较差，因为所发送的数据是 URL 的一部分。 在发送密码或其他敏感信息时绝不要使用 GET ！	POST 比 GET 更安全，因为参数不会被保存在浏览器历史或 web 服务器日志中。
可见性	数据在 URL 中对所有人都是可见的。	数据不会显示在 URL 中。

4. Websocket

Websocket 是一个 **持久化的协议**， 基于 http ， 服务端可以 **主动 push**

- 兼容：
  - FLASH Socket
  - 长轮询： 定时发送 ajax
  - long poll： 发送 –-> 有消息时再 response
- new WebSocket(url)
- ws.onerror = fn
- ws.onclose = fn
- ws.onopen = fn
- ws.onmessage = fn
- ws.send()

5. TCP三次握手

建立连接前，客户端和服务端需要通过握手来确认对方:

- 客户端发送 syn(同步序列编号) 请求，进入 syn_send 状态，等待确认
- 服务端接收并确认 syn 包后发送 syn+ack 包，进入 syn_recv 状态
- 客户端接收 syn+ack 包后，发送 ack 包，双方进入 established 状态

6. TCP四次挥手

- 客户端 -- FIN --> 服务端， FIN—WAIT
- 服务端 -- ACK --> 客户端， CLOSE-WAIT
- 服务端 -- ACK,FIN --> 客户端， LAST-ACK
- 客户端 -- ACK --> 服务端，CLOSED

7. Node 的 Event Loop: 6个阶段

- timer 阶段: 执行到期的setTimeout / setInterval队列回调
- I/O 阶段: 执行上轮循环残流的callback
- idle, prepare
- poll: 等待回调
    - 1. 执行回调
    - 2. 执行定时器
        - 如有到期的setTimeout / setInterval， 则返回 timer 阶段
        - 如有setImmediate，则前往 check 阶段
- check
    - 执行setImmediate
- close callbacks

跨域

1. 跨域

A. **什么是跨域?**

从当前url访问另一url时，只要协议、域名、端口有任何一个不同，就是跨域。

B. **为什么不能跨域?**

A. 什么是跨域?

从当前url访问另一url时，只要协议、域名、端口有任何一个不同，就是跨域。

B. 为什么不能跨域?

浏览器有一个同源策略，用来保护用户的安全。

如果没有这个策略的话，a网站就可以操作b网站的页面，这样将会导致b网站的页面发生混乱，甚至信息被获取，包括服务器端发来的session。

JSONP: 利用<script>标签不受跨域限制的特点，缺点是只能支持 get 请求

```
function jsonp(url, jsonpCallback, success) {
  const script = document.createElement('script')
  script.src = url
  script.async = true
  script.type = 'text/javascript'
  window[jsonpCallback] = function(data) {
    success && success(data)
  }
  document.body.appendChild(script)
}

```

CORS方案

设置 CORS: Access-Control-Allow-Origin：*

使用H5中的window.postMessage

解决方法

window.postMessage(message,targetOrigin）方法是html5新引进的特性，可以使用它来向其它的window对象发送消息，无论这个window对象是属于同源或不同源，目前IE8+、FireFox、Chrome、Opera等浏览器都已经支持window.postMessage方法。

应用场景

1）页面和其打开的新窗口的数据传递

2）.多窗口之间消息传递

3）.页面与嵌套的iframe消息传递

详细介绍：http://www.cnblogs.com/dolphinX/p/3464056.html

document.domain

应用场景

必须满足两个条件

1）页面中嵌入fiframe框架。

2）当前页面和iframe中的页面，主域、协议、端口必须完全一致。

5.图像ping

动态创建图像经常用于图像ping。图像ping是与服务器进行简单、单向的跨域通信的一种方式。请求数的数据是通过查询字符串形式发送的，而响应可以是任意内容，但通常是像素图或204响应。通过图像ping，浏览器得不到任何具体的数据，但通过侦听load和error事件，它能知道响应是什么时候接收到的

```
varimg=newImage();
img.onload=img.onerror=function(){
alert("Done!");
};
img.src="http://www.example.com/test?name=Nicholas";
```

缺点：一是只能发送get请求，

二是无法访问服务器的响应文本

图像ping只能用于浏览器与服务器的单向通信

安全

- XSS攻击: 注入恶意代码
  - cookie 设置 httpOnly
  - 转义页面上的输入内容和输出内容
- CSRF: 跨站请求伪造，防护:
  - get 不修改数据
  - 不被第三方网站访问到用户的 cookie
  - 设置白名单，不被第三方网站请求
  - 请求校验

框架：Vue

1. nextTick

在下次dom更新循环结束之后执行延迟回调，可用于获取更新后的dom状态

- 新版本中默认是microtasks, v-on中会使用macrotasks
- macrotasks任务的实现:
  - setImmediate / MessageChannel / setTimeout

2. 生命周期

- _init_

    - initLifecycle/Event，往vm上挂载各种属性
    - callHook: beforeCreated: 实例刚创建
    - initInjection/initState: 初始化注入和 data 响应性
    - created: 创建完成，属性已经绑定， 但还未生成真实dom
    - 进行元素的挂载： $el / vm.$mount()
    - 是否有template: 解析成render function
        - *.vue文件: vue-loader会将<template>编译成render function
    - beforeMount: 模板编译/挂载之前
    - 执行render function，生成真实的dom，并替换到dom tree中
    - mounted: 组件已挂载
- update:
    - 执行diff算法，比对改变是否需要触发UI更新
    - flushScheduleQueue
        - watcher.before: 触发beforeUpdate钩子 - watcher.run(): 执行watcher中的 notify，通知所有依赖项更新UI
    - 触发updated钩子: 组件已更新
- actived / deactivated(keep-alive): 不销毁，缓存，组件激活与失活
- destroy:
    - beforeDestroy: 销毁开始
    - 销毁自身且递归销毁子组件以及事件监听
        - remove(): 删除节点
        - watcher.teardown(): 清空依赖
        - vm.$off(): 解绑监听
    - destroyed: 完成后触发钩子

上面是vue的声明周期的简单梳理，接下来我们直接以代码的形式来完成vue的初始化

```
new Vue({})

// 初始化Vue实例
function _init() {
    // 挂载属性
    initLifeCycle(vm)
    // 初始化事件系统，钩子函数等
```

```
    initEvent(vm)
    // 编译slot、vnode
    initRender(vm)
    // 触发钩子
    callHook(vm, 'beforeCreate')
    // 添加inject功能
    initInjection(vm)
    // 完成数据响应性 props/data/watch/computed/methods
    initState(vm)
    // 添加 provide 功能
    initProvide(vm)
    // 触发钩子
    callHook(vm, 'created')

     // 挂载节点
    if (vm.$options.el) {
        vm.$mount(vm.$options.el)
    }
}

// 挂载节点实现
function mountComponent(vm) {
     // 获取 render function
    if (!this.options.render) {
        // template to render
        // Vue.compile = compileToFunctions
        let { render } = compileToFunctions()
        this.options.render = render
    }
    // 触发钩子
    callHook('beforeMounte')
    // 初始化观察者
    // render 渲染 vdom,
    vdom = vm.render()
    // update: 根据 diff 出的 patchs 挂载成真实的 dom
    vm._update(vdom)
    // 触发钩子
    callHook(vm, 'mounted')
}

// 更新节点实现
funtion queueWatcher(watcher) {
    nextTick(flushScheduleQueue)
}

// 清空队列
function flushScheduleQueue() {
     // 遍历队列中所有修改
    for(){
        // beforeUpdate
        watcher.before()

        // 依赖局部更新节点
```

```
61.         watcher.update()
62.         callHook('updated')
63.     }
64. }
65.
66. // 销毁实例实现
67. Vue.prototype.$destory = function() {
68.     // 触发钩子
69.     callHook(vm, 'beforeDestory')
70.     // 自身及子节点
71.     remove()
72.     // 删除依赖
73.     watcher.teardown()
74.     // 删除监听
75.     vm.$off()
76.     // 触发钩子
77.     callHook(vm, 'destoryed')
78. }
```

3. 数据响应(数据劫持)

看完生命周期后，里面的watcher等内容其实是数据响应中的一部分。数据响应的实现由两部分构成: **观察者(watcher)** 和 **依赖收集器(Dep)**，其核心是defineProperty这个方法，它可以 **重写属性的 get 与 set** 方法，从而完成监听数据的改变。

- Observe (观察者)观察 props 与 state
  - 遍历 props 与 state，对每个属性创建独立的监听器(watcher)
- 使用 defineProperty 重写每个属性的 get/set(defineReactive)
  - get: 收集依赖
    - Dep.depend()
      - watcher.addDep()
  - set: 派发更新
    - Dep.notify()
    - watcher.update()
    - queenWatcher()
    - nextTick
    - flushScheduleQueue
    - watcher.run()
    - updateComponent()

大家可以先看下面的数据相应的代码实现后，理解后就比较容易看懂上面的简单脉络了。

```
let data = {a: 1}
// 数据响应性
observe(data)

// 初始化观察者
new Watcher(data, 'name', updateComponent)
data.a = 2

// 简单表示用于数据更新后的操作
function updateComponent() {
    vm._update() // patchs
}

// 监视对象
function observe(obj) {
     // 遍历对象，使用 get/set 重新定义对象的每个属性值
    Object.keys(obj).map(key => {
        defineReactive(obj, key, obj[key])
    })
}

function defineReactive(obj, k, v) {
    // 递归子属性
    if (type(v) == 'object') observe(v)

    // 新建依赖收集器
    let dep = new Dep()
    // 定义get/set
    Object.defineProperty(obj, k, {
        enumerable: true,
        configurable: true,
        get: function reactiveGetter() {
             // 当有获取该属性时，证明依赖于该对象，因此被添加进收集器中
            if (Dep.target) {
                dep.addSub(Dep.target)
            }
            return v
        },
        // 重新设置值时，触发收集器的通知机制
        set: function reactiveSetter(nV) {
            v = nV
            dep.nofify()
        },
    })
}

// 依赖收集器
class Dep {
    constructor() {
        this.subs = []
```

```
    }
    addSub(sub) {
        this.subs.push(sub)
    }
    notify() {
        this.subs.map(sub => {
            sub.update()
        })
    }
}

Dep.target = null

// 观察者
class Watcher {
    constructor(obj, key, cb) {
        Dep.target = this
        this.cb = cb
        this.obj = obj
        this.key = key
        this.value = obj[key]
        Dep.target = null
    }
    addDep(Dep) {
        Dep.addSub(this)
    }
    update() {
        this.value = this.obj[this.key]
        this.cb(this.value)
    }
    before() {
        callHook('beforeUpdate')
    }
}
```

4. virtual dom 原理实现

- 创建 dom 树
- 树的diff，同层对比，输出patchs(listDiff/diffChildren/diffProps)
    - 没有新的节点，返回
    - 新的节点tagName与key不变， 对比props，继续递归遍历子树
        - 对比属性(对比新旧属性列表):
            - 旧属性是否存在与新属性列表中
            - 都存在的是否有变化
            - 是否出现旧列表中没有的新属性
    - tagName和key值变化了，则直接替换成新节点

- 渲染差异
    - 遍历patchs， 把需要更改的节点取出来
    - 局部更新dom

// diff算法的实现

```
function diff(oldTree, newTree) {
    // 差异收集
    let pathchs = {}
    dfs(oldTree, newTree, 0, pathchs)
    return pathchs
}

function dfs(oldNode, newNode, index, pathchs) {
    let curPathchs = []
    if (newNode) {
        // 当新旧节点的 tagName 和 key 值完全一致时
        if (oldNode.tagName === newNode.tagName && oldNode.key === newNode.key) {
            // 继续比对属性差异
            let props = diffProps(oldNode.props, newNode.props)
            curPathchs.push({ type: 'changeProps', props })
            // 递归进入下一层级的比较
            diffChildrens(oldNode.children, newNode.children, index, pathchs)
        } else {
            // 当 tagName 或者 key 修改了后，表示已经是全新节点，无需再比
            curPathchs.push({ type: 'replaceNode', node: newNode })
        }
    }

    // 构建出整颗差异树
    if (curPathchs.length) {
        if(pathchs[index]){
            pathchs[index] = pathchs[index].concat(curPathchs)
        } else {
            pathchs[index] = curPathchs
        }
    }
}

// 属性对比实现
function diffProps(oldProps, newProps) {
    let propsPathchs = []
    // 遍历新旧属性列表
    // 查找删除项
    // 查找修改项
    // 查找新增项
    forin(olaProps, (k, v) => {
        if (!newProps.hasOwnProperty(k)) {
            propsPathchs.push({ type: 'remove', prop: k })
        } else {
            if (v !== newProps[k]) {
                propsPathchs.push({ type: 'change', prop: k , value: newProps[k] })
            }
```

```
        }
    })
    forin(newProps, (k, v) => {
        if (!oldProps.hasOwnProperty(k)) {
            propsPathchs.push({ type: 'add', prop: k, value: v })
        }
    })
    return propsPathchs
}

// 对比子级差异
function diffChildrens(oldChild, newChild, index, pathchs) {
        // 标记子级的删除/新增/移动
    let { change, list } = diffList(oldChild, newChild, index, pathchs)
    if (change.length) {
        if (pathchs[index]) {
            pathchs[index] = pathchs[index].concat(change)
        } else {
            pathchs[index] = change
        }
    }

     // 根据 key 获取原本匹配的节点，进一步递归从头开始对比
    oldChild.map((item, i) => {
        let keyIndex = list.indexOf(item.key)
        if (keyIndex) {
            let node = newChild[keyIndex]
            // 进一步递归对比
            dfs(item, node, index, pathchs)
        }
    })
}

// 列表对比，主要也是根据 key 值查找匹配项
// 对比出新旧列表的新增/删除/移动
function diffList(oldList, newList, index, pathchs) {
    let change = []
    let list = []
    const newKeys = getKey(newList)
    oldList.map(v => {
        if (newKeys.indexOf(v.key) > -1) {
            list.push(v.key)
        } else {
            list.push(null)
        }
    })

    // 标记删除
    for (let i = list.length - 1; i>= 0; i--) {
        if (!list[i]) {
            list.splice(i, 1)
            change.push({ type: 'remove', index: i })
        }
```

```
    }

    // 标记新增和移动
    newList.map((item, i) => {
        const key = item.key
        const index = list.indexOf(key)
        if (index === -1 || key == null) {
            // 新增
            change.push({ type: 'add', node: item, index: i })
            list.splice(i, 0, key)
        } else {
            // 移动
            if (index !== i) {
                change.push({
                    type: 'move',
                    form: index,
                    to: i,
                })
                move(list, index, i)
            }
        }
    })

    return { change, list }
}
```

5. Proxy 相比于 defineProperty 的优势

- 数组变化也能监听到
- 不需要深度遍历监听

```
let data = { a: 1 }
let reactiveData = new Proxy(data, {
    get: function(target, name){
        // ...
    },
    // ...
})
```

6. vue-router

- mode
  - hash
  - history

- 跳转
    - this.$router.push()
    - <router-link to=""></router-link>
- 占位
    - <router-view></router-view>

7. vuex

- state: 状态中心
- mutations: 更改状态
- actions: 异步更改状态
- getters: 获取状态
- modules: 将state分成多个modules，便于管理

Dispatch ->action->commit->mutations->state->ui

将数据存储到vuex的state的中 更改数

Vuex是Vuex 类似 Redux 的状态管理器，用来管理Vue的所有组件状态。

为什么使用Vuex?

当你打算开发大型单页应用（SPA），会出现多个视图组件依赖同一个状态，来自不同视图的行为需要变更同一个状态。

遇到以上情况时候，你就应该考虑使用Vuex了，它能把组件的共享状态抽取出来，当做一个全局单例模式进行管理。这样不管你在何处改变状态，都会通知使用该状态的组件做出相应修改。

1、vuex有哪几种属性?

答：有五种，分别是 State、 Getter、Mutation 、Action、Model

1.1、state

state为单一状态树，在state中需要定义我们所需要管理的数组、对象、字符串等等，只有在这里定义了，在vue.js的组件中才能获取你定义的这个对象的状态。

1.2、getter

getter有点类似vue.js的计算属性，当我们需要从store的state中派生出一些状态，那么我们就需要使用getter，getter会接收state作为第一个参数，而且getter的返回值会根据它的依赖被缓存起来，只有getter中的依赖值（state中的某个需要派生状态的值）发生改变的时候才会被重新计算。

1.3、mutation

更改store中state状态的唯一方法就是提交mutation，就很类似事件。每个mutation都有一个字符串类型的事件类型和一个回调函数，我们需要改变state的值就要在回调函数中改变。我们要执行这个回调函数，那么我们需要执行一个相应的调用方法：

1.4、action

action可以提交mutation，在action中可以执行store.commit，而且action中可以有任何的异步操作。在页面中如果我们要用这个action，则需要执行store.dispatch

1.5、module

module其实只是解决了当state中很复杂臃肿的时候，module可以将store分割成模块，每个模块中拥有自己的state、mutation、action和getter。

2、vuex的State特性是?

答:

一、Vuex就是一个仓库，仓库里面放了很多对象。其中state就是数据源存放地，对应于与一般Vue对象里面的data

二、state里面存放的数据是响应式的，Vue组件从store中读取数据，若是store中的数据发生改变，依赖这个数据的组件也会发生更新

三、它通过mapState把全局的 state 和 getters 映射到当前组件的 computed 计算属性中

3、vuex的Getter特性是?

答:

一、getters 可以对State进行计算操作，就是Store的计算属性

二、 虽然在组件内也可以做计算属性，但是getters 可以在多组件之间复用

三、 如果一个状态只在一个组件内使用，是可以不用getters

store.state.count

store.commit('方法名称')

获取state

```
const app = new Vue({
  el: '#app',
  // 把 store 对象提供给 “store” 选项，这可以把 store 的实例注入所有的子组件
  store,
  components: { Counter },
  template: `
    <div class="app">
      <counter></counter>
    </div>
})
```

```
// 在单独构建的版本中辅助函数为 Vuex.mapState
import { mapState } from 'vuex'

export default {
  // ...
  computed: mapState({
    // 箭头函数可使代码更简练
    count: state => state.count,

    // 传字符串参数 'count' 等同于 `state => state.count`
    countAlias: 'count',

    // 为了能够使用 `this` 获取局部状态，必须使用常规函数
    countPlusLocalState (state) {
      return state.count + this.localCount
    }
  })
}
```

8.父子组件传值

a. 父组件传给子组件：子组件通过props方法接受数据;
b. 子组件通过$emit触发父组件的事件，$emit后面的参数是向父组件传参，注意，父组件的事件处理函数直接写函数名即可，不要加(),参数直接传递到了父组件的methods的事件处理函数了。

9.BusEvent

第一步：项目中创建一个js文件（我通常给它取个名字为bus.js），引入vue，创建一个vue实例，导出这个实例，代码如下（一共就两行）：

```
1. import Vue from 'Vue'
2. 2export default new Vue
```

第二步：在两个需要通信的两个组件中分别引入这个bus.js

```
1 import Bus from '这里是你引入bus.js的路径' // Bus可自由更换喜欢的名字
```

第三步：传递数据的组件里通过vue实例方法$emit发送事件名称和需要传递的数据。(发送数据组件)

1 Bus.$emit('click',data) // 这个click是一个自定义的事件名称，data就是你要传递的数据。

第四步：被传递数据的组件内通过vue实例方法$on监听到事件和接受到数据。（接收数据的组件）这里通常挂载监听在vue生命周期created和mounted当中的一个，具体使用场景需要具体的分析，这里不说这个。

```
1.  Bus.$on('click',target => {
2.      console.log(target)    //  注意：发送和监听的事件名称必须一致，target就是获取的数据，可以不写target。只要你喜欢叫什么都可以（当然了，这一定要符合形参变量的命名规范）
3.  })
```

通过以上的四步其实就已经实现了最简单的eventbus的实际应用了。

但是到这儿后，一定要注意一个最容易忽视，又必然不能忘记的东西，那就是清除事件总线eventBus.
不手动清除，它是一直会存在的，这样的话，有个问题就是反复进入到接受数据的组件内操作获取数据，原本只执行一次的获取的操作将会有多次操作。如上我所举的例子，只是打印多次传过来的数据。但你想想，实际开发中是不会这么简单的打印这个数据到控制台，本来只会触发并只执行一次，现在变成了多次，这个问题就非常严重了，你们各种脑补具体的项目开发场景吧。

第五步：在vue生命周期beforeDestroy或者destroyed中用vue实例的$off方法清除eventBus

```
1.  beforeDestroy(){
2.    bus.$off('click')
3.  }
```

总结一下，这里只是介绍如何使用eventBus来解决非父子组件通信，但是当项目较大较复杂时，并不适合。到那时，vuex才是vue给我们提供的最理想的方式。

10.Vue 插件管理，扩展

使用基础 Vue 构造器，创建一个“子类”。参数是一个包含组件选项的对象。

`data` 选项是特例，需要注意 - 在 `Vue.extend()` 中它必须是函数

```
<div id="mount-point"></div>
// 创建构造器
var Profile = Vue.extend({
  template: '<p>{{firstName}} {{lastName}} aka {{alias}}</p>',
  data: function () {
```

```
    return {
      firstName: 'Walter',
      lastName: 'White',
      alias: 'Heisenberg'
    }
  }
})
// 创建 Profile 实例，并挂载到一个元素上。
new Profile().$mount('#mount-point')
```

结果如下：

```
<p>Walter White aka Heisenberg</p>
```

Vue.component(id, [definition])

- **参数**：

  - `{string} id`
  - `{Function | Object} [definition]`

- **用法**：

注册或获取全局组件。注册还会自动使用给定的`id`设置组件的名称

```
// 注册组件，传入一个扩展过的构造器
Vue.component('my-component', Vue.extend({ /* ... */ }))

// 注册组件，传入一个选项对象 (自动调用 Vue.extend)
Vue.component('my-component', { /* ... */ })

// 获取注册的组件 (始终返回构造器)
var MyComponent = Vue.component('my-component')
```

11.Vue mixin

```
// 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!'
})
// => "hello!"
```

算法

其实算法方面在前端的实际项目中涉及得并不多，但还是需要精通一些基础性的算法，一些公司还是会有这方面的需求和考核，建议大家还是需要稍微准备下，这属于加分题。

1. 五大算法

- **贪心算法**: 局部最优解法
- **分治算法**: 分成多个小模块，与原问题性质相同
- **动态规划**: 每个状态都是过去历史的一个总结
- **回溯法**: 发现原先选择不优时，退回重新选择
- **分支限界法**

2. 基础排序算法

- 冒泡排序: 两两比较

```
function bubleSort(arr) {
    var len = arr.length;
    for (let outer = len ; outer >= 2; outer--) {
        for(let inner = 0; inner <=outer - 1; inner++) {
            if(arr[inner] > arr[inner + 1]) {
                [arr[inner],arr[inner+1]] = [arr[inner+1],arr[inner]]
            }
        }
    }
    return arr;
}
```

- 选择排序: 遍历自身以后的元素，最小的元素跟自己调换位置

```
function selectSort(arr) {
    var len = arr.length;
```

```
    for(let i = 0 ;i < len - 1; i++) {
        for(let j = i ; j<len; j++) {
            if(arr[j] < arr[i]) {
                [arr[i],arr[j]] = [arr[j],arr[i]];
            }
        }
    }
    return arr
}
```

- 插入排序: 即将元素插入到已排序好的数组中

```
function insertSort(arr) {
    for(let i = 1; i < arr.length; i++) {  //外循环从1开始，默认arr[0]是有序段
        for(let j = i; j > 0; j--) {  //j = i,将arr[j]依次插入有序段中
            if(arr[j] < arr[j-1]) {
                [arr[j],arr[j-1]] = [arr[j-1],arr[j]];
            } else {
                break;
            }
        }
    }
    return arr;
}
```

3. 高级排序算法

- 快速排序
    - 选择基准值(base)，原数组长度减一(基准值)，使用 splice
    - 循环原数组，小的放左边(left数组)，大的放右边(right数组);
    - concat(left, base, right)
    - 递归继续排序 left 与 right

```
function quickSort(arr) {
    if(arr.length <= 1) {
        return arr;  //递归出口
    }
    var left = [],
        right = [],
        current = arr.splice(0,1);
    for(let i = 0; i < arr.length; i++) {
        if(arr[i] < current) {
            left.push(arr[i])  //放在左边
        } else {
            right.push(arr[i]) //放在右边
        }
    }
    return quickSort(left).concat(current,quickSort(right));
```

```
}
```

- 希尔排序：不定步数的插入排序，插入排序
- 口诀: 插冒归基稳定，快选堆希不稳定

稳定性： 同大小情况下是否可能会被交换位置, 虚拟dom的diff，不稳定性会导致重新渲染；

4. 递归运用(斐波那契数列)： 爬楼梯问题

初始在第一级，到第一级有1种方法(s(1) = 1)，到第二级也只有一种方法(s(2) = 1)， 第三级(s(3) = s(1) + s(2))

```
function cStairs(n) {
    if(n === 1 || n === 2) {
        return 1;
    } else {
        return cStairs(n-1) + cStairs(n-2)
    }
}
```

5. 数据树

- 二叉树: 最多只有两个子节点
    - 完全二叉树
    - 满二叉树
        - 深度为 h, 有 n 个节点，且满足 n = 2^h - 1
- 二叉查找树: 是一种特殊的二叉树，能有效地提高查找效率
    - 小值在左，大值在右
    - 节点 n 的所有左子树值小于 n，所有右子树值大于 n

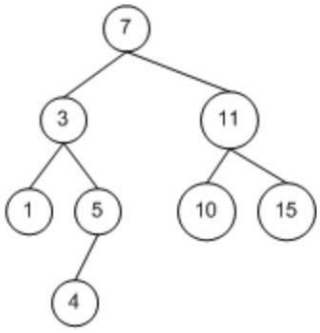


图: 二叉查找树(BST)

- 遍历节点
  - 前序遍历
    - 1. 根节点
    - 2. 访问左子节点，回到 1
    - 3. 访问右子节点，回到 1
  - 中序遍历
    - 3. 先访问到最左的子节点
    - 2. 访问该节点的父节点
    - 3. 访问该父节点的右子节点， 回到 1
  - 后序遍历
    - 3. 先访问到最左的子节点
    - 2. 访问相邻的右节点
    - 3. 访问父节点， 回到 1
- 插入与删除节点

6. 天平找次品

有n个硬币，其中1个为假币，假币重量较轻，你有一把天平，请问，至少需要称多少次能保证一定找到假币?

- 三等分算法:
    -
        1. 将硬币分成3组，随便取其中两组天平称量
        - 平衡，假币在未上称的一组，取其回到 1 继续循环
        - 不平衡，假币在天平上较轻的一组， 取其回到 1 继续循环

大家知道，React 现在已经在前端开发中占据了主导的地位。优异的性能，强大的生态，让其无法阻挡。博主面的 5 家公司，全部是 React 技术栈。据我所知，大厂也大部分以 React 作为主技术栈。React 也成为了面试中并不可少的一环。

进阶知识

框架: React

React 也是现如今最流行的前端框架，也是很多大厂面试必备。React 与 Vue 虽有不同，但同样作为一款 UI 框架，虽然实现可能不一样，但在一些理念上还是有相似的，例如数据驱动、组件化、虚拟 dom 等。这里就主要列举一些 React 中独有的概念。

1. Fiber

React 的核心流程可以分为两个部分:

- reconciliation (**调度算法**，也可称为 render):
    - 更新 state 与 props；
    - 调用生命周期钩子；
    - 生成 virtual dom；
        - 这里应该称为 Fiber Tree 更为符合；
    - 通过新旧 vdom 进行 diff 算法，获取 vdom change；
    - 确定是否需要重新渲染
- commit:

    - 如需要，则操作 dom 节点更新；

要了解 Fiber，我们首先来看为什么需要它?

- **问题**: 随着应用变得越来越庞大，整个更新渲染的过程开始变得吃力，大量的组件渲染会导致主进程长时间被占用，导致一些动画或高频操作出现卡顿和掉帧的情况。而关键点，便是 **同步阻塞**。在之前的调度算法中，React 需要实例化每个类组件，生成一颗组件树，使用 **同步递归** 的方式进行遍历渲染，而这个过程最大的问题就是无法 **暂停和恢复**。
- **解决方案**: 解决同步阻塞的方法，通常有两种: **异步** 与 **任务分割**。而 React Fiber 便是为了实现任务分割而诞生的。
- **简述**:
    - 在 React V16 将调度算法进行了重构， 将之前的 stack reconciler 重构成新版的 fiber reconciler，变成了具有链表和指针的 **单链表树遍历算法**。通过指针映射，每个单元都记录着遍历当下的上一步与下一步，从而使遍历变得可以被暂停和重启。
    - 这里我理解为是一种 **任务分割调度算法**，主要是 将原先同步更新渲染的任务分割成一个个独立的 **小任务单位**，根据不同的优先级，将小任务分散到浏览器的空闲时间执行，充分利用主进程的事件循环机制。
- **核心**:
    - Fiber 这里可以具象为一个 **数据结构**:

```
class Fiber {
    constructor(instance) {
        this.instance = instance
        // 指向第一个 child 节点
        this.child = child
        // 指向父节点
        this.return = parent
        // 指向第一个兄弟节点
        this.sibling = previous
    }
}
```

    - **链表树遍历算法**: 通过 **节点保存与映射**，便能够随时地进行 停止和重启，这样便能达到实现任务分割的基本前提；
        - 1、首先通过不断遍历子节点，到树末尾；
        - 2、开始通过 sibling 遍历兄弟节点；
        - 3、return 返回父节点，继续执行2；
        - 4、直到 root 节点后，跳出遍历；
    - **任务分割**，React 中的渲染更新可以分成两个阶段:

        - **reconciliation 阶段**: vdom 的数据对比，是个适合拆分的阶段，比如对比一部分树后，先暂停执行个动画调用，待完成后再回来继续比对。
        - **Commit 阶段**: 将 change list 更新到 dom 上，并不适合拆分，才能保持数据与 UI 的同步。否则可能由于阻塞 UI 更新，而导致数据更新和 UI 不一致的情况。
    - **分散执行**: 任务分割后，就可以把小任务单元分散到浏览器的空闲期间去排队执行，而实现的关键是两个新API: requestIdleCallback 与 requestAnimationFrame
        - 低优先级的任务交给requestIdleCallback处理，这是个浏览器提供的事件循环空闲期的回调函数，需要 pollyfill，而且拥有 deadline 参数，限制执行事件，以继续切分任务；
        - 高优先级的任务交给requestAnimationFrame处理；

```
// 类似于这样的方式
requestIdleCallback((deadline) => {
    // 当有空闲时间时，我们执行一个组件渲染；
    // 把任务塞到一个个碎片时间中去；
    while ((deadline.timeRemaining() > 0 || deadline.didTimeout) && nextComponent) {
        nextComponent = performWork(nextComponent);
    }
});

```

    - **优先级策略**: 文本框输入 > 本次调度结束需完成的任务 > 动画过渡 > 交互反馈 > 数据更新 > 不会显示但以防将来会显示的任务

Tips:

Fiber 其实可以算是一种编程思想，在其它语言中也有许多应用(Ruby Fiber)。核心思想是 任务拆分和协同，主动把执行权交给主线程，使主线程有时间空挡处理其他高优先级任务。

当遇到进程阻塞的问题时，**任务分割**、**异步调用** 和 **缓存策略** 是三个显著的解决思路。

感谢 @Pengyuan 童鞋，在评论中指出了几个 Fiber 中最核心的理念，感恩！！

2. 生命周期

在新版本中，React 官方对生命周期有了新的 **变动建议**:

- 使用getDerivedStateFromProps 替换componentWillMount；
- 使用getSnapshotBeforeUpdate替换componentWillUpdate；
- 避免使用componentWillReceiveProps；

其实该变动的原因，正是由于上述提到的 Fiber。首先，从上面我们知道 React 可以分成 reconciliation 与 commit 两个阶段，对应的生命周期如下:

- **reconciliation**:
  - componentWillMount
  - componentWillReceiveProps
  - shouldComponentUpdate
  - componentWillUpdate
- **commit**:
  - componentDidMount
  - componentDidUpdate
  - componentWillUnmount

在 Fiber 中，reconciliation 阶段进行了任务分割，涉及到 暂停 和 重启，因此可能会导致 reconciliation 中的生命周期函数在一次更新渲染循环中被 **多次调用** 的情况，产生一些意外错误。

新版的建议生命周期如下:

```
class Component extends React.Component {
  // 替换 `componentWillReceiveProps` ,
  // 初始化和 update 时被调用
  // 静态函数，无法使用 this
  static getDerivedStateFromProps(nextProps, prevState) {}

  // 判断是否需要更新组件
  // 可以用于组件性能优化
  shouldComponentUpdate(nextProps, nextState) {}

  // 组件被挂载后触发
  componentDidMount() {}

  // 替换 componentWillUpdate
  // 可以在更新之前获取最新 dom 数据
  getSnapshotBeforeUpdate() {}

  // 组件更新后调用
  componentDidUpdate() {}

  // 组件即将销毁
  componentWillUnmount() {}

  // 组件已销毁
  componentDidUnMount() {}
}
```

- **使用建议**:

- 在constructor初始化 state；
- 在componentDidMount中进行事件监听，并在componentWillUnmount中解绑事件；
- 在componentDidMount中进行数据的请求，而不是在componentWillMount；
- 需要根据 props 更新 state 时，使用getDerivedStateFromProps(nextProps, prevState)；
    - 旧 props 需要自己存储，以便比较；

```
public static getDerivedStateFromProps(nextProps, prevState) {
   // 当新 props 中的 data 发生变化时，同步更新到 state 上
   if (nextProps.data !== prevState.data) {
      return {
         data: nextProps.data
      }
   } else {
      return null1
   }
}
```

    - 可以在componentDidUpdate监听 props 或者 state 的变化，例如:

```
componentDidUpdate(prevProps) {
  // 当 id 发生变化时，重新获取数据
  if (this.props.id !== prevProps.id) {
          this.fetchData(this.props.id);
  }
}
```

    - 在componentDidUpdate使用setState时，必须加条件，否则将进入死循环；
    - getSnapshotBeforeUpdate(prevProps, prevState)可以在更新之前获取最新的渲染数据，它的调用是在 render 之后， update 之前；
    - shouldComponentUpdate: 默认每次调用setState，一定会最终走到 diff 阶段，但可以通过shouldComponentUpdate的生命钩子返回false来直接阻止后面的逻辑执行，通常是用于做条件渲染，优化渲染的性能。

3. setState

在了解setState之前，我们先来简单了解下 React 一个包装结构:
Transaction:

- **事务** (Transaction):
    - 是 React 中的一个调用结构，用于包装一个方法，结构为: **initialize – perform(method) – close**。通过事务，可以统一管理一个方法的开始与结束；处于事务流中，表示进程正在执行一些操作；

- setState: React 中用于修改状态，更新视图。它具有以下特点:
- **异步与同步**: setState并不是单纯的异步或同步，这其实与调用时的环境相关:
    - 在 **合成事件** 和 **生命周期钩子(除 componentDidUpdate)** 中，setState是"异步"的；
        - **原因**: 因为在setState的实现中，有一个判断: 当更新策略正在事务流的执行中时，该组件更新会被推入dirtyComponents队列中等待执行；否则，开始执行batchedUpdates队列更新；
            - 在生命周期钩子调用中，更新策略都处于更新之前，组件仍处于事务流中，而componentDidUpdate是在更新之后，此时组件已经不在事务流中了，因此则会同步执行；
            - 在合成事件中，React 是基于 **事务流完成的事件委托机制** 实现，也是处于事务流中；
        - **问题**: 无法在setState后马上从this.state上获取更新后的值。
        - **解决**: 如果需要马上同步去获取新值，setState其实是可以传入第二个参数的。setState(updater, callback)，在回调中即可获取最新值；
    - 在 **原生事件** 和 **setTimeout** 中，setState是同步的，可以马上获取更新后的值；
        - 原因: 原生事件是浏览器本身的实现，与事务流无关，自然是同步；而setTimeout是放置于定时器线程中延后执行，此时事务流已结束，因此也是同步；
- **批量更新**: 在 **合成事件** 和 **生命周期钩子** 中，setState更新队列时，存储的是 **合并状态**(Object.assign)。因此前面设置的 key 值会被后面所覆盖，最终只会执行一次更新；
- **函数式**: 由于 Fiber 及 合并 的问题，官方推荐可以传入 **函数** 的形式。setState(fn)，在fn中返回新的state对象即可，例如 this.setState((state, props) => newState)；
    - 使用函数式，可以用于避免setState的批量更新的逻辑，传入的函数将会被 **顺序调用**；
- **注意事项**:

    - setState 合并，在 合成事件 和 生命周期钩子 中多次连续调用会被优化为一次；
    - 当组件已被销毁，如果再次调用setState，React 会报错警告，通常有两种解决办法:
        - 将数据挂载到外部，通过 props 传入，如放到 Redux 或 父级中；
        - 在组件内部维护一个状态量 (isUnmounted)，componentWillUnmount中标记为 true，在setState前进行判断；

4. HOC(高阶组件)

HOC(Higher Order Componennt) 是在 React 机制下社区形成的一种组件模式，在很多第三方开源库中表现强大。

- **简述**:
    - 高阶组件不是组件，是 **增强函数**，可以输入一个元组件，返回出一个新的增强组件；
    - 高阶组件的主要作用是 **代码复用**，**操作** 状态和参数；
- **用法**:
    - **属性代理 (Props Proxy)**: 返回出一个组件，它基于被包裹组件进行 **功能增强**；
- **默认参数**: 可以为组件包裹一层默认参数；

```
function proxyHoc(Comp) {
    return class extends React.Component {
        render() {
            const newProps = {
                name: 'tayde',
                age: 1,
            }
            return <Comp {...this.props} {...newProps} />
        }
    }
}
```

- **提取状态**: 可以通过 props 将被包裹组件中的 state 依赖外层，例如用于转换受控组件:

```
function withOnChange(Comp) {
    return class extends React.Component {
        constructor(props) {
            super(props)
            this.state = {
                name: '',
            }
```

```
        }
        onChangeName = () => {
            this.setState({
                name: 'dongdong',
            })
        }
        render() {
            const newProps = {
                value: this.state.name,
                onChange: this.onChangeName,
            }
            return <Comp {...this.props} {...newProps} />
        }
    }
}
```

使用姿势如下，这样就能非常快速的将一个 Input 组件转化成受控组件。

```
const NameInput = props => (<input name="name" {...props} />)
export default withOnChange(NameInput)
```

- **包裹组件**: 可以为被包裹元素进行一层包装，

```
function withMask(Comp) {
  return class extends React.Component {
      render() {
          return (
              <div>
                  <Comp {...this.props} />
                    <div style={{
                      width: '100%',
                      height: '100%',
                      backgroundColor: 'rgba(0, 0, 0, .6)',
                    }}
              </div>
          )
      }
  }
}
```

- **反向继承** (Inheritance Inversion): 返回出一个组件，**继承于被包裹组件**，常用于以下操作:

```
function IIHoc(Comp) {
    return class extends Comp {
        render() {
            return super.render();
        }
    };
}
```

- **渲染劫持** (Render Highjacking)

- **条件渲染**: 根据条件，渲染不同的组件

```
function withLoading(Comp) {
    return class extends Comp {
        render() {
            if(this.props.isLoading) {
                return <Loading />
            } else {
                return super.render()
            }
        }
    };
}
```

- 可以直接修改被包裹组件渲染出的 React 元素树
- **操作状态** (Operate State): 可以直接通过 this.state 获取到被包裹组件的状态，并进行操作。但这样的操作容易使 state 变得难以追踪，不易维护，谨慎使用。
- **应用场景**:
    - **权限控制**，通过抽象逻辑，统一对页面进行权限判断，按不同的条件进行页面渲染:

```
function withAdminAuth(WrappedComponent) {
    return class extends React.Component {
        constructor(props){
            super(props)
            this.state = {
                isAdmin: false,
            }
        }
        async componentWillMount() {
            const currentRole = await getCurrentUserRole();
            this.setState({
                isAdmin: currentRole === 'Admin',
            });
        }
        render() {
            if (this.state.isAdmin) {
                return <Comp {...this.props} />;
            } else {
                return (<div>您没有权限查看该页面，请联系管理员！</div>);
            }
        }
    };
}
```

    - **性能监控**，包裹组件的生命周期，进行统一埋点:

```
function withTiming(Comp)  {
    return class extends Comp {
```

```
        constructor(props) {
            super(props);
            this.start = Date.now();
            this.end = 0;
        }
        componentDidMount() {
            super.componentDidMount && super.componentDidMount();
            this.end = Date.now();
            console.log(`${WrappedComponent.name} 组件渲染时间为 ${this.end - this.start} ms`);
        }
        render() {
            return super.render();
        }
    };
}
```

- **代码复用**，可以将重复的逻辑进行抽象。

- 使用注意:

1. **纯函数**: 增强函数应为纯函数，避免侵入修改元组件；

   - **避免用法污染**: 理想状态下，应透传元组件的无关参数与事件，尽量保证用法不变；

3. **命名空间**: 为 HOC 增加特异性的组件名称，这样能便于开发调试和查找问题；

4. **引用传递**: 如果需要传递元组件的 refs 引用，可以使用React.forwardRef；

5. **静态方法**: 元组件上的静态方法并无法被自动传出，会导致业务层无法调用；解决:
   - 函数导出
   - 静态方法赋值

6. **重新渲染**: 由于增强函数每次调用是返回一个新组件，因此如果在 Render 中使用增强函数，就会导致每次都重新渲染整个HOC，而且之前的状态会丢失；

5. Redux

Redux 是一个 **数据管理中心**，可以把它理解为一个全局的 data store 实例。它通过一定的使用规则和限制，保证着数据的健壮性、可追溯和可预测性。它

与 React 无关，可以独立运行于任何 JavaScript 环境中，从而也为同构应用提供了更好的数据同步通道。

- **核心理念**:
  - **单一数据源**: 整个应用只有唯一的状态树，也就是所有 state 最终维护在一个根级 Store 中；
  - **状态只读**: 为了保证状态的可控性，最好的方式就是监控状态的变化。那这里就两个必要条件：
    - Redux Store 中的数据无法被直接修改；
    - 严格控制修改的执行；
  - **纯函数**: 规定只能通过一个纯函数 (Reducer) 来描述修改；
- 大致的数据结构如下所示:

- **理念实现**:
  - **Store**: 全局 Store 单例， 每个 Redux 应用下只有一个 store， 它具有以下方法供使用:
    - getState: 获取 state；
    - dispatch: 触发 action, 更新 state；
    - subscribe: 订阅数据变更，注册监听器；

```
  // 创建
 const store = createStore(Reducer, initStore)

Action: 它作为一个行为载体，用于映射相应的 Reducer，并且它可以成为数据的载体，将数据从应用传递至 store 中，是 store 唯一的数据源；
// 一个普通的 Action
const action = {
    type: 'ADD_LIST',
    item: 'list-item-1',
}

// 使用：
store.dispatch(action)

// 通常为了便于调用，会有一个 Action 创建函数 (action creater)
funtion addList(item) {
    return const action = {
        type: 'ADD_LIST',
        item,
    }
}

// 调用就会变成:
dispatch(addList('list-item-1'))
```

    - **Reducer**: 用于描述如何修改数据的纯函数，Action 属于行为名称，而 Reducer 便是修改行为的实质；

```
// 一个常规的 Reducer
// @param {state}: 旧数据
// @param {action}: Action 对象
// @returns {any}: 新数据
const initList = []
function ListReducer(state = initList, action) {
    switch (action.type) {
        case 'ADD_LIST':
            return state.concat([action.item])
            break
        defalut:
            return state
    }
}
```

注意:

1. 遵守数据不可变，不要去直接修改 state，而是返回出一个 **新对象**，可以使用 assign / copy / extend / 解构 等方式创建新对象；
2. 默认情况下需要 **返回原数据**，避免数据被清空；
3. 最好设置 **初始值**，便于应用的初始化及数据稳定；

- **进阶**:
    - **React-Redux**: 结合 React 使用；
        - <Provider>: 将 store 通过 context 传入组件中；
        - connect: 一个高阶组件，可以方便在 React 组件中使用 Redux；
        1. 将store通过mapStateToProps进行筛选后使用props注入组件
        2. 根据mapDispatchToProps创建方法，当组件调用时使用dispatch触发对应的action
    - **Reducer 的拆分与重构**:
        - 随着项目越大，如果将所有状态的 reducer 全部写在一个函数中，将会 **难以维护**；
        - 可以将 reducer 进行拆分，也就是 **函数分解**，最终再使用 combineReducers()进行重构合并；
    - **异步 Action**: 由于 Reducer 是一个严格的纯函数，因此无法在 Reducer 中进行数据的请求，需要先获取数据，再dispatch(Action)即可，下面是三种不同的异步实现:
        - redex-thunk
        - redux-saga

    - redux-observable

6. React Hooks

React 中通常使用 **类定义** 或者 **函数定义** 创建组件:

在类定义中，我们可以使用到许多 React 特性，例如 state、 各种组件生命周期钩子等，但是在函数定义中，我们却无能为力，因此 React 16.8 版本推出了一个新功能 (React Hooks)，通过它，可以更好的在函数定义组件中使用 React 特性。

- **好处**:
    - 1、**跨组件复用**: 其实 render props / HOC 也是为了复用，相比于它们，Hooks 作为官方的底层 API，最为轻量，而且改造成本小，不会影响原来的组件层次结构和传说中的嵌套地狱；
    - 2、**类定义更为复杂**:
        - 不同的生命周期会使逻辑变得分散且混乱，不易维护和管理；
        - 时刻需要关注this的指向问题；
        - 代码复用代价高，高阶组件的使用经常会使整个组件树变得臃肿；
    - 3、**状态与UI隔离**: 正是由于 Hooks 的特性，状态逻辑会变成更小的粒度，并且极容易被抽象成一个自定义 Hooks，组件中的状态和 UI 变得更为清晰和隔离。
- **注意**:
    - 避免在 循环/条件判断/嵌套函数 中调用 hooks，保证调用顺序的稳定；
    - 只有 函数定义组件 和 hooks 可以调用 hooks，避免在 类组件 或者 普通函数 中调用；
    - 不能在useEffect中使用useState，React 会报错提示；
    - 类组件不会被替换或废弃，不需要强制改造类组件，两种方式能并存；
- **重要钩子***:
    - **状态钩子** (useState): 用于定义组件的 State，其到类定义中this.state的功能；

```
// useState 只接受一个参数: 初始状态
// 返回的是组件名和更改该组件对应的函数
const [flag, setFlag] = useState(true);
// 修改状态
setFlag(false)

// 上面的代码映射到类定义中:
this.state = {
    flag: true
}
```

```
const flag = this.state.flag
const setFlag = (bool) => {
    this.setState({
        flag: bool,
    })
}
```

- **生命周期钩子** (useEffect):

类定义中有许多生命周期函数，而在 React Hooks 中也提供了一个相应的函数 (useEffect)，这里可以看做componentDidMount、componentDidUpdate和componentWillUnmount的结合。

- useEffect(callback, [source])接受两个参数
  - callback: 钩子回调函数；
  - source: 设置触发条件，仅当 source 发生改变时才会触发；
  - useEffect钩子在没有传入[source]参数时，默认在每次 render 时都会优先调用上次保存的回调中返回的函数，后再重新调用回调；

useEffect(() => {

```
  // 组件挂载后执行事件绑定
    console.log('on')
    addEventListener()

    // 组件 update 时会执行事件解绑
    return () => {
        console.log('off')
        removeEventListener()
    }
}, [source]);
```

// 每次 source 发生改变时，执行结果(以类定义的生命周期，便于大家理解):
// --- DidMount ---
// 'on'
// --- DidUpdate ---
// 'off'
// 'on'
// --- DidUpdate ---
// 'off'
// 'on'
// --- WillUnmount ---
// 'off'

- 通过第二个参数，我们便可模拟出几个常用的生命周期:
  - componentDidMount: 传入[]时，就只会在初始化时调用一次；
- const useMount = (fn) => useEffect(fn, [])

  - componentWillUnmount: 传入[]，回调中的返回的函数也只会被最终执行一次；

  const useUnmount = (fn) => useEffect(() => fn, [])

  - mounted: 可以使用 useState 封装成一个高度可复用的 mounted 状态；

  ```
  const useMounted = () => {
    const [mounted, setMounted] = useState(false);
    useEffect(() => {
      !mounted && setMounted(true);
      return () => setMounted(false);
    }, []);
    return mounted;
  }
  ```

  - componentDidUpdate: useEffect每次均会执行，其实就是排除了 DidMount 后即可；

  ```
  const mounted = useMounted()
  useEffect(() => {
    mounted && fn()
  })
  ```

- **其它内置钩子**:
  - useContext: 获取 context 对象
  - useReducer: 类似于 Redux 思想的实现，但其并不足以替代 Redux，可以理解成一个组件内部的 redux:
    - 并不是持久化存储，会随着组件被销毁而销毁；
    - 属于组件内部，各个组件是相互隔离的，单纯用它并无法共享数据；
    - 配合useContext的全局性，可以完成一个轻量级的 Redux；(easy-peasy)
  - useCallback: 缓存回调函数，避免传入的回调每次都是新的函数实例而导致依赖组件重新渲染，具有性能优化的效果；

    - useMemo: 用于缓存传入的 props，避免依赖的组件每次都重新渲染；
    - useRef: 获取组件的真实节点；
    - useLayoutEffect:
        - DOM更新同步钩子。用法与useEffect类似，只是区别于执行时间点的不同。
        - useEffect属于异步执行，并不会等待 DOM 真正渲染后执行，而useLayoutEffect则会真正渲染后才触发；
        - 可以获取更新后的 state；
- **自定义钩子**(useXxxxx): 基于 Hooks 可以引用其它 Hooks 这个特性，我们可以编写自定义钩子，如上面的useMounted。又例如，我们需要每个页面自定义标题:

```
function useTitle(title) {
  useEffect(
    () => {
      document.title = title;
    });
}

// 使用:
function Home() {
    const title = '我是首页'
    useTitle(title)

    return (
        <div>{title}</div>
    )
}
```

7. SSR

SSR，俗称 **服务端渲染** (Server Side Render)，讲人话就是: 直接在服务端层获取数据，渲染出完成的 HTML 文件，直接返回给用户浏览器访问。

- **前后端分离**: 前端与服务端隔离，前端动态获取数据，渲染页面。
- **痛点**:
    - **首屏渲染性能瓶颈**:
        - 空白延迟: HTML下载时间 + JS下载/执行时间 + 请求时间 + 渲染时间。在这段时间内，页面处于空白的状态。

    - **SEO 问题**: 由于页面初始状态为空，因此爬虫无法获取页面中任何有效数据，因此对搜索引擎不友好。
        - 虽然一直有在提动态渲染爬虫的技术，不过据我了解，大部分国内搜索引擎仍然是没有实现。

最初的服务端渲染，便没有这些问题。但我们不能返璞归真，既要保证现有的前端独立的开发模式，又要由服务端渲染，因此我们使用 React SSR。

- **原理**:
    - Node 服务: 让前后端运行同一套代码成为可能。
    - Virtual Dom: 让前端代码脱离浏览器运行。
- **条件**: Node 中间层、 React / Vue 等框架。 结构大概如下:

- **开发流程**: (此处以 React + Router + Redux + Koa 为例)
    - 1、在同个项目中，**搭建** 前后端部分，常规结构:

build

public

src

client

server

- 2、server 中使用 Koa **路由监听** 页面访问:

```
import * as Router from 'koa-router'

const router = new Router()
// 如果中间也提供 Api 层
router.use('/api/home', async () => {
    // 返回数据
})

router.get('*', async (ctx) => {
    // 返回 HTML
})
```

    - 3、通过访问 url **匹配** 前端页面路由:

```
// 前端页面路由
import { pages } from '../../client/app'
import { matchPath } from 'react-router-dom'

// 使用 react-router 库提供的一个匹配方法
const matchPage = matchPath(ctx.req.url, page)
```

- 4、通过页面路由的配置进行 **数据获取**。通常可以在页面路由中增加 SSR 相关的静态配置，用于抽象逻辑，可以保证服务端逻辑的通用性，如:

```
class HomePage extends React.Component{
        public static ssrConfig = {
          cache: true,
         fetch() {
             // 请求获取数据
         }
    }
}
```

获取数据通常有两种情况:

- 中间层也使用 **http** 获取数据，则此时 fetch 方法可前后端共享；

```
const data = await matchPage.ssrConfig.fetch()
```

- 中间层并不使用 http，是通过一些 **内部调用**，例如 Rpc 或 直接读数据库 等，此时也可以直接由服务端调用对应的方法获取数据。通常，这里需要在 ssrConfig 中配置特异性的信息，用于匹配对应的数据获取方法。

```
// 页面路由
class HomePage extends React.Component{
    public static ssrConfig = {
        fetch: {
            url: '/api/home',
        }
    }
}

// 根据规则匹配出对应的数据获取方法
// 这里的规则可以自由，只要能匹配出正确的方法即可
const controller = matchController(ssrConfig.fetch.url)

// 获取数据
const data = await controller(ctx)
```

- 5、创建 Redux store，并将数据dispatch到里面:

```
import { createStore } from 'redux'
// 获取 Clinet层 reducer
// 必须复用前端层的逻辑，才能保证一致性；
import { reducers } from '../../client/store'

// 创建 store
const store = createStore(reducers)
```

```
// 获取配置好的 Action
const action = ssrConfig.action
// 存储数据
store.dispatch(createAction(action)(data))
```

- 6、注入 Store， 调用renderToString将 React Virtual Dom 渲染成 **字符串**:

```
import * as ReactDOMServer from 'react-dom/server'
import { Provider } from 'react-redux'

// 获取 Clinet 层根组件
import { App } from '../../client/app'

const AppString = ReactDOMServer.renderToString(
    <Provider store={store}>
        <StaticRouter
            location={ctx.req.url}
            context={{}}>
            <App />
        </StaticRouter>
    </Provider>
)
```

- 7、将 AppString 包装成完整的 html 文件格式；
- 8、此时，已经能生成完整的 HTML 文件。但只是个纯静态的页面，没有样式没有交互。接下来我们就是要插入 JS 与 CSS。我们可以通过访问前端打包后生成的asset-manifest.json文件来获取相应的文件路径，并同样注入到 Html 中引用。

```
const html = `
    <!DOCTYPE html>
    <html lang="zh">
        <head></head>
        <link href="${cssPath}" rel="stylesheet" />
        <body>
            <div id="App">${AppString}</div>
            <script src="${scriptPath}"></script>
        </body>
    </html>
`
```

    - 9、进行 **数据脱水**: 为了把服务端获取的数据同步到前端。主要是将数据序列化后，插入到 html 中，返回给前端。

```
import serialize from 'serialize-javascript'
// 获取数据
const initState = store.getState()
const html = `
    <!DOCTYPE html>
    <html lang="zh">
```

```
        <head></head>
        <body>
            <div id="App"></div>
            <script type="application/json" id="SSR_HYDRATED_DATA">${serialize(initState)}</script>
        </body>
    </html>
`

ctx.status = 200
ctx.body = html
```

Tips:

这里比较特别的有两点:

1. 使用了serialize-javascript序列化 store， 替代了JSON.stringify，保证数据的安全性，避免代码注入和 XSS 攻击；
2. 使用 json 进行传输，可以获得更快的加载速度；
    - 10、Client 层 **数据吸水**: 初始化 store 时，以脱水后的数据为初始化数据，同步创建 store。

```
const hydratedEl = document.getElementById('SSR_HYDRATED_DATA')
const hydrateData = JSON.parse(hydratedEl.textContent)

// 使用初始 state 创建 Redux store
const store = createStore(reducer, hydrateData)
```

8. 函数式编程

函数式编程是一种 **编程范式**，你可以理解为一种软件架构的思维模式。它有着独立一套理论基础与边界法则，追求的是 **更简洁、可预测、高复用、易测试**。其实在现有的众多知名库中，都蕴含着丰富的函数式编程思想，如 React / Redux 等。

- **常见的编程范式**:
    - 命令式编程(过程化编程): 更关心解决问题的步骤，一步步以语言的形式告诉计算机做什么；
    - 事件驱动编程: 事件订阅与触发，被广泛用于 GUI 的编程设计中；
    - 面向对象编程: 基于类、对象与方法的设计模式，拥有三个基础概念: 封装性、继承性、多态性；
    - 函数式编程
        - 换成一种更高端的说法，面向数学编程。怕不怕~😉

- **函数式编程的理念**:
    - **纯函数**(确定性函数): 是函数式编程的基础，可以使程序变得灵活，高度可拓展，可维护；
        - **优势**:
            - 完全独立，与外部解耦；
            - 高度可复用，在任意上下文，任意时间线上，都可执行并且保证结果稳定；
            - 可测试性极强；
        - **条件**:
            - 不修改参数；
            - 不依赖、不修改任何函数外部的数据；
            - 完全可控，参数一样，返回值一定一样: 例如函数不能包含new Date()或者Math.randon()等这种不可控因素；
            - 引用透明；
        - 我们常用到的许多 API 或者工具函数，它们都具有着纯函数的特点， 如split / join / map；
    - **函数复合**: 将多个函数进行组合后调用，可以实现将一个个函数单元进行组合，达成最后的目标；
        - **扁平化嵌套**: 首先，我们一定能想到组合函数最简单的操作就是 包裹，因为在 JS 中，函数也可以当做参数:
            - f(g(k(x))): 嵌套地狱，可读性低，当函数复杂后，容易让人一脸懵逼；
            - 理想的做法: xxx(f, g, k)(x)
        - **结果传递**: 如果想实现上面的方式，那也就是xxx函数要实现的便是: 执行结果在各个函数之间的执行传递；
            - 这时我们就能想到一个原生提供的数组方法: reduce，它可以按数组的顺序依次执行，传递执行结果；
            - 所以我们就能够实现一个方法pipe，用于函数组合:

```
// ...fs: 将函数组合成数组；
// Array.prototype.reduce 进行组合；
// p: 初始参数；
const pipe = (...fs) => p => fs.reduce((v, f) => f(v), p)
```

        - **使用**: 实现一个 驼峰命名 转 中划线命名 的功能:

```
// 'Guo DongDong' --> 'guo-dongdong'
// 函数组合式写法
const toLowerCase = str => str.toLowerCase()
```

```
const join = curry((str, arr) => arr.join(str))
const split = curry((splitOn, str) => str.split(splitOn));

const toSlug = pipe(
    toLowerCase,
    split(' '),
    join('_'),
    encodeURIComponent,
);
console.log(toSlug('Guo DongDong')
```

)

- **好处**:
  - 隐藏中间参数，不需要临时变量，避免了这个环节的出错几率；
  - 只需关注每个纯函数单元的稳定，不再需要关注命名，传递，调用等；
  - 可复用性强，任何一个函数单元都可被任意复用和组合；
  - 可拓展性强，成本低，例如现在加个需求，要查看每个环节的输出:

```
const log = curry((label, x) => {
    console.log(`${ label }: ${ x }`);
    return x;
});

const toSlug = pipe(
    toLowerCase,
    log('toLowerCase output'),
    split(' '),
    log('split output'),
    join('_'),
    log('join output'),
    encodeURIComponent,
);
```

Tips:

一些工具纯函数可直接引用lodash/fp，例如curry/map/split等，并不需要像我们上面这样自己实现；

- **数据不可变性**(immutable): 这是一种数据理念，也是函数式编程中的核心理念之一:
  - **倡导**: 一个对象再被创建后便不会再被修改。当需要改变值时，是返回一个全新的对象，而不是直接在原对象上修改；
  - **目的**: 保证数据的稳定性。避免依赖的数据被未知地修改，导致了自身的执行异常，能有效提高可控性与稳定性；

        - 并不等同于const。使用const创建一个对象后，它的属性仍然可以被修改；
        - 更类似于Object.freeze: 冻结对象，但freeze仍无法保证深层的属性不被串改；
        - immutable.js: js 中的数据不可变库，它保证了数据不可变，在 React 生态中被广泛应用，大大提升了性能与稳定性；
            - trie数据结构:
                - 一种数据结构，能有效地深度冻结对象，保证其不可变；
                - **结构共享**: 可以共用不可变对象的内存引用地址，减少内存占用，提高数据操作性能；
    - 避免不同函数之间的 **状态共享**，数据的传递使用复制或全新对象，遵守数据不可变原则；
    - 避免从函数内部 **改变外部状态**，例如改变了全局作用域或父级作用域上的变量值，可能会导致其它单位错误；
    - 避免在单元函数内部执行一些 **副作用**，应该将这些操作抽离成更独立的工具单元；
        - 日志输出
        - 读写文件
        - 网络请求
        - 调用外部进程
        - 调用有副作用的函数
- **高阶函数**: 是指 以函数为参数，返回一个新的增强函数 的一类函数，它通常用于:
    - 将逻辑行为进行 **隔离抽象**，便于快速复用，如处理数据，兼容性等；
    - **函数组合**，将一系列单元函数列表组合成功能更强大的函数；
    - **函数增强**，快速地拓展函数功能，
- **函数式编程的好处**:
    - 函数副作用小，所有函数独立存在，没有任何耦合，复用性极高；
    - 不关注执行时间，执行顺序，参数，命名等，能专注于数据的流动与处理，能有效提高稳定性与健壮性；
    - 追求单元化，粒度化，使其重构和改造成本降低，可维护、可拓展性较好；
    - 更易于做单元测试。
- **总结**:

- 函数式编程其实是一种编程思想，它追求更细的粒度，将应用拆分成一组组极小的单元函数，组合调用操作数据流；
- 它提倡着 纯函数 / 函数复合 / 数据不可变， 谨慎对待函数内的 状态共享 / 依赖外部 / 副作用；

进阶知识

Hybrid

随着 Web技术 和 移动设备 的快速发展，在各家大厂中，Hybrid 技术已经成为一种最主流最不可取代的架构方案之一。一套好的 Hybrid 架构方案能让 App 既能拥有 **极致的体验和性能**，同时也能拥有 Web技术 **灵活的开发模式、跨平台能力以及热更新机制**。因此，相关的 Hybrid 领域人才也是十分的吃香，精通Hybrid 技术和相关的实战经验，也是面试中一项大大的加分项。

1. 混合方案简析

Hybrid App，俗称 **混合应用**，即混合了 Native技术 与 Web技术 进行开发的移动应用。现在比较流行的混合方案主要有三种，主要是在UI渲染机制上的不同:

- **Webview UI**:
  - 通过 JSBridge 完成 H5 与 Native 的双向通讯，并 **基于 Webview** 进行页面的渲染；
  - **优势**: 简单易用，架构门槛/成本较低，适用性与灵活性极强；
  - **劣势**: Webview 性能局限，在复杂页面中，表现远不如原生页面；
- **Native UI**:
  - 通过 JSBridge 赋予 H5 原生能力，并进一步将 JS 生成的虚拟节点树(Virtual DOM)传递至 Native 层，并使用 **原生系统渲染**。
  - **优势**: 用户体验基本接近原生，且能发挥 Web技术 开发灵活与易更新的特性；
  - **劣势**: 上手/改造门槛较高，最好需要掌握一定程度的客户端技术。相比于常规 Web开发，需要更高的开发调试、问题排查成本；
- **小程序**

    - 通过更加定制化的 JSBridge，赋予了 Web 更大的权限，并使用双 WebView 双线程的模式隔离了 JS逻辑 与 UI渲染，形成了特殊的开发模式，加强了 H5 与 Native 混合程度，属于第一种方案的优化版本；
    - **优势**: 用户体验好于常规 Webview 方案，且通常依托的平台也能提供更为友好的开发调试体验以及功能；
    - **劣势**: 需要依托于特定的平台的规范限定

2. Webviev

Webview 是 Native App 中内置的一款基于 Webkit内核 的浏览器，主要由两部分组成:

- **WebCore 排版引擎**；
- **JSCore 解析引擎**；

在原生开发 SDK 中 Webview 被封装成了一个组件，用于作为 Web页面 的容器。因此，作为宿主的客户端中拥有更高的权限，可以对 Webview 中的 Web 页面 进行配置和开发。

Hybrid技术中双端的交互原理，便是基于 Webview 的一些 API 和特性。

3. 交互原理

Hybrid技术 中最核心的点就是 Native端 与 H5端 之间的 **双向通讯层**，其实这里也可以理解为我们需要一套 **跨语言通讯方案**，便是我们常听到的 JSBridge。

- JavaScript 通知 Native
    - **API注入**，Native 直接在 JS 上下文中挂载数据或者方法
        - 延迟较低，在安卓4.1以下具有安全性问题，风险较高
    - WebView **URL Scheme** 跳转拦截
        - 兼容性好，但延迟较高，且有长度限制
    - WebView 中的 **prompt/console/alert拦截**(通常使用 prompt)
- **Native 通知 Javascript**:
    - **IOS**: stringByEvaluatingJavaScriptFromString
- // Swift
- webview.stringByEvaluatingJavaScriptFromString("alert('NativeCall')
  ")

    - **Android**: loadUrl (4.4–)

```
// 调用js中的JSBridge.trigger方法
// 该方法的弊端是无法获取函数返回值；
webView.loadUrl("javascript:JSBridge.trigger('NativeCall')")
```

- **Android**: evaluateJavascript (4.4+)

```
// 4.4+后使用该方法便可调用并获取函数返回值；
mWebView.evaluateJavascript（"javascript:JSBridge.trigger('NativeCall')",　　new ValueCallback<String>() {
    @Override
    public void onReceiveValue(String value) {
        //此处为 js 返回的结果
    }
});
```

4. 接入方案

整套方案需要 Web 与 Native 两部分共同来完成:

- **Native**: 负责实现URL拦截与解析、环境信息的注入、拓展功能的映射、版本更新等功能；
- **JavaScirpt**: 负责实现功能协议的拼装、协议的发送、参数的传递、回调等一系列基础功能。

接入方式:

- **在线H5**: 直接将项目部署于线上服务器，并由客户端在 HTML 头部注入对应的 Bridge。
  - **优势**: 接入/开发成本低，对 App 的侵入小；
  - **劣势**: 重度依赖网络，无法离线使用，首屏加载慢；
- **内置离线包**: 将代码直接内置于 App 中，即本地存储中，可由 H5 或者 客户端引用 Bridge。
  - **优势**: 首屏加载快，可离线化使用；
  - **劣势**: 开发、调试成本变高，需要多端合作，且会增加 App 包体积

5. 优化方案简述

- **Webview 预加载**: Webview 的初始化其实挺耗时的。我们测试过，大概在100~200ms之间，因此如果能前置做好初始化于内存中，会大大加快渲染速度。

- **更新机制**: 使用离线包的时候，便会涉及到本地离线代码的更新问题，因此需要建立一套云端下发包的机制，由客户端下载云端最新代码包 (zip包)，并解压替换本地代码。
    - **增量更新**: 由于下发包是一个下载的过程，因此包的体积越小，下载速度越快，流量损耗越低。只打包改变的文件，客户端下载后覆盖式替换，能大大减小每次更新包的体积。
    - **条件分发**: 云平台下发更新包时，可以配合客户端设置一系列的条件与规则，从而实现代码的条件更新:
        - 单 **地区** 更新: 例如一个只有中国地区才能更新的版本；
        - 按 **语言** 更新: 例如只有中文版本会更新；
        - 按 App **版本** 更新: 例如只有最新版本的 App 才会更新；
        - **灰度** 更新: 只有小比例用户会更新；
        - **AB测试**: 只有命中的用户会更新；
- **降级机制**: 当用户下载或解压代码包失败时，需要有套降级方案，通常有两种做法:
    - **本地内置**: 随着 App 打包时内置一份线上最新完整代码包，保证本地代码文件的存在，资源加载均使用本地化路径；
    - **域名拦截**: 资源加载使用线上域名，通过拦截域名映射到本地路径。当本地不存在时，则请求线上文件，当存在时，直接加载；
- **跨平台部署**: Bridge层 可以做一套浏览器适配，在一些无法适配的功能，做好降级处理，从而保证代码在任何环境的可用性，一套代码可同时运行于 App内 与 普通浏览器；
- **环境系统**: 与客户端进行统一配合，搭建出 **正式 / 预上线 / 测试 / 开发**环境，能大大提高项目稳定性与问题排查；
- **开发模式**:
    - 能连接PC Chrome/safari 进行代码调试；
    - 具有开发调试入口，可以使用同样的 Webview 加载开发时的本地代码；
    - 具备日志系统，可以查看 Log 信息；

Webpack

1. 原理简述

Webpack 已经成为了现在前端工程化中最重要的一环，通过Webpack与Node的配合，前端领域完成了不可思议的进步。通过预编译，将软件编程中先进的思想和理念能够真正运用于生产，让前端开发领域告别原始的蛮荒阶段。深入理解Webpack，可以让你在编程思维及技术领域上产生质的成长，极大拓展技术边界。这也是在面试中必不可少的一个内容。

- **核心概念**
    - JavaScript 的 **模块打包工具** (module bundler)。通过分析模块之间的依赖，最终将所有模块打包成一份或者多份代码包 (bundler)，供 HTML 直接引用。实质上，Webpack 仅仅提供了 **打包功能** 和一套 **文件处理机制**，然后通过生态中的各种 Loader 和 Plugin 对代码进行预编译和打包。因此 Webpack 具有高度的可拓展性，能更好的发挥社区生态的力量。
        - **Entry**: 入口文件，Webpack 会从该文件开始进行分析与编译；
        - **Output**: 出口路径，打包后创建 bundler 的文件路径以及文件名；
        - **Module**: 模块，在 Webpack 中任何文件都可以作为一个模块，会根据配置的不同的 Loader 进行加载和打包；
        - **Chunk**: 代码块，可以根据配置，将所有模块代码合并成一个或多个代码块，以便按需加载，提高性能；
        - **Loader**: 模块加载器，进行各种文件类型的加载与转换；
        - **Plugin**: 拓展插件，可以通过 Webpack 相应的事件钩子，介入到打包过程中的任意环节，从而对代码按需修改；
- **工作流程** (加载 – 编译 – 输出)
    - 1、读取配置文件，按命令 **初始化** 配置参数，创建 Compiler 对象；
    - 2、调用插件的 apply 方法 **挂载插件** 监听，然后从入口文件开始执行编译；
    - 3、按文件类型，调用相应的 Loader 对模块进行 **编译**，并在合适的时机点触发对应的事件，调用 Plugin 执行，最后再根据模块 **依赖查找** 到所依赖的模块，递归执行第三步；
    - 4、将编译后的所有代码包装成一个个代码块 (Chuck)， 并按依赖和配置确定 **输出内容**。这个步骤，仍然可以通过 Plugin 进行文件的修改；
    - 5、最后，根据 Output 把文件内容一一写入到指定的文件夹中，完成整个过程；
- **模块包装**:

```
(function(modules) {
  // 模拟 require 函数，从内存中加载模块；
  function __webpack_require__(moduleId) {
    // 缓存模块
    if (installedModules[moduleId]) {
      return installedModules[moduleId].exports;
    }

    var module = installedModules[moduleId] = {
      i: moduleId,
      l: false,
      exports: {}
    };

    // 执行代码；
```

```
        modules[moduleId].call(module.exports, module, module.exports, __webpack_require__);

        // Flag: 标记是否加载完成；
        module.l = true;

        return module.exports;
    }

    // ...

    // 开始执行加载入口文件；
    return __webpack_require__(__webpack_require__.s = "./src/index.js");
})({
    "./src/index.js": function (module, __webpack_exports__, __webpack_require__) {
        // 使用 eval 执行编译后的代码；
        // 继续递归引用模块内部依赖；
        // 实际情况并不是使用模板字符串，这里是为了代码的可读性；
        eval(`
            __webpack_require__.r(__webpack_exports__);
            //
            var _test__WEBPACK_IMPORTED_MODULE_0__ = __webpack_require__("test", ./src/test.js");
        `);
    },
    "./src/test.js": function (module, __webpack_exports__, __webpack_require__) {
        // ...
    },
})
```

- **总结**:
    - **模块机制**: webpack 自己实现了一套模拟模块的机制，将其包裹于业务代码的外部，从而提供了一套模块机制；
    - **文件编译**: webpack 规定了一套编译规则，通过 Loader 和 Plugin，以管道的形式对文件字符串进行处理；

2. Loader

由于 Webpack 是基于 Node，因此 Webpack 其实是只能识别 js 模块，比如 css / html / 图片等类型的文件并无法加载，因此就需要一个对 **不同格式文件转换器**。其实 Loader 做的事，也并不难理解: **对 Webpack 传入的字符串进行按需修改**。例如一个最简单的 Loader:

```
// html-loader/index.js
module.exports = function(htmlSource) {
    // 返回处理后的代码字符串
    // 删除 html 文件中的所有注释
    return htmlSource.replace(/<!--[\w\W]*?-->/g, '')
```

```
6. }
```

当然，实际的 Loader 不会这么简单，通常是需要将代码进行分析，构建 **AST (抽象语法树)**， 遍历进行定向的修改后，再重新生成新的代码字符串。如我们常用的 Babel-loader 会执行以下步骤:

- babylon 将 ES6/ES7 代码解析成 AST
- babel-traverse 对 AST 进行遍历转译，得到新的 AST
- 新 AST 通过 babel-generator 转换成 ES5

Loader 特性:

- **链式传递**，按照配置时相反的顺序链式执行；
- 基于 Node 环境，拥有 **较高权限**，比如文件的增删查改；
- 可同步也可异步；

常用 Loader:

- file-loader: 加载文件资源，如 字体 / 图片 等，具有移动/复制/命名等功能；
- url-loader: 通常用于加载图片，可以将小图片直接转换为 Date Url，减少请求；
- babel-loader: 加载 js / jsx 文件， 将 ES6 / ES7 代码转换成 ES5，抹平兼容性问题；
- ts-loader: 加载 ts / tsx 文件，编译 TypeScript；
- style-loader: 将 css 代码以<style>标签的形式插入到 html 中；
- css-loader: 分析@import和url()，引用 css 文件与对应的资源；
- postcss-loader: 用于 css 的兼容性处理，具有众多功能，例如 **添加前缀，单位转换** 等；
- less-loader / sass-loader: css预处理器，在 css 中新增了许多语法，提高了开发效率；

编写原则:

- **单一原则**: 每个 Loader 只做一件事；
- **链式调用**: Webpack 会按顺序链式调用每个 Loader；
- **统一原则**: 遵循 Webpack 制定的设计规则和结构，输入与输出均为字符串，各个 Loader 完全独立，即插即用；

3. Plugin

插件系统是 Webpack 成功的一个关键性因素。在编译的整个生命周期中，Webpack 会触发许多事件钩子，Plugin 可以监听这些事件，根据需求在相应的时间点对打包内容进行定向的修改。

- 一个最简单的 plugin 是这样的:

```
class Plugin{
        // 注册插件时，会调用 apply 方法
        // apply 方法接收 compiler 对象
        // 通过 compiler 上提供的 Api，可以对事件进行监听，执行相应的
操作
        apply(compiler){
                // compilation 是监听每次编译循环
                // 每次文件变化，都会生成新的 compilation 对象并触发该
事件
        compiler.plugin('compilation',function(compilation) {})
        }
}
```

- **注册插件**:

```
// webpack.config.js
module.export = {
        plugins:[
                new Plugin(options),
        ]
}
```

- **事件流机制**:

Webpack 就像工厂中的一条产品流水线。原材料经过 Loader 与 Plugin 的一道道处理，最后输出结果。

- 通过链式调用，按顺序串起一个个 Loader；
- 通过事件流机制，让 Plugin 可以插入到整个生产过程中的每个步骤中；

Webpack 事件流编程范式的核心是基础类 **Tapable**，是一种 **观察者模式** 的实现事件的订阅与广播：

```
const { SyncHook } = require("tapable")

const hook = new SyncHook(['arg'])

// 订阅
hook.tap('event', (arg) => {
    // 'event-hook'
    console.log(arg)
})

// 广播
hook.call('event-hook')
```

Webpack 中两个最重要的类 Compiler 与 Compilation 便是继承于 Tapable，也拥有这样的事件流机制。

- **Compiler**: 可以简单的理解为 **Webpack 实例**，它包含了当前 Webpack 中的所有配置信息，如 options， loaders, plugins 等信息，全局唯一，只在启动时完成初始化创建，随着生命周期逐一传递；
- **Compilation**: 可以称为 **编译实例**。当监听到文件发生改变时，Webpack 会创建一个新的 Compilation 对象，开始一次新的编译。它包含了当前的输入资源，输出资源，变化的文件等，同时通过它提供的 api，可以监听每次编译过程中触发的事件钩子；
- **区别**:
    - Compiler 全局唯一，且从启动生存到结束；
    - Compilation 对应每次编译，每轮编译循环均会重新创建；
- **常用 Plugin**:
    - UglifyJsPlugin: 压缩、混淆代码；
    - CommonsChunkPlugin: 代码分割；
    - ProvidePlugin: 自动加载模块；
    - html-webpack-plugin: 加载 html 文件，并引入 css / js 文件；
    - extract-text-webpack-plugin / mini-css-extract-plugin: 抽离样式，生成 css 文件；
    - DefinePlugin: 定义全局变量；
    - optimize-css-assets-webpack-plugin: CSS 代码去重；
    - webpack-bundle-analyzer: 代码分析；
    - compression-webpack-plugin: 使用 gzip 压缩 js 和 css；
    - happypack: 使用多进程，加速代码构建；
    - EnvironmentPlugin: 定义环境变量；

4. 编译优化

- **代码优化**:
    - **无用代码消除**，是许多编程语言都具有的优化手段，这个过程称为 DCE (dead code elimination)，即 **删除不可能执行的代码**；
        - 例如我们的 UglifyJs，它就会帮我们在生产环境中删除不可能被执行的代码，例如:

```
var fn = function() {
    return 1;
    // 下面代码便属于 不可能执行的代码;
```

```
    // 通过 UglifyJs (Webpack4+ 已内置) 便会进行 DCE;
    var a = 1;
    return a;
}
```

    - **摇树优化** (Tree-shaking)，这是一种形象比喻。我们把打包后的代码比喻成一棵树，这里其实表示的就是，通过工具 "摇" 我们打包后的 js 代码，将没有使用到的无用代码 "摇" 下来 (删除)。即消除那些被 **引用了但未被使用** 的模块代码。
        - **原理**: 由于是在编译时优化，因此最基本的前提就是语法的静态分析，**ES6的模块机制** 提供了这种可能性。不需要运行时，便可进行代码字面上的静态分析，确定相应的依赖关系。
        - **问题**: 具有 **副作用** 的函数无法被 tree-shaking。
            - 在引用一些第三方库，需要去观察其引入的代码量是不是符合预期;
            - 尽量写纯函数，减少函数的副作用;
            - 可使用 webpack-deep-scope-plugin，可以进行作用域分析，减少此类情况的发生，但仍需要注意;
- code-spliting: **代码分割** 技术，将代码分割成多份进行 **懒加载** 或 **异步加载**，避免打包成一份后导致体积过大，影响页面的首屏加载;
    - Webpack 中使用 SplitChunksPlugin 进行拆分;
    - 按 **页面** 拆分: 不同页面打包成不同的文件;
    - 按 **功能** 拆分:
        - 将类似于播放器，计算库等大模块进行拆分后再懒加载引入;
        - 提取复用的业务代码，减少冗余代码;
    - 按 **文件修改频率** 拆分: 将第三方库等不常修改的代码单独打包，而且不改变其文件 hash 值，能最大化运用浏览器的缓存;
- scope hoisting: **作用域提升**，将分散的模块划分到同一个作用域中，避免了代码的重复引入，有效减少打包后的代码体积和运行时的内存损耗;
- **编译性能优化**:
    - 升级至 **最新** 版本的 webpack，能有效提升编译性能;
    - 使用 **dev-server / 模块热替换 (HMR)** 提升开发体验;
        - 监听文件变动 **忽略 node_modules** 目录能有效提高监听时的编译效率;
    - **缩小编译范围**:
        - modules: 指定模块路径，减少递归搜索;
        - mainFields: 指定入口文件描述字段，减少搜索;

        - noParse: 避免对非模块化文件的加载；
        - includes/exclude: 指定搜索范围/排除不必要的搜索范围；
        - alias: 缓存目录，避免重复寻址；
    - babel-loader:
        - 忽略node_moudles，避免编译第三方库中已经被编译过的代码；
        - 使用cacheDirectory，可以缓存编译结果，避免多次重复编译；
    - **多进程并发**:
        - webpack-parallel-uglify-plugin: 可多进程并发压缩 js 文件，提高压缩速度；
        - HappyPack: 多进程并发文件的 Loader 解析；
    - **第三方库模块缓存**:
        - DLLPlugin 和 DLLReferencePlugin 可以提前进行打包并缓存，避免每次都重新编译；
    - **使用分析**:
        - Webpack Analyse / webpack-bundle-analyzer 对打包后的文件进行分析，寻找可优化的地方；
        - 配置profile：true，对各个编译阶段耗时进行监控，寻找耗时最多的地方；
    - source-map:
        - 开发: cheap-module-eval-source-map；
        - 生产: hidden-source-map；

项目性能优化

1. 编码优化

编码优化，指的就是 在代码编写时的，通过一些 **最佳实践**，提升代码的执行性能。通常这并不会带来非常大的收益，但这属于 **程序猿的自我修养**，而且这也是面试中经常被问到的一个方面，考察自我管理与细节的处理。

- **数据读取**:
    - 通过作用域链 / 原型链 读取变量或方法时，需要更多的耗时，且越长越慢；
    - 对象嵌套越深，读取值也越慢；
    - **最佳实践**:

        - 尽量在局部作用域中进行 **变量缓存**；
        - 避免嵌套过深的数据结构，**数据扁平化** 有利于数据的读取和维护；
- **循环**: 循环通常是编码性能的关键点；
    - 代码的性能问题会再循环中被指数倍放大；
    - **最佳实践**:
        - 尽可能 **减少循环次数**；
            - 减少遍历的数据量；
            - 完成目的后马上结束循环；
        - 避免在循环中执行大量的运算，避免重复计算，相同的执行结果应该使用缓存；
        - js 中使用 **倒序循环** 会略微提升性能；
        - 尽量避免使用 for-in 循环，因为它会枚举原型对象，耗时大于普通循环；
- **条件流程性能**: Map / Object > switch > if-else

```
// 使用 if-else
if(type === 1) {

} else if (type === 2) {

} else if (type === 3) {

}

// 使用 switch
switch (type) {
        case 1:
                break;4
        case 2:
                break;
        case 3:
                break;
    default:
        break;
}

// 使用 Map
const map = new Map([
```

```
        [1, () => {}],
        [2, () => {}],
        [3, () => {}],
])
map.get(type)()

// 使用 Objext
const obj = {
        1: () => {},
        2: () => {},
        3: () => {},
}
obj[type]()
```

- **减少 cookie 体积**: 能有效减少每次请求的体积和响应时间；
  - 去除不必要的 cookie；
  - 压缩 cookie 大小；
  - 设置 domain 与 过期时间；
- **dom 优化**:
  - **减少访问 dom 的次数**，如需多次，将 dom 缓存于变量中；
  - **减少重绘与回流**:
    - 多次操作合并为一次；
    - 减少对计算属性的访问；
      - 例如 offsetTop， getComputedStyle 等
      - 因为浏览器需要获取最新准确的值，因此必须立即进行重排，这样会破坏了浏览器的队列整合，尽量将值进行缓存使用；
    - 大量操作时，可将 dom 脱离文档流或者隐藏，待操作完成后再重新恢复；
    - 使用DocumentFragment / cloneNode / replaceChild进行操作；
  - 使用事件委托，避免大量的事件绑定；
- **css 优化**:
  - **层级扁平**，避免过于多层级的选择器嵌套；
  - **特定的选择器** 好过一层一层查找: .xxx-child-text{} 优于 .xxx .child .text{}
  - **减少使用通配符与属性选择器**；
  - **减少不必要的多余属性**；

    - 使用 **动画属性** 实现动画，动画时脱离文档流，开启硬件加速，优先使用 css 动画；
    - 使用 <link> 替代原生 @import；
- **html 优化**:
    - **减少 dom 数量**，避免不必要的节点或嵌套；
    - **避免空标签**，能减少服务器压力，因为 src 为空时，浏览器仍然会发起请求
        - IE 向页面所在的目录发送请求；
        - Safari、Chrome、Firefox 向页面本身发送请求；
        - Opera 不执行任何操作。
    - 图片提前 **指定宽高** 或者 **脱离文档流**，能有效减少因图片加载导致的页面回流；
    - **语义化标签** 有利于 SEO 与浏览器的解析时间；
    - 减少使用 table 进行布局，避免使用
与<hr />；

2. 页面基础优化

- **引入位置**: css 文件<head>中引入， js 文件<body>底部引入；
    - 影响首屏的，优先级很高的 js 也可以头部引入，甚至内联；
- **减少请求** (http 1.0 – 1.1)，合并请求，正确设置 http 缓存；
- **减少文件体积**:
    - **删除多余代码**:
        - tree-shaking
        - UglifyJs
        - code-spliting
    - **混淆 / 压缩代码**，开启 gzip 压缩；
    - **多份编译文件按条件引入**:
        - 针对现代浏览器直接给 ES6 文件，只针对低端浏览器引用编译后的 ES5 文件；
        - 可以利用<script type="module"> / <script type="module">进行条件引入用
    - **动态 polyfill**，只针对不支持的浏览器引入 polyfill；
- **图片优化**:
    - 根据业务场景，与UI探讨选择 **合适质量，合适尺寸**；

    - 根据需求和平台，选择 **合适格式**，例如非透明时可用 jpg；非苹果端，使用 webp；
    - 小图片合成 **雪碧图**，低于 5K 的图片可以转换成 **base64** 内嵌；
    - 合适场景下，使用 **iconfont** 或者 **svg**；
- **使用缓存**:
    - **浏览器缓存**: 通过设置请求的过期时间，合理运用浏览器缓存；
    - **CDN缓存**: 静态文件合理使用 CDN 缓存技术；
        - HTML 放于自己的服务器上；
        - 打包后的图片 / js / css 等资源上传到 CDN 上，文件带上 hash 值；
        - 由于浏览器对单个域名请求的限制，可以将资源放在多个不同域的 CDN 上，可以绕开该限制；
    - **服务器缓存**: 将不变的数据、页面缓存到 内存 或 远程存储(redis等) 上；
    - **数据缓存**: 通过各种存储将不常变的数据进行缓存，缩短数据的获取时间；

3. 首屏渲染优化

- **css / js 分割**，使首屏依赖的文件体积最小，内联首屏关键 css / js；
- 非关键性的文件尽可能的 **异步加载和懒加载**，避免阻塞首页渲染；
- 使用dns-prefetch / preconnect / prefetch / preload等浏览器提供的资源提示，加快文件传输；
- 谨慎控制好 **Web字体**，一个大字体包足够让你功亏一篑；
    - 控制字体包的加载时机；
    - 如果使用的字体有限，那尽可能只将使用的文字单独打包，能有效减少体积；
- 合理利用 Localstorage / server-worker 等存储方式进行 **数据与资源缓存**；
- **分清轻重缓急**:
    - 重要的元素优先渲染；
    - 视窗内的元素优先渲染；
- **服务端渲染(SSR)**:
    - 减少首屏需要的数据量，剔除冗余数据和请求；
    - 控制好缓存，对数据/页面进行合理的缓存；
    - 页面的请求使用流的形式进行传递；
- **优化用户感知**:
    - 利用一些动画 **过渡效果**，能有效减少用户对卡顿的感知；
    - 尽可能利用 **骨架屏(Placeholder) / Loading** 等减少用户对白屏的感知；

    - 动画帧数尽量保证在 **30帧** 以上，低帧数、卡顿的动画宁愿不要；
    - js 执行时间避免超过 **100ms**，超过的话就需要做:
        - 寻找可 缓存 的点；
        - 任务的 分割异步 或 web worker 执行；

全栈基础

其实我觉得并不能讲前端的天花板低，只是说前端是项更多元化的工作，它需要涉及的知识面很广。你能发现，从最开始的简单页面到现在，其实整个领域是在不断地往外拓张。在许多的大厂的面试中，具备一定程度的 **服务端知识、运维知识，甚至数学、图形学、设计** 等等，都可能是你占得先机的法宝。

Nginx

轻量级、高性能的 Web 服务器，在现今的大型应用、网站基本都离不开 Nginx，已经成为了一项必选的技术；其实可以把它理解成 **入口网关**，这里我举个例子可能更好理解:

当你去银行办理业务时，刚走进银行，需要到入门处的机器排队取号，然后按指令到对应的柜台办理业务，或者也有可能告诉你，今天不能排号了，回家吧！

这样一个场景中，**取号机器就是 Nginx(入口网关)**。一个个柜台就是我们的业务服务器(办理业务)；银行中的保险箱就是我们的数据库(存取数据)；🤣

- **特点**:
    - 轻量级，配置方便灵活，无侵入性；
    - 占用内存少，启动快，性能好；
    - 高并发，事件驱动，异步；
    - 热部署，修改配置热生效；
- **架构模型**:
    - 基于 **socket 与 Linux epoll (I/O 事件通知机制)**，实现了 **高并发**；
        - 使用模块化、事件通知、回调函数、计时器、轮询实现非阻塞的异步模式；
        - 磁盘不足的情况，可能会导致阻塞；
    - **Master-worker 进程模式**:
        - Nginx 启动时会在内存中常驻一个 **Master 主进程**，功能:

        - 读取配置文件；
        - 创建、绑定、关闭 socket；
        - 启动、维护、配置 worker 进程；
        - 编译脚本、打开日志；
    - master 进程会开启配置数量的 **worker 进程**，比如根据 CPU 核数等:
        - 利用 socket 监听连接，不会新开进程或线程，节约了创建与销毁进程的成本；
        - 检查网络、存储，把新连接加入到轮询队列中，异步处理；
        - 能有效利用 cpu 多核，并避免了线程切换和锁等待；
  - **热部署模式**:
    - 当我们修改配置热重启后，master 进程会以新的配置新创建 worker 进程，新连接会全部交给新进程处理；
    - 老的 worker 进程会在处理完之前的连接后被 kill 掉，逐步全替换成新配置的 worker 进程；

- **配置**:
  - 官网下载；
  - 配置文件路径： /usr/local/etc/nginx/nginx.conf；
  - 启动: 终端输入 nginx，访问 localhost:8080 就能看到 Welcome...；
  - nginx –s stop: 停止服务；
  - nginx –s reload: 热重启服务；
  - 配置代理: proxy_pass
    - 在配置文件中配置即可完成；
  - server {
  - listen 80;
  - location / {
  - proxy_pass http://xxx.xxx.xx.xx:3000;
  - }
  - }

- **常用场景**:
  - **代理**:

        - 其实 Nginx 可以算一层 **代理服务器**，将客户端的请求处理一层后，再转发到业务服务器，这里可以分成两种类型，其实实质就是 **请求的转发**，使用 Nginx 非常合适、高效；
    - **正向代理**:
        - 即用户通过访问这层正向代理服务器，再由代理服务器去到原始服务器请求内容后，再返回给用户；
        - 例如我们常使用的 VPN 就是一种常见的正向代理模式。通常我们无法直接访问谷歌服务器，但是通过访问一台国外的服务器，再由这台服务器去请求谷歌返回给用户，用户即可访问谷歌；
        - **特点**:
            - 代理服务器属于 **客户端层**，称之为正向代理；
            - 代理服务器是 **为用户服务**，对于用户是透明的，用户知道自己访问代理服务器；
            - 对内容服务器来说是 **隐藏** 的，内容服务器并无法分清访问是来自用户或者代理；
    - **反向代理**:
        - 用户访问头条的反向代理网关，通过网关的一层处理和调度后，再由网关将访问转发到内部的服务器上，返回内容给用户；
        - **特点**:
            - 代理服务器属于 **服务端层**，因此称为反向代理。通常代理服务器与内部内容服务器会隶属于同一内网或者集群；
            - 代理服务器是 **为内容服务器服务** 的，对用户是隐藏的，用户不清楚自己访问的具体是哪台内部服务器；
            - 能有效保证内部服务器的 **稳定与安全**；
    - 反向代理的好处:
        - 安全与权限:
            - 用户访问必须通过反向代理服务器，也就是便可以在做这层做统一的请求校验，过滤拦截不合法、危险的请求，从而就能更好的保证服务器的安全与稳定；
        - 负载均衡: 能有效分配流量，最大化集群的稳定性，保证用户的访问质量；
    - 负载均衡:
        - 负载均衡是基于反向代理下实现的一种 **流量分配** 功能，目的是为了达到服务器资源的充分利用，以及更快的访问响应；

- 其实很好理解，还是以上面银行的例子来看: **通过门口的取号器，系统就可以根据每个柜台的业务排队情况进行用户的分，使每个柜台都保持在一个比较高效的运作状态，避免出现分配不均的情况**；
- 由于用户并不知道内部服务器中的队列情况，而反向代理服务器是清楚的，因此通过 Nginx，便能很简单地实现流量的均衡分配；
- Nginx 实现: Upstream模块， 这样当用户访问 http://xxx 时，流量便会被按照一定的规则分配到upstream中的3台服务器上；

```
http {
    upstream xxx {
        server 1.1.1.1:3001;
        server 2.2.2.2:3001;
        server 3.3.3.3:3001;
    }
    server {
        listen 8080;
        location / {
            proxy_pass http://xxx;
        }
    }
}
```

- **分配策略**:
  - **服务器权重(weight)**:
    - 可以为每台服务器配置访问权重，传入参数 weight，例如:
  - upstream xxx {
  - server 1.1.1.1:3001 weight=1;
  - server 2.2.2.2:3001 weight=1;
  - server 3.3.3.3:3001 weight=8;
  - }
  - **时间顺序(默认)**: 按用户的访问的顺序逐一的分配到正常运行的服务器上；
  - **连接数优先(least_conn)**: 优先将访问分配到列表中连接数队列最短的服务器上；
  - **响应时间优先(fair)**: 优先将访问分配到列表中访问响应时间最短的服务器上；

        - **ip_hash**: 通过 ip_hash 指定，使每个 ip 用户都访问固定的服务器上，有利于用户特异性数据的缓存，例如本地 session 服务等；
        - **url_hash**: 通过 url_hash 指定，使每个 url 都分配到固定的服务器上，有利于缓存；
    - **Nginx 对于前端的作用**:
        - **1. 快速配置静态服务器**，当访问 localhost:80 时，就会默认访问到 /Users/files/index.html；

```
server {
        listen 80;
        server_name localhost;

        location / {
                root   /Users/files;
                index  index.html;
        }
}
```

        - **2. 访问限制**: 可以制定一系列的规则进行访问的控制，例如直接通过 ip 限制:

```
# 屏蔽 192.168.1.1 的访问；
# 允许 192.168.1.2 ~ 10 的访问；
location / {
    deny  192.168.1.1;
    allow 192.168.1.2/10;
    deny  all;
}
```

        - **3. 解决跨域**: 其实跨域是 **浏览器的安全策略**，这意味着只要不是通过浏览器，就可以绕开跨域的问题。所以只要通过在同域下启动一个 Nginx 服务，转发请求即可；

```
location ^~/api/ {
     # 重写请求并代理到对应域名下
    rewrite ^/api/(.*)$ /$1 break;
    proxy_pass https://www.cross-target.com/;
}
```

        - **4. 图片处理**: 通过 ngx_http_image_filter_module 这个模块，可以作为一层图片服务器的代理，在访问的时候 **对图片进行特定的操作，例如裁剪，旋转，压缩等**；
        - **5. 本地代理，绕过白名单限制**: 例如我们在接入一些第三方服务时经常会有一些域名白名单的限制，如果我们在本地通过localhost进行开发，便无法完成功能。这里我们可以做一层本地代理，便可以直接通过指定域名访问本地开发环境；

```
server {
listen 80;
server_name www.toutiao.com;

location / {
    proxy_pass http://localhost:3000;
}
}
```

Docker

Docker，是一款现在最流行的 **软件容器平台**，提供了软件运行时所依赖的环境。

- **物理机**:
    - 硬件环境，真实的 **计算机实体**，包含了例如物理内存，硬盘等等硬件；
- **虚拟机**:
    - 在物理机上 **模拟出一套硬件环境和操作系统**，应用软件可以运行于其中，并且毫无感知，是一套**隔离的完整环境**。本质上，它只是物理机上的一份**运行文件**。
- **为什么需要虚拟机?**
    - **环境配置与迁移**:
        - 在软件开发和运行中，环境依赖一直是一个很头疼的难题，比如你想运行 node 应用，那至少环境得安装 node 吧，而且不同版本，不同系统都会影响运行。**解决的办法**，就是我们的包装包中直接包含运行环境的安装，让同一份环境可以快速复制到任意一台物理机上。
    - **资源利用率与隔离**:
        - 通过硬件模拟，并包含一套完整的操作系统，应用可以独立运行在虚拟机中，与外界隔离。并且可以在同一台物理机上，开启多个不

同的虚拟机启动服务，即一台服务器，提供多套服务，且资源完全相互隔离，互不影响。不仅能更好提高资源利用率率，降低成本，而且也有利于服务的稳定性。

- **传统虚拟机的缺点**:
    - **资源占用大**:
        - 由于虚拟机是模拟出一套 **完整系统**，包含众多系统级别的文件和库，运行也需要占用一部分资源，单单启动一个空的虚拟机，可能就要占用 100+MB 的内存了。
    - **启动缓慢**:
        - 同样是由于完整系统，在启动过程中就需要运行各种系统应用和步骤，也就是跟我们平时启动电脑一样的耗时。
    - **冗余步骤多**:
        - 系统有许多内置的系统操作，例如用户登录，系统检查等等，有些场景其实我们要的只是一个隔离的环境，其实也就是说，虚拟机对部分需求痛点来说，其实是有点过重的。
- Linux 容器:
    - Linux 中的一项虚拟化技术，称为 Linux 容器技术(LXC)。
    - 它在 **进程层面** 模拟出一套隔离的环境配置，但并没有模拟硬件和完整的操作系统。因此它完全规避了传统虚拟机的缺点，在启动速度，资源利用上远远优于虚拟机；
- Docker:
    - Docker 就是基于 Linux 容器的一种上层封装，提供了更为简单易用的 API 用于操作 Docker，属于一种 **容器解决方案**。
    - **基本概念**: 在 Docker 中，有三个核心的概念:
        - **镜像 (Image)**:
            - 从原理上说，镜像属于一种 **root 文件系统**，包含了一些系统文件和环境配置等，可以将其理解成一套 **最小操作系统**。为了让镜像轻量化和可移植，Docker 采用了 **Union FS 的分层存储模式**。将文件系统分成一层一层的结构，逐步从底层往上层构建，每层文件都可以进行继承和定制。这里从前端的角度来理解: **镜像就类似于代码中的 class，可以通过继承与上层封装进行复用**。
            - 从外层系统看来，一个镜像就是一个 Image **二进制文件**，可以任意迁移，删除，添加；
    - **容器 (Container)**:
        - 镜像是一份静态文件系统，无法进行运行时操作，就如class，如果我们不进行实例化时，便无法进行操作和使用。因此 **容器可以理解成镜像的实例**，即 new 镜像()，这样我们便可以创建、修改、操作容器；一旦创建后，就可以简单理解成一个轻量级的操作

系统，可以在内部进行各种操作，例如运行 node 应用，拉取 git 等；

- 基于镜像的分层结构，容器是 **以镜像为基础底层**，在上面封装了一层 **容器的存储层**；
  - 存储空间的生命周期与容器一致；
  - 该层存储层会随着容器的销毁而销毁；
  - 尽量避免往容器层写入数据；
- 容器中的数据的持久化管理主要由两种方式:
  - **数据卷 (Volume)**: 一种可以在多个容器间共享的特殊目录，其处于容器外层，并不会随着容器销毁而删除；
  - **挂载主机目录**: 直接将一个主机目录挂载到容器中进行写入；

- **仓库 (Repository)**:
  - 为了便于镜像的使用，Docker 提供了类似于 git 的仓库机制，在仓库中包含着各种各样版本的镜像。官方服务是 Docker Hub；
  - 可以快速地从仓库中拉取各种类型的镜像，也可以基于某些镜像进行自定义，甚至发布到仓库供社区使用；

结语

不知不觉，一个月又过去了，也终于完成了整个系列。其实下篇涉及的许多知识点都是有比较深的拓展空间，博主自己也水平有限，无法面面俱到，也许甚至会有些争议或者错误的见解，还望小伙伴们共同指出和纠正。希望这个面试系列能帮助到大家，好好地将这些知识点进行消化和理解，闭关修炼虽然辛苦，但现在已经是时候出山征战江湖，收割 Offer 啦~

整个系列其实仍然是属于浅尝辄止的阶段，后续如果大家想要继续提升，可以往自己感兴趣的方向进行深挖，例如:

- **全栈**: 那可能得更多的去了解 Node / Nginx / 反向代理 / 负载均衡 / PM2 / Docker 等服务端或者运维知识；
- **跨平台**: 可以学习 Hybrid / Flutter / React Native / Swift 等；
- **视觉游戏**: WebGL / 动画 / Three.js / Canvas / 游戏引擎 / VR / AR 等；
- **底层框架**: 浏览器引擎 / 框架底层 / 机器学习 / 算法等；

整理取材博主：
FE字节跳动 郭东东 面经
张鑫旭 css世界
灵魂画师牧码 画解算法