黑马程序员

传智播客
www.itcast.cn

面试宝典

就业部--前端小组

此次版本升级，是集合了上海校区前端学科开班以来就业学员提供的面试题，由就业部联合教研部整理出版。

HTML CSS

怎么让一个 div 水平垂直居中

```
<div class="parent">
  <div class="child"></div>
</div>
```

```
div.parent {
    display: flex;
    justify-content: center;
    align-items: center;
}
```

```
div.parent {
    position: relative;
}
div.child {
    position: absolute;
    top: 50%;
    left: 50%;
    transform: translate(-50%, -50%);
}
```

```
div.parent {
    display: grid;
}
div.child {
    justify-self: center;
    align-self: center;
}
```

```
div.parent {
```

```
    font-size: 0;
    text-align: center;
    &::before {
        content: "";
        display: inline-block;
        width: 0;
        height: 100%;
        vertical-align: middle;
    }
}
div.child{
  display: inline-block;
  vertical-align: middle;
}
```

已知如下代码，如何修改才能让图片宽度为 300px？ 注意下面代码不可修改

```
<img src="1.jpg" style="width:480px!important;”>
```

```
max-width: 300px
<!--或者-->
transform: scale(0.625,0.625)
```

负边距

有二个并排的盒子 给他们加 负边距 左边的和盒子给他负边距是左移，给右边的盒子负边距是左拉

shape-outside

shape-outside的CSS 属性定义了一个可以是非矩形的形状，相邻的内联内容应围绕该形状进行包装
但是你给他加背景颜色 你会发现他的可视大小还是矩形

flex布局下margin

在flex下的标签 加上margin的时候 标签元素会自适应

input的宽度

并不是给元素设置display:block就会自动填充父元素宽度。input 就是个例外，其默认宽度取决于size特性的值

定位特性

绝对定位和固定定位时，同时设置 left 和 right 等同于隐式地设置宽度

层叠上下文

在定位元素中，父盒子的z-index 小于兄弟元素的z-index时 ， 父盒子中的子元素 z-index在高也不会覆盖 父元素的兄弟元素

多个盒子等宽 居中

```
<main style="displat:table;width:100%;height:200px;boder-spacing:30px 0
">
    <div style="height:100%;"></div>
    <div></div>
    <div></div>
    <div></div>
</main>
```

定宽高比

css实现定宽高比的原理：padding的百分比是相对于其包含块的宽度，而不是高度

隐藏文本

text-indent:-2000px
font-size:0px

角向渐变

```
<div style="width:200px;height:200px;border-radius:50%;background:conic-gradient(red 0 33.3%, green 33.3% 66.6% , blue 66.6% 100%);"></div>
```

object-fit

图片在指定尺寸后，可以设置object-fit为contain或cover保持比例 (宽 和 高)

虚化

filter:blur(2)

自定义属性

使用c3 的自定义属性

```
div{
    --name: 18px;
    font-size: var(--name)
}
```

min-content/max-content

width:min-content
可以设置宽度为min-content和max-content，前者让内容尽可能地收缩，后者让内容尽可能地展开

resize

普通元素也可以像textarea那样resize

```
div{
overflow:hidden;
resize:auto
}
```

面包屑

```
<ul>
<li>首页</li>
<li>商品页</li>
<li>详情页</li>
</ul>

<style>
    ul{
        display:flex
    }
    li:before{
        content:"\27a5"
    }
    li:fish-child:before{
        content:""
    }
</style>
```

sticky footer

使用grid布局实现sticky footer

```
    <div class="box">
        <div>header</div>
        <div>body</div>
        <div>footer</div>
    </div>

    <style>
        .box{
            display:grid;
            min-height: 100%;
            grid-template-rows: auto 1fr auto;
        }
    </style>
```

Javascript 面试题

javascript的原始类型有哪些。

- boolean
- null
- undefined
- number
- string
- symbol

原始类型存储的都是值，是没有函数可以调用的
给一些原始数据类型调用方法的时候js给这些原始数据类型隐式转换了，所以有一些原始数据类型可以使用方法，但是他在使用这些方法的时候已经不是原始类型了。
所有的原始数据类型都是不可更改的

typeof 和 instanceof 的区别

typeof 可以用来判断原始数据的基本类型
其中除了null，其他的原始类型都可以正确的判断出来
但是typeof在判断对象的时候除了函数可以正确的判断以外，其他的数据类型都是无法正确的判断的，返回结果都是object
instanceof 是通过原型链的方式来判断数据类型的
所以instanceof可以正确的判断对象数据类型，但是缺点就是没有办法准确的判断出原始数据类型。

js中哪些数据是false

undefined， null， false， NaN， ''， 0， -0

js中对象怎么转换成原始数据类型

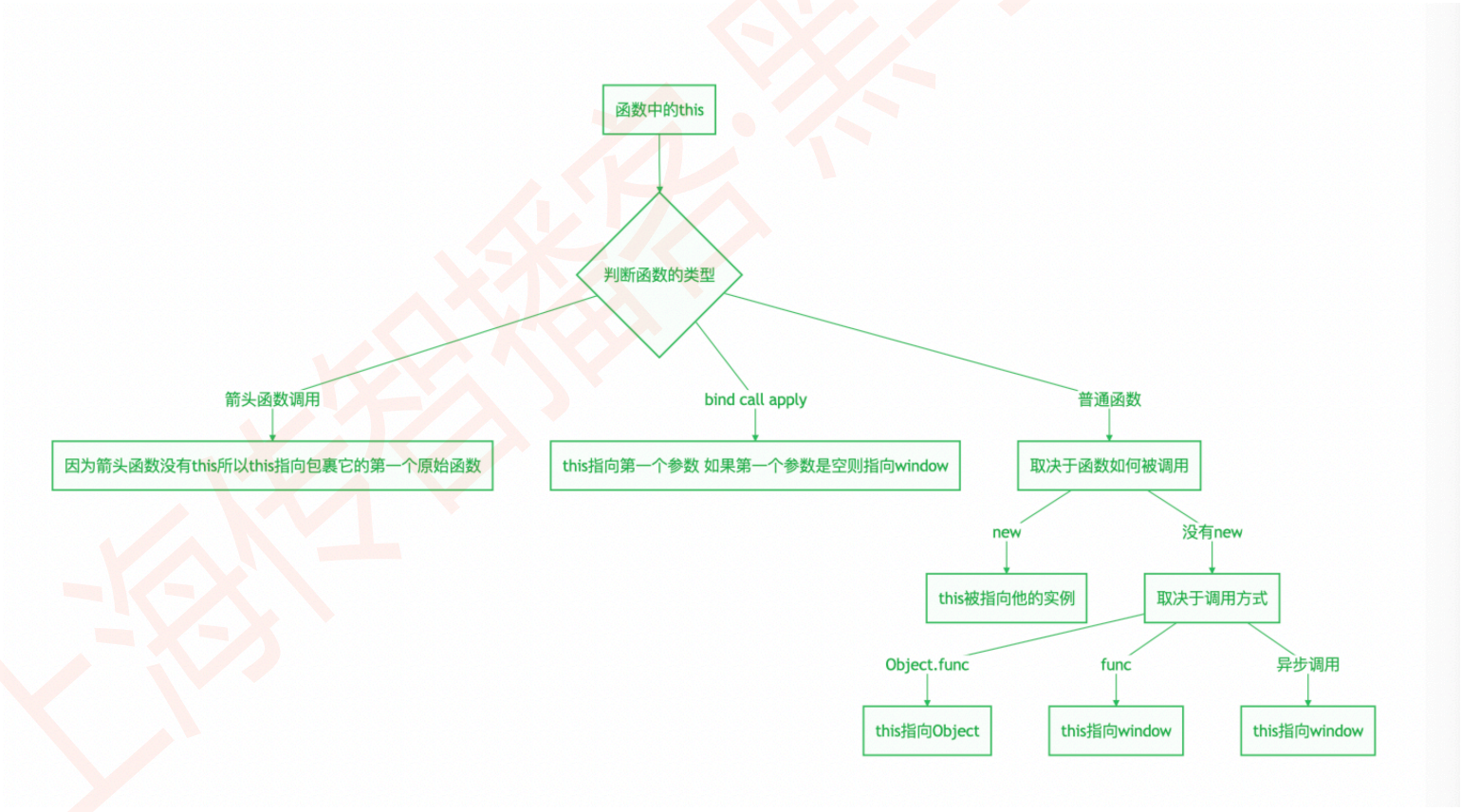
在js中对象在转换成原始数据类型的时候 是通过[[ToPrimitive]]函数来进行转换的

1. 如果已经是原始类型了，那就不需要转换了
2. 调用 x.valueOf()，如果转换为基础类型，就返回转换的值
3. 调用 x.toString()，如果转换为基础类型，就返回转换的值
4. 如果都没有返回原始类型，就会报错

我们也可以使用ES6的新特性Symbol来写一个假的ToPrimitive

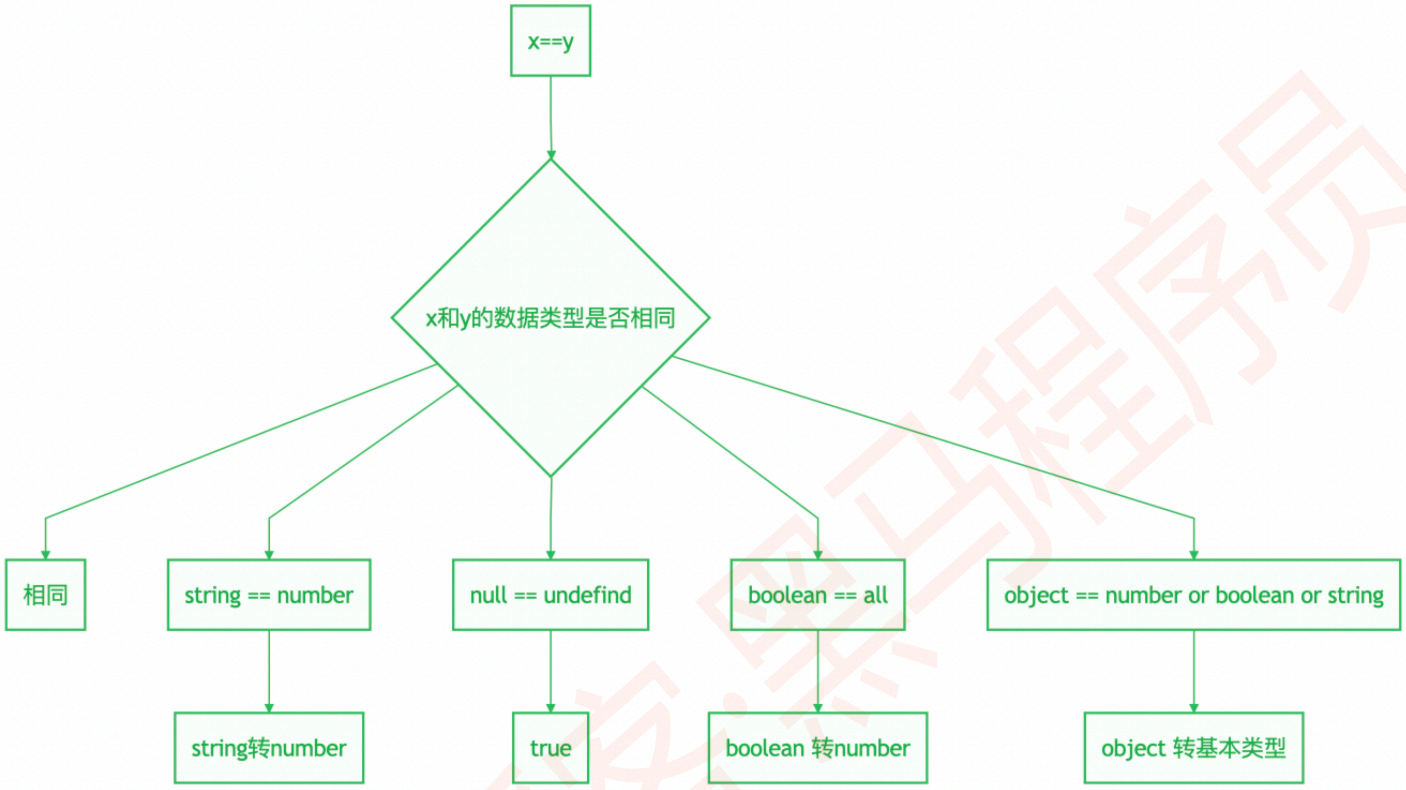
```
let obj = {
    myToString(){return '1'},
    myValueOf(){return 0},
    //对象的Symbol.toPrimitive属性，指向一个方法。
    //该对象被转为原始类型的值时，会调用这个方法，返回该对象对应的原始类型值。
    [Symbol.toPrimitive](){return 'symbol'}
}
obj + '' // 'symbol'
```

this



=== 和 == 的区别

== 在使用==进行比较的时候如果双方类型不一样就会进行类型转换
=== 就是判断二者的数据和类型是否相同



对象转原始类型 结果'[object Object]';
数组转原始类型 结果'';
相同复杂数据类型比较的是地址；

闭包

函数 A 内部有一个函数 B，函数 B 可以访问到函数 A 中的变量，那么函数 B 就是闭包。

深拷贝和浅拷贝

- 浅拷贝

  Object.assign 方法 会拷贝所有的属性值到新的对象中，如果属性值是对象的话，拷贝的是地址。
  使用es6的 ... 展开符

- 深拷贝

  JSON.parse(JSON.stringify(object))
  通常使用这个方法可以解决大部分问题但是它也有局限性

- 会忽略 undefined
- 会忽略 symbol
- 不能序列化函数
- 不能解决循环引用的对象

使用封装函数进行深拷贝 递归

```
function deepClone(obj){
    let objClone = Array.isArray(obj)?[]:{};
    if(obj && typeof obj==="object"){
        for(key in obj){
            if(obj.hasOwnProperty(key)){
                //判断ojb子元素是否为对象，如果是，递归复制
                if(obj[key]&&typeof obj[key] ==="object"){
                    objClone[key] = deepClone(obj[key]);
                }else{
                    //如果不是，简单复制
                    objClone[key] = obj[key];
                }
            }
        }
    }
    return objClone;
}
```

原型

每一个函数创建的时候都会有一个自带的属性，这个属性指向一个对象，这个对象就是原型。
每一个构造函数所创建出来的实例都可以调用这个原型里面的属性和方法。
原型本身也是一个对象，这个对象就是Object的实例，所以原型也可以调用到Object的原型，这样就组成了一个链式结构，这个就是原型链。

继承

class 继承

```
class Father{
 constructor(val) {
   this.val = val
 }
 my() {
   console.log(this.val)
 }
}
// class 中使用 extends来继承父类
class Son extends Father{
  constructor(val) {
  // super 类似于 Father.call(this , val)
   super(val)
   this.val = val
 }
}
```

AMD和CMD

插件加载方式
AMD 就是一上来就将我需要的插件下载下来
CMD 就是当我用到那个插件就下载那个插件

map, filter, reduce， find， forEach

- forEach 只是简单的遍历数组。

```
[1,2,3].forEach( num => console.log(num) )
```

- map 遍历原数组，生成新的数组，将数组的每一个元素拿出来处理完成之后在放回去

```
[1,2,3].map(num => num > 10 ? num : '0' + num)
```

- filter 遍历数组，生成新的数组，判断每一个值得返回结果是否为true，是true的放进去

```
[1,2,3].filter( num => num > 1 ) // [2 , 3]
```

- reduce主要是为了对所有数组进行累加，最后返回一个值，不改变原数组

```
[1,2,3].reduce( (add , val ,index , arr) => add + val , 0 )
```

- find 遍历数组找到第一个与之匹配的值

```
 [1,2,3].find( num => num > 1)
```

并发 && 并行

- 并发 同时分发出去多条任务，但是始终都是只有一条任务在执行
- 并行 同时分发出去多条任务，所有的任务同时执行

  从表面上来开，似乎并行比并发要好，但是并行是有限的，他的并行数量是根据硬件来的，有封顶，一旦达到封顶就会出现堵塞
  并发就不会出现这样的情况，如果有多个任务来了，并发并不会因为任务的数量而导致堵塞
  但是并发处理任务的效率是没有不能和并发比较的。

定时器 setTimeout、setInterval、requestAnimationFrame

- setTimeout 延迟定时器 设置一个时间，多少秒之后执行
- setInterval 间隙性定时器 设置一个时间 ， 没过一定的时间执行
- requestAnimationFrame 帧 页面每一帧调用一次，没有固定的时间。速度根据浏览器的性能来定的，比setTimeout和setInterval更加好。requestAnimationFrame大致每1000ms/60 （16.6ms）执行

  js是异步执行代码的，所以在执行过程中如果遇到消耗性能的代码就会出现定时器不准确的情况。
  所以可以使用requestAnimationFrame来执行某些需要异步执行的代码。
  我们可以使用requestAnimationFrame封装一个定时器

```
    let mySetInterval = (callback , userInterval) => {

        const Time = Date.now;
```

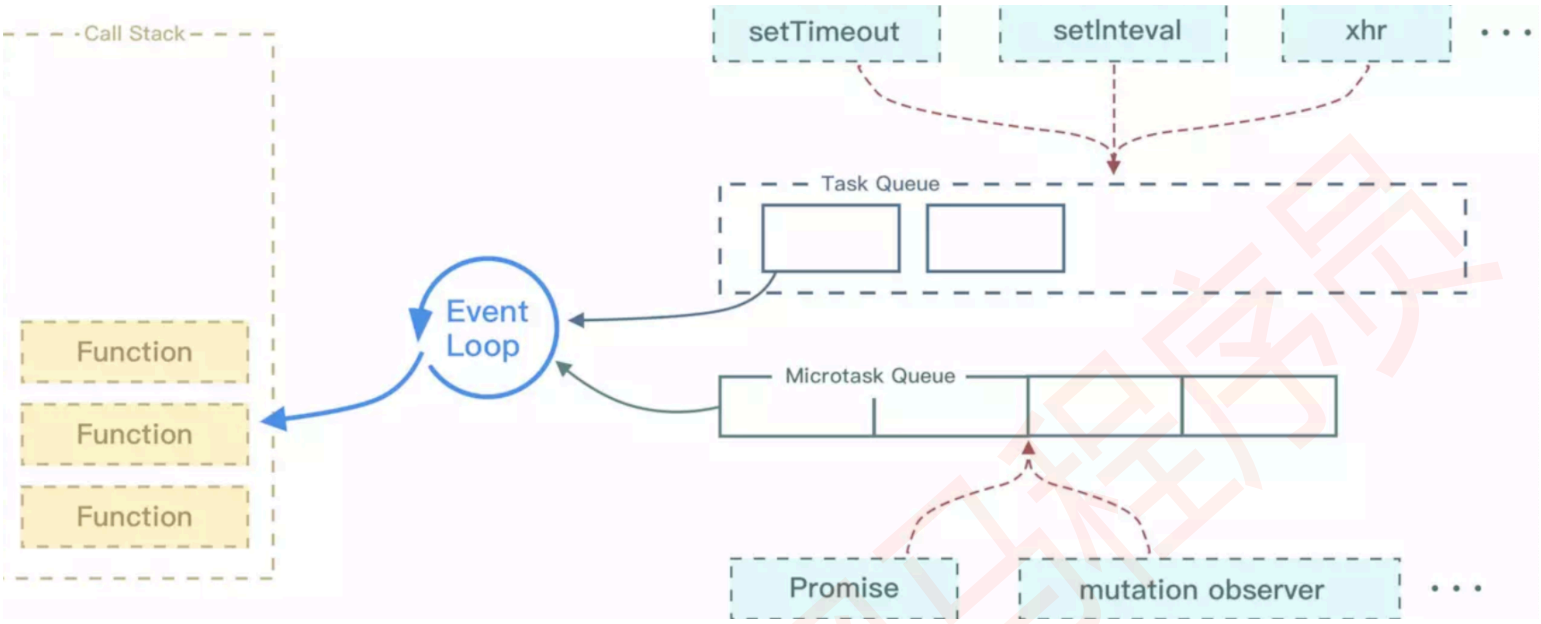
```
        // 获取当前时间戳
        let startTime = Time();
        // 开始时间戳
        let currentTime
        // 创建一个回调函数
        let myLoop = () => {
            currentTime = Time();
            let index = window.requestAnimationFrame(myLoop);
            // 判断当前时间
            if( currentTime -  startTime >= userInterval){
               startTime = currentTime = Time();
               // 调用callback
               callback.call(null,index);
            }


        }
       return window.requestAnimationFrame(myLoop)
    }
```

执行栈

CallStack 是JavaScript中执行代码的一种方式
JavaScript在执行代码的时候会将要执行的函数放到一个栈结构中，同时遵循先进后出的原则。
在开始执行代码的时候，JavaScript会先执行main函数然后在执行我们的代码，并且遵循先进后出的原则（函数）。

Event Loop

当我们执行js代码的时候，其实就是将我们的代码放到执行栈中执行，但是遇到异步代码的时候就会以另外一种方式进行。



JavaScript在代码运行到异步代码的时候会根据不同的异步，将代码放到对应的Task里面去
JavaScript一旦执行完执行栈里面的代码之后，Event Loop就会从task队列中拿出需要执行的代码并放入到执行栈中执行代码。
其中Task任务分为二种 microtask（微任务），macrotask（宏任务）
在 ES6 规范中，microtask 称为 jobs，macrotask 称为 task

- microtask（微任务）process.nextTick、Promise、MutationObserver 等
- macrotask（宏任务）setTimeout、setInterval、 setImmediate、script（整体代码）、I/O 操作等。

Event Loop 过程解析

```
 <script>
function func(num) {
  return function () {
    console.log(num)
  };
}
setTimeout(func(1));


async function async3() {
  await async4();
  console.log(8)
}
```

```
async function async4() {
  console.log(5)
}

async3();

function func2() {
  console.log(2);

  async function async1() {
    await async2();
    console.log(9)
  }

  async function async2() {
    console.log(5)
  }

  async1();

  setTimeout(func(4))
}


setTimeout(func2);

setTimeout(func(3));

new Promise(resolve => {
  console.log('Promise');
  resolve()
})
  .then(() => console.log(6))
  .then(() => console.log(7));

console.log(0);
 </script>
```

宏任务是一个一个的执行代码的；
微任务是将所有的任务执行完成之后在执行后面的代码的，
哪怕是在微任务阶段添加的微任务也会在当前阶段执行。

1. 首先第一步将script标签放到宏任务队列中，之后将script标签放入执行栈中

宏任务macrotask	微任务microtask	执行栈
script标签		script标签

1. 执行script标签里面的代码，逐行解析代码。并且将异步代码放到对应的Task中

宏任务macrotask	微任务microtask	执行栈
第一个 setTimeout	.then(6)	async4()
第二个 setTimeout		new Promise
第三个 setTimeout		console.log(0)

- 首先第一步遇到第一个 setTimeout，将定时器放到宏任务中；
- 第二步遇到async3() 。执行async3代码，然后调用async4(),运行里面的代码输出5，因为async3使用了async await 所以输出完成5之后await async4()会返回一个Promise，并且后面的代码会在Promise中执行。

```
async function async3() {
  await async4();
  console.log( 8 )
}

async function async4() {
  console.log( 5 )
}
async3();
// 转换
new Promise( resolve =>{
    console.log(5)
    resolve()
} ).then( () => {
```

```
    console.log( 8 )
} )
// 上述代码转换之后就是这样 ， 并且会在当前微任务结束之后在放入微任务队列
// new Promise里面的代码是同步执行的，如果遇到异步在调用Event Loop
```

1. 下一步开始执行微任务，首先执行async3()里面返回的Promise.then()打印8。然后执行下一个微任务6

   输出6之后返回一个新的Promise对象，这时候微任务里面又多了一个任务，这个时候会先执行微任务，不会执行宏任务。要等到所有的微任务全部执行完成之后才会执行宏任务。

宏任务macrotask	微任务microtask	执行栈
第一个 setTimeout	.then(7)	console.log(6)
第二个 setTimeout		
第三个 setTimeout		

微任务执行完成之后await放入微任务队列

宏任务macrotask	微任务microtask	执行栈
第一个 setTimeout	.then(8)	console.log(7)
第二个 setTimeout		
第三个 setTimeout		
宏任务macrotask	**微任务microtask**	**执行栈**
第一个 setTimeout		console.log(8)
第二个 setTimeout		
第三个 setTimeout		

1. 下一步微任务已经都执行完成了，所以开始执行宏任务，并且在调用async1()方法的时候又创建了一个宏任务和一个await

宏任务macrotask	微任务microtask	执行栈
打印4的setTimeout		func(1)
		func2()

		async1()
		func(3)

微任务执行完成之后放入await;

宏任务macrotask	微任务microtask	执行栈
打印4的setTimeout	await async2()的的Promise.then()	

执行宏任务阶段，会将宏任务里面的代码推到执行栈里面执行

1. 执行完成之后开始执行微任务

宏任务macrotask	微任务microtask	执行栈
打印4的setTimeout		console.log(9)

1. 微任务执行完成之后执行宏任务

宏任务macrotask	微任务microtask	执行栈
		console.log(4)

Node 中的 Event Loop 和浏览器中的是完全不相同的东西。

宏任务， 微任务队列 和 promise

```
setTimeout(()=>{
    console.log(1)
},0)
Promise.resolve().then(()=>{
    console.log(2)
})
console.log(3)
```

main script运行结束后(加载完成script标签之后会开始执行同步代码，main指script标签加载完成之后随便执行同步代码)，会有微任务队列和宏任务队列。然后根据main script中产生的微任务队列和宏任务队列，分别清空，这个时候是先清空微任务的队列，再去清空宏任务的队列。
微任务先执行，之后是宏任务。

```
setTimeout(()=>{
   console.log(1)
},0)
let a=new Promise((resolve)=>{
    console.log(2)
    resolve()
}).then(()=>{
   console.log(3)
}).then(()=>{
   console.log(4)
})
console.log(4)
```

这个要从Promise的实现来说，Promise的executor是一个同步函数，即非异步，立即执行的一个函数，因此他应该是和当前的任务一起执行的。而Promise的链式调用then，每次都会在内部生成一个新的Promise，然后执行then，在执行的过程中不断向微任务(microtask)推入新的函数，因此直至微任务(microtask)的队列清空后才会执行下一波的macrotask。

```
new Promise((resolve,reject)=>{
    console.log("promise1")
    resolve()
}).then(()=>{
    console.log("then11")
    new Promise((resolve,reject)=>{
        console.log("promise2")
        resolve()
    }).then(()=>{
        console.log("then21")
    }).then(()=>{
        console.log("then23")
    })
}).then(()=>{
    console.log("then12")
})
```

宏任务 微任务 二

```
async function async1() {
    console.log("async1 start");
    await  async2();
    console.log("async1 end");
}
async  function async2() {
    console.log( 'async2');
}
console.log("script start");
setTimeout(function () {
    console.log("settimeout");
});
async1()
new Promise(function (resolve) {
    console.log("promise1");
    resolve();
}).then(function () {
    console.log("promise2");
});
setImmediate(()=>{
    console.log("setImmediate")
})
process.nextTick(()=>{
    console.log("process")
})
console.log('script end');
```

队列执行start
第一轮:

current task: "script start", "async1 start", 'async2', "promise1", “script end”
micro task queue: [async,promise.then,process]
macro task queue: [setTimeout,setImmediate]

```
第二轮

current task: process, async1 end ,promise.then
micro task queue: []
macro task queue: [setTimeout,setImmediate]


第三轮

current task: setTimeout,setImmediate
micro task queue: []
macro task queue: []

最终结果: [script start, async1 start, async2, promise1, script end,proces
s,async1 end,promise2,setTimeout,setImmediate]
同样"async1 end","promise2"之间的优先级，因平台而异。
```

实现call、apply 及 bind 函数

代码实现

```
Function.prototype.myCall = function ( _this = window , ...val ) {

    _this._this = this;
    const result = _this._this(...val);
    delete _this._this;
    return result;
}

Function.prototype.myApply = function(_this = window , val){

    _this._this = this;
    const result = _this._this(...val);
    delete _this._this;
    return result
}
```

```
Function.prototype.myBind = function(_this = window , ...val){

    _this._this = this;
    return function(){
        const result = _this._this(...val);
        delete _this._this;
        return result;
    }
}
```

new

调用new干了什么?

- 创建一个空对象，并且 this 变量引用该对象
- 继承了函数的原型
- 属性和方法被加入到 this 引用的对象中。并执行了该函数
- 新创建的对象由 this 所引用，并且最后隐式的返回 this（如果返回的结果是对象或者函数就返回结果）

instanceof 的原理

instanceof主要的作用是判断对象的正确类型。
他的内部原理是通过判断原型链中能不能找到 类型的原型 Object，Array

```
    // 模仿indtsnceof实现的代码
    function myInstanceof(leftA , style){
        leftA = leftA.__proto__;
        if(leftA === null || !style) return false;
        else if(style.name === leftA.constructor.name) return true;
        return myInstanceof(leftA , style);
    }
```

addEventListener

addEventListener 方法将监听器注册到指定的标签上，当该标签触发指定事件的时候，指定的回调函数就会被执行
addEventListener 中可以传递三个参数 target.addEventListener(type, listener[, options]);

- type 表示监听事件类型的字符串。
- listener 当所监听的事件类型触发时，回调的一个函数。
- options 传入一个对象或者布尔
  布尔 capture 指定事件是否在捕获或冒泡阶段执行
  对象
- 布尔 capture 指定事件是否在捕获或冒泡阶段执行
- 布尔 once 表示当前事件最多只调用一次
- 布尔 passive 表示 listener 永远不会调用 preventDefault()。如果 listener 仍然调用了这个函数，客户端将会忽略它并抛出一个控制台警告。

为什么使用Object.create(null)初始化对象

使用Object.create(null)创建的对象是一个没有继承关系的对象，并且原型上的所有方法和属性全部被抛弃。

js处理树状数据

```
var list = [{
            id: 1,
            name: '超级管理',
            parent_id: 0
        },
        {
            id: 2,
            name: '用户管理',
            parent_id: 1
        },
        {
            id: 3,
            name: '部门管理',
            parent_id: 1
        },
        {
            id: 4,
```

```
            name: '日志管理',
            parent_id: 1
        },{
            id: 5,
            name: '操作用户',
            parent_id: 2
        },
        {
            id: 6,
            name: '查看用户',
            parent_id: 2
        },
        {
            id: 7,
            name: '用户新增',
            parent_id: 5
        },
        {
            id: 8,
            name: '用户删除',
            parent_id: 5
        }]
        /**
         * 树状的算法
         * @params list     代转化数组
         * @params parentId 起始节点
         */
        getTrees(list, 0)
        function getTrees(list, parentId) {
            let items = {};
            // 获取每个节点的直属子节点，*记住是直属，不是所有子节点
            for(let i = 0; i < list.length; i++) {
                let key = list[i].parent_id;
                if(items[key]) {
                    items[key].push(list[i]);/Users/xiaofeifei/Download
s/面试题/面试题（题目/11-数组,链表,二叉树.md
                } else {
                    items[key] = [];
```

```
                items[key].push(list[i]);
            }
        }
        return formatTree(items, parentId);
    }

    /**
     * 利用递归格式化每个节点
     */
    function formatTree(items, parentId) {
        let result = [];
        if(!items[parentId]) {
            return result;
        }
        for(let t of items[parentId]) {
            t.children = formatTree(items, t.id)
            result.push(t);
        }
        console.log(result)
        return result;
    }
```

浏览器存储

特性	周期	大小	方式
cookie	时间前后台都可以设置	4k	每一次请求都会自动携带在请求里面
localStorage	除非清理掉，否则永久在	5m	需要手动设置到请求体内
sessionStorage	页面关闭就清理	5m	需要手动设置到请求体中
indexDB	除非清理掉，否则永久在	根据本地硬件大小 基本就是用不完	需要手动设置到请求体中

作用
携带token(有销毁时间的)

一些用户的习惯类的信息，token(有销毁时间的)，css，js
当前页面的一些信息，需要暂时缓存下来
页面上的一些数据，配置，提示

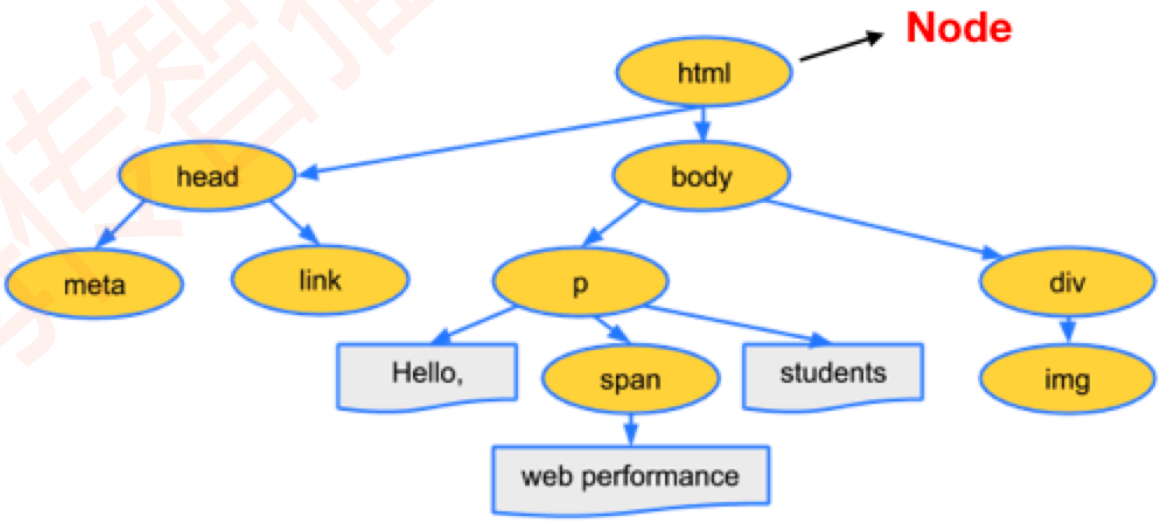
浏览器渲染机制

浏览器请求到HTML和CSS之后是怎么对他们进行处理的。

- 当浏览器拿到html之后，首先它是不认识这些东西的，所以它回家这些html字符串会通过词法分析，将这些字符串转化成标记(Token);

```
<span> 你好中国 </span>
<!--- 其中<span>是标记的开始 -->
<!--- 你好中国 是标记的结束 -->
<!--- </span> 是标记的结束 -->
```

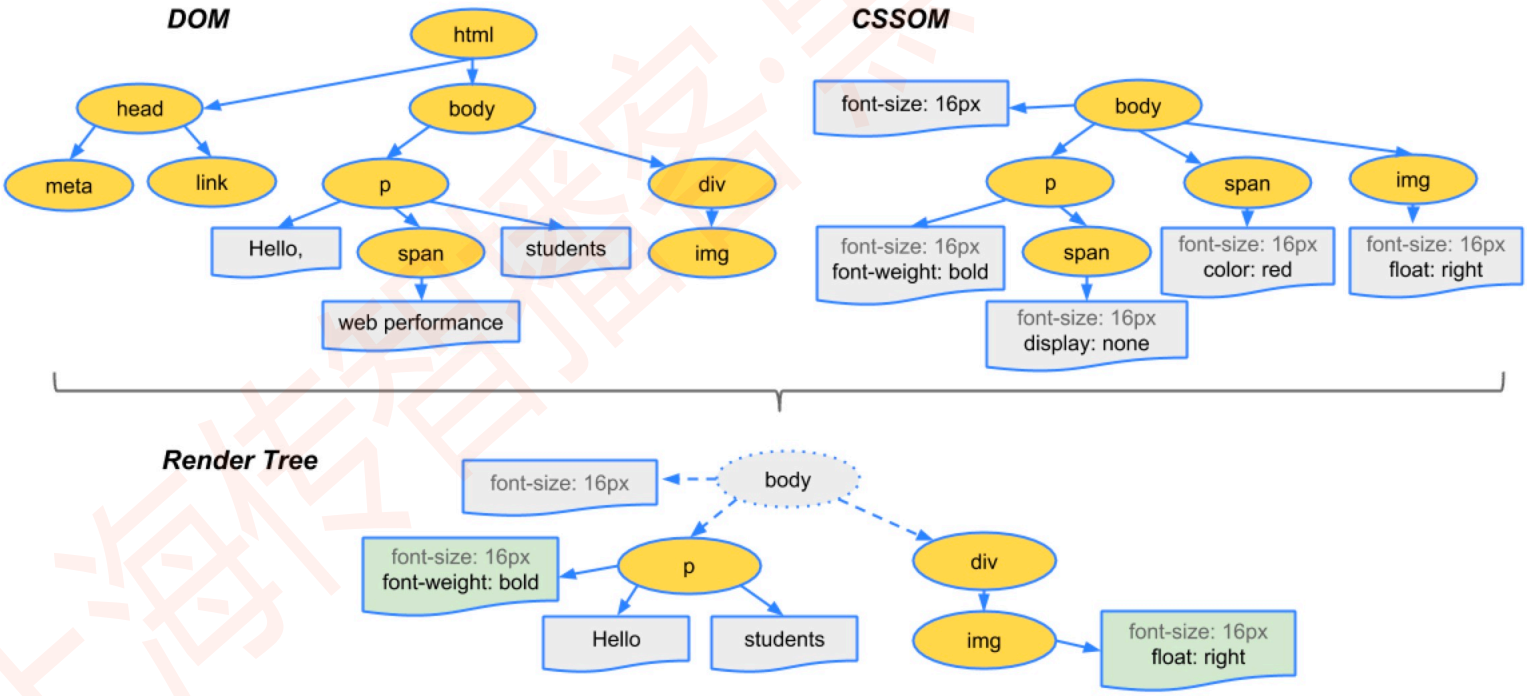
- 下一步就是将生成的标记转化成节点(Node),最后将这些节点根据不同的关系构建成一棵DOM树



- 当遇到css 的时候会开始绘制样式树，绘制样式树的时候和结构树差不都，都是先将字符串转化成标记在转化成节点，然后构建CSSOM树(DOM树也叫结构树，CSSOM树也叫样式树)

```
<style>
*{font-size:14px;}
```

```
/* 通配符添加样式的时候会遍历页面上所有的标签，并且添加样式。及其消耗性能 */
div{font-size:16px;}
/** 标签添加样式的时候会遍历所有的div标签并且添加样式 */
p span a{color:red}
/** 后代选择器找标签的时候 会先去找页面上所有的a标签，然后找这些标签上面的span标签最后在找上面的p标签，这样最后在符合条件的标签上面添加样式 */
p>span>.btn{color:red}
/** 子代选择器和后代选择器差不多 先找类btn标签，然后在他的父标签里面找span标签，最后在span标签的父标签里面找p标签 */
#btn{color:red}
/** 直接在页面中找id为btn的标签，性能消耗最小 */
</style>
```



当两颗树都绘制好之后就开始生成渲染树，生成渲染树的这个过程是非常消耗性能的

- 首先每一次添加新的样式都会造成部分结构树的重新渲染
- 每一个节点的display:none都不会渲染该节点
- 每一次重新渲染都会递归所有的DOM树和CSSOM树

页面渲染完成之后如果我在在对页面的样式或者结构修改的时候页面一般会发生二种情况

- 重绘（Repaint）重绘是当节点需要更改外观而不会影响布局的，比如改变 background-color 就叫称为重绘

- 回流（Reflow）回流是布局或者几何属性需要改变就称为回流。

回流必定会发生重绘，重绘不一定会引发回流。回流所需的成本比重绘高的多，改变父节点里的子节点很可能会导致父节点的一系列回流。
有一些操作会导致回流

- 改变 盒子 大小
- 改变字体
- 添加或删除样式
- 文字改变
- 定位或者浮动
- 盒模型
- 添加删除标签
- 获取盒子宽高

减少回流的方案

- 使用css3 的transform
- 减少display的使用
- CSS 选择符从右往左匹配查找，避免节点层级过多

箭头函数

箭头函数表达式的语法比函数表达式更短，并且没有自己的this，arguments，super或new.target。这些函数表达式更适用于那些本来需要匿名函数的地方，并且它们不能用作构造函数。

伪数组转真数组

```
let arr = {'0' : 'a' , '1': 'b' , '2':'c', 'length' : 3};
let newArr = [].slice.call(arr)

// es6
let newArr = Array.from(arr);
```

柯里化

柯里化是指这样一个函数(假设叫做createCurry)，他接收函数A作为参数，运行后能够返回一个新的函数。并且这个新的函数能够处理函数A的剩余参数。

// 实现一个Currie方法，使计算结果能够满足如下预期：

```
Currie(1)(2)(3)(4) = 10;
Currie(1, 2, 3)(4 , 5) = 15;
Currie(1)(2)(3)(4)(5) = 15;


function Currie(...nums){
// 将传入的数据 以数组形式保存起来
    let arrs = [...nums]
    //  创建一个方法 将我每一次传入的值保存到我的数组中
        function Add( ...nums ){
        arrs.push( ...nums )
        return Add;
     }
// 当最后阶段的时候函数会被放回，所以这里会隐式迭代，所以我们正好可以修改toString方法放回我们需要的数据；
    Add.toString = () => arrs.reduce( (a,b) => a+b );

   return Add;
}

 console.log(Currie(1,3,4)(3,46,2)(3))
```

节流函数

节流throttle
节流是会说预先设定一个执行周期，当调用动作的时刻大于等于执行周期则执行该动作，然后进入下一个新周期。

```
    <button>asd</button>
    <script>
        let btn = document.querySelector('button')
        let the = throttle( function(){
            console.log(this)
        } ,3000 )
        btn.onclick = the;
```

```
    function throttle(fun , time){
        // 定时器
        let timeOut;
        // 当前时间
        let startTime = new Date();
        // 返回方法调用
        return  function() {
            // 获取当前时间
            let curTime = new Date();
            // 关闭定时器
            clearTimeout(timeOut);
            // 判断时间 是否到了
            if( curTime - startTime >= time ){
                // 时间到了之后调用方法并且传入this
                fun.call(this);
                // 更新开始时间
                startTime = new Date();
            }else{
                // 没有达到时间的开启一个定时器
                timeOut = setTimeout(() => {
                    // 定时器时间到了之后调用方法
                    fun.call(this);
                    // 更新时间
                    startTime = new Date();
                }, time - (curTime - startTime));
            }

        }
        }
    </script>
```

函数防抖

debounce，去抖动。策略是当事件被触发时，设定一个周期延迟执行动作，若期间又被触发，则重新设定周期，直到周期结束，执行动作

```
<button>asd</button>

  <script>

      let btn = document.querySelector('button')
      let the = throttle( function(){
          console.log(this)
      } ,300 )
      btn.onclick = the;

      function throttle(fun , time){
          // 定时器
          let timeOut;
          // 返回方法调用
          return  function() {
              clearTimeout(timeOut)
              // 没有达到时间的开启一个定时器
              timeOut = setTimeout(() => {
                      // 定时器时间到了之后调用方法
                      fun.call(this);
              }, time);

          }
  }

      </script>
```

前端安全

XSS
XSS 简单一点就是黑客将代码注入到你的页面中然后运行代码
XSS 分两种类型： 持久型和非持久型。
持久型 就是指将代码注入你的服务器，当你在某个时刻调用的时候会直接运行代码，窃取到你的数据。
非持久型 一般通过修改 URL 参数的方式加入攻击代码，诱导用户访问链接从而进行攻击。

- 持久型： 比如说在输入密码的时候，我将一段后台语言的代码写在密码框里面，比方说java代码

写在密码框里面，这段代码被发送到后台之后可能直接就运行了，这样就会造成无法想象的后果

- 非持久型：一般都是在地址栏里面写一些转账之类的代码，诱导用户点击，但是现在绝大多数的浏览器会过滤到这些代码

xss 防御办法

- 转义字符

```
function xss(str){
    return str.replace(/&/g, '&amp;')
            .replace(/</g, '&lt;')
            .replace(/>/g, '&gt;')
            .replace(/"/g, '&quto;')
            .replace(/'/g, '&#39;')
            .replace(/`/g, '&#96;')
            .replace(/\//g, '&#x2F;')
}
```

- CSP

csp 设置一个白名单告诉浏览器，哪些数据可以请求，哪些数据不可以请求

```
<meta http-equiv="Content-Security-Policy:default-src ‘self’">
```

- 告诉浏览器只有本域名的资源可以请求。

CSRF(跨站请求伪造)
跨站请求伪造就是，黑客搭建一个自己的服务，当用户在未知情况下登录了这个网站，黑客就可以用你的身份发送请求给某些网站窃取资料
防御手段

- 不让第三方平台请求
- Token

事件劫持
事件劫持就是黑客创建一个网站，然后在网站中设置iframe标签，引入银行之类的网站. 然后作者在使用样式遮挡等操作诱导用户点击或者输入他想要的数据

- X-FRAME-OPTIONS

后台通过设置 X-FRAME-OPTIONS：DENY 就可以禁止iframe标签

证明判断json

最有效的方法

```
function isJson(str) {
        try {
            if (typeof JSON.parse(str) == "object") {
                return true;
            }
        } catch(e) {
        }
        return false;
    }
```

移动端1px

在实际开发过程中，手机端写1px像素的线条看上去会比实际上粗一点
一般情况下UI设计师眼中1px 指的是物理像素，而手机上的物理像素是完全不同的，每一个手机厂商的1px是不一样的，所以才会出现1px像素看起开不一样大的问题

解决方案

- 媒体查询利用设备像素比缩放，设置小数像素；（这种方法兼容性十分差，一般浏览器不支持小数）
- 设置 border-image （这种方法需要UI做1px的线）
- transform: scale(0.5) 方案 - 推荐: 很灵活 (缩小线的大小)

JS中的文档碎片

文档碎片 就是我们都知道，dom的操作是很消耗性能的，所以当我们需要连续操作dom的时候，我们可以先用代码将这些操作保存起来，然后代码处理完成之后在一次性添加到dom上去。这就叫做文档碎片。

```
let imgs = ['img1.png','img2.png','img3.png','img4.png','img5.png','img
6.png','img7.png']
// document.createDocumentFragment 创建一个虚拟的标签
let Dom = document.createDocumentFragment();
```

```
$.each(arr , (i ,item) => {
// 在虚拟的标签上添加图片
    Dom.appendChild($(`<img src="${item}"></img>`)[0])
})
// 然后在将这个虚拟的标签添加到真实的标签上，页面这个时候才会渲染
div.appendChild(Dom)
```

MutationObserver 构造函数是什么?

Mutation Observer API 用来监视 DOM 变动。DOM 的任何变动，比如节点的增减、属性的变动、文本内容的变动，这个 API 都可以得到通知。
Mutation Observer 监听是异步，并且在多个标签同时发生修改的时候之后调用一次

```
<input type="text" value="123">

    <script>
        // 首先我先创建一个 MutationObserver 实例。实例里面传入一个回调函数，这个回调函数的目的是为了触发事件之后调用这个方法。
        var observer = new MutationObserver(function (mutations, observer) {
        mutations.forEach(function(mutation) {
            console.log(mutation);
        });
        });
        // 使用observer实例里面的observe方法开始监听 第一个传入标签，第二个传入参数，监听class改动事件
        observer.observe( document.querySelector('input') , {
            'childList': true,
            'attributes':true,
            'characterData':true,
            attributeFilter:[ 'class']
        } )
        document.querySelector('input').addEventListener('keyup' , function(){
            this.classList.add(this.value)
        })
```

```
    </script>
```

iphonex 刘海头

```
.yourFooterClass {
  padding-top: env(safe-area-inset-top);
  padding-right: env(safe-area-inset-right);
  padding-bottom: 50px;  /* 兼容不支持 env( ) 的设备  */
  padding-bottom: calc(env(safe-area-inset-bottom) + 50px); /* 在 iphon
e x + 中本句才会生效 */
  padding-left: env(safe-area-inset-left);
}

constant(safe-area-inset-top)：在Viewport顶部的安全区域内设置量（CSS像素）
constant(safe-area-inset-bottom)：在Viewport底部的安全区域内设置量（CSS像素）
constant(safe-area-inset-left)：在Viewport左边的安全区域内设置量（CSS像素）
constant(safe-area-inset-right)：在Viewport右边的安全区域内设置量（CSS像素）
```

浏览器垃圾回收机制

v8 引擎的垃圾回收机制是怎么样的

- v8引擎是使用的准确式GC，GC算法采用了分代式垃圾回收机制
- v8引擎将内存空间分为二个部分，新生代和老年代，这二个部分里面的大小空间和生命周期是不一样的。

1. 新生代中的空间一般比较小，生命周期比较短。在新生代空间中，其中空间主要分为二部分，这二个空间一定会有一个是使用的另一个是空闲的，当其中一个部分的内存被填充满的时候，新生代算法就会启动将这部分里面存活的对象(正在使用的对象)转移到另外一个部分中，并且将失活的对象销毁。如此一个来回就是一个新生代的算法过程，并且会给每一个进行过一个来回的对象打上标记。

2. 老年代中的空间一般比较大，并且生命周期也很长。

   对象进入老年代算法中只有二种情况，在新生代中经历过一次轮回并且被打上标记的标签，第二种就是对象的大小占用了新生代的25%的对象

老年代阶段会遍历所有的对象，然后标记所有的存活的对象，然后销毁掉所有失活的对象。
最新版本的v8 引擎 新生代和老年代以及js是同时运行的。所以不会有什么性能消耗。

3. 清除对象会造成内存碎片，当碎片过多的时候就会启动压缩算法(将碎片压缩，然后移动存活的对象，在释放掉内存，释放完成之后在将对象重新移动进去)

判断对象的属性名是否在对象本身上面而不是原型上面

```
// 判断对象里面有没有这个属性名 而不是原型上面

function hasOwn( obj , key){
    return Object.prototype.hasOwnProperty.call(obj, key)
}
```

Object.seal 密封

Object.seal()方法封闭一个对象，阻止添加新属性并将所有现有属性标记为不可配置。当前属性的值只要可写就可以改变

Object.freeze 冷冻

Object.freeze() 方法可以冻结一个对象。一个被冻结的对象再也不能被修改；冻结了一个对象则不能向这个对象添加新的属性，不能删除已有属性，不能修改该对象已有属性的可枚举性、可配置性、可写性，以及不能修改已有属性的值。此外，冻结一个对象后该对象的原型也不能被修改。freeze() 返回和传入的参数相同的对象。

['1', '2', '3'].map(parseInt)

答案是[1,NaN,NaN]

```
// 首先 我们 看一下map
function map ( currentValue[, index[, array]] )
//第一个参数 是当前的value值 第二个是 当前下标 第三个是this也就是当前数组
```

```
// 我们在看一下parseInt
function parseInt(string, radix)
// 第一个 表示要被处理的值 第二个表示解析的基数

parseInt('1', 0) //radix为0时，且string参数不以“0x”和“0”开头时，按照10为基数处理。这个时候返回1
parseInt('2', 1) //基数为1（1进制）表示的数中，最大值小于2，所以无法解析，返回NaN
parseInt('3', 2) //基数为2（2进制）表示的数中，最大值小于3，所以无法解析，返回NaN
```

深度优先遍历 和广度优先遍历

深度优先遍历 指 在一个节点树中，找到一个没有被遍历过的顶点 深度优先遍历，当遍历到末尾时重新找到下一个没有被遍历过的节点继续向下遍历
广度优先遍历 指 在一个节点树中，找到节点树的顶点，依次逐层遍历直至全部遍历完成

异步解决方案的发展历程以及优缺点。

回调函数
缺点：回调地狱，不能用 try catch 捕获错误，不能 return
优点：解决了同步的问题
Promise
优点：解决了回调地狱的问题
缺点：无法取消 Promise ，错误需要通过回调函数来捕获
Generator
特点：可以控制函数的执行，可以配合 co 函数库使用
Async/await
优点是：代码清晰，不用像 Promise 写一大堆 then 链，处理了回调地狱的问题
缺点：await 将异步代码改造成同步代码，如果多个异步操作没有依赖性而使用 await 会导致性能上的降低。

Promise 构造函数是同步执行还是异步执行，那么 then 方法呢?

Promise new的时候会立即执行里面的代码 then是微任务 会在本次任务执行完的时候执行
setTimeout是宏任务 会在下次任务执行的时候执行

代码题1

```
var a = ?

if(a == 1 && a == 2 && a == 3){}

var a = {
    i: 1,
    toString(){ return this.i++ }
}
```

判断数组的方法， 以及他们的优缺点

- Object.prototype.toString.call()

  每一个继承 Object 的对象都有 toString 方法，如果 toString 方法没有重写的话，会返回 [Object type]，其中 type 为对象的类型。但当除了 Object 类型的对象外，其他类型直接使用 toString 方法时，会直接返回都是内容的字符串，所以我们需要使用call或者apply方法来改变toString方法的执行上下文。

```
    const arr = [1,2,3];
    Object.prototype.toString.call(arr) // "[object Array]"
```

- instanceof

  instanceof 的内部机制是通过判断对象的原型链中是不是能找到类型的 prototype
  但 instanceof 只能用来判断对象类型，原始类型不可以。并且所有对象类型 instanceof Object 都是 true。

```
[]  instanceof Array; // true
[]  instanceof Object; // true
```

- Array.isArray()

  用来判断对象是否数组

  Array.isArray()是ES5新增的方法，当不存在 Array.isArray() ， 可以用 Object.prototype.toString.call() 实现。

浏览器优化

现代浏览器大多都是通过队列机制来批量更新布局，浏览器会把修改操作放在队列中，至少一个浏览器刷新（即16.6ms）才会清空队列，但当你获取布局信息的时候，队列中可能有会影响这些属性或方法返回值的操作，即使没有，浏览器也会强制清空队列，触发回流与重绘来确保返回正确的值。
主要包括以下属性或方法

- offsetTop、offsetLeft、offsetWidth、offsetHeight
- scrollTop、scrollLeft、scrollWidth、scrollHeight
- clientTop、clientLeft、clientWidth、clientHeight
- width、height
- getComputedStyle()
- getBoundingClientRect()

我们应该避免频繁的使用上述的属性，他们都会强制渲染刷新队列

减少重绘与回流

css

- 使用 transform 替代 top
- 使用 visibility 替换 display: none ，因为前者只会引起重绘，后者会引发回流（改变了布局)
- 避免使用table布局，可能很小的一个小改动会造成整个 table 的重新布局。
- 尽可能在DOM树的最末端改变class，回流是不可避免的，但可以减少其影响。尽可能在DOM树的最末端改变class，可以限制了回流的范围，使其影响尽可能少的节点。
- 避免设置多层内联样式，CSS 选择符从右往左匹配查找，避免节点层级过多

```
<div>
  <a> <span></span> </a>
</div>
<style>
  span {
    color: red;
  }
  div > a > span {
    color: red;
  }
</style>
```

- 将动画效果应用到position属性为absolute或fixed的元素上，避免影响其他元素的布局，这样只是一个重绘，而不是回流，同时，控制动画速度可以选择 requestAnimationFrame，详见探讨

requestAnimationFrame。
- 避免使用CSS表达式，可能会引发回流。
- 将频繁重绘或者回流的节点设置为图层，图层能够阻止该节点的渲染行为影响别的节点，例如will-change、video、iframe等标签，浏览器会自动将该节点变为图层
- CSS3 硬件加速（GPU加速），使用css3硬件加速，可以让transform、opacity、filters这些动画不会引起回流重绘 。但是对于动画的其它属性，比如background-color这些，还是会引起回流重绘的，不过它还是可以提升这些动画的性能。

Javascript

- 避免频繁操作样式，最好一次性重写style属性，或者将样式列表定义为class并一次性更改class属性。
- 避免频繁操作DOM，创建一个documentFragment，在它上面应用所有DOM操作，最后再把它添加到文档中。
- 避免频繁读取会引发回流/重绘的属性，如果确实需要多次使用，就用一个变量缓存起来。
- 对具有复杂动画的元素使用绝对定位，使它脱离文档流，否则会引起父元素及后续元素频繁回流。

代码题 2

```
var b = 10;
(function b(){
    b = 20;
    console.log(b);
})();
// 在非匿名自执行函数中，函数变量为只读状态无法修改（级别小于传参）；
// 所以打印出来的是一个函数体
```

实现一个sleep(1000)

```
//Promise
const sleep = time => {
  return new Promise(resolve => setTimeout(resolve,time))
}
sleep(1000).then(()=>{
  console.log(1)
})
```

```
//Generator
function* sleepGenerator(time) {
  yield new Promise(function(resolve,reject){
    setTimeout(resolve,time);
  })
}
sleepGenerator(1000).next().value.then(()=>{console.log(1)})

//async
function sleep(time) {
  return new Promise(resolve => setTimeout(resolve,time))
}
async function output() {
  let out = await sleep(1000);
  console.log(1);
  return out;
}
output();

//ES5
function sleep(callback,time) {
  if(typeof callback === 'function')
    setTimeout(callback,time)
}

function output(){
  console.log(1);
}
sleep(output,1000);
```

代码题 3

```
var obj = {
    '2': 3,
    '3': 4,
    'length': 2,
```

```
    'splice': Array.prototype.splice,
    'push': Array.prototype.push
}
// push 的时候 如果不是数组会调用splice方法将他转换成伪数组
obj.push(1)
// push 添加的时候是根据的length;
obj.push(2)
console.log(obj)
```

call 和 apply 的区别是什么，那个性能会更好一些

Function.prototype.apply和Function.prototype.call 的作用是一样的，区别在于传入参数的不同
第一个参数都是，指定函数体内this的指向
第二个参数开始不同，apply是传入带下标的集合，数组或者类数组，apply把它传给函数作为参数，call从第二个开始传入的参数是不固定的，都会传给函数作为参数
call比apply的性能要好，平常可以多用call, call传入参数的格式正是内部所需要的格式

代码3

```
var a = {n: 1};
var b = a;
a.x = a = {n: 2};

console.log(a.x)
console.log(b.x)
```

首先第一步 进行赋值 a = {n:1} b = a
赋值 首先先考虑 优先级 . 最高 所以 a = {n:1 , x: undefined}
接下来a = {n:2} 但是a.x 还是 {n:1 , x: undefined} 所以 {n:1 , x : {n:2}};
赋值完成 a = {n:2} b = {n:1 , x:undefined } 在 a.x 赋值的时候 a指向老的 对象

代码4

```
// example 1
var a={}, b='123', c=123;
```

```
a[b]='b';

// c 的键名会被转换成字符串'123'，这里会把 b 覆盖掉。
a[c]='c';

// 输出 c
console.log(a[b]);
// example 2
var a={}, b=Symbol('123'), c=Symbol('123');

// b 是 Symbol 类型，不需要转换。
a[b]='b';

// c 是 Symbol 类型，不需要转换。任何一个 Symbol 类型的值都是不相等的，所以不会覆盖掉 b。
a[c]='c';

// 输出 b
console.log(a[b]);
// example 3
var a={}, b={key:'123'}, c={key:'456'};

// b 不是字符串也不是 Symbol 类型，需要转换成字符串。
// 对象类型会调用 toString 方法转换成字符串 [object Object]。
a[b]='b';

// c 不是字符串也不是 Symbol 类型，需要转换成字符串。
// 对象类型会调用 toString 方法转换成字符串 [object Object]。这里会把 b 覆盖掉。
a[c]='c';

// 输出 c
console.log(a[b]);
```

判断正确的地址

```
function isUrl(url) {
    const a = document.createElement('a')
    a.href = url
    return [
        /^(http|https):$/.test(a.protocol),
        a.host,
        a.pathname !== url,
        a.pathname !== `/${url}`,
    ].find(x => !x) === undefined
}
```

BigInt

BigInt是一种新的数据类型，用于当整数值大于Number数据类型支持的范围时。这种数据类型允许我们安全地对大整数执行算术操作，表示高分辨率的时间戳，使用大整数id，等等，而不需要使用库

Node

内存泄漏

指由于疏忽或错误造成程序未能释放已经不再使用的内存的情况。如果内存泄漏的位置比较关键，那么随着处理的进行可能持有越来越多的无用内存，这些无用的内存变多会引起服务器响应速度变慢，严重的情况下导致内存达到某个极限（可能是进程的上限，如 v8 的上限1.4G；也可能是系统可提供的内存上限）会使得应用程序崩溃。
在传统的语言中C/C++ 中存在着野指针的问题，对象在使用完成之后没有被立即释放掉，导致内存泄漏。但是像一些虚拟机运行的语言，java和JavaScript Golang 中使用了垃圾回收机制GC，这种机制会很好的控制内存的释放，解放程序员的工作
在node中 全局变量引用 未声明对象 事件监听 缓存大数据 会造成内存泄漏

```
let arr = [];
while(true)
  arr.push(1);
  // 当前代码会在运行期间一直添加数据，但是系统又无法将他释放掉最后会导致内存溢出

  let arr = [];
while(true)
  arr.push();
  // 因为push里面是空的，所以每一次添加都是不成功的,所以不会导致内存溢出
let arr = [];
while(true)
  arr.push(new Buffer(1000));
  // 每一次添加的都是Buffer内容 ， buffer类型不属于v8引擎的管控，所以在v8引擎中使用buffer所占用的内存并不会受到1.4g限制的影响，但是它会消耗系统的内存
```

热更新

后端热更新指 在代码运行期间不停止代码的同时又更新代码，更新完成之后可以立即调用
热更新插件auto-reload

Context 上下文

当时运行的环境本身
对于代码中某个值来说，上下文是指这个值所在的局部(全局)作用域对象

相对于进程而言，上下文就是进程执行时的环境，具体来说就是各个变量和数据，包括所有的寄存器变量、进程打开的文件、内存（堆栈）信息等

VM

沙箱 虚拟机上下文
可以让 JavaScript代码可以立即编译和运行，也可以编译，保存，以后运行

```
const vm = require('vm');

const x = 1;

const sandbox = { x: 2 };
vm.createContext(sandbox); // Contextify the sandbox.

const code = 'x += 40; var y = 17;';
// x and y are global variables in the sandboxed environment.
// Initially, x has the value 2 because that is the value of sandbox.x.
vm.runInContext(code, sandbox);

console.log(sandbox.x); // 42
console.log(sandbox.y); // 17

console.log(x); // 1; y is not defined.
```

crypto

crypto 模块提供了加密功能，包括对 OpenSSL 的哈希、HMAC、加密、解密、签名、以及验证功能的一整套封装。

为什么输入npm install 就可以自动安装对应的模块

npm 模块安装的机制

- 发出 npm install 命令
- 查询node_modules 目录之中是否已经存在指定模块
- 若存在，不在安装
- 若不存在

- npm 向 registry 查询模块压缩包的网址
- 下载压缩包到根目录下的.npm目录里
- 解压压缩包到当前项目的node_modules 目录

  npm 实现原理

- 执行工程自身preinstall

  当前npm工程如果定义了preinstall钩子此时会被执行。

- 确定首层依赖模块

  首先需要做的是确定工程中的首层依赖，也就是 package.json 中 dependencies 和 devDependencies 属性中直接指定的模块
  工程本身是整棵依赖树的根节点，每个首层依赖模块都是跟节点下面的一棵子树， npm会开启多进程从每个首层依赖模块开始逐步寻找更深层级的节点。

- 获取模块

  获取模块信息。在下载一个模块之前，首先要确定其版本，这是因为 package.json 中往往是semantic version，此时如果版本描述的文件中在（npm-shrinkwrap.json 或 package-lock.json）有该模块信息直接拿就可以了，如果没有则从从仓库获取。如packaeg.json 中某个包的版本是$^{1.1.0}$,npm就会去仓库中获取复合1.x.x形式的最新版本。
  获取模块实体，上一步会获取到模块的压缩包地址（resolved 字段），npm 会用此地址检查本地缓存，缓存中有就直接拿，如果没有则从仓库下载
  查找该模块依赖，如果有依赖则回到第1步，如果没有则停止。

- 模块扁平化（dedupe）

  上一步获取到的是一棵完整的依赖树，其中可能包含大量重复模块。比如 A 模块依赖于 loadsh，B 模块同样依赖于 lodash。在 npm3 以前会严格按照依赖树的结构进行安装，因此会造成模块冗余。

  从 npm3 开始默认加入了一个 dedupe 的过程。它会遍历所有节点，逐个将模块放在根节点下面，也就是 node-modules 的第一层。当发现有重复模块时，则将其丢弃。

  这里需要对重复模块进行一个定义，它指的是模块名相同且 semver 兼容。每个 semver 都对应一段版本允许范围，如果两个模块的版本允许范围存在交集，那么就可以得到一个兼容版本，而不必版本号完全一致，这可以使更多冗余模块在 dedupe 过程中被去掉。

  比如 node-modules 下 foo 模块依赖 lodash@$^{1.0.0，bar}$ 模块依赖 lodash@$^{1.1.0，则 1.1.0}$ 为兼容版本。

  而当 foo 依赖 lodash@$^{2.0.0，bar}$ 依赖 lodash@$^{1.1.0，则依据}$ semver 的规则，二者不存在兼容版本。会将一个版本放在 node_modules 中，另一个仍保留在依赖树里。

- 安装模块

  这一步将会更新工程中的 node_modules，并执行模块中的生命周期函数（按照 preinstall、install、postinstall 的顺序）。

- 执行工程自身生命周期

  当前 npm 工程如果定义了钩子此时会被执行（按照 install、postinstall、prepublish、prepare 的顺序）。
  最后一步是生成或更新版本描述文件，npm install 过程完成。

浏览器的事件循环和node的事件循环有什么区别

关于微任务和宏任务在浏览器的执行顺序是这样的

- 执行一次微任务
- 之后在执行完微任务队列

如此循环往复下去

Node

node中宏任务队列执行顺序是这样的

- timers定时器：本阶段执行已经安排的 setTimeout() 和 setInterval() 的回调函数。
- pending callbacks待定回调：执行延迟到下一个循环迭代的 I/O 回调。
- idle, prepare：仅系统内部使用。
- poll 轮询：检索新的 I/O 事件;执行与 I/O 相关的回调（几乎所有情况下，除了关闭的回调函数，它们由计时器和 setImmediate() 排定的之外），其余情况 node 将在此处阻塞。
- check 检测：setImmediate() 回调函数在这里执行。
- close callbacks 关闭的回调函数：一些准备关闭的回调函数，如：socket.on('close',...)。

微任务执行在node10之前：

- 执行完一个阶段的所有任务
- 执行完nextTick队列里面的内容
- 然后执行完微任务队列的内容

Node11以后：
和浏览器的行为统一，都是每执行一次宏任务就执行完微任务队列

常见的 task（宏任务） 比如：setTimeout、setInterval、script（整体代码）、 I/O 操作、UI

渲染等。

常见的 micro-task 比如: new Promise().then(回调)、MutationObserver(html5新特性) 等。

EcmaScript6.0

Proxy

Proxy用于修改某些操作的默认行为
Proxy可以理解为对在目标对象上面设置一层拦截

```
var obj = new Proxy({}, {
  get: function (target, key, receiver) {
    console.log(`getting ${key}!`);
    return Reflect.get(target, key, receiver);
  },
  set: function (target, key, value, receiver) {
    console.log(`setting ${key}!`);
    return Reflect.set(target, key, value, receiver);
  }
});

obj.count = 1
//  setting count!
++obj.count
//  getting count!
//  setting count!
//  2
```

var let const 的区别

- var 会变量提升 let和const不会
- var 在全局命名的变量会挂载到window上，let const不会
- let 和cosnt 有块级作用（暂时性死区），var没有
- let const 不允许重复命名

class

es6新增加了class方法， 但是class只是一个语法糖他和我们平时写的构造函数本质上没有

任何区别。

```
// 传统方式
function Point(x, y) {
  this.x = x;
  this.y = y;
}

Point.prototype.toString = function () {
  return '(' + this.x + ', ' + this.y + ')';
};

var p = new Point(1, 2);

// 新的方式
class Point {
  constructor(x, y) {
    this.x = x;
    this.y = y;
  }

  toString() {
    return '(' + this.x + ', ' + this.y + ')';
  }
}

class Point {
  // ...
}

typeof Point // "function"
Point === Point.prototype.constructor // true
// 本质上来说 class创建的构造函数和直接写的构造函数没有太大区别
```

取值函数（getter）和存值函数（setter）

```
class Defind{
    constructor(){}

    get name(){
        return 'get'
    }

    set name(val){
        console.log( 'set:'+val )
    }
}

let defind = new Defind();

defind.name ='xxf' // set:xxf;
defind.name // get
```

静态方法
类相当于实例的原型，所有在类中定义的方法，都会被实例继承。如果在一个方法前，加上static关键字，就表示该方法不会被实例继承，而是直接通过类来调用，这就称为“静态方法”。

```
class Foo {
  static classMethod() {
    return 'hello';
  }
}

Foo.classMethod() // 'hello'

var foo = new Foo();
foo.classMethod()
// TypeError: foo.classMethod is not a function
```

实例属性的新写法

```
class cl{
    constructor(){
        this._count = 1;
    }
}

// 新的写法
class cl{
    _count = 1;
    constructor(){

    }
}
```

Promise

Promise表示一个异步操作的最终结果，以之交互的方式主要有then方法，该方法主要注册了二个回调函数，用于接收Promise的最终结果和不能执行的原因。
Promise的状态主要有三种 等待，执行，拒绝

- 等待Pending 等待进入执行或拒绝状态；
- 执行resolve 不能进入其他状态，有一个不可变的最终结果。
- 拒绝Rejected 不能进入其他状态，有一个不可变得拒绝原因。

Promise.then(onResolve, onRejected) then接收二个参数

- 如果onResolve不是函数，其必须被忽略
- 如果onRejected不是函数，其必须被忽略

onResolve 特性

- promise执行结束后其必须被调用，第一个值就是promise的结果。
- promise结束前不可以被调用。
- onResolve调用次数不可以大于一次。

onRejected特性

- promise被拒绝执行后其必须被调用，其第一个参数为 promise 的拒绝原因

- promise结束前不可以被调用。
- onRejected调用次数不可以大于一次。
- onResolve 和 onRejected 必须被作为函数调用（即没有 this 值）

多次调用

- then 方法可以被同一个 promise 调用多次
- 当 promise 成功执行时，所有 onResolve 需按照其注册顺序依次回调
- 当 promise 被拒绝执行时，所有的 onRejected 需按照其注册顺序依次回调

返回

- then 方法必须返回一个 promise 对象

实现一个Promise

```
class MyPromise {

  constructor(func) {
    /**
     * 当前的状态
     * Pending 等待
     * Resolve 执行
     * Rejected 拒绝
     *
     *  */
    this.status = 'Pending';
    // 用户的回调函数
    // 队列
    this.queueResolve = [];
    this.queneRejected = [];
    // 用户的回调函数
    this._func = func;
    // 用户输入的内容
    this._val = null;
  }

  _myRejected(val) {
```

```
    setTimeout(() => {
      // 判断当前的状态是否是等待
      if (this.status !== 'Pending') return;
      // 将现在的状态修改成 执行
      this.status = 'Resolve';
      // 保存当前的数据
      this._val = val;
      this.queueResolve.map(call => call(val));

    }, 0)

  }

  _myResolve(val) {

    setTimeout(() => {
      // 判断当前的状态是否是等待
      if (this.status !== 'Pending') return;

      // 将现在的状态修改成 错误
      this.status = 'Rejected';
      // 保存当前的回调
      this._val = val;
      this.queneRejected.map(call => call(val));

    }, 0)
  }

  then(resolve, reject) {
    resolve = typeof resolve === "function" ? resolve : new Function().
bind(null);
    reject = typeof reject === "function" ? reject : new Function().bin
d(null);

    // let x = this.#func(this.#myRejected, this.#myResolve);

    if (this.status === 'Pending') {
```

```
      this.queueResolve.push(resolve);
      this.queneRejected.push(reject);

    } else if (this.status === 'Resolve')
      this._myResolve(this._func);
    else
      this._myRejected(this._func);

    return
  }

}
```

使用es6实现一个计数器

```
function count(){
  let myCount = 0;
  return function* func() {
      yield ++myCount;
      func();
  }
}
let myCount = count();
myCount().next().value // 1
myCount().next().value // 2
myCount().next().value // 3
```

一句话数组实现找最大值

Math.max(...arr)

一句话实现数组去重

new Set(...arr)

变量的结构赋值

允许按照一定模式，从数组和对象中提取值，对变量进行赋值，这被称为解构
数组

```
let [a, b, c] = [1, 2, 3];
let [foo, [[bar], baz]] = [1, [[2], 3]];
let [ , , third] = ["foo", "bar", "baz"];
let [head, ...tail] = [1, 2, 3, 4];
```

默认值
注意，ES6 内部使用严格相等运算符（===），判断一个位置是否有值。所以，只有当一个数组成员严格等于undefined，默认值才会生效。

```
let [foo = true] = [];
foo // true

let [x, y = 'b'] = ['a']; // x='a', y='b'
let [x, y = 'b'] = ['a', undefined]; // x='a', y='b'
```

对象的结构

```
let { bar, foo } = { foo: "aaa", bar: "bbb" };
foo // "aaa"
bar // "bbb"
```

字符串的遍历器接口

ES6 为字符串添加了遍历器接口，使得字符串可以被for...of循环遍历。

```
for (let codePoint of 'foo') {
  console.log(codePoint)
}
// "f"
// "o"
```

```
// "o"
```

includes(), startsWith(), endsWith()

传统的js中只提供了indexOf() 查找匹配对应的字符串。
es6 中提供了三种方法

```
includes()：返回布尔值，表示是否找到了参数字符串。
startsWith()：返回布尔值，表示参数字符串是否在原字符串的头部。
endsWith()：返回布尔值，表示参数字符串是否在原字符串的尾部。

let s = 'Hello world!';

s.startsWith('world', 6) // true
s.endsWith('Hello', 5) // true
s.includes('Hello', 6) // false
// 第一个参数是 需要匹配的数据 ，第二个参数是从第几个开始。
```

字符串模板

模板字符串（template string）是增强版的字符串，用反引号（`）标识。它可以当作普通字符串使用，也可以用来定义多行字符串，或者在字符串中嵌入变量。

```
let x = 1;
let y = 2;

`${x} + ${y} = ${x + y}`
// "1 + 2 = 3"

`${x} + ${y * 2} = ${x + y * 2}`
// "1 + 4 = 5"

let obj = {x: 1, y: 2};
`${obj.x + obj.y}`
// "3"
function fn() {
```

```
  return "Hello World";
}

`foo ${fn()} bar`
// foo Hello World bar
```

Number.isFinite(), Number.isNaN()

ES6 在Number对象上，新提供了Number.isFinite()和Number.isNaN()两个方法。Number.isFinite()用来检查一个数值是否为有限的（finite），即不是Infinity。

```
Number.isFinite(15); // true
Number.isFinite(0.8); // true
Number.isFinite(NaN); // false
Number.isFinite(Infinity); // false
Number.isFinite(-Infinity); // false
Number.isFinite('foo'); // false
Number.isFinite('15'); // false
Number.isFinite(true); // false
```

Number.isNaN()用来检查一个值是否为NaN。

```
Number.isNaN(NaN) // true
Number.isNaN(15) // false
Number.isNaN('15') // false
Number.isNaN(true) // false
Number.isNaN(9/NaN) // true
Number.isNaN('true' / 0) // true
Number.isNaN('true' / 'true') // true
```

它们与传统的全局方法isFinite()和isNaN()的区别在于，传统方法先调用Number()将非数值的值转为数值，再进行判断，而这两个新方法只对数值有效，Number.isFinite()对于非数值一律返回false, Number.isNaN()只有对于NaN才返回true，非NaN一律返回false。

Number.isInteger()

Number.isInteger()用来判断一个数值是否为整数。

```
Number.isInteger() // false
Number.isInteger(null) // false
Number.isInteger('15') // false
Number.isInteger(true) // false
Number.isInteger(25) // true
Number.isInteger(25.0) // true
Number.isInteger(25.1) // false
Number.isInteger(3.0000000000000002) // true
```

Math.trunc()

Math.trunc方法用于去除一个数的小数部分，返回整数部分。

```
Math.trunc(4.1) // 4
Math.trunc(4.9) // 4
Math.trunc(-4.1) // -4
Math.trunc(-4.9) // -4
Math.trunc(-0.1234) // -0
```

Math.sign()

Math.sign方法用来判断一个数到底是正数、负数、还是零。对于非数值，会先将其转换为数值。

```
Math.sign(-5) // -1
Math.sign(5) // +1
Math.sign(0) // +0
Math.sign(-0) // -0
Math.sign(NaN) // NaN
```

函数的默认传参

ES6 允许为函数的参数设置默认值，即直接写在参数定义的后面。

```
function func( num = 12 ){
    console.log(num)
}
func() // 12
func(15) // 15
```

rest 参数

S6 引入 rest 参数（形式为...变量名），用于获取函数的多余参数，这样就不需要使用arguments对象了。rest 参数搭配的变量是一个数组，该变量将多余的参数放入数组中。

```
function func(...arr){
    console.log(arr)
}
func(1,2,3) // [1,2,3]

function func1(a , ...arr){
    console.log(a , arr)
}
func(1,2,3) // 1 , [2,3]
```

扩展运算符

扩展运算符（spread）是三个点（...）。它好比 rest 参数的逆运算，将一个数组转为用逗号分隔的参数序列。

```
console.log(...[1, 2, 3])
// 1 2 3

console.log(1, ...[2, 3, 4], 5)
// 1 2 3 4 5

[...document.querySelectorAll('div')]
// [<div>, <div>, <div>]
```

Array.from

Array.from方法用于将两类对象转为真正的数组

```
let arr = {'0' : 'a' , '1': 'b' , '2':'c', 'length' : 3};
// es6
let newArr = Array.from(arr);
```

find() 和 findIndex()

数组实例的find方法，用于找出第一个符合条件的数组成员。它的参数是一个回调函数，所有数组成员依次执行该回调函数，直到找出第一个返回值为true的成员，然后返回该成员。如果没有符合条件的成员，则返回undefined。
数组实例的findIndex方法的用法与find方法非常类似，返回第一个符合条件的数组成员的位置，如果所有成员都不符合条件，则返回-1。

```
[1, 4, -5, 10].find((n) => n < 0)
// -5

[1, 5, 10, 15].findIndex(function(value, index, arr) {
  return value > 9;
}) // 2
```

数组拍平

数组拍平就是将多维数组转换成一维数组

```
[1, [2, [3]]].flat(Infinity)
// [1, 2, 3]
```

数组实例的 flat()，flatMap()

数组的成员有时还是数组，Array.prototype.flat()用于将嵌套的数组“拉平”，变成一维的数组。该方法返回一个新数组，对原数据没有影响。

```
[1, 2, [3, 4]].flat()
// [1, 2, 3, 4]
[1, 2, [3, [4, 5]]].flat()
// [1, 2, 3, [4, 5]]

[1, 2, [3, [4, 5]]].flat(2)
// [1, 2, 3, 4, 5]

[1, [2, [3]]].flat(Infinity)
// [1, 2, 3]
```

flatMap()方法对原数组的每个成员执行一个函数（相当于执行Array.prototype.map()），然后对返回值组成的数组执行flat()方法。该方法返回一个新数组，不改变原数组。

```
// 相当于 [[2, 4], [3, 6], [4, 8]].flat()
[2, 3, 4].flatMap((x) => [x, x * 2])
// [2, 4, 3, 6, 4, 8]
```

Object.is()

ES5 比较两个值是否相等
相等运算符（）和严格相等运算符（=）。它们都有缺点，前者会自动转换数据类型，后者的NaN不等于自身，以及+0等于-0。JavaScript 缺乏一种运算，在所有环境中，只要两个值是一样的，它们就应该相等。

```
+0 === -0 //true
NaN === NaN // false

Object.is(+0, -0) // false
Object.is(NaN, NaN) // true
```

Object.assign

Object.assign方法用于对象的合并，将源对象（source）的所有可枚举属性，复制到目标对

象（target）。

```
const target = { a: 1 };

const source1 = { b: 2 };
const source2 = { c: 3 };

Object.assign(target, source1, source2);
target // {a:1, b:2, c:3}
```

Symbol

ES6 引入了一种新的原始数据类型Symbol，表示独一无二的值。它是 JavaScript 语言的第七种数据类型，前六种是：undefined、null、布尔值（Boolean）、字符串（String）、数值（Number）、对象（Object）。
Symbol 值通过Symbol函数生成。这就是说，对象的属性名现在可以有两种类型，一种是原来就有的字符串，另一种就是新增的 Symbol 类型。凡是属性名属于 Symbol 类型，就都是独一无二的，可以保证不会与其他属性名产生冲突。

```
// 没有参数的情况
let s1 = Symbol();
let s2 = Symbol();

s1 === s2 // false

// 有参数的情况
let s1 = Symbol('foo');
let s2 = Symbol('foo');

s1 === s2 // false


let mySymbol = Symbol();

// 第一种写法
```

```
let a = {};
a[mySymbol] = 'Hello!';

// 第二种写法
let a = {
  [mySymbol]: 'Hello!'
};

// 第三种写法
let a = {};
Object.defineProperty(a, mySymbol, { value: 'Hello!' });

// 以上写法都得到同样结果
a[mySymbol] // "Hello!"
```

Symbol中的内置方法

Symbol.toPrimitive 对象的Symbol.toPrimitive属性，指向一个方法。该对象被转为原始类型的值时，会调用这个方法，返回该对象对应的原始类型值。

```
var obj = {
[Symbol.toPrimitive](){
    return 'symbol'
}
}

obj + ''  // ‘symbol’
```

Object.entries

Object.entries 以数组方式返回 keys 和 values 。

```
let obj = { '🐵为所欲为':'100' , '🙈不可以看':'50' , '🙉不可以听':'30' , '
```

```
🐵不可以说':'10' }

console.log( Object.entries(obj) )

[ [ '🐵为所欲为', '100' ],
  [ '🙈不可以看', '50' ],
  [ '🙉不可以听', '30' ],
  [ '🐵不可以说', '10' ] ]
```

padStart & padEnd

字符串填充

```
let obj = { '🐵为所欲为':'100' , '🙈不可以看':'50' , '🙉不可以听':'30' , '
🐵不可以说':'10' }
//count.padStart 从头开始填充
// 第一位填充 填充结果  第二位填充 内容
/**
let str = 10;
str.padStart(3 , '0') // 010
str.padStart(5 , '-') // ---10
*/
// padEnd 从尾部开始填充
Object.entries(obj).map(
    ([name , count]) => {
        console.log(`
        ${name.padEnd(20,' -')} Count: ${count.padStart(3,'0')}`)
    } )

    // 输出

        🐵为所欲为 - - - - - - - Count: 100

        🙈不可以看 - - - - - - - Count: 050

        🙉不可以听 - - - - - - - Count: 030
```

```
🐵不可以说 - - - - - - - Count: 010
```

Set & Map 数据结构

ES6 提供了新的数据结构 Set。它类似于数组，但是成员的值都是唯一的，没有重复的值。Set本身是一个构造函数，用来生成 Set 数据结构。

```
const s = new Set();

[2, 3, 5, 4, 5, 2, 2].forEach(x => s.add(x));

for (let i of s) {
  console.log(i);
}
// 2 3 5 4
```

ES6 提供了 Map 数据结构。它类似于对象，也是键值对的集合，但是“键”的范围不限于字符串，各种类型的值（包括对象）都可以当作键。也就是说，Object 结构提供了“字符串—值”的对应，Map 结构提供了“值—值”的对应，是一种更完善的 Hash 结构实现。如果你需要“键值对”的数据结构，Map 比 Object 更合适。

```
const m = new Map();
const o = {p: 'Hello World'};

m.set(o, 'content')
m.get(o) // "content"

m.has(o) // true
m.delete(o) // true
m.has(o) // false
```

Reflect

Reflect对象与Proxy对象一样，也是 ES6 为了操作对象而提供的新 API。

for...of 循环

ES6 借鉴 C++、Java、C# 和 Python 语言，引入了for...of循环，作为遍历所有数据结构的统一的方法。

```
const arr = ['red', 'green', 'blue'];

for(let v of arr) {
  console.log(v); // red green blue
}

var engines = new Set(["Gecko", "Trident", "Webkit", "Webkit"]);
for (var e of engines) {
  console.log(e);
}
// Gecko
// Trident
// Webkit

// 字符串
let str = "hello";

for (let s of str) {
  console.log(s); // h e l l o
}

// DOM NodeList对象
let paras = document.querySelectorAll("p");

for (let p of paras) {
  p.classList.add("test");
}

// arguments对象
function printArgs() {
  for (let x of arguments) {
    console.log(x);
  }
```

```
}
printArgs('a', 'b');
// 'a'
// 'b'
```

Generator

Generator 函数是 ES6 提供的一种异步编程解决方案，语法行为与传统函数完全不同。
Generator 函数有多种理解角度。语法上，首先可以把它理解成，Generator 函数是一个状态机，封装了多个内部状态。

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

var hw = helloWorldGenerator();
hw.next().value // hello
hw.next().value // world
hw.next().value // ending

function* Func(){
    for( let i = 0 ;i<3;i++ ){
      yield i;
    }
}
let func = Func();
func.next().value // 0
func.next().value // 1
func.next().value // 2
```

async

ES2017 标准引入了 async 函数，使得异步操作变得更加方便。
async 函数是什么？一句话，它就是 Promise 函数的语法糖。

```
function func(){
    return new Promise((resolve, reject) => {
        setInterval( () => resolve(10) , 1000 )
    });
}

async function myFun(){
    let num = await func();
    console.log(num)
}
myFunc() // 1秒钟之后打印1
```

Module

ES6.0 提供了模块的概念，我们可以向使用模块引用的方式引用模块

```
// ES6模块
import { stat, exists, readFile } from 'fs';
```

模块功能主要由两个命令构成：export和import。export命令用于规定模块的对外接口，import命令用于输入其他模块提供的功能。

export

一个模块就是一个独立的文件。该文件内部的所有变量，外部无法获取。如果你希望外部能够读取模块内部的某个变量，就必须使用export关键字输出该变量。

```
let num = 10;
let obj = {};
let arr = [];
let func = function(){}

export { num , obj , arr , func };

// 可以使用as重命名
export {
    num as myNum,
    obj as myObj,
```

```
    arr as myArr,
    func as myFunc
}
```

import 命令
使用export命令定义了模块的对外接口以后，其他 JS 文件就可以通过import命令加载这个模块

```
import { myNum , myObj , myArr , myFunc } from 'index.js';
import index from 'index.js';
```

ES6 模块与 CommonJS 模块的差异

- CommonJS 模块输出的是一个值的拷贝，ES6 模块输出的是值的引用。
- CommonJS 模块是运行时加载，ES6 模块是编译时输出接口。

循环加载 （阿里面试题 2018/12月）

“循环加载”（circular dependency）指的是，a脚本的执行依赖b脚本，而b脚本的执行又依赖a脚本。

```
// a.js
let a = require('b.js')
// b.js
let b = require('a.js')
```

通常，“循环加载”表示存在强耦合，如果处理不好，还可能导致递归加载，使得程序无法执行，因此应该避免出现。

CommonJS 模块的循环加载
CommonJS 模块的重要特性是加载时执行，即脚本代码在require的时候，就会全部执行。一旦出现某个模块被"循环加载"，就只输出已经执行的部分，还未执行的部分不会输出。

```


// a.js
exports.fl = false;
let b = require('/b.js');
```

```
console.log(`a.js ===> fl:${b.fl}`);
export.fl = true;
console.log( `a.js ===> 执行完成` )

// b.js
exports.fl = false;
let a = require('/a.js');
console.log(`b.js ===> fl:${a.fl}`);
export.fl = true;
console.log( `b.js ===> 执行完毕` )

// index.js
let a = require('/a.js');
let b = require('/b.js');
console.log( `代码执行完毕: a ==> fl:${a.fl}` ; b ==> fl:${b.fl} );

/**
b.js ===> fl: false;
b.js ===> 执行完毕
a.js ===> fl: true;
a.js ===> 执行完毕
代码执行完毕: a ==> fl: true ; b ==> fl: true
*/
```

CommonJS 输入的是被输出值的拷贝，不是引用
CommonJS 模块的重要特性是加载时执行，即脚本代码在require的时候，就会全部执行。一旦出现某个模块被"循环加载"，就只输出已经执行的部分，还未执行的部分不会输出。
执行顺序

- 首先开始 第一步加载a.js 这时候 a = a.js 的地址类似于 a = a{}
- 进入a.js 首先对外暴露一个属性fl a{ fl:false }
- 加载 b.js 并且将值赋值给b b = b{}
- 进入b.js 首先对外暴露一个属性fl b{fl : false}
- 这时候开始加载 a 文件 ， 但是内存中有a{} 将内存中的a{} 给a a = a{};
- 然后打印内容 b.js ===> fl: false; 这个时候 a里面fl是false
- 修改fl 的值为true b{ fl:true };
- 打印 b.js ===> 执行完毕
- 然后文件a 的 reuqire('b.js') 执行完成
- 打印 b.js ===> fl: true; 这个时候b里面的fl为true

- 修改 a.js 对外暴露的fl 为true a{fl:true}
- 打印 a.js ===> 执行完毕
- 然后读取 b文件，内存中已经有b文件了直接将地址改成内存中的地址
- 打印 代码执行完毕：a ==> fl: true ; b ==> fl: true

ES6 模块的循环加载
ES6 处理“循环加载”与 CommonJS 有本质的不同。ES6 模块是动态引用，如果使用import从一个模块加载变量（即import foo from 'foo'），那些变量不会被缓存，而是成为一个指向被加载模块的引用，需要开发者自己保证，真正取值的时候能够取到值。

```
// a.js
import {bar} from './b';
console.log('a.js');
console.log(bar);
export let foo = 'foo';

// b.js
import {foo} from './a';
console.log('b.js');
console.log(foo);
export let bar = 'bar';

/**
* 运行 a.js
b.js
ReferenceError: foo is not defined
*/
```

- 首先第一步 加载 a.js 内存里面有一个 a.js
- a.js 里面的 第一句 加载 b文件，这个时候内存里面加载 b文件 ， 并且开始运行b文件
- b.js 里面的 第一句话 加载 a文件 ， 这个时候内存中有a文件，然后就将 foo 指向 内存中的 a 文件，并且运行 a.js
- a.js 里面第一句 加载 b 内存中有b文件，所以 bar 指向 内存中的 b文件，并且运行b文件
- b.js 运行第一句 ， 因为 foo指向 a 文件 所以不用运行 执行后面的代码
- 第一句打印b.js
- 第二句打印foo 但是 foo虽然指向 a文件 ， 但是没有定义，因为a文件压根就没有运行完成，只是暂存的一个东西。不存在的，所以报错。

es5 的继承和 es6 的继承有什么区别

es5 中的继承主要有 原型链继承 和 构造函数式继承
es6 里面类是通过 class创建的。通过保留关键字extends 实现的继承 实际上这些关键字只是一些语法糖，底层实现还是通过原型链之间的委托关联关系实现继承
实际上这些关键字只是一些语法糖，底层实现还是通过原型链之间的委托关联关系实现继承

es6 中的class

- 首先 calss创建的类是会提升的，但是不会赋值，class创建的类首先会进入暂时性锁死，类似let cosnt （提升指进入TDZ class创建的类会 无视块级作用域屏蔽外部同名变量）
- class 声明内部会启用严格模式。
- class 的所有方法（包括静态方法和实例方法）都是不可枚举的
- class 的所有方法（包括静态方法和实例方法）都没有原型对象 prototype，所以也没有 [[construct]]，不能使用 new 来调用
- 必须使用 new 调用 class。
- class 内部无法重写类名。

let const 声明的变量在哪?

let const 声明的变量在一个块级作用域 （Script）中

Vue

vue除了Object.defineProperty 以外还可以用什么替代

- Vue3.0 Proxy

Vue里面computed是什么东西怎么用?

- 在vue模板里面使用表达式是非常便利的，但是设计它们的初衷是用来简单运算的。在实际开发过程中在模板里面放入过多的表达式会让项目可维护性大大降低。

```
    <div>
        {{ data.split('') ? data.split('').reverse().join('') : data.sp
lit('') }}
    </div>
```

- 对于这种复杂计算的数据，我们应该使用计算属性来解决;
- 计算属性中的方法是依赖于其中的值得，当计算属性中的值发生变化的时候，计算属性会更新

```
var vm = new Vue({
  el: '#example',
  data: {
    data: 'Hello'
  },
  computed: {
    // 计算属性的 getter
    reversedMessage: function () {
      // `this` 指向 vm 实例
      return this.data.split('').reverse().join('')
    }
  }
})
```

- 当this.data的值发生变化的时候，他所依赖的计算属性reversedMessage会重新计算并且调用

Vue里面的watch是什么东西怎么使用

- 虽然计算属性在大部分情况下都适用，但是在实际开发过程中需要一些自定义的监听器，当需要在执行异步或者一些开销比较大的操作中，监听器会比计算属相更加有效。

```
    watch:{
        // 需要监听的值
        question(){

        }
    }
```

计算属性和监听器看上去是差不多的，但是还是有区别。

watch和computed的区别

计算属性和监听属性都是希望在依赖数据发生变化的时候，被依赖的数据根据事先设定好的函数发生自动的变换。
watch 一个数据响应多个数据
computed 一个数据受到多个数据的影响
在实现原理上watch和computed是差不多的，vue的data值在初始化阶段都会被挂载上watcher观察者模式，当数据改变的时候会先调用watcher观察者模式，然后调用计算属性，和监听属性。本质上来说没有多大区别

vue组件之间怎么实现数据的双向绑定

```
<body>
    <script src="https://cdn.bootcss.com/vue/2.5.16/vue.js"></script>
    <div id="box">
        <new-input v-bind:name.sync="name"></new-input>
        {{name}}
        <input type="text" v-model="name" >

    </div>
    <script>
        Vue.component('new-input', {
            props: ['name'],
            data: function () {
                return {
```

```
                    newName: this.name
                }
            },
            template: '<label><input type="text" @keyup="changeName" v-
model="newName" /> 你的名字: </label>',
            methods: {
                changeName: function () {
                    this.$emit('update:name', this.newName);
                }
            },
            watch:{
                "name": function (v){
                    this.newName = v;
                }
            },

        });
        new Vue({
            el: '#box',
            data: {
                name: 'nick'
            }
        });
    </script>
```

首先使用vue中的sync方法，当父组件的变量发生变化的时候，子组件通过prop获取到的数据也会发生变化，但是在子组件里面修改prop传递过来的值就会报错。

所以就需要$emit将子组件的数据传递过去。从而实现组件之间的数据双向绑定。

2.2.0+ 的组件数据双向绑定

v-model

vue-router有哪几种导航钩子

vue一共有3种导航钩子

全局钩子

- 其中全局导航钩子有三种，前置守卫，后置钩子和全局解析守卫
- 前置守卫 router.beforeEach
- 后置守卫 router.afterEach
- 解析守卫 router.beforeResolve
- 其中解析守卫是在2.5版本中新增加的。解析和 router.beforeEach 类似，区别是：在导航被确认之前，同时在所有组件内守卫和异步路由组件被解析之后，解析守卫就被调用

路由独享钩子

- 即单个路由独享的导航钩子，他是在路由配置上直接定义的。

```
cont router = new VueRouter({
    routes: [
        {
            path: '/file',
            component: File,
            beforeEnter: (to, from ,next) => {
                // do someting
            }
        }
    ]
});
```

组件内导航钩子

- 组件内的导航钩子主要有这三种：beforeRouteEnter、beforeRouteUpdate、beforeRouteLeave。他们是直接在路由组件内部直接进行定义的

```
const File = {
    template: `<div>This is file</div>`,
    beforeRouteEnter(to, from, next) {
        // do someting
        // 在渲染该组件的对应路由被 confirm 前调用
    },
    beforeRouteUpdate(to, from, next) {
        // do someting
        // 在当前路由改变，但是依然渲染该组件是调用
    },
    beforeRouteLeave(to, from ,next) {
```

```
        // do someting
        // 导航离开该组件的对应路由时被调用
    }
}
```

导航解析流程说一下

1. 导航被触发
2. 在失活的组件里调用离开守卫
3. 调用全局的 beforeEach 守卫
4. 在重用的组件里调用 beforeRouteUpdate 守卫
5. 在路由配置里调用 beforEnter
6. 解析异步路由组件
7. 在被激活的组件里调用 beforeRouteEnter
8. 调用全局的 beforeResolve 守卫
9. 导航被确认
10. 调用全局的 afterEach 钩子
11. 触发 DOM 更新
12. 在创建好的实例调用 beforeRouteEnter 守卫中传给 next 的回调函数

vue的核心是什么?

vue的核心主要包含二点 响应式驱动还有组件系统

1. 其中响应式驱动，就是建立在es5 的一个方法Object.defineProperty和寄存器getter和setter方法实现的一种设计模型，观察者和订阅者模式。也可以称为基于依赖收集的观测机制，其中他们的核心就是VM（mvvm）视图和数据的双向绑定。
2. 组件系统的核心就是模块，初始化数据，接收外部传参，方法，生命钩子函数，私有资源。

组件是什么？模块是什么?

在实际开发过程中通常有二个开发模式，组件化开发和模块化开发。

1. 组件化：侧重于业务的解耦
   根据页面中的业务逻辑进行划分成组件
2. 模块化： 侧重于重用
   根据页面中的功能进行划分成模块

axios是什么

axios 是一个基于promise的http库，可以用在浏览器和nodeJS中

axios为什么不能再IE9浏览器里面使用

因为axios中使用了很多es6的方法所以axios需要转化一下才可以使用
安装 babel-polyfill 和 es6-promise 插件就可以解决了

axios怎么解决跨域

axios 是不支持jsonp的所以jsonp方法被否决了；
反向代理 就是搭建一个服务器将我的请求通过我的服务发送给对方服务器。因为服务器之间发送请求是不会出现跨域的问题的。
针对后台不支持OPTIONS，要么让后台使用cors插件，要么自己写一个后台，进行反向代理。

vuex是什么？怎么使用？哪种功能场景使用它?

Vuex 是一个专为 Vue.js 应用程序开发的状态管理器，采用 集中式存储 管理应用的所有组件的状态。

- state Vuex store实例的根状态对象，用于定义共享的状态变量。
- Action -动作，向store发出调用通知，执行本地或者远端的某一个操作（可以理解为store的methods）
- Mutations -修改器，它只用于修改state中定义的状态变量。
- getter -读取器，外部程序通过它获取变量的具体值，或者在取值前做一些计算（可以认为是store的计算属性）
  vuex 的使用场景：
- 数据需要在多个组件里面来回调用的
- 全局都要使用的方法，并且方法是需要保存不被浏览器清理掉的。
- 组件需要保存的数据

mvvm框架是什么？它和jquery的区别是什么？哪些场景适合?

- V View
- M model
- vm ViewModel

当视图层发生变化的时候会自动更新vm， vm的变化会自动同步到视图层

框架 是在开发过程中提供一整套的开发方案，你的开发是要根据框架提供的方法和规范来进行开发的。

jquery 是一个插件，在开发过程中只是提供了一点简便，主要是提供辅助性的作用。
jquery不是一个框架，commonJs + templateJs + jquery 一整套才算作半个框架。

vuex 页面刷新数据丢失问题怎么解决

'vuex-along'
使用 localStorage 实时保存vuex的数据；
使用 vuex-along 插件

```
export default new Vuex.Store({
  state:{...},
  ...
  plugins: [vuexAlong]
});
// 现在插件已经生效了
// 默认保存所有state
// 你可以通过相应API来设置
```

vue-loader是什么

解析和转换 .vue 文件，提取出其中的逻辑代码 script、样式代码 style、以及 HTML 模版 template，再分别把它们交给对应的 Loader 去处理。
css-loade：加载由 vue-loader 提取出的 CSS 代码。
vue-template-compiler：把 vue-loader 提取出的 HTML 模版编译成对应的可执行的 JavaScript 代码，这和 React 中的 JSX 语法被编译成 JavaScript 代码类似。预先编译好 HTML 模版相对于在浏览器中再去编译 HTML 模版的好处在于性能更好。
总结;vue-loader的作用就是提取。

$nextTick是什么?

因为vue是异步更新的，$nextTick是用来知道什么时候DOM更新完成的

vue在观察到数据发生变化的时候不会直接更新DOM的，而是开启一个队列，并缓冲在同一个事件循环中发生的所以数据改变，在缓冲时会去除所有的重复数据。从而避免不必要的DOM操作和计算。在下一个事件循环Vue刷新队列并执行实际（已去重的）工作
Vue会根据当前浏览器环境优先使用原生的Promise.then和MutationObserver，如果都不支持，就会采用setTimeout代替。
理论上，我们应该不用去主动操作DOM，因为Vue的核心思想就是数据驱动DOM，但在很多业务里，我们避免不了会使用一些第三方库，比如 popper.js、swiper等，这些基于原生

javascript的库都有创建和更新及销毁的完整生命周期，与Vue配合使用时，就要利用好$nextTick。

```
<div id="app">
    <div id="div" v-if="showDiv">这是一段文本</div>
    <button @click="getText">获取div内容</button>
</div>
<script>
var app = new Vue({
    el : "#app",
    data:{
        showDiv : false
    },
    methods:{
        getText:function(){
            this.showDiv = true;
            this.$nextTick(function(){
                  var text = document.getElementById('div').innerHTML;
                console.log(text);
            });
        }
    }
})
</script>
```

vue中 keep-alive 组件的作用

keep-alive：主要用于保留组件状态或避免重新渲染。
当二个组件被很频繁的调用的时候可以使用keep-alive标签进行缓存，这样页面就会从缓存中快速渲染，而不是重新渲染。

```
<!-- 逗号分隔字符串 -->
<keep-alive include="a,b">
  <component :is="view"></component>
</keep-alive>

<!-- 正则表达式 (使用 `v-bind`) -->
```

```
<keep-alive :include="/a|b/">
  <component :is="view"></component>
</keep-alive>

<!-- 数组 (使用 `v-bind`) -->
<keep-alive :include="['a', 'b']">
  <component :is="view"></component>
</keep-alive>
```

include 和 exclude 属性允许组件有条件地缓存。二者都可以用逗号分隔字符串、正则表达式或一个数组来表示
include - 字符串或正则表达式。只有名称匹配的组件会被缓存。
exclude - 字符串或正则表达式。任何名称匹配的组件都不会被缓存。

axios和ajax的区别

Ajax被认为是（Asynchronous JavaScript and XML）的缩写。现在，允许浏览器与服务器通信而无须刷新当前页面的技术都被叫做Ajax。
依赖的传输对象：XMLHttpRequest
Axios本质上也是对原生XHR的封装，只不过它是Promise的实现版本，符合最新的ES规范，它有以下几条特性：

- 从 node.js 创建 http 请求
- 支持 Promise API
- 客户端支持防止CSRF
- 提供了一些并发请求的接口（重要，方便了很多的操作）
- Axios既提供了并发的封装，也没有fetch的各种问题，而且体积也较小.

vue中的动态路由

即把某种模式匹配到的所有路由，全都映射到同个组件。使用动态路由参数来实现。
例如，我们有一个 User 组件，对于所有ID 各不相同的用户，都要使用这个组件来渲染。那么，我们可以在vue-router的路由路径中使用“动态路径参数”(dynamic segment)来达到这个效果。

```
const User = {
  template: '<div>User</div>'
}
```

```
const router = new VueRouter({
  routes: [
    // 动态路径参数 以冒号开头
    { path: '/user/:id', component: User }
  ]
})
```

这样，像/user/foo和/user/bar都将映射到相同的路由。
一个“路径参数”使用冒号 :标记。当匹配到一个路由时，参数值会被设置到this.$route.params，可以在每个组件内使用。

```
const User = {
  template: '<div>User {{ $route.params.id }}</div>'
}
```

vue-router如何响应 路由参数 的变化?

当使用路由参数的时候 比方说/login?id=1到/login?id=2，的时候，当前组件实例会被复用，所以说这个时候组件的生命周期钩子函数不会被触发，此时vue的路由参数怎么发送变化的?

1. 使用watch来监听

```
const User = {
  template: '...',
  watch: {
    '$route' (to, from) {
      // 对路由变化作出响应...
    }
  }
}
```

1. 或者使用2.2新增加的方法beforeRouteUpdate守卫

```
const User = {
  template: '...',
```

```
  // 在根目录下是失效的 APP.vue
  beforeRouteUpdate (to, from, next) {
    // react to route changes...
    // don't forget to call next()
  }
}
```

vue-router的几种实例方法以及参数传递

三种

实例方法	说明
this.$router.push(location, onComplete?, onAbort?)	这个方法会向 history 栈添加一个新的记录，所以，当用户点击浏览器后退按钮时，则回到之前的 URL。并且点击 等同于调用 router.push(...)。
this.$router.replace(location, onComplete?, onAbort?)	这个方法不会向 history 添加新记录，而是跟它的方法名一样 —— 替换掉当前的 history 记录，所以，当用户点击浏览器后退按钮时，并不会回到之前的 URL
this.$router.go(n)	这个方法的参数是一个整数，意思是在 history 记录中向前或者后退多少步，类似 window.history.go(n)。

在 2.2.0+，可选的在 router.push 或 router.replace 中提供 onComplete 和 onAbort 回调作为第二个和第三个参数。这些回调将会在导航成功完成 (在所有的异步钩子被解析之后) 或终止 (导航到相同的路由、或在当前导航完成之前导航到另一个不同的路由) 的时候进行相应的调用。

如果目的地和当前路由相同，只有参数发生了改变 (比如从一个用户资料到另一个 /users/1 -> /users/2)，你需要使用 beforeRouteUpdate 来响应这个变化 (比如抓取用户信息)。

vue各个组件怎么修改头部的title

首先安装插件vue-wechat-title
然后使用Vue.use安装插件vue-wechat-title

```
// main.js 文件里面的配置
import vueWechatTitle from 'vue-wechat-title'
```

```
Vue.use( vueWechatTitle )


// router index.js 文件里面的配置
  {
    path: '/',
      name: 'Entrance',
      component: Entrance,
      meta: {
      // 声明title
        title: '首页入口'
      }
    },
  }

  // 调用  在router-view 视图里面添加修改title的指令
  <router-view v-wechat-title="$route.meta.title"  />
```

vue-router的动态路由匹配以及使用

动态路由 即将某种模式匹配到的所有路由全部映射到通一个组件里面
假如我们有一个user组件，然后根据不同id来显示不同的信息，这个时候我们就需要使用动态路径参数来达到这个效果

```
const User = {
  template: '<div>User</div>'
}

const router = new VueRouter({
  routes: [
    // 动态路径参数 以冒号开头
    { path: '/user/:id', component: User }
  ]
})
```

这样，像/user/foo和/user/bar都将映射到相同的路由。

一个“路径参数”使用冒号 :标记。当匹配到一个路由时，参数值会被设置到this.$route.params，可以在每个组件内使用。
在 User 的模板，输出当前用户的 ID：

```
const User = {
  template: '<div>User {{ $route.params.id }}</div>'
}
```

vue-router路由的两种模式

1. hash模式：vue-router 默认 hash 模式 —— 使用 URL 的 hash 来模拟一个完整的 URL，于是当 URL 改变时，页面不会重新加载。
2. history模式：如果不想要很丑的 hash，我们可以用路由的 history 模式，这种模式充分利用 history.pushState API 来完成 URL 跳转而无须重新加载页面。

```
//设置mode属性，设置路由模式
const router = new VueRouter({
  mode: 'history',
  routes: [...]
})
```

不过这种模式要玩好，还需要后台配置支持。因为我们的应用是个单页客户端应用.
所以呢，你要在服务端增加一个覆盖所有情况的候选资源：如果 URL 匹配不到任何静态资源，则应该返回同一个 index.html 页面，这个页面就是你 app 依赖的页面。

在vuex中使用异步修改，为什么要异步修改?

vuex里面创建一个actions方法

```
const actions ={
    addAction(context){
        context.commit('add',10)
    },
    reduceAction({commit}){
        commit('reduce')
    }
```

```
}
```

为什么需要actions管理异步
在vue早期版本中mutaition里面直接写异步代码会报错，但是新版本中就不会出现报错的问题。
官方给出的建议就是mutation中必须同步执行。虽然说现在的版本中在某些情况下在里面写异步代码不仅不会报错，有的时候还会正确执行，但是还是不推荐使用。
在实际开发过程中会遇到这样的问题，第二个异步操作的条件依赖第一个异步操作的结果。当遇到这种情况的时候我们肯定希望被依赖的数据在拿到结果之后在处理。所以这时候我们就需要使用actions来处理。

vue组件的scoped属性的作用

当 style 标签有scoped 属性时，它的 CSS 只作用于当前组件中的元素
你可以在一个组件中同时使用有 scoped 和 非scoped 样式

```
<style>
/* 全局样式 */
</style>

<style scoped>
/* 本地样式 */
</style>
```

但有时在 vue-cli 中引入 UI库 后，修改UI库一些样式，可能不生效。

vue如何优化首屏加载速度?

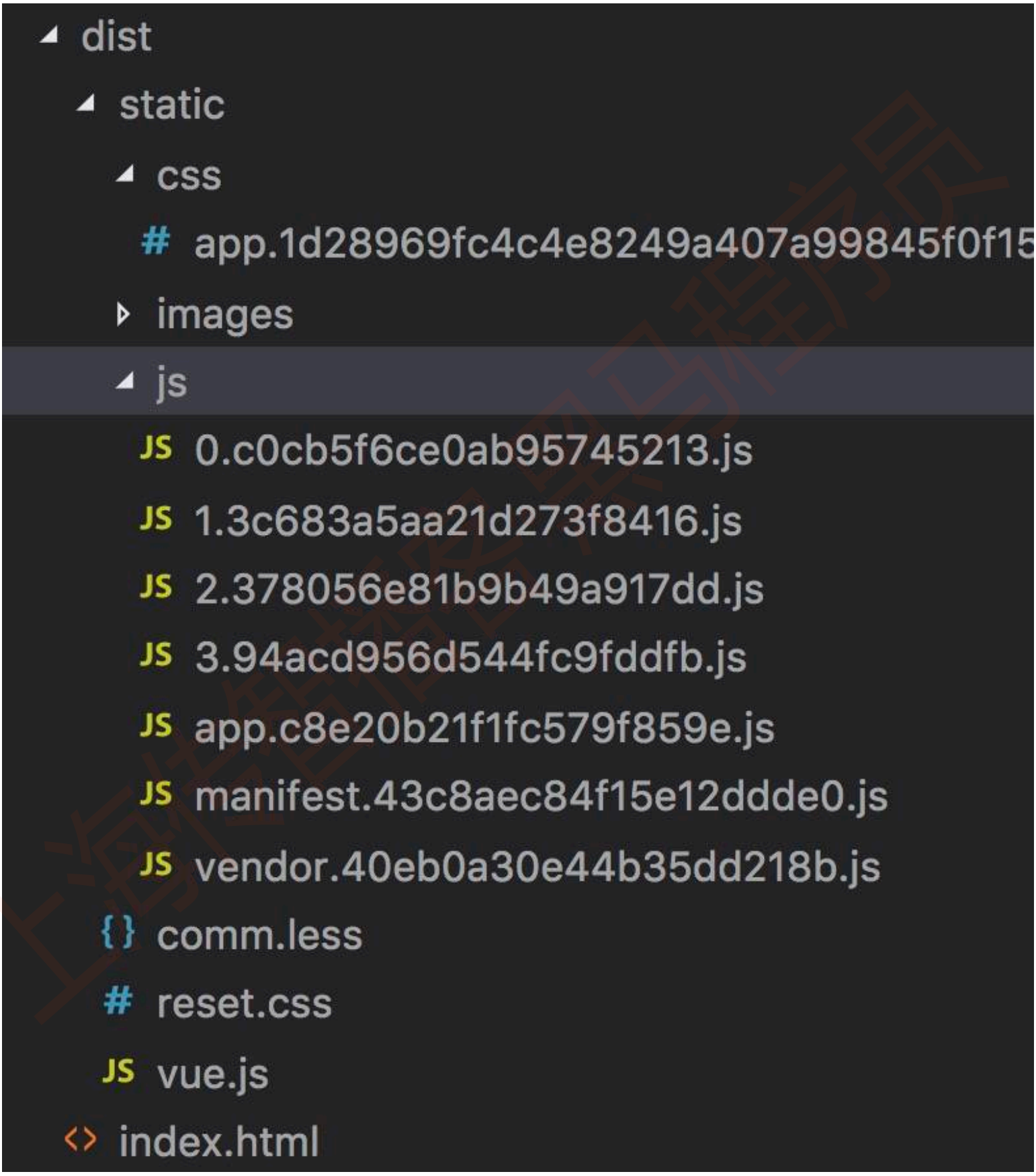
将引用的外部js、css文件剥离开来，不编译到vendor.js中，而是用资源的形式引用，这样浏览器可以使用多个线程异步将vendor.js、外部的js等加载下来，达到加速首开的目的。外部的库文件，可以使用CDN资源，或者别的服务器资源等。

1. 大文件定位：我们可以使用 webpack可视化插件Webpack Bundle Analyzer查看工程js文件大小，然后有目的的解决过大的js文件。
2. 路由的按需加载
3. 将JS文件引入方式放在 body 的最后
4. 用文档的cdn文件代替 npm 安装包
   用文档的cdn文件代替，而不用打包到vender里面去

```
<script src="https://cdn.bootcss.com/vue/2.3.3/vue.min.js"></script>
<script src="https://cdn.bootcss.com/axios/0.16.2/axios.min.js"></scrip
t>
```

1. UI库的按需加载：一般 UI库 都提供按需加载的方法，按照文档即可配置.
2. 开启 Gzip 压缩 在 config/index.js 设置 productionGzip 为 true，开启 Gzip 压缩

vue打包后会生成哪些文件?



生产index.html单页面文件
将组件中的 css 编译合并成一个 app.[hash].css 的文件，

js 则在合并后又拆解成 了 3 个文件：

- app.[hash].js 包含了所有 components 中的 js 代码
- vendor.[hash].js 包含了生产环境所有引用的 node_modules 中的代码
- mainfest.[hash].js 则包含了 webpack 运行环境及模块 化所需的 js 代码
- 0.[hash].js 则是 vue-router 使用了按需加载生产的js文件
  每块组件修改重新编译后不影响其他未修改的 js 文件 的 hash 值，这样能够最大限度地使用缓存，减少 HTTP 的请求数。

vue打包命令是什么?

npm run build
用于查看 vue-cli 生产环境部署资源文件大小的 npm命令
npm run build --report

观察者模式

观察者模式，它定义一种一对多的关系，让多个观察者对象同时监听某一主题对象，这个主题对象的状态发生变化的时后会通知所有的观察者对象，使他们能够自己更新自己

观察者模式的作用

1. 支持简单的广播通信，自动通知所有的订阅过的对象。
2. 页面载入完成之后，目标对象很容易于观察者之间建立一种动态的关联增加了灵活性。
3. 目标对象与观察者之间的抽象耦合关系能够单独扩展以及重用。

Vue.mixin的作用（混入）

混入 (mixins) 是一种分发 Vue 组件中可复用功能的非常灵活的方式。混入对象可以包含任意组件选项。当组件使用混入对象时，所有混入对象的选项将被混入该组件本身的选项。
组件内混入

```
var mixin = {
  methods: {
    foo: function () {
      console.log('foo')
    },
    conflicting: function () {
      console.log('from mixin')
```

```
    }
  }
}

var vm = new Vue({
  mixins: [mixin],
  methods: {
    bar: function () {
      console.log('bar')
    },
    conflicting: function () {
      console.log('from self')
    }
  }
})

vm.foo() // => "foo"
vm.bar() // => "bar"
vm.conflicting() // => "from self"
```

全局混入

```
// 为自定义的选项 'myOption' 注入一个处理器。
Vue.mixin({
  created: function () {
    var myOption = this.$options.myOption
    if (myOption) {
      console.log(myOption)
    }
  }
})

new Vue({
  myOption: 'hello!'
})
// => "hello!"
```

- mixins调用顺序

  从执行的先后顺序来说，混入对象的钩子将在组件自身钩子之前调用，如果遇到全局混入(Vue.mixin)，全局混入的执行顺序要前于混入和组件里的方法。

extends （单个组件继承，组件继承）

允许声明扩展另一个组件(可以是一个简单的选项对象或构造函数)，而无需使用Vue.extend。这主要是为了便于扩展单文件组件。
这和 mixins 类似。

```
var CompA = { ... }

// 在没有调用 `Vue.extend` 时候继承 CompA
var CompB = {
  extends: CompA,
  ...
}
```

- extends调用顺序
  > 执行的先后顺序和mixins一样，另外扩展的方法与原生的冲突时，扩展的方法不生效，这点跟混入一样。

mixins 和 extends的调用顺序

extends 顺序大于 mixins

extend

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

```
// 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'
    }
  }
})
```

单例组件，继承组件

- 继承组件

  组件继承 通过extends来实现的
  意思就是封装一个组件，然后将这个组件继承给其他组件。比方说

```
// 我封装了一个组件，组件里面包含一个比较常用的过滤器和路由返回
var CompA = {
    filters:{
        numAdd(n){
            return n+1
        }
    },
    methods:{
        butome(){
            this.$router.go(-1)
        }
    }
}

// 然后在其他组件里面继承这里面的方法
var CompB = {
  extends: CompA,
  ...
}
```

- 单例组件

在实际开发过程中经常遇到调用组件和销毁组件，一个经常被反复的销毁和创建。
单例组件就是用来解决这种问题，单例组件指：这个组件全局只有一个在调用，且永远只有一个，从创建到销毁都是这一个组件。

```
// 在App.vue 中创建一个组件
    <single-case></single-case>
// singleCase 组件中使用watch监听vuex中的变化 从而改变组件对应的数据
// 在其他地方要修改这个组件直接通过vuex来修改
```

vue的动态组件

- component

  vue 中提供了一个方法来实现动态组件，也称为元组件；
  依靠 is 的值，来决定哪个组件被渲染

```
<component :is="which_model"></component>
<button @click="switch('Game')">Game</button>
<button @click="switch('Class')">Class</button>
<button @click="switch('Question')">Question</button>
<button @click="switch('Status')">Status</button>
```

```
//引入组件
  import Class from '../Class/';
  import Question from '../Question/';
  import Game from '../Game/';
  import Status from '../status/';
   export default {
    name: "index",
    components: {
      Class,
      Game,
      Question,
      Status
    },
```

```
    data() {
      return {
        // 动态组件
        which_model: 'Game',

      }
    }
    methods:{
      // 点击切换组件
      switch(name){
        this.which_model = name;
      },

    }
  }
```

vue中如何编写可复用的组件?

在我们编写组件的时候一定要考虑组件的可复用性，可复用性越高，开发效率就越高。
可复用的组件主要提供了三个API prop、事件、slot

- prop 允许外部环境传递数据给组件，在vue-cli工程中也可以使用vuex等传递数据。
- 事件允许组件触发外部环境的 action
- slot 允许外部环境将内容插入到组件的视图结构内。

```
  <my-component
    :foo="bar"
    :bar="qux"
    //子组件调用父组件方法
    @event-a="doThis"
    @event-b="doThat">
    <!-- content -->
  <img slot="icon" src="..." />
  <p slot="main-text">Hello!</p>
 </my-component>
```

vue的组件传参

- 父组件给子组件传值

```
// 父组件
<template>
   <child v-bind:name="name"></child>
</template>
<script>
import child from "./child.vue"
  export default {
     components: {child},
     data(){
      return {name:"张三"}
     }
  }
</script>

// 子组件
<template>
  <div>{{name}}</div>
</template>
<script>
  export default {
  // 子组件定义props接收父组件传递的值
    props:["name"]
  }
</script>
```

- 子组件给父组件传值

```
<template>
  <child @childToParent="child"></child>
</template>
<script>
import child from "./child.vue"
  export default {
    components: {child},
    methods:{
```

```
      child(data){
        console.log(data);
      }
    }
  }
</script>
```

```
<template>
   <button @click="childClick">点击</button>
</template>
<script>
  export default {
    methods:{
      childClick(){
      // 使用$emit 方法调用子组件的方法如何破传递参数
        this.$emit("childToParent","I am child.");
      }
    }
  }
</script>
```

- 兄弟组件传参可以使用vuex

路由解耦

```
const User = {
  template: '<div>User {{ $route.params.id }}</div>'
}
const router = new VueRouter({
  routes: [
    { path: '/user/:id', component: User }
  ]
})

// 解耦
const User = {
```

```
  props: ['id'],
  template: '<div>User {{ id }}</div>'
}
const router = new VueRouter({
  routes: [
  // 使用props的方式将当前的id传递过去 ， 这样就不会太依赖于当前组件里面的路由参数了
    { path: '/user/:id', component: User, props: true },
  ]
})

<user :id="1"></user>
```

实现一个v-model

其实v-model只不过是一个语法糖而已,真正的实现靠的还是

- v-bind:绑定响应式数据
- 触发 input 事件 并传递数据 (重点)

```
<template>
  <div class="hello">
    <input type="text" v-myModel="num">
    <button @click="num++">点击</button>
    <p>{{num}}</p>
  </div>
</template>

<script>
  export default {
    name: 'HelloWorld',
    data() {
      return {
        num: 0
      }
    },
    directives: {
    // 创建一个myMode的自定义指令
```

```
      myModel: {
        // 指令的定义 el当前的组件本身 obj传递的
        bind: (el, obj ,Vnode ) => {
        // 将传递过来的值赋值给input标签
          el.value = obj.value
          // 给input标签添加键盘输入事件
          el.addEventListener('input' , function(){
          // 将用户输入的值给对应的属性
            Vnode.context[obj.expression] = this.value;
          })
        },
        update:(el , obj)=>{
        // 标签更新的时候修改标签的值
          el.value = obj.value;
        }
      }
    }
  }
</script>

<style scoped>

</style>
```

ssr 服务端渲染 什么是SSR

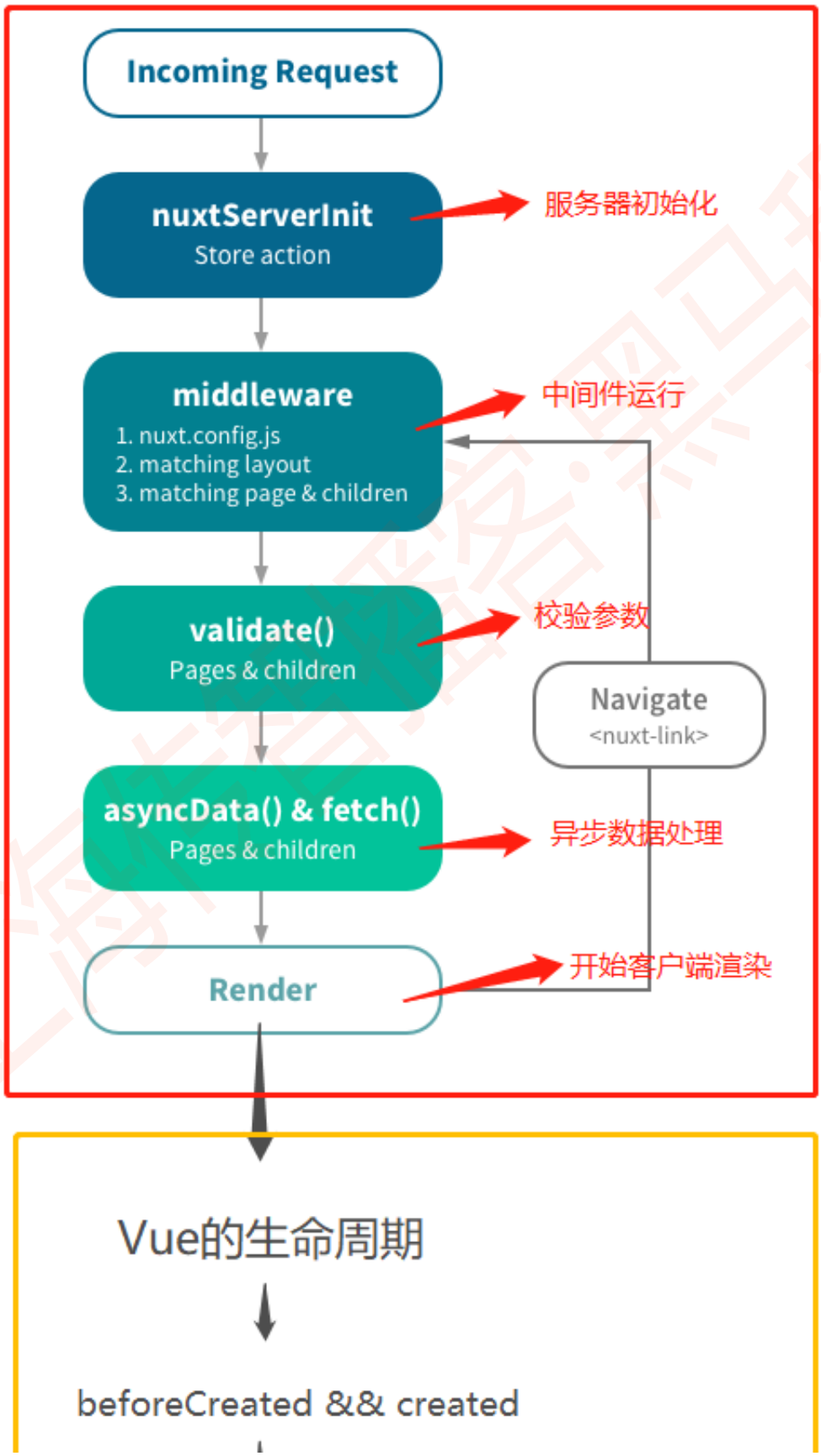
- Server Side Rendering（服务端渲染

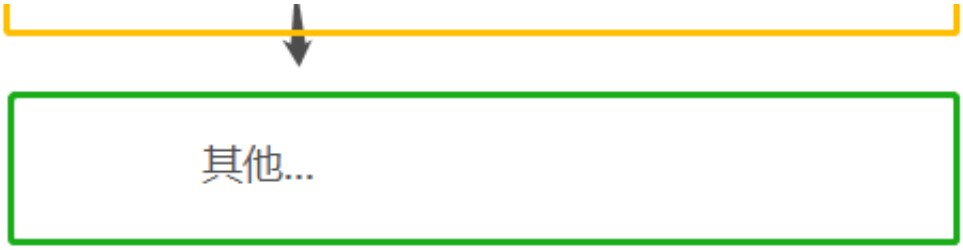
  SSR 目的是为了解决seo的问题的，对于一般的页面来说seo对于页面的影响不是很大，但是对于一些新闻，论坛类网站来说是致命的，因为框架类项目打包生成的页面是没有办法进行SEO的所以他们的关键信息没有办法暴露出去。
  SSR的原理主要就是将框架类型的页面交给后端来渲染，从而实现seo。
  主要适用场景

  - 客户端的网络比较慢
  - 客户端运行在老的或者直接没有 JavaScript 引擎上

nuxt

nuxt 是Vue的通用应用框架，是一个基于vue的SSR框架
他也有自己的生命周期





其中红色部分主要是服务端的生命周期
nuxt 的路由是根据页面自动生成的

yarn

yarn 和npm 差不多

1. yarn相比起npm，下载速度更加的块，他是通过多线程下载的。
2. yarn更加的安全
3. yarn更加稳定

```
// 初始化
yarn init
// 添加依赖
yarn add [package]
yarn add [package]@[version]
yarn add [package]@[tag]
// 升级
yarn upgrade [package]
yarn upgrade [package]@[version]
yarn upgrade [package]@[tag]
// 移除
yarn remove [package]
// 安装所有依赖
yarn
```

vue项目进行优化

- 使用npm run build --report命令进行大文件定位（Webpack Bundle Analyzer 插件）

- 路由的按需加载
- 将打包生成后 index.html页面 里面的JS文件引入方式放在 body 的最后
- 用文档的cdn文件代替 npm 安装包
- UI库的按需加载
- 开启 Gzip 压缩

vue-router 在什么阶段挂载上去的

vue-router 是在生命周期 beforeCreate 阶段挂载上去的

```
// vue-router 代码内部封装的代码
Vue.mixin({
  beforeCreate(){
    if(this.$options.router){
      this._router = this.$options.router;
      this._router.init(this);
    }
  }
})

// vue-router使用混入的方式在生命周期的beforeCreate阶段将router的调用代码启动
```

vuex 是在什么阶段挂载上去的

vuex 和 vue-router都是在 beforeCreate阶段挂载的

```
Vue.mixin({ beforeCreate: vuexInit });
```

怎么给动态的标签添加scoped

css 使用 >>>

```
<style scoped>
.a >>> .b { /* ... */ }
```

```
</style>

转换成
.a[data-v-f3f3eg9] .b { /* ... */ }
```

less 使用 /deep/

```
<style lang="less" scoped>
.searchforminline-out {
/deep/ input{
width: 50px;
}
</style>
```

vue页面中的定时器或者滚动事件报错

当vue项目中的组件被移除后，组件中的定时器和页面滚动事件并不会跟随组件一起移除掉，因此会导致出现一些错误，或影响其它组件的正常运行。 因此离开该页面需要移除这个监听的事件,destroyed在组件移除后执行。

vue-router 页面切换后保持在页面顶部而不是保持原先的滚动位置的办法

vue-router有提供一个方法scrollBehavior，它可以使切换到新路由时，想要页面滚到顶部，或者是保持原先的滚动位置，就像重新加载页面那样。 这个功能只在 HTML5 history 模式下可用。

```
const router = new VueRouter({
  routes: [...],
  scrollBehavior (to, from, savedPosition) {
    // return 期望滚动到哪个的位置
    if (savedPosition) {
        return savedPosition
    } else {
```

```
        return { x: 0, y: 0 }
      }
    }
  })
```

关于Vue背景图打包之后访问路径错误问题

检查config文件中的assetsPublicPath是否设置为'/'而不是'./' ,注意区分'/'为绝对路径，绝对路径从网站静态服务器根目录查询/static/img/图片，这样的路径就是正确的，如果设置为'./'，路径就变成了相对路径，相对路径会根据相对的文件目录来确定文件资源，会造成上面分析的问题

2.vue-cli创建的vue项目，会自带一个static目录，将出错图片放在该目录下面即可（正确解决办法） 查看vue-cli创建项目的webpack配置文件可以找到一个将static目录拷贝到dist目录中。根据对资源的规划不同，将需要打包的资源放在src/assets目录中，不需要打包的资源放入static目录中。

vue prefepch

vue和react为什么都要加key

vue和react都是caiyongdiff算法来对比新旧节点的，从而可以更新节点

key的主要作用

- 更加准确

可以在数据发生变化的时候快速找到对应的标签，减少性能的消耗。

- 更加快速

标签加上key之后 可以更加快速的找到修改的标签，而且并不需要重新遍历

vuex 是否会和双向绑定

在严格模式中使用Vuex，当用户输入时，v-model会试图直接修改属性值，但这个修改不是在mutation中修改的，所以会抛出一个错误。当需要在组件中使用vuex中的state时,有2种解决方案

- 在input中绑定value(vuex中的state)，然后监听input的change或者input事件，在事件回调中调用mutation修改state的值
- 使用带有setter的双向绑定计算属性。见以下例子（来自官方文档）

```
<input v-model="message">
computed: { message: { get () { return this.$store.state.obj.message },
 set (value) { this.$store.commit('updateMessage', value) } } }
```

Vue 的响应式原理中 Object.defineProperty 有什么缺陷？为什么在 Vue3.0 采用了 Proxy，抛弃了 Object.defineProperty

- Object.defineProperty无法监控到数组下标的变化，导致通过数组下标添加元素，不能实时响应；
- Object.defineProperty只能劫持对象的属性，从而需要对每个对象，每个属性进行遍历，如果，属性值是对象，还需要深度遍历。Proxy可以劫持整个对象，并返回一个新的对象。
- Proxy不仅可以代理对象，还可以代理数组。还可以代理动态增加的属性。

介绍下 webpack 热更新原理，是如何做到在不刷新浏览器的前提下更新页面

1. 当修改了一个或多个文件；
2. 文件系统接收更改并通知webpack；
3. webpack重新编译构建一个或多个模块，并通知HMR服务器进行更新；
4. HMR Server 使用webSocket通知HMR runtime 需要更新，HMR运行时通过HTTP请求更新jsonp；
5. HMR运行时替换更新中的模块，如果确定这些模块无法更新，则触发整个页面刷新。

首先，热模块更换（HMR）仍然是一个实验性功能。
HMR是一种在正在运行的应用程序中交换模块（以及添加/删除模块）的方法。您基本上可以更新已更改的模块而无需重新加载整页

vue 页面跳转不刷新的问题

在使用Vue-router做项目时，会遇到如/serviceId/:id这样只改变id号的场景。由于router-view是复用的，单纯的改变id号并不会刷新router-view，而这并不是我们所期望的结果
当然，我们可以在点击事件上加上router.go(0)，强制刷新整个页面来满足效果。但页面整体的刷新会使体验下降，并且作为个人也不是很能接受这样的方法。在查阅了一些资料后，发现可以有以下两种方法可以解决问题。

- 使用watch方法

```
watch: {
```

```
    '$route': function (to, from) {

        // 重新调用接口刷新数据
        this.newData()
    }
},
```

- 通过改变router-view中的key来达到刷新组件的目的，我现在用的就是这种方法（因为我使用的按需加载，所以加载组件也不会加载所有界面）

```
<router-view :key="activeDate"></router-view>
//我用了一个简单粗暴的方式来改变key，时间戳（捂脸）
//this.activeDate = new Date()
```

- 还有一种官方文档的方法
- 在组件内直接使用，前提是你用了vue-router:

```
beforeRouteLeave (to, from, next) {
    // 导航离开该组件的对应路由时调用
    // 可以访问组件实例 `this`
}
```

Vue组件

vue组件的分类

1. 在vue中v-router所产生的每一个页面本质上来说就是一个个组件，每一个组件主要承载了当前页面的样式和结构，其中也会包含一些数据处理，可视化和获取。整个页面的信息内容相对来说较为大，而且数据的传递仅仅就是一些简单的父子组件的信息较为少，而且复用性十分的差。

   在实际开发中，项目中的页面大多都是这样的，开发者根据业务的需求，多人协同开发。每一个开发者根据分配到的任务，去实现各自的功能，每一个路由相对来说依赖性较为小。

2. 独立组件，插件(swapper ,element)。在实际开发过程中我们经常需要用到一些插件，这一类插件主要是为了给我们提供方便，同时也可以给我们的项目减少问题的产生，减少开发成本。

3. 业务组件，这一类组件和独立组件很相似。当时还是有区别的。业务组件类似于在独立组件的基础上加装了业务逻辑，比方说请求数组的方式，根据业务的数据格式处理数据等，这种组件只在当前业务中才可以使用。

组件的三要素 prop event slot

prop 属性

定义了组件有哪些可以配置的属性，其中组件的核心功能也是使用这个来进行配置的。写通用组件时，props 最好用对象的写法，这样可以针对每个属性设置类型、默认值或自定义校验属性的值。

```
props: {
    // 基础类型检测 （`null` 意思是任何类型都可以）
    propA: Number,
    // 多种类型
    propB: [String, Number],
    // 必传且是字符串
    propC: {
      type: String,
      required: true
    },
    // 数字，有默认值
    propD: {
      type: Number,
      default: 100
    },
    // 数组 / 对象的默认值应当由一个工厂函数返回
    propE: {
      type: Object,
      default: function () {
        return { message: 'hello' }
      }
    },
    // 自定义验证函数
    propF: {
      validator: function (value) {
        return value > 10
```

```
      }
    }
  }
```

首先prop 传递数据是单向的，只用通过父级的修改数据才可以成功修改数据，并且还需要设置sync

```
    <button :myProp.sync="ID"><button>
```

slot 插槽

如果要给我们封装的组件添加内容我们就需要用到插槽。

```
// myConfig 组件
<div>
    <h2> 我是父组件 </h2>
    <slot>
        我是父组件的插槽
    </slot>
</div>

// myCom 页面
<div>
    <p> 这是我的页面 </p>
    <myConfig>
        <p> 我在插件里面 </p>
    </myConfig>
</div>

// 渲染结果
<div>
    <p> 这是我的页面 </p>
    <div>
        <h2> 我是父组件 </h2>
        <p> 我在插件里面 </p>
    </div>

</div>
```

具名插槽

元素可以用一个特殊的特性 name 来进一步配置如何分发内容。多个插槽可以有不同的名字。具名插槽将匹配内容片段中有对应 slot 特性的元素。其中也可以使用匿名插槽，作为找不到匹配的内容片段的备用插槽。

```
// myConfig 组件
<div>
```

```
    <h2> 我是父组件 </h2>
    <slot name="slot1">/slot>
    <slot></slot>
    <slot name="slot2"></slot>
</div>

// myCom 页面
<div>
    <p> 这是我的页面 </p>
    <myConfig>
        <template slot="slot1">
            <p> 我是slot1 </p>
        </template>
        <p> 我是一个p标签 </p>
        <p> 我是另外一个p标签 </p>
         <template slot="slot2">
            <p> 我是slot2 </p>
        </template>
    </myConfig>
</div>

// 渲染出来的页面

<div>
    <h2> 我是父组件 </h2>
    <p> 我是slot1 </p>
    <slot></slot>
    <p> 我是slot2 </slot>
</div>
```

封装一个组件

目录

```
- myLogin
  - index.js
  - myLogin.vue
```

main.js

```
// 加载插件
import myLogin from './assets/myLogin';
// 将插件挂载到Vue上
Vue.use(myLogin);
```

index.js

```
// 引入myLogin组件
import myLogin from './myLogin';

// 对外暴露install方法给Vue.use使用挂载到Vue上
export default {
  install(Vue) {
    // 在全局注册 myLogin 组件
    Vue.component('myLogin', myLogin);
  }
}
```

myLogin.vue

```
<template>
  <div class="my_login_box">
    <p>
      <span> 手机号： </span>

      <input type="text" ref="phone" v-model="phone">
      <!-- 写一个插槽，插槽名phone 插槽暴露数据phone 提供给slot-scope使用 -->
      <slot name="phone" :phone="myProps.phone"></slot>
    </p>
    <p>
      <span> 密码： </span>
      <input type="password" ref="password" v-model="word">
      <!-- 写一个插槽，插槽名word 插槽暴露数据word 提供给slot-scope使用 -->
      <slot name="word" :word="myProps.word"></slot>
    </p>
    <p>
      <button @click="submit"> 确定</button>
    </p>
  </div>
</template>

<script>
  export default {
    name: "myLogin",
    // 获取父组件传递过来的数据 phoneReg ，wordReg，prompt
    /**
     *
     * phoneReg 手机的正则匹配
     * wordReg 密码的正则匹配
     * prompt 提示信息
     * */
    props: {
      phoneReg: {
        // type 数据类型 ，default 默认值 ，不是原始数据类型需要使用函数return出来
        type: RegExp,
```

```
        default: () => /^1[34578]\d{9}$/
      },
      wordReg: {
        type: RegExp,
        default: () => /^[\w]{6,12}$/
      },
      prompt: {
        type: Object,
        default: () => {
          return {phone: '错误', word: '错误'}
        }
      }
    },
    data() {
      return {
        phone: '',
        word: '',
        myProps: {
          phone: '',
          word: ''
        }
      }
    },
    methods: {
      // 点击事件
      submit() {
        // 一上来清空提示
        this.myProps.phone = '';
        this.myProps.word = '';
        // 判断数据是否符合正则的需求
        if (!this.phoneReg.test(this.phone)) return this.myProps.phone = this.prompt.phone;
        if (!this.wordReg.test(this.word)) return this.myProps.word = this.prompt.word;
        // 信息判断正确之后调用父组件submit方法 ，并且将结果传递过去。
        this.$emit('submit', {phone: this.phone, word: this.word});
      }

    },
    // 一上来挂载 input 事件
    mounted() {
      // 获取this
      let _this = this;
      // 每一输入的时候触发事件，并且将当前的数据传递过去。
      this.$refs['phone'].addEventListener('input', function () {
        _this.$emit('phoneJudge', _this.phone)
      });

      this.$refs['password'].addEventListener('input', function () {

        _this.$emit('wordJudge', _this.word)
      });
```

```
    }
  }


</script>

<style scoped>
  .my_login_box {
    width: 300px;
    height: 450px;
    border: 1px solid #000;
  }
</style>
```

引用组件的页面

```
<template>
  <div class="hello">

    <!---  调用my-login 组件 ， 传入新的手机验证的正则  -->
    <!-- 添加点击事件 ， 二个输入框的的输入事件，同时传入新的提示信息 -->
    <my-login
      :phone-reg="/^[1][3,4,5,7,8][0-9]{9}$/"
      @submit="submit"
      @phoneJudge="phoneVerify"
      @wordJudge="wordVerify"
      :prompt="prompt"
    >
      <template slot-scope="props" slot="phone">
        <p>
          {{ props.phone }}
        </p>
      </template>
      <template slot-scope="props" slot="word">
        <p>
          {{ props.word }}
        </p>
      </template>

    </my-login>

  </div>
</template>

<script>


  export default {
    name: 'HelloWorld',
```

```
    data() {
      return {
        prompt: {phone: '你输入的手机号错误', word: '你输入的密码异常'}
      }
    },
    methods: {
      submit(val) {
        console.log(`用户点击确定了手机号:${val.phone};密码: ${val.word}`)
      },
      phoneVerify(val) {
        console.log(`用户正在操作手机号码输入框${val}`)
      },
      wordVerify(val) {
        console.log(`用户正在操作密码输入框${val}`)
      }
    }
  }
</script>

<!-- Add "scoped" attribute to limit CSS to this component only -->
<style scoped>

</style>
```

这个组件就是一个简单的登录组件。

vue组件的内置通讯

ref 给元素或者组件注册引用信息

```
// comA 组件
export default {
  data () {
    return {
      aData: 'xff'
    }
  },
  methods: {
    aFunc () {
      console.log('xff')
        }
  }
}

// comB 组件
<template>
  <com-a ref="classA"></com-a>
</template>
<script>
```

```
  export default {
    mounted () {
      const classA = this.$refs.classA;
      console.log(classA.aData);  // xff
     classA.aFunc();  // 打印xff
    }
  }
</script>
```

provide / inject 注入依赖

这对选项需要一起使用，以允许一个祖先组件向其所有子孙后代注入一个依赖，不论组件层次有多深，并在起上下游关系成立的时间里始终生效。
注入依赖可以作为vuex的一个替代方案，但是在官方建议中，注入依赖推荐在封装组件中使用，不推荐在业务中使用
provide 和 inject 绑定并不是可响应的。这是刻意为之的。然而，如果你传入了一个可监听的对象，那么其对象的属性还是可响应的。

```
<!--- app.vue -->
<template>
  <div>
    <router-view></router-view>
  </div>
</template>
<script>
  export default {
    provide () {
      return {
        app: this
      }
    },
    data(){
      return {
      userData:'123'
      }
    }
  }
</script>


<!--- 组件里面 --->
<template>
  <div>
    {{ app.userData }}
  </div>
</template>
<script>
  export default {
    inject: ['app']
  }
</script>
```

虽然可以获取到 app.vue 里面的userData 但是不是最新的，需要更新的时候，需要重新获取并且是异步获取

```
setTimout(()=> this.app.userData )
```

on与emit

emit会在当前组件实例上触发自定义事件，并且传递一些参数给监听器回调，一般来说都是在父级调用这个组件的时候，通过调用on的方式来监听自定义事件。

为什么要使用Vue构造器

常规的渲染组件的方式就是，在规定的地方渲染组件
如果组件的模板是从后台发过来的，需要动态的渲染组件。
需要实现一些提示框之类的页面，这种页面通常他的父级是在body里面的而不是在#app里面的。

Vue.extend 和 $mount 的使用

```
// 首先我们先要创建一个 构造器
    import Vue from 'vue'

    const MyCompunt = Vue.extend({
        template: '<span>{{ myText }}</span>',
        data(){
            return {
                myText:'this is Vue'
            }
        }
    })

// 创建构造器的实例
const myCompunt = new MyCompunt().$mount();

// 我们可以指定挂载的节点
new MyCompunt().$mount('#app');
// 不用 $mount，直接在创建实例时指定 el 选项
new MyCompunt( {el:'#app'} );
```

Render 函数

我们都知道vue核心是，虚拟DOM。
我们通常将我们的页面的结构逻辑都是写在template里面的，然后vue将我们的代码转换成虚拟DOM。
相比起DOM，虚拟DOM是通过js来处理的，所以消耗的性能相对来说很小。
Vue 推荐在绝大多数情况下使用 template 来创建你的 HTML。然而在一些场景中，你真的需要 JavaScript 的完全编程的能力，这时你可以用 render 函数，它比 template 更接近编译器。

Render 函数 的作用就是将template里面的结构转换成Vnode

```
    export default{
        data(){
            return {
            }
        },

        render:(createElement) => {
            return createElement(
// {String | Object | Function}
// 一个 HTML 标签字符串，组件选项对象，或者
// 解析上述任何一种的一个 async 异步函数。必需参数。
                'div',
// {Object}
// 一个包含模板相关属性的数据对象
// 你可以在 template 中使用这些特性。可选参数。
                {
                props: {
                    level: {
                      type: Number,
                      required: true
                    }
                  },
                   class: {
                        container: true
                    },
                    style: {
                      color: 'red'
                    }
                },
                // {String | Array}
  // 子虚拟节点 (VNodes)，由 `createElement()` 构建而成，
  // 也可以使用字符串来生成“文本虚拟节点”。可选参数。
  [
    '先写一些文字',
    createElement('h1', '一则头条'),
    createElement(MyComponent, {
      props: {
        someProp: 'foobar'
      }
    })
  ]

            )
        }
    }
```

template 和render的区别

```
// template 的形式
    <template>
    <div class="box">
      <p style="font-size:16px"> 你好 </p>
      <span v-if="show" > Vue </span>
      <span v-else > 地球 </span>
    </div>
</template>

<script>
    export default {
        name: "test",
        data(){
          return {
            show:true
          }
        }
    }
</script>


// render 形式
<script>
    export default {
        name: "test",
        data(){
          return {
            show:true
          }
        },
      render(createElement) {

    // <!--创建一个p标签-->
        let pNode = createElement('p' , {
          domProps: {
            innerHTML: '你好'
          },
          style:{
            'font-size':'16px'
          }
        });
        let hNode;
    // <!--创建一个h1标签-->
        if( this.show ) hNode = createElement('h1' , 'Vue');
        else hNode = createElement('h2' , '地球');
        // <!--返回div-->
        return createElement('div' , {
          class:'box'
        },[pNode , hNode])
      }
```

```
    }
</script>
```

递归组件

递归组件就是组件里面自己调用自己

```
<template>
    <ul>
        <my-ul></my-ul>
    </ul>
</template>
<script>
    export default {
        name: 'myUl'
    }
</script>
```

一般我们引入组件的时候都是import myUl from 'myUl',然后通过components注册到组件上。
但是如果没有结束条件的情况直接递归的话会报错的，所以针对递归组件我们需要加上结束条件

```
<template>
    <ul>
        <my-ul :add="add + 1 " v-if=" add<3 "></my-ul>
    </ul>
</template>
<script>
    export default {
        name: 'myUl',
        // :add 不可以加.sync 否则就会无法递归，因为所有的数据都会更新
         props: {
          add: {
            type: Number,
            default: 1
          }
        }
    }
</script>
```

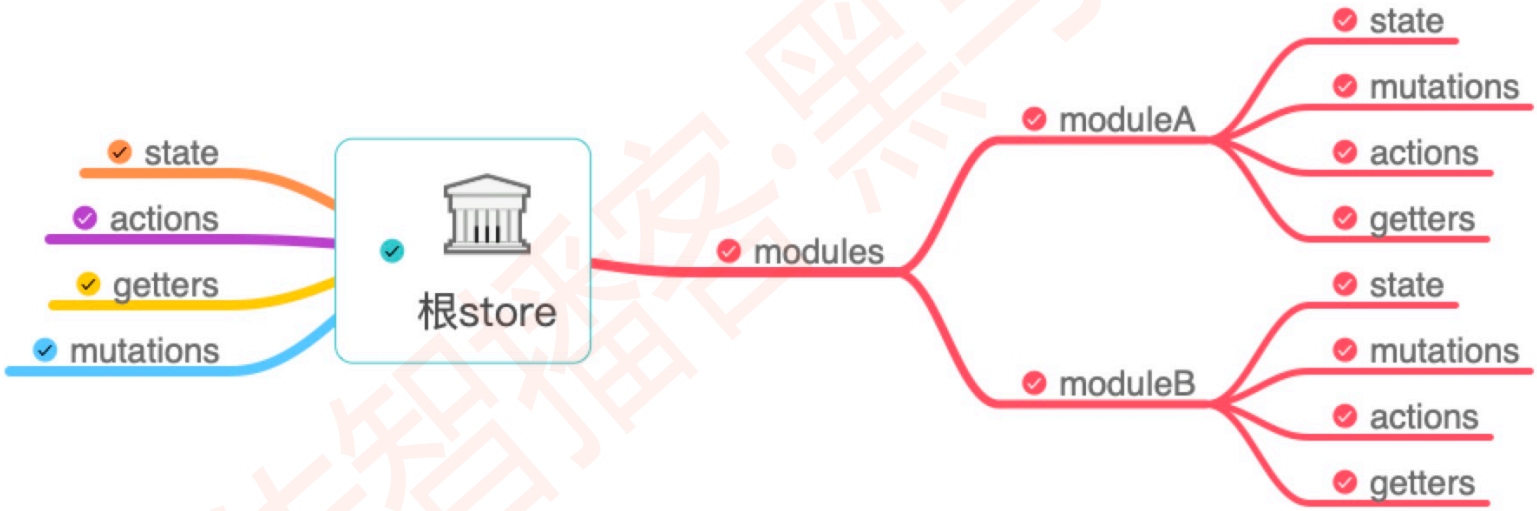
vue动态添加对象类型的值

Vue 不能检测对象属性的添加或删除

```
// 对象
export default {
  data () {
    return {
```

```
      obj: {
        name: 'Vue'
      }
    }
  },
  methods: {
    handler () {
      this.obj.name2 = 2;  // 不是响应性的
      this.$set(this.obj , 'name2' , 2) // 响应式
    }
  }
}
```

vuex 模块化的理解



我们按照模块化大方式设计vuex的时候，其实vuex将每一个模块都挂载到modules上面了，然后里面挂载了我设计的所有的模块

Vue面试题源码问题

Vue的响应式原理(暂定)

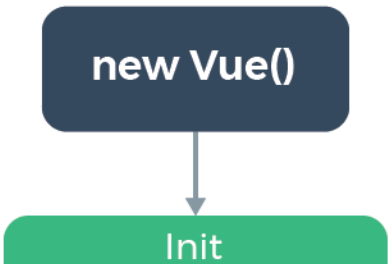
vue的响应式原理是数据劫持结合发布者，订阅者的方式来实现的，通过object.defineProperty来劫持各个属性的setter和getter，在数据发生变化的时候发布消息给订阅者，触发响应的监听回调。

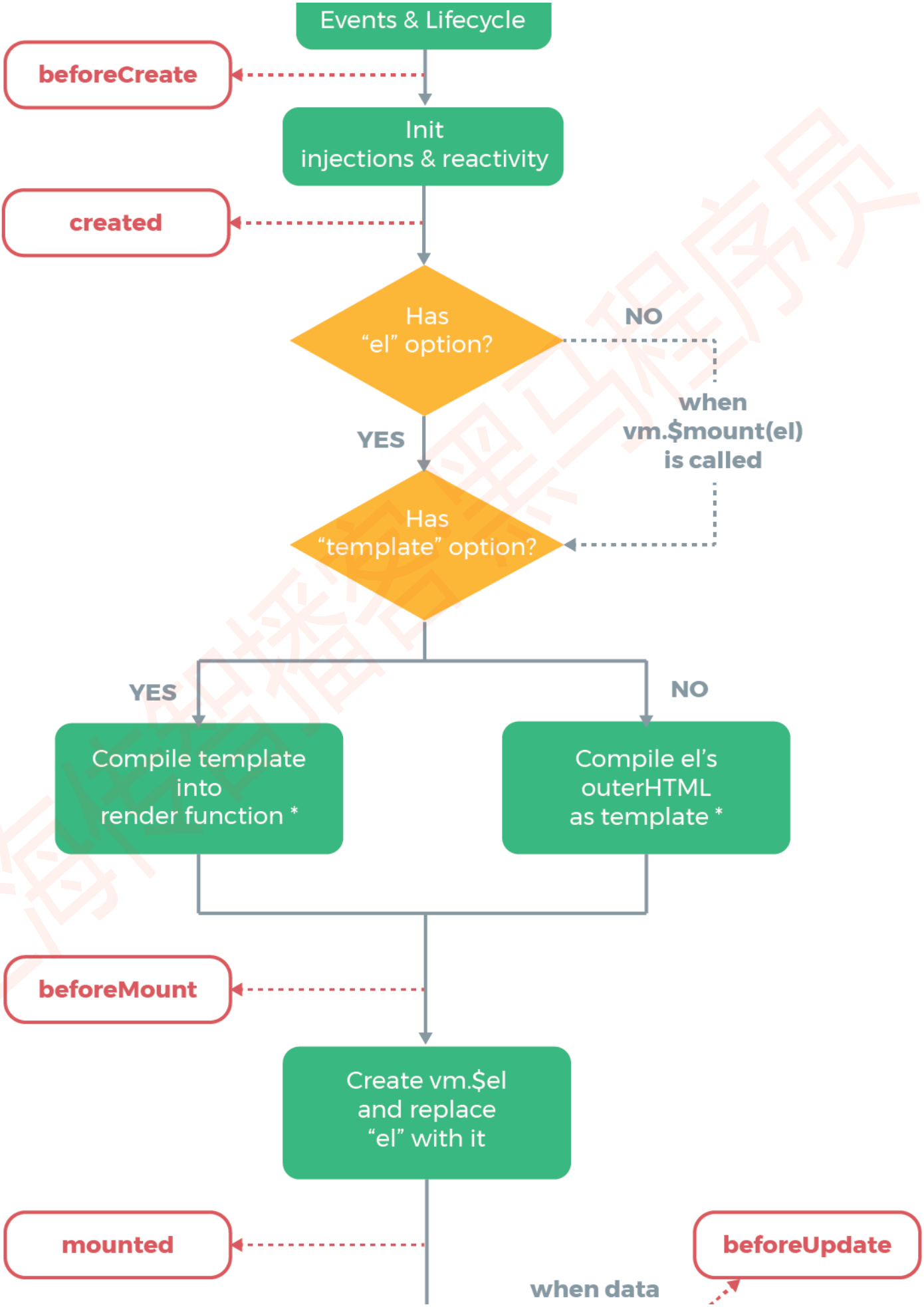
- 数据监听器Observer，能够对数据对象的所有属性进行监听，如有变动可拿到最新值并通知订阅者(Dep)
- 指令解析器Compile，对每个元素节点的指令进行扫描和解析，根据指令模板替换数据，以及绑定相应的更新函数
- Watcher，作为连接Observer和Compile的桥梁，能够订阅并收到每个属性变动的通知，执行指令绑定的相应回调函数，从而更新视图

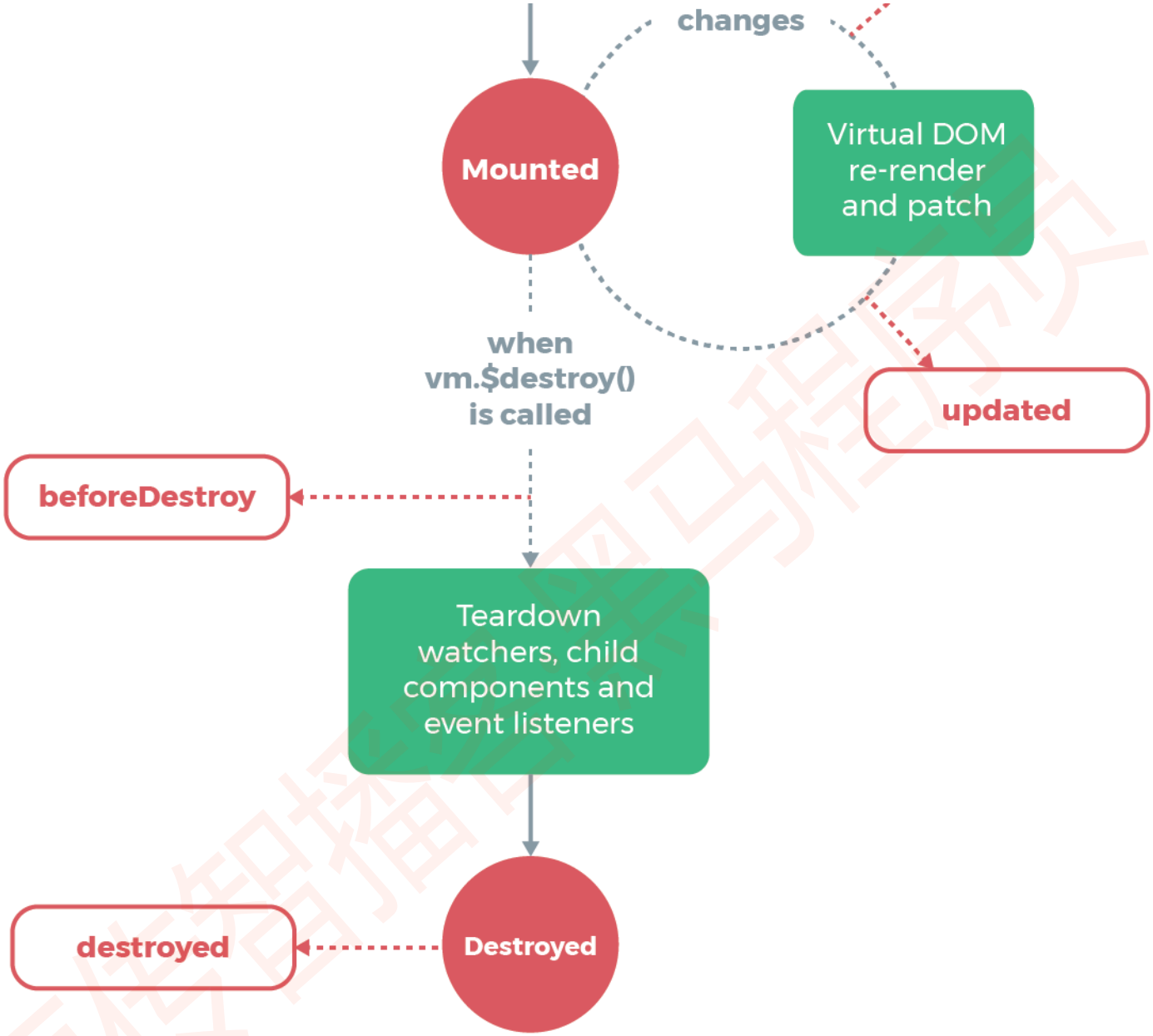
请详细说下你对vue生命周期的理解

vue 的生命周期是： vue 实例从创建到销毁，也就是从开始创建、初始化数据、编译模板、挂载Dom→渲染、更新→渲染、卸载等一系列过程。
在这个过程中也会运行一些叫做生命周期钩子的函数，这给了使用者在不同阶段添加自己的代码的机会。

生命周期钩子函数（11个）	详细
beforeCreate	在实例初始化之后，数据观测 (data observer) 和 event/watcher 事件配置之前被调用。
created	在实例创建完成后被立即调用。在这一步，实例已完成以下的配置：数据观测 (data observer)， 属性和方法的运算，watch/event 事件回调。然而，挂载阶段还没开始，$el 属性目前不可见。
beforeMount	在挂载开始之前被调用：相关的 render 函数首次被调用。
mounted	el 被新创建的 vm. *el*替换，并挂载到实例上去之后调用该钩子。如果*root*实例挂载了一个文档内元素，当*mounted*被调用时*vm.*el 也在文档内。
beforeUpdate	数据更新时调用，发生在虚拟 DOM 打补丁之前。这里适合在更新之前访问现有的 DOM，比如手动移除已添加的事件监听器。该钩子在服务器端渲染期间不被调用，因为只有初次渲染会在服务端进行。
updated	由于数据更改导致的虚拟 DOM 重新渲染和打补丁，在这之后会调用该钩子。
activated	keep-alive 组件激活时调用。该钩子在服务器端渲染期间不被调用。
deactivated	keep-alive 组件停用时调用。该钩子在服务器端渲染期间不被调用。
beforeDestroy	实例销毁之前调用。在这一步，实例仍然完全可用。该钩子在服务器端渲染期间不被调用。
destroyed	Vue 实例销毁后调用。调用后，Vue 实例指示的所有东西都会解绑定，所有的事件监听器会被移除，所有的子实例也会被销毁。该钩子在服务器端渲染期间不被调用。
errorCaptured（2.5.0+ 新增）	当捕获一个来自子孙组件的错误时被调用。此钩子会收到三个参数：错误对象、发生错误的组件实例以及一个包含错误来源信息的字符串。此钩子可以返回 false 以阻止该错误继续向上传播。




Events & Lifecycle
beforeCreate
Init
injections & reactivity
created
Has
“el” option?
NO
when
vm.$mount(el)
is called
YES
Has
“template” option?
YES
NO
Compile template
into
render function *
Compile el’s
outerHTML
as template *
beforeMount
Create vm.$el
and replace
“el” with it
mounted
beforeUpdate
when data



* template compilation is performed ahead-of-time if using a build step, e.g. single-file components

Vue 的 template 是如何编译成真正的 HTML 并做到双向绑定等等特殊功能的呢?

Vue 2.0 模板渲染过程

1. 首先第一步实例化一个vue项目
2. 模板编译是在vue生命周期的mount阶段进行的
3. 在mount阶段的时候执行了compile方法将template里面的内容转化成真正的html代码
4. parse阶段是将html转化成ast(抽象语法树)，用来表示代码的数据结构。在 Vue 中我把它理解为嵌套的、携带标签名、属性和父子关系的 JS 对象，以树来表现 DOM 结构。

```
    html: "<div id="test">texttext</div>"
    //html转换成ast
    ast: {
    // 标签类型
    type: 1,
    // 标签名
    tag: "div",
    // 标签行内属性列表
    attrsList: [{name: "id", value: "test"}],
    // 标签行内属性
    attrsMap: {id: "test"},
    // 标签关系 父亲
```

```
    parent: undefined,
    // 字标签属性列表
    children: [{
        type: 3,
        text: 'texttext'
      }
    ],
    plain: true,
    attrs: [{name: "id", value: "'test'"}],
    // 标签的标识符
    key: 1
  }
```

1. optimize 会对parse生成的ast树进行静态资源优化(静态内容指的是和数据没有关系，不需要每次都刷新的内容)
2. generate 函数，会将每一个ast节点创建一个内部调用的方法等待后面的调用。

```
<template>
  <div id="test">
    {{val}}
    <img src="http://xx.jpg">
  </div>
</template>

//最后输出
{render: "with(this){return _c('div',{attrs:{"id":"test"}},[[_v(_s(val))]),_v(" "),_m(0)])}"}
```

1. 在complie过程结束之后会生成一个render字符串，接下来就是 new watcher这个时候会对绑定的数据执行监听，render 函数就是数据监听的回调所调用的，其结果便是重新生成 vnode。当这个 render 函数字符串在第一次 mount、或者绑定的数据更新的时候，都会被调用，生成 Vnode。如果是数据的更新，那么 Vnode 会与数据改变之前的 Vnode 做 diff，对内容做改动之后，就会更新到我们真正的 DOM 上啦

你之前封装过过滤器吗？怎么封装的？

在vue-cli脚手架里面的src文件下面创建一个filter文件夹，里面在创建一个index.js文件

```
.
├── src
|   ├── filter
|   |    ├── index.js
|   |
|   └── main.js
└── ...
```

然后在index里面写入过滤器方法

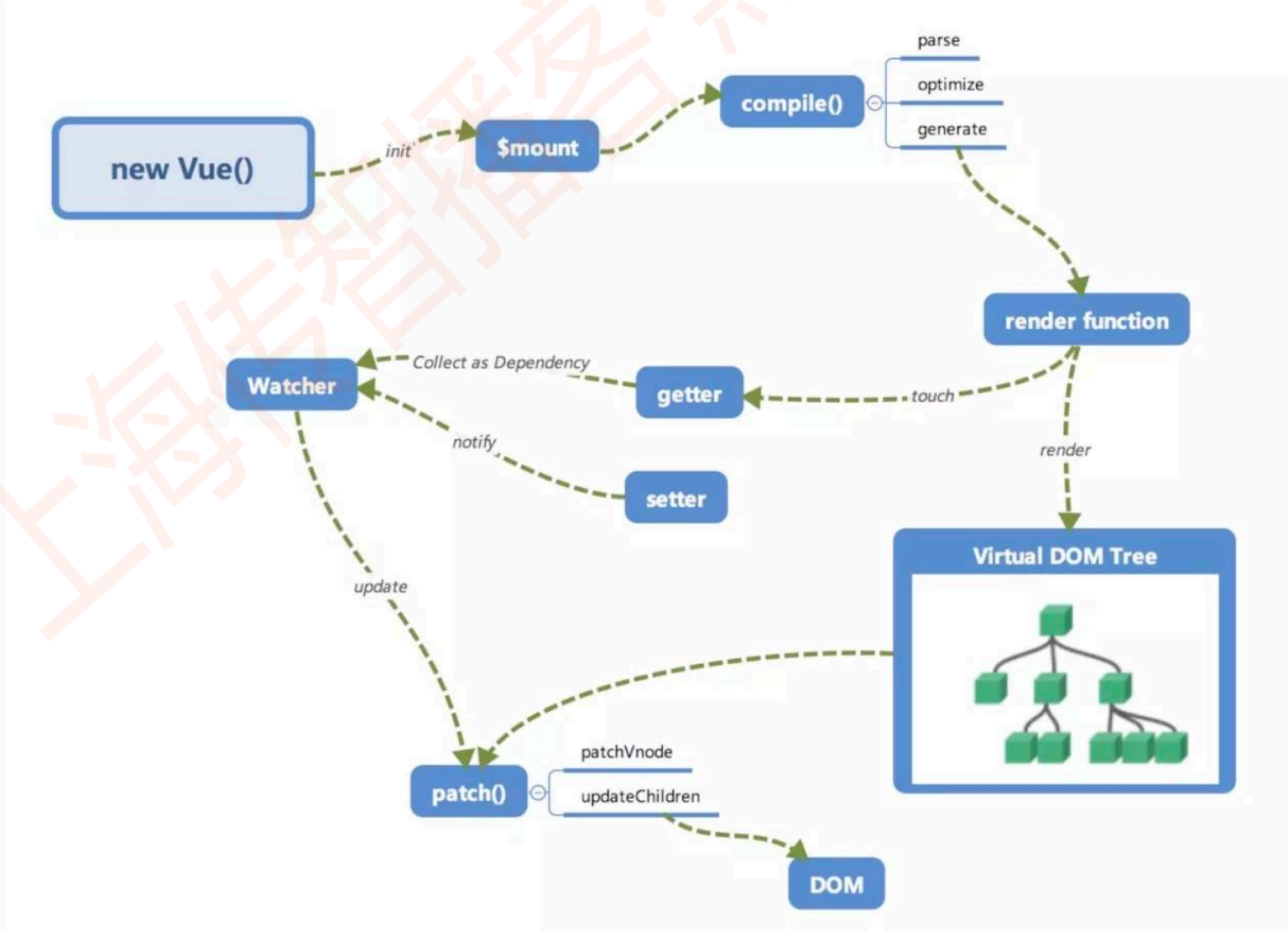
```
function filterOne(n){
    return n + 10;
}
function filterTwo(n){
    return n + 5;
```

```
}
export{
    filterOne,
    filterTwo
}
```

使用

```
// 找 filter/index.js
import * as filters from './filter/index.js'
//遍历所有导出的过滤器并添加到全局过滤器
Object.keys(filters).forEach((key) => {
  Vue.filter(key, filters[key]);
})
// 这样就可以在全局使用过滤器了。
// {{test | filterOne}}
```

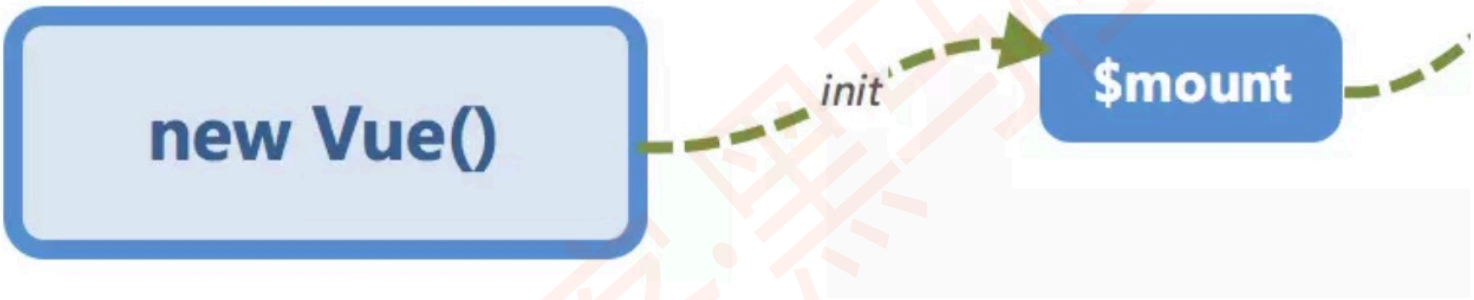
Vue.js的全局运行机制



流程分析：

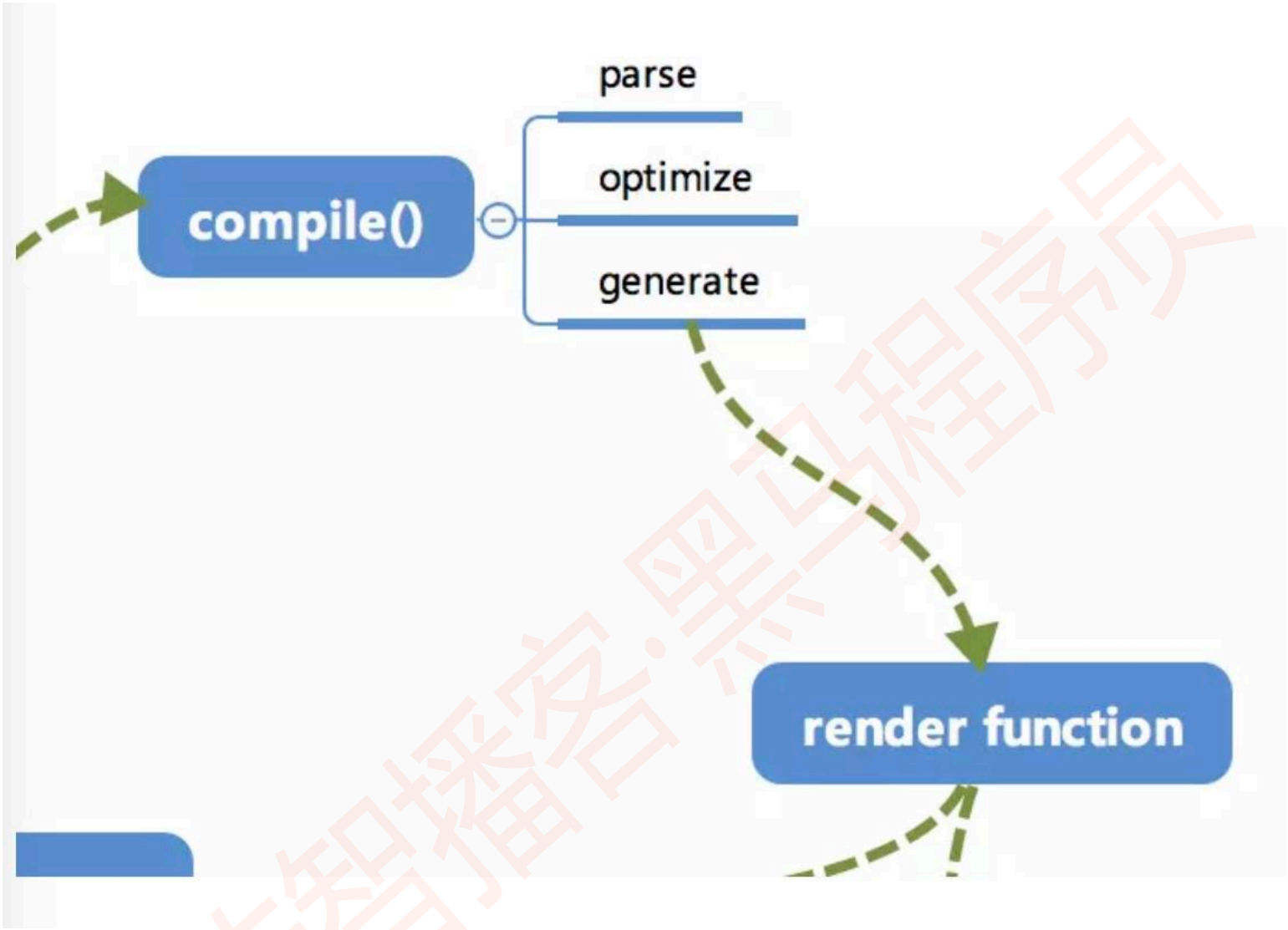
1. 先进行初始化及挂载：init以及mount
2. 再进行模板编译：compile，编译成渲染函数render function，参考链接：渲染函数&jsx
3. 再进行响应式依赖收集：render function => getter、setter => Watcher 进行update => patch 的过程以及使用队列来异步更新的策略。
4. 依赖收集的同时生产Virtual DOM：render function 被转化成 VNode 节点
5. 通过diff算法后进行patch更新视图

第一步：初始化及挂载



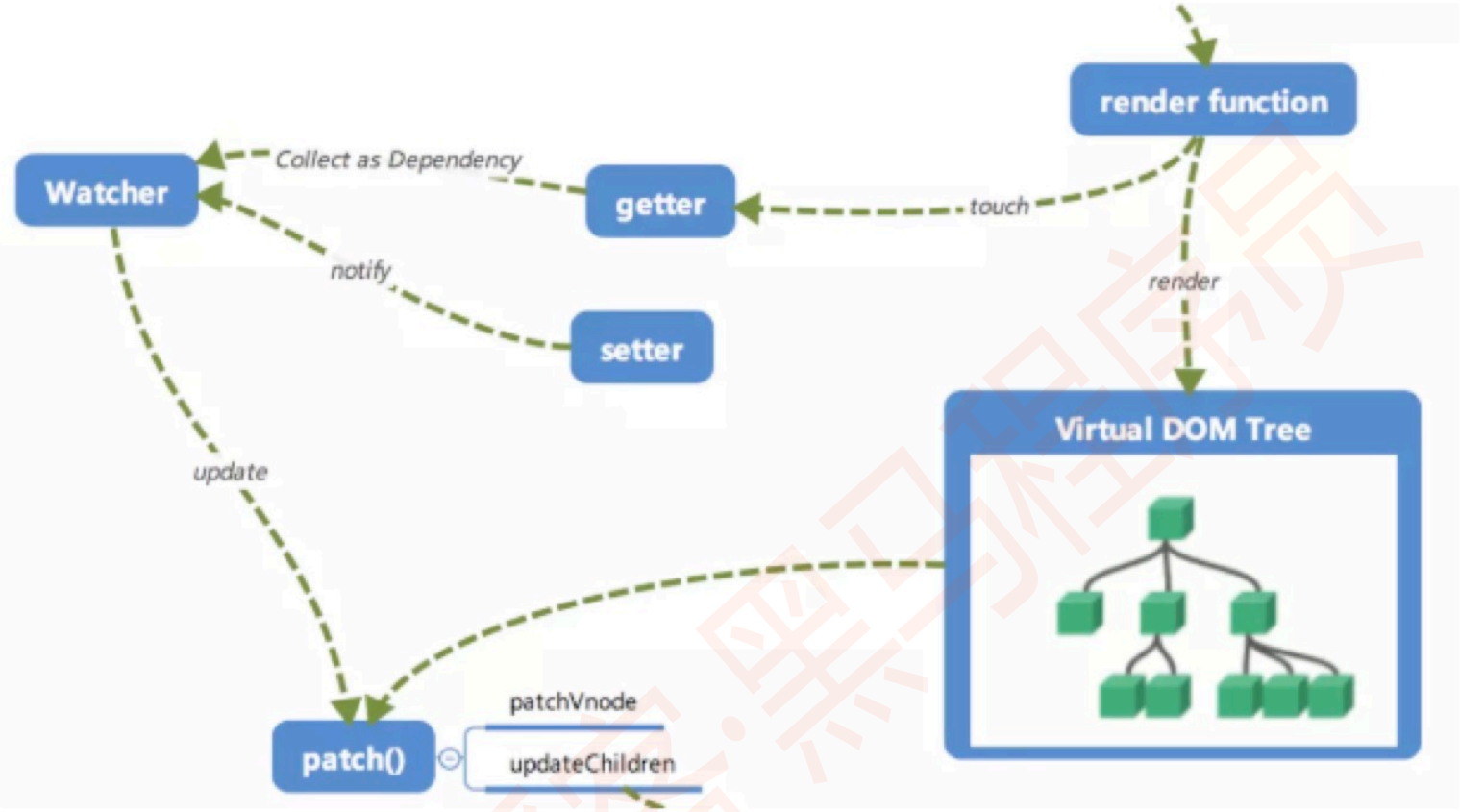
在 new Vue() 之后，Vue会调用_init内部方法进行初始化。他会初始化，生命周期，事件，props，methods、 data、 computed 与 watch 等。
其中最重要的部分就是Object.defineProperty设置setter 与 getter 函数用来实现「响应式」以及「依赖收集」

第二步：编译



Vue源码中虚拟DOM构建经历 template编译成AST语法树 -> 再转换为render函数 最终返回一个VNode(VNode就是Vue的虚拟DOM节点)
compile编译可以分成 parse、optimize 与 generate 三个阶段，最终需要得到 render function。

第三步：响应式



在init的时候通过Object.defineProperty 进行了绑定，它使的被设置的对象读取的时候触发getter方法，赋值的时候触发setter方法。
当 render function 被渲染的时候，因为会获取所需的值，所以会触发getter函数进行依赖收集
依赖收集的目的是将观察者watcher对象存放到当前闭包的订阅者dep的subs中，从而生成如下关系

修改对象值得时候会触发对应的setter， setter会通知之前的依赖收集得到的dep中每一个watcher，告诉它们自己的值改变了需要重新渲染视图。
这个时候watcher就开始调用update来更新视图，这中间还有patch和队列异步更新的策略。

第四步： Virtual DOM

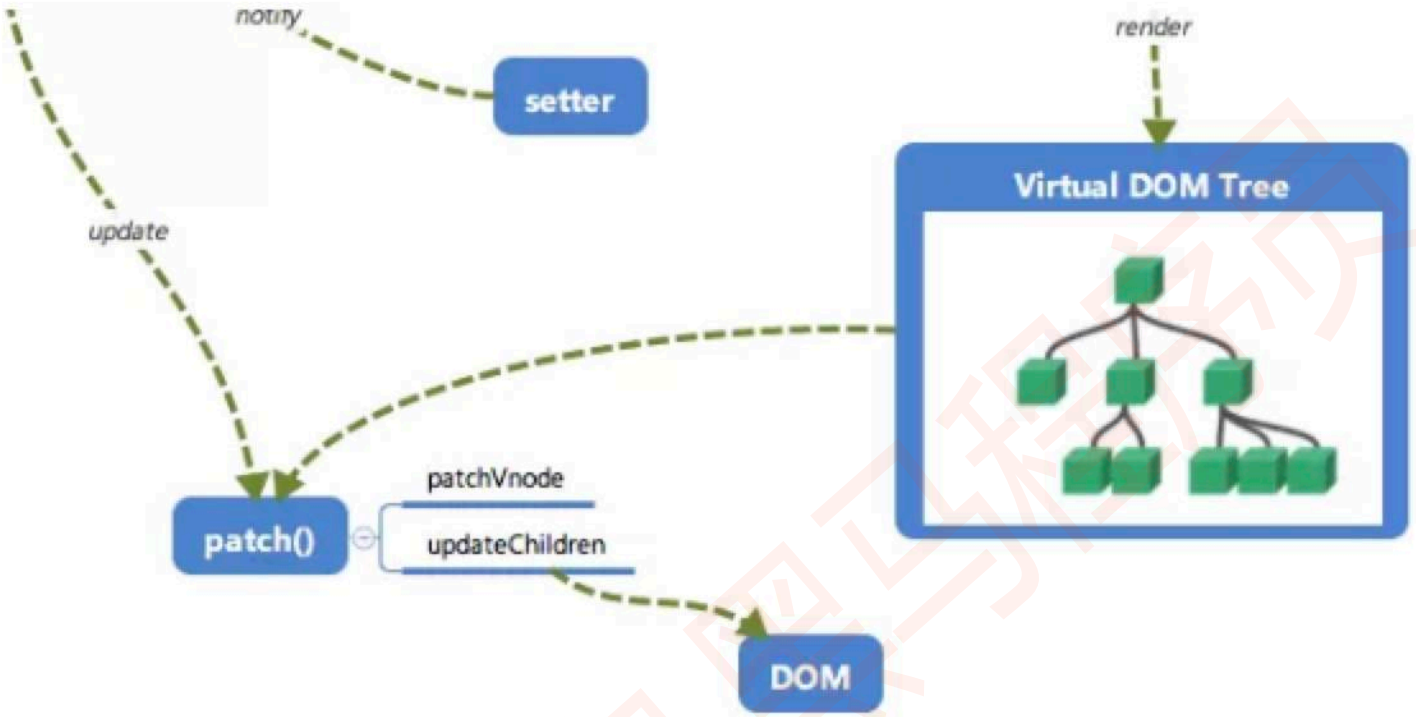
render function阶段的时候template会被转化成VNode节点，Virtual DOM其实就是一个以JavaScript对象作为基础的树（VNode节点）,其中用对象的属性来描述各个节点。实际上它就是一个对真实DOM的一种抽象，最后在通过各种操作将这个对象渲染到真实环境上。
由于Virtual DOM 是以js对象为基础的而不依赖于真实环境的所以他具有很好的跨平台性质。

```
{
    // 标签类型
    type: 1,
    // 标签名
    tag: "div",
    // 标签行内属性列表
    attrsList: [{name: "id", value: "test"}],
    // 标签行内属性
    attrsMap: {id: "test"},
    // 标签关系 父亲
    parent: undefined,
    // 字标签属性列表
    children: [{
        type: 3,
        text: 'texttext'
      }
    ],
    plain: true,
    attrs: [{name: "id", value: "'test'"}],
    // 标签的标识符
    key: 1
}
```

渲染后就可以得到

```
<div>
    <a>click me</a>
</div>
```

第五步：更新视图



修改一个对象的值得时候流程是 setter -> watcher -> update 的流程来精选修改的
更新视图的时候， 当数据发生变化的时候 执行render function就可以得到一个新的VNode节点。
接下来就是patch阶段，他会将新的数据和久的数据放到patch中去，然后进行diff算法，求出他们的差异值，最后只针对那一部分差异值做出修改。

vue插件安装

vue 中提供了一个Vue.use的方法安装插件.

```
vue.use(vuex)
```

插件安装的时候

```
// 简化
// 创建use方法 传入参数 插件 函数（对象也可以处理，需要看原版代码，因为原理大致相同，不做解释）
Vue.use = function (plugin: Function) {

    if (typeof plugin.install === 'function') {
    // 判断传入的插件是否提供了install方法，如果提供了方法就证明这个插件是符合vue规范,将this指向plugin，并且传入args(vue参数和方法/就代表vue)
      plugin.install.apply(plugin, args)
    } else if (typeof plugin === 'function') {
    // 没有传入参数判断是否是一个方法，是则调用它。
      plugin.apply(null, args)
    }
    return this
  }
```

vuex的内部运行机制

首先第一步安装插件

```
Vue.use(vuex)
```

vuex中遵循vue的规则提供了install方法

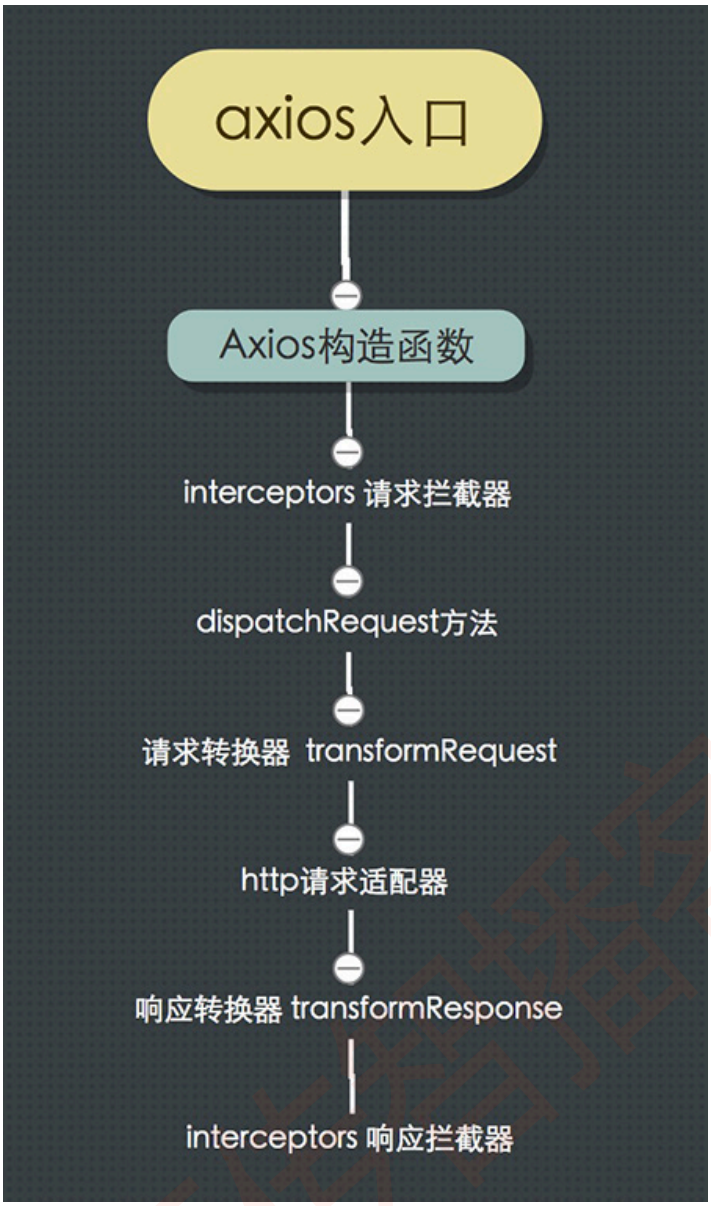
```
let Vue;
export function install (_Vue) {
    // 将vue参数和方法保存起来
     Vue = _Vue
    // 使用vue.mixin方法将vuexInit混淆进beforeCreate钩子中去
     Vue.mixin({ beforeCreate: vuexInit })
}
```

vue.mixin 混淆进beforeCreate钩子之后，每一个vm的实例都会去调用vuexInit方法。

```
  function vuexInit () {
    // this 指向vue this.$options指调用初始化Vue的时候传入的对象;
    const options = this.$options
    // 判断对象里面是否传入了store ， 如果传入了证明store是根节点直接将options.store 赋值给 this.$store。否则则说明不是根节点，从父节点的 $store中获取。
    if (options.store) {
    // 如果options.store是函数，调用函数返回结果。
      this.$store = typeof options.store === 'function'
        ? options.store()
        : options.store
    } else if (options.parent && options.parent.$store) {
      this.$store = options.parent.$store
    }
  }
```

axios的内部运行机制（暂定）

axios 是一个基于 Promise 的http请求库，可以用在浏览器和node.js中运行。



什么是 Virtual DOM?

一个真正的DOM元素是十分庞大的

```
> var div = document.createElement('div')
  var str = ''
  for (var key in div){
    str += key + ' '
  }
< "align title lang translate dir dataset hidden tabIndex accessKey draggable spellcheck contentEditable isContentEditable
  offsetParent offsetTop offsetLeft offsetWidth offsetHeight style innerText outerText webkitdropzone onabort onblur
  oncancel oncanplay oncanplaythrough onchange onclick onclose oncontextmenu oncuechange ondblclick ondrag ondragend
  ondragenter ondragleave ondragover ondragstart ondrop ondurationchange onemptied onended onerror onfocus oninput
  oninvalid onkeydown onkeypress onkeyup onload onloadeddata onloadedmetadata onloadstart onmousedown onmouseenter
  onmouseleave onmousemove onmouseout onmouseover onmouseup onmousewheel onpause onplay onplaying onprogress onratechange
  onreset onresize onscroll onseeked onseeking onselect onshow onstalled onsubmit onsuspend ontimeupdate ontoggle
  onvolumechange onwaiting click focus blur onautocomplete onautocompleteerror namespaceURI prefix localName tagName id
  className classList attributes innerHTML outerHTML shadowRoot scrollTop scrollLeft scrollWidth scrollHeight clientTop
  clientLeft clientWidth clientHeight onbeforecopy onbeforecut onbeforepaste oncopy oncut onpaste onsearch onselectstart
  onwheel onwebkitfullscreenchange onwebkitfullscreenerror previousElementSibling nextElementSibling children
  firstElementChild lastElementChild childElementCount hasAttributes getAttribute getAttributeNS setAttribute
  setAttributeNS removeAttribute removeAttributeNS hasAttribute hasAttributeNS getAttributeNode getAttributeNodeNS
  setAttributeNode setAttributeNodeNS removeAttributeNode closest matches webkitMatchesSelector getElementsByTagName
  getElementsByTagNameNS getElementsByClassName insertAdjacentElement insertAdjacentText insertAdjacentHTML
  createShadowRoot getDestinationInsertionPoints requestPointerLock getClientRects getBoundingClientRect scrollIntoView
  scrollIntoViewIfNeeded animate remove webkitRequestFullScreen webkitRequestFullscreen querySelector querySelectorAll
  ELEMENT_NODE ATTRIBUTE_NODE TEXT_NODE CDATA_SECTION_NODE ENTITY_REFERENCE_NODE ENTITY_NODE PROCESSING_INSTRUCTION_NODE
  COMMENT_NODE DOCUMENT_NODE DOCUMENT_TYPE_NODE DOCUMENT_FRAGMENT_NODE NOTATION_NODE DOCUMENT_POSITION_DISCONNECTED
  DOCUMENT_POSITION_PRECEDING DOCUMENT_POSITION_FOLLOWING DOCUMENT_POSITION_CONTAINS DOCUMENT_POSITION_CONTAINED_BY
  DOCUMENT_POSITION_IMPLEMENTATION_SPECIFIC nodeType nodeName baseURI isConnected ownerDocument parentNode parentElement
  childNodes firstChild lastChild previousSibling nextSibling nodeValue textContent hasChildNodes normalize cloneNode
  isEqualNode isSameNode compareDocumentPosition contains lookupPrefix lookupNamespaceURI isDefaultNamespace insertBefore
  appendChild replaceChild removeChild addEventListener removeEventListener dispatchEvent "
```

因为浏览器的标准就把 DOM 设计的非常复杂。当我们频繁的去做 DOM 更新，会产生一定的性能问题。
然而虚拟DOM就是使用js对象去抽象的描述一个DOM元素，它创建的元素相对来说就会比原生的DOM要简单很多

```
// vue 内部封装的 虚拟dom的VNode节点描述
// 这里的VNode就是对真实DOM的抽象描述
export default class VNode {
  tag: string | void;
  data: VNodeData | void;
  children: ?Array<VNode>;
  text: string | void;
  elm: Node | void;
  .
  .
  .
  }
// vue内部封装的VNode属性是有一点复杂的，当时我们在开发阶段直接调用它处理完的VNode节点就可以了

html: "<div id="test">texttext</div>"
    //html转换成VNode  (ast 树是十分接近VNode节点的，区别是内部的优化不同)
    VNode: {
    // 标签类型
    type: 1,
    // 标签名
    tag: "div",
    // 标签行内属性列表
    attrsList: [{name: "id", value: "test"}],
    // 标签行内属性
    attrsMap: {id: "test"},
    // 标签关系 父亲
    parent: undefined,
    // 字标签属性列表
    children: [{
        type: 3,
```

```
        text: 'texttext'
      }
    ],
    plain: true,
    attrs: [{name: "id", value: "'test'"}]
  }
```

怎么实现一个VNode

首先封装一个 VNode构造函数

```
class VNode {
    constructor (tag, data, children, text, elm) {
        /*当前节点的标签名*/
        this.tag = tag;
        /*当前节点的一些数据信息，比如props、attrs等数据*/
        this.data = data;
        /*当前节点的子节点，是一个数组*/
        this.children = children;
        /*当前节点的文本*/
        this.text = text;
        /*当前虚拟节点对应的真实dom节点*/
        this.elm = elm;
    }
}
```

我们有一个组件

```
<template>
  <span class="demo" v-show="isShow">
    This is a span.
  </span>
</template>
```

创建一个render function函数

```
function render () {
    return new VNode(
        'span',
        {
            /* 指令集合数组 */
            directives: [
                {
                    /* v-show指令 */
                    rawName: 'v-show',
                    expression: 'isShow',
                    name: 'show',
                    value: true
```

```
                }
            ],
            /* 静态class */
            staticClass: 'demo'
        },
        [ new VNode(undefined, undefined, undefined, 'This is a span.') ]
    );
}
```

使用render function 将HTML转换成VNode

```
{
    tag: 'span',
    data: {
        /* 指令集合数组 */
        directives: [
            {
                /* v-show指令 */
                rawName: 'v-show',
                expression: 'isShow',
                name: 'show',
                value: true
            }
        ],
        /* 静态class */
        staticClass: 'demo'
    },
    text: undefined,
    children: [
        /* 子节点是一个文本VNode节点 */
        {
            tag: undefined,
            data: undefined,
            text: 'This is a span.',
            children: undefined
        }
    ]
}
```

VNode就创建完成了，然后我们可以添加一些方法

```
// 创建一个空节点
function createEmptyVNode () {
    const node = new VNode();
    node.text = '';
    return node;
}
// 创建一个文本节点
function createTextVNode (val) {
  return new VNode(undefined, undefined, undefined, String(val));
```

```
}
// 克隆一个 VNode 节点
function cloneVNode (node) {
    const cloneVnode = new VNode(
        node.tag,
        node.data,
        node.children,
        node.text,
        node.elm
    );
    return cloneVnode;
}
```

vue是怎么进行数据更新视图的

在对model进行操作的时候， 会触发Dep中的watcher对象，watcher会调用对应的update来修改视图。最终是将新的VNode和老的VNode进行一个patch的过程，比对出其中的差异，然后只对其中的差异数据进行渲染到页面上

vue的跨平台是怎么做的

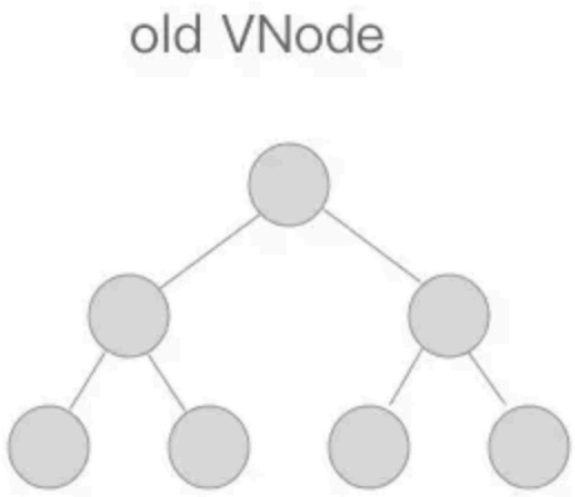
虚拟DOM本质上就是一个js对象，在实际开发过程中实际上你一直在对这行js对象做操作，我们只是需要在vue的适配层中添加响应的适配代码就可以实现了。

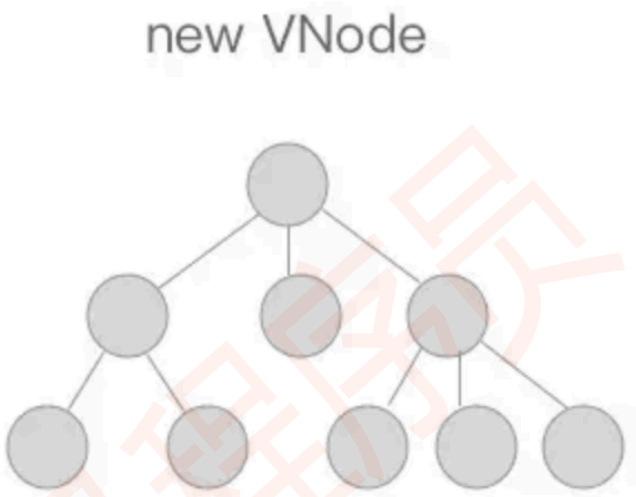
```
const nodeOps = {
    setTextContent (text) {
        if (platform === 'weex') {
            node.parentNode.setAttr('value', text);
        } else if (platform === 'web') {
            node.textContent = text;
        }
    },
    parentNode () {
        //......
    },
    removeChild () {
        //......
    },
    nextSibling () {
        //......
    },
    insertBefore () {
        //......
    }
}
```

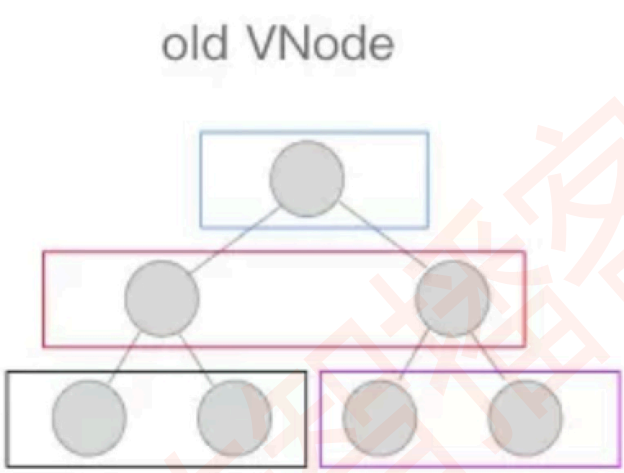
我们根据platform来区分平台，并且针对不同的平台做出相同的对外接口。

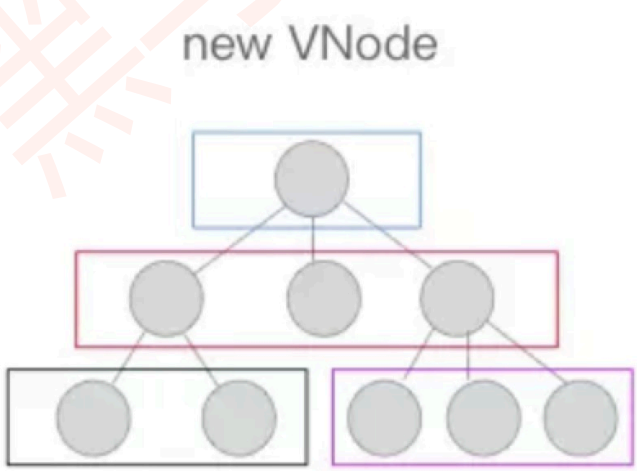
diff算法 （patch是什么）

patch 主要是对虚拟DOM所产生的数据进行差异比较的，是vue中的一个核心功能。
其中差异比较的核心就是diff算法









我们先对新数据和老数据精选划分。划分成四个部分对每一个部分进行比价，只对差异部分进行修改。这样做就会减少性能消耗。

proxy代理？数据双向绑定是怎么保护data中的值得？

代理的作用就是将props和data上的属性代理到VM实例上。
我么通常在调用属性的时候直接就通过this或者VM直接调用，但是我们的属性都是放在data或者props中的，那这是怎么做到的

```
// 封装Object.defineProperty的方法
const sharedPropertyDefinition = {
  enumerable: true,
  configurable: true,
  get: function(){},
  set: function(){}
}
// 封装 proxy代理函数
// target Vue的实例 (this)
// key 就是data或者props的key名字（你自己写的）通常就是key
// 所有前面加_的指内部私有变量
```

```
export function proxy (target: Object, key: string) {
  sharedPropertyDefinition.get = function proxyGetter () {
    return this._data[key]
  }
  sharedPropertyDefinition.set = function proxySetter (val) {
    this._data[key] = val
  }
  // 挂载数据双向绑定
  Object.defineProperty(target, key, sharedPropertyDefinition)
}

// 调用
function initData(vm){
    let data = vm.$options.data;
    const keys = Object.keys(data);
    let i = keys.length;
    while (i--) {
    const key = keys[i]
     proxy(vm, key)
    }
  }
}
```

响应式系统的依赖收集追踪原理

```
// 封装一个 观察者方法
class Observer{
// 创建观察者构造函数
    constructor(value){
        //判断传入的参数是否是一个对象
        if( !value || (typeof value !== 'object') ){
            return;
        }
        //调用walk方法对对象设置Object.defineProperty方法
        this.walk(value);
    }
    // walk方法给对象里面的所有属性添加Object.defineProperty方法
    walk(obj){
        const keys = Object.keys(obj)
        for (let i = 0; i < keys.length; i++) {
          defineReactive(obj ,keys[i])
        }
    }

}

// 给对象设置Object.defineProperty方法
function defineReactive(obj , key ,val){
// 创建一个dep订阅者实例
     const dep = new Dep();
```

```
        // 挂载 Object.defineProperty
        Object.defineProperty(obj, key, {
        // 该属性出现在对象的枚举中
            enumerable: true,
        // 该属性可以被删除
            configurable: true,
            // 访问该属性的时候将在订阅者方法中添加当前的观察者
            get: function reactiveGetter () {
                dep.addSub(Dep.target);
                return val;
            },
            // 设置属性方法的时候 调用的时候运行dep里面的subs里面的观察者方法
            set: function reactiveSetter (newVal) {
                if (newVal === val) return;
                val = newVal;
                dep.notify();
            }
        });
}
//订阅者 Dep
class Dep {
//静态方法 target 观察者
static target = Watcher;
    constructor () {
// 创建 订阅者调用栈
        this.subs = [];
    }
    // 将观察者添加到调用栈中
    addSub (sub) {
        this.subs.push(sub);
    }
// 调用观察者的更新方法
    notify () {
        this.subs.forEach((sub) => {
            sub.update();
        })
    }
}
//观察者
class Watcher {
    constructor () {
        Dep.target = this;
    }
// 更新视图
    update () {
        console.log("更新视图");
    }
}
```

mixins实现原理

```
// mixins 源码文件 src/core/global-api.js
import { mergeOptions } from '../util/index'
// 创建一个mixins方法
export function initMixin (Vue: GlobalAPI) {
  Vue.mixin = function (mixin: Object) {
  // 首先第一步将当前组件的所有参数 options 和传入的mixins一起使用mergeOptions方法合并
    Vue.options = mergeOptions(Vue.options, mixin)
  }
}


//  src/core/util/options.js,

/**
 * Merge two option objects into a new one.
 * Core utility used in both instantiation and inheritance.
 */
export function mergeOptions (
  parent: Object,
  child: Object,
  vm?: Component
): Object {
  // 省略不必要代码
  for (key in child) {
    if (!hasOwn(parent, key)) {
      mergeField(key)
    }
  }
  // 这句话就是在说 将child的属性 合并到options 中去
  // 其中strats 会根据你不同的类型使用不同的合并策略 props , methods 。。。。策略
  function mergeField (key) {
    const strat = strats[key] || defaultStrat
    options[key] = strat(parent[key], child[key], vm, key)
  }
  return options
}
// 判断对象里面有没有这个属性名 而不是原型上面

function hasOwn( obj , key){
    return Object.prototype.hasOwnProperty.call(obj, key)
}

// data的合并
/**
 * Helper that recursively merges two data objects together.
 */
function mergeData (to: Object, from: ?Object): Object {
  if (!from) return to
  let key, toVal, fromVal
```

```
  // 获取所有的 key 数组
  const keys = Object.keys(from)
  for (let i = 0; i < keys.length; i++) {
    key = keys[i]
    toVal = to[key]
    fromVal = from[key]
    // 判断key是否在对象上
    if (!hasOwn(to, key)) {
    // 如果不在对象上面则给这个data值添加监听器
      set(to, key, fromVal)
      // 判断值是否是对象 是对象则递归
    } else if (isPlainObject(toVal) && isPlainObject(fromVal)) {
      mergeData(toVal, fromVal)
    }
    // 所以为什么我们混入的data会不会被添加的原因
  }
  return to
}

// 生命周期混入
//可以发现 Vue 实例的生命周期函数最终都赋值成了一个数组，并对 mixins 中的进行了数组合并。这就是为什么组件 mixins 后的生命周期函数是依次执行的原因。
/**
 * Hooks and props are merged as arrays.
 */
function mergeHook (
  parentVal: ?Array<Function>,
  childVal: ?Function | ?Array<Function>
): ?Array<Function> {
  return childVal
    ? parentVal
      ? parentVal.concat(childVal)
      : Array.isArray(childVal)
        ? childVal
        : [childVal]
    : parentVal
}
```

谈谈你对vue的双向数据绑定原理的理解

vue.js 是采用数据劫持结合发布者-订阅者模式的方式，通过Object.defineProperty()来劫持各个属性的setter，getter，在数据变动时发布消息给订阅者，触发相应的监听回调。

具体步骤：

- 需要observe的数据对象进行递归遍历，包括子属性对象的属性，都加上 setter和getter。这样的话，给这个对象的某个值赋值，就会触发setter，那么就能监听到了数据变化
- compile解析模板指令，将模板中的变量替换成数据，然后初始化渲染页面视图，并将每个指令对应的节点绑定更新函数，添加监听数据的订阅者，一旦数据有变动，收到通知，更新视图
- Watcher订阅者是Observer和Compile之间通信的桥梁，主要做的事情是:

1、在自身实例化时往属性订阅器(dep)里面添加自己
2、自身必须有一个update()方法
3、待属性变动dep.notice()通知时，能调用自身的update()方法，并触发Compile中绑定的回调，则功成身退。

- MVVM作为数据绑定的入口，整合Observer、Compile和Watcher三者，通过Observer来监听自己的model数据变化，通过Compile来解析编译模板指令，最终利用Watcher搭起Observer和Compile之间的通信桥梁，达到数据变化 -> 视图更新；视图交互变化(input) -> 数据model变更的双向绑定效果。

Vue.js的template编译的理解

- 简而言之，就是先转化成AST树，再得到的render函数返回VNode（Vue的虚拟DOM节点）
  详情步骤：
  首先，通过compile编译器把template编译成AST语法树（abstract syntax tree 即 源代码的抽象语法结构的树状表现形式），compile是createCompiler的返回值，createCompiler是用以创建编译器的。另外compile还负责合并option。
  然后，AST会经过generate（将AST语法树转化成render funtion字符串的过程）得到render函数，render的返回值是VNode，VNode是Vue的虚拟DOM节点，里面有（标签名、子节点、文本等等）

React

一切前端概念，都是纸老虎

material-ui

css 样式的生命周期要么写在hooks里面
要么就用

```
withStyles(styles)(({classes, ...props}) => {
    return (
        <li className={classes.span}></li>
    )
})
```

创建一个脚手架

```
create-react-app my-app

// typescript
create-react-app my-app --typescript
```

clsx 插件

```
import clsx from 'clsx';

// Strings (variadic)
clsx('foo', true && 'bar', 'baz');
//=> 'foo bar baz'

// Objects
clsx({ foo:true, bar:false, baz:isTrue() });
//=> 'foo baz'
```

```
// Objects (variadic)
clsx({ foo:true }, { bar:false }, null, { '--foobar':'hello' });
//=> 'foo --foobar'

// Arrays
clsx(['foo', 0, false, 'bar']);
//=> 'foo bar'

// Arrays (variadic)
clsx(['foo'], ['', 0, false, 'bar'], [['baz', [['hello'], 'there']]]);
//=> 'foo bar baz hello there'

// Kitchen sink (with nesting)
clsx('foo', [1 && 'bar', { baz:false, bat:null }, ['hello', ['world']]]
, 'cya');
//=> 'foo bar hello world cya'
```

React 中 setState 什么时候是同步的，什么时候是异步的?

在React中，如果是由React引发的事件处理（比如通过onClick引发的事件处理），调用setState不会立即更新this.state，除此之外的setState调用会同步执行this.state。所谓“除此之外”，指的是绕过React通过addEventListener直接添加的事件处理函数，还有通过setTimeout/setInterval产生的异步调用。

原因：在React的setState函数实现中，会根据一个变量isBatchingUpdates判断是直接更新this.state还是放到队列中回头再说，而isBatchingUpdates默认是false，也就表示setState会同步更新this.state，但是，有一个函数batchedUpdates，这个函数会把isBatchingUpdates修改为true，而当React在调用事件处理函数之前就会调用这个batchedUpdates，造成的后果，就是由React控制的事件处理过程setState不会同步更新this.state。

React 中 《APP /》 和 APP 有什么区别

《APP /》 代表 ReactElement
APP 代表 React Component

React 中控制标签

Refs 提供来人一种方式， 允许我们访问DOm节点或在 render方法中创建的React元素

```
class CustomTextInput extends React.Component {
  constructor(props) {
    super(props);
    // 创建一个 ref 来存储 textInput 的 DOM 元素
    this.textInput = React.createRef();
    this.focusTextInput = this.focusTextInput.bind(this);
  }

  focusTextInput() {
    // 直接使用原生 API 使 text 输入框获得焦点
    // 注意：我们通过 "current" 来访问 DOM 节点
    this.textInput.current.focus();
  }

  render() {
    // 告诉 React 我们想把 <input> ref 关联到
    // 构造器里创建的 `textInput` 上
    return (
      <div>
        <input
          type="text"
          ref={this.textInput} />

        <input
          type="button"
          value="Focus the text input"
          onClick={this.focusTextInput}
        />
      </div>
    );
  }
}
```

react 组件传参

父组件向子组件通讯

通讯是单向的，数据必须是由一方传到另一方。在 React 中，父组件可以向子组件通过传 props 的方式，向子组件进行通讯。

```
class Parent extends Component{
  state = {
    msg: 'start'
  };

  componentDidMount() {
    setTimeout(() => {
      this.setState({
        msg: 'end'
      });
    }, 1000);
  }

  render() {
    return <Child_1 msg={this.state.msg} />;
  }
}

class Child_1 extends Component{
  render() {
    return <p>{this.props.msg}</p>
  }
}
```

子组件向父组件通讯

父组件可以通过传递 props 的方式，自顶而下向子组件进行通讯。而子组件向父组件通讯，同样也需要父组件向子组件传递 props 进行通讯，只是父组件传递的，是作用域为父组件自身的函数，子组件调用该函数，将子组件想要传递的信息，作为参数，传递到父组件的作用域中。

```
class Parent extends Component{
  state = {
    msg: 'start'
  };
```

```
  transferMsg(msg) {
    this.setState({
      msg
    });
  }

  render() {
    return <div>
        <p>child msg: {this.state.msg}</p>
        <Child_1 transferMsg = {msg => this.transferMsg(msg)} />
      </div>;
  }
}

class Child_1 extends Component{
  componentDidMount() {
    setTimeout(() => {
      this.props.transferMsg('end')
    }, 1000);
  }

  render() {
    return <div>
      <p>child_1 component</p>
    </div>
  }
}
```

兄弟组件间通讯

对于没有直接关联关系的两个节点，就如 Child_1 与 Child_2 之间的关系，他们唯一的关联点，就是拥有相同的父组件。参考之前介绍的两种关系的通讯方式，如果我们向由 Child_1 向 Child_2 进行通讯，我们可以先通过 Child_1 向 Parent 组件进行通讯，再由 Parent 向 Child_2 组件进行通讯，所以有以下代码。

```
class Parent extends Component{
  state = {
```

```
    msg: 'start'
  };

  transferMsg(msg) {
    this.setState({
      msg
    });
  }

  componentDidUpdate() {
    console.log('Parent update');
  }

  render() {
    return (
      <div>
        <Child_1 transferMsg = {msg => this.transferMsg(msg)} />
        <Child_2 msg = {this.state.msg} />
      </div>
    );
  }
}

class Child_1 extends Component{
  componentDidMount() {
    setTimeout(() => {
      this.props.transferMsg('end')
    }, 1000);
  }

  componentDidUpdate() {
    console.log('Child_1 update');
  }

  render() {
    return <div>
      <p>child_1 component</p>
    </div>
```

```
  }
}

class Child_2 extends Component{
  componentDidUpdate() {
    console.log('Child_2 update');
  }

  render() {
    return <div>
      <p>child_2 component: {this.props.msg}</p>
      <Child_2_1 />
    </div>
  }
}

class Child_2_1 extends Component{
  componentDidUpdate() {
    console.log('Child_2_1 update');
  }

  render() {
    return <div>
      <p>child_2_1 component</p>
    </div>
  }
}
```

react 调用子组件方法

- 在父组件this上先挂载一个child = ref 接收一个参数 ref 子组件传递过来的this
- <Child onRef={this.onRef}/》 在子组件上用props传递这个函数
- 子组件接收这个函数返回自己的this，如：在生命周期中返回这个this

```
import React, {Component} from 'react';

export default class Parent extends Component {
    render() {
```

```
        return(
            <div>
                <Child onRef={this.onRef} />
                <button onClick={this.click} >click</button>
            </div>
        )
    }
    //定义一个拿子组件返回值this的函数
    onRef = (ref) => {
        this.child = ref
    }
    //调用处理函数
    click = (e) => {
        this.child.myName()
    }

}

class Child extends Component {
    componentDidMount(){
        //通过pros接收父组件传来的方法
        this.props.onRef(this)
    }

    myName = () => alert('子组件fn')

    render() {
        return ('ch')
    }
}
```

函数组件可以使用ref吗？怎么解决

- 你不能在函数组件上使用 ref 属性，因为它们没有实例
- 将组件转化为一个 class
- 通过props的方式传递ref
- 使用forwardRef

```
const FancyButton = React.forwardRef( (props , ref) => <button ref={ref
} /> )
```

React setState 笔试题1

```
class Example extends React.Component {
  constructor() {
    super();
    this.state = {
      val: 0
    };
  }

  componentDidMount() {
    this.setState({val: this.state.val + 1});
    console.log(this.state.val);    // 第 1 次 log

    this.setState({val: this.state.val + 1});
    console.log(this.state.val);    // 第 2 次 log

    setTimeout(() => {
      this.setState({val: this.state.val + 1});
      console.log(this.state.val);  // 第 3 次 log

      this.setState({val: this.state.val + 1});
      console.log(this.state.val);  // 第 4 次 log
    }, 0);
  }

  render() {
    return null;
  }
};
```

- 第一次和第二次都是在react 自身生命周期内，触发是isBatchingUpdates 为true，所以并不会直接执行更新state，而是加入了dirtyComponents， 所以打印时获取的都是更新前的状态 0

- 二次setState 时，获取到this.state.val 都是 0 ，所以执行时都是将0 设置 1，在react内部会被合并掉，只执行一次。设置完成后state.val 值为 1 。
- setTimeout 中的代码，触发isBatchingUpdates 为true，所以能够直接进行更新，所以连着输出 2 3。

  在React的setState函数实现中，会根据一个变量 isBatchingUpdates 来判断是直接同步更新this.state还是放到队列中异步更新。React使用了事物机制，React的每个生命周期和合成事件都处在一个大的事务当中。在事务的前置钩子中调用batchedUpdates方法修改isBatchingUpdates变量为true，在后置钩子中将变量重置为false。原生绑定事件和setTimeout异步的函数没有进入到React的事务中，或者当他们执行时，刚刚的事务已经结束了，后置钩子触发了，所以此时的setState会直接进入非批量更新模式，表现在我们看来成为同步setState

对象重用池

```
const POOL_SIZE = 10;
const traverseContextPool = [];
function getPooledTraverseContext(
  mapResult,
  keyPrefix,
  mapFunction,
  mapContext,
) {
  if (traverseContextPool.length) {
    const traverseContext = traverseContextPool.pop();
    traverseContext.result = mapResult;
    traverseContext.keyPrefix = keyPrefix;
    traverseContext.func = mapFunction;
    traverseContext.context = mapContext;
    traverseContext.count = 0;
    return traverseContext;
  } else {
    return {
      result: mapResult,
      keyPrefix: keyPrefix,
      func: mapFunction,
      context: mapContext,
      count: 0,
```

```
    };
  }
}

function releaseTraverseContext(traverseContext) {
  traverseContext.result = null;
  traverseContext.keyPrefix = null;
  traverseContext.func = null;
  traverseContext.context = null;
  traverseContext.count = 0;
  if (traverseContextPool.length < POOL_SIZE) {
    traverseContextPool.push(traverseContext);
  }
}
```

react 中针对过多属性的对象使用了对象重用池的概念，维护一个大小固定的对象重用池，每一次从这个池子里面取一个对象赋值，用完之后就将对象上的属性置空后丢回池子。维护这个池子的用意就是提高性能，毕竟频繁创建销毁一个有很多属性的对象会销毁性能。

Vuex、Flux、Redux、Dva、MobX 谈谈你对转态管理的认识

不管是Vue还是react，都需要管理状态（state）,比如组件之间都有共享状态的需要。
父子组件，兄弟组件之间共享状态，往往需要写很多没有必要的代码，比如将状态提升到父组件里，或者给兄弟组件写父组件，十分的繁琐
如果不对状态进行有效的管理，状态在什么时候，由于什么原因，如何变化就会不受控制，就很难跟踪和测试了。如果没有经历过这方面的困扰，可以简单理解为会搞得很乱就对了。
根据这些思想，对于状态管理的解决思路就是：把组件之间需要共享的状态抽取出来，遵循特定的约定，统一来管理，让状态的变化可以预测。根据这个思路，产生了很多的模式和库。

- Store 模式

这种是一种最简单的方式
将状态存储到外部变量中来，也可以存储到一个全局变量里面去，但是有一个问题就是，这样修改不会留下任何变更过的记录，这样不利于代码的调试，所以我们就可以使用store模式来解决

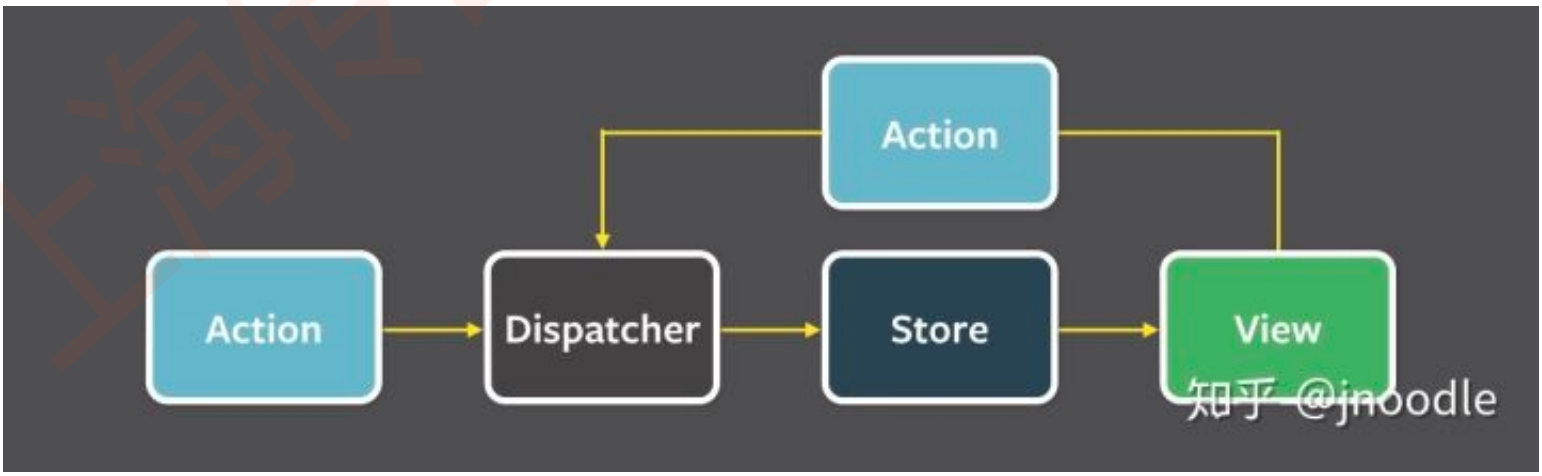
```
    let store = {
        state: {
```

```
            message: "xff"
        },
        setState(data){
        // 创建一个修改参数的action 用来记录修改
            this.state.message = data;
        },

        clearState(){
            this.state.message = ""
        }
    }
```

store的state来存数据， store 里面提供了一些action，这些action来控制state的改变，也就是不直接对state做改变，而是通过action来改变，因为大家都走action，这样我们就可以在出错的时候可以准确的定位到错误点
但是，并没有限制组件里面不能修改state，万一越级修改，可能会出现一些意想不到的问题，所以这个时候flux就诞生了

- Flux

  flux 其实是一种思想，就像MVC， MVVM之类的， 他给出了一些基本概念，所有的框架都可以根据他的思想来做一些实现
  flux将一个应用分成四个部分: View Action Dispatcher Store

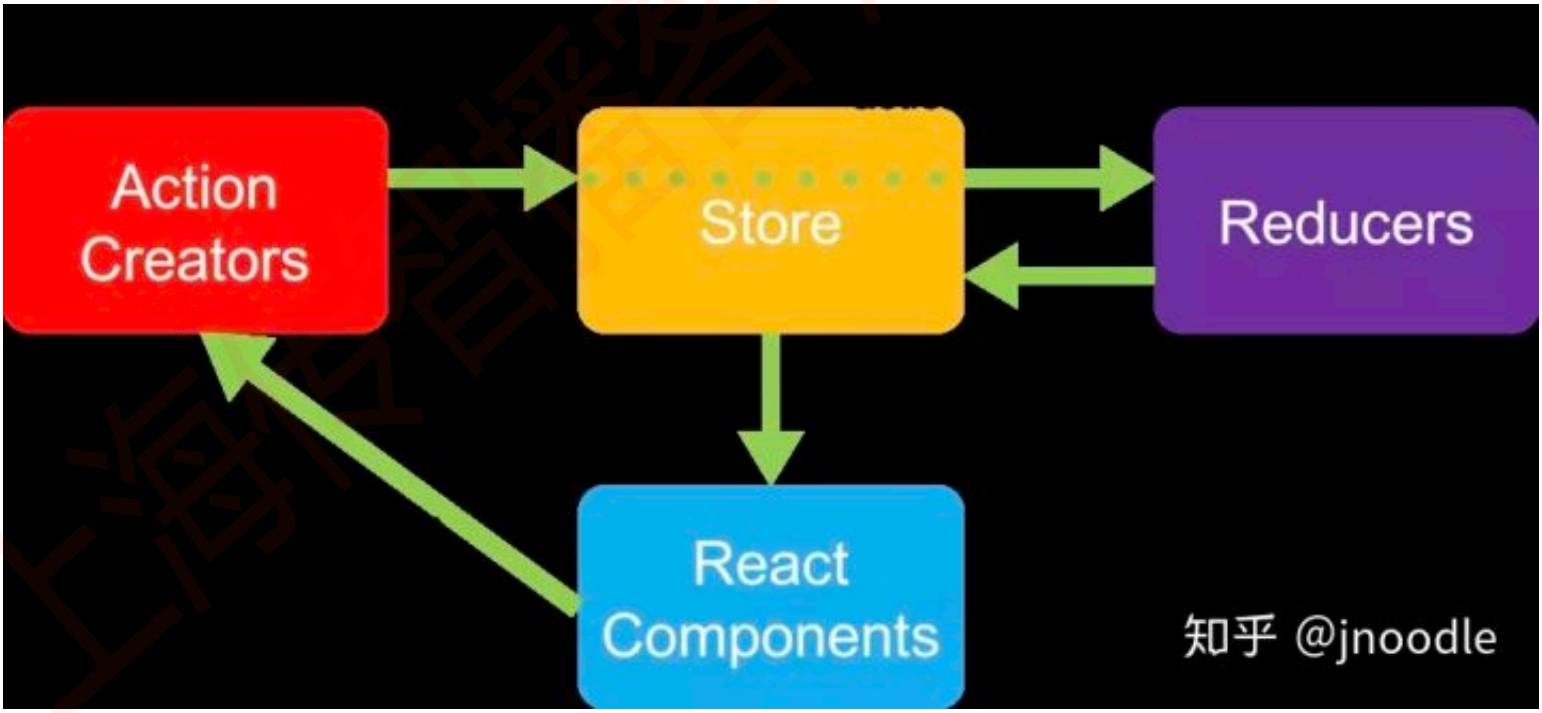


view 就是用来显示数据的，所谓的数据就是store，store就是专门用来存数据的地方，你可以将所有的数据都放在store里面，也可以分开放，所以可以拥有一堆的store。但是所有的View都有一个特点，就是store变了view也会跟着变。
View 不是光用来看的，一般都会有用户操作，用户点个按钮，改个表单啥的，就需要修改 Store。Flux 要求，View 要想修改 Store，必须经过一套流程，有点像我们刚才 Store 模式里

面说的那样。视图先要告诉 Dispatcher，让 Dispatcher dispatch 一个 action，Dispatcher 就像是个中转站，收到 View 发出的 action，然后转发给 Store。比如新建一个用户，View 会发出一个叫 addUser 的 action 通过 Dispatcher 来转发，Dispatcher 会把 addUser 这个 action 发给所有的 store，store 就会触发 addUser 这个 action，来更新数据。数据一更新，那么 View 也就跟着更新了。
这个过程有几个需要注意的点： Dispatcher 的作用是接收所有的 Action，然后发给所有的 Store。这里的 Action 可能是 View 触发的，也有可能是其他地方触发的，比如测试用例。转发的话也不是转发给某个 Store，而是所有 Store。 Store 的改变只能通过 Action，不能通过其他方式。也就是说 Store 不应该有公开的 Setter，所有 Setter 都应该是私有的，只能有公开的 Getter。具体 Action 的处理逻辑一般放在 Store 里。
Flux的最大特点就是数据都是单向流动的

- Redux

  Flux 有一些缺点（特点），比如一个应用可以拥有多个 Store，多个Store之间可能有依赖关系；Store 封装了数据还有处理数据的逻辑。
  所以大家在使用的时候，一般会用 Redux，他和 Flux 思想比较类似，也有差别。



Store
Redux 里面只有一个 Store，整个应用的数据都在这个大 Store 里面。Store 的 State 不能直接修改，每次只能返回一个新的 State。Redux 整了一个 createStore 函数来生成 Store。Store 允许使用 store.subscribe 方法设置监听函数，一旦 State 发生变化，就自动执行这个函数。这样不管 View 是用什么实现的，只要把 View 的更新函数 subscribe 一下，就可以实现 State 变化之后，View 自动渲染了。比如在 React 里，把组件的render方法或setState方法订阅进去就行。

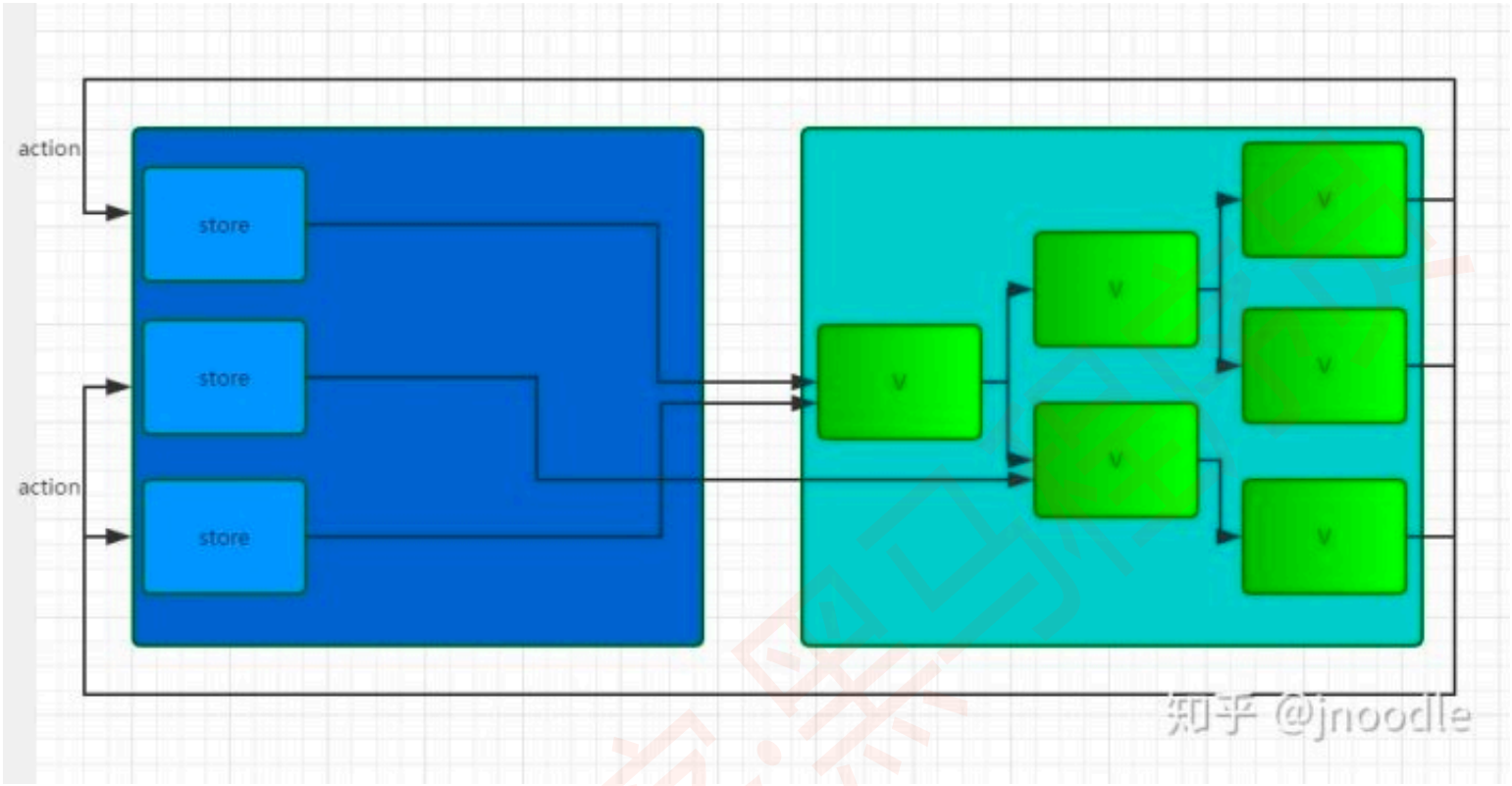
Action
和 Flux 一样，Redux 里面也有 Action，Action 就是 View 发出的通知，告诉 Store State 要改变。Action 必须有一个 type 属性，代表 Action 的名称，其他可以设置一堆属性，作为参数供 State 变更时参考。
Redux 可以用 Action Creator 批量来生成一些 Action。

Reducer
Redux 没有 Dispatcher 的概念，Store 里面已经集成了 dispatch 方法。store.dispatch()是 View 发出 Action 的唯一方法。
Redux 用一个叫做 Reducer 的纯函数来处理事件。Store 收到 Action 以后，必须给出一个新的 State（就是刚才说的Store 的 State 不能直接修改，每次只能返回一个新的 State），这样 View 才会发生变化。这种 State 的计算过程就叫做 Reducer。
Redux 有很多的 Reducer，对于大型应用来说，State 必然十分庞大，导致 Reducer 函数也十分庞大，所以需要做拆分。Redux 里每一个 Reducer 负责维护 State 树里面的一部分数据，多个 Reducer 可以通过 combineReducers 方法合成一个根 Reducer，这个根 Reducer 负责维护整个 State。

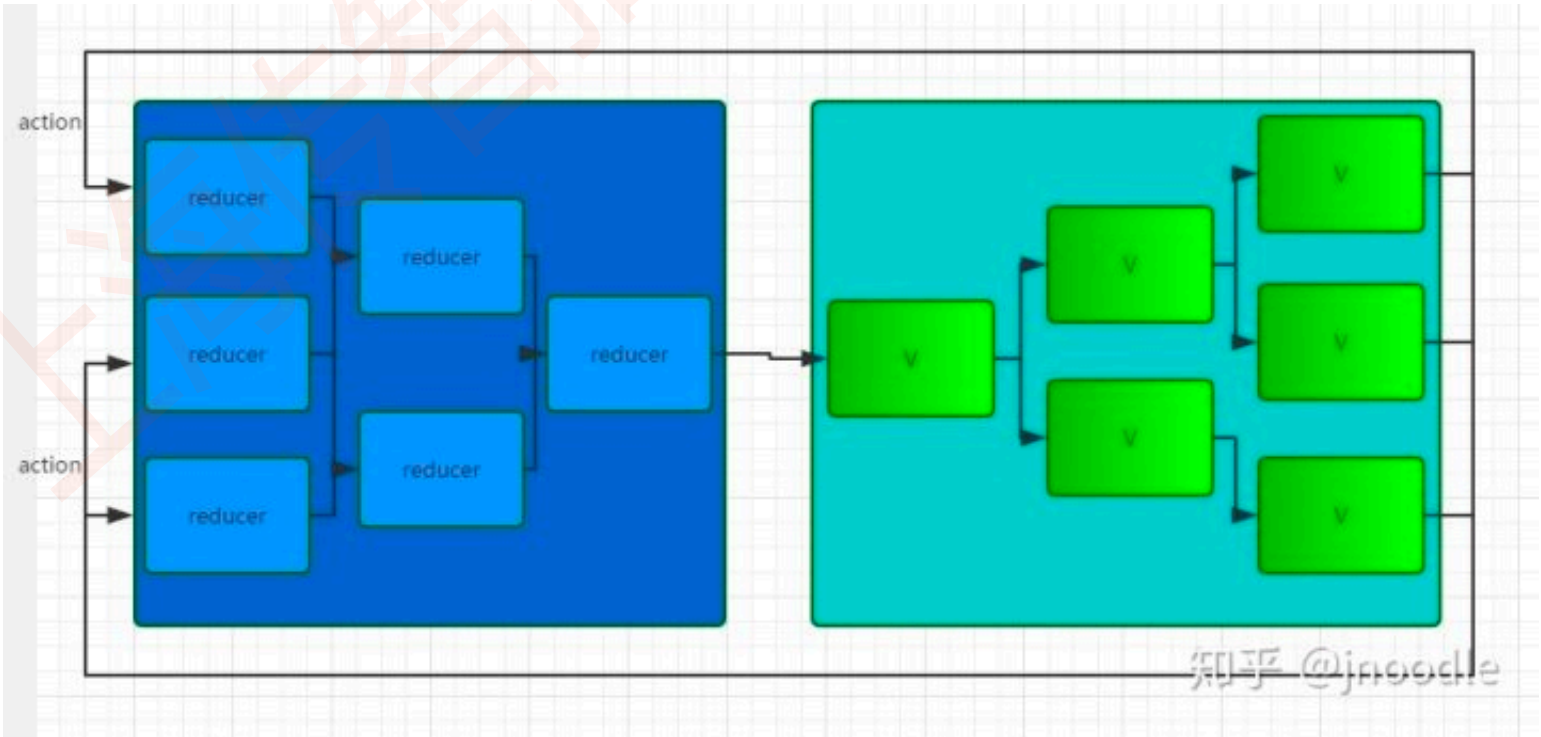
大致流程就是

1. 用户通过 View 发出 Action
2. 然后 Store 自动调用 Reducer，并且传入两个参数：当前 State 和收到的 Action。 Reducer 会返回新的 State 。
3. State 一旦有变化，Store 就会调用监听函数。
4. listener可以通过 store.getState() 得到当前状态。如果使用的是 React，这时可以触发重新渲染 View。

对比 Flux
和 Flux 比较一下：Flux 中 Store 是各自为战的，每个 Store 只对对应的 View 负责，每次更新都只通知对应的View



Redux 中各子 Reducer 都是由根 Reducer 统一管理的，每个子 Reducer 的变化都要经过根 Reducer 的整合：



简单来说，Redux有三大原则： 单一数据源：Flux 的数据源可以是多个。 State 是只读的：Flux 的 State 可以随便改。 * 使用纯函数来执行修改：Flux 执行修改的不一定是纯函数。

Redux 和 Flux 一样都是单向数据流。

- 中间件

  刚才说到的都是比较理想的同步状态。在实际项目中，一般都会有同步和异步操作，所以 Flux、Redux 之类的思想，最终都要落地到同步异步的处理中来。

  在 Redux 中，同步的表现就是：Action 发出以后，Reducer 立即算出 State。那么异步的表现就是：Action 发出以后，过一段时间再执行 Reducer。

  那怎么才能 Reducer 在异步操作结束后自动执行呢？Redux 引入了中间件 Middleware 的概念。

  其实我们重新回顾一下刚才的流程，可以发现每一个步骤都很纯粹，都不太适合加入异步的操作，比如 Reducer，纯函数，肯定不能承担异步操作，那样会被外部IO干扰。Action呢，就是一个纯对象，放不了操作。那想来想去，只能在 View 里发送 Action 的时候，加上一些异步操作了。比如下面的代码，给原来的 dispatch 方法包裹了一层，加上了一些日志打印的功能：

```
let next = store.dispatch;
store.dispatch = function dispatchAndLog(action) {
  console.log('dispatching', action);
  next(action);
  console.log('next state', store.getState());
}
```

  既然能加日志打印，当然也能加入异步操作。所以中间件简单来说，就是对 store.dispatch 方法进行一些改造的函数。

  Redux 提供了一个 applyMiddleware 方法来应用中间件
  这个方法主要就是把所有的中间件组成一个数组，依次执行。也就是说，任何被发送到 store 的 action 现在都会经过thunk，promise，logger 这几个中间件了。

- 处理异步

  对于异步操作来说，有两个非常关键的时刻：发起请求的时刻，和接收到响应的时刻（可能成功，也可能失败或者超时），这两个时刻都可能会更改应用的 state。一般是这样一个过程：

1. 请求开始时，dispatch 一个请求开始 Action，触发 State 更新为“正在请求”状态，View 重新渲染，比如展现个Loading啥的。
2. 请求结束后，如果成功，dispatch 一个请求成功 Action，隐藏掉 Loading，把新的数据更新到

State；如果失败，dispatch 一个请求失败 Action，隐藏掉 Loading，给个失败提示。

显然，用 Redux 处理异步，可以自己写中间件来处理，当然大多数人会选择一些现成的支持异步处理的中间件

- Dva

  Dva是什么呢？官方的定义是：dva 首先是一个基于 redux 和 redux-saga 的数据流方案，然后为了简化开发体验，dva 还额外内置了 react-router 和 fetch，所以也可以理解为一个轻量级的应用框架
  简单理解，就是让使用 react-redux 和 redux-saga 编写的代码组织起来更合理，维护起来更方便

- MobX

  任何源自应用状态的东西都应该自动地获得。译成人话就是状态只要一变，其他用到状态的地方就都跟着自动变

Vuex 的 mutation 和 Redux 的 reducer 中不能做异步操作

vuex 中 的观察工具 devtool 中的 mutation 日志。每一条 mutation 被记录，devtools 都需要捕捉到前一状态和后一状态的快照。然而，在上面的例子中 mutation 中的异步函数中的回调让这不可能完成：因为当 mutation 触发的时候，回调函数还没有被调用，devtools 不知道什么时候回调函数实际上被调用——实质上任何在回调函数中进行的状态的改变都是不可追踪的

---

redux 要在reducer中加入异步的操作，如果你只是单纯想执行异步操作，不会等待异步的返回，那么在reducer中执行的意义是什么。如果想把异步操作的结果反应在state中，首先整个应用的状态将变的不可预测，违背Redux的设计原则，其次，此时的currentState将会是promise之类而不是我们想要的应用状态，根本是行不通的

(组件的)状态(state)和属性(props)之间有何不同

- State 是一种数据结构，用于组件挂载时所需数据的默认值。State 可能会随着时间的推移而发生突变，但多数时候是作为用户事件行为的结果。
- Props(properties 的简写)则是组件的配置。props 由父组件传递给子组件，并且就子组件而言，props 是不可变的(immutable)。组件不能改变自身的 props，但是可以把其子组件的 props 放在一起(统一管理)。Props 也不仅仅是数据--回调函数也可以通过 props 传递。

受控组件(controlled component)

在 HTML 中，类似 input, textarea 和 select 这样的表单元素会维护自身的状态，并基于用户

的输入来更新。当用户提交表单时，前面提到的元素的值将随表单一起被发送。但在 React 中会有些不同，包含表单元素的组件将会在 state 中追踪输入的值，并且每次调用回调函数时，如 onChange 会更新 state，重新渲染组件。一个输入表单元素，它的值通过 React 的这种方式来控制，这样的元素就被称为"受控元素"。

高阶组件(higher order component)HOC

高阶组件是一个以组件为参数并返回一个新组件的函数。HOC 运行你重用代码、逻辑和引导抽象。最常见的可能是 Redux 的 connect 函数。除了简单分享工具库和简单的组合，HOC 最好的方式是共享 React 组件之间的行为。如果你发现你在不同的地方写了大量代码来做同一件事时，就应该考虑将代码重构为可重用的 HOC。

为什么建议传递给 setState 的参数是一个 callback 而不是一个对象

因为 this.props 和 this.state 的更新可能是异步的，不能依赖它们的值去计算下一个 state。

应该在 React 组件的何处发起 Ajax 请求

在 React 组件中，应该在 componentDidMount 中发起网络请求。这个方法会在组件第一次“挂载”(被添加到 DOM)时执行，在组件的生命周期中仅会执行一次。更重要的是，你不能保证在组件挂载之前 Ajax 请求已经完成，如果是这样，也就意味着你将尝试在一个未挂载的组件上调用 setState，这将不起作用。在 componentDidMount 中发起网络请求将保证这有一个组件可以更新了。

createElement 和 cloneElement 有什么区别?

React.createElement()JSX 语法就是用 React.createElement()来构建 React 元素的。它接受三个参数，第一个参数可以是一个标签名。如 div、span，或者 React 组件。第二个参数为传入的属性。第三个以及之后的参数，皆作为组件的子组件
React.cloneElement()与 React.createElement()相似，不同的是它传入的第一个参数是一个 React 元素，而不是标签名或组件。新添加的属性会并入原有的属性，传入到返回的新元素中，而就的子元素奖杯替换。

redux

- redux 是一个应用数据流框架，主要是解决了组件间状态共享的问题，原理是集中式管理，主要有三个核心方法，action，store，reducer，工作流程是 view 调用 store 的 dispatch 接收 action 传入 store，reducer 进行 state 操作，view 通过 store 提供的 getState 获取最新的数据，flux 也是用来进行数据操作的，有四个组成部分 action，dispatch，view，store，工作流程是 view 发出一个 action，派发器接收 action，让 store 进行数据更新，更新完成以后 store 发出 change，view 接受 change 更新视图。Redux 和 Flux 很像。主要区别在于 Flux 有多个可以改变应用状态的 store，在

Flux 中 dispatcher 被用来传递数据到注册的回调事件，但是在 redux 中只能定义一个可更新状态的 store，redux 把 store 和 Dispatcher 合并,结构更加简单清晰

- 新增 state,对状态的管理更加明确，通过 redux，流程更加规范了，减少手动编码量，提高了编码效率，同时缺点时当数据更新时有时候组件不需要，但是也要重新绘制，有些影响效率。一般情况下，我们在构建多交互，多数据流的复杂项目应用时才会使用它们

为什么浏览器无法读取JSX?

浏览器只能处理 JavaScript 对象，而不能读取常规 JavaScript 对象中的 JSX。所以为了使浏览器能够读取 JSX，首先，需要用像 Babel 这样的 JSX 转换器将 JSX 文件转换为 JavaScript 对象，然后再将其传给浏览器。

组件生命周期方法

组件在进入和离开DOM时要经历一系列生命周期方法，下面是这些生命周期方法。

- componentWillMount()
  在渲染前调用,在客户端也在服务端，它只发生一次。
- componentDidMount()
  在第一次渲染后调用，只在客户端。之后组件已经生成了对应的DOM结构，可以通过this.getDOMNode()来进行访问。 如果你想和其他JavaScript框架一起使用，可以在这个方法中调用setTimeout, setInterval或者发送AJAX请求等操作(防止异部操作阻塞UI)。
- componentWillReceiveProps()
  在组件接收到一个新的 prop (更新后)时被调用。这个方法在初始化render时不会被调用。
- shouldComponentUpdate()
  返回一个布尔值。在组件接收到新的props或者state时被调用。在初始化时或者使用forceUpdate时不被调用。 可以在你确认不需要更新组件时使用。
- componentWillUpdate()
  在组件接收到新的props或者state但还没有render时被调用。在初始化时不会被调用。
- componentDidUpdate()
  在组件完成更新后立即调用。在初始化时不会被调用。
- componentWillUnMount()
  组件从 DOM 中移除的时候立刻被调用。
- getDerivedStateFromError()
  这个生命周期方法在ErrorBoundary类中使用。实际上，如果使用这个生命周期方法，任何类都会变成ErrorBoundary。这用于在组件树中出现错误时呈现回退UI，而不是在屏幕上显示一些奇怪的错误。
- componentDidCatch()
  这个生命周期方法在ErrorBoundary类中使用。实际上，如果使用这个生命周期方法，任何类都会变成ErrorBoundary。这用于在组件树中出现错误时记录错误。

声明式编程

声明式编程是一种编程范式，它关注的是你要做什么，而不是如何做。它表达逻辑而不显式地定义步骤。这意味着我们需要根据逻辑的计算来声明要显示的组件。它没有描述控制流步骤。声明式编程的例子有HTML、SQL等

声明式编程 vs 命令式编程

```
const numbers = [1,2,3,4,5];

// 声明式
const doubleWithDec = numbers.map(number => number * 2);

console.log(doubleWithDec)

// 命令式
const doubleWithImp = [];
for(let i=0; i<numbers.length; i++) {
    const numberdouble = numbers[i] * 2;
    doubleWithImp.push(numberdouble)
}

console.log(doubleWithImp)
```

什么是函数式编程

- 不可变性(Immutability)

不可变性意味着不可改变。 在函数式编程中，你无法更改数据，也不能更改。 如果要改变或更改数据，则必须复制数据副本来更改

```
let student = {
    firstName: "testing",
    lastName: "testing",
    marks: 500
}

function changeName(student) {
    // student.firstName = "testing11" //should not do it
    let copiedStudent = Object.assign({}, student);
```

```
    copiedStudent.firstName = "testing11";
    return copiedStudent;
}

console.log(changeName(student));

console.log(student);
```

- 纯函数(Pure Functions)

  纯函数是始终接受一个或多个参数并计算参数并返回数据或函数的函数。 它没有副作用，例如设置全局状态，更改应用程序状态，它总是将参数视为不可变数据。

```
function add (num){ return num * 2 } // 函数式编程中 输入什么值 就会返回对应的结果，并且永远不会返回其他结果 ，就是指 我输入2 返回的永远都是 4 不会有其他结果
```

- 数据转换(Data Transformations)

  我们总是生成原始数据的转换副本，而不是直接更改原始数据
  前面虽然说了数据的不可变，但是我们可以复制一个原始数据的副本，然后将它进行数据转换

- 高阶函数 (Higher-Order Functions)

  高阶函数是将函数作为参数或返回函数的函数，或者有时它们都有。 这些高阶函数可以操纵其他函数
  Array.map，Array.filter和Array.reduce是高阶函数，因为它们将函数作为参数

- 递归

  递归是一种函数在满足一定条件之前调用自身的技术。只要可能，最好使用递归而不是循环。你必须注意这一点，浏览器不能处理太多递归和抛出错误

- 组合

  在React中，我们将功能划分为小型可重用的纯函数，我们必须将所有这些可重用的函数放在一起，最终使其成为产品。 将所有较小的函数组合成更大的函数，最终，得到一个应用程序，这称为组合

组件和不同类型

React 中一切都是组件。 我们通常将应用程序的整个逻辑分解为小的单个部分。 我们将每个单独的部分称为组件。 通常，组件是一个javascript函数，它接受输入，处理它并返回在UI中呈现的React元素。

- 函数/无状态/展示组件

  函数或无状态组件是一个纯函数，它可接受接受参数，并返回react元素。这些都是没有任何副作用的纯函数。这些组件没有状态或生命周期方法，这里有一个例子

```
import React from 'react';
import Jumbotron from 'react-bootstrap/Jumbotron';

export const Header = () => {
    return(
        <Jumbotron style={{backgroundColor:'orange'}}>
            <h1>TODO App</h1>
        </Jumbotron>
    )
}
```

- 类/有状态组件

  类或有状态组件具有状态和生命周期方可能通过setState()方法更改组件的状态。类组件是通过扩展React创建的。它在构造函数中初始化，也可能有子组件,这里有一个例子。

```
import React from 'react';
import '../App.css';
import { ToDoForm } from './todoform';
import { ToDolist } from './todolist';

export class Dashboard extends React.Component {

  constructor(props){
    super(props);

    this.state = {

    }
```

```
  }

  render() {
    return (
      <div className="dashboard">
          <ToDoForm />
          <ToDolist />
      </div>
    );
  }
}
```

- 受控组件

  受控组件是在 React 中处理输入表单的一种技术。表单元素通常维护它们自己的状态，而react则在组件的状态属性中维护状态。我们可以将两者结合起来控制输入表单。这称为受控组件。因此，在受控组件表单中，数据由React组件处理

- 非受控组件

  大多数情况下，建议使用受控组件。有一种称为非受控组件的方法可以通过使用Ref来处理表单数据。在非受控组件中，Ref用于直接从DOM访问表单值，而不是事件处理程序

- 容器组件

  容器组件是处理获取数据、订阅 redux 存储等的组件。它们包含展示组件和其他容器组件，但是里面从来没有html。

- 高阶组件

  高阶组件是将组件作为参数并生成另一个组件的组件。 Redux connect是高阶组件的示例。这是一种用于生成可重用组件的强大技术。

什么是PropTypes

随着时间的推移，应用程序会变得越来越大，因此类型检查非常重要。PropTypes为组件提供类型检查，并为其他开发人员提供很好的文档。如果react项目不使用 Typescript，建议为组件添加 PropTypes。
如果组件没有收到任何 props，我们还可以为每个组件定义要显示的默认 props。这里有一个例子。UserDisplay有三个 prop:name、address和age，我们正在为它们定义默认的props和 prop类型

react 中的样式应用

- 外部样式表

  在此方法中，你可以将外部样式表导入到组件使用类中。 但是你应该使用className而不是class来为React元素应用样式

- 内联样式

  在这个方法中，我们可以直接将 props 传递给HTML元素，属性为style

- 定义样式对象并使用它

  因为我们将javascript对象传递给style属性，所以我们可以在组件中定义一个style对象并使用它。下面是一个示例，你也可以将此对象作为 props 传递到组件树中。

```
import React from 'react';

const footerStyle = {
    width: '100%',
    backgroundColor: 'green',
    padding: '50px',
    font: '30px',
    color: 'white',
    fontWeight: 'bold'
}

export const Footer = () => {
    return(
        <div style={footerStyle}>
            All Rights Reserved 2019
        </div>
    )
}
```

什么是 Fragments

在React中，我们需要有一个父元素，同时从组件返回React元素。有时在DOM中添加额外的节点会很烦人。使用 Fragments，我们不需要在DOM中添加额外的节点。我们只需要用

React.Fragment 或才简写 <> 来包裹内容就行了。如下 所示

```
 // Without Fragments
return (
    <div>
       <CompoentA />
       <CompoentB />
       <CompoentC />
    </div>
)

// With Fragments
  return (
    <React.Fragment>
       <CompoentA />
       <CompoentB />
       <CompoentC />
    </React.Fragment>
  )

  // shorthand notation Fragments
  return (
    <>
       <CompoentA />
       <CompoentB />
       <CompoentC />
    </>
  )
```

如何提高性能

我们可以通过多种方式提高应用性能，以下这些比较重要：

- 适当地使用shouldComponentUpdate生命周期方法。 它避免了子组件的不必要的渲染。 如果树中有100个组件，则不重新渲染整个组件树来提高应用程序性能。
- 使用create-react-app来构建项目，这会创建整个项目结构，并进行大量优化。
- 不可变性是提高性能的关键。不要对数据进行修改，而是始终在现有集合的基础上创建新的集

合，以保持尽可能少的复制，从而提高性能。
- 在显示列表或表格时始终使用 Keys，这会让 React 的更新速度更快
- 代码分离是将代码插入到单独的文件中，只加载模块或部分所需的文件的技术。

react 核心思想

- 声明式编程

  命令式编程是解决做什么的问题，就像是下命令一样，关注于怎么做，而声明式编程关注于得到什么结果，在React中，我们只需要关注“目前的状态是什么”，而不是“我需要做什么让页面变成目前的状态”。React就是不断声明，然后在特定的参数下渲染UI界面。这种编程方式可以让我们的代码更容易被理解，从而易于维护。

- 组件化

  React天生组件化，我们可以将一个大的应用分割成很多小组件，这样有好几个优势。首先组件化的代码像一棵树一样清楚干净，比起传统的面条式代码可读性更高；其次前端人员在开发过程中可以并行开发组件而不影响，大大提高了开发效率；最重要的是，组件化使得复用性大大提高，团队可以沉淀一些公共组件或工具库。

- 单向数据流

  在React中数据流是单向的，由父节点流向子节点，如果父节点的props发生了变化，那么React会递归遍历整个组件树，重新渲染所有使用该属性的子组件。这种单向的数据流一方面比较清晰不容易混乱，另一方面是比较好维护，出了问题也比较好定位。

如何设计一个好组件

组件的主要目的是为了更好的复用，所以在设计组件的时候需要遵循高内聚低耦合的原则。

- 可以通过遵循几种设计模式原则来达到高复用的目的，比如单一职责原则：React推崇的是“组合”而非“继承”，所以在设计时尽量不设计大的组件，而是开发若干个单一功能的组件，重点就是每个组件只做一件事；开放/封闭原则，就是常说的对修改封闭，对扩展开放。在React中我们可以用高阶组件来实现。
- 使用高阶组件来实现组件的复用。高阶组件就是一个包装了另一个React组件的React组件，它包括属性代理（高阶组件操控着传递给被包裹组件的属性）和反向继承（实际上高阶组件继承被包裹组件）。我们可以用高阶组件实现代码复用，逻辑抽象。
- 使用容器组件来处理逻辑，展示组件来展示数据（也就是逻辑处理与数据展示分离）。比如可以在容器组件中进行数据的请求与处理，然后将处理后的数据传递给展示组件，展示组件只负责展示，这样容器组件和展示组件就可以更好地复用了。
- 编写组件代码时要符合规范，总之就是要可读性强、复用性高、可维护性好。

组件的render函数何时被调用

- 组件state发生改变时会调用render函数，比如通过setState函数改变组件自身的state值
- 继承的props属性发生改变时也会调用render函数，即使改变的前后值一样
- React生命周期中有个componentShouldUpdate函数，默认返回true，即允许render被调用，我们也可以重写这个函数，判断是否应该调用render函数

调用render时DOM就一定会被更新吗

不一定更新。

React组件中存在两类DOM，render函数被调用后， React会根据props或者state重新创建一棵virtual DOM树，虽然每一次调用都重新创建，但因为创建是发生在内存中，所以很快不影响性能。而 virtual dom的更新并不意味着真实DOM的更新，React采用diff算法将virtual DOM和真实DOM进行比较，找出需要更新的最小的部分，这时Real DOM才可能发生修改。

所以每次state的更改都会使得render函数被调用，但是页面DOM不一定发生修改。

进行远程数据加载时，应该在哪个周期中完成

最好是在componentDidMount中进行异步请求。如果我们将ajax请求放在生命周期其他函数中，如constructor或componentWIllMount中，我们并不能保证请求仅在组件挂载完毕后才响应。如果我们的数据请求在组件挂载前就完成，并调用setState函数将数据添加到组件状态中，对于未挂载的组件会报错。而在componentDidMount中进行ajax请求能有效避免这个问题。
React 下一代调和算法 Fiber 会通过开始或停止渲染的方式优化应用性能，其会影响到 componentWillMount 的触发次数。对于 componentWillMount 这个生命周期函数的调用次数会变得不确定，React 可能会多次频繁调用 componentWillMount。如果我们将 AJAX 请求放到 componentWillMount函数中，那么显而易见其会被触发多次，自然也就不是好的选择componentWillMount函数，这个方法是在render前立刻执行的，也是服务器渲染中唯一调用的钩子，其实除了同构的需求，通常情况下可以用constructor()方法代替。

在哪些生命周期中可以修改组件的state

- componentDidMount和componentDidUpdate
- constructor、componentWillMount中setState会发生错误：setState只能在mounted或mounting组件中执行
- componentWillUpdate中setState会导致死循环

不同父节点的组件需要对彼此的状态进行改变时应该怎么实现

- 在没有Flux之前，Facebook推荐使用事件机制，但是一旦应用中这种需求增多，事件和回调会满天飞
- 传递接口，就是需要root传递两个接口给A和B，当A想改变B的状态时，A调用root传递给它的接口，然后这个接口再调用root传给B的接口（这个方法也很不科学）
- 用Flux管理状态

state里应该有什么

应该有啥：

- 事件函数可能进行修改的会导致UI进行渲染的数据
  不应该有啥：
- 计算得出的值
- React组件
- props复制来的数据

如何对组件进行优化

- 使用上线构建（Production Build）：会移除脚本中不必要的报错和警告，减少文件体积
- 避免重绘：重写shouldComponentUpdate函数，手动控制是否应该调用render函数进行重绘
- 尽可能使用Immutable Data：尽可能不修改数据，而是重新赋值数据。这样在检测数据对象是否发生修改方面会非常快，因为只需要检测对象引用即可，不需要挨个检测对象属性的更改
- 在渲染组件时尽可能添加key，这样virtual DOM在对比的时候就更容易知道哪里是修改元素，哪里是新插入的元素

你对React生命周期了解吗?

(一) 装载过程

- 当组件第一次被渲染时，依次调用的函数：
  - static propTypes(createClass创建的话：propTypes)
  - static defaultProps(createClass创建的话： getDefaultProps)
  - constructor(初始化state, createClass创建的话： getInitalState)
  - componentWillMount
  - render
  - componentDidMount

(二) 更新过程

- 更新过程会依次调用以下生命周期函数，其中render函数和"装载"过程一样

- componentWillReceiveProps
- shouldComponentUpdate
- componentWillUpdate
- render
- componentDidUpdate
- 并不是所有的更新过程都会执行全部函数

(三) 卸载过程

- React组件的卸载过程只涉及一个函数 componentWillUnMount
  - 当React组件要从DOM树上删除之前，对应的 componentWillUnMount函数会被调用，所以这个函数适合做一些清理性的工作。

history.push 地址改了页面没有刷新

```
// history.js
    import { createBrowserHistory } from 'history'
    export default createBrowserHistory()
    // index.jsx
    import React from 'react';
    import { render } from 'react-dom';
    import { Router, Route } from 'react-router-dom';
    import history from './history.js';
    const Test = () => <div>Welcome to /test</div>;
    const App = () =>
      <div style={{textAlign: 'center'}}>
        <Router history={history}>
          <Route path="/test" component={Test} />
        </Router>
        <button onClick={() => history.push('/test')}>goto /test</b
utton>
        <button onClick={() => history.push('/')}>goto /</button>
      </div>;
    render(<App />, document.getElementById('root'));
```

react中异常怎么捕获

EerrorBoundary是16版本出来的，用来捕获异常

缺点 捕获不了

- 事件处理器
- 异步代码
- 服务端的渲染代码
- 在error boundaries区域内的错误

```
class ErrorBoundary extends React.Component {
constructor(props) {
super(props);
this.state = { hasError: false };
}
componentDidCatch(error, info) {
// Display fallback UI
this.setState({ hasError: true });
// You can also log the error to an error reporting service
logErrorToMyService(error, info);
}
render() {
if (this.state.hasError) {
  // You can render any custom fallback UI
  return <h1>Something went wrong.</h1>;
}
return this.props.children;
}
}
```

```
```

React Hooks

Hooks 是React版本16.8中的新功能。 请记住，我们不能在函数组件中使用state ， 因为它们不是类组件。Hooks 让我们在函数组件中可以使用state 和其他功能

Hook 不会影响你对 React 概念的理解。 恰恰相反，Hook 为已知的 React 概念提供了更直接的 API：props， state，context，refs 以及生命周期。稍后我们将看到，Hook 还提供了一种更强大的方式来组合他们

我们可以使用一些钩子，例如useState，useEffect，useContext，useReducer等

Hooks 的基本规则

- Hooks 应该在外层使用，不应该在循环，条件或嵌套函数中使用
- Hooks 应该只在函数组件中使用。

让我们看一个例子来理解 hooks。 这是一个函数组件，它采用props并在UI上显示这些props。 在useState钩子的帮助下，我们将这个函数组件转换为有状态组件。 首先，我们在第5行定义状态，这相当于

```
constructor(props) {
 super(props);
 this.state = {
     name:'myname', age:10, address:'0000 one street'
 }
}
```

useState返回两个项，一个是user，另一个是setUser函数。 user 是一个可以在没有 this关键字的情况下直接使用的对象，setUser是一个可以用来设置用户点击第21行按钮的状态的函数，该函数等效于以下内容。

```
this.setState({name:'name changed'})
```

```
import React, { useState } from "react";

export const UserDisplay = ({ name, address, age }) => {
    const [user, setUser] = useState({
        name: "myname",
```

```
        age: 10,
        address: "0000 onestreet"
    });

    return (
        <>
            <div>
                <div class="label">Name:</div>
                <div>{user.name}</div>
            </div>
            <div>
                <div class="label">Address:</div>
                <div>{user.address}</div>
            </div>
            <div>
                <div class="label">Age:</div>
                <div>{user.age}</div>
            </div>
            <button onClick={() => setUser({ name: "name changed" })}>
                Click me
            </button>
        </>
    );
};
```

Redux

核心概念

store , state , action , reducer

store 保险箱记录所有的状态 state
需要改变的时候需要告诉 dispatch 要干什么 action
处理变化的 reducer 拿到 state和action 生成新的 state

代码

```
import {createStore} from 'redux'
function counter (state = 0 , action){
    switch(action.type){
        case "加":
            return state++;
        case "减":
            return state--;
        default:
            return 10
    }
}

//创建store 初始化
const store = createStore(counter)
//获取state值
const init = store.getState()

function listener(){
    const current = store.getState();
    console.log(current)
}
// 挂载一个委托事件
store.subscribe(listener)

// 派发事件 传递action
```

```
store.dispatch({type: "加"})
```

React 源码

深入 setState 机制

- setState 异步更新

  setState 通过一个队列机制来实现state更新，当执行setState() 时，会将需要更新的state浅合并后放入状态队列，而不会立即更新state，队列机制可以高效的批量更新state。而如果不通过setState，直接修改this.state的值，则不会放入状态队列，当下一次调用setState对状态队列进行合并时，之前对this.state的修改将会被忽略，造成无法预知的错误
  在setState 官方文档中介绍：将nextState浅合并到当前state。这是在事件处理函数和服务器请求回调函数中触发UI更新的主要方法。不保证setState调用会同步执行，考虑到性能问题，可能会对多次调用做批处理

```
// count = 0
this.setState({count: state.count + 1});
this.setState({count: state.count + 1})
this.setState({count: state.count + 1})
// count === 1
```

本质上等同于：

```
Object.assign(state , {count : state.count + 1},{count : state.count + 1},{count : state.count + 1})
```

- 解决 多次调用setState 没有修改state的值的方法

```
function(state, props) => newState
```

  给setState 传入一个函数作为参数，这会将一个更新操作加入到更新队列里面去，在设置任何值之前，此操作会查询前一刻的state 和props. setState() 并不会立即改变this.state,而是会创建一个待执行的变动，调用此方法后访问this.state有可能会得到尚未改变的state

```
this.setState( (state , props) => ({count: state.count + props.num}) )
```

setState 工作机制

```
[{count: 1} , {count:1}].reduce( (state , props) => ({count: state.count + props.count}) , {count: 0} );
```

- setState 源码

```
// 更新 state
ReactComponent.prototype.setState = function(partialState, callback) {
    this.updater.enqueueSetState(this, partialState)
    if (callback) {
        this.updater.enqueueCallback(this, callback, 'setState')
    }
}

enqueueSetState: function(publicInstance, partialState) {
    var internalInstance = getInternalInstanceReadyForUpdate(
        publicInstance,
        'setState'
    )
    if (!internalInstance) {
        return
    }

    // 更新队列合并操作
    var queue = internalInstance._pendingStateQueue || (internalInstance._pendingStateQueue=[])
    queue.push(partialState)
    enqueueUpdate(internalInstance)
}

// 如果存在 _pendingElement、_pendingStateQueue、_pendingForceUpdate，则更新组件
performUpdateIfNecessary: function(transaction) {
    if (this._pendingElement != null) {
        ReactReconciler.receiveComponent(this, this._pendingElement, tr
```

```
ansaction, this._context)
    }

    if (this._pendingStateQueue != null || this._pendingForceUpdate) {
        this.updateComponent(transaction, this._currentElement, this._c
urrentElement, this._context, this._context)
    }
}
```

在react中，如果是由react引发的事件处理，调用setState不会同步更新this.state, 除此之外的setState调用会同步执行this.state, 所谓"除此之外",指的是绕过react通过addEventListener直接添加的事件处理函数，还有通过setTimeout/setInterval产生的异步调用

React.createElement || jsx

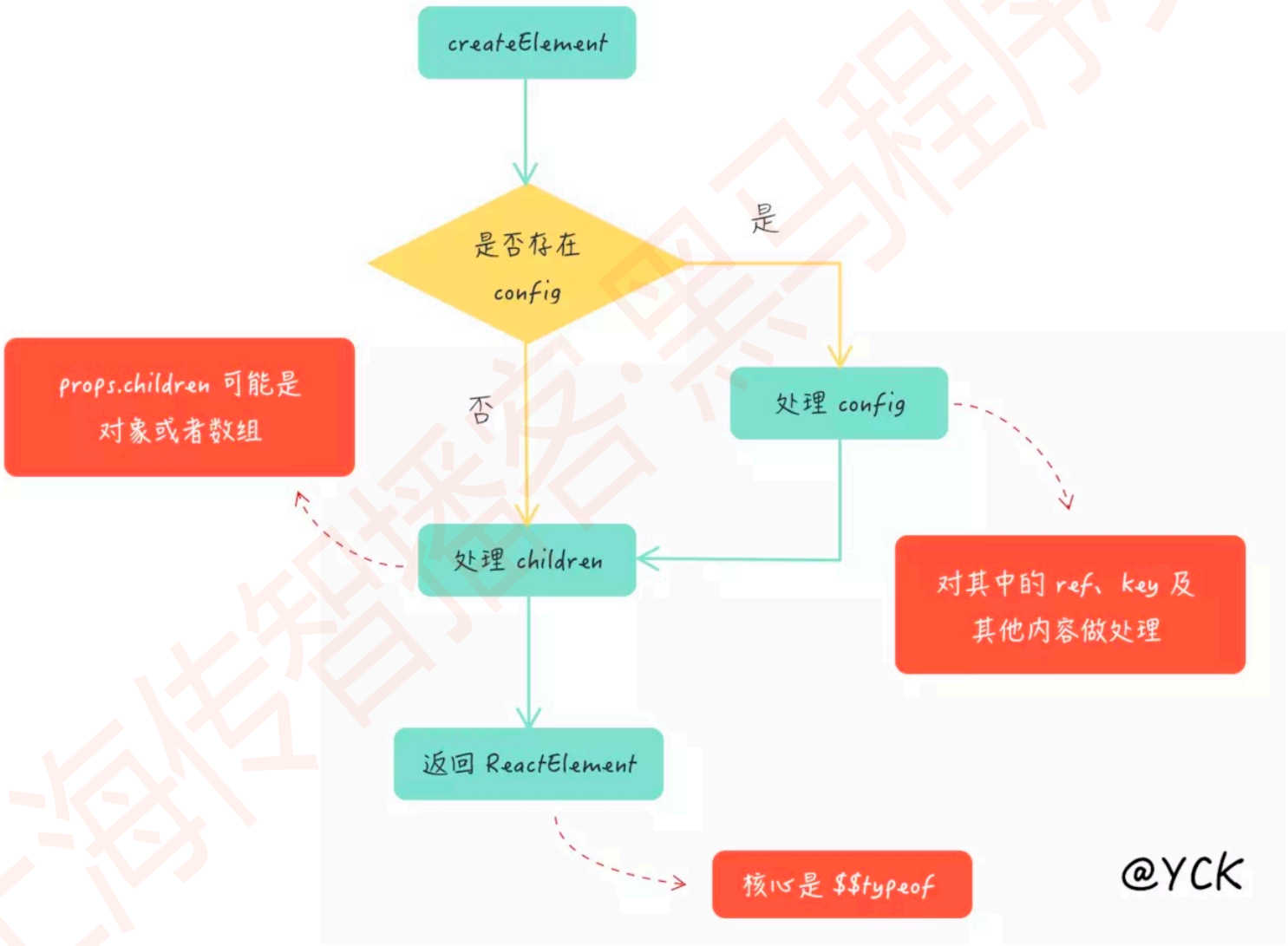
在写react代码的使用当我们写jsx的时候，首先第一点必须要引入react
jsx代码在执行期间会被babel编译为 React.createElement, 不引入react的话就不能使用React.createElement

```
<span class="spa1">name</div>

// 上面的代码会被编译成
React.createElement("span" , { class: "spa1" } , "name")
```

- 源码解析

```
export function createElement(type, config, children){
// 第一步 判断是否传入config 比如 <div className = "sp" /> 中的className 就会被解析到配置中去。
// 第二步 处理 children 如果他是一个值就将他转化成对象，如果传入多个值，就将它转换成数组
// 第三步 判断是否给props 设置defaultProps ，如果设置了，就判断props是否有值，只有当props的值为undefined 时才会设置默认值

// 最后一步会 调用ReactElement 方法帮助我们创建一个 react Element，然后会生成一个$$typeof 来标识 ReactElement
```

```
}


// 所以 <APP /> 代表着 ReactElement，APP 代表着 React Component
```



React Component

```
function Component(props, context, updater) {
  this.props = props;
  this.context = context;
  // refs 有好几种创建方式
```

```
  /**
  // 回调refs
  // 它能助你更精细地控制何时 refs 被设置和解除
   ref = { el => this.el = el } 这种创建方式是推荐的方式
   也可以通过 React.createRef来创建
   this.el = React.createRef()
   ref = {this.el}
  */
  this.refs = emptyObject;
  // We initialize the default updater but the real one gets injected b
y the
  // renderer.
 // ReactNoopUpdateQueue 提示警告的
  this.updater = updater || ReactNoopUpdateQueue;
}
// 当我们在组件中调用 setState 其实就是调用的这个方法
Component.prototype.setState = function(partialState, callback) {
// 首先第一步进行异常处理
  invariant(
    typeof partialState === 'object' ||
      typeof partialState === 'function' ||
      partialState == null,
    'setState(...): takes an object of state variables to update or a '
 +
      'function which returns an object of state variables.',
  );
  之后调用updater方法
  this.updater.enqueueSetState(this, partialState, callback, 'setState'
);
};
// 强制渲染
Component.prototype.forceUpdate = function(callback) {
  this.updater.enqueueForceUpdate(this, callback, 'forceUpdate');
};
// ComponentDummy 继承自 Component
function ComponentDummy() {}
ComponentDummy.prototype = Component.prototype;
function PureComponent(props, context, updater) {
```

```
  this.props = props;
  this.context = context;
  this.refs = emptyObject;
  this.updater = updater || ReactNoopUpdateQueue;
}
const pureComponentPrototype = (PureComponent.prototype = new Component
Dummy());
pureComponentPrototype.constructor = PureComponent;
// Avoid an extra prototype jump for these methods.
Object.assign(pureComponentPrototype, Component.prototype);
// 通过这个变量区别下普通的 Component
pureComponentPrototype.isPureReactComponent = true;
```

Refs

```
export function createRef() {
  const refObject = {
    current: null,
  };
  if (__DEV__) {
    Object.seal(refObject);
  }
  return refObject;
}
// 内部实现，当我们想使用ref，只需要取出其中的current对象就可以了

// 之前函数组件里面使用ref 可以通过 porps来传递，但是也可以用 forwardRef 去解决
这个问题

export default function forwardRef( render ){ // render = (props , ref
=> ReactNode)
    return {
        $$typeof:REACT_FORWARD_REF_TYPE,
        render
    }
}
```

```
const FancyButton = React.forwardRef((props, ref) => (
  <button ref={ref} className="FancyButton">
    {props.children}
  </button>
))
```

Typescript

ts 是什么

TypeScript 是 JavaScript 的一个超集，主要提供了类型系统和对 ES6 的支持，它由 Microsoft 开发，代码开源于 GitHub 上。

ts 的 数据类型

- 布尔

```
let fl: boolean = false;
```

- 数值

```
    let num: number = 2;
```

- 字符串

```
let str: string = 'qwe';
```

- 空值

```
// ts 中有空值的概念
// 在变量中空值基本没有什么作用 因为他只能赋值 undefined 和 null

let v: void = undefined

// 函数中就是函数没有返回值
function fun(): void{}
```

- Null 和 Undefined

```
let u: undefined = undefined;
```

```
let n: null = null;

// 他和 void的区别就是 Null 和 Undefined是所有类型的子类，也就是说所有类型的变量都可以被赋值成Null 和 Undefined

let num: number = 13;
let u: undefined = undefined;
num = u ; // 这个是没有任何问题的
```

- 任意值类型

```
let str: any = 'str';
str = 7;
```

- 类型推论

```
// 如果你在赋值阶段没有声明类型，ts会自动根据你的赋值推倒变量类型
let str = 'str'
str = 3 // 报错
// 相当于
let str: string = 'str'
str = 3
```

- 联合类型

```
// 联合类型（Union Types）表示取值可以为多种类型中的一种
let str: number | string ;
str = 'wqe';
str = 7;

// 访问联合类型的属性或方法
let str: number | string;
str.length // 报错 number没有length方法
```

- 接口

定义对象的类型

```
interface Person {
    name: string;
    age: number;
}

let tom: Person = {
    name: 'Tom',
    age: 25
};

// 可选属性
interface Person {
    name: string;
    age?: number;
}

let tom: Person = {
    name: 'Tom'
};

//任意属性
interface Person {
    name: string;
    age?: number;
    [propName: string]: any;
}

let tom: Person = {
    name: 'Tom',
    gender: 'male'
};

// 只读属性
// 有时候我们希望对象中的一些字段只能在创建的时候被赋值，那么可以用 readonly 定义只读属性
```

```
interface Person {
    readonly id: number;
    name: string;
    age?: number;
    [propName: string]: any;
}

let tom: Person = {
    id: 89757,
    name: 'Tom',
    gender: 'male'
};
```

- 数组的类型

```
let arr: number[] = [1,2,3,4]

// 数组泛型
let arr: Array<number> = [1,2,3,4]

// 用接口表示数组

interface list{
    [index: number]: number
}

let arr: list = [1,2,3,4]

//any
let list: any[] = [1,2,3,'qwe',{}]
```

- 函数声明

```
function fun(num: number , str: string): number{
        return 123
}
```

```
// typescript =>
// 在 TypeScript 的类型定义中，=> 用来表示函数的定义，左边是输入类型，需要用括号括起来，右边是输出类型。
let fun: (x: number , y: string) => number = function(x: number , y: string): number{
    return 123
}

// typescript 可选参数
function fun (x: number , y?: string){
// y 就是一个可选参数
}

// 剩余参数
 function fun(x: number , ...y: any[]){}

 // 重载

 function fun (x: string | number): string | number{}
```

- 类型断言

```
   function fun(num: string | number){
     (num as string).length
   }
```

- 全局声明变量

```
<!--
declare var 声明全局变量
declare function 声明全局方法
declare class 声明全局类
declare enum 声明全局枚举类型
declare namespace 声明（含有子属性的）全局对象
```

```
interface 和 type 声明全局类型
export 导出变量
export namespace 导出（含有子属性的）对象
export default ES6 默认导出
export = commonjs 导出模块
export as namespace UMD 库声明全局变量
declare global 扩展全局变量
declare module 扩展模块
-->
/// <reference /> 三斜线指令
```

- 声明全局文件

```
// jQuery
// src/jQuery.d.ts

declare var jQuery: (selector: string) => any;
// 声明文件必需以 .d.ts 为后缀

// 使用
// src/index.ts

jQuery('#foo');

// 第三方文件声明
npm install @types/jquery --save-dev
```

- 类型别名

```
type Name = string;
type NameResolver = () => string;
type NameOrResolver = Name | NameResolver;
function getName(n: NameOrResolver): Name {
    if (typeof n === 'string') {
        return n;
    } else {
```

```
        return n();
    }
}
```

- 字符串字面量类型

```
type listName = 'name' | 'age' | 'type'

function fun (x: number , y: listName){}
fun(1 , 'age');
fun(2 , 'type')
```

- 元组

```
// 数组合并了相同类型的对象，而元组（Tuple）合并了不同类型的对象

let list: [string , number] = [1,'aws']
// 直接对元组类型的变量进行初始化或者赋值的时候，需要提供所有元组类型中指定的项
```

- 枚举

```


// 枚举（Enum）类型用于取值被限定在一定范围内的场景，比如一周只能有七天，颜色限定为红绿蓝等

enum Days {Sun, Mon, Tue, Wed, Thu, Fri, Sat};
// 枚举成员会被赋值为从 0 开始递增的数字，同时也会对枚举值到枚举名进行反向映射


console.log(Days["Sun"] === 0); // true
console.log(Days["Mon"] === 1); // true
console.log(Days["Tue"] === 2); // true
console.log(Days["Sat"] === 6); // true

console.log(Days[0] === "Sun"); // true
console.log(Days[1] === "Mon"); // true
```

```
console.log(Days[2] === "Tue"); // true
console.log(Days[6] === "Sat"); // true

// 编译
var Days;
(function (Days) {
    Days[Days["Sun"] = 0] = "Sun";
    Days[Days["Mon"] = 1] = "Mon";
    Days[Days["Tue"] = 2] = "Tue";
    Days[Days["Wed"] = 3] = "Wed";
    Days[Days["Thu"] = 4] = "Thu";
    Days[Days["Fri"] = 5] = "Fri";
    Days[Days["Sat"] = 6] = "Sat";
})(Days || (Days = {}));

// 手动赋值

enum Days {Sun = 3, Mon = 1, Tue, Wed, Thu, Fri, Sat};

console.log(Days["Sun"] === 3); // true
console.log(Days["Wed"] === 3); // true
console.log(Days[3] === "Sun"); // false
console.log(Days[3] === "Wed"); // true

// 编译
var Days;
(function (Days) {
    Days[Days["Sun"] = 3] = "Sun";
    Days[Days["Mon"] = 1] = "Mon";
    Days[Days["Tue"] = 2] = "Tue";
    Days[Days["Wed"] = 3] = "Wed";
    Days[Days["Thu"] = 4] = "Thu";
    Days[Days["Fri"] = 5] = "Fri";
    Days[Days["Sat"] = 6] = "Sat";
})(Days || (Days = {}));

// 手动赋值的枚举项可以不是数字，此时需要使用类型断言来让 tsc 无视类型检查 (编译出的 js 仍然是可用的)：
```

```
enum Days {Sun = 7, Mon, Tue, Wed, Thu, Fri, Sat = ('S' as any)};
```

- 类

```
<!--public 修饰的属性或方法是公有的，可以在任何地方被访问到，默认所有的属性和方法都是 public 的
private 修饰的属性或方法是私有的，不能在声明它的类的外部访问
protected 修饰的属性或方法是受保护的，它和 private 类似，区别是它在子类中也是允许被访问的-->

class Animal {
    public name;
    public constructor(name) {
        this.name = name;
    }
}

let a = new Animal('Jack');
console.log(a.name); // Jack
a.name = 'Tom';
console.log(a.name); // Tom

class Animal {
    private name;
    public constructor(name) {
        this.name = name;
    }
}

let a = new Animal('Jack');
console.log(a.name); // Jack
a.name = 'Tom'; // 报错


class Animal {
    protected name;
    public constructor(name) {
```

```
        this.name = name;
    }
}

class Cat extends Animal {
    constructor(name) {
        super(name);
        console.log(this.name); // 子类中可以访问
    }
}

// 抽象类 abstract

// 首先，抽象类是不允许被实例化的
abstract class Animal {
    public name;
    public constructor(name) {
        this.name = name;
    }
    public abstract sayHi();
}

let a = new Animal('Jack'); // 报错

// 抽象类中的抽象方法必须被子类实现
abstract class Animal {
    public name;
    public constructor(name) {
        this.name = name;
    }
    public abstract sayHi(); // 报错
}

class Cat extends Animal {
    public eat() {
        console.log(`${this.name} is eating.`);
    }
}
```

```
let cat = new Cat('Tom');


// 正确结果
abstract class Animal {
    public name;
    public constructor(name) {
        this.name = name;
    }
    public abstract sayHi();
}

class Cat extends Animal {
    public sayHi() {
        console.log(`Meow, My name is ${this.name}`);
    }
}

let cat = new Cat('Tom');
```

- 类与接口

```
interface Alarm {
    alert();
}

interface Light {
    lightOn();
    lightOff();
}

class Car implements Alarm, Light {
    alert() {
        console.log('Car alert');
    }
    lightOn() {
```

```
        console.log('Car light on');
    }
    lightOff() {
        console.log('Car light off');
    }
}
```

- 泛型

```
// 泛型（Generics）是指在定义函数、接口或类的时候，不预先指定具体的类型，而在使用的时候再指定类型的一种特性。

function createArray(length: number, value: any): Array<any> {
    let result = [];
    for (let i = 0; i < length; i++) {
        result[i] = value;
    }
    return result;
}

createArray(3, 'x'); // ['x', 'x', 'x']

// 泛型

function createArray<T>(length: number, value: T): Array<T> {
    let result: T[] = [];
    for (let i = 0; i < length; i++) {
        result[i] = value;
    }
    return result;
}

createArray<string>(3, 'x'); // ['x', 'x', 'x']

// 多个类型参数
```

```
function swap<T, U>(tuple: [T, U]): [U, T] {
    return [tuple[1], tuple[0]];
}

swap([7, 'seven']); // ['seven', 7]

// 泛型约束
function loggingIdentity<T>(arg: T): T {
    console.log(arg.length); // 错误
    return arg;
}

// 定义接口 使用 extends 约束泛型
interface Lengthwise {
    length: number;
}

function loggingIdentity<T extends Lengthwise>(arg: T): T {
    console.log(arg.length);
    return arg;
}
// 泛型类
class GenericNumber<T> {
    zeroValue: T;
    add: (x: T, y: T) => T;
}

let myGenericNumber = new GenericNumber<number>();
myGenericNumber.zeroValue = 0;
myGenericNumber.add = function(x, y) { return x + y; };

// 泛型参数的默认类型
function createArray<T = string>(length: number, value: T): Array<T> {
    let result: T[] = [];
    for (let i = 0; i < length; i++) {
        result[i] = value;
    }
    return result;
```

```
}
```

杂项修改

什么是同源策略和跨域问题?

- 同源策略和跨域问题

  浏览器因为安全的原因，有一套策略，叫做同源策略。这套策略只在浏览器端才会有.
  当你的请求协议、域名或者端口有一个不同就会跨域。
  跨域时，浏览器实际上是拿到数据的，但是浏览器不会将数据给你，所以你也就没有办法获取到数据。
  跨域主要的作用是防止CSRF攻击，简单点说，CSRF 攻击是利用用户的登录态发起恶意请求。
  解决跨域的方案

  - JSONP 使用script标签向后台请求数据，但是只适用于get请求。
  - CORS 是让服务端设置Access-Control-Allow-Origin，这样浏览器就不会包跨域错误。
  - 反向代理 是搭建一个自己的服务器，让我的服务器请求数据，拿到数据之后在放回给我自己。

http和https的区别

1. https协议需要到ca申请证书，一般免费证书较少，因而需要一定费用。
2. http是超文本传输协议，信息是明文传输，https则是具有安全性的ssl加密传输协议。
3. http和https使用的是完全不同的连接方式，用的端口也不一样，前者是80，后者是443。
4. http的连接很简单，是无状态的；HTTPS协议是由SSL+HTTP协议构建的可进行加密传输、身份认证的网络协议，比http协议安全。

https的工作原理

1. 客户使用https的URL访问Web服务器，要求与Web服务器建立SSL连接。
2. Web服务器收到客户端请求后，会将网站的证书信息（证书中包含公钥）传送一份给客户端。
3. 客户端的浏览器与Web服务器开始协商SSL连接的安全等级，也就是信息加密的等级。
4. 客户端的浏览器根据双方同意的安全等级，建立会话密钥，然后利用网站的公钥将会话密钥加密，并传送给网站。
5. Web服务器利用自己的私钥解密出会话密钥。
6. Web服务器利用会话密钥加密与客户端之间的通信。

Jenkins

Jenkins是一个开源软件项目，是基于Java开发的一种持续集成工具，用于监控持续重复的工作，旨在提供一个开放易用的软件平台，使软件的持续集成变成可能
Jenkins 配合git可以自动化部署项目。

gos

git 管理平台，开源的一个项目，可以部署在公司内部，和GitHub很相似

restful api

API 设计规范，用于 Web 数据接口的设计。
动词 + 宾语

1. GET：读取（Read）
2. POST：新建（Create）
3. PUT：更新（Update）
4. PATCH：更新（Update），通常是部分更新
5. DELETE：删除（Delete）
   宾语必须是名词
6. /getAllCars
7. /createNewCar
8. /deleteAllRedCars
   具体的转态码
9. 1xx：相关信息
10. 2xx：操作成功
11. 3xx：重定向
12. 4xx：客户端错误
13. 5xx：服务器错误
    很好的可读性
14. http://www.artech.com/employees/c001（编号C001的员工）
15. http://www.artech.com/sales/2013/12/31（2013年12月31日的销售额）
16. http://www.artech.com/orders/2013/q4（2013年第4季度签订的订单） #### AOT
    AOT，Ahead Of Time，指运行前编译
17. 在程序运行前编译，可以避免在运行时的编译性能消耗和内存消耗
18. 可以在程序运行初期就达到最高性能
19. 可以显著的加快程序的启动
20. 在程序运行前编译会使程序安装的时间增加
21. 将提前编译的内容保存会占用更多的外

http缓存

http 在请求资源的时候会对资源进行缓存，其中缓存主要有二种 强制缓存和对比缓存

- 强制缓存：主要是在缓存数据未失效的情况下，可以直接使用缓存数据，后台主要通过修改响应头的数据说明缓存资源的失效规则，从而达到的一种缓存效果，当资源被存储到缓存数据库中的时候，浏览器就不会再向服务器请求资源的，而是直接从缓存数据库中获取资源
- 对比缓存和强制缓存大致相同，但是对比缓存每一次获取资源的时候都会去向后台确认，我的缓存资源是否失效，如果失效我就不从缓存数据库拿数据了。

js对象的三大特性

扩展 ， 密封，冻结

- 扩展 : 如果一个对象可以添加新的属性，则这个对象是可扩展的。
  让这个对象变的不可扩展，也就是不能再有新的属性

  Object.isExtensible 方法
  Object.preventExtensions 方法

- 密封： 密封对象是指那些不可 扩展 的，且所有自身属性都不可配置的（non-configurable）对象。

  Object.isSealed 方法
  Object.seal 方法

- 冻结: 一个对象是冻结的（frozen）是指它不可扩展，所有属性都是不可配置的（non-configurable），且所有数据属性（data properties）都是不可写的（non-writable）。

  Object.isFrozen 方法
  Object.freeze 方法
  浅冻结 与 深冻结

去重

已知如下数组，编写一个程序将数组扁平化去并除其中重复部分数据，最终得到一个升序且不重复的数组
var arr = [[1, 2, 2], [3, 4, 5, 5], [6, 7, 8, 9, [11, 12, [12, 13, [14]]]], 10];
[...new Set(arr.flat(Infinity))].sort((i , q) => i-q)

http/2

2015年，HTTP/2 发布。HTTP/2是现行HTTP协议（HTTP/1.x）的替代，但它不是重写，HTTP方法/状态码/语义都与HTTP/1.x一样。HTTP/2基于SPDY3，专注于性能，最大的一个

目标是在用户和网站间只用一个连接（connection）。
HTTP/2的特性：头部压缩、Server Push、多路复用 ， 二进制传输
HTTP/2 采用二进制格式传输数据，而非 HTTP 1.x 的文本格式，二进制协议解析起来更高效。 HTTP / 1 的请求和响应报文，都是由起始行，首部和实体正文（可选）组成，各部分之间以文本换行符分隔。HTTP/2 将请求和响应数据分割为更小的帧，并且它们采用二进制编码。

在 HTTP/2 中，有了二进制分帧之后，HTTP /2 不再依赖 TCP 链接去实现多流并行了，在HTTP/2中：

- 同域名下所有通信都在单个连接上完成。
- 单个连接可以承载任意数量的双向数据流。
- 数据流以消息的形式发送，而消息又由一个或多个帧组成，多个帧之间可以乱序发送，因为根据帧首部的流标识可以重新组装。

这一特性，使性能有了极大提升：

- 同个域名只需要占用一个 TCP 连接，使用一个连接并行发送多个请求和响应,消除了因多个TCP 连接而带来的延时和内存消耗。
- 并行交错地发送多个请求，请求之间互不影响。
- 并行交错地发送多个响应，响应之间互不干扰。
- 在HTTP/2中，每个请求都可以带一个31bit的优先值，0表示最高优先级， 数值越大优先级越低。有了这个优先值，客户端和服务器就可以在处理不同的流时采取不同的策略，以最优的方式发送流、消息和帧。

在 HTTP/1 中，我们使用文本的形式传输 header，在 header 携带 cookie 的情况下，可能每次都需要重复传输几百到几千的字节。
为了减少这块的资源消耗并提升性能， HTTP/2对这些首部采取了压缩策略：

- HTTP/2在客户端和服务器端使用“首部表”来跟踪和存储之前发送的键－值对，对于相同的数据，不再通过每次请求和响应发送；
- 首部表在HTTP/2的连接存续期内始终存在，由客户端和服务器共同渐进地更新;
- 每个新的首部键－值对要么被追加到当前表的末尾，要么替换表中之前的值

Server Push即服务端能通过push的方式将客户端需要的内容预先推送过去，也叫“cache push”。

- 可以想象以下情况，某些资源客户端是一定会请求的，这时就可以采取服务端 push 的技术，提前给客户端推送必要的资源，这样就可以相对减少一点延迟时间。当然在浏览器兼容的情况下你也可以使用 prefetch。
- 例如服务端可以主动把JS和CSS文件推送给客户端，而不需要客户端解析HTML时再发送这些请求。
- 服务端可以主动推送，客户端也有权利选择是否接收。如果服务端推送的资源已经被浏览器缓存

过，浏览器可以通过发送RST_STREAM帧来拒收。主动推送也遵守同源策略，换句话说，服务器不能随便将第三方资源推送给客户端，而必须是经过双方确认才行。

http/3

虽然 HTTP/2 解决了很多之前旧版本的问题，但是它还是存在一个巨大的问题，主要是底层支撑的 TCP 协议造成的
HTTP/2 使用了多路复用，一般来说同一域名下只需要使用一个 TCP 连接。但当这个连接中出现了丢包的情况，那就会导致 HTTP/2 的表现情况反倒不如 HTTP/1 了
因为在出现丢包的情况下，整个 TCP 都要开始等待重传，也就导致了后面的所有数据都被阻塞了。但是对于 HTTP/1.1 来说，可以开启多个 TCP 连接，出现这种情况反到只会影响其中一个连接，剩余的 TCP 连接还可以正常传输数据
那么可能就会有人考虑到去修改 TCP 协议，其实这已经是一件不可能完成的任务了。因为 TCP 存在的时间实在太长，已经充斥在各种设备中，并且这个协议是由操作系统实现的，更新起来不大现实
基于这个原因，Google 就更起炉灶搞了一个基于 UDP 协议的 QUIC 协议，并且使用在了 HTTP/3 上，HTTP/3 之前名为 HTTP-over-QUIC，从这个名字中我们也可以发现，HTTP/3 最大的改造就是使用了 QUIC

谈谈你对 TCP 三次握手和四次挥手的理解

三次握手

- 你好我是客户端
- 收到我是服务端
- 确认完毕咱们连接吧

四次挥手

- 你好，我要结束了
- 稍等，还有数据没有传输完成
- 我已经将数据传输完成了，随时结束
- 你结束吧，不用回复

服务器和客户端建立连接， 突然服务器瘫痪了，客户端会出现什么情况

- 服务端不重启， 客户端继续工作，就会发现对方没有回应，如果路由器聪明的话，就会返回目的地不可达
- 服务器重启后，客户端继续关注，然后服务器就会丢失用户数据，收到客户端数据之后就会响应 rst

前端模块化开发

IIFE: 使用自执行函数来编写模块化，特点： 在一个单独的函数作用域中执行代码，避免变量冲突

```
;(function(){
    return { data: [] }
})();
```

AMD: 使用requireJS来编写模块化，特点：依赖必须提前声明好。

```
define('./index.js' , function(code){})
```

CMD: 使用seaJS来编写模块化，特点：支持动态引入依赖文件。

```
define(function(require, exports, module) {
  var indexCode = require('./index.js');
});
```

CommonJS： nodejs 中自带的模块化。

```
var fs = require('fs')
```

UMD：兼容AMD，CommonJS 模块化语法。
webpack(require.ensure)：webpack 2.x 版本中的代码分割

ES Modules： ES6 引入的模块化，支持import 来引入另一个 js 。

```
import a from 'a';
```

cookie和token 都放在header中，为什么会劫持cookie，不会劫持token

- cookie: 登陆后后端生成一个sessionid放在cookie中返回给客户端，并且服务端一直记录着这个sessionid，客户端以后每次请求都会带上这个sessionid，服务端通过这个sessionid来验证身份之类的操作。所以别人拿到了cookie拿到了sessionid后，就可以完全替代你

- token：登陆后后端不返回一个token给客户端，客户端将这个token存储起来，然后每次客户端请求都需要开发者手动将token放在header中带过去，服务端每次只需要对这个token进行验证就能使用token中的信息来进行下一步操作了。
- xss：用户通过各种方式将恶意代码注入到其他用户的页面中。就可以通过脚本获取信息，发起请求，之类的操作。
- csrf：跨站请求攻击，简单地说，是攻击者通过一些技术手段欺骗用户的浏览器去访问一个自己曾经认证过的网站并运行一些操作（如发邮件，发消息，甚至财产操作如转账和购买商品）。由于浏览器曾经认证过，所以被访问的网站会认为是真正的用户操作而去运行。这利用了web中用户身份验证的一个漏洞：简单的身份验证只能保证请求发自某个用户的浏览器，却不能保证请求本身是用户自愿发出的。csrf并不能够拿到用户的任何信息，它只是欺骗用户浏览器，让其以用户的名义进行操作
- csrf例子：假如一家银行用以运行转账操作的URL地址如下：

```
http://www.examplebank.com/withdraw?account=AccoutName&amount=1000&for=PayeeName
```

那么，一个恶意攻击者可以在另一个网站上放置如下代码：

```
<img src="<http://www.examplebank.com/withdraw?account=Alice&amount=1000&for=Badman>">
```

如果有账户名为Alice的用户访问了恶意站点，而她之前刚访问过银行不久，登录信息尚未过期，那么她就会损失1000资金。

上面的两种攻击方式，如果被xss攻击了，不管是token还是cookie，都能被拿到，所以对于xss攻击来说，cookie和token没有什么区别。但是对于csrf来说就有区别了。
以上面的csrf攻击为例：

- cookie：用户点击了链接，cookie未失效，导致发起请求后后端以为是用户正常操作，于是进行扣款操作。
- token：用户点击链接，由于浏览器不会自动带上token，所以即使发了请求，后端的token验证不会通过，所以不会进行扣款操作。

浏览器缓存机制

- Service Worker

  Service Worker 是运行在浏览器背后的独立线程，一般可以用来实现缓存功能。使用 Service Worker的话，传输协议必须为 HTTPS。因为 Service Worker 中涉及到请求拦截，所以必须使

用 HTTPS 协议来保障安全。Service Worker 的缓存与浏览器其他内建的缓存机制不同，它可以让我们自由控制缓存哪些文件、如何匹配缓存、如何读取缓存，并且缓存是持续性的

- Memory Cache

  Memory Cache 也就是内存中的缓存，主要包含的是当前中页面中已经抓取到的资源,例如页面上已经下载的样式、脚本、图片等。读取内存中的数据肯定比磁盘快,内存缓存虽然读取高效，可是缓存持续性很短，会随着进程的释放而释放。 一旦我们关闭 Tab 页面，内存中的缓存也就被释放了

- Disk Cache

  Disk Cache 也就是存储在硬盘中的缓存，读取速度慢点，但是什么都能存储到磁盘中，比之 Memory Cache 胜在容量和存储时效性上
  在所有浏览器缓存中，Disk Cache 覆盖面基本是最大的。它会根据 HTTP Herder 中的字段判断哪些资源需要缓存，哪些资源可以不请求直接使用，哪些资源已经过期需要重新请求。并且即使在跨站点的情况下，相同地址的资源一旦被硬盘缓存下来，就不会再次去请求数据。绝大部分的缓存都来自 Disk Cache，关于 HTTP 的协议头中的缓存字段，我们会在下文进行详细介绍

- Push Cache

  Push Cache（推送缓存）是 HTTP/2 中的内容，当以上三种缓存都没有命中时，它才会被使用。它只在会话（Session）中存在，一旦会话结束就被释放，并且缓存时间也很短暂，在Chrome浏览器中只有5分钟左右，同时它也并非严格执行HTTP头中的缓存指令

强缓存

强缓存：不会向服务器发送请求，直接从缓存中读取资源，在chrome控制台的Network选项中可以看到该请求返回200的状态码，并且Size显示from disk cache或from memory cache。强缓存可以通过设置两种 HTTP Header 实现：Expires 和 Cache-Control

- Expires
- Cache-Control

协商缓存

协商缓存就是强制缓存失效后，浏览器携带缓存标识向服务器发起请求，由服务器根据缓存标识决定是否使用缓存的过程

- Last-Modified和If-Modified-Since
- ETag和If-None-Match

为什么通常在发送数据埋点请求的时候使用的是 1x1 像素的透明 gif 图片

- 能够完成整个 HTTP 请求+响应（尽管不需要响应内容）
- 触发 GET 请求之后不需要获取和处理数据、服务器也不需要发送数据
- 避免跨域（img 天然支持跨域）
- 执行过程无阻塞
- 相比 XMLHttpRequest 对象发送 GET 请求，性能上更好
- GIF的最低合法体积最小（最小的BMP文件需要74个字节，PNG需要67个字节，而合法的GIF，只需要43个字节）

token加密

- 需要一个secret（随机数）
- 后端利用secret和加密算法(如：HMAC-SHA256)对payload(如账号密码)生成一个字符串(token)，返回前端
- 前端每次request在header中带上token
- 后端用同样的算法解密

es6 代码转es5 代码

- ES6 转 ES5 目前行业标配是用 Babel，转换的大致流程如下：
- 解析：解析代码字符串，生成 AST；
- 转换：按一定的规则转换、修改 AST；
- 生成：将修改后的 AST 转换成普通代码。

中间人攻击

- 服务器向客户端发送公钥。
- 攻击者截获公钥，保留在自己手上。
- 然后攻击者自己生成一个【伪造的】公钥，发给客户端。
- 客户端收到伪造的公钥后，生成加密hash值发给服务器。
- 攻击者获得加密hash值，用自己的私钥解密获得真秘钥。
- 同时生成假的加密hash值，发给服务器。
- 服务器用私钥解密获得假秘钥。
- 服务器用加秘钥加密传输信息

前端加密

- PlanA
- 使用 Base64 / Unicode+1 等方式加密成非明文，后端解开之后再存它的 MD5/MD6 。
- PlanB

- 直接使用 MD5/MD6 之类的方式取 Hash ， 让后端存 Hash 的 Hash 。

https页面接收http请求

```
<meta http-equiv="Content-Security-Policy" content="upgrade-insecure-requests">

// 浏览器不兼容在后台设置
header("Content-Security-Policy: upgrade-insecure-requests");
```

握手阶段分成五步。

> 第一步，爱丽丝给出协议版本号、一个客户端生成的随机数（Client random），以及客户端支持的加密方法。
>
> 第二步，鲍勃确认双方使用的加密方法，并给出数字证书、以及一个服务器生成的随机数（Server random）。
>
> 第三步，爱丽丝确认数字证书有效，然后生成一个新的随机数（Premaster secret），并使用数字证书中的公钥，加密这个随机数，发给鲍勃。
>
> 第四步，鲍勃使用自己的私钥，获取爱丽丝发来的随机数（即Premaster secret）。
>
> 第五步，爱丽丝和鲍勃根据约定的加密方法，使用前面的三个随机数，生成"对话密钥"（session key），用来加密接下来的整个对话过程。

数据结构 设计模型

数据结构

是互相之间存在一种或多种特定关系的数据元素的集合

1. 数据 是描述客观事物的符号，是计算机中可以操作的对象，是能被计算机识别，并输入给计算机处理的符号集合

   程序设计 = 数据结构 + 算法

算法

解决特定问题求解步骤的描述，在计算机中表现为指令的有限序列，并且每条指令表示一个或多个操作

时间复杂度

执行次数	阶	非正式术语
12	O(1)	常数阶
2n+3	O(n)	线性阶
$3n^2+2n+1$	$O(n^2)$	平方阶
2n次方	O(n次方)	指数

数组

就是相同数据类型（可以是基本类型也可以是自定义对象）的元素按一定顺序排列的集合，它们在内存中按照这个先后顺序连续存放在一起

链表

链表是继数组之后第二种最通用的数据结构，是一种物理存储单元上非连续、非顺序的存储结构，数据元素的逻辑顺序是通过链表中的指针链接次序实现的

线性表

零个或多个数据元素的有限序列

栈：由系统自动分配，速度较快。但程序员是无法控制的。

堆：是由new分配的内存，一般速度比较慢，而且容易产生内存碎片不过用起来最方便。

观察者模式和订阅-发布模式的区别，各自适用于什么场景

观察者设计模式
在软件设计中是一个对象，维护一个依赖列表，当任何转态发生改变自动通知它们。

```
// 创建一个观察者模式
class Observers{
  // 消息队列 用来存储 消息
  observers = []
  // 消息分发 ，当有需要的时候进行消息分发
  notify(){
      this.observers.forEach( observer => observer.update() )
  }
// 消息添加
  attach(observer){
    this.observers.push(observer)
  }
}

// 消息
let observer = {
  update(){
    console.log("通知我")
  }
}

let observers = new Observers;
// 添加新的消息
observers.attach(observer);
// 分发消息
observers.notify()
```

发布-订阅模式

- 在发布-订阅模式，消息的发送方，叫做发布者（publishers），消息不会直接发送给特定的接收者，叫做订阅者。
- 发布者和订阅者不知道对方的存在。需要一个第三方组件，叫做信息中介，它将订阅者和发布者串联起来，它过滤和分配所有输入的消息。换句话说，发布-订阅模式用来处理不同系统组件的信息交流，即使这些组件不知道对方的存在。

```
// 订阅者模式
class Dep {
  // 创建订阅者调用栈
    subs = []
  // 将消息添加到调用栈中
  addSub(sub){
    this.subs.push(sub)
  }
  // 调用订阅者更新方法
  notify(){
    this.subs.forEach( sub => sub.update() )
  }
}
const dep = new Dep;
// 发布者
var publisher = {
  publish(sub) {
    sub.notify()
  }
}

// 用户
var subscribe = {
  update() {
    console.log('update')
  },
  subscribe(sub) {
    sub.addSub(this);
  }
}
```

```
publisher.publish(dep)
subscribe.subscribe(dep)
```

两种模式本质都是一样的，主要关键点都在于注册（添加到注册数组中）和触发（触发注册数组中的内容），只是订阅/发布模式对注册和触发进行了解耦。可以看到，使用订阅发布模式中发布者触发publish的时候，可以选择触发哪一些订阅者集合（因为publish参数传递了中间集合，可以定义多个pubsub集合），而观察者模式则只能触发所有的被观察对象

公司问题

1. 在未来6个月时间里，您觉得公司最大的挑战是什么?
2. 您计划如何克服这些挑战?
3. 过去6个月时间里公司克服的最大的挑战是什么?
4. 你如何衡量成功?
5. 公司如何衡量成功?
6. 公司有没有什么系统/机制用来确保
7. 每个人知道公司的前进目标?
8. 有什么问题或者疑虑可以提出来得到合理和通彻的回复?
9. 公司人员分配
10. 你们部门有几个人
11. 你为什么离职
12. 你住在哪里
13. 你的薪资体系
14. 之前你的薪资多少
15. 为什么来上海发展
16. 之前公司的项目管理，代码管理，自动构建工具